1. A High-Level Tour of All Things Algorithmic

The field of algorithmic computer science, sometimes also called *algorithmics* or *informatics*, has led to profound and transformative advances in almost every area of science and industry over the past decades. As a consequence, proficiency in algorithms and computational problem-solving has become a crucial skill for students and professionals in almost every corner of today's data driven world, not only those with narrow focus in computer science itself.

This book provides a comprehensive introduction to the study of algorithms. We begin in this chapter with a high-level survey of some of the history, foundations, applications, fundamental concepts, and main topic areas generally associated with the study of algorithms. This chapter and the next two make up the "preliminaries" section of the book, where we introduce basic terminology and techniques. A review of relevant mathematics appears in Chapter 2, and in Chapter 3 we discuss the fundamental topic of sorting as well as basic techniques for algorithm design and analysis.

Before we get too far, however, it is worthwhile to ask the question "what is an algorithm?". An algorithm is a precisely-characterized procedure for solving a computational problem. Informally, it is a computational "recipe". A good algorithm is simple to understand and implement, it makes efficient use of computational resources (e.g., time, memory, processors, network bandwidth, energy) and it provides an exact or suitably high-quality solution to its intended problem.

To give a quick example, suppose we wish to find the definition of "informatics" in an alphabetically-ordered n-word dictionary. A correct but slow algorithm for this task is a *linear search*: examine every page sequentially from the beginning of the dictionary until we find the target word. However, a far better algorithm is a *binary search*: open the dictionary to its middle word — in the author's dictionary that middle word is "janitor". Since "informatics" precedes "janitor", we can rule out the entire second half of the dictionary and repeat the process on just the first half. Every step halves the total number of words under consideration, so we make much faster progress toward our solution, reaching the target word in at most $\log_2 n$ steps. In a dictionary with $n = 1$ million words, linear search might examine every single word, while binary search will never look at more than 20. The difference is quite dramatic. For many problems in practice, algorithmic improvements can easily reduce computation time from years to seconds.

An algorithm is different from a computer program, and the study of algorithms is different from the study of computer programming, even though the two are often taught together. Algorithms are abstract problem-solving procedures that can be realized as computer programs, but you can certainly study algorithms without writing computer programs. Likewise, you can study aspects of programming (e.g., syntax and semantics of programming languages) that do not particularly involve algorithms. The two topics are closely related, however, since algorithms play an important role in most computer programs, and proficiency in computer programming generally provides a useful mental framework that helps in learning algorithms.

1.1 A Bit of History

Although algorithmic computer science is mostly a young field, its historical roots go back quite far. In the next few pages, we outline some of the most important historical developments that helped to shape the field.

The study of algorithms has certainly been driven by technological innovation, with the widespread use of powerful computing devices of all shapes and sizes. However, algorithms were perhaps even more crucial before the advent of the modern computer, when solving problems required painstaking manual calculation. For example, in order to solve a large system of linear equations to determine the orbit of the asteroid Pallas, the great mathematician Carl Friedrich Gauss developed in 1810 the algorithm we now know as Gaussian elimination. Many classical algorithms for performing numerical computation have similar origins.

The word *algorithm* itself comes from the name of a renown 9th century Persian mathematician Abu Ja'far Muhammad ibn Musa Al-Khwarizmi (one of his books, entitled "al-Mukhtasar fi Hisab al-Jabr wa l-Muqabala" is the origin of the term *algebra* as well). In his written work, Al-Khwarizmi describes arithmetic procedures for operating on numbers written in base 10. That his work was later translated and spread in influence across Europe is part of the reason the base-10 system is known as the system of "Arabic" numerals, even though it was initially developed in India.

1.1.1 Decidability: Hilbert, Gödel, Church, and Turing

The work of David Hilbert, one of the most prominent mathematicians of the early 20th century, helped shape some of our earliest thoughts on the theory of computation. In 1928, Hilbert posed two famous questions (with some technical details omitted for simplicity of discussion):

- Can one derive all of mathematics from a small set of fundamental axioms that is both *consistent* and *complete*? Consistent means that no mathematical statement can be proved to be both true and false, and complete means that it is possible to construct a proof of the truth or falseness of every statement.

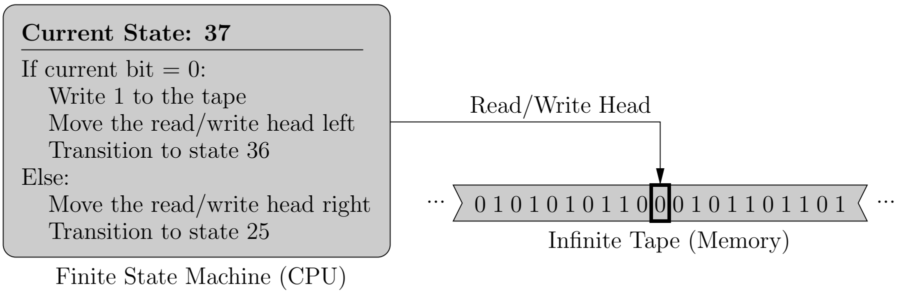


FIGURE 1.1: Diagram of a Turing machine.

- For every mathematical statement, does there exist an algorithm that can determine its truth or falseness?

In 1931 Kurt Gödel surprised the mathematical world by resolving the first question negatively, demonstrating that for any consistent set of axioms, there must exist statements that cannot be proved or disproved. We include a [short sketch] of Gödel's famous proof for the interested reader. It employs a clever technique called *diagonalization*, which shows that there are in some sense more mathematical statements than proofs (even though there are infinite numbers of both!), so there must exist some statement for which a corresponding proof does not exist.

Shortly after Gödel's result, Alan Turing and Alonzo Church both independently managed to resolve the second question also in the negative, providing examples of *undecidable problems* that cannot be solved by any algorithm, irrespective of the amount of time the algorithm is allowed to run. We now know of several natural undecidable problems, the most famous of them probably being the *halting problem*, which asks us to predict whether a given algorithm will terminate or run forever (in an endless loop) on a given input. The results of Church and Turing proved that there are fundamental theoretical limitations to the power of algorithms. These limitations rarely get in the way, however, since the vast majority of the problems we encounter in practice are quite clearly *decidable* (solvable by algorithms). The more troublesome limitation is that many practical problems seem not "efficiently" solvable by algorithms. [Short proof that the halting problem is undecidable]

1.1.2 Turing Machines and the Church-Turing Thesis

What is a computer? One of the main contributions of the work of Church and Turing was to formally characterize the notions of "computation" and "algorithms" in precise mathematical terms — a crucial prerequisite before we can prove rigorous theorems about these concepts, such as the existence of undecidable problems.

Turing characterized an algorithm in terms of a simple abstract computing machine now known as a *Turing machine*, shown in Figure 1.1. It contains a memory in the form of an infinite one-dimensional binary tape, as well as a processing unit that interacts with the tape via a read/write head. The processing unit is a *finite state machine*: at every time step it looks at its current state and the binary digit currently under the read/write head, and based on these it transitions to a new internal state, writes a new binary digit onto the tape (if desired), and shifts the tape left or right by one position (if desired). The tape is used by the Turing machine to read its input, to store the results of intermediate calculations, and to write its output. The machine starts in a designated "start state" and terminates once it reaches a designated "stop state".

Turing's bold claim was that this simplistic machine is "universal" in its ability to model any conceivable algorithm. Church made a similar claim based on an alternative model of computation known as the lambda calculus, which was soon shown to be equivalent to the Turing machine. The combined result of the work of Church and Turing is known as the *Church-Turing thesis*, and it asserts that every algorithm in the world can be represented by a Turing machine. As a consequence, one cannot get around the undecidability of problems like the halting problem by just building a fancy new type of computer.

There is no Nobel prize in computing, but in recognition of Turing's contributions, the *Turing award* is now given each year to the top researchers in computer science.

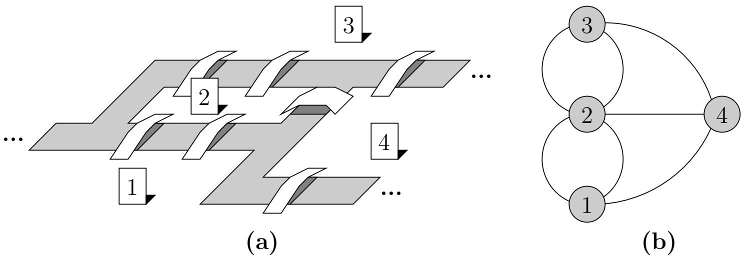
1.1.3 From Computability to Complexity Theory

The results of Church and Turing in the 1930s were fundamentally important in the area of *computability* — deciding what problems can and cannot be solved by algorithms of different types. The next big step in the development of the theory of computation was the study of *complexity theory*, which addresses how efficiently certain problems can be solved by algorithms.

There are often only seemingly minor differences between problems that are easy to solve and those that are vastly more difficult. As an example, consider the famous "Bridges of Königsberg" story, a favorite among mathematicians: In the early 18th century, Königsberg was a city in East Prussia with seven bridges spanning the rivers crossing through the city center, shown in Figure 1.2(a). As the story goes, the residents of Königsberg challenged themselves to find a path that crossed every bridge *exactly once*. The prolific mathematician Leonard Euler cleverly resolved this question (negatively) in 1735 by providing one of the original results in the area of *graph theory*. A *graph* is a collection of *nodes* and *edges*, where every edge connects a pair of nodes. As shown in Figure 1.2(b), Euler modeled the Königsberg bridges abstractly in terms of a graph where the nodes represent regions of the city and edges represent bridges connecting these regions. We now refer to a path through a graph that visits every edge exactly once an *Eulerian path*, since Euler was the first to characterize the precise mathematical conditions required for such a path to exist: the graph must be connected, and all but at most two nodes must have an even number of incident edges. [Simple proof]

It turns out that Eulerian paths are "easy" to compute. It is fairly simple to transform the constructive proof above into an algorithm that finds such a path (if it exists) in time proportional to the size of a graph; we will show how when we study graphs in Chapter **??**. Even for a billion-edge graph, a modern computer can therefore easily find an Eulerian path in a matter of seconds.

On the other hand, consider the similar problem of finding a *Hamiltonian* path —



(a) **(b)**

FIGURE 1.2: Illustrations of (a) the seven bridges of the city of Königsberg, and (b) their representation as the edges of a graph (more precisely a *multigraph*, since we have multiple edges between the same pair of nodes).

a path that visits every *node* exactly once, named after the 19th century mathematician William Hamilton. Although Hamiltonian paths and Eulerian paths may not appear that different, it turns out that the Hamiltonian path problem is *far* more difficult to solve efficiently. For an n-node graph, all known Hamiltonian path algorithms require time at least exponential in n. Even for small values like $n = 50$, this can take years even on the fastest of modern computers!

In the early 1970s, Richard Karp, Stephen Cook, and Leonid Levin dramatically advanced the field of complexity theory by developing the theory of *NP-completeness*. It allows us to group thousands of hard problems into a single *complexity class* (known as the *NP-complete* problems) by proving they all have in some sense "equivalent" hardness. An efficient solution for just one of the NP-complete problems would imply, via appropriate transformations, the existence of an efficient solution for *all* of the NP-complete problems; we discuss how this works in greater detail later in this chapter. Decades of research have yielded not a single efficient solution to any NP-complete problem, so we strongly suspect that these problems are not efficiently solvable, although nobody has managed to *prove* this. If someone asks you to produce an efficient algorithm for determining whether a graph has a Hamiltonian path (a problem known to be NP-complete), you can therefore decline to do so in a graceful manner, by pointing out that doing so would resolve what is perhaps the biggest open problem in the entire field of computer science.

The field of cryptography, now extremely important in practice due to the need for secure digital commerce, has its roots in complexity theory, since most algorithms for encrypting data crucially depend on the existence of hard problems so that decryption in the absence of the proper "key" is computationally infeasible.

1.2 Models of Computation

A mathematical proof only makes sense in the context of a certain system of fundamental axioms telling us the underlying assumptions we can make. Similarly, the *running time* of an algorithm (the number of operations it performs) only makes sense within the context of a specific *model of computation* that defines the primitive operations available in the abstract computing environment in which the algorithm executes.

Often, one algorithm will appear "better" than another simply because it has access to a wider range of operations in a more powerful model of computation. To illustrate this point with a ridiculous example, we can solve the NP-complete Hamiltonian path problem in a single time step if we define a (completely unrealistic) abstract computing environment supporting a "compute Hamiltonian path" instruction. It is to our advantage to adopt a computational model that reflects the capabilities and limitations of an actual modern computer. This not only enables straightforward implementation of our algorithms on an actual computer, but it also makes their running time analyses much more realistic.

We have already seen one model of computation so far in this chapter: the Turing machine. An algorithm in this model uses operations like "shift the tape left/right", "write a zero/one to the current location on the tape", or "test whether the value in the current location on the tape is a zero or one". Unfortunately, even a simple task like adding two binary numbers requires a painfully complex sequence of these very low-level operations. The Turing machine is a poor approximation of a modern computer, which can do things like adding two numbers in a single operation.

1.2.1 The RAM Model of Computation

In this book, we adopt the simple and widely-used computational model known as the *random access machine* (RAM), which roughly approximates a modern digital computer with a single processor. Data in the RAM model is processed at the granularity of fixed-length binary numbers known as *words*; modern computers typically have word sizes of 32 or 64 bits. We can perform simple arithmetic operations like addition, subtraction, multiplication, integer division, remainder, and comparison in a single step[1] on two words. There is a memory consisting of a long array of words, and we can store and retrieve words in a single step in a "random access" fashion. That is, we can access any word in memory directly in a single step via its numeric *address*, or index, within the memory.

The only slightly murky issue about the RAM model is the size of word. It is problematic to assume that words can be arbitrarily large, since this gives the model too much power — it could do unrealistic things like adding together two infinite-digit numbers in a single step. On the other hand, it is also problematic to assume that words contain at most some fixed constant number of bits (e.g., 64 bits), independent of the size of the problem instance we are solving. An algorithm receiving n words of data as input needs the ability to count as high as n, or else it wouldn't be able to index into the input in memory. Hence, word size must be at least $\log_2 n$ bits, since this is necessary to represent a number of size n. A commonly assumed word size is $k \log n$ bits, where n is the size of the input and k is some constant, although larger word sizes are sometimes considered for certain problems (e.g., sorting integers). In general, it is usually fine to ignore the issue of word size as long as we are not trying to do unreasonable things, like storing and operating

[1] This is called the "unit cost" RAM, since arithmetic operations on whole words are assumed to take a single step. One also finds a "logarithmic cost" RAM in which some arithmetic operations on integers of size at most C take time proportional to $\log C$. The logarithmic cost model makes sense if you look at the size of the physical circuits required to implement these operations.

on unrealistically large numbers. We will try to warn the reader when this is not the case.

Although the RAM is a reasonable model for a single-processor digital computer, some of its assumptions are overly simplistic. For example, the RAM assumes that all operations take the same amount of time, while in reality certain operations like multiplication may take longer to perform than simpler operations like addition. Also, memory accesses usually take substantially more time than primitive arithmetic operations, and as a result of caching, multiple memory accesses close to each-other take much less time than those spread out haphazardly throughout memory; we will elaborate on this point later in the chapter.

1.2.2 The Real RAM

One of the limitations of the RAM model is that it can only operate on integers (and also rational numbers, being ratios of integers). However, irrational numbers also arise in a variety of common problems, such as geometry problems involving distances in the 2D plane. Irrational numbers have no convenient representation in the RAM model, and this might seem perfectly reasonable since irrational numbers have no convenient representation on a digital computer. However, in order to simplify the algorithm design process, it is sometimes helpful to pretend that our model of computation can deal with irrational numbers. This gives a model known as the *real RAM*, which is the same as the standard RAM except it can store and operate on real numbers as well as integers. That is, words in memory can be designated as either integers or reals, or alternatively you can regard the model as having two separate memories, one for integers and the other for reals[2].

Many algorithms fundamentally require and exploit integrality in their input data, using tricks such as *lookup tables* and *hashing* (discussed further in Chapter 7) that use input elements as array indices or that require the ability to perform integer division[3]. Other algorithms make no such integrality assumptions, working equally well if they are fed integers as if they are fed real numbers. The distinction between these two types of algorithms is important to keep in mind. For lack of better terminology, we will often call them "RAM algorithms" versus "real RAM algorithms", although the way we have defined the real RAM as a superset of the RAM, it is certainly possible to run a RAM algorithm on the real RAM as long as we are careful to designate its input as integral (there would be little point to doing this, however, since we aren't using any of the features of the real RAM). RAM algorithms often run faster than their real RAM counterparts, since they can exploit the integrality of their input.

[2] Distinguishing between real versus integer words in memory is important for a subtle reason: we typically do not allow the real RAM to truncate a real number into an integer, since computation of $\lfloor x \rfloor$ and $\lceil x \rceil$ for a real number x makes the model unrealistically powerful, just like a standard RAM with unbounded word size. For example, the expression $\lfloor 2^k x \rfloor - 2\lfloor 2^{k-1} x \rfloor$ tells us the value of the kth bit after the decimal point in the binary representation of a real number x. Using this formula, we could therefore access an arbitrarily large amount of information carefully packed within the infinite digit string of a single real number x.

[3] An example of integer division: 17 divided by 5 yields 3 with a remainder of 2. Since we forbid use of the floor function on a real RAM, we can determine that the real number 17 divided by the real number 5 equals the real number 3.4, but we cannot easily obtain an integer quotient and remainder.

1.2.3 Comparison-Based Algorithms

For non-numeric problems like sorting and searching, a nice model of computation due to its simplicity is the *comparison-based* model, where arithmetic on input elements is forbidden; rather, we can only learn about elements of the input by comparing them pairwise. The comparison-based model is nearly identical to the RAM; the only difference is that if a memory location holds an element of data that comes directly from the input, then we must treat this element as a "black box" that can only be the subject of comparisons, not arithmetic operations. We will use the comparison model often when we study algorithms and data structures for searching and sorting. The model is ideal for these problems since it gives us very general algorithms that operate on any type of comparable data: integers, real numbers, text strings, etc. This generality comes at a price, however, since we shall see that many problems (e.g., sorting) can be solved faster on a RAM by exploiting integrality of the input.

1.2.4 Other Models

This book generally confines its discussion to the RAM, real RAM, and comparison-based models. However, just as computing systems come in many shapes and sizes, there are many more models of computation one can consider. Towards the end of this chapter, we briefly introduce the *cache-oblivious* model, which captures the performance of a realistic multi-level memory system much more faithfully than the RAM. We also briefly highlight models of *parallel* computation, which are becoming increasingly important due to the massive size of many modern computing problems, as well as the increasing availability of multi-core and distributed computing environments. Parallel models of computation are quite a bit more complicated since they need to describe how multiple processors are synchronized, how they share memory, how they communicate, and more.

There are also several more "exotic" choices for models of computation out there, many based on computers built to exploit physical principles such as:

- **Optics.** Shining light through a diffraction grating yields a pattern that gives the Fourier transform of the pattern on the grating, thereby "instantly computing" a Fourier transform.

- **Mechanics.** Shortest paths from node x in a network can be computed "instantly" by gravity by building the network out of balls and strings and suspending it from the ball representing x; the distance each node y falls relative to x represents the shortest path distance from x to y.

- **Quantum Mechanics.** A recent breakthrough of Peter Shor enables factoring large integers on a computer based on quantum principles much faster than we know how to factor on any other computational model.

- **Chemical Interaction.** A famous study of Leonard Adelman shows how the Hamiltonian path problem can be solved by mixing pieces of carefully constructed DNA representing parts of a graph in such a way that they would tend to bind together in a configuration representing a Hamiltonian path.

All of these have drawbacks that severely limit their viability in practice for general-purpose computation at the present time, but it is still good to keep in the back of your mind the idea that there may be a much better underlying "computer" possible for any particular problem you are considering. Nonetheless, even if fundamentally new models become common in the future due to dramatic changes in technology, you will still likely be well-served by a solid foundation in algorithmic problem-solving techniques based on the simpler models considered in this book.

1.3 How to Describe an Algorithm

Algorithms can be described in various levels of detail. For example, a binary search for some value v within a sorted array $A[1 \ldots n]$ can be described in high-level technical prose as follows:

> Compare v to the middle element of the array. If these match we are done. If v is smaller, recursively repeat our search on the first half of the array; if larger, repeat instead on the second half of the array. If our search narrows down to an empty subarray the process terminates, having determined that v is not present in A.

A good description of an algorithm leaves no important aspect of the algorithm's behavior unclear. Cumbersome minor details can often be safely omitted. For example, if our array has even length, then either $A[n/2]$ or $A[n/2+1]$ can serve as a "middle" element. A skilled programmer should be able to implement the algorithm being described without needing to spend time re-deriving key details.

For more detail, an algorithm may also be described like a computer program, either in terms of actual code or abstract *pseudocode*. Since this book focuses on high-level ideas rather than serving as a "practitioner's handbook", most of its algorithms are explained in high-level prose rather than code. There are plenty of other books and websites that provide good examples of specific algorithms in code. When we do provide examples of code, we use pseudocode for several reasons. As we see in Figure 1.3(a-b), pseudocode is quite similar in structure to most popular high-level programming languages, so implementing an algorithm based on a pseudocode description is straightforward. The generality of pseudocode also frees us from the necessity of releasing new revisions of the book every time a new language becomes popular. Furthermore, pseudocode avoids the administrative requirements of most programming languages (e.g., declaring variables), allowing us to focus entirely on algorithmic structure.

Iteration Versus Recursion. Loops and repetition are found in most algorithms, and these usually come in one of two flavors: iteration and recursion. For example, if we consider summing the contents of an array $A[1 \ldots n]$, an *iterative* algorithm would be described as looping sequentially through the array while maintaining a running sum. A *recursive* algorithm would add the first element $A[1]$ to the sum it gets when it recursively applies itself to the remainder of the array $A[2 \ldots n]$. The choice between describing an algorithm iteratively or recursively is often a matter of personal preference, although in many cases one of the two methods leads to a simpler exposition, implementation, or analysis. Pseudocode for a recursive

(a)

```
int bsearch(int A[], int n, int v)
{
   int left = 0, right = n - 1, mid;
   while (left <= right) {
     mid = (left + right)/2;
     if (v == A[mid]) return mid;
     if (v < A[mid]) right = mid - 1;
     else left = mid + 1;
   }
   return -1; /* Not Found */
}
```

(b)

```
Binary-Search:
$left \leftarrow 1$, $right \leftarrow n$
While $left \leq right$:
   $mid \leftarrow \lfloor (left + right)/2 \rfloor$
   If $v = A[mid]$:  Return $mid$
   If $v < A[mid]$:  $right \leftarrow mid - 1$
   If $v > A[mid]$:  $left \leftarrow mid + 1$
Return "Not Found"
```

(c)

```
Rec-Binary-Search($A, left, right, v$):
If $left > right$:  Return "Not Found"
$mid \leftarrow \lfloor (left + right)/2 \rfloor$
If $v = A[mid]$:  Return $mid$
If $v < A[mid]$:  Return Rec-Binary-Search($A$, $left$, $mid - 1$, $v$)
If $v > A[mid]$:  Return Rec-Binary-Search($A$, $mid + 1$, $right$, $v$)
```

FIGURE 1.3: The binary search algorithm: (a) in C/C++, and in pseudocode, (b) from an iterative perspective and (c) from a recursive perspective.

implementation of binary search is shown in Figure 1.3(c).

The Importance of Abstraction. Let us now reconcile our high-level means of describing an algorithm with the low-level RAM computational model, which is only capable of performing very simple fundamental operations like adding two words. In order to accurately analyze an algorithm's running time in this model, it might seem necessary to express the algorithm in the language of the RAM, which is similar to *assembly language* on a digital computer with a simple instruction set. An example of binary search written this way is shown Figure 1.4; note how every instruction is a fundamental operation like the addition or comparison of two words stored in CPU registers. This level of detail shows us precisely what the algorithm is doing, and allows us to compute the exact number of fundamental steps performed by the algorithm. However, as we shall see in the next section, this level of detail is unnecessary when we use *asymptotic analysis* to describe the running time. Moreover, such a low-level description rarely helps in explaining the algorithm's structure and the intuition behind its operation.

The discussion above motivates the importance of *abstraction* in computer science. When describing algorithms, we should try to focus as much as possible on only the most relevant high-level details and free our minds from distracting lower-level

```
 Initialize:  MOV r1, 0         Store left index in register r1
              LOAD r2, n        Right index in r2
              DEC r2            Decrement r2 (array is zero-based)
              LOAD r3, v        Value to search for in r3
   MainLoop:  CMP r1, r2        Compare r1 with r2
              JG NotFound       Jump if greater than
              ADD r0, r1, r2    Set r0 to r1 + r2
              SHR r0, 1         Shift r0 right by 1 (i.e., divide by 2)
              MOV r4, A         Move base address of A into r4
              ADD r4, r4, r0    Set r4 to address of A[r0]
              LOAD r5, r4       Load r5 with value of A[r0]
              CMP r3, r5        Compare r3 with r5
              JE Found          Equal? We've found our element...
              CMP r3, r5        Compare again
              JG SecondHalf     Greater? Restrict search to 2nd half
  FirstHalf:  MOV r2, r0        Set new right index
              DEC r2
              JMP MainLoop      Loop again
 SecondHalf:  MOV r1, r0        Set new left index
              INC r1
              JMP MainLoop      Loop again
      Found:  RET               Return (index of element is in r0)
   NotFound:  MOV r0, -1        Unsuccessful return value of -1
              RET               Return
```

FIGURE 1.4: Binary search written in (pseudo-)assembly language.

details. As this book progresses, we will use the algorithms and data structures we develop as "black boxes" to build successively larger and more complex algorithms. As a simple example, consider the 2-SUM problem: given an array $A[1 \ldots n]$, do two numbers exist in A summing to a specified value v? A simple algorithm for 2-SUM is the following: first *sort* A, then scan through it and use *binary search* to check for each element $A[i]$ whether a "partner" element of value $v - A[i]$ also exists in the array (remember that binary search requires a sorted array to work properly). By abstracting away the details of sorting and binary search, we have greatly simplified the exposition of our more sophisticated algorithm.

1.4 Characterizing Algorithm Performance

A good algorithm makes efficient use of computational resources, which can include processor, memory, network bandwidth, power consumption, and more. In this book, we focus most of our attention on minimizing *running time.*

1.4.1 Empirical Testing

An obvious way to develop understanding of the performance of an algorithm is by experimental measurement, by running the algorithm on inputs of different sizes.

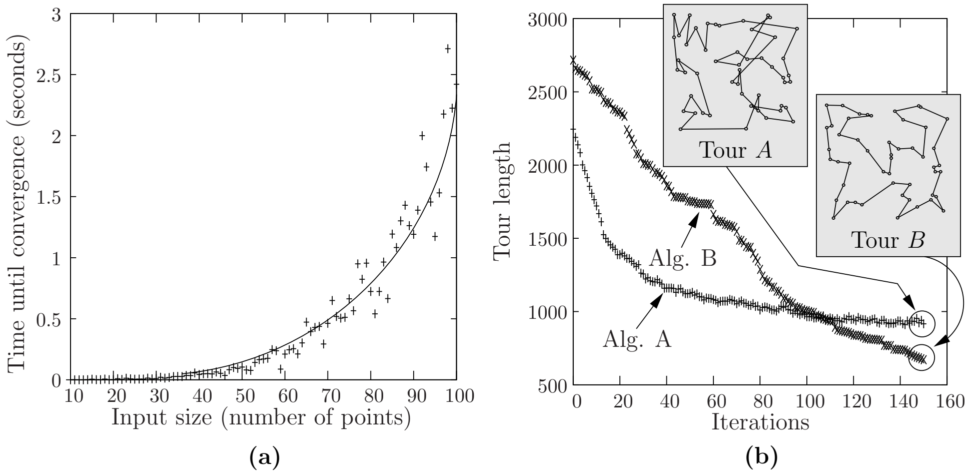


FIGURE 1.5: Empirical testing of algorithms for the traveling salesman problem: (a) running time until convergence measured on random inputs of different sizes, with a curve fit to this data; (b) convergence of two algorithms on an instance with $n = 50$ input points, along with the final tour produced by each one.

For example, in Figure 1.5(a) we investigate the running time of an algorithm for the famous traveling salesman problem (TSP), which asks us to find a minimum-length tour (Hamiltonian cycle) of a set of n cities. By running the algorithm on a set of randomly-generated inputs for varying n (here, we use n random points in the 2D plane), we can observe how the running time scales with problem size.

Being a hard problem, we do not know how to optimally solve the TSP quickly. Most TSP algorithms, including the one studied in (a), therefore typically converge to a sub-optimal but still reasonably good solution over a number of iterations. In Figure 1.5(b), we show how empirical testing gives us a good sense of the rate of convergence of two different algorithms: algorithm A converges more quickly, but algorithm B ultimately produces a shorter tour.

Empirical testing is a simple and effective way to measure the anticipated performance of an algorithm on a real computing environment. It is arguably the only feasible way to analyze some algorithms, particularly those that are too complex to analyze mathematically, or those whose performance on "real world" inputs often differs substantially from what we would predict based on mathematics alone. It is also the only way to measure the impact of aspects of a real computing environment that are not captured by our abstract computing model, such as memory caching artifacts; these can sometimes lead to surprising differences between actual and theoretically-predicted performance.

On the negative side, empirical testing does not tell us the whole picture in many ways. It only measures performance for a specific hardware platform, operating system, and programming language, and this may not translate perfectly to other systems. If you ask two highly-skilled programmers to implement the same algorithm, subtle differences in their code will likely give you different performance measurements. Most importantly, however, it can be very challenging to select an appropriate set of inputs on which to test. Ideally, one should use inputs that we expect to encounter in practice, although these may be so well-structured that they never reveal glaring deficiencies in the algorithm. Randomly-generated inputs are often used in empirical testing in the literature, although these can often lead to vastly-differing performance than real-world inputs. It can be a genuine challenge to conduct empirical testing of an algorithm in a truly rigorous fashion.

1.4.2 Mathematical Analysis Using Asymptotic Notation

The most important aspect of an algorithm's performance is how it scales with input size, and we can often predict this behavior by studying the mathematical structure of the algorithm. For example, after a fair amount of work, one could ascertain that an algorithm uses between $7n^2 - 3n + 2$ and $8n^2 + 17$ fundamental RAM operations to solve a problem of size n. Although this is a very precise set of bounds, it is almost too much detail — what really matters here is simply that running time scales as a quadratic function of input size. As n grows large, leading constant factors like the 7 or 8 as well as lower-order terms become increasingly irrelevant. We therefore say that our running time is on the order of n^2, which we write in using "Big Oh" notation as $O(n^2)$.

We say that a mathematical expression is $O(f(n))$ if it is bounded above by some constant times $f(n)$ as n grows sufficiently large. For example, the running time expression $8n^2 + 17$ is $O(n^2)$ since it is upper-bounded by $9n^2$ for $n \geq 5$. This is called *asymptotic analysis* because it describes asymptotic behavior as n grows very large, where the fastest-growing term dominates. Asymptotic analysis captures the essence of what matters in a running time, and allows us to analyze algorithms at a high level without the need for painstaking translation into assembly language to count individual operations. For example, if we want to check whether an n-element array contains two identical elements (the *element uniqueness problem*), we could iterate over every pair of elements and compare them. Since we are examining $\binom{n}{2} = n(n-1)/2 = O(n^2)$ pairs of elements and spending a constant amount of work (e.g., a comparison operation as well as a small amount of loop overhead) on each pair, this algorithm runs in $O(n^2)$ time.

Hidden Constants. Suppose algorithm A runs in $O(n)$ time, and algorithm B runs in $O(n^2)$ time. Since asymptotic expressions ignore leading constants and lower-order terms, it is actually impossible to predict without further information or empirical testing which will run faster in practice for a particular n. We know only that algorithm A will eventually triumph as n grows large. If the actual running times of A and B are $999999n$ and n^2 respectively, then indeed we lose some helpful information by saying only that "A's running time is $O(n)$". In this case, we may wish to point out that A's running time has a large *hidden constant*, so it is clear that among $O(n)$ algorithms, A is not terribly fast.

1.4.3 Worst-Case and Average-Case Behavior

If we are lucky, binary search might terminate after only a single comparison, although most invocations take more time. This is a common phenomenon, where the running time depends on input *structure* as well as size. There are "easy" inputs the algorithm can handle quickly and "hard" inputs that take longer to process. In this situation, we typically focus on *worst-case* running time. For binary search, every unsuccessful iteration narrows the size of the subarray we are searching by at least a factor of 2, so after at most $\log_2 n$ such iterations our search is narrowed to a single element and the algorithm terminates. Since every iteration involves a constant number of fundamental operations, binary search therefore has an $O(\log n)$ worst-case running time (note that we can simply say $O(\log n)$ instead of $O(\log_2 n)$ since logs of all bases differ only by constant factors, and leading constant factors disappear inside asymptotic expressions). Algorithm designers focus on worst-case behavior not because they are pessimists, but because this provides a very strong guarantee — no matter how bad our luck, or even if we receive input from a malicious adversary, binary search *always* runs in $O(\log n)$ time.

If worst-case inputs are rarely seen in practice, then *average-case* analysis may be preferable, where we assume a particular probability distribution over all possible inputs of size n (presumably the distribution we expect to see in practice) and compute the expected running time. For example, if we are equally likely to search for any of the n elements in our input array, the expected running time of binary search is still $O(\log n)$. Computing the average-case performance of a complicated algorithm is unfortunately often challenging from a mathematical perspective; for such algorithms, empirical testing may be the only way we can measure anticipated performance in practice.

1.4.4 Common Running Times

It is good to be intuitively familiar with common running times encountered during algorithmic analysis. The fastest possible running time is an $O(1)$ (i.e., *constant*) worst-case running time — upper bounded by a fixed universal constant independent of problem size. However, although parts of an algorithm often take $O(1)$ time, it is unusual for an entire algorithm with input size n to run in $O(1)$ time, since any *sublinear* algorithm (faster than $O(n)$ time) does not even have enough time to examine its entire input.

Logarithmic running times of $O(\log n)$ are typical for algorithms like binary search that in each step are able to reduce a problem's size by a constant factor (say, by half). More generally, running times like $O(\log^2 n)$ or $O(\log \log n)$ or $O(\sqrt{\log n})$ are known as *polylogarithmic* running times since they are bounded by some polynomial function of $\log n$. These are all very fast running times — even for a ridiculous input size like $n = 2^{100}$, binary search requires at most 100 iterations.

Running times like $O(n)$ (i.e., *linear*), $O(n \log n)$, $O(n^2)$ (i.e., *quadratic*), and $O(n^3)$ (i.e., *cubic*), are called *polynomial* running times since they are bounded by a polynomial in the input size n. Linear-time algorithms and $O(n \log n)$ algorithms have good performance, since these algorithms require roughly the same amount of time to execute as it takes to simply read their input. As we start moving toward quadratic running times and higher, we begin to encounter more substantial limitations on the sizes of problems we can quickly solve.

Beyond polynomial time, things get very bad very fast. Algorithms with *exponential* running times like $O(2^n)$ and *factorial* running times like $O(n!)$ can only solve very

				Running Time	Largest value of n for which computation takes at most...		
					1 millisec	1 hour	1 century
Polynomial (i.e., "efficient")	Sublinear	Polylogarithmic	Constant	$O(1)$	unlimited (no dependence on n)		
				$O(\alpha(n))$	very large (in excess of 10^{100})		
				$O(\log^* n)$	very large		
				$O(\log\log n)$	very large		
				$O(\sqrt{\log n})$	very large		
			Logarithmic	$O(\log n)$	very large		
				$O(\log^2 n)$	very large		
				$O(\sqrt{n})$	10^{12}	1.3×10^{25}	9.9×10^{36}
			Linear	$O(n)$	10^6	3.6×10^{12}	3.2×10^{18}
				$O(n \log n)$	6.3×10^4	9.9×10^{10}	5.7×10^{16}
				$O(n\sqrt{n})$	10^4	2.3×10^8	2.2×10^{12}
			Quadratic	$O(n^2)$	1000	1.9×10^6	1.8×10^9
			Cubic	$O(n^3)$	100	1.5×10^4	1.5×10^6
			Quasipoly	$O(n^{\log n})$	22	87	228
			Exponential	$O(2^n)$	19	41	61
			Factorial	$O(n!)$	9	15	20
				$O(n^n)$	7	11	15

FIGURE 1.6: List of common run times, ordered from fastest to slowest. To compute the approximate maximum value of n for which computation takes a certain amount of time, we assume that all logs are base 2 and that the leading constant in the $O(\cdot)$ expressions is such that our algorithm can perform 1 billion iterations per second (this is somewhat aggressive for modern personal computers at the time of writing). The special functions $\alpha(n)$ and $\log^* n$ are defined in the next chapter.

small problem instances even given centuries of computing time. These functions grow so quickly that even if we were to construct computing devices that were a million times faster than those of today, an algorithm running in, say, 2^n steps could still only solve problems that were of size $n + 20$ in roughly the same amount of time it takes to solve a problem of size n today. Unfortunately, exponential running times seem unavoidable for certain problems, such as the NP-complete problems.

"Efficient" Algorithms. To the computer scientist, the threshold between polynomial and super-polynomial (e.g., exponential) running time is particularly notable, since historically this has been regarded as the threshold between "efficient" and "inefficient". In fact, theoreticians often equate the word "efficient" with "polynomial time". It may seem a bit dubious to call an algorithm with polynomial running time $O(n^{100})$ "efficient" (in theoretical computer science, one actually does encounter such running times!), but one must bear in mind that this is still better than $O(2^n)$ for very large values of n. In this book we generally look for "efficient" solutions that not only run in polynomial time, but are viable in practice as well.

1.4.5 Output Sensitivity

Algorithms that solve numerical problems come in two flavors: *exact* algorithms terminate in a finite amount of time with an exact answer, even on real-valued input[4], while other algorithms converge asymptotically to a correct answer over time. For example, if we start with $x = 1$ and repeatedly set $x = x/2 + 1/x$, then this quickly approaches $x = \sqrt{2}$ [Why?]. However, since $\sqrt{2}$ is irrational, the algorithm theoretically never terminates, and only gets closer and closer to $\sqrt{2}$ the longer it runs. For iteratively-converging algorithms like this, running time is usually characterized in terms of *convergence rate*, the amount of time required to obtain each successive digit of accuracy in the output.

We could say that the running time of an iteratively-converging algorithm is *output-sensitive*, meaning that it depends on how much output precision is requested. We will see another form of output sensitivity when we study data structures, since the running time of a data structure query often depends on the amount of data returned by the query. For example, a query operation over a data structure of size n might have running time $O(k + \log n)$, where k is the size of the output, and $O(\log n)$ is the fixed overhead of the query independent of output size.

1.4.6 Input Sensitivity

For many algorithms, running time depends only on the number of input elements n. However, some RAM algorithms have *input-sensitive* running times that depend on n as well as the magnitude of the n integers provided as input. For instance, suppose our input consists of n integers in the range $0 \ldots C - 1$, so each is described by $\log_2 C$ bits. Our running time is *strongly polynomial* if it depends polynomially on just n (i.e., on the number of *words* in the input), and *weakly polynomial* if it depends polynomially on n and $\log C$ (i.e., on the number of *bits* in the input). Since complexity theory measures the true input size to an algorithm in bits, both types of algorithms rightfully run in "polynomial time". Given a choice between the two, strong polynomial time is generally preferred, for aesthetic simplicity as well as the peace of mind that large numbers in the input have no impact on running time. For some problems, however, if we know that C is small, we might be able to devise an input-sensitive algorithm with a weakly polynomial running time that is faster in practice. The distinction between strong versus weak polynomial time generally applies only to algorithms with integer inputs.

A running time depending polynomially on n and C (rather than log C) is said to be *pseudo-polynomial.* This term is somewhat misleading, since pseudo-polynomial running times do not count as polynomial running times at all, being exponential in the size of the input measured in bits (since C is exponentially large in $\log C$). Pseudo-polynomial running times like $O(nC)$ scale gracefully in terms of the number

[4] Since actual digital computers cannot store real numbers perfectly, even a so-called "exact" numerical algorithm might end up producing a solution that isn't exactly correct due to accumulated round-off errors (this is one danger of the real RAM, since it allows us to pretend this issue does not exist in practice). We address this pitfall in further detail in Chapters **??** and **??**.

of integer input elements, but not in terms of the size of these elements. A pseudo-polynomial algorithm might therefore take 100 times as long to run if the numbers in its input are multiplied by 100, so caution is advised when considering the use of such algorithms. [Examples of algorithms of the three types above]

1.4.7 Lower Bounds and Optimal Running Times

Since $O(\cdot)$ notation provides merely an asymptotic upper bound, we could truthfully claim that an algorithm runs in $O(n^2)$ time even if it actually runs in only constant time. In this case, we would say the $O(n^2)$ bound is not *tight.* To be more precise in our asymptotic analysis, let us introduce two more expressions. We use $\Omega(\cdot)$ ("Big Omega") to indicate an asymptotic lower bound. An algorithm has running time $\Omega(n^2)$ if the running time is bounded below by some constant time n^2 as n grows sufficiently large. We use the notation $\Theta(\cdot)$ to indicate both asymptotic lower and upper bounds; running time is $\Theta(n^2)$ if it is both $O(n^2)$ and $\Omega(n^2)$. One may wish to think of O, Ω, and Θ as the asymptotic equivalents of $\leq$, $\geq$, and $=$. That is, an $O(n^2)$ algorithm has a running time that is asymptotically no worse than n^2, an $\Omega(n^2)$ algorithm has running time that is asymptotically at least as bad as n^2, and a $\Theta(n^2)$ algorithm has a running time that grows precisely at an asymptotic rate of n^2, ignoring constant factors and lower-order terms.

We can often characterize worst-case running time more precisely using $\Theta(\cdot)$ notation. For example, it is correct to say that binary search runs in $O(\log n)$ time, but we convey more information by saying it runs in $\Theta(\log n)$ time in the worst case. In much of the technical literature, one finds $O(\cdot)$ being used when $\Theta(\cdot)$ is perhaps more appropriate. People will often say, for example, that a running time is $O(n^2)$ when they really mean either $\Theta(n^2)$, or $\Theta(n^2)$ in the worst case.

Consider the problem of computing the maximum value in an n-element array. We can easily solve this problem in $O(n)$ time (actually $\Theta(n)$ would be more precise). However, note that *any* algorithm that solves this problem must at least look at all n values in the input. As a result, we can say that there is a lower bound of $\Omega(n)$ on the worst-case running time of any algorithm that solves this problem, and that our $O(n)$ algorithm therefore has an *optimal* worst-case running time. In Chapter 3 we will learn how to prove more sophisticated lower bounds on certain problems. For example, any comparison-based or real RAM algorithm that sorts n elements must spend $\Omega(n \log n)$ time in the worst case.

A somewhat playful outlook of the field of algorithms is in terms of an epic struggle between two forces: the "good guys" who design efficient algorithms and prove that you *can* solve certain problems efficiently, and the "bad guys" who prove lower bounds indicating that you *cannot* solve certain problem efficiently. When the two sides meet at the same asymptotic running time, this means the complexity of a problem is in some sense resolved. However, there are many problems for which there is still an unsightly gap between the best-known upper bound and the best-known lower bound. For example, the problem of multiplying two $n \times n$ matrices has a trivial lower bound of $\Omega(n^2)$ due to the need to examine all input data, but the best known upper bound is currently $O(n^{2.3727})$. In this case, one would hope that clever researchers will eventually close the gap by proving a stronger lower bound or developing an algorithm that improves the upper bound. One might also hope that the upper bound would be the one to change, since nature would be somewhat cruel if an arbitrary-looking running time like $O(n^{2.3727})$ was actually optimal for such a fundamental problem!

1.5 Data Structures

The study of *data structures* goes hand in hand with the study of algorithms. Algorithmic problems are typically posed in terms of abstract mathematical entities such as sequences, sets, graphs, etc., which can be represented in memory several different natural ways. Choosing the best representation for our data is of fundamental importance when designing algorithms, since it can have a dramatic impact on performance. One often finds the subjects of "algorithms" and "data structures" treated separately, for instance in two different courses in an undergraduate computer science curriculum. However, these two subjects should ideally be studied together as part of the same whole, since they are intrinsically coupled. We will study data structures extensively in Chapters 4 through 9; for now, we simply wish to motivate their importance and highlight some key concepts. We also discuss two very fundamental data structures, *arrays* and *linked lists*, as a prerequisite for the next few chapters.

A data structure provides a concrete strategy for storing and interacting with data. For example, an array $A[1 \ldots n]$ is a very simple data structure consisting of n consecutive elements in memory, and it provides a simple and natural way to represent an abstract mathematical sequence $A_1 \ldots A_n$ (we use brackets $A[i]$ to denote the elements of an array, where subscripts A_i are used to represent elements in an abstract mathematical sequence).

Abstract Data Types. There is an important distinction to be made between the abstract specification of a data structure (e.g., a sequence, set, or map), also known as an *abstract data type*, and a concrete implementation of a data structure in accordance with this specification (e.g., an array). An abstract data type only describes the operations to be supported by a data structure, and does not prescribe any particular way of implementing these operations. For example, a data structure for a *dynamic sequence* $A_1 \ldots A_n$ must support the following fundamental operations:

- *Access*(A, i). Retrieves the value of the ith element, A_i.
- *Modify*(A, i, v). Sets the value of A_i to v.
- *Insert*(A, i, v). Inserts a new element of value v in the ith position of A. All former elements $A_i, A_{i+1}, \ldots$ are shifted upward by one index (so they become $A_{i+1}, A_{i+2}, \ldots$) to make room for the new element.
- *Delete*(A, i). Deletes A_i from the sequence. Former elements $A_{i+1}, A_{i+2}, \ldots$ are shifted downward by one index (to $A_i, A_{i+1}, \ldots$) to close the gap created by the deleted element.

There are often many different ways to implement a given abstract data type. For a dynamic sequence, the simplest two alternatives are arrays and linked lists, although later in Chapter 6 we will discuss many others, such as balanced trees and skip lists.

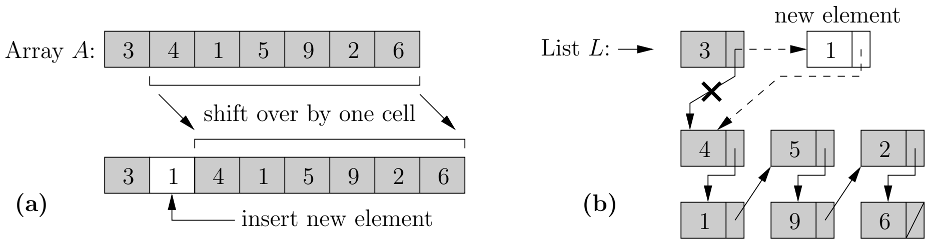


FIGURE 1.7: Insertion of a new element into (a) an array and (b) a linked list.

Static Versus Dynamic Data Structure. A *static* data structure is built once, after which it supports operations to query but not modify its state. Some data structures are so static that they can even operate from read-only memory after being built. *Dynamic* data structures like dynamic sequences allow both modification and query operations. As one might expect, dynamic data structures can be more challenging to design than their static counterparts.

Incremental, Decremental, and Fully-Dynamic Structures. Dynamic data structures come in three flavors: *incremental* structures can handle insertions of new elements but not deletions, *decremental* structures can handle deletions but not insertions, and *fully dynamic* structures can handle both. The fully dynamic case is, not surprisingly, often much more difficult. For example, a *priority queue* is a type of data structure that tracks the minimum of a dynamic set of elements. Incremental and decremental priority queues are relatively easy to implement, but the fully dynamic case leads to an entire chapter worth of discussion (Chapter 5).

Dynamic Algorithms Versus Data Structures. The traditional "one-off" model for solving an algorithmic problem asks us to read the input, perform some processing, write the output, and then terminate. Given a second instance differing little from the first, we might hope to solve it more efficiently by maintaining some of the state from the previous computation in an appropriate data structure. This is sometimes known as *re-optimizing* a solution, when re-solving an optimization problem after minor modification of its input. We can consider making nearly any algorithmic problem dynamic in this fashion, switching our outlook on the problem to one more centered on data structures.

1.5.1 Arrays Versus Linked Lists

Consider how to implement a dynamic sequence. If we use an array, then the *access* and *modify* operations run very quickly, in $O(1)$ time. However, arrays are not well-suited for insertion or deletion. As we see in Figure 1.7(a), inserting a new element into an array takes $\Theta(n)$ worst-case time since we first need to slide over potentially all of the array elements by one cell to make room for the new element. Similarly, deletion takes $\Theta(n)$ worst-case time since we may need to slide a large block of elements back by one cell to plug the hole created by the deleted element[5].

[5] More sophisticated data structures (e.g., problem 73) mitigate this issue by leaving periodic "gaps" in an array, so insertions and deletions do not need to displace so many elements.

	Array	Linked List	Balanced Tree or Skip List
Access	$O(1)$	$O(n)$	$O(\log n)$
Modify	$O(1)$	$O(1)$ $(+O(n))$	$O(\log n)$
Insert/Delete	$O(n)$	$O(1)$ $(+O(n))$	$O(\log n)$

FIGURE 1.8: Running times for the operations of three types of data structures for representing a dynamic sequence. To be more precise, we could say that the running time of *modify*, *insert*, and *delete* in a linked list is really $O(1)$ once we have scanned (in $O(n)$ time) to the appropriate location in the list. We will study balanced trees and skip lists later in Chapter 6.

Linked lists give us another simple way to implement a dynamic sequence. As shown in Figure 1.7(b), the elements of the sequence are stored in haphazard memory locations, and each element maintains a pointer to its successor. The end of the list is usually indicated by a special "null" pointer from the final element, or by including a dummy *sentinel* element as the final element[6]. The efficient operations on a linked list are exactly the opposite of those on an array. It takes $\Theta(n)$ time in the worst case to access a particular element given its index, as we must walk down the list from the beginning until we reach the desired element. However, once we have scanned to the appropriate location in a list, we can *modify* it in $O(1)$ time and also *insert* and *delete* in $O(1)$ time, since this requires only the modification of a small handful of pointers as opposed to the relocation of massive amounts of data (unfortunately, since we must include the scanning time in these operations, their running times are also technically $\Theta(n)$ in the worst case). Occasionally we will find it convenient to be able to walk backwards as well as forwards in a list, in which case we can use a *doubly-linked list* where each element points to both its predecessor and its successor[7].

As shown in Figure 1.8, there is a dramatic trade-off between the array and linked list in terms of which operations are efficient. The best data structure for implementing a dynamic sequence therefore depends on our particular application. If we expect to perform few insertions and deletions, the array may be ideal, and if all insertions and deletions are concentrated in a small area, the linked list may do better. Trade-offs of this sort are extremely common when we study data structures, and often there is no single "best" implementation for a particular type of data structure.

As a note to the introductory student, arrays and linked lists are such fundamental data structures that it is *absolutely crucial* to understand them well; for example, new students often mistakenly try to insert new elements in $O(1)$ time in the mid-

[6] If our list ends with a dummy sentinel element, then a simple way to delete element e given a pointer to e is simply to overwrite e with a copy of its successor. Without a dummy sentinel element, however, this trick no longer works (specifically, for deleting the final element), and we can only effectively delete e if we have a pointer to e's predecessor.

[7] Instead of storing the predecessor pointer p and successor pointer s separately, a clever space-saving trick is to store just the single value $p \oplus s$, where $\oplus$ is the XOR operation. Owing to how XOR works, this still allows effective navigation in either direction. For example, when traveling from p, we can take $p \oplus (p \oplus s) = s$ to recover s, and vice versa.

dle of an array. A programming background usually helps, since arrays and linked lists are familiar objects in most computer programs. If you are unsure about your background, arrays and linked lists are covered in excruciating detail in many other books, which you may wish to consult for reference. The following are two fun yet challenging problems that can test your mastery of linked lists and arrays.

Problem 1 (Loopy Linked Lists). Suppose we are given a pointer to the first element in an n-element linked list (we aren't given n, however), and told that it ends in a loop, where the last element points back to some earlier element in the list. We want to compute the number of elements in the list, n. This is easy to do if we use $O(n)$ extra memory, since we can attach a marker to each element as we scan through the list, making it easy to detect when we revisit an element. However, in some cases it may be undesirable to modify the list, such as in a parallel shared memory environment with several processes accessing the list. Try to devise a method that computes n in $O(n)$ time without modifying the list, and using only $O(1)$ auxiliary memory. As a hint, consider the looping and non-looping parts of the list separately. [Solution]

Problem 2 (Virtual Initialization). Arrays must typically be initialized prior to use, since when we allocate an array of n words of memory, they usually start out filled with "garbage" values (whatever data last occupied that block of memory). In this problem, we wish to design a data structure that behaves like an array (i.e., allowing us to retrieve the ith value and modify the ith value both in $O(1)$ time), but which allows for initialization to a specified value v in only $O(1)$ time, instead of the usual $\Theta(n)$ time. If we ask for the value of an element we have not modified since the last initialization, the result should be v. The data structure should occupy $O(n)$ space in memory (note that this could be twice or three times as large as the actual space we need to store the elements of the array), and it should function properly regardless of whatever garbage is initially present in this memory. As a hint, try to combine two different representations of the data in the array. [Solution]

The trick behind the first problem can be used in a surprisingly wide range of applications, from parallel processing (problem 51(l)) to infinite loop detection (problem 116) to factoring integers (Section **??**). As a consequence of the second problem, we can assume in theory that initialization of an array requires only constant time without affecting the asymptotic time or space requirements of our algorithms (however, this technique is rarely used, since it requires so much extra space).

1.5.2 Stacks and Queues

While on the subject of dynamic sequences, arrays, and linked lists, now is a good time to mention two useful related structures we often encounter. A *stack* is a type of data structure supporting these two operations:

- *Push*(e). Inserts a new element e into the stack.
- *Pop*. Removes and returns the most-recently-inserted element.

Elements pass through a stack in a *Last-In-First-Out* (LIFO) fashion (picture a stack of papers, where we only add or remove papers at the top of the stack). On the other hand, a *queue* is a type of data structure that follows the *First-In-First-Out* (FIFO) discipline (picture a line of people waiting at a busy ticket counter).

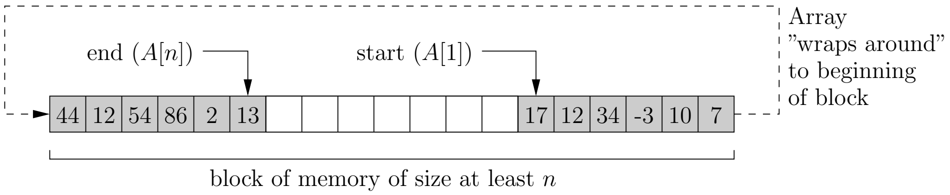


FIGURE 1.9: Illustration of a circular array $A[1 \ldots n]$ stored within a larger block of allocated memory.

Queues support the following two operations:

- *Enqueue*(e). Inserts a new element e into the queue.
- *Dequeue*. Removes and returns the least-recently-inserted element.

A stack is just a dynamic sequence in which insertions and removals occur at only one end. For a queue, insertions occur at one end and removals at the other. Both arrays and (doubly-)linked lists can easily handle insertion and deletion at the ends of a sequence in $O(1)$ time (e.g., by maintaining pointers to the first and last elements in a doubly-linked list). It is therefore quite straightforward to build stacks and queues whose operations all take only $O(1)$ time. In fact, we can also build a *double-ended queue* (abbreviated *deque* or *dequeue*, pronounced “deck”, and not to be confused with the *dequeue* operation above), in which we can insert and remove elements at both ends.

When adding elements to the end of an array, we need to be somewhat mindful of memory allocation issues. An array is typically stored within a block of memory allocated with some fixed size, and expansion beyond the lower or upper boundary of this block might overwrite memory reserved for other purposes. Fortunately, we can overcome this difficulty by using a *circular array*, shown in Figure 1.9, which logically wraps around within the block. By maintaining pointers to the first and last elements of the array, it is easy both to compute the index of any particular array element (accounting for wrap-around) and to insert or delete elements at the endpoints of the array, all in only $O(1)$ time.

1.6 Complexity Theory

Many algorithms courses focus on problems that we know how to solve efficiently (i.e., in polynomial time), leading to the mistaken impression that there is an efficient algorithm waiting to solve almost every problem one is likely to encounter. Sadly, nothing could be farther from the truth. Many important real-world problems seem to have no efficient algorithmic solution. We have already seen one such problem, the *Hamiltonian path problem*, asking for a path through a graph that visits every node exactly once. In this section, we introduce the concept of NP-hardness and how it can help us characterize and better understand many of the “hard” problems out there.

When given a problem like the Hamiltonian path problem for which we are unable to find an efficient (polynomial time) algorithm, two possibilities exist:

1. The problem is at fault for being genuinely "hard", or
2. The blame rests instead on our own lack of algorithmic design skills.

Our pride would certainly have us prefer option one, which we can establish by showing that our problem belongs to a class of known "hard" problems, with the *NP-hard* problems being by far the most popular. The NP-hard problems (formally defined in a moment) are closely related in a way such that a polynomial-time solution for any one would imply a polynomial-time solution for them all. Since nobody has managed to produce such a polynomial-time algorithm yet for *any* of the thousands of known NP-hard problems, we strongly suspect that these problems are in fact genuinely "hard".

1.6.1 The Complexity Classes P and NP

We now build towards a more precise description of the NP-hard problems. For starters, we focus on problems having "yes" or "no" answers, known as *decision problems*. In the field of complexity theory, we often focus on these problems, since a decision problem has a very simple mathematical representation as a (possibly infinite) set of binary strings, corresponding to binary encodings of all the inputs of the problem for which the answer is "yes". Conveniently, we don't lose too much generality by restricting our focus to decision problems. For example:

- The Hamiltonian path problem is not a decision problem since it asks for a *path* through a graph as output. However, an efficient solution for its *decision variant* (the problem "does this graph contain a Hamiltonian path?") could be used to solve it efficiently, by repeatedly deleting edges from a graph as long as the decision variant tells us that a Hamiltonian path remains after the deletion. At the end, only a Hamiltonian path remains. The Hamiltonian path problem is therefore solvable in polynomial time if and only if the same is true for its decision variant.
- The traveling salesman problem (TSP) is not a decision problem since it asks for the numeric *length* of a minimum tour of n cities (i.e., the minimum length of a Hamiltonian cycle of a graph where edges have varying lengths). However, an efficient algorithm for its decision variant (the problem "does there exist a tour of length at most X?") could be used to solve it efficiently, by binary searching on X. At each step, we use the result of the decision problem to test if a specific guess for X is too high or too low. Since only a polynomial number of invocations of the decision variant are needed[8], a polynomial-time algorithm for the decision variant again implies a polynomial time algorithm for the original problem.

[8] The input for an instance of the TSP, written in binary, contains a binary string describing the integer length of each edge. Since the optimal tour length is a sum of a subset of these numbers, it can be described by a binary string whose length is at most the input size in bits. Binary search for a number described by at most b bits takes $O(b)$ time, so the number of iterations of binary search is polynomially-bounded by the total input size in bits.

We define the *complexity class* P as the set of all decision problems that can be solved in polynomial time. Technically, this means polynomial time on a Turing machine, but a problem solvable in polynomial time on a Turing machine is also solvable in polynomial time on a RAM and vice-versa. We can easily simulate the operation of any Turing machine algorithm on a RAM, and we can also simulate the operation of any RAM algorithm on a Turing machine with only a polynomial blow-up in running time.

Now things get a bit weird. We define the complexity class NP (short for "nondeterministic polynomial") as the set of decision problems solvable in polynomial time by a *nondeterministic* algorithm. Nondeterminism is a purely theoretical concept that strikes many students as bizarre when first introduced. As an algorithm executes, it often reaches branch points where it sequentially iterates through several possible choices. For example, a trivial algorithm to search for a Hamiltonian path might enumerate all $n!$ orderings of the n nodes in a graph, checking each one in sequence to see if the necessary edges are present so as to form a Hamiltonian path. A nondeterministic algorithm has the (completely unrealistic!) ability to follow all of these branches at the same time, rather than in sequence, somewhat like a parallel computer with an unlimited number of processors. Since a nondeterministic algorithm follows every possible execution path simultaneously, it will identify a "yes" instance of a problem as such as long as some execution path confirms this fact in polynomial time (e.g., by discovering a valid Hamiltonian path).

A simpler way to think of NP is that it contains all decision problems for which a "yes" instance can be verified as such in polynomial time, as long as we are given a valid solution[9]. This tells our nondeterministic algorithm which execution path to follow, allowing it to conclude that this is a indeed a "yes" instance in polynomial time without the need for nondeterminism after all. The decision version of the Hamiltonian path problem is in NP since if a graph contains a Hamiltonian path, we can verify this fact in polynomial time if we are told the path (i.e., we can check that the path visits each node exactly once). Since it is usually much easier to verify a solution of a "yes" instance in polynomial time than it is to solve a problem outright in polynomial time, it is generally much easier to determine that a problem belongs to NP than P. NP contains a vast number of problems, including most of the problems in this book, many of them quite important in practice, and many of which we do not know how to solve efficiently.

1.6.2 NP-Hardness and the Satisfiability Problem

In the 1970s, Karp, Cook, and Levin defined the class of *NP-hard* problems. A problem is NP-hard if the existence of a polynomial-time algorithm for that problem would imply the existence of a polynomial-time algorithm for *every* problem in NP. A related term is *NP-complete*, which refers to an NP-hard problem that itself belongs to NP (in casual conversation, you may hear the terms NP-hard and NP-complete used somewhat interchangeably).

If someone manages to produce a polynomial-time algorithm for even a single NP-hard problem, this would imply that all problems in NP have polynomial-time

[9] More generally, we need to be able to verify a "yes" instance as such in polynomial time as long as we are provided with a suitable polynomial-length binary string of "advice".

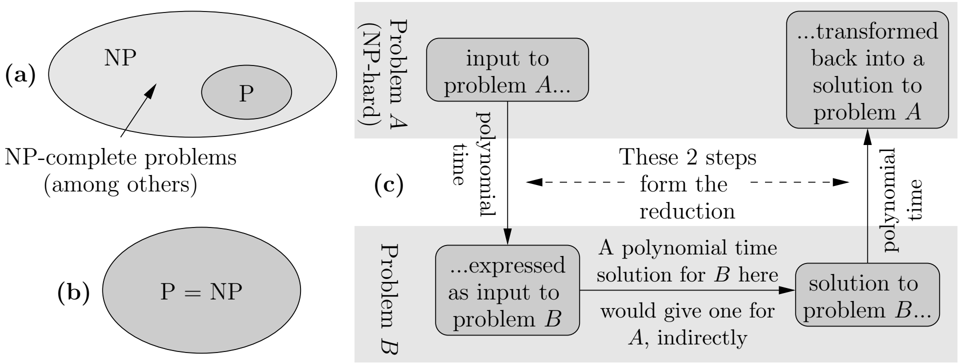

(a)
NP
P
NP-complete problems (among others)
(b)
P = NP
(c)
Problem A (NP-hard)
input to problem A...
polynomial time
...expressed as input to problem B
Problem B
These 2 steps form the reduction
A polynomial time solution for B here would give one for A, indirectly
solution to problem B...
polynomial time
...transformed back into a solution to problem A

FIGURE 1.10: The two possible relationships between P and NP: (a) P $\neq$ NP and (b) P = NP. The vast majority of computer scientists strongly believe that (a) is the correct relationship. The structure of a polynomial-time reduction from problem A to (a polynomially-solvable) problem B to solve problem A indirectly in polynomial time is shown in (c).

solutions and therefore also belong to P, meaning P and NP are the same class. However, since the world's smartest mathematicians and computer scientists have so far failed to solve any NP-hard problem in polynomial time, it is strongly suspected that the NP-complete problems are not in P, as depicted in Figure 1.10(a). However, this has never been successfully proven, and the question of whether or not P = NP remains one of the most challenging (and certainly the most famous) open problem in computer science today.

The first problem to be proven NP-hard was the *satisfiability* problem (abbreviated SAT), which asks whether a Boolean formula[10] such as $(x_1 \vee \overline{x}_2) \wedge x_2$ evaluates to true for some setting of its variables. The famous *Cook-Levin theorem* showed that SAT is NP-hard, since the polynomial-time solvability of any problem in NP can be translated into the question of determining satisfiability of a huge (but still polynomial-size) Boolean formula.

1.6.3 Proving NP-Hardness via Reductions

Fortunately, the Cook-Levin theorem does not need to be re-enacted every time we need to prove other problems are NP-hard. Now that we have one NP-hard problem, we can prove other problems are NP-hard by using polynomial-time *reductions*. The notion of a reduction is an important and fundamental concept in computer science, since it allows us to relate the hardness of one problem with that of another. If we can easily express an instance of problem A in terms of an "equivalent" instance of problem B, then: (i) we automatically get an algorithm for solving A if we already have an algorithm for solving B, and (ii) we know that B must be hard to solve if A is known to be hard to solve.

A prototypical reduction is shown in Figure 1.10(c). To show that problem B is

[10] The symbols $\vee$ and $\wedge$ mean OR and AND, and the notation $\overline{x}_1$ means NOT x_1.

NP-hard, we show that we could indirectly solve some known NP-hard problem A (and hence, every other problem in NP) in polynomial time, if only B were solvable in polynomial time. One could conceive of more complicated reductions than the one shown in Figure 1.10(c); for example, we could solve problem A in polynomial time by making not one, but a polynomial number of invocations of a polynomial-time algorithm for B, and by recombining all of the outputs in polynomial time into a single output for problem A. For most of the problems, however, the simpler style of reduction suffices. The direction of reduction is something new students often get wrong, since instinctively it might seem that we should prove a problem is NP-hard by reducing it to a known NP-hard problem, rather than the reverse (which is the correct direction). When in doubt, remember Figure 1.10(c).

We write $A \leq_p B$ to indicate that A has a polynomial-time reduction to B. It makes sense to use the "$\leq$" symbol since this is saying that A is "no harder" than B with respect to the question of polynomial-time solvability — in particular, that if B can be solved in polynomial time, then so can A. Using this notation, the Cook-Levin theorem says that $A \leq_p SAT$ for every problem A in NP. If we construct a polynomial-time reduction from SAT to some other problem B, then B must also be NP-hard, since $A \leq_p SAT \leq_p B$ for every problem A in NP.

Thousands of problems have been proven NP-hard, so one can find thousands of reduction proofs in the literature. These range from relatively straightforward proofs [Example: $SAT \leq_p 3SAT$] to more involved proofs that require some serious inspiration [Example: $SAT \leq_p CLIQUE$] [Example: $3SAT \leq_p PARTITION$]. Some of the more complicated NP-hardness proofs out there are true works of art!

Problem 3 (Reduction from the Partition Problem). The NP-hard *partition* problem is the following: given n numbers $a_1 \ldots a_n$, is there a partition of these numbers into two subsets whose sums are both equal? This problem is often useful in constructing NP-hardness reduction proofs. As a good practice exercise, give simple reduction proofs of the NP-hardness of the following problems from later in the book: the 0/1 knapsack problem (problem 199(b)), bin packing (problem 201), minimum-makespan scheduling (problem 203), and SONET ring loading (problem 200). [Solution]

Although P and NP are perhaps the best-known, many other interesting complexity classes of problems have been studied in the literature; the rabbit hole goes quite deep, so to speak. Complexity theory is an integral part of the foundation of theoretical computer science, and any student who is interested in pursuing serious study in computer science should certainly feel encouraged to investigate this field in greater depth.

1.7 Different Flavors of Algorithms

Every computational problem we study may be interesting to investigate from the perspective of several different "algorithmic models". For example, what if we are allowed to use randomness in our algorithm? What if we are interested in algorithms that give approximate, rather than exact, solutions? What if the input arrives "online" rather than all at once, and our algorithm must commit to certain decisions before it sees the entire input? What if our memory is not even large enough to store the entire input? What if we have access to a more powerful parallel or distributed computing environment? This section describes some of the more common flavors of algorithms the reader may encounter.

1.7.1 Randomized Algorithms

Randomized algorithms make random choices during their execution. For example, when we perform binary search, we normally compare our target word to the *middle* word in the dictionary, then recurse the first or second half as appropriate. Suppose instead that we compare against a *random* word (this perhaps better describes what happens when you are manually looking up a word in the dictionary, since it is difficult to find the exact middle word). We will see in the next chapter that this *randomized binary search* still runs in $O(\log n)$ time with high probability.

There are several pros and cons of using randomized algorithms. They can often be much simpler, elegant, and efficient than their *deterministic* (non-random) counterparts. Additionally, randomization can thwart a malicious adversary from trying to make our algorithm run slowly: every deterministic algorithm has bad inputs on which it runs the slowest, but many randomized algorithm have no particular bad inputs, since worst-case behavior is now only function of random chance. On the negative side, however, randomized algorithms can be more challenging to analyze (requiring tools from probability theory), they can be harder to debug (since they rarely run the same way twice), and random behavior in an algorithm may not be appropriate for certain mission-critical applications.

There are two main types of randomized algorithms:

- **"Las Vegas" Algorithms.** A Las Vegas algorithm always produces a correct answer. Randomness only manifests itself in the running time, which can vary based on the random choices made by the algorithm. With these algorithms, we typically wish to analyze the expected running time[11] or to prove an even stronger statement like "the running time is $O(n \log n)$ with high probability".
- **"Monte Carlo" Algorithms.** These algorithms have deterministic running times, but they will occasionally output an incorrect solution. This may seem quite objectionable, but we can usually reduce the probability of error to an amount that is negligible in practice (e.g., 10^{-22}, less than the probability that you are struck by lightning, hit by a meteor, and bitten by a shark all in the same year[12]).

A Las Vegas algorithm is always "better" than a Monte Carlo algorithm, since we can convert a Las Vegas algorithm into a Monte Carlo algorithm by terminating with a wrong answer if we exceed some deterministic time threshold. A Monte Carlo algorithm can be converted to a Las Vegas algorithm only if we have a procedure

[11] It is important to note that analysis of the expected running time of a randomized algorithm is different from the "average-case" analysis of a deterministic algorithm. In the first case, randomness occurs in the *algorithm*, and we are still performing a worst-case analysis over all possible inputs. In the second case, randomness occurs in the *input*, in that we assume a probability distribution over all possible inputs reflecting what we expect to see in practice.

[12] A rough estimate assuming independence of these events; it is not recommended to swim in shark-infested waters during an electrical storm at the peak of a meteor shower!

that checks a solution for correctness, in which case we keep repeating until we obtain a correct solution.

In order to use randomness, we modify our computational model to assume access to a stream of random bits. Unfortunately, most computers cannot generate truly random bits, so in practice we typically rely on a *pseudorandom number generator* to produce bits that seem "random enough"; we discuss methods for pseudorandom number generation in greater depth in Section 7.5. Since our mathematical analyses assume perfectly random bits, it is true that they technically may not apply once we start using pseudorandom bits; however, in practice one rarely notices grossly unanticipated behavior due to this discrepancy.

1.7.2 Approximation Algorithms

Important NP-hard (or even harder) problems show up in practice all the time. When faced with one of these problems, we know that it is probably not possible to find an optimal solution in polynomial time in the worst case. What can we do, then, if we need to solve large problem instances? One possibility is to try considering special cases that might have polynomial-time solutions. Another is to simply implement the fastest possible algorithm and hope that it runs in a reasonable amount of time — for many problems the worst-case behavior only results from a very small set of particularly ill-structured inputs that rarely occur in practice. Many researchers study *heuristics*, techniques that tend to improve running time in practice but that often do not have any theoretical performance guarantees (we will study these in detail in Chapter 13).

Another popular approach due to its emphasis on mathematical rigor is to investigate *approximation algorithms*. An approximation algorithm is a polynomial-time algorithm that delivers a solution[13] that is provably close to the correct, or "optimal" solution. The closer we can guarantee our solution will be to the optimal solution, the better our approximation algorithm. The tricky aspect of this, of course, lies in proving that a solution is close to an optimal solution we do not know or have the means to compute.

For an example, recall that the NP-hard satisfiability problem (SAT) involves finding an assignment of values to Boolean variables that satisfies some given Boolean expression. After a bit of simple algebra [Details], any Boolean expression can be written as an equivalent expression in *conjunctive normal form* (CNF), where we have a series of "OR" clauses that are all "ANDed" together:

$$\underbrace{(x_2 \vee \overline{x}_3 \vee x_4)}_{\text{clause 1}} \wedge \underbrace{(\overline{x}_1 \vee x_3)}_{\text{clause 2}} \wedge \underbrace{x_2}_{\text{clause 3}} \wedge \underbrace{(x_1 \vee \overline{x}_2 \vee \overline{x}_4)}_{\text{clause 4}}$$

To satisfy the original expression, we need to find an assignment of values to variables that satisfies all of these clauses. However, let us try instead to find an assignment that satisfies the greatest number of clauses. This is known as the MAX-SAT problem and it is NP-hard since it can be used to solve SAT (observe

[13] We should mention the following to avoid potential confusion: for many people, the word "solution" might imply a "correct" or "optimal" answer to a problem. However, when we discuss algorithms (especially approximation algorithms), we often also use the word "solution" to mean any feasible, but not necessarily optimal answer to a problem.

that we just described a simple reduction from SAT to MAX-SAT). A trivial approximation algorithm for MAX-SAT is to take the better of two solutions, one with all variables set to true, and the other with all variables set to false. Since each clause must evaluate to true in at least one of these solutions, one of the two solutions must satisfy at least half the clauses. Alternatively, if we randomly set each variable by flipping a fair coin, then one can easily show that this satisfies at least half the clauses in expectation. Since the optimal solution can at best satisfy all of the clauses, we say our algorithm is a 1/2-approximation algorithm since it gives a solution whose value is least 1/2 that of the optimal solution. The value 1/2 is called the *performance guarantee* of the algorithm, since it measures relative closeness to the optimal solution value. Some texts would say this algorithm has a performance guarantee of 2, since the solution generated by the algorithm is at most a factor of 2 away from the optimal solution. However, we adopt the popular convention that for maximization problems like MAX-SAT, the performance guarantee will be less than one, whereas for minimization problems it will be more than one. That is, a 1/2-approximation delivers a solution that is at least 1/2 optimal for a maximization problem, and a 2-approximation algorithm delivers a solution no worse than twice optimal for a minimization problem.

Approximation algorithms apply not only to NP-hard problems, but also to problems solvable in polynomial time for which the best known exact algorithms are too slow in practice. They range from very simple techniques like the ones above to much more complicated methods. The best-known approximation algorithm for MAX-SAT has a performance guarantee of roughly 0.77 and requires some very fancy mathematics to achieve this feat!

The Polynomial-Time Approximation Scheme. For some NP-hard problems we can achieve the ultimate approximation result: a *polynomial-time approximation scheme*, or PTAS. A PTAS is a polynomial-time algorithm that delivers a $(1-\varepsilon)$-approximate solution for any constant $\varepsilon > 0$ of our choosing (a $(1+\varepsilon)$-approximate solution, for minimization problems), so it allows us to approximate the optimal solution of a problem to arbitrary relative precision in polynomial time. Unfortunately, most PTAS algorithms have terribly impractical running times like $O(n^{1/\varepsilon})$. Such an algorithm, if used to find a solution within 1% of optimal, would carry a running time of $O(n^{100})$. While such algorithms may be of little use in practice, they are still important in theory because they help us establish the intrinsic hardness of a problem. One does find the occasional PTAS with a running time that is reasonable in practice. For example, in Section 11.4 we discuss a PTAS for the 0/1 knapsack problem that runs in only $O(\frac{n}{\varepsilon}\log n)$ time. If the running time of a PTAS has only a polynomial rather than exponential dependence on $1/\varepsilon$, as in this case, then we call it a *fully polynomial time approximation scheme* (FPTAS).

Inapproximability Results. NP-hard problems that admit an FPTAS are in some sense the "easiest" of the NP-hard problems. However, not all problems admit an FPTAS, or even a PTAS. We can often can prove *inapproximability* results stating that it is NP-hard even to approximate the optimal solution of some problem to within some particular factor. Based on these results, we can partition the class of NP-hard problems into subclasses based on approximability. For example, the class of *strongly NP-hard* problems contains problems that cannot admit an FPTAS unless P = NP, and the class of *MAXSNP-hard* problems contains problems that cannot admit a PTAS unless P = NP. Further details can be found in the endnotes, since we do not wish to inundate the reader with too much new terminology all at once. Inapproximability proofs range from being quite straightforward to those requiring some of the most advanced techniques known to the theoretical computer science community. [Simple example: bin packing is NP-hard to approximate to within better than a factor of 3/2]

Problem 4 (Impossible Inequalities). Suppose you are told that three different numbers a, b, and c should satisfy $a < b$, $b < c$, and $c < a$. It should be clear that it is impossible to assign values to a, b, and c so that all three of these inequalities are satisfied. The best we can do is satisfy two of the three. More generally, suppose we want to assign distinct values to n variables so as to satisfy the greatest possible number out of a set of m strict inequalities, each involving two of the variables. Please suggest a simple 1/2-approximation algorithm for this problem. It is interesting to note that no approximation algorithm with a better performance guarantee is currently known! [Solution]

1.7.3 Online Algorithms

An *online algorithm* does not have the opportunity to see its entire input before it must start making decisions. Online problems appear in many areas of practice, from routing to cache management to load balancing. The classical example of an online problem is the *buy or rent problem*: suppose whenever you go ice skating you have the option of renting a pair of skates for \$5 or buying a pair of skates for \$100. However, you do not know in advance how many times you will end up going skating. It would not make sense to buy skates if you only go skating once or twice, nor would it make sense to rent skates if you plan on making hundreds of visits to the ice rink. If you end up making n trips to the rink, the optimal solution is to rent if $n < 20$ and to buy otherwise. Unfortunately, you don't know n in advance. Let us therefore adopt the following strategy: for the first 19 trips to the rink we will rent, and on the 20th trip we buy. It is not hard to see that regardless of what n turns out to be, we will spend at most twice as much as the optimal solution. We therefore say that our strategy is 2-competitive, illustrating the use of *competitive analysis* as a popular way to characterize the performance of an online algorithm. The *competitive ratio* of an online algorithm is somewhat like the performance guarantee of an approximation algorithm. It specifies the relative distance, in the worst case, from the solution produced by the online algorithm to the optimal solution we could have computed in the "offline" case where we know the entire input in advance.

Problem 5 (CPU Power Management). Suppose a CPU can switch between two levels of power consumption. Any time it needs to process a computationally-intensive task, it needs to be in "high" power mode, where it consumes r_h units of energy per unit time. If there is no computationally-intensive task pending, it can switch down to "low" power mode, where it consumes only $r_l < r_h$ units of energy per unit time. It costs r_{hl} units of energy to switch from high to low power mode, and r_{lh} units of energy to switch from low to high power mode. Since we do not know when computationally-intensive tasks will be arriving in the future, we would like to design a power management strategy that achieves the lowest possible competitive ratio against an optimal strategy that knows the future. What is the lowest possible competitive ratio, and what strategy achieves it? [Solution]

Problem 6 (Dynamic TCP Acknowledgement). The transmission control protocol (TCP) is one of the primary protocols used for transmitting information on the Internet. TCP provides a mechanism for breaking up a communication session between two computers A and B into a stream of packets. Suppose we focus on the packets that A is sending to B. According to TCP, B should occasionally send "acknowledgement" packets back to A specifying which packets B has already received. Since packets tend to be numbered sequentially, B can acknowledge a consecutive range of packets from A using only a single acknowledgement packet, and this is preferable since it avoids the congestion that would be incurred if B sent out an acknowledgement for every single packet it receives. However, if B waits too long before sending out an acknowledgement, A may think that there is excess congestion in the network, and it may slow down its transmission of future packets. A reasonable objective in this case is to have B minimize the sum $a + d$, where a is the number of acknowledgement packets it sends out, and d is the total acknowledgement delay (sum over all incoming packets of the time between arrival of the packet and departure of its corresponding acknowledgement packet). Since B does not know when packets from A will be arriving in the future, it cannot hope to optimally minimize this objective. However, please show a simple acknowledgement policy for B that is 2-competitive. [Solution]

Problem 7 (Searching the Number Line). You are invited to a dinner party at a friend's house, but your friend neglected to mention his address — he only told you the name of his street and a description of his house. Starting from some point on his street, we would like to walk the minimum possible distance before discovering his house. We can think of this problem more abstractly as follows: we start at the origin on a number line and would like to walk the minimum possible total distance to a point whose position x is not known to us. If it were known, we would be able to walk a distance of just $|x|$. Not knowing x, however, consider the strategy of walking back and forth, each time venturing twice as far from the origin on the other side. See if you can obtain upper and lower bounds of 9 on the competitive ratio of this approach. [Solution]

1.7.4 Parallel and Distributed Algorithms

In this book we do not talk much about parallel computation since there is already so much to say about sequential algorithms, and since it is important to be well-grounded in the basics of sequential computation before one attempts to tackle parallel computation. However, parallel computation is rapidly emerging as an area of key importance. At the time of writing of this book, affordable parallelism is becoming available to the masses through multi-core CPUs as well as specialized parallel processors such as graphics processing units (GPUs). In the realm of distributed computing, large clusters of networked computers (sometimes called grids or clouds) are often used in many areas of research and industry for solving a host of important "big data" problems.

Approaches for parallel computation differ slightly depending on whether we have a "fine-grained" parallel computing environment with multiple small processors all wired together in a specific layout and working in tight synchronization, or a "coarse-grained" distributed computing environment composed of multiple types of independent computers that can exchange messages on a common network. In the first case, one tends to focus more on the exact layout we use to connect the processors (e.g., a two-dimensional grid) to obtain the best data flow. In the second case, we may worry more about developing robust algorithms that are tolerant of faults due to network connectivity outages or individual computer failures.

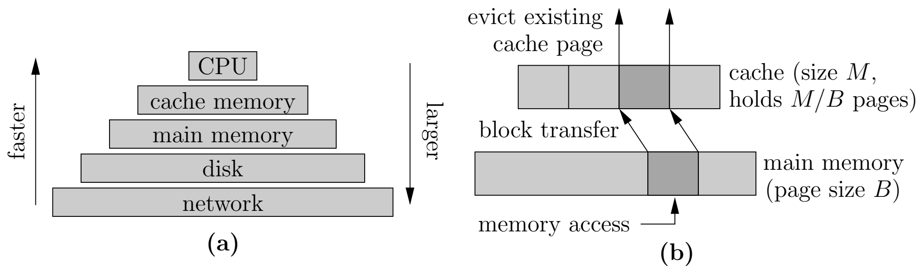

faster
CPU
cache memory
main memory
disk
network
larger
(a)
evict existing cache page
cache (size M, holds M/B pages)
block transfer
main memory (page size B)
memory access
(b)

FIGURE 1.11: Diagrams of (a) the hierarchical memory layout of a typical computer, with each layer containing more memory but being slower to access, and (b) the process of pulling a page of memory into the cache due to an un-cached memory access.

Models of parallel and distributed computation can be quite a bit more complicated than sequential models. The most straightforward parallel generalization of our RAM model is known as a *parallel* RAM or PRAM, and consists of multiple processors operating in parallel with a single shared memory. The tricky issue here of is what happens when two processors decide to access the same memory location at the same time, and there are several variants of the model based on how such collisions are resolved. Other models of parallel computation usually involve a distributed collection of independent processors, each with their own local storage, connected together in some sort of fixed topology like a mesh or a ring. There are many possible variations on these models depending on the topology of the network, whether or not computation is *synchronous* (all processors exchanging messages in a synchronized step-by-step fashion according to a global clock), whether or not processors can fail (or even worse, fail and output garbage), and so on.

Parallel computation is an important area of computer science because it investigates in some sense the true limits of computing devices subject only to the constraints of physics. There are quite a few algorithmic challenges to overcome when designing parallel algorithms, because many problems seem to be inherently non-parallelizable. Just because we have n processors available certainly does not mean we can speed up an algorithm by a factor of n. As an example, consider the simple problem of adding two n-bit binary numbers. We can do this sequentially in $O(n)$ time using the algorithm for addition we all learned in elementary school: add one digit at a time, propagating a “carry” bit along the way if needed. However, if we have all the processors in the world, it is not immediately clear how one would perform this computation any faster! Fortunately, there does exist a clever parallel approach called *carry lookahead addition* that uses $O(n)$ processors to add two n-bit numbers in only $O(\log n)$ time. [Details]

A factor of n is the best speedup we could hope to achieve over a sequential algorithm if we throw n parallel processors at a problem. Therefore, we would need an exponential number of processors to save us from the seemingly unavoidable exponential running times required to solve NP-hard problems. As problem sizes increase, the amount of physical space required for this many processors quickly becomes unreasonable. If our environment requires 2^n processors, then even for moderate values of n we would need more processors than there are atoms in the planet Earth! The only possibility that currently seems to offer some promise for potentially solving NP-hard problems quickly is *quantum computation*, a field that is now in its infancy, but which may offer exciting prospects for highly-efficient computation in the future. Quantum computers operate based on principles of quantum mechanics to achieve what is effectively an exponential amount of parallelism. Although research on quantum computing hardware is progressing, nobody has yet managed to build a general-purpose quantum computer. Of course, this has not stopped researchers from studying the theory of quantum algorithms. The most famous result in this area is probably Shor's algorithm for factoring an n-bit integer in polynomial time on a quantum computer. Factoring is currently not known to be NP-hard, although many researchers suspect that it cannot be solved in polynomial time on a standard RAM (this is a question of crucial importance, since widely-used cryptosystems like RSA rely on the intrinsic hardness of factoring).

1.7.5 Memory-Constrained Algorithms

Many problems in modern computing involve an overwhelming amount of data — more than can fit in memory at one point in time. For example, suppose we need to find the median element from an enormous un-ordered set of n numbers much larger than the memory of our computer, stored externally on read-only magnetic tape. Here, the input is so large that we can only process a small part of it at a time. In Chapter 3, we will see how to solve this problem with a randomized algorithm that makes a small number of passes over the tape. This is sometimes referred to as a *stream* model of computation, where we can only see a small window into the input as it streams by one or more times (we will see several algorithms for this model in Section 7.5). There are many computational situations in which we encounter massive data sets (e.g., the World Wide Web) from which an algorithm cannot hope to see more than a small piece at a time due to limited local storage. However, memory-constrained environments do not necessarily require massive data sets; for example, we may want to design algorithms for small inexpensive "embedded" microprocessors that have very limited memory.

Memory access time is actually the main bottleneck in many data structures and memory-intensive algorithms, since a single memory access can take orders of magnitude more time than a single arithmetic operation within the CPU. This is a notable weakness in our RAM model of computation, which assumes that every memory access takes "one unit of time", just as with any other operation. For the reader who is implementation-oriented, this point is quite important to keep in mind while reading this book. Even if two algorithms share the same asymptotic running time, one can run substantially faster than the other in practice if it is more carefully tuned for fast memory access.

Due to the way computer memories are designed, algorithms like binary search that jump around wildly through memory will run much slower than algorithms that perform the same number of memory accesses in a more highly localized fashion. Figure 1.11(a) depicts the hierarchical layers in the memory layout of a typical computing device. Between the CPU and main memory sits a memory "cache" that is much smaller than main memory but also much faster to access. Memory is divided into "pages" of size B words (typically B is around 1K), and our size-M

cache holds M/B of these. When a page in memory is accessed that does not reside in the cache, the entire page is transferred to the cache in one step[14]. Subsequent accesses to nearby memory locations are therefore much faster, since they reside in the cache. This sort of fast block transfer happens between most layers in memory — disk to memory, memory to cache, etc.

Cache-Oblivious Algorithms. For a memory-intensive algorithm, it is entirely reasonable to use the number of page transfers as the sole measurement of performance. Let $OPT(M, B)$ denote the minimum possible number of page transfers required in the worst case to solve a problem on a two-level memory system (idealized cache plus main memory) with parameters M (total cache size) and B (page size). If an algorithm knows M and B, it can carefully tune its performance to try and achieve this goal. However, M and B are often hard to know, being system-level parameters. A popular model of computation known as the *cache-oblivious* model suggests a nice concept in this situation: we say that an algorithm is cache-oblivious if it performs only $O(OPT(M, B))$ transfers on an idealized cache, despite the fact that the algorithm does not know M or B. Since this holds for every pair of adjacent layers in our memory hierarchy (even if they involve different values for the parameters M and B), a cache-oblivious algorithm therefore incurs at worst a constant-factor slowdown across the entire memory hierarchy compared to what could be achieved by an optimally-tuned algorithm.

Designing cache-oblivious versions of even common algorithms (e.g., sorting algorithms) can be an interesting challenge. For example, even the standard binary search algorithm is not cache-oblivious, since it requires $O(\log(n/B)) = O(\log n - \log B)$ transfers, whereas $OPT(M, B) = O(\log_B n) = O(\log n/ \log B)$ for the problem of searching in set of n elements (if they are organized appropriately in memory). In Section 7.4.3, we will see an elegant "stratified tree" data structure that effectively does provide us with a cache-oblivious version of binary search.

1.8 Closing Remarks

We have seen quite a few definitions and several very deep concepts introduced in this chapter. To the introductory reader: do not worry, these concepts will become clearer over time as we put them to use in the upcoming chapters. In the next chapter, we will acquire some useful mathematical tools, and then in Chapter 3 we will embark on a study of algorithms for sorting, during which we will also spend some time discussing how to formally prove correctness of an algorithm and analyze its running time. The reader is also encouraged to consult the endnotes for further discussion, references, and historical details.

[14] When a new page enters the cache, an old page must be evicted. Common policies for page replacement are to evict the least recently used (LRU) page, or to replace pages on a first-in-first-out (FIFO) basis. The policy for an *ideal* cache would be to replace the page whose next access will be farthest in ahead in the future, although there is no way the cache can know which page this is. Fortunately, one can show that if an algorithm only slows down by a constant factor on an idealized cache when the cache size M is halved, then its running time will only slow down by a constant factor using either LRU or FIFO page replacement as well. We therefore often assume our cache uses an ideal page replacement policy for simplicity. See problem 168 for further detail.