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

is so similar in many ways to classical algorithm design, we encourage you to become familiar with at least the basic ideas of algorithm design.

Question 3, what are the minimal resources required to solve a given computational problem, is the main focus of the next few sections. For example, suppose we are given two numbers, each n bits in length, which we wish to multiply. If the multiplication is performed on a single-tape Turing machine, how many computational steps must be executed by the Turing machine in order to complete the task? How much space is used on the Turing machine while completing the task?

These are examples of the type of resource questions we may ask. Generally speaking, computers make use of many different kinds of resources, however we will focus most of our attention on time, space, and energy. Traditionally in computer science, time and space have been the two major resource concerns in the study of algorithms, and we study these issues in Sections 3.2.2 through 3.2.4. Energy has been a less important consideration; however, the study of energy requirements motivates the subject of reversible classical computation, which in turn is a prerequisite for quantum computation, so we examine energy requirements for computation in some considerable detail in Section 3.2.5.

3.2.1 How to quantify computational resources

Different models of computation lead to different resource requirements for computation. Even something as simple as changing from a single-tape to a two-tape Turing machine may change the resources required to solve a given computational problem. For a computational task which is extremely well understood, like addition of integers, for example, such differences between computational models may be of interest. However, for a first pass at understanding a problem, we would like a way of quantifying resource requirements that is independent of relatively trivial changes in the computational model. One of the tools which has been developed to do this is the *asymptotic notation*, which can be used to summarize the *essential* behavior of a function. This asymptotic notation can be used, for example, to summarize the essence of how many time steps it takes a given algorithm to run, without worrying too much about the exact time count. In this section we describe this notation in detail, and apply it to a simple problem illustrating the quantification of computational resources – the analysis of algorithms for sorting a list of names into alphabetical order.

Suppose, for example, that we are interested in the number of gates necessary to add together two n-bit numbers. Exact counts of the number of gates required obscure the big picture: perhaps a specific algorithm requires $24n + 2\lceil \log n \rceil + 16$ gates to perform this task. However, in the limit of large problem size the only term which matters is the $24n$ term. Furthermore, we disregard constant factors as being of secondary importance to the analysis of the algorithm. The *essential* behavior of the algorithm is summed up by saying that the number of operations required scales like n, where n is the number of bits in the numbers being added. The asymptotic notation consists of three tools which make this notion precise.

The O ('big O') notation is used to set *upper bounds* on the behavior of a function. Suppose $f(n)$ and $g(n)$ are two functions on the non-negative integers. We say '$f(n)$ is in the class of functions $O(g(n))$', or just '$f(n)$ is $O(g(n))$', if there are constants c and n_0 such that for all values of n greater than n_0, $f(n) \leq cg(n)$. That is, for sufficiently large n, the function $g(n)$ is an upper bound on $f(n)$, up to an unimportant constant

factor. The big O notation is particularly useful for studying the worst-case behavior of *specific* algorithms, where we are often satisfied with an upper bound on the resources consumed by an algorithm.

When studying the behaviors of a *class* of algorithms – say the entire class of algorithms which can be used to multiply two numbers – it is interesting to set lower bounds on the resources required. For this the Ω ('big Omega') notation is used. A function $f(n)$ is said to be $\Omega(g(n))$ if there exist constants c and n_0 such that for all n greater than n_0, $cg(n) \leq f(n)$. That is, for sufficiently large n, $g(n)$ is a lower bound on $f(n)$, up to an unimportant constant factor.

Finally, the Θ ('big Theta') notation is used to indicate that $f(n)$ behaves the same as $g(n)$ asymptotically, up to unimportant constant factors. That is, we say $f(n)$ is $\Theta(g(n))$ if it is both $O(g(n))$ and $\Omega(g(n))$.

Asymptotic notation: examples

Let's consider a few simple examples of the asymptotic notation. The function $2n$ is in the class $O(n^2)$, since $2n \leq 2n^2$ for all positive n. The function 2^n is $\Omega(n^3)$, since $n^3 \leq 2^n$ for sufficiently large n. Finally, the function $7n^2 + \sqrt{n}\log(n)$ is $\Theta(n^2)$, since $7n^2 \leq 7n^2 + \sqrt{n}\log(n) \leq 8n^2$ for all sufficiently large values of n. In the following few exercises you will work through some of the elementary properties of the asymptotic notation that make it a useful tool in the analysis of algorithms.

Exercise 3.9: Prove that $f(n)$ is $O(g(n))$ if and only if $g(n)$ is $\Omega(f(n))$. Deduce that $f(n)$ is $\Theta(g(n))$ if and only if $g(n)$ is $\Theta(f(n))$.

Exercise 3.10: Suppose $g(n)$ is a polynomial of degree k. Show that $g(n)$ is $O(n^l)$ for any $l \geq k$.

Exercise 3.11: Show that $\log n$ is $O(n^k)$ for any $k > 0$.

Exercise 3.12: ($n^{\log n}$ **is super-polynomial**) Show that n^k is $O(n^{\log n})$ for any k, but that $n^{\log n}$ is never $O(n^k)$.

Exercise 3.13: ($n^{\log n}$ **is sub-exponential**) Show that c^n is $\Omega(n^{\log n})$ for any $c > 1$, but that $n^{\log n}$ is never $\Omega(c^n)$.

Exercise 3.14: Suppose $e(n)$ is $O(f(n))$ and $g(n)$ is $O(h(n))$. Show that $e(n)g(n)$ is $O(f(n)h(n))$.

An example of the use of the asymptotic notation in quantifying resources is the following simple application to the problem of sorting an n element list of names into alphabetical order. Many sorting algorithms are based upon the 'compare-and-swap' operation: two elements of an n element list are compared, and swapped if they are in the wrong order. If this compare-and-swap operation is the only means by which we can access the list, how many such operations are required in order to ensure that the list has been correctly sorted?

A simple compare-and-swap algorithm for solving the sorting problem is as follows: (note that `compare-and-swap(j,k)` compares the list entries numbered j and k, and swaps them if they are out of order)

```
for j = 1 to n-1
    for k = j+1 to n
        compare-and-swap(j,k)
    end k
end j
```

It is clear that this algorithm correctly sorts a list of n names into alphabetical order. Note that the number of compare-and-swap operations executed by the algorithm is $(n-1) + (n-2) + \cdots + 1 = n(n-1)/2$. Thus the number of compare-and-swap operations used by the algorithm is $\Theta(n^2)$. Can we do better than this? It turns out that we can. Algorithms such as 'heapsort' are known which run using $O(n \log n)$ compare-and-swap operations. Furthermore, in Exercise 3.15 you'll work through a simple counting argument that shows any algorithm based upon the compare-and-swap operation requires $\Omega(n \log n)$ such operations. Thus, the sorting problem requires $\Theta(n \log n)$ compare-and-swap operations, in general.

Exercise 3.15: (Lower bound for compare-and-swap based sorts) Suppose an n element list is sorted by applying some sequence of compare-and-swap operations to the list. There are $n!$ possible initial orderings of the list. Show that after k of the compare-and-swap operations have been applied, at most 2^k of the possible initial orderings will have been sorted into the correct order. Conclude that $\Omega(n \log n)$ compare-and-swap operations are required to sort all possible initial orderings into the correct order.

3.2.2 Computational complexity

> *The idea that there won't be an algorithm to solve it – this is something fundamental that won't ever change – that idea appeals to me.*
> – Stephen Cook

> *Sometimes it is good that some things are impossible. I am happy there are many things that nobody can do to me.*
> – Leonid Levin

> *It should not come as a surprise that our choice of polynomial algorithms as the mathematical concept that is supposed to capture the informal notion of 'practically efficient computation' is open to criticism from all sides. [...] Ultimately, our argument for our choice must be this:* ***Adopting polynomial worst-case performance as our criterion of efficiency results in an elegant and useful theory that says something meaningful about practical computation, and would be impossible without this simplification.***
> – Christos Papadimitriou

What time and space resources are required to perform a computation? In many cases these are the most important questions we can ask about a computational problem. Problems like addition and multiplication of numbers are regarded as efficiently solvable because we have *fast* algorithms to perform addition and multiplication, which consume

little *space* when running. Many other problems have no known fast algorithm, and are effectively impossible to solve, not because we can't find an algorithm to solve the problem, but because all known algorithms consume such vast quantities of space or time as to render them practically useless.

Computational complexity is the study of the time and space resources required to solve computational problems. The task of computational complexity is to prove *lower bounds* on the resources required by the best possible algorithm for solving a problem, even if that algorithm is not explicitly known. In this and the next two sections, we give an overview of computational complexity, its major concepts, and some of the more important results of the field. Note that computational complexity is in a sense complementary to the field of algorithm design; ideally, the most efficient algorithms we could design would match perfectly with the lower bounds proved by computational complexity. Unfortunately, this is often not the case. As already noted, in this book we won't examine classical algorithm design in any depth.

One difficulty in formulating a theory of computational complexity is that different computational models may require different resources to solve the same problem. For instance, multiple-tape Turing machines can solve many problems substantially faster than single-tape Turing machines. This difficulty is resolved in a rather coarse way. Suppose a problem is specified by giving n bits as input. For instance, we might be interested in whether a particular n-bit number is prime or not. The chief distinction made in computational complexity is between problems which can be solved using resources which are bounded by a *polynomial* in n, or which require resources which grow faster than any polynomial in n. In the latter case we usually say that the resources required are *exponential* in the problem size, abusing the term exponential, since there are functions like $n^{\log n}$ which grow faster than any polynomial (and thus are 'exponential' according to this convention), yet which grow slower than any true exponential. A problem is regarded as *easy*, *tractable* or *feasible* if an algorithm for solving the problem using polynomial resources exists, and as *hard*, *intractable* or *infeasible* if the best possible algorithm requires exponential resources.

As a simple example, suppose we have two numbers with binary expansions $x_1 \ldots x_{m_1}$ and $y_1 \ldots y_{m_2}$, and we wish to determine the sum of the two numbers. The total size of the input is $n \equiv m_1 + m_2$. It's easy to see that the two numbers can be added using a number of elementary operations that scales as $\Theta(n)$; this algorithm uses a polynomial (indeed, linear) number of operations to perform its tasks. By contrast, it is believed (though it has never been proved!) that the problem of factoring an integer into its prime factors is intractable. That is, the belief is that there is no algorithm which can factor an arbitrary n-bit integer using $O(p(n))$ operations, where p is some fixed polynomial function of n. We will later give many other examples of problems which are believed to be intractable in this sense.

The polynomial versus exponential classification is rather coarse. In practice, an algorithm that solves a problem using $2^{n/1000}$ operations is probably more useful than one which runs in n^{1000} operations. Only for very large input sizes ($n \approx 10^8$) will the 'efficient' polynomial algorithm be preferable to the 'inefficient' exponential algorithm, and for many purposes it may be more practical to prefer the 'inefficient' algorithm.

Nevertheless, there are many reasons to base computational complexity primarily on the polynomial versus exponential classification. First, historically, with few exceptions, polynomial resource algorithms have been much faster than exponential algorithms. We

might speculate that the reason for this is lack of imagination: coming up with algorithms requiring n, n^2 or some other low degree polynomial number of operations is often much easier than finding a natural algorithm which requires n^{1000} operations, although examples like the latter do exist. Thus, the predisposition for the human mind to come up with relatively simple algorithms has meant that in practice polynomial algorithms usually do perform much more efficiently than their exponential cousins.

A second and more fundamental reason for emphasizing the polynomial versus exponential classification is derived from the *strong Church–Turing thesis*. As discussed in Section 1.1, it was observed in the 1960s and 1970s that probabilistic Turing machines appear to be the strongest 'reasonable' model of computation. More precisely, researchers consistently found that if it was possible to compute a function using k elementary operations in some model that was *not* the probabilistic Turing machine model of computation, then it was always possible to compute the same function in the probabilistic Turing machine model, using at most $p(k)$ elementary operations, where $p(\cdot)$ is some *polynomial* function. This statement is known as the *strong Church–Turing thesis*:

> **Strong Church–Turing thesis**: *Any model of computation can be simulated on a probabilistic Turing machine with at most a polynomial increase in the number of elementary operations required.*

The strong Church–Turing thesis is great news for the theory of computational complexity, for it implies that attention may be restricted to the probabilistic Turing machine model of computation. After all, if a problem has no polynomial resource solution on a probabilistic Turing machine, then the strong Church–Turing thesis implies that it has no efficient solution on any computing device. Thus, the strong Church–Turing thesis implies that the entire theory of computational complexity will take on an elegant, model-independent form if the notion of efficiency is identified with polynomial resource algorithms, and this elegance has provided a strong impetus towards acceptance of the identification of 'solvable with polynomial resources' and 'efficiently solvable'. Of course, one of the prime reasons for interest in quantum computers is that they cast into doubt the strong Church–Turing thesis, by enabling the efficient solution of a problem which is believed to be intractable on all classical computers, including probabilistic Turing machines! Nevertheless, it is useful to understand and appreciate the role the strong Church–Turing thesis has played in the search for a model-independent theory of computational complexity.

Finally, we note that, in practice, computer scientists are not only interested in the polynomial versus exponential classification of problems. This is merely the first and coarsest way of understanding how difficult a computational problem is. However, it is an exceptionally important distinction, and illustrates many broader points about the nature of resource questions in computer science. For most of this book, it will be our central concern in evaluating the efficiency of a given algorithm.

Having examined the merits of the polynomial versus exponential classification, we now have to confess that the theory of computational complexity has one remarkable outstanding failure: it seems very hard to prove that there are interesting classes of problems which require exponential resources to solve. It is quite easy to give non-constructive proofs that *most* problems require exponential resources (see Exercise 3.16, below), and furthermore many interesting problems are *conjectured* to require exponential resources for their solution, but rigorous proofs seem very hard to come by, at least with the present

state of knowledge. This failure of computational complexity has important implications for quantum computation, because it turns out that the computational power of quantum computers can be related to some major open problems in *classical* computational complexity theory. Until these problems are resolved, it cannot be stated with certainty how computationally powerful a quantum computer is, or even whether it is more powerful than a classical computer!

Exercise 3.16: (Hard-to-compute functions exist) Show that there exist Boolean functions on n inputs which require at least $2^n/\log n$ logic gates to compute.

3.2.3 Decision problems and the complexity classes P and NP

Many computational problems are most cleanly formulated as *decision problems* – problems with a yes or no answer. For example, is a given number m a prime number or not? This is the *primality* decision problem. The main ideas of computational complexity are most easily and most often formulated in terms of decision problems, for two reasons: the theory takes its simplest and most elegant form in this form, while still generalizing in a natural way to more complex scenarios; and historically computational complexity arose primarily from the study of decision problems.

Although most decision problems can easily be stated in simple, familiar language, discussion of the general properties of decision problems is greatly helped by the terminology of *formal languages*. In this terminology, a *language* L over the *alphabet* Σ is a subset of the set Σ^* of all (finite) strings of symbols from Σ. For example, if $\Sigma = \{0, 1\}$, then the set of binary representations of even numbers, $L = \{0, 10, 100, 110, \ldots\}$ is a language over Σ.

Decision problems may be encoded in an obvious way as problems about languages. For instance, the primality decision problem can be encoded using the binary alphabet $\Sigma = \{0, 1\}$. Strings from Σ^* can be interpreted in a natural way as non-negative integers. For example, 0010 can be interpreted as the number 2. The language L is defined to consist of all binary strings such that the corresponding number is prime.

To solve the primality decision problem, what we would like is a Turing machine which, when started with a given number n on its input tape, eventually outputs some equivalent of 'yes' if n is prime, and outputs 'no' if n is not prime. To make this idea precise, it is convenient to modify our old Turing machine definition (of Section 3.1.1) slightly, replacing the halting state q_h with two states q_Y and q_N to represent the answers 'yes' and 'no' respectively. In all other ways the machine behaves in the same way, and it still halts when it enters the state q_Y or q_N. More generally, a language L is *decided* by a Turing machine if the machine is able to decide whether an input x on its tape is a member of the language of L or not, eventually halting in the state q_Y if $x \in L$, and eventually halting in the state q_N if $x \notin L$. We say that the machine has *accepted* or *rejected* x depending on which of these two cases comes about.

How quickly can we determine whether or not a number is prime? That is, what is the fastest Turing machine which decides the language representing the primality decision problem? We say that a problem is in **TIME**$(f(n))$ if there exists a Turing machine which decides whether a candidate x is in the language in time $O(f(n))$, where n is the length of x. A problem is said to be solvable in *polynomial time* if it is in **TIME**(n^k) for some finite k. The collection of all languages which are in **TIME**(n^k), for some k, is denoted **P**. P is our first example of a *complexity class*. More generally, a complexity

class is defined to be a collection of languages. Much of computational complexity theory is concerned with the definition of various complexity classes, and understanding the relationship between different complexity classes.

Not surprisingly, there are problems which cannot be solved in polynomial time. Unfortunately, proving that any given problem can't be solved in polynomial time seems to be very difficult, although conjectures abound! A simple example of an interesting decision problem which is believed not to be in **P** is the *factoring decision problem*:

> FACTORING: Given a composite integer m and $l < m$, does m have a non-trivial factor less than l?

An interesting property of factoring is that if somebody claims that the answer is 'yes, m does have a non-trivial factor less than l' then they can establish this by exhibiting such a factor, which can then be efficiently checked by other parties, simply by doing long-division. We call such a factor a *witness* to the fact that m has a factor less than l. This idea of an easily checkable witness is the key idea in the definition of the complexity class **NP**, below. We have phrased factoring as a decision problem, but you can easily verify that the decision problem is equivalent to finding the factors of a number:

Exercise 3.17: Prove that a polynomial-time algorithm for finding the factors of a number m exists if and only if the factoring decision problem is in **P**.

Factoring is an example of a problem in an important complexity class known as **NP**. What distinguishes problems in **NP** is that 'yes' instances of a problem can easily be verified with the aid of an appropriate witness. More rigorously, a language L is in **NP** if there is a Turing machine M with the following properties:

(1) If $x \in L$ then there exists a witness string w such that M halts in the state q_Y after a time polynomial in $|x|$ when the machine is started in the state x-blank-w.
(2) If $x \notin L$ then for all strings w which attempt to play the role of a witness, the machine halts in state q_N after a time polynomial in $|x|$ when M is started in the state x-blank-w.

There is an interesting asymmetry in the definition of **NP**. While we have to be able to quickly decide whether a possible witness to $x \in L$ is truly a witness, there is no such need to produce a witness to $x \notin L$. For instance, in the factoring problem, we have an easy way of proving that a given number has a factor less than m, but exhibiting a witness to prove that a number has no factors less than m is more daunting. This suggests defining **coNP**, the class of languages which have witnesses to 'no' instances; obviously the languages in **coNP** are just the complements of languages in **NP**.

How are **P** and **NP** related? It is clear that **P** is a subset of **NP**. The most famous open problem in computer science is *whether or not there are problems in* **NP** *which are not in* **P**, often abbreviated as the **P** $\neq$ **NP** problem. Most computer scientists believe that **P** $\neq$ **NP**, but despite decades of work nobody has been able to prove this, and the possibility remains that **P** = **NP**.

Exercise 3.18: Prove that if **coNP** $\neq$ **NP** then **P** $\neq$ **NP**.

Upon first acquaintance it's tempting to conclude that the conjecture **P** $\neq$ **NP** ought to be pretty easy to resolve. To see why it's actually rather subtle it helps to see couple of

examples of problems that are in **P** and **NP**. We'll draw the examples from *graph theory*, a rich source of decision problems with surprisingly many practical applications. A *graph* is a finite collection of *vertices* $\{v_1, \ldots, v_n\}$ connected by *edges*, which are pairs (v_i, v_j) of vertices. For now, we are only concerned with *undirected graphs*, in which the order of the vertices (in each edge pair) does not matter; similar ideas can be investigated for *directed graphs* in which the order of vertices does matter. A typical graph is illustrated in Figure 3.9.

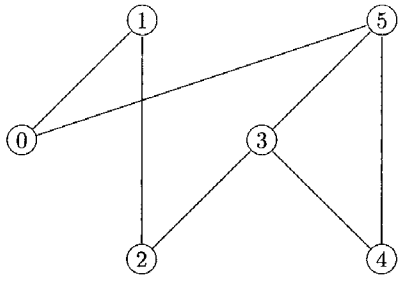

Figure 3.9. A graph.

A *cycle* in a graph is a sequence $v_1, \ldots, v_m$ of vertices such that each pair (v_j, v_{j+1}) is an edge, as is (v_1, v_m). A *simple cycle* is a cycle in which none of the vertices is repeated, except for the first and last vertices. A *Hamiltonian cycle* is a simple cycle which visits every vertex in the graph. Examples of graphs with and without Hamiltonian cycles are shown in Figure 3.10.

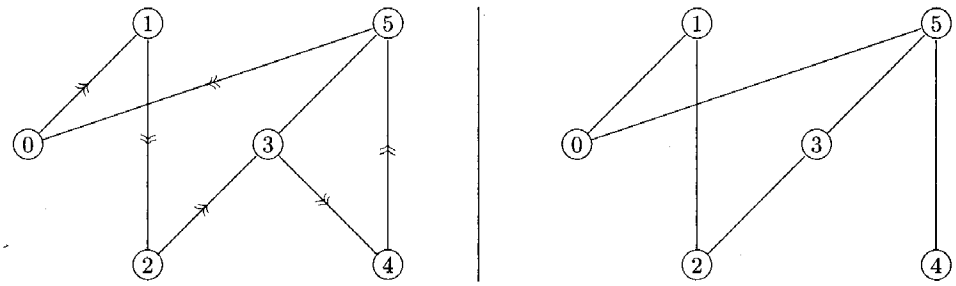

Figure 3.10. The graph on the left contains a Hamiltonian cycle, 0, 1, 2, 3, 4, 5, 0. The graph on the right contains no Hamiltonian cycle, as can be verified by inspection.

The *Hamiltonian cycle problem* (HC) is to determine whether a given graph contains a Hamiltonian cycle or not. HC is a decision problem in **NP**, since if a given graph has a Hamiltonian cycle, then that cycle can be used as an easily checkable witness. Moreover, HC has no known polynomial time algorithm. Indeed, HC is a problem in the class of so-called **NP**-complete problems, which can be thought of as the 'hardest' problems in **NP**, in the sense that solving HC in time t allows any other problem in **NP** to be solved in time $O(\text{poly}(t))$. This also means that if any **NP**-complete problem has a polynomial time solution then it will follow that **P** = **NP**.

There is a problem, the Euler cycle decision problem, which is superficially similar to HC, but which has astonishingly different properties. An *Euler cycle* is an ordering of the edges of a graph G so that every edge in the graph is visited exactly once. The Euler

cycle decision problem (EC) is to determine, given a graph G on n vertices, whether that graph contains an Euler cycle or not. EC is, in fact, exactly the same problem as HC, only the path visits edges, rather than vertices. Consider the following remarkable theorem, to be proven in Exercise 3.20:

Theorem 3.1: **(Euler's theorem)** A connected graph contains an Euler cycle if and only if every vertex has an even number of edges incident upon it.

Euler's theorem gives us a method for efficiently solving EC. First, check to see whether the graph is connected; this is easily done with $O(n^2)$ operations, as shown in Exercise 3.19. If the graph is not connected, then obviously no Euler cycle exists. If the graph is connected then for each vertex check whether there is an even number of edges incident upon the vertex. If a vertex is found for which this is not the case, then there is no Euler cycle, otherwise an Euler cycle exists. Since there are n vertices, and at most $n(n-1)/2$ edges, this algorithm requires $O(n^3)$ elementary operations. Thus EC is in **P**! Somehow, there is a structure present in the problem of visiting each edge that can be exploited to provide an efficient algorithm for EC, yet which does not seem to be reflected in the problem of visiting each vertex; it is not at all obvious why such a structure should be present in one case, but not in the other, if indeed it is absent for the HC problem.

Exercise 3.19: The REACHABILITY problem is to determine whether there is a path between two specified vertices in a graph. Show that REACHABILITY can be solved using $O(n)$ operations if the graph has n vertices. Use the solution to REACHABILITY to show that it is possible to decide whether a graph is connected in $O(n^2)$ operations.

Exercise 3.20: (Euler's theorem) Prove Euler's theorem. In particular, if each vertex has an even number of incident edges, give a constructive procedure for finding an Euler cycle.

The equivalence between the factoring decision problem and the factoring problem proper is a special instance of one of the most important ideas in computer science, an idea known as *reduction*. Intuitively, we know that some problems can be viewed as special instances of other problems. A less trivial example of reduction is the reduction of HC to the *traveling salesman* decision problem (TSP). The traveling salesman decision problem is as follows: we are given n cities $1, 2, \ldots, n$ and a non-negative integer distance d_{ij} between each pair of cities. Given a distance d the problem is to determine if there is a tour of all the cities of distance less than d.

The reduction of HC to TSP goes as follows. Suppose we have a graph containing n vertices. We turn this into an instance of TSP by thinking of each vertex of the graph as a 'city' and defining the distance d_{ij} between cities i and j to be one if vertices i and j are connected, and the distance to be two if the vertices are unconnected. Then a tour of the cities of distance less than $n+1$ must be of distance n, and be a Hamiltonian cycle for the graph. Conversely, if a Hamiltonian cycle exists then a tour of the cities of distance less than $n+1$ must exist. In this way, given an algorithm for solving TSP, we can convert it into an algorithm for solving HC without much overhead. Two consequences can be inferred from this. First, if TSP is a tractable problem, then HC is also tractable. Second, if HC is hard then TSP must also be hard. This is an example of a general technique known

as *reduction*: we've reduced the problem HC to the problem TSP. This is a technique we will use repeatedly throughout this book.

A more general notion of reduction is illustrated in Figure 3.11. A language B is said to be *reducible* to another language A if there exists a Turing machine operating in polynomial time such that given as input x it outputs $R(x)$, and $x \in B$ if and only if $R(x) \in A$. Thus, if we have an algorithm for deciding A, then with a little extra overhead we can decide the language B. In this sense, the language B is essentially no more difficult to decide than the language A.

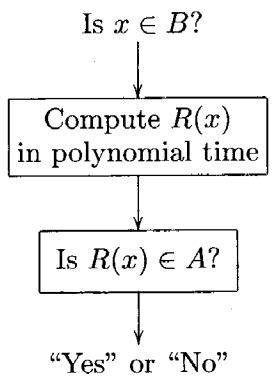

Figure 3.11. Reduction of B to A.

Exercise 3.21: (Transitive property of reduction) Show that if a language L_1 is reducible to the language L_2 and the language L_2 is reducible to L_3 then the language L_1 is reducible to the language L_3.

Some complexity classes have problems which are *complete* with respect to that complexity class, meaning there is a language L in the complexity class which is the 'most difficult' to decide, in the sense that every other language in the complexity class can be reduced to L. Not all complexity classes have complete problems, but many of the complexity classes we are concerned with do have complete problems. A trivial example is provided by **P**. Let L be any language in **P** which is not empty or equal to the set of all words. That is, there exists a string x_1 such that $x_1 \notin L$ and a string x_2 such that $x_2 \in L$. Then any other language L' in **P** can be reduced to L using the following reduction: given an input x, use the polynomial time decision procedure to determine whether $x \in L'$ or not. If it is not, then set $R(x) = x_1$, otherwise set $R(x) = x_2$.

Exercise 3.22: Suppose L is complete for a complexity class, and L' is another language in the complexity class such that L reduces to L'. Show that L' is complete for the complexity class.

Less trivially, **NP** also contains complete problems. An important example of such a problem and the prototype for all other **NP**-complete problems is the *circuit satisfiability* problem or CSAT: given a Boolean circuit composed of AND, OR and NOT gates, is there an assignment of values to the inputs to the circuit that results in the circuit outputting 1, that is, is the circuit *satisfiable* for some input? The **NP**-completeness of CSAT is known as the *Cook–Levin theorem*, for which we now outline a proof.

Theorem 3.2: **(Cook–Levin)** CSAT is **NP**-complete.

Proof

The proof has two parts. The first part of the proof is to show that CSAT is in **NP**, and the second part is to show that any language in **NP** can be reduced to CSAT. Both parts of the proof are based on *simulation* techniques: the first part of the proof is essentially showing that a Turing machine can efficiently simulate a circuit, while the second part of the proof is essentially showing that a circuit can efficiently simulate a Turing machine. Both parts of the proof are quite straightforward; for the purposes of illustration we give the second part in some detail.

The first part of the proof is to show that CSAT is in **NP**. Given a circuit containing n circuit elements, and a potential witness w, it is obviously easy to check in polynomial time on a Turing machine whether or not w satisfies the circuit, which establishes that CSAT is in **NP**.

The second part of the proof is to show that any language $L \in \mathbf{NP}$ can be reduced to CSAT. That is, we aim to show that there is a polynomial time computable reduction R such that $x \in L$ if and only if $R(x)$ is a satisfiable circuit. The idea of the reduction is to find a circuit which simulates the action of the machine M which is used to check instance-witness pairs, (x, w), for the language L. The input variables for the circuit will represent the witness; the idea is that finding a witness which satisfies the circuit is equivalent to M accepting (x, w) for some specific witness w. Without loss of generality we may make the following assumptions about M to simplify the construction:

(1) M's tape alphabet is $\triangleright$,0,1 and the blank symbol.
(2) M runs using time at most $t(n)$ and total space at most $s(n)$ where $t(n)$ and $s(n)$ are polynomials in n.
(3) Machine M can actually be assumed to run using time *exactly* $t(n)$ for all inputs of size n. This is done by adding the lines $\langle q_Y, x, q_Y, x, 0\rangle$, and $\langle q_N, x, q_N, x, 0\rangle$ for each of $x = \triangleright, 0, 1$ and the blank, artificially halting the machine after exactly $t(n)$ steps.

The basic idea of the construction to simulate M is outlined in Figure 3.12. Each internal state of the Turing machine is represented by a single bit in the circuit. We name the corresponding bits $\tilde{q}_s, \tilde{q}_1, \ldots, \tilde{q}_m, \tilde{q}_Y, \tilde{q}_N$. Initially, $\tilde{q}_s$ is set to one, and all the other bits representing internal states are set to zero. Each square on the Turing machine tape is represented by three bits: two bits to represent the letter of the alphabet ($\triangleright, 0, 1$ or blank) currently residing on the tape, and a single 'flag' bit which is set to one if the read-write head is pointing to the square, and set to zero otherwise. We denote the bits representing the tape contents by $(u_1, v_1), \ldots, (u_{s(n)}, v_{s(n)})$ and the corresponding flag bits by $f_1, \ldots, f_{s(n)}$. Initially the u_j and v_j bits are set to represent the inputs x and w, as appropriate, while $f_1 = 1$ and all other $f_j = 0$. There is also a lone extra 'global flag' bit, F, whose function will be explained later. F is initially set to zero. We regard all the bits input to the circuit as fixed, except for those representing the witness w, which are the variable bits for the circuit.

The action of M is obtained by repeating $t(n)$ times a 'simulation step' which simulates the execution of a single program line for the Turing machine. Each simulation step may be broken up into a sequence of steps corresponding in turn to the respective program lines, with a final step which resets the global flag F to zero, as

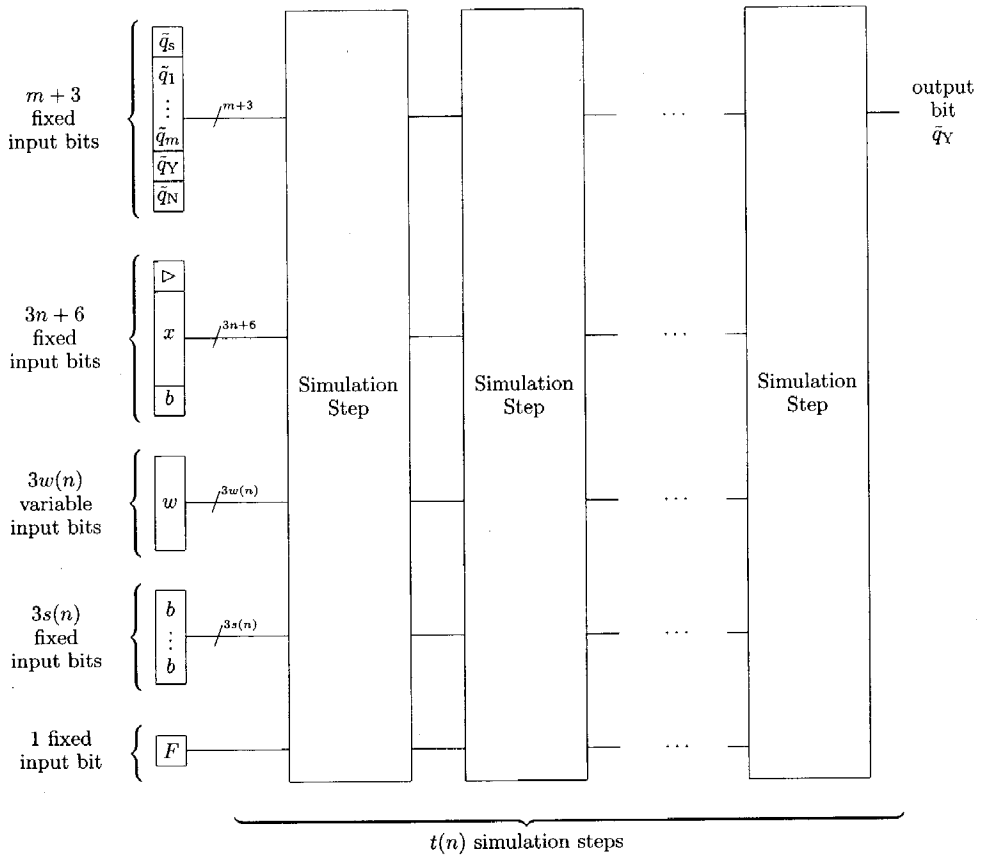

Figure 3.12. Outline of the procedure used to simulate a Turing machine using a circuit.

illustrated in Figure 3.13. To complete the simulation, we only need to simulate a program line of the form $\langle q_i, x, q_j, x', s\rangle$. For convenience, we assume $q_i \neq q_j$, but a similar construction works in the case when $q_i = q_j$. The procedure is as follows:

(1) Check to see that $\tilde{q}_i = 1$, indicating that the current state of the machine is q_i.
(2) For each tape square:
   (a) Check to see that the global flag bit is set to zero, indicating that no action has yet been taken by the Turing machine.
   (b) Check that the flag bit is set to one, indicating that the tape head is at this tape square.
   (c) Check that the simulated tape contents at this point are x.
   (d) If all conditions check out, then perform the following steps:
      1. Set $\tilde{q}_i = 0$ and $\tilde{q}_j = 1$.
      2. Update the simulated tape contents at this tape square to x'.
      3. Update the flag bit of this and adjacent 'squares' as appropriate, depending on whether $s = +1, 0, -1$, and whether we are at the left hand end of the tape.
      4. Set the global flag bit to one, indicating that this round of computation has been completed.

This is a fixed procedure which involves a constant number of bits, and by the universality result of Section 3.1.2 can be performed using a circuit containing a constant number of gates.

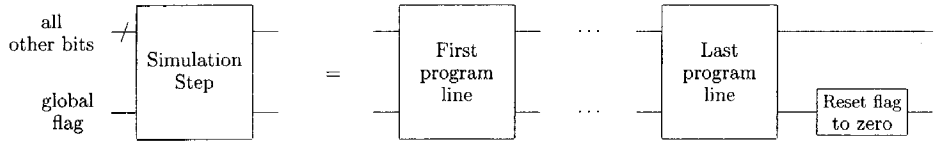

Figure 3.13. Outline of the simulation step used to simulate a Turing machine using a circuit.

The total number of gates in the entire circuit is easily seen to be $O(t(n)(s(n) + n))$, which is polynomial in size. At the end of the circuit, it is clear that $\tilde{q}_Y = 1$ if and only if the machine M accepts (x, w). Thus, the circuit is satisfiable if and only if there exists w such that machine M accepts (x, w), and we have found the desired reduction from L to CSAT. □

CSAT gives us a foot in the door which enables us to easily prove that many other problems are **NP**-complete. Instead of directly proving that a problem is **NP**-complete, we can instead prove that it is in **NP** and that CSAT reduces to it, so by Exercise 3.22 the problem must be **NP**-complete. A small sample of **NP**-complete problems is discussed in Box 3.3. An example of another **NP**-complete problem is the *satisfiability* problem (SAT), which is phrased in terms of a Boolean formula. Recall that a Boolean formula φ is composed of the following elements: a set of Boolean variables, $x_1, x_2, \ldots$; Boolean connectives, that is, a Boolean function with one or two inputs and one output, such as $\wedge$ (AND), $\vee$ (OR), and $\neg$ (NOT); and parentheses. The truth or falsity of a Boolean formula for a given set of Boolean variables is decided according to the usual laws of Boolean algebra. For example, the formula $\varphi = x_1 \vee \neg x_2$ has the satisfying assignment $x_1 = 0$ and $x_2 = 0$, while $x_1 = 0$ and $x_2 = 1$ is not a satisfying assignment. The satisfiability problem is to determine, given a Boolean formula φ, whether or not it is satisfiable by any set of possible inputs.

Exercise 3.23: Show that SAT is **NP**-complete by first showing that SAT is in **NP**, and then showing that CSAT reduces to SAT. (*Hint*: for the reduction it may help to represent each distinct wire in an instance of CSAT by different variables in a Boolean formula.)

An important restricted case of SAT is also **NP**-complete, the 3-satisfiability problem (3SAT), which is concerned with formulae in *3-conjunctive normal form*. A formula is said to be in *conjunctive normal form* if it is the AND of a collection of *clauses*, each of which is the OR of one or more *literals*, where a literal is an expression is of the form x or $\neg x$. For example, the formula $(x_1 \vee \neg x_2) \wedge (x_2 \vee x_3 \vee \neg x_4)$ is in conjunctive normal form. A formula is in *3-conjunctive normal form* or *3-CNF* if each clause has exactly three literals. For example, the formula $(\neg x_1 \vee x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \wedge (x_2 \vee x_3 \vee x_4)$ is in 3-conjunctive normal form. The 3-satisfiability problem is to determine whether a formula in 3-conjunctive normal form is satisfiable or not.

The proof that 3SAT is **NP**-complete is straightforward, but is a little too lengthy to justify inclusion in this overview. Even more than CSAT and SAT, 3SAT is in some sense

the **NP**-complete problem, and it is the basis for countless proofs that other problems are **NP**-complete. We conclude our discussion of **NP**-completeness with the surprising fact that 2SAT, the analogue of 3SAT in which every clause has two literals, can be solved in polynomial time:

Exercise 3.24: (2SAT **has an efficient solution**) Suppose φ is a Boolean formula in conjunctive normal form, in which each clause contains only two literals.

(1) Construct a (directed) graph $G(\varphi)$ with directed edges in the following way: the vertices of G correspond to variables x_j and their negations $\neg x_j$ in φ. There is a (directed) edge (α, β) in G if and only if the clause $(\neg\alpha \vee \beta)$ or the clause $(\beta \vee \neg\alpha)$ is present in φ. Show that φ is not satisfiable if and only if there exists a variable x such that there are paths from x to $\neg x$ and from $\neg x$ to x in $G(\varphi)$.

(2) Show that given a directed graph G containing n vertices it is possible to determine whether two vertices v_1 and v_2 are connected in polynomial time.

(3) Find an efficient algorithm to solve 2SAT.

Box 3.3: A zoo of NP-complete problems

The importance of the class **NP** derives, in part, from the enormous number of computational problems that are known to be **NP**-complete. We can't possibly hope to survey this topic here (see 'History and further reading'), but the following examples, taken from many distinct areas of mathematics, give an idea of the delicious melange of problems known to be **NP**-complete.

- CLIQUE (*graph theory*): A clique in an undirected graph G is a subset of vertices, each pair of which is connected by an edge. The *size* of a clique is the number of vertices it contains. Given an integer m and a graph G, does G have a clique of size m?
- SUBSET-SUM (*arithmetic*): Given a finite collection S of positive integers and a *target* t, is there any subset of S which sums to t?
- 0-1 INTEGER PROGRAMMING (*linear programming*): Given an integer $m \times n$ matrix A and an m-dimensional vector y with integer values, does there exist an n-dimensional vector x with entries in the set $\{0, 1\}$ such that $Ax \leq y$?
- VERTEX COVER (*graph theory*): A *vertex cover* for an undirected graph G is a set of vertices V' such that every edge in the graph has one or both vertices contained in V'. Given an integer m and a graph G, does G have a vertex cover V' containing m vertices?

Assuming that **P** $\neq$ **NP** it is possible to prove that there is a *non-empty* class of problems **NPI** (**NP** intermediate) which are neither solvable with polynomial resources, nor are **NP**-complete. Obviously, there are no problems known to be in **NPI** (otherwise we would know that **P** $\neq$ **NP**) but there are several problems which are regarded as being likely candidates. Two of the strongest candidates are the factoring and graph isomorphism problems:

GRAPH ISOMORPHISM: Suppose G and G' are two undirected graphs over the vertices $V \equiv \{v_1, \ldots, v_n\}$. Are G and G' isomorphic? That is, does there exist a one-to-one function $\varphi : V \rightarrow V$ such that the edge (v_i, v_j) is contained in G if and only if $(\varphi(v_i), \varphi(v_j))$ is contained in G?

Problems in **NPI** are interesting to researchers in quantum computation and quantum information for two reasons. First, it is desirable to find fast quantum algorithms to solve problems which are not in **P**. Second, many suspect that quantum computers will not be able to efficiently solve all problems in **NP**, ruling out **NP**-complete problems. Thus, it is natural to focus on the class **NPI**. Indeed, a fast quantum algorithm for factoring has been discovered (Chapter 5), and this has motivated the search for fast quantum algorithms for other problems suspected to be in **NPI**.

3.2.4 A plethora of complexity classes

We have investigated some of the elementary properties of some important complexity classes. A veritable pantheon of complexity classes exists, and there are many non-trivial relationships known or suspected between these classes. For quantum computation and quantum information, it is not necessary to understand all the different complexity classes that have been defined. However, it is useful to have some appreciation for the more important of the complexity classes, many of which have natural analogues in the study of quantum computation and quantum information. Furthermore, if we are to understand how powerful quantum computers are, then it behooves us to understand how the class of problems solvable on a quantum computer fits into the zoo of complexity classes which may be defined for classical computers.

There are essentially three properties that may be varied in the definition of a complexity class: the resource of interest (time, space, ...), the type of problem being considered (decision problem, optimization problem, ...), and the underlying computational model (deterministic Turing machine, probabilistic Turing machine, quantum computer, ...). Not surprisingly, this gives us an enormous range to define complexity classes. In this section, we briefly review a few of the more important complexity classes and some of their elementary properties. We begin with a complexity class defined by changing the resource of interest from time to *space*.

The most natural space-bounded complexity class is the class **PSPACE** of decision problems which may be solved on a Turing machine using a polynomial number of working bits, with no limitation on the amount of time that may be used by the machine (see Exercise 3.25). Obviously, **P** is included in **PSPACE**, since a Turing machine that halts after polynomial time can only traverse polynomially many squares, but it is also true that **NP** is a subset of **PSPACE**. To see this, suppose L is any language in **NP**. Suppose problems of size n have witnesses of size at most $p(n)$, where $p(n)$ is some polynomial in n. To determine whether or not the problem has a solution, we may sequentially test all $2^{p(n)}$ possible witnesses. Each test can be run in polynomial time, and therefore polynomial space. If we erase all the intermediate working between tests then we can test all the possibilities using polynomial space.

Unfortunately, at present it is not even known whether **PSPACE** contains problems which are not in **P**! This is a pretty remarkable situation – it seems fairly obvious that having unlimited time and polynomial spatial resources must be more powerful than having only a polynomial amount of time. However, despite considerable effort and in-

genuity, this has never been shown. We will see later that the class of problems solvable on a quantum computer in polynomial time is a subset of **PSPACE**, so proving that a problem efficiently solvable on a quantum computer is not efficiently solvable on a classical computer would establish that **P** $\neq$ **PSPACE**, and thus solve a major outstanding problem of computer science. An optimistic way of looking at this result is that ideas from quantum computation might be useful in proving that **P** $\neq$ **PSPACE**. Pessimistically, one might conclude that it will be a long time before anyone rigorously proves that quantum computers can be used to efficiently solve problems that are intractable on a classical computer. Even more pessimistically, it is possible that **P** = **PSPACE**, in which case quantum computers offer no advantage over classical computers! However, very few (if any) computational complexity theorists believe that **P** = **PSPACE**.

Exercise 3.25: (**PSPACE** $\subseteq$ **EXP**) The complexity class **EXP** (for *exponential time*) contains all decision problems which may be decided by a Turing machine running in exponential time, that is time $O(2^{n^k})$, where k is any constant. Prove that **PSPACE** $\subseteq$ **EXP**. (*Hint*: If a Turing machine has l internal states, an m letter alphabet, and uses space $p(n)$, argue that the machine can exist in one of at most $lm^{p(n)}$ different states, and that if the Turing machine is to avoid infinite loops then it must halt before revisiting a state.)

Exercise 3.26: (**L** $\subseteq$ **P**) The complexity class **L** (for *logarithmic space*) contains all decision problems which may be decided by a Turing machine running in logarithmic space, that is, in space $O(\log(n))$. More precisely, the class **L** is defined using a two-tape Turing machine. The first tape contains the problem instance, of size n, and is a read-only tape, in the sense that only program lines which don't change the contents of the first tape are allowed. The second tape is a working tape which initially contains only blanks. The logarithmic space requirement is imposed on the second, working tape only. Show that **L** $\subseteq$ **P**.

Does allowing more time or space give greater computational power? The answer to this question is yes in both cases. Roughly speaking, the *time hierarchy theorem* states that **TIME**$(f(n))$ is a proper subset of **TIME**$(f(n)\log^2(f(n)))$. Similarly, the *space hierarchy theorem* states that **SPACE**$(f(n))$ is a proper subset of **SPACE**$(f(n)\log(f(n)))$, where **SPACE**$(f(n))$ is, of course, the complexity class consisting of all languages that can be decided with spatial resources $O(f(n))$. The hierarchy theorems have interesting implications with respect to the equality of complexity classes. We know that

$$\mathbf{L} \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXP}. \tag{3.1}$$

Unfortunately, although each of these inclusions is widely believed to be strict, none of them has ever been proved to be strict. However, the time hierarchy theorem implies that **P** is a strict subset of **EXP**, and the space hierarchy theorem implies that **L** is a strict subset of **PSPACE**! So we can conclude that at least one of the inclusions in (3.1) must be strict, although we do not know which one.

What should we do with a problem once we know that it is **NP**-complete, or that some other hardness criterion holds? It turns out that this is far from being the end of the story in problem analysis. One possible line of attack is to identify special cases of the problem which may be amenable to attack. For example, in Exercise 3.24 we saw that the 2SAT problem has an efficient solution, despite the **NP**-completeness of SAT.

Another approach is to change the type of problem which is being considered, a tactic which typically results in the definition of new complexity classes. For example, instead of finding exact solutions to an **NP**-complete problem, we can instead try to find good algorithms for finding *approximate* solutions to a problem. For example, the VERTEX COVER problem is an **NP**-complete problem, yet in Exercise 3.27 we show that it is possible to efficiently find an approximation to the minimal vertex cover which is correct to within a factor two! On the other hand, in Problem 3.6 we show that it is not possible to find approximations to solutions of TSP correct to within any factor, *unless* **P** = **NP**!

Exercise 3.27: (Approximation algorithm for VERTEX COVER) Let $G = (V, E)$ be an undirected graph. Prove that the following algorithm finds a vertex cover for G that is within a factor two of being a minimal vertex cover:

$VC = \emptyset$
$E' = E$
do until $E' = \emptyset$
  let (α, β) be any edge of E'
  $VC = VC \cup \{\alpha, \beta\}$
  remove from E' every edge incident on α or β
return VC.

Why is it possible to approximate the solution of one **NP**-complete problem, but not another? After all, isn't it possible to efficiently transform from one problem to another? This is certainly true, however it is not necessarily true that this transformation preserves the notion of a 'good approximation' to a solution. As a result, the computational complexity theory of approximation algorithms for problems in **NP** has a structure that goes beyond the structure of **NP** proper. An entire complexity theory of approximation algorithms exists, which unfortunately is beyond the scope of this book. The basic idea, however, is to define a notion of reduction that corresponds to being able to efficiently reduce one approximation problem to another, in such a way that the notion of good approximation is preserved. With such a notion, it is possible to define complexity classes such as **MAXSNP** by analogy to the class **NP**, as the set of problems for which it is possible to efficiently verify approximate solutions to the problem. Complete problems exist for **MAXSNP**, just as for **NP**, and it is an interesting open problem to determine how the class **MAXSNP** compares to the class of approximation problems which are efficiently solvable.

We conclude our discussion with a complexity class that results when the underlying model of computation itself is changed. Suppose a Turing machine is endowed with the ability to flip coins, using the results of the coin tosses to decide what actions to take during the computation. Such a Turing machine may only accept or reject inputs with a certain probability. The complexity class **BPP** (for *bounded-error probabilistic time*) contains all languages L with the property that there exists a probabilistic Turing machine M such that if $x \in L$ then M accepts x with probability at least $3/4$, and if $x \notin L$, then M rejects x with probability at least $3/4$. The following exercise shows that the choice of the constant $3/4$ is essentially arbitrary:

Exercise 3.28: (Arbitrariness of the constant in the definition of BPP) Suppose k is a fixed constant, $1/2 < k \leq 1$. Suppose L is a language such that there exists a Turing machine M with the property that whenever $x \in L$, M accepts x with probability at least k, and whenever $x \notin L$, M rejects x with probability at least k. Show that $L \in \mathbf{BPP}$.

Indeed, the *Chernoff bound*, discussed in Box 3.4, implies that with just a few repetitions of an algorithm deciding a language in **BPP** the probability of success can be amplified to the point where it is essentially equal to one, for all intents and purposes. For this reason, **BPP** even more than **P** is the class of decision problems which is usually regarded as being efficiently solvable on a classical computer, and it is the quantum analogue of **BPP**, known as **BQP**, that is most interesting in our study of quantum algorithms.

3.2.5 Energy and computation

Computational complexity studies the amount of time and space required to solve a computational problem. Another important computational resource is *energy*. In this section, we study the energy requirements for computation. Surprisingly, it turns out that computation, both classical and quantum, can in principle be done without expending any energy! Energy consumption in computation turns out to be deeply linked to the *reversibility* of the computation. Consider a gate like the NAND gate, which takes as input two bits, and produces a single bit as output. This gate is intrinsically *irreversible* because, given the output of the gate, the input is not uniquely determined. For example, if the output of the NAND gate is 1, then the input could have been any one of 00, 01, or 10. On the other hand, the NOT gate is an example of a *reversible* logic gate because, given the output of the NOT gate, it is possible to infer what the input must have been.

Another way of understanding irreversibility is to think of it in terms of information erasure. If a logic gate is irreversible, then some of the information input to the gate is lost irretrievably when the gate operates – that is, some of the information has been erased by the gate. Conversely, in a reversible computation, no information is ever erased, because the input can always be recovered from the output. Thus, saying that a computation is reversible is equivalent to saying that no information is erased during the computation.

What is the connection between energy consumption and irreversibility in computation? *Landauer's principle* provides the connection, stating that, in order to erase information, it is necessary to dissipate energy. More precisely, Landauer's principle may be stated as follows:

> **Landauer's principle (first form)**: Suppose a computer erases a single bit of information. The amount of energy dissipated into the environment is *at least* $k_B T \ln 2$, where k_B is a universal constant known as *Boltzmann's constant*, and T is the temperature of the environment of the computer.

According to the laws of thermodynamics, Landauer's principle can be given an alternative form stated not in terms of energy dissipation, but rather in terms of entropy:

> **Landauer's principle (second form)**: Suppose a computer erases a single bit of information. The entropy of the environment increases by *at least* $k_B \ln 2$, where k_B is Boltzmann's constant.

Justifying Landauer's principle is a problem of physics that lies beyond the scope of this