Department of

Mathematical Sciences

Algorithms
MX4028

Ian Craw and John Pulham

October 25, 1999, Version 3.3

Additional copies may be obtained from:

Department of Mathematical Sciences
University of Aberdeen
Aberdeen AB9 2TY

DSN: mth199-101533-2

Contents

Foreword **ix**

1 Introduction **1**

1.1 Algorithms; Computer languages . . . 1

1.1.1 Programming . . . 2

1.1.2 An Implementation of the Euclidean Algorithm . . . 2

1.1.3 Recursion . . . 3

1.1.4 Algorithm Types . . . 3

1.2 An Algorithm — Hatching Polygons . . . 3

1.2.1 The Problem . . . 4

1.2.2 General Method . . . 5

1.2.3 Hatching Convex Polygons . . . 6

1.3 The Analysis of an Algorithm . . . 7

1.3.1 Timing . . . 7

1.3.2 A More Precise Analysis . . . 9

1.3.3 Computing detours: A_k^n . . . 9

1.3.4 Approximating H_n . . . 12

1.3.5 Conclusion . . . 13

2 Sorting **17**

2.1 Complications . . . 17

2.2 Two Simple Sorting Algorithms . . . 18

2.2.1 Minimum Element Sort . . . 18

2.2.2 Insertion Sort . . . 19

2.2.3 Proof of Algorithm . . . 19

2.2.4 A More Precise Analysis of Insertion Sort . . . 21

2.3 A better sort: Quicksort . . . 24

2.3.1 Timing the algorithm . . . 25

2.4 Merge Sort . . . 27

2.4.1 Merge Algorithm . . . 27

2.4.2 Merge Sort . . . 28

2.4.3 Timing the Algorithm . . . 29

2.5 Timings for Various Sorting Algorithms . . . 29

2.6 Optimal Sorting . . . 30

2.7 Optimal Merging . . . 31

2.7.1 Some Example Sort Implementations . . . 32

3 Abstract Data Types **37**

3.1 Introduction . . . 37

3.1.1 The ADT Stack . . . 37

3.1.2 The ADT Queue . . . 38

3.1.3 The ADT Array . . . 38
3.1.4 The ADT List . . . 40
3.1.5 The ADT Priority Queue . . . 41
3.2 Trees . . . 41
3.2.1 The ADT tree . . . 41
3.2.2 Traversals . . . 42
3.2.3 Binary Trees . . . 42
3.2.4 Huffman Codes . . . 43
3.3 Heapsort . . . 47

4 Grammers and Parsing 51
4.1 Formal Languages . . . 51
4.2 Formal Grammars . . . 52
4.3 First Example . . . 53
4.4 Second Example . . . 54
4.5 Arithmetic Expressions . . . 55
4.5.1 AE Grammer . . . 56
4.6 Postfix Expressions . . . 57
4.6.1 Postfix Grammer . . . 57
4.7 Converting Infix to Postfix . . . 58
4.7.1 FBIE Grammer . . . 59
4.8 Evaluating a Postfix Expression . . . 60

5 Random Numbers 65
5.1 What is a Random Number? . . . 65
5.2 Generating Random Numbers . . . 66
5.3 Applications of Random Numbers . . . 67
5.3.1 Generating Random Reals . . . 67
5.3.2 Generating Random Integers . . . 67
5.3.3 Choosing ‘with probability p’ . . . 68
5.3.4 Random Shuffles . . . 68
5.3.5 Random Samples . . . 69

6 The Fast Fourier Transform 73
6.1 Multiplying Polynomials . . . 73
6.2 Fourier Series . . . 73
6.3 The Fourier Matrix . . . 74
6.4 Products . . . 75
6.5 Properties of the Fourier Transform . . . 76
6.6 The Fast Fourier Transform. . . . 77
6.6.1 Timing the FFT . . . 78
6.7 Convolution as Filtering . . . 79
6.8 An application to Image Processing . . . 80
6.8.1 Gaussian Blur . . . 81
6.8.2 The Laplace Operator . . . 82

Appendices 85
Big O . . . 85
Solutions to Exercises . . . 86
References . . . 106
Index Entries . . . 107

List of Figures

1.1 A polygon, and a re-entrant polygon. 4
1.2 Hatching a polygon with horizontal lines. 4
1.3 A convex polygon, and a non-convex polygon. 4
1.4 A single hatching line passing in and out. 5
1.5 The range for h. 6
1.6 Hatching a convex polygon. 6
1.7 Flow chart to find the minimum element in a list. 8
1.8 Graph of $y = 1/x$. 13

2.1 Potential distributions of running times about the mean. 23
2.2 A decision tree for an element of S_3. 30
2.3 Merging files of different sizes. 31

3.1 Sorting the array of flags 39
3.2 A list implemented using cells and pointers. 40
3.3 The same list after an addition and a deletion. 41
3.4 A simple tree. 42
3.5 A complete binary tree: the only “missing” entries can be on the last row. 43
3.6 Binary trees representing the codes in Table 3.1 44
3.7 Huffman encoding tree 45
3.8 Adding the element “P” to a priority queue. 48
3.9 The resulting priority queue, after the heap condition is restored. 48
3.10 Removing the element “X” of highest priority from Fig. 3.9. 48
3.11 The resulting priority queue, after the heap condition is restored. 48

4.1 The generative structure of a sentence. 52

6.1 Input image. 81
6.2 Output after Gaussian filtering. 81
6.3 Zero crossings 81
6.4 Output of Laplacian filter. 82
6.5 Output of sharpening filter. 82

6 Flow chart to find the minimum element in a list. 88
7 Graph of $y = 1/x$. 88

List of Tables

1.1 Number of detours sorting elements of S_3. 10
1.2 Values of A_k^n for small values of n and k. 11
1.3 The growth of $T(n)$. 13

2.1 Timing insertion sort 23
2.2 Comparing growth rates of algorithms 26
2.3 A comparison of sort times using different sorting methods. 29

3.1 Code 1 has fixed length code; Code 2 has the prefix property. 44
3.2 Symbol frequencies used to build a Huffman Code. 45

4.1 Deriving an expression from a grammer 56
4.2 Possible follow-on symbols. 57
4.3 Converting expressions to postfix notation. 57
4.4 Translating an fbie 60

1 Counting detours for the elements of S_4 90
2 Symbol frequencies used to build a Huffman Code. 93
3 Symbol frequencies used to build a Huffman Code. 94
4 Symbol frequencies used to build a Huffman Code. 96
5 Symbol frequencies used to build a Huffman Code. 96

Foreword

This course[1] studies computer algorithms, considering their construction, validation and effectiveness. After a general introduction to the subject a number of specific topics will be covered. These may include: the problem of sorting data sets into order, the theory of formal grammars and problems such as the parsing of arithmetic expressions, the construction and use of pseudorandom numbers.

Course Numbers

The course exists in two incarnations; as a regular lecture course, and as a reading course. The lecture course can only run when a significant number of members of the class indicate that they wish to take it. In contrast the reading course can always run, but is clearly only there for the (very) small number of students who really want to learn about algorithms. To avoid confusion, there are different "Catalogue of Course" numbers for the two methods of delivery. The normal lecture course is called MX4002, or MX4052 when it is taken by someone who is not doing Honours, and so will be examined in January; in contrast, the reading course is called MX4028, or MX4078 if examined in January.[2] To avoid confusion and unnecessary duplication, the course will be labelled as MX4002 in these notes.

These Notes

The notes contain the material that I use when preparing the actual lectures; in that sense they are *my* lecture notes. They also approximate what you as a student may choose to write down from these lectures; in that sense they are *your* lecture notes. And in each case, they form an approximation: a lecture is a form of communication; I will alter things in the lecture whenever I think a change will improve communication, and you may choose to write down things from the lecture that I have not put in these notes.

> "Lectures were once useful, but now when all can read, and books are so numerous, lectures are unnecessary." *Samuel Johnson, 1799.*

Lecture notes have been around for centuries, either informally, as handwritten notes, or formally as textbooks. Recently improvements in typesetting have made it easier to produce "personalised" printed notes as here, but there has been no fundamental change. Experience shows that very few people are able to use lecture notes as a *substitute* for lectures; if it were otherwise, lecturing, as a profession would have died out by now. To put it another way, "any teacher who can be replaced by a teaching machine, deserves to be". So you should bear in mind that:

- these notes are intended to *complement* the experience you get from attending the lectures; and

[1] This is quoted directly from the corresponding entry in the Catalogue of Courses.
[2] At present it is wrongly designated as MX4528 — the course is *not* available in the second half session

- are available to *supplement* the notes that you take in the lectures.

There is significant value in taking your own notes; you are much more likely to see what is going on as you do so. I hope that by having this version available as well you can quickly correct any errors that occur as you take them.

Theses notes were originally written, in TEX, by Dr John Pulham to accompany a similar course that he gave for several years. This version was developed from these notes. This version conatins a slightly different selection of material. It is written in LATEX which allows a higher level view of the text, and simplifies the preparation of such things as the index on page 107 and numbered equations. You will find that most equations are not numbered, or are numbered symbolically. However sometimes I want to refer back to an equation, and in that case it is numbered within the section. Thus Equation (1.1) refers to the first numbered equation in Section 1 and so on.

The Web Version

These notes have evolved slowly and so many errors have been eliminated. However they continue to change and grow as we find ways to make them more useful. This means that there will be new errors, even though I have tried hard to eliminate them. A printed version is extremely convenient to use, but has the disadvantage that updating is expensive and inconvenient. In contrast the web is a very recent, but already almost universal medium, which offers convenient and very rapid updating. To take advantage of this these notes are also available as a set of linked file on the web, written in HTML (Hypertext Markup Language), the native language of the web. Since I can update these files (over 750 when I last counted) easily, this format will always have the most recent version of the notes. They are available at

`http://www.maths.abdn.ac.uk/~igc/tch/mx4002/notes/notes.html`

They can be reached from the Departmental web page by going through the *private* home page of Ian Craw.

At present they are in a form suitable for relatively simple browsers. However it is essential to use a graphics-based one, because each piece of mathematics has to be delivered as a separate image. In the future browsers may be able to cope with mathematics, although I think this unlikely to happen quickly. Typically you will use NETSCAPE, wherever it is available, to read the html version of the notes. The conversion from printed notes to html is automatic, and has some imperfections; I expect these to reduce as the technology matures, but you should be aware of the problem.

A pdf (portable document format) version is availablefrom the Web. The file can then be viewed using Adobe's freely available Acrobat reader, or may well display directly in your browser. In either case, the whole document, or selected pages, can then be printed. This is likely to give much better quality output than printing directly from the web, since the pdf version is based on the original Postscript rather than the derived HTML.

Putting these notes on the web has exposed them to a much wider audience. I thank Sean Sedwards and Dr. Steve Kelem for letting me know of mistakes even though they had nothing to do with the course! Many others commented on problems with the images, which aroise because some of the diagrams were drawn in FrameMaker. I have now redrawn the diagrams, either in xfig or using the `pstricks` package, and these problems should now be in the past.

The MX4028 Mailing List

There is a mailing list associated with this class. You can subscribe to it by sending email to `majordomo@maths.abdn.ac.uk` with a message that has an empty "subject" field and contains the single line `subscribe mx4028`. If you add your signature automatically, you can also add the line `end` after the subscribe request, and the signature will be ignored. You are encouraged to join this list. You then mail `mx4028@maths.abdn.ac.uk` to contribute to the list.

I have always been happy to deal with questions by email, and have made a point of publishing my email address both on the web and in the printed notes. This list provides a more general way of communicating in which both questions and their answers go to everyone on the list. Here are some reasons why this might be useful:

- It is rare for just one person in a class to have a given problem; by sending the answer to this list, others can benefit from it.
- When a topic causes general concern, the lectures can be changed to cover it in greater detail, or pick up the problem area.
- Often other members of the class can be of more help than the lecturer; this way everyone is aware of the problems and is invited to help or comment.
- I have always been careful to make statements about the examination in front of the whole class – the list provides a useful public forum.

Please note that this list is being maintained on the mathematics machines rather than the central University ones, so you need to mail `maths.abdn.ac.uk` to reach it.

Finally some points of netiquette.

- Please remember the usual courtesies; although this is email, it is still fairly public.
- If you send email directly to me, I will not copy it to this list without first getting your permission.
- The list is low security. Although you are technically able to impersonate someone else, please don't do so.
- Keep a copy of the message you get when you join the list to remind you how to leave; you are welcome to leave and re-join as often as you wish.

Sometimes it is useful to pass a message to the whole class. I believe that for most people, this list is more useful way than posting it on a notice board. One natural use would be to cancel a lecture if that has to be done unexpectedly. The message may not reach everyone, but those who read it will be saved the inconvenience of turning up.

Any more questions? Why not mail the list? You won't be the only one with them.

Background Reading

There is an enormous literature on this subject, although much of it has a different end in view from the one we take in this course. The aim here is to discuss appropriate algorithms, emphasising the underlying structures and the ability to assess algorithms. This is not a programming course, and so there is no explicit aim to cover the implementation details, which are needed to convert an algorithm into an actual working computer program. At times such details will influence the more abstract approach, but even in this case, our considerations will be general, and not tied to a particular language.

Many of the books on the subject are concerned with both the theory and implementation of algorithms. The two books which are closest to the spirit of this course both take this approach. One, (v Aho, Hopcroft & Ullamn 1983), implements in Pascal, while the other (Sedgwick 1995), is a C++ version of a book that originally implemented in Pascal. This is typical of many of he books you will find in the library; if it is well written, the implementation details should be easily omitted. A more advanced book (Knuth 1981), the "bible" of the subject for many, is much more concerned with the fundamentals.

Ian Craw
Department of Mathematical Sciences
Room 344, Meston Building
email: `Ian.Craw@maths.abdn.ac.uk`
www: http://www.maths.abdn.ac.uk/~igc
October 25, 1999

Chapter 1

Introduction

1.1 Algorithms; Computer languages

This course is going to study *algorithms*. Before we start to study them it would help if we knew what they were. A full and rigorous definition of what is meant by an algorithm is complicated and may be confusing at this stage. However, all the essential ideas about algorithms can be encapsulated in the single statement that an algorithm for solving a problem is a technique for solving that problem that can be programmed for a computer.

In rather more detail, an Algorithm is a finite and definite procedure for solving a problem. The finiteness is important; we cannot accept as algorithms methods which, for example, involve us in taking limits unless we have available a finite procedure for evaluating those limits. The definiteness is also important. We cannot accept as algorithms methods which involve making inspired guesses, like finding a clever substitution for an integral.

You may already have met the word *algorithm* in the context of Euclid's Algorithm for the calculation of the highest common factor of two integers. This is indeed an algorithm in our sense of the word. Let me write it down in the form of a procedure, using the kind of semi-formal language that we will normally adopt for describing algorithms.

```
algorithm  hcf(n,m)   // calculate hcf of positive integers n,m
begin
  repeat begin
    l = remainder on dividing n by m.
    n = m
    m = l
  until m = 0  // required hcf is n.
end
```

You should have no difficulty in translating this into a program in whatever computer language you happen to use. It can be proved to be *finite* — each operation in the procedure is finite and we only go round the loop a finite number of times (these things have to be proved, of course). It is manifestly *definite*, no guesses or choices are being made.

On the other hand, the problem of finding the decimal representation of $\sqrt{2}$ cannot be solved by an algorithm, so far as we know, because there are infinitely many decimal digits to be found and they cannot be described 'finitely' because they neither repeat nor follow a clear pattern. The problem of finding a specified finite number of the decimal digits of $\sqrt{2}$ *is* an algorithmic problem.

What is involved in the study of algorithms? Firstly, of course, there is the problem of actually finding algorithms to solve important problems. In addition there is the problem of evaluating those algorithms. Are they good or bad? Could we expect to do better by studying the problem further?

The usual measure of quality of an algorithm is its *speed* — does it do its job efficiently? Other significant criteria are: does it require a lot of computer space, is it easy to generalise, is it simple and logical and consequently easy to program without error?

Euclid's Algorithm is an example of a very good algorithm. It is undoubtedly a simple algorithm. It does not require much adjustment to cover all integers rather than just positive ones. It turns out that it is almost the quickest known way to solve the problem. Compare it, for example, with the equally correct method which starts with the lesser of n and m and successively reduces it by 1 until it divides both n and m. It would be a good first exercise for you to program these two methods and then compare their running times when required to find the h.c.f. of two six digit numbers.

1.1.1 Programming

This is probably the place to talk about programming, and the languages involved. Everything I do can be programmed in FORTRAN, Maple or in any other language that you know, and you will benefit by actually trying the algorithms out on a computer. It is worthwhile developing enough facility to try the last example quickly to get a feel for the times taken by various algorithms. However, this is *not* a programming course, and you should not spend too much time doing this. Unfortunately, although it is very good for numerical work, FORTRAN is by no means the most suitable language with which to investigate algorithms, both because it is hard to write it both legally and clearly, and also because it lacks certain facilities. To make the first point, note that the language used to describe the Euclidean Algorithm above is *not* FORTRAN. It is in fact a kind of pidgin C or Pascal. I hope you will find this *pseudocode* easy to use on paper without a formal description of what is acceptable; the idea is that anything that communicates is OK!

To see FORTRAN's lack of facilities, consider the following equivalent description of the Euclidean Algorithm.

```
algorithm hcf(n,m) // sets result to hcf of positive integers n,m
begin
  if (m = 0)
    return(n)
  else
    return(hcf(m, n mod m))
  endif
end
```

Arguably it is easier to see why this version of the algorithm is valid. The first statement is clear; if one of the numbers is zero, then we take the hcf to be the other one. The rest of the algorithm is the statement that the hcf of two numbers is the same as the hcf of the smaller, and of the remainder when the larger is divided by the smaller.

1.1.2 An Implementation of the Euclidean Algorithm

An alternative way to see how this, and subsequent algorithms behave is to try out an implementation written to show the steps involved. By trying it with different inputs, you can get an idea of efficiency, and also check your understanding of what happens with unusual input. An implementation of this algorithm in the *Java* language is available in the hypertext version of this document, enabling you to do this. I hope you can get access to a working version, and appreciate having code working right in the notes!

1.1.3 Recursion

This type of "divide and conquer" algorithm is very common in what follows; one way to solve a problem is to show how to reduce it to a smaller problem of the same (or a closely related) type. If that smaller problem itself can be reduced, then we can continue in this way. Ideally we end up only having to solve a very trivial version of the problem. A familiar model for this should be the evaluation of integrals using reduction formulae, for example $I_n = \int \sin^n x$. Using integration by parts, one expresses I_n in terms of I_{n-2}, and then only has to evaluate I_1 and I_0.

In this context, the method is known as *recursion*, because the function recursively calls itself. The method is not necessarily very efficient (as measured by run times) but it *is* very good for showing the logical structure of the program. The language FORTRAN does not support recursion, so only the first form of the algorithm is acceptable to it. In contrast, languages like Pascal, POP11, C etc are quite happy with recursions even though they have to unwind the recursion to perform the actual evaluation.

Another lack in FORTRAN is any built-in sophisticated form of structure, other than the double precision number, and the array. Of course, this sparseness is one reason why the language is attractive. However, we shall see later that writing effective algorithms, and generating new data structures are very closely related, and that, at least during explorations, it can be valuable to have non-numeric structures available. In particular, it can be helpful to have a "list" as a built in data type. For this reason I will also use POP11, as above to give examples. Again there should be no difficulties; if you don't speak POP11, just read it.

1.1.4 Algorithm Types

In this course I am not going to attempt to describe all known algorithms — that would require an encyclopedia and would also be futile. The important end result of a study of algorithms is that you should be able to write and evaluate your own algorithms for the problems that confront you, and which are probably not in the books anyway. To that end I am going to look at a small number of typical methods from a range of different contexts, so as to show you the type of thinking that is involved. These may then give you the basic hints as to how to proceed with your problems. An analogy with the integral calculus might make this point just as well. No sane calculus book will attempt to list all possible substitutions that work in evaluating integrals. You will be shown some substitutions that are famously effective in certain types of problems and you will be shown enough general examples to give you the general idea of how to proceed in other cases. Then you are left to your own devices.

However, it is worthwhile to give an indication of the range of applications which are usually covered. These include

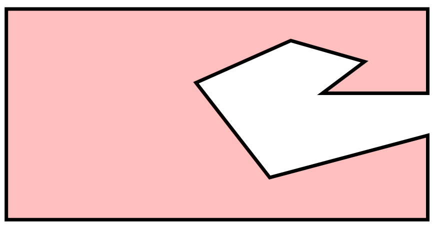
- sorting algorithms;
- searching algorithms;
- string processing and language;
- geometric algorithms;
- graph algorithms; and
- mathematical algorithms.

1.2 An Algorithm — Hatching Polygons

Our first example was clearly an algorithm; it was called an algorithm! We now illustrate the concept with algorithms which have a more geometric flavour.

1.2.1 The Problem

Suppose we have a *polygon* in the plane defined by a sequence of vertices $v_1, v_2, \ldots, v_n$. The polygon can be arbitrarily complicated but we will assume that it is not *re-entrant*; that is to say, no two edges of the polygon cross. Fig 1.1 illustrates this.



but not

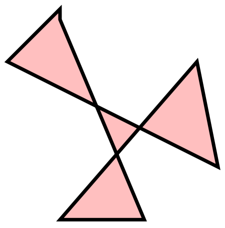


Figure 1.1: A polygon, and a re-entrant polygon.

With this qualification, our polygon will define a region — the *inside* of the polygon (not so clear in the re-entrant case). Our basic problem is going to be to *shade in* this region. To be precise, and to avoid computer technicalities, we want to *hatch* the region with a set of equally spaced straight lines, as shown in Fig 1.2.

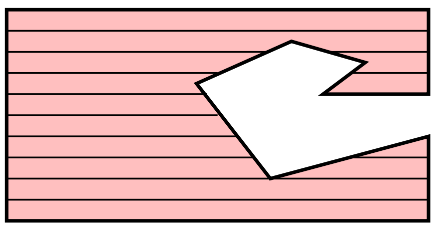
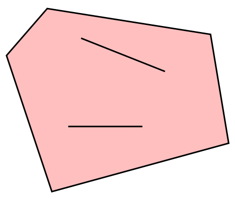


Figure 1.2: Hatching a polygon with horizontal lines.

To keep things simple we will assume that the hatching lines are to be horizontal (constant y), though the generalisation to sloping lines is very easy. We will assume that we have available the single computer-graphics instruction:

`draw a line from` (x_1, y_1) `to` (x_2, y_2)

A polygon is *convex* if it is not re-entrant and if the join of any two points in the region lies entirely within the region. Examples of a convex and a non-convex polygon are shown in Fig. 1.3; the non-convex one has an example line drawn with both end points in the polygon, but which does not remain completely within the polygon.



but not

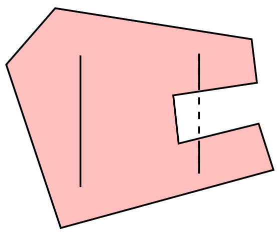


Figure 1.3: A convex polygon, and a non-convex polygon.

There is obviously a difference in difficulty between the problem of hatching a *convex* polygon and that of hatching a general one. We will start by solving the general case and then try to see whether we can simplify, and speed up, the method if we know that the polygon is convex.

1.2.2 General Method

The method is based on the following observation. Suppose we have any polygonal (non-reentrant) region in the plane. Then, special cases excepted, any line l either misses the region completely or else meets the polygon in an even number of points which, in order, are *into, out of, into, out of,. . . , into, out of* the region. (The special cases come when an edge lies on the line or when two intersection points degenerate into a single intersection at a vertex.)

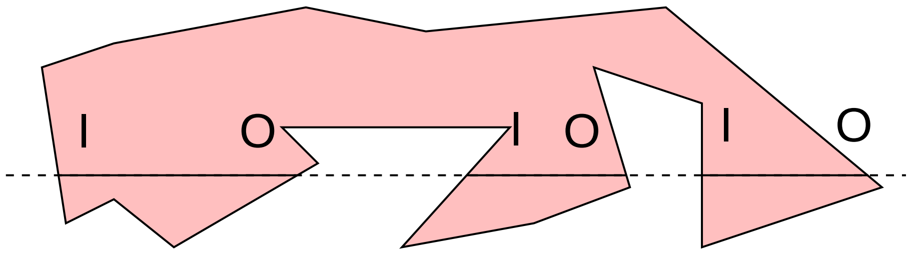


Figure 1.4: A single hatching line passing in and out.

If we are using the line l to hatch the region then we will want to join the first point to the second one, the third to the fourth, the fifth to the sixth and so on. With this observation we can now solve the hatching problem. The outline of the procedure is this:

```
for each hatching line:
  Go through the edges of the polygon in sequence,
    find out which ones cut the hatching line
  For each edge that does, store the intersection points in a list
  Sort the list of intersection points into order to give
    $(p_1, p_2, p_3, p_4, \ldots, p_{2n-1}, p_{2n})$
  Finally, draw in the lines
      $(p_1, p_2),\ (p_3, p_4),\ \ldots\ (p_{2n-1}, p_{2n})$
```

Like most such outlines this still leaves us with lots of work to do, but at least the remaining problems are more standard ones:

1. how do we describe the hatching lines;
2. how do we decide whether or not a hatching line meets a given edge; and
3. how do we sort the intersection points into order?

The first question is reasonable easy if the hatching is horizontal. The hatching lines are of the form $y = h$. To find the range of values of h we firstly find the lowest and highest vertices of the polygon, and let the corresponding values of h be `hmax` and `hmin`. The situation is shown in Fig. 1.5.

This involves running through the n vertices and picking out the min and max values of their y-coordinates. We then decide on the spacing between the lines.

Having done this our program will become

```
algorithm hatch // hatch a polygon with horizontal lines
begin
  h = hmin
  repeat begin
    do job for y = h
    h = h + dh
```

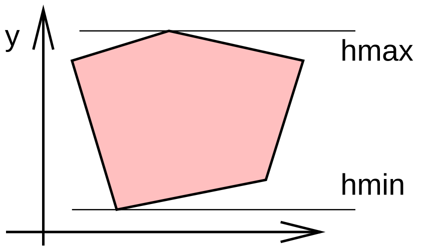


Figure 1.5: The range for h.

```
    until h >= hmax.
  end
```

For the second question, horizontal hatch lines make life very easy. The edge joining v_i to v_{i+1} will *fail* to meet the line $y = h$ if *either* both the y-coordinates of the vertices are less than h *or* both are greater than h. Algebraically, we could express the intersection condition as

$$(y_{i+1} - h)(y_i - h) \leq 0$$

If we have decided that an edge does meet the hatch line then we will want to find the intersection point (its x-coordinate will be a suitable number to store as a parameter along the hatch line). This is now just good old-fashioned coordinate geometry.

The equation of the line joining (x_i, y_i) to (x_j, y_j) can be written as

$$(x_j - x_i)(y - y_i) = (y_j - y_i)(x - x_i)$$

We want x when $y = h$. If $y_i \neq y_j$ then we have

$$x = x_i + \left(\frac{x_j - x_i}{y_j - y_i}\right)(h - y_i)$$

If $y_i = y_j$ then we have a problem. It is best to ignore the intersection.

We will consider the third question in detail later in the course.

1.2.3 Hatching Convex Polygons

If we happen to know that the polygon we are dealing with is convex then the above general algorithm for shading is very inefficient. The problem can be solved much more efficiently in this case. The idea, which avoids the sorting step above, is shown in Fig. 1.6.

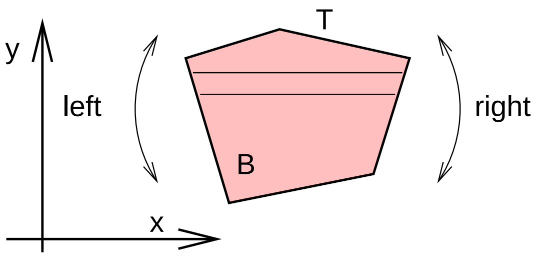


Figure 1.6: Hatching a convex polygon.

For the sake of variety let's start at the top this time. The advantage in this case is that we do not need to go searching through all the edges all the time. At any particular stage we are dealing with one ‘left’ edge and one ‘right’ edge. Whenever the horizontal level drops below the bottom end of one of these edges we move on to the next edge on that side. The starting and ending points of the hatches are calculated from the equations of the edges. The procedure stops when the h value drops below that of B.

1.3 The Analysis of an Algorithm

In this section we will study one simple algorithm in fairly full detail. This should clarify some of the basic ideas, and illustrate the type of information we are trying to extract in more general situations. The problem is “trivial”, but even so, the calculations to which we are led, are not completely trivial.

Algorithms are methods for solving problems, so we had better start by stating the problem:

Given a list$(k_1, k_2, \ldots, k_n)$ of numbers, find the smallest one.

This is hardly a profound problem and there is no difficulty in finding an algorithm to solve it. The following procedure works:

```
algorithm min(x,n) // to find the minimum of x =(x(1),...,x(n))
begin
  min = x(1)
  for i = 2 to n begin
    if ( x(i)< min )
      min = x(i)
  end
  return (min)
end
```

In words, we keep our “minimum element so far” in `min`. We put it equal to `x(1)` at the start and then move along the list, updating `min` whenever we come to an element that is smaller than it. When we get to the end of the list `min` will contain the smallest element. Strictly speaking we should now present a *proof* of the algorithm, but I think that we can get away with the word ‘trivial’ in this case.

Now that we have an algorithm for solving the problem the next job is to evaluate it. Such evaluations are normally competitive — there are a number of possible algorithms for solving a problem and we have to decide which is best. In this case we just have the one method and it is difficult to think of a significantly different way to solve the problem. Nevertheless we will proceed with our evaluation.

1.3.1 Timing

Our attention is going to focus on the *timing* of the algorithm. How long does it take to do the job? The obvious answer to this question is: it depends on which computer you are using. Certainly, but that does not make the question a silly one. Most present-day computers only vary in the speed with which they can perform elementary operations (machine instructions). If we know how many basic operations the method requires then we know how fast it will run on any particular machine. We will not go down to quite this level of detail because machines also differ somewhat in the nature of their elementary operations but we will always attempt to break our algorithms down into very simple steps.

Let me now rewrite the above algorithm in the form of a flow-chart using very basic operations (see Fig. 1.7). We will pretend that each of these operations takes the same time to execute on a computer. This is not precisely true but neither is it wildly false.

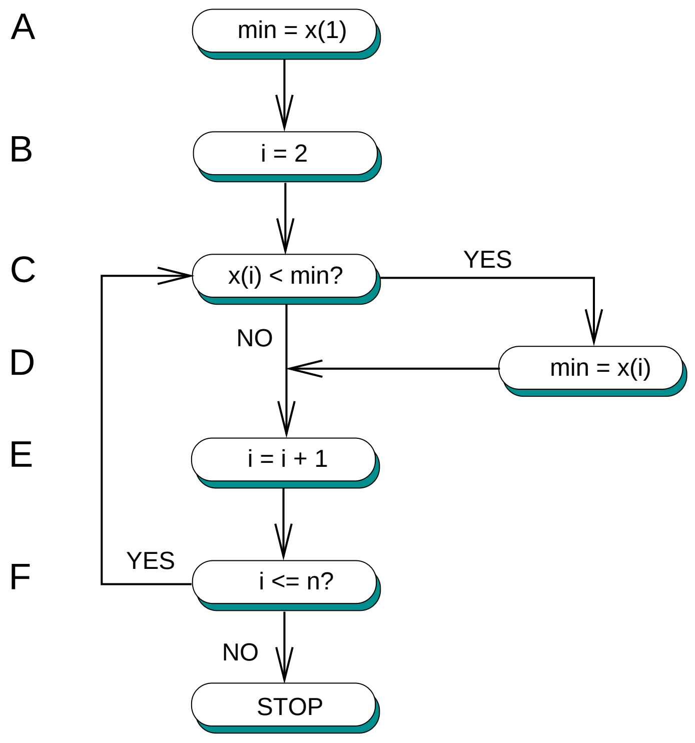


Figure 1.7: Flow chart to find the minimum element in a list.

Now we can start timing the algorithm. We just want to count how many times each box is executed. Boxes A and B are only executed once. Boxes C, E and F are on a loop that the program goes round $n-1$ times, so each of these is executed $n-1$ times. That leaves box D. This is where all our problems come.

Consider the list (4,5,3,6,1,2). As the algorithm scans along this it will start with `min=4`. It will then do a 'detour' on the third element to make `min=3` and it will make a final detour on the fifth element to give `min` its final value of 1. In all the box D gets executed twice.

On the other hand, if we apply the algorithm to the list (1,4,3,6,2,5) the detour box D will never be executed. `min` will get its correct value right at the start and will never need to be updated.

This is our problem. There is no fixed value for the number of detours. It depends entirely on the list being processed. We can make three obvious statements about the value d of the number of detours:

- First, the minimum value of d is zero. This will occur whenever the list has its smallest element in first place.
- Second, the maximum value of d for a list of n elements is $n-1$. This will occur when the list is in strictly decreasing order, so that the current value of the minimum has to be updated on each step.
- Thirdly, for lists of n elements d can take any value between these extremes (exercise).

These comments allow us to make the following statement. Suppose each box in the flow chart takes unit time to execute. Then the time $T(n)$ taken by the algorithm to find the minimum of n numbers lies in the range

$$3(n-1)+2 \leq T(n) \leq 3(n-1)+2+(n-1)$$

or

$$3n - 1 \leq T(n) \leq 4n - 2$$

For many purposes this might be all that you need to know. The timing is roughly proportional to the number of elements in the list, allowing for some variation due to the different number of detours made for different lists.

1.3.2 A More Precise Analysis

Nevertheless, it would be nice to have some more precise information about the value of d. That is the problem that we are going to study next.

We know that d varies from one list to another. What we really want to know is its general behaviour. In particular, it would be nice to know the *average value* of d for lists of length n. The usual difficulty with averages is that we have to know what we are averaging over and how we are taking that average. That seems simple enough in this case. We are, surely, averaging over all lists of length n. Perhaps so, but in that case we also need to know that probability of any particular list being chosen, so that we know how to weight the average. That's more difficult. It would seem silly to claim that all lists of n numbers are equally likely to be presented to the algorithm. Computers rarely deal with numbers of the order of 10^{10000}.

We will instead adopt a rather different approach. This will be used again in later chapters. Notice that as far as the algorithm is concerned the actual numbers in the list are irrelevant. All that matters is their relative ordering. What I mean by this is that the algorithm takes the same time to process the list $(3000, 2000, 1000, 4000)$ as it does to process the lists $(3, 2, 1, 4)$ or $(300, 22, 9, 1000)$. All that matters is that in each case the first element is the third smallest, the third element is the smallest and so on. [1]

Because of this observation we can make a critical simplification. Let us assume that we are dealing with lists of n distinct elements (I will leave consideration of lists with repeated elements to you). Then we may as well assume that the lists that we are dealing with are all the permutations of the list $(1, 2, 3, 4, \ldots, n)$. There is nothing limiting about this assumption. Let us now go further and make another assumption, this time a genuine one. Let us assume that *all* these permutations are *equally likely* to be presented to the algorithm. This may not be a reasonable assumption in some situations. It may be the case that in a certain application the vast majority of lists supplied to the algorithm are only slightly shuffled and as such do not represent a random sample from the set of all possible permutations. In that case the following analysis would not be correct. Our assumption is really a neutral assumption made in the absence of other information (and happens to make the mathematics easier, which is a very significant point!). So now we have a mathematically precise problem to solve.

> What is the average number of detours executed when applying the algorithm to the permutations of $(1, 2, 3, \ldots, n)$, granted that all such permutations are equally probable?

1.3.3 Computing detours: A_k^n

Let S_n denote the set of all permutations of $(1, 2, 3, \ldots, n)$. Recall that S_n has $n!$ elements. Let A_k^n be the number of permutations in S_n that require k detours when applying the algorithm. Then the average number of detours, denoted by $\mu(n)$ is given by

$$\mu(n) = \frac{1}{n!} \sum_k k A_k^n$$

Consider first the case when $n = 3$. We can list all the permutations in S_3 and work out the number of detours for each one.

[1] I am making the possibly doubtful assumption that the computer handles all numbers equally quickly. This may not be true in general but consideration of such problems would take us too far afield.

π	d
$(1,2,3)$	0
$(1,3,2)$	0
$(2,1,3)$	1
$(2,3,1)$	1
$(3,1,2)$	1
$(3,2,1)$	2

Table 1.1: Number of detours sorting elements of S_3.

In the above notation we have $A_0^3 = 2$, $A_1^3 = 3$ and $A_2^3 = 1$, giving a total of $3! = 6$. The average number of detours is

$$\mu(3) = \frac{0+0+1+1+1+2}{6} = \frac{0.A_0^3 + 1.A_1^3 + 2.A_2^3}{6} = \frac{5}{6}.$$

You should check for yourself in the same way, that $\mu(2) = \frac{1}{2}$ and, with a bit more effort, that $\mu(4) = \frac{13}{12}$.

Returning to the general case, note that we have already decided the following claims:

$$\begin{aligned} A_k^n &= 0 \quad \text{if} \quad k < 0 \quad \text{or} \quad k \geq n, \\ A_0^n &= (n-1)! \quad \text{(all permutations starting with a 1)}, \\ A_{n-1}^n &= 1 \quad \text{(only the permutation} \quad (n, n-1, \dots, 3, 2, 1)), \\ A_0^2 &= 1 \qquad A_1^2 = 1. \end{aligned}$$

Now consider the following argument. Divide S_n up into n subsets. The first, L_1, consisting of those that end with a 1, the second, L_2, consisting of those that end with a 2 and so on up to the last subset, L_n, consisting of those lists that end with an n. We now try to count how many of the elements of each of these subsets require k detours.

First consider L_1. In this set we know for certain that the last detour is made on the very last element, because 1 is the smallest element. So $k-1$ detours must have been made in the previous $n-1$ elements. Now the previous elements make up all permutations of $(2, 3, \dots, n)$, each one represented just once. So by our earlier argument about the irrelevance of the *names* of the elements, the number of permutations in L_1 that require k detours is A_{k-1}^{n-1}.

Now consider L_2. In this case we know for certain that we *do not* do a detour on the last element, because the 1 has already passed. So all k detours must occur in the first $n-1$ elements. By the same argument as in the previous case this means that the number of permutations in L_2 that require k detours is A_k^{n-1}.

The same argument applies to the other subsets $L_3, \dots, L_n$. In each case all k detours have to be made before we come to the last element because the last element is not 1.

Gathering all this information together we see that we obtain the following *recurrence relation* for A_k^n:

$$A_k^n = (n-1)A_k^{n-1} + A_{k-1}^{n-1} \qquad \text{for } n > 1,\ k > 0.$$

That was a typical combinatorial argument and it may very well leave you a bit confused. The best thing to do in that case is to work through the argument in a particular case, for example $n = 4$. You may then realise that behind the confusion lies a simple argument.

We are now in a position to work out lots of values of A_k^n by using this recurrence relation in conjunction with the initial and boundary values calculated before. Some values which can be obtained fairly easily are given in Table 1.2. For example, $A_3^6 = 5A_3^5 + A_2^5 = 5 \times 10 + 35 = 85$.

n						
1	1					
2	1	1				
3	2	3	1			
4	6	11	6	1		
5	24	50	35	10	1	
6	120	274	225	85	15	1

Table 1.2: Values of A_k^n for small values of n and k.

We can obviously carry this on as long as we wish, but it does not provide us with anything like a general formula for A_k^n. Compare this situation with that of the binomial coefficients. There we have a recurrence relation which is superficially very similar to the one for the A's:

$$\boxed{C_k^n = C_k^{n-1} + C_{k-1}^{n-1}}$$

from which we build up Pascal's triangle. But we also have the explicit formula

$$C_k^n = \frac{n!}{k!\,(n-k)!}$$

(allowing that we accept the factorials as 'explicit'). We have no such explicit formula for the A's.

Does the fact that we have not actually got an explicit formula for the A's destroy any hope of calculating the value of $\mu(n)$? As it happens it does not, though this is not yet obvious. What we will do next is to use the recurrence for the A's to obtain a corresponding recurrence relation for the μ's. The difference will turn out to be that the recurrence relation for the μ's is explicitly solvable. That is luck rather than judgment.

Take the definition of $\mu(n)$:

$$\mu(n) = \frac{1}{n!}\sum_k kA_k^n$$

and substitute A_k^n by the value given by its recurrence relation.

$$\mu(n) = \frac{1}{n!}\sum_k k((n-1)A_k^{n-1} + A_{k-1}^{n-1})$$

Now start to manipulate the right-hand side:

$$\begin{aligned}
\mu(n) &= \frac{n-1}{n!}\sum_k kA_k^{n-1} + \frac{1}{n!}\sum_k kA_{k-1}^{n-1}, \\
&= \frac{n-1}{n!}(n-1)!\mu(n-1) + \frac{1}{n!}\sum_k (k-1)A_{k-1}^{n-1} + \frac{1}{n!}\sum_k A_{k-1}^{n-1}, \\
&= \frac{n-1}{n}\mu(n-1) + \frac{(n-1)!}{n!}\mu(n-1) + \frac{(n-1)!}{n!}, \\
&= \mu(n-1) + \frac{1}{n}.
\end{aligned}$$

So we have obtained the recurrence relation

$$\boxed{\mu(n) = \mu(n-1) + \frac{1}{n}.}$$

Before going on to study this, perhaps I had better explain what was happening in the above chain of manipulations. The aim of the exercise was to replace all the A's by μ's, somehow or other. Nothing much happens in the first line. In the second line we note that the first sum is just the sum that would occur in the definition of $\mu(n-1)$, so we substitute appropriately, remembering the factorial. We also do a further, less obvious, bit of rearrangement. Basically, we change the kA_{k-1}^{n-1} term into $(k-1)A_{k-1}^{n-1} + A_{k-1}^{n-1}$. This will then allow us to interpret the first sum as part of the definition of $\mu(n-1)$ — remember that k is a dummy variable and it doesn't matter in the least that we suddenly choose to call it $k-1$. Finally, in the third line, recall that the sum of all the A's must equal the total number of permutations.

Now let's study the recurrence that we have obtained. We already know that $\mu(2) = 1/2$. So, using the recurrence,

$$\begin{aligned}\mu(3) &= \frac{1}{2} + \frac{1}{3},\\ \mu(4) &= \frac{1}{2} + \frac{1}{3} + \frac{1}{4},\\ \mu(5) &= \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5},\end{aligned}$$

and so on. It is now obvious that

$$\mu(n) = \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}.$$

Our conclusion is that the average number of detours made in applying the "minimum element" algorithm to the permutations of $(1, 2, \ldots, n)$, assuming that all are equally likely to occur, is

$$\boxed{\mu(n) = H_n - 1}$$

where H_n is the n^{th} *Harmonic Number* defined by

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k}.$$

1.3.4 Approximating H_n

Does this help? Only if we know something about H_n. This leads us into another important aspect of the study of algorithms, that of asymptotic estimation. Put in simple terms, can we find a simple approximate formula for $\mu(n)$ to replace the complicated exact formula that we have obtained above? In this case we can, and the result is quite famous.

Consider the diagram in Fig 1.8. By adding up the areas of the rectangles that lie *above* the graph we get

$$1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-1} > \int_1^n \frac{dx}{x} = \ln n.$$

Similarly, by adding up the areas of the rectangles that lie *below* the graph we get

$$\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} < \int_1^n \frac{dx}{x} = \ln n.$$

From these we easily deduce that

$$\ln n + \frac{1}{n} < H_n < \ln n + 1$$

This estimate can be improved greatly, but is already adequate for our purposes. It says that H_n is close to $\ln n$, and that the percentage error decreases to zero as $n \to \infty$.

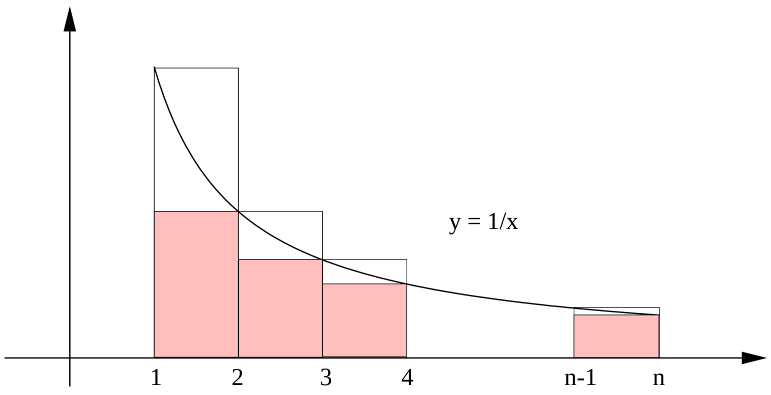


Figure 1.8: Graph of $y = 1/x$.

If you are interested, a better estimate is

$$H_n = \ln n + \gamma + \frac{1}{2n} + \cdots \qquad \text{where} \quad \gamma = 0.5772156649 \quad \text{is Euler's Constant.}$$

A proof, together with much more information is available in (Graham, Knuth & Patashnik 1988, Page 264). Don't take the title of this book at its face value: "Concrete" Mathematics is a blend of *Con*tinuous and dis*crete* mathematics, chosen to be of interest and relevance to this type of calculation!

Now we can go back to our study of the timing of our algorithm. Recall that we obtained a formula for the average running time of the algorithm which could now be written as

$$\begin{aligned} T(n) &= 3(n-1) + 2 + \mu(n), \\ &= 3n + H_n - 2, \\ &\approx 3n + \ln n. \end{aligned}$$

Where the approximation is meant to be applied if n is quite large. This is interesting. Let me put some values down in a table:

n	$3n$	$\ln n$
10	30	2.3
100	300	4.6
1000	3000	6.9
10000	30000	9.2

Table 1.3: The growth of $T(n)$.

So you see that, on average, there are really very few detours indeed. For a value of n like 10000 the detours only make up, on average, 0.03% of the time taken by the algorithm.

This shows a phenomenon that you need to get clear in your mind at an early stage: the logarithm function grows very slowly indeed.

1.3.5 Conclusion

The algorithm that we have produced for finding the least element of a list of numbers runs in time which is, on average, almost exactly proportional to the number of terms in the list.

This assertion was made on the assumption that the lists are lists of distinct elements and that all orders of elements in those lists are equally likely to occur.

Questions 1 (Hints and solutions start on page 86.)

Q 1.1. Sometimes it is fairly easy to think of a *better* algorithm; sometimes it can seem unlikely that one exists until a better one is demonstrated. How can you compute x^{55} using only 9 multiplications? How few multiplications do you need to calculate a polynomial of degree n?

How few multiplications do you need to compute the product of two $n \times n$ matrices? After some thought, try looking in the books for "Strassen's method" which dates from 1968!

Q 1.2. Look carefully through the procedure for hatching a polygon and try to pick out the 'special cases' that might occur and which might require modifications to the program to cope with them.

Q 1.3. What result does the procedure produce if presented with a re-entrant polygon like that in the first figure?

Q 1.4. How should the program be modified if we want all our hatch lines to be at angle α to the horizontal?

Q 1.5. A non-reentrant polygon has vertices $v_1, v_2, \ldots, v_n$. You are given a point (x, y). Find a reasonably quick way to decide whether or not this point is in the region defined by the polygon.

Q 1.6. You are given a list $v_1, v_2, \ldots, v_n$ of vertices in the plane. How would you set about deciding whether they defined a *convex* polygon?

Q 1.7. What might you do if you wanted to shade an annular region, like the gap between two squares, one inside the other?

Q 1.8. Change the minimum element algorithm so that it finds the maximum element instead. Convince yourself that the timing for this algorithm is the same as it was for the minimum element algorithm.

Q 1.9. Write an algorithm to find the two smallest elements in a list of numbers (the smallest and the second smallest). You should be able to make it a lot more efficient than finding the minimum twice over.

Q 1.10. How could the minimum element algorithm be speeded up if you knew in advance that all the elements of the list come from a finite set of size much smaller than the length of the list — for example, if the elements were all single digits? Assume that you know what set you are dealing with.

If the lists are random lists from this set estimate the timing of your algorithm.

Q 1.11. For once in your life, write down all the permutations of $(1, 2, 3, 4)$. Count the number of detours for each permutation and check that the average comes out as $13/12$.

Q 1.12. Use the recurrence relation for the A^n_k to add the next two rows to my table of values. Can you give an explicit formula for A^n_{n-2}?

Q 1.13. Refer to the picture of the graph of $y = 1/x$ earlier in the chapter. Notice that, because of the concavity of this graph, the diagonals of the small rectangles in the picture lie above the graph. Use this to obtain an improved estimate for H_n.

Q 1.14. Write an efficient algorithm, using a pseudocode of your choice, to find the minimum element of a given list of integers.

The algorithm is to be run on an abstract computer on which each basic operation; assignment, comparison between two numbers or incrementing a counter, takes one unit of time. Show that the time $T(n)$ for the algorithm to run on a list of length n satisfies $T(n) = O(n)$.

The timing of one such algorithm is known to be

$$T(n) = 3n - 1 + \mu(n)$$

where μ satisfies $\mu(n) = \mu(n-1) + \frac{1}{n}$ and $\mu(1) = 0$. Obtain an expression for $\mu(n)$ as a series. By comparing with the area under a suitable curve, show that $\mu(n)$ satisfies the inequality

$$\ln n + \frac{1}{n} - 1 < \mu(n) < \ln n.$$

and thus obtain estimates for $T(100)$.

Q 1.15. Write an efficient algorithm, using a pseudocode of your choice, to find the minimum element (`min`) and the second smallest element (`second`) of a given list of integers.

The algorithm is to be run on an abstract computer on which each basic operation (assignment, comparison between two numbers or incrementing a counter) takes one unit of time. Show that the time $T(n)$ for the algorithm to run on a list of length n satisfies $T(n) = O(n)$.

Explain what assumptions you would make to get the expected value of this running time. Illustrate your answer by doing explicit calculations in the case where $n = 4$. [You are *not* required to compute the expected running time in general.]

Chapter 2

Sorting

The general problem faced in this topic is to put into some kind of order a list of items for which such an ordering is defined. At its simplest this might mean putting a list of numbers into ascending order. More generally it includes doing things like putting lists of words into alphabetical order.

I will be assuming in this section that the *ordering* relation that we are using is what is known as a *linear ordering*. Let me be a little bit formal about this. A *partially ordered set* $(P, <)$ is a set P together with a relation $<$ which satisfies the conditions:

- if $x < y$ then it is not true that $y < x$.
- (in consequence) it is not true that $x < x$.
- if $x < y$ and $y < z$ then $x < z$.

A typical example might be to have P as the set of all citizens of the UK with $x < y$ if x is descended from y.

A partially ordered set is called a *linearly ordered set* if, in addition, we have the condition

- for any $x \neq y$ either $x < y$ or $y < x$.

This is obviously not true for the previous example. It is true for things like the integers or reals ordered in the usual way. It is also true for the set of words ordered lexicographically. Most sorting techniques are designed primarily to deal with lists taken from linearly ordered sets. Apart from that all that we really need to assume is that we have some algorithm for deciding whether $x < y$ or $x = y$ or $y < x$.

2.1 Complications

I will stick to simple problems but let me at least indicate some of the difficulties that can arise in real life.

The first problem is the size of the list to be ordered. I will write my algorithms using a list $\{x_n\}$ to represent the data. If you turn the algorithm into a computer program you will probably want to turn this list into something like a FORTRAN array. This may not be possible if the list is too large for the computer's memory. You may only be able to hold part of the data inside the computer at any time, the rest will probably have to stay on disc or tape. This is known as the problem of external sorting. I may say something about it later.

Another problem is the *stability* of the sorting method. Let me explain this by an example. Suppose you are an airline. You have a list of the passengers for the day's flights. Associated to each passenger is the number of his/her flight. You will probably want to sort the list into alphabetical order. No problem. Next you want to resort the list by flight number so as to get lists of passengers for each flight. Again, no problem — except that it would be very nice if, for each flight list, the names were still in alphabetical order. This is the problem of stable sorting. To be a bit more mathematical about it, suppose we have a list of items $\{x_i\}$ with x_a equal to x_b as far as the sorting comparison is concerned and with x_a before x_b in the list. The sorting method is *stable* if x_a is sure to come before x_b in the sorted list.

Finally, we have the problem of *key* sorting. The individual items to be sorted might be very large objects (e.g. complicated record cards). All sorting methods naturally involve a lot of moving around of the things being sorted. If the things are very large this might take up a lot of computing time — much more than that taken just to switch two integers in an array. In this kind of case the best approach is the leave the actual data alone and instead work with a list of 'markers' for the data. You shuffle the markers, not the data itself. If you really want to, you can reorder the actual data *after* you have determined its ordering.

2.2 Two Simple Sorting Algorithms

Let me present two algorithms for sorting a list of numbers into ascending order (they will work with any linearly ordered set). They are meant to be 'obvious' rather than good. We will consider good methods later. The main aim here is to get something 'on the table' so that we know what we have to improve on later.

The first algorithm is probably the most mindless of all. I will call it the 'minimum element sort'. It is probably the method that you would use by hand if you had a very small number of items to sort.

2.2.1 Minimum Element Sort

The recipe is as follows: go through the list and find the smallest element. Swap it with the first element of the list. Now find the smallest element of the rest of the list and swap it with the second element of the list. And so on.

Rather more formally: to sort $(x_1, \ldots, x_n)$ into ascending order

```
for i = 1 to n-1 begin
  find smallest of $(x_i, \ldots, x_n)$
  swap with $x_i$
end
```

Putting in all the details, we get:

```
algorithm minimumElementSort
begin
  for i = 1 to n-1 begin
    min = x(i)
    minp = i
    for j = i+1 to n begin
      if x(j) < min then begin
min = x(j)
minp = j
      end
    end
    t = x(i)
    x(i) = x(minp)
    x(minp) = t
  end
```

```
end
```

I think this method is obvious enough not to need a proof.

The other concern is its efficiency. So let us consider the timing of the algorithm. The most important part of the process is the repeated search for the extreme element. It can be shown that for k items this takes time roughly proportional to k. The other parts of the algorithm are of lower order. Thus the overall running time for the algorithm (for reasonably large values of n) should be roughly proportional to

$$\sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2} \approx \frac{n^2}{2}$$

We say that this is an $O(n^2)$ sorting algorithm.

2.2.2 Insertion Sort

Once more we want to sort a list $(x_1, \ldots, x_n)$ into ascending order. Let me state the full algorithm directly

```
algorithm insertionsort
begin
  for i= n-1 down to 1 begin
    temp = $x_i$
    j = i+1
    while(j <= n and $x_j$ < temp) begin
      $x_{j-1} = x_j$
      j=j+1
    end
    $x_{j-1}$ = temp
  end
end
```

To help you see what is happening, here is the algorithm working on the list $(4, 3, 6, 1, 5, 2)$.

$i = 5$	4	3	6	1	**5**	2	temp = 5
$i = 4$	4	3	6	**1**	2	5	temp = 1
$i = 3$	4	3	**6**	1	2	5	temp = 6
$i = 2$	4	**3**	1	2	5	6	temp = 3
$i = 1$	**4**	1	2	3	5	6	temp = 4
$i = 0$	1	2	3	4	5	6	temp =

2.2.3 Proof of Algorithm

This algorithm is sufficiently complicated to deserve a proof. We use 'induction'.

Claim: at the end of step 'i' the elements $x_i, \ldots, x_n$ are in order.

If so then at the end of step '1' (the end of the algorithm) $x_1, \ldots, x_n$ are in order. So we just have to prove the claim. It is certainly true after the first step '$n-1$' — x_{n-1} and x_n will have been swapped, if necessary, to get them into order. Suppose that at the *start* of step 'i' the elements $x_{i+1}, \ldots, x_n$ are in order. The effect of step 'i' is to move element x_i along the list until it comes to a bigger element, or to the end of the list. When this has been done $x_i, \ldots, x_n$ will be in order. QED.

Approximate Timing For large n the main component of the time taken by the algorithm is that taken in shuffling elements to the right (the *while* loop). This will be the sum of the shift times for each $i = n-1, \dots, 1$. This time, for each i, is proportional to the distance shifted.

In the above example

i	5	4	3	2	1
distance	1	0	3	2	3

giving a total distance of 9.

For n items the minimum number of shifts required is 0, if list originally in order. The maximum number of shifts required is

$$1 + 2 + 3 + \cdots + (n-1) = \frac{n(n-1)}{2},$$

and this happens if the list is in reverse order. The actual number of shifts will depend on the list, so this is another case where we really want to know the *average*. Once more we have the problem of how to work out the average.

Before doing this it will help if we make some simplifications and assumptions.

1. Let us assume that we are dealing with lists of distinct elements. The analysis would be somewhat different (as would the algorithm) if the list contained a high proportion of repeated elements. As a simple example, you would probably use a different method to put a list of 0's and 1's into order.

2. There is then no loss of generality in assuming that the elements of the list are the integers $1, 2, 3, \dots, n$ in some order. As far as the algorithm is concerned it is not the actual values of the elements that matter so much as their relative positions in the list. The list $(1, 4, 3, 2)$ takes the same time to sort as the list $(100, 400, 302, 220)$ and the algorithm goes through exactly the same moves for each.

3. Finally we add in the extra, and serious, assumption that all these permutations are equally likely to occur — so that we can talk in terms of a random permutation. This is the 'in the absence of any other information' assumption. It may be the case in certain applications that some permutations are more likely than others — for example, the list may just be an ordered list with a small number of elements swapped around. This would require a different analysis.

Now consider the algorithm. At stage 'i' we take element x_i and move it a distance $d \in \{0, 1, \dots, n-i\}$ through the list to its correct position.

x_i has not been touched by the algorithm up to this point so, on average, we have no information about its proper position in the list. In other words, all possible shifts for x_i are equally likely. (Note that this depends on our assumption that all permutations are equally likely.)

So the average distance moved by x_i is

$$(0 + 1 + 2 + \cdots + (n-i))/(n-i+1) = \left(\frac{1}{n-i+1}\right)\frac{(n-i)(n-i+1)}{2} = \frac{n-i}{2}$$

So, on average, the total number of shifts is

$$\sum_{i=1}^{n-1} \frac{n-i}{2} = \frac{n(n-1)}{4}$$

So the overall time for the sort should be roughly proportional to n^2. Once more, this is an $O(n^2)$ algorithm — though it should be significantly faster than the minimum element algorithm.

2.2.4 A More Precise Analysis of Insertion Sort

We can give a more precise analysis of the operation of insertion sort. It is going to be a bit complicated. We will find out in the next section that insertion sort is not a very good sorting method and should not be used except in special circumstances. So why waste time on a complicated analysis? Simply because it happens to be a reasonably nice piece of mathematics from which you might learn something.

The important quantity in the analysis of the timing of Insertion Sort is the total distance through which the elements of the list are moved to the right during the sort. Let us assume, as usual, that we are dealing with a permutation of $(1, 2, \ldots, n)$.

By the nature of the algorithm, the number of places that i is moved in the sort is equal to the number of elements that are less than i and to the right of i in the original list. For example, if you apply insertion sort to the list

$$\pi = (7, 4, 5, 1, 3, 2, 6)$$

you will find that 1,2 and 6 do not get moved at all, 3 is moved one place (2 is less than it), 5 is moved 3 places (1,3,2 are less than it) and so on.

Bearing this in mind let us associate to each permuted list $\pi \in S_n$ a list $\mathbf{m} = (m_1, m_2, \ldots, m_n)$ where m_i is the number of numbers in π that are less than i and to the right of it. For the permutation of the previous paragraph we have

$$\mathbf{m} = (0, 0, 1, 3, 3, 0, 6)$$

In this $m_5 = 3$ because, of the four numbers less than 5, three of them are to the right of 5 in π.

The lists $\mathbf{m}$ that we produce from permutations obviously have the property that $0 \leq m_i < i$, because there are only $i - 1$ positive numbers less than i. In particular m_1 must be 0 and m_2 can only take the values 0 or 1.

Let us now introduce the following set M_n:

$$M_n = \{(m_1, m_2, m_3, \ldots, m_n) : \qquad 0 \leq m_i < i, \qquad i = 1, n\}$$

Then what we have done is constructed a mapping

$$\phi : S_n \to M_n \qquad \pi \to \mathbf{m}_\pi.$$

Example $n = 3$

$$M_3 = \{(0,0,0), (0,0,1), (0,0,2), (0,1,0), (0,1,1), (0,1,2)\}.$$

Under ϕ we have

$$\begin{array}{ll} (1,2,3) \to (0,0,0) & (1,3,2) \to (0,0,1) \\ (2,1,3) \to (0,1,0) & (3,1,2) \to (0,0,2) \\ (2,3,1) \to (0,1,1) & (3,2,1) \to (0,1,2) \end{array}$$

Note that, in this case, ϕ has turned out to be a bijection — it pairs off the elements of S_3 and M_3.

Lemma 2.1. $|M_n| = n!$

Proof. There are i possibilities for the i^{th} component. All the components are independent. So the total number of lists is $1 \times 2 \times 3 \times \ldots \times n = n!$. □

Theorem 2.2. The map $\phi : S_n \to M_n$ is a bijection.

Proof. Since S_n and M_n have the same number of elements it will be enough to show either that ϕ is one-one or that it is onto. Each implies the other.

Let me 'prove' that ϕ is onto by doing an example. The general proof is just a formalisation of what I do in this example.

Consider $\mathbf{m} = (0, 1, 0, 2, 1) \in M_5$. We must find a permutation $\pi \in S_5$ such that $\phi(\pi) = \mathbf{m}$. We find it like this:

$m_2 = 1$	so	1	is on the right of 2	2 1
$m_3 = 0$	so	$1, 2$	are on the left of 3	2 1 3
$m_4 = 2$	so	$1, 3$	must be on the right of 4	2 4 1 3
$m_5 = 1$	so	3	must be on the right of 5	2 4 1 5 3

So the required permutation is $\pi = (2, 4, 1, 5, 3)$.

You should be able to convince yourself that there is nothing special about this example and that this mode of construction will work for any $\mathbf{m} \in M_n$. That would prove that ϕ is onto and hence a bijection. $\square$

Let us now return to Insertion Sort. If we are using it to sort $\pi \in S_n$ then the total number of moves that we make is

$$t(\pi) = \sum_{i=1}^{n} m_i \quad \text{where} \quad \mathbf{m} = \phi(\pi)$$

Since ϕ is a bijection, averaging over S_n is the same as averaging over M_n. So the average number of moves made in Insertion Sort when applied to a list in S_n, assuming that all lists are equally probable, is

$$\bar{t}(n) = \frac{1}{n!} \sum_{\mathbf{m} \in M_n} t(\mathbf{m})$$

The rest of the argument is algebra. First, we reorder the sum

$$\bar{t}(n) = \frac{1}{n!} \sum_{\mathbf{m} \in M_n} \left(\sum_{i=1}^{n} m_i \right) = \frac{1}{n!} \sum_{i=1}^{n} \left(\sum_{\mathbf{m} \in M_n} m_i \right)$$

Now m_i takes the possible values $0, 1, 2, 3, \ldots, i-1$ and takes each value $n!/i$ times. So

$$\sum_{\mathbf{m} \in M_n} m_i = \frac{n!}{i}(0 + 1 + 2 + 3 + \cdots + (i-1)) = \frac{n!}{i} \frac{i(i-1)}{2} = \frac{n!}{2}(i-1)$$

Therefore

$$\bar{t}(n) = \frac{1}{n!} \sum_{i=1}^{n} \frac{n!}{2}(i-1) = \frac{1}{2} \sum_{i=1}^{n} (i-1)$$

Performing the sum we end up with our final result for the average number of moves made when applying Insertion Sort to a list from S_n:

$$\bar{t}(n) = \frac{n(n-1)}{4}$$

This is, of course, the same result that we obtained earlier by less precise means.

We have found that the maximum value of $t(\pi)$ is $n(n-1)/2$, the minimum value is 0 and the average value is half way between them at $n(n-1)/4$. Does that tell us everything we need to know about the insertion sort? No it does not. In fact it tells us very little about what might happen in any particular application of insertion sort. This is really the old story with averages. The average is an extremely simple statistic. It gives you one important fact about a population,

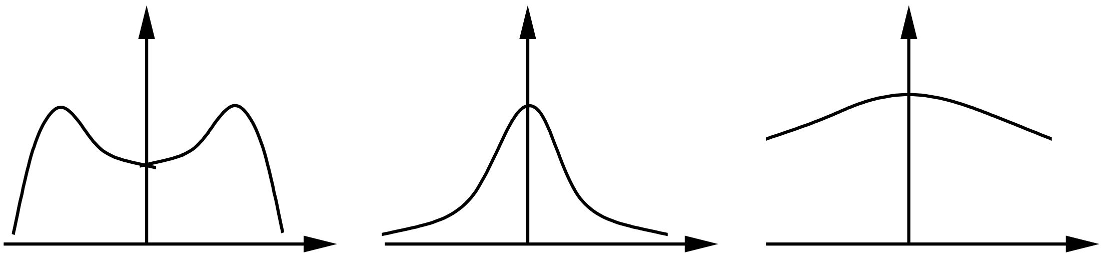


Figure 2.1: Potential distributions of running times about the mean.

but not more. At the moment we have no idea which, if any, of the graphs shown in Fig. 2.1 best represents the actual distribution of running times for Insertion sort.

The next statistical step after calculating the average (the mean) is to try to get some idea of the dispersal of the population away from the mean. The simplest approach to this is to calculate the *standard deviation*.

The standard deviation σ of the running times is, by definition,

$$\sigma^2 = \frac{1}{n!} \sum_{\pi \in S_n} (t(\pi) - \bar{t}(n))^2$$

I will leave the evaluation of this to you as a fairly heavy exercise in manipulation and summation.[1] The answer comes out as

$$\sigma^2 = \frac{2n^3 + 3n^2 - 5n}{72}$$

What does this tell us? The simplest statement that we can make is what is known as Chebychev's Inequality. This says that if we have a quantity f (a random variable) which has mean μ and standard deviation σ then

$$\Pr(|f - \mu| > k\sigma) < \frac{1}{k^2}$$

So, for example, the probability of f taking a value which is more than 4 standard deviations from the mean is less than 1/16. (I should point out that Chebychev's inequality tends to be rather 'pessimistic' and that distributions are usually closer to the mean than it suggests.)

Consider some values calculated from the expressions that we have obtained for the Insertion Sort timings.

n	μ	σ
20	95	15.4
50	612.5	59.8
100	247.5	168
500	62375	1866
1000	249750	5274

Table 2.1: Timing insertion sort

You can see that when n gets large σ becomes very small as a percentage of μ. This is because μ behaves like n^2 and σ like $n^{3/2}$. This tells us that, for large values of n, the time taken by insertion sort varies little, in percentage terms, from its mean value. For the case $n = 1000$, using Chebychev and the obvious symmetry of the distribution, we can say that

$$\Pr(t(\pi) < 200000) < 0.006$$

[1]You may find this quite hard. see Knuth (1998, Page 16) for the result, and techniques that help.

In other words, we are dealing with a distribution that has a very sharp peak concentrated at the mean and the timing for Insertion Sort should be very stable indeed. For large values of n a deviation in the time of more than 10% from the mean should be a rare occurrence.

2.3 A better sort: Quicksort

Let me now introduce you to another sorting technique, which uses a recursive 'divide and conquer' method. That is to say, it solves the problem of sorting a list by breaking the list into two parts and then applying the algorithm to each part in turn and so on recursively. There may or may not be some special technique used at the tail end of the recursion. Since recursive functions (functions that refer to themselves) are not officially allowed in FORTRAN these algorithms are more easily coded in other languages, like C or Pascal.

The idea here is very simple in its broad outline. We start with our list $\{x_1, \ldots, x_n\}$. We pick some element in the list, called the *separator*, and then rearrange the list so that all the elements that are less than or equal to the separator come before it and all the elements that are greater than the separator come after it. Having done this we then recursively apply the algorithm to each of these sublists. The recursion continues along any branch until its sublist shrinks down to zero or one element. (Actually it turns out to be a bit more efficient if we switch over to Insertion Sort when the lists drop to a length below about 10.)

The outline of the program would be:

```
algorithm quicksort(x,lo,hi)
// to sort $x_{lo}, \ldots, x_{hi}$
begin
  if hi > lo begin // i.e. if there is anything to sort
    choose separator s from list
    separate out the list into the form
    $(x_{lo}, \ldots, x_k, \quad s \quad , x_{k+2}, \ldots, x_{hi})$
    quicksort(x,lo,k)
    quicksort(x,k+2,hi)
  end
end
```

You then sort the whole list $x_1, \ldots, x_n$ by the call `quicksort(x,1,n)`.

The only detail left to be explained is how we do the separation. There is no problem in actually doing it. The problem is to make it efficient *and* to do it without introducing an extra array for temporary storage. I will state the algorithm and leave it to you to convince yourself that it works and does so in time proportional to the length of the list. The other problem is which element to choose as separator. The basic criterion, as we will see, is that the separation should be as even as possible. The obvious procedure of picking the first element can be a bit dangerous if the list is already nearly in order. People sometimes choose the 'middle' element and sometimes use a random number generator to pick an element at random. The choice does not affect the method.

```
algorithm separate(x,lo,hi,sep)
// rearranges the list $x_{lo}, \ldots, x_{hi}$ either side of $x_{sep}$.
begin
  xsep = x(sep)
  x(sep) = x(lo)
  x(lo) = xsep
  i = lo
```

```
08    j = hi
09    while( i < j ) begin
10      while( x(j) > xsep ) j = j - 1
11      x(i) = x(j)
12      while( i < j  and  x(i) <= xsep ) i = i + 1
13      x(j) = x(i)
14    end
15    x(i) = xsep
16  end
```

The algorithm starts in lines 1 to 3 by moving `x(sep)` to the start of the list, where it acts as a *sentinel*, ensuring that the `while` loop on line 10 necessarily terminates. It also makes available the separating value against which we compare everything in a temporary variable. Pointer `i` will always point to one after the top of the "low" list, while `j` will point to one before the "high" list.

The first while loop on line 10 starts at the top of the list, and drops down until it finds an element that is out of place. This is copied to its new position, effectively leaving a hole in position `j`. The first time this copy happens, the value `xsep` in `x(lo)` is overwritten, but the new value acts just as effectively as a sentinel. When the "increasing' while loop on line 12 starts, we have just put a "low" value in `x(i)`, so we move up the list until we find a "high' value. This is copied to the available hole in position `j` on line 13, and the whole process restarts. Note that each time other than the first that we enter the while loop on line 10, we have just put a "high" element in position `j`, so `j` is reduced by at least one each time.

At each stage of the algorithm everything below `i` and above `j` is properly ordered. Clearly $i \leq j$, and the algorithm is complete when $i = j$, in which case we fill in the hole with the separator, as in line 15. That leaves the problem of timing.

2.3.1 Timing the algorithm

Since the algorithm is recursive we expect to get a recurrence relation for the time taken. The following argument is going to be a bit vague in its use of the word 'average' in order to keep things simple. As usual we assume that the initial list is a permutation of $\{1, \dots, n\}$ and that all permutations are equally likely. Let T_n be the average time taken to *quicksort* n elements.

The sort consists of:

- doing the separation; and
- quicksorting the two sublists.

The above separation algorithm works in time proportional to n, say αn. Suppose the list splits into sublists of lengths k and $n-1-k$; this leaves one left over for the separator. Then on average

$$T_n = \alpha n + T_k + T_{n-1-k}$$

But we do not know the value of k in advance. All possible values are equally likely. So we take the average over all the possibilities:

$$\boxed{T_n = \alpha n + \frac{1}{n}\sum_{k=0}^{n-1}(T_k + T_{n-1-k})}$$

This simplifies at once (why?) to

$$T_n = \alpha n + \frac{2}{n}\sum_{k=0}^{n-1} T_k$$

This is a 'full-history' recurrence, and the usual way to deal with such things is to get rid of the history by elimination between two successive formulas. From the above

$$\frac{n}{2}(T_n - \alpha n) = T_0 + \cdots + T_{n-1}$$
$$\frac{n+1}{2}(T_{n+1} - \alpha(n+1)) = T_0 + \cdots + T_{n-1} + T_n$$

Subtract these and get, after a bit of rearranging,

$$\frac{T_{n+1}}{n+2} = \frac{T_n}{n+1} + \frac{\alpha(2n+1)}{(n+1)(n+2)}$$

Let $S_n = T_n/(n+1)$. Then

$$S_{n+1} = S_n + \frac{\alpha(2n+1)}{(n+1)(n+2)}$$

Hence

$$S_n = S_0 + \sum_{k=0}^{n-1} \frac{\alpha(2k+1)}{(k+1)(k+2)}$$

You get at the value of this sum by using partial fractions, just as you would if integrating. You can check that

$$\begin{aligned} S_n &= S_0 + 3\alpha \sum_{k=0}^{n-1} \frac{1}{k+2} - \alpha \sum_{k=0}^{n-1} \frac{1}{k+1}, \\ &= S_0 + 3\alpha(H_{n+1} - 1) - \alpha H_n = S_0 + 2\alpha H_n + \frac{3\alpha}{n+1} - 3\alpha. \end{aligned}$$

So we have, going back to T

$$T_n = 2\alpha(n+1)H_n + (S_0 - 3\alpha)(n+1) + 3\alpha$$

This is a bit complicated, but remember that $H_n \sim \ln n$. So, for large values of n we can reasonably say that

$$\boxed{T_n \propto n \ln n}$$

For large values of n this should be a huge improvement on the $O(n^2)$ time taken by our previous algorithms. To give you some idea of the improvement, consider the values in Table 2.2

n	n^2	$n \ln n$
10	100	23.0
50	2500	195.6
100	10000	460.5
500	250000	3107.3
1000	1000000	6907.8

Table 2.2: A comparison of n^2 and $n \log n$ for different values of n.

2.4 Merge Sort

There are a number of versions of this algorithm. I will present one which, though not the most efficient, is reasonably easy to describe. The important thing is the basic concept.

Before starting on the sorting algorithm I must introduce the concept of the *merging* of two ordered lists. This is simply the problem of taking two ordered lists and combining them into a single ordered list. This could be done by just sticking the two lists together and then sorting the result, but that is obviously stupid. We must make use of the fact that the two sublists are initially in order. The process we use is really quite obvious. We start at the start of each list and then 'deal' elements from each list into the new list one at a time, deciding at each step which list to take the next element from. The only slight complication in the algorithm is that we have to remember that one sublist may run out of elements before the other.

2.4.1 Merge Algorithm

This is an algorithm to merge the lists $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$ into a single ordered list $z_1, \ldots, z_{n+m}$. The lists x and y are initially in order.

```
algorithm merge(x,n,y,m,z)
begin
  xp = 1
  yp = 1
  zp = 1
  while( xp <= n and yp <= m ) begin
    if(x(xp) < y(yp) ) then begin
      z(zp) = x(xp)
      xp = xp + 1
    else
      z(zp) = y(yp)
      yp = yp + 1
    end
    zp = zp + 1
  end
  if( xp > n ) then begin // x ran out first
    for j = yp to m begin
      z(zp) = y(j)
      zp = zp + 1
    end
  end
  if( yp > m ) then begin // y ran out first
    for j = xp to n begin
      z(zp) = x(j)
      zp = zp + 1
    end
  end
end
```

In this algorithm `xp, yp, zp` point to the 'current position' in the x,y and z lists respectively. If the current x-value is smaller than the current y-value we add the x-value to z and move up `xp`. Otherwise we move the current y-value to z and move up `yp`. We then move up `zp`. This carries on until one or other of the original lists dries up. We then copy the remainder of the other list to z. The algorithm obviously works.

The time taken by this algorithm to merge the two lists is roughly proportional to the sum of the lengths of the two lists. This would be almost exact if both lists were exhausted at more or less the same time. The algorithm runs a bit faster if there is a long tail left to copy at the end.

2.4.2 Merge Sort

This is another 'divide and conquer' algorithm. We take our list and divide it in two at the 'mid-point'. We then sort each half, using this algorithm recursively, and use the above merge algorithm to combine the two halves into a single ordered list.

In outline:

```
algorithm mergesort(x)
begin
  let x1 be first half of x
  let x2 be second half of x
  mergesort(x1)
  mergesort(x2)
  merge(x1,x2,x)
end
```

Let me now make this a bit more precise. It will be convenient to introduce a *subroutine* with the following specification:

```
inmerge(x,i,j)
 $x_1,\ldots,x_n$ is a list and $1 \leq i < j \leq n$
 let m = [(i+j)/2] // the middle
 $x_i,\ldots,x_m$ are in order
 $x_{m+1},\ldots,x_j$ are in order
 the algorithm returns x with $x_i,\ldots,x_j$ in order.
```

This is just a re-write of the above merge algorithm with the notation changed a bit. To keep things simple you would probably need to introduce an extra 'storage array' to play the role of z in that algorithm, and then copy z back into the appropriate part of x.

Now for the full algorithm. I have written it rather untidily. You could tidy it up as an exercise.

```
algorithm mergesort(x,i,j)
// sorts into order the elements $x_i,\ldots,x_j$ of the list $x$]
begin
  m = [(i+j)/2]
  if j-i < 3 then sort the list crudely
  else begin
    mergesort(x,i,m)
    mergesort(x,m+1,j)
    inmerge(x,i,j)
  end
end
```

The complete list is of course then sorted with a call to `mergesort(x,1,n)`.

2.4.3 Timing the Algorithm

As before we look for a recurrence relation. We will ignore the special case $j - i < 3$ which occurs at the end of each branch. Let $T(n)$ be the average time to sort n elements. The sort consists of sorting two lists of 'half' the length and then combining them by a merge. So

$$T(n) = T(\text{floor}(n/2)) + T(\text{ceiling}(n/2)) + \alpha n.$$

We have already studied this recurrence and have found that, roughly speaking,

$$\boxed{T(n) \propto n \ln n}$$

So this algorithm, like Quicksort, is an $O(n \ln n)$ sorting algorithm.

2.5 Timings for Various Sorting Algorithms

Each type of sort was given a set of M randomly chosen lists of integers ($0 \leq k \leq 16383$) for the shown lengths n. The sorts were timed (in centiseconds per sort) using the following methods and numbers of lists.

Label	Method	M
A	Minimum Element Sort	200
B	Insertion Sort	500
C	Quicksort	500
D	Mergesort	500

The timings are given in Table 2.3.

n	Method A	Method B	Method C	Method D
100	10.5	5.8	2.0	3.7
200	41.0	22.2	4.6	8.6
300	90.9	49.4	7.4	14.6
400	160.8	86.9	10.3	19.4
500	250.6	135.5	13.3	27.3
600	360.1	194.6	16.4	32.5
700	489.6	264.8	19.6	37.3
800	638.8	346.0	23.0	43.3
900	807.9	437.0	26.2	51.8

Table 2.3: A comparison of sort times using different sorting methods.

Empirical Formulas for the data in Table 2.3 are as follows, where T is in centiseconds:

A) $T(n) \approx 0.001n^2$
B) $T(n) \approx 0.00054n^2$
C) $T(n) \approx 0.00043n \ln n$
D) $T(n) \approx 0.00083n \ln n$

These data confirm the theoretical analysis of the order of the timing, and also give some idea of the size of the contants of proportionality.

2.6 Optimal Sorting

The obvious question that now arises is: can we do still better? Could we, with a bit of ingenuity, devise a sorting technique that works in time which is faster than $O(n \ln n)$? I want to present an argument in this section that says that, under fairly broad assumptions, we cannot. $O(n \ln n)$ is essentially *optimal* for sorting n items.

We can undoubtedly do better in certain situations, usually when we have some extra information available about the elements of the list. For example, if we knew that every element of the list was one of the digits 0,1,. . . ,9 it would be easy to put together a sorting program that worked in time $O(n)$ (think about it).

But suppose that we are dealing with what are called *comparison-only* sorts where the only question that we can ask of two elements in the list is: which of the two is the bigger? In this case I hope to show you that we cannot do better than $O(n \ln n)$ even in principle.

I am actually going to study a slightly different problem, but its theory is equivalent to that of optimal sorting. The problem has to do with the old game of *Twenty Questions*. I choose a permutation π from S_n and you have to find out which one I have picked. You have to do this by asking me questions. The only questions that you are allowed to ask are of the form: *is element i bigger than element j?* The only answers I can give are *yes* or *no*. Can we find a lower limit for the maximum number of questions that you will have to ask me in order to find out what π is? That's the problem that we are going to study.

Let's consider the simple case of $n = 3$ first. A possible line of questioning could be the one shown in the tree shown in Fig 2.2 — a decision tree.

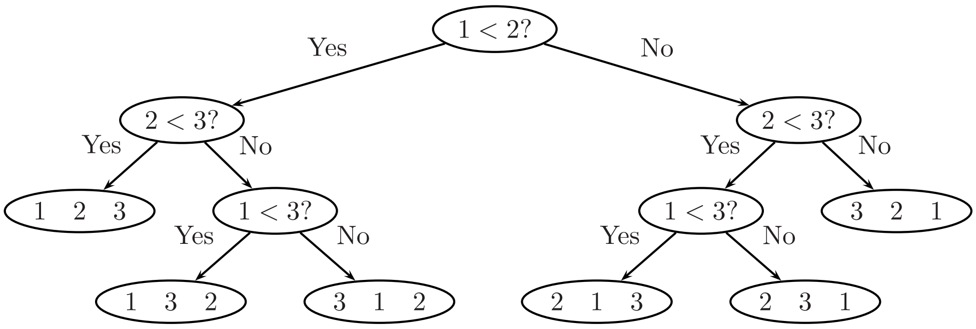


Figure 2.2: A decision tree for an element of S_3.

Notice that, with this arrangement of the questions you could not guarantee to be able to get the answer in fewer than 3 questions (though you might in some cases get it in 2). Could we rearrange the questions so as to guarantee to get the answer in two questions? No we could not, and we don't have to work through all possibilities to decide this.

The point is that each possible permutation has to appear as a *leaf* on the tree (and, by the logic of the tree, can only appear at one leaf). Now S_3 has $3! = 6$ elements, so any decision tree for S_3 must have precisely 6 leaves. You can easily check that every possible tree of *depth* 2 has at most 4 leaves. 4 is less than 6 so we cannot possibly distinguish all 6 permutations with 2 questions.

Now consider the general case of S_n. This has $n!$ elements, so a decision tree for S_n must have $n!$ leaves. You can easily convince yourself that a tree of depth d has *at most* 2^d leaves on it (the maximum for 3 questions was 8 leaves). So, if

$$2^d < n! \leq 2^{d+1}$$

we know that we cannot possibly find all $n!$ permutations in d questions though we *may* be able to find them in $d + 1$ questions (unlikely).

So what does this say about the value of d? Take logs of all parts of the inequality and get

$$d \ln 2 < \ln n! \leq (d+1) \ln 2$$

or

$$d < \frac{\ln n!}{\ln 2} \leq d + 1$$

So the minimum number of questions in which we can guarantee to be able to find any permutation is greater than

$$\text{opt} = \frac{\ln n!}{\ln 2}$$

To make more sense of this we now use Stirling's approximation

$$n! \approx \sqrt{2\pi n} n^n e^{-n}$$

or, taking logs,

$$\ln n! \approx \frac{1}{2} \ln(2\pi) + \frac{1}{2} \ln n + n \ln n - n$$

It is easy to see from this that

$$\text{opt} = O(n \ln n)$$

So $O(n \ln n)$ is the lower limit for the number of questions in which we can guarantee to be able to find any permutation from S_n.

The same logic applies to comparison-only sorting — which is just another way of deciding on the nature of a permutation. So we end up with the fact that it is logically impossible to produce comparison-only sorts that work in time better than $O(n \ln n)$.

2.7 Optimal Merging

The earlier discussion of the algorithm for merging two files leads to the following problem. We know that if we have two ordered files of lengths n and m respectively then they can be merged into a single ordered file in time proportional to $n + m$.

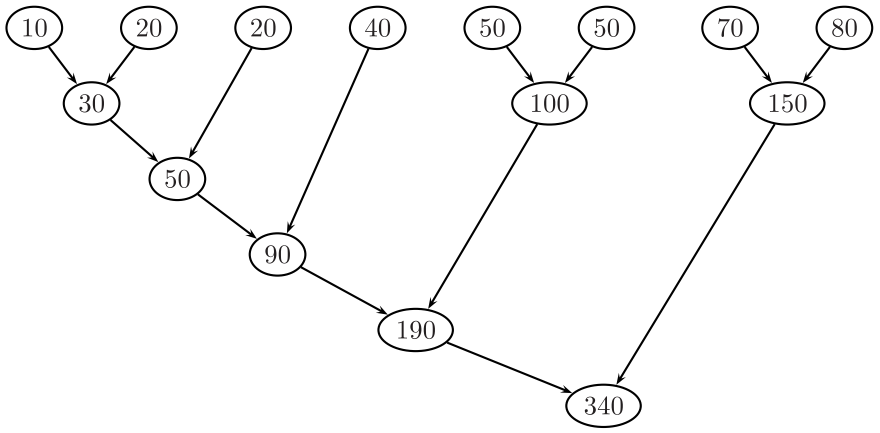


Figure 2.3: Merging files of different sizes.

Now suppose we have a number of ordered files of lengths $n_1, n_2, \ldots, n_k$. We want to combine them all into one big ordered file. We do it by combining files two at a time until they have all been combined. The question is: in what order do we combine the files?

As a simple example, suppose we have three files of lengths 20, 30 and 40. Call them A, B and C. If we combine A with B and then combine the result with C then the time taken is (up to a proportionality factor) $(20 + 30) + (50 + 40) = 140$. If, instead, we first combine B and C and combine the result with A the time taken is $(30+50)+(80+20) = 180$ which is significantly slower.

There is a very simple algorithm which tells you how to combine the files in a way that is as efficient as possible:

Algorithm: Of all the files available for combining at any stage, combine the two smallest (or two of the smallest if there is a choice).

As an illustration, suppose we have files $x_1, x_2, x_3, \ldots, x_8$ of sizes 10, 20, 20, 40, 50, 50, 70, 80. The tree diagram in Fig. 2.3 shows the sequence in which these files should be combined.

2.7.1 Some Example Sort Implementations

As with the Euclidean Algorithm, another way to learn how a sort algorithm behaves is to watch it working. An implementation of a number of sorting algorithms in the *Java* language is available in the hypertext version of this document, enabling you to do this.

Questions 2 (Hints and solutions start on page 90.)

Q 2.1. Just to check that you understand the basic notations, let

$$\pi_1 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 4 & 3 & 1 & 5 & 2 & 6 \end{pmatrix}, \qquad \pi_2 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 6 & 5 & 4 & 3 & 2 & 1 \end{pmatrix}$$

be permutations in S_6. Work out $\pi_1\pi_2$, $\pi_2\pi_1$, $\pi_1^{-1}\pi_2\pi_1$, π_1^3. Decompose π_1 and π_2 as products of disjoint cycles.

Show that the inverse of a cycle is a cycle. What is the inverse of $\langle a_1, a_2, \dots, a_k\rangle$?

Q 2.2. Suppose that a permutation $\pi \in S_n$ decomposes into a product of disjoint cycles as

$$\pi = \sigma_1\sigma_2\dots\sigma_k.$$

Find the smallest positive integer m such that $\pi^m = id$ (the *order* of π) in terms of the lengths of the sigmas.

Q 2.3. If we have a pack of 16 cards then the effect of a split-and-interleave shuffle is one of the following permutations, depending on which 'hand' is uppermost.

$$\pi_1 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 1 & 9 & 2 & 10 & 3 & 11 & 4 & 12 & 5 & 13 & 6 & 14 & 7 & 15 & 8 & 16 \end{pmatrix}$$

$$\pi_2 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 9 & 1 & 10 & 2 & 11 & 3 & 12 & 4 & 13 & 5 & 14 & 6 & 15 & 7 & 16 & 8 \end{pmatrix}$$

Break each permutation into a product of disjoint cycles and then find the smallest positive integers n, m such that $\pi_1^n = id$ and $\pi_2^m = id$. (If you've got paper and patience you can do the same for a pack of 52 cards.)

Q 2.4. Bearing all this stuff about decompositions in mind, what permutations in S_{10} have the highest degree, and what is that degree? (Once more: the *degree* of a permutation is the smallest positive integer n such that $\pi^n = id$, where id is the 'identity permutation' that changes nothing.)

What are the permutations of highest order in S_{20}? Please don't check through all $20! = 2,432,902,008,176,640,000$ cases!

Q 2.5. In the following parts I want you to write an algorithm to perform the desired calculation. At the least, sketch out the way in which such an algorithm would operate. At the most, if enthusiastic, write an actual program in some language.

Let us adopt the convention that a permutation is represented by an array $(a_1, a_2, \dots, a_k)$, standing for the permutation

$$\pi = \begin{pmatrix} 1 & 2 & \dots & k \\ a_1 & a_2 & \dots & a_k \end{pmatrix}$$

1. Given two permutations π_1 and π_2, calculate their product;
2. calculate the inverse of a permutation; and
3. decompose a permutation into a product of disjoint cycles.

You will need to decide on a useful form for the 'output'.

Q 2.6. Work through each sorting algorithm carefully for each of the following lists — pretend to be a computer.

5,4,6,8,2,1,3,9,0,7
0,1,2,3,4,5,6
6,5,4,3,2,1,0
1,1,1,2,2,2,2

Q 2.7. If, in Quicksort, we choose the first element of each sublist as separator at each stage then the very worst case of the algorithm comes when you present it with a list that is already in order!

In this case the algorithm achieves as little *dividing and conquering* as possible. If $T(n)$ is the running time for this case and the separation time is αn, show that $T(n) = \alpha n + T(n-1)$ and deduce that $T(n) = O(n^2)$.

Q 2.8. Which of the sorting algorithms are stable? Recall the definition of *stable*: We have a list of elements $x_1, \dots, x_n$. They may all be different but some are the same as far as the inequality is concerned (e.g. the elements may be pairs of numbers $x_i = (a, b)$ and we may be putting them into order according to the first element). Suppose that x_i and x_j are equal as far as the inequality is concerned. Then the sort is *stable* if it preserves the order in which x_i and x_j came in the list.

Q 2.9. Before looking at the notes, which contain the answer, try to think up an algorithm for *merging* two ordered list into a single ordered list. You are given lists $(x_1, \dots, x_n)$ and $(y_1, \dots, y_m)$ both of which are in increasing order and you want to put them together into a list $(z_1, \dots, z_{n+m})$ which is also in increasing order. Your method should not take time worse than $O(n+m)$ (it should be roughly proportional to $n+m$). When you have done this compare your result with the version in the notes.

Can you adapt the algorithm so that it finds the *union* of the two lists $x \cup y$? — i.e. there are to be no repeated elements in z.

Q 2.10. Suppose we want to sort a list $(x_1, \dots, x_n)$ into order. If we use Insertion Sort it will take time roughly proportional to n^2. Suppose that we do the following instead: first split the list into two halves, then apply insertion sort to each half, then merge the results by the algorithm of the previous question. Show that this should lead to a faster sorting routine if n is reasonably large. What about subdividing the list into a larger number of sublists?

Suppose that you don't split the list into two *halves* but just divide it randomly into two parts before starting the above process. Are you likely to get much of an advantage over direct Insertion Sort or does this depend on the split being almost exactly half way?

Q 2.11. Write a careful algorithm to decide the alphabetical order of two words. To be more precise: define a function `ord(a,b)` which, given the strings of letters `a` and `b` returns 1 if `a` is alphabetically before `b`, 0 if they are equal and -1 if `b` is alphabetically before `a`. You may assume that the end of each word is marked in the string by the 'letter' ϵ.

Q 2.12. We have been working with recursive programs, so let's look at recursive functions.

Let $P(n)$ be the number of ways of decomposing n as a sum of positive integers, ignoring order.

$$\text{e.g.} 4 = 4 = 3 + 1 = 2 + 2 = 2 + 1 + 1 = 1 + 1 + 1 + 1 \text{so} P(4) = 5$$

P is a surprisingly important function. It would be nice if we could calulate its values without having to write down all possible decompositions.

The easiest approach is to define a function $Q(n, m)$ which gives the number of ways of decomposing n using no numbers bigger than m. For example $Q(4, 1) = 1$, $Q(4, 2) = 3$, $Q(4, 3) = 4$. Obviously $P(n) = Q(n, n)$.

I am now going to give you a recursive definition of Q. I want you to show that it is correct in the following two senses: (1) starting with any (n, m) $(n, m \geq 1)$ it allows us to find $Q(n, m)$, and (2) the answer you get is correct.

The following are meant to be applied in order:

1. $Q(1, m) = 1$ if $m \geq 1$
2. $Q(n, 1) = 1$ if $n \geq 1$
3. $Q(n, m) = Q(n, n)$ if $m \geq n \geq 1$
4. $Q(n, n) = Q(n, n-1) + 1$ if $n \geq 2$
5. $Q(n, m) = Q(n, m-1) + Q(n-m, m)$

The usual problem you run into with such definitions is that it is possible to get trapped into an infinite loop from which there is no escape.

Whilst you are at it, work out a few values of $P(n)$.

Another famous recursive function is Ackerman's function. This was designed to be a nuisance. The function $A(n, m)$ is defined as follows:

1. $A(0, m) = m + 1$
2. $A(n, 0) = A(n-1, 1)$
3. $A(n, m) = A(n-1, A(n, m-1))$

Work out $A(3, 2)$. Roughly how many digits has $A(4, 2)$? If you had to write out $A(4, 4)$ would the forests of Sweden be sufficient, or would you have to use up the Amazon basin as well?

Q 2.13. In case you found the timing argument for insertion sort a bit too glib, try the following alternative:

Let $\pi \in S_n$ be a permutation of $\{1, \ldots, n\}$. Think of π as a (shuffled) list. For $i = 1, \ldots, n$ let $m_\pi(i)$ be the number of elements in the list π that are to the *right* of i and are less than it.

$$\text{if} \pi = (42153) \in S_5 \text{then} m_\pi(1) = 0, \quad m_\pi(2) = 1, \quad m_\pi(3) = 0 \text{etc}$$

Why is $m_\pi(1) + \cdots + m_\pi(n)$ equal to the number of 'moves' that need to be made in the insertion sort of $\pi \in S_n$?

Show that $0 \leq m_\pi(i) \leq i - 1 \text{for} i = 1, \ldots, n$.

$$\text{Let} P_n = \{(m_1, \ldots, m_n) \in \mathbb{Z}^n : 0 \leq m_i \leq i - 1 \qquad i = 1, \ldots, n\}$$

We have a mapping $\psi : S_n \to P_n$ given by $\psi(\pi) = \mathbf{m}_\pi$.
(1) Show that $|P_n| = n! = |S_n|$.
(2) We now want to prove that ψ is a *bijection*. Since both sets have the same number of elements it is sufficient to prove that ψ is *onto*. So we have to prove that, given $\mathbf{m} \in P_n$, we can find $\pi \in S_n$ such that $\psi(\pi) = \mathbf{m}$.

Show how to reconstruct $(4, 2, 1, 5, 3)$ from $\mathbf{m} = (0, 1, 0, 3, 1)$.
What perm $\pi \in S_n$ produces $\mathbf{m} = (0, 0, 1, 2, 0, 4, 1, 4, 6)$?
Can you now give a rough argument to show that ψ is onto?

For $\pi \in S_n$ let

$$\rho(\pi) = \sum_{i=1}^{n} m_\pi(i)$$

— the total number of 'moves' in the insertion sort of π. Then the average number of moves for all $\pi \in S_n$ is

$$\begin{aligned} av &= \frac{1}{n!} \sum_{\pi \in S_n} \rho(\pi) \\ &= \frac{1}{n!} \sum_{\mathbf{m} \in P_n} \left(\sum_{i=1}^{n} m_i \right) \qquad \text{why?} \\ &= \frac{1}{n!} \sum_{i=1}^{n} \sum_{\mathbf{m} \in P_n} m_i \end{aligned}$$

What is $\sum m_i$ over $\mathbf{m} \in P_n$? (Work out some simple examples.) Now calculate av.

Q 2.14. a) Describe the "Quicksort" algorithm for sorting a list and give pseudo-code for a routine

```
quicksort(x,lo,hi)
```

which sorts the sublist $x_{lo}, \ldots, x_{hi}$ of the list `x`. You may assume that the routine `separate(x,lo,hi,sep)`, in which a list is separated into two sublists based on the separator `sep`, is available. Illustrate your answer by "quicksorting" the list

6 4 8 2 11 5 3 7 9 1 10

[When making a "random" choice from a list, choose the item that occurs in the first position, so a "random" choice from $[2, 1, 3]$ would be 2.]

b) Let T_n be the average time taken to quicksort a list of n elements. Assuming that the separation algorithm applied to a list of length n takes time αn, derive the recurrence relation

$$T_n = \alpha n + \frac{2}{n} \sum_{k=0}^{n-1} T_k,$$

explaining what other simplifying assumptions have been made. Hence show that

$$\frac{1}{n+1} T_n = T_0 + \sum_{k=0}^{n-1} \frac{\alpha(2k+1)}{(k+1)(k+2)}.$$

c) How does Quicksort compare with insertion sort and Heapsort?

Chapter 3

Abstract Data Types

3.1 Introduction

One of the simplifications made in modern abstract algebra is the identification of appropriate structures in which to consider different problems. Thus for example Galois theory is most natural expressed working over a field, while the study of the abstract Euclidean algorithm is done in an integral domain. Each is a type of ring with additional structure, and each has the properties in common to all (commutative) rings. The gain is often suggested to be one of generality — that as soon as you know something is an integral domain, then you can use all the available results. But another gain is that the set of allowed operations is constrained. Thus there is no point in thinking about dividing in an integral domain, because the concept is not available, even though in a practical example of an integral domain, such as $\mathbb{Z}$, given $n \in \mathbb{Z}$, the inverse, $1/n$ is perfectly well defined.

A similar abstraction is useful when considering algorithms. It forces attention on the underlying structure, and disallows ad-hoc manipulations. The abstraction is known as an Abstract Data Type or (sometimes, for brevity) ADT. And like the abstract structures of algebra, an understanding of their utility can only follow a set of definitions, and simple examples to illustrate that the structures occur frequently in practice.

As a first example consider a restricted class of integers, say $\{n \in \mathbb{Z} : 0 \le n \le 100000000\}$. There are some situations when it would be helpful to have a name for such a class. And it may be helpful to name an object of this type as `annualSalaryInPence`. The set of allowed operations can then be sensibly restricted. Thus while it makes sense to compute $1000000 - 25$, doing arithmetical operations with an item of type `AnnualSalaryInPence` and an item of type `AgeInYears` is likely to be nonsense.[1]

We next consider some Abstract Data Types.

3.1.1 The ADT Stack

A **stack** is a collection of elements or items, for which the following operations are defined:

`create(S)` creates an empty stack `S`;

`isEmpty(S)` is a predicate [2] that returns "true" if `S` exists and is empty, and "false" otherwise;

`push(S,item)` adds the given `item` to the stack `S`[3] ; and

[1] Conventionally the class name is often written SomeClass, with an item of type SomeClass being given the name someClass.

[2] A predicate is a function which only takes the values "true" or "false."

[3] Here and elsewhere, we will assume that "all items are equivalent". In some computer languages this is valid, but with others (eg C++) which have strong forms of type checking, an attempt to put a real number into a stack of integers would be a compile time error.

`pop(S)` removes the most recently added item from the stack `S` and returns it as the value of the function;

The primitive `isempty` is needed to avoid calling `pop` on an empty stack, which should cause an error. In the real world, the `push` primitive should return an error when the stack is "full" and so no more items can be added. We ignore such complications here. An additional operation is sometimes defined, called `top(S)` which simply returns the last item to be added to the stack, without removing it. Show that this can be done in terms of the primitive operations described above.

The word "stack" is chosen by analogy with a stack of plates, where the last one placed on the top of the stack is usually the first one to be used. Thus a stack implements a "last in first out" ordering on a set of items.

3.1.2 The ADT Queue

A **queue** is a collection of elements, or items, for which the following operations are defined:

`create(Q)` creates an empty queue `Q`;

`isEmpty(Q)` is a predicate that returns "true" if `Q` exists and is empty, and "false" otherwise;

`add(Q,item)` adds the given `item` to the queue `Q`; and

`next(Q)` removes from the queue `Q` the least recently added item that remains in the queue, and returns it as the value of the function;

The primitive `isempty(Q)` is needed to avoid calling `next` on an empty queue, which should cause an error. As for a stack, we ignore the situation when the queue is "full"; a finite queue, in which the length is actually fixed by the problem, is a different structure.

The word "queue" is thus like the queue at a counter for service, in which customers are dealt with in the order in which they arrive.

An important point about Abstract Data Types; is that they describe properties of a structure without specifying an implementation in any way. Thus an algorithm which works with a "queue" data structure will work however it is implemented. The implementation suggested in Exercise. 3.1 may not be efficient, but the efficiency of an implementation can be separated from the details of the algorithm. We discuss below different ways in which stacks and queues can be implemented efficiently.

3.1.3 The ADT Array

An array is probably the most versatile or fundamental Abstract Data Type, left until now simply to show it was reasonable to consider others. An **array** is a finite sequence of storage cells, for which the following operations are defined:

`create(A,N)` creates an array `A` with storage for `N` items;

`A[i]=item` stores `item` in the `i`$^{\text{th}}$ position in the array `A`; and

`A[i]` returns the value of the item stored in the `i`$^{\text{th}}$ position in the array `A`.

Of course an array is another way of looking at a vector; the emphasis here is on the functions that create and access it. It is a matter of convenience how the array cells should be labelled. When it is necessary to be precise, we assume they are labelled $A[0], \ldots, A[N-1]$, as is the convention in eg C++, but the "more obvious" convention using $A[1], \ldots, A[N]$ is equally valid. It *is* important to stick with one convention or the other. Whatever FORTRAN or C may tell you, we regard accessing $A[i]$ as an error if a value has not previously been assigned to it!

The data storage in most computers can be regarded as an array, so in practice it provides the ultimate implementation for most of our other Abstract Data Types

Example 3.1. An array can be used to implement a stack.

Solution We implement a stack using an array of size N. The construction will fail if an attempt is made to push more than N elements onto the stack.

The `create(S)` operation is trivial. An array `A` is created, and associated with the stack `S`. In addition, an internal pointer or cursor `p` say, is initialised to point to the first cell in `A`. To be definite, we take this as 0;

`isEmpty(S)` is true iff `p = 0`

`push(S,item)` is implemented as `A[p] = item; p = p+1` and

`pop(S)` is implemented as `return(A[p]); p = p-1`

Of course there should be additional error checking for safety, to avoid problems if the capacity of array is exceeded.

It is clear at this stage that a stack *is* a relatively simple structure; we have just implemented it as an array in which the read-write pointer only moves one step between successive calls.

Example 3.2 (The three-coloured flags). You are given an array of length N, each of whose N members is either a red, white or blue token. the aim is to re-arrange the tokens in such a way that all the blue tokens appear before the white ones, and that all the red tokens appear last. The following tools are available:

- predicates `B(i)`, `W(i)` and `R(i)` which return true if and only if token `i` is respectively blue, white and red; and
- an operation `swap(i,j)` which interchanges the tokens in positions `i` and `j`; the case `i = j` is not excluded.

Each of the predicates may only be calculated one for each token. The aim is to accomplish the re-arrangement with the minimum number of calls to the `swap` operation.

Solution Our aim is to sort the tokens so that the known blue area B comes before the known white area W, and so that both of these come before the known red area R. In a general state we will have an area X of tokens of unknown colour which follows W and precedes R. Initially each of B, W and R is empty and X consists of the whole array. We use three variables to keep track of the current state of the array. Let `b` and `w` point to the tokens which come immediately after the blue and white areas respectively, and let `r` point to the token that comes immediately before the red area.

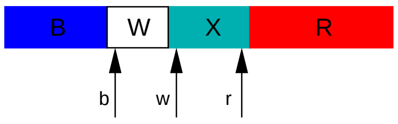


Figure 3.1: Sorting the array of flags

An intermediate state during the sorting of the array is shown in Fig. 3.1. Consider the following algorithm:-

```
b = w = 1;r = N;
while (w < r + 1) begin
  if (W(w)) w = w+1 // it was a white token
  else if (B(w)) begin // it was a blue token
    swap(w,b);b=b+1;w=w+1
    end
  else begin // it was a red token
    swap(r,w);r = r-1
  end
end // while
```

To see that the algorithm is finite, note that at each stage either w is increased or r is decreased. Thus the length $|X|$ of the region X, which is $r + 1 - w$ decreases each time, while the "sorted' condition; that

$$B < W < X < R \tag{3.1}$$

remains true. Since the algorithm terminates when the length of X becomes 0, we finish with a sorted list. We have seen that the loop is executed N times, and that there are $|B| + |R|$ calls to the routine `swap`.

3.1.4 The ADT List

A (linked) list is another fundamental data type, which can be considered almost as basic as the array. Just as eg FORTRAN regards the array as central, so the language **lisp** takes the list as the object in terms of which everything else is described.[4] If you want to take a closer look, try (Winston & Horn 1989) or (Wilensky 1986).

An **list** is a finite sequence of storage cells, for which the following operations are defined:

`create(l)` creates an empty list `l`;

`insert(item,i,list)` changes the list from the form $(a_1, \ldots , a_{i-1}, a_i, \ldots a_n)$ by inserting `item` between a_{a-1} and a_i; it is an error to call this unless `list` has at least $i - 1$ items before the call.

`delete(i,list)` deletes the item at position `i` from the list, returning `list` containing one fewer item; and

`read(i,list)` returns the item at position `i` in the list, without changing the list.

Note what is missing; it is easy (at least based on the assumption that the operations specified are quick) to traverse the list in the order it was created; but going "backwards" along the list may be hard. It is often convenient to speak of the first item as the *head* of the list, and the remainder of the list as the *tail*.

Example 3.3. Show how to implement a stack in terms of the Abstract Data Type list.

Solution This is clear; `push` simply adds the item to the head of the list, while `pop` returns the head of the list, and replaces the list by its tail.

There is an obvious implementation of a list in terms of an array; simply treat the list as a stack. However the additional operations, particularly those involved in inserting elements in the middle of the list, involve a great deal of copying. An alternative implementation is in terms of *pairs*, one member of which contains the item, while the other member points to the next pair in the list. A special pointer indicates that the list has come to an end. Such an arrangement is represented diagrammatically in Fig 3.2, which shows a list of five elements. The first pointer (ie the one pointing to "value1") can the be stored as the list.

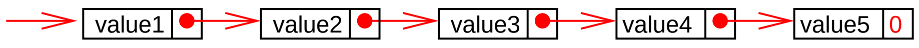


Figure 3.2: A list implemented using cells and pointers.

Surgery on such a list is then a simple matter of altering a few pointer values. Such surgery is shown in Fig 3.3. A common feature of list implementations is shown there in the hanging

[4]The name is an abbreviation of *l*ist *p*rocessing language, and is not an acronym for "*l*ots of *i*rrelevent *s*tupid *p*arentheses, as some who have tried to read it suggest. My own feeling is that lisp is so regular that it is likely to appeal to mathematicians.

pointer attached to "value3". It could be considered as a list in its own right, with the pointer still pointing to "value4". However it is usually difficult to tell this just by examining the pair itself, and the surgery shown might well have marooned this storage; it appears to be used, but nothing refers to it. In this circumstance, it is known as "garbage", and a specialised program known as a "garbage collector" is often used to recover such storage.

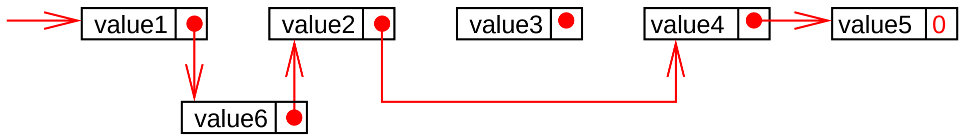


Figure 3.3: The same list after an addition and a deletion.

We shall not make a great deal of use of lists; but you should be aware that they form a real alternative to an array as a primitive Abstract Data Type, indeed computers have been built on this premise.

3.1.5 The ADT Priority Queue

A priority queue is a collection of elements or items, each of which has an associated priority.
The operations available are:-

`create` creates an empty priority queue;

`add(item)` adds the given `item` to the priority queue; and

`remove` removes the item with the highest priority from the queue and returns it as the value of the function.

Thus, unlike a queue, this is *not* a simple "first in — first out" structure. We take up the question below of whether it can be implemented simply; for the present, it is here to show there are Abstract Data Types which can be described easily, but which do not have an "obvious" quick implementation.

3.2 Trees

We have already seen examples of trees used in the work on sorting in Section 2. There are many flavours of trees; in this section we characterise them as Abstract Data Types and look at some of their uses.

3.2.1 The ADT tree

A **tree** is a finite set of elements or **nodes**. If the set is non-empty, one of the nodes is distinguished as the **root** node, while the remaining (possibly empty) set of nodes are grouped into subsets, each of which is itself a tree. This hierarchical relationship is described by referring to each such subtree as a **child** of the root, while the root is referred to as the **parent** of each subtree. If a tree consists of a single node, that node is called a **leaf** node.

It is a notational convenience to allow an empty tree. It is usual to represent a tree using a picture such as Fig. 3.4, in which the root node is A, and there are three subtrees rooted at B, C and D. The root of the subtree D is a leaf node, as are the remaining nodes, E, F, G, H and I. The node C has a single child I, while each of E, F, G and H have the same parent B. The subtrees rooted at a given node are taken to be *ordered*, so the tree in Fig. 3.4 is different from the one in which nodes E and F are interchanged. Thus it makes sense to say that the *first* subtree at A has 4 leaf nodes.

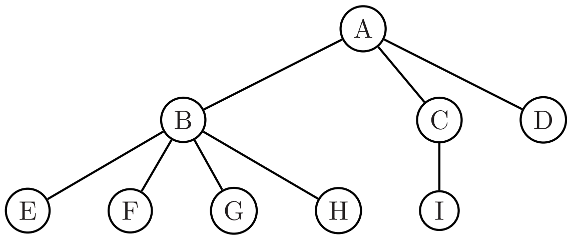


Figure 3.4: A simple tree.

Example 3.4. Show how to implement the Abstract Data Type tree using lists.

Solution We write `[A B C]` for the list containing three elements, and distinguish `A` from `[A]`. We can represent a tree as a list consisting of the root and a list of the subtrees in order. Thus the list-based representation of the tree in Fig 3.4 is

```
[A [[B [[E] [F] [G] [H]]] [C [I]] [D]]].
```

Note that for example we confuse `[D]` with `[D []]`, which we should write to be strictly correct, since `D` has an empty set of subtrees. To give an idea of how common the list representation is, my editor was happily matching brackets as I typed the above; in other words, when I typed the last "`]`", the cursor moved to the "`[`" before "A" to show the balancing bracket.

3.2.2 Traversals

It is often convenient to a single list containing all the nodes in a tree. This list may correspond to an order in which the nodes should be visited when the tree is being searched. We define three such lists here, the **preorder**, **postorder** and **inorder** traversals of the tree. The definitions themselves are recursive:

- if T is the empty tree, then the empty list is the preorder, the inorder and the postorder traversal associated with T;
- if $T = [N]$ consists of a single node, the list $[N]$ is the preorder, the inorder and the postorder traversal associated with T;
- otherwise, T contains a root node n, and subtrees $T_1, \ldots, T_n$: and
  - the *preorder* traversal of the nodes of T is the list containing N, followed, in order by the preorder traversals of $T_1 \ldots, T_n$;
  - the *inorder* traversal of the nodes of T is the list containing the inorder traversal of T_1 followed by N followed in order by the inorder traversal of each of $T_2, \ldots, T_n$.
  - the *postorder* traversal of the nodes of T is the list containing in order the postorder traversal of each of $T_1, \ldots, T_n$, followed by N.

3.2.3 Binary Trees

A **binary tree** is a tree which is either empty, or one in which every node:

- has no children; or
- has just a left child; or
- has just a right child; or
- has both a left and a right child.

We have already seen examples of binary trees; both Fig 2.2 considered in the discussion of optimal sorting, and Fig 2.3 on Page 31 which was used to illustrate the process of optimal merging, are binary trees, although in each case, the situation in which a node can have just one child does not occur. Note also that this is rather different from an *ordered* tree, since for a binary tree we distinguish between a node with a single left child, and one with a single right child. An alternative definition, which is "cleaner" is to first introduce the concept of a terminal, or "null" node, which has no children, and then define a binary tree to be one in which every node other than the null node has precisely two children.

A **complete** binary tree is a special case of a binary tree, in which all the levels, except perhaps the last, are full; while on the last level, any missing nodes are to the right of all the nodes that are present. An example is shown in Fig. 3.5.

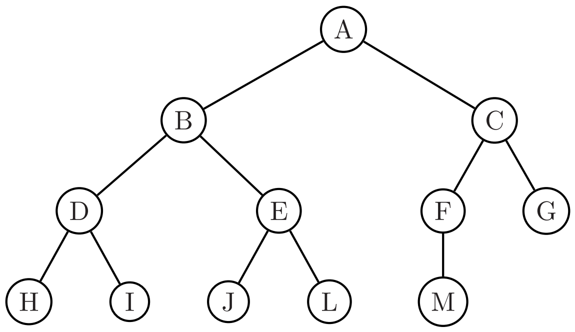


Figure 3.5: A complete binary tree: the only "missing" entries can be on the last row.

Example 3.5. Give a space - efficient implementation of a complete binary tree in terms of an array `A`. Describe how to pass from a parent to its two children, and vice-versa

Solution An obvious one, in which no space is wasted, stores the root of the tree in `A[1]`; the two children in `A[2]` and `A[3]`, the next generation at `A[4]` up to `A[7]` and so on. An element `A[k]` has children at `A[2k]` and `A[2k+1]`, providing they both exists, while the parent of node `A[k]` is at `A[k div 2]`. Thus traversing the tree can be done very efficiently.

3.2.4 Huffman Codes

We discuss here an example in which the binary tree structure is of value. Consider the problem of coding (in binary) a message consisting of a string of characters. This is routinely done in a computer system; the code used almost universally at present is known as ASCII[5], and allocates 8 bits to store each character. Thus A is represented using decimal 65, or 01000001 in binary etc. A more modern one which allows a much wider range of languages to be represented is Unicode, which allocates 16 bits to each character. This is used for example by the language Java, and is an extension of ASCII in that any ASCII character can be converted to Unicode by prefixing it with the zero byte. Although these codes are simple, there are obvious inefficiencies; clearly Unicode wastes at least half of the available space when storing plain ASCII.

Another source of inefficiency may lie in using the same number of bits to represent a common letter, like "e" as to represent "q" which occurs much less frequently. What if we permit character codes to have a variable length? An apparent difficulty is the need to have a neutral separator character to indicate the end of one character code, and so delimit it from the next. Say a code has the **prefix property** if no character code is the prefix, or start of the the code for another character. Clearly a code with the prefix property avoids this need to have additional separators, while permitting variable lengths. An obvious question is:

[5]American Standard Code for Information Interchange

- do codes with the prefix property exist; and if so
- is there a "best" one to use?

In Table 3.1 we give an example of such a prefix code for a small alphabet, and contrast it with a simple fixed length code. It is clear that there are savings in this case which make it worth going further. We will see shortly why the example has the prefix property; in the meantime check that the string "0000100111" in Code 2 decodes uniquely as "acbd".

Symbol	Code 1	Code 2
a	001	000
b	001	11
c	010	01
d	011	001
e	100	10

Table 3.1: Code 1 has fixed length code; Code 2 has the prefix property.

Consider now a binary tree, in which each leaf node is labelled with a symbol. We can assign a binary code to each symbol as follows: associate "0" with the path from a node to its left child, and "1" with the corresponding path to the right child. The code for a symbol is obtained by following the path from the root to the leaf node containing that symbol. The code necessarily has the prefix property; the tree property means that a leaf node cannot appear on a path to another leaf. Conversely it is clear how to associate a binary tree with a binary code having the prefix property; the code describes the shape of the tree down to the leaf associated with each symbol.

Of course a fixed length code necessarily has the prefix property. We show in Fig. 3.6 the binary trees corresponding to the two codes given in Table 3.1, thus incidentally demonstrating that the variable length code in the example does have the prefix property.

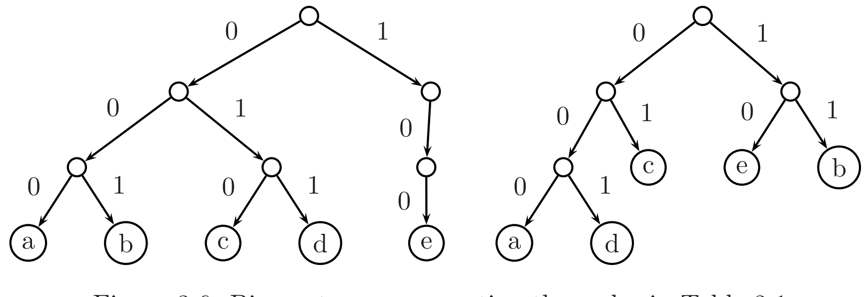


Figure 3.6: Binary trees representing the codes in Table 3.1

We now describe how to build the binary Huffman code for a given message. This code has the prefix property, and in a fairly useful sense turns out to be the *best* such code. We describe the code by building the corresponding binary tree. We start by analysing the message to find the frequencies of each symbol that occurs in it. Our basic strategy will be to assign short codes to symbols that occur frequently, while still insisting that the code has the prefix property. Our example will be build around the message

A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS [6]

The corresponding frequencies are given in Table 3.2; note that in this case, we choose to include the space symbol " ", written in the table as ␣.

[6]This idea is used frequently in (Sedgwick 1995) to illustrate algorithms.

Now begin with a collection (a forest) of very simple trees, one for each symbol to be coded, with each consisting of a single node, labelled by that symbol, and the frequency with which it occurs in the string. The construction is recursive: at each stage the two trees which account for the least total frequency in their root nodes are selected, and used to produce a new binary tree. This has, as its children the two trees just chosen: the root is then labelled with the total frequency accounted for by both subtrees, and the original subtrees are removed from the forest. The construction continues in this way until only one tree remains; that is then the Huffman encoding tree.[7]

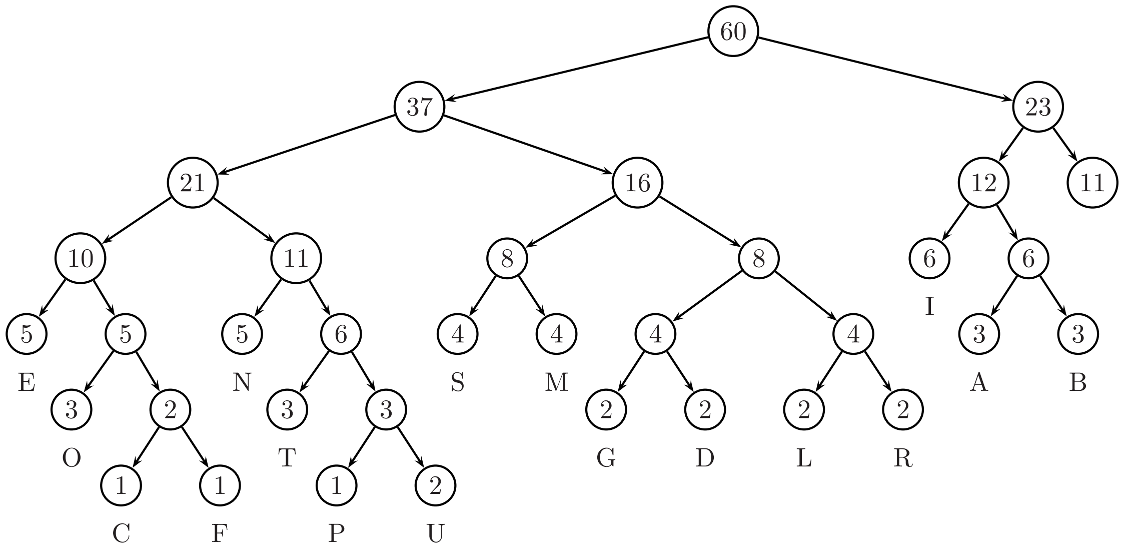


Figure 3.7: The Huffman encoding tree for the string "A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS".

The resulting Huffman encoding tree for our example string is shown in Fig 3.7. By construction, the symbols only occur at leaf nodes, and so the corresponding code has the prefix property. In the diagram, these leaf nodes still carry the frequencies used in their construction; formally once the tree has been built, the symbols which are shown below the leaves should replace the frequencies at the nodes. The right-most node is the symbol ␣. As already described, the character encoding is the read by traversing from the root to each leaf, recording "0" if the left hand branch is traversed, and "1" if the right hand one is taken. Thus "S" is encoded as "0100", while ␣ is "11" and "C" is "000110".

Definition 3.6. Let T be a tree with weigths $w_1, \dots w_n$ at its leaf nodes. The *weighted leaf path length* $L(T)$ of T is

$$L(T) = \sum_{i \in \mathrm{leaf}(T)} l_i w_i \tag{3.2}$$

where l_i is the **path length**; the length of the path from the root to node i.

[7]There are choices involved, so the Huffman tree is certainly not unique. It is thus more accurate to speak about *a* Huffmann tree; we shall do this in future.

I	A	B	D	M	E	O	C	F	G	S	T	L	R	N	P	U	␣
6	3	3	2	4	5	3	1	1	2	4	3	2	2	5	1	2	11

Table 3.2: Symbol frequencies used to build a Huffman Code.

We are interested in the case when the tree is an encoding tree and the weights are the frequency of occurrence of the symbols associated with the leaf nodes. In that case $L(T)$ is the length of the message after encoding, since at node i, the character occurs a total of w_i times, and requires l_i bits to encode it. We now show that a Huffman encoding tree gives the *best* encoding. Say that a binary tree T is *optimal* if $L(T)$ has its minimum value over all possible trees with the same set of leaf nodes.

Theorem 3.7. A Huffman tree is optimal.

Proof. We start with an apparently simple result.

Lemma 3.8. Suppose that nodes with weights w_i and w_j satisfy $w_i < w_j$. Then in an optimal tree the corresponding path lengths l_i and l_j satisfy $l_j \leq l_i$.

Proof. This is just the assertion that nodes that occur infrequently have long codes. Suppose we have a tree T associated with a given set of nodes having weights $\{w_k\}$, and that for some pair i, j, we have $w_i < w_j$ but $l_i < l_j$. Then

$$(w_j - w_i)(l_j - l_i) > 0 \quad \text{and so} \quad w_j l_j + w_i l_i > w_j l_i + w_i l_j.$$

Consider now the effect on the weighted leaf path length $L(T)$ of interchanging the weights on nodes i and j. The new weighted leaf path length is

$$L(T) - (w_j l_j + w_i l_i) + (w_j l_i + w_i l_j) < L(T).$$

Thus T was not optimal, since the new tree has a smaller weighted leaf path length. □

Lemma 3.9. Suppose that nodes in an optimal tree have weights w_i labelled so that $w_1 \leq w_2 \leq \cdots \leq w_n$. Then by relabelling if necessary subject to this constraint, we can also have $l_1 \geq l_2 \geq \cdots \geq l_n$.

Proof. Suppose conversely that $i < j$ but $l_i < l_j$. Since $i < j$, we have $w_i \leq w_j$. However if $w_i < w_j$, then by Lemma 3.8 we have $l_j \leq l_i$ since we are assuming the tree is optimal. But $l_j > l_i$, showing that we must have $w_i = w_j$. There is thus no loss if we interchange the labels i and j. We can continue to do this until we achieve the required consistent labelling of the corresponding node lengths. □

We can now show that a Huffman tree is optimal. This argument was adapted from Gersting (1993, Page 402). We establish the result by induction on the number n of leaf nodes in the tree. The result is clear for $n = 1$.

Next note that in any optimal binary tree, there are no nodes with single children — replacing the child by the parent produces a shorter weighted external path length.

Consider now a set of $n+1$ weights w_i with $n+1 \geq 2$, which by Lemma 3.9 we suppose to be ordered such that $w_1 \leq w_2 \leq \ldots \leq w_{n+1}$ in such a way that the corresponding paths lengths satisfy $l_1 \geq l_2 \geq \cdots \geq l_n$. Let T_{n+1} be an optimal tree for these weights with weighted leaf path length $L(T_{n+1})$. By our choice of labelling w_1 occurs on the longest path, as does its sibling w_j; since they are siblings, $l_1 = l_j$. Since $l_i \geq l_2 \geq l_j$, we have $l_1 = l_2$. Thus the new tree T'_{n+1} obtained by interchanging nodes 2 and j have the same weighted external path length.

Next construct a new tree T_n with a new "leaf" node $w = w_1 + w_2$ by combining w_1 and w_2 from T'_{n+1} to give a tree with n leaf nodes. Let H_n be a Huffman tree on these nodes, and note that, by construction the tree obtained by replacing the single node w in H_n by the nodes w_1 and w_2, the two smallest weights, is a Huffman tree H_{n+1}. By induction, we have $L(T_n) \geq L(H_n)$, since they have the same leaf nodes. We now calculate:

$$L(T_n) + w_1 + w_2 = L(T_{n+1}) \quad \text{and} \quad H(T_n) + w_1 + w_2 = H(T_{n+1}).$$

Thus $L(T_{n+1}) \geq L(H_{n+1})$. But T_{n+1} was optimal, so this inequality is an equality and H_{n+1} is optimal, as required. □

Remark 3.10. The above proof has to be done a little carefully. The (complete) binary tree having nodes with weights 1, 3, 2 and 2 is *not* a Huffman tree, but is optimal; however interchanging the second and third nodes does not affect the weighted leaf path length and *does* give a Huffman Tree. In the proof, this interchage is the step of creating T'_{n+1} from T_{n+1}

A little more can be said, centred round the need to have the coding tree available when decoding. Of course, for "general purpose" language, the letter frequencies are well known, and could be assumed. In general, the need to transmit the coding tree as well as the message reduces the effectiveness of the method a little. And it can be impractical to preprocess a message to get the exact frequencies of the symbols before any of the message is transmitted. There is a variant however, called *adaptive* Huffman coding, in which the frequencies are assumed initially to be all the same, and then adjusted in the light of the message being coded to reflect the actual frequencies. Since the decoder has access to the same information as the encoder, it can be arranged that the decoder changes coding trees at the same point as the encoder does; thus they keep in step. One way to do that is to introduce extra "control" symbols to the alphabet, and use these to pass such information.

Modern coding schemes such as zip (or gzip or pkzip) are based on these ideas.

3.3 Heapsort

So far we have argued that algorithms can often be best expressed in terms of an Abstract Data Type; in this section we reveres the process, and show how, thinking in terms of an ADT can give an algorithm. Implementing the algorithm is then the same as implementing the Abstract Data Type. Our problem is again one of sorting, but this time we base the algorithm on the priority queue of Section 3.1.5. Since we are sorting we have a natural priority associated with each item, namely the value that determines its sort order.[8] The algorithm is then trivial: we add elements as they are given to the priority queue, and when the queue is full, simply retrieve them in order.

Of course this begs many questions; certainly we can't get a useful algorithm without discussing an implementation. We thus chose a representation of a priority queue as a complete binary tree (Section 3.2.3) in which each node contains an element and its associated key or priority. In addition, we assume that these keys satisfy the **heap condition**, that:

> at each node, the associated key is larger than the keys associated with either child of that node.

Note that the node with the highest priority is necessarily at the root of the tree, thus we can clearly represent a one element priority queue in this way. We now show that the required operations of addition and removal can be done in such a way that the heap condition is still satisfied. Rather than give formal definitions of the operations, we illustrate them with an example, so consider a priority queue whose keys are the letters from the string "A SORTING EXAMP". We associate a priority with alphabetical order, with letters at the end of the alphabet being given the highest priority. The corresponding tree is drawn with each key used to label the associated node. Assume for the moment that Fig. 3.8, without the letter "P" represents an intermediate stage in the construction; you can certainly check that it satisfies the heap condition, and consider the situation when the letter "P" is added to this priority queue formed from the earlier letters from our string. The result is the complete binary tree shown in Fig. 3.8. To maintain the tree as a complete binary tree, there was no choice about where the new node was created, and the new tree no longer satisfies the heap condition. To restore that, we allow interchanges which permit the new node, with key "P" to move up the tree, swapping places with elements whose priority is lower. Doing this set of swaps gives Fig. 3.9, which now does satisfy the heap condition.

[8]Formally we have only considered comparison sorting up until now, but the passage from that to the "equivalent" problem of ordering a permutation accepted the existence of a global priority.

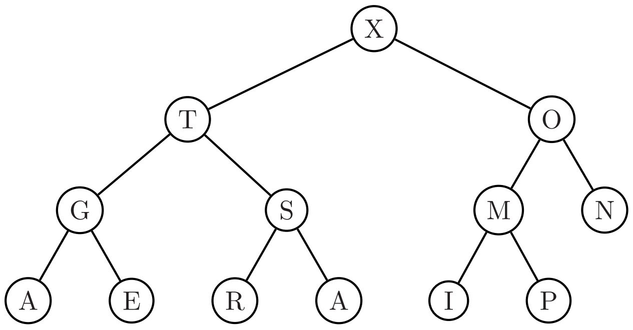


Figure 3.8: Adding the element “P” to a priority queue.

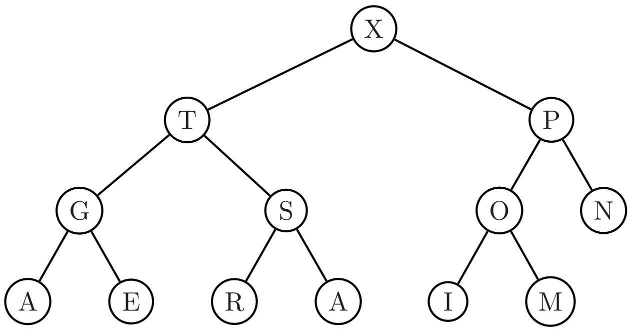


Figure 3.9: The resulting priority queue, after the heap condition is restored.

The “remove” operation is similar; we first get the shape right, and then restore the heap condition. Since removing an element leaves a complete binary tree with one fewer element, the first step can be done by removing the root element, and replacing it by the last element in the tree. This is shown in Fig. 3.10, where the `remove` operation is applied to the priority queue of Fig 3.9. In order to restore the heap condition, the new root element must be allowed to “drop down” the tree until the tree satisfies the heap condition again. At each stage, if the heap condition is violated, the root element is swapped for the larger of its two children, thus restoring the heap condition at that level. the resulting priority queue is shown in Fig 3.11.

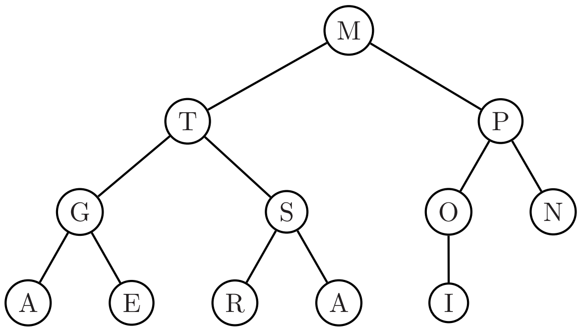


Figure 3.10: Removing the element “X” of highest priority from Fig. 3.9.

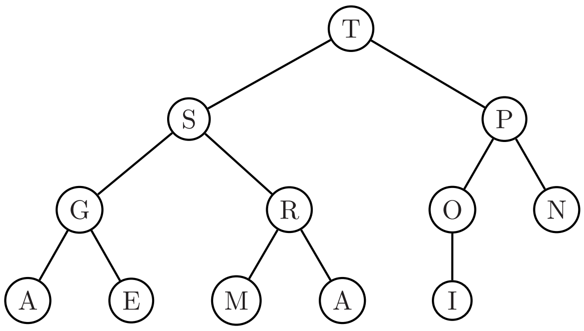


Figure 3.11: The resulting priority queue, after the heap condition is restored.

Finally note that in Example 3.5 we presented an array-based implementation of a complete binary tree in which moving up and down the tree was efficient. Certainly both the “add” and “remove” operations can be implemented in a time proportional to the depth of the tree, and hence in time $O(\log n)$ where n is the number of elements to be sorted. Since we do a total of $2n$ such operations in a complete implementation of Heapsort, we thus get a total time of $O(n \log n)$.

We already saw in Section 2.6 that we were not going to improve on this order of magnitude. And in fact the time constant is worse than that for quicksort by a factor of about 2, so quicksort remains the method of choice. So what is special about Heapsort? Note that we *always* complete the “add” and “remove” operations in $O(\log n)$ time; there is no “worst case”. Thus unlike quicksort, the worst case running time of Heapsort is the same as its average running time.

Questions 3 (Hints and solutions start on page 92.)

Q 3.1. Show how a queue can be implemented using two stacks. How many different stack operations are needed to implement each `add` or `next` operation of the queue?

Q 3.2. Show how to implement a queue using an array. Recall the command `i rem N`, which returns the remainder when i is divided by N.

Q 3.3. Assume now that at least half the tokens in the 'three coloured flags" array are red. Describe an algorithm which uses fewer calls to the `swap` routine.

Q 3.4. The operation of "flattening" a list is the removal of any sublist structure, while keeping all the elements. This the flattening of `[A [B C [[D] E] F] G]` is just `[A B C D E F G]`. Show that the flattening of the list representation of a tree is one of the preorder, inorder or postorder traversal. Which one?

Compute each of the traversals of the tree whose list representation is
`[A [[B [[E] [F] [G] [H]]] [C [I]] [D]]]`.

Q 3.5. Show that the definitions of a binary tree in terms of terminal nodes and in terms of left and right children, are equivalent.

Q 3.6. Show that the priority queue representation shown in Fig.3.9 is the one obtained by using the "add" operation described above on the first 13 letters of the string "A SORTING EXAMPLE" one by one in the given order to an empty priority queue. Continue the process to give the priority queue using all 15 letters.

Q 3.7. Verify the construction used to produce the Huffman encoding tree for the string "A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS".

Q 3.8. Describe a *Huffman* code and give examples of circumstances when it (or a variant) might be used. Illustrate your answer by constructing a binary Huffman code for the string

A TEST EXAMINATION ANSWER

As well as deriving the coding tree, you should give your code for the word ANSWER. Note that the string contains 12 distinct symbols, include the "space" symbol, and 25 symbols in all.

In what sense is a Huffman code optimal?

Q 3.9. Give a brief description of a priority queue and a complete binary tree. What does it mean to say that a complete binary tree satisfies the *heap condition*?

Describe the use of Heapsort to sort the string

HEAPSORT

Give the full construction of the heap in detail, and give details of the first step in its use.

Does Heapsort have any advantages over Quicksort?

Q 3.10. Describe a *Huffman* code and give examples of circumstances when it (or a variant) might be used. Illustrate your answer by constructing a binary Huffman code for the string

A HUFFMAN EXAMPLE PLEASE

As well as deriving the coding tree, you should give your code for the word PLEASE. Note that the string contains 12 distinct symbols, include the "space" symbol, and 24 symbols in all.

An alphabet which contains 2^n letters and and their associated probabilities is given and the corresponding symbols are derived using the Huffman construction. How *short* can the *longest* of these symbols be? What property of the corresponding probabilities would ensure that this minimum length is attained. How *long* can the *longest* symbol be? Give restrictions on the corresponding probabilities to ensure that this maximum length occurs.

Chapter 4

Grammers and Parsing

A good source of material relevent to this section is (Sedgwick 1995); in my edition, there are chapters on “String Searching”, Pattern Matching” and “Parsing” all of which are of interest.

If you type the following line in a FORTRAN program

```
X = (A + B)*C - D
```

you expect the computer, when running the program, to take the values of A and B, add them, multiply the answer by the value of C and then subtract the value of D — finally putting the result in X. How does the computer work out that that is what you want to happen? How does it ‘read’ the expression on the RHS? Equally, how does it know that the following is rubbish?

```
X = (A + B)C - * D))
```

The FORTRAN *compiler* has the job of making sense of what you have written and converting it into a series of machine operations. You will know by now that FORTRAN, like all other computer languages, requires you to write your programs in a very strict and formalised style. It imposes a *syntax* or *grammar* on you which you cannot ignore in any way (the compiler will accept `A*(B+C)` but not `A(B+C)`, despite the standard mathematical convention).

The general problem we face is this: the compiler (or compiler writer) defines a certain syntax for ‘acceptable statements’. You have to write your programs in this syntax. The compiler then has to be able to ‘decode’ what you have written so as to find out what to do. It also has to be able to decide whether what you have written actually *is* grammatical.

To study this problem it helps to introduce the concept of a Formal Language and the related concept of a Syntax (a Chomsky Transformational Grammar, to be precise). The definitions of these concepts are going to be rather abstract, but some examples should make things fairly clear.

4.1 Formal Languages

To define the concept of a Formal Language we need the ideas of an *Alphabet* and a *string*.

An **alphabet** is a finite set A. That’s all.

A **string** from the finite set A is just a finite ordered list of elements of A. Say that $\alpha = (a_1, a_2, a_3, \ldots, a_n)$ is a string from A if $a_i \in A$ for $i = 1, \ldots, n$. For historical reasons we usually use the notation $\alpha = a_1 a_2 a_3 \ldots a_n$ for a string. The *length* of a string is the number of things in the list. We usually add in one more string, the *empty string* ϵ which has nothing in it at all and has length 0.

We use the following notations: A^n is the set of all strings of length n from A. A^+ is the set of all strings from A, excluding the empty string. A^* is the set of all strings from A, including the empty string.

$$A^+ = \bigcup_{n\geq 1} A^n \qquad A^* = A^+ \cup \{\epsilon\}$$

I will use the obvious notation a^n for a string of n successive a's.

A **Formal Language** on the alphabet A is a subset $L \subset A^*$.

Bet that surprised you. Years of work by linguists and philologists and all we get is one idiot sentence as a definition. Don't be too put out, the real substance is yet to come. All we are saying so far is that our 'languages' are collections of strings of items. If you think in terms of a language like English the set A is not really the 'alphabet' that you learned in school, but the set of all *words* in the dictionary. The basic item is the word (or, more properly, stem). The 'strings' that we are talking about are then the *sentences* of English. From this point of view English is defined as a subset of all possible sentences formed from words from the (English) dictionary. Sentences like *"the cat sat on the mat"* are in this set but sentences like *'cat sat the on mat the"* are not.

The big question, of course, is: what is the subset? Which sentences are English and which are not. We cannot, in any practical sense, answer this question by *listing* the elements of the set because it is infinite. English allows constructions like: *I saw a cat and a dog and a giraffe and a banana and two sheep and three sheep and four sheep and . . . .* So there is no limit to the length of an English sentence. Instead, we would like to be able to specify the subset by its *properties.* The usual way to do this is to state a *grammar* of some kind — a set of rules which tell you what you are or are not allowed to do. That is going to be the next thing that we look at.

4.2 Formal Grammars

Suppose we have a Formal Language. That is to say, we have an Alphabet A and a certain subset $L \subset A^*$. How do we set about *defining L*? There are many approaches to this but one has become very standard in recent years, particularly in computer science. This approach is due to the well-known linguistic theorist Noam Chomsky (1959). He developed this approach in studying 'natural' languages, but it has since been taken over in a big way to describe computer languages. Chomsky's basic idea was to think of grammars as describing various possible *transformations* that can be performed on the sentences of a language. The application of chains of these transformations should be able to 'generate' all the sentences of the language, even though the list of transformations is itself finite. These transformations form the 'deep-structure' of the language.

To present a very trivial example, the basic notion is the **sentence**. The abstract object called a **sentence** can be transformed into the list **(subject) (verb) (object)**. The abstract object called a **subject** can be transformed into the list **(article) (noun)**. The items of this list can than be transformed into the *concrete* items: *(the)(cat)*. This type of process gives us the following 'generative structure' of the sentence *the cat ate the banana* which is illustrated in Fig 4.1.

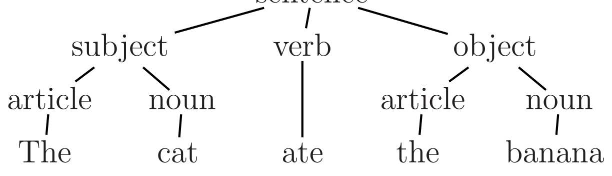


Figure 4.1: The generative structure of a sentence.

Let me now define what I mean by a Formal Grammar (or transformational grammar, or Chomsky grammar).

A **Formal Grammar** consists of:

- a finite set A — the Alphabet;
- a finite set $\mathcal{A}$ — the abstract Alphabet;
- an element $\sigma \in \mathcal{A}$ — the initial symbol; and
- a finite set P of *productions* of the form $\alpha \to \beta$ where $\alpha \in (A \cup \mathcal{A})^+$ and $\beta \in (A \cup \mathcal{A})^*$.

This needs some explaining. The alphabet A is something that we have already met (the set of words in English, for example). The abstract alphabet consists of those abstract objects like **sentence**, **noun** and **adjectival phrase** that occur in the partial transformation of a sentence. The initial symbol is the starting point for our transformations. In our example of English the initial symbol is **sentence**.

The Productions are the transformational rules, like

subject → **article noun**
or **noun** → *banana*.

Notice that our general definition allows mixtures of both concrete and abstract symbols on both sides of →, though there must be some abstracts on the left-hand side.

The **Formal Language** defined by a Formal Grammar is just the set of all strings in A^* that can be derived by using these productions (over and over again).

Let me use the notation $\alpha \Rightarrow \beta$ to mean that the string $\beta \in (A \cup \mathcal{A})^*$ can be derived, using the rules or productions of the formal grammar from the string α.

At this point I had better say goodbye to English and all other natural languages because their grammars are hugely complicated[1] and not fully understood. I will stick to the much simpler grammars that might occur in computer languages.

4.3 First Example

Consider the following formal grammar:

- The alphabet is $A = \{\mathrm{a}, \mathrm{b}\}$.
- The abstract alphabet is $\mathcal{A} = \{\sigma\}$.
- The initial symbol is σ.
- The productions are
  1. $\sigma \to \mathrm{a}\sigma$;
  2. $\sigma \to \sigma\mathrm{b}$;
  3. $\sigma \to \mathrm{a}$; and
  4. $\sigma \to \mathrm{b}$.

This is certainly a Formal Grammar. The obvious question to ask is: what language does it define? Which strings of a's and b's can we produce with these productions? Let me call the language L.

The best thing to do at this stage is usually to mess about a bit applying the rules just to see what kind of pattern emerges, if any. Obviously 'a' and 'b' are in L. If we apply (1) and then (4) we get 'ab' in L. If we apply (1) twice we get $\sigma \Rightarrow \mathrm{aa}\sigma$. This is still 'abstract', in that it still contains elements of the abstract alphabet. If we now apply (3) we get 'aaa' in L.

Let me now prove a theorem which will tell us what L looks like.

[1]To remind you of just how hard they are, think of parsing "Time flies like an arrow"; you should find three completely separate interpretations, one each with "time" as a noun, a verb and an adjective.

Theorem 4.1. $L = \{\mathrm{a}^n\mathrm{b}^m : n + m > 0\}$

Proof. Any string that we can generate by the rules, starting from σ, can contain at most one σ (look at (1) and (2). If it contains no σ's then it is fully concrete and a string of L.

So any application of (3) or (4) will make our string concrete and stop the process. This means that we generate strings in L by arbitrary combinations of (1) and (2) followed by precisely one application of either (3) or (4).

Now (1) followed by (2) gives $\sigma \to \mathrm{a}\sigma \to \mathrm{a}\sigma\mathrm{b}$ while (2) followed by (1) gives $\sigma \to \sigma\mathrm{b} \to \mathrm{a}\sigma\mathrm{b}$.

Note that these produce the same result. So the order of application of (1) and (2) does not matter. Thus we can simplify our earlier statement to saying that any string in L can be produced by a certain number of applications of (1), possibly zero, followed by a certain number of applications of (2), possibly zero, followed by either (3) or (4). You can easily check that this gives us

$$\sigma \to \mathrm{a}^i\sigma \to \mathrm{a}^i\sigma\mathrm{b}^j \to \mathrm{a}^{i+1}\mathrm{b}^j \quad \text{or} \quad \mathrm{a}^i\mathrm{b}^{j+1} \qquad (i, j \geq 0)$$

Thus at least one index has to be non zero. That proves the theorem. □

Now that we know what the language is we can go further and try to produce a 'parsing algorithm' for it. We want to write an algorithm which, given any string from A^* will decide whether or not it is in L and, if so, what is its structure (what are n and m?). This is very easy in this case. In more complicated languages, where we cannot find a neat description of the strings in the language, it can be very difficult. Indeed it can easily become impossible.

In fact the parsing algorithm is so easy in this case that I will leave it to you to write; see question 4.1

4.4 Second Example

Consider the following Formal Grammar:

- The alphabet is $A = \{\mathrm{a}, \mathrm{b}\}$.
- The abstract alphabet is $\mathcal{A} = \{\sigma\}$.
- The initial symbol is σ.
- The productions are
  1. $\sigma \to \sigma\sigma\mathrm{b}$; and
  2. $\sigma \to \mathrm{a}$;

This is actually a slightly simplified case of a very important grammar indeed. We will meet it again later in a more serious context. Again, let me call the resulting language L.

If you experiment a bit with this grammar you will probably get confused! The structure is not immediately obvious. So let me jump in straight away with the answer.

Consider the following property of strings in A^+:

Property P Let $\alpha \in A^+$ satisfy the following two properties:

1. Up to any point in α there are more a's than b's.
2. The number of a's in α is one more than the number of b's.

I claim that a string $\alpha \in A^+$ is in L iff it satisfies property P. This is very nice because, once more, property P is very easy to check

Theorem 4.2. If $\alpha \in L$ then α satisfies property P.

Proof. We use induction on the length of α. The only string in L of length 1 is 'a', which satisfies P. Suppose inductively that all strings in L of length $\leq n$ satisfy property P and that $n \geq 1$.

Let $\alpha \in L$ have length $n + 1$. Since $\sigma \Rightarrow \alpha$ and $\alpha \neq a$ we must have started its generation with the production $\sigma \to \sigma\sigma\text{b}$. So α must have the overall structure $\alpha = \beta\gamma\text{b}$ where $\beta, \gamma \in L$. Also β and γ both have length $\leq n$.

So, by the induction hypothesis, β and γ must satisfy P. This tells us all we need to know about α. It is clear that α satisfies P1 up to the end of β. The a's therefore start γ in profit and therefore stay in profit till the end of γ since γ satisfies P1. Furthermore, if there are k a's in β and h a's in γ then α has $k + h$ a's and $(k - 1) + (h - 1) + 1 = (k + h - 1)$ b's. So P1 is satisfied up to the end and P2 is also satisfied. Work this through for an example if you find it confusing.

So any string of length $n + 1$ in L satisfies P. So, by induction, all strings in L satisfy P. □

Theorem 4.3. If $\alpha \in A^+$ and α satisfies P then α is in L.

Proof. We use induction once more. The only string in A^+ of length 1 that satisfies P is 'a' and this is in L. Suppose inductively that all strings of length $\leq n$ that satisfy P are in L.

Let α be a string of length $n + 1$ which satisfies P. α starts with at least two a's ($n \geq 1$). It must contain a 'b'. So, somewhere, it contains the sequence 'aab'. So we can write

$$\alpha = \lambda aab\mu$$

Now let $\beta = \lambda a\mu$. You can easily check that, since α satisfies P, β satisfies P. Also β has length $< n$. So, by the induction hypothesis, β is in L.

The only way to get an 'a' into a string in L is to use rule (2). So we must have

$$\sigma \Rightarrow \lambda\sigma\mu \to \lambda\text{a}\mu$$

But, by rule (1),

$$\sigma \Rightarrow \lambda\sigma\mu \to \lambda\sigma\sigma\text{b}\mu \to \lambda\text{aab}\mu = \alpha$$

So $\sigma \Rightarrow \alpha$ and α must be in L.

So all strings of length $n+1$ in A^+ which satisfy P are in L, and thus by induction, all strings in A^+ that satisfy P are in L. □

4.5 Arithmetic Expressions

Let's now go back to where we started. Consider the strings that occur on the right-hand side of equals signs in programs — what are known as Arithmetic Expressions. I mean things like

$$a + b + c * d, \qquad a * (b - c/d) - e/f, \qquad a * b * ((c - d) * a - p * (u - v) * (r + s))/t$$

These expressions are made up of *variable names*, *binary operators* and *brackets*. There are many other things that can occur in such expressions in actual computer languages. For example, powers (**), unary minus ($-a + b$), actual numbers ($1.23 * a - 3.45$), and things like function and array references ($\sin(a + b) - p(4)$). We are going to ignore these complications (they are no more than complications) and concentrate on expressions made up of variables, the four basic binary operators and brackets. We also ignore *spaces*. Let us restrict things further by assuming that there are just 26 possible variables — the lower-case letters.

We are dealing with a language of some kind. An expression is a string of symbols from a certain alphabet. Some such strings are 'grammatical', as in the above examples, and others are not — for example

$$a + (b + \qquad a(b + c) \qquad ((+ + a - b - (($$

What is the underlying grammar. What decides whether a given string is or is not grammatical?

Let me now introduce the syntax for such **Arithmetic Expressions** (AE's). It produces the 'right' answers.

4.5.1 AE Grammer

The grammar for Arithmetic Expressions is as follows:

- The alphabet is $A = V \cup B \cup P$ where
  $V = \{a, b, c, \ldots, z\}$, B ={+,-,*,/} and P ={(,)};
- The abstract alphabet is $\mathcal{A} = \{$ **expr** , **var** , **op** $\}$
- The initial symbol is **expr** .
- The productions are
  1. **expr** → **expr** **op** **expr** ;
  2. **expr** → (**expr**);
  3. **expr** → **var** ;
  4. **var** → $a|b|c|\ldots|y|z$; and
  5. **op** → $+|-|*|/$

Note the way I have written (4) and (5). In fact (4) is actually 26 separate production rules. The vertical bars indicate alternatives.

At this point you probably expect me to *prove* that this grammar is correct — that it actually produces the right kind of expression. We cannot do this for the simple reason that we have not got any other definition to compare it with! All we have is a certain common-sense about what is or is not correct. If you play around with the grammar a bit and try to see what it is doing you will soon convince yourself that it is indeed doing the right things and following the right principles.

As an example, consider the expression $((a + b) * c + d)/a$. Can we derive this from the grammar? We can, as shown in Table 4.1, which should give you the basic idea.

expr	→	**expr** **op** **expr**	(1)
	→	**expr** / a	(5)(3)(4)
	→	(**expr**) / a	(2)
	→	(**expr** **op** **expr**) / a	(1)
	→	(**expr** + d) / a	(5)(3)(4)
	→	(**expr** **op** **expr** +d)/a	(1)
	→	(**expr** * c + d)/ a	(5)(3)(4)
	→	((**expr**)*c + d)/ a	(2)
	→	((**expr** **op** **expr**)*c + d)/ a	(1)
	→	((a + b)*c + d)/ a	(5)(3)(4)

Table 4.1: Deriving $((a + b) * c + d)/a$ from our grammer

Let's try another problem. How do we show that $(a + (b-))$ is *not* in the language? This is intrinsically more difficult. We have to show that there is no possible sequence of transformations that will produce this result. This particular example is not too difficult. We can see where the 'bad' bit comes — the $(b-)$. How do we show that this sequence of symbols cannot happen. What I will prove is that there is no way that our grammar will allow an **op** symbol to be followed by a right bracket.

Firstly, the only way to get the minus sign is by using (5) to replace the abstract symbol **op** . The only way we could have got the **op** symbol there in the first place was by using (1) — which is the only rule with **op** involved on the RHS. So, when **op** was first introduced it was followed by **expr** . So now the question becomes: could **expr** generate a string *starting* with a ')'? The answer to this is no. Rule (1) changes **expr** into something that starts with the abstract symbol **expr** , rule (2) changes **expr** into something that starts with a left-bracket and the only other relevant rule is (3), which produces a **var** which certainly cannot transform into a right-bracket. So our expression is not in the language.

That was rather long-winded, but I was trying to be precise. It has, in effect, produced a general theorem for us: no string in this language has a symbol from B followed by a right-bracket. We can produce lots of other theorems like this. In fact, I suggest that you try to prove the results embodied in Table 4.2, which indicates whether or not a given symbol can be followed by another given one in a grammatical expression. As an example, it shows that a symbol from B *cannot* be followd by a symbol from B, but can be followed by "(".

	→ V	→ B	→ (	→)
V	n	y	n	y
B	y	n	y	n
(	y	n	y	n
)	n	y	n	y

Table 4.2: Possible follow-on symbols.

4.6 Postfix Expressions

There is another syntax for arithmetic expressions that you may have met before if you have used a rather old-fashioned pocket calculator. This is called Postfix notation, or Reverse Polish (as in Pole, not table wax).

The expression $a + b * (c - d) + e$ written in postfix form is $abcd - * + e+$. This looks rather cryptic. You can get the idea most easily if you read it backwards as " add e to (add (times $(c - d)$ and b) to a)". A few more examples are shown in Table 4.3 which may serve better than an elaborate explanation:

a+(b+c)	→	abc++
(a+b)+c	→	ab+c+
a-b*c	→	abc*–
(a/b)*(c/d)	→	ab/cd/*
a/(b+c*d-e)	→	abcd*+e–/
a-b*c+d/e	→	abc*–de/+

Table 4.3: Converting expressions to postfix notation.

I hope that makes sense. Notice that the translation of expressions that are not fully bracketed (e.g. $a + b + c$) is ambiguous — but that does not matter.

4.6.1 Postfix Grammer

The Grammar for Postfix Expressions is as follows:

- The alphabet is $A = V \cup B$ where $V = \{a, b, c, \ldots, z\}$ and $B = \{+, -, *, /\}$
- The abstract alphabet is $\mathcal{A} = \{$ **expr** , **var** , **op** $\}$
- The initial symbol is **expr** .
- The productions are
  1. **expr** → **expr** **expr** **op** ;
  2. **expr** → **var** ;

3. **var** $\rightarrow a|b|c|\ldots|z$; and
4. **op** $\rightarrow +|-|*|/$

Note that, apart from the lack of brackets, the significant change is the *order* of the abstracts in (1). This grammar is actually a simple generalisation of the grammar that we studied in Example 2. Because of this it is now easy to prove the following result:

Theorem 4.4. A string α in A^+ is a postfix expression iff it satisfies the following property: there is one more **var** symbol in α than **op** symbols and up to any point in α there are more **var** symbols than **op** symbols.

You can prove it as an exercise (it just means making minor modifications in the corresponding proofs in Example 2).

So we can now write a simple 'syntax-checking' algorithm to decide whether or not a given string is in postfix form. (See below for explanations of isvar and isop.)

```
algorithm  postfix-checker(s,n)
// check the string $s_1,\ldots,s_n$ to see if it is postfix.
begin
  count = 0
  if (n = 1) then begin
    if isvar($s_1$) then begin
      return POSTFIX
    else
      return NOT POSTFIX
    end
  end
  for i=1 to n-1 begin
    if isvar( $s_i$ ) then count = count + 1
    if isop( $s_i$ ) then count = count - 1
    if count <= 0 then return NOT POSTFIX
  end
  if isop( $s_n$ ) and count $\neq$ 2 then return NOT POSTFIX
  if isvar( $s_n$ ) then return NOT POSTFIX
  return IS POSTFIX
end.
```

This routine uses the *predicates* `isvar(x)` and `isop(x)`. A predicate is just a function that returns the logical values TRUE or FALSE. The predicate `isvar(x)` will return TRUE iff `x` is a **var** symbol. The predicate `isop(x)` will return TRUE iff `x` is an **op** symbol.

The routine will work with much more general alphabets than I have been using, so long as `isvar()` and `isop()` are suitably redefined. In many computer languages the operator list is a bit longer, often including a power operator (like ** in FORTRAN). Some languages even allow you to define your own binary operators. In almost all computer languages the alphabet for **var** is much bigger, usually consisting of all strings of up to a given number of characters from a set which certainly includes all the letters and digits and may include a lot more.

None of this has any important effect on what I have been doing. The restriction to a simple alphabet was a matter of convenience rather than a simplification of the theory. The more complicated you make your alphabets the more difficult it becomes to write the `isvar()` predicate — but that is somebody else's problem.

4.7 Converting Infix to Postfix

The expression syntax that we started with (the usual one) is called an *infix* system, because the binary operator comes *between* its operands ($a + b$). The other system is called *postfix* because the operator comes *after* its operands (ab+). To every infix expression there corresponds a postfix expression that has the same effect. The converse is not true because, as we have seen, there is an ambiguity. The infix expression $a + b + c$ can be represented as either abc++ or ab+c+ in infix. The reason for the ambiguity is the lack of brackets in the infix expression. If the infix expression were 'fully bracketed' there would be no ambiguity. Thus $(a + b) + c$ pairs with ab+c+ and $a + (b + c)$ pairs with abc++.

I now want to look at the problem of converting infix expressions to postfix expressions. I will first simplify the problem by only considering 'fully bracketed' infix expressions. I will then look at the more difficult problem of handling general infix expressions.

The 'fully bracketed' version of an expression like $a + b * c/d$ is $(a + ((b * c)/d))$. Note at once that there is a very serious problem here. The above expression should not be bracketed as $((a + b) * (c/d))$ — that's something entirely different. This is where we meet the concept of *operator precedence*. You do multiplications *before* additions, and so on. The main reason for restricting to fully bracketed expressions is to avoid this problem.

Let me now introduce the syntax for **Fully Bracketed Infix Expressions** (fbie's).

4.7.1 FBIE Grammer

The Grammar for Fully Bracketed Infix Expressions is as follows:

- The alphabet is $A = V \cup B \cup P$ where
  $V = \{a, b, c, \ldots, z\}$, B ={+,-,*,/} and P ={(,)}
- The abstract alphabet is $\mathcal{A} = \{$ **expr** , **var** , **op** $\}$
- The initial symbol is **expr** .
- The productions are
  1. **expr** $\rightarrow$ (**expr** **op** **expr**)
  2. **expr** $\rightarrow$ **var**
  3. **var** $\rightarrow a|b|c|\ldots|z$; and
  4. **op** $\rightarrow +|-|*|/$

The difference between this and our previous infix syntax is that the original (1) and (2) have been compressed into the single production (1) which *forces* the presence of brackets.

The following are examples of *fbie*. You should check that they are derivable from the above grammar.

$$((a + b) - c) \qquad ((a * b)/((b - d)/e)) \qquad ((((a + b) * c) - d) + e)$$

I now want to present a conversion algorithm which will take a *fbie* and convert it into the corresponding postfix expression. As well as being of interest in its own right, it is also a good example of the use of a stack, introduced in Section 3.1.1. Given this construct, and an assumed implementation, the conversion routines are natural:

```
algorithm fbie-to-postfix(s,n)
// convert the fbie s of length n into postfix.
begin
  let ans be empty string
  initialise the stack
  for i = 1 to n begin
    if isvar($s_i$) then append $s_i$ to ans
    if isop($s_i$) then push($s_i$)
    if $s_i$ = ( then ignore
    if $s_i$ = ) then pop stack and append result to ans
  end
  return ans as result
end.
```

The action of the routine is to pump out the **var** symbols as they come in, as it should, but to put **op** symbols onto the stack — the pending tray — until a right bracket comes along to close off the latest (**expr** **op** **expr**) sequence. Then an **op** symbol is pumped out.

This is a very raw version of the routine. It would be better if the routine could pick up the fact that the input was not a proper *fbie* and stop with a useful error message, rather than crashing or producing a result that was not correct. The 'crash', if it occured, would be produced by an attempt to pop an empty stack (or trying to push onto a full stack — but that's an implementation fault rather than a logical one).

Let's work through an example. Suppose we want to translate

$$s = (((a + b) * (e - f)) + g)$$

Following the algorithm gives the trace shown in Table 4.4.

	item	action	ans	stack
1	(	ignore	empty	empty
2	(	ignore	empty	empty
3	(	ignore	empty	empty
4	a	append	a	empty
5	+	push	a	+
6	b	append	ab	+
7	)	pop & append	ab+	empty
8	*	push	ab+	*
9	(	ignore	ab+	*
10	e	append	ab+e	*
11	–	push	ab+e	*–
12	f	append	ab+ef	*–
13	)	pop & append	ab+ef–	*
14	)	pop & append	ab+ef–*	empty
15	+	push	ab+ef–*	+
16	g	append	ab+ef–*g	+
17	)	pop & append	ab+ef–*g+	empty

Table 4.4: Translating the fbie $s = (((a + b) * (e - f)) + g)$

The result is ab+ef-*g+. Note that the stack is empty at the end. Logically, it must be. If it is not then s was not a *fbie*. Again note that the action of the routine is to put **var** symbols into the answer as they come, but to put **op** symbols onto the 'pending' stack until we know what they are to be applied to. The occurence of a right bracket tells us that the most recent pending operation is to be activated, so we take it off the stack and put it into the answer.

4.8 Evaluating a Postfix Expression

Now we come to the nub of the problem. The compiler has input an expression in normal form from your program. It has decided that it is grammatically correct and has converted it into postfix form. Now it has to *understand* it — i.e. work out what it is asking for. The compiler does not actually perform the operations called for by the expression (that is done when you run the program) but it generates a stream of machine instructions that will have the effect of evaluating the expression. To give you a taste of what happens let me invent a totally fictitious compiler and equally fictitious machine language.

The expression $(a + b) * (c + d)$ would be converted into something like the following

```
fetch value of a into register
```

```
add value of b to register
put result into temporary store T
fetch value of c into register
add value of d to register
multiply value of T into register
```

The point is that the actual machine operations are usually rather elementary things (on small computers there would probably be far more instructions used than in this example simply because the instructions are that much more elementary).

Let's get on with the problem. The beauty of postfix expressions is that they are very easy to evaluate. That's why I converted into postfix in the first place. And here is the evaluation algorithm, which once more uses the two 'predicates' isvar(x) and isop(x):

```
01 algorithm evaluate(s,n)
02 // s is a postfix string of length n
03   for i = 1 to n begin
04     if isvar( s(i) )  then push(value of s(i) )
05     if isop( s(i) ) then begin
06       x = pop
07       y = pop
08       do y s(i) x and push result (note the order)
09     end
10   end
11   x = pop
12   return x
13 end.
```

The basic action is this: as each variable appears in the expression its value is pushed onto the stack (4). When an operation appears (5) the top two values are taken off the stack (6,7) and this operation is performed on them. The result is pushed back onto the stack (8). This means that, at any stage, the next operator applies to the previous two values on the stack. At the end there should be just one value left in the stack — the result. Pop this (11) and return it as the answer.

A compiler, as I said, does not actually perform the calculation — you are not running the program yet. At line 8 the compiler will write the machine code for performing the operation, rather than actually performing it.

Questions 4 (Hints and solutions start on page 97.)

Q 4.1. Write an algorithm in pseudocode which, given any string from A^* will decide if it is of the form $\{\mathrm{a}^n\mathrm{b}^m : n+m>0\}$, and if so, return the values of m and n.

Q 4.2. Consider the following Formal Grammar G, which generates the Formal Language L.

$A=\{a,b\}$ $\quad \mathcal{A}=\{\sigma\}$ $\quad$ initial $=\sigma$
productions:
1. $\sigma \to \sigma\sigma$
2. $\sigma \to a\sigma b$
3. $\sigma \to ab$

Show that every string in L has even length, starts with an a and ends with a b. Find all strings in L of lengths 2,4,6,8.

Prove that $\alpha \in A^+$ is in L iff it satisfies the property:

1. up to any point in α the number of b's does not exceed the number of a's; and
2. α has equal numbers of a's and b's.

What very familiar structure is described by this grammar?

Q 4.3. Find a grammar that generates the language $L=\{a^{n^2} \mid n \geq 1\}$. There are abstract considerations that show your grammar cannot be very simple.

Q 4.4. The following indicate the acceptable forms for REAL numbers in FORTRAN (excluding the exponent form $12.2E6$).

$$23.45, \quad 23, \quad .34, \quad 32., \quad -2.7, \quad -23, \quad -.7 \quad -8.$$

Write a grammar that will generate precisely such strings from the alphabet $A=\{0,1,2,\dots,9\}\cup\{+,-,.\}$. Extend the grammar to cover the exponent case as well.

Your grammar will presumably allow things like

$$1234565789122423423434534545655675667867 8.23525253152615263126$$

which FORTRAN may not like very much. Can you think of a way of working into the grammar the requirement that, for example, there are no more than 8 digits before the decimal point? There is no tidy answer.

Q 4.5. What language is generated by the following grammar?

$A=\{a,b,c\}$, $\quad \mathcal{A}=\{A,B,C\}$, $\quad$ initial $=A$
1. $A \to aABC$
2. $A \to aBC$
3. $CB \to BC$
4. $aB \to ab$
5. $bB \to bb$
6. $bC \to bc$
7. $cC \to cc$

Q 4.6. Extend the AE grammar so as to include the possibility of the 'unary minus' as in $-a+b$ or $-(b+c)*d$. You are not to allow things like $a+-b$, but $a*-b$ is marginal and probably to be included.

Q 4.7. Build up a 'succession' table for AE — saying which objects are allowed to follow which in a string generated by AE. The objects are: var, op, (,), start and end (where start and end are the start and end of the string). When you have it, check its validity by giving examples of all the successions that you claim can exist and proving that the rest cannot.

Q 4.8. Convert the following expressions first into fbie form and then into postfix form.

$$a+b-c, \quad a+b*d-c*a, \quad (a-b)*c/d, \quad a/b/c/d/(a+(b*c-d)/u)$$

Q 4.9. Consider the formal language $\mathfrak{L}$ given by the following grammar.

- The alphabet is $A = \{\mathrm{a}, \mathrm{b}, \mathrm{c}\}$.
- The abstract alphabet is $\mathcal{A} = \{\sigma, \alpha, \beta, \gamma\}$.
- The initial symbol is σ.
- The productions are
  1. $\sigma \to \mathrm{a}\sigma\beta\gamma$;
  2. $\sigma \to \mathrm{a}\beta\gamma$;
  3. $\gamma\beta \to \beta\gamma$;
  4. $\mathrm{a}\beta \to \mathrm{ab}$;
  5. $\mathrm{b}\beta \to \mathrm{bb}$;
  6. $\mathrm{b}\gamma \to \mathrm{bc}$; and
  7. $\mathrm{c}\gamma \to \mathrm{cc}$.

a) Prove that the string abc is in $\mathfrak{L}$.

b) Prove that the string a^2bcbc is *not* in $\mathfrak{L}$. If productions (4) to (7) are replaced by

$$(4'.) \quad \beta \to \mathrm{b}; \quad \text{and} \quad (5'.) \quad \gamma \to \mathrm{c}.$$

does this remain true?

c) Prove that the string $a^nb^nc^n$ is in $\mathfrak{L}$ for any $n \geq 1$.

d) Give a simple description of the strings in $\mathfrak{L}$ and give arguments to support your claim. [A formal proof is not expected]

Q 4.10. Consider the formal language $\mathfrak{L}$ given by the following grammar.

- The alphabet is $A = \{a\}$.
- The abstract alphabet is $\mathcal{A} = \{\sigma, \alpha, \beta, \gamma\}$.
- The initial symbol is σ.
- The productions are
  1. $\sigma \to \alpha\beta$;
  2. $\alpha\beta \to \alpha\alpha\beta$;
  3. $\alpha\beta \to \alpha a\gamma$;
  4. $\alpha a \to aa\alpha$;
  5. $\alpha\alpha\gamma \to \alpha\gamma$; and
  6. $a\alpha\gamma \to a$.

a) Prove that the string a^2 is in $\mathfrak{L}$.

b) Prove that the string a^4 is in $\mathfrak{L}$.

c) Let s be a string generated from σ. For each a in s, define the *wieght* of a to be 2^k, where k is the number of copies of α which occur in s to the left of a. Show that the sum of the weights of all copies of a in s is the same before and after an application of production (4).

d) Give a simple description of the strings in $\mathfrak{L}$ and give arguments to support your claim.

Q 4.11. Consider the formal language $\mathfrak{L}$ given by the following grammar.

- The alphabet is $A = \{a, b, c\}$.
- The abstract alphabet is $\mathcal{A} = \{\alpha, \beta, \gamma\}$.
- The initial symbol is α.
- The productions are
  1. $\alpha \to a\alpha\beta\gamma$;
  2. $\alpha \to a\beta\gamma$;
  3. $\gamma\beta \to \beta\gamma$;
  4. $a\beta \to ab$;
  5. $b\beta \to bb$;
  6. $b\gamma \to bc$;
  7. $c\gamma \to cc$.

a) Prove that the string $a^2b^2c^2$ is in $\mathfrak{L}$.

b) Prove that at every stage of the process, no member of the abstract alphabet appears to the left of any member of A.

c) Prove that production (2) is always applied exactly once, and that after it has been applied the resulting string is of the form $a^{n+1}\sigma$, where $n \geq 0$ and σ contains exactly $n + 1$ copies of both β and γ, ordered in such a way that up to any point in σ, there are at least as many copies of β as of γ.

d) Give a simple description of the strings in $\mathfrak{L}$ and give arguments to support your claim. [A formal proof is not expected.]

Chapter 5

Random Numbers

5.1 What is a Random Number?

Many computer programs, particularly those involved in 'simulation', require random choices of various kinds to be made. How does a computer, which is a totally deterministic device, make 'random' choices? What is a random choice for that matter? (see (Knuth 1981))

We will see later that most of the situations that require random choices to be made can be handled once we have 'random numbers' available. So, what's a random number? You will realise that this is a silly question as soon as you rephrase it in the form 'is 2 a random number?'. Randomness is not a property of individual numbers. More properly it is a property of infinite sequences of numbers.

I am now going to try to clarify what we mean by a random sequence of numbers. To keep things definite I will assume that we are talking about real numbers in the range $[0, 1)$.

What we are eventually aiming to define and construct is what is known as a random number generator. This is a function (black box) which, when given an input value **seed** (usually an integer), produces as output an infinite random sequence of real numbers $\{x_k\}$ with $x_k \in [0, 1)$.

$$\texttt{seed} \quad \rightarrow \boxed{\text{RAND}} \rightarrow \quad \{x_k\}$$

Furthermore, we require that different seeds produce different sequences and that, in some sense, 'related' seeds do not produce related sequences.

As far as I am aware, nobody has ever given an entirely convincing definition of the term 'random sequence'. On the other hand, everybody has a common-sense idea of what it means. We say things like: 'the sequence should have no pattern or structure'. More directly we might say that knowing $x_1, \ldots, x_n$ tells us nothing about $x_{n+1}, \ldots$.

One of the simplest attempts at a mathematical definition goes as follows.

Definition 5.1. A sequence $\{p_k\}$ in $[0, 1)^n$ is *uniformly distributed* on $[0, 1)^n$ if, for any box

$$A = [r_1, l_1] \times [r_2, l_2] \times \cdots \times [r_n, l_n]$$

in $[0, 1)^n$,

$$\lim_{k\to\infty} \frac{|\{p_1, p_2, \ldots, p_k\} \cap A|}{k} = |A|$$

where $|A|$ is the volume of A.

Definition 5.2. A sequence $\{x_k\}$ on $[0, 1)$ is n-distributed on $[0, 1)$ if the sequence $\{p_k\}$ given by

$$p_k = (x_{kn}, x_{kn+1}, \ldots, x_{kn+n-1}) \in [0, 1)^n$$

is uniformly distributed on $[0, 1)^n$.

We will require that a random sequence in $[0, 1)$ be ∞-distributed on $[0, 1)$, i.e. k-distributed for all k.

This turns out to be an inadequate definition of randomness, but it is good enough for all practical purposes. In particular, an ∞-distributed sequence will pass all the standard statistical tests for randomness. (If you want to be thoroughly perverse you could argue that the fact that it passes all such test is itself evidence of a certain non-randomness!)

5.2 Generating Random Numbers

Most computer languages have software 'random number generators'. These do not generate random sequences in the above sense (almost by definition, no algorithmic process can possibly do so). Instead, they produce what are called *pseudo-random numbers*. These are sequences of numbers that are sufficiently 'random' to pass all current statistical tests for randomness in 'realistic' cases.

Two quotations are in order here:

> A random sequence is a vague notion embodying the idea of a sequence in which each term is unpredictable to the uninitiated and whose digits pass a certain number of tests, traditional with statisticians and depending somewhat on the use to which the sequence is put. (Lehmer 1951).
>
> Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin. (Von Neumann 1951). "If you think you're a really good programmer,...read [Knuth's] Art of Computer Programming....You should definitely send me a resume if you can read the whole thing." – Bill Gates

There are many algorithms available (see (Knuth 1981) for some horror stories) but most languages now use what are called linear congruential generators.

A **Linear Congruential Generator** is determined by three positive integers a, b, m. We call a the *multiplier*, b the *increment* and m the *modulus*. It produces its pseudo-random sequence as follows:

- generate a sequence of integers $\{n_k\}$ $(0 \leq n_k < m)$ as follows:
  - specify n_0 - the *seed*; and
  - generate the rest by
$$n_{k+1} = an_k + b \pmod{m}.$$
- then let $x_k = \frac{n_k}{m} \in [0, 1)$.

This looks ridiculous. The x's are rational and there are only m values available for them: k/m for $k = 0, \ldots, m - 1$. However, in practical cases, m is usually something like
$$2^{32} = 4,294,967,296$$
and a is chosen carefully to give good results. (The choice of a is critical, the choice of b less important.)

It can be shown that the following conditions should be satisfied if we are to have a good generator:

- The modulus m should be large. This is usually not a problem because values like the one given above are very convenient for computer arithmetic.
- If, as often happens, m is a power of 2 then pick a such that $a \text{mod} 8 = 5$. This will have the effect of maximising the period of the necessarily cyclic sequence produced. The choice of value for a is the most delicate part of the whole design. Any suggested value for a should be submitted to a thing called the 'spectral test' before being used.

- The increment b should be coprime to m. A value like $b = 1$ is perfectly acceptable.

An acceptable choice of values would be:

$$m = 2^{32} \qquad a = 1664525 \qquad b = 1.$$

The theory behind these conditions is discussed in much more detail in (Knuth 1981, Section 3.2.1).

5.3 Applications of Random Numbers

There are many 'random processes' that are useful in various kinds of simulation. Most, if not all, of these can be generated once you have available a random number generator of the kind discussed above.

Let me now consider a few of them. Suppose throughout that rand() is a random number generator that, by successive calls, produces a random sequence uniformly distributed on $[0, 1)$.

5.3.1 Generating Random Reals

It is common to want a stream of random numbers uniformly distributed between, say, X and Y — rather than between 0 and 1. This is easily arranged

$$\text{xrand}(X, Y) = X + (Y - X) * \text{rand}()$$

i.e. just scale and shift. This does not affect the randomness in any way. Let me give a partial argument to justify that claim. We know that rand() generates a sequence $\{x_k\} \subset [0, 1)$ which is uniformly distributed. We want to show that the sequence produced by xrand(X, Y) has the same kind of property with respect to the interval $[X, Y)$. It is easiest to say this in probability language. Let $[u, v)$ be an interval in $[X, Y)$. We want the probability that a number produced by xrand(X, Y) falls in this range to be $(v - u)/(Y - X)$ — i.e. the subset gets precisely its fair share of elements. Now

$$\begin{aligned} u \leq \text{xrand}(X, Y) < v \qquad & \iff \qquad u \leq X + (Y - X) * \text{rand}() < v \\ & \iff \qquad \frac{u - X}{Y - X} \leq \text{rand}() < \frac{v - X}{Y - X} \end{aligned}$$

Now, you can easily check that this range for rand() lies in $[0, 1)$. The length of the range is

$$\frac{v - X}{Y - X} - \frac{u - X}{Y - X} = \frac{v - u}{Y - X}$$

So the probability that rand() falls in this range is, by the uniform distribution of rand(), $(v - u)/(Y - X)$. So this is the probablity that xrand(X, Y) gives a value in $[u, v)$. This is the value that we wanted.

Similar arguments would show that the sequence produced by xrand(X, Y) is k-distributed on $[X, Y)$ for all k.

5.3.2 Generating Random Integers

Suppose we want to generate a random stream of integers in the range $n, \ldots, m$. The function for this is:

$$\text{irand}(n, m) = [n + (m - n + 1) * \text{rand}()].$$

This works because:

- the result is certainly an integer in the range $n, \ldots, m$ — note that rand() < 1.
- To check randomness we just have to be sure that each integer is equally likely to be chosen at each stage. But k is chosen if

  $$k \leq n + (m - n + 1) * \text{rand}() < k + 1,$$

  i.e.

  $$\frac{k - n}{m - n + 1} \leq \text{rand}() < \frac{k + 1 - n}{m - n + 1}.$$

  The length of this range, which lies in $[0, 1)$, is independent of k (subtract and see). So, by the definition of uniform distribution for rand(), all values of k are equally likely to be chosen.

If you feel energetic you can also prove that the output of irand(n, m) has 'higher order randomness' in the sense that, for example, all pairs (i, j) with $n \leq i, j \leq m$ are equally likely to be chosen.

5.3.3 Choosing 'with probability p'

Very often random algorithms require you to 'choose this object with probability p' or 'take this option with probability p'. What does this mean and how do we do it? The basic meaning is that in a large number of such decisions you will perform the action proportion p of the time. But we're not doing it lots of times, we're just doing it once. Well, in that case you have to make a random decision as to whether or not to perform the action and the decision must be biased so as to give the correct proportion 'in the long run'.

The simplest case is that of choosing with probability 1/2. This means that we are equally likely to choose or not choose. The traditional method is to toss a coin. If we had to make a choice with probability 1/6 we might decide to throw a dice and take the action if it shows a six.

The general case can be handled by means of a standard random number generator. Suppose we want to take an option 'with probability p'. Then we use our random number generator to generate a random real in $[0, 1)$. If this number is less than p we take the option, otherwise we do not. This is correct, in terms of probability, because $[0, p)$ is proportion p of $[0, 1)$.

```
x =  rand()
if(x < p) take option
else don't take option.
```

So, for example, tossing a coin can be simulated by

```
x = rand()
if(x < 0.5) then say Heads
else say Tails
```

5.3.4 Random Shuffles

This is more difficult. We have a list $(x_1, \ldots, x_n)$ and we want to shuffle it randomly.

The following, surprisingly simple, algorithm does the trick:

```
algorithm shuffle($x$,$n$)   // shuffle $x_1, \ldots, x_n$
begin
  for i = n downto 2 begin
    k = irand(1,i)
    swap over $x_i$ and $x_k$
  end
end
```

Why does this work? It seems that it is not doing enough ‘mixing up’ to produce a genuinely random shuffle. We can only prove that it works if we have some notion of what we mean by a ‘random shuffle’. Let me keep things simple and say that a shuffling process is random if each element of the list can end up in any position in the list with equal probability. That is to say, in a list of n items there are n possible places that any given element can eventually occupy. I want each of these places to be equally likely.

So we now have to calculate the probability that our algorithm puts element x_i into position j, where $1 \leq i, j \leq n$. If you look carefully at the algorithm you will see that element x_i end up in position j if it is **not** chosen on the first $n - j$ steps of the algorithm and **is** chosen on the step $n - j + 1$. Since we know that irand(n, m) is genuine we can use elementary methods to find that the probability of x_i ending up in position j is

$$\frac{n-1}{n} \cdot \frac{n-2}{n-1} \cdot \frac{n-3}{n-2} \cdots \frac{j}{j+1} \cdot \frac{1}{j} = \frac{j}{n} \cdot \frac{1}{j} = \frac{1}{n},$$

which is exactly what we wanted.

5.3.5 Random Samples

The problem here is to choose a sample of k things at random from a list of n things. There are a number of possible problems here and I will just consider two of them:

Straightforward sampling. Given the list $(x_1, \ldots, x_n)$ use the shuffle algorithm given above, but stop it after k steps. The last k items in the list will then be a random sample (or the shuffling algorithm would not be random).

Conveyor Belt sampling It is often the case that the set to be sampled from is far too big to be held in memory. We have to be able to make our selection simply by looking at one item of the set at a time. The obvious analogy is that of a quality control inspector standing alongside a conveyor belt and having to choose, let’s say, 10 items from every 1000 that pass him. He has to be able to do this without backing up the conveyor belt.

```
algorithm runningsample(k, N)
// choose k items at random from N supplied
begin
  t = 0
  chosen = 0
  repeat begin
    if ((N-t)*rand() >= (k-chosen)) begin
      // pass over next item
      read in next item
      t = t + 1
    end
    else begin
      //  accept next item
      read in next item
      print out next item
      t = t + 1
      chosen = chosen + 1
    end
  end // repeat
  until (chosen = k)
end.
```

In this it is assumed that the data to be sampled from can be read item by item ('stream input') - 'read in next item' fetches the next item in the list.

The logic of this method is that if, at the ith item, we have so far chosen j items then we have $k-j$ items left to choose from the remaining $n-i+1$ items (including the ith). So it seems reasonable to choose the ith item with probability

$$\frac{k-j}{n-i+1}$$

and that's what the algorithm does. That seems too obvious to be true!

This is a simple routine but once more the real problem lies in trying to justify its claim to produce a random sample. Indeed, it is not obvious that the routine will succeed in getting its full sample before it runs out of input!

We cannot end up with more than k items because we stop the loop if we have got k. It is thus enogh to show that we cannot end up with *fewer* than k. Suppose conversely that we had ended up with $j<k$ items and that the last item *not* chosen was the ith. Then, at that stage, we had already chosen $j-(N-i)$ items. So the probability of choosing the ith was

$$\frac{k-(j-(N-i))}{N-i+1}=\frac{N-i+(k-j)}{N-i+1}\geq 1,$$

so we *must* have chosen it! Contradiction.

Let's now look at the claim that it produces *random* samples. Consider samples of size k. There are C_k^n of these in all and we want each of them to be an equally likely selection.

Let me start by looking at the simple case $k=2$. What is the probability of our choosing the subset $\{x_a, x_b\}$ from $(x_1, x_2, \ldots, x_n)$?

We must *not* pick $x_1, x_2, \ldots, x_{a-1}$, we *must* pick x_a, we must *not* pick $x_{a+1}, \ldots, x_{b-1}$, we *must* pick x_b and we must *not* pick $x_{b+1}, \ldots, x_n$. Turn all that into a probability calculation and you get the probability

$$p=\prod_{i=1}^{i=a-1}\left(\frac{n-i-1}{n-i+1}\right)\cdot\frac{2}{n-a+1}\cdot\prod_{i=a+1}^{b-1}\left(\frac{n-i}{n-i+1}\right)\cdot\frac{1}{n-b+1}\cdot\prod_{i=b+1}^{n}\left(\frac{n-i+1}{n-i+1}\right).$$

This looks horrible at first sight, but closer examination reveals all.

The denominators run in sequence from n down to 1. So the overall denominator is $n!$. We have a 2 and a 1 on the top (at the chosen points) and that gives us a factor $2!$. Finally, the rest of the numerator is

$$(n-2)(n-3)\cdots(n-a)\cdot(n-a-1)(n-a-2)\cdots(n-b+1)\cdot(n-b)\cdots 1=(n-2)!$$

Notice that the sequence steps over the joins properly. This gives us the factor $(n-2)!$. So the probability is

$$p=\frac{2!\,(n-2)!}{n!}=\frac{1}{C_2^n}\quad\text{as required.}$$

The general argument goes in just the same way. Suppose we specify a subset of k elements. Then we work out the probability of choosing this subset, as above. We find that the denominators run from n down to 1 once more — giving the $n!$. The 'chosen' elements give us the factor $1.2.3\ldots k=k!$. There remain $n-k$ numerators. They are all different, the largest is $n-k$ and the smallest is 1. So they must be precisely all the numbers from 1 to $n-k$ and so give the factor $(n-k)!$. Thusthe probability of choosing this subset is

$$p=\frac{k!\,(n-k)!}{n!}=\frac{1}{\binom{n}{k}}.$$

So all subsets of a given size are equally likely to be chosen, and we definitely have a random sample.

Questions 5 (Hints and solutions start on page 101.)

Q 5.1. Show that if X is any finite set and $\phi\colon X \to X$ is any mapping then, for any $x \in X$, the sequence $\{x_i\}$ defined by

$$x_0 = x \qquad x_{i+1} = \phi(x_i)$$

is *eventually* periodic; in other words, from some point in the sequence the sequence is periodic from there on.

Q 5.2. In this and the following questions, assume that you have available a random number generator RAND which produces random reals uniformly distributed on $[0, 1)$. So a statement like $x = RAND$ will give you a 'random number'.

Consider the following sets

$$\begin{aligned}
T_1 &= \{(x, y) \in \mathbb{R}^2 : 0 \le x, y < 1 \qquad y < x\}\\
T_2 &= \{(x, y) \in \mathbb{R}^2 : 0 \le x, y < 1 \qquad x + y < 1\}\\
T_3 &= \{(x, y, z) \in \mathbb{R}^3 : 0 \le x, y, z < 1 \qquad x + y + z < 1\}\\
T_4 &= \{(x, y, z) \in \mathbb{R}^3 : 0 \le x, y, z < 1 \qquad x + y + z = 1\}
\end{aligned}$$

A certain application requires you to produce a random sequence of points in T_1. How do you do this? Answer the same question for T_2.

Here are four ways to produce a sequence of points in T_3. Each application of the process produces one more point in the set. How do they compare with regard to efficiency? Which of the four actually produce a random sequence in T_3?

Method 1

```
    repeat x = RAND, y = RAND, z = RAND
    until $x + y + z < 1$.
```

Method 2

```
    repeat
        pick a random point $(x, y)$ in $T_2$,
        z = RAND
    until $x + y + z < 1$
```

Method 3

```
    pick a random point $(x, y)$ in $T_2$
    repeat z = RAND until $x + y + z < 1$
```

Method 4 pick a random point in T_2 pick z at random from $[0, 1 - x - y)$

Finally, show how to obtain a random sequence of points in T_4.

Q 5.3. Some half-witted geographer wants you to produce for him a sequence of points chosen randomly on the surface of the earth, for some kind of statistical experiment. How do you set about producing points at random on the surface of a sphere? (No, you don't pick a latitude at random and then a longitude at random. Why?)

Q 5.4. Show how to generate random $n \times n$ *symmetric* matrices with real elements in the range $(-1, 1)$.

Q 5.5. Suppose we want to generate random integers in the range $0 \ldots 10^n - 1$. One, slightly cranky, approach might be to choose each of the $n - 1$ digits $(0 \ldots 9)$ randomly and then stick them together to form the number. Does this give random integers? Prove it.

Q 5.6. The simplest algorithm for obtaining a random sample of k items from a list $x_1, \ldots, x_n$ is this:

```
count = 0
repeat
  pick i at random from $1,2, ... ,n$
  if $x_i$ has not yet been chosen then choose it and put count=count+1
until count = k
```

The only problem with this is its potential running time. The bigger k is compared to n the more 'doublers' we are going to have to reject.

The problem is to estimate how many choices we are going to have to make before we get the k items that we want. The following crude argument gives the right result. Let $w(n, k)$ be the average *waiting time*, i.e. number of choices made, in order to get a sample of k things from n items. Then $w(n, k)$ is $w(n, k-1)$ plus the expected number of choices needed to obtain the final item (you must have found $k-1$ items before you find the kth!). Show that

$$w(n,k) = w(n,k-1) + \frac{n}{n-k+1}$$

and deduce that

$$w(n,k) = n\left(\frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{n-k+1}\right) = n(H_n - H_{n-k})$$

where H_n is the nth harmonic number.

Show that if n is reasonably large then we have the approximation

$$\frac{w(n,k)}{k} \approx \frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right), \text{ where } \alpha = \frac{k}{n}$$

Q 5.7. In this question, you are given a random number generator `RAND()` which produces random reals uniformly distributed in $[0, 1)$.

a) You are required to choose k objects " at random" from a sequence of N objects — perhaps passing on a conveyor belt. You must decide whether a given object is to be selected *before* the next one becomes available. Give pseudocode for such an algorithm, and justify the assertion that it will produce exactly k items. [You are *not* asked to justify that the sample is a random sample.]

b) Let T be the triangle with vertices at $(0, 2)$, $(-1, 0)$ and $(1, 0)$. Describe an algorithm to generate a sequence of points $\{p_n\}$ in T "at random". Your answer should produce an additional point p_n for each pair of calls to `RAND()`. Explain briefly why your algorithm does what is required.

Chapter 6

The Fast Fourier Transform

6.1 Multiplying Polynomials

Suppose we have two polynomials

$$p(x) = a_0 + a_1x + a_2x^2 + \cdots a_{n-1}x^{n-1} \quad \text{and} \quad q(x) = b_0 + b_1x + b_2x^2 + \cdots b_{n-1}x^{n-1}, \tag{6.1}$$

each of degree $n-1$. We consider the problem of computing their product

$$\begin{aligned} p(x)q(x) = a_0b_0 + (a_1b_0 + a_0b_1)x + (a_2b_0 + a_1b_1 + a_0b_2)x^2 + \cdots \\ + (a_{n-1}b_0 + \cdots + a_0b_{n-1})x^{n-1} + \cdots \\ + (a_{n-1}b_{n-2} + a_{n-2}b_{n-1})x^{2n-3} + a_{n-1}b_{n-1}x^{2n-2}. \end{aligned}$$

As written, this product involves $2n-1$ terms and a total of

$$2(1 + 2 + \cdots + n - 1) + n = n^2$$

multiplications to compute all the coefficients. A natural question then arises as to whether we can manage with fewer multiplications. The aim of this section is to show that the work can be suitably re-arranged so that only $O(n \log n)$ multiplications are needed. The constant in the $O(n \log n)$ is really quite small; unlike Strassen's method of matrix multiplication, the underlying algorithm is of great practical importance.

The idea is simple, although we don't discuss it directly in this form. In order to determine $p(x)q(x)$, a polynomial of degree $2n-2$, it is enough to know its value at $2n-1$ points. Each of $p(x)$ and $q(x)$ can be determined at a fixed set of $2n-1$ points, and so the product polynomial can be determined at these points with $2n-1$ multiplications in $\mathbb{R}$ or $\mathbb{C}$. Thus provided we can efficiently evaluate each polynomial at the $2n-1$ points, and then reconstruct the product polynomial from its $2n-1$ known values, we have another way of multiplying the two polynomials.

6.2 Fourier Series

The crucial idea uses a variant of the Fourier transform, which is often first met as a way of representing a function in terms of sine and cosine of multiples of an angle. Recall that given f a function defined on the interval $[-\pi, \pi]$, or defined on $\mathbb{R}$ and periodic, with period 2π, we define the Fourier coefficients of f by

$$a_n = \frac{1}{2\pi}\int_{-\pi}^{\pi} f(t)\cos nt\,dt, \qquad \text{and} \qquad b_n = \frac{1}{2\pi}\int_{-\pi}^{\pi} f(t)\sin nt\,dt.$$

For many functions, we can reconstruct f from its Fourier coefficients, by using the Fourier series. In the case of a string fixed between two points, and initially held in a shape which is the graph of f, the Fourier (sine) coefficients have a physical interpretation; they represent the amount of the corresponding harmonic in the resulting vibration.

A trick often used to unite the Fourier sine and cosine series, even for real functions, is to define the complex Fourier coefficient

$$c_n = \frac{1}{2\pi}\int_{-\pi}^{\pi} f(t)\exp(-int)\,dt.$$

The usual series then follow by taking real and imaginary parts. There are many advantages in using this complex form, even though there is a less intuitive description for the coefficients.

In this section we apply these ideas to the **discrete Fourier transform**, the version which applies to a function f defined on the finite set $\mathbb{Z}_n$. As in the continuous case, it is helpful to think of the function as being periodic, so that $f(j+n) = f(j)$. The discrete transform has the advantage that we no longer have convergence worries; of course a finite sum exists, while an integral may not: but it does mean that the formulae are less familiar.

6.3 The Fourier Matrix

We need a simple result about complex numbers.

Lemma 6.1. Let $z^n = 1$, and assume that $z \neq 1$. Then

$$1 + z + z^2 + \cdots + z^{n-1} = 0.$$

Proof. It is trivial to verify the factorisation

$$z^n - 1 = (z-1)(1 + z + z^2 + \cdots + z^{n-1}).$$

The result follows, since we are given a zero of the left hand side. □

It is easy to describe the n^{th} roots of unity. Let $\omega = \exp(-2\pi i/n)$; then the roots are all of the form ω^k for some k with $0 \leq k < n$. Note also that ω satisfies the condition of the lemma. We now define the Fourier matrix $\mathcal{F}_n$ for any integer $n \geq 1$ by:

$$\mathcal{F}_n = \frac{1}{\sqrt{n}}\begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \dots & \omega^{(n-1)(n-1)} \end{pmatrix}.$$

Note that $\mathcal{F}_n$ is a complex symmetric matrix; so is the same as its transpose. Thus its adjoint (the transposed complex conjugate of the given matrix) is just the complex conjugate. We write this as $\bar{\mathcal{F}}_n$. It is given simply by replacing ω in the above by $\bar{\omega} = \exp(2\pi i/n)$.

A word about notation: we shall have quite a lot to do with ω. Note the choice of sign in the exponent; it was arbitrary, but this way the definition of $\mathcal{F}_n$ is correct. When we need to emphasise that we have an n^{th} root of unity, we shall write ω_n.

In the study of complex vector spaces, an interesting class of matrices are the *unitary* ones; those whose inverse is the adjoint. They are the complex equivalent of the *orthogonal* matrices, in which the inverse is just the transpose. An orthogonal matrix represents a rotation of given orthonormal axes (possibly with a reflection); a unitary matrix could be considered for this reason, as a complex rotation.

Theorem 6.2. The Fourier matrix $\mathcal{F}_n$ is a unitary matrix.

Proof. It is enough to show that $\bar{\mathfrak{F}}_n\mathfrak{F}_n = I_n$, where I_n is the identity matrix. Since the j^{th} row of $\bar{\mathfrak{F}}_n$ is

$$\frac{1}{\sqrt{n}}(1, \bar{\omega}^j, \bar{\omega}^{2j}, \ldots, \bar{\omega}^{(n-1)j})$$

while the k^{th} column of $\mathfrak{F}_n$ is

$$\frac{1}{\sqrt{n}}(1, \omega^k, \omega^{2k}, \ldots, \omega^{(n-1)k}),$$

and since $\bar{\omega}^k = w^{-k}$, the entry in the $(j,k)^{th}$ place of the product is just

$$\frac{1}{n}(1 + \omega^{(j-k)} + \omega^{2(j-k)} + \cdots + \omega^{(n-1)(j-k)}).$$

We now distinguish two cases. If $j = k$, this is just $(1 + 1 + \cdots + 1)/n = 1$, while if $j \neq k$, it is zero by the lemma, since $\omega^{(j-k)}$ is an n^{th} root of unity, but is not equal to 1. It follows that the product is the identity matrix as claimed. □

We can identify a complex valued function defined on $\mathbb{Z}^n$ with an element of $\mathbb{C}^n$ by $f \to \mathbf{f} = (f(0), f(1), \ldots, f(n-1))$.[1] The discrete Fourier transform of $\mathbf{f}$ is then defined to be the action of the Fourier matrix on $(f(0), f(1), \ldots, f(n-1))$. It will be convenient to write $\mathbf{f} \in \mathbb{C}^n$, or $\mathbf{f} = (f_1, f_2, \ldots, f_{n-1})$ although we think of $\mathbb{C}^n$ in this form as periodic functions on $\mathbb{Z}^n$.

Identifying the Fourier matrix as unitary gives one way of thinking of the discrete Fourier transform; as a complex rotation in $\mathbb{C}^n$. The idea that a rotation of the co-ordinate axes will preserve essential features of a problem, while possibly making the new co-ordinate representation easier to work with, is a familiar one. We thus begin a study which will show there are indeed some things which are easier to describe and manipulate in the transformed representation.

Corollary 6.3. Let $\mathbf{f} = (f_0, f_1, \ldots, f_{n-1})$ be a vector in $\mathbb{R}^n$ or $\mathbb{C}^n$, and define $\mathbf{F}$ in $\mathbb{C}^n$ by

$$F_p = \frac{1}{\sqrt{n}}\sum_{j=0}^{n-1} f_j \exp\left(\frac{-2\pi i j p}{n}\right) = \frac{1}{\sqrt{n}}\sum_{j=0}^{n-1} f_j \omega^{jp} \qquad (0 \leq p \leq n-1), \tag{6.2}$$

Then the transformation can be inverted to give

$$f_j = \frac{1}{\sqrt{n}}\sum_{p=0}^{n-1} F_p \exp\left(\frac{2\pi i j p}{n}\right) = \frac{1}{\sqrt{n}}\sum_{p=0}^{n-1} F_p \omega^{-jp} \qquad (0 \leq j \leq n-1). \tag{6.3}$$

Proof. Our definition gives $\mathbf{F} = \mathfrak{F}_n(\mathbf{f})$. Since the Fourier matrix is unitary, we have $\mathbf{f} = \bar{\mathfrak{F}}_n(\mathbf{F})$, which is the required result. □

6.4 Products

We have seen that the discrete Fourier transform is defined on $\mathbb{C}^n$. In addition to being a vector space, $\mathbb{C}^n$ becomes a commutative ring when a "pointwise" product is defined in the obvious way for $\mathbf{a}$ and $\mathbf{b}$ in $\mathbb{C}^n$ by

$$\mathbf{a}.\mathbf{b} = (a_1, a_2, \ldots, a_n).(b_1, b_2, \ldots, b_n) = (a_1b_1, a_2b_2, \ldots, a_nb_n).$$

There is however another product, the **convolution** product on $\mathbb{C}^n$ which is natural from a different viewpoint. For $\mathbf{a}$ and $\mathbf{b}$ in $\mathbb{C}^n$, define $\mathbf{a} \star \mathbf{b} \in \mathbb{C}^n$ by

$$(\mathbf{a} \star \mathbf{b})_j = \sum_{k=0}^{n-1} a_{j-k} b_k,$$

[1]We confuse the function f and the vector $\mathbf{f}$; it may have been clearer to drop the bold font vector notation.

where each index in the sum is interpreted mod n, so that for example $a_{-1} = a_{n-1}$ etc. This product arises from considering C^n as functions on $\mathbb{Z}_n$, the cyclic abelian group with addition mod n, as we did in the definition of the discrete Fourier transform. Let 1_k be the function on $\mathbb{Z}^n$ which takes the value 0 except at the integer k, where it takes the value 1. A natural way to define $1_j.1_k$ is as 1_{j+k}, where we interpret $j+k$ modulo n. This extends by linearity to the whole of $\mathbb{C}^n$; it is easy to check that the resulting product is the convolution just defined.[2] Convolution makes $\mathbb{C}^n$ into a (commutative) ring; it is easy to verify that convolution is an associative product. It is commutative since

$$(\mathbf{b} \star \mathbf{a})_j = \sum_{k=0}^{n-1} b_{j-k} a_k = \sum_{k=0}^{n-1} b_l a_{j-l},$$

where we have rewritten in terms of $l = j - k$,

$$= \sum_{l=j}^{j-n-1} a_{j-l} b_l = \sum_{l=0}^{n-1} a_{j-l} b_l = (\mathbf{a} \star \mathbf{b})_j,$$

since we can sum over $\mathbb{Z}_n$ in any order.

6.5 Properties of the Fourier Transform

In this section we note a number of useful properties of the Fourier transform. Recall that we write $\mathcal{F}_n \mathbf{f} = \mathbf{F}$; we shall sometimes write $\hat{\mathbf{f}}$ instead of F. It is often convenient to ignore the normalising constant $1/\sqrt{n}$ in the formula, in which case, we have

$$F_p = \sum_{j=0}^{n-1} f_j \omega^{jp}.$$

The constant is then explicitly inserted in the formula at some later stage.

Proposition 6.4. Let $\mathbf{f} \in \mathbb{R}^n$. Then $F_0 \in \mathbb{R}$, while for $p > 0$, $F_{n-p} = \bar{F}_p$. In particular, if n is even, then $F_{n/2} \in \mathbb{R}$.

Proof. Since by definition, ω is an n^{th} root of unity, we have $\omega^{n-j} = \omega^n \omega^{-j} = \bar{\omega}^j$; now compute. $\square$

Note also that, in general we have $F_0 = f_0 + f_1 + f_2 + \cdots + f_{n-1}$, while if n is even, so $\omega^{kn/2} = (-1)^k$, we have $F_{n/2} = f_0 - f_1 + f_2 + \cdots - f_{n-1}$.

Theorem 6.5 (Parseval's Theorem). Let $\mathbf{f} \in \mathbb{C}^n$ and let $\mathbf{F} \in \mathbb{C}^n$ be the discrete Fourier transform of $\mathbf{f}$. Then

$$\sum_{j=0}^{n-1} |f_j|^2 = \sum_{p=0}^{n-1} |F_p|^2.$$

Proof. Recall that for any complex number z, we have $|z|^2 = z\bar{z}$.

From the definitions, we have

$$\begin{aligned}\sum_{p=0}^{n-1} |F_p|^2 &= \frac{1}{n} \sum_p \Big| \sum_j f_j \omega^{pj} \Big|^2 = \frac{1}{n} \sum_p \sum_j f_j \omega^{pj} \sum_k \bar{f}_k \omega^{-pk} \\ &= \frac{1}{n} \sum_p \sum_j \sum_k f_j \bar{f}_k \omega^{p(j-k)} = \frac{1}{n} \sum_p \sum_j |f_j|^2\end{aligned}$$

[2]This is a very simple example of a *group algebra*, a construct central to modern harmonic analysis.

since the sum vanishes by Lemma 6.1 unless $j = k$. The result follows. □

Remark: For physical reasons, this sum is often referred to as the power spectrum; we see it is preserved under the Fourier transform. The result is usually known in the engineering literature as Rayleigh's Theorem. It is essentially a restatement of the fact that the Fourier matrix is unitary, and follows trivially from that fact by using a little more matrix theory as follows:

$$\begin{aligned} ||\mathcal{F}(\mathbf{f})||^2 &= < \mathcal{F}(\mathbf{f}), \mathcal{F}(\mathbf{f}) > = < \mathcal{F}^*\mathcal{F}(\mathbf{f}), \mathbf{f} > \\ &= < \mathbf{f}, \mathbf{f} > = ||\mathbf{f}||^2, \end{aligned}$$

where $\mathcal{F}^*$ is the adjoint, or transposed complex conjugate of $\mathcal{F}$, and we have used the fact that $\mathcal{F}$ is unitary.

These results are all of interest. However for our purposes, the crucial one is that the Fourier transform behaves nicely with respect to convolution. We already know that $\mathcal{F}_n$ is an isometry from $\mathbb{C}^n$ to $\mathbb{C}^n$; we now observe that is is an algebra isomorphism when appropriate products from Section 6.4 are chosen.

Theorem 6.6 (The Convolution Theorem). Let f and $\mathbf{g}$ be in $\mathbb{C}^n$. Then

$$\hat{\mathbf{f}}\hat{\mathbf{g}} = \mathcal{F}_n(\mathbf{f} \star \mathbf{g}).$$

In other words, the discrete Fourier transform maps convolution to pointwise multiplication.

Proof. This is a simple calculation, but relies heavily on the fact that each n-tuple is considered as periodic on $\mathbb{Z}_n$, so $f_j = f_{n+j}$ and so on.

$$\mathcal{F}_n(\mathbf{f} \star \mathbf{g})_p = \sum_k \sum_j f_j g_{k-j} \omega^{kp} = \sum_j f_j \omega^{jp} \sum_k g_{k-j} \omega^{(k-j)p}.$$

Since the sum is really over $\mathbb{Z}_n$, we can start anywhere, so

$$\mathcal{F}_n(\mathbf{f} \star \mathbf{g})_p = \sum_j f_j \omega^{jp} \sum_{k=j}^{n+j} g_{k-j} \omega^{(k-j)p},$$

and substituting $l = k - j$, and then rewriting with k instead of l,

$$= \sum_j f_j \omega^{jp} \sum_k g_k \omega^{kp} = \hat{f}_p \hat{g}_p.$$

□

6.6 The Fast Fourier Transform.

The convolution theorem suggest a different way of computing $\mathbf{a} \star \mathbf{b}$ for $\mathbf{a}, \mathbf{b} \in \mathbb{C}^n$; as the inverse transform of $\hat{\mathbf{a}}.\hat{\mathbf{b}}$. In other words, we first compute the discrete Fourier transforms of $\mathbf{a}$ and $\mathbf{b}$, then compute the pointwise product $\hat{\mathbf{a}}.\hat{\mathbf{b}}$, and finally compute the inverse discrete Fourier transform to obtain $\mathbf{a} \star \mathbf{b}$. However, the obvious way to calculate the discrete Fourier transform, using equation 6.2 involves n multiplications for each of the n terms. Thus two transforms, a pointwise product and and an inverse transform total $3n^2 + n$ multiplications, as opposed to the n^2 multiplications needed to do the convolution directly. We discuss in this section an algorithm which makes the computation of the Fourier transform *very* much more efficient, reducing the calculations from one of order n^2 to one of order $n \log n$. This arrangement of the calculation is known as the **fast Fourier transform** or **FFT**. The algorithm was brought to the attention of many people by Cooley and Tukey (1965), although as often happens it appears to have been discovered a number of times before that, with the idea going back to Gauss! There are now many variants; a good reference for both these and for code itself is (Press, Flannery, Teukolsky & Vetterling 1992, Chapter 12). The common feature is that the length of the vector should factor into a product of a large number of small primes; a very efficient version is when the image size is a power of two, and we present this.

Assume then we are calculating the Fourier transform of a vector of length $n = 2^m$, so $m = \log_2(n)$. To concentrate on essentials, we ignore the normalising factor. Write $\omega_n = \exp(-2\pi i/n)$ to emphasise the underlying image size. Then $(\omega_n)^n = 1$ and, since $n = 2^m$, we have $n = 2.2^{m-1} = 2n_1$ for $m \geq 1$. We have

$$F_p = \sum_{j=0}^{n-1} f_j \omega_n^{jp} = \sum_{j=0}^{2n_1-1} f_j \omega_{2n_1}^{jp}.$$

on re-writing in terms of n_1. This sum can be split into two terms, according to whether j is even or odd, so

$$\begin{aligned} F_p &= \sum_{j=0}^{n_1-1} f_{2j}\omega_{2n_1}^{2jp} + \sum_{j=0}^{n_1-1} f_{2j+1}\omega_{2n_1}^{(2j+1)p}, \\ &= \sum_{j=0}^{n_1-1} f_{2j}\omega_{n_1}^{jp} + \sum_{j=0}^{n_1-1} f_{2j+1}\omega_{n_1}^{jp}\omega_n^p. \end{aligned}$$

We consider this sum in more detail. Suppose first that $0 \leq p < n_1$. Then we can rewrite this as

$$F_p = \hat{f}_{\text{even}}(p) + \omega_n^p \hat{f}_{\text{odd}}(p) \tag{6.4}$$

where f_{even} is the restriction of f to the even elements in its domain, and similarly for f_{odd}. Further, if we consider $\hat{f}$ at a point in the second half of its domain, namely at a point $p + n_1$ for some p with $0 \leq p < n_1$, we see that

$$\omega_{n_1}^{j(p+n_1)} = \omega_{n_1}^{jp} \quad \text{and} \quad \omega_n^{p+n_1} = -\omega_n^p.$$

Thus we have

$$\hat{f}(p+n_1) = \hat{f}_{\text{even}}(p) - \omega_n^p \hat{f}_{\text{odd}}(p). \tag{6.5}$$

These two formulae are essentially the same, and show we can calculate a transform of size $n = 2^m$ in terms of two transforms of size 2^{m-1}.

6.6.1 Timing the FFT

We analyse the savings this method produces in the usual way. Let $\alpha(m)$ be the number of additions needed to calculate the FFT on a vector of length $n = 2^m$, and let $\mu(m)$ be the corresponding number of multiplications.

Lemma 6.7. For $m \geq 1$, $\mu(m) = m.2^{(m-1)}$, $\alpha(m) = m.2^m$.

Proof. Clearly $\mu(0) = \alpha(0) = 0$, while $\mu(1) = 1$, $\alpha(1) = 2$. More generally, the result follows by induction, since

$$\mu(m+1) = 2^m + 2\mu(m) \quad \text{and} \quad \alpha(m+1) = 2^{(m+1)} + 2\mu(m).$$

To check this, note that in addition to computing f_{even} and f_{odd}, Equation 6.4 gives an additional 2^m multiplications when computing a transform of length 2^{m+1}, while Equation 6.5 shows these same products, $\omega_n^p \hat{f}_{\text{odd}}(p)$, are all that are needed. The addition formula is even more straightforward. □

Thus we get the claimed reduction form $O(n^2)$ to $O(n \log n)$. In particular, this gives a method of implementing convolution using only $n + 3n\log(n)/2$ multiplications. Our original problem in Section 6.1 had two polynomials, each of degree $n-1$, whose product was (of course) of degree $2n-2$. We saw in Exercise 6.3 that the product polynomial could be computed using convolution, providing we worked in $\mathbb{C}^{2n}$, to avoid the unwanted "wrap-around" part of the definition of convolution. We now see that, provided we work in a space of dimension 2^m where m is the smallest integer such that $2n \leq 2^m$, then we replace n^2 multiplications with $O(m.2^m) = O(n \log n)$. If n is large, the saving can be very significant.

In fact a further speed-up can be obtained by arranging the data so it can be accessed simply. In order to compute $\hat{f}$ you need the values

$$\underbrace{\underbrace{f(0)f(4)}\,\underbrace{f(2)f(6)}} \qquad \text{and} \qquad \underbrace{\underbrace{f(1)f(5)}\,\underbrace{f(3)f(7)}}$$

as two 4 - point transforms to combine into the 8 - point transform as

$$\underbrace{\underbrace{\underbrace{f(0)f(4)}\,\underbrace{f(2)f(6)}}\,\underbrace{\underbrace{f(1)f(5)}\,\underbrace{f(3)f(7)}}}\,.$$

Each of these 4 - point transforms is calculated as two 2 - point transforms, as implied by the bracketing. Thus in order to do the calculation, it is convenient if the input data is re-ordered. If we write $f(\sigma(j))$ for the required re-ordering, so σ is a permutation of the set $\{1, 2, \ldots, n\}$, then $\sigma(j) =$ the bit reversal of the binary expansion of j. For example,

$$\sigma(6) = \sigma(110_2) = 011_2 = 3.$$

6.7 Convolution as Filtering

We claimed above that the fast Fourier transform was very important, but so far have presented it purely as a way of multiplying polynomials. For the remainder of this section we give a brief indication of a much more practical use for this algorithm. To do so, we first suggest another way of thinking about convolution.

Although convolution is symmetric, I want to interpret the action of convolving by a fixed element of $\mathbb{C}^n$, and to emphasise the asymmetry I will call the fixed element $\mathbf{m}$ rather than $\mathbf{b}$. We thus consider the map

$$\mathbf{a} \to \mathbf{a} \star \mathbf{m} \qquad (\mathbf{a} \in \mathbb{C}^n)$$

for a fixed element $\mathbf{m} \in \mathbb{C}^n$. We will call such a fixed element a **mask** or **filter**. How does filtering (convolving with a fixed mask) affect $\mathbf{a} \in \mathbb{C}^n$, often called the **signal**?

To get an initial understanding, it helps to choose a simple filter, so let

$$\mathbf{m} = (m_0, m_1, 0, \ldots, 0, m_{-1}) \qquad \text{and} \qquad \mathbf{a} = (a_0, a_1, a_2, \ldots, a_{n-2}, a_{n-1}).$$

From the definition, we have

$$\mathbf{a} \star \mathbf{m} = (a_{-1}m_1 + a_0m_0 + a_1m_{-1},\ a_0m_1 + a_1m_0 + a_2m_{-1}, \ldots \\ a_{n-3}m_1 + a_{n-2}m_0 + a_{n-1}m_{-1},\ a_{n-2}m_1 + a_{n-1}m_0 + a_0m_{-1}). \quad (6.6)$$

The circular wrap-around is essential as far as $\mathbf{m}$ is concerned, but can get in the way of understanding what happens to $\mathbf{a}$.

Example 6.8. Let $\mathbf{a} = (0, 0, 3, 7, 1, 4, 6, 2, 0, 0)$ and $\mathbf{m} = (4, 2, 0, 0, 0, 0, 0, 0, 0, 1)$. Evaluate $\mathbf{a} \star \mathbf{m}$.

Solution This is simple arithmetic, based on Equation 6.6.

$$\mathbf{a} \star \mathbf{m} = (0, 3, 19, 35, 22, 24, 34, 20, 4, 0).$$

It is worth doing to the calculation to discover an efficient way to set it out. One way is as follows: first flip **m**, and then consider sliding it along above **a**, multiplying elements in the same column, and assigning the sum of these products as the output associated with the current position of element zero of **m**. Here then is the calculation of the fourth element (ie the element with index 3) of $\mathbf{a} \star \mathbf{m}$:

$$\begin{array}{cccccccccc} 0 & 0 & 2 & 4 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 7 & 1 & 4 & 6 & 2 & 0 & 0 \end{array}$$

And $(\mathbf{a} \star \mathbf{m})_3 = 2.3 + 4.7 + 1.1$.

It may be easier now to see why the language of signals and filters was used. In this case, the filter **m** has acted as a smoothing filter; the two pronounced peaks in the original signal have been spread out; indeed, looking at **m** it may now be clear that it is indeed a prescription to smooth the central element by incorporating a contribution from either side. For convenience in calculation, this particular filter also amplified the signal. However the effect may have been easier to understand if the filter had been scaled so that the total mass (the sum of all the values in the filter) was set to 1. This would involve multiplying $\mathbf{a} \star \mathbf{m}$ by 1/7.

The effect of convolution has been to "do the same thing" to each of the elements of the incoming signal **a**. In this case it was easy to do the convolution directly because the support (the number of non -zero elements) of **m** was small. As the support becomes larger, the gain from implementing such a filter using Fourier transforms and the convolution theorem becomes significant.

In passing we note another name for **m**, since we can recognise **m** as the (reveresed) output corrsponding to input 1_0 in the notation of Section 6.4. It is thus sometimes called the *point spread function*, or perhaps the *instrument response function*. From this viewpoint, displaying the mask as $(m_0, \ldots, m_{n-1})$ is not particularly convenient; it is usually easier to visualise the effect if presented as

$$(\ldots, m_{-2}, m_{-1}, m_0, m_1, m_2, \ldots)$$

centered as closely as possible on m_0. Thus we would describe the mask of Example 6.8 as $(0, 0, 0, 1, 4, 2, 0, 0, 0, 0)$, and as being centered on the value 4.

6.8 An application to Image Processing

We can extend the definition of the discrete Fourier transform from vectors to arrays with only notational difficulty. It is of interest because an array has a very visual representation as a digitised image or picture, with $f(j, k)$ representing the brightness of the image at position (j, k). We thus define the Fourier transform of an image as the (complex valued) array F, given by

$$F(p, q) = \frac{1}{\sqrt{nm}} \sum_{j=0}^{m-1} \sum_{k=0}^{n-1} f(j, k) \exp\left(\frac{-2\pi i j p}{m}\right) \exp\left(\frac{-2\pi i k q}{n}\right).$$

Just as before we can write this as a double sum

$$F(p, q) = \frac{1}{\sqrt{nm}} \sum_{j=0}^{m-1} \sum_{k=0}^{n-1} f(j, k) \omega_m^{jp} \omega_n^{kq}.$$

There is a similar inversion formula to Equation 6.3, which can be checked in exactly the same way.

The implementation of the true 2 - dimensional transform can be done in two stages; for each fixed row, we take the transform of each column, and then with the transformed data in the corresponding position, take the transform of each row. Thus all the difficulties have been faced in the 1-dimensional case.

A digitised image rotuinely contains around 250,000 elements or *pixels*; there is thus a very significant saving to be made by using the FFT to implement convolution or masking as soon as the support of the mask becomes of significant size. We conclude with two examples of image masks which clearly do useful things, and which normally would be implemented in this way.

6.8.1 Gaussian Blur

As we saw in Example 6.8, an obvious use for masking is to implement a smoothing or blurring. Such an effect is easy to see on an image. We now approximate a normal, or Gaussian blur, obtained perhaps using a "soft" lens so the light spreads out on the focal plane, rather than all going to the "correct" spot. Such a Gaussian, of variance σ^2 is of the form

$$f(j,k) = \frac{1}{\sqrt{2\pi\sigma}} \exp\left(\frac{-(j^2+k^2)}{2\sigma^2}\right),$$

where conventionally we assume the function is zero outside a neighbourhood of radius 3σ. If we take $\sigma = 1$ we get the mask

$$\frac{1}{1003}\begin{pmatrix} 0 & 0 & 1 & 2 & 1 & 0 & 0 \\ 0 & 3 & 13 & 22 & 13 & 3 & 0 \\ 1 & 13 & 59 & 97 & 59 & 13 & 1 \\ 2 & 22 & 97 & 159 & 97 & 22 & 2 \\ 1 & 13 & 59 & 97 & 59 & 13 & 1 \\ 0 & 3 & 13 & 22 & 13 & 3 & 0 \\ 0 & 0 & 1 & 2 & 1 & 0 & 0 \end{pmatrix}.$$

The result of applying this mask, considered as centered on the largest value, to the image of Fig 6.1 is shown in Fig. 6.2. It is clear that a noticeable blur has been obtained as was expected.

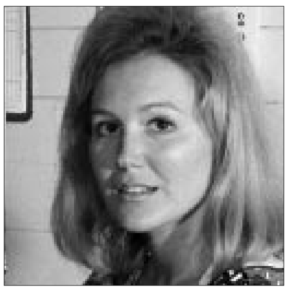


Figure 6.1: Input image.

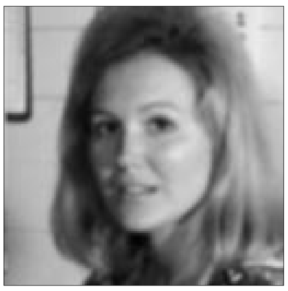


Figure 6.2: Output after Gaussian filtering.

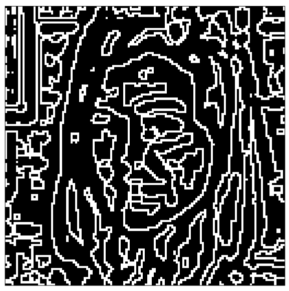


Figure 6.3: The zero-crossings of the image filtered by a difference of Gaussians.

It may be less clear why there is any interest in blurring an image. However it can form a useful intermediate step. The image shown in Fig 6.1 was filtered using a difference of Gaussians (DoG filter) at two different scales. The resulting filter is supposed to respond in a way very simlarly to the receptive fields in our eyes. The output is no longer always positive, so does not represent an image, but the pixels where the intensity changes sign are marked in Fig 6.3. The overall effect is that of a neurophysiologically plausible edge detector.

6.8.2 The Laplace Operator

We give one more example of why (linear) digital filtering, implemented via the FTT may be of interest. This involves the Laplacian filter given by the mask

$$\frac{1}{5}\begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix}.$$

This is the digital analogue of the negative Laplacian operator, ie

$$-\left(\frac{\partial}{\partial x^2}+\frac{\partial}{\partial y^2}\right).$$

The second derivative is evaluated as

$$\Big(f(y,x+\delta x)-f(y,x)\Big)-\Big(f(y,x)-f(y,x-\delta x)\Big).$$

Adding two orthogonal such approximations gives the mask shown. An example of the output of the Laplacian filter is given in Fig. 6.4.

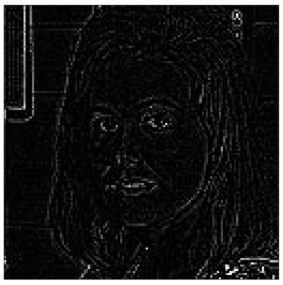


Figure 6.4: Output of Laplacian filter.

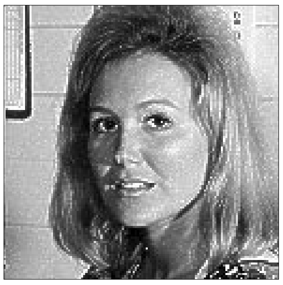


Figure 6.5: Output of sharpening filter.

There is another interpretation of the operator, which although completely unjustifiable, may indicate why the output from the Laplacian gives a “sharpened” version of the original image. We have seen above how to create an image blur(f) from f, using, for example, the mask

$$\frac{1}{5}\begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix}.$$

It can thus be argued that ($f-$blur(f)) defines a function called, perhaps sharp(f). But sharp(f) has mask

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}-\frac{1}{5}\begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix},$$

and this is precisely the Laplacian mask. We can thus argue that the Laplacian sharpens. In practice, the image in Fig. 6.4 is almost dark; we are seeing *only* the sharpness; to see more useful output, we need to add the sharpness back to the original image; this gives Fig. 6.5.

Summary You are probably beginning to recognise this sort of processing. For example it is now commonplace in television titles. Such processing must be done very quickly, even though the images are large, with perhaps $400,000$ elements in each. The FFT algorithm is sufficiently fast to do this in a pipeline, while pointwise multiplication can be very simply. The combination enables "special effects" processing to be done routinely, in situations where there is insufficient time to implement filtering using the naive "muliply and add at every point" algorithm.

Questions 6 (Hints and solutions start on page 105.)

Q 6.1. For each $l \in \mathbb{Z}^n$ define the **shift operator** S_l on $\mathbb{C}^n$ by $S_l f(j) = f(j-l)$, and as before, write $\omega = \exp(-2\pi i/n)$. Prove that

$$\mathcal{F}_n(S_l f)(p) = \omega^{lp}\mathcal{F}_n f(p)$$

Q 6.2. The Fourier matrices $\mathcal{F}_2$ and $\mathcal{F}_3$ are

$$\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix} \quad \text{and} \quad \frac{1}{\sqrt{3}}\begin{pmatrix}1 & 1 & 1\\ 1 & \omega & \omega^2\\ 1 & \omega^2 & \omega\end{pmatrix},$$

where

$$\omega = -\frac{1}{2} - \frac{i\sqrt{3}}{2} = \cos\left(\frac{2\pi}{3}\right) - i\sin\left(\frac{2\pi}{3}\right).$$

Check that these are both unitary matrices directly, and write down $\mathcal{F}_4$ and $\mathcal{F}_5$.

Q 6.3. Let $\mathbf{a} = (a_0, a_1, \dots, a_{n-1}, 0, \dots, 0)$ and $\mathbf{b} = (b_0, b_1, \dots, b_{n-1}, 0, \dots, 0)$ be points in $\mathbb{C}^{2n}$ Show that

$$\mathbf{a} \star \mathbf{b} = \big(a_0b_0, (a_1b_0 + a_0b_1), (a_2b_0 + a_1b_1 + a_0b_2), \cdots,$$
$$(a_{n-1}b_0 + a_{n-2}b_1 + \cdots + a_0b_{n-1}), \cdots, a_{n-1}b_{n-1}, 0\big).$$

Deduce that if $\mathbf{a}$ and $\mathbf{b}$ are the coefficients of polynomials $p(x)$ and $q(x)$ respectively, as in Equation6.1 then $p(x)q(x)$ has coefficients $\mathbf{a} \star \mathbf{b}$.

Q 6.4. Write out a full implementation of an 8 - point discrete Fourier transform.

Q 6.5. One way to look at the Fourier transform of a polynomial, considered as a point in $\mathbb{C}^n$, is in terms of evaluation of the polynomial at a fixed set of n fixed points. Verify this, and determine the fixed set of points.

Q 6.6. Let $z^n = 1$, and assume that $z \neq 1$. Verify that

$$1 + z + z^2 + \cdots + z^{n-1} = 0.$$

Let $\omega = \exp(-2\pi i/n)$. Verify that for any integer $n \geq 1$, the Fourier matrix $\mathcal{F}_n$ defined by

$$\mathcal{F}_n = \frac{1}{\sqrt{n}}\begin{pmatrix}1 & 1 & 1 & \dots & 1\\ 1 & \omega & \omega^2 & \dots & \omega^{n-1}\\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(n-1)}\\ \vdots & \vdots & \vdots & \ddots & \vdots\\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \dots & \omega^{(n-1)(n-1)}\end{pmatrix}$$

has inverse $\bar{\mathcal{F}}_n$, the complex conjugate of $\mathcal{F}_n$.

Explain how this fact can be used to derive an algorithm which takes time $O(n\log n)$ to multiply two polynomials of degree n. You are not required to prove anything but you should ensure that your notation is defined and that results you use, such as the convolution theorem, are clearly stated.

Appendices

Big O

We often end up with messy timing formulas like

$$T(n) = 2.456n^2 + \pi \ln(3n) + \frac{5}{7}[\sqrt{n+1}].$$

This sort of detail is rarely of any use to anyone. Users of algorithms usually want simpler results that are easier to understand. They want to know whether a given program is going to take seconds, minutes or years to complete. They are not usually interested in the difference between 1.23 seconds and 1.24 seconds.

Now, in the above formula the last two terms are quite small compared to the first, particularly if n is large.

For $n = 10$ it gives $T(10) = 245.6 + 10.69 + 2.14$

For $n = 100$ it gives $T(100) = 24560 + 17.92 + 7.14$

We would like to be able to say that $T(n)$ behaves essentially like n^2 (modulo a proportionality factor). This is much easier to understand and is essentially true.

I now want to introduce a notation that goes some way towards making this idea precise.

Let $f, g : \mathbb{R} \to \mathbb{R}$ be functions. We say that $f = O(g)$ if there are $x_0, M > 0$ such that

$$|f(x)| < M|g(x)| \quad \text{for} \quad x > x_0$$

i.e. there is some fixed multiple of g that eventually dominates f. Another way to say this is that $f(x)/g(x)$ is eventually bounded.

For example

$$\begin{aligned} n^2 + n + 1 &= O(n^2) \quad \text{since, for} \quad n > 1, \quad 3n^2 > n^2 + n + 1 \\ &= O(n^3) \quad \text{even more so} \\ &\neq O(n) \quad \text{because} \quad \frac{n^2+n+1}{n} \to \infty \quad \text{as} \quad n \to \infty \end{aligned}$$

Similarly, $n^3 - 4n^2 + n = O(n^3)$ and $\neq O(n^2)$.

Note that $n^2 + n + 1 = O(n^{100})$ is true, but not very helpful. We usually try to make Big-O statements as 'limiting' as possible.

A standard kind of statement in this subject is: *Algorithm A has $T(n) = O(n^2)$ and algorithm B has $T(n) = O(n^3)$. So A is the faster algorithm.* Be careful! This is usually correct, but could in some cases be misleading. To take a very silly example: if $f(n) = 10^{10}n^2$ and $g(n) = 10^{-10}n^3$ then $f = O(n^2)$ and $g = O(n^3)$ but $f(n) > g(n)$ so long as $n < 10^{20}$. In real life the proportionality factors are rarely that different, so Big-O statements are usually fairly safe for reasonably large values of n.

Solutions to Exercises

Solutions for Questions 1 (page 15).

Solution 1.1: Strassen's method can be found at (Sedgewick 1988, Page532).

Solution 1.2: The only two problems that can really occur have to do with the hatching line not meeting an edge 'cleanly'. The first is the case where an edge of the polygon is horizontal and happens to coincide over its length with the hatching line. In this case there is no unique intersection point. The best approach in this case is to ignore this edge — i.e. say that it does not intersect the hatch line.

The other problem comes when a hatch line passes through a vertex of the polygon. That vertex belongs to two edges, so it will be counted twice and add two points to the intersection list when you only really mean to add one point. This will, in general, screw up the sequence.

Solution 1.3: No solution offered.

Solution 1.4: The only real difference is in the algebra. The equations of our hatch-lines are now something like $y = mx + h$ where $m = \tan\alpha$. We now have a slightly more complicated problem in deciding whether or not a hatch-line meets an edge, but that's just algebra.

Solution 1.5: No solution offered.

Solution 1.6: No solution offered.

Solution 1.7: No solution offered.

Solution 1.8: No solution offered.

Solution 1.9: There seem to be two useful things to do — initialise carefully, and check each array element against the known *second* smallest element. Here is one reasonable attempt

```
algorithm twoSmallest(array x, length n)
    if n < 2 return (fail)
    // First set up min and min2 so we have min <= min2
    if (x(1) < x(2)) then
min  = x(1)
min2 = x(2)
    else
min  = x(2)
min2 = x(1)
    endif
    // and now check each element to see if it disturbs the definition
    // of the two as the lowest and second lowest.
    while (i <= n) do
if (x(i) < min2) then
    if (x(i) < min) then  // we have a new lowest
min2 = min // the second smallest must also change
min  = x(i)
    else // we have a new second lowest
min2 = x(i)
    endif
endif
    endwhile
    return min, min2
end
```

It *may* be better to maintain pointers to the two array entries which contain the required minima, but the gain, if any, is minimal.

Solution 1.10: No solution offered.

Solution 1.11: No solution offered.

Solution 1.12: To get an explicit formula for A^n_{n-2}, write $a_n = A^n_{n-2}$ and note from the recurrence relation that

$$a_n = (n-1)A^{n-1}_{n-2} + a_{n-1} = (n-1).1 + a_{n-1}$$

on using the fact that $A^n_{n-1} = A^{n-1}_{n-2} = 1$. Thus since $a_2 = 1$ we have

$$\begin{aligned} a_n &= (n-1) + (n-2) + \cdots + 2 + 1, \\ &= \frac{n(n-1)}{2}. \end{aligned}$$

Note that $a_3 = 3$, $a_4 = 6$ and $a_5 = 10$ as given in the table.

We can also ask about A^n_{n-3}. With the above experience, lets call this b_{n-1}. Then from the recurrence relation, we have $b_n = A^n_{n-2} = b_{n-1}$ and $b_2 = A^3_0 = 6$. Thus

$$\begin{aligned} b_n &= \frac{n^2(n-1)}{2} + b_{n-1} \\ &= \frac{1}{2}\sum_{k=3}^{n} k^2(k-1) + 6 \\ &= \frac{1}{2}\sum_{k=1}^{n} k^2(k-1) + 6 - 4/2 \\ &= \frac{1}{2}\left(\frac{n^2(n+1)^2}{4} - \frac{n(n+1)(2n+1)}{6}\right) + 4 \\ &= \frac{n(n+1)}{24}(3n^2 - n - 4) + 4. \end{aligned}$$

Clearly more off-diagonals can be evaluataed in this way if necessary.

Solution 1.13: No solution offered.

Solution 1.14: The algorithm is as follows:

```
algorithm min(x,n) // to find the minimum of x =(x(1),...,x(n))
begin
  min = x(1)
  for i = 2 to n begin
    if ( x(i)< min )
      min = x(i)
  end
  return (min)
end
```

In order to analyse the timing, we present the algorithm as a flowchart:

We just want to count how many times each box is executed. Boxes A and B are only executed once. Boxes C, E and F are on a loop that the program goes round $n-1$ times, so each of these is executed $n-1$ times. That leaves box D; suppose it is visited d times. Clearly the minimum value of d is zero, which occurs whenever the list has its smallest element in first place. And the maximum value of d for a list of n elements is $n-1$. This will occur when the

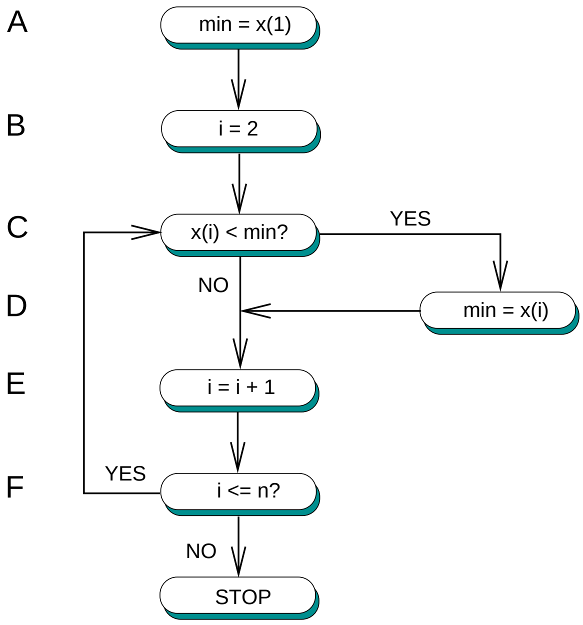


Figure 6: Flow chart to find the minimum element in a list.

list is in strictly decreasing order, so that the current value of the minimum has to be updated on each step.

Thus the time $T(n)$ taken by the algorithm to find the minimum of n numbers lies in the range

$$3(n-1)+2 \le T(n) \le 3(n-1)+2+(n-1)$$

or

$$3n-1 \le T(n) \le 4n-2.$$

In order to derive "average" values we assume that the lists that we are dealing with are all the permutations of the list $(1,2,3,4,\dots,n)$. There is nothing limiting about this assumption. We further assume that *all* these permutations are *equally likely* to be presented to the algorithm. Clearly $T(2) = 2.5$ since step D will be visited on half of all "average" inputs. Thus $\mu(2) = \frac{1}{2}$, and so

$$\mu(n) = \frac{1}{2} + \frac{1}{3} + \dots \frac{1}{n}.$$

Consider now the diagram in Fig 7.

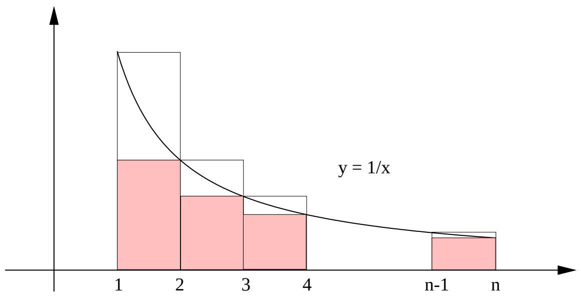


Figure 7: Graph of $y = 1/x$.

By adding up the areas of the rectangles that lie *above* the graph we get

$$1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-1} > \int_1^n \frac{dx}{x} = \ln n.$$

and so $\mu(n) > \ln n - 1 + \frac{1}{n}$. Similarly, by adding up the areas of the rectangles that lie *below* the graph we get

$$\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} < \int_1^n \frac{dx}{x} = \ln n$$

and $\mu(n) < \ln n$. Combining these we have

$$\ln n - 1 + \frac{1}{n} < \mu(n) < \ln n.$$

Thus

$$3n - 2 + \ln n + \frac{1}{n} < T(n) < 3n - 1 + \ln(n).$$

In particular, if $n = 100$ we have $302.6053 < T(n) < 303.6052$. [And the fact that this estimate can be much improved was noted in the text.]

Solution 1.15: Here is pseudocode for the minimum and second minimum element algorithm. I don't think it is in the spirit of this question to check that the list has at least three elements etc!

```
algorithm min_and_second(x)
  // To find the smallest and second smallest of x = (x(1)...x(n))
  begin
    if (x(1) < x(2)) then
      min = x(1)
      second = x(2)
    else
      min = x(2)
      second = x(1)
    endif
    for i = 3 to n begin
      if (x(i) < min) then
        second = min
        min = x(i)
      else if (x(i) < second) then
        second = x(i)
        endif
      else
        // do nothing
      endif
    end
    return (min AND second)
 end
```

Let $T(n)$ be the time taken to execute on a list of length n. The setup phase takes 1 test and 2 assigments. In the main loop, executed $(n-2)$ times, there is an initial assigment of `i`, a final test to see if `(i == n)` and either

- two tests and no assigments; or

- two tests and one assignment; or
- one test and two assigments.

giving either 4 or 5 operations in a given iteration. We ignore the cost of the return statement. Thus the total operation count satisfies

$$3 + 4(n-2) \leq T(n) \leq 3 + 5(n-2) \qquad \text{or} \qquad 4n - 5 \leq T(n) \leq 5n - 7.$$

In either case, and hence in general, the algorithm is $O(n)$.

We shall refer to either of the second two cases as involving a *detour*. Our aim is to find the "average" number of detours d; since $T(n) = 4n - 5 + d$, this will give the "average" time to complete the algorithm.

Note that the actual numbers in the list are irrelevant. All that matters is their relative ordering. Thus we assume that we are dealing with lists of n distinct elements (we ignore the possibility that the lists with repeated elements). Then we may as well assume that the lists that we are dealing with are all the permutations of the list $(1, 2, 3, 4, \dots, n)$. In addition assume that *all* these permutations are *equally likely* to be presented to the algorithm. Thus we seek the average number of detours executed when applying the algorithm to the permutations of $(1, 2, 3, \dots, n)$, granted that all such permutations are equally probable.

We now compute the number of detours that occur when applying the algorithm to each permutation in S_4. Note that a detour occurs precisely when we encounter a new minimum element in either third or fourth place in the original list. The computation is given in Table 1 We see there are 28 detours in all so the expected number of detours is 7/6 and the average time

π	d	π	d	π	d	π	d
(1,2,3,4)	0	(2,1,3,4)	0	(3,1,2,4)	1	(4,1,2,3)	1
(1,2,4,3)	0	(2,1,4,3)	0	(3,1,4,2)	1	(4,1,3,2)	2
(1,3,2,4)	1	(2,3,1,4)	1	(3,2,1,4)	1	(4,2,1,3)	1
(1,3,4,2)	1	(2,3,4,1)	1	(3,2,4,1)	1	(4,2,3,1)	2
(1,4,2,3)	1	(2,4,1,3)	1	(3,4,1,2)	2	(4,3,1,2)	2
(1,4,3,2)	2	(2,4,3,1)	2	(3,4,2,1)	2	(4,3,2,1)	2

Table 1: Counting detours for the elements of S_4

to sort a list of length 4 is $13\frac{1}{3}$ units.

Solutions for Questions 2 (page 33).

Solution 2.1: No solution offered.

Solution 2.2: No solution offered.

Solution 2.3: No solution offered.

Solution 2.4: No solution offered.

Solution 2.5: No solution offered.

Solution 2.6: No solution offered.

Solution 2.7: No solution offered.

Solution 2.8: No solution offered.

Solution 2.9: No solution offered.

Solution 2.10: No solution offered.

Solution 2.11: No solution offered.

Solution 2.12: No solution offered.

Solution 2.13: No solution offered.

Solution 2.14:

a) We start with our list $\{x_1, \ldots, x_n\}$, pick some element in the list, called the *separator*, and then rearrange the list so that all the elements that are less than or equal to the separator come before it and all the elements that are greater than the separator come after it. Having done this we then recursively apply the algorithm to each of these sublists. The recursion continues along any branch until its sublist shrinks down to zero or one element. The outline of the program is:

```
algorithm quicksort(x,lo,hi)
// to sort $x_{lo}, \ldots, x_{hi}$
begin
  if hi > lo begin // i.e. if there is anything to sort
    choose separator s from list
    separate out the list into the form
    $(x_{lo}, \ldots, x_k, \quad s \quad , x_{k+2}, \ldots, x_{hi})$
    quicksort(x,lo,k)
    quicksort(x,k+2,hi)
  end
end
```

The whole list is then sorted with a call to `quicksort(x,1,n)`. The choice of s can be made at random from the list.

We show the various stages in sorting the given list, drawing a box round the separator element. At each stage the list is shown after the call to `separate` and before recursive calls to the two sublists.

6	4	8	2	11	5	3	7	9	1	10
4	2	5	3	1	$\boxed{6}$	8	11	7	9	10
2	3	1	$\boxed{4}$	5	$\boxed{6}$	7	$\boxed{8}$	11	9	10
1	$\boxed{2}$	3	$\boxed{4}$	5	$\boxed{6}$	7	$\boxed{8}$	9	10	$\boxed{11}$
1	$\boxed{2}$	3	$\boxed{4}$	5	$\boxed{6}$	7	$\boxed{8}$	$\boxed{9}$	10	$\boxed{11}$

Note that the last sort was necessary; although the previous line was in fact sorted, this depended on the accidental positioning of 10 after 9 in the original array.

b) We assume that the initial list is a permutation of $\{1, \ldots, n\}$ and that all permutations are equally likely. Suppose the list splits into sublists of lengths k and $n - 1 - k$; this leaves one left over for the separator. Then on average

$$T_n = \alpha n + T_k + T_{n-1-k}$$

By our assumptions, all possible values of k are equally likely. We thus remove the dependence on k by taking the average over all the possibilities:

$$T_n = \alpha n + \frac{1}{n} \sum_{k=0}^{n-1} (T_k + T_{n-1-k}).$$

Since each of the T_k's appears twice in the sum, we obtain

$$T_n = \alpha n + \frac{2}{n}\sum_{k=0}^{n-1} T_k$$

as required. Using this relationship twice,

$$\begin{aligned} \frac{n}{2}(T_n - \alpha n) &= T_0 + \cdots + T_{n-1} \\ \frac{n+1}{2}(T_{n+1} - \alpha(n+1)) &= T_0 + \cdots + T_{n-1} + T_n. \end{aligned}$$

Subtracting these and rearranging gives

$$\frac{T_{n+1}}{n+2} = \frac{T_n}{n+1} + \frac{\alpha(2n+1)}{(n+1)(n+2)}.$$

c) Although insertion sort is simple to program without error, it runs in time $O(n^2)$, where n is the number of elements in the list. Although more complicated as we saw above, Quicksort has an average running time of $O(n \log n)$, and is very significantly faster for long lists than insertion sort.

Heapsort is another sorting method which runs in time $O(n \log n)$, and has the advantage that this also applies to the worst case running time, while Quicksort requires time $O(n^2)$ in the worst case. However the constant of proportionality is typically twice as big for Heapsort, so in general Quicksort is sufficiently faster to be the method of choice when speed is important, and there is no reason to believe the input is unusual.

Solutions for Questions 3 (page 49).

Solution 3.1: Pop the stack and push onto an auxiliary stack until the first stack is empty, and you can get at the bottom element. Then push them all back again, noting that at least they come out in the right order. So if there are n elements on the stack, it takes $2n$ operations to remove the next element from the queue: not a very bright idea!

Solution 3.2: Add at the end of the array, remove at the front, and roll round to avoid running out of space before the array is full. This means there is need for two points to be kept current all the time, say `f`, pointing to the front of the queue, and `b`, pointing to the back, where new elements will be added. If `f` < `b` things are as you expect, while if `f` > `b`, the queue "really" stretches from `f` to `b + N`, where `N` is the size of the array.

Solution 3.3: Since there are more red tokens than any other, we re-arrange the algorihtm to avoid performing a swap when we meet a red token. Here is the new algorithm:

```
b = w = 1;r = N; // initialise as before
while (w < r + 1) begin
  if (R(r)) r = r-1 // it was a red token
  else if (W(r)) begin // it was a white token
    swap(w,r);w=w+1
    end
  else begin // it was a blue token
    swap(b,w);swap(r,b);b=b+1;w=w+1
  end
end // while
```

The action on seeing a blue token is first to move region W up by swapping it's first member with the unknown token at the bottom of region X, and then to swap that unknown token with the one at the bottom of region R, which has just proved to be blue.

The total number of calls to `swap` is thus $|W|+2|B|$. This is faster than the original algorithm provided

$$|W| + 2|B| \leq |B| + |R|$$

and this occours iff $|W| + |B| \leq |R|$; in other words, when at least half of the tokens are red.

Solution 3.4: It is the preorder traversal.

Preorder:-	`[A B E F G H C I D]`
Inorder:-	`[E B F G H A I C D]`
Postorder:-	`[E F G H B I C D A]`

Solution 3.5: To pass from a "left-right children" description to a "terminal node" description, replace the empty tree by one consisting of a single terminal node. Next repleace every child node which is a leaf node by one having two children, each of which is a terminal node. Finally, if a node has a left child, add a terminal right node etc.

To pass from the "terminal node" description to the "left-right children" description, simlpy delete all terminal nodes.

Solution 3.6: Try it - going to Sedgewick if necessary for more help.

Solution 3.7: Note that it is not unique; where two symbols have the same frequency, it is necessary to make exactly the same choice as shown in the diagram in order to exactly reproduce the tree given in the notes.

Solution 3.8: A Huffman code is used when the input is from a limited character set, and contains information that is redundant; a typical example is English text, in which the letter "e" occurs significantly more often than random. One such application is in the "gzip" utility which is based on the Huffman code.

We describe how to build the binary Huffman code by building the corresponding binary tree. We start by analysing the message to find the frequencies of each symbol that occurs in it. Our basic strategy will be to assign short codes to symbols that occur frequently, while still insisting that the code has the prefix property. Our example will be build around the message

A TEST EXAMINATION ANSWER

The corresponding frequencies are given in Table 6.8.2; we write the space symbol " ", in the table as ␣.

A	T	E	S	X	M	I	N	O	W	R	␣
4	3	3	2	1	1	2	3	1	1	1	3

Table 2: Symbol frequencies used to build a Huffman Code.

Now begin with a collection (a forest) of very simple trees, one for each symbol to be coded, with each consisting of a single node, labelled by that symbol, and the frequency with which it occurs in the string. The construction is recursive: at each stage the two trees which account for the least total frequency in their root nodes are selected, and used to produce a new binary tree. This has, as its children the two trees just chosen: the root is then labelled with the total frequency accounted for by both subtrees, and the original subtrees are removed from the forest. The construction continues in this way until only one tree remains; that is then the Huffman encoding tree.

N	␣	A	X	M	O	W	T	E	R	S	I
3	3	4	1	1	1	1	3	3	1	2	2
			2		2				3		2
				4			6			5	
6				8							
			14						11		
					25						

Table 3: Symbol frequencies used to build a Huffman Code.

[NB The particular order in which the symbols are displayed was determined by trial and error in order to present the result in a tabular form in which only adjacent columns were merged. It is not necessary to give a presentation in this form; crossing are perfectly acceptable.]

We now code the word ANSWER using this coding tree. Although this is a prefix code and so spaces are *not* required, we include spaces between letters for clarity. We read up from the bottom of the table, assigning 0 to a choice of a left hand branch, and 1 to a choice of the right hand branch. The resulting code is then

010␣000␣1101␣01111␣101␣1100

The Huffman code has the prefix property; no character code is the prefix, or start of the the code for another character. In the same way that the Huffman code was constructed, we can associate a binary code with the prefix property to any binary tree. Given any binary tree with the same set of leaf nodes, and thus able to code the same symbols, the Huffman code is *optimal* in the sense that the corresponding coded message length is at least as short as that from the given tree.

Solution 3.9: A priority queue is a collection of elements or items, each of which has an associated priority. The operations available are:-

`create` creates an empty priority queue;

`add(item)` adds the given `item` to the priority queue; and

`remove` removes the item with the highest priority from the queue.

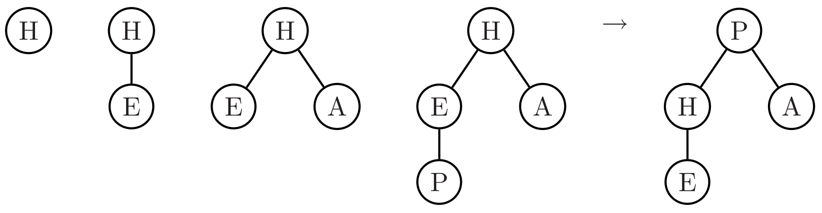
A complete binary tree is a tree which is either empty, or one in which every node:

- has no children; or
- has just a left child; or
- has both a left and a right child.

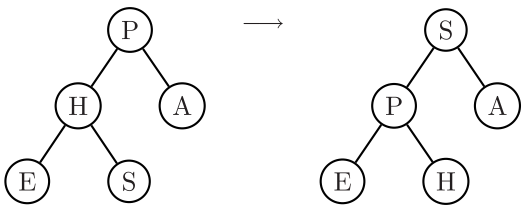
In addition, we require that all the levels, except perhaps the last, has both a left and a right child; while on the last level, any missing nodes are to the right of all the nodes that are present.

If we have a representation of a priority queue as a complete binary tree in which each node contains an element and its associated key or priority, it satisfies the*heap condition* if at each node, the associated key is larger than the keys associated with either child of that node.

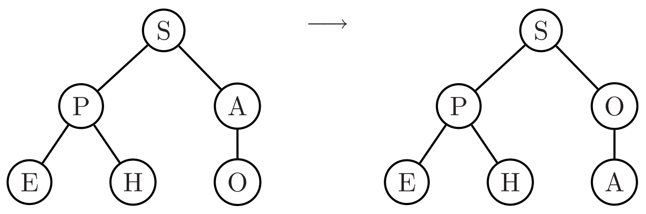
We now create a priority queue, represented as a complete binary tree, in which letters at the end of the alphabet have priority. When the first three characters are added as additional nodes in the binary tree, the new tree already satisfies the heap condition. Adding “P” violates that condition; “P” thus has to rise until it is again satisfied.



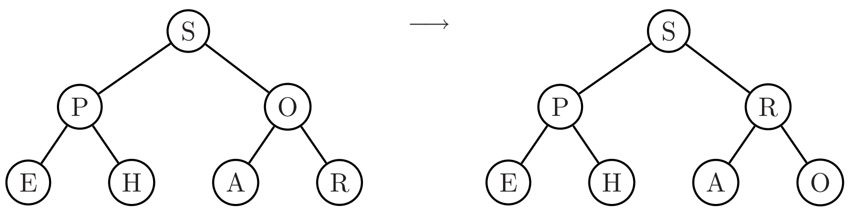
When "S" is added, it rises to the top of the tree before the heap condition holds.



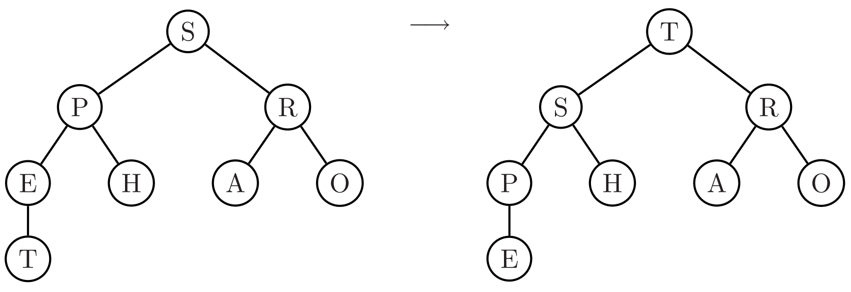
Now add "O"; it has to rises to be above "A"



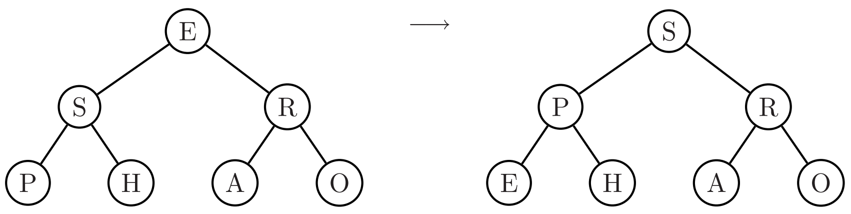
Now add "R"; this replaces "O" and fills the tree at level 2.



The final addition of "T" completes the tree, when it has been re-heaped.



The string is now sorted by removing the root node — this is the item with the highest priority remaining in the tree — and replacing it with the "last" node in the tree. The tree no longer satisfies the heap condition, and so the node that has just been promoted is allowed to fall again until the condition holds.



The root of the tree again contains the next character in order and is removed. This continues until the whole heap has been destroyed.

Finally, note that although in general Heapsort is slower than Quicksort by a factor of about 2, the "worst case" behaviour remains $O(n \log n)$, whereas Quicksort can become $O(n^2)$.

Solution 3.10: A Huffman code is used when the input is from a limited character set, and contains information that is redundant; a typical example is English text, in which the letter "e" occurs significantly more often than random. One such application is in the "gzip" utility which is based on the Huffman code.

We describe how to build the binary Huffman code by building the corresponding binary tree. We start by analysing the message to find the frequencies of each symbol that occurs in it. Our basic strategy will be to assign short codes to symbols that occur frequently, while still insisting that the code has the prefix property. Our example will be build around the message

A HUFFMAN EXAMPLE PLEASE

The corresponding frequencies are given in Table 6.8.2; we write the space symbol " ", in the table as ␣.

A	␣	H	U	F	M	N	E	X	P	L	S
4	3	1	1	2	2	1	4	1	2	2	1

Table 4: Symbol frequencies used to build a Huffman Code.

Now begin with a collection (a forest) of very simple trees, one for each symbol to be coded, with each consisting of a single node, labelled by that symbol, and the frequency with which it occurs in the string. The construction is recursive: at each stage the two trees which account for the least total frequency in their root nodes are selected, and used to produce a new binary tree. This has, as its children the two trees just chosen: the root is then labelled with the total frequency accounted for by both subtrees, and the original subtrees are removed from the forest. The construction continues in this way until only one tree remains; that is then the Huffman encoding tree.

A	␣	H	U	F	M	N	X	E	P	L	S
4	3	1	1	2	2	1	1	4	2	2	1
		2				2				3	
			4			4				5	
7				8					9		
			15								
					24						

Table 5: Symbol frequencies used to build a Huffman Code.

[NB The particular order in which the symbols are displayed happens to give a table with no crossings when listed in the order in which the symbols are gathered. It is not necessary to give a presentation in this form; crossing are perfectly acceptable.]

We now code the word PLEASE using this coding tree. Although this is a prefix code and so spaces are *not* required, we include spaces between letters for clarity. We read up from the bottom of the table, assigning 0 to a choice of a left hand branch, and 1 to a choice of the right hand branch. The resulting code is then

110␣1110␣10␣000␣1111␣10

A binary tree with 2^n nodes must have at least n levels, so the longest string must have at least n symbols. This bound is actually attained if the tree is a complete binary tree; one circumstance in which this will occur is if all the letters in the initial alphabet are equally probable.

To get the longest possible string, the tree must be completely unbalanced, with (for example) every left node being a leaf node, and with all growth taking place at the right. This means that apart from the final level, we have one symbol of each length, and so the longest symbol must have length $2^n - 1$. If the corresponding probabilities are $p_0 \geq \ldots \geq p_k$, where $k = 2^n$, then to ensure the tree builds in the way specified, we must have

$$p_i \geq \sum_{j=i+1}^{k} p_j$$

so the probability of a given symbol must be at least as great as the sum of the probabilities of all the less likely symbols.

Solutions for Questions 4 (page 62).

Solution 4.1: No solution offered.

Solution 4.2: No solution offered.

Solution 4.3: Here is my attempt. I first give the result, and only then suggest how to think about it.

- The alphabet is $A = \{\mathrm{a}\}$.
- The abstract alphabet is $\mathcal{A} = \{\sigma, B, C, D, E, S, T\}$.
- The initial symbol is σ.
- The productions are
  1. $\sigma \to \mathrm{a}$;
  2. $\sigma \to \mathrm{a}BT$;
  3. $T \to E$;
  4. $E \to \mathrm{a}$;
  5. $B\mathrm{a} \to \mathrm{a}B$;
  6. $BE \to \mathrm{aa}E$;
  7. $BT \to C\mathrm{aa}BS$;
  8. $BC \to C\mathrm{aa}B$;
  9. $\mathrm{a}C \to \mathrm{aa}DB$;
  10. $DB \to BD$;
  11. $\mathrm{a}D\mathrm{a} \to \mathrm{aa}D$; and
  12. $DS \to BT$.

Clearly this has something behind it. Start by thinking of T, S and E as terminal symbols. One is always present, in the last position, until the final production, when the string is made fully concrete. Note that only production 6 decreases the number of B's, so production 4 cannot be invoked until all B's have been removed.

The productions are based on the identity $(n+1)^2 = n^2 + 2n + 1$, and the idea behind it is that "at rest", when the string contains only 'a's, 'B's and a final T, there are n^2 copies of a, then n copies of B, (think of each B as aa) and finally one T - thought of also as a. For example, applying productions 2, 7, 9, 11, 10 and 12 leads to $aaaaBBT$. At this point a choice is made between production 7, which eventually increases the string from length n^2 to length $(n+1)^2$, and production 3, which makes the string fully concrete. Once production 3 has been used, only productions 4, 5 and 6 are relevant.

If the choice is made to apply production 7, the only option is to pass C through all the B's using production 8 until it converts to a D and then pass this D back down until it hits S. Each time C passes through B, an additional 2 a's are created, so as C passes up the string, an additional $2n$ copies of a are created as needed. The $(2n+1)^{\text{th}}$ extra a is then added before before passing D back to the terminator S, where it becomes a T again to repeat the process.

As you can see, production rules form an unusual programming language, but the principles are not very different from many others.

Solution 4.4: The question asked for a grammar, and did *not* ask for a proof that the grammar gave what was wanted — for the same reason that we didn't prove that the grammar AE gave arithmetic expressions; there was no other formal definition with which to compare.

Here is a suitable grammar.

- The alphabet is $A = \{0, 1, 2, \ldots, 9\} \cup \{+, -, .\}$;
- The abstract alphabet is $\mathcal{A} = \{$ **⟨full⟩**, **⟨optsign⟩**, **⟨unsigned⟩**, **⟨number⟩**, **⟨digit⟩**$\}$
- The initial symbol is **⟨full⟩**.
- The productions are
  1. **⟨full⟩** $\rightarrow$ **⟨optsign⟩⟨unsigned⟩**;
  2. **⟨unsigned⟩** $\rightarrow$ **⟨number⟩**.**⟨digit⟩⟨number⟩**;
  3. **⟨unsigned⟩** $\rightarrow$ **⟨digit⟩⟨number⟩**;
  4. **⟨number⟩** $\rightarrow$ **⟨digit⟩⟨number⟩**;
  5. **⟨number⟩** $\rightarrow \epsilon$, the empty string;
  6. **⟨digit⟩** $\rightarrow 0|1|2|\ldots|9$; and
  7. **⟨optsign⟩** $\rightarrow +|-|\epsilon$.

We can describe the actions to a certain extent. Starting with the initial symbol, we must pass to **⟨unsigned⟩** using production 1 so we optionally add a sign. An **⟨unsigned⟩** either contains a decimal point followed by at least one digit, or is a string of at least one digit. So we allow things like 0.0, or even .0, but dis-allow 345. in which the decimal point is not followed by a digit.

There is no real additional difficulty in coping with the exponential form $12.3E4$ provided the alphabet is extended to contain E; only one new production is needed, namely

⟨full⟩ $\rightarrow$ **⟨optsign⟩⟨unsigned⟩** E **⟨optsign⟩⟨digit⟩⟨number⟩**;

Of course this assume that things like $.7E-06$ are legal. Another way would be to have a new member of the abstract alphabet, say **⟨optE⟩**, which is defined in the obvious way to be either a correctly formatted "E" terminator, or null; this is then put at the end of the first production.

A boring solution to having a fixed length is to have a member **⟨8digits⟩** of the abstract alphabet, and an obvious production involving a **⟨digit⟩** and **⟨7digits⟩** etc.

Solution 4.5: No solution offered.

Solution 4.6: No solution offered.

Solution 4.7: No solution offered.

Solution 4.8: No solution offered.

Solution 4.9:

a) We have

$$\sigma \xrightarrow{(2)} \mathrm{a}\beta\gamma \xrightarrow{(4)} \mathrm{ab}\gamma \xrightarrow{(6)} \mathrm{abc}$$

and so abc is in $\mathfrak{L}$.

b) We first show that a^2bcbc is *not* in $\mathfrak{L}$. Any derivation must start as

$$\sigma \xrightarrow{(1)} \mathrm{a}\sigma\beta\gamma \xrightarrow{(2)} \mathrm{a}^2\beta\gamma\beta\gamma$$

since this is the only way to derive a string with exactly two copies of a. If we apply (3) at this stage, we are essentially forced to derive the string $a^2b^2c^2$, as in part (c). However making β and γ concrete involves using productions (4) to (7), and the opportunity to use each of these is limited. We must start by using (4), since the others involve b or c. The sequence

$$\mathrm{a}^2\beta\gamma\beta\gamma \xrightarrow{(4)} \mathrm{a}^2b\gamma\beta\gamma \xrightarrow{(6)} \mathrm{a}^2bc\beta\gamma$$

is then forced, and no further productions apply. Thus we cannot derive a^2bcbc.

If productions (4) to (7) are replaced by (4′) and (5′), we can derive a^2bcbc as follows:

$$\sigma \xrightarrow{(1)} \mathrm{a}\sigma\beta\gamma \xrightarrow{(2)} \mathrm{a}^2\beta\gamma\beta\gamma \xrightarrow{(4')} \mathrm{a}^2\mathrm{b}\gamma\mathrm{b}\gamma \xrightarrow{(5')} \mathrm{a}^2\mathrm{bcbc}.$$

c) We have the derivation

$$\sigma \xrightarrow{(1)} \mathrm{a}^{n-1}\sigma(\beta\gamma)^{n-1} \xrightarrow{(2)} \mathrm{a}^{n-1}\mathrm{a}(\beta\gamma)^n \xrightarrow{(3)} \mathrm{a}^{n-1}\mathrm{a}\beta^n\gamma^n \xrightarrow{(4)}$$
$$\mathrm{a}^{n-1}\mathrm{ab}\beta^{n-1}\gamma^n \xrightarrow{(5)} \mathrm{a}^n\mathrm{b}^{n-1}\mathrm{b}\gamma^n \xrightarrow{(6)} \mathrm{a}^n\mathrm{b}^{n-1}\mathrm{bc}\gamma^{n-1} \xrightarrow{(7)} \mathrm{a}^n\mathrm{b}^n\mathrm{c}^n.$$

Here, each odd labelled production is applied $(n-1)$ times, while each even labelled production is applied only once, and provides the "glue" to move to the next stage.

d) We have $\mathfrak{L} = \{a^nb^nc^n\}$; in other words, $\mathfrak{L}$ consists of an arbitrary string of a's followed by the same number of b's and concluding with the same number of c's

To see why this is so; why $a^nb^nc^n$ are the *only* strings which can be derived, we essentially follow the argument in part (b) It is not possible to make β or γ concrete unless we use production (4); since until that is used there is neither b nor c available. But production (4) cannot be used until after (2), which itself stops further use of (1). So at the end of this stage we have $a^n\beta ... \gamma$ and the intermediate symbols are equal numbers of β's and γ's. Note that we can never make concrete a string containing $c\beta$; and one will occur as it did in part (b) unless production (3) is fully utilised. In this latter case we derive $a^nb^nc^n$.

Solution 4.10:

a)

$$\sigma \xrightarrow{(1)} \alpha\beta \xrightarrow{(3)} \alpha a\gamma \xrightarrow{(4)} a^2\alpha\gamma \xrightarrow{(6)} aa$$

and so a^2 is in $\mathfrak{L}$.

b)

$$\sigma \xrightarrow{(1)} \alpha\beta \xrightarrow{(2)} \alpha\alpha\beta \xrightarrow{(3)} \alpha^2 a\gamma \xrightarrow{(4)} \alpha a^2\alpha\gamma \xrightarrow{(4)} a^2\alpha a\alpha\gamma \xrightarrow{(4)} a^4\alpha^2\gamma \xrightarrow{(5)} a^4\alpha\gamma \xrightarrow{(6)} a^4$$

and so a^4 is in $\mathfrak{L}$.

c) Consider the effect of applying a modified form of production (4) in which αa is replaced by $a\alpha$. The weight associated with this copy of a reduces from 2^{k+1} (say) to 2^k and the weights of all other a's remains the same. Now consider the effect of using the correct form of production (4). In addition to that loss of weight, and additional a is introduced, which also has weight 2^k. This restores the total weight of the string to what it was originally.

d) Note that production (1) is a forced first step and that afterwards there is exactly one β until production (3) is made, at which point there are no copies of β and only the last three productions are relevant. At this stage, the string has the form $\alpha^n\beta$ for some $n \geq 1$. We can get all such with $n \geq 1$ simply by applying production (2) a total of $n-1$ times.

Counting γ's shows we only use production (6) once, as the last step to make the string fully concrete. Until the string becomes fully concrete, γ remains as the last element. Thus production (5) can only be used at the right-hand end of the string, and so does not effect the total weight. At the start of this phase the total weight is 2^n. Just before we apply production (6), the string must be in the form $a^k\alpha\gamma$, and this has total weight k. Thus the final production is

$$a^{2^n}\alpha\gamma \xrightarrow{(6)} a^{2^n}$$

and $\mathfrak{L} = \{a^{2^n} \mid n \geq 1\}$.

Solution 4.11:

a) Here is the initial derivation.

$$\alpha \xrightarrow{(1)} a\alpha\beta\gamma \xrightarrow{(2)} a^2\beta\gamma\beta\gamma \xrightarrow{(3)} a^2\beta\beta\gamma\gamma \xrightarrow{(4)} a^2b\beta\gamma\gamma \xrightarrow{(5)} a^2b^2\gamma\gamma \xrightarrow{(6)} a^2b^2c\gamma \xrightarrow{(7)} a^2b^2c^2$$

and so $a^2b^2c^2$ is in $\mathfrak{L}$.

b) It is trivial that for the initial symbol α, our hypothesis H that no member of the abstract alphabet appears to the left of any member of A, is true. Assume inductively that H is true for the first n steps in a derivation, and note that none of the 7 productions given destroys H; it thus remains true after $n+1$ steps and the general result follows by induction.

c) There is at most one α in the string, and this always lies at the abstract concrete boundary until production (2) is used, at which point nether (1) nor (2) are relevant. So our initial productions are necessarily drawn from (1) and (3) until (2) is used, at which point, since each application of (1) introduces both an additional α and an additional $\beta\gamma$, there are exactly $n+1$ each of a, β and γ. The commutativity relation (3) can only move β to the left, so the result follows.

d) We claim that $\mathfrak{L} = \{a^{n+1}b^{n+1}c^{n+1} \mid n \geq 0\}$. One way is clear, since the following shows all the claimed strings lie in $\mathfrak{L}$.

$$\begin{aligned}\alpha &\xrightarrow{(1)} a^n\alpha(\beta\gamma)^n \xrightarrow{(2)} a^{n+1}(\beta\gamma)^{n+1} \xrightarrow{(3)} a^{n+1}\beta^{n+1}\gamma^{n+1} \xrightarrow{(4)} a^{n+1}b\beta^n\gamma^{n+1} \\ &\xrightarrow{(5)} a^{n+1}b^{n+1}\gamma^{n+1} \xrightarrow{(6)} a^{n+1}b^{n+1}c\gamma^n \xrightarrow{(7)} a^{n+1}b^{n+1}c^{n+1}.\end{aligned}$$

Here we apply productions 1,3,5 and 7 a total of n times, and the remaining ones once.

Conversely we must show that these are the *only* strings that can be produced.

Note that the argument for the second part shows we necessarily apply (1) and (3) in any order before applying (2). Consider now the stage *after* production (2) has been applied; then only (3) and (4) are relevant initially. After using (3) at most n times we necessarily use (4), at which point we introduce b into the string. We now have n more b's to make concrete, and this process can only occur at the single point where the left hand concrete symbols meet the abstract symbols on the right. Note also that as soon as we use production (6) we introduce a c at the abstract/concrete interface and so can only subsequently use production (7). We are thus forced to already have made all β's concrete before applying (7). In other words we are forced to essentially imitate the sequence of productions at the start of this section except for production (3) which can be applied at any time, provided it has been applied n times before production (6) is used.

Solutions for Questions 5 (page 71).

Solution 5.1: Since X is finite, the set $\{x_i \in X \mid i = 0, 1, 2, \ldots\}$ is itslef finite, and so there must be some pair $i, j > 0$ such that $x_i = x_{i+j}$. Choose the least such i for which this holds, and then choose the least such j. Then $x_{i+j+1} = \phi x_i + j = \phi x_i = x = i + 1$ etc and the sequence is periodic of period j starting at x_i.

Solution 5.2: There are several ways of getting a point "at random" in T_1; we look at them in turn. The most straightforward is to choose x = RAND and y = RAND, and then throw away the points not in T_1. This has the disadvantage that we throw away half of the points chosen. A quicker method depends on the fact that the line $y = x$ divides the square into two congruent triangles, and so accepts (x, y) if it lies in T_1 and otherwise accepts the point (y, x), which necessarily lies in T_1, if (x, y) does not (unless $x = y$, which we ignore, or throw away).

A third way can generalise a little more easily. First let t = RAND, write $x = \sqrt{t}$, choose $y = \text{RAND}[0, x)$ and choose the point (x, y). Clearly the chosen point lies in T_1; we have to show that this gives a random distribution. Given x, it is clear that y is chosen with uniform probability; we must thus show that x was chosen appropriately. In other words, that the probability dp of choosing a point in a strip of width dx at the point x is proportional to $x\,dx$, the area of the strip. Thus

$$dp = kx\,dx, \quad \text{and so } p = kx^2/2 = Kt.$$

Thus the probability is uniform, as required.

It is clear that the same arguments apply to choosing a point in T_2.

We now analyse ways of producing points in T_3. It is clear that Method 1 produces points chosen at random in T_3.

Method 2 is like Method 1, except that before the third call to RAND(), a check is made that $(x, y) \in T_2$, and the process abandoned if it is not. Clearly we can ommit a call to RAND() without affecting the outcome, if we know in advance that nothing will be produced. Thus this method *does* produce random points of T_2.

Note that Method 3 and Method 4 are effectively the same, although Method 4 is quicker; in each case, having picked (x, y) at random in T_2, we wait until some z with $0 \le z < 1 - x - y$ has been chosen before making another choice of (x, y). In particular, we see that the probability of a point lying in

$$\{(x, y, z) \mid (x_0 \le x \le x_0 + dx, y_0 \le y \le y_0 + dy, 0 \le z \le 1 - x_0 - y_0)\}$$

is proportional to $dx\,dy$, rather than $(1 - x_0 - y_0)\,dx\,dy$ as it should do if equal volumes are equally likely.

Random points in T_4: the triangle in question is equilateral, with side $\sqrt{2}$. We first show how to pick a point at random from the equilateral triangle T with vertices $(-1, 0)$, $(0, 1)$ and $(1, 0)$ whose side is of length 2, and then construct an affine map to the requierd triangle.

To choose a point at random in T, pick x = RAND() and $y = 2 *$ RAND(). If $(x, y) \in T$, accept it; otherwise choose $(-x, 2 - y) \in T$. This is essentially the same idea as in choosing a point of T_1.

There will be a unique affine map to T_4. A calculation shows it is

$$(x, y) \longrightarrow \left(-\frac{1}{2}x - \frac{1}{4}y + \frac{1}{2}, \frac{1}{2}x - \frac{1}{4}y + \frac{1}{2}, \frac{1}{1}y\right)$$

is the rquired map.

A less efficient, and *incorrect* way picks x', y' and z' using RAND(), so necessarily equally distributed random variables with $0 \le x', y', z' < 1$, and then use variables

$$x = \frac{x'}{x' + y' + z'}, \qquad y = \frac{y'}{x' + y' + z'}, \qquad z = \frac{z'}{x' + y' + z'}$$

Again we have $0 \le x', y', z' < 1$ equally distributed with $0 \le x, y, z < 1$ and they satisfy the required constraint. This amounts to choosing a point in the unit cube, and projecting onto the plane $x + y + z = 0$. It is now clear that there is a problem; the probability of picking a given point is proportional to the length of the line from the origin, through the point, to the edge of the unit cube, which can vary from 1, at a vertx to $\sqrt{3}$ along the main diagonal.

Solution 5.3: Assume the earth is a sphere of radius R, and that latitude and longitude are chosen at random. The area in which the point can lie if the latitude is between 0 and a very small angle $d\vartheta$ is approximately the area of a cylinder of radius R and height $R\,d\vartheta$, and so has area $2\pi R^2\,d\vartheta$. An equally likely choice is that the latitude lies between $\pi/2 - d\vartheta$ and $\pi/2$, in which case the chosen point lies in an approximately circular arc of radius $R\,d\vartheta$, with area $\pi R^2 (d\vartheta)^2$. Theses are very different; thus the choice thus does not lead to an equal probability of choosing points near the pole, and near the equator.

An easy way to choose points from the ball $x^2 + y^2 + z^2 \le 1$ is to choose each of x, y and z from RAND$(-1, 1)$ and then reject points which do not satisfy $x^2 + y^2 + z^2 \le 1$. The probability of acceptance is then the ratio of the volume of the sphere of radius 1 to the volume of the enclosing cube of side 2. This is $4\pi/3$ to 8, or 0.52. Points *on* the sphere can be chosen by first picking a point in the ball, and then projecting onto the sphere.

Alternatively, we can choose longitude from RAND$[0, 2\pi)$, and latitude from a distribution which reflects the resulting areas. for simplicity we choose a point at random in the northern hemisphere of a sphere of radius 1, and indicate at the end how the idea can be modified. Let $\vartheta = 2\pi * \text{RAND}()$, choose $x = \text{RAND}()$, and let

$$r = f(x) = \sqrt{2x - x^2}$$

Note that $f : [0, 1] \to [0, 1]$ is monotone increasing, and so has an inverse function. Then $0 \le r \le 1$, so the point with polar co-ordinates (r, ϑ) lies in the unit disc. We take this to be the equatorial disc, and pick as our point in the northern hemisphere the unique point which projects down to the point (r, ϑ). Clearly the angular distribution is uniform; to show that this choice is random, it is thus enough to show that the probability of choosing $0 \le a < r < b \le 1$ is the same as proportion of the area of the hemisphere which sits above this annulus.

To this end recall the classical result, which can also be easily proved using calculus, that the area of a spherical cap is the same as the area of a cylinder of the same radius as the sphere whose height is the height of the cap. The area of that portion of the unit hemisphere which sits above the annulus $0 \le a < r < b \le 1$ is thus

$$2\pi\left(1 - \sqrt{1 - b^2}\right) - 2\pi\left(1 - \sqrt{1 - a^2}\right) = 2\pi\left(\sqrt{1 - a^2} - \sqrt{1 - b^2}\right).$$

since the area of the hemisphere is 2π, we must show that the probability of choosing $0 \le a < r < b \le 1$ is $\left(\sqrt{1 - a^2} - \sqrt{1 - b^2}\right)$.

Let $g(r) = 1 - \sqrt{1-r^2}$, so $g(r) = x$, and f and g are inverse functions. Computing probabilities,

$$\begin{aligned}\Pr(a < r < b) &= \Pr(g(a) < g(r) < g(b)) \qquad \text{since } g \text{ is monotone,}\\ &= \Pr\left(1 - \sqrt{1-a^2} < x < 1 - \sqrt{1-b^2}\right)\\ &= \sqrt{1-a^2} - \sqrt{1-b^2} \qquad \text{since } x \text{ is uniformly distributed.}\end{aligned}$$

Of course this is the calculation used to determine f in the first place.

To map to the whole sphere, choose x at random from [-1,1); then use the sign of x to determine whether to go to the northern or southern hemispheres, and $|x|$ in place of x to determine r.

Solution 5.4: Let the matrix A have entries a_{ij} for $1 \leq i, j \leq n$. For $i \geq j$ define $a_{ij} = 2 * \text{RAND } - 1$, and let $aij = a_{ji}$ if $i < j$. Clearly A is symmetric, while each entry is uniformly distributed on $[-1, 1)$; to avoid the possibility of getting -1 as an entry, simply make another selection using the random number generator should it be chosen.

Solution 5.5: No solution offered.

Solution 5.6: Assume we have chosen $k-1$ items in the way described, and are trying to choose the k^{th} such item. We are successful at the first choice if any of the remaining $n-k+1$ items are chosen; this occurs with probability $(n-k+1)/n$. In the same way, we need two choices if we fail at the first attempt (probability $(k-1)/n$), and succeed at the second (probability $(n-k+1)/n$ as before); thus with total probability

$$\left(\frac{k-1}{n}\right)\frac{n-k+1}{n}.$$

Of course we may need three, four etc choices before we succeed. The *expected* number of choices is thus

$$\frac{n-k+1}{n}\left(1.1 + 2.\frac{k-1}{n} + 3.\left(\frac{k-1}{n}\right)^2 + \cdots\right)$$

Recall that

$$\begin{aligned}1 + 2x + 3x^2 + \cdots &= \frac{d}{dx}(1 + x + x^2 + \cdots) \qquad \text{if } |x| < 1,\\ &= \frac{d}{dx}(1-x)^{-1} = (1-x)^{-2} \qquad \text{if } |x| < 1,\end{aligned}$$

Thus the expected number of choices is

$$\frac{n-k+1}{n}\left(1 - \frac{k-1}{n}\right)^2 = \frac{n}{n-k+1}$$

as claimed.

As in the question, this then gives

$$w(n,k) = n\left(\frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{n-k+1}\right) = n(H_n - H_{n-k}).$$

Since $w(n, 1) = 1$, we have

$$\frac{w(n,k)}{k} = \frac{n}{k}\left(H_n - H_{n-k}\right)$$

and using the approximation $H_n \sim \log(n)$,

$$= \frac{1}{\alpha} \log\left(\frac{n}{n-k}\right) = \frac{1}{\alpha} \log\left(\frac{1}{1-\alpha}\right).$$

Solution 5.7:

a) We give the pseudocode for the Running Sample algorithm in Fig. **??**

```
algorithm runningsample(k, N)
// choose k items at random from N supplied
begin
  t = 0
  chosen = 0
  repeat begin
    if ((N-t)*rand() >= (k-chosen)) begin
      // pass over next item
      read in next item
      t = t + 1
    end
    else begin
      //  accept next item
      read in next item
      print out next item
      t = t + 1
      chosen = chosen + 1
    end
  end // repeat
  until (chosen = k)
end.
```

The logic of this method is that if, at the ith item, we have so far chosen j items then we have $k - j$ items left to choose from the remaining $N - i + 1$ items (including the ith). So it seems reasonable to choose the ith item with probability

$$\frac{k-j}{n-i+1}$$

and that's what the algorithm does.

We cannot end up with more than k items because we stop the loop if we have got k. It is thus enough to show that we cannot end up with *fewer* than k. Suppose conversely that we had ended up with $j < k$ items and that the last item *not* chosen was the ith. Then, at that stage, we had already chosen $j - (N - i)$ items. So the probability of choosing the ith was

$$\frac{k - (j - (N-i))}{N-i+1} = \frac{N - i + (k-j)}{N-i+1} \geq 1,$$

so we *must* have chosen it! Contradiction.

b) Our algorithm identifies the half of the equilateral triangle to the left of the x - axis with the remaining half of the square

$$S = \{(x, y) \mid 0 \leq x \leq 1, 0 \leq y \leq 2\},$$

namely the half above the line $y + 2x = 2$. Our algorithm is then

```
x = RAND();
y = RAND();
if ( y > 2 - 2*x) then
    x = -1 + x
    y = 2 - y
endif
return (x,y)
```

The pair (x, y) are chosen uniformly from a rectangle with the same area as the given triangle. Our algorithm aranges to use all the points produced, while the transformation which maps the part of the square outside the equilateral triangle to the "missing" part of the triangle is Euclidean and hence preserves areas.

Solutions for Questions 6 (page 84).

Solution 6.1: From the definition of the shift operator, we have

$$\begin{aligned}\mathfrak{F}_n(S_l f)(p) &= \frac{1}{\sqrt{n}}\sum_{j=0}^{n-1} f(j-l)\exp\left(\frac{-2\pi i j p}{n}\right)\\ &= \frac{1}{\sqrt{n}}\sum_{j=0}^{n-1} f(j-l)\omega^{(j-l)p}\omega^{lp} = \frac{1}{\sqrt{n}}\sum_{j=l}^{n+l-1} f(j-l)\omega^{(j-l)p}\omega^{lp}\end{aligned}$$

and substituting $k = j - l$, and then rewriting with j instead of k,

$$= \frac{1}{\sqrt{n}}\sum_{j=0}^{n-1} f(j)\omega^{jp}\omega^{lp} = \omega^{lp}\mathfrak{F}_n f(p).$$

Solution 6.2: No solution offered.

Solution 6.3: No solution offered.

Solution 6.4: No solution offered.

Solution 6.5: We compute from the definition of $\hat{p} = \mathfrak{F}(p)$.

$$\begin{aligned}\hat{p}(0) &= a_0 + a_1 + a_2 + \cdots + a_{n-1},\\ \hat{p}(1) &= a_0 + a_1\omega + a_2\omega^2 + \cdots + a_{n-1}\omega^{n-1},\\ \hat{p}(2) &= a_0 + a_1\omega^2 + a_2\omega^4 + \cdots + a_{n-1}\omega^{2(n-1)},\\ &\dots\\ \hat{p}(n-1) &= a_0 + a_1\omega^{n-1} + a_2\omega^{2(n-1)} + \cdots + a_{n-1}\omega^{(n-1)(n-1)}.\end{aligned}$$

Thus the fixed set of points are the nth roots of unity.

Solution 6.6: It is trivial to verify the factorisation

$$z^n - 1 = (z-1)(1 + z + z^2 + \cdots + z^{n-1}).$$

The result follows, since we are given a zero of the left hand side and $z \neq 1$.

It is enough to show that $\bar{\mathfrak{F}}_n\mathfrak{F}_n = I_n$,the identity matrix; we have then explicitly produced an inverse. Since the j^{th} row of $\bar{\mathfrak{F}}_n$ is

$$\frac{1}{\sqrt{n}}(1, \bar{\omega}^j, \bar{\omega}^{2j}, \dots, \bar{\omega}^{(n-1)j})$$

while the k^{th} column of $\mathcal{F}_n$ is

$$\frac{1}{\sqrt{n}}(1, \omega^k, \omega^{2k}, \dots, \omega^{(n-1)k}),$$

and since $\bar{\omega}^k = w^{-k}$, the entry in the $(j,k)^{th}$ place of the product is just

$$\frac{1}{n}(1 + \omega^{(j-k)} + \omega^{2(j-k)} + \cdots + \omega^{(n-1)(j-k)}).$$

We now distinguish two cases. If $j = k$, this is just $(1 + 1 + \cdots + 1)/n = 1$, while if $j \neq k$, it is zero by the lemma, since $\omega^{(j-k)}$ is an n^{th} root of unity, but is not equal to 1. It follows that the product is the identity matrix as claimed.

Let

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots a_{n-1} x^{n-1}$$

be a polynomial of degree $n-1$, which we identify with a point in $\mathbb{C}^n$ using the map $p \to (a_0, a_1, \dots, a_n)$. For $\mathbf{a}$ and $\mathbf{b}$ in $\mathbb{C}^n$, define $\mathbf{a} \star \mathbf{b} \in \mathbb{C}^n$ by

$$(\mathbf{a} \star \mathbf{b})_j = \sum_{k=0}^{n-1} a_{j-k} b_k,$$

where each index in the sum is interpreted mod n, so that for example $a_{-1} = a_{n-1}$ etc. Let $\mathbf{a} = (a_0, a_1, \dots, a_{n-1}, 0, \dots, 0)$ and $\mathbf{b} = (b_0, b_1, \dots, b_{n-1}, 0, \dots, 0)$ be points in $\mathbb{C}^{2n}$ Then

$$\mathbf{a} \star \mathbf{b} = \big(a_0 b_0, (a_1 b_0 + a_0 b_1), (a_2 b_0 + a_1 b_1 + a_0 b_2), \cdots,$$
$$(a_{n-1} b_0 + a_{n-2} b_1 + \cdots + a_0 b_{n-1}), \cdots, a_{n-1} b_{n-1}, 0\big).$$

A calculation shows that if $\mathbf{a}$ and $\mathbf{b}$ are the coefficients of polynomials $p(x)$ and $q(x)$ respectively, then $p(x)q(x)$ has coefficients $\mathbf{a} \star \mathbf{b}$, at least when embedded in C^{2n} to avoid circular wrap-around.

Let $\mathbf{a} = (a_0, a_1, \dots, a_{n-1})$ be a vector in $\mathbb{C}^n$, and define $\mathbf{F}$ in $\mathbb{C}^n$ by $\hat{\mathbf{a}} = \mathcal{F}_n(\mathbf{a})$. Since the Fourier matrix is unitary, we have $\mathbf{a} = \bar{\mathcal{F}}_n(\hat{\mathbf{a}})$, and so this transformation, the Discrete Fourier Transform, is invertible. In this context, we need the Convolution Theorem **Convolution Theorem:** Let $\mathbf{a}$ and $\mathbf{b}$ be in $\mathbb{C}^n$. Then

$$\mathcal{F}_n(\mathbf{a}) \mathcal{F}_n(\mathbf{b}) = \mathcal{F}_n(\mathbf{a} \star \mathbf{b}).$$

In other words, the discrete Fourier transform maps convolution to pointwise multiplication.

The convolution theorem suggest a different way of computing $\mathbf{a} \star \mathbf{b}$ at least when we embed $\mathbf{a}, \mathbf{b} \in \mathbb{C}^{2n}$; as the inverse transform of $\mathcal{F}_{2n}\mathbf{a} \cdot \mathcal{F}_{2n}\mathbf{b}$. In other words, we first compute the Discrete Fourier Transforms of $\mathbf{a}$ and $\mathbf{b}$, then compute the pointwise product $\mathcal{F}_{2n}\mathbf{a} \cdot \mathcal{F}_{2n}\mathbf{b}$, and finally compute the inverse Discrete Fourier Transform to obtain $\mathbf{a} \star \mathbf{b}$. It turns out by a careful arrangement of its operations, the Fourier transform can be computed using only $O(n \log n)$ operations. This is known as the Fast Fourier Transform.

Bibliography

Gersting, J. L. (1993), *Mathematical Structures for Computer Science*, third edn, W H Freeman.

Graham, R., Knuth, D. E. & Patashnik, O. (1988), *Concrete Mathematics*, Addison Wesley.

Knuth, D. E. (1981), *Seminumerical Algorithms*, Vol. 2 of *The Art of Computer Programming*, second edn, Addison-Wesley.

Knuth, D. E. (1998), *Sorting and Searching*, Vol. 3 of *The Art of Computer Programming*, second edn, Addison Wesley.

Press, W. H., Flannery, B. P., Teukolsky, S. A. & Vetterling, W. T. (1992), *Numerical recipes in C. The Art of Scientific Computing*, second edn, Cambridge University Press.

Sedgewick, R. (1988), *Algorithms*, second edn, Addison Wesley.

Sedgwick, R. (1995), *Algorithms in C++*, Addison-Wesley. Also available in C and Pascal.

v Aho, A., Hopcroft, J. E. & Ullamn, J. D. (1983), *Data Structures and Algorithms*, Addison-Wesley.

Wilensky, R. (1986), *Common LISPcraft*, Norton.

Winston, P. H. & Horn, B. K. (1989), *LISP 3rd Edition*, Addison Wesley.

Index Entries

Algorithm, 32
alphabet, 51
Arithmetic Expressions, 55
array, 38
arrays
    FFT, 80

binary tree, 42
bit reversal, 79
blur
    Gaussian, 81

child, 41
coin toss, 68
comparing n^2 and $n \log n$, 26
complete, 43
convex, 4
conveyor belt, 69
convolution, 75
Convolution Theorem, 77
Cooley, 78
course description, ix

discrete Fourier transform, 74

fast Fourier tarnsform, 77
fast Fourier transform, 77
FFT, 77
    timing, 78
filter, 79
    DoG, 81
    Laplacian, 82
Formal Grammar, 53
Formal Language, 52, 53
Fourier coefficients, 73
Fourier series, 74
Fully Bracketed Infix Expressions, 59

Gauss, 78

hatch, 4
heap condition, 47

increment, 66
inorder, 42
instrument response function, 80

leaf, 41
Lehmer, 66
Linear Congruential Generator, 66

linear congruential generators, 66
linearly ordered set, 17
lisp, 40
list, 40

mask, 79
matrix
  unitary, 74
modulus, 66
multiplier, 66

nodes, 41

parent, 41
Parseval's Theorem, 76
path length, 45
point spread function, 80
polynomials
  multiplying, 73
postorder, 42
power spectrum, 77
prefix property, 43
preorder, 42
probability p, 68
product
  convolution, 75
  pointwise, 75
products on $\mathbb{C}^n$, 75
pseudo-random numbers, 66

queue, 38

random integers, 67
random number, 65
random reals, 67
random sample, 69
random sequence, 65
Rayleigh's Theorem, 77
re-entrant, 4
recursion, 3
root, 41
rotation, 74
runningsample, 69

sample
  conveyor belt, 69
seed, 65
sentence, 52
sharpening, 82
shuffle, 68
signal, 79
smoothing filter, 80
sort
  insertion, 19
sorting S_3, 10, 11
spectral test, 66
stack, 37
Strassen's method, 73
string, 51
  vibrating, 74

teacher
  redundant, ix
timings
  sort, 29
tree, 41
Tukey, 78

uniformly distributed, 65

Von Neumann, 66