Algorithms

Keir Fraser

keir.fraser@cl.cam.ac.uk

CST IA

Easter 2006

12 Lectures, M/W/F

http://www.cl.cam.ac.uk/Teaching/2005/Algorithms/

Computer Laboratory
University of Cambridge

Contents

Preliminaries

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 and Diploma describe structures and algorithms specialised towards the application being covered, whether it be databases, compilation, graphics, operating systems, or whatever. These diverse fields often use similar data structures, such as lists, hash tables, trees and graphs, and often apply similar algorithms to perform basic tasks such as table lookup, sorting, searching, or operations on graphs. 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.

It might seem that the study of simple problems and the presentation of textbook-style code fragments to solve them would make this first simple course an ultimately boring one. But this should not be the case for various reasons. 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 text-books to see if there is a good standard solution to apply.

Almost all the data structures and algorithms that go with them 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.

Course content and textbooks

Even a cursory inspection of standard texts related to this course should be daunting. 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 all the 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 and a consequently poor exam result, you will need to buy (and study!) a textbook, for which this handout is *not* a substitute. The following two textbooks are good candidates to support this course. Both are also plausible candidates for the long-term reference shelf that any computer scientist will keep: they are not the sort of text that one studies just for one course or exam then forgets.

- Cormen, Leiserson, Rivest, Stein. *Introduction to Algorithms, Second edition.* MIT Press, 2001.
  A heavyweight book at 1100 pages long, and naturally covers a little more material at slightly greater depth than the other texts listed here. It includes careful mathematical treatment of the algorithms that it discusses, and would be a natural candidate for a reference shelf. Despite its bulk and precision this book is written in a fairly friendly and non-daunting style.

- Sedgewick. *Algorithms, Third edition* (first volume). Addison-Wesley, 2004.
  A respectable and generally less daunting book which comes in variants which give sample implementations of the algorithms that it discusses in various concrete programming languages. I recommend the Java version only because that is the primary language you are taught during the Tripos, but it really does not matter that much which one you choose.

What is in these notes

The first thing to make clear is that these notes are not in any way a substitute for having your own copy of one of the recommended textbooks. For this particular course the standard texts are sufficiently good and sufficiently cheap that there is no point in trying to duplicate them. Instead these notes will provide skeleton coverage of the material used in the course, and of some that although not used this year may be included next. They may be useful places to jot references to the page numbers in the main texts where full explanations of various points are given, and can help when organising revision.

These notes are not a substitute for attending lectures or buying and reading the textbooks. In places the notes contain little more than topic headings, while even when they appear to document a complete algorithm they may gloss over important details.

The lectures will not slavishly follow these notes, and for examination purposes it can be supposed that questions will be set on what was either lectured directly or was very obviously associated with the material as lectured, so that all diligent students will have found it while doing the reading of textbooks properly associated with taking a seriously technical course like this one.

This course assumes some knowledge (but not very much detailed knowledge) of programming in traditional "procedural" language. Some familiarity with the C language would be useful, but being able to program in Java is sufficient. Examples given may be written in a notation reminiscent of these languages, but little concern will be given (in lectures or in marking examination scripts) to syntactic details. Fragments of program will be explained in words rather than code when this seems best for clarity.

For the purpose of practising appropriate examination questions, many questions from previous years' 1B Data Structures and Algorithms course are relevant. Going back as far as 1997, the following questions could be attempted after attending the appropriate lectures: 05/6/1 (algorithm construction), 05/4/3 (hashing), 04/4/3 (splay trees), 03/4/3 (hashing), 03/3/3 (heapsort), 02/4/4 (shellsort), 01/3/5 (quicksort, order statistsics), 00/3/5 (order statistics), 99/6/1 (splay trees, hashing), 99/5/1 (heapsort), 98/3/6 (heaps), and 97/3/6 (algorithm construction).

Acknowledgements

These notes are closely based on those originally written by Arthur Norman in 1994, and combined with some details from Roger Needham's notes of 1997. They also contain various restructuring cleanups made by Frank Stajano in 2005.

Chapter 1

Fundamentals

An *algorithm* is a systematic process for solving some problem. This course will take the word 'systematic' fairly seriously. It will 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 reducing the amount of memory that they use) so a justification that the intended performance will be attained is very important.

1.1 Algorithm analysis

1.1.1 Costs and scaling

How should we measure costs? The problems considered in this course are all ones where it is reasonable to have a single program that will accept input data and eventually deliver a result. We look at the way costs vary with the data. For a collection of problem instances we can assess solutions in two ways — either by looking at the cost in the worst case or by taking an average cost over all the separate instances that we have. Which is more useful? Which is easier to analyse?

In most cases there are "large" and "small" problems, and somewhat naturally the large ones are costlier to solve. The next thing to look at is how the cost grows with problem size. In this lecture course, size will be measured informally by whatever parameter seems natural in the class of problems being looked at. For instance when we have a collection of n numbers to put into ascending order the number n will be taken as the problem size. For any combination of algorithm (A) and computer system (C) to run the algorithm on, the cost[1] of solving a particular instance (P) of a problem might be some function $f(A, C, P)$. This

[1] Time in seconds, perhaps

will *not* tend to be a nice tidy function! If one then takes the greatest value of the function f as P ranges over all problems of size n one gets what might be a slightly simpler function $f'(A, C, n)$ which now depends just on the size of the problem and not on which particular instance is being looked at.

1.1.2 Asymptotic notation

The above is still much too ugly to work with, and the dependence on the details of the computer used adds quite unreasonable complication. The way out of this is first to adopt a generic idea of what a computer is, and measure costs in abstract "program steps" rather than in real seconds, and then to agree to ignore constant factors in the cost-estimation formula. As a further simplification we agree that all small problems can be solved pretty rapidly anyway, and so the main thing that matters will be how costs grow as problems do.

To cope with this we need a notation that indicates that a load of fine detail is being abandoned. The one used is called Θ notation (there is a closely related one called "O notation" (pronounced as big-Oh)). If we say that a function $g(n)$ is $\Theta(h(n))$ what we mean is that there is a constant k such that for all sufficiently large n we have $g(n)$ and $h(n)$ within a factor of k of each other.

If we did some very elaborate analysis and found that the exact cost of solving some problem was a messy formula such as $17n^3 - 11n^2 \log(n) + 105n \log^2(n) + 77631$ then we could just write the cost as $\Theta(n^3)$ which is obviously much easier to cope with, and in most cases is as useful as the full formula.

Sometimes is not necessary to specify a lower bound on the cost of some procedure — just an upper bound will do. In that case the notation $g(n) = O(h(n))$ would be used, and that we can find a constant k such that for sufficiently large n we have $g(n) < kh(n)$.

Note that the use of an $=$ sign with these notations is really a little odd, but the notation has now become standard.

The use of Θ and related notations seem to confuse many students, so here are some examples:

1. $x^2 = O(x^3)$
2. x^3 is **not** $O(x^2)$
3. 1.001^n is **not** $O(n^{1000})$ — but you probably never thought it was anyway.
4. x^5 can probably be computed in time $O(1)$ (if we suppose that our computer can multiply two numbers in unit time).
5. $n!$ can be computed in $O(n)$ arithmetic operations, but has value bigger than $O(n^k)$ for any fixed k.
6. A number n can be represented by a string of $\Theta(\log n)$ digits.

Please note the distinction between the value of a function and the amount of time it may take to compute it.

1.1.3 Growth rates

Suppose a computer is capable of performing 1000000 "operations" per second. Make yourself a table showing how long a calculation would take on such a machine if a problem of size n takes each of $\log(n), n, n\log(n), n^2, n^3$ and 2^n operations. Consider $n = 1, 10, 100, 1000$ and 1000000. You will see that the there can be real practical implications associated with different growth rates. For sufficiently large n any constant multipliers in the cost formula get swamped: for instance if $n > 25$ then $2^n > 1000000n$ — the apparently large scale factor of 1000000 has proved less important that the difference between linear and exponential growth. For this reason it feels reasonable to suppose that an algorithm with cost $O(n^2)$ will out-perform one with cost $O(n^3)$ even if the O notation conceals a quite large constant factor weighing against the $O(n^2)$ procedure[2].

1.1.4 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, one can set up a single array of integers that is as large as you ever need.

There are of course practical ways in which this idealisation 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 is at present very common) 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 a few gigabytes of memory: a large quantity for personal machines maybe but a problem for large scientific calculations on supercomputers now, and one for workstations quite soon. The resolution is that the width of integer subscript/address calculation has to increase as the size of a computer or problem does, and 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 working.

[2] Of course there are some practical cases where we never have problems large enough to make this argument valid, but it is remarkable how often this slightly sloppy argument works well in the real world.

It is normal in the analysis of algorithms to ignore these problems and assume that element of an array `a[i]` 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.

On chip cache stores in modern processors are beginning to invalidate the last paragraph. In the good old days a memory reference use to take unit time (4μsecs, say), but now machines are much faster and use super fast cache stores that can typically serve up a memory value in one or two CPU clock ticks, but when a cache miss occurs it often takes between 10 and 30 ticks, and within 5 years we can expect the penalty to be more like 100 ticks. Locality of reference is thus becoming an issue, and one which most text books ignore.

1.1.5 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 — the specification of an algorithm (or an Abstract Data Type) there may be some integers, and in the idealised 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. 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.

1.1.6 Worst, average and amortised costs

Usually the simplest way of analysing an algorithms 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 (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 that could invalidate analysis based on uniform probabilities. For worst case analysis it is only necessary to study one limiting case; for average analysis the time taken for every case of an algorithm must be accounted for and this makes the mathematics a lot harder (usually).

Amortised analysis 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 re-organisation of the data. Amortised analysis treats the cost of this re-organisation as the joint responsibility of all the operations previously performed on the data structure and provide a firm basis for determining if it was worth-while. Again it is typically more technically demanding than just single-operation worst-case analysis.

1.1.7 Estimation of cost via recurrence formulae

Consider particularly the case of divide and conquer (Section 1.2.3). Suppose that for a problem of size n the division and combining steps involve $O(n)$ basic operations[3]. Suppose furthermore that the division stage splits an original problem of size n into two sub-problems each of size $n/2$. Then the cost for the whole solution process is bounded by $f(n)$, a function that satisfies

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

where k is a constant ($k > 0$) that relates to the real cost of the division and combination steps. This recurrence can be solved to get $f(n) = \Theta(n \log(n))$.

More elaborate divide and conquer algorithms may lead to either more than two sub-problems to solve, or sub-problems that are not just half the size of the original, or division/combination costs that are not linear in n. There are only a few cases important enough to include in these notes. The first is the recurrence that corresponds to algorithms that at linear cost (constant of proportionality k) can reduce a problem to one smaller by a fixed factor α:

$$g(n) = g(\alpha n) + kn$$

where $\alpha < 1$ and again $k > 0$. This has the solution $g(n) = \Theta(n)$. If α is close to 1 the constant of proportionality hidden by the Θ notation may be quite high and the method might be correspondingly less attractive than might have been hoped.

A slight variation on the above is

[3] I use O here rather than Θ because I do not mind much if the costs are less than linear.

$$g(n) = pg(n/q) + kn$$

with p and q integers. This arises when a problem of size n can be split into p sub-problems each of size n/q. If $p = q$ the solution grows like $n \log(n)$, while for $p > q$ the growth function is n^β with $\beta = \log(p)/\log(q)$.

A different variant on the same general pattern is

$$g(n) = g(\alpha n) + k, \quad \alpha < 1, \quad k > 0$$

where now a *fixed* amount of work reduces the size of the problem by a factor α. This leads to a growth function $\log(n)$.

1.2 Algorithm design

Before presenting collections of specific algorithms this section presents a number of ways of understanding algorithm design. None of these are guaranteed to succeed, and none are really formal recipes that can be applied, but they can still all be recognised among the methods documented later in the course.

1.2.1 Recognise 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.

1.2.2 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.

1.2.3 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 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.

In the most common and useful cases both the dividing and combining stages will have linear cost in terms of the problem size — certainly one expects them to be much easier tasks to perform than the original problem seemed to be. Mergesort will provide a classical illustration of this approach.

1.2.4 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 reasons best described as "historical" this process is known as dynamic programming. It has applications in various tasks related to combinatorial search — perhaps the simplest example is the computation of Binomial Coefficients by building up Pascal's triangle row by row until the desired coefficient can be read off directly. The computation of Fibonacci numbers is another trivial example.

More significant examples are covered in Cormen, Leiserson and Rivest. Let's briefly summarise one here: matrix chain multiplication.

If A is a $p \times q$ matrix and B is a $q \times r$ matrix, the product AB is a $p \times r$ matrix and can be computed using $p \times q \times r$ scalar multiplications. If we wish to compute the product of 6 matrices: $A_1A_2A_3A_4A_5A_6$ we have a wide choice of the order in which we do the multiplications. We naturally would like to choose an order which minimises 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 minimises the number of scalar multiplications needed. Suppose $m(i, j)$ is the minimum number of scalar multiplications to compute the product $A_i \ldots A_j$ then m can be defined as a recursive function based on:

$$m(i, j) = i = j \rightarrow 0, \min_{i \le k < j} \{m(i, k) + m(k + 1, j) + p_{i-1}p_kp_j\}$$

[Notation: $b \rightarrow x, y$ means **if** b **then** x **else** y]
For the above problem the answer is $m(1, 6) = 15125$ scalar multiplications. A simple implementation of this function will compute $m(i, j)$ in exponential time, but it can be computed 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 is to modify the simple minded recursive definition of m 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 before returning. This technique is known as memoisation.

1.2.5 Greedy algorithms

Many algorithms involve some sort of optimisation. 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. A classic example of this, not covered in this course, is the algorithm for finding a minimal spanning subtree of a weighted graph.

1.2.6 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. This first part will occasionally reach a dead end, and this is where the second, the backtracking, part comes in. It has kept extra information around about when the first part made choices, and it unwinds all calculations back to the most recent choice point then resumes the search down another path. The language Prolog makes an institution of this way of designing code. The method is of great use in many graph-related problems.

1.2.7 Hill climbing

Hill climbing is again for optimisation problems. It first requires that you find (somehow) some form of feasible (but presumably not optimal) solution to your problem. Then it looks for ways in which small changes can be made to this solution to improve it. A succession of these small improvements might lead eventually to the required optimum. Of course proposing a way to find such improvements does not of itself guarantee that a global optimum will ever be reached: as always the algorithm you design is not complete until you have proved that it always ends up getting exactly the result you need.

1.2.8 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 analyse 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. Shellsort can be viewed this way, as can the various elaborate ways of ensuring that binary trees are kept well balanced.

1.2.9 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 the bound might be constructed. A properly proved lower bound can also prevent wasted time seeking improvement where none is possible.

1.3 Data structures

Typical programming languages such as Pascal, C or Java provide primitive data types such as integers, reals, boolean values and strings. They allow these to be organised into arrays, 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 will 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 this data structure then calls for further code design for the implementation of other necessary but less frequently performed operations.

This section introduces some simple and fundamental data types. Variants of all of these will be used repeatedly in later sections as the basis for more elaborate structures.

1.3.1 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 and 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 out out 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. Note that languages like ML provide these facilities but not (in the core language) arrays, so sometimes it will be worth being aware when the fast indexing of arrays is essential for the proper implementation of an algorithm.

1.3.2 Singly-linked lists

These provide the simplest example of use of record data types, and of using links. In C one might define a node in a linked list as

```
struct node
{ int value;         /* Just do lists of integers here */
  struct node *next; /* Pointer to next node in the list */
};
```

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 (Section 1.4.1), and are the data structure of choice in many places where flexible representation of variable amounts of data is wanted.

1.3.3 Doubly-linked lists

A feature of lists is that from one item you can progress along the list in one direction very easily, but it is no way of going in the other direction (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

```
struct dll_node
{ int value;
  struct node *prev, *next;
};
```

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.

1.4 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 (the idea is generally considered good for program maintainability as well, but that is no great concern for this particular course). 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.

1.4.1 The STACK abstract data type

Here is a specification for an Abstract Data Type called STACK:

`make_empty_stack():` manufactures an empty stack.

`is_empty_stack(s):` `s` is a stack. Returns `TRUE` if and only if it is empty.

`push(x,s):` `x` is an integer, `s` is a stack. Returns a non-empty stack which can be used with `top` and `pop`. `is_empty_stack(push(x,s))=FALSE`.

`top(s):` `s` is a non-empty stack; returns an integer. `top(push(x,s))=x`.

`pop(s):` `s` is a non-empty stack; returns a stack. `pop(push(x,s))=s`.

The idea here is that the definition of an ADT is forced to collect all the essential details and assumptions about how a structure must behave (but the expectations about common patterns of use and performance requirements are generally kept separate). It is then possible to look for different ways of mechanising 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(make_empty_stack())`. This 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 this must be specified just as clearly as it specifies all other behaviour.

The ADT for a stack given above does not make allowance for the `push` operation to fail, although on any real computer with finite memory it will inevitably be possible to do enough successive pushes to exhaust some resource. This limitation of a practical realisation 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. In this case the `push` and `pop` operations work by modifying stacks in place, so after use of `push(x,s)` the original `s` is no longer available. 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.

1.4.2 The LIST abstract data type

The type LIST will be defined by specifying the operations that it must support. The 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 but distinct. Such issues will be ducked for now. Also type-checking issues about the types of items stored in lists will be skipped over here, although most examples that just illustrate the use of lists will use lists of integers.

`make_empty_list():` manufactures an empty list.

`is_empty_list(s):` `s` is a list. Returns `TRUE` if and only if `s` is empty.

`cons(x,s):` `x` is anything, `s` is a list. `is_empty_list(cons(x,s))=FALSE`.

`first(s):` `s` is a non-empty list; returns something. `first(cons(x,s))=x`

`rest(s):` `s` is a non-empty list; returns a list. `rest(cons(x,s))=s`

`set_rest(s,s'):` `s` and `s'` are both lists, with `s` non-empty. After this call `rest(s')=s`, regardless of what `rest(s)` was before.

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 `set_first` operation to update list contents (as well as chaining) in place, or a `equal` test to see if two non-empty lists were manufactured by the same call to the `cons` operator. Applications of lists that do not need `set_rest` may be able to use different implementations of lists.

1.4.3 The QUEUE abstract data types

In the STACK ADT given earlier as an example, the item removed by the `pop` operation was the most recent one added by `push`. A QUEUE[4] 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. Looking at their ADTs suggests that stacks and queues will have very similar interfaces. It is sometimes possible to take an algorithm that uses one of them and obtain an interesting variant by using the other.

1.4.4 The TABLE abstract data type

This section is going to concentrate on finding information that has been stored in some data structure. The cost of establishing the data structure to begin with will be thought of as a secondary concern. As well as being important in its own right, this is a lead-in to a later section which extends and varies the collection of operations to be performed on sets of saved values.

For the purposes of this description we will have just one table in the entire universe, so all the table operations implicitly refer to this one. Of course a more general model would allow the user to create new tables and indicate which ones were to be used in subsequent operations, so if you want you can imagine the changes needed for that.

`set(key,value)`: This stores a value in the table. At this stage the types that keys and values have is considered irrelevant.

`get(key)`: If for some key value k an earlier use of `set(`k, v`)` has been performed (and no subsequent `set(`k, v'`)` followed it) then this retrieves the stored value v.

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 will 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 $0, \ldots, n$ 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 $a, \ldots, b$) then subtracting a from key values gives a suitable index for use with a vector.

[4]sometimes referred to a FIFO: First In First Out.

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 is good.

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 on the front of the list, assured that `get` will 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 (at the start or the end of the list?). 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

$$f(n) = f(n/2) + \Theta(1)$$

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

Chapter 2

Sorting

This is a big set-piece topic: any course on algorithms is bound to discuss a number of sorting methods. The volume 3 of Knuth is dedicated to sorting and the closely related subject of searching, so don't think it is a small or simple topic! However much is said in this lecture course there is a great deal more that is known.

2.1 Basic concepts

2.1.1 Minimum cost of sorting

If I have n items in an array, and I need to end up with them in ascending order, there are two low-level operations that I can expect to use in the process. The first takes two items and compares them to see which should come first. 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. The second critical operation is that of rearranging data in the array, and it will prove convenient to express that in terms of "interchanges" which swap the contents of two nominated array locations.

In extreme cases either comparisons or interchanges[1] may be hugely expensive, leading to the need to design methods that optimise 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: If there are n items in an array then $\Theta(n)$ exchanges suffice to put the items in order. In the worst case $\Theta(n)$ exchanges are needed. Proof: identify the smallest item present, then 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.

[1] Often if interchanges 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.

The bound is $n-1$ not 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. 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 I must perform exchanges that touch each position in the array, and that certainly means I need $n/2$ exchanges, which is good enough to establish the $\Theta(n)$ growth rate. Tighter analysis should show that a full $n-1$ exchanges are in fact needed in the worst case.

Assertion: Sorting by pairwise comparison, assuming that all possible arrangements of the data are equally likely as input, necessarily costs at least $\Theta(n\log(n))$ comparisons. Proof: 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 $\log(n!)$ is about $n\log(n)$. 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 and 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 the $log(n!) = \Theta(n\log(n))$.

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

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

$$\log(n!) \leq n\log(n)$$

The first $n/2$ terms are all greater than or equal to $\log(n/2) = \log(n) - 1$, so

$$\log(n!) \geq \frac{n}{2}(\log(n) - 1)$$

Thus for large enough n, $\log(n!) \geq kn\log(n)$ where $k = 1/3$, say.

2.1.2 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 what order the corresponding records should end up in in the output list. "Stable" sorting demands that in such cases the order of items in the input is 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 an arbitrary sorting method will rearrange the data in a stable way. This clearly increases overheads a little.

2.2 Selection sort

We saw earlier that an array with n items in it could be sorted by performing $n-1$ exchanges. This provides the basis for what is perhaps the simplest sorting algorithm: 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 array comparing each successive item with the smallest one found earlier. If there are m items to scan then the minimum finding clearly costs $m-1$ comparisons. The whole insertion sort process does this on sub-arrays of size $n, n-1, \ldots, 1$. Calculating the total number of comparisons involved requires summing an arithmetic progression: after lower order terms and constants have been discarded we find that the total cost is $\Theta(n^2)$. 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 organisation of the data.

2.3 Bubble sort

Due its continued popularity as a beginner's demonstrator for sorting, bubble sort is probably the best known quadratic sorting algorithm. It is also one of the simplest to describe and analyse. In its most basic form bubble sort iterates $n-1$ times, each time comparing every adjacent pair of values and exchanging any that are out of order. Clearly the number of comparisons is $(n-1)^2$ and so bubble sort has quadratic time complexity. One common enhancement is to exit the outer loop early if an iteration completes without performing any exchanges. This can allow bubble sort to complete in better than quadratic time for nearly-sorted inputs. Despite this, the next sorting algorithm we will look at is nearly always preferable to bubble sort.

2.4 Insertion sort

Insertion sort is another classic quadratic-complexity sorting algorithm. When the first k items of the array have been sorted the next is inserted in place by letting it sink to its rightful place: it is compared against item k, and if less a swap moves it down. If such a swap is necessary it is compared against position

$k-1$, and so on. This clearly has worst case costs $\Theta(n^2)$ in both comparisons and data movement. It does however compensate a little: if the data was originally already in the right order then insertion sort does no data movement at all and only does $n-1$ comparisons, and is optimal. Insertion sort is the method of practical choice when most items in the input data are expected to be close to the place that they need to end up.

Now suppose that data movement is very cheap, but comparisons are very expensive. Rather than linearly scanning elements k, $k-1$ and so on to determine where to sink the next item, a binary search can be used to identify where the item should go, and this search can be done in $\lceil \log_2(k) \rceil$ comparisons. Then some number of exchange operations (at most k) put 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

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

which is bounded by $\log((n-1)!) + n$. This effectively attains the lower bound for general sorting by comparisons that we set up earlier. But remember that this method still has high (typically quadratic) data movement costs.

2.5 Shell sort

Shell sort is an elaboration on insertion sort that looks at its worst aspects and tries to do something about them. The idea is to precede the insertion sort with something that gets items to roughly their correct positions, in the hope that the insertion sort will then have linear cost. Shell sort does this by executing a collection of sorting operations on subsets of the original array. If s is some integer then a stride-s sort will sort s subsets of the array — the first of these will be the one with elements at positions $0, s, 2s, 3s, \ldots$, the next will use positions $1, s+1, 2s+1, 3s+1, \ldots$, and so on. Such sub-sorts can be performed for a sequence of values of s starting large and gradually shrinking so that the last pass is a stride-1 sort (which is just an ordinary insertion sort). Now the interesting questions are whether the extra sorting passes pay their way, what sequences of stride values should be used, and what will the overall costs of the method amount to?

It turns out that there are definitely some bad sequences of strides, and that a simple way of getting a fairly good sequence is to use the one which ends $\ldots 13, 4, 1$ where $s_{k-1} = 3s_k + 1$. For this sequence it has been shown that Shell sort's costs grow at worst as $n^{1.5}$, but the exact behaviour of the cost function is not known, and is probably distinctly better than that. This must be one of the smallest and most practically useful algorithms that you will come across where analysis has got really stuck — for instance the sequence of strides given above is known not to be the best possible, but nobody knows what the best sequence is.

Although Shell sort does not meet the $\Theta(n\log(n))$ target for the cost of sorting, it is easy to program and its practical speed on reasonable size problems is fully acceptable.

2.6 Quicksort

The idea behind Quicksort is quite easy to explain, and when properly implemented and with 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 (eg. if all the items in the input list are identical), and although in almost all cases Quicksort executes in time proportional to $n\log(n)$ (with a quite small constant of proportionality) for worst-case input data it can be as slow as n^2. It is strongly recommended that you study the description of Quicksort in one of the textbooks 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, off-by-one errors, and so on.

The idea behind Quicksort is to select some value from the array and use that as a "pivot". A selection procedure partitions the values so that the lower portion of the array holds values less than the pivot and the upper part holds only larger values. This selection can be achieved by scanning in from the two ends of the array, exchanging values as necessary. 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.

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 $n\log(n)$. But in the worst case the array might be split very unevenly — perhaps at each step only one item 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! Then it is easy to set up a recurrence formula that will be satisfied by the average cost:

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

where the sum adds up the expected costs corresponding to all the (equally probable) ways in which the partitioning might happen. This is a jolly equation to solve, and after a modest amount of playing with it it can be established that the average cost for Quicksort is $\Theta(n\log(n))$.

Note also that the space complexity of Quicksort is not $\Theta(1)$ because of the recursive invocations, each of which will require its own copy of local variables and other function overheads. Although in most cases the space overhead is $\Theta(\log(n))$ a straightforward implementation will have worst-case space overhead $\Theta(n)$.

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 extremely good it does have a quite unsatisfactory (albeit uncommon) worst case. It should therefore not be used in applications where the worst-case costs are unacceptable (perhaps because of safety implications). The decision about whether to use Quicksort for average good speed or a slightly slower but guaranteed $n\log(n)$ method can be a delicate one.

There are various improvements and variations on the Quicksort algorithm that can yield better performance. One common enhancement is to recurse only on the smaller partition and deal with the larger partition using iteration. Many compilers will perform this optimisation automatically (by detecting the 'tail recursion') and it reduces the worst-case space overhead to $\Theta(\log(n))$. Another optimisation is to use a simpler sorting method for small partitions, where a simple inner loop is more important than asymptotic complexity. One trick is to ignore small partitions completely and instead execute a final insertion sort over the entire array (*Exercise:* why does this have linear behaviour?).

2.7 Heapsort

Despite its good average behaviour there are circumstances where one might want a sorting method that is guaranteed to run in time $n\log(n)$ whatever the input. Despite the fact that 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 use of this term quite unrelated to the use of the term "heap" in free-storage management) has many other applications.

Consider an array that has values stored in it subject to the constraint that the value at position k is greater than (or equal to) those at positions $2k$ and $2k+1$.[2] The data in such an array is referred to as a heap. The root of the heap is the item at location 1, and it is clearly the largest value in the heap.

[2] supposing that those two locations are still within the bounds of the array

Heapsort consists of two phases. The first takes an array full or arbitrarily ordered data and rearranges it so that the data forms a heap. Amazingly this can be done in linear time. The second stage 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 elements. Repeating this step will leave the full set of data in order in the array. Each heap reconstruction step has a cost proportional to the logarithm of the amount of data left, and thus the total cost of heapsort ends up bounded by $n \log(n)$.

2.8 Mergesort

Quicksort and Heapsort both work in-place, i.e. they do not need any large amounts of space beyond the array in which the data resides. If this constraint can be relaxed then a fast and simple alternative is available in the form of Mergesort. Observe that 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. The resulting sorted array has to be separate from the two input ones.

This observation leads naturally to the familiar $f(n) = 2f(n/2) + kn$ recurrence for costs, and this time there are no special cases or oddities. Thus Mergesort guarantees a cost of $n \log(n)$, is simple and has low time overheads, all at the cost of needing the extra space to keep partially merged results.

In practice, $n/2$ words of workspace is sufficient for a straightforward implementation.

2.9 Extras

2.9.1 Quantitative comparison

Attached to these notes, and available on the course materials webpage, you will find simple C implementations of the sorting methods discussed so far. As well as providing reasonable reference implementations they also allow us to measure and observe the behaviour of the various algorithms in practice.

Figure 2.1 gives the number of instructions executed to sort a vector of 100,000 integers using seven different methods and four different settings of the initial data. The data settings are:

Ordered: The data is already order.

Reversed: The data is in reverse order.

Random: The data consists of random integers in a wide range (very few duplicates).

Random 0..999: The data consists of random integers in the range 0..9 (very many duplicates).

Method	Ordered	Reversed	Random	Random 0..9
insertion	0.000s	7.903s	4.058s	3.524s
selection	7.399s	8.091s	7.236s	7.466s
bubble	15.699s	50.054s	54.328s	51.594s
shell	0.005s	0.007s	0.027s	0.010s
quick	5.985s	6.702s	0.016s	0.010s
merge	0.023s	0.029s	0.037s	0.033s
heap	0.027s	0.025s	0.034s	0.024s

Figure 2.1: Execution times for various sorting algorithms.

It is interesting to note that insertion sort is indeed a very good method to choose when the data is already very nearly sorted. Also, there is clearly a price to pay for guaranteed $n \log(n)$ behaviour: heapsort and mergesort are consistently slower than shell sort, and slower than quicksort for random data. The poor performance of quicksort for ordered data is due to the simple-minded implementation which does not take any care to pick a good pivot value.

Figures 2.2 and 2.3 show the behaviour of three sorting algorithms in terms of the array indexes they access over time[3].

2.9.2 Order statistics

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 us productive to generalise to find the item that ranks at position k in the data. For a total n items, the median corresponds to taking the special case $k = n/2$. Clearly $k = 1$ 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 $n \log(n)$ for the sorting.

Two variants on Quicksort are available that solve the problem. One has linear cost in the average case, but has a quadratic worst-case cost; it is fairly simple.

[3]The apparent periodic pattern in selection sort accesses should be ignored. I plot only one array access per hundred and this produces artefacts when coupled with the very uniform and repetitive access pattern of selection sort.

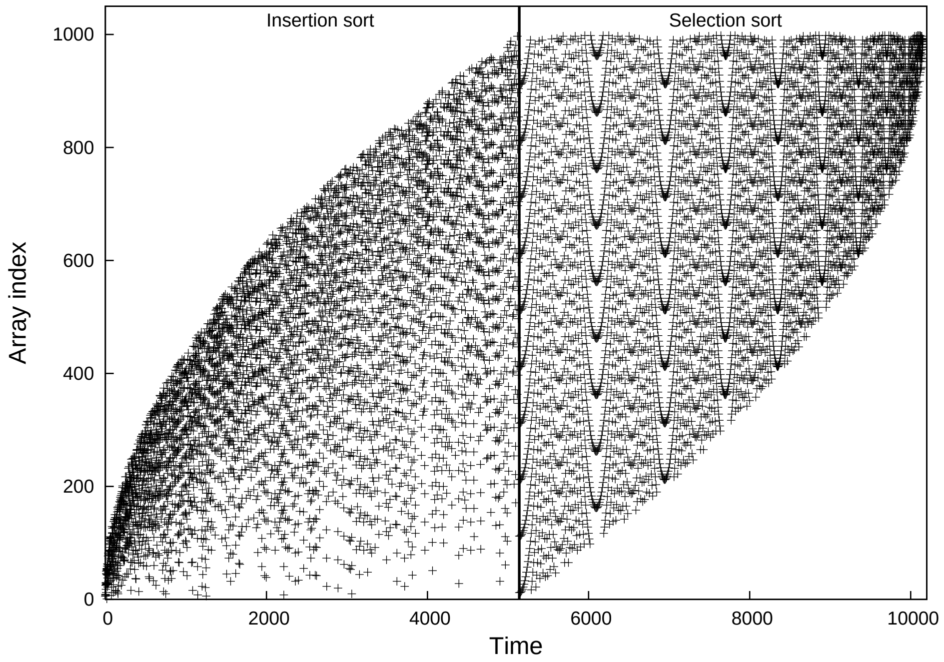


Figure 2.2: Array-access maps for quadratic sorting methods (insertion sort and selection sort).

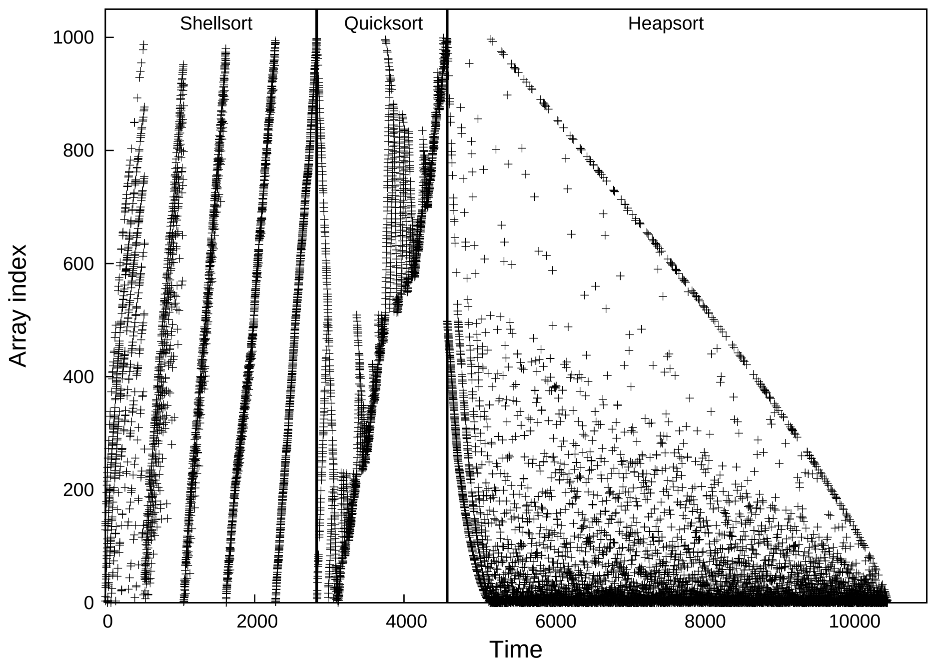


Figure 2.3: Array-access maps for faster sorting methods (shellsort, quicksort and heapsort).

The other is more elaborate to code and has a much higher constant of proportionality, but guarantees linear cost. In cases where guaranteed performance is essential the second method may have to be used.

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, that each selection stage divides out values neatly into two even sets) is $f(n) = f(n/2) + Kn$, and the solution to this exhibits linear growth.

The more elaborate method works hard to ensure that the pivot used will not fall too close to either end of the array. It starts by clumping the values into groups each of size 5. It selects the median value from each of these little sets. It then calls itself recursively to find the median of the $n/5$ values it just picked out. This is then the element it uses as a pivot. The magic here is that the pivot chosen will have $n/10$ medians lower than it, and each of those will have two more smaller values in their sets. So there must be $3n/10$ values lower than the pivot, and equally $3n/10$ larger. This limits the extent to which things are out of balance. In the worst case after one reduction step we will be left with a problem $7/10$ of the size of the original. The total cost now satisfies

$$f(n) = An/5 + f(n/5) + f(7n/10) + Bn$$

where A is the (constant) cost of finding the median of a set of size 5, and Bn is the cost of the selection process. Because $n/5 + 7n/10 < n$ the solution to this recurrence grows just linearly with n.

2.9.3 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 $n \log(n)$. Two particular cases are common enough to be of at least occasional importance.

The **counting sort** can be used when the values to be sorted are integers that live in a known range, and where the range is smaller than the number of values to be processed. 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 be the amount of data to be sorted) and initialise it to zero. 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 in the array size and in the size of the key range:

this method becomes increasingly unattractive as the possible key range expands. 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 where in the output array data should be moved to. This does not compromise the overall linear cost.

Another method, the **bucket sort** is used 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 (see later in these notes!), and this prediction is used to insert items in it, then apart from some local effects of clustering that data has been sorted.

Chapter 3

Searching

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 sub-sets of the possible operations can be best optimised leads to the discussion of a large range of sometimes quite elaborate representations and procedures. It would be possible to fill a whole long lecture course with a discussion of the options, but here just some of the more important (and more interesting) will be covered.

3.1 The abstract data type

In the following `S` stands for a set, `k` is a key and `x` is an item present in the set. It is supposed that each item contains a key, and that the keys are totally ordered.

`make_empty_set(), is_empty_set(S)`: basic primitives for creating and testing for empty sets.

`insert(S,x)`: Modify the set `S` so as to add a new item `x`.

`search(S,k)`: Discover if an item with key `k` is present in the set, and if so return it. If not return that fact.

`delete(S,k)`: If `k` is present in the set `S` then change `S` to remove that item.

`sort(S)`: Returns a list of the items in `S` in sorted order.

3.2 Linked lists

Before looking at more complex data structures, it is worthwhile considering representing set information in a simple singly-linked list. If we keep the list items in sorted order then clearly `iterate` can be implemented in $O(n)$ time, but unfortunately that is also the time complexity for `insert`, `search` and `delete` (How many list items must they visit on average, assuming a uniform distribution of keys?).

If we know that searches usually query only a small subset of the items in the list then it may make sense to implement a *move-to-front* list. This adds a second stage to the search operation: after finding the required item it is deleted from the list and reinserted as the head item. (How does this affect the performance of an unsuccessful search? What about `sort`?).

3.3 Hash tables

Even if the keys used do have an order relationship associated with them it may be worthwhile looking for a way of building a table without using it. Binary search makes locating things in a table easier by imposing a very good coherent structure — hashing 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. Two variants arise.

Chaining. We can arrange that the locations in the array hold little linear lists that collect all keys that has 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.

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.

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 do pretty well. If the number of items stored is much smaller than the size of the hash table both adding and retrieving data should have constant (i.e.$\Theta(1)$) cost. The analysis of expected costs for tables that have a realistic load is of course more complex.

3.4 Binary search trees

For `insert`, `search` and `delete` it is very reasonable to use binary trees. Each node will contain an item and references to two sub-trees, one for all items lower than the stored one and one for all that are higher. Searching such a tree is simple. 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.

```
class Node {
    int k, v;   /* key, value (item) */
    Node l, r;  /* left, right children */
};

int search(Node t, int k) {
    while (t.k != k) {
        if (t.k < k) t = t.r; else t = t.l;
        if (t == null) return -1;
    }
    return t.v;
}
```

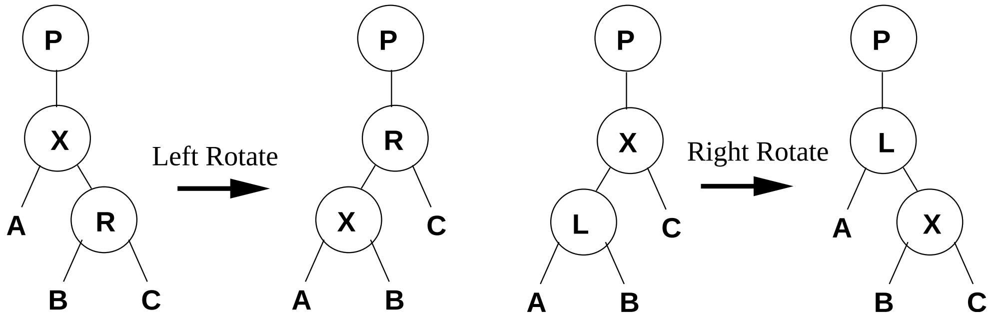
To insert in a tree one searches to find where the item ought to be and then insert there. Deleting a leaf node is easy. To delete a non-leaf feels harder, and there will be various options available. One will be to exchange the contents of the non-leaf cell with either the largest item in its left subtree or the smallest item in its right subtree. 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.

If trees are created by inserting items in random order they usually end up pretty well balanced, and all operations on them have cost proportional to their depth, which will be $\log(n)$. A worst case is when a tree is made by inserting items in ascending order, and then the tree degenerates into a list. It would be nice to be able to re-organise 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 one of the more sensible compromises.

3.5 Tree rotations

An essential element of the guaranteed $\log(n)$ BST algorithms we will look at is a local transformation called a *rotation*. A single rotation affects just two nodes and three links and does *not* affect the ordering properties that define a BST. The locality of rotations make them a simple but powerful tool for restructuring a tree to make searches more efficient.

Two forms of rotation may be applied to a BST node X: *left rotation* affects X and its right child R, making X the left child of R; *right rotation* affects X and its left child L, making X the right child of L. These are illustrated below:



One immediate application of these transformations is to implement a *move-to-root* tree, similar to the move-to-front list we considered earlier. The pseudocode for this is surprisingly straightforward if we define it recursively. The search returns `null` if the key is not in the tree, otherwise it returns a reference to the node, which is now root of the tree:

```
Node search(Node t, int k) {
    Node x;
    if (t == null) return null;
    if (k < t.k) {
        x = search(t.l, k);        /* new subtree root, or null */
        if (x == null) return null;
        t.l = x.r; x.r = t; t = x; /* right rotate */
    } else if (k > t.k) {
        x = search(t.r, k);        /* new subtree root, or null */
        if (x == null) return null;
        t.r = x.l; x.l = t; t = x; /* left rotate */
    }
    return t;
}
```

Note that I am not claiming any special properties beyond those of a BST for the simple move-to-root approach. It is intended as a demonstration of the power of simple localised transformations to have a significant effect on tree structure. We will wield this power later to build search trees with provably better worst-case bounds than an 'unbalanced' BST.

3.6 2-3-4 Trees

Binary trees had one key and two pointers in each node. The leaves of the tree are indicated by null pointers. 2-3-4 trees generalise 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 realisation that with a larger branching factor 2-3-4 trees will generally be shallower than pure binary trees.

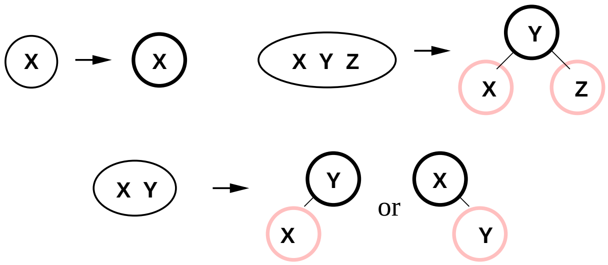
Inserting into a 2-3-4 node also turns out to be fairly easy, and what is even better is that it turns out that a simple insertion process automatically leads to balanced trees. Search down through the tree looking for where the new item must be added. If the place where it must be added is a 2-node or a 3-node then it can be stuck in without further ado, converting 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 it. 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 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 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 $\log(n)$ down from the root.

I will not discuss deletions from trees here, although once you have mastered the details of insertion it should not seem (too) hard.

3.7 Red-black trees

It might be felt wasteful and inconvenient to have trees with three different sorts of nodes, or ones with enough space to be 4-nodes when they will often want to be smaller. A way out of this concern is to represent 2-3-4 trees in terms of binary trees that are provided with one extra bit per node. The idea is that a red binary node is used as a way of storing extra pointers, while a black node stands for a regular 2-3-4 node. The resulting trees are known as red-black trees. 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 tree is at worst twice that of a 2-3-4 tree. These are the equivalences between 2-3-4 nodes and red-black subtrees:

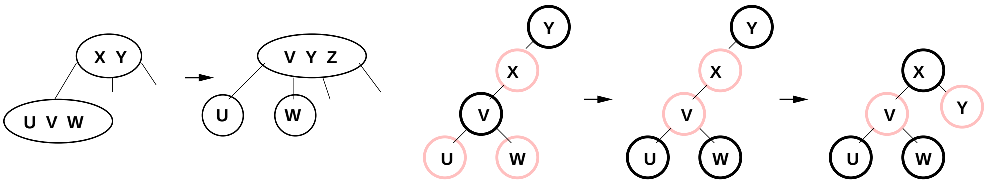


Two rules must be obeyed by any valid red-black tree. They follow fairly directly from the above equivalences between 2-3-4 and red-black nodes:

1. If a node is red then both its children are black.

2. Every simple path from a node to a descendant leaf contains the same number of black nodes.

Insertions and node splitting in red-black trees just has to follow the rules that were set up for 2-3-4 trees. This turns out to be very simple: splitting a root 4-node into three 2-nodes is a simple recolouring operation (the two red children become black). Splitting a non-root 4-node flips the colours of the three red-black nodes. This raises one complication, if the parent is a 3-node (it cannot be a 4-node of course) oriented the 'wrong way'. In that case a rotation and further recolouring is required to construct a valid red-black 4-node, as illustrated below.



Textbooks often describe a bottom-up algorithm for inserting into red-black trees. This is a little more complex than the top-down method, but it helps to remember the isomorphism between 2-3-4 and red-black trees, and take into account that if you work purely in the 'red-black' space you may choose certain tree transformations that are not obvious from the 2-3-4 representation (perhaps because they require fewer rotations, which is an issue only in the red-black representation). An excellent exercise for the reader is to sketch the equivalent 2-3-4 transformations for the red-black insertion and deletion cases presented in Cormen, Leiserson and Rivest.

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 not that complicated.

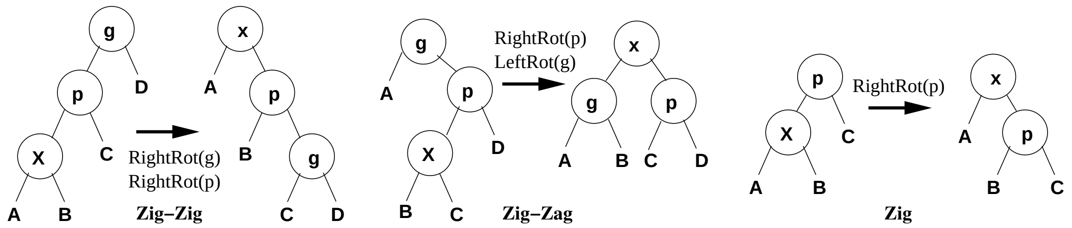


Figure 3.1: The three splay transformations (zig-zig, zig-zag and zig) when the node to be splayed (`x`) is the left child of its parent. The rotation steps are listed alongside each illustration (left or right rotation, and the node it is applied to).

3.8 Splay trees

A splay tree is a simple binary tree representation combined with a move-to-root strategy that gives it the following remarkably attractive properties.

1. The amortised cost of each operation is $O(\log(n))$ for trees with n nodes.
2. The cost of a sequence of splay operations is within a constant factor of the same sequence of accesses into any static tree, even one that was built explicitly to perform well on the sequence.
3. Asymptotically, the cost of a sequence of splay operations that only accesses a subset of the elements will be as though the elements that are not accessed are not present at all.

The proof of these properties is beyond the scope of this course. By *amortised complexity* we mean the per-operation cost averaged over a long enough sequence of operations.

In a simple implementation of splay trees, each node contains a key, a value, left and right child pointers, and a parent pointer. Any child pointer may be null, but only the root node has a null parent pointer. Apart from this the representation is identical to that of an ordinary binary search tree.

Whenever an `insert` or `lookup` operation is performed, the accessed node (with key `X`, say) is promoted to the root of the tree using the sequence of transformations illustrated in Figure 3.1. Note that only the transformations in which the splay node is the *left* child of its parent are included. The excluded cases are exactly symmetric. The final transformation (involving only two nodes rather than three) is used only when `X` is within one position of the root. Note that these transformations are exactly the same as performing a series of single rotations at the parent of `X`, *except* for the 'zig-zig' case which first rotates at the grandparent. Without this special double-rotation rule the desirable properties of splay trees are lost.

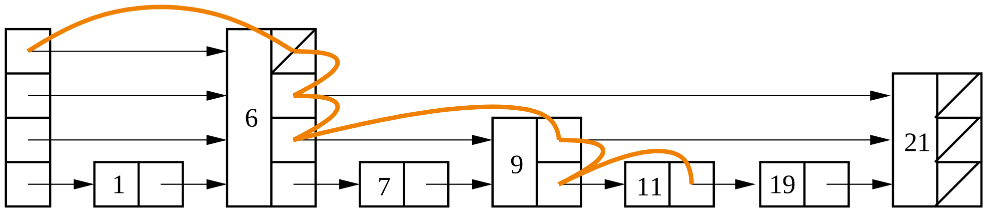


Figure 3.2: Example of searching in a skip list.

3.9 Skip lists

The final search structure we will look at takes us back to the singly-linked list, but with a twist. The insight is that the linear search cost of the linked list could be avoided if we could jump quickly to roughly the correct position in the structure. Skip lists achieve this by maintaining multiple *levels* of linked list, which we will calls levels $0 \ldots n-1$. Level 0 is a regular linked list: every item is included in it in sorted order. Level k (> 0) is expected to contain approximately half the nodes that are present in level $k-1$ (and it contains no nodes that are not present in level $k-1$). This is achieved by "tossing a coin" whenever a new item is inserted into the structure, to decide up to what level the node should be inserted: the probability of inserting up to level k is chosen to be roughly $1/2^{k+1}$.

Figure 3.2 shows an example skip list, and the strategy for searching it (in this case, for key 11). Note how we have n head pointers rather than just one. A search proceeds by searching at level $n-1$ until it finds a key greater than or equal to the search key. It then drops to the next level down and continues in the same way until it reaches the lowest level.

Assuming a good distribution of nodes at each level in the structure, the skip list achieves $\log(n)$ search times for any distribution of input keys without the implementation complexity of a binary serach tree. However, note that the search loop is more complex than that of a binary search tree and this tends to hurt performance when the structure is mostly read-only.

Conclusion

One of the things that is not revealed very directly by these notes is just how much detail will appear in the lectures when each topic is covered. The various textbooks recommended range from informal hand-waving with little more detail than these notes up to heavy pages of formal proof and performance analysis. In general you will have to attend the lectures to discover just what level of coverage is given, and this will tend to vary slightly from year to year.

The algorithms that are discussed here (and indeed many of the ones that have got squeezed out for lack of time) 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.