UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Algorithms

Academic year 2013–2014

Lent term 2014

`http://www.cl.cam.ac.uk/teaching/1314/Algorithms/`

`frank.stajano--algs@cl.cam.ac.uk` (author of this handout)

`thomas.sauerwald@cl.cam.ac.uk`

Revised 2014 edition

Revision 8 of 2014-01-06 00:28:25 +0000 (Mon, 06 Jan 2014).

Contents

Preliminaries

Course content and textbooks

This course is about some of the coolest stuff a programmer can do.

Most real-world programming is conceptually pretty simple. The undeniable difficulties come primarily from size: enormous systems with millions of lines of code and complex APIs that won't all comfortably fit in a single brain. But each piece usually does something pretty bland, such as moving data from one place to another and slightly massaging it along the way.

Here, it's different. We look at pretty advanced hacks—those ten-line chunks of code that make you want to take your hat off and bow.

The only way really to understand this material is to program and debug it yourself, and then run your programs step by step on your own examples, visualizing intermediate results along the way. You might think you are fluent in n programming languages but you aren't really a programmer until you've written and debugged some hairy pointer-based code such as that required to cut and splice the doubly-linked lists used in Fibonacci trees. (Once you do, you'll know why.)

However the course itself isn't about programming: it's about designing and analysing algorithms and data structures—the ones that great programmers then write up as tight code and put in libraries for other programmers to reuse. It's about finding smart ways of solving difficult problems, and about measuring different solutions to see which one is actually smarter.

In order to gain a more than superficial understanding of the course material you will also need a full-length textbook, for which this handout is *not* a substitute. The one I recommend is a classic, adopted at many of the top universities around the world, and which in 2012 was *the* most cited book in computer science:

> [CLRS3] Cormen, Leiserson, Rivest, Stein. *Introduction to Algorithms, Third edition.* MIT press, 2009. ISBN 978-0-262-53305-8.

A heavyweight book at about 1300 pages, it covers a little more material and at slightly greater depth than most others. It includes careful mathematical treatment of the algorithms it discusses and is a natural candidate for a reference shelf. Despite its bulk and precision this book is written in a fairly friendly style. At some point it was also *the computer science book with the highest number of citations*! You can't properly call yourself a computer scientist if CLRS3 is not on your bookshelf. It's the default text for this course: by all means feel free to refer to other books too, but chapter references in the chapter headings of these notes are to this textbook.

Other algorithms textbooks worth consulting as optional additional references include at least Sedgewick; Kleinberg and Tardos; and of course the legendary Knuth. Full bibliographic details are in the syllabus and on the course web page. However none of these textbooks covers *all* the topics in the syllabus, so you're still better off getting yourself a copy of CLRS3 (which by the way, in spite of its quality, is also the cheapest of the bunch).

Note also that a growing number of usually accurate algorithm and data structure descriptions can be found on Wikipedia. Once you master the material in this course, and especially after you've earned some experience by writing and debugging your own implementation, consider improving any descriptions that you find lacking or unclear.

What is in these notes

These notes are meant as a clear and concise reference but they are not a substitute for having your own copy of the recommended textbook. They can help you navigate through the textbook and can help when organizing revision. I recommend that you come to each lecture with this handout *and* a paper notebook, and use the latter to take notes and sketch diagrams. In your own time, use the same notebook to write down your (attempts at) solutions to exercises as you prepare for upcoming supervisions.

These notes contain short exercises, highlighted by boxes, that you would do well to solve as you go along to prove that you are not just reading on autopilot. They tend to be easy (most are meant to take not more than five minutes each) and are therefore insufficient to help you really *own* the material covered here. For that, program the algorithms yourself[1] and solve problems found in your textbook or assigned by your supervisor. There is a copious supply of past exam questions at `http://www.cl.cam.ac.uk/teaching/exams/pastpapers/` under "Algorithms", "Algorithms I" "Algorithms II" and "Data Structures and Algorithms" but be sure to choose (or ask your supervisor to help you choose) ones that are covered by this year's syllabus, because the selection of topics offered in these courses has evolved throughout the years.

Acknowledgements and history

I wrote my first version of these notes in 2005, for a 16-lecture course[2] entitled "Data Structures and Algorithms", building on the excellent notes for that course originally written by Arthur Norman and then enhanced by Roger Needham (my academic grandfather—the PhD supervisor of my PhD supervisor) and Martin Richards. I hereby express my gratitude to my illustrious predecessors. In later years the course evolved into two (Algorithms I and II, to first and second year students respectively) and the aggregate number of lectures gradually expanded from 16 to 27, until the 2013–2014 reorganization that brought all this material back into the first year, for a 24-lecture course I now teach with Dr Sauerwald.

[1]The more programs you write to recreate what we show you in the lectures, the more you will really own this material.

[2]16 lectures including 4 on what effectively was remedial discrete mathematics, thus in fact only 12 lectures of data structures and algorithms proper.

Although I don't know where they are, from experience I am pretty sure that these notes still contains bugs, as all non-trivial documents do. Consult the course web page for the errata corrige. I am grateful to Kay Henning Brodersen, Sam Staton, Simon Spacey, Rasmus King, Chloë Brown, Rob Harle, Larry Paulson, Daniel Bates, Tom Sparrow, Marton Farkas, Wing Yung Chan, Tom Taylor, Trong Nhan Dao, Oliver Allbless, Aneesh Shukla, Christian Richardt, Long Nguyen, Michael Williamson, Myra VanInwegen, Manfredas Zabarauskas, Ben Thorner, Simon Iremonger, Heidi Howard, Tom Sparrow, Simon Blessenohl, Nick Chambers, Nicholas Ngorok, Miklós András Danka and particularly Alastair Beresford and Jan Polášek for sending me corrections to previous editions. If you find any more corrections and email them to me, I'll credit you in any future revisions (unless you prefer anonymity). The responsibility for any remaining mistakes remains of course mine.

Chapter 1

What's the point of all this?

1.1 What is an algorithm?

An **algorithm** is a systematic recipe for solving a problem. By "systematic" we mean that the problem being solved will have to be specified quite precisely and that, before any algorithm can be considered complete, it will have to be provided with a proof that it works and an analysis of its performance. In a great many cases, all of the ingenuity and complication in algorithms is aimed at making them fast (or at reducing the amount of memory that they use) so a justification that the intended performance will be attained is very important.

In this course you will learn, among other things, a variety of "prior art" algorithms and data structures to address recurring computer science problems; but what would be especially valuable to you is acquiring the skill to invent new algorithms and data structures to solve difficult problems you weren't taught about. The best way to make progress towards that goal is to participate in this course actively, rather than to follow it passively. To help you with that, here are three difficult problems for which you should try to come up with suitable algorithms and data structures. The rest of the course will eventually teach you good ways to solve these problems but you will have a much greater understanding of the answers and of their applicability if you attempt to solve these problems on your own before you are given the solution.

1.2 DNA sequences

In bioinformatics, a recurring activity is to find out how "similar" two given DNA sequences are. For the purposes of this simplified problem definition, assume that a DNA sequence is a string of arbitrary length over the alphabet {A, C, G, T} and that the degree of similarity between two such sequences is measured by the length of their longest common subsequence, as defined next. A subsequence T of a sequence S is any string obtained by dropping zero or more characters from S; for example, if $S =$ AGTGTACCCAT, then the following are valid subsequences: AGGTAAT (=AG~~T~~GTA~~CCC~~AT), TACAT (=~~AGTG~~TA~~C~~C~~C~~AT), GGT (=~~A~~G~~T~~GT~~ACCCAT~~); but the following are not: AAG, TCG. You must find an algorithm that, given two sequences X and Y of arbitrary but finite lenghts, returns a sequence Z of maximal length that is a subsequence of both X and Y[1].

You might wish to try your candidate algorithm on the following two sequences: X = `CCGTCAGTCGCG`, Y = `TGTTTCGGAATGCAA`. What is the longest subsequence you obtain? Are there any others of that length? Are you sure that there exists no longer common subsequence? How long do you estimate your algorithm would take to complete, on your computer, if the sequences were about 30 characters each? Or 100? Or a million?

1.3 Bestseller chart

Imagine an online bookstore with millions of books in its catalogue. For each book, the store keeps track of how many copies it sold. Every day the new sales figures come in and a web page is compiled with a list of the top 100 best sellers. How would you generate such a list? How long would it take to run this computation? How long would it take if, hypothetically, the store had *trillions* of different items for sale instead of merely millions? Of course you could re-sort the whole catalogue each time and take the top 100 items, but can you do better? And is it cheaper to maintain the chart up to date after each sale or to recompute it from scratch once a day?

1.4 Database indexing

Imagine a very large database (e.g. microbilling for a telecomms operator or bids history for an online auction site), with several indices over different keys, so that you can sequentially walk through the database records in order of account number but also, alternatively, by transaction date or by value or by surname. Each index has one entry per record (containing the key and the disk address of the record) but there are so many records that even the indices (never mind the records) are too large to fit in RAM and must themselves be stored as files on disk. What is an efficient way of retrieving a particular record given its key, if we consider scanning the whole index linearly as too slow? Can we arrange the data in some other way that would speed up this operation? And, once you have thought of a specific solution: how would you keep your new indexing data structure up to date when adding or deleting records to the original database?

1.5 Questions to ask

I recommend you spend some time attacking the three problems above, as seriously as if they were exam questions, before going any further with the course. Then, after each new lecture, ask yourself whether what you learnt that day gives any insight towards a better solution. The first and most obvious question (and the one often requiring the greatest creativity) is of course:

- What strategy to use? What is the algorithm? What is the data structure?

But there are several other questions that are important too.

[1]There may be several.

- Is the algorithm correct? How can we prove that it is?
- How long does it take to run? How long would it take to run on a much larger input? Besides, since computers get faster and cheaper all the time, how long would it take to run on a different type of computer, or on the computer I will be able to buy in a year, or in three years, or in ten?
- If there are several possible algorithms, all correct, how can we compare them and decide which is best? If we rank them by speed on a certain computer and a certain input, will this ranking carry over to other computers and other inputs? And what other ranking criteria should we consider, if any, other than speed?

Your goal for this course should be to learn general methods for answering all of these questions, regardless of the specific problem.

Chapter 2

Sorting

Chapter contents

Review of complexity and O-notation. Trivial sorting algorithms of quadratic complexity. Review of merge sort and quicksort, understanding their memory behaviour on statically allocated arrays. Minimum cost of sorting. Heapsort. Stability. Other sorting methods including sorting in linear time. Median and order statistics. Expected coverage: about 4 lectures.

Our look at algorithms starts with sorting, which is a big topic: any course on algorithms, including *Foundations of Computer Science* that precedes this one, is bound to discuss a number of sorting methods. Volume 3 of Knuth (almost 800 pages) is entirely dedicated to sorting (covering over two dozen algorithms) and the closely related subject of searching, so don't think this is a small or simple topic! However much is said in this lecture course, there is a great deal more that is known.

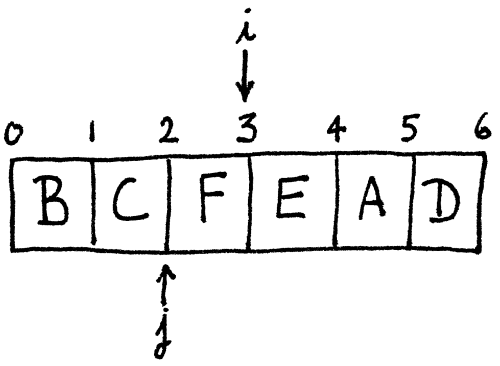
Some lectures in this chapter will cover algorithms (such as insertion sort, merge sort and quicksort) to which you have been exposed before from a functional language (ML) perspective. While these notes attempt to be self-contained, the lecturer may go more quickly through the material you have already seen. During this second pass you should pay special attention to issues of memory allocation and array usage which were not apparent in the functional presentation.

2.1 Insertion sort

Let us approach the problem of sorting a sequence of items by modelling what humans spontaneously do when arranging in their hand the cards they were dealt in a card game: you keep the cards in your hand in order and you insert each new card in its place as it comes.

We shall look at data types in greater detail later on in the course but let's assume you already have a practical understanding of the "array" concept: a sequence of adjacent "cells" in memory, indexed by an integer. If we implement the hand as an array `a[]` of adequate size, we might put the first card we receive in cell `a[0]`, the next in cell `a[1]` and so on. Note that one thing we cannot actually do with the array, even though it is natural with lists or when handling physical cards, is to insert a new card between `a[0]` and `a[1]`: if we need to do that, we must first shift right all the cells after `a[0]`, to create an unused space in `a[1]`, and then write the new card there.

Let's assume we have been dealt a hand of n cards[1], now loaded in the array as `a[0]`, `a[1]`, `a[2]`, ..., `a[`n`-1]`, and that we want to sort it. We pretend that all the cards are still face down on the table and that we are picking them up one by one in order. Before picking up each card, we first ensure that all the preceding cards in our hand have been sorted.



A little aside prompted by the diagram. A common source of bugs in computing is the off-by-one error. Try implementing insertsort on your own and you might accidentally trip over an off-by-one error yourself, before producing a debugged version. You are hereby encouraged to acquire a few good hacker habits that will reduce the chance of your falling into that particular pitfall. One is to number items from zero rather than from one—then most "offset + displacement" calculations will just work. Another is to adopt the convention used in this diagram when referring to arrays, strings, bitmaps and other structures with lots of linearly numbered cells: the index always refers to the position *between* two cells and gives its name to the cell to its right. (The Python programming language, for example, among its many virtues, uses this convention consistently.) Then the difference between two indices gives you the number of cells between them. So for example the subarray `a[2:5]`, containing the elements from `a[2]` included to `a[5]` excluded, has $5 - 2 = 3$ elements, namely `a[2] = "F"`, `a[3] = "E"` and `a[4] = "A"`.

When we pick up card `a[`i`]`, since the first i items of the array have been sorted, the next is inserted in place by letting it sink towards the left down to its rightful place: it is compared against the item at position j (with j starting at $i - 1$, the rightmost of the already-sorted elements) and, if smaller than it, a swap moves it down. If the new element does move down, then so does the j pointer; and then the new element is again compared against the one in position j and swapped if necessary, until it gets to its place. We can write down this algorithm in pseudocode as follows:

[1]Each card is represented by a capital letter in the diagram so as to avoid confusion between card numbers and index numbers. Letters have the obvious order implied by their position in the alphabet and thus $A < B < C < D\ldots$, which is of course also true of their Ascii or Unicode code.

```
0  def insertSort(a):
1      """BEHAVIOUR: Run the insertsort algorithm on the integer
2      array a, sorting it in place.
3
4      PRECONDITION: array a contains len(a) integer values.
5
6      POSTCONDITION: array a contains the same integer values as before,
7      but now they are sorted in ascending order."""
8
9      for i from 1 included to len(a) excluded:
10         # ASSERT: the first i positions are already sorted.
11
12         # Insert a[i] where it belongs within a[0:i].
13         j = i - 1
14         while j >= 0 and a[j] > a[j + 1]:
15             swap(a[j], a[j + 1])
16             j = j - 1
```

Pseudocode is an informal notation that is pretty similar to real source code but which omits any irrelevant details. For example we write `swap(x,y)` instead of the sequence of three assignments that would normally be required in many languages. The exact syntax is not terribly important: what matters more is clarity, brevity and conveying the essential ideas and features of the algorithm. It should be trivial to convert a piece of well-written pseudocode into the programming language of your choice.

Exercise 1
Assume that each `swap(x, y)` means three assignments (namely `tmp = x; x = y; y = tmp`). Improve the insertsort algorithm pseudocode shown in the handout to reduce the number of assignments performed in the inner loop.

2.2 Is the algorithm correct?

How can we convince ourselves (and our customers) that the algorithm is correct? In general this is far from easy. An essential first step is to specify the objectives as clearly as possible: to paraphrase Virgil Gligor, who once said something similar about attacker modelling, without a specification the algorithm can never be correct or incorrect—only surprising!

In the pseudocode above, we provided a (slightly informal) specification in the documentation string for the routine (lines 1–7). The *precondition* (line 4) is a request, specifying what the routine expects to receive from its caller; while the *postcondition* (lines 6–7) is a promise, specifying what the routine will do for its caller (provided that the precondition is satisfied on call). The pre- and post-condition together form a kind of "contract", using the terminology of Bertrand Meyer, between the routine and its caller[2]. This is a good way to provide a specification.

[2]If the caller fails to uphold her part of the bargain and invokes the routine while the precondition

There is no universal method for proving the correctness of an algorithm; however, a strategy that has very broad applicability is to reduce a large problem to a suitable sequence of smaller subproblems to which you can apply mathematical induction[3]. Are we able to do so in this case?

To reason about the correctness of an algorithm, a very useful technique is to place key *assertions* at appropriate points in the program. An assertion is a statement that, whenever that point in the program is reached, a certain property will always be true. Assertions provide "stepping stones" for your correctness proof; they also help the human reader understand what is going on and, by the same token, help the programmer debug the implementation[4]. Coming up with good invariants is not always easy but is a great help for developing a convincing proof (or indeed for discovering bugs in your algorithm while it isn't correct yet). It is especially helpful to find a good, meaningful invariant at the beginning of each significant loop. In the algorithm above we have an invariant on line 10, at the beginning of the main loop: the i^{th} time we enter the loop, it says, the previous passes of the loop will have sorted the leftmost i cells of the array. How? We don't care, but our job now is to prove the inductive step: assuming the assertion is true when we enter the loop, we must prove that *one* further pass down the loop will make the assertion true when we reenter. Having done that, and having verified that the assertion holds for the trivial case of the first iteration ($i = 0$; it obviously does, since the first *zero* positions cannot possibly be out of order), then all that remains is to check that we achieve the desired result (whole array is sorted) at the end of the last iteration.

Check the recommended textbook for further details and a more detailed walkthrough, but this is the jist of a powerful and widely applicable method for proving the correctness of your algorithm.

Exercise 2
Provide a useful invariant for the inner loop of insertion sort, in the form of an assertion to be inserted between the "while" line and the "swap" line.

is not true, the routine cannot be blamed if it doesn't return the correct result. On the other hand, if the caller ensures that the precondition is true before calling the routine, the routine will be considered faulty unless it returns the correct result. The "contract" is as if the routine said to the caller: "provided you satisfy the precondition, I shall satisfy the postcondition".

[3]Mathematical induction in a nutshell: "How do I solve the case with k elements? I don't know, but assuming someone smarter than me solved the case with $k-1$ elements, I could tell you how to solve it for k elements starting from *that*"; then, if you also independently solve a starting point, e.g. the case of $k = 0$, you've essentially completed the job.

[4]Many modern programming languages allow you to write assertions as program statements (as opposed to comments); then the expression being asserted is evaluated at runtime and, if it is not true, an exception is raised; this alerts you as early as possible that something isn't working as expected, as opposed to allowing the program to continue running while in an inconsistent state.

2.3 Computational complexity

2.3.1 Abstract modelling and growth rates

How can we estimate the time that the algorithm will take to run if we don't know how big (or how jumbled up) the input is? It is almost always necessary to make a few simplifying assumptions before performing cost estimation. For algorithms, the ones most commonly used are:

1. We only worry about the worst possible amount of time that some activity could take.
2. Rather than measuring absolute computing times, we only look at *rates of growth* and we ignore constant multipliers. If the problem size is n, then $100000 f(n)$ and $0.000001 f(n)$ will both be considered equivalent to just $f(n)$.
3. Any finite number of exceptions to a cost estimate are unimportant so long as the estimate is valid for all large enough values of n.
4. We do not restrict ourselves to just reasonable values of n or apply any other reality checks. Cost estimation will be carried through as an abstract mathematical activity.

Despite the severity of all these limitations, cost estimation for algorithms has proved very useful: almost always, the indications it gives relate closely to the practical behaviour people observe when they write and run programs.

The notations big-O, Θ and Ω, discussed next, are used as short-hand for some of the above cautions.

2.3.2 Big-O, Θ and Ω notations

A function $f(n)$ is said to be $O(g(n))$ if there are constants k and N, all > 0, such that $0 \leq f(n) \leq k \cdot g(n)$ whenever $n > N$. In other words, $g(n)$ provides an upper bound that, for sufficiently large values of n, $f(n)$ will never exceed[5], except for what can be compensated by a constant factor. In informal terms: $f(n) \in O(g(n))$ means that $f(n)$ grows *at most* like $g(n)$.

A function $f(n)$ is said to be $\Theta(g(n))$ if there are constants k_1, k_2 and N, all > 0, such that $0 \leq k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n)$ whenever $n > N$. In other words, for sufficiently large values of n, the functions $f()$ and $g()$ agree within a constant factor. This constraint is much stronger than the one implied by Big-O. In informal terms: $f(n) \in \Theta(g(n))$ means that $f(n)$ grows *exactly* like $g(n)$.

Some authors also use $\Omega()$ as the dual of $O()$ to provide a lower bound. In informal terms: $f(n) \in \Omega(g(n))$ means that $f(n)$ grows *at least* like $g(n)$.

Some authors also use lowercase versions of $O()$ and $\Omega()$ to make a subtle point. The "big" versions describe asymptotic bounds that might or might not be tight; informally,

[5]We add the "greater than zero" constraint to avoid confusing cases of a $f(n)$ with a high growth rate dominated by a $g(n)$ with a low growth rate because of sign issues, e.g. $f(n) = -n^3$ which is $< g(n) = n$ for any $n > 0$.

$O()$ is like $\leq$ and $\Omega()$ is like $\geq$. The "small" versions, instead, describe asymptotic bounds that are definitely not tight: informally, $o()$ is like $<$ and $\omega()$ is like $>$.

Here is a very informal[6] summary table:

	If...	then $f(n)$ grows ... $g(n)$.	$f(n)$ ... $g(n)$
small-o	$f(n) \in o(g(n))$	strictly more slowly than	$<$
big-o	$f(n) \in O(g(n))$	at most like	$\leq$
big-theta	$f(n) \in \Theta(g(n))$	exactly like	$=$
big-omega	$f(n) \in \Omega(g(n))$	at least like	$\geq$
small-omega	$f(n) \in \omega(g(n))$	strictly more quickly than	$>$

Note that none of these notations says anything about $f(n)$ being a computing time estimate, even though that will be the most common use in this lecture course.

Note also that it is common to say that $f(n) = O(g(n))$, with "=" instead of "$\in$". This is formally incorrect but it's a broadly accepted custom, so we shall adopt it too from now on.

Various important computer procedures have costs that grow as $O(n \log(n))$ and a gut-feeling understanding of logarithms will be useful to follow this course. Formalities apart, the only really important thing to understand about logarithms is that they tell you how many digits there are in a numeral. If this isn't intuitive, then any fancy algebra you may be able to perform on logarithms will be practically useless.

In the proofs, the logarithms will often come out as ones to base 2—which, following Knuth, we shall indicate as "lg": for example, $\lg(1024) = \log_2(1024) = 10$. But observe that $\log_2(n) = \Theta(\log_{10}(n))$ (indeed a stronger statement could be made—the ratio between them is utterly fixed); so, with Big-O or Θ or Ω notation, there is no real need to worry about the base of logarithms—all versions are equally valid.

The following exercise contains a few examples that may help explain, even if (heavens forbid) you don't actually do the exercise.

Exercise 3

$$\begin{aligned}
|sin(n)| &= O(1) \\
|sin(n)| &\neq \Theta(1) \\
200 + sin(n) &= \Theta(1) \\
123456n + 654321 &= \Theta(n) \\
2n - 7 &= O(17n^2) \\
\lg(n) &= O(n) \\
\lg(n) &\neq \Theta(n) \\
n^{100} &= O(2^n) \\
1 + 100/n &= \Theta(1)
\end{aligned}$$

For each of the above "=" lines, identify the constants k, k_1, k_2, N as appropriate. For each of the "$\neq$" lines, show they can't possibly exist.

[6] For sufficiently large n, within a constant factor and blah blah blah.

Please note the distinction between the value of a function and the amount of time it may take to compute it: for example $n!$ can be computed in $O(n)$ arithmetic operations, but has value bigger than $O(n^k)$ for any fixed k.

2.3.3 Models of memory

Through most of this course there will be a tacit assumption that the computers used to run algorithms will always have enough memory, and that this memory can be arranged in a single address space so that one can have unambiguous memory addresses or pointers. Put another way, we pretend you can set up a single array of integers that is as large as you will ever need.

There are of course practical ways in which this idealization may fall down. Some archaic hardware designs may impose quite small limits on the size of any one array, and even current machines tend to have but finite amounts of memory, and thus upper bounds on the size of data structure that can be handled.

A more subtle issue is that a truly unlimited memory will need integers (or pointers) of unlimited size to address it. If integer arithmetic on a computer works in a 32-bit representation (as was common until very recently, and still is on most embedded systems) then the largest integer value that can be represented is certainly less than 2^{32} and so one can not sensibly talk about arrays with more elements than that. This limit represents only 4 gigabytes of main memory, which these days is the amount installed by default in the kind of run-of-the-mill computer you can pick up in a supermarket next to your groceries. The solution is obviously that the width of integer subscripts used for address calculation has to increase with the logarithm of the size of a memory large enough to accommodate the problem. So, to solve a hypothetical problem that needed an array of size 10^{100}, all subscript arithmetic would have to be done using 100 decimal digit precision.

It is normal in the analysis of algorithms to ignore these problems and assume that any element `a[i]` of an array can be accessed in unit time, however large the array is. The associated assumption is that integer arithmetic operations needed to compute array subscripts can also all be done at unit cost. This makes good practical sense since the assumption holds pretty well true for all problems—or at least for most of those you are actually likely to *want* to tackle on a computer.

Strictly speaking, though, on-chip caches in modern processors make the last paragraph incorrect. In the good old days, all memory references used to take unit time. Now, since processors have become faster at a much higher rate than memory[7], CPUs use super fast (and expensive and comparatively small) cache stores that can typically serve up a memory value in one or two CPU clock ticks; however, when a cache miss occurs, it often takes tens or even hundreds of ticks. Locality of reference is thus becoming an issue, although one which most textbooks on algorithms still largely ignore for the sake of simplicity of analysis.

[7]This phenomenon is referred to as "the memory gap".

2.3.4 Models of arithmetic

The normal model for computer arithmetic used here will be that each arithmetic operation[8] takes unit time, irrespective of the values of the numbers being combined and regardless of whether fixed or floating point numbers are involved. The nice way that Θ notation can swallow up constant factors in timing estimates generally justifies this. Again there is a theoretical problem that can safely be ignored in almost all cases: in the specification of an algorithm (or of an Abstract Data Type) there may be some integers, and in the idealized case this will imply that the procedures described apply to arbitrarily large integers, including ones with values that will be many orders of magnitude larger than native computer arithmetic will support directly[9]. In the fairly rare cases where this might arise, cost analysis will need to make explicit provision for the extra work involved in doing multiple-precision arithmetic, and then timing estimates will generally depend not only on the number of values involved in a problem but on the number of digits (or bits) needed to specify each value.

2.3.5 Worst, average and amortized costs

Usually the simplest way of analyzing an algorithm is to find the worst-case performance. It may help to imagine that somebody else is proposing the algorithm, and you have been challenged to find the very nastiest data that can be fed to it to make it perform really badly. In doing so you are quite entitled to invent data that looks very unusual or odd, provided it comes within the stated range of applicability of the algorithm. For many algorithms the "worst case" is approached often enough that this form of analysis is useful for realists as well as pessimists.

Average case analysis ought by rights to be of more interest to most people, even though worst case costs may be really important to the designers of systems that have real-time constraints, especially if there are safety implications in failure. But, before useful average cost analysis can be performed, one needs a model for the probabilities of all possible inputs. If in some particular application the distribution of inputs is significantly skewed, then analysis based on uniform probabilities might not be valid. For worst case analysis it is only necessary to study one limiting case; for average analysis the time taken for every case of an algorithm must be accounted for—and this, usually, makes the mathematics a lot harder.

Amortized analysis[10] is applicable in cases where a data structure supports a number of operations and these will be performed in sequence. Quite often the cost of any particular operation will depend on the history of what has been done before; and, sometimes, a plausible overall design makes most operations cheap at the cost of occasional expensive internal reorganization of the data. Amortized analysis treats the cost of this re-organization as the joint responsibility of all the operations previously performed on the data structure and provides a firm basis for determining if it was worthwhile. It is typically more technically demanding than just single-operation worst-case analysis.

[8]Not merely the ones on array subscripts mentioned in the previous section.

[9]Or indeed, in theory, larger than the whole main memory can even hold! After all, the entire RAM of your computer might be seen as just one very long binary integer.

[10]To be studied in the *Algorithms II* course; peek ahead to chapter 17 of CLRS3 if you are curious.

A good example of where amortized analysis is helpful is garbage collection, where it allows the cost of a single large expensive storage reorganization to be attributed to each of the elementary allocation transactions that made it necessary. Note that (even more than is the case for average cost analysis) amortized analysis is not appropriate for use where real-time constraints apply.

2.4 How much does insertion sort cost?

Having understood the general framework of asymptotic worst-case analysis and the simplifications of the models we are going to adopt, what can we say about the cost of running the insertion sort algorithm we previously recalled? If we indicate as n the size of the input array to be sorted, and as $f(n)$ the very precise (but very difficult to accurately represent in closed form) function giving the time taken by our algorithm to compute an answer on the worst possible input of size n, on a specific computer, then our task is not to find an expression for $f(n)$ but merely to identify a much simpler function $g(n)$ that works as an upper bound, i.e. a $g(n)$ such that $f(n) = O(g(n))$. Of course a loose upper bound is not as useful as a tight one: if $f(n) = O(n^2)$, then $f(n)$ is also $O(n^5)$, but the latter doesn't tell us as much.

Once we have a reasonably tight upper bound, the fact that the big-O notation eats away constant factors allows us to ignore the differences between the various computers on which we might run the program.

If we go back to the pseudocode listing of insertsort found on page 13, we see that the outer loop of line 9 is executed exactly $n-1$ times (regardless of the values of the elements in the input array), while the inner loop of line 14 is executed a number of times that depends on the number of swaps to be performed: if the new card we pick up is greater than any of the previously received ones, then we just leave it at the rightmost end and the inner loop is never executed; while if it is smaller than any of the previous ones it must travel all the way through, forcing as many executions as the number of cards received until then, namely i. So, in the worst case, during the i^{th} invocation of the outer loop, the inner loop will be performed i times. In total, therefore, for the whole algorithm, the inner loop (whose body consists of a constant number of elementary instructions) is executed a number of times that won't exceed the n^{th} triangular number, $\frac{n(n+1)}{2}$. In big-O notation we ignore constants and lower-order terms, so we can simply write $O(n^2)$.

Note that it is possible to implement the algorithm slightly more efficiently at the price of complicating the code a little bit, as suggested in another exercise on page 13.

Exercise 4
What is the asymptotic complexity of the variant of insertsort that does fewer swaps?

2.5 Minimum cost of sorting

We just established that insertion sort has a worst-case asymptotic cost dominated by the square of the size of the input array to be sorted (we say in short: "insertion sort has quadratic cost"). Is there any possibility of achieving better asymptotic performance with some other algorithm?

If I have n items in an array, and I need to rearrange them in ascending order, whatever the algorithm there are two elementary operations that I can plausibly expect to use repeatedly in the process. The first (comparison) takes two items and compares them to see which should come first[11]. The second (exchange) swaps the contents of two nominated array locations.

In extreme cases either comparisons or exchanges[12] may be hugely expensive, leading to the need to design methods that optimize one regardless of other costs. It is useful to have a limit on how good a sorting method could possibly be, measured in terms of these two operations.

Assertion 1 (lower bound on exchanges). If there are n items in an array, then $\Theta(n)$ exchanges always suffice to put the items in order. In the worst case, $\Theta(n)$ exchanges are actually needed.

Proof. Identify the smallest item present: if it is not already in the right place, one exchange moves it to the start of the array. A second exchange moves the next smallest item to place, and so on. After at worst $n-1$ exchanges, the items are all in order. The bound is $n-1$ rather than n because at the very last stage the biggest item has to be in its right place without need for a swap—but that level of detail is unimportant to Θ notation.

Exercise 5
The proof of Assertion 1 (lower bound on exchanges) convinces us that $\Theta(n)$ exchanges are always *sufficient*. But why isn't that argument good enough to prove that they are also *needed*?

Conversely, consider the case where the original arrangement of the data is such that the item that will need to end up at position i is stored at position $i+1$ (with the natural wrap-around at the end of the array). Since every item is in the wrong position, you must perform enough exchanges to touch each position in the array and that certainly means at least $n/2$ exchanges, which is good enough to establish the $\Theta(n)$ growth rate. Tighter analysis would show that more exchanges are in fact needed in the worst case.

Assertion 2 (lower bound on comparisons). Sorting by pairwise comparison, assuming that all possible arrangements of the data might actually occur as input, necessarily costs at least $\Omega(n \lg n)$ comparisons.

Proof. As you saw in *Foundations of Computer Science*, there are $n!$ permutations of n items and, in sorting, we in effect identify one of these. To discriminate between that

[11]Indeed, to start with, this course will concentrate on sorting algorithms where the *only* information about where items should end up will be that deduced by making pairwise comparisons.

[12]Often, if exchanges are costly, it can be useful to sort a vector of pointers to objects rather than a vector of the objects themselves—exchanges in the pointer array will be cheap.

many cases we need at least $\lceil \log_2(n!) \rceil$ binary tests. Stirling's formula tells us that $n!$ is roughly n^n, and hence that $\lg(n!)$ is about $n \lg n$.

Note that this analysis is applicable to any sorting method that uses any form of binary choice to order items, that it provides a lower bound on costs but does not guarantee that it can be attained, and that it is talking about worst case costs. Concerning the last point, the analysis can be carried over to average costs when all possible input orders are equally probable.

For those who can't remember Stirling's name or his formula, the following argument is sufficient to prove that $\lg(n!) = \Theta(n \lg n)$.

$$\lg(n!) = \lg n + \lg(n-1) + \ldots + \lg(1)$$

All n terms on the right are less than or equal to $\lg n$ and so

$$\lg(n!) \leq n \lg n.$$

Therefore $\lg(n!)$ is bounded by $n \lg n$. Conversely, since the lg function is monotonic, the first $n/2$ terms, from $\lg n$ to $\lg(n/2)$, are all greater than or equal to $\lg(n/2) = \lg n - \lg 2 = (\lg n) - 1$, so

$$\lg(n!) \quad \geq \quad \frac{n}{2}(\lg n - 1) + \lg(n/2) + \ldots + \lg(1) \quad \geq \quad \frac{n}{2}(\lg n - 1),$$

proving that, when n is large enough, $n \lg n$ is bounded by $k \lg(n!)$ (for $k = 3$, say). Thus $\lg(n!) = \Theta(n \lg n)$.

2.6 Selection sort

In the previous section we proved that an array of n items may be sorted by performing no more than $n - 1$ exchanges. This provides the basis for one of the simplest sorting algorithms known: selection sort. At each step it finds the smallest item in the remaining part of the array and swaps it to its correct position. This has, as a sub-algorithm, the problem of identifying the smallest item in an array. The sub-problem is easily solved by scanning linearly through the (sub)array, comparing each successive item with the smallest one found so far. If there are m items to scan, then finding the minimum clearly costs $m - 1$ comparisons. The whole selection-sort process does this on a sequence of sub-arrays of size $n, n-1, \ldots, 1$. Calculating the total number of comparisons involved requires summing an arithmetic progression, again yielding a triangular number and a total cost of $\Theta(n^2)$. This very simple method has the advantage (in terms of how easy it is to analyse) that the number of comparisons performed does not depend at all on the initial organization of the data, unlike what happened with insert-sort.

```
def selectSort(a):
    """BEHAVIOUR: Run the selectsort algorithm on the integer
    array a, sorting it in place.

    PRECONDITION: array a contains len(a) integer values.

    POSTCONDITION: array a contains the same integer values as before,
    but now they are sorted in ascending order."""

    for k from 0 included to len(a) excluded:
        # ASSERT: the array positions before a[k] are already sorted

        # Find the smallest element in a[k:END] and swap it into a[k]
        iMin = k
        for j from iMin + 1 included to len(a) excluded:
            if a[j] < a[iMin]:
                iMin = j
        swap(a[k], a[iMin])
```

We show this and the other quadratic sorting algorithms in this section not as models to adopt but as examples of the kind of wheel one is likely to reinvent before having studied better ways of doing it. Use them to learn to compare the trade-offs and analyze the performance on simple algorithms where understanding what's happening is not the most difficult issue, as well as to appreciate that coming up with asymptotically better algorithms requires a lot more thought than that.

Exercise 6
When looking for the minimum of m items, every time one of the $m-1$ comparisons fails the best-so-far minimum must be updated. Give a permutation of the numbers from 1 to 7 that, if fed to the Selection sort algorithm, maximizes the number of times that the above-mentioned comparison fails.

2.7 Binary insertion sort

Now suppose that data movement is cheap (e.g. we use pointers, as per footnote 12 on page 20), but comparisons are expensive (e.g. it's string comparison rather than integer comparison). Suppose that, part way through the sorting process, the first k items in our array are neatly in ascending order, and now it is time to consider item $k+1$. A binary search in the initial part of the array can identify where the new item should go, and this search can be done in $\lceil \lg(k) \rceil$ comparisons. Then we can drop the item in place using at most k exchanges. The complete sorting process performs this process for k from 1 to n, and hence the total number of comparisons performed will be

$$\lceil \lg(1) \rceil + \lceil \lg(2) \rceil + \ldots + \lceil \lg(n-1) \rceil$$

which is bounded by

$$\lg(1) + 1 + \lg(2) + 1 + \ldots + lg(n-1) + 1$$

and thus by $\lg((n-1)!) + n = O(\lg(n!)) = O(n \lg n)$. This effectively attains the lower bound for general sorting that we set up earlier, in terms of the number of comparisons. But remember that the algorithm has high (quadratic) data movement costs. Even if a swap were a million times cheaper than a comparison (say), so long as both elementary operations can be bounded by a constant cost then the overall asymptotic cost of this algorithm will be $O(n^2)$.

```
def binaryInsertSort(a):
    """BEHAVIOUR: Run the binary insertion sort algorithm on the integer
    array a, sorting it in place.

    PRECONDITION: array a contains len(a) integer values.

    POSTCONDITION: array a contains the same integer values as before,
    but now they are sorted in ascending order."""

    for k from 1 included to len(a) excluded:
        # ASSERT: the array positions before a[k] are already sorted

        # Use binary partitioning of a[0:k] to figure out where to insert
        # element a[k] within the sorted region;

        ### details left to the reader ###

        # ASSERT: the place of a[k] is i, i.e. between a[i-1] and a[i]

        # Put a[k] in position i. Unless it was already there, this
        # means right-shifting by one every other item in a[i:k].
        if i != k:
            tmp = a[k]
            for j from k - 1 included down to i - 1 excluded:
                a[j + 1] = a[j]
            a[i] = tmp
```

Exercise 7
Code up the details of the binary partitioning portion of the binary insertion sort algorithm.

2.8 Bubble sort

```
def bubbleSort(a):
    """BEHAVIOUR: Run the bubble sort algorithm on the integer
    array a, sorting it in place.

    PRECONDITION: array a contains len(a) integer values.

    POSTCONDITION: array a contains the same integer values as before,
    but now they are sorted in ascending order."""

    repeat:
        # Go through all the elements once, swapping any that are out of order
        didSomeSwapsInThisPass = False
        for k from 0 included to len(a) - 1 excluded:
            if a[k] > a[k + 1]:
                swap(a[k], a[k + 1])
                didSomeSwapsInThisPass = True
    until didSomeSwapsInThisPass == False
```

Another simple sorting method, similar to Insertion sort and very easy to implement, is known as Bubble sort. It consists of repeated passes through the array during which adjacent elements are compared and, if out of order, swapped. The algorithm terminates as soon as a full pass requires no swaps.

Bubble sort is so called because, during successive passes, “light” (i.e. low-valued) elements bubble up towards the “top” (i.e. the cell with the lowest index, or the left end) of the array. Like Insertion sort, this algorithm has quadratic costs in the worst case but it terminates in linear time on input that was already sorted. This is clearly an advantage over Selection sort.

Exercise 8
Prove that Bubble sort will never have to perform more than n passes of the outer loop.

2.9 Mergesort

Given a pair of sub-arrays each of length $n/2$ that have already been sorted, merging their elements into a single sorted array is easy to do in around n steps: just keep taking the lowest element from the sub-array that has it. In a previous course (*Foundations of Computer Science*) you have already seen the sorting algorithm based on this idea: split the input array into two halves and sort them recursively, stopping when the chunks so small that they are already sorted, and then merge the two sorted halves into one sorted array.

```
0  def mergeSort(a):
1      """*** DISCLAIMER: this is purposefully NOT a model of good code
2     (indeed it may hide subtle bugs) but it is a useful starting point
3     for our discussion. ***
4
5      BEHAVIOUR: Run the merge sort algorithm on the integer array a,
6      returning a sorted version of the array as the result. (Note that
7      the array is NOT sorted in place.)
8
9      PRECONDITION: array a contains len(a) integer values.
10
11     POSTCONDITION: a new array is returned that contains the same
12     integer values originally in a, but sorted in ascending order."""
13
14     if len(a) < 2:
15         # ASSERT: a is already sorted, so return it as is
16         return a
17
18     # Split array a into two smaller arrays a1 and a2
19     # and sort these recursively
20     h = int(len(a) / 2)
21     a1 = mergeSort(a[0:h])
22     a2 = mergeSort(a[h:END])
23
24     # Form a new array a3 by merging a1 and a2
25     a3 = new empty array of size len(a)
26     i1 = 0 # index into a1
27     i2 = 0 # index into a2
28     i3 = 0 # index into a3
29     while i1 < len(a1) or i2 < len(a2):
30         # ASSERT: i3 < len(a3)
31         a3[i3] = smallest(a1, i1, a2, i2) # updates i1 or i2 too
32         i3 = i3 + 1
33     # ASSERT: i3 == len(a3)
34     return a3
```

Compared to the other sorting algorithms seen so far, this one hides several subtleties, many to do with memory management issues, which may have escaped you when you studied it in ML:

- Merging two sorted sub-arrays (lines 24–32) is most naturally done by leaving the two input arrays alone and forming the result into a temporary buffer (line 25) as large as the combination of the two inputs. This means that, unlike the other algorithms seen so far, we cannot sort an array in place.

- The recursive calls of the procedure on the sub-arrays (lines 21–22) are easy to write in pseudocode and in several modern high level languages but they may involve additional acrobatics (wrapper functions etc) in languages where the size of the arrays handled by a procedure must be known in advance. The best programmers among you will learn a lot (and maybe find hidden bugs in the pseudocode above) by implementing mergesort in a programming language such as C, without automatic memory management.

- Merging the two sub-arrays is conceptually easy (just consume the "emerging" item from each deck of cards) but coding it up naïvely will fail on boundary cases, as the following exercise highlights.

Exercise 9
Can you spot any problems with the suggestion of replacing the line that assigns to `a3[i3]` with the more explicit and obvious `a3[i3] = min(a1[i1], a2[i2])`? What would be your preferred way of solving such problems? If you prefer to leave that line as it is, how would you implement the procedure `smallest` it calls? What are the trade-offs between your chosen method and any alternatives?

Exercise 10
In one line we return the same array we received from the caller, while in another we return a new array created within the mergesort subroutine. This asymmetry is suspicious. Discuss potential problems.

How do we evaluate the running time of this recursive algorithm? The invocations that don't recurse have constant cost but for the others we must write a so-called *recurrence relation*. If we call $f(n)$ the cost of invoking mergesort on an input array of size n, then we have

$$f(n) = 2f(n/2) + kn,$$

where the first term is the cost of the two recursive calls (lines 21–22) on inputs of size $n/2$ and the second term is the overall cost of the merging phase (lines 24–32), which is linear because a constant-cost sequence of operations is performed for each of the n elements that is extracted from the sub-arrays `a1` and `a2` and placed into the result array `a3`.

To solve the recurrence, i.e. to find an expression for $f(n)$ that doesn't have f on the right-hand side, let's "guess" that exponentials are going to help (since we split the input in two each time, doubling the number of arrays at each step) and let's rewrite the formula[13] with the substitution $n = 2^m$.

[13]This is just an ad-hoc method for solving this particular recurrence, which may not work in other cases. There is a whole theory on how to solve recurrences in chapter 4 of CLRS3.

$$
\begin{aligned}
f(n) &= \underline{f(2^m)} \\
&= 2f(2^m/2) + k2^m \\
&= \underline{2\utilde{f(2^{m-1})} + k2^m} \\
&= 2\utilde{(2f(2^{m-2}) + k2^{m-1})} + k2^m \\
&= 2^2 f(2^{m-2}) + k2^m + k2^m \\
&= 2^2 \underline{\underline{f(2^{m-2})}} + 2 \cdot k2^m \\
&= 2^2 \underline{\underline{(2f(2^{m-3}) + k2^{m-2})}} + 2 \cdot k2^m \\
&= 2^3 f(2^{m-3}) + k2^m + 2 \cdot k2^m \\
&= 2^3 f(2^{m-3}) + 3 \cdot k2^m \\
&= \ldots \\
&= 2^m f(2^{m-m}) + m \cdot k2^m \\
&= f(1) \cdot 2^m + k \cdot m2^m \\
&= k_0 \cdot 2^m + k \cdot m2^m \\
&= k_0 n + kn \lg n.
\end{aligned}
$$

We just proved that $f(n) = k_0 n + kn \lg n$ or, in other words, that $f(n) = O(n \lg n)$. Thus Mergesort is the first sorting algorithm we discuss in this course whose running time is better than quadratic. Much more than that, in fact: mergesort guarantees the *optimal* cost of $n \lg n$, is relatively simple and has low time overheads. Its main disadvantage is that it requires extra space to hold the partially merged results. The implementation is trivial if one has another empty n-cell array available; but experienced programmers can get away with just $n/2$. Theoretical computer scientists have been known to get away with just *constant* space overhead[14].

Exercise 11
Never mind the theoretical computer scientists, but how do you mergesort in $n/2$ space?

An alternative is to run the algorithm bottom-up, doing away with the recursion. Group elements two by two and sort (by merging) each pair. Then group the sorted pairs two by two, forming (by merging) sorted quadruples. Then group those two by two, merging them into sorted groups of 8, and so on until the last pass in which you merge two large sorted groups. Unfortunately, even though it eliminates the recursion, this variant still requires $O(n)$ additional temporary storage, because to merge two groups

[14]Cfr. Jyrki Katajainen, Tomi Pasanen, Jukka Teuhola. “Practical in-place mergesort”. *Nordic Journal of Computing* 3:27-–40, 1996. Note that real programmers and theoretical computer scientists tend to assign different semantics to the word “practical”.

of k elements each into a $2k$ sorted group you still need an auxiliary area of k cells (move the first half into the auxiliary area, then repeatedly take the smallest element from either the second half or the auxiliary area and put it in place).

Exercise 12
Justify that the merging procedure just described will not overwrite any of the elements in the second half.

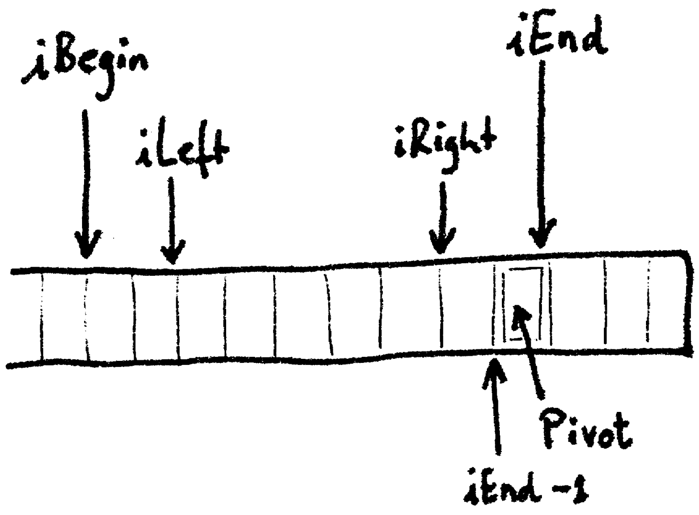
Exercise 13
Write pseudocode for the bottom-up mergesort.

2.10 Quicksort

Quicksort is the most elaborate of the sorting algorithms you have already seen, from a functional programming viewpoint, in the *Foundations* course. The main thing for you to understand and appreciate in this second pass is how cleverly it manages to sort the array in place, splitting it into a “lower” and a “higher” part without requiring additional storage. You should also have a closer look at what happens in presence of duplicates.

The algorithm is relatively easy to explain and, when properly implemented and applied to non-malicious input data, the method can fully live up to its name. However Quicksort is somewhat temperamental. It is remarkably easy to write a program based on the Quicksort idea that is wrong in various subtle cases (e.g. if all the items in the input list are identical) and, although in almost all cases Quicksort turns in a time proportional to $n \lg n$ (with a quite small constant of proportionality), for worst case input data it can be as slow as n^2. There are also several small variants. It is strongly recommended that you study the description of Quicksort in your favourite textbook and that you look carefully at the way in which code can be written to avoid degenerate cases leading to accesses off the end of arrays etc.

The core idea of Quicksort, as you will recall, is to select some value from the array and use that as a “pivot” to split the other values into two classes: those smaller and those larger than the pivot. What happens when applying this idea to an array? A selection procedure partitions the values so that the lower portion of the array holds values not exceeding that of the pivot, while the upper part holds only larger values. This selection can be performed in place by scanning in from the two ends of the array, exchanging values as necessary. Then the pivot is placed where it belongs, so that the array contains a first region (still unsorted) with the low values, then the pivot, then a third region (still unsorted) with the high values. For an n element array it takes about n comparisons and data exchanges to partition the array. Quicksort is then called recursively to deal with the low and high parts of the data, and the result is obviously that the entire array ends up perfectly sorted.



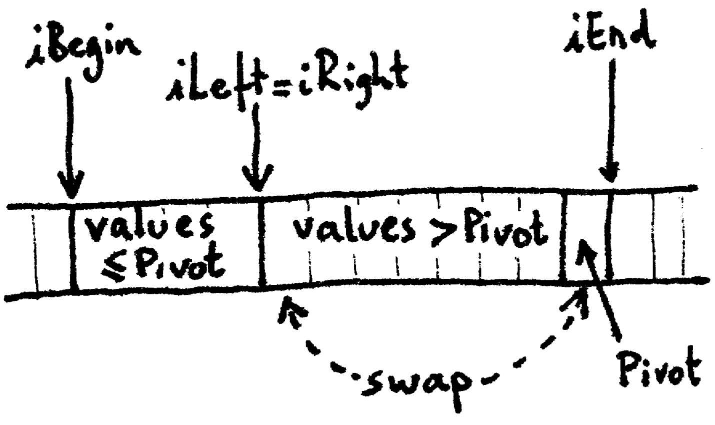
Let's have a closer look at implementing Quicksort. Remember we scan the array (or sub-array) from both ends to partition it into three regions. Assume you must sort the sub-array `a[iBegin:iEnd]`, which contains the cells from `a[iBegin]` (included) to `a[iEnd]` (excluded)[15]. We use two auxiliary indices `iLeft` and `iRight`. We arbitrarily pick the last element in the range as the pivot: `Pivot = A[iEnd - 1]`. Then `iLeft` starts at `iBegin` and moves right, while `iRight` starts at `iEnd - 1` and moves left. All along, we maintain the following invariants:

- `iLeft` $\leq$ `iRight`
- `a[iBegin:iLeft]` only has elements $\leq$ `Pivot`
- `a[iRight:iEnd - 1]` only has elements $>$ `Pivot`

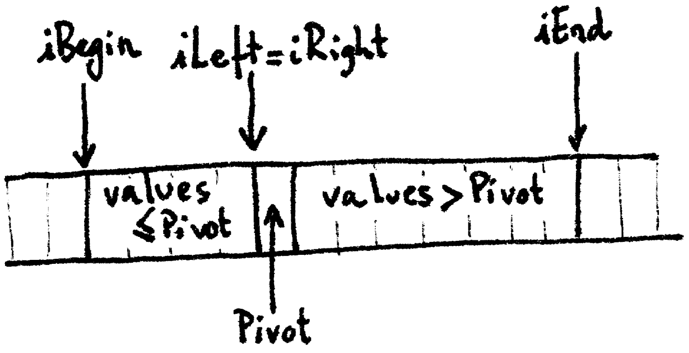
So long as `iLeft` and `iRight` have not met, we move `iLeft` as far right as possible and `iRight` as far left as possible without violating the invariants. Once they stop, if they haven't met, it means that `A[iLeft]` $>$ `Pivot` (otherwise we could move `iLeft` further right) and that `A[iRight - 1]` $\leq$ `Pivot` (thanks to the symmetrical argument[16]). So we swap these two elements pointed to by `iLeft` and `iRight - 1`. Then we repeat the process, again pushing `iLeft` and `iRight` as far towards each other as possible, swapping array elements when the indices stop and continuing until they touch.

[15]See the boxed aside on page 12.

[16]Observe that, in order to consider `iRight` the symmetrical mirror-image of `iLeft`, we must consider `iRight` to be pointing, conceptually, at the cell to its *left*, hence the `- 1`.



At that point, when `iLeft` = `iRight`, we put the pivot in its rightful place between the two regions we created, by swapping `A[iRight]` and `[iEnd - 1]`.



We then recursively run Quicksort on the two smaller sub-arrays `a[iBegin:iLeft]` and `a[iRight + 1:iEnd]`.

Now let's look at performance. Consider first the ideal case, where each selection manages to split the array into two equal parts. Then the total cost of Quicksort satisfies $f(n) = 2f(n/2) + kn$, and hence grows as $O(n \lg n)$ as we proved in section 2.9. But, in the worst case, the array might be split very unevenly—perhaps at each step only a couple of items, or even none, would end up less than the selected pivot. In that case the recursion (now $f(n) = f(n-1) + kn$) will go around n deep, and therefore the total worst-case costs will grow to be proportional to n^2.

One way of estimating the average cost of Quicksort is to suppose that the pivot could equally probably have been any one of the items in the data. It is even reasonable to use a random number generator to select an arbitrary item for use as a pivot to ensure this.

Exercise 14
Can picking the pivot at random *really* make any difference to the expected performance? How will it affect the average case? The worst case? Discuss.

Then it is easy to set up a recurrence formula that will be satisfied by the average cost:

$$f(n) = kn + \frac{1}{n}\sum_{i=1}^{n}\Big(f(i-1) + f(n-i)\Big)$$

where the sum adds up the expected costs corresponding to all the (equally probable) ways in which the partitioning might happen. After some amount of playing with this equation, it can be established that the average cost for Quicksort is $\Theta(n \lg n)$.

Quicksort provides a sharp illustration of what can be a problem when selecting an algorithm to incorporate in an application. Although its average performance (for random data) is good, it does have a quite unsatisfactory (albeit uncommon) worst case. It should therefore not be used in applications where the worst-case costs could have safety implications. The decision about whether to use Quicksort for good average speed or a slightly slower but guaranteed $O(n \lg n)$ method can be a delicate one.

There are a great number of small variants on the Quicksort scheme and the best way to understand them for an aspiring computer scientist is to code them up, sprinkle them with diagnostic print statements and run them on examples. There are good reasons for using the median[17] of the mid point and two others as the pivot at each stage, and for using recursion only on partitions larger than a preset threshold. When the region is small enough, Insertion sort may be used instead of recursing down. A less intuitive but probably more economical arrangement is for Quicksort just to *return* (without sorting) when the region is smaller than a threshold; then one runs Insertion sort over the messy array produced by the truncated Quicksort.

Exercise 15
Justify why running Insertion sort over the messy array produced by the truncated Quicksort might not be as stupid as it may sound at first. How should the threshold be chosen?

2.11 Median and order statistics using Quicksort

The **median** of a collection of values is the one such that as many items are smaller than that value as are larger. In practice, when we look for algorithms to find a median, it is wise to expand to more general **order statistics**: we shall therefore look for the item that ranks at some parametric position k in the data. If we have n items, the median corresponds to taking the special case $k = n/2$, while $k = 1$ and $k = n$ correspond to looking for minimum and maximum values.

One obvious way of solving this problem is to sort that data: then the item with rank k is trivial to read off. But that costs $O(n \lg n)$ for the sorting.

Two variants on Quicksort are available that solve the problem. One, like Quicksort itself, has linear cost in the average case, but has a quadratic worst-case cost. The other is more elaborate to code and has a much higher constant of proportionality, but guarantees

[17] Cfr. section 2.11.

linear cost. In cases where guaranteed worst-case performance is essential the second method might in theory be useful; in practice, however, it is so complicated and slow that it is seldom implemented[18].

Exercise 16
What is the smallest number of pairwise comparisons you need to perform to find the smallest of n items?

Exercise 17
(More challenging.) And to find the *second* smallest?

The simpler scheme selects a pivot and partitions as for Quicksort, at linear cost. Now suppose that the partition splits the array into two parts, the first having size p, and imagine that we are looking for the item with rank k in the whole array. If $k < p$ then we just continue be looking for the rank-k item in the lower partition. Otherwise we look for the item with rank $k-p$ in the upper one. The cost recurrence for this method (assuming, unreasonably optimistically, that each selection stage divides out values neatly into two even sets) is $f(n) = f(n/2) + kn$, whose solution exhibits linear growth as we shall now prove. Setting $n = 2^m$ as we previously did (and for the same reason), we obtain

$$\begin{aligned}
f(n) &= f(2^m) \\
&= f(2^m/2) + k2^m \\
&= \underline{f(2^{m-1})} + k2^m \\
&= \underline{f(2^{m-2}) + k2^{m-1}} + k2^m \\
&= f(2^{m-3}) + k2^{m-2} + k2^{m-1} + k2^m \\
&= \ldots \\
&= f(2^{m-m}) + k2^{m-(m-1)} + \ldots + k2^{m-2} + k2^{m-1} + k2^m \\
&= f(2^0) + k(2^1 + 2^2 + 2^3 + \ldots 2^m) \\
&= f(1) + 2k(2^m - 1) \\
&= k_0 + k_1 2^m \\
&= k_0 + k_1 n
\end{aligned}$$

which is indeed $O(n)$, QED.

As with Quicksort itself, and using essentially the same arguments, it can be shown that this best-case linear cost also applies to the average case; but, equally, that the worst-case, though rare, has quadratic cost.

[18]We won't recommend or describe the overly elaborate (though, to some, perversely fascinating) guaranteed-linear-cost method here. It's explained in your textbook if you're curious: see CLRS3, 9.3.

2.12 Heapsort

Despite the good average behaviour of Quicksort, there are circumstances where one might want a sorting method that is guaranteed to run in time $O(n \lg n)$ whatever the input[19], even if such a guarantee may cost some increase in the constant of proportionality.

Heapsort is such a method, and is described here not only because it is a reasonable sorting scheme, but because the data structure it uses (called a **heap**, a term that is also used, but with a totally different meaning, in the context of free-storage management) has many other applications.

Consider an array that has values stored in all its cells, but with the constraint (known as "the heap property") that the value at position k is greater than (or equal to) those at positions[20] $2k+1$ and $2k+2$. The data in such an array is referred to as a heap. The heap is isomorphic to a binary tree in which each node has a value at least as large as those of its children—which, as one can easily prove, means it is also the largest value of all the nodes in the subtree of which it is root. The root of the heap (and of the equivalent tree) is the item at location 0 and, by what we just said, it is the largest value in the heap.

The data structure we just described, which we'll use in heapsort, is also known as a max-heap. You may also encounter the dual arrangement, appropriately known as min-heap, where the value in each node is at least as *small* as the values in its children; there, the root of the heap is the *smallest* element (see section 4.7).

Note that any binary tree that represents a binary heap must have a particular "shape", known as **almost full binary tree**: every level of the tree, except possibly the last, must be full, i.e. it must have the maximum possible number of nodes; and the last level must either be full or have empty spaces only at its right end. This constraint on the shape comes from the isomorphism with the array representation: a binary tree with any other shape would map back to an array with "holes" in it.

Exercise 18
What are the minimum and maximum number of elements in a heap of height h?

The Heapsort algorithm consists of two phases. The first phase takes an array full of unsorted data and rearranges it in place so that the data forms a heap. Amazingly, this can be done in linear time, as we shall prove shortly. The second phase takes the top (leftmost) item from the heap (which, as we saw, was the largest value present) and swaps it to the last position in the array, which is where that value needs to be in the final sorted output. It then has to rearrange the remaining data to be a heap with one fewer element. Repeating this step will leave the full set of data in order in the array. Each heap reconstruction step has a cost bounded by the logarithm of the amount of data left, and thus the total cost of Heapsort ends up being bounded by $O(n \lg n)$, which is optimal.

The auxiliary function `heapify` (lines 22–36) takes a (sub)array that is almost a heap and turns it into a heap. The assumption is that the two (possibly empty) subtrees of the

[19] Mergesort does; but, as we said, it can't sort the array in place.

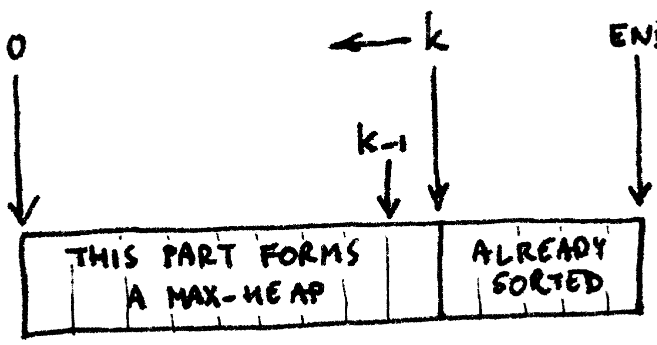
[20] Supposing that those two locations are still within the bounds of the array, and assuming that indices start at 0.

root are already proper heaps, but that the root itself may violate the max-heap property, i.e. it might be smaller than one or both of its children. The `heapify` function works by swapping the root with its largest child, thereby fixing the heap property for that position. What about the two subtrees? The one not affected by the swap was already a heap to start with, and after the swap the root of the subtree is certainly $\leq$ than its parent, so all is fine there. For the other subtree, all that's left to do is ensure that the old root, in its new position further down, doesn't violate the heap property there. This can be done recursively. The original root therefore sinks down step by step to the position it should rightfully occupy, in no more calls than there are levels in the tree. Since the tree is "almost full", its depth is the logarithm of the number of its nodes, so `heapify` is $O(\lg n)$.

```
 0 def heapSort(a):
 1     """BEHAVIOUR: Run the heapsort algorithm on the integer
 2     array a, sorting it in place.
 3
 4     PRECONDITION: array a contains len(a) integer values.
 5
 6     POSTCONDITION: array a contains the same integer values as before,
 7     but now they are sorted in ascending order."""
 8
 9     # First, turn the whole array into a heap
10     for k from floor(END/2) excluded down to 0 included: # nodes with children
11         heapify(a, END, k)
12
13     # Second, repeatedly extract the max, building the sorted array R-to-L
14     for k from END included down to 1 excluded:
15         # ASSERT: a[0:k] is a max-heap
16         # ASSERT: a[k:END] is sorted in ascending order
17         # ASSERT: every value in a[0:k] is <= than every value in a[k:END]
18         swap(a[0], a[k - 1])
19         heapify(a, k - 1, 0)
20
21
22 def heapify(a, iEnd, iRoot):
23     """BEHAVIOUR: Within array a[0:iEnd], consider the subtree rooted
24     at a[iRoot] and make it into a max-heap if it isn't one already.
25
26     PRECONDITIONS: 0 <= iRoot < iEnd <= END. The children of
27     a[iRoot], if any, are already roots of max-heaps.
28
29     POSTCONDITION: a[iRoot] is root of a max-heap."""
30
31     if a[iRoot] satisfies the max-heap property:
32         return
33     else:
34         let j point to the largest among the existing children of a[iRoot]
35         swap(a[iRoot], a[j])
36         heapify(a, iEnd, j)
```

The first phase of the main `heapSort` function (lines 9–11) starts from the bottom of the tree (rightmost end of the array) and walks up towards the root, considering each node as the root of a potential sub-heap and rearranging it to be a heap. In fact, nodes with no children can't possibly violate the heap property and therefore are automatically heaps; so we don't even need to process them—that's why we start from the midpoint `floor(END/2)` rather than from the end. By proceeding right-to-left we guarantee that any children of the node we are currently examining are already roots of properly formed heaps, thereby matching the precondition of `heapify`, which we may therefore use. It is then trivial to put an $O(n \lg n)$ bound on this phase—although, as we shall see, it is not tight.

In the second phase (lines 13–19), the array is split into two distinct parts: `a[0:k]` is a heap, while `a[k:END]` is the "tail" portion of the sorted array. The rightmost part starts empty and grows by one element at each pass until it occupies the whole array. During each pass of the loop in lines 13–19 we extract the maximum element from the root of the heap, `a[0]`, reform the heap and then place the extracted maximum in the empty space left by the last element, `a[k]`, which conveniently is just where it should go[21]. To retransform `a[0:k - 1]` into a heap after placing `a[k - 1]` in position `a[0]` we may call `heapify`, since the two subtrees of the root `a[0]` must still be heaps given that all that changed was the root and we started from a proper heap. For this second phase, too, it is trivial to establish an $O(n \lg n)$ bound.



Now, what was that story about the first phase actually taking *less* than $O(n \lg n)$? Well, it's true that all heaps are at most $O(\lg n)$ tall, but many of them are much shorter because most of the nodes of the tree are found in the lower levels[22], where they can only be roots of short trees. So let's redo the budget more accurately.

level	num nodes in level	height of tree	max cost of heapify
0	1	h	kh
1	2	$h-1$	$k(h-1)$
2	4	$h-2$	$k(h-2)$
...			
j	2^j	$h-j$	$k(h-j)$
...			
h	2^h	0	0

[21]Because all the items in the right part are $\geq$ than the ones still in the heap, since each of them was the maximum at the time of extraction.

[22]Indeed, in a full binary tree, each level contains one more node than the *whole* tree above it.

The cost for level j is the "max cost of heapify" for a node in that level, i.e. $k(h-j)$, times the number of nodes in that level, 2^j. The cost for the whole tree, as a function of the number of levels, is simply the sum of the costs of all levels:

$$\begin{aligned}
C(h) &= \sum_{j=0}^{h} 2^j \cdot k(h-j) \\
&= k\frac{2^h}{2^h}\sum_{j=0}^{h} 2^j(h-j) \\
&= k2^h \sum_{j=0}^{h} 2^{j-h}(h-j) \\
&\quad \ldots \text{let } l = h-j \ldots \\
&= k2^h \sum_{l=0}^{h} l2^{-l} \\
&= k2^h \sum_{l=0}^{h} l\left(\frac{1}{2}\right)^l
\end{aligned}$$

The interesting thing is that this last summation, even though it is a monotonically growing function of h, is in fact bounded by a constant, because the corresponding series converges to a finite value if the absolute value of the base of the exponent (here $\frac{1}{2}$) is less than 1:

$$|x| < 1 \quad \Rightarrow \quad \sum_{i=0}^{\infty} ix^i = \frac{x}{(1-x)^2}.$$

This means that the cost $C(h)$ grows like $O(2^h)$ and, if we instead express this in terms of the number of nodes in the tree, $C(n) = O(n)$ and not $O(n \lg n)$, QED.

Heapsort therefore offers at least two significant advantages over other sorting algorithms: it offers an asymptotically optimal worst-case complexity of $O(n \lg n)$ *and* it sorts the array in place. Despite this, on non-pathological data it is still usually beaten by the amazing Quicksort.

2.13 Stability of sorting methods

Data to be sorted often consists of records made of key and payload; the key is what the ordering is based upon, while the payload is some additional data that is just carried around in the rearranging process. In some applications one can have keys that should be considered equal, and then a simple specification of sorting might not indicate the order in which the corresponding records should end up in the output list. "Stable" sorting demands that, in such cases, the order of items in the input be preserved in the output. Some otherwise desirable sorting algorithms are not stable, and this can weigh against them.

If stability is required, despite not being offered by the chosen sorting algorithm, the following technique may be used. Extend the records with an extra field that stores their original position, and extend the ordering predicate used while sorting to use comparisons on this field to break ties. Then, any arbitrary sorting method will rearrange the data in a stable way, although this clearly increases space and time overheads a little (but by no more than a linear amount).

Exercise 19
For each of the sorting algorithms seen in this course, establish whether it is stable or not.

2.14 Faster sorting

If the condition that sorting must be based on pair-wise comparisons is dropped it may sometimes be possible to do better than $O(n \lg n)$ in terms of asymptotic costs. In this section we consider three algorithms that, under appropriate assumptions, sort in linear time. Two particular cases are common enough to be of at least occasional importance: sorting integer keys from a fixed range (*counting sort*) and sorting real keys uniformly distributed over a fixed range (*bucket sort*). Another interesting algorithm is *radix sort*, used to sort integer numerals of fixed length[23], which was originally used to sort punched cards mechanically.

2.14.1 Counting sort

Assume that the keys to be sorted are integers that live in a known range, and that the range is fixed regardless of the number of values to be processed. If the number of input items grows beyond the cardinality of the range, there will necessarily be duplicates in the input. If no data is involved at all beyond the integers, one can set up an array whose size is determined by the range of integers that can appear (not by the amount of data to be sorted) and initialize it to all 0s. Then, for each item in the input data, w say, the value at position w in the array is incremented. At the end, the array contains information about how many instances of each value were present in the input, and it is easy to create a sorted output list with the correct values in it. The costs are obviously linear.

If additional satellite data beyond the keys is present (as will usually happen) then, once the counts have been collected, a second scan through the input data can use the counts to indicate the exact position, in the output array, to which each data item should be moved. This does not compromise the overall linear cost.

During the second pass, the fact of scanning the items in the order in which they appear in the input array ensures that items with the same key maintain their relative order in the output. Thus counting sort is not only fast but also stable.

[23] Or, more generally, any keys (including character strings) that can be mapped to fixed-length numerals in an arbitrary base.

Exercise 20
Give detailed pseudocode for the counting sort algorithm (particularly the second phase), ensuring that the overall cost stays linear. Do you need to perform any kind of precomputation of auxiliary values?

2.14.2 Bucket sort

Assume the input data is guaranteed to be uniformly distributed over some known range (for instance it might be real numbers in the range 0.0 to 1.0). Then a numeric calculation on the key can predict with reasonable accuracy where a value must be placed in the output. If the output array is treated somewhat like a hash table (cfr. section 4.6), and this prediction is used to insert items in it, then, apart from some local effects of clustering, that data has been sorted.

To sort n keys uniformly distributed between 0.0 and 1.0, create an array of n linked lists and insert each key k to the list at position $\lfloor k \cdot n \rfloor$. This phase has linear cost. (We expect each list to be one key long on average, though some may be slightly longer and some may be empty.) In the next phase, for each entry in the array, sort the corresponding list with insertsort if it is longer than one element, then output it.

Insertsort, as we know, has a quadratic worst-case running time. How does this affect the running time of bucket sort? If we could assume that the lists are never longer than a constant k, it would be trivial to show that the second pass too has linear costs in the worst case. However we can't, so we need to resort to a rather more elaborate argument. But it is possible to prove that, under the assumption that the input values are uniformly distributed, the *average-case* (but not worst-case) overall running time of bucket sort is linear in n.

2.14.3 Radix sort

Historically, radix sort was first described in the context of sorting integers encoded on punched cards, where each column of a card represented a digit by having a hole punched in the corresponding row. A mechanical device could be set to examine a particular column and distribute the cards of a deck into bins, one per digit, according to the digit in that column. Radix sort used this "primitive" to sort the whole deck.

The obvious way to proceed is perhaps to sort on the most significant digit, then recursively for each bin on the next significant digit, then on the next, all the way down to the least significant digit. But this would require a great deal of temporary "desk space" to hold the partial mini-decks still to be processed without mixing them up.

Radix sort instead proceeds, counter-intuitively, from the least significant digit upwards. First, the deck is sorted into $b = 10$ bins based on the least significant digit. Then the bins are juxtaposed, in order, to reform a full deck, and this is then sorted according to the next digit up. But the per-digit sorting method used is chosen to be stable, so that cards that have the same second digit still maintain their relative order, which was induced by the first (least significant) digit. The procedure is repeated going upwards towards the most significant digits. Before starting pass i, the digits in positions 0 to $i - 1$ have already been sorted. During pass i, the deck is sorted on the digit in position

i, but all the cards with the same i digit remain in the relative order determined by the remaining least significant digits. The result is that, once the deck has been sorted on the most significant digit, it is fully sorted. The number of passes is equal to the number of digits (d) in the numerals being sorted and the cost of each pass can be made linear by using counting sort.

Exercise 21
Why couldn't we simply use counting sort in the first place, since the keys are integers in a known range?

Note that counting sort does not sort in place; therefore, if that is the stable sort used by radix sort, neither does radix sort. This, as well as the constants hidden by the big-O notation, must be taken into account when deciding whether radix sort is advantageous, in a particular application, compared to an in-place algorithm like quicksort.

Chapter 3

Algorithm design

Chapter contents

Strategies for algorithm design: dynamic programming, divide and conquer, greedy algorithms and other useful paradigms.
Expected coverage: about 3 lectures.

There exists no general recipe for designing an algorithm that will solve a given problem—never mind designing an optimally efficient one. There are, however, a number of generally useful strategies, some of which we have seen in action in the sorting algorithms discussed so far. Each design paradigm works well in at least some cases; with the flair you acquire from experience you may be able to choose an appropriate one for your problem.

In this chapter we shall first study a powerful technique called dynamic programming. After that we shall name and describe several other paradigms, some of which you will recognize from algorithms we already discussed, while others you will encounter later in the course or in *Algorithms II*. None of these are guaranteed to succeed in all cases but they are all instructive ways of approaching algorithm design.

3.1 Dynamic programming

Sometimes it makes sense to work up towards the solution to a problem by building up a table of solutions to smaller versions of the problem. For historical reasons, this process is known as "dynamic programming", but the use of the term "programming" in this context comes from operations research rather than computing and has nothing to do with our usual semantics of writing instructions for a machine: it originally meant something like "finding a plan of action".

Dynamic programming is related to the strategy of "divide and conquer" (section 3.3.3, q.v., but already seen in mergesort and quicksort) in that it breaks up the original problem recursively into smaller problems that are easier to solve. But the essential difference is that here the subproblems may overlap. Applying the divide and conquer approach in this setting would inefficiently solve the same subproblems again and again along different branches of the recursion tree. Dynamic programming, instead, is based on computing the solution to each subproblem only once: either by remembering the intermediate solutions and reusing them as appropriate instead of recomputing them; or, alternatively, by deriving out the intermediate solutions in an appropriate bottom-up order that makes it unnecessary to recompute old ones again and again.

This method has applications in various tasks related to combinatorial search. It is difficult to describe it both accurately and understandably without having demonstrated it in action so let's use examples.

An instructive preliminary exercise is the computation of Fibonacci numbers:

Exercise 22
Leaving aside for brevity Fibonacci's original 1202 problem on the sexual activities of a pair of rabbits, the Fibonacci sequence may be more abstractly defined as follows:

$$\begin{cases} F_0 = 1 \\ F_1 = 1 \\ F_n = F_{n-2} + F_{n-1} \quad \text{for } n \geq 2 \end{cases}$$

(This yields $1, 1, 2, 3, 5, 8, 13, 21, \ldots$)
In a couple of lines in your favourite programming language, write a *recursive* program to compute F_n given n, using the definition above. And now, finally, the question: how many function calls will your recursive program perform to compute F_{10}, F_{20} and F_{30}? First, guess; then instrument your program to tell you the actual answer.

Where's the dynamic programming in this? We'll come back to that, but note the contrast between the exponential number of recursive calls, where smaller Fibonacci numbers are recomputed again and again, and the trivial and much more efficient iterative solution of computing the Fibonacci numbers bottom-up.

A more significant example is the problem of finding the best order in which to perform matrix chain multiplication. If A is a $p \times q$ matrix and B is a $q \times r$ matrix, the product $C = AB$ is a $p \times r$ matrix whose elements are defined by

$$c_{i,j} = \sum_{k=0}^{q-1} a_{i,k} \cdot b_{k,j}.$$

The product matrix can thus be computed using $p \cdot q \cdot r$ scalar multiplications (q multiplications[1] for each of the $p \cdot r$ elements). If we wish to compute the product $A_1 A_2 A_3 A_4 A_5 A_6$ of 6 matrices, we have a wide choice of the order in which we do the matrix multiplications.

[1]As well as $q - 1$ sums.

Exercise 23
Prove (an example is sufficient) that the order in which the multiplications are performed may dramatically affect the total number of scalar multiplications—despite the fact that, since matrix multiplication is associative, the final matrix stays the same.

Suppose the dimensions of A_1, A_2, A_3, A_4, A_5 and A_6 are respectively 30×35, 35×15, 15×5, 5×10, 10×20 and 20×25. Clearly, adjacent dimensions must be equal or matrix multiplication would be impossible; therefore this set of dimensions can be specified by the vector $(p_0, p_1, \ldots, p_6) = (30, 35, 15, 5, 10, 20, 25)$. The problem is to find an order of multiplications that minimizes the number of scalar multiplications needed.

Observe that, with this notation, the dimensions of matrix A_i are $p_{i-1} \times p_i$. Suppose that i, j, k are integer indices, that $A_{i..j}$ stands for the product $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$ and that $m(i, j)$ is the minimum number of scalar multiplications to compute the product $A_{i..j}$. Then m can be defined as a recursive function:

$$m(i,j) = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{k \in [i,j)} \{m(i,k) + m(k+1,j) + p_{i-1}p_kp_j\} & \text{if } i \neq j \end{cases}$$

With a sequence $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$ of $j - i + 1$ matrices, $j - i$ matrix multiplications are required. The last such matrix multiplication to be performed will, for some k between i and j, combine a left-hand cumulative matrix $A_{i..k}$ and a right-hand one $A_{k+1..j}$. The cost of that matrix multiplication will be $p_{i-1} \cdot p_k \cdot p_j$. The expression above for the function $m(i, j)$ is obtained by noting that one of the possible values of k must be the one yielding the minimum total cost and that, for that value, the two contributions from the cumulative matrices must also each be of minimum cost.

For the above numerical problem the answer is $m(1, 6) = 15125$ scalar multiplications. A naïve recursive implementation of this function will compute $m(i, j)$ in time exponential in the number of matrices to be multiplied[2], but the value can be obtained more efficiently by computing and remembering the values of $m(i, j)$ in a systematic order so that, whenever $m(i, k)$ or $m(k + 1, j)$ is required, the values are already known. An alternative approach, as hinted at above, is to modify the simple-minded recursive definition of the $m()$ function so that it checks whether $m(i, j)$ has already been computed. If so, it immediately returns with the previously computed result, otherwise it computes and saves the result in a table before returning. This technique is known as *memoization* (this not a typo for "memorization"—it comes from "jotting down a memo" rather than from "memorizing"). For the previous example, the naïve implementation computes $m(1, 6) = 15125$ in 243 invocations of $m()$, while a memoized version yields the same result in only 71 invocations (try it).

Going back to general principles, dynamic programming tends to be useful against problems with the following features:

[2]Do not confuse the act of computing m (minimum number of multiplications, and related choice of which matrices to multiply in which order) and the act of computing the matrix product itself. The former is the computation of a *strategy* for performing the latter cheaply. But *either* of these acts can be computationally expensive.

1. There exist many choices, each with its own "score" which must be minimized or maximized (optimization problem). *In the example above, each parenthesization of the matrix expression is a choice; its score is the number of scalar multiplications it involves.*

2. The number of choices is exponential in the size of the problem, so brute force is generally not applicable. *The number of choices here is the number of possible binary trees that have the given sequence of matrices (in order) as leaves.*

3. The structure of the optimal solution is such that it is composed of optimal solutions to smaller problems. *The optimal solution ultimately consists of multiplying together two cumulative matrices; these two matrices must themselves be optimal solutions for the corresponding subproblems because, if they weren't, we could substitute the optimal sub-solutions and get a better overall result.*

4. There is overlap: in general, the optimal solution to a sub-problem is required to solve several higher-level problems, not just one. *The optimal sub-solution $m(2, 4)$ is needed in order to compute $m(2, 5)$ but also to compute $m(1, 4)$.*

Because of the fourth property of the problem, a straightforward recursive divide-and-conquer approach will end up recomputing the common sub-solutions many times, unless it is turned into dynamic programming through memoization.

The non-recursive, bottom-up approach to dynamic programming consists instead of building up the optimal solutions incrementally, starting from the smallest ones and going up gradually.

To solve a problem with dynamic programming one must define a suitable sub-problem structure so as to be able to write a kind of recurrence relation that describes the optimal solution to a sub-problem in terms of optimal solutions to smaller sub-problems.

3.2 Greedy algorithms

Many algorithms involve some sort of optimization. The idea of "greed" is to start by performing whatever operation contributes as much as any single step can towards the final goal. The next step will then be the best step that can be taken from the new position and so on. The procedures for finding minimal spanning sub-trees, described in the *Algorithms II* course, are examples of how greed can sometimes lead to good results. Other times, though, greed can get you stuck in a local maximum—so it's safest to rely on a correctness proof before blindly using a greedy algorithm.

Most problems that can be solved by a greedy algorithm can also be solved by dynamic programming: both strategies exploit the optimal structure of the sub-problems. However the greedy strategy, when it applies, is a much cheaper way of reaching the overall optimal solution. Let's start with an example.

A university has a sports hall that can be used for a variety of activities, but only one at a time. The various sports societies of the University propose bookings for the hall, in the form of (start, finish) time segments, and the management wishes to maximize the number of proposals it can satisfy. More formally, we are given a set $S = \{a_1, a_2, \ldots, a_n\}$ of n activities, each of the form $a_i = (s_i, f_i)$, sorted for convenience in order of finishing time ($f_1 \leq f_2 \leq \ldots \leq f_n$), and we wish to find a maximum-size subset of S in which no two activities overlap.

The number of subsets of S is 2^n, since each element of S can either be or not be in the subset. Examining each subset to establish whether its activities are compatible and then selecting a maximum-cardinality subset among those that pass this filter has exponential cost and is therefore infeasible other than for very small n.

Using the dynamic programming approach, we note that assuming activity a_i to be a member of the optimal subset identifies two further subsets: excluding the activities that overlap with a_i, we have a first set S_{iL} with activities that complete before a_i starts, and a second set S_{iR} with activities that start after a_i finishes. If the optimal solution does include a_i, then it must consist of a_i together with the optimal solution for S_{iL} and the optimal solution for S_{iR}. (It's easy to show that, if it didn't, a "more optimal" solution could be built by including them instead. This is the usual "cut and paste" argument used in dynamic programming.) If, conversely, the optimal solution doesn't include a_i, then by the same argument the union of $\{a_i\}$, the optimal solution for S_{iL} and the optimal solution for S_{iR} is still the best solution we can produce that includes a_i.

We don't know whether any particular a_i will be included in the optimal solution, but we know that at least one of them will, so we span all possible values of i and pick the one yielding the subset of greatest cardinality. If we indicate with $opt(X)$ the cardinality of the maximum subset of non-overlapping activities in set X, then we can define the function recursively as:

$$opt(\{\}) = 0$$

$$opt(S) = \max_{0 \leq i \leq n} (opt(S_{iL}) + 1 + opt(S_{iR}))$$

with S_{iL} and S_{iR} defined as above. We can then either use recursion[3] and memoize, or use iteration and compute the subsets bottom-up, thus in either case bringing the costs down from exponential to polynomial in n.

This is a great improvement, but the greedy strategy takes a more daring gamble: instead of keeping all options open, and deciding on the maximum only after having tried all possibilities and having unwound all the levels of the memoized recursion, the greedy strategy says "hey, I bet I know which of these activities is in the subset"; then, assuming the guess is correct, the optimal solution must consist of *that* activity plus the optimal

[3]The notation I used above is simplified for clarity of explanation, but incomplete. To recurse down into subproblems we need a more complicated notation such as the one used in the CLRS3 textbook. The simplified notation used above is OK at the top level of the problem but is insufficiently powerful to express that, lower down, I want (say) the activities that complete before $a_{i'}$ starts *but* within those that start after the higher-level a_i finishes. The full notation in CLRS3 defines S_{ij} as the set of activities that start after the end of a_i and finish before the start of a_j, thus

$$S_{ij} = \{a_k \in S \quad : \quad f_i \leq s_k \leq f_k \leq s_j\}.$$

And then you'd need some extra notational acrobatics (not quite specified in the book) to express the top-level starting set S, which can't be generated from the above definition as S_{ij} by assigning in-range values to i and j. The cardinality of the best possible overall solution containing activity a_i will still be $opt(S_{iL}) + 1 + opt(S_{iR})$ but we will now also be able to speak of the best possible solution containing activity a_k *for the subset of activities that start after the end of* a_i *and finish before the start of* a_j. This additional expressive power is required to recurse into the subproblems.

solution to the remaining ones that don't overlap with it; and so forth. Very similar in structure to dynamic programming but much more direct because we immediately commit to one choice at every stage, discarding all the others a priori. (Of course this relies on being able to make the correct choice!)

In this example the greedy strategy is to pick the activity that finishes first, based on the intuition that it's the one that leaves most of the rest of the timeline free for the allocation of other activities. Now, to be able to use the greedy strategy safely on this problem, we must prove that this choice is indeed optimal; in other words, that the $a_i \in S$ with the smallest f_i is included in an optimal solution for S. (That would be a_1, by the way, since we conveniently said that activities were numbered in order of increasing finishing time.)

Proof by contradiction. Assume there exists an optimal solution $O \subset S$ that does not include a_1. Let a_x be the activity in O with the earliest finishing time. Since a_1 had the smallest finishing time in all S, $f_1 \leq f_x$. There are two cases: either $f_1 \leq s_x$ or $f_1 > s_x$. In the first case, $s_1 < f_1 \leq s_x < f_x$, we have that a_1 and a_x are disjoint and therefore compatible, so we could build a better solution than O by adding a_1 to it; so this case cannot happen. We are left with the second case, in which there is overlap because a_1 finishes after a_x starts (but before a_x finishes, by hypothesis that a_1 is first to finish in S). Since a_x is first to finish in O, no activity in O occurs before a_x, thus no activity in O overlaps with a_1. Thus we could build another equally good optimal solution by substituting a_1 for a_x in O. Therefore there will always exist an optimal solution that includes a_1, QED. This tells us that the greedy strategy works well for this problem.

The general pattern for greedy algorithms is:

1. Cast the problem as one where we make a (greedy) choice and are then left with *just one* smaller problem to solve.

2. Prove that the greedy choice is always part of an optimal solution.

3. Prove that there's optimal substructure, i.e. that the greedy choice plus an optimal solution of the subproblem yields an optimal solution for the overall problem.

Because both greedy algorithms and dynamic programming exploit optimal substructure, one may get confused as to which of the two techniques to apply: using dynamic programming where a greedy algorithm suffices is wasteful, whereas using a greedy algorithm where dynamic programming is required will give the wrong answer. Here is an example of the latter situation.

The knapsack problem, of which a bewildering array of variations have been studied, can be described as follows (this will be the "0-1 knapsack" flavour). A thief has a knapsack of finite carrying capacity and, having broken into a shop, must choose which items to take with him, up to a maximum weight W. Each item i has a weight w_i, which is an integer, and a value v_i. The goal of the thief is to select a subset of the available items that maximizes the total value while keeping the total weight below the carrying capacity W.

This problem has optimal substructure because, with the usual cut-and-paste argument, it is easy to show that, having picked one item i that belongs to the optimal subset, the overall optimal solution is obtained by adding to that item the optimal solution to the 0-1 knapsack problem with the remaining items and with a knapsack of capacity $W - w_i$.

The greedy strategy might be to pick the item of greatest value. But we can easily prove with a counterexample that this won't lead to the optimal solution. Consider a carrying capacity of $W = 18$ kg, an item 1 of weight $w_1 = 10$ kg and value $v_1 = 101$ £ and two items 2 and 3 each of weight 9 kg and value 100 £. Following the stated strategy, the thief would take item 1 and would not have space for anything else, for a total value of 101 £, whereas the optimal solution is to take items 2 and 3 and go away with 200 £.

A perhaps smarter greedy strategy might be to pick the item with the highest £/kg ratio. This would have worked in the previous example but here too we can show it doesn't in general.

Exercise 24
Provide a small counterexample that proves that the greedy strategy of choosing the item with the highest £/kg ratio is not guaranteed to yield the optimal solution.

It can be shown, however, that the greedy strategy of choosing the item with the highest £/kg ratio is optimal for the fractional (as opposed to 0-1) knapsack problem, in which the thief can take an arbitrary amount, between 0 and 100%, of each available item[4].

3.3 Overview of other strategies

This course is too short to deal with every possible strategy at the same level of detail as we did for dynamic programming and greedy algorithms. The rest of this chapter is therefore just an overview, meant essentially as hints for possible ways to solve a new problem. In a professional situation, reach for your textbook to find worked examples (and, sometimes, theory) for most of these approaches.

3.3.1 Recognize a variant on a known problem

This obviously makes sense! But there can be real inventiveness in seeing how a known solution to one problem can be used to solve the essentially tricky part of another. The Graham Scan method for finding a convex hull[5], which uses as a sub-algorithm a particularly efficient way of comparing the relative positions of two vectors, is an illustration of this.

3.3.2 Reduce to a simpler problem

Reducing a problem to a smaller one tends to go hand in hand with inductive proofs of the correctness of an algorithm. Almost all the examples of recursive functions you have ever seen are illustrations of this approach. In terms of planning an algorithm, it amounts

[4]Works with gold dust but not with plasma TVs.

[5]To be covered in the *Algorithms II* course.

to the insight that it is not necessary to invent a scheme that solves a whole problem all in one step—just a scheme that is guaranteed to make non-trivial progress.

Quicksort (section 2.10), in which you sort an array by splitting it into two smaller arrays and sorting these on their own, is an example of this technique.

This method is closely related to the one described in the next section.

3.3.3 Divide and conquer

This is one of the most important ways in which algorithms have been developed. It suggests that a problem can sometimes be solved in three steps:

1. **Divide:** If the particular instance of the problem that is presented is very small, then solve it by brute force. Otherwise divide the problem into two (rarely more) parts, usually with all of the sub-components being the same size.

2. **Conquer:** Use recursion to solve the smaller problems.

3. **Combine:** Create a solution to the final problem by using information from the solution of the smaller problems.

This approach is similar to the one described in the previous section but distinct in so far as we need an explicit recombination step.

In the most common and useful cases, both the dividing and combining stages will have linear cost in terms of the problem size—certainly we expect them to be much easier tasks to perform than the original problem seemed to be. Mergesort (section 2.9) provides a classical example of this approach.

3.3.4 Backtracking

If the algorithm you need involves a search, it may be that backtracking is what is needed. This splits the conceptual design of the search procedure into two parts: the first just ploughs ahead and investigates what it thinks is the most sensible path to explore, while the second backtracks when needed. The first part will occasionally reach a dead end and this is where the backtracking part comes in: having kept extra information about the choices made by the first part, it unwinds all calculations back to the most recent choice point and then resumes the search down another path. The Prolog language makes an institution of this way of designing code. The method is of great use in many graph-related problems.

3.3.5 The MM method

This approach is perhaps a little frivolous, but effective all the same. It is related to the well known scheme of giving a million monkeys a million typewriters for a million years (the MM Method) and waiting for a Shakespeare play to be written. What you do is give your problem to a group of students (no disrespect intended or implied) and wait a few months. It is quite likely they will come up with a solution that any individual is unlikely to find. Ross Anderson once did this by setting a Tripos exam question on the problem and then writing up the edited results, with credit to the candidates, as an academic paper[6].

Sometimes a variant of this approach is automated: by systematically trying ever increasing sequences of machine instructions, one may eventually find one that has the desired behaviour. This method was once applied to the following C function:

```
int sign(int x) {
    if (x < 0) return -1;
    if (x > 0) return  1;
    return 0;
}
```

The resulting code for the i386 architecture was 3 instructions excluding the return, and for the m68000 it was 4 instructions.

In software testing, this method is the foundation for *fuzzing*: throw lots of random data at the program and see if it crashes or violates its internal assertions.

3.3.6 Look for wasted work in a simple method

It can be productive to start by designing a simple algorithm to solve a problem, and then analyze it to the extent that the critically costly parts of it can be identified. It may then be clear that, even if the algorithm is not optimal, it is good enough for your needs; or it may be possible to invent techniques that explicitly attack its weaknesses. You may view under this light the various elaborate ways of ensuring that binary trees are kept well balanced (sections 4.4 and 4.5).

3.3.7 Seek a formal mathematical lower bound

The process of establishing a proof that some task must take at least a certain amount of time can sometimes lead to insight into how an algorithm attaining that bound might be constructed—we did something similar with Selectsort (section 2.6). A properly proved lower bound can also prevent wasted time seeking improvement where none is possible.

[6]Ross Anderson, "How to cheat at the lottery (or, Massively parallel requirements engineering)", *Proc. Annual Computer Security Applications Conference, Phoenix, AZ*, 1999.

Chapter 4

Data structures

Chapter contents

Primitive data structures. Abstract data types. Pointers, stacks, queues, lists, trees. Hash tables. Binary search trees. Red-black trees. B-trees. Priority queues and heaps.
Expected coverage: about 5 lectures.

Typical programming languages such as C or Java provide primitive data types such as integers, reals, boolean values and strings. They allow these to be organized into arrays[1] which generally have a statically determined size. It is also common to provide for record data types, where an instance of the type contains a number of components, or possibly pointers to other data. C, in particular, allows the user to work with a fairly low-level idea[2] of a pointer to a piece of data. In this course a "data structure" will be implemented in terms of these language-level constructs, but will always be thought of in association with a collection of operations that can be performed with it and a number of consistency conditions which must always hold. One example of this would be the structure "sorted vector" which might be thought of as just a normal array of numbers but subject to the extra constraint that the numbers must be in ascending order. Having such a data structure may make some operations (for instance finding the largest, smallest and median numbers present) easier, but setting up and preserving the constraint (in that case ensuring that the numbers are sorted) may involve work.

Frequently, the construction of an algorithm involves the design of data structures that provide natural and efficient support for the most important steps used in the algorithm, and these data structures then call for further code design for the implementation of other necessary but less frequently performed operations.

[1]Which we have already used informally from the start.

[2]A C pointer is essentially a memory address, and it's "low level" in the sense that you can increment it to look at what's at the next address, even though nothing guarantees that it's another item of the appropriate type, or even that it's a memory address that you are allowed to read. A pointer in a higher level language might instead only give you access to the item being pointed to, without facilities for accessing adjacent portions of memory.

4.1 Implementing data structures

This section introduces some fundamental data types and their machine representation. Variants of all of these will be used repeatedly as the basis for more elaborate structures.

4.1.1 Machine data types: arrays, records and pointers

It first makes sense to agree that boolean values, characters, integers and real numbers will exist in any useful computer environment. At the level of abstraction used in this course it will generally be assumed that integer arithmetic never overflows, that floating point arithmetic can be done as fast as integer work and that rounding errors do not exist. There are enough hard problems to worry about without having to face up to the exact limitations on arithmetic that real hardware tends to impose! The so called "procedural" programming languages provide for vectors or arrays of these primitive types, where an integer index can be used to select a particular element of the array, with the access taking unit time. For the moment it is only necessary to consider one-dimensional arrays.

It will also be supposed that one can declare record data types, and that some mechanism is provided for allocating new instances of records and (where appropriate) getting rid of unwanted ones. The introduction of record types naturally introduces the use of pointers.

This course will not concern itself much about type security (despite the importance of that discipline in keeping whole programs self-consistent), provided that the proof of an algorithm guarantees that all operations performed on data are proper.

4.1.2 Vectors and matrices

Some autors use the terms "vector" and "array" interchangeably. Others make the subtle distinction that an array is simply a raw low-level type provided natively by the programming language, while a vector is an abstract data type (section 4.2) with methods and properties. We lean towards the second interpretation but won't be very pedantic on this.

A vector supports two basic operations: the first operation (read) takes an integer index and returns a value. The second operation (write) takes an index and a new value and updates the vector. When a vector is created, its size will be given and only index values inside that pre-specified range will be valid. Furthermore it will only be legal to read a value after it has been set—i.e. a freshly created vector will not have any automatically defined initial contents. Even something this simple can have several different possible implementations.

At this stage in the course we will just think about implementing vectors as blocks of memory where the index value is added to the base address of the vector to get the address of the cell wanted. Note that vectors of arbitrary objects can be handled by multiplying the index value by the size of the objects to get the physical offset of an item in memory with respect to the base, *i.e.* the address of the 0-th element.

There are two simple ways of representing two-dimensional (and indeed arbitrary multi-dimensional) matrices. The first takes the view that an $n \times m$ matrix can be implemented as an array with n items, where each item is an array of length m. The other representation starts with an array of length n which has as its elements the *addresses* of the starts of a collection of arrays of length m. One of these needs a multiplication (by

m) for every access, the other an additional memory access. Although there will only be a constant factor between these costs, at this low level it may (just about) matter; but which works better may also depend on the exact nature of the hardware involved[3].

Exercise 25
Draw the memory layout of these two representations for a 3×5 matrix, pointing out where element (1,2) would be in each case.

There is scope for wondering about whether a matrix should be stored by rows or by columns (for large matrices and particular applications this may have a big effect on the behaviour of virtual memory systems), and how special cases such as boolean matrices, symmetric matrices and sparse matrices should be represented.

4.1.3 Simple lists and doubly-linked lists

A simple and natural implementation of lists is in terms of a record structure. The list is like a little train with zero or more wagons (carriages), each of which holds one list value (the payload of the wagon) and a pointer to rest of the list (i.e. to the next wagon[4], if there is one). In C one might write

```
struct ListWagon {
    int payload; /* We just do lists of integers here */
    struct ListWagon *next; /* Pointer to the next wagon, if any */
};
```

where all lists are represented as pointers to `ListWagon` items. In C it would be very natural to use the special `NULL` pointer to stand for an empty list. We have not shown code to allocate and access lists here.

In other languages, including Java, the analogous declaration would hide the pointers:

```
class ListWagon {
    int payload;
    ListWagon next; /* The next wagon looks nested, but isn't really. */
};
```

There is a subtlety here: if pointers are hidden, how do you represent lists (as opposed to wagons)? Can we still maintain a clear distinction between lists and list wagons? And what is the sensible way of representing an empty list?

Exercise 26
Show how to declare a variable of type list in the C case and then in the Java case. Show how to represent the empty list in the Java case. Check that this value (empty list) can be assigned to the variable you declared earlier.

[3]Note also that the multiplication by m may be performed very quickly with just a shift if m is a power of 2.

[4]You might rightfully observe that it would be more proper to say "to a train with one fewer wagon". Congratulations—you are thinking like a proper computer scientist. Read on and do the exercises.

Exercise 27
As a programmer, do you notice any uncomfortable issues with your Java definition of a list? *(Requires some thought and O-O flair.)*

A different but actually isomorphic view will store lists in an array. The items in the array will be similar to the C `ListWagon` record structure above, but the `next` field will just contain an integer. An empty list will be represented by the value zero, while any non-zero integer will be treated as the index into the array where the record with the two components of a non-empty list can be found. Note that there is no need for parts of a list to live in the array in any especially neat order—several lists can be interleaved in the array without that being visible to users of the abstract data type[5]. In fact the array in this case is roughly equivalent to the whole memory in the case of the C `ListWagon`.

If it can be arranged that the data used to represent the `payload` and `next` components of a non-empty list be the same size (for instance both might be held as 32-bit values) then the array might be just an array of storage units of that size. Now, if a list somehow gets allocated in this array so that successive items in it are in consecutive array locations, it seems that about half the storage space is being wasted with the `next` pointers. There have been implementations of lists that try to avoid that by storing a non-empty list as a payload element plus a boolean flag (which takes one bit[6]) with that flag indicating if the next item stored in the array is a pointer to the rest of the list (as usual) or is in fact itself the rest of the list (corresponding to the list elements having been laid out neatly in consecutive storage units).

Exercise 28
Draw a picture of the compact representation of a list described in the notes.

The variations on representing lists are described here both because lists are important and widely-used data structures, and because it is instructive to see how even a simple-looking structure may have a number of different implementations with different space/time/convenience trade-offs.

The links in lists make it easy to splice items out from the middle of lists or add new ones. Lists provide one natural implementation of stacks (see section 4.2.1), and are the data structure of choice in many places where flexible representation of variable amounts of data is wanted.

A feature of lists is that, from one item, you can progress along the list in one direction very easily; but, once you have taken the `next` of a list, there is no way of returning (unless you independently remember where the original head of your list was). To make it possible to traverse a list in both directions one might define a new type called Doubly Linked List (DLL) in which each wagon had both a `next` and a `previous` pointer. The following equation

[5]Cfr. section 4.2.

[6]And thereby slightly reduces the space available for the payload.

```
w.next.previous == w
```

would hold for every wagon `w` except the last, while the following equation

```
w.previous.next == w
```

would hold for every wagon `w` except the first[7]. Manufacturing a DLL (and updating the pointers in it) is slightly more delicate than working with ordinary uni-directional lists. It is normally necessary to go through an intermediate internal stage where the conditions of being a true DLL are violated in the process of filling in both forward and backwards pointers.

4.1.4 Graphs

We won't give a formal definition of a graph until the *Algorithms II* course, but you probably already have a reasonable mental image of a bunch of nodes (vertices) with arrows (edges) between them. If a graph has n vertices then it can be represented by an $n \times n$ "adjacency matrix", which is a boolean matrix with entry g_{ij} true if and only if the the graph contains an edge running from vertex i to vertex j. If the edges carry data (for instance the graph might represent an electrical network and we might need to represent the impedance of the component on each edge) then the cells of the matrix might hold, say, complex numbers (or whatever else were appropriate for the application) instead of booleans, with some special value reserved to mean "no link".

An alternative representation would represent each vertex by an integer, and have a vector such that element i in the vector holds the head of a list (an "adjacency list") of all the vertices connected directly to edges radiating from vertex i.

The two representations clearly contain the same information, but they do not make it equally easily available. For a graph with only a few edges attached to each vertex, the list-based version may be more compact, and it certainly makes it easy to find a vertex's neighbours, while the matrix form gives instant responses to queries about whether a random pair of vertices are joined, and can be more compact (especially when there are very many edges, and if the bit-array is stored in packed form to make full use of machine words). We shall have much more to say about graphs in the *Algorithms II* course.

4.2 Abstract data types

When designing data structures and algorithms it is desirable to avoid making decisions based on the accident of how you first sketch out a piece of code. All design should be motivated by the explicit needs of the application. The idea of an Abstract Data Type (ADT) is to support this[8]. The specification of an ADT is a list of the operations that may be performed on it, together with the identities (invariants) that they satisfy. This specification does *not* show how to implement anything in terms of any simpler

[7]Yet another variant would be the circular doubly linked list, in which these equations would hold for all elements without exceptions. We shall use circular doubly linked lists in *Algorithms II* when we implement Fibonacci heaps.

[8]The idea is generally considered good for program maintainability as well, but that is not the primary concern of this particular course.

data types. The user of an ADT is expected to view this specification as the complete description of how the data type and its associated functions will behave—no other way of interrogating or modifying data is available, and the response to any circumstances not covered explicitly in the specification is deemed undefined.

In Java, the idea of the Abstract Data Type can be expressed with the `interface` language construct, which looks like a class with all the so-called "signatures" of the class methods (i.e. the types of the input and output parameters) but without any implementations. You can't instantiate objects from an interface; you must first derive a genuine class from the interface and then instantiate objects from the class. And you can derive several classes from the same interface, each of which implements the interface in a different way—that's the whole point. In the rest of this chapter I shall describe ADTs with pseudocode resembling Java interfaces. One thing that is missing from the Java `interface` construct, however, is a formal way to specify the invariants that the ADT satisfies; on the other hand, at the level of this course we won't even attempt to provide a formal definition of the semantics of the data type through invariants, so I shall informally just resort to comments in the pseudocode.

Using the practice we already introduced when describing sorting algorithms, each method will be tagged with a possibly empty *precondition*[9] (something that must be true before you invoke the method, otherwise it won't work) then with an imperative description (labelled *behaviour*) of what the method must do and finally with a possibly empty *postcondition* (something that the method promises will be true after its execution, provided that the precondition was true when you invoked it).

Examples given later in this course should illustrate that making an ADT out of even quite simple operations can sometimes free one from enough preconceptions to allow the invention of amazingly varied collections of implementations.

4.2.1 The Stack abstract data type

Let us now introduce the Abstract Data Type for a Stack: the standard mental image is that of a pile of plates in your college buttery. The distinguishing feature of this structure is that the only easily accessible item is the one on top of the stack. For this reason this data structure is also sometimes indicated as LIFO, which stands for "Last in, first out".

[9]Some programming languages allow you to enter such invariants in the code not just as comments but as *assertions*—a brilliant feature that is unfortunately underused by all but the wisest programmers.

```
ADT Stack {
  boolean isEmpty();
  // BEHAVIOUR: return true iff the structure is empty.

  void push(item x);
  // BEHAVIOUR: add element <x> to the top of the stack.
  // POSTCONDITION: isEmpty() == false.
  // POSTCONDITION: top() == x

  item pop();
  // PRECONDITION: isEmpty() == false.
  // BEHAVIOUR: return the element on top of the stack.
  // As a side effect, remove it from the stack.

  item top();
  // PRECONDITION: isEmpty() == false.
  // BEHAVIOUR: Return the element on top of the stack (without removing it).
}
```

In the ADT spirit of specifying the semantics of the data structure using invariants, we might also add that, for each stack `s` and for each item `x`, after the following two-operation sequence

```
s.push(x)
s.pop()
```

the return value of the second statement is `x` and the stack `s` "is the same as before"; but there are technical problems in expressing this correctly and unambiguously using the above notation, so we won't try. The idea here is that the definition of an ADT should collect all the essential details and assumptions about how a structure must behave (although the expectations about common patterns of use and performance requirements are generally kept separate). It is then possible to look for different ways of implementing the ADT in terms of lower level data structures.

Observe that, in the Stack type defined above, there is no description of what happens if a user tries to compute `top()` when `isEmpty()` is `true`, i.e. when the precondition of the method is violated. The outcome is therefore undefined, and an implementation would be entitled to do *anything* in such a case—maybe some garbage value would get returned without any mention of the problem, maybe an error would get reported or perhaps the computer would crash its operating system and delete all your files. If an ADT wants exceptional cases to be detected and reported, it must specify this just as clearly as it specifies all other behaviour.

The stack ADT given above does not make allowance for the `push` operation to fail—although, on any real computer with finite memory, it must be possible to do enough successive pushes to exhaust some resource. This limitation of a practical realization of an ADT is not deemed a failure to implement the ADT properly: at the level of abstraction of this introductory algorithms course, we do not really admit to the existence of resource limits!

There can be various different implementations of the Stack data type, but two are especially simple and commonly used. The first represents the stack as a combination of an array and an index (pointing to the “top of stack”, or TOS). The `push` operation writes a value into the array and increments the index[10], while `pop` does the converse. The second representation of stacks is as linked lists, where pushing an item just adds an extra cell to the front of a list, and popping removes it. In both cases the `push` and `pop` operations work by modifying stacks in place, so (unlike what might happen in a functional language such as ML) after invoking either of them the original stack is no longer available.

Stacks are useful in the most diverse situations. The page description language PostScript is actually, as you may know, a programming language organized around a stack (and the same is true of Forth, which may have been an inspiration). Essentially in such languages the program is a string of tokens that include operands and operators. During program execution, any operands are pushed on the stack; operators, instead, pop from the stack the operands they require, do their business on them and finally push the result back on the stack. For example, the program

```
3 12 add 4 mul 2 sub
```

computes $(3 + 12) \times 4 - 2$ and leaves the result, 58, on the stack. This way of writing expressions is called Reverse Polish Notation and one of its attractions is that it makes parentheses unnecessary (at the cost of having to reorder the expression and making it somewhat less legible).

Exercise 29
Invent (or should I say “rediscover”?) a linear-time algorithm to convert an infix expression such as
`(3+12)*4 - 2`
into a postfix one without parentheses such as
`3 12 + 4 * 2 -`.
By the way, would the reverse exercise have been easier or harder?

4.2.2 The List abstract data type

We spoke of linked lists as a basic low level building block in section 4.1.3, but here we speak of the ADT, which we define by specifying the operations that it must support. Note how you should be able to implement the List ADT with any of the implementations described in 4.1.3 (wagons and pointers, arrays and so forth) although sometimes the optimizations will get in the way (e.g. packed arrays). The List version defined here will allow for the possibility of re-directing links in the list. A really full and proper definition of the ADT would need to say something rather careful about when parts of lists are really the same (so that altering one alters the other) and when they are similar in structure and values but distinct[11]. Such issues will be ducked for now but must be clarified before writing programs, or they risk becoming the source of spectacular bugs.

[10]Note that stacks growing in the reverse direction are also plausible and indeed frequent. (Why is that?)

[11]For example, does the `tail()` method return a copy of the rest of the list or a pointer to it? And similarly for `setTail()`.

```
ADT List {
  boolean isEmpty();
  // BEHAVIOUR: Return true iff the structure is empty.

  item head(); // NB: Lisp people might call this ‘‘car’’.
  // PRECONDITION: isEmpty() == false
  // BEHAVIOUR: return the first element of the list (without removing it).

  void prepend(item x); // NB: Lisp people might call this ‘‘cons’’.
  // BEHAVIOUR: add element <x> to the beginning of the list.
  // POSTCONDITION: isEmpty() == false
  // POSTCONDITION: head() == x

  List tail(); // NB: Lisp people might call this ‘‘cdr’’.
  // PRECONDITION: isEmpty() == false
  // BEHAVIOUR: return the list of all the elements except the first (without
  // removing it).

  void setTail(List newTail);
  // PRECONDITION: isEmpty() == false
  // BEHAVIOUR: replace the tail of this list with <newTail>.
}
```

You may note that the List type is very similar to the Stack type mentioned earlier. In some applications it might be useful to have a variant on the List data type that supported a `setHead()` operation to update list contents (as well as chaining) in place, or an `isEqualTo()` test. Applications of lists that do not need `setTail()` may be able to use different implementations of lists.

4.2.3 The Queue and Deque abstract data types

In the Stack ADT, the item removed by the `pop()` operation was the most recent one added by `push()`. A Queue[12] is in most respects similar to a stack, but the rules are changed so that the item accessed by `top()` and removed by `pop()` will be the oldest one inserted by `push()` (we shall rename these operations to avoid confusion). Even if finding a neat way of expressing this in a mathematical description of the Queue ADT may be a challenge, the idea is simple. The above description suggests that stacks and queues will have very similar interfaces. It is sometimes possible to take an algorithm that uses a stack and obtain an interesting variant by using a queue instead; and vice-versa.

[12]Sometimes referred to as a FIFO, which stands for “First In, First Out”.

```
ADT Queue {
  boolean isEmpty();
  // BEHAVIOUR: return true iff the structure is empty.

  void put(item x);
  // BEHAVIOUR: insert element <x> at the end of the queue.
  // POSTCONDITION: isEmpty() == false

  item get();
  // PRECONDITION: isEmpty() == false
  // BEHAVIOUR: return the first element of the queue, removing it
  // from the queue.

  item first();
  // PRECONDITION: isEmpty() == false
  // BEHAVIOUR: return the first element of the queue, without removing it.


}
```

A variant is the perversely-spelt Deque (double-ended queue), which is accessible from both ends both for insertions and extractions and therefore allows four operations:

```
ADT Deque {
  boolean isEmpty();

  void putFront(item x);
  void putRear(item x);
  // POSTCONDITION for both: isEmpty() == false

  item getFront();
  item getRear();
  // PRECONDITION for both: isEmpty() == false
}
```

The Stack and Queue may be seen as subcases of the Deque in which two of the four operations[13] are disabled.

4.2.4 The Table abstract data type

In computer science, the word "table" is used in several distinct senses: think of a database table, an HTML table, a table of allowed values and so on. Right now we are concerned with the kind of table that associates values (e.g. telephone numbers) with keys (e.g. names) and allows you to look up a value if you supply the key. Note that, within the table, the mapping between keys and values is a function[14]: you cannot have different values associated with the same key. For generality we assume that keys are of type `Key` and that values are of type `Value`.

[13]One `put` and one `get`, for obvious reasons.
[14]As opposed to a generic relation. In other words, only one arrow goes out of each source element.

```
ADT Table {
  boolean isEmpty();
  // BEHAVIOUR: return true iff the structure is empty.

  void set(Key k, Value v);
  // BEHAVIOUR: store the given (<k>, <v>) pair in the table.
  // If a pair with the same <k> had already been stored, the old
  // value is overwritten and lost.
  // POSTCONDITION: get(k) == v

  Value get(Key k);
  // PRECONDITION: a pair with the sought key <k> is in the table.
  // BEHAVIOUR: return the value associated with the supplied <k>,
  // without removing it from the table.

  void delete(Key k);
  // PRECONDITION: a pair with the given key <k> has already been inserted.
  // BEHAVIOUR: remove from the table the key-value pair indexed by
  // the given <k>.
}
```

Observe that this simple version of a table does not provide a way of asking if some key is in use, and it does not mention anything about the number of items that can be stored in a table. Practical implementations may concern themselves with both these issues.

Probably the most important special case of a table is when the keys are known to be drawn from the set of integers in the range $0..n$ for some modest n. In that case the table can be modelled directly by a simple vector, and both `set()` and `get()` operations have unit cost. If the key values come from some other integer range (say $a..b$) then subtracting a from key values gives a suitable index for use with a vector.

If the number of keys that are actually used is much smaller than the number $b - a$ of items in the range that the keys lie in, this vector representation becomes inefficient in space, even though its time performance remains optimal.

Exercise 30
How would you deal efficiently with the case in which the keys are English words? *(There are several possible schemes of various complexity that would all make acceptable answers provided you justified your solution.)*

For sparse tables one could try holding the data in a list, where each item in the list could be a record storing a key-value pair. The `get()` function can just scan along the list, searching for the key that is wanted; if one is not found, the function behaves in an undefined way. But now there are several options for the `set()` function. The first natural one just sticks a new key-value pair at the front of the list, allowing `get()` to be coded so as to retrieve the first value that it finds. The second one would scan the list and, if a key was already present, it would update the associated value in place. If the required key was not present it would have to be added.

Exercise 31
Should the new key-value pair added by `set()` be added at the start or the end of the list? Or elsewhere?

Since in this second case duplicate keys are avoided, the order in which items in the list are kept will not affect the correctness of the data type, and so it would be legal (if not always useful) to make arbitrary permutations of the list each time it was touched.

If one assumes that the keys passed to `get()` are randomly selected and uniformly distributed over the complete set of keys used, the linked list representation calls for a scan down (an average of) half the length of the list. For the version that always adds a new key-value pair at the head of the list, this cost increases without limit as values are changed. The other version keeps that cost down but has to scan the list when performing `set()` operations as well as `get()`s.

To try to get rid of some of the overhead of the linked list representation, keep the idea of storing a table as a bunch of key-value pairs but now put these in an array rather than a linked list. Now suppose that the keys used are ones that support an ordering, and sort the array on that basis. Of course there now arise questions about how to do the sorting and what happens when a new key is mentioned for the first time—but here we concentrate on the data retrieval part of the process. Instead of a linear search as was needed with lists, we can now probe the middle element of the array and, by comparing the key there with the one we are seeking, can isolate the information we need in one or the other half of the array. If the comparison has unit cost, the time needed for a complete look-up in a table with elements will satisfy

$$f(n) = f(n/2) + k$$

and the solution to this recurrence shows us that the complete search can be done in $\Theta(\lg n)$.

Exercise 32
Solve the recurrence, again with the trick of setting $n = 2^m$.

Another representation of a table that also provides $\lg n$ costs is obtained by building a binary tree, where the tree structure relates very directly to the sequence of comparisons that could be done during binary search in an array. If a tree of n items can be built up with the median key from the whole data set in its root, and each branch is similarly well balanced, the greatest depth of the tree will be around $\lg n$.
Having a linked representation makes it fairly easy to adjust the structure of a tree when new items need to be added, but details of that will be left until section 4.3. Note that, in such a tree, all items in the left sub-tree come before the root in sorting order, and all those in the right sub-tree come after.

4.2.5 The Set abstract data type

There are very many places in the design of larger algorithms where it is necessary to have ways of keeping sets of objects. In different cases, different operations will be important, and finding ways in which various subsets of the possible operations can be best optimized leads to the discussion of a large range of sometimes quite elaborate representations and procedures. We shall cover some of the more important (and more interesting) options in this course.

Until we are more specific about the allowed operations, there isn't much difference between a Set and the Table seen in the previous section: we are still storing key-value pairs, and keys are unique. But now we *are* going to be more specific, and define extra operations for the Set by extending the basic ones introduced for the Table. We may think of many plausible extensions that are quite independent of each other. Since this is a course on data structures and not on object-oriented programming, in the pseudocode I shall gloss over the finer points of multiple inheritance, mixin classes and diamond diagrams, but I hope the spirit is clear: the idea is that you could form a Set variant that suits your needs by adding almost any combination of the following methods to the Table ADT seen earlier.

Simple variants could just add elementary utilities:

```
boolean hasKey(Key x);
// BEHAVIOUR: return true iff the set contains a pair keyed by <x>.

Key chooseAny();
// PRECONDITION: isEmpty() == false
// BEHAVIOUR: Return the key of an arbitrary item from the set.
```

For a more sophisticated variant, let us introduce the assumption that there exists a total order on the set of keys—something that the Table did not require[15]. We may then meaningfully introduce methods that return the smallest or largest key of the set, or the next largest or next smallest with respect to a given one.

```
Key min();
// PRECONDITION: isEmpty() == false
// BEHAVIOUR: Return the smallest key in the set.

Key max();
// PRECONDITION: isEmpty() == false
// BEHAVIOUR: Return the largest key in the set.

Key predecessor(Key k);
// PRECONDITION: hasKey(k) == true
// PRECONDITION: min() != k
// BEHAVIOUR: Return the largest key in the set that is smaller than <k>.

Key succcessor(Key k);
// PRECONDITION: hasKey(k) == true
// PRECONDITION: max() != k
```

[15]One might argue that the fairly arbitrary names of Set and Table should be reversed, since in mathematics one of the distinguishing characteristics of a set is that it is unordered.

```
  // BEHAVIOUR: Return the smallest key in the set that is larger than <k>.
```

Another interesting and sometimes very useful feature is the ability to form a set as the union of two sets. Note how a proper ADT definition would have to be much more careful about specifying whether the original sets are preserved or destroyed by the operation, as well as detailing what to do if the two sets contain pairs with the same key but different values.

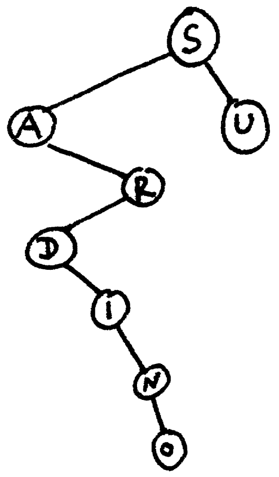
```
  Set unionWith(Set s);
  // BEHAVIOUR: Return the set obtained by forming the union of this
  // set and <s>.
```

The remaining sections in this chapter will describe implementations of specific variations on the Table/Set theme, each with its own distinctive features and trade-offs in terms of supported operations and efficiency in space and time.

4.3 Binary search trees

A binary search tree requires a key space with a total order and implements the Set variant that also supports the computation of `min()`, `max()`, `predecessor()` and `successor()`.

Each node of the binary tree will contain an item (consisting of a key-value pair[16]) and pointers to two sub-trees, the left one for all items with keys smaller than the stored one and the right one for all the items with larger keys.



Searching such a tree is simple: just compare the sought key with that of the visited node and, until you find a match, recurse down in the left or right subtree as appropriate. The maximum and minimum values in the tree can be found in the leaf nodes discovered by following all left or right pointers (respectively) from the root.

[16]If the value takes up more than a minimal amount of space, it is actually stored elsewhere as "satellite data" and only a pointer is stored together with the key.

Exercise 33
(Clever challenge, straight from CLRS3—exercise 12.2-4.) Professor Bunyan thinks he has discovered a remarkable property of binary search trees. Suppose that the search for key k in a binary search tree ends up in a leaf. Consider three sets: A, the keys to the left of the search path; B, the keys on the search path; and C, the keys to the right of the search path. Professor Bunyan claims that any three keys $a \in A$, $b \in B$, and $c \in C$ must satisfy $a \leq b \leq c$. Give a smallest possible counterexample to the professor's claim.

To find the successor[17] of a node x whose key is k_x, look in x's right subtree: if that subtree exists, the successor node s must be in it—otherwise any node in that subtree would have a key between k_x and k_s, which contradicts the hypothesis that s is the successor. If the right subtree does not exist, the successor may be higher up in the tree. Go up to the parent, then grand-parent, then great-grandparent and so on until the link goes up-and-right rather than up-and-left, and you will find the successor. If you reach the root before having gone up-and-right, then the node has no successor: its key is the highest in the tree.

Exercise 34
Why, in BSTs, does this up-and-right business find the successor? Can you sketch a proof?

Exercise 35
(Important.) Prove that, in a binary search tree, if node n has two children, then its successor has no left child.

To insert in a tree, one searches to find where the item ought to be and then inserts there. Deleting a leaf node is easy. To delete a non-leaf is harder, and there will be various options available. A non-leaf with only one child can be simply replaced by its child. For a non-leaf with two children, an option is to replace it with its successor (or predecessor—either will work). Then the item for deletion can't have two children (cfr. exercise above) and can thus be deleted in one of the ways already seen; meanwhile, the newly moved up object satisfies the order requirements that keep the tree structure valid.

Exercise 36
Prove that this deletion procedure, when applied to a valid binary search tree, always returns a valid binary search tree.

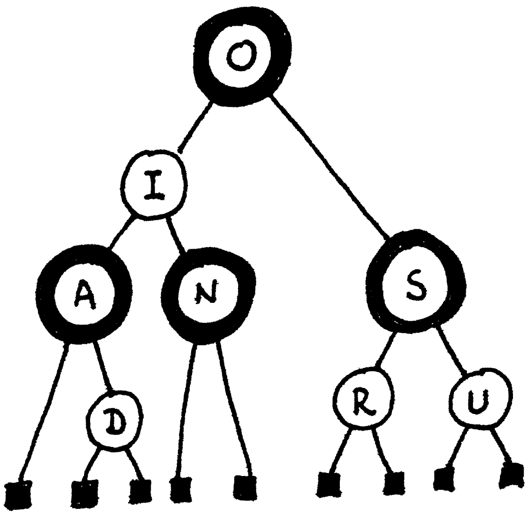
[17]The case of the predecessor is analogous, except that left and right must be reversed.

Most of the useful operations on binary search trees (`get()`, `min()`, `max()`, `successor()` and so on) have cost proportional to the depth of the tree. If trees are created by inserting items in random order, they usually end up pretty well balanced, and this cost will be $O(\lg n)$. But the tree pictured as an example at the beginning of this section, while valid, is instead very unbalanced: it has a depth almost equal to its number of nodes. An easy way to obtain a worst-case BST is to build it by inserting items in ascending order: then the tree degenerates into a linear list of height n. It would be nice to be able to re-organize things to prevent that from happening. In fact there are several methods that work, and the trade-offs between them relate to the amount of space and time that will be consumed by the mechanism that keeps things balanced. The next section describes two (related) sensible compromises.

4.4 Red-black trees and 2-3-4 trees

4.4.1 Definition of red-black trees

Red-black trees are special binary search trees that are guaranteed always to be reasonably balanced: no path from the root to a leaf is more than twice as long as any other.

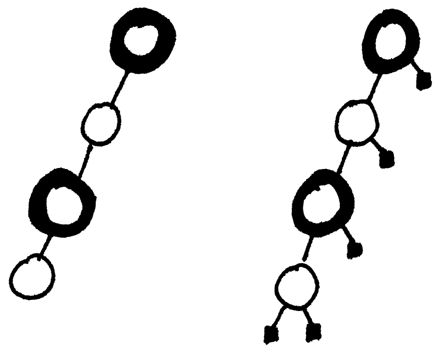


Formally, a red-black tree can be defined as a binary search tree that satisfies the following five invariants.

1. Every node is either red or black.
2. The root is black.
3. All leaves are black and never contain key-value pairs[18].
4. If a node is red, both its children are black.
5. For each node, all paths from that node to descendent leaves contain the same number of black nodes.

[18]Since leaves carry no information, they are sometimes omitted from the drawings; but they are necessary for the consistency of the remaining invariants.

The most important property is clearly the last: the others are only relevant as a framework that allows us to enforce the last property.



The above formulation sometimes confuses people, leading alert readers wondering whether a linear list of alternating black and red nodes wouldn't match the properties and yet be maximally unbalanced. Or do the rules say anywhere that each node of the binary tree is forced to have two children? The answer is somewhat subtle. No, the rules don't require each node to have two children; however each binary node will have two *pointers* to its children and, if either or both of these children is missing, the corresponding pointer(s) will be null. The subtlety is that we consider all these null pointers to be black leaves. You could imagine memory position 0, where null pointers point, to be holding the archetypal black leave; and, since leaves carry no information, we don't need to store distinct ones—an indicator that "there's a black leaf there" is sufficient, and that's precisely what the null pointer does. But what matters is that those hidden leaves are there, so the linear list is not a valid Red-Black tree because the paths from the root to the little black leaves hanging off the side of each node in the chain violate the fifth invariant.
From invariant 4 you see that no path from root to leaf may contain two consecutive red nodes. Therefore, since each path starts with a black root (invariant 2) and ends with a black leaf (invariant 3), the number of red nodes in the path is at most equal to that of the black nodes[19]. Since, by invariant 5, the number of black nodes in each path from the root to any leaf, say b, is the same across the tree, all such paths have a node count between b and $2b$.

Exercise 37
What are the smallest and largest possible number of nodes of a red-black tree of height h?

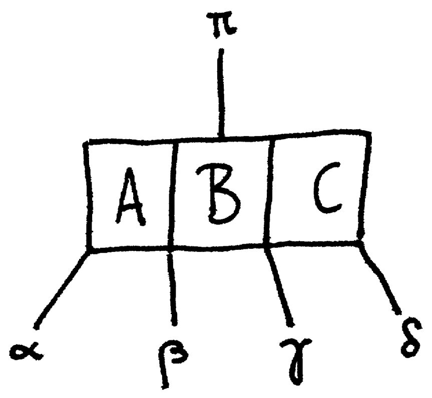
It is easy to prove from this that the maximum depth of an n-node red-black tree is $O(\lg n)$, which is therefore the time cost of `get()`, `min()`, `max()`, `successor()` and so on. But what about `set()`, i.e. inserting a new item? Finding the correct place in the tree involves an algorithm very similar to that of `get()`, but we also need to preserve the red-black properties and this can be tricky. There are complicated recipes to follow,

[19]Minus one if we count the empty leaf as one of the nodes, but see footnote 18.

based on left and right rotations, for restoring those properties after an insertion; but, to justify them, we must first take a detour and discuss the related family of (non-binary) 2-3-4 trees[20].

4.4.2 2-3-4 trees

Binary trees had one key and two outgoing pointers in each node. 2-3-4 trees generalize this to allow nodes to contain more keys and pointers. Specifically they also allow 3-nodes, with three outgoing pointers and therefore 2 keys, and 4-nodes, with 4 outgoing pointers and therefore 3 keys. As with regular binary trees, the pointers are all to subtrees which only contain key values limited by the keys in the parent node: in the following picture, the branch labelled β leads to a subtree containing keys k such that "A" $\leq k \leq$ "B".



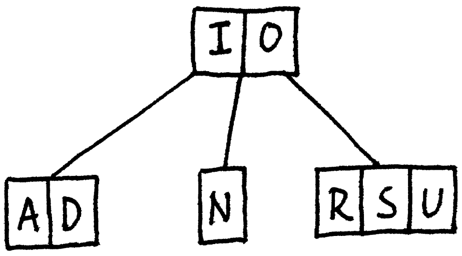
Searching a 2-3-4 tree is almost as easy as searching a binary tree. Any concern about extra work within each node should be balanced by the realization that, with a larger branching factor, 2-3-4 trees will generally be shallower than pure binary trees.

Inserting into a 2-3-4 node also ends up being fairly easy, and—even better—it turns out that a simple insertion process automatically leads to balanced trees. To insert, search down through the tree looking for where the new item must be added. If an item with the same key is found, just overwrite it[21]—the structure of the tree does not change at all in this trivial case. If you can't find the key, continue *until the bottom level* to reach the place where the new key should be added. If that place is a 2-node or a 3-node, then the item can be stuck in without further ado, upgrading that node to a 3-node or 4-node respectively. If the insertion was going to be into a 4-node, something has to be done to make space for the newcomer. The operation needed is to decompose the 4-node into a pair of 2-nodes before attempting the insertion—this then means that the parent of the original 4-node will gain an extra key and an extra child. To ensure that there will be room for this, we apply some foresight. While searching down the tree to find where to make an insertion, if we ever come across a 4-node we split it immediately; thus, by the time we go down and look at its offspring and have our final insertion to perform, we can be certain that there are no 4-nodes in the tree between the root and where we are. If

[20]Note that 2-3-4 trees are one of the very few topics in this course's syllabus that are not discussed in the CLRS3 textbook—so pay special attention, as the explanation is illuminating.

[21]Meaning: replace the old value (or pointer to satellite data) with the new.

the root node gets to be a 4-node, it can be split into three 2-nodes, and this is the only circumstance when the height of the tree increases.



The key to understanding why 2-3-4 trees remain balanced is the recognition that splitting a 4-node (other than the root) does not alter the length of any path from the root to a leaf of a tree. Splitting the root increases the length of all paths by 1. Thus, at all times, all paths through the tree from root to a leaf have the same length. The tree has a branching factor of at least 2 at each level, and so, in a tree with n items, all items will be at worst $\lg n$ down from the root.

Deletions require some more intellectual effort and attention to detail. We shall revisit them when talking of B-trees in section 4.5.

4.4.3 Understanding red-black trees

Let's now get back to red-black trees, which are really nothing more than a trick to transport the clever ideas of 2-3-4 trees into the more uniform and convenient realm of pure binary trees. The 2-3-4 trees are easier to understand but less convenient to code, because of all the complication associated with having three different types of nodes and several keys (and therefore comparison points) for each node. The binary red-black trees have complementary advantages and disadvantages.

The idea is that we can represent any 2-3-4 tree as a red-black tree: a red node is used as a way of providing extra pointers, while a black node stands for a regular 2-3-4 node. Just as 2-3-4 trees have the same number (k, say) of nodes from root to each leaf, red-black trees always have k black nodes on any path, and can have from 0 to k red nodes as well. Thus the depth of the new red-black tree is at worst twice that of a 2-3-4 tree. Insertions and node splitting in red-black trees just have to follow the rules that were set up for 2-3-4 trees.

The key to understanding red-black trees is to map out explicitly the isomorphism between them and 2-3-4 trees. So, to set you off in the right direction…

Exercise 38
For each of the three possible types of 2-3-4 nodes, draw an isomorphic "node cluster" made of 1, 2 or 3 red-black nodes. The node clusters you produce must:

- Have the same number of keys, incoming links and outgoing links as the corresponding 2-3-4 nodes. as the corresponding 2-3-4 nodes.
- Respect all the red-black rules when composed with other node clusters.

Searching a red-black tree involves exactly the same steps as searching a normal binary tree, but the balanced properties of the red-black tree guarantee logarithmic cost. The work involved in inserting into a red-black tree is quite small too. The programming ought to be straightforward, but if you try it you will probably feel that there seem to be uncomfortably many cases to deal with, and that it is tedious having to cope with both each case and its mirror image. But, with a clear head, it is still fundamentally OK.

4.5 B-trees

The trees described so far (BSTs, red-black trees and 2-3-4 trees) are meant to be instantiated in dynamically allocated main memory. With data structures kept on disc, instead, it is sensible to make the unit of data fairly large—perhaps some size related to the natural storage unit that your physical disc uses (a sector, cluster or track). Minimizing the total number of separate disc accesses will be more important than getting the ultimately best packing density. There are of course limits, and use of over-the-top data blocks will use up too much fast main memory and cause too much unwanted data to be transferred between disc and main memory along with each necessary bit.

B-trees are a good general-purpose disc data structure. We start by generalizing the idea of a sorted binary tree to a tree with a very high branching factor. The expected implementation is that each node will be a disc block containing an alternation of pointers to sub-trees and keys[22]. This will tend to define the maximum branching factor that can be supported in terms of the natural disc block size and the amount of memory needed for each key. When new items are added to a B-tree it will often be possible to add the item within an existing block without overflow. Any block that becomes full can be split into two, and the single reference to it from its parent block expands to the two references to the new half-empty blocks. For B-trees of reasonable branching factor, any reasonable amount of data can be kept in a quite shallow tree: although the theoretical cost of access grows with the logarithm of the number of data items stored, in practical terms it is constant.

Each node of a B-tree has a lower and an upper bound on the number of keys it may contain[23]. When the number of keys exceeds the upper bound, the node must be split; when the number of keys goes below the lower bound, the node must be merged with another one (potentially triggering other rearrangements). The tree as a whole is characterized by an integer parameter $t \geq 2$ called the **minimum degree** of the B-tree: each node must have between t and $2t$ pointers[24], and therefore between $t - 1$ and $2t - 1$ keys. There is a variant known as B*-tree ("b star tree") in which non-root internal nodes must be at least $2/3$ full, rather than at least $1/2$ full as in the regular B-tree. The formal rules can be stated as follows.

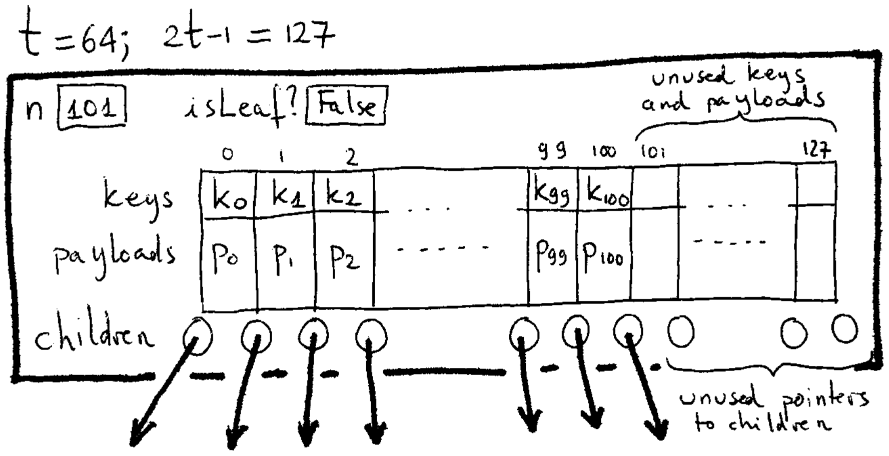
1. There are internal nodes (with keys and payloads and children) and leaf nodes (without keys or payloads or children).

[22]More precisely key-value pairs, as usual, since the reason for looking up a key is ultimately to retrieve the value or satellite data associated with it. In practice the "payload" shown in the picture below each key will often be a *pointer* to the actual payload, unless values are of small and constant size.

[23]Except that no lower bound is imposed on the root, otherwise it would be impossible to represent B-trees that were nearly empty.

[24]See footnote 23 again.

2. For each key in a node, the node also holds the associated payload[25].
3. All leaf nodes are at the same distance from the root.
4. All internal nodes have at most $2t$ children; all internal nodes except the root have at least t children.
5. A node has c children iff it has $c - 1$ keys.



The algorithms for adding new data into a B-tree ensure that the tree remain balanced. This means that the cost of accessing data in such a tree can be guaranteed to remain low even in the worst case. The ideas behind keeping B-trees balanced are a generalization of those used for 2-3-4-trees[26] but note that the implementation details may be significantly different, firstly because the B-tree will have such a large branching factor and secondly because all operations will need to be performed with a view to the fact that the most costly step is reading a disc block. In contrast, 2-3-4-trees are typically used as in-memory data structures so you count memory accesses rather than disc accesses when evaluating and optimizing an implementation.

4.5.1 Inserting

To insert a new key (and payload) into a B-tree, look for the key in the B-tree in the usual way. If found, update the payload in place. If not found, you'll be by then in the right place at the bottom level of the tree (the one where nodes have keyless leaves as children); on the way down, whenever you find a full node, split it in two on the median key and migrate the median key and resulting two children to the parent node (which by inductive hypothesis won't be full). If the root is full when you start, split it into three nodes (yielding a new root with only one key and adding one level to the tree). Once you get to the appropriate bottom level node, which won't be full or you would have split it on your way there, insert there.

[25]Or, more likely, a pointer to it, unless the payload has a small fixed size comparable to that of the pointer.

[26]Structurally, you can view 2-3-4 trees as a subcase of B-trees with $t = 2$.

Exercise 39
(The following is not hard but it will take somewhat more than five minutes.)
Using a soft pencil, a large piece of paper and an eraser, draw a B-tree with $t = 2$, initially empty, and insert into it the following values in order:

$$63, 16, 51, 77, 61, 43, 57, 12, 44, 72, 45, 34, 20, 7, 93, 29.$$

How many times did you insert into a node that still had room? How many node splits did you perform? What is the depth of the final tree? What is the ratio of free space to total space in the final tree?

4.5.2 Deleting

Deleting is a more elaborate affair because it involves numerous subcases.

You can't delete a key from anywhere other than a bottom node (i.e. one whose children are keyless leaves), otherwise you upset its left and right children that lose their separator. In addition, you can't delete a key from a node that already has the minimum number of keys. So the general algorithm consists of creating the right conditions and then deleting (or, alternatively, deleting and then readjusting).

To move a key to a bottom node for the purpose of deleting it, swap it with its successor (which must be in a bottom node). The tree will have a temporary inversion, but that will disappear as soon as the unwanted key is deleted.

Exercise 40
Prove that, if a key is not in a bottom node, its successor, if it exists, must be.

To refill a node that has too few keys, use an appropriate combination of the following three operations, which rearrange a local part of a B-tree in constant time preserving all the B-tree properties.

Merge The first operation *merges* two adjacent brother nodes and the key that separates them from the parent node. The parent node loses one key.

Split The reverse operation *splits* a node into three: a left brother, a separating key and a right brother. The separating key is sent up to the parent.

Redistribute The last operation *redistributes* the keys among two adjacent sibling nodes. It may be thought of as a merge followed by a split in a different place[27], and this different place will typically be the centre of the large merged node.

[27]We say "thought of" because such a merge might be disallowed as a stand-alone B-tree operation—the resulting node might end up having more than the allowed number of keys.

Each of these operations is only allowed if the new nodes thus formed respect their min and max capacity constraints.

Here is then the complete pseudocode algorithm to delete a key k from the B-tree:

```
def delete(k):
    """B-tree method for deleting key k.
    PRECONDITION: k is in this B-tree.
    POSTCONDITION: k is no longer in this B-tree."""

    if k is in a bottom node B:
        if B can lose a key without becoming too small:
            delete k from B locally
        else:
            refill B (see below)
            delete k from B locally
    else:
        swap k with its successor
        # ASSERT: now k is in a bottom node
        delete k from the bottom node with a recursive invocation
```

To refill a node B that currently has the min number of keys:

```
def refill(B):
    """B-tree method for refilling node B.
    PRECONDITION: B is an internal node of this B-tree, with t-1 keys.
    POSTCONDITION: B now has more than t-1 keys."""

    if either the left or right sibling of B can afford to lose any keys:
        redistribute keys between B and that sibling
    else:
        # ASSERT: B and its siblings all have the min number of keys, t-1
        merge B with either of its siblings
        # ...this may require recursively refilling the parent of B,
        # because it will lose a key during the merge.
```

4.6 Hash tables

A hash table implements the general case of the Table ADT, where keys may not have a total order defined on them. In fact, even when the keys used *do* have an order relationship associated with them, it may be worth looking for a way of building a table without using this order. Binary search makes locating things in a table easier by imposing a coherent and ordered structure; hashing, instead, places its bet the other way, on chaos.

A hash function $h(k)$ maps a key of arbitrary length onto an integer in the range 1 to N for some N and, for a good hash function, this mapping will appear to have hardly any pattern. Now, if we have an array of size N, we can try to store a key-value pair with key k at location $h(k)$ in the array. Unfortunately, sometimes two distinct keys k_1 and k_2 will map to the same location $h(k_1) = h(k_2)$; this is known as a *collision*. There are two main strategies for handling it.

Chaining. We can arrange that the locations in the array hold little linear lists that collect all the keys that hash to that particular value. A good hash function will distribute keys fairly evenly over the array, so with luck this will lead to lists with average length n/N if n keys are in use[28].

Exercise 41
(Trivial) Make a hash table with 8 slots and insert into it the following values:

$$15, 23, 12, 20, 19, 8, 7, 17, 10, 11.$$

Use the hash function

$$h(k) = (k \bmod 10) \bmod 8$$

and, of course, resolve collisions by chaining.

Open addressing. The second way of using hashing is to use the hash value $h(n)$ as just a first preference for where to store the given key in the array. On adding a new key, if that location is empty then well and good—it can be used; otherwise, a succession of other probes are made of the hash table according to some rule until either the key is found to be present or an empty slot for it is located. The simplest (but not the best) method of collision resolution is to try successive array locations on from the place of the first probe, wrapping round at the end of the array. Note that the open addressing table, unlike the chaining one, may become full, and that its performance decreases significantly when it is nearly full; implementations will typically double the size of the whole table once occupancy goes above a certain threshold.

Exercise 42
Non-trivial Imagine redoing the exercise above but resolving collisions by open addressing. When you go back to the table to retrieve a certain element, if you land on a non-empty location, how can you tell whether you arrived at the location for the desired key or on one occupied by the overspill from another one? *(Hint: describe precisely the low level structure of each entry in the table.)*

Exercise 43
How can you handle deletions from an open addressing table? What are the problems of the obvious naïve approach?

[28]Note that n might be $\gg N$.

The worst case cost of using a hash table can be dreadful. For instance, given some particular hash function, a malicious user could select keys so that they all hashed to the same value. But, on average, things work pretty well. If the number of items stored is much smaller than the size of the hash table, then both adding and retrieving data should have $\Theta(1)$ (constant) cost. The analysis of expected costs for tables that have a realistic load is of course more complex.

4.6.1 A short note on terminology

Confusingly, some people (not us) use the terms "open hashing" to refer to "chaining" and "closed hashing" to refer to "open addressing"—be ready for it. But we shall instead consistently use the terminology in CLRS3.

4.6.2 Probing sequences for open addressing

Many strategies exist for determining the sequence of slots to visit until a free one is found. We may describe the probe sequence as a function of the key k and of the attempt number j (in the range from 0 to $N-1$). Using a Java-like pseudocode to show argument types:

```
int probe(Key k, int j);
// BEHAVIOUR: return the array index to be probed at attempt <j>
// for key <k>.
```

So as not to waste slots, the curried function obtained by fixing the key to any constant value must be a permutation of $0, \ldots, N-1$ (the range of indices of the array). In other words, we would not like probe sequences that, for some keys, failed to explore some slots.

Linear probing This easy probing function just returns $h(k) + j \bmod N$. In other words, at every new attempt, try the next cell in sequence. It is always a permutation. Linear probing is simple to understand and implement but it leads to *primary clustering*: many failed attempts hit the same slot *and* spill over to the same follow-up slots. The result is longer and longer runs of occupied slots, increasing search time.

Quadratic probing With quadratic probing you return $h(k)+cj+dj^2 \bmod N$ for some constants c and d. This works much better than linear probing, provided that c and d are chosen appropriately. However it still leads to *secondary clustering* because any two keys that hash to the same value will yield the same probing sequence.

Double hashing With double hashing the probing sequence is $h_1(k)+j \cdot h_2(k) \bmod N$, using two different hash functions h_1 and h_2. As a consequence, even keys that hash to the same value (under h_1) are in fact assigned different probing sequences. It is the best of the three methods in terms of spreading the probes across all slots, but of course each access costs an extra hash function computation.

4.7 Priority queues and heaps

If we concentrate on the operations `set()`, `min()` and `delete()`, subject to the extra condition that the only item we ever delete will be the one just identified as the minimum one in our set, then the data structure we have is known as a **priority queue**. As the name says, this data structure is useful to keep track of a dynamic set of "clients" (e.g. operating system processes, or graph edges), each keyed with its priority, and have the highest-priority one always be promoted to the front of the queue, regardless of when it joined. Another operation that this structure must support is the "promotion" of an item to a more favourable position in the queue, which is equivalent to rewriting the key of the item.

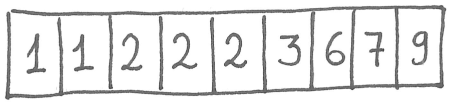
A priority queue is thus a data structure that holds a dynamic set of items and offers convenient access to the item with highest priority[29], as well as facilities for extracting that item, inserting new items and promoting a given item to a priority higher than its current one.

```
ADT PriorityQueue {
  void insert(Item x);
  // BEHAVIOUR: add item <x> to the queue.

  Item first(); // equivalent to min()
  // BEHAVIOUR: return the item with the smallest key (without
  // removing it from the queue).

  Item extractMin(); // equivalent to delete(), with restriction
  // BEHAVIOUR: return the item with the smallest key and remove it
  // from the queue.

  void decreaseKey(Item x, Key new);
  // PRECONDITION: new < x.key
  // PRECONDITION: Item <x> is already in the queue.
  // POSTCONDITION: x.key == new
  // BEHAVIOUR: change the key of the designated item to the designated
  // value, thereby increasing the item's priority (while of course
  // preserving the invariants of the data structure).

  void delete(Item x);
  // PRECONDITION: item <x> is already in the queue.
  // BEHAVIOUR: remove item <x> from the queue.
  // IMPLEMENTATION: make <x> the new minimum by calling decreaseKey with
  // a value (conceptually: minus infinity) smaller than any in the queue;
  // then extract the minimum and discard it.
}

ADT Item {
  // A total order is defined on the keys.
  Key k;
```

[29]"Highest priority" by convention means "earliest in the sorting order" and therefore "numerically smallest" in case of integers. Priority 1 is higher than priority 3.

```
  Payload p;
}
```

As for implementation, you could simply use a sorted array, but you'd have to keep the array sorted at every operation, for example with one pass of bubble-sort, which gives linear time costs for any operations that change the priority queue.



Exercise 45
Why do we claim that keeping the sorted-array priority queue sorted using bubble sort has linear costs? Wasn't bubble sort quadratic?

Operation	Cost with sorted array
creation of empty queue	$O(1)$
`first()`	$O(1)$
`insert()`	$O(n)$
`extractMin()`	$O(n)$
`decreaseKey()`	$O(n)$
`delete()`	$O(n)$

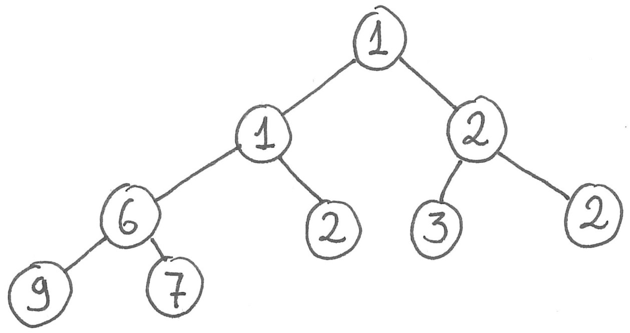
But we can do better than this.

4.7.1 Binary heaps

A good representation for a priority queue is the binary heap, which is the data structure implicitly used in the heapsort algorithm[30] of section 2.12. It is a clever yet comparatively simple construction that allows you to read out the highest priority item in constant time cost (without removing it from the queue) and lets you achieve $O(\lg n)$ costs for all other priority queue operations.

A min-heap is a binary tree that satisfies two additional invariants: it is "almost full" (i.e. all its levels except perhaps the lowest have the maximum number of nodes, and the lowest level is filled left-to-right) and it obeys the "heap property" whereby each node has a key less than or equal to those of its children.

[30]Except that heapsort uses a max-heap and here we use a min-heap.



As a consequence of the heap property, the root of the tree is the smallest element. Therefore, to read out the highest priority item, just look at the root (constant cost). To insert an item, add it at the end of the heap and let it bubble up (following parent pointers) to a position where it no longer violates the heap property (max number of steps: proportional to the height of the tree). To extract the root, read it out, then replace it with the element at the end of the heap, letting the latter sink down until it no longer violates the heap property (again the max number of steps is proportional to the height of the tree). To reposition an item after decreasing its key, let it bubble up towards the root (again in no more steps than the height of the tree, within a constant factor).

Since the tree is balanced (by construction, because it is always "almost full"), its height never exceeds $O(\lg n)$, which is therefore the asymptotic complexity bound on all the priority queue operations that alter the tree.

Operation	Cost with binary min-heap
creation of empty heap	$O(1)$
`first()`	$O(1)$
`insert()`	$O(\lg n)$
`extractMin()`	$O(\lg n)$
`decreaseKey()`	$O(\lg n)$
`delete()`	$O(\lg n)$

These logarithmic costs represent good value and the binary heap, which is simple to code and compact to store[31], is therefore a good choice, in many cases, for implementing a priority queue.

4.7.2 Binomial heaps

For some applications you might need to merge two priority queues (each with at most n elements) into a larger one. With a binary heap, a trivial solution is to extract each of the elements from the smaller queue and insert them into the other, at a total cost bounded by $O(n \lg n)$. A smarter solution is to concatenate the two underlying arrays and then heapify the result in $O(n)$ time.

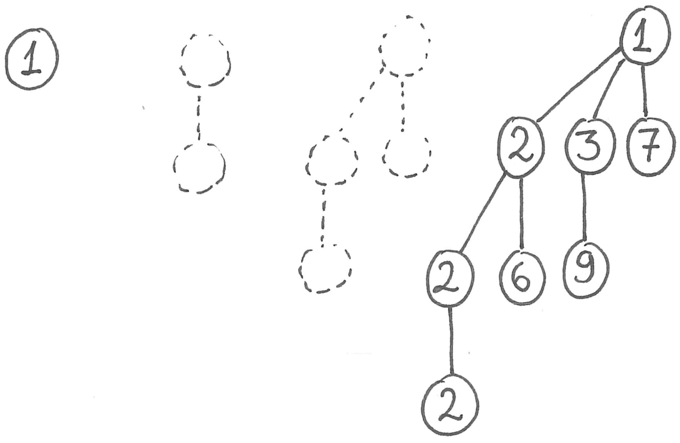
A more complex implementation of the priority queue is the binomial heap, whose main additional advantage is that it allows you to merge two priority queues, still at a time cost not exceeding $O(\lg n)$.

[31] The array representation does not even require any extra space for pointers.

Exercise 46
As a comparison, what is the most efficient algorithm you can think of to merge two binary heaps? What is its complexity?

```
ADT BinomialHeap extends PriorityQueue {
  void merge(BinomialHeap h);
  // BEHAVIOUR: combine the current heap with the supplied heap h. In
  // the process, make the supplied heap h empty and incorporate all its
  // elements into the current heap.
}
```

A binomial heap is a forest of binomial trees, with special properties detailed below.



A **binomial tree** (not heap) **of order 0** is a single node, containing one `Item`.

A **binomial tree of order** k is a tree obtained by combining two binomial trees of order $k-1$, by appending one of the trees to the root of the other as the (new) leftmost child[32]. By induction, it contains 2^k nodes (since the number of nodes doubles at each new order). By induction, the number of child subtrees of the root of the tree (known as the *degree* of the tree) is k, the same as the tree's order (since at each new order the tree gains one more child). By induction, the height of the tree is also k, again same as the tree's order (since at each new order the tree grows taller by one level, because the new child is as tall as the previous order's tree and is shifted down by one).

A **binomial heap** is a collection of binomial trees (at most one for each tree order), sorted by increasing size, each obeying the "heap property" by which each node has a key less than or equal to those of its children. If the heap contains n nodes, it contains $O(\lg n)$ binomial trees and the largest of those trees[33] has degree $O(\lg n)$.

[32]Note that a binomial tree is not a binary tree: each node can have an arbitrary number of children. Indeed, by "unrolling" the recursive definition above, you can derive an equivalent one that says that a tree of order k consists of a root node with k children that are, respectively, binomial trees of all the orders from $k-1$ down to 0.

[33]And hence *a fortiori* also each of the trees in the heap.

Exercise 47
Draw a binomial tree of order 4.

Exercise 48
Give proofs of each of the stated properties of binomial trees (trivial) and heaps (harder until you read the next paragraph—try before doing so).

The following property is neat: since the number of nodes and even the *shape* of the binomial tree of order k is completely determined a priori, and since each binomial heap has at most one binomial tree for any given order, then, given the number n of nodes of a binomial heap, one can immediately deduce the orders of the binomial trees contained in the heap just by looking at the "1" digits in the binary representation of n. For example, if a binomial heap has 13 nodes (binary $1101 = 2^3 + 2^2 + 2^0$), then the heap must contain a binomial tree of order 3, one of order 2 and one of order 0—just so as to be able to hold precisely 13 nodes.

The operations that the binomial heap data structure provides are implemented as follows.

`first()` To find the element with the smallest key in the whole binomial heap, scan the roots of all the binomial trees in the heap, at cost $O(\lg n)$ since there are that many trees.

`extractMin()` To extract the element with the smallest key, which is necessarily a root, first find it, as above, at cost $O(\lg n)$; then cut it out from its tree. Its children now form a forest of binomial trees of smaller orders, already sorted by decreasing size. Reverse this list of trees[34] and you have another binomial heap. Merge this heap with what remains of the original one. Since the merge operation itself (*q.v.*) costs $O(\lg n)$, this is also the total cost of extracting the minimum.

`merge()` To merge two binomial heaps, examine their trees by increasing tree order and combine them following a procedure similar to the one used during binary addition with carry with a chain of full adders.

> "binary addition" procedure. Start from order 0 and go up. At each position, say that for tree order j, consider up to three inputs: the tree of order j of the first heap, if any; the tree of order j of the second heap, if any; and the "carry" from order $j-1$, if any. Produce two outputs: one tree of order j (or none) as the result for order j, and one tree of order $j+1$ (or none) as the carry from order j to order $j+1$. All these inputs and outputs are either empty or they are binomial trees. If all inputs are

[34]An operation linear in the number of child trees of the root that was just cut off. Since the degree of a binomial tree of order k is k, and the number of nodes in the tree is 2^k, the number of child trees of the cut-off root is bounded by $O(\lg n)$.

empty, all outputs are too. If exactly one of the three inputs is non-empty, that tree becomes the result for order j, and the carry is empty. If exactly two inputs are non-empty, combine them to form a tree of order $j+1$ by appending the tree with the larger root to the other; this becomes the carry, and the result for order j is empty. If three inputs are non-empty, two of them are combined as above to become the carry towards order $j+1$ and the third becomes the result for order j.

The number of trees in each of the two binomial heaps to be merged is bounded by $O(\lg n)$ (where by n we indicate the total number of nodes in both heaps together) and the number of elementary operations to be performed for each tree order is bounded by a constant. Therefore, the total cost of the merge operation is $O(\lg n)$.

`insert()` To insert a new element, consider it as a binomial heap with only one tree with only one node and merge it as above, at cost $O(\lg n)$.

`decreaseKey()` To decrease the key of an item, proceed as in the case of a normal binary heap within the binomial tree to which the item belongs, at cost no greater than $O(\lg n)$ which bounds the height of that tree.

Operation	Cost with binomial heap
creation of empty heap	$O(1)$
`first()`	$O(\lg n)$
`insert()`	$O(\lg n)$
`extractMin()`	$O(\lg n)$
`decreaseKey()`	$O(\lg n)$
`delete()`	$O(\lg n)$
`merge()`	$O(\lg n)$

Although the programming complexity is greater than for the binary heap, these logarithmic costs represent good value and therefore implementing a priority queue with a binomial heap is a good choice for applications where an efficient merge operation is required. If however there is no need for efficient merging, then the binary heap is less complex and somewhat faster.

Having said that, when the cost of an algorithm is *dominated* by specific priority queue operations, and where very large data sets are involved, as will be the case for some of the graph algorithms of chapter 6 when applied to a country's road network, or to the Web, then the search for even more efficient implementations is justified. We shall describe even more efficient (albeit much more complicated) priority queue implementations in the next chapter. If your analysis shows that the performance of your algorithm is limited by that of your priority queue, you may be able to improve asymptotic complexity by switching to a Fibonacci heap or a van Emde Boas tree. Since achieving the best possible computing times may occasionally rely on the performance of data structures as elaborate as these, it is important at least to know that they exist and where full details are documented. But be aware that the constant hidden by the big-O notation can be quite large.

Chapter 5

Advanced data structures

Chapter contents

Amortized analysis: aggregate analysis, potential method. Fibonacci heaps. Van Emde Boas trees. Disjoint sets. Amortized analysis. Binomial heaps. Fibonacci heaps. van Emde Boas trees. Disjoint sets.
Expected coverage: about 5 lectures.

This chapter discusses some advanced data structures that will be used to improve the performance of some of the graph algorithms to be studied in chapter 6. We also introduce amortized analysis as a more elaborate way of computing the asymptotic complexity in cases where worst-case analysis of each operation taken individually would give a needlessly pessimistic and insufficiently tight bound.

Note that these advanced data structures do not generally offer support for finding an item given its key; it is tacitly assumed that the calling client program will maintain pointers to the nodes of the data structure it uses so that, when the client invokes a method on a specific node, the data structure already knows where that node is, as opposed to having to look for it. Therefore, when a method accepts or returns a parameter such as `Item x` in the Abstract Data Type description, what we implicitly mean is that, at the implementation level, `x` is a pointer to the node holding that item *within the data structure.*

5.1 Amortized analysis

Textbook

Study chapter 17 in CLRS3.

Some advanced data structures support operations that are ordinarily fast but occasionally very slow, depending on the current state of the data structure. If we performed our usual worst-case complexity analysis, such operations would have to be rated according to their very slow worst case every time they are invoked. However, since we know that they *usually* run much faster than that, we might obtain a correct but needlessly pessimistic bound. Amortized analysis takes the viewpoint that the cost of these occasional expensive operations might in some cases be spread, for accounting purposes, among the more frequent cheap ones, yielding a tighter and more useful bound.

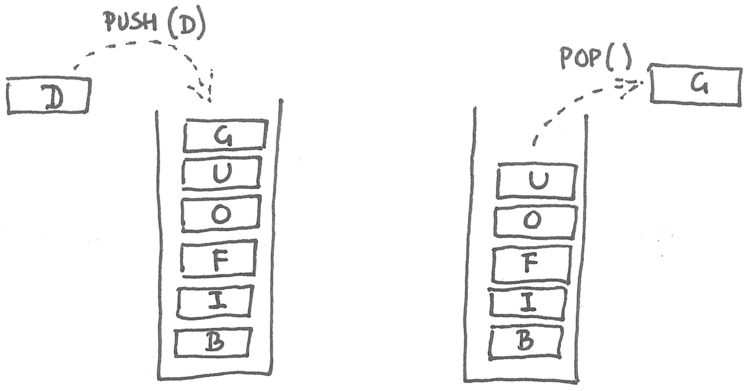
It is important to understand that amortized analysis *still* gives a worst-case bound (it is *not* a probabilistic estimate of the average case), but one that only applies to a sequence of operations over the lifetime of the data structure. Individual operations may exceed the stated amortized cost but the amortized bound will be correct for the whole sequence.

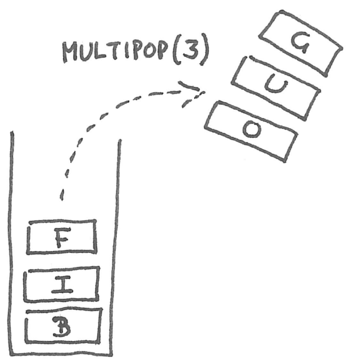
We introduce two ways of performing amortized analysis: aggregate analysis and the potential method.

5.1.1 Aggregate analysis

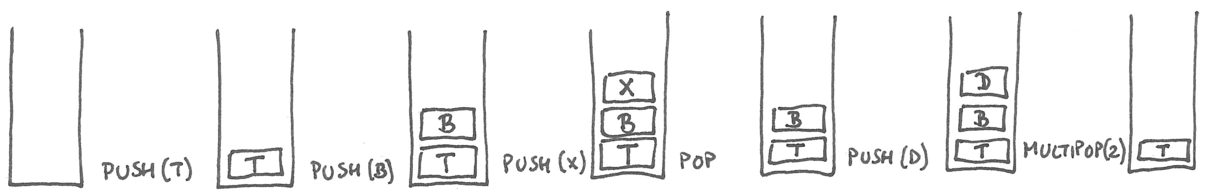
In aggregate analysis the general idea is to take a sequence of operations, compute its total cost and then divide it by the number of operations in the sequence to obtain the amortized cost of each. All operations are rated at the same cost even though in reality some of them were cheaper and some were more expensive. An example will hopefully clarify.

Let's assume you have a stack data structure whose methods are the usual `void push(Item x)` and `Item pop()`, plus a special `void multipop(int n)` that removes and discards the top n elements from the stack (or however many are available if the stack contains fewer than n). What is the cost of a sequence of n operations, each of which could be `push`, `pop` or `multipop`, on an initially empty stack? The "traditional" worst-case analysis would tell us that, while `push` and `pop` have constant cost, a single `multipop` could cost up to n (since there could be up to n elements in the stack) and therefore the worst-case costs are $O(n^2)$.





This asymptotic bound is correct but it is not tight. It is easy to see that, since each item must be pushed before it is popped (whether on its own or as part of a `multipop`), the combined number of all elementary popping operations (whether from `pop` or in bursts from `multipop`) cannot exceed the number of pushing operations, in turn bounded by n which is the total number of operations in the sequence. Therefore aggregate analysis tells us that a sequence of n operations can cost at most $O(n)$ and that therefore the *amortized* cost of each of the n operations is $O(1)$.



The example should make it clear that, if the amortized cost of an operation is low (e.g. the amortized cost of `multipop` is $O(1)$), this does *not* mean that all individual calls of that operation will be bounded by that low constant cost[1]. It does however mean that, in a sequence of n operations, the *overall* cost cannot build up to more than n times the (low) amortized cost of the individual operations.

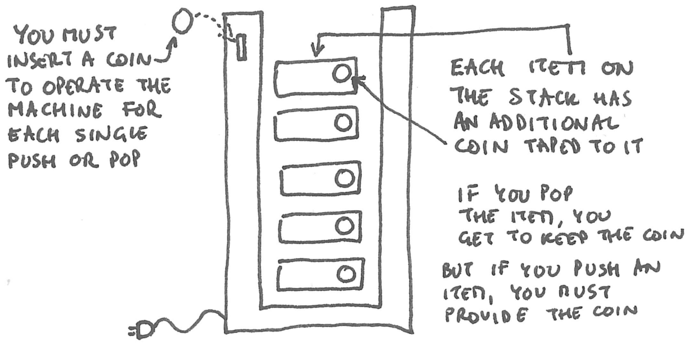
[1]Indeed it is still true that the worst-case non-amortized cost of a single `multipop` is $O(n)$, not $O(1)$.

5.1.2 The potential method

While aggregate analysis always attributes the same share of the cost to each of the operations in the sequence, the potential method can be more specific: it can give different amortized costs for different operations in the sequence.

The potential method works on the intuitive idea that, in order to compensate for the occasional expensive runs of some operations, if we are going to prove that their amortized cost is in fact lower than their worst-case cost when taken individually, we might "save" some "currency" somewhere, for the purpose of "repaying" the excessive costs when needed and still remain within the budget of the (lower) amortized cost when the overall sequence of operations is taken into account.

The "currency" is a totally fictitious entity that is not actually paid to anyone; it is just a way of accounting for the fact that, if an operation is frugal, that is to say it actually costs less than advertised, the "savings" can be used to "cover up" for a later operation that costs more than advertised. If we do that with savings that accumulate over many frugal operations, we may be able to cover up so well for occasional expensive operations that we lower their asymptotic complexity.


YOU MUST INSERT A COIN TO OPERATE THE MACHINE FOR EACH SINGLE PUSH OR POP
EACH ITEM ON THE STACK HAS AN ADDITIONAL COIN TAPED TO IT
IF YOU POP THE ITEM, YOU GET TO KEEP THE COIN
BUT IF YOU PUSH AN ITEM, YOU MUST PROVIDE THE COIN

Let's revisit the stack example. Let's say the cost of one elementary `push` or `pop` is one currency unit[2]. Imagine this currency unit as a gold coin, if you wish. To push a plate on the stack in the cafeteria, you must pay a gold coin (insert coin into machine, whatever). Same for popping a plate from the stack. For a `multipop` of 4 plates, you'd have to pay 4 coins. Now, imagine that at each push you actually sacrifice *two* gold coins: one to operate the machine and the other as a saving—you tape the second gold coin to the plate before pushing it on the stack. If you do that, the cost of a simple push becomes two currency units instead of one. Interestingly, that's still $O(1)$, because the constant factor is absorbed by the big-O notation. But what happens to `pop`? You pay one coin for operating the machine and taking off the plate, but you receive one coin that had been taped to the plate when it was pushed. So you end up doing `pop` at zero cost! And what about `multipop`? Amazingly, there too, even if you pop four plates (and therefore must pay four coins to operate the machine four times), you get back that many coins from the plates themselves, so you end up paying nothing in that case as well. Therefore, with the potential method, the amortized cost of `push` is $O(1)$ while the amortized cost of `pop` and

[2]That's another way to say that these elementary operations have constant cost.

`multipop` is zero[3]. Note that it is still the case, as before, that a sequence of n operations will cost no more than $O(n)$.

The point of the potential method is to devise an amortized cost structure that, despite being somewhat arbitrary, can be proved always to be an upper bound for the actual costs incurred, *for any possible sequence* of operations. To ensure that, it is essential to be always in credit.

If we call $\hat{c}_i$ the amortized cost of operation i, c_i its real cost and Φ_i the potential stored in the data structure before operation i, then the amortized cost (two coins to push one plate) is the same as the real cost (one coin to operate the machine) plus the increase in the potential of the data structure (one more coin deposited in the stack, taped to the plate):

$$\hat{c}_i = c_i + \Delta\Phi_i = c_i + \Phi_{i+1} - \Phi_i.$$

For a sequence of operations for i from 0 to $n-1$, the total amortized cost is:

$$\sum_{i=0}^{n-1} \hat{c}_i = \sum_{i=0}^{n-1} (c_i + \Delta\Phi_i) = \sum_{i=0}^{n-1} c_i + \Phi_n - \Phi_0.$$

If you want to be able to claim that the amortized cost is an upper bound for the real cost, you must be able to say that

$$\sum_{i=0}^{n-1} \hat{c}_i \geq \sum_{i=0}^{n-1} c_i$$

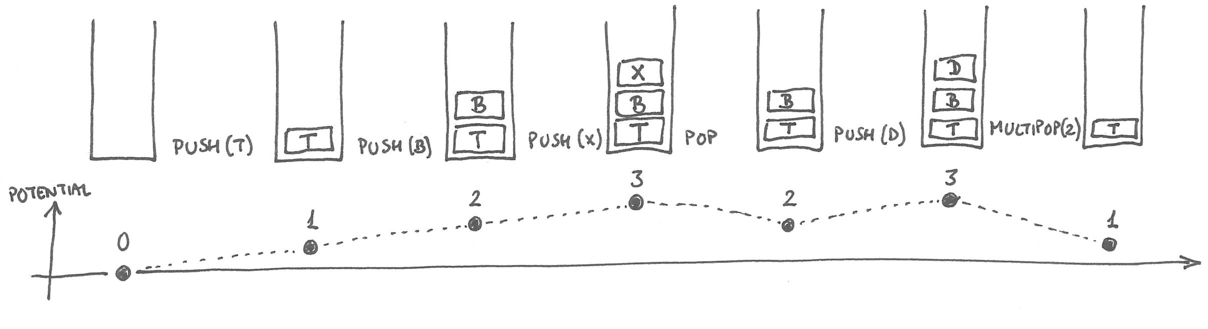
which, when you subtract the previous equation from it, becomes

$$0 \geq -\Phi_n + \Phi_0$$

and therefore

$$\Phi_n \geq \Phi_0.$$

In other words, since the above must hold for any sequence and therefore for any n, the potential of the data structure must never at any time go below what it was initially. We may simplify the expression by arbitrarily setting $\Phi_0 = 0$ when the initially empty structure is created and saying that the potential must be nonnegative at all times.



[3]There is a subtlety here related to the cost of doing `pop` or `multipop` on an empty stack; if we wanted to account for the cost of these failed calls, we'd have to be more meticulous in the above, and we'd end up with $O(1)$ amortized costs for all three operations.

In the example, the potential stored in the stack data structure was represented by the coins on the plates—one currency unit per plate. So the "potential function" Φ of the data structure was essentially the number of plates on the stack, which can never be negative. Choosing a potential function that starts at zero and can never become negative will guarantee that your real costs will never exceed your amortized costs for *any* possible sequence of operations. This is what allows you to take the amortized costs as valid asymptotic bounds for the real costs.

5.2 Fibonacci heaps

Textbook

Study chapter 19 in CLRS3.

The Fibonacci heap is yet another implementation of the priority queue, also featuring a merge facility like the binomial heap. In exchange for additional coding complexity and occasional delays, it offers an amazing *constant amortized cost* for all priority queue operations that don't remove nodes from the queue, and logarithmic costs for those that do. It is a "lazy" data structure in which many operations are performed very quickly but in which other less frequently used operations require a major and potentially lengthy clean-up and rearrangement of the internals of the data structure.

Using the Fibonacci heap may be advantageous, compared to a binomial or a binary heap, in cases where we expect the number of cheap operations to be asymptotically dominant over that of expensive operations.

It must however be noted that the additional programming complexity required by Fibonacci heaps means that the big-O notation hides a fairly large constant; this, and the extra implementation effort required, means that the move to a Fibonacci heap is only justified when the priority queues to be processed are really large.

5.2.1 Historical motivation and design criteria

The Fibonacci heap was created out of a desire to improve the asymptotic running time of the Dijkstra algorithm (section 6.4.2) for finding shortest paths from a single source on a weighted directed graph[4]. The limiting factor on the running time of the Dijkstra algorithm is the efficiency of the priority queue used to hold the vertices. The overall cost of Dijkstra is dominated by $|V|$ times the cost of `extractMin()` plus $|E|$ times the cost of `decreaseKey()`, where V and E are the sets of vertices and edges of the graph[5]. With a standard binary heap implementation for the priority queue, each of these individual costs is $O(\lg V)$; so the overall cost of Dijkstra is $O(V \lg V + E \lg V)$ which, on a connected graph, becomes $O(E \lg V)$ because $|E| \geq |V|$. This highlights that the critical operation is

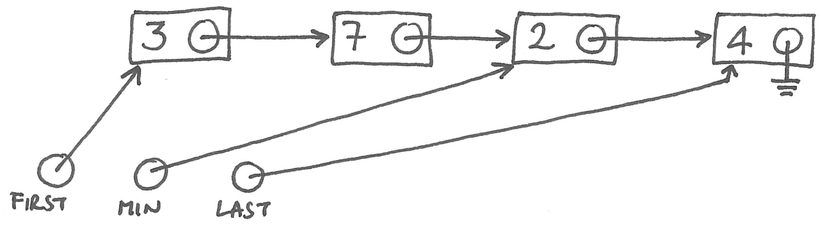
[4]This subsection contains forward references to some graph algorithms to be studied later and may be a little hard to understand fully until then. If so, please don't worry and come back to it later. Or follow the references and read ahead.

[5]See the remark on notation at the end of subsection 6.1.2 on page 110.

`decreaseKey()`: if we could reduce its cost without increasing that of the other operations, we would reduce the overall cost of Dijkstra. The development of the Fibonacci heap came out of this insight. It led to a data structure where, in amortized terms, `extractMin()` still costs $O(\lg V)$ but `decreaseKey()`, which is performed more frequently, has only constant cost. The overall amortized cost of Dijkstra then becomes $O(V \lg V + E)$ which is a definite improvement over $O(E \lg V)$. The same arguments bring an improvement from $O(E \lg V)$ to $O(V \lg V + E)$ to Prim's algorithm too (section 6.3.2).

5.2.2 Core ideas

One of the main ideas behind the Fibonacci heap is to use a lazy data structure in which many operations have constant cost because they do only what's necessary to compute the answer and then (unlike what happens with binomial heaps) return immediately without performing any tidying up. For example, in a binomial heap, both `insert()` and `merge()` require the elaborate binary-addition-like operation in which trees of the same degree are combined to form a tree of higher degree[6], possibly requiring further recombination with another existing tree and so on. Since each combining step takes constant cost, the `insert()` and `merge()` methods each take time bounded by the number of trees in the longest of the two binomial heaps (roughly speaking), which is $O(\lg n)$. We might instead imagine a lazy data structure (no tidying up after each insertion) in which the nodes are simply held in a linked list, with two extra pointers to indicate respectively the node with the smallest key (`min`) and the last node in the list. There, `insert()` and `merge()` would each only have constant cost: splicing in the new vertex or sublist at the front, and updating the `min` pointer after a single comparison. This gives $O(1)$ instead of $O(\lg n)$ for these two operations.

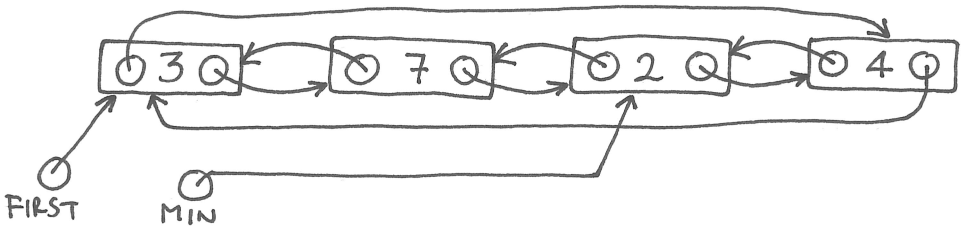


In a binomial heap, returning the smallest value without removing it from the heap, as done by `first()`, requires a linear search among the roots of the trees, which costs $O(\lg n)$ since that's the bound on the number of trees. In our linked list data structure we just return the item pointed by `min`, at cost $O(1)$. The `decreaseKey()` method is equally cheap: you decrease the key of the designated node and, after one comparison, update the `min` pointer if necessary. Cost is $O(1)$ as opposed to the $O(\lg n)$ of the binomial heap where the key might have to navigate up the tree towards the root.

There is a catch, of course: in our lazy data structure as described so far, we end up paying $O(n)$ for `extractMin()`. The extraction itself is not the real problem: yes, it

[6]In the original 1987 paper by Fredman and Tarjan that introduced Fibonacci trees, the number of children of a node is indicated as its *rank*, not its *degree*. We use *degree* in these notes to avoid confusion with a different definition of the term *rank* that is used in the context of disjoint sets (section 5.4), but be aware that in the literature you may find *rank* to mean "number of children".

would require $O(n)$ to find, via sequential traversal, the predecessor of the node to be removed, but this can easily be brought back to $O(1)$ by using a doubly-linked list instead of a regular list, at no increase in asymptotic cost for any of the other previously described operations.



The real, unavoidable cost is finding the second-smallest item (the new minimum after the removal) so as to be able to update the `min` pointer; for the lazy data structure this requires scanning the whole list, which costs $O(n)$. This negates all the benefits of making the other operations $O(1)$ because for example Dijkstra would end up costing $O(V^2 + E)$. On sparse graphs where $|E| = O(V)$, the performance of Dijkstra using the list-based lazy structure is $O(V^2)$ which is much worse than the $O(V \lg V)$ achieved with a binomial or binary heap[7].

The trick, then, is to perform some moderate amount of tidying up along the way so that searching for the new minimum will cost less than $O(n)$. The strategy used by the Fibonacci heap is in some sense a halfway-house between the flat list and the binomial heap. As far as `insert()`, `merge()` and `first()` are concerned, we operate similarly to what happens with the flat list[8]. Whenever we perform `extractMin()`, though, after having extracted the minimum we rearrange all the remaining vertices into something very similar to a binomial heap; and only then we look for the minimum, now at reduced cost. This may sound suspicious at first, given the expected cost of building an entire binomial heap; but we are only claiming a logarithmic cost for `extractMin()` *in amortized terms*: the actual cost will be higher.

Having conveyed these general intuitions, there is no point in discussing things any further without first explaining precisely how Fibonacci heaps are implemented. So let's now do that. Once that is understood, we'll get back to the analysis and compute precise amortized costs for all operations.

5.2.3 Implementation

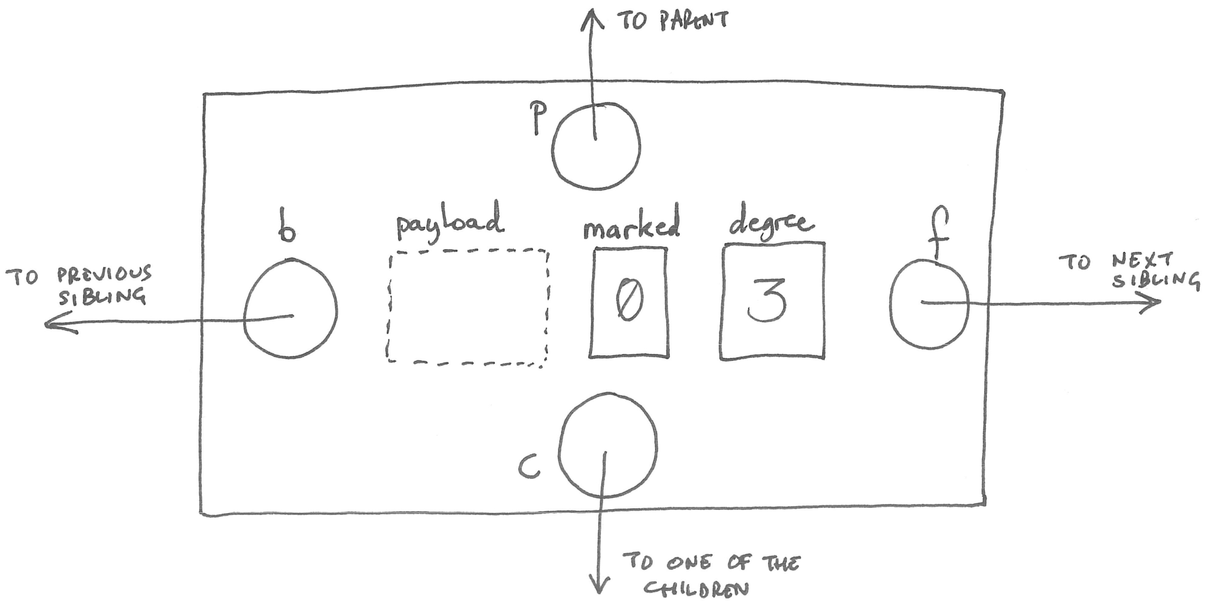
A Fibonacci heap is implemented as a forest of min-heap-ordered[9] trees. The heap also maintains a `min` pointer to the root with the smallest key among all the roots. The roots are linked together in a circular doubly-linked list, in no particular order. For each node, the children of the node are also linked together in a circular doubly-linked list, in no particular order. Each node has a pointer to its unique parent (`None` if the node is a

[7]It is however true that $O(V^2 + E)$ is better than $O(E \lg V)$ on dense graphs where $|E| = O(V^2)$: there, it becomes $O(V^2)$ versus $O(V^2 \lg V)$.

[8]Indeed a Fibonacci heap is indistinguishable from a flat doubly-linked list with `min` pointer so long as only `insert()`, `merge()` and `first()` are performed on it.

[9]Each node with a parent has a key not smaller than that of the parent.

root) and a pointer to one of its children (None if the node is a leaf; the other children are reachable via the circular list). Here is the layout of the FibNode data structure:

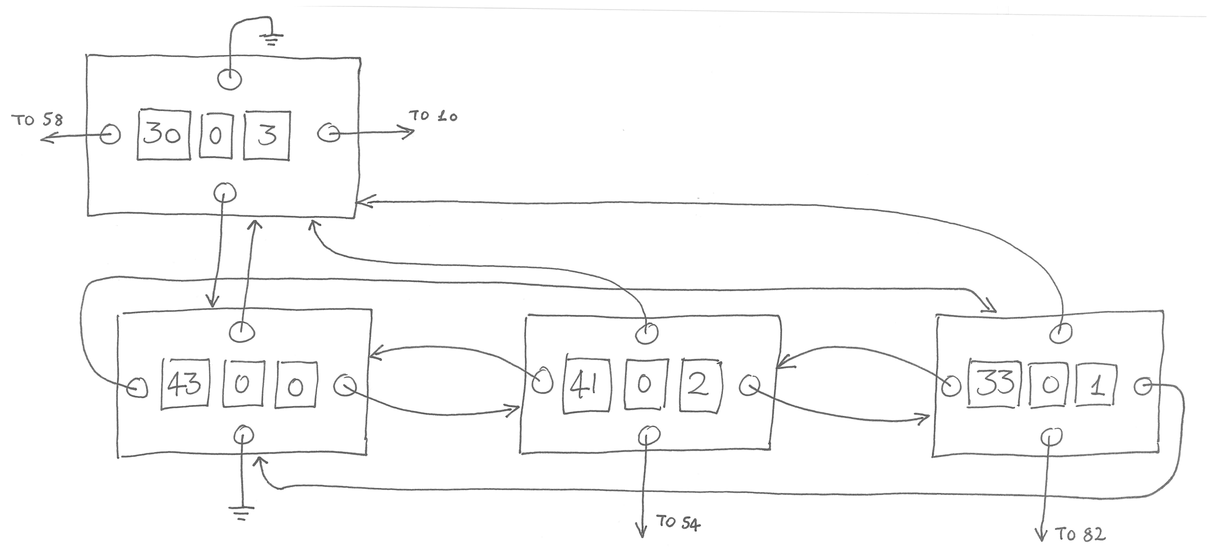


```
ADT FibNode extends Item {
  FibNode b; // back pointer; previous item in sibling list
  FibNode f; // forward pointer; next item in sibling list
  FibNode p; // parent pointer; direct ancestor
  FibNode c; // child pointer; other children reachable via sibling list
  Boolean marked; // set iff this node is not a root and has lost a child
  int degree; // number of children
}
```

Next is an ASCII-art example of a Fibonacci heap with 6 trees (not showing all the criss-crossing pointers, though). Each two-digit integer in the diagram stands for a full FibNode, of which it displays the payload. Horizontally-connected nodes, such as 43, 41 and 33, are siblings: they are actually members of a doubly-linked list.

```
06  58  30               10      40  44
 |       |                |       |
 +       +---+-------+    +---+   +
 |       |   |       |    |   |   |
77      43  41      33   70  32  50
             |       |        |
             +---+   +        +
             |   |   |        |
            54  66  82       51
                 |
                 +
                 |
                76
```

The following diagram magnifies a four-node portion of the previous Fibonacci heap, showing all the fields and pointers of each `FibNode`.



A **peculiar constraint** that the Fibonacci heap imposes on its trees, beyond that of being min-heap-ordered, is the following: after a node has been made a child of another node, it may lose at most one child before having to be made a root. In other words: while a root node (such as the one with key 30 in the above example) may lose arbitrarily many children without batting an eyelash, a non-root node (such as 41) may only lose at most *one* child node; after that, if it is to lose any other child nodes, it must be cut off from its tree and promoted to being a standalone root. The reason for this constraint will be explained during the asymptotic analysis: its ultimate purpose is to bound the time of `extractMin()` by ensuring that a node with n descendants (children, grandchildren etc) has a degree (number of children) bounded by $O(\lg n)$. The way the constraint is enforced is to have a boolean flag (the `marked` field) in each node, which is set as soon as the non-root node[10] loses its first child. When attempting to cut a child node from a non-root parent, if the parent is `marked` then the parent must be cut off as well and made a root (and unmarked), and so on recursively. This maintains the required invariant.

Exercise 49
Explain with an example why, if the "peculiar constraint" were not enforced, it would be possible for a node with n descendants to have more than $O(\lg n)$ children.

As is typical with priority queues, there is no efficient support for finding the node that holds a given key. Therefore the methods of a Fibonacci heap take already-created `FibNode`s as parameters and return `FibNode`s where appropriate. The calling code is responsible for allocating and deallocating such nodes and maintaining pointers to them.

[10]Note that, by line 5 of the `FibNode` ADT definition on page 88, a root node is never marked. And if a marked node is ever made into a root, it must then lose its mark.

`FibHeap constructor()`

Behaviour: Create an empty heap.
Implementation: Set `min` to `None` (constant cost).

`FibHeap constructor(FibNode n)`

Behaviour: Create a heap containing just node `n`.
Implementation: Make `min` point to this node and adjust `n`'s own pointers consistently with the fact that `n` is now the only node in the circular doubly-linked list of roots (constant cost)[11].

`void insert(FibNode n)`

Behaviour: Add node `n` to the heap.
Precondition: `n` is not in the heap.
Postcondition: The nodes now in this heap are those it had before plus `n`.
Implementation: create a heap with just `n` (constant cost) and merge it into the current heap (constant cost, *cfr* `merge()`).

`void merge(FibHeap h)`

Behaviour: Take another heap `h` and move all its nodes to this heap.
Postcondition: `h` is empty. The nodes now in this heap are those it had before plus those that used to be in `h`.
Implementation: Splice the circular list of roots of `h` into the circular list of roots of this heap (constant cost). Compare the two `min` and point to the winner (constant cost).

`FibNode first()`

Behaviour: Return (a pointer to) the node containing the minimum key, without removing that node from the heap.
Precondition: This heap is not empty.
Postcondition: This heap is exactly the same as it was before invoking the method.
Implementation: return `min` (constant cost).

`FibNode extractMin()`

Behaviour: Return (a pointer to) the node containing the minimum key, but also remove that node from the heap.
Precondition: This heap is not empty.
Postcondition: This heap contains the same nodes it had previously (though not necessarily in the same layout), minus the one that contained the minimum key.
Implementation: Cut off the root pointed by `min` from the circular list of roots of this heap (constant cost). Cut off this root from its children (cost proportional to the number of children, because you must reset the `p` pointer of each child). Splice the circular list of children into the circular list of roots (constant cost). Repeatedly perform the "linking step" while it is possible to do so: if any two trees in the circular list of roots have the same degree k, i.e. the same number of child subtrees, combine them into a single tree of degree $k+1$ by making the tree with the larger root key become a child of the other (cost to be discussed later). Set `min` to point to the root

[11] Conceptually this is a private constructor, used only by the `insert()` method and not accessible to external clients.

with the smallest key (cost proportional to the total number of trees now in the root list, also to be discussed later). Finally, at constant cost, return the (previous) minimum node that was cut off from the root list and from its children at the start of the operation. (Total cost to be discussed later.)

`void decreaseKey(FibNode n, int newKey)`

Behaviour: Decrease the key of node `n` to `newKey`.
Precondition: `n` is in this heap and `newKey` < `n.key`.
Postcondition: This heap contains the same nodes as it did previously (though not necessarily in the same layout), except that `n` now has a new key.
Implementation: Decrease `n.key` to the new value, then check if `n.key` is now smaller than the parent node's key. If it hasn't, the job is completed, at constant cost. If it has become smaller, the min-heap ordering has been violated. Restore it at constant cost by cutting off `n`, ensuring its `marked` flag is reset and making `n` into a new root, where it can have as low a key as it wants without disturbing the min-heap property. As you cut off the node, you must however obey the "peculiar constraint": if the node's parent is marked, it must also be cut off and unmarked (and its ancestors too, recursively, if necessary, causing a series of so-called *cascading cuts*). If the node's parent is not marked, and is not a root, it must be marked. (Cost of cascading cuts to be discussed later.)

`void delete(FibNode n)`

Behaviour: Remove the given node from the heap.
Precondition: `n` is in this heap.
Postcondition: This heap contains the same nodes as before this method was invoked, except for `n` which has been removed.
Implementation: As with any priority queue, you may implement `delete()` as a combination of `decreaseKey()` and `extractMin()`. Invoke `decreaseKey()` on node `n` with a `newKey` of $-\infty$, making that node the one with the smallest key in the heap. Then perform `extractMin()` to remove it. (Cost: the sum of the costs of `decreaseKey()` and `extractMin()`.)

5.2.4 Amortized analysis

Using the potential method we shall prove that `extractMin()` and `delete()` run in amortized $O(\lg n)$ cost and that all other operations, particularly `decreaseKey()`, run in amortized constant cost.

We define the potential function Φ of the Fibonacci heap as $t + 2m$ where t is the number of trees and m is the number of marked nodes. This means, using the previous metaphor, that we must pay, or rather deposit in the data structure, an extra gold coin whenever we create a new tree at the root level and we must deposit two gold coins whenever we mark a node; however we do get back one gold coin whenever we remove a tree and we get back two coins whenever we unmark a node. The value of a gold coin is taken to be sufficient to cover the highest of the constant costs identified in the preliminary analysis in the previous section.

Let's now review each of the methods and assess its amortized cost.

`FibHeap constructor()`

No trees and no nodes are created, so $\Phi = 0$. Amortized cost is equal to real cost, i.e. constant.

`FibHeap constructor(FibNode n)`

We make a new (singleton) tree, so we deposit an extra gold coin in it on top of the real cost. The amortized cost is still constant; it's actually higher than the real cost, but the difference is hidden by the big-O notation.

`void merge(FibHeap h)`

We rewrite a fixed number of pointers to splice together the two doubly-linked lists of roots. The roots of `h` already carry one gold coin each so there is no change in overall potential when we transfer them to this heap. The amortized cost is the same as the real cost, i.e. constant.

`void insert(FibNode n)`

Creating the singleton heap forces us to pay a gold coin but that's still only a constant cost. We just saw that merging has amortized constant cost and therefore here too the overall amortized cost is constant (though slightly higher than the real cost).

`FibNode first()`

The potential is not affected so the amortized cost is the same as the real cost: constant.

`FibNode extractMin()`

Now things start to get interesting. Cutting off the node pointed by `min` from the root list yields a gold coin which repays us of any one-off work done. If c is the degree (= number of children) of that minimum node, for each of the c children we must perform a constant amount of work to reset the `p` (parent) pointer to `None`, plus we must deposit a gold coin into the child node because we are creating a new root-level tree. So we are paying the equivalent of $2c$ gold coins for this part. Then we must do the linking step: if any two trees have the same degree, we join them into a single tree of degree one higher by making one the child of the other. This action may have to be repeated many times, depending on how many trees there are (however many there were before, minus one, plus c) *but* we don't worry about it because each join (whose real cost is a constant since it's a fixed number of pointer manipulations) pays for itself with the gold coin released by the tree that becomes a child of the other. So, in amortized terms, the whole linking step is free of charge. Next, finding the smallest root among the remaining trees after the linking step requires a linear search of all of the remaining trees. How many are there? heap with n nodes, call $d(n)$ the highest degree of any node in that heap; then, since the outcome of the linking step is that no two trees have the same degree, the number of trees is necessarily bounded by $d(n) + 1$ (the case in which we have the trees of all possible degrees from 0 to $d(n)$). Since $c \leq d(n)$ by definition of d, the overall amortized cost of `extractMin()`, including also the $2c$ mentioned previously, is $O(d(n))$. All that remains to do in order to support the original claim is therefore to prove that $d(n) = O(\lg n)$, which we shall do in section 5.2.5. The intuition is that we don't want any tree in a Fibonacci heap to grow "wide and shallow", otherwise the root will have too many children compared to the total number of nodes of the tree and the number of children will exceed $O(\lg n)$; and the "peculiar constraint" has precisely the effect of preventing "wide and shallow" trees. In summary, the amortized cost of `extractMin()` is $O(\lg n)$ even though the real cost may be much higher because of the hidden work involved in the linking step (which gets repaid by spending some of the gold coins that were stored in the data structure).

`void decreaseKey(FibNode n, int newKey)`

If the node doesn't need to move as a consequence of its key having been decreased, the potential is unchanged by the operation and therefore the amortized cost is the same as the real cost, i.e. constant. In the case where the node must be cut (because its key is now smaller than that of its parent, which would violate the min-heap property), let's assume that there are $p \geq 0$ marked ancestors above it that also need to be made into roots with cascading cuts. Each *marked* node that is cut releases two gold coins as it gets unmarked (remember roots are all unmarked). One coin covers for the cost of cutting and splicing, while the other coin is stored in the newly-created root. Therefore, any arbitrarily long chain of cascading cuts *of marked nodes* will exactly repay for itself in amortized terms. The only possibly non-marked node among those to be cut is `n` itself, the one whose key has been decreased. If `n` is unmarked (and it may or may not be) we must pay a constant cost for cutting and splicing it and then we must store a gold coin in the new tree root; furthermore, if the parent of `n` was unmarked and was not a root, we must mark it and store two coins in it, at a total worst-case cost of up to four coins (one stored in the new root, two in the newly-marked node, and one coin's worth to cover all the actual work of cutting, splicing and marking). This is still a constant with respect to n, therefore the overall amortized cost still won't exceed $O(1)$, even though the real cost may be much higher because of the hidden work involved in the cascading cuts.

`void delete(FibNode n)`

As previously noted, this is simply a two-step sequence of `decreaseKey()`, which costs $O(1)$ amortized, and `extractMin()`, which costs $O(\lg n)$ amortized; therefore in total this operation costs $O(\lg n)$ amortized.

The following table summarizes the amortized costs of all the methods of a Fibonacci heap: all methods have constant amortized cost except those that remove nodes from the heap, which have logarithmic cost.

Operation	Cost (amortized) with Fibonacci heap
creation of empty heap	$O(1)$
`first()`	$O(1)$
`insert()`	$O(1)$
`extractMin()`	$O(\lg n)$
`decreaseKey()`	$O(1)$
`delete()`	$O(\lg n)$
`merge()`	$O(1)$

5.2.5 Why Fibonacci?

To complete the complexity analysis of `extractMin()` we must prove that the maximum degree of any node in an n-node Fibonacci heap is bounded by $O(\lg n)$; if we do so, then we have shown that the whole `extractMin()` operation is $O(\lg n)$. The full story will also explain the reason for the Fibonacci name assigned to the data structure.

We want to prove that, by building Fibonacci heaps in the way we described (including in particular the "linking step" during `extractMin()` and the "peculiar constraint" enforced by node marking in `decreaseKey()`), if a node has k children then the number of nodes in the subtree rooted at that node is at least exponential in k. In other words we shall prove that, if a subtree contains n nodes (root of subtree included), the root of that subtree has at most $O(\lg n)$ children.

Consider a node x, with children, and consider its child nodes in the order in which they were attached to x, from earliest to latest. They could have only been attached during a linking step, which happens between two trees of the same degree. Call y the k-th child of x. When y was appended to x, x must have had at least $k-1$ children (even though it may have had more, since deleted) and so y itself must have had at least $k-1$ children itself at that time, because it had to be of x's degree in order to be linked to x. Since then, y may have lost at most one child but not more—otherwise, thanks to the peculiar constraint, it would have been cut off and made into a root. Therefore (thesis) the k-th child of x has at least $k-2$ children.

Now let's define the function $N(j)$ as the minimum possible number of nodes of a subtree rooted at a node x of degree j and let's seek a closed expression for it. If node x has degree 0, meaning no children, then the whole subtree contains just the original node x: $N(0) = 1$. If node x has degree 1 (one child), the minimum tree is one where that child has no children, so the tree only has these two nodes (x and its child) and $N(1) = 2$. The thesis above tells us how many children each of the children of x from the second onwards must have at least: the second node must have at least 0, the third node at least 1, the fourth node at least 2, the k-th node at least $k-2$. Therefore, assuming that each of these children is itself root of a subtree with the minimum possible number of nodes, the overall minimum number of nodes in the subtree rooted at x will be the sum of all these contributions:

$$N(j) = \overset{\text{root}}{1} + \overset{\text{child } 1}{1} + \overset{\text{child } 2}{N(2-2)} + \overset{\text{child } 3}{N(3-2)} + \ldots + \overset{\text{child } j}{N(j-2)} = 2 + \sum_{l=0}^{j-2} N(l). \tag{5.1}$$

From this we can prove inductively that

$$N(j) = F(j+2) \tag{5.2}$$

where $F(k)$ is the k-th Fibonacci number[12]. For the base of the induction, it's easy to verify that the relation holds for $j = 0$ and $j = 1$; for the inductive step, let's assume it

[12]The well-known Fibonacci sequence is defined as $F(0) = 0$, $F(1) = 1$ and $F(i+2) = F(i) + F(i+1)$. Each number is the sum of its two predecessors: 0, 1, 1, 2, 3, 5, 8, 13, 21... The sequence occurs in many contexts in nature and art and the ratio of two successive numbers converges to the irrational number $\phi = \frac{1+\sqrt{5}}{2}$, known as the golden ratio. Note that this ϕ is totally unrelated to the potential Φ introduced in section 5.1.2.

works for j and check whether it does for $j+1$:

$$\begin{aligned}
\text{LHS} &= N(j+1) = \\
&= 2 + \sum_{l=0}^{j-1} N(l) = \\
&= 2 + \sum_{l=0}^{j-2} N(l) + N(j-1) \overset{(5.1)}{=} \\
&= N(j) + N(j-1) \overset{(5.2)}{=} \\
&= F(j+2) + F(j+1) = \\
&= F(j+3) = \\
&= \text{RHS}
\end{aligned}$$

QED.

It works! So the number of nodes in a subtree of degree j is at least equal to the $(j+2)$-th Fibonacci number. Since the Fibonacci numbers grow exponentially (it can be proved that $F(j+2) \geq \phi^j$), we have that the total number of nodes in a subtree of degree j is exponential in j:

$$N(j) \geq \phi^j$$

and therefore, taking the log of both sides,

$$\log_\phi N(j) \geq j$$

which means that $j \leq K \lg N(j)$ for some constant K. So we have finally proved that, in a Fibonacci heap, if a subtree has n nodes, the root of the subtree has degree $O(\lg n)$.

5.3 van Emde Boas trees

Textbook

Study chapter 20 in CLRS3.

Just when you thought that the Fibonacci heaps were the asymptotically fastest and most elaborate way of implementing a priority queue, here comes another amazing data structure that goes even further—the van Emde Boas tree, or “vEB tree”.

The Fibonacci heap already has amortized constant cost for most of its operations and it’s hard to improve on that; but is still has amortized $O(\lg n)$ costs for `extractMin()` and `delete()`. The vEB tree manages to bring those costs down to an amazing $O(\lg \lg n)$! The three main trade-offs are:

1. *all* operations cost $O(\lg \lg n)$, even the ones that were $O(1)$ with the Fibonacci heap;
2. the vEB tree requires the keys to be (unique) integers of a specified maximum size in bits;

3. unlike the Fibonacci heap, the vEB tree does not offer an efficient merge operation.

The interface exposed by the vEB tree is essentially that of the ordered dynamic set[13]: `member()`, `insert()`, `delete()`, `min()`, `max()`, `pred()`, `succ()`. All these operations are performed in $O(\lg\lg n)$ time and, by combining them in the obvious way, one can also perform in $O(\lg\lg n)$ time the typical priority queue operations of `extractMin()`, `first()`, `decreaseKey()`.

Exercise 50
Assume that `insert()`, `delete()`, `min()`, `max()`, `pred()`, `succ()` all have complexity $O(f(n))$. Define `extractMin()`, `first()` and `decreaseKey()` in terms of the previous primitives, without exceeding $O(f(n))$ complexity.

But the vEB tree is a pretty elaborate data structure so we'll get to it in stages rather than describing the final product straight away. First, though, instead of speaking of n, the number of keys held in the priority queue, we'll speak of u, the size of the universe of keys, or in other words the number of *possible* keys in the data structure (for example $u = 2^{64}$, if each key is a 64-bit integer). The costs for the vEB tree, unlike those for the Fibonacci heap, are in fact a function of u (the number of distinct keys that the data structure could potentially hold), not of n (the number of keys actually stored in the data structure); so, to be more precise, with this definition we should have said in the above that the costs of the dynamic set operations supported by the vEB tree are $O(\lg\lg u)$.

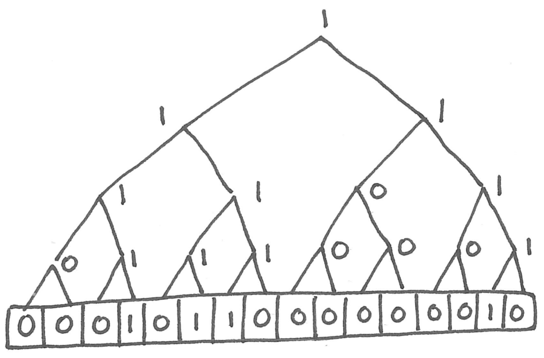
5.3.1 Array

The first attempt to speed up operations such as `insert()` and `delete()` is to use an array of bits of size u, with `A[k] == 1` if and only if the item whose key is k is in the data structure; then `insert()` and `delete()` will cost $O(1)$. But `min()`, `max()`, `pred()`, `succ()` will cost an unacceptable $O(u)$, because we'll have to scan the array linearly, skipping all the 0s, until we find the next 1.

5.3.2 Binary tree

We can augment the array with a binary tree whose leaves are the array entries. Each node of the tree is labelled with one "summary" bit for its subtree, which is 1 iff the subtree rooted there has any nonzero leaves. In other words, the bit value of each node is the OR of the values of its two children. This allows us to skip empty contiguous regions more efficiently. For example, to find the minimum, start from the root and follow the leftmost "1" pointers until you get to a leaf. The cost is the height of the tree, i.e. $O(\lg u)$.

[13]Except that we disregard satellite data because all that matters here is the keys themselves. Indeed, we don't even have a `search` method but just a `member` method that says whether a given key is in the data structure or not.

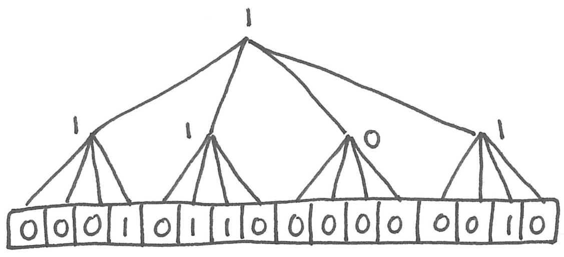


Finding the successor (the next cell with a "1") of a designated value[14] is slightly more elaborate. You go up until the up-arrow goes right and the other (right) child has a "1" summary[15]. Then you go down that right child of that ancestor and find the minimum of that subtree. The worst possible cost is twice the height of the tree, so again $O(\lg u)$.

Inserting a new leaf is easy: set it and all its ancestors to 1, again at cost $O(\lg u)$. Deleting is not quite symmetrical, though: you set the leaf to 0 but then, as you go up the tree, the value of each ancestor must be recomputed as the OR of its two children. The cost remains $O(\lg u)$, since the work per node is constant.

5.3.3 Two-level tree

In an attempt to speed up operations even further, we force the tree to have a constant height of just two levels (instead of a variable height of $\lg u$ levels). Each node of the tree, root included, must therefore have $\sqrt{u}$ children. Again, each node stores one bit which is the OR of all its children. The procedures to find the minimum (maximum) or the successor (predecessor) are conceptually the same as before, but because the tree has constant height the total cost is $O(\sqrt{u})$.



The procedures for inserting and deleting are also conceptually the same as before, but deletion is more expensive.

[14]Note that the successor of x can be sought regardless of whether x is or isn't a member of the data structure, i.e. without cell x necessarily having to be set to 1 itself.

[15]If you go up and reach the root without the up-arrow ever going right, you were already at the rightmost cell of the array so you have no successor. If the up-arrow eventually goes right but the right child has a "0" summary, then there are no bits set after your position and therefore you have no successor either.

Exercise 51
How much do insertion and deletion cost for the two-level tree? Why?

5.3.4 Proto-vEB tree

We continue with the idea of a tree of degree $\sqrt{u}$, but this time we make the data structure recursive: if u is the size of the universe of keys at a given level in the structure, at the next level down the structure shrinks by a factor of $\sqrt{u}$. Imagine for example that, at the top level, $u = 2^{64}$. Then the root node, to cater for $u = 2^{64}$ leaves, will have $\sqrt{u} = 2^{32}$ children, each with $\sqrt{u} = 2^{32}$ leaves. From the point of view of one of these children, the size of the universe of keys will be $u = 2^{32}$ (that's a new u for this level), so that node will in turn have $\sqrt{u} = 2^{16}$ children, each with $\sqrt{u} = 2^{16}$ leaves. And so forth until the base level where $u = 2$.

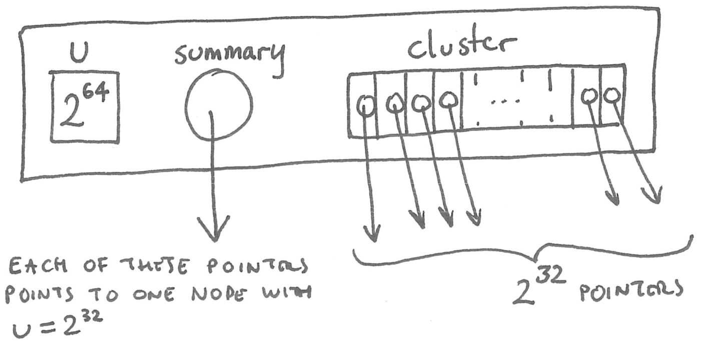
We also rearrange the tree slightly: previously, each node contained one summary bit with the OR of all its children. Now, although the same information is still there, we store it somewhere else. Node (cluster) n, of size u, has $\sqrt{u}$ children of size $\sqrt{u}$. It also has another (anomalous) child, called "summary", which itself is a proto-vEB node of size $\sqrt{u}$ (structurally identical to all the other children of n) but whose values are the indices of the children of n (clusters of size $\sqrt{u}$) that are not empty. And each node restarts numbering its own children from 0.

Each proto-vEB node with $u > 2$ thus contains the following three fields:

`u:` an integer giving the size of the universe of keys for this node; this node can store all integers from 0 to $u - 1$.

`summary:` a pointer to a proto-vEB node of size $\sqrt{u}$ containing one summary bit for each child of the current node.

`cluster[]:` an array with $\sqrt{u}$ cells, each being a pointer to a proto-vEB node of size $\sqrt{u}$.



At the bottom level, the proto-vEB node with $u = 2$ does not have the summary pointer and stores actual bits, rather than pointers, in the two cells of its cluster.

We assume for simplicity that $u = 2^{2^k}$ for some integer k, so that the square root of u is always an integer, at all levels in the tree. Another way of saying the same thing is that the number of bits of u is a power of 2 (at all levels) and that therefore it is always possible to split the bit string of a key in two halves of equal length. Given any key $x \in [0, u-1]$, if we call `high(x)`, or h, the top half of the binary representation of x and `low(x)`, or l, the bottom half[16], then x is stored in the h-th child of the root node and it is key number l within that node. (We use the function `index(h, l)` to join back the two halves: `index(high(x), low(x)) == x`.)

For example, if a proto-vEB tree of size $u = 2^4$ stores the key 13 (binary 1101), the root node (which has $\sqrt{u} = 2^2$ clusters with $\sqrt{u} = 2^2$ keys each) stores it in its cluster number 3 (binary 11, the most significant half of 1101) and at position 1 (binary 01) in that cluster. Note well how cluster number 3 has a size of 4, not 16, and thus only holds keys from 0 to 3. The key it holds is indeed 1, not 13, which wouldn't fit. You can only reconstruct the full value of the key if you look at the whole path: you can't read it off from the leaf. (Remember that as you do the following exercise.)

Exercise 52
Sketch a picture of a proto-vEB tree of size $u = 16$ representing the set 2, 3, 4, 5, 7, 14, 15.

Where did these weird ideas come from? Why are we doing this at all? The objective is to attain the $O(\lg\lg u)$ complexity bound. We can show that the recurrence

$$T(u) = T(\sqrt{u}) + O(1)$$

is solved by the desired

$$T(u) = O(\lg\lg u)$$

and this suggests that, to attain that recurrence, we should build a recursive structure where the size of each node is the square root of the size of the parent. Then if a recursive procedure for, say, finding the minimum, did on each level an amount of work proportional to the size of a node at that level, it would be described by the above recurrence and would achieve the desired running time.

Let's first show, informally, that the above recurrence is indeed solved by the given function. We use the substitution $u = 2^m$, implying $m = \lg u$.

$$\begin{aligned} T(u) &= T(\sqrt{u}) + O(1) \\ T(2^m) &= T(2^{\frac{m}{2}}) + O(1) \\ &= T(2^{\frac{m}{4}}) + 2O(1) \\ &= T(2^{\frac{m}{8}}) + 3O(1) \\ &= T(2^{\frac{m}{16}}) + 4O(1) \\ &= T(2^{\frac{m}{2^l}}) + lO(1) \end{aligned}$$

[16]Note also that, in reality, the functions `high()` and `low()` must take u as a second parameter, in order to know how many bits to chop off.

$$\begin{aligned}
&= T(2^{\frac{m}{m}}) + \lg m \cdot O(1) \\
&= T(2) + O(1) \lg m \\
&= O(\lg m) \\
&= O(\lg \lg u)
\end{aligned}$$

QED.

Let's now look specifically at finding the minimum.

```
0  def min(self):
1      """In this class method for a protoVEB object,
2      self is a protovEB node of size u.
3      Return the minimum key in self, or None if self holds no keys."""
4
5      if self.u == 2:
6          handle base case by brute force at constant cost
7      else:
8          minCluster = self.summary.min()
9          if minCluster == None:
10             return None
11         else:
12             h = minCluster
13             l = self.cluster[minCluster].min()
14             return index(h, l)
```

We see that the worst-case costs involve *two* recursive calls of `min()` from within itself (lines 8 and 13), on proto-vEB nodes of size $\sqrt{u}$. This yields the recurrence

$$T(u) = 2T(\sqrt{u}) + O(1)$$

which unfortunately solves to $T(u) = O(\lg u)$, not $O(\lg \lg u)$.

Exercise 53
Prove that the recurrence

$$T(u) = 2T(\sqrt{u}) + O(1)$$

has the solution

$$T(u) = O(\lg u).$$

Some of the other data structure methods have even worse performance. Consider for example the case of looking for the successor of a given key. As with the two-level tree, we go up until the ancestor node has a child to the right of us, and that child's summary bit is 1. Then we go down that child, all the way to the leaves, looking for the minimum. To obtain a recurrence for this operation, let's rephrase it as a top-down recursive procedure that matches the hierarchy of the data structure: the successor of k, if it exists, is either

k's successor within the cluster that contains k (in case k is not the maximum of its own cluster), or it's the minimum of the next cluster that contains anything.

```
0  def successor(self, k):
1      """In this method, self is a protovEB node of size u.
2      k is a key in the range 0..u-1.
3      Return the key that is the successor of k in self,
4      or None if there is none."""
5
6      if self.u == 2:
7          handle base case by brute force at constant cost
8      else:
9          h = high(k, self.u)
10         l = low(k, self.u)
11         successorInCluster = self.cluster[h].successor(l)
12         if successorInCluster == None:
13             nextCluster = self.summary.successor(h)
14             if nextCluster == None:
15                 return None
16             else:
17                 return nextCluster.min()
18         else:
19             return successorInCluster
```

Here we see that, in the worst case, the `successor()` method calls itself twice, in lines 11 and 13, each time on a node of size $\sqrt{u}$, but then also invokes `min()` on a node of size $\sqrt{u}$, in line 17. This gives us a recurrence of $T(u) = 2T(\sqrt{u}) + O(\lg \sqrt{u})$, where the factor of 2 in front of the T term comes from the two recursive invocations of `successor()` from within itself, while the $O(\lg \sqrt{u})$ is the cost of `min()` as derived above.

Exercise 54
Prove that the recurrence

$$T(u) = 2T(\sqrt{u}) + O(\lg \sqrt{u})$$

has the solution

$$T(u) = O(\lg u \lg \lg u).$$

Therefore, with the proto-vEB tree, computing the successor is asymptotically slower than computing the minimum.

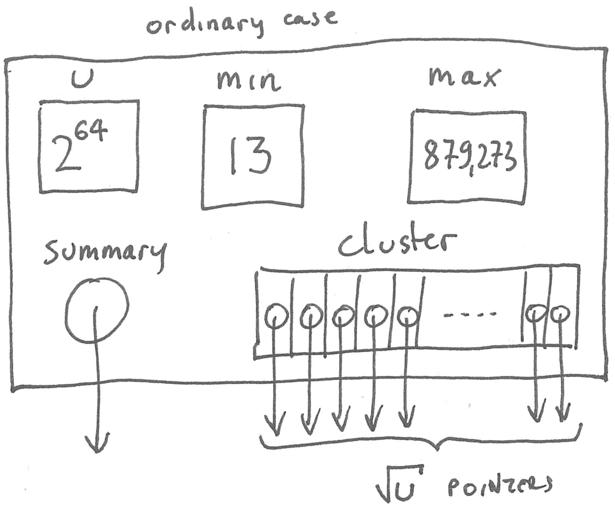
But, despite these disappointments, the proto-vEB tree still gives us hints of what to do next.

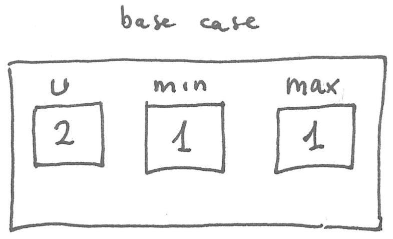
5.3.5 vEB tree

The vEB tree can be thought of as a proto-vEB tree optimized so that its methods have to make at most one recursive call. This is done to ensure that the recurrence does not gain a multiplying factor in front of the inner $T(\sqrt{u})$.

The variations to achieve this result are as follows.

- The vEB node contains all the fields of the proto-vEB node, plus two more that store the minimum and maximum key held in the node and its descendants.
- The minimum key for the node is actually only stored in the `min` field; it is *not* also stored in any of the clusters. The maximum, instead, is. (Can be confusing.)
- If a node contains no values, both the `min` and the `max` are set to `None`.
- In the base case of $u = 2$, the node does not have clusters nor summary. Its content can be deduced simply by observing its `min` and `max`.





Exercise 55
Deceptively difficult. Do not skip.
Sketch a picture of a vEB tree of size $u = 16$ representing the set 2, 3, 4, 5, 7, 14, 15. OK to study the textbook and handout first, but keep them closed while doing this exercise. No matter how good you are, you will almost certainly get some details wrong. That's OK. Check the textbook *after* having drawn your solution and mark in red all the items that are different, understanding why. Then *the next day* do the exercise again, with textbook closed. Iterate until you get no errors. Will take several days (no shame in that).
You may think you understand vEB trees but you actually don't until you successfully complete this exercise.

CLRS3 also describes another trick that allows $\lg u$ to be odd instead of even, but we are not going to worry about that detail in this course. It's just formal rules to decide which of the two "halves", h and l, gets the extra bit (the most significant part, h, is the one that ends up with one more bit if there is an odd number of them in $\lg u$).

What happens to the class methods and their costs with these customizations?

The minimum and maximum are trivially read off the corresponding fields of the node, so the `min()` and `max()` methods now only cost $O(1)$. Of course, whenever a data structure caches some information, we must be suspicious and wonder how much it costs to keep this cache up to date. So, what happens with `insert()`? If the node is empty, insert the value directly (into both the `min` and `max` fields, which previously held `None`, but without touching `summary` or `cluster` because the minimum is not stored in the clusters) at constant cost. Otherwise, insert the value `h|l` into both `cluster` and `summary`. However only one of these two calls is recursive: if `cluster[h]` was empty, insertion of `l` into `cluster[h]` has constant cost[17] and insertion of `h` into `summary` is recursive. If, on the other hand, `cluster[h]` wasn't empty, insertion of `l` into `cluster[h]` is recursive but insertion of `h` into `summary` is not even needed because `summary` necessarily already contained `h`. The resulting recurrence is $T(u) = T(\sqrt{u}) + O(1)$, which is $O(\lg \lg u)$.

What about finding the successor?

```
 0 def successor(self, k):
 1     """In this method, self is a vEB node of size u.
 2     k is a key in the range 0..u-1.
 3     Return the key that is the successor of k in self,
 4     or None if there is none."""
 5
 6     if self.u == 2:
 7         handle base case by brute force at constant cost
 8     elif self.min != None and k < self.min:
 9         return self.min
10         # NB: OK to ask for successor of a key not in self
11     else:
12         h = high(k, self.u)
13         l = low(k, self.u)
14         maxInCluster = self.cluster[h].max
15         if maxInCluster != None and l < maxInCluster:
16             # the successor of k is in k's own cluster
17             return index(h, self.cluster[h].successor(l))
18         else:
19             # the successor of k is in another cluster
20             nextCluster = self.summary.successor(h)
21             if nextCluster == None:
22                 return None
23             else:
24                 return index(nextCluster, self.cluster[nextCluster].min)
```

Does cluster h (the one containing k) also contain k's successor? With a proper vBE we can tell without a recursive call, just by checking with a single lookup (line 14) whether k is the maximum of its cluster (line 15). We then call `successor()` recursively only once per invocation, either on the key's own cluster in the "then" branch (line 17) or on the summary in the "else" branch (line 20). No path through this method has more than one recursive call of `successor()`, so the recurrence is again $T(u) = T(\sqrt{u}) + O(1)$, which is $O(\lg \lg u)$.

[17] It's just a matter of setting the `min` and `max` to `l`.

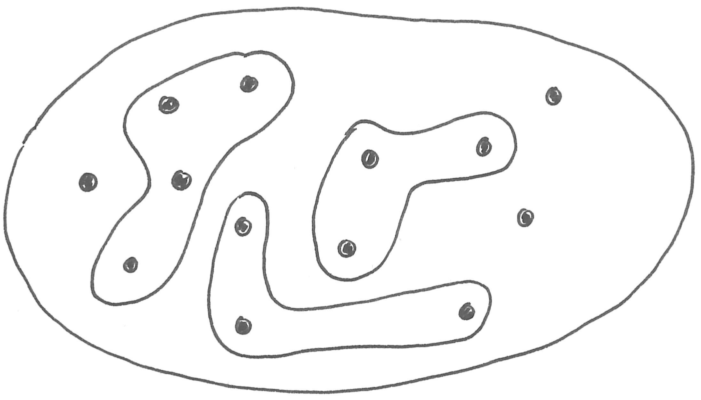
Deletion is more complicated (we won't reproduce a listing here but there is one in CLRS3), and so is its analysis because there is a possible path through the procedure in which two recursive calls are made. However it is possible to prove that, when this happens, one of the two calls takes constant time. Therefore the recurrence that applies is still $T(u) = T(\sqrt{u}) + O(1)$, which is $O(\lg \lg u)$.

5.4 Disjoint sets

Textbook

Study chapter 21 in CLRS3.

The disjoint set data structure, sometimes also known as union-find or merge-find, is used to keep track of a dynamic collection of disjoint sets (sets with no common elements) over a given universe of elements.



An example application can be seen in Kruskal's minimum spanning tree algorithm (section 6.3.1), where this data structure is used to keep track of the connected components of a forest: every connected component is stored as the set of its vertices and checking whether two vertices belong to the same disjoint set in the collection or to two different ones tells us whether adding an edge between them would introduce a cycle in the forest or not. Most other uses of this data structure are still generally related to keeping track of equivalence relations[18].

The available operations on a `DisjointSet` object, which despite its name is actually a *collection* of disjoint sets, allow us to add a new singleton set to the collection by supplying the element it contains (`makeSet()`), find the set that contains a given element (`findSet()`), and finally form the union of two sets in the collection (`union()`).

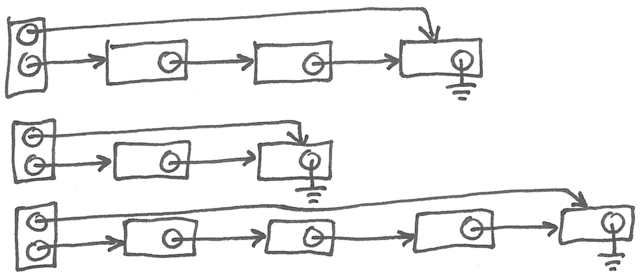
Conceptually, the `makeSet()` and `findSet()` methods return a "handle", which is anything that can act as a unique identifier for a set. If you invoke `findSet()` twice without modifying the set between the two calls, you will get the same handle. In practice, an implementer might choose to use one of the set elements as the handle.

[18]An equivalence relation is one that is reflexive, symmetric and transitive, such as "is of the same color as" or "has the same mother as" or "costs the same as".

```
ADT DisjointSet {
  Handle makeSet(Item x);
  // PRECONDITION: none of the existing sets already contains x.
  // BEHAVIOUR: create a new set {x} and return its handle.

  Handle findSet(Item x);
  // PRECONDITION: there exists a set that contains x.
  // BEHAVIOUR: return the handle of the set that contains x.

  Handle union(Handle x, Handle y);
  // PRECONDITION: x != y.
  // BEHAVIOUR: merge those two sets into one and return handle of new set.
  // Nothing is said about whether the returned handle matches either
  // or none of the supplied ones (i.e. whether set x absorbs y,
  // set y absorbs x or both sets get destroyed and a new one created).
}
```

5.4.1 List implementation

A relatively simple implementation of this data structure uses linked lists to store the sets. The `makeSet()` method creates a new one-item list and takes constant time. The `union()` method, too, can be made to take constant time if the `DisjointSet` object maintains, for each list in the collection, a pointer to the last element.



The `findSet()` method requires finding the head of the list that contains the given element[19]; however, in a singly-linked list, this cannot be done by following the list pointers because they go *away* from the list head, not towards it. If we add a further pointer from each list node to the head of the corresponding list, then we can implement `findSet()` in $O(1)$, but then `union()` is no longer $O(1)$ because we must update all the pointers-to-head of all the nodes of the list being appended. Indeed, one can easily construct pessimal input sequences for which the cost of $O(n)$ operations is $\Theta(n^2)$, yielding an amortized cost of $O(n)$ for each disjoint set operation. The following example does. Perform a sequence of $2n-1$ operations, of which n are `makeSet()` and $n-1$ are `union()`, according to the

[19]Remember (see the comments that open this chapter on page 80) that the `Item x` passed by the caller to the `findSet()` method is actually already a pointer to the list node that contains the set element of interest—you don't have to worry about finding the required set element in the data structure.

following pattern: keep alternating between making a new set and appending the long list with all the previous elements to the short one of the newly-made singleton set[20].

```
d = DisjointSet()
h0 = d.makeSet(x0)

h1 = d.makeSet(x1)
h0 = d.union(h0, h1)
h2 = d.makeSet(x2)
h0 = d.union(h0, h2)
h3 = d.makeSet(x3)
h0 = d.union(h0, h3)
h4 = d.makeSet(h4)
h0 = d.union(h0, h4)
  .
  .
  .
```

As the example implicitly suggests, the smart thing to do is instead to append the shorter list to the longer one; doing this requires us to keep track of the length of each list, but this is not a major overhead. With such a **weighted union** heuristic, it is possible to prove that the cost of a sequence of m operations on n elements[21] goes down to $O(m + n \lg n)$ time.

5.4.2 Forest implementation

A more elaborate representation stores each set in a separate tree (rather than list), with each node pointing to its parent. The `makeSet()` method creates a new root-only tree at cost $O(1)$, while `findSet()` returns the root of the tree by navigating up the edges of the tree, at cost $O(h)$ where h is the height of the tree. The `union()` operation appends the first tree to the second by making the root of the first tree a child of the root of the second[22], at cost $O(1)$.

With the same kind of reasoning that suggested the weighted union heuristic, we may improve the performance of `union()` by ensuring that the operation won't generate unnecessarily tall trees: we do this by keeping track of the **rank** of each tree (an upper bound on the height of the tree) and always appending the tree of smaller[23] rank to the other. This **union by rank** heuristic ensures that the rank of the resulting tree is either the same as that of the taller of the two trees or, at worst, one greater, if the two original trees had the same rank. In other words, the maximum rank of the trees in the collection grows as slowly as possible.

Another speedup is obtained by adopting the **path compression** heuristic, which at every `findSet(x)` "flattens" the path from `x` to the root of the tree. In other words, `x` and all the intermediate nodes between it and the root are reparented to become direct

[20]For any given implementation following the above naïve strategy, you can always achieve this pessimal append by appropriately choosing the order of the parameters in the call to `union()`.

[21]In other words, a sequence of m operations, with $m > n$, of which n are `makeSet()` and $m - n$ are `findSet()` or `union()`.

[22]These trees are not binary.

[23]Or equal.

children of the root. Stored ranks are not adjusted (which is why they end up being only upper bounds on the heights of the respective trees, rather than exact values). This operation costs no more than the $O(h)$ of the original `findSet()`, asymptotically, since all these nodes had to be visited in order to find the root anyway.

Exercise 56
If we are so obsessed with keeping down the height of all these trees, why don't we just maintain all trees at height ≤ 1 all along?

It can be shown with some effort that, if we adopt both "union by rank" and "path compression", the cost of a sequence of m operations on n items is $O(m \cdot \alpha(n))$, where $\alpha()$ is an integer-valued monotonically increasing function (related to the Ackermann function) that grows extremely slowly. Since $\alpha(n)$ is still only equal to 4 for n equal to billions of billions of times the number of atoms in the observable universe, it can be assumed that, for practical applications of a disjoint set, the value $\alpha(n)$ is bounded by the constant value 5 and may be ignored in the O notation. Therefore, "for all practical purposes", the performance of the forest implementation of the disjoint set with these two heuristics on a sequence of m operations is $O(m)$, meaning that the *amortized* cost per operation is constant.

Chapter 6

Graph algorithms

Chapter contents

Graph representations. Breadth-first and depth-first search. Topological sort. Minimum spanning tree. Kruskal and Prim algorithms. Single-source shortest paths: Bellman-Ford and Dijkstra algorithms. All-pairs shortest paths: matrix multiplication and Johnson's algorithms. Maximum flow: Ford-Fulkerson method. Matchings in bipartite graphs.
Expected coverage: about 5 lectures.

Graph theory can be said to have originated in 1736, when Euler proved the impossibility of constructing a closed path that would cross exactly once each of the 7 bridges of the city of Königsberg (Prussia).

Exercise 57
Build a "7-bridge" graph with this property. Then look up Euler and Königsberg and check whether your graph is or isn't isomorphic to the historical one. (Hint: you don't *have* to reconstruct the historical layout but note for your information that the river Pregel, which traversed the city of Königsberg, included two islands, which were connected by some of the 7 bridges.) Finally, build the *smallest* graph you can find that has this property.

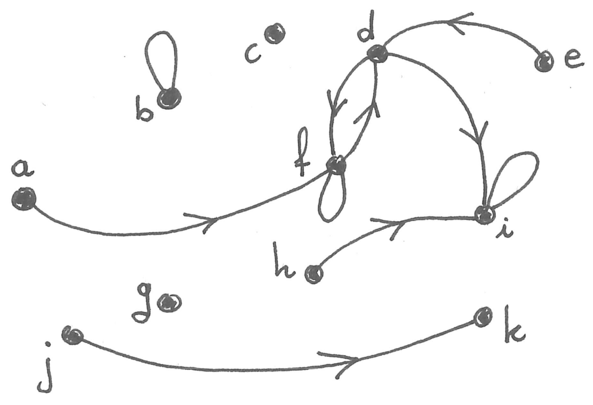
6.1 Basics

Textbook

Study chapter 22 in CLRS3.

6.1.1 Definitions

The general heading "graphs" covers a number of useful variations. Perhaps the simplest case is that of a general **directed** graph: this has a set V of vertices and a set E of ordered pairs of vertices that are taken to stand for directed edges. Such a graph is isomorphic to, and is therefore a possible representation for, a relation $R : V \to V$. Note that it is common to demand that the ordered pairs of vertices be all distinct, and this rules out having parallel edges. In some cases it may also be useful either to assume that for each vertex v, the edge (v, v) is present (in which case the relation is **reflexive**), or to demand that no edges joining any vertex to itself can appear (in which case the relation is **antireflexive**). The graph is called **undirected** when the edges have no arrows, i.e. when (v_1, v_2) is the same edge as (v_2, v_1), as is the case for the graph of the Königsberg bridges problem.



A sequence of zero or more edges[1][2] from vertex u to vertex v in a graph, indicated as $u \rightsquigarrow v$, forms a **path**. If each pair of vertices in the entire graph has a path linking them, then the graph is **connected**. A non-trivial[3] path from a vertex back to itself is called a **cycle**. Graphs without cycles have special importance, and the abbreviation **DAG** stands for Directed Acyclic Graph. An undirected graph without cycles is a **tree**[4], but not vice versa (because some trees are directed graphs). A directed tree is always a DAG, but not vice versa (think of the frequently-occurring diamond-shaped DAG). If the set of vertices V of a graph can be partitioned into two sets, L and R say, and each edge of the graph has one end in L and the other in R, then the graph is said to be **bipartite**.

[1]Zero when $u \equiv v$. Conversely, when a path $u \rightsquigarrow v$ consists of precisely one edge, we iron out the squiggles in the arrow and indicate it as $u \to v$.

[2]If the graph is directed, then all the edges of a path must go in the same direction. In other words, $a \to b \leftarrow c$ is not a path between a and c.

[3]Here, by "non-trivial" we mean a path with more than one edge; when a cycle has only one edge we call it a **loop**.

[4]Or a **forest** if made of several disconnected components.

Exercise 58
Draw in the margin an example of each of the following:

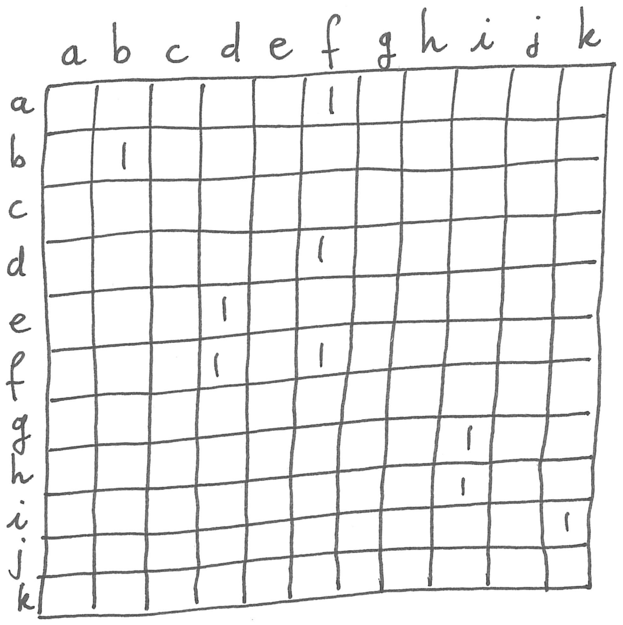
1. An anti-reflexive directed graph with 5 vertices and 7 edges.
2. A reflexive directed graph with 5 vertices and 12 edges.
3. A DAG with 8 vertices and 10 edges.
4. An undirected tree with 8 vertices and 10 edges.
5. A tree that is *not* “an undirected graph without cycles”.
6. A graph without cycles that is *not* a tree.

Actually, I cheated. Some of these can’t actually exist. Which ones? Why?

The definition of a graph can be extended to allow values to be associated with each edge—these will usually be called **weights** even though they may represent other concepts such as distances, carrying capacities, costs, impedances and so on. Graphs can be used to represent a great many things, from road networks to register use in optimizing compilers, to databases, to electrical circuits, to timetable constraints, to web pages and hyperlinks between them. The algorithms using them discussed here are only the tip of an important iceberg.

6.1.2 Graph representations

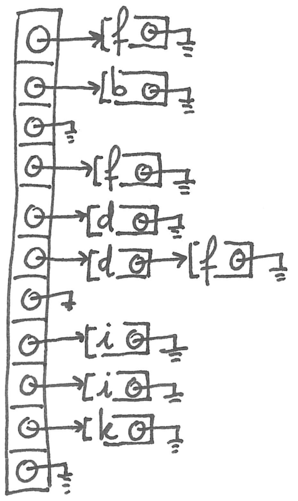
Barring specialized representations for particular applications, the two most obvious ways to represent a graph inside a computer are with an adjacency matrix or with adjacency lists.



	a	b	c	d	e	f	g	h	i	j	k
a						1					
b		1									
c											
d						1					
e				1							
f				1		1					
g									1		
h									1		
i											1
j											
k											

The **adjacency matrix**, especially suitable for **dense** graphs[5], is a square $|V| \times |V|$ matrix W in which w_{ij} gives the weight of the edge from vertex i to vertex j (or, for unweighted graphs, a boolean value indicating the presence or absence of an edge). This format makes manipulation easy but obviously has $O(|V|^2)$ storage costs.

Therefore, for **sparse** graphs[5], a more economical solution is often preferred: for each vertex v, an **adjacency list** (usually unsorted) contains the vertices that can be reached from v in one hop. For weighted graphs, the weight of the corresponding edge is recorded alongside each destination vertex.



A comment on notation: the computational complexity of a graph algorithm will usually be a function of the cardinality of the sets V and E; however, given that E and V on their own as sets have no meaning in a big-O formula, writing down the cardinality bars has no disambiguating function and only makes the formulae longer and less legible, as in $O(|E| + |V| \lg |V|)$. Therefore the convention is to omit the bars within the big-O notation and write instead things like $O(E + V \lg V)$.

6.1.3 Searching (breadth-first and depth-first)

Many problems involving graphs are solved by systematically searching. This usually means following graph edges so as to visit all the vertices. Note that we assume that we have access to a list of all the vertices of the graph and that we can access it regardless of the arrangement of the graph's edges; this way, when a search algorithm stops before visiting all the vertices, we can restart it from one of the vertices still to be visited. This "backdoor" does not, as some might think, defeat the point of graph searching: in general, the objective is not merely to access the vertices but to visit them according to the structure of the graph—for example to establish what is the maximum distance between any two vertices in the graph[6], which by the way is known as the **diameter** of the graph.

[5]A dense graph is one with many edges ($|E| \approx |V|^2$) and a sparse graph is one with few ($|E| \ll |V|^2$).

[6]I should be more precise when saying something like this. In this instance I am referring to a "maximum of minimums"; in other words, imagine to compute, for every two vertices in the graph, the shortest path between them; then I want the longest of these shortest paths. Otherwise the definition of *maximum distance* might be inconsistent, as one might be able to take arbitrarily many detours round

The two main strategies for inspecting a graph are *depth-first* and *breadth-first*. In both cases we may describe the search algorithm as a vertex colouring procedure. All vertices start out as white, the virginal colour of vertices that have never been visited. One vertex is chosen as source of the search: from there, other vertices are explored following graph edges. Each vertex is coloured grey as soon as it is first visited, and then black once we have visited all its adjacent vertices. Depending on the structure of the graph, for example if a graph has several disconnected components, it may be necessary to select another source if the original one has been made black but there still exist white vertices.

```
def bfs(G, s):
    """Run the breadth-first search algorithm on the given graph G
    starting from the given source vertex s."""

    assert(s in G.vertices())

    # Initialize graph and queue:
    for v in G.vertices():
        v.predecessor = None
        v.d = Infinity # .d = distance from source
        v.colour = "white"
    Q = Queue()

    # Visit source vertex
    s.d = 0
    s.colour = "grey"
    Q.insert(s)

    # Visit the adjacents of each vertex in the queue
    while not Q.isEmpty():
        u = Q.extract()
        assert (u.colour == "grey")
        for v in u.adjacent():
            if v.colour == "white":
                v.colour = "grey"
                v.d = u.d + 1
                v.predecessor = u
                Q.insert(v)
        u.colour = "black"
```

In the case of **breadth-first** search, from any given vertex we visit all the adjacent vertices in turn before going any deeper. This is achieved by inserting each vertex into a queue as soon as it gets visited and, for each vertex extracted from the queue, by visiting all its unvisited (white) adjacent vertices before considering it done (black). Only white vertices are considered for insertion in the queue and they get painted grey on insertion and black on extraction; as a result, the queue contains only grey vertices and there are

the cycles of the graph while navigating from one vertex to the other. I should also specify whether the length of a path is considered to be the number of edges on it, as is the case for the graph diameter; or—as indeed makes more sense for many other applications—the sum of the weights of the edges on it.

no grey vertices other than those in the queue. Consider the colour grey as indicating that a vertex has been visited by the procedure, but not completed.

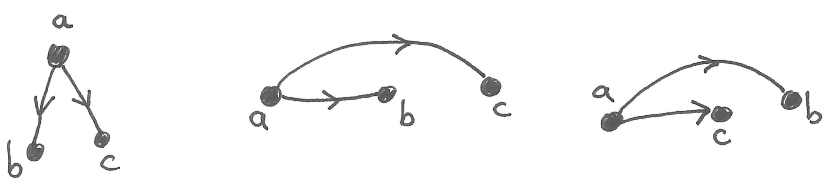
Depth-first search, instead, corresponds to the most natural recursive procedure for walking over a tree: from any particular vertex, the whole of one sub-tree is investigated before any others are looked at at all. In the case of a generic graph rather than a tree, a little extra care is necessary to avoid infinite loops—but the colouring, otherwise unnecessary with trees, helps with that.

```
def dfs(G, s):
    """Run the depth-first search algorithm on the given graph G
    starting from the given source vertex s."""

    assert(s in G.vertices())

    # Initialize graph:
    for v in G.vertices():
        v.predecessor = None
        v.colour = "white"

    dfsRecurse(G, s)

def dfsRecurse(G, s):
    s.colour = "grey"
    s.d = time() # .d = discovery time
    for v in s.adjacent():
        if v.colour == "white":
            v.predecessor = s
            dfsRecurse(G, v)
    s.colour = "black"
    s.f = time() # .f = finish time
```

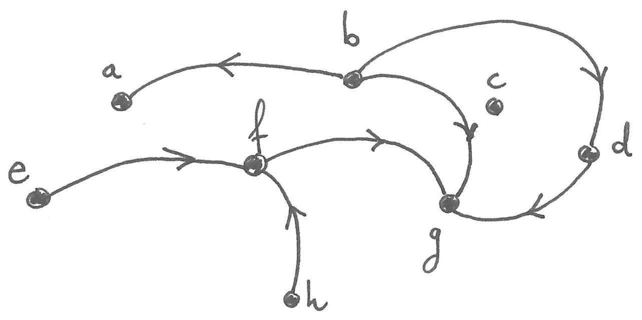
Breadth-first search can often avoid getting lost in fruitless scanning of deep parts of the tree, but the queue that it uses often requires more memory than the stack implicitly used by depth-first search.

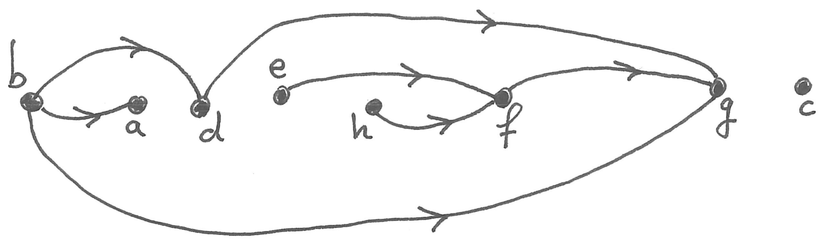
6.2 Topological sort

The problem of topological sort is that of linearizing a directed acyclic graph (DAG): you have a DAG and you wish to output its vertices in such an order that "all the arrows go forward", i.e. no edge goes from a vertex v to a vertex u that was output before v. In a project management scenario you might think of the vertices of the DAG as actions and of the edges as dependencies between actions; then the topological sort provides a sequence in which the actions can actually be performed. It ought to be clear that, for a generic DAG, there will usually be more than one valid linearizing sequence for its vertices—a Λ-shaped three-vertex tree provides a trivial example.



Below is a less trivial DAG and one possible linearization for it.





A deceptively simple algorithm, due to Knuth, solves the problem in $O(V+E)$ using a depth-first search. The strategy is to perform an exhaustive depth-first search that visits all the vertices in the graph (restarting from any leftover white vertex if the recursive search returns before having painted all the vertices black) and then to output the vertices in reverse order of finishing time, where the finishing time of a vertex is defined as the time at which it gets coloured black. But why should this work at all?

Exercise 59
Draw a random DAG in the margin, with 9 vertices and 9 edges, and then perform a depth-first search on it, marking each vertex with two numbers as you proceed—the discovery time (the time the vertex went grey), then a slash and the finishing time (the time it went black). Then draw the linearized DAG by arranging the vertices on a line in reverse order of their finishing time and reproducing the appropriate arrows between them. Do the arrows all go forward?

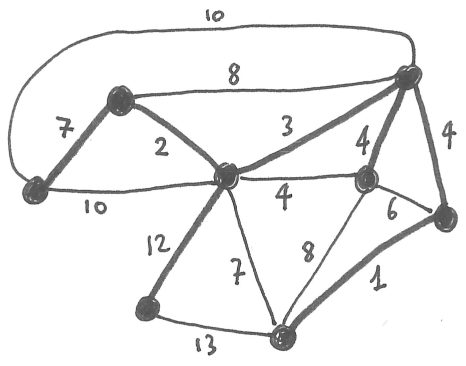
The insight is that, in order to blacken vertex u, I must have already blackened all of its descendents. It is easy to see that this is true for the vertices in u's adjacency list but a little more care is needed to deal with those further down.

Exercise 60
Develop a proof of the correctness of the topological sort algorithm. *(Requires some thought.)*

6.3 Minimum spanning tree

Textbook

Study chapter 23 in CLRS3.



Given a connected undirected graph where the edges have all been labelled with non-negative weights, the problem of finding a **minimum spanning tree** is that of finding the sub-graph of minimum total weight[7] that links all vertices. This must necessarily be a tree. Suppose it isn't: then either it is a forest, in which case it fails to connect all vertices, which contradicts the hypothesis; or it contains a cycle. Removing any one edge from the cycle would leave us with a graph with fewer edges and therefore (since no edges have negative weight) of lower or equal weight, but still connecting all the vertices, again contradicting the hypothesis, QED. Note that a graph may have more than one minimum spanning tree—these would be distinct trees with the same total weight.

Exercise 61
Find, by hand, a minimum spanning tree for the graph drawn in CLRS3 figure 23.4.(a) (trying not to look at nearby figures). Then see if you can find any others.

A generic algorithm that finds minimal spanning subtrees involves growing a subgraph A by adding, at each step, a **safe** edge[8] of the full graph, defined as one that ensures that the resulting subgraph is still a subset of some minimum spanning tree.

[7]Not, obviously, "of minimum number of edges", as that is a trivial problem—*any* tree touching all the vertices is one.

[8]With respect to A.

```
def minimumSpanningTree(G):
    A = empty set of edges
    while A does not span all vertices yet:
        add a safe edge to A
```

At this level of abstraction it is still relatively easy to prove that the algorithm is correct, because we are handwaving away the comparatively difficult task of choosing a safe edge.

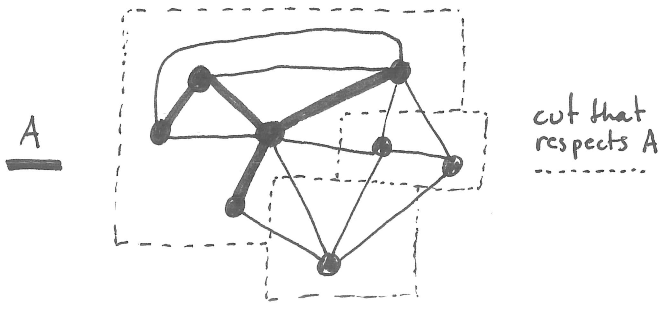
We shall soon describe two instantiations of this generic algorithm that differ in the criterion they use to *choose* a safe edge at each iteration. Before that, though, let's develop a criterion for *recognizing* safe edges.

We start with a couple of definitions. Given a graph $G = (V, E)$, a **cut** is a partition of G's vertices into at least two sets. Given a (possibly non-connected) subgraph A of G, a cut of G **respects** A if and only if no edge of A goes across the cut[9].

Now the theorem: given a graph G and a subgraph A that is a subset of a minimum spanning tree of G, for any cut that respects A, the lightest edge of G that goes across the cut is safe for A.

To prove this, imagine the full minimum spanning tree T of which A is by hypothesis a subgraph and call e_l the lightest edge across the chosen cut. If $e_l \in T$, the theorem is satisfied. Are there any alternatives? If $e_l \notin T$, then adding it to T introduces a loop (because the two endpoints of e_l were already connected to each other through edges of T, by hypothesis of T being a spanning tree of G). For topological reasons, this loop must contain at least one other edge going across the cut and distinct from e_l: call it (or any one of them if there are several) e_x.

I recommend you work out the theorem as you go along on the partial graph below, figuring out where T is and so on. With a pre-completed graph you would learn less than with one you reconstruct by yourself. The highlighted edges are in A, whereas the dotted lines represent the cut.



Now call T_l the spanning tree[10] you get from T by substituting e_l for e_x (note that $e_x \in T$, $e_l \in T_l$). Since e_l is the lightest edge across the cut by hypothesis, either it weighs the same as e_x (in which case T_l is another equally good minimum spanning tree,

[9]We say that an edge "goes across the cut" if and only if the two endpoints of the edge belong to different sets of the partition.

[10]T_l is indeed a tree because we started from a tree T, we added an edge between two distinct vertices, thus introducing a loop, then removed another edge of that loop, yielding another tree over the same vertices. It is a spanning tree because it still connects the same set of vertices as T, which was a spanning tree.

and therefore e_l is safe since it belongs to an MST and the theorem is satisfied), or e_l weighs strictly less than e_x (but then T_l is a spanning tree of smaller weight than T, which contradicts the hypothesis that T is a *minimum* spanning tree, so this case can't happen). QED.

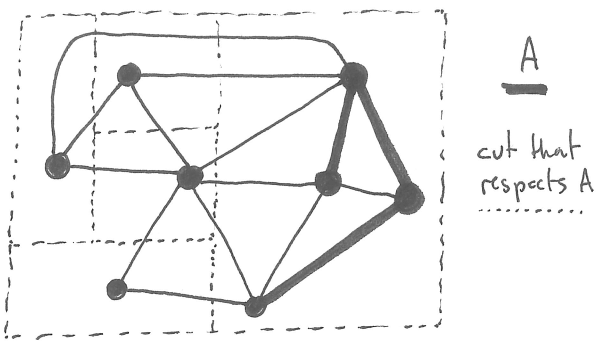
Note that, although for a given cut there is only one safe *edge* (namely the lightest, unless there are several *ex aequo* minimum weight edges across the cut), one still has a wide choice of possible *cuts* that respect the subgraph A formed so far. The two algorithms that follow, as announced, use this freedom in different ways.

6.3.1 Kruskal's algorithm

Kruskal's algorithm allows A to be a forest during execution. The algorithm maintains a set A of edges, initially empty, that will eventually become the MST. The initial cut is one that partitions each vertex into a separate set. The edge to be added at each step is the lightest one that does not add a cycle to A. Once an edge is added to A, the sets of its endpoints edge are merged, so that the partition contains one fewer set. In other words the chosen cut is, at all times, the one that partitions each connected[11] cluster of vertices into its own set.

```
def kruskal(G):
    """Apply Kruskal's algorithm to graph G.
    Return a set of edges from G that form an MST."""

    A = Set() # The edges of the MST so far; initially empty.
    D = DisjointSet()
    for v in G.vertices():
        D.makeSet(v)
    E = G.edges()
    E.sort(key=weight, direction=ascending)

    for edge in E:
        startSet = D.findSet(edge.start)
        endSet = D.findSet(edge.end)
        if startSet != endSet:
            A.append(edge)
            D.union(startSet, endSet)
    return A
```

[11]Connected via edges in $A \subset E$, that is—not merely connected via any edges in E.



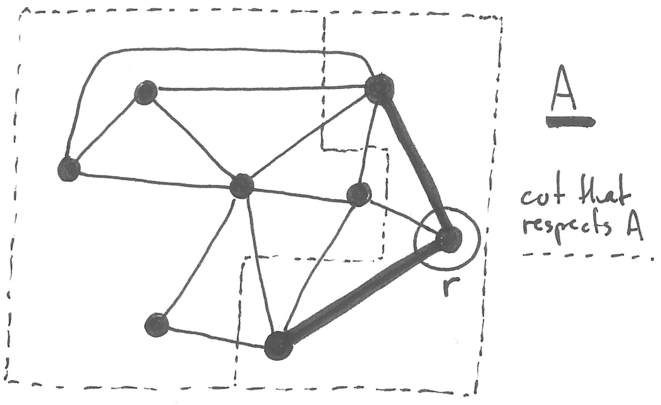
In the main `for` loop starting on line 11, all the edges of the graph are analyzed in increasing order of weight. Any that don't add a loop to the ones already chosen are selected and brought into A, the MST-so-far. To find out whether a candidate edge would form a cycle when added to the edges already in set A, we maintain a disjoint-set D (see section 5.4) which has a set for each connected component of the forest being formed in A. If the candidate edge has its endpoints in two separate components, then it won't add a cycle and it is safe to take it as an edge of the MST-so-far (with appropriate updating of D, since it will connect the two previously disconnected components); otherwise that edge would form a cycle and so it must be rejected.

The cost of this main loop (lines 11–16) is easily estimated: $|E|$ times the cost of two `findSet()` and one `union()`. With the asymptotically-fastest disjoint-set implementation known, a sequence of any m `makeSet()`, `findSet()` and `union()` operations will take $O(m\alpha(n))$, or in practice[12] just $O(m)$ which in our case is $O(V + 3E) = O(E)$. So the main loop actually costs even less than the setting-up operations in the first part of the algorithm (lines 4–9), where the dominant cost is a standard $O(E \lg E)$ for the sorting of the set of edges (line 9). Noting that $|E| \leq |V|^2$ and therefore $\lg |E| \leq 2 \lg |V|$ and therefore $O(\lg E) = O(\lg V)$, we can also write the cost of Kruskal's algorithm as $O(E \lg V)$.

6.3.2 Prim's algorithm

Prim's algorithm, starting from a designated root vertex r, forces A to be a tree throughout the whole operation. The cut is the one that partitions the vertices into just two sets, those touched by A and the others. The edge to be added is the lightest one that joins a new vertex to the tree built so far.

[12]Again, see section 5.4 for more on the disjoint-set data structure, on $\alpha(n)$ and on why it is OK to treat $\alpha(n)$ as a constant in this formula.



```
 0  def prim(G, r):
 1      """Apply Prim's algorithm to graph G starting from root vertex r.
 2      Return the result implicitly by modifying G in place: the MST is the
 3      tree defined by the .predecessor fields of the vertices of G."""
 4
 5      Q = MinPriorityQueue()
 6      for v in G.vertices():
 7          v.predecessor = None
 8          if v == r:
 9              v.key = 0
10          else:
11              v.key = Infinity
12          Q.insert(v)
13
14      while not Q.isEmpty():
15          u = Q.extractMin()
16          for v in u.adjacent():
17              w = G.weightOfEdge(u,v)
18              if Q.hasItem(v) and w < v.key:
19                 v.predecessor = u
20                 Q.decreaseKey(item=v, newKey=w)
```

Each vertex is added to a priority queue, keyed by its distance to the MST-so-far. The main `while` loop on lines 14–20 is executed once for each vertex. At each step we pay for an `extractMin()` from the priority queue to extract the closest vertex `u` and add it to the MST-so-far. All its adjacent vertices are then examined and, if appropriate (that is to say: if reaching them via `u` makes them closer to the MST-so-far than they previously were), we perform `decreaseKey()` on them. How many times does this happen? Since no adjacency list is longer than $|V|$, the answer is at worst $O(V^2)$. But this may be an overestimate on sparse graphs: a tighter bound is obtained through aggregate analysis by observing that the sum of the lengths of all the adjacency lists is $2|E|$ (each edge counted twice, once by each of its endpoints); so, during a full run of the algorithm, the inner `for` loop on lines 16–20 is executed a number of times bounded by $O(E)$. This $O(E)$ is equal to $O(V^2)$ in dense graphs, but can go down to $O(V)$ in sparse graphs[13]. So the cost of Prim is

[13]Or even less for graphs that are not connected, but this is not the case here by hypothesis.

$O(V)$ times the cost of `extractMin()` plus $O(E)$ times the cost of `decreaseKey()`. With a regular binary heap implementation for the priority queue, these two operations cost $O(\lg V)$ each, yielding an overall cost for Prim of $O(V \lg V + E \lg V) = O(E \lg V)$—same as the best we achieved with Kruskal.

If the priority queue is implemented with a Fibonacci heap (see section 5.2), where the cost of `extractMin()` stays at $O(\lg V)$ but the cost of `decreaseKey()` goes down to (amortized) constant, the cost of Prim's algorithm goes down to $O(V \lg V + E)$, improving on Kruskal.

Exercise 62
Starting with fresh copies of the graph you used earlier, run these two algorithms on it by hand and see what you get. Note how, even when you reach the same end result, you may get to it via wildly different intermediate stages.

6.4 Shortest paths from a single source

Textbook

Study chapter 24 in CLRS3.

The problem is easily stated: we have a weighted directed graph; two vertices are identified and the challenge is to find the shortest route through the graph from one to the other.

An amazing fact is that, for sparse graphs, the best ways so far discovered of solving this problem may do as much work as a procedure that sets out to find distances from the source to *all* the other vertices in the entire graph. This illustrates that, if we think in terms of particular applications (for instance distances in a road atlas in this case) but then try to make general statements, our intuition on graph problems may be misleading.

While we are considering algorithms to discover the shortest paths from a designated source vertex s to any others, let us indicate as $v.\delta$ the length[14] of the shortest path from source s to vertex v, which we may not know yet, and as $v.d$ the length of the shortest path *so far discovered* from s to v. Initially, $v.d = +\infty$ for all vertices except s (for which $s.d = s.\delta = 0$); all along, $v.d \geq v.\delta$; and finally, on completion of the algorithm, $v.d = v.\delta$.

[14] Or weight—we use these terms more or less interchangeably in this section, because so does the literature. Do not get confused. In particular, we usually do not mean "number of edges" when we speak of the "length" of a path, but rather the sum of the weights of all its edges.

Exercise 63
Give a formal proof of the intuitive "triangle inequality"

$$v.\delta \leq u.\delta + w(u,v)$$

(where $w(u,v)$ is the weight of the edge from vertex u to vertex v) but covering also the case in which there is actually no path between s and v.

In some graphs it may be meaningful to have edges with negative weights. This complicates the analysis. In particular, if a negative-weight *cycle* exists on a path from s to v, then there is no finite-length shortest path between s and v—because it is always possible to find an even "shorter" path (i.e. one with a lower total weight, despite it having more edges) by running through the negative-weight cycle one more time. On the other hand, in graphs with negative-weight edges but no negative-weight cycles, shortest paths are well defined, but some algorithms may not work because their proof of correctness relies on the assumption that edge weights are never negative. For example the Dijkstra algorithm presented in section 6.4.2, although very efficient, gives incorrect results in presence of negative weigths. There exist, though, other algorithms capable of dealing with non-degenerate negative weight cases: one of them is the Bellman-Ford algorithm introduced in section 6.4.1.

An important step in most shortest-path algorithms is called **relaxation**: given two vertices u and v, each with its current "best guess" $u.d$ and $v.d$, and joined by an edge (u,v) of weight $w(u,v)$, we consider whether we would discover a shorter path to v by going through (u,v). The test is simple: if $u.d + w(u,v) < v.d$, then we can improve on the current estimate $v.d$ by going through u and (u,v). Doing so is indicated as "relaxing the edge (u,v)".

A variety of useful lemmas (microtheorems) can be proved about the properties of $.d$ and $.\delta$: one of them is the **triangle inequality** mentioned in the preceding exercise.

Another one, which we'll use later to prove the correctness of Dijkstra's algorithm, is the **convergence lemma**: if $s \rightsquigarrow u \rightarrow v$ is a shortest path from s to v, and at some time $u.d = u.\delta$, and at some time after that the edge (u,v) is relaxed, then, from then on, $v.d = v.\delta$.

Another one, which is helpful in proving the correctness of the Bellman-Ford algorithm to be examined next, is the **path relaxation lemma**: if $p = (s, v_1, v_2, \ldots, v_k)$ is a shortest path from s to v_k, and the edges of p are relaxed in the order $(s, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, even if other relaxation steps elsewhere in the graph are intermixed with those on the edges of p, then after these relaxations we have that $v_k.d = v_k.\delta$.

Consult your textbook for proofs, and for more lemmas.

6.4.1 The Bellman-Ford algorithm

Bellman-Ford computes shortest paths to all vertices from a designated source s, even in the presence of negative edge weights, so long as no negative weight *cycles* are reachable from s. If any are, it reports their existence and refuses to give any other results, because in such a graph shortest paths cannot be defined.

```
0  def bellmanFord(G, s):
1      """Apply Bellman-Ford's algorithm to graph G, starting from
2      source vertex s, to find single-source shortest paths to all vertices.
3      Return a boolean indicating whether the graph is free from
4      negative cycles reachable from s.
5
6      Return the single-source-shortest-paths (only valid if the
7      above-mentioned boolean is true) by modifying G in place: for each
8      vertex v, the length of the shortest path from source to it is
9      left in v.d and the shortest paths themselves are indicated by the
10     .predecessor fields of the vertices."""
11
12     assert(s in G.vertices())
13     for v in G.vertices():
14         v.predecessor = None
15         v.d = Infinity
16     s.d = 0
17
18     repeat |V|-1 times:
19         for e in G.edges():
20             # Relax edge e.
21             if e.start.d + e.weight < e.end.d:
22                 e.end.d = e.start.d + e.weight
23                 e.end.predecessor = e.start
24
25     # If, after all this, further relaxations are possible,
26     # then we have a negative cycle somewhere
27     # and all the previous .d and .predecessor results are worthless.
28     for e in G.edges():
29         if e.start.d + e.weight < e.end.d:
30             return False
31     return True
```

Bellman-Ford works by considering each edge in turn and relaxing it if possible. This full pass on all the edges of the graph (lines 19–23) is repeated a number of times equal to the maximum possible length of a shortest path, $|V| - 1$ (lines 18–23). The algorithm therefore has a time complexity of $O(VE)$. It is not hard to show that, after that many iterations, in the absence of negative-weight cycles, no further relaxations are possible and that the distances thus discovered for each vertex are indeed the smallest possible ones.

The core of the correctness proof is based on the path relaxation lemma (page 121). For any vertex v for which a shortest path p of finite length exists from s to it, that path may contain at most $|V|$ vertices. As far as this path is concerned, the purpose of the i-th round of the outer loop in line 18 is to relax the i-th edge of the path. This guarantees that, regardless of any other relaxations that the algorithm also performs in between, it will eventually relax all the edges of p in the right order and therefore, by the lemma, it will achieve $v.d = v.\delta$.

An $O(E)$ post-processing phase (lines 28–31) then detects any negative-weight cycles. If none are found, the distances and paths discovered in the main phase are returned as valid.

6.4.2 The Dijkstra algorithm

The Dijkstra[15] algorithm only works on graphs without negative-weight edges but, on those, it is more efficient than Bellman-Ford.

Starting from the source vertex s, the algorithm maintains a set S of vertices to which shortest paths have already been discovered. The vertex $u \notin S$ with the smallest $u.d$ is then added to S (to justify this move one should prove that, at that time, $u.d = u.\delta$) and then all the edges starting from u are relaxed. This sequence of operations is repeated until all vertices are in S, in a strategy reminiscent of Prim's algorithm.

```
0  def dijkstra(G, s):
1      """Apply Dijkstra's algorithm to graph G, starting from
2      source vertex s, to find single-source shortest paths to all vertices.
3
4      Return the single-source-shortest-paths (only valid if the graph
5      has no negative-weight edges) by modifying G in place: for each
6      vertex v, the length of the shortest path from source to it is
7      left in v.d and the shortest paths themselves are indicated by the
8      .predecessor fields of the vertices."""
9
10     assert(s in G.vertices())
11     assert(no edges of G have negative weight)
12     Q = MinPriorityQueue() # using .d as each item's key
13     for v in G.vertices():
14         v.predecessor = None
15         if v == s:
16             v.d = 0
17         else:
18             v.d = Infinity
19         Q.insert(v)
20
21     # S (initially empty) is the set of vertices of G no longer in Q.
22     while not Q.isEmpty():
23         u = Q.extractMin()
24         assert(u.d == u.delta) # NB we can't actually _compute_ u.delta.
25         for v in u.adjacent():
26             # Relax edge (u,v).
27             if Q.hasItem(v) and u.d + G.weightOfEdge(u,v) < v.d:
28                 v.predecessor = u
29                 Q.decreaseKey(item = v, newKey = u.d + G.weightOfEdge(u,v))
```

As can be seen, if any vertex v is unreachable from s, its $v.d$ will stay at $+\infty$ throughout and it will be processed, as will any others in that situation, in the final round(s) of the `while` loop of lines 22–29; but the algorithm will still terminate without problems. Conversely, any vertex v whose $v.d$ is still $+\infty$ after completion is indeed unreachable from s.

[15]It's a Dutch name: pronounce as [ˈdɛɪkstra].

Computational complexity

Performance-wise, to speed up the extraction of the vertex with the smallest $u.d$, we keep the unused vertices in a min-priority queue Q where they are keyed by their $.d$ attribute. Then, for each vertex, we must pay for one `extractMin()` plus as many `decreaseKey()` as there are vertices in the adjacency list of the original vertex. We don't know the length of each individual adjacency list, but we know that all of them added together have $|E|$ edges (since it's a directed graph). So in aggregate we pay for $|V|$ times the cost of `extractMin()` and $|E|$ times the cost of `decreaseKey()`. With a regular binary heap implementation for the priority queue, these two operations cost $O(\lg V)$ each, yielding an overall cost for Dijkstra of $O(V \lg V + E \lg V) = O(E \lg V)$. Same as Prim.

As with Prim, though, we can however do better if we make the priority queue faster: you will recall that the development of Fibonacci heaps (section 5.2) was originally motivated by the desire to speed up the Dijkstra algorithm. Using a Fibonacci heap to hold the remaining vertices, the `decreaseKey()` cost is reduced to amortized $O(1)$, so the overall cost of the algorithm goes down to $O(V \lg V + E)$.

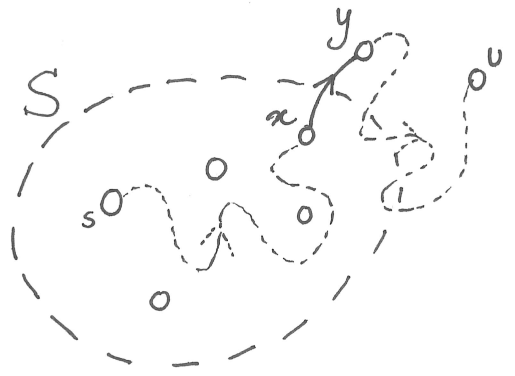
Correctness proof

To prove the correctness of Dijkstra's algorithm the crucial point, as already noted, is to prove that, for the vertex u that we extract from the queue in line 23, it is indeed the case that the assertion on line 24, i.e. $u.d = u.\delta$, holds. We prove this by contradiction.

Since $u.d$ can never be $< u.\delta$ by definition of $.d$ and $.\delta$, the only two cases left are "$u.d = u.\delta$" and "$u.d > u.\delta$". Let's imagine for the sake of argument that there is a vertex u that, when extracted from Q in line 23, has

$$u.d > u.\delta. \tag{6.1}$$

Consider the *first* such vertex we encounter while running the algorithm and consider the shortest path[16] from s to u.



Represent this shortest path as $s \rightsquigarrow x \rightarrow y \rightsquigarrow u$, where (x, y) is the first edge along this path for which $x \in S$ and $y \notin S$. At the time (line 23) of adding u to S, since u was the chosen vertex and thus the one with the smallest $.d$ among all the vertices outside S,

$$u.d \leq y.d. \tag{6.2}$$

[16]Detour: we should also prove that a shortest path exists, for which it is sufficient to prove that u is reachable—and it is, otherwise $u.d = u.\delta = +\infty$, contradicting hypothesis (6.1)

Since the path $s \rightsquigarrow x \rightarrow y$ matches all the conditions of the convergence lemma (the path is a subpath of a shortest path and hence a shortest path itself; $x.d = x.\delta$ because $x \in S$ and u, found after x, is the first vertex to violate that property; and (x, y) was relaxed when x was added to S), we have that, at the time of adding u to S,

$$y.d = y.\delta. \tag{6.3}$$

Combining equations (6.1), (6.2) and (6.3) we have $u.\delta < u.d \leq y.d = y.\delta$, which implies

$$u.\delta < y.\delta. \tag{6.4}$$

However, since $s \rightsquigarrow x \rightarrow y \rightsquigarrow u$ is a shortest path by hypothesis, on the assumption that all edges have non-negative weights then reaching u via y must cost at least as much as reaching y and therefore $u.\delta \geq y.\delta$. This directly contradicts (6.4), proving that there cannot be any vertex u satisfying (6.1) when extracted from Q at line 23, QED.

6.5 Shortest paths between any two vertices

Textbook

Study chapter 25 in CLRS3.

It is of course possible to compute all-pairs shortest paths simply by running the single-source shortest path algorithm on each vertex. If the graph contains negative edges, running Bellman-Ford on all vertices will cost $O(V^2E)$, which is $O(V^4)$ for dense graphs and $O(V^3)$ for sparse graphs. For a graph without negative edges, using Dijkstra and Fibonacci heaps we can compute all-pairs shortest paths in $O(V^2 \lg V + VE)$. Interestingly, there are ways in which we can do better.

Note that, in this section, we compute the weight of the shortest path between two vertices without worrying about keeping track of the edges that form that path. You might consider addressing that issue but I am not making this a boxed exercise as it may take you a while. The most interesting way to find out about it is clearly to write an actual program.

6.5.1 All-pairs shortest paths via matrix multiplication

Let's first look at a method inspired by matrix multiplication, which works even in the presence of negative-weight edges, though necessarily we require the absence of negative-weight *cycles*.

Represent the graph with an adjacency matrix W whose elements are defined as:

$$w_{i,j} = \begin{cases} \text{the weight of edge } (i,j) & \text{for an ordinary edge;} \\ 0 & \text{if } i = j; \\ \infty & \text{if there is no edge from } i \text{ to } j. \end{cases}$$

We want to obtain a matrix of shortest paths L in which $l_{i,j}$ is the weight of the shortest path from i to j (∞ if there is no path).

Let $L^{(m)}$ be the matrix of shortest paths that contain no more than m edges. Then $W = L^{(1)}$.

Exercise 64
Write out explicitly the elements of matrix $L^{(0)}$.

Let's build $L^{(2)}$:

$$l^{(2)}_{i,j} = \min\left(l^{(1)}_{i,j}, \quad \min_{k=1,n}(l^{(1)}_{i,k} + w_{k,j})\right).$$

In other words, the shortest path consisting of at most two steps from i to j is either the shortest one-step path or the shortest two-step path, the latter obtained by fixing the endpoints at i and j and trying all the possible choices for the intermediate vertex k. Similarly, we build $L^{(3)}$ from $L^{(2)}$ by adding a further step: it's either the shortest path of at most two steps, or the shortest path of at most three steps; the latter obtained by trying all possible combinations for an additional last edge, with the first two steps given by a shortest two-step path.

$$l^{(3)}_{i,j} = \min\left(l^{(2)}_{i,j}, \quad \min_{k=1,n}(l^{(2)}_{i,k} + w_{k,j})\right).$$

We proceed using the same strategy for $L^{(4)}$, $L^{(5)}$ and so on.

We can also simplify the formula by noting that $w_{j,j} = 0 \quad \forall j$ and that therefore the term with $k = j$ reduces from $(l^{(2)}_{i,k} + w_{k,j})$ to $(l^{(2)}_{i,j} + w_{j,j}) = (l^{(2)}_{i,j} + 0)$ and is thus equal to the first $l^{(2)}_{i,j}$ inside the big brackets; thus we don't need to have two "rounds" of minimum to include it and we can simply write, in the general case:

$$l^{(m+1)}_{i,j} = \min\left(l^{(m)}_{i,j}, \quad \min_{k=1,n}(l^{(m)}_{i,k} + w_{k,j})\right) = \min_{k=1,n}(l^{(m)}_{i,k} + w_{k,j}).$$

It is easy to prove that this sequence of matrices $L^{(m)}$ converges to L after a finite number of steps, namely $n-1$. Any path with more than $n-1$ steps must revisit a vertex and therefore include a cycle. Under the assumption that there are no negative-weight cycles, the path cannot be shorter than the one with fewer edges obtained from it by removing the cycle. Therefore

$$L^{(n-1)} = L^{(n)} = L^{(n+1)} = \ldots = L.$$

The operation through which we compute $L^{(m+1)}$ from $L^{(m)}$ and W is reminiscent of the matrix multiplication

$$P = L^{(m)} \cdot W.$$

Compare

$$l_{i,j}^{(m+1)} = \min_{k=1,n} (l_{i,k}^{(m)} + w_{k,j})$$

and

$$p_{i,j} = \sum_{k=1,n} l_{i,k}^{(m)} \cdot w_{k,j}$$

to discover the mapping:

$$\begin{array}{rcl} \min & \leftrightarrow & + \\ + & \leftrightarrow & \cdot \\ \infty & \leftrightarrow & 0 \\ 0 & \leftrightarrow & 1 \end{array}$$

Exercise 65
To what matrix would $L^{(0)}$ map? What is the role of that matrix in the corresponding algebraic structure?

It is possible to prove that this mapping is self-consistent. Let's therefore use a "matrix product" notation and say that $L^{(m+1)} = L^{(m)} \cdot W$. One matrix "product" takes $O(n^3)$ and therefore one may compute the all-pairs shortest path matrix $L^{(n-1)}$ with $O(n^4)$ operations. Note however that we are not actually interested in the intermediate $L^{(m)}$ matrices! If n is, say, 738, then instead of iterating $L^{(m+1)} = L^{(m)} \cdot W$ to compute

$$L^{(1)}, L^{(2)}, L^{(3)}, L^{(4)}, \ldots, L^{(736)}, L^{(737)}$$

we can just repeatedly square W and obtain the sequence

$$L^{(1)}, L^{(2)}, L^{(4)}, L^{(8)}, \ldots, L^{(512)}, L^{(1024)}.$$

Since all $L^{(m)}$ are equal to L for $m = n - 1$ onwards, we have that $L^{(1024)} = L^{(737)} = L$ and we don't have to hit the target exactly—it's OK to keep squaring until we exceed it. Since the number of steps in the second sequence is only $\lceil \lg n \rceil$ instead of $n - 1$, this method brings the cost down to $O(n^3 \lg n)$—also known as $O(V^3 \lg V)$ since n, the order of the matrix, is the number of vertices in the graph.

This repeated matrix squaring algorithm for solving the all-pairs shortest paths problem with negative edges is easy to understand and implement. At $O(V^3 \lg V)$ it beats the iterated Bellman-Ford whenever $O(E) > O(V \lg V)$, although it loses on very sparse graphs where the reverse inequality holds. Another option for the case with negative edges, not discussed in these notes, is the Floyd-Warshall algorithm, which is $O(V^3)$ (see textbook). Even better, however, is Johnson's algorithm, discussed next.

6.5.2 Johnson's algorithm for all-pairs shortest paths

The Johnson algorithm, like the iterated matrix squaring, the iterated Bellman-Ford and the Floyd-Warshall algorithms, runs on graphs with negative edges but without negative cycles. The clever idea behind Johnson's algorithm is to turn the original graph into a new graph that only has non-negative edges *but the same shortest paths*, and then to run Dijkstra on every vertex.

The Johnson algorithm includes one pass of Bellman-Ford at a cost of $O(VE)$; but since the final step of iterating Dijkstra on all vertices costs, as we said, at least $O(V^2 \lg V + VE)$ even in the most favourable case of using Fibonacci heaps, we are not increasing the overall complexity with this single pass of Bellman-Ford. Note that $O(VE)$ is $O(V^3)$ for dense graphs but it may go down towards $O(V^2)$ for sparse graphs; in the latter case, therefore, Johnson's algorithm reduces to $O(V^2 \lg V)$, offering a definite improvement in asymptotic complexity over Floyd-Warshall's $O(V^3)$.

The most interesting part of the Johnson algorithm is the **reweighting** phase in which we derive a new graph without negative edges but with the same shortest paths as the original. This is done by assigning a new weight to each edge; however, as should be readily apparent, the simple-minded strategy of adding the same large constant to the weight of each edge would not work, as that would change the shortest paths[17].

We introduce a fake source vertex s, and fake edges (s, v) of zero weight from that source to each vertex $v \in V$. Then we run Bellman-Ford on this augmented graph, computing distances from s for each vertex. As a side effect, this tells us whether the original graph contains any negative weight cycles[18]. If there are no negative cycles, we proceed. If we indicate as $v.\delta$ the shortest distance from s to v as computed by Bellman-Ford (and left in $v.d$ at the end of the procedure), we assign to each edge (u, v) of weight $w(u, v)$ a new weight $\hat{w}(u, v) = u.\delta + w(u, v) - v.\delta$. Adding the above equality to the triangle inequality $u.\delta + w(u, v) \geq v.\delta$, we get

$$\hat{w}(u, v) + u.\delta + w(u, v) \geq u.\delta + w(u, v) - v.\delta + v.\delta$$

and, simplifying,

$$\hat{w}(u, v) \geq 0,$$

meaning that the reweighted edges are indeed all non-negative, as intended.

Let's now show that this reweighting strategy also preserves the shortest paths of the original graph. Let $p = v_0 \to v_1 \to \ldots \to v_k$ be any path in the original graph, and let $w(p) = w(v_0, v_1) + w(v_1, v_2) + \ldots + w(v_{k-1}, v_k)$ be its weight. The weight of p in the reweighted graph is $\hat{w}(p) = w(p) + v_0.\delta - v_k.\delta$, since all the $v.\delta$ of the intermediate vertices cancel out. Therefore, once we fix the start and end vertices v_0 and v_k, whatever path we choose from one to the other (including ones that do not visit those intermediate vertices v_1, v_2 etc, and ones that visit other vertices altogether), the weight of the path in the old and in the reweighted graph differ only by a constant (namely $v_0.\delta - v_k.\delta$), which implies that the path from v_0 to v_k that was shortest in the original graph is still the shortest in the new graph, QED.

[17] Just imagine adding such a large constant that the original weights become irrelevant.

[18] It does if and only if the augmented graph does, since the vertex and edges we introduced do not add or remove any cycles to the original graph.

So, after reweighting, we can run Dijkstra from each vertex on the reweighted graph, at the stated costs, and obtain shortest paths. The actual lengths of these shortest paths in the original graph are recovered in constant cost for each path: $w(p) = \hat{w}(p) - v_0.\delta + v_k.\delta$. The total cost of this final phase is therefore proportional to the number of shortest paths, $O(V^2)$, and it does not affect the overall asymptotic cost.

6.6 Maximum flow

Textbook

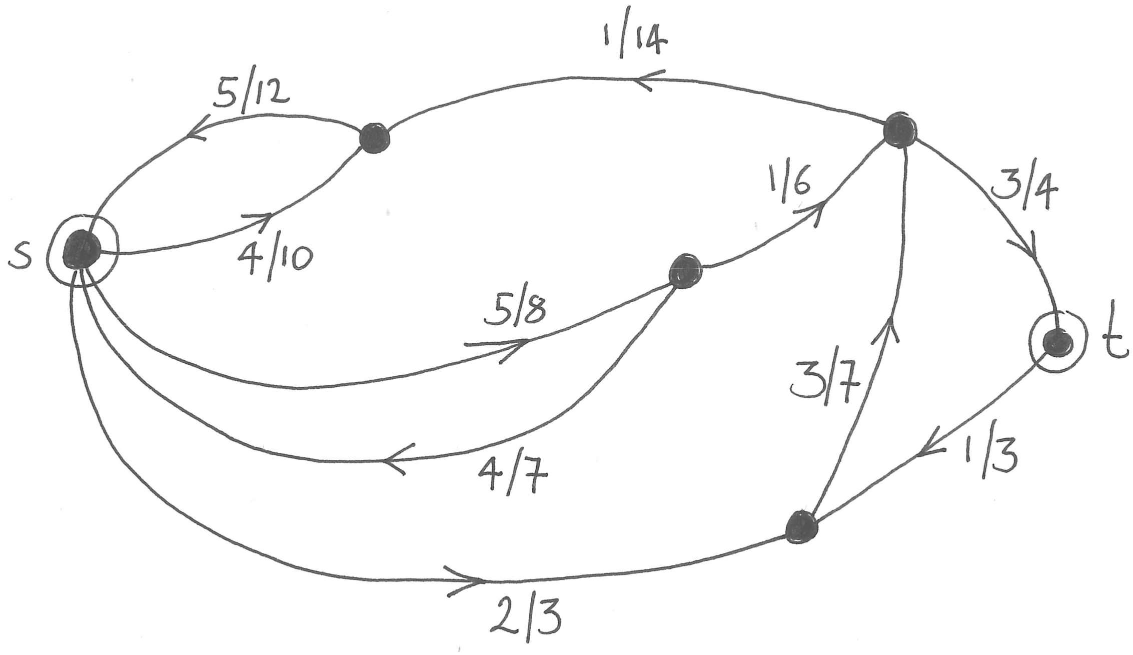
Study chapter 26 in CLRS3.

If the edge weights are taken to represent capacities (how many cubic metres per second, or cars per hour, or milliamperes, can flow through this link?) an interesting question is to determine the maximum flow that can be established between the two designated source and sink vertices[19]. Whatever the entity that is flowing, be it vodka, IP packets or lemmings, we assume in this model that it can't accumulate in the intermediate vertices[20], that the source can generate an infinite supply of it, and that the sink can absorb an infinite amout of it, with the only constraints to the flow being the capacities of the edges. Note that the actual flow through an edge at any given instant is quite distinct from the maximum flow that that edge can support. The two values are usually separated by a slash[21]: for example an edge labelled $3/16$ is currently carrying a flow of 3 and has a maximum capacity of 16.

[19]The multiple-sources and multiple-sinks problem is trivially reduced to the single-source and single-sink setting: add a dummy super-source linked to every actual source by edges of infinite capacity and proceed similarly at the sink side with a dummy super-sink.

[20]Therefore the total flow into an intermediate vertex is equal to the total flow out of it.

[21]Not to be taken as meaning division but rather "out of".



In order to deal with flow problems, we introduce a few useful concepts and definitions.

A **flow network** is a directed graph $G = (V, E)$, with two vertices designated respectively as source and sink, and with a **capacity** function $c : V \times V \to \mathbb{R}^+$ that associates a non-negative real value to each pair of vertices u and v. The capacity $c(u, v)$ is 0 if and only if there is no (u, v) edge; otherwise it is noted as the number after the slash on the label of the (u, v) edge, representing the maximum possible amount that can flow directly from u to v through that edge. A **flow** is a function $f : V \times V \to \mathbb{R}$ that associates a (possibly negative) real value to each pair of vertices, subject to the following constraints:

- $f(u, v) \le c(u, v)$;
- $f(u, v) = -f(v, u)$;
- and "flow in = flow out" for every vertex except source and sink.

Note that the flow $f(x, y)$ can be nonzero even if there is no edge from x to y: for that it is enough to have an edge from y to x carrying the opposite amount. Conversely, when an (x, y) edge and a (y, x) edge both exist and each carry some amount, say a and b respectively, then the flow between x and y is meant as the "net" flow: $f(x, y) = a - b$ and $f(y, x) = b - a$. This is consistent with the constraints presented above.

The **value** of a given flow f is a scalar value $|f| \in \mathbb{R}$ equal to the total flow out of the source: $|f| = \sum_{v \in V} f(s, v)$. Owing to the properties above, it is also equal to the total flow into the sink.

The **residual capacity** $c_f : V \times V \to \mathbb{R}^+$ is a function that, given a flow network G and a flow f, indicates for each pair of vertices the extra amount of flow that can be pushed between the vertices of the pair through the edge(s) that directly join them, if any, on top of what is currently flowing, without exceeding the capacity constraint of the 0, 1 or 2 edges that directly join the two vertices. So, for any two vertices u and v, we have that $c_f(u, v) = c(u, v) - f(u, v)$. Note that we do not require the original graph to have an (u, v) edge in order for $c_f(u, v)$ to be greater than zero: if the graph has a (v, u) edge that is currently carrying, say, 4/16, then one can, with respect to that situation, increase the flow from u to v by up to 4 units by *cancelling* some of the flow that is already being carried in the opposite direction—meaning that there is, in that situation, a positive residual capacity from u to v of $c_f(u, v) = 4$ (the latter not to be confused with $c(u, v)$, which instead stays at zero throughout in the absence of a (u, v) edge). Note also that the residual capacity $c_f(u, v)$ can exceed the capacity $c(u, v)$ if the current net flow from u to v is negative: first, by increasing the flow in the direction from u to v, you reduce the absolute value of the negative flow, eventually bringing $f(u, v)$ to zero; then you may keep on increasing the flow until you reach $c(u, v)$.

Exercise 66
Draw suitable pictures to illustrate and explain the previous comments. *Very easy; but failure to do this, or merely seeing it done by someone else, will make it unnecessarily hard to understand what follows.*

The **residual network** $G_f = (V, E_f)$ is the graph obtained by taking all the edges with a strictly positive residual capacity c_f and labelling them with that capacity. It is itself a flow network, because the residual capacity is also non-negative.

An **augmenting path** for the original network is a sequence of edges from source to sink in the residual network. The residual capacity of that path is the maximum amount we can push through it, equal of course to the residual capacity of its lightest edge.

We are now ready to describe how to find the maximum flow—or, making use of the above definitions for greater accuracy, to describe how to find, among all the possible flows on the given flow network G, the one whose value $|f|$ is the highest.

6.6.1 The Ford-Fulkerson maximum flow method

We call Ford-Fulkerson a "method" rather than an "algorithm" because it is in fact an algorithm blueprint whose several possible instantiations may have different properties and different asymptotic complexity. The general pattern is simply to compute the residual network, find an augmenting path on it, push the residual capacity through that path (thereby increasing the original flow) and repeat while possible. The algorithm terminates when there is no augmenting path in the residual network.

```
def fordFulkerson(G):
    initialize flow to 0 on all edges
    while an augmenting path can be found:
        push as much extra flow as possible through it
```

Contrary to most other graph algorithms examined in this course, the running time does not just depend on the number of vertices and edges of the input graph but on the *values* of its edge labels.

At each pass through the loop, the value of the flow increases. Since all edge capacities are finite, the maximum flow is finite. The method might fail to terminate if it took an infinite number of steps for the flow to converge to its maximum value. For this to happen, the increments must become infinitesimal. Since the increment in line 3 is the greatest possible given the chosen augmenting path, its value is obtained as a difference of previously computed capacities (since we start from a null flow in line 1). Therefore, if all capacities are integers, every increment will also be an integer and the flow will increase by at least 1 at each pass through the loop, thereby bringing the algorithm to completion in a finite number of steps.

So, integer capacities for all edges are a sufficient condition for termination. Conversely, it is indeed possible[22] to construct elaborate graphs with irrational capacities where Ford-Fulkerson does not terminate.

As for running time in the case of integer capacities, line 3 changes the flow of each edge along the augmenting path, so its running time is proportional to the length of the path and is therefore bounded by the length in edges of the longest path without cycles, $|V| - 1$. On the other hand, finding such a path, if it exists, for example with Breadth First Search or Depth First Search, will cost up to $O(V + E)$. Each iteration thus costs $O(E)$. In the worst case, each pass increases the flow only by 1, so the loop may be executed up to $|f^*|$ times (the value of the maximum flow). So, in total, the running time is bounded by $O(E \cdot |f^*|)$. Note how this growth rate is independent and therefore potentially higher than any polynomial function of $|V|$ and $|E|$, since the capacity values on the edge labels are independent of the shape or size of the graph.

With a maximum flow of high numerical value and a consistently pessimal choice of augmenting path, Ford-Fulkerson might take billions of operations to compute a solution even on a tiny graph with only 5 edges. Specific implementations of Ford-Fulkerson, however, can improve on this sad situation by defining some useful criterion for selecting an augmenting path in line 2, as opposed to leaving the choice entirely open. The so-called **Edmonds-Karp** algorithm, for example, derived from Ford-Fulkerson by choosing as augmenting path through a breadth-first search, thereby picking one with the smallest number of edges, can be proved to converge in $O(VE^2)$ time.

Another possibility, also suggested by Edmonds and Karp, is to select the augmenting path based on the greatest residual capacity. There are still others.

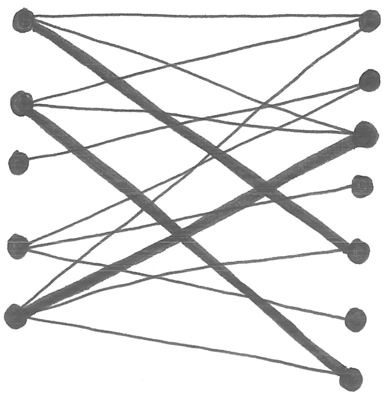
6.6.2 Bipartite graphs and matchings

A **matching** in a bipartite graph[23] is a collection of edges such that each vertex of the graph is the endpoint of at most one of the selected edges. A maximal matching is then obviously as large a subset of the edges that has this property as is possible. Why might one want to find a matching? Well, bipartite graphs and matchings can be used to represent many resource allocation problems. For example, vertices on the left might be final-year students and vertices on the right might be final year project offers. An edge would represent the interest of a student in a particular project. The maximum cardinality matching would be the one satisfying the greatest number of student requests,

[22]Possible, but far from trivial or intuitive: Ford and Fulkerson's own original example had 10 vertices and 48 edges. The minimal example appeared years later in Uri Zwick, "The smallest networks on which the Ford-Fulkerson maximum flow procedure may fail to terminate", *Theoretical Computer Science* 148, 165-170 (1995).

[23]As you may recall, the definition of *bipartite graph* was given in section 6.1.1 on page 109.

assuming that no two students can choose the same project and that no student can do more than one project.



A simple direct search through all possible combinations of edges would provide a direct way of finding maximal matchings, but would have costs growing exponentially with the number of edges in the graph. This, even for small graphs, is not a feasible strategy.

A particularly smart way of solving the bipartite matching problem is to recast it as a maximum flow problem in the following way. Introduce a fake super-source to the left of the bipartite graph and a fake super-sink to the right. Add edges of unit capacity from the super-source to all the left side vertices and from all the right side vertices to the super-sink. This is very similar to the construction mentioned in footnote 19 (section 6.6, page 129) to transform a multiple-sources and multiple-sinks flow problem into one with a single source and a single sink. Here, however, we give all edges (including the ones touching the super-source and super-sink) unit capacity. If we constrain all flows to be integers, assigning unit capacities to the edges ensures that no two edges will start or end on the same internal vertex and therefore that all flows will also be valid matchings. All that's left is then to find the maximum flow (itself a valid matching, and necessarily the one with the greatest number of edges) using the Ford-Fulkerson method.

A more complicated variant, which we won't discuss, is that of the *weighted* matching problem: this is where a bipartite graph has the edges labelled with values and it is necessary to find the matching that maximizes the sum of weights on the edges selected.

Chapter 7

Parallel algorithms

Chapter contents

Dynamic multithreading. Work and span. Greedy scheduler. Determinacy races.
Expected coverage: about 1 lecture.

Textbook

Study chapter 27 in CLRS3.

Moore's law, about transistor count in state-of-the-art ICs doubling every two years, surprisingly continues to hold after decades; but clock speeds no longer increase at the same rate. To what extent can the abundance of silicon gates be used to compensate for lack of progress in processor speed? While it's obvious that a processor with $4\times$ the clock frequency can be expected to execute a CPU-bound task four times faster, what can be said of a processor with $4\times$ the number of cores?

Imagine we have to perform a CPU-bound job that requires 1 trillion (10^{12}) machine instructions. A processor that executes 1 billion instructions per second will take 1000 seconds (about 17 minutes) to complete it. If a $10\times$ faster processor existed[1], it would only take 100 seconds, or just over one and a half minute. What if instead we had 10 processors (or 10 cores), each with the same power as the original? For the whole 1-trillion-instruction job to take only 100 seconds, each of the 10 processors would have to run solidly for the whole 100 seconds, in order to execute its quota of 100 billion instructions. If any of the 10 processors were idle for part of that time, 100 seconds would not be sufficient to execute an aggregate 1 trillion instructions[2]. In order to keep each processor busy we need to:

- partition the work equally among all the available processors; and

[1]But maybe it doesn't exist—and, even if it did, it would probably cost too much.
[2]We ignore overheads for scheduling etc.

- ensure that no processor is ever tied up waiting for a result that another processor still has to produce.

Whether we are able to achieve this or not is a function of both the problem itself and of the algorithm we write to solve it. Only in the most favourable cases will we be able to achieve a speedup equal to the number of processors; in general, that will only be an upper bound. We may not be able to achieve the bound if:

- we cannot split the problem into as many independent pieces as there are processors; if we only manage to split the problem into 4 pieces, and there are 10 processors, at least 6 of them will be idle at any time;

- we can split the problem into as many pieces as there are processors, but the pieces are of uneven size; in that case some processors will finish before others and will have to sit there twiddling their thumbs;

- we can split the work into many pieces, but those pieces depend on each other's results; so a processor, despite having an allocated piece of problem it could work on, must wait idly until another processor finishes its piece and supplies a result that is needed as input.

Any time one of the 10 processors is idle, for any of the above reasons, it does not contribute towards the quota of 1 trillion instructions that need to be executed to finish the job. Clearly, then, it will take more than 100 seconds to get to completion, and the speedup factor gained by having $10\times$ more processors will be less than $10\times$.

Taking an example from cryptography, brute-force key-search is easy to parallelize: you can partition the key space into arbitrarily many regions and give each region to a single processor. Trying the keys in one region is completely independent from trying the keys in another region, so each processor can work fully independently of the others. So long as the regions are of equal size, the speedup factor to search the whole key space[3] is equal to the number of processors assigned to the job.

Taking another example from cryptography that goes in the opposite direction, sometimes we build on purpose a function that is hard to parallelize. If, as you should, you salt and hash your passwords in the back-end, you want the function that goes from password to hash to take a long time to execute, so that attackers who learn the hash and salt still won't be able to find the original password by brute force. To do that, you might actually compute the hash of the hash of the hash ... a million times. And you like the fact that there is no easy way to parallelize the computation of $h^{1,000,000}(\mathit{salt}, \mathit{password})$, in which each of the million individual hash computations in the chain requires the result of the previous one.

So what do we do in this chapter? Well, we will barely have time to scratch the surface, but our aim is to introduce some intellectual tools to reason about parallel algorithms and evaluate their performance in a quantitative way.

[3]Of course, as soon as the sought key is found, the whole process is terminated early—and there's no telling whether the time to actually find the key is decreased by the same factor. But it all makes sense again when talking of *worst case* execution times (search whole keyspace to find that the key was the very last one to be tried).

7.1 Programming model: dynamic multithreading

There are various possible architectures for parallel computing: we won't explore them all. We shall model the one that is common in the mainstream microprocessors that are popular at the time of writing, where several processor cores have access to the same memory (a *shared memory* architecture, as opposed to the *distributed memory* case in which each processor has its own local memory not accessible to the others).

As for the programming model, with *static threading* there is a fixed pool of threads, each representing a (possibly virtual) processor, all having access to a common memory. The programmer is responsible for partitioning the work among the available threads, and doing so in a balanced way is difficult and error-prone. A middleware layer, the *concurrency platform*, can take care of this task and give the programmer the illusion of having as many processors as necessary. Behind the scenes, the concurrency platform allocates jobs to threads in a way that balances the load. This is *dynamic multithreading*, the model we are going to discuss.

In the formulation of the CLRS3 textbook, we describe parallel algorithms by adding three special concurrency keywords to the pseudocode: `spawn`, `sync`, `parallel`, with the following meanings.

`spawn`: An optional prefix to a procedure call statement. Call the indicated procedure, but in a separate thread. In the current thread, just continue. (If you were assigning the result of the procedure call to a variable, as in `y = spawn f(x)`, then the variable will retain the old value until the spawned procedure returns.)

`sync`: Block execution until all the threads previously spawned from within the current procedure[4] have completed.

`parallel`: An optional prefix to the standard looping keyword `for` that causes each iteration of the loop to take place in its own thread.

Note that the `spawn` and `parallel` keywords *allow* the concurrency platform to execute pieces of code in parallel, but they do not require it. Programs with those keywords can generate arbitrarily many threads and it is up to the scheduler to assign them to actual processors.

Removing all occurrences of these three keywords from the pseudocode yields the serial version of the algorithm.

A *strand* is a sequence of statements that does not contain concurrency keywords. We can visualize the execution of the algorithm as a graph (a DAG) in which each strand is a vertex[5] and the edges between vertices represent procedure calls, returns from calls, procedure spawns, return from spawns, as well as the act of continuing sequentially within the same procedure instance.

[4]Including threads spawned by spawned children down to arbitrary depths.

[5]In case of recursive calls, a given strand in the source code may appear in the graph as several vertices, one per instance of the procedure.

7.2 Performance analysis

7.2.1 Definitions

$P =$ number of processors.

$T_P =$ running time of a computation on P processors.

$T_1 =$ **work** of a computation: time to execute that computation on a single processor (hence T_P with $P = 1$).

$T_\infty =$ **span** of a computation: execution time of the critical path in the DAG[6]. Indicated as the time for $P = \infty$ because, if we had infinite processors available, the time to execute the computation would be determined by the critical path.

From these definitions come two seemingly obvious (but actually deceptively subtle) laws that give lower bounds on the time it takes to execute the computation on P processors.

Work law: $T_P \geq \frac{T_1}{P}$

The running time on P processors can't be any shorter than if all of them work all the time, with no wasted cycles.

Span law: $T_P \geq T_\infty$

The running time on P processors can't be any shorter than the time it would take on arbitrarily many processors.

Further definitions:

- The **speedup** of a computation, when run on P processors, is the multiplying factor that says how much faster the computation goes on P processors compared to when it runs on just one processor: speedup $= T_1/T_P$.
- We call it **linear speedup** iff the speedup is proportional to the number of processors used, i.e. iff $\frac{T_1}{T_P} = \Theta(P)$.
- We call it **perfect linear speedup** iff the speedup is exactly equal to the number of processors used, i.e. iff $\frac{T_1}{T_P} = P$.
- The **parallelism** of a computation is defined as the work divided by the span: T_1/T_∞. Given the definition of speedup, we can intuitively interpret the parallelism of a computation as the maximum speedup factor one can possibly obtain by providing arbitrarily many processors.

  We can prove that the parallelism is also:

[6]Equivalent to the maximum among the execution times of all the paths in the DAG.

  - The average amount of work (in terms of number of active processors) that can be performed in parallel at each step of the critical path.
  - A limit on the number of processors that you can add before losing the ability to achieve perfect linear speedup.

- The **parallel slackness** of a computation, when run on P processors, is the factor by which the parallelism of the computation exceeds the parallelism of the machine (i.e. the number of available processors). By definition it's therefore $\frac{T_1/T_\infty}{P} = \frac{T_1}{P \cdot T_\infty}$. A high parallel slackness is good for achieving high speedup factors, i.e. "not wasting processor time".

In the next section we shall see that, with high parallel slackness (i.e. if you give it some margin, meaning a computation that is highly parallelizable), a good scheduler can achieve nearly perfect linear speedup.

Exercise 67
Prove that T_1/T_∞ is the average amount of work that can be performed in parallel at each step of the critical path.

Exercise 68
Prove that, if $P > T_1/T_\infty$, the computation cannot achieve perfect linear speedup.

7.2.2 Scheduling

In this programming model, the programmer doesn't say which processor executes what strand: the scheduler decides. The scheduler sees requests for parallelism as they come but doesn't know in advance what procedure is about to be spawned or when a spawned procedure will return. What policy should the scheduler follow? How do we know if it's any good?

A simple but effective strategy is that of the *greedy scheduler*: in each step, assign as many strands to processors as possible. We are going to prove that it's reasonably good.

First, assume that each strand takes unit time[7]. At each time unit there will be zero or more strands ready to be executed and zero or more strands blocked in a `sync` operation. We call each step **complete** if the number of strands ready to be executed is at least P; **incomplete** otherwise.

[7]If not, you could always chop up longer strands into chains of unit-time strands.

Theorem

With a greedy scheduler, the time T_P to run a computation on P processors is bounded by

$$T_P \quad \leq \quad \frac{T_1}{P} + T_\infty.$$

Proof: Each step is either complete or incomplete. The total number of complete steps (in which all processors are assigned a work unit) cannot exceed the total number of work units divided by the number of processors[8]:

$$\text{complete steps} \quad \leq \quad T_1/P.$$

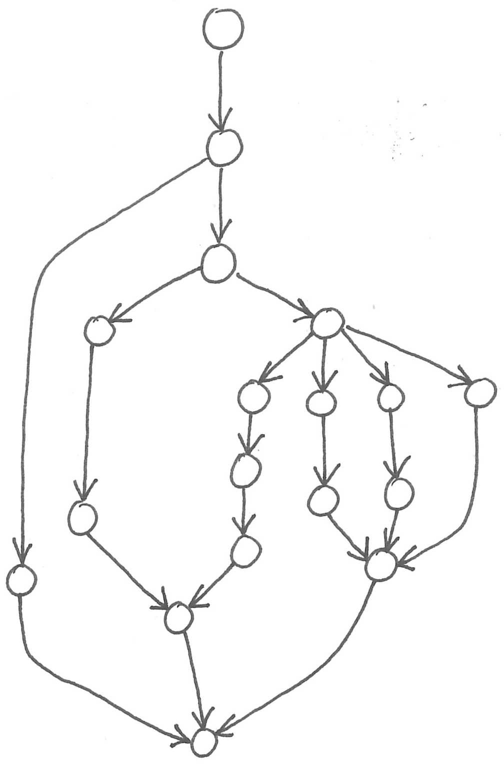
On the other hand, consider an incomplete step. There are fewer than P strands ready to execute. Each strand ready to execute is a source vertex in the execution DAG[9]. Take the longest path from a source to a sink in the DAG: after execution of the incomplete step, that path will be shorter by at least one, because the source vertex will have been executed (and the next vertex will be a source at that point). Therefore each incomplete step shortens the longest path from source to sink by one. Since the longest path is the critical path, the number of incomplete steps is bounded by the span:

$$\text{incomplete steps} \quad \leq \quad T_\infty.$$

Each step is either complete or incomplete, hence by adding the two inequalities we obtain the thesis.

[8]This is a kind of dual of the work law.

[9]Because, if it had any incoming arrows, it would be waiting for some other strand to complete and it thus wouldn't be ready to execute.



> **Theorem**
>
> The performance of a greedy scheduler is within 2× of optimal.

Proof: given the bounds imposed by the work law and the span law, even the optimal scheduler cannot take less time than $T_P^* = \max(\frac{T_1}{P}, T_\infty)$. From the previous theorem,

$$T_P \quad \leq \quad \frac{T_1}{P} + T_\infty \quad \leq \quad 2\max(\frac{T_1}{P}, T_\infty) \quad = \quad 2T_P^*$$

QED.

> **Theorem**
>
> If you use a greedy scheduler and have sufficiently high parallel slackness, you get almost perfect linear speedup.

Proof: High parallel slackness means

$$\frac{T_1}{P \cdot T_\infty} \gg 1 \quad \Rightarrow \quad \frac{T_1}{P} \gg T_\infty \quad \Rightarrow \quad \frac{T_1}{P} + T_\infty \approx \frac{T_1}{P}.$$

Plugging this approximation into the first theorem above, and combining with the work law,

$$T_P \leq \frac{T_1}{P} + T_\infty \approx \frac{T_1}{P} \leq T_P \quad \Rightarrow \quad \frac{T_1}{P} \approx T_P \quad \Rightarrow \quad \frac{T_1}{T_P} \approx P$$

QED.

7.2.3 Race conditions

A parallel algorithm is **deterministic** if its results on a given input do not depend on the order in which instructions are scheduled; it is **nondeterministic** if the results on the same input might change between executions (by virtue of the scheduler selecting different combinations of ready threads in different runs).

For everyone's sanity, especially yours, you are strongly advised not to write nondeterministic programs. People do, but usually not on purpose. The main cause of nondeterminism is a *race condition*—a fertile source of spectacular bugs that are very hard to reproduce and hence debug.

A **determinacy race** occurs when two or more threads access the same memory location in parallel and at least one performs a write.

Imagine two parallel threads accessing variable x, initialized to 0. Each thread consists of the code `x = x + 1`. What is going to be the value in x after executing and syncing the two threads? It might be reasonable to expect that the answer will be 2 regardless of the order in which the threads execute.

However, by our definition above, this situation contains a determinacy race. Although the problem is not obvious when looking at the high level instructions, when each assignment is expanded into its machine code components, and when we consider that the machine code instructions for the high level assignment may not be executed atomically, we see where the race might occur. Each `x = x + 1` assignment expands into a machine code sequence similar to

```
LOAD R from x
INC R
STORE R into x
```

where R is a processor register and x is a memory address. If these low-level instructions for the two threads are interleaved, we might have a situation where both threads read x while it is still 0, both increment it to 1 and both write it back as 1. The final result in x is 1, different from what would have happened if all the instructions of one thread had been executed before those of the other.

Exercise 69
Construct a minimal case of determinacy race with two threads accessing variable x but only one of them writing to it. Show at least two ways of sequencing the machine code instructions that will cause different results.

There are ways of handling races using synchronization primitives such as the mutex. The strategy we use here is simply to ensure that strands that may execute in parallel are independent. Any code with a determinacy race will be deemed illegal (for this programming model, it is possible to determine statically whether a code has a determinacy race or not).

Exercise 70
If you can solve this one without help, you are pretty good.
As written, several listings in chapter 27 of CLRS3 (third printing) contain an unintended determinacy race. Which listings? Where is the race? How could such a bug occur?
(When I noticed this, I wrote to Professor Leiserson, who wrote the chapter, and he confirmed I had found a severe bug, which would be fixed in the fourth printing.)

7.3 Case study: chess program

The CLRS3 textbook reports an interesting tale from the development of the award-winning chess program *Socrates.

The development machine had $P = 32$ processors. The target machine, not available to the developers, was to have 512. The running time on 32 processors was 65 seconds. One of the programmers found a way to bring this down to 40 seconds! Major improvement, right? Yet this change to the code was not incorporated in the final version, because an analysis based on work and span showed that it would have actually slowed down the execution on the target machine. How is this possible? And how could they figure it out before being able to try the code on the $P = 512$ machine?

The first theorem of the greedy scheduler tells us that $T_P \leq \frac{T_1}{P} + T_\infty$: treating the inequality as an equation to obtain an approximation for the running time instead of a bound, we get $T_P = \frac{T_1}{P} + T_\infty$; solving it for the span we get $T_\infty = T_P - \frac{T_1}{P}$.

With the original code, the work was $T_1 = 2048$, with $T_{32} = 65$. The approximation for the span thus gave $T_\infty = 65 - 2048/32 = 65 - 64 = 1$. Extrapolating these results to $P = 512$ gave $T_{512} = 2048/512 + 1 = 4 + 1 = 5$.

After the change in the code, instead, the work was $T'_1 = 1024$, with $T'_{32} = 40$. The approximation for the span gave $T'_\infty = T'_P - \frac{T'_1}{P} = 40 - 1024/32 = 40 - 32 = 8$. Extrapolating to 512 processors gave $T'_{512} = 1024/512 + 8 = 2 + 8 = 10$. In other words, the "optimization" would have made the code twice as slow on the target machine!

We note that the original code had pretty high parallelism of $T_1/T_\infty = 2048/1 = 2048$, which gave a parallel slackness of at least 4 even on the highly parallel target machine. The rearranged code, instead, exhibited a much lower parallelism of $1024/8 = 128$ giving a pathetic parallel slackness factor of just 1/4 on the target machine. This meant that the hardware parallelism of the target machine could not be fully exploited by the changed code and that's why the original code performed better.

The moral of this story is to trust the work and span metrics more than raw execution times when evaluating and predicting the performance of parallel algorithms.

Chapter 8

Geometric algorithms

Chapter contents

Intersection of segments. Convex hull: Graham's scan, Jarvis's march.
Expected coverage: about 1 lecture.

Textbook

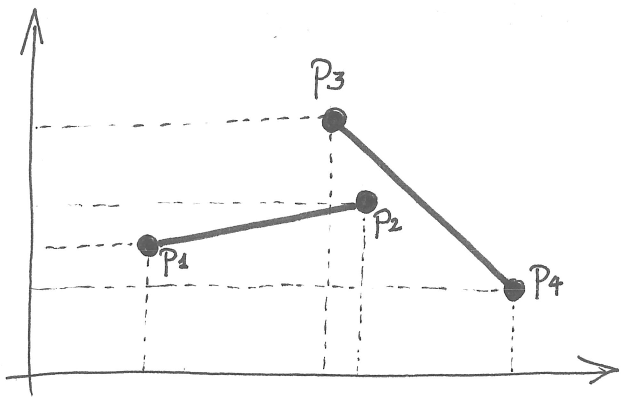
Study chapter 33 in CLRS3.

The topics included here show off some of the techniques that can be applied and provide some basic tools to build upon. Many more geometric algorithms are presented in the computer graphics courses. Large scale computer aided design and the realistic rendering of elaborate scenes will call for plenty of carefully designed data structures and algorithms.

Note that in this overview chapter we work exclusively in two dimensions: upgrading the algorithms presented here to three dimensions will *not*, in general, be a trivial extension.

8.1 Intersection of line segments

Our first problem is, given two line segments $\overline{p_1p_2}$ and $\overline{p_3p_4}$ (4 endpoints p_1, p_2, p_3, p_4 and 8 coordinates $x_1, y_1, x_2, \ldots, y_4$), to determine whether they intersect—that is, whether they have any points in common. Since the input is of fixed size (8 real numbers), it should come as no surprise that this can be solved in $O(1)$. To make matters more interesting, then, we shall show how to compute this result efficiently, without using any trigonometry or even division. To that effect, let's first take a little detour.



8.1.1 Cross product

Our friend here, and for the rest of the chapter, is the cross product. A point $p = (x, y)$ in the Cartesian plane defines a point-vector $\vec{p}$ from the origin $(0, 0)$ to point p. The cross product $\vec{p_1} \times \vec{p_2}$ is defined as a third vector $\vec{p_3}$, orthogonal to the plane of $\vec{p_1}$ and $\vec{p_2}$, of magnitude given by the area of the parallelogram of sides $\vec{p_1}$ and $\vec{p_2}$, and such that $\vec{p_1}, \vec{p_2}$ and $\vec{p_3}$ follow the "right hand rule".

Exercise 71

- Draw a 3D picture of $\vec{p_1}, \vec{p_2}$ and $\vec{p_3}$.
- Draw a 2D picture of $\vec{p_1}, \vec{p_2}$ and the parallelogram.
- Prove that the absolute value of the determinant of the matrix $\begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix}$, equal to $x_1y_2 - x_2y_1$, gives the magnitude of $\vec{p_3}$ and that its sign says whether $\vec{p_3}$ "comes out" of the plane ($\odot$) or "goes into" it ($\otimes$).

Having completed the exercise, you will readily understand that the sign of the cross product $\vec{p_1} \times \vec{p_2}$ tells us whether the (shortest) rotation that brings $\vec{p_1}$ onto $\vec{p_2}$ is positive (= anticlockwise) or negative. In other words, by looking at the sign of $\vec{p_1} \times \vec{p_2}$, we can answer the question "on which side of $\vec{p_1}$ does $\vec{p_2}$ lie?". This can be used to provide efficient solutions to several problems. Let's get back to one of them.

8.1.2 Intersection algorithm

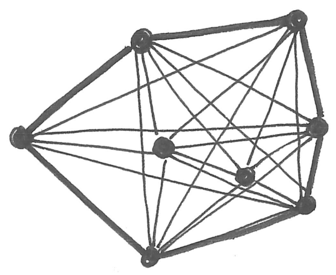
Do the two line segments $\overline{p_1p_2}$ and $\overline{p_3p_4}$ intersect? (Note that we are no longer dealing with point-vectors.) We examine a tree of possibilities. Take the two endpoints p_1 and p_2 of the first segment and consider their position with respect to the second segment $\overline{p_3p_4}$. Do p_1 and p_2 lie on the same side or on opposite sides of $\overline{p_3p_4}$? If they lie on the same side, it's clear that the two segments **don't intersect**. If they lie on opposite sides that's a good start but[1], if we are unlucky, they only intersect the *continuation* of $\overline{p_3p_4}$, not the segment proper. So we should also turn the tables and verify the position of p_3 and p_4 with respect to $\overline{p_1p_2}$. If these are also on opposite sides, then the two segments **intersect**; if they are on the same side, the segments **don't intersect**.

In the above we have nonchalantly ignored the boundary case in which one of the endpoints being examined lies precisely on the line containing the other segment. When this happens, the relevant cross product is $\vec{0}$. We must detect all such cases and then check whether the point, as well as being collinear with the endpoints of the segment, also lies *between* them; if it does, in any of the possible configurations, the verdict will be that the segments **intersect**; if this never happens, instead, the segments **don't intersect**.

It is instructive to note how a substantial portion of the programming effort of producing a correct algorithm must go into the handling of this boundary case. This is not an isolated occurrence in computer graphics.

8.2 Convex hull of a set of points

The convex hull of a collection Q of points in the plane is the smallest convex[2] polygon such that all the points in Q are either inside the polygon or on its boundary.



Exercise 72
Sketch an algorithm (not the best possible: just any vaguely reasonable algorithm) to compute the convex hull; then do a rough complexity analysis. Stop reading until you've done that. *(Requires thought.)*

We present two algorithms[3] for building the convex hull of a set Q of n points. The first, Graham's scan, runs in $O(n \lg n)$ time. The second, Jarvis's march, takes $O(nh)$ where h is the number of vertices of the hull. Therefore the choice of one or the other

[1]Micro-exercise in mid-sentence: sketch in the margin a case in which they do and yet there is no intersection.

[2]A convex polygon is one in which, given any two points inside the polygon (boundary included), no point of the segment that joins them is outside the polygon. For examples of non-convex polygons think of shapes with indentations such as a star or a cross.

[3]Several others have been invented—see the textbook.

depends on whether you expect most of the points to be on the convex hull or strictly inside it.

Both methods use a technique known as "rotational sweep": they start at the lowest point and examine the others in order of increasing polar angle[4] from a chosen reference point, adding vertices to the hull in a counterclockwise fashion. This leads to an interesting sub-problem: how can you efficiently sort a group of point-vectors by the angles that they form with the positive x axis?

The simple-minded answer is to compute that polar angle (call it $\theta(\vec{p})$) for each point-vector $\vec{p}$ using some inverse trigonometric function of $\vec{p}$'s coordinates and then to use it as the key for a standard sorting algorithm. There is, however, a better way. Here, too, the cross product is our friend. Actually...

Exercise 73
How can you sort a bunch of points by their polar coordinates using efficient computations that don't include divisions or trigonometry? (Hint: use cross products.) Don't read beyond this box until you've found a way.

Hey! I said stop reading! I know, the eye is quick, the flesh is weak... but just do the exercise first, OK? You'll be much better off at the exam if you work out this kind of stuff on your own *as you go along.*

Anyway, the trick is that there is no need to *compute* the actual angles in order to sort the point-vectors by angle—all that is needed is a plug-in replacement for the core comparison activity of "is this point-vector's polar angle greater than that of that other point-vector?". This, regardless of the actual values of the angles, is equivalent to asking whether a point-vector is to the left or the right of another; therefore the question is answered by the sign of their cross product, as follows:

$$\theta(\vec{p}) \lesseqqgtr \theta(\vec{q}) \quad \Longleftrightarrow \quad \vec{p} \times \vec{q} \gtreqqless 0.$$

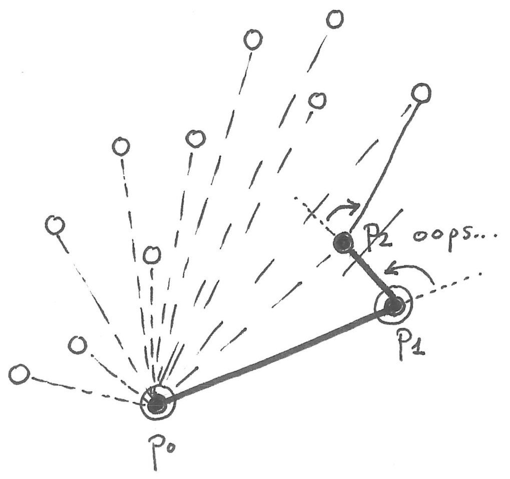
8.2.1 Graham's scan

Although the details are a bit tricky, the basic idea of the algorithm is quite straightforward: starting from the bottom point, with the origin on the bottom point, we go through all the others in order of their polar angle; each point is visited only once and is retained in a stack if it looks (so far) as being on the hull; it may be removed later by backtracking if we discover it wasn't after all.

In greater detail, the algorithm is as follows. We start from the point, call it p_0, with the smallest y coordinate (the leftmost such point in case of ties). We sort all other points based on their polar angle from p_0. We then examine these points in that order to build the boundary of the hull polygon. At any instant we have a "most recently discovered

[4]The polar angle of a point p_1 with respect to a reference point p_0 is the angle between the positive x axis and the point-vector $\vec{p_1} - \vec{p_0}$. In other words, place the butt end of an East-pointing arrow on p_0, then rotate that arrow around p_0 until it points towards p_1. The angle travelled is the polar angle of p_1 with respect to p_0.

vertex" (initially p_0) and, by drawing a segment from that to the point being examined, we obtain a new candidate side for the hull. If this new segment "turns left" with respect to the previous one, then good, because it means that the hull built so far is still convex. If instead it "turns right", then it's bad, because this introduces a concavity. In this case we must eliminate the "most recently discovered vertex" from the boundary (we now know it wasn't really a vertex of the hull after all—it must be an internal point if it's caught at the bottom of a concavity) and backtrack, reattaching the segment to the *previous* vertex in the hull. Then we must again check which way the end segment turns with respect to its predecessor, and we must keep backtracking if we discover that it still turns right. We continue backtracking until the end segment does not introduce a concavity. This is basically it: we proceed in this way until we get back to the starting point p_0.



The boundary case is that in which the new segment doesn't turn either left or right but instead "goes straight": this indicates a situation with more than two collinear "vertices" along the edge of the hull and of course we must eliminate the inner ones from the list of vertices since they're actually just ordinary points in the middle of an edge.

The backtracking procedure may appear complex at first but it is actually easy to implement with the right data structure—namely a stack. We just keep feeding newly found vertices into the stack and pop them out if they are later found to contribute to concavities in the boundary.

A formal proof of correctness of the algorithm is in the textbook.

As for complexity analysis, the initial step of sorting all the points by their polar angle can be performed in $O(n \lg n)$ using a good sorting algorithm and the cross product trick mentioned earlier[5]. It can be shown that no other section of the algorithm takes more than $O(n)$; therefore the overall complexity of Graham's scan is $O(n \lg n)$.

8.2.2 Jarvis's march

Here too we start from the bottommost leftmost point of the set and work our way anticlockwise, but this time we don't pre-sort all the points. Instead, we choose the point with the *minimum* polar angle with respect to the current point and make it a vertex of

[5]Actually, the use of arcsin instead of the cross product, though much less efficient, would not, of course, change the asymptotic complexity.

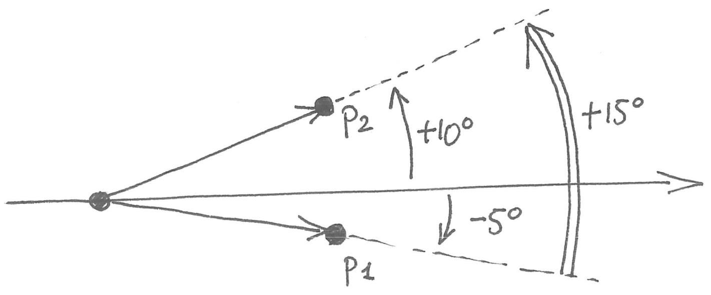
the hull: it must definitely be on the boundary if it's the first one we encounter while sweeping up from the "outside" region. We then make this new-found vertex our new reference point and we again compute the minimum polar angle of all the remaining points. By proceeding in this way we work our way up along the right side of the hull, eventually reaching the topmost point. We then repeat the procedure symmetrically on the other side, restarting from the bottommost point. Finally, if necessary, we deal with the boring but trivial boundary cases of "flat top" or "flat bottom".

Complexity analysis is easy: finding the minimum is an $O(n)$ operation given that each individual comparison is $O(1)$; this operation is repeated once for every vertex of the hull and therefore the overall complexity of Jarvis's march is $O(nh)$.

Exercise 74
Imagine a complex CAD situation in which the set contains about a million points. Which of the two algorithms we have seen would you use to compute the convex hull if you expected it to have about 1,000 vertices? And what is the rough boundary value (for the expected number of vertices of the hull) above which you would use one algorithm instead of the other? (Is this a trick question? If so, why?)

The motivation for performing distinct left and right passes is to be able to use the cross-product trick—but there are subtleties. Let's see.

Distinct left and right passes are needed if one considers the polar angles with respect to the shifted (positive or negative) x axis *and* wants to use the cross-product trick. This is because you can't get accurate results from the cross-product trick if the angle spanned by the two vectors being multiplied is greater than π.



Why? Assume p_1 is at 5 degrees below the x axis and p_2 is at 10 degrees above: then the shortest rotation from p_1 to p_2 is positive (anticlockwise), specifically $+15$ degrees, and indicates that p_1 has a smaller polar angle than p_2. This is indeed correct if we consider p_1 to have polar angle -5 degrees and p_2 to have $+10$ degrees, but it's incorrect if we say that we normalize all angles to within the 0 to 2π range, because in that case p_1 has polar angle of $+355$ degrees which is *not* smaller than p_2's! Normalizing angles to any other range spanning 2π (eg $-\pi$ to $+\pi$) might fix the problem for this particular counterexample but not in general: any two points on either side of the discontinuity (in this case at angle π; in the previous case at angle 0) and separated by less than π would exhibit the same problem, regardless of where the discontinuity is placed.

It is possible to use a single anticlockwise pass if one *computes* the polar angles with respect to the shifted positive x axis (using trigonometry), thus without using the cross-product trick (but that costs more, by a constant factor).

A smarter programmer will however use both a single anticlockwise pass *and* the cross-product trick by considering polar angles not with respect to the x axis but with respect to the last segment of the hull found so far.

Finally, have your say

Before the end of term, please visit the online feedback pages and leave an anonymous comment, explicitly mentioning both the good and the bad aspects of this course. If you don't, your own opinion of my work will necessarily be ignored. Instead, I'd be very grateful to have a chance to hear about it.

End of lecture course

Thank you, and best wishes for the rest of your Tripos.