Main content

Chapter 3

The Efficiency of Algorithms

Chapter Introduction

After studying this chapter, you will be able to:

Describe algorithm attributes and why they are important
Explain the purpose of efficiency analysis and apply it to new algorithms to determine the order of magnitude of their time efficiencies
Describe, illustrate, and use the algorithms from the chapter, including: sequential and binary search, selection sort, data cleanup algorithms, pattern-matching
Explain which orders of magnitude grow faster or slower than others

Describe what a “suspected intractable” problem is, giving one or more examples, and the purpose of approximation algorithms that partially solve them

3.1 Introduction

Finding algorithms to solve problems of interest is an important part of computer science. Any algorithm that is developed to solve a specific problem has, by definition, certain required characteristics (see the formal definition in Chapter 1, Section 1.3.1), but are some algorithms better than others? That is, are there other desirable but nonessential characteristics of algorithms?

Consider the automobile: There are certain features that are part of the “definition” of a car, such as four wheels and an engine. These are the basics. However, when purchasing a car, we almost certainly take into account other things, such as ease of handling, style, and fuel efficiency. This analogy is not as superficial as it seems—the properties that make better algorithms are in fact very similar.

Asmall fontAmedium fontAlarge font

Add Bookmark to this Page

help?

Main content

Chapter Contents

3.2 Attributes of Algorithms

First and foremost, we expect *correctness* from our algorithms. An algorithm intended to solve a problem must, again by formal definition, give a result and then halt. But this is not enough; we also want the result to be a correct solution to the problem. You could consider this an inherent property of the definition of an algorithm (like a car being capable of transporting us where we want to go), but it bears emphasizing. An elegant and efficient algorithm that gives wrong results for the problem at hand is worse than useless. It can lead to mistakes that are enormously expensive or even fatal.

Determining that an algorithm gives correct results might not be as straight-forward as it seems. For one thing, our algorithm might indeed be providing correct results—but to the wrong problem. This can happen when we design an algorithm without a thorough understanding of the real problem we are trying to solve, and it is one of the most common causes of “incorrect” algorithms. Also, once we understand the problem, the algorithm must provide correct results for all possible input values, not just for those values that are the most likely to occur. Do we know what all those correct results are? Probably not, or we would not be writing an algorithm to solve this problem. But there may be a certain standard against which we can check the result for reasonableness, thus giving us a way to determine when a result is obviously incorrect. In some cases, as noted in Chapter 1, the correct result may be an error message saying that there is no correct answer. There may also be an issue of the accuracy of the result we are willing to accept as correct. If the "real" answer is π, for example, then we can only approximate its decimal value. Is 3.14159 close enough to "correct"? Is 3.1416 close enough? What about 3.14? Computer scientists often summarize these two views of correctness by asking, "Are we solving the right problem? Are we solving the problem right?"

If an algorithm to solve a problem exists and we determine, after taking into account all the considerations of the previous paragraph, that it gives correct results, what more can we ask? To many mathematicians, this would be the end of the matter. After all, once a solution has been obtained and shown to be correct, it is no longer of interest (except possibly for use in obtaining solutions to other problems). This is where computer science differs significantly from theoretical disciplines such as pure mathematics and begins to take on an "applied" character more closely related to engineering or applied mathematics. The algorithms developed by computer scientists are not merely of academic interest. They are also intended to be *used*.

Suppose, for example, that a road to the top of a mountain is to be built. An algorithmic solution exists that gives a correct answer for this problem in the sense that a road is produced: Just build the road straight up the mountain. Problem solved. But the highway engineer knows that the road must be usable by real traffic and that this constraint limits the grade of the road. The existence and correctness of the algorithm is not enough; there are practical considerations as well.

The practical considerations for computer science arise because the algorithms developed are executed in the form of computer programs running on real computers to solve problems of interest to real people. Let's consider the "people aspect" first. A computer program is seldom written to be used only once to solve a single instance of a problem. It is written to solve many instances of that problem with many different input values, just as the sequential search algorithm of Chapter 2 would be used many times with different lists of telephone numbers and associated names, and different target *NUMBER* values. Furthermore, the problem itself does not usually "stand still." If the program is successful, people will want to use it for slightly different versions of the problem, which means they will want the program slightly enhanced to do more things. Therefore, after a program is written, it needs to be maintained, both to fix any errors that are uncovered through repeated usage with different input values and to extend the program to meet new requirements. A great deal of time and money are devoted to *program maintenance*. The person who has to modify a program, either to correct errors or to expand its functionality, often is not the person who wrote the original program. To make program maintenance as easy as possible, the algorithm the program uses should be easy to understand. *Ease of understanding*, clarity, "ease of handling"—whatever you want to call it—is a highly desirable characteristic of an algorithm.

On the other hand, there is a certain satisfaction in having an "elegant" solution to a problem. *Elegance* is the algorithmic equivalent of style. The classic example, in mathematical folklore, is the story of the German mathematician Karl Frederick Gauss (1777–1855), who was asked as a school-child to add up the numbers from 1 to 100. The straightforward algorithm of adding 1 + 2 + 3 + 4 + … + 100 by adding one number at a time can be expressed in pseudocode as follows:

This algorithm can be executed to find that the sum has the value 5,050. It is fairly easy to read this pseudocode and understand how the algorithm works. It is also fairly clear that if we want to change this algorithm to one that adds the numbers from 1 to 1,000, we only have to change the loop condition to

- 3.

  While *x* is less than or equal to 1,000, do Steps 4 and 5

  However, Gauss noticed that the numbers from 1 to 100 could be grouped into 50 pairs of the form

  so that the sum equals . This is certainly an elegant and clever solution, but is it easy to understand? If a computer program just said to multiply
  with no further explanation, we might guess how to modify the program to add up the first 1,000 numbers, but would we really grasp what was happening enough to be sure the modification would work? (The Practice Problems at the end of this section discuss this.) Sometimes elegance and ease of understanding work at cross-purposes; the more elegant the solution, the more difficult it may be to understand. Do we win or lose if we have to trade ease of understanding for elegance? Of course, if an algorithm has both characteristics—ease of understanding and elegance—that's a plus.

  Now let's consider the real computers on which programs run. Although these computers can execute instructions very rapidly and have some memory in which to store information, time and space are not unlimited resources. The computer scientist must be conscious of the resources consumed by a given algorithm, and if there is a choice between two (correct) algorithms that perform the same task, the one that uses fewer resources is preferable. The term used to describe an algorithm's careful use of resources is **efficiency**. Efficiency, in addition to *correctness, ease of understanding*, and *elegance*, is an extremely desirable attribute of an algorithm.

  Because of the rapid advances in computer technology, today's computers have much more memory capacity and execute instructions much more quickly than computers of just a few years ago. Efficiency in algorithms might seem to be a moot point; we can just wait for the next generation of technology and it won't matter how much time or space is required. There is some truth to this, but as computer memory capacity and processing speed increase, people find more complex problems to be solved, so the boundaries of the computer's resources continue to be pushed. Furthermore, we will see in this chapter that there are algorithms that consume so many resources that they will never be practical, no matter what advances in computer technology occur.

  How should we measure the time and space consumed by an algorithm to determine whether it is efficient? Space efficiency can be judged by the amount of information the algorithm must store in the computer's memory to do its job, in addition to the initial data on which the algorithm is operating. If it uses only a few extra memory locations while processing the input data, the algorithm is relatively space efficient. If the algorithm requires almost as much additional storage as the input data itself takes up, or even more, then it is relatively space inefficient.

  How can we measure the time efficiency of an algorithm? Consider the sequential search algorithm shown in Figure 2.13, which finds a person's name given his or her telephone number in a reverse telephone directory in which the telephone numbers are not arranged in numerical order. How about running the algorithm on a real computer and timing it to see how many seconds (or maybe what small fraction of a second) it takes to find a name or announce that the phone number is not present? The difficulty with this approach is that there are three factors involved, each of which can affect the answer to such a degree as to make whatever number of seconds we come up with rather meaningless.

1. On which computer will we run the algorithm? Should we use a smartphone, a modest laptop, or a supercomputer capable of doing many trillions of calculations per second?
2. Which reverse telephone book (list of numbers) will we use: one limited to a relatively small geographic area or one that covers all listed numbers in the North American Numbering Plan?
3. Which number will we search for? What if we pick a number that happens to be first in the list? What if it happens to be last in the list?
   Simply timing the running of an algorithm is more likely to reflect machine speed or variations in input data than the efficiency (or lack thereof) of the algorithm itself.

This is not to say that you can't obtain meaningful information by timing an algorithm. Using the same input data (for example, searching for the same number in the same reverse directory) and timing the algorithm on different machines gives a comparison of machine speeds because the task is identical. Using the same machine and the same reverse directory, but searching for different numbers, gives an indication of how the choice of *NUMBER* affects the algorithm's running time on that particular machine. This type of comparative timing is called **benchmarking**. Benchmarks are useful for rating one machine against another and for rating how sensitive a particular algorithm is with respect to variations in input on one particular machine.

However, what we mean by an algorithm's time efficiency is an indication of the amount of "work" required by the algorithm itself. It is a measure of the inherent efficiency of the method, independent of the speed of the machine on which it executes or the specific input data being processed. Is the amount of work an algorithm does the same as the number of instructions it executes? Not all instructions do the same things, so perhaps they should not all be "counted" equally. Some instructions are carrying out work that is fundamental to the way the algorithm operates, whereas other instructions are carrying out peripheral tasks that must be done in support of the fundamental work. To measure time efficiency, we identify the fundamental unit (or units) of work of an algorithm and count how many times the work unit is executed. Later in this chapter, we will see why we can ignore peripheral tasks.

3.3 Measuring Efficiency

The study of the efficiency of algorithms is called the **analysis of algorithms**, and it is an important part of computer science. As a first example of the analysis of an algorithm, we'll look at the sequential search algorithm that we created to solve the reverse telephone lookup problem.

3.3.1 Sequential Search

The pseudocode description of the **sequential search algorithm** from Chapter 2 appears in Figure 3.1, where we have assumed that the list contains n entries instead of 10,000 entries.

Figure 3.1Sequential Search Algorithm

The central unit of work is the comparison of the *NUMBER* being searched for against a single phone number in the list. The essence of the algorithm is the repetition of this task against successive numbers in the list until *NUMBER* is found or the list is exhausted. The comparison takes place at Step 4, within the loop body composed of Steps 4 through 7. Peripheral tasks include setting the initial value of the index *i* and the initial value of *Found*, writing the output, adjusting *Found*, and moving the index forward in the list of numbers. Why are these considered peripheral tasks?

Setting the initial value of the index and the initial value of *Found* requires executing a single instruction, done at Step 2. Writing output requires executing a single instruction, either at Step 5 if *NUMBER* is in the list or at Step 9 if *NUMBER* is not in the list. Note that instruction 5, although it is part of the loop, writes output at most once (if *NUMBER* equals). Similarly, setting *Found* to YES occurs at most once (if *NUMBER* equals) at Step 6. We can ignore the small contribution of these single-instruction executions to the total work done by the algorithm.
Moving the index forward is done once for each comparison, at Step 7. We can get a good idea of the total amount of work the algorithm does by simply counting the number of comparisons and then multiplying by some constant factor to account for the index-moving task. The constant factor could be 2 because we do one index move for each comparison, so we would double the work. It could be less because it is less work to add 1 to i than it is to compare *NUMBER* digit by digit against . As we will see later, the precise value of this constant factor is not very important.
So again, the basic unit of work in this algorithm is the comparison of *NUMBER* against a list element. One comparison is done at each pass through the loop in Steps 4 through 7, so we must ask how many times the loop is executed. Of course, this depends on when, or if, we find *NUMBER* in the list.

The minimum amount of work is done if *NUMBER* is the very first number in the list. This requires only one comparison because *NUMBER* has then been found and the algorithm exits the loop after only one pass. This is the *best case*, requiring the least work. The *worst case*, requiring the maximum amount of work, occurs if *NUMBER* is the very last number in the list or is absent. In either of these situations, *NUMBER* must be compared against all *n*numbers in the list before the loop terminates because *FOUND* gets set to YES (if *NUMBER* is the last number in the list) or because the value of the index *i* exceeds *n* (if *NUMBER* is not in the list).

When *NUMBER* occurs somewhere in the middle of the list, it requires somewhere between 1 (the best case) and *n* (the worst case) comparisons. If we were to run the sequential search algorithm many times with random *NUMBERs* occurring at various places in the list and count the number of comparisons done each time, we would find that the average number of comparisons done is about *n/2*. (The exact average is actually slightly higher than *n/2*; see Exercise 5 at the end of the chapter.) It is not hard to explain why an average of approximately *n/2* comparisons are done (or the loop is executed approximately *n/2* times) when *NUMBER* is in the list. If *NUMBER* occurs halfway down the list, then roughly *n/2*comparisons are required; random *NUMBERs* in the list occur before the halfway point about half the time and after the halfway point about half the time, and these cases of less work and more work balance out.

This means that the average number of comparisons needed to find a *NUMBER* that occurs in a 10-element list is about 5, in a 100-element list about 50, and in a 1,000-element list about 500. With small values of *n*—say, a few hundred or a few thousand numbers—the values of *n/2* (the average case) or *n* (the worst case) are small enough that a computer could execute the algorithm quickly and get the desired answer in a fraction of a second. However, computers are generally used to solve not tiny problems but very large ones. Therefore, we are usually interested in the behavior of an algorithm as the size of a problem (*n*) gets very, very large. In our reverse directory example, the total number of publicly listed telephone numbers exceeds 350,000,000. (Cell phone numbers are usually unlisted.) If the sequential search algorithm were executed on a computer that could do 50,000 comparisons per second, it would require on the average about

or nearly an hour just to do the comparisons necessary to locate a specific number. Including the constant factor for advancing the index, the actual time needed would be even greater. It would require almost 2 hours to do the comparisons required to determine that a number is not in the reverse directory! Sequential search is not sufficiently time efficient for large values of *n* to be useful as a reverse directory lookup algorithm.

Information about the number of comparisons required to perform the sequential search algorithm on a list of *n* numbers is summarized in Figure 3.2. Note that the values for both the worst case and the average case depend on *n*, the number of numbers in the list. The bigger the list, the more work must be done to search it. Few algorithms do the same amount of work on large inputs as on small inputs, simply because most algorithms process the input data, and more data to process means more work. The work an algorithm does can usually be expressed in terms of a formula that depends on the size of the problem input. In the case of searching a list of numbers, the input size is the length of the list.

Figure 3.2

Number of comparisons to find NUMBER in a list of *n* numbers using sequential search

Best Case	Worst Case	Average Case
1	*n*	*n/2*

Let’s say a word about the space efficiency of sequential search. The algorithm stores the list of numbers/names and the target *NUMBER* as part of the input. The only additional memory required is storage for the index value *i* and the *Found* indicator. Two single additional memory locations are insignificant compared with the size of the list, just as executing a single instruction to initialize the value of *i* and *Found* is insignificant beside the repetitive comparison task. Therefore, sequential search uses essentially no more memory storage than the original input requires, so it is very space efficient.

While we have presented sequential search in the context of searching for a particular telephone number from an unsorted list of numbers, the algorithm applies to searching any unordered list of items for one particular “target” item. And again, although it does not seem to be a time-efficient algorithm for a large value of *n*, it is the best approach available to search an unsorted list.

3.3.2 Order of Magnitude—Order *n*

When we analyzed the time efficiency of the sequential search algorithm, we glossed over the contribution of the constant factor for the peripheral work. To see why this constant factor doesn't particularly matter, we need to understand a concept called *order of magnitude*.

The worst-case behavior of the sequential search algorithm on a list of *n* items requires *n*comparisons, and if *c* is a constant factor representing the peripheral work, it requires *cn*total work. Suppose that *c* has the value 2. Then the values of *n* and 2*n* are

n	2*n*
1	2
2	4
3	6
and so on	

These values are shown in Figure 3.3, which illustrates how the value of 2*n*, which is the total work, changes as *n* changes. We can add to this graph to show how the value of *cn*changes as *n* changes, where as well as (see Figure 3.4; these values of c are completely arbitrary). Figure 3.5 presents a different view of the growth rate of *cn* as *n*changes for these three values of *c*.

Figure 3.3

Figure 3.4 for various values of *c*

Figure 3.5 for various values of *c*

Both Figures 3.4 and 3.5 show that the amount of work *cn* increases as *n* increases, but at different rates. The work grows at the same rate as *n* when , at twice the rate of *n* when , and at half the rate of *n* when . However, Figure 3.4 also shows that all of these graphs follow the same basic straight-line shape of *n*. Anything that varies as a constant times *n* (and whose graph follows the basic shape of *n*) is said to be of **order of magnitude *n***, written Θ(*n*) and pronounced "order *n*." We will classify algorithms according to the order of magnitude of their time efficiency. Sequential search is therefore an Θ(*n*) algorithm (an order-*n* algorithm) in both the worst case and the average case.

Flipping Pancakes

A problem posed in the *American Mathematical Monthly* in 1975 by Jacob Goodman concerned a waiter in a café where the cook produced a stack of pancakes of varying sizes. The waiter, on the way to delivering the stack to the customer, attempted to arrange the pancakes in order by size, with the largest on the bottom. The only action available was to stick a spatula into the stack at some point and flip the entire stack above that point. The question is: What is the maximum number of flips ever needed for any stack of *n* pancakes? This number, , is known as the *nth pancake number*.
Here's a fairly simple algorithm to arrange the pancakes. Put the spatula under the largest pancake, as shown in (a) in the figure, and flip. This puts the largest pancake on top [(b) in the figure]. Put the spatula at the bottom of the unordered section (in this case at the bottom) and flip. This puts the largest pancake on the bottom [(c) in the figure], where it belongs. Repeat with the rest of the pancakes. Each pancake therefore requires two flips, which would give a total of 2*n* flips required. But the last two pancakes require at most one flip; if they are already in order, no flips are needed, and if they are out of order, only one flip is needed. So this algorithm requires at most flips in the worst case, which means that . Are there other algorithms that require fewer flips in the worst case?
A faculty member at Harvard University posed this question to his class; several days later, a sophomore from the class came to his office with a better algorithm. This algorithm, which requires at most (5*n* + 5)/3 flips, was published in the journal *Discrete Mathematics* in 1979. The authors were William Gates (the student) and Christos Papadimitriou.

Yes, *that* William Gates!

3.3.3 Selection Sort

The sequential search algorithm solves a very common problem: **searching** a list of items to locate a particular item. Another very common problem is that of **sorting** a list of items into order—either alphabetical or numerical order. The registrar at your institution sorts students in a class by name, a mail-order business sorts its customer list by name, and the IRS sorts tax records by Social Security number. In this section, we'll examine a sorting algorithm and analyze its efficiency.

1. Answer

Suppose we have a list of numbers to sort into ascending order—for example, 5, 7, 2, 8, 3. The result of sorting this list is the new list 2, 3, 5, 7, 8. The **selection sort algorithm** performs this task. The selection sort "grows" a sorted subsection of the list from the back to the front. We can look at "snapshots" of the progress of the algorithm on our sample list, using a vertical line as the marker between the unsorted section at the front of the list and the sorted section at the back of the list in each case. At first the sorted subsection is empty; that is, the entire list is unsorted. This is how the list looks when the algorithm begins.

Later, the sorted subsection of the list has grown from the back so that some of the list members are in the right place.

Finally, the sorted subsection of the list contains the entire list; there are no unsorted numbers, and the algorithm stops.

At any point, then, there is both a sorted and an unsorted section of the list. A pseudocode version of the algorithm is shown in Figure 3.6.

Figure 3.6Selection sort algorithm

Before we illustrate this algorithm at work, take a look at Step 4, which finds the largest number in some list of numbers. We developed an algorithm for this task in Chapter 2(Figure 2.14). A detailed version of the selection sort algorithm would replace Step 4 with the instructions from this existing algorithm. New algorithms can be built up from "parts" consisting of previous algorithms, just as a recipe for pumpkin pie might begin with the instruction, "Prepare crust for a one-crust pie." The recipe for pie crust is a previous algorithm that is now being used as one of the steps in the pumpkin pie algorithm.

Let's follow the selection sort algorithm. Initially, the unsorted section is the entire list, so Step 2 sets the marker at the end of the list.

Step 4 says to select the largest number in the unsorted section—that is, in the entire list. This number is 8. Step 5 says to exchange 8 with the last number in the unsorted section (the whole list). To accomplish this exchange, the algorithm must determine not only that 8 is the largest value but also the location in the list where 8 occurs. The Find Largest algorithm from Chapter 2 provides both these pieces of information. The exchange to be done is

After this exchange and after the marker is moved left as instructed in Step 6, the list looks like

The number 8 is now in its correct position at the end of the list. It becomes the sorted section of the list, and the first four numbers are the unsorted section.

The unsorted section is not empty, so the algorithm repeats Step 4 (find the largest number in the unsorted section); it is 7. Step 5 exchanges 7 with the last number in the unsorted section, which is 3.

After the marker is moved, the result is

The sorted section is now 7, 8 and the unsorted section is 5, 3, 2.

Repeating the loop of Steps 4 through 6 again, the algorithm determines that the largest number in the unsorted section is 5, and exchanges it with 2, the last number in the unsorted section.

After the marker is moved, we get

Now the unsorted section (as far as the algorithm knows) is 2, 3. The largest number here is 3. Exchanging 3 with the last number of the unsorted section (that is, with itself) produces no change in the list ordering. The marker is moved, giving

When the only part of the list that is unsorted is the single number 2, there is also no change in the list ordering produced by carrying out the exchange. The marker is moved, giving

The unsorted section of the list is empty, and the algorithm terminates.

To analyze the amount of work the selection sort algorithm does, we must first decide on the unit of work to count. When we analyzed sequential search, the unit of work that we measured was the comparison between the item being searched for and the items in the list. At first glance, there seem to be no comparisons of any kind going on in the selection sort. Remember, however, that there is a subtask within the selection sort: the task of finding the largest number in a list. The algorithm from Chapter 2 for finding the largest value in a list begins by taking the first number in the list as the largest so far. The largest-so-far value is compared against successive numbers in the list; if a larger value is found, it becomes the largest so far.

When the selection sort algorithm begins, the largest-so-far value, initially the first number, must be compared with all the other numbers in the list. If there are *n* numbers in the list, *n*− 1 comparisons must be done. The next time through the loop, the last number is already in its proper place, so it is never again involved in a comparison. The largest-so-far value, again initially the first number, must be compared with all the other numbers in the unsorted part of the list, which will require *n* − 2 comparisons. The number of comparisons keeps decreasing as the length of the unsorted section of the list gets smaller, until finally only one comparison is needed. The total number of comparisons is

Reviewing our sample problem, we can see that the following comparisons are done:

- To put 8 in place in the list 5, 7, 2, 8, 3 |

  Compare 5 (largest so far) to 7

- 7 becomes largest so far
  Compare 7 (largest so far) to 2

  Compare 7 (largest so far) to 8

- 8 becomes largest so far
  Compare 8 to 3

  8 is the largest

  Total number of comparisons: 4 (which is 5 – 1)

- To put 7 in place in the list 5, 7, 2, 3 | 8

  Compare 5 (largest so far) to 7

- 7 becomes largest so far
  Compare 7 to 2

  Compare 7 to 3

  7 is the largest

  Total number of comparisons: 3 (which is 5 – 2)

- To put 5 in place in the list 5, 3, 2 | 7, 8

  Compare 5 (largest so far) to 3

  Compare 5 to 2

  5 is the largest

  Total number of comparisons: 2 (which is 5 – 3)

- To put 3 in place in the list 2, 3 | 5, 7, 8

  Compare 2 (largest so far) to 3

  3 is the largest

  Total number of comparisons: 1 (which is 5 2 4)

To put 2 in place requires no comparisons; there is only one number in the unsorted section of the list, so it is of course the largest number. It gets exchanged with itself, which produces no effect. The total number of comparisons is .
The sum

turns out to be equal to

(Recall from earlier in this chapter how Gauss computed a similar sum.) For our example with five numbers, this formula says that the total number of comparisons is (using the first version of the formula):

which is the number of comparisons we had counted.

Figure 3.7 uses this same formula

to compute the comparisons required for larger values of *n*. Remember that *n* is the size of the list we are sorting. If the list becomes 10 times longer, the work increases by much more than a factor of 10; it increases by a factor closer to 100, which is .

Figure 3.7

Comparisons required by selection sort

Length *n* of List to Sort		Number of Comparisons Required
10	100	45
100	10,000	4,950
1,000	1,000,000	499,500

The selection sort algorithm not only does comparisons, it also does exchanges. Even if the largest number in the unsorted section of the list is already at the end of the unsorted section, the algorithm exchanges this number with itself. Therefore, the algorithm does *n*exchanges, one for each position in the list to put the correct value in that position. With every exchange, the marker gets moved. However, the work contributed by exchanges and marker moving is so much less than the amount contributed by comparisons that it can be ignored.

We haven't talked here about a best case, a worst case, or an average case for the selection sort. This algorithm does the same amount of work no matter how the numbers are initially arranged. It has no way to recognize, for example, that the list might already be sorted at the outset.

A word about the space efficiency of the selection sort: The original list occupies *n* memory locations, and this is the major space requirement. Some storage is needed for the marker between the unsorted and sorted sections and for keeping track of the largest-so-far value and its location in the list, used in Step 4. Surprisingly, the process of exchanging two values at Step 5 also requires an extra storage location. Here's why. If the two numbers to beexchanged are at position *X* and position *Y* in the list, we might think the following two steps will exchange these values:

1. Copy the current value at position *Y* into position *X*
2. Copy the current value at position *X* into position *Y*
   The problem is that after Step 1, the value at position *X* is the same as that at position *Y*. Step 2 does not put the original value of *X* into position *Y*. In fact, we don't even have the original value of position *X* anymore. In Figure 3.8(a), we see the original *X* and *Y* values. At Figure 3.8(b), after execution of Step 1, the current value of position *Y* has been copied into position *X*, writing over what was there originally. At Figure 3.8(c), after execution of Step 2, the current value at position *X* (which is the original *Y* value) has been copied into position *Y*, but the picture looks the same as Figure 3.8(b).

Figure 3.8An attempt to exchange the values at *X* and *Y*

Here's the correct algorithm, which makes use of one extra temporary storage location that we'll call *T*.

1. Copy the current value at position *X* into location *T*
2. Copy the current value at position *Y* into position *X*
3. Copy the current value at location *T* into position *Y*

Figure 3.9 illustrates that this algorithm does the job. In Figure 3.9(a), the temporary location contains an unknown value. After execution of Step 1 (Figure 3.9(b)), it holds the current value of *X*. When *Y*'s current value is put into *X* at Step 2 (Figure 3.9(c)), *T* still holds the original *X* value. After Step 3 (Figure 3.9(d)), the current value of *T* goes into position *Y*, and the original values of *X* and *Y* have been exchanged. (Step 5 of the selection sort algorithm is thus performed by another algorithm, just as Step 4 is.)

Figure 3.9 Exchanging the values at *X* and *Y*

All in all, the extra storage required for the selection sort, over and above that required to store the original list, is slight. Selection sort is space efficient.

Practice Problems

For each of the following lists, perform a selection sort and show the list after each exchange that has an effect on the list ordering:

4, 8, 2, 6

Answer

12, 3, 6, 8, 2, 5, 7

Answer

D, B, G, F, A, C, E, H

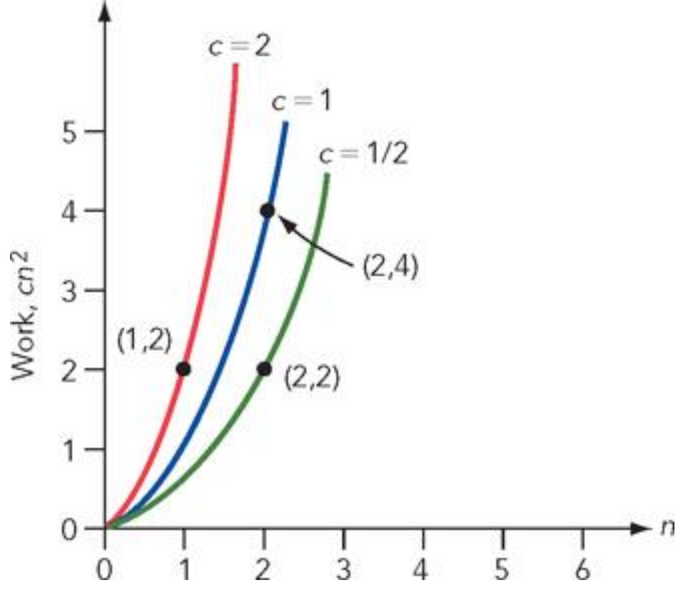
Answer

3, 7, 12, 16, 21

3.3.4 Order of Magnitude—Order n^2

We saw that the number of comparisons done by the selection sort algorithm does not grow at the same rate as the problem size *n*; it grows at approximately the *square* of that rate. An algorithm that does cn^2 work for any constant *c* is **order of magnitude** , or $\Theta(n^2)$. Figure 3.10 shows how cn^2 changes as *n* changes, where c = 1 , 2, and 1/2. The work grows at the same rate as n^2 when c = 1, at twice that rate when c = 2, and at half that rate when c = 1/2. But all three graphs in Figure 3.10 follow the basic shape of n^2, which is different from all of the straight-line graphs that are of $\Theta(n)$. Thus, we have come up with two different "shape classifications": one including all graphs that are $\Theta(n)$ and the other including all graphs that are $\Theta(n^2)$.

Figure 3.10 for various values of *c*



If it is not important to distinguish among the various graphs that make up a given order of magnitude, why is it important to distinguish between the two different orders of magnitude *n* and ? We can find the answer by comparing the two basic shapes *n* and , as is done in Figure 3.11.

Figure 3.11 A comparison of *n* and n^2

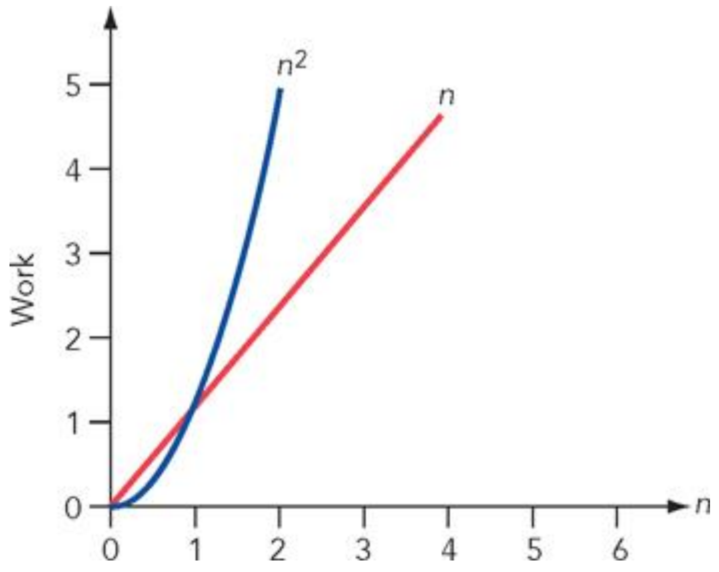
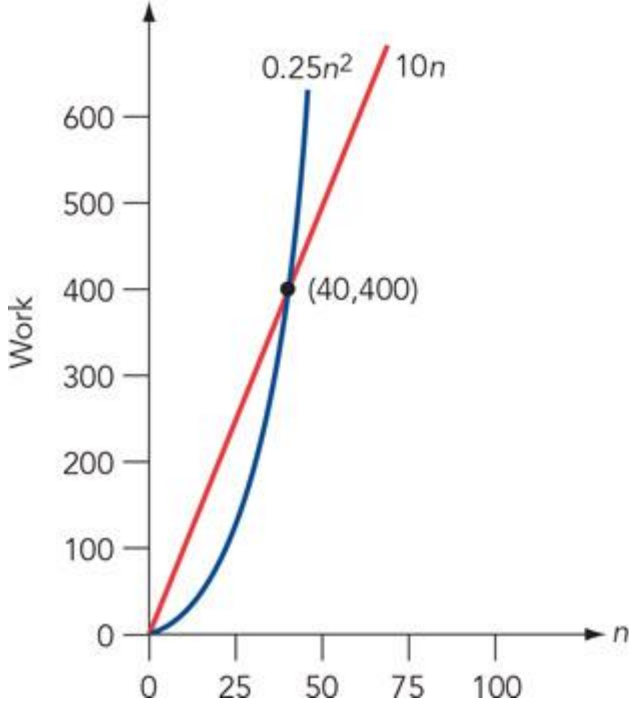


Figure 3.11 illustrates that grows at a much faster rate than *n*. The two curves cross at the point (1,1), and for any value of *n* larger than 1, has a value increasingly greater than *n*. Furthermore, anything that is order of magnitude eventually has larger values than anything that is of order *n*, no matter what the constant factors are. For example, Figure 3.12shows that if we choose a graph that is but has a small constant factor (to keep the values low), say , and a graph that is but has a larger constant factor (to pump the values up), say 10*n*, it is still true that the graph eventually has larger values. (Note that the vertical scale and the horizontal scale are different.)

Figure 3.12 For large enough *n*, $0.25n^2$ has larger values than 10*n*



Selection sort is an algorithm (in all cases) and sequential search is an algorithm (in the worst case), so these two algorithms are different orders of magnitude. Because these algorithms solve two different problems, this is somewhat like comparing apples and oranges—what does it mean? But suppose we have two different algorithms that solve the same problem and count the same units of work but have different orders of magnitude. Suppose that algorithm A does units of work to solve a problem with input size *n*and that algorithm B does 100*n* of the same units of work to solve the same problem. Here algorithm B's factor of 100 is *1 million times larger* than algorithm A's factor of 0.0001. Nonetheless, when the problem gets large enough, the inherent inefficiency of algorithm A causes it to do more work than algorithm B. Figure 3.13 shows that the "crossover" point occurs at a value of 1,000,000 for *n*. At this point, the two algorithms do the same amount of work and therefore take the same amount of time to run. For larger values of *n*, the algorithm A runs increasingly slower than the order-*n* algorithm B. (Input sizes in the millions are not that uncommon; as of April 2017, Facebook had 1.97 billion monthly active users.)

Figure 3.13

A comparison of two extreme algorithms

	Number of Work Units Required	
	Algorithm A	**Algorithm B**
n		**100*n***
1,000	100	100,000
10,000	10,000	1,000,000
100,000	1,000,000	10,000,000
1,000,000	100,000,000	100,000,000

	Number of Work Units Required	
	Algorithm A	Algorithm B
n		100*n*
10,000,000	10,000,000,000	1,000,000,000

As we have seen, if an algorithm and an algorithm exist for the same task, then for large enough *n*, the algorithm does more work and takes longer to execute, regardless of the constant factors for peripheral work. *This is the rationale for ignoring constant factors and concentrating on the basic order of magnitude of algorithms*.
As an analogy, the two shape classifications and may be thought of as two different classes of transportation, the "walking" class and the "driving" class, respectively. The walking class is fundamentally more time consuming than the driving class. Walking can include jogging, running, and leisurely strolling (which correspond to different values for *c*), but compared with any form of driving, these all proceed at roughly the same speed. The driving class can include driving a MINI Cooper and driving a Ferrari (which correspond to different values for *c*), but compared with any form of walking, these proceed at roughly the same speed. In other words, varying c can make modest changes within a class, but changing to a different class is a quantum leap.
Given two algorithms for the same task, we should usually choose the algorithm of the lesser order of magnitude because for large enough *n*, it always "wins out." It is for large values of *n* that we need to be concerned about the time resources being used and, as we noted earlier, it is often for large values of *n* that we are seeking a computerized solution in the first place.

Note, however, that for smaller values of *n*, the size of the constant factor is significant. In Figure 3.12, the 10*n* line stayed above the curve up to the crossover point of because it had a large constant factor relative to the factor for . Varying the factors changes the crossover point. If 10*n* and represented the work of two different algorithms for the same task, and if we are sure that the size of the input is never going to exceed 40, then the algorithm is preferable in terms of time resources used. (To continue the transportation analogy, for traveling short distances—say, to the end of the driveway—walking is faster than driving because of the overhead of getting the car started, and so on. But for longer distances, driving is faster.)

The Tortoise and the Hare

One way to compare performance among different makes of computers is to give the number of arithmetic operations, such as additions or subtractions of real numbers, that each one can do in 1 second. These operations are called *floating-point operations*, and computers are often compared in terms of the number of **flops** (floating-point operations per second) they can crank out.

In March 2016, AMD (Advanced Micro Devices) announced a new graphics card with dual processors and an overall speed of 16 *teraflops* (16 trillion floating-point operations per second; that's 16,000,000,000,000). Such a card would be useful for a PC targeted toward video-game playing, although such a machine could perform general-purpose computing as well. In November 2016, the Sunway TaihuLight at the National SuperComputing Center in Wuxi, China, was declared the world's top-speed supercomputer. It is a parallel processor system with 10,649,600 core processors. It performs at the rate of 93 *petaflops* (93 quadrillion floating-point operations per second; that's 93,000,000,000,000,000). The supercomputer is almost 6,000 times faster than the gaming computer. In some sense this is a meaningless comparison because the two machines are optimized for entirely different purposes. Nonetheless, in terms of raw speed, the stage is set for the race between the tortoise and the hare.

Not fair, you say? We'll see. Let's suppose the gaming machine is assigned to run an algorithm, whereas the supercomputer gets an algorithm for the same task. The work units are floating-point operations, and for simplicity, we'll take the constant factor to be 1 in each case. Here are the timing results:

n	Desktop	Supercomputer
1,000	0.0000000000625 sec	0.0000000000108 sec
100,000,000	0.00000625 sec	0.108 sec
10,000,000,000,000	0.625 sec	1,075,268,817 sec = 34 years

Out of the gate—that is, for relatively small values of n such as 1,000—the supercomputer has the advantage and takes slightly less time. When *n* reaches 100,000,000, however, the supercomputer is falling behind. And for the largest value of *n*, the desktop leaves the supercomputer in the dust. The difference in order of magnitude between the algorithms was enough to slow down the mighty supercomputer and let the desktop pull ahead, chugging along doing its more efficient algorithm. Where would one need to perform 10,000,000,000,000 operations? Complex problems involving weather simulations, biomedical research, and economic modeling might utilize such number-crunching applications. Both China and the United States are pursuing development of *exaflop* supercomputers capable of a billion billion calculations per second.
The point of this little tale is not to say that supercomputers will be readily replaced by desktop gaming computers! It is to demonstrate that the order of magnitude of the algorithm being executed can play a more important role than the raw speed of the computer.

However, making assumptions about the size of the input on which an algorithm will run can be dangerous. A program that runs quickly on small input size may at some point be selected, perhaps because it seems efficient, to solve instances of the problem with large input size, at which point the efficiency may go down the drain! (Software that served Facebook well when it was a startup company with 1,000 active users may not translate satisfactorily to managing 1.97 billion active users.) Part of the job of program documentation is to make clear any assumptions or restrictions about the input size the program was designed to handle.

Comparing algorithm efficiency only makes sense if there is a choice of algorithms for the task at hand. Are there any tasks for which a choice of algorithms exists? Yes; because sorting a list is such a common task, a lot of research has gone into finding good sorting algorithms. Selection sort is one sorting algorithm, but there are many others, including the bubble sort, described in Exercises 11, 12, 13, 14 at the end of this chapter. You might wonder why people don't simply use the one "best" sorting algorithm. It's not that simple. Some algorithms (unlike the selection sort) are sensitive to what the original input looks like. One algorithm might work well if the input is already close to being sorted, whereas another algorithm might work better if the input is random. An algorithm such as selection sort has the advantage of being relatively easy to understand. If the size of the list, *n*, is fairly small, then an easy-to-understand algorithm might be preferable to one that is more efficient but more obscure.

3.4 Analysis of Algorithms

3.4.1 Data Cleanup Algorithms

In this section, we'll look at three different algorithms that solve the same problem—the *data cleanup problem*—and then do an analysis of each. Suppose a survey includes a question about the age of the person filling out the survey, and that some people choose not to answer this question. When data from the survey are entered into the computer, an entry of 0 is used to denote "no response" because a legitimate value for age would have to be a positive number. For example, assume that the age data from 10 people who completed the survey are stored in the computer as the following 10-entry list, where the positions in the list range from 1 (far left) to 10 (far right).

Eventually, the average age of the survey respondents is to be computed. Because the 0 values are not legitimate data—including them in the average would produce too low a value—we want to perform a "data cleanup" and remove them from the list before the average is computed. In our example, the cleaned data could consist of a 10-element list, where the seven legitimate elements are the first seven entries of the list, and some quantity—let's call it *legit*—has the value 7 to indicate that only the first seven entries are legitimate. An alternative acceptable result would be a seven-element list consisting of the seven legitimate data items, in which case there is no need for a *legit* quantity.

The Shuffle-Left Algorithm. Algorithm 1 to solve the data cleanup problem works in the way we might solve this problem using a pencil and paper (and an eraser) to modify the list. We proceed through the list from left to right, pointing with a finger on the left hand to keep our place, and passing over nonzero values. Every time we encounter a 0 value, we squeeze it out of the list by copying each remaining data item in the list one cell to the left. We could use a finger on the right hand to move along the list and point at what to copy next. The value of *legit*, originally set to the length of the list, is reduced by 1 every time a 0 is encountered. (Sounds complicated, but you'll see that it is easy.)

The original configuration is

Because the first cell on the left contains a 0, the value of *legit* is reduced by 1, and all of the items to the right of the 0 must be copied one cell left. After the first such copy (of the 24), the scenario looks like

After the second copy (of the 16), we get

And after the third copy (of the 0), we get

Proceeding in this fashion, we find that after we copy the last item (the 27), the result is

Because the right-hand finger has moved past the end of the list, one entire shuffle-left process has been completed. It required copying nine items. We reset the right-hand finger to start again.

We must again examine position 1 for a 0 value because if the original list contained 0 in position 2, it would have been copied into position 1. If the value is not 0, as is the case here, both the left-hand finger and the right-hand finger move forward.

Moving along, we pass over the 16.

Another cycle of seven copies takes place to squeeze out the 0; the result is

The 36, 42, 23, and 21 are passed over, which results in

and then copying three items to squeeze out the final 0 gives

The left-hand finger is pointing at a nonzero element, so another advance of both fingers gives

At this point, we can stop because the left-hand finger is past the number of legitimate data items . In total, this algorithm (on this list) examined all 10 data items, to see which ones were 0, and copied items.
A pseudocode version of the shuffle-left algorithm to act on a list of n items appears in Figure 3.14. The quantities *left* and *right* correspond to the positions where the left-hand and right-hand fingers point, respectively. You should trace through this algorithm for the preceding example to see that it does what we described.

Figure 3.14The shuffle-left algori thm for data cleanup

To analyze the time efficiency of an algorithm, you begin by identifying the fundamental units of work the algorithm performs. For the data cleanup problem, any algorithm must examine each of the *n* elements in the list to see whether they are 0. This gives a base of at least work units.
The other unit of work in the shuffle-left algorithm is copying numbers. The best case occurs when the list has no 0 values because no copying is required. The worst case

occurs when the list has all 0 values. Because the first element is 0, the remaining elements are copied one cell left and *legit* is reduced from *n* to . After the 0 in position 2 gets copied into position 1, the first element is again 0, which again requires copies and reduces *legit* from to . This repeats until *legit* is reduced to 0, a total of *n* times. Thus there are *n* passes, during each of which copies are done. The algorithm does
copies. If we were to draw a graph of , we would see that for large *n*, the curve follows the shape of . The second term can be disregarded because as *n* increases, the term grows much larger than the *n* term; the term dominates and determines the shape of the curve. The shuffle-left algorithm is thus an algorithm in the worst case.
The shuffle-left algorithm is space efficient because it only requires four memory locations to store the quantities *n, legit, left*, and *right* in addition to the memory required to store the list itself.

The Copy-Over Algorithm. The second algorithm for solving the data cleanup problem also works as we might if we decided to write a new list using a pencil and paper. It scans the list from left to right, copying every legitimate (nonzero) value into a new list that it creates. After this algorithm is finished, the original list still exists, but so does a new list that contains only nonzero values.

For our example, the result would be

Every list entry is examined to see whether it is 0 (as in the shuffle-left algorithm), and every nonzero list entry is copied once (into the new list), so altogether seven copies are done for this example. This is fewer copies than the shuffle-left algorithm requires, but a lot of extra memory space is required because an almost complete second copy of the list is stored. Figure 3.15 shows the pseudocode for this copy-over algorithm.

Figure 3.15 **The copy-over algorithm for data cleanup**

The best case for this algorithm occurs if all elements are 0; no copies are done so the work is just the work to examine each list element and see that it is 0. No extra space is used. The worst case occurs if there are no 0 values in the list. The algorithm copies all *n* nonzero elements into the new list and doubles the space required. Combining the two types of work units, we find that the copy-over algorithm is only in time efficiency even in the worst case because examinations and copies still equal steps.
Comparing the shuffle-left algorithm and the copy-over algorithm, we see that no 0 elements is the best case of the first algorithm and the worst case of the second, whereas all 0 elements is the worst case of the first and the best case of the second. The second algorithm is more time efficient and less space efficient. This choice is called the *time/space tradeoff*—you gain something by giving up something else. Seldom is it possible to improve both dimensions at once, but our next algorithm accomplishes just that.

The Converging-Pointers Algorithm. For the third algorithm, imagine that we move one finger along the list from left to right and another finger from right to left. The left finger slides to the right over nonzero values. Whenever the left finger encounters a

0 item, we reduce the value of *legit* by 1, copy whatever item is at the right finger into the left-finger position, and slide the right finger one cell left. Initially in our example

And because a 0 is encountered at position *left*, the item at position *right* is copied into its place, and both *legit* and *right* are reduced by 1. This results in

The value of *left* increases until the next 0 is reached.

Again, the item at position *right* is copied into position *left*, and *legit* and *right* are reduced by 1.

The item at position left is still 0, so another copy takes place.

From this point, the left finger advances until it meets the right finger, which is pointing to a nonzero element, and the algorithm stops. Once again, each element is examined to see whether it equals 0. For this example, only three copies are needed—fewer even than for algorithm 2, but this algorithm requires no more memory space than algorithm 1. The pseudocode version of the converging-pointers algorithm is given in Figure 3.16.

Figure 3.16**The converging-pointers algorithm for data cleanup**

The best case for this algorithm, as for the shuffle-left algorithm, is a list containing no 0 elements. The worst case, as for the shuffle-left algorithm, is a list of all 0 entries. With such a list, the converging-pointers algorithm repeatedly copies the element at position *right* into the first position, each time reducing the value of *right. Right* goes from *n* to 1, with one copy done at each step, resulting in copies. This algorithm is in the worst case. Like the shuffle-left algorithm, it is space efficient. It is possible in this case to beat the time/space trade-off, in part because the data cleanup problem requires no particular ordering of the nonzero elements in the “clean” list; the converging-pointers algorithm moves these elements out of their original order.
It is hard to define what an “average” case is for any of these algorithms; the amount of work done depends on how many 0 values there are in the list and perhaps on where in the list they occur. If we assume, however, that the number of 0 values is some percentage of *n*and that these values are scattered throughout the list, then it can be shown that the shuffle-left algorithm will still do work, whereas the converging-pointers algorithm will do . Figure 3.17 summarizes our analysis, although it doesn’t reflect the three or four extra memory cells needed to store other quantities used in the algorithms, such as *legit, left*, and *right*.

Figure 3.17

Analysis of three data cleanup algorithms

	1. Shuffle-left		**2. Copy-over**		**3. Converging-pointers**	
	Time	Space	Time	Space	Time	Space
Best case		*n*		*n*		*n*
Worst case		*n*		*2n*		*n*
Average case		*n*				*n*

Let's emphasize again the difference between an algorithm that is in the amount of work it does and one that is . In an algorithm, the work is proportional to *n*. Hence if you double *n*, you double the amount of work; if you multiply *n* by 10, you multiply the work by 10. But in an algorithm, the work is proportional to the *square* of *n*. Hence if you double *n*, you multiply the amount of work by 4; if you multiply *n* by 10, you multiply the work by 100.

This is probably a good place to explain why the distinction between *n* and 2*n* is important when we are talking about space, but we simply classify *n* and 8000*n* as when we are talking about units of work. Units of work translate into time when the algorithm is executed, and time is a much more elastic resource than space. Whereas we want an algorithm to run in the shortest possible time, in many cases there is no fixed limit to the amount of time that can be expended. There is, however, always a fixed upper bound on the amount of memory that the computer has available to use while executing an algorithm, so we track space consumption more closely.

3.4.2 Binary Search

The sequential search algorithm searches a list of *n* items for a particular item; it is an algorithm. Another algorithm, the **binary search algorithm**, is more efficient but it works only when the search list is already sorted.

To understand how binary search operates, let us go back to the problem of searching for *NUMBER* in a reverse telephone directory, but now we assume that the directory is sorted in increasing numerical order by phone number. As we noted in Chapter 2, you would not search for (555) 123-4567 in such a directory by starting with the first number and proceeding sequentially through the list. Instead you would look for this number near the middle of the list, and if you didn't find it immediately, you would continue your search on the front half or the back half of the list.

This is exactly how the binary search algorithm works on a sorted list. It first looks for *NUMBER* at roughly the halfway point in the list. If the number there equals *NUMBER*, the search is over. If *NUMBER* comes numerically before the number at the halfway point, then the search is narrowed to the front half of the list, and the process begins again on this smaller list. If *NUMBER* comes numerically after the number at the halfway point, then the search is narrowed to the back half of the list, and the process begins again on *this* smaller list. The algorithm halts when *NUMBER* is found or when the sublist becomes empty.

Practice Problems

In the data cleanup problem, suppose the original data list is

2	0	4	1

Write the data list after completion of algorithm 1, the shuffle-left algorithm.

Answer

Write the two data lists after completion of algorithm 2, the copy-over algorithm.

Answer

Write the data list after completion of algorithm 3, the converging-pointers algorithm.

Answer

Make up a data list such that Step 11 of the converging-pointers algorithm (Figure 3.16) is needed.

Answer

For example,

Figure 3.18 gives a pseudocode version of the binary search algorithm on a sorted *n*-element list. Here *beginning* and *end* mark the beginning and end of the section of the list under consideration. Initially the whole list is considered, so at first *beginning* is 1 and *end* is *n*. If *NUMBER* is not found at the midpoint *m* of the current section of the list, then setting *end*equal to one less than the midpoint (Step 9) means that at the next pass through the loop, the front half of the current section is searched.
Setting *beginning* equal to one more than the midpoint (Step 10) means that at the next pass through the loop, the back half of the current section is searched. Thus, as the algorithm proceeds, the *beginning* marker can move toward the back of the list, and the *end* marker can move toward the front of the list. If the *beginning* marker and the end marker cross over—that is, *end* becomes less than *beginning*—then the current section of the list is empty and the search terminates. Of course it also terminates if *NUMBER* is found.

Figure 3.18Binary search algorithm (list must be sorted)

Let's do an example, using seven telephone numbers sorted in increasing order. The following list shows not only the numbers in the list but also their locations in the list. For clarity, we have removed the punctuation from the numbers and written them simply as 10-digit numerals.

Suppose we search this list for the number 3597211488. We set *beginning* to 1 and *end* to 7; the midpoint between 1 and 7 is 4. We compare 3597211488 with the position 4 number, 5656170224. The number 3597211488 is less than 5656170224, so the algorithm sets *end* to (Step 9) to continue the search on the front half of the list, The midpoint between and is 2, so we compare 3597211488 with the position 2 number, 3563278900. The number 3597211488 is greater than 3563278900, so the algorithm sets *beginning* to (Step 10) in order to continue the search on the back half of this list, namely

At the next pass through the loop, the midpoint between and is 3, so we compare 3597211488 with the position 3 number, 3597211488. We have found the target number; the corresponding name can be printed and *Found* changed to YES. The loop terminates, and then the algorithm terminates.
Now suppose we search this same list for the number 7213350096. As before, the first midpoint is 4, so 7213350096 is compared with 5656170224. Because 7213350096 > 5656170224, the search continues with , on the back half:

The midpoint is 6, so 7213350096 is compared with 7719215281. Because 7213350096 < 7719215281, the search continues with , on the front half:

The midpoint is 5, so 7213350096 is compared with 6485551285. Because 7213350096 > 6485551285, *beginning* is set to 6 to continue the search on the "back half" of this list. The algorithm checks the condition at Step 4 to see whether to repeat the loop again and finds that *end* is less than *beginning* (,). The loop is abandoned, and the algorithm moves on to Step 11 and indicates that our target number is not in the list.
It is easier to see how the binary search algorithm operates if we list the locations of the numbers checked in a treelike structure. The tree in Figure 3.19 shows the possible search locations in a seven-element list. The search starts at the top of the tree, at location 4, the middle of the original list. If the number at location 4 is *NUMBER*, the search halts. If *NUMBER* comes after the number at location 4, the right branch is taken and the next location searched is location 6. If *NUMBER* comes before the number at location 4, the left branch is taken and the next location searched is location 2.
If *NUMBER* is not found at location 2, the next location searched is either 1 or 3. Similarly, if *NUMBER* is not found at location 6, the next location searched is either 5 or 7.

In Figure 3.18, the binary search algorithm, we assume in Step 5 that there is a middle position between *beginning* and *end*. This happens only when there is an odd number of elements in the list. Let us agree to define the "middle" of an even number of entries as the end of the first half of the list. With eight elements, for example, the midpoint position is location 4.

With this understanding, the binary search algorithm can be used on sorted lists of any size. And of course, the list items need not be numbers; they could also be text items that are sorted alphabetically.

Like the sequential search algorithm, the binary search algorithm relies on comparisons, so to analyze the algorithm, we count the number of comparisons as an indication of the work done. The best case, as in sequential search, requires only one comparison—the target is located on the first try. The worst case, as in sequential search, occurs when the target is not in the list. However, we learn this much sooner in binary search than in sequential search. In our list of seven telephone numbers, only three comparisons are needed to determine that 7213350096 is not in the list. The number of comparisons needed is the number of circles in some branch from the top to the bottom of the tree in Figure 3.19. These circles represent searches at the midpoints of the whole list, half the list, one quarter of the list, and so on. This process continues as long as the sublists can be cut in half.

Figure 3.19 **Binary search tree for a seven-element list**

Let's do a minor mathematical digression here. The number of times a number *n* can be cut in half and not go below 1 is called the *logarithm of n to the base 2*, which is abbreviated lg *n*(also written in some texts as *n*). For example, if *n* is 16, then we can do four such divisions by 2:
so lg . This is another way of saying that . In general,

Figure 3.20 shows a few values of *n* and lg *n*. From these, we can see that as *n* doubles, lg *n*increases by only 1, so lg *n* grows much more slowly than *n*. Figure 3.21 shows the two basic shapes of *n* and lg *n* and again conveys that lg *n* grows much more slowly than *n*.

Figure 3.20 **Values for *n* and lg *n***

Figure 3.21 **A comparison of *n* and lg *n***

Remember the analogy we suggested earlier about the difference in time consumed between algorithms, equivalent to various modes of walking, and algorithms, equivalent to various modes of driving? We carry that analogy further by saying that algorithms of **order of magnitude lg n**, , are like various modes of flying. Changing the coefficients of lg *n* can mean that we go from a Cessna 172 Skyhawk (top speed about 185 miles per hour) to an F-35 Lightning II fighter plane (top speed about 1200 miles per hour) but flying, in any form, is still a fundamentally different—and much faster—mode of travel than walking or driving.
Suppose we are doing a binary search on *n* items. In the worst case, as we have seen, the number of comparisons is related to the number of times the list of length *n* can be halved. Binary search does comparisons in the worst case (see Exercise 31 at the end of the chapter for an exact formula for the worst case). As a matter of fact, it also does comparisons in the average case (although the exact value is a smaller number than in the worst case). This is because most of the items in the list occur at or near the bottom of the tree, where the maximum amount of work must be done. As Figure 3.19 shows, relatively few locations, where the target might be found and the algorithm terminate sooner, are higher in the tree.
The two search algorithms, binary search and sequential search, differ in the order of magnitude of the work they do. Binary search is an algorithm, whereas sequential search is an algorithm, in both the worst case and the average case. To compare the binary search algorithm with the sequential search algorithm, suppose there are 100

elements in the list. In the worst case, sequential search requires 100 comparisons, and binary search requires 7 . In the average case, sequential search requires about 50 comparisons, and binary search 6 or 7 (still much less work). The improvement in binary search becomes even more apparent as the search list gets longer. For example, if , then in the worst case, sequential search requires 100,000 comparisons, whereas binary search requires 17 (because and). If we wrote two programs, one using sequential search and one using binary search, and ran them on a computer that can do 1,000 comparisons per second, then to determine that an item is not in the list (the worst case) the sequential search program would use

or 1.67 minutes, just to do the necessary comparisons, disregarding the constant factor for advancing the index. The binary search program would use

to do the comparisons, disregarding a constant factor for updating the values of *beginning*and *end*. This is quite a difference.

Suppose our two programs are used with the 350,000,000 numbers we assume are in the reverse telephone directory. On the average, the sequential search program needs about

(over 2 days!) just to do the comparisons to find a number in the list, whereas the binary search program needs (because and) about

This is an even more impressive difference. Furthermore, it's a difference due to the inherent inefficiency of an algorithm compared with an algorithm; the difference can be mitigated but not eliminated by using a faster computer. If our computer does 50,000 comparisons per second, then the average times become about

or nearly an hour for sequential search and about

for binary search. The sequential search alternative is simply not acceptable. That is why analyzing algorithms and choosing the best one can be so important. We also see, as we noted in Chapter 2, that the way the problem data are organized can greatly affect the best choice of algorithm.

The binary search algorithm works only on a list that has already been sorted. An unsorted list could be sorted before using a binary search, but sorting also takes a lot of work, as we have seen. If a list is to be searched only a few times for a few particular items, then it is more efficient to do sequential search on the unsorted list (a few tasks). But if the list is to be searched repeatedly, it is more efficient to sort it and then use binary search: one task and many tasks, as opposed to many tasks.
As to space efficiency, binary search, like sequential search, requires only a small amount of additional storage to keep track of beginning, end, and midpoint positions in the list. Thus, it is space efficient; in this case, we did not have to sacrifice space

efficiency to gain time efficiency. But we did have to sacrifice generality—binary search works only on a sorted list whereas sequential search works on any list.

3.4.3 Pattern Matching

The pattern-matching algorithm in Chapter 2 involves finding all occurrences of a pattern of the form within text of the form . Recall that the algorithm simply does a "forward march" through the text, at each position attempting to match each pattern character against the text characters. The process stops only after text position , when the remaining text is not as long as the pattern so that there could not possibly be a match. This algorithm is interesting to analyze because it involves two measures of input size: *n*, the length of the text string, and *m*, the length of the pattern string. The unit of work is comparison of a pattern character with a text character.
Surprisingly, both the best case and the worst case of this algorithm can occur when the pattern is not in the text at all. The difference hinges on exactly *how* the pattern fails to be in the text. The best case occurs if the first character of the pattern is nowhere in the text, as in

In this case, comparisons are required, trying (unsuccessfully) to match with in turn. Each comparison fails, and the algorithm slides the pattern forward to try again at the next position in the text.
The maximum amount of work is done if the pattern *almost* occurs everywhere in the text. Consider, for example, the following case:

Starting with , the first text character, the match with the first pattern character is successful. The match with the second text character and the second pattern character is also successful. Indeed characters of the pattern match with the text before the *m*th comparison proves a failure. The process starts over from the second text character, . Once again, *m* comparisons are required to find a mismatch. Altogether, *m* comparisons are required for each of the starting positions in the text.
Another version of the worst case occurs when the pattern is found at each location in the text, as in

This results in the same comparisons as are done for the other worst case, the only difference being that the comparison of the last pattern character is successful.

Unlike our simple examples, pattern matching usually involves a pattern length that is short compared with the text length, that is, when *m* is much less than *n*. In such cases, is essentially *n*. The pattern-matching algorithm is therefore $\Theta(n)$ in the best case and $\Theta(m \times n)$ in the worst case.
It requires somewhat improbable situations to create the worst cases we have described. In general, the forward-march algorithm performs quite well on text and patterns consisting of ordinary words. Other pattern-matching algorithms are conceptually more complex but require less work in the worst case.

help?

Main content

Chapter Contents

Summary

Figure 3.22 shows an order-of-magnitude summary of the time efficiency for the algorithms we have analyzed.Figure 3.22

Order-of-magnitude time efficiency summary

Problem	Unit of Work	Algorithm	Best Case	Worst Case	Average Case		
Searching	Comparisons	Sequential search	1	$\Theta(n)$	$\Theta(n)$		
				Binary search	1	$\Theta(\lg n)$	$\Theta(\lg n)$
Sorting Comparisons	and exchanges	Selection sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$		
Data cleanup	Examinations and copies	Shuffle-left	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$		
				Copy-over	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
				Converging-pointers	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Pattern matching	Character comparisons	Forward march	$\Theta(n)$	$\Theta(m \times n)$			

3.5 When Things Get Out of Hand

We have so far found examples of algorithms that are $\Theta(\lg n)$, $\Theta(n)$, and in time efficiency. Order of magnitude determines how quickly the values grow as n increases. An algorithm of order $\lg n$ does less work as n increases than does an algorithm of order n, which in turn does less work than one of order . The work done by any of these algorithms is no worse than a constant multiple of , which is a polynomial in n. Therefore, these algorithms are **polynomially bounded** in the amount of work they do as n increases.

Some algorithms must do work that is not polynomially bounded. Consider four cities, A, B, C, and D, that are connected as shown in Figure 3.23, and ask the following question: Is it possible to start at city A, go through every other city exactly once, and end up back at A? Of course, we as humans can immediately see in this small problem that the answer is “yes” and that there are two such paths: A-B-D-C-A and A-C-D-B-A. However, an algorithm doesn’t get to “see” the entire picture at once, as we can; it has available to it only isolated facts such as “A is connected to B and to C,” “B is connected to A and to D,” and so on. If the number of *nodes* and connecting *edges* is large, humans also might not “see” the solution immediately. A collection of nodes and connecting edges is called a *graph*. A path through a graph that begins and ends at the same node and goes through all other nodes exactly once is called a *Hamiltonian circuit*, named for the Irish mathematician William Rowan Hamilton (1805–1865). If there are *n* nodes in the graph, then a Hamiltonian circuit, if it exists, must have exactly *n* links. In the case of the four cities, for instance, if the path must go through exactly A, B, C, D, and A (in some order), then there are five nodes on the path (counting A twice) and four links.

Figure 3.23Four connected cities

Our problem is to determine whether an arbitrary graph has a Hamiltonian circuit. An algorithm to solve this problem examines all possible paths through the graph that are the appropriate length to see whether any of them are Hamiltonian circuits. The algorithm can trace all paths by beginning at the starting node and choosing at each node where to go next. Without going into the details of such an algorithm, let’s represent the possible paths with four links in the graph of Figure 3.23. Again, we use a tree structure. In Figure 3.24, A is the tree “root,” and at each node in the tree, the nodes directly below it are the choices for the next node. Thus, any time B appears in the tree, it has the two nodes A and D below it because edges exist from B to A and from B to D. The “branches” of the tree are all the possible paths from A with four links. Once the tree has been built, an examination of the paths shows that only the two dark paths in the figure represent Hamiltonian circuits.

Figure 3.24Hamiltonian circuits among all paths from A in Figure 3.23 with four links

The number of paths that must be examined is the number of nodes at the bottom level of the tree. There is one node at the top of the tree; we’ll call the top of the tree level 0. The number of nodes is multiplied by 2 for each level down in the tree. At level 1, there are 2 nodes; at level 2, there are nodes; at level 3, there are nodes; and at level 4, the bottom of the tree, there are nodes.

Suppose we are looking for a Hamiltonian circuit in a graph with *n* nodes and two choices at each node. The bottom of the corresponding tree is at level *n*, and there are paths to examine. If we take the examination of a single path as a unit of work, then this algorithm must do units of work. This is more work than any polynomial in *n*. An algorithm is called an **exponential algorithm**. Hence the trial-and-error approach to solving this Hamiltonian circuit problem is an exponential algorithm. (We could improve on this algorithm by letting it stop tracing a path whenever a repeated node different from the starting node is encountered, but it is still exponential. If there are more than two choices at a node, the amount of work is even greater.)

Figure 3.25 shows the four curves lg *n*, *n*, , and . The rapid growth of is not really apparent here, however, because that curve is off the scale for values of *n* above 5. Figure 3.26 compares these four curves for values of *n* that are still small, but even so, is already far outdistancing the other values.

Figure 3.25Comparison of lg *n*, *n*, , and

Figure 3.26 Comparisons of lg *n*, *n*, , and for larger values of *n*

To appreciate fully why the order of magnitude of an algorithm is important, let's again imagine that we are running various algorithms as programs on a computer that can perform a single operation (unit of work) in 0.0001 second. Figure 3.27 shows the amount of time it takes for algorithms of Θ(lg *n*), Θ(*n*), , and to complete their work for various values of *n*.

Figure 3.27

A comparison of four orders of magnitude

	n			
Order	**10**	**50**	**100**	**1,000**
lg *n*	0.0003 sec	0.0006 sec	0.0007 sec	0.001 sec
n	0.001 sec	0.005 sec	0.01 sec	0.1 sec
n^2	0.01 sec	0.25 sec	1 sec	1.67 min
2^n	0.1024 sec	3,570 years	4 × 1016 centuries	*Too big to compute!!*

The expression grows unbelievably fast. An algorithm of can take so long to solve even a small problem that it is of no practical value. Even if we greatly increase the speed of the computer, the results are much the same. We now see more than ever why we added *efficiency* as a desirable feature for an algorithm and why future advances in computer technology won't change this. No matter how fast computers get, they will not be able to solve a problem of size using an algorithm of in any reasonable period of time.

The algorithm we have described here for testing an arbitrary graph for Hamiltonian circuits is an example of a *brute force algorithm*—one that beats the problem into submission by trying all possibilities. In Chapter 1, we described a brute force algorithm for winning a chess game; it consisted of looking at all possible game scenarios from any given point on and then picking a winning one. This is also an exponential algorithm. Some very practical problems have exponential solution algorithms. For example, an email message that you send over the Internet is routed along the shortest possible path through intermediate computers from your mail server computer to the destination mail server computer. An exponential algorithm to solve this problem would examine all possible paths to the destination and then use the shortest one. As you can imagine, the Internet uses a better (more efficient) algorithm than this one!

Are there problems for which no polynomially bounded algorithm exists? Such problems are called **intractable**; they are solvable, but the solution algorithms all require so much work as to be virtually useless. The Hamiltonian circuit problem is suspected to be such a problem, but we don't really know for sure! No one has yet found a solution algorithm that works in polynomial time, but neither has anyone proved that such an algorithm does not exist. This is a problem of great interest in computer science. A surprising number of problems fall into this "suspected intractable" category. Here's another one, called the *bin-packing problem*: Given an unlimited number of bins of volume *X* units and given *n* objects, all of volume between 0 and *X*, find the minimum number of bins needed to store the *n* objects. A brute force algorithm would try all possibilities, which again is not a polynomial algorithm. Any manufacturer who ships sets of various items in standard-sized cartons or anyone who wants to store variable-length video clips on a set of DVDs in the most efficient way would be interested in a polynomial algorithm that solves this minimization problem.

Problems for which no known polynomial solution algorithm exists are sometimes approached via **approximation algorithms**. These algorithms don't give the exact answer to the problem, but they provide a close approximation to a solution (see Exercise 2 of Chapter 1). For example, an approximation algorithm to solve the bin-packing problem is to take the objects in order, put the first one into bin 1, and stuff each remaining object into the first bin that can hold it. This (reasonable) approach may not give the absolute minimum number of bins needed, but it gives a first cut at the answer. (Anyone who has watched passengers stowing carry-on baggage in an airplane has seen this approximation algorithm at work.)

For example, suppose a sequence of four objects with volumes of 0.3, 0.4, 0.5, and 0.6 are stored in bins of size 1.0 using the "first-fit" algorithm described previously. The result requires three bins, which would be packed as shown in Figure 3.28. However, this is not the optimal solution (see Exercise 39 at the end of the chapter).

Figure 3.28A first-fit solution to a bin-packing problem

In Chapter 12, we will learn that there are problems that cannot be solved algorithmically, even if we are willing to accept an extremely inefficient solution.

Practice Problems

Consider the following graph:

Draw a tree similar to Figure 3.24 showing all paths from A and highlighting those that are Hamiltonian circuits (these are the same two circuits as before). How many paths must be examined?

Answer

38 paths

The following tree shows all paths with two links that begin at node A in some graph. Draw the graph.

Answer

If an algorithm were determined to have an order of magnitude of , do you think it would be classified as polynomial or exponential? Explain.
Answer

Algorithms of are polynomial algorithms because *n* is raised to a constant power. An algorithm of is an exponential algorithm. It is the nonconstant exponent that makes this value grow so quickly, as opposed to polynomial algorithms. An algorithm of would still be considered an exponential algorithm because its exponent is *n*. In fact, such an algorithm grows even faster than one of order because of the nonconstant base.

3.6 Summary of Level 1

We defined computer science as the study of algorithms, so it is appropriate that Level 1 was devoted to exploring algorithms in more detail. In Chapter 2, we discussed how to represent algorithms using pseudocode. Pseudocode provides us with a flexible language for expressing the building blocks from which algorithms can be constructed. These building blocks include assigning a particular value to a quantity, choosing one of two next steps on the basis of some condition, or repeating steps in a loop.

We developed algorithmic solutions to three very practical problems: searching a list of items for a particular target value, finding the largest number in a list of numbers, and searching for a particular pattern of characters within a segment of text. In Chapter 3, we noted that computer scientists develop algorithms to be *used* and thus there is a set of desirable properties for algorithms, including ease of understanding, elegance, and efficiency, in addition to correctness. Of these, efficiency—which may be either time efficiency or space efficiency—is the most easily quantifiable.

A convenient way to classify the time efficiency of algorithms is by examining the order of magnitude of the work they do. Algorithms that are of differing orders of magnitude do fundamentally different amounts of work. Regardless of the constant factor that reflects peripheral work or how fast the computer on which these algorithms execute, for problems with sufficiently large input, the algorithm of the lowest order of magnitude requires the least time.

We analyzed the time efficiency of the sequential search algorithm and discovered that it is an $\Theta(n)$ algorithm in both the worst case and the average case. We found a selection sort algorithm that is , we found a binary search algorithm that is $\Theta(\lg n)$, and we analyzed the pattern-matching algorithm from Chapter 2. By examining the data cleanup problem, we learned that algorithms that solve the same task can indeed differ in the order of magnitude of the work they do, sometimes by employing a time/ space trade-off. We also learned that there are algorithms that require more than polynomially bounded time to complete their work and that such algorithms may take so long to execute, regardless of the speed of the computer on which they are run, that they provide no practical solution. Some important problems may be intractable—that is, have no polynomially bounded solution algorithms.
Some computer scientists work on trying to decide whether a particular problem is intractable. Some work on finding more efficient algorithms for problems—such as searching and sorting—that are so common that a more efficient algorithm would greatly improve productivity. Still others seek to discover algorithms for new problems. Thus, as we said, the study of algorithms underlies much of computer science. But everything we have done so far has been a pencil-and-paper exercise. In terms of the definition of computer science that we gave in Chapter 1, we have been looking at the formal and mathematical properties of algorithms. It is time to move on to the next part of that definition: the hardware realizations of algorithms. When we execute real algorithms on real computers, those computers are electronic devices. How does an

electronic device “understand” an algorithm and carry out its instructions? We begin to explore these questions in Chapter 4.