Exercises

1.1-1
Give a real-world example that requires sorting or a real-world example that requires computing a convex hull.

1.1-2
Other than speed, what other measures of efficiency might one use in a real-world setting?

1.1-3
Select a data structure that you have seen previously, and discuss its strengths and limitations.

1.1-4
How are the shortest-path and traveling-salesman problems given above similar? How are they different?

1.1-5
Come up with a real-world problem in which only the best solution will do. Then come up with one in which a solution that is "approximately" the best is good enough.

1.2 Algorithms as a technology

Suppose computers were infinitely fast and computer memory was 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 demonstrate 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. And memory may be inexpensive, but it is not free. Computing time is therefore a bounded resource, and so is space in memory. You should use these resources wisely, and algorithms that are efficient in terms of time or space will help you do so.

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, in Chapter 2, we will see two algorithms for sorting. The first, known as ***insertion sort***, takes time roughly equal to $c_1 n^2$ 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 $c_2 n \lg n$, 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 shall 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 $c_1 n \cdot n$ and merge sort's running time as $c_2 n \cdot \lg n$. 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 $n = 1000$, $\lg n$ is approximately 10, and when n equals one million, $\lg n$ 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 $\lg n$ vs. n will more than compensate for the difference in constant factors. No matter how much smaller c_1 is than c_2, there will always be 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, 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

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

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 we sort 100 million numbers: where insertion sort takes more than 23 days, merge sort takes under four hours. In general, as the problem size increases, so does the relative advantage of merge sort.

Algorithms and other technologies

The example above shows that we 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, and
- fast networking, both wired and wireless.

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. However, it would also require algorithms for certain 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.

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 separates the truly skilled programmers from the novices. 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.