## Contents

### PRELIMINARIES
1. A High-Level Tour of All Things Algorithmic  
2. Useful Mathematical Concepts  
3. Sorting and Algorithm Analysis  

### DATA STRUCTURES
4. Amortized Analysis  
5. Priority Queues  
6. Sequences and Comparison-Based Search Structures  
7. Hashing and Integer Search Structures  
8. Range Queries  
9. String and Document Searching  

### OPTIMIZATION
10. Greedy Algorithms  
11. Dynamic Programming  
12. Integer and Linear Programming  
13. Heuristics and Exhaustive Search
## Detailed Contents

### PRELIMINARIES

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A High-Level Tour of All Things Algorithmic</td>
<td>3</td>
</tr>
<tr>
<td>1.1</td>
<td>A Bit of History</td>
<td>4</td>
</tr>
<tr>
<td>1.1.1</td>
<td>Decidability: Hilbert, Gödel, Church, and Turing</td>
<td>4</td>
</tr>
<tr>
<td>1.1.2</td>
<td>Turing Machines and the Church-Turing Thesis</td>
<td>5</td>
</tr>
<tr>
<td>1.1.3</td>
<td>From Computability to Complexity Theory</td>
<td>6</td>
</tr>
<tr>
<td>1.2</td>
<td>Models of Computation</td>
<td>7</td>
</tr>
<tr>
<td>1.2.1</td>
<td>The RAM Model of Computation</td>
<td>8</td>
</tr>
<tr>
<td>1.2.2</td>
<td>The Real RAM</td>
<td>9</td>
</tr>
<tr>
<td>1.2.3</td>
<td>Comparison-Based Algorithms</td>
<td>10</td>
</tr>
<tr>
<td>1.2.4</td>
<td>Other Models</td>
<td>10</td>
</tr>
<tr>
<td>1.3</td>
<td>How to Describe an Algorithm</td>
<td>11</td>
</tr>
<tr>
<td>1.4</td>
<td>Characterizing Algorithm Performance</td>
<td>13</td>
</tr>
<tr>
<td>1.4.1</td>
<td>Empirical Testing</td>
<td>13</td>
</tr>
<tr>
<td>1.4.2</td>
<td>Mathematical Analysis Using Asymptotic Notation</td>
<td>15</td>
</tr>
<tr>
<td>1.4.3</td>
<td>Worst-Case and Average-Case Behavior</td>
<td>15</td>
</tr>
<tr>
<td>1.4.4</td>
<td>Common Running Times</td>
<td>16</td>
</tr>
<tr>
<td>1.4.5</td>
<td>Output Sensitivity</td>
<td>18</td>
</tr>
<tr>
<td>1.4.6</td>
<td>Input Sensitivity</td>
<td>18</td>
</tr>
<tr>
<td>1.4.7</td>
<td>Lower Bounds and Optimal Running Times</td>
<td>19</td>
</tr>
<tr>
<td>1.5</td>
<td>Data Structures</td>
<td>20</td>
</tr>
<tr>
<td>1.5.1</td>
<td>Arrays Versus Linked Lists</td>
<td>21</td>
</tr>
<tr>
<td>1.5.2</td>
<td>Stacks and Queues</td>
<td>23</td>
</tr>
<tr>
<td>1.6</td>
<td>Complexity Theory</td>
<td>24</td>
</tr>
<tr>
<td>1.6.1</td>
<td>The Complexity Classes P and NP</td>
<td>25</td>
</tr>
<tr>
<td>1.6.2</td>
<td>NP-Hardness and the Satisfiability Problem</td>
<td>26</td>
</tr>
<tr>
<td>1.6.3</td>
<td>Proving NP-Hardness via Reductions</td>
<td>27</td>
</tr>
<tr>
<td>1.7</td>
<td>Different Flavors of Algorithms</td>
<td>28</td>
</tr>
<tr>
<td>1.7.1</td>
<td>Randomized Algorithms</td>
<td>29</td>
</tr>
<tr>
<td>1.7.2</td>
<td>Approximation Algorithms</td>
<td>30</td>
</tr>
<tr>
<td>1.7.3</td>
<td>Online Algorithms</td>
<td>32</td>
</tr>
<tr>
<td>1.7.4</td>
<td>Parallel and Distributed Algorithms</td>
<td>33</td>
</tr>
<tr>
<td>1.7.5</td>
<td>Memory-Constrained Algorithms</td>
<td>35</td>
</tr>
<tr>
<td>1.8</td>
<td>Closing Remarks</td>
<td>36</td>
</tr>
</tbody>
</table>
2 Useful Mathematical Concepts

2.1 Notation, Definitions, and Assumptions

2.1.1 Floors, Ceilings, and Other Pesky Details

2.1.2 Asymptotic Notation

2.1.3 Special Functions

2.1.4 Graphs and Trees

2.2 Doing the Math

2.3 Recurrences

2.3.1 “Divide and Conquer” Recurrences

2.3.2 Linear Recurrences

2.4 Probability

2.4.1 The Basics

2.4.2 “With High Probability” Results

2.4.3 Random Variables and Linearity of Expectation

2.4.4 Tail Inequalities

2.5 Linear Algebra

2.6 Other Useful Mathematical Topics

3 Sorting and Algorithm Analysis

3.1 Algorithm Design Techniques

3.1.1 Iterative Refinement: Bubble Sort

3.1.2 Incremental Construction: Insertion Sort

3.1.3 Divide and Conquer: Merge Sort

3.2 Arguing Correctness and Running Time

3.2.1 Correctness Proofs Based on Induction

3.2.2 Different Ways of Adding up the Running Time

3.2.3 Inversions and Potential Arguments

3.3 Lower Bounds

3.3.1 Modeling Algorithms by Decision Trees

3.3.2 Randomized and Algebraic Decision Trees

3.3.3 Lower Bounds via Reductions

3.4 Quicksort

3.5 In-Place Sorting

3.6 Stable Sorting

3.7 Selection

3.8 Topological Sorting

3.9 Sorting Integers

3.9.1 Counting Sort

3.9.2 Radix Sort

3.10 Advanced Example: Parametric Search

3.11 Closing Remarks and Additional Problems

4 Amortized Analysis

4.1 Example: Buffering Writes to Disk

4.2 Methods for Amortized Analysis

4.2.1 Accounting Tricks with Running Time
4.2.2 Potential Functions ........................................ 118
4.3 Example: The Min-Queue ..................................... 119
4.4 Example: Dynamic Memory Allocation ...................... 122
4.5 Example: Functional Queues .................................. 125
4.6 Example: Disjoint Sets ....................................... 127
  4.6.1 List-Based Implementations .............................. 127
  4.6.2 Tree-Based Implementations ............................ 129
4.7 Example: Persistent Data Structures ......................... 130

5 Priority Queues ................................................. 133
  5.1 Priority Queues and Sorting ................................. 134
  5.2 Unsorted and Sorted Arrays ................................. 135
  5.3 Binary Heaps ................................................ 136
     5.3.1 Storing a Binary Heap in an Array .................... 136
     5.3.2 Operations on a Binary Heap ......................... 137
     5.3.3 Heap Sort ............................................ 138
  5.4 Mergeable Priority Queues ................................ 139
     5.4.1 Randomized Mergeable Binary Heaps ................... 139
     5.4.2 Leftist Heaps, Skew Heaps, and Other Relatives Based on Null Path Merging ......................... 140
     5.4.3 Binomial Heaps ....................................... 143
     5.4.4 Fibonacci Heaps ..................................... 145
  5.5 Integer Priority Queues ..................................... 147
     5.5.1 The Radix Tree ...................................... 148
     5.5.2 Multilevel Buckets: Another Route to the Radix Tree .......................... 148
     5.5.3 Monotone Integer Priority Queues and the Radix Heap .......................... 149
  5.6 Additional Problems ........................................ 150

6 Sequences and Comparison-Based Search Structures ............ 155
  6.1 The Binary Search Tree .................................... 157
     6.1.1 Traversals and Sorting ................................ 158
     6.1.2 Min and Max, and the BST as a Priority Queue ........ 160
     6.1.3 Predecessor, Successor, and Inexact Search ............ 160
     6.1.4 Augmenting a BST, Select, and Rank .................... 162
     6.1.5 Encoding a Sequence in a BST .......................... 163
     6.1.6 Rotations ............................................ 164
     6.1.7 Split and Join ....................................... 165
     6.1.8 Maintaining Augmented Data ........................... 166
  6.2 Balanced Binary Search Trees ............................... 166
     6.2.1 Height-Balanced (AVL) Trees ........................... 167
     6.2.2 Balancing Based on Subtree Size ....................... 167
     6.2.3 Red-Black and Path-Balanced Trees ..................... 167
     6.2.4 Randomly-Structured Binary Search Trees ................ 168
     6.2.5 Treaps ............................................... 169
     6.2.6 Amortized Rebalancing and Scapegoat Trees ............ 171
     6.2.7 Splay Trees ......................................... 173
  6.3 B-Trees ...................................................... 175
  6.4 Skip Lists ................................................... 177
## 7 Hashing and Integer Search Structures

### 7.1 Hash Tables
- Collision Resolution
- Guidelines for Designing a Good Hash Function
- Choosing the Size of a Hash Table

### 7.2 Universal Hashing
- Acing Your Next Job Interview
- Further Examples of Universal Hash Functions
- Higher Degrees of Independence

### 7.3 Collision-Free (Perfect) Hashing
- Dynamic Perfect Hashing
- Cuckoo Hashing

### 7.4 Radix Trees
- Reducing Space: Path Contraction and Indirection
- The Y-Fast Tree
- Stratified / van Emde Boas (vEB) Trees

### 7.5 Further Hashing Applications
- Pseudorandom Number Generation
- Load Balancing and Consistent Hashing
- Distributed Hash Tables
- Problem Decomposition, MapReduce
- Fingerprinting Large Objects
- Security and Data Integrity
- Bloom Filters and Sketching
- Compressive Sensing and its Relatives
- Locality-Sensitive Hashing
- High-Dimensional Nearest Neighbor Search

## 8 Range Queries

### 8.1 One-Dimensional Range Queries
- Static Range Queries and Prefix Sums
- Trees Atop Arrays
- Binary Indexed Trees

### 8.2 Stabbing Queries and Interval Trees

### 8.3 Range Minimum Queries
- Lowest Common Ancestors
- Cartesian Trees
- Range Minimum Queries in Trees
- Separator-Based Tree Decomposition

### 8.4 Multi-Dimensional Range Queries
- BSP Trees, k-d-Trees, Quadtrees, and Recursive Spatial Decomposition
- Priority Search Trees
- Range Trees

### 8.5 Dynamic Trees
- Heavy-Light Decomposition
- Euler Tour Trees
- Node-Disjoint Dynamic Paths
9 String and Document Searching

9.1 Approaches Based on Hashing 244
  9.1.1 Hashing Substrings 244
  9.1.2 Inverted Indices 245
9.2 Building a String Matching Automaton 246
  9.2.1 Regular Expression Matching 248
  9.2.2 Beyond Regular Expressions 249
9.3 Suffix Arrays 249
  9.3.1 Building a Suffix Array 250
  9.3.2 The Burrows-Wheeler Transform 251
9.4 Tries and Suffix Trees 252
  9.4.1 Converting Between Suffix Arrays and Suffix Trees 255
9.5 Extensions and Advanced Topics 256
  9.5.1 Multiple Patterns 256
  9.5.2 Multiple Texts 256
  9.5.3 Approximate String Matching 257
  9.5.4 Spell Checking 258
  9.5.5 Predictive Completion 259
  9.5.6 Text Compression 259
9.6 Additional Problems 259

OPTIMIZATION

10 Greedy Algorithms 263

10.1 Example: Scheduling on a Single Machine 264
  10.1.1 Proof by Contradiction: The Exchange Argument 265
  10.1.2 Proof by Induction 266
  10.1.3 Greedy Methods and Online Algorithms 268
10.2 Example: Compression with Huffman Codes 269
  10.2.1 Brief Aside: Entropy Bounds and Arithmetic Coding 270
10.3 Example: Minimum Spanning Trees 272
  10.3.1 Kruskal’s Algorithm 272
  10.3.2 Brief Aside: Matroids 273
10.4 Example: Matching Problems 274
  10.4.1 Matchings in Graphs 275
  10.4.2 Greedy Approximation Algorithms 276
10.5 Example: Packing and Covering Problems 276
  10.5.1 Brief Aside: Duality 278
  10.5.2 General Covering and Packing Problems 279
10.6 Additional Problems 281

11 Dynamic Programming 287

11.1 Example: Shortest Paths in Acyclic Graphs 289
  11.1.1 Building up Optimal Solutions to Subproblems 289
  11.1.2 Dynamic Programs as Computations on DAGs 292
  11.1.3 Top-Down Versus Bottom-Up Computation 294
11.2 Example: The Knapsack Problem 297
11.3 Example: Optimal Binary Search Trees 301
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.4</td>
<td>DP and Approximation Algorithms</td>
<td>304</td>
</tr>
<tr>
<td>11.5</td>
<td>Advanced Tricks</td>
<td>306</td>
</tr>
<tr>
<td>11.5.1</td>
<td>Reducing Space Usage</td>
<td>306</td>
</tr>
<tr>
<td>11.5.2</td>
<td>Speedups via Quadrangle/Monge Inequalities</td>
<td>307</td>
</tr>
<tr>
<td>11.5.3</td>
<td>Speedups via Monotone Matrix Searching</td>
<td>307</td>
</tr>
<tr>
<td>11.6</td>
<td>Additional Problems</td>
<td>308</td>
</tr>
<tr>
<td>12</td>
<td>Integer and Linear Programming</td>
<td>311</td>
</tr>
<tr>
<td>12.1</td>
<td>Linear Programming</td>
<td>313</td>
</tr>
<tr>
<td>12.1.1</td>
<td>Formulating Problems as Linear Programs</td>
<td>314</td>
</tr>
<tr>
<td>12.1.2</td>
<td>The Geometry of Linear Programs</td>
<td>316</td>
</tr>
<tr>
<td>12.1.3</td>
<td>Basic Solutions</td>
<td>317</td>
</tr>
<tr>
<td>12.2</td>
<td>Solving Linear Programs</td>
<td>318</td>
</tr>
<tr>
<td>12.2.1</td>
<td>The Simplex Algorithm</td>
<td>318</td>
</tr>
<tr>
<td>12.2.2</td>
<td>The Ellipsoid Algorithm</td>
<td>320</td>
</tr>
<tr>
<td>12.2.3</td>
<td>Interior Point Methods</td>
<td>320</td>
</tr>
<tr>
<td>12.3</td>
<td>Integrality</td>
<td>321</td>
</tr>
<tr>
<td>12.4</td>
<td>Relaxation and Rounding</td>
<td>324</td>
</tr>
<tr>
<td>12.5</td>
<td>Linear Programming Duality</td>
<td>328</td>
</tr>
<tr>
<td>12.5.1</td>
<td>Complementary Slackness</td>
<td>330</td>
</tr>
<tr>
<td>12.5.2</td>
<td>The Dual Simplex Algorithm</td>
<td>330</td>
</tr>
<tr>
<td>12.5.3</td>
<td>Primal-Dual Algorithms</td>
<td>331</td>
</tr>
<tr>
<td>12.5.4</td>
<td>Connections with Game Theory</td>
<td>332</td>
</tr>
<tr>
<td>13</td>
<td>Heuristics and Exhaustive Search</td>
<td>335</td>
</tr>
<tr>
<td>13.1</td>
<td>Iterative Refinement of a Single Solution</td>
<td>336</td>
</tr>
<tr>
<td>13.1.1</td>
<td>Local Search</td>
<td>336</td>
</tr>
<tr>
<td>13.1.2</td>
<td>Simulated Annealing</td>
<td>339</td>
</tr>
<tr>
<td>13.1.3</td>
<td>Alternating Iterative Refinement</td>
<td>340</td>
</tr>
<tr>
<td>13.1.4</td>
<td>Multiscale Methods</td>
<td>343</td>
</tr>
<tr>
<td>13.2</td>
<td>Refining a Population of Solutions</td>
<td>345</td>
</tr>
<tr>
<td>13.2.1</td>
<td>Combining Solutions</td>
<td>346</td>
</tr>
<tr>
<td>13.2.2</td>
<td>Genetic Algorithms and Other Bio-Inspired Methods</td>
<td>347</td>
</tr>
<tr>
<td>13.3</td>
<td>Branch and Bound</td>
<td>348</td>
</tr>
<tr>
<td>13.3.1</td>
<td>Dynamic Programming and A* Search</td>
<td>352</td>
</tr>
<tr>
<td>13.3.2</td>
<td>Alpha-Beta Search</td>
<td>352</td>
</tr>
<tr>
<td>13.3.3</td>
<td>Breadth-First Search and Iterative Deepening</td>
<td>352</td>
</tr>
<tr>
<td>13.3.4</td>
<td>Adding Constraints with Cutting Planes</td>
<td>353</td>
</tr>
<tr>
<td>13.4</td>
<td>Constraint Satisfaction</td>
<td>354</td>
</tr>
</tbody>
</table>
Preliminaries
1. A High-Level Tour of All Things Algorithmic

The field of algorithmic computer science, sometimes also called *algorithmics* or *informatics*, has led to profound and transformative advances in almost every area of science and industry over the past decades. As a consequence, proficiency in algorithms and computational problem-solving has become a crucial skill for students and professionals in almost every corner of today’s data driven world, not only those with narrow focus in computer science itself.

This book provides a comprehensive introduction to the study of algorithms. We begin in this chapter with a high-level survey of some of the history, foundations, applications, fundamental concepts, and main topic areas generally associated with the study of algorithms. This chapter and the next two make up the “preliminaries” section of the book, where we introduce basic terminology and techniques. A review of relevant mathematics appears in Chapter 2, and in Chapter 3 we discuss the fundamental topic of sorting as well as basic techniques for algorithm design and analysis.

Before we get too far, however, it is worthwhile to ask the question “what is an algorithm?”. An algorithm is a precisely-characterized procedure for solving a computational problem. Informally, it is a computational “recipe”. A good algorithm is simple to understand and implement, it makes efficient use of computational resources (e.g., time, memory, processors, network bandwidth, energy) and it provides an exact or suitably high-quality solution to its intended problem.

To give a quick example, suppose we wish to find the definition of “informatics” in an alphabetically-ordered $n$-word dictionary. A correct but slow algorithm for this task is a *linear search*: examine every page sequentially from the beginning of the dictionary until we find the target word. However, a far better algorithm is a *binary search*: open the dictionary to its middle word — in the author’s dictionary that middle word is “janitor”. Since “informatics” precedes “janitor”, we can rule out the entire second half of the dictionary and repeat the process on just the first half. Every step halves the total number of words under consideration, so we make much faster progress toward our solution, reaching the target word in at most $\log_2 n$ steps. In a dictionary with $n = 1$ million words, linear search might examine every single word, while binary search will never look at more than 20. The difference
is quite dramatic. For many problems in practice, algorithmic improvements can easily reduce computation time from years to seconds.

An algorithm is different from a computer program, and the study of algorithms is different from the study of computer programming, even though the two are often taught together. Algorithms are abstract problem-solving procedures that can be realized as computer programs, but you can certainly study algorithms without writing computer programs. Likewise, you can study aspects of programming (e.g., syntax and semantics of programming languages) that do not particularly involve algorithms. The two topics are closely related, however, since algorithms play an important role in most computer programs, and proficiency in computer programming generally provides a useful mental framework that helps in learning algorithms.

1.1 A Bit of History

Although algorithmic computer science is mostly a young field, its historical roots go back quite far. In the next few pages, we outline some of the most important historical developments that helped to shape the field.

The study of algorithms has certainly been driven by technological innovation, with the widespread use of powerful computing devices of all shapes and sizes. However, algorithms were perhaps even more crucial before the advent of the modern computer, when solving problems required painstaking manual calculation. For example, in order to solve a large system of linear equations to determine the orbit of the asteroid Pallas, the great mathematician Carl Friedrich Gauss developed in 1810 the algorithm we now know as Gaussian elimination. Many classical algorithms for performing numerical computation have similar origins.

The word algorithm itself comes from the name of a renown 9th century Persian mathematician Abu Ja’far Muhammad ibn Musa Al-Khwarizmi (one of his books, entitled “al-Mukhtasar fi Hisab al-Jabr wa l-Muqabala” is the origin of the term algebra as well). In his written work, Al-Khwarizmi describes arithmetic procedures for operating on numbers written in base 10. That his work was later translated and spread in influence across Europe is part of the reason the base-10 system is known as the system of “Arabic” numerals, even though it was initially developed in India.

1.1.1 Decidability: Hilbert, Gödel, Church, and Turing

The work of David Hilbert, one of the most prominent mathematicians of the early 20th century, helped shape some of our earliest thoughts on the theory of computation. In 1928, Hilbert posed two famous questions (with some technical details omitted for simplicity of discussion):

- Can one derive all of mathematics from a small set of fundamental axioms that is both consistent and complete? Consistent means that no mathematical statement can be proved to be both true and false, and complete means that it is possible to construct a proof of the truth or falseness of every statement.
1.1. A BIT OF HISTORY

For every mathematical statement, does there exist an algorithm that can determine its truth or falseness?

In 1931 Kurt Gödel surprised the mathematical world by resolving the first question negatively, demonstrating that for any consistent set of axioms, there must exist statements that cannot be proved or disproved. We include a short sketch of Gödel’s famous proof for the interested reader. It employs a clever technique called diagonalization, which shows that there are in some sense more mathematical statements than proofs (even though there are infinite numbers of both!), so there must exist some statement for which a corresponding proof does not exist.

Shortly after Gödel’s result, Alan Turing and Alonzo Church both independently managed to resolve the second question also in the negative, providing examples of undecidable problems that cannot be solved by any algorithm, irrespective of the amount of time the algorithm is allowed to run. We now know of several natural undecidable problems, the most famous of them probably being the halting problem, which asks us to predict whether a given algorithm will terminate or run forever (in an endless loop) on a given input. The results of Church and Turing proved that there are fundamental theoretical limitations to the power of algorithms. These limitations rarely get in the way, however, since the vast majority of the problems we encounter in practice are quite clearly decidable (solvable by algorithms). The more troublesome limitation is that many practical problems seem not “efficiently” solvable by algorithms. [Short proof that the halting problem is undecidable]

1.1.2 Turing Machines and the Church-Turing Thesis

What is a computer? One of the main contributions of the work of Church and Turing was to formally characterize the notions of “computation” and “algorithms” in precise mathematical terms — a crucial prerequisite before we can prove rigorous theorems about these concepts, such as the existence of undecidable problems.

Turing characterized an algorithm in terms of a simple abstract computing machine now known as a Turing machine, shown in Figure 1.1. It contains a memory in the form of an infinite one-dimensional binary tape, as well as a processing unit that interacts with the tape via a read/write head. The processing unit is a finite state machine: at every time step it looks at its current state and the binary digit
currently under the read/write head, and based on these it transitions to a new internal state, writes a new binary digit onto the tape (if desired), and shifts the tape left or right by one position (if desired). The tape is used by the Turing machine to read its input, to store the results of intermediate calculations, and to write its output. The machine starts in a designated “start state” and terminates once it reaches a designated “stop state”.

Turing’s bold claim was that this simplistic machine is “universal” in its ability to model any conceivable algorithm. Church made a similar claim based on an alternative model of computation known as the lambda calculus, which was soon shown to be equivalent to the Turing machine. The combined result of the work of Church and Turing is known as the Church-Turing thesis, and it asserts that every algorithm in the world can be represented by a Turing machine. As a consequence, one cannot get around the undecidability of problems like the halting problem by just building a fancy new type of computer.

There is no Nobel prize in computing, but in recognition of Turing’s contributions, the Turing award is now given each year to the top researchers in computer science.

1.1.3 From Computability to Complexity Theory

The results of Church and Turing in the 1930s were fundamentally important in the area of computability — deciding what problems can and cannot be solved by algorithms of different types. The next big step in the development of the theory of computation was the study of complexity theory, which addresses how efficiently certain problems can be solved by algorithms.

There are often only seemingly minor differences between problems that are easy to solve and those that are vastly more difficult. As an example, consider the famous “Bridges of Königsberg” story, a favorite among mathematicians: In the early 18th century, Königsberg was a city in East Prussia with seven bridges spanning the rivers crossing through the city center, shown in Figure 1.2(a). As the story goes, the residents of Königsberg challenged themselves to find a path that crossed every bridge exactly once. The prolific mathematician Leonard Euler cleverly resolved this question (negatively) in 1735 by providing one of the original results in the area of graph theory. A graph is a collection of nodes and edges, where every edge connects a pair of nodes. As shown in Figure 1.2(b), Euler modeled the Königsberg bridges abstractly in terms of a graph where the nodes represent regions of the city and edges represent bridges connecting these regions. We now refer to a path through a graph that visits every edge exactly once an Eulerian path, since Euler was the first to characterize the precise mathematical conditions required for such a path to exist: the graph must be connected, and all but at most two nodes must have an even number of incident edges. [Simple proof]

It turns out that Eulerian paths are “easy” to compute. It is fairly simple to transform the constructive proof above into an algorithm that finds such a path (if it exists) in time proportional to the size of a graph; we will show how when we study graphs in Chapter ???. Even for a billion-edge graph, a modern computer can therefore easily find an Eulerian path in a matter of seconds.

On the other hand, consider the similar problem of finding a Hamiltonian path —
a path that visits every node exactly once, named after the 19th century mathematician William Hamilton. Although Hamiltonian paths and Eulerian paths may not appear that different, it turns out that the Hamiltonian path problem is far more difficult to solve efficiently. For an $n$-node graph, all known Hamiltonian path algorithms require time at least exponential in $n$. Even for small values like $n = 50$, this can take years even on the fastest of modern computers!

In the early 1970s, Richard Karp, Stephen Cook, and Leonid Levin dramatically advanced the field of complexity theory by developing the theory of $NP$-completeness. It allows us to group thousands of hard problems into a single complexity class (known as the $NP$-complete problems) by proving they all have in some sense “equivalent” hardness. An efficient solution for just one of the NP-complete problems would imply, via appropriate transformations, the existence of an efficient solution for all of the NP-complete problems; we discuss how this works in greater detail later in this chapter. Decades of research have yielded not a single efficient solution to any NP-complete problem, so we strongly suspect that these problems are not efficiently solvable, although nobody has managed to prove this. If someone asks you to produce an efficient algorithm for determining whether a graph has a Hamiltonian path (a problem known to be NP-complete), you can therefore decline to do so in a graceful manner, by pointing out that doing so would resolve what is perhaps the biggest open problem in the entire field of computer science.

The field of cryptography, now extremely important in practice due to the need for secure digital commerce, has its roots in complexity theory, since most algorithms for encrypting data crucially depend on the existence of hard problems so that decryption in the absence of the proper “key” is computationally infeasible.

### 1.2 Models of Computation

A mathematical proof only makes sense in the context of a certain system of fundamental axioms telling us the underlying assumptions we can make. Similarly, the running time of an algorithm (the number of operations it performs) only makes sense within the context of a specific model of computation that defines the primitive operations available in the abstract computing environment in which the algorithm
executes.

Often, one algorithm will appear “better” than another simply because it has access to a wider range of operations in a more powerful model of computation. To illustrate this point with a ridiculous example, we can solve the NP-complete Hamiltonian path problem in a single time step if we define a (completely unrealistic) abstract computing environment supporting a “compute Hamiltonian path” instruction. It is to our advantage to adopt a computational model that reflects the capabilities and limitations of an actual modern computer. This not only enables straightforward implementation of our algorithms on an actual computer, but it also makes their running time analyses much more realistic.

We have already seen one model of computation so far in this chapter: the Turing machine. An algorithm in this model uses operations like “shift the tape left/right”, “write a zero/one to the current location on the tape”, or “test whether the value in the current location on the tape is a zero or one”. Unfortunately, even a simple task like adding two binary numbers requires a painfully complex sequence of these very low-level operations. The Turing machine is a poor approximation of a modern computer, which can do things like adding two numbers in a single operation.

### 1.2.1 The RAM Model of Computation

In this book, we adopt the simple and widely-used computational model known as the random access machine (RAM), which roughly approximates a modern digital computer with a single processor. Data in the RAM model is processed at the granularity of fixed-length binary numbers known as words; modern computers typically have word sizes of 32 or 64 bits. We can perform simple arithmetic operations like addition, subtraction, multiplication, integer division, remainder, and comparison in a single step\(^1\) on two words. There is a memory consisting of a long array of words, and we can store and retrieve words in a single step in a “random access” fashion. That is, we can access any word in memory directly in a single step via its numeric address, or index, within the memory.

The only slightly murky issue about the RAM model is the size of word. It is problematic to assume that words can be arbitrarily large, since this gives the model too much power — it could do unrealistic things like adding together two infinite-digit numbers in a single step. On the other hand, it is also problematic to assume that words contain at most some fixed constant number of bits (e.g., 64 bits), independent of the size of the problem instance we are solving. An algorithm receiving \(n\) words of data as input needs the ability to count as high as \(n\), or else it wouldn’t be able to index into the input in memory. Hence, word size must be at least \(\log_2 n\) bits, since this is necessary to represent a number of size \(n\). A commonly assumed word size is \(k \log n\) bits, where \(n\) is the size of the input and \(k\) is some constant, although larger word sizes are sometimes considered for certain problems (e.g., sorting integers). In general, it is usually fine to ignore the issue of word size as long as we are not trying to do unreasonable things, like storing and operating

---

\(^1\)This is called the “unit cost” RAM, since arithmetic operations on whole words are assumed to take a single step. One also finds a “logarithmic cost” RAM in which some arithmetic operations on integers of size at most \(C\) take time proportional to \(\log C\). The logarithmic cost model makes sense if you look at the size of the physical circuits required to implement these operations.
on unrealistically large numbers. We will try to warn the reader when this is not
the case.

Although the RAM is a reasonable model for a single-processor digital computer,
some of its assumptions are overly simplistic. For example, the RAM assumes that
all operations take the same amount of time, while in reality certain operations
like multiplication may take longer to perform than simpler operations like addi-
tion. Also, memory accesses usually take substantially more time than primitive
arithmetic operations, and as a result of caching, multiple memory accesses close
to each other take much less time than those spread out haphazardly throughout
memory; we will elaborate on this point later in the chapter.

1.2.2 The Real RAM

One of the limitations of the RAM model is that it can only operate on integers
(and also rational numbers, being ratios of integers). However, irrational numbers
also arise in a variety of common problems, such as geometry problems involving
distances in the 2D plane. Irrational numbers have no convenient representation in
the RAM model, and this might seem perfectly reasonable since irrational numbers
have no convenient representation on a digital computer. However, in order to
simplify the algorithm design process, it is sometimes helpful to pretend that our
model of computation can deal with irrational numbers. This gives a model known
as the real RAM, which is the same as the standard RAM except it can store and
operate on real numbers as well as integers. That is, words in memory can be
designated as either integers or reals, or alternatively you can regard the model as
having two separate memories, one for integers and the other for reals.²

Many algorithms fundamentally require and exploit integrality in their input data,
using tricks such as lookup tables and hashing (discussed further in Chapter 7)
that use input elements as array indices or that require the ability to perform
integer division.³ Other algorithms make no such integrality assumptions, working
equally well if they are fed integers as if they are fed real numbers. The distinction
between these two types of algorithms is important to keep in mind. For lack of
better terminology, we will often call them “RAM algorithms” versus “real RAM
algorithms”, although the way we have defined the real RAM as a superset of the
RAM, it is certainly possible to run a RAM algorithm on the real RAM as long
as we are careful to designate its input as integral (there would be little point to
doing this, however, since we aren’t using any of the features of the real RAM).
RAM algorithms often run faster than their real RAM counterparts, since they can
exploit the integrality of their input.

²Distinguishing between real versus integer words in memory is important for a subtle reason:
we typically do not allow the real RAM to truncate a real number into an integer, since computation
of \([x]\) and \([x]\) for a real number \(x\) makes the model unrealistically powerful, just like a standard
RAM with unbounded word size. For example, the expression \(2^k \lfloor x \rfloor - 2^k \lfloor x - 1 \rfloor\) tells us the value
of the \(k\)th bit after the decimal point in the binary representation of a real number \(x\). Using
this formula, we could therefore access an arbitrarily large amount of information carefully packed
within the infinite digit string of a single real number \(x\).

³An example of integer division: 17 divided by 5 yields 3 with a remainder of 2. Since we forbid
use of the floor function on a real RAM, we can determine that the real number 17 divided by
the real number 5 equals the real number 3.4, but we cannot easily obtain an integer quotient and
remainder.
1.2.3 Comparison-Based Algorithms

For non-numeric problems like sorting and searching, a nice model of computation due to its simplicity is the comparison-based model, where arithmetic on input elements is forbidden; rather, we can only learn about elements of the input by comparing them pairwise. The comparison-based model is nearly identical to the RAM; the only difference is that if a memory location holds an element of data that comes directly from the input, then we must treat this element as a “black box” that can only be the subject of comparisons, not arithmetic operations. We will use the comparison model often when we study algorithms and data structures for searching and sorting. The model is ideal for these problems since it gives us very general algorithms that operate on any type of comparable data: integers, real numbers, text strings, etc. This generality comes at a price, however, since we shall see that many problems (e.g., sorting) can be solved faster on a RAM by exploiting integrality of the input.

1.2.4 Other Models

This book generally confines its discussion to the RAM, real RAM, and comparison-based models. However, just as computing systems come in many shapes and sizes, there are many more models of computation one can consider. Towards the end of this chapter, we briefly introduce the cache-oblivious model, which captures the performance of a realistic multi-level memory system much more faithfully than the RAM. We also briefly highlight models of parallel computation, which are becoming increasingly important due to the massive size of many modern computing problems, as well as the increasing availability of multi-core and distributed computing environments. Parallel models of computation are quite a bit more complicated since they need to describe how multiple processors are synchronized, how they share memory, how they communicate, and more.

There are also several more “exotic” choices for models of computation out there, many based on computers built to exploit physical principles such as:

- **Optics.** Shining light through a diffraction grating yields a pattern that gives the Fourier transform of the pattern on the grating, thereby “instantly computing” a Fourier transform.

- **Mechanics.** Shortest paths from node $x$ in a network can be computed “instantly” by gravity by building the network out of balls and strings and suspending it from the ball representing $x$; the distance each node $y$ falls relative to $x$ represents the shortest path distance from $x$ to $y$.

- **Quantum Mechanics.** A recent breakthrough of Peter Shor enables factoring large integers on a computer based on quantum principles much faster than we know how to factor on any other computational model.

- **Chemical Interaction.** A famous study of Leonard Adelman shows how the Hamiltonian path problem can be solved by mixing pieces of carefully constructed DNA representing parts of a graph in such a way that they would tend to bind together in a configuration representing a Hamiltonian path.
1.3 How to Describe an Algorithm

Algorithms can be described in various levels of detail. For example, a binary search for some value \( v \) within a sorted array \( A[1\ldots n] \) can be described in high-level technical prose as follows:

Compare \( v \) to the middle element of the array. If these match we are done. If \( v \) is smaller, recursively repeat our search on the first half of the array; if larger, repeat instead on the second half of the array. If our search narrows down to an empty subarray the process terminates, having determined that \( v \) is not present in \( A \).

A good description of an algorithm leaves no important aspect of the algorithm’s behavior unclear. Cumbersome minor details can often be safely omitted. For example, if our array has even length, then either \( A[n/2] \) or \( A[n/2 + 1] \) can serve as a “middle” element. A skilled programmer should be able to implement the algorithm being described without needing to spend time re-deriving key details.

For more detail, an algorithm may also be described like a computer program, either in terms of actual code or abstract pseudocode. Since this book focuses on high-level ideas rather than serving as a “practitioner’s handbook”, most of its algorithms are explained in high-level prose rather than code. There are plenty of other books and websites that provide good examples of specific algorithms in code. When we do provide examples of code, we use pseudocode for several reasons. As we see in Figure 1.3(a-b), pseudocode is quite similar in structure to most popular high-level programming languages, so implementing an algorithm based on a pseudocode description is straightforward. The generality of pseudocode also frees us from the necessity of releasing new revisions of the book every time a new language becomes popular. Furthermore, pseudocode avoids the administrative requirements of most programming languages (e.g., declaring variables), allowing us to focus entirely on algorithmic structure.

**Iteration Versus Recursion.** Loops and repetition are found in most algorithms, and these usually come in one of two flavors: iteration and recursion. For example, if we consider summing the contents of an array \( A[1\ldots n] \), an iterative algorithm would be described as looping sequentially through the array while maintaining a running sum. A recursive algorithm would add the first element \( A[1] \) to the sum it gets when it recursively applies itself to the remainder of the array \( A[2\ldots n] \). The choice between describing an algorithm iteratively or recursively is often a matter of personal preference, although in many cases one of the two methods leads to a simpler exposition, implementation, or analysis. Pseudocode for a recursive
1. int bsearch(int A[], int n, int v)
2. {
3.  int left = 0, right = n - 1, mid;
4.  while (left <= right) {
5.    mid = (left + right)/2;
6.    if (v == A[mid]) return mid;
7.    if (v < A[mid]) right = mid - 1;
8.    else left = mid + 1;
9.  }
10. return -1; /* Not Found */
11. }

Binary-Search:
1. left ← 1, right ← n
2. While left ≤ right:
3.  mid ← ⌊(left + right)/2⌋
4.  If v = A[mid]: Return mid
5.  If v < A[mid]: right ← mid - 1
6.  If v > A[mid]: left ← mid + 1
7.  Return "Not Found"

Rec-Binary-Search(A, left, right, v):
1. If left > right: Return "Not Found"
2. mid ← ⌊(left + right)/2⌋
3. If v = A[mid]: Return mid
4. If v < A[mid]: Return Rec-Binary-Search(A, left, mid - 1, v)
5. If v > A[mid]: Return Rec-Binary-Search(A, mid + 1, right, v)

Figure 1.3: The binary search algorithm: (a) in C/C++, and in pseudocode,
(b) from an iterative perspective and (c) from a recursive perspective.

implementation of binary search is shown in Figure 1.3(c).

The Importance of Abstraction. Let us now reconcile our high-level means
of describing an algorithm with the low-level RAM computational model, which
is only capable of performing very simple fundamental operations like adding two
words. In order to accurately analyze an algorithm’s running time in this model, it
might seem necessary to express the algorithm in the language of the RAM, which
is similar to assembly language on a digital computer with a simple instruction
set. An example of binary search written this way is shown Figure 1.4; note how
every instruction is a fundamental operation like the addition or comparison of two
words stored in CPU registers. This level of detail shows us precisely what the
algorithm is doing, and allows us to compute the exact number of fundamental
steps performed by the algorithm. However, as we shall see in the next section,
this level of detail is unnecessary when we use asymptotic analysis to describe the
running time. Moreover, such a low-level description rarely helps in explaining the
algorithm’s structure and the intuition behind its operation.

The discussion above motivates the importance of abstraction in computer science.
When describing algorithms, we should try to focus as much as possible on only
the most relevant high-level details and free our minds from distracting lower-level
1. Initialize: MOV r1, 0
   Store left index in register r1
2. LOAD r2, n
   Right index in r2
3. DEC r2
   Decrement r2 (array is zero-based)
4. LOAD r3, v
   Value to search for in r3
5. MainLoop: CMP r1, r2
   Compare r1 with r2
6. JG NotFound
   Jump if greater than
7. ADD r0, r1, r2
   Set r0 to r1 + r2
8. SHR r0, 1
   Shift r0 right by 1 (i.e., divide by 2)
9. MOV r4, A
   Move base address of A into r4
10. ADD r4, r4, r0
    Set r4 to address of A[r0]
11. LOAD r5, r4
    Load r5 with value of A[r0]
12. CMP r3, r5
    Compare r3 with r5
13. JE Found
    Equal? We’ve found our element...
14. CMP r3, r5
    Compare again
15. JG SecondHalf
    Greater? Restrict search to 2nd half
16. FirstHalf: MOV r2, r0
    Set new right index
17. DEC r2
18. JMP MainLoop
    Loop again
19. SecondHalf: MOV r1, r0
    Set new left index
20. INC r1
21. JMP MainLoop
    Loop again
22. Found: RET
    Return (index of element is in r0)
23. NotFound: MOV r0, -1
    Unsuccessful return value of -1
24. RET
    Return

**Figure 1.4:** Binary search written in (pseudo-)assembly language.

details. As this book progresses, we will use the algorithms and data structures we develop as “black boxes” to build successively larger and more complex algorithms. As a simple example, consider the 2-SUM problem: given an array \( A[1...n] \), do two numbers exist in \( A \) summing to a specified value \( v \)? A simple algorithm for 2-SUM is the following: first sort \( A \), then scan through it and use binary search to check for each element \( A[i] \) whether a “partner” element of value \( v - A[i] \) also exists in the array (remember that binary search requires a sorted array to work properly). By abstracting away the details of sorting and binary search, we have greatly simplified the exposition of our more sophisticated algorithm.

### 1.4 Characterizing Algorithm Performance

A good algorithm makes efficient use of computational resources, which can include processor, memory, network bandwidth, power consumption, and more. In this book, we focus most of our attention on minimizing running time.

#### 1.4.1 Empirical Testing

An obvious way to develop understanding of the performance of an algorithm is by experimental measurement, by running the algorithm on inputs of different sizes.
For example, in Figure 1.5(a) we investigate the running time of an algorithm for the famous traveling salesman problem (TSP), which asks us to find a minimum-length tour (Hamiltonian cycle) of a set of \( n \) cities. By running the algorithm on a set of randomly-generated inputs for varying \( n \) (here, we use \( n \) random points in the 2D plane), we can observe how the running time scales with problem size. Being a hard problem, we do not know how to optimally solve the TSP quickly. Most TSP algorithms, including the one studied in (a), therefore typically converge to a sub-optimal but still reasonably good solution over a number of iterations. In Figure 1.5(b), we show how empirical testing gives us a good sense of the rate of convergence of two different algorithms: algorithm A converges more quickly, but algorithm B ultimately produces a shorter tour.

Empirical testing is a simple and effective way to measure the anticipated performance of an algorithm on a real computing environment. It is arguably the only feasible way to analyze some algorithms, particularly those that are too complex to analyze mathematically, or those whose performance on “real world” inputs often differs substantially from what we would predict based on mathematics alone. It is also the only way to measure the impact of aspects of a real computing environment that are not captured by our abstract computing model, such as memory caching artifacts; these can sometimes lead to surprising differences between actual and theoretically-predicted performance.

On the negative side, empirical testing does not tell us the whole picture in many ways. It only measures performance for a specific hardware platform, operating system, and programming language, and this may not translate perfectly to other systems. If you ask two highly-skilled programmers to implement the same algorithm, subtle differences in their code will likely give you different performance...
1.4. CHARACTERIZING ALGORITHM PERFORMANCE

measurements. Most importantly, however, it can be very challenging to select an appropriate set of inputs on which to test. Ideally, one should use inputs that we expect to encounter in practice, although these may be so well-structured that they never reveal glaring deficiencies in the algorithm. Randomly-generated inputs are often used in empirical testing in the literature, although these can often lead to vastly-differing performance than real-world inputs. It can be a genuine challenge to conduct empirical testing of an algorithm in a truly rigorous fashion.

1.4.2 Mathematical Analysis Using Asymptotic Notation

The most important aspect of an algorithm’s performance is how it scales with input size, and we can often predict this behavior by studying the mathematical structure of the algorithm. For example, after a fair amount of work, one could ascertain that an algorithm uses between $7n^2 - 3n + 2$ and $8n^2 + 17$ fundamental RAM operations to solve a problem of size $n$. Although this is a very precise set of bounds, it is almost too much detail — what really matters here is simply that running time scales as a quadratic function of input size. As $n$ grows large, leading constant factors like the 7 or 8 as well as lower-order terms become increasingly irrelevant. We therefore say that our running time is on the order of $n^2$, which we write in using “Big Oh” notation as $O(n^2)$.

We say that a mathematical expression is $O(f(n))$ if it is bounded above by some constant times $f(n)$ as $n$ grows sufficiently large. For example, the running time expression $8n^2 + 17$ is $O(n^2)$ since it is upper-bounded by $9n^2$ for $n \geq 5$. This is called asymptotic analysis because it describes asymptotic behavior as $n$ grows very large, where the fastest-growing term dominates. Asymptotic analysis captures the essence of what matters in a running time, and allows us to analyze algorithms at a high level without the need for painstaking translation into assembly language to count individual operations. For example, if we want to check whether an $n$-element array contains two identical elements (the element uniqueness problem), we could iterate over every pair of elements and compare them. Since we are examining $\binom{n}{2} = n(n-1)/2 = O(n^2)$ pairs of elements and spending a constant amount of work (e.g., a comparison operation as well as a small amount of loop overhead) on each pair, this algorithm runs in $O(n^2)$ time.

Hidden Constants. Suppose algorithm $A$ runs in $O(n)$ time, and algorithm $B$ runs in $O(n^2)$ time. Since asymptotic expressions ignore leading constants and lower-order terms, it is actually impossible to predict without further information or empirical testing which will run faster in practice for a particular $n$. We know only that algorithm $A$ will eventually triumph as $n$ grows large. If the actual running times of $A$ and $B$ are $999999n$ and $n^2$ respectively, then indeed we lose some helpful information by saying only that “$A$’s running time is $O(n)$”. In this case, we may wish to point out that $A$’s running time has a large hidden constant, so it is clear that among $O(n)$ algorithms, $A$ is not terribly fast.

1.4.3 Worst-Case and Average-Case Behavior

If we are lucky, binary search might terminate after only a single comparison, although most invocations take more time. This is a common phenomenon, where the
running time depends on input *structure* as well as size. There are “easy” inputs the algorithm can handle quickly and “hard” inputs that take longer to process. In this situation, we typically focus on *worst-case* running time. For binary search, every unsuccessful iteration narrows the size of the subarray we are searching by at least a factor of 2, so after at most $\log_2 n$ such iterations our search is narrowed to a single element and the algorithm terminates. Since every iteration involves a constant number of fundamental operations, binary search therefore has an $O(\log n)$ worst-case running time (note that we can simply say $O(\log n)$ instead of $O(\log_2 n)$ since logs of all bases differ only by constant factors, and leading constant factors disappear inside asymptotic expressions). Algorithm designers focus on worst-case behavior not because they are pessimists, but because this provides a very strong guarantee — no matter how bad our luck, or even if we receive input from a malicious adversary, binary search always runs in $O(\log n)$ time.

If worst-case inputs are rarely seen in practice, then *average-case* analysis may be preferable, where we assume a particular probability distribution over all possible inputs of size $n$ (presumably the distribution we expect to see in practice) and compute the expected running time. For example, if we are equally likely to search for any of the $n$ elements in our input array, the expected running time of binary search is still $O(\log n)$. Computing the average-case performance of a complicated algorithm is unfortunately often challenging from a mathematical perspective; for such algorithms, empirical testing may be the only way we can measure anticipated performance in practice.

### 1.4.4 Common Running Times

It is good to be intuitively familiar with common running times encountered during algorithmic analysis. The fastest possible running time is an $O(1)$ (i.e., *constant*) worst-case running time — upper bounded by a fixed universal constant independent of problem size. However, although parts of an algorithm often take $O(1)$ time, it is unusual for an entire algorithm with input size $n$ to run in $O(1)$ time, since any *sublinear* algorithm (faster than $O(n)$ time) does not even have enough time to examine its entire input.

*Logarithmic* running times of $O(\log n)$ are typical for algorithms like binary search that in each step are able to reduce a problem’s size by a constant factor (say, by half). More generally, running times like $O(\log^2 n)$ or $O(\log \log n)$ or $O(\sqrt{\log n})$ are known as *polylogarithmic* running times since they are bounded by some polynomial function of $\log n$. These are all very fast running times — even for a ridiculous input size like $n = 2^{100}$, binary search requires at most 100 iterations.

Running times like $O(n)$ (i.e., *linear*), $O(n \log n)$, $O(n^2)$ (i.e., *quadratic*), and $O(n^3)$ (i.e., *cubic*), are called *polynomial* running times since they are bounded by a polynomial in the input size $n$. Linear-time algorithms and $O(n \log n)$ algorithms have good performance, since these algorithms require roughly the same amount of time to execute as it takes to simply read their input. As we start moving toward quadratic running times and higher, we begin to encounter more substantial limitations on the sizes of problems we can quickly solve.

Beyond polynomial time, things get very bad very fast. Algorithms with *exponential* running times like $O(2^n)$ and *factorial* running times like $O(n!)$ can only solve very
1.4. CHARACTERIZING ALGORITHM PERFORMANCE

<table>
<thead>
<tr>
<th>Running Time</th>
<th>Largest value of $n$ for which computation takes at most...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 millisecond</td>
</tr>
<tr>
<td>Constant</td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td>$O(\alpha(n))$</td>
</tr>
<tr>
<td></td>
<td>$O(\log^* n)$</td>
</tr>
<tr>
<td></td>
<td>$O(\log \log n)$</td>
</tr>
<tr>
<td></td>
<td>$O(\sqrt{\log n})$</td>
</tr>
<tr>
<td></td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td></td>
<td>$O(\log^2 n)$</td>
</tr>
<tr>
<td></td>
<td>$O(\sqrt{n})$</td>
</tr>
<tr>
<td></td>
<td>$O(n)$</td>
</tr>
<tr>
<td></td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td></td>
<td>$O(n \sqrt{n})$</td>
</tr>
<tr>
<td></td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td></td>
<td>$O(n^3)$</td>
</tr>
<tr>
<td></td>
<td>$O(n^{\log n})$</td>
</tr>
<tr>
<td></td>
<td>$O(2^n)$</td>
</tr>
<tr>
<td></td>
<td>$O(n!)$</td>
</tr>
<tr>
<td></td>
<td>$O(n^n)$</td>
</tr>
</tbody>
</table>

**Figure 1.6:** List of common run times, ordered from fastest to slowest. To compute the approximate maximum value of $n$ for which computation takes a certain amount of time, we assume that all logs are base 2 and that the leading constant in the $O(\cdot)$ expressions is such that our algorithm can perform 1 billion iterations per second (this is somewhat aggressive for modern personal computers at the time of writing). The special functions $\alpha(n)$ and $\log^* n$ are defined in the next chapter.

small problem instances even given centuries of computing time. These functions grow so quickly that even if we were to construct computing devices that were a million times faster than those of today, an algorithm running in, say, $2^n$ steps could still only solve problems that were of size $n + 20$ in roughly the same amount of time it takes to solve a problem of size $n$ today. Unfortunately, exponential running times seem unavoidable for certain problems, such as the NP-complete problems.

**“Efficient” Algorithms.** To the computer scientist, the threshold between polynomial and super-polynomial (e.g., exponential) running time is particularly notable, since historically this has been regarded as the threshold between “efficient” and “inefficient”. In fact, theoreticians often equate the word “efficient” with “polynomial time”. It may seem a bit dubious to call an algorithm with polynomial running time $O(n^{100})$ “efficient” (in theoretical computer science, one actually does encounter such running times!), but one must bear in mind that this is still better than $O(2^n)$ for very large values of $n$. In this book we generally look for “efficient”
solutions that not only run in polynomial time, but are viable in practice as well.

1.4.5 Output Sensitivity

Algorithms that solve numerical problems come in two flavors: exact algorithms terminate in a finite amount of time with an exact answer, even on real-valued input\(^4\), while other algorithms converge asymptotically to a correct answer over time. For example, if we start with \( x = 1 \) and repeatedly set \( x = x/2 + 1/x \), then this quickly approaches \( x = \sqrt{2} \) [Why?]. However, since \( \sqrt{2} \) is irrational, the algorithm theoretically never terminates, and only gets closer and closer to \( \sqrt{2} \) the longer it runs. For iteratively-converging algorithms like this, running time is usually characterized in terms of convergence rate, the amount of time required to obtain each successive digit of accuracy in the output.

We could say that the running time of an iteratively-converging algorithm is output-sensitive, meaning that it depends on how much output precision is requested. We will see another form of output sensitivity when we study data structures, since the running time of a data structure query often depends on the amount of data returned by the query. For example, a query operation over a data structure of size \( n \) might have running time \( O(k + \log n) \), where \( k \) is the size of the output, and \( O(\log n) \) is the fixed overhead of the query independent of output size.

1.4.6 Input Sensitivity

For many algorithms, running time depends only on the number of input elements \( n \). However, some RAM algorithms have input-sensitive running times that depend on \( n \) as well as the magnitude of the \( n \) integers provided as input. For instance, suppose our input consists of \( n \) integers in the range \( 0 \ldots C - 1 \), so each is described by \( \log_2 C \) bits. Our running time is strongly polynomial if it depends polynomially on just \( n \) (i.e., on the number of words in the input), and weakly polynomial if it depends polynomially on \( n \) and \( \log C \) (i.e., on the number of bits in the input).

Since complexity theory measures the true input size to an algorithm in bits, both types of algorithms rightfully run in “polynomial time”. Given a choice between the two, strong polynomial time is generally preferred, for aesthetic simplicity as well as the peace of mind that large numbers in the input have no impact on running time. For some problems, however, if we know that \( C \) is small, we might be able to devise an input-sensitive algorithm with a weakly polynomial running time that is faster in practice. The distinction between strong versus weak polynomial time generally applies only to algorithms with integer inputs.

A running time depending polynomially on \( n \) and \( C \) (rather than \( \log C \)) is said to be pseudo-polynomial. This term is somewhat misleading, since pseudo-polynomial running times do not count as polynomial running times at all, being exponential in the size of the input measured in bits (since \( C \) is exponentially large in \( \log C \)). Pseudo-polynomial running times like \( O(nC) \) scale gracefully in terms of the number

---

\(^4\)Since actual digital computers cannot store real numbers perfectly, even a so-called “exact” numerical algorithm might end up producing a solution that isn’t exactly correct due to accumulated round-off errors (this is one danger of the real RAM, since it allows us to pretend this issue does not exist in practice). We address this pitfall in further detail in Chapters ?? and ??.
of integer input elements, but not in terms of the size of these elements. A pseudo-

polyomial algorithm might therefore take 100 times as long to run if the numbers 
in its input are multiplied by 100, so caution is advised when considering the use 
of such algorithms. [Examples of algorithms of the three types above]

1.4.7 Lower Bounds and Optimal Running Times

Since \( O(\cdot) \) notation provides merely an asymptotic upper bound, we could truthfully 

claim that an algorithm runs in \( O(n^2) \) time even if it actually runs in only constant 

time. In this case, we would say the \( O(n^2) \) bound is not tight. To be more precise in 

our asymptotic analysis, let us introduce two more expressions. We use \( \Omega(\cdot) \) ("Big 

Omega") to indicate an asymptotic lower bound. An algorithm has running time 

\( \Omega(n^2) \) if the running time is bounded below by some constant time \( n^2 \) as \( n \) grows 
sufficiently large. We use the notation \( \Theta(\cdot) \) to indicate both asymptotic lower and 

upper bounds; running time is \( \Theta(n^2) \) if it is both \( O(n^2) \) and \( \Omega(n^2) \). One may wish 
to think of \( O, \Omega, \) and \( \Theta \) as the asymptotic equivalents of \( \leq, \geq, \) and \( = \). That is, an 

\( O(n^2) \) algorithm has a running time that is asymptotically no worse than \( n^2 \), an 

\( \Omega(n^2) \) algorithm has running time that is asymptotically at least as bad as \( n^2 \), and 
a \( \Theta(n^2) \) algorithm has a running time that grows precisely at an asymptotic rate 
of \( n^2 \), ignoring constant factors and lower-order terms.

We can often characterize worst-case running time more precisely using \( \Theta(\cdot) \) nota-
tion. For example, it is correct to say that binary search runs in \( O(\log n) \) time, but we 

convey more information by saying it runs in \( \Theta(\log n) \) time in the worst case. 

In much of the technical literature, one finds \( O(\cdot) \) being used when \( \Theta(\cdot) \) is perhaps 
more appropriate. People will often say, for example, that a running time is \( O(n^2) \) 
when they really mean either \( \Theta(n^2) \), or \( \Omega(n^2) \) in the worst case.

Consider the problem of computing the maximum value in an \( n \)-element array. We 
can easily solve this problem in \( O(n) \) time (actually \( \Theta(n) \) would be more precise). 
However, note that any algorithm that solves this problem must at least look at all 
\( n \) values in the input. As a result, we can say that there is a lower bound of \( \Omega(n) \) 
on the worst-case running time of any algorithm that solves this problem, and that 
our \( O(n) \) algorithm therefore has an optimal worst-case running time. In Chapter 
3 we will learn how to prove more sophisticated lower bounds on certain problems. 
For example, any comparison-based or real RAM algorithm that sorts \( n \) elements 
must spend \( \Omega(n \log n) \) time in the worst case.

A somewhat playful outlook of the field of algorithms is in terms of an epic struggle 
between two forces: the “good guys” who design efficient algorithms and prove 
that you can solve certain problems efficiently, and the “bad guys” who prove lower 
bounds indicating that you cannot solve certain problem efficiently. When the two 
sides meet at the same asymptotic running time, this means the complexity of a 
problem is in some sense resolved. However, there are many problems for which 
there is still an unsightly gap between the best-known upper bound and the best-
known lower bound. For example, the problem of multiplying two \( n \times n \) matrices 
has a trivial lower bound of \( \Omega(n^2) \) due to the need to examine all input data, but the 
best known upper bound is currently \( O(n^{2.3727}) \). In this case, one would hope that 
clever researchers will eventually close the gap by proving a stronger lower bound 
or developing an algorithm that improves the upper bound. One might also hope
that the upper bound would be the one to change, since nature would be somewhat
cruel if an arbitrary-looking running time like $O(n^{2.3727})$ was actually optimal for
such a fundamental problem!

1.5 Data Structures

The study of data structures goes hand in hand with the study of algorithms. Algo-
rithmic problems are typically posed in terms of abstract mathematical entities
such as sequences, sets, graphs, etc., which can be represented in memory several
different natural ways. Choosing the best representation for our data is of funda-
mental importance when designing algorithms, since it can have a dramatic impact
on performance. One often finds the subjects of “algorithms” and “data structures”
treated separately, for instance in two different courses in an undergraduate com-
puter science curriculum. However, these two subjects should ideally be studied
together as part of the same whole, since they are intrinsically coupled. We will
study data structures extensively in Chapters 4 through 9; for now, we simply wish
to motivate their importance and highlight some key concepts. We also discuss two
very fundamental data structures, arrays and linked lists, as a prerequisite for the
next few chapters.

A data structure provides a concrete strategy for storing and interacting with data.
For example, an array $A[1 \ldots n]$ is a very simple data structure consisting of $n$
consecutive elements in memory, and it provides a simple and natural way to rep-
resent an abstract mathematical sequence $A_1 \ldots A_n$ (we use brackets $A[i]$ to denote
the elements of an array, where subscripts $A_i$ are used to represent elements in an
abstract mathematical sequence).

Abstract Data Types. There is an important distinction to be made between
the abstract specification of a data structure (e.g., a sequence, set, or map), also
known as an abstract data type, and a concrete implementation of a data structure
in accordance with this specification (e.g., an array). An abstract data type only
describes the operations to be supported by a data structure, and does not pre-
scribe any particular way of implementing these operations. For example, a data
structure for a dynamic sequence $A_1 \ldots A_n$ must support the following fundamental
operations:

- $Access(A, i)$. Retrieves the value of the $i$th element, $A_i$.
- $Modify(A, i, v)$. Sets the value of $A_i$ to $v$.
- $Insert(A, i, v)$. Inserts a new element of value $v$ in the $i$th position of $A$. All
  former elements $A_i, A_{i+1}, \ldots$ are shifted upward by one index (so they become
  $A_{i+1}, A_{i+2}, \ldots$) to make room for the new element.
- $Delete(A, i)$. Deletes $A_i$ from the sequence. Former elements $A_{i+1}, A_{i+2}, \ldots$
  are shifted downward by one index (to $A_i, A_{i+1}, \ldots$) to close the gap created
  by the deleted element.

There are often many different ways to implement a given abstract data type. For a
dynamic sequence, the simplest two alternatives are arrays and linked lists, although
later in Chapter 6 we will discuss many others, such as balanced trees and skip lists.
Static Versus Dynamic Data Structure. A static data structure is built once, after which it supports operations to query but not modify its state. Some data structures are so static that they can even operate from read-only memory after being built. Dynamic data structures like dynamic sequences allow both modification and query operations. As one might expect, dynamic data structures can be more challenging to design than their static counterparts.

Incremental, Decremental, and Fully-Dynamic Structures. Dynamic data structures come in three flavors: incremental structures can handle insertions of new elements but not deletions, decremental structures can handle deletions but not insertions, and fully dynamic structures can handle both. The fully dynamic case is, not surprisingly, often much more difficult. For example, a priority queue is a type of data structure that tracks the minimum of a dynamic set of elements. Incremental and decremental priority queues are relatively easy to implement, but the fully dynamic case leads to an entire chapter worth of discussion (Chapter 5).

Dynamic Algorithms Versus Data Structures. The traditional “one-off” model for solving an algorithmic problem asks us to read the input, perform some processing, write the output, and then terminate. Given a second instance differing little from the first, we might hope to solve it more efficiently by maintaining some of the state from the previous computation in an appropriate data structure. This is sometimes known as re-optimizing a solution, when re-solving an optimization problem after minor modification of its input. We can consider making nearly any algorithmic problem dynamic in this fashion, switching our outlook on the problem to one more centered on data structures.

1.5.1 Arrays Versus Linked Lists

Consider how to implement a dynamic sequence. If we use an array, then the access and modify operations run very quickly, in $O(1)$ time. However, arrays are not well-suited for insertion or deletion. As we see in Figure 1.7(a), inserting a new element into an array takes $\Theta(n)$ worst-case time since we first need to slide over potentially all of the array elements by one cell to make room for the new element. Similarly, deletion takes $\Theta(n)$ worst-case time since we may need to slide a large block of elements back by one cell to plug the hole created by the deleted element. More sophisticated data structures (e.g., problem 73) mitigate this issue by leaving periodic “gaps” in an array, so insertions and deletions do not need to displace so many elements.
Linked lists give us another simple way to implement a dynamic sequence. As shown in Figure 1.7(b), the elements of the sequence are stored in haphazard memory locations, and each element maintains a pointer to its successor. The end of the list is usually indicated by a special “null” pointer from the final element, or by including a dummy sentinel element as the final element\textsuperscript{6}. The efficient operations on a linked list are exactly the opposite of those on an array. It takes $\Theta(n)$ time in the worst case to access a particular element given its index, as we must walk down the list from the beginning until we reach the desired element. However, once we have scanned to the appropriate location in a list, we can modify it in $O(1)$ time and also insert and delete in $O(1)$ time, since this requires only the modification of a small handful of pointers as opposed to the relocation of massive amounts of data (unfortunately, since we must include the scanning time in these operations, their running times are also technically $\Theta(n)$ in the worst case). Occasionally we will find it convenient to be able to walk backwards as well as forwards in a list, in which case we can use a doubly-linked list where each element points to both its predecessor and its successor\textsuperscript{7}.

As shown in Figure 1.8, there is a dramatic trade-off between the array and linked list in terms of which operations are efficient. The best data structure for implementing a dynamic sequence therefore depends on our particular application. If we expect to perform few insertions and deletions, the array may be ideal, and if all insertions and deletions are concentrated in a small area, the linked list may do better. Trade-offs of this sort are extremely common when we study data structures, and often there is no single “best” implementation for a particular type of data structure.

As a note to the introductory student, arrays and linked lists are such fundamental data structures that it is absolutely crucial to understand them well; for example, new students often mistakenly try to insert new elements in $O(1)$ time in the mid-

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
         & Array & Linked List & Balanced Tree or Skip List \\
\hline
Access   & $O(1)$ & $O(n)$     & $O(\log n)$         \\
\hline
Modify   & $O(1)$ & $(+O(n))$  & $O(\log n)$         \\
\hline
Insert/Delete & $O(n)$ & $(+O(n))$  & $O(\log n)$         \\
\hline
\end{tabular}
\caption{Running times for the operations of three types of data structures for representing a dynamic sequence. To be more precise, we could say that the running time of modify, insert, and delete in a linked list is really $O(1)$ once we have scanned (in $O(n)$ time) to the appropriate location in the list. We will study balanced trees and skip lists later in Chapter 6.}
\end{table}

\textsuperscript{6}If our list ends with a dummy sentinel element, then a simple way to delete element $e$ given a pointer to $e$ is simply to overwrite $e$ with a copy of its successor. Without a dummy sentinel element, however, this trick no longer works (specifically, for deleting the final element), and we can only effectively delete $e$ if we have a pointer to $e$’s predecessor.

\textsuperscript{7}Instead of storing the predecessor pointer $p$ and successor pointer $s$ separately, a clever space-saving trick is to store just the single value $p \oplus s$, where $\oplus$ is the XOR operation. Owing to how XOR works, this still allows effective navigation in either direction. For example, when traveling from $p$, we can take $p \oplus (p \oplus s) = s$ to recover $s$, and vice versa.
dle of an array. A programming background usually helps, since arrays and linked lists are familiar objects in most computer programs. If you are unsure about your background, arrays and linked lists are covered in excruciating detail in many other books, which you may wish to consult for reference. The following are two fun yet challenging problems that can test your mastery of linked lists and arrays.

**Problem 1 (Loopy Linked Lists).** Suppose we are given a pointer to the first element in an \( n \)-element linked list (we aren’t given \( n \), however), and told that it ends in a loop, where the last element points back to some earlier element in the list. We want to compute the number of elements in the list, \( n \). This is easy to do if we use \( O(n) \) extra memory, since we can attach a marker to each element as we scan through the list, making it easy to detect when we revisit an element. However, in some cases it may be undesirable to modify the list, such as in a parallel shared memory environment with several processes accessing the list. Try to devise a method that computes \( n \) in \( O(n) \) time without modifying the list, and using only \( O(1) \) auxiliary memory. As a hint, consider the looping and non-looping parts of the list separately. [Solution]

**Problem 2 (Virtual Initialization).** Arrays must typically be initialized prior to use, since when we allocate an array of \( n \) words of memory, they usually start out filled with “garbage” values (whatever data last occupied that block of memory). In this problem, we wish to design a data structure that behaves like an array (i.e., allowing us to retrieve the \( i \)th value and modify the \( i \)th value both in \( O(1) \) time), but which allows for initialization to a specified value \( v \) in only \( O(1) \) time, instead of the usual \( \Theta(n) \) time. If we ask for the value of an element we have not modified since the last initialization, the result should be \( v \). The data structure should occupy \( O(n) \) space in memory (note that this could be twice or three times as large as the actual space we need to store the elements of the array), and it should function properly regardless of whatever garbage is initially present in this memory. As a hint, try to combine two different representations of the data in the array. [Solution]

The trick behind the first problem can be used in a surprisingly wide range of applications, from parallel processing (problem 51(l)) to infinite loop detection (problem 116) to factoring integers (Section ??). As a consequence of the second problem, we can assume in theory that initialization of an array requires only constant time without affecting the asymptotic time or space requirements of our algorithms (however, this technique is rarely used, since it requires so much extra space).

### 1.5.2 Stacks and Queues

While on the subject of dynamic sequences, arrays, and linked lists, now is a good time to mention two useful related structures we often encounter. A **stack** is a type of data structure supporting these two operations:

- **Push** \((e)\). Inserts a new element \( e \) into the stack.
- **Pop**. Removes and returns the most-recently-inserted element.

Elements pass through a stack in a **Last-In-First-Out** (LIFO) fashion (picture a stack of papers, where we only add or remove papers at the top of the stack). On the other hand, a **queue** is a type of data structure that follows the **First-In-First-Out** (FIFO) discipline (picture a line of people waiting at a busy ticket counter).
Queues support the following two operations:

- **Enqueue(e).** Inserts a new element $e$ into the queue.
- **Dequeue.** Removes and returns the least-recently-inserted element.

A stack is just a dynamic sequence in which insertions and removals occur at only one end. For a queue, insertions occur at one end and removals at the other. Both arrays and (doubly-)linked lists can easily handle insertion and deletion at the ends of a sequence in $O(1)$ time (e.g., by maintaining pointers to the first and last elements in a doubly-linked list). It is therefore quite straightforward to build stacks and queues whose operations all take only $O(1)$ time. In fact, we can also build a **double-ended queue** (abbreviated *deque* or *dequeue*, pronounced “deck”, and not to be confused with the *dequeue* operation above), in which we can insert and remove elements at both ends.

When adding elements to the end of an array, we need to be somewhat mindful of memory allocation issues. An array is typically stored within a block of memory allocated with some fixed size, and expansion beyond the lower or upper boundary of this block might overwrite memory reserved for other purposes. Fortunately, we can overcome this difficulty by using a **circular array**, shown in Figure 1.9, which logically wraps around within the block. By maintaining pointers to the first and last elements of the array, it is easy both to compute the index of any particular array element (accounting for wrap-around) and to insert or delete elements at the endpoints of the array, all in only $O(1)$ time.

### 1.6 Complexity Theory

Many algorithms courses focus on problems that we know how to solve efficiently (i.e., in polynomial time), leading to the mistaken impression that there is an efficient algorithm waiting to solve almost every problem one is likely to encounter. Sadly, nothing could be farther from the truth. Many important real-world problems seem to have no efficient algorithmic solution. We have already seen one such problem, the *Hamiltonian path problem*, asking for a path through a graph that visits every node exactly once. In this section, we introduce the concept of NP-hardness and how it can help us characterize and better understand many of the “hard” problems out there.
When given a problem like the Hamiltonian path problem for which we are unable to find an efficient (polynomial time) algorithm, two possibilities exist:

1. The problem is at fault for being genuinely “hard”, or
2. The blame rests instead on our own lack of algorithmic design skills.

Our pride would certainly have us prefer option one, which we can establish by showing that our problem belongs to a class of known “hard” problems, with the \( \text{NP-hard} \) problems being by far the most popular. The \( \text{NP-hard} \) problems (formally defined in a moment) are closely related in a way such that a polynomial-time solution for any one would imply a polynomial-time solution for them all. Since nobody has managed to produce such a polynomial-time algorithm yet for any of the thousands of known \( \text{NP-hard} \) problems, we strongly suspect that these problems are in fact genuinely “hard”.

### 1.6.1 The Complexity Classes P and NP

We now build towards a more precise description of the \( \text{NP-hard} \) problems. For starters, we focus on problems having “yes” or “no” answers, known as \textit{decision problems}. In the field of complexity theory, we often focus on these problems, since a decision problem has a very simple mathematical representation as a (possibly infinite) set of binary strings, corresponding to binary encodings of all the inputs of the problem for which the answer is “yes”. Conveniently, we don’t lose too much generality by restricting our focus to decision problems. For example:

- The Hamiltonian path problem is not a decision problem since it asks for a \textit{path} through a graph as output. However, an efficient solution for its \textit{decision variant} (the problem “does this graph contain a Hamiltonian path?”) could be used to solve it efficiently, by repeatedly deleting edges from a graph as long as the decision variant tells us that a Hamiltonian path remains after the deletion. At the end, only a Hamiltonian path remains. The Hamiltonian path problem is therefore solvable in polynomial time if and only if the same is true for its decision variant.

- The traveling salesman problem (TSP) is not a decision problem since it asks for the numeric \textit{length} of a minimum tour of \( n \) cities (i.e., the minimum length of a Hamiltonian cycle of a graph where edges have varying lengths). However, an efficient algorithm for its decision variant (the problem “does there exist a tour of length at most \( X \)?”) could be used to solve it efficiently, by binary searching on \( X \). At each step, we use the result of the decision problem to test if a specific guess for \( X \) is too high or too low. Since only a polynomial number of invocations of the decision variant are needed\(^8\), a polynomial-time algorithm for the decision variant again implies a polynomial time algorithm for the original problem.

\(^8\)The input for an instance of the TSP, written in binary, contains a binary string describing the integer length of each edge. Since the optimal tour length is a sum of a subset of these numbers, it can be described by a binary string whose length is at most the input size in bits. Binary search for a number described by at most \( b \) bits takes \( O(b) \) time, so the number of iterations of binary search is polynomially-bounded by the total input size in bits.
We define the complexity class \( P \) as the set of all decision problems that can be solved in polynomial time. Technically, this means polynomial time on a Turing machine, but a problem solvable in polynomial time on a Turing machine is also solvable in polynomial time on a RAM and vice-versa. We can easily simulate the operation of any Turing machine algorithm on a RAM, and we can also simulate the operation of any RAM algorithm on a Turing machine with only a polynomial blow-up in running time.

Now things get a bit weird. We define the complexity class \( NP \) (short for “non-deterministic polynomial”) as the set of decision problems solvable in polynomial time by a non-deterministic algorithm. Nondeterminism is a purely theoretical concept that strikes many students as bizarre when first introduced. As an algorithm executes, it often reaches branch points where it sequentially iterates through several possible choices. For example, a trivial algorithm to search for a Hamiltonian path might enumerate all \( n! \) orderings of the \( n \) nodes in a graph, checking each one in sequence to see if the necessary edges are present so as to form a Hamiltonian path. A nondeterministic algorithm has the (completely unrealistic!) ability to follow all of these branches at the same time, rather than in sequence, somewhat like a parallel computer with an unlimited number of processors. Since a nondeterministic algorithm follows every possible execution path simultaneously, it will identify a “yes” instance of a problem as such as long as some execution path confirms this fact in polynomial time (e.g., by discovering a valid Hamiltonian path).

A simpler way to think of \( NP \) is that it contains all decision problems for which a “yes” instance can be verified as such in polynomial time, as long as we are given a valid solution\(^9\). This tells our nondeterministic algorithm which execution path to follow, allowing it to conclude that this is indeed a “yes” instance in polynomial time without the need for nondeterminism after all. The decision version of the Hamiltonian path problem is in \( NP \) since if a graph contains a Hamiltonian path, we can verify this fact in polynomial time if we are told the path (i.e., we can check that the path visits each node exactly once). Since it is usually much easier to verify a solution of a “yes” instance in polynomial time than it is to solve a problem outright in polynomial time, it is generally much easier to determine that a problem belongs to \( NP \) than \( P \). \( NP \) contains a vast number of problems, including most of the problems in this book, many of them quite important in practice, and many of which we do not know how to solve efficiently.

### 1.6.2 NP-Hardness and the Satisfiability Problem

In the 1970s, Karp, Cook, and Levin defined the class of \( NP\)-hard problems. A problem is \( NP\)-hard if the existence of a polynomial-time algorithm for that problem would imply the existence of a polynomial-time algorithm for every problem in \( NP \). A related term is \( NP\)-complete, which refers to an \( NP\)-hard problem that itself belongs to \( NP \) (in casual conversation, you may hear the terms \( NP\)-hard and \( NP\)-complete used somewhat interchangeably).

If someone manages to produce a polynomial-time algorithm for even a single \( NP\)-hard problem, this would imply that all problems in \( NP \) have polynomial-time

\(^{9}\)More generally, we need to be able to verify a “yes” instance as such in polynomial time as long as we are provided with a suitable polynomial-length binary string of “advice”.

---

DRAFT. Not for redistribution. Copyright © Brian C. Dean
1.6. COMPLEXITY THEORY

Figure 1.10: The two possible relationships between P and NP: (a) P ≠ NP and (b) P = NP. The vast majority of computer scientists strongly believe that (a) is the correct relationship. The structure of a polynomial-time reduction from problem A to (a polynomially-solvable) problem B to solve problem A indirectly in polynomial time is shown in (c).

solutions and therefore also belong to P, meaning P and NP are the same class. However, since the world’s smartest mathematicians and computer scientists have so far failed to solve any NP-hard problem in polynomial time, it is strongly suspected that the NP-complete problems are not in P, as depicted in Figure 1.10(a). However, this has never been successfully proven, and the question of whether or not P = NP remains one of the most challenging (and certainly the most famous) open problem in computer science today.

The first problem to be proven NP-hard was the satisfiability problem (abbreviated SAT), which asks whether a Boolean formula such as \( (x_1 \lor x_2) \land x_2 \) evaluates to true for some setting of its variables. The famous Cook-Levin theorem showed that SAT is NP-hard, since the polynomial-time solvability of any problem in NP can be translated into the question of determining satisfiability of a huge (but still polynomial-size) Boolean formula.

1.6.3 Proving NP-Hardness via Reductions

Fortunately, the Cook-Levin theorem does not need to be re-enacted every time we need to prove other problems are NP-hard. Now that we have one NP-hard problem, we can prove other problems are NP-hard by using polynomial-time reductions. The notion of a reduction is an important and fundamental concept in computer science, since it allows us to relate the hardness of one problem with that of another. If we can easily express an instance of problem A in terms of an “equivalent” instance of problem B, then: (i) we automatically get an algorithm for solving A if we already have an algorithm for solving B, and (ii) we know that B must be hard to solve if A is known to be hard to solve.

A prototypical reduction is shown in Figure 1.10(c). To show that problem B is

---

10The symbols \( \lor \) and \( \land \) mean OR and AND, and the notation \( \overline{x}_1 \) means NOT \( x_1 \).
NP-hard, we show that we could indirectly solve some known NP-hard problem A (and hence, every other problem in NP) in polynomial time, if only B were solvable in polynomial time. One could conceive of more complicated reductions than the one shown in Figure 1.10(c); for example, we could solve problem A in polynomial time by making not one, but a polynomial number of invocations of a polynomial-time algorithm for B, and by recombining all of the outputs in polynomial time into a single output for problem A. For most of the problems, however, the simpler style of reduction suffices. The direction of reduction is something new students often get wrong, since instinctively it might seem that we should prove a problem is NP-hard by reducing it to a known NP-hard problem, rather than the reverse (which is the correct direction). When in doubt, remember Figure 1.10(c).

We write $A \leq_p B$ to indicate that $A$ has a polynomial-time reduction to $B$. It makes sense to use the “$\leq$” symbol since this is saying that $A$ is “no harder” than $B$ with respect to the question of polynomial-time solvability — in particular, that if $B$ can be solved in polynomial time, then so can $A$. Using this notation, the Cook-Levin theorem says that $A \leq_p SAT$ for every problem $A$ in NP. If we construct a polynomial-time reduction from SAT to some other problem $B$, then $B$ must also be NP-hard, since $A \leq_p SAT \leq_p B$ for every problem $A$ in NP.

Thousands of problems have been proven NP-hard, so one can find thousands of reduction proofs in the literature. These range from relatively straightforward proofs [Example: $SAT \leq_p 3SAT$] to more involved proofs that require some serious inspiration [Example: $SAT \leq_p CLIQUE$] [Example: $3SAT \leq_p PARTITION$]. Some of the more complicated NP-hardness proofs out there are true works of art!

**Problem 3 (Reduction from the Partition Problem).** The NP-hard partition problem is the following: given $n$ numbers $a_1 \ldots a_n$, is there a partition of these numbers into two subsets whose sums are both equal? This problem is often useful in constructing NP-hardness reduction proofs. As a good practice exercise, give simple reduction proofs of the NP-hardness of the following problems from later in the book: the 0/1 knapsack problem (problem 199(b)), bin packing (problem 201), minimum-makespan scheduling (problem 204), and SONET ring loading (problem 200). [Solution]

Although P and NP are perhaps the best-known, many other interesting complexity classes of problems have been studied in the literature; the rabbit hole goes quite deep, so to speak. Complexity theory is an integral part of the foundation of theoretical computer science, and any student who is interested in pursuing serious study in computer science should certainly feel encouraged to investigate this field in greater depth.

### 1.7 Different Flavors of Algorithms

Every computational problem we study may be interesting to investigate from the perspective of several different “algorithmic models”. For example, what if we are allowed to use randomness in our algorithm? What if we are interested in algorithms that give approximate, rather than exact, solutions? What if the input arrives “online” rather than all at once, and our algorithm must commit to certain decisions before it sees the entire input? What if our memory is not even large
1.7. DIFFERENT FLAVORS OF ALGORITHMS

enough to store the entire input? What if we have access to a more powerful parallel or distributed computing environment? This section describes some of the more common flavors of algorithms the reader may encounter.

1.7.1 Randomized Algorithms

Randomized algorithms make random choices during their execution. For example, when we perform binary search, we normally compare our target word to the middle word in the dictionary, then recurse the first or second half as appropriate. Suppose instead that we compare against a random word (this perhaps better describes what happens when you are manually looking up a word in the dictionary, since it is difficult to find the exact middle word). We will see in the next chapter that this randomized binary search still runs in $O(\log n)$ time with high probability.

There are several pros and cons of using randomized algorithms. They can often be much simpler, elegant, and efficient than their deterministic (non-random) counterparts. Additionally, randomization can thwart a malicious adversary from trying to make our algorithm run slowly: every deterministic algorithm has bad inputs on which it runs the slowest, but many randomized algorithm have no particular bad inputs, since worst-case behavior is now only function of random chance. On the negative side, however, randomized algorithms can be more challenging to analyze (requiring tools from probability theory), they can be harder to debug (since they rarely run the same way twice), and random behavior in an algorithm may not be appropriate for certain mission-critical applications.

There are two main types of randomized algorithms:

- **“Las Vegas” Algorithms.** A Las Vegas algorithm always produces a correct answer. Randomness only manifests itself in the running time, which can vary based on the random choices made by the algorithm. With these algorithms, we typically wish to analyze the expected running time\(^{11}\) or to prove an even stronger statement like “the running time is $O(n \log n)$ with high probability”.

- **“Monte Carlo” Algorithms.** These algorithms have deterministic running times, but they will occasionally output an incorrect solution. This may seem quite objectionable, but we can usually reduce the probability of error to an amount that is negligible in practice (e.g., $10^{-22}$, less than the probability that you are struck by lightning, hit by a meteor, and bitten by a shark all in the same year\(^{12}\)).

A Las Vegas algorithm is always “better” than a Monte Carlo algorithm, since we can convert a Las Vegas algorithm into a Monte Carlo algorithm by terminating with a wrong answer if we exceed some deterministic time threshold. A Monte Carlo algorithm can be converted to a Las Vegas algorithm only if we have a procedure

\(^{11}\)It is important to note that analysis of the expected running time of a randomized algorithm is different from the “average-case” analysis of a deterministic algorithm. In the first case, randomness occurs in the algorithm, and we are still performing a worst-case analysis over all possible inputs. In the second case, randomness occurs in the input, in that we assume a probability distribution over all possible inputs reflecting what we expect to see in practice.

\(^{12}\)A rough estimate assuming independence of these events; it is not recommended to swim in shark-infested waters during an electrical storm at the peak of a meteor shower!
that checks a solution for correctness, in which case we keep repeating until we obtain a correct solution.

In order to use randomness, we modify our computational model to assume access to a stream of random bits. Unfortunately, most computers cannot generate truly random bits, so in practice we typically rely on a pseudorandom number generator to produce bits that seem “random enough”; we discuss methods for pseudorandom number generation in greater depth in Section 7.5. Since our mathematical analyses assume perfectly random bits, it is true that they technically may not apply once we start using pseudorandom bits; however, in practice one rarely notices grossly unanticipated behavior due to this discrepancy.

1.7.2 Approximation Algorithms

Important NP-hard (or even harder) problems show up in practice all the time. When faced with one of these problems, we know that it is probably not possible to find an optimal solution in polynomial time in the worst case. What can we do, then, if we need to solve large problem instances? One possibility is to try considering special cases that might have polynomial-time solutions. Another is to simply implement the fastest possible algorithm and hope that it runs in a reasonable amount of time — for many problems the worst-case behavior only results from a very small set of particularly ill-structured inputs that rarely occur in practice. Many researchers study heuristics, techniques that tend to improve running time in practice but that often do not have any theoretical performance guarantees (we will study these in detail in Chapter 13).

Another popular approach due to its emphasis on mathematical rigor is to investigate approximation algorithms. An approximation algorithm is a polynomial-time algorithm that delivers a solution\(^{13}\) that is provably close to the correct, or “optimal” solution. The closer we can guarantee our solution will be to the optimal solution, the better our approximation algorithm. The tricky aspect of this, of course, lies in proving that a solution is close to an optimal solution we do not know or have the means to compute.

For an example, recall that the NP-hard satisfiability problem (SAT) involves finding an assignment of values to Boolean variables that satisfies some given Boolean expression. After a bit of simple algebra [Details], any Boolean expression can be written as an equivalent expression in conjunctive normal form (CNF), where we have a series of “OR” clauses that are all “ANDed” together:

\[
\frac{(x_2 \lor \overline{x}_3 \lor x_4)}{\text{clause 1}} \land \frac{(\overline{x}_1 \lor x_3)}{\text{clause 2}} \land \frac{x_2}{\text{clause 3}} \land \frac{(x_1 \lor \overline{x}_2 \lor \overline{x}_4)}{\text{clause 4}}
\]

To satisfy the original expression, we need to find an assignment of values to variables that satisfies all of these clauses. However, let us try instead to find an assignment that satisfies the greatest number of clauses. This is known as the MAX-SAT problem and it is NP-hard since it can be used to solve SAT (observe

\[^{13}\text{We should mention the following to avoid potential confusion: for many people, the word “solution” might imply a “correct” or “optimal” answer to a problem. However, when we discuss algorithms (especially approximation algorithms), we often also use the word “solution” to mean any feasible, but not necessarily optimal answer to a problem.}\]
that we just described a simple reduction from SAT to MAX-SAT). A trivial approximation algorithm for MAX-SAT is to take the better of two solutions, one with all variables set to true, and the other with all variables set to false. Since each clause must evaluate to true in at least one of these solutions, one of the two solutions must satisfy at least half the clauses. Alternatively, if we randomly set each variable by flipping a fair coin, then one can easily show that this satisfies at least half the clauses in expectation. Since the optimal solution can at best satisfy all of the clauses, we say our algorithm is a 1/2-approximation algorithm since it gives a solution whose value is least 1/2 that of the optimal solution. The value 1/2 is called the performance guarantee of the algorithm, since it measures relative closeness to the optimal solution value. Some texts would say this algorithm has a performance guarantee of 2, since the solution generated by the algorithm is at most a factor of 2 away from the optimal solution. However, we adopt the popular convention that for maximization problems like MAX-SAT, the performance guarantee will be less than one, whereas for minimization problems it will be more than one. That is, a 1/2-approximation delivers a solution that is at least 1/2 optimal for a maximization problem, and a 2-approximation algorithm delivers a solution no worse than twice optimal for a minimization problem.

Approximation algorithms apply not only to NP-hard problems, but also to problems solvable in polynomial time for which the best known exact algorithms are too slow in practice. They range from very simple techniques like the ones above to much more complicated methods. The best-known approximation algorithm for MAX-SAT has a performance guarantee of roughly 0.77 and requires some very fancy mathematics to achieve this feat!

The Polynomial-Time Approximation Scheme. For some NP-hard problems we can achieve the ultimate approximation result: a polynomial-time approximation scheme, or PTAS. A PTAS is a polynomial-time algorithm that delivers a \((1 - \varepsilon)\)-approximate solution for any constant \(\varepsilon > 0\) of our choosing (a \((1 + \varepsilon)\)-approximate solution, for minimization problems), so it allows us to approximate the optimal solution of a problem to arbitrary relative precision in polynomial time. Unfortunately, most PTAS algorithms have terribly impractical running times like \(O(n^{1/\varepsilon})\). Such an algorithm, if used to find a solution within 1% of optimal, would carry a running time of \(O(n^{100})\). While such algorithms may be of little use in practice, they are still important in theory because they help us establish the intrinsic hardness of a problem. One does find the occasional PTAS with a running time that is reasonable in practice. For example, in Section 11.4 we discuss a PTAS for the 0/1 knapsack problem that runs in only \(O(\frac{n}{\varepsilon} \log n)\) time. If the running time of a PTAS has only a polynomial rather than exponential dependence on \(1/\varepsilon\), as in this case, then we call it a fully polynomial time approximation scheme (FPTAS).

Inapproximability Results. NP-hard problems that admit an FPTAS are in some sense the “easiest” of the NP-hard problems. However, not all problems admit an FPTAS, or even a PTAS. We can often can prove inapproximability results stating that it is NP-hard even to approximate the optimal solution of some problem to within some particular factor. Based on these results, we can partition the class of NP-hard problems into subclasses based on approximability. For example, the class of strongly NP-hard problems contains problems that cannot admit an FPTAS unless \(P = NP\), and the class of MAXSNP-hard problems contains problems that cannot admit a PTAS unless \(P = NP\). Further details can be found in the endnotes,

1.7. DIFFERENT FLAVORS OF ALGORITHMS
since we do not wish to inundate the reader with too much new terminology all at once. Inapproximability proofs range from being quite straightforward to those requiring some of the most advanced techniques known to the theoretical computer science community. [Simple example: bin packing is NP-hard to approximate to within better than a factor of $3/2$]

**Problem 4 (Impossible Inequalities).** Suppose you are told that three different numbers $a$, $b$, and $c$ should satisfy $a < b$, $b < c$, and $c < a$. It should be clear that it is impossible to assign values to $a$, $b$, and $c$ so that all three of these inequalities are satisfied. The best we can do is satisfy two of the three. More generally, suppose we want to assign distinct values to $n$ variables so as to satisfy the greatest possible number out of a set of $m$ strict inequalities, each involving two of the variables. Please suggest a simple $1/2$-approximation algorithm for this problem. It is interesting to note that no approximation algorithm with a better performance guarantee is currently known! [Solution]

### 1.7.3 Online Algorithms

An online algorithm does not have the opportunity to see its entire input before it must start making decisions. Online problems appear in many areas of practice, from routing to cache management to load balancing. The classical example of an online problem is the *buy or rent problem*: suppose whenever you go ice skating you have the option of renting a pair of skates for $5$ or buying a pair of skates for $100$. However, you do not know in advance how many times you will end up going skating. It would not make sense to buy skates if you only go skating once or twice, nor would it make sense to rent skates if you plan on making hundreds of visits to the ice rink. If you end up making $n$ trips to the rink, the optimal solution is to rent if $n < 20$ and to buy otherwise. Unfortunately, you don’t know $n$ in advance. Let us therefore adopt the following strategy: for the first 19 trips to the rink we will rent, and on the 20th trip we buy. It is not hard to see that regardless of what $n$ turns out to be, we will spend at most twice as much as the optimal solution. We therefore say that our strategy is 2-competitive, illustrating the use of competitive analysis as a popular way to characterize the performance of an online algorithm. The competitive ratio of an online algorithm is somewhat like the performance guarantee of an approximation algorithm. It specifies the relative distance, in the worst case, from the solution produced by the online algorithm to the optimal solution we could have computed in the “offline” case where we know the entire input in advance.

**Problem 5 (CPU Power Management).** Suppose a CPU can switch between two levels of power consumption. Any time it needs to process a computationally-intensive task, it needs to be in “high” power mode, where it consumes $r_h$ units of energy per unit time. If there is no computationally-intensive task pending, it can switch down to “low” power mode, where it consumes only $r_l < r_h$ units of energy per unit time. It costs $r_{hl}$ units of energy to switch from high to low power mode, and $r_{lh}$ units of energy to switch from low to high power mode. Since we do not know when computationally-intensive tasks will be arriving in the future, we would like to design a power management strategy that achieves the lowest possible competitive ratio against an optimal strategy that knows the future. What is the lowest possible competitive ratio, and what strategy achieves it? [Solution]
Problem 6 (Dynamic TCP Acknowledgement). The transmission control protocol (TCP) is one of the primary protocols used for transmitting information on the Internet. TCP provides a mechanism for breaking up a communication session between two computers $A$ and $B$ into a stream of packets. Suppose we focus on the packets that $A$ is sending to $B$. According to TCP, $B$ should occasionally send “acknowledgement” packets back to $A$ specifying which packets $B$ has already received. Since packets tend to be numbered sequentially, $B$ can acknowledge a consecutive range of packets from $A$ using only a single acknowledgement packet, and this is preferable since it avoids the congestion that would be incurred if $B$ sent out an acknowledgement for every single packet it receives. However, if $B$ waits too long before sending out an acknowledgement, $A$ may think that there is excess congestion in the network, and it may slow down its transmission of future packets. A reasonable objective in this case is to have $B$ minimize the sum $a + d$, where $a$ is the number of acknowledgement packets it sends out, and $d$ is the total acknowledgement delay (sum over all incoming packets of the time between arrival of the packet and departure of its corresponding acknowledgement packet). Since $B$ does not know when packets from $A$ will be arriving in the future, it cannot hope to optimally minimize this objective. However, please show a simple acknowledgement policy for $B$ that is 2-competitive. [Solution]

Problem 7 (Searching the Number Line). You are invited to a dinner party at a friend’s house, but your friend neglected to mention his address — he only told you the name of his street and a description of his house. Starting from some point on his street, we would like to walk the minimum possible distance before discovering his house. We can think of this problem more abstractly as follows: we start at the origin on a number line and would like to walk the minimum possible total distance to a point whose position $x$ is not known to us. If it were known, we would be able to walk a distance of just $|x|$. Not knowing $x$, however, consider the strategy of walking back and forth, each time venturing twice as far from the origin on the other side. See if you can obtain upper and lower bounds of 9 on the competitive ratio of this approach. [Solution]

1.7.4 Parallel and Distributed Algorithms

In this book we do not talk much about parallel computation since there is already so much to say about sequential algorithms, and since it is important to be well-grounded in the basics of sequential computation before one attempts to tackle parallel computation. However, parallel computation is rapidly emerging as an area of key importance. At the time of writing of this book, affordable parallelism is becoming available to the masses through multi-core CPUs as well as specialized parallel processors such as graphics processing units (GPUs). In the realm of distributed computing, large clusters of networked computers (sometimes called grids or clouds) are often used in many areas of research and industry for solving a host of important “big data” problems.

Approaches for parallel computation differ slightly depending on whether we have a “fine-grained” parallel computing environment with multiple small processors all wired together in a specific layout and working in tight synchronization, or a “coarse-grained” distributed computing environment composed of multiple types of independent computers that can exchange messages on a common network. In the first case, one tends to focus more on the exact layout we use to connect the processors (e.g., a two-dimensional grid) to obtain the best data flow. In the second case, we may worry more about developing robust algorithms that are tolerant of faults due to network connectivity outages or individual computer failures.
Models of parallel and distributed computation can be quite a bit more complicated than sequential models. The most straightforward parallel generalization of our RAM model is known as a parallel RAM or PRAM, and consists of multiple processors operating in parallel with a single shared memory. The tricky issue here of is what happens when two processors decide to access the same memory location at the same time, and there are several variants of the model based on how such collisions are resolved. Other models of parallel computation usually involve a distributed collection of independent processors, each with their own local storage, connected together in some sort of fixed topology like a mesh or a ring. There are many possible variations on these models depending on the topology of the network, whether or not computation is synchronous (all processors exchanging messages in a synchronized step-by-step fashion according to a global clock), whether or not processors can fail (or even worse, fail and output garbage), and so on.

Parallel computation is an important area of computer science because it investigates in some sense the true limits of computing devices subject only to the constraints of physics. There are quite a few algorithmic challenges to overcome when designing parallel algorithms, because many problems seem to be inherently non-parallelizable. Just because we have \( n \) processors available certainly does not mean we can speed up an algorithm by a factor of \( n \). As an example, consider the simple problem of adding two \( n \)-bit binary numbers. We can do this sequentially in \( O(n) \) time using the algorithm for addition we all learned in elementary school: add one digit at a time, propagating a “carry” bit along the way if needed. However, if we have all the processors in the world, it is not immediately clear how one would perform this computation any faster! Fortunately, there does exist a clever parallel approach called carry lookahead addition that uses \( O(n) \) processors to add two \( n \)-bit numbers in only \( O(\log n) \) time. [Details]

A factor of \( n \) is the best speedup we could hope to achieve over a sequential algorithm if we throw \( n \) parallel processors at a problem. Therefore, we would need an exponential number of processors to save us from the seemingly unavoidable exponential running times required to solve NP-hard problems. As problem sizes increase, the amount of physical space required for this many processors quickly becomes unreasonable. If our environment requires \( 2^n \) processors, then even for...
moderate values of $n$ we would need more processors than there are atoms in the planet Earth! The only possibility that currently seems to offer some promise for potentially solving NP-hard problems quickly is *quantum computation*, a field that is now in its infancy, but which may offer exciting prospects for highly-efficient computation in the future. Quantum computers operate based on principles of quantum mechanics to achieve what is effectively an exponential amount of parallelism. Although research on quantum computing hardware is progressing, nobody has yet managed to build a general-purpose quantum computer. Of course, this has not stopped researchers from studying the theory of quantum algorithms. The most famous result in this area is probably Shor’s algorithm for factoring an $n$-bit integer in polynomial time on a quantum computer. Factoring is currently not known to be NP-hard, although many researchers suspect that it cannot be solved in polynomial time on a standard RAM (this is a question of crucial importance, since widely-used cryptosystems like RSA rely on the intrinsic hardness of factoring).

### 1.7.5 Memory-Constrained Algorithms

Many problems in modern computing involve an overwhelming amount of data — more than can fit in memory at one point in time. For example, suppose we need to find the median element from an enormous un-ordered set of $n$ numbers much larger than the memory of our computer, stored externally on read-only magnetic tape. Here, the input is so large that we can only process a small part of it at a time. In Chapter 3, we will see how to solve this problem with a randomized algorithm that makes a small number of passes over the tape. This is sometimes referred to as a *stream* model of computation, where we can only see a small window into the input as it streams by one or more times (we will see several algorithms for this model in Section 7.5). There are many computational situations in which we encounter massive data sets (e.g., the World Wide Web) from which an algorithm cannot hope to see more than a small piece at a time due to limited local storage. However, memory-constrained environments do not necessarily require massive data sets; for example, we may want to design algorithms for small inexpensive “embedded” microprocessors that have very limited memory.

Memory access time is actually the main bottleneck in many data structures and memory-intensive algorithms, since a single memory access can take orders of magnitude more time than a single arithmetic operation within the CPU. This is a notable weakness in our RAM model of computation, which assumes that every memory access takes “one unit of time”, just as with any other operation. For the reader who is implementation-oriented, this point is quite important to keep in mind while reading this book. Even if two algorithms share the same asymptotic running time, one can run substantially faster than the other in practice if it is more carefully tuned for fast memory access.

Due to the way computer memories are designed, algorithms like binary search that jump around wildly through memory will run much slower than algorithms that perform the same number of memory accesses in a more highly localized fashion. Figure 1.11(a) depicts the hierarchical layers in the memory layout of a typical computing device. Between the CPU and main memory sits a memory “cache” that is much smaller than main memory but also much faster to access. Memory is divided into “pages” of size $B$ words (typically $B$ is around 1K), and our size-$M$
cache holds $M/B$ of these. When a page in memory is accessed that does not reside in the cache, the entire page is transferred to the cache in one step\textsuperscript{14}. Subsequent accesses to nearby memory locations are therefore much faster, since they reside in the cache. This sort of fast block transfer happens between most layers in memory — disk to memory, memory to cache, etc.

**Cache-Oblivious Algorithms.** For a memory-intensive algorithm, it is entirely reasonable to use the number of page transfers as the sole measurement of performance. Let $OPT(M, B)$ denote the minimum possible number of page transfers required in the worst case to solve a problem on a two-level memory system (idealized cache plus main memory) with parameters $M$ (total cache size) and $B$ (page size). If an algorithm knows $M$ and $B$, it can carefully tune its performance to try and achieve this goal. However, $M$ and $B$ are often hard to know, being system-level parameters. A popular model of computation known as the *cache-oblivious* model suggests a nice concept in this situation: we say that an algorithm is cache-oblivious if it performs only $O(OPT(M, B))$ transfers on an idealized cache, despite the fact that the algorithm does not know $M$ or $B$. Since this holds for every pair of adjacent layers in our memory hierarchy (even if they involve different values for the parameters $M$ and $B$), a cache-oblivious algorithm therefore incurs at worst a constant-factor slowdown across the entire memory hierarchy compared to what could be achieved by an optimally-tuned algorithm.

Designing cache-oblivious versions of even common algorithms (e.g., sorting algorithms) can be an interesting challenge. For example, even the standard binary search algorithm is not cache-oblivious, since it requires $O(\log(n/B)) = O(\log n - \log B)$ transfers, whereas $OPT(M, B) = O(\log_B n) = O(\log n/\log B)$ for the problem of searching in set of $n$ elements (if they are organized appropriately in memory). In Section 7.4.3, we will see an elegant “stratified tree” data structure that effectively does provide us with a cache-oblivious version of binary search.

### 1.8 Closing Remarks

We have seen quite a few definitions and several very deep concepts introduced in this chapter. To the introductory reader: do not worry, these concepts will become clearer over time as we put them to use in the upcoming chapters. In the next chapter, we will acquire some useful mathematical tools, and then in Chapter 3 we will embark on a study of algorithms for sorting, during which we will also spend some time discussing how to formally prove correctness of an algorithm and analyze its running time. The reader is also encouraged to consult the endnotes for further discussion, references, and historical details.

\[\textsuperscript{14}\text{When a new page enters the cache, an old page must be evicted. Common policies for page replacement are to evict the least recently used (LRU) page, or to replace pages on a first-in-first-out (FIFO) basis. The policy for an ideal cache would be to replace the page whose next access will be farthest in ahead in the future, although there is no way the cache can know which page this is. Fortunately, one can show that if an algorithm only slows down by a constant factor on an idealized cache when the cache size $M$ is halved, then its running time will only slow down by a constant factor using either LRU or FIFO page replacement as well. We therefore often assume our cache uses an ideal page replacement policy for simplicity. See problem 168 for further detail.}\]
2. Useful Mathematical Concepts

Just as you can follow a recipe from a cookbook without really learning to cook, it is certainly possible to study algorithms while avoiding mathematics and proofs. This may, in fact, be preferable for the practitioner who simply wants help implementing fundamental algorithms. However, our goal in this text is to learn how to cook, so to speak. A sound mathematical background is a fundamental prerequisite for success in the design and analysis of novel algorithms for novel problems.

Algorithmic study is well-suited for those who appreciate mathematics, and can help develop such an appreciation for those with limited prior mathematical exposure, since can learn much about mathematics as a side benefit of studying algorithms. We assume the reader has some background in discrete mathematics, but we stress that most of the techniques we use should be quite accessible even for those without extensive training. Algorithmic analysis mostly entails clever application of basic mathematical concepts, rather than deep results from advanced mathematics.

In this chapter, we review mathematical techniques used throughout the rest of the book. We begin with basic notation and terminology, proof techniques, and helpful “tricks”. Next, we cover three topics in greater detail that we will find particularly useful: methods for solving recursively-defined expressions called recurrences (to analyze recursive algorithms), techniques from probability theory (to analyze randomized algorithms), and important concepts from linear algebra (to analyze some of the more advanced mathematical algorithms in this book). Finally, we highlight a few remaining bits of prerequisite knowledge that may benefit the reader from other areas of mathematical study. The reader with a relatively strong mathematical background is still advised to peruse this chapter, since it includes discussion and problems covering a number of interesting algorithmic applications.

2.1 Notation, Definitions, and Assumptions

Our notation is fairly standard and should not differ appreciably from any other mathematical text.

Sets. We expect the reader to be familiar with standard notation for manipulating sets. For example, \( S \cap T \) denotes the intersection of sets \( S \) and \( T \), and \( S \cup T \) denotes
their union. There are several “formulaic” ways to prove certain types of statements regarding sets. We can prove that $S = T$ by arguing that $S \subseteq T$ and also $T \subseteq S$. To show that $x \in S \cap T$, we could prove separately that $x \in S$ and $x \in T$. To argue that $S \subseteq T$, we might take an arbitrary element $x \in S$ and show that $x \in T$ as well. As a shorthand for summing over a set, we sometimes use a set-valued argument to a function:

$$f(S) = \sum_{x \in S} f(x) \quad \text{and} \quad g(S, T) = \sum_{x \in S, y \in T} g(x, y).$$

We often use interval notation to describe a contiguous set of numbers. For example, $[0, 1]$ denotes the closed set of all real numbers between 0 and 1, with both endpoints included in the set, and $[5, 10)$, denotes the set of all real numbers between 5 and 10, including the endpoint 5 but not 10. The Cartesian product of two sets, $A \times B$, is the set of all pairs $(a, b)$ where $a \in A$ and $b \in B$. For example, $[0, 1] \times [0, 1]$ is the unit square in the 2D plane with corners $(0, 0)$ and $(1, 1)$.

**Logarithms.** Logs show up everywhere in the study of algorithms. As with binary search, many algorithms successively reduce a problem of size $n$ to a smaller equivalent problem of size at most $n/k$ (for binary search, $k = 2$). It takes $\log_k n$ such phases to reduce a problem of size $n$ down to a trivial problem of size 1. By $\log n$ we mean $\log_2 n$ by default; we write $\ln n$ for logarithms in base $e$. The base of a logarithm usually doesn’t matter in an asymptotic running time expression, since logs of different bases differ only by constant factors. The reader should be comfortable manipulating expressions involving logarithms. For example, $\log n^k = k \log n$, $2^{\log n} = n$, and $a^{b \log b} = b \log a$.

**Sequences.** We use subscripts to write a sequence of elements as $A_1, A_2, \ldots, A_n$. If the elements represent characters drawn from some alphabet, we sometimes call the sequence a string. The words list and array can also refer to a sequence, but we typically use these only for sequences that are specifically implemented as linked lists or arrays. We use the notation $A[1], A[2], \ldots, A[n]$ for the elements of a sequence stored in an array in memory. Our sequences typically start at index 1 instead of index 0. Finally, a substring or subarray refers to a contiguous block of elements $A_1 \ldots A_j$ within a sequence/string/array, while a subsequence refers to a subset of the elements of the sequence (taken in the same order as they appear in the sequence), which may not be a single contiguous block.

Sequence $S$ is lexicographically smaller than another sequence $S'$ if $S_j < S'_j$ at the first index $j$ where they disagree, or if there is no disagreement but $S$ is shorter than $S'$. Lexicographic ordering is a natural way to order objects that have multiple components, like sequences, strings, and vectors. For text strings, lexicographic ordering corresponds to our familiar notion of alphabetic ordering.

**Number Bases.** We expect the reader to be familiar with the notion of different number bases. In particular, since digital computers naturally represent all numbers in binary (base 2), the reader should be comfortable with expressing numbers in binary. For example, we would write the base-10 number 19 in binary as 10011, since $19 = 16(1) + 8(0) + 4(0) + 2(1) + 1(1)$. In any number base, we call the leftmost digit of a number the most significant digit and the rightmost digit the least significant digit.

**Factorials, Permutations, and Combinations.** The number of permutations (different orderings) of $n$ distinct elements is $n! = 1 \times 2 \times 3 \times \ldots \times n$. The number
of different subsets of \( k \) elements we can choose from a set of \( n \) distinct elements is given by the binomial coefficient \( \binom{n}{k} \), where

\[
\binom{n}{k} = \frac{n!}{k!(n-k)!} \leq n^k
\]

The term \( \binom{n}{k} \) is read “\( n \) choose \( k \)” and is sometimes written elsewhere as \( C(n,k) \) or \( nC_k \). For example, if we have an algorithm that examines every pair of elements in an \( n \)-element array, it will examine \( \binom{n}{2} = n(n-1)/2 \leq n^2 \) pairs of elements.

### 2.1.1 Floors, Ceilings, and Other Pesky Details

In the literature on algorithms, it is customary to allow for a bit of “sloppiness” in certain aspects of our notation. This lets us avoid minor, irrelevant details in order to explain concepts more clearly at a high level. For example, we might say that binary search looks at the middle element of an array, \( A[n/2] \), and then decides to search within only the first half or second half of \( A \). If our array has an odd number of elements, then \( n/2 \) will not be an integer, so we should technically say \( \lfloor n/2 \rfloor \) or \( \lceil n/2 \rceil \). Similarly, the first “half” of the array might not contain exactly half the elements. However, thanks to our use of asymptotic notation, details like this rarely make a difference in our final asymptotic running time. Hence, for simplicity we tend to say things like “\( n/2 \)” and “half” knowing that there may actually be an implicit floor or ceiling required.

To give some more examples, suppose we say that “our algorithm partitions an input of size \( n \) into \( \sqrt{n} \) blocks each of size \( \sqrt{n} \)” or “our algorithm runs in \( \log n \) phases”. In the first case, if \( n \) is not a perfect square, this implicitly means that our blocks may have size \( \lfloor \sqrt{n} \rfloor \) or \( \lceil \sqrt{n} \rceil \) and that the last “leftover” block may be smaller than the rest. In the second case, if \( n \) is not a power of 2 then perhaps there will actually be \( \lfloor \log_2 n \rfloor \) full phases and one left-over partial phase. Presumably, these are minor details, straightforward to figure out during implementation, and irrelevant to our final asymptotic running time. If desired, we can often avoid the need for special cases like those above by adding “dummy” elements to our input (say, to increase the size to an exact power of 2). Since this usually only inflates the input size by a constant factor, it rarely changes our asymptotic running time.

### 2.1.2 Asymptotic Notation

Back in Section 1.4, we introduced asymptotic notation and motivated its importance in describing algorithmic running times. We learned that:

- An expression like the running time of an algorithm is said to be \( O(f(n)) \) if it is bounded above by some constant times \( f(n) \) as \( n \) grows sufficiently large. For example, \( \sqrt{n} \), \( n^2 \), and \( 1000n^2 + n \) are all \( O(n^2) \).
- An expression is \( \Omega(f(n)) \) if it is bounded below by a constant times \( f(n) \) for sufficiently large \( n \). For example, \( n^2 - 10n + 2 \), \( n^3 \), and \( 2^n \) are all \( \Omega(n^2) \).
- An expression is \( \Theta(f(n)) \) if it is both \( O(f(n)) \) and \( \Omega(f(n)) \). Since it includes both an upper and lower bound, a \( \Theta(\cdot) \) bound gives us a precise characteriza-
tion of the rate of growth (up to constant factors) of an algorithm’s running time. For example, \(n^2, 3n^2\) and \(7n^2 - 5n\) are all \(\Theta(n^2)\).

We can round out this list with the following additional definitions:

- An expression \(f(n)\) is said to be \(o(g(n))\) if the limit of \(f(n)/g(n)\) goes to zero as \(n\) grows sufficiently large. Here, \(f\) grows “strictly” slower than \(g\), so this is a stronger statement than saying \(f(n)\) is \(O(g(n))\). For example, \(3n^2\) is \(o(n^3)\).

- An expression \(f(n)\) is said to be \(\omega(g(n))\) if the limit of \(f(n)/g(n)\) goes to infinity as \(n\) grows sufficiently large. Here, \(f\) grows “strictly” faster than \(g\), so this is a stronger statement than saying \(f(n)\) is \(\Omega(g(n))\). For example, \(n^2.1\) is \(\omega(n^2)\).

Informally, one should think of \(O, \Omega, \Theta, o,\) and \(\omega\) as the equivalents of \(\leq, \geq, =, <,\) and \(>\) when comparing the asymptotic growth of functions.

We often use asymptotic expressions as “placeholders”. For example, we might write a polynomial as \(7n^3 - 3n^2 + o(n^2)\) where the \(o(n^2)\) serves as a placeholder to indicate the presence of lower order terms. Please be careful when performing algebra involving these asymptotic placeholders. If we perform \(\Theta(n^2)\) iterations of an algorithm whose running time is \(O(n^7)\), we can rightfully multiply these to obtain a total running time of \(O(n^9)\). However, it does not make sense to divide both sides of an equation by \(o(n^2)\). When in doubt, go back to the definitions above to check if a particular arithmetic manipulation makes sense.

### 2.1.3 Special Functions

On occasion, we will encounter two special functions that grow extremely slowly, much slower even than \(\log n\). These functions are generally excellent news in terms of efficiency, since they give us very small running times even for huge values of \(n\).

**The \(\log^* n\) Function.** Imagine that you have a calculator with an “\(f(n)\)” key, so for example pressing the key three times would transform \(n\) into \(f(f(f(n)))\). Assuming \(f\) is decreasing, we define \(f^*(n)\) as the number of key presses needed to decrease \(n\) down to a value that is at most 1. For example, if \(f(n) = n - 2\), then \(f^*(n) = \lceil n/2 \rceil\), since \(n/2\) subtractions of 2 are needed. If \(f(n) = n/2\), then \(f^*(n) = \lceil \log n \rceil\). This “star” operator is important when dealing with recursively-defined algorithms, since if an algorithm solves a problem of size \(n\) by recursively solving subproblems of size \(f(n)\), then the algorithm will recurse for \(f^*(n)\) levels. For example, each step of binary search reduces a problem of size \(n\) to a recursive subproblem of size \(f(n) = n/2\), so binary search recurses to a depth of \(f^*(n) = O(\log n)\).

The star operator applied to \(f(n) = \log n\) gives the function \(\log^* n\), one of the slowest-growing functions we occasionally encounter in the analysis of algorithms. To give you a sense of just how slowly this function grows, we have \(\log^* n = 5\) if \(n = 2^{65536}\), a number much larger than the number of atoms in the observable universe. For all practical purposes, we can think of \(\log^* n\) as being constant, although from a theoretical standpoint we cannot simply ignore such terms since they do grow as functions of \(n\).
Problem 8 (Practice Applying the Star Operator). If \( f(n) = \sqrt{n} \), give a simple proof that \( f^*(n) = \Theta(\log \log n) \). Repeated square roots never decrease below one, so please stop at 2 instead of 1 in the definition of \( f^* \) (typically any constant is fine, as long as we only care about the asymptotic form of the answer). [Solution]

Problem 9 (The Logsum Function, Binary Codes, and Unbounded Search).
The function \( \logsum(n) = \log n + \log \log n + \log \log \log n + \ldots \) (with \( \log^* n \) terms) plays an important role in several algorithmic results. For example, suppose we want to encode in a single binary stream a sequence of positive integers, with no bounds on their sizes. We could of course write each number in binary, but we also need to encode information on the number of bits we will use, since otherwise we won’t know where one number stops and another starts. The clever Elias “Omega” code uses \( \logsum(n) + O(\log^* n) \) bits to encode \( n \) by writing \( n \) in binary after recursively encoding \( \lceil \log n \rceil \), the number of bits required for \( n \) (so we encode \( \lceil \log n \rceil \) in binary using \( \lceil \log \log n \rceil \) bits, before which we write its length \( \lceil \log \log \log n \rceil \) bits, and so on). Inspired by this approach, consider now the related problem of unbounded search for a positive integer \( n \), where we are given no upper bound on \( n \). A simple approach taking at most \( 2 \lceil \log n \rceil \) comparisons to find \( n \) is to start with \( x = 1 \) and successively double \( x \) until \( x \geq n \) (taking \( \lceil \log n \rceil \) steps), then to binary search the range \([1, x]\) for \( n \) (taking another \( \lceil \log n \rceil \) steps). For a challenge, can you show how to improve on the first half of the algorithm in order to perform unbounded search in only \( \logsum(n) + O(\log^* n) \) comparisons? As a hint, \( x \) is just a power of two, so recursively search for \( \log x \). How is the unbounded search problem essentially equivalent to the binary encoding problem above? [Solution]

The Inverse Ackermann Function. Designing a function that grows even more slowly than \( \log^* n \) is as simple as just adding more stars. For example, \( \log^{**} n \) is the number of times we need to iteratively apply the \( \log^* n \) function to \( n \) before we reach a value no larger than 1. For \( n > 4 \), the value of \( \log^{***} n \) is at least 3 no matter how many stars we use, and if we add enough stars, we will eventually reach a point where \( \log^{**} n = 3 \). We define the inverse Ackermann function, \( \alpha(n) \), as one plus the minimum number of stars required to make \( \log^{***} n \leq 3 \) (we add one to ensure that \( \alpha(n) \) is always at least positive). On occasion, we will use a stronger two-term variant \( \alpha(m, n) \leq \alpha(n) \), giving one plus the minimum number of stars required to make \( \log^{***} n \leq 3 + m/n \). One can show that \( \alpha(n) = o(\log^{**} n) \) for any constant number of stars, so inverse Ackermann running time bounds are even faster than those involving multiply-iterated logs. Almost every appearance of an inverse Ackermann running time originates from just one place — the analysis of a prominent data structure for maintaining disjoint sets, discussed in Section 4.6.

The Ackermann function is a famous recursively-defined function designed to grow extremely quickly. Many variants of the Ackermann function appear in the literature, and consequently one can also find different ways of defining inverse Ackermann functions, all of which lead to essentially the same asymptotic behavior. We don’t use the Ackermann function directly in this book, so we won’t say any more about it here in the main text. However, we have included further detail in the endnotes, for the interested reader.

2.1.4 Graphs and Trees

Graphs are particularly prominent objects in computer science, due to their flexibility in being able to model so many different things. Every graph consists of a set
of nodes (sometimes called vertices or points) and edges (sometimes called links or lines), where each edge connects a pair of nodes. Depending on our physical interpretation of nodes and edges, a graph can represent virtually any kind of network (transportation, communication, social, etc.), as well as a more abstract object like a mathematical relation.

Graphs can be undirected or directed. The word “graph” technically implies an undirected graph, while “digraph” specifies a directed graph; we sometimes use the term “graph” generically to refer to both. An edge directed from node $i$ to node $j$ in a directed graph (sometimes called an arc) is written as an ordered pair $(i,j)$, while an undirected edge between $i$ and $j$ in a graph is written as a set $\{i,j\}$, since this has no directional implications. The notation $ij$ is also sometimes used for describing either a directed or undirected edge from $i$ to $j$. We usually assume no edge is a self loop — connecting a node to itself. We generally also assume that our graphs are simple, meaning there is at most one edge $ij$ for any pair of nodes $(i,j)$. Otherwise, if multiple “parallel” edges can run between the same pair of nodes, we have a multigraph. The degree of a node is the number of edges incident to the node. In a directed graph, we distinguish indegree (number of incoming edges) and outdegree (number of outgoing edges). We often consider weighted graphs by associating numeric “weights” with edges and/or nodes. Depending on the application, these may have physical meanings as costs, values, capacities, lengths, and so on.

By convention, $n$ represents the number of nodes in a graph, and $m$ represents the number of edges. A graph with few edges (where $m$ is closer to $n$) is said to be sparse, while a graph with many edges (where $m$ is closer to $n^2$) is said to be dense. Some algorithms are better suited to sparse graphs, and others dense graphs. In practice, most large graphs (e.g., the directed graph depicting the link structure of the World Wide Web) tend to be quite sparse. The running time of a graph algorithm will usually depend on both $n$ and $m$. Since simple graphs satisfy

---

Figure 2.1: Examples of: (a) a connected graph with a spanning tree highlighted, (b) a graph with two connected components, (c) a directed graph where there is no directed path from $i$ to $j$, but there is a directed path from $j$ to $i$, (d) a rooted tree of height 2.
2.1. NOTATION, DEFINITIONS, AND ASSUMPTIONS

$m \leq n^2$, we can technically replace $m$ with $O(n^2)$ to write these running times solely in terms of $n$. However, this is generally not a good idea, since in a sparse graph, an $O(m)$ running time is more like $O(n)$ than $O(n^2)$, and we might discourage someone from using a fast $O(m)$ algorithm if we describe its running time solely as $O(n^2)$.

Paths and Cycles. A path is a series of nodes connected by edges; we can regard this either as a sequence of nodes or as a sequence of edges. A path that starts and ends at the same node is called a cycle. In a directed graph, paths and cycles contain edges that are consistently directed. A path is simple if it contains no cycles, and a cycle is simple if it contains no cycles shorter than itself. When we say “path” and “cycle” in this book, we mean a simple path or cycle. We use the term walk for a path that need not be simple. The length of a path or cycle is the number of edges it contains (or in a weighted graph, the sum of its edge weights). A Hamiltonian path or cycle visits every node exactly once, and an Eulerian path or cycle follows every edge exactly once.

Connectivity. Nodes $i$ and $j$ are adjacent if they are directly connected by an edge, and connected if they are connected by a path. In a directed graph, there can possibly be an edge or path from $i$ to $j$ but not in the reverse direction. A graph is connected if every pair of nodes is connected; for example, Figure 2.1(a) is connected while Figure 2.1(b) consists of two separate connected components. In Chapter ?? we extend the notion of connectivity and connected components to directed graphs, as well as higher orders of connectivity (e.g., we can say that two nodes are highly connected if they are connected by a large number of edge-disjoint paths, or alternatively if we need to remove a large number of edges to separate them into distinct connected components).

Trees. A tree is a graph that is connected and acyclic (having no cycles). Often we designate a specific node in a tree as a root, allowing us to orient the tree so that every node has a well-defined parent (except the root) and set of children, as shown in Figure 2.1(d). A tree with no designated root is called a free tree; the word “tree” by itself can mean either a free tree or a rooted tree. We will sometimes take a free tree and “root it”, designating one of its nodes as a root, after which we can then operate on it like a rooted tree. Rooted trees play a key role in most data structures, as we shall see in Chapters 4 through 9. As opposed to reality, rooted trees in computer science are usually drawn growing downward from a root at the top. Accordingly, the depth of a node in a rooted tree is its distance downward from the root, and the height of a rooted tree is the maximum depth over all nodes. Node $i$ is an ancestor of node $j$ in a rooted tree if $j$ appears inside the subtree rooted at $i$ (so $i$ lies on the path from $j$ up to the root). A node of degree 1 (in a rooted tree, a node with no children) is called a leaf. Other nodes are called internal nodes.

Subgraphs and Spanning Trees. A subset of the edges of a graph is called a subgraph. A subset of nodes induces a subgraph consisting of all edges with both endpoints in the subset. A particularly common subgraph in algorithmic computing is a spanning tree — a subset of edges that connects together all nodes and contains no cycles, as shown in Figure 2.1(a). For example, a spanning tree in a communication network gives us a minimal subset of edges within which we can still route information between all pairs of nodes. This simplifies the routing task as well, since within any tree there is a unique path joining any given pair of nodes. In the case where we want to broadcast information from one node $r$ to all other
nodes, we end up with a tree rooted at $r$. Such trees are often thought of as being “directed” outward from the root $r$. In Chapter ?? we will more formally study directed notions of trees, known as branchings and arborescences.

**Representing a Graph in Memory.** Two common ways to represent a graph are using an *adjacency matrix* or with *adjacency lists*. As shown in Figure 2.2(a), the adjacency matrix is an $n \times n$ matrix in which the $(i, j)$ entry is 1 if there is an edge from $i$ to $j$, and 0 otherwise. For an undirected graph, the matrix is symmetric since the $(i, j)$ and $(j, i)$ entries are equal. Adjacency matrices are well-suited for dense graphs; otherwise, they can be a liability both in terms of space and running time. A graph algorithm utilizing adjacency matrices typically cannot have a better running time than $\Theta(n^2)$ due to the need to examine the entire input matrix. Furthermore, although adjacency matrices allow us to query whether an edge $(i, j)$ is present in only $O(1)$ time, they require $\Theta(n)$ time to enumerate the neighbors of a node, regardless of how many neighbors there are.

Figure 2.2(b) shows a graph represented by adjacency lists, where we maintain for every node $i$ an array or linked list of the neighbors of $i$. In a directed graph, we usually only maintain edges directed out of $i$, although it is sometimes convenient to maintain a list of incoming edges as well. Adjacency lists are ideal for sparse graphs, since they require only $\Theta(m + n)$ space (linear in the size of the graph).
Sometimes we store adjacency lists in a fancier data structure such as a hash table (Chapter 7) in order to allow for fast queries to see if an edge \((i, j)\) is present, as well as fast insertion and deletion of edges.

**Representing a Tree in Memory.** Since a free tree on \(n\) nodes contains only \(n - 1\) edges, it is typically represented just like any other sparse graph, using adjacency lists. In the event that our tree is rooted, we can augment our adjacency lists to indicate which neighbor is the parent and which are the children. Another common way to store a rooted tree in memory is shown in Figure 2.2(c), where each node maintains a pointer to its parent, first child, and previous and next sibling. Although the children of a node are stored in a particular order, it depends on our application as to whether or not this ordering is significant or not. *Binary* trees are particularly common among rooted trees, where each node has at most two children, each designated as a left or right child. A node can have no children, only a left child, only a right child, or both. We usually store a binary tree as shown in Figure 2.2(d), where each node points to its parent, left child, and right child.

We will eventually need quite a bit more graph terminology, but we will postpone introducing it until it is needed, when we study graphs in much greater detail.

### 2.2 Doing the Math

Many of our formal mathematical arguments are structured as either proofs by *induction* or by *contradiction*. We assume the reader is reasonably well-versed in both techniques. At the beginning of the next chapter, we will see how induction is commonly used to prove an algorithm is correct and analyze its running time.

**Problem 10 (Proof Practice).** This problem provides some fun and challenging “warm up” exercises to re-familiarize the reader with proofs by induction and by contradiction.

(a) Prove by induction that any integer of the form \(4^k - 1\) must be a multiple of 3. [*Solution*]

(b) Prove by induction *Kraft’s inequality*, which states that \(\sum_i 2^{-d_i} \leq 1\) in any binary tree, where the sum is over every leaf \(i\), and \(d_i\) is depth of leaf \(i\) (the root has depth zero). [*Solution*]

(c) Given a set of \(n\) points in the 2D plane, a line is *odd* if it contains an odd number of points from the set, and *even* otherwise. Argue by contradiction that among all vertical and horizontal lines, there cannot be a single unique odd line. [*Solution*]

(d) A *separator* in a connected graph is a set of nodes whose removal breaks the graph into two or more different connected components. Prove by contradiction that every connected graph on \(n\) nodes must have either a separator containing at least \(\sqrt{n}\) nodes or a simple path containing at least \(\sqrt{n}\) nodes. [*Solution*]

(e) In a group of \(n\) people, suppose each person has at least \(n/2\) friends. Please prove by contradiction that it is possible to seat everyone around a circular table so that all adjacent pairs are friends. [*Solution*]

In the rest of this section, we highlight several techniques, tricks, and insights that can be helpful for the mathematical analysis of an algorithm.
Theorem 2.3: (a) A well-known proof by picture of the Pythagorean Theorem $a^2 + b^2 = c^2$, where the same area occupied by a square of side length $c$ on the left is occupied by two squares of side lengths $a$ and $b$ on the right, and (b) proof by picture that the mediant of two fractions (the slope of the dashed line segment) lies between these two fractions (the slopes of the solid line segments).

Proof by Picture. Our first recommendation is to express mathematical ideas visually if at all possible, since this can be especially helpful for developing intuition for a mathematical concept. As the saying goes, “a picture is worth a thousand words”. If we can translate a mathematical problem into a picture, this often provides useful structural insight, and sometimes even leads to an elegant proof.

For example, in Figure 2.3(a) we see a famous visual proof of the Pythagorean Theorem, which says that the side lengths $a$, $b$, and $c$ (the hypotenuse) of a right triangle satisfy $a^2 + b^2 = c^2$. The fact that our equation involves squared terms suggests that we should draw squares of side lengths $a$, $b$, and $c$, so we can relate their areas. In Figure 2.3(b), we see a simple proof by picture that the mediant of two fractions $a_1/b_1 < a_2/b_2$, defined as $(a_1 + a_2)/(b_1 + b_2)$, lies between the two original fractions in value (assuming the $a$’s and $b$’s are all positive). Here, since our problem involves ratios, it is natural to visualize these as slopes of lines.

Factorials and Stirling’s Approximation. If complicated mathematical expressions arise in our analysis, it can be helpful to know how to bound certain problematic terms. For example, factorials appear often in problems such as sorting, since there are $n!$ ways to permute $n$ elements. We often try to bound unwanted $n!$ terms with something more manageable, for example $n! \geq 2^n$ (for $n \geq 4$) or $n! \leq n^n$. However, both of these bounds are relatively weak as $n$ grows large, since $2^n = o(n!)$ and $n! = o(n^n)$. For a better bound,

$$\log n! = \Theta(n \log n).$$

This is easy to show, since $n!$ is between $(n/2)^{n/2}$ and $n^n$, the log of both being $\Theta(n \log n)$. Stirling’s approximation says that $n!$ behaves asymptotically like $\sqrt{2\pi n}(n/e)^n$, and also provides useful bounds on the binomial coefficient $\binom{n}{k}$:

$$\left(\frac{n}{k}\right)^k \leq \binom{n}{k} \leq \left(\frac{ne}{k}\right)^k.$$

Many variants of Stirling’s approximation exist that can provide stronger bounds on $n!$. However, these are rarely necessary for algorithmic analysis.
Convenient Exponential Bounds. A commonly-used bound involving $e^x$ is

$$1 + x \leq e^x.$$ 

Looking at the graphs of $1 + x$ and $e^x$, we see that this inequality is only tight near $x = 0$, where equality is attained. For a stronger bound, we could take more terms from the Taylor expansion $e^x = 1 + x + x^2/2! + x^3/3! + \ldots$ for a higher-order polynomial bound like $1 + x + x^2/2 \leq e^x$ (for $x \geq 0$). However, the simpler linear bound is usually all we need.

A useful generalization of the bound above states that for $x \geq -1$ and $y \geq 1$,

$$1 + xy \leq (1 + x)^y \leq e^{xy},$$

The right-hand bound follows directly from $1 + x \leq e^x$, and we will prove the left-hand bound two different ways over the next few pages.

To illustrate a prototypical use of this bound, many algorithms follow an “iterative refinement” strategy where they start with a sub-optimal solution and repeatedly improve it over a series of iterations until it finally becomes optimal. If we start with a solution of value 1 and make only additive improvements in constant increments, then our algorithm could take $\Theta(V)$ total iterations, where $V$ is the value of an optimal solution. However, if we make geometric improvements (increasing the value by some constant multiplicative factor — say at least 1%), then our solution doubles every 100 iterations, since $(1 + 1/100)^{100} \geq 2$. Therefore, it only takes at most $100 \log V = O(\log V)$ iterations to reach an optimal solution. You may have seen similar uses of this bound when computing the compound interest on an investment. For example, if you earn 5% interest per year, then your investment will double at least every 20 years since $2 \leq (1 + 0.05)^{20} \leq e$ (the actual amount, approximately 2.6533, is much closer to $e \approx 2.718$ than 2, since $(1+1/n)^n$ converges to $e$ as $n$ grows large).

As another example, if each of $n$ iterations of an algorithm fails independently with probability at most $1/(2n)$, then the probability the entire algorithm succeeds is at least $(1 - 1/(2n))^n \geq 1/2$ (we will see another way to reach this conclusion later when we study the “union bound” in probability theory).

The Harmonic Series. The harmonic series,

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n},$$

appears in many of our analyses. It is useful to know that the sum of the first $n$ terms of the harmonic series behaves like $\ln n$:

$$H_n \in [\ln n, 1 + \ln n].$$

The standard [proof] of this fact uses the clever trick of bounding a discrete summation using a continuous function. You will find many algorithms with running times containing log terms on account of the fact that $H_n = \Theta(\log n)$.

Proving Equalities Using a Pair of Inequalities. To show that $A = B$, we often prove separately that $A \leq B$ and that $B \leq A$. Although simple, this approach is used countless times in mathematical proofs. On a related note, we often show that two statements $A$ and $B$ are equivalent by proving separately that $A$ implies
CHAPTER 2. USEFUL MATHEMATICAL CONCEPTS

B, and that B implies A, and we can show that two subsets A and B are equal by proving separately that A ⊇ B and that B ⊇ A.

Proving Inequalities by Minimization or Maximization. Suppose we want to show that \( f(n) \geq g(n) \), otherwise written \( f(n) - g(n) \geq 0 \). If \( f \) and \( g \) happen to be easy to minimize (say, if they are continuous functions with easily computable derivatives), we can prove that \( f(n) - g(n) \geq 0 \) by showing that the minimum possible value of \( f(n) - g(n) \) is no smaller than 0. For example, recall our earlier inequality that \( 1 + xy \leq (1 + x)^y \) for \( x \geq -1 \) and \( y \geq 1 \). A reasonably easy way to prove this (after having taken a multi-variable calculus course) is by showing that the function \( f(x, y) = (1 + x)^y - (1 + xy) \) attains a minimum value of 0 (at \( y = 1 \)), if we minimize over all possible choices for \( x \geq -1 \) and \( y \geq 1 \).

Minimum, Average, and Maximum. For any set of numbers \( a_1 \ldots a_n \), the average value lies between the minimum and the maximum:

\[
\min_{i=1 \ldots n} a_i \leq \frac{1}{n} \sum_{i=1}^{n} a_i \leq \max_{i=1 \ldots n} a_i.
\]

This obvious yet important fact is used again and again in algorithmic analyses.

Creative Ways to Add. Many of our analyses add things up in clever ways, such as by reversing the order of a double summation (e.g., adding a table column-by-column instead of row-by-row). For example, suppose \( f_i \) denotes the number of distinct integer factors of the integer \( i \), and that we want to know the sum \( f_1 + \ldots + f_n \). Letting the “indicator function” \([x|i]\) represent the value 1 if \( x \) divides \( i \), and 0 otherwise, an effective way to estimate the sum is

\[
\sum_{i=1}^{n} f_i = \sum_{i=1}^{n} \sum_{x=1}^{n} [x|i] = \sum_{x=1}^{n} \sum_{i=1}^{n} [x|i] \approx \sum_{x=1}^{n} n/x = nH_n \approx n \ln n.
\]

This is related to a technique called double counting, where we add up the same quantity two different ways to show that two expressions are equal. For example, if we sum the degrees of all nodes in a graph, we count each endpoint of each edge once. If we add up all the edges and then multiply by two, we also count each endpoint of each edge once (since each edge has two endpoints). Hence, the sum of degrees in any graph is \( 2m \).

Problem 11 (Nested Set Enumeration). Let \( S_0 \) be a set of \( n \) elements. It is well known that if we step through every subset \( S_1 \subseteq S_0 \), we will visit exactly \( 2^n \) different subsets \( S_1 \). But what if for each subset \( S_1 \) we also visit all of its subsets? See if you can argue via a simple double-counting argument that the inner loop executes exactly \( 3^n \) times, and that this number becomes \( (k + 1)^n \) if we generalize the code to include \( k \) nested loops. [Solution]

You probably know that to find the sum of an arithmetic series \( a + (a + 1) + (a + 2) + \ldots + b \), you simply take the average term value \( (a + b)/2 \) and multiply by the
number of terms \((b - a + 1)\). For example,

\[
1 + 2 + 3 + \ldots + n = n(n + 1)/2 = \Theta(n^2).
\]

Series of all shapes and sizes play an important role in many of our proofs, so it is good to know how to add them all up. For example we will often encounter the geometric series

\[
1 + x + x^2 + x^3 + \ldots + x^n = \frac{1 - x^{n+1}}{1 - x},
\]

the infinite geometric series

\[
1 + x + x^2 + x^3 + \ldots = \frac{1}{1 - x} \quad \text{(for } |x| < 1),
\]

and variants of the hybrid arithmetic-geometric series

\[
1 + 2x + 3x^2 + 4x^3 + \ldots = \frac{1}{(1 - x)^2} \quad \text{(for } |x| < 1).
\]

Although it is good to know formulas for adding up different types of series, it is perhaps even better to know how to re-derive these sorts of formulas quickly from scratch when needed. [Useful tricks for summing series]

**Exploiting Convexity or Concavity.** The reader should know what it means for a set or function to be *convex*. As shown in Figure 2.4, a set is convex if a line segment connecting any two points in the set lies entirely within the set. Intuitively, convex sets do not have any “indentations”. A function is convex if the space above the function is a convex set, and concave if the space below the function is a convex set. Convex functions curve upward and form “bowl” shapes, while concave functions curve downward and form “hill” shapes. Convexity is particularly important in the study of optimization, since convex functions and convex constraints give us particularly well-behaved optimization problems. We will discuss convexity in this setting in far greater mathematical detail in Chapter ??.

A line segment connecting two points on a convex function will over-estimate the function, and a tangent line (plane or hyperplane, in higher dimensions) will underestimate the function; the reverse is true for concave functions. This is useful since
it allows us to bound parts of a convex or concave function with a much simpler linear function. For example, recall the inequality $1 + xy \leq (1 + x)^y$ (for $x \geq -1$ and $y \geq 1$). This follows immediately from the fact that $f(x) = (1 + x)^y$ is convex for $x \geq -1$ and $y \geq 1$, and $1 + xy$ is the equation of the tangent line to this function at $x = 0$. The ability to bound a convex or concave function by a linear function is beautifully generalized by Jensen’s inequality: given $n$ numbers $x_1 \ldots x_n$ associated with nonnegative weights $\lambda_1 \ldots \lambda_n$ (where $\sum \lambda_i = 1$), then

$$f(\lambda_1 x_1 + \ldots + \lambda_n x_n) \leq \lambda_1 f(x_1) + \ldots + \lambda_n f(x_n)$$

if $f$ is convex, and the reverse inequality holds if $f$ is concave. For example, in Figure 2.4(c), you can see that $f(\frac{1}{3}x_1 + \frac{1}{3}x_2) \leq \frac{1}{3}f(x_1) + \frac{1}{3}f(x_2)$.

**Problem 12 (Relating Arithmetic and Geometric Means).** Using the fact that log is a concave monotonically increasing function, give a simple proof of the useful property that the geometric mean of $n$ numbers $(x_1 x_2 \ldots x_n)^{1/n}$ is at most their arithmetic mean $(x_1 + x_2 + \ldots + x_n)/n$. [Solution]

Later in this chapter, we will learn about the expected value $E[X]$ of a “random variable” $X$. Since expected value is nothing more than a weighted sum of values $X$ can take (weighted by the corresponding probabilities of $X$ taking those values), Jensen’s inequality tells us the useful property that $f(E[X]) \leq E[f(X)]$ if $f$ is convex, with the reverse being true if $f$ is concave.

### 2.3 Recurrences

Many algorithms break a large problem into smaller subproblems of the same form, recursively solve these, and then somehow recombine their solutions to obtain a solution for the original problem. It is often convenient to express the running time of these recursive algorithms using recursively-defined expressions called recurrences.

Here is a simple example. Given a numeric array $A[1 \ldots n]$, the maximum value subarray problem asks us to find a contiguous subarray $A[i \ldots j]$ having maximum sum (we assume the array contains some negative numbers, since otherwise the problem is trivial). The obvious $\Theta(n^3)$ time “brute force” solution is to loop over all $\binom{n}{2} = \Theta(n^2)$ possible subarrays $A[i \ldots j]$, and to sum each one in $O(n)$ time. We can easily improve this to $\Theta(n^2)$ total time using prefix sums: first precompute an array $P[1 \ldots n]$ where $P[j] = A[1] + \ldots + A[j]$. This is easy to do in $\Theta(n)$ time by scanning through $A$ while maintaining a running sum. After computing $P$, the sum of $A[i \ldots j]$ is now given in constant time by $P[j] - P[i - 1]$. As it turns out, one can do much better still. Here, we describe a recursive algorithm running in $\Theta(n \log n)$ time, and in Chapter 11 we will learn an even simpler $\Theta(n)$ algorithm based on the technique of dynamic programming!

For our recursive solution, think of $A[1 \ldots n]$ as two half-sized arrays $L = A[1 \ldots n/2]$ and $R = A[n/2 + 1 \ldots n]$. The answer is then given by the best of three solutions:

- The best solution entirely within $L$, which we can find by recursively applying our algorithm to $L$,
2.3. RECURRENCES

- The best solution entirely within $R$, which we can find by recursively applying our algorithm to $R$, and
- The best solution spanning both $L$ and $R$, which we find by taking the best suffix of $L$ added to the best prefix of $R$. These are both easy to compute in $\Theta(n)$ time by computing suffix sums of $L$ (scanning backward through $L$ keeping a running sum), and also prefix sums of $R$ (scanning forward through $R$ keeping a running sum).

In total, our algorithm makes two recursive calls to subproblems of size $n/2$ and spends $\Theta(n)$ additional time outside these calls. If $T(n)$ denotes the running time of our algorithm on an input of size $n$, we can therefore write the following recursive formula for $T(n)$:

$$T(n) = 2T(n/2) + \Theta(n).$$

As a base case, since a problem of constant size can be solved in constant time we have $T(n) = O(1)$ for $n = O(1)$.

In this section, we learn how to solve common types of recurrences to produce an explicit (non-recursive) formula for $T(n)$. For instance, the solution to the example above is $T(n) = \Theta(n \log n)$.

2.3.1 “Divide and Conquer” Recurrences

If our recursive algorithm breaks a problem of size $n$ into “multiplicatively smaller” subproblems (e.g., 2 subproblems of size $n/2$, or maybe 5 subproblems of size $n/3$ and 6 of size $n/10$), we call it a divide and conquer algorithm. The divide and conquer strategy (decompose large problem into subproblems, recursively solve subproblems, recombine subproblem solutions to obtain solution to initial problem) is a widely-applicable and powerful algorithm design methodology. We will continue to discuss this technique in the next chapter when we introduce several algorithms for sorting based on divide and conquer. Letting $T(n)$ denote the running time required to solve a problem of size $n$, divide and conquer algorithms typically give us recurrences of the form

$$T(n) = a_1T(n/b_1) + a_2T(n/b_2) + \ldots + a_kT(n/b_k) + f(n).$$

That is, we decompose a problem of size $n$ into $a_1$ subproblems of size $n/b_1$, $a_2$ subproblems of size $n/b_2$, and so on. The function $f(n)$ represents the time spent outside our recursive calls, both initially decomposing the original problem and in later recombining the solutions to our recursively-solved subproblems.

In the discussion that follows, we assume we are dealing with recurrences like the one above where the $a$'s and $b$'s are constants, and where $f(n)$ is polynomially-bounded. We initially consider functions of the form $f(n) = \Theta(n^\alpha)$, and later we show how to handle slightly more complicated functions like $f(n) = \Theta(n^\alpha \log^\beta n)$ with an added logarithm term. This is broad enough to cover most common divide and conquer algorithms.

**Useful Simplifications.** There are several easy ways to simplify a frightening divide and conquer recurrence like

$$T(n) = 2T(n/6 + \sqrt{n} + 1) + 3T(\lfloor n/8 \rfloor) + 17n^3 \log n - 3n^2 \log^4 n + 5.$$
so that it takes the form above. First, we can replace the non-recursive part

\[(17n^3 \log n - 3n^2 \log^4 n + 5)\]

with an asymptotic placeholder \(\Theta(n^3 \log n)\), since leading constants and lower-order terms in this part won’t affect the asymptotic solution. In fact, we can even drop the \(\Theta()\) entirely:

\[T(n) = 2T(n/6 + \sqrt{n} + 1) + 3T(\lfloor n/8 \rfloor) + n^3 \log n.\]

Small additive offsets like \(+\sqrt{n} + 1\) inside recursive calls also have no impact on the final asymptotic solution, and can safely be ignored. In fact, this is surprisingly true even for larger additive offsets of magnitude up to \(O(n/\log^2 n)\). Accordingly, we can disregard floor and ceiling functions, which are just additive offsets of at most one. After applying these changes, we have the much simpler recurrence

\[T(n) = 2T(n/6) + 3T(n/8) + n^3 \log n,\]

whose asymptotic solution is the same as that of the original recurrence.

**Solving Recurrences by Expansion.** A general approach for solving any recurrence is to expand it out algebraically and look for patterns. For example, take the simple recurrence \(T(n) = 2T(n/2) + n\) for our maximum value subarray problem:

\[
T(n) &= 2T(n/2) + n
= 2[2T(n/4) + n/2] + n \quad \text{(expanding } T(n/2))
= 4T(n/4) + n + n
= 4[2T(n/8) + n/4] + n + n \quad \text{(expanding } T(n/4))
= 8T(n/8) + n + n + n
\vdots
= nT(\leq 1) + n + n + \ldots + n
\overset{O(1)}{=} \log n \text{ terms}
= \Theta(n \log n).
\]

**Tree Expansions.** Suppose we arrange the terms resulting from algebraic expansion in their natural hierarchical layout as a tree, then add everything up level by level, as shown in Figure 2.5. It turns out that the level sums always give a geometric series, which we can sum with minimal effort, owing to the following insight:

- A decreasing geometric series (even an infinite one), where each term decays by a constant factor, behaves asymptotically like its first term. For example,

\[n^2 + (3/4)n^2 + (3/4)^2n^2 + \ldots = \Theta(n^2).\]

- By symmetry, an increasing geometric series behaves asymptotically just like its last term.

We can tell if the geometric series obtained from our recursion tree is decreasing, increasing, or unchanging after expanding the tree out by only a single level (which is often possible to do mentally, after some practice). Knowing the nature of the series is all we need to solve the recurrence. For example, consider a recurrence with the simple form

\[T(n) = aT(n/b) + n^\alpha.\]
If our geometric series decreases as we scan down the tree, then the root contribution $n^\alpha$ is dominant, so the recurrence solves to $T(n) = \Theta(n^\alpha)$. If the series remains unchanging, then each of the $\log_b n$ levels contributes $n^\alpha$, so the answer is $T(n) = \Theta(n^\alpha \log n)$. Finally, if the series increases, then the contribution from the leaves will be dominant. Each leaf contributes $O(1)$, and the number of leaves in a tree with depth $\log_b n$ and branching factor $a$ is $a^{\log_b n} = n^{\log_b a}$. The solution is therefore $T(n) = \Theta(n^p)$, where $p = \log_b a$.

Stated more concisely, if $T(n) = aT(n/b) + n^\alpha$ (with $T(n) = O(1)$ for $n = O(1)$ as a base case), then

$$T(n) = \begin{cases} 
\Theta(n^\alpha) & \text{if } \alpha > p \quad \text{(decreasing series)} \\
\Theta(n^\alpha \log n) & \text{if } \alpha = p \quad \text{(unchanging series)} \\
\Theta(n^p) & \text{if } \alpha < p \quad \text{(increasing series)} 
\end{cases}$$

where $p = \log_b a$. We have simplified this formula a bit by noting that we can tell if...
the geometric series is increasing, unchanging, or decreasing by simply comparing \( \alpha \) and \( p \). If you forget this, however, you can always determine the appropriate case by expanding out the tree by one level. The simple formula above is sufficient to solve the vast majority of the recurrences we will ever encounter.

**Multiple-Term Recurrences.** Consider now a recurrence of the form

\[
T(n) = a_1 T(n/b_1) + a_2 T(n/b_2) + \ldots + a_k T(n/b_k) + n^\alpha.
\]

with multiple recursive terms. We can solve this exactly as before, by expanding it into a tree and summing a geometric series. The only difference is that the tree is no longer “level” at the bottom, since \( n \) decreases at different rates down different branches. Accordingly, the level-by-level contribution starts out as a geometric series but then behaves slightly differently toward the bottom of the tree once some branches start ending. However, this only ends up changing the answer in the third case above (an increasing series), where the contribution from the leaves is dominant:

\[
T(n) = \begin{cases} 
\Theta(n^\alpha) & \text{if } \alpha > p \quad \text{(decreasing series)} \\
\Theta(n^\alpha \log n) & \text{if } \alpha = p \quad \text{(unchanging series)} \\
\Theta(n^p) & \text{if } \alpha < p \quad \text{(increasing series)} 
\end{cases}
\]

where \( p \) is the solution to \( a_1/b_1^p + a_2/b_2^p + \ldots + a_k/b_k^p = 1 \). [Proof]

For a single-term recurrence, this more complicated formula boils down to \( a/b^p = 1 \), the solution of which is \( p = \log_b a \) as before. In the multiple-term case, however, it may not be possible to find an analytic solution for \( p \). Therefore, while for a single-term recurrence you can easily determine the appropriate case above by directly comparing \( \alpha \) and \( p \), this is now more difficult, since computing the exact value of \( p \) may not be possible. There are two ways around this. The simplest is probably to expand the tree by a single level as before, since this will still reveal the nature of the geometric series. Alternatively, let \( g(x) = a_1/x_1^p + a_2/x_2^p + \ldots + a_k/x_k^p \). Since \( g \) is a decreasing function and \( g(p) = 1 \), we know that \( \alpha > p \) if \( g(\alpha) < 1 \) and that \( \alpha < p \) if \( g(\alpha) > 1 \).

**Extra Logarithmic Factors.** Consider finally a recurrence of the form

\[
T(n) = a_1 T(n/b_1) + a_2 T(n/b_2) + \ldots + a_k T(n/b_k) + n^\alpha \log^\beta n,
\]

where \( \beta \geq 0 \). Here, we can initially ignore the extra \( \log^\beta n \) term while resolving the nature of our geometric series. The term then re-appears in the solution unless we are in the increasing case where the leaves dominate:

\[
T(n) = \begin{cases} 
\Theta(n^\alpha \log^\beta n) & \text{if } \alpha > p \quad \text{(decreasing series)} \\
\Theta(n^\alpha \log^{\beta+1} n) & \text{if } \alpha = p \quad \text{(unchanging series)} \\
\Theta(n^p) & \text{if } \alpha < p \quad \text{(increasing series)} 
\end{cases}
\]

where \( p \) is again the solution to \( a_1/b_1^p + a_2/b_2^p + \ldots + a_k/b_k^p = 1 \).

**Problem 13 (Practice Solving Recurrences).** Practice makes perfect. See if you can solve the following divide and conquer recurrences with a minimal amount of calculation. [Solutions]

(a) \( T(n) = 6T(n/3) + \Theta(n^2) \).
2.3. RECURRENCES

(b) $T(n) = 5T(n/4) + \Theta(n)$.
(c) $T(n) = 3T(n/2) + \Theta(n\sqrt{n})$.
(d) $T(n) = 4T(n/2) + \Theta(n^2)$.
(e) $T(n) = T(n/6) + T(n/2) + T(n/3) + \Theta(n\log^2 n)$.
(f) $T(n) = 4T([n/3]) + 5T(n/2 + \log^3 n) + 2T(2n/3 + 7) + n^6 \log^2 n + n^{19} - 50$.
(g) $T(n) = T(\sqrt{n}) + 1$. Here, try making a change of variables (to $m = \log_2 n$) to transform this recurrence into a more familiar form. See also problem 8.

2.3.2 Linear Recurrences

In a linear recurrence, such as $T(n) = T(n - 1) + \Theta(n)$, each recursive subproblem is “additively” smaller than the original, rather than “multiplicatively” smaller as with the divide and conquer recurrence. Linear recurrences are perhaps more often found in other fields (e.g., signal processing). In algorithms, they tend to arise in conjunction with iterative (rather than recursive) algorithms that sequentially construct a solution one step at a time. For example, the recurrence above could describe an algorithm that solves a problem of size $n$ by first solving a subproblem of size $n - 1$ (say, with all but one of its input elements present) and then doing $\Theta(n)$ work to properly adjust for the presence of the final input element.

The general form of a linear recurrence is

$$T(n) = a_1T(n - 1) + a_2T(n - 2) + \ldots + a_kT(n - k) + f(n).$$

For example, we can describe the famous sequence of Fibonacci numbers using the linear recurrence $F_n = F_{n-1} + F_{n-2}$, where $F_0 = 0$ and $F_1 = 1$ as base cases. We need $k$ base cases if the recurrence contains $k$ recursive terms.

While linear recurrences can certainly be used to model and solve for algorithmic running times, we tend not to use these in this book, because it is often easier to analyze iterative algorithms in a more direct fashion. For those who are interested, however, we include here a short discussion of how to solve general linear recurrences (beyond just using algebraic expansion, which often works quite effectively).

**Fibonacci Numbers.** Solving a linear recurrence is one of many ways to obtain an explicit formula for the $n$th Fibonacci number, which surprisingly always outputs an integer value:

$$F_n = \frac{1}{\sqrt{5}} \left[ \left( \frac{1 + \sqrt{5}}{2} \right)^n - \left( \frac{1 - \sqrt{5}}{2} \right)^n \right].$$

Since $|(1 - \sqrt{5})/2| < 1$, this term decays away as $n$ grows large and the contribution from the $(1 + \sqrt{5})/2$ term dominates. Therefore, the Fibonacci numbers grow at an exponential rate:

$$F_n = \Theta(\phi^n), \text{ where } \phi = \frac{1 + \sqrt{5}}{2} \approx 1.618.$$ 

This will be useful to know later for the analysis of several data structures, such as AVL trees and Fibonacci heaps. The number $\phi$ is known as the golden ratio, and like $\pi$ and $e$ it plays an important role in the answer to several fundamental mathematical questions.
2.4 Probability

Randomization often provides a way to design algorithms that are fast, simple, and elegant. To analyze them, however, we need to equip ourselves with tools from probability theory. This section summarizes most of the tools we will need.

2.4.1 The Basics

A probability space is a set of all possible fundamental outcomes of some random trial, each with an associated nonnegative probability, where these probabilities all sum to 1. An event $E$ is a set of outcomes. Its probability, $\Pr[E]$, is the sum of the probabilities of these outcomes. For example, if our random trial involves rolling two 6-sided dice, there are 36 different possible outcomes, $(1,1), (1,2), (1,3), \ldots, (6,6)$, each occurring with probability $1/36$. The event that both dice sum to 4 includes outcomes $(1,3)$, $(2,2)$, and $(3,1)$, and therefore has probability $3/36 = 1/12$. The natural way to informally interpret this probability is that if we repeatedly perform random trials (each involving a roll of two dice), on average we expect to see a sum of 4 once every 12 trials.

If all outcomes in a random experiment are equally likely, then the probability of an event $E$ is just the number of outcomes in $E$ divided by the total number of possible outcomes. This observation usually reduces a probability question to a combinatorics question. For example, what is the probability of seeing exactly 7 heads in 10 coin flips? There are $2^{10}$ different sequences of coin flips (these are our different outcomes), and $\binom{10}{7}$ of them belong to the event that we see exactly seven heads, so the probability of our event is $\binom{10}{7}/2^{10} \approx 0.117$.

Since the probabilities of all possible outcomes sum to 1, we know that

$$\Pr[E] = 1 - \Pr[\bar{E}]$$

where $\bar{E}$, the complement of $E$, is the set of all outcomes not in $E$ (in other words, the event that $E$ does not occur). It is often easier to directly compute $\Pr[\bar{E}]$ rather than $\Pr[E]$; for instance, the probability we see heads at least once in 10 coin flips is $1 - 1/2^{10}$, where $1/2^{10}$ is the probability of the complement, that we see all tails.

Since events are nothing more than sets of outcomes, we can use set union and intersection to express relationships between them. If $E_1$ and $E_2$ are two events,
then \( E_1 \cup E_2 \) is the event that either one or both events occur, and \( E_1 \cap E_2 \) is the event that both occur.

**The Union Bound.** The probability that either \( E_1 \) or \( E_2 \) (or both) occurs is

\[
\Pr[E_1 \cup E_2] = \Pr[E_1] + \Pr[E_2] - \Pr[E_1 \cap E_2].
\]

This is obvious from the Venn diagram pictured in Figure 2.6, since \( \Pr[E_1] + \Pr[E_2] \) counts all the outcomes in \( E_1 \) and \( E_2 \), but counts the outcomes in \( E_1 \cap E_2 \) twice — so we compensate by subtracting out \( \Pr[E_1 \cap E_2] \). A more general form this idea is the inclusion-exclusion principle, where we compute \( \Pr[E_1 \cup \ldots \cup E_k] \) for a union of \( k \) events by adding their individual probabilities, subtracting the probabilities of all intersecting pairs of events, adding the probabilities of all intersecting triples of events, and so on in an alternating fashion. If we only want a rough upper bound, however, we can say that

\[
\Pr[E_1 \cup E_2 \cup \ldots \cup E_k] \leq \Pr[E_1] + \Pr[E_2] + \ldots + \Pr[E_k],
\]

where equality holds only if all events are disjoint (\( E_i \cap E_j = \emptyset \) for all pairs of events). This is known either as the union bound or as Boole’s inequality, and it is an extremely useful tool in many of our analyses. It follows easily from the fact that every outcome in \( E_1 \cup \ldots \cup E_k \) has its probability counted exactly once on the left-hand side, but potentially several times on the right-hand side, depending on the number of events in which the outcome is contained.

The union bound tells us that the failure probability of a complex system is at most the sum of the failure probabilities of its individual parts. A machine with 200 parts, each failing with probability at most \( 10^{-6} \), has an overall probability of failure (i.e., of one or more parts failing) at most \( 200 \times 10^{-6} \). In the context of a randomized algorithm, suppose an algorithm spends \( O(f(n)) \) time on a single “generic” input element with probability at least \( 1 - 1/(2n) \), so it fails to process this element quickly enough with probability at most \( 1/(2n) \). Taking a union bound over all \( n \) elements, the probability of failure on any element is at most \( 1/2 \), so consequently the algorithm runs in \( O(nf(n)) \) time with probability at least \( 1/2 \).

**Conditional Probability.** The probability of event \( A \) given that event \( B \) occurs is written as

\[
\Pr[A \mid B] = \frac{\Pr[A \cap B]}{\Pr[B]}.
\]

Otherwise written, \( \Pr[A \cap B] = \Pr[A \mid B] \Pr[B] \). Since by symmetry we also have \( \Pr[A \cap B] = \Pr[B \mid A] \Pr[A] \), we can easily derive Bayes’ rule,

\[
\Pr[A \mid B] = \frac{\Pr[B \mid A] \Pr[A]}{\Pr[B]},
\]

which is useful for relating \( \Pr[A \mid B] \) and \( \Pr[B \mid A] \). We will use Bayes rule quite often when we study machine learning later in Chapter ??.

**Independence.** Two events \( A \) and \( B \) are independent if knowledge about the occurrence of one event does not change the probability of occurrence of the other.

\[\text{1For those familiar with calculus, conditional probabilities behave in much the same way as the chain rule for derivatives. For example, the derivative of } f(g(h(x))) \text{ is } f'(g(h(x)))g'(h(x))h'(x). \text{ Similarly, we can write } \Pr[A \cap B \cap C] = \Pr[A \mid B, C] \Pr[B \mid C] \Pr[C].\]
Formally, $A$ and $B$ are independent if and only if any of the following equivalent conditions are true:

- $\Pr[A | B] = \Pr[A],$
- $\Pr[B | A] = \Pr[B],$ or
- $\Pr[A \cap B] = \Pr[A] \Pr[B].$

Independence is often used to simplify the probability of both $A$ and $B$ occurring, $\Pr[A \cap B],$ by replacing it with the product $\Pr[A] \Pr[B].$

### 2.4.2 “With High Probability” Results

Monte Carlo randomized algorithms can give incorrect answers if they are unlucky. In order to persuade anyone to use them, we need to ensure that the probability of an incorrect answer is extremely small. We would like our algorithms to succeed with high probability, which usually means the following in the computing literature:

We say a Monte Carlo algorithm with input size $n$ is correct with high probability if it fails with probability at most $1/n^c$ for any constant $c > 0$ of our choosing. More formally, given any constant $c > 0$, we can set the hidden constant in our running time appropriately such that

$$\Pr[\text{algorithm outputs incorrect answer}] \leq 1/n^c$$

for sufficiently large values of $n$.

The interesting (and to some, confusing) aspect of this definition is the part where $c$ is allowed to be any constant of our choosing. In order to claim a high probability bound, we must be able to reduce the failure probability to $1/n^2$, $1/n^{10}$, $1/n^{1000}$, or $1/n^c$ for any other constant $c$. The impact of our choice of $c$ shows up only in the hidden constant in the running time of our algorithm, so it does not change the overall asymptotic running time.

To see that the definition above is a reasonable way to define “with high probability”, observe that you might be happy with an algorithm that only fails at most 1% of the time, but other situations may call for an even more robust algorithm that fails at most 0.001% of the time. It isn’t really satisfactory for us to pick any fixed constant threshold, since there is no obvious choice that would be universally acceptable. Instead, we use a threshold based on the input size, $n$. For example, we could say that a Monte Carlo algorithm is correct with high probability if it fails with probability at most $1/2^n$ on inputs of size $n$. This way, we are virtually guaranteed that large inputs are handled correctly. It is true that the bound is weaker for small inputs, but this is reasonable since with small inputs, there are only a limited number of random outcomes that can occur, and if even one of these is bad, then our overall probability of failure may be somewhat large. Just as the most important feature of running time is how it scales with problem size, the most meaningful definition of “with high probability” also scales with problem size. It

DRAFT. Not for redistribution. Copyright © Brian C. Dean
turns out that an exponentially-small error bound like $1/2^n$ is a bit too difficult to achieve with most problems, so the standard definition above requires a slightly weaker polynomially-small error bound. Of course, nobody will complain if your algorithm manages to satisfy an even stronger bound.

**Boosting Success Probability Through Repetition.** Suppose we have a Monte Carlo randomized algorithm that produces a correct answer with probability at least $1/2$. As long as the algorithm has the ability to detect when it makes a mistake, we can boost its success probability from a constant guarantee like $1/2$ to a high probability guarantee by simply running $O(\log n)$ independent trials of the algorithm. If we perform, say $c \log n$ trials, then our algorithm only fails if every individual trial fails, and this happens with probability at most $(1/2)^{c \log n} = 1/n^c$. Since we can select $c$ to be any constant we like, this gives us a high probability guarantee of success. If we want a failure probability of at most $1/n^{10}$, we choose $c = 10$. If we want a failure probability of at most $1/n^{100}$, we choose $c = 100$. In any case, we still perform only $O(\log n)$ iterations of our algorithm, with $c$ disappearing as the hidden constant.

**Problem 14 (Multicolored Marbles).** You are given a jar containing $n$ marbles. Either (i) all the marbles are blue, or (ii) half the marbles are blue and the other half are red. By examining only $O(\log n)$ marbles, how can we determine with high probability of correctness which scenario is reality? [Solution]

**Problem 15 (Property Testing).** The problem above is a trivial example from the domain of property testing, where we want to develop a very fast (typically sublinear time) randomized test that can distinguish two cases: whether an object has some property, or whether it is “far” from having the property. Objects close to having the property do not need to be handled correctly; for example, the fast randomized test from the solution of the problem above would be unable to readily distinguish between all marbles being blue, or all but one being blue. Another simple example, which we examine in this problem, is distinguishing whether an array $A[1 \ldots n]$ of distinct numbers is either (i) in sorted order, or (ii) far from being sorted, in that any increasing subsequence has length at most $(1 - \varepsilon)n$ (remember that a subsequence is not necessarily contiguous within the array).

Our algorithm is simple: we call an array index $i$ consistent if a binary search for $A[i]$ correctly reaches this element; if the array is sorted, then all indices are consistent. Our algorithm runs binary searches to test consistency of $O(1/\varepsilon \log n)$ independently-chosen random indices, so its total running time is $O(1/\varepsilon \log^2 n)$. We guess that the array is sorted if all are consistent. Otherwise, we guess it is far from being sorted. For a challenge, please argue that this algorithm distinguishes cases (i) and (ii) above with high probability. In case (i), we always get the right answer, so we can focus on case (ii). Letting $C$ be the set of all consistent indices in $A$, please show that $C$ is an increasing subsequence. In case (ii) we must therefore assume $|C| < (1 - \varepsilon)n$; however, please show that this will cause one of our binary searches to identify an inconsistent index with high probability. [Solution]

**Problem 16 (Trends in Random Sequences).** Suppose we take a sequence of $n$ distinct numbers arranged in random order. We call a trend a contiguous subsequence that is either increasing or decreasing. Please argue that with high probability (at least $1 - 1/n^c$ for any constant $c$ of our choosing), the length of the longest trend in our array will be $\Theta(\log n)$. Why does this result also show that in a random length-$n$ binary string, the longest contiguous run of 0’s or 1’s must have length $\Theta(\log n)$ with high probability? Although this problem can be easily approached with Chernoff bounds (Section 2.4.4), you can also solve it directly by proving separate upper and lower bounds — i.e., by showing that the longest trend has length $O(\log n)$ and $\Omega(\log n)$ with high probability. [Solution]
Problem 17 (The Birthday Paradox). The birthday paradox is a famous mathematical puzzle stated as follows: assuming everyone has a birthday chosen at random, how many people must you gather together in a room before there is a reasonable chance (say, probability at least 1/2) that two of them share the same birthday? The problem is called a “paradox” because most people regard its answer, 23, as being surprisingly low. In general, the threshold where shared birthdays become likely is roughly where the number of people reaches the square root of the number of possible birthdays. In this problem, we investigate the mathematics behind the birthday paradox as well as some of its algorithmic implications. Along the way, we will have a chance to review many of the concepts we have covered so far in this chapter, such as independence, the union bound, and high probability guarantees.

(a) Suppose that we have \( n \) individuals, each with a birthday chosen independently at random from \( m \) possible days. Let \( D \) be the event that all \( n \) individuals have different birthdays. Please write an exact mathematical expression for \( \Pr[D] \). Using bounds from earlier in this chapter, please show that (i) \( \Pr[D] \geq 3/4 \) if \( n \leq \sqrt{m}/2 \), and (ii) \( \Pr[D] \leq 1/e \) if \( n \geq 2\sqrt{m} \). For simplicity, please assume \( m \) is a perfect square. [Solution]

(b) Using a union bound over all \( \binom{n}{2} \) pairs of individuals, please show that \( \Pr[D] \geq 1/2 \) if \( n \leq \sqrt{m} \). This result will play an important role in the analysis of hash table data structures in Chapter 7, where we exploit the fact that if \( n \) elements are randomly mapped to a table of size \( n^2 \), there is at least a 1/2 probability that no two elements collide at the same location. [Solution]

(c) In a simple peer-to-peer network made up of \( m \) computers, it may be the case that the only way to find a particular object (e.g., a file) is to query computers one-by-one until we find a computer hosting the object we seek. In order to speed this process up, suppose we take each object of interest and replicate it on \( \sqrt{m} \) computers throughout the network, chosen arbitrarily. To search for an object, we probe \( O(\sqrt{m} \log m) \) randomly-chosen computers and succeed if one of them has the object in question. Please argue that this procedure guarantees a high probability of success. [Solution]

(d) In a lake with \( m \) fish, show how to estimate \( m \) to within some constant factor with high probability by catching, marking, and releasing only \( O(\sqrt{m} \log m) \) random fish. The result of problem 32 may help. [Solution]

High Probability Guarantees for Las Vegas Algorithms. With Las Vegas randomized algorithms, the output is always correct but the running time can vary depending on luck. It may be hard to convince someone to use the algorithm unless we can persuade them that it is unlikely that the running time will be significantly larger than some target running time. In other words, we would like an algorithm with a running time guarantee that holds with high probability.

We say a Las Vegas algorithm with input size \( n \) runs in \( O(f(n)) \) time with high probability if it fails to run in \( O(f(n)) \) time with probability at most \( 1/n^c \), where \( c > 0 \) is any constant of our choosing. More formally, given any any constant \( c > 0 \), we can find a constant \( k \) such that

\[
\Pr[\text{running time exceeds } k f(n)] \leq 1/n^c
\]

for sufficiently large values of \( n \).

We define a “high probability” running time bound for a Las Vegas algorithm in a very similar fashion to that of a Monte Carlo algorithm, in terms of a polynomially-
small failure probability that scales with problem size\(^2\).

For a simple example, consider using binary search within a sorted array \(A[1 \ldots n]\). In each iteration, we examine some “pivot” element \(A[j]\) and then narrow our remaining search to either \(A[1 \ldots j-1]\) or \(A[j+1 \ldots n]\). It is natural to choose \(j = n/2\), since this lets us reduce the size of the problem under consideration by a factor of 2 in each iteration and achieve a worst-case running time of \(O(\log n)\). However, what if we instead choose \(j\) uniformly at random from \(1 \ldots n\)? The resulting algorithm, which we call randomized binary search, is perhaps closer to what happens in practice when we want to look up a word in a dictionary. Since it is hard to open the dictionary to the page containing the exact middle word, we might instead just open to a random page. As you might suspect, randomized binary search runs in \(O(\log n)\) time with high probability. If we want a guarantee that its running time will be \(O(\log n)\) with probability at least \(1 - 1/n^{10}\), we need only select an appropriately large hidden constant for the \(O(\log n)\) running time. If we want a guarantee of at least \(1 - 1/n^{100}\), we select a larger hidden constant. Regardless of the constant we choose, the running time bound is still of the form \(O(\log n)\).

**Randomized Reduction.** In a few pages, we will learn how to use Chernoff bounds to prove high probability guarantees. Chernoff bounds are extremely useful and powerful tools in the analysis of randomized algorithms, but they are often somewhat complicated to apply. In order to make our proofs as easy as possible, we can derive from the Chernoff bound the following:

| The Randomized Reduction Lemma. | Suppose we have a problem with input size \(n\), and we run an algorithm that in each iteration effectively reduces the size of the problem under consideration by some amount. If every iteration of our algorithm reduces the size of a problem to at most some constant fraction of its original size \(q \in [0, 1)\) with at least some constant probability \(p \in (0, 1]\) (where the probability at each step is independent of the others), then the algorithm requires only \(O(\log n)\) iterations with high probability. |

The randomized reduction lemma\(^3\) is quite natural since it builds upon a very familiar principle: if we start with a problem of size \(n\) and apply an algorithm that, in each iteration, reduces the effective problem size to at most some constant fraction \(q \in [0, 1)\) of its original size, then the algorithm will perform \(O(\log n)\) iterations. The prototypical example of this principle is binary search, with \(q = 1/2\). All the lemma above says is that we achieve the same logarithmic performance (with high probability) as long as there is a good chance of problem size reduction in each iteration.

Remarkably, this lemma (which we prove later) is the only tool we need in order to argue almost of the high probability bounds in this book! For example, with randomized binary search we can easily show that each iteration reduces our problem

---

\(^2\)In fact, it is actually meaningless to define “with high probability” in this case using a fixed constant threshold as a failure probability. If an algorithm has an expected running time of \(O(n^2)\) time (we will say in minute what “expected” means), then we can adjust the hidden constant in the \(O(\cdot)\) notation using Markov’s Inequality (also discussed shortly) to change the probability that it fails to run in \(O(n^2)\) time to any constant of our choosing: 1%, 0.001%, and so on.

\(^3\)Note that this is a name you will only find here in this book, since the lemma does not seem to appear by any common name elsewhere.
to at most $q = 2/3$ of its original size with probability at least $p = 1/3$: let $A$ denote the $n/3$ smallest elements in our array, let $B$ denote the $n/3$ middle elements, and let $C$ denote the $n/3$ largest elements. With probability 1/3, we choose a pivot element from $B$ and reduce our problem to at most 2/3 of its original size, since this eliminates either $A$ or $C$ from consideration (depending on the comparison with the pivot). Therefore, the randomized reduction lemma immediately tells us that randomized binary search runs in $O(\log n)$ time with high probability.

Note that the randomized reduction lemma can be applied even in situations when we aren’t explicitly reducing the size of a problem. For example, we could start with a problem of unit size and in each iteration expand by a constant factor with some constant probability, stopping when we reach $n$. We could also identify some other parameter associated with our algorithm’s state that satisfies the properties of the lemma — either shrinking or expanding by some constant factor with constant probability in each iteration.

**Union Bounds and High Probability Results.** Our definition of “with high probability” meshes particularly well with the union bound. In particular, if we can prove that some property holds with high probability for a generic element in our input (failure probability at most $1/n^c$), then the union bound tells us that this property also holds with high probability for all $n$ elements in our input (with failure probability at most $n \times 1/n^c = 1/n^{c-1}$, which we can still set to any level of our choosing by selecting $c$ appropriately). This simplifies many of our proofs by allowing us to focus on proving a high probability result for only a single element or smaller subproblem, rather than for the entire input taken as a whole. For example, in the next chapter we will show that the randomized quicksort algorithm runs in $O(n \log n)$ time with high probability by first observing that it spends $O(\log n)$ work on *any single element* with high probability (using the randomized reduction lemma), and then by applying a union bound to conclude that it spends $O(\log n)$ work per element on *all of the $n$ input elements* also with high probability.

**Non-Uniform Subproblem Sizes.** If we perform a randomized binary search over only a length-$k$ subarray of a larger length-$n$ array, the randomized reduction lemma tells us this will run in $O(\log k)$ time, but only with high probability with respect to $k$ (i.e., with a failure bound of the form $1/k^c$). This can make it difficult to use the union bound to aggregate $n$ high probability results for differently-sized subproblems to obtain a global high probability bound; to do this, we would need a failure bound of the form $1/n^c$ for each subproblem. Fortunately, when we later prove the randomized reduction lemma, we will show that it actually guarantees an $O(\log n)$ running time with failure probability $1/n^c$ for any $n \geq k$. We can therefore aggregate high probability results over non-uniform subproblem sizes with little trouble — a subtle, but important point.

The following problems provide good examples of the randomized reduction lemma in action (often in conjunction with the union bound).

**Problem 18 (Properties of Random Permutations).** This problem and the next will give us some practice using the randomized reduction lemma.

(a) Consider a random permutation of $\{1, 2, \ldots, n\}$. Please argue that the length of its longest sequential subsequence is $O(\log n)$ with high probability. A sequential subsequence is a subsequence whose elements are increasing by 1 as we move from left
to right. For example, in the ordering 7, 3, 1, 4, 5, 2, 6, the elements in the longest sequential subsequence are underlined\(^4\). [Solution]

(b) Suppose we step through a random permutation of \(n\) distinct numbers and count the number of times we encounter an element that is the largest element seen so far. Please show that we will encounter only \(O(\log n)\) such elements with high probability. As a hint, you may want to think backward instead of forward. [Solution]

(c) Consider a random permutation of \(\{1, 2, \ldots, n\}\) stored in an array \(A[1 \ldots n]\). In any permutation, we can trace out the cycle containing an element by repeatedly moving from our current index \(i\) to the element at index \(A[i]\), until we return to our starting point. Using this operation, we can decompose any permutation into a collection of disjoint cycles — this is a common alternative method for representing a permutation. Please show that a random length-\(n\) permutation decomposes into \(O(\log n)\) cycles with high probability. As a hint, first show that irrespective of the value of \(k\), the probability that the cycle traced out starting from a fixed starting element has length \(k\) is exactly \(1/n\). [Solution]

Problem 19 (Load Balancing with a Random Assignment). Probability theorists are quite fond of “balls in bins” problems where we throw \(n\) balls at random into \(m\) bins (in this problem, we assume \(n = m\) for simplicity). If we interpret balls as unit-sized computational jobs and bins as machines, we are constructing a random assignment of jobs to machines. This type of situation often arises in practice when we want to serve traffic for a website that is so popular that a single web server will not suffice. Here, we often locate a set of \(m\) web servers behind a “switch” that randomly assigns incoming requests (jobs) to web servers. This approach balances the load by assigning each server on average \(n/m\) jobs (exactly one job, under our assumption that \(n = m\)). Please show that each machine receives at most \(O(\log n)\) jobs with high probability (that is, with probability at least \(1 - 1/n^c\), for any constant \(c\) of our choosing), so there is little danger of substantially overloading a machine\(^5\). [Solution]

Problem 20 (Properties of Random Trees). There are several nice ways to build a random \(n\)-node labeled tree (with nodes numbered with labels \(1 \ldots n\))\(^6\). In this problem, we consider the following simple method: start from node 1, and for each \(i\) from \(2 \ldots n\) in sequence, attach node \(i\) to a randomly chosen node in \(1 \ldots i - 1\).

(a) Please show that the maximum node degree in our random tree is \(O(\log n)\) with high probability. [Solution]

(b) Please argue that the diameter (length of the longest path) of our random tree is \(O(\log n)\) with high probability. [Solution]

\(^4\)We discuss some related results to this problem later in the book: in problem 208 we develop an \(O(n \log n)\) algorithm for finding the longest increasing subsequence, and in problem ?? we prove the Erdős–Szekeres theorem, which states that any length-\(n\) sequence must have either an increasing or decreasing subsequence consisting of at least \(\sqrt{n}\) elements.

\(^5\)Note that with a more detailed Chernoff bound analysis, this bound can be improved slightly to \(O(\log n / \log \log n)\). There are also some interesting related results achievable by probing multiple bins: for example, if you place each ball in whichever of two randomly chosen bins is least full, then a much more sophisticated analysis shows that the expected maximum fullness is only \(O(\log \log n)\).

\(^6\)One elegant method is using the so-called Prüfer code — a method for encoding any \(n\)-node tree (with nodes labeled \(1 \ldots n\)) as an integer sequence of length \(n - 2\) (with each element in the range \(1 \ldots n\)). The mapping is one-to-one (thereby providing a nice proof that there are exactly \(n^{n-2}\) labeled trees on \(n\) nodes), and can be performed in either direction — tree to sequence, or sequence to tree — in \(\Theta(n)\) time. Since the number of occurrences of \(x\) in the sequence is one less than the degree of node \(x\) in the tree, problem 19 tells us that if we build a random \(n\)-node tree from a randomly-generated Prüfer code (equivalent to throwing \(n - 2\) balls into \(n\) bins), its maximum degree will be \(O(\log n)\) with high probability. See also problem ?? on generating a random spanning tree of a graph, and see the endnotes for a brief discussion of properties of random graphs in general.
CHAPTER 2. USEFUL MATHEMATICAL CONCEPTS

Problem 21 (The Coupon Collector Problem). Suppose that you see a special advertisement on your box of breakfast cereal claiming that inside the box you will find one of $n$ different types of special coupons, each equally likely. Please show that you will find at least one of each coupon type with high probability if you open $O(n \log n)$ boxes of cereal. As a hint, try to prove the slightly stronger result that by the time you have accumulated all $n$ coupon types, you will have found only $O(\log n)$ copies of any single particular coupon type with high probability. [Solution]

Problem 22 (Randomly Spreading Information in a Distributed Network). Suppose we have a distributed network of $n$ processors where one processor wants to broadcast a message to the others. Suppose further that we are operating under a “synchronous” model of distributed computation where time proceeds in steps according to some global clock, and in each time step a processor can exchange a single message with another processor.

(a) In each time step, suppose each processor that has heard the message contacts a random processor and sends it the message (note that if we are unlucky, the other processor might have already heard the message). This is called a gossip protocol. Using the tools from this chapter, see if you can prove that all $n$ processors will hear the message after only $O(\log n)$ steps with high probability (with probability at least $1 - 1/n^c$ for any constant $c > 0$ of our choosing). As a hint, consider separately the number of steps until some constant fraction of the processors has initially heard the message, and then the number of steps required to spread to the remaining processors. [Solution]

(b) Suppose we implement a gossip protocol by “pulling” rather than “pushing” the message. That is, in each time step, each processor without the message contacts a random other processor and asks for a copy of the message (which it may or may not have, and even if it does have the message, it may not be able to communicate if it is already talking to a different processor, since each processor can only participate in a single communication session per step). Please try to prove that this variant also distributes a message from a single processor to all $n$ processors in $O(\log n)$ steps with high probability. [Solution]

2.4.3 Random Variables and Linearity of Expectation

We now turn to the second major concept in our study of probability theory: random variables. In contrast, to a standard (non-random) variable that represents a single value, a random variable is associated with a probability distribution over values. For example, if $X$ represents the smaller of the face values when we roll two 6-sided dice, its distribution is

\[
\begin{align*}
1 & : \frac{11}{36} \\
2 & : \frac{9}{36} \\
3 & : \frac{7}{36} \\
4 & : \frac{5}{36} \\
5 & : \frac{3}{36} \\
6 & : \frac{1}{36}
\end{align*}
\]

If we actually perform a random trial by rolling two dice, then we can think of sampling a value according to this distribution to instantiate the value of $X$. In other words, $X$ serves as a “placeholder” for a value between 1 and 6 that would only materialize after we perform a random trial (although we never actually replace $X$ with a concrete value in this fashion). A random variable assigns a numeric value to every possible outcome associated with some random experiment; in the example above, the outcome $(3, 5)$ maps to the value $X = 3$.

Two random variables $X$ and $Y$ are independent if knowledge of the value of $X$ does not change the probability distribution associated with $Y$, and vice versa. If
X is a random variable, then expressions like \( X = v \) or \( X \geq v \) represent events (i.e., the event that \( X \) takes the value \( v \), or the event that \( X \) takes a value at least as large as \( v \)), so it makes sense to write \( \Pr[X = v] \) or \( \Pr[X \geq v] \). Please refrain from writing \( \Pr[X] \) as this makes no sense, since random variables are not events!

**Expectation.** The *expected value* (also called the *mean*) of a random variable \( X \) is the “center of mass” for \( X \)'s probability distribution, defined as follows for a discrete random variable:

\[
E[X] = \sum_v v \Pr[X = v].
\]

That is, \( E[X] \) is the sum over all possible values \( v \) that \( X \) can take, of \( v \) weighted by the probability of \( X \) taking the value \( v \).

Here are some examples:

- If \( X \) denotes the minimum of the values obtained by two rolls of a 6-sided die, then \( E[X] = 1 \cdot \frac{11}{36} + 2 \cdot \frac{9}{36} + 3 \cdot \frac{7}{36} + 4 \cdot \frac{5}{36} + 5 \cdot \frac{3}{36} + 6 \cdot \frac{1}{36} = 91/36 \).
- If \( X \) denotes the number of heads we see when flipping 100 fair coins, then \( E[X] = 50 \). If the coins are biased and show heads with probability \( \frac{3}{4} \) and tails with probability \( \frac{1}{4} \), then \( E[X] = 75 \).
- If we repeatedly flip a biased coin (showing heads with probability \( \frac{1}{10} \)) and \( T \) denotes the number of flips up to and including the first time we see heads, then \( E[T] = 10 \).
- If \( T \) denotes the amount of time we spend performing a linear search for a randomly-chosen element in an \( n \)-element array, then \( E[T] = \Theta(n) \).
- If \( T \) denotes the amount of time we spend performing a binary search for a randomly-chosen element in a sorted \( n \)-element array, then \( E[T] = \Theta(\log n) \).

The running time of a Las Vegas algorithm is a random variable. It is small if the algorithm is lucky and large if the algorithm is unlucky in its random choices. The simplest question you can generally ask about a Las Vegas algorithm is what is its expected running time. If our expected running time is, say, \( O(n \log n) \), this gives a sense of what we can expect “on average” when we run the algorithm, but it still leaves open the possibility that the running time may have high variability, and may take much more than \( O(n \log n) \) time with some non-negligible probability. In order to convince even the most stubborn critic that our algorithm is reliable enough to use, we may want to attempt to prove the much stronger result that it runs in \( O(n \log n) \) time with high probability.

**Problem 23 (Expected Versus High Probability Results).** It is worth noting that while a high probability running time bound of \( O(f(n)) \) is generally regarded as much

\[\text{DRAFT. Not for redistribution. Copyright © Brian C. Dean}\]
stronger than an expected running time bound of $O(f(n))$, neither bound technically implies the other. To illustrate this fact, please describe the probability distribution for the running time of a hypothetical randomized algorithm that is $O(f(n))$ in expectation but not $O(f(n))$ with high probability. Next, show how we could have a running time that is $O(f(n))$ with high probability, but not $O(f(n))$ in expectation. \[\text{Solution}\]

**Linearity of Expectation.** A random variable representing the running time of even a simple randomized algorithm is usually too complicated to allow computation of its expected value using the definition above. Fortunately, **linearity of expectation** lets us compute expected values of complicated random variables by decomposing them into sums of much simpler random variables. For instance, suppose $Z = 2X + 3Y$ where $X$ and $Y$ are simple random variables, and $Z$, defined in terms of $X$ and $Y$, is also a random variable, albeit with a slightly more complicated distribution\(^8\). In this case, we can take advantage of the fact that $\mathbb{E}[:]$ is a **linear** operator: if $c$ is a constant and $X$ and $Y$ are random variables, then $\mathbb{E}[cX] = c\mathbb{E}[X]$ and $\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]$. It is especially noteworthy that $X$ and $Y$ do not need to be independent. We can apply linearity of expectation to any weighted sum of random variables, and this is what makes it such a powerful and useful principle. Applied to the example above, we have $\mathbb{E}[Z] = \mathbb{E}[2X + 3Y] = 2\mathbb{E}[X] + 3\mathbb{E}[Y]$, so $\mathbb{E}[Z]$ is easy to compute as long as we know $\mathbb{E}[X]$ and $\mathbb{E}[Y]$. Linearity of expectation is trivial to prove based on the definition of expectation. \[\text{Short proof}\]

Let us work through a short example just to illustrate a common usage of the technique. Suppose we want to compute $\mathbb{E}[X]$ where $X$ denotes the number of heads we see in 100 flips of a fair coin. The random variable $X$ has a somewhat complicated distribution, so computing $\mathbb{E}[X]$ directly would involve solving

$$
\mathbb{E}[X] = \sum_v v \Pr[X = v] = \sum_{v=0}^{100} v \binom{100}{v} / 2^{100},
$$

which, while feasible, is perhaps more complicated than we might like. Instead, let us decompose $X$ into a sum of much simpler random variables:

$$
X = X_1 + X_2 + \ldots + X_{100},
$$

where $X_i$ takes the value 1 if the $i$th flip is heads, 0 otherwise. A 0/1-valued random variables like $X_i$ is called an **indicator** random variable since its value indicates whether or not a certain event has occurred. The expected value of an indicator random variable is easy to compute:

$$
\mathbb{E}[X_i] = \sum_v v \Pr[X_i = v] = 0 \cdot \Pr[X_i = 0] + 1 \cdot \Pr[X_i = 1] = \Pr[X_i = 1] = 1/2.
$$

\(^8\)In Chapter ??, we will see that the distribution of a sum of random variables is given by the convolution of their individual distributions.
Note that $E[X_i]$ is just the probability of the event for which $X_i$ serves as an indicator. This is true for any indicator random variable — its expected value is just the probability of its associated event. We now have

$$E[X] = E[X_1 + \ldots + X_{100}] = E[X_1] + \ldots + E[X_{100}] = 50,$$

which is the result we intuitively expect. Linearity of expectation is one of the most valuable tools we have for analyzing randomized algorithms, and we will use it on many occasions. The reader is encouraged to learn it well.

**Expectations of Products.** Since expectations of sums behave so nicely, what happens with products? Unfortunately, $E[XY] = E[X]E[Y]$ is *not* true in general. If this property does hold, then we say $X$ and $Y$ are *uncorrelated*. Since independent random variables are always uncorrelated, we can always decompose the expectation of a product of independent random variables into a product of expectations. Please take care not to confuse this with linearity of expectation for sums, which applies to any sum of random variables regardless of independence.

**Expected Trials Until Success.** If we roll a 6-sided die, then $Pr[E] = 1/6$ if $E$ is the event that we roll a 2. If we keep rolling until the event $E$ occurs, how many trials do we expect to perform, including the final successful trial? The answer is what we might intuitively expect: 6. This is an example of a very useful principle: if we perform a sequence of independent trials, where each trial succeeds with probability $p$, then the expected number of trials up to and including the first success is $1/p$. Similarly, if each trial succeeds with probability *at least* $p$, then we expect *at most* $1/p$ trials. [Very short proof]

**Problem 24 (Linearity of Expectation Practice).** This problem demonstrates how we can use linearity of expectation to compute the expectation of a complicated random variable.

(a) **Exchanging Hats.** Suppose $n$ people in a room are each wearing different hats. If they all exchange hats according to a random permutation, how many people do you expect to receive their original hat? [Solution]

(b) **Balls in Bins.** If we randomly throw $n$ balls into $m$ bins, what is the expected number of balls that end up in each bin? Among all $\binom{n}{2}$ pairs of balls, how many pairs do we expect to “collide” by landing in the same bin? What is the expected number of empty bins? For the last question, an approximate answer is fine. [Solution]

(c) **The Coupon Collector Problem.** Recall problem 21: inside your box of breakfast cereal you will find one of $n$ different types of special coupons, each equally likely. Please show that you expect to open $\Theta(n\log n)$ boxes of cereal before you have collected at least one of each coupon type. As a hint, decompose the sequence of boxes we check into a series of “phases”, where during one phase we have collected exactly $k$ distinct coupons and we are opening boxes in hopes of finding a coupon of any of the remaining $n - k$ types. [Solution]

(d) **Revisiting the Randomized Reduction Lemma.** Consider a randomized algorithm whose input consists of $n$ elements, where in each iteration of the algorithm there is at least some constant probability of reducing our problem to at most some constant fraction of its original size. In this setting, we learned how the randomized reduction lemma guarantees that we spend only $O(\log n)$ iterations with high probability. Just for fun, use linearity of expectation to show an analogous (and slightly weaker) result that we spend $O(\log n)$ iterations in expectation. [Solution]
(e) **Prefix Maxima.** Suppose in an array $A[1 \ldots n]$ that you want to compute for each index $j$ the maximum of $A[1 \ldots j]$. This is easy to do in $\Theta(n)$ time, of course, by scanning through $A$ and keeping a running maximum. As an exercise, however, consider the following alternative method: process the elements of $A$ in random order. For each element $A[j]$, scan backward down to $A[1]$ keeping a running maximum. During this scan, we stop at the first element $A[i]$ previously processed, since we will have already computed and stored the maximum of $A[1 \ldots i]$, making it unnecessary to scan these elements again. What is the expected running time of this algorithm? [Solution]

**Linearity of Expectation over a Random Number of Trials.** Suppose $X_1, X_2$, etc., represent random variables with identical distributions, so they all have the same expectation $E[X_i]$. Linearity of expectation tells us how to compute the expected sum of a fixed number $n$ of these variables: $E[X_1 + \ldots + X_n] = nE[X_i]$. However, what if $n$ itself is a random variable? This happens quite often in randomized algorithm analysis, for example if we are performing a random number of iterations of a subroutine that itself involves randomization. In this case, intuition would suggest that $E[X_1 + \ldots + X_n] = E[n]E[X_i]$, and indeed this is the case, provided there is no dependence between the $X_i$'s and the number of trials $n$. This result goes by the name of Wald’s Theorem.

**Problem 25 (Network Contention Resolution).** Suppose $n$ processors are all simultaneously trying to transmit information on a shared network cable, where each processor has a stream of packets to transmit. In each time step, each processor can decide if it should attempt to transmit a packet, or if it should remain idle. The goal is to have exactly one processor transmit in each step, but this can be difficult to achieve since the processors have no means of coordinating with each-other. If more than one processor attempts to transmit, the transmissions all “collide” and fail (and the processors can detect when this happens by listening to the network, so they know to retry transmission of the same packets later).

(a) If the processors all know the value of $n$, then let each one independently attempt transmission during each time step with probability $p = 1/n$. Please show that the expected number of time steps until a successful transmission in this case is approximately $e$, and (if you know calculus) please argue $1/n$ is in fact the optimal value to choose for $p$ in order to maximize throughput. [Solution]

(b) If our processors don’t know the value of $n$, let each processor attempt to transmit with probability $1/2$. If it decides to transmit and fails due to a collision, then it becomes “dormant” and waits until a time step occurs when it hears no other transmissions before it wakes up and starts attempting to transmit again (again with probability 1/2 per time step). Please argue that we expect $O(\log n)$ time steps to elapse between successful transmissions. [Solution]

**“Per-Element” Expected Running Time Analysis.** Recall from our discussion of high probability results that the union bound allows us easily to carry over a high-probability bound for a single “generic” element in our input to all $n$ element elements in our input. For example, if we can show that our algorithm spends $O(\log n)$ time on a generic input element with high probability, then it also spends $O(n \log n)$ total time with high probability. Linearity of expectation allows us to make a similar argument when we analyze the expected running time of an algorithm. If $X$ denotes the total running time of our algorithm and $X_i$ denotes the running time spent only on element $i$, then it is easy to compute $E[X] = E[X_1] + \ldots + E[X_n]$. 

---

DRAFT. Not for redistribution. Copyright © Brian C. Dean
by first computing the “per element” expected running times $E[X_1] \ldots E[X_n]$. For example, if our algorithm spends $O(\log n)$ expected time on a generic input element, then linearity of expectation tells us that it spends $O(n \log n)$ total expected time.

Common Types of Distributions. Most random variables we will encounter have probability distributions belonging to a handful of well-known classes. The simplest of these is the Bernoulli distribution, which takes the value 1 with probability $p$ and 0 with probability $1 - p$. Bernoulli random variables are also called indicator random variables when they are used to indicate whether or not a particular event happens. For example, if we flip a biased coin that comes up heads with probability $p$ and tails with probability $1 - p$, then we could say that our indicator variable takes the value 1 to indicate that the coin comes up heads. The expected value of this indicator variable is exactly $p$, the probability of its associated event.

A random variable has a geometric distribution if its value indicates the number of trials until a particular Bernoulli event is successful. For example, if $X$ denotes the number of flips of our biased coin up to and including the first trial where it comes up heads, then $X$ has a geometric distribution. It is called a “geometric” distribution since the probability of exactly $k$ trials, $\Pr[X = k] = p(1 - p)^k$, decays in a geometric fashion with $k$. As we just mentioned, a geometric distribution derived from a Bernoulli event occurring with probability $p$ has expected value $1/p$.

Problem 26 (Simulating a Biased Coin with Unbiased Coin Flips). Suppose we would like to simulate the outcome of flipping a biased coin that comes up heads with probability $p$ and tails with probability $1 - p$, where $p$ could potentially even be an irrational number. All we have at our disposal is a single fair coin. Can you think of a simple method that requires only $O(1)$ expected flips of our fair coin to simulate one flip of the biased coin? As a hint, first write out $p$ in binary.

Solution

Suppose now that we flip $n$ biased coins, each heads with probability $p$. We can denote whether each individual coin comes up heads by using indicator variables $X_1 \ldots X_n$, and we can write the total number of heads as $X = X_1 + \ldots + X_n$. A random variable like $X$ that is a sum of Bernoulli variables has a binomial distribution, so named because $\Pr[X = k]$ (the probability we see $k$ heads in $n$ flips) is given by the binomial coefficient formula $\binom{n}{k} p^k (1 - p)^{n-k}$. Using linearity of expectation, we find that $E[X] = E[X_1] + \ldots + E[X_n] = np$. Binomial distributions tend to be tightly concentrated around their means, a property we will exploit when we introduce Chernoff bounds momentarily.

The binomial distribution is often approximated by a distribution shaped like the Taylor series expansion of $e^{np}$ known as the Poisson distribution, which is well-studied in probability theory due to its numerous properties and applications.

Problem 27 (The Odds Algorithm and Secretary Problems). The secretary problem is a famous probabilistic puzzle where you interview $n$ candidates for a secretary position in random order, each with a distinct skill level that becomes known to you only at the interview. After interviewing each candidate, you can either hire them (forgoing the chance to interview any future candidates) or dismiss them in hopes of finding a better
The goal is to maximize your chances of hiring the best candidate, and it turns out we can achieve this by interviewing and dismissing the first \(1/e\) fraction of all candidates, then to hire any candidate from that point on who is the best seen so far.

A remarkably simple algorithm, known as the odds algorithm, solves not only this problem but a generalization that has several other applications. In a sequence of \(n\) independent random trials, let \(p_i\) be the probability that trial \(i\) “succeeds”, let \(q_i = 1 - p_i\), and let \(r_i = p_i/q_i\) be the odds of success (the ratio of probability for versus against). Our goal is to observe the outcome of each successive trial (success or not), choosing a point at which to stop that maximizes our probability of stopping on the last successful event. For the secretary problem, \(p_i = 1/i\), since success corresponds to interviewing a candidate who is the best seen so far. However, this scenario applies to many other problems, such as betting on the last time a stock price will jump up, or guessing when to stop at an open parking space in hopes of parking at the closest available space to your place of work.

Letting \(R_t = r_t + r_{t+1} + \ldots + r_n\) and \(Q_t = q_{t+1} \cdots q_n\), let \(t\) be the largest index\(^{10}\) at which \(R_t \geq 1\) (or we set \(t = 1\) if no such index exists). Please show that an optimal strategy is to stop at the first successful event at or beyond index \(t\), and that this gives a probability of \(R_t Q_t\) of stopping on the last successful event. \([\text{Solution}]\)

2.4.4 Tail Inequalities

In addition to computing the expected value of a random variable, we are often interested in the probability that the random variable deviates significantly from its expectation. For example, we will feel much safer using a randomized algorithm whose expected running time is \(\Theta(n^2)\) if we know that its running time tends to fall consistently around \(\Theta(n^2)\) and not vary too widely. There are several common methods for bounding the probability of drawing a sample from the “tail” of a distribution (the part of the distribution that is far away from the expected value).

Markov’s Inequality. If \(X\) is any nonnegative random variable, then

\[
\Pr[X \geq kE[X]] \leq \frac{1}{k}.
\]

This is known as Markov’s inequality. It tells us that if our algorithm runs in 100 expected units of time, then the probability it runs longer than 400 units of time is at most 1/4 and the probability it runs longer than 900 units of time is at most 1/9. The inequality is “tight” in the sense that we cannot hope for a better bound if all we know about \(X\) is its expected value. We can typically only prove fairly weak results using Markov’s inequality, but in many cases we will find it sufficient for the task at hand. \([\text{A simple proof of Markov’s inequality}]\)

Problem 28 (From Expected to High Probability Guarantees). Given any Las Vegas algorithm with expected running time \(O(f(n))\), how can we construct a Las Vegas algorithm whose running time is \(O(f(n) \log n)\) with high probability? See if you can find a solution that does not require knowledge of the hidden constant in the original expected running time guarantee. \([\text{Solution}]\)

---

\(^{10}\)If every trial has success probability \(p\), our knowledge of geometric random variables tells us to expect \(1/p\) trials until the first success. Equivalently, we keep a running sum of success probabilities and stop when this reaches 1. Note that the odds algorithm has a pleasantly symmetric form for guessing the last success, where we sum the odds in reverse order until we reach a sum of 1.
Problem 29 (Randomized Reduction with Reduction in Expectation). Please prove that an algorithm will take \( O(\log n) \) iterations either in expectation or with high probability if the expected size of our problem instance after each iteration is at most a constant factor \( q \in [0, 1) \) times its current value (and satisfaction of this property in each iteration is independent of other iterations). [Solution]

Chebyshev’s Inequality. The variance of a random variable \( X \) is the expected squared deviation from \( X \)’s mean:

\[
\text{Var}[X] = \mathbb{E}[(X - \mathbb{E}[X])^2].
\]

Applying Markov’s inequality to the random variable \((X - \mathbb{E}[X])^2\), we obtain

\[
\Pr[(X - \mathbb{E}[X])^2 \geq k^2 \text{Var}[X]] \leq \frac{1}{k^2},
\]

which is usually written as

\[
\Pr[|X - \mathbb{E}[X]| \geq k \text{Std}[X]] \leq \frac{1}{k^2},
\]

where \( \text{Std}[X] = \sqrt{\text{Var}[X]} \) is the standard deviation of \( X \) (we take the square root of the variance to reduce it back to a quantity on the same scale as \( X \)’s original distribution). The inequality above is called Chebyshev’s inequality, and it is a useful tool for analyzing a random variable whose variance is easy to compute. For example, it tells us that the probability \( X \) falls more than 2 standard deviations away from \( \mathbb{E}[X] \) is at most 1/4, and the probability that \( X \) deviates from \( \mathbb{E}[X] \) by more than 5 standard deviations is at most 1/25.

Chebyshev’s inequality is stated most simply in terms of the z-score of a sample of a random variable \( X \), defined as \((X - \mathbb{E}[X])/\text{Std}[X]\), telling us how many standard deviations \( X \) lies above or below its mean. Chebyshev’s inequality tells us that there is at most a \( 1/k^2 \) probability that the z-score of a random variable has absolute value \( k \) or larger.

We don’t use Chebyshev’s inequality too much in this book, so we will spare the reader a lengthy discussion of properties of variances and standard deviation. We will limit our discussion to three useful facts:

- \( \text{Var}[X] = \mathbb{E}[X^2] - (\mathbb{E}[X])^2 \) is another common formula used to compute variance; it is easily obtained by expanding out \( \mathbb{E}[(X - \mathbb{E}[X])^2] \).
- If \( k \) is a constant and \( X \) is a random variable, then \( \text{Var}[kX] = k^2 \text{Var}[X] \).
- If \( X \) and \( Y \) are uncorrelated random variables (which is true if they are independent), then \( \text{Var}[X + Y] = \text{Var}[X] + \text{Var}[Y] \).

Problem 30 (Polling and Monte Carlo Estimation). Here we illustrate a typical use of Chebyshev’s inequality. Suppose we wish to estimate the value of \( \pi \) by randomly sampling points from a unit square in which a circle is inscribed. As we sample more and more points, the fraction of the points landing inside the circle should approach the ratio of the area of the circle to that of the square, \( \pi/4 \). Suppose we want to compute an estimate of \( \pi \) in this fashion that is accurate to within an additive error of at most \( \varepsilon \) with 99% probability. According to Chebyshev’s inequality, approximately how many points do we need to sample? Similarly, suppose we are conducting a poll to estimate...
what fraction of the population wants to vote for candidate \( A \) in an election. According to Chebyshev’s inequality, approximately how many people must we sample at random before we have an estimate that is accurate to within an additive error of \( \varepsilon \) with 99\% probability. [Solution]

**Chernoff Bounds.** Some of the most powerful bounds in our arsenal are the Chernoff bounds, which tell us that a binomial random variable tends to be very tightly concentrated near its expected value. For example, we can write the number of heads in \( n \) coin flips as \( X = X_1 + \ldots + X_n \) where each \( X_i \) is an independent indicator random variable that takes the value 1 if coin flip \( i \) is heads, 0 otherwise. Here, we anticipate that the number of heads will almost always end up very close to \( \mathbb{E}[X] = n/2 \).

There are many different forms of Chernoff bounds, depending on whether we want a bound on deviation below or above \( \mathbb{E}[X] \), and whether we want a bound on the probability of absolute or relative deviation from \( \mathbb{E}[X] \). Letting \( X = X_1 + \ldots + X_n \) be a sum of independent indicator random variables, we have

1. \( \Pr[X \leq \mathbb{E}[X] - \varepsilon] \leq e^{-2\varepsilon^2/n} \)
2. \( \Pr[X \geq \mathbb{E}[X] + \varepsilon] \leq e^{-2\varepsilon^2/n} \)
3. \( \Pr[X \leq (1 - \varepsilon)\mathbb{E}[X]] \leq e^{-\varepsilon^2\mathbb{E}[X]/2} \)
4. \( \Pr[X \geq (1 + \varepsilon)\mathbb{E}[X]] \leq \begin{cases} e^{-\varepsilon^2\mathbb{E}[X]/4} & \text{if } \varepsilon \leq 2e - 1 \\ 2^{-(\varepsilon+1)\mathbb{E}[X]} & \text{if } \varepsilon \geq 2e - 1 \end{cases} \)

As an example, what is the probability we see more than 75 heads in 100 coin flips? According to the second bound above, \( \Pr[X \geq 75] \leq e^{-25^2/50} = e^{-12.5} < 0.000004 \), which is quite unlikely! [For the interested reader, a proof of our Chernoff bounds]

Since the full-fledged Chernoff bounds above can sometimes be cumbersome to use, that is why we earlier developed the randomized reduction lemma as a simpler interface to the Chernoff bounds. Using Chernoff bounds, we can now give a short [proof] of the randomized reduction lemma.

**Problem 31 (Boosting Success Probabilities With Two-Sided Errors).** Suppose we have a Monte Carlo algorithm for solving a decision problem (so the output is either “yes” or “no”). We say that our algorithm has only a one-sided error if it correctly outputs “no” for all “no” inputs, but for a “yes” input it might erroneously output “no” with constant probability \( p < 1 \) (or vice versa: it could answer “yes” inputs correctly and make mistakes on “no” inputs). In this case, we have shown earlier that by running \( O(\log n) \) independent trials of the algorithm we can obtain a high probability bound of success (at least \( 1 - 1/n^c \) for any constant \( c > 0 \) of our choosing). Consider now the slightly trickier case of two-sided error, where our algorithm might mistakes on both “yes” and “no” inputs. Please show that if the probability of making such a mistake is at most some constant \( p < 1/2 \), then we can again use \( O(\log n) \) independent trials of our algorithm to obtain an answer that is correct with high probability. [Solution]

**Problem 32 (Improving Robustness by Taking the Median).** Suppose you have an algorithm whose output is accurate to within some \( 1 \pm \varepsilon \) factor with constant probability strictly greater than 1/2. Please show that by taking the median of \( O(\log n) \)
invocations of the algorithm, we get an answer that is accurate to within a $1 \pm \varepsilon$ factor with high probability. How is this result related to that of the preceding problem? [Solution]

Problem 33 (Interpolation Search). This problem serves as a challenging “grand finale” for our review of probability theory. Suppose we have a sorted array $A[1 \ldots n]$ whose elements were independently drawn from the uniform distribution over $[0, 1]$ prior to sorting\footnote{We can apply this technique to any probability distribution as long as we know the inverse of its cumulative density function, which effectively allows us to map samples from the complicated distribution to “equivalent” samples over the uniform distribution on $[0, 1]$.}, and that we would like to find the element in $A$ closest in value to a target value $v$. An obvious solution is binary search, running in $O(\log n)$ time. However, since we know more here about the structure of $A$, we can actually achieve an $O(\log \log n)$ expected running time using a biased version of binary search known as interpolation search. The general idea is illustrated with an example: in an array $A[1 \ldots 1000]$, we would expect to find $v = 0.95$ at roughly index 950, so we choose this as our next guess, as opposed to the middle index binary search would have chosen.

At each step of the search, we are considering some subarray $A[i \ldots j]$, where we have observed $A[i]$ and $A[j]$ but not yet any of the elements in between. Due to the uniform distribution of our array elements, we expect the contents of $A[i]$ up to $A[j]$ in a linear fashion\footnote{Conditioning on the fact that we have observed several array elements including $A[i]$ and $A[j]$ but not yet $A[i+1 \ldots j-1]$, the subarray $A[i+1 \ldots j-1]$ still behaves as if it was obtained by sorting an array of numbers independently chosen uniformly from the range $[A[i], A[j]]$. This subtle point is important, since it allows us to assume every subarray we encounter during our search behaves probabilistically just like the initial array.}. Let $\lambda = \frac{v - A[i]}{A[j] - A[i]} \in [0, 1]$ be the relative distance at which we expect to find $v$ between $A[i]$ and $A[j]$. Interpolating by this amount between $i + 1$ and $j - 1$, we choose the next index to visit as

$$k = \text{Round}((i + 1)(1 - \lambda) + (j - 1) \lambda),$$

where the function $\text{Round}(x)$ rounds $x$ up to $\lfloor x \rfloor$ with probability proportional to the fractional part of $x$, or down to $\lfloor x \rfloor$ otherwise (this provides a clean, unbiased way to round $x$ to an integer value). Normally, we would compare $v$ with $A[k]$ and then recurse left or right accordingly, but just to help simplify our analysis, let us consider looking at $A[k-1]$, $A[k]$, and $A[k+1]$, stopping if $v$ lies between $A[k-1]$ and $A[k+1]$, or recursing left on $A[i \ldots k-1]$ or right on $A[k+1 \ldots j]$ otherwise. This lets us argue that the probabilities of recursing left or right are both at most $1/2$ [proof] (whereas in the standard approach where we only look at $A[k]$, one of these probabilities would be larger than $1/2$ and harder to bound, making the rest of our particular style of analysis more complicated).

In keeping with the way we have explained many other randomized algorithms, we show that interpolations search runs in $O(\log \log n)$ expected time using the randomized reduction lemma. Let $m = j - i - 1$ be the number of interior elements in our current subarray, and let $B = \log[m^2 \lambda(1 - \lambda)]$ be the log of the product of the number of these elements we anticipate on the left and right of our target element. We regard $B$ as our “effective problem size”, and claim that this quantity reduces by some constant fraction in each iteration with some constant probability. Since $B \leq 2 \log n$ initially, the randomized reduction lemma tells us that we expect $O(B) = O(\log \log n)$ total iterations.

Assume $\lambda \leq 1/2$ (a symmetric argument works if $\lambda \geq 1/2$), and let $\varepsilon = \frac{A[j]-A[i]}{\sqrt{m}}$. Please use a Chernoff bound to show that $\Pr[A[k-1] > v + \varepsilon]$ is at most some constant strictly less than $1/2$. Then, show that $B$ decreases to at most some constant fraction of its current value unless (i) we recurse to the right, or (ii) $A[k-1] > v + \varepsilon$. Since $\Pr[(i)] \leq 1/2$ and you have just shown that $\Pr[(ii)]$ is at most a constant less than $1/2$, each iteration will indeed reduce our effective problem size with at least some constant probability, thereby fulfilling the conditions of the randomized reduction lemma. [Solution]
2.5 Linear Algebra

Most of the serious “number crunching” going on in the world involves linear algebra in some way or another. Linear systems and linear optimization play such a crucial role in such a broad range of computational applications that any reader pursuing a career in computation is highly encouraged to develop some level of familiarity with at least basic linear algebra. In recent years, an increasing number of elegant connections have been made between linear algebra and other seemingly unrelated algorithmic problems, particularly graph problems. For example, we will use linear algebra techniques later in the book to compute shortest paths, transitive closures, graph clusters, linear orderings, and matchings, and to count directed walks in a DAG, spanning trees, arborescences, and directed Eulerian tours. Prior knowledge of linear algebra is therefore helpful for several parts of this book. Here, we review some of the fundamentals.

Our notation and terminology is fairly standard. All vectors are column vectors; we can express the vector \( x \) as a row vector by writing \( x^\top \), where the superscript \( T \) denotes the transpose operation. For a vector \( x \) with \( n \) components, we refer to these components individually as either \( x(1) \ldots x(n) \) or \( x_1 \ldots x_n \). We assume the reader is familiar with the dot product (or inner product) of two vectors, \[ x \cdot y = x^\top y = \sum_{i=1}^{n} x_i y_i, \]
as well as its geometric interpretation: if \( y \) is a unit vector, then \( x \cdot y \) gives the length of the projection of \( x \) in the direction of \( y \) (we elaborate more on this in Section ?? when we study computational geometry). Accordingly, two vectors \( x \) and \( y \) are perpendicular, or orthogonal, if \( x \cdot y = 0 \). When we study computational geometry we will define and make extensive use of the cross product of a pair of three-dimensional vectors. When we write \( x \leq y \) for two length-\( n \) vectors \( x \) and \( y \), this means that \( x_i \leq y_i \) for each component \( i = 1 \ldots n \).

**Independence and Rank.** A set of vectors is linearly independent if we cannot express one of them as a weighted sum of the others. The rank of a set of vectors is the maximum number of linearly independent vectors in the set, and accordingly the rank of a matrix \( A \) is the maximum number of linearly independent rows of \( A \) (the same number tells us the maximum number of linearly independent columns in \( A \)). A linear algebra enthusiast would also describe the rank of a set of vectors as the dimensionality of the subspace spanned by all of their weighted combinations. A matrix is singular if its rows or columns are not linearly independent, or equivalently if its rank is less than the number of rows or columns.

**Matrix-Vector and Matrix-Matrix Multiplication.** It is assumed the reader knows how to multiply a matrix by a vector and a matrix by a matrix: The matrix-vector product \( Ax \) gives us a vector whose entries are the dot products between \( x \) and rows of \( A \). Equivalently, it gives us a sum of \( A \)'s columns, each weighted by the components of \( x \). The matrix product \( AB \) gives us a matrix in which the \((i,j)\) entry is the dot product of the \(i\)th row of \( A \) and the \(j\)th column of \( B \). Matrix multiplication is associative \( (AB)C = A(BC) \), and distributive \( A(B + C) = AB + AC \), but not necessarily commutative (in general, \( AB \neq BA \)). The inverse of an \( n \times n \) matrix \( A \), denoted \( A^{-1} \), is the unique \( n \times n \) matrix such that \( AA^{-1} = A^{-1}A = I \),
where $I$ is the $n \times n$ identity matrix (all zeros, with ones down the diagonal). Every $n \times n$ matrix has an inverse unless it is singular. In Chapter 22, we will learn how to define an object called the pseudo-inverse, that plays a similar role to the inverse of either a singular or non-square matrix.

Problem 34 (Using Matrix Multiplication to Count Directed Walks). This problem illustrates just one use of linear algebra to solve a simple graph problem. Let $A$ be the adjacency matrix of a directed graph, so $A_{ij} = 1$ if there is a directed edge from node $i$ to node $j$, and $A_{ij} = 0$ otherwise. Prove, using induction on $k$, that the $(i, j)$ entry of $A^k$ tells us the number of different directed walks from $i$ to $j$ that are exactly $k$ edges in length. [Solution]

Orthonormal Bases. In $n$-dimensional space, a set of mutually orthogonal unit vectors $v_1 \ldots v_n$ is called an orthonormal basis, and it defines what we could consider as a new set of “coordinate axes” into which we can project any point in space (i.e., the coordinates of point $p$ along these new axes will be $p \cdot v_1 \ldots p \cdot v_n$). We say an $n \times n$ matrix $A$ is orthonormal if it has orthonormal rows or columns. Orthonormal matrices have many nice properties. For example, the length of a vector does not change when we multiply it by an orthonormal matrix, and if $A$ is orthonormal then $A^{-1}$ is simply $A^\top$, the transpose of $A$. In Chapter 22, we will learn several approaches for constructing an orthonormal basis for the subspace spanned by a set of vectors.

Special Classes of Matrices. Orthonormal matrices are one nice class of matrix we will often encounter. Others include symmetric matrices (square matrices $A$ for which $A = A^\top$), symmetric positive semi-definite matrices (symmetric matrices $A$ for which $x^\top Ax \geq 0$ for any vector $x$), symmetric positive definite matrices (similarly defined, only $x^\top Ax > 0$ for all $x \neq 0$), permutation matrices (the identity matrix with its rows or columns re-ordered), and Markov matrices (nonnegative matrices whose rows each sum to 1). We will say more about each of these classes later in the book.

Norms and Distances. The length of a vector $x$ is given by
\[
||x|| = \sqrt{x^\top x} = \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2},
\]
and accordingly the distance between two $n$-dimensional points $x$ and $y$ is given by the length of the vector joining them:
\[
||x - y|| = \sqrt{(x - y)^\top(x - y)} = \sqrt{(x_1 - y_1)^2 + \ldots + (x_n - y_n)^2}.
\]
There are several other common norms we can use to measure lengths of vectors, and hence also distance. Above we have used the common $L_2$, or Euclidean norm $||x||_2$ (which we usually write as just $||x||$ without the subscript 2). Other common vector norms are the $L_1$ and $L_\infty$ norms,
\[
||x||_1 = |x_1| + |x_2| + \ldots + |x_n|, \quad ||x||_\infty = \max_i |x_i|.
\]
When used to measure distance, the $L_1$ distance between $x$ and $y$,
\[
||x - y||_1 = |x_1 - y_1| + |x_2 - y_2| + \ldots + |x_n - y_n|,
\]
are given by $||x||_p = (|x_1|^p + |x_2|^p + \ldots + |x_n|^p)^{1/p}$. Note: $L_p$ norms are defined for all $p \geq 1$, but $L_0$ norms are only defined for $p = 0$. In general, the length of a vector $x$ as measured with the $L_p$ norm is given by $||x||_p = (|x_1|^p + |x_2|^p + \ldots + |x_n|^p)^{1/p}$.

DRAFT. Not for redistribution. Copyright © Brian C. Dean
is sometimes called the “Manhattan” distance between \(x\) and \(y\) since in 2 dimensions \(||x - y||_1\) corresponds to the distance one would along a grid of city streets from point \(x\) to point \(y\). The \(L_1\) and \(L_\infty\) distances are sometimes nice to use since they avoid producing irrational numbers, whereas the \(L_2\) (Euclidean) distance measure involves a square root.

### 2.6 Other Useful Mathematical Topics

The study of algorithms gives us an opportunity to learn useful ideas and techniques from many different areas of mathematics. In this section, we briefly highlight some of the principal remaining mathematical topics not yet discussed above that we will encounter throughout the rest of the book.

**Metrics.** A metric is a distance function over a set of elements (e.g., points in space, nodes in a graph) that is nonnegative, symmetric, and that obeys the triangle inequality. A distance function \(d\) satisfies the triangle inequality if \(d(x, z) \leq d(x, y) + d(y, z)\) for all ordered triples \((x, y, z)\). In other words, the “direct” distance from \(x\) to \(z\) is never more than the distance from \(x\) to \(z\) when traveling through an intermediate point \(y\) (this is the same notion as the triangle inequality you learned in elementary geometry, which states that one side of a triangle can be no larger than the sum of the other two sides). The \(L_2\) (Euclidean), \(L_1\) (Manhattan), and \(L_\infty\) distance functions are all metrics. We will see several other examples throughout this book. In Chapter ?? we will see a version of Euclidean distance scaled by the shape of an underlying point set known as Mahalanobis distance. When we study graphs in more detail, we will learn that both shortest path distance and also commute time distance form metrics over the set of all nodes in a graph.

**Problem 35 (Finding a Central Element).** Consider a set of \(n\) elements \(x_1 \ldots x_n\) belonging to a larger set \(U\) on which we have defined some distance metric \(d\). For example, \(x_1 \ldots x_n\) could be points in \(k\)-dimensional space with \(d(x, y)\) giving the standard Euclidean distance between points \(x\) and \(y\). We wish to find a “central” element \(x^* \in U\) whose average distance to our \(n\) points \(D(x^*) = \frac{1}{n} \sum_i d(x^*, x_i)\) is minimal (this is equivalent to finding a point \(x^*\) minimizing the total distance \(\sum_i d(x^*, x_i)\) summed over all points). In many cases, this can be rather difficult; for example, finding a consensus ranking under the inversion distance metric (Problem 63) and finding a consensus string under the edit distance metric (Section 9.5.3) are examples of NP-hard optimization problems that fit into this framework. In fact, even for \(n\) points in the 2-dimensional plane and Euclidean distances, it is known that there is no explicit formula for \(x^*\). Please show that if we simply choose one of our \(n\) input points \(x_i\) as our answer (whichever one minimizes \(D(x_i)\)), then we achieve a 2-approximation. Along the way, please argue that if we pick one of our input points \(x_i\) uniformly at random, then \(E[D(x_i)] \leq 2D(x^*)\). [Solution]

**Continuous Mathematics; Analysis and Calculus.** We encounter problems of a continuous nature quite often in practice. To give a few examples, we may want to solve a nonlinear system of equations, find the roots of a polynomial, fit a polynomial or exponential curve to a set of points, minimize a linear or nonlinear function over some set of constraints, approximate the area bounded by a nonlinear curve, or study the evolution over time of a system of differential equations. In this book, the reader will occasionally benefit from knowledge of derivatives (gradients,
2.6. OTHER USEFUL MATHEMATICAL TOPICS

Figure 2.7: Equivalent combinatorial objects: (a) forests of rooted ordered trees on \( n \) nodes are equivalent to binary trees on \( n \) nodes by re-interpreting each node’s “first child” and “next sibling” pointers as “left child” and “right child”; (b) a “full” binary tree on \( n \) nodes (each node being a leaf or having two children) can be traversed (as in Figure 6.3(a)) to obtain an equivalent sequence of \( 2n - 2 \) balanced parentheses, or similarly a binary sequence of length \( 2n - 2 \) where each prefix has no more 0s than 1s (known as Dyck sequences), or a sequence of \(+1\)s and \(-1\)s of length \( 2n - 2 \) where every prefix has a nonnegative sum, equivalent to a “landscape” that never crosses underground — here, following an edge downward emits \((-1, 1, \text{or} +1, \text{and upward emits } 0, \text{or} -1\). Part (c) shows equivalence between full binary trees on \( 2n + 1 \) nodes, triangulations of convex polygons with \( n + 2 \) sides, non-crossing perfect matchings in convex polygons with \( 2n \) vertices, and staircase partitions built from \( n \) rectangles.

in higher dimensions), and how to approximate a function locally in terms of a low-degree polynomial — i.e., the first few terms of its Taylor expansion. These techniques will be particularly useful in Chapter ?? when we study algorithms for optimization of continuous functions.

Combinatorics and Counting. Many algorithmic problems come directly from combinatorics, involving counting, enumerating, or randomly sampling discrete objects of a particular type. For example, in Section ?? we study the problem of counting the number of spanning trees in a graph — a famous problem with an elegant solution. For many problems, we can ask not only for a single solution but also for an algorithm that counts or enumerates all solutions. Since there may be exponentially many, we typically want to count them in polynomial time\(^{14}\) and enumerate them in polynomial time per solution.

Algorithms deal with many types of “combinatorial objects” (e.g., sequences, trees, graphs) that often end up closely related or even equivalent to each-other, allowing algorithms designed for one type of object to be easily adapted for others. Examples are shown in Figure 2.7; those shown in parts (b) and (c) are just a small subset of dozens of objects with a “nested parenthesis” structure, countable by the famous Catalan numbers. [Further details on equivalence and structure of these objects]

\(^{14}\)There is one slight nuisance we should bear in mind for counting problems, particularly those in which the answer can be exponentially large. The RAM model of computation usually only permits us \( O(\log n) \) bits in a word, allowing us to count only “polynomially” high — up to \( n^c \) for some constant \( c \). If we need to count to some “exponentially” high number like \( 2^n \), this requires at least \( n \) bits. We can resolve this issue either by relaxing the model of computation or by penalizing our running times to account for arithmetic on large numbers.
CHAPTER 2. USEFUL MATHEMATICAL CONCEPTS

Problem 36 (Indexing and Sampling Combinatorial Objects). If we are considering a large set of \( k \) possible objects (e.g., permutations, combinations, graphs, trees, etc.), it is sometimes helpful to find a convenient mapping between these objects and the integers \( 0 \ldots k - 1 \). After doing this, we can easily enumerate all \( k \) objects by stepping through \( i = 0 \ldots k - 1 \) and asking our mapping to produce the \( i \)th object. It is also easy to randomly sample an object by applying the mapping to a random value of \( i \) chosen uniformly between 0 and \( k - 1 \). For simplicity, since \( k \) might be exponentially large, please feel welcome to ignore the issue of word size for parts (a) and (b) below.

(a) Suppose the objects we want to index are the \( n! \) different permutations of an array containing \( n \) distinct elements. Give a \( \Theta(n) \) algorithm that constructs the \( i \)th such permutation in lexicographic order given any \( i \in \{0, 1, \ldots, n! - 1\} \), and give a \( \Theta(n) \) algorithm that performs the inverse mapping as well, taking an ordering and producing \( i \) as output. [Solution]

(b) Consider the same problem for combinations, rather than permutations. That is, we want to index all \( k \)-element subsets of an array containing \( n \) distinct elements. Show how to map between a subset (stored in an array of size \( k \)) and an index \( i \in \{0, 1, \ldots, \binom{n}{k} - 1\} \) in \( O(k) \) time. [Solution]

(c) To generate a random \( k \)-element subset of an \( n \)-element array, consider the following approach: for each \( i \) from \( n - k + 1 \) up to \( n \), choose a random index \( j \in \{1, \ldots, i\} \). If the \( j \)th element is not in our subset already, add it, otherwise add the \( i \)th element instead. We can implement this method in \( O(k) \) expected time with hashing (Chapter 7). Please show that it does indeed produce a \( k \)-element subset chosen uniformly at random. [Solution]

Information Theory and Entropy. A nonnegative vector \( x \) with components summing to one can be regarded as a probability distribution. The entropy of this vector, defined by

\[
H(x) = \mathbf{E}[-\log_2 x_i] = -\sum_{i=1}^{n} x_i \log x_i
\]

is maximized at \( H(x) = \log_2 n \) for a vector \( x = (\frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n}) \) representing a uniform distribution, and minimized at \( H(x) = 0 \) for a vector like \( x = (0, 0, 0, 1, 0) \) with all its mass isolated in a single component. Entropy reflects the amount of disorder or uncertainty inherent in a probability distribution, or alternatively the expected amount of “surprise” inherent in drawing a sample from it, measured in the expected number of bits required to describe the sample. For example, a sample from the distribution described by \( x = (\frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n}) \) is equally likely to be any of its \( n \) components, so we need \( \log_2 n \) bits to describe the result, whereas we need zero bits to describe the completely predictable result from sampling from \( x = (0, 0, 0, 1, 0) \). Entropy is an important concept in data compression, since data described by a low-entropy distribution can be compressed more easily (e.g., English text, with its non-uniform letter frequencies, can be more easily compressed than a text with roughly equal frequencies of ‘A’ through ‘Z’). We elaborate on connections with data compression in Section 10.2.1. Entropy also provides a natural formula for measuring the extent to which the mass within a vector is globally spread out in a somewhat uniform fashion versus locally clumped within a few components.

If two random variables \( X \) and \( Y \) are not independent, then revealing the instantiation of \( X \) lowers the uncertainty (entropy) for \( Y \)’s distribution. The amount of reduction in entropy for \( Y \) is called the mutual information between \( X \) and \( Y \); we
get the same number if we measure the reduction in \(X\)'s entropy as a consequence of revealing \(Y\). This is a good measure of dependency between \(X\) and \(Y\), and we use it as the basis of a nice graph clustering algorithm in Section ??.

**Modular Arithmetic, Algebraic Fields.** It is good to be comfortable with arithmetic on integers modulo some integer \(n\). Here we identify a number only with its remainder when divided by \(n\), so for example 8 and 1 are the same number in arithmetic modulo 7. We write this fact as \(8 \equiv 1 \mod 7\), which reads “8 is congruent to 1, modulo 7”. After every arithmetic operation, we “reduce” the result by taking its remainder modulo \(n\) so it drops back into the range \(0 \ldots n - 1\). For example, in modulo 7 arithmetic, we obtain 2 when we add 4 and 5, when we subtract 6 from 1, and when we multiply 5 and 6. We often use arithmetic modulo \(n\) to ensure the numbers during a computation don’t grow too large.

In an equation like \(x + a \equiv b \mod n\), we can solve for \(x\) by adding \(-a\) to both sides, just like with ordinary arithmetic. Moreover, if \(n = p\) where \(p\) is a prime number, then remarkably division is also possible! In other words, if \(ax \equiv b \mod p\) and \(a \neq 0\), then there is a unique multiplicative inverse \(a^{-1}\) satisfying \(aa^{-1} \equiv 1 \mod p\) that we can multiply by both sides to solve for \(x\). For example, if \(3x \equiv 5 \mod 7\), then we can multiply both sides by 5 (the multiplicative inverse of 3, when working modulo 7) to obtain \(x \equiv 4 \mod 7\). We will use the existence of unique multiplicative inverses modulo a prime on several occasions in this book, notably when we study hashing in Chapter 7. To compute multiplicative inverses modulo a prime efficiently, we can use an extended version of Euclid’s algorithm, described in Section ??.

Since arithmetic modulo a prime supports addition, subtraction, multiplication, and division, as well as a few extra properties to which we are accustomed in normal arithmetic (e.g., addition and multiplication are commutative and associative, multiplication distributes over addition, etc.), this system of arithmetic is known as an algebraic field. Many mathematical algorithms that use only addition, subtraction, multiplication, and division (for example solving a system of linear equations or interpolating a polynomial) work perfectly well in any field, so just as you can use them for problems involving real numbers, you can also use them to solve problems involving arithmetic modulo a prime.

**Polynomial Root Bounds.** Consider a degree-\(n\) polynomial \(A(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0\) that is not identically zero. As a consequence of the fundamental theorem of algebra, we know that \(A\) can have at most \(n\) roots (values of \(x\) for which \(A(x) = 0\)). This holds in any field, including arithmetic modulo a prime \(p\). Since this system of arithmetic only involves the \(p\) different numbers 0, 1, \ldots, \(p - 1\), another way to state our polynomial root bound is by saying that for any degree-\(n\) polynomial \(A(x)\) that is not identically zero, at most an \(n/p\) fraction of all possible settings for \(x\) will result in \(A(x) \equiv 0 \mod p\). As it turns out, this bound generalizes to multivariate\(^{15}\) polynomials as well: if \(A(x_1, \ldots, x_k)\) is a multivariate polynomial of degree \(n\) (not identically zero), then at most an \(n/p\) fraction of all possible variable settings for \(x_1 \ldots x_k\) will result in \(A(x_1, \ldots, x_k) \equiv 0 \mod p\) [Short proof]. This bound typically comes into play in randomized algorithms, where we can say that a random choice for \(x_1 \ldots x_k\) is unlikely to be a root of \(A\) as long as

\(^{15}\) A multivariate polynomial is a polynomial involving several variables, such as \(A(x_1, x_2, x_3) = 7x_1x_2^2 + 3x_1^2x_3^2 - 2x_2x_3\). Here, the degree of a term is the sum of the exponents of the variables in the term (e.g., the degree of the term \(3x_1^2x_3^3x_3\) is \(2 + 3 + 1 = 6\), and the overall degree of a polynomial is the maximum term degree.
Problem 37 (Secret Sharing). This problem illustrates a simple yet powerful result that makes use of polynomial root bounds.

(a) Show that the $n$ coefficients of a degree-$(n-1)$ polynomial are uniquely determined if we specify the value of the polynomial at $n$ or more different points, even if we are performing arithmetic modulo a prime $p$. [Solution]

(b) Suppose $n$ people want to share a secret (some integer $a$ in the range $0 \ldots p-1$, where $p$ is prime). For extra security, they want to distribute information about the secret among themselves so that it takes at least $k$ people cooperating together to determine the secret. That is, any subset of $k$ or more people should be able to determine the secret, and any subset of $k-1$ or fewer people should not be able to learn anything about the secret. How can we use the preceding fact to accomplish this? [Solution]

Problem 38 (Verifying Matrix Multiplication). This problem serves as a nice conclusion for our chapter since it demonstrates an elegant combination of many of the techniques we have talked about so far. Suppose we have two $n \times n$ matrices $A$ and $B$ that we would like to multiply to obtain an $n \times n$ product matrix $C$. Matrix multiplication, as we will learn in Chapter ??, takes a substantial amount of time — $O(n^3)$ time to multiply matrices in the usual straightforward fashion, and $O(n^{2.3727})$ time using sophisticated and very complicated techniques. By contrast, we can multiply a matrix by a vector in only $O(n^2)$ time. This allows us to develop an remarkably simple randomized algorithm, initially due to the Latvian mathematician Rusins Freivalds, that verifies the result of a matrix multiplication faster than we know how to multiply matrices in the first place. The idea is simple, given three matrices $A$, $B$, and $C$ as input, we want to check whether $AB = C$. To do this, we pick a random vector $x$ whose $n$ components are all either 0 or 1 each with probability $1/2$, and we compare $ABx$ to $Cx$. Note that this takes only $O(n^2)$ time, since $ABx = A(Bx)$, so we can first multiply $B$ times $x$ and then multiply the resulting vector by $A$. If $ABx \neq Cx$, we know for a fact that $AB \neq C$. However, if $ABx = Cx$ we may suspect that $AB = C$ but there is some chance we are making a mistake. This approach also gives us a preview of the use of “hashing” (Chapter 7) to compare large complicated objects by first mapping them down to simpler objects.

Please show that $\Pr[ABx = Cx \mid AB \neq C] \leq 1/2$, so our algorithm has a probability of at most $1/2$ of mistakenly claiming that $AB = C$ when actually $AB \neq C$. As a hint, you may first want to consider the simpler problem of showing that $\Pr[a \cdot x = b \cdot x \mid a \neq b] \leq 1/2$ if $a$ and $b$ are length-$n$ vectors and $x$ is a randomly-chosen length-$n$ vector whose components are each independently set to 0 or 1 with probability $1/2$. Show also how you can achieve a high probability bound for correctness by running multiple times, or by using integers modulo a large prime $p$ instead of just zeros and ones. [Solution]
3. Sorting and Algorithm Analysis

In elementary school, students begin their studies by learning the fundamental topics of reading, writing, and arithmetic. In the study of algorithms, students begin by learning to sort. Given $n$ comparable elements (e.g., numbers or text strings), the sorting problem asks us to arrange them in nondecreasing order. Sorting is a truly fundamental algorithmic topic. It has been the focus of hundreds of publications in the computer science literature, and it is almost always the first substantial topic covered in any algorithms course. Vast amounts of computing power are spent solving sorting problems in practice, and sorting is also a key preprocessing step or subroutine for many more sophisticated algorithms.

This chapter serves not only as an introduction to sorting algorithms, but also as an introduction to fundamental techniques for designing algorithms and analyzing their correctness and running time. Sorting is the ideal domain for such a discussion, due to the wide variety of common sorting algorithms out there. We also study two problems closely related to sorting: selection (finding the $k$th largest element in an unsorted sequence), and topological sorting a partially-ordered sequence (where only some pairs of elements can be meaningfully compared).

We assume for simplicity that our input comes in an array $A[1 \ldots n]$, which we then want to permute so that $A[1] \leq A[2] \leq A[3] \leq \ldots \leq A[n]$. Many sorting algorithms can easily be adapted to work directly on linked lists with no degradation in running time. When sorting large records, it is often faster in practice to sort an array of pointers to these records, since this involves moving less memory around.

Most of the algorithms in this chapter are designed for the comparison-based model of computation, where we assume that input elements can be compared pairwise, and nothing more. If our input elements are numbers, we can also consider using the RAM or real RAM models. The real RAM turns to be not very different from the comparison-based model for sorting. We will soon show that there is an $\Omega(n \log n)$ worst-case lower bound for sorting in both of these models, so sorting algorithms running in $O(n \log n)$ time such as merge sort and quicksort are optimal in this setting. In the RAM model, since we can assume our input consists of integers, we will be able to design algorithms with input-sensitive running times that can conceivably run faster, as long as they are sorting integers that are sufficiently small. For example, radix sort only takes $\Theta(n)$ time to sort $n$ integers of size at most $n^c$, where $c$ is a constant.
3.1 Algorithm Design Techniques

Our discussion begins with three simple sorting algorithms that illustrate some of the most basic techniques of algorithm design. These will then serve as examples for how to argue both the correctness and running time of an algorithm.

3.1.1 Iterative Refinement: Bubble Sort

Iterative refinement involves starting with an arbitrary (probably incorrect) solution and repeatedly improving it with small modifications until it eventually becomes correct. Many prominent algorithms are based on this simple idea, including a wide range of heuristics for obtaining good solutions to hard problems (Chapter 13) and algorithms for solving optimization problems (Chapter ??).

At the heart of any iterative refinement algorithm is usually a subroutine that either proclaims a solution to be correct, or identifies some aspect of the solution that can be modified to make it more correct. For example, if we scan through an array \( A \) and realize it is not sorted, then there must be some adjacent out-of-order pair of elements \( A[i] > A[i + 1] \) that we can swap to make the array “more sorted” (we will formalize this notion in a few pages when we talk about inversions). This leads to the \( O(n^2) \)-time bubble sort algorithm, which repeatedly scans through our array, swapping any adjacent out-of-order pairs of elements it finds, stopping once the array becomes sorted. Its name reflects the way it causes small elements to slowly drift toward the front of the array just as bubbles drift to the top of a pool of water.

3.1.2 Incremental Construction: Insertion Sort

When sorting a stack of papers, you may instinctively use an insertion sort, where you maintain two stacks of papers: those sorted so far, and the remaining unsorted ones. In every step, you insert a paper from the unsorted stack into its proper
3.1. ALGORITHM DESIGN TECHNIQUES

location in the sorted stack. Insertion sort is an example of an algorithm design
technique called *incremental construction*, where a solution is built up step by step,
one element at a time.

When applying insertion sort to an array \( A[1 \ldots n] \), the “sorted stack” consists of
some prefix \( A[1 \ldots j] \) whose elements are in sorted order. We expand this prefix in
each iteration, as shown in Figure 3.1(a), by moving the next element \( A[j + 1] \) into
its proper location so as to leave \( A[1 \ldots j + 1] \) sorted. This is sometimes done by
taking \( A[j + 1] \) and repeatedly swapping it backwards as long as it is preceded by
a larger element. It takes \( O(n) \) time to process each successive element, for a total
running time of \( O(n^2) \).

We can also think of incremental algorithms from a recursive, rather than iterative,
point of view. For example, we could say that insertion sort first recursively sorts
\( A[1 \ldots n - 1] \), then inserts the final element \( A[n] \) into its proper location so the whole
array becomes sorted. The difference is only in how we choose to think about the
algorithm; both variants perform essentially the same operations and have identical
running times.

### 3.1.3 Divide and Conquer: Merge Sort

Many algorithms construct a large solution out of recursively-computed solutions to
smaller instances of the same problem. With incremental construction, we succes-
vively add one element at a time, taking a solution to a subproblem of size \( n - 1 \) and
somehow augmenting it with one additional element. By way of contrast, *divide and
conquer* algorithms tend to decompose a larger problem in a more “multiplicative”
fashion — for example, splitting a problem of size \( n \) into 2 subproblems of size \( n/2 \).

A popular sorting algorithm based on divide and conquer is *merge sort*, based on
the fundamental process of merging two sorted sequences into one larger sorted
sequence. Merging is quite straightforward — to merge two sorted stacks of paper,
you can repeatedly compare the front pages of both stacks, always selecting the
smaller for the next page in the merged stack. As shown in Figure 3.1(b), we
merge two arrays \( A \) and \( B \) the same way. We maintain pointers \( i \) and \( j \) to the
“front” elements of both arrays, and every iteration we select the smaller of these
to be the next element in the merged array, advancing its corresponding pointer.
Alternatively, from a recursive point of view, we select the smaller of \( A[1] \) and \( B[1] \)
to be first in the merged array, and the rest of the merged array is formed by
recursively merging the leftover parts of \( A \) and \( B \). It takes \( \Theta(n) \) time to merge two
arrays of combined length \( n \), since each step places one element into its correct final
position in the merged array.

To merge sort an array \( A \), we recursively merge sort the first half \( A[1 \ldots n/2] \) and
second half \( A[n/2 + 1 \ldots n] \), then merge the results together. If you prefer an
iterative outlook, the same process can be described as in Figure 3.2: regard an
\( n \)-element array as \( n \) adjacent 1-element sorted arrays, then merge these pairwise
to obtain \( n/2 \) adjacent 2-element sorted arrays, then \( n/4 \) adjacent 4-element sorted
arrays, and so on. Both processes do the same work, and differ only in whether
we prefer a “top down”, recursive perspective (arguably the more natural way to
describe a divide and conquer algorithm like merge sort) or a “bottom up”, iterative
perspective. Merge sort runs in \( \Theta(n \log n) \) time, as we shall see in a moment.
CHAPTER 3. SORTING AND ALGORITHM ANALYSIS

3.2 Arguing Correctness and Running Time

Although it may seem obvious that the three sorting algorithms above are correct, caution is advised — algorithms that seem correct at first glance may often fail for subtle reasons, and only with a clear mathematical analysis can one establish correctness beyond any reasonable doubt. We can argue correctness at many levels of detail. In this book, we tend to stay at a high level, although for mission-critical applications one can also argue correctness of an algorithm or computer program in excruciating detail.

In this section, we discuss techniques for analyzing correctness and running time. Correctness is the most important of the two, of course, since there is still value in a correct algorithm whose running time we do not fully understand, but a fast incorrect algorithm is typically of little use. Showing that an algorithm eventually terminates is also a key part of proving correctness. For iterative refinement algorithms, termination usually implies that the algorithm has reached a correct solution, so the entire correctness argument reduces to proving termination.

3.2.1 Correctness Proofs Based on Induction

Many correctness arguments are structured as proofs by induction. This is particularly well suited for algorithms that build large solutions out of recursively-
constructed smaller solutions, since it allows us to assume by induction on problem size that our algorithm will correctly solve any subproblem of strictly smaller size. Consider merge sort for example:

**Claim:** Merge sort correctly sorts any array of length \( n \).

**Proof:** This is easily proved using induction on \( n \). As a base case (a key part of any inductive proof!), correctness is trivially verified for \( n = 1 \). For \( n > 1 \), induction tells us that our recursive calls to merge sort will properly sort the first and second halves of the array. All that remains is to argue correctness of our merging algorithm, which we also accomplish using induction. Exactly how we apply induction in this case, however, depends on whether we are phrasing the merge as an iterative (i.e., loop-based) algorithm or as a recursive algorithm. Recall that a simple recursive way to merge two sorted lists \( A \) and \( B \) is to take the smaller of their initial elements as the first element in the merged list, and then to complete the output by recursively merging the left-over contents of \( A \) and \( B \). Here, correctness easily follows from induction on the combined length of \( A \) and \( B \): the first element we choose is clearly the smallest overall, and hence the correct element to place first in the merged list. We then claim by induction that the remainder of \( A \) and \( B \) will be merged correctly, since this constitutes a smaller problem instance.

**Iterative Algorithms and Loop Invariants.** Induction proofs are also often applied to iterative algorithms. Here, we usually apply induction on the number of iterations of our algorithm, and our inductive hypothesis is called a loop invariant. An invariant is a property that always holds at key points during an algorithm’s execution, and a loop invariant is a condition that remains true at the beginning of every iteration of a loop. As a simple example, consider randomly permuting the contents of an array \( A[1 \ldots n] \) as follows:

For \( j = 1 \ldots n \):
   - Let \( i \) be a random number in \( \{1, \ldots, j\} \)
   - Swap \( A[i] \) and \( A[j] \)

A suitable loop invariant here is that the elements in \( A[1 \ldots j-1] \) are equally likely to appear in any of their \( (j-1)! \) possible permutations. This is true as a base case when the loop begins, and one can show that it is maintained by each iteration of the loop [Easy proof]. Upon termination, when \( j \) reaches \( n+1 \), the invariant implies that we have indeed randomly permuted \( A \)’s contents.

**Problem 39 (Correctness of Iterative Merging).** Using an appropriate loop invariant, please give a short proof of correctness for the iterative approach for merging shown in Figure 3.1(b). [Solution]

**3.2.2 Different Ways of Adding up the Running Time**

There are several methods we can use to determine the total running time of an algorithm, depending on its structure.

**Solving a Recurrence.** For recursive algorithms, we usually compute running time by solving a recurrence. For example, if \( T(n) \) denotes the running time of merge sort on \( n \) elements, we know that \( T(n) = 2T(n/2) + \Theta(n) \), since we are performing two recursive sorts each on \( n/2 \) elements, followed by a \( \Theta(n) \) merge
operation. The solution of this recurrence gives the running time of merge sort, \( T(n) = \Theta(n \log n) \).

**Loop Counting.** For iterative algorithms, simple loop counting often suffices. As shown in Figure 3.2, we can think of merge sort as an iterative algorithm that takes \( n \) singleton elements and merges them pairwise to obtain \( n/2 \) sorted lists of length 2, then merges these pairwise to obtain \( n/4 \) sorted lists of length 4, and so on for \( \log n \) such phases. Since merging is a linear-time operation, each phase (merging \( n/k \) sorted lists of length \( k \)) takes \( \Theta(n) \) time, for a total of \( \Theta(n \log n) \) time\(^1\).

**Running Time Spent per Element.** Instead of adding up the total time spent in each step of an algorithm, summed over all steps during the algorithm’s execution, it is sometimes more convenient to consider the total running time spent on each individual input element, summed over all elements. This adds up the same amount of total work, just in a different, more convenient order. Using merge sort as our example again (Figure 3.2), it takes \( \Theta(n) \) time to merge two sorted arrays of combined length \( n \), which is \( O(1) \) time per element taking part in the merge. The total amount of work merge sort spends on a single element of data is therefore proportional to the number of merge operations in which the element takes part. How many merges happen to a single element? If you put yourself in the perspective of an element of data, you will find that every time you take part in a merge, you end up in a sorted subarray twice as large as before. This will happen \( \log n \) times before you end up in a sorted array of length \( n \). We therefore spend \( \Theta(\log n) \) time per element, for a total running time of \( \Theta(n \log n) \). This sort of “per element” outlook on running time can be highly useful in the analysis of many algorithms.

### 3.2.3 Inversions and Potential Arguments

Another common and powerful technique for proving correctness and running time is the use of a potential function argument. A potential function maps the state of our algorithm to a nonnegative integer. If we can guarantee that this must decrease during each iteration of the algorithm, then termination is inevitable. This approach is commonly used with algorithms based on iterative refinement, where our potential function usually reflects the amount of “incorrectness” inherent in the algorithm’s current solution. There is a natural physical analogy suggested by the use of the term “potential”: we can think of a potential function as telling us the amount of “energy” stored in our current state, with our algorithm acting in the role of gravity, pulling in a direction that decreases this potential energy.

For sorting, a natural potential function is the number of inversions in our array. A pair of elements \( A[i] \) and \( A[j] \) with \( i < j \) constitutes an inversion if \( A[i] > A[j] \); that is, the elements are ordered incorrectly with respect to their positions. A sorted array has no inversions, and a reverse-sorted array has the maximum possible number of inversions, \( \binom{n}{2} \), since every pair is an inversion. Inversions show up often in the study of permutations and sorting, and since inversion count is a natural way to measure “unsortedness”, it is often a good choice for a potential function. By

---

\(^1\)Recall from Section 2.3 (Figure 2.5) that solving a recurrence involves adding up the work done by an algorithm at each “level” of recursion. If you look at Figure 3.2 closely, you will also see a tree of recursive subproblems whose work we have added up level by level. Hence, in this case, the analysis of our iterative outlook on merge sort turns out to be just another way of doing the same math we used to solve the merge sort recurrence.
swapping two adjacent out-of-order elements \( A[i] > A[i + 1] \), we correct a single inversion, leaving all others unchanged. For example, this tells us that bubble sort must terminate, since each scan through the array (except the last) performs at least one such swap, decreasing the inversion count.

Potential functions are often useful for analyzing running time. For example:

- **Bubble Sort.** Let \( A' \) denote the sorted version of our array \( A \), and suppose our potential function tells us the size of the largest suffix of our array such that \( A[j \ldots n] = A'[j \ldots n] \) (i.e., the number of elements at the end of \( A \) that are in their correct final positions). Each scan of bubble sort increases this potential by at least one. The first scan pulls the largest element, through repeated swaps, to its correct final position at the end of the array, then the second scan pulls the second-largest element back to the second-to-last position, and so on. Since our potential cannot exceed \( n \), we perform at most \( n \) scans, each taking \( \Theta(n) \) time, for a total running time of \( O(n^2) \).

- **Insertion Sort.** Consider inversion count as a potential function. Since insertion sort has a fixed \( \Theta(n) \) overhead for scanning the array and then corrects one inversion for each backward swap it makes, the running time is \( \Theta(n + I) \) time, where \( I \) is the number of inversions in our input array. This is still \( \Theta(n^2) \) in the worst case, but can potentially be much faster on nearly-sorted arrays\(^2\), where it can even outperform merge sort. For example, if \( I = \Theta(n) \), then insertion sort runs in \( \Theta(n) \) time, while merge sort runs in \( \Theta(n \log n) \) time (note that merge sort always runs in \( \Theta(n \log n) \) time, even when given an already-sorted input).

When we study amortized analysis in the next chapter, we will make somewhat more sophisticated running time arguments by using potential functions that can both increase and decrease, rather than moving in a single monotonic direction.

**Problem 40 (Inversions and Invariants).** Suppose we wish to design a sorting algorithm in which the only operation available is a cyclic shift of 3 consecutive elements. For example, a left cyclic shift on the middle 3 characters of the string ‘SORTING’ yields ‘SOITRNG’. Using inversions and invariants, please characterize the inputs for which sorting is possible in this setting, and show how to sort them in \( O(n^2) \) time. [Solution]

### 3.3 Lower Bounds

Bubble sort, insertion sort, and merge sort are all comparison-based algorithms. The comparison model is ideal for sorting, allowing us to develop general-purpose algorithms that can sort any type of comparable data (e.g., integers, real numbers, text strings, etc.). However, this generality has a price, as there is an \( \Omega(n \log n) \) lower bound on the worst-case running time of any comparison-based algorithm for sorting \( n \) elements.

\(^2\)Later in problem 105, we will design an “adaptive” variant of insertion sort whose running time scales gracefully as a function of \( I \) between \( \Theta(n) \) for a sorted array (with no inversions) up to \( \Theta(n \log n) \) for a reverse-sorted array (with the maximum possible number of inversions).
3.3.1 Modeling Algorithms by Decision Trees

We can easily show an upper bound of $O(n \log n)$ on the running time required to sort by demonstrating an $O(n \log n)$ algorithm, like merge sort. Arguing a lower bound is trickier, however, since this involves proving that any comparison-based sorting algorithm, no matter how bizarre or clever, must take $\Omega(n \log n)$ steps in the worst case. To make such a claim, we first show how any comparison-based sorting algorithm can be abstractly represented using a simple algorithmic model called a decision tree, then we show that any decision tree must make $\Omega(n \log n)$ comparisons in the worst case in order to properly sort $n$ elements.

A sorting algorithm in the form of a decision tree is shown in Figure 3.3. At every step, it compares two elements and branches based on the result. The more comparisons the algorithm makes, the more it learns about the ordering of the input elements. When we reach a leaf, the algorithm terminates and declares how to rearrange the input in sorted order. The height of the tree is the number of comparisons required in the worst case. Any deterministic (not randomized) comparison-based sorting algorithm can be abstractly represented this way, as a series of decision trees, one for each input size $n$.

If we are sorting $n$ distinct input elements, there are $n!$ different orderings we might receive as input. Each one needs to be re-arranged differently in order to sort them all properly. Our decision tree therefore must have at least $n!$ leaves. Otherwise, two different input orderings would find their way to the same leaf\(^3\), and one of them would therefore be sorted incorrectly, since the same permutation would be applied to both. Since a decision tree of height $h$ can have at most $2^h$ leaves, we need $2^h \geq n!$, so $h \geq \log n! = \Theta(n \log n)$ by Stirling’s approximation. The height of our decision tree (i.e., the number of comparisons needed to sort $n$ elements in the

\(^3\)This is an example of the so-called pigeonhole principle: if there are more pigeons than pigeonholes, then two pigeons must end up in the same pigeonhole. This obvious yet important concept is used in many combinatorial proofs.
worst case) must therefore be $\Omega(n \log n)$.

**Problems Involving Equality Testing.** The result of a comparison is one of three outcomes: $=$, $<$, or $>$. Since any sorting algorithm must still properly sort in the special case where we have distinct elements, it suffices to consider only the cases $<$ and $>$ for the argument above. Equality plays a more central role in several other problems, however. For example:

- **Element Uniqueness.** Given $n$ elements, the *element uniqueness problem* asks if they are all distinct or if there exist two equal elements.

- **Set Disjointness.** Given two sets $A$ and $B$ each with at most $n$ elements, the *set disjointness problem* asks whether $A$ and $B$ share any elements in common (i.e., we want to test if $A \cap B = \emptyset$).

- **Set Equality.** Given two sets $A$ and $B$ with $|A| = |B| = n$, the *set equality problem* involves testing whether $A = B$.

For each of these, we can still use decision trees to argue an $\Omega(n \log n)$ worst-case lower bound in the comparison model, although they seem to require slightly more nuanced proofs than the one above for sorting [Details]. Interestingly, we will see in Chapter 7 how to use “hashing” to solve all three problems in $\Theta(n)$ expected time in the RAM model, illustrating well the importance of our underlying computational model.

**Problem 41 (Lower Bounds for Group Testing).** Suppose up to $k$ out of $n$ individuals have a particular disease (typically with $k$ much smaller than $n$). You would like to identify them with a minimum number of expensive tests. Of course, you could test all $n$ individuals separately, but it can be much less costly to test groups of individuals. If we group together, say, blood samples from a set $S$ of individuals, the outcome of a single test tells us whether (i) at least one individual in $S$ has the disease, or (ii) nobody in $S$ has the disease. You may have seen problems of this sort in recreational mathematics involving the identification of coins of incorrect weight by weighing groups of coins on a balance. Your solution can be “adaptive” in that results from one aggregate test can influence your choice of which group to test next (we consider the more challenging non-adaptive case later in problem 118). Please show a simple algorithm that identifies all of the sick individuals with $O(k \log n)$ tests, and use a decision tree argument to show that $\Omega(k \log ^2 \frac{n}{k})$ tests are necessary in the worst case. [Solution]

### 3.3.2 Randomized and Algebraic Decision Trees

To extend our $\Omega(n \log n)$ worst-case lower bound to *randomized* comparison-based sorting algorithms, let us think of a randomized algorithm abstractly as just a probability distribution over deterministic algorithms. That is, imagine taking all the random bits our algorithm will consume during its lifetime, and using these up front to select a deterministic algorithm from a huge table. After this initial selection, the algorithm is purely deterministic, since all of its formerly-random behavior has been fully specified. Owing to this insight, we can model a randomized

---

4We find it convenient to use concepts from topological sorting in our proofs here, so you may want to read ahead to Section 3.8 before listening to the proof details.
n-element comparison-based sorting algorithm as a probability distribution over
decision trees. Since each of these trees must still sort properly (i.e., it must have
Ω(n log n) height), our Ω(n log n) worst-case bound on comparisons still applies.

Of course, we rarely apply worst-case analysis to a randomized algorithm; it is far
more typical to look at expected running time. Later in Section 12.5.4, we will see
how to use techniques from game theory to prove lower bounds on the expected
performance of randomized algorithms, and we will establish an Ω(n log n) lower
bound on the expected running time of any randomized n-element comparison-
based sorting algorithm. This bound is closely related to the “average case” result
that any deterministic comparison-based algorithm must take Ω(n log n) expected
time to sort n elements that have been permuted at random. Showing this is
straightforward, by modifying the proof above to show that a decision tree with n!
leaves must have Ω(n log n) average leaf depth.

It becomes more challenging to argue lower bounds when we move away from the
comparison model towards a more general model like the real RAM, where input
data is now numeric. For example, if we consider the element uniqueness problem
defined above on a set of n real numbers A[1...n], we could conceivably solve this
problem by evaluating the product of A[i] – A[j] over all pairs (i, j) and making
only a single comparison to test if this product is equal to zero. While it may
seem unlikely that this approach could possibly yield an efficient solution, the
point is that it no longer suffices to consider only comparisons in order to obtain a
meaningful lower bound.

To help us obtain lower bounds in the real RAM model, we turn to a generalization
of the decision tree known as the algebraic computation tree, which models any real
RAM algorithm by a tree containing both branching nodes at which we perform
comparisons, and non-branching nodes at which we perform elementary operations
such as addition, multiplication, division, etc. Further discussion of this model is
beyond the scope of this book. However, one can show that sorting, as well as
the other three example problems above (element uniqueness, set disjointness, and
set equality) all have Ω(n log n) worst-case lower bounds in the real RAM model
through the use of algebraic computation trees.

3.3.3 Lower Bounds via Reductions

Now that we have non-trivial lower bounds for a small handful of problems, we
can transfer these to other problems using reductions, somewhat like we used
polynomial-time reductions to transfer hardness results from one problem to an-
other in Section 1.6.3.

As an example, consider the problem of rearranging an array to group equal elements
into contiguous blocks (not necessarily appearing in sorted order). An instance of
the element uniqueness problem can be reduced to this problem, by grouping equal
elements and then performing a Θ(n) scan to test whether each element is distinct
from its neighbors. If we could group equal elements in time faster than O(n log n),
the same would therefore be true for element uniqueness, contradicting its Ω(n log n)
worst-case lower bound in the comparison and real RAM models. Hence, it must

5 Surprisingly, even though it seems to involve Θ(n^2) terms, this product can be evaluated in
only Θ(n log^2 n) time using the Fast Fourier Transform, as we shall see in Chapter ??.
3.4. QUICKSORT

A: \texttt{partition}(A, i):
≥ pivot ≤ pivot

"pivot" A[i]

Figure 3.4: Illustration of the result of partitioning an array \(A[1 \ldots n]\) about a pivot element \(A[i]\). Although have drawn the output of the partition separate from the input, note that partitioning is usually done “in place” in the same memory space as the original array \(A[1 \ldots n]\).

also take \(\Omega(n \log n)\) time in these models to group equal elements.

Problem 42 (Reduction Practice). Please use reductions from sorting or other problems above (element uniqueness, set disjointness, set equality) to show \(\Omega(n \log n)\) worst-case bounds for the following problems.

(a) Consider the following problems in the comparison model: (i) counting the number of occurrences of a most-frequently-occurring element in an \(n\)-element array, and (ii) given two sets \(A\) and \(B\) with \(|A| + |B| = n\), compute \(A \cap B\), \(A \cup B\), or \(A \setminus B\). [Solution]

(b) Many problems in computational geometry inherit lower bounds from sorting, element uniqueness, or their relatives. Consider the following problems in the real RAM model, since they involve points with numeric coordinates: (i) the closest pair problem asks us to find the closest pair of points within a set of \(n\) points in the plane, and (ii) the congruence testing problem gives us two \(n\)-point sets \(A\) and \(B\) in the 2D plane, and asks whether \(B\) can be obtained from \(A\) by applying a rigid transformation — a translation plus a rotation plus (possibly) a reflection. [Solution]

3.4 Quicksort

One of the most popular and widely-used sorting algorithms is quicksort, which like merge sort is based on the principle of divide and conquer. In fact, quicksort is almost symmetric to merge sort in its operation: merge sort performs two recursive sorts followed by a linear-time merge, whereas quicksort performs a linear-time partition followed by two recursive sorts. Partitioning is shown in Figure 3.4. Given a specific array element (known as a pivot), we rearrange the array into a block of elements less than or equal to the pivot, followed by the pivot element, followed by a block of elements greater than or equal to the pivot. The pivot element will
Figure 3.5: Two methods for partitioning an array in place. In (a), we scan pointers $i$ and $j$ inward from the ends of the array, stopping when $A[i]$ is larger than the pivot and $A[j]$ is smaller than the pivot. We then swap $A[i]$ and $A[j]$ and continue scanning inward, until the pointers meet. in (b), we scan two pointers from left to right. In each step, if $A[j + 1]$ is larger than the pivot, we simply advance $j$; otherwise, we swap $A[i + 1]$ and $A[j + 1]$ and advance both $i$ and $j$.

then be in its correct final location, and quicksort finishes by recursively sorting the subarrays left and right of the pivot.

Partitioning is very straightforward if we allocate a temporary buffer the same size as our array to hold the output. In this case, we simply scan through our array and copy out all the elements less than the pivot to the beginning of the buffer, and all those greater than the pivot to the end of the buffer. However, partitioning is more commonly performed “in place” without the need for an auxiliary buffer. Two common approaches for doing this are explained in Figure 3.5.

**Choosing a Good Pivot.** The tricky aspect of quicksort is how to choose the pivot. Ideally, we should choose the median element, which we will shortly learn how to find in only $\Theta(n)$ time. Since the median evenly splits our array into two recursive subproblems of size $n/2$, the running time of quicksort is then described by the recurrence $T(n) = 2T(n/2) + \Theta(n)$, with the $\Theta(n)$ term representing both the time spent finding the median and the time spent partitioning. In total, this solves to $T(n) = \Theta(n \log n)$. Unfortunately, the $\Theta(n)$ running time required to find the median element has a rather high hidden constant, so this version of quicksort is likely to run significantly slower than merge sort in practice, even though the two share the same asymptotic running time.

Our partition doesn’t need to be perfectly balanced. Even if we could only guarantee at least a 1% – 99% split we would still end up with a running time of $\Theta(n \log n)$, albeit with a much higher hidden constant (since the solution of $T(n) = T(n/100) + T(99n/100) + \Theta(n)$ is still $\Theta(n \log n)$). On the other hand, if we always pick a very bad pivot like the minimum or maximum element, the running time can be as slow as $\Theta(n^2)$, since each $\Theta(n)$ partition operation makes very little progress, leaving us with a subproblem of size $n - 1$.

A simple but flawed strategy for selecting a pivot element is just to pick whichever element happens to be at the beginning or end of the array. Unfortunately, if our array is already sorted (a common occurrence in practice), this is a very bad choice. A slightly more popular heuristic is to look at the elements that appear first, last, and in the middle of the array and to use the median of these three as a pivot. This “median of three” approach can still lead to a running time of $\Theta(n^2)$ on contrived inputs, but in practice it has been observed to perform reasonably well.
Randomized Quicksort. Suppose we pick the pivot element at random\(^6\), an approach that seems intuitively sensible since it should generate reasonable partitions most of the time. Indeed, this variant, known as randomized quicksort, runs in \(\Theta(n \log n)\) time both in expectation and with high probability. Randomized quicksort is simple to implement and competitive with merge sort in practice.

To show that the expected running time of randomized quicksort is \(\Theta(n \log n)\), we use linearity of expectation to decompose its running time into a sum of much simpler random variables. There are several nice ways to do this:

- We can add up the total number of comparisons performed by the algorithm (which is asymptotically the same as its running time), by defining an indicator random variable for each pair of elements taking the value 1 if they are compared, and 0 otherwise. [Details]
- We can add up the total expected work spent on subproblems of different sizes. That is, we take the sum, over all \(k = 1\ldots n\), of the expected work spent partitioning all subproblems of size \(k\) seen during execution, the expected number of which is surprisingly easy to compute. [Details]
- We can show that \(O(\log n)\) expected time is spent on each individual element in our array, using the randomized reduction lemma. [Details]

For a high probability bound, we observe from the last approach that the randomized reduction lemma also tells us that we spend \(O(\log n)\) time on each element with high probability. By taking a union bound over all elements, the total running time is therefore also \(O(n \log n)\) with high probability.

Problem 43 (Matching Nuts and Bolts). We are given \(n\) nuts of different sizes and \(n\) corresponding bolts, and we must match these together, determining for each nut the unique bolt of the same size. However, it is difficult to compare two nuts or two bolts — it is only possible to compare a nut against a bolt to see if the nut fits the bolt, if it is too small, or if it is too large. Describe a randomized algorithm running in \(O(n \log n)\) time with high probability that will properly match the nuts and bolts. [Solution]

Stopping Early. Quicksort spends a lot of time on recursive function call overhead for very small subproblems. Therefore, a common trick to improve performance in practice is to “bottom out” of its recursion earlier by leaving sufficiently small arrays (say, with at most 4 elements) unsorted. This gives a final array that is nearly sorted, to which we apply insertion sort as a postprocessing step.

Quicksort and Binary Search Trees. In Chapter 6, we will learn about a powerful and versatile data structure called a binary search tree, which essentially encodes the recursive tree of all subproblems generated by quicksort. By saving this information, we can “dynamize” the sorting process, quickly computing changes to a sorted ordering in response to insertion or deletion of elements. Many fundamental algorithms related to sorting, such as binary search, quicksort, and quickselect (discussed shortly) are elegantly expressed in the context of binary search trees; we postpone further discussion until Chapter 6.

\(^6\)Or, equivalently, we can use a simple deterministic pivoting strategy such as choosing the first element in the array, after randomly permuting the array as a preprocessing step.
3.5 In-Place Sorting

One advantage of quicksort over merge sort is the fact that quicksort can run in place. That is, it rearranges the contents of the input array using only the memory space allocated for the array, and almost no auxiliary storage. By contrast, merging $n$ elements seems to require that we allocate a temporary array of size $n$ to hold the output of the merge.

In modern times when computer memory is relatively cheap, there tends to be less emphasis on the importance of space versus running time. Indeed, time rather than space is typically the limiting resource for most algorithms in practice. However, for the case of sorting, memory usage tends to be studied a bit more closely, since many applications involve sorting very large data sets.

The strictest definition of an “in-place” algorithm would allow $O(1)$ extra words of memory in addition to the memory occupied by the input. However, this can make the implementation of recursive algorithms difficult, since there is often not enough stack space to store the state of unfinished recursive subproblems. Therefore, we typically call an algorithm “in-place” if it uses only $O(\log n)$ extra words of memory.

Our earlier analysis of randomized quicksort showed that its stack depth is $O(\log n)$ with high probability, although the worst case is still $\Theta(n)$ if we have phenomenally bad luck in our selection of pivot elements. If we want to ensure $O(\log n)$ stack depth in the worst case irrespective of pivot quality, a clever trick is to have quicksort always recurse on the smaller of its two subproblems first. Since each such recursive call halves the size of our current subproblem, this limits recursion depth to $O(\log n)$. The second recursive call in quicksort is what is known as tail recursion. Since it is the last thing done by the quicksort function, there is no need for additional stack space to enact this call, since we don’t need to save the state of the current function invocation. We simply re-use the same stack frame from the current invocation, effectively jumping right back to the start of the quicksort function with new parameters.

Problem 44 (In-Place Merge Sort). Standard merge sort does not operate in place, since it merges two arrays of combined length $n$ into a newly-allocated memory buffer of length $n$. In this problem, your goal is to develop an in-place $O(n \log n)$ sorting algorithm based fundamentally on merging (it is of course easy to achieve this goal using other methods like quicksort). As a hint, your algorithm may be need to be somewhat “asymmetric”, either by splitting into subproblems that are different in size, or by performing different tasks on different subproblems. As another hint, you can use the standard merging procedure with yet-unsorted parts of an array serving as scratch space. For example, if $A$, $B$, $C$, and $D$ denote four quarters of an array, you can merge $A$ and $B$ together into the space occupied by $C$ and $D$, while the elements of $C$ and $D$ become scrambled as they are swapped back into the space formerly occupied by $A$ and $B$. [Solution]

3.6 Stable Sorting

A sorting algorithm is stable if it leaves equal elements in the same relative order as they were prior to the sort. This is useful when sorting large multi-field records. For example, consider the email messages in your inbox, each with a “from” address, a
3.7 SELECTION

If you sort according to date, and then stably by “from” address, then messages with the same “from” address would remain sorted by date. Bubble sort, insertion sort, and merge sort are all stable, provided we implement them carefully. With quicksort, all methods we know for performing fast (linear time) in-place partitioning seem inherently unstable, so stability seems hard to achieve without sacrificing in-place operation. Stability and in-place operation seem somewhat at odds with each-other when it comes to fast sorting algorithms: merge sort is stable but not in-place, and quicksort, in-place merge sort (Problem 44), and heap sort (introduced in Chapter 5) are in-place but not stable. The question of whether stable and in-place sorting is possible in the comparison-based model in \( O(n \log n) \) time remained open until the late 1970s, when algorithms fulfilling all of these requirements were finally discovered. However, to this day, all such algorithms remain extremely complicated. By contrast, as we see in the next problem, we can sort stably and in place in a very simple and elegant fashion if we are willing to settle for an \( O(n \log^2 n) \) running time.

**Problem 45 (Stable and In-Place Sorting).** In this problem we construct variants of merge sort and quicksort that are stable, in place, and also reasonably efficient. These are the author’s favorite sorting algorithms since they provide a very elegant demonstration of the power of the divide and conquer principle.

(a) To begin with, consider the problem of swapping two adjacent blocks in memory in place. That is, we have an array \( A[1\ldots n] \) in which we identify a left block \( L = A[1\ldots k] \) and a right block \( R = A[k+1\ldots n] \), and we would like to rearrange the contents of the array from \( LR \) to \( RL \), with minimal additional memory usage. How can we do this in \( \Theta(n) \) time? Note that \( L \) and \( R \) do not necessarily have the same size. [Solution]

(b) Using divide and conquer, design an \( O(n \log n) \) stable in-place algorithm for merging. Building on this, we obtain a \( O(n \log^2 n) \) stable in-place version of merge sort. The result from (a) may be of help. [Solution]

(c) Using divide and conquer, design an \( O(n \log n) \) stable in-place algorithm for partitioning. Building on this, we obtain a \( O(n \log^2 n) \) stable in-place version of randomized quicksort. The result from (a) may again be of help. [Solution]

(d) Using the \( O(n \log n) \) stable in-place merging algorithm from part (b), give a simple alternative solution to problem 44. As a hint, start by dividing your array in half. [Solution]

Any sorting algorithm can be made stable by using additional memory (thus sacrificing in-place operation). To do this, we augment each element with its initial index within the array and modify our comparison operator to break ties between equal elements using these indices.

3.7 Selection

Selection is a close relative of the sorting problem, asking us to find the \( k \)th largest element (some might say \( k \)th smallest) in an unordered array \( A[1\ldots n] \). This element is said to have rank \( k \), and is also called the \( k \)th order statistic of the sequence.
Examples include \( k = 1 \) (the minimum), \( k = n \) (the maximum), and \( k = n/2 \) (the median, although if \( n \) is even, there is no unique median, and elements of ranks \( k = n/2 \) and \( k = n/2 + 1 \) could both rightfully be called “medians”).

Selection is easy to solve in \( O(n \log n) \) time by first sorting our array and then outputting \( A[k] \). For the special cases \( k = 1 \) or \( k = n \), it is easy to perform selection in \( \Theta(n) \) time by simply scanning the array while keeping track of the smallest (or largest) element we see. Selection at other ranks is less trivial, although we shall see in this section how to select for the element of any rank \( k \) in only \( \Theta(n) \) time. The median element is a particularly useful case, since it is commonly used as a splitting point in divide-and-conquer algorithms like quicksort to break a large problem two equal pieces.

**Quickselect.** We can solve the selection problem for any \( k \) in \( \Theta(n) \) expected time using quickselect, a close relative of randomized quicksort. We first partition \( A \) about a randomly-chosen pivot element\(^7\), after which we know the rank \( r \) of the pivot, since the pivot ends up in its correct final location if the array were sorted. If we are lucky enough that \( k = r \), the pivot is the answer and we are done. Otherwise, by comparing \( k \) with \( r \) we can determine whether the element we seek lies on the left or right side of the pivot: if \( k < r \), we recursively select for the element of rank \( k \) from subarray \( A[1 \ldots r - 1] \), and if \( k > r \) we recursively select for the element of rank \( k - r \) in the subarray \( A[r + 1 \ldots n] \). Quickselect is nearly identical in structure to randomized quicksort, except we recurse on only one of the two subarrays created by the partition operation, not both. This difference is sufficient to give only a \( \Theta(n) \) expected running time\(^8\). [Proof]

**Deterministic Selection.** Using quickselect in conjunction with a clever but somewhat complicated approach for choosing suitable pivots deterministically, we can also solve the selection problem for any \( k \) in \( \Theta(n) \) worst-case time. This is a nice result in theory, but since the resulting algorithm is rather complicated and also involves a large hidden constant, quickselect is usually the preferred method in practice. [How to perform deterministic selection in linear time]

**Problem 46 (Selection of Multiple Order Statistics).** The traditional selection problem involves computing one order statistic in a length-\( n \) array. Suppose instead that we have a list of \( k \) order statistics \( s_1 \leq s_2 \leq \cdots \leq s_k \) that we would like to find. We can clearly compute all of these in \( O(nk) \) time by applying a linear-time selection algorithm for each one individually. However, please show how to select for all \( k \) order statistics in only \( O(n \log k) \) time. As an extreme case, if we set \( s_j = j \) for \( j = 1 \ldots k = n \), this boils down to essentially sorting, in \( O(n \log n) \) time. [Solution]

**Problem 47 (In-Place Selection from Read-Only Memory).** We wish to perform selection from an array stored in read-only memory in an in-place setting. Rearranging the array elements is therefore prohibited, as is using more than \( O(\log n) \) auxiliary storage. Give a randomized algorithm for this problem that runs in \( O(n \log n) \) time with high probability. Interestingly, we do not know how to achieve a similarly-fast running time with a purely deterministic algorithm. [Solution]

\(^7\)It is important to note that partitioning on the median is not an option here (as with quicksort), since finding the median requires selection, the problem we are currently trying to solve!

\(^8\)This is one of the few randomized algorithms we will study where an expected running time analysis gives a stronger bound than a high probability analysis. If we want a high probability result for quickselect, \( O(n \log n) \) is actually the best bound we can claim (and it is trivial to show this, since the algorithm is performing a subset of the work of randomized quicksort).
3.8 Topological Sorting

So far, we have made the fundamental assumption that every pair of input elements is comparable. In this case, we say our input is a **totally-ordered set** or a **total ordering**. In many cases, however, our input may instead be a **partially-ordered set** (poset, also called a **partial ordering**), where only certain pairs of input elements can be meaningfully compared. The result of a comparison in a poset can be `<`, `>`, or “incomparable”.

The poset in Figure 3.6(a) shows constraints governing the order in which you can put on different articles of clothing. It also shows how we can represent a poset by a **directed acyclic graph**, or DAG. A directed path in this graph between two elements $i$ and $j$ indicates that $i$ is less than $j$ (in our example, this means $i$ must precede $j$ when getting dressed). Our graph cannot have any directed cycles, since if two nodes $i$ and $j$ were on a directed cycle, there would exist directed paths from $i$ to $j$ and also from $j$ to $i$, leaving it unclear which element is smaller.

In the example above, an edge from underwear to shoes is unnecessary since it is implied by transitivity: underwear is less than pants, which is in turn less than shoes. When we draw the DAG representation of a poset, we typically omit as many implied edges as possible to simplify the diagram. The DAG with the fewest edges that represents a given poset is known as the **transitive reduction** of the poset, and the DAG with all implied edges present is called the **transitive closure**. We will see how to compute both of these in Chapter ??.

Every DAG can be **topologically sorted** to produce a total ordering of its nodes (known as a **topological ordering**) that is consistent with the partial ordering implied by the DAG. As seen in Figure 3.6(b), all edges point consistently from left to right when we arrange the DAG according to its topological ordering. Many topological orderings may be valid for a given DAG; as an extreme example, any ordering is valid if our DAG has no edges.
Topological orderings have many applications. The instance shown in Figure 3.6 can be viewed as a scheduling problem, where nodes represent tasks and edges represent precedence constraints between these tasks; DAGs arise commonly in this setting. A topological ordering here gives a linear task ordering that respects all of the precedence constraints. Just as sorting can be a useful preprocessing step for many algorithms, topological sorting is a common preprocessing step for algorithms dealing with DAGs. We will see several examples of this when we study graphs in greater detail later in the book.

There are several ways to topologically sort a DAG in linear time, which in this case means $O(m + n)$ time if the DAG has $n$ nodes and $m$ edges. We discuss one method here and another, based on a simple graph algorithm called depth-first search, in Chapter ???. It is easy to show that every DAG always has at least one “source” node, with no incoming edge. If not, you could start at any node and walk backward, moving from one node to the next by always stepping backward along an arbitrary incoming edge. Since the graph has a finite number of nodes, this process would eventually revisit a node and close a directed cycle, thereby contradicting the fact that the graph is acyclic. Any source node is safe to place at the beginning of a topological ordering, since no other node needs to come before it. Therefore, we can generate a valid topological ordering by repeatedly finding and removing source nodes. With care, we can implement this in linear time. \[Details\]

### 3.9 Sorting Integers

It is finally time to depart from the familiar comparison-based model and consider RAM algorithms for sorting integers in the range $0 \ldots C - 1$. If $C$ is small, this allows us to sort faster than by using comparison-based algorithms.

#### 3.9.1 Counting Sort

Suppose that we have an array $A[1 \ldots n]$ containing only zeros and ones. We can easily sort $A$ in linear time by simply counting the zeros and ones, and then building a new sorted array containing the appropriate number of each.

This idea generalizes easily to the case where $A$ contains integers in the range $0 \ldots C - 1$. Here, we use an auxiliary array $N[0 \ldots C - 1]$, where $N[v]$ counts the number of copies of the value $v$ in $A$. To build $N$ after initializing its entries to zero, we increment $N[A[i]]$ for every $i = 1 \ldots n$. We can then scan through $N$ to re-build $A$ in sorted order. The resulting sorting algorithm, known as counting sort, runs in $\Theta(n + C)$ time, which is linear time as long as $C = O(n)$. With care, we can even implement it in a stable fashion. \[Implementation details\]

#### 3.9.2 Radix Sort

Using multiple invocations of counting sort, we can build a more powerful sorting algorithm known as radix sort that sorts in linear time if $C = O(n^c)$ for some constant $c$. 

---

DRAFT. Not for redistribution. Copyright © Brian C. Dean
3.9. SORTING INTEGERS

Radix sort is illustrated in Figure 3.7. We first write our integers down in some number base, or radix (hence the name of the sort), which in our example is base 10. We then sort according to each successive digit position, starting with the least significant. Radix sort actually originated as mechanical technique for sorting stacks of punched cards. Each card contained a number written in binary encoded by a series of punched and non-punched holes, and the cards were repeatedly fed through a mechanical sorting device, once for every digit, and sorted by separating cards with punched digits from those with non-punched digits.

The subroutine we use to perform each of the digit sorts can be any stable sorting algorithm, although we use counting sort since it is the natural choice for sorting small integers. If we ideally write our integers in base \( n \), then each digit assumes values from 0...\( n - 1 \) and each individual counting sort runs in \( \Theta(n) \) time. Our numbers will have at most \( \log n \) digits, which is constant as long as \( C = \Theta(n^c) \) for some constant \( c \). The total running time is therefore linear as long as \( C = \Theta(n^c) \) with \( c \) constant. Stability is crucial for our individual digit sorts, since if two numbers agree in their most significant digit (our final sorting pass), we want them to remain ordered properly by their lower-order digits, as they will be at this point thanks to induction.

**Problem 48 (Sorting Fractions).** Radix sort can sort \( n \) integers of magnitude \( n^c \) for \( c = \Theta(1) \) in linear time. Please show how to sort \( n \) fractions \( \frac{a_1}{b_1}, \frac{a_2}{b_2}, \ldots, \frac{a_n}{b_n} \) also in linear time, where the \( a_i \)'s and \( b_i \)'s are integers in the range 1...\( n^c \) for \( c = \Theta(1) \). As a hint, can you convert this problem back to one involving sorting small integers? [Solution]

---

Subtle point: A common assumption with the RAM model of computation is that each word contains \( \Theta(\log n) \) bits, which is exactly large enough to hold an integer of magnitude \( n^c \) for \( c = \Theta(1) \). If the \( n \) numbers being radix sorted each fit into a single \( \Theta(\log n) \)-bit machine word, then radix sort therefore takes only \( \Theta(n) \) time. Furthermore, if we plan to sort larger numbers on a machine with word size \( \Theta(\log n) \), it takes \( \Theta(n \log^2 C) \) words to store \( n \) numbers of magnitude \( C \), matching the running time for radix sort (so this running time cannot be improved, since it takes the same amount of time just to examine all the input words). The only way to allow for stronger RAM-based sorting algorithms is to consider slightly larger word sizes (which is common practice in the sorting literature). For word sizes slightly larger than \( \Theta(\log n) \) bits, it is still unknown if linear-time sorting is possible – see the endnotes for further discussion.
Problem 49 (Sorting Variable-Length Strings). Standard radix sort works from the least significant digit to the most significant. In this problem, we do the opposite, demonstrating a variant called forward radix sort to sort a collection of variable-length strings $A_1 \ldots A_n$ in standard lexicographical (alphabetical) order. Let us suppose we have a word size of $\Theta(\log n)$ bits, so radix sorting a list of $n$ integer words would take $\Theta(n)$ time. We treat each string $A_i$ as an array of words, from the first “most significant” word down to the “least significant”. If we only look at, say, the 7 most significant words of $A_i$, we may not be able to distinguish $A_i$ from some of the other input strings, if they also agree in these 7 initial words. Let $d_i$ be the minimum number of words (in terms of a prefix of $A_i$) that we need to examine in order to be able to distinguish $A_i$ from the other input strings. If we let $D = \sum d_i$, then theoretically we should be able to sort $A_1 \ldots A_n$ by examining only $D$ total words. Note that this framework also applies to sorting real numbers (say, scaled to the range $[0,1]$) specified as arrays of digits from most significant down to least significant – here it is crucial to use a forward approach (examining most significant digits first) rather than the backward approach of standard radix sort, since a real number might have an unbounded number of digits.

(a) Note that we typically do not know $d_1 \ldots d_n$ at the outset. Show how to compute these values in only $O(nd_{\text{max}})$ time, where $d_{\text{max}} = \max d_i$. Once we know $d_{\text{max}}$, observe that we can apply standard radix sort to sort $A_1 \ldots A_n$ in $O(nd_{\text{max}})$ time. [Solution]

(b) The $nd_{\text{max}}$ term above is somewhat unsightly. Can you improve the running time for both determining $d_1 \ldots d_n$ as well as sorting $A_1 \ldots A_n$ to just $O(D)$? [Solution]

Input Sensitivity Versus Insensitivity: A Key Distinction. The running times of counting sort and radix sort are input-sensitive, depending on the magnitude $C$ of our input data. Since the running time of counting sort depends directly on $C$, we say counting sort has a pseudo-polynomial running time, whereas radix sort has a weakly polynomial running time since its running time depends only on $\log C$ (see Section 1.4.6 to review this distinction). All of our other sorting algorithms have strongly polynomial running times (depending only on $n$) since they run in the comparison-based or real RAM models of computation. There is an important distinction here to be made between these two types of algorithms: input-sensitive RAM algorithms (e.g., counting and radix sort) versus input-insensitive comparison-based or real RAM algorithms (e.g., merge sort and quicksort). For many problems we study, it will be possible to devise both types of algorithms. The choice of which is better depends entirely on the application; neither is inherently “better”. With sufficiently small integer input data, input-sensitive approaches may be superior. Otherwise, input-insensitive algorithms can be perhaps aesthetically more appealing and provide some peace of mind, since they do not depend on the magnitude of their input data. The choice between input-insensitive and input-sensitive, or more generally between RAM and real RAM (or comparison-based) algorithms will be a running theme throughout much of this book.

3.10 Advanced Example: Parametric Search

To reinforce the techniques introduced in this chapter, we close with a case study showing how they can be elegantly combined to solve a more challenging problem.

Suppose you take $n$ tests in an algorithms class, earning on the $i$th test $b_i$ points out of maximum of $m_i$ possible points. As is sometimes common, you are allowed
to drop \( k \) tests, with the final grade being determined by a set \( S \) containing only the remaining \( n - k \) tests:

\[
g(S) = \frac{\sum_{i \in S} b_i}{\sum_{i \in S} m_i}.
\]

Our goal is to decide which \( n - k \) tests to keep so that our grade \( g(S) \) is maximized; let \( g^* \) denote this maximum value. This is known as the optimal subset selection problem, and although it may seem solvable by simply discarding the \( k \) tests with the lowest \( b_i / m_i \) ratios, this is actually not the case. You may wish to pause and convince yourself of this fact by constructing an appropriate counterexample.

**Binary Search on the Answer.** As with any new problem, we might approach it by attempting to apply one of our main algorithm design techniques. Divide and conquer, specifically in the form of binary search, yields some useful initial insight. For many optimization problems, it turns out to be much easier to check whether the solution value \( g \) is yes if and only if the \( n \) tests whose sum is positive, which we can answer by finding the maximum sum of a subarray in \( B \) whose sum is positive, than to solve the problem outright. Any time this is the case, we can home in on the optimal solution value \( g^* \) quickly using binary search on \( x \). To check if \( x < g^* \), we want to know if there exists some subset \( S \) of tests with \( |S| = n - k \) such that

\[
\frac{\sum_{i \in S} b_i}{\sum_{i \in S} m_i} > x,
\]

and by rearranging terms, we see that this is the same question as whether or not there exists some subset \( S \) with \( |S| = n - k \) such that

\[
\sum_{i \in S} -m_i x + b_i > 0.
\]

If we define \( y_i = -m_i x + b_i \), we now simply want to know whether there is a set of \( n - k \) tests whose \( y_i \)'s add up to a positive amount\(^{10}\). The answer to this question is yes if and only if the \( n - k \) largest \( y_i \)'s add up to a positive amount, so we could answer it by sorting the \( y_i \)'s and adding up the largest \( n - k \) of them. Better yet, we can use selection to obtain an answer in only \( \Theta(n) \) time, by selecting for the \( k \)th largest value of \( y_i \) and adding up the \( n - k \) values above this point. Let us call this \( \Theta(n) \)-time algorithm for checking whether \( x \) is too low or too high \( A(x) \).

**Thinking Graphically.** Our notation \( y_i = -m_i x + b_i \) suggests a useful way to visualize the algorithm \( A(x) \) and the problem above in general — remember that you can almost always obtain better insight into solving a problem if you can turn it into a picture! If we regard each test \( i \) as the line \( y_i = -m_i x + b_i \) with slope \(-m_i\) and \( y \)-intercept \( b_i \), then we can picture our input as a collection of negative-slope lines \( A[i] + \ldots + A[j] \) (with \( j - i + 1 \) terms), which we further simplify to \((A[i] - x) + \ldots + (A[j] - x) > 0\). Defining \( B[i] = A[i] - x \), we now want to know whether there exists a subarray of \( B \) of length at least \( k \) whose sum is positive, which we can answer by finding the maximum sum of a subarray in \( B \) of length at least \( k \)— a problem that is easy to solve in \( \Theta(n) \) time (see, e.g., Section 8.1.1). We have therefore effectively replaced “average” with “sum” in our objective.

\(^{10}\)The “binary search on answer” technique is particularly effective for problems like this one involving optimization of a ratio, since through algebraic term re-arrangement it often reduces the problem to one having a simpler (non-ratio) objective. For example, given a numeric array \( A[1 \ldots n] \), suppose we want to find a window \( i \ldots j \) of length at least \( k \) with the largest average value (without the length \( \geq k \) constraint, the problem would be trivial, the answer being just the single largest element in \( A \)). To test if a guess \( x \) for the answer is too small, we ask whether there exists a window \( i \ldots j \) such that \( \frac{1}{j-i+1}(A[i] + \ldots + A[j]) > x \), otherwise written as \( A[i] + \ldots + A[j] > x + \ldots + x \) (with \( j - i + 1 \) terms), which we further simplify to \((A[i] - x) + \ldots + (A[j] - x) > 0\). Defining \( B[i] = A[i] - x \), we now want to know whether there exists a subarray of \( B \) of length at least \( k \) whose sum is positive, which we can answer by finding the maximum sum of a subarray in \( B \) of length at least \( k \)— a problem that is easy to solve in \( \Theta(n) \) time (see, e.g., Section 8.1.1). We have therefore effectively replaced “average” with “sum” in our objective.
Figure 3.8: Visualizing the optimal subset selection problem as the geometric problem of finding the rightmost value of \( x \) at which the \( y \) values of the \( n-k \) highest lines is still nonnegative.

lines as in Figure 3.8. In this geometric context, we want to find the value of \( x \) at which the sum of the \( y \) values of the \( n-k \) highest lines at evaluated at \( x \) is zero. Since our lines have negative slope, observe that the sum of the \( y \) values of the \( n-k \) highest lines decreases as we move \( x \) to the right, and increases as we move \( x \) to the left. This monotonicity is what makes binary search possible.

**Solving the Problem by Already Knowing the Answer.** Binary search on \( x \) to find \( g^* \) will work fine in practice, although this approach still leaves something to be desired, as it only converges to the optimal solution over time, rather than giving an exact answer after a number of steps polynomial in \( n \). On the real RAM, it could theoretically even take infinitely long to converge if \( g^* \) is irrational. Let us therefore consider a slightly different approach. Suppose we magically knew the \( n-k \) highest lines when evaluated at \( x = g^* \). Observe that this is enough to compute \( g^* \) in \( \Theta(n) \) time, since we can add together the linear functions representing these \( n-k \) lines to get a single linear function of \( x \), set it equal to zero, and solve for \( x \).

Now all we need to do is compute the highest \( n-k \) lines at \( x = g^* \). Normally, we would do this in \( \Theta(n) \) time by using quickselect to select for the \( k \)th highest line at \( x = g^* \), and then partitioning on this line to obtain the \( n-k \) lines above it. This may all seem futile, however, since we don’t know \( g^* \), so we don’t know what value of \( x \) to plug in when evaluating our lines. Accordingly, let us keep the \( y_i \)’s in the form of generic linear functions (e.g., \(-5x + 3\)), while running the algorithm above. This doesn’t cause problems until we reach a comparison, where we are now comparing two linear functions, such as \(-5x + 3\) versus \(-7x + 9\). The result of this comparison depends on \( x \); here, the first function is smaller if \( x < 3 \) and larger if \( x > 3 \). Remembering that we are planning to evaluate these functions at \( x = g^* \), all we need to know to resolve the comparison is therefore whether or not \( g^* < 3 \), and we can answer this question by running \( A(3) \).

Our approach now makes sense. As we run our high-level selection algorithm with the \( y_i \)’s represented generically as linear functions, we pause at every comparison run an invocation of \( A(x) \) to test the “breakpoint” value of \( x \) that is necessary to
resolve the comparison. Since every comparison of the high-level \( \Theta(n) \) algorithm invokes a lower-level \( \Theta(n) \)-time algorithm, the total running time is \( \Theta(n^2) \).

**A Clever Use of Divide and Conquer.** One final idea gives a dramatic improvement in running time. Our high-level algorithm based on quickselect performs a number of partition operations, each one comparing all the elements in an array against a specific pivot element. Since these comparisons don’t depend on each other (they could in theory be done in parallel), we can therefore use divide and conquer based on selection and binary search to resolve \( n \) such comparisons in only \( \Theta(n) \) time plus only \( O(\log n) \) invocations of \( A \) (instead of \( n \) invocations as before). This drops our total running time to only \( O(n \log n) \) in expectation. [Full details]

The technique above, known as *parametric search*, is useful in a wide range of problems, particularly in computational geometry. It is also an interesting example of how ideas from parallel computation can be used in novel ways to speed up sequential (non-parallel) algorithms. For our purposes, it serves as a wonderful example of what one can achieve using smart combinations of basic design techniques. See the endnotes for further references regarding the optimal subset selection problem, as this problem even admits a \( \Theta(n) \) solution!

### 3.11 Closing Remarks and Additional Problems

The ideal sorting algorithm would be comparison-based, stable, in-place, deterministic, run in \( O(n \log n) \) worst-case time (faster for nearly-sorted inputs), and require only \( O(n) \) memory writes. So far we only how to achieve certain specific combinations of these features, so despite the fact that sorting is such a fundamental problem, it continues to challenge even the best computing researchers.

Sorting is a key step in many algorithms (e.g., “sweep line” algorithms in computational geometry), and we will see problems involving sorting throughout this book. When we study data structures in the next few chapters, we will learn even more ways to sort, using data structures like priority queues and binary search trees.

**Problem 50 (Sorting as a Means of Grouping).** One common use of sorting during preprocessing is simply to group similar elements together. As a nice example of this approach in action, please describe an \( O(n^3 \log n) \) algorithm that takes \( n \) points in the 2D plane as input and counts the number of parallelograms whose corners all come from the point set. Can you count rectangles as well (not necessarily aligned with the \( x \) and \( y \) axes)? [Solution]

**Problem 51 (Divide and Conquer Practice).** This problem contains a collection of nice practice problems that can be solved elegantly using the divide and conquer principle.

(a) **Repeated Squaring.** Suppose we want to compute \( A^k \), where \( A \) is an \( n \times n \) matrix. We could do this by starting with \( A \) and successively multiplying by \( A \) for \( k - 1 \) iterations. However, since matrix multiplication is a time-consuming operation (the straightforward approach for multiplying two \( n \times n \) matrices gives has an \( O(n^3) \) running time), we would like to use as few multiplications as possible. Please show how to compute \( A^k \) using only \( O(\log k) \) matrix multiplications and additions (note that matrix additions are much less costly — these only take \( O(n^2) \) time). As a hint, remember
that the technique you are developing goes by the name of repeated squaring. Despite its simplicity, this method is quite powerful, and it is commonly used in practice for raising any complicated object (e.g., a matrix or a polynomial, or any other object for which multiplication is costly) to a large power. [Solution]

(b) Median Among Two Sorted Arrays. You are given two sorted arrays $A[1 \ldots n/2]$ and $B[1 \ldots n/2]$ as input. Please show how to compute the median among all the $n$ elements contained in both arrays in $O(\log n)$ time. For a challenge, what is the best performance guarantee you can offer (in terms of $n$ and $k$) if $A$ and $B$ contain $k$ and $n-k$ elements rather than $n/2$ elements each? What if you want to find the median element among $k$ sorted arrays each containing $n/k$ elements? [Solution]

(c) Find the Missing Element. You are given two arrays: $A$, containing $n$ elements, and $B$, containing $n-1$ of the elements present in $A$. Design a comparison-based algorithm running in $\Theta(n)$ time that determines which element in $A$ does not appear in $B$. How quickly can you solve the more general problem variant where $k$ elements are left out of $B$? What about the similar problem where $B$ contains $n+1$ elements — including all of the elements in $A$, but with one of them occurring twice — where we wish to locate the duplicated element? [Solution]

(d) Cyclic Shift of an Increasing Sequence. Suppose $A[1 \ldots n]$ is obtained by performing a $k$-step right cyclic shift of a length-$n$ array whose elements form a strictly increasing sequence (a $k$-step right cyclic shift moves every element to an index $k$ steps ahead, with the $k$ final elements “wrapping around” and becoming the $k$ first elements). Given $A$, how can you determine $k$ in $O(\log n)$ time? [Solution]

(e) Monotone Matrix Searching. Let $A$ be an $n \times n$ matrix whose entries are distinct numbers, and let $M(i)$ denote index of the minimum element in row $i$. We say $A$ is monotone if $M(1) \leq M(2) \leq \ldots \leq M(n)$. Please give an $O(n \log n)$ algorithm for computing $M(1) \ldots M(n)$ in an $n \times n$ monotone matrix. [Solution]

(f) Approximate Binary Search. Suppose you want to find an unknown value $v$ in the range $1 \ldots n$. If $v$ is an integer, we can of course binary search for $v$ in $O(\log n)$ time by first guessing $n/2$, then seeing if this guess is too high or too low, and successively halving our search range in each step. Suppose, however, that $v$ is not necessarily an integer, and that we are content to find an approximate solution within the range $(1-\epsilon)v \ldots (1+\epsilon)v$, where $\epsilon$ is some constant. As it turns out, we can solve this problem in only $O(\log \log n)$ time by using the geometric mean rather than the arithmetic mean as our guess for each step in the binary search. That is, when we are searching the interval $[a,b]$, our next guess will be $\sqrt{ab}$. Please prove that this does indeed give us an $O(\log \log n)$ running time. As a hint, consider problem 8. [Solution]

(g) Longest Line of Sight. You are given an array $H[1 \ldots n]$ containing distinct numbers that describe the heights of $n$ people standing in a line. We say person $i$ and person $j > i$ can see each-other if $H[i]$ and $H[j]$ are both larger than all of the elements in $H[i+1 \ldots j-1]$. Please show how to locate a farthest pair of individuals $i$ and $j$ that can still see each-other in $O(n \log n)$ time. [Solution]

(h) Printing a Linked List Backwards. You are given as input a pointer to the first element of an $n$-element linked list (without being told $n$), and asked to print out the contents of the list in reverse order without physically modifying the elements of the list in any way. This is trivial to do in $\Theta(n)$ time if we allow $\Theta(n)$ extra space, but somewhat more interesting if we wish to use much less auxiliary memory while still maintaining a fast running time. Please describe an $O(n \log n)$ time solution using

---

11 In Section 11.5, we will see monotone matrices with even more structure, known as totally monotone matrices, for which we can actually compute $M(1) \ldots M(n)$ in only $\Theta(n)$ time (for the case of a monotone matrix there is actually an $\Omega(n \log n)$ worst-case lower bound in the comparison model). As we will see, this result has surprisingly many applications!

12 In problem 131, we improve the running time to $\Theta(n)$.
only $O(\log n)$ auxiliary words of memory. Can you generalize your solution to run in $O(kn)$ time using $O(kn^{1/k})$ space for any $k = 1 \ldots \log n$? Observe that $k = 1$ gives the trivial solution where we use $\Theta(n)$ extra space, and that $k = \log n$ leads to the time and space bounds of $O(n \log n)$ and $O(\log n)$ above. [Solution]

(i) Finding a Local Minimum. Let $f(x)$ be a function defined over all integers $x$ in the range $1 \ldots n$. We say $x$ is a local minimum of $f$ if $f(x) \leq f(x-1)$ and $f(x) \leq f(x+1)$, where we treat $f(0) = f(n+1) = +\infty$. Please show how to compute a local minimum of $f$ by evaluating $f$ at only $O(\log n)$ points. Next, let $g(x,y)$ be a function defined over integers $x$ and $y$ in the range $1 \ldots n$. We say $(x,y)$ is a local minimum if $g(x,y)$ is no larger than any of the four neighboring values $g(x \pm 1, y)$ and $g(x, y \pm 1)$ (again, we treat $g(x,y)$ as being infinite if $x$ or $y$ is not in the range $1 \ldots n$). Please show how to compute a local minimum of $g$ with only $O(n)$ function evaluations. [Solution]

(j) Cake Cutting. Suppose you want to divide up a cake among $n$ people so that every person feels like he or she has received a fair-sized piece. This is easy if everyone values every part of the cake equally, since the solution in this case is just to give $1/n$ of the cake to everyone. However, in practice we might find that different individuals might have non-uniform preferences over different parts of the cake (e.g., I might like the side of the cake with more frosting, and you might prefer the side of the cake with more sprinkles on top). To be somewhat more formal, let us consider the unit interval $[0, 1]$ as a long one-dimensional “cake”. Each of our $i = 1 \ldots n$ participants must be served a piece of cake that corresponds to a contiguous piece of this interval. The preferences of participant $i$ are described by a valuation function $f_i(x)$ that specifies his/her value for the interval $[0, x]$. Each $f_i(x)$ function is continuous, monotonically increasing, and ranges from $0$ up to $f_i(1) = 1$ (i.e., the whole cake is worth 1 unit to each participant). Note that we can find the value of any subinterval $[a, b]$ by taking $f_i(b) - f_i(a)$. In addition to $f_i(x)$, suppose we also have access to its inverse function $f_i^{-1}(y)$, for example $f_i^{-1}(1/2)$ would tell us the unique point $x$ such that exactly half of the value for participant $i$ lies in $[0, x]$ and the other half lies in $[x, 1]$. Suppose that it takes $O(1)$ time to evaluate $f_i(x)$ or $f_i^{-1}(y)$. Design an $O(n \log n)$ time algorithm that computes a subinterval $[a_i, b_i]$ for each participant $i$ such that $f_i(b) - f_i(a) \geq 1/n$ (i.e., each participant receives what he or she perceives to be a “fair” piece of the cake). [Solution]

(k) Counting Distant Pairs of Points. You are given $n$ points $(x_1, y_1) \ldots (x_n, y_n)$, where the $x_i$’s and $y_i$’s are integers in the range $1 \ldots n$. Two points $(x_i, y_i)$ and $(x_j, y_j)$ are said to be $(a,b)$-distant if $|x_i - x_j| \geq a$ and $|y_i - y_j| \geq b$. Given $a$ and $b$ as input, please describe an $O(n \log n)$ divide-and-conquer algorithm for counting the number of pairs of $(a,b)$-distant points. For a warm-up, you may want to consider $a = b = n/2$. [Solution]

(l) The “Firing Squad” Problem. This is a classical problem in parallel algorithm design that has an elegant solution based on the principle of divide and conquer. Suppose we have $n$ processors connected in a line. Each processor is synchronized with a global clock, and each time the clock ticks the processor can perform $O(1)$ work as well as send $O(1)$ information to its neighbors (so it takes $n - 1$ time steps for a message from one end of the line to propagate down to the other end). However, each processor is somewhat limited in that it only has a constant number of bits of memory, independent of $n$. This prohibits processors from doing things like counting to $n$ (which would require $\log_2 n$ bits of memory). Our goal is to design an algorithm to run continuously on these processors (each processor should run the same algorithm) such that if we input a special message to one of the endpoint processors at some time, all processors should simultaneously enter a special state (e.g., they should all “fire”) at some point $O(n)$ time steps later. Can you devise a simple algorithm that accomplishes this goal? [Solution]
Problem 52 (Frequently-Occurring Elements). We can easily solve the problem of finding the most-frequently-occurring element in an \(n\)-element array in \(O(n \log n)\) time by sorting, and we also have an \(\Omega(n \log n)\) worst-case lower bound on this problem in the comparison model via a simple reduction from the element uniqueness problem. However, if the most-frequently-occurring element appears more than \(n/2\) times in the array, we can solve this special case in only \(\Theta(n)\) time (how do we do this?) This motivates us to try and design an algorithm for finding the most-frequently-occurring element whose running time scales gracefully between \(\Theta(n)\) and \(\Theta(n \log n)\), depending on the number of occurrences \(k\) of the most-frequently-occurring element. Please show how to solve this problem in \(O(n \log(n/k))\) time, and if you are feeling ambitious, try to prove a matching lower bound in the comparison-based model. [Solution]

Problem 53 (k-Way Merging). Consider the problem of merging \(k\) previously-sorted lists containing \(n\) total elements. There are several ways to solve this problem in \(O(n \log k)\) time. For example, one can perform the standard iterative merging algorithm using a data structure such as a binary heap or balanced binary search tree to maintain the \(k\) leading elements in our lists; in every iteration, we select the minimum such element in \(O(\log k)\) time, adding it next to the merged sequence. Please comment on how we might instead solve this problem using a divide-and-conquer approach. Next, argue that there is a matching worst-case lower bound of \(\Omega(n \log k)\) in the comparison-based model for any algorithm that performs \(k\)-way merging of \(n\) elements. [Solution]

Problem 54 (Sorting A Sequence Drawn from a Small Universe). Suppose we have array of \(n\) elements drawn from a universe of only \(k \leq n\) distinct elements, and we wish to sort this array using a comparison-based sorting algorithm. If \(k\) is small we would hope to improve on the \(O(n \log n)\) running time for generic sorting algorithms.

(a) Can you modify merge sort to solve this sorting problem in only \(O(n \log k)\) time? Your algorithm will not be told the value of \(k\). [Solution]

(b) Show that randomized quicksort applied to this problem runs in \(O(n \log k)\) time with high probability, even though it also does not know the value of \(k\). [Solution]

(c) Argue that there is a lower bound of \(\Omega(n \log k)\) on the worst-case running time of any comparison-based algorithm that solves this problem, even if the algorithm knows the \(k\) distinct values appearing throughout the array. [Solution]

(d) Suppose you are given two strings (arrays) \(A[1 \ldots n]\) and \(B[1 \ldots n]\) whose characters (elements) are drawn from an alphabet \(\Sigma\) (e.g., the letters A through Z). We wish to detect whether or not \(A\) and \(B\) are anagrams — that is, whether or not you can transform \(A\) into \(B\) by permuting its elements. If \(k = |\Sigma|\), show that in the comparison-based model of computation one can detect anagrams in \(O(n \log k)\) time, and also show that there is a matching lower bound of \(\Omega(n \log k)\) on the worst-case running time of any anagram detection algorithm. [Solution]

Note that another nice solution for sorting in \(O(n \log k)\) time will become apparent once we learn in Chapter 6 to use balanced binary search trees as “maps” — in that case, we insert our \(n\) values in to a balanced binary search tree where each element is augmented with a frequency count (this takes \(O(n \log k)\) time, since the tree contains \(k\) elements), and then we traverse the tree to enumerate its contents in sorted order.

Problem 55 (The Zero/One Sorting Theorem). The well-known zero/one sorting theorem is very useful for proving correctness of complicated sorting algorithms. It states that a comparison-based sorting algorithm is correct if and only if it correctly sorts sequences consisting of just zeros and ones. Therefore, in order to argue correctness of a complicated sorting algorithm, it suffices to argue that it sorts any sequence of zeros and ones. Please give a short proof of the zero/one sorting theorem. [Solution]
Problem 56 (Chain and Block Sorting). Let us think of an array $A[1 \ldots n]$ as $k$
interleaved “chains” of elements. The first chain is $A[1], A[k+1], A[2k+1], \ldots$, the second
is $A[2], A[k+2], A[2k+2], \ldots$, and so on. We say an array $A[1 \ldots n]$ is $k$-chain-sorted if
each of its $k$ chains is sorted. We can also think of an array $A[1 \ldots n]$ in terms of $n/k$
“blocks” of size $k$. The first block is $A[1 \ldots k]$, the second is $A[k+1 \ldots 2k]$, and so on. We
say $A$ is $k$-block-sorted if each of its $n/k$ blocks is sorted.

(a) What is the fastest possible running time (in the comparison model) for making an
array $k$-chain-sorted or $k$-block-sorted? What is the fastest possible running time
(also in the comparison model) for sorting an array that is already $k$-chain-sorted and
$k$-block-sorted? [Solution]

(b) Take an $n$-element array that is $k$-chain-sorted and $k$-block-sorted. For notational
simplicity, let $A = \min(k, n/k)$ and let $B = \max(k, n/k)$. Suppose we want to search
for a particular element based on its value. Give algorithms that perform this task in
$O(A \log B)$ time (fairly easy), $O(A+B)$ time (slightly more difficult), and $O(A \log (1+
B/A))$ time (a bit more difficult). Note that the last running time dominates both of
the other two. [Solution]

(c) An $n$-element array that is both $k$-chain-sorted and $k$-block-sorted for $k = \sqrt{n}$ can
be visualized as the row-order contents of a $\sqrt{n} \times \sqrt{n}$ matrix of elements whose rows
and columns are all sorted. In the preceding problem, we showed how to search in
such a structure in $O(\sqrt{n})$ time. How quickly can you search in the $d$-dimensional
generalization of this structure? For example, if $d = 3$, then your data would be
stored in a 3-dimensional array of size $n^{1/3} \times n^{1/3} \times n^{1/3}$ that is sorted along each
dimension. [Solution]

(d) Show that if we $k$-block-sort (by sorting each of its blocks independently) an array that
is already $k$-chain-sorted, the array remains $k$-chain-sorted, and vice-versa. Show also
that for any $k$ and $k'$, if we $k'$-chain sort (by sorting each of its chains independently)
an array that is already $k$-chain-sorted, then the array stays $k$-chain-sorted. Is the
same true for $k$-block-sorting and $k'$-block-sorting? [Solution]

Problem 57 (Shell Sort). Shell sort, named after its creator Donald Shell, is a
method for speeding up insertion sort by allowing elements to be moved longer distances
initially. In the context of the preceding problem, it involves making an array $k$-chain-
sorted (by calling insertion sort on each of its $k$ chains independently) for decreasing
values of $k$ coming from a so-called increment sequence. For example, suppose we use the
increment sequence $\{1, 3, 7, 15, \ldots\}$ of integers one less than powers of two, so we start
with $k$ being the largest increment in this sequence less than $n$, and end with $k = 1$.
Increment sequences always end with $k = 1$, since this ensures the final array is sorted
(note that 1-chain-sorted means the same thing as sorted). Shell sort runs in $O(n^{1.5})$
time for the increment sequence above, but a better sequence (at least in theory, although
sometimes not in practice) is the set containing all integers of the form $2^i 3^j$, of which there
are $\Theta(\log^2 n)$ elements at most $n$. Using the result from the last part of the preceding
problem, we know that when it comes time to $2^i 3^j$-chain-sort our array, it will already be
$2^{i+1} 3^j$-chain-sorted and $2^i 3^{j+1}$-chain-sorted. Show that this implies that $k$-chain-sorting
for each increment $k$ takes only $\Theta(n)$ time, leading to an overall running time of $\Theta(n \log^2 n)$,
which is nearly the best attainable for Shell sort (see the endnotes for further details). As
a hint, remember that insertion sort runs quickly when there are few inversions in the
arrays being sorted. [Solution]

Problem 58 (Selection in a Sorted Matrix). Building on your solution to
problem 56 above, please now consider the problem of selecting the $k$th largest element
from an $n \times n$ matrix whose rows and columns are both sorted. For a challenge, see if
you can solve this problem in only $\Theta(n)$ time. Please assume for simplicity that all entries
in the matrix are distinct. As a hint, try partitioning the matrix into small blocks (e.g.,
Figure 3.9: Diagrams of (a) a comparator, (b) a 3-stage sorting network that sorts 4 inputs, and (c) an n-element sorting network that implements merge sort: The blocks labeled $S_{n/2}$ recursively sort inputs of length $n/2$, and the block labeled $M_n$ merges their output into a sorted list of length $n$.

2 $\times$ 2 or 3 $\times$ 3 in size), and construct a smaller matrix by replacing each block with its minimum or maximum; by recursively solving a selection problem in this smaller matrix, you should be able to exclude from consideration many of the elements in the original matrix. [Solution]

Problem 59 (Parallel Sorting Networks). In this problem we briefly delve into the realm of parallel sorting algorithms. A common model for parallel sorting is the sorting network, a circuit taking $n$ numbers as input that produces their sorted ordering as output. An example of a sorting network is shown in Figure 3.9(b). Sorting networks are built from blocks called comparators (Figure 3.9(a)) that compare two input numbers and output the numbers in sorted order. A sorting network is built from a series of stages (or layers), each of which contains several comparators. However, each stage can be at most one comparator “deep” (i.e., a signal may not pass through two different comparators in the same stage). The goal of a sorting network is to minimize the number of stages, as this corresponds to the running time of the network. Although it does not seem possible to map any of our most common $O(n \log n)$ comparison-based sorting algorithms (e.g., merge sort and quicksort) to an efficient parallel sorting network, there are several elegant approaches for constructing efficient sorting networks. We discuss two of these below.

(a) Batcher’s Odd-Even Merge Sort. Suppose we wish to merge two length-$n/2$ sorted arrays $A$ and $B$ into a single length-$n$ sorted array $C$. Consider the following somewhat strange “odd-even” recursive approach for performing this merge: the odd elements of $C$ (i.e., $C[1]$, $C[3]$, etc.) are obtained by recursively merging the odd elements of $A$ and $B$, and the even elements of $C$ are obtained by recursively merging the even elements of $A$ and the even elements of $B$. After running these recursive merges and filling up $C$, we make one final postprocessing pass in which we compare successive pairs of elements in $C$: $C[1]$ versus $C[2]$, $C[3]$ versus $C[4]$, and so on, swapping any pairs that are out of order. Please use the zero/one sorting theorem to prove that this algorithm merges correctly. Next, show how to build a sorting network based on this merge operation that requires $O(\log^2 n)$ total stages. [Solution]

(b) Bitonic Sorting. Another way to merge two sorted length-$n/2$ sequences is to concatenate the first with the reversal of the second and then to sort the resulting bitonic sequence. We call a sequence bitonic if it increases up until some point and then decreases, or if it is a “cyclic shift” of such a sequence. Although it may seem like we’re moving in the wrong direction by converting a merging problem back into a sorting problem, it turns out that bitonic sorting is a natural and easy problem on a sorting
network. Suppose \( A[1 \ldots n] \) contains a bitonic sequence. In a single stage of our sorting network, we place comparators between \( A[1] \) and \( A[n/2 + 1] \), \( A[2] \) and \( A[n/2 + 2] \), and so on. After executing this stage, show (using the zero/one sorting theorem) that the elements in \( A[1 \ldots n/2] \) must all be no larger than the elements in \( A[n/2 + 1 \ldots n] \), and that moreover, these two length \( n/2 \)-subarrays must also be bitonic. As a consequence, we can now continue to sort these two half-sized subproblems recursively in parallel. Show that merging according to this strategy requires \( O(\log n) \) stages, and that this leads to a parallel version of merge sort that requires \( O(\log^2 n) \) total stages.

**Problem 60 (The Post Office Problem).** We know the locations of \( n \) businesses in the downtown section of our city — these locations are simply \((x, y)\) coordinates in the plane. Since the streets in downtown are shaped like a grid, we measure distances using the \( L_1 \), or “Manhattan” metric, in which the distance between two points \((x_1, y_1)\) and \((x_2, y_2)\) is given by \( |x_1 - x_2| + |y_1 - y_2| \).

(a) Suppose we wish to choose the coordinates of a post office such so as to minimize the sum of distances between the post office and each business. Show how to find a suitable location for the post office in \( \Theta(n) \) time. If you want to start with a simpler problem, consider first the one-dimensional variant where the businesses are points on a number line. Can you generalize your solution to work in higher dimensions?

(b) Let us now assign weights \( w_1 \ldots w_n \) to the businesses, indicating the relative important of the post office being close to each business. We now wish to find a location for the post office which minimizes the weighted sum of distances to the businesses. Show how to solve this problem in \( \Theta(n) \) time.

**Problem 61 (Counting Inversions).** In this problem we will attempt to count the inversions in an \( n \)-element array.

(a) Modify the merge sort algorithm to count the total number of inversions in an array while still running in \( O(n \log n) \) time. Try to extend this algorithm to count the number of inversions per array element. We define the number of inversions for an element \( A[j] \) as the number of elements preceding it in the array that have larger values than \( A[j] \). We only consider preceding elements so we count each inversion exactly once — summing up all of the per-element inversion counts should therefore give the total number of inversions.

(b) Suppose we would like to efficiently count inversions in an array of integers in the range \( 0 \ldots C - 1 \). Show first that if \( C = O(1) \), then we can do this in linear time. If \( C = O(n^c) \) for \( c = O(1) \), then we can sort in linear time via radix sort but it seems difficult to exactly count inversions in linear time. However, we can approximate the number of inversions in linear time. Consider summing up for each element in our array the distance between its current index in the array and its rank (i.e., its index within the array once the array is sorted). The resulting measure, \( F \), is known as Spearman’s Footrule. Show that we can compute this quantity in linear time, and that \( I \leq F \leq 2I \) where \( I \) denotes the number of inversions in our array.

**Problem 62 (Counting Subarrays).** You are given as input an array \( A[1 \ldots n] \) of real numbers and a target value \( v \).

(a) Please describe how to count the number of subarrays \( A[i \ldots j] \) with sum at least \( v \) in \( O(n \log n) \) time.
(b) Please describe how to count the number of subarrays $A[i \ldots j]$ with median at least $v$ in $O(n \log n)$ time, or better yet, $O(n)$ time (for a subarray of even size, this means that at least half its elements must be no smaller than $v$). [Solution]

(c) Please describe how to count the number of subarrays $A[i \ldots j]$ with mean at least $v$ in $O(n \log n)$ time. [Solution]

Problem 63 (Rank Aggregation). There are many situations in practice where we might want to solve a rank aggregation problem that involves finding an ordering of $n$ elements that is “reasonably close” to a set of $k$ different orderings. For example, in an athletic competition with $n$ athletes, we may have $k$ judges who each independently rank the athletes, after which we need to generate a single ranking that agrees the greatest extent with the rankings of all $k$ judges. Alternatively, when conducting a web search, it may be more robust to somehow combine the rankings of $n$ pages by $k$ different search engines, rather than to use the rankings returned by only a single search engine. Suppose, given $k$ ranked lists $L_1 \ldots L_k$ on the same set of $n$ elements, that we wish to construct a single ranked list $L^*$ minimizing $\sum_{i=1}^k I(L_i, L^*)$, where $I(a, b)$ denotes the number of inversions between lists $a$ and $b$. This problem is NP-hard, although it can be approximated well (see also the endnotes for more details). Here, please show that if we choose one of the $k$ input orderings uniformly at random, then this gives a 2-approximate solution. This fact implies that if we take the best of the $k$ orderings, we also obtain a 2-approximate solution, since the minimum of a set of numbers is always no larger than the average. You can either solve this problem directly or, if you prefer, using the result of problem 35. In the second case, all you need to show is that the inversion distance between two orderings is a metric: for any three orderings $a$, $b$, and $c$, the triangle inequality $I(a, b) + I(b, c) \geq I(a, c)$ holds. We discuss an alternate 2-approximate solution based on network flow algorithms in problem ?? [Solution]

Problem 64 (Fun With Intervals). You are given $n$ intervals $[a_1, b_1] \ldots [a_n, b_n]$ on the number line. For simplicity, assume all interval endpoints are distinct (this assumption is not fundamentally required for any of the results below).

(a) Two intervals can be related in 3 possible ways: they can nest (one interval within the other), they can disjoint (not overlapping at all), or they can cross (overlapping but not nesting). A set of intervals is said to be laminar if there are no crossing pairs of intervals; that is, if two intervals overlap at all, they must nest. Laminar intervals exhibit a sort of “balanced parenthesis” structure. Give an $O(n \log n)$ algorithm for testing whether or not a set of $n$ intervals is laminar. [Solution]

(b) For a challenge, see if you can prove an $\Omega(n \log n)$ worst-case lower bound on the running time of any deterministic comparison-based algorithm that solves part (a). [Solution]

(c) Suppose for each interval in our set we wish to compute its containment count (the number of intervals contained within the specified interval), its inclusion count (the number of other intervals containing the specified interval) as well as its overlap count (the number of other intervals overlapping the specified interval, not counting those containing or contained within the interval). Show how to compute these quantities for each of the $n$ input intervals in $O(n \log n)$ time. [Solution]

Problem 65 (Nesting Sets). Suppose we have a collection of $n$ sets $S_1 \ldots S_n$, each containing some subset of the elements $\{1, 2, \ldots, m\}$. For each set, we are given as input a list of the elements it contains. Two sets $S_i$ and $S_j$ are said to be disjoint if $S_i \cap S_j = \emptyset$, to nest if $S_i \subseteq S_j$ or $S_j \subseteq S_i$, and to cross otherwise. If every pair of sets is disjoint or nesting, we say our collection of sets is laminar. Please design an $O(mn)$ algorithm for checking whether a collection of sets is laminar. Can you improve the running time to $O(m + n + k)$, where $k$ is the total number of elements in all the input sets? [Solution]
3.11. CLOSING REMARKS AND ADDITIONAL PROBLEMS

Problem 66 (Average and Median Distance on a Line). In this problem we investigate the computation of statistical information about the set of \( \binom{n}{2} \) distances between \( n \) points on a number line. As input, you are given the locations \( x_1 \ldots x_n \) of these points (not necessarily sorted).

(a) Give an \( O(n \log n) \) algorithm for computing the average distance among all pairs of points. [Solution]

(b) Give a randomized algorithm running in \( O(n \log n) \) expected time that computes the \( k \)th largest of all \( \binom{n}{2} \) pairwise distances. [Solution]

(c) For a challenge, can you find a deterministic \( O(n \log n) \) algorithm for the problem from part (b)? [Solution]

Problem 67 (Sorting with Substring Reversals). The reversal distance between sequences \( A \) and \( B \) is the minimum number of substring reversals required to transform \( A \) into \( B \) (often we consider \( B \) to be the sorted order of \( A \), asking us to sort \( A \) with a minimum number of substring reversals). The problem of computing reversal distance (or equivalently, sorting by reversals) has application in computational biology, where the reversal distance between two strands of DNA gives some indication of their evolutionary similarity. It is NP-hard\(^{14} \), even in the special case where we are only allowed to reverse prefixes of our sequence, affectionately known as the pancake flipping problem since it models the situation where we are trying to sort a stack of pancakes of different sizes using a spatula with which we can only flip over a group of pancakes at the top of the stack\(^{15} \).

(a) A simple approach for pancake flipping is to place the largest pancake at the bottom of the stack (if it isn’t already there) by first flipping it up to the top, and the reversing the entire stack. We then proceed to place the second-largest pancake, and so on, only moving pancakes that need to be moved. Please argue that this is actually a 2-approximation algorithm. As a hint, a useful concept here is the notion of a breakpoint, which exists between two pancakes if they are not neighbors in the final sorted sequence. [Solution]

(b) Now consider the general problem of sorting with substring reversals. A natural greedy algorithm for this problem is to reverse, repeatedly, a substring that reduces the number of breakpoints in our sequence as much as possible. Since a reversal can only affect breakpoints at its endpoints, the best we can hope to do is reduce the number of breakpoints by 2. The worst we can do is fail to reduce the number of breakpoints at all (so we need to be careful in proving that the algorithm actually terminates). As a tie-breaking mechanism, if the algorithm can only manage to decrease the number of breakpoints by 1, then it should make its choice in such a way that it leaves a decreasing strip if possible. A strip is just a substring of elements between any two consecutive breakpoints, which can be either increasing or decreasing. For a challenge, prove that the greedy algorithm is a 2-approximation algorithm. As a starting point, show that the greedy algorithm uses only \( B - 1 \) reversals to sort a sequence with \( B \) breakpoints, as long as it has at least one decreasing strip (hence, if there are not any decreasing strips initially, our first reversal will create one, so we will use at most \( B \) reversals in total). [Solution]

---

\(^{14}\)The general problem of computing the reversal distance between two strings is known to be NP-hard, although somewhat surprisingly its “signed” variant can be solved in polynomial time by a rather complicated algorithm (further references can be found in the endnotes). The signed variant is particularly relevant to some biological situations where elements of our sequence each have an inherent notion of directionality; for example, by reversing the substring \( BCD \) in \( ABCDE \) we would end up with \( AD'C'B'E \) where \( X' \) denotes the element \( X \) oriented in the reverse direction. Our sequence of reversals in the signed case must leave every element oriented in the forward direction. In this problem, we consider only the “unsigned” variant.

\(^{15}\)Its signed variant is known as the “burned pancake” problem, where each pancake has a burned side, and all of these need to end up facing downward.
Problem 68 (Perfect Shuffles and In-Place Matrix Rearrangement). When playing cards, a perfect shuffle involves taking the top and bottom halves of a deck of cards and interleaving the two. If we think of the deck of cards as an array of length \( n \) having elements \( A[1..n/2] \) followed by \( B[1..n/2] \), then after a perfect shuffle the array would contain \( A[1], B[1], A[2], B[2], \ldots, A[n/2], B[n/2] \). Here, we design elegant divide-and-conquer algorithms for perfect shuffles and related problems that operate in place.

(a) Design in-place \( O(n \log n) \) algorithms for performing (i) a perfect shuffle and (ii) the inverse of a perfect shuffle. As a hint, consider problem 45. [Solution]

(b) Suppose we store an \( m \times n \) matrix in row-major form — as an array of length \( N = mn \) in which we list the elements of the matrix one row at a time from left to right. By taking the transpose of such a matrix, we convert it into column-major form — an array of length \( N \) listing the columns of the matrix one at a time, each one from top to bottom. The perfect shuffle is equivalent to computing the transpose of a \( 2 \times n/2 \) matrix, and more generally the transpose of an \( m \times n \) matrix can be seen as an \( m \)-ary perfect shuffle, where we divide a sequence into \( m \) equal blocks and then interleave their contents. Using the result from (a) as black box, show how to compute the transpose of an arbitrary matrix in place in \( O(N \log^2 N) \) time. [Solution]

(c) Consider again an \( m \times n \) matrix stored in row-major form in an array of length \( N = mn \). The in-place matrix reblocking problem involves dividing up the matrix into \( a \times b \) blocks. That is, we view the matrix as an \( m/a \times n/b \) matrix of blocks, each block having size \( a \times b \). In order to reflect this restructuring of the matrix, we wish to rearrange its elements in memory so that the blocks are listed in row-major form, and the contents of each block are also listed in row-major form. This allows us to more easily apply divide and conquer algorithms that operate by partitioning a matrix into blocks. Please show how to reblock a matrix, in place, in \( O(N \log^2 N) \) time. [Solution]

Problem 69 (In-Place Permutation). As a follow-up to the previous problem, we now develop a simple \( O(n \log n) \) in-place algorithm for performing an arbitrary \( n \)-element permutation, as long as we also know the inverse of the permutation (this is true for the perfect shuffle and matrix reblocking and transposition permutations). Consider permuting an array \( A[1..n] \) according to some permutation \( \pi(1) \ldots \pi(n) \), where \( \pi(i) \) specifies the new index into which the element \( A[i] \) should be moved. We are also given the inverse of our permutation \( \pi^{-1}(1) \ldots \pi^{-1}(n) \), so \( \pi^{-1}(\pi(i)) = i \) and \( \pi(\pi^{-1}(i)) = i \). We apply our permutation by shifting elements around its “cycles”. For example, we move \( A[1] \) to index \( \pi(1) \), which displaces element \( A[\pi(1)] \) which we then move to index \( \pi(\pi(1)) \), and so on, until eventually, we close a cycle and place an element back at index 1. We can trace out such a cycle starting from any element in our array, and if that element happens to have the smallest index among all elements in the cycle, we call that element the leader of the cycle. Our algorithm scans through all array indices \( i = 1 \ldots n \) and rotates the cycle containing element \( i \) only if \( i \) is the cycle leader, thereby ensuring each cycle is rotated only once.\(^{16} \)

The only remaining problem is to determine if index \( i \) is the leader of its cycle. To do this, we could scan forward to \( \pi(i), \pi(\pi(i)) \), and so on, until we either encounter an index smaller than \( i \) (in which case \( i \) was not the leader) or we return to \( i \) (in which case \( i \) was the leader). This may take too long if our permutation has long cycles, so instead we walk outward simultaneously in both directions, visiting \( \pi(i) \) and \( \pi^{-1}(i) \), then \( \pi(\pi(i)) \) and \( \pi^{-1}(\pi^{-1}(i)) \), and so on, until we again reach an index less than \( i \) or return to \( i \). Try to prove that regardless of our permutation, this bidirectional scan results in an \( O(n \log n) \) total worst-case running time for our permutation algorithm.\(^{17} \) [Solution]

\(^{16}\) Normally, we would ensure each cycle is rotated only once by simply marking its elements when they are rotated. However, that wouldn’t give an in-place algorithm, so we use the leader trick instead.

\(^{17}\) You may want to also consider how this analysis leads to an \( O(n \log n) \) time solution to problem 131 as well.

DRAFT. Not for redistribution.

Copyright © Brian C. Dean
Data Structures
4. Amortized Analysis

Just as you can work more efficiently if the items in your office are well organized, an algorithm can operate more efficiently if its data is organized well in memory. This leads us to the study of data structures, which we cover in extensive detail over the next six chapters. Expertise in data structures is a fundamental part of any successful algorithm designer’s skill set.

Our study of data structures officially began back in Section 1.5, where we introduced several preliminary concepts, including arrays, linked lists, stacks, queues, and importantly, the distinction between the specification of a data structure in terms of the operations it should support, and its concrete implementation. There are often many ways to implement a particular type of data structure, each with its advantages and disadvantages — for example, an array or a linked list can serve as an implementation of an abstract mathematical sequence.

This chapter discusses amortized analysis, a useful method for analyzing data structures with non-uniform performance characteristics (of which there are many). For example, suppose each invocation of a data structure operation usually takes $\Theta(n)$ time, but once in a while it may take $\Theta(n^2)$ time to perform some periodic “housekeeping”. Standard worst-case analysis (e.g., saying only “the worst-case running time is $\Theta(n^2)$ per invocation”) does not serve our data structure well. It might scare away potential users, since it makes the data structure sound much slower than it actually is. Amortized analysis provides a much more accurate way to characterize performance by looking at worst-case running time averaged over a sequence of operations. This simplifies the description of the running time by smoothing away non-uniformity, and also does a much better job of faithfully capturing the true performance of the structure.

4.1 Example: Buffering Writes to Disk

Disk access involves much higher latency than memory access, since we typically need to wait for a spinning magnetic platter to rotate until our data physically lies underneath the read/write head of the disk. At this point, an entire block of data (e.g., 1K worth) is transferred to or from the disk all at once. In this setting, it makes sense to buffer, or “cache” disk accesses. For example, instead of writing a
stream of data directly to disk, we write to a temporary buffer in memory that, when full, is sent to the disk with a single block transfer.

Suppose we have an array in memory that holds 100 elements of data. It takes 2000 units of time to write its contents to disk, clearing the array in the process. Our interface to this structure consists of a single operation, `write`, which inserts one element of data into the array. Each invocation of `write` takes only 1 unit of time, except every 100th invocation takes 2000 extra units of time (2001 in total).

Since worst-case invocations happen infrequently, we get a much more accurate understanding of performance by looking at worst-case running time amortized (i.e., averaged) over a sequence of invocations, rather than considering only the worst-case running time of a single, isolated invocation:

| The amortized running time of an operation is $f(n)$ if any sequence of invocations (say, $k$ of them) requires at most $kf(n)$ time. |

In our example above, `write` runs in 21 units of amortized time, since any sequence of $k$ `write` operations involves $\lceil k/100 \rceil \leq k/100$ “expensive” invocations, and therefore takes at most $k + 2000(k/100) = 21k$ total time.

**“Amortized” Versus “Worst-Case” Data Structures.** Suppose we have a choice between structure A, supporting `write` in 21 amortized units of time, and structure B, supporting `write` in 21 units of time in the worst case. We typically call A an “amortized” data structure and B a “worst-case” data structure\(^1\). The worst-case data structure may look much more appealing, since it avoids the occasional slow invocation we might experience with an amortized structure. However, this difference would only be noticeable in a “real-time” setting where the response time of every single invocation is crucially important. For example, a data structure for searching a library catalog in response to queries over the web should answer every query as quickly as possible. However, from the perspective of the total running time of a larger algorithm using the `write` operation, there is absolutely no difference whether we use A or B. Since the algorithm makes a sequence of calls (say, $k$ of them) to the `write` operation, it will spend at most $21k$ total units of time in either case. This is why amortized data structures are so useful. They are often far easier to design and implement than their worst-case counterparts, and they are every bit as good when serving as building blocks for a more sophisticated algorithm.

### 4.2 Methods for Amortized Analysis

The simplest way to perform amortized analysis, known as *aggregate analysis*, comes directly from the definition of amortized running time above. We first determine the worst-case total running time for an arbitrary sequence of $k$ invocations, and then divide by $k$. Unfortunately, this technique generally only works well for very simple data structures, where it is easy to figure out the total running time for an arbitrary sequence of invocations. For most non-trivial data structures, it can be quite hard to make this calculation, especially when the sequence contains several

\(^1\)Be careful not to let this nomenclature mislead you — remember that amortized bounds still give us worst-case guarantees, just over a sequence of invocations instead of for a single invocation.
different types of operations (e.g., \textit{insert} and \textit{delete}) interleaved in an arbitrary fashion.

\subsection{Accounting Tricks with Running Time}

As a financial term, amortization refers to paying off a large amount of money in small installments rather than as a single lump sum. If you need to pay $12,000 at the end of each year for a mortgage on a house, you might set aside $1,000 each month leading up to the due date of the payment, thereby amortizing the large payment into 12 smaller installments that are perhaps easier to manage from a bookkeeping standpoint. You still pay the same amount of money in the end. The only difference is that you account for some payments earlier than they are due (later is not allowed, since we would go into debt). Amortization in data structures is similar. We can view it as an accounting trick where we account for running time earlier than it actually happens, by mentally overcharging ourselves during earlier cheap invocations to generate a “credit” that is redeemed later to pay for expensive invocations.

In terms of the disk buffer example, we know that after 100 \textit{write} operations we will need to make a “payment” of 2000 units of running time to flush the contents of the buffer to disk. In anticipation of this, let us charge each \textit{write} operation 20 extra units of running time, so that by the 100th operation we will have built up 2000 units of credit, exactly enough to pay for the expensive invocation of \textit{write}. We get the same 21-unit amortized cost here for \textit{write} as we did with aggregate analysis, although our understanding of this cost is now based on management of credit through our accounting scheme. For a “cheap” invocation, 1 unit goes toward the actual immediate cost of the \textit{write}, and 20 units is invested as a credit. The author likes to picture a $20 bill sitting on every element in our buffer that represents a prepaid credit of 20 units of work that we can draw upon when needed. When we reach an “expensive” invocation, we charge ourselves the same 21 units, and now we can redeem all the credit sitting in our data structure to pay off the 2001 units of work required for this invocation. By smoothing out its non-uniformity, we now get a much simpler description of the running time of \textit{write}: 21 units of amortized cost for every invocation.

It is important to remember that the data structure is not changing. We are only playing accounting games with the analysis, accounting for some work slightly earlier than it happens. Just as in the financial example, we are only allowed to account for work earlier than it actually happens, never later (i.e., we are only allowed to build up a credit, not a debt). This ensures that for any sequence of invocations, the sum of the amortized running times we charge ourselves will properly bound the actual running time of the sequence. Our amortized running times may be “fictitious” numbers, but they still give legitimate worst-case performance bounds.

The approach above — leaving credits in certain places in our structure to pay for anticipated expensive operations — is sometimes known as the \textit{accounting method}. In the simplest possible case, we might overcharge the operation of inserting an element into a data structure for all future work we will spend on that element. In slightly more complicated cases, we may leave behind small amounts of credit in areas of a data structure any time they are modified, in order to pay for expensive
periodic “housekeeping” operations. Since we generally only need to perform expensive restructuring on areas of the data structure that have been heavily modified, these areas would presumably contain large amounts of credit.

### 4.2.2 Potential Functions

A more formal and mathematically rigorous way to perform amortized analysis is using a potential function, a function of the current state of our data structure that tells us the total amount of credit currently stored in the entire structure (this use of potential functions is similar to, but slightly more sophisticated than what we saw in Section 3.2.3, when we first introduced the use of potential functions to help compute running time). While the accounting method keeps track of credits on individual elements or in specific parts of a data structure, the potential function method lumps all of this into a single number. In our amortized buffer example, we could define a potential function \( \phi = 20x \), where \( x \) denotes the current number of elements in the buffer\(^2\). The potential function \( \phi \) expresses the same idea as with our earlier accounting method analysis, namely that each element in the data structure contributes 20 units of credit. However, we no longer think of leaving 20 units of credit on a particular element (e.g., a $20 bill), but rather we think of paying 20 units towards the total potential of the structure (e.g., $20 in the bank).

Amortized analysis is all about accounting for the cost of certain invocations earlier than they actually occur. This results in two types of invocations: “cheap” invocations that we overcharge in order to contribute credit into the data structure (so the amortized cost of the invocation is higher than its true immediate cost), and “expensive” invocations that we pay for using stored-up credit (so the amortized cost is lower than the true cost). Both of these ideas are captured in the following equation, which gives us an easy formulaic way to compute the amortized cost of each invocation of an operation once we have defined a potential function \( \phi \):

\[
a_j = c_j + (\phi_j - \phi_{j-1}),
\]

where \( a_j \) is the amortized cost of the \( j \)th invocation in some sequence of invocations, \( c_j \) is the true (immediate) cost of this invocation, and \( \phi_j \) denotes the value of our potential function right after the \( j \)th invocation (so \( \phi_j - \phi_{j-1} \) is the change resulting from the \( j \)th invocation).

To illustrate how easy this formula makes our analysis, consider the example of the amortized buffer. We can figure the amortized cost of write by considering two cases. For a “cheap” invocation, we have

\[
a_j = c_j + (\phi_j - \phi_{j-1}) = 1 + (20) = 21,
\]

since the buffer increases in size by one element, leading to a 20 unit increase in potential on top of the one unit of true cost we pay for the invocation. Here, we

---

\(^2\)One should always be able to determine the potential associated with a data structure by inspecting its present state, knowing nothing about the history of how we reached this state. Potential functions should therefore be defined only in terms of the current state of the data structure, not in terms of the historical sequence of operations we called to reach this state. For example, we should avoid defining the potential function above as “20 times the number of write operations since the last time the buffer was flushed”.

are overcharging the current operation so that we will have enough credit in our potential function to pay for future “expensive” invocations. When we reach such an “expensive” invocation, we then have

\[ a_j = c_j + (\phi_j - \phi_{j-1}) = 2001 + (-1980) = 21, \]

since the buffer decreases in size by 99 elements, leading to a 99 \times 20 = 1980 unit decrease in potential that offsets the high actual cost of emptying the buffer. In both cases, we obtain an amortized cost of 21, matching exactly the result we obtained using aggregate analysis or the accounting method. If we design our potential function correctly, our amortized costs usually balance out like this in the end.

The key to a successful potential function analysis is, not surprisingly, the design of a good potential function. Usually, we can do this by simply summing up the total credit we would get using the accounting method, but occasionally we will find a data structure for which the design of a good potential function requires a bit of divine inspiration (perhaps the best example being splay trees in Chapter 6). In addition to giving us the amortized costs we want (using the formula above), a potential function must always satisfy two important conditions: it must be non-negative (otherwise we would go into “debt”, having accounted for work later than it actually happens), and it must be zero initially. If these two conditions hold, then the total amortized cost of any sequence of \( k \) invocations is given by

\[ \sum_{j=1}^{k} a_j = \sum_{j=1}^{k} (c_j + \phi_j - \phi_{j-1}) = \left( \sum_{j=1}^{k} c_j \right) + \left( \sum_{j=1}^{k} \phi_j \right) - \left( \sum_{j=1}^{k} \phi_{j-1} \right) \geq \sum_{j=1}^{k} c_j, \]

so our fictitious amortized running times do indeed give us a proper upper bound on the total running time actually spent by the sequence of invocations.

Now that we are familiar with how to perform amortized analysis, we will work through a number of increasingly sophisticated examples for the rest of the chapter.

### 4.3 Example: The Min-Queue

Suppose we wish to augment a standard FIFO queue so it supports the operation \textit{find-min}, which identifies the element of minimum value in the queue. It is important that we can only locate this element but not remove it, since otherwise we would be dealing with a priority queue, a somewhat more ambitious undertaking (the entire next chapter is devoted to priority queues). This structure, which we call a \textit{min-queue}, should ideally support the operations \textit{enqueue}, \textit{dequeue}, and \textit{find-min} all in \( O(1) \) time. While it is difficult (albeit possible) to accomplish this in a worst-case setting, it ends up being reasonably simple to implement a min-queue so that all operations run in \( O(1) \) amortized time.

The min-queue is our first example of an amortized data structure where several different operations participate in the amortized analysis. In this case, the definition of amortized running time extends in a natural fashion. For example, with two operations \( A \) and \( B \), we would say that \( A \) has amortized running time \( f(n) \) and \( B \) has amortized running time \( g(n) \) if any sequence of \( a \) invocations of \( A \) and \( b \) invocations of \( B \) requires at most \( af(n) + bg(n) \) time. When multiple operations...
work together in an amortized data structure, our amortized analysis follows the same general idea as before: earlier cheap invocations of certain operations are overcharged to build up sufficient credit to pay for expensive invocations of other operations later on.

There are several ways to implement a min-queue. We describe one of them here, others\footnote{Some might consider it overkill to include discussion of five different techniques for a relatively minor data structure like a min-queue, but the author strongly believes that there is much insight to be gained from looking at the same problem or structure in multiple different ways.} in Section 4.5 and problems 70, 71, and 133. Suppose we maintain the elements in our queue in either a circular array or a doubly-linked list, no different than with a standard queue. As shown in Figure 4.1, new elements entering the back of the queue (the left side) are colored white, and elements being removed from the front (the right side) are colored gray. In general, if we scan from the back of the queue forward, there will be a block of white elements followed by gray elements. We maintain a pointer to the current minimum white element and the current minimum gray element, so the \texttt{find-min} operation is easy to implement in $O(1)$ time by comparing these two elements. Finally, each gray element maintains a pointer to the smallest gray element behind it in the queue, so if we happen to remove the minimum gray element, we can use its associated pointer to find the new minimum gray element. The only problem with this approach is that eventually we might remove all the gray elements, leaving only white elements. If we then happen to remove the minimum white element, we have no efficient means of locating the new minimum white element, since white elements are not equipped with extra pointers like gray elements. Therefore, if a \texttt{dequeue} operation ever finds itself trying to remove a white element, it first executes a $\Theta(n)$ recoloring step, where it scans through the entire contents of the queue from back to front, coloring every element gray and initializing its extra pointer.

Every min-queue operation takes $O(1)$ time, except for the occasional slow invocation of \texttt{dequeue} that runs in $\Theta(n)$ time due to the need to recolor. However, since an element will participate in at most one recoloring during its lifetime in the structure, we can overcharge the \texttt{enqueue} operation an extra $O(1)$ worth of time to pay for this eventual cost in advance. This example illustrates the power and simplicity of the accounting method: we simply overcharge each \texttt{enqueue} operation by an extra $O(1)$ worth of time, so every white element will have an $O(1)$ credit associated with it. These credits are sufficient to pay for the eventual $\Theta(n)$ operation of recoloring the min-queue when it contains $n$ white elements. It is quite straightforward to convert this idea into a more formal analysis using potential functions. [Further details]
4.3. EXAMPLE: THE MIN-QUEUE

**Problem 70 (An Alternative Min-Queue).** In this problem we analyze another simple way to implement the min-queue, shown in Figure 4.2. We store elements in a circular array or linked list, as with a standard queue. Every time we enqueue a new element \( e \), we scan forward from \( e \) and remove all elements ahead of \( e \) that are larger than \( e \) (since, due to \( e \)'s presence, these elements have no chance of ever becoming the minimum). As a result, the contents of the queue will be decreasing, making it easy to locate the minimum element (this also guarantees that the set of elements deleted during an enqueue operation will form a contiguous block). For example inserting 6 into a queue containing 9, 7, 3, 2, 0 would "shortcut out" the 9 and 7, leaving 6, 3, 2, 0. However, in order for dequeue to work properly, this process cannot just delete elements; rather, we "deactivate" them by moving them into linked lists between the active elements in our queue, so the FIFO order of all elements in the queue is preserved. Show that this implementation gives us an \( O(1) \) worst-case running time for dequeue and find-min, and an \( O(1) \) amortized running time for enqueue. [Solution]

**Problem 71 (Building a Min-Queue From a Pair of Min-Stacks).** Please show how to easily build a min-stack (a stack supporting push, pop, and find-min) with all operations taking \( O(1) \) worst-case time. Then, show how to use a pair of “back to back” min-stacks to build a min-queue. Further, show that this approach can be extended to allow for insertions and removals in \( O(1) \) amortized time at both ends of the queue, with find-min still taking \( O(1) \) worst-case time. [Solution]

**Sloppiness in Constants in Amortized Analysis.** When we perform amortized analysis, we often temporarily suspend the use of \( O(\cdot) \) notation for convenience. For example, in the min-queue we might say that a recoloring operation on \( n \) elements takes exactly \( n \) units of time, and that we overcharge each enqueue operation by 1 extra unit of time to pay for this cost. To be fully precise, we should really say that recoloring takes \( \Theta(n) \) time and that the enqueue operation is overcharged by some constant \( c \) units of time, where \( c \) is the hidden constant in the \( \Theta(n) \) time bound.
Figure 4.3: The middle block depicts the “equilibrium” state right after an expansion or a contraction, when \( n = B/2 \). Subsequent insertions (on the left) are overcharged by 2 units so they each contribute 2 units of credit (shown here as $2 bills). When our memory block fills up, it will therefore contain at least \( B \) units of credit, which is sufficient to pay for the cost of transferring its \( B \) elements during the next expansion. Subsequent deletions (on the right) are each overcharged 1 unit (shown as $1 bills placed on the slot left open by a deleted element), so that when \( n \) drops to \( B/4 \) we will have sufficient credit to pay the \( B/4 \) units required to transfer the contents of the block during the next contraction.

Unfortunately, this becomes awkward very quickly when we have many operations to consider (and hence many different constants to manage), so the simplest solution is typically just to pretend that every constant-time operation takes precisely 1 unit of time. The resulting bounds will still end up being asymptotically valid.

Being Lazy. Amortized data structures often gain simplicity by being “lazy”, only updating their internal state in one efficient batch operation at the last possible minute rather than investing extra work keeping it perfectly up-to-date all the time. With the min-queue for example, we don’t worry about maintaining extra augmented information for the white elements until this becomes absolutely necessary. This lazy outlook is quite common in data structure design.

4.4 Example: Dynamic Memory Allocation

We often don’t know in advance the total number of elements that we will end up storing in a data structure. This is rarely a problem for “pointer-based” structures such as linked lists and binary search trees (Chapter 6), where each element of data lives in its own individually-allocated block of memory. The positions of these blocks within the memory as a whole are not so important, since they are linked together in an appropriate fashion using pointers. In contrast, “array-based” data structures such as hash tables (Chapter 7) and of course, arrays, are somewhat more difficult to manage. These data structures must fundamentally be stored in a single contiguous block of memory, and if this block fills up, it may not be possible to expand it since memory on either side may have already been allocated for other purposes. In this situation, we usually allocate a new, larger memory block (typically twice as large), and transfer the entire contents of our data structure into the new block. This is a costly operation to perform, but since it occurs infrequently, we shall see that it
behaves well from an amortized perspective.

**Memory Allocation.** Before we proceed, let us say a few words on the subject of memory allocation. The underlying mechanism used by an operating system to allocate blocks of memory can be quite complicated, and it is not something we want to worry about too much as algorithm designers. For simplicity, we generally assume that all memory allocation requests require only \(O(1)\) time (this assumption is perhaps a bit optimistic, but not unreasonable in practice). Therefore, when an algorithm requests a block of memory of some specified size \(B\), it receives in \(O(1)\) time a pointer to some contiguous block of \(B\) words of memory that are not initialized in any way. To initialize the block, we could spend \(\Theta(B)\) time, or we could use the virtual initialization result of problem 2 to effectively reduce the initialization time to \(O(1)\) (although this is rarely done in practice, due to the extra overhead required).

**Block Expansion and Contraction.** Suppose we are storing an \(n\)-element stack within an allocated block of memory of size \(B\) (a stack gives us a simple concrete example, but the technique here is quite general and applies to most array-based structures). We would like to maintain the property that \(B = \Theta(n)\). In other words, we should always use only the amount of memory we actually need, at least to within some constant factor. The prototypical method for achieving this result is the following:

- If \(n\) grows as large as \(B\) due to insertions, we perform an *expansion* by transferring the structure into a newly-allocated block of size \(2B\).
- If \(n\) drops down to \(B/4\) due to deletions\(^4\), we perform a *contraction* by transferring the structure into a newly-allocated block of size \(B/2\).

It is easy to analyze the amortized performance of this scheme using the accounting method, as explained in Figure 4.3, and also to formalize this analysis using potential functions. [Details]

The expansion / contraction technique above contributes only an additional \(O(1)\) amortized time to the running time of *insert* and *delete*, so as long as you are content with adding “amortized” quantifiers to your running times, it is generally safe to assume that any \(n\)-element array-based data structure can be modified so that its underlying block of memory expands and contracts as necessary to ensure that the data structure always occupies \(\Theta(n)\) space. In fact, as we see in the next problem, we can sometimes even achieve this result in a worst-case setting.

**Problem 72 (De-Amortization).** In this problem we develop a systematic way to remove amortization from certain types of data structures, converting amortized bounds into worst-case bounds. Consider maintaining a queue on \(n\) elements, implemented using a circular array stored in a memory block that doubles or halves in size according to the

---

\(^4\)One might initially suggest halving the block size when \(n\) drops to \(B/2\) rather than \(B/4\), but this does not work well. If \(n = B - 1\), then the next insertion results in an expansion, after which \(n = B/2\) and a deletion would cause a contraction, after which \(n = B - 1\) and an insertion would cause an expansion, and so on. Each of the insertions and deletions in this alternating sequence takes \(\Theta(n)\) time, which is far worse than the \(O(1)\) amortized bound we are trying to obtain. Amortization is a wonderful technique, but it cannot magically reduce the running time of an operation that genuinely takes \(\Theta(n)\) time per invocation.
CHAPTER 4. AMORTIZED ANALYSIS

Problem 73 (Ordered File Maintenance). In this section, we addressed the difficulties posed by arrays when they need to expand or contract. Here, we tackle another drawback of arrays — their inability to support efficient insertion or deletion of interior elements, as this involves shifting around large amounts of memory to keep the array contiguous. Perhaps the best resolution of this problem will arrive in Chapter 6, when we see how to achieve $O(\log n)$ time for insert and delete by encoding a sequence in a tree. However, as an interesting exercise, let us consider what happens if we relax the constraint that an array must be contiguous. This leads to what is sometimes called an ordered file maintenance structure where we leave periodic “gaps” in an array, much like your library leaves periodic gaps in its shelving so that new books can be added with minimal re-shelving of existing books. As shown in Figure 4.4, suppose we break a length-$m$ array $A$ into $\sqrt{m}$ consecutive blocks of size $\sqrt{m}$, each of which behaves like the expanding / contracting array outlined above, holding between $\sqrt{m}/4$ and $\sqrt{m}$ elements (ideally $\sqrt{m}/2$ elements). The only difference is that when one of these blocks becomes too full or too empty, we rebuild the entire structure into a new freshly allocated memory buffer, rebalancing the allocation of its $n$ elements among the blocks by setting $m = 2n$ so that each block is half full. Observe that $m/4 \leq n \leq m$ always holds, so we are still space-efficient to within a constant factor ($m = \Theta(n)$). In an auxiliary array $B[0\ldots\sqrt{m}]$, we maintain in $B[j]$ the total number of elements stored in blocks $1\ldots j$ (so $B[j] - B[j-1]$ is the number of elements stored in just block $j$).

(a) Since $A$ contains gaps, it is no longer possible to perform random access — jumping directly to the $i$th non-gap element — in constant time. However, show that using the information in $B$, we can still locate the $i$th non-gap element in $O(\log n)$ time. Further, show that if we maintain the elements in $A$ in sorted order, we can still “binary search” for the element of a specific value in $O(\log n)$ time. [Solution]

(b) Show how to insert and delete in only $O(\sqrt{n})$ amortized time \footnote{Here is a subtle point, as this is the first non-constant amortized time bound we have seen: “$n$” usually denotes the size or number of elements in a data structure. An $O(n)$ amortized bound might seem somewhat ambiguous, since it refers to an average over a sequence of invocations during which $n$ may change. To remedy this, in non-constant amortized bounds, we usually take $n$ to represent an upper bound on the structure’s size during the period of analysis.} (using the result from (a) to find the desired location for the insertion/deletion). [Solution]

We can regard this structure, with $O(\sqrt{n})$ blocks of size $O(\sqrt{n})$, as a 2-level hierarchical decomposition of our array. By increasing the number of levels of hierarchy to the point where we essentially have a binary tree built on top of our array, a fancier generalization of the gap rebalancing approach above leads to $O(\log^2 n)$ amortized running times for insert and delete; see the endnotes for further details.
4.5 Example: Functional Queues

Functional programming, in languages like Lisp, Haskell, or OCaml, differs substantially from imperative programming in languages like C or Java. In a functional setting, objects in memory typically cannot be modified once created, and consequently we get the nice property that a function called with the same values always yields the same result (by contrast, a C function depending on external global variables might return different results if the global state changes). Functional algorithm design involves a much greater use of recursion, since iteration usually requires reassigning a loop variable again and again. The reader is highly encouraged to give functional programming a try, since this provides a very different mental perspective on algorithmic thinking that can dramatically sharpen your skills at recursion.

Rather than using arrays and random access, functional data structures are almost exclusively “pointer-based”, often built from nothing more than pairs of objects. A list can be built using a series of nested pairs (first element, (second element, (third element, ...))). This representation as a pair of (first element, rest of list) is ideal for recursive manipulation. For example, if \( f(L) \) is a function summing the contents of list \( L \), then \( f \) can just return \( \text{first}(L) + f(\text{second}(L)) \), where first and second refer to the first and second elements of the pair stored in \( L \). If \( S \) points to a list representing a stack, we can push a new element onto the stack by returning \((\text{new element}, S)\), and we can pop an element by returning \(\text{second}(S)\). Note that it is simple to work with \(k\)-tuples of elements as well as pairs, since these can be built from composition of a small number of pairs.

More precisely, we should say pairs of pointers to objects. Since objects don’t change after creation, a pointer is perfectly suitable as a stable reference to an object. It is much cheaper in terms of memory to store multiple references to the same object as pointers; making explicit copies of an object is never necessary, since the original object in memory will never change.
While the (first element, rest of structure) schema works quite well for “one-sided” data structures like stacks, we need to be more careful with “two-sided” structures like queues, where interaction occurs at both ends. For example, we might consider storing a queue as a 3-tuple \((F, M, L)\), where \(F\) and \(L\) are the first and last elements, and \(M\) points to the middle contents of the queue, recursively stored as another of these 3-tuples. This gives a simple recursive method to \textit{enqueue} (insert on the left) a new element \(x\), by returning \((x, \text{enqueue}(F, M), L)\). Unfortunately, this takes \(\Theta(n)\) time, since the recursion propagates through the entire structure.

We achieve better performance by introducing a small amount of “slack”, allowing \(F\) and \(L\) to hold zero or one elements. We also make \(M\) a recursive queue of \textit{pairs} of elements (so its inner structure is a queue of pairs of pairs of elements, and so on). To enqueue \(x\), we now return \((x, M, L)\) if \(F\) is empty, or otherwise if \(F\) already contains an element (say, \(y\)), then we return \((\text{empty}, \text{enqueue}((x, y), M), L)\), propagating the operation so that we now recursively enqueue the pair \((x, y)\) into \(M\). The \textit{dequeue} (delete from right) operation is symmetric, possibly causing a recursive dequeue from \(M\). Pictures of these operations are shown in Figure 4.5.

Although recursion might propagate quite far into the structure as before, our added slack causes lengthy propagation to happen infrequently, making \textit{enqueue} and \textit{dequeue} both run in only \(O(1)\) amortized time [Details]. By maintaining an additional “current minimum” field along with each \((F, M, L)\) triple, we get yet another means of building a min-queue with constant amortized performance bounds.

\textbf{Problem 74 (Functional Double-Ended Queues).} Suppose we want to be able to insert and delete at both ends of our queue. Show that the structure above no longer maintains its constant amortized performance bounds when used in this context, but that if we allow \(F\) and \(L\) to hold between zero and \textit{two} elements, then we once again can use the structure in a way that provides \(O(1)\) amortized guarantees. [Solution]

\textbf{Recursive Slowdown.} We can think of many data structures as being built in “levels” (in the example above, from the outermost level inward), where operations start at the first level and occasionally propagate to higher levels. As long as we can limit the rate of propagation between successive levels, amortization generally works. For example, note how our added slack causes the \textit{enqueue} operation to propagate to the next level only once in every two invocations. This is effectively like passing along only \(1/2\) a unit of work to level \(x + 1\) for every unit of work processed at level \(x\); and since \(1/2 + 1/4 + \ldots = 1\), each unit of original work therefore propagates to at most one total additional unit of work across all levels. This notion is sometimes known as \textit{recursive slowdown}, and it is a common feature of many amortized structures\(^7\).

Prototypical examples of recursive slowdown are shown in the following problems, where we repeatedly increment a binary counter or a permutation. Each increment causes a change to the least-significant digits, and these changes propagate but with increasingly slower frequency to more significant digits.

\(^7\)A mechanical analog of this idea appears in a fun kinetic sculpture by artist Arthur Ganson, where a motor turns the first of a series of twelve gears that each turn progressively slower, the final gear being embedded in a concrete block. As in our data structures, plenty of work is happening on one end, but very little of it propagates to the other side.
4.6. Example: Disjoint Sets

**Problem 75 (Enumerating Subsets by Incrementing a Binary Counter).** Suppose we store the digits of an $n$-bit binary number in an array $A$ of length $n$. We wish to increment our number from 0 to $2^n - 1$, updating $A$ accordingly. For example, if $n = 3$, we start with $A = (0, 0, 0)$, then we toggle the last entry to obtain $A = (0, 0, 1)$, eventually reaching $A = (1, 1, 1)$ after several increments. Please describe how to implement an increment operation on $A$ that runs in only $O(1)$ amortized time\(^8\) (assuming the entries in $A$ all start out at zero). One motivation for this problem is to quickly enumerate all $2^n$ subsets of an $n$-element set $S = \{1, 2, \ldots, n\}$. We can represent each subset with a length-$n$ array of zeros and ones (known as an *incidence vector*), where a one corresponds to an element present in the set and a zero corresponds to an element absent from the set. By incrementing our binary counter, we can enumerate through all successive subsets of $S$ in $O(1)$ amortized time per subset. [Solution]

**Problem 76 (Enumerating Permutations).** Consider a length-$n$ array $A = (1, 2, 3, \ldots, n)$. We would like to step through all $n!$ permutations of $A$, updating the array as we go to represent each subsequent permutation. Permutations should be generated in lexicographic order; for example, with $n = 3$ we start with $A = (1, 2, 3)$, then move to $A = (1, 3, 2)$, $A = (2, 1, 3)$, $A = (2, 3, 1)$, $A = (3, 1, 2)$, and finally $A = (3, 2, 1)$. Please describe how to implement an operation `next-perm` with $O(1)$ amortized running time that modifies $A$ to produce the next permutation in lexicographic order\(^9\). [Solution]

4.6 Example: Disjoint Sets

*Disjoint set* data structures (also sometimes called *union/find* structures) play an important role in a surprising number of algorithms, perhaps most notably Kruskal’s algorithm for the minimum spanning tree problem (Section ??). They maintain a collection of $n$ elements partitioned into non-overlapping sets (so each element belongs to precisely one set), and support the operations:

- **Make-Set**$(e)$. Add a new element $e$, belonging to a single-element set $\{e\}$.
- **Find-Set**$(e)$. Return a value identifying the set containing element $e$. This is usually a pointer to some “canonical” element in the set. Typically, we use `find-set` to determine if two elements $e$ and $e'$ are in the same set by checking if `find-set(e) = find-set(e')`.
- **Union**$(s_i, s_j)$. Join together sets $s_i$ and $s_j$, so all elements formerly belonging to these two sets now belong to a single larger set. The sets $s_i$ and $s_j$ are usually specified by providing their canonical elements.

4.6.1 List-Based Implementations

There are several simple ways to implement a disjoint set data structure. For example, if we store all the elements in one large array or list, we can augment each\(^8\)**Gray codes** (problem ??) give a method for generating all $2^n$ binary numbers in an $O(1)$ worst-case setting, where each successive number differs from the former in a single digit. We also mention in the endnotes an amusing way to obtain an $O(1)$ worst-case mechanism via the introduction of a fictitious “2” digit, which while “cheating” in some sense, still has useful applications.

\(^9\)**Heap’s algorithm** gives a method for generating all $n!$ permutations in an $O(1)$ worst-case setting, where each successive permutation differs from the former by a single swap. See the endnotes for references.
Figure 4.6: Disjoint set implementations: (a) a single large array or list of all elements, each pointing to the canonical element of its current set, (b) each set maintained as a doubly-linked list of elements, with the canonical element defined as the first element in each list, (c) each set maintained as a doubly-linked list of elements, with each element also pointing to the canonical element in its set, and (d) each set maintained as a tree, where the root is the canonical element.

element with a pointer to the canonical element in its set, as shown in Figure 4.6(a). This allows make-set and find-set to run in $O(1)$ time, but union($s_i, s_j$) takes $\Theta(n)$ time since we need to scan all the elements, updating any pointer to the canonical element of $s_i$ so that it now points instead to the canonical element of $s_j$ (or vice versa). In Figure 4.6(b), we maintain the elements in each set in a doubly-linked list, with the canonical element of each set being its first element. It is easy to join two such lists in $O(1)$ time (assuming we also maintain a pointer to the end of each list), so make-set and union take $O(1)$ time. However, now find-set takes $\Theta(n)$ time in the worst case, since it involves scanning from an element backward to the canonical element at the beginning of its list.

Union by Rank. We benefit in terms of efficiency by combining the two ideas above, shown in Figure 4.6(c). Sets are stored as doubly-linked lists, and each element also maintains a pointer to the canonical element in its set, allowing find-set to run in $O(1)$ time. Computing the union of two sets requires $O(1)$ time to physically join their lists, plus the time to scan one set and update its canonical element pointers so they point to the canonical element in the other set. In order to re-route as few pointers as possible, we use a natural heuristic called union by rank, where we re-assign the pointers in the smaller set, since this obviously involves less work (this typically requires keeping track of the sizes of our sets\textsuperscript{10}, which we store as annotations attached to their canonical elements).

Union by rank causes at most $\log n$ pointer reassignments to any particular element during its lifetime in the structure, since any time an element’s pointer changes,\textsuperscript{10}We can use the following clever trick to avoid maintaining augmented size information: during union($s_i, s_j$), we scan $s_i$ and $s_j$ simultaneously, stopping when we reach the end of the smaller set. This takes time proportional to the size of the smaller set, which we will be spending anyways during subsequent pointer re-assignment.

\textsuperscript{10}
the set containing that element must at least double in size (since its set is being merged with a larger set). More than \( \log n \) such doublings are impossible, since this would result in a set larger than \( n \) elements. By charging an element up front for all its pointer re-assignments, we get amortized running time bounds of \( O(\log n) \) for \textit{make-set} and \( O(1) \) for \textit{union}.

### 4.6.2 Tree-Based Implementations

Another elegant and popular approach involves storing the elements in each set as a rooted tree, as shown in Figure 4.6(d), with the root being the canonical element. Every element maintains only a single pointer to its parent (no child pointers are needed), so we can easily take the union of two trees in \( O(1) \) time by making the root of one point to the root of the other as its new parent. The only potentially slow operation is \textit{find-set}(e), which requires scanning upward from \( e \) until we reach the root of its tree. In the degenerate case where \( e \) is at the bottom of a tree shaped like a long path, this could take \( \Theta(n) \) time.

**Union by Rank.** We can improve the worst-case running time of \textit{find-set} to \( O(\log n) \) by keeping the height of each tree bounded at \( O(\log n) \), using a variant of \textit{union by rank}. We augment each root with a number we call its \textit{rank}, initially zero for a tree consisting of a single element. When linking together two trees, we make the smaller-rank tree a child of the root of the larger-rank tree. We can link in either direction in the case of a tie, in which case we increment the rank of the root of the resulting tree; this is the only situation where a rank ever increases. Note that the rank of a tree corresponds exactly to its height, so all we need to do is show that ranks cannot exceed \( \log n \). This is easily shown by arguing that the a tree of rank \( r \) must contain at least \( 2^r \) elements (so a tree of rank larger than \( \log n \) cannot exist, since it would contain more than \( 2^{\log n} = n \) elements). Observe that the only way to create a tree of rank \( r \) is to link two trees of rank \( r-1 \), which by induction on \( r \) both must contain at least \( 2^{r-1} \) elements, so the resulting tree must contain at least \( 2^r \) elements. After being created, a tree of rank \( r \) can only grow, if smaller-rank trees are linked into it; it can never shrink.

**Path Compression.** Our tree-based implementation also gains efficiency when we employ \textit{path compression}. Observe that there is no advantage whatsoever for our trees to have large depth. In fact, the ideal tree shape is that of set \( s_4 \) in Figure 4.6(d), where every non-root element links directly to the root. Path compression brings a tree closer to this perfect shape: after executing \textit{find-set}(e) and identifying the root of \( e \)'s tree, we make a second pass upward from \( e \), linking \( e \) and all its ancestors directly to the root. This does not change the asymptotic running time of \textit{find-set}, and it makes future \textit{find-set} operations potentially much faster.

**Problem 77 (Potential Function Analysis of Path Compression).** Please show that path compression (by itself, without union by rank) gives \( O(\log n) \) amortized bounds for \textit{find-set} and \textit{union}. In your analysis, use the potential function \( \phi = \sum_e \log s(e) \), where \( s(e) \) denotes the number of elements in the subtree rooted at element \( e \). As a hint: consider separately “big” versus “small” steps during the path compression process, where a big step from \( e \) up to its parent \( p \) occurs when \( s(p) \geq 2s(e) \). Note that there are at most \( \log n \) big steps during any call to \textit{find-set}, so paying for these directly is fine; since there can be many more small steps, these will need to be paid using potential. [Solution]
Combining Union by Rank with Path Compression. Something remarkable happens when we use both union by rank\footnote{Since the height of a tree can now shrink due to path compression, the rank of a root element no longer corresponds to the height of a tree. However, we still perform union by rank the same was as before: associate with every element a rank, and always link the smaller-rank root as a child of the larger-rank root when taking the union of two trees. In case of a tie, we can link in either direction, after which we increment the rank of the resulting root.} and path compression: the amortized running time of every disjoint set operation drops to only $O(\alpha(n))$, with $\alpha(n)$ being the extremely slowly-growing inverse Ackermann function we defined back in Section 2.1.3 (e.g., $\alpha(n) \leq 4$ if $n$ is the number of atoms in the observable universe). In fact, using the two-term variant of the inverse Ackermann function (also defined in Section 2.1.3) we can say something slightly stronger — that any sequence of $m \geq n$ operations takes $O(m\alpha(m,n))$ time on a disjoint set structure with $n$ elements. In particular, $O(\alpha(m,n)) = O(1)$ if $m$ is ever-so-slightly larger than $n$ in an asymptotic sense. For example, even if $m \geq n \log^* \cdot \cdot \cdot n$ (with any constant number of stars), then $O(\alpha(m,n)) = O(1)$ and we get a linear performance guarantee of just $\Theta(m)$ for our $m$ disjoint set operations. Surprisingly, despite the apparent mathematical complexity of the inverse Ackermann function, the amortized analysis of this structure is still relatively painless. \footnotesize{[Details]}

The original 1975 paper by Robert Tarjan describing this result (and also showing that the bounds above are tight) remarks that “This is probably the first and maybe the only existing example of a simple algorithm with a very complicated running time”. Indeed, even today, decades later, it is hard to find any other algorithm with comparable simplicity whose running time has such a complex mathematical characterization. Any time you encounter an $O(\alpha(n))$ running time in the world of algorithms, there is a good chance it originates from this data structure.

### 4.7 Example: Persistent Data Structures

Most data structures are ephemeral, maintaining only their present state. In this section, we show an elegant method for automatically transforming certain types of ephemeral data structures into persistent data structures, which allow interaction with historical versions of the structure. We focus on making a structure partially persistent, supporting updates to its present state as well as read-only queries against any past version of the structure (full persistence, where we also allow modification to past versions, is usually somewhat more involved).

Among its applications, partial persistence can be useful for transforming solutions to “offline” problems, where we know queries up front, so they work for “online” problems, where queries arrive later. If we know queries up front, an algorithm can scan its input, pausing at appropriate locations to answer queries. However, with persistence, we don’t need to know the queries in advance. We can go ahead and scan the input, and later when a query arrives, we can back up to the appropriate location in time to answer it. In Section ??, we apply this method to several prominent geometric “sweep line” algorithms.

The technique we describe below applies to any pointer-based data structure (e.g.,
a linked list or binary tree), where each node in the structure uses $O(1)$ memory and is pointed at by at most $k = O(1)$ other nodes\footnote{This requirement of constant indegree is satisfied by most pointer-based structures, such as linked lists ($k = 1$), doubly-linked lists ($k = 2$), and binary trees ($k = 3$ if we maintain a pointer to each node from its left child, right child, and parent). A notable exception is the class of disjoint set data structures from the preceding section, since their canonical elements often have large numbers of incoming pointers.}. All access to the structure starts at a designated “root” node, such as the first node in a linked list.

Suppose our original structure supports queries in $O(Q)$ time. We would like to add partial persistence without substantially slowing down update operations, while also keeping the query time close to $O(Q)$. To get started, here are several simple ways to achieve partial persistence that are not quite ideal in terms of efficiency:

- **Wholesale Copying.** Suppose after each update that we make a fresh copy of the entire structure, where the copies are organized by maintaining a time-sorted array of pointers to their roots. Although update time is absolutely terrible, query time is relatively fast at $O(Q + \log T)$, where $T$ denotes the number of historical versions we are tracking (i.e., the number of updates), and the additive $O(\log T)$ term comes from binary searching the length-$T$ array of root pointers to look up the root of the correct historical version.

- **Modification History in Nodes.** Let us store updates to nodes in the nodes themselves, so each node maintains its own private modification list in the form of a time-sorted array. Any time a node is updated, we add an entry to the end of its modification list, which stays sorted since modifications are conveniently made in increasing order of time. By storing these arrays using
the amortized doubling technique from Section 4.4, each node modification (which previously took \(O(1)\) time) now requires only \(O(1)\) additional amortized time, thereby only adding an “amortized” quantifier to our total update time. However, query time slows down a bit, to \(O(Q \log T)\), since each node examination requires not \(O(1)\) time as before, but \(O(\log T)\) time in order to binary search for the appropriate historical record in its modification list.

By using aspects of both approaches, we can achieve a highly effective solution in which updates gain an “amortized” quantifier, and queries take \(O(Q + \log T)\) time. To do this, we maintain a modification list at each node capable of holding at most \(k + 1\) entries, where \(k = O(1)\) is the maximum number of incoming pointers to any node. If we make \(k\) or fewer modifications to some node \(i\), we can therefore store the result within \(i\) itself, as shown in Figure 4.7(b). If we modify \(i\) again, then since \(i\)’s modification list is full, we leave this modification list in place and split off a new copy of node \(i\) with only a single entry in its modification list. As shown in Figure 4.7(c), this effectively moves the active version of node \(i\) to a new location in memory, so we need to make updates in all the nodes pointing at \(i\) so they now (as of this point in time) point to the new location. If any of these nodes also have full modification lists, then we split them as well, potentially resulting in a sequence of cascading splits that could be quite extensive in the worst case. However, since these happen infrequently, we can show that they contribute only \(O(1)\) additional amortized time to each original update operation. The only node we do not copy is the root node, whose modification list is allowed to grow without bound. The need to binary search this list at the beginning of each query is what contributes the additive \(O(\log T)\) to our query time. Aside from this, queries run in the same time as in the original structure. Later in problem 106, we will see how to achieve this same result using randomization instead of amortization.

This is another good example of the “recursive slowdown” idea we mentioned when discussing functional queues in Section 4.5, since the short modification lists attached to our nodes serve to buffer updates and hence slow their propagation backward to other nodes by an appropriate factor. In fact, functional data structures have the advantage of being naturally persistent, since historical versions of the structure are never changed. All we need to do is maintain a list of historical roots to be able to access any past version of the structure.

It is usually ill-advised to apply persistence to an already “amortized” data structure. For example, right before a periodic housekeeping operation, queries in an amortized structure might run quite slowly. This is usually not a problem, since the structure will speed up after housekeeping. With persistence, however, we can continue to issue historical queries to the data structure as of this inefficient state. Please therefore exercise caution when mixing amortization and persistence. In addition, the persistence technique above only works on data structures for which query operations are read-only. A data structure that re-configures itself on queries (e.g., the splay tree in Chapter 6) cannot do so in the read-only context of a historical query.
5. Priority Queues

Anyone who has ever waited impatiently in line, for example at the ticket counter of a crowded train station, is all too familiar with the notion of a queue. Standard queues exhibit First-In-First-Out (FIFO) behavior, where those entering the queue first are also the first to leave. There are some situations, however, where FIFO behavior is not desired. For instance, in the emergency room of a hospital, patients waiting for treatment are typically considered in order according to the urgency of their condition. This is known as a priority queue, where elements enter the queue with associated priorities, and the element with the highest priority is always the first to leave. The priority queue is one of the most fundamental types of data structures, one you are likely to encounter often in practice. In this chapter we discuss several ways to implement a priority queue, focusing in particular on those that are the most popular, the most efficient, and the most illustrative of elegant principles in data structure design.

To be considered a priority queue, a data structure must at the very least support two basic operations:

- *Insert*(\(e, k\)). Insert a new element \(e\) with key \(k\).
- *Remove-Min*. Remove an element with minimum key from the priority queue, and return this element.

Every element stored in a priority queue has an associated key, and by convention elements with lower keys are usually taken to have higher priority, although if desired, it is just as easy to adopt the opposite convention. Several other operations are commonly supported by priority queues:

- *Build*(\(e_1 \ldots e_n, k_1 \ldots k_n\)). Constructs a priority queue in a batch setting on \(n\) elements \(e_1 \ldots e_n\) and their corresponding keys \(k_1 \ldots k_n\).
- *Find-Min*. Return a pointer to, but do not remove, an element with minimum key.
- *Decrease-Key*(\(e, \Delta k\)). Decrease the key of element \(e\) by \(\Delta k\).
- *Increase-Key*(\(e, \Delta k\)). Increase the key of element \(e\) by \(\Delta k\).
Delete(e). Remove a specified element $e$ from the priority queue.

Due to redundancies among these operations, we may not need to implement all of them:

- We can implement remove-min using find-min followed by delete,
- We can implement find-min by calling remove-min followed by insert to put the element that was removed back into the priority queue,
- We can implement both decrease-key and increase-key using delete and insert by removing an element and re-inserting it with a new key, and
- We can implement delete by calling decrease-key to lower the key of an element so it becomes the minimum, then by calling remove-min.

Often, a priority queue maintains only pointers to data records, rather than the records themselves. This can improve efficiency, especially if our data records are large, since we need to shuffle less memory around. If our data records are already stored in some other data structure (as is often the case), we can view our priority queue as a lightweight indexing structure built “on top of” this existing structure, giving it the added functionality of a priority queue. Most important, however, is that not unlike your digestive system, once an element is inserted into a priority queue, it is generally not easy to access until it re-emerges from the other side, in response to a call to remove-min. If we need to access elements from outside the priority queue (say, if we need to call decrease-key or delete), we need to maintain pointers to them from the outside, since priority queues do not support an efficient means of finding elements (by contrast, dictionary data structures, discussed extensively in the next two chapters, are specifically designed to efficiently find elements based on their keys). Therefore, if $e$ is an element stored inside a priority queue representing data record $d$ stored outside the structure, we typically maintain a pointer from $d$ to $e$ so that we can still access $e$ if needed. Having said all of this, for simplicity we henceforth ignore this issue and pretend that we are storing just a set of keys in our priority queue.

5.1 Priority Queues and Sorting

Priority queues and sorting share much in common. We can easily sort $n$ elements by building a priority queue on them and calling remove-min $n$ times, causing the elements to exit the queue in sorted order. We will shortly see a very nice sorting algorithm, heap sort, based on this technique.

Recall that we divided sorting algorithms into two main categories: input-insensitive algorithms that work in the comparison-based or real RAM models of computation, and input-sensitive algorithms that sort integers in the RAM model. Priority queues can be similarly classified, and we cover both types in this chapter, starting with comparison-based structures. In this setting, the $\Omega(n \log n)$ worst-case lower bound for comparison-based sorting implies that insert or remove-min must run in $\Omega(\log n)$ worst-case time, or else we could sort in faster than $O(n \log n)$ time using $n$ calls to insert followed by $n$ calls to remove-min.
A priority queue is stable if it acts like a FIFO queue for equal elements, causing them to exit in the same order they enter. When used to sort, stable priority queues give us stable sorting algorithms. We can make any priority queue stable by augmenting elements with sequence numbers giving the time they were inserted, using these to break ties among equal elements.

### 5.2 Unsorted and Sorted Arrays

One can implement a simple priority queue using nothing more than an array, possibly maintained in sorted order. The results are not terribly efficient, but they serve as a good baseline for our initial discussion:

**The Unsorted Array.** By storing elements in an unsorted array, we can easily insert in $O(1)$ time by appending to the end of the array. However, remove-min takes $\Theta(n)$ worst-case time since we need to scan the array to find the minimum element. Once we have found the minimum element, we can delete it (or any other element) in $O(1)$ time by swapping it with the final element in the array and decreasing the array size — this avoids the problem of leaving a “gap” in the array. Decrease-key and increase-key both obviously run in $O(1)$ time.

**The Sorted Array.** By maintaining elements in (say, decreasing) sorted order, we achieve the opposite tradeoff. The minimum will always be the final element, so remove-min takes only $O(1)$ time. However, now insert, delete, decrease-key, and increase-key all take $\Theta(n)$ worst-case time, in order to keep the array sorted.

The results above are summarized in Figure 5.1. Linked lists give us essentially the same performance guarantees, except they make deletion slightly easier in the sorted case. In both cases, however, there is a dramatic tradeoff between the fundamental operations insert and remove-min, with one always running in $\Theta(n)$ worst-case time. Our next data structure, the binary heap, provides some middle ground by supporting all operations in $O(\log n)$ time.
5.3 Binary Heaps

Binary heaps are probably the most well-known and commonly-used priority queue data structures, owing to their simplicity, speed, and ease of implementation. Since these are such common objects in computer science, you will often hear them simply called “heaps”. However, the term heap refers more generally to a family of tree data structures that satisfy the heap property: \( \text{key}(\text{parent}(e)) \leq \text{key}(e) \) for every element \( e \). As shown in Figure 5.2(a), the heap property imposes a “vertical” ordering on a tree, forcing the smallest element to reside at the root. Since the minimum element is in such an obvious place, heap-ordered trees are common in many priority queue implementations.

5.3.1 Storing a Binary Heap in an Array

An \( n \)-element binary heap is an array \( A[1...n] \) that we mentally visualize as a heap-ordered binary tree, as shown in Figure 5.2. The tree has a special shape, being almost complete, with every level completely filled in except the last, which is filled from left to right up to some point. An almost complete tree can be mapped to our array \( A \) in a natural fashion: the root corresponds to \( A[1] \), the two children of the root to \( A[2] \) and \( A[3] \), the four grandchildren of the root to \( A[4...7] \), and so on. Moreover, we can easily jump around in the array \( A \) to mimic movement in the tree. For any element \( A[i] \) in the array, its parent in the tree has index \( \lfloor i/2 \rfloor \) and its two children are located at indices \( 2i \) and \( 2i + 1 \) (for those accustomed to zero-based indexing of arrays, you will need to change these formulas slightly).

The fact that we can visualize, navigate, and manipulate the structure like a tree while storing it in a nothing more than a single lightweight array is one of the main advantages of the binary heap. By contrast, most other tree-based data structures are somewhat more cumbersome to implement, since each node is stored in an individually-allocated block of memory, and each node maintains pointers to its parent and children.

Note that the array representing a heap is not sorted or even nearly sorted (e.g., it can have \( \Theta(n^2) \) inversions), even though the heap property stipulates that small elements generally precede larger elements, with \( A[1] \) being the smallest overall.
5.3.2 Operations on a Binary Heap

It is straightforward to implement all standard priority queue operations on a binary heap after first building two fundamental operations called *sift-up* and *sift-down*. These operations are common to all heaps, not just binary heaps, and are designed to move around elements in the heap so as to enforce or restore the heap property. Their names give us some intuition behind their functionality: *sift-up* causes small “light” elements to drift upward toward the top of the heap, and *sift-down* forces large “heavy” elements to sink down toward the bottom.

*sift-up*$(i)$ takes $A[i]$ and pushes it up the tree as far as possible by repeatedly exchanging it with its parent, as long as the parent element has a larger key. Calling *sift-up*$(i)$ will repair a single violation of the heap property between $A[i]$ and its parent. Similarly, *sift-down*$(i)$ will repair a violation of the heap property between $A[i]$ and its children, by pulling $A[i]$ as far down in the heap as possible by repeatedly swapping it with its smallest child (as long as this child has a smaller key). Both *sift-up* and *sift-down* run in $O(\log n)$ time on a binary heap, since they both require at worst time proportional to the height of the tree, and an almost-complete binary tree on $n$ elements has height $O(\log n)$.

The remaining priority queue operations are now very simple to implement using *sift-up* and *sift-down*:

- To *insert* a new element, we increment the size of the heap, $n$, place the new element in $A[n]$ (corresponding to the next open slot in the bottom row of the heap), and call *sift-up* on it, to fix any potential violation of the heap property with its parent.

- When we perform a *decrease-key* operation on $A[i]$, this might also potentially break the heap property between $A[i]$ and its parent, so we call *sift-up*$(i)$.

- Similarly, *increase-key* calls *sift-down* on an element after increasing its key, since we may need to correct a potential violation of the heap property with its children.

- *Remove-min* swaps $A[1]$ and $A[n]$ and decrements $n$, so the element we want to remove is deposited at the end of our array, now one position beyond the end of the heap. We are left with an element at the root (formerly $A[n]$) that might now violate the heap property with its children, so we call *sift-down* on the root.

- *Delete* is similar to *remove-min*. We delete $A[i]$ by swapping it with $A[n]$ and decrementing $n$, then calling both *sift-up*$(i)$ and *sift-down*$(i)$ to correct any potential heap violations created by substituting an arbitrary element (formerly $A[n]$) in place of $A[i]$.

All of these operations call either *sift-up* or *sift-down* or both, so they all run in $O(\log n)$ time. [Animated explanation of heap operations]

**Building a Binary Heap in Linear Time.** To build a binary heap on $n$ elements, we could start with an empty heap and make $n$ calls to *insert*, although this takes $\Theta(n \log n)$ time in the worst case where we insert elements in decreasing order, with
every element being pulled all the way up to the root by \textit{sift-up}. Surprisingly, there is an even better way to build a binary heap in only $\Theta(n)$ time: starting with our $n$ elements in an unordered array $A[1 \ldots n]$, we simply call \textit{sift-down}(i) on each element $i$ from $n$ down to 1 in sequence. It is not immediately obvious why this approach would build a valid binary heap or that it runs in $\Theta(n)$ time, but we can show both using a bit of careful analysis. \cite{Careful analysis}

Note that the approach above builds a binary heap \textit{in place}, converting an arbitrary array into a heap-ordered array. Also note that the fast $\Theta(n)$ running time for \textit{build} does not violate the $\Omega(n \log n)$ lower bound for comparison-based sorting, since it still takes $O(n \log n)$ time to remove the elements from a binary heap in sorted order.

\textbf{Problem 78 (Random Insertion in Binary Heaps).} Please show that it takes only $\Theta(n)$ time in expectation to insert $n$ elements into a binary heap in random order (compared to the $\Theta(n \log n)$ worst-case running time we get from inserting in decreasing order). As a hint, try to apply the result of problem 18(b) to various locations in the structure. \cite{Solution}

\textbf{Problem 79 (Heaps of Heaps).} In this problem, we reduce the running time of \textit{insert} to $O(1)$ amortized time\footnote{Achieving $O(1)$ amortized time for \textit{insert}, by itself, is actually not usually a big win in terms of total running time, since every element is usually the victim of a \textit{remove-min} call down the road, so it still incurs $O(\log n)$ work during its lifetime in the structure.}. Suppose we are “lazy” and insert elements in $O(1)$ time into a temporary array $T$ rather than into our priority queue. When \textit{remove-min} is called, we build a new binary heap out of $T$ in linear time. As a result, we end up maintaining a large number of binary heaps, one representing each run of \textit{insert} operations between successive calls to \textit{remove-min}. How can we tie these together into one structure supporting \textit{insert} in $O(1)$ amortized time and \textit{remove-min} in $O(\log n)$ amortized time? \cite{Solution}

\textbf{Problem 80 (B-Heaps).} A natural generalization of the binary heap is the $B$-heap, which is structured as an almost-complete $B$-ary tree rather than an almost-complete binary tree, so every node in the tree has $B$ children rather than 2 children. It is easy to generalize the operations of a binary heap to work on a $B$-heap, as well as the “bottom up” method for building the structure in $\Theta(n)$ time. Please comment on what effect, if any, this generalization will have on the running time of \textit{sift-up}, \textit{sift-down}, and the fundamental priority queue operations. Next, suppose we have an application that is likely to perform $k$ times as many invocations of \textit{decrease-key} as \textit{remove-min} (for example, Dijkstra’s shortest path algorithm calls \textit{decrease-key} $m$ times and \textit{remove-min} $n$ times on a graph with $n$ nodes and $m$ edges). What is the best choice for $B$ in this case? \cite{Solution}

\subsection{5.3.3 Heap Sort}

We have already described how to sort using a generic priority queue: first \textit{build} the queue and then perform $n$ consecutive \textit{remove-min} operations. In a binary heap, this leads to a particularly nice result: after building a binary heap out of an unsorted array $A[1 \ldots n]$, the array actually ends up in decreasing sorted order as a “side effect” of calling \textit{remove-min} $n$ times. The first \textit{remove-min} swaps $A[1]$ (the minimum) with $A[n]$ and then decrements the size of the heap, leaving the minimum at the end of the array. The next \textit{remove-min} swaps $A[1]$ (the second-smallest element) with $A[n-1]$, and so on. Of course, if we wanted our array to end up in forward sorted order, we could have used a “max” binary heap, from which we always remove the maximum instead of the minimum.
This algorithm, called *heap sort*, is easy to implement, takes $O(n \log n)$ time, sorts in place, and runs very quickly in practice. Heap sort is not stable since the binary heap is not a stable heap, but if in-place operation and a deterministic $O(n \log n)$ running time are important, heap sort may well be the sorting algorithm of choice. Its closest competitor with these features, the deterministic $O(n \log n)$ variant of quicksort, runs much slower in practice.

### 5.4 Mergeable Priority Queues

In order to develop more efficient data structures than the binary heap, we need to enter the realm of the mergeable priority queues (sometimes called meldable priority queues). In addition to the standard operations from before, a mergeable priority queue supports the new operation *merge* (sometimes called *meld*), taking two priority queues and merging them into a single priority queue, after which they no longer exist as separate individual structures.

In this section, we discuss two main classes of mergeable priority queues. The first class consists of nearly half a dozen highly-related data structures (randomized mergeable binary heaps, leftist heaps, skew heaps, and several others) that are built from a single heap-ordered tree. The second class contains priority queues that are built from multiple heap-ordered trees of different sizes and shapes, including the binomial heap and its more sophisticated relative, the Fibonacci heap.

#### 5.4.1 Randomized Mergeable Binary Heaps

While the binary heap itself does not support an efficient *merge* operation, there are many close relatives that do. The most elegant of the bunch, in the author’s opinion, is the *randomized mergeable binary heap*, which performs all priority queue operations, including *merge*, in $O(\log n)$ time with high probability.

The randomized mergeable binary heap is nothing more than a heap-ordered binary tree. It can be have arbitrary shape and may be quite unbalanced\(^2\), so instead of storing it in a simple array like the binary heap, we store it like most other trees. Each element resides in its own block of memory, and maintains a pointer to its parent, left child, and right child. Since the tree height may be significantly larger than $O(\log n)$, the use of *sift-up* and *sift-down* is no longer prudent. Fortunately, we can now abandon *sift-up* and *sift-down* entirely and implement every priority queue operation in a very simple fashion using *merge*:

- To *insert* a new element into a heap $H$, we construct a new single-element heap and merge it with $H$.
- We implement *remove-min* by removing the root element and merging its two former child subtrees (which are themselves valid heap-ordered trees).

\(^2\)In the next chapter, we will spent quite a bit of effort trying to keep binary trees balanced for performance reasons. By contrast, the heap-ordered trees in this section can be quite unbalanced with no negative consequences. Even a degenerate tree in the shape of a long $n$-element sorted path would be acceptable.
• To decrease the key of an element $e$, we splice out the subtree rooted at $e$, decrease $e$’s key, then merge this subtree back into the original tree. Since $e$ is a root element when its key is reduced, there is no danger of violating the heap property.

• Deleting element $e$ is accomplished by replacing $e$ with the result of merging its two child subtrees.

• We can implement increase-key in terms of delete and insert.

Since all of these operations involve a constant number of calls to merge, they all run in $O(\log n)$ time with high probability as long as the same is true for merge.

Randomization gives us an extremely simple approach for merging two heaps $H_1$ and $H_2$. Assuming $H_1$ has the smaller root, its root becomes the root of the merged tree, and we merge $H_2$ recursively into one of its child subtrees. As shown in Figure 5.3, if a fair coin toss results in heads, we recursively merge $H_2$ with the left subtree of $H_1$. If we see tails, we recursively merge $H_2$ with the right subtree of $H_1$. The process ends when we reach the bottom of one of the trees and try to merge some heap $H_1$ with an empty heap $H_2$, in which case the result is just $H_1$. The fact that this runs in $O(\log n)$ time with high probability follows immediately from the randomized reduction lemma: with probability $1/2$, we choose to merge with the smaller of $H_1$’s two subtrees, in which case we effectively reduce the size of $H_1$ to at most half of its current value.

5.4.2 Leftist Heaps, Skew Heaps, and Other Relatives Based on Null Path Merging

Another natural way to merge two heap-ordered trees $H_1$ and $H_2$ is as follows: select a path in $H_1$ from its root down to an “empty space” at the bottom of the tree (known as a null path), select a similar path in $H_2$, and merge along these paths, as shown in Figure 5.4. Due to the heap property, null paths are sorted from top to bottom, and consequently the process of merging along these paths is completely analogous the familiar process of merging two sorted sequences.

One way to merge two sorted sequences $s_1$ and $s_2$ (say, with $s_1$ having the smaller initial element) is by taking the initial element of $s_1$ followed by the result of recursively merging the rest of $s_1$ with $s_2$. Similarly, we merge heaps $H_1$ and $H_2$ along null paths (say, with $H_1$ having the smaller root), by recursively merging $H_2$ with either the left or right subtree of $H_1$, depending on the direction of the null path in
5.4. MERGEABLE PRIORITY QUEUES

Figure 5.4: Merging two heap-ordered trees along a null path in each tree. Rather than using “null” pointers to indicate the lack of a child at the bottom of a tree, we have marked the bottom of each tree using dummy “sentinel” elements.

$H_1$. If you prefer the iterative outlook on merging two sequences, a similar process works for heaps, where we step two pointers in tandem down our null paths, always taking a step from the pointer to the smaller element, splicing the heaps together as we go. Both the recursive and iterative approaches are completely equivalent. As in the case of merging two sequences, they are just two different ways of looking at the same process.

Since the amount of time required to merge two heap-ordered trees along null paths is proportional to the combined heights of these paths, we clearly want to select two paths of low height in order to merge quickly. The randomized mergeable binary heap does this in perhaps the simplest possible fashion by choosing paths at random — moving left or right at each step according to the result of a fair coin flip. This gives null paths of length $O(\log n)$ with high probability, thanks to our earlier analysis with the randomized reduction lemma. There are several other common alternatives for choosing good null paths, however, which we outline below.

Size-Augmented and Null-Path-Length Augmented Heaps. We could “de-randomize” the randomized mergeable binary heap by augmenting each element in our heap with a count of the number of elements in its subtree. If we then choose a path by repeatedly stepping to whichever of our child subtrees has a smaller size, this gives a deterministic approach for finding a path of height at most $\log n$, since every step downward at least halves the size of our current subtree. Similarly, we
could augment each element \( e \) with its null path length, \( npl(e) \), which gives the length of the shortest null path downward from \( e \). By repeatedly stepping to whichever child has a smaller value, this also gives a deterministic approach for finding a null path of height at most \( \log n \).

**Leftist Heaps.** The leftist heap is a heap where we augment each element with its null path length and also maintain the invariant that \( npl(\text{left-child}(e)) \geq npl(\text{right-child}(e)) \) for every element \( e \). This results in a tree that is somewhat “left heavy” in which the best way to find a path for merging is to follow the right spine of the tree, which has height at most \( \log n \). We therefore always merge two trees by merging along their right spines, and then walk back up the right spine of the merged tree swapping children anywhere necessary to restore our invariant. This approach is essentially the same as our approach above where we augment each element with its null path length, except we always treat the child with smaller null path length as the right child.

**Skew Heaps.** The skew heap is a relaxed, amortized cousin of the leftist heap that is simpler to implement (perhaps even almost as simple as the randomized mergeable binary heap) but slightly more intriguing to analyze. Like the leftist heap, we merge two heaps by merging their right spines and then adjusting the structure of the merged tree slightly. The readjustment step, however, is now much easier: we just walk up the right spine of the merged heap and swap the two children of every element along the way (except the lowest). As one can prove, this results in the tree being sufficiently “leftist” that merge (and hence every other priority queue operation) runs in \( O(\log n) \) amortized time. [Detailed analysis of skew heaps]

An alternative view of the skew heap that perhaps illustrates its operation more clearly is the following: let us imagine that every element is augmented with a coin whose state is either heads or tails. To merge two heaps \( H_1 \) and \( H_2 \) (with \( H_1 \) having the smaller root), we look at the coin at the root of \( H_1 \); if heads, we merge \( H_2 \) recursively with the left subtree of \( H_1 \), and if tails, we merge with the right subtree. We then flip over the coin on the root of \( H_1 \), toggling its state. Intuitively, this would seem to keep \( H_1 \) somewhat “balanced”, by alternatively merging into its left and right subtrees. Indeed, if you think carefully about the operation of this structure, you will see that it behaves exactly the same as the skew heap as described above; we have only described it in a top-down recursive manner\(^3\). This top-down description also highlights the similarity between the randomized mergeable binary heap and the skew heap, with the only difference being that where the randomized mergeable binary heap flips a coin, the skew heap flips the coin over.

Skew heaps and their distant relatives splay trees (Section 6.2.7) are known as self-adjusting data structures since they can (somewhat miraculously) continually adjust their structure so as to remain efficient despite maintaining no additional augmented information to help them in this task. In fact, among the five different “null path” mergeable heap implementations we have seen above, the randomized mergeable binary heap and skew heap stand out in that they do not require us to keep any augmented state in our heap. The other approaches require augmenting elements with either null path lengths or subtree sizes, and while this extra informa-

\(^3\)To make the correspondence even more direct, we can always recursively merge with the right subtree, and then swap the left and right subtrees. This gives the same mechanics of merging with alternating subtrees, but avoids the need to explicitly store the state of the coin.
5.4. MERGEABLE PRIORITY QUEUES

Figure 5.5: Recursive construction a rank-4 binomial tree by (a) linking two rank-3 binomial trees, or (b) joining rank-0 through rank-3 trees as siblings under a new common root.

Operation can be easily updated with no running time penalty during the merge process, it does cause trouble for the operations delete, decrease-key, and increase-key, as we discover in the following problem.

**Problem 81 (Lazy Deletion).** Consider the three approaches we discussed above: size-augmented heaps, null-path-augmented heaps, and leftist heaps. In each of these, delete, decrease-key, and increase-key are problematic due to the need to maintain augmented information. Why is this, and how can we fix this issue by being somewhat “lazy” so that all priority queue operations run in $O(\log n)$ amortized time? [Solution]

5.4.3 Binomial Heaps

In this section we describe the binomial heap, an elegant priority queue data structure that performs all fundamental operations including merge in $O(\log n)$ time in the worst case. Using the binomial heap as a stepping stone, we then describe the Fibonacci heap, a more sophisticated relative designed to speed up decrease-key to run in $O(1)$ amortized time.

In all of the tree-based priority queues we have seen so far, elements are stored in a single heap-ordered tree. By contrast, the elements in a binomial heap are divided up into a collection of heap-ordered trees. Each element may potentially have many children, stored in a doubly-linked list. Therefore, along with each element we store its rank (number of children), a pointer to its parent, a pointer to its first child, and pointers to its previous and next siblings.

The trees in a binomial heap come in specific shapes, known as binomial trees, that are built in a recursive fashion, as shown in Figure 5.5. There is a unique shape associated with every rank, where the rank of a tree is the number of children of its root element. A rank-0 tree consists of a single element. More complicated trees are recursively constructed: we either link two trees of rank $j-1$ (one as a child of the other) to obtain a rank $j$ tree, or we join trees of ranks $0 \ldots j-1$ under a new root to obtain a tree of rank $j$. It is not hard to see from this construction that a tree of rank $j$ has depth $j$ and contains $2^j$ elements, $2^{j-1}$ of which are leaves. The name “binomial” tree comes from the fact that number of elements at depth $d$ in a rank- $j$ tree is exactly the binomial coefficient $\binom{j}{d}$.

Binomial trees arise in a number of interesting algorithmic situations. For example,
suppose you broadcast a message from a single source node to all other nodes in
a network in a number of rounds, where in each round, every node that has heard
the message so far transmits the message to a distinct node that has not yet heard
it. The resulting transmission pattern will have the form of a binomial tree. In
Chapter 8, we will also study a close relative of the binomial tree known as the
binary indexed tree.

As shown in Figure 5.6, a binomial heap is built from a collection of binomial trees,
at most one of each rank, whose roots are all connected together in a doubly-linked
list. It is clear that there can be at most \( \log n \) such trees represented in the list,
since a tree of rank larger than \( \log n \) would contain more than \( 2^{\log n} = n \) elements.
Since all of the trees in the binomial heap satisfy the heap property, the minimum
element must reside at one of their root elements, so we can find it in \( O(\log n) \) time
by scanning the root list.

Owing to the fact that binomial trees have sizes that are powers of two, there is a
only one unique configuration of trees that represents a valid binomial heap on \( n \)
elements, corresponding precisely to the binary representation of \( n \). As shown in
Figure 5.6, if we form a binary number in which the \( j^{th} \) bit indicates the presence
or absence of a tree of rank \( j \), this gives the binary representation of \( n \).

We will describe how to merge two binomial heaps in \( O(\log n) \) time in a moment, but
let us first see how easy it is to write the remaining fundamental operations in terms
of merge. To insert an element in \( O(\log n) \) time, we build a new 1-element binomial
heap and merge it with our existing heap. Remove-min is also simple to express
in \( O(\log n) \) time in terms of merge. Observe from Figure 5.6 that if we remove the
root corresponding to the minimum element, the linked list of child subtrees of this
root is itself a valid binomial heap. Therefore, to remove the minimum element, we
splice out the root containing the minimum element and merge the binomial heap
consisting of its children back into the main heap. Decrease-key works the same as
in a regular binary heap, by using sift-up. Notice that sift-up doesn’t change the
shape of any of our trees, and it runs in \( O(\log n) \) time since every element belongs
to a tree of height \( O(\log n) \). We can write delete in terms of decrease-key followed
by remove-min, and we can write increase-key by deleting an element and then
re-inserting it with a new key. Note that we could implement increase-key using
sift-down, but this would be somewhat slower (\( O(\log^2 n) \) time) since each node has
up to \( \log n \) children; recall from problem 80 that as opposed to sift-up, sift-down
becomes slower when nodes have more children.

All that remains is to show how to merge two binomial heaps \( H_1 \) and \( H_2 \) in \( O(\log n) \)
time. If \( H_1 \) and \( H_2 \) have \( n_1 \) and \( n_2 \) elements, the merge process corresponds exactly
to the binary addition of \( n_1 \) and \( n_2 \). This makes intuitive sense, because the root
lists in \( H_1 \) and \( H_2 \) reflect the binary composition of \( n_1 \) and \( n_2 \), and we are producing
a merged heap whose root list needs to reflect the binary composition of \( n_1 + n_2 \).
When adding two binary numbers, we add each bit position starting from the least
significant; when we find several 1 bits in position \( j \), we add two of them to form
a carry bit that is added to position \( j + 1 \). Translated to our merge operation, we
splice together the two root lists, starting from the smallest rank, and whenever we
find more than one tree of rank \( j \), we link them together in \( O(1) \) time to form a
“carry” tree of rank \( j + 1 \). [Detailed explanation of the merge operation]

If we insert successive elements into a binomial heap, the resulting merge operations
5.4. MERGEABLE PRIORITY QUEUES

Figure 5.6: Illustration of a binomial heap: a linked list of binomial trees, at most one of each rank. Here we are storing \( n = 21 = 10101_2 \) elements, and since \( 21 = 2^4 + 2^2 + 2^0 \), we need to use a tree of rank 4, a tree of rank 2, and a tree of rank 0. The dashed rectangle illustrates that the children of a root node form a valid binomial heap by themselves.

are identical in structure to the process of incrementing a binary counter, which takes only \( O(1) \) amortized time (problem 75). Therefore, successive insertions into a binomial heap take only \( O(1) \) amortized time each, so we can build a binomial heap in \( \Theta(n) \) time using \( n \) calls to \( \text{insert} \).

5.4.4 Fibonacci Heaps

The Fibonacci heap cleverly extends the binomial heap to support \( \text{decrease-key} \) in \( O(1) \) amortized time, which yields improvements in efficiency for several important algorithms, such as Dijkstra’s shortest path algorithm and Jarník’s (Prim’s) minimum spanning tree algorithm. However, these gains tend to be more theoretical than practical, as they rarely outweigh the added overhead of the Fibonacci heap (for example with Dijkstra’s shortest path algorithm, the author has not yet found even a single real-world scenario where Fibonacci heaps are advantageous).

Starting from a binomial heap, two key enhancements appear in the Fibonacci heap:

- **“Lazy” Maintenance of the Root List.** In the binomial heap we are careful to maintain at most one tree of each rank. By contrast, the Fibonacci heap allows multiple trees of the same rank in the root list. This actually simplifies many operations; for example, we can \( \text{insert} \) a new element in \( O(1) \) time by linking a new rank-0 tree into the root list, and we can \( \text{merge} \) two heaps in \( O(1) \) time by linking their root lists together. The list of roots can grow quite long as a result. However, whenever we perform a \( \text{remove-min} \) operation, we do a bit of housekeeping and \( \text{consolidate} \) the list of root elements, linking together equal-rank trees until there are only \( O(\log n) \) root elements, at most one of each rank. This causes \( \text{remove-min} \) to run in \( O(R + \log n) \) time, where \( R \) is initial length of the root list. To amortize this properly, we charge only \( O(\log n) \) running time to the \( \text{remove-min} \) operation, and we charge \( O(1) \) additional time to any operation (e.g., \( \text{insert} \)) that creates a new root, so that each root element always maintains a credit of \( O(1) \) units that can be spent to pay for future consolidations. \( \text{Merge} \) only takes \( O(1) \) worst-case time, since it simply links two root lists together without drawing any credit from them. [Further details of the consolidation process].
Figure 5.7: Running times for the operations of several mergeable heaps. The class of “null path” mergeable binary heaps from Sections 5.4.1 and 5.4.2 includes randomized mergeable heaps (whose running time bounds hold with high probability), skew heaps (whose running time bounds are amortized), and also leftist heaps, “size-augmented” heaps, and “null-path-length-augmented” heaps. The running time bounds of these last three are deterministic as long as they don’t need to support delete, decrease-key, or increase-key, and otherwise they are amortized.

• **“Lazy” Maintenance of Tree Shapes.** The trees in a binomial heap are maintained in very strict shapes (binomial trees). For the Fibonacci heap, we relax this constraint slightly and allow up to one child to be deleted from every non-root element. Whenever we delete the child of a non-root element, we remember this fact by marking the element. If we attempt to delete a child from a marked element $e$ (i.e., if we attempt to delete a second child from $e$), we first detach $e$ from its parent and then insert $e$ into the list of roots. This forces us to mark $e$’s former parent, unless it was also previously marked, in which case we also detach it from its parent, and so on. The resulting series of cuts may potentially cascade up the tree and is known as a **cascading cut**. Root elements are somewhat special: they may lose more than one child to deletion, and they are never marked. Any marked element that is cut from its parent and made a root becomes unmarked in the process.

The machinery above for marking and cutting may seem a bit mysterious at first, but it gives us two important properties. First, it allows us to perform decrease-key in $O(1)$ amortized time. To decrease $e$’s key, we detach $e$ from its parent, decrease $e$’s key, and re-insert $e$’s subtree it into the list of roots. The act of detaching $e$ from its parent could initiate a cascading cut, but it turns out that large cascading cuts happen infrequently. This is another excellent example of the recursive slowdown principle (Section 4.5), since it takes two deletions from an element to cause a cut to propagate to its parent, thereby slowing the rate of a cascading cut by $1/2$ every step up the tree. Amortized analysis of decrease-key is actually quite simple. We associate one unit of credit with every marked node, allowing it to pay for its part.
5.5. INTEGER PRIORITY QUEUES

Figure 5.8: Illustrations of (a) a simple two-list priority queue data structure of elements whose keys only take the values zero and one, and (b) the generalization of this structure to a radix tree. Each leaf node in the radix tree contains a single element, whose key is written above in binary.

in any cascading cut. The only place decrease-key needs to actually pay is at the very end of the cut, where it finally reaches an unmarked element e. Here, it pays 1 unit to delete a child from e and mark e, and then 1 more unit to install the necessary credit on e.

The second important property afforded by the cutting machinery above is the following: in a binomial heap, a tree of rank \( j \) contains exactly \( 2^j \) elements. This fact was important to guarantee that there were \( O(\log n) \) subtrees in the heap. For the case of the Fibonacci heap, it is no longer the case that a rank \( j \) tree contains exactly \( 2^j \) elements, since decrease-key can detach subtrees within a tree. This is slightly worrisome, since the amortized \( O(\log n) \) running time of remove-min depends on the fact that only \( O(\log n) \) trees remain after consolidating the root list. Fortunately, since we are careful to delete at most one child from every non-root element, we can show that a tree of rank \( j \) must contain at least \( F_{j+2} + 2 \) elements, where \( F_j \) denotes the \( j^{th} \) Fibonacci number [Proof]. As one might guess, this fact explains how Fibonacci heaps get their name. Since \( F_j \) is exponentially large in \( j \) (one can show that \( F_{j+2} \geq \phi^j \), where \( \phi \approx 1.618 \) is the so-called “golden ratio”), a tree of rank larger than \( \log \phi n \) cannot exist in an \( n \)-element Fibonacci heap, so our root list will still contain only \( O(\log n) \) elements after consolidation, as desired.

5.5 Integer Priority Queues

Having described quite a few comparison-based priority queues, we now switch gears and focus on “input-sensitive” data structures that operate in the RAM model of computation and assume keys are integers in the range \( 0 \ldots C - 1 \). If \( C \) is suitably small with respect to \( n \), these data structures can potentially outperform their comparison-based counterparts. To motivate this, consider the extremely simple case where \( C = 2 \), in which keys have values of zero or one. In this case, we can easily implement every priority queue operation in \( O(1) \) time as shown in Figure 5.8(a) by maintaining a pair of doubly-linked lists: one for the elements whose keys have value zero, and the other for elements whose keys have value one.
5.5.1 The Radix Tree

The two-list data structure for \( C = 2 \) shown in Figure 5.8(a) can be generalized via recursion to a much more versatile and powerful structure called a \textit{radix tree}, shown in Figure 5.8(b). A radix tree is a binary tree whose left subtree is a radix tree recursively constructed from all keys having a most significant bit of zero, and whose right subtree is a radix tree constructed from keys having most significant bit of one. At the next level, the tree branches on the 2\textsuperscript{nd} most significant bit, and so on down the tree. Elements are stored in leaf nodes, and the root-to-leaf path for each element corresponds precisely to its binary representation — zero for a step to the left, one for a step to the right. Every key is represented by \( \log C \) bits\(^4\), so the tree has height \( \log C \). Since we do not store empty subtrees, a radix tree on \( n \) elements requires only \( O(n \log C) \) space, because we store \( \log C \) nodes along the root-to-leaf path for each element.

The radix tree is actually a very general data structure that has many uses beyond serving as a good priority queue, which we discuss in Chapter 7. Used as a priority queue, it supports all fundamental operations in \( O(\log C) \) time, all relatively easy to implement. For example, to \textit{insert} a new element with key \( k \), we walk downward from the root according to the binary representation of \( k \), adding nodes when necessary until we finally deposit the new element as a leaf. To remove the minimum, we first locate the leaf containing this element by walking down from the root (always moving left when possible), and after removing it we walk back up the tree cleaning up any subtrees that become empty as a result.

5.5.2 Multilevel Buckets: Another Route to the Radix Tree

Another way to generalize our trivial two-list structure for \( C = 2 \) to higher values of \( C \) is to build it out “horizontally”, as shown in Figure 5.9(a), giving an array \( A[0...C - 1] \) of \( C \) “buckets”, where \( A[k] \) points to a list of all elements with key \( k \). Here, all standard priority queue operations take \( O(1) \) time except \textit{remove-min}, which takes \( \Theta(C) \) worst-case time since it requires scanning through \( A \) to find the first non-empty bucket.

For large values of \( C \), either the \( \Theta(C) \) worst-case running time of \textit{remove-min} or the \( \Theta(C) \) space required to store the structure may be prohibitively large. To remedy this, we again generalize our structure in a hierarchical fashion, arriving at what is sometimes called a \textit{multilevel bucket} data structure. For example, by taking \( C = 100 \) with \( k = 2 \) levels of hierarchy, we get the structure shown in Figure 5.9(b). The top level contains \( \sqrt{C} \) “super buckets”, each corresponding to a range of \( \sqrt{C} \) different key values. A non-empty top-level bucket points to a second-level array of \( \sqrt{C} \) buckets, each containing a list of elements with a specific key.

We can easily extend such a data structure from a tree of height 2 to one of height \( k \), where each tree node is an array of buckets of length \( B = C^{1/k} \). If we take \( k = \log C \), we get a structure with \( \log C \) levels where every bucket array has length \( B = C^{1/\log C} = 2 \). This structure should look familiar, as it is precisely the radix tree! In fact, the multilevel bucket data structure is nothing more than a \( B \)-ary radix tree, where we write our keys in base \( B \), and at every node we branch in one

\(^4\)We usually assume here that \( C \) is a power of two, making \( \log C \) an integer.
of $B$ ways based on the value of a particular digit. For the tree in Figure 5.9(b), we write our keys in base 10.

Suppose our the tree has height $k$, so each node has a “width” of $B = C^{1/k}$ buckets. Since insert, delete, decrease-key, and increase-key only require walking vertically through the tree, they take $O(k)$ time. This is constant for the 2-level structure in Figure 5.9(b), but $O(\log C)$ in a radix tree. By contrast, remove-min requires scanning horizontally through each node (to locate the first non-empty bucket) as it walks down the tree, so its running time is $O(kB) = O(kC^{1/k})$. For the 2-level structure in Figure 5.9(b), this is $O(\sqrt{C})$; for the radix tree, it is $O(\log C)$. As you can see, the radix tree balances the cost of all operations at $O(\log C)$.

5.5.3 Monotone Integer Priority Queues and the Radix Heap

We can often gain efficiency in the special case of a monotone integer priority queue, where it is guaranteed that the sequence of minimum elements removed from the queue will be monotonically nondecreasing. That is, if $k$ denotes the key of the most-recently-removed minimum element, then we promise never to insert an element with key less than $k$, and we also promise never to decrease the key of an existing element to a value less than $k$. Such monotonic behavior arises in many important applications; for example, Dijkstra’s shortest path algorithm uses a monotone priority queue to processes the nodes of a graph in nondecreasing order of distance from a given source node.

For many applications (again, Dijkstra’s algorithm is a good example), the number of calls to decrease-key is expected to be significantly higher than the number of calls to insert and remove-min. This was one of the main motivations for developing the Fibonacci heap, since it allows for decrease-key to run in $O(1)$ amortized time. It turns out that monotonicity allows the radix tree / multilevel bucket data structure to perform decrease-key also in $O(1)$ amortized time, giving a data structure often called a radix heap.

The radix heap is a “lazy” radix tree. Monotonicity tells us that elements will be removed from the left subtree of the root for a while, but if remove-min ever hap-
## 5.6 Additional Problems

We have seen quite a few priority queue data structures throughout this chapter, and in the next two chapters we will see even more data structures that can function as priority queues. In particular, all of the comparison-based search structures in the next chapter can serve as a priority queue (with \(O(\log n)\) performance per operation), and in Chapter 7 we will revisit the radix tree and also develop highly-efficient integer search structures such as the Y-fast tree and the van Emde Boas structure that can perform priority queue operations in only \(O(\log \log C)\) time (depending on implementation, this bound may be amortized or in expectation).
5.6. ADDITIONAL PROBLEMS

Problem 83 (Incremental Priority Queues). An incremental priority queue supports three operations: insert(e), remove-max, and increase-priority(e). Elements entering the structure start out with priority zero, and this is incremented every time increase-priority is called. Remove-max removes and returns an element having maximum priority. This structure is useful for several applications, such as computing maximum adjacency orderings in graphs (Chapter ??). Please describe how to build an incremental priority queue in which all operations take $O(1)$ time (amortized is acceptable, but worst-case is preferred). [Solution]

Problem 84 (Range-Bounded Priority Queues). An integer priority queue is range-bounded if the total range of key values stored within the queue never exceeds some small number $R$ (presumably much smaller than $C$). Give a universal technique that allows us to take any integer priority queue whose running time depends on $C$ and adapt it for the range-bounded case, replacing all instances of $C$ in the running time with $R$ in the process. [Solution]

Problem 85 (Min-Max Heaps). If we want to maintain a set of $n$ elements so that both the minimum and the maximum can be located and removed efficiently, one possibility is to maintain two separate binary heaps, one of them an “min” heap and the other a “max” heap. However, another clever solution is to use only a single binary heap that satisfies a modified version of the heap property: at even-depth elements, we satisfy a “max” heap property (i.e., every element is the largest in its subtree) while at odd-depth elements, we satisfy a “min” heap property (i.e., every element is the largest in its subtree). Show how to modify the operations of a binary heap so they work in this extended setting and still require only $O(\log n)$ time. [Solution]

Problem 86 (The Offline Priority Queue Problem). Due to the comparison-based sorting lower bound, either insert or remove-min must run in $\Omega(\log n)$ worst-case time in the comparison model. This is true even in the “offline” case where we are told the sequence of insert and remove-min operations in advance. However, suppose we have a set of keys whose sorted ordering is already known. For example, let us take the set of values $S = \{1, 2, 3, \ldots, n\}$. Consider now the offline priority queue problem where we are told in advance a sequence of insert and remove-min operations, where the elements being inserted come from $S$. Each value in $S$ is inserted at most once, so there are no more than $2n$ total operations. For example, if we are given the input “I4 I1 I5 R R I3 R I2 R” (where I means we insert $x$, and R means we call remove-min, then the correct output should be “I 4 3 2 5”). Design an algorithm that computes the answers to all the remove-min queries in only $O(na(n))$ time. As a hint, the target running time of this algorithm suggests a particular data structure you should use. [Solution]

Problem 87 (Multi-Dimensional Monotonic Array Priority Queues). In problem 56, we investigated the time required to find an element in a multi-dimensional array that is “monotonic” in the sense that its elements are sorted along each dimension. In the two-dimensional case, this looks like a $\sqrt{n} \times \sqrt{n}$ array whose rows and columns are in sorted order. In the three-dimensional case, it becomes an $n^{1/3} \times n^{1/3} \times n^{1/3}$ array whose elements are monotonically increasing if we move forward along any single dimension (e.g., $A[i, j + 1, k] \geq A[i, j, k]$). Using the observation that one can “sift” effectively (in a similar fashion to the sift operations in a binary heap) in one of these structures if an element is modified, please discuss how to build a priority queue whose underlying implementation is a monotonic multi-dimensional array. Please describe the running time of all major priority queue operations (insert, remove-min, decrease-key, increase-key, and delete) in terms of $n$ as well as the dimensionality, $d$, and comment on what should be the optimal choice for $d$. [Solution]

Problem 88 (Median Filtering with Binary Heaps). An $n$-element array of noisy data can be smoothed using a median filter, by sliding a length-$k$ window over
the sequence, outputting at each location the median element within the window. There are several nice ways to implement a median filter in $O(n \log k)$ time (the best possible in the comparison model). For example, we can store the contents of the window in a balanced binary search tree (discussed in the next chapter), which supports the operations insert, delete, and select (for finding the median) all in $O(\log k)$ time. Since a balanced binary search tree can be complicated to implement, however, please discuss how you might implement a median filter in $O(n \log k)$ time using a pair of binary heaps instead.

In general, show how to build a data structure out of two binary heaps capable of tracking the $r$th order statistic of a dynamic data set (with $r$ known in advance and unchanging over the lifetime of the structure) such that insertion or deletion takes only $O(\log n)$ time.

[Solution]

Problem 89 (Level-Order Encoding of an Arbitrary Binary Tree). We have emphasized earlier that one of the benefits of a binary heap — being shaped like an almost-complete binary tree — is that one can represent the heap in memory using nothing more than a simple array. Moreover, we can step around within the array as if we were moving in the tree, since the left and right children of the node at index $i$ are located at indices $2i$ and $2i + 1$. In this problem, we show that a generalization of this mapping allows us to represent any static binary tree within an array, also allowing for easy tree-based movement. As shown in Figure 5.11, we map an $n$-node tree into an array $A[1 \ldots 2n + 1]$ in a level-by-level fashion, storing the nodes of the tree as well as a set of dummy “external” nodes representing null spaces at the bottom of the tree (these are stored as zeros in the array). Note that the almost-complete binary heap is just a special case of this mapping, where all the external nodes lie at the end of the array. Movement from parent to child generalizes in a pleasantly simple way: please prove that the children of the node at index $i$ live at indices $2A[i]$ and $2A[i] + 1$. We will revisit this mapping later in problem 136, when we use it to develop a “succinct” data structure for encoding a static, rooted tree. [Solution]
5.6. ADDITIONAL PROBLEMS

Problem 90 (Priority Queues Based on Braun Trees). A Braun tree (named after one of the first researchers to investigate the structure) is a binary tree that is “perfectly balanced” in the sense that for every node, its left subtree is either the same size or one element larger than its right subtree (and as a consequence, the tree has $O(\log n)$ height — see also problem 98 and Section 6.2.6 to see examples of these types of structures used in the context of balancing a binary search tree). To store a Braun tree in memory, each node resides in its own block of memory and maintains pointers to its parent, left child, and right child. In our present application, however, each node does not need to maintain information about the size of its subtree. In this problem, we show how Braun trees give us an elegant means of building priority queues, which happen to be particularly well-suited for functional programming environments.

(a) Given a pointer to the root of a Braun tree satisfying the heap property, show how to insert a new element in $O(\log n)$ time while maintaining the heap property. As a hint, you may want to draw inspiration from the way nodes in a skew heap swap their children. [Solution]

(b) Given a pointer to the root of a Braun tree satisfying the heap property, show how to remove the root (the minimum element) in $O(\log^2 n)$ time. Similarly, given a pointer to an arbitrary element, show how to delete that element in $O(\log^2 n)$ time, thereby enabling support for operations like decrease-key and increase-key. All of your operations should preserve the heap property. [Solution]

(c) For a challenge, show how to compute the size of a Braun tree in $O(\log^2 n)$ time, given only a pointer to its root. [Solution]

Problem 91 (A “Binomial Heap” Comprised of Sorted Arrays). At the beginning of this chapter, we briefly discussed the use of a single sorted array as a simple but inefficient means of implementing a priority queue. In this problem, we consider a generalization of this idea that uses several sorted arrays (or linked lists, if you think this is preferable) of different lengths. Each priority queue element will be stored in one of these arrays. Let us require that for each array in our collection (say it has length $L$), the next-largest array must have length at least $2L$. Since this property ensures that we will have at most $O(\log n)$ different sorted arrays in our data structure, find-min should take $O(\log n)$ time. To ensure that this property is maintained, we augment each array with its length, and we store these arrays in a linked list in order of their lengths. To insert an element $e$ into the structure, we insert a new length-1 array containing only $e$. If this happens to violate the property above (i.e., if there already exists a length-1 array), we repeatedly merge the array containing the new element with the next-largest array until the property finally becomes satisfied (using the same linear-time merge operation as in merge sort). To perform remove-min, we first locate the minimum element at the end of one of our arrays and then remove it by shortening the array. Again, this might violate the property above if our array shrinks to less than twice the size of the next-smallest array, so in this case we again perform successive merges until the property is restored. Both insert and remove-min have $\Theta(n)$ worst-case running times. However, for a challenge, can you prove that both insert and remove-min run in only $O(\log n)$ amortized time? Is it possible to implement the remaining priority queue operations decrease-key, increase-key, and delete in $O(\log n)$ amortized time as well? [Solution]

Problem 92 (Combining Disjoint Sets and Mergeable Priority Queues). Mergeable priority queues maintain a collection of priority queues on disjoint sets of elements, so that pairs of priority queues can be merged together efficiently. This should sound very much like the disjoint set problem (Section 4.6), where we maintain disjoint sets of elements that can be quickly unioned together. The only disjoint set operation that is not commonly offered by mergeable priority queues is the find operation, which reports the identifier of the set (in this case, the priority queue) to which a specific element belongs.
In this problem, we show how to take any fast priority queue (not even one supporting \textit{merge}) and make it mergeable, while also supporting the disjoint set \textit{find} operation. Suppose we take the highly efficient tree-based data structure for disjoint sets from Section 4.6 (supporting \textit{union} and \textit{find} in $O(\alpha(n))$ amortized time) and modify it slightly. In each tree, we will store elements only at leaves, and each non-leaf (internal) node will maintain a priority queue. Please determine what we should store in these priority queues so that if our original priority queue operations run in $O(T)$ amortized time, the operations \textit{insert}, \textit{remove-min}, \textit{find}, and \textit{merge/union} all now take $O(T\alpha(n))$ amortized time in this hybrid data structure. \[\text{Solution}\]
If you are stranded on a desert island with only one data structure, this chapter will help convince you that the data structure of choice should be a binary search tree. Binary search trees and their relatives skip lists and B-trees (also discussed in this chapter) are extremely powerful and versatile structures. They are used for countless applications in practice, and they give us efficient ways to implement sequences and dictionaries, perhaps the most fundamental data structure types.

**Sequences.** An element in a dynamic sequence is inserted, deleted, or accessed according to its numeric index within the sequence. As we showed in Section 1.5, arrays and linked lists are the simplest data structures we have available for representing a sequence, but both have serious drawbacks: arrays take $\Theta(n)$ worst-case time to insert or delete elements in the middle of a sequence, and linked lists take $\Theta(n)$ worst-case time to scan to an element given its index. In this chapter, we will see how binary search trees and their relatives allow us to **access**, **modify**, **insert**, and **delete** elements anywhere in a sequence in only $O(\log n)$ time.

**Dictionaries.** In a dictionary, each record of data has an associated key that is used to access the record. For example, a set of student records might be keyed on name, allowing us to quickly look up the record for a student given their name. A dictionary might consist of nothing more than a set of individual elements (e.g., a literal dictionary, containing text strings). It can also serve as a fast indexing structure built on top of an existing database. For example, in our database of student records, we might build separate dictionary structures keyed on name and student ID, allowing fast lookups based on either field\(^1\). Dictionaries support the fundamental operations **insert**, **delete**, and **find**, where **find**(k) locates an element with key k. Binary search trees and the other structures we discuss in this chapter suppose these operations in $O(\log n)$ time.

**Sets and Maps.** A dictionary corresponds to notion of an abstract mathematical set, since it represents a set of keys and supports efficient set membership testing.

---
\(^1\)As with many other data structures, we often don’t store large multi-field data records directly inside a dictionary itself. Rather, each element in the dictionary contains a key and a pointer to the larger data record it represents. This involves less memory shuffling, and eliminates redundancy when we build multiple dictionaries to index the same larger database on different fields.
1. set<int> S; // S is a set of integers
2. int x;
(a) 3. while (cin >> x) {
   4. if (S.count(x) > 0) cout << "Value " << x << " duplicated.\n";
   5. else S.insert(x);
}
(b) 1. map<string, int> C; // C is a map from strings to integers
2. string s, most_freq = "";
3. while (cin >> s) {
   4. C[s] = C[s] + 1; // C[s] counts the occurrences of string s
   5. if (C[s] > C[most_freq]) most_freq = s;
}
6. cout << most_freq << " appears " << C[most_freq] << " times.\n";

Figure 6.1: Examples of sets and maps using the C++ Standard Template Library (which usually implements them using binary search trees under the hood): in (a), we check for duplicate integers appearing in the input, thereby solving the element uniqueness problem, and (b) determines a string appearing most frequently in our input. For those not accustomed to C++, cin and cout are used for input and output, and S.count(x) returns 1 if x is in S, 0 otherwise.

You will often hear dictionaries called sets; the two terms are more or less synonymous. By augmenting each element in a dictionary with an associated value, we get another common type of data structure called a map, which stores a collection of (key, value) pairs. Many programming languages provide built-in support for maps using array notation, where A[k] refers to the value associated with key k. This allows us to pretend that we are indexing an “array” using strings or other complicated objects (sometimes this is known as an associative array). For example:

grade["student name"] = 95.

Due to its simple array-like appearance, novice programmers often fail to appreciate that underneath the hood, this statement is doing something very different from a standard array access — it is looking up the element with key “student name” in the dictionary named “grade”, and assigning its associated value to 95. This mechanism also conveniently gives us the ability to emulate a large sparsely-filled array using memory proportional only to the number of elements actually stored in the array. For example, if we start with an empty structure and say

is_prime[2305843009213693951] = true,

this adds a single element to our structure, with key 2305843009213693951 and value true, taking dramatically less space than an actual physical array of length 2305843009213693951. For all practical purposes, however, we can still use this structure as if it were an actual array. Since binary search trees and their relatives support find in $O(\log n)$ time, accessing an element is only slightly slower than the $O(1)$ time you would get from a genuine array (and in the next chapter, we will see how to achieve constant access time using hash tables).

Additional prototypical examples of sets and maps in code are shown in Figure 6.1.
Although a basic dictionary only needs to support `insert`, `delete`, and `find`, all of the structures in this chapter provide far more versatility, supporting many other useful operations:

- **Inexact search.** If `find(k)` fails to find a key of value `k`, we can still find elements in our dictionary with nearby keys.

- **Priority queue functionality.** These structures support `find-min` and `find-max`, and hence can also be used as priority queues (e.g., supporting `remove-min` by `find-min` plus `delete`).

- **Selection (rank-based access).** We can retrieve the element of any rank `r`. The special cases `r = 1` and `r = n` give the minimum and maximum, and `r = n/2` gives the median.

- **Range queries.** We can quickly count or enumerate in sorted order the elements in any given range `[a, b]`.

- **Sorting.** We can output the contents of our dictionary in sorted order.

Of course, if all you need is a basic dictionary, these structures still work fine, although you may also want to consider using a hash table, the subject of the next chapter. Hash tables are extremely fast RAM dictionary structures (requiring integer keys) that support the basic `insert`, `delete`, and `find` operations in only constant time, but that lack much of the added versatility of binary search trees and their relatives.

### 6.1 The Binary Search Tree

As shown in Figure 6.2, a binary search tree (BST) is a binary tree whose elements satisfy the **binary search tree property**: for any element `e` in the tree (say, having key `k`), elements in `e`’s left subtree have keys less than `k`, and elements in `e`’s right subtree have keys greater than `k`. The BST property and the heap property (from
the last chapter) are the two most common structural properties around which tree data structures are designed. We can think of the BST property as a “horizontal” property, with elements ordered from left to right, whereas the heap property is a “vertical” property, with elements ordered from top to bottom. To allow easy navigation, each element \( e \) in a BST typically maintains its key, \( \text{key}(e) \), a parent pointer, \( \text{parent}(e) \), and pointers \( \text{left}(e) \) and \( \text{right}(e) \) to its left and right children.

The BST property allows us to implement most BST operations in a straightforward recursive fashion. For example, in a BST with root element \( r \):

- To \textit{find} an element with key \( k \), the root is the answer if \( k = \text{key}(r) \). Otherwise, we recursively call \textit{find}(\( k \)) on the left subtree of the root if \( k < \text{key}(r) \), or on the right subtree of the root if \( k > \text{key}(r) \). This explains how the binary search tree gets its name, since the recursive process of finding a key is analogous to the process of binary searching in a sorted array.

- To \textit{insert} an element with key \( k \), we recursively insert into the left subtree of the root if \( k < \text{key}(r) \), or the right subtree of the root if \( k > \text{key}(r) \).

- To \textit{delete} an element \( e \), we replace \( e \) with the merged contents of \( e \)’s two subtrees using an operation called \textit{join} with a simple recursive implementation that we will discuss shortly. Here (as in many other data structures), we define \textit{delete} to take a pointer directly to an element of data, so we can separate its functionality from \textit{find}. If all we know is the key \( k \) of the element we want to delete, we would instead call \textit{delete}(\textit{find}(\( k \))).

These operations, as well as most other BST operations, run in \( O(h) \) time on a BST of height \( h \). This could be as bad as \( \Theta(n) \) if the BST is nothing more than a long path of depth \( n - 1 \) (and we can easily end up with a BST in this shape, say if we insert its elements in sorted order). However, we will soon learn how to restructure a BST during insertions and deletions so it stays \textit{balanced}. A balanced BST satisfies \( h = O(\log n) \), so all fundamental operations run in \( O(\log n) \) time. Note that a balanced tree does not need to be “perfectly” balanced, as, say, a binary heap, with every level completely filled in except the lowest. For now, we will continue quoting running times of \( O(h) \) for our BST operations, but bear in mind that these will all become \( O(\log n) \) once we learn how to maintain balance.

For simplicity, we assume that all keys in our BSTs (and other dictionary structures) are distinct. Duplicate keys are not particularly problematic, though. For example, we can maintain a single record in our structure for each distinct key value, which points to a linked list of all elements sharing that key. Sometimes we modify the BST property to specify where equal elements go, although this must be done with caution. For example, if we say that equal elements go in the left subtree, then a BST containing \( n \) equal elements would need to be shaped like a single left path, which is maximally unbalanced and hence very inefficient.

### 6.1.1 Traversals and Sorting

A \textit{traversal} is a walk through a tree that enumerates its elements in some well-specified ordering. A particularly common BST traversal is the \textit{inorder} traversal,
6.1. THE BINARY SEARCH TREE

Euler tour traversal: a b a c e c f c a d g d a

(a) Euler tour traversal: a b a c e c f c a d g d a

Inorder traversal: (5 + ((3 × 7) − (12 / 3)))
Preorder traversal: + 5 − × 3 7 / 12 3
Postorder traversal: 5 3 7 × 12 3 / − +

(b) Inorder traversal: (5 + ((3 × 7) − (12 / 3)))
Preorder traversal: + 5 − × 3 7 / 12 3
Postorder traversal: 5 3 7 × 12 3 / − +

Figure 6.3: Different tree traversals: (a) an Euler tour traversal, and (b) inorder, preorder, and postorder traversals of a binary tree that represents a mathematical expression; observe that these correspond to the infix, prefix, and postfix representations of the expression (so parsing an expression into a tree plus a traversal can be used to transform between these representations). For clarity, we have included parentheses in the inorder traversal.

which recursively traverses the left subtree of the root, then prints the root, then recursively traverses the right subtree of the root. This takes only $\Theta(n)$ time, and prints the contents of a BST in sorted order, as a consequence of the BST property.

The inorder traversal gives us yet another $O(n \log n)$ algorithm for sorting, sometimes called BST sort: first build a balanced BST on $n$ elements by inserting each one in $O(\log n)$ time, then enumerate its contents in sorted order with a $\Theta(n)$ inorder traversal. Since the BST is comparison-based (along with every other data structure in this chapter), the $\Omega(n \log n)$ worst-case lower bound for comparison-based sorting implies a worst-case lower bound of $\Omega(\log n)$ on the time required to insert an element into a BST.

It is useful to think of the inorder traversal of a BST as a sequence of $n$ elements “encoded by” the BST. Since we are currently using the BST to implement a dictionary, this sequence is the sorted ordering of the elements in the dictionary. We will revisit this viewpoint in a few pages when we learn how to encode an arbitrary sequence within the inorder traversal of a BST.

Traversals for Non-Binary Trees. As opposed to the inorder traversal, which only makes sense in binary trees, several other common types of traversals, shown in Figure 6.3, are used in both binary and non-binary trees:

- A preorder traversal prints the root, and then recursively visits the subtrees of the root from left to right. Preorder traversals are often used in tree computations that work downward from the root to the leaves.

- A postorder traversal recursively visits the subtrees of the root from left to right, then prints the root. Postorder traversals are often used in tree computations that work their way up from the leaves to the root.

- An Euler tour, or Eulerian traversal, walks “around the perimeter” of a tree as it prints the root, then the first subtree of the root, then the root again, then
the second subtree of the root, and so on, ending with the root. The traversal is so-named because it corresponds to an Eulerian cycle of the tree with its edges doubled (in fact, it is almost more of a traversal of the edges in a tree than its nodes). Each edge is followed twice (down and later up), so the Euler tour traversal of an \( n \)-node tree has length \( 2n - 1 \). We could also call this a \textit{depth-first} traversal, since it visits the same sequence of nodes as \textit{depth-first search}, a common graph traversal algorithm we will learn in Chapter ??.

All three of these traversals run in \( \Theta(n) \) time on a rooted \( n \)-node tree. We will see another non-binary generalization of the inorder traversal similar to the Euler tour traversal when we study \( B \)-trees later in the chapter.

\textbf{Problem 93 (Identifying Traversals).} Given a sequence of \( n \) elements, devise an algorithm that tests in \( \Theta(n) \) time whether this sequence corresponds to the inorder, preorder, postorder, or Euler tour traversal of some BST. You may assume the tree contains only distinct elements. [\textbf{Solution}]

\subsection*{6.1.2 Min and Max, and the BST as a Priority Queue}

To find the minimum element in a BST in \( O(h) \) time, we walk down the “left spine” of the tree, starting from the root and repeatedly following left child pointers until we reach an element with no left child. Similarly, we can compute the maximum element in a BST in \( O(h) \) time by walking down its right spine. In a balanced BST, this allows us to implement the operations \texttt{find-min} and \texttt{find-max} in \( O(\log n) \) time.

Since it supports \texttt{find-min} and \texttt{find-max}, the balanced BST can function as a priority queue whose asymptotic performance matches the binary heap. It can perform \texttt{insert} and \texttt{remove-min} (\texttt{find-min} followed by \texttt{delete}), as well as every other fundamental priority queue operation, in \( O(\log n) \) time. However, the balanced BST is probably not the data structure of choice if all we need is a priority queue, since the binary heap is far simpler to implement. Furthermore, the balanced BST cannot match the performance of the fancier priority queues discussed in the last chapter (e.g., it cannot perform \texttt{decrease-key} in \( O(1) \) amortized time like a Fibonacci heap).

\subsection*{6.1.3 Predecessor, Successor, and Inexact Search}

The predecessor of element \( e \), \( \texttt{pred}(e) \), is the element right before \( e \) in the inorder traversal (i.e., the next-smallest element in the BST). The successor of \( e \), \( \texttt{succ}(e) \), is the element right after \( e \) in the inorder traversal (i.e., the next-largest element in the BST). Given a pointer to \( e \), we can easily locate \( \texttt{pred}(e) \) and \( \texttt{succ}(e) \) in \( O(h) \) time, as explained in Figure 6.4.

In addition to \( \texttt{pred}(e) \) and \( \texttt{succ}(e) \), it is also easy to implement the related operations \( \texttt{pred}(k) \) and \( \texttt{succ}(k) \) that return the closest element before or after a given key value \( k \) (if \( k \) is present in the BST, they return the element with this key, just like \( \texttt{find} \)). Implementing \( \texttt{pred}(k) \) and \( \texttt{succ}(k) \) in \( O(h) \) time is quite simple: an unsuccessful call to \( \texttt{find}(k) \) will terminate on an element \( e \) that is either \( \texttt{pred}(k) \) or \( \texttt{succ}(k) \), after which a single call to either \( \texttt{pred}(e) \) or \( \texttt{succ}(e) \) can be used to locate the other of these two elements.
6.1. THE BINARY SEARCH TREE

Figure 6.4: To compute $\text{succ}(x) = y$, we either (a) take the minimum element in $x$’s right subtree, if $x$ has a right child, or otherwise (b) find the first “right parent” we encounter walking upward from $x$. Similarly, to compute $\text{pred}(y) = x$, we either (b) take the maximum element in $y$’s left subtree, if $y$ has a left child, or otherwise (a) find the first “left parent” we encounter walking upward from $y$. Observe that these cases are completely symmetric; for example, in (a), if $x$ is the first “left parent” of $y$, then $y$ is the minimum in $x$’s right subtree.

The operations above allow a BST to perform *inexact searches*, finding elements whose keys are close in value to a key we want. For example, when searching for a partial book title in a library catalog, we can find the closest matches using $\text{pred}(k)$ or $\text{succ}(k)$, then we can scroll through a list of nearby matches using repeated calls to $\text{pred}(e)$ or $\text{succ}(e)$, until we find the book we want. We can also answer a *range query* of the form “tell me all the elements in the range $[a,b]$ in sorted order” by first finding $e = \text{succ}(a)$, and then by repeatedly stepping from $e$ to $\text{succ}(e)$ until $\text{key}(e) > b$. As we see in the next problem, this actually runs quite fast.

**Problem 94 (Successive Successors).** $\text{pred}(e)$ and $\text{succ}(e)$ have good amortized performance. Please show that if we start at the minimum element in a BST, only $\Theta(n)$ time is required for $n - 1$ successive calls to $\text{succ}(e)$ (and that this essentially performs an inorder traversal of the tree). Furthermore, show that performing $k$ repeated calls to $\text{succ}(e)$ starting from any element in a BST (say, in response to a range query asking us to output all the elements in some interval $[a,b]$) requires only $O(h + k)$ time. This doesn’t quite fit our standard definition of amortized running time, but we could say repeated calls to $\text{succ}$ (or repeated calls to $\text{pred}$) run in $O(1)$ amortized time as long as we pay a one-time penalty of $O(h)$ up front. [Solution]

**Deletion After Swapping with a Predecessor or Successor.** Our preferred method for deleting an element in a BST is by replacing it with the *join* of its two subtrees (we discuss the join operation in a moment). However, another common method for deletion involves the use of $\text{pred}$ and $\text{succ}$. It is trivial to delete an element with no children (i.e., a leaf), and also easy to delete an element with only one child, since we can just replace the element with its child. To delete an element $e$ with two children, we note that both $\text{pred}(e)$ and $\text{succ}(e)$ can have at most one child, so we can first swap $e$ with one of these elements, and then delete $e$ from its new location, where it now has at most one child. Replacing $e$ with $\text{pred}(e)$ or $\text{succ}(e)$ is “safe”, since this does not create any violation of the BST property.
6.1.4 Augmenting a BST, Select, and Rank

The BST can be modified to support rank-based access, via the following operations:

- **Select**\( (r) \). Returns the element having the \( r \)th largest key (i.e., the element of rank \( r \), or equivalently the element at index \( r \) within an inorder traversal). The minimum element has rank \( r = 1 \), the maximum has rank \( r = n \), and the median has rank \( r = n/2 \).

- **Rank**\( (e) \). Returns the rank of element \( e \) — that is, the index of \( e \) within an inorder traversal (this is the “inverse” of the select operation).

To implement these both in \( O(h) \) time, we first need to augment each element \( e \) in our BST with its subtree size, \( size(e) \). We keep this information up to date by modifying insert and delete. When a new element \( e \) is inserted, we increment the subtree sizes along the path from the root down to \( e \) (as shown in Figure 6.5(a)). Deletion of \( e \) decrements these values. Note that these changes preserve the asymptotic \( O(h) \) running times of insert and delete.

The rank of \( e \) is the number of BST elements \( e' \) with \( key(e') \leq key(e) \). As shown in Figure 6.5(b), these elements \( e' \) include all the “left parents” of \( e \) along the path up to the root, as well as their left subtrees. To implement rank\( (e) \) in \( O(h) \) time, we count these while walking from \( e \) up to the root. The size of a subtree is counted quickly by inspecting its root, thanks to our augmented information.

To implement select\( (r) \) in \( O(h) \) time, we compute the rank \( R \) of the root by adding one to its left subtree size. If \( r = R \), the root is the answer. Otherwise, if \( r < R \), we recursively call select\( (r) \) in the left subtree of the root, and if \( r > R \) we recursively call select\( (r - R) \) in the right subtree of the root. This approach is directly analogous to the quickselect algorithm for selection from an array, so we can view the BST augmented with subtree sizes to support select and rank (sometimes known as an order statistic tree) as a dynamic version of quickselect.

Select and rank are just two of many extended operations we can build by carefully augmenting a BST. As another simple example, by augmenting each element
6.1. THE BINARY SEARCH TREE

with direct pointers to its predecessor, successor, and the minimum and maximum elements in its subtree, we get $O(1)$ worst-case implementations of $\text{find-min}(e)$, $\text{find-max}(e)$, $\text{pred}(e)$, and $\text{succ}(e)$ (note that we can maintain this extra information without compromising the $O(h)$ running time of $\text{insert}$ or $\text{delete}$).

**Problem 95 (Randomly Sampling from a BST).** If each element $e$ in a BST has an associated frequency count $f_e$, how can we augment our BST to support the ability to sample a random element $e$ with probability proportional to $f_e$ in $O(h)$ time, assuming we can generate a random number in $[0, 1]$ in $O(1)$ time? See also problem 114. [Solution]

**Problem 96 (Inversion Counting with a BST).** Please show how to count the number of inversions in a length-$n$ array in $O(n \log n)$ time using a BST (see also problems 61 and 122). For simplicity, please assume the BST is balanced, so every BST operation runs in $O(\log n)$ time (we will learn how to achieve balance shortly). [Solution]

### 6.1.5 Encoding a Sequence in a BST

The BST is quite effective as a dictionary data structure. In this section, we discuss the second major application of the BST: representing an arbitrary $n$-element sequence $A_1 \ldots A_n$. As shown in Figure 6.6, the natural modification of the “BST property” in this case ensures that the sequence corresponds to the inorder traversal of the tree. We store some element $A_i$ at the root, we store $A_1 \ldots A_{i-1}$ in the left subtree of the root, and we store $A_{i+1} \ldots A_n$ in the right subtree of the root.

Since our elements are not sorted from left to right, the $\text{find}$ operation is now meaningless and no longer used. Instead, we access elements in the sequence by index using $\text{select}$ and $\text{rank}$ (so it is necessary to augment the tree with subtree sizes). $\text{Select}(r)$ returns the element at index $r$ within the sequence, and $\text{rank}(e)$ returns the index of element $e$. We can also use $\text{pred}$ and $\text{succ}$ to step between consecutive sequence elements, and $\text{find-min}$ and $\text{find-max}$ to jump directly to the beginning or end of the sequence. Elements are inserted by rank (index) as well, using $\text{select}$ to find the proper location within the tree to attach a new element.

Recall from Section 1.5 that arrays and linked lists both have significant drawbacks when representing a dynamic sequence. The array can access elements by index in
only $O(1)$ time, but it takes potentially $\Theta(n)$ time to insert and delete elements in the middle of the sequence. The linked list has the opposite trade-off: $\Theta(n)$ time to access an element by index, but then $O(1)$ time to insert or delete. The balanced BST balances these two extremes, supporting all operations in $O(\log n)$ time.

Since we no longer search by key when we encode a sequence, it makes slightly less sense to call this usage of our data structure a “binary search tree”, and some would argue that “order statistic tree” is more appropriate in this context. However, we use the term “binary search tree” for both cases, since it really is the same data structure playing both roles, and all of our ensuing discussion (e.g., with maintaining balance) applies in both cases. In both roles, the BST is perhaps best viewed as a structure that fundamentally encodes a sequence from left to right within its inorder traversal. When used as a dictionary, this sequence is the sorted ordering of the elements in the dictionary, enabling key-based access as well as rank-based access.

### 6.1.6 Rotations

Soon, we will discuss methods to keep a BST balanced, by strategically moving elements during insert and delete via some mechanism that preserves the BST property. The most common such mechanism is a rotation about some edge in the tree. Rotations come in two symmetric flavors, left rotations and right rotations. As seen in Figure 6.7, the inorder traversal of a BST is unchanged after either direction of rotation. The reader can probably guess how these might be used to maintain balance. For example, if subtree $A$ in Figure 6.7 becomes too deep, a right rotation may improve the overall balance of the tree.

We occasionally use rotations to pull an element to the root of a tree (by repeatedly rotating with its parent), or down to a leaf (by repeatedly rotating with a child). For example, we could delete an element in $O(\log n)$ time with high probability by removing it after first re-locating it to a leaf by repeatedly rotating it with a random child — effectively following a random null path, as we did back in Section 5.4.1.

**Problem 97 (Rotation Distance).** The rotation distance between two $n$-element binary trees $T_1$ and $T_2$ is the minimum number of rotations required to transform $T_1$ so that it has the same “shape” as $T_2$. The question of how efficiently one can compute the exact rotation distance between two trees is actually an open problem. Here, see if you can devise an algorithm that performs $O(n)$ rotations to transform between any pair of $n$-element trees provided as input. [Solution]
6.1. THE BINARY SEARCH TREE

Problem 98 (Balancing a Tree with Rotations). A BST with \( n = 2^x - 1 \) elements can be restructured so as to be “perfectly” balanced, with every level filled in completely. We could do this in \( \Theta(n) \) time with divide and conquer: start with an inorder traversal, place the middle element at the root, then recursively build perfectly balanced subtrees on the left and right. Here, we develop a more “in place” approach, based on rotations. First, show how to transform any BST into a right path in \( \Theta(n) \) time. Next, show how to make multiple passes down this path while performing left rotations (each time roughly halving the path length), so that in \( \Theta(n) \) time we end with a perfectly-balanced BST. If you are feeling ambitious, show how this approach can be extended to BSTs of arbitrary size, so that \(|\text{size}(\text{left}(e)) - \text{size}(\text{right}(e))| \leq 1\) holds for every element \( e \) in the final balanced tree (these trees are similar to Braun trees — problem 90). [Solution]

6.1.7 Split and Join

Two elegant but often under-appreciated BST operations are **split** and **join**, which are defined as follows in the context of a dictionary:

- **Split** \( (e) \). Break a BST into two BSTs, one containing keys at most \( \text{key}(e) \), the other containing keys greater than \( \text{key}(e) \).

- **Join** \( (T_1, T_2) \). Join two BSTs \( T_1 \) and \( T_2 \) (with all keys in \( T_1 \) being less than those in \( T_2 \)) into a single BST.

**Split** is easy to implement in \( O(h) \) time by rotating \( e \) up to the root and detaching its right subtree. Equivalently, we can use recursion: detach \( e \)'s right subtree if \( e \) is the root, or otherwise recursively split the subtree containing \( e \). To join \( T_1 \) and \( T_2 \), we could rotate the maximum element of \( T_1 \) to its root and attach \( T_2 \) as its right child, or alternatively rotate the minimum of \( T_2 \) to its root and attach \( T_1 \) as its left child. One can also implement **join** recursively (the author’s preference), as shown in Figure 6.8(a), by taking \( T_1 \) and recursively joining it with the left subtree of \( T_2 \), or by taking \( T_2 \) and recursively joining it with the right subtree of \( T_1 \). Both approaches run in time bounded by the combined heights of \( T_1 \) and \( T_2 \).
**Insert** and **delete** can be easily built using **split** and **join**. To delete element \( e \), we replace \( e \) with the **join** of its two subtrees. To insert element \( e \) with key \( k \), we can split our tree on \( \text{pred}(k) \), then join the resulting two pieces back together as children of \( e \). Whereas the “standard” BST **insert** procedure inserts an element as a leaf, this method inserts a new element at the root.

**Cutting and Linking BSTs and Sequences.** When performing a range query for all elements in some range \([a, b]\), we can use two calls to **split** to obtain a BST with keys smaller than \( a \), a BST with keys in \([a, b]\), and a BST with keys larger than \( b \). By joining the first and last BSTs back together, we have effectively pulled the BST representing the answer to our query completely out of the original tree. If our trees represent sequences instead of dictionaries, this same approach allows us to efficiently **cut** a smaller sequence out of the original (shown in Figure 6.8(b)), after which we can **paste** it back in at any location with one call to **split** and two calls to **join**. We will build on this idea further when we introduce **dynamic tree** data structures in Section 8.5.

### 6.1.8 Maintaining Augmented Data

One issue to keep in mind when we perform rotations (or any other operation that modifies the structure of a BST, such as **insert**, **delete**, **split**, or **join**) is how to keep augmented information in the tree (e.g., subtree sizes) up to date. This is a general concern with data structure design — we often augment a structure to give it extra functionality, but on the other hand, we now need to spend extra work maintaining this augmented data whenever the structure is modified.

Most common types of BST augmentations are **locally** recomputable, where we can recompute the augmented information attached to \( e \) in \( O(1) \) time in response to a change in the augmented information of one of \( e \)'s children. For example, \( \text{size}(e) = 1 + \text{size}(\text{left}(e)) + \text{size}(\text{right}(e)) \) can be updated in \( O(1) \) time if \( \text{size}(\text{left}(e)) \) or \( \text{size}(\text{right}(e)) \) changes. As a general rule, any locally-recomputable augmentation can be easily maintained without affecting the asymptotic running time of any standard operation that changes the structure of a BST (e.g., **insert**, **delete**, **split**, or **join**)\(^2\). The intuition behind this is as follows: a modification originating at some element \( e \) could cause a chain of augmentation updates, each in \( O(1) \) time, propagating from \( e \) up to the root (say, along a path of length \( p \)). However, since the current operation is modifying \( e \) to begin with, it presumably has already walked down this path to reach \( e \), so it has already spent \( O(p) \) time.

### 6.2 Balanced Binary Search Trees

The key to good performance with a BST is maintaining balance, so \( h = O(\log n) \). Many balancing approaches appear in the literature and in practice, and we have chosen several these to present here, since each one highlights useful and elegant techniques for data structure design.

\(^2\)Moreover, this will remain true even when we study **B**-trees later in the chapter, even though nodes in a **B**-tree can have many more than two children.
Balancing mechanisms fall into three groups. We begin this section with worst-case balancing mechanisms, which maintain \( h = O(\log n) \) always and which provide \( O(\log n) \) worst-case guarantees for all standard BST operations. We then discuss randomized mechanisms, where balance is always maintained with high probability, and all operations therefore run in \( O(\log n) \) time with high probability. Finally, we discuss amortized mechanisms, where strict balance is not always maintained, but any sequence of \( k \) operations still takes \( O(k \log n) \) worst-case time.

### 6.2.1 Height-Balanced (AVL) Trees

A tree is height-balanced if for every element \( e \), the heights of \( e \)'s left and right subtrees differ by at most one. Height-balanced trees are also known as AVL trees after their inventors Adel’son-Vel’skiï and Landis. An easy induction proof shows that \( h = O(\log n) \) for any height-balanced tree. [Details]

To implement an AVL tree, we augment each element with the height of its subtree; this is a locally-recomputable augmentation, so it does not affect the asymptotic running time of insert or delete to maintain. Every time we insert or delete an element, we check to make sure the resulting tree remains height-balanced. If not, we can restore height balance in \( O(\log n) \) time by performing a small number of carefully chosen rotations. [Details]

### 6.2.2 Balancing Based on Subtree Size

Let \( \alpha \) be any constant in \( [1/2, 1) \). A tree is \( \alpha \)-balanced if for every element \( e \), both the left and right subtrees of \( e \) contain at most \( \alpha \cdot \text{size}(e) \) elements. Since our subtree size drops by a constant factor every step down the tree we take, it is easy to see that \( h = O(\log n) \) for any \( \alpha \)-balanced tree.

Balancing mechanisms based on this idea in the literature are known as “weight-balanced” trees or trees of “bounded balance” (sometimes called BB\( [\alpha] \) trees). As with the AVL tree, they restore balance after an insertion or deletion breaks the \( \alpha \)-balance property by performing a small number of rotations in \( O(\log n) \) time (for appropriate values of \( \alpha \)). The details are somewhat messy\(^3\), however, and are omitted from our present discussion. Instead, we will focus on the use of this property as part of a simple amortized balancing scheme described in a few pages.

### 6.2.3 Red-Black and Path-Balanced Trees

Perhaps due to their prominence in a number of well-known algorithms texts, red-black trees seem to be particularly popular in practice. We color every element in a BST either red or black such that two invariants are maintained: the number of black elements on every root-to-leaf path must be the same, and we cannot have an adjacent (parent-child) pair of red elements. A simple induction argument shows that this invariant implies that \( h = O(\log n) \). [Details]

\(^3\)The details are messy enough that the original journal paper describing this balancing mechanism contained subtle errors in its analysis.
After our invariant breaks due to an insertion or deletion, we can restore it in $O(\log n)$ time by rotations and by recoloring elements. We omit the somewhat tedious details of how this is done, since the red-black balancing mechanism turns out to be equivalent to the conceptually simpler mechanism used to balance a “2-3-4 tree”, which we cover later in the chapter.

A nice mathematical generalization of the red-black tree goes by the name of a path-balanced tree, where edges are assigned numeric weights such that all root-to-leaf paths have the same sum. For the specific case of a red-black tree, an edge leading down into a red node has weight zero and an edge leading into a black node has weight one. Rebalancing in this setting involves a mixture of edge reweightings and rotations in order to achieve some desired property of the weight sequences (e.g., no two adjacent zeros, for the red-black tree), although as with the red-black tree, the details are often somewhat cumbersome. At the end of this the chapter, however, we highlight a simple randomized mechanism (equivalent to a skip list, a close relative of the BST) that keeps a tree path-balanced, with all operations running in $O(\log n)$ time with high probability.

6.2.4 Randomly-Structured Binary Search Trees

When we studied quicksort, we alluded to its strong similarities with the BST, which essentially encodes the tree of all recursive subproblems generated by quicksort. As explained in Figure 6.9, the process of building a BST performs exactly the same comparisons as quicksort, just in a different order, allowing us to deduce properties of BSTs given what we already know about quicksort. For example, the recursion depth in quicksort corresponds to the height of its associated BST. This is particularly useful for randomly-structured BSTs — built from an initially-empty...
6.2. BALANCED BINARY SEARCH TREES

BST by inserting \( n \) elements in random order — since this process is analogous to randomized quicksort. Given this direct correspondence, we can now leverage our knowledge of randomized quicksort to conclude that the following all hold in expectation and with high probability:

<table>
<thead>
<tr>
<th>Randomized quicksort</th>
<th>Randomly-structured BSTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spends ( O(\log n) ) comparisons on any specific element</td>
<td>Depth of any specific element is ( O(\log n) )</td>
</tr>
<tr>
<td>Spends ( O(\log n) ) comparisons per element on every array element</td>
<td>Height of entire tree is ( O(\log n) )</td>
</tr>
<tr>
<td>( O(n \log n) ) total comparisons</td>
<td>( O(n \log n) ) total time to build</td>
</tr>
</tbody>
</table>

Unfortunately, even though a randomly-built BST starts out balanced, subsequent insertions and deletions (which are not random, since they are controlled by the user) may lead to imbalance over time. However, we can cleverly modify insert and delete to keep the tree always randomly structured, as if we had just built it randomly from scratch. This approach, which we call the randomized BST (RBST), yields a tree that always satisfies \( h = O(\log n) \) with high probability.

To insert an element \( e \) into an \( n \)-element tree while preserving its random structure, we flip a biased coin and with probability \( \frac{1}{n+1} \), we insert \( e \) at the root\(^4\), either using split as described in Section 6.1.7, or equivalently by inserting \( e \) at a leaf as in a standard unbalanced BST, then rotating it up to the root. Otherwise, we recursively insert \( e \) into either the left or right subtree of the root, based on its key or intended rank (this recursive call again flips a coin to determine if \( e \) should be inserted at the root of its subtree, and so on). [Proof that this keeps the tree randomly structured]

To delete element \( e \), we replace \( e \) by the result of joining its two subtrees, using a biased version of the recursive join operation (Figure 6.8(a)) that preserves random structure. It takes two randomly-structured trees \( T_1 \) and \( T_2 \) (with elements of \( T_1 \) preceding those in \( T_2 \)) and joins them into a single randomly-structured tree. Suppose \( n = n_1 + n_2 \), where \( n_1 \) and \( n_2 \) denote the number of elements in \( T_1 \) and \( T_2 \). With probability \( n_1/n \) we recursively join \( T_2 \) with the right subtree of \( T_1 \)'s root and otherwise (with probability \( n_2/n \)), we recursively join \( T_1 \) with the left subtree of \( T_2 \)'s root\(^5\). [Proof that this keeps the tree randomly structured]

6.2.5 Treaps

The treap is a curious hybrid between a binary search tree and a binary heap. As shown in Figure 6.10, each element in a treap contains two keys, a “BST” key and “heap” key. The BST keys satisfy the BST property, and the heap keys satisfy the heap property. If you ignore the heap keys, the structure therefore looks like a BST, and if you ignore the BST keys, the structure looks like a heap.

The treap gives us another simple randomized mechanism for maintaining balance in a BST by keeping it in a randomly-structured state. It is actually equivalent to the RBST mechanism we just described, even though it may look quite different (in the

\(^4\)This seems reasonable since if we had built the resulting \((n + 1)\)-element tree randomly from scratch, there would be a \( \frac{1}{n+1} \) chance of \( e \) being inserted first, and hence located at the root.

\(^5\)These bias factors seem reasonable, since we want every element in the joined tree to have an equal probability of being its root. If \( T_1 \) has twice as many elements as \( T_2 \), it should therefore be twice as likely that \( T_1 \)'s root (i.e., a random element in \( T_1 \)) stays at the root of the joined tree.
In the literature, treaps were proposed several years before RBSTs. If we take a BST and assign each element a random distinct heap key, then there is only one valid “shape” the resulting treap can assume, and this is the shape of a randomly-structured tree. The element with minimum heap key (effectively a random element) must reside at the root, and then the BST property dictates which elements must belong to the left and right subtrees. The roots of these subtrees are again uniquely determined by the heap property, and so on. The resulting tree is the same we would get by inserting all elements into a BST in order by their heap keys, which is random order. Therefore, a treap with randomly-chosen distinct heap keys is balanced with high probability. The same is true even if heap keys are not necessarily distinct, as long as they are randomly chosen from a sufficiently large range.

Insertion and deletion in a treap requires maintaining both the BST and heap properties. To insert a new element \( e \), we assign it a random heap key and then use the standard BST insertion procedure to insert \( e \) as a leaf in our tree, ignoring heap keys for the moment. This satisfies the BST property, but we may now violate the heap property between \( e \) and \( \text{parent}(e) \). To fix this, we use the standard sift-up(\( e \)) heap operation (Section 5.3.2), which we now implement using rotations so it maintains the BST property. Deletion of an element \( e \) can be done several ways; perhaps the cleanest is to replace \( e \) with the join of its two subtrees \( T_1 \) and \( T_2 \), where now the usual choice between “recursively merge \( T_1 \) into the left subtree of \( T_2 \)” or “recursively merge \( T_2 \) into the right subtree of \( T_1 \)” (Figure 6.8(a)) is determined by whichever of \( T_1 \) and \( T_2 \) has the smaller heap key at its root. Alternatively, we could remove \( e \) trivially if it is a leaf or has only one child, or otherwise, just as in a standard BST, we swap \( e \) with \( \text{pred}(e) \) or \( \text{succ}(e) \) and then remove \( e \) from the tree. We then repair violations of the heap property (since we have replaced \( e \) with an element having a different heap key) by calling sift-up or sift-down, which are again implemented using rotations so they preserve the BST property.

A BST can be used to encode a dictionary or a sequence. Accordingly, we can also encode a sequence in the “BST” part of a treap. In this context, we often use the elements of the sequence themselves as the heap keys (rather than a separate set of keys), leading to a data structure known as a Cartesian tree that we will study in Chapter 8. In the literature, one sometimes finds the term “Cartesian tree” used as a synonym of “treap”. In this book, however, we treat the Cartesian tree as a relative of treap where the “BST” part of the structure is used to encode a sequence, rather than a dictionary. Although they are closely related to treaps, Cartesian
6.2. BALANCED BINARY SEARCH TREES

trees look somewhat different since their elements each have only a single key, and since they are typically not balanced. In addition to Cartesian trees, Chapter 8 introduces yet another close relative of the treap, known as a **priority search tree**, which is used for encoding a set of points in the 2D plane, where x coordinate plays the role of a BST key and y coordinate serves as a heap key.

**Problem 99 (Searching for Nearby Elements).** In some variants of BSTs, it is faster to walk to an element \( e \) from a previously-located element \( e' \) close to \( e \) in rank space, than to search for \( e \) starting from scratch at the root. Suppose we walk from \( e' \) to \( e \) in a treap (or an RBST, being equivalent in structure). Please show that the path we follow has expected length \( O(\log \Delta) \), where \( \Delta = |\text{rank}(e) - \text{rank}(e')| + 1 \). As a hint, use linearity of expectation, in much the same way we analyzed the expected running time of randomized quicksort by counting expected comparisons between elements. [Solution]

**Problem 100 (Writes Versus Reads).** Please argue that the process of building a treap on \( n \) elements involves only \( \Theta(n) \) expected individual memory writes (versus \( \Theta(n \log n) \) expected memory reads). This can be advantageous since writes are often less efficient than reads due to caching overhead. Note that this is equivalent to showing \( \Theta(n) \) expected rotations, so you may want to proceed by linearity of expectation, letting \( X_{ij} \) be an indicator random variable taking the value 1 if the element of rank \( i \) is rotated upward to unseat the element of rank \( j \). If you can show that \( E[X_{ij}] = \Theta(1/r^2) \), with \( r = |i-j|+1 \), then \( E[\sum X_{ij}] \) will be \( \Theta(n) \), as desired. Why does the result of this problem not hold if we use \( n \) calls to *insert* in an RBST instead of a treap? [Solution]

### 6.2.6 Amortized Rebalancing and Scapegoat Trees

In addition to \( \text{size}(e) \), let us attach an “imbalance counter”, \( \text{count}(e) \), to each element \( e \), giving an upper bound on \( |\text{size}(\text{left}(e)) - \text{size}(\text{right}(e))| \). We increment \( \text{count}(e) \) any time an insertion or deletion occurs in \( e \)'s subtree, since any such operation can at worst increase this imbalance count by one. Specifically, when we *insert* or *delete* an element, we walk from the root down to the point of insertion/deletion, doing the following for each element \( e \) along the way:

1. Increment (for *insert*) or decrement (for *delete*) the value of \( \text{size}(e) \).
2. Increment \( \text{count}(e) \).
3. If \( \text{count}(e) > \lceil \frac{1}{3} \text{size}(e) \rceil \), we stop the insertion/deletion process and rebuild \( e \)'s subtree from scratch (taking into account the inserted or deleted element) in \( \Theta(\text{size}(e)) \) time so it is “perfectly” balanced, say, as discussed in problem 98. During the process, we reset \( \text{count}(e') \) to \( |\text{size}(\text{left}(e')) - \text{size}(\text{right}(e'))| \) for every element \( e' \) in \( e \)'s subtree. Note that this ensures that \( \text{count}(e') \leq \lceil \frac{1}{3} \text{size}(e') \rceil \) for every such \( e' \), so these elements will not trigger this rebalancing step again until they take part in subsequent insertions or deletions.

This approach is a prototypical example of the “lazy” amortized design philosophy, where we leave parts of a data structure relatively unattended until they become sufficiently modified so as to warrant wholesale rebuilding. By rebuilding a subtree when its imbalance count exceeds \( \frac{1}{3} \) of its size\(^6\), our tree stays \( \frac{2}{3} \)-balanced, so its

---

\(^6\)We use \( \frac{1}{3} \) since it makes the math work out nicely, but any constant in \((0, 1)\) would also work.
Figure 6.11: The impact of using single rotations to pull the lowest element in a degenerate path-shaped tree to the root (on left), and splaying the lowest element (on right). As a side effect, splaying roughly halves the depth of the tree.

height is $O(\log n)$ following our earlier analysis of $\alpha$-balanced trees. Large batch rebuilds may cause insert and delete to run slowly in the worst case, but they happen with sufficiently low frequency that we can prove $O(\log n)$ amortized time bounds for both operations. [Details]

The amortized rebalancing idea above leads to another related mechanism, sometimes called either a scapegoat tree or a general balanced tree, which maintains $O(\log n)$ worst-case height while remarkably storing no augmented information at all! The main idea here is that if we notice an insertion occurring at too large a depth (at depth exceeding $\log_3 \frac{n}{2}$), then we know one of its ancestor nodes is to blame (i.e., is a “scapegoat”) for being too unbalanced, so we walk up the tree, rebalancing each subtree as we go, until the issue has been corrected, with the depth of our tree no longer exceeding $\log_3 \frac{n}{2}$. With a similar analysis as above, we can show that insert and delete both still run in $O(\log n)$ amortized time. [Details]

Problem 101 (Rank-Sensitive Priority Queues). In the comparison model, any data structure (e.g., a priority queue) that always knows its minimum element must require $\Omega(\log n)$ worst-case time for either insert or delete. However, this bound only really applies when we are inserting or deleting elements of low rank (close to the minimum). Here, we build a so-called rank-sensitive priority queue, supporting find-min in $O(1)$ time and insert and delete in $O(\log(n/r))$ time (the best possible in the comparison model), where $r$ is the rank of the element being inserted or deleted. Such data structures are ideal if we only occasionally need the functionality of a priority queue; otherwise, an average element takes only $O(1)$ expected time to insert or delete. We show two elegant ways to obtain rank-sensitive priority queues by modifying different types of balanced BSTs. Note that unlike BSTs, which can find any element quickly, these structures only support an efficient find-min operation just like any other standard priority queue.

(a) Suppose we relax the implementation of a BST with amortized rebalancing described above, so that every right subtree remains “unbuilt”, much like with a radix heap (Section 5.5.3). Please show how such a structure can be used to implement insert and delete in $O(\log(n/r))$ amortized time. [Solution]

(b) Another way to implement a rank-sensitive priority queue is to modify a treap. Typically, we store the “actual” elements in the BST keys of a treap, setting the heap keys randomly to ensure balance. Suppose instead that we store the actual elements in our
structure in the heap keys, and choose the BST keys at random. Please show how to implement such a structure so that insert and delete run in $O(\log(n/r))$ expected time. As a hint, try not to store the BST keys explicitly. [Solution]

6.2.7 Splay Trees

If we rotate an element to the root every time it is accessed, then frequently-accessed elements should end up near the root, making them faster to access over time. Unfortunately, this heuristic alone does not seem to give any provable guarantees with respect to balance, but a slight generalization using “double rotations” remarkably does. When rotating element $e$ to the root, we look two steps ahead, at $e$’s parent and grandparent:

- If the parent and grandparent are out-of-line (e.g., $e$ is a left child but parent($e$) is a right child), then we rotate $e$ upward twice, just as before.

- If $e$ is in line with its parent and grandparent (e.g., $e$ and parent($e$) are both left children), we perform the next two rotations in reverse order, first rotating along the parent-grandparent edge, and then along the edge from $e$ to its parent.

We continue rotating $e$ up the tree in two-step increments in this fashion, until either $e$ reaches the root, or $e$ lands one step away from the root, in which case we perform a single rotation to place $e$ at the root. This process is known as splaying $e$, and as we see in Figure 6.11, it can be much more effective at “flattening out” a tree as a side effect than single rotations alone. A splay tree is a BST in which an element is splayed every time it is accessed (say, with find or select).

An element is inserted into a splay tree just as in a standard BST, then splayed. To delete an element, we splay it and then replace it with the join of its two subtrees. Splay trees are particularly adept at implementing split and join. To execute split($e$), we splay $e$ and detach its right subtree. To join trees $T_1$ and $T_2$, we can either splay the maximum element of $T_1$ and attach $T_2$ as its right child, or splay the minimum element of $T_2$ and attach $T_1$ as its left child.

Like skew heaps (Section 5.4.2), splay trees are known as self-adjusting data structures, since all they do is blindly apply simple local restructuring rules, without relying on any augmented information such as subtree sizes. Despite this apparent lack of sophistication, one can show that any sequence of $k$ operations in a splay tree (starting from an empty tree) takes $O(k \log n)$ time, so even though they do not necessarily adhere to any strict notion of balance, splay trees support all standard BST operations in $O(\log n)$ amortized time. [Detailed analysis]

There is nothing particularly special about the specific restructuring rule above we use for splaying, since many similar multi-rotation local restructuring rules have now been shown in the literature to give similar performance bounds. The rule above tends to be the most popular, however, since it comes from the original paper by Sleator and Tarjan that introduced splay trees.

Properties of Splay Trees. Splay trees are known to satisfy many impressive properties, and they are conjectured to satisfy many others. Consider an access
sequence \( S \) in which we access \( m \) elements \( e_1, e_2, \ldots, e_m \) one after the other in an \( n \)-element splay tree.

- **The Dynamic Finger Property.** Just as with RBSTs and treaps (problem 99), splay trees support efficient searching for an element \( e \) close to a previously-accessed element \( e' \). If \( \Delta_j = |\text{rank}(e_j) - \text{rank}(e_{j-1})| + 1 \), then a splay tree can execute \( S \) in \( O(m + n + \sum_j \log \Delta_j) \) time, thereby effectively accessing \( e_j \) in only \( O(\log \Delta_j) \) amortized time. Note also that this implies that it takes only \( \Theta(n) \) time to access all \( n \) elements in a splay tree in order, matching the performance of a traditional inorder traversal (and also note the distinction between an inorder traversal and the process of accessing — and therefore splaying — all the elements of a tree in order).

- **The Working Set Property.** Let \( D_j \) denote the number of distinct elements accessed prior to \( e_j \) starting from the last time we accessed this same element. A splay tree can execute \( S \) in \( O(m + n \log n + \sum_j \log D_j) \) time, thereby effectively accessing each element \( e_j \) in amortized time \( O(\log D_j) \). If we only access elements in a small “working set” whose size \( k \) is much smaller than \( n \), all accesses therefore only take \( O(\log k) \) amortized time.

- **The Static Optimality Property.** Even though a splay tree doesn’t know \( S \) in advance, the time it takes to execute \( S \) is at worst a constant factor times that of a static BST (whose shape cannot change over time) designed to execute \( S \) in a minimal amount of time. Chapter 11 shows how to compute such an optimal static BST in \( O(n^2) \) time using dynamic programming.

Most of these are easy to prove by modifying the basic amortized analysis of splay trees, with the notable exception of the dynamic finger property, proved in a pair of journal papers totaling 85 pages! [Proofs of the remaining properties]

A prominent open question in the field of data structures today is whether or not splay trees satisfy the **dynamic optimality conjecture**, which claims that splay trees take only a constant factor more time than any tree that can dynamically reconfigure itself with rotations, even one that knows the access sequence in advance (further elaboration appears in the endnotes). If true, this would be quite remarkable!

**Problem 102 (A Hybrid FIFO Queue and Priority Queue).** In Chapter 4, we discuss several implementations of a min-queue, supporting enqueue and dequeue in \( O(1) \) amortized time and the ability to find, but not extract, the minimum in \( O(1) \) time. Here, we add the ability to delete arbitrary elements (including the minimum) in \( O(\log n) \) amortized time. It is conceivable that a splay tree can achieve these bounds if we simply store the contents of our FIFO queue as a sequence within the splay tree, and augment each element with a pointer to the minimum element in its subtree — see the section of the endnotes on the deque conjecture for splay trees for more information. However, in this problem, we use an approach that is simpler to analyze by using a pair of splay trees. In addition to the constant amortized bounds for enqueue and find-min, please show how to achieve deletion of an arbitrary element \( e \) in \( O(\log m) \) amortized time, where \( m \) is the number of elements from \( e \) onward in the queue. For example, dequeue should run in \( O(1) \) amortized time, since \( m = 1 \), and if we enqueue an element and then subsequently remove it, then \( m = n \). Such a data structure has been called a queap in the literature, being a hybrid between a queue and a heap. As a hint, use the dynamic finger property. [Solution]
6.3 B-Trees

We now leave the world of binary trees behind as we introduce the B-tree, where nodes contain multiple elements and may have multiple children. A non-leaf node with \( k \) elements has \( k + 1 \) children, as shown in Figure 6.12, from which one can easily see how the BST property generalizes in this situation. Just like a BST, the B-tree also encodes a sequence from left to right that we can recover by a traversal similar to an Euler tour traversal. When used as a dictionary, the sequence is sorted. B-trees are “complete” in the sense that all leaves have the same depth.

Every B-tree is parameterized by a number \( B > 1 \) describing the number of elements a node may contain. Every node except the root must have between \( B - 1 \) and \( 2B - 1 \) elements, inclusive (and therefore \( B \ldots 2B \) children). The root is special, having between 1 and \( 2B - 1 \) elements (i.e., there is no lower bound). The B-tree with \( B = 2 \) is called a 2-3-4 tree, since every node may have 2, 3, or 4 children. As we mentioned earlier in the chapter, there is a direct correspondence between 2-3-4 trees and red-black trees, so we can generally avoid discussion of the somewhat involved mechanism for balancing a red-black tree by instead looking at the same process from the simpler viewpoint of balancing a 2-3-4 tree. [Further details]

In practice, \( B \) is usually quite large (say, in the thousands), making the height of the tree (at most \( \lceil \log_B n \rceil \)) quite small. B-trees are often used for large datasets stored on slow block-transfer media like disk, where accessing data is very slow, but we can transfer an entire “page” of data into main memory all at once. By setting \( B \) just large enough that a page from disk contains an entire node, this allows us to perform only \( O(\log_B n) \) disk accesses for each B-tree operation (versus \( O(\log_2 n) \) for a BST). For example, if \( B = 2^{10} = 1024 \), a B-tree needs to access the disk roughly 10 times less often than a BST. If we keep the top two or three levels of the B-tree in main memory, we can usually reach any element with only one or two additional disk accesses.

Locating an element in a B-tree by its key or its rank (if we have augmented nodes with subtree sizes) is straightforward. Just as with a BST, we walk down the tree guided by the BST property or subtree sizes. Along the way, we examine \( O(\log_B n) \) nodes, each containing \( O(B) \) elements, for a total running time of \( O(B \log_B n) \). However, remember that in the common use of the B-tree in an external memory setting, the “\( O(\log_B n) \)” part usually matters much more than the “\( O(B) \)” part.

**Insertion and Deletion in a B-Tree.** B-trees support a notion of rotation just like BSTs, as shown in Figure 6.13(a). By rotating elements through a parent node,
a node can transfer elements to or from its adjacent siblings. Other fundamental
B-tree operations include splitting a node into two adjacent siblings, and the reverse
operation of joining two adjacent siblings together, shown in Figure 6.13(b). We
only split nodes that are at least full (with at least $2B - 1$ of elements), since
otherwise we risk creating nodes that are “underfull”, containing fewer than $B - 1$
elements. Similarly, we only join adjacent siblings if both contain at most the
minimum possible number $B - 1$ of elements, since otherwise we risk creating a
node that is too full. Rotations and the split and join operations all require $O(B)$
time if the elements in a node are stored in an array, as is typical.

In order to insert into a B-tree, we first search to find a leaf node into which our
new element should be placed. If this leaf node contains fewer than $2B - 1$ elements
we can simply insert the new element. If not, we have two alternatives: if one of
our siblings is not full, we free up space for the new element by donating one of our
elements to the sibling with a rotation. Otherwise, we can insert the new element
and then split the current node. Since splitting a node donates an element to its
parent, this may in turn cause the parent to donate a key to a sibling via rotation
or be split. If this chain of splits propagates far enough, we may end up splitting
the root, and this is the only way our B-tree increases in height.

Deletion is completely symmetric, possibly resulting in a propagating chain of join
operations that, if it reaches the root, can decrease the overall height of the tree.
We generally only delete from leaf nodes. To delete an element $e$ from a non-leaf
node, we can first swap $e$ with its predecessor or successor, much like deletion from
a standard BST. Both of these elements will belong to leaf nodes if $e$ does not.

Both insert and delete run in $O(B \log_B n)$ time, since they access $O(\log_B n)$ nodes
each of size $O(B)$. This asymptotic running time is unaffected even if we need
to maintain augmented information in the tree (e.g., subtree sizes), as long as the
augmented information is locally-recomputable.

One can improve the running time of every standard B-tree operation in theory
by storing the $O(B)$ elements in each node not as an array, but instead as a tiny
balanced BST. This reduces the time required to interact with each node (e.g.,
searching a node, or splitting / joining nodes) from $O(B)$ to $O(\log B)$, improving
the overall running time for each B-tree operation to $O(\log B \log_B n) = O(\log n)$. This would probably not sensible in practice, though, since $B$ is usually not large enough to make $O(\log B)$ with a larger hidden constant a big win over $O(B)$ with a small hidden constant.

**Problem 103 (Constant Amortized Memory Writes).** Assuming $B = O(1)$ for simplicity, insertion and deletion in a B-tree require $O(\log n)$ individual memory reads but only $O(1)$ amortized memory writes\(^7\), saving time in most computing environments where writes are more costly than reads. Please prove this amortized constant bound. As a hint, you should determine the amount of “credit” associated with a node in 4 states: underfull by one element, minimally full, maximally full, and overfull by one element (the credit in all other cases can be set to zero). [Solution]

**Leaf-Oriented Trees.** Sometimes we design BSTs or B-trees with elements stored only in leaves, known as leaf-oriented trees. If used as a dictionary, we would typically augment interior (non-leaf) nodes so they store the minimum and maximum keys present in their subtrees; this information is sufficient to allow find and all other standard tree operations to work as efficiently as a non-leaf-oriented structure. A leaf-oriented B-tree whose leaves are all connected in a long doubly-linked list (to facilitate enumeration and range queries) is known as a B+-tree, and it is perhaps the most popular of several variants / extensions of B-trees found in practice.

### 6.4 Skip Lists

The downside of implementing a dictionary using a sorted doubly-linked list is the excessive time it takes to scan to a particular location in the list (say, to locate an element based on its key or rank). We can alleviate this problem, however, by introducing an “express” list — just like an express train or bus, this list makes fewer stops, getting you close to your destination much faster. The skip list is a data structure built on this idea, and it can be made to support all fundamental operations on dynamic sequences and dictionaries in $O(\log n)$ time with high probability, just like with an RBST or treap.

Figure 6.14 illustrates the structure of a skip list, a collection of doubly-linked lists stacked on top of each-other, where the lowest (level zero) list contains every element (in sorted order, for a dictionary), and each successive level contains a subset of roughly half the elements on the level beneath it. We maintain separate records in memory for an element on each of the levels in which it is present, connected together by “vertical” pointers in a doubly-linked list. Each list is preceded by a

\(^7\)One must be slightly careful with the definition of a “B-tree” for this result to apply. The original definition in the literature allowed each node to have only between $B$ and $2B - 1$ children, which in fact may initially seem more natural since splitting an overfull node now results in exactly two minimally-filled nodes, and joining a minimally-filled node with an underfull node now results in a node exactly at its upper capacity. However, this variant behaves poorly from an amortized perspective since an insert can set off a cascade of splits going all the way to the root, after which a delete of the same element would effectively undo all of these with joins, putting the tree back in its original state. An alternating series of such operations therefore requires $\Theta(\log n)$ memory writes per operation, whereas if we just add a small amount of extra slack, allowing nodes to have up to $2B$ children, then we get instead a constant amortized bound. If we relax the upper bound further, allowing maximum node sizes above $2B$, this only helps from an amortized perspective.
Figure 6.14: A skip list. Each horizontal link is augmented with its “skip count” (only nonzero skip counts are shown above).

special “start” element that we ensure is present in every level.

To find an element based on its key or rank, we start at the beginning of the top-level list and scan to the right, moving downward whenever we realize that the next link would skip too far ahead. Rank-based access requires augmenting each horizontal pointer with its “skip count” (much like augmenting a BST with subtree sizes), so we can keep track of our accumulated rank as we move through the structure. All other extended BST operations, such as find-min / find-max, pred / succ, split / join, range queries, and inorder traversals, can be translated in a straightforward manner to the skip list.

The insert operation employs randomness in a clever way: to insert an element, we find the appropriate location for it in the level-0 list and insert a copy there. We then repeatedly flip a fair coin and keep inserting copies into successively higher levels as long as we flip heads. That is, whenever we insert an element at level \( k \), we flip a fair coin and with probability \( 1/2 \), we also insert it at level \( k + 1 \). This ensures that roughly half the elements appear in the level 1 list, roughly one quarter appear in the level 2 list, and so on. One can show that the total number of levels is \( O(\log n) \), and that the total space occupied by the structure is \( \Theta(n) \) (both bounds holding with high probability). To delete an element, we first find it and then remove all copies from every level on which it exists. With a simple analysis based on randomized reduction, we can show that all standard BST operations run on a skip list in \( O(\log n) \) time with high probability. [Analysis of skip lists]

Skip Lists Versus BSTs. Skip lists are elegant from a theoretical perspective and efficient and easy to implement in practice. They also provide a nice alternative to the tree-based data structures discussed earlier in the chapter. In fact, they were originally introduced as a simpler alternative to the balanced BST. Interestingly, with a bit of care, one can turn the skip list “on its side” and view it as yet another type of BST with a randomized balancing mechanism involving upward rotation based on random coin flips whose complexity is roughly on par with that of RBSTs and treaps. The resulting BST has edge weights that all sum to the same value along every root-to-leaf path (owing to the fact that all elements in the bottom row of a skip list live at the same depth relative to the starting point), so this structure could also be characterized as a randomly-balanced type of path-balanced tree. [Interpreting a skip list as a BST].
One final advantage of skip lists is that by inserting \( n \) elements into a skip list and then reading them out in sorted order, we can obtain a comparison-based sorting algorithm that makes \( O(n \log n) \) memory reads but only \( \Theta(n) \) memory writes (both bounds holding with high probability). Among the other data structures described in this chapter, only the treap, \( B \)-tree, and red-black tree (equivalent to a \( B \)-tree for \( B = 2 \)) offer this feature (see problems 100 and 103). The fact that we make only a small number of memory writes per data structure update is particularly useful with the persistence technique described in Section 4.7, since the size of a persistent data structure is directly determined by the number of memory writes we make over the lifetime of the structure. Skip lists also support the ability (like treaps, RBSTs, and splay trees) to move from a previous element \( e' \) to a new element \( e \) in (expected) time proportional to the logarithm of the rank difference between the two.

In Section 7.5.3 in the next chapter, we will see how to use ideas inspired by skip lists to route information efficiently through a distributed hash table, much like in the following problem.

**Problem 104 (Level Ancestors).** The level ancestor problem involves preprocessing a static \( n \)-node rooted tree (not necessarily a binary tree, and not necessarily balanced in any way) so that we can quickly answer queries of the form “what is element \( e' \)’s ancestor at depth \( d \) in the tree?”. In other words, given any element \( e \) we need to be able to quickly jump an arbitrary “distance” upward along the path from \( e \) to the root. Level ancestors are easy to compute on a path (i.e., a non-branching tree), since we can simply store the path in an array and jump to the appropriate ancestor index in \( O(1) \) time. In this problem, we investigate fast data structures for this problem on arbitrary trees; many of these structures are motivated in part by the same general ideas as skip lists.

(a) Suppose we augment every node in the tree with \( \log n \) pointers that send us upward 1, 2, 4, 8, etc. nodes in the tree. Show how to build such a data structure using \( O(n \log n) \) time and space, and show how it can answer level ancestor queries in \( O(\log n) \) time. Next, please show how to use skip lists to improve this result so that we spend only \( \Theta(n) \) expected preprocessing time and space, such that queries can be answered in \( O(\log n) \) time with high probability. [Solution]

(b) Suppose we decompose a tree into node-disjoint paths as follows: we first locate the longest root-to-leaf path and remove it from the tree. This potentially splits the tree into several disjoint trees, and on each of them we proceed to again locate and remove the longest root-to-leaf path, until eventually every node belongs to some path in our decomposition. Show how to build such a path decomposition in \( \Theta(n) \) time/space. Now, we employ a neat trick: take each path of length \( L \) and extend it upward to a length of \( 2L \) by including the \( L \) most immediate ancestors above it in the tree (it is possible that fewer than \( L \) ancestors exist, if we are near the top of the tree). Our paths may now cease to be disjoint, but this is fine since they still occupy only linear space (the combined length of all extended paths is at most \( 2n \)). Please show how we achieve \( \Theta(n) \) preprocessing time/space and \( O(\log n) \) query time with this structure. As a hint: show that you can move upward through the tree using steps of exponentially increasing size. [Solution]

(c) Finally, see if you can combine (a) and (b) to obtain \( O(n \log n) \) preprocessing time and space, with only \( O(1) \) query time (as mentioned in the endnotes, the preprocessing bounds can even be improved to \( \Theta(n) \)). [Solution]

**Problem 105 (Adaptive Sorting Algorithms).** When we studied sorting algorithms in Chapter 3, we learned that insertion sort has a running time of \( \Theta(n + I) \), where
I denotes the number of inversions present in our input sequence. If I is small (say, on the order of n), then insertion sort can run much faster than our favorite $\Theta(n \log n)$ sorting algorithms such as merge sort and quicksort. In fact, there must exist some “cutoff” value of I, below which it is preferable to use insertion sort, and above which it is preferable to use merge sort or quicksort. This discontinuity is somewhat unsightly — it would be nice to have only one sorting algorithm that does well for all values of I. Such a sorting algorithm is said to be adaptive with respect to I. Many researchers have studied adaptive sorting algorithms, some of which scale gracefully in terms of I and others that involve different measures of disorder in a sequence; see the endnotes for further references. Suppose we implement insertion sort by maintaining the sorted prefix of our array in a skip list. Show that this gives a running time of $O(n + n \log(1 + I/n))$ with high probability, which is always at least as fast as both $O(n + I)$ and $O(n \log n)$ for all values of I. \[Solution\]

**Problem 106 (A “Skip List” Approach to Persistence).** In this problem, we describe the high-level details of a randomized analog of the amortized technique introduced in Section 4.7 for making a pointer-based data structure persistent. Our approach works in exactly the same setting as the approach from Section 4.7: we start with an ephemeral (non-persistent) structure in which every element occupies $O(1)$ memory and has at most $k = O(1)$ incoming pointers from other elements (e.g., $k = 3$ for a BST with parent pointers), where every element is accessed by following a sequence of pointers starting from a designated root node. Every node $x$ should be augmented with a time-sorted “modification list”, where each entry contains a time stamp $t$ as well as the complete contents of node $x$ as of time $t$. Any pointer to node $x$ in this new persistent structure actually points at an entry in a $x$’s modification list, rather than at $x$ directly. As opposed to the method in Section 4.7 in which modification lists were limited to $k+1$ entries before they were split\(^8\), here we store each modification list as an array or linked list that can grow without bound (the root list should be an array, to facilitate binary searching).

To modify node $x$ as of time $t$, we add a new entry to the end of $x$’s modification list (in only $O(1)$ time) with time stamp $t$. We then flip a biased coin: with probability $1 - p$, we do nothing more, and with probability $p$, we modify all of the nodes currently pointing at $x$ so they now point to the new entry in $x$’s modification list. Modification of these “parent” nodes is done just as when we modified $x$, by adding a new record to their modification lists, and flipping coins to see if the modification should be propagated backward yet further, much like in a skip list. Any time a modification propagates backward to the root, it stops at that point. Alternatively, we can allow propagation of a modification to continue backward from the root. In this case, we introduce a new root node pointing at the original one, and again keep flipping coins to see if the modification should propagate backward even more. This effectively makes the modification of the root have the character of a skip list, instead of keeping it as one large time-sorted list that we binary search. You may want to focus on the array version for this problem since its analysis is slightly simpler, owing to the fact that we can avoid the extra analysis involved with the root.

Please show how to choose an appropriate value of $p$ that (a) adds only $O(1)$ expected time to the operation of modifying any node, and that (b) results in $O(Q + \log T)$ expected query time, where $Q$ is an upper bound on the query time of the original ephemeral structure, and $T$ is the number of original update operations we have invoked (equivalently, the number of points in time at which we have historical versions to track). Observe that these bounds match the amortized bounds from Section 4.7. You may need to fill in yet-unspecified details of the structure as part of your solution. \[Solution\]

\(^8\)Now that we know B-trees, it is worth noting the strong resemblance between our partial persistence amortized splitting mechanism and the cascading split mechanism in a B-tree, which also exhibits good amortized behavior.
This chapter continues our discussion of *dictionary* data structures from the last chapter, only now in the RAM model of computation. Using a *hash table*, we will see how to support the basic dictionary operations *insert*, *delete*, and *find* in only constant time (possibly amortized or in expectation). This improves on the logarithmic time bounds of the comparison-based structures from the last chapter\(^1\), and lets us build extremely fast *set* and *map* data structures, which have wide-ranging applicability. Hash tables, and the underlying technique of *hashing* on which they are based, are absolutely fundamental concepts in computing, with tremendous influence both in theory and practice\(^2\). Proficiency in hashing is a strong indicator of a well-trained computing professional.

We begin this chapter with a detailed discussion of hash tables, focusing particularly on *universal* hashing, the concept most hash tables have to thank for their strong theoretical performance guarantees. Since hash tables generally only support the basic dictionary operations *insert*, *delete*, and *find*, we next turn our attention to more general RAM-based “search structures” that also support many of the extended operations provided by balanced binary search trees and their relatives from the last chapter (e.g., *find-min* / *find-max* and *pred* / *succ*). If our keys are integers in the range \(0 \ldots C - 1\), we will see how a *radix tree* can implement *insert*, *delete*, and *find*, as well as many extended operations in \(O(\log C)\) time, and we will see how to improve this to \(O(\log \log C)\) (possibly amortized or in expectation) with the *Y-fast tree* and its equivalent relative, the *van Emde Boas (vEB) structure*. As a consequence, we can now sort in the RAM model in \(O(n \log \log C)\) expected time and build a RAM priority queue whose operations take \(O(\log \log C)\) time (possibly amortized and/or in expectation).

At the end of the chapter, we show how the more general idea of *hashing* (mapping a complicated object down to a simpler representation, often a single integer) has numerous applications beyond just data structures, in domains like security, distributed algorithms and data management, and the analysis of massive data sets.

\(^1\)Although to be fair, this is a lopsided comparison, since the RAM is a stronger model of computation. Comparison-based structures like binary search trees are essentially the best one can do within the confines of the comparison-based model, where lower bounds on sorting and other related problems pose fundamental obstacles to achieving fast running times.

\(^2\)Hashing has even entered popular culture with the advent of #hashtags, used as dictionary keys behind the scenes to help group together related social media postings.
7.1 Hash Tables

Suppose we want to design a RAM dictionary capable of storing integer keys in
the range 0...C−1. Since integer keys can be used to index into an array, one
solution is to use a direct access table A[0...C−1], where an element with key k
is stored in A[k], or if there is no such element, then A[k] would be set to some
designated “uninitialized” value. Although this structure is trivial to implement
and supports insert, delete, and find all in O(1) worst-case time, it has a critical
drawback: memory usage. Since C is usually very large (often larger than our entire
memory), the direct access table is usually not feasible due to space considera-3
tions. Ideally, a dictionary storing n keys should occupy only Θ(n) space.

Rather than storing our n elements in a direct access table of size C, we instead use
a much smaller hash table A[0...m−1] where typically m = Θ(n). Each element
is mapped to a location in the table using a hash function, h(k), taking a key
k ∈ {0,...,C−1} and mapping it to a table index h(k) ∈ {0,...,m−1}. To find
an element with key k, we would therefore inspect A[h(k)]. We usually assume that
h(k) can be evaluated in O(1) time. However, we can also design hash functions
that work with larger items and hence take longer to evaluate — say, if we want to
store or index a collection of text strings or even large files. We discuss methods for
hashing large objects later in the chapter, in Section 7.5.5. For now, we continue to
assume that our keys are just integers in 0...C−1.

Hashing works quite well except for the case when two elements are mapped to
the same location, known as a collision. Collisions are unavoidable in any func-
tion mapping a large range down to a smaller range, so we should always expect
collisions no matter what hash function we use. Most of the difficulty with hash
tables lies in dealing with collisions gracefully — although collisions don’t cause
a properly-designed hash table to malfunction, they usually do slow it down. To
minimize collisions in the first place, a good hash function spreads elements across
the m locations in our table as haphazardly as possible. The less predictable our
function, the better it generally performs. This explains the origin of the term
“hashing”, since the word “hash”, when used as a verb, can mean to jumble or
mix up. Hash functions that feel “random” are generally good, and as we will see
shortly, randomization is indeed often used to build effective hash functions.

7.1.1 Collision Resolution

There are several good ways to deal with collisions. We discuss perhaps the two
simplest here, both of which happen to sound like rather unpleasant forms of torture:
probing and chaining. Both work well and are popular methods in practice, though
chaining tends to be easier to analyze mathematically.

Resolving Collisions with Probing. With probing (also called open addressing),
the hash table is a single array of size m ≥ n. As shown in Figure 7.1, our n elements
of data are each stored directly in this array, with the key of each element stored
alongside it. Unused table entries are marked as such during initialization. To

---

3One might also worry about the time required to initially set all the entries to “uninitialized”,
although in theory we could use virtual initialization (problem 2) to reduce this from Θ(C) down
to O(1) at the expense of even more memory.
Set of (key, value) pairs:

- (657, accounting)
- (364, criminology)
- (821, English poetry)
- (113, cosmology)
- (774, holography)
- (513, arithmetic)

Hash function:
\[ h(k) = k \mod 10 \]

Hash table:

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(821, English poetry)</td>
</tr>
<tr>
<td>1</td>
<td>(364, criminology)</td>
</tr>
<tr>
<td>2</td>
<td>(513, arithmetic)</td>
</tr>
<tr>
<td>3</td>
<td>(774, holography)</td>
</tr>
<tr>
<td>4</td>
<td>(113, cosmology)</td>
</tr>
<tr>
<td>5</td>
<td>[deleted item]</td>
</tr>
<tr>
<td>6</td>
<td>(364, criminology)</td>
</tr>
<tr>
<td>7</td>
<td>(513, arithmetic)</td>
</tr>
<tr>
<td>8</td>
<td>(657, accounting)</td>
</tr>
</tbody>
</table>

**Figure 7.1:** An example of a hash table where linear probing is used to resolve collisions. Our elements of data are (key, value) pairs representing subject categories from the Dewey decimal system. The element formerly in position 6 of the hash table has been deleted, and we can deduce that this must have occurred after the element (364, criminology) was inserted, since otherwise this element would have filled the slot left open by the deleted element. If next we were to insert (874, Latin lyric poetry), then we would probe locations 4 and 5 before finally placing this new element in position 6.

Insert a new element \( e \) with key \( k \), we start at index \( h(k) \). If unused, we store \( e \) in this position. Otherwise, we scan forward sequentially until we find an unused location, wrapping around back to the beginning if we scan past the end of the table. To find an element with key \( k \), we start scanning from index \( h(k) \) until we locate an element with our desired key, or until we reach an unused entry, in which case we conclude \( k \) is not present in the table. Deletion is straightforward except for one small subtlety: we cannot simply delete an element by marking its array entry as “unused”, as this might break subsequent find operations, causing them to stop scanning too early. Instead, we mark the entry of our dearly-departed element with a so-called tombstone annotation, telling find operations to keep scanning, but instructing insert operations to stop and re-use the entry.

Probing works well for a sparsely-filled table. However, once \( n \) grows almost as large as \( m \), large “clumps” of consecutive elements can appear, leading to increasingly slow performance, since larger clumps attract larger numbers of collisions. To alleviate this problem, it is sometimes recommended to use fancier “probing patterns”. For example, the approach above uses linear probing, where we examine array entries \( (h(k) + i) \mod m \) for \( i = 0, 1, 2, \ldots \) in sequence. Instead, we may consider probing locations \( (h(k) + g(i)) \mod m \), where \( g \) is a more complicated function; for example, quadratic probing uses \( g(i) = ai^2 + bi \), where \( a \) and \( b \) are constants. Alternatively (see problem 109), we could probe locations \( h_1(k), h_2(k), \ldots \) according to a sequence of different hash functions that are either implicitly defined or explicitly generated and stored on demand. By making the probe sequence skip around in a more haphazard fashion, we may be less prone to generate large clumps. However, this can also lead to rather poor cache performance, so it not at all a foregone conclusion that there will be any improvement in practice. Empirical testing is advised.
Problem 107 (Batch Construction with Linear Probing.). Suppose we wish to insert \( n \) elements into a hash table of size \( m = \Theta(n) \) \((m \geq n)\) using linear probing. Analyzing the time required for \( n \) successive calls to \textit{insert} is tricky and also highly dependent on our choice of hash function. However, please show how we can build the hash table in \( \Theta(n) \) time and space in a “batch” fashion, irrespective of our hash function. [Solution]

Depending on our choice of hash function and probing strategy, mathematical running time analysis can be somewhat challenging. In a few pages, we will have the tools to create hashing schemes based on probing where \textit{insert} and \textit{find} both run in \( O(1) \) expected time (these two running times always match, since inserting an element requires the same amount of probing as it takes to subsequently find it). Deletion of an element takes only \( O(1) \) worst case time, assuming that we adopt our usual convention of defining \textit{delete} so it takes a pointer directly to an element in the table, to separate its functionality from \textit{find}.

Resolving Collisions with Chaining. In a hash table using \textit{chaining}, each entry points to a linked list containing all of the elements hashing to that entry, as shown in Figure 7.2. To insert an element, we hash its key to an index in the table, then \textit{prepend} (not \textit{append}) the element to its linked list. This takes \( O(1) \) worst-case time as long as we can evaluate our hash function in \( O(1) \) time. Deletion takes \( O(1) \) worst-case time given a pointer directly to an element, since we can just unlink the element from its enclosing list (here, we would need to use doubly-linked lists or end our lists with dummy sentinel elements).

To \textit{find} an element with key \( k \), we search the linked list attached to the entry \( h(k) \) to see if we find an element with key \( k \). Assuming we can hash in constant time, this takes \( O(1 + L) \) total time if we end up searching a list of length \( L \). In a perfect world, our hash function would distribute our \( n \) elements uniformly throughout our size-\( m \) table, so every linked list would have length at most \( \lceil n/m \rceil \), and \textit{find} would therefore run in \( O(1 + n/m) \) time. In reality, this is too much to hope for in the worst case, but when we study universal hashing shortly, we will see how to use...
randomization to guarantee an $O(1 + n/m)$ expected running time for $\text{find}$, which becomes $O(1)$ expected time if we ensure that $m = \Theta(n)$.

### 7.1.2 Guidelines for Designing a Good Hash Function

The first line of defense against collisions is our choice of hash function. In the next section, we will design hash functions that give strong provable performance guarantees for $\text{insert}$, $\text{delete}$, and $\text{find}$, allowing us to design hash tables that perform well both in theory and in practice. However, one also finds “ad hoc” hash functions used quite commonly in practice, and despite their lack of provable guarantees, these usually perform reasonably well as long as they follow a few general guidelines.

A good hash function should fully utilize “all the bits” of a key. For example, the natural function $h(k) = \lfloor mk/C \rfloor$ that just linearly rescales $0 \ldots C - 1$ down to $0 \ldots m - 1$ places too much emphasis on the “higher-order” part of a key. Keys close together in value will tend to collide, which is quite undesirable. We generally want $h(k)$ and $h(k + 1)$ to differ substantially and unpredictably. Moreover, $C$ and $m$ are often both powers of 2, in which case this function completely ignores the lower-order bits of $k$. Another natural function, $h(k) = k \mod m$, places too much emphasis on the “lower-order” part of a key. Keys spaced out by multiples of $m$ will collide, and this can be particularly bad if $m$ is a power of 2, since then the hash function completely ignores the higher-order bits in a key. A close relative prone to the same issue is $h(k) = (ak) \mod m$ for some parameter $a$. In fact, this variant can be dangerous for another reason — as we will see in Chapter ??, if $a$ and $m$ share common factors, then $(ak) \mod m$ can only output a subset of the indices $\{0, \ldots, m - 1\}$ in our table. For example, if $a$ and $m$ are both even, then $(ak) \mod m$ must also be even.

The “universal” hash functions we will introduce shortly give several nice examples of simple functions that circumvent the pitfalls above.

### 7.1.3 Choosing the Size of a Hash Table

Since we usually do not know in advance the number of elements $n$ we will end up storing, we typically use amortized table expansion and contraction (Section 4.4) to maintain the size of our table always at $m = \Theta(n)$. This ensures that $\text{find}$ runs in $O(1)$ expected time with both chaining (if we use universal hashing) and probing (if we use appropriate hash functions and keep $m$ at least a small constant factor larger than $n$). For example, if $n$ grows too large, we might re-hash the contents of our table into a newly-allocated table of size, say, $2m$. Note that rehashing is usually necessary when $m$ changes, since most hash functions depend on $m$, so changing $m$ might change where every element is mapped.\(^4\)

Rebuilding a table takes only linear time\(^5\), so by our standard amortized rebuilding analysis, it contributes only $O(1)$ additional amortized time for each call to $\text{insert}$ or $\text{delete}$. We must therefore add “amortized” qualifiers to these operations.

\(^4\)Hash functions that do not require substantial rehashing when resizing a table are sometimes known as consistent hash functions. We discuss these later in the chapter.

\(^5\)This is immediate with chaining since $\text{insert}$ only takes $O(1)$ time, and with probing, we get this by using the result of problem 107.
CHAPTER 7. HASHING AND INTEGER SEARCH STRUCTURES

7.2 Universal Hashing

If we want to prove any sort of reasonable performance bounds for hashing, we generally cannot use worst-case analysis. Assume $C > m(n - 1)$, which is typically true, since $C$ is usually much larger than both $m$ and $n$. In this case, the “pigeonhole principle” tells us that for any hash function, we can always find a bad set of $n$ keys that all hash to the same location. Otherwise, if at most $n - 1$ key values hash to each of the $m$ table locations, then the total number of possible keys $C$ can be at most $m(n - 1)$. Using this set of $n$ bad keys will turn our hash table into essentially a glorified linked list, with $\Theta(n)$ worst-case performance for find. To prove good performance bounds for hashing, we therefore need to leave worst-case analysis behind and consider instead the expected performance of hash functions that involve randomly-chosen parameters.

The Infeasibility of Fully-Random Hashing. At the beginning of our algorithm’s execution, suppose we select a hash function uniformly at random from all possible functions mapping $\{0, \ldots, C - 1\}$ down to $\{0, \ldots, m - 1\}$. Since this function maps $n/m$ expected elements to each table entry$^6$, find now runs in $O(1+n/m)$ expected time when we resolve collisions with chaining, which is good news. The bad news is that it takes $\Theta(C)$ space to represent a truly random hash function, to remember where each of the $C$ possible input keys should be mapped. With this amount of space usage, we might as well use a direct access table!

Partially-Random Hashing. To overcome the memory issues above, we choose a partially-random hash function specified by a small number of random parameters. Such a function is often “random enough” for our analytical needs if it satisfies strong universality, one of the most important concepts in hashing:

A randomized hash function $h$ is strongly universal if for any pair of keys $k_1 \neq k_2$ and any two table locations $y_1, y_2$, we have

$$\Pr[h(k_1) = y_1 \text{ and } h(k_2) = y_2] = O(1/m^2),$$

where the probability is over the selection of random parameters built into the function.

For example, if we randomly choose integers $a$ and $b$ from $\{0, 1, \ldots, mC - 1\}$, then

$$h(k) = \left\lfloor \frac{(ak + b) \mod (mC)}{C} \right\rfloor$$

is strongly universal, since $\Pr[h(k_1) = y_1 \text{ and } h(k_2) = y_2] \leq 1.25/m^2$ for any two keys $k_1 \neq k_2$ and any two table locations $y_1, y_2$.

A fully-random hash function would map a given key $k_1$ to each table location with probability $1/m$, and it would map a pair of keys $k_1 \neq k_2$ to any two specific table locations with probability $1/m^2$. Strongly universal hashing provides this same guarantee of pairwise independence, only asymptotically, since we allow a

---

$^6$For a randomized hash function to work, we need to ensure that the keys being inserted into our hash table have no dependence on our random choice of hash function. That is, we cannot have a malicious adversary that peers inside the state of our algorithm after it has chosen its hash function, and then chooses a bad set of keys for it.
7.2. UNIVERSAL HASHING

probability bound of $O(1/m^2)$ (some definitions of strong universality are more strict, using $1/m^2$). Hence, strongly universal hashing behaves like fully random hashing as long as we only look at pairs of keys at a time. If we add a third key to the picture, its mapping might exhibit some dependency\(^7\) on the mappings of the first two keys.

By taking a union bound over all $m$ pairs of locations $y_1 = y_2$, strong universality implies a slightly weaker property known simply as universality:

A randomized hash function $h$ is universal if for any pair of different keys $k_1 \neq k_2$, we have

$$\Pr[h(k_1) = h(k_2)] = O(1/m),$$

where the probability is over the selection of random parameters built into the function.

Now for the punchline: universal hashing (and therefore also strongly universal hashing) guarantees that when we resolve collisions in a hash table with chaining, $\text{find}$ runs in $O(1 + n/m)$ expected time. This is easy to prove using linearity of expectation [Proof]. If we use amortized table resizing to maintain $m = \Theta(n)$, a chained universal hash table therefore supports $\text{insert}$ and $\text{delete}$ both in $O(1)$ amortized time, and $\text{find}$ in $O(1)$ expected time.

7.2.1 Acing Your Next Job Interview

Now that we know the fundamentals of universal hashing, it is worth stepping back for a moment to look at some simple yet common problems we can now solve easily and efficiently. These sorts of problems tend to appear quite frequently in job interviews for computing positions, since job interviewers need to differentiate (e.g., “hash”) candidates according to their computational problem solving skills, and knowledge of hashing is an excellent metric to use in this regard.

For example, suppose you are asked during an interview: “given an array containing $n$ integers, how many distinct numbers are present if we remove duplicates?” You could answer this concisely by saying you would insert the numbers one by one into a universal hash table, skipping over the numbers that already exist in the table. The number of elements stored in the hash table at the end gives the desired answer. In terms of running time, we make at most $n$ calls to $\text{insert}$, each running in $O(1)$ amortized time, and we make $n$ calls to $\text{find}$, each running in $O(1)$ expected time. The total running time is therefore $\Theta(n)$ in expectation. To ensure the job is yours, you could mention that using a balanced binary search tree instead of a hash table would give a running time of $O(n \log n)$, which is optimal in the comparison-based model since any faster algorithm would allow you to circumvent the $\Omega(n \log n)$ worst-case lower bound on element uniqueness. If you want to ensure landing the

---

\(^7\)Many strongly universal hash functions are specified by two random parameters (like $a$ and $b$ above), so if we knew the locations $y_1$ and $y_2$ where two keys $k_1$ and $k_2$ are mapped, we could then solve the two equations $h(k_1) = y_1$ and $h(k_2) = y_2$ to determine these parameters, which in turn would determine where a third element would be mapped. Soon, we will see how to build hash functions with $r$-wise independence by incorporating $r$ random parameters.
job and a high salary offer, you could also mention the technique we will discuss later in Section 7.5.1 for how to estimate the answer to this problem when the hash table above would be too large to fit in memory.

There are many common problems taking $\Omega(n \log n)$ worst-case time in the comparison or real RAM models that we can now solve in linear expected time with universal hashing. Examples include finding the intersection, difference, or union of two sets (problem 42), detecting whether two arrays are anagrams (having the same count of each distinct element; problem 54), and re-arranging the contents of an array so as to group equal elements together (Section 3.3.3). Many of these are favorites for job interviewers as well. You may want to pause for a moment to make sure you can now formulate and analyze the hashing-based solutions to these problems with ease.

### 7.2.2 Further Examples of Universal Hash Functions

Several universal families of hash functions have been studied in the literature. The one we mentioned earlier is particularly simple and fast to evaluate, especially if $m$ and $C$ are powers of two (as is typical), since the division and remainder operations can be implemented with right shifts and bitwise ANDs.

**Multiplicative Hashing.** If $m$ and $C$ are powers of two, another option is to choose a random odd integer $a \in \{1, 3, 5, \ldots, C - 1\}$, after which

$$h(k) = \left\lfloor \frac{ak}{C/m} \right\rfloor \mod m$$

is universal (but not strongly universal), since it satisfies $\Pr[h(k_1) = h(k_2)] \leq 2/m$ for any $k_1 \neq k_2$ [Proof]. This is sometimes called multiplicative hashing in the literature, and it can also be evaluated very quickly, using right shifts for division and a bitwise AND for the remainder operation.

**Tabulated Hashing.** Continuing to assume $m$ and $C$ are powers of two, let us regard any key $k$ as a binary string of $b = \log C$ bits $k_1k_2\cdots k_b$. Suppose for each digit $i \in \{1, \ldots, b\}$ that we independently construct a 2-element table $H_i$ whose entries $H_i[0]$ and $H_i[1]$ are randomly chosen from $\{0, \ldots, m - 1\}$. Then

$$h(k) = H_1[k_1] \oplus H_2[k_2] \oplus \ldots \oplus H_b[k_b],$$

where $\oplus$ denotes the XOR operation, is strongly universal. Otherwise stated, we hash each of the bits of our key independently in a completely random fashion, and XOR the results together. This is known as tabulated hashing, due to its use of small lookup tables, and it works equally well if we break our key into larger chunks rather than individual bits. For example, a very practical way to hash a 32-bit integer $k$ is to regard it as a sequence of four bytes $k_1k_2k_3k_4$ (each in the range $0 \ldots 255$). We can then use the hash function

$$h(k) = H_1[k_1] \oplus H_2[k_2] \oplus H_3[k_3] \oplus H_4[k_4],$$

where each of the $H_i$'s is a length-256 array containing random integers independently chosen during preprocessing from $\{0, \ldots, m - 1\}$. This function is also strongly universal. [Proof that tabulated hashing is strongly universal]
**Linear Hashing.** The earliest universal hash functions described in the literature were also quite simple, although they require selection of an arbitrary (not necessarily random) prime number $p \geq C$ during initialization (we will see how to generate prime numbers in Chapter ??). If we now choose a random integer $a \in \{1, 2, \ldots, p - 1\}$, then

$$h(k) = (ak \mod p) \mod m,$$

is universal (but not strongly universal), with $\Pr[h(k_1) = h(k_2)] \leq 2/m$ for any two keys $k_1 \neq k_2$. The related function

$$h(k) = ((ak + b) \mod p) \mod m,$$

with $b$ randomly chosen from $\{0, \ldots, p - 1\}$, is strongly universal; we will prove shortly that $\Pr[h(k_1) = h(k_2)] \leq 1/m^2$ for any two keys $k_1 \neq k_2$. We call this the linear hashing method. Note that as opposed to multiplicative and tabulated hashing, linear hashing makes no assumptions about $m$ and $C$ being powers of two.

**Problem 108 (Universal Versus Fully-Random Hashing).** Recall from problem 19 that if we randomly map $n$ elements to a size-$n$ table, then with high probability each entry receives $O(\log n)$ elements ($O(\log n / \log \log n)$, with a more careful analysis). Strongly universal hashing gives us a much weaker guarantee.

(a) Please design a strongly-universal hash function for mapping $n$ elements into a size-$n$ table that always sends $\Omega(\sqrt{n})$ elements to some table entry. [Solution]

(b) Please show that with probability at least $1/2$, any universal hash function mapping $n$ elements into a size-$n$ table sends $O(\sqrt{n})$ elements to every entry. Hint: what is the expected total number of collisions? [Solution]

### 7.2.3 Higher Degrees of Independence

A hash function is strongly $r$-universal if given any set of $r$ different keys $k_1 \ldots k_r$ and any $r$ table locations $y_1 \ldots y_r$,

$$\Pr[h(k_i) = y_i \text{ for all } i \in \{1, \ldots, r\}] = O(1/m^r).$$

While strong universality (the special case where $r = 2$) gives a hash function with (asymptotic) pairwise independence, strong $r$-universal hashing gives (asymptotic) $r$-wise independence. We can easily generalize the linear hashing method above to create a strongly $r$-universal function by using a higher-degree polynomial:

$$h(k) = ((a_{r-1}k^{r-1} + \ldots + a_1k + a_0) \mod p) \mod m,$$

where $p \geq C$ is an arbitrary prime number, and each $a_i$ is chosen independently at random from $\{0, \ldots, p - 1\}$. [Proof of strong universality, also for the special case of linear hashing]

We will see another use of high-degree polynomial hash functions in a few pages when we use them to hash large objects.

**Probing Revisited.** It was only recently shown that 5-wise independence (e.g., using a degree-4 polynomial hash function) suffices to obtain $O(1)$ expected running
times for \textit{insert} and \textit{find} when we use linear probing, as long as we keep \( m \) at least some small constant factor larger than \( n \) [Somewhat complicated details]. This level of independence is necessary since some 4-wise independent hash functions can lead to \( \Omega(\log n) \) expected time guarantees.

\textbf{Problem 109 (Probing with Successive Universal Hash Functions).} Another probing approach that behaves well in theory uses multiple simple hash functions instead of one 5-wise independent function. Consider probing locations \( h_1(k), h_2(k), \ldots \) where the \( h_i \)'s are universal hash functions generated and stored on demand. Please show that this leads to an \( O(1) \) expected running time for \textit{insert}, and hence also for \textit{find} (as long as we keep \( m \) at least some small constant factor larger than \( n \)). Please also show that we only generate \( O(\log n) \) hash functions with high probability. \textbf{[Solution]}

\section*{7.3 Collision-Free (Perfect) Hashing}

With universal hashing, the probability of two different keys colliding is at most \( \frac{c}{m} \), for some constant \( c \). In a set of \( n \) keys, a union bound over all \( \binom{n}{2} \) pairs of keys gives

\[ \Pr[\text{any pair of keys collides}] \leq \frac{c}{m} \binom{n}{2} \leq \frac{cn^2}{2m}. \]

Setting the table size to \( m = cn^2 = \Theta(n^2) \), we therefore see zero collisions with probability at least \( 1/2 \).

A hash table supporting \textit{find} in \( O(1) \) worst-case time is called a \textit{perfect} hash table. According to the analysis above, we can build a static perfect hash table on \( n \) keys (not supporting further insertions after it is built) if we are willing to tolerate \( \Theta(n^2) \) space. To do this, we universally hash \( n \) elements into a table of size \( m = cn^2 \), stopping and repeating the entire process with a different randomly-chosen universal hash function if we find even a single collision. Since the probability of success is at least \( 1/2 \), we expect at most two such attempts.

The analysis above is essentially the same analysis we conducted for the famous “birthday paradox” (problem 17), which states that if everyone is randomly assigned one of \( m \) possible birthdays, then \( n \approx \sqrt{m} \) is the rough threshold for the size of a group of people where we start to see shared birthdays. In the context of (universal) hashing, this result says that for a hash table of size \( m \), we expect to start seeing our first collisions after inserting roughly \( n \approx \sqrt{m} \) elements.

\textbf{Achieving Linear Space.} A clever two-level hashing approach can reduce the space used by our static perfect hash table from \( \Theta(n^2) \) to \( \Theta(n) \). We first universally hash \( n \) elements into a size-\( n \) table. There will almost certainly be collisions, which we resolve by chaining for the moment. For each index \( i \) in our table (say, containing \( b_i \) elements), we then use the approach above to build a collision-free second-level universal hash table of size \( \Theta(b_i^2) \) on its elements in \( \Theta(b_i^2) \) expected time. As shown in Figure 7.3, we can now \textit{find} an element in in \( O(1) \) worst-case time with two hash lookups, one at the top level and the other in a second-level table.

It takes \( \Theta(n) \) space and time to build the top-level table, and \( \Theta(\sum_i b_i^2) \) space and expected time to build the second-level tables. Universality of the top-level hash table implies that \( E[\sum_i b_i^2] = \Theta(n) \) [Simple proof], so the entire process takes linear
expected space and time. It is easy to convert the space bound to $\Theta(n)$ in the worst case, preserving linear expected construction time, by re-selecting the parameters in our top-level function until $\sum_i b_i^2$ turns out sufficiently small. [Details]

**Problem 110 (Bucket Sort).** Suppose we are sorting $n$ integers $A[1\ldots n]$, each drawn independently from the uniform distribution $[0, C - 1]$, where $C$ is a multiple of $n$ (this is just for simplicity of analysis; it is not a fundamental restriction). Let us sort $A$ as follows: first create an array of $n$ buckets $B[0\ldots n - 1]$, and then “hash” each element $A[i]$ to the bucket $B[\lfloor nA[i]/C \rfloor]$ (so the first $1/n$ fraction of the values of $[0, C - 1]$ end up in bucket $B[0]$, the next $1/n$ fraction in $B[1]$, etc.). We then insertion sort the contents of each bucket, and enumerate the sorted contents of $B[0], B[1], \ldots, B[n - 1]$ in order to produce the sorted ordering of $A$. Using what you know about the analysis of static perfect hashing, please show that this bucket sort algorithm runs in $O(n)$ expected time. [Solution]

**Problem 111 (Hashing Trees).** The two-level hashing concept above generalizes nicely, leading to a “tree” of hash tables. Suppose we hash $n$ elements into a size $n$ table, resolving collisions with chaining for now. This table is the root of our tree. For each entry containing $k > 1$ elements, we then build a second-level hash table of size $k$. Collisions in second-level tables are resolved with third-level tables, and so on, until each leaf of the tree is a hash table with no collisions. Based on the results of problems 8 and 108, please prove that with strongly universal hashing, the depth of such a tree can be $\Omega(\log \log n)$,

![Figure 7.3: A static perfect hash table. Collisions in the primary universal hash function are resolved by hashing into collision-free secondary tables. If cell $i$ contains $b_i$ elements, then the second-level table attached to cell $i$ has size $c b_i^2 = \Theta(b_i^2)$; we assume $c = 1$ above.](image-url)
but that the expected depth is $O(\log \log n)$ under universal hashing. As a hint, use the randomized reduction lemma. How does this compare to the expected depth if we were to hypothetically use fully-random hashing? [Solution]

Double Displacement. There are several other nice ways to build a static perfect hash table on $n$ keys in $\Theta(n)$ expected time and $\Theta(n)$ space. One such method, called double displacement, has an elegant geometric interpretation where we place our keys on a grid, then shuffle the rows and columns of the grid so that each key ends up in a unique column (which is its hash). In addition to having a relatively simple analysis, this method can also be derandomized to obtain an $O(n \log n)$ worst-case construction time. [Full details]

7.3.1 Dynamic Perfect Hashing

So far, our perfect hash tables have all been static, not supporting the ability to add new elements after construction while maintaining an $O(1)$ worst-case running time for find. However, using these as black boxes, we can now build a linear-space dynamic perfect hashing structure supporting insert and delete in $O(1)$ expected amortized time$^8$ and find in $O(1)$ worst-case time.

Suppose we maintain $n$ elements in a universal hash table of size $m = \Theta(n)$. As shown in Figure 7.4, sets of colliding elements are stored in second-level static perfect hash tables, which are rebuilt from scratch any time they are changed by an insertion or deletion. Since each static perfect hash structure requires linear

\[^8\text{It can be somewhat confusing initially to see a running time that is both "expected" and "amortized". To clarify, an expected amortized time of }O(f(n))\text{ means that any sequence of }k\text{ operations should run in }O(kf(n))\text{ expected time.}\]
7.3. COLLISION-FREE (PERFECT) HASHING

![Diagram](image)

Figure 7.5: Examples of cuckoo hash tables. The single-table variant and its associated graph are shown in (a), and the two-table variant and its associated bipartite graph are shown in (b). An insertion of key $k_1$ in both cases could displace $k_2$ which in turn will displace $k_3$.

space, the entire two-level structure occupies $\Theta(n)$ total space. *Find* clearly takes $O(1)$ worst-case time. For *insert* and *delete*, we use the same analysis as with *find* from universal hashing: if $k$ elements collide in some entry of a chained universal hash table, then it takes $O(k)$ time to search them during a *find* operation, and universality implies that $k = O(1)$ in expectation for a generic *find* operation. In our new structure, if $k$ elements collide in some entry of our top-level universal hash table, then it takes $O(k)$ expected time to rebuild them into a second-level static perfect hash table; this is again $O(1)$ expected time for a generic *insert* or *delete* operation due to universality. Finally, we need to add an “amortized” quantifier to *insert* and *delete* since we periodically resize the top-level hash table to ensure $m = \Theta(n)$, just as we have done in the past.

7.3.2 Cuckoo Hashing

We can implement *Cuckoo hashing* — a particularly simple method for dynamic perfect hashing — with either one large table $T$ of size $\Theta(n)$ (usually just larger than $2n$) or two tables $T_1$ and $T_2$ both of size larger than $n$. We use amortized rebuilding to ensure the tables remain sufficiently large. Instead of a single hash function, we use two functions $h_1$ and $h_2$. In the single-table variant, an element of key $k$ will be found at one of two locations, either $T[h_1(k)]$ or $T[h_2(k)]$. In the two-table variant, the two locations are $T_1[h_1(k)]$ and $T_2[h_2(k)]$. It is now trivial to *find* an element in $O(1)$ worst-case time, since there are only two places to look for it. A nice way to visualize a Cuckoo hash table is as a graph (Figure 7.5), where each key corresponds to an edge between its two possible locations.
To insert an element with key $k$, we place the element at either of its two valid locations if one is empty. Otherwise, we claim one of these location (chosen arbitrarily), kicking out whatever element resides there. This explains the name “Cuckoo hashing”, since the European Cuckoo is a bird that takes over the nest of another bird. Since every element can live in one of two possible locations, the newly-evicted element now relocates itself to its alternate place of residence. This may in turn evict another element, and so on, until we finally reach an empty table entry or until we have evicted a chain of $L = \Theta(\log n)$ elements, in which case we give up and rebuild the entire structure from scratch with a new random choice of hash functions $h_1$ and $h_2$. Rebuilding make take several attempts, but only a constant number in expectation. When we study graphs in Chapter ??, we will show in problem ?? how to build a Cuckoo hash table on $n$ elements in $\Theta(n)$ time or deduce that this is impossible, given our choice of $h_1$ and $h_2$.

**Cuckoo Hashing Caveats.** One drawback of Cuckoo hashing is that its standard analysis requires the use of $L$-universal hash functions (i.e., $O(\log n)$-universal hash functions) to ensure the desired $O(1)$ expected amortized guarantees for insert and delete. While we showed earlier how to construct such functions, that particular construction involves a polynomial of degree $\Theta(\log n)$, and therefore takes $\Theta(\log n)$ time to evaluate. The only known constructions of $L$-universal hash functions that can be evaluated in $O(1)$ time are impractically complicated, and require more than linear space to represent. In practice, we therefore often use simpler hash functions, but we must do so with care. For example, it has recently been shown that the linear and multiplicative universal hash functions we defined earlier, as well as even some classes of functions with even higher levels of independence, can in some situations fail to allow any valid assignment of keys to table slots with high probability. Tabulation-based hashing has, however, been shown to work well both in theory and practice, so that may be the safest approach to use. Further elaboration on this and several other surprising properties of tabulated hashing can be found in the endnotes. [Standard analysis of Cuckoo hashing]

### 7.4 Radix Trees

Hash tables are extremely fast, but only support the fundamental dictionary operations insert, delete, and find. If we need any of the extended operations (e.g., pred, succ, find-min, find-max, select, and rank) supported by binary search trees and their relatives, we need to consider other types of integer search structures. As a trade-off for supporting a more robust set of operations, these will all have input-sensitive running times depending on $C$. We begin with the radix tree, introduced back in Section 5.5.1 as a fast way to implement a RAM priority queue. Radix trees also work well as general RAM dictionaries, supporting insert, delete, find and all of the extended operations above in $O(\log C)$ time.

As shown in Figure 7.6(a), the (binary) radix tree is a binary tree of height $\log C$. Elements are stored in leaves, where the root-to-leaf path to an element encodes $h_1(k) \neq h_2(k)$ for every key $k$ in the table. We could achieve this by rebuilding the table with a new random choice of $h_1$ and $h_2$ if we ever try to insert a key $k$ for which $h_1(k) = h_2(k)$, or also by setting $h_2(k) = (h_1(k) + g(k)) \mod m$, where $g(k)$ hashes to the range $1 \ldots m - 1$.

---

With the single-table variant, we must choose our hash functions carefully to ensure that $h_1(k) \neq h_2(k)$ for every key $k$ in the table. We could achieve this by rebuilding the table with a new random choice of $h_1$ and $h_2$ if we ever try to insert a key $k$ for which $h_1(k) = h_2(k)$, or also by setting $h_2(k) = (h_1(k) + g(k)) \mod m$, where $g(k)$ hashes to the range $1 \ldots m - 1$. 

---
the binary representation of its key — left edges are 0s and right edges are 1s. Operations on a radix tree are straightforward and quite similar to those in binary search trees, so we leave their details to the reader to fill in. We focus here on binary radix trees, although \( B \)-ary radix trees are also easy to construct; for example, see Section 5.5.1. We typically assume when dealing with binary radix trees that \( C \) is a power of two, so \( \log C \) is integer-valued.

Radix trees have many applications in practice. As an example, routers on the Internet often maintain a routing table containing entries like “All packets destined for IP addresses of the form 192.168.0.0/16 should be sent to output port X”. The notation 192.168.0.0/16 means all 32-bit IP addresses starting with the 2 bytes (16 bits) 192 and 168. In a radix tree of height 32 (if we are using 32-bit IP addresses), we would attach this particular routing entry to the interior node at depth 16 corresponding to the 16-bit representation of the two bytes 192 and 168, since it applies to all 32-bit addresses in the subtree rooted at that node. When a new packet arrives with destination IP address \( A \), we then walk from the leaf corresponding to \( A \) up to the root, using the first, and therefore most specific, rule we encounter along the way (for example, a rule for 192.168.0.0/16 should take precedence over a less-specific rule for 192.0.0.0/8).

---

**Figure 7.6**: Reducing the space required to store a radix tree in (a), using (b) path compression and (c) indirection. The search path to the element of key 1001 is shown in each case. In (c), the small nodes and dotted edges are not actually part of the tree; they are merely shown for reference to see what we have succeeded in removing from the tree.
7.4.1 Reducing Space: Path Contraction and Indirection

An \( n \)-element radix tree occupies \( O(n \log C) \) space since each element \( e \) requires the storage of \( \log C \) internal nodes along the path from the root down to \( e \). We can reduce this down to just \( \Theta(n) \) using two different techniques, path contraction and indirection, both of which are quite powerful and widely-applicable in the study of data structures.

**Path Contraction.** Suppose we remove all non-branching nodes, as shown in Figure 7.6(b), thereby contracting every path in our tree into a single “long” edge. Every such edge is now labeled with multi-digit binary identifier (stored in a single machine word\(^{10} \)) for the path it represents. Every internal node is now a branching node, and any such binary tree with \( n \) leaves has exactly \( n - 1 \) internal nodes (since each internal branching point adds one to the leaf count). Compressed trees of this form are sometimes called PATRICIA trees in the literature, where the acronym stands for “Practical Algorithm to Retrieve Information Coded in Alphanumeric”, the name of a 1968 paper on this subject by Donald R. Morrison. It is easy to modify the implementation of all radix tree operations so they work natively on a compressed tree with no performance penalty.

**Indirection.** Rather than storing individual elements in a data structure, suppose we instead store blocks of elements, with each block stored its own separate data structure. This technique — with a coarse-grained “high level” structure storing blocks of data encoded by “low level” structures — is known as indirection. Since a radix tree occupies \( O(n \log C) \) space, we can reduce this to \( \Theta(n) \) if we only store \( O(n/\log C) \) actual elements in the tree. To do this, we use indirection and store at the leaves of our radix tree blocks of between \( \log C \) and \( 2 \log C \) consecutive elements, each stored in a separate array, as shown in Figure 7.6(c). The minimum element in each block serves as a “canonical” element representing the entire block in the top-level radix tree. Since only the canonical elements are stored in the tree, the total space usage is now just \( \Theta(n) \) for the tree plus the blocks\(^{11} \).

Operations on our two-level structure now consist of two parts, one that interacts with the high-level radix tree, and the other interacting with a low-level block of elements. For example, \( \text{find}(k) \) calls \( \text{pred}(k) \) in the top-level radix tree to locate the block to which our element should belong, after which we search for \( k \) within this block. Both parts take only \( O(\log C) \) time. The \textit{insert} and \textit{delete} operations may occasionally need to split or join blocks, or to rotate elements between adjacent blocks (just like in a \textit{B}-tree) to maintain the \( \log C \ldots 2 \log C \) size constraint on the blocks, but this still only requires \( O(\log C) \) time, since splitting, joining, and rotation all cause at most two operations in the high-level radix tree.

---

\(^{10}\)When we store a collection of variable-length binary numbers, it can be hard to tell the difference between numbers like 0110 and 110, since both of them end up stored as the number 6 in a machine word. For this reason, we typically also store the number of digits in the number, or alternatively we might prepend a dummy 1 bit to the front of each number so leading zero bits are no longer a problem (note that this encoding scheme labels the nodes of a radix tree the same way we index the nodes in a binary heap).

\(^{11}\)If \( n < \log C \), we need to be slightly careful since to store anything at all in the leaves of a radix tree requires \( \Theta(\log C) \) space, which is worse than \( \Theta(n) \) space. However, in this case our elements fit into a single block, so there is no need for the radix tree yet. Only once we reach \( n \geq \log C \) will we bring the full radix tree into the picture.
7.4. The Y-Fast Tree

The standard radix tree does not win on performance against a balanced binary search tree, since $C \geq n$ if all our keys are distinct (a typical assumption), and therefore $O(\log C)$ is no better than the $O(\log n)$ running time per operation provided by the balanced binary search tree. However, we can do several things to speed up the radix tree:

- We can store each element simultaneously in a universal hash table to make $\text{find}$ run in $O(1)$ expected time.
- We can augment each tree node with a direct pointer to its subtree minimum and maximum, improving $\text{find-min}$ and $\text{find-max}$ to run in $O(1)$ time.
- For the operations $\text{pred}$ and $\text{succ}$, recall these have two variants: $\text{pred}(e)$ and $\text{succ}(e)$ find the predecessor and successor of an element $e$, and $\text{pred}(k)$ and $\text{succ}(k)$ find the next-smallest or next-largest element given a key value $k$. We can speed up the first variant to run in $O(1)$ time by maintaining a pointer directly from each element to its predecessor and successor.

The augmentations above can easily be maintained without sacrificing the asymptotic $O(\log C)$ running time of $\text{insert}$ or $\text{delete}$.

The only remaining “slow” operations taking $O(\log C)$ time are $\text{insert}$, $\text{delete}$, $\text{pred}(k)$, $\text{succ}(k)$, $\text{select}$, and $\text{rank}$. We now speed these up (except for $\text{select}$ and $\text{rank}$) to run in only $O(\log \log C)$ time. This gives us a RAM dictionary whose operations run in $O(\log \log C)$ time (except for $\text{select}$ and $\text{rank}$), a range query structure capable of finding all $k$ elements in some range $[a, b]$ in $O(k + \log \log C)$ time, a RAM priority queue whose operations run in $O(\log \log C)$ time, and a RAM sorting algorithm running in $O(n \log \log C)$ time. Depending on implementation, some of these bounds will be amortized and/or in expectation.
The structure we now describe is called a \textit{Y-fast} tree (its name comes from a sequence of historical data structures called P-fast trees, Q-fast trees, and X-fast trees). Following this, we discuss an alternative data structure called a \textit{stratified tree} or a \textit{van Emde Boas (vEB) structure}, which is actually equivalent to the Y-fast tree but provides another useful perspective.

The Y-fast tree is a radix tree that uses hashing to quickly implement $\text{pred}(k)$ and $\text{succ}(k)$. We begin by augmenting our tree so that every node maintains pointers to the minimum and maximum elements (leaves) in its subtree, and every element (leaf) is augmented with a pointer to its predecessor and successor. Additionally, we build a universal hash table containing the IDs of all nodes in the tree. As shown in Figure 7.7, a node ID is the binary string (stored as an integer in a single machine word) corresponding to the path from the root down to that node. Recall that we may need to prepend a dummy 1 bit to each node ID or store also the number of bits in each node ID, since otherwise we might have trouble distinguishing IDs like 011 and 0011 with leading zeros.

Consider now the $\text{pred}(k)$ and $\text{succ}(k)$ operations. If $k$ is present in the data structure (which we can determine in $O(1)$ expected time by calling $\text{find}(k)$), we are done. Otherwise, we quickly locate its lowest present ancestor (LPA), shown in Figure 7.7. If LPA($k$) has a right child, then $\text{succ}(k)$ is the minimum in LPA($k$)'s right subtree (from which we can follow a predecessor link to determine the predecessor of $k$). Similarly, if LPA($k$) has a left child, then $\text{pred}(k)$ is the maximum in LPA($k$)'s left subtree. The only challenge remaining is to compute LPA($k$), whose node ID is the longest prefix of $k$'s log $C$-bit binary representation that exists in our node ID hash table. We can therefore binary search over prefix lengths in $O(\log \log C)$ expected time. That is, we initially check if the first half of $k$'s binary representation is the ID of some node present in our hash table. If yes, we try a prefix of $3/4$ of $k$'s binary length. If not, we try a prefix length of $1/4$ of $k$’s binary length, and so on.

We can improve the running time of insert and delete to $O(\log \log C)$ amortized by using indirection. Suppose we group consecutive elements stored in the leaves of a radix tree into blocks each containing $\Theta(\log C)$ elements as before. If we store each block in a small balanced binary search tree, then operations within a block take only $O(\log \log C)$ time. Insertions and deletions in the top-level radix tree still take $O(\log C)$ time, but these only need to happen when blocks are subject to splits, joins, or rotations, which happen infrequently enough to give us our desired amortized bounds. As an added benefit, indirection also reduces the space required to store the Y-fast tree to $\Theta(n)$. [Complete details]

### 7.4.3 Stratified / van Emde Boas (vEB) Trees

The final data structure we discuss in this section is an elegant result due to Peter van Emde Boas, sometimes called a \textit{stratified tree}, that “vertically” decomposes a tree in a recursive fashion. A radix tree of height $h = \log C$ has room for $C = 2^h$ elements at its leaves, so a radix tree of height $h/2$ has at most $\sqrt{C} = 2^{h/2}$ leaves. Therefore, by splitting a radix tree in half height-wise (Figure 7.8(a)), we get a top-level radix tree with up to $\sqrt{C}$ leaves, and each leaf in this structure is the root of a low-level radix tree having up to $\sqrt{C}$ leaves. After dividing into half-height trees, we then recursively subdivide these into quarter-height trees, and so on.
Although the original van emde Boas (vEB) structure is not described in terms of radix trees, we will describe it in a way that highlights its equivalence with the Y-fast tree. We start with a radix tree, augmented as before so that every node maintains a pointer to the minimum and maximum element (leaf) in its subtree, and every element (leaf) maintains a pointer to its predecessor and successor. We also maintain a universal hash table of node IDs. Recall that we can answer \( \text{pred}(k) \) and \( \text{succ}(k) \) queries in this structure in \( O(1) \) time once we have located LPA(\( k \)). In the Y-fast tree, we do this in \( O(\log \log C) \) time by binary searching over prefixes of the \( \log C \)-bit binary representation of \( k \). The same process has a nice interpretation in terms of our stratified tree. Let us write the binary representation of \( k \) as \( k_hk_l \), where \( k_h \) contains the “high” half of \( k \)’s bits, and \( k_l \) contains the “low” half. For example, if \( k = 10111100 \) in binary, then \( k_h = 1011 \) and \( k_l = 1100 \). We now perform a single hash lookup to check in \( O(1) \) expected time whether a node with ID \( k_h \) is present in the “middle row” of our radix tree, as shown in Figure 7.8(b). If so, we recursively search the low-level tree rooted at this node for LPA(\( k_l \)) — note that all nodes in this tree all agree in the high-order half of their IDs, so we are effectively now dealing with just the low-order halves of all node IDs. Otherwise, we recursively search the top-level tree for LPA(\( k_h \)) — in this case, we know that the low-order bits of LPA(\( k \)) are all zero, and we are effectively only dealing with the high-order halves of all node IDs. Each step narrows our search to a tree of half its original height (or equivalently, to a binary string of half its original length), so the entire process locates LPA(\( k \)) in only \( O(\log \log C) \) expected time. Moreover, it is easy to see the equivalence between this recursive search in the stratified tree and the binary search over prefixes in the Y-fast tree.

Figure 7.8: Illustration of (a) one level of the recursive decomposition of a stratified tree into a top-level tree \( T \) and low-level trees \( L_i \), and (b) the same recursive structure overlaid on a radix tree to build a vEB structure.
Problem 112 (Cache-Oblivious BST Layout). The stratified tree decomposition has other useful applications other than speeding up a radix tree. Suppose we want to store a large static $n$-element dictionary in an external memory that supports block reads of size-$S$ blocks. Here, $\Theta(\log_S n)$ block reads are necessary in the worst case to find an element based on its key, and a $B$-tree with $B = \Theta(S)$, achieves this asymptotic bound. Hardware parameters are often hard to know, however, so we might like to design a cache-oblivious data structure (see Section 1.7.5) for which the number of block reads is always $O(\log_S n)$ in the worst case even if we do not know $S$. Let us take a $B$-tree with $B = \sqrt{n}$. We then store the contents of each node ($\Theta(\sqrt{n})$ elements for each non-root node) recursively in another $B$-tree, this time with $B = \sqrt[4]{n}$, and so on. Please show that this is essentially performing the stratified decomposition above, and also that this structure does indeed require at most $O(\log_S n)$ block reads for find, irrespective of the block $S$. This structure is in some sense the cache-oblivious way to perform binary search. [Solution]

7.5 Further Hashing Applications

Hashing goes well beyond just data structures. Its central idea of mapping a complex object down to a simpler representation has far-ranging application in many computing subfields. For example, in machine learning, we often represent complicated objects using low-dimensional feature vectors. After “hashing” a person down to just the vector (height, weight), for instance, we might still be able to predict gender with reasonable accuracy. In this example and others, some might argue that the term “hashing” is less appropriate, since our mapping lacks the intentionally haphazard nature we often strive to achieve with most hash functions. Nonetheless, to “hash” is sometimes spoken more generally to describe any mapping from large and complex to small and simple. In this section, we highlight other prominent instances of this general idea that have substantial algorithmic impact.

7.5.1 Pseudorandom Number Generation

Hash functions are usually designed to behave “randomly”, so they are ideal for generating pseudorandom numbers. To generate $C$ pseudorandom numbers in the range $\{0, \ldots, m - 1\}$, we evaluate $h(0) \ldots h(C - 1)$. For pseudorandom numbers that are pairwise independent or higher, we can use a hash function satisfying this condition. For example, linear strongly universal hashing satisfies pairwise independence. Sometimes this is viewed as “randomness amplification”, using a small number of truly-random parameters baked into a hash function to produce a much larger sequence of numbers having a more limited degree of independence.

As a nice example, consider the problem of estimating the number of distinct elements, $k$, appearing in a massive data stream of length $n$. We would normally solve this problem by storing the distinct stream elements in a hash table of size $\Theta(k)$, except here $k$ can be much larger than the size of our memory. Suppose the distinct elements in our stream are themselves integers distributed uniformly at random. In this case, roughly half will be multiples of two (ending with 0 in binary), one quarter will be multiples of four (ending with 00 in binary), and so on. If we therefore take the maximum number of 0s we see at the end of any number (when written in
7.5. FURTHER HASHING APPLICATIONS

binary), this gives a reasonable estimate of \( \log k \). Unfortunately, stream elements may not be randomly distributed, but we can achieve the same effect by looking at hashes of the elements. That is, we map the elements in our stream through a hash function, effectively making the stream look “random” but preserving the relative frequencies of its elements. By aggregating several independent estimates (see, e.g., problem 32) made in parallel as the stream goes by, we can ultimately refine this technique to produce a final estimate that with high probability is accurate to within constant relative error, using only \( \Theta(\log n) \) total space. [Full details]

7.5.2 Load Balancing and Consistent Hashing

Large websites often operate from a bank of servers, allowing them to handle high load and also provide fault tolerance if servers fail. To assign incoming packets to servers in a balanced fashion, we could use “round robin” assignment (cycling through the list of servers), or even simpler, we could just assign each packet to a random server. However, these approaches do not provide consistency of assignment. Packets from the same source IP address might be mapped to different servers over time, and this is undesirable since servers often maintain some amount of state with respect to each of their active connections (e.g., a “shopping cart”, for an on-line store). By hashing packets to servers based on source IP address, the random nature of our hash function gives the same load balancing we would expect from a random assignment, with the added benefit of consistency.

With most common hash functions, changing the size of a hash table can dramatically change where most elements are mapped. This can be problematic here if servers are removed or added — say, if a server becomes unresponsive or unreachable for a short period of time. When a server goes offline, we should ideally re-distribute only its load temporarily among the other servers, leaving the assignments of all other packets unchanged. Again, we want a hashing scheme that provides a measure of consistency in response to changes in hash table size. We can achieve consistent hashing of this sort in several ways. Often, instead of hashing packets to servers, we instead think of hashing both packets and servers to a third space, where proximity between packets and servers determines our assignment\(^{12}\). For example, in Figure 7.9(a), we have mapped packets and servers to an address space that logically wraps around in a circle, where each packet is assigned to the next server following it in clockwise order. As stated, this is not an ideal solution, since removal of a server causes all of its traffic to fail over to the next server in sequence, potentially overloading it. We can do better by considering each server to be a collection of virtual servers, all mapped to different locations on the circle. Temporary removal of all of these virtual instances for a failing server causes that server’s load to be spread more uniformly across the other servers.

7.5.3 Distributed Hash Tables

It is now common to find data sets far larger than can fit within the memory of a single machine, making it necessary to distribute the data across a network of servers. This can be achieved using distributed hash tables (DHTs), which are a way to store and retrieve data in a distributed system. DHTs are designed to be scalable and robust, able to handle failures and load imbalances. They achieve this by using a combination of hashing, replication, and routing to distribute the data across the network of servers. DHTs are commonly used in peer-to-peer networks, where no central server is responsible for the data storage and retrieval. The idea of mapping a server to “network coordinates” in a virtual geometric space is common in the networking literature, as it can be useful for problems such as routing.

\(^{12}\)The idea of mapping a server to “network coordinates” in a virtual geometric space is common in the networking literature, as it can be useful for problems such as routing.
servers. A distributed hash table (DHT) is a distributed data structure behaving like a large virtual hash table, where calls to insert, delete, and find can be issued from any participating server. Well-designed DHTs gracefully handle server additions and removals, they should have no single point of failure (say, a special “root” server that must be consulted for every query), and they should not require servers to have global knowledge of the entire network, since this would scale poorly.

The circular hashing scheme described above for load balancing leads to one popular approach for implementing a DHT, known as the “Chord” DHT, since servers maintain links across the circle that resemble chords in geometry. Servers and elements of data are all hashed to a common circular address space, with each element stored in a hash table on the server immediately following it in clockwise order. As before, each server is often split into several virtual servers so that its load can be more uniformly redistributed in the event of failure. Similarly, each element may also be hashed to several locations on the circle (and hence stored on several machines), so that isolated server failures don’t destroy all existing copies of the element.

The main challenge here is routing. No server knows the global list of all other servers, so it must relay insert, delete, and find requests to the small handful of other servers it does know, so these ultimately reach their intended destinations. By maintaining a short list of servers immediately preceding and following it on the circle, each server can forward requests around the circle, but the number of hops involved could be quite large. To improve performance, we borrow inspiration from the skip list: as shown in Figure 7.9(b), each server also maintains a list of servers at exponentially-increasing distances around the circle (e.g., roughly 1, 2, 4, 8, etc., hops away), thereby reducing the number of hops from linear to logarithmic until a request reaches its destination. Through periodic communication, each server keeps its links up-to-date in response to server additions and removals.

Figure 7.9: Consistent hashing around a circle is shown in (a), where removal of all instances of server 1 causes its traffic to be temporarily reassigned to the other servers in a uniform fashion. The long-distance links from a single server in the “Chord” distributed hash table are shown in (b).
7.5. FURTHER HASHING APPLICATIONS

7.5.4 Problem Decomposition, MapReduce

Taking “divide and conquer” to the next level, problems can sometimes be broken down (i.e., “hashed”) into very large numbers of subproblems that can be processed in a nearly independent fashion. For example, if we run an quadratic-time algorithm on \( m \) subproblems of uniform size \( n/m \), this gives a total running time of \( m \times O((n/m)^2) = O(n^2/m) \), a speedup of \( m \) compared to running the algorithm without partitioning. Bucket sort (problem 110) is an excellent example of precisely this sort of speedup. As another simple example, suppose we are searching for words in a dictionary that are anagrams of each-other — containing the same count of each letter, or equivalently, words that become identical when their letters are sorted. If we apply a hash function to the sorted contents of each word (e.g., using polynomial hashing from the next section), we ensure that anagrams will collide. We can now afford to apply a slower (say, quadratic time) method for partitioning into anagrams within each set of colliding words, being far smaller than the original dictionary.

Problem 113 (Finding Near Neighbors). The approach above is common in geometric algorithms, where we “spatially hash” a set of objects in order to partition them by location, after which we further process each set of nearby objects. To give a nice example, recall that we have already seen how to solve the element uniqueness problem (determine if any two elements in an array \( A[1...n] \) are equal) in \( \Theta(n) \) expected time using hashing. Consider now asking whether two array elements have values within some specified distance \( k \) of each-other.

(a) Please solve this problem in \( \Theta(n) \) expected time and \( \Theta(n) \) space. [Solution]
(b) Given \( n \) points in the plane \((x_1,y_1),...,(x_n,y_n)\) with integer coordinates, show how to find two points within distance \( k \) of each-other (or determine that no such pair exists) in \( \Theta(n) \) expected time and \( \Theta(n) \) space. [Solution]

Problem 114 (Sampling from a Discrete Distribution). Following the same theme of decomposing a problem into smaller pieces that are much more efficient to process, consider the following problem: given a probability distribution \( p_1...p_n \) over \( n \) elements (where \( \sum p_i = 1 \)), we would like to process the distribution in \( \Theta(n) \) time so that we can sample an element (according to its associated probability) in \( O(1) \) time. As a hint, map the elements into “buckets” each representing \( 1/n \) of probability space, so that after a bucket is chosen with uniform probability, it only takes \( O(1) \) additional time to select an element from that bucket. If you like, you can assume the probabilities have been rescaled to integers, in order to avoid real-valued arithmetic. [Solution]

Large-scale problem decomposition can be particularly effective when applied to truly massive data sets. In this case, we often spread the work of problem decomposition across a distributed network of processors, each responsible for decomposing a small part of the input. Similarly, we often distribute the work of solving the resulting subproblems, with each processor being assigned only a small range of subproblems. As part of an increasingly popular framework called MapReduce, the first stage is sometimes known as mapping, and the second as reducing. MapReduce is now built into several widespread programming environments, its popularity stemming from the way it simplifies the implementation of distributed algorithms in the style above. All one needs to do is implement two functions, one for mapping and one for reducing. The framework handles the remaining technical details for instantiating these in parallel, and for routing the output of the mappers to the
appropriate reducers — a task perfectly suited for a distributed hash table, where each parcel of data generated by a mapper is keyed based on the subproblem to which it belongs (i.e., the reducer to which it should be routed).

As a simple example, we could use the anagram detection problem above. Suppose we have a large collection of text phrases stored on a distributed network of machines, in which we would like to identify anagrams. Each mapping task would take a collection of phrases and hash them in a way that ensures collision among anagrams. Each reducing task would then process all the phrases hashing to a particular value, looking for anagrams. We can tune the number of reducing tasks by changing the range of output values for our hash function.

7.5.5 Fingerprinting Large Objects

Many applications involve hashing a large object (e.g., a web page or file on disk) into a single integer, often called a fingerprint. We can test equality between two objects very quickly by just comparing their fingerprints. If the fingerprints differ, the original objects must differ. If the fingerprints match, then we strongly suspect the objects are the same. It is possible we could be seeing a hash collision between two different objects, but for an appropriate choice of hash function mapping to a sufficiently large range of fingerprints, we can decrease the probability of such false positives so they are essentially negligible. As a prototypical application, a web search engine might want to remove duplicate pages from its database. After fingerprinting, this problem reduces to removing duplicates from a large set of integers, a problem easily solved using a standard hash table.

We can regard any large object as just an integer array $A[0 \ldots n-1]$, with each $A[i] \in \{0, \ldots, C-1\}$. Perhaps the simplest and most common way to hash $A$ down to a single integer small enough to fit in a machine word is with a polynomial hash function, where we take the array elements to be coefficients of a polynomial


and then set $h(A) = A(x) \mod p$, where $p \geq C$ is an arbitrary prime (say, small enough to fit into a machine word), and $x$ is randomly chosen from $\{0, \ldots, p−1\}$ during initialization. As in other hashing schemes, we use arithmetic modulo $p$ to keep our numbers from growing too large.

If $A$ and $A'$ are two different length-$n$ arrays, then $\Pr[h(A) = h(A')] \leq \frac{n-1}{p}$, so we can reduce this collision probability to a miniscule level by making $p$ sufficiently large. For example, we avoid collisions with high probability by choosing $p \geq n^c$ with $c$ constant. The $\frac{n-1}{p}$ collision probability follows from the fact that two different polynomials $A(x)$ and $A'(x)$ of degree $n-1$ can satisfy $A(x) \equiv A'(x) \pmod{p}$ for at most $n-1$ of the $p$ possible choices of $x$. Such a value of $x$ would be a root of the difference polynomial $A(x) − A'(x)$, and a nonzero polynomial of degree $n-1$ like this can have at most $n-1$ roots, even modulo a prime $p$.

We can easily compute $h(A)$ in only $O(n)$ time using Horner’s rule (Section ??): start with $A[n−1]$, multiply by $x$, then add $A[n−2]$, then multiply by $x$ again, then add $A[n−3]$, and so on, ending with the addition of $A[0]$. All the while, we keep reducing the result modulo $p$ so it always fits within a machine word. Moreover,
after precomputing a table of \( x^i \mod p \) for \( i \in \{-1, \ldots, n-1\} \)\(^{13}\), we can update \( h(A) \) in only \( O(1) \) time after changing \( A \) by modifying one of its elements, or by adding or removing an element at one of its endpoints. This will be particularly useful in Chapter 9 when we use hashing to solve string matching problems.

**Problem 115 (Alternative Methods for Hashing Large Objects).** This problem highlights two other common ways to hash an array \( A[0 \ldots n-1] \), with each \( A[i] \in \{0, \ldots, C-1\} \), down to a small integer in \( O(n) \) time.

(a) **Dot Product with a Random Vector.** Regard \( A \) as a length-\( n \) vector, and let \( h(A) = A \cdot X \mod p = (A[0]X[0] + \ldots + A[n-1]X[n-1]) \mod p \), where \( p \geq C \) is a prime number, and \( X[0] \ldots X[n-1] \) are each chosen independently from \( \{0, \ldots, p-1\} \) during preprocessing. Please show that for any two length-\( n \) arrays \( A \neq A' \), we have \( \Pr[h(A) = h(A')] = 1/p \), so we again avoid collisions with high probability as long as \( p \) is sufficiently large. For a hint at the solution, refer to problem 38, which uses essentially the same underlying mathematics. [Solution]

(b) **Reduce Modulo a Random Prime.** Regard \( A \) as a large integer whose digits are \( A[n-1] \ldots A[0] \) when written in base \( C \). In terms of the polynomial \( A(x) \), this number is precisely \( A(C) \). Let \( h(A) = A(C) \mod p \), where \( p \) is a prime number chosen randomly from a set of \( r \) different alternatives during preprocessing. Please show that for any two arrays \( A \neq A' \), we have \( \Pr[h(A) = h(A')] \leq \frac{n}{r \log C} \), so yet again we avoid collisions with high probability as long as \( r \) is sufficiently large. In Chapter ??, we will learn how to generate random prime numbers. [Solution]

**Problem 116 (Detecting Infinite Loops).** If we are watching the execution of a computer program, we know it has entered an infinite loop if the contents of its memory ever becomes identical to the memory contents at some point in the past. Using the trick from problem 1, please show how to instrument any computer program so that we can detect infinite looping (with high probability), using asymptotically the same time and space as the original program. Assume that the number of steps we run our program is sufficiently small so as to fit into a single machine word. [Solution]

### 7.5.6 Security and Data Integrity

Hashing has extensive applications in the domain of computer security. A system administrator may keep a record of the fingerprints of important files on a machine, which can be used later to detect tampering. Fingerprints are also used to certify the integrity of important digital records, often in combination with other cryptographic methods such as digital signatures (discussed in Section ??). For example, by including the hash of the contents of a packet as a short “checksum” before sending it across a network, we can detect corruption during transit. We can also vastly improve the security of a computer system by storing passwords as fingerprints rather than “in the clear”. It is still easy to verify a user’s password in this setting.

Hash functions are often hard to “invert” (i.e., it can be difficult to find a key that hashes to some specific given value), and those designed for cryptographic or security purposes are specifically tailored to prevent attempts at inversion, to make it hard to falsify data or reverse engineer passwords from their hashes. The standardized **Secure Hash Algorithm** (SHA-1 and SHA-2) family of hash functions are the two

\(^{13}\)In Chapter ??, we will see how to compute the multiplicative inverse \( x^{-1} \) modulo \( p \) efficiently using Euclid’s algorithm.
most popular such functions in practice, producing as output fingerprints ranging from 160 bits (for SHA-1) up through 512 bits (for the longest variant of SHA-2). To find a collision by brute force for, say, SHA-1, one would need to check roughly $2^{80}$ different keys according to the birthday paradox\textsuperscript{14}. To date, no colliding pairs of keys for any of these hash functions have ever been discovered.

### 7.5.7 Bloom Filters and Sketching

The *Bloom filter*, named after its creator, Burton Bloom, provides a very compact representation of a set that trades space for accuracy of membership queries. Calling \texttt{find}(\texttt{k}) simply returns “yes” or “no”, and additional satellite data cannot be associated with keys. There is a chance \texttt{find}(\texttt{k}) might return “yes” even if \texttt{k} is not present, although the probability of these false positives can be set arbitrarily low at the expense of extra query time or space. The structure supports insertion but not deletion of keys, at least not in its most basic incarnation.

A Bloom filter starts as a zero-filled binary array $A[1 \ldots m]$ where $m > n$, along with $c$ independent hash functions $h_1 \ldots h_c$. Insertion of a key $k$ sets $A[h_1(k)] \ldots A[h_c(k)]$ all to 1, and \texttt{find}(\texttt{k}) returns true only if $A[h_1(k)] \ldots A[h_c(k)]$ are all set to 1, as shown in Figure 7.10. Both operations run in $O(c)$ time. Clearly, \texttt{find}(\texttt{k}) always succeeds if \texttt{k} has been inserted into the structure. If \texttt{k} has not been inserted, then \texttt{find}(\texttt{k}) can produce a false positive result if $A[h_1(k)] \ldots A[h_c(k)]$ all happen to be 1, but we can reduce the probability of this by making $c$ larger (sacrificing speed) or making $m$ larger (sacrificing space). Making $c$ too large, however, can cause the false positive rate to climb again, since this fills the structure with too many 1s. [Analysis of Bloom filters]

An adequately functioning Bloom filter requires linear space: $m > n$ bits for storing $n$ elements. Its underlying idea, however, motivates the design of several related

\textsuperscript{14}One might actually worry about using $2^{80}$ memory even more than taking $2^{80}$ units of time. That is, it may seem necessary to store each hash we check in a hash table so the hash of every new random key we consider can be quickly compared to those generated earlier. However, if we use our hash function to generate a pseudorandom sequence of keys $x_0, x_1, x_2, \ldots$, where $x_i = h(x_{i-1})$, then this sequence behaves like an implicitly-defined linked list, ending with a loop whose initial element tells us the hash code generated by $h(x_i)$ and $h(x_j)$ for $x_i \neq x_j$. We can find this using linear time but only constant memory with the result of problem 1.
randomized “sketch” data structures that can summarize the contents of a massive data stream using only logarithmic space. For example, we showed earlier how to estimate the number of distinct elements appearing in a length-$n$ stream with only logarithmic space. Here, we discuss two elegant structures for estimating frequencies (occurrence counts) of values in a length-$n$ stream, also using just logarithmic space. These are often used to identify elements occurring with high frequency, sometimes known as “heavy hitters”. As a common example, a router might want to inspect a stream of data packets to learn which IP source or destination addresses account for the most traffic.

As shown in Figure 7.11, both the count-min sketch and count sketch involve $c = \Theta(\log n)$ arrays $A_1 \ldots A_c$ of counters, each array having length $O(1/\varepsilon)$. We create hash functions $h_1 \ldots h_c$ indexing into each array. For every successive value $v$ we observe in the data stream as it passes by, we call update($v$), which modifies the counter at index $h_i(v)$ in each array $A_i$. For count-min sketch, the update is just an increment. For count sketch, we randomly choose between an increment or a decrement, but using hashing to do this in a consistent way: we define hash functions $g_1 \ldots g_c$ mapping stream values to $\{-1, +1\}$, and add $g_i(v)$ to $A_i[h_i(v)]$.

An estimate of the frequency $f_v$ of any value $v$ can be obtained at any time by calling query($v$). For count-min sketch, this returns the minimum of the values at $A_i[h_i(v)]$ across all arrays $i = 1 \ldots c$. This will always be an over-estimate, since $f_v$ has already been added into each of these counters. It is likely, however, that at least one of these counters has not been “contaminated” by too much additional weight from other values, so the minimum gives an estimate in the range $[f_v, f_v + \varepsilon n]$ with high probability [Detailed analysis]. With count sketch, observe that value $v$ contributes either $+f_v$ to the counter $A_i[h_i(v)]$ (if $g_i(v) = +1$) or $-f_v$ (if $g_i(v) = -1$). Hence, $A_i[h_i(v)]g_i(v)$ is a good estimator for $f_v$. Indeed, $\mathbb{E}[A_i[h_i(v)]g_i(v)] = f_v$, since value $v$ contributes $f_v$ and all other values contribute zero in expectation, with their mappings in $g$ being equally likely to be positive or negative. We improve our final estimate by taking the median of these estimates across all arrays $i = 1 \ldots c$. Count sketch gives an estimate in $[f_v - \varepsilon n, f_v + \varepsilon n]$ with high probability. Otherwise written, this error bound is $\pm \varepsilon ||f||_1$, where $f$ is the vector of frequencies for all values (since $||f||_1 = n$). An alternate bound we can show – stronger for more uniform frequency distributions – is $\pm \varepsilon ||f||_2$, although this requires each counter array to have $O(1/\varepsilon^2)$ size. [Detailed analysis]
With the count-min sketch, if all we want to do is estimate whether \( v \) appears in the data stream \( (f_v \geq 1) \), it suffices to just set the counters \( A_i[h_i(v)] \) to 1 instead of incrementing them during an update. Observe that this leads to what is essentially a multi-array variant of the Bloom filter, so we might think of the Bloom filter as a special case of the count-min sketch.

Problem 117 (Deterministic Algorithms to Find Frequent Elements). A majority element in a data stream \( x_1 \ldots x_n \) is an element with frequency strictly more than \( n/2 \). If you are told that a stream contains a majority element, it can be identified with a simple algorithm taking linear time and just constant space, making only a single pass through the stream. The algorithm keeps track of two values, a value \( v \) (initialized to null) and a count \( c \) (initialized to zero). For each stream element \( x_i \) it examines in sequence, it increments \( c \) if \( x_i = v \), and decrements \( c \) if \( x_i \neq v \); if \( c \) drops to zero, we replace \( v \) with \( x_i \) and set \( c = 1 \). If you are unsure if the stream contains a majority element, we can run two passes, the first producing some output \( x \), and the second checking if \( x \) is indeed a majority element.

(a) Please show that if there is a majority element, then this element will reside in \( v \) at termination. [Solution]

(b) A straightforward generalization of the algorithm above can find elements occurring with frequency exceeding \( n/k \) using only \( \Theta(k) \) space (so the approach above is the special case where \( k = 2 \)). We maintain \( k-1 \) pairs \((v_1, c_1) \ldots (v_{k-1}, c_{k-1})\) similar to the (value, counter) pair above. For each stream element \( x_i \) we inspect, if there is some pair \((v_j, c_j)\) with \( v_j = x_i \), we increment \( c_j \). Otherwise, we decrement all of the counters \( c_1 \ldots c_{k-1} \), and if one of these drops to zero, we select any such pair and replace it with \((x_i, 1)\). Please argue that any stream element with frequency exceeding \( n/k \) must end up as one of the \( v_i \)'s at termination (so in particular, we could verify whether these \( k-1 \) elements are indeed high-frequency with a second pass over the stream). [Solution]

7.5.8 Compressive Sensing and its Relatives

The process of mapping a large data stream \( x_1 \ldots x_n \) to a shorter sketch \( A[1 \ldots m] \) inherently loses information, and hence we can usually only extract limited information about \( x \) from the sketch. However, in some cases — for instance if we know most of the mass in \( x \) is concentrated in only a few of its elements — we can surprisingly “invert” \( A \) to obtain a good estimate of \( x \). Here, we usually consider linear sketches, where each element of the sketch \( A[j] \) is a linear combination of elements in \( x \). This process is known as compressive sensing (also compressed sensing) since it seeks to recover a large signal from a compressed representation. It has many potential applications, since real-world signals (e.g., audio, image data) are often quite sparse when viewed in the right “basis”. For example, as we will see in Chapter ??, a photograph may contain only a few significant frequency components when we look at its Fourier transform. Given a signal that is sparse in some basis, we would like to approximately reconstruct it from a small number of samples from its linear projection into some other basis (i.e., our sketch) in which its sparsity may not be so apparent. Hashing is one of several techniques that can be used for “sparse recovery” problems of this sort.

For a simple example, suppose each \( A[j] \) is the sum of a random subset of elements in \( x \), given by a hash function \( g_j \) mapping \( i \in \{1, \ldots, n\} \) to \{0, 1\}, so \( A[j] = \sum_i x_ig_j(i) \).
Here, a sketch of size \( m = O(k \log n) \) is all we need to exactly reconstruct with high probability any length-\( n \) stream \( x \) that is \( k \)-sparse — having \( \|x\|_0 \leq k \), where \( \|x\|_0 \) is the number of nonzero entries\(^{15} \) in \( x \). The intuition here is similar to the count-min sketch: each of the \( k \) nonzero entries in \( x \) likely appears in at least one location of the sketch uncontaminated by other entries in \( x \). \[ \text{[Full details]} \]

**Problem 118 (Problems Related to Sparse Recovery).** Here, we investigate two other related problems that can be approached with techniques very similar to those used in the example above.

(a) **Group Testing.** Suppose up to \( k \) of a set of \( n \) individuals has a particular disease. By pooling blood samples, you can test any group of individuals in a single step, but if the test comes back positive, all you know is that one or more members of the group have the disease. We would like to identify the diseased individuals with a minimum number of tests. In problem 41, we showed how to use only \( O(k \log n) \) tests in an “adaptive” fashion, where each test can depend on the results of previous tests. We can achieve the same bound in a non-adaptive setting, however. Let the \( m = O(k \log n) \) groups being tested be determined by \( m \) random subsets as above. In this case, our sketch \( A[1 \ldots m] \) is defined identically, except \( A[j] = \max_i x_i g_j(i) \) (here, \( x_i = 1 \) denotes an individual with the disease, and \( x_i = 0 \) is a healthy individual). Please show how to reconstruct \( x \) from \( A \) with high probability. \[ \text{[Solution]} \]

(b) **Checksums and Error Correcting Codes.** Consider a length-\( n \) binary message \( x_1 \ldots x_n \) in which up to \( k \) bits might be corrupted. We would like to identify these bits by comparing the corrupted version of \( x \) with an \( m \)-bit checksum \( A[1 \ldots m] \) pre-computed on the original \( x \). As above, suppose each bit in the checksum corresponds to a random set of bits in \( x \), where we define \( A[j] = \bigoplus_i x_i g_j(i) \) (\( \bigoplus \) denotes the XOR operation). Show that we can determine which bits were corrupted\(^{16} \) with high probability with a checksum of length \( m = O(k \log n) \). \[ \text{[Solution]} \]

---

**L_1 Minimization and Linear Programming.** Since each \( A[j] \) is defined by a linear equation in \( x_1 \ldots x_n \), a linear sketch is a linear system of \( n \) variables in \( m \) equations, which in this context has no single unique solution for \( x \) since \( n \) is much larger than \( m \). If we believe \( x \) should have most of its mass concentrated in a few components, we can attempt to recover \( x \) by minimizing \( \|x\|_0 \) subject to our system of linear constraints — that is, we want to find a sparsest solution to an under-determined linear system. Although this is unfortunately NP-hard, a sparse solution often “magically” materializes if we instead minimize \( \|x\|_1 = \sum_i |x_i| \), an objective we can minimize in polynomial time over a system of linear constraints by solving a linear program (see Chapter 12). For those well-versed in linear algebra, we include a brief discussion of why this approach tends to work.

Due to the propensity of the \( \|x\|_1 \) objective to produce sparse solutions, it is often used as a “regularizing” term added as a penalty to other objective functions when sparse solutions are desired. For example, instead of just minimizing \( f(x) \), we might minimize instead \( f(x) + \lambda \|x\|_1 \) for some appropriate value of \( \lambda \) to encourage solutions

---

\(^{15}\)Recall that in general, we define the \( L_p \) norm of \( x \) as \( \|x\|_p = (\sum_i |x_i|^p)^{1/p} \). In the limit as \( p \to 0 \), this does indeed converge to the number of non-zero components in \( x \).

\(^{16}\)A complicating aspect we do not consider here, but that is important in most constructions of error-correcting codes, is that bits in the checksum itself might also be corrupted by the same process that corrupted \( x \) (e.g., a noisy communication channel).
that are sparse. Further comments on regularization in optimization appear in Chapter ??.

7.5.9 Locality-Sensitive Hashing

Collisions in hashing are often undesirable. However, in *locality-sensitive hashing*, they are actually the most useful aspect of a hashing scheme! Since similar objects often collide when hashed, we carry this idea one step further and try to design a randomized hash function for which collision probability directly reflects object similarity. If $\text{sim}(x, y)$ is the similarity between two complicated objects $x$ and $y$, our goal is to build a hash function $h$ for which $\Pr[h(x) = h(y)] = \text{sim}(x, y)$, where the probability is taken over the random parameters baked into the hash function. Here are three prominent examples:

- **Negated $L_1$ Distance.** Let $x$ and $y$ be two points in $[0, 1]^d$, the $d$-dimensional unit cube. We often use distance to measure dissimilarity, so we can convert this into a similarity measurement by negation:

  $$\text{sim}_D(X, Y) = 1 - \frac{1}{d}\|x - y\|_1.$$  

  Here, we use $L_1$ distance, which between two points in a $d$-dimensional unit cube can be at most $d$. By dividing by $d$ and then subtracting from one, we therefore obtain a measurement of similarity in $[0, 1]$.

- **Correlation.** If $x$ and $y$ are vectors in high-dimensional space, we often measure similarity by taking their correlation, $\hat{x} \cdot \hat{y}$, where $\hat{x} = x/\|x\|$ denotes $x$ normalized to have unit Euclidean length. Correlation measures the cosine of the angle between $x$ and $y$, ranging from $-1$ for vectors pointing in opposite directions to $+1$ for vectors pointing in the same direction. Since we prefer measurements of similarity in the range $[0, 1]$, we use a re-scaled version:

  $$\text{sim}_C(x, y) = \frac{1}{2} \left(1 + \hat{x} \cdot \hat{y}\right).$$

- **Jaccard Similarity.** If $X$ and $Y$ are two sets, it is natural to measure their similarity in terms of relative overlap,

  $$\text{sim}_J(X, Y) = \frac{|X \cap Y|}{|X \cup Y|},$$

  This is known as their *Jaccard* similarity, and it always lies in the range $[0, 1]$.

In each of these three settings, we can design a simple hash function for which collision probability reflects similarity.

For negated $L_1$ distance, we select a random coordinate $i \in \{1, \ldots, d\}$ and a random threshold $t$, and hash point $x$ to 1 if $x_i \geq t$, and 0 otherwise. It is easy to show that $\Pr[h(x) = h(y)] = \text{sim}_D(x, y)$. [Details]

---

A common alternate definition of a locality-sensitive hash function is a function for which $\Pr[h(x) = h(y)] \geq p$ if $\text{sim}(x, y)$ is above some threshold $t$, but $\Pr[h(x) = h(y)] \leq p'$ (with $p' < p$) if $\text{sim}(x, y)$ is below some smaller threshold $t' < t$. Hence, collision probabilities still allow us to differentiate between similar and less-similar objects.
For correlation, we pick a random hyperplane through the origin described by its normal vector \( \hat{v} \). We hash point \( x \) to 1 if \( x \cdot \hat{v} \geq 0 \), otherwise we hash \( x \) to 0. It is easy to show that \( \Pr[h(x) = h(y)] \approx \operatorname{sim}_C(x, y) \). [Details]

For Jaccard similarity, we use a \textit{min-wise} hash function, defined for a set \( X \) as \( h(X) = \min\{f(e) : e \in X\} \), where \( f \) assigns a unique random number to every element\(^\text{18}\) in our universe \( U \). Since \( h(X) = h(Y) \) only if the minimum element in \( X \cup Y \) comes from \( X \cap Y \), we have \( \Pr[h(X) = h(Y)] = \frac{|X \cap Y|}{|X \cup Y|} = \operatorname{sim}_J(X, Y) \).

### Hashing to a Single Bit

Several of the example hash functions above map a complicated object down to just a single bit. In fact, we can achieve this with a simple modification to any locality-sensitive hash function. Starting with a function \( h \) satisfying \( \Pr[h(x) = h(y)] = \operatorname{sim}(x, y) \), consider sending its output through a secondary hash function \( g \) mapping to \( \{0, 1\} \). Using an appropriate universal hash function for \( g \), we have

\[
\Pr[g(x) = g(y)] = \begin{cases} 
1 & \text{if } x = y \\
1/2 & \text{if } x \neq y
\end{cases}
\]

If we now expand out \( \Pr[g(h(x)) = g(h(y))] \) by conditioning on whether or not \( h(x) = h(y) \), we get

\[
\Pr[g(h(x)) = g(h(y)) \mid h(x) = h(y)] \cdot \Pr[h(x) = h(y)] + \Pr[g(h(x)) = g(h(y)) \mid h(x) \neq h(y)] \cdot \Pr[h(x) \neq h(y)],
\]

so \( \Pr[g(h(x)) = g(h(y))] = (\operatorname{sim}(x, y) + 1)/2 \). The binary function \( g(h(x)) \) therefore still gives a collision probability reflecting object similarity, albeit slightly re-scaled.

### Hashing to a Binary String

We can now hash an object down to a single bit so that collision probability estimates object similarity. To obtain a more accurate estimate, we can repeat this with \( d \) independent hash functions, thereby hashing an object \( x \) to a \( d \)-bit binary string \( s_x \). Object similarity between \( x \) and \( y \) is now given by the average number of bits that agree between \( s_x \) and \( s_y \), which we often write as \( 1 - H(s_x, s_y)/d \), where \( H(s_x, s_y) \) is the number of bits that differ between \( s_x \) and \( s_y \), known as their Hamming distance (note that this is actually the same as our formula for negated \( L_1 \) distance, since Hamming distance is the same thing as \( L_1 \) distance, only in the context of binary strings). Locality-sensitive hashing therefore reduces similarity computation of potentially large, complex objects, down to the conceptually simpler problem of Hamming distance computation in binary strings. A nice application of this is shown in the following section.

### 7.5.10 High-Dimensional Nearest Neighbor Search

In the next chapter, we will learn several classical data structures for finding \textit{near neighbors} in a geometric point set, but these unfortunately only work well in very

\(^{18}\)Ideal min-wise hash functions are generally not possible to construct due to space considerations, since we would need to store where every element in our universe \( U \) (a very large set) is mapped under \( f \). We therefore usually try to build \textit{approximate} min-wise hash functions described by limited amounts of randomness (much the same idea as with universal hashing), where \( \Pr[h(X) = h(Y)] \approx \operatorname{sim}_J(X, Y) \). See the endnotes for further references to results in this area.
low-dimensional spaces, having time and/or space bounds that scale exponentially in terms of dimension (this phenomenon is widespread in computational geometry, and is sometimes known as the “curse of dimensionality”). It is quite common in many applications, however, to encounter near neighbor problems in extremely high-dimensional spaces — say, with thousands of dimensions. For example, in machine learning, we might represent objects by high-dimensional feature vectors, where we try to infer the classification of a query object by looking at its nearest neighbors in a training set of pre-classified objects.

Locality-sensitive hashing helps to simplify proximity questions like near neighbor queries by reducing them to equivalent proximity questions over points in \( \{0, 1\}^d \) (i.e., length-\(d\) binary strings) using the Hamming distance metric. Note that the dimension \(d\) of this new “binary” problem is somewhat unrelated to the dimension of our original data before applying locality-sensitive hashing, but that \(d\) is typically also quite large. Even so, the simpler structure of the new problem allows us to use hashing to build approximate near neighbor structures that perform well even if \(d\) is quite large. Given a distance threshold \(\Delta\) and desired approximation factor \(c > 1\), our goal is build a static data structure on \(n\) data points in \(\{0, 1\}^d\) such that given any query point \(q\):

- If there exists a point \(p\) in our data set with \(H(p, q) \leq \Delta\), the query returns an approximate near neighbor — a point \(p'\) with \(H(p', q) < c\Delta\).

- If \(H(p, q) \geq c\Delta\) for every point \(p\) in our data set, the query returns “No near neighbor”.

By querying multiple instances of this structure built for distance thresholds \(\Delta = 1, c, c^2, c^3, \ldots\), we can find a \(c^2\)-approximate nearest neighbor of \(q\), since if the distance to the true nearest neighbor of \(q\) lies in the range \([c^x, c^{x+1}]\), then the structure will return a point at distance at most \(c^{x+2}\) when run at the threshold \(\Delta = c^{x+1}\).

To find approximate near neighbors, the main idea is to hash each data point \(p\) to a lower-dimensional point by projecting it onto a random subset of coordinates chosen during preprocessing, with each coordinate \(i = 1 \ldots d\) retained independently with some probability \(\alpha\). The lower we set \(\alpha\), the more likely distant points will collide when hashed. For example, points \(p_i = 01111010\) and \(p_j = 11110000\) collide if we retain their even coordinates (underlined).

Consider a query point \(q\) for which a nearby point \(p\) with \(H(p, q) \leq \Delta\) exists in our dataset. Our hash table makes it easy to enumerate all the points \(p'\) in our dataset colliding with \(q\). If we have chosen \(\alpha\) to be low enough, then it is likely that we will find in this set a “good” point \(p'\) (with \(H(p', q) < c\Delta\)). However, if we chose \(\alpha\) to be too low, we will end up also searching through a large number of “bad” points (with \(H(p', q) \geq c\Delta\)). By analogy, if you are trying to catch a rare breed of fish, you need to cast a large enough net to ensure catching at least one specimen of the rare breed, but if you cast too large a net, you will waste time sorting through the many other fish you catch as collateral damage.

By choosing \(\alpha\) appropriately and also querying several independent copies of our data structure, we can answer \(c\)-approximate nearest neighbors queries correctly with high probability in time \(O(dn^{1/c})\), and with \(O(nd + n^{1+1/c})\) space usage (ignoring extra logarithmic factors for simplicity, to highlight the dominant terms.
in each bound). Observe that these bounds are respectively sublinear and sub-quadratic in $n$, with no exponential dependence on dimension $d$. For large values of $n$, the query time bound above can be much better than the naïve approach that finds a nearest neighbor in $\Theta(dn)$ time by simply checking every point in the data set. [Full details]
For those who work with databases, the following should look familiar:

```sql
SELECT title, author FROM books_in_store
    WHERE price <= 100
    AND publication_date >= 1990
    AND page_count BETWEEN 500 AND 750;
```

This is a database query written in structured query language (SQL), asking for a list of titles and authors for books in a database table (books_in_store) satisfying certain conditions on price, publication date, and page count. Databases are one of many prominent applications that depend on fast data structures for range queries, the subject of this chapter.

Range queries come in many forms. As shown in Figure 8.1, we can ask range queries in dictionaries (sets), sequences, trees, and multi-dimensional point sets. For the database example above, we can regard a database of $n$ records each containing $d$ fields abstractly as a set of $n$ points in $d$-dimensional space\(^1\), so range queries in databases also fall under the umbrella of geometric range queries over multi-dimensional point sets.

In this chapter, we study several elegant approaches for building range query structures. We consider both static and dynamic data structures, where in the dynamic case we also look at range update queries, asking us to modify all the elements in a range in a batch fashion. Many of the ideas we use are based on binary search trees, so the reader may want to review Chapter 6 as helpful prerequisite.

### 8.1 One-Dimensional Range Queries

In a sequence $A_1 \ldots A_n$, a range query typically asks for some aggregate statistic (e.g., sum, min, max) over a range $A_i \ldots A_j$. In a dictionary (or equivalently, a one-dimensional set of points), we usually want to count or enumerate the elements in

---

\(^1\)Some dimensions like “author name” may not be numeric, but this usually will not matter since most of the data structures in this chapter are comparison-based.
a range \([a, b]\). We typically specify a range by its endpoint values \(a\) and \(b\), although most of the structures we discuss can also be adapted for rank-based access (e.g., enumerating elements between the 50th and 60th percentile).

**Range Queries Using BSTs.** The balanced binary search tree (BST) is commonly used for one-dimensional range queries. Its close relatives, the \(B\)-tree and skip list, behave similarly, with \(B\)-trees being used in practice particularly often for databases stored outside main memory (e.g., on disk).

The standard picture of a range query in a balanced BST is shown in Figure 8.2(a). Elements in the range \([a, b]\) consist of some of the \(O(\log n)\) elements along the search paths down to \(\text{succ}(a)\) and \(\text{pred}(b)\) (since \(a\) and \(b\) themselves may not be present), as well as the contents of the \(O(\log n)\) subtrees hanging in the interior of these two paths. We refer to this subset of the tree, containing of all the elements in \([a, b]\), as the **footprint** of our query.

We can enumerate the \(k\) elements in \([a, b]\) in \(O(k + \log n)\) time several ways:

- Starting from \(e = \text{succ}(a)\), we can step through successive elements with repeated calls to \(e = \text{succ}(e)\) until \(e\)'s key exceeds \(b\).
- Equivalently, we can use recursion: for a tree with root key \(r\), we recursively issue the same range query in its left subtree if \(a < r\), we then print the root if \(r \in [a, b]\), and then we recursively issue the same range query in its right subtree if \(b > r\).
8.1. ONE-DIMENSIONAL RANGE QUERIES

Figure 8.2: The structure of a range query over \( [a, b] \) in (a) a balanced BST, and (b) a splay tree. In (a), the elements answering the query range from \( \text{succ}(a) \) (the smallest element \( \geq a \)) to \( \text{pred}(b) \) (the largest element \( \leq b \)) and consist of some of the elements on the search paths down to \( \text{succ}(a) \) and \( \text{pred}(b) \), plus the contents of the subtrees hanging in the interior of these search paths. In (b), we have isolated the contents of \( [a, b] \) into a single subtree by splaying \( \text{succ}(\text{pred}(b)) \) (the smallest element \( > b \)) and then \( \text{pred}(\text{succ}(a)) \) (the largest element \( < a \)). We often add dummy \( -\infty \) and \( +\infty \) elements to our tree to ensure for simplicity that these flanking elements always exist.

The recursive approach may strike you as being quite similar in structure to an inorder traversal, and this is no accident, since the result is a printout of just the subset of the inorder traversal within \( [a, b] \).

Enumerating the \( k \) elements in a range is slow if \( k \) is large, due to the unavoidable output-sensitive \( \Theta(k) \) term in our running time. However, we can count the elements in a range in \( O(\log n) \) time (with no dependence on \( k \)), as long as each element is augmented with its subtree size. To do this, we need only consult the \( O(\log n) \) bold elements in Figure 8.2, corresponding to the individual elements along the search paths down to \( \text{succ}(a) \) and \( \text{pred}(b) \) that lie within the range, as well as the roots of the subtrees hanging off these paths in the interior of the range. Equivalently, we can count the elements in the range by taking \( \text{rank} (\text{pred}(b)) - \text{rank}(\text{succ}(a)) + 1 \).

If used to store a sequence \( A_1 \ldots A_n \) rather than a dictionary, a balanced BST can answer range queries (e.g., sum, min, or max over a range \( A_i \ldots A_j \)) in \( O(\log n) \) time by again aggregating information from the bold elements in Figure 8.2, as long as we augment the tree appropriately. For example, to answer range minimum queries, we augment each node so it knows the minimum in its subtree.

One-dimensional range queries are particularly nice in splay trees, since by splaying the elements immediately flanking a range on the left and right in \( O(\log n) \) amortized time, we can isolate the footprint of the query into a single subtree\(^2\), as shown in Figure 8.2(b). We then only need to enumerate the contents of this one subtree or examine the augmented information at its root. In Section 6.1.7, we showed another outlook on this process using \textit{split} and \textit{join} to pull the subtree representing

\(^2\)The second splay operation should unseat the root element and only move it one step downward, so we must ensure that its final step is not an out-of-order double rotation; we would use two single rotations instead.
any range completely out of a BST in $O(\log n)$ time (amortized, if using splay trees).

**Problem 119 (Range Updates in BSTs).** A range update operation in a sequence $A_1 \ldots A_n$ modifies all of the elements in a contiguous subsequence $A_i \ldots A_j$ in a batch fashion. Here, we consider how to implement range updates very quickly for sequences encoded in balanced BSTs.

(a) **Range Modification.** Consider the range-add$(i,j,v)$ operation that adds a common value $v$ to each element in $A_i \ldots A_j$. Please show how to modify the implementation of a balanced BST so it can also support range updates in $O(\log n)$ time (possibly amortized). As a hint, try attaching a value $\Delta(e)$ to each element $e$ indicating that every element in $e$’s subtree should have its value increased by $\Delta(e)$. [Solution]

(b) **Range Reversal.** Suppose we wish to add to a balanced BST the flip$(i,j)$ operation that reverses the order of the elements in $A_i \ldots A_j$. How can we do this so that all operations still run in $O(\log n)$ (possibly amortized) time? [Solution]

For range queries over integers, we have the option of using fast RAM data structures like the Y-fast tree or van Emde Boas structure from the preceding chapter. However, in order to simplify our discussion, we will focus on comparison-based structures for the remainder of this chapter.

### 8.1.1 Static Range Queries and Prefix Sums

Range queries are typically much easier in the static case, where our underlying data set does not change. For example, to enumerate the elements in $[a,b]$ in a static dictionary in $O(k + \log n)$ time, we simply store the dictionary in a sorted array, binary search for the two endpoints $a$ and $b$, and write out all the elements in between. No fancy data structures are needed.

**Prefix Sums.** In a static sequence $A_1 \ldots A_n$, we can answer range-sum queries in $O(1)$ time after spending $\Theta(n)$ time precomputing a sequence $B_1 \ldots B_n$ of prefix sums, where $B_j = A_1 + \ldots + A_j$. Now, range-sum$(i,j) = B_j - B_{i-1}$. The simple yet powerful idea that “range sum equals difference of two prefix sums” appears often in data structure design. For example, in the next chapter we will see how to compute the hash of an arbitrary substring of a larger string in $O(1)$ time by taking an appropriate “difference” of two prefix hashes computed during preprocessing.

**Successive Differences.** In the other direction, consider building a length-$n$ sequence $D$ of successive differences, where $D_1 = A_1$, and for $i > 1$ we have $D_i = A_i - A_{i-1}$. Observe that this is the inverse of the prefix sum transformation: starting with $A$, if we compute prefix sums and then take successive differences, we get back $A$. Likewise, we get back $A$ if we take successive differences and then compute prefix sums. For those who know calculus, you can think of taking prefix sums and successive differences as discrete analogs of integration and differentiation.

Since $A$ can be perfectly recovered from either the prefix sum or successive difference sequences, these provide equivalent, or “dual” representations of sequences that can be advantageous in some applications. For example, we have already seen how to obtain $O(1)$-time range sums by storing $A$ in its prefix sum representation. If we store $A$ in successive difference form, this allows $O(1)$-time range **updates:** to add $x$
to every element in $A_i \ldots A_j$, this only requires adding $+x$ to $D_i$ and $-x$ to $D_{j+1}$.

Switching representations can sometimes allow us to express a problem in a more recognizable or mentally intuitive form. For example, given a sequence $A_1 \ldots A_n$, consider finding a subarray of $A$ with maximum sum, a well-studied problem that can be solved in many different ways, in as little as $\Theta(n)$ time (see Section 2.3 and problem 207). Given that the sum of $A_{i+1} \ldots A_j$ is equivalent to the difference $P_j - P_i$ of two prefix sums, we could invent a completely equivalent problem like the following: “Given a sequence $P_1 \ldots P_n$ of daily stock prices, what is the maximum profit $P_j - P_i$ you can obtain by buying one share of stock on some day $i$ and selling it on some later day $j > i$?” This new outlook on the problem can help us discover good solutions. For example, what is the best starting day $i$ for a particular ending day $j$? Since we want to maximize $P_j - P_i$ where $P_j$ is fixed, we want to choose $P_i$ to be as small as possible over $i < j$, and we can answer this in $O(1)$ time after precomputing an array of prefix minima in $\Theta(n)$ time. What if we make the problem harder by imposing lower and upper bounds $[a, b]$ on the size of our window? In this case, we can use the same approach — select the best $i$ for each $j$ — only now instead of a simpler prefix minimum query problem we get a range minimum query problem, asking us to find the minimum of $P_i$ over all $i \in [j - b + 1, j - a + 1]$.

In a few pages (problem 126), we will see how to answer this query in $O(1)$ time as well, after appropriate preprocessing in $\Theta(n)$ time.

**Problem 120 (Two-Dimensional Range Sums and Updates).** In this problem, we investigate how to generalize the notions of prefix sums and successive differences
Figure 8.4: A protein as a kinematic chain. The “backbone” of the protein human serum albumin (the most abundant protein in human blood) is shown in (a). Proteins are built from sequences of amino acids, each of which share the same 6 atoms along the backbone and differ only in their “R” parts, as shown in (b). Each amino acid has a rigid structure, and is connected to its neighbors by two rotatable bonds. These bond angles collectively determine the 3-dimensional shape of the protein.

to two or more dimensions.

(a) First, given an $n \times n$ matrix $A$, please show how to preprocess $A$ in $\Theta(n^2)$ time and space so that we can query for the sum of any rectangular submatrix in $O(1)$ time. Next, starting with an $n \times n$ matrix $A$ of zeros, show how to perform updates in $O(1)$ time that each increment the contents of some rectangular submatrix, after which we can compute the contents of $A$ after $\Theta(n^2)$ postprocessing time. [Solution]

(b) Extending your solution from part (a), suppose you are given a set of $n$ points in $d > 1$ dimensions. Show how to preprocess the points in $\Theta(n^d)$ time and space so that you can then query for the number of points in any $d$-dimensional rectangular volume (with sides parallel to the coordinate axes) in $O(2^d + d \log n)$ time. [Solution]

### 8.1.2 Trees Atop Arrays

The balanced BST and its relatives can perform fast range queries in a fully-dynamic sequence, allowing modification as well as insertion or deletion of elements. If we only need to support modification but not insertion or deletion, a much simpler alternative approach is to use a complete binary tree built “on top of” an array $A[1 \ldots n]$. As shown in Figure 8.3, each node is augmented with aggregate information (e.g., sum, min, max) for the part of the sequence it covers. Compared to the BST, this has the advantage of allowing access to any element $A[i]$ in only $O(1)$ time (versus $O(\log n)$, for the balanced BST). Modification of $A[i]$ still takes $O(\log n)$ time, since afterward we need to walk up the tree updating augmented information. A range query over $A[i \ldots j]$ can be answered by consulting the roots of $O(\log n)$ subtrees hanging in the middle of the search paths down to $A[i]$ and $A[j]$, just like with the balanced BST.

A clever way to encode a binary tree on top of an array $A[1 \ldots n]$ using a single
8.1. ONE-DIMENSIONAL RANGE QUERIES

int prefix_sum(int j) {
    int sum=0;
    for (; j>0; j-=(j&-j))
        sum += B[j];
    return sum;
}

void modify(int j, int v) {
    for (; j<=n; j+=(j&-j))
        B[j] += v;
}

Figure 8.5: Short C implementations of the prefix-sum operation (which computes A[1] + ... + A[j]) and modify operation (which adds v to A[j]) in a binary indexed tree. The bitwise operation (j&-j) isolates the least-significant 1 bit in j's binary representation, owing to the way negative numbers are stored in binary. For example, j-=(j&-j) zeros out j's least significant 1 bit.

array B is explained in Figure 8.3(b). The length of B can be up to 3n, since part of the tree at the end can “overhang” the array. This method can be viewed as an alternative to the binary heap as means of encoding of a tree within an array. Like the binary heap, we can move around as if we were in the tree by performing arithmetic on our index in the array. An advantage of this encoding over the binary heap, however, is that it allows fast addition or removal of elements at the ends of A (say, if A represents a stack or queue).

Problem 121 (Maximum-Value Subarray Queries). In an array A[1...n], suppose we want to support an operation maximum-range-sum that reports the endpoints i and j of the range A[i...j] achieving the maximum sum (note that this problem is generally only interesting if our array contains some negative numbers). Please describe an approach that allows us to modify elements in O(log n) time and answer maximum-range-sum queries in O(1) time. [Solution]

By using more exotic operations in place of the usual sum, min, or max, we can address a surprising number of applications. For example, consider a kinematic chain, a sequence of n rigid elements connected by movable joints like a robot arm or protein molecule (Figure 8.4). If M_i is a matrix transformation (a translation times a rotation) from element i − 1 to element i, then we can obtain the position of any element j relative to the start of the chain using the product M_1 ⋯ M_j. Building a tree atop the multiplicative sequence M_1 ⋯ M_n, we can therefore modify the angle of any joint in O(log n) time and also query for the position of any element in O(log n) time.

8.1.3 Binary Indexed Trees

The binary indexed tree encodes a sequence so that we can modify an element or compute any prefix-sum in O(log n) time. Since a range sum is the difference of two prefix sums, we can therefore also implement range-sum in O(log n) time. While these capabilities are also present in a balanced BST and a tree atop an array, the binary indexed tree provides them with minimal space and with remarkably succinct implementations, as seen in Figure 8.5.
In a slight variation of our previous “tree atop array” idea, here we encode an array $A[1 \ldots n]$, using a single array $B[1 \ldots n]$. We do not need to store the original array $A$ explicitly, since we can recover $A[j]$ by taking $\text{prefix-sum}(j) - \text{prefix-sum}(j - 1)$. As shown in Figure 8.6(a), we take a “right-justified” binary tree atop $A$, augment it with subtree sums, and store the value of the highest node at each position in $B$.

To compute $\text{prefix-sum}(j) = A[1] + \ldots + A[j]$, we add the elements in $B$ corresponding to the roots of the $O(\log n)$ subtrees covering $A[1 \ldots j]$, just as before. However, the indices of these elements within $B$ correspond in a nice way to the binary representation of $j$: they are the numbers we get when we start with $j$ and successively zero out the least significant 1 bit in $j$’s binary representation (a very simple operation in code, as seen in Figure 8.5). We modify $A[j]$ by an additive amount $v$ by adding $v$ to all the subtree roots covering $A[1 \ldots j]$. This also has a simple binary interpretation in terms of repeatedly incrementing just the final 1 bit in $j$, stopping when our index exceeds $n$.

The binary indexed tree highlights an elegant duality between binary trees and binomial trees (Section 5.4.3), showing us that the binary indexed tree is perhaps more correctly regarded as an application of binomial trees. In Figures 8.6(b) and 8.6(c), we see that the union of all possible query and update paths form binomial trees. This also shows how one can map a binomial tree into an array, allowing movement within the tree based on bitwise index manipulation, much like we can encode a binary tree in an array using the level-order encoding of the binary heap or the interleaved encoding of a tree atop an array we just learned.
8.2 Stabbing Queries and Interval Trees

The “inverse” of a range query is a stabbing query, where we build a data structure on a set of ranges, and then ask it to count or enumerate all the ranges “stabbed” by a query point. In one dimension, this translates to a data structure for storing a set of intervals \([a_1, b_1] \ldots [a_n, b_n]\), so that we can quickly count or enumerate the intervals containing a given query point \(x\). Interestingly, we can turn these (and many other stabbing queries) back into equivalent range queries by regarding each interval \([a_i, b_i]\) as a point \((a_i, b_i)\), and then asking for all points in the rectangular region of 2D space described by \(a_i \leq x \) and \(b_i \geq x\). We discuss efficient structures for these sorts of 2D range queries in a few pages.

A well-known data structure for one-dimensional stabbing queries over a static set of intervals is the interval tree, and we can illustrate its construction in terms of a general divide-and-conquer approach that will help us build several other stabbing and range query structures later in the chapter. Consider the following three problems, ordered from easiest to hardest:

1. Build a data structure for answering stabbing queries on a static set of \(n\) intervals whose right endpoints all coincide at the point \(x^*\).

2. Build a data structure for answering stabbing queries on a static set of \(n\) intervals that all contain some point \(x^*\).

3. Build a data structure for answering stabbing queries on an arbitrary static set of \(n\) intervals.

A simple data structure for problem 1 is just a sorted array of the left endpoints of all our intervals. This allows us to count the number of intervals containing a query point \(x\) in \(O(\log n)\) time (via binary search), and to enumerate this set of \(k\) intervals in \(\Theta(1 + k)\) time (by stepping through the array until we reach a left endpoint larger than \(x\)).
Given a solution to problem 1, let us now show how we can systematically generalize it to solve problems 2 and 3. We solve problem 2 with two “back to back” copies of a data structure for problem 1. This involves splitting each interval into two sub-intervals at the point $x^*$, after which we use one structure to handle queries for values $x > x^*$, and the other to handle queries for values $x < x^*$.

To solve problem 3, we use recursion: letting $x_{mid}$ be the median of the $2n$ endpoints of our intervals, we can divide our intervals into sets $L$ (intervals entirely to the left of $x_{mid}$), $M$ (intervals containing $x_{mid}$), and $R$ (intervals entirely to the right of $x_{mid}$). As shown in Figure 8.7(a), we build a binary tree where the root contains a data structure for problem 2 to handle stabbing queries for the set $M$. The left and right subtrees are recursively built from $L$ and $R$. To execute a stabbing query, we first query the root data structure to count or enumerate all the intervals in $M$ containing our query point $x$, after which we recursively query either the left subtree or the right subtree, depending on whether $x < x_{mid}$ or $x > x_{mid}$. Due to our choice of $x_{mid}$, both $L$ and $R$ will contain at most $n/2$ intervals, implying that the entire interval tree has height $O(\log n)$, so we can answer problem 3 with $O(\log n)$ queries to a data structure for problem 2. We can therefore enumerate the $k$ intervals containing a query point in $O(k + \log n)$ time and count the number of such intervals in $O(2n)$ time.

Using amortized rebuilding (Section 6.2.6) as well as a balanced BST instead of a sorted array for problem 1, we can make the interval tree dynamic, supporting fast insertion and deletion of intervals as well as the above bounds on query time.

Problem 123 (Stabbing Queries and Persistent Data Structures). Persistent data structures (Section 4.7) are often useful for range and stabbing query problems. Suppose we are given a static set of $n$ intervals on the number line, which we should preprocess in $O(n \log n)$ time and $\Theta(n)$ space, so that we can subsequently enumerate the $k$ intervals stabbed by a query point in $O(k + \log n)$ time, or count these intervals in $O(\log n)$ time. Show how to solve this problem using a persistent balanced BST. [Solution]

Problem 124 (Segment Trees). Another prominent one-dimensional interval stabbing structure is a segment tree. As shown in Figure 8.8(a), the $2n$ endpoints of a set of $n$ intervals partition the number line into at most $2n + 1$ regions, each with an associated “overlap count” (the number of intervals overlapping in that region). Suppose we store this sequence of regions and their overlap counts in a balanced BST. Please show how we can handle insertions and deletions of intervals in $O(\log n)$ time, query for the number of intervals stabbed by a query point in $O(\log n)$ time, and enumerate the $k$ intervals stabbed by a query point in $O(k + \log n)$ time. Hint: break each interval into $O(\log n)$ segments stored at different locations in the tree, bringing the total space usage to $O(n \log n)$. [Solution]

Problem 125 (Interval Stabbing with an Augmented BST). Suppose we store $n$ intervals in a balanced BST keyed on left endpoint, where we augment each element with the maximum right endpoint in its subtree. Please show how this structure can enumerate the $k$ intervals stabbed by a query point in $O(k + \log n)$ time. Is it easy to count stabbed intervals without dependence on $k$ in the running time using this structure? [Solution]

---

3It is usually simpler to choose one of our $2n$ endpoints at random as $x_{mid}$. Using the randomized reduction lemma, we can show that this leads to the same performance bounds, only with high probability. This applies to many of the structures in this chapter (e.g., priority search trees, kd-trees).
8.3 Range Minimum Queries

In a sequence $A_1 \ldots A_n$, the range-min($i, j$) query should report the minimum of $A_i \ldots A_j$. Here, we consider range-min (and equivalently, range-max) queries (RMQs) in a static sequence. These have a surprisingly broad set of algorithmic applications, and it will be useful to study them in greater depth.

We can easily answer RMQs in $O(\log n)$ time, even in a dynamic sequence, using balanced BSTs or trees atop arrays. In the static case, $O(1)$ query time is also easy to achieve if we compromise on preprocessing time and space. Since a query is characterized by its two endpoints $i$ and $j$, there are only $\Theta(n^2)$ different possible queries, so we can spend $\Theta(n^2)$ preprocessing time and space building a table of answers to all of them. In this section, we will see how to achieve $O(1)$ query time along with just $\Theta(n)$ preprocessing time and space. We build some preliminary results in support of this goal in the following sequence of problems:

Problem 126 (Fixed-Length RMQ Structures). Suppose you are told that all RMQs will be over windows of length $k$. Can you devise a simple data structure that requires only $\Theta(n)$ preprocessing time and space and answers queries in $O(1)$ time? As a hint, start by partitioning the sequence into $n/k$ blocks of length $k$. [Solution]

Problem 127 (Constant Query Time with Nearly-Linear Space). Here, we answer arbitrary RMQs in $O(1)$ time, given $\Theta(n \log n)$ preprocessing time and space.

(a) Show how to achieve the bounds above by precomputing answers for every window whose length is a power of two. [Solution]

(b) Show how to achieve the bounds above by following the three-step process we used to build an interval tree. [Solution]

Problem 128 (Answering Long RMQs). By using insight from the two preceding problems, show how to answer RMQs of length at least $\log n$ in $O(1)$ time, given $\Theta(n)$ preprocessing time and space. As a hint, divide the sequence into $\frac{n}{\log n}$ blocks of length $\log n$, giving enough slack to build a slightly more time- and space-intensive structure on the sequence of block minima. [Solution]
Figure 8.9: The process of (a) converting an LCA problem into an equivalent RMQ problem by writing out node depths along an Eulerian traversal of a tree, and (b) converting an RMQ problem into an equivalent LCA problem by building a Cartesian tree. Below the array of node depths in part (a) is its compressed binary representation, with 1 indicating an increase and 0 a decrease.

The preceding solutions give $O(1)$ query time in linear space except for the surprisingly challenging case of short RMQs (length at most $\log n$). We will see how to handle these also in $O(1)$ time in the next sections after learning additional tools.

Problem 129 (Recursive Bootstrapping). By taking the solution of problem 128 and building recursive structures inside each size-$\log n$ block, show how to achieve a query time of $O(\log^* n)$ using $\Theta(n \log^* n)$ total space. Then, using this result as a black box, show that we can use essentially the same high-level recursive approach, only now with a block size of $\log^* n$, to achieve a query time of $O(\log^{**} n)$ with $\Theta(n \log^{**} n)$ total space. Ultimately, show that repeated application of this process leads to a structure with query time $O(\alpha(n))$ taking $O(n \alpha(n))$ space. [Solution]

8.3.1 Lowest Common Ancestors

The lowest common ancestor (LCA) of two nodes $i$ and $j$ in a rooted (not necessarily binary) tree is the node where the paths from the root down to $i$ and $j$ diverge (Figure 8.8(b)). LCAs have numerous applications — for example, in string matching problems involving suffix trees (Section 9.4), for verifying minimum spanning trees (Section ??), for building evolutionary trees (Section ??), and for encoding the connectivity of a graph (Section ??). LCAs have many simpler uses as well; for instance, they can help us answer “is $i$ an ancestor of $j$?” queries in a rooted tree.
The static LCA and RMQ problems are “equivalent” in that they can be transformed into each-other in $\Theta(n)$ time. As a benefit, the data structures we develop here for RMQs will also be able to compute LCAs in $O(1)$ time, given $\Theta(n)$ preprocessing time and space. To transform an LCA problem on a tree $T$ into an equivalent RMQ problem on a sequence, we perform an Eulerian traversal of $T$, writing down the depths of the nodes we visit in sequence. As shown in Figure 8.9(a), the LCA of $i$ and $j$ is given by the node of minimum depth along the part of the traversal between the first and last occurrences of nodes $i$ and $j$, and we can locate this node using an RMQ on the sequence of node depths.

### 8.3.2 Cartesian Trees

To transform an RMQ problem into an equivalent LCA problem, we build a Cartesian tree, a rooted binary tree derived from a sequence $A_1 ... A_n$ as shown in Figure 8.9(b). We place a minimum element (say, $A_i$) at the root, and build the left and right subtrees of the root by recursively computing Cartesian trees from $A_1 ... A_{i-1}$ and $A_{i+1} ... A_n$. Cartesian trees satisfy the heap property, and the inorder traversal of a Cartesian tree yields the original sequence from which it was constructed. The LCA of nodes $A_i$ and $A_j$ in the Cartesian tree corresponds to the minimum in the range $A_i ... A_j$, since the range $A_i ... A_j$ remains contiguous and intact during the construction process outlined above, until it is broken apart by our first selection of an element from the range — the minimum — which becomes the LCA of all the elements in the range.

**Problem 130 (Building Cartesian Trees in Linear Time).** The “top down” construction approach described above illustrates well the structure of the Cartesian tree, but is actually not the fastest way to build one. A Cartesian tree can be built from a sequence $A_1 ... A_n$ inductively in $\Theta(n)$ total time. In the $k$th step of this process, we take a Cartesian tree for $A_1 ... A_{k-1}$ and modify it so it becomes a Cartesian tree for $A_1 ... A_k$ by adding $A_k$ to the bottom of the right spine of the tree (we maintain a pointer to this location) and then rotating it upward until the heap property is restored. Armed with your knowledge of amortized analysis, please argue that this takes $\Theta(n)$ total time. [Solution]

Cartesian trees are nearly equivalent to treaps (Section 6.2.5); in fact, treaps are sometimes called Cartesian trees in the computing literature. The distinction lies in the two roles of a BST itself. Recall that a BST can encode either a dictionary or a sequence. In a treap, elements have separate “BST” and “heap” keys, so the “BST” part acts like a dictionary. In a Cartesian tree, elements have only a single key, and the “BST” part of the structure serves to encode a sequence. As opposed to treaps, which we often use as a balancing mechanism for BSTs, it is normal for Cartesian trees to be completely unbalanced. A Cartesian tree built from a sorted sequence is just a single long path.

**Answering Short RMQs.** Suppose we take an RMQ problem, translate it into an equivalent LCA problem, and then translate this back to an equivalent RMQ problem. This might sound ridiculous, but it yields an RMQ problem with very special structure, over a sequence of integers where adjacent elements, being depths of adjacent nodes in a Cartesian tree, differ by exactly 1. As shown in Figure 8.9(a), we can encode this sequence in a compressed binary format where 1 represents an
increase and 0 a decrease. Moreover, the location of the minimum in any sequence can be determined from its binary representation — it is the point at which we have built up the greatest excess of 0s versus 1s. RMQs of length at most \(\log n\) therefore correspond to queries over binary strings of length at most \(\log n\), of which there are at most \(\Theta(n)\) possibilities, allowing us to store all possible results in a lookup table of size \(\Theta(n)\). Combined with the structures we developed earlier for longer RMQs, this gives a solution for both RMQs and LCAs with \(O(1)\) query time, given \(\Theta(n)\) preprocessing time and space. [Further details]

The trick above of building a coarse-grained structure for queries of size at least \(\log n\) alongside a complicated lookup table for smaller queries is somewhat common in the world of advanced data structures. Structures of this sort heavily exploit the fact that word size in the RAM model is at least \(\Theta(\log n)\) bits, allowing the use of lookup tables and other bitwise parallelism tricks to solve logarithmic-sized problems in only \(O(1)\) time. If the reader would like further practice applying these techniques, see problem 136 below.

Problem 131 (Domination Radius). In an array \(A[1\ldots n]\) containing distinct elements, the domination radius of element \(A[i]\) is the maximum value of \(k\) such that \(A[i]\) is the largest number in the range \(A[i-k\ldots i+k]\). We treat \(A[0]\) and \(A[n+1]\) as having value \(+\infty\), so this range never extends beyond the endpoints of \(A\).

(a) How can we compute the domination radius of every element in \(A\) in \(\Theta(n)\) time? [Solution]

(b) Show how to obtain an \(\Theta(n)\)-time solution for problem 51(g). [Solution]

Problem 132 (Min and Max Filtering in Image Processing). In a black and white image stored as an \(n \times n\) binary matrix, erosion is the operation of replacing each pixel with the minimum over some “structuring element” (usually a square, diamond, or circle) centered on the pixel, and dilation involves taking the maximum, giving what is sometimes called the Minkowski sum of the image and the structuring element. As shown in Figure 8.10, these operations are quite useful in binary image processing; they can also be used more generally in non-binary images. Here, we focus on the simple case of a \(k \times k\) square structuring element, with \(k\) odd (a diamond is similar, involving just a 45-degree rotation of a square). Irrespective of \(k\), please show how to perform erosion or dilation in just \(\Theta(n^2)\) time. [Solution]

Problem 133 (Yet Another Way to Build a Min-Queue). Please show how to use a Cartesian tree to build a min-queue (Section 4.3) with operations running in \(O(1)\) amortized time. [Solution]

Problem 134 (Extended RMQs). Let us extend the RMQ problem by considering queries of the form “tell me all elements in the range \(A_i \ldots A_j\) having value at most \(V\)”. Can you solve this problem with \(\Theta(n)\) preprocessing time and space, and only \(\Theta(1+k)\) query time, where \(k\) denotes the number of elements written as output? [Solution]

Problem 135 (Distinct Element Range Queries). Another type of range query that has useful applications in string matching is “tell me all the distinct values appearing in the range \(A_i \ldots A_j\)”. In the RAM model of computation, show how to build a structure to answer such a query in \(\Theta(1+k)\) time, where \(k\) denotes the number of values written as output. You should take \(\Theta(n)\) expected preprocessing time and \(\Theta(n)\) space. As a hint, how can the result of the previous problem help us identify the first occurrence of each distinct value in the range? [Solution]

Problem 136 (Succinct Data Structures for Static Binary Trees). The re-
search field of succinct data structures looks at how to build fully-functional data structures for which the number of bits of memory usage is minimal, down to its leading constant factor. For example, as $n$ grows large, there is a lower bound of roughly $2n$ on the number of bits required to describe an $n$-node binary tree, due to the number of shapes such a tree can assume. A succinct data structure for representing such a tree would allow for easy navigation around the tree (say, stepping from a node to its parent or children in constant time, just as if we had represented the tree in the usual way with pointers), while using only $2n + o(n)$ bits, which converges to $2n$ as $n$ grows large. In this problem, we design such a structure using similar tricks as our RMQ structure above.

(a) Given a length-$n$ binary string, the operation $\text{rank}(i)$ tells the prefix sum of the bits up to position $i$. Please show how to build a data structure using $o(n)$ extra bits of storage (e.g., $O(n/\log n)$ bits) supporting $\text{rank}$ in $O(1)$ worst-case time. Hint: divide the string into blocks and use lookup tables for queries of sub-logarithmic size, as

---

$^4$As discussed in Section 2.6, this number is the $n$th Catalan number, $\frac{1}{n+1} \binom{2n}{n}$, the base-2 logarithm of which converges to $2n$ as $n$ grows large. This logarithm is a lower bound on the number of bits required to represent any static $n$-node tree, just as the logarithm of the number of distinct inputs gives a lower bound on running time for an algorithm modeled as a decision tree (Section 3.3.1).
with the RMQ structure above. You may even need to use a three-level hierarchical decomposition into blocks of blocks rather than just two levels of blocks. [Solution]

(b) Given a length-n binary string, the operation $\text{select}(i)$ tells the index of the $i$th 1 bit.\footnote{If we think of our binary string as encoding a dictionary of integers in the range 0 \ldots n - 1, where a 1 bit corresponds to an integer present in the dictionary, then $\text{rank}$ and $\text{select}$ are the same as the identically-named operations we introduced when we studied BSTs in Chapter 6.} For a more substantial challenge, please show how to build a data structure using $o(n)$ bits of extra storage supporting $\text{select}$ in $O(1)$ worst-case time. As a hint, you will again need to build a hierarchical structure that divides the string into blocks of blocks (using lookup tables to answer queries of sub-logarithmic size on the smallest blocks), and you should define blocks not as regions of the string having a certain length, but as regions of the string containing a certain number of 1 bits. [Solution]

(c) Using the result of problem 89, show how to encode any static $n$-node binary tree using $2n + o(n)$ bits of storage, so that we can still navigate the tree efficiently by stepping from any node to its parent or children in $O(1)$ worst-case time. [Solution]

**Problem 137 (Constant-Time Queries in Static Trees)**. For each problem below, you are given a static $n$-node tree (not necessarily rooted or balanced) as input.

(a) **Range Sum Queries.** Suppose each edge in our tree has an associated weight, provided to you as input. Please show how to preprocess the tree in $\Theta(n)$ time and space so that we can then handle range-sum queries in $O(1)$ worst-case time. The $\text{range-sum}(i, j)$ operation takes pointers to two nodes $i$ and $j$ in the tree and should return the sum of the edge weights on the unique path through the tree from $i$ to $j$.

(b) **Edge-On-Path Queries.** Please show how to preprocess our tree in $\Theta(n)$ time and space so that we can then handle edge-on-path queries in $O(1)$ worst-case time. The $\text{edge-on-path}(i, j, e)$ operation takes pointers to two nodes $i$ and $j$ as well as an edge $e$ (identified by specifying its two endpoint nodes) and determines whether $e$ lies on the unique tree path from $i$ to $j$. [Solution]

(c) **Routing Tables.** Please show how to preprocess our tree in $\Theta(n)$ time and space so that we can handle next-node-on-path queries in $O(1)$ worst-case time. The $\text{next-node-on-path}(i, j)$ operation should tell you which neighboring node of $i$ you should step to if you want to travel from $i$ to $j$. This is one of the fundamental questions facing routers in computer networks, since their job is to forward packets one step closer to their respective destinations. [Solution]

### 8.3.3 Range Minimum Queries in Trees

Range queries in trees are one notch more general than range queries in sequences, since a sequence is just a special case of a tree shaped like a single path. An RMQ in a tree asks for the minimum edge label along the unique path between two nodes $i$ and $j$. To answer such queries in $O(1)$ time, much like before, we build a Cartesian tree. The Cartesian tree of a tree $T$ can be built by selecting the minimum edge in $T$ as the root of the Cartesian tree. Removal of this edge breaks $T$ into two pieces that are then recursively processed. Eventually $T$ is broken down into individual nodes, which become the leaves of the Cartesian tree. An example is shown in Figure 8.11. Just as with a sequence, the answer to an RMQ along the path from node $i$ to node $j$ in $T$ is given by $\text{LCA}(i, j)$ in the Cartesian tree. After building and preprocessing the Cartesian tree for fast LCAs, we can therefore answer RMQs in $O(1)$ time.
The top-down construction method above helps to describe the structure of a Cartesian tree built from a tree, but just as with a sequence, top-down is actually not the fastest method of construction. Consider instead a bottom-up approach starting with the $n$ individual nodes in our original tree $T$. For each edge $e$ in decreasing order by value, we then merge its endpoints into a single subtree rooted at $e$. This slowly builds up larger and larger subtrees, until the very last edge (with minimum value) becomes the final root. When we reach Chapter ?? and discuss minimum spanning trees, you will see that this approach is nothing more than Kruskal’s famous minimum spanning tree algorithm, when run on an input graph that is itself a tree. In fact, since a Cartesian tree encodes precisely the hierarchical sequence of operations performed by Kruskal’s algorithm, we might suggest the term “Kruskal tree” to describe a Cartesian tree built from a free tree. The running time is $O(n \log n)$, since it requires sorting the edges, and this is the best possible in the comparison model, according to the next problem.

**Problem 138 (Lower Bounds for Building Cartesian Trees).** It turns out that while we can build a Cartesian tree from a sequence in $\Theta(n)$ time, it takes slightly longer to build a Cartesian tree from a free tree. Please show that in the comparison model, the lower bound for sorting imposes a worst-case lower bound of $\Omega(n \log n)$ for any algorithm that turns a free tree into a Cartesian tree. [Solution]

**Problem 139 (RMQs in Node-Weighted Trees).** How can we achieve the same performance as above ($O(n \log n)$ preprocessing time, $O(1)$ query time) for RMQ problems on trees where nodes, rather than edges, have associated values? [Solution]

**Applications of Disjoint Sets.** The bottom-up construction algorithm above (i.e., Kruskal’s algorithm) depends on the fast disjoint set data structures described in Section 4.6 to quickly determine, for each edge $e = ij$ we process, the subtrees
containing $i$ and $j$ (find-set operations), and to merge these together into one logical subtree (a union operation). Disjoint set data structures turn out to be useful for a number of other LCA and RMQ problems in trees. They can be used to develop a very simple $O((m + n)\alpha(n))$ time algorithm for the off-line LCA problem, where we are given an $n$-node rooted tree along with $m$ LCA queries in advance [Details]. A more sophisticated generalization can even solve the off-line RMQ problem in a tree (where we are given $m$ RMQ queries in advance) in the same time. [Details]

### 8.3.4 Separator-Based Tree Decomposition

In any $n$-node tree, we can find in $\Theta(n)$ time a separator node\(^6\) whose removal splits the tree into pieces each containing at most $n/2$ nodes [Short justification]. This gives us a natural way to build an $O(\log n)$-depth hierarchical decomposition of any tree: find and remove a separator node (which becomes the root of our decomposition), then recurse on the remaining pieces, as shown in Figure 8.12. This also gives a natural approach for building divide and conquer algorithms on trees, such as our familiar “three step” method for designing range and stabbing query data structures. For example, suppose we want to build an RMQ structure on a free tree with separator node $i$. We can decompose this task into successively more complex structures that:

1. Handle RMQs having node $i$ as one endpoint,
2. Handle RMQs containing node $i$, and
3. Handle arbitrary RMQs.

\(^6\)Note that this result is true for nodes but not edges. A star-shaped tree has no edge whose removal splits the tree into small pieces.
To answer (1), we root the tree at \( i \) and store at every other node \( j \) the value of an RMQ from \( i \) to \( j \) (this is easily done with a single \( \Theta(n) \) time traversal). For (2), we use two calls to the data structure for (1). Finally, for (3) we use divide and conquer: the root of our data structure contains a data structure for (2), and its subtrees are constructed by removing \( i \) and recursively building RMQ structures on the remaining pieces of the tree. In total, this uses \( O(n \log n) \) preprocessing time and space, and it answers RMQs in \( O(\log n) \) time.

If desired, we can improve the query time to \( O(1) \) by preprocessing the structure to support fast LCAs, since the LCA of \( i \) and \( j \) immediately tells us where to find the structure of type (2) we need to query for our answer. For example, a range query over the path from \( f \) to \( i \) in Figure 8.12 would be answered by consulting the type (2) data structure residing in node \( a = \text{LCA}(f, i) \).

**Problem 140 (Stabbing Queries in Trees).** Suppose you are given an \( n \)-node tree as well as a set of \( p \) paths, each specified by its two endpoints. After preprocessing the tree in \( O(p + n \log n) \) time and space, we would like the ability to query any edge \( e \) to determine the \( k \) paths passing through \( e \) in \( O(k + \log n) \) time. Please solve this problem using the three-step divide and conquer approach above. [Solution]

### 8.4 Multi-Dimensional Range Queries

We now move to range queries over \( n \) points in \( d > 1 \) dimensions, where we must contend with somewhat weaker results, thanks to the so-called “curse of dimensionality” — the exponential time or space dependence on dimension that seems unavoidable in most geometric problems. For simplicity, we tend to focus here on static problems as well as orthogonal range queries (over a \( d \)-dimensional rectangle aligned with the coordinate axes). If we are willing to invest huge amounts of space (e.g., \( \Theta(n^d) \) space), we can still solve these problems in logarithmic query time for any \( d = O(1) \) using precomputed lookup tables (problem 120), but this is rarely practical for problems with even moderately large \( n \) or \( d \).

#### 8.4.1 BSP Trees, kd-Trees, Quadtrees, and Recursive Spatial Decomposition

The natural generalization of a binary search tree in higher dimensions is a **binary space partition** (BSP) tree, where we recursively divide space according to straight cuts (lines for \( d = 2 \), planes or hyperplanes for \( d > 2 \)). The root stores the top-level cut (a line, plane, or hyperplane), and its left and right subtrees contain points on both sides of the cut, respectively. Subdivision stops when we reach a region of space containing at most one point, so each leaf represents either an empty region or a single point. The entire structure occupies \( \Theta(n) \) space, and its height is \( O(\log n) \) if we always choose cuts that divide the remaining points in half (random cuts often work just as well, leading to logarithmic height bounds with high probability).

**kd-Trees and Quadtrees.** A particularly common special case is the **kd-tree** (short for \( k \)-dimensional tree), where we cycle through cuts parallel to the coordinate axes. We subdivide at the root according to dimension 1, then at the next level according
to dimension 2, and so on. In three dimensions, for example, we cycle through cuts in $x$, $y$, and $z$. Closely related to the kd-tree is the quadtree, along with its higher dimensional variants (e.g., the octree in 3D), where each node branches $2^d$ ways along all $d$ dimensions simultaneously — into four quadrants in 2D, eight octants in 3D, and so on. However, the extreme branching factor of higher-dimensional relatives of the quadtree unfortunately becomes somewhat prohibitive once dimensionality climbs much higher than $d = 2$ or $d = 3$.

Figure 8.13 shows examples of a kd-tree, a quadtree, and the more general binary space partition tree. We usually consider these structures to be static for simplicity. If they need to be dynamic, amortized rebalancing (Section 6.2.6) is perhaps the simplest technique we can use to maintain balance along with fast amortized runtimes for insertion and deletion of points.

**Range Queries.** It is quite straightforward to perform a query over any range $R$ (not just an orthogonal range) in any tree representing a recursive decomposition of space. Like other range queries, we traverse the “footprint” of the query within the tree, using a recursive traversal that is pruned any time it ventures outside $R$. Every node in the tree represents a region of space (an axially-aligned rectangle, for kd-trees, quadtrees, etc.). If we visit a node $i$ whose associated spatial region intersects $R$, we recursively pass the query along to $i$’s children; otherwise, we prune

---

**Figure 8.13:** Examples of (a) a kd-tree, (b) a quadtree, and (c) a binary space partition tree over a 2-dimensional point set.
8.4. MULTI-DIMENSIONAL RANGE QUERIES

Figure 8.14: Leaf regions in a kd-tree contained in the footprint of an orthogonal range query are highlighted in (a). In (b), we classify the gender of the unknown point incorrectly based on just its nearest neighbor, but correctly by taking a majority vote of several nearest neighbors. Leaf regions in the footprint of a query in a kd-tree for the 7 nearest neighbors of a query point are shown in (c). In (d), a region-based quadtree is used for adaptive mesh refinement, computing the electric potential in the vicinity of a charged disc.

the search and back up. If node $i$ is a leaf, it contains at most a single point, so we test whether this point belongs to $R$. Figure 8.14(a) shows the leaf regions in the footprint of an orthogonal range query in a kd-tree, a relatively simple case since all intersection checks involve axially-aligned boxes.

Even though orthogonal range queries in a kd-tree are perhaps the simplest queries in this setting, they still take $\Theta(k + n^{1-1/d})$ worst-case time for enumeration and $\Theta(n^{1-1/d})$ worst-case time for counting, since the footprint of a query can contain as many as $\Theta(n^{1-1/d})$ leaves whose spatial regions intersect $R$ but whose associated points lie just outside $R$; these regions cause “wasted work”, since their traversal produces no output points. Running time in practice is often faster than the worst case, although it still tends to scale rather poorly with dimension, so one should exercise caution when considering these structures for high-dimensional applications. The quadtree also can perform well in practice in low dimensions, although its theoretical worst-case performance is terrible. For example, even a 2D orthogonal range counting query can take $\Theta(n)$ worst-case time in a quadtree. [Full details]
Nearest Neighbor Queries. Trees representing recursive spatial decompositions are adept at answering nearest neighbor queries, at least in low-dimensional spaces (the approximate nearest neighbor structures we studied in Section 7.5.10 are perhaps better-suited for high-dimensional problems). Figure 8.14(b) shows a prototypical application, where we classify an unknown object based on a majority vote among its nearest neighbors in a set of pre-classified training data.

Searching for the $k$ nearest neighbors of a query point $p$ in a tree $T$ is an example of “branch and bound” search (Section 13.3), where we start traversing all of $T$ but then prune away branches whenever possible. During the traversal, we keep track of the closest $k$ neighbors of $p$ found so far. When examining a node $i$, we always recursively visit the child of $i$ containing $p$ first, as this is more likely to lead us to near neighbors of $p$ more quickly. If we ever reach a node $i$ representing a region of space that is sufficiently far from $p$ (farther than any of the $k$ best neighbors found so far), we prune the search, back up, and try other branches. The footprint of a nearest neighbor search through a kd-tree is shown in Figure 8.14(c). [Further animated details]

Region-Based Versus Point-Based Structures. A tree representing a recursive spatial decomposition is point-based if each cut is chosen to go through a point in a data set, or region-based if cuts are made instead through free space. So far, we have focused on point-based variants of our structures, but region-based decompositions have several useful applications as well. For example, they are often used to approximate some function at varying levels of granularity across space. We might, for instance, try to compress an image using fine-grained subdivision over regions with fine detail, but only coarse-grained subdivision over regions with nearly uniform color. Another prominent application (Figure 8.14(d)) is when we solve physical modeling problems in science and engineering with finite element methods. Here, we discretize space into a fine-grained mesh in regions where our solution exhibits detailed structure, and a coarse-grained mesh in regions where the solution is less interesting (e.g., nearly constant or linear).

8.4.2 Priority Search Trees

A few pages ago, we learned about Cartesian trees and highlighted their similarity with treaps. We now introduce a third structure in this family, the priority search tree (PST). In $O(k + \log n)$ time, the PST can enumerate the $k$ points in any “three-sided” query region in the plane — extending to infinity in one direction. We assume the region extends to infinity in the $-y$ direction, as shown in Figure 8.15, although we can easily handle the other three directions by building three additional PSTs on rotated copies of our point set.

To build a priority search tree in $O(n \log n)$ time (Figure 8.15), we store the point with minimum $y$ coordinate at the root, then we split the remaining points on the median $x$ coordinate, $x_{mid}$, into left and right subtrees, which are then recursively constructed. The value of $x_{mid}$ is stored at the root, so we can remember the split point between the left and right subtrees. The structure occupies linear space and its height is $O(\log n)$. Although a PST is not exactly a BST, it retains much of the same functionality as if we had built a BST on our points, keyed on $x$ coordinate. In particular, we can still binary search for a point based on its $x$ coordinate in
8.4. MULTI-DIMENSIONAL RANGE QUERIES

Figure 8.15: (a) An example of a 3-sided query over the region \([x_1, x_2] \times (-\infty, y]\), and (b) the construction of a priority search tree: the point with minimum \(y\) coordinate, \(M\), is placed at the root and augmented with the median \(x\) coordinate, \(x_{\text{mid}}\). The remaining points are evenly split using \(x_{\text{mid}}\) into left and right subtrees (points with \(x = x_{\text{mid}}\) can also be evenly split between the two). In (a), the footprint of a query is shaded — it looks like the standard footprint of a one-dimensional query, except due to the heap property in \(y\), only the “top” parts of the subtrees matching the query range in \(x\) are relevant. Enumeration is pruned any time it ventures below these shaded regions.

\[O(\log n)\] time. Moreover, the PST satisfies the heap property with respect to \(y\) coordinate, so like the treap and Cartesian tree, it can also be considered a hybrid between a type of BST and a heap.

To enumerate the points in a three-sided region \([x_1, x_2] \times (-\infty, y]\) with a PST, we again recursively visit the footprint of the query region. We prune the search any time we step out of the footprint (shaded in Figure 8.15(a)) by landing on a node with position above the \(y\) cutoff for our query; due to the heap property, the entire subtree of this node must also lie above \(y\). It is easy to show that the entire process takes \(O(k + \log n)\) time. [Further details]

PSTs Versus Persistent BSTs. A persistent BST gives us another nice way to achieve the same functionality as the PST. In fact, the persistent BST is even more powerful: not only does it support \(O(k + \log n)\) time range enumeration like the PST, but it also supports \(O(\log n)\) time range counting, a feature the PST lacks. Suppose we insert our \(n\) points in increasing order of \(y\) into a persistent balanced BST keyed on \(x\). In the middle of this process, the points in the BST will look like those in the shaded region in 8.15(a), as the horizontal line at position \(y\) continually sweeps upward. To issue a query over the range \([x_1, x_2] \times (-\infty, y]\), we simply query our BST over the range \([x_1, x_2]\) as of the historical point in time when our insertions reached \(y\).

Problem 141 (From Three-Sided to Four-Sided Queries). Using the same three-step divide and conquer process we used to design the interval tree, please show how to build a simple data structure out of PSTs (or persistent balanced BSTs) that can enumerate the points in a four-sided region \([x_1, x_2] \times [y_1, y_2]\) in \(O(k + \log n)\) time.

DRAFT. Not for redistribution. Copyright © Brian C. Dean
Figure 8.16: A range tree built from a two-dimensional point set (a), with elements outlined in bold matching an orthogonal range query. Each subtree root is augmented with a $y$-ordered list (i.e., a one-dimensional range tree) of the points in that subtree, and the lists of the subtrees matching our query in $x$ are binary searched to filter out the points matching the query also in $y$. In (b), a trade-off curve for a multi-objective optimization problem is shown, with the region dominated by point $p$ shaded.

Using persistent balanced BSTs instead of PSTs, discuss how this structure can also count the points in a four-sided region in $O(\log n)$ time. How much space does your structure require? [Solution]

Problem 142 (Path Stabbing in Trees, Revisited). Please solve problem 140 using a priority search tree. [Solution]

We might also recommend that the reader revisit problem 134, as its solution may be more familiar and intuitive now that we understand priority search trees.

### 8.4.3 Range Trees

In this section we build a $d$-dimensional orthogonal range searching structure inductively, by iteratively “composing” one-dimensional structures in a $d$-level fashion. Using this technique, we answer a $d$-dimensional range query by issuing $O(\log n)$ range queries in $d - 1$ dimensions, which in turn are broken up into $O(\log^{2} n)$ range queries in $d - 2$ dimensions, and so on, giving $O(\log^{d} n)$ time in total.

A one-dimensional static range tree is a sorted array, which can easily answer range queries via binary search. To build a static range tree from $n$ points in $d > 1$ dimensions, we sort the points along dimension 1 and build a tree atop this sorted array. As shown in Figure 8.16, we augment each tree node with a $(d-1)$-dimensional range
tree (built from just dimensions 2...d) for the points in its subtree. This might look worrisome in terms of space; however, each point stored in a d-dimensional range tree appears in only $O(\log n)$ of these $(d-1)$-dimensional structures, and hence in only $O(\log^2 n)$ $(d-2)$-dimensional structures, and so on, leading to a total space bound (as well as a bound on construction time) of $O(n \log^{d-1} n)$.

To answer a d-dimensional orthogonal range query, the points matching our range in dimension 1 can be represented by at most $2 \log n$ subtrees, as shown in Figure 8.16; this is the same idea we have now seen many times throughout the chapter. To the roots of those subtrees, we then issue $(d-1)$-dimensional range queries to further filter these points according to dimensions 2...d. Query time is $O(\log^d n)$ for counting or $O(k + \log^d n)$ for enumeration. We can further reduce the $\log^d n$ term to $\log^{d-1} n$ using a trick known as fractional cascading, where we use additional pointers to reduce the number of binary searches at the lowest level from $O(\log n)$ down to one. [Details]

**Problem 143 (Finding Non-Dominated Points).** Suppose we have an optimization problem in which our goal is maximize two objectives: (i) time savings, and (ii) cost savings. Unfortunately, it is often not possible to maximize both simultaneously, owing to a “trade-off” curve (Figure 8.16(b)), governing the best we can do for one objective with respect to the other. Consider n distinct potential solutions, each given by a point of the form (time savings, cost savings). Point $p = (x_p, y_p)$ dominates point $q = (x_q, y_q)$ if $x_p \geq x_q$ and $y_p \geq y_q$. In this case, $p$ is just as good as $q$ in both objectives (and strictly better in at least one, since $p \neq q$), so we can safely discard $q$ from consideration. We would like to find all non-dominated solutions, also known as Pareto-optimal solutions in the literature.

(a) As a warm-up, please show how to find all the non-dominated points in set of n points in the 2D plane in $O(n \log n)$ time. Can you think of several different ways to solve this problem? Does a Cartesian tree of some form expose the set of all non-dominated points in a convenient fashion? [Solution]

(b) The problem above naturally generalizes to $d > 2$ dimensions. Please show how to solve it using range trees in $O(n \log^{d-2} n)$ time. [Solution]

**Problem 144 (Range Queries on Subarrays).** In this problem, we highlight applications of two-dimensional range queries and range trees.

(a) **Counting Containing and Contained Intervals.** Given a static set of n intervals, please show how to preprocess it in $O(n \log n)$ time and space so that for any query interval $I$, we can count the number of intervals in our set containing $I$, the number of intervals intersecting $I$, and the number of intervals contained within $I$, all in $O(\log n)$ time. If we want to enumerate the k intervals answering any of these queries, how can we do this in $O(k + \log n)$ time without using range trees? [Solution]

(b) **Counting Distinct Elements.** Given an n-element array, please show how to preprocess it in $O(n \log n)$ time and space so that we can subsequently count the number of distinct elements in any subarray in $O(\log n)$ time. Hint: see problem 135. [Solution]

(c) **Subarray Order Statistics.** Given an n-element array, please show how to preprocess it in $O(n \log n)$ time and space so that we can subsequently query for the rank-k element in any subarray in $O(\log^2 n)$ time. [Solution]
CHAPTER 8. RANGE QUERIES

8.5 Dynamic Trees

We are now experts in manipulating sequences. By storing a sequence in a balanced BST, we can perform range queries and updates in $O(\log n)$ time. As shown previously in Section 6.1.7 we can cut a sequence into two pieces in $O(\log n)$ time using the split operation, and we can link two sequences together end-to-end in $O(\log n)$ time using the join operation.

Phrased more in the context of a graph, these results let us implement a dynamic path — an $n$-node path with values assigned to its nodes or edges that supports all of the following in $O(\log n)$ time:

- Range queries (e.g., “find the minimum value along the subpath from node $i$ to node $j$’),
- Range updates (e.g., “increase every value on the subpath from node $i$ to node $j$ by 3”),
- Cutting (e.g., “split a dynamic path into two separate dynamic paths by removing edge $e$”), and
- Linking (e.g., “join two dynamic paths end-to-end by adding a new edge $e$”).

Data structures for dynamic trees go one step beyond paths to support these operations (range queries, range updates, linking, and cutting) on trees. Most dynamic tree implementations also support a find-root($i$) operation that identifies some canonical element (which we call the root) of the tree containing node $i$. Just like the find-set operation in a disjoint set data structure allows us to test whether two elements belong to the same set, this allows us determine if nodes $i$ and $j$ belong to the same tree by testing if find-root($i$) = find-root($j$).

Dynamic trees are a key building block for several highly-efficient algorithms, for example in some of the fastest known algorithms for network flows (Sections ?? and ??) and shortest paths (Section ??), and in data structures for maintaining dynamic connectivity in graphs (Section ??).

8.5.1 Heavy-Light Decomposition

If we want the ability to perform fast range queries and updates but not linking and cutting (i.e., if we are working with a static tree shape, rooted arbitrarily), we can use a heavy-light decomposition, shown in Figure 8.17(a). An edge from parent $i$ to child $j$ is “heavy” if $j$’s subtree contains at least half the nodes in $i$’s subtree, and “light” otherwise. Since each node has at most one heavy edge descending from it (and therefore at most two incident heavy edges), the heavy edges form a collection of disjoint paths, which we store individually in balanced BSTs.

Within a heavy path, range queries or updates take only $O(\log n)$ time. Moreover, since stepping down through a light edge halves the size of our current subtree, an arbitrary $i$-$j$ query path contains at most $2\log n$ such edges — up to $\log n$ from LCA($i, j$) to $i$, and another $\log n$ from LCA($i, j$) to $j$. Any path therefore decomposes into up to $O(\log n)$ individual light edges connecting $O(\log n)$ heavy path segments, giving a query / update time of $O(\log^2 n)$. 

DRAFT. Not for redistribution. Copyright © Brian C. Dean
8.5. DYNAMIC TREES

![Diagram of tree structures](image)

**Figure 8.17:** (a) A heavy-light decomposition of a rooted tree, where an i-j query path decomposes into $O(\log n)$ individual light edges plus $O(\log n)$ heavy (bold) path segments. (b) The process of linking two trees by performing splits and joins in their Euler tours, stored in balanced BSTs. (c) A tree represented as a collection of node-disjoint paths, which allow easy range queries/updates from $i$ to $j$, but need to be reconfigured to allow easy queries/updates from $i$ to $k$.

### 8.5.2 Euler Tour Trees

If we want the ability to perform linking and cutting effectively, instead of range queries and updates, the Euler tour tree (ET-tree) may suit our needs. The ET-tree offers an elegant way to encode an arbitrary tree $T$ within a balanced BST, by taking the Euler tour traversal sequence from $T$ and storing it as a sequence in a balanced BST. We then use **split** and **join** to perform linking and cutting in $O(\log n)$ time (amortized, if we use splay trees), as demonstrated in Figure 8.17(b).

**Problem 145 (Path Range Queries with Edge Updates).** ET-trees can support some types of path range queries, as long as updates occur on single nodes or edges instead of entire paths. Given a static tree with edge weights, please describe a simple implementation of both **range-sum$(i, j)$** (which finds the sum of edge weights along the path from $i$ to $j$) and **edge-update$(e, v)$** (which updates the weight of edge $e$ to a new value $v$) both in $O(\log n)$ worst-case time. What modifications to your structure are necessary if nodes rather than edges are weighted? [Solution]

### 8.5.3 Node-Disjoint Dynamic Paths

As shown in Figure 8.17(c), suppose we store a tree using an arbitrary collection of node-disjoint dynamic paths. A range query or update from $i$ to $j$ is now easy if $i$ and $j$ belong to the same path. If not, we reconfigure our decomposition so $i$
and \( j \) become part of the same dynamic path, after which the operation is easy to complete. This involves performing an appropriate series of \textit{cut} and \textit{link} operations among our paths to build up one long path containing \( i \) and \( j \) (sometimes called “solidifying” the path from \( i \) to \( j \)). Splay trees are particularly nice to use for this purpose, not only because they support easy linking and cutting, but since one can extend their original analysis to show that this leads to an \( O(\log n) \) amortized running time for all dynamic tree operations: range queries and updates, linking and cutting, \textit{find-root}, and more. [Analysis]
Consider how often you search for phrases in a text document, messages in your inbox, files on your desktop, or web pages on the Internet. All of these involve the use of fast data structures for text and document searching, which have become a crucial part of the algorithmic technology we now use on a daily basis to help us stay afloat in a sea of data. This chapter describes the most common techniques used for text searching, with occasional short detours to mention related topics such as regular expressions and parsing, text compression, inexact search, predictive text completion, and spell checking.

The most basic problem in this domain is the classical *string matching* problem, asking us to find every occurrence of a pattern string $P[1...m]$ within a larger text string $T[1...n]$. The problem is easy to solve in $O(mn)$ time by a “brute force” test of the pattern against every offset within the text. That is, we test if $P[1...m]$ matches $T[1...m]$, then we test if $P[1...m]$ matches $T[2...m+1]$, and so on, spending $O(m)$ time at each of the $n - m + 1 = O(n)$ offsets within the text. If we are lucky, the time spent at each offset is much smaller than $\Theta(m)$, since we can stop scanning the instant we encounter a non-matching character. Nonetheless, this approach is typically too slow to solve large-scale problems in practice, since $n$ is in the billions or worse for many applications (e.g., searching the DNA sequence of the human genome, or searching collections of pages on the World Wide Web). Fortunately, we will see several approaches in this chapter that require time proportional roughly to the length of the pattern, not the text. That is, we can search for a needle in a preprocessed haystack using time proportional to the size of the needle, not the haystack.

The content in this chapter is related to several other topics we cover later in this book. For example, since a text search over a large document collection can potentially return millions of results, we often identify the “best” search results by analysis of citations / links between documents, as we discuss when we study *centrality* (Section ??) and *ranking* (Section ??). Textual information is also frequently the subject of different applications in data mining and machine learning (Chapter ??). The interested reader may also want to independently investigate the many advanced topics in computational text and document analysis that lie beyond the scope of this book, such as natural language processing, sentiment analysis, machine translation, speech processing, and much more.
CHAPTER 9. STRING AND DOCUMENT SEARCHING

9.1 Approaches Based on Hashing

Recall from Section 7.5.5 that we often hash a large object down to a smaller integer “fingerprint” that serves as a proxy for the larger object, allowing fast approximate comparison between two objects by comparing their fingerprints instead. By hashing strings to small integer fingerprints, this approach gives us several effective methods for string and document searching. Throughout this section, we assume all strings are arrays of integers, so we can work within the RAM model (as in many other algorithmic domains, string matching algorithms can be quite sensitive to our underlying model of computation).

9.1.1 Hashing Substrings

To find all matches of a pattern \( P[1 \ldots m] \) within a longer text \( T[1 \ldots n] \), we can compare a hash of \( P \) with the hash of a length-\( m \) window that slides through \( T \), an approach named Karp-Rabin string matching after its original authors. The key to making this approach run quickly is the use of a hash function that we can update in \( O(1) \) time when the window slides over by one position. This is perhaps most easily accomplished if we use a polynomial hash function (Section 7.5.5), where we treat the characters in a substring \( T[i \ldots j] \) as coefficients of a large polynomial. A hash \( h_{ij} \) of the substring is given by evaluating the polynomial at a random value \( x \in \{0, 1, \ldots, p - 1\} \) chosen during initialization, where \( p \) is a large prime:

\[
h_{ij} = (T[i]x^{j-i} + \ldots + T[j-2]x^2 + T[j-1]x + T[j]) \mod p.
\]

All of our arithmetic is done modulo \( p \) to keep our numbers small enough to fit into a single machine word. When we slide the window by one character to \( T[i+1 \ldots j+1] \), we update its hash accordingly by subtracting the first term, multiplying by \( x \), and adding a new final term:

\[
h_{i+1,j+1} = ((h_{ij} - T[i]x^{j-i})x + T[j+1]) \mod p.
\]

This can be evaluated in \( O(1) \) time if we precompute \( x^{j-i} \mod p \), so the entire algorithm runs in \( \Theta(n) \) time.

Like any method that relies on hashing to test equality between complex objects, this is a “Monte Carlo” randomized algorithm that could output false positive matches in the event of a hash collision. However, by setting \( p \) sufficiently large, we can make such collisions extremely unlikely. Recall from Section 7.5.5 that the probability of a collision is at most \( (m - 1)/p \leq (n-1)/p \leq n/p \), so by a union bound over all \( n - m + 1 \leq n \) offsets of the window, the probability of any false positive match is at most \( n^2/p \). Setting \( p \geq n^c \) for a large constant \( c \), we therefore achieve correct operation with high probability. If desired, we could also verify each offset selected as a match in \( O(m) \) extra time per offset.

Hashing Arbitrary Substrings with Prefix Hashes. A generalization of the approach above allows us to hash a substring of any length in \( O(1) \) time, after first computing the hash of each prefix \( T[1 \ldots j] \) as a preprocessing step. This is quite similar to the way we answered range sum queries in Section 8.1.1 by taking the difference of two precomputed prefix sums. After spending \( \Theta(n) \) time precomputing
9.1. APPROACHES BASED ON HASHING

the hash $h_j$ of each prefix $T[1 \ldots j]$,

$$h_j = \left( T[1]x^{j-1} + T[2]x^{j-2} + \ldots + T[j-1]x + T[j] \right) \mod p$$

$$= (xh_{j-1} + T[j]) \mod p,$$

as well as a lookup table of $x^0 \ldots x^{n-1} \mod p$, the hash of an arbitrary substring $T[i \ldots j]$ is then easy to compute in $O(1)$ time:

$$h_{ij} = (h_j - h_{i-1}x^{j-i+1}) \mod p.$$

We can even make this dynamic by storing the text as a sequence in a binary search tree or one of its relatives. By augmenting each node with a hash of the substring represented by its subtree, we can support insertion, deletion, and modification of a character, as well as computation of any substring hash, all in $O(\log n)$ time.

9.1.2 Inverted Indices

The fundamental job of a search engine is to maintain an indexing structure over a large document collection that allows fast retrieval of documents matching a query word or phrase. Typically, we store each unique word $w$ in a large hash table, augmented with a list of every occurrence of $w$ in the document set. This is sometimes known as an “inverted” index because it maps words to documents instead of the document-to-word mapping we get from just the documents themselves. For example, our index might contain:

- ‘algorithm’: (5, 3), (5, 7), (5, 13), (6, 2), (6, 12), ...
- ‘string’: (6, 11), (7, 15), ...

That is, ‘algorithm’ occurs as (document 5, word 3), as (document 5, word 7), and so on. If the user searches for ‘string algorithm’, we can do hash lookups on ‘string’ and ‘algorithm’ to quickly locate their respective lists of occurrences, which we then intersect to find occurrences of the entire phrase. In the example above, ‘string algorithm’ occurs in document 6, words 11-12.

We typically number the documents alphabetically or in order by importance (e.g., see Section ??). By keeping lists of occurrences sorted in this order (by document number then by word position), we can often avoid computing the entire result set for a query. For example, if we only need to display the top 10 results for ‘string algorithm’, we may only need to compute a very small prefix of the intersection of the lists above, continuing the computation only if the user requests more results.

More sophisticated queries might involve Boolean expressions like (‘string’ or ‘document’) and ‘matching’. To answer such a query, we would need to take the union of the occurrences of ‘string’ and ‘document’, intersected with the occurrences of ‘matching’. Researchers in the domain of information retrieval have studied numerous heuristics for performing these sorts of operations quickly.

**Problem 146 (Double Binary Search for Sorted Array Intersection).** To give one example of the sorts of algorithmic refinements we can consider during document searching, suppose you search for a phrase ‘A B’, where ‘A’ occurs $n$ times and ‘B’ occurs
Figure 9.1: A snapshot in the middle of Morris-Pratt string matching. Upon finding a mismatch at position \( j \) in the text, we shift the pattern forward according to the longest border of the partial match of the pattern. A border of a string is a substring that occurs as both a prefix and a suffix, so when we advance the pattern according to a border of its partial match, the part of the pattern before \( j \) will still match. The next comparison will be ‘y’ versus ‘y’, causing \( j \) to advance.

Hash-based indices are also useful for approximate matches of a pattern within a text, such as with the popular “BLAST” algorithm for searching long DNA sequences. The key idea here is that an approximate match usually contains an exact match of at least some smaller part of the pattern, so we can consult a hash-based index for short substrings of the pattern to see where they occur in the text, building outward from these seed locations to conduct more elaborate searches for approximate matches. We say more about approximate string matching later in this chapter.

### 9.2 Building a String Matching Automaton

In this section we describe a very simple linear-time string matching technique, due to Morris and Pratt, that preprocesses a length-\( m \) pattern in \( \Theta(m) \) time and space so it can be subsequently matched against any length-\( n \) text in \( \Theta(n) \) time. Unlike some of the other techniques in this chapter, the Morris-Pratt algorithm easily runs in linear time even in the comparison-based model.

Recall how string matching by “brute force” loops over all \( n - m + 1 \) possible offsets of the pattern against the text, checking each one for a match in \( O(m) \) time. The
Morris-Pratt algorithm cleverly improves this approach by using information in each partial match to advance the pattern more quickly. As shown in Figure 9.1, suppose we are checking offset $i$, and that we have currently built a partial match up to position $j$ in the text, where we finally find a disagreement. We can avoid backing up the comparison index $j$ if we advance the pattern according to a border of its partial match so far — a substring that is both a prefix and a suffix. If the partial match has several valid border strings, we choose the longest one, since this advances the pattern by the smallest amount (choosing a different border might advance the pattern too much and skip over valid matches). Fortunately, there is a simple $\Theta(m)$ algorithm for precomputing the length of the longest border of every prefix of the pattern [Details]. After running this during preprocessing, every step of the Morris-Pratt algorithm takes $O(1)$ time and either matches a character (increasing $j$, leaving $i$ unchanged) or advances the pattern (increasing $i$, leaving $j$ unchanged). There can be at most $n$ steps of either type, so the entire algorithm runs in $\Theta(m + n) = \Theta(n)$ time.

An alternative outlook on the Morris-Pratt technique is to view it as a $\Theta(m)$ method for transforming the pattern into an automaton capable of matching against any length-$n$ text in $\Theta(n)$ time. As shown in Figure 9.2, the automaton has a node, or “state”, for each character in the pattern, representing a partial match up to but not including that character. We start in the leftmost state, having matched no characters yet. Once built, the automaton gives us a purely mechanical means of performing string matching. Starting from the first character in the text ($j = 1$), we simply follow the appropriate edges in the automaton depending on matches or mismatches between our current state and the current character in the text. Matches cause $j$ to increment and move us ahead one state, and mismatches cause $j$ to remain unchanged while we back up to some earlier state (corresponding to an advance of the pattern according to the longest border of its partial match).

The Morris-Pratt algorithm, and a slightly optimized variant known as the Knuth-Morris-Pratt algorithm [Details] are both popular methods for solving string matching problems in practice. In addition to finding all occurrences of a pattern within a text, the (Knuth-)Morris-Pratt algorithm tells us, for each index $i$ within the text,
CHAPTER 9. STRING AND DOCUMENT SEARCHING

the length of the pattern prefix \( P[1 \ldots] \) that partially matches \( T[i \ldots] \). This extra information can be quite useful, as the following problem illustrates.

Problem 147 (Allowing One Mismatching Character). How can you build on the (Knuth-)Morris-Pratt approach to find in \( \Theta(n) \) time all indices at which a pattern matches exactly or with at most one mismatching character? [Solution]

9.2.1 Regular Expression Matching

String matching with an automaton can be generalized to accommodate more sophisticated types of patterns called regular expressions that are quite common in the world of software development. They are prominently supported in most modern programming languages (e.g., Perl, Python, Javascript), and often used in scripting applications with command-line utilities like grep, which prints the lines of a text file matching a regular expression.

Regular expressions offer two powerful features: \( x|y \) matches either \( x \) or \( y \), and \( x^* \) matches zero or more repetitions of \( x \). These can be combined to build surprisingly expressive patterns. Here are few simpler examples:

- \( \text{alg}((\text{orithm})|(\text{ebra})) \) matches ‘algorithm’ or ‘algebra’.
- \( (01)|(10)|(0(10)^*)|(1(01)^*) \) matches any binary string without repeated digits.
- \( 01([1-9][0-9]^*) \) matches a nonnegative integer, where \([0-9]\) is shorthand for \(011213|4|5|6|7|8|9)\), matching any character in the range \(0\ldots9\).
- \( [0-9]^*7[0-9]^* \) matches any string of digits containing a seven.
- \( ((2[0-3])|([0-1][0-9])):[0-5][0-9] \) matches a 24-hour time specification like 23:32.

With some care, a regular expression can be automatically translated into an automaton that enables simple mechanical pattern matching against any text. As shown in Figure 9.3, each state in the automaton has an outgoing edge corresponding to each character from the alphabet. These labels guide our movement from state to state as we scan through the text, and if we end in a “match” state, we deem that the pattern has matched the text. The only added complication with automata generated from regular expressions is that they might be nondeterministic. In a deterministic automaton (e.g., generated by the (Knuth-)Morris-Pratt algorithm), we are always in exactly one state at any point in time, allowing us to match against a length-\( n \) pattern in just \( \Theta(n) \) time, each transition taking just \( O(1) \) time. Nondeterminism allows us to be in multiple states at any point in time\(^1\), as a consequence of states from which “\( \varepsilon \)-transitions” emanate (\( \varepsilon \) representing the empty string). Upon entering such a state, we immediately branch and follow all of its outgoing edges simultaneously, without consuming a character from the text. In an automaton with \( m \) states, this means each “step” might need to keep track

\(^1\text{This is the same concept we discussed back in Section 1.6.1, when we defined the class NP of problems that could be solved in polynomial time using a nondeterministic algorithm.}\)
of up to \( m \) separate steps taken from all the possible states in which we could reside, raising the worst-case running time for matching against a length-\( n \) pattern to \( \Theta(mn) \). However, in practice, matching is still usually quite fast.

### 9.2.2 Beyond Regular Expressions

Although regular expressions give us a substantial amount of flexibility in describing complex textual patterns, in the broader field of parsing their capabilities are actually quite limited. More sophisticated approaches are required for, say, compiling a computer program or interpreting the structure of a sentence written in some natural language like English — applications that generally go beyond the scope of this book. See problem 217 for an example of parsing using a “context free” grammar, a system that is slightly more expressive than a regular expression.

Regular expressions are also some of the earliest objects typically encountered in a course on the theory of computation. We can abstractly view an algorithm for solving a yes/no decision problem as a pattern matching procedure: we feed in a string of 0’s and 1’s encoding its input, and the algorithm should declare a “match” for all the input strings that represent “yes” instances. In this context, the regular expression and its associated automaton can be regarded as a simplistic model of computation. For example, an algorithm in this model can solve the problem of recognizing binary strings with no successively repeated digits (Figure 9.3(a)). However, this model has its limitations. For instance, since it has limited “memory”, no finite regular expression can encode an algorithm that recognizes binary strings with the same number of 0’s and 1’s. For that task and many others, we need to turn to more powerful models of computation, such as the Turing machine.

### 9.3 Suffix Arrays

The preceding methods for string matching over a length-\( n \) text in \( \Theta(n) \) time may still be of limited use when \( n \) is huge. As an alternative, we will now learn how to preprocess a large text so that we can perform pattern matching in time roughly
proportional to the length of the pattern, with little or no dependence on the length of the text. We investigate two closely-related data structures for this problem, the suffix array in this section and the suffix tree in the next section.

A suffix of a text \( T[1 \ldots n] \) is a substring of the form \( T[i \ldots n] \). Suppose we take all \( n \) suffixes of a length-\( n \) text and arrange them in sorted order. To represent this ordering, we don’t need to explicitly store the suffixes — we only need to maintain an array specifying the starting index of each one, since the suffixes themselves can easily be reconstructed if needed by indexing into the original text. Pictured in Figure 9.4(a), this structure is called a suffix array. Being nothing more than an \( n \)-element integer array, it occupies just \( \Theta(n) \) space.

After building a suffix array, we can search for all occurrences of a length-\( m \) pattern in only \( O(m \log n) \) time. Due to the sorted nature of the suffix array, all occurrences of the pattern will occur in a contiguous block within the array, so we can represent the set of all occurrences by reporting the upper and lower endpoints of the block. If \( -\infty \) and \( +\infty \) denote fictitious characters that are respectively less than and greater than all other characters in our alphabet, then a binary search for the pattern followed by \( -\infty \) will land on the upper endpoint, and a binary search for the pattern followed by \( +\infty \) will land on the lower endpoint. While binary search in a length-\( n \) array normally takes just \( O(\log n) \) time, here each step involves a comparison against the pattern, and hence takes \( O(m) \) time.

**Problem 148 (Finding Matches in Order).** The block of consecutive indices in a suffix array matching a pattern is not necessarily in sorted order. If the user expects to receive matches listed in order of occurrence within the text, we could of course sort the block, but this would be excessive if there are millions of matches and the user only wants to see the first few of them. Please show, using appropriate data structures taking only \( \Theta(n) \) space and \( O(n \log n) \) preprocessing time, how to read off the indices of the first \( k \) matches from the text in only \( O((m + k) \log n) \) total time. \[Solution\]

**Problem 149 (Longest Common Prefixes).** A suffix array is often augmented with an array containing the lengths of the longest common prefixes between adjacent suffixes, as shown in Figure 9.4(a). In addition to helping us later build a suffix tree out of a suffix array, this allows us to reduce the time required to search for a length-\( m \) pattern to only \( O(m + \log n) \). Please show how longest common prefixes allow us to achieve this speedup. You may need to make use of additional data structures taking \( \Theta(n) \) space and preprocessing time to build. \[Solution\]

### 9.3.1 Building a Suffix Array

The time required to build a suffix array is the same as the time required to sort the characters in our text: \( \Theta(n \log n) \) worst-case in the comparison model, and \( \Theta(n) \) on a RAM if the characters are sufficiently small integers (say, of magnitude at most \( n^c \) for \( c = O(1) \), which we can radix sort in linear time). We clearly cannot build a suffix array any faster, since the suffix array tells us precisely the sorted ordering of the characters in the text. On the other hand, we will see in this section how to build a suffix array in \( \Theta(n) \) time after the characters in the text are sorted. Note that this is much faster than the “naïve” approach of just sorting suffixes directly, since that can take up to \( \Theta(n^2 \log n) \) time, as each comparison in the sort involves an \( O(n) \)-time comparison between two suffixes.
Figure 9.4: A suffix array for the string ‘dereference’ is shown in (a), augmented with an array of longest common prefixes. In (b), we show an equivalent sorted list of rotations of the text, where the first column corresponds to the suffix array, and the last to the Burrows-Wheeler Transform.

After sorting the characters in the text, we can build a suffix array in linear time using a divide-and-conquer approach known as the skew algorithm due to its use of asymmetry. It recursively builds a suffix array using only suffixes starting at indices that are not multiples of 3, and then uses this to quickly determine the correct ordering of the remaining suffixes. The two sets of suffices (those starting at indices that are multiples of 3, and those not) are then merged to obtain a suffix array for the entire text, exploiting the asymmetry of the decomposition in a clever way to simplify the merging process. [Full details]

If we want a suffix array augmented with longest common prefixes, the skew algorithm can generate these during its execution. Alternatively, there is a simple method to build the longest common prefix array in $\Theta(n)$ time afterwards from just the suffix array and the original text. [Details]

### 9.3.2 The Burrows-Wheeler Transform

A close relative of the suffix array is the Burrows-Wheeler Transform (BWT). As shown in Figure 9.4(b), sorting the rotations of a text is equivalent to sorting suffixes. The first column of the result gives the suffix array. The string given in the last column is the BWT of our text. It is easy to see from the figure how one can compute the BWT in $\Theta(n)$ time starting from a suffix array. Remarkably and quite surprisingly, however, the BWT can also be inverted: given only the string representing the BWT, we can recover the original text in just $\Theta(n)$ time\(^2\). [Details]

\(^2\)As a small technicality, we need to terminate our string with a “dummy” end of string character (typically denoted $\$$) for the BWT inversion to work properly.
The main application of the BWT is in data compression, since it maps a length-$n$ string to another length-$n$ string in which similar characters tend to be grouped together, making it easier to compress. For example, we can use:

- *Run-length* encoding, replacing long runs of the same character with a single copy plus a counter, or

- *Move-to-front* encoding, replacing each re-occurring instance of a character with a (presumably short) binary code giving a count of the number of distinct characters in the span since its previous occurrence (i.e., the number of characters it would advance if it were moving to the front of a list of distinct characters ordered by most recent time of occurrence).

Both encodings are easy to apply and also decode quickly, giving an effective and fast encoding / decoding scheme when combined with the BWT and its inverse. The suffix array also tends to group similar characters together, but unlike the BWT, the suffix array does not help much with compression since it requires the full original text to accompany it. Researchers have also studied ways to do fast pattern searching natively in compressed strings, with the BWT usually being an important ingredient — see the endnotes for further references.

### 9.4 Tries and Suffix Trees

A *trie* (pronounced “try”) is a tree encoding a set of sequences “vertically” in its root-to-leaf paths. Figure 9.5 shows a small example. Another good example is the radix tree (Sections 5.5.1 and 7.4), which encodes numbers vertically in a digit-by-digit fashion. Tries are convenient data structures to use for matching a pattern against a set of strings, where the pattern arrives one character at a time, guiding...
9.4. TRIES AND SUFFIX TREES

Figure 9.6: Examples of (a) an uncompressed suffix tree, and (b) a compressed suffix tree, both encoding the string ‘dereference$’. The compressed tree only has 4 non-leaf nodes, each augmented with its “letter depth”, giving the length of the substring of the text represented by the path down to the node. Substrings on edges are concisely encoded, as shown in (c), by starting and ending indices within the text.

our movement down the tree. Much like the automata we discussed earlier, each successive character tells us the next child edge to follow. We succeed in matching a string when we reach a leaf\(^3\). If our pattern carries us down to an internal node \(i\), then the leaves in \(i\)’s subtree tell us all the strings matching the pattern so far.

A suffix tree is a trie built from all the suffixes of our text, as shown in Figure 9.6(a). Children of each node are typically stored in sorted order in a linked list or array. By terminating our text with a dummy end-of-string character, we can ensure a one-to-one mapping between suffixes and leaves, and we label each leaf with the starting index of its corresponding suffix. To search for a length-\(m\) pattern, we follow the characters of the pattern down the tree. If we end at node \(i\) after successfully matching the entire pattern, then the leaves in \(i\)’s subtree tell us the indices of all the matches in the text.

Searching in a suffix tree is quite fast, although the exact running time depends on our computational model and other details. If we assume our strings contain

\(^3\)We usually assume that each string ends with a special dummy end-of-string character (typically denoted $), so that there will be a nice one-to-one correspondence between strings and leaves. Otherwise, a string might end at an internal node, like ‘to’ in Figure 9.5. The only situation where this is not necessary is when the set of strings is prefix-free, with no string being a prefix of any other. Huffman codes (Section 10.2) are a good example.
characters drawn from an alphabet of constant size (a common assumption with suffix trees), then each step down the tree takes constant time, so it takes $O(m)$ time to match a length-$m$ pattern. Otherwise, if the children of each node are stored in a sorted array, each step would require an $O(\log n)$ binary search, giving total a running time of $O(m \log n)$, the same as with a suffix array. In the RAM model, we could technically store the children of each node in a (perfect) hash table to reduce the time back down to $O(m)$, but this might be overkill in practice.

To economize on space, we usually store suffix trees in compressed form, as shown in Figure 9.6(b), by removing non-branching internal nodes (as in Section 7.4.1) and by then implicitly describing the substring labeling each edge by just two integers, giving the starting and ending indices of the substring within the text. Afterwards, we have a tree with $n$ leaves in which every internal node is branching, so the entire tree occupies just $\Theta(n)$ space. Searching is still easy within this compressed format. In fact, it now takes only $\Theta(k)$ time to enumerate the $k$ matches within the subtree matching our pattern, since this subtree must have only $\Theta(k)$ size if it has $k$ leaves.

We will see how to quickly build a suffix tree efficiently in just a moment, but first let us see how suffix trees allow us to efficiently solve a host of common string processing problems. For example, consider the longest common extension problem: preprocess a text $T[1 \ldots n]$ so that for any two starting indices $i$ and $j$, we can quickly determine the length of a match between $T[i \ldots]$ and $T[j \ldots]$. Suppose we build $T$ into a suffix tree and augment each internal node with its letter depth, as shown in Figure 9.6(b). If we preprocess the tree to support constant-time lowest common ancestor (LCA) queries as in Section 8.3, then we answer longest com-
mon extension queries in constant time by returning the letter depth of the LCA of leaves $i$ and $j$. The following problem provides additional examples to help gain familiarity with string processing using a suffix tree.

**Problem 150 (Suffix Tree Practice).** For all the problems below, please assume for simplicity that our alphabet size is constant, allowing us to build a suffix tree from a length-$n$ text in $\Theta(n)$ time and to search for a length-$m$ pattern in $O(m)$ time. Please show how to solve all of the following problems in $\Theta(n)$ time using a suffix tree.

(a) **Longest Common Substring.** Given two strings $S_1$ and $S_2$ of combined length $n$, find a longest substring occurring in both $^4$. [Solution]

(b) **Longest Palindrome.** A palindrome is a string that reads the same forward as backward. Please find a longest palindromic substring of a length-$n$ text. [Solution]

(c) **Longest Recurring Substring.** Please find a longest substring occurring in two different locations in a length-$n$ text. Can you modify your solution to consider only substrings that occur in two non-overlapping locations? [Solution]

(d) **Repeat Length.** Given a length-$n$ text $T$, find all values of $k$ such that $T$ consists of a repeating substring of length $k$. The last repetition at the end of $T$ may be incomplete. For example, if $T$ is ‘abcabcabcabcaabb’, then $k$ can be 4, 8, or 12. [Solution]

### 9.4.1 Converting Between Suffix Arrays and Suffix Trees

Assuming we store the children of each node in sorted order, a suffix tree takes the same amount of time to build as sorting the characters in our text, just like a suffix array. In fact, the suffix tree and suffix array are so closely related that we can convert between the two in $\Theta(n)$ time. If we enumerate the leaves during a left-to-right traversal of a suffix tree, we get a suffix array. To go the other direction, the crucial insight is that the suffix tree is really just a Cartesian tree of a suffix array, or more precisely of the array of longest common prefixes (LCPs) attached to a suffix array. Figure 9.7 illustrates this correspondence. Recall from Section 8.3 that a Cartesian tree recursively partitions an array into smaller pieces by branching on the minimum value in the array. As seen in Figure 9.7, the minimum value in our LCP array is zero, with each zero representing a switch of the initial character in the suffix array (e.g., moving from suffix ‘ference’ to suffix ‘nce’, which changes the initial character from ‘f’ to ‘n’). If we split the suffix array on all of these zeros, we therefore split it into blocks that correspond to all of the different starting characters in our text — which is exactly how the root of a suffix tree should branch!

Observe that LCPs end up as the letter depths of the interior nodes in our suffix tree$^5$. The two quantities actually play analogous roles: where an algorithm using suffix arrays would use LCPs, an equivalent algorithm over a suffix tree would use letter depths. For example, recall how a suffix tree finds the longest common extension of $T[i...]$ and $T[j...]$ by taking the letter depth of the LCA of $i$ and $j$. If we remember from Section 8.3 how a range minimum query in an array is equivalent

---

$^4$In Chapter 11 we will investigate a similar class of problems that involves finding longest common subsequences (a subsequence is a subset of the characters of a string that is not necessarily contiguous, as is a substring).

$^5$In the other direction, if we convert a suffix tree to a suffix array, we also easily construct a corresponding LCP array to accompany the suffix array by writing down the letter depth of the lowest common ancestor between each pair of leaves during our traversal.
to an LCA query in its Cartesian tree, we see that we can also compute the longest common extension of $T[i\ldots]$ and $T[j\ldots]$ in a suffix array by computing a range minimum in the LCP array between the locations of suffixes $i$ and $j$.

**Problem 151 (Mapping Suffix Tree Algorithms to Suffix Arrays).** As an exercise to help understand the correspondence between suffix trees and suffix arrays, try to solve all four parts of problem 150 using algorithms expressed in the context of a suffix array rather than the context of a suffix tree. For some of these tasks, you may find it easier to re-interpret the corresponding suffix tree solution in the context of a suffix array (e.g., using range minimum queries over LCPs instead of letter depths of LCAs). For others, you may find it easier to work directly with the suffix array from scratch, without first starting from an algorithm on a suffix tree. [Solution]

### 9.5 Extensions and Advanced Topics

String and text processing is an extensive domain of study. Before closing this chapter, we briefly highlight several advanced topics that build on the techniques we have already learned.

#### 9.5.1 Multiple Patterns

Consider matching a set of patterns $P_1 \ldots P_k$ against the same text, looking for all matches of the form $(i, j)$ where some pattern $P_j$ matches at offset $i$ within the text. If we first build a trie out of patterns as in Figure 9.5, then we can check for matches at any offset $T[i\ldots]$ within the text by following the characters $T[i]$, $T[i+1]$, and so on as we walk down the trie. While straightforward, this approach can unfortunately be somewhat slow since we restart from the root of the trie at every offset of the text. However, in almost the exact same way the (Knuth-)Morris-Pratt algorithm speeds up “brute force” string matching, the Aho-Corasick algorithm adds extra “mismatch” edges to the trie of patterns to create an automaton capable of finding matches with a single pass over the text, with no backing up. Its running time is essentially the best possible: $\Theta(n + m + r)$, where $n$ is the length of the text, $m$ is the combined length of all the patterns, and $r$ is the total number of matches (note that this could conceivably be a very large number, if many patterns match at each index of the text). [Algorithm details]

#### 9.5.2 Multiple Texts

To support pattern matching across multiple documents, a common approach is to concatenate the documents into one long string\(^6\) that is then converted into a suffix array or tree. In this situation, we often want to know just which documents match the pattern. Unfortunately, a suffix array or tree will enumerate every single individual match within the documents, causing us to sort through perhaps millions of matches only to finally learn that they came from just a small handful of documents. When this happens, we can save considerable time by only sorting the matches that actually come from the documents we care about.

\(^6\)We must be careful to end each document with a dummy end-of-string character, to avoid spurious matches of the pattern against a suffix of one document followed by a prefix of the document after it.
documents. This is easy to remedy, however. Using a suffix array, let us augment each index in the array with its document ID. Within the block of suffixes matching a pattern, we therefore want to find all the distinct document IDs (say, \( k \) of them), and this can be done in \( \Theta(1 + k) \) time using the result of problem 135.

### 9.5.3 Approximate String Matching

In order to search for “approximate” matches of a pattern in a text, we first need to define how to measure the quality of an approximate match between two strings. One particularly common measurement of distance between strings is *edit distance*, giving the minimum cost of transforming one string into another using some combination of character insertions, deletions, and modifications\(^7\). Alternatively, we can measure similarity between two strings by finding an optimal *alignment* between them, showing the best way to establish a correspondence between their respective characters. These measurements are sometimes applied at a higher level of granularity (e.g., words or lines of text, instead of individual characters); for example, the Unix `diff` utility finds a likely correspondence between the lines of two similar text files, suggesting where lines might have been inserted, deleted, or changed. We will see how to compute both edit distance and optimal alignment using dynamic programming in Chapter 11 (problem 213).

Searching a long text for approximate matches of a pattern can be quite challenging, since algorithms for computing edit distance or optimal alignment are somewhat slow, so it is generally not feasible to apply them at every single offset within the text. Often, we run these algorithms to look for approximate matches around a restricted subset of seed locations in the text identified by exact matches with shorter substrings of the pattern; the “BLAST” search tool in bioinformatics is a good example of this approach. Another common type of approximate string matching problem involves the presence of *wildcard* characters (sometimes called “don’t care” characters) that successfully match any other character. For example, if ‘?’ denotes a wildcard character, then ‘a??or?t?m’ matches both ‘algorithm’ and ‘arboretum’.

Although brute-force string matching or matching with regular expressions can accommodate wildcard characters, both techniques carry \( \Theta(mn) \) worst-case running times, which may not be suitable for long patterns. In Chapter 7?, we will see a surprising application of the Fast Fourier Transform (FFT) that enables us to find all matches in the presence of wildcards in nearly linear time.

**Problem 152 (Pattern Matching with up to \( k \) Wildcards or Mismatches).** Suppose we wish to find all offsets within a length-\( n \) text that either (1) match a length-\( m \) pattern containing \( k \) wildcard characters, or (2) match a length-\( m \) pattern exactly, except in up to \( k \) mismatching characters. Please show how to solve both problems in \( O(kn) \) time by using an \( O(1) \) algorithm for longest common extensions. As an extension of (2), what if we wish to find all offsets within the text that result in a minimum number of mismatches? Please show how we can also solve this problem in \( O(kn) \) time, where \( k \) is the number of mismatches at these locations (note that in this case, as opposed to the first part of the problem, you do not know \( k \) in advance). [Solution]

---

\(^7\)Edit distance is actually a full-fledged distance metric (Section 2.6), since it satisfies the triangle inequality. This can be helpful in several ways; for example, if we want to find a “consensus” string \( s \) minimizing the sum of edit distances \( \sum d(s, s_i) \) to a set of \( n \) other strings \( s_1 \ldots s_n \) (an NP-hard problem), we can obtain a 2-approximate solution using the result of problem 35.
9.5.4 Spell Checking

While it is easy to quickly check if a word $w$ is misspelled by looking it up in a dictionary of correctly-spelled words (say, stored as a hash table or a trie), it is slightly less trivial to suggest alternative correct spellings for $w$ that a user might have intended. Industrial-strength spell checkers can be quite complicated, although they are often based on straightforward ideas. As one example, we can search the dictionary for exact matches of slight modifications to $w$. For example, we could do a dictionary lookup after applying to $w$ all possible single-character insertions, deletions, or modifications, or we can achieve the same effect faster using a more sophisticated data structure:

**Problem 153 (Dictionary Lookup with at Most One Mismatch).** Suppose you are given a collection of $k$ strings $s_1 \ldots s_k$ of combined length $n$, which you would like to preprocess so that given any query string $q$ of length $m$, you can check in $O(m)$ time whether or not $q$ is one of the strings in your collection. If there are no exact matches, we might like to know if $q$ matches any of our strings after insertion, deletion, or modification of just one character. Please show how to answer this question in $O(m)$ time using a data structure taking $\Theta(n)$ time and space to build. For simplicity, please assume our alphabet size is constant. As a hint, see problem 147. [Solution]

Another approach is to rank candidate words $w'$ from the dictionary according to how well they match $w$, using a measurement like edit distance or optimal alignment score, although those approaches might not account for the fact that ‘snd’ is more likely a typo for ‘and’ than ‘end’, since ‘and’ is a more common English word. A more statistically-grounded approach might use Bayes rule:

$$
\Pr[\text{intended } w' \mid \text{typed } w] = \frac{\Pr[\text{typed } w \mid \text{intended } w'] \Pr[\text{intended } w']}{\Pr[\text{typed } w]}
$$

Since $\Pr[\text{typed } w] = 1$, this formula boils down to $\Pr[\text{typed } w \mid \text{intended } w']$, which we can estimate using an edit distance measurement as before\(^8\), biased by $\Pr[\text{intended } w']$, which we can estimate using the frequency of word $w'$ in a large document corpus. Since it might be too expensive to apply a formula like this to every word $w'$ in our dictionary, we might first compute a smaller pool of candidate words $w'$ to check. For example, we can use all the words having the same first letter and similar length as $w$, or we could check also all colliding words under a hash function that computes a “phonetic signature” — mapping a word to a short string that corresponds to how the word is pronounced, rather than how the word is spelled (so ‘fosforesens’ might be recognized as a misspelling of ‘phosphorescence’).

If you have access to a large enough collection of documents, you may have sufficient data to use the surrounding context of $w$ to improve spell checking quality via a similar Bayesian approach as above. For example, ‘pattern marching’ may be a typo for ‘pattern matching’, a two-word combination which occurs much more often. As this example shows, context can even help identify errors that are not misspellings.

\(^8\)Here, we would need to use a “probabilistic” formulation of edit distance, where the output value gives the probability that $w'$ becomes transformed into $w$ given the relative probabilities of typographical errors like inserted characters, deleted characters, or mistyped characters.
9.5.5 Predictive Completion

Modern cell phones and search engines are fond of suggesting likely completions of text as it is being typed. This shares much in common with spell checking, although with more emphasis on completing a partial fragment; for example, a spell checker might not be so bold as to suggest ‘algorithm’ as a replacement for ‘algo’, but this would be highly likely as a completion.

Combined with the prior techniques for spell checking, a trie is the ideal data structure for suggesting completions of a partially-typed word or sentence. If we store a dictionary of words in a trie, then by walking down the trie as we type, the content of our current subtree represents all possible completions of our partial word. We can also build a “word level” trie out of a collection of short phrases, where each node contains a word and branches based on the following word. In this case, our current subtree tells us prospective completions for a partial phrase.

9.5.6 Text Compression

Data compression is a rather extensive domain of study in its own right, encompassing many ideas that go well beyond the scope of this book. However, if we focus on lossless compression, where data can be exactly reconstructed from its compressed encoding, then many of the fundamental building blocks used in popular compression schemes are based on techniques discussed in this chapter. We have already discussed some of these alongside the Burrows-Wheeler transform.

Tries show up quite often in compression techniques. For example, dictionary-based methods, such as the popular Lempel-Ziv family of compression algorithms (LZ77, LZ78, LZW), work by finding long patterns that occur multiple times, replacing each redundant instance with a shorter reference to its initial occurrence. A trie serves as a convenient way to store and match against the dictionary of patterns generated during the encoding process, and is commonly used in the LZ78 encoding system in particular (so named because it was proposed in 1978).

In the next chapter, we will introduce Huffman coding, which uses a trie to represent binary codes for individual characters in our alphabet. Compression is achieved by using short codes for frequently-occurring characters; for example, a Huffman code based on letter frequencies from English text encodes the most-frequent letter ‘e’ with the short binary string 100 and the least-frequent letter ‘z’ with the much longer string 1101000100. Since ‘e’ appears roughly 200 times more often than ‘z’, this can save substantial space.

9.6 Additional Problems

Problem 154 (Circular String Matching). Suppose either our text, pattern, or both are “circular” strings in the sense that we treat their first and last characters as being adjacent. The scenario arises in computational biology when objects like proteins sometimes fold into ring-like structures. Please show how to find all occurrences where a length-\(m\) pattern matches a longer length-\(n\) text in only \(\Theta(n)\) time (even in the comparison model). In the context of a circular pattern, a “match” means that we can cut the pattern
somewhere to unfold it into a length-\(m\) string that matches a substring of the text. See if you can find solutions that work for the case of a circular text, a circular pattern, or both.

Problem 155 (Minimum Rotation of a String). Among all circular rotations of a string, consider finding the one that is smallest lexicographically (i.e., the first row of the Burrows-Wheeler transform, not counting the row starting with \$). This problem is somewhat related to circular string matching, as it gives a canonical way to linearize a circular string; for example, to test if one string \(S\) is a circular rotation of another string \(T\), we could search for \(S\) within \(T\) written twice consecutively, or alternatively we could just check if \(S\) and \(T\) have the same minimum rotation. Please give a simple method for computing the minimum rotation of a length-\(n\) string in just \(\Theta(n)\) time even in the comparison model. As a hint, if you examine \(k\) characters when comparing rotations starting at two different starting indices, then this should allow you to exclude \(k - 1\) starting indices from future consideration.

Problem 156 (Finding all Palindromes in a String). In problem 150 we learned how to find a longest palindrome in a text \(T[1\ldots n]\) in \(\Theta(n)\) time using a suffix tree. Here, we develop a simpler alternative solution that does not require fancy data structures, and that in fact solves the more general problem of computing the palindrome “radius” around every character in the text. As a preprocessing step, let us double the length of \(T\) by inserting the same “dummy” character \(d\) in between each of the original characters in \(T\), so our string now reads \(T[1], d, T[2], d, T[3], \ldots, d, T[n]\). This allows us to assume for simplicity that all palindromic substrings of \(T\) have odd length, so they each have a unique center character. We now scan through \(T\) from left to right. As shown in Figure 9.8, suppose we have determined all of the palindrome radii up to index \(i - 1\), and that we are currently scanning outward to position \(j\) in the process of determining the palindrome radius around center point \(i\). Please show how to either advance \(i\) or \(j\) spending only constant time, leading to an overall running time of \(\Theta(n)\).

Problem 157 (Divide-and-Conquer String Algorithms). The principle of divide and conquer can be quite useful for designing a variety of string searching algorithms. Two nice examples are below. For both of these problems, recall that the linear-time (Knuth-)Morris-Pratt algorithm tells us the length of the longest prefix of a pattern that matches at each index within a text.

(a) Please show how find all the longest palindromes in a string in \(O(n \log n)\) time using a divide-and-conquer algorithm.

(b) A tandem repeat in a string is a substring that occurs twice in a row with no intervening characters. For example, ‘xyz’ is a tandem repeat in ‘arqxyzzxyf’. Please show how to find all \(k\) occurrences of tandem repeats in a length-\(n\) string in \(O(k + n \log n)\) time using a divide-and-conquer algorithm.
Optimization
Optimization, the process of finding the best solution from a set of feasible alternatives, is a key component of nearly every technical discipline. To give a few broad examples, most fields of science attempt to build simple theoretical models that accurately describe the behavior of physical systems; optimization (in this context, known as regression) is used to fit the parameters of such a model so it agrees with experimental data. In fields like physics and computational chemistry, the evolution of a system over time (e.g., a protein molecule as it folds, or a planet orbiting the sun) is described by a minimum-energy solution to a set of equations, and hence can be predicted using optimization. In most areas of engineering, we solve optimization problems to design systems that are as efficient, robust, and cost-effective as possible. In the field of machine learning, classification algorithms can optimize their parameters to “learn” from a set of known training data, in order to make better predictions against unknown test data (also another example of regression). If you study the field of operations research, you may spend a majority of your time studying optimization problems that have useful applications in many areas of practice, such as transportation, logistics, scheduling, routing, and many more.

This chapter is the first of five that focus on general algorithmic techniques for solving optimization problems. The first three study greedy algorithms, dynamic programming, and linear programming, all powerful techniques with broad applicability. We then study various “heuristics” for dealing with computationally hard problems, and finally techniques for continuous optimization problems.

**Greedy Algorithms.** Contrary to what you might have learned as a child, being greedy is sometimes good. Computer scientists are very fond of greedy algorithms because they are typically fast, simple to describe and implement, and widely applicable. Greedy algorithms give optimal or nearly-optimal solutions for a broad range of optimization problems in practice.

How do greedy algorithms operate? As a trivial example, suppose we know the skill levels of \( n \) prospective players in some sport, from which we want to build a team of \( k \) players having the largest possible total skill. A prototypical greedy solution would choose players one by one until the team is full, at each step picking the most skillful player still available. Greedy algorithms are the simplest examples of the “incremental construction” design principle applied to optimization problems. They build up a solution one element at a time, making each successive choice
“greedily”, giving the best short-term improvement to our partial solution. Most greedy algorithms never go back and revise old decisions, building a solution very quickly in just a single pass. In our example, once someone is added to the team, that decision is never reconsidered.

Although greedy algorithms are perhaps the simplest methods for solving many optimization problems, their inappropriate use when they do not apply is a very common error among novice algorithm designers. Many problems may at first glance appear greedily-solvable, but actually require more sophisticated techniques like dynamic programming (the subject of the next chapter). A common mistake among beginning students is to apply a greedy algorithm to a few simple inputs, to observe that it computes an optimal solution in each case, and then to conclude that it must give an optimal solution for every instance. Of course, it is unwise to draw such a conclusion for any algorithm without a more careful analysis; however, the sheer simplicity of a greedy algorithm can often tempt us to forgo the careful analysis and instead trust our intuition. In the pages that follow, we will study several techniques one can use to mathematically justify the correctness of a greedy algorithm. Such a formal justification can be very important in developing appropriate confidence that a given greedy algorithm works, particularly for more difficult problems in which our intuition may lead us astray.

As with many algorithmic concepts, the best way to develop a knack for recognizing greedily-solvable problems and analyzing greedy algorithms is by working through examples. The concepts in this chapter are therefore illustrated using progressively more complex examples, each one drawn from a domain of problems to which greedy algorithms have notable applicability. Along the way, we also highlight the use of greedy methods in online and approximation algorithms, two subfields where greedy techniques are often found. An extensive collection of practice problems is included at the end of the chapter.

10.1 Example: Scheduling on a Single Machine

Scheduling problems involve computing an assignment of jobs to one or more machines over time. Greedy methods have been successfully applied to many of these, including the following simple example. Suppose we need to schedule $n$ jobs on a single machine, subject to the following constraints:

- Each job requires exactly one unit of time to process,
- Each job $j$ has an associated integer release time, $r_j$, known in advance, specifying the time at which it is released into the machine and made available for processing. We cannot schedule job $j$ at any point in time earlier than $r_j$,
- The machine can process at most a single job at once, and
- The machine is non-preemptive; that is, once it starts a job, it continues working on that job until completion. It cannot temporarily interrupt a job, work on something else for a bit, and then return to finish the job later.

Our goal is to schedule the jobs so as to minimize the sum of their weighted completion times, $\sum_j w_j C_j$, where $w_j$ is the weight of job $j$ (higher weight meaning higher
10.1. EXAMPLE: SCHEDULING ON A SINGLE MACHINE

Figure 10.1: Graphical depiction of (a) an example instance for our scheduling problem, and (b) an optimal schedule for this instance. Each job \( j \) is represented by a unit-width box whose height corresponds to the weight, \( w_j \), of the job. The objective value for schedule in (b) is \( \sum_j w_j C_j = 9 \cdot 1 + 14 \cdot 2 + 6 \cdot 3 + 12 \cdot 5 = 115 \).

priority) and \( C_j \) is the time it completes\(^1\). Figure 10.1 shows an example instance with 4 jobs and their optimal schedule. Sometimes the objective of this problem is written as minimizing the average weighted completion time of the jobs, \( \frac{1}{n} \sum_j w_j C_j \), and sometimes it is written as minimizing the total or average weighted flow time over all the jobs, \( \sum_j w_j F_j \), where the flow time \( F_j = C_j - r_j \) of job \( j \) indicates the total time \( j \) spends in the system. All of these are essentially equivalent in that a schedule minimizing one of them clearly minimizes the others as well.

To build an optimal schedule, observe that a job \( j \) of high weight should intuitively be scheduled as early as possible, or else the term \( w_j C_j \) will contribute a substantial penalty to the objective. This suggests a greedy algorithm where we simulate the arrival of jobs over time, always choosing from the set of available jobs the one having maximum weight to schedule next. This greedy choice rule is easy to implement in \( O(n \log n) \) time by first sorting on release time and then using a binary heap keyed on weight to hold the set of active jobs as we simulate their arrival and schedule them one by one. Our algorithm is quite simple and intuitive, and indeed for most problems that allow greedy solutions, the right algorithm is usually one that is simple and feels “natural”.

10.1.1 Proof by Contradiction: The Exchange Argument

Even for simple greedy algorithms, it is still important to justify correctness in a mathematically convincing fashion. One standard way of doing this is with a so-called exchange argument: we hypothetically suppose, for purposes of contradiction, that our greedy solution is not optimal on some input. It follows that any optimal solution deviates from our greedy solution somewhere, at a point where it used a non-greedy decision. In our scheduling example, this might be a point in time where the greedy solution chose job \( j \), but an optimal schedule chose some other job \( j' \) with \( w_{j'} < w_j \). We then consider making an exchange in the optimal solution so it looks more like the greedy solution; for example, what if the optimal solution had selected job \( j \) instead of \( j' \) at this point in time? If we can argue that such a modification does not negatively impact the objective value of our optimal solution,

\(^1\)In case you wonder why \( C_j \) is a capital letter and \( w_j \) is lowercase, this is a convention used in the scheduling literature to indicate that \( C_j \) is an output while \( w_j \) is an input.
we can typically establish a contradiction — either that the greedy solution was just as good as the optimal solution, or that the supposedly “optimal” solution couldn’t have been optimal in the first place. [Detailed exchange argument for our scheduling example]

\[ \]

**10.1.2 Proof by Induction**

A second popular approach for proving optimality of greedy algorithms uses induction on the size of our problem instance, arguing that our greedy algorithm optimally solves every instance of size \( n \) under the assumption (by induction) that it optimally solves every instance of size strictly smaller than \( n \). To use this approach, it helps to phrase our algorithm in the following recursive form:

1. The algorithm initially makes a greedy choice and commits to some small piece of the solution.
2. Next, the algorithm applies itself recursively to the “left-over” part of the problem in order to complete the solution.

Two key properties must now hold for our algorithm to be optimal:

- **Greedy Choice.** Our algorithm must satisfy the *greedy choice property*, which states that the partial solution built in step (1) can always be extended to reach an optimal solution, so in some sense the algorithm is heading “in the right direction”. This is often shown using an exchange argument. [Proof that our example algorithm satisfies the greedy choice property]

- **Optimal Substructure.** Our problem must satisfy the *optimal substructure property*, which states that completing any partial solution is equivalent to optimally solving a smaller “left-over” instance of the same problem\(^2\). In our scheduling example, once the algorithm schedules its initial job at time \( t \), the best way to complete the schedule is to optimally schedule the remaining \( n-1 \) jobs so as to minimize their sum of weighted completion times (where the release times of all jobs are now considered to be at least \( t+1 \), so they cannot overlap the job we just scheduled).

Optimality follows as a natural consequence of these two properties. Greedy choice tells us an optimal solution is still reachable after step (1), and the optimal substructure property tells us that to reach this optimal solution, we need to optimally solve the smaller “left-over” problem in step (2). Since this left-over problem has the same form as the original, our algorithm will indeed optimally solve it due to induction on problem size.

The following problems offer a good opportunity to practice applying and analyzing greedy algorithms on relatives of our basic scheduling problem.

\(^2\)Optimal substructure is a key property in optimization, since it tells us that optimal solutions to a problem contain within them optimal solutions to smaller subproblems. This has important algorithmic implications. For example, we will see in the next chapter how optimal substructure enables the use of *dynamic programming* to solve large problem by combining solutions to smaller subproblems, even when the greedy choice property fails to hold.
Problem 158 (Another Greedy Scheduling Algorithm). Another natural greedy approach to our scheduling problem is the following: sort the jobs in decreasing order by weight, then for each job in sequence, greedily try to schedule it as early as possible (depending on its release time and the jobs that have already been scheduled). Does this approach also compute an optimal schedule? Can you implement this algorithm in only $O(n \log n)$ time? [Solution]

Problem 159 (Non-Unit Processing Times). Assume that all jobs are available at the beginning of time (i.e., all jobs have $r_j = 0$), and let us associate an integer processing time $p_j$ for each job $j$, specifying the time the job needs to run. Show that our original greedy algorithm no longer satisfies the greedy choice property, but that if we schedule our jobs in decreasing order of $w_j/p_j$, then we once again minimize the sum of weighted completion times. [Solution]

Problem 160 (An Alternate Objective). The problem above asks us to schedule $n$ jobs, each with an associated weight $w_j$ and processing time $p_j$, on a single machine, so as to minimize the weighted sum of completion times $\sum w_j C_j$. Another popular objective in this setting is to minimize $\sum w_j (1 - \alpha^{-C_j})$, where $\alpha > 1$ is some specified constant. Observe that jobs completing early contribute very little to this objective, while jobs completing late contribute almost their entire weight $w_j$. Please give an $O(n \log n)$ greedy algorithm for optimally ordering the jobs according to this new objective. [Solution]

Problem 161 (Scheduling with Deadlines). Let’s assume that all jobs are available at the beginning of time (i.e., all jobs have $r_j = 0$), and that each job $j$ has an associated processing time $p_j$ as well as a deadline $d_j$ (note that there are no job weights here). We would ideally like to produce a schedule where $C_j \leq d_j$ for all jobs $j$, but this may not be possible. Devise an $O(n \log n)$ greedy algorithm which minimizes the maximum lateness over all jobs. The lateness of job $j$ is naturally defined as $L_j = C_j - d_j$; note that this can be a negative number if job $j$ completes before its deadline. [Solution]

Problem 162 (Precedence Constraints). Let us generalize the objective for the preceding problem slightly. Associate with every job $j$ a monotonically increasing penalty function $f_j(C_j)$ that tells us the amount of penalty we incur if job $j$ completes at time $C_j$. We would like to order our jobs so as to minimize the largest penalty among all jobs (the preceding problem is a special case where the penalty of a job corresponds to its lateness). As in the preceding problem, each job $j$ also has a processing time $p_j$ and a release time $r_j = 0$. Show how to construct an optimal solution in $O(n^2)$ time using a simple greedy algorithm. To make things a bit more interesting, extend your solution to allow for precedence constraints among our jobs. We can model these using a directed acyclic graph (DAG), where the $n$ nodes represent jobs and a directed path from node $i$ to node $j$ means that job $i$ must be completed before job $j$ starts. Your algorithm should run in $O(m + n \log n)$ time, where $m$ is the number of edges in precedence graph. [Solution]

Problem 163 (Special Cases of Deadline-Constrained Scheduling). Suppose we are given $n$ jobs, where each job $j$ has a processing time $p_j$, a deadline $d_j \geq p_j$ by which it must be completed, and a value $v_j$. We would like to schedule a maximum-value subset of jobs on a single machine in a non-preemptive fashion (i.e., jobs must be processed in their entirety in a single contiguous block of time). This problem is NP-hard since if all deadlines are equal to a common value $C$, it is nothing more than the famous NP-hard “knapsack problem” (e.g., see Section 11.2).

(a) In the special case where all jobs have unit value we can solve the deadline-constrained scheduling problem optimally in $O(n \log n)$ time using a simple greedy algorithm. Try to devise such an algorithm. As a hint, consider the jobs in increasing order by their deadlines, and you may on occasion wish to unschedule a previously scheduled job (this is somewhat uncharacteristic of greedy algorithms, since they typically do not revise past decisions). [Solution]
(b) In this special case where all jobs have unit processing times, show that we can also construct an optimal solution using a simple $O(n \log n)$ greedy algorithm. [Solution]

Problem 164 (Two-Stage Flow Shop Scheduling). A flow shop scheduling problem requires that we send $n$ jobs through a series of $k$ sequential stages on an assembly line. Each job must be processed by each of stages $1, 2, \ldots, k$ in order, but the ordering of the jobs on each stage can be arbitrary (e.g., we can process job 1 followed by job 2 on the first stage, then job 2 followed by job 1 on the second stage). The stages operate in parallel, so they can both process (different) jobs at the same time. In this problem we consider the simple case of $k = 2$ stages, since the problem is NP-hard for $k \geq 3$. Suppose for each job $i$ we know the time $a_i$ required to process it on stage 1 and the time $b_i$ required to process it on stage 2. Please describe an optimal greedy algorithm for scheduling jobs that minimizes the makespan of our schedule (the time the final job completes in stage 2). As a hint, you may want to consider first consider the simpler case where $a_i \leq b_i$ for all jobs $i$, and also the case where $a_i \geq b_i$ for all jobs $i$. [Solution]

Problem 165 (Two-Stage Scheduling with Non-Uniform Machines). You need to process $n$ jobs, all of unit size, on $m$ machines, each running at a different (known) speed. Please show how to schedule the jobs so as to minimize the makespan (latest completion time) of the entire schedule. Next, suppose we have two “stages” of machines: $m_1$ machines in the first stage and $m_2$ in the second. Again, all machines run at different speeds. Each job needs to be processed by some machine in stage 1 (any will do) and then later by some machine in stage 2 (again, any will do); the job can wait as long as necessary between these two steps. For a challenge, please devise a fast algorithm for computing a schedule of minimum makespan. [Solution]

The scheduling problems above are just the tip of the iceberg, as vast numbers of different scheduling problems have been studied in the literature. For those interested in more detail, the endnotes contain a table of all the scheduling problems found throughout this book, as well as a discussion of how scheduling problems are classified based on objective, machine environment, and other constraints.

10.1.3 Greedy Methods and Online Algorithms

Recall from Section 1.7.3 that an online algorithm does not see the entire input to a problem right away, but rather must make decisions as input gradually arrives over time. Greedy algorithms are rather common in such online settings, since they only consider short-term consequences of their decisions, and in an online setting, short-term information is all we have available.

As an example, suppose that for our example scheduling problem that we do not know the release times of jobs in advance, and that instead we only learn about jobs when they haphazardly arrive at our machine as time progresses. The machine does not know what jobs, if any, will be arriving in the future. It is not difficult to see that our original greedy algorithm operates correctly in this online setting. It still produces an optimal schedule, doing just as well as we could have done if we had the advantage of knowing what jobs would be arriving in the future.

Problem 166 (Greedy Online Scheduling). Suppose we are in charge of scheduling jobs (all of unit duration) on a single machine. In each time step, a set of jobs may arrive, each one having an associated value and deadline. After its deadline elapses, we
can no longer schedule a job. In each time step, we must decide which of our currently-available jobs we wish to process (we can process exactly one job per time step). It may not be possible to schedule every job, so our goal is to schedule a set of jobs of maximum total value. Unfortunately, it is not always possible to compute an optimal solution due to the online nature of the problem, since we do not necessarily know which jobs (if any) will be arriving in the future. Nonetheless, please argue that the simple greedy strategy that always schedules the pending job of maximum value gives us at least half the value of an optimal schedule. That is, argue that this greedy strategy is 2-competitive. [Solution]

Problem 167 (Self-Organizing Linked Lists). We want maintain a linked list on \(n\) elements with associated access probability \(p_1 \ldots p_n\). The linked list is subjected to a long sequence of queries, in which the elements are each accessed at random with frequencies according to the \(p_i\)'s. To access the \(k^{th}\) element in this list requires walking down the list from its beginning, and hence requires \(\Theta(k)\) time.

(a) Suppose we know \(p_1 \ldots p_n\). How do we arrange the elements in the list so as to minimize the expected access time for a single generic element? [Solution]

(b) In an online setting we may not know the access probabilities. Using a technique somewhat reminiscent of splay trees (Section 6.2.7) let us employ the greedy move-to-front heuristic: whenever an element is accessed, we move it to the front of the list, hoping that frequently-accessed elements will therefore tend to stay near the front of the list. Argue that this approach is 2-competitive — i.e., that the expected access time per element is at most twice that of part (a). Please assume that sufficiently many access have occurred to bring us into “steady state”, where the initial ordering of elements in the list no longer has significant influence. [Solution]

Problem 168 (Cache Page Replacement). Most computer systems employ a small, fast “cache” memory capable of holding several pages of data from main memory at once; typical page sizes tend to be in the 1K through 4K range. Memory access is very fast if it happens to be from a page residing in the cache. Otherwise, it causes a cache miss, where we bring the relevant page into the cache. When this happens, we must decide which page currently in the cache to evict. Common “greedy” strategies are to evict the least recently used (LRU) page in the cache, or to impose a first in first out (FIFO) discipline on the cache pages. In the long run, neither strategy may turn out to be optimal, since the best policy (if we magically knew all future accesses yet to come) would be to evict the page that will be accessed farthest ahead in the future — the so-called ideal cache policy. Although the LRU and FIFO policies can cause substantially more page faults than an ideal cache, they are reasonably competitive against an ideal cache with less memory. Please show that both the LRU and FIFO policies with a cache size of \(2n\) blocks are \(\frac{2}{1+\sqrt{n}} \approx 2\)-competitive against an ideal cache with a size of just \(n\) blocks, in terms of number of page faults on any memory access sequence. Assume both caches start out empty. As a hint, any sequence involving at most \(2n\) faults for LRU and FIFO cannot fault on the same page twice, so consider consecutive fault sequences of this nature and argue about how many faults they cause on the ideal cache. [Solution]

10.2 Example: Compression with Huffman Codes

In the previous chapter, we studied several approaches for “lossless” data compression, allowing for decompression with no loss in data integrity. Given a string of symbols with non-uniform frequencies of occurrence, another popular method of lossless compression is to assign variable-length binary codes to the symbols, so that short codes correspond to symbols occurring more frequently. A convenient
way to represent such a code is with a binary trie, shown in Figure 10.2(b). Each leaf represents a symbol, encoded with the binary string representing the path from the root down to the leaf; left edges are 0s and right edges are 1s. The resulting code is *prefix-free*, with no symbol’s code being a prefix of any other. This allows for unambiguous decoding by simply walking down the tree as directed by the successive bits of an encoded string. Whenever we reach a leaf node, we emit its corresponding symbol and start over from the root. The process is quite efficient.

For each symbol \( i \), let \( p_i \) denote the fraction of all locations in our string in which it appears, and let \( d_i \) denote its depth in the tree (i.e., the number of bits in its binary code). We would like to design a tree that minimizes \( \sum_i p_i d_i \), the average number of bits per symbol used in our encoding. It turns out this is easy to do with a simple greedy algorithm, generating what is known as a *Huffman code*. As seen in Figure 10.2(a), we start with each symbol being its own individual node, and we then repeatedly select two nodes having minimum frequency of occurrence and merge them together into a single subtree (whose frequency is their sum), repeating until we have built the entire tree. By storing all outstanding nodes in a binary heap keyed on frequency, construction takes only \( O(n \log n) \) time, since each step involves two deletions and one insertion in the heap. An exchange argument shows that the resulting tree does indeed minimize \( \sum_i p_i d_i \). \[Details\]

### 10.2.1 Brief Aside: Entropy Bounds and Arithmetic Coding

Take a random string of symbols, each independently generated according to a probability distribution \( p \) (symbol \( i \) being generated with probability \( p_i \)). Claude Shannon’s famous *source coding theorem* relates the compressibility of this string to the entropy of \( p \), \( H(p) = -\sum_i p_i \log p_i \). It says that on average, \( H(p) \) bits are necessary to represent each symbol or else data loss is likely. Huffman codes nearly achieve this bound, using \( \sum_i p_i d_i \leq H(p) + 1 \) bits per symbol on average. For example, if \( p \) is the length-26 vector of letter frequencies in English text, then \( H(p) \approx 4.165 \), versus \( \sum_i p_i d_i \approx 4.194 \) for a Huffman tree.\[Proof\]

To store a long \( n \)-digit base-10 number on a digital computer, it first needs to be converted to binary. The commonly-used *binary-coded decimal* format does this in perhaps the most straightforward fashion, using \( \lceil \log_2 10 \rceil = 4 \) bits to represent each digit separately. This takes \( 4n \) bits, even though writing the entire number in binary only takes \( \lceil n \log_2 10 \rceil \approx 3.322n \) bits.\[Proof\] The slight loss comes from our insistence on using an integer number of bits for each symbol, the same reason we see the “+1” term in the Huffman coding bound above. Accumulated over an entire length-\( n \) string generated by a random source, Huffman coding uses at most

\[This ignores the small amount of overhead involved in transmitting the encoding tree along with the encoded string, so the decoder knows the binary code for each symbol.\]

\[English text isn’t a string of randomly-chosen symbols though, so \( H(p) \) here isn’t a true lower bound on compressibility. Knowledge of, say, the preceding few characters lowers the entropy of the next symbol quite a bit. For example, the missing character in entrop is almost certainly ‘y’, and if we build a Huffman code not on single characters but on *bigrams* (pairs of adjacent characters) using their frequencies found in English text, this uses only about 3.73 bits per symbol on average. For simplicity, we are also ignoring the need to encode spaces and punctuation.\]

\[On the other hand, if we write the entire number in binary, we lose the ability to easily examine individual digits in its base-10 representation, which the binary-coded decimal approach allows. Researchers have studied the problem of efficient encoding while still allowing \( O(1) \)-time inspection of individual digits; see the endnotes for further detail.\]
10.2. EXAMPLE: COMPRESSION WITH HUFFMAN CODES

Figure 10.2: Building a variable-length binary code from the string ‘TRY GREEDY TREE CODE’. In (a), we merge two nodes with minimum frequencies into a single subtree, the first step in building the Huffman tree shown in (b). This tree encodes ‘TREE’ by the binary string ‘1011000101’. In (c), we depict our probability distribution over symbols as a partition of the unit interval \([0, 1]\) (we’ve used the same order as the leaves of the Huffman tree for convenience in the figure, but the order here can be arbitrary). Here, any binary value \(x = 0.011 \ldots\) codes for ‘E’, since this unambiguously belongs to E’s interval. We can describe the same encoding with a binary radix tree built atop \([0, 1]\).

\[ nH(p) + n \text{ bits in expectation, compared to the theoretical lower bound of } nH(p). \]

We can do slightly better using **arithmetic coding**, an alternative method that is sufficiently interesting and elegant that it is worth a brief diversion to explain.

As seen in Figure 10.2(c), imagine subdividing the unit interval \([0, 1]\) into subintervals representing our symbols, in arbitrary order, with lengths proportional to their frequencies. We can now encode a symbol with a point \(x \in [0, 1]\), providing just enough digits of accuracy in \(x\)’s binary representation so there is no ambiguity about the identity of the subinterval containing \(x\). Having a random source is equivalent to generating \(x \in [0, 1]\) uniformly at random\(^6\). Just as with Huffman coding, this encoding can be viewed in terms of a binary radix trie, where the digits of \(x\) are generated by walking down an implicit binary tree built on top of \([0, 1]\).

The true power of arithmetic coding lies not in coding individual symbols, but an entire length-\(n\) string, which we can abstractly view as a single “symbol” coming from a much more complicated probability distribution — imagine further subdividing each interval in Figure 10.2(c) to represent the second symbol, and so on. Despite its exponential size, the implicit structure of the resulting subdivision still allows

\[ \text{\footnotesize \textsuperscript{6}The reader may want to revisit problem 26, which is equivalent to having a random source and just two subintervals, corresponding to heads and tails in a biased coin flip.} \]
efficient encoding and decoding, using only $nH(p) + O(1)$ expected bits\footnote{We are ignoring here the communication required to inform the other party about the frequencies of the symbols in our string (so the other party knows how $[0, 1]$ should be partitioned).}, essentially matching the lower bound above. This same approach also gives a nice method for base conversion of a number $x \in [0, 1]$. For example, to convert $x = 0.0110101$ in base 2 into base 7, we would set $p = \left[\frac{1}{7}, \frac{2}{7}, \ldots, \frac{6}{7}\right]$ (recursively dividing the unit interval evenly by 7s, since our target is base 7), then we would begin decoding $x$, stopping at whatever precision is desired. \[\text{[Further discussion]}\]

**Problem 169 (Adding Two Numbers in a Streaming Environment).** Suppose two $n$-digit numbers in base $b = O(1)$ stream by, most significant digit first, and we want to output their sum, also in base $b$. Using similar tricks as above, show how to do this in a low-memory environment, using only $O(\log n)$ bits of storage. \[\text{[Solution]}\]

### 10.3 Example: Minimum Spanning Trees

Our next example comes from the domain of algorithmic graph theory. It is the well-known problem of computing a minimum spanning tree (MST) in a graph, a problem that demonstrates well the simplicity and elegance of greedy methods. Although there are many interesting things one could say about the MST problem, we’ll focus our discussion at the moment on only those aspects which involve greedy techniques. In Chapter ??, we cover this problem in much greater detail.

The input to the MST problem is a connected graph with $n$ nodes and $m$ edges, each edge having an associated cost. A spanning tree is a subset of edges of our graph that connects together (i.e., spans) all $n$ nodes of our graph, and that contains no cycles. Our goal is to compute a spanning tree for which the sum of its edge costs is minimized. Figure 10.3 shows an example graph and its MST. A stereotypical application for the MST problem might be for a power company to connect a set of cities into a single circuit as cheaply as possible (here the cost of an edge between two cities represents the cost of building an electrical connection between the two). MSTs have many other applications in practice, and are often also used as building blocks for more sophisticated algorithms.

#### 10.3.1 Kruskal’s Algorithm

Perhaps the simplest and best-known algorithm for computing a minimum spanning tree is a greedy algorithm known as Kruskal’s algorithm. The algorithm is very simple: add edges one by one in increasing order of cost, skipping over any edge that creates a cycle with edges we have added previously. The running time turns out to be dominated by the time required to sort the edges by cost, $O(m \log m)$, which is typically written as $O(m \log n)$ since $m \leq n^2$ (so $\log m \leq 2 \log n$) We won’t provide further details into the implementation of the algorithm here, in particular how to answer “does this edge create a cycle?” queries in an efficient fashion, since this is discussed in detail in Chapter ??, For now, we focus on proving that Kruskal’s algorithm generates an optimal spanning tree, which we can show with a simple exchange argument. \[\text{[Simple exchange argument]}\]
Problem 170 (Unique MST if Edge Costs are Distinct). Use an exchange argument to show that if all edges in a graph have distinct costs, then the graph has a unique minimum spanning tree. [Solution]

Problem 171 (Kruskal’s Algorithm in Reverse). Consider the following “backwards” variant of Kruskal’s algorithm: process edges in decreasing order of cost, deleting each edge in sequence unless this would cause the graph to become disconnected. Does this process yield an MST? [Solution]

10.3.2 Brief Aside: Matroids

Matroids are mathematical structures that capture and generalize many problems solvable by greedy algorithms. For example, Kruskal’s algorithm and its correctness follow directly from the fact that the minimum spanning tree problem can be modeled by a graphic matroid.

More precisely, a matroid is a special type of set system, collection of \( n \) sets over a universe of \( m \) elements. The author likes to visualize set systems in terms of a bipartite graph, as shown in Figure 10.3(b). In order for a set system to be a matroid, it must be hereditary (if set \( A \) belongs to the system, then so must all of its subsets), and it must satisfy the augmentation property: if \( A \) and \( B \) are sets in the system and \( |A| < |B| \), then there must exist an element in \( B - A \) we can add to \( A \) to obtain another set in the system. To give a concrete example, the elements in a graphic matroid are the \( m \) edges in a graph, and a set of edges belongs to our system if it contains no cycles (i.e., if the set is a forest). Here, the augmentation property tells us that if we take two acyclic sets of edges \( A \) and \( B \) with \( |A| < |B| \), then we can always find some edge \( e \in B - A \) such that \( A \cup \{e\} \) also contains no cycles. [Simple proof of this property]

If we associate a value with every element in a set system, then we can locate a set of maximum value using a simple incremental greedy algorithm: start with an empty set, process the elements in decreasing order of value, adding each element in sequence to our current set unless this would create a set that doesn’t belong to the system. The astute reader will notice that this algorithm is exactly the same as Kruskal’s algorithm, only applied in the somewhat more general setting of
a matroid. Its analysis is identical to that of Kruskal’s algorithm, except phrased more generally in terms of sets and elements instead of spanning trees and edges.\footnote{It is interesting to note that this is a bi-directional result, as one can show that if this greedy algorithm works, then our underlying set system must be a matroid.} Kruskal’s algorithm is just a special case of this more general algorithm run on a graphic matroid.\footnote{Technically, the matroid approach corresponds to the variant of Kruskal’s algorithm that computes a maximum spanning tree by choosing edges from highest cost to lowest cost. However, the maximum spanning tree problem is essentially the same as the minimum spanning tree problem; the two can be transformed into each-other by negating the costs of all edges.}

**Problem 172 (Minimum Spanning Pseudotree).** A pseudotree is either a tree or a tree plus one additional edge that forms a unique cycle. A pseudoforest is a collection of pseudotrees, just like a forest is a collection of trees. In the same way the graphic matroid is built from all the forests in a graph, the pseudoforests in a graph also form a matroid, known as a bicircular matroid. Please prove that this is indeed a matroid, and show how we can use this to build a greedy algorithm — similar in flavor to Kruskal’s algorithm — that computes a minimum-cost spanning pseudotree of any connected graph. Do not worry about the running time of this algorithm; we will see later in problem ?? that we can surprisingly solve this problem faster than we currently know how to compute an MST! \[Solution\]

Matroids are an important class of objects in the study of combinatorial optimization, since they provide an abstract general framework that captures many prominent problems. They can help you prove certain results in a single step that would have otherwise taken substantially more work. Problem ?? in Chapter ?? gives another glimpse into their utility.

### 10.4 Example: Matching Problems

Matching problems involve pairing up objects in some optimal fashion. They come in many flavors, and we will later spend all of Chapter ?? studying them in depth. Since only a few matching problems can be optimally solved by a greedy algorithm, the main point of this section is to highlight the use of greedy methods as approximation algorithms. As a warm-up, let’s start with some simple matching problems that do have optimal greedy solutions:

**Problem 173 (Ballroom Matching).** You are a dance instructor for a class of \(n\) boys with heights \(b_1 \ldots b_n\) and \(n\) girls with heights \(g_1 \ldots g_n\). You need to decide on a matching of the boys and girls (i.e., a one-to-one pairing up between the boys and girls), in which boys and girls of similar heights end up together.

(a) Give a greedy algorithm which finds a matching that minimizes the maximum height difference over all pairs of boys and girls. Justify its correctness. \[Solution\]

(b) Give a greedy algorithm which finds a matching that minimizes the sum of height differences over all pairs of boys and girls.\footnote{An interesting and surprisingly more challenging variant of this problem occurs when there are fewer boys than girls, and every boy must be matched (so a subset of girls will end up unmatched). For an \(O(n \log n)\) solution to this more complicated variant, see problem ??.} Justify its correctness. \[Solution\]
10.4. Example: Matching Problems

Matching problems are usually described in the context of a graph: nodes indicate objects to be matched, and edges represent valid pairings between these objects, possibly with associated values. If we are pairing up elements between two different sets, the graph is bipartite, as shown in Figure 10.4(a). Since each node can be matched with at most one other node, a matching is just a subset of edges, no two of which share a common endpoint.

Two common objectives are to compute either maximum-cardinality or maximum-value matchings, where the goals are respectively to include as many edges or as much total value as possible. Examples are shown in Figure 10.4. Both problems can be optimally solved in polynomial time, as we shall see in Chapter ??, although the algorithms in question are somewhat complicated and not quite fast enough for some

Figure 10.4: Examples of problem instances for (a-b) maximum-cardinality matching in a bipartite graph and (c) maximum-value matching in a general graph. In (a) and (b), we show how bipartite matching is used inside a “crossbar switch” in a communication network: (a) depicts the problem as a bipartite graph, and (b) shows the physical wiring pattern in the switch. Packets of information arrive and queue up at the input ports of the switch, each destined for a specific output port. In each time step, each input port can accept one packet from anywhere in its queue, and each output port can send at most one outgoing packet. To maximize the throughput, we therefore want to route as large a matching as possible in each time step.

Problem 174 (Matching Students). You are a science teacher with $2n$ students. Through working with the students over time, you’ve been able to determine the intelligence level of each student (i.e., each student has an associated number which represents his/her intelligence). For their next project, the students must be matched together into $n$ teams of two students each. You’d like to make this matching as fair as possible, so no team is exceptionally smart or exceptionally “nonsmart”. Describe (and analyze) a greedy algorithm which computes a matching that minimizes the combined intelligence of the smartest team. If you’ve picked the right algorithm, you should find that it also simultaneously maximizes the combined intelligence of the weakest team. [Solution]

10.4.1 Matchings in Graphs

Matching problems are usually described in the context of a graph: nodes indicate objects to be matched, and edges represent valid pairings between these objects, possibly with associated values. If we are pairing up elements between two different sets, the graph is bipartite, as shown in Figure 10.4(a). Since each node can be matched with at most one other node, a matching is just a subset of edges, no two of which share a common endpoint.

Two common objectives are to compute either maximum-cardinality or maximum-value matchings, where the goals are respectively to include as many edges or as much total value as possible. Examples are shown in Figure 10.4. Both problems can be optimally solved in polynomial time, as we shall see in Chapter ??, although the algorithms in question are somewhat complicated and not quite fast enough for some
applications, such as the example of crossbar switches in communication networks shown in Figure 10.4. Using greedy methods, we can compute approximately-optimal solutions to these problems much more efficiently.

### 10.4.2 Greedy Approximation Algorithms

Since greedy algorithms are so simple and fast, they are often applied in practice even when it is understood that they might not always compute an optimal solution. They still tend to generate solutions that are reasonably good in many cases. As we will see in Chapter 13, many heuristics for speeding up exhaustive searches (e.g., branch and bound) employ greediness to improve performance.

Even if a greedy algorithm seems to perform well in practice, we might still like to have a rigorous guarantee on how badly it might perform (relative to an optimal solution) in the worst case. As an example, consider the following greedy algorithm for maximum-cardinality matching: start with an empty matching, and as long as there exists an edge whose endpoints are not already matched, pick any such edge and add it to our matching. This algorithm is extremely simple and fast, running in $\Theta(m)$ time, and we can show that it is a $1/2$-approximation algorithm — it always computes a matching that contains at least half the number of edges of an optimal matching. We can prove this fact by performing an exchange argument, just as with many of our preceding greedy algorithms. In this case, however, we will lose up to a factor of two during the process of modifying an optimal solution so it coincides with the greedy solution [Proof]. In Chapter ??, we will see in problem ?? how to generalize this approach to build a $(1-\varepsilon)$-approximation algorithm that runs in $\Theta(m/\varepsilon)$ time, which is $\Theta(m)$ time as long as $\varepsilon$ is constant.

**Problem 175 (Minimum Node Cover).** The well-known minimum node cover problem (also commonly called the minimum vertex cover problem) is the following: given a graph, select a subset of as few nodes as possible which collectively cover all of the edges of the graph (i.e., every edge must have at least one of its endpoints included in the cover). The problem is NP-hard, but we can approximate its solution very efficiently. Suppose we run the linear-time greedy algorithm above for approximate maximum-cardinality matching, and select both endpoints of every edge in its output. Argue that (i) this set is a feasible cover, and that (ii) it contains no more than twice as many nodes as an optimal cover. [Solution]

**Problem 176 (Approximate Maximum-Value Matching).** Consider the problem of finding a maximum-value matching in a graph. Describe a natural greedy algorithm that computes a matching with value at least half that of an optimal matching. Your algorithm should run in $O(m \log n)$ time. [Solution]

### 10.5 Example: Packing and Covering Problems

Suppose you arrive at your first day of orientation at a new university, and are presented with a variety of activities in which you may participate. Each of these $n$ activities occupies some interval of time during the day. Since some of them overlap, you may not be able to attend them all; however, you would like to take part in as many as possible. In other words, we would like to select a subset
Figure 10.5: Examples of problem instances for (a) the maximum-cardinality interval packing problem (otherwise known as the activity scheduling problem), and (b) the minimum-cardinality interval covering problem. An optimal solution for each instance is shaded. Note that these intervals are only one-dimensional, even though we have arranged them with a certain amount of vertical spacing for visual clarity.

of pairwise-disjoint intervals having maximum cardinality. This is known as the **activity selection problem**, and it is commonly used as an example to illustrate the process of designing a greedy algorithm. Figure 10.5(a) shows an instance of this problem along with its optimal solution.

Activity selection is a **packing** problem, involving packing an optimal set of elements into our solution subject to a restriction that elements in the solution must not conflict with each other. More formally, the activity scheduling problem could be called a **maximum-cardinality interval packing problem**. The matching problems we just studied are also packing problems, involving packing disjoint edges in a graph.

**Problem 177 (Optimally Solving the Activity Scheduling Problem).** Consider a greedy algorithm which repeatedly selects, from all remaining intervals (those still disjoint from our solution so far), an interval ending the earliest. Prove that this algorithm gives an optimal solution to the activity selection problem, and show how to implement it in $O(n \log n)$ time. [Solution]

**Problem 178 (Approximating the Activity Scheduling Problem).** Consider a greedy algorithm which repeatedly selects, from all remaining intervals (those still disjoint from our solution so far), an interval of smallest duration. Show that this may not give an optimal solution to the activity scheduling problem, but that it gives a solution containing at least half the number of intervals in an optimal solution. [Solution]

**Problem 179 (Overlap Counts).** Define the **overlap count** of an interval to be the number of other intervals it intersects (two intervals touching only at an endpoint do not overlap). For the activity-scheduling problem, consider the greedy algorithm which repeatedly selects, from all remaining intervals (those still disjoint from our solution so far), an interval of minimum overlap count. Does this algorithm give an optimal solution? Note that there could be several intervals that tie for the minimum overlap count — in this case, is it important how we break the tie? [Solution]

A related and equally important class of problems is that of **covering** problems. Suppose, for example, that you own a swimming pool and need to hire lifeguards to watch the pool all day long. This problem has the same input as the activity selection problem: a set of $n$ intervals, each representing the duration of time covered by a lifeguard that you could potentially hire. Since lifeguards are expensive, you
want to hire the minimum number of lifeguards while still ensuring that every point in time is covered by at least one lifeguard. Formally, we would call this a \textit{minimum-cardinality interval cover problem}.

Whereas packing problems are maximization problems, (e.g., packing as many disjoint elements as possible), covering problems involve minimization (e.g., covering something with as few elements as possible). Both types are frequent targets for greedy methods. When solving a packing problem, we want to repeatedly select “good” items (perhaps of small size, or perhaps of high value with respect to their size) that are disjoint from those already selected. When solving a covering problem, we want to repeatedly select items that cover most efficiently whatever it is that remains to be covered (e.g., maximizing the amount of additional coverage, or covering additional elements at minimum cost per element).

\textbf{Problem 180 (Minimum-Cardinality Interval Cover).} Develop a greedy algorithm for finding a minimum-cardinality interval cover. \textbf{[Solution]}

\textbf{Problem 181 (Minimum-Area Interval Cover).} Suppose that instead of paying a fixed amount of money per lifeguard, you pay each lifeguard an amount of money proportional to the duration of the interval of time when he/she is on duty. Therefore, you would like to solve the \textit{minimum-area interval covering problem}: select a subset of intervals which covers the entire day, but which minimizes the sum of their lengths (the \textit{area} of the cover). Although this problem can be optimally solved by dynamic programming, we can use our solution to the minimum-cardinality interval covering problem to obtain a good approximation. Show that a minimum-cardinality cover has an area at most twice that of the minimum-area cover for the same problem instance. \textbf{[Solution]}

\subsection{10.5.1 \hspace{1pt} Brief Aside: Duality}

In the context of optimization, \textit{duality} is a min-max relationship between two intrinsically-related “dual” problems, one involving minimization and the other maximization. Usually, the two problems are related via an inequality of the form

\[
\text{Maximum solution of problem A} \leq \text{Minimum solution of problem B},
\]

where sometimes we even have equality between the two. Duality relationships are a useful tool for certifying optimality or near-optimality of solutions. If we can find solutions for A and B that are equal in value, then the inequality above implies that both must be optimal for their respective problems. If we can find solutions differing in value by only $\Delta$, then we know each solution is at most $\Delta$ units away from being optimal for its respective problem. For a nice concrete example, consider the following problem:

\textbf{Problem 182 (Minimum Multicut on a Path).} The input to this problem a set of $n$ intervals on a number line, each representing a communication session between its two endpoints taking place along a shared communication line. Your task is cut the line in a minimum number of locations so as to disconnect all $n$ sessions. Each session must be cut at a point strictly between its endpoints. Devise a greedy algorithm which optimally solves this problem. \textbf{[Solution]}
10.5. Example: Packing and Covering Problems

This problem is dual to the activity scheduling problem, enabling an elegant alternative means of proving optimality of our greedy algorithms for both problems. Given a set of input intervals, let \( I \) be the set of intervals in any feasible solution to the activity scheduling problem, and let \( C \) be the set of cuts in any feasible solution to the minimum multicut problem above. It is easy to see that \( |I| \leq |C| \), since the intervals in \( I \) are disjoint from each other, so each one requires its own individual cut if we are to separate them all. If we can come up with solutions for which \( |I| = |C| \), then it follows that both must be optimal for their respective problems. It is no accident that both problems have very similar greedy algorithms. By examining these closely, we see that they do indeed generate such a pair of equal solutions, so both algorithms are therefore optimal. [Further elaboration]

Duality between two problems can often highlight useful structural relationships between the two, and often helps lend insight into the development of algorithms as well. This will be particularly true when we study linear programming duality in Chapter 12, where we shall see that the linear programming formulations of packing and covering problems are indeed duals of each other.

10.5.2 General Covering and Packing Problems

Covering and packing problems abound in practice. We have seen several simple example above, and we will introduce many more throughout this book, particularly in Chapter ?? when we survey common problems on graphs. Here, we discuss the set cover and set packing problems\(^{11}\), high-level problems that capture most other covering and packing problems because their input is a generic set system, with \( n \) sets over a universe \( U \) containing \( m \) elements. Due to their generality, both prob-

---

\(^{11}\)A helpful hint on deciphering problem names: a “minimum X cover” problem asks us to compute a cover composed of Xs. For example, a minimum interval cover is a minimum cover comprised of intervals, a minimum node cover is a minimum cover comprised of nodes, and a minimum set cover is a minimum cover comprised of sets. The same convention holds for packing problems as well.
lems are NP-hard. They can both be approximated via simple greedy techniques, although neither problem allows for very strong approximation guarantees.

**Set Cover.** The minimum-cardinality set cover problem asks us to choose a minimum collection of sets whose union is \( U \) (i.e., a collection of sets that covers the universe of all elements). If our sets have associated costs, we can also ask for a set cover of minimum total cost. Figure 10.6 shows an example of the minimum-cost set cover problem involving the selection of a minimum-cost group of foods that covers a universe of essential nutrients.

Set cover contains all of our previous covering problems as special cases. For example, in the node cover problem, \( U \) is the set of all edges of a graph, and there is one set for every node containing the edges incident to that node. For the interval cover problem, we can take \( U \) to be the set of all interval endpoints, and each interval corresponds to the set of endpoints covered by that interval.

A simple greedy algorithm can approximate the set cover problem reasonably well. Let’s define the per-element cost of each input set as its cost divided by the number of elements it covers (if we’ve already built a partial solution, we only count those elements not already covered by the partial solution). A natural greedy algorithm is to repeatedly select a set of minimum per-element cost, until all elements in \( U \) are covered. This always yields a solution whose cost is at most \( 1 + \ln m \) times the cost of an optimal solution [Short proof]. It turns out that this is essentially the best one can hope to accomplish, since it is actually NP-hard to achieve an approximation factor better than \( c \log m \) for some constant \( c > 0 \).

**Set Packing.** The maximum-cardinality set packing problem asks for a collection of as many sets as possible that are all pairwise disjoint. A more general variant is the maximum-value set packing problem, where we assign values to our sets and seek a maximum-value collection of pairwise disjoint sets. Both of these problems are not only NP-hard, but also hard to approximate to within a factor of \( n^{1/2 - \varepsilon} \) for any positive \( \varepsilon \) as long as \( P \neq ZPP \) (a conjecture about the complexity classes \( P \) and \( ZPP \) that is widely believed to be true, just like the more famous conjecture that \( P \neq \text{NP} \)). On the positive side, however, we can match this approximation bound using simple greedy algorithms, an exercise we leave for the reader:

**Problem 183 (Maximum Set Packing).** In this problem, we analyze simple greedy approximation algorithms for set packing problems.

(a) For the maximum-cardinality set packing problem, consider the greedy algorithm that repeatedly selects a minimum-cardinality set disjoint from those already selected. Show that this is a \( \frac{1}{\sqrt{m}} \)-approximation algorithm, where \( m = |U| \). [Solution]

(b) Please try to find a simple \( \frac{1}{2\sqrt{m}} \)-approximation algorithm for the maximum-value set packing problem. If you are feeling ambitious, see if you can improve the performance guarantee to \( \frac{1}{\sqrt{m}} \). [Solution]

**Problem 184 (The Minimum Test Collection Problem).** Suppose you have \( n \) different diseases and \( m \) different tests, where each test returns “positive” for some subset of the diseases, and “negative” for the other diseases. Given which diseases produce positive and negative test results for each test, we would like to find the smallest possible subset of the tests that suffices to identify all of the diseases. In other words, for each two diseases \( x \) and \( y \), there should be at least one test in our subset that gives different results
10.6. ADDITIONAL PROBLEMS

for $x$ and $y$ — otherwise we could not tell $x$ and $y$ apart. See if you can design a greedy $O(\log n)$-approximation algorithm for this NP-hard problem. [Solution]

**Problem 185 (The Maximum Coverage Problem).** Another NP-hard problem very closely related to the set cover problem is the maximum-value coverage problem. The input to the problem consists of a collection of $n$ sets over a size-$m$ universe, where each element in the universe has an associated value $v_1 \ldots v_m$. We are only allowed to select $k$ of these sets, however, and we would like to select the sets so that the value of the elements in their union is as large as possible. In other words, we want to cover the greatest amount of value using only $k$ sets.

(a) Show that the natural greedy algorithm for this problem gives a $(1-1/e)$-approximate solution. [Solution]

(b) The minimum set cover problem with a size-$m$ universe is NP-hard even to approximate to within a factor of $c \log m$, for some constant $c > 0$. Please show that this implies that the maximum coverage problem is NP-hard to approximate to within a factor of $1-1/e + \epsilon$ for any $\epsilon > 0$, by showing how one can develop an approximation algorithm for minimum set cover based on an approximation algorithm for maximum coverage. [Solution]

**10.6 Additional Problems**

After reading an entire chapter containing nothing but problems solvable by greedy algorithms, one may be tempted to go about trying to apply greedy algorithms to every problem in sight. Unfortunately, while many problems certainly have greedy solutions, many others do not. The reader is now faced with the challenge of differentiating between these types of problems. Our hope is that the intuition and collection of techniques discussed in this chapter will simplify this task. This task will also be easier after reading the next chapter, highlighting problems on which greedy techniques fall short.

**Problem 186 (Minimum Node Multicut on a Tree).** A nice generalization of the preceding problem involves a communication network shaped like a tree rather than a path. Suppose you are given a set of $k$ communication sessions that are taking place in an $n$-node tree. Each session is described simply by a pair of nodes $(i, j)$, since in any tree there is a unique path between any two nodes.

(a) It is NP-hard problem to find a minimum set of edges whose removal disconnects all $k$ communication sessions. Just for fun, prove this fact by showing that if you could solve this problem in polynomial time, even on a star-shaped tree (a tree with one central node connected to $n$ leaves), then you could easily compute a minimum node cover (problem 175) in polynomial time. [Solution]

(b) Fortunately, the problem of finding a minimum set of nodes in a tree whose removal disconnects all $k$ sessions is much easier. Can you solve this problem in $O(n + k)$ time using a greedy algorithm? [Solution]

**Problem 187 (Maximum Multicommodity Flow on a Path).** As in problem 182, suppose we have $n$ communication sessions (described by intervals) that we would like to route through a one-dimensional network (think of it as a number line) with $C$ units of capacity everywhere on the line. Session $i$ involves the transmission of $d_i$ units of data per unit time from point $x_i$ to point $y_i$. For each session $i$, we can choose to disable
CHAPTER 10. GREEDY ALGORITHMS

it, enable it fully, or to transmit data at only a fraction of the maximum rate \( d_i \). If we enable session \( i \) at a rate \( r d_i \) (\( 0 \leq r \leq 1 \)), this will consume \( r d_i \) units of capacity only on the segment between \( x_i \) and \( y_i \), leaving the rest of the network unaffected. Our goal is to maximize the total sum of transmission rates of all sessions. Please show how to solve this problem in \( O(n \log n) \) time with a simple greedy algorithm. [Solution]

**Problem 188 (The Quiz Problem).** You are taking an oral quiz with \( n \) questions, where each question \( j \) has an associated point value \( v_j \). It is up to you to decide the order in which you will attempt the problems. Each problem in sequence that you answer correctly contributes to the total number of points you earn for the quiz, and the quiz terminates upon your first wrong answer (and you receive no points for this wrong answer). Suppose you have estimated the probabilities \( p_1 \ldots p_n \) with which you can correctly solve each of the \( n \) problems. How do you choose an ordering of the questions that maximizes the expected amount of value when you take the quiz? What if you receive points for final question answered incorrectly as well? As a hint (which applies to several greedily-solvable problems), you can sometimes find the right greedy choice rule by “reverse engineering” an exchange argument — that is, go through the motions of making an exchange argument with an unspecified greedy choice rule, allowing us to ultimately determine the initial rule that would make the argument work. [Solution]

**Problem 189 (Boxes of Donuts).** You are the proud owner of a donut shop. At the end of the day, you are left with \( n \) total donuts, \( a_i \) donuts of type \( i \), for \( i = 1 \ldots k \), where \( \sum a_i = n \). Right before closing, you receive an unusual order: as many boxes of donuts as possible, where each box must contain \( T \) donuts all of different types. Show how to optimally fill this order using a greedy algorithm. [Solution]

**Problem 190 (Shortest Non-Subsequence).** Given a length-\( n \) string \( S \), please show how to compute in \( \Theta(n) \) expected time the shortest string \( T \) using only characters from \( S \) that does not appear as a subsequence of \( S \). [Solution]

**Problem 191 (The Shopping Bag Problem).** Suppose we wish to place \( n \) items in a shopping bag, each with an associated weight and strength. The items in the bag are stacked vertically on top of each-other, and we would like to minimize the risk of squashing any item. In particular, we define the risk of squashing an item to be the combined weight of all items stacked above it minus its strength. Devise a greedy algorithm which stacks the items so as to minimize the maximum risk. Does this problem remind you of any other greedy problem we’ve seen already? [Solution]

**Problem 192 (Ballroom Matching Revisited).** Recall the Ballroom Matching problem from section 10.4, in which we are given the heights of \( n \) boys and \( n \) girls as input, and we would like to construct a matching between the boys and girls so each pair has similar height.

(a) Suppose we are not allowed to pair up a boy and a girl if their heights differ by more than \( H \). It may no longer then be possible to match all the boys and girls. Describe a greedy algorithm that constructs a matching of maximum cardinality. [Solution]

(b) Suppose the boys and girls are picky about the speed of music to which they will dance. Specifically, we are given for each boy \( i \) a range \([b_i, B_i]\) of speeds to which he is willing to dance, and for each girl \( j \) we are given a range \([g_j, G_j]\) of speeds to which she is willing to dance. Given that one may only pair up a boy and a girl if their “tempo intervals” overlap, devise a greedy algorithm that computes a maximum-cardinality matching. [Solution]

(c) Suppose, for a difficult dance involving lifts, that each girl must be matched to a boy that is (i) at least as tall as her, and (ii) at least as heavy as her. Given the heights and weights of the boys and girls, please show how to match as many couples as possible, and also show how to implement your algorithm in only \( O(n \log n) \) time. [Solution]
Problem 193 (Interval Packing and Covering Revisited). Here are a few more variants of the interval packing and covering problems introduced in Section 10.5.

(a) Consider again the maximum-cardinality interval packing problem (a.k.a. the activity scheduling problem) and the minimum-cardinality interval covering problem. For both of these problems, suppose our intervals are part of a circle rather than a line segment. How efficiently can you solve these problems now? [Solution]

(b) Suppose we wish to solve a generalization of the activity-scheduling problem for $k$ individuals, so at most $k$ activities can be attended at any one point in time. Show how to select the maximum number of activities in this setting. Also, consider the analogous generalization of the minimum-cardinality interval covering problem, when every point must be covered not once, but by at least $k$ different intervals. Give an efficient greedy solution for this problem. [Solution]

(c) Suppose we need to reserve rooms for a conference. There will be $n$ meetings during the conference, provided to us as a set of time intervals $[a_1, b_1], \ldots, [a_n, b_n]$. Two meetings that overlap in time cannot be scheduled in the same room. Give a simple greedy algorithm that determines an assignment of meetings to rooms that uses the minimum possible total number of rooms. Viewed differently, this problem is the same as that of packing 2-dimensional axially-aligned rectangles in a disjoint fashion. Each rectangle has fixed starting and ending $x$ coordinates as well as unit height (in the $y$ dimension), and we can slide the rectangles around in the $y$ direction so as to pack them into a strip having minimum $y$ extent. [Solution]

Problem 194 (Designing a Dartboard). A fun problem is that of assigning scores to the sectors of a dartboard. For obvious reasons, a particular arrangement of scores is more “interesting” if there tends to be a large difference between the scores of adjacent sectors. More formally, let’s say we wish to maximize the sum of squares of the score differences over all pairs of adjacent sectors. Given a set of $n$ scores $s_1 \ldots s_n$ to allocate to the $n$ sectors of the dartboard, show how to optimally arrange these scores with a greedy algorithm. [Solution]

Problem 195 (The Fleet Range Problem). Suppose you have a fleet of $n$ vehicles with fuel capacities $c_1 \ldots c_n$ and fuel consumption rates $r_1 \ldots r_n$. All vehicles travel at the same speed and start at a common location. You would like to send the vehicles out in a convoy such that all of them eventually return back to the starting point. Any vehicles leaving the convoy early can donate excess fuel to the remaining vehicles before turning
back (and this is the only time it can exchange fuel with the other vehicles). Devise a greedy algorithm that determines the greatest possible distance such a convoy can reach.

**Problem 196 (Making Change).** This is a classic greedy problem. Given a set of \( n \) coin denominations, we wish to make change for an amount of money \( C \) using as few coins as possible. All coin denominations and the value of \( C \) are integers, and you may assume that the smallest denomination has value 1 (thus ensuring that a solution always exists). The problem is NP-hard, but a greedy algorithm gives an optimal solution for several common special cases. Show that a greedy algorithm optimally solves the problem if each denomination evenly divides the next-highest denomination.

**Problem 197 (The Uncapacitated Exact Transportation Problem with Only Two Sources).** Suppose we have 2 factories and \( n \) warehouses. Our factories supply exactly \( s_1 \) and \( s_2 \) units of goods, respectively, and warehouses \( 1 \ldots n \) have demands of \( d_1 \ldots d_n \), with \( \sum d_i = s_1 + s_2 \). There is a cost \( c_{ij} \) per unit of goods shipped from factory \( i \) to warehouse \( j \). Our goal is to determine a minimum-cost assignment from factories to warehouses so that exactly \( s_i \) units depart from factory \( i \) and exactly \( d_j \) units arrive at warehouse \( j \). This is a special case of a more general problem known as the transportation problem, which can be solved using a variety of methods (see problem 225 for more details). Please give an optimal \( O(n \log n) \) greedy algorithm for this problem.

**Problem 198 (Two-Hub Hierarchical Routing).** You are given as input \( n \) points in the 2D plane. Two of these points must be designated as hubs, and each point must be assigned to exactly one of these hubs, so as to minimize the total pairwise routing cost summed up over all pairs of points. If two points are connected to the same hub, their pairwise routing cost is the sum of their distances to that hub. If they are connected to different hubs, their routing cost is the sum of the distances from the points to their respective hubs plus the distance between the two hubs. Please show how to solve this problem optimally in \( O(n^3 \log n) \) time; the solution of the preceding problem may be of some help.

**Problem 199 (The Knapsack Problem).** This is another classic problem. We are given a knapsack of integer capacity \( C \) and \( n \) items, each with a positive size and a positive value (neither of which is necessarily an integer). Each item taken by itself is small enough to fit in the knapsack. We would like to select a maximum-value subset of the items that fits in the knapsack.

(a) Please show that the knapsack problem can be optimally solved in \( O(n \log n) \) time with a simple greedy algorithm if either all items have unit size, or all items have unit value. Can you improve the running time to \( \Theta(n) \)?

(b) Suppose that we can put fractions of items in the knapsack (this is known as the fractional knapsack problem). Of course, if we put only, say 30% of an item in the knapsack, we only get to count 30% of its value towards the objective value of our solution. Give an \( O(n \log n) \) greedy algorithm which optimally solves this special case of the problem, and then try to reduce its running time to \( \Theta(n) \).

(c) Consider now the 0/1 knapsack problem, where we are only allowed to place zero or one copies of each item in the knapsack; that is, we cannot fractionally add items to the knapsack. Describe a simple greedy 1/2-approximation algorithm for this problem with running time \( O(n \log n) \). Can you improve the running time to \( \Theta(n) \)?

(d) Try to build an \( O(n \log n) \) algorithm that obtains a 2/3-approximate solution for the 0/1 knapsack problem. As a hint, “guess” the single item in the solution having maximum value, and then apply the preceding algorithm as quickly as possible (using an appropriate data structure may help).

(e) A common generalization of the 0/1 knapsack problem is known as the multiple knapsack problem.
### 10.6. ADDITIONAL PROBLEMS

**sack problem**, where as input we are given not one knapsack, but \( m \) different knapsacks with capacities \( C_1 \ldots C_m \). The objective is now to maximize the value we can collectively pack into all of the knapsacks. Devise a simple greedy \( 1/4 \)-approximation algorithm for the multiple knapsack problem. [Solution]

(f) The **knapsack cover problem** is in some sense the opposite of the 0/1 knapsack problem. We have a collection of \( n \) items each with a positive size and cost, and we would like to compute a minimum-cost subset of these items whose total size is at least as big as a some specified “capacity”, \( C \). For example, the items could be types of food where size corresponds to nutritional content. In this case we seek a minimum-cost subset of the foods that collectively provides sufficient nutritional value. Give a simple greedy algorithm that solves the natural fractional version of this problem, and then see if you can obtain a \( 2 \)-approximation algorithm for this problem. [Solution]

(g) An interesting (also NP-hard) variant of the knapsack problem is one that allows us to overfill the knapsack by at most a single item. In other words, a solution is feasible if removing its largest item allows the remaining items to fit within capacity. This type of problem might occur in a situation where we are packing a container such as a railway car with an open top. Can you devise a simple \( 2 \)-approximation algorithm for this variant? [Solution]

**Problem 200 (SONET Ring Loading).** A great deal of traffic on the Internet is currently carried by Synchronous Optical Networks (SONETs), which are often laid out in the shape of a circular ring. We would like to route \( n \) communication sessions through such a network. Each session is described by the location of its two endpoints on the circle, and the amount of demand (i.e., information) that must be routed between these points. We can route each session between its endpoints in one of two ways: clockwise or counterclockwise, and we would like to route each connection so as to minimize the maximum load over the entire network. The load of a point on the network is defined to be the total demand being routed through that point (we don’t count demand for sessions ending exactly at that point). This problem is NP-hard (although it can be solved in polynomial time if all demands are one). Show that we can approximate the optimal solution within a factor of 2 using a simple greedy algorithm. [Solution]

**Problem 201 (Bin Packing).** In the well-known bin packing problem, we are given a collection of \( n \) items and their associated sizes (all no larger than 1). Our goal is to pack the items into unit-sized bins while using as few bins as possible. The problem is NP-hard.

(a) Show how to approximate the optimal solution within a factor of 2 using a greedy algorithm. [Solution]

(b) Consider the following simple greedy algorithm: process the items in decreasing order by size, attempting to fit each one into any bin already in use. If an item doesn’t fit into any of these, open a new bin for the item. This algorithm gives a solution that uses at most \( (3/2)OPT + 1 \) bins, where \( OPT \) denotes the number of bins required by an optimal solution. For a challenge, try to prove this. [Solution]

**Problem 202 (Bin Covering).** Given \( n \) items with varying sizes \( s_1 \ldots s_n \), the bin covering problem asks us to place these items into a maximum number of unit-size bins so that every bin ends up filled at least to its capacity. For example, suppose the \( n \) items represent different pieces of candy, and we would like to give away a maximum number of bags of candy where each bag is at least of some specified fullness. This problem is NP-hard; however, show how to approximate its optimal solution within a factor of 2 using a greedy algorithm. [Solution]

**Problem 203 (Time-Dependent Scheduling).** Consider the problem of sequencing \( n \) jobs on a single machine, where the processing time of job \( j \) is given by a linear function \( a_j t + b_j \), with \( t \) being the time at which the job is started, and \( a_j \) and \( b_j \) both.
being positive. In this setting, \( b_j \) is the initial (“default”) processing time of job \( j \), while
\( a_j \) specifies how the processing time might increase over time as the job becomes more
involved due to its being postponed. Please give a simple \( O(n \log n) \) greedy algorithm for
ordering the jobs to minimize the makespan of the schedule (the latest completion time
over all jobs). [Solution]

Problem 204 (Minimum-Makespan Scheduling on Parallel Machines).
Suppose we have \( n \) jobs, each with associated processing times \( p_1 \ldots p_n \), and we wish
to schedule them by assigning each to one of \( m \) available machines. Our objective is
to minimize the makespan of the schedule (the time at which the most-heavily-loaded
machine completes its processing). This well-known scheduling problem is NP-hard, but
it can be approximated using greedy methods.

(a) Consider the greedy algorithm which processes jobs in arbitrary order, assigning each
in sequence to the machine which is currently the least heavily loaded. Show that this
algorithm produces a schedule having at most twice the optimal makespan. [Solution]

(b) A bit trickier: Show how the same algorithm delivers a schedule of makespan at most
4/3 times optimal if it schedules the jobs in decreasing order of processing time. It
will help to distinguish large jobs (with processing time larger than one third of the
optimal makespan) from the remaining small jobs. [Solution]

(c) In an open shop scheduling environment, every job must be processed by every machine.
This might correspond, for example, to a factory in which each of the \( m \)
machines performs some useful function in assembling a product. A machine can still
process only one job at a time, a job can only be processed by one machine at a
time, and the jobs do not need to visit machines in any specified order (as is the case
with more complicated “shop” scheduling environments). In a manner very similar
to part (a), devise and analyze a greedy 2-approximation algorithm for this problem.
[Solution]

Problem 205 (Successive Divisibility). For many NP-hard problems, greedy algorithms are known to optimally solve certain special cases. In this problem we investigate
special cases of several NP-hard problems that involve input elements with successively
divisible sizes (see also problem 196 on making change, for another example of this type
of special case).

(a) For minimum-makespan scheduling (problem 204, suppose the processing time of each
job evenly divides the processing time of the next-largest job. For the knapsack prob-
lem (problem 199), suppose each item size evenly divides the size of the next-largest
item, and further suppose you can place multiple copies of the same item into the
knapsack. For bin packing (problem 201) and bin covering (problem 202), suppose
each item size evenly divides the size of the next-largest item. Please give optimal
greedy algorithms for each of these special cases. [Solution]

(b) With each of the four proceeding problems, show how to obtain a 2-approximation
algorithm for arbitrary problem instances by first rounding item sizes to nearby powers
of 2, optimally solving the rounded instance, and then reverting item sizes back to their
original values. [Solution]
Optimization is all about making intelligent decisions. In the preceding chapter, we saw how to build up solutions with a sequence of “greedy” decisions focusing only on the short-term improvement of a partial solution at each step. For many problems, however, being near-sighted like this is detrimental, and we need to consider longer-term consequences of our decisions in order to reach an optimal solution. This leads to a powerful albeit confusingly-named technique called dynamic programming (DP), which we study in detail in this chapter.

As a motivating example to explain the high-level idea behind DP, suppose we go to the post office to mail a package that requires $C$ cents in postage. We find a vending machine that sells three types of stamps: 49-cent stamps, 34-cent stamps, and 1-cent stamps. Our goal is to purchase the smallest number of stamps whose total value is exactly $C$. One’s first instinct might be to consider the natural greedy algorithm for this problem: repeatedly select the largest stamp whose value doesn’t exceed our remaining postage. However, the short-sighted nature of this approach leads to trouble down the road. For example, if $C = 68$, the greedy algorithm first chooses a 49-cent stamp, but then is stuck using nineteen 1-cent stamps, while the optimal solution uses just two 34-cent stamps. It therefore seems that we need to look farther ahead when making decisions.

How do we make a more informed decision about which stamp to purchase first? According to Figure 11.1, after making this first decision, we are left with a smaller subproblem of the same form as the original problem. If we already knew the optimal solutions to the three subproblems reachable from the initial decision, the decision becomes easy, since we now understand its long-term ramifications — we would simply choose the first stamp in order to land on the subproblem that requires the minimum number of remaining stamps. This is a crucial insight behind DP: large problems often become much easier to solve after first computing solutions to smaller subproblems embedded within them.

To solve the postage problem with DP, we decompose it into a series of subproblems of the same form. We first compute the minimum number of stamps required to

---

1At the time of writing of this chapter, 49 cents is the current US rate for first-class mail, and 34 cents is the current rate for postcards.

2In the previous chapter, we called this the optimal substructure property. Optimal substructure is perhaps the most important property we need a problem to satisfy for it to allow a DP solution.
make 1 cent in postage, then the minimum number of stamps required to make 2 cents in postage, and so on. Letting $M(c)$ denote the minimum number of stamps required to make exactly $c$ cents in postage, our subproblems are therefore the computation of $M(1)$, $M(2)$, and so on, ending with $M(C)$, the problem we originally wanted to solve. From Figure 11.1, we can deduce that

$$M(c) = 1 + \min\{M(c-1), M(c-34), M(c-49)\}$$

Our entire DP algorithm now consists of applying this formula to compute $M(c)$ for $c = 1$ up to $C$, treating $M(c) = 0$ for $c = 0$ and $M(c) = +\infty$ for $c < 0$ as base cases. By solving subproblems in this order, the formula above allows us to compute each $M(c)$ in just constant time, since we have conveniently already computed the solutions of all the smaller subproblems on which it depends. Our total running time is therefore $\Theta(C)$.

Many algorithmic techniques solve large problems by breaking them into smaller subproblems whose solutions are then recombined to solve the larger problem. For example, a “divide and conquer” algorithm might break a problem of size $n$ into two subproblems of size $n/2$, and an “incremental construction” algorithm might recursively solve a subproblem of size $n - 1$ and then augment it with one final element. DP belongs to this same family of approaches, only specialized to solve optimization problems. Unfortunately, the name “dynamic programming” does little to convey the essence of the technique. Historically, the term “programming” refers not to computer programming but to “mathematical programming”, meaning the formulation and solution of optimization problems. Its closest synonym in this context is probably “planning”. The “dynamic” part of the name refers to the fact that the technique was originally introduced as a means of solving dynamic multi-stage planning problems involving decision-making over time. We have tried to convey the spirit of this idea in our motivation above.

DP is a powerful, highly versatile technique that applies to a wide range of problems in practice. Although it is based on a simple premise, DP can be a relatively difficult

---

3Defining $M(c) = +\infty$ for $c < 0$ ensures that it will never be optimal to choose a solution that lands on a subproblem with $c < 0$, since this represents an infeasible situation.
technique for the introductory student to master. Therefore, like the last chapter, this chapter is also built from a series of examples that highlight the common designs and applications of DP algorithms.

11.1 Example: Shortest Paths in Acyclic Graphs

As our first major example, we consider the well-known single-source shortest path problem. Its input is a directed graph with \( n \) nodes and \( m \) directed edges, each with an associated cost, where we would like to find a shortest (i.e., minimum-cost) path from a designated source node \( s \) to every other node in the graph. An example is pictured in Figure 11.2(a). The shortest path problem is an extremely important and well-studied problem in algorithmic graph theory, one to which we will devote an entire chapter later on. For now, we consider the special case of computing shortest paths through a directed acyclic graph (DAG). We can solve the single-source shortest path problem in a DAG in \( \Theta(m + n) \) time (i.e., linear time). This is the best running time possible, given that we need to at least examine all of the nodes and edges in the graph. Many practical problems involve computation of shortest paths through a DAG; the following is a prototypical example:

**Problem 206 (Project Scheduling).** DAGs often appear in scheduling problems with precedence constraints. Suppose we have \( n \) jobs to schedule, each with associated processing times \( p_1, \ldots, p_n \), and in addition we are given a list of \( m \) precedence constraints among these jobs. Each precedence constraint is a pair \((i, j)\) specifying that job \( i \) must be completed before we can start working on job \( j \). We can work on multiple jobs in parallel, but not if they are related directly or indirectly via precedence constraints. Please show how to use acyclic shortest paths to compute the minimum possible makespan (the latest completion time over all the jobs), as well as for each job, the following two quantities: its earliest start time (the earliest time we may possibly expect to be able to work on the job), and its latest start time (the latest possible time we could start working on the job without adversely affecting the optimal makespan). [Solution]

Our linear-time shortest path algorithm for DAGs can actually be cleverly adapted to compute shortest paths in a general directed graph. When we study shortest paths in Chapter ??, we will see how this leads to the Bellman-Ford algorithm, which computes single-source shortest paths in \( O(mn) \) time.

11.1.1 Building up Optimal Solutions to Subproblems

We now highlight the main steps involved in the design of a DP algorithm, while we build a solution for the acyclic shortest path problem.

(i) **Decompose the problem into a sequence of increasingly larger subproblems, all having the same structure.** For some problems, this step is quite straightforward; for others, determining the right subproblem decomposition can be quite challenging. For acyclic shortest paths and many other problems involving a DAG, a natural subproblem decomposition arises from its topological ordering. Recall from Chapter 3 that we can topologically sort
the nodes of a DAG in linear time, ordering them in a line so that all edges point from left to right. Figure 11.2(a) shows an example. We assume the source node \( s \) is the leftmost node in the ordering (node 1), since nodes to the left of \( s \) would be unreachable from \( s \) and can be ignored. Letting \( c[j] \) denote the shortest path cost from \( s \) to node \( j \) (after topologically sorting as a preprocessing step), our DP algorithm will compute \( c[1], c[2], \ldots, c[n] \) as its sequence of subproblem solutions. Subproblems are often of this form (i.e., “what is the best solution up to and ending at position \( j \)” for \( j = 1, 2, \ldots \)), corresponding to solutions of increasingly-larger “prefixes” of the full problem.

(ii) **Express the solution of a large subproblem recursively in terms of solutions to smaller subproblems.** A shortest \( s \rightarrow j \) path arrives at \( j \) along some final edge \((i, j)\), where \( i < j \) is an earlier node in our topological ordering. As shown in Figure 11.2(b), we can therefore compute \( c[j] \) by looking for the best such incoming edge — the one minimizing the cost of the edge \((i, j)\) itself (denoted by \( \text{cost}(i, j) \)) plus the cost of the optimally solving the left-over subproblem of finding a shortest path from \( s \) to \( i \):

\[
c[j] = \min_{\text{incoming edges } (i,j)} \{c[i] + \text{cost}(i,j)\}.
\]

Note that this formula only depends on solutions to smaller subproblems we will already have computed — that is, values \( c[i] \) for \( i < j \).

These two steps represent 90% of the difficult work in developing our algorithm. We now only need to specify appropriate base cases (here, \( c[1] = 0 \)) and to loop over all subproblems in the appropriate order to solve them. According to Figure 11.3(a), this involves nothing more than wrapping the formula above in a “for” loop. The final implementation is quite simple, and this is typical for DP algorithms; the
11.1. EXAMPLE: SHORTEST PATHS IN ACYCLIC GRAPHS

0. Initialization: $c[1] \leftarrow 0$, $c[2 \ldots n] \leftarrow +\infty$, $b[1 \ldots n] \leftarrow \text{nil}$

1. For $j \leftarrow 2$ to $n$:
   2. For all incoming edges $(i,j)$ of node $j$:
      (a) 3. If $c[i] + \text{cost}(i,j) < c[j]$:
      4. $c[j] \leftarrow c[i] + \text{cost}(i,j)$
      5. $b[j] \leftarrow i$

1. For $i \leftarrow 1$ to $n-1$:
   2. For all outgoing edges $(i,j)$ of node $i$:
      (b) 3. If $c[i] + \text{cost}(i,j) < c[j]$:
      4. $c[j] \leftarrow c[i] + \text{cost}(i,j)$
      5. $b[j] \leftarrow i$

Figure 11.3: Computing shortest paths through a DAG, assuming it has already been topologically sorted. In (a), we compute $c[j]$ for $j = 1 \ldots c[n]$ in sequence, while (b) is quite similar but uses “pushing” instead of “pulling”. Both approaches also compute backpointers $b[1 \ldots n]$.

Pulling Versus Pushing. In order to compute $c[j]$ using the formula from (ii), each node in the DAG needs to know its incoming edges. Unfortunately, directed graphs are often stored by having nodes keep track of their outgoing, rather than incoming edges. Figure 11.3(b) shows how we can adapt our solution in this case. Since each subproblem knows which larger problems are consumers of its solution, it “pushes” its optimal solution forward to these larger problems, instead of “pulling” solutions from smaller subproblems. The value of $c[j]$ is no longer computed by processing the explicit formula from (ii) when we visit node $j$, but rather it is computed implicitly during the course of visiting the earlier subproblems upon which $c[j]$ depends. By the time our algorithm reaches node $j$, the value $c[j]$ will already be computed, so all of the work spent on this subproblem will be invested in pushing its value forward to larger subproblems.

Whether “pulling” or “pushing” is appropriate depends on problem structure, for example whether subproblems have access to backward links to smaller subproblems or forward links to larger subproblems (in fact, these two approaches are sometimes called backward and forward DP). When both methods apply, they generally give the same running time.

Reconstructing a Solution via Traceback. As discussed so far, our DP algorithm computes the cost of the shortest path to every node, but not the actual paths themselves. This is a common trait of DP algorithms — we end up with the value of the optimal solution, but not its structure (e.g., the individual elements
we selected to be part of the solution). Fortunately, it is usually easy to find the structure as well, by storing for each subproblem not only the value of its optimal solution, but also a backpointer to the previous subproblem(s) on which its value is based. In our example, we compute the shortest path cost \( c[j] \) by minimizing over \( j \)'s incoming edges, so the pseudocode in Figure 11.3 computes a backpointer \( b[j] \) to the node \( i \) corresponding to which edge \((i,j)\) was best among these. After the algorithm terminates, we can then follow the sequence of backpointers from any node \( j \) back to the source to trace out the shortest \( s \sim j \) path in reverse. This technique works for nearly any DP algorithm. By following the so-called traceback path of backpointers starting from the final subproblem, we can reconstruct (in reverse) the chain of decisions we used to optimally solve that problem. These decisions may tell us, for example, the set of elements that were included in the solution.

Now that you are familiar with the basics, here are some simple problems you can use to practice formulating DP solutions:

**Problem 207 (Maximum Value Subarray).** Given an array \( A[1 \ldots n] \) of numbers (both positive and negative), we would like to find a subarray \( A[i \ldots j] \) whose sum is maximized. We introduced this problem back in Section 2.3, where we showed that the obvious “brute force” solution to this problem runs in \( \Theta(n^3) \) time, although we can reduce this to \( \Theta(n^2) \) or even \( \Theta(n \log n) \) with a small amount of extra cleverness. Show how to solve it in only \( \Theta(n) \) time using DP. [Solution]

**Problem 208 (Longest Increasing Subsequence).** Given an array \( A[1 \ldots n] \) of numbers, a subsequence of \( A \) is just a subset of its elements \( A[i_1], A[i_2], \ldots, A[i_k] \) where \( i_1 < i_2 < \ldots < i_k \). Note that the elements in a subsequence do not need to form a contiguous block (in contrast to a subarray). We say a subsequence is increasing if \( A[i_1] < A[i_2] < \ldots < A[i_k] \). Give an \( O(n^2) \) DP algorithm that finds a longest increasing subsequence of \( A \) — that is, a subsequence containing a maximum number of elements. We will improve this running time to \( O(n \log n) \) later in the book in problem ?? by leveraging a duality relationship between this problem and the problem of partitioning a sequence into a minimum number of disjoint non-increasing subsequences. [Solution]

**Problem 209 (Interval Covering and Packing).** Let us revisit the interval packing and covering problems from Section 10.5 in the previous chapter. As input, we are given a set of \( n \) intervals, each one being a subinterval of \([0,1]\) on the number line. A packing of intervals is a subset of intervals that are all mutually disjoint from each-other, and a covering of intervals is a set of intervals whose union is \([0,1]\).

(a) If our intervals have values \( v_1 \ldots v_n \), give an \( O(n^2) \) DP algorithm that computes a maximum-value interval packing. Can you improve its running time to \( O(n \log n) \)? [Solution]

(b) If our intervals have costs \( c_1 \ldots c_n \), give an \( O(n^2) \) DP algorithm that computes a minimum-cost interval cover. Can you improve it to \( O(n \log n) \) time? [Solution]

### 11.1.2 Dynamic Programs as Computations on DAGs

We can gain some additional insight into the problems we have studied thus far by showing how they share quite a bit of common structure. In fact, the reason we chose shortest acyclic path as our first main example problem is that this problem is somewhat “canonical” in that all of the problems we have seen so far can be viewed as special cases of it (or of the equivalent longest path problem in a DAG).
11.1. EXAMPLE: SHORTEST PATHS IN ACYCLIC GRAPHS

Figure 11.4: Representing the (a) postage problem, (b) maximum subarray problem, (c) maximum interval packing problem, and (d) longest increasing subsequence problem as a shortest or longest path problem in a DAG. To reduce clutter, we have abbreviated some of the edges out of the “start” node and into the “end” node in (b), (c), and (d). See also Figure 12.4(b) for a picture of an interval covering problem as a shortest path problem in a DAG.

Consequently, the DP algorithms for solving these problems can be viewed as just specializations of the generic algorithm we devised earlier to solve the shortest path problem in a DAG.

• Figure 11.4(a) shows our earlier postage problem expressed as a shortest path problem in a DAG. Here, we want to make $C = 8$ cents worth of postage using a combination of 1-cent, 2-cent, and 5-cent stamps, and we can view this as computing a shortest path through a DAG whose nodes represent the amount of postage we have accumulated. Edges each have unit cost, and represent the use of 1-cent stamps (horizontal edges), 2-cent stamps (lower edges), and 5-cent stamps (upper edges).

• Figure 11.4(b) shows how to express a maximum value subarray problem over the sequence $A = 4, -7, 6, -2, 3, -5$ as a longest path problem through a DAG. Edge values incident to the “dummy” start and end nodes are zero. The value of a path from start to end corresponds to the sum of a subarray.

• In Figure 11.4(c), we see how to represent the maximum interval packing problem as a longest path problem in a DAG whose nodes represent intervals, with an edge joining interval $i$ with some later interval $j$ if $i$ and $j$ are disjoint. Here, a longest path from a dummy start node (with edges outgoing to all intervals) to a dummy end node (with edges incoming from all intervals) corresponds to a maximum interval packing.

• Figure 11.4(d) shows the longest increasing subsequence problem on the sequence $A = 5, 2, 8, 4, 7$ similarly represented as a longest path problem, where there is an edge from $A[i]$ to $A[j]$ if $i < j$ and $A[i] < A[j]$.
CHAPTER 11. DYNAMIC PROGRAMMING

Figure 11.5: The exponential-sized tree of all recursive calls generated when computing the Fibonacci number $F_7$ using the straight recursive `FibRec` function from Figure 11.6(a). By employing memoization, the function `FibMemo` from Figure 11.6(b) reduces the running time dramatically by visiting only the subproblems shown in bold.

If you are a visual thinker, you may want to continue trying to visualize DP algorithms as computations on DAGs of subproblems, even for the more sophisticated examples later in the chapter that do not map directly to shortest or longest path problems. Nodes correspond to subproblems, and each node has incoming edges from the subproblems on which its value depends. The graph is acyclic since edges only go from smaller to larger subproblems. You might even want to view this sort of DAG as an abstract computing machine, since each node represents the computation of a simple function based on values it receives from its incoming edges. As we will see moving forward, subproblems can often also be naturally arranged in a sequence, grid, or tree, giving other nice options for visual layout.

11.1.3 Top-Down Versus Bottom-Up Computation

Since a DP formulation tells us a recursive formula for solving large subproblems, we might be tempted to use nothing more than straight recursion as our solution. As a simple example, the $n$th Fibonacci number is recursively defined as $F_n = F_{n-1} + F_{n-2}$, so we could use the code in Figure 11.6(a) to compute $F_n$ in a purely recursive manner. Although this will ultimately give a correct answer, it will take an exceptionally long time to do so. The tree of recursively-generated subproblems (Figure 11.2) grows at an exponential rate, having $\Theta(\phi^n)$ nodes.

The drawback of straight recursion is that we end up solving the same subproblems over and over. To remedy this, we employ a technique known as memoization. Once we have solved a subproblem, we store its solution in a table (or in keeping with the name of the technique, we make a “memo” of its solution). If that subproblem is ever generated again, we can return its solution immediately without further recursion. Although memoization only adds a small amount of code to our implementation (Figure 11.6(b)), its running time benefits are substantial, as computing $F_n$ now requires only $\Theta(n)$ time (ignoring for a moment the issue of word size, since the number of digits in $F_n$ grows at an exponential rate as $n$ increases).
11.1. EXAMPLE: SHORTEST PATHS IN ACYCLIC GRAPHS

FibRec\(j\):
1. If \(j = 0\): Return 0
2. If \(j = 1\): Return 1
3. Return FibRec\(j - 1\) + FibRec\(j - 2\)

Initialization: \(f[0] \leftarrow 0, f[1] \leftarrow 1, f[2 \ldots n] \leftarrow \text{nil}\)

FibMemo\(j\):
1. If \(f[j] \neq \text{nil}\): Return \(f[j]\)
2. \(f[j] \leftarrow \text{FibMemo}(j - 1) + \text{FibMemo}(j - 2)\)
3. Return \(f[j]\)

(a) During a town festival, \(n\) local businesses are planning to give away free products and food, but only at certain times of the day. In particular, business \(i\) will be giving away

FIGURE 11.6: Computing the \(n\)th Fibonacci number (a) via straightforward recursion (in exponential time), and (b) using top-down recursion plus “memoization” (in linear time). In (c), we see the more standard DP approach that solves subproblems in a bottom-up fashion.

In contrast to the top-down recursive approach above, most DP algorithms are traditionally implemented in a “bottom-up” fashion, looping over all possible subproblems from smallest to largest, as in Figure 11.6(c). However, the “bottom-up” and “top-down with memoization” approaches are generally equivalent in terms of worst-case running time, since both ultimately involve solving the same collection of subproblems, each at most once. In our example here, both run in \(\Theta(n)\) time. In fact, it is possible that for some problems, a top-down approach could run faster if it manages to only visit only a small subset of all possible subproblems during its recursive expansion (e.g., suppose we had defined \(F_n = F_{n-61} + F_{n-83}\)). However, since the bottom-up approach is “leaner”, involving only loops and no recursive function overhead, it may still win in practice. Both methods can be useful to keep in your algorithmic toolbox.

From our discussion above, we have now accumulated sufficient insight to characterize the types of problems for which DP is an effective solution technique:

• The problem must satisfy the optimal substructure property, allowing us to express the solution of large subproblems recursively in terms of solutions of smaller subproblems,

• The total number of distinct subproblems should be rather small, and

• A top-down recursive algorithm would end up solving the same subproblems multiple times, making straightforward recursion inadvisable.

Problem 210 (More Practice Problems). The only way to become an expert at DP is by solving as many problems as possible. Here are a few more examples of simple problems to work through for practice.

(a) During a town festival, \(n\) local businesses are planning to give away free products and food, but only at certain times of the day. In particular, business \(i\) will be giving away
Dynamic programming is frequently used in typsetting applications, to avoid ugly gaps next to the right margin. (b) Box:

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
\end{array}
\]

\(\text{Box: } #1\) \(\text{Box: } #2\) \(\text{Box: } #3\)

Dynamic programming is frequently used in typsetting applications, to avoid ugly gaps next to the right margin.

Figure 11.7: Illustrations for problems: (a) stacking conical frustra, (b) building bridges, (c) box packing, and (d) typesetting.

free items only at time \(T_i\). We know, for every pair of businesses \((i, j)\), the amount of time \(t_{ij} > 0\) it takes to walk from \(i\) to \(j\) (you may assume the \(t_{ij}\)'s satisfy the triangle inequality, so the fastest way to walk from \(i\) to \(j\) is the direct route that takes \(t_{ij}\) units of time). Please design an \(O(n^2)\) DP algorithm that computes the maximum number of businesses we can successfully visit. We start at business 1 (not necessarily the business with minimum \(T_i\)) at time 0. [Solution]

(b) A sequence of \(n\) crates with sizes \(s_1 \ldots s_n\) arrives via truck at a train station and must be packed onto a series of rail cars. Each rail car holds a minimum total size of \(b\) units and a maximum total size of \(B\) units. Rail cars must be loaded according to the order in which crates arrive. For example, crates 1 \ldots 4 might be placed on the first rail car, after which that car departs and then crates 5 and 6 are loaded onto the next car, and so on. Using DP, please determine if it is possible to pack the crates at all, and if so, what is the minimum number of rail cars we need. Try to achieve a running time of \(O(n^2)\). For an extra challenge, can you improve the running time to \(O(n \log n)\) or even \(O(n)\)? [Solution]

(c) Given a numeric sequence \(A_1 \ldots A_n\), please describe an \(O(n^2)\) algorithm for computing a monotonically nondecreasing sequence \(B_1 \ldots B_n\) minimizing the distance between \(A\) and \(B\), defined as \(\sum_i |A_i - B_i|\). As a hint, refer back to problem 60. [Solution]

(d) A conical frustum is a horizontal slice of a cone. You are given a set of \(n\) conical frustra, specified by the radii of their bases \(R_1 \ldots R_n\), the smaller radii of their tops \(r_1 \ldots r_n\), and their heights \(h_1 \ldots h_n\). Using a subset of these frustra, you would like to build a tower that is as tall as possible. Of course, in the interest of stability, you can only stack frustum \(i\) on top of frustum \(j\) if \(R_i \leq r_j\), as shown in Figure 11.7(a). Please design an \(O(n \log n)\) dynamic programming algorithm that determines the height of the tallest possible tower. [Solution]

(e) Consider the diagram shown in Figure 11.7(b). You are told the \(x\)-coordinates of \(n\) cities along the northern bank of a river, and also of \(n\) cities along the southern bank. The cities along each bank are numbered from 1 \ldots \(n\), and you would like to build bridges to connect equally-numbered pairs of cities with bridges. However, you are
not allowed to build two bridges that cross each-other. Give an $O(n^2)$ DP algorithm that determines the maximum number of bridges you can build. [Solution]

(f) You are responsible for packaging a shipment of $n$ rectangular objects into boxes. The objects will arrive in a particular order $1 \ldots n$ on a conveyor belt, and you know the heights $h_1 \ldots h_n$ and widths $w_1 \ldots w_n$ of all the objects in advance. You must split the sequence of objects into contiguous groups that are then to be packed into boxes. As shown in Figure 11.7(a), each group is arranged from left to right and then packed into a rectangular box whose dimensions are as small as possible. You are given a per-unit-length cost $A$ of box material (so a set of boxes having total perimeter $P$ would cost $AP$ in total), and you are also given a per-unit-area penalty $B$ for each unit of empty space we pack. Subject to these costs, please give an $O(n^2)$ DP algorithms for computing a minimum-cost box packing. [Solution]

(g) DP is frequently used in typesetting applications, to avoid ugly gaps next to the right margin, as shown in Figure 11.7(d). Suppose you are told the widths $w_1 \ldots w_n$ of $n$ sequential words that you would like to typeset on a page of width $W$. Depending on how you choose to insert line breaks, there will be some amount of space left over on each line between the last word on the line and the right-hand margin — we call these spaces gaps. Please devise an $O(n^2)$ DP algorithm that minimizes the sum of squared lengths of all gaps except the gap on the last line of text. [Solution]

(h) You are managing a large project consisting of $n$ tasks. Each task $i$ has an associated value $v_i$ you receive for completing the task and a time $t_i$ marking the exact moment at which the task must be started, if you choose to perform the task at all. Unfortunately, The duration of each task is not known with certainty until after the task is completed: task $i$ takes either $d_i$ or $d'_i$ units of time to complete, each with probability $1/2$ (and also independent of the duration of all other tasks). As time progresses, you must decide for each task $i$ (when we reach time $t_i$ and find that no other task is currently pending) whether you will skip the task or perform the task. If you elect to perform task $i$, no other tasks can be performed until $i$ is completed. Please give an $O(n \log n)$ time dynamic programming algorithm that computes a scheduling policy of maximum expected value. [Solution]

(i) You are again managing a large project with $n$ tasks. Each task $i$ has an associated value $v_i$ you receive for completing the task, a known duration $d_i$, and a time $t_i$ at which the task must be started. It may not be possible to perform all the tasks, so you would like to perform a subset of tasks of maximum total value. Furthermore, you can direct your workers to “rush” up to two tasks (a rushed task takes only half the duration it normally would require), but since rushing a task requires significant effort you cannot perform these two rushed tasks in a row. Please show how to compute an optimal schedule in $O(n \log n)$ time using dynamic programming. [Solution]

11.2 Example: The Knapsack Problem

So far, all of our DP problems have been “one dimensional”, involving a single linear chain of subproblems. These are among the simplest DP problems, and the best suited for introducing the technique. We now move from one dimension to two, introducing a common type of DP algorithm that involves filling in a two-dimensional table of subproblem solutions row by row.

Our next example is a classic problem in discrete optimization called the knapsack problem. Its input consists of $n$ items each having sizes $s_1 \ldots s_n$ and values $v_1 \ldots v_n$, and the goal is to compute a maximum-value collection of items that can be packed
into a knapsack of some specified capacity $C$. We focus on the integer knapsack problem, where $C$ and all item sizes are integers; values can still be non-integral.

In problem 199 in the preceding chapter, we showed how to solve the fractional knapsack problem (where fractional quantities of items may be placed in the knapsack) in $\Theta(n)$ time using greedy techniques. However, when we require whole items to be placed in the knapsack in their entirety, the problem becomes much more challenging — NP-hard, in fact. Consequently, the algorithms we develop will have only pseudo-polynomial running times\(^4\) like $O(nC)$. Although these algorithms are technically not “efficient”, they are widely used in practice for problems in which $C$ is not too large.

A natural one-dimensional subproblem decomposition would involve solving for the best way to fill smaller knapsacks of capacity 1, 2, and so on, ending with our final capacity of $C$. However, this doesn’t quite work for the following reason: suppose our first decision involves inserting item 7 into the knapsack. To complete our solution, we would like to solve the remaining subproblem “What is the maximum value one can pack into a capacity $C-s_7$ knapsack without using item 7?”. Unfortunately, all we know is the solution to the subproblem “What is the maximum value one can pack into a capacity $C-s_7$ knapsack?”. In other words, our subproblems don’t contain enough information to remember which items we have already used. To address this, we use a formulation involving a two-dimensional space of subproblems:

\[
V[i, j]: \text{ The maximum value one can pack into a knapsack of capacity } j \text{ using only some subset of items } 1 \ldots i.
\]

\[
V[i, j] = \max\{V[i-1, j], V[i-1, j-s_i] + v_i\}.
\]

Base cases: $V[i=0, j] = V[i, j=0] = 0$, $V[i, j < 0] = -\infty$.

Each subproblem $V[i, j]$ is solved in $O(1)$ time by deciding between the best possible value we can obtain (a) if we do not include item $i$, or (b) if we do. Filling in the table of all subproblem solutions row-by-row for $i = 1 \ldots n$ and $j = 1 \ldots C$ takes $\Theta(nC)$ time, after which the optimal solution to the overall problem resides in $V[n, C]$. A traceback path starting from this location reveals the set of items selected for inclusion in the optimal solution. [More detailed elaboration on this solution]

The following problems give us more practice solving similar “two-dimensional” DP problems where we fill in a table of subproblem solutions one row at a time.

**Problem 211 (Additional Knapsack Variants).** This problem highlights some common variants of the knapsack problem.

(a) **Multiple Copies of Items Allowed.** If multiple copies of each item can be placed in the knapsack, please show how this problem can by solved with a simple “one-dimensional” formulation leading to an $O(nC)$ DP algorithm. [Solution]

(b) **High Multiplicity Knapsack.** Suppose we specify along with each item $i$ a count $k_i$ governing the maximum number of copies of item $i$ one can include in a solution. For a challenge, please give an $O(nC)$ DP algorithm for this variant. [Solution]

\(^4\)Recall from Section 1.4.6 that an $O(nC)$ running time is pseudo-polynomial since it depends directly on $C$; a dependence on $\log C$ would be necessary for a (weakly-)polynomial running time.
11.2. EXAMPLE: THE KNAPSACK PROBLEM

(c) **Integer Values.** Please give a fast DP algorithm for solving a 0/1 knapsack problem in which item values rather than sizes are integers. [Solution]

**Problem 212 (Subset Sum).** Given a set \( S = \{a_1 \ldots a_n\} \) of \( n \) positive integers and a target value \( T \), the NP-hard *subset sum problem* involves finding a subset of \( S \) whose elements sum to \( T \), if one exists. Give a DP algorithm that solves a slightly more general problem: find a subset of \( S \) whose elements sum to a value as close to \( T \) as possible. Your algorithm should run in \( O(nC) \) time, where \( C = \sum_i a_i \). [Solution]

**Problem 213 (Edit Distance and Sequence Alignment).** In this problem we investigate several algorithmic approaches for “aligning” two similar strings to measure their similarity or dissimilarity. Algorithms for these problems and their relatives have been studied extensively in the literature. For example, in bioinformatics they are used to measure similarity between related DNA sequences. A related technique called *dynamic time warping* is used for aligning general time series data, often to discover patterns occurring at different rates (e.g., motion capture data of an individual walking versus running, or speech signal data of the same sound but spoken at different speeds). Related results appear also in problems ?? and ??.

(a) **Edit Distance.** A common way to measure dissimilarity between two objects is to compute the cost for transforming one into the other. The minimum cost of transforming from string \( A \) to string \( B \) is known as the *edit distance* between \( A \) and \( B \) (sometimes also *Levenshtein distance*, named after the first author to introduce the concept). While transforming \( A \) into \( B \), we are allowed to use the following operations: insertion of a new character into \( A \) (at a cost of \( C_1 \)), deletion of a character from \( A \) (at a cost of \( C_4 \)), and replacement of a character in \( A \) with another character (at a cost of \( C_5 \)). Given these costs, design an \( O(mn) \) dynamic programming algorithm that computes the edit distance between \( A \) (length \( m \)) and \( B \) (length \( n \)). [Solution]

(b) **Maximum-Similarity Alignment.** Let us define a character-to-character similarity function \( f(c_1, c_2) \) that expresses the similarity of two characters \( c_1 \) and \( c_2 \). Another way to compare two strings \( A \) and \( B \), we first insert spaces (i.e., “empty” characters) into various locations in both strings so that the two have equal length, such that at no common location do we find a space in both strings. This is known as an *alignment* of \( A \) and \( B \). We can compute the value of this alignment by summing up the similarity of each pair of corresponding characters across the two strings (our similarity function \( f \) also specifies the value of aligning a character against a space — often this value is zero). Our goal is to find an alignment with maximum similarity. For example, if we have strings \( A = \text{"SIMILAR"} \) and \( B = \text{"ALIGNMENT"} \), then if characters nearby in the alphabet are treated as being very similar, the optimal alignment might be:

\[-S-I-M-I-L-A-R\]
\[-A-L-I-N-G-M-E-N-T\]

Give an \( O(mn) \) DP algorithm that computes an alignment of maximum similarity between strings of length \( m \) and \( n \). Show also how the edit distance problem is nothing more than a special case of this problem. [Solution]

(c) **Pattern Matching with up to \( k \) Differences.** Suppose you want to find every offset within a larger text at which a specified pattern approximately matches, meaning here that the edit distance is at most \( k \) (with unit costs for insertion, deletion, and modification used in the edit distance calculation). For a challenge, please show how to extend the solution to problem 152 to solve this problem in \( O(kn) \) time, where \( k \) is provided as input. Just as in problem 152, please extend your algorithm to report the offsets achieving minimum edit distance also in \( O(kn) \) time, where \( k \) is the minimum edit distance over all offsets, and as such is not known to you in advance. As a result, this gives an \( O(kn) \) algorithm for computing the unit-cost edit distance \( k \) between two strings of combined length \( n \). [Solution]
Problem 214 (Common Subsequences and Supersequences). Here are a few more well-studied string problems that typically involve DP solutions.

(a) In the longest common subsequence problem, you are given two strings of lengths \(m\) and \(n\), and you wish to find the longest possible string that is a subsequence of each of them (recall that a subsequence does not need to be contiguous). Use DP to solve this problem in \(O(mn)\) time. [Solution]

(b) The shortest common supersequence problem gives you two strings of lengths \(m\) and \(n\), and asks for the shortest string that contains each of them as a subsequence. Use dynamic programming to solve this problem in \(O(mn)\) time. [Solution]

(c) The longest common subsequence of two strings \(A\) and \(B\) of length \(m\) and \(n\) can be computed in \(O(mn)\) time using the solution to part (a). However, if \(A\) and \(B\) share very few characters in common, one can solve the problem much faster. For each character \(c\) in \(A\), let \(r_A(c)\) denote the number of occurrences of \(c\) in \(A\), let \(r_B(c)\) be the number of occurrences of \(c\) in \(B\), and let \(r = \sum_c r_A(c)r_B(c)\). See if you can give a clever reduction from the longest common subsequence problem to a longest increasing subsequence problem (problem 208) over a sequence of length \(r\). Using the fast algorithm for longest increasing subsequence we develop later in problem ??, we can therefore solve the longest common sequence problem in just \(O(r \log r)\) time. In the worst case \((r = mn)\), this is inferior to our previous dynamic programming approach; however, for smaller values of \(r\), it can be much faster. [Solution]

Problem 215 (One-Dimensional Clustering). We study clustering and other data analysis problems extensively in Chapter ???. Whereas most clustering problems are quite challenging in higher dimensions, they are often approachable with DP in one dimension. Consider dividing points \(x_1 \ldots x_n\) into a specified number of \(k\) clusters, as in Figure 11.8. The \(k\)-cluster problem asks us to do this by minimizing the largest cluster diameter (its maximum point minus its minimum point). Other objectives ask us to compute not only a partition into clusters, but also to designate a center point for each cluster, which may not necessarily be one of the \(n\) points in our input. Clustering problems with objectives of this type include the \(k\)-means problem (minimize the sum of squared distances between each of our points and its respective cluster center), the \(k\)-medians problem (minimize the sum of distances between each point and its cluster center), and the \(k\)-center problem (minimize the largest distance from any point to its cluster center). It is not too difficult to show that the \(k\)-center and \(k\)-cluster problems are equivalent in one dimension.

(a) For each of the clustering problems above (\(k\)-cluster, \(k\)-means, \(k\)-medians, and \(k\)-center), give an \(O(n^2k)\) DP algorithm that generates an optimal solution. Does it affect the complexity of your algorithm if you also provide a lower bound and an upper bound on the number points allowed in a cluster? [Solution]

(b) If you are feeling ambitious, give an \(O(nk)\) solution to the \(k\)-center problem. [Solution]

(c) Consider the \(k\)-center problem in which \(k\) of the input points are designated as “anchor” points that must end up belonging to different clusters. Subject to this new constraint, show how to compute an optimal solution in \(\Theta(n)\) time. [Solution]
Figure 11.9: An optimal binary search tree (drawn sideways, so it fits more easily on the page) built from a collection of the most common ingredients extracted from a large set of recipes. Based on the relative frequencies of the ingredients, the average lookup in this tree will examine only 3.44 nodes, even though two thirds of the elements require either 4 or 5 node examinations to locate.

11.3 Example: Optimal Binary Search Trees

A balanced binary search tree (BST) is a natural choice for storing a dictionary of \( n \) elements, allowing fast lookup via the \textit{find} operation in just \( O(\log n) \) time. However, if we are also given access probabilities \( p_1 \ldots p_n \) for the elements, we can do even better: the \textit{optimal} BST problem asks us to tune the shape of our BST to minimize the expected search depth for a randomly-chosen element (element \( i \) being chosen with probability \( p_i \)). As an example, Figure 11.9 gives the optimal BST on a set of common cooking ingredients.

To compute the optimal BST on a set of elements ordered \( 1 \ldots n \), the natural first decision is to pick which element \( r \) goes at the root, after which the left and right subtrees can be recursively constructed from elements \( 1 \ldots r - 1 \) and \( r + 1 \ldots n \). These smaller subproblems are themselves instances of the optimal BST problem, leading us to the following DP formulation. [Detailed derivation]

\[
D[i, j]: \text{Expected search depth for an optimal BST constructed from only elements } i \ldots j, \text{ normalized by multiplying by } p_{ij} = p_i + \ldots + p_j.
\]

\[
D[i, j] = p_{ij} + \min_{r=i..j} \{ D[i, r-1] + D[r+1, j] \}.
\]

Base case: \( D[i, i-1] = 0 \).

This \( O(n^3) \) DP algorithm represents a different structure than we have seen before, a structure that is common to many other problems. To begin with, our \( n \times n \) table
CHAPTER 11. DYNAMIC PROGRAMMING

of subproblem solutions \( D \) isn’t filled in row by row, but diagonal by diagonal. We first compute \( D[i, i] \) for all \( i \), then \( D[i, i+1] \) for all \( i \), then \( D[i, i+2] \), and so on, until we eventually solve the top-level problem \( D[1, n] \) involving all the elements \( 1 \ldots n \). Furthermore, our subproblems decomposition is more “divide and conquer” than “incremental construction” in flavor, as we are decomposing a problem into two large pieces in each recursive step, instead of repeatedly adding one element at a time to build up incrementally larger solutions to prefixes of our problem. You may hear this sometimes called “nested parenthesis” DP, as motivated by the following problem involving multiplication of a chain of matrices \( A_1 A_2 \ldots A_n \), since the top-level decision involves parenthesizing the expression into two high-level blocks \((A_1 \ldots A_r)\) and \((A_{r+1} \ldots A_n)\) that are then multiplied together. Problems amenable to this type of decomposition tend to be those involving objects with a similar “balanced parenthesis” structure, such as the large class of combinatorially-equivalent structures introduced in Section 2.6.

Problem 216 (Matrix Chain Multiplication). Many applications require the multiplication of sequences of matrices. Suppose you are given a sequence of \( n \) matrices \( A_1 \ldots A_n \), and you would like to compute their product \( A_1 A_2 \ldots A_n \). Since matrix multiplication is associative, we can parenthesize such an expression as we see fit, and the parenthesization we choose can have a significant impact on the amount of time it takes to evaluate the expression. For example, suppose we have four matrices \( A_1(1 \times 100) \), \( A_2(100 \times 100) \), \( A_3(100 \times 100) \), and \( A_4(100 \times 1) \). If we compute the product \((A_1 A_2)(A_3 A_4)\) using the straightforward algorithm for matrix multiplication, this requires 20100 individual multiplications. However, if we instead compute \((A_1(A_2 A_3))A_4\), this requires in excess of 1 million multiplications! Devise an \( O(n^3) \) DP algorithm that optimally parenthesizes our matrix expression so as to minimize the number of individual multiplications required for its evaluation. [Solution]

Problem 217 (Parsing with a Context-Free Grammar). A context-free grammar is a set of rules that can be applied to generate various types of strings. For example, consider the set of all strings consisting of one or more 0s followed by the same number of 1s. A simple context-free grammar that generates this set consists of the two rules (also called productions) \( S \rightarrow 01 \) and \( S \rightarrow 0S1 \). Here, the symbols 0 and 1 are known as terminal symbols since they may exist in our final string generated by the grammar, and the symbol \( S \) is called a nonterminal symbol since it cannot be present in our final string. To generate a string using our grammar, we start with the initial string \( S \) and then apply a sequence of productions in some order until we obtain a string containing only terminal symbols, at which point we stop. Any time we have a string containing nonterminal symbols, we must continue to apply productions (each of which replaces a single nonterminal symbol with a string containing terminal and nonterminal symbols) until we finally get rid of all the nonterminals. Context-free grammars are quite powerful in their ability to describe many different classes of strings — for example valid arithmetic expressions, computer programs, and so on. They are called “context free” because each production (e.g., \( S \rightarrow 0S1 \)) has only a single nonterminal symbol on its left-hand side, so it can be applied to any instance of \( S \) in our string regardless of its surrounding context. It turns out that any context-free grammar can be converted fairly easily into a simple canonical form known as Chomsky normal form (CNF), where each production transforms a single nonterminal symbol into either two nonterminal symbols (e.g., \( S \rightarrow AB \), with uppercase letters denoting nonterminal symbols, as is the usual convention), or a single terminal symbol (e.g., \( S \rightarrow 0 \)). Given a context-free grammar \( G \) with \( k \) productions written in CNF form, please describe a simple \( O(n^3k) \) DP algorithm for deciding whether or not \( G \) can generate a given string of length \( n \). [Solution]
Problem 218 (Prediction of RNA Folding). In biology, RNA molecules are long polymers (chains) consisting of four types of bases (small molecular units that can hook together into a chain): Adenine (A), Guanine (G), Cytosine (C), and Uracil (U). The structure of these bases is such that A and U tend to stick together, as do G and C. These affinities (and many other factors) cause RNA molecules to fold into certain shapes. Since folded RNA molecules have been found to carry out many crucial biological functions (catalyzing reactions, etc.), an important problem in computational biology is the prediction of the folding structure of an RNA molecule based on its sequence of bases. In this problem, we concern ourselves only with the secondary structure of an RNA folding; that is, what parts of the RNA chain bond to what other parts, and not the geometric aspects of how this folding is realized in three dimensions. Figure 11.10(a) illustrates a folding of a short RNA molecule, which we can also depict as in Figure 11.10(b) by drawing the chain of bases and indicating which are linked. We call a folding pseudoknot-free if the pairing of bases is properly nested as in our example (i.e., no lines cross). It is not too difficult to see that pseudoknot-free foldings look like trees. Many RNA sequences are known to have pseudoknot-free foldings, so we will focus on this case since it is computationally much simpler. An RNA molecule tends to fold into a configuration that minimizes its free energy. Every A-U or C-G bond formed has a stabilizing effect and lowers the free energy. On the other hand, if these bonds bend the structure so it has sharp “hairpin” loops, this can also raise the free energy. Many different techniques have been proposed for modeling the free energy of a particular folding. Let us be as simple as possible and try to compute a pseudoknot-free folding that maximizes the number of A-U and C-G pairings. Please give an $O(n^3)$ DP algorithm that solves this problem. [Solution]
11.4 DP and Approximation Algorithms

Just like greedy methods, DP also plays an important role in many approximation algorithms. In this section, we will see in particular how DP helps in building a polynomial-time approximation scheme (PTAS) for several prominent problems; recall from Section 1.7.2 that a PTAS delivers a solution whose value differs from that of an optimal solution by only \((1 \pm \varepsilon)\), for any constant \(\varepsilon > 0\).

**Example: Approximating the Knapsack Problem.** In the previous chapter (problem 199) we developed a simple greedy \(1/2\)-approximation algorithm for the knapsack problem that runs in \(\Theta(n)\) time. We now improve on this by developing an \(O\left(\frac{n}{\varepsilon} \log n\right)\) time algorithm that finds a set of items fitting in the knapsack whose value is at least \((1 - \varepsilon)OPT\), where \(OPT\) denotes the value of an optimal solution.

We start by running our greedy \(1/2\)-approximation algorithm to obtain a solution value \(V\) that is a good initial estimate of \(OPT\), since \(OPT/2 \leq V \leq OPT\). We then “discretize” the values of our \(n\) items by rounding each one down to the next-lowest multiple of \(\varepsilon V/n\). Since an optimal solution contains at most \(n\) items, this rounding step decreases the value of an optimal solution by at most \(\varepsilon V \leq \varepsilon OPT\), so by optimally solving the resulting discretized problem, we will find a solution whose value is at least \((1 - \varepsilon)OPT\), our desired approximation guarantee. Note that the guarantee holds even when we then “un-round” our values back to their original settings, since this can only increase the value of our solution.

To solve the discretized problem, we use the DP algorithm from problem 211(c), which optimally solves the knapsack problem with integer values. Thanks to our discretization step, our item values are now effectively integers in a small range, since they are all multiples of \(\varepsilon V/n\) and at most \(OPT \leq 2V\). If we re-scale everything for convenience so that \(\varepsilon V/n = 1\), then our values become integers in the range \(0\ldots 2n/\varepsilon\). This gives \(\Theta(n/\varepsilon)\) subproblems in our DP formulation, each solved in \(\Theta(n)\) time, for a total running time of \(\Theta(n^2/\varepsilon)\). We will shortly improve this to \(O\left(\frac{n}{\varepsilon} \log n\right)\) using additional tricks.

**Problem 219 (A PTAS for Deadline-Constrained Scheduling).** Suppose we are given \(n\) jobs, where each job \(j\) has a processing time \(p_j\), a deadline \(d_j \geq p_j\), and a value \(v_j\). We would like to non-preemptively schedule a maximum-value subset of jobs on a single machine such that deadline constraints are respected. This problem is NP-hard since if all deadlines are equal to a common value \(C\), it is nothing more than the knapsack problem. In problem 163, we showed how a greedy algorithm optimally solves the special case where all values are 1. For the general case, show how to construct a PTAS by modifying our preceding PTAS for the knapsack problem. [Solution]

**Treating Large and Small Elements Separately.** Greedy algorithms are usually effective for “fine-grained” problem instances — involving continuous quantities or many small elements. Conversely, DP algorithms are generally more effective on “coarse-grained” instances involving only large elements. We illustrate both of these phenomena in the following two problems, then show how to combine these two regimes together to build approximation algorithms where small elements are processed greedily and large elements are handled by DP.
Problem 220 (A Greedy PTAS for Problems with Small Elements). For problem instances involving only small elements, greedy algorithms often provide a PTAS.

(a) Suppose all items in a capacity-\( C \) knapsack problem have size at most \( \varepsilon C \). Show that the natural greedy algorithm provides a PTAS in this case. [Solution]

(b) The minimum-makespan scheduling problem is introduced in problem 204. Show how the greedy algorithm from part (a) of that problem is a PTAS if it is known that the processing time of each job is at most \( \varepsilon OPT \), where \( OPT \) denotes the optimal makespan. [Solution]

(c) The bin packing problem is introduced in problem 201. A natural greedy algorithm for this problem is to consider items in any order, and for each one in sequence, to place it into any bin in which it fits, opening a new bin only when necessary. If all item sizes are at most \( \varepsilon \), then this algorithm is not technically a PTAS according to our exact definition above since it does not necessarily use at most \( (1 + \varepsilon)OPT \) bins\(^5\). However, one can show something very close: please prove that the greedy algorithm uses at most \( (1 + 2\varepsilon)OPT + 1 \) bins as long as \( \varepsilon < 1/2 \). [Solution]

Problem 221 (Using DP for Problems with Large Elements). Consider the three problems in problem 220 above: knapsack, minimum-makespan scheduling, and bin packing. For problem instances involving only suitably large elements, we can develop either an optimal solution or PTAS for each problem using DP.

(a) Consider the knapsack problem in which every item has size at least \( \varepsilon C \). Please show how to optimally solve this problem in polynomial time with DP, assuming \( \varepsilon \) is a constant. As a hint, how many items can fit in the knapsack? [Solution]

(b) Consider the minimum-makespan scheduling problem in which each job has a processing time of at least \( \varepsilon OPT \), where \( OPT \) denotes the optimal makespan. Construct a PTAS for this problem using DP. As a hint, round all processing times up to the nearest multiple of \( \varepsilon^2OPT \) and then show that there are only polynomially-many distinct configurations of jobs that can be scheduled on a single machine. [Solution]

(c) Consider the bin packing problem in which there are a only a constant number of distinct item sizes, and show that this problem can be solved in polynomial time using DP. As before, consider how many configurations of items can fit in a single bin. [Solution]

(d) Consider the bin packing problem in which all item sizes are at least \( \varepsilon \). Construct a PTAS for this problem using the following scheme: sort the items by size and partition this sorted list into blocks each of which (except possibly the last) contains \( n\varepsilon^2 \) items. Construct a new problem instance by rounding down the size of each item to that of the smallest item in its block. This instance will have only a constant number of distinct item sizes, so we can solve it optimally using the DP algorithm from the previous part. The only challenging aspect that remains is proving that that this solution will use at most \( (1 + \varepsilon)OPT \) bins, where \( OPT \) denotes the number of bins required in an optimal solution for the original problem instance. As a hint, it may help to make a comparison to yet another problem instance in which we round the items in each block up to the largest item size in the block. [Solution]

\(^5\)You may recall from back in Section 1.7.2 that it is NP-hard to approximate the bin packing problem to within a factor better than \( 3/2 \). While this does rule out the possibility of a PTAS as defined as a \((1 + \varepsilon)\)-approximation, it still leaves open the possibility of achieving an approximation bound similar to that of a PTAS but with a small additive term. This type of approximation result is sometimes known as an asymptotic PTAS, since as our problem instances (and hence also \( OPT \)) grow larger and larger, the additive term becomes less significant and our approximation bound begins to behave more like a purely multiplicative bound.
By carefully merging these approaches, we can build approximation algorithms that use a combination of DP (for large elements) and greedy methods (for small elements) to achieve strong approximation results for general instances. For example, we can obtain a PTAS for all instances of the minimum-makespan scheduling problem \[\text{Complete algorithm}\] and an asymptotic PTAS for all instances of the bin packing problem \[\text{Complete algorithm}\].

**Problem 222 (Approximating the Knapsack Problem, Revisited).** Recall that we recently developed a PTAS for the knapsack problem running in \(O(n^2/\varepsilon)\) time.

(a) Can you show how to improve the running time of our knapsack PTAS to \(O(n \log n)\)? As a hint, treat separately items of large value (say, greater than \(\varepsilon V\), where \(V\) is the value given by our \(\Theta(n)\)-time greedy 1/2-approximation algorithm), and those of small value. [Solution]

(b) Consider the variant of the knapsack problem where we are allowed to place arbitrarily many copies of each item in the knapsack. Why does our former PTAS not work for this variant, and how can we develop an alternate PTAS that does work? [Solution]

### 11.5 Advanced Tricks

We have now seen three main classes of DP algorithms. In Section 11.1 we studied “one-dimensional” problems, where we solve a sequentially solve a single linear chain of subproblems. In Section 11.2 we studied “two-dimensional” problems, where we fill in a table of subproblem solutions one row at a time. Finally, Section 11.3 studies two-dimensional problems with “balanced parenthesis” structure, where subproblem solutions are filled in one diagonal at a time. Although there are several other common dynamic program “templates” (e.g., dynamic programs in trees that solve a subproblem for each subtree during a post-order traversal, as in problem 229), the three above are probably the most common. In this section, we highlight several elegant general tricks for improving the time or space usage of DP algorithms belonging to these three classes.

#### 11.5.1 Reducing Space Usage

Consider the two-dimensional “row-by-row” problems we studied in Section 11.2 (e.g., knapsack, subset sum, edit distance, maximum similarity alignment, longest common subsequence, one-dimensional clustering). For all of these problems, we can save space while filling in our table of subproblem solutions by noting that each row depends only on subproblems in the preceding row. To give an example, consider the knapsack problem with \(n\) items and capacity \(C\), which we can solve in \(\Theta(nC)\) time by filling an \(n \times C\) table of subproblem solutions one row at a time. Since we only need to maintain the current and previous row as we fill the table, we can obtain the value of an optimal solution to this problem using only \(\Theta(C)\) total space. Unfortunately, if we do not retain the entire \(n \times C\) table of subproblem solutions and their associated backpointers, we seemingly lose the ability to recover the structure of this optimal solution, since we no longer have a usable traceback path. Our first advanced trick is a clever divide and conquer method that allows us to recover the
structure of an optimal solution while still only using space proportional to the size of a single row in our table, without harming the asymptotic running time of our DP algorithm. [Full details]

11.5.2 Speedups via Quadrangle/Monge Inequalities

Consider now the “balanced parenthesis” problems we studied in Section 11.3 (e.g., optimal binary search trees, matrix chain multiplication). Let $A$ denote our two-dimensional table of subproblem solutions, so for example with optimal binary search tree construction, $A[i, j]$ is the cost of building an optimal binary search tree on a range of elements $i \ldots j$, by selecting some root $k$ in this range and then recursively building optimal binary search trees on subranges $i \ldots k−1$ and $k+1 \ldots j$.

This problem and many other DP problems satisfy useful monotonicity properties, allowing us to achieve dramatic speedups in running time. For this example, if we determine the optimal root $k$ for range $i \ldots j$, then after increasing either $i$ or $j$, the optimal root for our new subproblem $k'$ tracks in a monotonic fashion: $k' \geq k$.

If we are filling in a diagonal of our DP subproblem table, say by solving ranges $i \ldots i+99, i+1 \ldots i+100, i+2 \ldots i+101$, we would normally need to loop over 100 prospective roots when solving each range. Thanks to monotonicity, however, we need only one linear scan through the $n$ possible roots during this entire process, ultimately saving a factor of $n$ in our final running time.

The structural property often enabling this sort of speedup is the quadrangle inequality (also known as the Monge property), which states that $A[i, j] + A[i', j'] \leq A[i', j] + A[i, j']$ for all $i \leq i'$ and $j \leq j'$. This formula may look slightly imposing, but it is really nothing more than a four-point generalization of the triangle inequality based on the fact that the combined length of the two diagonals in any quadrilateral can be no shorter than the combined length of two opposite sides$^6$.

An alternative, slightly simpler characterization for quadrangle/Monge matrices is that they must satisfy $A[i, j] + A[i+1, j+1] \leq A[i+1, j] + A[i, j+1]$ for all $(i,j)$. As it turns out, the subproblem tables for many “balanced parenthesis” dynamic programs satisfy the quadrangle/Monge property; this is true for optimal binary search trees, matrix chain multiplication, and optimal polygon triangulation (problem 224). As shown by Knuth and Yao, this property enables speedup by a factor of $n$ using monotonicity as described above, improving running time from $O(n^3)$ to $O(n^2)$ for all three of these example problems. [Full details]

11.5.3 Speedups via Monotone Matrix Searching

Here, we discuss another class of DP-solvable problems we can speed up by exploiting monotonicity. In Section 11.1, we showed how many problems are equivalent to acyclic shortest path computation, more specifically the calculation of a shortest path from node 1 to node $n$ along a topological ordering of nodes numbered $1 \ldots n$. We can abstractly specify any such problem with a matrix $C$ whose $(i,j)$ entry gives the cost of an edge from node $i$ to node $j$. Since our acyclic graph only contains edges $(i,j)$ with $i < j$, we set $C[i,j] = +\infty$ for $i \geq j$. Let $M(i)$ denote

$^6$For more geometric intuition on the quadrangle inequality, please consult problem ?? at the end of Chapter ??.
the index of the minimum entry in row $i$ of $C$ (in the event of a tie, we take the rightmost minimum entry). We say $C$ is monotone if $M(1) \leq M(2) \leq \ldots \leq M(n)$, and we say $C$ is totally monotone if every submatrix of $C$ is monotone (actually, it suffices simply to prove that every $2 \times 2$ submatrix of $C$ is monotone). It is also fairly easy to prove that $C$ is totally monotone if it satisfies the quadrangle/Monge properties above. The problem of computing $M(1) \ldots M(n)$ in a totally monotone matrices arises in several situations (e.g., see problem ??). As a result, substantial research was devoted to this problem in the 1980s, culminating in a simple and elegant divide-and-conquer algorithm known as the SMAWK algorithm (named for its authors: Shor, Moran, Aggarwal, Wilber, and Klawe), which is remarkably capable of computing $M(1) \ldots M(n)$ in only $\Theta(n)$ time, despite the $n \times n$ size of our matrix. [Details of the SMAWK algorithm]

Many one-dimensional DP problems (for example, problem 210(g) on optimal type-setting) have a cost matrix $C$ that is totally monotone. For such problems, calculation of $M(1) \ldots M(n)$ in linear time can be used to solve a batch of $n$ subproblems in just $\Theta(n)$ time. Still more DP problems can be decomposed into an “interleaved” set of one-dimensional problems, each with totally monotone cost matrices [Example: sequence alignment problems with gaps]. Our final trick is a somewhat sophisticated algorithm, sometimes called the LARSH algorithm (named after its authors, Larmore and Schieber), that makes clever use of the SMAWK algorithm to deliver a factor of $n$ speedup for these problems [Full details]. We can also leverage this approach to save a factor of $n$ in the running time of certain two-dimensional row-by-row problems (e.g., one-dimensional clustering) for which the computation of each row of subproblem solutions is equivalent to a one-dimensional problem with a totally monotone cost matrix. [Details]

11.6 Additional Problems

Problem 223 (Block Stacking). You are given a set of $n$ unit-height blocks with widths $w_1 \ldots w_n$. Your goal is to stack these blocks to the maximum possible height. As shown in Figure 11.11(a), the blocks must be stacked in order from block 1 to block $n$, and for stability each level of the stack must be no wider than the level upon which it rests. Show how to solve this problem efficiently using dynamic programming. [Solution]
Problem 224 (Optimal Convex Polygon Triangulation). A convex polygon $P$ is specified by giving the coordinates $(x_1, y_1) \ldots (x_n, y_n)$ of its $n$ vertices in clockwise order. As shown in Figure 11.11(b), a triangulation of $P$ is a collection of line segments that partitions $P$ into a set of triangles, where each segment runs between two of the vertices of $P$. We wish to compute a triangulation of $P$ that minimizes the combined lengths of these added segments. Show how to compute such an optimal triangulation in $O(n^3)$ time. [Solution]

Problem 225 (The Transportation Problem with Only Two Sources). Suppose we have $n$ factories that produce goods to be shipped to a collection of $m$ warehouses. The $i$th factory supplies $s_i$ (an integer) units of goods, and the $j$th warehouse has a demand for $d_j$ units of goods. For each pair $(i, j)$, we have a transportation constraint that says we can ship at most $u_{ij}$ units of goods from factory $i$ to warehouse $j$, and each unit of goods shipped from $i$ to $j$ costs $c_{ij}$ units. We would like to determine how many units of goods to ship from each factory to each warehouse so that (i) all demand is satisfied — $d_j$ units arrive at each warehouse $j$, (ii) at most $s_i$ units are shipped out of each factory $i$, (iii) at most $u_{ij}$ units are shipped from factory $i$ to warehouse $j$. Subject to these constraints, we would like to incur the least amount of cost possible. In Chapters ?? and ??, we will see how to solve this problem (called the transportation problem) in polynomial time using network flow techniques. Here, we consider only the special case with $n = 2$ factories, since this is a nice DP problem (recall also that there is a simple greedy solution for the restricted case with no upper capacities $u_{ij}$ and exact supply/demand constraints, as we saw in problem 197). Please show how to solve this problem in $O(mS^2)$ time, where $S = s_1 + s_2$ is the total amount of supply at both factories. [Solution]

Problem 226 (Longest Arithmetic Progression). Given a set $S$ of $n$ numbers, we would like to find the longest arithmetic progression (either increasing or decreasing) that is a subset of $S$. For example, in the set $\{4, 2, 5, 7, 3, 12, -8, 17, 20, 21\}$, the longest arithmetic progression is the sequence $2, 7, 12, 17$. See if you can devise an $O(n^3)$, or better still, $O(n^2)$, DP algorithm for this problem. [Solution]

Problem 227 (Shortest Bitonic Tour). This is a “classic” DP problem. Suppose you are given a list of $n$ cities ordered from west to east, and you are told an $n \times n$ matrix of distances between every pair of cities. You would like to construct a path of minimum total length that starts at the eastmost city, travels west until it reaches the westmost city, then travels east again until it returns to the eastmost city. In the process, the path must visit every city exactly once except the eastmost city, which it visits twice. Please solve this problem in $O(n^2)$ time using DP. Later, in problem ??, we consider the more general problem in which you can make $k$ passes back and forth rather than just 1. [Solution]

Problem 228 (Disk Scheduling). Suppose we need to read $n$ blocks of data that are spread out across a disk. For the purposes of this problem, let us model a disk as having a single read/write head that moves back and forth across a linear array of blocks. For simplicity, we adopt the slightly unrealistic assumption that the amount of time required to move the read/write head from block $i$ to block $j$ is proportional to $|i - j|$ (the distance between the two blocks). Once the read/write head reaches one of our $n$ input blocks, it reads it instantly, so the only time-consuming operation is the movement of the head. Given the locations of our $n$ blocks as well as the starting location of the read/write head, devise an $O(n^2)$ dynamic programming algorithm that minimizes the total time required to read all the blocks. [Solution]

Problem 229 (DP Algorithms on Trees). It is often the case that special cases of NP-hard problems can be solved in polynomial time. In this problem we investigate special cases of NP-hard graph problems that are easily solved on trees via DP. See Section ?? for a more complete discussion of these problems.

(a) A subset of nodes in a graph is an independent set if there is no edge connecting any
pair of these nodes, and a node cover (also commonly called a vertex cover) if every edge has at least one endpoint in the set. If we associate values with the nodes of a graph, the maximum-value independent set problem asks for an independent set of maximum total value. If nodes have associated costs, we can similarly ask for a minimum-cost node cover. Both problems are NP-hard in general graphs. Please show how to solve them both in $\Theta(n)$ time on an $n$-node tree using DP. [Solution]

(b) A subset of nodes in a graph is a dominating set if it covers all the nodes in the graph. That is, every node not in the set must at least have one of its neighbors in the set. If we associate costs with the nodes in a graph, we can consider the minimum-cost dominating set problem, which is NP-hard on general graphs. Please show how to solve it in $\Theta(n)$ time on an $n$-node tree using DP. [Solution]

(c) A distance-$k$ dominating set is a subset of the nodes of a graph such that every node not in the set has at least one node in the set within distance $k$ of it. Stated differently, every node we select for our set has the ability to cover, or “dominate”, every node within distance $k$, and we wish to cover every node in the entire graph. For a challenge, please give an $O(kn)$ DP algorithm for the minimum-cost distance-$k$ dominating set problem in a tree. [Solution]

(d) As a challenge, see if you can design a $\Theta(n)$ “greedy” algorithm for the minimum-cardinality distance-$k$ dominating set problem in an $n$-node tree. [Solution]

(e) Suppose we associate a numeric “weight” with every node in a tree, some of which may potentially be negative. For a specified value of $k$, we would like to find a subtree induced by a selection of exactly $k$ leaves (that is, a tree consisting of just those leaves plus any internal nodes that are necessary to connect them all together) of maximum total weight. Please show how to solve it in $O(nk^2)$ time. For a challenge, can you refine your analysis of the same algorithm to show that it runs in $O(n^2)$ time, even if you want the answer for every value of $k$? [Solution]

Problem 230 (Asymmetric Binary Search). Suppose we would like to find the value of some unknown quantity $X$ whose value is known to live in the range $1 \ldots n$. Our only means of learning about $X$’s value is to make repeated guesses, where we can tell if each successive guess is too high or too low. For each guess that is too high, we incur a penalty of $P_{\text{high}}$, and for each guess that is too low, we incur a penalty of $P_{\text{low}}$. Devise an $O(n^2)$ DP algorithm that determines a guessing strategy that minimizes the maximum possible penalty we might incur. If $P_{\text{high}}$ and $P_{\text{low}}$ are integers of magnitude at most $B$, please show an alternate formulation achieving a running time of $O(B \log n)$. [Solution]

Problem 231 (Scheduling with Release Times and Deadlines). Please use DP to give an optimal offline algorithm for problem 166 (i.e., where we know the release times of all jobs in advance). [Solution]
12. Integer and Linear Programming

Linear programs (LPs) are optimization problems involving the minimization or maximization of a linear objective function in the presence of linear equality and inequality constraints. Due to their computational tractability, rich theoretical structure, and ubiquity in practice, LPs have become a key part of the foundation of modern optimization theory.

For example, the LP

Minimize: \[ 4x_1 + 6x_2 \]
Subject to: \[
5x_1 + 6x_2 \geq 30 \\
2x_1 + 3x_2 \geq 9 \\
x_1 \geq 0 \\
x_2 \geq 0
\]

attains its optimal objective value of 26 at \((x_1, x_2) = (4, 5/3)\). An integer linear program (ILP or IP) is a linear program whose variables are additionally constrained to have integer values (often just the values zero or one). In the example above, if \(x_1\) and \(x_2\) are constrained to be integers, then the optimal objective value changes to 28, attained at \((x_1, x_2) = (4, 2)\). We call a problem a mixed integer linear program (MILP or MIP) if only some of its variables have integrality constraints.

Most optimization problems can expressed, or formulated, in the generic format above, known as a mathematical program\(^1\), where we want to find an optimal setting for a vector of \(n\) decision variables \(x = (x_1, \ldots, x_n)\) that is feasible with respect to a set of constraints. The set of feasible solutions often has a meaningful geometric description, giving us helpful geometric insight into many problems. For example, the feasible region of an LP is always a convex polyhedron, as shown in Figure 12.1 for our example problem above.

Mathematical programs come in many shapes and sizes. LPs are perhaps the simplest, having a linear objective and linear constraints. Non-linear programming problems involve either non-linear objectives or constraints; common special cases

---

\(^1\)As we mentioned in the preceding chapter, the term “programming” here means something different from computer programming — “optimization” or “planning” are perhaps the closest synonyms to the original historical meaning of the term in this context.
involve quadratic programming and convex programming, where our objective is respectively a quadratic function or a convex function. At a high level, we often divide the world of all optimization problems into two main categories: discrete problems (e.g., integer LPs), where we optimize over a discrete set of potential solutions, and continuous optimization problems (e.g., minimizing $f(x) = 2x^3 + x^2 - 4x + 3$ over all $x \in [0, 1]$), where we optimize over a continuous space of solutions. LPs lie at the boundary of these two realms, since they optimize over a continuous set of feasible solutions, but as we shall see in a moment, we need only consider a discrete set of special so-called basic solutions when solving an LP.

In terms of complexity, LPs are one of the broadest classes of optimization problems that we can solve in polynomial time. Any problem you can formulate as an LP, you can therefore solve in polynomial time, and this includes a tremendous number of prominent problems throughout this book — shortest paths, minimum spanning trees, network flows, matchings, and many others. Note that with many of these highly-structured problems, linear programming itself is usually not the preferred method of solution, since we will be able to solve them much faster using specialized combinatorial algorithms that fully exploit their underlying structural properties, rather than by solving them as “generic” LPs. However, even for problems better suited for combinatorial algorithms, insights from interpreting them in a linear programming context can often still help us understand a great deal about their underlying structure, and how they relate to other similar problems. Many well-known algorithms like label correcting shortest path algorithms, augmenting path maximum flow algorithms, and the network simplex algorithm for minimum cost circulation, are essentially just direct specializations of the more general simplex algorithm for linear programming. Many other algorithms spend their early childhood in a setting based entirely on linear programming, but then mature into stand-alone techniques that make sense in their own right; we will see examples later when we study so-called primal-dual algorithms.

Adding integrality constraints to an LP introduces substantial complexity, as integer LPs are in general NP-hard to solve. They are very much worth solving, however,
due to their prominence in practice — indeed, nearly every discrete optimization problem in this book can be formulated as an integer LP. Since techniques for approaching integer LPs often use tools from linear programming, we will study several of these in this chapter, and also several in the next chapter when we embark on a detailed discussion of heuristics for solving hard problems.

12.1 Linear Programming

Let \( x = (x_1, \ldots, x_n) \) be a vector of the \( n \) decision variables we wish to set in order to maximize the linear objective function

\[
c^\top x = c_1 x_1 + c_2 x_2 + \ldots + c_n x_n,
\]

where \( c = (c_1, \ldots, c_n) \) is the vector of coefficients in our objective function. We say \( x \) is a feasible solution\(^2\) if it satisfies a system of \( m \) linear constraints:

\[
\begin{align*}
a_{11} x_1 + a_{12} x_2 + \ldots + a_{1n} x_n &\leq b_1 \\
a_{21} x_1 + a_{22} x_2 + \ldots + a_{2n} x_n &\leq b_2 \\
& \vdots \\
a_{m1} x_1 + a_{m2} x_2 + \ldots + a_{mn} x_n &\leq b_m.
\end{align*}
\]

These could be a mixture of equality or inequality constraints; let us assume for now they are all inequality (\( \leq \)) constraints. Letting \( A \) be the \( m \times n \) matrix whose entries are the \( a_{ij} \)'s, we can write our entire LP concisely using matrix-vector notation as

Maximize: \( c^\top x \)

Subject to: \( Ax \leq b. \)

With a small amount of algebra, every LP can be written in this form, known as its canonical form. To do so, we replace each equality constraint with a pair of \( \leq \) and \( \geq \) inequality constraints, and we negate \( \geq \) inequality constraints to obtain \( \leq \) constraints. If our objective originally involved minimization, we negate \( c \) to obtain an equivalent maximization objective.

Every LP can also be re-written in what is called standard form:

Minimize: \( c^\top x \)

Subject to: \( Ax = b \)
\[ x \geq 0, \]

a convenient form for certain algorithms (e.g., the simplex algorithm), since it involves only equality constraints and nonnegative variables. Converting between canonical and standard form is straightforward. [Details]

A solution is optimal if it achieves the best objective value among all feasible solutions. We often speak loosely of “the optimal solution” for a problem, even though

\(^2\)It is worth mentioning again that the word “solution” in this context does not necessarily imply an optimal, or even feasible answer to a problem; it simply means a setting of our vector \( x \) of decision variables.
Figure 12.2: Illustrations of (a) finding the center of a point set under the $L_1$ distance metric, with the dotted line representing a “ball” of constant radius using $L_1$ distance; (b) allocating work to factories to minimize the maximum pollution encountered across all locations (for example, operation of all three factories above at high rates might cause too much pollution at location X); (c) finding a separating hyperplane (if it exists) between two classes of points; (d) fitting a linear function to a set of points to minimize either the sum or maximum of the errors (dotted lines) across all points.

for many problems there is not necessarily a unique optimal solution $x$. Some problems may not even have an optimal solution. For example, the problem “Maximize $x_1$ subject to $x_1 \geq 0$” has no optimal solution because it is unbounded, and the problem “Minimize $x_1$ subject to $x_1 \geq 2$ and $x_1 \leq 1$” has no feasible solution to begin with. An LP that is neither unbounded nor infeasible will have a well-defined finite optimum solution value.

12.1.1 Formulating Problems as Linear Programs

Now that we know what LPs look like, it is useful to see some concrete examples of prototypical problems solvable with linear programming. The examples below will give you practice formulating LPs, including common “tricks” used to transform certain nonstandard-looking constraints into linear constraints. For example, to minimize $\max\{x_1, x_2, \ldots, x_n\}$, we can instead minimize $y$ such that $y \geq x_i$ for all $i = 1 \ldots n$. Since $|x| = \max\{x, -x\}$, the same trick allows us to handle problems involving absolute values. This can be useful, for example, with an objective like the $L_1$ norm $||x||_1 = |x_1| + \ldots + |x_n|$, which you may recall from Section 7.5.8 is commonly used as a heuristic when we want to find “sparse” solutions (with many of the $x_i$’s set to zero).
Problem 232 (Center of a Point Set Under the $L_1$ Distance Metric). Given $n$ points $x_1 \ldots x_n$ in $d$-dimensional space, their center is a point $p$ minimizing the maximum distance from $p$ to one of the $x_i$’s — see Figure 12.2(a). If we measure distance using the $L_1$ metric, show how to find a center using linear programming. [Solution]

Problem 233 (Balancing Negative Side Effects of Resource Allocation). Suppose you control $n$ factories. Factory $i$ runs at a rate of $r_i$ (a number you are in charge of selecting), where $r_i$ can be at most some maximum capacity $c_i$. Running a factory at a higher rate produces more pollution. Given a discretization of space into $m$ cells, as in Figure 12.2(b), suppose that cell $j$ receives $A_{ij}r_i$ pollution from the operation of factory $i$ at rate $r_i$; presumably, $A_{ij}$ is large if factory $i$ is spatially close to cell $j$. Please show how to allocate a total of $R$ units of work among the factories (so their rates must sum to at least $R$) so as to minimize the maximum amount of pollution in any one cell. [Solution]

Problem 234 (Perceptrons and Separating Hyperplanes). As shown in Figure 12.2(c), two sets of points are said to be linearly separable if we can find a hyperplane that perfectly separates the two sets from each-other. Finding such a separating hyperplane is a fundamental problem in machine learning, where we want to develop a simple predictive model that can be trained to distinguish one class of data from another, where each element of data is identified by a “feature vector” in $d$-dimensional space (see also Figure 8.14(a) for another nice example of such a problem). After learning a separating hyperplane, we can use this to predict the classification of new, unlabeled data elements, depending on which side of the hyperplane they fall. Known as a perceptron, this single-hyperplane classifier is one of the simplest methods for geometric classification; we discuss more sophisticated variants later in Chapter ???. Please show that if we are given a linearly-separable collection of points (each labeled as one of two classes), then we can find a separating hyperplane using linear programming. [Solution]

Problem 235 (Min-Sum and Min-Max Linear Regression). Suppose we want to optimize the parameters of a linear model (a line in 2D, a plane in 3D, or a hyperplane in higher dimensions) so it gives the best fit for a set of data. As shown in Figure 12.2(d), the 2D case involves choosing parameters $m$ and $b$ so as to fit the line $y = mx + b$ to a number of data points $(x_1, y_1), \ldots, (x_n, y_n)$. For any line we choose, we get an error of $|y_i - (mx_i + b)|$ (shown with a dotted segment) in predicting the $y$-coordinate of the $i$th point. Please show how to use an LP to minimize (1) the sums of all errors, and (2) the largest error. Please show how your answers generalize to higher-dimensional linear fitting problems as well. Another very common objective, that of minimizing the sum of squared errors, will be considered later in Chapter ?? when we study “least squares” problems. [Solution]

Problem 236 (Discounted-Reward Markov Decision Processes). A Markov process moves around a graph or state space randomly, where at each discrete time step, we move from our current node/state to a randomly-selected neighboring node/state (we study random walks like this extensively later in Section ??). A Markov decision process (MDP) is a popular, powerful model in the field of stochastic optimization that generalizes this idea by adding in the ability to partially control the walk. An MDP is specified by $n$ states with associated values $v_1 \ldots v_n$, along with $m$ “actions”. Along with each (state, action) pair $(s, a)$ is an associated probability distribution over all states, specifying where we end up if we execute action $a$ while in state $s$ (for example, we might choose an action that tries to bring us to a particular state, but there could be some slight chance of “failure” that brings us to an unwanted state). An MDP therefore executes as follows: we begin in a designated start state, choose an action, probabilistically move to a new state, choose another action (based on observing our current state), and so on. Each time we enter state $s$, we obtain value $v_s$, and we would like to obtain as much total value as possible by staying in valuable states. In order to make our optimization objective finite, we discount future rewards by some factor $\alpha < 1$, so we only obtain value $v_s\alpha^t$ by visiting state $s$.
at time $t$ (this is a common option in multi-stage optimization, since near-term rewards typically have larger value due to the inherent uncertainty involved with predicting longer-term rewards). We would like to determine an optimal policy $P$ (maximizing our expected total discounted value obtained over all time) specifying the action $P_s$ that should always be performed if we find ourselves in state $s$. Please show how to compute an optimal policy using linear programming. [Solution]

12.1.2 The Geometry of Linear Programs

LPs have a useful geometric interpretation. For an LP $\max \{ c^\top x : Ax \leq b \}$ in canonical form, each inequality constraint defines a half-space — the set of all points in $n$ dimensions on one side of a hyperplane (a line in 2D, or a plane in 3D). The intersection of the $m$ half-space constraints in our LP collectively define a convex polyhedron of feasible solutions, as shown in Figure 12.3(a). This polyhedron may extend to infinity in some directions (in which case our problem might end up being unbounded), or it may be empty if our problem is infeasible. The objective “Maximize $c^\top x$” asks us to find a feasible solution as far in the direction of the objective vector $c$ as possible, since the dot product $c^\top x$ tells us the extent to which $x$ lies in the direction indicated by $c$ (see also Section ?? for additional remarks on dot products and projections). A nice physical analogy is to treat $c$ as the direction of gravity, pulling our optimal solution through the feasible region until it comes
to rest supported by a subset of constraints. Convexity dictates that an optimal solution lies at a corner point of the feasible polyhedron, assuming the problem is feasible and not unbounded.

LPs in standard form also have a convenient geometric interpretation. Here, we use the fact that the matrix-vector product \( Ax \) tells us the linear combination 
\[
a_1x_1 + a_2x_2 + \ldots + a_nx_n
\]
of the columns \( a_1, \ldots, a_n \) of \( A \). Our constraints \( Ax = b, x \geq 0 \) therefore ask us to express \( b \) as a nonnegative-weighted combination of \( A \)'s columns. Since the objective is to minimize 
\[
c_1x_1 + c_2x_2 + \ldots + c_nx_n
\]
optimize over a continuous space, we often regard them as discrete problems because we can effectively restrict our attention to the discrete set of all basic solutions. For example, we will soon discuss how the famous simplex algorithm solves an LP by stepping through a succession of “adjacent” feasible basic solutions (sharing \( n - 1 \) constraints in common) until reaching an optimal solution.

We will study LP duality in a moment, showing how LPs in canonical and standard form are “dual” to each other, with constraints (rows of \( A \)) in one problem being related to variables (columns of \( A \)) in the other. Accordingly, while a basic solution for a canonical-form LP is defined for every set of \( n \) linearly-independent constraints, a basic solution for a standard-form LP is defined for every set of \( m \) linearly-independent columns of \( A \). There is a unique way to set the decision variables corresponding to these columns so they sum to \( b \) (this is given by the solution of the \( m \times m \) linear system \( A'x = b \), where \( A' \) contains just the \( m \) columns in question). Setting the remaining \( n - m \) “nonbasic” variables to zero, this gives our basic solution \( x \), satisfying \( Ax = b \). This solution is feasible if \( x \geq 0 \) — that is, if it forms \( b \) with a nonnegative combination of columns from \( A \).

Since the feasible region of an LP is convex, any weighted average \( \lambda x + (1 - \lambda)x' \), \( \lambda \in [0, 1] \), of two feasible solutions \( x \) and \( x' \) (known as a convex combination) must also be a feasible solution, since it lies somewhere on the segment joining \( x \) and \( x' \).
Feasible basic solutions are special in that they are the only feasible solutions that cannot be expressed as convex combinations of two or more other solutions. This fact can be helpful in proving a particular solution is a basic solution, and also in establishing special properties of basic solutions for certain problems. For example, when we study network flow problems later in the book, we will see that their basic solutions have an underlying “spanning tree” structure, which can help us design effective algorithms like the network simplex algorithm (Section ??).

12.2 Solving Linear Programs

In this section, we give a brief overview of the most fundamental approaches for solving LPs. Note that in this day and age, we rarely need to write an LP solver from scratch. Just as with many other standard numerical problems (e.g., solving linear systems, eigenvalue/eigenvector computation), one can find a vast array of software packages (many freely available) that use far more sophisticated and highly-tuned methods than we have space to describe here. Nonetheless, it is still important to know what is happening “underneath the hood”, in order to understand why certain LPs might be easier to solve than others in practice.

One of the most pressing open algorithmic questions in linear programming today is whether or not there exists a strongly polynomial, combinatorial algorithm for linear programming, with a running time depending only on $m$ and $n$. Of the algorithms we now discuss, the simplex algorithm is a combinatorial algorithm whose worst-case running time depends only on $m$ and $n$, but exponentially so, while the ellipsoid and interior point methods have only weakly-polynomial running times, depending on the number of bits required to describe our input.

12.2.1 The Simplex Algorithm

Perhaps the best-known LP algorithm, the simplex algorithm is based on iterative refinement, moving from one feasible basic solution to another until it reaches optimality. The algorithm usually works with a standard-form problem $\min\{c^\top x : Ax = b, x \geq 0\}$, where a feasible basic solution consists of $m$ nonnegative “basic” variables, with the remaining $n - m$ “nonbasic” variables set to zero (recall that the basic variables correspond to columns of $A$, and give the weights required to combine those columns to obtain $b$). In each iteration, the simplex algorithm tries to decrease its objective cost $c^\top x$ by moving from its current feasible basic solution $x$ to an “adjacent” feasible basic solution (sharing $m - 1$ basic variables in common, with one formerly nonbasic variable becoming basic). If no such move is possible, we will have reached a locally-optimal solution. As we will see later in Chapter ??, due to convexity of our objective and the feasible region, this must also be a globally-optimal solution for the entire LP.

To perform each move, we compute the reduced cost of each nonbasic variable, telling us the rate at which our objective $c^\top x$ would change if we were to increase the value of that nonbasic variable while adjusting the other basic variables accordingly to compensate, so we maintain $Ax = b$. Our solution can be improved (decreasing its total cost $c^\top x$) if we find any nonbasic variable with negative reduced cost, by
increasing its value (thereby adding it to our basis) and adjusting the other basic variables to maintain \( Ax = b \) until one drops to zero (thereby leaving the basis). This operation is known as a pivot. [Further details on reduced costs and pivoting]

A feasible basic solution to an LP is optimal if and only if all nonbasic variables have nonnegative reduced costs. This is one way to write a simple set of optimality conditions for characterizing when a feasible solution is optimal. Optimality conditions are generally important to derive for any problem you are considering. They not only help us understand the problem’s underlying structure, but they usually lead directly to a means of solution by iterative refinement: as long as the optimality conditions fail to hold, modify our current solution to push it more towards satisfaction of these conditions (via a pivot, in the case of the simplex algorithm).

Several interesting situations can occur while the simplex algorithm pivots its way towards an optimal basis. It can encounter a pivot operation that would like to increases a nonbasic variable infinitely far, in which case we will realize our problem has an unbounded solution. It can also introduce a nonbasic variable whose value cannot be increased at all (since some basic variable is already set to zero, and therefore cannot be decreased further to preserve \( Ax = b \)), resulting in a pivot that merely expresses the same solution \( x \) with a different set of basic variables. This situation, known as degeneracy, is one of the most troublesome issues for the simplex algorithm, since it can cause us to cycle endlessly among different basic solutions representing the same point \( x \) unless we take care to prevent this behavior [Further details]. One must also consider how to find a feasible basic solution at which to start. A common solution to this problem is to introduce “dummy” variables into our problem that can be used as our initial basic variables; by assigning these large coefficients in the objective function, they will all be eventually pivoted out before we reach an optimal basis. [Further details]

In terms of running time, there are contrived instances that can produce an exponential number of pivots in the worst case, so the simplex algorithm sadly does not run in polynomial time. However, this is one of many algorithms whose worst-case theoretical performance rarely occurs in practice, so unless we have a problem involving a substantial amount of degeneracy, the simplex algorithm typically runs quite fast. Much research has been devoted to investigating whether a polynomial variant of the simplex algorithm can be constructed, say, by choosing nonbasic variables for pivoting in a more intelligent fashion. Perhaps the most impressive result along these lines is a highly non-trivial recent result of Spielman and Teng showing that the simplex algorithm has polynomial smoothed running time — that is, polynomial expected running time after its input parameters are subject to a small random perturbation. Since real-world problem data often has some small amount of random “noise” baked in, this gives a nice justification for why the simplex algorithm often runs quickly in practice.

**Column Generation.** Since all we need to know to run each iteration of the simplex algorithm is the identity of a nonbasic variable with negative reduced cost, we can actually run the algorithm implicitly on a huge problem with an exponential number of variables, as long as we have some algorithm that identifies a nonbasic variable of negative reduced cost, or certifies that all nonbasic variables have nonnegative cost. Often, we can perform this task in polynomial time using a simple combinatorial algorithm [Example: finding a maximum flow of minimum cost]. This
trick is known as *column generation*, since it starts with a minimal representation of $A$ containing columns corresponding only to basic variables, and adds new columns "on the fly" as they are needed, whenever we pivot in a new nonbasic variable. Working with this reduced $A$ matrix containing only the columns generated so far is far less expensive than working with the full $A$ matrix and its exponentially many columns. When we study duality, we will see that the same approach applied to the dual of a problem (using the *dual simplex algorithm*) will allow us to solve problems with massive numbers of constraints in an effective manner using *row generation*.

### 12.2.2 The Ellipsoid Algorithm

The *ellipsoid algorithm*, developed in the 1970s, was the first approach to achieve a polynomial running time for solving an LP. It actually solves the seemingly-easier problem of just finding any feasible solution, but solving this in polynomial time is enough to get a weakly-polynomial algorithm for finding an optimal solution. Starting with the LP $\max \{c^\top x : Ax \leq b\}$ in canonical form, if we add a constraint $c^\top x \geq z$ and then find a feasible solution, we know the optimal solution value of the original problem is at least $z$. By repeating this process via binary search on $z$ for polynomially-many steps, we eventually home in on an optimal solution.

At a high level, the ellipsoid algorithm is fairly straightforward. It computes a series of progressively smaller *ellipsoids* (elongated spheres) that each contain the feasible region of an LP, terminating upon reaching an ellipsoid whose center is inside the feasible region. The key subroutine at each step is checking whether the current ellipsoid center is in the feasible region, or whether it violates some constraint. If it is feasible, we are done. If we identify a violated constraint, this constraint can be used to help us shrink the ellipsoid so it still contains the entire feasible region [*Mathematical details*]. In fact, the ellipsoid algorithm can solve LPs in polynomial time even if they have exponentially many constraints, as long as we can provide an algorithm, known as a *feasibility oracle*, that can check in polynomial time if a given point is feasible, identifying a violated constraint if not. [*Example: shortest path network interdiction*]

### 12.2.3 Interior Point Methods

Although the ellipsoid algorithm runs in polynomial time, it is usually much slower than the simplex algorithm in practice, so we tend to use it mainly for theoretical results — say, to argue that an implicit LP with exponentially many constraints can be solved in polynomial time. In the 1980s, researchers developed another class of methods, known as *interior point* methods, which have both polynomial running times and also highly practical performance. Interior point methods can outperform the simplex algorithm, particularly for large problem instances.

True to their name, interior point methods approach an optimal solution via a series of steps through the interior of the feasible region, in contrast to the simplex algorithm, which approaches the optimal solution by walking along the boundary of the feasible region, and the ellipsoid algorithm, which approaches from outside the feasible region. Most variants of interior point methods are based on nonlinear optimization techniques (Chapter ??) applied to a modified problem where explicit
constraints are replaced by penalty terms in the objective to ensure that movement in the direction of the objective vector does not leave the feasible region. Due to the number of different variants of these methods and the amount of mathematical detail needed for their analysis, we omit further discussion at this point. Please consult the endnotes for additional references.

12.3 Integrality

LPs with extra integrality constraints are quite common in practice, although sadly NP-hard to solve in general. In this section and the next (and also in parts of the next chapter), we discuss how to use LP-based tools in various ways to address these problems, some of which can be quite challenging to solve.

For starters, we could be lucky. It turns out that LP formulations of many common (and usually “easier”) problems are sufficiently well-behaved that they satisfy the integrality property, with every basic solution having integer-valued decision variables $x_1 \ldots x_n$ (the objective value $c^T x$ could still be non-integral if the coefficients in $c$ are non-integral). As long as we use a standard LP solver based on a method like the simplex algorithm that hands us a basic solution, we can therefore obtain an integer-valued solution to any such LP with no extra effort on our part\(^3\). Several prominent optimization problems have LP formulations that satisfy the integrality property, including shortest paths, flow problems, matchings, minimum spanning trees, and many more. All of these problems can therefore be solved as generic LPs. However, as with all of these examples, the fact that a problem has an integral LP formulation usually means that it can also be solved in strongly polynomial time via faster combinatorial techniques. Even in such cases, however, looking at a problem as an LP can often give us helpful insight into its structure.

Example: Minimum-Cost Interval Cover. To make our discussion concrete, consider an example problem taking as input a set $I$ of $n$ intervals, each being a subinterval of $[0, 1]$. Interval $i \in I$ has associated cost $c_i > 0$, and our goal is to compute a minimum-cost subset of intervals whose union completely covers $[0, 1]$. An application of this problem might be scheduling employees on shifts. Each prospective shift has an associated cost, and we want to choose a minimum-cost set of shifts so that at least one employee is present at all times throughout the day. This problem is quite well-structured from an LP perspective, satisfying integrality and a number of other useful properties. Not surprisingly, therefore, it can also be addressed by several other algorithmic techniques. We solved it in $O(n \log n)$ time in problem 209 using dynamic programming, whereas the simpler variant in which all intervals have unit cost can be solved in $O(n \log n)$ time with a greedy algorithm (problem 180). Yet more variants are studied in problems 193 and ??.

The endpoints of the intervals in $I$ partition $[0, 1]$ into $O(n)$ timeslots. Think of a timeslot as a maximal-length subinterval along which the “cross section” of intervals

\(^3\)If several basic solutions tie for optimality, then any convex combination of them will also be optimal, so optimal nonbasic fractional solutions might also exist. Algorithms like interior point methods, which don’t deal with basic solutions, might end up finding a solution of this type, so if we were to use one of these algorithms, we might need to apply a “basis identification” post-processing step (many of which have been proposed in the research literature, and which run in polynomial time) to move from an optimal nonbasic solution to an optimal basic solution.
in $I$ does not change, as shown in Figure 12.4(a). If we let $T$ denote the set of all timeslots, then we write $i \sim t$ if interval $i \in I$ is active during timeslot $t \in T$. Let us associate a decision variable $x_i$ with every interval $i \in I$, with the understanding that $x_i = 1$ if interval $i$ belongs in an optimal solution, and $x_i = 0$ otherwise. We can now write the minimum-cost interval cover problem as an LP with added integrality constraints as follows:

Minimize: $\sum_{i \in I} c_i x_i$

Subject to: $\sum_{i \sim t} x_i \geq 1$ for all $t \in T$

$x_i \in \{0, 1\}$ for all $i \in I$.

The objective $\sum_i c_i x_i$ reflects the cost of the set of intervals chosen for our solution, and we have a “covering” constraint for each timeslot $t \in T$ stipulating that we must select at least one of the intervals spanning $t$ in our solution. By dropping the integer constraints, we can “relax” our formulation to a standard LP:

Minimize: $\sum_{i \in I} c_i x_i$

Subject to: $\sum_{i \sim t} x_i \geq 1$ for all $t \in T$

$x_i \geq 0$ for all $i \in I$.

The main difference is that each $x_i$ can now take fractional values. A more direct relaxation of $x_i \in \{0, 1\}$ would have given $0 \leq x_i \leq 1$, but as is typical, we omit the $x_i \leq 1$ part since an optimal solution will clearly never involve setting any $x_i > 1$.

The LP above is integral, always having an optimal solution with $x_i \in \{0, 1\}$ for all $i = 1 \ldots n$. We can therefore “get away with” using the simpler LP relaxation of the problem instead of its original more complicated integer LP formulation.

There are several common ways to prove integrality results like this one. For example, we can demonstrate some algorithm that always constructs an integer-valued optimal LP solution. In Section 12.5.3, we will develop a “primal-dual” algorithm for generating such an optimal integer-valued solution for the interval covering LP above. Alternatively, we can show how any optimal fractional LP solution can be “rounded” to a feasible integer-valued solution without making its objective value any worse. In problem 238 shortly, we will see how to do this for the interval covering LP above. Quite commonly, however, we establish integrality by showing that the structure of our problem earns it admission into a broader class of problems known to satisfy integrality, such as:

- **Consecutive Ones.** Each row (or alternatively, each column) of our constraint matrix $A$ must have a single contiguous block of 1s, all other entries being zero. Note that each constraint $\sum_{i \sim t} x_i \geq 1$ above satisfies this property.

- **Network Matrices.** Each row (or alternatively, each column) of our constraint matrix $A$ must have a single +1 and a single -1 entry, all others being

---

4Every feasible solution $x \in \{0, 1\}^n$ corresponds to what is called the binary incidence vector of some valid interval cover. An incidence vector is a common way of encoding a subset when solving a mathematical program; for example, we would encode the subset $\{2, 4, 5\}$ of the ground set $\{1, \ldots, 6\}$ as the incidence vector $(0, 1, 0, 1, 1, 0)$, with ones corresponding to the elements that are present in the subset.
12.3. INTEGRALITY

Figure 12.4: (a) An example instance of the minimum-cost interval cover problem, with an optimal solution shaded. Dashed lines indicate how time is partitioned into timeslots, within each of which the set of active intervals does not change. In (b), we see an equivalent formulation of this problem as a shortest path problem in a DAG, where the cost of an edge is equal to the cost of the interval it points at. Generalization to covering a circle is shown in (c), an instance with three unit-cost intervals. If we alternatively generalize the problem to covering a tree with a minimum-cost collection of paths, we might get a solution like the one pictured in (d).

zero. These matrices so-named because they are often used to encode directed graphs. For example, a row with a +1 in column $i$ and a −1 in column $j$ might represent an edge from node $i$ to node $j$ (or vice versa, depending on the application)\(^5\).

- **Total Unimodularity.** If the determinant of every square submatrix of $A$ is either −1, 0, or 1, then $A$ is a *totally unimodular* matrix, and the LP $\max \{c^T x : Ax \leq b\}$ will satisfy integrality for any integer vector $b$.

- **Total Dual Integrality.** The LP $\max \{c^T x : Ax \leq b\}$ is *totally dual integral* if its “dual” LP satisfies integrality for any integer vector $c$ (we will discuss LP duality shortly). Any such LP also satisfies integrality. [Details]

\(^5\)LPs based on “row” network matrices can often be solved as shortest path problems, as highlighted in problem ??, For example, if our example LP is reformulated so that it involves a “row” network matrix, this (perhaps unsurprisingly) gives the same shortest path problem as if we had expressed the dynamic programming solution of the interval covering problem as an equivalent shortest path problem in a directed acyclic graph, as shown in Figure 12.4(b). LPs based on “column” network matrices can often be solved as network flow problems, as shown in problem ??, We discuss network matrices further in Chapter ?? in connection with another special matrix called the *Laplacian*. 
The properties above appear in order of generality: any problem satisfying consecutive ones can be easily reformulated using a network matrix, any LP based on a network matrix is totally unimodular, and total unimodularity implies total dual integrality [Details]. Since our example problem satisfies the consecutive ones property, it satisfies all of these.

To give some idea of how sensitive the integrality property can be, the natural LP formulation of the interval covering problem on a circle is not integral, since the unique optimal solution to the instance shown in Figure 12.4(c) involves selecting half of each interval. This provides one total unit of coverage all around the circle, at a total cost of just 3/2. The generalization of interval covering to a tree, shown in Figure 12.4(d), is NP-hard, so its LP relaxation ought not be integral, or else we would be able to solve an NP-hard problem in polynomial time. However, if we can root the tree so every path in our input set is “vertically” oriented, with one endpoint being an ancestor of the other, then the problem becomes integral again! This can be shown as a consequence of total unimodularity, and we also give an algorithm for constructing an integer optimal solution later in problem 241.

12.4 Relaxation and Rounding

For most problems (e.g., NP-hard problems), integrality is too much to ask. Usually, if we take the natural integer LP formulation of a problem and “relax” it to a standard LP by removing integrality constraints, we get an optimal solution that is fractional. However, this fractional LP relaxation of a problem can still be quite useful. In this section, we use it to develop approximation algorithms, and in the next chapter, we use it to build heuristics that search for optimal solutions.

**Relaxation.** When confronted with a hard optimization problem, we often consider “relaxing” it to an easier problem by weakening or removing constraints. Here, we focus on relaxing integer LPs to standard LPs by removing integrality constraints. This is a particularly common type of relaxation, although many others are found in the optimization literature. In general, relaxation is helpful in giving us a new problem that can be much easier to solve. Moreover, *the optimal solution of a relaxed problem is always least as good as that of the original problem in terms of objective value.* This follows easily from the fact that optimal solutions of the original problem are still feasible; however, relaxation might expand the set of feasible solutions to include new solutions that have even better objective value. For example, if we consider the circular variant of the interval covering problem shown in Figure 12.4(c), the optimal integer solution must select at least two intervals, for a total cost of 2, while the optimal fractional solution uses half of each interval for an even lower total cost of only 3/2.

The fact that relaxation lets us compute a better-than-optimal solution value can be quite helpful. For example, it can be difficult to assess the quality of a candidate solution $x$ for an NP-hard problem, since the optimal solution value of an NP-hard problem is hard to know. However, if $x$ is at most 17 units away in objective value from the optimal solution value of a relaxation of our problem (which we can compute efficiently), then we know $x$ is also at most 17 units away from an optimal solution to the original NP-hard problem. Duality, which we study shortly, is often
used in a similar way, to obtain bounds on distance to optimality. In this section, we focus on approximation algorithms, where we want to find a solution that is provably close to optimal. Again, since the exact optimal solution value is usually hard to know, we often bound the distance to optimality by comparing against the optimal solution of a relaxation of the original problem.

**Rounding.** The obvious downside of relaxation is that it can give us a new problem that differs substantially from the problem we actually want to solve. The solution of an LP relaxation might tell us, for example, to route 2.3 trucks from a factory to a warehouse, or to schedule 1.5 pilots to fly an airplane. One doesn’t need to be a pilot to see the difficulty with this situation. To fix things, we look for a procedure to convert our better-than-optimal relaxed solution back to a feasible solution to the original problem, hopefully without compromising quality too much in the process, so we still end up close to optimal. In the context of converting the fractional solution of an LP relaxation back to a feasible integer LP solution, we call this process *rounding.*

The best way to round is usually highly problem-specific. In simple cases, we follow the literal meaning of “rounding” and choose for each decision variable whether to round it up or down. However, even this can be non-trivial, since if we have $n$ decision variables, there are $2^n$ possible integer solutions we can reach in this fashion, only a handful of which may even be feasible and hard to find. In other situations, we may opt to use a more sophisticated method that does much more than simply round decision variables up or down.

To be useful for building a strong approximation algorithm, a rounding algorithm should enable us to argue good bounds on the quality of its output relative to that of the initial fractional solution. Many such techniques have been studied, with prominent examples including the following:

- **Exploiting Basic Solution Structure.** For many relaxations, we can show that an optimal basic solution has special structure that can make rounding easier. For example, a basic solution to the fractional knapsack problem has $n - 1$ variables in \( \{0, 1\} \), with at most a single variable taking a fractional value in \([0, 1]\). Leveraging this fact, we can develop a PTAS for the knapsack problem (different from the one we discussed in the last chapter), which generalizes to give a PTAS for the more complicated “vector” knapsack problem, where item sizes are vector-valued [Further details]. Another example we will see in Chapter ?? is the exploitation of laminarity in basic solutions to obtain a 2-approximation algorithm for the challenging Steiner network problem.

- **Threshold Rounding.** A very simple approach for rounding is to round up to 1 all fractional variables above a certain threshold (usually 1/2), rounding all others down to zero. The idea here is that if we only round up variables that are already 1/2 or larger, then we at most double the cost of our solution. This leads to a particularly simple 2-approximation algorithm for the minimum node cover problem [Further details]. In fact, the minimum node cover problem is one of a handful of problems whose fractional LP relaxations are known to be half-integral, where we can find an optimal solution \( x \) with each \( x_i \in \{0, 1/2, 1\} \). Other good examples are the straightforward fractional relaxation of the integer program for computing a matching in a general graph.
CHAPTER 12. INTEGER AND LINEAR PROGRAMMING

(Chapter ??), and the multiway cut problem (Chapter ??). Threshold-based rounding is often used in conjunction with such problems.

• Randomized Rounding. Many fractional relaxations involve decision variables with values in the range \([0, 1]\), which we can interpret as probabilities. A natural rounding approach here is to round up each variable \(x_i\) independently with probability \(x_i\). That is, a variable \(x_i = 0.99\) would have a 99% chance of being rounded up. This seems somewhat reasonable, since the fractional solution seems to be telling us that the extent to which each variable “wants to be included” in our solution. Using a form of aggressive randomized rounding, we can easily build an \(O(\log n)\)-approximation algorithm for the minimum set cover problem (Section 10.5.2), or equivalently for the minimum dominating set problem in a graph (Section ??). [Further details]

• Rounding to an Integral Special Case. Although a problem may not have an integral LP relaxation, one of its special cases might, which we can sometimes exploit to obtain a good approximation algorithm for the general problem. Two simple examples include the NP-hard interval cover problem on a tree (Figure 12.4(d)) [Further details], and the NP-hard minimum multicut problem in a tree [Further details], both of which admit simple 2-approximation algorithms via this approach.

Problem 237 (SONET Ring Loading Revisited). Please describe a simple 2-approximation algorithm for the NP-hard SONET ring loading problem (problem 200), based on rounding the solution of an LP relaxation of the problem. [Solution]

Problem 238 (Carthéodory’s Theorem Applied to Rounding). For problems with integral LP relaxations, rounding is usually not necessary, since we can get an optimal integer-valued solution immediately from solving the LP relaxation. Here, we highlight one practical situation where rounding still does play a useful role, allowing us to introduce another form of randomized rounding as well as Carthéodory’s theorem, which tells us that for an LP with a bounded feasible region in \(n\) dimensions, any feasible solution \(x^*\) can be written as a convex combination of at most \(n + 1\) feasible basic solutions: 

\[ x^* = \sum \lambda_j x^{(j)} \]

where the \(\lambda_j\)’s are nonnegative and sum to one (however, the theorem generally does not say how to do this in an algorithmically efficient fashion). For instance, any point in a triangle in the 2D plane can be written as a convex combination of its three corners. In the case of an LP satisfying integrality, we can therefore write any solution \(x^*\) as a nonnegative-weighted combination of \(n + 1\) integer-valued solutions \(x^{(1)} \ldots x^{(n+1)}\).

(a) The minimum-cost interval covering problem we used as our earlier example doesn’t quite fit the requirements of Carthéodory’s theorem, since its feasible region is unbounded (e.g., we can take any \(x_i\) and increase it by an arbitrary amount while remaining feasible). Generalizations of Carthéodory’s theorem have been devised for situations like this, but for simplicity, we can effectively restore boundedness by working only with optimal solutions. For a challenge, please devise an efficient algorithm for decomposing any optimal fractional solution to our minimum-cost interval covering LP into a convex combination of optimal integer solutions. [Solution]

(b) Assuming we can decompose \(x^*\) into a convex combination of integer solutions as above, a common rounding approach is to pick one of the \(x^{(j)}\)’s at random, with \(x^{(j)}\) being picked with probability \(\lambda_j\). Argue that this approach preserves the cost of our solution in expectation: 

\[ E[c^T x^{(j)}] = c^T x^* \]

As a consequence, argue that one of the \(x^{(j)}\)’s must have objective value at least as good as \(x^*\), giving another way to prove
integrality of our LP relaxation. Similarly, argue that the value of each decision variable in $x^*$ is preserved in expectation: $E[x_{i}^{(j)}] = x_{i}^*$. In the context of shift scheduling, a fractional solution with $x_{i}^* = 1/3$ might correspond to a shift (interval) that we only want to include in our solution 1/3 of the time — say, one out of every three days on average. If each day we pick a random integer-valued schedule according to the rounding scheme above, the fact that $E[x_{i}^{(j)}] = 1/3$ means that we indeed include interval $i$ with this desired frequency. In general, we can simulate any fractional solution (in a time-averaged sense) by multiplexing a series of random integer solutions over time. Another example of this idea appears in problem ??.

Several other prominent rounding techniques appear in this book. For example, in Chapter ??, we will see methods for rounding fractional flows and bipartite assignments to produce “unsplit” flows and assignments (a special case of a more general technique sometimes known as pipage rounding). In Chapter ??, we will introduce the techniques of ball rounding (also called region growing) for rounding the fractional relaxation of a cut problem, and the technique of iterated rounding for approximating network design problems. Also, in Chapter ??, we will study the use of rounding based on spectral relaxations for other cut problems. Many other rounding techniques appear in the research literature on approximation; the advanced reader is advised to consult a specialized textbook for further examples.

**Problem 239 (Integrality Gap).** The integrality gap of a fractional relaxation measures how far apart the relaxation can be from its original problem in terms of objective value. Assuming we are dealing with a minimization problem, the integrality gap of a relaxation is the largest possible ratio, over all inputs, of the optimal solution value of the original problem to that of the relaxation. For example, the natural LP relaxation for circular interval cover has integrality gap at least 4/3, since for the instance shown in Figure 12.4(c), the optimal integer solution has cost 2 while the optimal fractional solution has cost 3/2. The integrality gap of a relaxation is intrinsically related to the approximation guarantees we can achieve using a rounding algorithm based on that relaxation. If a relaxation satisfies the integrality property, its integrality gap is one.

(a) If the integrality gap of a relaxation is at least $x$, then any rounding-based approximation algorithm using that relaxation generally cannot have an approximation guarantee better than $x$. Please argue why this is the case, and also show that the integrality gap of the natural LP relaxation for circular interval cover is at least 2. [Solution]

(b) If there exists a rounding-based approximation algorithm with performance guarantee $y$ using some relaxation, then the integrality gap of that relaxation cannot exceed $y$. Please argue why this is the case, and give a rounding-based 2-approximation algorithm for the circular interval cover problem$^6$, thereby showing that the integrality gap for its natural LP relaxation is at most 2 (and hence equal to 2, when combined with the result from the previous part). [Solution]

If the integrality gap of a relaxation is too large, we sometimes try to remedy this by adding more constraints, in order to “tighten” the relaxation and make its optimal value closer to that of the original problem.

---

$^6$Note that a 2-approximation algorithm for the circular interval covering problem, while nice as an illustrative example, is not particularly exciting, since the problem can easily be solved exactly in polynomial time by first “guessing” one of the intervals in the optimal solution, then by computing the remainder of the solution (a non-circular minimum-cost interval covering problem) using dynamic programming. The minimum-cardinality variant can be solved even more efficiently; see problem 193.
12.5 Linear Programming Duality

We introduced some of the high-level ideas behind duality in optimization back in Section 10.5.1, showing how certain pairs of problems are “dual” to each-other, with a max-min relationship between their optimal solution values — the maximum of one problem being at most the minimum of the other. LP duality is one of the most prominent forms of duality we see in optimization, and also one of the most important aspects of the theory of linear programming. It not only gives us valuable structural insight into problems and their solutions, but it also provides a useful framework that can be leveraged in algorithm design.

For every LP, we can construct a dual LP by “transposing” the problem in a certain mechanical fashion. Conveniently, the dual of any problem \( \max \{ c^\top x : Ax \leq b \} \) in canonical form is the problem \( \min \{ b^\top y : A^\top y = c, y \geq 0 \} \) in standard form. If the original, or primal, problem involves minimization, its dual is a maximization problem, and vice versa. For every variable in the primal problem, there is a corresponding constraint in the dual, and for every constraint in the primal problem, there is a corresponding variable in the dual\(^7\). The objective coefficient vector \( c \) and the right-hand side vector \( b \) exchange roles, and the constraint matrix \( A \) is transposed. If we take the dual of a problem twice, we end up with the original primal problem.

To give a concrete example, the fractional relaxation of our minimum-cost interval covering example problem,

\[
\text{Minimize: } \sum_{i \in I} c_i x_i \\
\text{Subject to: } \sum_{i \sim t} x_i \geq 1 \text{ for all } t \in T \\
\quad \quad \quad \quad x_i \geq 0 \text{ for all } i \in I,
\]

dualizes to

\[
\text{Maximize: } \sum_{t \in T} y_t \\
\text{Subject to: } \sum_{i \sim t} y_t \leq c_i \text{ for all } i \in I \\
\quad \quad \quad \quad y_t \geq 0 \text{ for all } t \in T.
\]

**Intuition Behind the Dual.** For many common LPs, the dual problem has a natural interpretation, so a good way to check if you have properly formed the dual is to see if the dual problem “makes sense” as a problem in its own right. For example, the dual of the fractional interval covering problem above is a “fractional timeslot packing” problem. Its goal is to allocate a maximum total amount of weight among the timeslots in \( T \) such that at most \( c_i \) weight is allocated to the timeslots spanned by interval \( i \).

There are many ways to develop a stronger intuition about the “meaning” of the dual problem and how it relates to the primal problem. For example, a geometric

\(^7\)When dualizing a problem that is not in canonical or standard form, one must be slightly careful to follow the right rules for dualizing variables and constraints. Equality constraints always dualize to variables that are unrestricted in sign, and vice versa. For a maximization problem, every \( \leq (\geq) \) constraint dualizes to a variable that is constrained to be \( \geq 0 (\leq 0) \), and every \( \geq 0 (\leq 0) \) variable dualizes to a \( \geq (\leq) \) constraint.
12.5. LINEAR PROGRAMMING DUALITY

Figure 12.5: A geometric interpretation of duality: the primal problem max\{c^T x : Ax \leq b\} strives to find the farthest feasible point \(x^*\) in the direction of the objective \(c\). The dual problem \(\min\{b^T y : A^T y = c, y \geq 0\}\) seeks to express \(c\) with an “optimal” nonnegative combination of columns of \(A^T\) (denoted \(A^T_1, A^T_2, \text{etc.}\)), corresponding to rows of \(A\), or constraints in the primal. If \(c\) is a gravitational force pulling on \(x^*\), the dual solution tells us the forces exerted on the tight constraints supporting \(x^*\) (two of them in this case, shown as bold lines). These forces combine to match \(c\) exactly.

relationship between the two is shown in Figure 12.5. As we see in the figure, dual variables reflect the importance of constraints in the primal problem: dual variables are zero for constraints that are not tight, and they are large for constraints that play a key role in supporting the optimal primal solution. For this reason, the dual is useful when performing sensitivity analysis, looking at the impact of small changes to primal constraints on its optimal objective value. Constraints corresponding to dual variables with value zero can usually be perturbed slightly with no consequence, while those with higher corresponding dual values have a more substantial impact. We will revisit this issue in a moment when we discuss complementary slackness and the dual simplex algorithm.

**Relating Objective Values.** Consider a primal problem with an objective of maximizing \(c^T x\) and its dual, with an objective of minimizing \(b^T y\). If the primal problem is infeasible, the dual will be unbounded, and vice versa. Otherwise, if \(x\) and \(y\) are any two feasible solutions to their respective problems, then we always have

\[ c^T x \leq b^T y. \]

This is known as weak duality, and it is quite easy to prove. [Details] Moreover, if \(x^*\) and \(y^*\) are optimal solutions to their respective problems, then equality is attained:

\[ c^T x^* = b^T y^*. \]

This is known as strong duality, and it is slightly more challenging to prove. [Details]

Perhaps the most important application of these relationships is that we can use them to certify the optimality or near-optimality of an LP solution. In order to show that the solution of an LP is optimal, we need only demonstrate a feasible dual solution with matching objective value (and in this case, we know the dual solution must also be optimal). If we find feasible primal and dual solutions whose objectives differ by some amount \(\Delta\), then we know that each solution is at most \(\Delta\) units away from optimality.
12.5.1 Complementary Slackness

We currently know two ways to certify optimality of a solution to an LP: we can either check that all variables have nonnegative reduced cost, or we can demonstrate a feasible dual solution with matching objective value. We now introduce a third method. A feasible LP solution $x$ might satisfy some inequality constraints with equality. We have called such constraints tight, and we call the others slack. Similarly, we say a variable $x_i$ constrained to be nonnegative is tight if $x_i = 0$, and slack if $x_i > 0$ (the same notion applies for a variable constrained to be nonpositive). If $x$ is a feasible solution to an LP and $y$ is a feasible solution to its dual, then $x$ and $y$ satisfy the complementary slackness property if any time a variable or constraint (in either problem) is slack, its corresponding dual constraint or variable must be tight; both cannot be slack. It is easy to prove that $x$ and $y$ are both optimal for their respective problems if and only if they satisfy complementary slackness.

Complementary slackness gives a nice correspondence between basic solutions of a primal LP and its dual. Recall that a basic solution for $\max \{c^T x : Ax \leq b\}$ is a set of $n$ tight constraints (corresponding to linearly-independent rows of $A$), where the other $m - n$ constraints are generally slack. For the dual problem $\min \{b^T y : A^T y = c, y \geq 0\}$, a basic solution consists of $n$ variables that are generally slack (corresponding to linearly-independent columns of $A$). The remaining $m - n$ variables are tight. Complementary slackness gives us a correspondence between the tight/slack constraints/variables in the primal basis and the slack/tight variables/constraints in the dual basis. Using this relationship, we can easily compute an optimal basic solution to the dual problem given an optimal basic solution to the primal problem, and vice versa.

12.5.2 The Dual Simplex Algorithm

Due to the correspondence between basic solutions in an LP and its dual, we can solve an LP by running the simplex algorithm on its dual, which in the context of the primal appears as if we are moving from basic solution to basic solution, maintaining “optimality” (i.e., satisfaction of the reduced cost optimality conditions) as an invariant at all times, and terminating when we finally reach a feasible solution. This approach is known as the dual simplex algorithm, and it highlights two common ways to perform iterative refinement in general for any optimization problem:

- **Primal Iterative Refinement**: Maintain feasibility at all times while working towards optimality.
- **Dual Iterative Refinement**: Maintain satisfaction of optimality conditions at all times while working towards feasibility.

**Re-Optimizing Solutions.** The original simplex algorithm can be effective at quickly re-solving (also known as re-optimizing) a problem after changing some of the objective coefficients or inserting/deleting variables, since it typically only requires a small number of additional pivots to converge back to an optimal basis. By symmetry, the dual simplex algorithm can be useful for re-optimizing a problem after changing entries in the right-hand-side vector, or after inserting or deleting...
12.5. LINEAR PROGRAMMING DUALITY

A straightforward analysis shows that this algorithm produces an integer-valued primal solution $x$ and a dual solution $y$ with equal objectives, so both are therefore

---

DRAFT. Not for redistribution. Copyright © Brian C. Dean
optimal [Details]. Moreover, if we carefully reformulate the algorithm above, we can transform it into a “stand alone” $O(n \log n)$ combinatorial procedure with no dependence on LP tools or terminology. In fact, with enough manipulation, we end up with essentially the same method as the $O(n \log n)$ dynamic programming algorithm we devised back in problem 209. [Details]

**Problem 240 (Minimum-Cost Dominating Set in a Tree).** Please design a fast primal-dual algorithm (much like the one above) for computing a minimum-cost dominating set in a tree, described in Problem 229(b). [Solution]

**Problem 241 (Covering a Rooted Tree with Vertical Paths).** We mentioned earlier that the minimum-cost interval covering problem in a tree (Figure 12.4(d)) is generally NP-hard, but solvable in polynomial time if the tree is rooted such that every path in our input set is “vertically” oriented — with one endpoint being an ancestor of the other. Please describe a simple primal-dual algorithm for solving this special case, assuming the tree is given to you rooted appropriately. [Solution]

One of the original applications of the primal-dual approach was to solve a “weighted” problem (with an objective function like $\sum_i c_i x_i$) as a series of “unweighted” problems (with objective functions like $\sum_i x_i$) in a systematic fashion. For example, suppose we modify our primal-dual algorithm for the minimum-cost interval covering problem so we no longer incrementally construct the primal solution $x$. At each iteration of the main loop in step 2, we instead look at the set $S \subseteq I$ of all currently-tight dual constraints, and note that complementary slackness tells us that only variables $x_i$ for $i \in S$ can be increased above zero. We therefore solve a minimum-cardinality interval cover problem (e.g., with the simple greedy method from Section 10.5) using only intervals in $S$. If this is able to cover all of $[0, 1]$, we return it as our final answer; otherwise, we proceed as before, increasing the dual variable $y_t$ for the leftmost timeslot $t$ that we are unable to cover, thereby adding new tight dual constraints to $S$, so our next invocation of a minimum-cardinality cover algorithm will have more chance of success. With careful analysis, we can show that this also gives an optimal solution. [Further elaboration]

**Approximation Algorithms.** Primal-dual algorithms are quite prominent in the realm of approximation. Here, we simultaneously construct solutions to an integer LP and the dual of its fractional relaxation, trying to approximately respect complementary slackness throughout the process. The distance between the objective values of the resulting solutions is then used to establish our approximation guarantee. A simple example of this approach is a 2-approximation for the minimum-cost node cover problem [Details]. Primal-dual algorithms are generally preferred to those based on relaxation and rounding, since they are usually simple, combinatorial, fast, and avoid solving a generic LP as a preprocessing step.

**12.5.4 Connections with Game Theory**

The simplest optimization problems involve minimizing or maximizing a single objective. It is also common to find problems involving several different objectives, all of which may be hard to satisfy simultaneously — in this setting, we often produce as output a “frontier” of Pareto-optimal solutions (see Problem 143), rather than any one single solution. Taking things even further, many optimization settings
involve a large number of independent agents who are each “selfishly” trying to optimize purely for their own benefit. For example, cars driving through an automobile transportation network are all trying to optimize their individual travel times, with little regard for any sort of “centralized” common objective (we consider this particular problem in further detail in Section ??).

The field of game theory studies systems involving multiple agents that are all trying to optimize independently for their own well-being. In fact, there has recently been substantial interest in the subfield of algorithmic game theory, where we try to design effective algorithmic mechanisms to coordinate systems of this sort — for example to perform collective decision-making with voting, or resource allocation via auctions\footnote{Even mechanisms like voting that may appear simple can sometimes involve surprisingly stubborn mathematical hurdles. For example, if you look up Arrow’s impossibility theorem, you will see that there are fundamental limitations on the design of an effective voting mechanism. We discuss auctions in more detail later in Chapter ??}. Here, we discuss fundamental results from game theory that are intricately related to linear programming and duality.

Consider a game between two players, a row player who can choose from a set of $m$ strategies, and a column player with $n$ available strategies. A payoff matrix tells us the amount won by both players for every possible pair of strategies they might play against each-other. For example, in the well-known “prisoners’ dilemma” game (Figure 12.6(a)), two prisoners suspected of participating in the same crime are interrogated in separate rooms, and each is given the option of confessing (implicating the other prisoner), or staying silent. If both confess, they each spend 5 years in jail. If neither confesses, they each spend only 1 year in jail, and if only one confesses, then that prisoner is rewarded by being released, while the other spends 10 years in jail. In a zero-sum game, one player’s gain is the other’s loss, such as in the classic “rock, paper, scissors” example shown in Figure 12.6(b). Each cell in the payoff matrix therefore needs only to contain a single number, giving the payoff for the row player. We henceforth focus on zero-sum games.

Beyond choosing a single, “pure”, strategy, a player may opt to choose a probability distribution over strategies, known as a mixed strategy. We denote a mixed strategy over $m$ pure strategies with a length-$m$ nonnegative vector $x$ with $\sum x_i = 1$, where $x_i$ is the probability of choosing pure strategy $i$. Letting $A$ denote our $m \times n$ payoff matrix for the row player, and letting $x$ and $y$ be mixed strategies for the row and column players, the expected payoffs are $\langle A, y \rangle_x$ for the row and $\langle x, A^T \rangle_y$ for the column.

\begin{tabular}{|c|c|c|}
  \hline
  & Confess & Stay Silent \\
  \hline
  Confess & -5, -5 & 0, -10 \\
  Stay Silent & -10, 0 & -1, -1 \\
  \hline
\end{tabular}

\begin{tabular}{|c|c|c|c|}
  \hline
  & Rock & Paper & Scissors \\
  \hline
  Rock & 0, 0 & -1, 1 & 1, -1 \\
  Paper & 1, -1 & 0, 0 & -1, 1 \\
  Scissors & -1, 1 & 1, -1 & 0, 0 \\
  \hline
\end{tabular}

\begin{figure}
\caption{Examples of payoff matrices for two-player games, where each entry of the form $(r, c)$ gives the payoffs for the row and column players if they play a certain pair of strategies against each-other. In (a), we see the “prisoners’ dilemma” game, and (b) is the familiar “rock, paper, scissors” game.}
\end{figure}
column players, the expected payoff for the row player can then be written simply as $x^\top Ay$, an objective the row player wishes to maximize, and that the column player wishes to minimize. LP duality can now be used to show von Neumann’s famous minimax theorem, allowing us to reverse the order of maximization and minimization in the zero-sum game setting above. If $A$ is any $m \times n$ matrix, then

$$\max_x \min_y x^\top Ay = \min_y \max_x x^\top Ay,$$

where $x$ and $y$ are constrained to be mixed strategies [Proof]. If the game is viewed sequentially, one player picking a strategy and then the other, then the left-hand side gives the row player’s outlook: pick an optimal mixed strategy $x$ so as to limit the effectiveness of the best response $y$ of the column player (knowing $x$). The right-hand side gives the viewpoint of the column player, selecting the best mixed strategy $y$ that limits the effectiveness of the best response $x$ the row player can choose (knowing $y$). Von Neumann’s theorem tells us that both interpretations lead to the same result. In fact, the result above is a just special case of the theorem where equality is attained in a much more general max-min relationship, where for any matrix $A$, we have $\max_i \min_j A_{ij} \leq \min_j \max_i A_{ij}$, or equivalently for any function $f(x, y)$ we have $\max_x \min_y f(x, y) \leq \min_y \max_x f(x, y)$. [Simple proof]

**Nash Equilibria.** Perhaps the most famous result in game theory is that every game always admits a mixed Nash equilibrium — a set of mixed strategies for the various players that is a “fixed point” in the sense that no player has any incentive to switch to a different strategy, subject to the choices of the other players remaining the same. For example, the unique mixed Nash equilibrium for the prisoners’ dilemma game has both prisoners confessing, even though this is paradoxically worse for them both than if they were to both remain silent!9 For the rock-paper-scissors example, the unique equilibrium is for each player to select a pure strategy uniformly at random. In a zero-sum game, a mixed Nash equilibrium consists of mixed strategies $x^*$ and $y^*$ that are both simultaneously optimal in response to each-other:

$$\max_x \min_y x^\top Ay = \min_y (x^*)^\top Ay = (x^*)^\top Ay^* = \max_x x^\top Ay^* = \min_y \max_x x^\top Ay.$$

This result follows from the LP underpinnings of the max-min duality relationship above, with $x^*$ and $y^*$ being optimal solutions of an LP and its dual. [Details]

**Yao’s Minimax Theorem.** A nice application of the duality framework above is Yao’s minimax theorem, which allows us to prove lower bounds on the expected running time of a randomized algorithm (for simpler worst-case lower bounds, we can use methods such as decision trees, introduced back in Chapter 3). Our model here is a zero-sum game between a malicious adversary, who chooses a probability distribution over possible inputs to make our algorithm run slowly, and our randomized algorithm, which picks from a probability distribution over deterministic algorithms with a goal of running quickly. For any distribution $x$ over inputs (i.e., for any mixed strategy for the adversary), the minimax framework above lets us lower bound the expected running time of any randomized algorithm by the expected running time of the best deterministic algorithm that is tuned to deal with input from the distribution $x$. As a consequence, we can show for example that any randomized comparison-based sorting algorithm must take $\Omega(n \log n)$ expected time. [Further details]

9Indeed, game theory often provides a good mathematically-grounded way to explain why individuals or groups often act in ways that seem paradoxical or non-intuitive.
13. Heuristics and Exhaustive Search

In the real world, we routinely grapple with “hard” computational problems with no efficient algorithms in sight. Indeed, some of the most crucial and cutting-edge problems faced by science and industry are extremely complex, most likely NP-hard or worse. At the end of the day, these problems still need good solutions that work well in practice, even though things may look dire from a theoretical perspective.

This chapter studies heuristics — algorithmic techniques that typically lack any sort of theoretical guarantee for either running time or solution quality, but that nonetheless tend to perform well in practice\(^1\). Heuristics are often under-emphasized by the theory community, since there is less to say about them from a mathematical perspective, and since theoreticians often prefer dealing with hard problems by developing approximation algorithms with provable performance guarantees. However, for many applications, even a reasonable result like a 2-approximation algorithm may not be good enough, and stronger PTAS algorithms with guarantees like \((1+\varepsilon)\) often have impractical running times like \(O(n^{1/\varepsilon})\). To the mathematician’s chagrin, heuristics sometimes feel more like art than science. Success with heuristics can depend on your experience as an implementor, and even just plain old luck.

On the bright side, people solve hard computational problems every day, even with fairly large input sizes. Not all problems that look hard in theory end up being hard in practice, mostly because real-world inputs often have special structure and bear little resemblance to worst-case inputs. Fortunately for us, the universe does not always behave like our hypothetical malicious adversary.

The Importance of Testing. A priori, it can be difficult to know whether a “hard” problem will end up being well-behaved in practice. Empirical testing is usually required, and as we mentioned back in Section 1.4.1, we need to be quite careful when performing such testing if we want to obtain reliable results. For example, if we run a head-to-head comparison of different heuristics only on a few inputs, on very small inputs, or on completely random inputs, then we may not end up accurately predicting performance on authentic real-world inputs. Our

\(^1\)Some authors use the word “heuristic” to mean only an algorithm that provides no guarantee on its solution quality, rather than a technique that speeds up the running time in practice for an “exact” algorithm (one that always outputs an optimal solution). We use the term for both purposes. These two classes of techniques are closely related, after all, since you can typically terminate an “exact” algorithm early to obtain an algorithm with a fast running time, but with no guarantee on solution quality.
testing must take into consideration that some heuristics deliver better solutions the longer they run — perhaps one heuristic shows rapid improvement but then gets stuck in a plateau, while another makes slow progress but ultimately reaches a better solution. In addition, most heuristics involve a number of user-specified parameters, which need to be tuned properly during the testing process or else our results may be unreliable. For example, a genetic algorithm might behave completely differently if we change its underlying population size or mutation rate. To be sure our parameters are tuned well, we may even optimize them using a secondary heuristic, sometimes called a metaheuristic.

13.1 Iterative Refinement of a Single Solution

Many heuristics are based on iterative refinement, continually improving either a single solution or a population of solutions over time. In this section, we discuss techniques that refine a single solution, usually starting with a solution that is generated randomly or by some other simple heuristic like a greedy algorithm. In a few pages, we switch gears and discuss population-based approaches.

13.1.1 Local Search

Local search (or neighborhood search) methods refine a solution $x$ by searching for better solutions in a small “neighborhood” around $x$, characterized by making small changes to $x$. By repeatedly moving to a better neighboring solution, we will eventually reach a solution that is locally optimal — at least as good as any other solution in its neighborhood.

Prototypical examples of local search neighborhoods are shown in Figure 13.1:

- In (a), we see a common neighborhood of size $\Theta(n^2)$ used for the traveling salesman problem (Section ??), where a tour of a graph is modified by removing any two of its edges and replacing them with "diagonals".

- Order-based optimization problems ask us to find an optimal ordering of $n$ input elements; examples include many types of scheduling problems, rank aggregation (problem 63), and even the traveling salesman problem, since a tour of a graph is just an ordering of its nodes. For such problems, we might consider the neighborhoods in (b) consisting of all $\Theta(n^2)$ pairwise swaps of elements, or of all $n - 1$ adjacent swaps. The neighborhood shown in (a), re-interpreted in this context, consists of all $\Theta(n^2)$ subarray reversals.

- In (c), we are trying to partition a set of points into two clusters of equal size (Section ?? contains more detail on how to characterize a "good" clustering). A simple $\Theta(n^2)$-size neighborhood here is to consider swapping the roles of two points in opposite clusters.

\footnote{Although the term “metaheuristic” is used in slightly different ways, it generally refers to a combination of several heuristics, possibly arranged in a pipeline (each one further refining the output of another), in a competitive parallel setting (where we run several independent heuristics simultaneously in hopes that one of them finds a high-quality solution), or in a hierarchical configuration (with one controlling / optimizing the parameters of the others).}
13.1. Iterative Refinement of a Single Solution

- The maximum independent set problem (Section ??) asks us to find the largest set of nodes in a graph where no two are adjacent. Here, we might consider a neighborhood of size \( \Theta(n^3) \) consisting of removing one node and adding two others to the solution. Note that although any neighboring solution will be better, most will likely not be feasible, an issue we will revisit shortly.

**Neighborhood Search Strategy.** Our movement strategy can vary according to two common possibilities. We can either move to a better neighboring solution the instant we discover it, or we can search the entire neighborhood and then move to the best neighboring solution. The first approach takes faster steps, but the second approach might take better steps. If using the first approach, it may be advisable to search the neighborhood in random order, or else we might concentrate too much energy on refining just one location. For example, with the adjacent swap neighborhood in Figure 13.1(b), if we always scan from left to right and make the first available swap, we may end up neglecting the right-hand side of the ordering.

**Neighborhood Size.** Small neighborhoods allow each iteration to run quickly, but they increase the likelihood of getting stuck at a locally-optimal solution of low quality. Larger neighborhoods might give us higher-quality solutions, but at the expense of each iteration running slower. A reasonable compromise might be to search in a layer-by-layer fashion. For example, we might first visit all neighboring solutions only 1 swap away; if we fail to find any better solutions, then we may search neighboring solutions only 2 swaps away, and so on, up to some desired limit. Some local search algorithms are even capable of searching neighborhoods of exponential size by using a clever combinatorial algorithm. [Example: the traveling salesman problem with bitonic neighborhoods]

**Escaping Local Optima.** A fundamental drawback of local search methods is that they only produce locally optimal solutions. To mitigate this issue, we often
re-run the algorithm from a large number of random starting points, taking the best solution as our final answer. Another popular remedy is tabu search, which allows occasional movement from a solution \( x \) to a worse neighboring solution \( x' \) (say, right after reaching a local optimum), in hopes that we can climb out of reach of one local optimum and enter a region from which we will converge to another better one. To prevent the next iteration from simply turning around and moving back from \( x' \) to \( x \), we temporarily add \( x \) to a list of forbidden solutions. The structure and size of this list may vary depending on the particular problem, and usually requires a certain amount of empirical testing to determine. In a moment, we will discuss how the technique of simulated annealing also helps us escape from local optima.

**Feasibility Issues.** As we have seen in the preceding chapter, iterative refinement algorithms generally come in two flavors, primal algorithms that maintain feasibility of a solution while improving its quality, and dual algorithms that maintain satisfaction of optimality conditions while working towards feasibility. In the realm of heuristics, since we lack guarantees on final solution quality, we generally focus on primal methods, since a feasible but not-quite-optimal solution is usually preferred to an “optimal” but not-quite-feasible one. That being said, preserving feasibility at all times can sometimes be a challenge. For example, with packing and covering problems (e.g., the maximum independent set problem shown in Figure 13.1(d)), we often end up a large number of infeasible solutions in any natural neighborhood we might define. By insisting on remaining feasible, we may waste a large amount of time searching through infeasible neighboring solutions. Sometimes we do allow movement to infeasible solutions, but we penalize them so we don’t stray too far from feasibility, and we often increase the penalty over time to ensure that our final solution is feasible. Another common approach involves applying a simple algorithm to “repair” an infeasible solution any time it is generated. For example, to pare down a set of nodes \( S \) so it becomes an independent set, we might repeatedly remove a node from \( S \) with a maximum number of neighbors in \( S \), proceeding greedily in this fashion until no two nodes in \( S \) are adjacent.

**Problem 242 (Iterative Refinement for Parallel Machine Scheduling).** In most instances of iterative refinement, it is hard to prove strong bounds on solution quality and time until convergence. For example, in Section ?? we discuss a 3/2-approximation algorithm for the traveling salesman problem in the common special case where its edge costs satisfy the triangle inequality. However, if we try solving this problem with iterative refinement using the standard 2-edge-swap neighborhood shown in Figure 13.1(a), it can take an exponential number of steps to converge to a local optimum, which in turn can only be guaranteed to be within a factor of \( \Theta(\sqrt{n}) \) of optimal (in practice, of course, one would expect much better performance). There are some problems for which reasonable bounds can be shown, however. One nice example appears in Section ??, where we use local search to approximate the metric \( k \)-medians problem. Here, we show another good example: the NP-hard minimum-makespan scheduling problem, defined in problem 204. Starting with an arbitrary assignment of jobs to machines, suppose we repeatedly move from one assignment to another by taking a largest job on a most-loaded machine and moving it to a least-loaded machine, as long as this improves the makespan.

(a) Like problem 204, show that this gives a 2-approximation at termination. [Solution]
(b) Please argue that it takes only \( O(mn) \) steps to reach a local minimum. As a hint, if job \( j \) is moved from machine \( a \) to machine \( b \), then try to show that no job of the same size or larger will ever be moved from \( b \) to \( a \). [Solution]
13.1. ITERATIVE REFINEMENT OF A SINGLE SOLUTION

Figure 13.2: A fictitious but prototypical example of the type of decomposition given by nonnegative matrix factorization applied to a term-document matrix \( A \), where \( A_{ij} \) counts occurrences of term \( i \) in document \( j \). The factorization \( A \approx XY \) provides insight into how our documents can be described using combinations of a small number of underlying latent features, which here have the interpretation of subject-specific vocabularies. These features appear as output in the columns of \( X \), while the columns in \( Y \) tell us how to express each document approximately as a nonnegative-weighted combination of these features. In this case, the document we have highlighted — ostensibly a report on how cardiac genomics data can be stored in a splay tree — can be described well by a combination of the term frequency vectors for feature 1 ("biology") and feature 4 ("computer science").

13.1.2 Simulated Annealing

Annealing is the process of gradually cooling down a hot substance like molten metal so that it can be smoothly formed into some desired shape. If cooling happens too quickly, it might end up hardening into an undesired irregular conformation, much like a local search algorithm getting stuck at a local minimum. Simulated annealing mimics this idea in the context of optimization. We start at a high “temperature”, where we are allowed to move fluidly among neighboring solutions, even those that are worse. As we begin to cool down, however, it becomes increasingly likely that we move only towards better solutions.

A standard simulated annealing algorithm generates random neighbors \( x' \) of our current solution \( x \). Any time it finds a better neighbor \( x' \), it moves there. Otherwise, it moves to \( x' \) with probability like \( e^{-\Delta/T} \), where \( \Delta \) measures the degradation in objective value between \( x \) and \( x' \), and \( T \) is a measure of temperature. As we decrease \( T \) according to some predetermined “cooling schedule”, it becomes increasingly likely that each iteration will move in an improving direction, although there is still some probabilistic fluidity allowing us to escape from local minima by chance.

Simulated annealing smoothly interpolates between two different algorithms, starting with a random walk through the space of all solutions\(^3\), and transitioning to a pure local search algorithm. It is also has a nice biological interpretation, where each generation in an evolutionary process introduces small random mutations, with probabilistic selection pressure in favor of keeping beneficial changes.

\(^3\)Later, when we discuss the Metropolis-Hastings algorithm, we will see how a random walk like this can be useful in helping us sample a random solution.
CHAPTER 13. HEURISTICS AND EXHAUSTIVE SEARCH

Figure 13.3: In (a), we show the result of $k$-means clustering to divide a set of points into $k = 3$ clusters, with centers indicated; cluster boundaries partition space into Voronoi cells (Section ??). In (b), we have computed the Gaussian mixture model (a weighted sum of $k = 3$ two-dimensional Gaussian distributions) most likely to have generated the cloud of points below it.

13.1.3 Alternating Iterative Refinement

Often, local search methods update only part of a solution in each step. This is particularly common for problems with a “bipartite” structure, where a solution is composed of two main parts, and it is much easier to update just one of these while holding the other fixed.

Nonnegative Matrix Factorization. As an example from machine learning, Figure 13.2 shows a prototypical use of nonnegative matrix factorization, which seeks to decompose an $m \times n$ matrix $A$ into a product $XY$ of two nonnegative matrices $X$ ($m \times k$) and $Y$ ($k \times n$), with $k$ typically small\(^4\). We would like the product $XY$ to be close to $A$, so a reasonable objective is to minimize the sum of squared entries in the error matrix $A - XY$, subject to the constraint that $X$ and $Y$ must be nonnegative. This objective is NP-hard to minimize exactly, due to lack of convexity. However, if we hold $X$ fixed, the problem of solving for just $Y$ becomes convex and quite tractable, and vice versa when solving for $X$ with $Y$ fixed (these problems decompose into independent instances of nonnegative least squares problems, which we show how to solve in the next chapter). By alternating between updates to $X$ and $Y$, we can converge to a locally-optimal solution efficiently.

Lloyd’s Heuristic for $k$-Means Clustering. A nice example from the realm of data clustering is the popular $k$-means clustering objective, asking us to partition a set of points into $k$ clusters, assigning each cluster a “center” point so we minimize

\(^4\)Later in Chapter ??, we will see other similar methods like principal component analysis (PCA) that also give low-rank approximation of the same form $A \approx XY$. However, since entries of $X$ and $Y$ might be negative, this sort of decomposition is often harder to interpret in terms of its underlying intuitive “meaning” than the one given by nonnegative matrix factorization.
the sum of squared distances from points to their respective cluster centers. An example is shown in Figure 13.3(a). This objective is NP-hard to minimize exactly for general $k$, even in just two dimensions. Nonetheless, a simple alternating iterative refinement algorithm, often known as Lloyd’s heuristic, usually performs quite well in practice. It is based on two simple observations:

- Given a fixed set of locations for the $k$ cluster centers, it is easy to partition the points into clusters to minimize the $k$-means objective. To do so, we just assign each point to the nearest center.

- Given a fixed partition of points into $k$ clusters, it is easy to compute a center for each cluster to minimize the $k$-means objective. Within each cluster (say, containing points $x_1, \ldots, x_n$), this boils down to finding a center $c$ minimizing $f(c) = \sum_i ||c - x_i||^2$, which is easily solved by setting $c = \frac{1}{n} \sum_i x_i$ to the mean of the points in the cluster (this is how $k$-means gets its name).

To run Lloyd’s heuristic, we start with a random set of cluster centers and repeatedly apply these two updates in alternation until convergence is achieved.

**Mixture Modeling and Expectation Maximization.** Despite its popularity, $k$-means clustering has notable shortcomings. It is not adept at dealing with outliers (e.g., a single point far away might end up in a cluster all by itself), and can be confused by non-circular clusters or clusters of non-uniform size. A more robust approach is to use Gaussian mixture modeling, shown in Figure 13.3(b), where we treat our points as samples generated by a probability distribution composed of a weighted combination of $k$ Gaussian distributions. Beyond just clustering, this is also useful for approximating the likely shape of a “well behaved” unknown continuous distribution given only a set of discrete samples from it, allowing us to build a much smaller and simpler model describing a large set of points.

Backing up a bit, a Gaussian distribution in $d$ dimensions is specified by a length-$d$ vector $\mu$ giving its mean (center location), and a $d \times d$ standard deviation matrix $\Sigma$ describing its spread and shape (e.g., whether it is highly concentrated around its mean, and whether it is circular versus elliptical). The formula for a Gaussian distribution with parameters $\mu$ and $\Sigma$ is

$$G(x \mid \mu, \Sigma) = \frac{1}{\sqrt{(2\pi)^d |\Sigma|}} \exp\left(-\frac{1}{2}(x - \mu)^T \Sigma^{-1} (x - \mu)\right),$$

where $|\Sigma|$ denotes the determinant of $\Sigma$, and the somewhat intimidating leading factor simply ensures that its integral is one across all space — an important feature for any proper probability distribution.

It is easy to fit the parameters of a Gaussian to maximize the probability that it generates a given set of points $x_1, \ldots, x_n$ by setting $\mu = \frac{1}{n} \sum x_i$ to the mean and $\Sigma = X^T X$ to the covariance matrix of our data (here, $X$ is the $n \times d$ matrix

---

5 There are several other nice interpretations of this objective, which we discuss in Chapter ?? when we study clustering more extensively.

6 One easy way to see this is by taking the gradient of $f$, $\nabla f(c) = 2 \sum_i (c - x_i)$, and setting it equal to zero.

7 We often set the parameters of a statistical model (like a Gaussian) to maximize the probability that it outputs a set of observed data. This is known as maximum likelihood estimation.
whose $i$th column is $x_i - \mu_i$). If we believe a single Gaussian is sufficient to model our data, we are therefore done. Typically, however, our data comes from a more complicated distribution that is better modeled as a mixture of $k > 1$ Gaussians, with an aggregate probability distribution function of the form

$$f(x) = \sum_{i=1}^{k} w_i G(x | \mu_i, \Sigma_i),$$

where the $\mu_i$’s and $\Sigma_i$’s are parameters of the $i$th Gaussian, and $w_1 \ldots w_k$ are non-negative weights summing to one. The weight $w_i$ is the probability that a randomly-generated point from the aggregate distribution comes from the $i$th Gaussian.

We can now learn the parameters ($\mu_i$’s, $\Sigma_i$’s, and $w_i$’s) in the mixture model above using the same high-level alternating iterative refinement approach as with Lloyd’s algorithm. Given the model parameters, we first compute for each input point $x_i$ the probability distribution $p_i = (p_{i1} \ldots p_{ik})$, where $p_{ij}$ is the probability that $x_i$ was generated by the $j$th Gaussian. Given the $p_{ij}$’s, we then update the model parameters. Every such pair of alternating updates increases the probability that our input points were generated by the model. [Further details]

After convergence, if we think of each Gaussian as a “cluster”, then data point $x_i$ belongs not to one single cluster, but a distribution $p_i$ over clusters. This is sometimes known as a “fuzzy” clustering, and can be more informative in the event that a point rightfully sits on the boundary of several clusters.

One of the steps in alternating iterative refinement often involves updating latent (hidden) variables that are not explicit model parameters (the $p_i$’s, in this example). This special case of alternating iterative refinement is commonly known in the field of machine learning as the technique of expectation maximization\footnote{Like “dynamic programming”, the name “expectation maximization” does little to help convey the nature of the technique, which is usually described as alternation between an “expectation” step that updates latent variables, and a “maximization” step that updates the remaining explicit model parameters.}. We can use this approach to learn the parameters in any mixture model, even though mixtures of Gaussians are perhaps the most common due to the mathematical convenience.

---

**Figure 13.4:** Alternating projections.
13.1. Iterative Refinement of a Single Solution

of the Gaussian distribution. Another prominent application involves learning the parameters of a *Hidden Markov Model*, which we discuss in Chapter ??.

**Alternating Projections.** Given two convex sets \( S_1 \) and \( S_2 \), suppose we want to find a point in their intersection \( S_1 \cap S_2 \). Using iterative refinement, we can do this very easily by starting at an arbitrary point and alternatively projecting onto \( S_1 \) and \( S_2 \), as shown in Figure 13.4(a). Projection of \( x \) onto a convex set \( S \) involves finding a point \( x' \in S \) minimizing the distance \( ||x - x'|| \). As we will see in the next chapter, minimization of a convex objective function like this over a convex set is usually considered to be within the realm of “easy” optimization problems. Each step decreases our distance to \( S_1 \cap S_2 \), ensuring convergence. [Simple proof]

The same alternating projection approach can find a point in the intersection of several convex sets, if we cycle through projecting onto each one. Among other applications, this gives a particularly simple way to solve a linear system, by cycling through projecting onto each linear constraint in the system (a line in the 2D plane in the example shown in Figure 13.4(b), or more generally a hyperplane). However, although this method may look attractive due to the simplicity of projection onto a hyperplane, slow convergence can result from a “poorly conditioned” system with nearly-parallel constraints, as in Figure 13.4(c).

An alternative, more popular alternating iterative refinement method for solving linear systems is to project along one coordinate at a time. When we reach hyperplane \( i \), we use it to solve for a new value of \( x_i \), holding all other variables fixed. In the 2D example shown in Figure 13.4(d), we alternate between using line 1 to update the variable \( x_1 \) and using line 2 to update the variable \( x_2 \). This technique is called *Jacobi relaxation*, and in Chapter ?? we will study it and other relatives in greater detail, looking in particular at what conditions are necessary for convergence. In many cases, the method converges quite rapidly, but unlike the alternating projection approach above, it can also fail to converge if we are not careful. For example, if we reverse the roles of lines 1 and 2 in our example, we will follow the same trajectory but spiraling outward rather than inward!

**Fixed Point Interpretations.** You will often find iterative refinement methods (both alternating and not) described in the style of fixed point approaches, where we produce a series of solutions \( x_0, x_1, x_2 \), and so on by repeated application of some function \( f \), so \( x_k = f(x_{k-1}) \). In this case, convergence happens when we approach a fixed point of \( f \), where \( x = f(x) \). For example, back in Section 1.4.5 we showed that \( f(x) = x/2 + 1/x \) leads us to a fixed point at \( x = \sqrt{2} \). In our more recent example of nonnegative matrix factorization, our goal equation \( A = XY \) is a fixed point for a simple and widely-used alternating update rule where we scale each entry \( X_{ij} \) by \( (AY)^T_{ij}/(XYY^T)_{ij} \) and then scale each entry \( Y_{ij} \) by \( (X^TA)_{ij}/(X^TXY)_{ij} \). Many iterative techniques in optimization have an alternative, often equivalent fixed point interpretation. It is always a good thing to have different viewpoints like this available in our toolbox, since they can often lend additional insight and lead to simpler exposition or analysis.

13.1.4 Multiscale Methods

In the previous chapter, we showed many examples of the “relax and round” method for designing approximation algorithms. Since this often generates a solution that
is nearly optimal, it can also serve as an excellent starting point for an iterative refinement heuristic — an approach we might call “relax, round, and refine”.

In this setting, a common method of relaxation is to coarsen our input, building a new instance that is structurally similar to the original but smaller, and hence easier to solve. After solving the coarse-grained instance, we “project” its solution back onto the original instance and then refine, hoping that only minimal refinement will be necessary. Sometimes, we do this for several levels, coarsening multiple times until we reach an instance small enough to solve efficiently, then repeatedly projecting back and refining until we finally obtain a good solution to the original instance. Examples of common types of these so-called multiscale methods are shown in Figure 13.5:

- In (a), we want to find the boundary between foreground and background in a picture (the author’s cat, in this example). This is an image segmentation problem, a fundamental problem in the field of machine vision, for which we know several effective algorithmic solutions. Here, we use an active contour

---

9Use of the word “relax” is appropriate here since we are creating a problem similar to the original but easier. However, since we are not simply relaxing constraints, in this case we do not necessarily have the property — common to many other relaxations — that the solution of the relaxed problem is no worse than that of the original in terms of objective value.

10See also problem ?? and Section ??.
method, which iteratively refines the shape of a closed curve as it slowly contracts inward like a rubber band, using an objective function that favors (1) smooth curvature, and (2) adherence to edges in the image between light and dark pixels. After a number of iterations, the contour in (a) has not quite converged fully. However, we do reach convergence if we run the same number of iterations in (b) on the image downsampled by a factor of two. We can therefore save time by first segmenting a coarsened image, then projecting the resulting contour back to our full-resolution image for minor refinement. Segmentation sometimes even gives a higher-quality solution on a coarsened image than on the original, since noise and other less relevant small-scale features are smoothed away.

• In (c), we use a multigrid method to find the numerical solution to a system of equations across a discretized grid, often modeling some physical phenomenon like heat flow or mechanical stress propagation. It can be expensive to solve the problem outright on a fine-grained grid, and even worse, we might not achieve proper convergence if we start with an arbitrary (say, uniform) solution too far away from the correct solution. To remedy this, we first solve the problem on a coarse grid and then interpolate the resulting solution onto the fine grid. Further refinement on the fine grid is therefore much more likely to converge, and to converge quickly. We sometimes use the same approach across several levels of discretization, or use a more sophisticated adaptive method that coarsens only parts of the grid deemed necessary (e.g., see Figure 8.14(d) and its accompanying discussion).

• Multiscale techniques can be applied successfully to several problems on graphs, such as clustering and linear arrangement problems. These require a procedure that can “coarsen” a graph while preserving its structural properties, of which several have been studied. In (d), we have done this by greedily finding a matching among pairs of nodes that have low algebraic distance (discussed in detail later in Chapter ??). We compute for each node $i$ a short vector $v_i$ such that the distance $||v_i - v_j||$ is likely small between a pair of nodes that are similar in the context of the graph. To build our matching, we scan the edges in increasing order of distance, trying to add each one to the matching unless one or both endpoints are already matched. We then contract pairs of matched nodes into “supernodes”, giving a new graph with roughly half the nodes that behaves structurally like the original. In the coarsened graph, some nodes and edges now have higher “weight”, since they represent aggregations of nodes and edges in the original graph.

Multiscale methods provide the best currently-known heuristic solutions for a number of prominent problems in practice.

13.2 Refining a Population of Solutions

By iteratively refining a population of solutions, not only can individual solutions improve in an independent fashion as before, but we open up a realm of possibilities for solutions to improve by interacting with each-other.
13.2.1 Combining Solutions

A fundamental operation shared by most population-based methods is combining two solutions $x$ and $x'$ to produce a new solution inheriting characteristics of both. If $x'$ is the better solution, one interpretation of this operation is that it tries to improve $x$ by moving to a new solution in the direction of $x'$. Another quite common interpretation draws inspiration from evolutionary biology: by “breeding together” $x$ and $x'$, we hope produce “children” that are genetically superior to both parents (inheriting only the best traits from both), and that are also substantially different from both parents, far beyond what neighborhood search around $x$ or $x'$ would be capable of producing.

The most effective way to combine two solutions depends on both the problem at hand and our solution representation\textsuperscript{11}. For points $x$ and $x'$ in Euclidean space, the most natural approach is to use a convex combination $\lambda x + (1 - \lambda)x'$, as shown in Figure 13.6(a). By our choice of $\lambda \in [0, 1]$, we can adjust the level of contribution of $x$ versus $x'$. For solutions encoded using binary vectors, we might use one-point or two-point crossover, shown in Figure 13.6(b)-(c), where we switch from one solution to the other at one or two randomly-chosen indices. This process mimics crossover between two parent DNA strands in biology, and as a result it is quite commonly used in genetic algorithms (discussed shortly). In fact, we often use the term “crossover” to describe any operation that mixes two solutions. A less common (and anecdotally, typically less effective) option is shown in Figure 13.6(d), where a coin flip determines whether each bit comes from $x$ or $x'$.

If our solutions are represented by more complicated objects (e.g., trees, geometric shapes), we may need to be somewhat more creative in defining how we mix them together. For example, with the traveling salesman problem and other order-based optimization problems, our solution is a permutation of $n$ elements. Several reason-

---

\textsuperscript{11}It is worth stressing the importance of solution representation in all forms of optimization. For many problems, we can represent solutions several different ways — for example, a subset of elements $S$ could be represented as a list of all the elements in $S$, or alternatively as a binary incidence vector $x$ with $x_i = 1$ for all elements $i \in S$. Our choice of representation can have a tremendous impact on the design and overall efficiency of heuristics and other methods.
able crossover operations on permutations have been proposed. Perhaps the most common is shown in Figure 13.6(e), where we take part of one parent solution up to a random index, then take the remaining elements in the order they occur in the other parent solution.

### 13.2.2 Genetic Algorithms and Other Bio-Inspired Methods

Biological evolution is nature’s heuristic for optimizing the quality, or “fitness” of organisms\(^\text{12}\). Inspired by this process, a genetic algorithm simulates the evolution of a population of solutions as it improves over several generations. Each generation contains a new set of solutions produced by the following operations:

- **Mutation**, where we apply a small random change (or perhaps one step of a traditional local search algorithm) to a solution from the previous generation.
- **Crossover**, where we combine two solutions from the previous generation.
- **Immigration**, where we introduce a completely new solution, often generated randomly or with some other simple heuristic.
- **Elitism**, where the best few solutions are directly copied from the previous generation to ensure they are not lost.

With mutation and crossover, we choose solutions from the previous generation with probability proportional to their fitness (e.g., see problem 114), so better solutions are more likely to pass along their genetic material to future generations, just as with biological evolution.

A poorly-tuned genetic algorithm can perform quite badly, since it is searching in a very primitive way that makes no effort to exploit any special structure our problem might have. However, with sufficient effort and careful tuning, genetic algorithms can actually be made to work surprisingly well. Tuning the rate of mutation/immigration versus crossover is particularly important. With too much crossover, a population might lose its diversity. For example, if you start with \(n\) points in space and repeatedly replace points with convex combinations of other points, this will eventually collapse to a single point. The only way to counteract this inward pressure from crossover is using mutation and immigration. Of course, without sufficient crossover, a genetic algorithm loses its only functional advantage beyond what local search can achieve by itself.

Researchers have studied many variations on the basic genetic algorithm template, as well as a large number of related approaches that are also biologically-inspired. After a thorough review of the literature, a recent undergraduate research project directed by the author turned up nearly twenty organisms after which different optimization heuristics have been modeled. For example, instead of selecting two random solutions (biased by fitness) as parents for crossover, we might choose one to be the best solution from the prior generation, modeling the way breeding in a

\(^{12}\)As a secondary objective, evolution also seems to optimize for “genetic tolerance” — an individual’s ability to produce satisfactory offspring when subject to crossover with a large range of other partners. A reasonably “fit” individual with good reproductive potential is generally better than a very “fit” individual that cannot spawn viable offspring.
wolf pack tends to favor “alpha” individuals (pack leaders). To improve diversity, we might force our population to split into sub-populations that evolve separately and only occasionally mix, as might happen due to geological changes in habitats. We might boost fitness for solutions highly distinct from all others to encourage exploration of new regions in solution space, the way a root system in a plant spreads out to optimize its volumetric coverage. Many other ideas are possible.

**Social Intelligence.** A theme in many bio-inspired population refinement methods is the exploitation of the “social intelligence” of the population as a whole. For example, a solution might pick \( k \) other solutions to consult, using crossover to move in the direction of the best of these — somewhat like a bee receives information from its peers upon their return to the hive. The resulting spread of information through the population helps coordinate its evolution, with each individual solution driven by a mixture of local pressure (as with any other neighborhood search algorithm) and also global influence due to emergent knowledge of the collective population. This generic idea is sometimes called particle swarm optimization or swarm intelligence, and many of its variants have been studied in the heuristics literature. The most prominent example is probably ant colony optimization, which models the way ants leave behind trails of chemicals called pheromones that attract other ants. As more and more ants find their way to a food source, they build an increasingly-stronger trail that attracts other ants, while trails that lead nowhere useful eventually evaporate. We might mimic this behavior by augmenting the space we are searching with “beacons” that attract other solutions but decay in strength over time\(^\text{13}\), thereby adding a form of social “memory” into our heuristic.

### 13.3 Branch and Bound

Sometimes, it is necessary to find the optimal solution of a hard problem exactly. This takes iterative refinement methods off the table, since they inherently tend to find only locally optimal solutions. Instead, we must resort to some form of exhaustive search through the space of all possible solutions. In this section, we discuss how to do this intelligently with a simple method called branch and bound, and in the next section we discuss the related process of constraint satisfaction.

To make our discussion concrete, we adopt the NP-hard minimum-cost node cover problem as a running example. Given an \( n \)-node graph where each node \( i \) has associated cost \( c_i \), we want to find a minimum-cost subset of nodes that covers every edge. We have studied this problem previously in Sections 10.4.2, 12.4, and 12.5.1. As an integer linear program, it can be written as

\[
\text{Minimize: } \sum_{i=1}^{n} c_i x_i \\
\text{Subject to: } x_i + x_j \geq 1 \quad \text{for all edges } ij \\
x_i \in \{0, 1\} \quad \text{for all nodes } i = 1 \ldots n,
\]

where \( x_i = 1 \) indicates that node \( i \) belongs to our solution. There are \( 2^n \) different settings to consider for the decision variables \( x_1 \ldots x_n \), and our job is to enumerate through these quickly, hopefully using far less than \( \Theta(2^n) \) time in practice.

\(^{13}\)By contrast, the tabu search method, mentioned earlier, could be implemented using repulsive beacons, in order to drive the search away from one local minimum towards others.
The branch and bound technique is aptly named. The “branch” part refers to searching an implicit tree of all possible solutions by recursively enumerating all the different ways to set our decision variables. For the “bound” part, we prune the search whenever we reach a partial solution that cannot lead to any improvement over the best solution found so far. In the example in Figure 13.7, we first consider either setting \( x_1 = 0 \) or \( x_1 = 1 \). Both cases lead to a smaller residual subproblem that we solve recursively by branching further on other variables, ultimately forming our subproblem tree. Leaves represent complete solutions, while all other nodes represent partial solutions. If variables are binary-valued, we get a binary tree. If some variable \( x_i \) has a larger domain (say, \( x_i \in \{0, 1, 2, 3, 4\} \)), then higher branching factors are possible. In this case, it would be natural to branch 5 ways corresponding to the 5 possible values of \( x_i \). However, it is still common to branch in a binary fashion — say, giving subproblems where \( x_i \in \{0, 1\} \) and \( x_i \in \{2, 3, 4\} \). The reason is that if we are lucky enough to prune away one of these two aggregate branches, we can discard a large chunk of our search space in one fell swoop.

Our search is usually depth-first, also called backtracking search. We recursively walk down the tree until we reach a complete solution or prune the search, then back up and explore other branches. Each subtree of the root is completely explored before moving on to the next. In a moment, we will see how some problems are better suited for a breadth-first search, an alternative approach that searches in a level-by-level fashion downward from the root. Depth-first and breadth-first search are also very common methods for searching graphs, as we will see in Chapter ??.
Variable and Branch Selection. At each point in the search, we need to select the next variable on which to branch, and the order in which to explore its different possible values — this is worth careful consideration, as it can have a dramatic impact on running time. The most natural strategy here is to be greedy. In our node cover example, we might choose a node \(i\) whose selection would minimize the per-edge cost of any additional edges we cover, and we would recursively explore all solutions reachable by setting \(x_i = 1\) before next trying \(x_i = 0\). The first complete solution produced by our algorithm will therefore be the same as the solution produced by a greedy algorithm, after which we backtrack and continue exploring other prospective solutions. An advantage of this approach is that it tends to find high-quality solutions quickly, leading to more effective pruning.

Many other heuristics for variable and branch selection have been studied. To give another example, we might branch on the variable whose value is the closest to an integer in a fractional relaxation of our problem. For example, if \(x_i = 0.99\) after solving a linear programming relaxation of our problem, we might take this as a hint to try setting \(x_i = 1\) and then \(x_i = 0\).

Problem 243 (Variable Fixing). Suppose every decision variable \(x_i\) in our problem is constrained to take a value in \(\{0, 1\}\). If we solve a fractional linear programming relaxation of our problem and obtain a solution that happens to set \(x_i = 0\) or \(x_i = 1\), this unfortunately does not mean that there exists an optimal integer-valued solution where \(x_i\) is similarly set. However, show that by comparing the objective value of the linear programming relaxation, the objective value of the best solution found so far, and the reduced cost of \(x_i\) (defined in Section 12.2.1), that we can sometimes conclude that it is safe to fix the value of \(x_i\) permanently, speeding up our remaining search. [Solution]

Lower Bound Computation and Tree Pruning. Suppose we have built up a partial solution \(x\) during the search. We can obviously prune the search at this point if \(x\) is infeasible. We can also prune if \(x\) is already worse than the best complete solution found so far, known as the incumbent solution. A slightly more
intelligent form of pruning is shown in Figure 13.8. Assuming our problem involves
minimization, we can prune the search if the cost of \( x \) plus a lower bound on the
cost of its completion exceeds the cost of the incumbent solution. Assuming we
have a good method for lowering bounding the cost of completing \( x \), this method can
prune solutions very early in the search, for significant savings in running time.

There are several common ways to obtain lower bounds on the cost of completing
a solution. Some are computationally intensive, possibly slowing down the search,
but this extra effort is often well-spent if it leads to substantially better pruning.

- We might use just a simple formula. For our node cover example, if \( c \) is the
minimum cost of a node, \( \Delta \) is the maximum node degree, and \( u \) is the number
of yet-uncovered edges, then our solution needs at least \( u/\Delta \) additional nodes
to be selected, so its completion cost is at least \( cu/\Delta \). Even rather crude
bounds like this can make a surprising difference in performance, and they
have the advantage of being very quick to evaluate. Another common example
from geometric path-finding problems (e.g., finding the shortest path through
a geometric scene with obstacles) is the straight line distance to the goal point.

- We could exploit duality, using the objective value of any feasible dual solution
as a lower bound. The dual of a minimum-cardinality node cover problem
(where every node has unit cost) is a maximum-cardinality matching problem,
so we can use the size of a greedily-computed maximal matching (Section
10.4.2) for a lower bound. In the more general case where nodes have non-
unit costs, the dual of our node cover problem is a type of edge packing
problem, for which a fast greedy solution is still easy to obtain [Details]. See
also Section ?? in the next chapter on how to use Lagrangian relaxation to
obtain lower bounds via “Lagrangian” duality.

- We could use the objective value of a relaxation of our problem. For linear
programming relaxations, note that each step through our subproblem tree
involves adding or removing a single constraint (e.g., \( x_1 = 0 \)). In this situation,
the dual simplex algorithm (Section 12.5.2) can usually re-optimize our
solution quickly, instead of solving an entire linear program from scratch.

Problem 244 (Fast Branch and Bound Approaches for Knapsack). Please
describe a means of performing branch and bound to find an optimal solution to the knap-
sack problem (Section 11.2), using appropriate data structures so that you can check for
each node in the subproblem tree if it can be pruned away in only \( O(\log n) \) time. [Solution]

Problem Simplification and Preprocessing. To get as much of a head start
as possible, we can often employ variable fixing (problem 243) or problem-specific
observations to permanently set as many variables as possible during preprocessing.
In our node cover example, suppose some node \( i \) has only one incident edge, and that
this edge connects \( i \) to a node \( j \) of no higher cost. In this case, we can always find
an optimal solution with \( x_i = 0 \) and \( x_j = 1 \), allowing us to permanently set these
variables and also remove their incident edges from the problem. If this rule ever
becomes applicable during the ensuing search process, this tells us which variables
we should set next as we continue searching.
13.3.1 Dynamic Programming and $A^*$ Search

Branch and bound and “top down” dynamic programming are actually quite similar techniques. Both involve recursive search of a tree of subproblems that represents the sequential construction of a full solution. The main difference is how pruning happens. Let’s use the phrase “lower bound pruning” for the branch and bound pruning method described above. By contrast, dynamic programming uses “pruning by memoization”, storing the solution of every distinct subproblem in a table, and pruning the search any time it comes across a subproblem that has already been solved.

Both methods of pruning can be used simultaneously, but this is only sensible under the right circumstances. When solving a hard problem like minimum-cost node cover, the number of distinct subproblems in our search tree is usually so large that storing their solutions in a table would take excessive space. We therefore rarely add pruning by memoization to an existing branch and bound search in this setting. For easier problems with fewer distinct subproblems (i.e., problems already well-suited for dynamic programming), we can sometimes speed up dynamic programming by also using lower bound pruning, a technique known as $A^*$ search. However, while extra pruning has tremendous impact on an exponential-scale branch and bound search, it usually buys us less with smaller-scale problems like those we approach with dynamic programming — particularly when we take into account the extra cost associated with lower bound computation. Empirical testing, as always, is advised. When we study shortest paths in graphs in Chapter ??, we will see that this method has much in common with a version of Dijkstra’s famous shortest path algorithm when used with modified “reduced” edge costs.

13.3.2 Alpha-Beta Search

Suppose we want to build a Chess engine, using a board evaluation function $f$ that returns a large value if the chess board is in a favorable state. We would like to plan a sequence of moves so that we end up in a state maximizing $f$, but this is complicated by the presence of an opponent who makes moves interleaved with ours, and whose goal is to minimize $f$. Similar to branch and bound, we can model the sequence of moves for both players with a game tree. The root branches according to all possible moves for player 1, the next level branches according to responses for player 2, and so on in alternation. Since our objective alternates between maximization and minimization for even and odd levels in the tree, pruning our search becomes slightly more nuanced, and we often employ a popular pruning technique designed specifically for this situation known as alpha-beta search. [Details]

13.3.3 Breadth-First Search and Iterative Deepening

Game trees are one good example involving searching a tree of potentially infinite depth. For example, an infinite path down the tree might correspond to a stalemate where two players move the same pieces back and forth forever. In this and other situations where we are searching a tree involving decisions that can cycle endlessly or a tree whose depth is excessively large, depth-first search becomes inadvisable.
Instead, we may want to search in a breadth-first manner, examining all nodes that are one step away from the root, then all nodes that are two steps away from the root, and so on, stopping when we reach some maximum prescribed depth. At this point, we may not have reached the bottom of the tree and found a complete solution, but by searching several “moves” ahead, we may learn enough to help make our next few decisions in a more informed fashion (at which point we may repeat the search to further extend our planning horizon).

A serious drawback of breadth-first search is space, since we need to store all the subproblems in the current level as we generate the next level, and the number of subproblems per level usually grows at an exponential rate. We therefore often simulate breadth-first search using depth-first search with iterative deepening, where we run a depth-first search down to at most a depth of one, then we run another depth-first search down to at most a depth of two, and so on. In a tree with branching factor $b \geq 2$ and depth $d$, this approach uses only $\Theta(d)$ memory, compared to the $\Theta(b^d)$ memory needed by breadth-first search. Moreover, iterative deepening only inflates our running time by a small constant factor. Assuming for simplicity that it takes $b^d$ units of time to search down to level $d$ using depth-first or breadth-first search, the running time for iterative deepening down to level $d$ would be

$$1 + b + b^2 + b^3 + \ldots + b^d = \frac{b^{d+1} - 1}{b - 1} = \Theta(b^d),$$

thanks to the fact that an increasing geometric series behaves asymptotically just like its final term.

### 13.3.4 Adding Constraints with Cutting Planes

Consider a problem formulated as a linear program with integrality constraints on its variables. As shown in Figure 13.9, the feasible region of its linear programming relaxation is a convex polyhedron. Feasible solutions for the original integer linear program are lattice points within this polyhedron. If we solve the relaxation and
are lucky enough to get one of these lattice points as our answer, we will have found the optimal solution to the original integer problem. However, even if we end up with a fractional solution, we can still make progress by adding a new constraint called a cutting plane. This constraint is designed to “tighten” the relaxation by cutting off our optimal fractional solution from the feasible set while preserving feasibility of all integer points. The dual simplex algorithm can then be used to re-solve the relaxation, since it is adept at re-optimization after the addition of new constraints. If our optimal solution is another fractional point, we add another cutting plane, continuing until we finally reach an integer solution. 

A standard method for generating cutting planes

Although this approach can work well for solving integer linear programs, it often ends up in a state where each successive cutting plane makes very little improvement (the first few may help a lot, but over time they tend to help less and less). In this case, we may gain by combining cutting planes with branch and bound. One common way to do this, cut and branch, involves adding some number of cutting planes as a preprocessing step to tighten our relaxation, and then to run branch and bound. This can help speed up the branch and bound traversal significantly, since the lower bounds we compute by linear programming relaxation will be much better, allowing for more effective pruning. Another popular approach is to periodically add cutting planes during the branch and bound process, a method whose variants are known as branch and cut. Many sophisticated general-purpose integer linear programming solvers on the market today employ some variant of this technique.

### 13.4 Constraint Satisfaction

In a constraint satisfaction problem, each decision variable has an associated domain of values it can possibly take, and we are given a set of constraints that restrict the feasible values we can assign to certain subsets of variables (often just pairs of variables). A prototypical example is graph coloring, shown in Figure 13.10(a), where the goal is to assign each node a numeric “color” (in this case, from the set \{1, 2, 3\}) so that no two adjacent nodes are given the same color. Graph coloring problems have many useful applications, and we study them in further detail in Chapter ??.

We often try to use the fewest total colors; given an constraint satisfaction algorithm for producing a feasible assignment of colors to nodes, we could wrap this in a binary search to determine the minimum number of colors necessary.

In their full generality, constraint satisfaction problems can involve continuous as well as discrete variables, and they can ask for an optimal solution according to some objective function, instead of just a feasible solution. Since this characterization is quite broad, we focus our attention here on perhaps the most typical scenario in practice, where we search for a single feasible solution to problem with discrete variables, like the graph coloring problem above. In fact, graph coloring is a good generic model for almost any constraint satisfaction problem of this type, if we phrase the problem in terms of its underlying constraint graph. The nodes of the constraint graph correspond to decision variables, each with an associated discrete domain of values (e.g., colors), and the edges represent constraints that restrict the possible settings for pairs of decision variables. If we happen to have a problem with constraints governing sets of more than two variables, then our constraint
13.4. CONSTRAINT SATISFACTION

Figure 13.10: Examples of constraint satisfaction problems: (a) graph coloring, (b) labeling edges from a 2D wireframe image of a 3D object to deduce their role in 3D, (c) the \( n \)-queens problem in Chess, and (d) the popular Sudoku problem. The constraint-restricted domains of some of the decision variables above are shown as of the current state of the search.

The constraint graph becomes a hypergraph, where “edges” correspond to subsets of nodes instead of pairs of nodes (do not worry — this generalization does not particularly affect the viability of the algorithms we will discuss shortly).

Examples of constraint satisfaction problems are shown in Figure 13.10. Figure 13.10(b) highlights an application in machine vision, where we wish to label the edges in a 2D wireframe image to deduce their role in 3D. An arrow label indicates an edge on the boundary of the three-dimensional shape, with empty space to the left side of the arrow and the solid shape to the right. Edges labeled either plus or minus denote interior edges that respectively fold in a mountain or valley configuration. At each junction, only certain combinations of edge labels are feasible. For example, once the two arrow labels are applied to the circled junction, the only possible feasible setting for the third edge is plus. Note that this example involves non-binary constraints, since some junctions constrain the values of up to three decision variables (edge labels). Figure 13.10(c) shows the famous \( n \)-queens Chess problem, involving placement of \( n \) mutually non-attacking queens on an \( n \times n \) board. We have a decision variable for every row with domain \( \{1, 2, \ldots, n\} \) telling us the location of the queen within that row. The placement of a queen in any row restricts the placement of queens in all other rows, so our constraint graph has all pairs of variables connected by binary constraints. Figure 13.10(d) shows a Sudoku puzzle,
another recreational example. Each empty square corresponds to a decision variable, and it is connected via an edge in the constraint graph to every other empty square in the same row, column, and $3 \times 3$ block.

The process of solving a constraint satisfaction problem involves an exhaustive recursive search through tree of subproblems constructed by setting one variable at a time, exactly like branch and bound. In fact, we might want to regard this process as a variant of branch and bound where the challenge lies in just finding a feasible solution, instead of an optimal solution. The tree pruning aspect of branch and bound carries over as well, since as we set variables during our search, we can use the constraints in our problem to restrict the values that other variables can possibly take, allowing us to prune away branches from the search any time we realize that some variable has no more valid settings.

As with branch and bound, our approach for variable and branch selection can have a dramatic impact on the overall search time. Typically, we choose the next variable on which to branch to be the one constrained to have the smallest number of feasible values, breaking ties by choosing the variable involved in the largest number of constraints. The intuition here is that setting such a variable will allow us to highly constrain and therefore prune the rest of the search as quickly as possible. If any variable $x_i$ is forced to take just a single value, then we only stand to gain from setting that variable next, since this could help us further constrain the rest of our search. When choosing the order of values to try for each variable, we often try the “least-constraining” values first (those values that leave as many possibilities open for other remaining variables as possible), thereby maximizing our chances of success in finding a valid setting for the remaining variables.

One of the key challenges with constraint satisfaction is knowing how far forward to propagate the ramifications of each variable setting through our constraint graph. On one extreme, we do not propagate any information, setting each variable we encounter according to the valid settings it can take based on the variables we have already tentatively set. Whenever we encounter some variable with no valid settings, we prune the search and backtrack. On the other extreme, we maintain a set $S_i$ for every variable $x_i$ containing all its feasible settings, based on the variables we have already set during our search. Each time we try a setting of $x_i$, we propagate the ramifications of this setting through the constraint graph, removing entries that now become invalid from the domains $S_j$ for all other variables $x_j$ directly or indirectly dependent on $x_i$ (do not forget that when we backtrack or try a different setting for $x_i$, we need to carefully undo these changes). This makes each step of our search more expensive, since it could result in a propagation of knowledge across the entire constraint graph. However, it can also help us prune the search more quickly, since we can realize further in advance whenever some set $S_j$ becomes empty. The most successful balance between “search” and “forward propagation” might be somewhere in the middle. For instance, we might only propagate the ramifications of setting $x_i$ to its immediate neighbors in the constraint graph, since this might allow us to prune our search effectively based on “local” constraints without wasting time propagating information globally across the entire solution at every step of our search. As you might expect, empirical testing is generally the only way to determine, on a problem-by-problem basis, the best balance between search and forward propagation.