Contents

Foreword

There are already many well-written algorithm textbooks or lecture notes online, and what we teach in this module is mostly fairly standard material. Hence writing a complete set of lecture notes for this module feels a bit reinventing the wheel, but every now and then people ask for it, so here it is. There are inevitably many mistakes; please do point them out to me.

Occasionally you will see the reference [CLRS], which refer to the textbook:

- T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, 3rd edition; ISBN: 978-0-262-53305-8, MIT Press, 2009.

The exercises at the end of each chapter will be discussed in surgery sessions. **Please note the following important points:**

- You are expected to have attempted the questions on your own before coming to the surgery sessions.
- We may not cover every question. Solution outline will be on the web after each surgery session.
- Questions marked * are more "interesting" questions.
- Only questions similar to those marked # are likely to appear in a class test.

The best way to learn algorithms is to attempt a lot of exercises. There are many more exercises in the recommended textbooks. I was hoping to compile a huge set of questions we used in the past (or indeed, will use in the future) in coursework and exams, but that may or may not happen in time this year. Nevertheless you will be given some past assignments/exams, and please do make good use of them.

Chapter 1

Basic Concepts

1.1 What is an algorithm?

You have probably heard of the word "algorithm" many times throughout your degree; however, your previous understanding of the word "algorithm" is probably very different from what you are going to learn in this module. One way of defining an algorithm is as *a step-by-step procedure for solving a computational problem.* This immediately brings up two questions:

- What is a "computational problem"?
- What is a "step"?

We will not define precisely what is meant by "computational problems" here. This requires a formal notion such as "languages" and Turing machines which you may have learnt in CO2011. But informally, a computational problem is something that can be solved using a set of standard computational operations (such as those found in the instruction set of a CPU). We distinguish two types of problems:

- *Decision problems*, where the answer is just yes or no. An example is: given a map, determine whether it is possible to colour all the regions with at most three colours, with the usual restriction that countries sharing borders have to be assigned different colours.
- *Optimisation problems*, where we want to find the maximum or minimum value of something under certain constraints. An example is: giving a set of tasks, each with a deadline and a profit, find a way to schedule them to maximise the total profit of tasks completed before their deadlines.

Often these problems can be converted into one another: for example, "find the minimum number of colours required to colour this map" can be solved by a series of yes/no questions "can this map be coloured in k colours," with $k = 1, 2, \ldots$

It should be emphasised that an algorithm must consist of a *finite* sequence of *well-defined* operations. Without them being well-defined, it would not be possible to analyse their correctness and efficiency.

Programs vs. algorithms. An algorithm expresses the high level idea to solve a problem; the idea almost always does not depend on the choice of the programming language, or the architecture of the underlying machine. Thus algorithm design is not concerned with these "little details." In a sense, an algorithm is what is left unchanged when you translate from one programming language to another. At this moment this is perhaps all very abstract; hopefully by the end of the module, when you have seen many examples, you will see what this means.

In principle, an algorithm, being just an "idea," can be expressed in any way, for example just plain English. However, this often lacks precision, and so to express our ideas clearly we sometimes use **pseudocode**: something that looks like programming languages, but with no fixed syntax as long as it is clear what it means.[1] The following is some example pseudocode:[2]

Algorithm 1.1 A first pseudocode example.

Input: Array $A[1..n]$ of size n
Output: A number x in A

```
x ← A[1]
for i = 2 to n do
  if A[i] > x then
    x ← A[i]
  end if
end for
print x
```

Even though it probably uses syntax different from the programming languages you know, it is not difficult to see what this algorithm is trying to do. In fact, this is more "code" than "algorithm," as the "idea" here is really just "scan the array element by element." Later, when we deal with more complicated algorithms, we will not spell out this level of detail when describing algorithms.

1.2 Why study algorithm design?

Many important real-life problems need efficient algorithms, because they deal with very large inputs (e.g. DNA sequences, or a road map with tens of thousands of roads and towns), and an inefficient algorithm simply makes the system unusable. Unfortunately, the most straightforward algorithm is seldom the most efficient. There are inefficiency in those algorithms, and sometimes a deep understanding the structure of the problem is required for us to take advantage of it to produce more efficient algorithms. Often, these algorithms are non-trivial; it is not even clear why they perform the task correctly! There are, however, some general algorithm design paradigms that turned out to be useful for a wide variety of problems. We study algorithms to learn these techniques.

[1] Indeed you will see that the psuedocode in the lecture notes uses a different syntax from those in the Powerpoint slides...

[2] Note that in this example, the array is 1-based, i.e., the first element is $A[1]$ and the last $A[n]$. Most modern programming languages are 0-based with first element $A[0]$ and last $A[n-1]$. This is the kind of implementation details that do not affect the design of algorithms. We will use whichever is convenient for the particular algorithm in question, making it clear which way it is. You are also free to use either one when writing your own algorithms in this module.

We should emphasise here that algorithm design is not about programming tricks. For example, some of you may know a few tricks, such as optimising where to put certain statements inside or outside a loop, etc. This does not improve the scalability of the algorithm, something we will explain a bit later. What we do instead in algorithm design is to use a different way of thinking – "algorithmic thinking," which requires a very different mindset. To quote one of your textbook's author, Steven Skiena:

> Designing the right algorithm for a given application is a major creative act... being a successful algorithm designer requires more than book knowledge. It requires a certain attitude – the right problem-solving approach.

1.3 Efficiency of algorithms

We analyse algorithms both for their *correctness* and their *efficiency*. But let's focus on efficiency here. Often, there are many different algorithms for solving the same problem. Naturally, we want to pick the "best," i.e., most efficient one. There are a few different criteria against which we can measure the efficiency of algorithms. The most common one is *time*. Occasionally we are also interested in the amount of *space* used by the algorithm. This refers to the amount of working memory (i.e., excluding the memory for storing the inputs and outputs) used by the algorithm. Often there is a tradeoff between time and space: you can save time by using more space.

When analysing the time taken by an algorithm, we are not interested in the number of seconds/minutes/hours taken. This is because it depends on various factors like the speed of the CPU, or the skill of the programmer, which have nothing to do with the ingenuity of the design of the algorithm. Instead, we measure the running time of an algorithm by the number of "steps" it takes. What is a "step"? That depends on your *computational model*. For example, a CPU has a fixed set of instructions, where each instruction performs a well-defined action that finishes in a fixed (constant) amount of time. An ADD instruction adds two 32-bit binary numbers in a fixed number of clock cycles, irrespective of what the exact numbers being added are. Thus we may count one ADD instruction as a "step."

But this still leaves a lot of ambiguity, and in fact you may wonder how that could be a meaningful notion, given that different devices or processors have different instruction sets. Perhaps one instruction in some device is as powerful as 1000 instructions in another one? Fortunately, as far as we know, all "reasonable" computational models have "equivalent" computational powers.[3] Furthermore, the exact *number* of steps does not matter, rather it is about how it *grows*: a scalability issue which we will return to.

Worst-case running time. Even if we fixed one algorithm, the running time is not unique, as it typically depends on the input to the algorithm. For example, consider the *search problem* of finding a target element T from a given array A. One algorithm, the sequential search or linear search, simply checks each element in A one by one until either the element is found or all elements are checked. Sometimes, if you are lucky, the algorithm only needs to check $A[1]$ and it matches the target and so it finishes in just one "step." This is the best case scenario. The worst case, however, happens when the target is at the last position of A (or not in A at all), in which case it takes n "steps."

[3] This is the complexity-theoretic version or extension of the Church-Turing thesis.

We are interested in the *worst-case running time* of an algorithm. In other words, we want to know what is the *maximum* number of steps the algorithm takes *over all possible inputs* (of a certain size). There are a number of reasons why we are interested in this. Worst-case analysis gives an upper limit of the running time for any input, so we have a guarantee as to when the execution would stop, so you do not need to wait forever. Other measures, such as average-case running time, are possible, but have their own difficulties. It is not always easy to define what "average" is – there needs to be knowledge or assumption of the input distribution. For example, what is the "average" time to search an array? Is it roughly $n/2$ steps since "on average" the target is in the middle? It may well be, but maybe not, depending on the particular problem. Average-case running time is also often more difficult to analyse. For these reasons, unless otherwise stated, we are always interested in worst-case running time.

We will use the following running example to illustrate these concepts. First we define the problem:

Problem P: Coin weighing.
You are given 8 coins, exactly one of which is counterfeit and is lighter. Using a pan balance, find the counterfeit coin.

Here, the computational model is not a computer, but a pan balance. A "step" is one application of it: put some coins on either side and observe the outcome, which is one of three possibilities: either the left side is heavier, or the right side is heavier, or they are equal. You are not allowed to use other things like a digital scale, since that would be a different computational model. We want an efficient algorithm, i.e., one that takes the fewest steps or weighings.

Consider the following algorithm A1 for P:

Name the coins A, B, ..., H. Weigh A : B. If one side is lighter, report it and finish. Otherwise we know that both are genuine. Repeat the procedure for C : D, E : F, G : H.

What is the best-case input? Worst-case input?
What is the worst-case running time of this algorithm?

Consider the following algorithm A2 for P:

Weigh ABCD : EFGH. One side got to be lighter, which contains the counterfeit coin. Suppose ABCD is lighter (the opposite case is similar). In the second step weigh AB : CD. Again, one side is lighter, say AB. In the third step weigh A : B.

What is the worst-case running time of this algorithm?
Is it more efficient than A1?

1.4 Asymptotic complexity of algorithms

Scalability. We are usually not interested in the performance of an algorithm for a fixed size input; rather, we want to know how its performance changes *as the input size grows*. This is because we want to know how *scalable* an algorithm is. Naturally, when the input gets bigger, most algorithms will take a longer time to run, but how much more? Ideally we want it to grow only proportionally (i.e., if the input size doubles then the time also doubles). In many (but not all) cases, this is the best thing one can hope for. But there are less scalable – sometimes much less scalable – algorithms. To quantify this, we indicate the running time or **time complexity** of an algorithm as a *function of its input size*. The input size is usually denoted by n. For example, an algorithm may take at most $10000n$ steps for any input of size n (remember we want worst-case running times). Another one might take $500n^2$ steps. A third one might take 2^n steps. Which is "more scalable"? To see this, suppose we run them on a machine that executes 100,000 steps per second. Here is how they fare with different input sizes:

n	$10000n$	$500n^2$	2^n
1	0.1s	0.005s	0.00002s
10	1s	0.5s	0.01024s
50	5s	12.5s	357 years
100	10s	50s	4×10^{17} years

Table 1.1: Growth rate of three functions.

Clearly, the first one grows proportional to the input size. The second one is somewhat worse; it was initially faster but then becomes slower than the first one. The last one grows crazily even with very reasonably-sized inputs, and thus scales very badly.

The big-O notation. Recall the big-O notation that you learned in your first year? You might not have been told, back then, what it was for. Well, this is what it is for. You are going to see it in almost every page here. It captures this notion of scalability, also known as the **order of growth** or the **asymptotic** time complexity. Here is its formal definition:

Definition 1.1 *For two functions $f(n)$ and $g(n)$, we say $f(n)$ is $O(g(n))$, or informally*[4] *$f(n) = O(g(n))$, if there exists constants c and n_0 such that*

$$f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0$$

Intuitively, this means "for large enough n, $f(n)$ is at most a constant times larger than $g(n)$." In the asymptotic sense, this means "f grows slower than (or same as) g."[5] For example, Figure 1.1 shows (not to scale) $f(n) = 0.5n + 4$ and $g(n) = n^2 + 3$, and $f(n) = O(g(n))$ because apart from an initial segment, g is always bigger than f.

[4]Strictly speaking this is incorrect as $O(g(n))$ is a set of functions. However, this informal use of the = sign is very commonplace. You will even see it as part of normal arithmetic like $n^2 + O(n)$. The = sign is therefore not symmetric: for example, don't write "$O(n^2) = 3n^2 + 2n - 1$." Also, as the use of big-O is to simplify the expressions, don't write "$n^2 = O(3n^2 + 2n - 1)$" even though it is technically correct.

[5]Here is an unfortunate confusion of terminology: f *grows slower* than g because f's values increase much more slowly than g's; but if these values are running times, it means f represents the time complexity of a *faster* algorithm! Also, note that Big-O can be used for other things, not just time complexities.

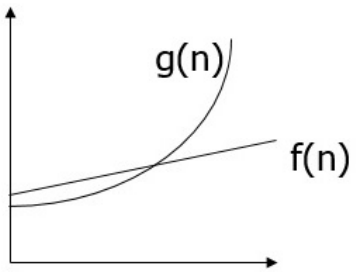


Figure 1.1: An example where $f(n) = O(g(n))$.

The big-O notation is ideally suited to describe the time complexities of algorithms, because of its following properties:

- It ignores *lower order terms*. In an expression like $2n^2 + 3n - 1$, anything other than the highest power term $2n^2$, such as $3n$ and 1, are called lower order terms.

  Why do we want to ignore lower order terms? Except for the most trivial algorithms, the time complexity of an algorithm is often very difficult to state precisely down to something like $2n^3 + 3n^2 + 4n + 5$. Furthermore, such detail is pointless. An algorithm with time complexity $2n^3 + 3n^2 + 4n + 5$ and another one with $2n^3 + 7n^2 + 8n + 9$ has practically the same efficiency when n is large, because the first term $2n^3$ "dominates" the value of the expression when n is large and the other terms are insignificant. (Exercise 1-3 asks you to experiment with this using Excel or other similar programs.)

- It ignores *constant factors*. This refers to the c in the definition. For example, $2n^2$ and $3n^2$ are both $O(n^2)$ and are therefore "the same." These constants do not measure the performance of algorithms well, because it is affected by external factors like machine speeds. It also does not measure the scalability: both $2n^2$ and $3n^2$ quadruple when n doubles. If you want to improve the performance by a constant factor, you can always buy a faster processor, but you cannot improve the scalability (e.g. from $O(n^2)$ to $O(n)$) by doing that. Indeed, this is precisely why we need algorithm design.

- It ignores what happens when n is small. This refers to the n_0 in the definition. We only care about their performances when n is large; any poor performance for "small" input sizes are ignored.

The following are some common time complexities (in increasing order):

- $O(1)$: This is a special notation used to denote a constant (independent of n).
- $O(\log n)$: Logarithmic, slower than linear (in fact much slower).
- $O(n)$: Linear, i.e. proportional.
- $O(n \log n)$: A time complexity that somehow arises very often.
- $O(n^2)$: Quadratic. Slower but usually acceptable except with really large inputs.

- $O(n^3)$: Even slower.
- $O(2^n)$: Exponential. Completely unacceptably slow.

The big-O notation can also be used with multiple variables. For example, if the input consists of two arrays of sizes m and n, or a graph with n vertices and m edges, that the time complexities could be $O(m+n)$ (linear), $O(mn)$, etc.

Tractability. We usually say the running time is *efficient* if it is a *polynomial* in the input size, i.e. $O(n^k)$ for constant k. Otherwise (e.g. exponential, $O(2^n)$), we regard it as inefficient. Looking back at Table 1.1 you will see that exponential functions do grow much faster than polynomial functions. This division between efficient and inefficient algorithms may seem a bit arbitrary: after all, a polynomial running time of n^{100} is slower than an exponential running time of 1.001^n for all but the most colossal-sized inputs. But this definition withstood the test of time and turned out to be a good measure, both theoretically and practically. Also, it is very rare to see these kinds of exotic running times: often, if someone discovered a (say) $O(n^{14})$ time algorithm, it gets quickly improved to some lower order polynomials.

Problems that do not have polynomial time algorithms are called *intractable*. Usually, nothing significantly better than brute force (search all possibilities) is known. This also relates to the class of problems known as NP-complete, where we *believe* they have no efficient algorithms but no one managed to prove that.

1.5 Upper and lower bounds

> **Warning:** confusion ahead!

All we have discussed so far refer to the time taken to solve a problem by running an algorithm. These are *upper bounds*:

Definition 1.2 *An* **upper bound** *(on the time complexity)* **of an algorithm** *is the worst-case number of operations sufficient to solve a problem by that algorithm.*

This is the worst-case complexity of an algorithm: under any input, the algorithm will finish after that many steps. For example, we can say that the Algorithm A1 for the coin weighing problem has an upper bound of 4, and Algorithm 1.1 has an upper bound of $O(n)$, because they never take more than that number of steps.

As we said before, often there are many algorithms for the same problem, and obviously we want "good" (efficient) algorithms, which means we want the upper bound to be as small as possible.

There are a number of confusing technicalities about upper bounds. Upper bounds may be "loose" (i.e., not tight). Suppose someone proved that a certain algorithm A has an upper bound of $O(n^3)$. It is also technically correct to say that A has an upper bound of $O(n^4)$, or indeed $O(n^{1000})$, since that many steps are clearly *sufficient* (a lot more than sufficient) for A to finish. They are rather pointless, but technically correct, statements.

But this kind of deliberate silliness is not the only reason for such looseness. It is often difficult to analyse algorithms exactly, and, for example, we may only be able give a "loose"

proof that A takes $O(n^3)$ time, when in fact it may never require more than $O(n^2)$ time on any input; it is not always easy to figure out what the worst input of an algorithm is. Indeed it may be possible that in the future some other people can prove that A indeed takes $O(n^2)$ time. Other times, we can establish the tightness of the upper bound: for example we prove that A runs in $O(n^2)$ time on *all* inputs and that A does require n^2 steps on *some* input. In this case we sometimes (confusingly) refer this as the lower bound of an algorithm, i.e., the number of operations really taken by that algorithm on some input. (This is not to be confused with the lower bound of a problem, described next.)

We often use the same concept on problems rather than algorithms:

Definition 1.3 *An* **upper bound** *(on the time complexity)* **of a problem** *is the worst-case number of operations sufficient to solve the problem by some (known) algorithm.*

So for example, as you will see later, there are sorting algorithms that run in $O(n \log n)$ time and $O(n^2)$ time, respectively. We can then say that sorting has an upper bound of $O(n \log n)$. (Similar to the reasoning we have just seen, we can also say that sorting has an upper bound of $O(n^2)$ – it is a correct upper bound, just not the best upper bound.)

As an algorithm designer, we pursue better and better upper bounds. But when do we stop? It is natural that problems have an inherent complexity and require a certain minimum number of steps to solve, no matter how clever the algorithms are. This is what we call *lower bounds*:

Definition 1.4 *A* **lower bound** *(on the time complexity)* **of a problem** *is the worst-case number of operations necessary to solve the problem by any algorithm.*

The lower bound is for a problem; it has nothing to do with any specific algorithm. A problem does not have a lower bound x just because a known algorithm for it takes x steps, or that all known algorithms take at least x steps. It is a proof that it is impossible to have a more efficient algorithm, including yet-to-be-discovered ones. How can we possibly prove such a thing? We will cover this in Chapter 5.

An algorithm has optimal time complexity, or is simply called *optimal*, if its upper bound matches the lower bound of the problem. The job of an algorithm designer is to design algorithms as close to optimal as possible; this means reducing the gap between upper and lower bounds. If they match, then we know that the algorithm we have is optimal. This also means we do not just reduce the upper bounds (design better algorithms), but also try to prove better, stronger lower bounds – and a lower bound is better if it is bigger, because it closes the gap.[6]

In an ideal world, all problems would have matching upper and lower bounds. However, many important and sometimes simple problems do not yet have optimal algorithms. We do not even know what is the optimal algorithm for multiplying two n-digit integers!

> Look at the coin weighing problem and the two algorithms A1 and A2 again. A1 has an upper bound of 4, and A2 has an upper bound of 3. A common mistake is then to simply say that P has an upper bound of 4 and a lower bound of 3. What's wrong with this?

[6] This is perhaps counterintuitive to beginners, who think "surely it is bad news that something is hard to do, and a bigger lower bound must be worse news?" But by doing this we are getting closer to the "truth" – what the optimal time complexity of the problem is.

For the upper bound: clearly 3 steps are sufficient to solve the problem P (since A2 can do that). Hence P has an upper bound of 3. (Technically 4 is also an upper bound, but it is not the best possible upper bound.)

For the lower bound: we know nothing about the lower bound of P (for now; wait till Chapter 5...) There might be algorithms better than A1 or A2 waiting to be discovered.

But suppose someone told you that they proved a lower bound of 2 for P. So now there is a gap between the best upper bound so far (3) and the best lower bound so far (2). So we should either try to design a better algorithm (to reduce the upper bound to 2) or to find a stronger lower bound proof (to raise the lower bound to 3) so that the two bounds match. Until then, we do not know whether the upper/lower bounds we have are optimal.

We want to have notations like big-O to denote asymptotic lower bounds, but big-O is not suitable because the direction of the inequality is wrong. So we have these notions:

Definition 1.5 *For two functions $f(n)$ and $g(n)$, we say $f(n)$ is $\Omega(g(n))$ if there exists constants c and n_0 such that*

$$f(n) \geq c \cdot g(n) \text{ for all } n \geq n_0$$

Definition 1.6 *For two functions $f(n)$ and $g(n)$, we say $f(n)$ is $\Theta(g(n))$ if there exists constants c and n_0 such that*

$$f(n) = c \cdot g(n) \text{ for all } n \geq n_0$$

Informally, $f(n) = \Omega(g(n))$ if f grows faster than (or same as) g, and $f(n) = \Theta(g(n))$ if f grows at the same rate as g. Big-Ω is usually used to express lower bounds, while Big-Θ is used to express optimal bounds or when we want to emphasise that certain quantities grow exactly at a certain rate (and not just loosely upper- or lower-bounded).

As a very rough analogy, we have

$$\begin{aligned} f(n) = O(g(n)) &\iff f \leq g \\ f(n) = \Omega(g(n)) &\iff f \geq g \\ f(n) = \Theta(g(n)) &\iff f = g \end{aligned}$$

but the (in)equality signs must be interpreted in an asymptotic sense.

1.6 Analysing time complexities of simple algorithms

The time complexity of algorithms that can be represented by simple procedural code can be analysed easily. Suppose we want to analyse the time complexity of the following algorithm:

Algorithm 1.2 An example algorithm.

Input: 2-D array $A[1..n][1..n]$
Output: a number max

```
 1: max ← 0
 2: for i = 1 to n do
 3:    sum ← 0
 4:    for j = 1 to n do
 5:       sum ← sum + A[i][j]
 6:       if sum > max then
 7:          max ← sum
 8:       end if
 9:    end for
10: end for
11: print max
```

First, remember that a "step" is a basic operation that takes constant time. We can assume that all standard primitive operations supported by a CPU (or programming language) take constant time. This includes arithmetic $(+, -, \times, /)$, comparisons $(<, =, >)$, reading from and writing into a given memory location (assuming no seek time is needed), etc. For example, lines 1, 3, 5, on their own, takes constant time.[7]

A program consists of a sequence of statements; naturally, for consecutive statements, the time to execute them is the sum of the times to execute each of them, since they are run sequentially.

Loops are also simple: we just need to know how many times the loop is executed, then multiply this by the time it takes to execute one iteration of the content of the loop.

Statements of the form "If A then B else C" has a time complexity that is the maximum of that of the sum of A and B, or the sum of A and C. Remember testing for A takes time; also, we won't go into both branches, just one of B or C, and we take the worst case.

Finally, if there are function calls, then the time it takes can be analysed separately.

Now let's apply these to the above algorithm. There are two nested for loops. Starting with the innermost, lines 5–8, on its own, take $O(1)$ time. But the j loop is iterated n times, so the time complexity of lines 4–9 (on its own, ignoring what's outside) is $O(n)$. The outer for loop repeats n times, the contents being a constant time statement (line 3) and the j loop that takes $O(n)$ time; so in total lines 2–10 take $n \times O(n) = O(n^2)$ time. Lines 1 and 11 take $O(1)$ time. Thus in total the algorithm takes $O(n^2)$ time.

Since this is a first example, we have done it slowly, step-by-step. With more experience you will be able to simply "see" that this is $O(n^2)$ "because" it has two for loops, and this kind of elaborate explanation is not needed.

Question: what is this algorithm doing?

[7] Please do not say line 5 takes $O(2)$ or $O(3)$ time just because you think there are two steps, + and ←, or three steps if you count the memory access, because of two reasons: first, as we said, $O(1)$ is a special notation. Second, what constitutes one "step," as we also said, depends on factors like the instruction set and is not important for the asymptotic analysis of algorithms.

Exercises

#1-1 **(Upper and lower bounds)**

Three algorithms A1, A2 and A3 for a certain problem P are known. Their worst-case time complexities on inputs of size n are $5n^2$, $2n + 4$ and 3^n respectively. Is each of the following statements true or false?

(a) A3 is always slower than A1.

(b) $O(n)$ is an upper bound on the time complexity of P.

(c) $O(2017^n)$ is an upper bound on the time complexity of P.

(d) $\Omega(n)$ is a lower bound on the time complexity of P.

(e) A2 is an optimal algorithm, i.e., its time complexity matches the lower bound of P.

1-2 **(Revision of logarithms)**

When analysing algorithms, we often encounter the log function. Recall that $x = \log_b y$ if $y = b^x$. b is called the base. Usually in computer science, log is in base 2. Below are some useful formulas: for any a, b, n,

$$\begin{aligned} \log_b a^n &= n \log_b a \\ \log_b n &= \log_a n / \log_a b \\ a^{\log_b n} &= n^{\log_b a} \end{aligned}$$

(a) If we start with a number n and in each step we reduce it by half, how many steps does it take to make it equal to (or smaller than) 1?

(b) Let $f(n) = \log n$ and $g(n) = 2^n$. What are $f(g(n))$ and $g(f(n))$?

1-3 **(Big-O notation)**

(a) Prove formally that $3n^2 + 2n - 1 = O(n^2)$.

(b) Prove formally that $3n^2 + 2n - 1 = O(n^3)$.

(c) When we write $O(\log n)$, we don't need to write down the base of the log. Why?

(d) Using Microsoft Excel or other software, plot the graph of the following functions, and hence arrange their asymptotic complexity in ascending order: n, $\log n$, $\sqrt{n}$, $n^{1.5}$, 1.5^n, $n/\log n$, $3^{\log n}$.

1-4 **(Dry-running an algorithm)**

Consider the following algorithm. The input is an array $A[1..n]$. The output is a two-dimensional array $B[1..n][1..n]$ where the entries $B[i][j]$ are filled only if $i \leq j$.

```
for i = 1 to n do
  for j = i to n do
    B[i][j] ← 0
    for k = i to j do
      B[i][j] ← B[i][j] + A[k]
    end for
```

end for
end for

#(a) If $A = [-1, 4, 3, -2]$, what is the output B? In general, what does $B[i][j]$ contain?

#(b) Analyse the time complexity of this algorithm.

(c) Give a more efficient algorithm to compute B and analyse its time complexity.

Chapter 2

Some Elementary Data Structures and Algorithms

Almost always, our algorithms are there to process data. We need efficient ways to represent the data we are going to process, or the intermediate working data that the algorithm produces. This is not just about the space usage, but also the time it takes to operate on the data. The efficiency of algorithms is highly dependent on the use of appropriate data structures. Computer scientist Niklaus Wirth even authored a book titled "Algorithms + Data Structures = Programs."

Different algorithms require different number of each type of operations. Some common operations on data include insertion, deletion, editing and searching. Different data structures support different operations with a different time complexity. There is often a tradeoff with the various competing objectives like space usage and time efficiency, and the choice depends on the particular algorithm in question. For example, if an algorithm requires a lot of insertion and deletion but not much else, then it makes sense to choose a data structure that supports these operations efficiently, perhaps at the expense of being slower at other operations or in its space usage. Another algorithm that never requires insertion or deletion, for example, can choose a data structure that does not support that, or is inefficient at doing that.

Some common data structures that we discuss in this chapter are: arrays, linked lists, stacks and queues.

2.1 Arrays and linked lists

The **array** is the simplest and most common data structure. Its most important property is that it supports *direct access*: any element $A[i]$ can be accessed in $O(1)$ (constant) time, irrespective of how large the array is or what i is. This is not as trivial as it sounds: this is only true because the array elements occupy contiguous memory locations.

If we want to search for an element, we simply check each element one by one (Algorithm 2.1 later), thus potentially reading the entire array and therefore has worst-case $O(n)$ time.

Appending an element at the end is easy (but may need memory allocation). But if we want to insert an element at a specific position (and maintain the order of the existing elements), we need to shift all elements behind it one position, to create the empty space, which can take $O(n)$ time. Similarly, in deletion, if we want to avoid leaving an empty space, then shifting elements takes $O(n)$ time.

A **linked list** is conceptually a number of nodes linked together by *pointers* – ways of locating the next node. In a physical realisation (i.e., in computer memory) the elements are not necessarily located next to each other; the pointer records the address of the next node so we know where to find it.

You may well have used linked lists or other data structures already in your programs, since programming languages like Java have built-in support for them. However, we need to understand how they work in order to analyse their running times. Here we will use a C/C++ style implementation for illustration. It is a more "natural" language to talk about things like pointers, but don't worry if you don't know C/C++.

A node is an object with two fields, one for the data and one for the pointer. We access a list by having a pointer to the head, the first node of the list.

```
struct node {
  int data;   // assume our data are integers
  node *next; // pointer to next node
}

node *head = new node;
```

A linked list does not support direct access. To get to the i-th element, you have to travel along or *traverse* the list step by step. The following is how to search, which takes $O(n)$ time for a list with n elements:

```
node* search(int x) {
  node *temp = head;
  while (temp != null && temp->data != x)
    temp = temp->next;
  return temp; // null if not found
}
```

The advantage of linked lists is that insertion or deletion can be done in constant time, assuming we already have a pointer to the location to insert to or delete from (for example as the result of a previous search operation). This is because we only need to reassign a few pointers. See Figure 2.1.

```
// insert x after node y
void insert(int x, node *y) {
  node *t = new node; // allocate memory for new node
  t->num = x;
  t->next = y->next;
  y->next = t;
}

// delete element pointed to by x (after x)
void delete(node *x) {
  x->next = x->next->next;
}
```

(Note that the element is merely "bypassed" and not actually "deleted," and still occupies memory space. In C/C++ you would need a free/delete operation, and in Java this is handled by garbage collection.)

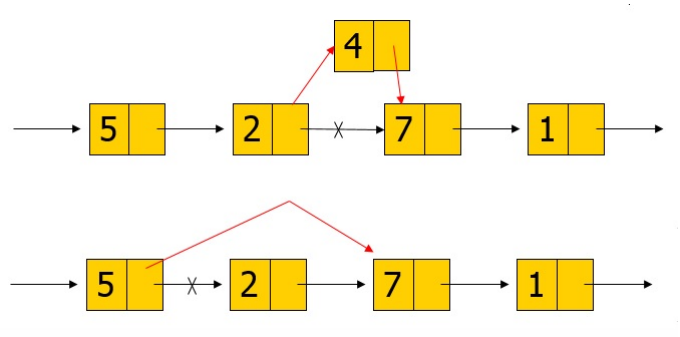


Figure 2.1: Linked list insertion (top) and deletion (bottom).

For simplicity, in the above code we have not fully handled error checking or boundary conditions, e.g. what happens when it is already at the end of the list and there is no "next" element etc. These can be added without affecting the time complexity.

We can also have *doubly linked lists*, where each element has both a pointer to the next one as well as a pointer to the previous one; or *circular linked lists*, where the end of the list points back to the head. They support easier navigation through the list, but takes more time to update after insertion/deletion and also takes up more space (both by a constant factor).

2.2 Abstract data structures: stacks and queues

A **stack** is a *last-in-first-out* (LIFO) data structure. Insertion and deletion of elements can only happen at the same end (called the *top*). Imagine a stack of plates: you cannot (easily) add or remove one from the middle. More precisely, the operations supported are:

- **Push**(v): insert v to stack S
- **Pop**(): return and remove top element from S
- **Top**(): return (but not remove) top element in S

In contrast, a **queue** is a *first-in-first-out* (FIFO) data structure. Insertion can only be made at one end (the *tail*), and deletion only at at other end (the *head*). The operations supported are:

- **Enqueue**(v): insert v into queue Q
- **Dequeue**(): return and remove first element from Q

Stacks and queues are examples of **abstract data structures**. They are specified by the set of operations they can perform and their running times, without details of how they are done. Essentially it is like classes in object-oriented languages that encapsulate the inner working

details from the programmer, and only expose an "interface." Algorithm designers only need to look at the interface when considering which data structure to use.

Abstract data structures are implemented using other data structures. Sometimes they can be implemented in more than one ways. For example, both stacks or queues can be implemented using arrays or linked lists in a way that support constant time per operation. In the following we discuss their implementations using arrays. (Exercise 2-1 asks you to implement them using linked lists.) For convenience (to ignore issues of memory allocation), we assume the stack/queue has a fixed maximum size (100).

```
class Stack {
  int S[0..99];
  int top = -1;

  void push(int x) {
    if (top == 99) print "stack overflow";
    else { top++; S[top] = x; }
  }

  int pop() {
    if (top == -1) print "stack underflow";
    else { top--; return S[top+1]; }
  }
}
```

The above is the stack implementation. It uses a `top` pointer to point to (i.e. record the index of) the position of the top of the stack. $S[0..top]$ are the cells occupied by elements.

Using a similar idea for a queue may sound straightforward at first, but it requires more care. The problem is with deletion: after the head element left, we cannot shift the elements all to the left by one position, since it will take $O(n)$ time. We can use two pointers, `h` and `t`, to point to the head and tail of the queue respectively, with the head moving to the next position when an element is deleted. Thus the elements are not always occupying from the beginning of the array, but in the range $Q[h..t-1]$. But after a while there will be a lot of wasted empty space at the beginning of Q. So we need to "wrap around" when the tail reaches the last element (imagine the array being circular). This also means we may have a somewhat counterintuitive situation where the tail is in front of the head. The following pseudocode implements this idea.

```
class Queue {
  int Q[0..99]; // NB: size 100 but stores only 99 elements
  int h = 0, t = 0;

  void Enqueue(int x) {
    if ((t+1)%100 == h) print "queue full";
    else {
      Q[t] = x;
      t = (t+1)%100;
    }
  }
```

```
  int Dequeue() {
    if (t == h) print "queue empty";
    else {
      x = Q[h];
      h = (h+1)%100;
      return x;
    }
  }
}
```

Note that although the array has size 100, the queue only stores a maximum of 99 elements. Otherwise, you cannot distinguish between a full queue and an empty queue. (Pause for a moment and consider why.)

2.3 Searching

Searching is a very fundamental problem: given an array A with n elements, report the location of an element x in A, or that it does not appear in A. A trivial algorithm is simply to search one by one:

Algorithm 2.1 Linear search

Input: Array $A[1..n]$, target x
Output: index i such that $A[i]$ is equal to x

```
for i = 1 to n do
  if A[i] == x then
    return i
  end if
end for
print "not found"
```

This is **linear search**. Clearly, its time complexity is $O(n)$. It can be shown that this is optimal if the input does not have any other known properties. However, a linear search time is often unacceptable, since we usually search from a vast collection of data.

If the elements are already sorted, we can do better. The idea of **binary search** is to reduce the search space by half by checking the middle element. Suppose the elements are sorted in increasing order, and we are looking for an element x. If the middle element is $> x$, then x can only be in the first half; if the middle element is $< x$, then x can only be in the second half. We are now faced with the same problem but of smaller (half) size. We can then repeat the same idea on the remaining half, either via *recursion* or by a controlled iteration. We will explain more about recursion in the next chapter; here we present a non-recursive version. We use two indices lo and hi to indicate the range of the array we are searching; at any point of the execution, $A[lo..hi]$ is where the target may still possibly be.

Algorithm 2.2 Binary search, non-recursive version.

Input: Array $A[1..n]$, target x
Output: index i such that $A[i]$ is equal to x

```
lo ← 1, hi ← n
while lo ≤ hi do
  mid ← round((lo + hi)/2)    // rounding
  if A[mid] == x then
    return mid    // found
  else if A[mid] > x then
    hi ← mid − 1    // lower half
  else
    lo ← mid + 1    // upper half
  end if
end while
```

What is the time complexity of binary search? Each execution of the contents inside the while loop takes $O(1)$ time. So the question amounts to how many times is the while loop executed. Observe that each execution of the loop reduces the range $(hi - lo + 1)$ by about half. The value $hi - lo + 1$ is n at the beginning, and 1 at the end. Thus, assuming n is a power of 2, the value decreases as the sequence $n \to n/2 \to n/4 \to \ldots \to 2 \to 1$ which we know (from Exercise 1-2) has $\log_2 n$ steps. Thus the overall time complexity is $O(\log n)$. It is therefore much better than linear search, which has a linear time complexity.

2.4 Sorting

Another natural and very fundamental problem in computer science is *sorting*: given n items that are "comparable" (i.e., one can assign $<$, $=$ or $>$ to any two given elements), arrange them in ascending (or descending) order. The most natural example is of course sorting numbers, but many other things like strings can be sorted as long as an ordering can be assigned to them.

Here we study two simple sorting algorithms. Later we will introduce more advanced algorithms that are more efficient.

Selection sort. This is based on the idea of repeatedly finding the smallest element, swap it to the first position, and repeat for the remaining subarray. Or in pseudocode:

Algorithm 2.3 Selection sort

Input: Array $A[1..n]$

```
for i = 1 to n − 1 do
  min ← i
  for j = i + 1 to n do
    if A[j] < A[min] then
      min ← j    // min is the index of the current minimum
    end if
    swap the values of A[i] and A[min]
  end for
end for
```

The algorithm maintains the invariant that when the outer loop finishes for a certain i, $A[1..i]$ contains the i smallest numbers in sorted order.

Its time complexity is easy to analyse. There are two for loops: The outer i for loop is executed $n-1$ times. The inner j for loop is executed at most $n-1$ times, for each value of i. (To be precise it is executed $n-i$ times, but this is at most $n-1$. It can be shown that even when counting precisely with $n-i$, the big-O time complexity is unaffected.) The contents inside the nested loops take $O(1)$ time per execution. Thus the total time complexity is $O(n^2)$.

Insertion sort. Here we use a different idea: we process elements one by one, while maintaining an initial part $A[1..i]$ of A to be sorted. Each time we consider the next element $A[i+1]$, and insert it in the correct position within $A[1..i]$ to create a bigger sorted array $A[1..i+1]$. Repeat the procedure to further extend the sorted part until the whole array is sorted.

Algorithm 2.4 Insertion sort

```
Input: Array A[1..n]
  for i = 2 to n do
    j ← i − 1
    x ← A[i]
    while j > 0 and x < A[j] do
      A[j + 1] ← A[j]
      j ← j − 1
    end while
    A[j + 1] ← x
  end for
```

The while loop considers elements in $A[1..i-1]$, starting from the end, and for each element check if it is larger than $A[i]$. If so, the insertion point is before this element, so we move this one position to the right to create a space for possible insertion. We keep doing this until we encounter an element smaller than $A[i]$, which means we should now place $A[i]$ into this space.

It is easy to see that the time complexity is again $O(n^2)$: the outer for loop clearly executes at most n times, and for each such iteration, the inner while loop iterates at most n times.

Some discussions. Before we leave this topic we discuss a few minor points.

First, in insertion sort, we do not really need $O(n)$ time to find the correct position to insert. The subarray $A[1..i]$ is already sorted; hence we can simply use binary search for the correct position! Incorporating this idea gives us *binary insertion sort*. Its total number of comparisons made is reduced to $O(n \log n)$. However, since we need to move the elements after insertion (which takes $O(n)$ time), the total time is still $O(n^2)$. But this might be useful if comparison is a very expensive operation.

Both selection sort and insertion sort take $O(n^2)$ time, but they have different properties. In selection sort, once a position is found for an element, it remains there; this is not true for insertion sort. But insertion sort benefits from partially-sorted input: if the input is "more or less" sorted, it is typically faster, since the inner while loop will not need to go too far before finding the correct insertion point. Insertion sort can also be done *on-line*, meaning that it can run without waiting all the inputs to arrive. In other words, it can be run if the data is being "streamed in." This is not the case for selection sort.

Exercises

2-1 **(Stacks and queues)**

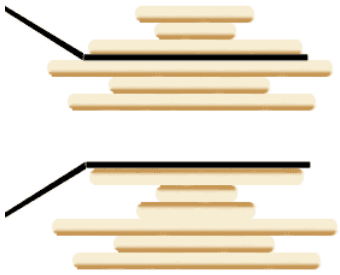
Implement stacks and queues using linked lists so that all operations are supported in $O(1)$ time.

2-2 **(Sorting)**

What is the worst-case input for sorting 5 numbers in ascending order using insertion sort? How many comparisons are needed? What if it is n numbers instead of 5?

*2-3 **(Pancake sorting)**

There is a stack of n pancakes in different sizes. We want to arrange them in increasing order of sizes from top to bottom. The only operation allowed is to insert a spatula immediately below a certain pancake, and flip all the pancakes above (see picture). This counts as one step.



Give an algorithm that sorts any stack of n pancakes with at most $2n - 3$ flips in the worst case. (Hint: consider selection sort and/or recursion.) [1]

*2-4 **(A fatal search algorithm)**

The Computer Science department is moving into a new building with n floors, and the professors are given a task of determining the "lethal height" of the building, i.e., the floor x where people falling from it or above will die while people falling from floor $x - 1$ or below will not. They have one resource: their own lives.

(a) A simple algorithm is to ask a professor to try jumping out of each floor (starting from the bottom) until he or she dies. What is the worst-case number of jumps required?

(b) The above method is too slow. Assuming many professors are willing to help, give an algorithm that only takes $O(\log n)$ jumps. What is the maximum number of lives it will take?

(c) The method (a) above uses only one life but is too slow, whereas the method (b) is faster but may take many lives. Give an algorithm that uses at most two professors and is asymptotically faster than that in (a).

[1] In 1979 Bill Gates, the Microsoft founder, improved this upper bound to $5n/3$, and gave a lower bound of $17n/16$. Another 30 years passed before someone else improved the upper bound further to $18n/11$. It is still far away from the best lower bound currently known, $15n/14$. Determining the optimal bound remains an open problem.

Chapter 3

Recursion and Recurrences

You should know from your programming classes that functions/methods can call themselves. Similarly, a *recursive algorithm* is an algorithm that calls itself. People new to programming often find the idea very confusing. Well, people didn't invent recursion just for fun or just to give students headaches; it is a very powerful tool in algorithm design. In fact, most of the algorithmic techniques that will be introduced in this module can be interpreted as recursion,[1] and if you manage to get your head round it, it is actually a simple idea and leads to simple proofs.

3.1 The Tower of Hanoi problem

To illustrate, we introduce the *Tower of Hanoi* problem. In this problem, there are three pegs A, B and C, and n discs of different sizes, all initially in peg A, sorted in increasing order of size. We want to move all n discs to peg C with the minimum number of moves, subject to the following rules:

- Only one disc, from the top of a peg, can be moved in each step
- At no point can bigger discs be on top of smaller ones

There is a simple recursive algorithm for the problem. It is based on the observation that, if you know how to solve the problem with $n-1$ discs, you can solve it with n discs:

1. Recursively move the top $n-1$ discs from A to B (leaving the bottom disc at A untouched, as if it is just part of the table/ground)
2. Move the bottom disc from A to C
3. Recursively move the $n-1$ discs from B to C (again leaving the bottom disc at C untouched)

See, for example, `https://en.wikipedia.org/wiki/Tower_of_Hanoi` for an animation. (Try to watch carefully to convince yourself the "recursiveness" of the solution.)

[1] See, for example, this book: Udi Manber, *Introduction to Algorithms: A Creative Approach*, ISBN 0201120372, Addison-Wesley, 1989.

Clearly, this algorithm gives a correct solution, in the sense that the moves are legal, that it terminates, and that the final configuration is what we want. But does it complete it in the minimum number of moves? What is the number of moves made?

Working out the algorithm on small values of n gives

number of discs, n	1	2	3	4	5	10	100
number of moves, $T(n)$	1	3	7	15	31	?	?

We cannot use the methods in Chapter 1 to analyse the time complexity (or in this case, the number of moves) of the Tower of Hanoi algorithm. It recursively calls itself, so its time complexity, as a function of n, also depends on itself! So first, let's try to write down a formula to relate the time complexity to itself, based on the workings of the algorithm.

Let $T(n)$ be the number of moves made by the algorithm when given n discs. The algorithm first invokes itself to move the top $n-1$ discs, thus making $T(n-1)$ moves (we don't yet know what this number is, but just carry on...); then it makes a single move to move the biggest disc; finally it invokes itself again, costing another $T(n-1)$ moves. Therefore we have the formula

$$T(n) = 2T(n-1) + 1 \tag{3.1}$$

This allows us to work out the number of moves made by the algorithm for any given n. For example, $T(6) = 2T(5) + 1 = 2(31) + 1 = 63$, since we already know that $T(5) = 31$. However, it would be very inefficient to work it out one by one like this. What if we want to find $T(100)$? Do we have to work through all 99 previous values? We want to be able to express $T(n)$ purely as a function of n, and not as a function of itself. For example, for the above formula, it turns out that the correct solution is $T(n) = 2^n - 1$. You can check with the values of the table to see it is indeed correct.[2]

Formulas like (3.1) are called **recurrences**. It expresses a function, which is usually time complexity for our purposes, as a function of itself, but with smaller arguments. It must also come with a **base case**, which is where the recursion stops, because it cannot continue indefinitely. Usually, the base case is reached when n is sufficiently small so that it is trivial to solve the problem without recursion, and the number of steps will then also be obvious. In Tower of Hanoi, we should specify the base case $T(1) = 1$ (or $T(0) = 0$). The process of obtaining an explicit solution from the recurrence is calling *solving* the recurrence.

There are two things you really should know how to do, when given a recursive algorithm: first, to write down a recurrence formula that expresses the time complexity of the algorithm; and second, to solve the recurrence.

3.2 Binary search

As another example, consider binary search (Algorithm 2.2). It can be expressed, perhaps more naturally, as a recursive algorithm. Recursive algorithms call themselves with different parameters each time; this can be done by passing a different subarray or (like below) passing

[2] Which, by the way, means the algorithm makes exponentially many moves; for example, if there are 20 discs and we make a move every second, it will take more than 12 days; and if there are 40 discs it takes close to 35,000 years! And in fact this algorithm is optimal – no other algorithm can solve the problem with fewer moves – although we will not prove it here.

the indices to indicate the range of the array to be considered. It therefore means that you need to separately specify how we begin running the algorithm; for the one below, we begin by calling Binary-Search$(A, 1, n, x)$.

Algorithm 3.1 Binary-Search(A, i, j, x): a recursive version.

Input: Array A, indices i, j of A, target x

```
1: if i > j then
2:    return "not found"    // base case
3: else
4:    mid ← round((i + j)/2)
5:    if A[mid] == x then
6:       return mid    // found
7:    else if A[mid] > x then
8:       Binary-Search(A, i, mid − 1, x)    // lower half
9:    else
10:      Binary-Search(A, mid + 1, j, x)    // upper half
11:   end if
12: end if
```

The recurrence for the number of element comparisons made by this algorithm is $T(n) = T(n/2) + 1$, since it makes one comparison first (line 5), then invokes one recursive call, of size (number of elements) half of the original. Note that despite the appearance of two recursive calls (lines 8 and 10), only one of them is invoked every time.

We will later see that the solution of this recurrence is $T(n) = O(\log n)$, which is unsurprising given that we already know the non-recursive version of binary search runs in that amount of time. To solve this, and many other similar recurrences, in the next section we introduce general methods for solving recurrences.

3.3 Methods for solving recurrences

3.3.1 Iterative substitution

Let's consider a simple recurrence:

$$T(n) = T(n-1) + n, \qquad T(1) = 1 \tag{3.2}$$

$T(n)$ is expressed in terms of $T(n-1)$, so the argument (the thing inside the bracket) gets smaller. The formula also means that we can express $T(n-1)$ in terms of $T(n-2)$, i.e.

$$T(n-1) = T(n-2) + (n-1)$$

and similarly

$$T(n-2) = T(n-3) + (n-2)$$

People often got confused by this, as they do not see that the n is a "dummy" variable. We can replace every occurrence of n by $n-1$, or indeed anything; in effect (3.2) is not one formula but many in one:

$$\begin{aligned} T(2) &= T(1)+2, \\ T(3) &= T(2)+3, \\ T(4) &= T(3)+4, \ldots \end{aligned}$$

However, please be careful that the replacement must be carried out faithfully. For example, replacing n by $\frac{n}{2}$ in the formula $T(n) = T(\frac{n}{2}) + \frac{1}{3}n^2$ gives

$$T(\frac{n}{2}) = T(\frac{n/2}{2}) + \frac{1}{3}(\frac{n}{2})^2$$

If we apply it all the way we get

$$\begin{aligned} T(n) &= T(n-1)+n \\ &= T(n-2)+(n-1)+n \\ &= T(n-3)+(n-2)+(n-1)+n \\ &= \ldots \\ &= T(1)+2+3+\ldots+n \\ &= 1+2+3+\ldots+n \\ &= n(n+1)/2 \end{aligned}$$

To apply this method, one typically goes through these steps:

1. "Unroll" a few steps of expansion. Typically, the expansion results in a trail of summation terms. It is a good idea not to add them up initially, but observe the pattern that forms.
2. From this, obtain a general expression after k "unrollings." Usually this involves the $T()$ term becoming $T(n/2^k)$ or similar, with a trail of addition with about k terms.
3. Usually, the summation trail forms either an *arithmetic progression* or a *geometric progression*, where there are known formulas for their sums (but you may need to extract some common factors out first):

$$\begin{aligned} 1+2+\ldots+n &= n(n+1)/2 \\ 1+r+r^2+\ldots+r^n &= \frac{r^{n+1}-1}{r-1} \\ 1+r+r^2+\ldots+r^n &< \frac{1}{1-r} \quad \text{if } 0<r<1 \end{aligned}$$

   (The third formula is a simplified form of the second one, because r^{n+1} becomes very small if $0 < r < 1$ and n is very large.)
4. Work out what value of k (in terms of n) makes the $T()$ term reaches the base case. Then we can substitute the base case into the $T()$ term, and any appearance of k remaining can be replaced by something in n. At this point the whole expression involves n only and is therefore "solved." Perform any final simplifications needed.

Here is a more complicated example: $T(n) = 2T(n/2) + n^2$.

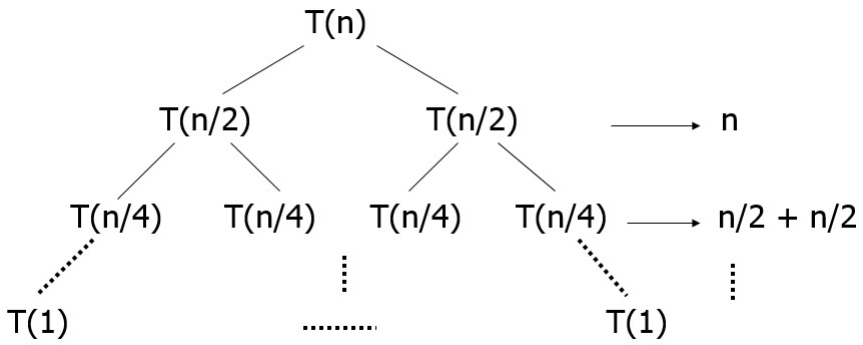
$$\begin{aligned}
T(n) &= 2T(n/2) + n^2 \\
&= 2(2T(n/4) + (n/2)^2) + n^2 \\
&= 2^2T(n/2^2) + n^2/2 + n^2 \\
&= 2^2(2T(n/2^3) + (n/2^2)^2) + n^2/2 + n^2 \\
&= 2^3T(n/2^3) + n^2/2^2 + n^2/2 + n^2 \\
&= \ldots \\
&= 2^kT(n/2^k) + (1/2^{k-1} + \ldots + 1/2 + 1)n^2 \\
&= nT(1) + \frac{1-(1/2)^k}{1-1/2}n^2 \text{ when } n/2^k = 1, \text{ i.e. } k = \log n \\
&= cn + 2(1-1/n)n^2 \\
&= 2n^2 + cn - 2n \\
&= O(n^2)
\end{aligned}$$

Note that we did not specify the base case here. In general, if we are only interested in the big-O solution of the recurrence, the base case does not matter. The above assumes $T(1) = c$ for some constant c, and the value of c has no effect on the answer.

Another common problem people have is to make mistakes with indices, brackets etc. Please make sure you are aware of the following (and apply them correctly):

$$(a/b)^p = a^p/b^p; \quad (a^p)^q = a^{pq}; \quad a^{-1} = 1/a; \quad a^0 = 1$$

The process of iterative expansion can also be represented by a **recursion tree**. The following is the recursion tree for the recurrence $T(n) = 2T(n/2) + n$:



The leaves are the base cases and the extra summation terms are on the right. The solution of the recurrence is the sum of all rightmost and bottommost terms. This may be helpful for some kinds of recurrences.

3.3.2 Induction

You should already know what **mathematical induction** is; but if not, see the appendix at the end of this chapter. It turns out that induction is very highly related to recursion; in fact

they are, in a very real sense, two sides of the same thing. It is therefore natural that induction can be used to solve recurrences.

We can use induction to solve recurrences *only if* we have a guess on the solution. For example, consider the recurrence from the Tower of Hanoi problem, $T(n) = 2T(n-1)+1, T(1) = 1$. We first "guess" that the solution is $T(n) = 2^n - 1$. This can be deduced from the table, or notice that the recurrence indicates that $T(n)$ "roughly" doubles for every n, and that $T(1) = 2^1 - 1 = 1$.

So in order to prove $T(n) = 2^n - 1$, first consider the base case. When $n = 1$, the recurrence says $T(1) = 1$, and this agrees with our guess as $2^1 - 1 = 1$. Then we consider the induction step:

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \quad \text{(by definition of the recurrence)} \\ &= 2(2^{n-1} - 1) + 1 \quad \text{(by induction hypothesis)} \\ &= 2^n - 2 + 1 \\ &= 2^n - 1 \end{aligned}$$

This proves that our guess is correct.

One has to be very careful when using induction together with Big-O. Consider the following incorrect proof. It claims to prove that the solution of the recurrence $T(n) = 2T(n/2) + n$ is $T(n) = O(n)$. The base case is trivial. In the induction step:

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2O(n/2) + n \\ &= 2O(n) + n \\ &= O(n) + n \\ &= O(n) \end{aligned}$$

The first line is just the definition; the second line is applying the induction hypothesis (to the value $n/2$); the last three lines are all properties of big-O, that it absorbs constant factors.

Why is this wrong? It is because the "constant" hidden by the big-O is no longer a constant after repeated substitutions. Had we used the substitution method, $T(n/2)$ would be reduced to $T(n/4)$, and then $T(n/8)$, ... and there are a lot of terms being added.

Indeed, if we spell out the statement we want to prove without using big-O, i.e., $T(n) \leq cn$ for some constant c, then the problem will reveal itself: $T(n) = 2T(n/2)+n \leq 2(cn/2)+n = (c+1)n$ which cannot be smaller than cn for any c!

If we instead try $T(n) \leq cn \log n$, it will go through:

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &\leq 2c(n/2)\log(n/2) + n \quad \text{(induction hypothesis)} \\ &= cn(\log n - 1) + n \quad \text{(since } \log(n/2) = \log n - \log 2\text{)} \\ &= cn \log n - cn + n \\ &\leq cn \log n \text{ for } c \geq 1 \end{aligned}$$

3.3.3 Master Theorem

Many recurrences we encounter in the analysis of algorithms are of the form

$$T(n) = aT(n/b) + O(n^d)$$

where a, b, d are constants. Because the process of solving recurrences is so tedious and error-prone, someone decided to solve it once and for all and developed a general formula. This is what is called the *master theorem*. It says:[3]

Theorem 3.1 (Master Theorem) *If $T(n) = aT(n/b) + O(n^d)$, where a, b, d are constants, then*

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{if } d < \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^d) & \text{if } d > \log_b a \end{cases}$$

We will not give a proof here, but basically it is just a big iterative substitution with different limiting cases.

> Hint: A easy way to memorise the formula is that: the answer is simply n to the power of the larger of the two: d or $\log_b a$. Unless when the two are equal, in which case we throw in an extra $\log n$ factor.

Worked example. Consider the binary search recurrence $T(n) = T(n/2) + 1$. We have $a = 1, b = 2, d = 0$ (note that $n^0 = 1$, hence $d = 0$), and $\log_b a = \log_2 1 = 0$ which is equal to d. Hence the second case applies and the answer is $O(n^0 \log n) = O(\log n)$.

Still, you need to know the more elementary ways of solving recurrences, because:

- Not all recurrences can be solved by the master theorem. For example, $T(n) = T(n-1)+1$ cannot be solved by it because $T(n-1)$ is not of the form $T(n/b)$, i.e. n divided by a constant.
- It only gives big-O answers, so if you require an exact answer you will need one of the other methods.
- To make sure you know how to solve recurrences "manually" we will sometimes explicitly ban you from using it.

3.4 Floors and ceilings

So far all our recursive algorithms and recurrences conveniently sweep under the carpet the situation when things cannot be divided "nicely." For example, in binary search, if n is even, the $n - 1$ elements (excluding the middle one) cannot be divided into two sets of exactly the same size, so the recurrence $T(n) = T(n/2) + 1$ is not exactly correct.

We first introduce some notations:

Definition 3.2 *The* **floor** *of a real number x, denoted by $\lfloor x \rfloor$, is the largest integer smaller than or equal to x. The* **ceiling** *of a real number x, denoted by $\lceil x \rceil$, is the smallest integer larger than or equal to x.*

[3] Actually what we present here is a weak form of the theorem; the full one can be found in [CLRS].

In other words, floor is rounding down and ceiling is rounding up; thus for example $\lfloor 3.7 \rfloor = 3$, $\lceil 5.2 \rceil = 6$, and $\lceil 7 \rceil = \lfloor 7 \rfloor = 7$.

So, for example, if an algorithm divides the problem into two halves and solves them recursely, it may have a recurrence like $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n)$ instead of just $T(n) = 2T(n/2) + O(n)$. However, we will almost always omit these complications; we may state that we assume n is power of 2. This makes n always divisible by 2 at any step of the algorithm. Without this, applying iterative substitution methods would be impossible. In almost all cases, it can be shown that it does not affect the result. Another way of looking at this is to consider this assumption as increasing the input size by at most a constant factor. For example, an array with 17 elements can be rounded up to 32 elements (by inserting some sentinel values at the end) to become a power of 2, and 32 is less than double of 17.

3.5 Appendix: Tutorial on mathematical induction

Mathematical induction, or simply *induction*, is a very useful way of proving mathematical statements, usually those involving positive integers. To prove that a proposition $P(n)$ is true for all positive integers n, we only need to establish that:

- $P(1)$ is true, i.e. the statement is true when $n = 1$. This is the *base case*.
- For any k, $P(k) \Rightarrow P(k+1)$, i.e., if $P(k)$ is true then $P(k+1)$ is true. Note that we are only proving the implication, i.e., under the assumption that $P(k)$ is true, we want to establish $P(k+1)$. This is called the *inductive step*, and the assumption is called the *induction hypothesis*.

The idea is that the two are combined to give a "domino" effect: since $P(1)$ is true, applying the inductive step with $k = 1$ shows $P(2)$ is true; and since $P(2)$ is true, applying the inductive step again with $k = 2$ shows $P(3)$ is true, and so on.

Worked example. Suppose we want to prove that "the sum of cubes of the first n positive integers equals the square of the sum of the first n positive integers," or in formula

$$1^3 + 2^3 + \ldots + n^3 = (1 + 2 + \ldots + n)^2$$

for all n. Since we already know that $1 + 2 + \ldots + n = n(n+1)/2$, this is equivalent to saying that

$$1^3 + 2^3 + \ldots + n^3 = n^2(n+1)^2/4$$

When $n = 1$, clearly the left hand side is 1^3 and the right hand side is $(1^2)(2^2)/4$, both equal to 1. Hence we established the base case.

Suppose we have

$$1^3 + 2^3 + \ldots + k^3 = k^2(k+1)^2/4$$

for some k. We want to use this induction hypothesis to prove that

$$1^3 + 2^3 + \ldots + k^3 + (k+1)^3 = (k+1)^2(k+2)^2/4$$

Every k in the formula is replaced by $k+1$. This means the left hand side now has an extra term (it had k terms before and now has $k+1$ terms). Now we replace all but the last term of the

chain of addition using the induction hypothesis. Then it is a matter of algebraic manipulation:

$$\begin{aligned} 1^3 + 2^3 + \ldots + k^3 + (k+1)^3 &= k^2(k+1)^2/4 + (k+1)^3 \\ &= (k+1)^2[k^2/4 + (k+1)] \\ &= (k+1)^2(k^2 + 4k + 4)/4 \\ &= (k+1)^2(k+2)^2/4 \end{aligned}$$

So we complete the induction step and hence the statement we want to prove is true by mathematical induction.[4]

Variants. There are many variants of the basic induction technique. For example the inductive step can be

$$\text{All of } P(1), P(2), \ldots, P(k) \text{ together } \Rightarrow P(k+1)$$

Furthermore, different base cases and induction steps can be combined to prove different things: for example, one can have three base cases $P(1)$, $P(2)$ and $P(3)$, and the inductive step $P(k) \Rightarrow P(k+3)$. This also proves $P(n)$ for all n. Or if we have a base case $P(1)$ and the inductive step $P(k) \Rightarrow P(2k)$, this proves $P(n)$ for all powers of 2 (i.e., $n = 1, 2, 4, 8, \ldots$)

Exercises

#3-1 **(Solving recurrences)**

Solve the following recurrences using repeated substitution, giving the answer in big-O notation. Verify your solution using the Master Theorem.

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

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

3-2 **(Mathematical induction)**

Prove the following formulas by mathematical induction on n.

(a) $1 + 2 + 3 + ... + n = n(n+1)/2$

(b) $1 + r + r^2 + ... + r^{n-1} = (r^n - 1)/(r - 1)$ for $r \neq 1$

3-3 **(Different ways of using induction)**

For what values of n is $P(n)$ true, if we have the base case(s) and induction hypothesis in each of the following?

(a) $P(2)$ is true, and $P(k) \Rightarrow P(k+2)$

(b) $P(1)$ is true, $P(2)$ is true, and ($P(k)$ and $P(k+1)$) $\Rightarrow P(k+2)$

[4] Alternatively, see a "proof by picture" in `https://en.wikipedia.org/wiki/Squared_triangular_number`

Chapter 4

Divide and Conquer

In this chapter we introduce the first of our major algorithm design technique – divide and conquer, and illustrate it with a number of practical problems, including sorting and integer multiplication.

The general principle of divide and conquer is very simple. If we don't know how to solve a big problem, break it down into smaller subproblems which hopefully are easier to solve:

- Partition a problem into different parts
- Solve the subproblems
- Combine the solutions to form the overall solution

But the distinguishing feature of this paradigm when applied to the design of algorithms is that, we partition the problem in such a way that the subproblems are just *smaller versions of the same problem*. Hence we simply apply recursion to the subproblems. We therefore do not need to care how the smaller subproblems are solved. What actually happens is that they will be reduced to even smaller subproblems, and so on...[1] Of course, we cannot divide into smaller and smaller subproblems forever; when they become small enough, we reached the *base cases* where they can be solved "trivially."

4.1 A first example: finding both max and min

Consider the following problem: given n numbers, find their maximum and minimum. Our objective is to use as few comparisons between the numbers as possible.

An obvious solution is to find the maximum and minimum separately: we already know how to find the maximum using sequential search (Algorithm 1.1), which takes $n-1$ comparisons in the worst case. Obviously the same can be applied to find the minimum, in another $n-1$ comparisons. This gives a total of $2n-2$ comparisons. Can we do better?

A moment of thought would reveal that in fact $2n-3$ comparisons are enough: after finding the maximum, there are only $n-1$ numbers left from which to find the minimum. But this

[1] This is something beginners tend to get confused; they try to "get to the bottom" by "unravelling" every level in designing the algorithm or in analysis. In fact you should not do that; this is precisely the power of recursion. (Although we will give you exercises to trace the executions to make sure you understand how recursion actually works...) Often it feels like the solution just appears as if by magic, and it might not be clear where in the algorithm was the work actually done!

improvement is tiny. Can we do still better? So let's try applying the divide and conquer paradigm. First, divide the n numbers into two equal-sized halves, in some arbitrary way. Then we recursively find the maximum and minimum within each of the two halves. The solutions of these subproblems give us the solution of the original problem: the minimum of the n numbers must be the minimum in the half that it went to, and so we only need to compare the two minima to find the global minimum. The same goes for the maxima. We therefore have the algorithm:

Algorithm 4.1 MaxMin(S)

Input: An array S with n numbers
Output: max and min of elements in S

```
1: if n == 1 then    // base case, size = 1, no comparisons needed
2:    max ← S[1]; min ← S[1]
3: else if n == 2 then    // base case, size = 2, one comparison
4:    if S[1] > S[2] then
5:       max ← S[1]; min ← S[2]
6:    else
7:       max ← S[2]; min ← S[1]
8:    end if
9: else
10:   Divide S into S1 and S2, each with half of the elements
11:   (max1, min1) ← MaxMin(S1)    // recursion
12:   (max2, min2) ← MaxMin(S2)
13:   if max1 > max2 then
14:      max ← max1
15:   else
16:      max ← max2
17:   end if
18:   if min1 < min2 then
19:      min ← min1
20:   else
21:      min ← min2
22:   end if
23: end if
24: return (max, min)
```

(Here we assume for convenience that the function returns a pair of numbers.)

Let's analyse the number of comparisons, $T(n)$, made by this recursive algorithm. Assume n is a power of 2. We have $T(n) = 2T(n/2) + 2$ since there are two recursive calls, each involving $n/2$ numbers. Afterwards we make two extra comparisons (lines 13 and 18), hence the $+2$. There are two base cases, $T(1) = 0, T(2) = 1$, corresponding to the cases in lines 1 and 3 respectively.[2] Using iterative substitution, this can be solved:

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

[2]Why do we need two base cases? Part of the reason is because if n is odd then it cannot be divided evenly. But see Exercise 4-2 for another reason.

$$\begin{aligned} &= 2(2T(n/4)+2)+2 \\ &= 2^2T(n/2^2)+2^2+2 \\ &= 2^kT(n/2^k)+(2^k+2^{k-1}+\ldots+2) \\ &= (n/2)T(2)+2(n/2-1) \\ &= 3n/2-2 \end{aligned}$$

And thus we reduced the number of comparisons from approximately $2n$ to about $1.5n$.

4.2 Merge sort

We saw some slow, $O(n^2)$ time, sorting algorithms in Chapter 2. Now let's apply this technique to sorting! Naturally, we can divide the n numbers into two halves and sort the two halves recursively. But then what? How do we combine the two sorted halves into a single sorted list (efficiently)? But assuming we have such a procedure Merge(), a sorting algorithm follows:

Algorithm 4.2 MergeSort(A)

Input: Array A of size n
Output: A sorted array B containing the same elements as A

```
if size of A == 1 then
  return A    // base case, nothing to sort
end if
B1 ← MergeSort(A[1..n/2])
B2 ← MergeSort(A[n/2 + 1..n])
B ← Merge(B1, B2)
return B
```

Now let's consider the actual merging. Imagine you have two piles of exam scripts, both sorted in student numbers, and you want to merge them into one big sorted pile. The first one should clearly come from either the top of the first pile or the top of the second pile. So you move the smaller one to the output pile, revealing a new exam script in the pile you just removed from. Then you simply repeat the procedure. So this is the merge algorithm:

Algorithm 4.3 Merge

Input: Two sorted arrays $A[1..n]$ and $B[1..n]$
Output: Sorted array C merging A and B

```
i ← 1; j ← 1; k ← 1;    // pointer to arrays A, B, C
while i ≤ n and j ≤ n do
  if A[i] < B[j] then
    C[k] ← A[i]; i++; k++;
  else
    C[k] ← B[j]; j++; k++;
  end if
end while
Copy (append) all remaining elements from A or B to C
```

Clearly, everything inside the while loop can be done in $O(1)$ time (per iteration). How many times is the while loop executed? Observe that each time the loop is executed, one of i or j is increased by exactly one. They start at 1, and one of them exceeds n when the loop finishes. Hence the loop must have been executed at most $O(n)$ times. (The actual number varies from about n to about $2n$, depending on how i and j advance; for example i may reach the end first with j hardly moved, or they may increment in an "interleaved" way.) The copying after the while loop also takes $O(n)$ time. Hence the overall time complexity is $O(n)$.

The time complexity of merge sort is therefore given by the recurrence $T(n) = 2T(n/2) + O(n)$, where the two $T(n/2)$ are the time taken by the two recursive calls and the $O(n)$ is for the merge. It therefore follows (from the Master Theorem) that $T(n) = O(n \log n)$. It is therefore more efficient than the $O(n^2)$ time sorting algorithms from Chapter 2.

Later we will see that this is in fact the best possible time complexity for sorting. However, merge sort is not really that good in practice. One reason is that it involves copying elements to a different array; in other words it is not *in-place*.

4.3 Quicksort

Sometimes the same algorithm design technique can be applied in different ways to give different algorithms for the same problem. Here we devise a different sorting algorithm using divide and conquer differently.

In merge sort, most of the computations are spent in the merge() step. The divide step is trivial. Can we find a more efficient way to merge things, perhaps at the expense of dividing more slowly? Here is an idea: instead of dividing into two halves arbitrarily, we move the "small" numbers to one end of the array, and the "large" numbers to the other end. Recursively solve these two subproblems. Once the two subarrays are sorted, they can just be left there, next to each other, and no merging is needed!

This more careful divide step we call it *partition*. More specifically, we choose a *pivot* element, around which we decide whether an element is "large" or "small." This pivot can be any element in the array, but (for the convenience of our specific variant of Partition below) we choose the last element of the array. Then we move all the small elements toward the head of the array, and the big one towards the end. The pivot should be separating the two halves.

There are numerous ways this can be done; the easiest is perhaps to copy each element to a separate array. However below we use one particular variant which is possibly better in practice. Here is the pseudocode:

Algorithm 4.4 Partition(A, p, r)

Input: Array A, indices p, r

Output: Reposition elements in $A[p..r]$ such that everything in $A[p..q-1]$ is smaller than or equal to $A[q]$, and everything in $A[q+1..r]$ is larger than $A[q]$, where q is the index of partition point returned

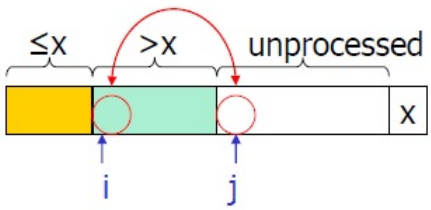
```
x ← A[r]    // pivot
i ← p − 1
for j = p to r − 1 do
  if A[j] ≤ x then
    i++
    swap(A[i], A[j])
  end if
end for
swap(A[i + 1], A[r])
return i + 1
```

Intuitively, i and j in the pseudocode maintain two regions of the array: when outside the if block, $A[p..i]$ stores the small numbers, $A[i+1..j-1]$ stores the large numbers, and the rest are unprocessed. Each time, a new number $A[j]$ is considered. If it is large, then it simply stays there as it is already adjacent to the end of the large region, and we increase the size of the large region by incrementing j (which we do anyway since we also advance to the next unprocessed element in the next step). If it is small, we have to move it, but in order to avoid shifting everything, we swap it with the first number in the large region. See the figure.

After the for loop is finished, we perform one final step to move the pivot so that it sits between the small and large regions (again without shifting the whole region).

It is easy to see that its running time is $O(n)$.

Now that we have a efficient partition procedure, the sorting algorithm follows immediately:



Algorithm 4.5 QuickSort(A, p, r)

Input: Array A, indices p and r

Output: Sorted array $A[p..r]$

```
if p < r then
  q ← Partition(A, p, r)
  QuickSort(A, p, q − 1)
  QuickSort(A, q + 1, r)
end if
```

The time complexity of Quicksort is given by the same recurrence as in merge sort, since there are two subproblems, each of size half of... wait, nope – what are the sizes of the two subproblems? There is no guarantee that the two subproblems are of the same size. It depends on what the pivot is. If the pivot is "the middle element," then indeed the two subproblems have size roughly $n/2$. *If this happens at every level*

of recursion, then we have the recurrence

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

which gives a solution of $T(n) = O(n \log n)$.

But what about an unbalanced partition? The worst situation is when the pivot is the smallest or the largest element, resulting in one subproblem having $n - 1$ elements and the other one empty. Again, if it happens at every level of recursion, then

$$T(n) = T(n - 1) + O(n)$$

which gives a solution of $T(n) = O(n^2)$. As usual, we are interested in the worst-case time complexity, so it turns out that in the worst-case, Quicksort is less efficient than merge sort and is in the same category as selection or insertion sort.

Quicksort (with our version of Partition) also has the peculiar property that its worst-case performance is triggered when the input is already sorted. This is because the largest number is chosen as the pivot, and hence the partition is the most unbalanced; the subproblem is again a sorted array and hence this goes on at every level of recursion. For this and other reasons, in practice some other variations of Quicksort are used.

But it is called Quicksort, surely it got this name for a reason?

4.3.1 Average-case analysis and randomized algorithms

Recall we discussed in Chapter 1 the distinction between worst-case and average-case analysis. In average-case analysis we need to know (or assume) the distribution of inputs. For sorting, for example, we could assume that all possible permutations of n numbers are equally likely. If we make such an assumption, then Quicksort can be shown (we will not do it here) to take $O(n \log n)$ time on average.

A somewhat related but different concept is the use of **randomization**. A **randomized** algorithm is one that uses some random values in the algorithm. Those that don't are called **deterministic** algorithms. There are two classes of randomized algorithms. *Las Vegas algorithms* only use randomization to improve the worst-case time (by avoiding the worst-case) but the output is always the same. *Monte Carlo algorithms*, on the other hand, may produce incorrect outputs with some probability as well. Here, of course we want the output to always be correctly sorted.

One typical use of randomization is to avoid pathological inputs. We can, for example, randomly permute the elements before sorting. Or we can randomly choose an element in the array as pivot (rather than the last one). This avoids us repeatedly running into the same worst-case type of input, and it can be shown that the expected (average) running time of Quicksort using a random pivot is $O(n \log n)$.

Note the following conceptual difference: in average-case analysis, we consider many different inputs, or equivalently a random distribution of inputs, and analyse the performance of a deterministic algorithm. The "averaging" or randomness comes from the inputs. Whereas in randomized algorithms, no matter which (fixed) input we consider, the algorithm gives a certain performance on average over its random choices. The randomness come from the algorithm itself.

In practice, Quicksort is one of the fastest sorting algorithms, and is widely used.

4.3.2 Comparing merge sort and quicksort

Before we move on we give a brief comparison of these two sorting algorithms. Both are based on the principle of divide and conquer; and both divide the problem into two subproblems and take $O(n)$ extra work aside from the subproblems. But they are different in several interesting aspects:

- In merge sort, the two subproblems always have equal size; but in Quicksort, the two subproblems may not have equal size.
- In merge sort, the divide step is trivial (just any way woul do), the combine is the difficult part (merge sorted lists); whereas in Quicksort, divide is difficult (find good split), but combine is trivial (just put them together).
- Merge sort has optimal worst-case time complexity $O(n \log n)$, yet it is not good in practice; on the other hand, although Quicksort has a worst-case time complexity of $O(n^2)$, its average time complexity is $O(n \log n)$ and works better in practice.

4.4 Order statistics

From the previous discussions, we see that we can make Quicksort efficient if we can guarantee a good partitioning, which we can if choose the *median* as the pivot. The median is the "middle number" of a set of numbers when ranked in increasing order; more precisely, here we define it as the $\lceil n/2 \rceil$-th smallest number in an n-element set.[3]

More generally, the k-th **order statistic** is the k-th smallest number in a set of n mumbers. For example, the minimum is the first order statistic, and the median is the $\lceil n/2 \rceil$-th order statistic.

Can we find the k-th order statistic efficiently, for some given k? For $k = 1$ or n, we know how to do that in $O(n)$ time, since this is just finding minimum or maximum. Here are some attempts for other values of k:

(1) An obvious solution is to simply sort the n numbers, then report the k-th one. This clearly takes $O(n \log n)$ time. But intuitively, we might be able to do better, since sorting does far more than what we ask for: sorting tells us *every* order statistic!

(2) Here is another solution: find the minimum, throw it away, find the minimum from the rest (which is the 2nd smallest number in the original set), throw it away... repeat until we get to the k-th smallest number. This takes $O(nk)$ time, so if k is a constant then this is linear time. However, if $k = n/2$, for example, then it is even slower than sorting.

So let's try to apply the divide and conquer method again and see what we can do. Recall that the Partition procedure (Algorithm 4.4) not only rearranges the numbers but also returns the index of the "partition point." Observe that, with this index, we know which part of the (post-partition) array the k-th order statistic is in. Formally, let $q = \text{Partition}(A, 1, n)$. Then

- If $k = q$, we are lucky and the pivot is exactly the required order statistic.

[3] There are ambiguity on what is the "middle number" when n is even, and there are alternative definitions, e.g. to take the mean of the two middle numbers.

- If $k < q$, the k-th order statistic must be on the "smaller" side, i.e. in $A[1..q-1]$. Moreover it is still the k-th smallest element in this set.
- If $k > q$, the k-th order statistic must be on the "larger" side, i.e. in $A[q+1..n]$. However, it now becomes the $(k-q)$-th order statistic there, since the smallest q numbers will be removed.

This therefore (again) allows us to reduce the problem to smaller versions of the same problem. Here is the algorithm:

Algorithm 4.6 Select(A, p, r, k)

Input: Array A, indices p, r, integer k that is $\leq r - p + 1$
Output: the k-th smallest number in $A[p..r]$

```
if p == r then
  return A[p]    // base case
end if
q ← Partition(A, p, r)
len ← q − p + 1
if k == len then
  return A[q]
else if k < len then
  return Select(A, p, q − 1, k)
else
  return Select(A, q + 1, r, k − len)
end if
```

The time complexity of Select(), like Quicksort, depends on how balanced the partitions are. In the best case, if Partition() always gives a good split, then the time complexity is given by $T(n) = T(n/2) + O(n)$, which gives $T(n) = O(n)$. (Note that we are only making one of the two recursive calls each time.) However, if it is most unbalanced at every level, then $T(n) = T(n-1) + O(n)$, giving $T(n) = O(n^2)$.

Similar to Quicksort, randomization can be used to give a better worst-case running time: if we choose a random pivot, it can be shown that the expected running time of Select() is $O(n)$.

4.4.1 Further explanation on Quicksort and Select's performances

When analysing Quicksort we introduced average-case analysis as an explanation of its practical good performance. Here we give a bit more discussion on why Select and Quicksort perform well in practice.

We know the worst-case situation is with an extremely unbalanced partition (1 vs. $n-1$). But even with a quite unbalanced partition, as long as the smaller side is a *constant fraction* of the whole, no matter how small, Select() still gives a $O(n)$ time complexity. For example, even if the partition always results in 10:90 split, the recurrence is $T(n) = T(9n/10) + O(n)$ (because the worst-case is that the next call is on the bigger part) but the solution is still $T(n) = O(n)$. Similar argument goes for Quicksort.

It turns out that we actually can find order statistics in *worst-case* linear time. It involves a more elaborate median-finding procedure which itself relies on recursion; we will not discuss

it here (see [CLRS]). And, equipped with this modified version to find order statistics, we can use this worst-case $O(n)$ time algorithm to find the median first, and use it as the pivot in Partition(). This will give a truly worst-case $O(n \log n)$ time Quicksort algorithm.[4]

4.5 Integer multiplication

4.5.1 Bit model on the time complexity of basic arithmetic operations

So far we have always assumed that any standard arithmetic operations take constant time, irrespective of the sizes of the numbers concerned. This works if they fit into machine word size (e.g. 32 bits) - it is just one CPU instruction. However, for very large numbers, such as those with thousands of bits, then this is no longer a reasonable assumption. Such computations are actually quite common, for example in cryptography. An alternative, closer to reality model, assumes that each bit operation takes constant time. So adding two bits takes $O(1)$ time but adding two n-bit numbers may take, say, $O(n)$ time – this is indeed the case for the straightforward method. We add the two least significant bits, then move on to the two bits one position to the left, adding the carry bit if needed. Thus there are at most two additions (one of them for the carry bit) per bit position. For example, in base-10 we have: (for more intuitive illustration, we will often switch between base-2 and base-10 examples; it does not affect the time complexity by more than a constant factor)

```
  2 4 1 7 5
+   3 4 8 9
-----------
  2 7 6 6 4
```

What about multiplication? If anybody still remembers, this is how we multiply two n-bit (or n-digit) numbers back in school:

```
        3 4 1 5
x       1 2 1 3
--------------
  3 4 1 5
    6 8 3 0
      3 4 1 5
      1 0 2 4 5
--------------
  4 1 4 2 3 9 5
```

(The exact way of presentation may be different from what you learnt in school, but the basic idea should be the same.) We produce n intermediate rows, one per the multiplication of one digit from one of the numbers, to all n digits of the other number. Thus each row takes $O(n)$ time to produce. They have to be added up: there are up to $2n$ columns, and n rows, so each column of addition takes $O(n)$ time and in total it takes $O(n^2)$ time to get to the answer.[5]

[4] This worst-case version of Select() calls itself once, then calls Partition(), then calls itself again. If you think too much about it, it starts to get dizzy: let's call Quicksort Q, Partition P, and this modified select S'. So, Q calls S' (to find the median), then P, then Q twice. But S' itself calls S', P and then S'...

[5] So, if you have always thought at school that multiplication is more complicated and time-consuming than addition, you were right even though you had no concept of asymptotic complexity back then...

4.5.2 Divide and conquer multiplication

Assume n is a power of 2. We try to apply divide and conquer and break down a n-bit by n-bit multiplication into some $(n/2)$-bit by $(n/2)$-bit multiplications. First we need to understand how the bit (digit) values relate to the actual values of the numbers.

Suppose A is an n-bit number with bits $x_1, x_2, \ldots, x_n$, a is an $n/2$-bit number with bits $x_1, x_2, \ldots, x_{n/2}$, and b is an $n/2$-bit number with bits $x_{n/2+1}, x_{n/2+2}, \ldots, x_n$. Then

$$A = 2^{n/2}a + b$$

For example (switching to base 10), the 6-digit number 123456 is related to its two 3-digit components 123 and 456 by

$$123456 = 10^3(123) + 456$$

So, if we want to multiply an n-bit number $A = 2^{n/2}a + b$ with another n-bit number $B = 2^{n/2}c + d$, we can rewrite this as

$$A \times B = (2^{n/2}a + b)(2^{n/2}c + d) = 2^n(ac) + 2^{n/2}(ad + bc) + bd$$

Each of ac, ad, bc and bd are $n/2$-bit by $n/2$-bit multiplications. Once we got the answers to these subproblems (by recursion), the final answer can be computed from the above formula using some extra additions and multiplications to 2^x for some x. Note that multiplication to 2^x (and the calculation of 2^x itself) are not actually multiplications; it merely appends x zeros to the number (just like in base 10, where $10^x \cdot y$ is simply y appended by x zeros), which can be done in computers in bit shifts. These all take only $O(n)$ time. Thus the time complexity of this divide and conquer algorithm is given by the recurrence

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

Unfortunately, its solution is $O(n^2)$, so after going through all this trouble, it is not faster than the basic algorithm!

4.5.3 Karatsuba's algorithm

But it turns out that we need only one trick to get a better complexity. Observe that we do not need ad and bc separately, just their sum $ad + bc$. Furthermore

$$ad + bc = (a + b)(c + d) - ac - bd$$

(simply expand the bracket on the right hand side to see this.) We are going to compute ac and bd anyway; thus, instead of two multiplications, we can only make one more multiplication, the product of $a + b$ and $c + d$. This gives us the value of $ad + bc$ using the above formula. Of course, it involves some extra additions and subtractions, but they take only linear time, and we are spending linear time on other parts anyway.

Since we now have three subproblems instead of four, the recurrence becomes

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

and its solution is $O(n^{\log_2 3}) \leq O(n^{1.59})$. Thus we got a faster algorithm![6]

[6]The idea can be further extended to give even faster algorithms, by dividing each number into more than two parts. But this approach is not currently the best one. For a problem as basic as this, it is not known what the optimal time complexity is; the currently best algorithms are all extremely complicated and have time complexity given by some very complicated functions.

There is, actually, some glossing-over in our above discussion. Each of a, b, c, d are $n/2$-bits, but $a+b$ and $c+d$ may have $n/2+1$ bits after the addition. Thus the product $(a+b)(c+d)$ is, strictly speaking, an $(n/2+1)$-by-$(n/2+1)$ bit multiplication. However, it can be shown that this does not affect the asymptotic time complexity.

Exercises

#4-1 **(Merge sort)**

Show the execution of the merge sort algorithm when the input is [5, 3, 8, 7, 1, 4, 2, 6].

4-2 **(MaxMin revisited)**

In the analysis of MaxMin (Section 4.1) the recurrence was stopped at the base case $T(2) = 1$ and gave an answer of $T(n) = 3n/2 - 2$. Had we used the $T(1) = 0$ base case instead (and ignored the base case at $n = 2$), the solution would become

$$\begin{aligned} T(n) &= \ldots \\ &= 2^k T(n/2^k) + 2(2^k - 1) \\ &= nT(1) + 2(n-1) \text{ when } k = \log n \\ &= n(0) + 2(n-1) \\ &= 2n - 2 \end{aligned}$$

which is different! What is happening?!

#4-3 **(Partition)**

Show the execution of the Partition algorithm when the input is [5, 3, 8, 7, 1, 4, 2, 6]. Assume the last element of the array is chosen as the pivot.

#4-4 **(Quicksort)**

Show the execution of the Quicksort algorithm when the input is [5, 3, 8, 7, 1, 4, 2, 6].

#4-5 **(Order statistics)**

Show how Select() finds the 3rd order statistic from A = [5, 3, 8, 7, 1, 4, 2, 6]. Show the parameters in each recursive call of Select().

*4-6 **(Finding second minimum)**

In Section 4.4 we discussed a simple way of finding the k-th order statistic, by repeatedly finding a minimum and throwing it away, which takes $O(nk)$ time. Using this approach, the second smallest element can be found using $2n - 3$ comparisons. But this is not the best possible. Describe informally how to find the second smallest element using at most $n + \log n - 2$ comparisons. Assume n is a power of 2. (*Hint*: consider a knock-out tournament.)

#4-7 **(Integer multiplication)**

Consider the multiplication of 1234 and 5678 using Karatsuba's algorithm. Use a diagram to show all the subproblems.

*4-8 **(Maximum contiguous subarray)**

Given an array $A[1..n]$ with n elements, we want to find the *maximum-sum subarray*, i.e., the subarray $A[i..j]$ starting with $A[i]$ and ending with $A[j]$ such that the sum $A[i] + A[i+1] + \cdots + A[j]$ is maximum over all possible subarrays. For example, if $A = [2, -10, 6, 4, -1, 7, 3, -8]$, then the maximum-sum subarray is $[6, 4, -1, 7, 3]$ with sum 19. Clearly, if all numbers in the array are positive, the maximum occurs when we take the whole array. We define an empty array (with 0 elements) to have a sum 0. So for example if all numbers in A are negative, the maximum sum is 0.

(a) A direct approach is to try all possible subarrays, compute their sums and take the maximum. Show how to do this in $O(n^2)$ time. (*Hint*: look at Exercise 1-4 again.)

Consider using divide and conquer and divide A into two subarrays $A_1 = A[1..n/2]$ and $A_2 = [n/2 + 1..n]$. Note that the maximum-sum subarray of A must either (i) lie entirely within A_1, or (ii) lie entirely within A_2, or (iii) cross the boundary between A_1 and A_2.

(b) Give an $O(n)$ time algorithm to find the maximum-sum subarray crossing the boundary. (*Hint*: "grow" on each side of the boundary.)

(c) Hence give an $O(n \log n)$ time algorithm for the problem. Assume n is a power of 2.

Chapter 5

Lower Bounds

Recall the coin weighing problem in Chapter 1:

> Given: 8 coins, one of which is counterfeit and is lighter; a pan balance (that shows only which side is heavier)
> Goal: to find the counterfeit coin using as few weighings as possible

We described some algorithms that can solve this problem in worst-case 4 weighings or 3 weighings, respectively. These are *upper bounds*, and we always want to improve (decrease) the upper bound (find better algorithms). Can we do better? What about 2 weighings? 1 weighing?

In fact, 2 weighings are sufficient, and here is the algorithm, represented as a tree:

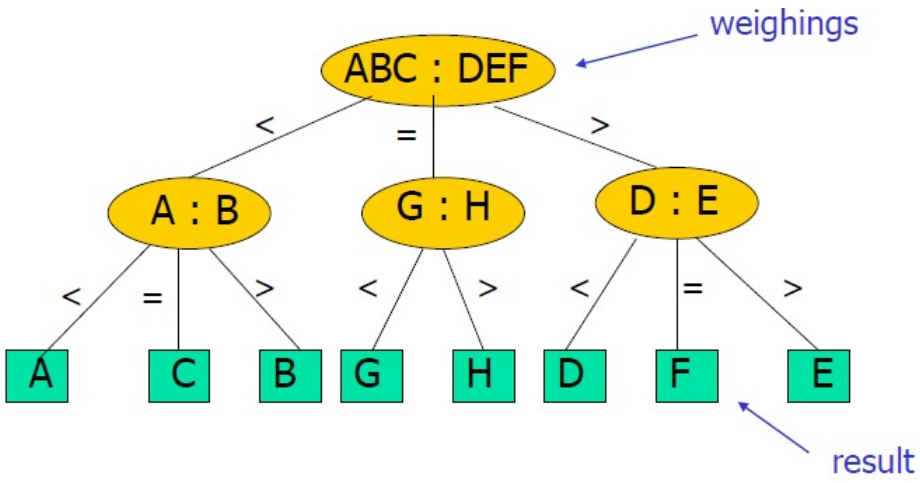


Figure 5.1: An algorithm that takes only two weighings.

Remember, we never end our pursuit for better algorithms until we found the optimal one. But how do we know we cannot do even better? This is about proving lower bounds.

5.1 Decision trees

Recall the definition of *lower bound*: the number of operations necessary to solve a problem by any algorithm. We are going to prove a lower bound of 2 weighings for this coin weighing problem. In other words, no algorithm can find the counterfeit coin in fewer than 2 weighings. Thus the above algorithm must be optimal.

Here is the rough argument. The counterfeit coin can be any one of the eight coins, which initially are completely indistinguishable, and in the end we have to identify the correct one. So there are 8 different configurations/outcomes we have to distinguish between. Each weighing differentiates at most 3 possibilities, because it only gives 3 different outcomes. So, 1 weighing gives 3 outcomes, 2 weighings gives $3 \times 3 = 9$ outcomes, and so 1 weighing must not be enough and 2 weighings potentially might be.

More formally, we can represent the actions of an algorithm using a **decision tree**. Fix a problem P that we want to solve. All algorithms for P (those we concern ourselves with here...) are representable by a tree. *Different algorithms correspond to different trees*: some taller, some fatter, some more skewed, some more balanced, ... but there are some properties that must be *the same for any tree that solves* P, because it is confined by the definition of P and the computational model. These are:

- *The number of leaves*: this corresponds to the total number of different outcomes that we want to distinguish.
- *The branching factor*, the number of branches coming out of a node: this corresponds to the number of different possible actions after one "step."

Each particular path from the root to a leaf of a given decision tree corresponds to one particular sequence of actions taken by that algorithm, and the number of steps taken (i.e. time) is equal to the length (number of edges) of this path. In Figure 5.1, every possible path has the same length of 2; but in general different paths may have different lengths. The worst-case number of steps taken by the algorithm, therefore, corresponds to the *height* of the tree, i.e., the longest of all paths, or equivalently how "deep" the deepest node is. As we want algorithms with good (small) worst-case number of steps, we want algorithms that have a small height. In other words we prefer fat and shallow trees rather than slim and tall trees. Finding a lower bound on the worst-case number of steps taken by an algorithm is equivalent to finding a lower bound on the height of *any* tree that satisfies the two fixed properties (number of leaves and branching factor).

It is in fact easy to establish a relationship between those two fixed quantities and the height of a tree. Consider a tree with branching factor b, height h and n leaves. At the first level (under the root) there can be at most b nodes. Each of these can have up to b children of their own, so at the second level there are up to b^2 nodes. In general, level k has at most b^k nodes. To have n leaves we therefore need $b^h \geq n$, which means

$$h \geq \lceil \log_b n \rceil$$

where the ceiling is because the height must be an integer. This formula allows us to easily work out a lower bound simply by finding out what the total number of different outcomes (n) and the branching factor (b) are.

In our coin weighing problem, we have $n = 8$ and $b = 3$, so the lower bound is $\lceil \log_3 8 \rceil = 2$, so two weighings are necessary and our algorithm is therefore optimal.

Note: you are not expected to know the exact value of logs like $\log_3 8 \approx 1.8927$, but you should know its integer range (i.e. between 1 and 2 in this case because $3^1 = 3$ and $3^2 = 9$).

Here are some common misconceptions:

- Whenever you are asked to prove a lower bound, **do not** draw me a tree like the one in Figure 5.1. I cannot emphasise this enough. A tree shows one specific algorithm, but a lower bound is an argument about all possible algorithms, i.e., all possible trees. Showing one tree with a height of 2 says nothing about the height of other trees.
- Having a lower bound of 2 does not mean there exists an algorithm that takes 2 steps. All it proves is that 1 step is not sufficient, and 2 steps might be sufficient.
- Decision trees are not the only method to prove a lower bound. Sometimes the bound it manages to prove is rather weak, i.e., not the best possible, and other techniques may prove stronger bounds.

5.2 Optimal algorithm for n coins

As usual, we are not really that interested in solving a problem with a fixed input size. So let's generalise the problem to n coins instead of 8. Since the number of different outcomes is n and the number of branches is 3, the decision tree argument gives a lower bound of $\lceil \log_3 n \rceil$.

Is there an algorithm that matches this bound? It is easy to generalise our previous algorithm. Assume n is a power of 3. Simply divide the set of coins into three equal parts, and weigh two parts against each other on the balance. If one part is lighter, it contains the counterfeit coin; if they are equal, the counterfeit coin is in the third part. Recursively apply the algorithm to that part. Or in pseudocode: (start the algorithm with $\mathrm{CW}(A, 1, n)$)

Algorithm 5.1 CW(A, i, j)

Input: Array A of coins; indices i and j

```
if i == j then
  return i    // base case
end if
k ← (j − i + 1)/3
Weigh A[i..i + k − 1] against A[i + k..i + 2k − 1]
if A[i..i + k − 1] lighter then
  CW(A, i, i + k − 1)
else if A[i + k..i + 2k − 1] lighter then
  CW(A, i + k, i + 2k − 1)
else    // equal
  CW(A, i + 2k, j)
end if
```

The number of weighings $T(n)$ made by this algorithm can be analysed by solving the recurrence $T(n) = T(n/3) + 1, T(1) = 0$:

$$\begin{aligned} T(n) &= T(n/3) + 1 \\ &= T(n/9) + 1 + 1 \\ &= \dots \\ &= T(n/3^k) + k \\ &= T(1) + \log_3 n \\ &= \log_3 n \end{aligned}$$

Thus, for n being a power of 3, it matches the lower bound.

> Question: the lower bound is $\lceil \log_3 n \rceil$ and the upper bound is $\log_3 n$, which means if $\log_3 n$ is not an integer, the lower bound is higher! But this is impossible. What happened?

5.3 Lower bound for sorting

Recall we studied $O(n \log n)$ time sorting algorithms. We want to show that they are optimal by giving a matching lower bound.

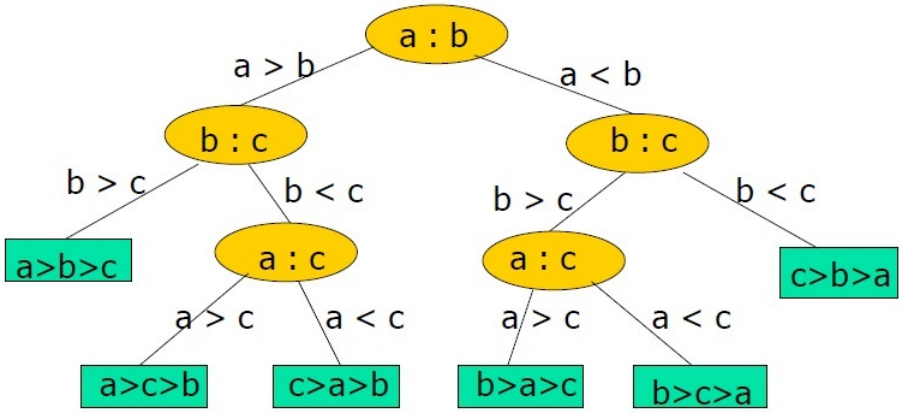
A trivial lower bound is $\Omega(n)$: clearly, you need to read all the inputs to give a correct sort, and that takes at least linear time. But this is weaker than what we want.

We now show an $\Omega(n \log n)$ lower bound. Actually, what we do is to focus on the number of comparisons between elements, and we prove a lower bound in the *comparison-based model*, where the only allowed operation on the elements is to compare two of them directly. It is a natural enough model; since we are ordering elements, it seems natural that comparing two elements is the key operation. However, there are other algorithms that do not fall under this model and seemingly beat the $O(n \log n)$ barrier (all with caveats; see [CLRS].) We prove a lower bound on the number of comparisons needed by any algorithm under this model.

We first make an assumption that the n values to be sorted are all different (no equal values). This is a special case of the more general scenario, and a lower bound for a more

restricted situation is also a valid lower bound for the more general scenario, since the more general the situation is, the more difficult it is to give efficient algorithms. The reason we do this is that it simplifies the counting in the next step.

We need to work out the number of different outcomes (leaves). This is not just the number of elements n itself; rather, it is the number of ways that n elements can be permuted, since this is what a correct sorting algorithm needs to be able to distinguish. For example, for 3 elements there are 6 outcomes ($a > b > c, a > c > b, b > a > c, b > c > a, c > a > b, c > b > a$). This figure shows one possible algorithm to distinguish them:



For n elements, the number of permutations is given by $n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1 = n!$ (n factorial). The branching factor is 2, since under our assumption the equal case cannot happen and each comparison only gives 2 possible outcomes ($>, <$). So the height of the decision tree, and a lower bound of sorting, is given by $\lceil \log_2 n! \rceil$.

This doesn't look like the $n \log n$ we are hoping for, but we are going to show that $\lceil \log_2 n! \rceil$ is indeed $\Omega(n \log n)$. It is easy to show that $\lceil \log_2 n! \rceil$ is $O(n \log n)$:

$$n! = 1 \times 2 \times \ldots \times n \leq n \times n \times \ldots \times n = n^n$$

Hence $\log n! \leq \log n^n = n \log n$. But this is not the direction we want: we want to show the number of steps is *at least* some constant times $n \log n$. Instead, consider the following:[1]

$$\begin{aligned} n! &= 1 \times 2 \times \ldots \times (n/2) \times \ldots \times n \\ &\geq 1 \times 1 \times \ldots \times 1 \times (n/2) \times \ldots \times (n/2) \\ &= (n/2)^{n/2} \end{aligned}$$

We get the second line by replacing each term in the first half of the chain $(1, 2, \ldots, n/2)$ with 1, and each term in the second half $(n/2 + 1, \ldots, n)$ with $n/2$. Clearly this can only make the product smaller. Yes, we are giving up *a lot*, but it turns out we are still left with enough: $n/2$ copies of $n/2$. Therefore $\log n! \geq \log(n/2)^{n/2} = (n/2) \log(n/2) = (n/2)(\log n - 1) = \Omega(n \log n)$.

[1] There are more "direct" ways to lower-bound $n!$, for example using something called *Stirling's approximation*. You are not required to know that, or to come up with this "creative" way of lower-bounding the value of $n!$.

Exercises

5-1 **(Coin weighing variants)**

In each of the following variations to the coin weighing problem, give optimal upper and lower bounds:

(a) There are 9 coins, and exactly one of them is counterfeit.

(b) There are 10 coins, and exactly one of them is counterfeit.

(c) There are 8 coins, and at most one of them is counterfeit.

(d) There are 9 coins, and at most one of them is counterfeit.

In all cases the counterfeit coin is lighter than genuine ones.

5-2 **(Sorting with equal values)**

Consider the problem of arranging three numbers a, b, c in increasing order. Some of the numbers may be equal and this need to be identified as well. The basic operation is the comparison of two numbers, which tells you which one is larger or they are equal.

(a) List all the possible outcomes. (Some possible outcomes are $a < b < c, b = c < a, a = b = c$, etc.)

(b) What is the lower bound on the number of comparisons needed?

(c) Give an optimal algorithm for solving this problem. Represent your algorithm using a decision tree.

*5-3 **(Heavier/lighter variant)**

Suppose there are 12 coins, exactly one of which has slightly different weight than the rest, but we do not know whether it is heavier or lighter than the others. We want to identify this coin, and to determine whether it is heavier or lighter than other coins. Again we are only using a pan balance.

(a) Show that 3 weighings are always necessary in the worst case, no matter what algorithm is used.

(b) Show that 3 weighings are not enough in the worst case if there are 13 coins, no matter what algorithm is used. (Note that the decision tree argument by itself is not enough to prove it!)

(c) Going back to 12 coins, is it actually possible to have an algorithm using at most 3 weighings in the worst case?

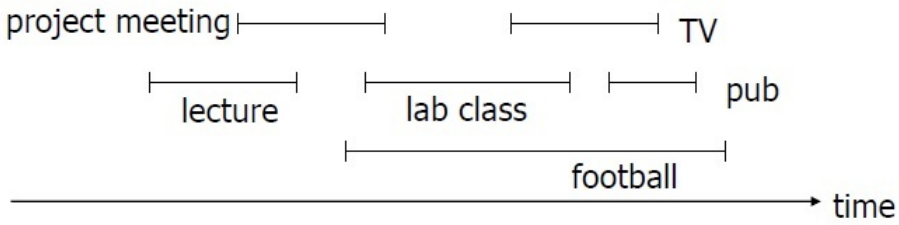
Chapter 6

Greedy Algorithms

In this chapter we introduce the second of our major algorithm design technique – greedy algorithms. In Chapter 8 we will see further problems that can be solved with the greedy principle.

6.1 An interval selection problem

Consider the following problem. You are given a set of activities, each with a starting time and a finishing time. The goal is to participate in as many activities as possible, without conflicting times. (Note: we are maximising the *number* of activities, and not, for example, their total length. Often different objectives lead to different solutions and require different algorithms.) For example, the following figure shows six activities.



For such a small example, it is easy to see – by visual inspection – that the best solution is lecture + lab class + pub, for a total of 3 activities. But can we have a systematic way (in other words, design an algorithm) to solve the problem?

Before we describe possible algorithms, let's give a slightly more abstract way of describing the problem. Each activity can be considered as an *interval*. An interval i has a starting time $s(i)$ and a finishing time $f(i)$. Two intervals i and j are in conflict if $s(i) < s(j) < f(i)$ or $s(j) < s(i) < f(j)$. Given a set of intervals, our objective is to identify the largest subset of intervals such that no two of them are in conflict.

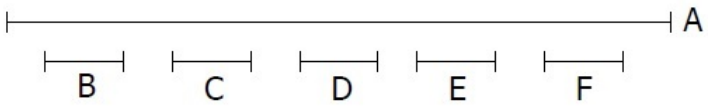
Very often, an abstract formalisation of the problem is useful. Problems arising from different contexts may turn out the same or very similar. Of course, in this particular case, it is pretty much just a change of wording from "activity" to "interval," but there are more complicated transformations (or *reductions*), and sometimes completely different looking problems turn out to be equivalent.

Now let's consider some natural attempts to get an algorithm:

Algorithm 6.1 Earliest Arrival First

Repeatedly choose the "leftmost" interval (i.e., the one with the earliest starting time) that is not in conflict with already chosen ones.

This seems a natural choice: basically we just take on whatever that arrives if we are free. But it is not a correct algorithm. On the previous example, it chooses lecture and then football, which is not optimal. In fact, its solution can be very bad: consider this input



The algorithm would choose the single long interval, but the optimal solution chooses all the $n - 1$ short intervals.

Algorithm 6.2 Shortest Interval First

Repeatedly choose the shortest interval that is not in conflict with already-chosen ones.

This also sounds reasonable: a short interval seems less likely to be in conflict with other intervals. But it is easy to come up with examples to show that it is not optimal; for example two long, adjacent (non-overlapping) intervals and a short interval overlapping both of them.

Algorithm 6.3 Earliest Finishing First

Repeatedly choose the interval with the earliest finishing time (again, as long as it is not in conflict with already-chosen ones).

This sounds very similar to, but is different from, Algorithm 6.1 (which chooses the earliest *starting* time). The idea here is to leave as much time as possible for other intervals, which again might be reasonable.

It can be checked that it produces the correct solution for all the previous examples. Of course, returning the correct solution for a few examples does not mean the algorithm is always correct. Even if you spent a long time and failed to find an example that makes it fail, it does not mean there isn't one. To prove that an algorithm is correct is therefore often a non-trivial task. But to prove it is not correct, you merely need to demonstrate one **counterexample**.

It turns out that Algorithm 6.3 is indeed correct. We will revisit Algorithm 6.3 in a moment, but first let's take a look us this "class" of algorithms we have just proposed.

6.2 Principles of greedy algorithms

The above are examples of **greedy algorithms**. The basic principle of greedy algorithms is to iteratively make the "best" choice as seen at that point, based on optimising some "local" criterion that may or may not lead to long-term optimal solutions (hence greedy). It builds its

solutions incrementally. Typically, after making a choice, a smaller subproblem arises, which can then be solved by the same algorithm recursively (or iteratively).[1]

Greedy algorithms have the following properties:

- They are usually simple to describe and easy to understand.
- They usually have a fast running time (e.g. $O(n \log n)$).
- They don't often produce optimal solutions, because of their "short-sightedness" (imagine playing chess by only maximising the "advantage" you gain at every move, without looking further ahead.)
- They sometimes give "good" solutions in the sense that they are not too far away from the optimal solution, e.g. by a constant factor, but sometimes the solutions can be terribly bad.

We have seen examples where seemingly reasonable greedy algorithms are not correct, and some others are in fact correct. It is therefore not always obvious that a given greedy algorithm is correct or not, and we need proofs of their correctness. (Exercise 6-3 gives another example of such a problem.)

6.3 Analysis of earliest finishing first

Time complexity. First we give a somewhat more precise description of Algorithm 6.3:

Algorithm 6.4 IntervalSelection($S[1..n]$)

Input: A set S of intervals, each with starting time $s()$ and finishing time $f()$; S is sorted in increasing order of $f()$
Output: A, the set of chosen intervals

```
A ← S[1]
k ← 1
for i = 2 to n do
  if s(S[i]) ≥ f(S[k]) then
    A ← A ∪ {S[i]}
    k ← i
  end if
end for
return A
```

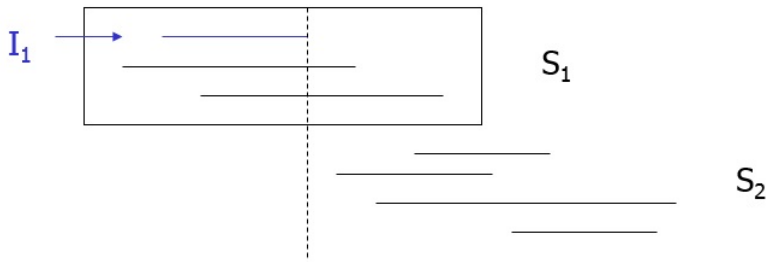
There actually are some subtleties in the above pseudocode. The variable k indexes the last interval added to A. This allows the algorithm to avoid naively checking the new interval with every other interval in A for overlapping. Each time a new interval $S[i]$ is being considered, it checks if $s(S[i]) \geq f(S[k])$, which if true would mean $S[i]$ does not overlap $S[k]$ (and also does not overlap any other interval in A – since they are all "in front of" $S[k]$) and so $S[i]$ can be chosen. Otherwise, if $s(S[i]) < f(S[k])$, then $S[i]$ must overlap $S[k]$. (It cannot be "wholly

[1] You might notice that this is rather vague, and indeed we are not really *defining* what greedy algorithms are. More precise definitions require more mathematical concepts such as *matroids*; see e.g. [CLRS].

before" $S[k]$ and not overlap with it, because that means $f(S[i]) < f(S[k])$ and $S[i]$ would have been considered before $S[k]$ in the sorted order.)

Next we analyse the running time of the algorithm. Sorting the intervals by finishing time takes $O(n \log n)$ time. The for loop is executed $O(n)$ times, each doing only a constant amount of work (ignoring the issue of data structure for A). Therefore, in total it takes $O(n \log n)$ time.

Correctness. Finally we formally prove the correctness of the algorithm. We do this by induction on the number of intervals, n. The base case $n = 0$ is trivial. For the general case with n intervals, let I_1 be the interval with the earliest finishing time. We split the n intervals into two groups: S_1, those that start before I_1 finishes; and S_2, those that start after I_1 finishes. See the figure below.



Observe that any feasible solution – in particular, the optimal solution S^* – can choose at most one interval from S_1. And choosing I_1 is "safe," because it is not in conflict with any interval in S_2. In other words, if S^* chose some other interval in S_1, it can be replaced with I_1 with no harm. Similarly, if S^* does not include anything from S_1, it can add I_1 with no harm. S^* is then left with a smaller set of intervals, S_2. The greedy algorithm also picks I_1, and since everything in S_1 overlaps with I_1, it will also next choose intervals from S_2. Thus both S^* and the greedy algorithm pick I_1 and then find a solution from S_2, but by induction, the greedy algorithm gives an optimal solution for S_2! Hence our algorithm must return the optimal solution for the input with all n intervals.

6.4 The knapsack problem

Consider the following problem: you are given a set of objects, each of weight w_i and value v_i, and a knapsack with a maximum carrying capacity of W. The goal is to find the set of objects with the maximum value that fits into the knapsack (i.e., have total weight at most W). For example, given three items: (2kg, \$60), (3kg, \$100), (5kg, \$120), and with $W = 6$kg, the optimal solution is to pick the first and second item, giving a total value of \$160.

Here are some "sensible" greedy algorithms for the problem. One can pick items in increasing order of weights (repeatedly add the smallest-weight item until nothing can fit into the knapsack); or pick items in decreasing order of values (might make sense to pick the most valuable items first). It is easy to come up with counterexamples to show that neither of them always give optimal solutions.

A perhaps more sensible algorithm is to compute the value/weight ratio (the "density," i.e. value per unit weight) for each object, and pick items in decreasing order of density. But it is also not a correct algorithm. Here is a counterexample: (1.01kg, \$60), (1.01kg, \$60), (2kg, \$100), $W = 2$kg.

In fact, no one knows of any simple greedy algorithm that solves this problem efficiently and correctly[2]. However, if we assume the items can be divided into arbitrary *fractional* sizes and that a fraction of an item gives value proportional to its weight (consider the items to be liquids, gold sand, etc.), then there is a greedy algorithm that always gives optimal solutions. In fact it is the one that picks items in decreasing order of density:

Algorithm 6.5 Fractional knapsack

```
Sort the objects in descending order of value/weight ratio
TotalWeight ← 0
for object i = 1, 2, ... do
  if TotalWeight + w_i ≤ W then
    add object i to knapsack
    TotalWeight ← TotalWeight + w_i
  else
    add a fraction of object i that makes TotalWeight equals W
    return
  end if
end for
```

Using the same example as before, with $W = 6$kg and the 3 items sorted in decreasing order of density:

Weight	Value	Density
3kg	\$100	\$33.33/kg
2kg	\$60	\$30/kg
5kg	\$120	\$24/kg

The algorithm takes all of (3kg, \$100), all of (2kg, \$60), and 1kg from (5kg, \$120). The total value it gets is $\$(100 + 60 + 120/5) = \$184$.

We will not give a full proof of correctness here. In the greedy algorithm, the vector of quantities (as a fraction) of items chosen, in descending order of value/weight ratio, is always $(1, 1, \ldots, 1, x, 0, 0, \ldots, 0)$ where $0 \leq x \leq 1$. (For the above example this vector is (1, 1, 1/5).) So, roughly speaking, if another solution picks less than 1 for some of the high-density items (and a higher fraction in some other items), we can always exchange for higher density items to increase the total value while keeping the same total weight.

[2]This problem is actually *weakly NP-complete*; we will not go into the details here.

Exercises

6-1 **(Knapsack)**

Consider a simpler version of the knapsack problem where the value of an object is simply its weight, i.e. $v_i = w_i$ for all i. You cannot take part of an object. For each of the following greedy algorithms, either show that it returns an optimal solution, or give a counterexample to show it doesn't. Make the counterexamples as bad as you can.

(a) Repeatedly choose the maximum value object.

(b) Repeatedly choose the minimum weight object.

*6-2 **(More interval selection algorithm)**

Consider this algorithm for interval selection:

Repeatedly choose the interval that overlaps with the smallest number of other intervals; if there is a tie, choose one arbitrarily. Once an interval is chosen, all intervals it overlaps with (and that interval itself) are removed from future consideration (both for being picked and for the purpose of counting overlaps).

Give a counterexample to show that this does not always return an optimal solution.

6-3 **(Coin changing)**

The UK coin system consists of coins of the following values: 1, 2, 5, 10, 20, 50, 100 and 200 (in pence). We want to make a certain amount of money using the minimum number of coins. For example for 47p we use two 20p, one 5p and one 2p for a total of four coins. A natural greedy algorithm is to choose the maximum-value coin which is same as or below the target value, subtract this from the target value to form a new target value, and repeat the procedure until target = 0.

This greedy algorithm gives optimal solutions for the UK system, but not for all systems. Give a set of coin values and a target amount that the greedy algorithm does not give the optimal solution.

Chapter 7

Elementary Graph Algorithms

Graphs are very important mathematical objects in computer science. Its usefulness comes in its ability to model many situations that involve "relations" between "entities," such as road networks, computer networks, hyperlink relation between web pages, social acquaintances, and many more less obvious ones.

Most of the remaining part of this module is related to graphs. In this chapter we recap on basic graph terminologies and introduce a few elementary algorithms.

7.1 Basic graph-theoretic concepts

A graph G is an ordered pair (V, E), where V is the set of vertices, and E is the set of edges joining the vertices. We usually use n to denote number of vertices, and m the number of edges. **Undirected graphs** are graphs where edges have no directions; in **directed graphs** edges have directions. The **degree** of a vertex is the number of edges with that vertex as an endpoint. For directed graphs we may wish to distinguish between *in-degree* and *out-degree*, the number of edges going into or out of a vertex. A **path** is a sequence of edges where consecutive edges in the sequence share a vertex. A **cycle** is a path with the same starting and finishing vertex. A **simple** graph is one where there are no multiple edges between the same pair of vertices (in the same direction), and no edges that go back to the same vertex (self-loop). An undirected graph is **connected** if any two vertices are reachable by a path. For the case of directed graphs the issue is more complicated, and will be discussed in Section 7.4. A **tree** is a connected graph with no cycles.

Unless otherwise stated, all graphs we consider are simple and connected.

Two simple graph properties. We state two simple properties that will be useful later.

Proposition 7.1 *For a connected, undirected graph,* $n - 1 \leq m \leq n(n-1)/2$.

The minimum $n - 1$ is attained when the graph is just a line (path); each new vertex added to it requires an edge. The maximum occurs when it is a complete graph, i.e., there is an edge between any two vertices, so the number of edges is $\binom{n}{2}$. (Another way to prove this is: every vertex has $n - 1$ incident edges and there are n vertices, but each edge is counted twice.)

This simple formula also states an important fact: the number of edges is always between linear ($\Theta(n)$) and quadratic ($\Theta(n^2)$) in the number of vertices. This allows us to classify graphs

into *sparse* graphs, where there are few edges (say $m = \Theta(n)$), and *dense* graphs where they are many edges (say $m = \Theta(n^2)$).[1]

Proposition 7.2 *The sum of degrees of all vertices equals twice the number of edges.*

The proof of this is extremely simple. Each edge $e = (v_i, v_j)$ contributes 1 to the degree of v_i and 1 to the degree of v_j. So, as we add up the degrees of all vertices: $deg(v_1) + deg(v_2) + \ldots + deg(v_n)$, each edge contributes twice to this sum.

7.2 Representation of graphs

Two data structures are commonly used to represent graphs. We want to know their space usage, and the time complexities of common operations.

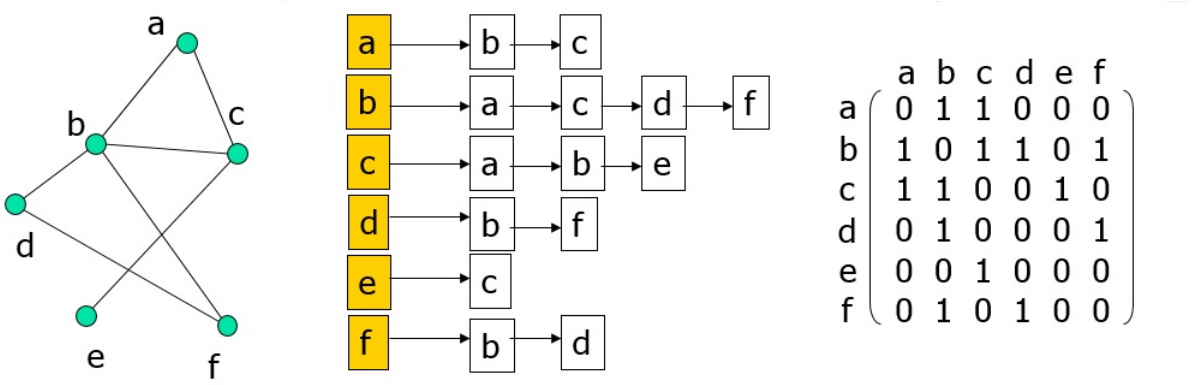


Figure 7.1: Left: an example graph. Middle: adjacency list. Right: adjacency matrix.

Adjacency list. It consists of n lists, one for each vertex. Each is a linked list of all the neighbours of that vertex.

It uses $O(n + m)$ space: there are n head nodes, and the total number of non-head nodes is equal to the total degree of the vertices, which we know (Proposition 7.2) is $2m$. It is therefore relatively good for sparse graphs: graphs with fewer edges take up less memory.

To list all the neighbours of a node, simply traverse the list, in $O(1)$ time per neighbour.

To check whether two nodes are adjacent, it takes time proportional to the degree of one of the vertices, by traversing its list and looking for the other vertex.

Adjacency matrix. It is a 2-dimensional n by n array, where the (i, j)-th entry is 1 if there is an edge from vertex i to vertex j, and 0 otherwise.

Clearly, it takes $O(n^2)$ space, irrespective of how many edges there are; which is therefore (comparatively) good for dense graphs.

Listing all the neighbours of a vertex takes $O(n)$ time, by scanning the corresponding row of the matrix and identify the entries with 1s.

[1] This is not a precise definition...

Checking whether two given nodes are adjacent can be done in $O(1)$ time, simply by checking whether that matrix entry is 1 or 0.

Both representations can be naturally generalised to directed graphs and weighted graphs (for example putting the edge weight instead of 0/1 in the adjacency matrix).

7.3 Graph traversal

A fundamental operation on graphs is to explore it: starting at some vertex, visit all the vertices/edges in a systematic way. We want to do it systematically, rather than just wander around randomly, so as to avoid repeating or missing parts of the graph. Graph traversals have many important applications, for example in game tree search in AI.

The two main approaches to graph traversals are breadth first search (BFS) and depth first search (DFS).

7.3.1 BFS

Breadth-first search, as the name suggests, prefers breadth over depth. It finishes visiting vertices close to the starting vertex before visiting those further away. More precisely, vertices are visited in order of their distance (number of edges) from the starting vertex: first those at one hop, then two hops, and so on. Imagine a wavefront spreading out from the starting vertex (see Figure 7.2).

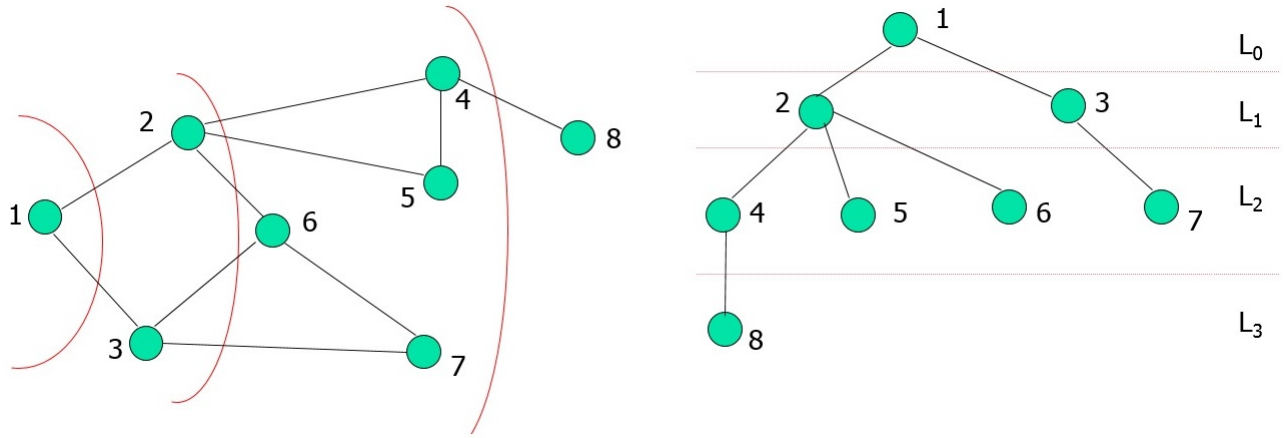


Figure 7.2: Left: a graph with an "extending wavefront." Right: BFS tree.

We can describe the action of BFS in high-level pseudocode:

Algorithm 7.1 BFS, high-level

Input: Graph G, starting vertex s

```
i ← 0
L_0 ← {s}    // layer 0
while not all nodes explored do
  for each edge (u, v) where u in L_i and v not in L_j, for some j ≤ i do
    add v to L_{i+1}
  end for
  i++
end while
```

Each vertex will then be assigned a level (the L_i's) which is its distance from s. Note that, for example, node 7 is at level 2, because it was discovered via 1-3-7, and not the longer path 1-2-6-7.

The order and the way the vertices are discovered encodes a parent-child relationship among the vertices: if a vertex v is first discovered while visiting u (as u's neighbour), we can view v as the "child" of u. A **BFS tree** is a tree that represents this relationship. It also shows the levels of nodes of a BFS traversal. See Figure 7.2.

We can also describe BFS in more detail, specifying the data structure used. It turns out that BFS can be easily implemented using a queue. Recall that a queue is a FIFO data structure (Chapter 2). Each time a new vertex is visited, its neighbours are added to the end of the queue (if not already inserted). This way, we ensure that the order we visit the vertices are consistent with what BFS requires. Note that no particular order is specified when inserting the neighbours of one vertex into the queue. We also need an additional array to keep track of whether a vertex was discovered already.

Algorithm 7.2 BFS using a queue

Input: Graph G, starting vertex s

```
Discovered[s] ← true, Discovered[u] ← false for all other u
Enqueue(Q, s)
while Q not empty do
  u ← Dequeue(Q)
  for each edge (u, v) do
    if Discovered[v] == false then
      Discovered[v] ← true
      Enqueue(Q, v)
    end if
  end for
end while
```

Running it on the graph in Figure 7.2, the contents of the queue changes as follows:

```
[1] -> [2 3] -> [3 4 5 6] -> [4 5 6 7] -> [5 6 7 8] -> [6 7 8] -> ...
```

What is the running time of BFS? The while loop executes n times (each node is enqueued exactly once and dequeued exactly once). For each iteration of the while loop, the inner for loop seems like it will execute at most m times, one per neighbour. Thus it gives a bound of

$O(nm)$. It is correct but we can get a tighter analysis. Consider the *total* number of for loop iterations over the entire course of the algorithm (not just one iteration of the while loop). Each vertex is enqueued and dequeued exactly once. When it is dequeued we check all its neighbours, which (using an adjacency list) can be done in time proportional to the number of neighbours. Thus the total number of iterations is equal to the sum of degrees of all vertices, but we know (Proposition 7.2) that it is $2m$. Enqueuing and dequeuing takes at most $O(n)$ time. Therefore in total it takes $O(n + m)$ time, which is linear in the size of the graph.

7.3.2 DFS

Another way of traversing a graph is called **depth-first search**. Here the idea is to go as deep into a graph as possible until a dead end is reached, at which point we backtrack and visit some other branch. Essentially, you are just excited to meet new neighbours and every time you find a new one, you go to find their own neighbours and so on, ignoring the old ones (until later).

The following is a recursive way to implement this idea correctly.

Algorithm 7.3 DFS, recursive version

Input: Graph G, starting vertex s
  Mark all nodes as unexplored
  Call DFS(G, s)

  Procedure DFS(G, u)
  Mark u as explored
  for each edge (u, v) **do**
    if v is unexplored **then**
      DFS(G, v)
    end if
  end for

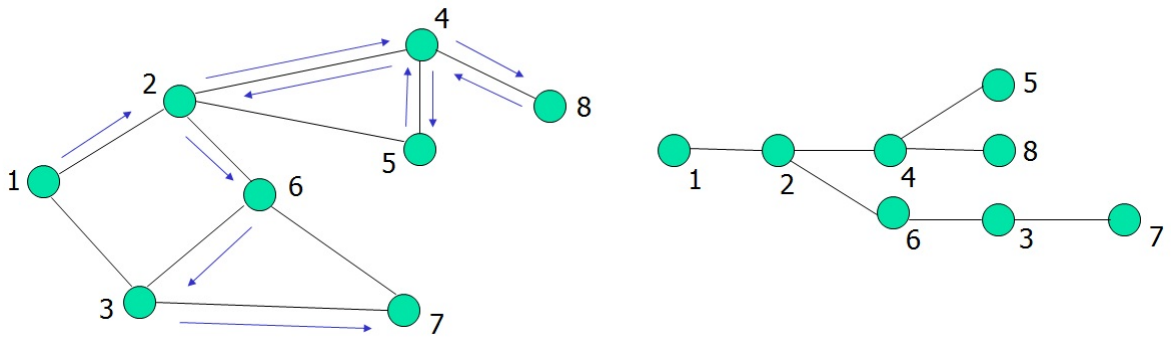


Figure 7.3: Left: A possible DFS traversal of the same graph. Right: DFS tree.

Those older neighbours that you know are there but haven't met yet, you keep them "in the back of your head" to be revisited later. It turns out that a stack is the right data structure for

this task. The algorithm is almost exactly the same as BFS, just replacing the queue with the stack:[2]

Algorithm 7.4 DFS using a stack

```
Set Explored[u] ← false for all nodes
Push(S, s)    // S is a stack
while S is not empty do
  u ← Pop(S)
  if Explored[u] == false then
    Explored[u] ← true
    for each edge (u, v) where Explored[v] == false do
      Push(S, v)
    end for
  end if
end while
```

Running it on the graph in Figure 7.3, the contents of the stack changes as follows: (here we deliberately insert vertices in reverse numerical order into the stack so that they pop out in the "natural" order; as in BFS, no specific order is prescribed by the algorithm)

```
[1] -> [3 2] -> [3 6 5 4] -> [3 6 5 8 5] -> [3 6 5 8] -> [3 6 5] ->
[3 6] -> [3 7 3] -> [3 7 7] -> [3 7] -> [3] -> []
```

The running time of DFS is analysed similarly as BFS. Essentially, each execution of the for loop corresponds to an edge, and hence it was executed only $O(m)$ times over the course of the algorithm. The total runtime is again $O(m + n)$.

Similar to BFS, we can construct a DFS tree (Figure 7.3).

7.4 Connected components

If an undirected graph is connected, a BFS or DFS starting at any vertex will reach all other vertices. However, if the graph is not connected, the traversal will not reach everything; those reachable vertices (together with the starting vertex) are in the same **connected component**. An undirected graph that is not connected can always be subdivided into a number of connected components that together cover the entire graph.

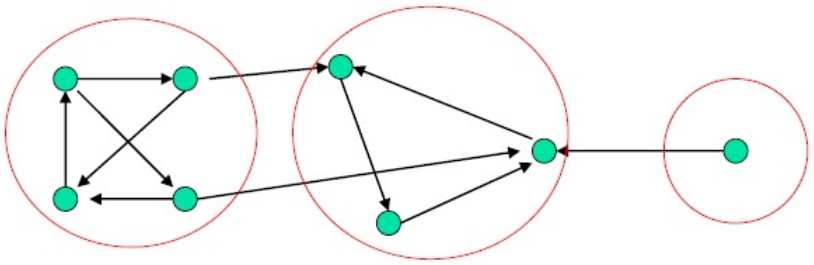
So, to check for connectedness or identify the connected components, simply start traversal at an arbitrary vertex and see if it reaches all vertices. If not, then restart the traversal at some vertex not yet reached. Since the traversal of each connected component takes time linear to the size of that component, the total time this procedure takes is still linear, i.e. $O(m + n)$.

For directed graphs, since edges have directions, if there is a path from u to v, it does not mean there is also a path from v to u. Hence the issue of connectedness is more complicated. We can define connectedness by ignoring the edge directions, and if this underlying undirected

[2] You might notice that there is another difference: the line that assigns true to the Explored[] array is in a different place (hence also the different name, Discovered[] vs. Explored[]). This means the same vertex can be in the stack multiple times. This is necessary to reflect correctly the order the vertices are processed. For example, (in this particular traversal) node 3 is "discovered" very early on as a neighbour of 1, but is "explored" as a child of 6. There are, actually, further subtleties in this issue, which we would rather not go into here.

graph is connected, we say the directed graph is **weakly connected**. But we can define another type of connectivity: a directed graph is **strongly connected** if any two vertices u and v are *mutually reachable* by some paths, i.e., there is a path from u to v and a (separate) path from v to u. If a directed graph is not strongly connected, then we can similarly define the **strongly connected components** (SCC) of a graph as the subgraphs, all of which themselves are strongly connected.

For example, the graph below is not strongly connected, because for example those inside the middle circle cannot reach any vertex outside that circle. But it has three SCCs (represented by the three circles). Within each SCC, the vertices are all reachable from one other (though the path from u to v and from v to u may well be very different).



Given a directed graph, we want to test whether it is strongly connected, and if not, find all its SCCs. A (directed) traversal as before, starting at some vertex u, will not work on its own, because even if u can reach all other vertices, it does not say anything about the reachability from others to u.

A naive approach is to simply perform n directed traversals, one starting from each vertex. If all traversals report "yes," in that they all can reach all vertices, then any two vertices must be mutually reachable, and thus the graph is strongly connected. If at least one of them report "no" then the graph is clearly not strongly connected.

However, this is obviously n times slower than the undirected one. In fact, with some tricks, two traversals are sufficient! Here is the algorithm for testing strong connectivity:

1. Pick any vertex s. Run BFS on G starting from s.
2. Run BFS on G^{rev} starting from s, where G^{rev} is a graph obtained by reversing every edge of G.
3. Report no if some nodes not reachable in either traversals, yes otherwise.

Obviously, if the algorithm reports "no" in Step 3, some vertex is not reachable from some other vertex and therefore G is not strongly connected. The interesting case is when the algorithm reports "yes." Consider any two vertices u and v. We have not directed tested the reachability between them, but we know from Step 1 that s can reach both u and v; we also know from Step 2 that both u and v can reach s. Therefore, u can reach v by going via s ($u \rightsquigarrow s \rightsquigarrow v$) and similarly v can reach u via s ($v \rightsquigarrow s \rightsquigarrow u$). Thus any u and v must be mutually reachable.

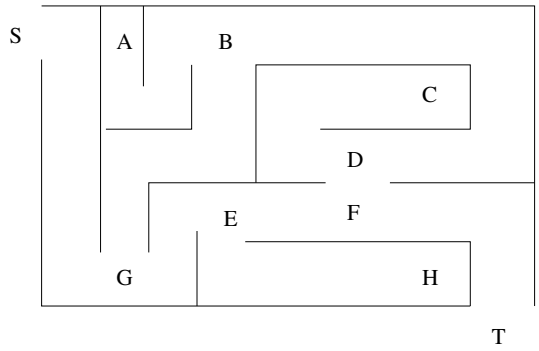
The running time of this algorithm is still $O(m+n)$, as its simply does BFS twice. Reversing the graph can also be done in $O(m+n)$ time. To find the actual SCCs, note that the set of

nodes reachable by both traversals is an SCC containing s. Hence this identifies one SCC. The remaining ones can be found one by one similarly. (Note however that this may not be linear time. A more careful approach is required, which is not covered here.)

Exercises

#7-1 **(Graph traversal)**

The figure below shows a maze where S is the entrance and T is the exit. Show a BFS traversal of the maze by giving the BFS tree and the contents of the queue after each step. Repeat using DFS and a stack.



*7-2 **(The celebrity problem)**

In a party there are n people. We call a person a *celebrity* if everyone in the party knows him/her but s/he knows no one (other than him/herself). The problem is to identify the celebrity if there is one, or report that there are none. (There cannot be two or more celebrities – why?) You are only allowed to ask questions of the form: "Does A know B?"

(a) Give a straightforward way to confirm whether a particular person is a celebrity or not, using at most $2(n-1)$ questions.

(b) From (a) it follows trivially that we can solve the problem in $O(n^2)$ questions. (How?) However we can do much better. Give an algorithm that uses at most $3(n-1)$ questions. (!!)

Most algorithms on graphs take $O(n^2)$ time or more when the input is given as an adjacency matrix, simply because we have to read the entire matrix. However this is not always true. Given the adjacency matrix representation of a directed graph, we want to determine whether the graph has a *sink*, i.e. a vertex with $n-1$ incoming edges and no outgoing edges.

(c) What is the relationship with the celebrity problem? What is an efficient algorithm for this problem?

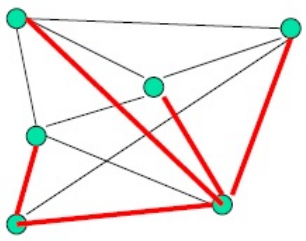
Chapter 8

Greedy Algorithms on Graphs

In this chapter we consider two important graph problems that can be solved in a greedy way: minimum spanning trees and shortest paths.

8.1 Minimum spanning trees

Suppose you are given a map with a number of cities, and you want to build a set of roads that connects them so that all cities are reachable one-another. Obviously there are many ways to do this, and we want the total length of the roads to be as small as possible. (Note that this does not necessarily give the shortest length between a given pair of cities.) For example, in the figure here, the red (thick) edges are a possible solution to connect all the nodes, although not necessarily the one with the shortest total length.



Let's formulate the problem in a more abstract, graph-theoretic model. We are given a connected, undirected graph G, where edges have weights (which, for example, represent distances, but they don't have to be), and the goal is to find a subset of edges that connect all vertices with the minimum total weight. This is called the **minimum spanning tree** (MST) problem.

To explain the name: the solution we want must be "spanning" all vertices; and it must be a tree (i.e., it contains no cycles) because, if there is a cycle, we can remove one edge in the cycle and all vertices remain connected, and the resulting solution is better (assuming all edge weights are positive). Thus the solution required is a *spanning tree*.

So how do we find an MST? Can we apply greedy algorithms we learned in Chapter 6 to this problem?

8.1.1 Kruskal's algorithm

Let's consider the following natural idea: repeatedly add the shortest unused edge. But we must also take care not to form cycles; so we only add an edge if it does not form a cycle with already chosen edges, i.e., it must connect two different connected components. This is called **Kruskal's algorithm**. At a high level, it can be described by this pseudocode (there are many details to be filled in later):

Algorithm 8.1 Kruskal's algorithm

Input: Graph G
Output: A minimum spanning tree T of G

```
E ← sorted list of edges (by weight)
T ← G without edges
while T has fewer than n − 1 edges do
  Remove first edge e = (u, v) from E
  if u, v are in different components in T then
    add e to T
  end if
end while
```

Consider the example in Figure 8.1. The edges are added to T in this order: (a, g), (c, d), (b, g), (c, g), (d, e), (a, f). Edges like (a, b) are considered but not added. Note that during the algorithm there are multiple disconnected parts of the tree that will eventually be connected.

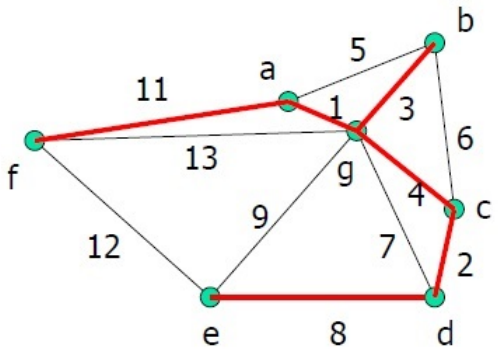


Figure 8.1: Example graph for Kruskal's algorithm.

Correctness. Recall that the correctness of greedy algorithms is not always obvious. We need to prove that this algorithm does always produce the MST. It clearly produces a spanning tree (it has $n - 1$ edges so must span all vertices); the question is whether it is the minimum one. We can prove that by contradiction. The following is a sketch of the proof.[1]

Suppose there is a graph G where Kruskal's algorithm produces a tree T^K, the actual minimum spanning tree is T^*, and T^K and T^* are not the same. Therefore, there must be an edge e in T^* that is not in T^K. Edge e separates T^* into two parts, S_1 and S_2. Let x and y be the endpoints of e in S_1 and S_2, respectively. Since e is not in T^K, T^K must connect x and y via another path P. In this path P, there is at least one edge e' connecting S_1 and S_2. See Figure 8.2.

This edge e' must have smaller weight than e, since T^K adds edges in increasing order of weights and chose e' ahead of e. Hence we can remove e from T^* and add e' to T^* to get another spanning tree with smaller weight; this contradicts the optimality of T^*.

[1] See if you can find the details we glossed over...

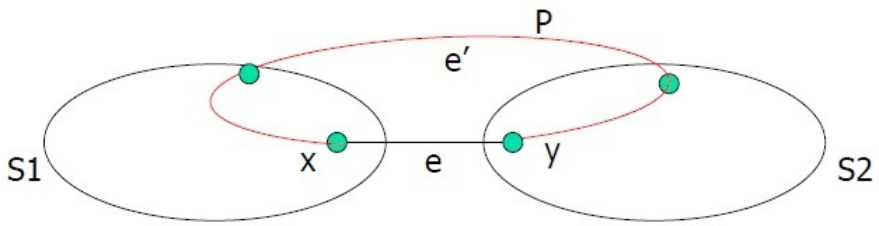


Figure 8.2: Correctness of Kruskal's algorithm.

Running Time. Sorting the m edges by weight takes $O(m \log m)$ time, which is the same as $O(m \log n)$ (recall that $m = O(n^2)$, so $\log m = O(2 \log n)$). Inside the while loop, the time complexity depends on two important operations:

1. Checking whether two vertices are already connected (i.e., in the same component): This is performed $O(m)$ times in total, over all executions of the loop, since we test each edge one by one.

2. Connecting two vertices (components) together when an edge is added: this is performed $O(n)$ times in total, over all executions of the loop, since the final MST has $n - 1$ edges.

Note that the above is counting the number of times the steps are executed; we are yet to find out the time complexity of performing each individual step. So how to perform each of these operations efficiently? To answer this question, we take a brief detour to look at something called *disjoint set data structures*.

8.1.2 Disjoint set data structures

Suppose we want a data structure for storing a number of sets. Each set contains elements from a common *ground set*. The contents of the sets may change over time, but any two sets never have overlapping elements, i.e. they are *disjoint*. The data structure should support the following operations efficiently:

- Find(u): given an element u, return a "name" or ID of the set containing u. Often, a simple name is just some element in the set, also called "set representatives." (Though it has to be the same one in repeated calls, i.e., the representative cannot change unless the contents of the set has changed.)

- Union(u, v): merge the two sets that contain u and v, if they are not already in the same set. We can without loss of generality assume that u and v are already the set representatives (not just any arbitrary element in the set), since we can always take an extra Find() to get them (and often it will have been done anyway).

These data structures are also called *union-find data structures*. Assuming we have such data structures, we can use them to support Kruskal's algorithm as follows. The ground set is the set of vertices. At any stage of the algorithm, each disjoint set represents the vertices connected by some MST edges. Initially each vertex begins in its own set. To check whether

an edge $e = (u, v)$ should be added, i.e., whether u, v are not in the same component, just check whether Find(u) = Find(v). If not, we add that edge, which means we need to union the disjoint sets with Union(u, v).

For example, with Figure 8.1 again, the disjoint sets evolve as follows:
Initially: $\{a\}\{b\}\{c\}\{d\}\{e\}\{f\}\{g\}$
Add (a, g): $\{a, g\}\{b\}\{c\}\{d\}\{e\}\{f\}$
Add (c, d): $\{a, g\}\{b\}\{c, d\}\{e\}\{f\}$
Add (b, g): $\{a, b, g\}, \{c, d\}\{e\}\{f\}$
...
Below we consider some possible union-find data structures and their time complexities.

Attempt 1: Array. In this approach we simply use an array A to keep track of the set names of the elements: $A[i]$ stores the set name of element i. For example, if the disjoint sets are {1,2,7} {3,4} {5} {6}, then A could be [1, 1, 3, 3, 5, 6, 1].

With such an array, Find(i) is easy: just return $A[i]$, which can be done in $O(1)$ time. But Union(i, j) is more difficult: it needs to change all entries that currently have the value of either $A[i]$ or $A[j]$ to a same value. For example, if we perform Union(1,3) on the above example, A needs to become [1, 1, 1, 1, 5, 6, 1]. The entire array needs to be scanned, hence it takes $O(n)$ time.

Attempt 2: Linked list. Put all elements in the same disjoint set in a linked list. The element at the head of the list is the set representative.

Find(u) requires $O(n)$ time, as we have to go through all lists to find where u appears. Union(u, v) requires stitching the two lists together, which can be done in $O(1)$ time (if we use doubly-circular linked lists, for example).

Attempt 3: Array-and-linked-list. As the name implies, this is basically combining the previous two approaches. We use a 3-by-n array to keep track of a number of things for each element. The first row contains the set name (as in Attempt 1); the second row contains the pointer of next element in the set, which is implicitly a linked list representation; and finally, for reasons to be explained later, we also want to keep track of the size of the set, which is stored in the third row. (Note that in the following example, the element names are not stored; they are assumed to be $\{1, 2, \ldots, n\}$ and the i-th column is for element i.)

To illustrate how it is updated, consider initially all 7 elements are in their own set. Then we have

name	1	2	3	4	5	6	7
next	0	0	0	0	0	0	0
size	1	1	1	1	1	1	1

Here 0 denotes end of the list (null pointer). After Union(1,7), we have

name	1	2	3	4	5	6	1
next	7	0	0	0	0	0	0
size	2	1	1	1	1	1	/

$1 \to 7$

At the right of the table we visualise the linked lists recorded in the second row (single-element lists are not shown for clarity). Note that it is only a visualisation for your understanding; it does not need to be separately recorded as it is already in the array.

Similarly, after Union(3,4):

name	1	2	3	3	5	6	1
next	7	0	4	0	0	0	0
size	2	1	2	/	1	1	/

$1 \to 7, 3 \to 4$

After Union(2,7):

name	1	1	3	3	5	6	1
next	2	7	4	0	0	0	0
size	3	/	2	/	1	1	/

$1 \to 2 \to 7, 3 \to 4$

When we merge two sets of unequal sizes, as in the above, we apply the **weighted-union heuristic**: we only spend time *proportional to the size of the smaller set*. This requires some trickery: we cannot scan the entire array to change the set names (as in Attempt 1); and when we join the linked lists we must not spend time traversing the long list to find its end. Hence we use a somewhat special procedure:[2]

1. Start with the head of the longer list. (We know which is longer since we have the size field.)
2. Update the size field of the head to the combined size.
3. Change the next pointer to point to the head of the shorter list.
4. Travel along the shorter list (i.e., use the "next" value to identify the next column), modifying the names (first row) of those elements along the way.
5. When we reached the end of the shorter list, set the next value to where the head were pointing to originally (i.e., the second element in the longer list).

As a more complicated example, consider the next step Union(3,7):

name	1	1	1	1	5	6	1
next	3	7	4	2	0	0	0
size	5	/	/	/	1	1	/

$1 \to 3 \to 4 \to 2 \to 7$

8.1.3 Running time of Kruskal's with array-and-linked-list

With this new data structure, Find(u) is still in $O(1)$ time. Union(u, v) is also still $O(n)$ time in the worst case, since we may need to update a lot of entries (e.g. merging two lists of length $n/2$).

However, we can give a better analysis to show that, although each Union() on its own may take $O(n)$ time in the worst case, all $n-1$ unions together take at most $O(n \log n)$ time. The argument is as follows: Every time a union occurs, only entries in the smaller set are modified. Afterwards, the size of the combined set is at least double that of the original. Therefore, the sizes increase from $1, 2, 4, 8, \ldots, n$, and this process has at most $\log n$ steps (Exercise 1-2); hence each entry can be modified at most $\log n$ times.

The overall running time of the algorithm is thus $O(m \log n)$ for sorting, $O(m)$ for all the $O(m)$ Finds, and $O(n \log n)$ for all the $n-1$ Unions. Their sum is thus $O(m \log n)$.

[2]This is by no means the only possible way, but we will adopt this one here.

8.1.4 Prim's algorithm

There are actually other greedy ways of finding MSTs. One such idea is to "grow" the MST from a certain vertex by picking the minimum-weight "outgoing" edge. This adds a new vertex to the (now bigger) component, and we repeat until every vertex is connected. This is called **Prim's algorithm**. For example, in Figure 8.1, if we start at a, then we first pick the shortest edge going out from a, which is (a, g). Now the component has two vertices $\{a, g\}$, and the shortest edge connecting one of them to the outside world is (g, b), which is added to the tree. This then carries on until all vertices are connected.

We will not prove its correctness here. Note that at any time, there is only one partially built MST, unlike in Kruskal's whether there are multiple disconnected ones (in other words it has a forest).

We need to be able to find the minimum outgoing edge efficiently. A naive approach is to check all (remaining) edges to find the next minimum; this takes $O(mn)$ time in total since you have to perform this $n - 1$ times and each time you find the minimum from at most m edges. But we can do better. We keep track of the best distance (so far) of each vertex outside the growing component (S) from within S. Each time a new vertex u is added to S, we only need to check those vertices who are neighbours of u, and see if the new edges going out from u improve their best distances (i.e., if $d(u, v)$ is smaller than $D[v]$ for u's neighbour v).

Algorithm 8.2 Prim's algorithm

Input: Graph G where $d()$ stores the edge weights; starting vertex s

```
1: for each vertex v do
2:    D[v] ← d(s, v)    // initialise
3: end for
4: S ← {s}
5: while S ≠ V do
6:    find u in V − S with minimum D[u]
7:    add u to S
8:    for each v in V − S do
9:       D[v] ← min(D[v], d(u, v))
10:   end for
11: end while
```

Worked example. The following tables show the execution of the algorithm for the graph in Figure 8.1, with f as the starting vertex. Initially

v	a	b	c	d	e	g
$D[v]$	11	∞	∞	∞	12	13

The smallest entry is 11, which means we connect to vertex a and now consider its outgoing edges. For example the distance to b is improved, while that to e isn't. The updated values are

v	a	b	c	d	e	g
$D[v]$	/	5	∞	∞	12	1

We pick the new smallest value, 1, and consider outgoing edges of g, and update the table again. This repeats until all vertices are processed.

The running time of Prim's algorithm depends on the data structure we use to support two key operations used by the algorithm:

- Find the minimum D value (line 6): this is performed $O(n)$ times.
- Change the D values (line 9): this is performed $O(m)$ times. Line 9 in fact only needs to be considered for neighbours v of u. Thus the for loop takes time proportional to the degree of u, and the total number of times line 9 is executed is equal to the sum of degrees of all vertices, which is $2m$.

If we simply use an array to store D, finding minimum takes $O(n)$ time, and updating a value takes $O(1)$ time. Thus the total time is $O(m \cdot 1 + n \cdot n) = O(n^2)$. But in the following we will introduce the *heap* data structure, which can support either of these operations in $O(\log n)$ time. So the total time is $O(m \log n + n \log n) = O(m \log n)$.

Which of these two (array or heap) is better therefore depends on the number of edges: for sparse graphs where $m = O(n)$, the heap is better, but for dense graphs where $m = \Theta(n^2)$, the array is better.

8.1.5 The heap data structure

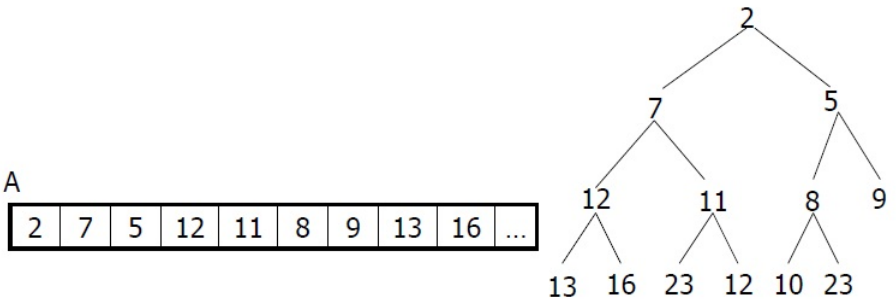
A **heap**[3] is an abstract data structure that maintains a set of elements and supports three operations efficiently: insertion, deleting the minimum, and updating the values of elements. Conceptually, it stores the elements in a *complete binary tree*, such that the elements obey a special property. In more detail:

- A binary tree is *complete* if each level is full except possibly the last, and that nodes at the last level are filled left to right.

  A complete binary tree is *balanced*, i.e., the depth of any two leaves differ by at most 1. As a result, the height of an n-node heap is $O(\log n)$.

- The *heap property* requires that the element at a node must not be larger than either of its children.[4] Note that the order of the two children is distinguishable, but there is no requirement that the smaller child has to be on the left.

Although conceptually a heap is a tree-based data structure, it can easily be represented as an array $A[1..n]$, where all the elements are simply laid in a breadth-first left-to-right order (see figure below), and all the parent-child relationships can be identified in $O(1)$ time: Parent($A[i]$) = $A[\lfloor i/2 \rfloor]$, Left-child($A[i]$) = $A[2i]$, Right-child($A[i]$) = $A[2i+1]$.



[3] If you have heard this term from memory management in programming languages, no it is a completely different thing.

[4] This is for min-heaps; it is possible to have the opposite type, max-heap, where the parent must not be smaller. Unless otherwise specified, all heaps here are assumed to be min-heaps since this is what we need.

Here is how the heap supports each of the operations in $O(\log n)$ time (See Figure 8.3):

1. **Decrease value**: (For our purposes we only need to handle the reduction of values, but a heap can also handle increase of values in a similar way.) Updating a value is easy: just write to that array entry. But once a value is reduced, it may become smaller than its parent, violating the heap property. If that happens, we swap that element with its parent. But this "bubble up" may need to happen again at the next level since the element may still be smaller than its new parent. This keeps going until either it is not smaller than its parent, or it reaches the root. Each "swap" takes $O(1)$ time and since the height of the heap is $O(\log n)$, the total time is also $O(\log n)$.

2. **Insertion**: We first put the new element at the next available position (i.e., to the right of the rightmost node at the bottommost level, unless it is full in which case we start a new level). This new element may violate the heap property with its parent, so we perform a similar "bubble up" process, again in $O(\log n)$ time.

3. **Delete minimum**: Finding the minimum is easy: it is always at the root. But we cannot just remove it and leave an empty space. What we do is to move the last (bottommost level, rightmost) element to the position of the root first; this new root may now violate the heap property, with one or both of its children. We then "sift down" the violations by swapping. Note that if it is smaller than both its children, the smaller one should be swapped with the parent.

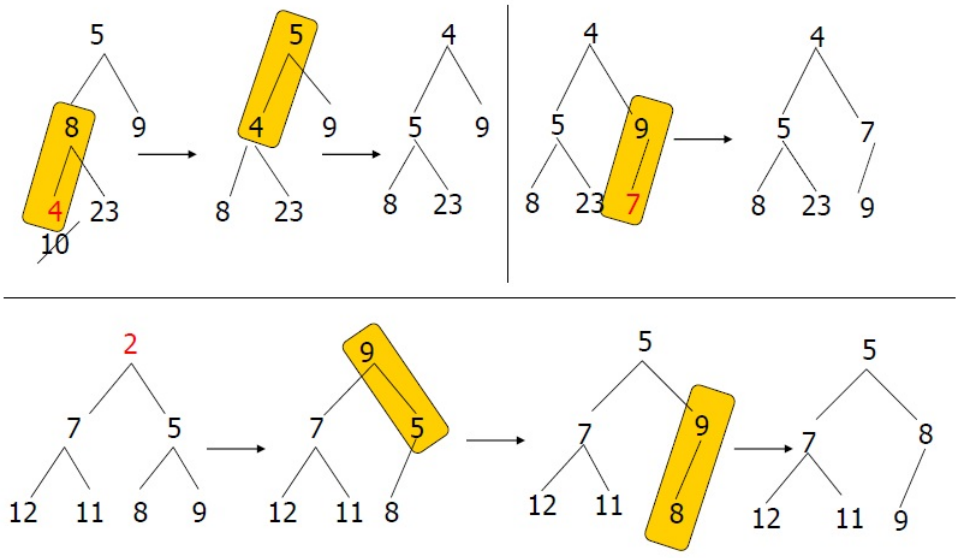


Figure 8.3: Heap operations. Top left: decrease value. Top right: insertion. Bottom: delete minimum.

8.2 Shortest paths

Finding the shortest path between two points obviously has a lot of applications. Not just can it model distances on a map, the "distance" can represent other things such as time or cost. In our abstract model, we are given a directed graph G with edge weights, and a starting vertex s. The length of a path is the sum of weights of all edges on that path. The goal is to find the shortest paths from s to *each* of the other vertices. This is called the **single source shortest path** (SSSP) problem.

At first sight, this might appear a more general and more difficult problem than just finding the shortest path of an s-t (source-destination) pair, which is probably what we need most of the time. This is however not true: if you want to find the shortest s-t path but you did not find the shortest distance from s to some other vertex u, it may be that the path from s to u and then from u to t is better. Hence, this SSSP problem is actually not more difficult, and we will develop algorithms for it.

Let's first look at some naive approaches. We could simply try all possible paths, calculate their distances and find the minimum one. The problem of this is that there can be exponentially many paths: for example, in Figure 8.4 (top), if there are n stages then there are 2^n possible paths (because there are two choices, top or bottom, at every stage). Such an algorithm would therefore be extremely inefficient.

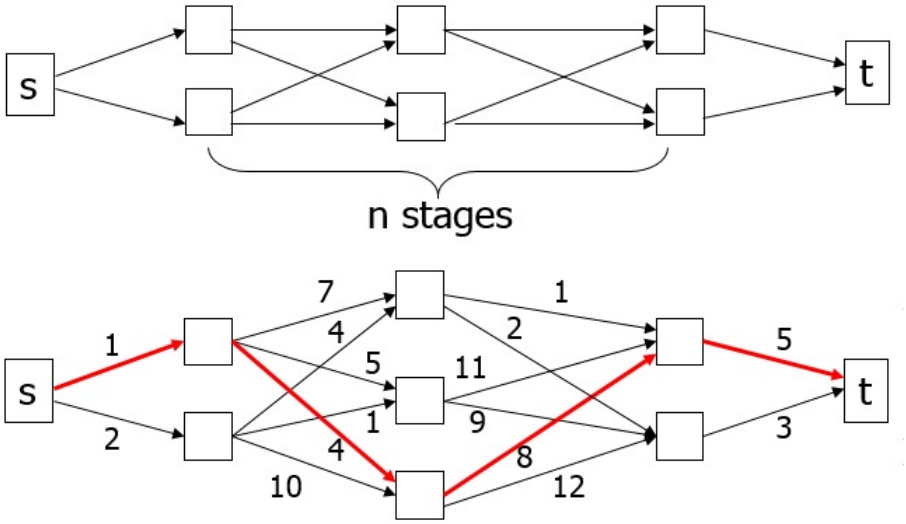


Figure 8.4: Top: exponentially many paths. Bottom: greedily selecting shortest outgoing edge does not work: it chooses the path 1-4-8-5 but the optimal one is 2-4-2-3.

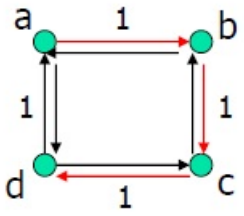
Alternatively, in some purely greedy sense, one might just pick the shortest outgoing edge at every node, starting at s.[5] But this is not a correct algorithm. Figure 8.4 (bottom) shows an example.

Before we move on, we describe the **optimal substructure** property satisfied by the shortest path problem. It means that, if P is a shortest path from s to t, and it goes through an intermediate vertex x, then the sub-path of P from s to x (call it P_1) must itself be a shortest

[5]Many people mistook this as Dijkstra's algorithm – it is not!

path from s to x. (And similarly for the sub-path from x to t.) The reason is that, if there is an alternative s-x path P' that is better than P_1, we can simply replace P_1 with P', and continue onto the original x-t path, to give a better path than P, contradicting its optimality. This can also be seen in Figure 8.4, where the path 1-7-1-5 is not optimal because 1-7 can be replaced by 2-4 (and 1-5 can also be replaced by 2-4).

This sounds completely obvious, and it is a property common to problems that can be solved greedily, and is also key to the technique of dynamic programming which we will learn in the next chapter. But note that this seemingly obvious property is not enjoyed by other, similar, problems. For example, consider the problem of finding the *longest* simple path in a graph. It does not have optimal substructure, as can be seen in this graph: the longest path from a to d is a-b-c-d, but the sub-path c-d itself is not the longest path from c to d.



There is no known polynomial time algorithm for the longest path problem (it is in fact NP-complete), despite it looking like just the opposite of shortest paths.

8.2.1 Dijkstra's algorithm

Recall that in BFS (Section 7.3.1), we used an "extending wavefront" approach to visit vertices. Vertices are visited in order of their distance (number of edges) from the starting vertex. Thus BFS is in fact a shortest path algorithm when the length of each edge is 1; the "level" of the node in the BFS tree is its distance from s.

When edge lengths are not 1, we can simulate it by pretending that it is formed by multiple unit-length edges: a length-3 edge is just virtually 3 length-1 edges. But of course, if there is a length-100 edge, we don't want to take 100 steps to cover it. A moment of reflection reveals that only the shortest outgoing edge matters. This gives rise to the following algorithm:

Algorithm 8.3 Dijkstra's algorithm

Input: Graph G, source vertex s

```
1: for each vertex v do
2:    D[v] ← d(s, v)
3:    P[v] ← s if v is neighbour of s, otherwise nil
4: end for
5: S ← {s}
6: while S ≠ V do
7:    find u in V − S with minimum D[u]
8:    add u to S
9:    for each v in V − S do
10:      if D[u] + d(u, v) < D[v] then
11:         D[v] ← D[u] + d(u, v)
12:         P[v] ← u
13:      end if
14:   end for
15: end while
```

The D array stores the provisional best distances to each of the vertices. Each time we look for the next vertex with the smallest D value, i.e., closest to s, that is being reached for the first time. (Imagine pouring water down at s and watch how they spread.) For this vertex u, the algorithm checks all its neighbours, to see if we now have a better distance by reaching it via u: by spending a distance of $D[u]$ from s to u, then add a distance of $d(u, v)$ following the edge (u, v). If this is shorter, update the D value of v.

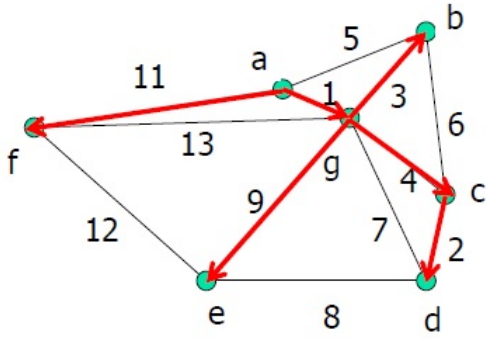


Figure 8.5: Example graph for Dijkstra. The red (thick) edges are edges of the shortest path tree.

Worked example. The following tables show an example execution of the graph in Figure 8.5, with a as the starting vertex. Initially

v	b	c	d	e	f	g
$D[v]$	5	∞	∞	∞	11	1
$P[v]$	a	/	/	/	a	a

First g has the minimum value, so we consider all neighbours of g to see what can be updated. For example, $D[b]$ should be updated since $D[g] + d(g, b) = 1 + 3 = 4$ is better than the old $D[b]$. But $D[f]$ should not be updated as $D[g] + d(g, f) > D[f]$. After one round we get

v	b	c	d	e	f	g
$D[v]$	4	5	8	10	11	1
$P[v]$	g	g	g	g	a	a

Then we find the smallest D value (excluding g that has been considered), which is b, and repeat the process. On this occasion the table does not change as no better path was found. Then this process goes on. In the end, we have

v	b	c	d	e	f	g
$D[v]$	4	5	7	10	11	1
$P[v]$	g	g	c	g	a	a

When the algorithm finishes, the D array stores the shortest distances to each of the vertices. But this is just a number, not a path. This is where $P[]$, or the **predecessor array**, comes into play. $P[v]$ records *the vertex just before reaching the destination* v for the current best path found. Each time D is updated, a better path to v is identified, and u is the "stop" just before the destination. This is sufficient to reconstruct the actual shortest paths: if in the end $P[v_1] = v_4$, for example, it means the shortest path to v_1 has v_4 as its last stop. So we check

$P[v_4]$ to see what is the last stop before v_4, and so on.

For the above example, we see that all the shortest paths are: $a \to g \to b$, $a \to g \to c$, $a \to g \to c \to d$, $a \to g \to e$, $a \to f$, and $a \to g$. In fact, all the shortest paths can be represented more concisely in a **shortest path tree**, represented by the red (directed) edges in Figure 8.5.

Observe that Dijkstra's algorithm is very similar to Prim's algorithm for MST; the only difference is in lines 10–13 on how to update D. Specifically, in Prim's we are just interested to find the closest vertex from anywhere within the growing component S; in Dijkstra we want to find the closest vertex from s. This similarity means that the running time of Dijkstra's is identical to Prim's: $O(n^2)$ for array, $O(m \log n)$ for heap.

Exercises

#8-1 **(Kruskal's algorithm)**

Consider the graph given by the adjacency matrix below, where non-zero entries represent edge weights, zero or infinity means there is no edge. Find the minimum spanning tree of this graph using Kruskal's algorithm and the array-and-linked-list data structure with the weighted union heuristic. You should:

- Write down the sequence of Find() and Union() operations performed
- Show the contents of the data structure after each step.

	v_1	v_2	v_3	v_4	v_5
v_1	0	1	3	∞	5
v_2	1	0	3	∞	6
v_3	3	3	0	4	5
v_4	∞	∞	4	0	2
v_5	5	6	5	2	0

#8-2 **(Prim's algorithm)**

Repeat the above using Prim's algorithm, starting at vertex v_1. Use an array to store the distance information. Show the contents of the data structure after each step.

8-3 **(Minimum spanning trees)**

The following is a divide and conquer algorithm for finding a minimum spanning tree in a graph: divide the vertex set into two parts, each having $n/2$ vertices and such that each of the two subgraphs is still connected. Recursively find a minimum spanning tree for each part. Then find the shortest edge connecting the two parts.

Show that this algorithm is not correct.

#8-4 **(Heap)**

Starting with an empty heap, show the contents of the heap after each of these operations:

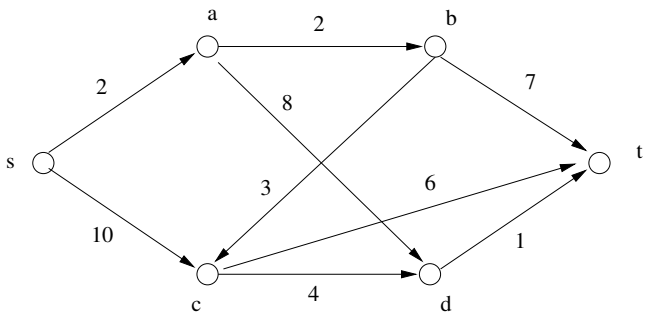
insert 1, insert 6, insert 4, insert 5, delete min, insert 2, insert 8, change 8 to 3, delete min

8-5 **(Heapsort)**

How can a heap be used to perform sorting? What is the time complexity?

#8-6 **(Dijkstra's algorithm)**

Consider the graph below. Find the shortest paths from s to all other vertices using Dijkstra's algorithm with the array data structure. Show the contents of your data structure after each step.



Chapter 9

Dynamic Programming

In this chapter we introduce our third major algorithm design technique, *dynamic programming* (DP). The "programming" here does not mean writing computer programs, but (back then before there is such a thing called computer programs) refers to a way of tabulating numbers to compute solutions. The best way to learn it is to see how it is used; so we will see how this technique is applied to a number of important problems, including sequence comparisons and shortest paths.

9.1 Weighted interval selection and separated array sum

Recall the activity selection problem in Chapter 6: given a set of activities occupying different time intervals, we want to pick a maximum number of non-overlapping activities. This implicitly assumes that each activity is equally important. If this is not the case, one way to formulate this is to assign a "value" or "weight" to each activity, and our objective becomes finding a subset of non-overlapping intervals so that the total value of the chosen intervals is as large as possible.

This problem generalises the original (unweighted) problem: if all intervals have the same value, then maximising the total value is the same as maximising the number of intervals.

Previous greedy algorithms do not work anymore in this more general model. It is easy to come up with examples where the earliest finishing first algorithm does not work correctly, which is hardly surprising given that it does not consider the weights at all. One might try some other greedy approaches that incorporate the weights, like picking the interval with largest weight first, or largest weight per length first; but none of these are optimal and in fact, there seems to be no way of "locally" adding an interval "safely" to construct solutions incrementally, as in greedy algorithms.

Let's turn our attention to a seemingly different problem. We will explain the reason at the end.

The separated-array-sum problem. Given an array A with n elements, choose a subset of elements so that any two chosen elements are separated by 2 cells or more, and that the sum is as large as possible.

Here we assume that an empty subset has sum 0. For example, suppose $A = [2, 4, 1, 3, 1, 6, 5, 4, 2]$. We cannot pick both $A[6]$ and $A[7]$ (6 and 5) because they are "too close" to each other. The best solution for this example is to pick $A[2], A[6]$ and $A[9]$, giving a sum of $4 + 6 + 2 = 12$.

It might seem natural to use a greedy algorithm that repeatedly picks the largest element, as long as it is not "too close" to elements already chosen. But this is not correct: a simple counterexample is the array $[6, 1, 7, 6]$. Greedy would have picked 7, but the optimal solution is 6+6. Other greedy approaches also fail to work. It seems we need some new ideas.

An important observation and a recursive formulation. Consider the last element, $A[n]$. We do not know whether $A[n]$ should be chosen as part of the solution; but we know that

Either $A[n]$ is part of the solution, or it is not.

Yes, this is a completely trivial, tautological statement. But pause here for a moment and make sure you truly understand what it means (and why it is trivial)! This is the kind of observations that underline the design of all dynamic programming algorithms.

This trivial observation allows us to *reduce the problem to a smaller subproblem* of the same form, and hence apply recursion:

- If $A[n]$ is part of the solution, then $A[n-1]$ and $A[n-2]$ cannot be; consider $A[1..n-3]$ in the next step
- If $A[n]$ is not part of the solution, consider $A[1..n-1]$ in the next step

But we don't know which one is correct – so we simply *try both and see which solution is better!*

Let's formulate this a bit more formally. First we have to define subproblems. Let $S(i)$ be the value (sum of chosen numbers) of the solution considering only the first i elements, i.e. $A[1..i]$. The reason we do this is because the recursion (the way we define it) always removes elements from the end, and the subproblems are therefore always of this form. What we want to find in the end is $S(n)$. Based on the idea we described, we can write down a recursive formula:

$$S(i) = \max(S(i-3) + A[i], S(i-1))$$

It is a maximum of two choices, the first corresponding to picking $A[i]$, and the second corresponding to not picking $A[i]$.

Like all recursion we need base cases. Here we have $S(0) = 0$ because an empty array has sum zero; $S(1) = \max(0, A[1])$ because with just one element, we will pick it unless it is negative, in which case we would rather have an empty sum; and $S(2) = \max(0, A[1], A[2])$ because we can pick at most one of $A[1]$ or $A[2]$ (or neither). We need three base cases because the recursive formula may "jump" from i to $i-3$; this way, we guarantee that the recursion will be "caught."

It is straightforward to turn this formula into a recursive algorithm: (call $S(n)$ to get the solution.)

Algorithm 9.1 $S(i)$ // for first i elements

```
if i == 0 then
  return 0    // three base cases
else if i == 1 then
  return max(0, A[1])
else if i == 2 then
  return max(0, A[1], A[2])
else
  return max(S(i − 3) + A[i], S(i − 1))
end if
```

Unfortunately, this algorithm is very inefficient. In fact it takes exponential time! The reason is that it will produce exponentially many subproblems; see Figure 9.1.

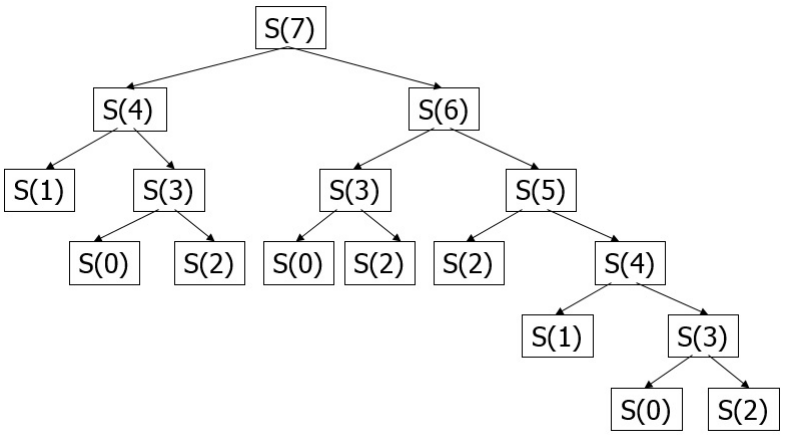


Figure 9.1: The tree of recursive calls made. Notice the redundancy.

Memorising solutions. However, we are just one trick away from getting an efficient algorithm. Observe that many of the subproblems, and indeed entire subtrees, are repeated in the figure. There is no point in recomputing those subproblems so many times, since they obviously have the same answers (and this is precisely why the algorithm is slow). Instead, we can memorise previously computed solutions: use an array to store the solutions of subproblems computed before, and only compute fresh if they have not been previously computed. In pseudocode:

Algorithm 9.2 Top-down version.

$M[0] \leftarrow 0, M[1] \leftarrow \max(0, A[1]), M[2] \leftarrow \max(0, A[1], A[2])$, other $M[] \leftarrow \infty$
call $S(n)$

Procedure $S(i)$:
if $M[i] == \infty$ **then** // not computed before
  $M[i] \leftarrow \max(S(i-3) + A[i], S(i-1))$
end if
return $M[i]$

Here ∞ (the mathematical symbol for infinity) does not actually have to be infinity, but any sort of indicator to indicate that the entry has not been computed before.

Removing recursion. We can simplify the algorithm even further and remove the recursion to convert it into an equivalent iterative algorithm. This comes from the observation that the recursion is only called on smaller values of i. Hence, if we compute the array M in the "correct" order (in this case, increasing i), every time the necessary array entry is required (according to the recursive formula), it must have been computed already. Hence we can eliminate recursion completely and obtain the following extremely simple algorithm:

Algorithm 9.3 Bottom-up version.

$M[0] \leftarrow 0, M[1] \leftarrow \max(0, A[1]), M[2] \leftarrow \max(0, A[1], A[2])$
for $i = 3$ **to** n **do**
  $M[i] \leftarrow \max(M[i-3] + A[i], M[i-1])$
end for
return $M[n]$

Time and space complexity. The time complexity of this last algorithm is clearly $O(n)$, since it has a for loop that executes $O(n)$ times and the content inside takes $O(1)$ time per iteration.

In dynamic programming algorithms we are also interested in the **space complexity** of the algorithm, i.e., the amount of working memory used, since we are, in effect, trading space for time. The algorithm uses an array M of size n, and thus it takes $O(n)$ space.

Finding the actual solution. The above algorithm computes the value of $S()$, which is the *value* of the objective function (i.e. the sum), but not the *actual solution* (i.e. which elements to pick). But it is easy to get that information. First, add to the above code an additional array to record the information on which of the two cases is chosen at every step:

Algorithm 9.4 With extra array to store choices made.

$M[0] \leftarrow 0, M[1] \leftarrow$ ... // same base cases as before
$Take[0] \leftarrow$ **false**
$Take[1] \leftarrow (A[1] > 0)$ // take if only 1 item and is positive
$Take[2] \leftarrow (A[2] > A[1]$ **and** $A[2] > 0)$ // take item 2 if only 2 items, it's the bigger one and is positive
for $i = 3$ **to** n **do**
  if $M[i-3] + A[i] > M[i-1]$ **then**
    $M[i] \leftarrow M[i-3] + A[i]; Take[i] \leftarrow$ **true**
  else
    $M[i] \leftarrow M[i-1]; Take[i] \leftarrow$ **false**
  end if
end for

With this information, we can recover the solution:

Algorithm 9.5 Traceback algorithm.

$i \leftarrow n$
while $i > 0$ **do**
  if $Take[i] ==$ **true then**
    print "Take item i"
    $i \leftarrow i - 3$
  else
    $i \leftarrow i - 1$
  end if
end while

If we apply this method to the array $A = [2, 4, 1, 3, 1, 6, 5, 4, 2]$, these are the computed values of $M[]$ and $Take[]$:

i	0	1	2	3	4	5	6	7	8	9
A		2	4	1	3	1	6	5	4	2
M	0	2	4	4	5	5	10	10	10	12
$Take$	F	T	T	F	T	F	T	F	F	T

Note that the $Take$ array (despite its name) does not tell you directly the solution of the problem of size n; in other words, the solution is not to simply pick those items with "true" in that array. Rather, we have to "trace back": since $Take[9]$ is true, we pick $A[9]$ and recursively look at $Take[6]$. It is again true, so pick $A[6]$ and recursively look at $Take[3]$. It is false, so do not take $A[3]$, and recursively look at $Take[2]$, and so on.

From separated-array-sum back to interval selection. Having solved the separated-array-sum problem, we now return to the weighted interval selection problem. In fact the two problems are very similar. Let $S(i)$ denote the total value in the optimal solution for the first i intervals $\{I_1, \ldots, I_i\}$ (ordered in increasing finishing times). The optimal solution either

includes I_i and discards all intervals that overlap it, leaving a subproblem with fewer intervals; or discards I_i and leaves a subproblem $S(i-1)$. Thus this is similar to the array sum problem, except that the "gap" is not necessarily 2 but depends on how the intervals overlap. This is also why they have to be ordered in finishing times: if I_j is the "first" (going backwards from I_{i-1}) interval that does not overlap with I_i, then all intervals before I_j will also not overlap with it. Thus we only need to check for one gap, although it is variable-sized.

9.2 Principles of dynamic programming

For a problem to admit a dynamic programming algorithm, it typically has the following properties:

1. A recursive formulation. The problem must be such that it can be solved building on solutions of smaller subproblems of the same form.

2. **Optimal substructure.** Roughly speaking, it means that the recursive formulation is defined in such a way, so that the optimal solution of a subproblem must also form part of the optimal solution of the bigger problem. This allows us to use the optimal solution of the subproblem without worrying how the other parts of the problem may interfere with it. For example, in the separated-array-sum example, the optimal solution for the entire problem $A[1..9]$ is $\{4, 6, 2\}$. The subproblem involved is $A[1..6]$, and the optimal solution for $A[1..6]$ is $\{4, 6\}$, which is used to build the solution $\{4, 6, 2\}$. (This is also a property of greedy algorithms: we already mentioned this in Section 8.2, and in fact the interval selection problem in Chapter 6 also has this property.)

3. **Overlapping subproblems.** Typically, the recursion will only reduce the size of the problem by a small amount; in addition, many of the subproblems will appear repeatedly over the course of the recursion. This differs from divide and conquer. This is also why we can use a table to memorise solutions of subproblems; it would not have provided any advantage if the same subproblems are not repeatedly encountered. This is what we observed in Figure 9.1.

All dynamic programming algorithms (that you will encounter in this module) follow the same basic structure. The only part that is specific for each individual problem is the recursive formulation; the rest are fairly standard stuff. Therefore, to design a dynamic programming algorithm we typically follow these steps:

Step 1: Define a recursive formulation that utilises the structure of the optimal solution. This is the difficult part. For this module, often we will define the subproblem for you, but you still need to work out the recursive formula.

Step 2: Compute the numerical value of the optimal solution using a table. Note that this table always only stores the values (cost/profit) we are trying to minimise/maximise. There are two approaches:

In the *top-down* approach, we start the recursive call from the largest subproblem, descending to smaller subproblems. Each time we first check whether we have computed

that subproblem previously by looking up the values from a table. The first time a subproblem is solved, its value is stored in the table. As subproblems are only solved when they are required, in theory not all table entries are always filled.

In the *bottom-up* approach, we iteratively fill the table starting from the smallest subproblems, in an order such that recursion is not required because the necessary smaller subproblems must be solved already. This means there is no recursion overhead.

Most of the time, we will describe our algorithms using the bottom-up approach.

Step 3: Construct the actual solution from the table using traceback. The traceback part is essentially identical for every DP algorithm and thus we will often omit it. First an extra table stores the choice made at each step; from this we can trace back the path (the list of subproblems) followed and reconstruct the solution step by step.

9.3 Sequence comparisons

In many applications, we can given sequences of characters, and we have to determine how similar they are, or identify the parts of the sequences that are similar and "align" them nicely, or to find the way to change one sequence to the other with the fewest changes. For example, the sequences could be DNA sequences, where the character set is A,C,G,T representing the four nucleotides; or in spell checking, where we want to identify which word is closest to a mis-spelt word.

9.3.1 Longest common subsequence

Consider two sequences of letters S_1 and S_2, where the letters come from some fixed alphabet. A *subsequence* is a subset of letters from a sequence in its original order.[1] For example, if S_1 = `abcaabcabc` and S_2 = `aabcabbabcc`, then `accab` is a subsequence of S_1, but not of S_2. There are numerous other subsequences, of course. A *common subsequence* of S_1 and S_2 is a subsequence of both sequences. For example `ababbc` is a common subsequence of S_1 and S_2. Again there are many other common subsequences. A *longest common subsequence* (LCS) is a common subsequence with maximum length. The figure here shows `abcababc` as another common subsequence, which is also the longest.

Our goal is to compute the LCS (and its length). A long common subsequence is an indication that the two sequences are highly similar, and a short one suggests they may not be that similar.[2]

Structure of optimal solution. First, observe that (from the above picture) that a common subsequence between two sequences define a non-crossing matching between the two sequences.

Consider two sequences $A[1..m]$ and $B[1..n]$. If $A[m] = B[n]$, i.e., the last characters match, then it is always "safe" to match them; this reduces to a smaller subproblem where both strings are shortened by one.

[1] Not to be confused with a *substring* which requires the chosen letters to be consecutive.

[2] There are other definitions of similarity where LCS may not be a good measure. For example, `abcde` and `edcba` might be considered similar (why?) but have a short LCS. We will not consider them here though.

Otherwise, if $A[m] \neq B[n]$ (doesn't match), then at least one of them, or both, cannot be part of the LCS, because otherwise a crossing would happen. This again allows us to reduce to a smaller subproblem, where one or both of the strings is shortened by one.

To turn this observation into a recursive formulation, let $LCS(i, j)$ be the length of LCS of $A[1..i]$ and $B[1..j]$, i.e., the first i letters of A and the first j letters of B. So we have, for any $i > 0$ and $j > 0$,

$$LCS(i,j) = \begin{cases} LCS(i-1, j-1) + 1 & \text{if } A[i] = B[j] \\ \max \begin{cases} LCS(i-1, j) \\ LCS(i, j-1) \end{cases} & \text{if } A[i] \neq B[j] \end{cases}$$

What are the base cases? We cannot apply the recursion if one of the strings becomes empty. When that happens, the length of the LCS must be zero: the only common subsequence between an empty string and another string is the empty string. Hence $LCS(0, j) = 0$ and $LCS(i, 0) = 0$.

This immediately gives us a recursive algorithm; like before, we need a table to store solutions and first check the table to see if the subproblem was computed previously (top-down approach). Because the subproblems are parameterised by i and j, where each of them can take on values from 0 to m (or n), we need a 2-dimensional table of size $m + 1$ by $n + 1$.

Bottom-up algorithm. Just like the previous problem, we can develop a bottom-up, non-recursive algorithm. It is important to note the right order to fill in the entries in the table, so that every time an element is needed, it is already computed. Since this is a 2-dimensional problem, the situation is slightly more complicated. But observe each table entry (i, j) depends on at most 3 other entries: $(i-1, j-1), (i, j-1), (i-1, j)$. All these would have been computed already if we use a standard double-for-loop like this:

Algorithm 9.6 LCS($A[1..m], B[1..n]$)

```
Let L be (m + 1) by (n + 1) 2D array with all entries initialised to zero
for i = 1 to m do
  for j = 1 to n do
    if A[i] == B[j] then
      L[i][j] ← L[i − 1][j − 1] + 1
    else
      L[i][j] ← max(L[i − 1][j], L[i][j − 1])
    end if
  end for
end for
```

Figure 9.2 is an example of the L array computed when the two strings are `abcbdab` and `bdcaba`. The table is filled row-by-row, column-by-column. The recursive formula means that if $A[i] = B[j]$, then the entry is simply 1 plus the element immediately to the top-left of the current cell; if $A[i] \neq B[j]$, the entry is equal to either the one immediately above, or immediately to the left, whichever is bigger. See the two cells with red arrows as examples. Thus it can be calculated quite easily even manually.

It is convenient to write down the letters alongside the first row/column, so it is easy to see which letters you are comparing. Furthermore, this neatly shows the meaning of the values in the table: the value in a cell is the length of the LCS of the two strings up to the corresponding

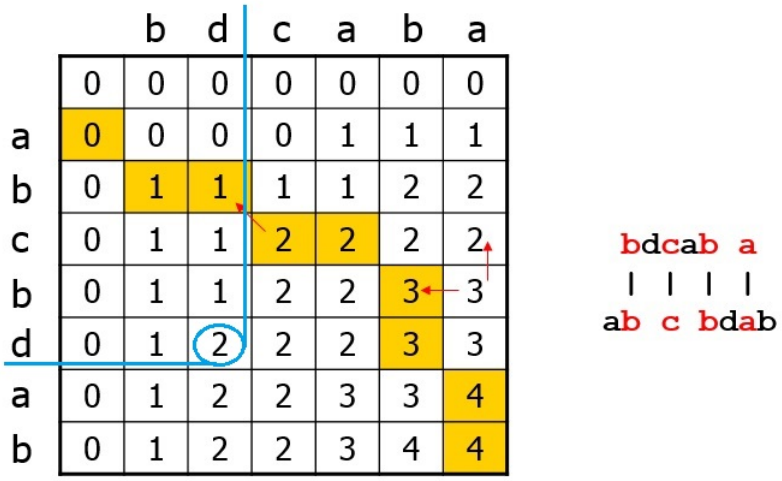


Figure 9.2: Example LCS execution.

positions. For example, the blue circled entry in Figure 9.2 corresponds to the length of the LCS between `abcbd` and `bd`.

Time and space complexity. Clearly, we used a table with $m+1$ rows and $n+1$ columns, hence the space complexity is $O(mn)$.

As for the time complexity, there are two nested for loops, and the content inside the for loop takes only $O(1)$ time to compute since it only involves a constant number of entries. Hence the time complexity is $O(mn)$.

Traceback. The table stores the lengths of the LCSs; it does not tell you what the actual LCSs are. To do that, we need to store, at each cell, which of the choices we have picked, or equivalently what directions do the entry of this cell come from. As an exercise on paper, we can draw arrows to indicate where the entries come from, as in Figure 9.2. In a real implementation of course there won't be arrows, but a separate 2-D array with entries that indicate the direction to follow.

Starting from the bottom-right corner of the table, we can follow the arrows to trace back a path to the beginning. This defines a LCS: each diagonal move means the corresponding characters in the two strings are "matched," and each horizontal or vertical move means a letter in one of the strings is skipped. The yellow shaded cells in Figure 9.2 is one possible path, and lead to the LCS `bcba` on the right.

Note that there can be multiple paths that can be traced (because sometimes different choices give the same value), and this corresponds to different LCSs (that has the same, maximum, length).

> Exercise: find another LCS of the same length from this table.

9.3.2 Sequence alignment

A very similar and highly related problem is the *sequence alignment* or the *edit distance* problem. We are given two sequences $A[1..m]$ and $B[1..n]$, and we want to change one into the other with some editing operations: insertion, deletion or character substitution. Each operation has a "cost," and we want to find the best sequence of editing operations, i.e. the one that has the minimum total cost. For example, changing the string `cutecat` to `catcafe` may be done by replacing the u with an a, delete the e, replace the last t with f and append an extra e, which may or may not be the optimal sequence depending on the costs of the various operations. The problem is also called sequence alignment because, as you will see, it also produces an alignment between sequences, which is important in bioinformatics applications.

Let's develop a recursive formulation for this problem. Here, we assume all operations have the same cost of 1. (Different costs can be dealt with in a very similar manner.) Let $D(i, j)$ be the edit distance between $A[1..i]$ and $B[1..j]$. If the last two characters $A[i]$ and $B[j]$ match, we only need to find the optimal way to edit $A[1..i-1]$ to $B[1..j-1]$ with no additional cost afterwards. If they don't match, then one of three things must happen: either $A[i]$ is deleted and $A[1..i-1]$ somehow transforms into $B[1..j]$; or $A[1..i]$ transforms into $B[1..j-1]$ and then $B[j]$ is inserted; or $A[1..i-1]$ transforms into $B[1..j-1]$ and then $A[i]$ is replaced with $B[j]$. Then for any $i > 0$ and $j > 0$,

$$D(i,j) = \begin{cases} D(i-1,j-1) & \text{if } A[i] = B[j] \\ 1 + \min \begin{cases} D(i-1,j) \\ D(i,j-1) \\ D(i-1,j-1) \end{cases} & \text{if } A[i] \neq B[j] \end{cases}$$

The base cases are $D(i, 0) = i$ and $D(0, j) = j$, because when one of the sequences is empty, the best way to transform to the other sequence is by a series of insertions/deletions.

A dynamic programming algorithm then follows easily; just replace the recursive formula (and the base cases) of Algorithm 9.6 with the ones above.

The following is an example execution of the algorithm.

		b	d	c	a	b	a
	0	1	2	3	4	5	6
a	1	1	2	3	3	4	5
b	2	1	2	3	4	3	4
c	3	2	2	2	3	4	4
b	4	3	3	3	3	3	4
d	5	4	3	4	4	4	4
a	6	5	4	4	4	5	4
b	7	6	5	5	5	4	5

Edit distance = 5

Alignment:

```
bdcab a
 |  |  |
ab c bdab
```

Figure 9.3: Example edit distance algorithm execution.

9.4 Shortest paths in directed acyclic graphs

Recall that in Dijkstra's algorithm (Algorithm 8.3), the core part is to check for shorter distances to each vertex:

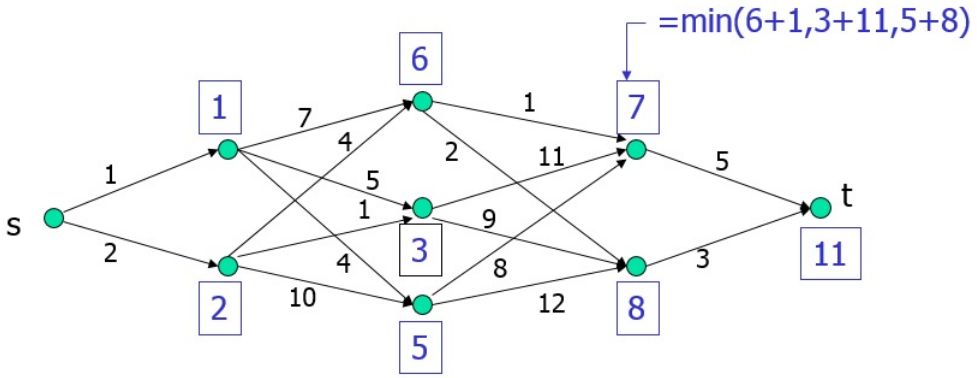
if $D[u] + d(u, v) < D[v]$ **then**
  $D[v] \leftarrow D[u] + d(u, v)$
end if

This is actually a key operation, which we call the *edge update procedure*, that appears again and again in all shortest paths algorithms that we study. This operation is always "safe": if there is a $D[v]$ value, it always means there is a path of that distance to v, and it is updated (to a smaller value) only if there is a better path with that shorter distance.

In fact, most shortest path algorithms simply apply this edge update procedure "in some right order" to all the edges. In Dijkstra, this correct order is based on the D values, i.e., how far the vertex is away from s. In this section, we will see that for some special types of graphs, it can be applied in a different way so as to find shortest paths more efficiently.

A **directed acyclic graph** (DAG) is a directed graph without a (directed) cycle. Shortest paths can be computed more efficiently in these types of graphs, using ideas that resemble dynamic programming.

For example, consider the following "multi-stage graph." Vertices are arranged in "stages" (each column is a stage) and the edges only connect vertices from stage i to stage $i + 1$. The shortest path from s to t must pick a vertex at each stage.

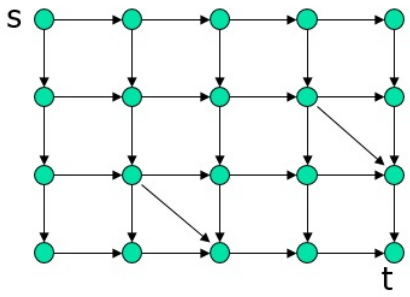


We simply need to apply the edge update procedure once to each edge, in the "correct" order, which is left to right. For example, the shortest distance to the top vertex at the third stage (which is 7) can be computed from the shortest distances to all vertices to the second stage (which are 6, 3, 5) and the edge distances between stages 2 and 3, since any path to the third stage must pass through one vertex at the second stage. So the time complexity is $O(m)$.

This can be viewed as dynamic programming: the subgraph consisting of only the first i stages is a subproblem of size i, and the optimal solution to the subproblem of size $i - 1$ can be used to compute the optimal solution of size i.

As another example, consider a *grid graph*, where vertices are arranged in a rectangular grid and there are horizontal and vertical edges between any two adjacent vertices (in one direction). Some diagonal edges also exist.

Suppose for example all horizontal and vertical edges have weight 2, and all diagonal edges have weight 3. Let $D(i,j)$ denote the shortest distance from s to the node at row i, column j. Since there is at most three incoming edges to that node, the shortest path must go through one of those edges, and hence



$$D(i,j) = \min(D(i-1,j)+2, D(i,j-1)+2, D(i-1,j-1)+3)$$

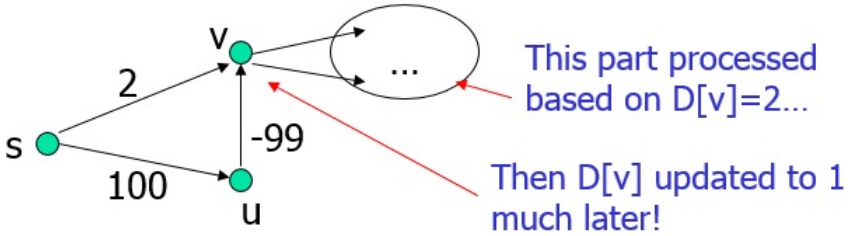
(if the corresponding edges exist). But this is basically the same formula for edit distance, if the cost of insertion/deletion is 2 and substitution is 3! It can therefore be solved in time linear in the number of vertices/edges of the graph.

More generally, for any DAG, we can assign labels to vertices so that any edge always goes from a smaller vertex to a larger vertex (this is called *topological sort*; details omitted here). This allows us to apply the edge update procedure only once, following this natural order, to give correct shortest paths.

In DAG, this kind of DP-based approach also allows us to solve more general problems. It can solve other optimisation problems such as longest paths (by simply replacing min with max; note that for non-DAG, there is no efficient algorithm at all for longest paths). Those algorithms also work for graphs with negative edge weights, something we consider next.

9.5 Negative edges and cycles

Recall that Dijkstra's algorithm can be seen as some kind of an "extending wavefront" approach. It does not always work for graphs with negative edge weights. Consider the example graph below. After one iteration of Dijkstra's algorithm, it would have set $D[v]$ to 2 since it found the path s-v (the path s-u-v is shorter but not found yet at this point). This value of $D[v]$ will then be used to calculate further D values, for vertices in the circled part of the graph. When eventually it was found that $D[v]$ can be improved to 1, what should we do? The original Dijkstra would not recompute the D values of those circled vertices based on this improved $D[v]$, which therefore gives a wrong answer. We can recalculate those things, but then the time complexity is no longer the same; furthermore, who is to say that this $D[v]$ may not improve further later? When do we even terminate?



In this section we develop an algorithm for finding shortest paths when the graph has negative edges. Not only is it more general, but it turns out that similar ideas can be found in algorithms for decentralised settings, i.e., no node has global information (e.g. distributed routing algorithms).

Before that, we have to get something out of the way first: negative cycles. Now that we allow negative edges, we may have introduced negative cycles, i.e., cycles with negative total edge weight. This causes a problem because a graph with negative cycles does not have a well-defined shortest path: each time you go around a negative cycle, the length of the path is reduced so one can simply go round a negative cycle indefinitely to get a $-\infty$ length. But for those graphs with negative edges but not negative cycles, shortest paths are still well-defined. So from now on, *we assume that the graphs we consider have no negative cycles.*

We first establish two simple but essential properties of shortest paths:

Proposition 9.1 *Any shortest path does not contain cycles.*

This is true because, if a shortest path contains a positive (or zero) cycle, we can remove it and the total weight is not worsened. And we already assumed that negative cycles do not exist.

Proposition 9.2 *Any shortest path has at most $n-1$ edges.*

This is because a path that does not visit the same vertex twice (if it does it would form a cycle, but the previous property already ruled that out) must be visiting a new vertex for each new edge along the path. Thus, if a path has more than $n-1$ edges, then it must reach more than n vertices (including the starting vertex), which contradicts that the graph has only n vertices.

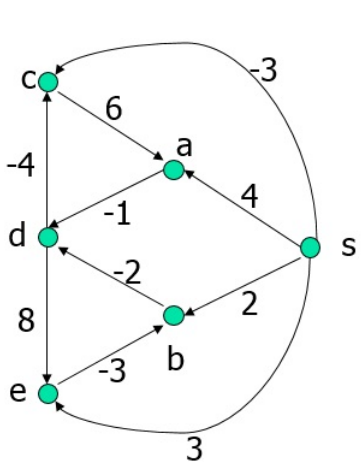
Next we develop a dynamic programming formulation. Since DP requires the notion of "smaller" subproblems, we artificially introduce a parameter, the number of edges. Let $S(i, v)$ denote the length of the shortest path from s to v *using a path with at most i edges.* First, observe that the path that gives $S(i, v)$ either uses fewer than i edges, or it uses exactly i edges. This is (again) a completely trivial statement. In the first case, we can apply recursion, since it is now a "smaller" problem (with fewer edges). In the second case (it uses exactly i edges), it must consist of a path with $i-1$ edges from s to some vertex w, followed by a single edge (w, v). The first part must itself be an optimal path from s to w (this is where the optimal substructure property is needed), which we can find by recursion. We don't know which vertex w is, but we can try all vertices and take the best one! We also don't know which of the two cases (fewer than i edges or exactly i edges) gives the optimal solution, so we again try both possibilities. So we have

$$S(i, v) = \min \begin{cases} S(i-1, v) \\ \min_w(S(i-1, w) + d(w, v)) \quad \text{(minimum over all } w\text{)} \end{cases}$$

This gives us an algorithm:

Algorithm 9.7 Proto-Bellman-Ford

```
1: M[0][s] ← 0, M[0][v] ← ∞ for all other v    // initialise
2: for i = 1 to n − 1 do
3:    for each node v do    // in any order
4:       compute M[i][v] using formula above
5:    end for
6: end for
```



	0	1	2	3	4	5
s	0	0	0	0	0	0
a	∞	4	3	3	2	0
b	∞	2	0	0	0	0
c	∞	-3	-3	-4	-6	-6
d	∞	∞	0	-2	-2	-2
e	∞	3	3	3	3	3

Example: S(2, a) =
min(S(1,a), S(1,c)+d(ca))

Figure 9.4: An example graph and an execution of Algorithm 9.7.

The for loop runs up to $n-1$ only because we know that a shortest path can have at most $n-1$ edges (Proposition 9.2). Figure 9.4 shows an example execution.

This algorithm runs in $O(n^3)$ time. To see this, the two for loops each have at most n iterations. But line 4 require not $O(1)$ time but $O(n)$ time, because it has to find the minimum among n quantities according to the recursive formula. This is your first example where filling in an entry in the table takes more than constant time (you will see more examples later!)

The space complexity is $O(n^2)$, due to the 2-D table.

Simplifying the algorithm. Observe that $S(i,*)$ depends on $S(i-1,*)$ only; there is therefore no need to keep the earlier entries $S(i-2,*),\ldots,S(0,*)$. We therefore only need the current column and the previous column, and this allows reuse of memory and cut the space complexity down to $O(n)$. In fact, we can take this idea further and simplify the recursive formula: let $M[v]$ be an 1-D array. Then

$$M[v] = \min(M[v], \min_w\{M[w] + d(w,v)\})$$

We can further simplify the algorithm to give the **Bellman-Ford algorithm**[3] below. We omit its proof of correctness here. Another way of interpreting this algorithm is that it applies the edge update procedure to all edges. Previously, we said that all we need is to apply them "in some right order." We don't know the right order here, so we choose an arbitrary one and do it a "large enough" number of times – it turns out that $n-1$ times are enough.

[3]Some people/books use the name "Bellman-Ford" to refer to Algorithm 9.7, which here we call proto-Bellman-Ford (not an official name). Here we follow [CLRS] and call this one Bellman-Ford.

Algorithm 9.8 Bellman-Ford algorithm

Input: Graph G with distance values $d()$, starting vertex s

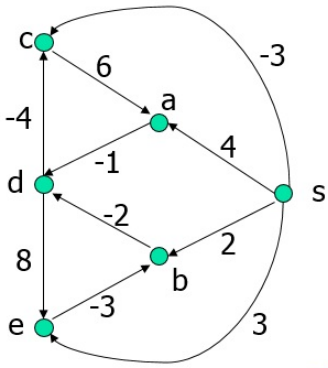
```
D[s] ← 0, D[v] ← ∞ for all other v
for i = 1 to n − 1 do
  for each edge (u, v) do    // in any order
    if D[v] > D[u] + d(u, v) then
      D[v] ← D[u] + d(u, v)
      P[v] ← u
    end if
  end for
end for
```

As in other shortest path algorithms, the D array only stores the distances, not the paths, but the paths can be reconstructed by a predecessor array (P in the above).

It is easy to see that the time complexity of this algorithm is $O(mn)$ and its space complexity is $O(n)$.

Note that in this algorithm, i no longer means the number of edges; it has no obvious meaning. Also, note that the same D (and P) value can be updated more than once in one iteration.

Worked example. The following shows an example execution. The (arbitrary) edge order used was: $(sa), (sb), (sc), (se), (ca), (ad), (bd), (eb), (dc), (de)$



initial

v	a	b	c	d	e
D[v]	∞	∞	∞	∞	∞

i = 1

v	a	b	c	d	e
D[v]	~~4~~ 3	~~2~~ 0	~~-3~~ -4	~~2~~ 0	3

i = 2

v	a	b	c	d	e
D[v]	2	0	-4	-2	3

(Values updated more than once in one round)

And so on...

9.6 All-pairs shortest paths

Up till now, all shortest path problems we have are single-source shortest path (SSSP) problems, i.e., find the shortest paths from a starting vertex s to all other vertices. Here, we consider the all-pairs shortest path (APSP) problem: given a graph, find the shortest paths between *every pair of vertices*.

An obvious way to solve this problem is to apply the SSSP algorithm n times, once for each vertex as the starting vertex. Thus in the case of positive edge weights, we can apply Dijkstra's

algorithm to give a time complexity of (say using a heap) $O(m \log n) \times n = O(mn \log n)$. For general edge weights, applying Bellman-Ford algorithm instead gives a time of $O(mn) \times n = O(mn^2)$. For dense graphs, these can be up to $O(n^3 \log n)$ and $O(n^4)$ respectively.

We can attempt to apply the same idea in Bellman-Ford to here. Let $v_1, v_2, \ldots, v_n$ be the vertices. Let $D_{ij}^{(k)}$ denote the shortest distance from v_i to v_j using at most k edges. Then the same idea gives the recursive formula

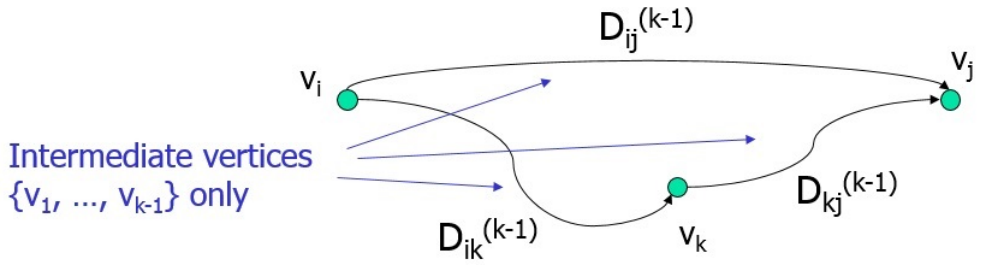
$$D_{ij}^{(k)} = \min(D_{ij}^{(k-1)}, \min_w \{D_{iw}^{(k-1)} + d(w, j)\})$$

This gives an $O(n^4)$ time algorithm: a 3-D $n \times n \times n$ table where each entry takes $O(n)$ time to fill. There are some tricks that can improve this to $O(n^3 \log n)$, but we will not cover them here.

9.6.1 The Floyd-Warshall algorithm

It turns out that we can develop a faster algorithm, using dynamic programming but with an alternative DP formulation. Instead of using the number of edges as a parameter, we arbitrarily limit ourselves on the range of vertices that can be used to form the path. Specifically, let $D_{ij}^{(k)}$ denote the shortest distance from vertex v_i to v_j, *using intermediate vertices with labels at most* k. So $D_{ij}^{(0)}$ means no intermediate vertices are allowed, and the path must be the direct edge from v_i to v_j; and $D_{ij}^{(n)}$ means any intermediate vertices are allowed, which is therefore the answer we want.

Consider a pair of vertices v_i and v_j. Observe that the shortest path $D_{ij}^{(k)}$ either uses v_k as an intermediate vertex, or it does not (i.e., uses intermediate vertices with label at most $k-1$). This allows us to *reduce to smaller subproblems*: If it uses v_k as an intermediate vertex, then the two paths $v_i \rightsquigarrow v_k$ and $v_k \rightsquigarrow v_j$ uses intermediate vertices with label at most $k-1$. If it does not use v_k, then we directly have a smaller subproblem since the path only uses intermediate vertices with labels at most $k-1$. We don't know which of these two is better, so we try both and pick the shorter one.



More formally, the recursion is:

$$D_{ij}^{(k)} = \min(D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)})$$

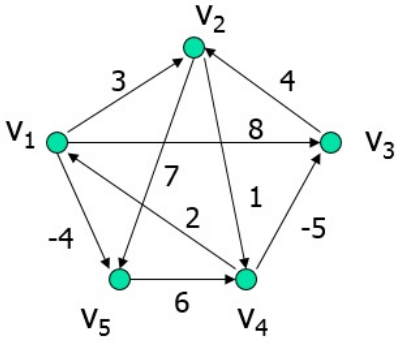
We can arrange the $D_{ij}^{(k)}$ in matrices, where $D^{(k)}$ is an $n \times n$ matrix, $k = 0, 1, \ldots, n$, and $D_{ij}^{(k)}$ is the entry in the i-th row, j-th column of $D^{(k)}$.

Algorithm 9.9 Floyd-Warshall algorithm

$D^{(0)} \leftarrow$ adjacency matrix
$P^{(0)}_{ij} \leftarrow i$ if (i,j) is an edge, otherwise nil
for $k = 1$ **to** n **do**
  for $i = 1$ **to** n **do**
    for $j = 1$ **to** n **do**
      if $D^{(k-1)}_{ij} < D^{(k-1)}_{ik} + D^{(k-1)}_{kj}$ **then**
        $D^{(k)}_{ij} \leftarrow D^{(k-1)}_{ij}$; $P^{(k)}_{ij} \leftarrow P^{(k-1)}_{ij}$
      else
        $D^{(k)}_{ij} \leftarrow D^{(k-1)}_{ik} + D^{(k-1)}_{kj}$; $P^{(k)}_{ij} \leftarrow P^{(k-1)}_{kj}$
      end if
    end for
  end for
end for

The above algorithm implements the idea. As before, the D matrices only store the lengths of the shortest paths, and to recover the actual paths we use additional matrices P which stores the predecessor information. In the case where $D^{(k-1)}_{ij}$ is better, the predecessor is unchanged (same as $P^{(k-1)}_{ij}$), but in the case where $D^{(k-1)}_{ik} + D^{(k-1)}_{kj}$ is better, the predecessor is not necessaraily k but the last stop in the second leg of this path, $P^{(k-1)}_{kj}$.

Worked example. Here are the contents of the arrays when the graph on the right is used as an input:



$$D^{(0)}: \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \quad P^{(0)}: \begin{bmatrix} / & 1 & 1 & / & 1 \\ / & / & / & 2 & 2 \\ / & 3 & / & / & / \\ 4 & / & 4 & / & / \\ / & / & / & 5 & / \end{bmatrix}$$

$$D^{(1)}: \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \quad P^{(1)}: \begin{bmatrix} / & 1 & 1 & / & 1 \\ / & / & / & 2 & 2 \\ / & 3 & / & / & / \\ 4 & 1 & 4 & / & 1 \\ / & / & / & 5 & / \end{bmatrix}$$

$$D^{(2)}: \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \quad P^{(2)}: \begin{bmatrix} / & 1 & 1 & 2 & 1 \\ / & / & / & 2 & 2 \\ / & 3 & / & 2 & 2 \\ 4 & 1 & 4 & / & 1 \\ / & / & / & 5 & / \end{bmatrix}$$

$$D^{(3)}: \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \quad P^{(3)}: \begin{bmatrix} / & 1 & 1 & 2 & 1 \\ / & / & / & 2 & 2 \\ / & 3 & / & 2 & 2 \\ 4 & 3 & 4 & / & 1 \\ / & / & / & 5 & / \end{bmatrix}$$

$$D^{(4)}: \begin{bmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix} \quad P^{(4)}: \begin{bmatrix} / & 1 & 4 & 2 & 1 \\ 4 & / & 4 & 2 & 1 \\ 4 & 3 & / & 2 & 1 \\ 4 & 3 & 4 & / & 1 \\ 4 & 3 & 4 & 5 & / \end{bmatrix}$$

$$D^{(5)}: \begin{bmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix} \quad P^{(5)}: \begin{bmatrix} / & 3 & 4 & 5 & 1 \\ 4 & / & 4 & 2 & 1 \\ 4 & 3 & / & 2 & 1 \\ 4 & 3 & 4 & / & 1 \\ 4 & 3 & 4 & 5 & / \end{bmatrix}$$

It is easy to see that the time complexity of the algorithm is $O(n^3)$ because of the three nested for loops. The space complexity can be only $O(n^2)$ since each D and P array has $n \times n$ elements and there is no need to keep all previous arrays; just keep the most recent one would do.

Exercises

9-1 **(Fibonacci numbers)**

The *Fibonacci numbers* are defined by the formulas

$$F_0 = 0, F_1 = 1, \text{ and } F_n = F_{n-1} + F_{n-2} \text{ for } n \geq 2$$

That is, each number is the sum of the previous two. So the first few Fibonacci numbers are 0, 1, 1, 2, 3, 5, 8, 13, 21.

(a) Give a simple recursive algorithm to compute the n-th Fibonacci number F_n.

(b) Consider instead the following algorithm:

$F[0] \leftarrow 0; F[1] \leftarrow 1$
for $i = 2$ **to** n **do**
  $F[i] \leftarrow F[i-2] + F[i-1]$
end for
print $F[n]$

What is the time and space complexity of the algorithm?

(c) Consider the following algorithm:

$x \leftarrow 0; y \leftarrow 1$
for $i = 1$ **to** n **do**
  $x \leftarrow x + y$
  $y \leftarrow x + y$
end for
print x and y

Trace the execution of the algorithm for $n = 4$, showing the values of x and y immediately after the contents of the for loop is executed every time. What is this algorithm doing? What is its time and space complexity?

#9-2 **(Longest common subsequence)**

Find the longest common subsequence of the sequences TTACG and ATACAG. Show the contents of the dynamic programming table.

#9-3 **(Edit distance)**

Find the edit distance between the two sequences TTACG and ATACAG, if the cost of each insertion or deletion is 2 and the cost of each substitution is 3. You should write down the recursive formula for the optimal cost $D(i, j)$ and show the contents of the dynamic programming table.

9-4 **(Greedy algorithm for LCS?)**

A student proposes the following algorithm for finding the LCS of two sequences:

(1) First, remove from each sequence characters that do not appear in the other sequence. Denote the resulting sequences by $A[1..m]$ and $B[1..n]$.

(2) If $A[1] = B[1]$, then match $A[1]$ to $B[1]$ as part of the LCS, and recursively solve the subproblem $A[2..m]$, $B[2..n]$.

(3) Otherwise ($A[1] \neq B[1]$), let x be the earliest position at which $A[1]$ appears in B, and y be the earliest position at which $B[1]$ appears in A. If $x \leq y$, match $A[1]$ to $B[x]$ and recursively solve the subproblem $A[2..m], B[x+1..n]$. If $x > y$, then match $B[1]$ to $A[y]$ and recursively solve the subproblem $A[y+1..m], B[2..n]$.

Show that this algorithm does not always return the longest common subsequence.

*9-5 **(Coin changing revisited)**

Consider the problem of making an exact amount of n using the minimum number of coins from a set of k coin denominations $c_1, c_2, \ldots, c_k$, where $1 = c_1 < c_2 < \cdots < c_k$. (For example the UK coins have denominations {1, 2, 5, 10, 20, 50, 100, 200}.) In Exercise 6-3 we noted that the greedy algorithm does not always give the optimal solution.

(a) Let $N(i, j)$ denote the minimum number of coins required to make an exact amount of i, using only coins from the set $c_1, c_2, \ldots, c_j$. Give a recursive formula for $N(i, j)$. Also, write down the base case. (*Hint:* either you use one (or more) coin of value c_j, or you use none. Either way you reduce at least one of i or j.)

(b) Give a dynamic programming algorithm for the problem. Analyse the time and space complexity of the algorithm, in terms of n and k.

(c) Illustrate how your algorithm works with $n = 6$ and coin denominations {1, 3, 4}, by showing the contents of the dynamic programming table.

*9-6 **(Sentence segmentation)**

You are given a string of length n which is possibly formed from a sequence of valid words but without spaces and punctuation marks. An example is

```
isitherealready
```

Your task is to determine whether the string can be broken down into a sequence of valid words. Sometimes there can be more than one ways; sometimes there can be none. (Note that here we only care about whether each word is valid, but not whether the whole sentence makes sense or is grammatically correct.)

You have a function Dict(s) which takes a string s and returns true if s is a valid word in a dictionary, and false otherwise. Assume this function takes constant time.

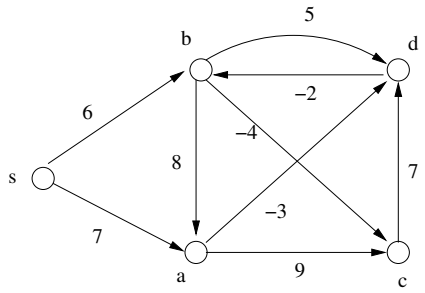
(a) Let $S(i)$ denote the truth value (true or false) of whether the first i characters in the string can be decomposed into a sequence of valid words. Give a recursive formula for $S(i)$. Also, write down the base case. (Note that since $S()$ records either true or false, your recursive formula will need to use logical operators like AND, OR rather than the usual +, max, min etc.)

(b) Hence, give a dynamic programming algorithm for the problem.

(c) Analyse the time and space complexity of your algorithm.

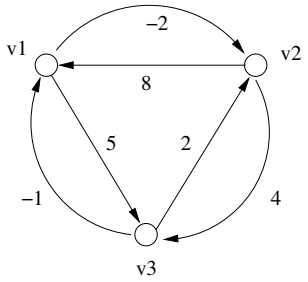
#9-7 **(Bellman-Ford algorithm)**

Find the distances of the shortest paths from s to all other vertices in the graph below using the Bellman-Ford algorithm. Show the contents of the data structure after each round.



#9-8 **(Floyd-Warshall algorithm)**

Find the shortest paths between all pairs of vertices in the graph below using the Floyd-Warshall algorithm. Show the distance and predecessor matrices after each round.



*9-9 **(Maximum contiguous subarray revisited)**

Recall the following problem discussed in Exercise 4-8: given an array $A[1..n]$ with n elements, we want to find the *maximum-sum subarray*, i.e. the subarray $A[i..j]$ starting with $A[i]$ and ending with $A[j]$ such that the sum $A[i]+A[i+1]+\cdots+A[j]$ is maximum over all possible subarrays. For example, if $A = [2, -10, 6, 4, -1, 7, 3, -8]$, then the maximum-sum subarray is $[6, 4, -1, 7, 3]$ with sum 19. In Exercise 4-8, we used divide and conquer and gave an $O(n \log n)$ time algorithm. In fact, we can use dynamic programming to give an $O(n)$ time algorithm!

Let $G(i)$ denote the sum of the maximum contiguous subarray from $A[1..i]$, and let $S(i)$ denote the sum of the maximum contiguous subarray from $A[1..i]$ with the additional restriction that the subarray must end with $A[i]$ (this includes the case of an empty subarray with sum 0.) Derive recursive formulas for $G(i)$ and $S(i)$. Hence give an $O(n)$ time algorithm for the problem.

Chapter 10

Network Flow

In this chapter we consider another graph problem, of finding the maximum flow. The solution does not belong to the three big types of algorithmic techniques we discussed, but it is itself a very important topic. We also illustrate the idea of *problem reduction* or *transformation* by showing how network flows can be used to solve a kind of matching problem.

10.1 Flows and networks

Often we can use graphs to model problems related to the concept of flows. For example, it could be traffic on roads, a network of water pipes, evacuation plans of a building, etc. Figure 10.1 is an example of a flow network. The edge labels denote the maximum amount of "traffic" that can flow through in unit time. We want to answer the question of how much traffic between two points can this network support without "overflowing" (like congestion).

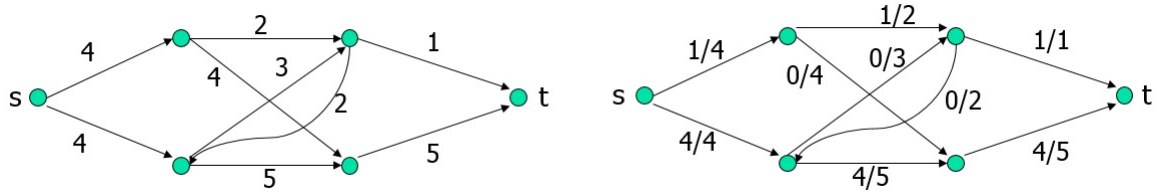


Figure 10.1: Left: an example flow network. Right: Same network with a possible flow.

More formally, we are given a directed graph. Each edge $e = (u, v)$ has a *capacity* c_e, which means at most c_e units of flow can pass through the edge from u to v. Note that this is a directed graph, so this edge does not allow any flow from v to u. Each edge can admit a *flow*, a number f_e that is not larger than the edge's capacity. In other words, $f_e \leq c_e$ for every e: this is called the *capacity constraint*.

The above is only about one edge, but of course in a network, the edges are connected together at vertices. All vertices (except two special ones below) satisfy the *conservation constraints*: the total incoming flow into the vertex must be equal to the total outgoing flow. In other words, flows do not appear out of nowhere or disappear, and there should be no "traffic congestion." But the traffic have to originate from somewhere: there is a special vertex called

the *source* s where all flows originate from. And there is the opposite *sink* vertex t which, as the name suggests, is a vertex with only incoming flows and where the flows "drain away."

We use the notation f/c on an edge to denote a flow f on an edge with capacity c. Figure 10.1 also shows a possible flow in the network. It can be checked that the flow is indeed conserved.

The *value* of a flow is the total amount of flow that goes from s to t. Because of the conservation constraints, the total amount of flow coming out from s is equal to the total amount of flow going into t; thus this is well-defined. In Figure 10.1, the flow value is 5.

Given such a network, our objective is to find the maximum flow from the source to the sink. We want to find both the maximum flow value (a number) as well as the actual flow, i.e., the flow on each edge. The flow in Figure 10.1 is not the maximum and can be improved (how?)

Greedy algorithm fails. One tempting idea might be to increase flows on edges incrementally in some order. In other words, we greedily (locally) add flows: start with an all-zero flow, find some path from s to t that has positive remaining capacity, assign some flow along this path, and repeat. For example, in Figure 10.1 we can add one more unit of flow along the path 1/4, 0/4, 4/5.

Unfortunately this is not a correct algorithm. For example, consider the graph in Figure 10.2 (top left). One could easily have chosen the three straight edges in the middle as a path, and assign a flow of 20 along this path. Afterwards, nothing more can be done and we are "stuck." But the maximum flow is in fact 30, by diverting 10 units into the lower arc and also pushing 10 units in the upper arc. The incremental method gets stuck because we need to *reduce* the flow on the middle edge (from 20/30 to 10/30) to increase overall flow!

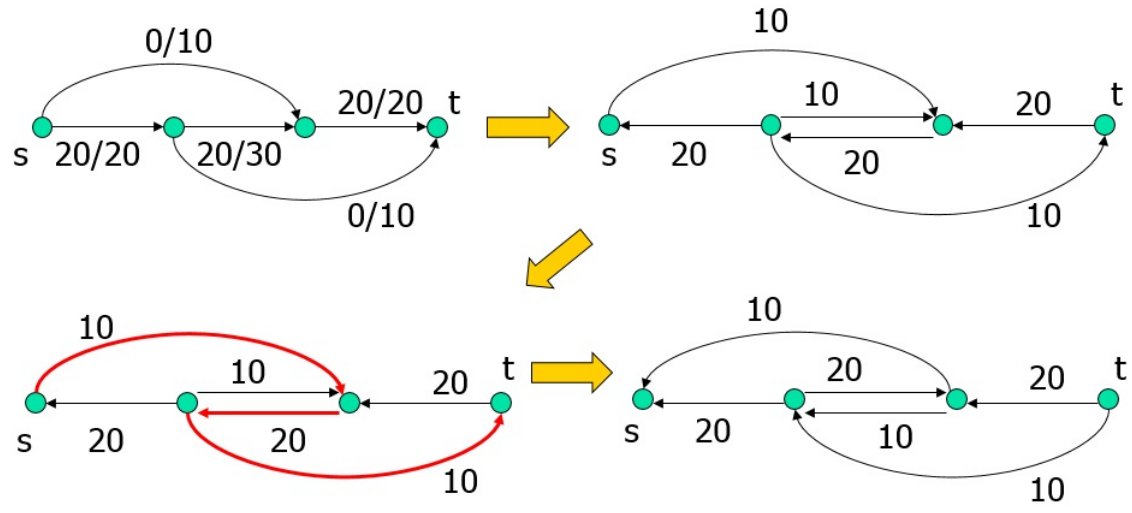


Figure 10.2: Top left: after assigning 20 units to the straight edges we are stuck. Top right: residual network from this flow. Bottom left: augmenting path. Bottom right: updated residual network. There are no more augmenting paths.

10.2 The Ford-Fulkerson algorithm

It might sound like we would need some kind of backtracking to deal with the problem mentioned above. But in fact the following would do. Given a network with capacities and a flow, we can construct a **residual network**, where

(i) the nodes are the same as the original network; and

(ii) for each edge (u, v) with flow f and capacity c, create two edges in the residual network: (u, v) with capacity $c - f$, and (v, u) with capacity f.

Intuitively, the forward edge (in the same direction as in the original network) records the remaining capacity, i.e., how much more flow you can "push" along the edge. The reverse edge allows one to push some flow in the reverse direction, which effectively "undoes" a previously assigned flow. The maximum amount of flow you can undo is the flow that is already assigned; hence this is the capacity of the reverse edge. This allows us to resolve from the situation of "getting stuck;" for example, in Figure 10.2 the residual network corresponding to the flow in the top-left is shown in the top-right. (Edges with zero capacity are not drawn.) There is now a path (the red, thick one in bottom left) from s to t in the residual network; this is called an **augmenting path**. We can now increase the flow by adding a flow along this path. The amount that can be increased is determined by the *minimum* of all edge capacities on this path, because that is what constrains the flow. In other words, if an augmenting path have edge capacities $c_1, c_2, \ldots, c_k$, a flow of value $\min(c_1, c_2, \ldots, c_k)$ can be pushed along this path. With the increased flow, the residual network should then be updated to reflect the new flow. Then we simply repeat the process, i.e., to find a new augmenting path in this updated residual network. Eventually, no more augmenting paths can be found. and the algorithm ends. This is the **Ford-Fulkerson algorithm**:

Algorithm 10.1 Ford-Fulkerson

```
set flow on all edges to zero
construct residual network
while there is an augmenting path p = (e_1, e_2, ...) do
  increase flow on each edge of p by min c(e_i)
  update residual network
end while
```

The algorithm does not specify which augmenting path to use, or how to find one. Any such path can be used, and, for example, breadth first search can be used to give one such path.

It can be shown that, if there is no more augmenting path in the residual network, then the flow is maximum.

Worked example. Figure 10.3 shows a slightly larger example. In the end, the dashed line divides those vertices reachable from s from those unreachable from s, after no more augmenting paths can be found.

The final flow on each edge can be recovered back from the final residual network.

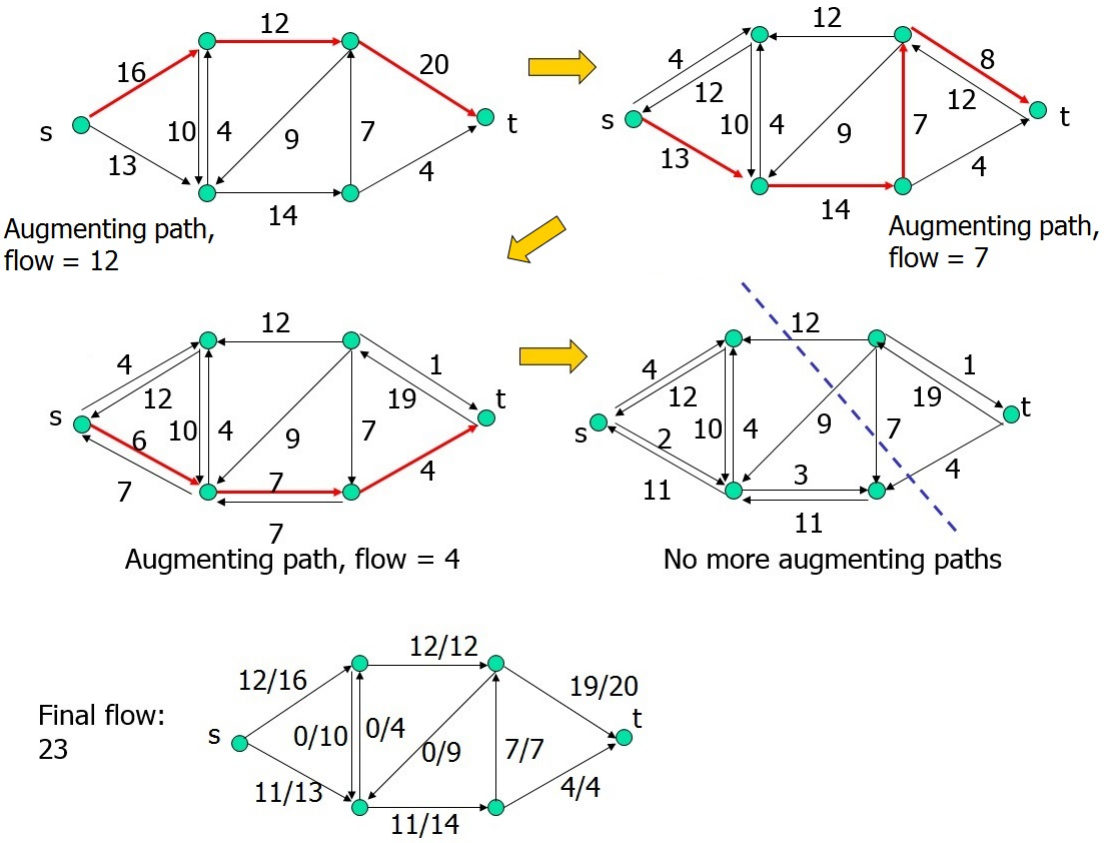


Figure 10.3: Step by step execution of the Ford-Fulkerson algorithm.

10.2.1 Running time of Ford-Fulkerson

It turns out that the running time of Ford-Fulkerson, as stated, is not straightforward to analyse. For simplicity, here we assume all the edge capacities are integers. We first state the following:

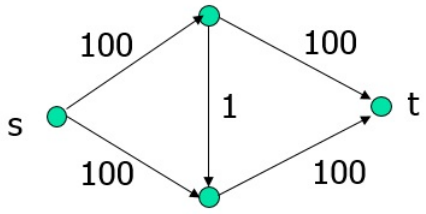
Theorem 10.1 (Integrality Theorem) *If all the edge capacities are integers, then there is a maximum flow with all flow values being integers.*

The execution of the algorithm is already a proof of this theorem – if all capacities are integers, the capacity of any augmenting path must also be an integer, and the flow added at each step is an integer, and hence the new residual capacities are all integers, and so on.

Note that the theorem does not say that all maximum flows must have this property – just one of them does. (There could be multiple flows with the same maximum value.)

With this, we can state that the running time of the algorithm is $O(mF)$, where m is the number of edges and F is maximum flow value found. The reason is as follows. Each iteration of the while loop can be completed in $O(m)$ time, because constructing the residual network takes $O(m)$ time, finding one augmenting path takes $O(m+n) = O(m)$ time, and updating the residual network also takes $O(m)$ time. How many iterations does the while loop make? Since all capacities are integers, each augmenting path found must have capacity at least 1, and hence increases the flow value by at least one. So there can be at most F iterations.

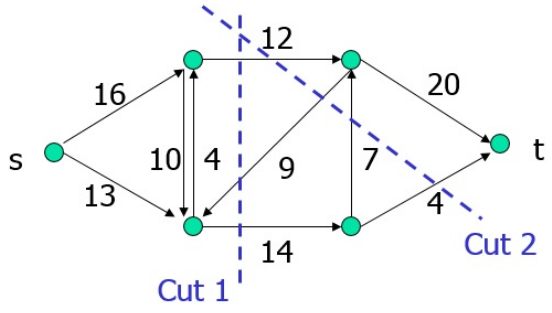
Note that this running time is somewhat different from the others we have seen, because it depends on the value of the answer we will find. We call this *output-sensitive*. This is not ideal – if one multiply all capacities of a network by 10 times, then this suggests the time complexity also increases by 10 times, which is of course quite unreasonable. But this bound is actually tight, i.e., the time complexity can actually be that bad. Consider for example the network below. Obviously, the maximum flow is 200, by passing 100 along the top and 100 along the bottom. But Ford-Fulkerson could, in principle, repeatedly find augmenting paths that use the edge with capacity 1 (it will flip repeatedly in the residual networks); thus it will take 200 iterations to complete!



If we select augmenting paths more carefully, we can prove a better running time, and indeed there are faster algorithms, but we won't cover them here.

10.3 Minimum cut

We did not prove why the Ford-Fulkerson algorithm returns the maximum flow, but we will discuss something that relate to it – the concept of a minimum cut. First, we define an *s-t cut* in a graph as a partition of the vertices into two parts A, B such that s is in A, and t is in B. The *capacity of a cut* is defined as the total capacity of the edges crossing the cut from A to B (but not from B to A). In the figure below (this is the same network as Figure 10.3), Cut 1 has capacity 26 and Cut 2 has capacity 23. There are, of course, numerous other cuts, each with a possibly different capacity; one (or more) of them has the minimum capacity – it is called the *minimum cut*. In this example Cut 2 is the minimum cut – no other cuts have a smaller value.



You might notice this minimum cut has the same value as the maximum flow in Figure 10.3. In fact, the line that divides the two parts in the minimum cut is also the same line in Figure 10.3. This is not a coincidence:

Theorem 10.2 (Max-flow-min-cut) *The maximum flow value of a network is the same as the value of the minimum cut.*

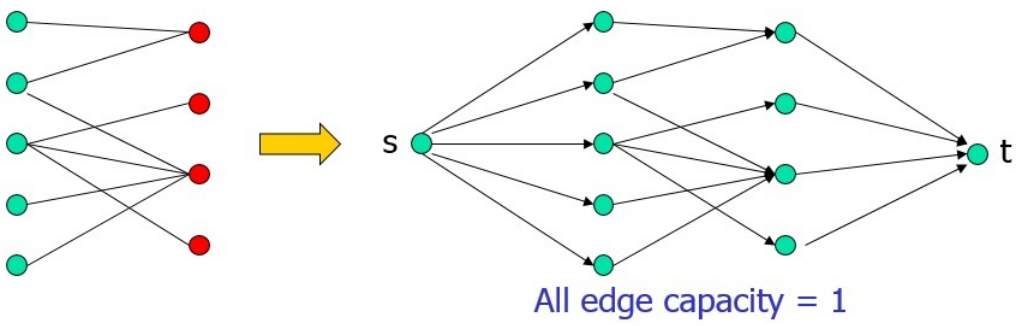
Again, we do not prove it here (it is integral to the proof that Ford-Fulkerson is optimal), but we can give some intuition. Draw any s-t cut in the graph. Any flow must pass through this cut, from the s side to the t side, using edges of this cut. Therefore, if this cut has capacity x, no flow of value larger than x can pass through it. (Imagine the border between two countries and a finite number of crossing points, each with a certain capacity.) But this argument is true for any cut, and so also for the minimum cut; therefore the maximum flow is never larger than the minimum cut. In some sense, a minimum cut is a "bottleneck" of the graph, the "narrowest" part that constrains how much traffic can flow through. (This explains why max flow is at most min cut, but why max flow is indeed equal to min cut (and not strictly smaller) requires another argument.)

The Ford-Fulkerson algorithm also gives us one way of finding a location of the min cut (not just its value). When the algorithm finishes, some vertices are reachable from s using the remaining capacities and some vertices are unreachable; the boundary between these two sets of vertices is one such location. Note that the location of a min cut may not be unique and there can be others (not found by this method).

10.4 Bipartite matching

Matching is a very common algorithmic problem and arises in many different contexts. For example, a dating service tries to match men and women; students need to be allocated to projects; university places need to be allocated to UCAS applicants; and so on. In all these problems, there are two "types" of "objects," and each object of one type has some preference on the objects of the other type. A matching is an one-to-one assignment between some objects of different types. For example, two students cannot be assigned to the same project, and a student cannot be assigned to two projects. We want to find a *maximum matching*, i.e., the number of matched pairs is as large as possible.

These types of matching problems can be modelled in **bipartite graphs**, which are graphs where the vertices can be partitioned to two disjoint subsets such that all edges of the graph only connect from one part to the other (i.e., there are no edges connecting vertices of the same "type"). The vertices on the two sides represent the objects of the two types, and there is an edge between two vertices if the two corresponding objects are allowed to match each other (i.e., they satisfy the preference conditions).

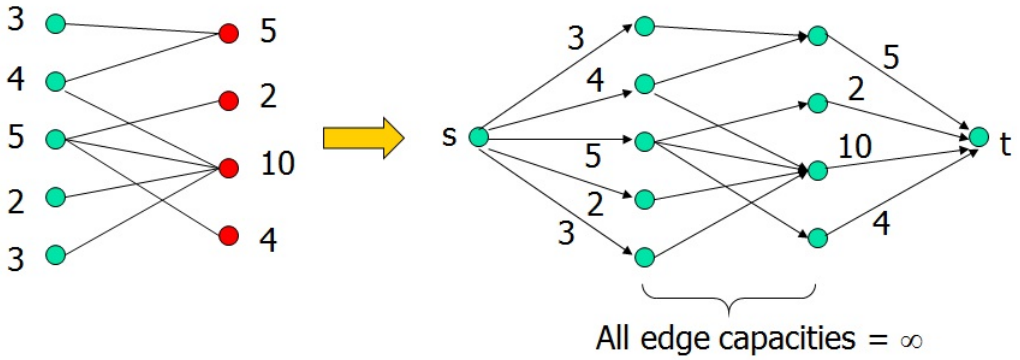


It turns out that we can solve the maximum matching problem by reducing (transforming) a bipartite matching problem into a network flow problem. First, add a source vertex s and connect it to all vertices on one side of the bipartite graph. Similarly, add a sink vertex t and connect all vertices on the other side to it. Make sure all edges are oriented to form a directed graph from s to t. Assign a capacity of 1 to all edges in the graph.

We claim that we can solve the matching problem by solving the max flow problem in this network. Specifically, we mean that the max flow of the network equals the value of the max matching. It is obvious that a matching of size x always gives a flow of value x; just follow the chosen edges in the matching and add those edges connecting s and t. Since it is a matching, none of the chosen edges in the matching share any vertices and hence the paths formed in the flow are disjoint (except at s and t), and each path gives a flow of 1.

The opposite direction needs a bit more attention. A flow of value x always gives a matching of size x because of the following. By the integrality theorem, there exist a max flow where all the flow values on edges are either 0 or 1 (because all edge capacities are 1). We pick all edges with flow 1 on them, and those in the middle (bipartite) part of the graph must define a valid matching. Since all source edges have capacity 1, a flow coming into a left-side vertex has value at most 1, and by the integrality theorem they cannot be split into two or more edges with fractional flow values; it can only go to one edge. Similarly, on the sink side, there cannot be multiple incoming flows going into a vertex because it will then lead to a larger-than-1 flow along the edge that connects to the sink, which is not possible. Thus a one-to-one matching is defined.

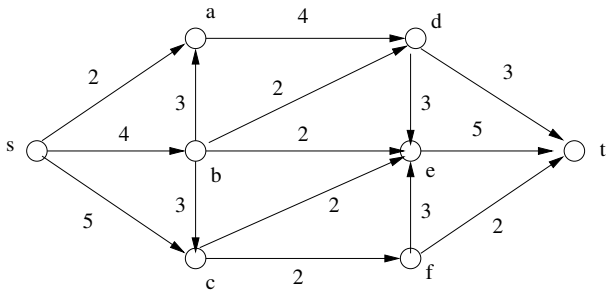
The above describes the most basic version of matching. There are many other variations of matching problems that can be modelled as flows, for example by using different edge capacities. Here we show one example. Suppose there are several depots with different amount of supplies of goods, and several retail sites with certain demands. We want to match the supplies to the demands so that the maximum amount is matched. As shown in the figure below, we can construct a similar biparite graph as before. The supply and demand requirements are modelled using edge weights: a depot that has a supply of x units can be modelled by an edge from s to that vertex with capacity x, because that means no more than x units of flow can come out of that vertex. Note that the flow that come out may be split (shared) between different retail sites. The same happens at the t side. Now, those edges at the "middle" do not need any constraints, *but we still need to specify a capacity* for this to be a proper network flow problem. We can therefore specify a capacity of infinity.



Exercises

#10-1 **(Ford-Fulkerson algorithm)**

Find the maximum flow from s to t in the network in the figure below using the Ford-Fulkerson algorithm. Show the residual network and augmenting path in each step.



#10-2 **(Minimum cut)**

What is the value of each of the following cuts for the flow network in Exercise 10-1 (where S and T are the sets of vertices belonging to the two sides of the cut)? Give the location of a minimum cut (and its value).

- $S = \{s, a, b, c\}, T = \{d, e, f, t\}$
- $S = \{s, a\}, T = \{b, c, d, e, f, t\}$
- $S = \{s, a, c, e\}, T = \{b, d, f, t\}$

10-3 **(Bipartite matching)**

A hospital has certain amounts of blood supply for each type of blood. One day a major accident happened and a large number of patients require blood transfusion. The blood types of the supply and of the patients are given in the table below. The rules for blood transfusion are the following: patients can always receive blood of the same type as their own; patients of type AB can receive any type of blood; and any patient can receive type O blood. The problem is to determine whether the blood supply is sufficient, what additional supply is needed (if any), and how the blood supply should be used to allow transfusions for all patients.

blood type	supply	demand
O	50	45
A	36	42
B	11	8
AB	8	3

(a) Describe in words how this can be transformed into a network flow problem. You should state clearly what the vertices, edges and edge capacities are.

(b) Construct the network according to (a) for this particular example.

(c) Find the maximum flow and hence solve the blood transfusion problem.