
1 The Role of Algorithms in Computing

What are algorithms? Why is the study of algorithms worthwhile? What is the role of algorithms relative to other technologies used in computers? This chapter will answer these questions.

1.1 Algorithms

Informally, an *algorithm* is any well-defined computational procedure that takes some value, or set of values, as *input* and produces some value, or set of values, as *output* in a finite amount of time. An algorithm is thus a sequence of computational steps that transform the input into the output.

You can also view an algorithm as a tool for solving a well-specified *computational problem*. The statement of the problem specifies in general terms the desired input/output relationship for problem instances, typically of arbitrarily large size. The algorithm describes a specific computational procedure for achieving that input/output relationship for all problem instances.

As an example, suppose that you need to sort a sequence of numbers into monotonically increasing order. This problem arises frequently in practice and provides fertile ground for introducing many standard design techniques and analysis tools. Here is how we formally define the *sorting problem*:

Input: A sequence of n numbers $\langle a_1, a_2, \dots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \dots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

Thus, given the input sequence $\langle 31, 41, 59, 26, 41, 58 \rangle$, a correct sorting algorithm returns as output the sequence $\langle 26, 31, 41, 41, 58, 59 \rangle$. Such an input sequence is

called an *instance* of the sorting problem. In general, an *instance of a problem*¹ consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

Because many programs use it as an intermediate step, sorting is a fundamental operation in computer science. As a result, you have a large number of good sorting algorithms at your disposal. Which algorithm is best for a given application depends on—among other factors—the number of items to be sorted, the extent to which the items are already somewhat sorted, possible restrictions on the item values, the architecture of the computer, and the kind of storage devices to be used: main memory, disks, or even—archaically—tapes.

An algorithm for a computational problem is *correct* if, for every problem instance provided as input, it *halts*—finishes its computing in finite time—and outputs the correct solution to the problem instance. A correct algorithm *solves* the given computational problem. An incorrect algorithm might not halt at all on some input instances, or it might halt with an incorrect answer. Contrary to what you might expect, incorrect algorithms can sometimes be useful, if you can control their error rate. We'll see an example of an algorithm with a controllable error rate in Chapter 31 when we study algorithms for finding large prime numbers. Ordinarily, however, we'll concern ourselves only with correct algorithms.

An algorithm can be specified in English, as a computer program, or even as a hardware design. The only requirement is that the specification must provide a precise description of the computational procedure to be followed.

What kinds of problems are solved by algorithms?

Sorting is by no means the only computational problem for which algorithms have been developed. (You probably suspected as much when you saw the size of this book.) Practical applications of algorithms are ubiquitous and include the following examples:

- The Human Genome Project has made great progress toward the goals of identifying all the roughly 30,000 genes in human DNA, determining the sequences of the roughly 3 billion chemical base pairs that make up human DNA, storing this information in databases, and developing tools for data analysis. Each of these steps requires sophisticated algorithms. Although the solutions to the various problems involved are beyond the scope of this book, many methods to solve these biological problems use ideas presented here, enabling scientists to accomplish tasks while using resources efficiently. Dynamic programming, as

¹ Sometimes, when the problem context is known, problem instances are themselves simply called “problems.”

in Chapter 14, is an important technique for solving several of these biological problems, particularly ones that involve determining similarity between DNA sequences. The savings realized are in time, both human and machine, and in money, as more information can be extracted by laboratory techniques.

- The internet enables people all around the world to quickly access and retrieve large amounts of information. With the aid of clever algorithms, sites on the internet are able to manage and manipulate this large volume of data. Examples of problems that make essential use of algorithms include finding good routes on which the data travels (techniques for solving such problems appear in Chapter 22), and using a search engine to quickly find pages on which particular information resides (related techniques are in Chapters 11 and 32).
- Electronic commerce enables goods and services to be negotiated and exchanged electronically, and it depends on the privacy of personal information such as credit card numbers, passwords, and bank statements. The core technologies used in electronic commerce include public-key cryptography and digital signatures (covered in Chapter 31), which are based on numerical algorithms and number theory.
- Manufacturing and other commercial enterprises often need to allocate scarce resources in the most beneficial way. An oil company might wish to know where to place its wells in order to maximize its expected profit. A political candidate might want to determine where to spend money buying campaign advertising in order to maximize the chances of winning an election. An airline might wish to assign crews to flights in the least expensive way possible, making sure that each flight is covered and that government regulations regarding crew scheduling are met. An internet service provider might wish to determine where to place additional resources in order to serve its customers more effectively. All of these are examples of problems that can be solved by modeling them as linear programs, which Chapter 29 explores.

Although some of the details of these examples are beyond the scope of this book, we do give underlying techniques that apply to these problems and problem areas. We also show how to solve many specific problems, including the following:

- You have a road map on which the distance between each pair of adjacent intersections is marked, and you wish to determine the shortest route from one intersection to another. The number of possible routes can be huge, even if you disallow routes that cross over themselves. How can you choose which of all possible routes is the shortest? You can start by modeling the road map (which is itself a model of the actual roads) as a graph (which we will meet in Part VI and Appendix B). In this graph, you wish to find the shortest path from one vertex to another. Chapter 22 shows how to solve this problem efficiently.

- Given a mechanical design in terms of a library of parts, where each part may include instances of other parts, list the parts in order so that each part appears before any part that uses it. If the design comprises n parts, then there are $n!$ possible orders, where $n!$ denotes the factorial function. Because the factorial function grows faster than even an exponential function, you cannot feasibly generate each possible order and then verify that, within that order, each part appears before the parts using it (unless you have only a few parts). This problem is an instance of topological sorting, and Chapter 20 shows how to solve this problem efficiently.
- A doctor needs to determine whether an image represents a cancerous tumor or a benign one. The doctor has available images of many other tumors, some of which are known to be cancerous and some of which are known to be benign. A cancerous tumor is likely to be more similar to other cancerous tumors than to benign tumors, and a benign tumor is more likely to be similar to other benign tumors. By using a clustering algorithm, as in Chapter 33, the doctor can identify which outcome is more likely.
- You need to compress a large file containing text so that it occupies less space. Many ways to do so are known, including “LZW compression,” which looks for repeating character sequences. Chapter 15 studies a different approach, “Huffman coding,” which encodes characters by bit sequences of various lengths, with characters occurring more frequently encoded by shorter bit sequences.

These lists are far from exhaustive (as you again have probably surmised from this book’s heft), but they exhibit two characteristics common to many interesting algorithmic problems:

1. They have many candidate solutions, the overwhelming majority of which do not solve the problem at hand. Finding one that does, or one that is “best,” without explicitly examining each possible solution, can present quite a challenge.
2. They have practical applications. Of the problems in the above list, finding the shortest path provides the easiest examples. A transportation firm, such as a trucking or railroad company, has a financial interest in finding shortest paths through a road or rail network because taking shorter paths results in lower labor and fuel costs. Or a routing node on the internet might need to find the shortest path through the network in order to route a message quickly. Or a person wishing to drive from New York to Boston might want to find driving directions using a navigation app.

Not every problem solved by algorithms has an easily identified set of candidate solutions. For example, given a set of numerical values representing samples of a signal taken at regular time intervals, the discrete Fourier transform converts

the time domain to the frequency domain. That is, it approximates the signal as a weighted sum of sinusoids, producing the strength of various frequencies which, when summed, approximate the sampled signal. In addition to lying at the heart of signal processing, discrete Fourier transforms have applications in data compression and multiplying large polynomials and integers. Chapter 30 gives an efficient algorithm, the fast Fourier transform (commonly called the FFT), for this problem. The chapter also sketches out the design of a hardware FFT circuit.

Data structures

This book also presents several data structures. A *data structure* is a way to store and organize data in order to facilitate access and modifications. Using the appropriate data structure or structures is an important part of algorithm design. No single data structure works well for all purposes, and so you should know the strengths and limitations of several of them.

Technique

Although you can use this book as a “cookbook” for algorithms, you might someday encounter a problem for which you cannot readily find a published algorithm (many of the exercises and problems in this book, for example). This book will teach you techniques of algorithm design and analysis so that you can develop algorithms on your own, show that they give the correct answer, and analyze their efficiency. Different chapters address different aspects of algorithmic problem solving. Some chapters address specific problems, such as finding medians and order statistics in Chapter 9, computing minimum spanning trees in Chapter 21, and determining a maximum flow in a network in Chapter 24. Other chapters introduce techniques, such as divide-and-conquer in Chapters 2 and 4, dynamic programming in Chapter 14, and amortized analysis in Chapter 16.

Hard problems

Most of this book is about efficient algorithms. Our usual measure of efficiency is speed: how long does an algorithm take to produce its result? There are some problems, however, for which we know of no algorithm that runs in a reasonable amount of time. Chapter 34 studies an interesting subset of these problems, which are known as NP-complete.

Why are NP-complete problems interesting? First, although no efficient algorithm for an NP-complete problem has ever been found, nobody has ever proven that an efficient algorithm for one cannot exist. In other words, no one knows whether efficient algorithms exist for NP-complete problems. Second, the set of

NP-complete problems has the remarkable property that if an efficient algorithm exists for any one of them, then efficient algorithms exist for all of them. This relationship among the NP-complete problems makes the lack of efficient solutions all the more tantalizing. Third, several NP-complete problems are similar, but not identical, to problems for which we do know of efficient algorithms. Computer scientists are intrigued by how a small change to the problem statement can cause a big change to the efficiency of the best known algorithm.

You should know about NP-complete problems because some of them arise surprisingly often in real applications. If you are called upon to produce an efficient algorithm for an NP-complete problem, you are likely to spend a lot of time in a fruitless search. If, instead, you can show that the problem is NP-complete, you can spend your time developing an efficient approximation algorithm, that is, an algorithm that gives a good, but not necessarily the best possible, solution.

As a concrete example, consider a delivery company with a central depot. Each day, it loads up delivery trucks at the depot and sends them around to deliver goods to several addresses. At the end of the day, each truck must end up back at the depot so that it is ready to be loaded for the next day. To reduce costs, the company wants to select an order of delivery stops that yields the lowest overall distance traveled by each truck. This problem is the well-known “traveling-salesperson problem,” and it is NP-complete.² It has no known efficient algorithm. Under certain assumptions, however, we know of efficient algorithms that compute overall distances close to the smallest possible. Chapter 35 discusses such “approximation algorithms.”

Alternative computing models

For many years, we could count on processor clock speeds increasing at a steady rate. Physical limitations present a fundamental roadblock to ever-increasing clock speeds, however: because power density increases superlinearly with clock speed, chips run the risk of melting once their clock speeds become high enough. In order to perform more computations per second, therefore, chips are being designed to contain not just one but several processing “cores.” We can liken these multi-core computers to several sequential computers on a single chip. In other words, they are a type of “parallel computer.” In order to elicit the best performance from multicore computers, we need to design algorithms with parallelism in mind. Chapter 26 presents a model for “task-parallel” algorithms, which take advantage of multiple processing cores. This model has advantages from both theoretical and

² To be precise, only decision problems—those with a “yes/no” answer—can be NP-complete. The decision version of the traveling salesperson problem asks whether there exists an order of stops whose distance totals at most a given amount.

practical standpoints, and many modern parallel-programming platforms embrace something similar to this model of parallelism.

Most of the examples in this book assume that all of the input data are available when an algorithm begins running. Much of the work in algorithm design makes the same assumption. For many important real-world examples, however, the input actually arrives over time, and the algorithm must decide how to proceed without knowing what data will arrive in the future. In a data center, jobs are constantly arriving and departing, and a scheduling algorithm must decide when and where to run a job, without knowing what jobs will be arriving in the future. Traffic must be routed in the internet based on the current state, without knowing about where traffic will arrive in the future. Hospital emergency rooms make triage decisions about which patients to treat first without knowing when other patients will be arriving in the future and what treatments they will need. Algorithms that receive their input over time, rather than having all the input present at the start, are *online algorithms*, which Chapter 27 examines.

Exercises

1.1-1

Describe your own real-world example that requires sorting. Describe one that requires finding the shortest distance between two points.

1.1-2

Other than speed, what other measures of efficiency might you need to consider in a real-world setting?

1.1-3

Select a data structure that you have seen, and discuss its strengths and limitations.

1.1-4

How are the shortest-path and traveling-salesperson problems given above similar? How are they different?

1.1-5

Suggest a real-world problem in which only the best solution will do. Then come up with one in which “approximately” the best solution is good enough.

1.1-6

Describe a real-world problem in which sometimes the entire input is available before you need to solve the problem, but other times the input is not entirely available in advance and arrives over time.

1.2 Algorithms as a technology

If computers were infinitely fast and computer memory were free, would you have any reason to study algorithms? The answer is yes, if for no other reason than that you would still like to be certain that your solution method terminates and does so with the correct answer.

If computers were infinitely fast, any correct method for solving a problem would do. You would probably want your implementation to be within the bounds of good software engineering practice (for example, your implementation should be well designed and documented), but you would most often use whichever method was the easiest to implement.

Of course, computers may be fast, but they are not infinitely fast. Computing time is therefore a bounded resource, which makes it precious. Although the saying goes, “Time is money,” time is even more valuable than money: you can get back money after you spend it, but once time is spent, you can never get it back. Memory may be inexpensive, but it is neither infinite nor free. You should choose algorithms that use the resources of time and space efficiently.

Efficiency

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, Chapter 2 introduces two algorithms for sorting. The first, known as *insertion sort*, takes time roughly equal to c_1n^2 to sort n items, where c_1 is a constant that does not depend on n . That is, it takes time roughly proportional to n^2 . The second, *merge sort*, takes time roughly equal to $c_2n \lg n$, where $\lg n$ stands for $\log_2 n$ and c_2 is another constant that also does not depend on n . Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We’ll see that the constant factors can have far less of an impact on the running time than the dependence on the input size n . Let’s write insertion sort’s running time as $c_1n \cdot n$ and merge sort’s running time as $c_2n \cdot \lg n$. Then we see that where insertion sort has a factor of n in its running time, merge sort has a factor of $\lg n$, which is much smaller. For example, when n is 1000, $\lg n$ is approximately 10, and when n is 1,000,000, $\lg n$ is approximately only 20. Although insertion sort usually runs faster than merge sort for small input sizes, once the input size n becomes large enough, merge sort’s advantage of $\lg n$ versus n more than compensates for the difference in constant factors. No matter how much smaller c_1 is than c_2 , there is always a crossover point beyond which merge sort is faster.

For a concrete example, let us pit a faster computer (computer A) running insertion sort against a slower computer (computer B) running merge sort. They each must sort an array of 10 million numbers. (Although 10 million numbers might seem like a lot, if the numbers are eight-byte integers, then the input occupies about 80 megabytes, which fits in the memory of even an inexpensive laptop computer many times over.) Suppose that computer A executes 10 billion instructions per second (faster than any single sequential computer at the time of this writing) and computer B executes only 10 million instructions per second (much slower than most contemporary computers), so that computer A is 1000 times faster than computer B in raw computing power. To make the difference even more dramatic, suppose that the world's craftiest programmer codes insertion sort in machine language for computer A, and the resulting code requires $2n^2$ instructions to sort n numbers. Suppose further that just an average programmer implements merge sort, using a high-level language with an inefficient compiler, with the resulting code taking $50n \lg n$ instructions. To sort 10 million numbers, computer A takes

$$\frac{2 \cdot (10^7)^2 \text{ instructions}}{10^{10} \text{ instructions/second}} = 20,000 \text{ seconds (more than 5.5 hours) ,}$$

while computer B takes

$$\frac{50 \cdot 10^7 \lg 10^7 \text{ instructions}}{10^7 \text{ instructions/second}} \approx 1163 \text{ seconds (under 20 minutes) .}$$

By using an algorithm whose running time grows more slowly, even with a poor compiler, computer B runs more than 17 times faster than computer A! The advantage of merge sort is even more pronounced when sorting 100 million numbers: where insertion sort takes more than 23 days, merge sort takes under four hours. Although 100 million might seem like a large number, there are more than 100 million web searches every half hour, more than 100 million emails sent every minute, and some of the smallest galaxies (known as ultra-compact dwarf galaxies) contain about 100 million stars. In general, as the problem size increases, so does the relative advantage of merge sort.

Algorithms and other technologies

The example above shows that you should consider algorithms, like computer hardware, as a *technology*. Total system performance depends on choosing efficient algorithms as much as on choosing fast hardware. Just as rapid advances are being made in other computer technologies, they are being made in algorithms as well.

You might wonder whether algorithms are truly that important on contemporary computers in light of other advanced technologies, such as

- advanced computer architectures and fabrication technologies,
- easy-to-use, intuitive, graphical user interfaces (GUIs),
- object-oriented systems,
- integrated web technologies,
- fast networking, both wired and wireless,
- machine learning,
- and mobile devices.

The answer is yes. Although some applications do not explicitly require algorithmic content at the application level (such as some simple, web-based applications), many do. For example, consider a web-based service that determines how to travel from one location to another. Its implementation would rely on fast hardware, a graphical user interface, wide-area networking, and also possibly on object orientation. It would also require algorithms for operations such as finding routes (probably using a shortest-path algorithm), rendering maps, and interpolating addresses.

Moreover, even an application that does not require algorithmic content at the application level relies heavily upon algorithms. Does the application rely on fast hardware? The hardware design used algorithms. Does the application rely on graphical user interfaces? The design of any GUI relies on algorithms. Does the application rely on networking? Routing in networks relies heavily on algorithms. Was the application written in a language other than machine code? Then it was processed by a compiler, interpreter, or assembler, all of which make extensive use of algorithms. Algorithms are at the core of most technologies used in contemporary computers.

Machine learning can be thought of as a method for performing algorithmic tasks without explicitly designing an algorithm, but instead inferring patterns from data and thereby automatically learning a solution. At first glance, machine learning, which automates the process of algorithmic design, may seem to make learning about algorithms obsolete. The opposite is true, however. Machine learning is itself a collection of algorithms, just under a different name. Furthermore, it currently seems that the successes of machine learning are mainly for problems for which we, as humans, do not really understand what the right algorithm is. Prominent examples include computer vision and automatic language translation. For algorithmic problems that humans understand well, such as most of the problems in this book, efficient algorithms designed to solve a specific problem are typically more successful than machine-learning approaches.

Data science is an interdisciplinary field with the goal of extracting knowledge and insights from structured and unstructured data. Data science uses methods

from statistics, computer science, and optimization. The design and analysis of algorithms is fundamental to the field. The core techniques of data science, which overlap significantly with those in machine learning, include many of the algorithms in this book.

Furthermore, with the ever-increasing capacities of computers, we use them to solve larger problems than ever before. As we saw in the above comparison between insertion sort and merge sort, it is at larger problem sizes that the differences in efficiency between algorithms become particularly prominent.

Having a solid base of algorithmic knowledge and technique is one characteristic that defines the truly skilled programmer. With modern computing technology, you can accomplish some tasks without knowing much about algorithms, but with a good background in algorithms, you can do much, much more.

Exercises

1.2-1

Give an example of an application that requires algorithmic content at the application level, and discuss the function of the algorithms involved.

1.2-2

Suppose that for inputs of size n on a particular computer, insertion sort runs in $8n^2$ steps and merge sort runs in $64n \lg n$ steps. For which values of n does insertion sort beat merge sort?

1.2-3

What is the smallest value of n such that an algorithm whose running time is $100n^2$ runs faster than an algorithm whose running time is 2^n on the same machine?

Problems

1-1 *Comparison of running times*

For each function $f(n)$ and time t in the following table, determine the largest size n of a problem that can be solved in time t , assuming that the algorithm to solve the problem takes $f(n)$ microseconds.

	1 second	1 minute	1 hour	1 day	1 month	1 year	1 century
$\lg n$							
\sqrt{n}							
n							
$n \lg n$							
n^2							
n^3							
2^n							
$n!$							

Chapter notes

There are many excellent texts on the general topic of algorithms, including those by Aho, Hopcroft, and Ullman [5, 6], Dasgupta, Papadimitriou, and Vazirani [107], Edmonds [133], Erickson [135], Goodrich and Tamassia [195, 196], Kleinberg and Tardos [257], Knuth [259, 260, 261, 262, 263], Levitin [298], Louridas [305], Mehlhorn and Sanders [325], Mitzenmacher and Upfal [331], Neapolitan [342], Roughgarden [385, 386, 387, 388], Sanders, Mehlhorn, Dietzfelbinger, and Dementiev [393], Sedgewick and Wayne [402], Skiena [414], Soltys-Kulinicz [419], Wilf [455], and Williamson and Shmoys [459]. Some of the more practical aspects of algorithm design are discussed by Bentley [49, 50, 51], Bhargava [54], Kochenderfer and Wheeler [268], and McGeoch [321]. Surveys of the field of algorithms can also be found in books by Atallah and Blanton [27, 28] and Mehta and Sahni [326]. For less technical material, see the books by Christian and Griffiths [92], Cormen [104], Erwig [136], MacCormick [307], and Vöcking et al. [448]. Overviews of the algorithms used in computational biology can be found in books by Jones and Pevzner [240], Elloumi and Zomaya [134], and Marchisio [315].