Part A: Divide and Conquer; Sorting and Searching

Wayne Goddard, School of Computing, Clemson University, 2019

Convex Hulls: An Example 1
A1.1 Graham Scan ................................. 1
A1.2 Jarvis March ................................. 2
A1.3 Divide and Conquer ......................... 3

Order Analysis 6
A2.1 Algorithm Analysis ......................... 6
A2.2 Order Notation .............................. 6
A2.3 Combining Functions ....................... 7

Divide and Conquer 8
A3.1 Master Theorem for Recurrences .......... 8
A3.2 Quicksort .................................. 8
A3.3 How Fast is Quicksort? ..................... 9
A3.4 Merge Sort ................................ 10
A3.5 Optimality of Sorting ..................... 11
A3.6 Good Algorithms ........................... 11

More Divide and Conquer 13
A4.1 Multiplying Long Integers ............... 13
A4.2 Matrix Multiplication ...................... 14
A4.3 Closest Pair ................................ 15

Finding the Median 17
A5.1 A Generalization ........................... 17
A5.2 Using Median for Rank Selection ........ 17
A5.3 The Pseudo-Median ....................... 18
A5.4 Finding a Pseudo-Median ............... 19
A5.5 Analysis .................................. 20
Chapter A1: Convex Hulls: An Example

A polygon is **convex** if any line segment joining two points on the boundary stays within the polygon. Equivalently, if you walk around the boundary of the polygon in counterclockwise direction you always take left turns.

The **convex hull** of a set of points in the plane is the smallest convex polygon for which each point is either on the boundary or in the interior of the polygon. One might think of the points as being nails sticking out of a wooden board: then the convex hull is the shape formed by a tight rubber band that surrounds all the nails. A **vertex** is a corner of a polygon. For example, the highest, lowest, leftmost and rightmost points are all vertices of the convex hull. Some other characterizations are given in the exercises.

![Diagram of convex hull](image)

We discuss three algorithms for finding a convex hull: Graham Scan, Jarvis March and Divide & Conquer. We present the algorithms under the **assumption** that:

- no 3 points are collinear (on a straight line)

### A1.1 Graham Scan

The idea is to identify one vertex of the convex hull and sort the other points as viewed from that vertex. Then the points are scanned in order.

Let \( x_0 \) be the leftmost point (which is guaranteed to be in the convex hull) and number the remaining points by angle from \( x_0 \) going counterclockwise: \( x_1, x_2, \ldots, x_{n-1} \). Let \( x_n = x_0 \), the chosen point. (We’re assuming no two points have the same angle from \( x_0 \).)

The algorithm is simple to state with a single stack:

### Graham Scan

1. Sort points by angle from \( x_0 \)
2. Push \( x_0 \) and \( x_1 \). Set \( i = 2 \)
3. While \( i \leq n \) do:
   - If \( x_i \) makes left turn w.r.t. top 2 items on stack then \{ push \( x_i \); \( i++ \) \}
   - else \{ pop and discard \}
To prove that the algorithm works, it suffices to argue that:

- **A discarded point is not in the convex hull.** If \( x_j \) is discarded, then for some \( i < j < k \) the points \( x_i \rightarrow x_j \rightarrow x_k \) form a right turn. So, \( x_j \) is inside the triangle \( x_0, x_i, x_k \) and hence is not on the convex hull.

  ![Diagram of convex hull](image)

- **What remains is convex.** This is immediate as every turn is a left turn.

The running time: Each time the while loop is executed, a point is either stacked or discarded. Since a point is looked at only once, the loop is executed at most \( 2n \) times. There is a constant-time subroutine for checking, given three points in order, whether the angle is a left or a right turn (Exercise). This gives an \( O(n) \) time algorithm, apart from the initial sort which takes time \( O(n \log n) \). (Recall that the notation \( O(f(n)) \), pronounced “order \( f(n) \)”, means “asymptotically at most a constant times \( f(n) \).”)

### A1.2 Jarvis March

This is also called the **wrapping algorithm**. This algorithm finds the points on the convex hull **in the order** in which they appear. It is quick if there are only a few points on the convex hull, but slow if there are many.

Let \( x_0 \) be the leftmost point. Let \( x_1 \) be the first point counterclockwise when viewed from \( x_0 \). Then \( x_2 \) is the first point counterclockwise when viewed from \( x_1 \), and so on.

```
Jarvis March
i = 0
while not done do
    \( x_{i+1} = \) first point counterclockwise from \( x_i \)
```

Finding $x_{i+1}$ takes linear time. The while loop is executed at most $n$ times. More specifically, the while loop is executed $h$ times where $h$ is the number of vertices on the convex hull. So Jarvis March takes time $O(nh)$.

The best case is $h = 3$. The worst case is $h = n$, when the points are, for example, arranged on the circumference of a circle.

**A1.3 Divide and Conquer**

Divide and Conquer is a popular technique for algorithm design. We use it here to find the convex hull. The first step is a Divide step, the second step is a Conquer step, and the third step is a Combine step.

The idea is to:

**Divide and conquer**

1. Divide the $n$ points into two halves.
2. Find convex hull of each subset.
3. Combine the two hulls into overall convex hull.

Part 2 is simply two recursive calls. Note that, if a point is in the overall convex hull, then it is in the convex hull of any subset of points that contain it. (Use characterization in exercise.) So the task is: given two convex hulls, find the convex hull of their union.

♦ **Combining two hulls**

It helps to work with convex hulls that do not overlap. To ensure this, all the points are presorted from left to right. So we have a left and right half, and hence a left and right convex hull.

Define a bridge as any line segment joining a vertex on the left and a vertex on the right that does not cross the side of either polygon. What we need are the upper and lower bridges. The following produces the upper bridge.
1. Start with any bridge. For example, a bridge is guaranteed if you join the rightmost vertex on the left to the leftmost vertex on the right.

2. Keeping the left end of the bridge fixed, see if the right end can be raised. That is, look at the next vertex on the right polygon going clockwise, and see whether that would be a (better) bridge. Otherwise, see if the left end can be raised while the right end remains fixed.

3. If made no progress in Step 2 (cannot raise either side), then stop else repeat Step 2.

We need to be sure that one will eventually stop. Is this obvious?

Now, we need to determine the running time of the algorithm. The key is to perform Step 2 in constant time. For this it is sufficient that each vertex has a pointer to the next vertex going clockwise and going counterclockwise. Hence the choice of data structure: we store each hull using a **doubly linked circular linked list**.

It follows that the total work done in a merge is proportional to the number of vertices. And as we shall see from a later chapter, this means that the overall algorithm takes time $O(n \log n)$.

**Exercises**

1. Find the convex hulls for the following list of points using the three algorithms presented.

\[(30, 60)\]

\[(0, 30), (15, 25), (50, 10), (20, 0)\]

\[(5, 40)\]

\[(0, 30), (70, 30), (55, 20), (50, 10)\]
2. Give a quick calculation which tells one whether three points make a left or a right turn.

3. Discuss how one might deal with collinear points in the algorithms.

4. Show that a point $D$ is on the convex hull if and only if there do not exist points $A, B, C$ such that $D$ is inside the triangle formed by $A, B, C$.

5. Give a good algorithm for the convex layers problem. The convex layers are the convex hull, the convex hull of what remains, etc.

6. Assume one has a fast convex hull subroutine that returns the convex hull in order. Show how to use the subroutine to sort numbers.
Chapter A2: Order Analysis

A2.1 Algorithm Analysis

The goal of algorithmic analysis is to determine how the running time behaves as \( n \) gets large. The value \( n \) is usually the size of the structure or the number of elements it has. For example, traversing an array takes time proportional to \( n \) time while a single array access is assumed to take constant time.

We want to measure either time or space requirements of an algorithm. Time is the number of atomic operations executed. We cannot count everything: we just want an estimate. So, depending on the situation, one might count: arithmetic operations (usually assume addition and multiplication atomic, but not for large integer calculations); comparisons; procedure calls; or assignment statements. Ideally, pick one which simple to count but mirrors the true running time.

A2.2 Order Notation

We define big-O:

\[
f(n) \text{ is } \mathcal{O}(g(n)) \text{ if the growth of } f(n) \text{ is at most the growth of } g(n).
\]

So \( 5n \) is \( \mathcal{O}(n^2) \) but \( n^2 \) is not \( \mathcal{O}(5n) \). Note that constants do not matter; saying \( f \) is \( \mathcal{O}(\sqrt{n}) \) is the same thing as saying \( f \) is \( \mathcal{O}(\sqrt{22}n) \).

The order (or growth rate) of a function is the simplest smallest function that it is \( \mathcal{O} \) of. It ignores coefficients and everything except the dominant term.

**Example.** Some would say \( f(n) = 2n^2 + 3n + 1 \) is \( \mathcal{O}(n^3) \) and \( \mathcal{O}(n^2) \). But its order is \( n^2 \).

Terminology: The notation \( \mathcal{O}(1) \) means constant-time. Linear means proportional to \( n \). Quadratic means \( \mathcal{O}(n^2) \). Sublinear means that the ratio \( f(n)/n \) tends to 0 as \( n \to \infty \) (sometimes written \( o(n) \)).

**Example.** Long Arithmetic Long addition of two \( n \)-digit numbers is linear. Long multiplication of two \( n \)-digit numbers is quadratic.

(Check!)
A2.3 Combining Functions

- **ADD.** If $T_1(n)$ is $O(f(n))$ and $T_2(n)$ is $O(g(n))$, then $T_1(n) + T_2(n)$ is max$(O(f(n)), O(g(n)))$.
  That is, when you add, the larger order takes over.

- **MULTIPLY.** If $T_1(n)$ is $O(f(n))$ and $T_2(n)$ is $O(g(n))$, then $T_1(n) \times T_2(n)$ is $O(f(n) \times g(n))$.

**Example.** $(n^4 + n) \times (3n^3 - 5) + 6n^6$ has order $n^7$

For consecutive blocks, the overall running time is their sum and hence the maximum. For loops, the overall running time is how many times the body is executed times the average case of the body. One can get an upper bound for loops by taking an upper bound for these two quantities.

**Example. Primality Testing** The algorithm is

```java
for(int y=2; y<N; y++)
    if( N%y==0 )
        return false;
    return true;
```

This takes $O(\sqrt{N})$ time if the number is not prime, since then the smallest factor is at most $\sqrt{N}$. But if the number is prime, then it takes $O(N)$ time. And, if we write the input as a $B$-bit number, this is $O(2^{B/2})$ time. (Can one do better?)

The **log base 2** of a number is how many times you need to multiply 2 together to get that number. That is, $\log n = L$ when $2^L = n$. Unless otherwise specified, computer science log is always base 2. So it gives the number of bits. The function $\log n$ grows forever, but it grows (much) slower than any power of $n$.

**Example.** Binary search takes $O(\log n)$ time.
Chapter A3: Divide and Conquer

In this chapter we consider divide and conquer: this is essentially a special type of recursion. In divide and conquer, one:

- divides the problem into pieces,
- then conquers the pieces,
- and re-assembles.

An example of this approach is the convex hull algorithm. We divide the problem into two pieces (left and right), conquer each piece (by finding their hulls), and re-assemble (using an efficient merge procedure). Binary search can also be viewed as divide and conquer.

A3.1 Master Theorem for Recurrences

One useful result for analyzing divide-and-conquer algorithms is the “Master Theorem” for a certain family of recurrence relations:

Consider the recurrence

\[ T(n) = aT(n/b) + f(n). \]

Then

- if \( f(n) \ll n^{\log_b a} \) then \( T(n) = O(n^{\log_b a}) \).
- if \( f(n) \approx n^{\log_b a} \) then \( T(n) = O(n^{\log_b a} \log n) \).
- if \( f(n) \gg n^{\log_b a} \) and \( \lim_{n \to \infty} \frac{af(n/b)}{f(n)} < 1 \) then \( T(n) = O(f(n)) \).

**Example.** If \( T(n) \) denotes the time taken for the divide-and-conquer convex hull algorithm (ignoring the initial sort), then we obtain the recurrence

\[ T(n) = 2T(n/2) + O(n). \]

This solves to \( O(n \log n) \).

A3.2 Quicksort

Several of the common sorts use divide-and-conquer. For example, recall *Quicksort*, invented by Hoare in 1962. Say one starts with \( n \) distinct numbers in a list. Call the list \( A \). Then Quicksort does:
Quicksort (A:valuelist)
1. Choose an element as the Key.
2. Split the list into two sublists $A_<$ and $A_>$ (called buckets).
   The bucket $A_<$ contains those elements smaller than the Key and
   the bucket $A_>$ contains those elements larger than the Key.
3. Use Quicksort to sort both buckets recursively.

There remain questions including:

1) Implementation
2) Choosing the key
3) Speed
4) Storage required
5) Is this the best we can do?
6) Problems with the method.

The beauty of Quicksort lies in the storage requirement: the sorting takes place “in situ”
and very little extra memory is required.

A3.3 How Fast is Quicksort?

To analyze the speed, we focus on the number of comparisons between data items. We
count only the comparisons. We will come back to whether this is valid or not. But
even this counting is hard to do.

◊ Worst case

What is the worst case scenario? A very uneven split. For example, our Key might be
the minimum value. Then we compare it with every element in the list only to find that
the bucket $A_<$ is empty. Then when we sort $A_>$ we might again be unlucky and have
the minimum value of that bucket as the Key. In fact, if the list was already sorted
we would end up comparing every element with every other element for a total of $\binom{n}{2}$
comparisons.

We can analyze the worst case (when the list is in fact already sorted) another way:
The first step breaks the list up into $A_<$ which is empty and $A_>$ which contains $n - 1$
items. This takes $n - 1$ comparisons. Then $A_>$ is split using $n - 2$ comparisons, and
leaves a basket of $n - 2$ items. So number of comparisons is:

\[
(n - 1) + (n - 2) + (n - 3) + \ldots + 2 + 1 = n(n - 1)/2 \approx n^2/2.
\]
Best case

The best case is when the list is split evenly each time. (Why?) In this case the size of the largest bucket goes down by a factor of 2 each time.

At the top level we use \( n - 1 \) comparisons and then have to sort the buckets \( A_\leq \) and \( A_\geq \) which have approximately \( (n-1)/2 \) elements each. To make the arithmetic simpler, let’s say that we use \( n \) comparisons and end up with two buckets of size \( n/2 \).

Let \( f(n) \) denote the number of comparisons needed by Quicksort in the best case. We then have the recurrence relation:

\[
f(n) = n + 2f(n/2)
\]

with the boundary condition that \( f(1) = 0 \). One can then check that the solution to this, at least in the case that \( n \) is a power of 2, is

\[
f(n) = n \log_2 n
\]

Average case

Of course, we are actually interested in what happens in real life. Fortunately, the typical behavior of Quicksort is much more like \( n \log_2 n \) than \( n^2 \). We do not explore this here—but see Exercise 4.

A3.4 Merge Sort

Another sort that uses divide and conquer is Merge Sort.

\textbf{MergeSort (A:valuelist)}

1. Arbitrarily split the list into two halves.
2. Use MergeSort to sort each half.
3. Merge the two sorted halves.

One divides the list into two pieces just by slicing in the middle. Then one sorts each piece using recursion. Finally one is left with two sorted lists. And must now combine them. The process of combining is known as merging.

How quickly can one merge? Well, think of the two sorted lists as stacks of exam papers sitting on the desk with the worst grade on top of each pile. The worst grade in the entire list is either the worst grade in the first pile or the worst grade in the second pile. So compare the two top elements and set the worst aside. The second-worst grade is now found by comparing the top grade on both piles and setting it aside. Etc.
Why does this work?

How long does merging take? Answer: One comparison for every element placed in the sorted pile. So, roughly \( n \) comparisons where \( n \) is the total number of elements in the combined list. (It could take less. When?)

Merge Sort therefore obeys the following recurrence relation

\[
M(n) = n + 2M(n/2).
\]

(Or rather the number of comparisons is like this.)

We’ve seen before that the solution to this equation is \( n \log_2 n \). Thus Merge Sort is an \( n \log n \) algorithm in the worst case.

What is the drawback of this method?

### A3.5 Optimality of Sorting

So we have sorting algorithms that take time proportional to \( n \log_2 n \). Is this the best we can do? Yes, in some sense, as we will show later.

### A3.6 Good Algorithms

Recursion is often easy to think of but does not always work well. Consider the problem of calculating the \( n \)th Fibonacci number, which is defined by

\[
a_n = a_{n-1} + a_{n-2}
\]

with initial values \( a_1 = 1 \) and \( a_2 = 1 \). Recursion is terrible! Rather do iteration: calculate \( a_3 \) then \( a_4 \) etc.

The key to good performance in divide and conquer, is to partition the problem as evenly as possible, and to save something over the naïve implementation.

### Exercises

1. Illustrate the behavior of Quicksort and Merge Sort on the following data: 2, 4, 19, 8, 9, 17, 5, 3, 7, 11, 13, 16

2. In your favorite programming language, code up the Quicksort algorithm. Test it on random lists of length \( 10^i \) for \( i = 0, 1, \ldots \), and comment on the results.

3. Suppose we have a list of \( n \) numbers. The list is guaranteed to have a number which appears more than \( n/2 \) times on it. Devise a good algorithm to find the Majority element.
4. Let \( q(n) \) be the average number of data-comparisons required for Quicksort on a randomly generated list.

a) Explain why \( q(1) = 0 \), \( q(2) = 1 \) and \( q(3) = 2\frac{2}{3} \)

b) Explain why the following recurrence holds:

\[
q(n) = n - 1 + \frac{2}{n} \sum_{i=0}^{n-1} q(i)
\]

c) Show that the solution to the above recurrence is:

\[
q(n) = 2(n + 1) \sum_{k=1}^{n} \frac{1}{k} - 4n
\]

d) Use a calculator, computer or mathematical analysis to look at the asymptotics of \( q(n) \).

5. a) Suppose we have a collection of records with a 1-bit key \( K \). Devise an efficient algorithm to separate the records with \( K = 0 \) from those with \( K = 1 \).

b) What about a 2-bit key?

c) What has all this to do with sorting? Discuss.

6. Consider the recurrence:

\[
f(n) = 2f(n/2) + cn \quad f(1) = 0
\]

Prove that this has the solution \( f(n) = cn \log_2 n \) for \( n \) a power of 2.

7. Consider the following program:

```python
function Fibonacci(n)
    if n<2 then return n
    else return Fibonacci(n-1) + Fibonacci(n-2)
```

Analyze the time used for this algorithm.
Chapter A4: More Divide and Conquer

Here are some more divide-and-conquer algorithms.

A4.1 Multiplying Long Integers

Another example of divide-and-conquer is the problem of multiplying long integers. The following is based on the discussion by Aho, Hopcroft, and Ullman.

Consider the problem of multiplying two $n$-bit integers $X$ and $Y$. At elementary school we learnt long multiplication: if we take two $n$-digit numbers then long multiplication takes $O(n^2)$ (quadratic) time. (Why?)

We can apply divide and conquer. For example, suppose we split $X$ and $Y$ into two halves:

$$X := \begin{bmatrix} A \\ B \end{bmatrix} \quad Y := \begin{bmatrix} C \\ D \end{bmatrix}$$

Then $X = A2^{n/2} + B$ and $Y = C2^{n/2} + D$. The product of $X$ and $Y$ can now be written as:

$$XY = AC2^n + (AD + BC)2^{n/2} + BD$$

This product requires four multiplications and three additions. To add two integers takes linear time. (Why?) So we obtain the recurrence:

$$T(n) = 4T(n/2) + O(n)$$

Alas, the solution to this is again quadratic, and in practice this algorithm is worse than normal long multiplication.

However, consider the following formula:

$$XY = AC \cdot 2^n + \left[ (A - B)(D - C) + AC + BD \right] 2^{n/2} + BD$$

At first glance this looks more complicated. But we only need three multiplications to do this: $AC$, $BD$ and $(A - B)(D - C)$. Therefore we obtain the recurrence:

$$T(n) = 3T(n/2) + cn$$

whose solution is $T(n) = O(n^{\log_2 3}) = O(n^{1.59})$.

There is some more work needed to write this down in a program. We need to consider shifts and also deal with negative integers. Left as an exercise.
A4.2 Matrix Multiplication

Consider the problem of multiplying two \( n \times n \) matrices, that is with \( n \) rows and \( n \) columns. Recall that the result is itself an \( n \times n \) matrix, where the entry in row \( i \) column \( j \) of the result is the “dot product” of row \( i \) in the first matrix with column \( j \) in the second matrix. Each dot product multiplies the corresponding entries in the two arrays and sums this up. Thus, computing each entry in the product takes \( n \) multiplications and there are \( n^2 \) entries for a total of \( O(n^3) \) work. Can one do better?

Strassen devised a better method which has the same basic flavor as the multiplication of long integers. The key idea is to save one multiplication on a small problem and then use recursion.

We look first at multiplying two \( 2 \times 2 \) matrices. This would normally take 8 multiplications.

\[
A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \quad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} \quad AB = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}
\]

But it turns out we can do it with 7 multiplications. You can check the following. If and one calculates the following products:

\[
m_1 = (a_{21} + a_{22} - a_{11})(b_{22} - b_{12} + b_{11}) \\
m_2 = a_{11}b_{11} \\
m_3 = a_{12}b_{21} \\
m_4 = (a_{11} - a_{21})(b_{22} - b_{12}) \\
m_5 = (a_{21} + a_{22})(b_{12} - b_{11}) \\
m_6 = (a_{12} - a_{21} + a_{11} - a_{22})b_{22} \\
m_7 = a_{22}(b_{12} + b_{21} - b_{11} - b_{22})
\]

then the product of \( A \) and \( B \) is given by

\[
AB = \begin{pmatrix} m_2 + m_3 & m_1 + m_2 + m_5 + m_6 \\ m_1 + m_2 + m_4 + m_7 & m_1 + m_2 + m_4 + m_5 \end{pmatrix}
\]

Wow! But what use is this?

What we do is to use this idea recursively. If we have a \( 2n \times 2n \) matrix, we can split it into four \( n \times n \) sub-matrices:

\[
A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}
\]

and form seven products \( M_1 \) up to \( M_7 \), and the overall product \( AB \) can thus be calculated with seven multiplications of \( n \times n \) matrices. Since adding matrices is clearly proportional to their size, we obtain a recurrence relation:

\[
f(n) = 7f(n/2) + O(n^2)
\]

whose solution is \( O(n^{\log_2 7}) \), an improvement on \( O(n^3) \).
A4.3 Closest Pair

The problem is to find, given \( n \) points in the plane, the pair of points that are the closest. Quadratic time is trivial. But one can do better.

We again use a divide and conquer algorithm. The presentation of the algorithm assumes no two points are coincident.

Presort the points from left to right. Then:

<table>
<thead>
<tr>
<th>Closest Pair</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Divide the ( n ) points into left and right halves, called ( L ) and ( R ).</td>
</tr>
<tr>
<td>2. Find the closest pair inside ( L ) and inside ( R ).</td>
</tr>
<tr>
<td>3. Determine the overall closest pair.</td>
</tr>
</tbody>
</table>

Again it is the merge Step (3) that does all the work. The overall closest pair is either inside \( L \), inside \( R \), or has one point in \( L \) and one in \( R \).

Let \( \delta \) be the smaller of the smallest distance in \( L \) and the smallest distance in \( R \). Consider a strip \( \delta \) either side of the dividing line. If in fact there are two points that are closer than \( \delta \) apart, then they must both lie inside the strip.

Of course there could be many points in the strip. But on each side they are at least \( \delta \) apart. This means—and this is the key observation—that the points in the strip are spread out vertically. (If one draws the circle of radius \( \delta/2 \) around each point in say \( L \), none of the circles intersect.) In particular, if one considers a rectangle on one side with width \( \delta \) and any fixed height, there is a limit to the number of points inside the rectangle.

So the idea is to presort all the points from top to bottom. When the merge step is reached, the points in \( L \) are examined in order from top to bottom. For each point, there is a small window of points in \( R \) that one needs to worry about. In particular, it can be shown that:
for each given point \( v_ℓ \) in \( L \), if there is a point \( v_r \) in \( R \) that is closer than \( δ \) to it, then \( v_r \) must be one of the 3 points either side of \( v_ℓ \) in the overall vertical ordering.

So, as we process the points \( v_ℓ \) in \( L \), we advance the window of points in \( R \) as needed. Thus the total work in Step (3) is at most \( O(n) \). By the Master Theorem, this means that the overall algorithm runs in \( O(n \log n) \) time.

**Exercises**

1. Illustrate the multiplication of 1234 and 5678 using the algorithm described above (converted to decimal).

2. Create a class for long integers. Then provide the operations shift, addition and multiplication. Then calculate the square of the 50th Fibonacci number.

3. Use Strassen’s algorithm to multiply the matrices

\[
\begin{pmatrix}
1 & 2 \\
3 & 4
\end{pmatrix}
\quad \text{and} \quad
\begin{pmatrix}
5 & 6 \\
7 & 8
\end{pmatrix}
\]

4. Implement the closest-point algorithm.

5. Give a good algorithm to determine if, given \( n \) points in the plane, any 3 points form a line.
Chapter A5: Finding the Median

The median of a collection is the middle value (in sorted order). For example, the collection \{3, 1, 6, 2, 8, 4, 9\} has a median of 4: There are three values smaller than 4 and three values larger. The answer to a median question when there is an even number of values is the smaller of the two middle values. (There are other possible definitions but this is convenient.)

We want an algorithm for finding the median of a collection of \(n\) values. One possibility would be to sort the list. But one hopes for a faster method. One can try recursion or iteration, but at first glance they don’t seem to work.

Blum, Floyd, Pratt, Rivest, and Tarjan provided a linear-time algorithm: it uses \(O(n)\) comparisons. It is a bit hairy at places.

A5.1 A Generalization

One approach to the median problem is to define a generalization: the rank selection problem. The input is a collection of \(n\) numbers and a rank \(k\). The output must be the value that is the \(k\)th smallest value. For example, for the above collection, 4 has rank 4 and 8 has rank 6. The minimum always has rank 1.

Thus the median of a collection with \(n\) values has rank \(\lceil n/2 \rceil\) (round up): it’s \(n/2\) if \(n\) is even and \((n + 1)/2\) if \(n\) is odd.

A5.2 Using Median for Rank Selection

Okay. We do not know how to find the median. But I now argue that if (and that’s a big if) we have an algorithm for median, then we can derive an algorithm for rank-selection:

We use the median algorithm to break the list in half. And keep only the relevant half.

How much work does this take?
**RankSelect** \((A: \text{valuelist}, k: \text{integer})\)

- **R1:** Calculate the median \(m\) of the list \(A\).
- **R2:** Determine the set \(A_<\) of elements smaller than the median.
  
  Determine the set \(A_>\) of elements larger than the median.
- **R3:** If \(k < \lceil n/2 \rceil\) then return \(\text{RankSelect}(A_<, k)\)
  
  If \(k > \lceil n/2 \rceil\) then return \(\text{RankSelect}(A_>, k - \lceil n/2 \rceil)\)

Else return \(m\)

Suppose that the median procedure takes \(cn\) comparisons where \(c\) is some constant. (That is, suppose we have a linear-time median algorithm.) The number of comparisons for Step R1 the first time is \(cn\). The number for Step R2 the first time is \(n\). The total for the first two steps is therefore \(n(1 + c)\) comparisons.

The second time we have a list half the size. So our work is half that of the original. The number of comparisons for Step R1 the second time is \(cn/2\). The number for Step R2 the second time is \(n/2\). The total for the first two steps is \(n(1 + c)/2\) comparisons.

Hence the total number of comparisons is

\[
n(1 + c) + \frac{3n(1 + c)}{2} + \frac{3n(1 + c)}{4} + \frac{n(1 + c)}{8} + \frac{n(1 + c)}{16} + \frac{n(1 + c)}{32} + \ldots
\]

If this sum continues forever, by formula for geometric series it would add up to

\[2n(1 + c)\]

So the total work is at most \(2(1 + c)n\). This is linear! (The beauty of the geometric series.)

**A5.3 The Pseudo-Median**

We still don’t have a median algorithm. There are two ideas needed:

(1) that the above algorithm is still efficient if one uses only an *approximation* to the median; and

(2) one can actually find an approximation to the median.

We define a *pseudo-median* as any value such that at least 30% of the values lie above the pseudo-median and at least 30% of the values lie below the pseudo-median. (The value 30% is chosen to make the presentation easier.)

Suppose we have an algorithm that finds a pseudo-median. Then we can use it for rank selection in exactly the same way as above. (And hence we can find the median!)
**RankSelect** (A:valuelist, k:integer)

R1: Calculate a pseudo-median \( m \) of the list \( A \).
R2: Determine the set \( A_\leq \) of elements smaller than \( m \).
    Determine the set \( A_\geq \) of elements larger than \( m \).
    Calculate the rank \( r \) of \( m \) in \( A \).
R3: If \( k < r \) then return RankSelect(\( A_\leq \), \( k \))
    If \( k > r \) then return RankSelect(\( A_\geq \), \( k - r \))
    Else return \( m \)

Hence we have a function **RankSelect** which uses as a subroutine the function **Pseudo-Median**.

### A5.4 Finding a Pseudo-Median

How do we find a pseudo-median? What we do is:

*take a sample of the data and find the median of the sample.*

This algorithm is bizarre but beautiful: we now have a function **Pseudo-Median** which uses as a subroutine the function **RankSelect**.

**Pseudo-Median** (B:valuelist)

P1: Break up the list \( B \) into quintets (groups of 5).
P2: Calculate the median in each quintet (by brute-force). Call the result the representative of the quintet. There are \( n/5 \) representatives.
P3: Return the median of the set of representatives:
    namely RankSelect(reps,\( n/10 \))

Does this produce a pseudo-median?

Well, consider the resultant value \( m_r \) which is the median of the representatives. This value is bigger than \( n/10 \) representatives. And for each representative that \( m_r \) is bigger than, \( m_r \) is bigger than two other members of the quintet. Thus there are at least \( 3n/10 \) values which \( m_r \) is guaranteed to be bigger than. And similarly there are \( 3n/10 \) values which \( m_r \) is guaranteed to be smaller than. Hence \( m_r \) is guaranteed to be a pseudo-median.
Note the intricate recursion: RankSelect calls PseudoMedian which calls RankSelect and so on. The reason why this works is that the problem size is getting smaller at each stage. (When implementing this algorithm, one must be very careful not to get an infinite loop.

### A5.5 Analysis

The calculation of the median in a quintet takes at most 10 comparisons. (Since sorting the quintet takes this many comparisons!) So the total number of comparisons for Step P2 is $2n$.

Now, let us return to the RankSelect algorithm modified to use PseudoMedian. Let $g(n)$ be the number of comparisons needed by RankSelect on a list of size $n$ in the worst case. Then the number of comparisons for the three steps are:

- **Step R1**: $2n + g(n/5)$
- **Step R2**: $n$
- **Step R3**: $g(7n/10)$

This gives us the recurrence:

$$g(n) = g(n/5) + g(7n/10) + 3n$$

which has solution $g(n) = 30n$. (Plug it in and check!)

### Exercises

1. Implement the median-finding algorithm.

2. Run your program on random lists of length $10^i$ for $i = 0, 1, \ldots$ Comment on the results.

3. Why quintets? Use either theory or a modification of your code to compare triplets with quintets.