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 \( \lim_{n \to \infty} a f(n/b)/f(n) < 1 \) 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.

\(\diamond\) 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 $A_<$ and $A_>$ 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.

```
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.