UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Algorithms I

Part Ia CST, Part Ia NST, Part I PPS

Academic year 2009–2010

Easter term 2010

Lecturer for 2010: Dr Robert Harle

Course handout by Frank Stajano

`http://www.cl.cam.ac.uk/Teaching/0910/AlgorithI/`

Revised 2010 edition

Revision 5 of 2009-07-22 18:06:07 +0100 (Wed, 22 Jul 2009).

Contents

Introduction

Just as Mathematics is the "Queen and servant of science", the material covered by this course is fundamental to all aspects of computer science. Almost all the courses given in the Computer Science Tripos describe data structures and algorithms specialized towards the application being covered, whether it be networking, operating systems, databases, compilation, graphics, cryptography or whatever. These diverse fields often use data structures such as lists, hash tables and trees, and often use algorithms to perform basic tasks, such as table lookup, sorting or searching, that are common across applications. It is the purpose of this course to give a broad understanding of such commonly used data structures and their related algorithms. As a byproduct, you will learn to reason about the correctness and efficiency of programs.

The course is driven by the idea that, if you can analyse a problem well enough, you ought to be able to find the best way of solving it. That usually means the most efficient procedure or representation possible. Note that this is the best solution not just among all the ones that we can think of at present, but the best from among all solutions that there ever could be, including ones that might be extremely elaborate or difficult to program and still to be discovered. A way of solving a problem will (generally) only be accepted if we can demonstrate that it *always* works. This, of course, includes proving that the efficiency of the method is as claimed.

Most problems, even the simplest, have a remarkable number of candidate solutions. Often, slight changes in the assumptions may render different methods attractive. An effective computer scientist needs to have a good awareness of the range of possibilities that can arise, and a feel of when it will be worth checking textbooks to see if there is a good standard solution to apply.

Almost all the data structures and algorithms presented here are of real practical value and, in a great many cases, a programmer who failed to use them would risk inventing dramatically worse solutions to the problems addressed, or, of course in rare cases, finding a new and yet better solution—but be unaware of what has just been achieved!

Several techniques covered in this course are delicate, in the sense that sloppy explanations of them will miss important details, and sloppy coding will lead to code with subtle bugs. Beware!

A final feature of this course is that a fair number of the ideas it presents are really ingenious. Often, in retrospect, they are not difficult to understand or justify, but one might very reasonably be left with the strong feeling of "I wish I had thought of that" and an admiration for the cunning and insight of the originator.

Despite its fundamental nature, the subject is still a young one: the n-th editions of the recommended textbooks often contain extra algorithms that had not yet been invented when the corresponding $n-1$-th editions were published. New algorithms and improvements are still being found and there is a good chance that some of you will find fame (if not fortune) from inventiveness in this area. Good luck! But first you need to learn the basics.

Course content and textbooks

Even a cursory inspection of standard texts related to this course should be daunting. There is a great variety of textbooks on data structures and algorithms; their quality varies, and so does the subset of topics that they cover. There are some incredibly long books full of amazing detail, and there can be pages of mathematical analysis and justification for even simple-looking programs. This is in the nature of the subject. An enormous amount is known, and proper precise explanations of it can get quite technical. Fortunately, this lecture course does not have time to cover everything, so it will be built around a collection of sample problems or case studies. The majority of these will be ones that are covered well in most textbooks, and which are chosen for their practical importance as well as their intrinsic intellectual content.

There is no getting away from the fact that, unless you are content with a superficial understanding of the course material, you will need to study a textbook, for which this handout is not a substitute, and reimplement and debug the algorithms on a computer.

Contrary to the situation that existed until I took over, I have now restructured the syllabus of the two algorithms courses so that there exists at least one good textbook, namely Cormen et al, covering all the topics in it. This textbook—which, compared to its competitors, is even reasonably cheap—is therefore the recommended choice. Four textbooks[1] have been carefully considered as good candidates to support this course and their relative merits are discussed next.

All the books mentioned here are plausible candidates for the long-term reference shelf that any good computer scientist will keep: they are not the sort of text that one studies just for one course or exam and then forgets. Different books will appeal to different readers so feel free to pick any, so long as you do pick one, but be warned that only the first one covers all the topics in Algorithms I and II[2].

- Cormen, Leiserson, Rivest, Stein. *Introduction to Algorithms, Second edition.* MIT press, 2001. ISBN 0-262-53196-8.

  A heavyweight book at over 1100 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 and is also the cheapest of the ones I have considered and examined. I have therefore structured the syllabus around it, limiting the choice of topics to those covered by this text. Chapter references in the chapter headings of these notes are to this textbook. A third edition has been announced but was not yet available at the time of writing these notes.

- Sedgewick. *Algorithms, Third edition, in Java* (2 volumes). Addison-Wesley, 2004. ISBN 0-201-36120-5 and 0-201-36121-3.

  A well established and well written book, now in two volumes, perhaps slightly less encyclopædic than Cormen but often the clearest of the four texts surveyed here.

[1] Or textbook sets—two of them consist of multiple volumes.

[2] One reasonable way around it is to buy your favourite textbook from the list and study any missing topics on a library copy of Cormen. Another is to buy Cormen and consult a library copy of another if there are any points where Cormen doesn't satisfy you.

Had it not been for its cost, I would have probably made it the preferred choice for this course. This book comes in several variants which give sample implementations of the algorithms in various concrete programming languages. The version I evaluated was the one based on Java, since this is the main imperative language you are taught in the Tripos, but you would do equally well with any other.

- Kleinberg, Tardos. *Algorithm Design.* Addison-Wesley, 2006. ISBN 0-321-29535-8.

  An up-to-date and innovative book, well written and with a captivating problem-oriented approach. Also the lightest of the bunch, at "only" about 800 pages. It is however dedicated primarily to algorithms and does not have enough to say on data structures.

- Knuth. *The Art of Computer Programming, Third edition* (3 volumes so far; a boxed set is also available). Addison-Wesley, 1997. ISBN 0-201-89683-4, 0-201-89684-2 and 0-201-89685-0.

  When you look at the date of first publication of this series, and then observe that it is still in print[3], you will understand that it is a classic. Even though some aspects of the presentation are now outdated (e.g. many procedures are described by means of programs written in a specially invented imaginary assembly language called MIX), this is still the closest thing we have to a bible of the field. If you believe you are among the top 10% apprentice computer scientists in your course, consider studying Algorithms on the Knuths. Don't do it just for geek points, though, as this choice requires a considerable investment in several dimensions—brain cycles, time and wallet.

The URL `http://www.nist.gov/dads/` is a useful dictionary of algorithms and data structures.

A growing number of mostly accurate descriptions of algorithms and data structures can be found on Wikipedia. These write-ups are often based on the explanations in Cormen (occasionally misunderstood by the page authors). Once you master the material in this course, consider improving any you find lacking.

What is in these notes

These notes are not a substitute for buying and studying one of the recommended textbooks. For this particular course there exist very good standard texts and they are sufficiently cheap that there is no point in trying to duplicate them. Instead, these notes convey just the crucial ideas about the topics studied in the course. They may be useful for quickly reading up on a topic before attending the corresponding lecture and they can help when organizing revision.

You are encouraged to come to each lecture with this handout *and* a paper notebook, and use the latter to take notes, sketch diagrams and solve exercises as you prepare for upcoming supervisions. The margin isn't large enough for that.

[3]It is actually still being written and rewritten by its legendary author, who has relinquished most other commitments, including email, in the hope that he will complete it before passing away.

Even when these notes appear to document a complete algorithm, they may gloss over important details for brevity. Those relying only on these notes, as opposed to lectures and textbooks, may end up with the impression that some topics are simpler than they actually are.

These notes contain short exercises, highlighted by boxes, that students who end up getting first-class marks usually solve as they go along, to prove to themselves that they are not just reading on autopilot. Such exercises 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, resort to writing and debugging your own implementations, solving problems in your textbooks, solving problems set by your supervisor and, ultimately, to solving past exam questions. There is a copious supply of these on the course's web page but be sure to choose (or ask your supervisor to help you choose) ones that are covered by this year's syllabus, as the structure of the course has changed several times over the years.

Acknowledgements

A student handout for *Data Structures and Algorithms*, the predecessor of this course, was originally written by Arthur Norman and then enhanced by Roger Needham and Martin Richards: I hereby express my gratitude to my illustrious predecessors. I thoroughly revised and restructured those notes when I took over the course in the academic year 2005–2006 and again in 2008–2009 when it became *Algorithms I*. I also did minor revisions in all the other years I taught it.

I am grateful to Kay Henning Brodersen, Sam Staton, Simon Spacey, Rasmus King, Chlo Brown, Rob Harle, Larry Paulson and particularly Alastair Beresford, as well as a number of others who preferred to remain anonymous, for sending me corrections and pointing out problems in previous edition of these notes. If you find any more and send them to me, I'll credit you in next year's edition (unless you prefer anonymity). The responsibility for any remaining mistakes remains of course mine.

Alastair Beresford and Andy Rice also deserve special thanks for their invaluable help in transforming my earlier "microchallenges" into formally assessed "ticks".

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 solutions.

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? What makes 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 Top 100

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 the store had *billions* of different items for sale (not just books) instead of merely millions of them? Of course you could re-sort the whole catalogue each time and take the top 100 items, but how can you do better?

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 whatever. 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 inefficient? 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.

- Is the algorithm correct? How can we prove that it is?

[1]There may be several.

- 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

Contents

Review of insertion sort, merge sort and quicksort. Understanding their memory usage with arrays. Heapsort. Other sorting methods. Finding the minimum and maximum.
Expected coverage: 4–5 lectures.
Refer to the following chapters in Cormen *et al.*: 2, 6, 7, 8, 9.

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, 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.

When we pick up card `a[`k`]`, since the first $k-1$ 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 $k-1$ and, if smaller than it, a swap moves it down. If it did move down, the new element is now compared against the one in position $k-2$ and again swapped if necessary, then $k-3$ and so on until it gets to its place.

We can write down this algorithm in pseudocode as follows:

```
def insertSort(a):
    '''BEHAVIOUR: Run the insertsort 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 to len(a)-2:
        assert(the first k positions are already sorted)

        # Pick up item k+1 (call it a[j]) and let it sink to its correct place
        j = k+1
        while j > 0 and a[j-1] > a[j]:
            swap(a[j-1], a[j])
            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

Assume that each `swap(x, y)` means three assignments (namely `tmp = x; x = y; y = tmp`). Improve the insertsort algorithm on page 12 to reduce the number of assignments performed in the inner loop. (Answer in your textbook.)

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. This is a good way to provide a specification.

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[1]. 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[2]. 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 or end of each significant loop. In the algorithm above we have an invariant on line 10, at the beginning of the main loop: the k^{th} time we enter the loop, it says, the previous passes of the loop will have sorted the leftmost k cells of the array. How? We

[1]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.

[2]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.

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 ($k = 0$; 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

Provide a useful invariant for the inner loop, in the form of an assertion to be inserted between lines 14 and 15.

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, as you know, 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 $100000f(n)$ and $0.000001f(n)$ 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, modulo a constant factor and for sufficiently large values of n, $f(n)$ will never exceed[3]

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.

Some authors also use $\Omega()$ as the dual of $O()$ to provide a lower bound.

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.

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.

Here are a few examples that may help explain:

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

Exercise

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.

[3]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 positive n).

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.

Exercise

For each of the following functions $f(x)$, find an asymptotic formula for $f(2x)/f(x)$. This tells you how f grows when its input doubles. For $f(x) = x^2$, for example, the result is 4, telling you that the value of f quadruples when its input doubles.

$$\begin{aligned} f(x) &= 1 \\ f(x) &= x \\ f(x) &= \lg x \\ f(x) &= x \lg x \\ f(x) &= x^2 \\ f(x) &= 2^x \end{aligned}$$

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 dangerously close to the amount installed by default in the kind of 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 the computer (or 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 all 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[4], 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.

2.3.4 Models of arithmetic

The normal model for computer arithmetic used here will be that each arithmetic operation 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[5]. 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

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

[5]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.

taken for every case of an algorithm must be accounted for—and this, usually, makes the mathematics a lot harder.

Amortized analysis[6] 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.

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 simply 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 12, we see that the outer loop of line 9 is executed exactly n times (regardless of the actual values in the input), 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 k. So, in the worst case, during the k^{th} invocation of the outer loop, the inner loop will be performed k 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 $O(n^2)$.

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

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 the boxed exercise on page 13.

Exercise

What is the asymptotic complexity of that variant of insertsort?

End of Lecture (perhaps)

Expect lecture 1 to finish approximately here—but no promises.

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[7]. The second (exchange) swaps the contents of two nominated array locations.

In extreme cases either comparisons or exchanges[8] 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.

[7]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.

[8]Often, if exchanges seem 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.

Exercise

This proves that $\Theta(n)$ exchanges are always *sufficient*. But why isn't the above argument good enough to prove that they are also *needed*? Do not proceed until you have an answer.

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 $O(n \lg n)$ 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 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 $\lg(n/2) = \lg n - 1$, 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 $n - 1$ 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 to len(a)-1:
        assert(the 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 to len(a)-1:
            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 sort 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

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 very cheap, but comparisons are very expensive. 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 some number of exchange operations (at most k) will drop the item in place. 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 $\lg((n-1)!) + n = O(\lg(n!)) = O(n \lg n)$. 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 the 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 0 to len(a)-1:
        assert(the positions before a[k] are already sorted)

        # Use binary partitioning of a[0..k-1] 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 between a[i] and a[i+1])

        # Put a[k] where it belongs, shifting everyone else in the
        # sorted region to the right by one cell
        tmp = a[k]
        for j from k-1 to i:
            a[j+1] = a[j]
        a[i] = tmp
```

Exercise

Code up the details of the binary partitioning portion of this algorithm.

2.8 Bubble sort

A further simple sort 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) 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.

```
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 to len(a)-2:
            if a[k] > a[k+1]:
                swap(a[k], a[k+1])
                didSomeSwapsInThisPass = true
    until didSomeSwapsInThisPass == false
```

Exercise

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

2.9 Mergesort

Given a pair of arrays each of length $n/2$ that have already been sorted, merging the data into a single sorted list is easy to do in around n steps: just keep taking the lowest element from the sub-array that has it. You have seen a full sorting algorithm based on this idea: split the input array recursively into chunks small enough that there is no difficulty in sorting them (or even no *need* to sort them) and then merge them back two by two into progressively larger chunks.

```
def mergeSort(a):
    '''*** DISCLAIMER: this is purposefully NOT a perfect bug-free example,
    but it is a useful starting point for our discussion. ***

    BEHAVIOUR: Run the merge sort algorithm on the integer
    array a, returning a sorted version of the array as the result.

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

    POSTCONDITION: a new array is returned that contains the same
    integer values originally in a, but sorted in ascending order.'''

    if len(a) < 2:
        assert(a is so small that it is already sorted)
        return a

    # Split array a into two smaller arrays a1 and a2
    # and sort these recursively
    h = len(a)/2
    a1 = mergeSort(a[0..h])
    a2 = mergeSort(a[h+1..end])

    # Form a new array a3 by merging a1 and a2
    a3 = new empty array of size len(a)
    i1, i2, i3 = 0, 0, 0 # these index into arrays a1, a2, a3 respectively
    while i1 < len(a1) or i2 < len(a2):
        a3[i3++] = smallest(a1, i1, a2, i2) # this call updates i1 or i2 too

    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 22–26) is most naturally done by leaving the two input arrays alone and forming the result into a temporary buffer (line 23) 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 19–20) 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.
- Merging the two sub-arrays is conceptually easy (just consume the "emerging" item from each deck of cards) but coding it up naively will fail on boundary cases, as the following exercise highlights.

Exercise

Can you spot any problems with the suggestion of replacing line 26 with the more obvious `a3[i3++] = min(a1[i], 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

In line 14 we return the same array we received from the caller, while in line 28 we return a new one we created in line 23. This asymmetry is suspicious. What kind of memory management problems can we expect from it? (Hint: cleaning up.)

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 19–20) on inputs of size $n/2$ and the second term is the overall cost of the merging phase (lines 22–26), 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[9] with the substitution $n = 2^m$.

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

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 guarantees the optimal cost of $n \lg n$, is relatively simple and has low time overheads. Its only disadvantage is to require 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.

Exercise

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

Note that a practical fast implementation, unlike one appealing to mathematicians and æsthetes, would not recurse down to single-element arrays; it would instead probably resort to Insertion sort once the array chunks got down to a few elements.

[9] 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 your textbook.

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 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

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

Exercise

Write pseudocode for the bottom-up mergesort.

End of Lecture (perhaps)

Expect lecture 2 to finish approximately here—but no promises.

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 from indices `iBegin` (included) to `iEnd` (excluded)[10]. We shall indicate this contiguous range as `[iBegin..iEnd)`. We use two auxiliary indices `iLeft` and `iRight`. We arbitrarily pick the last element in the range, `A[iEnd-1]`, as the pivot. 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`
- `[iBegin..iLeft)` only has elements $\leq$ `Pivot`
- `[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]` $\leq$ `Pivot` (thanks to the symmetrical argument). So we swap these two elements pointed to by `iLeft` and `iRight`. 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. 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 `[iBegin..iLeft)` and `[iRight+1..iEnd)`.

[10] A common source of bugs in computing is the off-by-one error. 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 things from zero rather than from one—then most "offset + displacement" calculations will just work. Another is to adopt the convention used in the ascii-art diagram in this section 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.

```
                                            Pivot
                                               .
   iBegin      iLeft  -->    <-- iRight iEnd-1 .   iEnd
     |           |                   |       | .    /
     |           |                   |       | .  /
     |           |                   |       | . |
     v           v                   v       v . v
-+---+---+---+---+---+---+---+---+---+---+---+-.-+---+---+
                                               .         |
 |   |   |   |   |   |   |   |   |   |   |   | P |   |   |
                                                         |
-+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

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 the total 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

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[11] 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

Justify why this 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 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[12].

Exercise

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

[11]Cfr. section 2.11.

[12]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 Cormen, 9.3.

Exercise

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

The simpler scheme selects a pivot and partitions as for Quicksort. 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. 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

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

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.

End of Lecture (perhaps)

Expect lecture 3 to finish approximately here—but no promises.

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[13], 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[14] $2k+1$ 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.

Exercise

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

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.

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 item from the heap (which, as we saw, was the largest value present) and swaps it to 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.

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

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

```
 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 len(a)/2 downto 0: # only look at nodes with children
11         heapify(a, k)
12
13     # Second, repeatedly extract the max, building the sorted array R-to-L
14     for k from len(a)-1 downto 0:
15         assert(a[k+1..end] is already sorted)
16         swap(a[0], a[k]) # extract the max, a[0], and put it where it should be
17         heapify(a[0..k-1], 0)
18
19
20 def heapify(a, i):
21     '''BEHAVIOUR: Within array a, consider the subtree rooted at a[i] and
22     make it into a max-heap if it isn't one already.
23
24     PRECONDITION: the children of a[i], if any, are already roots of
25     max-heaps.
26
27     POSTCONDITION: a[i] is root of a max-heap.'''
28
29
30     if a[i] satisfies the max-heap property:
31         return
32     else:
33        let j point to the largest of the two children of a[i]
34        assert(a[i] < a[j]) # undesirable because a[i] ought to be max
35        swap(a[i], a[j])
36        heapify(a, j)
```

The auxiliary function `heapify` (lines 20–36) 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 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 $O(\lg n)$.

The first phase of the main `heapSort` function starts from the bottom of the tree 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. 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, the array is split into two distinct parts: `a[0..k]` is a heap, while `a[k+1..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. At each step we extract the maximum element from the root of the heap, replace it with the last element of the heap and and put the maximum in the empty space left by the last element, which coincidentally is exactly its ultimate resting place! Then all that’s left is to retransform `a[0..k-1]` into a heap, which we can do by calling `heapify` since the two subtrees must still be heaps given that all that changed was the root and we started from a proper heap before that. For this second phase, too, it is trivial to establish an $O(n \lg n)$ bound.

Now, what’s this 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[15], 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

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:

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

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

$$
\begin{aligned}
&= k2^h \sum_{l=0..h} l2^{-l} \\
&= k2^h \sum_{l=0..h} l \left(\frac{1}{2}\right)^l
\end{aligned}
$$

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.

End of Lecture (perhaps)

Expect lecture 4 to finish approximately here—but no promises.

2.13 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 $O(n \lg n)$. Two particular cases are common enough to be of at least occasional importance.

The first is when the values to be sorted are integers that live in a known range, and where the range is fixed regardless of the number of values to be processed. Assuming the number of input items grows beyond the cardinality of the range, there will necessarily be duplicates in the list. 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 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.

Exercise

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

Another case is when 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.

2.14 Stability of sorting methods

Often data to be sorted consists of records containing a key value that the ordering is based upon plus 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 the records to be sorted are extended to hold an extra field that stores their original position, and if the ordering predicate used while sorting is extended 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 overheads a little.

Exercise

For each of the sorting algorithms seen so far, establish whether it is stable or not.

Chapter 3

Algorithm design

Contents

Ideas for algorithm design: dynamic programming, divide and conquer, greedy algorithms and other useful paradigms.
Expected coverage: 2 lectures.
Refer to the following chapters in Cormen *et al.*: 2, 15.

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 clever techniques, 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 are going to 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 has nothing to do with the usual semantics of writing instructions for a computer: it originally meant something like "finding a plan of action".

Dynamic programming is related to the strategy of "divide and conquer" (section 3.2.3, 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

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}? 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 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 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.

Exercise

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.

We naturally would like to choose an order which minimizes the number of scalar multiplications. 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. This set of dimensions can be specified by the vector $(p_0, p_1, \ldots, p_6) = (30, 35, 15, 5, 10, 20, 25)$ and 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 simple implementation of this function will compute $m(i,j)$ in time exponential in the number of matrices to be multiplied[1], 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").

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

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*

[1]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 very expensive computationally.

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.

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 required 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.

End of Lecture (perhaps)

Expect lecture 5 to finish approximately here—but no promises.

3.2 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. 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.

3.2.1 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[2], which uses as a sub-algorithm a particularly efficient way of comparing the relative positions of two vectors, is an illustration of this.

3.2.2 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 to the insight that it is not necessary to invent a scheme that solves a whole problem all in one step—just some process 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.

3.2.3 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.

3.2.4 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.

[2] To be covered in the Algorithms II course.

3.2.5 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.

3.2.6 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 on the problem and then writing up the edited results, with credit to the candidates, as an academic paper[3]!

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.

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

3.2.8 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.

Chapter 4

Data structures

Contents

Abstract data types. Pointers, stacks, queues, lists, trees. Hash tables. Binary search trees. Red-black trees. B-trees. Priority queues and heaps.
Expected coverage: 6 lectures.
Refer to the following chapters in Cormen *et al.*: 10, 11, 12, 13, 18, 19, 20, 21.

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 (which we have already used informally from the start), where the arrays generally have 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 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.

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. It will generally be assumed that integer arithmetic never overflows, that the 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[1].

Exercise

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[2], 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. I 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?

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

[2]You might rightfully observe that it would perhaps 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

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.

Exercise

Are there any other 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 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[3]. 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 first element (as usual) plus a boolean flag (which takes one bit[4]) 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

Draw a picture of the above representation.

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

[3]Cfr. section 4.2.

[4]And thereby slightly reduces the space available for the actual element.

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. Scanning forwards down a list is easy. Lists provide one natural implementation of stacks, 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 of course you independently remember where the original head of your list was). To make it possible to traverse a list in both directions one could define a new type called DLL (for Doubly Linked List) in which each wagon had both a `next` and a `previous` pointer. The following equation

```
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. 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

If a graph has n vertices then it can be represented by an “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 (especially when there are very many edges, and if the bit-array is stored packed to make full use of machine words) can be more compact. We shall have much more to say about graphs in the Algorithms II course.

End of Lecture (perhaps)

Expect lecture 6 to finish approximately here—but no promises.

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[5]. The specification of an ADT is a list of the operations that may be performed on it, together with the identities that they satisfy. This specification does *not* show how to implement anything in terms of any simpler 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*[6] (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.

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

[6]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.

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".

```
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() is 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 semi-meaningful value would get returned, 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: an algorithms course does 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 a counter. The `push` operation writes a value into the array and increments the counter, 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) after invoking either of them the original stack is no longer available.

Stacks pop up in the most diverse places. 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). Very roughly, 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

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 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 optimizations 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[7]. Such issues will be ducked for now. Again, type-checking issues about the types of items stored in lists will be skipped over.

```
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[8] 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()` (one would re-name these operations on a queue from those on a stack to reflect this). Even if finding a neat way of expressing this in a mathematical description of the Queue ADT may be a challenge, the idea is not. 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.

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

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

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

```
  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[9] 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 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[10]: 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`.

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

[9] One `put` and one `get`, for obvious reasons.
[10] As opposed to a generic relation.

```
  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) == value

  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. Particular 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 range $(b - a)$ that they lie in, this vector representation becomes inefficient in space, even though its time performance remains optimal.

Exercise

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, it 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

Should it be added at the start or the end of the list? Or elsewhere?

If 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 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

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[11]. 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
```

[11]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.

End of Lecture (perhaps)

Expect lecture 7 to finish approximately here—but no promises.

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[12]) 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.

[12]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

(Clever challenge, straight from Cormen—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[13] 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-left, then the node has no successor: its key is the highest in the tree.

Exercise

Why does this up-and-right business find the successor? Can you sketch a proof?

Exercise

(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. One will be to exchange the contents of the non-leaf cell with either the predecessor or the successor (whichever one is a descendant rather than an ancestor—from the result of the previous exercise you can prove that at least one of the two is). Then the item for deletion is in a leaf position and can be disposed of without further trouble; meanwhile, the newly moved up object satisfies the order requirements that keep the tree structure valid.

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

Exercise

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

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 a worst case is when a tree is made by inserting items in ascending order: then the tree degenerates into a 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.

End of Lecture (perhaps)

Expect lecture 8 to finish approximately here—but no promises.

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[14].
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.

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

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[15]. 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

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, 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[16].

4.4.2 2-3-4 trees

Binary trees had one key and two 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 which have 2 keys and 3 pointers, and 4-nodes with 3 keys and 4 pointers. As with regular binary trees, the pointers are all to sub-trees which only contain key values limited by the keys in the parent node.

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[17]—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. If the insertion was going to be into a 4-node, something has to be done to make space for

[15]Minus one if we count the empty leaf as one of the nodes, but see footnote 14.

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

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

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 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 all items in a tree with n items in will be at worst $\lg n$ down from the root.

Deletions require substantial 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 stand 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

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, internal links and external links 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.

End of Lecture (perhaps)

Expect lecture 9 to finish approximately here—but no promises.

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 key values[18]. 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

[18]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.

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[19]. 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[20], 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).
2. For each key in a node, the node also holds the associated payload (or, more likely, a pointer to it, unless the payload has a small fixed size comparable to that of the pointer).
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[21] 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

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

[20]See footnote 19 again.

[21]Structurally, you can view 2-3-4 trees as a subcase of B-trees: just take $t = 2$.

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.

Exercise

(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, 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

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, and this different place will typically be the centre of the large merged node.

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 algorithm to delete a key k from the B-tree:

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

```
if either of B's siblings can afford to lose any keys:
    redistribute keys between B and that sibling
else:
    (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)
```

End of Lecture (perhaps)

Expect lecture 10 to finish approximately here—but no promises.

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 very good coherent structure; hashing, instead, places its bet the other way, on chaos.

A hash function $h(k)$ maps a key 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[22].

Exercise

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 simple hash function

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

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 already 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.

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

Exercise

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

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

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 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 $N-1$).

```
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 $h(k) + j \bmod N$. 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.

End of Lecture (perhaps)

Expect lecture 11 to finish approximately here—but no promises.

4.7 Priority queues and various types of 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), 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.

```
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(Key old, Key new);
  // PRECONDITION: new < old
  // PRECONDITION: an item with key <old> is already in the queue.
  // BEHAVIOUR: change, from <old> to <new>, the key of the item identified
  // by key <old>, thereby increasing its priority.
}

class Item {
  // A total order is defined on the keys.
  Key k;
  Payload p;
}
```

You could implement a priority queue simply with a sorted array, but you'd have to keep the array sorted at every operation, for example with one pass of bubble-sort.

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

Exercise

If we are using bubblesort, why did I indicate the costs as linear rather than quadratic?

We can do better than this.

4.7.1 Binary heap

A good representation for a priority queue is a regular binary heap (cfr. Heapsort in section 2.12), where the minimum item is instantly available and the other operations can be performed in logarithmic time.

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 is filled left-to-right) and each node has a key less than those of its two children.

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 properties (max number of steps: height of the tree, so $\lg n$). To extract the root, do so, then replace it with the element at the end of the heap, letting the latter sink down (under the heavier of its two children) until it no longer violates the heap properties (again the max number of steps is 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).

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

All these costs are quite reasonable.

4.7.2 Binomial heap

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, all you can do is extract each of the elements from the smaller queue and insert them into the other, at a total cost bounded by $O(n \lg n)$. The binomial heap is an elaborate data structure that provides much more efficient (namely $O(\lg n)$) support for merging two instances.

A **binomial tree** (not heap) of order 0 is a single node.

A binomial tree of order k is a tree obtained by combining two binomial trees of order $k-1$, with the root of one becoming the leftmost child of the root of the other[23]. By induction, it contains 2^k nodes.

A **binomial heap** is a set of binomial trees (at most one for each tree order) with each node of each tree having a key greater than that of its parent. It can easily be seen that, if the heap has n nodes, it has $O(\lg n)$ binomial trees and the largest of those trees has $O(n)$ nodes.

`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 $O(\lg n)$.

`extractMin()` To extract the minimum, after finding it as you do in `first()`, remove it from its tree—this yields a forest of binomial trees of smaller order. Take them in increasing order (just reverse the list of subroots, which has $O(\lg n)$ elements) and you have another binomial heap. Merge it with what remains of the original one. Since the merge operation itself, as we shall see next, costs $O(\lg n)$, this is also the total cost of extracting the minimum.

`merge()` To merge two binomial heaps, start by examining their trees in increasing order. If at most one of the two heaps has a tree of a given order, no problem—so will it be for the resulting heap. If both heaps have a tree of order j, append the tree with the larger root to the other and proceed as you do when you encounter a carry in binary addition. The amount of work for each order is bounded by a constant, so in total 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, at cost no greater than $O(\lg n)$.

	sorted array	binary heap	binomial heap
creation of empty queue	$O(1)$	$O(1)$	$O(1)$
`first()`	$O(1)$	$O(1)$	$O(\lg n)$
`insert()`	$O(n)$	$O(\lg n)$	$O(\lg n)$
`extractMin()`	$O(n)$	$O(\lg n)$	$O(\lg n)$
`decreaseKey()`	$O(n)$	$O(\lg n)$	$O(\lg n)$
`merge()`	$O(n)$	$O(n \lg n)$	$O(\lg n)$

[23]Note that a binomial tree is not a binary tree.

4.7.3 Fibonacci heap

The Fibonacci heap is a complex variation on the binomial heap that offers an amazing $O(1)$ amortized cost for all operations that don't remove nodes from the queue, while still retaining $O(\lg n)$ cost for `extractMin()`. 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. Since achieving the best possible computing times may occasionally rely on the performance of data structures as elaborate as the Fibonacci heap, it is important at least to know that they exist and where full details are documented. Those of you who find all the material in this course both fun and easy should look such data structures up in their textbook and try to produce a good implementation. They will see them in use in Algorithms II.

Final remarks

The algorithms and data structures that are discussed here and in Algorithms II occur quite frequently in real applications, and they can often arise as computational hot-spots where quite small amounts of code limit the speed of a whole large program. Many applications call for slight variations of adjustments of standard algorithms, and in other cases the selection of a method to be used should depend on insight into patterns of use that will arise in the program that is being designed.

A recent issue in this field involves the possibility of some algorithms being covered (certainly in the USA, and less clearly in Europe) by patents. In some cases the correct response to such a situation is to take a license from the patent owners; in other cases, even a *claim* to a patent may cause you to want to invent your very own new and different algorithm to solve your problem in a royalty-free way.

Thank you for attending my lectures and best wishes for the rest of your Tripos!