Data Structures and Algorithms

CST IB, CST II(General) and Diploma

Martin Richards `mr@cl.cam.ac.uk`

October 2000

Computer Laboratory
University of Cambridge

(Version November 14, 2000)

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.

1 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 possiblities 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 used 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.

The subject is a young one and many of the the algorithms I will be covering had not been discovered when I attended a similar course as a Diploma student in 1962. 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.

2 Course content and textbooks

Even a cursory inspection of standard texts related to this course should be daunting. There as 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. From year to year some of the other topics will change, and this includes the possibility that lectures will cover material not explicitly mentioned in these notes.

A range of text books will be listed here, and the different books suggested all have quite different styles, even though they generally agree on what topics to cover. It will make good sense to take the time to read sections of several of them in a library before spending a lot of money on any – different books will appeal to different readers. All the books mentioned are 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.

Corman, Leiserson and Rivest, "Introduction to Algorithms". A heavyweight book at 1028 pages long, and naturally covers a little more material at slightly greater deapth than the other texts listed here. It includes careful mathematical treatment of the algorithms that it discusses, and would be a natuaral candidate for a reference shelf. Despite its bulk and precision this book is written in a fairly friendly and non-daunting style, and so against all expectations raised by its length it is my first choice suggestion. The paperback edition is even acceptably cheap.

Sedgewick, "Algorithms" (various editions) is a repectable and less daunting book. As well as a general version, Sedgewick's book comes in variants which give sample implementations of the algorithms that it discusses in various concrete programming languages. I suspect that you will probably do as well to get the version not tied to any particular language, but its up to you. I normally use the version based on C.

Aho, Hopcroft and Ullman, “Data, Structures and Algorithms”. Another good book by well-established authors.

Knuth, “The Art of Computer Programming, vols 1-3”. When you look at the date of publication of this series, and then observe that it is still in print, you will understand that it is a classic. Even though the presentation is now outdated (eg. many procedures are described by giving programs for them written in a specially invented imaginary assembly language called MIX), and despite advances that have been made since the latest editions this is still a major resource. Many algorithms are documented in the form of exercises at the end of chapters, so that the reader must either follow through to the original author’s description of what they did, or to follow Knuth’s hints and re-create the algorithm anew. The whole of volume 3 (not an especially slender tome) is devoted just to sorting and searching, thus giving some insight into how much rich detail can be mined from such apparently simple problems.

Manber, “Introduction to Algorithms” is strong on motivation, case sudies and exercises.

Salomon, “Data Compression” is published by Springer and gives a good introduction to many data compression methods including the Burrows-Wheeler algorithm.

Your attention is also drawn to **Graham, Knuth and Patashnik “Concrete Mathematics”**. It provides a lot of very useful background and could well be a great help for those who want to polish up their understanding of the mathematical tools used in this course. It is also an entertaining book for those who are already comfortable with these techniques, and is generally recommended as a “good thing”. It is especially useful to those on the Diploma course who have had less opportunity to lead up to this course through ones on Discrete Mathematics.

The URL `http://hissa.ncsl.nist.gov/~black/CRCDict` by Paul Black is a potentially useful dictionary-like page about data structures and algorithms.

3 Related lecture courses

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.

1B students will be able to look back on the 1A Discrete Mathematics course, and should therefore be in a position to understand (and hence if necessary reproduce in examination) the analysis of recurrence formulae that give the computing time of some methods, while Diploma and Part 2(G) students should take these as results just quoted in this course.

Finite automata and regular expressions arise in some pattern matching algorithms. These are the subject of a course that makes a special study of the capabilities of those operations that can be performed in strictly finite (usually VERY small amounts of) memory. This in turn leads into the course entitles "Computation Theory" that explores (among other things) just how well we can talk about the limits of computability without needing to describe exactly what programming language or brand of computer is involved. A course on algorithms (as does the one on computation theory) assumes that computers will have as much memory and can run for as long as is needed to solve a problem. The later course on "Complexity Theory" tightens up on this, trying to establish a class of problems that can be solved in "reasonable" time.

4 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.

For the purpose of guessing what examination question might appear, two suggestion can be provided. The first involves checking past papers for questions relating to this course as given by the current and previous lecturers — there will be plenty of sample questions available and even though the course changes slightly from year to year **most** past questions will still be representative of what could be asked this year. A broad survey of past papers will show that from time to time successful old questions have been recycled: who can tell if this practice will continue? The second way of spotting questions is to inspect these notes and imagine that the course organisers have set one question for every 5 cm of printed notes (I believe that the density of notes means that there is about enough material covered to make this plausible).

It was my intention to limit the contents of this course to what is covered well in the Sedgewick and CLR books to reduce the number of expensive books you will need. I have failed, but what I have done is to give fuller treatment in these notes of material not covered in the standard texts. In addition, I plan to place fragments of code, algorithm animations and copies of relevant papers on the Web. They will, I hope, be accessible via: `http://www.cl.cam.ac.uk/Teaching/2000/DSAlgs/`. For convenience, copies of some of the files will reside in `/group/clteach/mr/dsa/` on Thor.

5 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.

5.1 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 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.

5.2 Big-Θ 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 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 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** $O(x^2)$

[1] Time in seconds, perhaps

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.

5.3 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 $n > 25$ the $2^n > 1000000n$ — 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].

5.4 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

[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.

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.

5.5 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 maintainablity 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.

To help make this clearer, 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.[3]

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

[3]There are real technical problems associated with the "=" sign here, but since this is a course on data structures not an ADTs it will be glossed over. One problem relates to whether s is in fact still valid after **push(x, s)** has happened. Another relates to the idea that equality on data structures should only relate to their observable behaviour and should not concern itself with any user-invisible internal state.

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 must 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(**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.

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

5.6 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.

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 assummption 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.

5.7 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 — an 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.

5.8 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 applica- bility 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.

A good example of where amortised analysis is helpful is garbage collection (see later) where it allows the cost of a single large expensive storage reorganisation 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) amortised analysis is not appropriate for use where real-time constraints apply.

6 Simple Data Structures

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.

6.1 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[4]. The introduction of record types naturally introduces the use of pointers. Note that languages like ML provide these facility 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. Another issue made visible by ML is that of updatability: in ML the special constructor **ref** is needed to make a cell that can have its contents changed. Again it can be worthwhile to observe when algorithms are making essential use of update-in-place operations and when that is only an incidental part of some particular encoding.

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.

6.2 "LIST" as an 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.

[4] Ways of arranging this are discussed later

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.

6.3 Lists implemented using arrays and using records

A simple and natural implementation of lists is in terms of a record structure. In C one might write

```
typedef struct Non_Empty_List
{ int first;          /* Just do lists of integers here */
  struct List *rest; /* Pointer to rest */
} Non_Empty_List;

typedef Non_Empty_List *List;
```

where all lists are represented as pointers. 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 ML the analogous declaration would be

```
datatype list = empty |
                non_empty of int * ref list;
fun make_empty_list() = empty;
fun cons(x, s) = non_empty(x, ref s);
fun first(non_empty(x,_)) = x;
fun rest(non_empty(_, s)) = !s;
```

where there is a little extra complication to allow for the possibility of updating the rest of a list. A rather different view, and one more closely related to real machine architectures, will store lists in an array. The items in the array will be similar to the C `Non_Empty_List` record structure shown above, but the `rest` 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 ADT.

Controlling the allocation of array items in applications such as this is the subject of a later section.

If it can be arranged that the data used to represent the first and rest components of a non-empty list are the same size (for instance both might be held as 32-bit values) 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 **rest** 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) 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).

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

The links in lists make it easy to splice items out from the middle of lists or add new ones. 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.

6.4 Double-linked Lists

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 **rest** 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 Double Linked List) containing operators

LHS_end: a marker used to signal the left end of a DLL.

RHS_end: a marker used to signal the right end of a DLL.

rest(s)**:** s is DLL other than RHS_end, returns a DLL.

previous(s)**:** s is a DLL other than LHS_end; returns a DLL. Provided the rest and previous functions are applicable the equations $\mathbf{rest}(\mathbf{previous}(s)) = s$ and $\mathbf{previous}(\mathbf{rest}(s)) = s$ hold.

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.

6.5 Stack and queue abstract data types

The STACK ADT was given earlier as an example. Note that the item removed by the **pop** operation was the most recent one added by **push**. A QUEUE[5] is in most respects similar to a stack, but the rules are changed so that the item

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

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.

6.6 Vectors and Matrices

The Computer Science notion of a vector is of something that supports two operations: the first takes an integer index and returns a value. The second operation 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 realisations.

At this stage in the course I will just think about implementing vectors as 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 the array.

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

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

6.7 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 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 with the edges being resistors joining various points in it) then the matrix might have integer elements (say) instead of boolean ones, 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 (and "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.

7 Ideas for 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.

7.1 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. See the Graham Scan method for finding a convex hull as an illustration of this.

7.2 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.

7.3 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. Merge-sort will provide a classical illustration of this approach.

7.4 Estimation of cost via recurrence formulae

Consider particularly the case of divide and conquer. Suppose that for a problem of size n the division and combining steps involve $O(n)$ basic operations[6] 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

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

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

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 $\log(n)$.

7.5 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.

7.6 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. See the procedures noted later on for finding minimal spanning sub-trees as examples of how greed can lead to good results.

7.7 Back-tracking

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.

7.8 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.

7.9 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.

7.10 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.

7.11 The MM Method

The section 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 (research) students (no disrespect intended or implied) and wait a few months. It is quite likely they will come up with a solution any individual is unlikely to find. I have even seen a variant of this approach automated — by systematically trying ever increasing sequences of machine instructions until one it found that has the desired behaviour. It was applied to the follow C function:

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

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

8 The TABLE 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.

8.1 Operations that must be supported

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.

clear_table(): After this the contents of the table are considered undefined.

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.

8.2 Performance of a simple array

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.

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.

8.3 Sparse Tables — linked list representation

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 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.

8.4 Binary search in sorted array

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 $\Theta(\log(n))$.

8.5 Binary Trees

Another representation of a table that also provides $\log(n)$ costs is got by building a binary tree, where the tree structure relates very directly to the sequences 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 similarly well balanced, the greatest depth of the tree will be around $\log(n)$ [Proof?]. 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 later. 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.

8.6 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 made 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. 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.

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. Now what about some analysis of expected costs for tables that have a realistic load?

9 Free Storage Management

One of the options given above as a model for memory and basic data structures on a machine allowed for records, with some mechanism for allocating new instances of them. In the language ML such allocation happens without the user having to think about it; in C the library function `malloc` would probably be used, while C++, Java and the Modula family of languages will involve use of a keyword `new`.

If there is really no worry at all about the availability of memory then allocation is very easy — each request for a new record can just position it at the next available memory address. Challenges thus only arise when this is not feasible, i.e. when records have limited life-time and it is necessary to re-cycle the space consumed by ones that have become defunct.

Two issues have a big impact on the difficulty of storage management. The first is whether or not the system gets a clear direct indication when each previously-allocated record dies. The other is whether the records used are all the same size or are mixed. For one-sized records with known life-times it is easy to make a linked list of all the record-frames that are available for re-use, to add items to this "free-list" when records die and to take them from it again when new memory is needed. The next two sections discuss the allocation and re-cycling of mixed-size blocks, then there is a consideration of ways of discovering when data structures are not in use even in cases where no direct notification of data-structure death is available.

9.1 First Fit and Best Fit

Organise all allocation within a single array of fixed size. Parts of this array will be in use as records, others will be free. Assume for now that we can keep adequate track of this. The "first fit" method of storage allocation responds to a request for n units of memory by using part of the lowest block of at least n units that is marked free in the array. "Best Fit" takes space from the smallest free block with size at least n. After a period of use of either of these schemes the pool of memory can become fragmented, and it is easy to get in a state where there is plenty of unused space, but no single block is big enough to satisfy the current request.

Questions: How should the information about free space in the pool be kept? When a block of memory is released how expensive is the process of updating the free-store map? If adjacent blocks are freed how can their combined space be fully re-used? What are the costs of searching for the first or best fits? Are there patterns of use where first fit does better with respect to fragmentation than best fit, and vice versa? What pattern of request-sizes and requests would lead to the worst possible fragmentation for each scheme, and how bad is that?

9.2 Buddy Systems

The main message from a study of first and best fit is that fragmentation can be a real worry. Buddy systems address this by imposing constraints on both the sizes of blocks of memory that will be allocated and on the offsets within the array where various size blocks will be fitted. This will carry a space cost (rounding up the original request size to one of the approved sizes). A buddy system works by considering the initial pool of memory as a single big block. When a request comes for a small amount of memory and a block that is just the right size is not available then an existing bigger block is fractured in two. For the exponential buddy system that will be two equal sub-blocks, and everything works neatly in powers of 2. The pay-off arises when store is to be freed up. If some block has been split and later on both halves are freed then the block can be re-constituted. This is a relatively cheap way of consolidating free blocks.

Fibonacci buddy systems make the size of blocks members of the Fibonacci sequence. This gives less padding waste than the exponential version, but makes re-combining blocks slightly more tricky.

9.3 Mark and Sweep

The first-fit and buddy systems reveal that the major issue for storage allocation is not when records are created but when they are discarded. Those schemes processed each destruction as it happened. What if one waits until a large number of records can be processed at once? The resulting strategy is known as "garbage collection". Initial allocation proceeds in some simple way without taking any account of memory that has been released. Eventually the fixed size pool of memory used will all be used up. Garbage collection involves separating data that is still active from that which is not, and consolidating the free space into usable form.

The first idea here is to have a way of associating a mark bit with each unit of memory. By tracing through links in data structures it should be possible to identify and mark all records that are still in use. Then almost by definition the blocks of memory that are not marked are not in use, and can be re-cycled. A linear sweep can both identify these blocks and link them into a free-list (or whatever) and re-set the marks on active data ready for the next time. There are lots of practical issues to be faced in implementing this sort of thing!

Each garbage collection has a cost that is probably proportional to the heap size[7], and the time between successive garbage collections is proportional to the amount of space free in the heap. Thus for heaps that are very lightly used the long-term cost of garbage collection can be viewed as a constant-cost burden on each allocation of space, albeit with the realisation of that burden clumped together in big chunks. For almost-full heaps garbage collection can have very high overheads indeed, and a practical system should report a "store full" failure somewhat before memory is completely choked to avoid this.

[7]Without giving a more precise explanation of algorithms and data structures involved this has to be a rather woolly statement. There are also so called "generational" garbage collection methods that try to relate costs to the amount of data changed since the previous garbage collection, rather than to the size of the whole heap.

9.4 Stop and Copy

Mark and Sweep can still not prevent fragmentation. However imagine now that when garbage collection becomes necessary you can (for a short time) borrow a large block of extra memory. The "mark" stage of a simple garbage collector visits all live data. It is typically easy to alter that to copy live data into the new temporary block of memory. Now the main trick is that all pointers and cross references in data structures have to be updated to reflect the new location. But supposing that can be done, at the end of copying all live data has been relocated to a compact block at the start of the new memory space. The old space can now be handed back to the operating system to re-pay the memory-loan, and computing can resume in the new space. Important point: the cost of copying is related to the amount of live data copied, and not to to the size of the heap and the amount of dead data, so this method is especially suited to large heaps within which only a small proportion of the data is alive (a condition that also makes garbage collection infrequent) . Especially with virtual memory computer systems the "borrowing" of extra store may be easy — and good copying algorithms can arrange to make almost linear (good locality) reference to the new space.

9.5 Ephemeral Garbage Collection

This topic will be discussed briefly, but not covered in detail. It is observed with garbage collection schemes that the probability of storage blocks surviving is very skewed — data either dies young or lives (almost) for ever. Garbage collecting data that is almost certainly almost all still alive seems wasteful. Hence the idea of an "ephemeral" garbage collector that first allocates data in a short-term pool. Any structure that survives the first garbage collection migrates down a level to a pool that the garbage collector does not inspect so often, and so on. The bulk of stable data will migrate to a static region, while most garbage collection effort is expended on volatile data. A very occasional utterly full garbage collection might purge junk from even the most stable data, but would only be called for when (for instance) a copy of the software or data was to be prepared for distribution.

10 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.

10.1 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[8] 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. 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 $log(n!) = \Theta(n \log(n))$.

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

[8] 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.

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.

10.2 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.

10.3 Simple sorting

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 — 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.

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 $\lceil \log_2(k) \rceil$ comparisons[9]. 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 is has high (typically quadratic) data movement costs).

One final simple sort method is worth mentioning. Insertion sort is a perhaps a combination of the worst features of the above two schemes. 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.

[9]From now on I will not bother to specify what base my logarithms use — after all it only makes a constant-factor difference.

10.3.1 Insertion sort

The following illustrates insertion sort.

```
B|E W I L D E R M E N T    all to left of | are sorted
# #
B E|W I L D E R M E N T
  # #
B E W|I L D E R M E N T
  # * *
B E I W|L D E R M E N T
    # * *                  # = read, * = read-write
B E I L W|D E R M E N T
# * * * * *
B D E I L W|E R M E N T
    # * * * *
B D E E I L W|R M E N T
          # * *
B D E E I L R W|M E N T
          # * * *
B D E E I L M R W|E N T
      # * * * * * *
B D E E E I L M R W|N T
              # * * *
B D E E E I L M N R W|T
                  # * *
B D E E E I L M N R T W|   everything now sorted
```

10.4 Shell's Sort

Shell's 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 by something that will get items to roughly the correct position, in the hope that the insertion sort will then have linear cost. The way that Shellsort does this is to do 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 $1, s+1, 2s+1, 3s+1, \ldots$, the next will use positions $2, s+2, 2s+2, 3s+2, \ldots$, 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's 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's 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.

The following attempts to illustrate Shell's sort.

```
1   2   3   4   5   6   7   8   9   10  11  12
B   E   W   I   L   D   E   R   M   E   N   T
#               #                                step of 4
B   E   W   I   L   D   E   R   M   E   N   T
    *               *
B   D   W   I   L   E   E   R   M   E   N   T
        *               *
B   D   E   I   L   E   W   R   M   E   N   T
            #               #
B   D   E   I   L   E   N   R   M   E   W   T
                #               *                ( # = read, * = read+write)
B   D   E   I   L   E   N   R   M   E   W   T
                    #               *
B   D   E   I   L   E   N   R   M   E   W   T
                        #               #
B   D   E   I   L   E   N   R   M   E   W   T
                            #               #
B   D   E   I   L   E   N   R   M   E   W   T
#   #                                            step of 1
B   D   E   I   L   E   N   R   M   E   W   T
    #   #
B   D   E   I   L   E   N   R   M   E   W   T
        #   #
B   D   E   I   L   E   N   R   M   E   W   T
            #   #
B   D   E   I   L   E   N   R   M   E   W   T
        #   *   *   *
B   D   E   E   I   L   N   R   M   E   W   T
                    #   #
B   D   E   E   I   L   N   R   M   E   W   T
                        #   #
B   D   E   E   I   L   N   R   M   E   W   T
                    #   *   *   *
B   D   E   E   I   L   M   N   R   E   W   T
            #   *   *   *   *   *   *
B   D   E   E   E   I   L   M   N   R   W   T
                                    #   #
B   D   E   E   E   I   L   M   N   R   W   T
                                    #   *   *
B   D   E   E   E   I   L   M   N   R   T   W    final result
```

10.5 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 turns in a 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 etc.

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 $\Theta(n \log(n))$.

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 unsatisfatory (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 average good speed of a slightly slower but guaranteed $n \log(n)$ method can be a delicate one.

There are good reasons for using the median of of the mid point and two others as the median at each stage, and using recursion only on the smaller partition.

When the region is smaller enough insertin sort should be used.

```
 1  2  3  4  5  6  7  8  9 10 11 12
|B  E  W  I  L  D  E  R  M  E  N  T|
|#              #                 #| => median D
|#  *           *  #  #  #  #  #  #|
|B  D  W  I  L  E  E  R  M  E  N  T|
|   # |#  #  #                     | partition point
|B| D |W  I  L  E  E  R  M  E  N  T| do smaller side first
|-|   |                            | insertion sort
 B    |W  I  L  E  E  R  M  E  N  T|
      |#           #              #| => median T
      |*                          *|
      |T  I  L  E  E  R  M  E  N  W|
      |*                       *   |
      |N  I  L  E  E  R  M  E  T  W|
      |   #  #  #  #  #  #  #| #   | partition point
      |N  I  L  E  E  R  M  E| T |W| do smaller side first
      |                      |   |-| insertion sort
      |N  I  L  E  E  R  M  E|    W
      |#        #           #|       => median E
      |*                    *|
      |E  I  L  E  E  R  M  N|
      |*           *  #  #   |
      |E  I  L  E  E  R  M  N|
      |   *        *         |
      |E  E  L  E  I  R  M  N|
      |   *     *            |
      |E  E  L  E  I  R  M  N|
      |      *  *            |
      |E  E  E  L  I  R  M  N|
      |    |   |             |       partition point
      |E  E| E |L  I  R  M  N|
      |----|   |             |       insertion sort
       E  E    |L  I  R  M  N|
               -------------         insertion sort
                I  L  M  N  R

 B  D  E  E  E  I  L  M  N  R  T  W  final result
```

10.5.1 Possible improvements

We could consider using an $O(n)$ algorithm to find the true median since its use would guarantee Quicksort being $O(n \log(n))$.

When the elements being sorted have many duplicates (as in surnames in a telephone directory), it may be sensible to partition into three sets: elements less than the median, elements equal to the median, and elements greater than the median. Probably the best known way to do this is based on the following counter intuitive invariant:

```
 | elements=m | elements<m | ...... | elements>m | elements=m |
               ^            ^      ^            ^
               |            |      |            |
               a            b      c            d
```

Quoting Bentley and Sedgewick, the main partitioning loop has two inner loops. The first inner loop moves up the index `b`: it scans over lesser elements, swaps equal element with `a`, and halts on a greater element. The second inner loop moves down the index `c` correspondingly: it scans over greater elements, swaps equal elements with `d`, and halts on a lesser element. The main loop then swaps the elements pointed to by `b` and `c`, incrementing b and decrementing `c`, and continues until `b` and `c` cross. Afterwards the equal elements on the edges are swapped to the middle, without any extraneous comparisons.

10.6 Heap Sort

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 modest 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 $2k+1$.[10] 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.

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 $n \log(n)$.

Further details of both parts of heapsort can be found in the textbooks and will be given in lectures.

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

```
1  2 3  4 5 6 7  8 9 10 11 12
B  E W  I L D E  R M E N T  start heapify
            *            *
B  E W  I L T E  R M E N D
          *          # *   (#=read *=read+write)
B  E W  I N T E  R M E L D
        *        * #
B  E W  R N T E  I M E L D
     #      # #
B  E W  R N T E  I M E L D
   *    * #      # *
B  R T  M N T E  I E E L D
*  # *      * #          *
W  R T  M N D E  I E E L B  heapify done
*                        *
B  R T  M N D E  I E E L| W
*  # *      # *         |
T  R E  M N D B  I E E L| W
*                      *
L  R E  M N D B  I E E| T W
*  * #  # *          #|
R  N E  M L D B  I E E| T W
*                    *
E  N E  M L D B  I E| R T W
*  * #  * #      * #|
N  M E  I L D B  E E| R T W
*                  *
E  M E  I L D B  E| N R T W
*  * #  # *       |
M  L E  I E D B  E| N R T W
*                *
E  L E  I E D B| M N R T W
*  * #  * #    |
L  I E  E E D B| M N R T W
*             *
B  I E  E E D| L  M N R T W
*  * #  # *  |
I  E E  E B D| L  M N R T W
*           *
D  E E  E B| I L  M N R T W
*  # *     |
E  E D  E B| I L  M N R T W
*         *
B  E D  E| E I L  M N R T W
*  * #  *|
E  E D  B| E I L  M N R T W
*       *
B  E D| E E I L  M N R T W
*  * #|
E  B D| E E I L  M N R T W
*    *
D  B| E  E E I L  M N R T W
#  #|
D  B| E  E E I L  M N R T W
*  *
B| D E  E E I L  M N R T W  all done
```

10.7 Binary Merge in memory

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[11]. 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.

The following illustrated the basic merge sort mechanism.

```
|B|E|W|I|L|D|E|R|M|E|N|T|
|* *| |* *| |* *| |* *| |
|B E|W|I L|D|E R|M|E N|T|
|*** *|*** *|*** *|*** *|
|B E W|D I L|E R M|E N T|
|***** *****|***** *****|
|B D E I L W|E E M N R T|
|*********** ***********|
|B D E E E I L M N R T W|
```

In practice, having some $n/2$ words of workspace makes the programming easier. Such an implementation is illustrated below.

[11] There is scope for a lengthy discussion of the amount of stack needed by Quicksort here.

```
B E W I L D E R M E N T
*****                                  insertion sort
B E W I L D E R M E N T
      *****                            insertion sort
B E W D I L E R M E N T
            *****                      insertion sort
B E W D I L E M R E N T
                  *****                insertion sort
B E W D I L E M R E N T
            #     #       *            merge EMR
              #   #         *           with ENT
              #     #         *
                #   #           *      (#=read *=write)
                #     #           *
                      #             *
B E W D I L               E E M N R T
B E W D I L               E E M N R T
#     #     *                          merge BEW
  #   #       *                         with DIL
  #     #       *
    #   #         *
    #     #         *
    #                 *
            B D E I L W   E E M N R T
            B D E I L W   E E M N R T
*           #             #            merge BDEILW
  *           #           #             with EEMNRT
    *           #         #
      *           #       #
        *         #         #
          *       #           #
            *       #         #
              *       #       #
                *     #         #
                  *   #           #
                    * #             #
                      *
B D E E E I L M N R T W                sorted result
```

10.8 Radix sorting

To radix-sort from the most significant end, look at the most significant digit in the sort key, and distribute that data based on just that. Recurse to sort each clump, and the concatenation of the sorted sub-lists is fully sorted array. One might see this as a bit like Quicksort but distributing n ways instead of just into two at the selection step, and preselecting pivot values to use.

To sort from the bottom end, first sort your data taking into account just the last digit of the key. As will be seen later this can be done in linear time using a distribution sort. Now use a stable sort method to sort on the next digit of the key up, and so on until all digits of the key have been handled. This method was popular with punched cards, but is less widely used today!

10.9 Memory-time maps

Figure 1 shows the behaviour of three sorting algorithms.

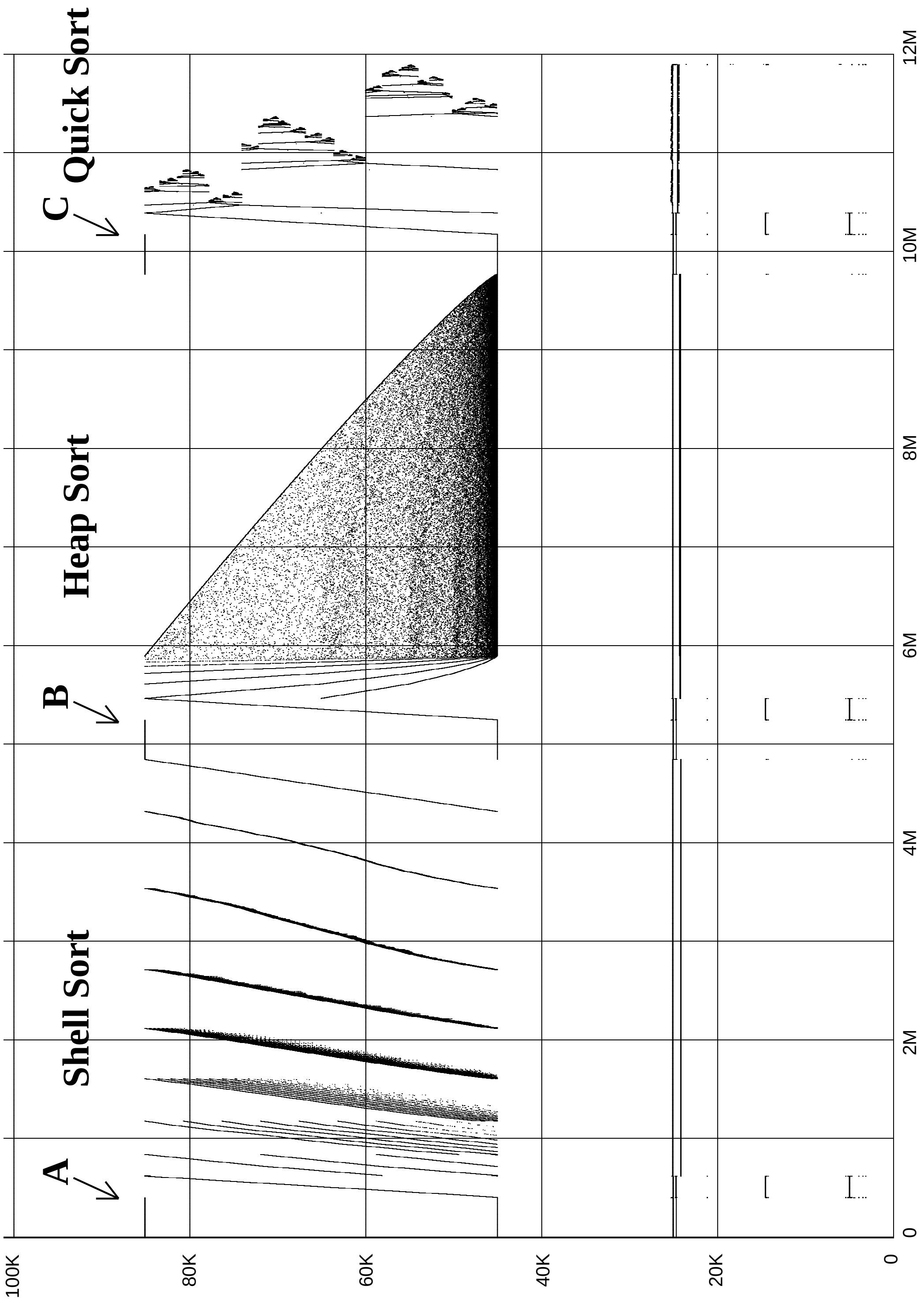


Figure 1: Memory-time maps for three sorting algorithms

10.10 Instruction execution counts

Figure 2 gives the number of instructions executed to sort a vector of 5000 integers using six 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 the range 0..9999999.

Random 0..999: The data consists of random integers in the range 0..999.

Method	Ordered	Reversed	Random	Ramdom 0..999
insertion	99,995	187,522,503	148,323,321	127,847,226
shell	772,014	1,051,779	1,739,949	1,612,419
quick	401,338	428,940	703,979	694,212
heap	2,093,936	1,877,564	1,985,300	1,973,898
tree	125,180,048	137,677,548	997,619	5,226,399
merge	732,472	1,098,209	1,162,833	1,158,362

Figure 2: Instruction execution counts for various sorting algorithms

10.11 Order statistics (eg. median finding)

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. 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.

10.12 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 first is 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. 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 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, and this prediction is used to insert items in it, then apart from some local effects of clustering that data has been sorted.

10.13 Parallel processing sorting networks

This is another topic that will just be mentioned here, but which gets full coverage in some of the textbooks. Suppose you want to sort data using hardware rather than software (this could be relevant in building some high performance graphics engine, and it could also be relevant in routing devices for some networks). Suppose further that the values to be sorted appear on a bundle of wires, and that a primitive element available to you has two such wires as inputs and transfers its two inputs to output wires either directly or swapped, depending on their relative values. How many of these elements are needed to sort that data on n wires? How should they be connected? How many of the elements does each signal flow through, and thus how much delay is involved in the sorting process?

11 Storage on external media

For the next few sections the cost model used for memory access is adjusted to take account of reality. It will be assumed that we still have a reasonable sized conventional main memory on our computer and that accesses to that have unit cost. But it will be supposed that the bulk of the data to be handled does not fit into main memory and so resides on tape or disc, and that it is necessary to pay attention to the access costs that this implies.

11.1 Cost assumptions for tapes and discs

When Knuth's series of books were written magnetic tapes formed the mainstay of large-scale computer storage. Since then discs have become larger, cheaper and more reliable, and tape-like devices are really only used for archival storage. Thus the discussions here will ignore the large and entertaining but archaic body of knowledge about how best to sort data using two, three or four tape drives that can or can not read and write data backwards as well as forwards.

The main assumption to be made about external storage will be that it is slow — so slow that using it well becomes almost the only important issue for an algorithm. The next characteristic will be that sequential access and reading/writing fairly large blocks of data at once will be the best way to maximise data transfer. Seeking from one place on a disc to another will be deemed expensive.

There will probably be an underlying expectation in this discussion that the amount of data to be handled is roughly between 10 Mbytes and 10 Gbytes. Much less data than that does not justify thinking about external processing, while much larger amounts may raise additional problems (and may be infeasible, at least this year)

11.2 B-trees

With data structures kept on disc it is sensible to make the unit of data fairly large — perhaps some size related to the natural unit that your disc uses (a sector or track size). Minimising 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. The idea starts by generalising 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 alternate pointers to sub-trees and key values. This will tend to define the maximum branching factor that can be supported in terms of the natural disc block size and the amount of memory needed for each key. When new items are added to a B-tree it will often be possible to add the item within an existing block without overflow. Any block that becomes full can be split into two, and the single reference to it from its parent block expands to the two references to the new half-empty blocks. For B-trees of reasonable branching factor any reasonable amount of data can be kept in a quite shallow tree — although the theoretical cost of access grows with the logarithm of the number of data items stored in practical terms it is constant.

The algorithms for adding new data into a B-tree arrange that the tree is guaranteed to remain balanced (unlike the situation with the simplest sorts of trees), and 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 generalisation of those used for 2-3-4-trees (that are discussed later in these notes) but note that the implementation details may be significantly different, firstly because the B-tree will have such a large branching factor and secondly all operations will need to be performed with a view to the fact that the most costly step is reading a disc block (2-3-4-trees are used as in-memory data structures so you could memory program steps rather than disc accesses when evaluating and optimising an implemantation).

11.3 Dynamic Hashing (Larsen)

This is a really neat way in which quite modest in-store index information can make it possible to retrieve any item in just one disc access. Start by viewing all available disc blocks as buckets in a hash table. Take the key to be located, and compute a hash function of it — in an ideal world this could be used to indicate which disc block should be read. Of course several items can probably be stored in each disc block, so a certain number of hash clashes will not matter at all. Provided no disc block ever becomes full this satisfies our goal of single disc-transfer access.

Further ingenuity is needed to cope with full disc blocks while still avoiding extra disc accesses. The idea applied is to use a small in-store table that will indicate if the data is in fact stored in the disc block first indicated. To achieve this instead of computing just one hash function on the key it is necessary to compute two. The second one is referred to as a signature. For each disc block we record in-store the value of the largest signature of any item that is allowed in that block. Now a comparison of our signature with the value stored in this table allows us to tell (without going to the disc) if the required data is present on its first choice disc block.

If not then we go back to the key and use a second choice pair of hash functions to produce a new potential location and signature, and again our in-store table indicates if the data could be stored there. By having a sequence of hash functions that will eventually propose every possible disc block this sort of searching should eventually terminate. Note that if the data involved is not on the disc at all we find that out when we read the disc block that it would be on if it were present. Unless the disc is almost full it will probably only take a few hash calculations and in-store checks to locate the data, and remember that a very great deal of in-store calculation can be justified to save even one disc access.

As has been seen, recovering data stored this way is quite easy. What about adding new records? Well, one can start by following through the steps that locate and read the disc block that the new data would seem to live on. If the data is already stored there it can be updated. If it is not there but the disc block has free space then the new record can be added (and that may require that the main signature table be updated if the new data has a larger signature than any one previously used on that block). Otherwise the block overflows. Any items in it with largest signature (which may be the new record, or may be ones that were there already) are moved to an in-store buffer, the signature table entry is reduced (to one less than this largest signature) and the block is written back to disc. That leaves some records (often just one) to be re-inserted elsewhere in the table, and of course the signature table shows that it can not live in the block that has just been inspected. The insertion process continues by looking to see where the next available choice for storing that record would be.

Once again for lightly loaded discs insertion is not liable to be especially expensive, but as the system gets close to being full a single insertion could cause major rearrangements. Note however that most large databases have very many more instances of read or update-in-place operations than of ones that add new items. CD-ROM technology provides a case where reducing the number of (slow) read operations can be vital, but where the cost of creating the initial data structures that go on the disc is almost irrelevant.

11.3.1 A tiny example

To illustrate Larsen's method we will imagine a disc with just 5 blocks each capable of holding up to 4 records. We will assume there are 26 keys, `A` to `Z`, each with a sequence of probe/signature pairs. The probes are in the range `0..4`, and the signatures are in the range `0..7`. In the following table, only the first two probe/signature pairs are given (the remaining ones are not needed here).

```
A(1/6)(2/4) B(2/4)(0/5) C(4/5)(3/6) D(1/7)(2/1) E(2/0)(4/2)
F(2/1)(3/3) G(0/2)(4/4) H(2/2)(1/5) I(2/4)(3/6) J(3/5)(4/1)
K(3/6)(4/2) L(4/0)(1/3) M(4/1)(2/4) N(4/2)(3/5) O(4/3)(3/6)
P(4/4)(1/1) Q(3/5)(2/2) R(0/6)(3/3) S(3/7)(4/4) T(0/1)(1/5)
U(0/2)(2/6) V(2/3)(3/1) W(1/4)(4/2) X(2/5)(1/3) Y(1/6)(2/4)
Z(1/0)(3/5)
```

After adding `A(1/6)`, `B(2/4)`, `C(4/5)`, `D(1/7)`, `E(2/0)`, `F(2/1)`, `G(0/2)` and `H(2/2)`, the blocks are as follows:

```
G...  DA..  BHFE  ....  C...     keys
2     76    4210        5        signatures
7     7     7     7     7        in-memory table
```

The keys in each block are shown in decreasing signature order. Continuing the process, we find the next key (`I(2/4)`) should be placed in block 2 which is full, and (worse) `I`'s current signature of 4 clashes with the largest signature in the block (that of `B`), so both `I` and `B` must find homes in other blocks. They each use their next probes/signature pairs, `B(0/5)` and `I(3/6)`, giving the following result:

```
BG..  DA..  HFE.  I...  C...     keys
52    76    210.  6     5        signatures
7     7     3     7     7        in-memory table
```

The in-memory entry for block 2 disallows that block from ever holding a key whose signature is greater than 3.

11.4 External Sorting

There are three major observations here. The first is that it will make very good sense to do as much sorting as possible internally (using your favourite in-store method), so a major part of any external sorting method is liable to be breaking the data up into store-sized chunks and sorting each of those. The second point is that variants on merge-sort fit in very well with the sequential access patterns that work well with disc drives. The final point is that with truly large amounts of data it will almost certainly be the case that the raw data has well known statistical properties (including the possibility that it is known that it is almost in order already, being just a modification of previous data that had itself been sorted earlier), and these should be exploited fully.

12 Variants on the SET 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 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.

12.1 Operations that must be supported

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. In cases where some of the operations (for instance **maximum** and **minimum**) are not used these conditions might be relaxed.

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

choose_any(S): if S is non-empty this should return an arbitrary item from S.

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, x): x is an item present in the set S. Change S to remove x from it.

minimum(S): return the item from S that has the smallest key.

maximum(S): return the item from S that has the largest key.

successor(S, x): x is in S. Find the item in S that has the next larger key than the key of x. If x was the largest item in the heap indicate that fact.

predecessor(S, x): as for successor, but finds the next smaller key.

union(S, S'): combine the two sets S and S' to form a single set combining all their elements. The original S and S' may be destroyed by this operation.

12.2 Tree Balancing

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.

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.

12.3 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.

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 "black" nodes stand for the regular 2-3-4 nodes. 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. Insertions and node splitting in red-black trees just has to follow the rules that were set up for 2-3-4 trees.

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.

12.4 Priority Queues and Heaps

If we concentrate on the operations **insert**, **minimum** 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.

A good representation for a priority queue is a heap (as in Heapsort), where the minimum item is instantly available and the other operations can be performed in logarithmic time. Insertion and deletion of heap items is straightforward. They are well described in the textbooks.

12.5 More elaborate representations

So called "Binomial Heaps" and "Fibonacci Heaps" have as their main characteristic that they provide efficient support for the **union** operation. If this is not needed then ordinary heaps should probably be used instead. Attaining the best available computing times for various other algorithms may rely on the performance of datastructures as elaborate as these, so 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 these methods up in a textbook and try to produce a good implementation!

12.6 Ternary search trees

This section is based on a paper by Jon Bentley and Robert Sedgewick (`http://www.cs.princeton.edu/~rs/strings`, also on Thor).

Ternary search trees have be around since the early 1960s but their practical utility when the keys are strings has be largely overlooked. Each node in the tree contains a character, *ch* and three pointers L, N and R. To lookup a string, its first character is compared with that at the root and the L, N or R pointer followed, depending on whether the string character was, respectively, less, equal or greater than the character in the root. If the N pointer was taken the position in the string is advanced one place. Strings are terminated by a special character (a zero byte). As an example, the following ternary tree contains the words MIT, SAD, MAN, APT, MUD, ADD, MAG, MINE, MIKE, MINT, AT, MATE, MINES added in that order.

```
                                            |
       -------------------------------------M---------
      |                                     |         |
     -A-              ----------------------I----    -S-
      |              |                      |    |    |
  ----P----         -A-              -------T-  -U-  -A-
 |    |    |         |              |       |    |    |
-D-  -T-  -T-    ----N----      ----N-     -*-  -D-  -D-
 |    |    |    |    |    |    |    |            |    |
-D-  -*-  -*-  -G-  -*-  -T-  -K-  -E-          -*-  -*-
 |              |         |    |    |
-*-            -*-       -E-  -E-  -*---
                          |    |        |
                         -*-  -*-      -S-
                                        |
                                       -*-
```

Ternary trees are easy to implement and are typically more efficient in terms of character operations (comparison and access) than simple binary trees and hash tables, particularly when the contained strings are long. The predecessor (or successor) of a string can be found in logarithmic-time. Generating the strings in sorted order can be done in linear time. Ternary trees are good for “partial matches” (sometimes crossword puzzle lookup) where “don't-care” characters can be in the string being looked up). This makes the structure applicable to Optical Character Recognition where some characters of a word may have been recognised by the optical system with a high degree of confidence but others have not.

The N pointer of a node containing the end-of-string character can be used to hold the value associated the string that ends on that node.

Ternary trees typically take much more store than has tables, but this can be reduced replacing subtrees in the structure that contain only one string by a pointer to that string. This can be done with an overhead of three bits per node (to indicate which of the three pointers point to such strings).

12.7 Splay trees

The material in this section is based on notes from Danny Sleator of CMU and Roger Whitney of San Diego State University.

A splay tree is a simple binary tree repesentation of a table (providing **insert**, **search**, **update** and **delete** operations) combined with a move-to-root strategy that gives it the following remarkably attractive properties.

1. The amortized 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 the first two is beyond the scope of this course, but the third is “obvious”!

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. If k is the key in a node, the key in its left child will be less than k and the key in the right child will be greater that k. No other node in the entire tree will have key k.

Whenever an **insert**, **lookup** or **update** operation is performed, the accessed node (with key `X`, say) is promoted to the root of the tree using the following sequence of transformations.

```
      Z           X               W                 X
     / \         / \             / \               / \
    Y   d  ==>  a   Y           a   V     ==>     V   d
   / \             / \             / \           / \
  X   c           b   Z           b   X         W   c
 / \                 / \             / \       / \
a   b               c   d           c   d     a   b

      Z            X               W               X
     / \         /   \            / \            /   \
    W   d  ==>  W     Z          a   Y   ==>    W     Y
   / \         / \   / \            / \        / \   / \
  a   X       a   b c   d          X   d      a   b c   d
     / \                          / \
    b   c                        b   c

      Y           X                V             X
     / \         / \              / \           / \
    X   c  ==>  a   Y            a   X   ==>   V   c
   / \             / \              / \       / \
  a   b           b   c            b   c     a   b
```

The last two transformations are used only when `X` is within one position of the root. Here is an example where `A` is being promoted.

```
              G          G            G           A
             / \        / \          / \        /   \
            F   8      F   8        F   8      1     F
           / \        / \          / \             /   \
          E   7      E   7        A   7           D      G
         / \        / \          / \            /   \   / \
        D   6 ===> D   6  ===>  1   D   ===>   B     E 7   8
       / \        / \             /   \       / \   / \
      C   5      A   5           B     E     4   C 5   6
     / \        / \             / \   / \       / \
    B   4      1   B           2   C 5   6     3   4
   / \            / \             / \
  A   3          2   C           3   4
 / \                / \
1  2               3   4
```

Notice that promoting `A` to the root improves the general bushiness of the tree. Had the (reasonable looking) transformations of the form been used:

```
      Z           X               W                 X
     / \         / \             / \               / \
    Y   d  ==>  a   Z           a   V     ==>     W   d
   / \             / \             / \           / \
  X   c           Y   d           b   X         a   V
 / \             / \                 / \           / \
a   b           c   c               c   d         b   c
```

then the resulting tree would not be so bushy (try it) and the nice statistical properties of splay tree would be ruined.

13 Pseudo-random numbers

This is a topic where the obvious best reference is Knuth (volume 1). If you look there you will find an extended discussion of the philosophical problem of having a sequence that is in fact totally deterministic but that you treat as if it was unpredictable and random. You will also find perhaps the most important point of all stressed: a good pseudo-random number generator is not just some complicated piece of code that generates a sequence of values that you can not predict anything about. On the contrary, it is probably a rather simple piece of code where it is possible to predict a very great deal about the statistical properties of the sequence of numbers that it returns.

13.1 Generation of sequences

In many cases the programming language that you use will come with a standard library function that generates "random" numbers. In the past (sometimes even the recent past) various such widely distributed generators have been very poor. Experts and specialists have known this, but ordinary users have not. If you can use a random number source provided by a well respected purveyor of high quality numerical or system functions then you should probably use that rather than attempting to manufacture your own. But even so it is desirable that computer scientists should understand how good random number generators can be made.

A very simple class of generators defines a sequence a_i by the rule $a_{i+1} = (Aa_i + B) \text{mod} C$ where A, B and C are very carefully selected integer constants. From an implementation point of view, many people would really like to have $C = 2^{32}$ and thereby use some artefact of their computer's arithmetic to perform the mod$C$ operation. Achieving the same calculation efficiently but not relying on low-level machine trickery is not especially easy. The selection of the multiplier A is critical for the success of one of these congruential generators — and a proper discussion of suitable values belongs either in a long section in a textbook or in a numerical analysis course. Note that the entire state of a typical linear congruential generator is captured in the current seed value, which for efficient implementation is liable to be 32 bits long. A few years ago this would have been felt a big enough seed for most reasonable uses. With today's faster computers it is perhaps marginal. Beware also that with linear congruential generators the high order bits of the numbers computed are much more "random" than the low order ones (typically the lowest bit will just alternate $0, 1, 0, 1, 0, 1, \ldots$).

There are various ways that have been proposed for combining the output from several congruential generators to produce random sequences that are better than any of the individual generators alone. These too are not the sort of thing for amateurs to try to invent!

A simple-to-program method that is very fast and appears to have a reputation for passing the most important statistical tests involves a recurrence of the form

$$a_k = a_{k-b} + a_{k-c}$$

for offsets b and c. The arithmetic on the right hand side needs to be done modulo some even number (again perhaps 2^{32}?). The values $b = 31, c = 55$ are known to work well, and give a very long period[12]. There are two potential worries about this sort of method. The first is that the state of the generator is a full c words, so setting or resetting the sequence involves touching all that data. Secondly although additive congruential generators have been extensively tested and appear to behave well, many details of their behaviour are not understood — our theoretical understanding of the multiplicative methods and their limitations is much better.

13.2 Probabilistic algorithms

This section is provided to get across the point that random numbers may form an essential part of some algorithms. This can at first seem in contradiction to the description of an algorithm as a systematic procedure with fully analysed behaviour. The worry is resolved by accepting a proper statistical analysis of behaviour as valid.

We have already seen one example of a random numbers in an algorithms (although at the time it was not stressed) where it was suggested that Quicksort could select a (pseudo-) random item for use as a pivot. That made the cost of the whole sorting process insensitive to the input data, and average case cost analysis just had to average over the explicit randomness fed into pivot selection. Of course that still does not correct Quicksort's bad worst-case cost — it just makes the worst case depend on the luck of the (pseudo-) random numbers rather than on the input data.

Probably the best known example of an algorithm which uses randomness in a more essential way is the Miller-Rabin test to see if a number is prime. This test is easy to code (except that to make it meaningful you would need to set it up to work with multiple-precision arithmetic, since testing ordinary machine-precision integers to see if they are prime is too easy a task to give to this method). Its justification relies upon more mathematics than I want to include in this course. But the overview is that it works by selecting a sequence of random numbers. Each of these is used in turn to test the target number — if any of these tests indicate that the target is **not** prime then this is certainly true. But the test used is such that if the input number was in fact composite then each independent

[12] provided at least one of the initial values is odd the least significant bits of the a_k form a bit-sequence that has a cycle of length about 2^{55}.

random test had a chance of at least 1/2 of detecting that. So after s tests that all fail to detect any factors there is only a 2^{-s} chance that we are in error if we report the number to be prime. One might then select a value of s such that the chances of undetected hardware failure exceed the chances of the understood randomness causing trouble!

14 Data Compression

File storage on distribution discs and archive tapes generally uses compression to fit more data on limited size media. Picture data (for instance Kodak's Photo CD scheme) can benefit greatly from being compressed. Data traffic over links (eg. fax transmissions over the phone lines and various computer-to-computer protocols) can be effectively speeded up if the data is sent in compressed form. This course will give a sketch of some of the most basic and generally useful approaches to compression. Note that compression will only be possible if the raw data involved is in some way redundant, and the foundation of good compression is an understanding of the statistical properties of the data you are working with.

14.1 Huffman

In an ordinary document stored on a computer every character in a piece of text is represented by an eight-bit byte. This is wasteful because some characters (' ' and 'e', for instance) occur much more frequently than others ('z' and '#' for instance). Huffman coding arranges that commonly used symbols are encoded into short bit sequences, and of course this means that less common symbols have to be assigned long sequences.

The compression should be thought of as operating on abstract symbols, with letters of the alphabet just a particular example. Suppose that the relative frequencies of all symbols are known in advance[13], then one tabulates them all. The two least common symbols are identified, and merged to form a little two-leaf tree. This tree is left in the table and given as its frequency the sum of the frequencies of the two symbols that it replaces. Again the two table entries with smallest frequencies are identified and combined. At the end the whole table will have been reduced to a single entry, which is now a tree with the original symbols as its leaves. Uncommon symbols will appear deep in the tree, common ones higher up. Huffman coding uses this tree to code and decode texts. To encode a symbol a string of bits is generated corresponding to the combination of left/right selections that must be made in the tree to reach that symbol. Decoding is just the converse — received bits are used as navigation information in the tree, and

[13] this may either be because the input text is from some source whose characteristics are well known, or because a pre-pass over the data has collected frequency information.

when a leaf is reached that symbol is emitted and processing starts again at the top of the tree.

A full discussion of this should involve commentary on just how the code table set up process should be implemented, how the encoding might be changed dynamically as a message with varying characteristics is sent, and analysis and proofs of how good the compression can be expected to be.

The following illustrates the construction of a Huffman encoding tree for the message: `PAGFBKKALEAAGTJGGGQ`. There are 5 `G`s, 4 `A`s, 2 `K`s etc.

```
============leaves=============  ==============nodes==============
P  F  B  L  E  T  J  Q  K  A  G (symbols)
1  1  1  1  1  1  1  1  2  4  5 (frequencies)
#  #  #  #  #  #  #  #  #  #  #
|  |  |  |  |  |  |  |  |  |  |
 --+----------------------------(2)LL               node types
4  4  |  |  |  |  |  |  |  |  | 3#
       --+-------------------------(2)LL          LL = leaf+leaf
      4  4  |  |  |  |  |  |  |  | 3#             LN = leaf+node
             --+----------------------(2)LL       NN = node+node
            4  4  |  |  |  |  |  |  | 3#
                   --+-------------------(2)LL
                  4  4  |  |  |  |  |  | 3#
                         --------+----------(4)LN
                        3  |  |  3  |  |  | 2#
                           |  |      --+-------(4)NN
                           |  |     3  3  |  | 2#
                            --------------+-------(6)LN
                           3  |           3  |  | 2#
                              |               --+----(8)NN
                              |              2  2  | 1#
                               --------------------+----(11)LN
                              2                    2  | 1#
                                                       --+-(19)NN
4  4  4  4  4  4  4  4  3  3  2                       1  1 0
     Huffman code lengths
```

The sum of the node frequencies $(2+2+2+2+4+4+6+8+11+19 = 60)$ is the total length of the encoded message $(4+4+4+4+4+4+4+4+2*3+4*3+5*2 = 60)$.

We can allocate Huffman codes as follows:

Symbol	Frequencies	code length	Huffman codes		
0	0	-			
.	.	.			
.	.	.			
A	4	3		101	
B	1	4	0111		
C	0	-			
D	0	-			
E	1	4	0110		
F	1	4	0101		
G	5	2			11
H	0	-			
I	0	-			
J	1	4	0100		
K	2	3		100	
L	1	4	0011		
M	0	-			
N	0	-			
O	0	-			
P	1	4	0010		
Q	1	4	0001		
R	0	-			
S	0	-			
T	1	4	0000		
.	.	.			
.	.	.			
255	0	-			
Smallest code for each length:			0000	100	11
Number of codes for each length:			8	2	1

All the information in the Huffman code table is contained in the code length column which can be encoded compactly using run length encoding and other tricks. The compacted representation of a typical Huffman table for bytes used in a large text file is usually less than 40 bytes. First initialise `s=m=0`, then read nibbles until all symbols have lengths.

```
nibble
 0000  -4       m-=4;      if(m<1)  m+=23;   )
 0001  -3       m-=3;      if(m<1)  m+=23;   )
 0010  -2       m-=2;      if(m<1)  m+=23;   )
 0011  -1       m-=1;      if(m<1)  m+=23;   )
                                             )  len[s++] = m;
 0100  M  bbbb  m+=bbbb+4; if(m>23) m-=23;   )
 0101  +1       m+=1;      if(m>23) m-=23;   )
 0110  +2       m+=2;      if(m>23) m-=23;   )
 0111  +3       m+=3;      if(m>23) m-=23;   )

 1000  R1      ) use a sequence of these to encode
 1001  R2      ) an integer n = 1,2,3,... then
 1010  R3      )
 1011  R4      )        for(i=1; i<=n; i++) len[s++] = m;

 1100  Z1      ) use a sequence of these to encode
 1101  Z2      ) an integer n = 1,2,3,... then
 1110  Z3      )
 1111  Z4      )        for(i=1; i<=n; i++) len[s++] = 0;
```

Using this scheme, the Huffman table can be encoded as follows:

```
zero to  65             Z1   Z4  Z3       (+65)
                         1+4*(4+4*3)      = 65
         65 len[A]=3    +3
         66 len[B]=4    +1
zero to  69             Z2
         69 len[E]=4  )
         70 len[F]=4  ) R2
         71 len[G]=2    -2
zero to  74             Z2
         74 len[J]=4    +2
         75 len[K]=3    -1
         76 len[L]=4    +1
zero to  80             Z3
         80 len[P]=4  )
         81 len[Q]=4  ) R2
zero to  84             Z2
         84 len[T]=4    R1
zero to 256             Z4   Z2   Z2  Z2   (+172)
                         4+4*(2+4*(2+4*2)) = 172
stop
```

The table is thus encoded in a total of 20 nibbles (10 bytes). Finally, the message: `PAGFB...` is encoded as follows:

```
P    A   G  F    B    ...
0010 101 11 0101 0111 ...
```

Since the encoding of the Huffman table is so compact, it makes sense to transmit a new table quite frequently (every 50000 symbols, say) to make the encoding scheme adaptive.

14.2 Run-length encoding

If the data to be compressed has runs of repeated symbols, then run length encoding may help. This method represents a run by the repeated symbol combined with its repetition count. For instance, the sequences `AAABBBBXAABBBBYZ...` could be encoded as `3A4BX2A4BYZ...` This scheme may or not use additional symbols to encode the counts. To avoid additional symbols, the repeat counts could be placed after the first two characters of a run. The above string could be encoded as `AA1BB2XAA0BB2YZ...` Many further tricks are possible.

The encoding of the raster line represention of a fax uses run length encoding, but since the are only two symbols (black and white) only the repeat counts need be sent. These are then Huffman coded using separate tables for black and the white run lengths. Again there are many additional variants to the encoding of faxes.

14.3 Move-to-font buffering

Huffman coding, in its basic form, does not adapt to take account of local changes in symbol frequencies. Sometimes the occurrence of a symbol increases the probability that it will occur again in the near future. Move-to-front buffering sometimes helps in this situation. The mechanism is as follows. Allocate a vector (the buffer) and place in it all the symbols needed. For 8-bit ASCII text, a 256 byte vector would be used and initialiased with value 0..255. Whenever a symbol is processed, it is replaced by the integer subscript of its position in the vector. The symbol is then placed at the start of the vector moving other symbols up to make room. The effect is that recently accessed symbols will be encoded by small integers and those that have not be referenced for a long time by larger ones.

There are many variants to this scheme. Since cyclic rotation of the symbols is relatively expensive, it may be worth moving the symbol just one position towards the front each time. The result is still adaptive but adapts more slowly. Another scheme is to maintain frequency counts of all symbol accesses and ensure that the symbols in the vector are always maintained in decreasing frequency order.

14.4 Arithmetic Coding

One problem with Huffman coding is that each symbol encodes into a whole number of bits. As a result one can waste, on average, more than half a bit for each symbol sent. Arithmetic encoding addresses this problem. The basic idea is to encode the entire string as a number in the range $[0, 1)$ (**notation:** $x\epsilon[a, b)$ means $a \leq x < b$, and $x\epsilon[a, b]$ means $a \leq x \leq b$). Example: `314159` (a string containing only the characters `1` to `9`) can be encoded as the number: .3141590. It can be decoded by repeated multiplication by 10 and the removal of the integer part.

.3141590	3
.141590	1
.41590	4
.1590	1
.590	5
.90	9
.0	eof

Note that 0 is used as the end-of-file marker. The above encoding is optimal if the the digits and `eof` have equal equal probability.

When the symbol frequencies are not all equal non uniform splitting is used. Suppose the input string consists of the characters `C A A B C C B C B eof` where the character frequencies are `eof`(1), `A`(2), `B`(3) and `C`(4). The initial range $[0, 1)$ is split into subranges with widths proportional to the symbol frequencies: `eof`$[0, .1)$, `A`$[.1, .3)$, `B`$[.3, .6)$ and `C`$[.6, 1.0)$. The first input symbol (`C`) selects its range ($[.6, 1.0)$. This range is then similarly split into subranges `eof`$[.60, .64)$, `A`$[.64, .72)$, `B`$[.72, .84)$ and `C`$[.84, 1.00)$, and the appropriate one for `A`, the next input symbol, is $[.64, .72)$. The next character (`A`) refines the range to $[.644, .652)$. This process continues until all the input has been processed. The lower bound of the final (tiny) range is the arithmetic encoding of the original string.

Assuming the character probabilities are independent, the resulting encoding will be uniformly distributed over $[0, 1]$ and will be, in a sense, optimal.

The problem with arithmetic encoding is that we do no wish to use floating point aritmetic with unbounded precision. We would prefer to do it all with 16 (or 32 bit) integer arithmetic, interleaving reading the input with output of the binary digits of the encoded number. Full details involved in making this all work are, of course, a little delicate! As an indication of how it is done, we will use 3-bit arithmetic on the above example.

Since decoding will also use 3-bits, we must be able to determine the first character from the first 3-bits of the encoding. This forces us to adjust the frequency counts to have a cumulative total of 8, and for implementation reason no symbol other than `eof` may have a count less than 2. For our data the following allocation could be chosen.

```
000 001 010 011 100 101 110 111
 0   1   2   3   4   5   6   7
eof  A   A   B   B   C   C   C
```

In the course of the encoding we may need symbol mappings over smaller ranges down to nearly half the size. The following could be used.

```
R     0   1   2   3   4   5   6   7
8    eof  A   A   B   B   C   C   C
7    eof  A   B   B   C   C   C
6    eof  A   B   B   C   C
5    eof  A   B   C   C
```

These scaled ranges can be computed from the original set by the transformation: `[l,h] -> [1+(l-1)*R/8, h*R/8]`. For example, when `C[5,7]` is scaled to fit in a range of size 6, the new limits are:

`[1+(5-1)*6/8, 7*6/8] = [4,5]`

The exact formula is unimportant, provided both the encoder and decoder use the same one, and provided no symbols get lost.

The following two pages (to be explained in lectures) show the arithmetic encoding and decoding of the of the string: `C A A B C w` based on the frequency counts given above. We are using `w` to stand for `eof`.

14.4.1 Encoding of CAABCw

```
              0   0   0   0   1   1   1   1
              0   0   1   1   0   0   1   1
              0   1   0   1   0   1   0   1
C           |-w---A---A---B---B--+C+++C+++C+|
                                 =1===1===1=           => 1
                                  0   1   1
  double                          1   0   1
                            0   0   1   1   1   1
                            1   1   0   0   1   1
                            0   1   0   1   0   1
A                         |-w--+A+--B---B---C---C-|
                               =0=                     => 0
                                1
  double                        1
                             =1===1=                   => 1
                              1   1
  double                      0   1
                         =1===1===1===1=               => 1
                          0   0   1   1
  double                  0   1   0   1
                  0   0   0   0   1   1   1   1
                  0   0   1   1   0   0   1   1
                  0   1   0   1   0   1   0   1
A               |-w--+A+++A+--B---B---C---C---C-|
                     =0===0=                           => 0
                      0   1
  double              1   0
                  0   0   1   1
                  1   1   0   0
  double#         0   1   0   1
          0   0   0   0   1   1   1   1
          1###1###1###1###0###0###0###0
          0   0   1   1   0   0   1   1
          0   1   0   1   0   1   0   1
B       |-w---A---A--+B+++B+--C---C---C-|
                      0   1
                      1###0
                      1   0
  double#             1   0
                  0   0   1   1
                  1###1###0###0
                  1###1###0###0
                  1   1   0   0
  double#         0   1   0   1
          0   0   0   0   1   1   1   1
          1###1###1###1###0###0###0###0
          1###1###1###1###0###0###0###0
          1###1###1###1###0###0###0###0
          0   0   1   1   0   0   1   1
          0   1   0   1   0   1   0   1
C       |-w---A---A---B---B--+C+++C+++C+|
                             =1===1===1=               => 1
                             =0===0===0=               => 0
                             =0===0===0=               => 0
                             =0===0===0=               => 0
                              0   1   1
  double                      1   0   1
                        0   0   1   1   1   1
                        1   1   0   0   1   1
                        0   1   0   1   0   1
w                     |+w+--A---B---B---C---C-|        => 010
```

14.4.2 Decoding of 101101000010

```
101 101000010      0   0   0   0   1   1   1   1
                   0   0   1   1   0   0   1   1
                   0   1   0   1   0   1   0   1
101 101000010    |-w---A---A---B---B-+(C)++C+++C+|        => C
                                      =1===1===1=
                                       0   1   1
    double                             1   0   1
011 01000010                     0   0   1   1   1   1
                                 1   1   0   0   1   1
                                 0   1   0   1   0   1
011 01000010                   |-w-+(A)+-B---B---C---C-|  => A
                                    =0=
                                     1
    double                           1
110 1000010                       =1===1=
                                   1   1
    double                         0   1
101 000010                    =1===1===1===1=
                               0   0   1   1
    double                     0   1   0   1
010 00010              0   0   0   0   1   1   1   1
                       0   0   1   1   0   0   1   1
                       0   1   0   1   0   1   0   1
010 00010            |-w--+A++(A)+-B---B---C---C---C-|    => A
                          =0===0=
                           0   1
    double                 1   0
100 0010               0   0   1   1
                       1   1   0   0
    double#            0   1   0   1
1#00 010       0   0   0   0   1   1   1   1
               1###1###1###1###0###0###0###0
               0   0   1   1   0   0   1   1
               0   1   0   1   0   1   0   1
1#00 010     |-w---A---A--+B++(B)+-C---C---C-|            => B
                           0   1
                           1###0
                           1   0
    double#                1   0
1##00 10               0   0   1   1
                       1###1###0###0
                       1###1###0###0
                       1   1   0   0
    double#            0   1   0   1
1###01 0       0   0   0   0   1   1   1   1
               1###1###1###1###0###0###0###0
               1###1###1###1###0###0###0###0
               1###1###1###1###0###0###0###0
               0   0   1   1   0   0   1   1
               0   1   0   1   0   1   0   1
1###01 0     |-w---A---A---B---B--(C)++C+++C+|            => C
                                  =1===1===1=
                                  =0===0===0=
                                  =0===0===0=
                                  =0===0===0=
                                   0   1   1
    double                         1   0   1
010                          0   0   1   1   1   1
                             1   1   0   0   1   1
                             0   1   0   1   0   1
010                        |(w)+-A---B---B---C---C-|      => w
```

14.5 Lempel Ziv

This is a family of methods that have become very popular lately. They are based on the observation that in many types of data it is common for strings to be repeated. For instance in a program the names of a user's variables will tend to appear very often, as will language keywords (and things such as repeated spaces). The idea behind Lempel-Ziv is to send messages using a greatly extended alphabet (maybe up to 16 bit characters) and to allocate all the extra codes that this provides to stand for strings that have appeared earlier in the text.

It suffices to allocate a new code to each pair of tokens that get put into the compressed file. This is because after a while these tokens will themselves stand for increasingly long strings of characters, so single output units can correspond to arbitrary length strings. At the start of a file while only a few extra tokens have been introduced one uses (say) 9-bit output characters, increasing the width used as more pairs have been seen.

Because the first time any particular pair of characters is used it gets sent directly (the single-character replacement is used on all subsequent occasions[14]) a decoder naturally sees enough information to build its own copy of the coding table without needing any extra information.

If at any stage the coding table becomes too big the entire compression process can be restarted using initially empty ones.

[14] A special case arises when the second occurrence appears immediately, which I will skip over in these abbreviated notes

14.5.1 Example

```
Encode:  A B R A C A D A B R A R A R A B R A C A D A B R A eof

input    code   bits        table

                             0: eof
                             1: A
                             2: B
                             3: C
                             4: D
                             5: R
A       =>  1    001         6: AB
B       =>  2    010         7: BR
R       =>  5    101         8: RA    -- change to 4 bit codes
A       =>  1   0001         9: AC
C       =>  3   0011        10: CA
A       =>  1   0001        11: AD
D       =>  4   0100        12: DA
AB      =>  6   0110        13: ABR
RA      =>  8   1000        14: RAR
RAR     => 14   1110        15: RARA
ABR     => 13   1101        16: ABRA  -- change to 5 bit codes
AC      =>  9  01001        17: ACA
AD      => 11  01011        18: ADA
ABRA    => 16  10000        19: ABRAw
eof     =>  0  00000

Total length: 3*3 + 8*4 + 4*5 = 61 bits

Decoding

 bits  code   output    table            partial
                                          entry
                         0: eof   - eof
                         1: A     - A
                         2: B     - B
                         3: C     - C
                         4: D     - D
                         5: R     - R
  001    1  =>   A                        6: A?
  010    2  =>   B       6: AB    1 B     7: B?
  101    5  =>   R       7: BR    2 R     8: R?   -- now use 4 bits
 0001    1  =>   A       8: RA    5 A     9: A?
 0011    3  =>   C       9: AC    1 C    10: C?
 0001    1  =>   A      10: CA    3 A    11: A?
 0100    4  =>   D      11: AD    1 D    12: D?
 0110    6  =>   AB     12: DA    4 A    13: AB?
 1000    8  =>   RA     13: ABR   6 R    14: RA?
 1110   14  =>   RAR    14: RAR   8 R    15: RAR? -- careful!
 1101   13  =>   ABR    15: RARA 14 A    16: ABR? -- now use 5 bits
01001    9  =>   AC     16: ABRA 13 A    17: AC?
01011   11  =>   AD     17: ACA   9 D    18: AD?
10000   16  =>   ABRA   18: ADA  11 A    19: ABRA?
00000    0  =>   eof
```

14.6 Burrows-Wheeler Block Compression

Consider the string: `A B A A B A B A B B B .`
Form the matrix of cyclic rotations and sort the rows:

```
       unsorted                     sorted

A B A A B A B A B B B .      . A B A A B A B A B B B
B A A B A B A B B B . A      A A B A B A B B B . A B
A A B A B A B B B . A B      A B A A B A B A B B B .
A B A B A B B B . A B A      A B A B A B B B . A B A
B A B A B B B . A B A A      A B A B B B . A B A A B
A B A B B B . A B A A B      A B B B . A B A A B A B
B A B B B . A B A A B A      B . A B A A B A B A B B
A B B B . A B A A B A B      B A A B A B A B B B . A
B B B . A B A A B A B A      B A B A B B B . A B A A
B B . A B A A B A B A B      B A B B B . A B A A B A
B . A B A A B A B A B B      B B . A B A A B A B A B
. A B A A B A B A B B B      B B B . A B A A B A B A
```

In general the last column is much easier to compact (using run-length encoding, move-to-front buffering followed by Huffman) than the original string (particularly for natural text). It also has the surprising property that it contains just sufficient information to reconstruct the original string. Observe the following two diagrams:

```
. A B A A B A B A B B B------------*  . A B A A B A B A B B B
A A B A B A B B B . A B----------* |  A A B A B A B B B . A B
A B A A B A B A B B B .          | |  A B A A B A B A B B B .
A B A B A B B B . A B A          | |  A B A B A B B B . A B A
A B A B B B . A B A A B--------* | |  A B A B B B . A B A A B
A B B B . A B A A B A B------* | | |  A B B B . A B A A B A B
B . A B A A B A B A B B----* | | | *--B . A B A A B A B A B B
B A A B A B A B B B . A    | | | *----B A A B A B A B B B . A
B A B A B B B . A B A A    | | *------B A B A B B B . A B A A
B A B B B . A B A A B A    | *--------B A B B B . A B A A B A
B B . A B A A B A B A B--* *----------B B . A B A A B A B A B
B B B . A B A A B A B A  *------------B B B . A B A A B A B A
```

```
 i    L[i] C[ .   A   B ] P[i]

              0   0   0
 0      B             +---> 0
              0   0   1
 1      B             +---> 1
              0   0   2
 2      .     +-----------> 0
              1   0   2
 3      A         +-------> 0
              1   1   2
 4      B             +---> 2
              1   1   3
 5      B             +---> 3
              1   1   4
 6      B             +---> 4
              1   1   5
 7      A         +-------> 1
              1   2   5
 8      A         +-------> 2
              1   3   5
 9      A         +-------> 3
              1   4   5
10      B             +---> 5
              1   4   6
11      A         +-------> 4
              1   5   6
              |   |   |
            0-+-1-+-6-+-12
           /   /     \
          /   /       \
         /   /         \
        /   /           \
       / . /  A A A A A  \    B B B B B B
      /   /               \
     /   /                 \
    0   1   2   3   4   5   6   7   8   9   10  11
    |   |                   |
    .0  A0  A1  A2  A3  A4  B0  B1  B2  B3  B4  B5
    B0  B1  .0  A0  B2  B3  B4  A1  A2  A3  B5  A4
```

```
11           .0                                      .
10   B0<------*                                      B
 9    *--------------------->B4                      B
 8                            *------------->B5      B
 7                                            *->A4  A
 6                       B3<----------------------*  B
 5                        *------------->A3          A
 4                   B2<-----------------*           B
 3                    *------------->A2              A
 2               A0<------------------*              A
 1       B1<------*                                  B
 0        *--------------------->A1                  A
```

The Burrows-Wheeler algorithm is about the same speed as unix compress or gzip and, for large files, typically compresses more successfully (eg. compress:1,246,286, gzip:1,024,887, BW:856,233 bytes).

14.6.1 Sorting the suffixes

Sorting the suffixes is an important part of the Burrows-Wheeler algorithm. Care must be taken to do it efficiently bearing in mind that the original string may be tens of mega-bytes in length and may contain long repeats of the same character and may have very long identical substrings.

The following shows some suffixes before and after sorting.

```
 0 ABRACADABRARARABRACADABRA.    24 A.
 1 BRACADABRARARABRACADABRA.     21 ABRA.
 2 RACADABRARARABRACADABRA.      14 ABRACADABRA.
 3 ACADABRARARABRACADABRA.        0 ABRACADABRARARABRACADABRA.
 4 CADABRARARABRACADABRA.         7 ABRARARABRACADABRA.
 5 ADABRARARABRACADABRA.         17 ACADABRA.
 6 DABRARARABRACADABRA.           3 ACADABRARARABRACADABRA.
 7 ABRARARABRACADABRA.           19 ADABRA.
 8 BRARARABRACADABRA.             5 ADABRARARABRACADABRA.
 9 RARARABRACADABRA.             12 ARABRACADABRA.
10 ARARABRACADABRA.              10 ARARABRACADABRA.
11 RARABRACADABRA.               22 BRA.
12 ARABRACADABRA.                15 BRACADABRA.
13 RABRACADABRA.                  1 BRACADABRARARABRACADABRA.
14 ABRACADABRA.                   8 BRARARABRACADABRA.
15 BRACADABRA.                   18 CADABRA.
16 RACADABRA.                     4 CADABRARARABRACADABRA.
17 ACADABRA.                     20 DABRA.
18 CADABRA.                       6 DABRARARABRACADABRA.
19 ADABRA.                       23 RA.
20 DABRA.                        13 RABRACADABRA.
21 ABRA.                         16 RACADABRA.
22 BRA.                           2 RACADABRARARABRACADABRA.
23 RA.                           11 RARABRACADABRA.
24 A.                             9 RARARABRACADABRA.
```

Assume the length of the string is `N`, Allocate two vectors `Data[0..N-1]` and `V[0..N-1]`. For each `i`, initialize `Data[i]` to contain the first 4 characters of the suffix starting at position `i`, and initialize all `V[i]=i`. The elements of `V` will be used as subscripts into `Data`. The following illustrates the sorting process.

Index	V Data	V1	VD	VC	VB	VR	VA
0:	0->ABRA	24->A...					24->A000
1:	1->BRAC	0->ABRA					21->A001
2:	2->RACA	7->ABRA					14->A002
3:	3->ACAD	14->ABRA					0->A003
4:	4->CADA	21->ABRA					7->A004
5:	5->ADAB	3->ACAD					17->A005
6:	6->DABR	17->ACAD					3->A006
7:	7->ABRA	5->ADAB					19->A007
8:	8->BRAR	19->ADAB					5->A008
9:	9->RARA	10->ARAR					12->A009
10:	10->ARAR	12->ARAB					10->A010
11:	11->RARA	1->BRAC			22->B000		
12:	12->ARAB	8->BRAR			15->B001		
13:	13->RABR	15->BRAC			1->B002		
14:	14->ABRA	22->BRA.			8->B003		
15:	15->BRAC	4->CADA		18->C000			
16:	16->RACA	18->CADA		4->C001			
17:	17->ACAD	6->DABR	20->D000				
18:	18->CADA	20->DABR	6->D001				
19:	19->ADAB	2->RACA				23->R000	
20:	20->DABR	9->RARA				13->R001	
21:	21->ABRA	11->RARA				16->R002	
22:	22->BRA.	13->RABR				2->R003	
23:	23->RA..	16->RACA				11->R004	
24:	24->A...	23->RA..				9->R005	

Putting 4 characters in each data element allows suffixes to be compared 4 characters at a time using aligned data accesses. It also allows many slow cases to be eliminated.

`V1` is `V` after radix sorting the pointers on the first 2 characters of the suffix.

`VD` is the result of sorting the suffixes starting with the rarest letter (`D`) and then replacing the least significant 3 characters of each element with a 24-bit integer giving the sorted position (within the `Ds`). This replacement sorts into the same position as the original value, but has the desirable property that it is distinct from all other elements (improving the efficiency of later sorts).

`VC`, `VB`, `VR` and `A` are the results of apply the same process to suffixes starting with `C`, `B`, `R` and `A`, respectively. When this is done, the sorting of suffixes is complete.

14.6.2 Improvements

The first concerns long repeats of the same character.

For a given letter, `C` say, sort all suffixes starting with `C`X where X is any letter different from `C`. The sorting of suffixes starting `CC` is now easy. The following is based on the string:

```
 A B C C C D C C B C C C D D C C C C B A C C D C C A C B B
            ...
            C C(B)C C C D D

          / D C(C)A C     >-+      earliest suffix starting C...
sorted   |  C C(C)B A     >-|-+
         |  C A(C)B B       | |
          \ D C(C)B C     >-|-|-+
                            | | |
          / A B(C)C C       * | |       C D(C)C A C
         |  B C(C)C D         * | +---< C C(C)C B A
         |  C D(C)C B           * |       D(C)C B C
         |  C B(C)C C             * +-< D C(C)C C B A
         |  B C(C)C D               *     D(C)C C C B A
not      |  D D(C)C C               *     B(C)C C D C C B
sorted   |  D C(C)C C             * |   C B(C)C C D D
         |  C C(C)B A           * | |     A(C)C D C C A
         |  B A(C)C D         * | | +-< B C(C)C D C C B
          \ C D(C)C A       * | | +---< B C(C)C D D
                            | | |
          / A C(C)D C C A >-|-|-+
sorted   |  C C(C)D C C B >-|-+
          \ C C(C)D D     >-+       latest suffix starting C...

            C C(D)C C A
            ...
```

A second strategy, not described here, improves the handling of suffixes containing long repeated patterns (eg. `ABCABCABC...`).

15 Algorithms on graphs

The general heading "graphs" covers a number of useful variations. Perhaps the simplest case is that of a general directed graph — this has a set of vertices, and a set of ordered pairs of vertices that are taken to stand for (directed) edges. Note that it is common to demand that the ordered pairs of vertices are all distinct, and this rules out having parallel edges. In some cases it may also be useful either to assume that for each vertex v that the edge (v, v) is present, or to demand that no edges joining any vertex to itself can appear. If in a graph every edge (v_1, v_2) that appears is accompanied by an edge joining the same vertices but in the other sense, i.e. (v_2, v_1), then the graph is said to be *undirected*, and the edges are then though of as unordered pairs of vertices. A chain of edges in a graph form a *path*, and if each pair of vertices in the entire graph have a path linking them then the graph is *connected*. A non-trivial path from a vertex back to itself is called a *cycle*. Graphs without cycles have special importance, and the abbreviation *DAG* stands for Directed Acyclic Graph. An undirected graph without cycles in it is a *tree*. If the vertices of a graph can be split into two sets, A and B say, and each edge of the graph has one end in A and the other in B then the graph is said to be *bipartite*. The definition of a graph can be extended to allow values to be associated with each edge — these will often be called weights or distances. Graphs can be used to represent a great many things, from road networks to register-use in optimising compilers, to databases, to timetable constraints. The algorithms using them discussed here are only the tip of an important iceberg.

15.1 Depth-first and breadth-first searching

Many problems involving graphs are solved by systematically searching. Even when the underlying structure is a graph the structure of a search can be regarded as a tree. The two main strategies for inspecting a tree are depth-first and breadth-first. Depth-first search corresponds to the most natural recursive procedure for walking over the tree. A feature is that (from any particular node) the whole of one sub-tree is investigated before the other is looked at at all.

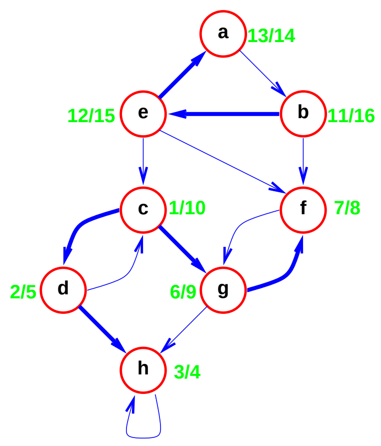
The recursion in depth-first search could naturally be implemented using a stack. If that stack is replaced by a queue but the rest of the code is unaltered you get a version of breadth-first search where all nodes at a distance k from the root get inspected before any at depth $k+1$ are visited. Breadth-first search can often avoid getting lost in fruitless scanning of deep parts of the tree, but the queue that it uses often requires much more memory than depth-first search's stack.

15.1.1 Finding Strongly Connected Components

This is a simple but rather surprising application of depth-first searching that can solve the problem in time $O(n)$.

A *strongly connected component* of a directed graph is a maximal subset of its vertices for which paths exist from any vertex to any other. The algorithm is as follows:

1) Perform depth first search on graph $G(V, E)$, attaching the discovery time ($d[u]$) and the finishing time ($f[u]$) to every vertex (u) of G. For example, the following shows a graph after this operation.

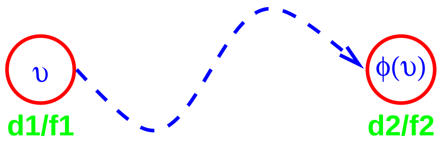


Next, we define the forefather $\phi(u)$ of a vertex u as follows:

$$\begin{array}{rl} & \phi(u) = w \\ \text{where} & w \epsilon V \\ \text{and} & u \longrightarrow \cdots \longrightarrow w \\ \text{and} & \forall w'(u \longrightarrow \cdots \longrightarrow w' \Rightarrow f[w'] \leq f[w]) \end{array}$$

$$\begin{array}{rl} \text{Clearly,} & \text{since } u \longrightarrow \cdots \longrightarrow u \\ & f[u] \leq f[\phi(u)] \qquad\qquad (1) \end{array}$$

It has the important property that $\phi(u)$ and u are in the same strongly connected component. Informal proof:



$$\begin{array}{rl} \text{Either} & u = \phi(u) \\ \text{or} & u \neq \phi(u) \\ & \text{if } d2 < f2 < d1 < f1 \\ & \text{or } d1 < d2 < f2 < f1 \\ & \qquad \text{contradicts of (1)} \\ & \text{if } d1 < f1 < d2 < f2 \\ & \qquad \text{contradicts } u \longrightarrow \cdots \longrightarrow \phi(u) \\ & \text{so } d2 < d1 < f1 < f2 \\ & \text{ie } \phi(u) \longrightarrow \cdots \longrightarrow u \\ \text{so} & u \text{ and } \phi(u) \text{ are in the same} \\ & \text{strongly connected component.} \end{array}$$

Continuing with the algorithm.

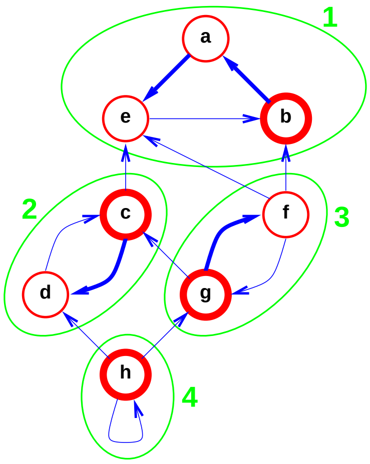
2) Find the vertex r with largest $f[r]$ that is not in any strongly connected component so far identified. Note that r must be a forefather.

3) Form the set of vertices $\{u|\phi(u) = r\}$ – i.e. the strongly connected component containing r. This set is the same as $\{u|u \longrightarrow \cdots \longrightarrow r\}$

   This set is the set of vertices reachable from r in the graph $G^T = G$ with all its edges reversed. This set can be found using DFS on G^T.

4) Repeat from (2) until all components have been found.

The resulting strongly connected components are shown below.



15.2 Minimum Cost Spanning Tree

Given a connected undirected graph with n edges where the edges have all been labelled with "lengths", the problem of finding a minimum spanning tree is that of finding the shortest sub-graph that links all vertices. This must necessarily be a tree. For suppose it were not, then it would contain a cycle. Removing any one edge from the cycle would leave the graph strictly smaller but still connecting all the vertices.

One algorithm that finds minimal spanning subtrees involves growing a sub-graph by adding (at each step) that edge of the full graph that (a) joins a new vertex onto the sub-graph we have already and (b) is the shortest edge with that property.

The main questions about this are first how do we prove that it works correctly, and second how do we implement it efficiently.

15.3 Single Source shortest paths

This starts with a (directed) graph with the edges labelled with lengths. Two vertices are identified, and the challenge is to find the shortest route through the graph from one to the other. An amazing fact is that for sparse graphs the best ways of solving this known may do as much work as a procedure that sets out to find distances from the source to **all** the other points in the entire graph. One of the things that this illustrates is that our intuition on graph problems may mis-direct if we think in terms of particular applications (for instance distances in a road atlas in this case) but then try to make statements about arbitrary graphs.

The approved algorithm for solving this problem is a form of breadth-first search from the source, visiting other vertices in order of the shortest distance from the source to each of them. This can be implemented using a priority queue to control the order in which vertices are considered. When, in due course, the selected destination vertex is reached the algorithm can stop. The challenge of finding the very best possible implemementation of the queue-like structures required here turns out to be harder than one might hope!

If all edges in the graph have the same length the priority queue management for this procedure collapses into something rather easier.

15.4 Connectedness

For this problem I will start by thinking of my graph as if it is represented by an adjacency matrix. If the bits in this matrix are a_{ij} then I want to consider the interpretation of the graph with matrix elements defined by

$$b_{ij} = a_{ij} \vee \bigvee_k (a_{ik} \wedge a_{kj})$$

where $\wedge$ and $\vee$ indicate **and** and **or** operations. A moment or two of thought reveals that the new matrix shows edges wherever there is a link of length one or two in the original graph.

Repeating this operation would allow us to get all paths of length up to $4, 8, 16, \ldots$ and eventually all possible paths. But we can in fact do better with a program that is very closely related:

```
for k = 1 to n do
  for i = 1 to n do
    for j = 1 to n do a[i,j] = a[i,j] | (a[i,k] & a[k,j]);
```

is very much like the variant on matrix multiplication given above, but solves the whole problem in one pass. Can you see why, and explain it clearly?

15.5 All points shortest path

Try taking the above discussion of connectedness analysis and re-working it with **add** and **min** operations instead of boolean **and** and **or**. See how this can be used to fill in the shortest distances between all pairs of points. What value must be used in matrix elements corresponding to pairs of graph vertices not directly joined by an edge?

15.6 Bipartite Graphs and matchings

A matching in a bipartite graph is a collection of edges such that each vertex of the graph is included in at most one of the selected edges. A maximal matching is then obviously as large a subset of the edges that has this property as is possible. Why might one want to find a matching? Well bipartite graphs and matchings can be used to represent many resource allocation problems.

Weighted matching problems are where bipartite graphs have the edges labelled with values, and it is necessary to find the matching that maximises the sum of weights on the edges selected.

Simple direct search through all possible combinations of edges would provide a direct way of finding maximal matchings, but would have costs growing exponentially with the number of edges in the graph — even for small graphs it is not a feasible attack.

A way of solving the (unweighted) matching problem uses "augmenting paths", a term you can look up in AHU or CLR.

16 Algorithms on strings

This topic is characterised by the fact that the basic data structure involved is almost vanishingly simple — just a few strings. Note however that some "string" problems may use sequences of items that are not just simple characters: searching, matching and re-writing can still be useful operations whatever the elements of the "strings" are. For instance in hardware simulation there may be need for operations that work with strings of bits. The main problem addressed will one that treats one string as a pattern to be searched for and another (typically rather longer) as a text to scan. The result of searching will either be a flag to indicate that the text does not contain a substring identical to the pattern, or a pointer to the first match. For this very simple form of matching note that the pattern is a simple fixed string: there are no clever options or wildcards. This is like a simple search that might be done in a text editor.

16.1 Simple String Matching

If the pattern is n characters long and the text is m long, then the simplest possible search algorithm would test for a match at positions 1, 2, 3, ... in turn, stopping on a match or at the end of the text. In the worst case testing for a match at any particular position may involve checking all n characters of the pattern (the mismatch could occur on the last character). If no match is found in the entire text there will have been tests at $m - n$ places, so the total cost might be $n \times (m - n) = \Theta(mn)$. This worst case could be attained if the pattern was something like `aaaa...aaab` and the text was just a long string `aaa...aaa`, but in very many applications the practical cost grows at a rate more like $m + n$, basically because most that mis-matches occur can be detected after looking at only a few characters.

16.2 Precomputation on the pattern

Knuth-Morris-Pratt: Start matching your pattern against the start of the text. If you find a match then very good — exit! But if not, and if you have matched k out of your n characters then you know exactly what the first k characters of the text are. Thus when you come to start a match at the next position in effect you have already inspected the first $k-1$ places, and you should know if they match. If they do then you just need to continue from there on. Otherwise you can try a further-up match. In the most extreme case of mis-matches after a failure in the first match at position n you can move the place that you try the next match on by n places. Now then: how can that insight be turned into a practical algorithm?

Boyer-Moore: The simple matching algorithm started checking the first character of the pattern against the first character of the text. Now consider making the first test be the nth character of each. If these characters agree then go back to position $n-1$, $n-2$ and so on. If a mis-match is found then the characters seen in the text can indicate where the next possible alignment of pattern against text could be. In the best case of mis-matches it will only be necessary to inspect every nth character of the text. Once again these notes just provide a clue to an idea, and leave the details (and the analysis of costs) to textbooks and lectures.

Rabin-Karp: Some of the problem with string matching is that testing if the pattern matches at some point has possible cost n (the length of the pattern). It would be nice if we could find a constant cost test that was almost as reliable. The Rabin-Karp idea involves computing a hash function of the pattern, and organising things in such a way that it is easy to compute the same hash function on each n character segment of the text. If the hashes match there is a high probability that the strings will, so it is worth following through with a full blow by blow comparison. With a good hash function there will almost never be false matches leading to unnecessary work, and the total cost of the search will be proportional to $m+n$. It is still possible (but very very unlikely) for this process to deliver false matches at almost all positions and cost as much as a naive search would have.

16.2.1 Tiny examples

Brute force

```
x x x C x x x B x x x x x x A x x x B x x x A x x C
x x x A . . . . . . . . . .
  x x x . . . . . . . . . . .
    x x . . . . . . . . . . . .
      x . . . . . . . . . . . . .
        x x x A . . . . . . . . . .
          x x x . . . . . . . . . . .
            x x . . . . . . . . . . . .
              x . . . . . . . . . . . . .
                x x x A . . . . . . . . . .
                  x x x A . . . . . . . . . .
                    x x x A x x x B x x x A x x
Comparisons: 42
```

Knuth-Morris-Pratt

```
x x x A x x x B x x x A x x
#                                                     Fail at  1 =>  shift  1
  x x x A x x x B x x x A x x
x #                                                   Fail at  2 =>  shift  2
    x x x A x x x B x x x A x x
x x #                                                 Fail at  3 =>  shift  3
      x x x A x x x B x x x A x x
x x x #                                               Fail at  4 =>  shift  1
  x x x A x x x B x x x A x x
x x x A #                                             Fail at  5 =>  shift  5
          x x x A x x x B x x x A x x
x x x A x #                                           Fail at  6 =>  shift  6
            x x x A x x x B x x x A x x
x x x A x x #                                         Fail at  7 =>  shift  7
              x x x A x x x B x x x A x x
x x x A x x x #                                       Fail at  8 =>  shift  4
        x x x A x x x B x x x A x x
x x x A x x x B #                                     Fail at  9 =>  shift  9
                  x x x A x x x B x x x A x x
x x x A x x x B x #                                   Fail at 10 =>  shift 10
                    x x x A x x x B x x x A x x
x x x A x x x B x x #                                 Fail at 11 =>  shift 11
                      x x x A x x x B x x x A x x
x x x A x x x B x x x #                               Fail at 12 =>  shift  9
                  x x x A x x x B x x x A x x
x x x A x x x B x x x A #                             Fail at 13 =>  shift 13
                          x x x A x x x B x x x A x x
x x x A x x x B x x x A x #                           Fail at 14 =>  shift 14
                            x x x A x x x B x x x A x x
```

The table above shows how much the pattern must be shifted based on the position of the first mismatching character.

```
x x x C x x x B x x x x x x A x x x B x x x A x x C
x x x A . . . . . . | . . .                   |       Fail at  4 =>  shift  1
  . . x . . . . . . | . . . .                 |       Fail at  3 =>  shift  3
        x x x A . . | . . . . . . .           |       Fail at  4 =>  shift  1
          x x x . . | . . . . . . .           |       Fail at  3 =>  shift  3
                x x x A . . . . . . . . . .   |       Fail at  4 =>  shift  1
                  . . x A . . . . . . . . . . |       Fail at  4 =>  shift  1
                    . . x A x x x B x x x A x x       Success
Comparisons: 30
```

Boyer-Moore

First compute the table giving the minimum pattern shift based on the position of the first mismatch.

```
x x x A x x x B x x x A x x
                          #                           Fail at 14 =>  shift  2
    x x x A x x x B x x x A x x
                        # x                           Fail at 13 =>  shift  1
  x x x A x x x B x x x A x x
                      # x x                           Fail at 12 =>  shift  3
      x x x A x x x B x x x A x x
                    # A x x                           Fail at 11 =>  shift 12
                        x x x A x x x B x x x A x x
                  # x A x x                           Fail at 10 =>  shift 12
                        x x x A x x x B x x x A x x
                # x x A x x                           Fail at  9 =>  shift 12
                        x x x A x x x B x x x A x x
              # x x x A x x                           Fail at  8 =>  shift  8
                x x x A x x x B x x x A x x
            # B x x x A x x                           Fail at  7 =>  shift  8
                x x x A x x x B x x x A x x
          # x B x x x A x x                           Fail at  6 =>  shift  8
                x x x A x x x B x x x A x x
        # x x B x x x A x x                           Fail at  5 =>  shift  8
                x x x A x x x B x x x A x x
      # x x x B x x x A x x                           Fail at  4 =>  shift  8
                x x x A x x x B x x x A x x
    # x x x x B x x x A x x                           Fail at  3 =>  shift  8
                x x x A x x x B x x x A x x
  # x x x x x B x x x A x x                           Fail at  2 =>  shift  8
                x x x A x x x B x x x A x x
# x x A x x x B x x x A x x                           Fail at  1 =>  shift  8
                x x x A x x x B x x x A x x
```

We will call the shift in the above table the *mismatch shift*.

```
                          x                 => dist['x'] = 14
x x x A x x x B x x x A x x
                          A                 => dist['A'] = 12
    x x x A x x x B x x x A x x
                          B                 => dist['B'] =  8
            x x x A x x x B x x x A x x
   any other character -> y                 => dist['y'] =  0
                            x x x A x x x B x x x A x x
```

If ch is the text character that mismatchs position j of the pattern and then the pattern must be shifted by at least j-dist[ch]. We will call this the *character shift*. On a mismatch, the Boyer-Moore algorithm shifts the pattern by the larger of the mismatch shift and the character shift.

```
x x x C x x x B x x x x x x A x x x B x x x A x x C
. . . . . . . . . . . . . x                       ch A at 14 => shift 2
    . . . . . . . B x x x A x x                   fail at  8 => shift 8
                    . . . . . . x B x x x A x x   success
Comparisons: 16
```

Rabin-Karp

In this example we search for `8979` in `3141592653589793`, using the hash function `H(dddd) = dddd mod 31`. So for example, `H(3141)=10`, `H(1415)=20`, `H(4159)=5`, `H(1592)=11`, etc. The hash value of our pattern is 20.

```
Note that H(bcde) = (10*((H(abcd)+10*31-a*M) mod 31) + e) mod 31
                    where M = 1000 mod 31 (=8)

       eg H(1415) = (10*((H(3141)+10*31-3*8) mod 31) + 5) mod 31 = 20

3  1  4  1  5  9  2  6  5  3  5  8  9  7  9  3
        10 20                                     hash match
   8  9  7  9                                     fail
3  1  4  1  5  9  2  6  5  3  5  8  9  7  9  3
               5 11  5 27 18 25 26 24  7 20       hash match
                                 8  9  7  9       success
```

17 Geometric Algorithms

A major feature of geometric algorithms is that one often feels the need to sort items, but sorting in terms of x co-ordinates does not help with a problem's structure in the y direction and vice versa: in general there often seems no obvious way of organising the data. The topics included here are just a sampling that show off some of the techniques that can be applied and provide some basic tools to build upon. Many more geometric algorithms are presented in the computer graphics courses. Large scale computer aided design and the realistic rendering of elaborate scenes will call for plenty of carefully designed data structures and algorithms.

17.1 Use of lines to partition a plane

Representation of a line by an equation. Does it matter what form of the equation is used? Testing if a point is on a line. Deciding which side of a line a point is, and the distance from a point to a line. Interpretation of scalar and vector products in 2 and 3 dimensions.

17.2 Do two line-segments cross?

The two end-points of line segment a must be on opposite sides of the line b, and vice versa. Special and degenerate cases: end of one line is on the other, collinear but overlapping lines, totally coincident lines. Crude clipping to provide cheap detection of certainly separate segments.

17.3 Is a point inside a polygon

Represent a polygon by a list of vertices in order. What does it mean to be "inside" a star polygon (eg. a pentagram) ? Even-odd rule. Special cases. Winding number rule. Testing a single point to see if it is inside a polygon vs. marking all interior points of a figure.

17.4 Convex Hull of a set of points

What is the convex hull of a collection of points in the plane? Note that sensible ways of finding one may depend on whether we expect most of the original points to be on or strictly inside the convex hull. Input and output data structures to be used. The package-wrapping method, and its n^2 worst case. The Graham Scan. Initial removal of interior points as an heuristic to speed things up.

17.5 Closest pair of points

Given a rectangular region of the plane, and a collection of points that lie within this region, how can you find the pair of points that are closest to one another? Suppose there are n points to begin with, what cost can you expect to pay?

Try the "divide and conquer" approach. First, if n is 3 or less just compute the roughly $n^2/2$ pair-wise lengths between points and find the closest two points by brute-force. Otherwise find a vertical line that partitions the points into two almost equal sets, one to its left and one to its right. Apply recursion to solve each of these sub-problems. Suppose that the closest pair of points found in the two recursive calls were a distance δ apart, then either that pair will be the globally closest pair or the true best pair straddles the dividing line. In the latter case the best pair must lie within a vertical strip of width 2δ, so it is just necessary to scan points in this strip. It is already known that points that are on the same side of the original dividing line are at least δ apart, and this fact makes it possible to speed up scanning the strip. With careful support of the data structures needed internally (which includes keeping lists of the points sorted by both x and y co-ordinate) the running time of the strip-scanning can be made linear in n, which leads to the recurrence

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

for the overall cost $f(n)$ of this algorithm, and this is enough to show that the problem can be solved in $\Theta(n \log(n)))$. Note (as always) that the sketch given here explains the some of the ideas of the algorithm, but not all the important details, which you can check out in textbooks or (maybe) lectures!

18 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[15] 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.

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.

[15]Highly reccomended anyway!