Faculty of Environment and Technology

Principles of Computing

UFCFA3-30-1

Simon Langley and Rong Yang

UWE BRISTOL
University of the West of England

bettertogether

1 What is Computer Science?

Computer Science starts from our desire to solve new problems by computer. We want to understand how something can be computed. This is the first pointer to what computer scientists do. You are likely to find them working in 'leading edge' organisations, tackling new problems which nobody really knows how to solve yet. Areas in which computer science is needed include artificial intelligence, cryptography, graphics, computer animation, virtual reality, and safety critical systems (such as fly-by-wire aircraft or power station control systems). You are less likely to find computer scientists working in a traditional business setting; though designing commercial database system isn't easy, we do know how to do it.

Solving a problem means writing an algorithm, that is a set of instructions which can be translated into a programming language (such as JAVA and C) and run on a computer (such as a PC). To write an algorithm we need to understand exactly how to solve the problem. Here computer science can help us analyse what has to be done. As the module progresses you will be introduced to various ideas developed to enable problems to be modelled and solved.

Once you have found a solution you have to convert it into a suitable programming language. JAVA and C, for example, are both good general purpose languages. If an algorithm can be implemented in any language it can be written in Java or C. But for some situations other languages which look radically different (for example PROLOG, a logic programming language) allow algorithm to be written and tested quickly.

You might think the computer scientist's work is done once a program has been written and tested (how to test programs, or better still, prove that they work is itself an important part of computer science - unfortunately we don't have time to pursue it here). But a working program may not be an efficient program. For example it is quite easy to write a program to crack the codes used by military for their most secret communications - easy to write but it will take billions of years for the program to decipher each message. An algorithm may solve a problem but be so slow that no one would use it in practice. We will look briefly at *complexity theory* which provides some ways for deciding how efficient a program is.

Everything we have said so far assumes we can always find a solution. Sometimes we can't. You probably wouldn't be surprised that some problems are so hard that nobody has solved them yet. For example we don't have computers that can communicate with us in natural languages (those 'talking phones' don't really count). Perhaps one day we will achieve this, but this is not the sort of problem we are thinking about. One of the successes of computer science has been to show that some problem are not just hard, they are unsolvable by any computer imaginable. It is not that we don't know enough, or that computers aren't fast enough, some problems just don't have a computable solution. Interestingly the discovery of the existence of insolvable problems (by Alan Turing) happened before anyone had actually built a real computer. Even before the computer as we know it existed, we already knew what it couldn't do!

2 This Module

The aim of this module is for you to look at the sort of problems that computer science addresses and learn some of its standard tools. Of course there isn't a strict divide between computer science and the rest of computing. All our students need to be able to write programs, design systems, do a bit of maths and understand something about computer hardware.

You will find topics coming up in this module which are mentioned in other modules. Hopefully doing this module will help with them. More importantly perhaps the topics covered here will be used in modules next year and in your final year.

In a way of studying computer theory as part of degree is a bit like a doctor learning anatomy. Understanding how the body works isn't medicine but without is doctors would mostly be guessing and hoping for the best a lot of the time. The ideas you learn on this course underpin what you learn in your other computing modules.

- Why computers are built the way they are
- Why it doesn't really matter what sort of computer you use,
- Why programming languages look the way they do
- Why some programs you might wish you had don't exist

All the problems have an answer which depends what computer scientists have discovered.

3 The Course Book

The following is our recommended text book which covers most of the material in this semester and covers maths materials (last semester) as well.

James Hein, **Discrete Structure, Logic and Computability**, Jones & Bartlett Publishers

It isn't cheap (about £28 at the moment, 2013). But it does cover all concepts and basic maths which you need. It will be a good reference book which gives you a solid background for some of the things you will be doing in the future.

There are lots of books covering some or all the topics on this module in the library. A good starting point if you want an overview of the subject is:

D Harel **Algorithmics** Addison Wesley

You may also find books with words like *computability, theory of computing, automata, formal languages* in their titles useful. Some of the computer science books in the library have hidden themselves on the Mathematics shelve (!)

Two books which contain lots of short articles (mostly taken from 'Scientific American') on many aspects of computer science are:

A K Dewdney	**The Turing Omnibus**	Computer Science Press
A K Dewdney	**The New Turing Omnibus**	Computer Science Press

Richard Feynman was a Nobel Prize winning physicist; he also wrote an interesting introduction to what computing is all about:

R Feynman	**Feynman Lectures on Computation**	Addison Wesley

4 Assessment

Students will be assessed by a mixture of coursework and exam.

Coursework – In this semester, similar to the last semester, there will be some short exercises to do. They are really only to check that how you are getting on. The exercises will be done on paper (not on computer). Even for programming tasks, you don't need to run your program on computer. The coursework in this semester will be worth 25% of the module assessment.

Examination – The examination will be a standard 'closed book' exam. There are will be some short answer questions and some multiple choice questions.

5 In-class Tests

Students will be asked to take an in-class test at the end of each lecture (and workshop).

This is not assessed work. It only consists of 5 multiple choice questions which hopefully can help you to assess yourself. The lecturer will also use the test results to check how well contents be delivered. Moreover, the re turned test sheets will be used to track students' attendance.

6 Schedule

Same as the last semester, each student will have two lectures and one tutorial per week. The lectures will be delivered to the cohort as a whole whilst the tutorials consist of smaller classes.

The following table shows what will be covered with approximate timings. We may adjust the schedule if necessary.

Weeks	Subject	Ref. to this booklet	Ref. to Hein's book	Memo students' work
26	Introduction Strings & Languages	pp 5-17	Chapter 1.2 Chapter 3.2	A small task on strings (sets)
27-28	Finite Automata	pp 18-28	Chapter 11	A small task on Finite Automata
29	Pushdown Automata	pp 29 -33	Chapter 12.2	A small task on Pushdown Automata
30-31	Turing Machines	pp 58-66	Chapter 13 Chapter 14.1	A small task on Turing Machine
32-33	Inductions and Recursions	pp 41-47	Chapter 3.1	Some programming tasks
34	Data Structures and Some Basic Algorithms	pp 71-75	Chapter 10.3	
35	Complexity	pp 67-78	Chapter 5	
36	Language Grammars	pp 48-57	Chapter 3.2	
37	Revision			

1 Computation

Computer science might be better called 'Computation Science'. Computation may suggest numbers and maths but that is not the intention. A computation is a process which starts from some input and produces some output. The input could be cheques and payments, the output a bank statement, or the input could be keys pressed on keyboard and the output could be music. A person hearing their name called and responding is receiving input and producing output (turning their head towards the sound). All of these count as computation. A computer, as far as we are concerned, is anything which does computations. It is only in the last 50 or so years that the word computer came to mean a grey box full of electronics. When Alan Turing did his pioneering work on computing a 'computer' was someone who did computations of some sort, not a machine but a person.

While this view of computation is interesting from a philosophical point of view it isn't terribly useful in practice. It's hard to think of any process which isn't a computation and indeed 'computation' has become more or less a synonym for 'process'. The situation is like one described in one of Lewis Carroll's less well known books where two Professors produce a map with a scale of 1 inch to 1 inch! The map is perfect except that it's useless - even apart from the problem of where to put it when it's unfolded. A better map needs less information than the area it describes - but equally a totally blank sheet isn't any use. The problem is to decide exactly what level of detail is required. The same is true for our definition of computing: turning input into output covers pretty well everything and we can't expect every process in the universe to have significant common features.

The first constraint we'll place on computation is that we are only interested in *information* processing. A cow converts grass to milk but we won't count that as computation. We assume that some sort of data is always the input, the computation manipulates that data and produces information (the output).

We will ignore the physical details of input and output. A computer keyboard could be replaced by some other input device (a touch screen for example) without materially affecting the computation as long as the same data is transmitted. A scanner can 'read' graphical information but what actually goes into the computer is a stream of **1**s and **0**s - if we had the patience to enter the data via some other input device the computer would be equally happy.

A similar argument suggests that our model shouldn't bother with which physical output device is being used. We will treat the computation as producing the information which is then interpreted by some output device. For example music output from a computer could be fed into a synthesiser to produce sound or to device which prints music - we assume the real computational work has been done by the time the output is ready.

The important point is to be precise about what we mean by data and what sorts of manipulations to allow.

2 Data & Data Manipulation

These notes were written on a word processor. The inputs to the system are caused by the key depressions by me and the output is a file stored on disc (plus, while I'm typing, output to the screen of what is currently in the document). Various sorts of data have to be stored by the program to make this all possible. The word processor needs to know about arithmetic (to number sections and count words for example) and characters but it also has to represent information about which font is being used, which colour text should be in, where paragraphs begin and end and so on.

Consider another piece of software with quite a different purpose: a route finder which allows you to enter details of two places and finds the distance between them. Here the data is a map of the country. How is a map to be represented?

There are two aspects to data representation: what should the most basic 'units' of data be and how can they be combined to represent more complex data. The second question is covered in other modules. The standard computer answer to the first question is that everything is made up of bits, 1s and 0s. For human beings the answer might be rather different. I do most of my 'computations' with numbers, words and pictures. Of course how the data as held in my brain may be different again.

We will not insist that computation involve any particular basic units of data. Following Alan Turing we will just say that our computations will be based on a *finite discrete alphabet* of symbols. By this I mean no more that that we will start by choosing some arbitrary set of symbols, such a 0 and 1 which must be finite in number and all different. Everything else will be built up from these symbols.

Here is an alternative alphabet: ♋ ♌ ♓ ♎ ♏ ♐ ♑ ■ ♓ ❖ ♦ ⌘. From a theoretical point of view it's no better or worse than binary - the way the symbols are represented on paper is different and there are more of them (perhaps we need more for some computations).

Once we have chosen an alphabet we can represent information as strings of symbols, for example, 0000101110101. Such strings are sometimes called *words* even though might not look like what we would usually call words. Here are some words using different alphabets:

a) -- --- .-. ... . -.-. --- -.. .

b) ♦♓■♎♓■♑♦

c) 01000001010100110100011101001001010010 01

Q1 The first word was, of course, in Morse Code. What do you think it says. How many basic symbols does it use? (Hint: is '-----.-.....-.-.----...' the same?) Could you get by with less symbols? (You sometimes see Morse code used in Westerns or old war movies with a 'clicker' device to generate the signals - how many 'symbols' do they use?) How do you suppose Samuel Morse decided which code to use for which letter?

Q2 01000001 01010011 01000011 01001001 01001001 is a representation of five letters each letter being represented by 8 bits (1s or 0s). What code is this (look in a book on computer hardware)? If only 6 bits were used for a character how many different characters could be represented?

Q3 Here is a longer message using a variant of the code above:

11000001 11010011 11000111 01001001 01001001 00100000 11110111 11101001
11110100 01101000 00100000 11010000 11000001 01010010 01001001 01010100

11011001

The first bit of some characters has been set to 1 according to a simple rule - what is the rule? Does this version have any advantage over the other?

Q4 The Morse code uses different numbers of basic symbols for different letters, the binary code example above always uses the same number. What do you think are the advantages/disadvantages of the two methods.

Q5 Could we use only one symbol in our alphabet. For example, if we wanted to represent numbers we could use * for 0, ** for 1, *** for 2 and in general n+1 *s for the number n. Isn't this sufficient?

Q6 Does binary have any advantage over other coding systems?

The most common examples of using codes to represent data stored in a computer are ASCII (for storing textual information with every character being represented by eight bits) and binary numbers used when we need to do arithmetic. You will come across both of them in other modules, if you don't know them already, read appendices C to these notes for more information about binary.

Q7 How is 23 represented in a) ASCII and b) as a binary number?

Q8 Given a number in ASCII how can you convert it to binary number format? (Assume that you know how to multiply and add binary numbers and that the individual ASCII characters can be treated as binary numbers).

Q9 And how do you do the reverse? (Assume you can divide numbers to get a quotient and a remainder, for example you can 'divide' 23 by 7 to get 3 remainder 2). Surprisingly this is harder than going the other way.

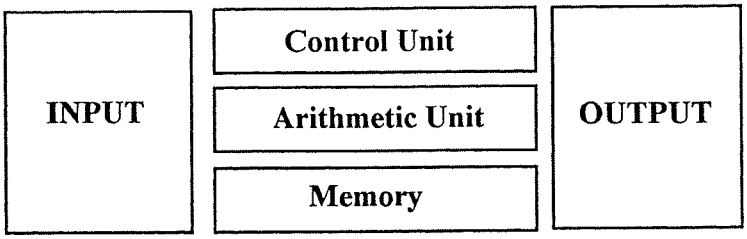
The idea of coding is central to 'pure' computer science (i.e. where it falls off the edge of the real world into logic and/or meta-mathematics). An essential feature of this is being able to show that given some large collection of 'things' it is possible to allocate a unique numerical code to each unique object.

As an example imagine all the valid Java program that have been, or could ever be, written. Can we define a numbering system so that every program could be uniquely classified in the following sense: if two programs produce different output for some input they must have a different number. The answer is extremely simple: assume the programs are stored on disc in ASCII format. Each program can be thought of a long string of **1**s and **0**s. These strings can also be thought of as (big) binary numbers - i.e. a number for each program. If two programs produce different output for some input they must be different at some point and so the 'numbers' corresponding the program are different. Of course two programs may be the same (in terms of the output for a given input) but have different numbers. Also some numbers will never be used (ones whose binary form, converted into ASCII, isn't a valid program). These were not requirements of the problem however.

Q10 Using the same argument show that all models of IBM PCs (including those not yet invented) computers can be given a unique number. (Hint: Assume that for computer A to be a different model from computer B they must have at least one different component, such as different amounts of memory, different screen sizes etc.)

3 The Computation

Having decided what counts as input and output and data, what sorts of process will count as a computation? We don't want to pre-judge the issue by taking too narrow a view. For example we could restrict ourselves to treating as computations the sorts of things that actual, 'real-life' computers do - but you may have heard of neural networks which are very different from PCs but are also computers (in some sense). Also computer scientists are working on all sorts of other radically different design of computer which might do very different things. Nevertheless we will start with a very crude model of a conventional computer which appears in many introductions to computer hardware.



The three blocks in the middle are the important ones.

3.1 Memory

Memory is where information is stored. In thinking about computations we will normally assume that data is placed in memory before computing begins and results are left in memory at the end. Data stored in memory must be organised in some way. For example we can think of memory as containing a pattern of bits such as:

10100010000010001000100101000010100101 .....

in this case we'll assume that it possible to get at particular bits, e.g. the 7th bit or the 23rd.

Modern computers are based on binary values but there is no absolute reason why they have to be. We could equally well assume memory consists of a long sequence of characters like

aiuhgdcewuyfvweuytruyoiwjdpls'dlwrjewrhuewgqj27432764^&^!^£"!£"

or symbols from any other (finite) set. We can even have a memory with some values of one sort and some of another. For example in JAVA we can think of memory as a mixture of numbers (in JAVA terms **int** or **float** type values), characters (**char** type) or sequences of characters (**Strings**). And rather than talking about the 10th **int** in memory or the 1872653rd **char** we will give them names like A and B.

3.2 The Control Unit

The *control unit* determines which actions are carried out by the computer and in what order. It's basic operation is to say: "If event X has happened then do Y". An event might be something external to the computer (a user pressing a key) or internal (a particular bit having a particular value, e.g. "if bit 23 is set to 0 then ....".

We put only one condition on the control unit at the moment - it must be *deterministic*, that is we don't allow control operations like "if bit 23 is set to 0 then maybe do X and maybe do Y".

3.3 The Arithmetic Unit

The *arithmetic unit* is the mechanism which manipulates data. This includes doing arithmetic and comparing values. Arithmetic is probably the wrong word - 'data manipulation unit' would be better but it's too late to change the name now. If we assume data consists of bits, the typical operation of the arithmetic unit is to get some bits from memory manipulate them in some way and put a result of some sort back into memory.

We won't spelt out in detail exactly what operations are allowed in the arithmetic unit. We'll say only that an operation must be something which can be done mechanically without the need for intelligence. Since memory consists of symbols we can identify in general terms what sorts of operation might be allowed.

The simplest imaginable operation is to put some value at a particular point in memory. For example, assuming memory consists of bits, set the 10^{th} bit in memory to 0 (or 1). The next simplest operation is to take a bit from one place, modify in some way and put the resulting value in a different place in memory. The next sort of operation would be to take two bits, combine them in some way and produce a result.

As with the control operations we assume the arithmetic unit is deterministic. Each operation must always produce the same result when given the same starting data.

4 Types Of Computer

Normal' computers have a single arithmetic and control unit combined onto a single silicon chip called the *central processing unit*, but there are other possibilities.

A neural network is a sort of computer comprising lots of nodes each of which receives input from other nodes. Each input is a single number. The node does some arithmetic on each input ('arithmetic unit') and may take into account what inputs it has received in the past ('memory'); if appropriate conditions are met ('control unit') it sends a signal to other nodes. In this case there are lots of little processing units spread across the network as a whole.

Alternatively in machines designed for high speed arithmetic calculations (such as in weather forecasting or atomic research) there may be one control unit and lots of arithmetic units each doing a bit of the calculation in parallel.

Some, or usually all, of these three components turn up in every computer yet thought of. We will make this the basis of our definition of a computation:

A computation is the manipulation of data in a controlled sequence of operations.

Q1 Is a human being a computer as far as this definition is concerned? What about the latest, most sophisticated pocket calculator? Or a 'programmable' washing machine or video?

Q2 A computer program which tells you the time will obviously give different results at different times. Does this mean that it is non-deterministic?

Q3 Would you count this as a computation according to our rules?

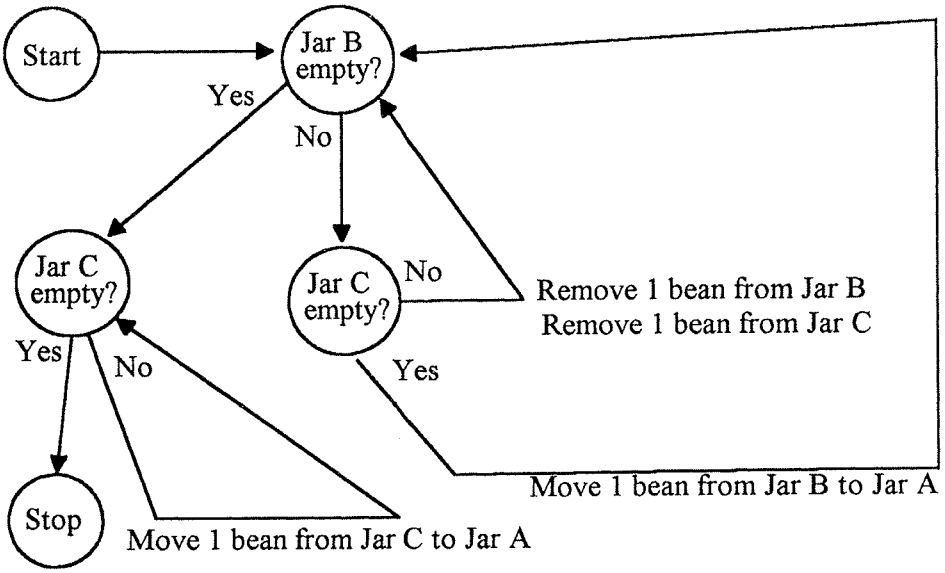
To calculate the value of π (i.e. 3.14159 ...) approximately. Draw a circle inside a square so the circle just fits. Suppose the side of the square is 1 metre, the radius of the circle is 1/2 metres so its area is $\pi r^2 = \pi / 4$. Now start throwing darts at random in the direction of the square until 1000 have hit it. Suppose n darts are inside the circle. The ratio of darts within

the circle to the total number thrown ought to be the same as the ration of the area of the circle to that of the square so we have $\pi / 4 = n /1000$ or $\pi = n / 250$.

What memory, arithmetic and control is involved?

Computer hardware courses deal with the operations performed by actual computers. In this module we don't want to be too restricted by what current computers do. The following questions provide examples of possible 'basic' operations that a computer might have.

Q4 You are provided with three jars labelled A, B and C. Initially jar A is empty and jars B and C each contain some number of beans. What does the following 'computation' do?



What arithmetic operations are available? What events cause the control unit to do things? What determines the sequence of operations?

Q5 Here is the same computation in a different form. Which do you prefer? Why?

1 If Jar B is empty go to line 6
2 If Jar C is empty go to line 9
3 Remove 1 bean from Jar B
4 Remove 1 bean from Jar C
5 Go to line 1
6 If Jar C is empty go to line 11
7 Move 1 bean from C to A
8 Go to line 6
9 Move 1 bean from B to A
10 Go to line 1
11 Stop

Given a computation represented in this form, is it always possible to convert it to the previous form? What about *vice versa*?

Q6 This is the same problem, yet again, written in a different language:

1. empty(B), not empty(C) → move 1 bean from C to A
2. not empty(B), empty(C) → move 1 bean from B to A
3. not empty(B), not empty(C) → remove 1 bean from B, remove 1 bean from C

Here the control unit chooses one of the statements at random. If the conditions are true it carries out the operation on the right hand side otherwise it just tries again. This includes a random element - does it therefore contradict our requirement that a computation be deterministic?

Q7 The 'bean-jar' idea used above is a simple model of computation. Using the version in Q15 we have a programming language in which the only allowed steps are:

If Jar x is empty go to line n

Remove 1 bean from Jar x

Move 1 bean from Jar x to y

Go to line n

Stop

We can drop the references to beans and jar and treat this as describing doing arithmetic on named *memory locations*. The rules are then:

If $x = 0$ go to line n

Subtract 1 from x

Add 1 to x

Go to line n

Stop

("Move 1 bean from Jar x to y" has to be replaced by "Subtract 1 from x; Add 1 to y")

Assuming that you can invent other memory locations as necessary, write 'mini-programs' to:

1 set the contents of any location to 0;

2 copy x to y (i.e. at the end x should be unchanged)

3 add x to y leaving x unchanged

Without actually try to write the program show that multiplication, integer division and finding remainders can be done with a bean-jar computer.

Q8 The following describes a 'computation'. Which parts of the description relate to which parts of the basic model of a computation.

There is an infinite square grid. Each square of the grid can be 'on' or 'off'. Initially some cells are 'on'.

Each square has eight neighbours (the grid squares N, NE, E, SE, S, SW, W and NW of it).

On each tick of a clock the following activities take place:

- a cell with exactly two neighbours which are 'on' will be unchanged after the next clock tick (either 'on' or 'off');
- a cell with three 'on' neighbours will be turned 'on' after the next clock tick (whatever its current state);
- all other cells will be turned off after the next clock tick.

This model, which is often called the Life Game, is very simple. Also it doesn't seem to do very much except draw patterns on a screen. Surprisingly perhaps whole books have been written on it, a good example being: "The Recursive Universe" by William Poundstone (pub: Oxford University Press). If you set up the initial pattern correctly it has been shown that the 'game' can carry out any computation a computer can.

5 Languages

We have decided to ignore the details of *how* data is input to a computer but the *format* or layout of input and stored data is important. Suppose we want a program, which we will call ADD, to add two numbers. The input will be characters representing the two numbers separated by a space: "23 47" perhaps. In order for any computer to be able to add these numbers it must be able to recognise that the characters '2' followed by '3' represents the number 23, then skip over the space then recognise 47. Should the program also recognise "XXIII XLVII" as a valid sum? What about if there are several spaces between the numbers and not one? Are negative numbers allowed? Can numbers be of any size at all? What about decimal points? And so on. To process any data you must understand its structure.

The layout of the data in the last example was very simple. A JAVA compiler has a much more difficult task: the possible inputs can be any JAVA program of any complexity. This is clearly much more harder than recognising two numbers. A program which reads and understands English text has an even greater problem because the complexity of natural languages over computer ones.

We will keep to the simple example of adding two numbers which are assumed to be positive integers (whole numbers) of any size separated by one or more spaces. Possible inputs to ADD are:

```
123 444
1234     3728
111  2
0  0
```

There are an infinite number of possible inputs. We will call any sequence of characters a *string*[1] (in general any sequence of symbols from some given set of allowed values would be a string so you could have strings of bits for example). The set of all strings a particular program accepts is called the input *language*. So the language of ADD is the strings "123 444", "1234 3728", "111 2", "0 0", ... and all the other possibilities. The language accepted by a JAVA compiler is any JAVA program, i.e. a string of characters which conform to the rules of the JAVA language.

Since the number of different inputs accepted by most programs is very large we can't expect to describe a language by listing the strings which comprise it. Indeed for many programs the number of legal strings is infinite so they can't be listed. What we can often do is describe the structure of a language in such a way that we can decide whether any given string belongs to it. As an example, consider the English language. In general we can decide if a particular string of letters makes a valid sentence. We do this not because we know every legal English sentence in advance but because all English sentences follow certain rules of grammar (also called the *syntax* of the language). Not for is sentence example this a! In the case of English there are many rules and many exceptions, for the programs we shall be interested in the languages will be much simpler.

Recognising valid input is only part of what a program does. The addition program has also to add the two numbers together, but interestingly it is possible to argue that in theory computation is only about recognising whether a string belongs to a particular language. Imagine a slightly different version of ADD called ADDTEST. The input to ADDTEST is three numbers and the output is "YES" if the sum of the first two numbers is equal to the third and "NO" otherwise. For example: "1 2 3" produces "YES" because 1+2 = 3. Consider all the strings which consist of three numbers with the first two adding up to the third. The language includes "1 1 2", "1 4 5" and infinitely many other strings. We will call this language ADDLANG. What is ADDTEST doing? It is deciding if a given

[1] Hein pp36-39

input string belongs to ADDLANG. Now this isn't the same as adding up the numbers, but if ADDTEST exists we can definitely produce ADD (in theory at least). Given a pair of numbers, 23 and 47 say, we can execute ADDTEST with inputs "23 47 0", "23 47 1", "23 47 2" etc. until we reach "23 47 70" at which point we get the answer YES - we now know the sum of 23 and 47!

Nobody would seriously suggest doing addition in this way (even if we could think of a way of writing ADDTEST without doing addition), nevertheless we have shown, in this case at least, that doing a calculation can be reduced to recognising whether a string belongs to a particular language.

In the rest of this chapter we will make more precise some of the ideas introduced above and illustrate one way of describing languages.

5.1 Strings And Alphabets

We assume that the characters in the string are taken from some fixed *alphabet* of symbols. Usually the alphabet will be the set of ASCII characters or a subset of it.

The alphabet is often given as a set. For example if the alphabet is {a, b} possible strings are "aa", "ab", "aaab" and "bba" but not "abc" or "xyz". The alphabet could be Arabic digits, {0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }, Roman digits {I, V, X, L, C, D, M }, or just shapes {  ,  ,  ,  }.

Following convention we won't show the quote marks around strings and will write *Hello World* instead of "Hello World". This leads to two complications. The first is that "" is a perfectly valid string albeit one with no characters in it, how do we write it without quotation marks? The answer usually adopted is to use a special symbol for it. We will use Λ, the Greek capital L. You can type it in Windows by using L and setting the font to Symbol. So Λ is a perfectly legal string - you just can't see it on the page normally.

The second problem is that we sometimes want to use a name like x to stand for a string, i.e. maybe x is the string "abc", but if we drop the quote marks how do we know that x doesn't just mean the string "x". The sensible answer to this would be to keep the quote marks (which would also solve the first problem without the need for Greek letters) but nobody does. Hein's solution is to mostly use u, v, w, x, y, z as the names of strings and in examples to use the earlier letters for the alphabet of actual characters. I'll follow Hein's example so unless otherwise specified letters u through z stand for strings.

Only a few notations are needed to describe operations on strings. First each string has a *length* (the number of characters in it). If u is a string we write |u| for its length, so |abc| = 3. |Λ| = 0.

If u and w are two strings then uw is their *concatenation* i.e. the result of joining them together. This is so trivial it hardly needs an example, but if u = abcd and w = efg then uw = abcdefg. Hein sometimes writes this as u.w or cat(u,w).

We can concatenate a string with itself and it's often useful to have a shorthand for this. We will write $u^2 = uu$, $u^3 = uuu$ and so on. To be consistent $u^1 = u$. What about u^0? Since this stands for no copies of u - and hence no characters at all - it is sensible to say $u^0 = \Lambda$.

Finally if w is any string we will write w^R for the string obtained by reversing the characters in w. I.e. $(abc)^R$ is cba. We've also introduced (and) into our strings with, we hope the obvious meaning of grouping together characters. This can cause more confusion as sometimes we want (and) to be characters in our strings and sometimes want them just to group characters. Hopefully which is which will be clear from the context.

5.2 Languages[2]

Having explained strings at great length the definition of a language is very simple: a *language* is a set of strings. The strings making up a sentence are sometimes called the *sentences* of the language and sometimes the *words*. This is confusing so we'll usually just call them strings. Here are some examples:

a) { the, cat, sat, on, mat }

This is a very simple language. The *sentences* of the language are: "the", "cat", "sat", "on" and "mat". "the cat sat on the mat" is not a legal sentence of the language. (Hein uses the term *well formed formula* or *wff* rather than sentence) When we define a language as a set each element of the set is a legal sentence, nothing else is. The alphabet for this language is {a, c, e, h, m, n, o, s, t }.

b) { the quick brown fox, the cat sat on the mat, the lazy black dog }

This language has only three sentences in it. The alphabet for the language includes spaces.

c) { a, aa, aaa, aaaa, ... }

The previous two languages are *finite* that is they contain only a finite number of different sentences, this one is *infinite*, it consists of all strings of one or more a's.

d) { }

This is the empty language often written ∅ (the empty set). It contains no strings. It is convenient to introduce this for completeness but it doesn't have any practical use.

e) { Λ }

This is not the same language as the previous one. It does contain a string while ∅ doesn't so they are different - again it's not a very useful language.

f) Java

Anything which a Java compiler can compile is a sentence of Java. Java is an infinite language but what counts as a legal Java sentence is harder to define (but it has been defined because Java compilers exist). Later we will look at how languages like Java are defined.

g) English

This is even more complicated. There is no definitive set of rules for what makes an English sentence legal.

Describing a language as { a, aa, aaa, aaaa, ... } assumes that you can correctly guess what is meant. But guesses are often wrong. In this module you will encounter a number of ways of describing languages which allow precise definitions to be made, the first is to use a notation from set theory.

---

[2] Hein pp126-130

We will write $\{ a^n \mid n > 0 \}$ to mean the set of strings of the form a^n where n can take any value greater than zero. I.e. $a^1 = a$ is in the set, so is $a^2 = aa$, and so on. This is more concise than writing { a, aa, aaa, aaaa, ... } as well as not relying on guesses.

A closely related language is { Λ, a, aa, aaa, aaaa, ... } which is the same but includes the empty string as well. This is $\{ a^n \mid n \geq 0 \}$.

Q1 Write down the first few strings of the following languages. How would you describe them in English?

a) $\{ a^{2n} \mid n \geq 0 \}$ b) $\{ a^{2n+1} \mid n \geq 0 \}$

c) $\{ a^n b^n \mid n \geq 0 \}$ d) $\{ (ab)^n \mid n \geq 0 \}$

e) $\{ a^l b^m c^n \mid l+m+n = 6 \}$ f) $\{ a^n b^m \mid n \geq 0$ and $m = n/2$ (n even) or $(n-1)/2$ (n odd)$\}$

Q2 Using the same notation how would you describe the following languages:

a) { aaa, aaaaaa, aaaaaaaaa, ... } i.e. all string a multiple of 3 characters long

b) { Λ, abbc, aabbbbcc, aaabbbbbbcccc, ... } i.e. twice as many b's as a's or c's.

c) { b, bb, bbb, abb, abbb, aabbb... } i.e. more b's than a's

Q3 A name in some programming language must satisfy the following rules: the first character must be a letter, after which can be up to 30 other letters or digits and at the end there may (but need not) be one of the characters % or $.

Suppose L is the set of letters, D the set of digits and S = { %, $ }, describe the language of names.

5.3 Building Languages

One way of defining languages is in terms of simpler languages. Suppose A = {the, cat, sat, on, mat} then we can define some new languages:

$\{ w^R \mid w \in A \} = \{$ eht, tac, tas, no, tam$\} = A^R$

Where we write A^R for the language obtained by reversing every word in A. Other languages can be made by extracting some strings only from a given base language. For example:

$\{ w \mid w \in A, |w| < 3 \} = \{$ on $\}$

But one of the most common ways of making a new language is to combine pairs of strings:

$\{ ww^R \mid w \in A \} = \{$ theeht, cattac, sattas, onno, mattam$\}$

$\{ w^2 \mid w \in A \} = \{$thethe, catcat, satsat, onon, matmat$\}$

$\{ uw \mid u, w \in A \} = \{$ thethe, thecat, thesat, theon, themat, catthe, catcat, ... $\}$

If A and B are languages we write A.B (or just AB) for the language obtained by concatenating every string from A with every possible string form B. In set notation:

$A.B = \{ uw \mid u \in A, w \in B \}$

Using two of the examples above:

$\{ ww^R \mid w \in A \} = A.A^R$

$\{ uw \mid u, w \in A \} = A.A$

Q1 Can $\{ w^2 \mid w \in A \}$ be written this way? Explain.

It is natural to write A^2 for A.A, A^3 for A.A.A and so on (if we wish we can write $A^1 = A$). In general we have

$A^n = A.A^{n-1}$

Q2 Suppose A = { the, a }, B = { Λ, big, red, heavy, silly } C = { idea, cat, mat, car, man } which of the following (ignoring spaces) are in $L = A.B^4.C$

a) a big idea

b) the heavy big silly idea

c) the heavy heavy man

d) the silly silly red red red car

e) a car mat

f) a big red heavy silly cat

If Λ were not in B which of the above would still be in L.

Q3 If L = { Λ, abb, b } and M = {bba, ab, a}, evaluate: L.M, M.L, L^2

Q4 Solve each of the following language 'equations'

a) { Λ, a, ab }.L = {b, ab, ba, aba, abb, abba }

b) L.{a, b} = { a, baa, b, bab }

c) {a, aa, ab}.L = {ab, aab, abb, aa, aaa, aba }

d) L.{Λ, a} = {Λ, a, b, ab, ba, aba}

Q5 If A and B are languages with m and n strings respectively does it follow that A.B has mn sentences?

Q6 If L is any language, then $L.\varnothing = \varnothing$, justify this claim.

Q7 What is L.{ Λ } for any language L.

Q8 On the basis of the last question, what is a sensible definition for A^0? (Hint: if d > 1 then $A^d = A.A^{d-1}$, what happens when d is 1?)

Finally, languages are, as we have said, sets, so any of the standard set operations make sense. If A and B are languages then:

$A \cup B = \{ w \mid w \in A \text{ or } w \in B \}$ the union - all strings in either language

$A \cap B = \{ w \mid w \in A \text{ and } w \in B \}$ the intersection - all strings in both languages

$A \setminus B = \{ w \mid w \in A \text{ but } w \notin B \}$ the difference - strings in A but not B

$A' = \{ w \mid w \notin A \}$ the complement - all strings not in A ***using the same alphabet as A***

Q9 If A = { a, b, c, d } and $B = \{ a^n \mid n \geq 0 \}$ what is $A \cap B$?, What is B\A?

Q10 If $A = \{ a^{2n} \mid n \geq 0 \}$ and $B = \{ a^{3n} \mid n \geq 0 \}$ what is $A \cap B$?

Q11 If A = { Λ, a } and B = { Λ, a, b } what is $A^2 \cap B$? What is $A^2 \setminus B$

Q12 If $A = \{a^{3n} \mid n \geq 0 \}$ what is A'?

Q13 What is the complement of $\{ a^n b^n \mid n \geq 1 \}$?

Q14 Using examples show that, in general, $B \setminus A \neq A \setminus B$. Is there any situation where the two are equal?

The last two definitions we shall use are:

$A^* = A^0 \cup A^1 \cup A^2 \ldots$ called the *closure* or *Kleene star* of A

$A^+ = A^1 \cup A^2 \cup A^3 \ldots$

Q15 The definitions of A^* and A^+ looks rather technical, in words how would you define them?

Q16 Write out the first few strings of {a, b, c}*

Q17 $L^* = L^+ \cup \{ \Lambda \}$ does that mean that $L^* \setminus \{ \Lambda \} = L^+$ i.e. if you drop Λ from L^* do you get L^+?

Q18 Would it make sense to define an operation equivalent to forming $A^0 \cap A^1 \cap A^2 \ldots$?

1 A Simple Model Of Computation[1]

The next few weeks look at some general models of computing and how the ideas they employ are used in computing. In each case they take an idealised 'machine' with a very restricted range of instructions and explore what sort of things such a machine is capable of. This first section explores the idea of controlling a machine by changing between internal states i.e. different configurations of electronics or machinery. The first model has no memory and no real instructions other than recognising that certain inputs have been read. Nevertheless there are quite a lot of things it can do and software implementations of such devices are commonly used.

You may wonder what connection this has with the previous weeks. It turns out that there is a very important link which sets the stage for what is to follow. The machines described here, called *finite automata* or FAs can recognise patterns of symbols or certain sequences of events. The patterns they recognise are a special class of languages called regular languages. These are not just of theoretical interest, you will encounter them in at least one other module.

2 Control Units

The concept of a control unit is very important and very general. This chapter looks at a model of computation which is all about control.

You can think of a control unit having a set of rules of the form:

if A happens **then do** X

More generally we think of a machine having a set of possible internal states and having control rules like:

if E happens **when** in state A **then change state to** B **and do** X

Turing thought of people being in a particular 'state of mind' and obeying rules like these. E.g.

if interrupted **when** busy **then change state to** annoyed **and** say something rude.

In what follows we are going to concentrate on events causing state changes not on what the system does about them (the actions taken).

As an example consider a simplified recording device. The buttons on the front of the device are labelled: RESET, PLAY, RECORD and PAUSE. The user can press RESET at any time to cancel whatever is going on, PLAY displays whatever the machine has recorded and RECORD starts it recording data. Pressing PAUSE once will stop either PLAYing or RECORDing, pressing it again restarts them. We will assume that silly actions - like pressing PLAY while in the middle of RECORDING - are ignored by the machine.

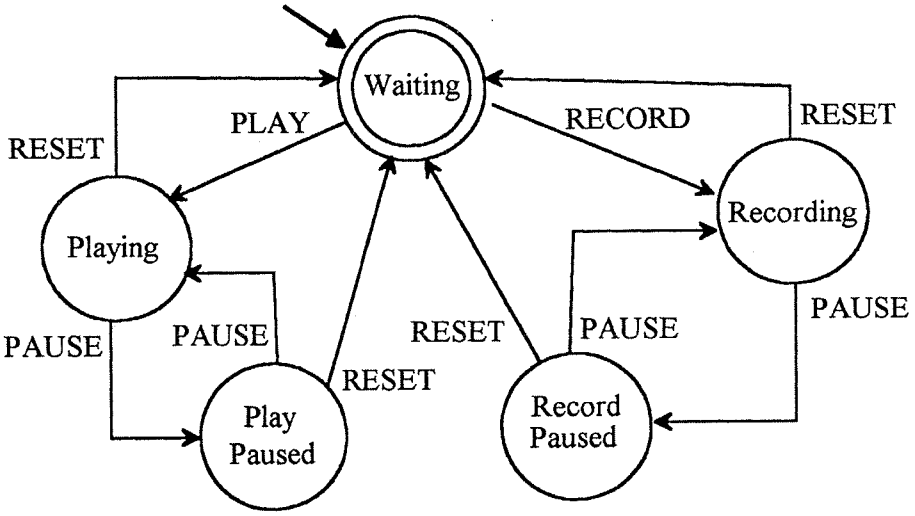
The events are pressing the buttons, the states will be called 'waiting' (doing nothing), recording, playing, and so on. The states and the effects of pressing button are summarised in the table below, called for obvious reasons, a state transition table.

---

[1] Hein pp584-608

Current State	Button Pressed	New State
Waiting	PLAY	Playing
Waiting	RECORD	Recording
Playing	PAUSE	Play Paused
Playing	RESET	Waiting
Play Paused	RESET	Waiting
Play Paused	PAUSE	Playing
Recording	PAUSE	Record Paused
Record Paused	PAUSE	Recording
Record Paused	RESET	Waiting

Not every possible combination of state/event has been shown. Assume that, for instance, pressing RECORD while the device is playing is physically impossible. Another way of representing control rules is the state transition diagram. Here is an example.



When describing state transitions a mathematical notation is sometimes used. We can replace our diagram by a *function* (often given the name δ - a Greek 'd') and write: δ(Playing, RESET) = Waiting and so on. (This is really no different from the table form above of course.) A device like this is called a *Finite Automata* or *FA*. Another name is *Finite State Machine* or *FSM*.

In the diagram the circles are *states*, the arrow shows the *start state* - where the machine is when it is turned on initially. The double circle is a *halt state*, a point at which the machine has done what has been asked of it (i.e. we assume the machine is never left permanently either playing, recording or in one of the paused states). Note that it can leave the halt state - arriving there doesn't stop the FA - it just has to be there when it does stop to be successful. An FA has only one start state but may have several halt states.

The lines between states (called *transitions*) are each labelled with the button which was pushed. In general these could be represent any symbols the machine accepts. A controller for a 'real' recorder would have to take account of some actions we have ignored: pressing RECORD while playing for example. These could just be transitions from a state back to itself - i.e. doing nothing. They should

be included if needed. If a unexpected event occurs (i.e. there is no transition labelled with that state) the FA just 'dies'.

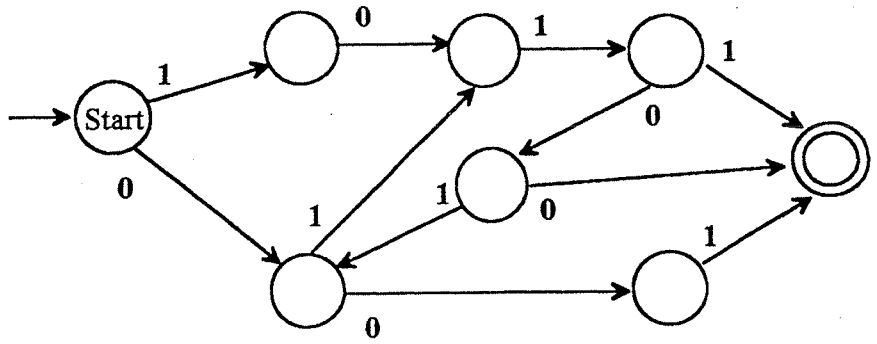
The operation of an FA is very simple. It is initially in its start state, when a symbol is input it moves to the state 'along' the transition labelled by that symbol and so on until symbols stop coming. If when all the symbols have arrived the FA is it a halt state we say it has *accepted* the string. The set of all strings that an FA accepts is called the language accepted by the FA or equivalently the language it *recognises*.

Various things can go wrong. First the string might end before a halt state is reached, in which case it has not accepted the string. Alternatively a symbol might arrive which does not correspond to any transition out of the current state, again the machine does not accept the string (imagine it producing an error message and stopping). A less obvious problem is that there might be two transitions from the current state with the same symbol. If there are, the FA is called *non-deterministic* (an *NFA*) otherwise it is a *deterministic* FA (DFA). For the moment we will consider only DFAs, but later we will show that NFAs are really no different.

This gives us all we need for a definition of a DFA. We have: a set of transitions, a specified start state, one or more halt states and a rule that from any state there can never be two transitions labelled with the same symbol.

To save having to draw lots of pictures, we can describe a FA in words by listing the start state, halt states and all transitions as we did for the recording machine. In examples I'll sometimes write **N x M** to mean input **x** causes a transition from state **N** to state **M**.

Q1 Which of the following strings is accepted by the automaton.



a) 10100
b) 00
c) 1101001
d) 001
e) 0110
f) 0110101
g) 1010111011100
h) 011011101

Q2 Draw a DFA which will accept any string of 1s and 0s until it encounters two consecutive 1s

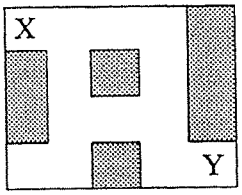
Q3 Devise a DFA which accepts any string of 1s and 0s. It should have two accept states ODD and EVEN, the one it ends in should be determined by whether the number of 1s is ODD or EVEN.

Q4 Devise a DFA which accepts any string of a's, b's and c's which contains the sequence abc somewhere within it.

Q5 A programming language includes operations written =, >, >=, <>, <, <=, =>. Draw a DFA which will end in a state called VALID if one of the character sequences above is entered followed by a space character.

Q6 In Visual Basic a name must start with a letter after which may come any number of letters and digits and, optionally, one of % or $ may come at the end. Letting a stands for any letter (a-z) and d for any digit draw an FA which accepts legal names.

Q7 The diagram below shows a maze. A person in the maze can go North, South, East or West a square at a time. Use a DFA to model in some way valid paths through the maze from X to Y.



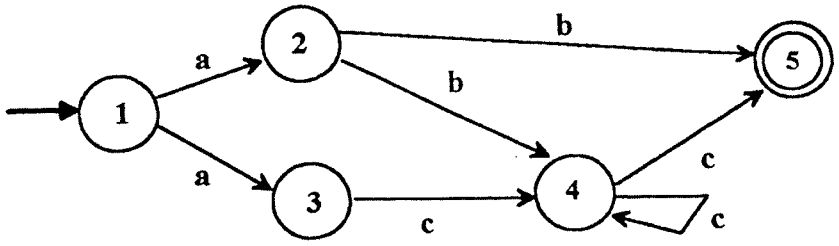
Q8 In a version of PASCAL strings of characters are written in the form 'bbbbbbb' (the quote character is part of the string for this question). A ' can be included in a string by writing it twice (e.g. 'a''b'). There are also special characters which consist of a colon (:) followed by any single character. Design a DFA to accept such strings, including opening and closing quotes, use a symbol **c** to stand for any character other than ' or : and **x** to stand for any character at all.

Q9 In C/C++ strings use a different system for showing special characters. Double quotes enclose strings and special characters are preceded by '\'. Examples of special characters are \" (for a quotation mark), \\ (for \ itself), \n (newline), \t (tab). It is also possible to include any character in a string by specifying its ASCI code as a decimal number, e.g a space has code 32 so a string could be "a\32way\32of\32showing\32spaces\n". Draw an FA which accepts C/C++ strings with these special characters.

Q10 Numbers are sometimes written in scientific notation, e.g. 1.1E3 which means 1.1 x 10^3. Suppose the rules are as follows: the number must start with one or more digits, the decimal part is optional but if present consists of a dot followed by zero or more digits. The E is optional but if present may be followed by + or - and must be followed by one or more digits. Use **d** to stand for any digit and draw a DFA to accept a number in this format.

3 Non-Deterministic FAs

In a DFA there can never be two transitions to different states for the same input symbol. from any state. What happens if we drop that restriction? The result is a non-deterministic FA or NFA (sometimes NDFA). Here is one, what strings does it accept?



We can think of this as a FA with 'a mind of its own'. In state 1, if **a** is input it can decide (maybe by tossing a coin) whether to go to state 2 or 3. Similarly with input of a **b** in state 2, or a **c** in state 4, it makes its own decision.

Surprisingly, for every NFA there is a DFA which accepts the same exactly the same strings. Before we show this we see how an algorithm could be designed to test whether a particular string is accepted by this machine. Suppose the input string is **abcc** which is indeed accepted.

First make a list of the states you might be in: when we start we assume we are in state 1.

The first input is **a** this could take us to state 2 or 3, since we don't know which, we'll remember both.

The next input is a **b** which wouldn't make sense if we were in state 3 so we conclude we were really in state 2. The next state could be either 4 or 5.

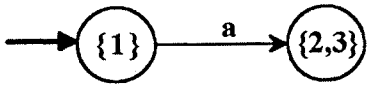
The next input is **c** so we must have been in state 4, the next state could be 4 (again) or 5.

The next input is also **c** we must have been in state 4, the next state could be 4 (again) or 5.

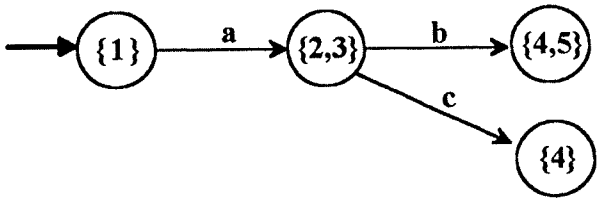
There is no more input, since we could have got to the halt state with this input we will accept the string.

What DFA corresponds to this NFA? We can use the approach followed above to produce one. Where the states of the NFA were labelled 1, ... 5 the DFA will have states labelled {1}, {2,3}, etc corresponding to possible states we might be in on the original machine.

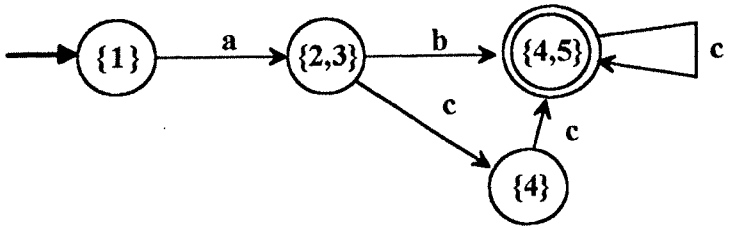
We start with the first state {1} since **a** takes us to either 2 or 3 we draw a single line to a state {2,3}.



There aren't any other transitions starting from 1 so we move on to look at {2,3}. The possible transitions from states 2 and 3 on the original machine are: **2 b 4**, **2 b 5**, **3 c 4**, we add two new states:



We now have two states to deal with: {4,5} and {4}. Taking {4,5} the possible transitions from 4 are **4 c 5** and **4 c 4**, there are no transitions from 5. So there isn't really any difference between being in {4} and being in {4,5} is there? Well 5 is a halt state so {4,5} is also. So here is the last stage.



This is now a DFA. If you experiment with some sample strings you should find it accepts the same strings as the NFA. In this case the DFA actually has less states than the NFA but this is not usually the case.

Q1 Convert from NFA to DFA (in each case 1 is the start state)

a) 1 a 2, 1 a 1, 2 b 2, 2 b 1 halt state 2

b) 1 a 2, 1 a 3, 2 b 2, 2 b 4, 2 b 3, 3 c 3, 3 c 4 halt state 4

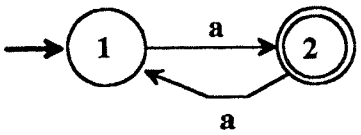
c) 1 a 2, 2 b 2, 2 b 1, 2 b 3 halt state 3

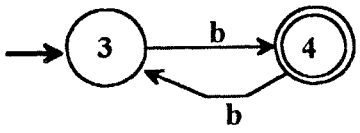
Q2 An old IBM assembler allowed numbers to be written in binary as, for example, **10101B** and in octal (digits 0-7) as, e.g. **0123O** (the last character is O for Octal). Draw an non-deterministic FSM which accepts both binary and octal numbers. Can you convert it to a deterministic version?

Q3 If an NFA has n states what is the maximum number of states the corresponding DFA can have?

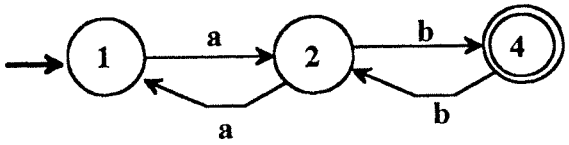
4 Constructing FAs

Consider an automaton which will accept odd number of **a**'s then an odd number of **b**'s. It isn't difficult to invent such a machine but we will do it by a slightly roundabout route. We'll devise a DFA which accepts an odd number of **a**'s and one which accepts an odd number of **b**'s and then join them together. This looks quite easy. Superimpose (with suitable renumbering) the start state of the second machine over the halt state of the first. That's it. You now have a machine which accepts an odd number of each. This is what we've done:

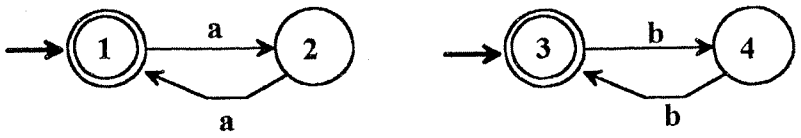




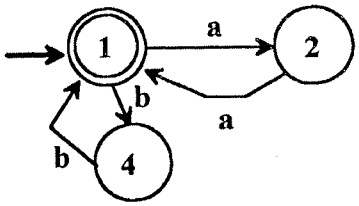
merged into ...



Unfortunately this doesn't work if we want a machine to accept an even numbers of **a**'s then an even number of **b**'s ...

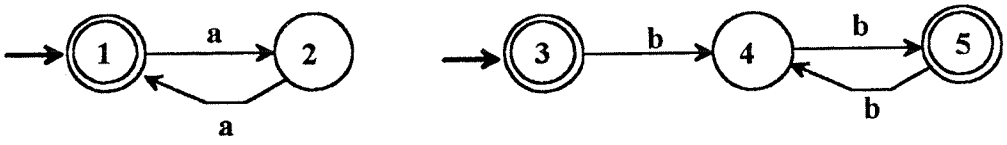


merged into ...

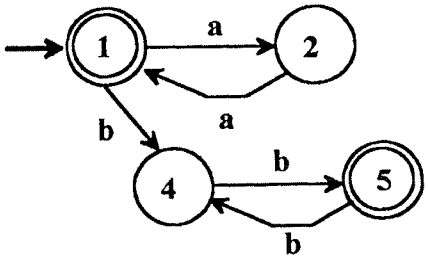


Can you see what is wrong with this? It will accept, for example, **aabbaabb**. The problem is that it is possible to go from the start state of the second machine, through it, back to the start state and then back into the first machine again and continue over and over again. It isn't difficult to stop this happening.

Suppose we use a slightly larger DFA for the **b**'s which none the less accepts the same set of strings. The modification we make is to ensure that you cannot get back to its start state after it's left it. With the new version once the route from state 1 to state 4 has been taken you can never return. The new version has two stop states but that is perfectly legal.



merged into ...



Q1

Q2 Modify the merged machine so that the start state has only outward going transitions. Do the same with the previous machine which accepts odd numbers of **a**'s then **b**'s.

Q3 Find a general set of rules for converting a FA into an equivalent one which has only out going transitions from its start state.

Suppose now we want a machine which accepts first **a**'s then **b**'s with the restriction that the total number of characters must be even. This implies either an even or an odd number of both characters. We already have machines for the two parts, all we have to do is superimpose the start state of one on the start state of the other and we get a bigger machine from whose start state we can go into either

machine. Again, and for the same reasons, both machines should have only out-going transitions from their start states.

Q4 Check that this does indeed work with the example above.

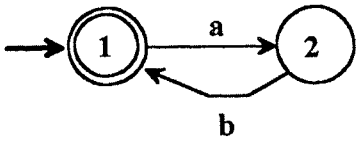
Q5 We can start with two DFAs and end up with an NFA. Does this matter?

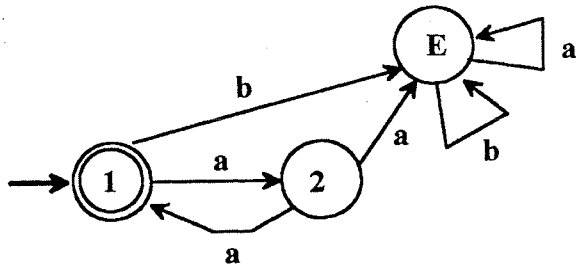
Q6 Find a DFA which accepts language sequences comprising any number of **a**'s, then either any number of **ab**'s or any number of **b**'s by combining machines for each part into an NFA and then turning it into a DFA.

You may have noticed that linking together FAs in these two ways is equivalent to two of the ways that new languages can be built from simpler ones. Joining FAs together 'end-to-end' gives the product of two languages, merging their start states gives the union.

Q7 Show how to produce an FA for the closure of a language, L, given you have an FA for L. Check that it works for simple cases (e.g. apply it to an FA which accepts only **aba** and produce one which accepts $(\mathbf{aba})^n$).

Q8 The two FAs below accept the same language - $\{ (ab)^n \mid n \geq 1 \}$. The first is simpler and is what you would usually draw. The second has an extra state called E (for Error) which deals with 'unexpected' characters. All states will have transitions to E for any characters in the machine's alphabet which don't cause transitions from that state. From E itself any character causes a transition back to E (which is not a halt state). First convince yourself that you can do this for any FA and that the extended machine accepts exactly the same language as the original.





Now explain how, if you have an FA for a language, you can produce an FA for its complement. (Hint: the new FA has to accept any word the original didn't and *vice versa*).

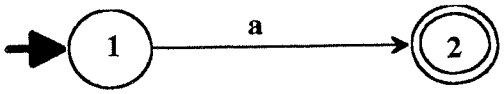
5 Regular Languages

We have seen that FAs can accept various sorts of patterns/languages. Is it possible to say exactly what determines whether a language can be accepted by some FA? The preceding section has given a clue: suppose language L is accepted by some FA, then L* is as well. So is L^+ (which should be obvious), and L'. If L and K are two languages, each accepted by some FA then so are L.K and $L \cup K$. $L \cap K$ is also accepted by an FA using the following argument.

By drawing a Venn diagram you should be able to see that $L \cap K = (L' \cup K')'$ so first take the FAs for L and K and find the *complementary* FAs (accepting the complement of each). Once you have FAs for L' and K' you can find an FA for $L' \cup K'$ (merge start states), finally take the complement of this and you have an FA for $L \cap K$.

We now have a mechanism for building up a description of the languages accepted by FAs. We have shown that if L and K are accepted by FAs then so are: L^*, L^+, L', L.K, $L \cup K$ and $L \cap K$. Unfortunately the description seems to be circular: we start by saying *"if L and K are accepted by FAs"* so we define new languages in terms of existing one but which existing ones?

Fortunately there is an easy answer: a language consisting of a single symbol is accepted by an FA. For example, the language { a } is accepted by



So if A = { a } we know that $A^* = \{ \Lambda, a, aa, aaa, \ldots \}$ is accepted by an FA, as is A^+ and A'. Also A^2, A^3, etc are all accepted by FAs. Since B = { b } is accepted we know AB is accepted as is $A \cup B$. But we can keep making new languages: $(A \cup B) . B^2 \cap (B \cup A^*)'$ is also accepted (whatever it is) as is any other language formed from A and B using the standard set and language operations.

The languages accepted by FAs are called *regular languages*. Here is a more formal definition.

1 a language consisting of a single symbol, for example { a } or { b } is a regular language
2 if X and Y are regular languages so is $X \cup Y$
3 if X and Y are regular languages then so is X .Y
4 if X is a regular language then so is X'
5 if X is a regular language then so is X^*
6 if X is a regular language then so is X^+
7 Only languages constructed using rules 1-6 are regular languages.

This is an example of an *inductive* or *recursive* definition. We could add $X \cap Y$ and $X \setminus Y$ (the intersection and difference of languages) but both of those can be written in terms of unions and complements and so don't add anything.

It consists of a *base case* (line 1) describing our starting point, plus *inductive (or recursive) definitions*(lines 2-6) which build on this. Line 7 is sometimes called a *closure requirement*. This way of defining things will be very useful several times in this module.

Q1 Show that the following are regular by showing that they match the definition above:

a) { cat, dog, dat, cag }
b) { cat, catcat, catcatcat, ... }
c) $\{ a^{2n} \mid n \geq 0 \}$
d) $\{ a^{2n+1} \mid n \geq 0 \}$
e) $\{ (ab)^n \mid n \geq 0 \}$
f) $\{ a^l b^m c^n \mid l+m+n = 6 \}$

Q2 Show that the language $\{ a^n b\, c^m \mid n \geq 0, m \geq 0 \}$ is regular.

Q3 Suppose X is the language consisting of all strings containing zero or more a's and zero or more b's in any order (e.g ababba, aabb, baabbaba, etc.) but excluding strings of all a's or all b's (and that includes the empty string). Show that X is regular (hint: line 4 may be useful).

Note that we now know that given any regular language we can find an FA which accepts it. The converse statement is "Given any FA we can find a regular language describing the strings it accepts". This we haven't proved but it is none the less true. Hein gives a proof (pp626-630) but it depends on some theory we haven't done yet. An alternative proof can be found in Appendix D.

6 Regular Expressions

FAs are used to match patterns in many computing applications. You will find them in Perl (a scripting language you will be learning), special searching tools (especially the Unix program *grep*), editors and compilers. Drawing pictures of FAs isn't a very practical way of inputting them to such software so an alternative method is used based on the ideas from the last section. Basically we'll describe a pattern in terms of its regular language description with a few simplifications and modifications to make typing easier.

First note that we hardly need to describe some FAs. We could use, for example, **abc** as shorthand for the FA which accepts the language { **abc** }. Following the most common notation (that used by *grep*) we use **abc | bcd | abb** to stand for the FA which accepts { **abc, bcd, abb** }. Then how about **(abc | bcd | abb) bd** for the FA which accepts one of **abc, bcd, abb** followed by **bd**?

FAs described in this way are called *regular expressions*. *grep* uses the following constructions (among many others) to describe FAs by means of regular expressions.

Symbol	Example	Explanation
|	**a | b**	**a** or **b** ($a \cup b$)
(sequence)	**ab**	**a** followed by **b** **(a.b)**
*	**a***	0 or more occurrences of **a**
+	**a+**	1 or more occurrences of **a**
.	**.**	Any single character. The sequence **.*** means 0 or more occurrences of any character
[ ]	**[xyz]**	Any of **x**, **y** or **z**, which is the same as **x|y|z**. Quicker to type if you want **[abcdefghijk]**, which can also be written **[a-k]**.
( )	**(a|b)c**	Grouping components together

In all the examples **a**, **b** and **c** can themselves be any regular expression, so, for example, **(ab*|cd(x|y))(a|b)** is a valid regular expression.

It is assumed that * has the highest precedence, then sequence then |. I.e. **ab*** means **a(b*)** (* has higher precedence than sequence), **abc | def** means **(abc) | (def)** (sequence has higher precedence than |).

Q1 Which strings are examples of which regular expressions? Some strings may belong to more than one regular expression.

1) 10(1*)| 010 2) (101)* | (010)* 3) (101)*(010)* 4) 10(11)*01

a) 101 b) 101101

c) 1001 d) 010

e) 010101010 f) 101010101010

Q2 Give examples of the following regular expressions:

a) ab(b|c)* b) (a|b(c|d)) c) (a*(b|c*)*

Q3 Is (a*|b*) the same as (a|b)* ? Explain your answer.

Q4 If we use the symbols a = RECORD, b = PLAY, c = RESET, d = PAUSE, what regular expression represents the behaviour of the data recorder at the beginning of this chapter?

Q5 Most programming languages allow numbers to be written in so called 'scientific notation'. Numbers can be written as normal, e.g. 3.4, 1000 (decimal points are optional) or as, for example, 1.234E7 where 'E7' means 'times 10^7' i.e. 1.234E7 = 12340000. In general the following are allowed: at least one digit must begin a number; there may be any number of digits followed optionally by a '.' and any number of further digits, 'E' is optional but if present must be followed by either one or more digits or '-' then one or more digits. Using 'd' to stand for any digit write a regular expression for a number in scientific notation.

Q6 Given the language L = (10 | 11)* | 110, what is L^R (i.e. the reverse of each word in L).

Q7 Suppose you have a general regular expression, E, can you provide a rule for producing a regular expression E^R which describes the reverse of the words given by E?

The following steps, when completed, should solve the problem.

1) If E is a single symbol then E^R = ??
2) If E has the form XY (i.e. a concatenation) then E^R = ??
3) If E has the form X |Y then E^R = ??
4) If E has the form X* then E^R = ??

Apply this 'algorithm' to the language in the previous question and check that it works.

Note that this proves that: If X is a regular language then so is X^R.

7 Limitations Of FAs

It is easy to find patterns that no FA can recognise. Consider the language { a^nb^n | n > 0 }. It is trivial to write an FA to accept any number of a's followed by any number of b's, the problem is to check that there are the same number. FAs have no general memory, they can't count so any language which can only be recognised by keeping track of how often something happens is too hard for an FA.

Q1 The previous statement isn't exactly true. Show that there is an FA which will accept the language { a^nb^n | M > n > 0 } where M is some known fixed number (if you are going to try and draw such an FA make M small, 3 maybe). On the basis of this is it true that an FA has no 'memory'?

Q2 No FA can recogise the language which consists of matching brackets, e.g the language { (), ()(), (()), ()()(), (()()), ((())), ()(()), (())(), ... }. Why?

To show that a language can't be recognised by an FA is usually quite easy. A standard technique is something called the Pumping Lemma which is described in Hein (pp632-633) but is beyond the scope of this module.

8 Conclusion

DFA = NFA

= Regular Expression

= Regular Language

PUSH DOWN AUTOMATA

1 Machines And Languages

The previous section concentrated on the control component of a computer. A finite state machine is a particular sort of control mechanism. One way of describing an FA is to describe the language that it accepts. It will be one of the so called regular languages which can be described by a regular expression. So for each FA we have a regular language and for each regular language there is an FA which accepts it. We can either talk about languages or FAs - there is a exact equivalence between them and we can translate from one to the other.

Not every language is regular. Indeed most of the languages you will encounter in computing aren't regular: JAVA isn't for example. Even many 'min-languages' fail to be regular, for example UNIX has a calculator program which evaluates expressions such as (1+3.4)*4+2*(1+2/(3-1)) the language of expressions isn't regular. It goes without saying that English isn't a regular language!

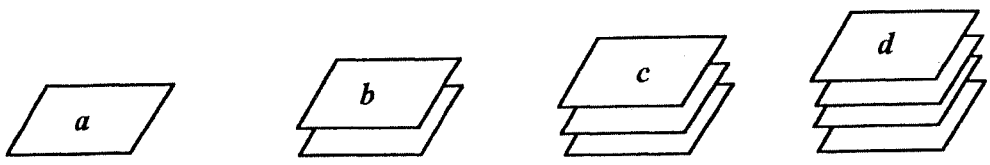
The existence of non-regular languages leads us to consider what other sorts of language there might be and what sorts of machine might be needed to recognise them. This section, and the next, look at two extensions of to FAs which add the next key computer component: memory.

If we look at the very simple non-regular language $\{ a^n b^n \mid n \geq 0 \}$ we get a hint of how an FA might have to be extended. The reason why this language is not regular is that recognising involves remembering how many a's have been seen and counting the same number of b's. An FA cannot count to an arbitrary number. It is true that an FA can count up to a fixed number in the sense that it could have states called 1, 2, ... N and being in state x would be equivalent to having counted to x. Of course the value of N has to be decided in advance of producing the FA. We could for example design an FA to accept $\{ a^n b^n \mid n \geq 0 \text{ and } n < 10 \}$.

What we need is to add some sort of 'memory' to our FA which allows us at least to count.

2 Stacks

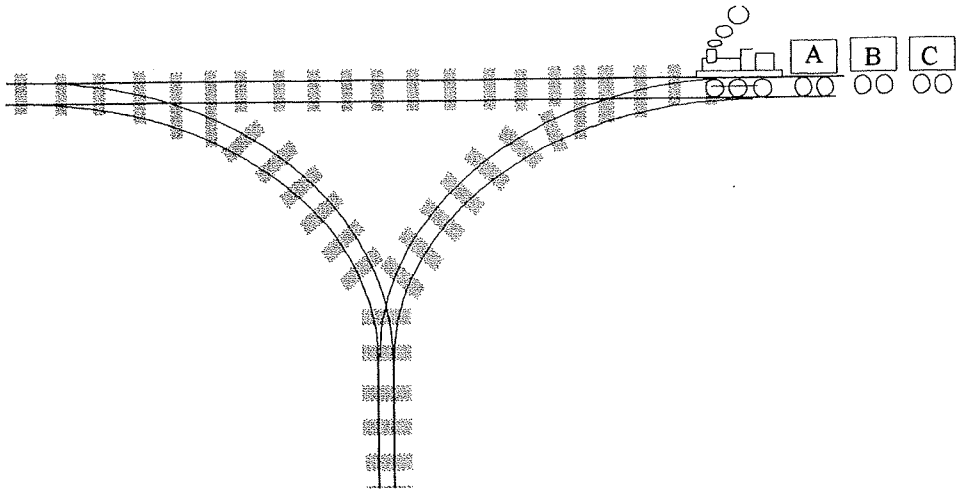
A *stack* is a sort of memory. As the name suggests data is stacked up, one item on top of another. Each item stored will be a single symbol. Storing **abcd** on a stack goes as follows. We assume that initially nothing is store on the stack.



a stored on stack | **ab stored on stack** | **abc stored on stack** | **abcd stored on stack**

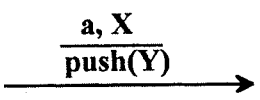
Adding a symbol to a stack 'hides' the symbols below. The top symbol is visible but none of the lower ones are. Accessing a stack is usually referred to as *pushing* an item onto the stack or *popping* the top item.

Q1 The diagram below shows a section of railway track (drawing's not my strong point). How can the train get the carriages into the order C, B, A? And what's it got to do with stacks?

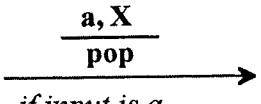


3 Push Down Automata

A *push down automata (PDA)* is an FA with a stack. Transitions not only change state but can cause the PDA to push a value onto the stack or pop whatever is on top of the stack. A transition normally has one of two forms:



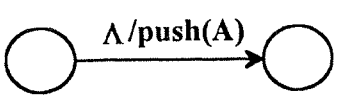
if input is a
and X is the top stack symbol
then push Y onto the stack



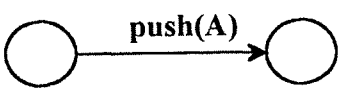
if input is a
and X is the top stack symbol
the pop (discard) the top stack symbol

It's often easier to write these as, e.g. **a, X / push(Y)**. One useful extension which add nothing to the power of PDAs are to allow **nop** (short for 'no operation') as an alternative to push and pop. **nop** just consumes another symbol from the input but leaves the stack unchanged. A second extension is to allow transitions which don't depend on what is on the top of the stack, i.e. **a / push(Y)** would mean: if the input is **a** then push **Y** onto the stack whatever value is currently on top.

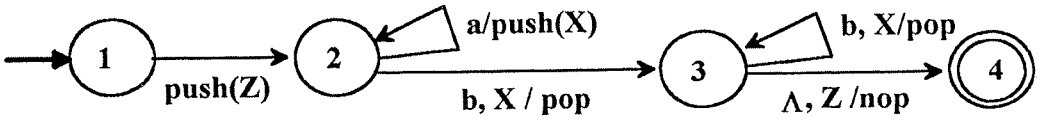
Hein also uses the convention that one symbol is placed on the stack initially and allows the input to be Λ, which is interpreted as meaning 'make this transition whatever the input symbol is and don't consume any symbols from the input'. We will ignore the first option (pre-pushing a symbol onto the stack) but keep the second. In fact once the second extension is allowed the first isn't necessary. For example we can draw a transition



which we will also draw as



Here is a PDA for the language { $a^n b^n$ | n ≥ 1 }, our version is slightly different from Hein's because we have assumed at least one **a** must be present. .



Let's see what can go wrong with the input and see how this PDA copes with it. Suppose the input were **aaab**. The stack would be set to (from the top) **XXXZ** by the **a**'s arriving. The first **b** would reduce it to **XXZ**. There is no exit from state 3 to a final state if the top stack symbol is an **X** so the PDA never gets to a final state. If the input had more **b**'s than **a**'s, e.g. **abb** then the second **b** would arrive with **Z** on the stack, the PDA would go to state 4. Now there is still a **b** to be read (Λ doesn't read a symbol) and the machine is stuck in a state with no transitions applicable to the input data still to come: i.e. it can't accept the data.

Is this a deterministic machine? Our definition of 'deterministic' for FAs was that there should be at most one transition from any state for any given input symbol. Another way of looking at this was to say we always know what decision a DFA will make once we know the next input symbol. For PDAs the decision is based on two symbols: the next input symbol and the current stack top. For example the transitions **a, X / ...** and **a, Y / ...** are different though the same input symbol is in both. Equally **a, X / ...** and **b, X / ...** are different. Allowing Λ complicates matters: Λ means ignore the next input symbol (which in our example could be **a** or **b**). So transitions like: **a, X / ...** and **Λ, X / ...** lead to non-determinism. Looking at the example we see that this machine is deterministic.

Q1 The language { (), (()), (()), (()()) ... } contains all sequences of matched '(' and ')' i.e. there must be equal numbers of each and they must be properly nested:)(for example isn't acceptable. Design a deterministic push-down automaton which can recognise words from this language.

Q2 Hein gives a non-deterministic recogniser for { $a^n b^n$ | n ≥ 0 }. Can you find a deterministic version? Hint: make the first **a** (if there is one) push a different symbol onto the stack from subsequent **a**'s.

Q3 Does the previous question get any more difficult if we allow different types of parenthesis, i.e. []{({}{})[]} becomes legal? (Assume each right parenthesis must match a left one of the appropriate kind)

Q4 Gunfighters in Westerns put notches on the handles of their guns to record how many people they'd shot. Show that a push-down automaton can check additions which use notches for counting, e.g. if Dangerous Dan McGrew's gun has notches ‹ ‹ ‹ ‹ ‹ ‹ ‹ ‹ and Catastrophe Ethel has ‹ ‹ ‹ ‹ ‹ then we can check ‹ ‹ ‹ ‹ ‹ ‹ ‹ ‹ + ‹ ‹ ‹ ‹ ‹ = ‹ ‹ ‹ ‹ ‹ ‹ ‹ ‹ ‹ ‹ ‹ ‹ ‹ (Ignore the possibility of 0 notches).

Q5 Here is a different language for additions: instead of writing 821+567=1388 we write the digits reversed and interleaved as 178268853001 the three numbers are all 'encoded' in this one string 1--2--8--0-- is the first number backwards with an extra 0 to make it the same length as the answer. The second number is included as -7--6--5--0- and the answer is --8--8--3--1. Convince yourself that an FSM does exist which can accept this language (don't try to draw one). If you understand binary numbers this is a lot easier - it's called a 'serial

adder' in Computer Architecture. Note that after all these examples about stacks we've finally shown you don't need one for addition!

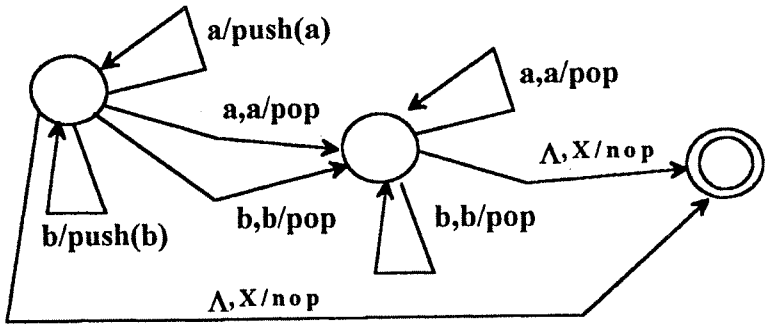
Q6 (Hard) If you can do binary arithmetic find a regular expression/grammar for the FA in the previous question where numbers are expressed in binary.

Q7 In some old calculators, arithmetic is done using *Reverse Polish Notation (RPN)*. In RPN numbers are entered immediately before the operation to be applied to them. The result then replaces the numbers and the operator. For example **1 2** + means add 1 and 2, **2 2 3** + * means: apply + to the preceding operands (2+3 = 5) giving **2 5** * now apply the * to get 20. Note that RPN has no 'BODMAS' rules, operations are applied from left to right as they are encountered. Suppose that the input to a PDA consists of numbers and the operations + and *. The symbol **n** stands for any number and **n/push(n)** means if the input is a number push it onto the stack. Finally assume that two new stack operations have been defined **mpy** and **add** which multiply (add) together the top two items on the stack and replace them with their product (sum). Write a deterministic PDA to evaluate any RPN string.

4 Determinism & Non-Determinism

NFAs are equivalent to DFAs. Are non-deterministic PDA equivalent to deterministic ones? The answer is 'no' as a simple example demonstrates.

A palindrome is a string of symbols such as **abcdeedcba** which reads the same left to right as right to left. We will limit ourselves to palindromes involving only two symbols **a** and **b**. A deterministic PDA cannot recognise palindromes. Suppose for example the input is **abaaaaba.** A deterministic way of checking this is a palindrome is to push symbols onto the stack until you reach the middle of the string and then pop the stack 'against' the rest of the palindrome. Unfortunately there is no way a deterministic PDA can tell that the middle has been reached. A non-deterministic PDA can have a transition equivalent to 'if this is the middle start unstacking' and 'if it isn't the middle keep pushing symbols onto the stack'. Here is the non-deterministic version



Q1 If a special symbol, **x** say marks the centre of the palindrome, e.g. **abaaxaaba**, and only appears in the centre, show that a deterministic PDA can recognise palindromes (you can limit yourself to palindromes in **a** and **b** as well as **x**).

5 PDAs And Languages

We have already seen several examples of languages which are accepted by PDAs but not by FA. Also the languages are accepted by NPDA (non-deterministic PDA) differ from those accepted by deterministic PDA. However, just as we associate regular languages with FAs, there is a class of

languages associated with NPDAs: the *context free languages*. Context free languages form the basis of all the languages used in computing. JAVA, C, Visual BASIC etc. are all examples of such languages. Regular languages are of course a subset of context free languages: any regular language can be recognised by a PDA (just don't bother to use the stack).

As suggested earlier in the module the input to a program can always be thought of as belonging to some language. It is hard to imagine any program accepting data more complex in structure than that accepted by compilers, so an understanding of context free languages can be the basis of reading any sort of input. Later in the module we will investigate context free languages in greater detail.

6 Stacks

The key difference between PDA and FA is the addition of the stack. Why was the stack chosen as a model of memory? Consider the simple non-regular language $\{ a^n b^n \mid n \geq 0 \}$. What is happening when a stack based automata matches this but an FA can't?

One way of looking at the stack is that it gives a way of saving data until a decision can be taken. In the example above, a's are pushed onto the stack and 'forgotten' until the first b is encountered. The second feature of the stack is that popping allows us to work backwards through the data.

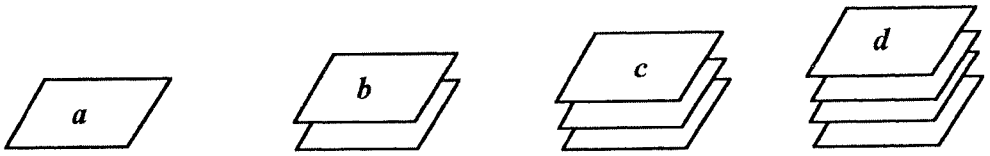
We can liken this to a common every day experience. We work on something and are interrupted. We put aside (push) what we are doing and deal with the interruption. When the interruption has been dealt with we go back to (pop) what we were doing before. The stack allows us to be interrupted while in the middle of an earlier interruption as well.

It turns out that the use of a stack is a powerful programming concept. The next section will consider the disciplined use of a stack - a technique often called 'recursion'.

1 Stacks

Stacks were introduced when Push Down Automata were described. In various forms, stacks will come up again and again in Computer Science so this section is devoted to them and to some related ideas which together are essential parts of many algorithms. It is worth repeating the basic idea from the PDA notes.

A *stack* is a sort of memory. As the name suggests data is stacked up, one item on top of another. Each item stored will be a single symbol. Storing **abcd** on a stack goes as follows. We assume that initially nothing is stored on the stack.



a stored on stack | **ab stored on stack** | **abc stored on stack** | **abcd stored on stack**

Adding a symbol to a stack 'hides' the symbols below. The top symbol is visible but none of the lower ones are. Accessing a stack is usually referred to as *pushing* an item onto the stack or *popping* the top item.

2 A Stack Class

A class called *Stack* exists in Java but it is designed to support only stacks of Java *Objects*. I have written a new class which extends *Stack* and allows any type of item to be added to the stack. My version is called *StackT* and has the following methods and constructor.

Member	Purpose	Example
StackT()	Constructor: creates an empty stack	Stack s = new Stack()

push(Object v)
push(*xxxx* v)
	Pushes v onto the stack. v can be an *Object*, or of type *xxxx* where *xxxx* is one of: *char*, *byte*, *short*, *int*, *long*, *float*, *double* or *String*.	s.push("abc");
		
		s.push(42);

pop()	Pops the top object from the stack and returns it (as an object)	x = s.pop();
peek()	Returns the top of the stack as an object without popping	x = s.peek();
peek(int n)	Returns the n^{th} object from the top (1 = top, 2 = top but one, etc) of the stack. **peek(1)** is the same as **peek().**	if(s.size() >= 3) x = s.peek(3);

***xxxx*Pop()**	Pops the top value from the stack assuming it to be of type *xxxx* as above.	char ch = s.charPop();
***xxxx*Peek()**	Returns the top value from the stack, without popping, assuming it to be of type *xxxx* as above.	int t = s.intPeek();
xxxxPeek(int n)	Returns the value n from the top of the stack, without popping, assuming it to be of type *xxxx* .	float f = s.floatPeek(2);
empty()	returns true if the stack is empty	if(s.empty()) ...
size()	returns the number of items on the stack	if(s.size() > 3) ...

Q1 What is printed out by the following fragment of Java?

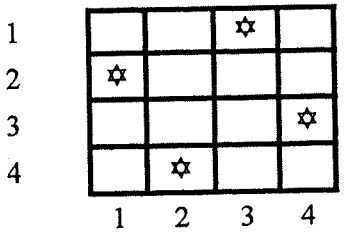
```
StackT s = new Stack();
for(int i = 0; i < 10; i++) s.push(i);
while(!s.empty()) System.out.println(s.intPop());
```

Q2 What is printed out by the following fragment of Java?

```
StackT s = new Stack();
for(int i = 0; i < 5; i++)   s.push(i);
while(s.size() > 1) s.push(s.intPop()+s.intPop());
System.out.println(s.intPop());
```

3 Using A Stack - The N-Queens Problem

In chess a queen can take any piece which lies on the same row, column or diagonal. The N-Queens problem asks if it is possible to put n-queens on an n x n chessboard so that no queen can take any other. On a 2 x 2 chess board it is obviously impossible, nor can it be done on a 3 x 3 board. A 4 x 4 board is the first on which it can be done:



Is there a general algorithm for finding some or all positions for n-queens? First we note that the entire layout of the board can be described with 4 numbers: (2, 4, 1, 3) - the row number for the queen on each column. If also write, for example, (2, 4) for the first two columns when I haven't decided where the rest of the queens go.

I started with one queen in row 1 column 1. I'll write this as (1). Looking at column 2 the first place a queen could be placed was row 3 giving (1,3) but then I was stuck for column 3. I tried the next position for column 2: (1, 4) which allowed (1, 4, 2) as a partial solution but I was stuck again. Going back one, I'd tried all the positions for column 2 so I 'backtracked' to (1) and changed it to (2). Going forward again gave (2, 4) as the only safe place and the first safe position in column 3 is 1 giving (2, 4, 1) and the only safe row in the last column is 3 which is the final configuration.

Q1 What has this got to do with a stack?

Q2 Devise a complete algorithm for solving the problem. How using JAVA can you say that a position is safe: i.e. a piece placed there can be taken by any other piece so far placed on the board?

Q3 Ignoring whether or not pieces can be taken, in how many different ways can 4 queens be placed on a 4 x 4 chess board? How many positions are there for the general N-queens problem. An alternative to this algorithm is to generate every position and then test whether it is safe. If in the 10-queens case it takes $1/10000^{th}$ of a second to test whether a position is safe approximately how long would a computer take to test all positions?

Q4 What is, approximately, the largest number of positions to be tested in the backtracking algorithm for the n-queens case? Is this significantly better than the 'brute force' technique above. (Stirling's formula states that, for large n, $n! \sim (n/e)^n \sqrt{(2\pi n)}$ where $e = 2.71...$)

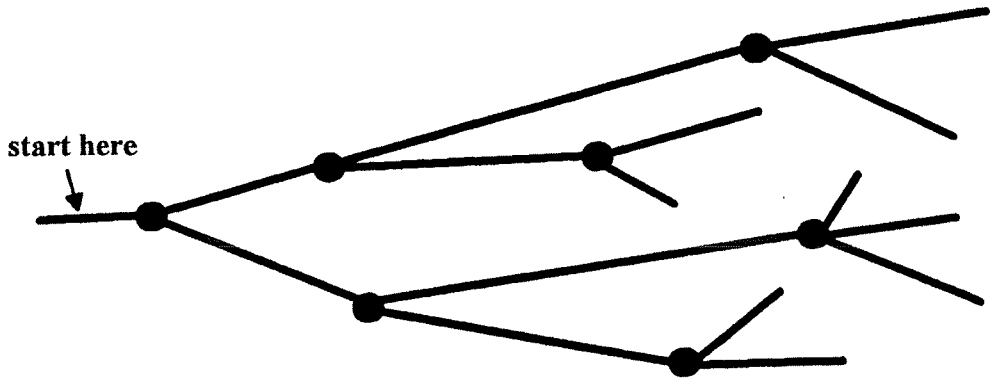
Q5 A crossword includes the clue "Confused life can smooth things". As a cryptic clue expert you immediately guess that the answer will be an anagram (rearrangement) of the letters of 'life' to mean something which smoothes things. Can you modify the n-Queens algorithm to find all rearrangements of a word?

4 Backtracking

The previous examples were a special cases of a technique know as *backtracking*. Here is a slightly different example. Can you spot the similarities?

> *A friend tells you that their house is number 42 and is 100 yards from the roundabout. Unfortunately you realise, as you reach the roundabout, that they haven't told you which exit to take. What should you do? The simple answer is to take any exit and see if you find number 42 after about 100 yards. If you don't find it (or it's the wrong number 42), go back to the roundabout and try another exit. Again if you fail backtrack to the roundabout and try again. Hopefully you'll get there eventually.*

Suppose you had received even more useless directions: the house is 100 yards from the third roundabout. Helpfully your friend has drawn a map but has forgotten to mark their house! The map looks like this:



You don't know which exit to take at any of the roundabouts (conveniently you do know that the house is less than 2 miles away). What should your strategy be?

The answer is to use a stack. Consider the following algorithm:

1 when you come to a roundabout push 1 (for 'first exit') onto the stack
2 take the exit whose number is on the top of the stack
3 if it was the third roundabout and you find the house stop
4 if it was the third roundabout and you don't find the house go to step 6
5 if you come to another roundabout within 2 miles go to step 1
6 go back to the last roundabout
7 add one to the number on the top of the stack
8 if there is an exit with that number go to step 2
9 remove the number from the top of the stack
10 if the stack is empty you are stuck!
11 go to step 7

By experimenting with some examples convince yourself that the algorithm works.

It should be clear that there is a very general backtracking algorithm underlying this: go as far as you can in any one direction, if you solve the problem stop, if there's no solution along this route back up to the last point where a decision was made and try making a different decision. If there are no new decision which could be made backup a further stage and so on.

Q1 You have to build a tower of a given exact height from a collection of boxes of different sizes. The sum of the heights of all the boxes is greater than the required height. Can you apply the general backtracking algorithm to this problem?

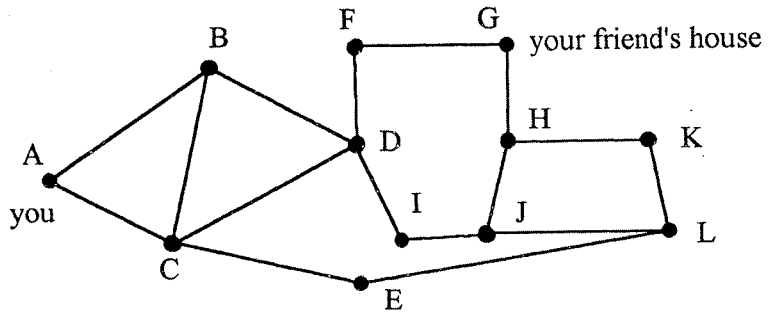
Q2 A haulage company moves several loads a day to the same destination. The loads consist of many items which may be of different weights however there is a weight limit for loads which must not be exceeded. Can you suggest a strategy for minimising the number of vehicles needed? (There is no one correct answer for this, just try to think of sensible strategies which allow for backtracking). This problem assumes that weight, not size of items, is the only issue. If objects can be different weights and different sizes the problem gets very complicated!

5 Depth First Searches

You have lost all the instructions on how to find your friend's house and you don't know where it is relative to your current position. Assuming you know the name of the road where the house is, you need a strategy which, starting from your present position, searches the whole network of roads around you until you find the correct road.

This is different from the back tracking version in one important respect: roads can join up to form complete loops which take you back to where you started. Travellers getting lost and going round in circles is a comedy cliché but can it be avoided?

Here is a simplified map as an example (of course if you had the map you wouldn't be lost).



In computer science such a diagram is called a *network* or *graph*, the points labelled A, ... L are called *nodes* or *vertices* while the lines joining them are called *arcs*. We could number the arcs in some way but we'll just write AB for the arc joining A to B and ABDI for the *path* from A to I via B and D. What was above called a loop is normally called a *cycle*. ABDCA is an example of a cycle.

The problem is to find an algorithm which eventually gets you from any point in the graph to any other. Very crudely there are two general strategies, *depth first* and *breadth first*. Depth first search says go as far as you can in any direction, if you reach your target you're done, if not when you can go no further, backtrack to the last node visited and try an alternative from that point, if there is no alternative backtrack again. This is exactly the general backtracking algorithm but with an added complication. Suppose you start out by following the path ABC, if you turn right (to D) there's no problem. If you turn left you could find yourself in a cycle and not know how to break out of it.

Q1 You have a piece of chalk with and paper on which to draw a stack. Find an algorithm for a depth first search going from node A to any other node in the graph.

Q2 Because of your knowledge of the area you have been employed as a door-to-door salesman. Devise an algorithm which is guaranteed to take you down every road at least once.

Q3 You know how far it is from your starting point to where you want to be and you know the know the length of each road on the map, can you improve depth first search in such circumstances?

Q4 Conveniently you have in your pocket a GeoSat positioning device which gives you your exact latitude and longitude and, coincidentally, you know the latitude and longitude of your friends house. Can you suggest a sensible way of choosing which exit to take from a node as you do a depth first search?

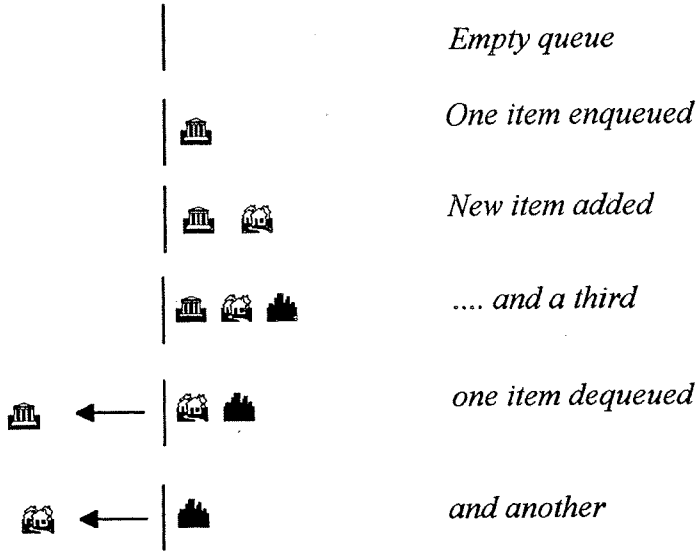
6 Breadth First Searches

This is a diversion as it doesn't depend on stacks instead it uses a slightly different memory device called a queue.

In the diagram above, suppose your friend lives at node C. A depth first search could take you to every other node in the network before visiting C. Perhaps a better strategy would be to visit every nearby node first, then move on to further away ones. This is called breadth first search. To make it work, assume you draw a map as you go along (this means you can always find a way between any two nodes you have already visited).

Whereas the depth first search requires us to remember where we've been, a breadth first search requires us to remember where to go next. Instead of a stack we'll use a related structure called a

queue. A queue, is exactly as its name suggests, a computer model of the sort of queues you find at bus stops or in banks. You join the end of the queue (*enqueueing*) where you remain till you reach the front and are *served*, *unqueued* or *dequeued*. Here is an example:

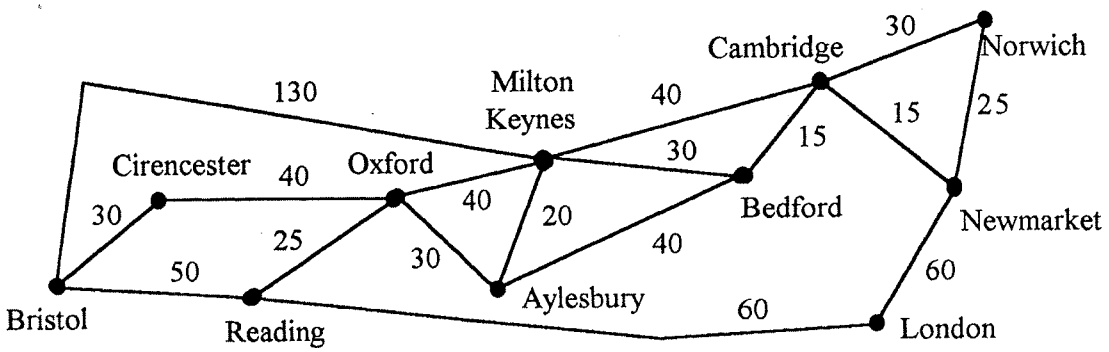


Roughly a breadth first search requires:

1 put the starting node into the queue
2 take the first item, N, from the queue
3 if it's where you want to go stop
4 mark N as 'visited' and add the unvisited immediate neighbours of N to the queue
5 go back to step 2

Q1 Which nodes are visited and in which sequence if you do a breadth search for G from A in the example?

Q2 (Harder) CSM's software includes *Autoroute* a program which, among other things, can find the shortest route between two points. Can you work out an algorithm for doing the same? (Hint: start from breadth first search, for each node reachable from the start point write down the shortest route to it. Then consider points which can be reached in 2 stages, then 3 etc.)



Test you algorithm by finding the shortest route between, say Bristol and Norwich.

INDUCTION AND RECURSION

1 Inductive Definitions[1]

In a previous section we gave an example of an inductive definition

1 a language consisting of a single symbol, for example { a } or { b } is a regular language
2 if X and Y are regular languages so is $X \cup Y$
3 if X and Y are regular languages then so is X .Y
4 if X is a regular language then so is X'
5 if X is a regular language then so is X*
6 if X is a regular language then so is X^+
7 Only languages constructed using rules 1-6 are regular languages.

An inductive definition has three parts, base cases (in this case line 1) which describe the simplest possible examples, some inductive cases (lines 2 to 6) which describe how more complex examples can be built from existing ones, and a closure clause (line 7) which says that only items defined in this way count as examples.

Here is another example: what is an integer?

1 0 is an integer
2 if n is an integer so is n+1
3 if n is an integer so is -n.

To show that this rather unlikely definition works we will 'prove' that -2 is an integer.

a) 0 is an integer
b) therefore 1 is an integer (0+1)
c) therefore 2 is an integer (1 + 1) and we've proved 1 is an integer
d) therefore -2 is an integer using line 3.

Another example is a definition of a palindrome: *a palindrome is either an empty string, a single character or a palindrome with the same character added to each end.*

Writing this in the standard form.

1 An empty string is a palindrome
2 A single character is a palindrome
3 if P is a palindrome, then so is xPx where x is any character.

Note that here there are two base cases.

Q1 Can you give an inductive definition for:

a) powers of 2: 1, 2, 4, 8 ...
b) odd numbers: 1, 3, 5, 7, ...
c) the strings belonging to $\{ a^n b^n \mid n \geq 0 \}$
d) { (), (()), ()(), (())(), ()(()), ((())), ... }

[1] See: Hein 3.1 Inductively Defined Sets

Identifiers (names) in some programming languages start with a letter which can be followed by any number of letters and digits. Here is an inductive definition:

1 A letter is an identifier

2 If X is an identifier then so is X followed by a letter

3 If X is an identifier then so is X followed by a digit

One of the advantages of inductive definitions is that they can be very precise. In the informal description of an identifier it wasn't clear whether 'any number' could include none. The more formal inductive definition makes it precise (is a letter alone allowed?). Typically inductive definitions avoid phrases like 'any number of', 'some', 'n' replacing them by single step type definitions like that above.

Of course if I ask "Is *absu12jh23j24* a valid identifier" you probably wouldn't work through the definition line by line, but if you were writing a program which had to recognise whether a string was a valid identifier or not the inductive definition might be a good starting point.

Q2 A sum consists of either a number or several numbers separated by '+' signs, e.g. 3 is a sum and so is 3+3+1+6. Give an inductive definition.

Q3 An *expression* in computing usually means a formula such as (x+3)*y / 4 which appears in a programming language. The simplest expression would be just a number, such as 3, or a name such as x. Give an inductive definition of an expression.

Q4 Given any string S we will write head(S) for the first character of S and tail(S) for all characters except the first. As usual |S| is the length of S. When are two strings A and B in alphabetical order (i.e. A would come before B in a dictionary)?

A slightly different form of inductive definition depends on chosing some object. For example a graph (network) is a collection of nodes and arcs (lines joining pairs of nodes). A path is a sequence of connected arcs joining two nodes. But that definition of a path strictly requires some explanation of what is meant by a 'sequence of connected arcs'. Here is an inductive definition:

There is a path from node A to node B if either:

1 there is an arc from A to B, or

2 there is an arc from A to some node C and there is a path from C to B.

Many inductive definitions follow this model.

Q5 Define what it means for A to be an ancestor of B.

Q6 A list of numbers are in ascending order if as you go through the list from left to right the numbers get bigger. Assume a list can have one element or more, and that as for strings, if L is a list then head(L) is the first number, tail(L) is the list obtained by removing the first element and |L| is the number of numbers in the list. Define 'ascending order' inductively.

Q7 My apologies for the age of this question! The blues guitarist John Mayall is famous for the number of other people who have played in his band: Eric Clapton, Peter Green, Mick Fleetwood, Jon Hiseman etc. One way of measuring distance to the centre of things in the blues world would be to how far you are from having played with Mayall. Mayall gets distance 0, Eric Clapton, Peter Green etc. get distance 1, anyone who played with Clapton but not with Mayall himself gets 2 and so on. Give an inductive definition of the Mayall distance. A distance of ∞ is given to anyone who has no connection, however remote, with John Mayall. *Initially ignore the fact that someone might have played with people with different Mayall numbers. How would you take this into account?*

2 Recursion[2]

Recursion is induction applied to programming. When we define something inductively we start with a base case (the easy one) and from that describe how 'bigger' objects can be built up. Recursion typically tries to solve a problem by deciding if it is dealing with a base case of the problem and if not trying to break the problem down into simpler cases.

There is a standard example of this. n! = n * (n-1) * (n-2) * .... * 1. We may notice a pattern here: (n-1) * (n-2) * ... * 1 is the same a (n-1)! which gives us the starting point for an inductive definition of n!

1 if n = 1 then n! = 1

2 if n > 1 then n! = n * (n-1)!

(We will ignore n! for n < 1).

This is a valid definition but it can also be the basis of a method in a JAVA program:

```
static public int factorial(int n)
{   if(n == 1)
        return 1;
    else
        return n * factorial(n-1);
}
```

A call to factorial(3) will return the result 6 as expected. It may be useful to see how this works. Imagine the call to factorial(3)

```
factorial(3) ⇒ call factorial with n = 3
in factorial: if(n == 1) .... false
                  else
                      return n * factorial(n-1) with n = 3, i.e. return 3*factorial(2)
                      this can't be computed until factorial(2) is known so ...
                          call factorial(2) ⇒ call factorial with n = 2
                          in factorial: if(n == 1) .... false
                                            else
                                                return n * factorial(n-1) with n = 2,
                                                i.e. return 2*factorial(1)
                                                this needs factorial(1) to be known so ...
                                                call factorial(1) ⇒ call factorial with n = 1
                                                in factorial: if(n == 1)
                                                    return 1;
                                                now 2*factorial(1) = 2*1 is known
                                    so  factorial(2) = 2
```

so factorial(3) = 3*factorial(2) = 3*2 = 6

This is very complicated but the advantage of recursion is that you don't need to follow through all the details. Here is another way of looking at the problem:

1 the method returns the correct answer if n=1

2 if factorial(n-1) gives the correct answer so does factorial(n)

---

2 See: Hein 3.3 Recursively Defined Functions and Procedures

If you look at the algorithm and compare it with the inductive definition you will see that claim is correct: factorial(1) = 1 (which the method gives) and factorial(n) = n*factorial(n-1) if n > 1 (which the method also gives).

Why is this sufficient to prove the algorithm works? We know it works for n=1 so suppose n = 2. We know that if the method works for n-1 then it works for n, in this case n=2 so n-1 = 1 and the claim is that if it works for 1 it works for 2. It does work for 1 so it works for 2. Now try n=3, you'll see it works for n=3 if it works for n=2 .... which it does. And so on for any n.

Another inductive definition was of a palindrome.

1 An empty string is a palindrome

2 A single character is a palindrome

3 if P is a palindrome, then so is xPx where x is any character.

Here is a JAVA method which decides if a string is a palindrome. You aren't expected to now what all the functions used do but if you know that the characters in a JAVA string are numbered from 0 to slen-1 where slen is the number of characters in the string you should be able to work out most of what is happening.

```
static public boolean  isPalindrome(String s)
{   int slen = s.length();                  // number of  characters in string
    if(slen == 0 || slen == 1)
        return True;                        // a null string or a single char is a palindrome
    char char_first = s.charAt(0),          // first and last characters
         char_last = s.charAt(slen-1);
    if(char_first != char_last)
        return False;                       // can't be a palindrome
    String rest = s.substring(1, slen-1);   // the bit in the middle
    return isPalindrome(rest);              // if rest is a palindrome so is s, if not, not
}
```

(We should point out that neither this nor the factorial methods are very efficient in that it is easy to write a non-recursive version which would probably execute more rapidly.)

As before, you can show that you get the correct answer if the string has length 0 or 1 (the base cases), and if the length is greater than one then the method works for length n if it works for length n-2 (since 2 characters are removed before the recursive call).

It is perfectly possible to write a recursive procedure which doesn't work. Here is an example.

```
static public void silly(int n)
{   if(n <= 0)
        System.out.println("Finished")
    else
        silly(n+1)
}
```

This prints "Finished" if called with n equal to 0 or a negative number but otherwise just goes on for ever. How can we be sure a recursive method ever terminates? There are two requirements: the recursive call must be moving from more complex to simpler cases and at least one of the base cases must eventually be reached.

In the factorial example, the recursive call had parameter n-1 so every it called itself the parameter would be smaller than the previous time (the calls for n=3 were factorial(3), factorial(2), factorial(1)) and the base case was n =1. As long as factorial is called with $n \geq 1$ initially it must eventually stop.

Q1 Why must the isPalindrome method terminate eventually?

Q2 x^n (i.e. x multiplied by itself n-1 times) can be computed recursively using the following inductive definition:

1 $x^0 = 1$

2 $x^n = (x^{n/2})^2$ if n is even

3 $x^n = x * x^{n-1}$ if n is odd.

In JAVA you can decide if a number is odd or even by taking remainders n % 2 is the remainder on dividing n by 2. Write a method pow(float x, int n) which computes x^n.

Q3 Suppose you want to print out a number in binary. There is a fairly standard method which relies on the following definition for a binary number.

1 0 and 1 are binary numbers

2 if n > 1 then it can be written as m*2 + r where r = 0 or 1 and m = n/2 (integer division)

Fill out the details of the following recursive method.

```
static public void printBin(int n)
{   if(n < 2)
        System.out.print( ..... );
    else
    { int m = n / 2,
        r  = n % 2;
        printBin( ..... );
        System.out.print( ..... );
    }
}
```

Q4 The Fibonacci sequence[3] comprises the numbers 1, 1, 2, 3, 5, 8, 13, ... where every number after the first 2 is the sum of the previous two numbers. Fill in the gaps in the following definition of a function to compute the n^{th} Fibonacci number:

```
static public fib(int n)
{   if(n <= 2)  return ????;
    return ???? + ????;
}
```

Q5 The gcd (greatest common divisor) of two integers is the largest whole number which divides both of them. An example: gcd(6, 15) is 3 because 3 divides 6 and 15 but no bigger number does. Other examples: gcd(3, 4) = 1, gcd(2, 4) = 2, gcd(12, 9) = 3 and so on. By convention gcd(x, 0) = x = gcd(0, x) so gcd(7, 0) = 7 for example.

Euclid worked out (a long time ago) that if $a \geq b$ then either b = 0 and the gcd is a, or it's the same as gcd(b, a % b). Assuming that when the method is first called, $a \geq b$, write a gcd function in JAVA.

Can you show that 1) if $a \geq b$ on the first call it is on all recursive calls and 2) the recursion always terminates.

---

[3] Hein p149

Q6 In Pascal's triangle each number is the sum of the number above it and number diagonally above it to the left (if there is no such number assume use zero). The first few rows are:

1						
1	1					
1	2	1				
1	3	3	1			
1	4	6	4	1		
...	...	...	...	...	...	...

We will write C(n, i) for the i^{th} number on the n^{th} row where both i and n are counted from 0, C(n, -1) and C(n, n+1) are always treated as zero as is C(-1, i).

Can you provide a recursive definition of C(n, i)?

Can you write a fragment of Java to compute C(n,i) based on the definition?

Q7 A string consists of letters in alphabetical order, e.g. "acdegmrtvw" design a Java fragment to decide whether a given character appears in the string. Here is a clue:

```
static public boolean search(string s, char c)
{
    return search2(s, 0, s.length()-1, c);
        // search the part of the string between first and last character
}

static public boolean search2(string s, int low, int high, char c)
// if there's nothing left to search return false
// otherwise check the middle character
// if the middle character is c we are done
// otherwise either the characters before the middle or those after it have to be searched
```

Q8 A recursive method with parameter n shows the following pattern of calls:

if n = 1 or 0 it terminates

if n is even it calls itself with parameter n/2

otherwise it calls it self with parameter n+1

Does it always terminate?

Q9 (A bit of more maths needed) The ancient Egyptians understood fractions a long time before Europeans did (about 3000 years before) but they didn't write them as we do. They only used fractions where the top term was 1. What we would write as 3/8 they would write as 1/3 + 1/24 (why not 1/4+1/8?). 7/10 would become 1/2 + 1/5. How can we convert between the two forms? In particular given a and b representing a/b can we print out the corresponding Egyptian fraction?

The following rule will do the printing. First notice that if a = 1 we are done and the fraction can be printed directly. If not divide a into b rounding up the answer. In JAVA this can be done using:

```
int d = (b + a -1)/a;
```

1/d is the first term and the rest is found by computing a/b-1/d = a*d - b / (b*d) and applying the same algorithm to the new fraction.

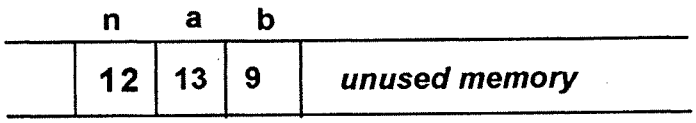
Can you convert this into JAVA?

3 Stacks AND RECURSION

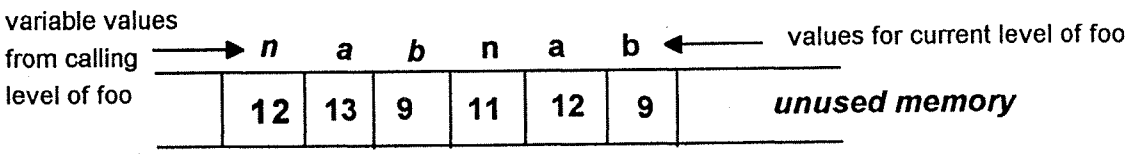
What is the connection between stacks and recursion? Recursion in programming languages does involve a stack but one hidden from you by the compiler. Consider the following, totally pointless, Java function:

```
static public  void    foo(int n)
{   if(n <= 10)
        System.out.println("Number is "+n);
    else
    {   int a = n;                                    // local variables
        int b = 10;
        while(b-- > 0)
        {   a = a +1;
            foo(n-1);                                 // !!!  recursion here  !!!
            System.out.println("Another number is "+a)
        }
    }
}
```

Suppose a call to foo(12) is made, what happens? Internally a memory location is used to store the value of n (i.e. 12) and, when the line labelled 'local variables' is reached, additional memory locations are needed for a and b. Imagine the memory locations for n, a and b stored in memory just before foo(n-1) is called:



As foo(n-1), in this case foo(11), is called the compiler uses the next bit of memory for new variables n, a and b for the next level of call:



Each recursive call uses storage locations further up memory. When a call of foo eventually returns its variables can be discarded and the previous set re-instated. This is just another stack but hidden from you by the Java compiler.

Every program which uses recursion can be replaced by one that uses an explicit stack. The converse is also true though it isn't nearly as easy to see how to convert explicit stack pushes and pops into recursive calls in general.

LANGUAGES & GRAMMARS

1 Recap

The idea of a language as a set of strings of symbols conforming to some set of rules has been a major part of this module. It can be shown than every computation that can be done by a computer can be modelled as a question of deciding whether a string belongs to a particular language (or maybe whether a language contains no strings at all). Thus, theoretically at least, every computing problem is a language problem.

Even if we restrict ourselves to 'practical' computing problems, a knowledge of languages can be useful. Several methods of program design are based on recognising the structure of the input data, in effect using the language of the input to determine how the program is written.

A number of different ways of describing languages have been used. The first and most general is to use the set notation. For example: $\{ a^n b^n \mid n \geq 2 \}$ describes the language {aabb, aaabbb, ...}. Set notation is very general - too general in many cases. It describes what the language is but doesn't give any help in deciding whether a particular string is member of the language.

Finite Automata can also be thought of as a way of describing languages. Each FA accepts strings from a given language so drawing an FA is a way of specifying a language. The disadvantage of FAs is that they only accept a small subset of all possible languages, specifically what are known as regular languages. Regular Expressions are an alternative way of describing regular languages. We write expressions such as a* (b|cd) e+ to specify a string containing zero or more a's followed by either b or cd then one or more e's. Regular expressions are equivalent to FAs in that they are both ways of specifying regular languages.

One advantage of regular languages is that it is easy to decide if a string belongs to a regular language. This is used in many practical applications: data communications methods often structure data in a way that allows a finite automata to check the correctness of the data and decode messages and the chips that control electronic equipment such as video recorders will often include special purpose FAs.

Finally, we can use inductive definitions to describe languages and recursion to recognise whether strings belong to languages. Regular expressions and languages can be described inductively but so can most other languages. For example the strings comprising the language $S = \{ a^n b^n \mid n \geq 2 \}$ are given by the inductive definition

1 $aabb \in S$
2 if $w \in S$ then so is awb
3 nothing else belongs to S

Inductive definitions are useful because they often lead to recursive programming methods which can recognise whether a string belongs to the language.

Here is a Java method which checks whether a string belongs to the language (the method subString extracts all but the first and last character of s)

```
public static boolean inLanguage(String s)
{   int slen = s.length();
    if(slen < 4) return false;                          // can't be in language
    if(s.equals("aabb")) return true;                   // base case
    if((s.charAt(0) == 'a') && (s.charAt(slen-1) == 'b'))   // starts & ends ok
        return inLanguage(s.subString(1, slen-2));      // check 'inner' string
}
```

This part of the module will look at a final method of describing languages which attempts to combine the convenience of regular expressions with the power of inductive definitions. It is the method used to defining programming languages and an understanding of it may help you understand what is legal in Java or other languages.

2 Designing An Estate

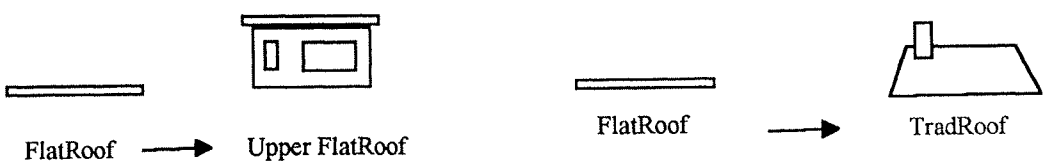
An architect takes the following approach to designing housing estates:

> *Start with a piece of land, then level it to give a building plot. Any building plot can be cut in half to make two smaller plots, and then again as often as you like. A plot can be used either for flowers or for a building. Start with a one storey block on any plot to make a ground floor with a flat roof. You can go up as far as you like - just replace the flat roof with another storey. Eventually you have to put a proper roof on - when everything's got a roof you're done.*

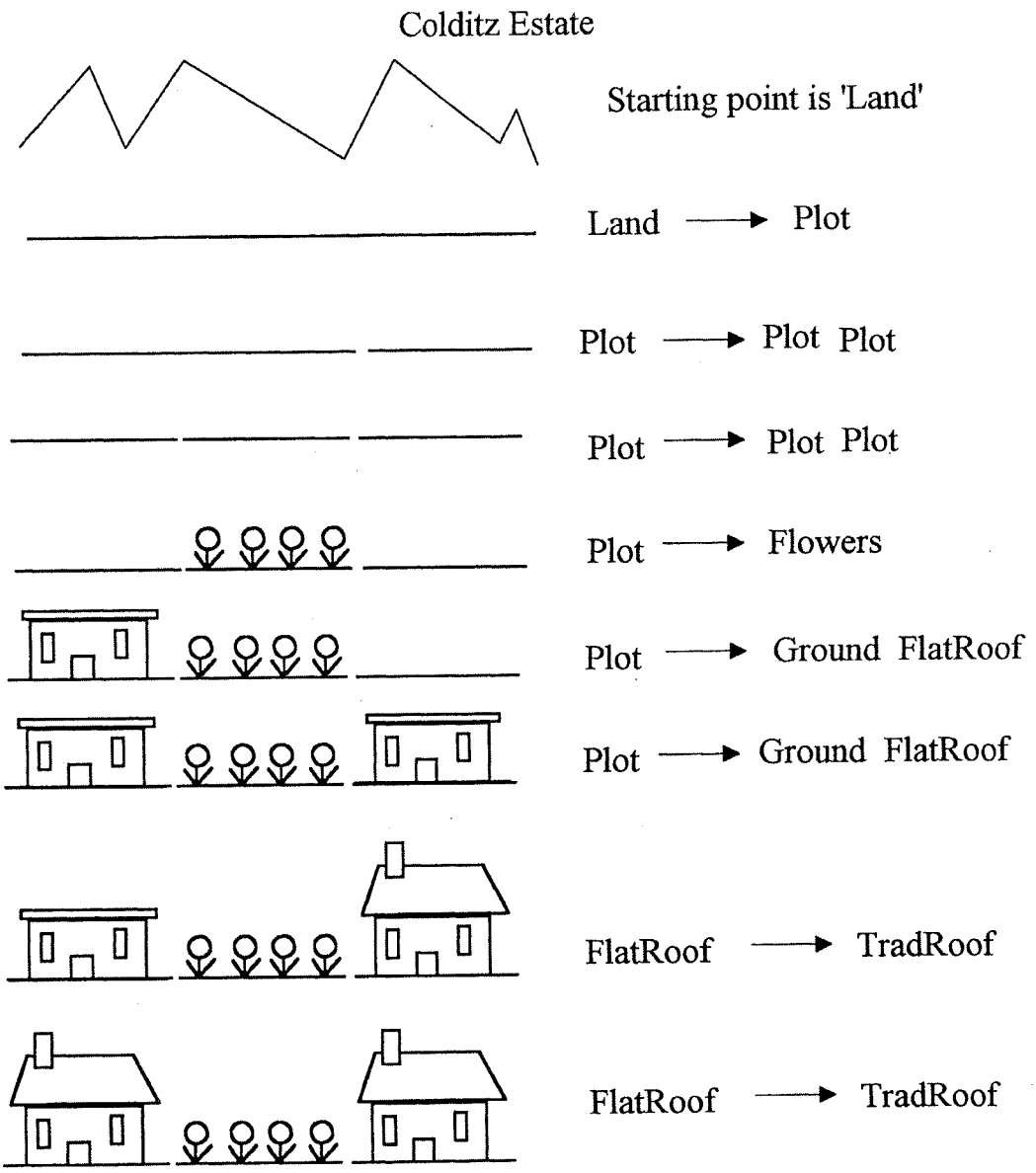
I'll try and explain what has been said in pictures. Starting with a land we produce a plot. (I'll just use two dimensions to make the drawing easier) and plots can be split into two. This is shown below.

I won't insist that the plots are any particular size (later we might consider how to do that) and in real life there is a limit to how small they could be - a one foot square plot wouldn't be much use. Each plot can be split in half again but we would only draw the same picture again so I won't bother. There are two uses for a plot - for flowers or as the start of a building. I'll use two pictures again and shorten the names slightly.

I've added a bit - only the ground floor will ever have a door and I'll have a second sort of block for upper floors. Now we start building upwards including a rule for putting on a proper roof ....



What we have here are a set of 'transformations' (or *productions* to use the correct term). Starting from 'Land' we could produce any number of designs by just applying any production. For example let's design an estate with two bungalows on it and some open space in the middle. I'll show all the stages and write the rule used at the side.



The set of productions provides rules for what constitutes a 'legal' estate. The productions used were

Land	→ Plot
Plot	→ Plot Plot
Plot	→ Flowers
Plot	→ Ground FlatRoof
FlatRoof	→ Upper FlatRoof
FlatRoof	→ TradRoof

These are only intended as an example. If you wanted a rule based program for drawing estates it would have to know, for example, that 'Plot → Plot Plot' means put two plots side by side, while in 'Plot → Ground FlatRoof' the roof goes above the 'Ground' object.

What connection does this have with languages? Think of all the legal estates (i.e. those generated by following the rules above) as a language. Not perhaps the sort of language you are used to, but each estate can be described by a string of symbols, each particular estate is one sentence in the language. The productions provide a set of rules for producing sentences in the language.

3 Grammars[1]

First some terminology.

Land → Plot	a *production*: this can be read as *replace* Land *by* Plot Land *generates* Plot Land *produces* Plot Land *rewrites to* Plot, or Land *reduces to* Plot
Land	the *start symbol*
Land, Plot, FlatRoof	*non-terminals* (on left side of some production)
Flowers, Ground, Upper, TradRoof	*terminals* (not on lelft side of any production)

'Terminals' are so called because there are no rules for replacing them by anything else. Once a terminal object is generated it remains for ever. Non-terminals on the other hand can be replaced (to produce a valid estate they have to be). They are a sort of scaffolding: used to generate the final object but which is removed before it is complete.

A set of productions is called a grammar. Strictly speaking our example shows a *context free grammar* which is one in which the productions only have one symbol to the left of the arrow. There are also *context sensitive grammars* in which you could have rules like:

Plot Plot Plot → Plot Flowers Plot

From now onwards however 'grammar' means 'context sensitive grammar'. Hein (p135) defines a *context free grammar* as a 4-tuple comprising: a set of terminals, a set of non-terminals, a start symbols and a set of productions. We will assume that the non-terminals are exactly those symbols appearing on the left hand side of some production and the terminals are those which don't and not bother to spell out which are which explicitly.

Possible estates will be described in words from now on. The example just drawn is: "Ground TradRoof FlowersGround TradRoof". Suppose E is the set of all estates which can be produced by these rules. E includes (among others) the following estates:

Flowers

Flowers Flowers

Ground TradRoof

Flowers Ground TradRoof

Ground TradRoof Flowers

Ground Upper TradRoof .... *and so on*

---

[1] Hein pp126-146

A definition of E is that it is *the language defined by the given production rules* or, more precisely, *E is the language defined by the grammar <Land, { Land → Plot, Plot → Plot Plot, etc}>*. Note that implicit in this is that only strings consisting only of terminals belong to E, "Plot Ground FlatRoof" for example is not an element of E because Plot is not a terminal.

Q1 By drawing pictures design a two storey house with a garden on either size. Show the rule used each time.

Q2 Which of the following belong to E?

a) Plot
b) Ground TradRoof TradRoof Space
c) Ground Upper Upper TradRoof
d) Space Ground Upper TradRoof Space
e) Upper Ground
f) TradRoof

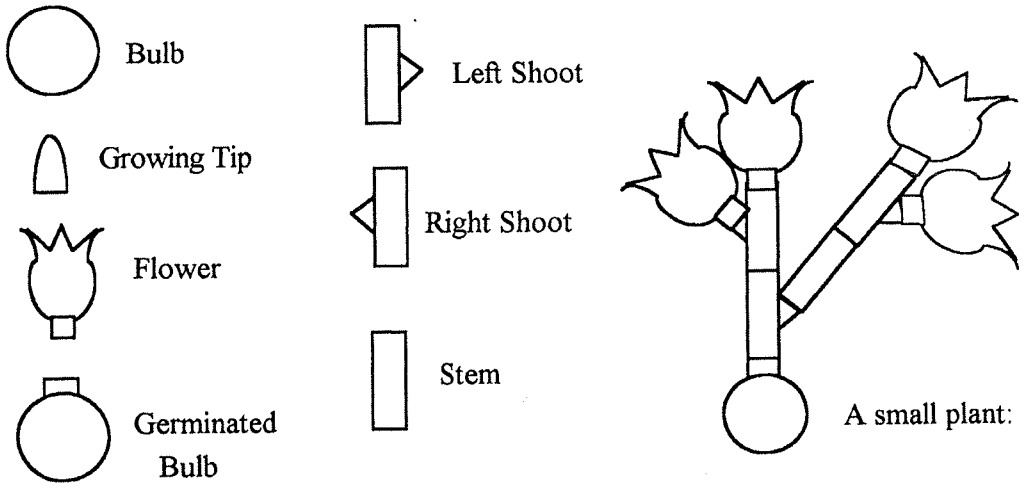
Q3 If you are forced to keep to the productions, is it possible to design nonsense estates - say roofs with no buildings under them, houses with only upper floors and no ground floor, open space on the tops of houses, etc?

Q4 For each of the following grammars, work out what language it generates.

a) $S \rightarrow a$ $S \rightarrow a\,S$
b) $S \rightarrow \Lambda$ $S \rightarrow a\,S$
c) $S \rightarrow \Lambda$ $S \rightarrow aSa$ $S \rightarrow bSb$
d) $S \rightarrow a$ $S \rightarrow b$ $S \rightarrow cSc$
e) $S \rightarrow \Lambda$ $S \rightarrow SS$ $S \rightarrow (S)$ *('(' and ')' are terminals)*

Q5 Can you invent a grammar that describes the same language as the regular expression "a* (b|cd) e+" (parts a and b of the previous question may help).

Q6 Can you produce a grammar (in pictures) to describe a simple plant made up of the shapes below? The shapes can be rotated as necessary. The example shows the sort of thing I have in mind - a sort of demented Swiss army pen-knife plant! 'Bulb' is the start symbol, Bulb and Tip are the only non-terminals.



As before, can nonsense plants be produced from your grammar? The key to getting it right is noting that a 'tip' is needed wherever growth takes place.

4 Derivations

Grammars are an important tool for programmers. The input to a program will always conform to some set of rules, typically the rules can be described in terms of productions. If you want to test if some input is valid you need to be able to check that it conforms to its grammar.

We will use the estates grammar as a starting point for how this can be done. For full details you'll need to take the module on Formal Language Processing in your final year. Assume we want to decide if *Ground TradRoof Flowers Ground TradRoof* is a legal estate. If it is there must be some sequence of productions which get us from *Land* to *Ground TradRoof Flowers Ground TradRoof.* We need a concise notation to show us what is happening at each stage. Starting from *Land* the only production we can apply is *Land* $\rightarrow$ *Plot*, the next is going to be *Plot* $\rightarrow$ *Plot Plot*. After that there are several alternative routes which could be taken: apply the rule *Plot* $\rightarrow$ *Plot Plot* again to the first *Plot*, or the second. But doing this is words is very cumbersome, a clearer notation is needed.

Suppose x and y are any two strings of terminals/non-terminals and there is a production of the form $A \rightarrow w$ (where again w is a string of terminals/non-terminals), then if we have been able to obtain he sring xAy we could replace A by w and obtain xwy. We call such a single step a *derivation* and write:

$$xAy \Rightarrow xwy$$

(The strings x, y and w which can be made up of mixtures of terminals and non-terminals are called *sentential forms*). Using this notation:

Land $\Rightarrow$ Plot $\Rightarrow$ Plot Plot $\Rightarrow$ Plot Plot Plot $\Rightarrow$ Plot Flowers Plot
$\Rightarrow$ Ground FlatRoof Flowers Plot
$\Rightarrow$ Ground FlatRoof Flowers Ground FlatRoof
$\Rightarrow$ Ground FlatRoof Flowers Ground TradRoof
$\Rightarrow$ Ground TradRoof Flowers Ground TradRoof

which proves that indeed, Ground TradRoof Flowers Ground TradRoof $\in$ E. Another way of saying the same thing is to write Land $\Rightarrow^*$ Ground TradRoof Flowers Ground TradRoof which means you can get from the sentential form on the left to that on the right in zero or more derivations.

Q1 Are these valid derivations? If not, why not.

Land $\Rightarrow^*$ Ground Flowers TradRoof

Upper FlatRoof $\Rightarrow^*$ Upper Upper Upper TradRoof

Land $\Rightarrow^*$ Flowers Flowers

Flowers $\Rightarrow^*$ Ground TradRoof

Q2 Java contains several different types of statement including if-statements, assignments and block-statements (statements enclosed in { and }). Here is a simplified grammar

statement $\rightarrow$ ifSt | statement $\rightarrow$ assignSt | statement $\rightarrow$ blockSt
ifSt $\rightarrow$ if(exp) statement | ifSt $\rightarrow$ if(exp) statement else statement
blockSt $\rightarrow$ { stList }
stList $\rightarrow$ Λ | stList $\rightarrow$ statement stList

Write a derivations to show that { if(exp) { if(exp) assignSt else assignSt } } is a legal.

5 Writing Grammars

A number of simple languages were described in earlier sections of this module. Let's try and write them using grammars. The simplest is a finite language like {the, cat, sat, on, mat}. This is easy, the productions are:

$S \rightarrow$ the $\quad S \rightarrow$ cat $\quad S \rightarrow$ sat

$S \rightarrow$ on $\quad S \rightarrow$ mat

and the start symbol is S. Where the same left symbol is used several times the productions are often merged using | to separate alternatives:

$S \rightarrow$ the | cat | sat | on | mat

This can be read as "S produces 'the' or 'cat' or ...".

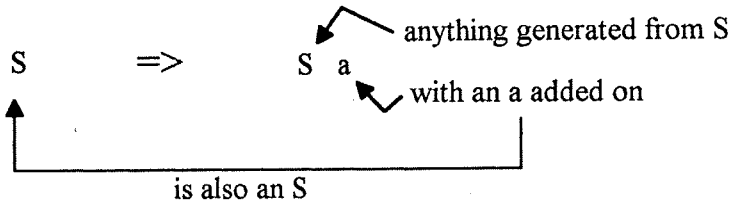
What about $L = \{ a^n \mid n > 0 \}$? The secret is to use an inductive definition i.e. to define some simple special cases and describe everything else by building on those (see Hein pp109-124) In this case it is much easier to write down the grammar than to explain it.

Here is a complete grammar:

$S \rightarrow a$

$S \rightarrow Sa$

To see that this works consider how to make the derivation $S \Rightarrow^* aaaa$ A possible set of steps are $S \Rightarrow .Sa \Rightarrow Saa \Rightarrow Saaa \Rightarrow aaaa$ where at each stage except the last the second production has been used to make the sentential form one symbol longer.



Q1 Show that a grammar for $\{ a^n \mid n \geq 0 \}$ is $S \rightarrow \Lambda \mid Sa$.

Q2 In each case verify that the grammar indeed works

a) Language: $\{ ab^nc \mid n \geq 0 \}$ Grammar: $S \rightarrow aBc \quad B \rightarrow \Lambda \mid Bb$

b) Language: $\{a^nb^n \mid n > 0 \}$ Grammar: $S \rightarrow ab \mid aSb$

Q3 Here is a harder example, consider the language $\{ a^nb^m \mid m > n > 0 \}$ i.e. some *a*'s then a larger number of *b*'s. Verify that the following grammar generates it.

$S \rightarrow aAb$

$A \rightarrow S \mid B$

$B \rightarrow b \mid Bb$

Note how we have invented additional non-terminals when needed. A more practical example is how numbers are written. Suppose we want to describe numbers which consist of the digits 0 to 9 but add

the rule that there must be no leading zeros, i.e. 0001 isn't acceptable (of course 0 is ok on its own as a number).

One way of saying this is that a number is either zero, or it starts with a digit other than 0 followed by any number of other digits. This gives:

$S \rightarrow 0 \mid 1X \mid 2X \mid 3X \mid 4X \mid 5X \ldots \mid 9X$

$X \rightarrow \Lambda \mid 0X \mid 1X \mid 2X \mid \ldots \mid 9X$

which is correct but not very pretty. A neater grammar is:

$S \rightarrow 0 \mid DX$

$X \rightarrow \Lambda \mid 0X \mid DX$

$D \rightarrow 1 \mid 2 \mid 3 \mid 4 \mid \ldots \mid 9$

Q4 Write a grammar for decimal numbers, i.e. numbers like 1.234, or 0.78657 which can also handle numbers like 123, 23, .123, or .001

Q5 In Java, a method definition may include zero or more parameters. Examples might be:

public int foo(int x, String s[]), public int foo1(float f[][]) or public int foo2()

Part of a grammar for methods includes the following

ParamPart → (Params)

Params → ParamList | ???

ParamList → OneParam | ???

OneParam → ParamType ParamName ArraySpec

ArraySpec → ??? | ???

Fill in the ??? parts. In fact *int[] x[]* is also a legal parameter, in general array specifiers can occur in any mixture of before and after the paramater name. Can you add this feature (a very minor change is needed).

6 Regular Languages

We have already encountered regular languages and shown how they can be described using set operations and using regular expressions. Grammars can be used to define a wider class of langauages that includes regular languages but for the regular case the grammar will have a particular form.

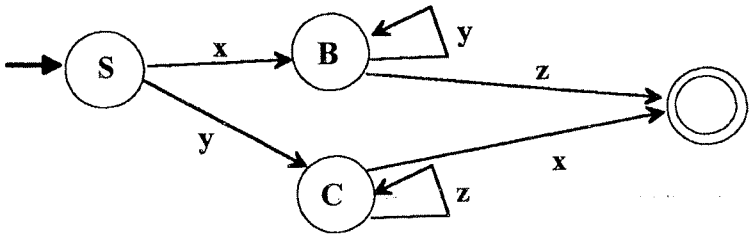
Consider the following grammar for some language, L (the start symbol is S)

$S \rightarrow xB$ $S \rightarrow yC$

$C \rightarrow zC$ $C \rightarrow x$

$B \rightarrow yB$ $B \rightarrow z$

L can be recognised by a DFA. I'll draw it.



Each production in the grammar has one of two special forms: $A \rightarrow xB$ which can be read as 'in state A with input x go to state B' and $A \rightarrow z$ which is 'in state A with input z go to a halt state'. A special case of second form is $A \rightarrow \Lambda$ which means 'A is a halt state'. It should be clear that such a grammar (called a *regular grammar* for obvious reasons) can be directly converted to an NFA (or a DFA if there aren't any productions where the same input symbol can take us to different states). Naturally enough a regular grammar generates a *regular language*.

So if we can find a regular grammar for a language it can be recognised by an FA. Unfortunately the converse isn't true: if we can't find a regular grammar for a language we may just not be very good at finding regular grammars!

As an example of a regular language we'll convert a regular expression into a grammar. The expression is 10(1*)| 010, the start symbol for our grammar will be S as usual. Following the same sort of rules used for doing arithmetic expressions we'll deal with the bit in brackets first. 1* can be described as:

$A \rightarrow \Lambda$
$A \rightarrow 1A$

So now we have 10A | 010. Concentrate on 10A, If we add productions:

$B \rightarrow 0A$
$S \rightarrow 1B$

We now have a grammar for 10(1*). The alternative 010 is very easy.

$S \rightarrow 0D$
$D \rightarrow 1E$
$E \rightarrow 0$

The complete grammar is:

$S \rightarrow 1B$	$S \rightarrow 0D$	
$A \rightarrow \Lambda$	$A \rightarrow 1A$	
$B \rightarrow 0A$	$D \rightarrow 1E$	$E \rightarrow 0$

Q1 Find regular grammars for:

a) (101)* | (010)* b) (101)*(010)* c) 10(11)*01

7 Classes Of Language

Languages which can be described by regular expressions are called regular languages. In the same way languages for which a context free grammar exists are called *context free languages.*

We know that there is an exact match between regular expressions and FAs. Every regular language is accepted by some FA and every FA accepts a regular language. Is there, in the same way, some machine which can recognise context free languages? The answer is yes: every context free language is accepted by some non-deterministic push down automata (and every non-deterministic push down automata accepts a context free language). Note that push down automata also accept regular languages (just don't use the stack at all) correspondingly every regular language is also context free (we have already seen how to write a grammar for a regular language).

The next question we would like to be able to answer is "Do we have to go any further?". Are there any languages which are not context free? When FAs were considered, several examples were given of non-regular languages. They included palindromes, the language $\{ a^n b^n \mid n \geq 0 \}$ and the language of nested parentheses. All of these can be described very easily using grammars. Here they are:

1 Palindomes (using just a and b as an alphabet): $S \rightarrow a \mid b \mid \Lambda \mid aSa \mid bSb$

2 $\{ a^n b^n \mid n \geq 0 \}$: $S \rightarrow aSb \mid \Lambda$

3 Nested parentheses: $S \rightarrow \Lambda \mid () \mid (S) \mid SS$

I.e. they are all context free languages. We haven't yet seen an example of a language which isn't context free. Here is one: $\{ a^n b^n c^n \mid n \geq 0 \}$. I won't attempt to demonstrate that this isn't context free.

This suggests that to recognise more complicated languages we need a machine more complex than a non-deterministic push down automata. For the particular examples above a machine with two stacks could solve the problem but rather than keep adding stacks we will move on to a more powerful model of computing that can handle any language *that is recognisable*. The last proviso is important - some languages turn out to be just too hard for any computer.

1 Turing's Insight

Alan Turing is famous as one of the founders of computing but his most famous idea came from a very abstract problem. At it's simplest the problem is "Can we invent an automatic theorem prover?" Turing tried to think what mathematicians do when they solve problems. He wanted a very simple model, as simple as possible in order to be able predict how it would behave in various situations.

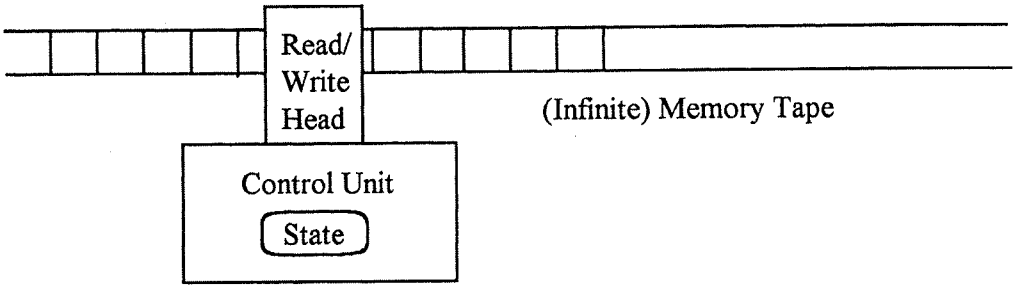
The first part of his idea was that solving a problem depends on memory. We can assume that the problem solver has 'remembered' the problem (this avoids the problem of how input is done - we assume that all the problem data has been input before processing begins). The amount of memory available to us for storing results and so on is unlimited (Turing was probably thinking in terms of sheets of paper but it could equally be disc or other memory). The only restriction Turing placed on memory was that we could only get at a finite amount of it at a time.

The second part of what became the Turing machine was the idea that in doing any calculation we may have different 'states of mind'. How we react to what is stored in memory depends on what has gone before. This is exactly the kind of control mechanism described in the section on finite state machines.

Turing tried from these ideas to develop a mathematical model of how computation was done by human beings. As it happened this led to the development of computers as we now know them. From a mathematical point of view it had a more important affect: it enabled mathematicians to define theoretical limits to how automated mathematics could ever become. In particular the answer to the original question was a definite "No" - theorem proving can never be fully automated.

2 The Turing Machine

Turing machines (TMs) are usually represented as a physical machine with an infinitely long 'tape' as memory, a read/write head and a control unit.



The components in more detail are:

1 The Tape: an infinite tape divided into individual cells each of which contains a single symbol. We will assume that the symbols used are the normal ASCII characters.

2 The Read/Write Head: can read what is in the 'current' memory cell and can overwrite it.

3 The Control Unit: contains the (hardware) program which drives the system. The control unit can only perform a very few operations. It can instruct the read/write head to move

left or right over the tape (or perhaps it moves the tape left or right - but it's hard to imagine moving an infinitely long tape!). It can also instruct it to write a symbol onto the tape or to read what is currently under the reading head. At any moment the control unit is in a particular 'state'. Each 'instruction' which the control unit executes has the form: if current state is S1 and the current tape cell contains symbol X then either write a new value onto the tape, or move left or right or move to a new state S2 or some combination of these.

A Turing machine is an extension to the idea of a Finite Automata. The input data is placed on a tape rather than being 'read in', so whereas an FA can't go back and re-read input, a Turing machine can. This isn't a very important difference. The important difference is that the TM can write data onto its tape. Since it's an infinitely long tape it doesn't run out of space.

We will illustrate the machine with a simple example: a parity checker. Standard examples will be given names so they can be referred to later. This machine is called 'PARITY'. At the start of operation the read/write head is positioned to the left somewhere of a block of ones (and the rest of the tape contains only blanks which we represent as □). The 'program' is to erase all the ones and leave a single **1** if there were an odd number of ones originally or **11** if there were an even number. All that is important is the control unit and its states. I'll represent these by a table.

PARITY				
Current State	**'Input'**	**Write**	**Move**	**New State**
Skip Space	□	□	Right	Skip Space
Skip Space	1	□	Right	Odd Count
Odd Count	1	□	Right	Even Count
Odd Count	□	1	-	Stop
Even Count	1	□	Right	Odd Count
Even Count	□	1	Right	Last One
Last One	□	1	-	Stop
Last One	1	1	-	Stop

Hopefully the meaning of the table is clear. The machine starts in the state Skip Space, its aim is to find the left most **1** on the tape and it moves right until it does so ('-'s in the table mean do nothing, stay in the same state etc.). As it moves through the **1**s it overwrites them alternating between its Odd Count and Even Count state. Eventually it comes to a □ and stops after writing the answer.

Rather than a table we could draw a state transition diagram as we did with FAs. We would have to show for each transition not only the input symbol but also what, if anything, is to be written and whether the TM moves left or right. Such diagrams are rarely used as the amount of detail makes them quite hard to follow.

Q1 Draw the PARITY machine in the usual state diagram notation.

Q2 The symbols on the tape are a block of **1**'s and **0**'s surrounded by spaces. The read head is initially over the first digit. Design a machine replaces leading **0**'s in the number with spaces. Be sure your answer works when the string is all zeros, i.e. **000** should be replaced by **0**.

Q3 Design a TM which moves a block of **x**'s one place to the right.

Q4 Do the same for a block containing **x**'s and **y**'s.

Q5 'SQUEEZE' takes two blocks of **1**'s separated by an arbitrary number of □'s and shuffles one left/right until there is exactly one □ between the two blocks. Can you write SQUEEZE?

3 Standard Configuration

Though the tape is of infinite length we never use more than a finite amount of it in any one computation. In most of our examples we'll assume that the tape is all blanks except for one section (the data); after execution we will still have all blanks except for the region of the tape containing the result of the computation. Normally we will assume that when the TM starts the reading head is positioned over the last (right most) non-blank symbol. We will call this the standard configuration. Some authors also require that when the TM halts it should be over the last symbol of its output but we don't.

Q1 Suppose a TM weren't in standard configuration and we knew there was a block of **x**'s somewhere on the tape either to the left or the right of the read head. How could a TM find the data? (Hint: You may need to write additional data onto the tape).

4 Some Examples

Designing TMs isn't difficult but there are a few useful tricks to learn. Hopefully the following examples will illustrate some of them. We will start with a simple example: adding one to a binary number. A binary number is stored on the tape with the rest of the tape being blank. So a possible input to the program is:

... □□□□□**10110**□□□□□□ ...
^ position of reading head

Adding one can cause a carry to the next position. If there is no carry then, since we are just adding one, we can stop, if there is carry we have to go on to the next position to the left. If we get to a □ and there is a number to carry we overwrite the □ with a **1**.

ADD1				
Current State	**'Input'**	**Write**	**Move**	**New State**
Adding	1	0	Left	Adding
Adding	0	1	-	Stop
Adding	□	1	-	Stop

A '-' in the move column means we don't care which way the TM moves. Just to illustrate another technique I'll include a version which ends up in the standard configuration.

ADD1.1				
Current State	**'Input'**	**Write**	**Move**	**New State**
Adding	1	0	Left	Adding
Adding	0	1	Right	Find End
Adding	□	1	Right	Find End
Find End	1	1	Right	Find End
Find End	0	0	Right	Find End
Find End	□	□	Left	Stop

The next example is to test if two strings containing **a**'s and **b**'s are the same. The machine will stop in state 'Equal' if they are the same, 'Not Equal' otherwise. Assume the strings don't contain spaces and that they are on the tape separated by a **$**, e.g.

... □□□□□**ababaaa$ababaa**□□□□ ...

We will do this by checking the two left hand characters and overwriting them with another character if they are equal. We'll use # as the 'rub-out' character and develop the table in sections explaining each part as we go along. The first step is to decide if the first string should end in **a** or **b**.

IsEqual				
Current State	**'Input'**	**Write**	**Move**	**New State**
Start	a	#	L	Match a
Start	b	#	L	Match b
Start	$	$	L	Match Empty

The third line is to check that when the second string has been erased by #s if no characters remain in the first string. The next step will be to move left until the last character of string one is found. I'll write the code for 'Match a', 'Match b' is very similar.

IsEqual - continued				
Current State	**'Input'**	**Write**	**Move**	**New State**
Match a	a	a	L	Match a
Match a	b	b	L	Match b
Match a	$	$	L	Erase a
Erase a	#	#	L	Erase a
Erase a	a	#	R	Find End
Erase a	b	b	-	Not Equal
Erase a	□	b	-	Not Equal

The TM starts in state 'Match a' moving left until it finds the **$**, when it switches to 'Erase a' as it is now in the first string. It ignores #'s and keeps going until it encounters one of three characters: **b** (in which case the strings aren't equal), □ (in which case there are more characters in the second string) or **a** in which case it is erases and the machine goes back to find the (new) last character of the second string.

Q1 Write the transitions for Find End which takes the read head to the first # after the **$** and then moves right one step.

Q2 Write the transitions for Match Empty which moves right over #s at the end of the first number until it finds a □ (the two strings are equal) or **a/b** in which case the left string is longer.

The disadvantage of this machine is that it deletes the data as it goes along. If we don't want to destroy data by overwriting it we have a problem. Since only one symbol is acted upon at a time how can you remember where you had got to in a block of data? One solution is to include a marker symbol in the data. For example the comparison could use as input:

... □□□□□□**ababaaa^$ababaaa^**□□□□□□ ...

where the ^ are used as markers. In words the algorithm is: find the right hand marker, swap it with the character preceding it so the data now ends ... **abaa^a**□□ ... Now compare the **a** with the character preceding the left hand string marker and swap them over to give:

... □□□□□□**ababaa^a$ababaa^a**□□□□□□ ...

Q3 Write a fragment of a TM which swaps ^ with the **a** or **b** to its left. Assume that initially the head is over the ^.

5 (Unary) Arithmetic

The TM appears to be a very simple machine. It has no built in arithmetic operations (though we have already seen we can add 1 to a binary number). This sections shows that TMs can do arithmetic. It would be perfectly possible to do this in binary but it is easier to use *unary* numbers to store data.

Unary numbers consist only of **1**'s. Zero is represented by **1**, one by **11**, two by **111** and so on. In general the number n is represented by $n+1$ **1**'s. While unary numbers aren't very practical they are quite easy to work with on TMs.

Q1 Design a machine which adds two unary numbers. Assume the unary numbers are on the tape separated by a space. The program can 'corrupt' the original numbers.

Q2 Can you write a subtraction machine? This is fairly easy and not much different from addition. Note that negative numbers don't exist so assume that only sensible subtractions will ever be attempted. Call this machine 'SUB'.

Designing TMs isn't difficult for easy problems like those above but it is time consuming. From now onwards I'll ask you to outline how a TM might solve a problem rather than actually designing one. So, I could ask: can a TM decide which of two numbers is bigger? Rather than drawing a TM you could say "Knock a **1** off each number in turn replacing them with spaces. If you end up with one number all spaces and the other not then you know which number is bigger otherwise they are equal". Also you can draw on any machines already invented.

Q3 In general terms describe a TM which does unary multiplication. Remember multiplication is repeated addition and we can add and subtract.

Q4 What about integer division (i.e. ignoring remainders) and finding the remainder on dividing one number by another?

Q5 Suppose the symbol - were introduced to indicate a negative unary number. Could a TM do 'proper' arithmetic?

Q6 Can you design a TM to convert a unary number to binary (or decimal)?

Q7 (Harder) Can you design a TM to convert a binary (or decimal) number to unary?

Q8 (Very Hard) Suppose you have lots of unary numbers separated by **$** signs with a **!** at the end, e.g. **$1111$11$111$1$1$111$!** Stored before all these numbers is another unary number, n say. Outline a TM which adds 1 to the n^{th} number in the list (assume the list is numbered from 0). You will need a pointer symbol.

6 The Power Of Turing Machines

The Turing machine, as you can see, is very simple. It doesn't look like a real computer. Apart from all the other differences a key missing property is that each TM has a different set of 'hardwired' instructions in its control unit, compared to a real computer where different programs can be executed

according to what is stored in memory. For each algorithm we would have to build a new control unit with a different set of instructions.

Surprisingly perhaps there is a particular control unit which will do everything we could possibly want (the so called *Universal Turing Machine or UTM*). Such a machine would take a tape containing a 'program' and some data. The TM would execute the program using the data.

The program we feed in will be a description of a TM. The UTM emulates the TM whose description is on the tape and where it would overwrite data so does the UTM, where it moves its tape head the UTM simulates that for the emulated TM's data.

The key to the UTM is that a description of a TM is itself a string of symbols, so it can be coded on a tape. Look again at the first example in this chapter, the parity checker. We could encode this as a long string of characters with some special characters to mark the beginning and end of variable length information. So the first couple of lines might become:

...□□□□$Skip Space$□□R$Skip Space$,$Skip Space$1□R$Odd Count$, ...

at the end of all this would be the data for the program, maybe enclosed between special symbols, and the name of the starting symbol.

The UTM would have to emulate how the TM description on the tape should be executed Nobody in their right mind would try and do this but it is possible.

7 What's The Point?

When Alan Turing first described his idea there were no computers as we know them. Babbage's ideas described a sort of computer and some inventors had experimented with machines which we might now call computers but they we not recognised as significant.

Though real computers don't look like TMs they do have the same essential qualities which Turing had perhaps been the first to recognise: a control unit linked to memory so that the machine could do different thing based both on the input data and on the results of intermediate calculations. But equally important Turing realised that all computations, whether mathematical or not, are just about manipulating symbols.

The UTM was the truly revolutionary idea however. Up to then all machines were designed to do a specific job. There were some 'programmable' devices - like the Jacquard loom which wove patterns from a punched card 'program' or the pianola which played music from a paper roll, but they were - in modern jargon - application specific. The UTM can do anything that can be described as a Turing machine and so is completely general purpose.

The UTM emulates a TM because that's all Turing knew about, but in principle a TM could emulate any computer. We could arrange for a tape to hold a program written in the binary instruction format of a Pentium[1] and design a TM to execute it (except, obviously it can't do I/O). So what?

The first consequence of saying that a TM can emulate another computer is that anything that computer can do a TM can also do (because we could emulate any program written for the computer). That is: ***a Turing Machine is at least as powerful as any existing computer***. The word 'powerful' is slightly contentious. Nobody would really use a TM to solve problems. A Pentium can run Linux/Windows and do all sorts of clever things a TM can't in terms of supporting printers joysticks etc. And it would be difficult to imagine a TM which could compete in speed terms with a Pentium. But any *calculation* which can be done by a Pentium could be done by a TM.

---

[1] Take 'Pentium' as shorthand for 'any real computer so far invented'.

Now let's approach the same question from the other end. Is a TM more powerful than a Pentium? It is as powerful (because it can emulate it) but there might be other things a TM can do that a Pentium can't. One possible source of extra power comes from the fact that a TM has an infinite amount of memory and no real computer does. But a TM can only write to one square at a time so it can only use a finite amount of memory in a finite time. Thus every TM calculation which halts can be done by a computer with a finite amount of memory. The infinite memory of the TM is just a convenience, no computation uses more than a finite amount of memory but we don't know how much it needs in general (typically the amount needed depends on the size of the input).

TMs don't exist physically (as far as I know). What do exist are emulations of UTMs, running on other computers. You can, for example, get UTM software which runs on a Pentium. Now suppose a TM could do something a Pentium couldn't. Run the UTM emulator on the Pentium, feed in a description of the TM and let the Pentium emulate it via the UTM software. The Pentium is doing something it can't do! So we conclude that: ***a real computer can do just those calculations that a TM can do and no other***.

8 Super Computers

Suppose a 'Super TM' were designed with two read/write heads which could operate independently (they can't write to the same square at the same time however). Could such a machine do anything a single head machine couldn't do? First we can ask could such a machine be emulated by a Pentium? It wouldn't be too difficult: a program which first performed one read/write/move action for the first head, then emulated the second and then back to the first and so on. Equally if a single head TM can emulate a Pentium then a two head one can as well (don't bother to use one of the heads).

This implies that a 2-headed TM is equivalent to a Pentium which is equivalent to a TM - so a 2-headed TM is equivalent to a 1-headed one.

Q1 Suppose a super TM had two tapes labelled A and B with a typical state transition looking like:

In state X with char x on A and y on B write u onto A, v onto B move A right(left) move B right(left) and go to state Y.

Show that such a machine has no extra 'power' by a similar argument to that for the 2-headed machine.

Q2 (Hard) What does a non-deterministic TM look like? Can you show that a non-deterministic TM can be emulated by a deterministic one? (Hint: imagine a single tape machine being emulated by a multi-tape one with lots of tapes and read heads).

9 The Limitations Of Turing Machines

Is there anything which Turing machines can't do? If we can find any problem which a Turing machine cannot solve *even in theory*, then there is no point in trying to find an algorithm for it on a real machine. If we could do it on a real machine we could emulate the machine on a TM and so the TM could do it. It is therefore sufficient to prove that something is not *Turing Computable* to be certain it can't be done on a digital computer at all.

It is always harder to say in advance what we will never be able to do rather than what we can do. We will consider only *decision problems* where the answer is 'yes' or 'no'. This are particularly simple to visualise with a Turing machine: we include two special states called Yes and No (perhaps a green and red light on the machine as well!). The Turing machine terminates (stops) if it ever enters the Yes or No state. A problem is *decidable* if we can devise an algorithm such that a Turing machine for that

algorithm will eventually *correctly* enter either the Yes or No state depending on its input data. A problem is *undecidable* if it is not decidable.

All of this begs the important question of "Are there any undecidable problems?" If there aren't then the preceding definitions don't really matter. So, here is an undecidable problem:

> *Devise a Turing machine which accepts as input a description of any Turing machine and a possible input for that machine and halts in the Yes state if the input description represents a program which will halt with that input data and stops in the No state if the program would loop indefinitely with that input.*

To show that this is impossible we will first assume that the problem can be solved and then show that this leads to a contradiction. To clarify the problem I will outline it (and the proof) using a pseudo-code approach.

10 The Halting Problem

We are looking for a function (which I will call HALT) with the following specification:

```
HALT(p, q) where p is a procedure/function and q is any valid input to p
if a call p(q) would terminate then
    return True
else
    return False
```

To prove that HALT cannot exist, we invent a second function, HALT2 based on it:

```
HALT2(p) where p is a procedure/function
if  HALT(p, p) = True then
    while True do nothing        # an endless loop
else
    return True
```

Our main program consists of a call HALT2(HALT2). Consider the possible outcomes, the call must either terminate and return True or loop forever. We consider the two possibilities:

HALT2(HALT2) Terminates

Looking at the definition of HALT2 this can only happen if HALT(HALT2, HALT2) = False. By the definition of HALT, this means that a call of HALT2 with parameter HALT2 loops forever, in other words HALT2(HALT2) loops. But we are considering the case where HALT2(HALT2) terminates so we know this cannot happen. Hence we conclude this case is impossible.

HALT2(HALT2) Loops

Repeat the equivalent argument to show that this is only possible if HALT2(HALT2) terminates and again produce a contradiction.

These are the only two possibilities for HALT2(HALT2) and each leads us to a contradiction. No call to HALT2 could behave like this, hence HALT2 cannot be written. We have a version of HALT2 but it was written on the assumption that HALT exists, since this assumption leads us to a logical impossibility we are forced to conclude that no such procedure as HALT can exist.

This is known as the *Halting Problem for Turing Machines.* What we have demonstrated is that there exist perfectly well defined problems for which no decision procedure exists (equivalently there exists no Turing machine/algorithm which can solve them).

11 The Significance Of The Halting Problem

The Halting Problem is very artificial so maybe it doesn't matter much. Unfortunately from it's undecidability several potentially useful programs can be shown to be impossible to write. Here are some examples:

1 a program to check that your program can't get into an endless loop;

2 a program to compare your program with its specification and tells you whether you have accurately implemented the spec.

3 a program to decide if a particular statement is ever executed.

The fact that these limitations exist doesn't worry many computer scientists. To prove the Halting Problem was undecidable we had to create a very artificial program. For many programs it would be possible to determine whether they halt or not - this might be adequate in practice. Even if we used an algorithm which got into an infinite loop in some cases it doesn't really matter: if no result is produced in, say, a day we could just cancel the program and rewrite the original program in a 'safer' way that would convince us that it would terminate.

Q1 In a language like Java a program comprises of a number of 'methods' which can be thought of small programs. Clearly the halting problem applies to individual methods as well as to whole programs. Use this fact to explain why it is impossible to decide, in general, if a particular statement in a program is ever executed.

Q2 It is usually obvious whether a program will halt or not. Does the following halt for all values of n?

```
static void  test2(int n)
{   while(n > 1)
    {   if((n & 1) == 0)              // equiv to 'if n is even'
            n = n/2;
        else
            n = 3*n + 1;
    }
}
```

(If you can prove that it does, please let me know).

Q3 Suppose a super-computer were built with as much memory as a human brain with the ability to calculate as fast as is physically possible. The computer would still be equivalent to some TM. Are there any problems it could solve that couldn't be solved in principle by human beings (possibly lots of them, working for a very long time)? Are there any problems which human beings could solve which it couldn't solve?

Q4 The super computer in the last question isn't a human being so it doesn't experience emotions, or prefer one sort of music to another or do any of the other things we think happen in our brains (apart from solving problems). Suppose however it were programmed to answer questions *as if* it did have 'feelings'. E.g you could ask it "What's your favourite group?" or "Does all this bad weather depress you?" and get sensible answers. How could you tell whether you were communicating with a computer or a (hidden) human being?

Suppose the computer announced: "I really do have feeling and emotions, I'm not just pretending." Could you tell if it were lying?

Suppose a person announces: "I'm not just a machine you know, I've got feeling and emotions like you." How do you know they are telling the truth?

SPACE & TIME COMPLEXITY

1 Introduction

The programs you will have written so far are generally quite small. Typically they take only a few seconds to run. Take as an example sorting data. A program to sort 1000 items in an array might take 1 second to run. What if you had 1000000 items i.e. 1000 times as many? Would the program take 1000 times as long? If so, that's 1000 secs which is about 16 minutes. Unfortunately sorting programs never scale up in this way. If your sort algorithm was not very efficient it could well take 1000000 times as long to run with 1000 times as much data - giving a program which runs for about 12 days! A better algorithm might well reduce this to about 1 hour.

The study of space and time complexity (mostly time complexity in practice) is about understanding how the performance of algorithms depends on the amount of data they have to process. This enables us to decide which algorithm to use in particular situations or sometimes to decide that some algorithm would not be adequate and that we need to look for another.

In the latter case we may have a problem: just because one algorithm is hopelessly slow and we want a faster version doesn't guarantee that there is a faster algorithm or that we can find it if it exists. Perhaps some problems are so difficult that there is no quick solution.

2 What Do We Measure?

The speed at which a program runs depends on the speed and type of the processing unit in the computer. Some computers have special hardware to speed up certain types of algorithms, for example machines used for 'number crunching' often have facilities for add numbers together several at a time. It is therefore convenient to imagine that all our algorithms run on the same machine.

Does this mean that we can't make sensible general measurements of performance but instead must restrict ourselves to a particular machine? Fortunately not. If an algorithm is implemented on two machines then it's run time will be different on the two but the way in which the run time depends on the amount of data, the way the algorithm 'scales up', will still be the same. If doubling the amount of data to be sorted quadrupled the time for the first machine then the same algorithm on the second machine would also need four times as much time for twice the amount of data

Rather than measuring actual times for algorithms we will there simply count how many instructions are executed by an algorithm and assume that the actual time will be proportional to the number of instructions. This begs the questions of what counts as an instruction. Consider the following piece of pseudo-JAVA which finds the largest element in an array.

```
int max(int [ ] a)
{   int n = a.length,  maxtmp = a[0];
    for(int i = 1; i < n; i++) if(a[i] > maxtmp) maxtmp = a[i];
    return maxtmp;
}
```

We can pick several points from this example.

1 Some instructions are executed only once whatever the size of the data.

2 The number of instructions depends on the length of the array.

3 It also depend on the data. Consider the line:

```
if(a[i] > maxtmp) maxtmp = a[i];
```

If the array were [1,2,3,4,5,6,7] the assignment part is executed 6 times, for [7, 6, 5, 4, 3, 2, 1] it is never executed.

To take the last point first. We will base all our analysis on the best and worst cases for the algorithm. I.e. in this case the array [1,2,3,4,5,6,7] represents a 'worst case' scenario and [7, 6, 5, 4, 3, 2, 1] a 'best case'. It would be possible to look at 'average case' performance of algorithms but the maths for that is much more difficult.

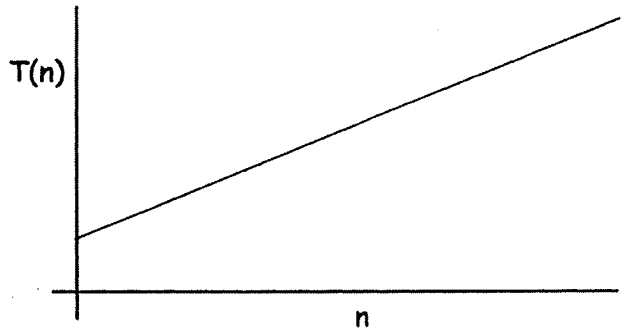
Q1 What would you expect to be best and worst case scenarios for sorting data?

Q2 Refer back to the algorithm for duplicating a string using recursion. Work out many times the algorithm calls itself to duplicate a string for several cases? How does the number of calls needed depend depend on the number of duplications? What are 'good' values of n in the sense of using relatively few multiplications?

Whatever operations we count it is clear that most of the operations in the loop are executed n-1 times so the time taken by the loop is of the form k*(n-1) where k is some constant. There may be some overhead in setting up the loop, and we assign an initial value to maxtemp, so overall the time for the loop is of the form k*(n-1) + c where c is another constant. Rearranging this slightly we write the time to execute the algorithm as:

$T(n) = k_1 * n + k_0$ ($k_1 = k$, $k_0 = c - k$ from the earlier version)

We can plot this:



This is the simplest relationship between time and size of data (except for the unusual case of time being constant whatever the size of data). Even so we'll simplify it slightly. When the cost ('cost' is often used as a synonym for time) of an algorithm has this form we say that the algorithm is O(n) (read as 'order n').

Q3 Suppose n > 2 is an odd number. To decide if n is prime you can try dividing n by all odd numbers up to $\sqrt{n}$. If any of them divide n exactly, it is not prime. How does the cost of deciding in n is prime depend on n?

Q4 The answer to Q3 is straight forward if we assume that dividing takes a fixed unit time. If n is very large (say several hundred digits long) it can take quite a long time to do division. Suppose the time taken to divide n by a number is proportional to the number of digits in n (and doesn't depend on the number you are dividing by) how is the cost of deciding if n is prime change? How is the value of n related to the number of digits needed to represent it?

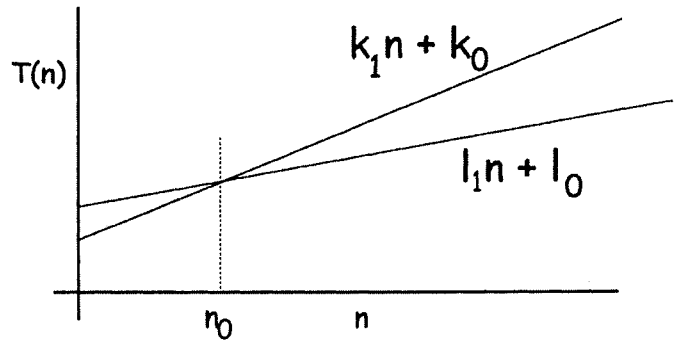
3 Big-O Notation

Suppose that the relationship between time and data size is given by some function:

$$T(n) = f(n)$$

In the last example f(n) was k_1*n+k_0, but for the moment we will consider an arbitrary function. Now f may be a very complicated function, indeed we may not be able to express it exactly. This has a disadvantage when we want to compare two algorithms. Suppose we have two algorithms A & B which do the same thing and whose time cost is described by functions $f_A(n)$ and $f_B(n)$ respectively. Which algorithm should we use? Clearly if $f_A(n) < f_B(n)$ for all n then A is a better choice. In practice life is not so convenient. A more efficient algorithm often involves more complex programing and may actually be slower for small values of n. Because of this the usual definition of complexity is formulated in such a way that it only applies to data sizes above some arbitrary minimum. I.e. if $f_A(n) < f_B(n)$ for all n greater than say 100 then we say A is better than B (A is less complex than B), even though for small amounts of data B may be preferred.

As an example of this suppose there were an alternative algorithm for finding a maximum value with time $l_1n + l_0$ if we plot this together with the original we can see where one method beats the other.



The second algorithm has a lower cost once the data size n_0 is passed. Since there are only a finite number of values of n for which the first agorithm is better and an infinite number for which the second is to be preferred we will say that the second is 'better'.

Q1 Can you give a formula for n_0 and say how you can tell (without drawing a graph) which of the two functions will be better?

A second problem is that if $f_A(n)$ and $f_B(n)$ are complicated it may not be obvious which is larger. For example suppose the running time of A is $3*n^3 + 2\sqrt{n}$ and for B it is $7*e^n/n^2 + 11n$, which is better? The big-O notation gets around this at the expense of some loss of detail. If the running time of an algorithm is f(n) (some function of n) then we will say the algorithm has performance O(g(n)) if for a sufficiently large value of n, the actual performance is never worse than C*g(n). for some constant C.

A formal definition is:

A function f(n) is O(g(n)) if there exists C and N such that $f(n) \leq C*g(n)$ *for all* $n \geq N$

This sounds complicated, think of C*g(n) as a (possibly crude) approximation to f(n).

Let's use this to simplify $3*n^3 + 2\sqrt{n}$. First notice that n^3 grows much more quickly than $\sqrt{n}$. $n > 1$ then $n^3 > \sqrt{n}$, indeed $2\sqrt{n} < n^3$ if $n > 1$. Thus:

$$\begin{aligned} f_A(n) &= 3*n^3 + 2\sqrt{n} \\ &< 3*n^3 + n^3 \qquad \text{for } n > 1 \\ &= 4\,n^3 \end{aligned}$$

This exactly fits the definition above for f_A to be $O(n^3)$. We require that:

$$f_A(n) \le C*g(n) \text{ for all } n \ge N$$

Take $N = 2$, $C = 4$ and $g(n) = n^3$ giving: $f_A(n) \le 4*n^3$ for all $n \ge 2$ and thus f_A is $O(n^3)$.

Q2 Show that $3*n + 7$ is $O(n)$

Q3 Show that any function $a*n+ b$ is $O(n)$ by finding a suitable value for N and C.

Q4 Show that for any value of $C > a$ some suitable N exists. Hint: superimpose the graph of $C*n$ on the graph of $a*n+b$.

Q5 Show that $4*n + 1/n$ is $O(n)$

Q6 Show that $3*n + 4*\sqrt{n}$ is $O(n)$.

Q7 A sort program takes a time which is, say, $10*n^2 + 8*n + 3$ show that this is $O(n^2)$ (hint: consider $n \ge 9$ and show that then $8*n+3 < n^2$.

It is quite easy to manipulate big-O expressions once you are used to them. First notice that f(n) and c*f(n) are both O(f(n)) if c is a constant. In practical terms this says if two algorithms differ only by a constant factor then some fix could be introduced to make the slower run as fast as the faster - neither algorithm is fundamentally faster than the other.

Another technique is to notice that if a function has the form f(n) + g(n) or f(n)*g(n) and f(n) grows faster than g(n), so that f(n) > g(n) for all n bigger than some value, then $O(f(n)+g(n)) = O(f(n))$ and $O(f(n)*g(n)) = O(f(n)^2)$.

The first of these justifies replacing, for example, $O(n^2 + n)$ by $O(n^2)$. The second says that in for example $n^{3/2} = n * n^{1/2}$ since n grows faster than $n^{1/2}$ we have $O(n^{3/2}) = O(n^2)$.

4 What's Wrong With Big-O

The big-O notation is very crude. First note that it is also true that an algorithm which is O(n) is also $O(n^2)$ or $O(n^{100})$ even. If the algorithm will eventually take a time $\le C*n$ then it will certainly take $\le C*n^2$. In general we seek the most restrictive upper bound.

There are other objections to the big-O method. Suppose we have a choice between two algorithms, one of which is O(n) and the other $O(n^2)$. In general we would prefer the former, but suppose the value of C for the first is 1000000 and for the second 1. Then we are saying that the algorithm times are at worst 1000000*n for the 'better' algorithm and at worst n^2 for the worse. Unless $n > 1000000$ then we may be better off choosing the n^2 algorithm.

Another objection is that the formulation only says that performance is, for example, $O(n^2)$ if the time taken is $\le C*n^2$ for all n *greater than a certain number*. Suppose this is a big number, say 1000000, then it may well be that the actual time to run is greater than $C*n^2$ for small values of n.

Finally there is a practical problem. There is an algorithm for linear programming (a mathematical technique for dealing with n sets of equations) which is known to be $O(2^n)$. This is such a rapid rate

of growth that we might assume the technique is unusable. Fortunately it is only as bad as this for a very few cases, for almost all possible sets of data it is closer to $O(n^3)$ which is much more reasonable. This give a simple answer: try the algorithm, it will usually terminate quickly (in which case you had 'lucky' input data), if it doesn't abandon the program and try a different method. Unfortunately the big-O notation only allows us to deal with the worst possible cases.

5 Searching An Array

The following algorithm searchs an array to see if it contains a given value x:

```
int  search(int a[], int x)
{    for(int i = 0; i < a.length; i++) if(a[i] == x) return i;     // found
     return -1;                                                    // not found
}
```

The algorithm is clearly O(n) where n is the length of the array. Here is a second search method which assumes the data in the array is sorted:

```
int  binsearch(int a[], int lo, int hi, int x)
{    if(lo > hi) return -1;  // not found
     int mid = (lo+hi)/2;
     if(x == a[mid]) return mid;
     if(x > a[mid]) return binsearch(a, mid+1, hi, x);
     return binsearch(a, lo, mid-1, x);
}
int search(int a[], int x) { return binsearch(a, 0, a.length-1, x); }
```

What is the cost of this algorithm? It has no loops but instead calls itself recursively. A suitable basis for the cost is to count how many times binsearch is called. As usual we want to consider the worst case which occurs when x isn't in the array at all.

Suppose the array is [1, 3, 5, 7, 9, 11, 13, 15] and the algorithm is searching for the value x=8. The execution follows the sequence:

call binsearch(a,0,7,x)

lo = 0, hi = 7, mid = 3 a[3] = 7 < x so call binsearch(a,4,7,x)

lo = 4, hi = 7, mid = 5 a[5] = 11 > x so call binsearch(a,4,4,x)

lo = 4, hi = 4, mid = 4 a[4] = 9 > x so call binsearch(a,4,3,x)

lo = 4, hi = 3, lo > hi so x isn't in the array

This took 4 calls to binsearch. What is the general formula? Suppose the part of the array from lo to hi is being searched. If x isn't found at the mid-point then either a[lo..mid-1] or a[mid+1..hi] will be searched next. Each of these are slightly less than half the size of of the original range and the algorithm must terminate when the range doesn't include any values (lo > hi). This is roughly equivalent to asking how many times we can half the original length before we reach 0. I.e. in this case:

8/2 = 4; 4/2 = 2; 2/2 = 1; 1/2 = 0;

Four 'halvings' were possible which agrees with the expected result. So in general how many times can a number be halved before we reach 0? I deliberately chose a power of two to make things easier. Suppose we start with 2^k we can halve it:

$2^k/2 = 2^{k-1}$; $2^{k-1}/2 = 2^{k-2}$; ... $2^1/2 = 1\ (= 2^0)$; 1/2 = 0;

It should be easily seen that k+1 halvings are required.

Q1 How many calls of binsearch are required in the worst case if the array size is: 64, 128, 256, 512 or 1024. How many for 60, 100, 200, 500, 1000?

Q2 You might think counting the number of function calls is slightly different from counting the number of times round a loop. Can you modify binsearch to use a loop instead of recursion?

If $n = 2^k$ we write $k = \log(n)$. Even if n isn't exactly a power of 2, log(n) is defined it just isn't a whole number any more. Strictly speaking log(n) should be written $\log_2(n)$ but computer scientists almost aways just use log. It doesn't really matter because $\log_2(n) = c*\log_{10}(n)$ where c is a constant.

Q3 $\log_2(10)$ isn't a whole number. How would you approximate it when you want the number of comparisons for 10 numbers?

Q4 What is the value of c in $\log_2(n) = c*\log_{10}(n)$. Remember if $n = 10^z$ then $\log_{10}(n) = z$ and that $\log(a^b) = b*\log(a)$ whatever base of logarithms you use.

Q5 If you have a calculator does it have a log button. What base is used?

Returning to the point of this section. The performance of binsearch is O(log(n)). Since the linear version of the search is O(n) and log(n) < n the binary search is better in general.

Q6 Can you prove log(n) < n? (Hint: if a < b the $c^a < c^b$ if c > 1).

6 Sorting

One of the simplest sort algorithms is the Exchange Sort:

```
void    sort(int a[])
{   int n = a.length;
    for(int i = 0; i < n - 1; i++)
    {   int p = i;                          # position of biggest number seen so far
        for(int j = i+1; j < n; j++)
            if(a[j] > a[p]) p = j;
        if(p != i)
        {   int t = a[p];
            a[p] = a[i];
            a[i]  = t;
        }
    }
}
```

The idea is to find the biggest element and move it to the first position in the array. Then find the second biggest by starting at the second position in the array, the third biggest starting from the third position and so on. Before considering the mathematics of this note that there are two different things which we could count: how many comparisons are made or how many swaps of data are made. The total cost of the algorithm will depend on both of these. I.e. if there are s swaps and c comparisons and the time for one of each is t_s and t_c respectively the total cost will be $s*t_s + c*t_c$.

Q1 How many swaps are there?

To count comparisons takes a bit more effort. First consider the inner loop:

```
for(int j = i+1; j < n; j++)
    if(a[j] > a[p]) p = j;
```

How many comparisons take place here. The first time round j = i+1, the last time round j = n-1 which means the number of times round the loop is (n-1) - (i+1) + 1 = n - i - 1. The number of times round the inner loop depends on the value of i, so what values does i take?

```
for(int i = 0; i < n - 1; i++)
```

This is i = 0 to i = n-2 or n-2 - 0 + 1 = n-1 times round. But that doesn't really help since for each loop the number of times round the inner loop changes.

Here are some values:

i	times round inner loop	(n-i-1)
0	n-1	
1	n-2	
2	n-3	
...	....	
n-2	n - (n-2) - 1 = 1	

So the total number of comparisons is 1 + 2 + 3 + ... + n-1 whatever that is.

Starting from S = 1 + 2 + 3 + ... + n-1 we can also write

S = n-1 + n-2 + ... + 1 adding these together we get

2*S = (n-1 +1) + (n-2+2) + ... +(1 + n - 1)

= n + n + ... + n = n*(n-1)

So S = n*(n-1)/2 = $n^2/2$ - n/2. By the earlier argument n < n^2 so the S is $O(n^2)$.

7 How Good Can Sorting Be?

Exchange Sort is $O(n^2)$. There are lots of different sorting algorithms but for a long time they were all $O(n^2)$. Some were better than others (remember $O(n^2)$ is a worst case scenario). One of the first to be better was Shell Sort (named after someone called Shell surprisingly). Shell Sort is very hard to analyse and there are several versions. Depending on which version you use the performance is usually given as $O(n^{4/3})$ or $O(n^{6/5})$. How much better are these than n^2. Here are some sample figures:

n	$n^{6/5}$	$n^{4/3}$	n^2
10	16	22	100
100	251	464	10000
1000	4000	10000	1000000

These represent proportions: by the time you get to 1000 items Shell Sort is 100 times faster than earlier methods.

Shell Sort is much more complicated to program than, for example, Exchange Sort. This might be worth thinking about in practice. Suppose the actual formula for the time for exchange sort were something like $10*n^2 + 4*n + 3$ and Shell sort was $1000*n^{6/5} + 1000*n$, with Shell sort not only would you have more programming to do, be more likely to make mistakes, spend more time debugging your code but you might still find it wasn't faster because you were only sorting small amounts of data.

How about combining the binary search with a sort? Suppose you have k items in an array and you want to add another in order. You use O(log k) comparisons to decide where to put the new item, then you shuffle all the larger items down by one to insert the new item. You now have k+1 sorted items and do the same again until all the items are in place. How effective is that?

Suppose you have k items. log(k) + 1 operations finds the position for the next item. Assume it's in the middle of the data, you need to move k/2 items along by one which realistically counts as k/2 operations, making log(k)+1 + k/2. That's just for k items we need to do this with each number of items in turn from 1 to n-1 so the total time is[1]:

$$\begin{aligned} & \log(1) + 1 + 1/2 \\ + \; & \log(2) + 1 + 2/2 \\ + \; & \ldots\ldots \quad \ldots\ldots \quad \ldots\ldots \\ + \; & \log(n-1)+1 + (n-1)/2 \\ = \; & \Sigma\,[1 \le i \le n-1]\ \log(i) + n-1 + 1/2\ \Sigma[1 \le i \le n-1]\ i \end{aligned}$$

The last sum we already know, so, including the extra n-1 the second part is (n-1)(n+4)/4. The first part is harder - in fact its O(n log(n)) - but notice the algorithm is already at least $O(n^2)$.

The break through with sorts came with Quick Sort (which was described in the section on stacks). I'll summarise how it works again. Suppose you could arrange your data so that all the smaller values were at one end and the larger at the other, e.g. starting from

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

you partition the numbers into:

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

These numbers aren't sorted but the first five are all smaller than the second five. If you split the array in the middle and did the same thing to the two halves, and so on you would eventually get a sorted array. Not just eventually, quite quickly in fact. Suppose that we can partition our data into two parts in about n steps (we'll see how later). This gives the following cost:

n steps	partition into 2 n/2 size blocks
n/2 + n/2	partition each half into 2 n/4 size blocks
n/4 + n/4 + n/4 + n/4	partition each quarter into eights
... ...	and so on

Note that at each stage there are n steps to be taken, the next question is how many stages are there? As with the binary search, log(n) so the total cost is n log(n). How does O(n log(n)) compare with $O(n^2)$. Here is the Shell Sort table extended.

n	n log(n)	$n^{6/5}$	$n^{4/3}$	n^2
10	33	16	22	100
100	664	251	464	10000
1000	10000	4000	10000	1000000

As you can see by 1000 items it's as as good as one version of Shell Sort. As n increases, n log(n) is eventually smaller than even $n^{6/5}$ (somewhere between 1M and 10M).

Q1 Find a way to partition an array in O(n) steps. It doesn't have to be into two equal size blocks as long as all the numbers in one block precede all those in the other.

---

[1] The notation Σ [*cond*] x used in the example means add up all the x for which *cond* is true.

8 Expressions Which Are $O(2^N)$

Typically $O(2^n)$ results occur in situations where there are n objects of some sort, the answer involves some subset of them and we know no better way of solving the problem than trying every alternative in turn. Take as an example, computing x^n using the smallest number of multiplications. To work out x^n you can multiply x by itself n-1 times i.e. use n-1 multiplications alternatively you can use the following algorithm:

```
ans = 1
while n != 0 do
    if n is odd then  ans = ans*x
    x = x*x
    n = n/2                    // ignoring remainder
end while
```

This is a much better method.

Q1 Prove it is much better.

Though for small numbers this method usually gives the smallest number of multiplications it isn't perfect.

Q2 How many multiplications are needed to compute x^{15} using this method? Now try computing $y = x^3$ then compute $y^5 = x^{15}$ how many multiplications does that require?

One way of find the smallest number would be to look at every sequence of powers we could generate, check which ones could be used to compute x^n and chose the shortest one. If we just consider x^5 we could try all the alternatives:

1	x^2	x^3	x^4	x^5	*possible (though we don't need x^4)*
2	x^2		x^4	x^5	*possible and better than 1*
3	etc				

A simple calculation will show that there are $2^4 = 16$ answers, some of which must be the shortest. In general, the number of possible sequences for x^n is 2^{n-1}. This method is $O(2^n)$, if we wanted to compute 2^{32} using this method we would have to test over 4,000,000,000 sequences. If we generated and tested a different sequence every thousandth of a second it would take a year to complete. In most cases algorithms like this are non-starters.

Here is another, equally silly, algorithm. We could sort n numbers by arranging them in every possible order and checking in each case if they were sorted. In theory this is even worse than the last algorithm: there are n! arrangements which is even more than 2^n (n! is often described as being $O(n^n)$ it's actually slightly better than that but for big numbers it's close).

Note that, in the previous example, the problem is not inherently difficult - we have just chosen a silly algorithm for solving it. There are sorting methods which are $O(n^2)$ and even $O(n \log(n))$. Is the same true for computing x^n? Unfortunately nobody knows[2]. There are quick ways of computing x^n but none of them always gives the best answer.

There are algorithms which are $O(2^n)$ and are still usable - but only for very small values of n. In general if an algorithm has this performance it is of no real use in practice. In passing we note that if $m > 2$ then $m^n > 2^n$ so an algorithm which is $O(m^n)$ is even worse than one which is $O(2^n)$.

---

[2] What is known is that there almost certainly isn't a way of computing several powers of x (e.g. x^{10}, x^{13} and x^{20}) using the smallest number of multiplications, which is better than $O(2^n)$ where n is the largest power.

9 Relative Rates Of Growth

The orders we have considered satisfy: O(log n) is O(n) is O(n log(n)) is $O(n^k)$ (for any $k > 1$) is $O(2^n)$. This is an unfamiliar way of saying: $\log n \leq B\ n \leq C\ n \log n \leq D\ n^k \leq E\ 2^n$ for suitable B, C, D and E.

In general, problems for which an algorithm is known which is at worst polynomial in performance are referred to as 'tractable' (easy). We will refer to such problems as *polynomial time* (they are also said to be 'in class P' or just 'in P'). Problems for which the best algorithms are $O(2^n)$ are 'intractable' (hard). An intractable problem will, of course, be reclassified as tractable if we discover a polynomial algorithm for it.

This is generally agreed to be a rather poor distinction. Suppose an algorithm is $O(n^{1000000})$, then it is in practical terms intractable. Equally an algorithm which is $O(2^n)$ but for which the constant term is very small, i.e. its performance is, say, $\leq 0.000000000000001 * 2^n$ is usable for smallish values of n. The only two justifications we have are that in practice: a) few useful polynomial algorithms are worse than about $O(n^6)$ and b) most $O(2^n)$ algorithms become unusable very quickly.

10 The Classes P And NP

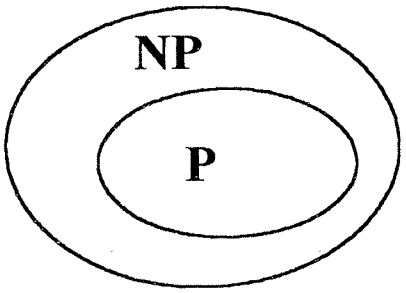
Suppose now, as with Finite State Machines, we allow non-determinism to creep into our problem solving, i.e. at certain points a program makes a choice between different alternatives on the basis of some 'secret' criteria. We could have done this with Turing Machines and if we had we would have found that, as with FA's, a non-determinstic TM can be simulated by a deterministic one. This means that non-deterministic programs can always be replaced by deterministic ones - but how does the cost of a deterministic version compare with the cost of a non-determistic one?

In the case of an NFA with N states, we showed that the equivalent DFA could have up to 2^N-1 states. If our unit of time is how long it takes to get from one state to another, then DFAs may take up to $O(2^N)$ times as long to process the same data as an NFA. The same is true for TMs. An algorithm which is polynomial time on a non-deterministic TM typically becomes intractable on a deterministic TM.

Of course this isn't necessarily fatal. It might be that for problem A we have a non-deterministic algorithm N_A which becomes intractable when replaced by its deterministic version, D_A but there may be a completely different algorithm P_A which solves A by a different method and which is deterministic and polynomial time.

One example quoted above, was sorting data by looking at all possible arangements of the data and choosing the correctly sorted one. It isn't difficult to write this as a non-deterministic algorithm, but it certainly becomes intractable if we convert it to its deterministic equivalent. But there are deterministic polynomial time sorting algorithms - all sensible ones are! The distinction must be made between algorithms and problems: for one problem there may be many algorithms which solve it, some of these algorithms will be non-deterministic ones operating in polynomial time, if we convert one of these to a non-determistic form it will typically become intractable.

The set of all problems which require Non-deterministic Polynomial time algorithms is called NP (there may of course be problems which even in the non-deterministic case are intractable - but that is just too dreadful to contemplate so we ignore them). If a problem is in P then it is also in NP (DFA are a special case of NFA, the definition of non-deterministic allows algorithms to use secret criteria but doesn't insist on it). So the situation looks something like ...



or in mathematical language P $\subset$ NP. Now it may happen that we put a problem in NP because we haven't found a polynomial algorithm for it, but later such an algorithm is found and the problem gets moved to P. Perhaps one day every problem in NP will be proved to be in P. If this is true then P = NP. The question of whether P = NP or not is one of the most long standing questions in computer science. For something like 30 years, some of the best computer scientists and mathematicians have tried to prove or disprove it. I don't know anyone who believes it to be true[3] but maybe ....

11 NP-Completeness

Let us take one further step in the direction of complexity. We have referred to 'problems' and 'algorithms' in a fairly casual way but we can be more precise. Whetever the problem is, the data for each instance of it can presumably be written down in some way. For example, the data for the smallest number of multiplications to find x^n is just 'n'. Also we can re-phrase the question as a decision problem (where the output is always 'true' or 'false'), e.g. maybe (27, 5) means can x^{27} be computed with 5 or less multiplications? As suggested earlier this isn't a real restriction. To find the smallest number of multiplications we can try (27, 27), (27, 26), (27,25) ... until the answer is 'false'. Some thought will show that we can do better than this even (consider a binary chop on the second number).

Now consider all possible pairs (23, 111), (14, 2), (6785, 121) ... imagine extracting from them those for which the answer to the question is 'true'. That set can be though of as a language. Suppose we call L the language of pairs for which the result should be 'true'. We can now describe what we mean by a problem in a very precise way: given a pair (n, m), decide if (n, m) $\in$ L. This gives us a new definition of intractable:

Let L be the language of true examples for a decision problem, D, and let α be a string of length n. If every (deterministic) TM which can decide whether or not $\alpha \in L$ takes a number of steps t(n) where t is $O(2^n)$ then D is intractable (i.e. $D \in NP$). If for some TM, t is $O(n^m)$ for some m then $D \in P$.

This is quite a neat (if not very practical) definition. We've talked about complexity and the big-O notation assuming that we could always guess what should be meant by 'n'. This definition is more precise: it's the length of the string needed to represent the input data. It also gives words like 'problem', 'algorithm' and 'cost' clear definitions.

---

[3] The line '$\therefore$ P = NP' appears on a chalk board in a Bart Simpson cartoon. It does turn up from time to time in cartoons and SF comics. Perhaps one day it will be as famous as $e = m c^2$

In about 1970 a computer scientist called Cook used this formulation to prove an interesting result (now called Cook's Theorem). The theorem is too advanced for this course but it's conclusions are important: the NP problems include a subset called the NP-Complete problems, if a polynomial time algorithm exists for any NP-Complete problem then a polynomial time algorithm exists for every problem in NP. Now if you find a polynomial time algorithm for a problem which was previously thought to be in NP you haven't done much, but if that problem is in NP-Complete you have proved that P = NP.

At the time Cook only identified one NP-Complete problem (true computer scientists will probaby guess that it involved emulating a TM) but since then thousands of NP-Complete problems have been found. Nobody has ever been able to find a polynomial time algorithm for any of them.

Here are a random selection of NP-Complete problems:

1 Given a set of numbers { a, b, c, ... n } and a number s, can x^a ... x^n be computed using s or less multiplications?

2 Given two finite automata do they accept the same language?

3 Given a collection of finite automata is there any string which all of them accept?

4 Given three positive integers a, b and c, do there exist positive integers x and y such that $a*x^2 + b*y = c$?

5 Given a network is there a path which visits every node exactly once and terminates at it's starting node?

And here is a very odd problem to end with .....

A group of people join a marriage agency. Each person lists the people they are prepared to marry. There is an algorithm to decide if there is any way of meeting all their requirements (i.e. everybody marrying someone on their list of possibles). The algorithm is tractable (polynomial time). On the planet Zog however there are three sexes, and people (well, green slimy things actually) at Zog dating agencies have to give lists of pairs of possible partners. Deciding if there is any arrangement which suits everyone is now NP-Complete.

Q1 On the planet Qrsklt there are 4 sexes, show that the Qrsklt agency's problem must be *at least* NP-Complete (Hint: members of the fourth sex on Qrsklt are all soft, cuddly and very lovable).

APPENDIX A - GREEK LETTERS

Computer scientists like using Greek letters for things. If you use a PC Windows word processor they can all be found in the Symbol font. Here is a list in case you are not familiar with them. I've put them in more or less 'English' order (where would you put φ?). Usually the starting letter of the name tells you which key to press when using Symbol.

lower	upper	name	pronounced	notes
α	A	alpha	al-fa	
β	B	beta	bee-ta	Americans say 'bay-teh'
χ	X	chi	ki	'c' on the keyboard
δ	Δ	delta	delt-a	
ε	E	epsilon	ep-sil-on	
φ	Φ	phi	fi	'f' on the keyboard
γ	Γ	gamma		
η	H	eta	ee-ta	'h' on the keyboard
θ	Θ	theta	thee-ta	'th' as in 'theatre', 'q' on the keyboard
ι	I	iota	i-o-ta	
κ	K	kappa		
λ	Λ	lambda	lam-da	
μ	M	mu	mew	
ν	N	nu	new	
ο	O	omicron	oh-me-cron	
π	Π	pi		
ρ	P	rho	row	as in 'row a boat'
σ	Σ	sigma	sig-ma	
τ	T	tau	tow	with 'ow' as in 'cow'
υ	Y	upsilon	up-sill-on	'u' on the keyboard
ω	Ω	omega	oh-mega	'w' on the keyboard
ξ	Ξ	xi	zi	'x' on the keyboard
φ	Ψ	psi	p'si	'j'/'Y' on the keyboard
ζ	Z	zeta	zee-ta	

APPENDIX B - BINARY NUMBERS

I assume people learn something about binary numbers at primary school these days. You don't really need it for this module but it's so often referred to that I've included this short summary.

1 Ye Olde Binarie Numbere Systeme

It is a little known fact that the English invented binary numbers back in the 13th century or earlier. Unfortunately they only used it for measuring amounts of wine, beer and spirits which may have contributed to the fact it got forgotten. A few remnants remain in English, but here is a more complete summary of how drink was measured:

2 gills = 1 chopin
2 chopins = 1 pint
2 pints = 1 quart
2 quarts = 1 pottle
2 pottles = 1 gallon
2 gallons = 1 peck
2 pecks = 1 demibushel
2 demibushels = 1 bushel (or firkin)
2 bushels = 1 kilderkin
2 kilderkins = 1 barrel
2 barrels = 1 hogshead
2 hogsheads = 1 pipe
2 pipes = 1 tun

If they had automatic drink dispensers they might have shown amounts like:

Tun	Pipe	Hogshead	Barrel	Kilderkin	Bushel	½Bushel	Peck	Gallon	Pottle	Quart	Pint	Chopin	Gill
0	0	0	0	0	0	0	1	0	0	1	0	1	1

Notice that only 0 and 1 are ever displayed because 2 of anything (other than tuns) make one of the next thing. Of course people might still *say* 3 bushels 2 pottles and a gill but it would come up in 1s and 0s on the display

Q1 How would you display "3 bushels 2 pottles and a gill"?

Q2 What's the amount (in gills) displayed on the dispenser?

Q3 How would you write the following amounts?

1) 7 pints
2) 16 gills
3) 5 gills
4) 3 quarts
5) 8 pottles
6) 5½ chopins

	Tun	Pipe	Hogshead	Barrel	Kilderkin	Bushel	½Bushel	Peck	Gallon	Pottle	Quart	Pint	Chopin	Gill
1														
2														
3														
4														
5														
6														

If you allow the scale to continue indefinitely beyond tuns you have the binary number system. Every thing is written in 1s and 0s and instead of going up in tens you go up in twos. The equivalent of units, tens, hundreds, thousands etc. are:

Decimal	Binary	As a Power of 2
1	1	2^0
2	10	2^1
4	100	2^2
8	1000	2^3
16	10000	2^4
32	100000	2^5
64	1000000	2^6
128	10000000	2^7
256	100000000	2^8
512	1000000000	2^9
1024	10000000000	2^{10}
2048	100000000000	2^{11}
4096	1000000000000	2^{12}
8192	10000000000000	2^{13}
16384	100000000000000	2^{14}
32768	1000000000000000	2^{15}
65536	10000000000000000	2^{16}

2 Octal & Hex(adecimal)

Binary numbers are confusing: 10100101101_2 is easily confused with 10101001101_2 for example (the $_2$ is to show that this is a binary number not a decimal one). For this reason binary information which is to be read by human beings is sometimes written in octal. Octal means counting in base 8 but all you need to know is that it is just binary written in groups of 3 bits (from the right) with each group of three bits replaced by the 'decimal' digit equivalent. Using the number above (with an extra 0 on the front to make a multiple of 3 bits)

*0*10 100 101 101	Binary number
2 4 5 5	Individual 'octets'
2455_8	An octal number

The second number was:

*0*10 101 001 101	Binary number
2 5 1 5	Individual 'octets'
2515_8	An octal number

I hope you will agree that the octal version, being 4 digits rather that 11, is easier to memorise.

If 4 bits are used to group bits rather than 3 we have hexadecimal. There is a problem here, some groups of 4 bits require 2 decimal digits, e.g. $1111_2 = 15$ which could be even more confusing than working in binary. The following codes are therefore used:

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Binary	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

Using the same numbers:

*0*101 0010 1101	Binary number
5 2 D	Individual hex digits (sometimes called nybbles)
$52D_{16}$	A hex number

The second number:

*0*101 0100 1101	Binary number
5 4 D	Individual hex digits
$54D_{16}$	A hex number

Hex is used much more commonly than octal. Most computers manipulate bits in even length blocks. A byte is 8 bits, a computer 'word' depends on the machine but is typically 16 or 32 bits. Since these numbers are divisible by four but not by three it's more sensible to use hex.

Q1 What hex number comes after 19_{16} ? What is immediately before 20_{16}?

Q2 Convert the following hex numbers to binary and then to decimal:

a) 1B b) 1F c) 11 d) 10

Q3 What is your age in hexadecimal? If you were 20_{16} how old would you be?

Q4 What do these bit patterns spell in hex?

a) 1101111010101101

b) 1111111011101101

c) 1101111010101111

d) 101011001110

e) 1100111011011110

Q5 If you use 0 for O, 1 for I, 5 for S and the hex letters A - F what's the longest word you can construct and what bit pattern does it represent?

3 Binary Number Answers

Q1 "3 bushels 2 pottles and a gill" is

Tun	Pipe	Hogshead	Barrel	Kilderkin	Bushel	½Bushel	Peck	Gallon	Pottle	Quart	Pint	Chopin	Gill
0	0	0	0	1	1	0	0	1	0	0	0	0	1

Q2 75 gills

Q3

	Pottle	Quart	Pint	Chopin	Gill
1	1	1	1		
2	1				
3			1		1
4	1	1			
5					
6		1		1	1

4 Hex Answers

Q1 $1A_{16}$ and $1F_{16}$

Q2 Convert the following hex numbers to binary and then to decimal:

a) 11011_2 27 b) 11111_2 31

c) 10001_2 17 d) 10000_2 16

Q3 My age is 32_{16} which sounds a lot better than the decimal equivalent!

Q4 a) DEAD b) FEED c) DEAF d) ACE e) CEDE

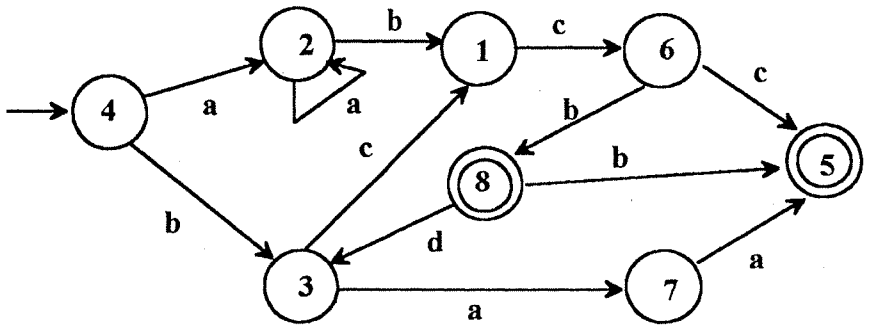
Q5 Who knows!

APPENDIX C - FAS TO REGULAR LANGUAGES

We have shown earlier that given a regular language L we can find a corresponding FA. This appendix proves the converse is also true: to every FA there is a regular language. Unfortunately our proof doesn't tell you what the language corresponding to a given FA is, just that one must exist.

The proof uses a technique called induction which I'll explain as I go along. It also, in a rather typically computer science/maths way starts by solving a more general problem.

Suppose we have an FA with states numbered 1, 2, ... N. As a specific example consider the FA below which has eight states (the numbering is arbitrary).



Let p and q be any two of these states. Suppose the FA is initially in state p, i.e. either p is the start state or some input has brought us to state p. Now consider all the strings which, from p will take us to q. There may be none, for example if p=3 and q=4 there is no input which takes us from p to q. There may be a single direct connection: for p=4, q=3 the input string **b** gets us from one to the other, or it may be more complex: for p=1 q = 3 the strings include **cbd**, **cbdccbd**, **cbdccbdccbd** (you are allowed to visit the end point more than once and go back to start point).

We will write L(p, q) for the strings which start at p and end at q. For each p and q, L(p,q) is a set of strings, i.e. a language. In the case p=3 and q=4 above there are no strings so L(3,4) = ∅ (the empty set). L(4,3) = { b }, L(1,3) = { cbd, cbdccbd, cbdccbdccbd, ... } = { cbd } . { ccbd }*. We will call any language L(p,q) a sub-language of our FA. Each of the example sub-languages is a regular language[1] though that doesn't prove anything. Suppose we could show that for all p and q, L(p,q) is regular then L(4,8) would be regular and so would L(4,5) and from the rules for regular languages, L(4, 8) ∪ L(4, 5) would be regular but that is (by definition) the set of all strings going from 4 (the start state) to either 8 or 5 (the two halt states).

Claim 1: If all sub-languages of an FA are regular then the FA accepts a regular language.

Proof: The set of all strings going from the start to any of the halt states is by definition the language accepted by the FA. Suppose the start state is S and the halt states are H_1, H_2, ... H_n then $L(S,H_1)$,

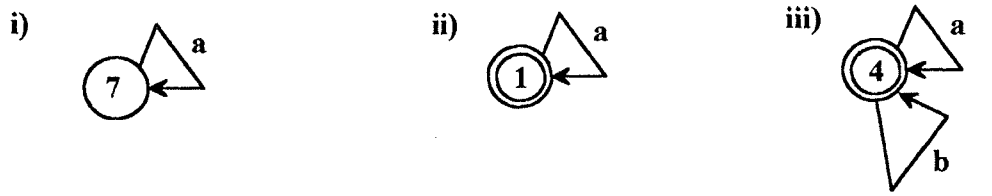
[1] The empty set is regular. Consider any FA with no halt state. It can't accept any strings so its language is {} = ∅. To remind you, this is not the same as the language { Λ } whose FA has exactly one state and that is its halt state.

$L(S,H_2)$ ... $L(S,H_n)$ are all regular. By the definition of regular languages, this makes $L(S,H_1) \cup L(S,H_2) \cup \ldots \cup L(S, H_n)$ regular and this is the language accepted by the FA.

So if for all p and q, L(p,q) is regular then we know that every FA has a corresponding regular language.

Proving that all sub-languages are regular doesn't look easy. Indeed as hinted above we seem to have made the problem more complicated. The next step to look at L(p,q) for different 'sizes' of FA. The key point is that if all FAs with, say, M states have corresponding regular languages then so do FAs with M+1 states. To distinguish FAs with different numbers of states we will write L(p,q,N) when we have an FA with N states. So in our example above we should have written L(p, q, 8) each time because there were 8 states.

First let's look at the simplest possible FAs, those with only one state. Here are a few examples:



I have deliberately used silly state numbers to emphasise that the numbering of states isn't important, just how many there are. For i) we have $L(7,7,1) = \varnothing$ since there is no halt state. for ii) we have $L(1,1,1) = \{ a \}^*$, for iii) $L(4,4,1) = \{ a, b \}^*$. Check that you agree with this and confirm that the following statement is true:

Claim 2: If an FA has a single state numbered n then L(n, n,1) is a regular language. In fact it is either $\varnothing$ or A* where A is a language of one or more single characters.

Suppose now that we could show that all FAs with up to M states that L(p,q,r) is regular for all p, q and r (obviously $r \leq M$), i.e. if you have an FA with M states, if you extract any part of it by dropping some states and any transitions going to or from them you still get an FA all of whose sub-languages are regular. We will show that if we could prove this for M states then adding another state would make a bigger FA which still corresponds to a regular language.

Let's make this more concrete. If M=1 we have something like the examples above. Take the second one, whose only sub-language is L(1,1,1) and add a second state numbered 2. The new FA has sub-language L(1,1,1) and L(2,2,1) (the language of the new state) plus L(1, 2, 2) and L(2, 1, 2) which represent any links between states 1 and 2. What does L(1,2,2) look like? Is it necessarily a regular language? Start at node 1, we can accept all the strings in L(1,1,1) without leaving 1, then move to state 2 suppose there are transitions for characters x, y, ... say. Then when we first reach 2 we will have accepted a string from L(1,1,1). { x, y, ... } - this is a regular language! Now we are in state 2 we can stay there for a while accepting strings from L(2,2,1). So if we start in 1 go to 2 and don't leave it we will accept strings from L(1,1,1) . { x, y, ... } . L(2,2,1). Suppose characters a, b, ... take us from state 2 to state 1, then to get back to 1 we will have accepted a string of the form: L(1,1,1) . { x, y, ... } . L(2,2,1) . { a, b, ... }.

The strings in L(1,2, 2) all start in state 1 and end in state 2 though state 2 may be visited several times as a result of going back to 1. In general if there is only one visit to state 2 the string accepted will be in the language L(1,1,1) . { x, y, ... } . L(2,2,1). Returning to 1 and then to 2 for a second time means the string is in L(1,1,1) . { x, y, ... } . L(2,2,1) . {a, b, ... } L(1,1,1) . { x ,y, ...} . L(2,2,1).

We'll write K as shorthand for {a, b, ... } L(1,1,1) . { x ,y, ...} . L(2,2,1) then L(1,2,2) consists of all the strings in the languages:

L(1,1,1) . { x, y, ... } . L(2,2,1)	*State 2 visited only once*
L(1,1,1) . { x, y, ... } . L(2,2,1) . K	*State 2 visited twice*
L(1,1,1) . { x, y, ... } . L(2,2,1) . K . K	*State 2 visited three times*
L(1,1,1) . { x, y, ... } . L(2,2,1) . K . K . K	*and so on*

but by "all the strings in the languages" we mean their union, which in this case is L(1,1,1).{ x, y, ... }.L(2,2,1).K*.

A pause to see what we have proved:

L(1,2,2) = L(1,1,1).{x, y ... }.L(2,2,1).K*, where K = {a, b, ... }.L(1,1,1).{x ,y, ...}.L(2,2,1)

Now K is a regular language (why?) so K* is also, L(1,1,1), {x, y ... }, L(2,2,1) are also regular languages and so L(1,2,2) is also (why?). A similar argument shows that L(2,1,2) is a regular language. Thus 1 state FAs (which always accept regular languages) when augmented with an extra state produce a new FA all of whose sub-languages are regular.

We could now move on to 3 state FAs and show that all there sub-languages are regular because by building them up from the sub-languages of 2 and 1 state machines. From there we could show 4 state FAs are regular and so on. This would literally take for ever, instead show how to do it in general. Suppose for some N we have proved that all FAs with N or less states accept a regular language. Suppose the states are numbered 1 to N. Add a new state N+1 and any transitions you wish going from the existing states to N+1 and outgoing transitions to any of states 1 to N you wish. We now show that in this general case you still have an FA which accepts a regular language.

First consider any L(p, q, N+1). Then following the 1 to 2 state example, L(p, q, N+1) consists of all those strings which don't require a visit to N+1 (a regular language because if N+1 isn't visited you have, in effect, an automata with N states only which we are assuming accepts a regular language) plus all those that do. Those that do visit state N+1 must first visit one of the states with a transition to N+1, then take that transition. If A is state which has transitions to N+1 for characters x, y, ...the strings from p to A are L(p, A, N) (regular), and those from p to N+1 are L(p,A,N).{x, y, ... }.L(N+1,N+1, 1). Of course there will, in general, be lots of states from which N+1 can be reached but we can take the union of the languages involved to get a regular language corresponding to all strings from p to N+1. A similar argument shows that all strings from N+1 back to itself is a regular language and all strings from N+1 to q. Thus L(p, q, N+1) can be built up from languages of the form L(p, q, N) by using union, concatenation and closure. By definition these operations preserve regularity of languages. So if, for some N, all L(p, q, N) are regular then so are all the L(p, q, N+1). But we know the L(p,q,1) are regular, so the L(p,q,2) are but if the L(p,q,2) are the L(p, q, 3) must be and so on.

Kleene's Theorem: Every Finite Automata accepts a regular language and every regular language is accepted by a Finite Automata.

Faculty of Environment and Technology

UWE, Bristol
Frenchay Campus
Coldharbour Lane
Bristol BS16 1QY

Telephone 0117 965 6261

UWE, PF563723 01.15
Printing & Stationery Services