Computability, Algorithms and Complexity
Nominally 20 lectures

© Ian Hodkinson
Department of Computing
Imperial College
180 Queen's Gate
London SW7 2BZ
UK

email imh@doc.ic.ac.uk)

Introduction

This course has three parts:

I: computability theory,
II: algorithms,
III: complexity theory.

In Part I we develop a model of computing, and use it to examine the fundamental properties and limitations of computers in principle (notwithstanding future advances in hardware or software). Part II examines some algorithms of interest and use, and Part III develops a classification of problems according to how hard they are to solve.

Parts I and III are fairly theoretical in approach, the aim being to foster understanding of the intrinsic capabilities of computers, real and imagined. Some of the material was crucial for the development of modern computers, and all of it has interest beyond its applications. But there are also practical reasons for teaching it:

- It is a good thing, perhaps sobering for computer scientists, to understand more about what computers can and can't do.
- You can honourably admit defeat if you know a problem is impossible or hopelessly difficult to solve. It saves your time. E.g., I heard that a programmer in a large British company was recently asked to write a program to check whether some communications software would 'loop' or not. We will see in §5 that this is an impossible task, in general.

- The material we cover, especially in Part I, is part of the 'computing culture', and all computer scientists should have at least a nodding acquaintance with it.
- The subject is is of wide, indeed interdisciplinary interest. Popular books like Penrose's (see list above) and Hofstadter's 'Gödel, Escher, Bach' cover our subject, and there was quite a famous West End play ('Breaking the Code') about Turing's work a few years ago. The 'Independent' printed a long article on Gödel's theorem on 20.6.92, in which it was said 'It is a measure of the achievement of Kurt Gödel that his Incompleteness Theorem, while still not considered the ideal subject with which to open a dinner party conversation, is fast becoming one of those scientific landmarks — like Einstein's Theory of Relativity and Heisenberg's Uncertainty Principle — that educated people, even those with no scientific training, feel obliged to know something about.' Lucky you: we do Gödel's theorem in §5!

Part I: Computability theory

Formalising *computable problem.* Turing machines and Church's thesis. Examples. Relation of Turing machines to variants (multi-tape, 2-dimensional machines). Comparison of Turing machines with alternative formalisms. Universal Turing machine. Unsolvable problems, reducibility theory.

---

1. What is an algorithm?

We begin Part I with a problem that could pose difficulties for those who think computers are 'all-powerful'. To analyse the problem, we then discuss the general notion of an algorithm (as opposed to particular algorithms), and why it is important.

1.1. THE PROBLEM

At root, Part I of this course is about paradoxes, such as:

The least number that is not definable by an English sentence having fewer than 100 letters.

(The paradox is that we have just defined this number by such a sentence. Think about it!) C.C. Chang and H.J. Keisler kindly dedicated their book 'Model Theory' to all those people who haven't got a book dedicated to them. (Is it dedicated to you or not?)

Paradoxes like this often arise because of self-reference within the statement. The first one implicitly refers to all (short) English sentences, including itself. The second refers implicitly to all books, including 'Model Theory'. Now computing also uses languages — formal programming languages — that are capable of self-reference (for example, programs can alter, debug, compile or run other programs). Are there similar paradoxes in computing?

Here is a candidate. Take a high-level imperative programming language such as Modula_2. Each program is a string of English characters (letters, numbers, punctuation, etc). So we can list all the syntactically correct programs in alphabetical order, as $P_1, P_2, P_3, \ldots,$. Every program occurs in this list.

Each P_n will output some string of symbols, possibly the empty string. We can treat it as outputting a string of binary bits (0 or 1). Most computers

work this way — if the output appears to us as English text, this is because the binary output has been treated as ASCII (for example), and decoded into English.

Now consider the following program P:

```
1   repeat forever
2       generate the next program Pn in the list
3       run Pn as far as the nth bit of the output
4       if Pn terminates or prompts for input before the nth bit is output then
5           output 1
6       else if the nth bit of Pn's output is 0 then
7           output 1
8       else if the nth bit of Pn's output is 1 then
9           output 0
10      end if
11  end repeat
```

- This language is not quite Modula_2, but the idea is the same — certainly we could write it formally in Modula_2.
- Generating and running the next program (lines 2 and 3) is easy — we generate all strings of text in alphabetical order, and use an interpreter to check each string in turn for syntactic errors. If there are none, the string is our next program, and the interpreter can run it. This is slow, but it works.
- We assume that we can write an interpreter in our language — certainly we can write a Modula_2 interpreter in Modula_2 (at least if we allow a few non-standard features such as dynamic memory allocation).
- In each trip round the loop, the interpreter is provided with the text of the next program, P_n, and the number n. The interpreter runs P_n, halting execution if (a) P_n itself halts, (b) P_n prompts for input or tries to read a file, or (c) P_n has produced n bits of output.
- All other steps of P are easy to implement.

So P is a legitimate program. So P is in the list of P_n's. Which P_n is P?

Suppose that P is P_7 say. Then P has the same output as P_7. Now on the seventh loop of P, P_7 (i.e., P) will be generated, and run as far as its seventh output bit. The possibilities are:

1. P_7 halts or prompts for input before it outputs 7 bits (impossible, as the code for $P = P_7$ has no HALT or READ statement!)
2. P_7 does output bit 7, and it's 0. Then P outputs 1 (look at the code above). But this 1 will be the 7^{th} output bit of $P = P_7$, a contradiction!
3. P_7 does output bit 7, and it's 1. Then P outputs 0 (look at the code again). But this 0 will be P's 7^{th} output bit, and $P = P_7$!

This is a contradiction: if P_7 outputs 0 then P outputs 1, and vice versa; yet P was supposed to be P_7. So P is not P_7 after all.

In the same way we can show that P is not P_n for any n, because P differs from P_n at the n^{th} place of its output. So P is not in our list of programs. This is impossible, as the list contains all programs of our language!

[Exercise: what is wrong?]

Paradoxes might not be too worrying in a natural language like English. We might suppose that English is vague, or the speaker is talking nonsense. But we think of computing as a precise engineering-mathematical discipline. It is used for safety-critical applications. Certainly it should not admit any paradoxes. We should therefore examine our 'paradox' very carefully.

It may be that it comes from some quirk of the programming language. Perhaps a better version of Modula_2 or whatever would avoid it. In Part I of the course our aim is first to show that the 'paradox' above is extremely general and occurs in all reasonable models of computing. We will do this by examining a very simple model of a computer. In spite of its simplicity we will give evidence for its being *fully general*, able to do anything that a computer — real or imagined — could.

We will then rediscover the 'paradox' in our simple model. I have to say at this point that there is no real paradox here. The argument above contained an implicit assumption. [What?] Nonetheless, there is still a problem: the implicit assumption cannot be avoided, because if it could, we really would have a paradox. So we cannot 'patch' our program P to remove the assumption!

But now, because our simple model is so general, we are forced to draw fundamental conclusions about the limitations of computing itself. Certain precisely-stated problems are unsolvable by a computer even in principle. (We cannot write a patch for P.)

There are lots of unsolvable problems! They include:

- checking mechanically whether an arbitrary program will halt on a given input (the 'halting problem')
- printing out all the true statements about arithmetic and no false ones (Gödel's incompleteness theorem).
- deciding whether a given sentence of first-order predicate logic is valid or not (Church's theorem).

Undeniably these are problems for which solutions would be very useful.

In Part III of the course we will apply the idea of self-reference again in the area of NP-completeness — not now to the question of what we can

compute, but to how fast can we compute it. Here our results will be more positive in tone.

1.2. WHAT IS AN ALGORITHM?

To show that our 'paradox' is not the fault of bad language design we must take a very general view of computing. Our view is that computers (of any kind) implement *algorithms.* So we will examine what an algorithm is.

First a definition from Chambers Dictionary.

algorithm, al'go-ridhm, n. a rule for solving a mathematical problem in a finite number of steps. [Root: Late Latin *algorismus*, from the Arabic name *al-Khwārazmi*, the native of Khwārazm (Khiva), i.e., the 9th century mathematician Abu Ja'far Mohammed ben Musa.]

We will improve on this, as we'll see.

1.2.1. Early algorithms

One of the earliest algorithms was devised between 400 and 300 B.C. by Euclid: it finds the highest common factor of two numbers, and is still used. The sieve of Erastothenes is another old algorithm. Mohammed al-Khwā))razmi is credited with devising the well-known rules for addition, subtraction, multiplication and division of ordinary decimal numbers.

Later examples of machines controlled by algorithms include weaving looms (1801, the work of J. M. Jacquard, 1752–1834), the player piano or piano-roll (the Pianola, 1897 — arguable, as there is an analogue aspect (what?)), and the 1890 census tabulating machine of Herman Hollerith, who is immortalised in the 'H' of the Fortran FORMAT statement (eg., FORMAT 4Habcd). These machines all used holes punched in card. In the 19th century Charles Babbage planned a multi-purpose calculating machine, the 'analytical engine', also controlled by punched cards.

1.2.2. Formalising *Algorithm*

In 1900 the great mathematician David Hilbert asked whether there is an algorithm that answers every mathematical problem. So people tried to find such an algorithm, without success. Soon they began to think it couldn't be done! Eventually some asked: can we prove that there's no such algorithm? This question involved issues quite different from those needed to devise algorithms. It raised the need to be precise about what an algorithm actually is: to formalise the notion of 'algorithm'.

Why did no-one give a precise definition of *algorithm* in the preceding two thousand years? Perhaps because most questions on algorithms are of the form *find one to solve this problem I've got.* This can be done without a formal definition of *algorithm*, because we know an algorithm when we see one. Just as an elephant is easy to recognise but hard to define, you can write a program to sort a list without knowing exactly what an algorithm is. It is enough to invent something that intuitively is an algorithm, and that solves the problem in question. We do this all the time.

But suppose we had a problem (like Hilbert's) for which many attempts to find an algorithmic solution had failed. Then we might suspect that the task is impossible, so we would like to prove that no algorithm solves the problem. To have any hope of doing this, it is clearly essential to define precisely what an algorithm is, because we've got to know what counts as an algorithm. Similarly, to answer questions concerning all algorithms we need to know exactly what an algorithm is. Otherwise, how could we proceed at all?

1.2.3. Why formalise *Algorithm?*

As we said, we formalise *algorithm* so that we can reason about algorithms in general, and (maybe) prove that some problems have no algorithmic solution. Any formalisation of the idea of an *algorithm* should be:

- precise and unambiguous, with no implicit assumptions, so we know what we are talking about. For maximum precision, it should be phrased in the language of mathematics.
- simple and without extraneous details, so we can reason easily with it.
- general, so that all algorithms are covered.

Once formalised, an idea can be explored with rigour, using high-powered mathematical techniques. This can pay huge dividends. Once gravity was formalised by Newton as $F = Gm_1m_2/r^2$, calculations of orbits, tides etc. became possible, with all that that implies. Pay-offs from the formalisation of *algorithm* included the modern programmable computer itself.[1] This is quite a spectacular pay-off! Others include the answer to Hilbert's question, related work in complexity (see Part III) and more besides.

1.2.4. *Algorithm* formalised

The notion of an algorithm was not in fact made formal until the mid 1930s, by mathematicians such as Alan Turing in England and (independently) Alonzo Church in America. Church and Turing used their formalisations to show that some mathematical problems have no algorithmic solution — they are unsolvable. (Turing used our 'paradox' to do this.) Thus, after 35 years, Hilbert's question got the answer NO.

[1] This is, of course, an arguable historical point; Hodges' book examines the historical background.

Turing invented the primitive computer called (nowadays!) the *Turing machine.* The Turing machine first appeared in his paper in the reading list, in 1936, some ten years before 'real' computers were invented[2]. Turing's formalisation of the notion of an algorithm was: *an algorithm is what a Turing machine implements.*

We will describe the Turing machine at length below. We will see that it is precise and simple, just as a formalisation should be. However, to claim that it is fully general — covering all known and indeed all conceivable algorithms — may seem rash, especially when we see how primitive a Turing machine is. But Turing gave substantial evidence for this in his paper, evidence which has strengthened over the years, and the usual view nowadays is that the Turing machine is fully general. For historical reasons this view is known as *Church's thesis*, or sometimes (better) as the *Church-Turing thesis.* We will examine the evidence for it after we have seen what a Turing machine is.

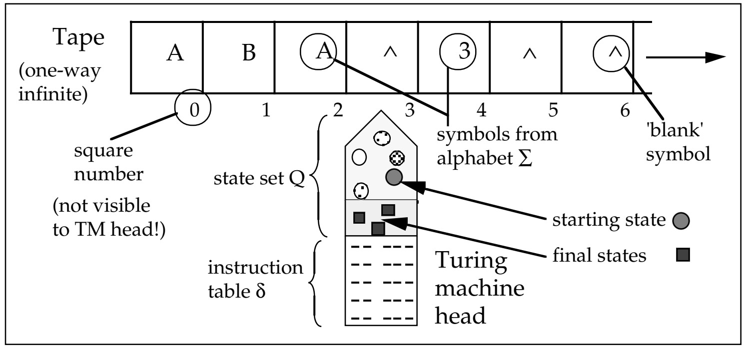
1.3. TURING MACHINES

We, also, will use Turing machines to formalise the concept of *algorithm.* Here we explain in outline what a Turing machine is; we'll do it formally in §2. As we go through, think about how the Turing machine, our formalisation of *algorithm,* fits our requirements of *precision* and *simplicity.* Afterwards we'll say more about its *generality* and why we use it in this course.

1.3.1. Naming of Parts

There are several, mildly different but equally powerful, versions of the TM in the textbooks. We now explain what our chosen version of the TM is, and what it does. In a nutshell, a Turing machine consists of a head that moves up and down a tape, reading and writing as it goes. At each stage it's in one of finitely many 'states'. It has an instruction table that tells it what to do at each step, depending on what it's reading and what state it's in.

[2] Turing later became one of the pioneers in their development.



1.1 A Turing machine

1.3.1.1. The tape

The <u>main memory</u> of a TM is a 1-way-infinite *tape*, viewed as laid out from left to right. The tape goes off to the right, forever. It is divided into *squares* , numbered 0, 1, 2, … ; these numbers are for our convenience and are <u>not</u> seen by the Turing machine.

1.3.1.2. The alphabets

In each square of the tape is written a single *symbol*. These symbols are taken from some finite *alphabet*. We will use the Greek letter *sigma* (Σ) to denote the alphabet. The alphabet Σ is part of the Turing machine. Σ is just a set of symbols, but it will always be finite with at least two symbols, one of which is a special *blank* symbol which we always write as '∧'. Subject to these restrictions a Turing machine can have any finite alphabet Σ we like.

A blank in a square really means that the square is empty. Having a symbol for 'empty' is convenient — we don't have to have a special case for empty squares, so things are kept <u>simple</u>.

1.3.1.3. The read/write head

The TM has a single *head*, as on a tape recorder. The head can read and write to the tape.

At any given moment the head of the TM is positioned over a particular square of the tape — the *current square.* At the start, the head is over square 0.

1.3.1.4. The set of states

The TM has a finite set Q of *states*. There is a special state q_0 in Q, called the *starting* or *initial state.* The machine begins in the starting state, and changes state as it goes along. At any given stage, the machine will be 'in' some particular state in Q, called the *current state.* The current state is one of

the two factors that determine, at each stage, what it does next (the other is the character in the square where the head is). The state of the TM corresponds roughly to the <u>current instruction</u> together with the contents of the <u>registers</u> in a conventional computer. It gives our 'current position' within the algorithm.

1.3.2. Starting a TM; input

A Turing machine begins a run in the initial state, with its head over square 0. At the beginning, the tape will contain a finite number (possibly 0) of non-blank symbols, left-justified; this string of non-blank symbols constitutes the *input* to the Turing machine. The rest of the tape squares will be blank (i.e., they contain ∧).

1.3.3. The run of the TM

A *run* is a step-by-step computation of the TM. At each step of a run:

(a) the head reads the symbol on the current tape square (the square where the head now is).

Then the TM does three things.

(b) First the head writes some symbol from Σ to the current tape square.

Then:

(c) the TM may move its head left or right along the tape by one square

(d) the TM goes into a new state.

Now the next step begins: it does (a)-(d) again, perhaps making different choices in (b)-(d) this time. And so on, step by step.

Note that:

- The TM writes <u>before</u> moving the head.
- In (b), the TM could write the same symbol as it just read. So in this case the tape contents will not change.
- Similarly, in (d) the state of the TM may not change (as perhaps in a loop). In (c), the head may not move.

Also notice that the tape will always contain only finitely many non-blank symbols, because at the start only finitely many squares are not blank, and at most one square is altered at each step.

1.3.3.1. The instruction table

At each step (b)-(d) above there are 'choices' to be made. Which symbol to write? Which way to move? And which state to enter? The answers depend <u>only</u> on:

(i) which symbol the machine reads from the current tape square;

(ii) the current state of the machine.

The machine has an *instruction table,* telling it what to do when, in any given state, a given symbol is read. We write the instruction table as δ, the

Greek letter *delta.* δ corresponds to the program of a conventional computer. It is in effect just a list with five columns:

current_state;	current_symbol;	new_state;	new_symbol;	move
current_state;	current_symbol;	new_state;	new_symbol;	move
current_state;	current_symbol;	new_state;	new_symbol;	move
.....	.....	.....	.....	.....

Knowing the current state and symbol, the Turing machine can read down the list to find the relevant line, and take action according to what it finds. To avoid ambiguity, no pair <current_state; current_symbol> should occur in more than one line of the list[3]. (You might think that every such pair should occur somewhere in the list, but in fact we don't insist on this: see *Halting* below.)

Clearly the 'programming language' is very low-level, like assembler. This fits our wish to keep things simple. But we will see some higher-level constructs for TMs later.

1.3.4. Stopping a TM; output

The run of a TM can terminate in just three different ways.

1. Some states of Q are designated special states, called *final* or *halting states.* We write F for the set of final states. F is a subset of Q. If the machine gets into a state in F, then it stops there and then. In this case we say it *halts and succeeds,* and the *output* is whatever is left on the tape, from square 0 up to (but not including) the first blank.
2. Sometimes there may be *no applicable instruction* in a given state when a particular symbol is read, because the pair <current_state; current_symbol> does not occur in the instruction table δ. If so, the TM is stuck: we say that it *halts and fails.* The output is *undefined.*
3. If the head is over square 0 of the tape and tries to move left along the tape, we count it as 'no applicable instruction' (because there is no tape square to the left of square 0, so the TM is stuck again). So in this case the machine also *halts and fails*. Again, the output is undefined.

Of course the machine may never halt — it may go on running forever. If so, the output is again undefined. Eg., it may be writing the decimal expansion of π on the tape 'to the last place' (there is a Turing machine that does this). Or it may get into a *loop:* i.e., at some point of the run, its 'configuration' (state, tape and head position) are exactly the same as at some earlier point, so that from then on, the same configurations will recycle again, forever. (A machine computing π never does this, as the tape keeps changing as more digits are printed. It never halts, but it doesn't loop, either.)

---

3 In Part III we drop this condition!

1.3.5. Summary

The Turing machine has a 1-way infinite tape, a read/write head, and a finite set of states. It looks at its state and reads the current square, and then writes, moves and changes state according to its instruction table. Get-State, Read, Write, Move, Next-State. It does this over and over again, until it halts, if at all. And that's it!

1.4. Why use Turing machines?

Although the Turing machine is based on 1930s technology, we will use it in this course because:

- It fits the requirements that the formalisation of *algorithm* should be precise and simple. (We'll make it even more precise in §2.) Its generality will be discussed when we come to Church's thesis — the architecture of the Turing machine allows strong intuitive arguments here.
- It remains the most common formalisation of *algorithm.* Researchers, research literature and textbooks usually use Turing machines when a formal definition of computability is needed, so after this course you'll be able to understand them better.
- It is the standard benchmark for reasoning about the time or space used by an algorithm (see part III).
- It crops up in popular material such as articles in New Scientist and Scientific American, and books by the likes of D. Hofstadter.
- It is now part of the computing culture. Its historical importance is great and every computer scientist should be familiar with it.

Why not adopt (say) a Cray YMP as our model? We could, but it would be too complex for our work. Our aim here is to study the concept of computability. We are concerned with which problems can be solved in principle, and not (yet) with practicality. So a very simple model of a computer, whose workings can be explained fully in a page or two, is better for us than one that takes many manuals to describe, and may have unknown bugs in it. And one can prove that a TM can solve exactly the same problems as an *idealised* Cray with unlimited memory!

1.4.1. How and why is a TM idealised?

A TM is an *idealised computer,* because the amounts of time and tape memory that it is allowed to use are unbounded. This is not to say that it can use infinitely much time or memory. It can't (unless it runs forever, eg. when it 'loops'). Think of a computer with infinitely many disk drives and RAM chips, which we allow to work on a problem for many years or even centuries. However long it runs for, at the end it will have executed only finitely many instructions. Because it can access only a finite amount of memory per instruction, on termination it will only have used a finite amount of disk space and RAM. But if we only gave it a fixed finite number of disks, if it ran for long enough it might fill them all up and run out of memory.

So our idealisation is this: only finitely much memory and time will get to be used in any given calculation, or *run*; but we set no limit on how much can be used.

We make these idealisations because our notion of *algorithm* should not depend on the state of technology, or on our budgets. For example, the function $f(x) = x^2$ on integers is intuitively computable by al-Khwārazmi's multiplication algorithm, although no existing computer could compute it for $x > 10^{10^{20}}$ (say). A TM can compute x^2 for all integers x, because it can use as much time and memory as it needs for the x in question. So idealising gives us a better model.

Nonetheless, the notion of being computable using only so much time or space is an important refinement of the notion of *computable*. It gives us a formal measure of the *complexity* (difficulty) of a problem. In Part III we will examine this in detail.

1.5. CHURCH'S THESIS

Why should we believe — with Church and Turing — that such a primitive device is a good formalisation of *algorithm* and could calculate not only all that a modern computer can, but anything that is in principle calculable?

First, is there anything to formalise at all? Maybe any definition of algorithm has exceptions, and there are exceptions to the exceptions, and so on. In fact, this seems not to be so. Though the Turing machine looks very different to Church's alternative formalisation of *algorithm*[4], exactly the same things turned out to be algorithms under either definition! Their definitions were *equivalent*.

Now if two people independently put forward different-looking definitions of *algorithm* that turn out to be equivalent, we can take it as evidence for the correctness of both. Such a 'coincidence' hints that there is in nature a genuine class of things that are algorithms, one that fits most definitions we offer.

4 Alonzo Church (c. 1935) used the lambda calculus — the basis of LISP.

This and other considerations (below) made Church put forward his famous Thesis, which says that the definition is the correct one.

This is also known as the Church-Turing thesis, and, when phrased in terms of Turing machines, it is certainly argued for in Turing's 1936 paper, which was written without knowing Church's work. But the shorter title is more common, though less just.

1.5.1. What does Church's thesis say?

Roughly, it says: A problem can be solved by an algorithm if and only if it can be solved by a Turing machine. More formally, it says that a function is computable if and only if it is computable by a Turing machine.

1.5.2. What does it mean?

When we see Turing machines in action below, it will be clear that each one implements an algorithm (because we know an algorithm when we see one)[5]. So few people would reject the *if* direction (←) of the thesis. The heart of the thesis lies in the *only if* (→) direction: every algorithmically-solvable problem can be solved by a Turing machine.

It is important to understand the status of this statement. It is not a theorem. It cannot be proved: that's why it's called a thesis.

Why can't we prove it? Is it that there are infinitely many algorithms (obviously), so to check that each of them can be implemented by a Turing machine would take infinitely long and so is impossible? No! I agree that if there were finitely many algorithms, we could in principle check that each one can be implemented by a Turing machine. But the fact that there are infinitely many is not of itself a fatal problem, as there might be other ways of showing that every algorithm can be implemented by a Turing machine than just checking them one by one. It is not impossible to reason about infinite collections. Compare: there are infinitely many right triangles; but we have little difficulty in showing (some!) properties of all of them, such as "the square of the hypotenuse is equal to the sum of the squares of the other two sides".

No; the real problem is that, although the notion of a Turing machine is completely precise (we will give a mathematical definition below), we have seen that the notion of an algorithm is an intuitive, informal one, with roots going back two thousand years. We can't prove Church's thesis, because it is not — cannot be — stated precisely enough.

5 Some would say that a Turing machine only implements an algorithm if we can be sure that its computation will terminate, or even that we know how long it will take.

Instead, Church's thesis is more like a definition of *algorithm*. It says: 'Here is a mathematical model', and it asks us to accept — and in this course we do accept this — that any algorithm that we could possibly imagine fits the model and could be implemented by a Turing machine.

So Church's thesis is the claim that the Turing machine is a fully general formalisation of *algorithm.*

This is rather analogous to a scientific theory. For example, Newton's theory of gravity says that gravity is an attractive force that acts between any two bodies and depends on their masses and the square of the distance separating them. This formalises our intuitive idea of gravity, and the formalisation has been immensely useful. But we could not prove it correct.

Of course, Newton's theory of gravity was falsified by experiment. In the same way, Church's thesis could in a sense be disproved, if we found something that intuitively was an algorithm but that we could prove was not implementable by a Turing machine. We would then have to revise the thesis.

1.5.3. Evidence for the thesis

Given a new scientific theory, we would check its predictions by experimenting, and conduct 'thought experiments' to study its consequences. Since Church's thesis formalises the notion of *algorithm,* which is absolutely central to computer science, we had better examine carefully the evidence for its correctness. This evidence also depends on 'observations' and 'thought experiments'. In Turing's original paper, three kinds of evidence are suggested:

(a) Giving examples of large classes of numbers which are computable.
(b) A proof of the equivalence of two definitions (in case the new definition has a greater intuitive appeal).
(c) A direct appeal to intuition.

Let us examine these.

(a). Turing machines can do a wide range of algorithmic-like activities. They can compute arithmetical and logical functions, partial derivatives, do recursion, etc. In fact, no-one has yet found an algorithm that cannot be implemented by a Turing machine.

(b). All other suggested definitions of algorithm have turned out to be equivalent (in a precise sense) to Turing machines. These include:

- Software: the lambda-calculus (due to Church), production systems (Emil Post), partial recursive functions (Stephen Kleene), present-day computer languages

- Hardware: register or counter machines, idealisations of our present-day computers, idealised parallel machines, and idealised neural nets.

They all look very different, but can solve (at best) precisely the same problems as Turing machines. As we will see, various souped-up versions of the Turing machine itself — even *non-deterministic* variants — are also equivalent to the basic model.

The essential features of the Turing machine are:

- its computations work in a discrete way, step by step, acting on only a finite amount of information at each stage
- it uses finite but unbounded storage.

Any model with these two features leads to an equivalent definition of *algorithm.*

(c). There are intuitive arguments that any algorithm could be implemented by a Turing machine. In his paper, Turing imagines someone calculating (computing) by hand.

> 'It is always possible for the computer to break off from his work, to go away and forget all about it, and later to come back and go on with it. If he does this he must leave a note of instructions (written in some standard form) explaining how the work is to be continued. … We will suppose that the computer works in such a desultory manner that he never does more than one step at a sitting. The note of instructions must enable him to carry out one step and write the next note. Thus the state of progress of the computation at any stage is completely determined by the note of instructions and the symbols on the tape. … This [combination] may be called the "state formula". We know that the state formula at any given stage is determined by the state formula before the last step was made, and we assume that the relation of these two formulae is expressible. In other words we assume that there is an axiom A which expresses the rules governing the behaviour of the computer, in terms of the relation of the state formula at any stage to the state formula at the preceding stage. If this is so, we can construct a machine to write down the successive state formulae, and hence to compute the required number.' (p.253-4).

So for Turing, any calculation that a person can do on paper could be done by a Turing machine: type (c) (ie. intuitive) evidence for Church's thesis. He also showed that π, e, etc. can be printed out by a TM (type (a) evidence), and in an appendix proved the equivalence to Church's λ-calculus formalisation (type (b)).

1.5.4. Other paradigms of computing

We can vary the notion of *algorithm* by dropping the requirement that it must take only finitely many steps. This leads to new notions of a *problem*, such as the 'problem' an operating system or word processor tries to solve, and has given rise to work on *reactive systems.* These are not supposed to terminate with an answer, but to keep running forever; their behaviour over time is what is of interest.

Is Church's thesis really true then? Can Turing machines do interactive work? Well, as the 'specification' for an interactive system corresponds to a function whose input and output are 'infinite' (as the interaction can go on forever), the Turing machine model needs modifying. <u>But the basic Turing machine hardware is still adequate — it's only how we use it that changes.</u> For example, every time the Turing machine reaches a halting state, we might look at its output, overwrite it with a new input of our choice (depending on the previous output), and set it off again from the initial state. We could model a word processor like this. The collection of all the inputs and outputs (the 'behaviour over time/at infinity') is what counts now. This is research material and is beyond the scope of the course. See Harel's book for more information on reactive systems.

1.6. SUMMARY OF SECTION

We viewed computers as implementing (running) algorithms. We gave a worrying 'paradox' in a Modula_2-like language. To find out how serious it is for computing, we needed to make the notion of *algorithm* completely precise (formal). We discussed early algorithms, and Hilbert's question which prompted the formalising of the vague, intuitive notion of *algorithm*. Turing's formalisation was via *Turing machines*, and we explained what a Turing machine is. We finally discussed *Church's thesis,* saying that Turing machines can implement <u>any</u> algorithm. Since this is really a definition so can't be proved, we looked at evidence for it.

2. Turing machines and examples

We must now define Turing machines more precisely, using mathematical notation. Then we will see some examples and programming tricks.

2.1. WHAT EXACTLY IS A TURING MACHINE?

<u>Definition</u>: A Turing machine is a 6-tuple $M = (Q,\Sigma,I,q_0,\delta,F)$, where:

Q is a finite non-empty set. The elements of Q are called *states*.

Σ is a finite set of at least two elements or symbols. Σ is called the *alphabet* of M. We require that $\wedge \in \Sigma$.

I is a non-empty subset of Σ, with $\wedge \notin I$. I is called the *input alphabet* of M.

$q_0 \in Q$. q_0 is called the *starting state,* or *initial state.*

$\delta : (Q\backslash F)\times\Sigma \to Q\times\Sigma\times\{-1,0,1\}$ is a partial function, called the *instruction table* of M. ($Q\backslash F$ is the set of all states in Q but not in F.)

F is a subset of Q. F is called the set of *final* or *halting states* of M.

2.1.1. Explanation

Q, Σ, q_0 and F are self-explanatory, and we'll explain I in §2.2.2 below. Let us examine the instruction table δ. If q is the current state and s the character of Σ in the current square, $\delta(q,s)$ (if defined) will be a triple $(q',s',d) \in Q\times\Sigma\times\{0,\pm1\}$. This represents the <u>instruction</u> to make q' the next state of M, to write s' in the old square and to move the head in direction d: -1 for left, 0 for no move, +1 for right.

So the line

q s q' s' d

of the 'instruction table' of §1.3.3.1 is represented formally as

$$\delta(q,s) = (q',s',d).$$

We can represent the entire table as a partial function δ in this way, by letting δ(first two symbols) = last three symbols, for each line of the table. The table and δ carry the same information. Functions are more familiar mathematical objects than 'tables', so it is now standard to use a function for the instruction table. But it is not essential: Turing used tables in his original paper.

Note that $\delta(q,s)$ is undefined if $q \in F$ (why?). Also, δ is a <u>partial</u> function: it is undefined for those arguments (q,s) that didn't occur in the table. So it's OK to write $\delta : Q\times\Sigma \to Q\times\Sigma\times\{-1,0,1\}$, rather than $\delta : (Q\backslash F)\times\Sigma \to Q\times\Sigma\times\{-1,0,1\}$, since δ is partial anyway.

2.2. INPUT AND OUTPUT OF A TURING MACHINE

We now have to discuss the <u>tape contents</u> of a TM. First some notation to help us.

2.2.1. Words

A *word* is a finite string of symbols. Example: $w = abaa\wedge\wedge aab\wedge$ is a word. The *length* of w is 10 (note that the blanks '$\wedge$' count as part of the word and contribute to its length). If Σ is a set, a *word of* Σ is a finite string of elements of Σ. We write Σ^* for the set of all words of Σ. So the above word w is in $\{a,b,c,\wedge\}^*$, even though c is not used. Remember: a *word* of Σ is an *element* of Σ^*. There is a unique word of length 0, and it lies in any Σ^*; we write this *empty word* as ε. Also, each symbol in Σ is already a word of Σ, of length 1.

Clearly if w, w' are words of Σ then we can form a new word of Σ by writing w' straight after w. We denote this *concatenation* by ww', or, when it is clearer, w.w'. We also define well-known functions head: $\Sigma^* \to \Sigma \cup \{\varepsilon\}$ and *tail* : $\Sigma^* \to \Sigma^*$ by: if $s \in \Sigma$ and $w \in \Sigma^*$ then

head(s.w) = s   tail(s.w) = w   head(ε) = tail(ε) = ε

So eg., head(abaa∧∧aab∧) = a

tail(abaa∧∧aab∧) = baa∧∧aab∧

2.2.2. The input word

A Turing machine $M = (Q,\Sigma,I,q_0,\delta,F)$ starts a run with its head positioned over square 0 of the tape. Left-justified on the tape is some word w of I. Recall that I is the input alphabet of M, and does not contain $\wedge$. So w contains no blanks.

So for example, if the word is $w = w_0w_1w_2\ldots w_{n-1} \in I^*$, then w_0 goes in square 0, w_1 in square 1, and so on, up to square n-1. The rest of the tape (squares n, n+1, n+2, …) contains only blanks. The contents of the tape are:

Tape							
	w_0	w_1	w_2	…	w_{n-1}	$\wedge$	$\wedge$
	0	1	2		n-1	n	n+1

2.1 Tape with contents w

The word w is the *input* of M for the coming run. It is the initial data that we have provided for M.

Note that w can have any finite length ≥0. M will probably want to read all of w. How does M know where w ends? Well, M can just move its head rightwards until its head reads a blank '$\wedge$' on the tape. Then it knows it has reached the end of the input. This is why w must contain no blanks, and why the remainder of the tape is filled up with blanks ($\wedge$). If w were allowed to have blanks in it, M could not tell whether it had reached the end of w. Effectively, $\wedge$ is the 'end-of-data' character for the input.

Of course, M might put blanks anywhere on the tape when it is running. In fact it can write any letters from Σ. The extra letters of $\Sigma \setminus I$ are used for scratch work, and we call them *scratch characters.*

2.2.3. Output of a Turing machine

Like the input, the *output* of a Turing machine $M = (Q,\Sigma,I,q_0,\delta,F)$ is a word in Σ^*. The output depends on the input. Just as the input is what is on the tape to begin with, so the output is what is on the tape at the end of the run, up to but not including the first blank on the tape — assuming M halts successfully. If, on a certain input, M halts and fails, or does not halt, then the output for that input is undefined.

Recall that at each stage, only finitely many characters on the tape are non-blank. So the output is a finite word of Σ^*. It can be the empty word, or involve symbols from Σ that are not in I, but it never contains $\wedge$.

2.2.3.1. Exercise

Consider the Turing machine $M = (\{q_0,q_1,q_2\},\{1,\wedge\},\{1\},q_0,\delta,\{q_2\})$, with instruction table:

q_0	1	q_1	$\wedge$	1
q_0	$\wedge$	q_2	$\wedge$	0
q_1	1	q_0	1	1

So δ is given by: $\delta(q_0,1) = (q_1,\wedge,1)$, $\delta(q_0,\wedge) = (q_2,\wedge,0)$, $\delta(q_1,1) = (q_0,1,1)$.

List the successive configurations of the machine and tape until M halts, for inputs 1111, 11111 respectively. What is the output of M in each case?

2.2.4. Definition: the input-output function of M

Given a Turing machine $M = (Q,\Sigma,I,q_0,\delta,F)$, we can define a partial function $f_M : I^* \to \Sigma^*$ by:

$f_M(w)$ is the output of M when given input w. The function f_M is called the *input-output function* of M, or the *function computed by* M.

f_M is a partial function. It is not defined on any word w of I^* such that M halts and fails or does not halt when given input w.

2.2.4.1. Exercise

Let M be in the previous exercise. Let 1^n abbreviate 111…1 (n times). For which n is $f_M(1^n)$ defined?

2.2.5. Church's thesis formally

Let I, J be any alphabets (finite and non-empty). Let A be some algorithm all of whose inputs come from I^* and whose outputs are always in J^*. (For example, if A is al-Khwārazmi's decimal addition algorithm, then we can take I and J to be {0,1,…,9}.) Consider a Turing machine $M = (Q,\Sigma,I,q_0,\delta,F)$, for some Σ containing I and J. We say that M *implements* A if for any word w $\in I^*$, if w is given to A and to M as input, then A has an output if and only if M does, and in that case their output is the same. If you like, M computes the same function as A.

We can now state Church's thesis formally as follows:

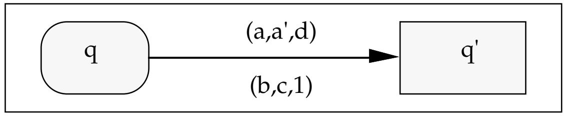
'Given any algorithm, there is some Turing machine that implements it.' Or: 'Any algorithmically computable function is Turing-computable — computable by some Turing machine.' Or: 'for any finite Σ and any function $f : \Sigma^* \to \Sigma^*$, f is computable iff there is a Turing machine M such that $f = f_M$'.

This is formal, but it is still vague, as the intuitive notion of 'algorithm' is still (has to be) involved.

2.3. Representing Turing machines

2.3.1. Flowcharts of Turing machines

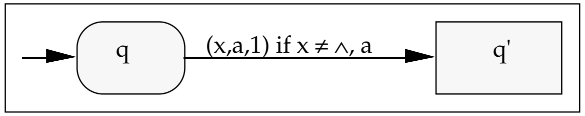
Written as a list of 5-tuples, the instruction table δ of a TM M can be hard to understand. We will often find it easier to represent M as a *graph* or *flowchart*. The nodes of the flowchart are the states of M. We use square boxes for the final states, and round ones for other states.



2.2 part of a flowchart of a TM

The arrows between states represent the instruction table δ. Each arrow is labelled with one or more triples from $\Sigma \times \Sigma \times \{-1,0,1\}$. If one of the labels on the arrow from state q to state q' is (a,a',d), this means that if M reads a from the current square while in state q, it must write a', then take q' as its new state, and move the head by d (+1 for right, 0 for 'no move', and -1 for left). Thus, for each $(q,a) \in Q \times \Sigma$, if $\delta(q,a) = (q',a',d)$ then we draw an arrow from state q to state q', labelled with (a,a',d).

By allowing multiple labels on an arrow, as in Fig. 2.2, we can combine all arrows from q to q' into one. We can attach more than one label to an arrow either by listing them all, or (shorthand) by using a *variable* (s,t,x,y,z, etc.), and perhaps attaching conditions. So for example, the label '(x,a,1) if $x \neq \wedge$, a' from state q to state q' in Fig. 2.3 below means that when in state q, if any symbol other than $\wedge$ or a is read, then the head writes a and moves right, and the state changes to q'. It is equivalent to adding lots of labels (b,a,1), one for each $b \in \Sigma$ with $b \neq \wedge$, a.



2.3 labels having variables

The starting state is indicated by an (unlabelled) arrow leading to it from nowhere (so q is the initial state in Fig. 2.3). All other arrows must have labels.

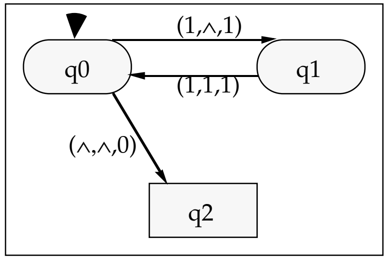
2.3.1.1. Exercises

1. No arrows leave any final state. How does this follow from the definition in §2.1? Can there be a non-final (ie. round) state from which no arrows come, and what would happen if the TM got into such a state?

2. Fig. 2.4 is a flowchart of the Turing machine of the exercises (§§2.2.3.1, 2.2.4.1) above. Try doing the exercises using the flowchart. Is it easier?

2.3.1.2. Warning

Because δ is a function, each state of a flowchart should have no more than one arrow labelled (a,?,?), for any $a \in \Sigma$. And if you forget an arrow or label, the machine might halt and fail wrongly.



2.4 Simple TM as a flowchart

2.3.2. Turing machines as pseudo-code

Another way of representing a Turing machine is in an imperative *pseudo-computer language.* The language is not a formal one: its syntax is usually made up as appropriate for the problem in hand. The permitted basic operations are only Turing machine reads, writes and head movements. However, rather complicated control structures are allowed, such as *if-then* statements and *while* and *until* loops. A Turing machine usually implements if-then statements by using different states. It implements loops by repeatedly returning to the same state.

2.3.2.1. Warning

Pseudo-code makes programming Turing machines less repetitive, as if-then structures etc. are needed very frequently. Many-track and many-tape machines (see later) are represented more easily.

However, there is a risk when writing pseudo-code that we depart too far from the basic state-changing idea of the Turing machine. The code must represent a real Turing machine. Whatever code we write, we must always be <u>sure</u> that it can <u>easily</u> be turned into an actual Turing machine. Assuming Church's thesis, this will always be possible; but <u>it should always be obvious how to do it</u>. For example,

```
solve the problem
halt & succeed
```

is not acceptable pseudocode, nor is

```
count the number of input squares
if it is even then halt and succeed else halt and fail
```

Nested loops should also be watched for — how are they implemented?

Halting: include a statement for halt & succeed, as above. For halt & fail, include a 'halt & fail' statement explicitly, or just arrange that no instruction is applicable.

2.3.3. Example: deleting characters

Fix an alphabet I. Let us define a TM M with:

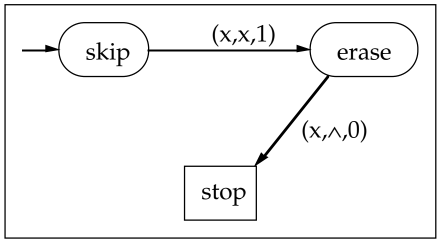
$$f_M(w) = \text{head}(w) \text{ for each } w \in I^*,$$

where the function *head* is as in §2.2.1. M will have three states: *skip, erase,* and *stop*. So Q = {*skip, erase, stop*}. *Skip* is the start state, and *stop* is the only halting state. We can take the alphabet Σ to be $I \cup \{\wedge\}$. δ is given by:

$\delta(skip,x) = (erase, x, 1)$ for all $x \in \Sigma$

$\delta(erase,x) = (stop, \wedge, 0)$ for all $x \in I$

So $M = (Q,\Sigma,I,skip,\delta,\{stop\})$. M can be pictured like this:



2.5 machine for *head(w)*

The <u>names</u> of the states are not really needed in a flowchart, but they can make it more readable.

In pseudo-code:

```
move right
write ∧
halt & succeed
```

Note the <u>close</u> correspondence between the two versions.

We did not need to erase the entire input word, because the output of a Turing machine is defined (§1.3.4, 2.2.3) to be the characters on the tape <u>up to one square before the first blank</u>. Here, we made square 1 blank, so the output will consist of the symbol in square 0, if it is not blank, or ε if it is.

2.3.3.1. Exercise: Unary notation, unary addition

We can represent the number n on the tape by 111…1 (n times). This is *unary notation.* So 0 is represented by a blank tape, 2 by two 1's followed by blanks, etc. For short, we write the string 111…1 of n 1's as 1^n. In this course, 1^n will NOT mean 1x1x1…x1 (n times). Note: 1^0 is ε.

Suppose I = {1,+}. Draw a flowchart for a Turing machine M with input alphabet I, such that $f_M(1^n.+.1^m) = 1^{n+m}$. (Eg., if the input is '111+11', the output is '11111'.) So M adds unary numbers. (There is a suitable machine with 4 states. Beware of the case n = m = 0.) Write a pseudo-code version of M.

2.4. <u>EXAMPLES OF TURING MACHINES</u>

We will now see more examples of Turing machines. Because a Turing machine is so simple, programming one can be a tedious matter. Fortunately, over the years TM hackers have hit upon several useful shorthand devices. The examples will illustrate some of these 'programming techniques' for TMs. They are:

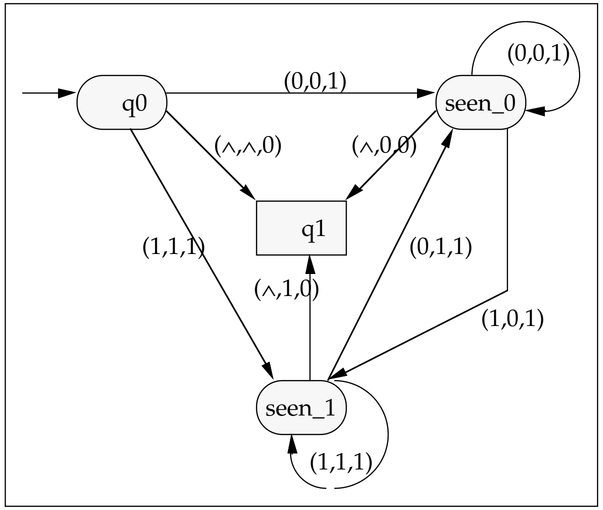
- storing finite amounts of data in the state
- multi-track tapes
- subroutines

Warning: These devices are to help the programmer. <u>They involve no change to the definition of a TM</u>. (In §3 we will consider genuine variants of the TM that make for even easier programming — though these are no more powerful in theory, as we would expect from Church's thesis.)

2.4.1. Storing a finite amount of information in the state

This is a very useful technique. First an example.

2.4.1.1. *Shifting a word to the right*



2.6 a shifter: $f_M(w) = head(w).w$

We want a Turing machine M such that $f_M = head(w).w$ for all $w \in \{0,1\}^*$. So M shifts its input one square to the right, leaving the first character alone. See Fig. 2.6 for a solution.

2.4.1.2. *Generalising the shifting machine*

The M above only works for inputs in $\{0,1\}^*$, but we could design a similar machine $M_I = (Q_I, I\cup\{\wedge\}, I, q_0, \delta_I, F_I)$ to shift a word of I^* to the right, where I is <u>any</u> finite alphabet. If I has more than 2 symbols then M_I would need more states than M above (how many?). But the idea will be the same for each I, so we would like to express M_I <u>uniformly</u> in I.

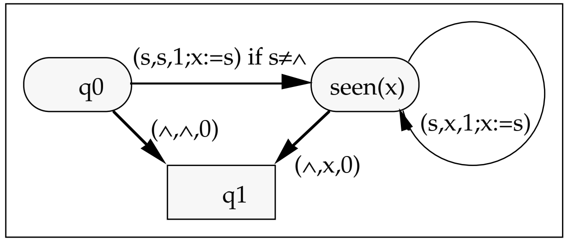
<u>Suppose</u> we could introduce into Q_I a special state *seen(x)* with a *parameter*, x that can take any value in I. We could then use x to remember the symbol just read. Using *seen(x)*, the table δ_I can be given very simply as follows:

$\delta_I(q_0,a) = (seen(a),a,1)$ for all a in I

$\delta_I(seen(a),b) = (seen(b),a,1)$ for all a, b in I

$\delta_I(seen(a),\wedge) = (q_1,a,0)$ for all a in I.

Or, in a flowchart:



2.7 the 'shifter' TM drawn using parameters in states

Each arrow leading to *seen(x)* is labelled with one or more 4-tuples. The last entry of each 4-tuple is an 'assignment statement', saying what x becomes when *seen(x)* is entered.

The pseudo-code will use a variable x. x can take only finitely many values. We need not mention the initial write, as we only need specify writes that actually alter the tape.

```
read current symbol & put it into x
move right
repeat until x = ∧
    swap current symbol with x
    move right
end repeat
halt & succeed
```

This will work for any I.

<u>In fact we can use states like *seen(x)* without changing the formal definition of the Turing machine at all!</u> We just observe that whilst it's convenient to view *seen(x)* as a single state with a parameter x, we could equally get away with the collection seen(a), seen(b),… of states, one for each letter in I, if we are prepared to draw them all in and connect all the arrows correctly. This is a bit like multiplication: 3x4 is convenient, but if we only have addition we can view this as shorthand for 3+3+3+3.

What we do is this. For each letter a of I we introduce <u>a single state,</u> called *seen(a)*, or if you prefer, *seen_a*.[6] Because I is finite, this introduces only finitely many states. So the resulting state set is finite, and so is allowed by the definition of a Turing machine. In fact, if $I = \{a_1,\ldots,a_n\}$ then $Q_I = \{q_0, q_1, seen(a_1), \ldots, seen(a_n)\}$: ie. n+2 states in all. Then δ_I as above is just a partial function from $Q_I \times (I\cup\{\wedge\})$ into $Q_I \times (I\cup\{\wedge\}) \times \{0,\pm1\}$. So our machine is $M_I = (Q_I, I, (I\cup\{\wedge\}), q_0, \delta_I, F)$ — a genuine Turing machine!

[6] I prefer the notation *(seen,a)*, since then we can simply define $Q_I = \{q_0,q_1\} \cup \{seen\} \times I$.

So although *seen(x)* is *conveniently viewed by us* as a single state with a parameter ranging over I, *for the Turing machine* it is really many states, namely *seen(a_1), seen(a_2), … seen(a_n)*, one for each element of I.

So we can in effect allow parameters x in the states of Turing machines, so long as x can take only finitely many values. Doing so is just a useful piece of *notation*, to help us write programs. This notation represents the idea of storing a finite amount of information in the state (as in the registers on a computer).

NOTE: we cannot store any parameter x that can take infinitely many values, for that would force the underlying genuine state set Q to be infinite, in contravention of the definition of a Turing machine. So, e.g, for any I we get a Turing machine M_I that works for I. M_I is built in a uniform way, but we do not (can't) get a single Turing machine M that works for any I!

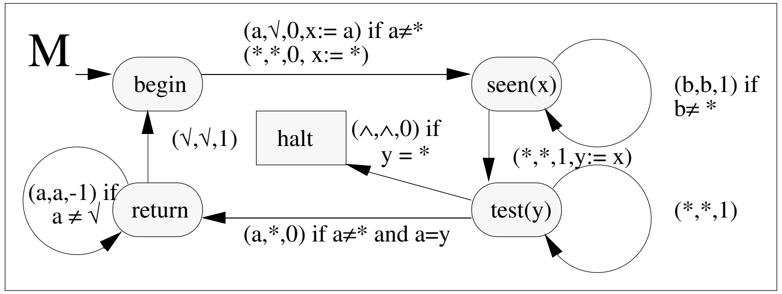
2.4.1.3. *Harder example: testing whether two strings are equal*

We design a Turing machine M with input alphabet I, such that $f_M(w_1,w_2)$ is defined if $w_1 = w_2$ (but we don't care what value it has), and undefined otherwise.

First, how can a TM take more than one argument as input? We saw in (§2.3.3.1) a TM to calculate n+m in unary. Its arguments were 1^n and 1^m, separated by '+'. So here we assume that I contains a delimiter, '*', say, and w_1, w_2 are words of I not containing '*'. I.e., the input tape to M looks like this:

2.8 initial tape of M

We will use a parameter to remember the last character seen. We will also need to tick off characters once we have checked them. So we let M have full alphabet $\Sigma = I \cup \{\wedge, \surd\}$, where √ ('tick') is a new character not in I. We will overwrite each character with √, once we've checked it. So we get:



2.9 TM to check if $w_1 = w_2$

M overwrites the leftmost unchecked character of w_1 with √, passing to state *seen(x)* and remembering what it was using the parameter x of 'seen'. (But if x is *, this means it has checked all of w_1, so it only remains to make sure there are no more uncompared characters of w_2.) Then it moves right until it sees *, when it jumps to state *test(y),* remembering x as y. In this state it moves past all *'s (which are the checked characters of w_2). It stops when it finds a character — a, say — that isn't * (i.e., a is the first unchecked character of w_2). It compares a with y, the remembered character of w_1. There are three possibilities:

- a = ∧ and y = *. So all characters of w_1 have been checked ahgainst w_2 without finding a difference, and w_2 has the same length as w_1. Hence $w_1 = w_2$, so M hals and succeeds (state *halt*).
- a ≠ y. M has found a difference, so it halts & fails (there's no applicable instruction in state *test*(y)).
- a=y and y ≠ *. So the characters match. M overwrites the current character (a) with *, and returns left until it sees a √. One move right then puts it over the next character of w_1 to check, and the process repeats.

Questions: Try M on the inputs 123*123, 12*13, 1*11, 12*1, *1, 1* and * (in the last three, w_1, w_2 or both are empty (ε)). What is the output of M in each case? What would go wrong if the 'begin → seen' arrow was just labelled (a,√,0,x:= a)?

Please don't worry if you found that hard; Turing machines that need as many as 5 states (not counting any parameters) are fairly rare, and anyway we'll soon see ways to make things easier. By the way, it's a good idea to write your Turing machines using as few states as you can.

2.4.1.4. *Exercises for states with parameters*

1. Design a Turing machine T_I to calculate the function *tail* : I* → I*.

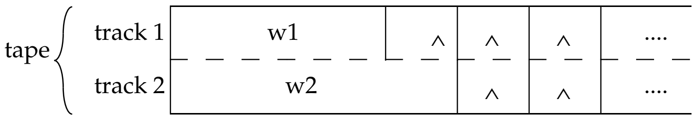
2. Design a Turing machine M that checks that the first character of its input does not appear elsewhere in the input. How will you make M output the answer?

2.4.2. Multiple tracks

Above, we found it convenient to put (finite amounts of) data in the state of a Turing machine. So a state took the form q(x) or (q,x), where x could take any of finitely many values. Then we could specify the instruction table more easily. In the same way, many problems would be simpler to solve with Turing machines if we were allowed to use a tape with more than one *track* — as on a stereo cassette, which has four tracks all told.

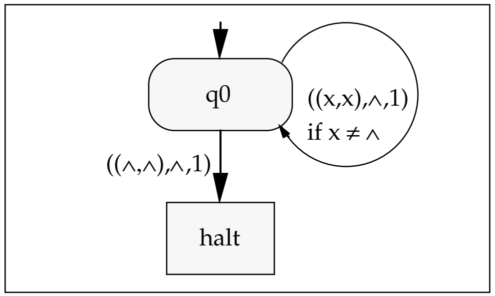
The string comparison example shows how useful this can be. The 'M' of Fig 2.9 above was pretty complex. Wouldn't it be easier to use two tracks? As before, let's cheat for a moment and do this.

We would like the tape of M to have two tracks, with w_1 on the first track and w_2 on the second track:



2.10 two-track tape for word-comparison TM, M

Then M can simply move its head along the tape, testing at each stage whether the characters in tracks 1 and 2 are the same:



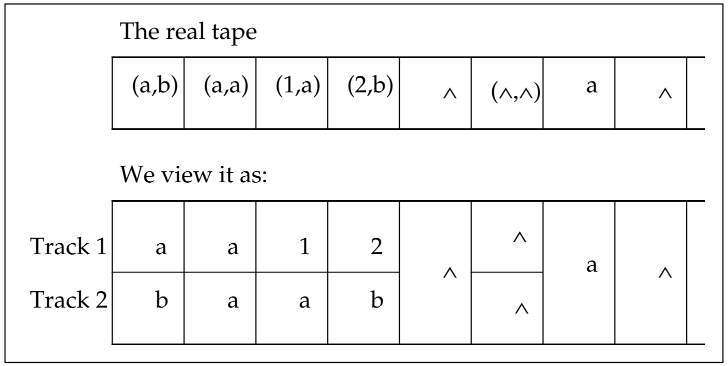
2.11 flowchart for M

In Fig 2.10 we write (x,y) as notation for a square having x in track 1 and y in track 2. M halts and fails if it finds a square with different symbols in tracks 1 and 2.

The two-track M is much easier to design. So it might be useful for Turing machines in general to be able to have a multi-track tape. In fact, as

with states, <u>we can effectively divide the tape into tracks without modifying the formal definition of the Turing machine</u>.

To divide the tape into n tracks we add a finite number of new individual symbols of the form $(a_1,a_2,\ldots,a_n)$ to Σ, where $a_1,\ldots, a_n$ are any symbols. Each $(a_1,a_2,\ldots,a_n)$ is a single symbol, in Σ, and may be written to and read from the tape as usual. But whenever $(a_1,a_2,\ldots,a_n)$is in a square, we can <u>view</u> this square as divided into n parts, the i^{th} part containing the 'single' symbol a_i. So if n=2 say, and many squares have pairs of the form (x,y) in them, the tape begins to look as though it is divided into two tracks:



2.12 tracks on tape

If the only symbols on the tape are $\wedge$ and symbols of the form $(a_1,a_2,\ldots,a_n)$, we can consider the tape as actually divided into n tracks. Note that $\wedge \neq (\wedge,\wedge)$.

2.4.2.1. WARNING:

The tuples $(a_1,a_2,\ldots,a_n)$ are just single symbols in the Turing machine's alphabet. The tracks only help us to think about Turing machine operations — they exist only in the mind of the programmer. No change to the definition of a Turing machine has been made.

Compare <u>arrays</u> in an ordinary computer. The array A(5,6) will usually be implemented as a 1-dimensional memory area of 30 contiguous cells. The division into a 2-dimensional array is done in software.

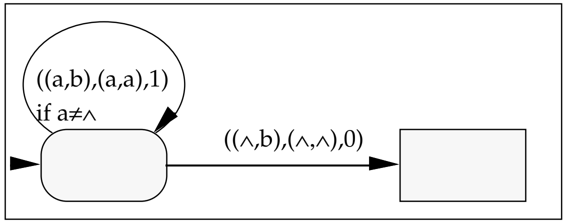
Note that we cannot divide the tape into infinitely many tracks — this would violate the requirement that Σ be finite. (But see 2-dimensional Turing machines in §3.)

2.4.2.2. What we can do with tracks

Because a Turing machine can write and move according to exactly what it reads, it can effectively read from and write to the tracks <u>independently</u>.

Thus eg. it can shift a single track right by one square (cf. §2.4.1.1). In fact, anything we can do with a 1-track machine we can also do on any given track of a multi-track machine.

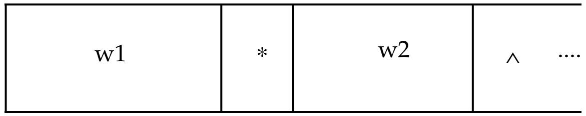
Cross-track operations are also possible. For example, this Turing machine copies track 1 as far as its first blank to track 2:



2.13 track copier

2.4.2.3. String comparison revisited

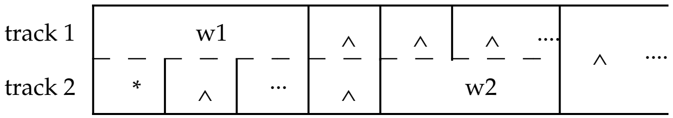
Now let's see in detail how to solve the string comparison problem using 2 tracks. The input is w_1*w_2 as before: all on one track.



2.14 Initial tape contents

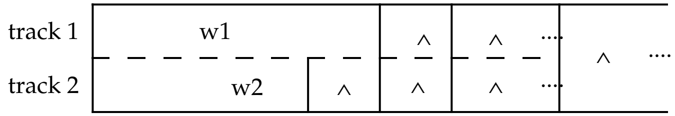
The Turing machine we want will have three stages:

• Stage 1: replace the 1-track input by 2 tracks, with w_1 left-justified on track 1, and w_2 , with len(w_1)+1 blanks before it, on track 2). This part can be done much as in Fig 2.13 (how exactly?) Then return to square 0. The resulting tape has 2 tracks as far as the input went; after that, it has only one. Also, while we're at it, we mark square 0 with a '*' in track 2:



2.15 Tape after Stage 1

• Stage 2: shift w_2 left to align it with w_1. E.g., use some version of *tail* (Ex. 2.1.4.3) repeatedly, until the * is gone.



2.16 Tape after Stage 2

• Stage 3: compare the tracks as far as their first ∧'s, halting & failing if a difference is found. This is easy.

Exercise: work out the details.

So compring two words is easier with 2 tracks. But tapes with more than one track are useful even if there's only one input. Here's an example (• • below); we'll see others in §3.

2.4.3. Implicit marking of square 0 of the tape

As our Turing machines halt and fail if they leave the left hand end of the tape, it helps when programming to be able to tell when the head is in square 0. We have seen the need for this in the examples. We want to mark square 0 with a special symbol, '*', say. Then when the head reads '*', we know it is in square 0.

But square 0 may contain an important symbol already, which would be lost if we overwrote it with '*'.

There are several ways to manage here:

- • • Create an <u>extra track</u>, with '*' in square 0 and blanks in the remaining squares. To see if the head is in square 0, just read the new track.
- • For each a in Σ add a <u>new character</u> a* (or (a,*)) to Σ. To initialise, replace the character b in square 0 by b*. From then on, write a starred character iff you read one. So square 0 is always the only square with a starred character. This is much the same as adding an extra track.
- • Include * as a <u>special character</u> of Σ. To initialise, shift the input right one place and insert * in square 0. Then carry out all operations on squares 1,2,.., using * as a left end marker. This involves some tedious copying but works OK. Not recommended!
- • Write your TM carefully so it doesn't need to return to square 0. This is possible surprisingly often, but few can be bothered to do it.

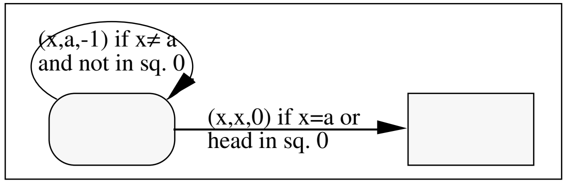
2.4.3.1. CONVENTION

Because we can always know when in square 0 (by using one of these ways), we will assume that *a Turing machine always knows when the head is over square 0 of the tape. Square 0 is assumed to be implicitly marked.* This saves us having to mention the extra track explicitly when describing the machine, and so keeps things simple.

2.4.3.2. *Examples of implicit marking (<u>fragments</u> of TMs)*

```
repeat until read a or square 0 reached
    write a
    move left
end repeat
halt & succeed
```

Or in a flowchart:



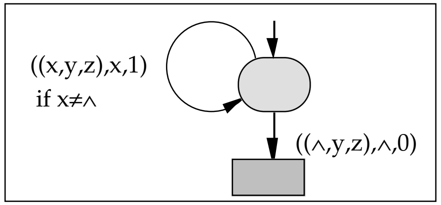
2.17 implicit marking of sq. 0 in flowcharts

2.4.3.3. *Setting up and removing tracks*

In the string comparison example (§2.4.2.3) above, the two arguments w_1, w_2 were provided on a 1-track tape, one after the other (Fig. 2.8). We then put them on different tracks (Fig. 2.14). If there were 16 arguments, we could put them left-justified on a 16-track tape in a similar way (think about how to do it).

But often it is best to set up tracks *dynamically:* i.e., as we go along. This saves doing it all at the beginning. (Besides, however much of the tape we set up as 2 tracks initially, we might want to use even more of the tape later, so every so often we'd have to divide more of the tape into tracks, which is messy.) So, each time our machine enters a square that is <u>not</u> divided into 2 tracks (i.e., doesn't have a symbol of the form (a,b) in it), it immediately replaces the symbol found — a, say — by the pair $(a,\wedge)$, and then carries on. This is so easy to do (just add instructions of the form $(q,a,q,(a,\wedge),0)$ to δ, for all non-pairs $a\in\Sigma$) that we won't often mention the setting up of tracks explicitly.

Similarly, when M has finished its calculations using many tracks, the output will have to be presented on a single track tape, as per the definition of *output* in §2.2.3. Assuming that the answer is on track 1, M will erase all tracks but the first, so that the tape on termination has a single track that looks like track 1 of the 'scratch' tape. It need only do this as far as the first $\wedge$ in track 1. Eg. for a three-track scratch tape, assuming M has its head in square 0:



2.18 returning to a single track

2.4.4. Subroutines

It is quite in order to string several Turing machines together. Informally, when a final state of one is reached, the state changes to the initial state of the next in the chain.

This can be done formally by collecting up the states and instruction tables of all the machines in the chain, and for each final state of one machine, adding a new instruction that changes the state to the initial state of the next machine in the chain, without altering the tape or moving the head. The number of states in the 'chain' machine is the sum of the numbers of states for the individual machines, so is finite. Thus we obtain a single Turing machine from the machines in the chain; again we have not changed the basic definition of the Turing machine. We will use this technique repeatedly.

WARNING: when control passes to the next Turing machine in the chain, the head may not be over square 0. Moreover, the tape following the 'input' may contain the previous machine's scratchwork and so not be entirely blank. Each machine's design should allow for these possibilities, eg. by returning the head to square 0 before starting, or otherwise.

2.4.5. Problems

We end this section with some problems that illustrate the techniques of this section.

2.4.5.1. *Exercise: Unary subtraction*

Suppose $I = \{1,-\}$. Design a Turing machine $M = (Q,\Sigma,I,q_0,\delta,F)$ such that $f_M(1^n.-.1^m) = 1^{n-m}$ if $n\geq m$ and ε otherwise. M performs subtraction on unary numbers.

2.4.5.2. *Exercise: Unary multiplication*

Suppose $I = \{1,*\}$. Design a Turing machine M such that $f_M(1^n.*.1^m) = 1^{nm}$. Hint: use 3 tracks and repeated addition and subtraction.

Design a TM that given an input some binary number, tests whether it is prime. Here again, a 3-track machine is useful [see Hopcroft & Ullman, p154].

2.4.5.3. Exercise: Inverting words

Let I be an alphabet. Find a Turing machine $M = (Q,\Sigma,I,q_0,\delta,F)$ such that $f_M(w)$ is the reverse of w. Eg: use storage in states and marking of square 0.

2.4.5.4. Exercise: Unary-binary conversion

Design a machine M to add 1 to a binary number (much easier than in decimal!). That is, if if n>0 is a number let 'n' $\in \{0,1\}^*$ be the binary expansion of n, without leading zeros, written with the least significant digits on the left (say). Define '0' = 0. We then require f_M('n') = 'n+1' for all $n \geq 0$.

Extend M to a machine that converts a unary number to its binary equivalent. Hint: use two tracks.

Design a Turing machine that converts a binary number to unary.

2.5. Summary of Section

We defined a Turing machine formally, as a finite state machine with a finite symbol alphabet and a 1-way infinite tape. We explained why we chose it as our formalisation of *algorithm*, and how it is idealised from a real computer. We discussed Church's thesis.

We explained input and output for Turing machines and defined the input-output function f_M for a Turing machine M. We saw that a Turing machine can be represented as a flowchart or by pseudo-code. We gave examples of Turing machines that solve particular problems: unary-binary conversion, arithmetical operations, etc. We considered ways of programming Turing machines more easily: storing finite amounts of data in the state set (cf. registers), using many tracks on the tape (cf. arrays), and chaining Turing machines (cf. subroutines).