3 Introduction to computer science

In natural science, Nature has given us a world and we're just to discover its laws. In computers, we can stuff laws into it and create a world.
– Alan Kay

Our field is still in its embryonic stage. It's great that we haven't been around for 2000 years. We are still at a stage where very, very important results occur in front of our eyes.
– Michael Rabin, on computer science

Algorithms are the key concept of computer science. An algorithm is a precise recipe for performing some task, such as the elementary algorithm for adding two numbers which we all learn as children. This chapter outlines the modern theory of algorithms developed by computer science. Our fundamental model for algorithms will be the *Turing machine*. This is an idealized computer, rather like a modern personal computer, but with a simpler set of basic instructions, and an idealized unbounded memory. The apparent simplicity of Turing machines is misleading; they are very powerful devices. We will see that they can be used to execute any algorithm whatsoever, even one running on an apparently much more powerful computer.

The fundamental question we are trying to address in the study of algorithms is: what resources are required to perform a given computational task? This question splits up naturally into two parts. First, we'd like to understand what computational tasks are possible, preferably by giving explicit algorithms for solving specific problems. For example, we have many excellent examples of algorithms that can quickly sort a list of numbers into ascending order. The second aspect of this question is to demonstrate *limitations* on what computational tasks may be accomplished. For example, lower bounds can be given for the number of operations that must be performed by any algorithm which sorts a list of numbers into ascending order. Ideally, these two tasks – the finding of algorithms for solving computational problems, and proving limitations on our ability to solve computational problems – would dovetail perfectly. In practice, a significant gap often exists between the best techniques known for solving a computational problem, and the most stringent limitations known on the solution. The purpose of this chapter is to give a broad overview of the tools which have been developed to aid in the analysis of computational problems, and in the construction and analysis of algorithms to solve such problems.

Why should a person interested in *quantum* computation and *quantum* information spend time investigating *classical* computer science? There are three good reasons for this effort. First, classical computer science provides a vast body of concepts and techniques which may be reused to great effect in quantum computation and quantum information. Many of the triumphs of quantum computation and quantum information have come by combining existing ideas from computer science with novel ideas from quantum

mechanics. For example, some of the fast algorithms for quantum computers are based upon the Fourier transform, a powerful tool utilized by many classical algorithms. Once it was realized that quantum computers could perform a type of Fourier transform much more quickly than classical computers this enabled the development of many important quantum algorithms.

Second, computer scientists have expended great effort understanding what resources are required to perform a given computational task on a classical computer. These results can be used as the basis for a comparison with quantum computation and quantum information. For example, much attention has been focused on the problem of finding the prime factors of a given number. On a classical computer this problem is believed to have no 'efficient' solution, where 'efficient' has a meaning we'll explain later in the chapter. What is interesting is that an efficient solution to this problem *is* known for quantum computers. The lesson is that, for this task of finding prime factors, there appears to be a *gap* between what is possible on a classical computer and what is possible on a quantum computer. This is both intrinsically interesting, and also interesting in the broader sense that it suggests such a gap may exist for a wider class of computational problems than merely the finding of prime factors. By studying this specific problem further, it may be possible to discern features of the problem which make it more tractable on a quantum computer than on a classical computer, and then act on these insights to find interesting quantum algorithms for the solution of other problems.

Third, and most important, there is learning to *think* like a computer scientist. Computer scientists think in a rather different style than does a physicist or other natural scientist. Anybody wanting a deep understanding of quantum computation and quantum information must learn to think like a computer scientist at least some of the time; they must instinctively know what problems, what techniques, and most importantly what problems are of greatest interest to a computer scientist.

The structure of this chapter is as follows. In Section 3.1 we introduce two models for computation: the Turing machine model, and the circuit model. The Turing machine model will be used as our fundamental model for computation. In practice, however, we mostly make use of the circuit model of computation, and it is this model which is most useful in the study of quantum computation. With our models for computation in hand, the remainder of the chapter discusses resource requirements for computation. Section 3.2 begins by overviewing the computational tasks we are interested in as well as discusing some associated resource questions. It continues with a broad look at the key concepts of *computational complexity*, a field which examines the time and space requirements necessary to solve particular computational problems, and provides a broad classification of problems based upon their difficulty of solution. Finally, the section concludes with an examination of the energy resources required to perform computations. Surprisingly, it turns out that the energy required to perform a computation can be made vanishingly small, provided one can make the computation reversible. We explain how to construct reversible computers, and explain some of the reasons they are important both for computer science and for quantum computation and quantum information. Section 3.3 concludes the chapter with a broad look at the entire field of computer science, focusing on issues of particular relevance to quantum computation and quantum information.

3.1 Models for computation

> *...algorithms are concepts that have existence apart from any programming language.*
> – Donald Knuth

What does it mean to have an *algorithm* for performing some task? As children we all learn a procedure which enables us to add together any two numbers, no matter how large those numbers are. This is an example of an algorithm. Finding a mathematically precise formulation of the concept of an algorithm is the goal of this section.

Historically, the notion of an algorithm goes back centuries; undergraduates learn Euclid's two thousand year old algorithm for finding the greatest common divisor of two positive integers. However, it wasn't until the 1930s that the fundamental notions of the modern theory of algorithms, and thus of computation, were introduced, by Alonzo Church, Alan Turing, and other pioneers of the computer era. This work arose in response to a profound challenge laid down by the great mathematician David Hilbert in the early part of the twentieth century. Hilbert asked whether or not there existed some algorithm which could be used, in principle, to solve all the problems of mathematics. Hilbert expected that the answer to this question, sometimes known as the *entscheidungsproblem*, would be yes.

Amazingly, the answer to Hilbert's challenge turned out to be no: there is no algorithm to solve all mathematical problems. To prove this, Church and Turing had to solve the deep problem of capturing in a mathematical definition what we mean when we use the intuitive concept of an algorithm. In so doing, they laid the foundations for the modern theory of algorithms, and consequently for the modern theory of computer science.

In this chapter, we use two ostensibly different approaches to the theory of computation. The first approach is that proposed by Turing. Turing defined a class of machines, now known as *Turing machines*, in order to capture the notion of an algorithm to perform a computational task. In Section 3.1.1, we describe Turing machines, and then discuss some of the simpler variants of the Turing machine model. The second approach is via the *circuit model* of computation, an approach that is especially useful as preparation for our later study of quantum computers. The circuit model is described in Section 3.1.2. Although these models of computation appear different on the surface, it turns out that they are equivalent. Why introduce more than one model of computation, you may ask? We do so because different models of computation may yield different insights into the solution of specific problems. Two (or more) ways of thinking about a concept are better than one.

3.1.1 Turing machines

The basic elements of a Turing machine are illustrated in Figure 3.1. A Turing machine contains four main elements: (a) a *program*, rather like an ordinary computer; (b) a *finite state control*, which acts like a stripped-down microprocessor, co-ordinating the other operations of the machine; (c) a *tape*, which acts like a computer memory; and (d) a *read-write tape-head*, which points to the position on the tape which is currently readable or writable. We now describe each of these four elements in more detail.

The *finite state control* for a Turing machine consists of a finite set of *internal states*,

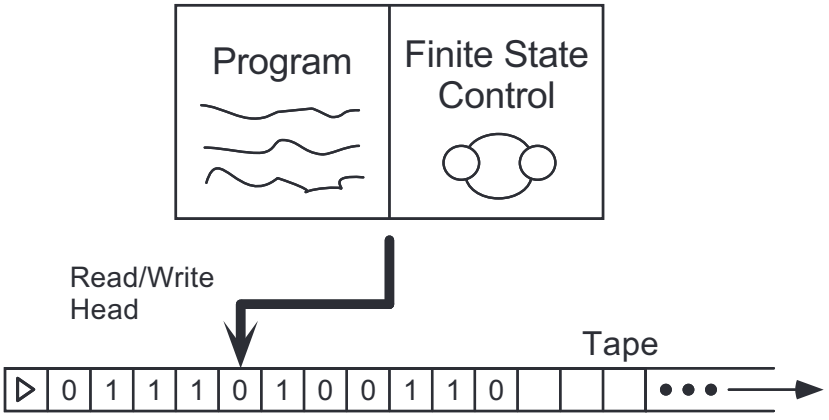


Figure 3.1. Main elements of a Turing machine. In the text, blanks on the tape are denoted by a 'b'. Note the ▷ marking the left hand end of the tape.

$q_1, \ldots, q_m$. The number m is allowed to be varied; it turns out that for m sufficiently large this does not affect the power of the machine in any essential way, so without loss of generality we may suppose that m is some fixed constant. The best way to think of the finite state control is as a sort of microprocessor, co-ordinating the Turing machine's operation. It provides temporary storage off-tape, and a central place where all processing for the machine may be done. In addition to the states $q_1, \ldots, q_m$, there are also two special internal states, labelled $q_{\rm s}$ and $q_{\rm h}$. We call these the *starting state* and the *halting state*, respectively. The idea is that at the beginning of the computation, the Turing machine is in the starting state $q_{\rm s}$. The execution of the computation causes the Turing machine's internal state to change. If the computation ever finishes, the Turing machine ends up in the state $q_{\rm h}$ to indicate that the machine has completed its operation.

The Turing machine *tape* is a one-dimensional object, which stretches off to infinity in one direction. The tape consists of an infinite sequence of *tape squares*. We number the tape squares $0, 1, 2, 3, \ldots$. The tape squares each contain one symbol drawn from some *alphabet*, Γ, which contains a finite number of distinct symbols. For now, it will be convenient to assume that the alphabet contains four symbols, which we denote by 0, 1, b (the 'blank' symbol), and ▷, to mark the left hand edge of the tape. Initially, the tape contains a ▷ at the left hand end, a finite number of 0s and 1s, and the rest of the tape contains blanks. The *read-write tape-head* identifies a single square on the Turing machine tape as the square that is currently being accessed by the machine.

Summarizing, the machine starts its operation with the finite state control in the state $q_{\rm s}$, and with the read-write head at the leftmost tape square, the square numbered 0. The computation then proceeds in a step by step manner according to the *program*, to be defined below. If the current state is $q_{\rm h}$, then the computation has halted, and the *output* of the computation is the current (non-blank) contents of the tape.

A *program* for a Turing machine is a finite ordered list of *program lines* of the form $\langle q, x, q', x', s\rangle$. The first item in the program line, q, is a state from the set of internal states of the machine. The second item, x, is taken from the alphabet of symbols which may appear on the tape, Γ. The way the program works is that on each machine cycle, the Turing machine looks through the list of program lines in order, searching for a line $\langle q, x, \cdot, \cdot, \cdot\rangle$, such that the current internal state of the machine is q, and the symbol

being read on the tape is x. If it doesn't find such a program line, the internal state of the machine is changed to q_{h}, and the machine halts operation. If such a line is found, then that program line is *executed*. Execution of a program line involves the following steps: the internal state of the machine is changed to q'; the symbol x on the tape is overwritten by the symbol x', and the tape-head moves left, right, or stands still, depending on whether s is -1, $+1$, or 0, respectively. The only exception to this rule is if the tape-head is at the leftmost tape square, and $s = -1$, in which case the tape-head stays put.

Now that we know what a Turing machine is, let's see how it may be used to compute a simple function. Consider the following example of a Turing machine. The machine starts with a binary number, x, on the tape, followed by blanks. The machine has three internal states, q_1, q_2, and q_3, in addition to the starting state q_{s} and halting state q_{h}. The program contains the following program lines (the numbers on the left hand side are for convenience in referring to the program lines in later discussion, and do not form part of the program):

$$
\begin{aligned}
1:&\ \langle q_{\mathrm{s}}, \triangleright, q_1, \triangleright, +1\rangle\\
2:&\ \langle q_1, 0, q_1, b, +1\rangle\\
3:&\ \langle q_1, 1, q_1, b, +1\rangle\\
4:&\ \langle q_1, b, q_2, b, -1\rangle\\
5:&\ \langle q_2, b, q_2, b, -1\rangle\\
6:&\ \langle q_2, \triangleright, q_3, \triangleright, +1\rangle\\
7:&\ \langle q_3, b, q_{\mathrm{h}}, 1, 0\rangle.
\end{aligned}
$$

What function does this program compute? Initially the machine is in the state q_{s} and at the left-most tape position so line 1, $\langle q_{\mathrm{s}}, \triangleright, q_1, \triangleright, +1\rangle$, is executed, which causes the tape-head to move right without changing what is written on the tape, but changing the internal state of the machine to q_1. The next three lines of the program ensure that while the machine is in the state q_1 the tape-head will continue moving right while it reads either 0s (line 2) or 1s (line 3) on the tape, over-writing the tape contents with blanks as it goes and remaining in the state q_1, until it reaches a tape square that is already blank, at which point the tape-head is moved one position to the left, and the internal state is changed to q_2 (line 4). Line 5 then ensures that the tape-head keeps moving left while blanks are being read by the tape-head, without changing the contents of the tape. This keeps up until the tape-head returns to its starting point, at which point it reads a $\triangleright$ on the tape, changes the internal state to q_3, and moves one step to the right (line 6). Line 7 completes the program, simply printing the number 1 onto the tape, and then halting.

The preceding analysis shows that this program computes the constant function $f(x) = 1$. That is, regardless of what number is input on the tape the number 1 is output. More generally, a Turing machine can be thought of as computing functions from the non-negative integers to the non-negative integers; the initial state of the tape is used to represent the input to the function, and the final state of the tape is used to represent the output of the function.

It seems as though we have gone to a very great deal of trouble to compute this simple function using our Turing machines. Is it possible to build up more complicated functions using Turing machines? For example, could we construct a machine such that when two numbers, x and y, are input on the tape with a blank to demarcate them, it will

output the sum $x + y$ on the tape? More generally, what class of functions is it possible to compute using a Turing machine?

It turns out that the Turing machine model of computation can be used to compute an enormous variety of functions. For example, it can be used to do all the basic arithmetical operations, to search through text represented as strings of bits on the tape, and many other interesting operations. Surprisingly, it turns out that a Turing machine can be used to simulate all the operations performed on a modern computer! Indeed, according to a thesis put forward independently by Church and by Turing, the Turing machine model of computation *completely captures* the notion of computing a function using an algorithm. This is known as the *Church–Turing thesis*:

The class of functions computable by a Turing machine corresponds exactly to the class of functions which we would naturally regard as being computable by an algorithm.

The Church–Turing thesis asserts an equivalence between a rigorous mathematical concept – function computable by a Turing machine – and the intuitive concept of what it means for a function to be computable by an algorithm. The thesis derives its importance from the fact that it makes the study of real-world algorithms, prior to 1936 a rather vague concept, amenable to rigorous mathematical study. To understand the significance of this point it may be helpful to consider the definition of a *continuous function* from real analysis. Every child can tell you what it means for a line to be continuous on a piece of paper, but it is far from obvious how to capture that intuition in a rigorous definition. Mathematicians in the nineteenth century spent a great deal of time arguing about the merits of various definitions of continuity before the modern definition of continuity came to be accepted. When making fundamental definitions like that of continuity or of computability it is important that good definitions be chosen, ensuring that one's intuitive notions closely match the precise mathematical definition. From this point of view the Church–Turing thesis is simply the assertion that the Turing machine model of computation provides a good foundation for computer science, capturing the intuitive notion of an algorithm in a rigorous definition.

A priori it is not obvious that every function which we would intuitively regard as computable by an algorithm can be computed using a Turing machine. Church, Turing and many other people have spent a great deal of time gathering evidence for the Church–Turing thesis, and in sixty years no evidence to the contrary has been found. Nevertheless, it is possible that in the future we will discover in Nature a process which computes a function not computable on a Turing machine. It would be wonderful if that ever happened, because we could then harness that process to help us perform new computations which could not be performed before. Of course, we would also need to overhaul the definition of computability, and with it, computer science.

Exercise 3.1: (Non-computable processes in Nature) How might we recognize that a process in Nature computes a function not computable by a Turing machine?

Exercise 3.2: (Turing numbers) Show that single-tape Turing machines can each be given a number from the list $1, 2, 3, \ldots$ in such a way that the number uniquely specifies the corresponding machine. We call this number the *Turing number* of the corresponding Turing machine. (*Hint*: Every positive integer has

a unique prime factorization $p_1^{a_1} p_2^{a_2} \ldots p_k^{a_k}$, where p_i are distinct prime numbers, and $a_1, \ldots, a_k$ are non-negative integers.)

In later chapters, we will see that quantum computers also obey the Church–Turing thesis. That is, quantum computers can compute the same class of functions as is computable by a Turing machine. The difference between quantum computers and Turing machines turns out to lie in the *efficiency* with which the computation of the function may be performed – there are functions which can be computed much more efficiently on a quantum computer than is believed to be possible with a classical computing device such as a Turing machine.

Demonstrating in complete detail that the Turing machine model of computation can be used to build up all the usual concepts used in computer programming languages is beyond the scope of this book (see 'History and further reading' at the end of the chapter for more information). When specifying algorithms, instead of explicitly specifying the Turing machine used to compute the algorithm, we shall usually use a much higher level *pseudocode*, trusting in the Church–Turing thesis that this pseudocode can be translated into the Turing machine model of computation. We won't give any sort of rigorous definition for pseudocode. Think of it as a slightly more formal version of English or, if you like, a sloppy version of a high-level programming language such as C++ or BASIC. Pseudocode provides a convenient way of expressing algorithms, without going into the extreme level of detail required by a Turing machine. An example use of pseudocode may be found in Box 3.2 on page 130; it is also used later in the book to describe quantum algorithms.

There are many variants on the basic Turing machine model. We might imagine Turing machines with different kinds of tapes. For example, one could consider two-way infinite tapes, or perhaps computation with tapes of more than one dimension. So far as is presently known, it is not possible to change any aspect of the Turing model in a way that is physically reasonable, and which manages to extend the class of functions computable by the model.

As an example consider a Turing machine equipped with multiple tapes. For simplicity we consider the two-tape case, as the generalization to more than two tapes is clear from this example. Like the basic Turing machine, a two-tape Turing machine has a finite number of internal states $q_1, \ldots, q_m$, a start state q_s, and a halt state q_h. It has two tapes, each of which contain symbols from some finite alphabet of symbols, Γ. As before we find it convenient to assume that the alphabet contains four symbols, $0, 1, b$ and $\triangleright$, where $\triangleright$ marks the left hand edge of each tape. The machine has two tape-heads, one for each tape. The main difference between the two-tape Turing machine and the basic Turing machine is in the program. Program lines are of the form $\langle q, x_1, x_2, q', x_1', x_2', s_1, s_2 \rangle$, meaning that if the internal state of the machine is q, tape one is reading x_1 at its current position, and tape two is reading x_2 at its current position, then the internal state of the machine should be changed to q', x_1 overwritten with x_1', x_2 overwritten with x_2', and the tape-heads for tape one and tape two moved according to whether s_1 or s_2 are equal to $+1, -1$ or 0, respectively.

In what sense are the basic Turing machine and the two-tape Turing machine equivalent models of computation? They are equivalent in the sense that each computational model is able to *simulate* the other. Suppose we have a two-tape Turing machine which takes as input a bit string x on the first tape and blanks on the remainder of both tapes,

except the endpoint marker $\triangleright$. This machine computes a function $f(x)$, where $f(x)$ is defined to be the contents of the first tape after the Turing machine has halted. Rather remarkably, it turns out that given a two-tape Turing machine to compute f, there exists an equivalent single-tape Turing machine that is also able to compute f. We won't explain how to do this explicitly, but the basic idea is that the single-tape Turing machine *simulates* the two-tape Turing machine, using its single tape to store the contents of both tapes of the two-tape Turing machine. There is some computational overhead required to do this simulation, but the important point is that in principle it can always be done. In fact, there exists a Universal Turing machine (see Box 3.1) which can simulate any other Turing machine!

Another interesting variant of the Turing machine model is to introduce randomness into the model. For example, imagine that the Turing machine can execute a program line whose effect is the following: if the internal state is q and the tape-head reads x, then flip an unbiased coin. If the coin lands heads, change the internal state to $q_{i_{\rm H}}$, and if it lands tails, change the internal state to $q_{i_{\rm T}}$, where $q_{i_{\rm H}}$ and $q_{i_{\rm T}}$ are two internal states of the Turing machine. Such a program line can be represented as $\langle q, x, q_{i_{\rm H}}, q_{i_{\rm T}} \rangle$. However, even this variant doesn't change the essential power of the Turing machine model of computation. It is not difficult to see that we can simulate the effect of the above algorithm on a deterministic Turing machine by explicitly 'searching out' all the possible computational paths corresponding to different values of the coin tosses. Of course, this deterministic simulation may be far less efficient than the random model, but the key point for the present discussion is that the class of functions computable is not changed by introducing randomness into the underlying model.

Exercise 3.3: (Turing machine to reverse a bit string) Describe a Turing machine which takes a binary number x as input, and outputs the bits of x in reverse order. (*Hint*: In this exercise and the next it may help to use a multi-tape Turing machine and/or symbols other than $\triangleright, 0, 1$ and the blank.)

Exercise 3.4: (Turing machine to add modulo 2) Describe a Turing machine to add two binary numbers x and y modulo 2. The numbers are input on the Turing machine tape in binary, in the form x, followed by a single blank, followed by a y. If one number is not as long as the other then you may assume that it has been padded with leading 0s to make the two numbers the same length.

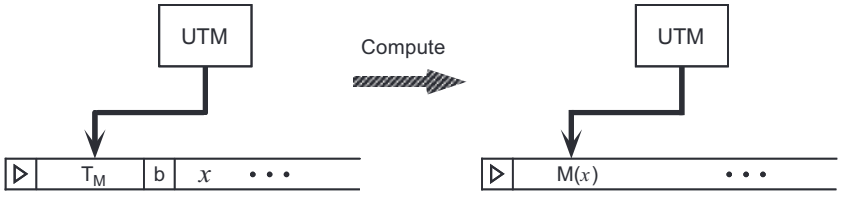
Let us return to Hilbert's entscheidungsproblem, the original inspiration for the founders of computer science. Is there an algorithm to decide all the problems of mathematics? The answer to this question was shown by Church and Turing to be no. In Box 3.2, we explain Turing's proof of this remarkable fact. This phenomenon of *undecidability* is now known to extend far beyond the examples which Church and Turing constructed. For example, it is known that the problem of deciding whether two topological spaces are topologically equivalent ('homeomorphic') is undecidable. There are simple problems related to the behavior of dynamical systems which are undecidable, as you will show in Problem 3.4. References for these and other examples are given in the end of chapter 'History and further reading'.

Besides its intrinsic interest, undecidability foreshadows a topic of great concern in computer science, and also to quantum computation and quantum information: the dis-

Box 3.1: The Universal Turing Machine

We've described Turing machines as containing three elements which may vary from machine to machine – the initial configuration of the tape, the internal states of the finite state control, and the program for the machine. A clever idea known as the *Universal Turing Machine* (UTM) allows us to fix the program and finite state control once and for all, leaving the initial contents of the tape as the only part of the machine which needs to be varied.

The Universal Turing Machine (see the figure below) has the following property. Let M be any Turing machine, and let T_M be the Turing number associated to machine M. Then on input of the binary representation for T_M followed by a blank, followed by any string of symbols x on the remainder of the tape, the Universal Turing Machine gives as output whatever machine M would have on input of x. Thus, the Universal Turing Machine is capable of simulating any other Turing machine!



The Universal Turing Machine is similar in spirit to a modern programmable computer, in which the action to be taken by the computer – the 'program' – is stored in memory, analogous to the bit string T_M stored at the beginning of the tape by the Universal Turing Machine. The data to be processed by the program is stored in a separate part of memory, analogous to the role of x in the Universal Turing Machine. Then some fixed hardware is used to run the program, producing the output. This fixed hardware is analogous to the internal states and the (fixed) program being executed by the Universal Turing Machine.

Describing the detailed construction of a Universal Turing Machine is beyond the scope of this book. (Though industrious readers may like to attempt the construction.) The key point is the existence of such a machine, showing that a single fixed machine can be used to run any algorithm whatsoever. The existence of a Universal Turing Machine also explains our earlier statement that the number of internal states in a Turing machine does not matter much, for provided that number m exceeds the number needed for a Universal Turing Machine, such a machine can be used to simulate a Turing machine with any number of internal states.

tinction between problems which are easy to solve, and problems which are hard to solve. Undecidability provides the ultimate example of problems which are hard to solve – so hard that they are in fact impossible to solve.

Exercise 3.5: (Halting problem with no inputs) Show that given a Turing

machine M there is no algorithm to determine whether M halts when the input to the machine is a blank tape.

Exercise 3.6: (Probabilistic halting problem) Suppose we number the probabilistic Turing machines using a scheme similar to that found in Exercise 3.2 and define the probabilistic halting function $h_p(x)$ to be 1 if machine x halts on input of x with probability at least $1/2$ and 0 if machine x halts on input of x with probability less than $1/2$. Show that there is no probabilistic Turing machine which can output $h_p(x)$ with probability of correctness strictly greater than $1/2$ for all x.

Exercise 3.7: (Halting oracle) Suppose a *black box* is made available to us which takes a non-negative integer x as input, and then outputs the value of $h(x)$, where $h(\cdot)$ is the halting function defined in Box 3.2 on page 130. This type of black box is sometimes known as an *oracle* for the halting problem. Suppose we have a regular Turing machine which is augmented by the power to call the oracle. One way of accomplishing this is to use a two-tape Turing machine, and add an extra program instruction to the Turing machine which results in the oracle being called, and the value of $h(x)$ being printed on the second tape, where x is the current contents of the second tape. It is clear that this model for computation is more powerful than the conventional Turing machine model, since it can be used to compute the halting function. Is the halting problem for this model of computation undecidable? That is, can a Turing machine aided by an oracle for the halting problem decide whether a program for the Turing machine with oracle will halt on a particular input?

3.1.2 Circuits

Turing machines are rather idealized models of computing devices. Real computers are *finite* in size, whereas for Turing machines we assumed a computer of unbounded size. In this section we investigate an alternative model of computation, the *circuit model*, that is equivalent to the Turing machine in terms of computational power, but is more convenient and realistic for many applications. In particular the circuit model of computation is especially important as preparation for our investigation of quantum computers.

A circuit is made up of *wires* and *gates*, which carry information around, and perform simple computational tasks, respectively. For example, Figure 3.2 shows a simple circuit which takes as input a single bit, a. This bit is passed through a NOT gate, which flips the bit, taking 1 to 0 and 0 to 1. The wires before and after the NOT gate serve merely to carry the bit to and from the NOT gate; they can represent movement of the bit through space, or perhaps just through time.

More generally, a circuit may involve many input and output bits, many wires, and many logical gates. A *logic gate* is a function $f : \{0,1\}^k \rightarrow \{0,1\}^l$ from some fixed number k of *input bits* to some fixed number l of *output bits*. For example, the NOT gate is a gate with one input bit and one output bit which computes the function $f(a) = 1 \oplus a$, where a is a single bit, and $\oplus$ is modulo 2 addition. It is also usual to make the convention that no loops are allowed in the circuit, to avoid possible instabilities, as illustrated in Figure 3.3. We say such a circuit is *acyclic*, and we adhere to the convention that circuits in the circuit model of computation be acyclic.

Box 3.2: The halting problem

In Exercise 3.2 you showed that each Turing machine can be uniquely associated with a number from the list $1, 2, 3, \ldots$. To solve Hilbert's problem, Turing used this numbering to pose the *halting problem*: does the machine with Turing number x halt upon input of the number y? This is a well posed and interesting mathematical problem. After all, it is a matter of some considerable interest to us whether our algorithms halt or not. Yet it turns out that there is no algorithm which is capable of solving the halting problem. To see this, Turing asked whether there is an algorithm to solve an even more specialized problem: does the machine with Turing number x halt upon input of the same number x? Turing defined the *halting function*,

$$h(x) \equiv \begin{cases} 0 & \text{if machine number } x \text{ does not halt upon input of } x \\ 1 & \text{if machine number } x \text{ halts upon input of } x. \end{cases}$$

If there is an algorithm to solve the halting problem, then there surely is an algorithm to evaluate $h(x)$. We will try to reach a contradiction by supposing such an algorithm exists, denoted by HALT(x). Consider an algorithm computing the function TURING(x), with pseudocode

```
TURING(x)

y = HALT(x)
if y = 0 then
        halt
else
        loop forever
end if
```

Since HALT is a valid program, TURING must also be a valid program, with some Turing number, t. By definition of the halting function, $h(t) = 1$ if and only if TURING halts on input of t. But by inspection of the program for TURING, we see that TURING halts on input of t if and only if $h(t) = 0$. Thus $h(t) = 1$ if and only if $h(t) = 0$, a contradiction. Therefore, our initial assumption that there is an algorithm to evaluate $h(x)$ must have been wrong. We conclude that there is no algorithm allowing us to solve the halting problem.

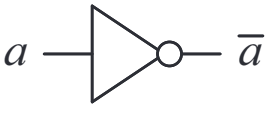


Figure 3.2. Elementary circuit performing a single NOT gate on a single input bit.

There are many other elementary logic gates which are useful for computation. A partial list includes the AND gate, the OR gate, the XOR gate, the NAND gate, and the NOR gate. Each of these gates takes two bits as input, and produces a single bit as output. The AND gate outputs 1 if and only if both of its inputs are 1. The OR gate outputs 1 if

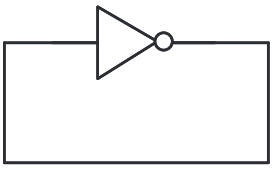


Figure 3.3. Circuits containing cycles can be unstable, and are not usually permitted in the circuit model of computation.

and only if at least one of its inputs is 1. The XOR gate outputs the sum, modulo 2, of its inputs. The NAND and NOR gates take the AND and OR, respectively, of their inputs, and then apply a NOT to whatever is output. The action of these gates is illustrated in Figure 3.4.

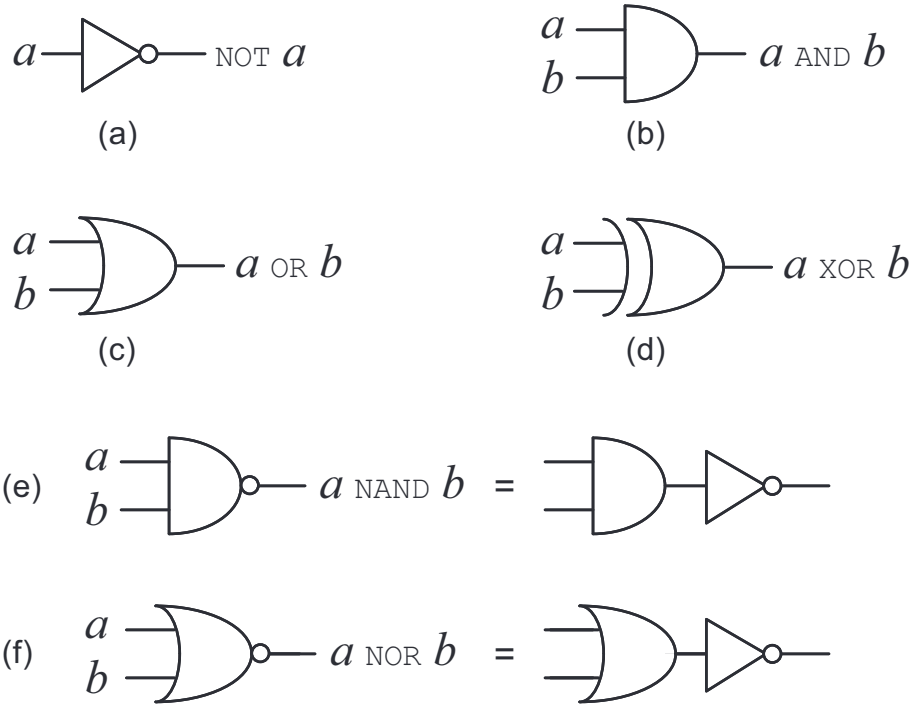


Figure 3.4. Elementary circuits performing the AND, OR, XOR, NAND, and NOR gates.

There are two important 'gates' missing from Figure 3.4, namely the FANOUT gate and the CROSSOVER gate. In circuits we often allow bits to 'divide', replacing a bit with two copies of itself, an operation referred to as FANOUT. We also allow bits to CROSSOVER, that is, the value of two bits are interchanged. A third operation missing from Figure 3.4, not really a logic gate at all, is to allow the preparation of extra *ancilla* or *work* bits, to allow extra working space during the computation.

These simple circuit elements can be put together to perform an enormous variety of computations. Below we'll show that these elements can be used to compute any function whatsoever. In the meantime, let's look at a simple example of a circuit which adds two n bit integers, using essentially the same algorithm taught to school-children around the

world. The basic element in this circuit is a smaller circuit known as a *half-adder*, shown in Figure 3.5. A half-adder takes two bits, x and y, as input, and outputs the sum of the bits $x \oplus y$ modulo 2, together with a carry bit set to 1 if x and y are both 1, or 0 otherwise.

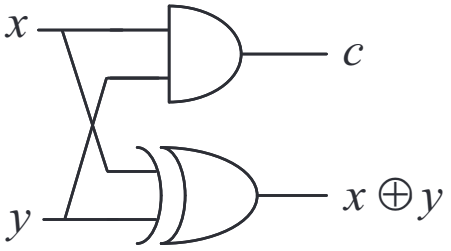


Figure 3.5. Half-adder circuit. The carry bit c is set to 1 when x and y are both 1, otherwise it is 0.

Two cascaded half-adders may be used to build a *full-adder*, as shown in Figure 3.6. A full-adder takes as input three bits, x, y, and c. The bits x and y should be thought of as data to be added, while c is a carry bit from an earlier computation. The circuit outputs two bits. One output bit is the modulo 2 sum, $x \oplus y \oplus c$ of all three input bits. The second output bit, c', is a carry bit, which is set to 1 if two or more of the inputs is 1, and is 0 otherwise.

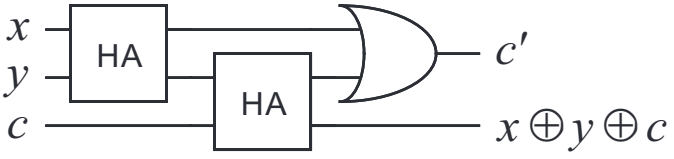


Figure 3.6. Full-adder circuit.

By cascading many of these full-adders together we obtain a circuit to add two n-bit integers, as illustrated in Figure 3.7 for the case $n = 3$.

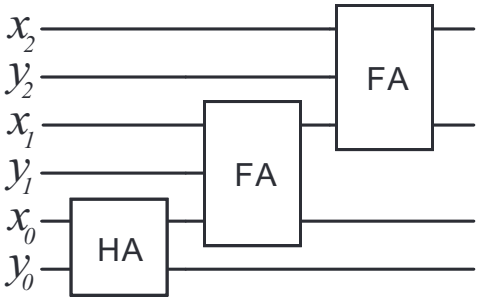


Figure 3.7. Addition circuit for two three-bit integers, $x = x_2x_1x_0$ and $y = y_2y_1y_0$, using the elementary algorithm taught to school-children.

We claimed earlier that just a few *fixed* gates can be used to compute *any* function $f : \{0,1\}^n \to \{0,1\}^m$ whatsoever. We will now prove this for the simplified case of a function $f : \{0,1\}^n \to \{0,1\}$ with n input bits and a single output bit. Such a function

is known as a *Boolean function*, and the corresponding circuit is a *Boolean circuit*. The general universality proof follows immediately from the special case of Boolean functions. The proof is by induction on n. For $n = 1$ there are four possible functions: the identity, which has a circuit consisting of a single wire; the bit flip, which is implemented using a single NOT gate; the function which replaces the input bit with a 0, which can be obtained by ANDing the input with a work bit initially in the 0 state; and the function which replaces the input with a 1, which can be obtained by ORing the input with a work bit initially in the 1 state.

To complete the induction, suppose that any function on n bits may be computed by a circuit, and let f be a function on $n + 1$ bits. Define n-bit functions f_0 and f_1 by $f_0(x_1, \ldots, x_n) \equiv f(0, x_1, \ldots, x_n)$ and $f_1(x_1, \ldots, x_n) \equiv f(1, x_1, \ldots, x_n)$. These are both n-bit functions, so by the inductive hypothesis there are circuits to compute these functions.

It is now an easy matter to design a circuit which computes f. The circuit computes both f_0 and f_1 on the last n bits of the input. Then, depending on whether the first bit of the input was a 0 or a 1 it outputs the appropriate answer. A circuit to do this is shown in Figure 3.8. This completes the induction.

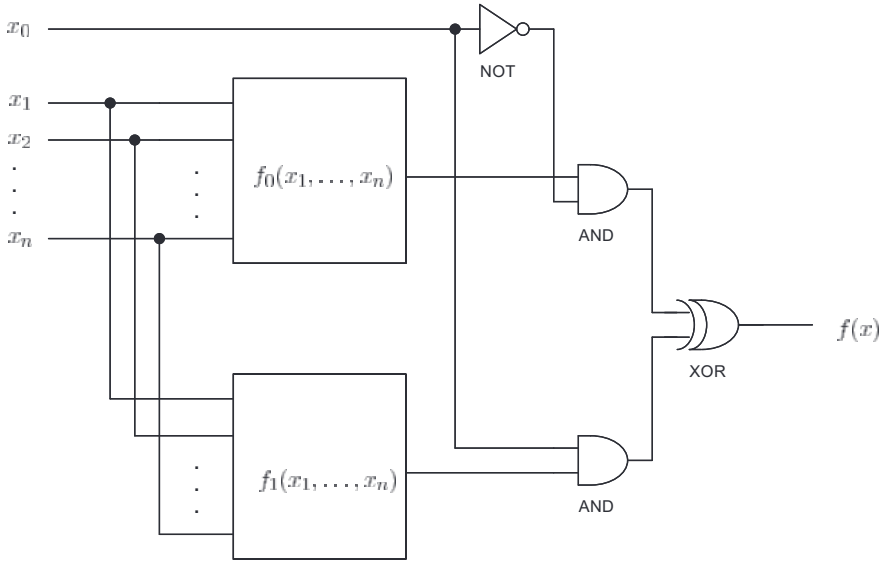


Figure 3.8. Circuit to compute an arbitrary function f on $n + 1$ bits, assuming by induction that there are circuits to compute the n-bit functions f_0 and f_1.

Five elements may be identified in the universal circuit construction: (1) wires, which preserve the states of the bits; (2) ancilla bits prepared in standard states, used in the $n = 1$ case of the proof; (3) the FANOUT operation, which takes a single bit as input and outputs two copies of that bit; (4) the CROSSOVER operation, which interchanges the value of two bits; and (5) the AND, XOR, and NOT gates. In Chapter 4 we'll define the quantum circuit model of computation in a manner analogous to classical circuits. It is interesting to note that many of these five elements pose some interesting challenges when extending to the quantum case: it is not necessarily obvious that good quantum wires for the preservation of qubits can be constructed, even in principle, the FANOUT

operation cannot be performed in a straightforward manner in quantum mechanics, due to the no-cloning theorem (as explained in Section 1.3.5), and the AND and XOR gates are not invertible, and thus can't be implemented in a straightforward manner as unitary quantum gates. There is certainly plenty to think about in defining a quantum circuit model of computation!

Exercise 3.8: (Universality of NAND) Show that the NAND gate can be used to simulate the AND, XOR and NOT gates, provided wires, ancilla bits and FANOUT are available.

Let's return from our brief quantum digression, to the properties of classical circuits. We claimed earlier that the Turing machine model is equivalent to the circuit model of computation. In what sense do we mean the two models are equivalent? On the face of it, the two models appear quite different. The unbounded nature of a Turing machine makes them more useful for abstractly specifying what it is we mean by an algorithm, while circuits more closely capture what an actual physical computer does.

The two models are connected by introducing the notion of a *uniform circuit family*. A *circuit family* consists of a collection of circuits, $\{C_n\}$, indexed by a positive integer n. The circuit C_n has n input bits, and may have any finite number of extra work bits, and output bits. The output of the circuit C_n, upon input of a number x of at most n bits in length, is denoted by $C_n(x)$. We require that the circuits be *consistent*, that is, if $m < n$ and x is at most m bits in length, then $C_m(x) = C_n(x)$. The function computed by the circuit family $\{C_n\}$ is the function $C(\cdot)$ such that if x is n bits in length then $C(x) = C_n(x)$. For example, consider a circuit C_n that squares an n-bit number. This defines a family of circuits $\{C_n\}$ that computes the function, $C(x) = x^2$, where x is any positive integer.

It's not enough to consider unrestricted families of circuits, however. In practice, we need an algorithm to build the circuit. Indeed, if we don't place any restrictions on the circuit family then it becomes possible to compute all sorts of functions which we do not expect to be able to compute in a reasonable model of computation. For example, let $h_n(x)$ denote the halting function, restricted to values of x which are n bits in length. Thus h_n is a function from n bits to 1 bit, and we have proved there exists a circuit C_n to compute $h_n(\cdot)$. Therefore the circuit family $\{C_n\}$ computes the halting function! However, what prevents us from using this circuit family to solve the halting problem is that we haven't specified an algorithm which will allow us to build the circuit C_n for all values of n. Adding this requirement results in the notion of a uniform circuit family.

That is, a family of circuits $\{C_n\}$ is said to be a *uniform circuit family* if there is some algorithm running on a Turing machine which, upon input of n, generates a *description* of C_n. That is, the algorithm outputs a description of what gates are in the circuit C_n, how those gates are connected together to form a circuit, any ancilla bits needed by the circuit, FANOUT and CROSSOVER operations, and where the output from the circuit should be read out. For example, the family of circuits we described earlier for squaring n-bit numbers is certainly a uniform circuit family, since there is an algorithm which, given n, outputs a description of the circuit needed to square an n-bit number. You can think of this algorithm as the means by which an engineer is able to generate a description of (and thus build) the circuit for any n whatsoever. By contrast, a circuit family that is not uniform is said to be a *non-uniform* circuit family. There is no algorithm to construct

the circuit for arbitrary n, which prevents our engineer from building circuits to compute functions like the halting function.

Intuitively, a uniform circuit family is a family of circuits that can be generated by some reasonable algorithm. It can be shown that the class of functions computable by uniform circuit families is exactly the same as the class of functions which can be computed on a Turing machine. With this uniformity restriction, results in the Turing machine model of computation can usually be given a straightforward translation into the circuit model of computation, and vice versa. Later we give similar attention to issues of uniformity in the quantum circuit model of computation.

3.2 The analysis of computational problems

The analysis of computational problems depends upon the answer to three fundamental questions:

(1) **What is a computational problem?** Multiplying two numbers together is a computational problem; so is programming a computer to exceed human abilities in the writing of poetry. In order to make progress developing a general theory for the analysis of computational problems we are going to isolate a special class of problems known as *decision problems*, and concentrate our analysis on those. Restricting ourselves in this way enables the development of a theory which is both elegant and rich in structure. Most important, it is a theory whose principles have application far beyond decision problems.

(2) **How may we design algorithms to solve a given computational problem?** Once a problem has been specified, what algorithms can be used to solve the problem? Are there general techniques which can be used to solve wide classes of problems? How can we be sure an algorithm behaves as claimed?

(3) **What are the minimal resources required to solve a given computational problem?** Running an algorithm requires the consumption of *resources*, such as time, space, and energy. In different situations it may be desirable to minimize consumption of one or more resource. Can we classify problems according to the resource requirements needed to solve them?

In the next few sections we investigate these three questions, especially questions 1 and 3. Although question 1, 'what is a computational problem?', is perhaps the most fundamental of the questions, we shall defer answering it until Section 3.2.3, pausing first to establish some background notions related to resource quantification in Section 3.2.1, and then reviewing the key ideas of *computational complexity* in Section 3.2.2.

Question 2, how to design good algorithms, is the subject of an enormous amount of ingenious work by many researchers. So much so that in this brief introduction we cannot even begin to describe the main ideas employed in the design of good algorithms. If you are interested in this beautiful subject, we refer you to the end of chapter 'History and further reading'. Our closest direct contact with this subject will occur later in the book, when we study quantum algorithms. The techniques involved in the creation of quantum algorithms have typically involved a blend of deep existing ideas in algorithm design for classical computers, and the creation of new, wholly quantum mechanical techniques for algorithm design. For this reason, and because the spirit of quantum algorithm design