Procedure BAD-FACTORIAL(n)
Input and Output: Same as FACTORIAL.
1. If \( n = 0 \), then return 1 as the output.
2. Otherwise, return \( \text{BAD-FACTORIAL}(n + 1)/(n + 1) \).

If we were to call BAD-FACTORIAL(1), it would generate a recursive call of BAD-FACTORIAL(2), which would generate a recursive call of BAD-FACTORIAL(3), and so on, never getting down to the base case when \( n \) equals 0. If you were to implement this procedure in a real programming language and run it on an actual computer, you would quickly see something like a "stack overflow error."

We can often rewrite algorithms that use a loop in a recursive style. Here is linear search, without a sentinel, written recursively:

Procedure RECURSIVE-LINEAR-SEARCH(A, n, i, x)
Inputs: Same as LINEAR-SEARCH, but with an added parameter \( i \).
Output: The index of an element equaling \( x \) in the subarray from \( A[i] \) through \( A[n] \), or NOT-FOUND if \( x \) does not appear in this subarray.
1. If \( i > n \), then return NOT-FOUND.
2. Otherwise \( (i \leq n) \), if \( A[i] = x \), then return \( i \).
3. Otherwise \( (i \leq n \text{ and } A[i] \neq x) \), return RECURSIVE-LINEAR-SEARCH(A, n, i + 1, x).

Here, the subproblem is to search for \( x \) in the subarray going from \( A[i] \) through \( A[n] \). The base case occurs in step 1 when this subarray is empty, that is, when \( i > n \). Because the value of \( i \) increases in each of step 3's recursive calls, if no recursive call ever returns a value of \( i \) in step 2, then eventually \( i \) becomes greater than \( n \) and we reach the base case.

Further reading

Chapters 2 and 3 of CLRS [CLRS09] cover much of the material in this chapter. An early algorithms textbook by Aho, Hopcroft, and Ullman [AHU74] influenced the field to use asymptotic notation to analyze algorithms. There has been quite a bit of work in proving programs correct; if you would like to delve into this area, try the books by Gries [Gri81] and Mitchell [Mit96].

3 Algorithms for Sorting and Searching

In Chapter 2, we saw three variations on linear search of an array. Can we do any better? The answer: it depends. If we know nothing about the order of the elements in the array, then no, we cannot do better. In the worst case, we have to look through all \( n \) elements because if we don’t find the value we’re looking for in the first \( n - 1 \) elements, it might be in that last, \( n \),th element. Therefore, we cannot achieve a better worst-case running time than \( \Theta(n) \) if we know nothing about the order of the elements in the array.

Suppose, however, that the array is sorted into nondecreasing order: each element is less than or equal to its successor in the array, according to some definition of "less than." In this chapter, we shall see that if an array is sorted, then we can use a simple technique known as binary search to search an \( n \)-element array in only \( O(\log n) \) time. As we saw in Chapter 1, the value of \( \log n \) grows very slowly compared with \( n \), and so binary search beats linear search in the worst case.\(^1\)

What does it mean for one element to be less than another? When the elements are numbers, it’s obvious. When the elements are strings of text characters, we can think of a lexicographic ordering: one element is less than another if it would come before the other element in a dictionary. When elements are some other form of data, then we have to define what “less than” means. As long as we have some clear notion of “less than,” we can determine whether an array is sorted.

Recalling the example of books on a bookshelf from Chapter 2, we could sort the books alphabetically by author, alphabetically by title, or, if in a library, by call number. In this chapter, we’ll say that the books are sorted on the shelf if they appear in alphabetical order by author, reading from left to right. The bookshelf might contain more than one book by the same author, however; perhaps you have several works by William Shakespeare. If we want to search for not just any book by

\(^1\)If you are a non-computer person who skipped the section "Computer algorithms for computer people" in Chapter 1, you ought to read the material about logarithms on page 7.
Shakespeare, but a specific book by Shakespeare, then we would say that if two books have the same author, then the one whose title is first alphabetically should go on the left. Alternatively, we could say that all we care about is the author’s name, so that when we search, anything by Shakespeare will do. We call the information that we are matching on the key. In our bookshelf example, the key is just the author’s name, rather than a combination based first on the author’s name and then the title in case of two works by the same author.

How, then, do we get the array to be sorted in the first place? In this chapter, we’ll see four algorithms—selection sort, insertion sort, merge sort, and quicksort—to sort an array, applying each of these algorithms to our bookshelf example. Each sorting algorithm will have its advantages and its disadvantages, and at the end of the chapter we’ll review and compare these sorting algorithms. All of the sorting algorithms that we’ll see in this chapter take either $\Theta(n^2)$ or $\Theta(n \log n)$ time in the worst case. Therefore, if you were going to perform only a few searches, you’d be better off just running linear search. But if you were going to search many times, you might be better off first sorting the array and then searching by running binary search.

Sorting is an important problem in its own right, not just as a preprocessing step for binary search. Think of all the data that must be sorted, such as entries in a phone book, by name; checks in a monthly bank statement, by check numbers and/or the dates that the checks were processed by the bank; or even results from a Web-search engine, by relevance to the query. Furthermore, sorting is often a step in some other algorithm. For example, in computer graphics, objects are often layered on top of each other. A program that renders objects on the screen might have to sort the objects according to an “above” relation so that it can draw these objects from bottom to top.

Before we proceed, a word about what it is that we sort. In addition to the key (which we’ll call a sort key when we’re sorting), the elements that we sort usually include as well what we call satellite data. Although satellite data could come from a satellite, it usually does not. Satellite data is the information that is associated with the sort key and should travel with it when elements are moved around. In our bookshelf example, the sort key is the author’s name and the satellite data is the book itself.

I explain satellite data to my students in a way that they are sure to understand. I keep a spreadsheet with student grades, with rows sorted alphabetically by student name. To determine final course grades at the end of the term, I sort the rows, with the sort key being the column containing the percentage of points obtained in the course, and the rest of the columns, including the student names, as the satellite data. I sort into decreasing order by percentage, so that rows at the top correspond to A’s and rows at the bottom to D’s and E’s. Suppose that I were to rearrange only the column containing the percentages and not move the entire row containing the percentage. That would leave the student names in alphabetical order regardless of percentages. Then the students whose names appear early in the alphabet would be happy while the students with names at the end of the alphabet—not so much.

Here are some other examples of sort keys and satellite data. In a phone book, the sort key would be the name and the satellite data would be the address and phone number. In a bank statement, the sort key would be the check number and the satellite data would include the amount of the check and the date it cleared. In a search engine, the sort key would be the measure of relevance to the query and the satellite data would be the URL of the Web page, plus whatever other data the search engine stores about the page.

When we work with arrays in this chapter, we will act as though each element contains only a sort key. If you were implementing any of the sorting algorithms here, you would have to make sure that you move the satellite data associated with each element, or at least a pointer to the satellite data, whenever you move the sort key.

In order for the bookshelf analogy to apply to arrays in a computer, we need to assume that the bookshelf and its books have two additional features, which I admit are not terribly realistic. First, all books on the bookshelf are the same size, because in a computer array, all array entries are the same size. Second, we can number the positions of the books on the bookshelf from 1 to $n$, and we will call each position a slot. Slot 1 is the leftmost slot, and slot $n$ is the rightmost. As you have probably guessed, each slot on the bookshelf corresponds to an array entry.

I also want to address the word “sorting.” In common speech, sorting can mean something different from how we use it in computing.

---

2Dartmouth uses E, not F, to indicate a failing grade. I don’t know why for sure, but I would guess that it simplified the computer program that converts letter grades to numeric grades on a 4.0 scale.
My Mac's online dictionary defines "sort" by "arrange systematically in groups; separate according to type, class, etc.": the way that you might "sort" clothing for example, with shirts in one place, pants in another place, and so on. In the world of computer algorithms, sorting means to put into some well-defined order, and "arranging systematically in groups" is called "bucketing," "bucketizing," or "binning."

**Binary search**

Before we see some sorting algorithms, let's visit binary search, which requires the array being searched to be already sorted. Binary search has the advantage that it takes only \( O(\lg n) \) time to search an \( n \)-element array.

In our bookshelf example, we start with the books already sorted by author name, left to right on the shelf. We'll use the author name as the key, and let's search for any book by Jonathan Swift. Now, you might figure that because the author's last name starts with "S," which is the 19th letter of the alphabet, you could go about three-quarters of the way over on the shelf (since 19/26 is close to 3/4) and look in the slot there. But if you have all of Shakespeare's works, then you have several books by an author whose last name comes between Swift, which could push books by Swift farther to the right than you expected.

Instead, here's how you could apply binary search to finding a book by Jonathan Swift. Go to the slot exactly halfway over on the shelf, find the book there, and examine the author's name. Let's say that you've found a book by Jack London. Not only is that not the book you're searching for, but because you know that the books are sorted alphabetically by author, you know that all books to the left of the book by London can't be what you're searching for. By looking at just one book, you have eliminated half of the books on the shelf from consideration! Any books by Swift must be on the right-hand half of the shelf. So now you find the slot at the halfway point of just the right-hand half and look at the book there. Suppose that it's by Leo Tolstoy. Again, that's not the book you're searching for, but you know that you can eliminate all books to the right of this one: half of the books that remained as possibilities. At this point, you know that if your bookshelf contains any books by Swift, then they are in the quarter of the books that are to the right of the book by London and to the left of the book by Tolstoy. Next, you find the book in the slot at the midpoint within this quarter under consideration. If it's by Swift, you are done. Otherwise, you can again eliminate half of the remaining books. Eventually, you either find a book by Swift or you get to the point at which no slots remain as possibilities. In the latter case, you conclude that the bookshelf contains no books by Jonathan Swift.

In a computer, we perform binary search on an array. At any point, we are considering only a subarray, that is, the portion of the array between and including two indices; let's call them \( p \) and \( r \). Initially, \( p = 1 \) and \( r = n \), so that the subarray starts out as the entire array. We repeatedly halve the size of the subarray that we are considering until one of two things happens: we either find the value that we're searching for or the subarray is empty (that is, \( p \) becomes greater than \( r \)). The repeated halving of the subarray size is what gives rise to the \( O(\lg n) \) running time.

In a little more detail, here's how binary search works. Let's say that we're searching for the value \( x \) in array \( A \). In each step, we are considering only the subarray starting at \( A[p] \) and ending at \( A[r] \). Because we will be working with subarrays quite a bit, let's denote this subarray by \( A[p..r] \). At each step, we compute the midpoint \( q \) of the subarray under consideration by taking the average of \( p \) and \( r \) and then dropping the fractional part, if any: \( q = \lfloor (p + r)/2 \rfloor \). (Here, we use the "floor" operation, \([ ]\), to drop the fractional part. If you were implementing this operation in a language such as Java, C, or C++, you could just use integer division to drop the fractional part.) We check to see whether \( A[q] \) equals \( x \); if it does, we are done, because we can just return \( q \) as an index where array \( A \) contains \( x \).

If instead, we find that \( A[q] \neq x \), then we take advantage of the assumption that array \( A \) is already sorted. Since \( A[q] \neq x \), there are two possibilities: either \( A[q] > x \) or \( A[q] < x \). We first handle the case where \( A[q] > x \). Because the array is sorted, we know that not only is \( A[q] \) greater than \( x \), but also—thinking of the array as laid out from left to right—that every array element to the right of \( A[q] \) is greater than \( x \). Therefore, we can eliminate from consideration all elements at or to the right of \( A[q] \). We will start our next step with \( p \) unchanged, but with \( r \) set to \( q - 1 \):

```
A ... >x >x >x >x >x >x >x ...
```

\[
q = \lfloor (p + r)/2 \rfloor
\]

\[
\text{new } r
\]
If instead we find that \( A[q] < x \), we know that every array element at or to the left of \( A[q] \) is less than \( x \), and so we can eliminate these elements from consideration. We will start our next step with \( r \) unchanged, but with \( p \) set to \( q + 1 \):

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
\hline
& A & \cdots & <x & <x & <x & <x & <x & \cdots \\
\hline
p & \quad & \quad & \quad & \quad & \quad & \quad & \quad & \quad \\
q & \quad & \quad & \quad & \quad & \quad & \quad & \quad & \quad \\
r & \quad & \quad & \quad & \quad & \quad & \quad & \quad & \quad \\
\hline
\end{array}
\]

new \( p \)

Here is the exact procedure for binary search:

**Procedure Binary-Search** \( (A, n, x) \)

**Input and Outputs**: Same as **Linear-Search**.

1. Set \( p \) to 1, and set \( r \) to \( n \).
2. While \( p \leq r \), do the following:
   - A. Set \( q \) to \( \lfloor (p + r)/2 \rfloor \).
   - B. If \( A[q] = x \), then return \( q \).
   - C. Otherwise (\( A[q] \neq x \)), if \( A[q] > x \), then set \( r \) to \( q - 1 \).
   - D. Otherwise (\( A[q] < x \)), set \( p \) to \( q + 1 \).
3. Return **Not-Found**.

The loop in step 2 does not necessarily terminate because \( p \) becomes greater than \( r \). It can terminate in step 2B because it finds that \( A[q] \) equals \( x \) and returns \( q \) as an index in \( A \) where \( x \) occurs.

In order to show that the **Binary-Search** procedure works correctly, we just need to show that \( x \) is not present anywhere in the array if **Binary-Search** returns **Not-Found** in step 3. We use the following loop invariant:

At the start of each iteration of the loop of step 2, if \( x \) is anywhere in the array \( A \), then it is somewhere in the subarray \( A[p \ldots r] \).

And a brief argument using the loop invariant:

**Initialization**: Step 1 initializes the indices \( p \) and \( r \) to 1 and \( n \), respectively, and so the loop invariant is true when the procedure first enters the loop.

**Maintenance**: We argued above that steps 2C and 2D adjust either \( p \) or \( r \) correctly.

**Termination**: If \( x \) is not in the array, then eventually the procedure gets to the point where \( p \) and \( r \) are equal. When that happens, step 2A computes \( q \) to be the same as \( p \) and \( r \). If step 2C sets \( r \) to \( q - 1 \), then at the start of the next iteration, \( r \) will equal \( p - 1 \), so that \( p \) will be greater than \( r \). If step 2D sets \( p \) to \( q + 1 \), then at the start of the next iteration, \( p \) will equal \( r + 1 \), and again \( p \) will be greater than \( r \). Either way, the loop test in step 2 will come up false, and the loop will terminate. Because \( p > r \), the subarray \( A[p \ldots r] \) will be empty, and so the value \( x \) cannot be present in it. Taking the contrapositive of the loop invariant (see page 22) gives us that if \( x \) is not present in the subarray \( A[p \ldots r] \), then it is not present anywhere in array \( A \). Therefore, the procedure is correct in returning **Not-Found** in step 3.

We can also write binary search as a recursive procedure:

**Procedure Recursive-Binary-Search** \( (A, p, r, x) \)

**Input and Output**: Inputs \( A \) and \( x \) are the same as **Linear-Search**, as is the output. The inputs \( p \) and \( r \) delineate the subarray \( A[p \ldots r] \) under consideration.

1. If \( p > r \), then return **Not-Found**.
2. Otherwise (\( p \leq r \)), do the following:
   - A. Set \( q \) to \( \lfloor (p + r)/2 \rfloor \).
   - B. If \( A[q] = x \), then return \( q \).
   - C. Otherwise (\( A[q] \neq x \)), if \( A[q] > x \), then return \( \text{Recursive-Binary-Search}(A, p, q - 1, x) \).
   - D. Otherwise (\( A[q] < x \)), return \( \text{Recursive-Binary-Search}(A, q + 1, r, x) \).

The initial call is **Recursive-Binary-Search** \( (A, 1, n, x) \).

Now let’s see how it is that binary search takes \( O(\log n) \) time on an \( n \)-element array. The key observation is that the size \( r - p + 1 \) of the subarray under consideration is approximately halved in each iteration of the loop (or in each recursive call of the recursive version, but let’s focus on the iterative version in **Binary-Search**). If you try all the cases, you’ll find that if an iteration starts with a subarray of \( s \) elements, the next iteration will have either \( \lfloor s/2 \rfloor \) or \( s/2 - 1 \) elements, depending on whether \( s \) is even or odd and whether \( A[q] \) is greater than or less than \( x \). We have already seen that once the subarray size gets down to 1,
the procedure will finish by the next iteration. So we can ask, how many iterations of the loop do we need to repeatedly halve a subarray from its original size \( n \) down to a size of 1? That would be the same as the number of times that, starting with a subarray of size 1, we would need to double its size to reach a size of \( n \). But that’s just exponentiation: repeatedly multiplying by 2. In other words, for what value of \( x \) does \( 2^x \) reach \( n \)? If \( n \) were an exact power of 2, then we’ve already seen on page 7 that the answer is \( \lg n \). Of course, \( n \) might not be an exact power of 2, in which case the answer will be within 1 of \( \lg n \). Finally, we note that each iteration of the loop takes a constant amount of time, that is, the time for a single iteration does not depend on the size \( n \) of the original array or on the size of the subarray under consideration. Let’s use asymptotic notation to suppress the constant factors and low-order term. (Is the number of loop iterations \( \lg n \) or \( \lfloor \lg n \rfloor + 1 \)? Who cares?)

We get that the running time of binary search is \( O(\lg n) \).

I used \( O \)-notation here because I wanted to make a blanket statement that covers all cases. In the worst case, when the value \( x \) is not present in the array, we halved and halved and halved until the subarray under consideration was empty, yielding a running time of \( \Theta(\lg n) \). In the best case, when \( x \) is found in the first iteration of the loop, the running time is \( \Theta(1) \). No \( \Theta \)-notation covers all cases, but a running time of \( O(\lg n) \) is always correct for binary search—as long as the array is already sorted.

It is possible to beat \( \Theta(\lg n) \) worst-case time for searching, but only if we organize data in more elaborate ways and make certain assumptions about the keys.

**Selection sort**

We now turn our attention to **sorting**: rearranging the elements of the array—also known as **permuting** the array—so that each element is less than or equal to its successor. The first sorting algorithm we’ll see, selection sort, is the one I consider the simplest, because it’s the one I came up with when I first needed to design a sorting algorithm. It is far from the fastest.

Here is how selection sort would work for sorting books on a bookshelf according to author names. Go through the entire shelf and find the book whose author’s name comes earliest in the alphabet. Let’s say that it’s by Louisa May Alcott. (If the shelf contains two or more books by this author, choose any one of them.) Swap the location of this book with the book in slot 1. The book in slot 1 is now a book by an author whose name comes first alphabetically. Now go through the bookshelf, left to right, starting with the book in slot 2 to find the book in slots 2 through \( n \) whose author name comes earliest in the alphabet. Suppose that it’s by Jane Austen. Swap the location of this book with the book in slot 2, so that now slots 1 and 2 have the first and second books in the overall alphabetical ordering. Then do the same for slot 3, and so on. Once we have put the correct book into slot \( n – 1 \) (perhaps it’s by H. G. Wells), we are done, because there’s only one book left (say, a book by Oscar Wilde), and it’s in slot \( n \) where it belongs.

To turn this approach into a computer algorithm, change the bookshelf to an array and the books to array elements. Here’s the result:

![Procedure SELECTION-SORT(A,n)](image)

Finding the smallest element in \( A[i \ldots n] \) is a variant on linear search. First declare \( A[i] \) to be the smallest element seen in the subarray so far, and then go through the rest of the subarray, updating the index of the smallest element every time we find an element less than the current smallest. Here’s the refined procedure:

![Procedure SELECTION-SORT(A,n)](image)
This procedure has “nested” loops, with the loop of step 1B nested within the loop of step 1. The inner loop performs all of its iterations for each individual iteration of the outer loop. Notice that the starting value of \( j \) in the inner loop depends on the current value of \( i \) in the outer loop. This illustration shows how selection sort works on an array of six elements:

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
12 & 9 & 3 & 7 & 14 & 11
\end{array}
\rightarrow
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
3 & 9 & 12 & 7 & 14 & 11
\end{array}
\rightarrow
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
3 & 7 & 12 & 9 & 14 & 11
\end{array}
\]

The initial array appears in the upper left, and each step shows the array after an iteration of the outer loop. The darker shaded elements hold the subarray that is known to be sorted.

If you wanted to use a loop invariant to argue that the `SELECTION-SORT` procedure sorts the array correctly, you would need one for each of the loops. This procedure is simple enough that we won’t go through the full loop-invariant arguments, but here are the loop invariants:

- At the start of each iteration of the loop of step 1, the subarray \( A[1..i-1] \) holds the \( i-1 \) smallest elements of the entire array \( A \), and they are in sorted order.
- At the start of each iteration of the loop of step 1B, \( A[\text{smallest}] \) is the smallest element in the subarray \( A[i..j-1] \).

What is the running time of `SELECTION-SORT`? We’ll show that it is \( \Theta(n^2) \). The key is to analyze how many iterations the inner loop makes, noting that each iteration takes \( \Theta(1) \) time. (Here, the constant factors in the lower and upper bounds in the \( \Theta \)-notation may differ, because the assignment to `smallest` may or may not occur in a given iteration.) Let’s count the number of iterations, based on the value of the loop variable \( i \) in the outer loop. When \( i \) equals 1, the inner loop iterates for \( j \) running from 2 to \( n \), or \( n - 1 \) times. When \( i \) equals 2, the inner loop iterates for \( j \) running from 3 to \( n \), or \( n - 2 \) times. Each time the outer loop increments \( i \), the inner loop runs one time less. In general, the inner loop runs \( n - i \) times. In the last iteration of the outer loop, when \( i \) equals \( n - 1 \), the inner loop iterates just one time. Therefore, the total number of inner-loop iterations is

\[(n - 1) + (n - 2) + (n - 3) + \cdots + 2 + 1.\]

This summation is known as an arithmetic series, and here’s a basic fact about arithmetic series: for any nonnegative integer \( k \),

\[k + (k - 1) + (k - 2) + \cdots + 2 + 1 = \frac{k(k + 1)}{2} \cdot \frac{1}{2}.

Substituting \( n - 1 \) for \( k \), we see that the total number of inner-loop iterations is \((n - 1)n/2, or \((n^2 - n)/2. \) Let’s use asymptotic notation to get rid of the low-order term \(( - n) \) and the constant factor \((1/2) \). Then we can say that the total number of inner-loop iterations is \( \Theta(n^2) \). Therefore, the running time of `SELECTION-SORT` is \( \Theta(n^2) \). Notice that this running time is a blanket statement that covers all cases. Regardless of the actual element values, the inner loop runs \( \Theta(n^2) \) times.

Here’s another way to see that the running time is \( \Theta(n^2) \), without using the arithmetic series. We’ll show separately that the running time is both \( O(n^2) \) and \( \Omega(n^2) \); putting the asymptotic upper and lower bounds together gives us \( \Theta(n^2) \). To see that the running time is \( O(n^2) \), observe that each iteration of the outer loop runs the inner loop at most \( n - 1 \) times, which is \( O(n) \) because each iteration of the inner loop takes a constant amount of time. Since the outer loop iterates \( n - 1 \) times, which is also \( O(n) \), the total time spent in the inner loop is \( O(n) \) times \( O(n) \), or \( O(n^2) \). To see that the running time is \( \Omega(n^2) \), observe that in each of the first \( n/2 \) iterations of the outer loop, we run the inner loop at least \( n/2 \) times, for a total of at least \( n/2 \) times \( n/2 \), or \( n^2/4 \) times. Since each inner-loop iteration takes a constant amount of time, we see that the running time is at least a constant times \( n^2/4 \), or \( \Omega(n^2) \).

Two final thoughts about selection sort. First, we’ll see that its asymptotic running time of \( \Theta(n^2) \) is the worst of the sorting algorithms that we’ll examine. Second, if you carefully examine how selection sort operates, you’ll see that the \( \Theta(n^2) \) running time comes from the comparisons in step 1Bi. But the number of times that it moves array elements is only \( \Theta(n) \), because step 1C runs only \( n - 1 \) times. If moving array elements is particularly time-consuming—perhaps because they are large or stored on a slow device such as a disk—then selection sort might be a reasonable algorithm to use.

**Insertion sort**

Insertion sort differs a bit from selection sort, though it has a similar flavor. In selection sort, when we decided which book to put into the \( i \)th slot, the books in the first \( i \) slots were the first \( i \) books *overall*, sorted
alphabetically by author name. In insertion sort, the books in the first \( i \) slots will be the same books that were originally in the first \( i \) slots, but now sorted by author name.

For example, let's suppose that the books in the first four slots are already sorted by author name, and that, in order, they are books by Charles Dickens, Herman Melville, Jonathan Swift, and Leo Tolstoy. Let's say that the book that starts in slot 5 is by Sir Walter Scott. With insertion sort, we shift the books by Swift and Tolstoy by one slot to the right, moving them from slots 3 and 4 to slots 4 and 5, and then we put the book by Scott into the vacated slot 3. At the time that we work with the book by Scott, we don't care what books are to its right (the books by Jack London and Gustave Flaubert in the figure below); we deal with them later on.

To shift the books by Swift and Tolstoy, we first compare the author name Tolstoy with Scott. Finding that Tolstoy comes after Scott, we shift the book by Tolstoy one slot to the right, from slot 4 to slot 5. Then we compare the author name Swift with Scott. Finding that Swift comes after Scott, we shift the book by Swift one slot to the right, from slot 3 to slot 4, which was vacated when we shifted the book by Tolstoy. Next we compare the author name Herman Melville with Scott. This time, we find that Melville does not come after Scott. At this point, we stop comparing author names, because we have found that the book by Scott should be to the right of the book by Melville and to the left of the book by Swift. We can put the book by Scott into slot 3, which was vacated when we shifted the book by Swift.

To translate this idea to sorting an array with insertion sort, the subarray \( A[1..i-1] \) will hold only the elements originally in the first \( i-1 \) positions of the array, and they will be in sorted order. To determine where the element originally in \( A[i] \) goes, insertion sort marches through \( A[1..i-1] \), starting at \( A[i-1] \) and going toward the left, shifting each element greater than this one by one position to the right. Once we find an element that is not greater than \( A[i] \) or we hit the left end of the array, we drop the element originally in \( A[i] \) into its new position in the array.

**Procedure** \textsc{insertion-sort}(\( A, n \))

\textbf{Inputs and Result:} Same as \textsc{selection-sort}.

1. For \( i = 2 \) to \( n \):
   a. Set key to \( A[i] \), and set \( j \) to \( i - 1 \).
   b. While \( j > 0 \) and \( A[j] > \text{key} \), do the following:
      i. Set \( A[j+1] \) to \( A[j] \).
      ii. Decrease \( j \) (i.e., set \( j \) to \( j - 1 \)).
   c. Set \( A[j+1] \) to key.

The test in step 1B relies on the "and" operator being short circuiting: if the expression on the left, \( j > 0 \), is false, then it does not evaluate the expression on the right, \( A[j] > \text{key} \). If it did attempt to access \( A[j] \) when \( j \leq 0 \), an array indexing error would occur.

Here is how insertion sort works on the same array as we saw on page 34 for selection sort:

Once again, the initial array appears in the upper left, and each step shows the array after an iteration of the outer loop of step 1. The darker shaded elements hold the subarray that is known to be sorted. The loop invariant for the outer loop (again, we won't prove it) is the following:

At the start of each iteration of the loop of step 1, the subarray \( A[1..i-1] \) consists of the elements originally in \( A[1..i-1] \), but in sorted order.

The next illustration demonstrates how the inner loop of step 1B works in the above example when \( i \) equals \( 4 \). We assume that the subarray \( A[1..3] \) contains the elements originally in the first three array
positions, but now they are sorted. To determine where to place the element originally in $A[4]$, we save it in a variable named $key$, and then shift each element in $A[1..3]$ that is greater than $key$ by one position to the right:

$$
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
3 & 9 & 12 & 7 & 14 & 11 \\
\end{array}
\rightarrow
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
1 & 3 & 9 & 12 & 7 & 14 \\
\end{array}
$$

$\text{key} = 7$

$$
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
3 & 9 & 12 & 7 & 14 & 11 \\
\end{array}
\rightarrow
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
3 & 7 & 9 & 12 & 14 & 11 \\
\end{array}
$$

The darker shaded positions show where elements move to. In the last step shown, the value of $A[1]$, 3, is not greater than the value of $key$, 7, and so the inner loop terminates. The value of $key$ drops into the position just to the right of $A[1]$, as the last step shows. Of course, we have to save the value originally in $A[i]$ into $key$ in step 1A, because the first iteration of the inner loop overwrites $A[i]$.

It is also possible that the inner loop terminates because the test $j > 0$ comes up false. This situation occurs if $key$ is less than all the elements in $A[1..i-1]$. When $j$ becomes 0, each element in $A[1..i-1]$ has been shifted to the right, and so step 1C drops $key$ into $A[1]$, right where we want it.

When we analyze the running time of insertion sort, it becomes a bit trickier than selection sort. The number of times that the inner loop iterates in the selection sort procedure depends only on the index $i$ of the outer loop and not at all on the elements themselves. For the insertion sort procedure, however, the number of times that the inner loop iterates depends on both the index $i$ of the outer loop and the values in the array elements.

The best case of insertion sort occurs when the inner loop makes zero iterations every time. For that to happen, the test $A[j] > key$ must come up false the first time for each value of $i$. In other words, we must have $A[i-1] \leq A[i]$ every time that step 1B executes. How can this situation happen? Only if the array $A$ is already sorted when the procedure starts. In this case, the outer loop iterates $n - 1$ times, and each iteration of the outer loop takes a constant amount of time, so that insertion sort takes only $\Theta(n)$ time.

The worst case occurs when the inner loop makes the maximum possible number of iterations every time. Now the test $A[j] > key$ must always come up true, and the loop must terminate because the test $j > 0$ comes up false. Each element $A[i]$ must travel all the way to the left of the array. How can this situation happen? Only if the array $A$ starts in reverse sorted order, that is, sorted into nonincreasing order. In this case, for each time the outer loop iterates, the inner loop iterates $i - 1$ times. Since the outer loop runs with $i$ going from 2 up to $n$, the number of inner-loop iterations forms an arithmetic series:

$$1 + 2 + 3 + \cdots + (n - 2) + (n - 1),$$

which, as we saw for selection sort, is $\Theta(n^2)$. Since each inner-loop iteration takes constant time, the worst-case running time of insertion sort is $\Theta(n^2)$. In the worst case, therefore, selection sort and insertion sort have running times that are asymptotically the same.

Would it make sense to try to understand what happens on average with insertion sort? That depends on what an “average” input looks like. If the ordering of elements in the input array were truly random, we would expect each element to be larger than about half the elements preceding it and smaller than about half the elements preceding it, so that each time the inner loop runs, it would make approximately $(i - 1)/2$ iterations. That would cut the running time in half, compared with the worst case. But $1/2$ is just a constant factor, and so, asymptotically, it would be no different from the worst-case running time: still $\Theta(n^2)$.

Insertion sort is an excellent choice when the array starts out as “almost sorted.” Suppose that each array element starts out within $k$ positions of where it ends up in the sorted array. Then the total number of times that a given element is shifted, over all iterations of the inner loop, is at most $kn$. Therefore, the total number of times that all elements are shifted, over all inner-loop iterations, is at most $kn$, which in turn tells us that the total number of inner-loop iterations is at most $kn$ (since each inner-loop iteration shifts exactly one element by one position). If $k$ is a constant, then the total running time of insertion sort would be only $\Theta(n)$, because the $\Theta$-notation subsumes the constant factor $k$. In fact, we can even tolerate some elements moving a long distance in the array, as long as there are not too many such elements. In particular, if $l$ elements can move anywhere in the array (so that each of these elements can move by up to $n - 1$ positions), and the remaining $n - l$ elements can move at most $k$ positions, then the total number of shifts is at most $l(n - 1) + (n - l)k = (k + l)n - (k + 1)l$, which is $\Theta(n)$ if both $k$ and $l$ are constants.

If we compare the asymptotic running times of insertion sort and selection sort, we see that in the worst case, they are the same. Insertion sort is better if the array is almost sorted. Selection sort has one
advantage over insertion sort, however: selection sort moves elements \( \Theta(n) \) times no matter what, but insertion sort could move elements up to \( \Theta(n^2) \) times, since each execution of step 1B of INSERTION-SORT moves an element. As we noted on page 35 for selection sort, if moving an element is particularly time-consuming and you have no reason to expect that insertion sort's input approaches the best-case situation, then you might be better off running selection sort instead of insertion sort.

**Merge sort**

Our next sorting algorithm, merge sort, has a running time of only \( \Theta(n \lg n) \) in all cases. When we compare its running time with the \( \Theta(n^2) \) worst-case running times of selection sort and insertion sort, we are trading a factor of \( n \) for a factor of only \( \lg n \). As we noted on page 7 back in Chapter 1, that’s a trade you should take any day.

Merge sort does have a couple of disadvantages compared with the other two sorting algorithms we have seen. First, the constant factor that we hide in the asymptotic notation is higher than for the other two algorithms. Of course, once the array size \( n \) gets large enough, that doesn’t really matter. Second, merge sort does not work in place: it has to make complete copies of the entire input array. Contrast this feature with selection sort and insertion sort, which at any time keep an extra copy of only one array entry rather than copies of all the array entries. If space is at a premium, you might not want to use merge sort.

We employ a common algorithmic paradigm known as divide-and-conquer in merge sort. In divide-and-conquer, we break the problem into subproblems that are similar to the original problem, solve the subproblems recursively, and then combine the solutions to the subproblems to solve the original problem. Recall from Chapter 2 that in order for recursion to work, each recursive call must be on a smaller instance of the same problem that will eventually hit a base case. Here’s a general outline of a divide-and-conquer algorithm:

1. **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
2. **Conquer** the subproblems by solving them recursively. If they are small enough, solve the subproblems as base cases.
3. **Combine** the solutions to the subproblems into the solution for the original problem.

When we sort the books on our bookshelf with merge sort, each subproblem consists of sorting the books in consecutive slots on the shelf. Initially, we want to sort all \( n \) books, in slots 1 through \( n \), but in a general subproblem, we will want to sort all the books in slots \( p \) through \( r \). Here’s how we apply divide-and-conquer:

1. **Divide** by finding the number \( q \) of the slot midway between \( p \) and \( r \).
   We do so in the same way that we found the midpoint in binary search: add \( p \) and \( q \), divide by 2, and take the floor.
2. **Conquer** by recursively sorting the books in each of the two subproblems created by the divide step: recursively sort the books that are in slots \( p \) through \( q \), and recursively sort the books that are in slots \( q + 1 \) through \( r \).
3. **Combine** by merging the sorted books that are in slots \( p \) through \( q \) and slots \( q + 1 \) through \( r \), so that all the books in slots \( p \) through \( r \) are sorted. We’ll see how to merge books in a moment.

The base case occurs when fewer than two books need to be sorted (that is, when \( p \geq r \)), since a set of books with no books or one book is already trivially sorted.

To convert this idea to sorting an array, the books in slots \( p \) through \( r \) corresponding to the subarray \( A[p \ldots r] \). Here is the merge sort procedure, which calls a procedure MERGE\((A, p, q, r)\) to merge the sorted subarrays \( A[p \ldots q] \) and \( A[q + 1 \ldots r] \) into the single sorted subarray \( A[p \ldots r] \):

**Procedure Merge-Sort\((A, p, r)\)**

**Inputs:**
- \( A \): an array.
- \( p, r \): starting and ending indices of a subarray of \( A \).

**Result:** The elements of the subarray \( A[p \ldots r] \) are sorted into nondecreasing order.

1. If \( p \geq r \), then the subarray \( A[p \ldots r] \) has at most one element, and so it is already sorted. Just return without doing anything.
2. Otherwise, do the following:
   A. Set \( q \) to \( \lfloor (p + r)/2 \rfloor \).
   B. Recursively call Merge-Sort\((A, p, q)\).
   C. Recursively call Merge-Sort\((A, q + 1, r)\).
   D. Call Merge\((A, p, q, r)\).
Although we have yet to see how the MERGE procedure works, we can look at an example of how the MERGE-SORT procedure operates. Let's start with this array:

\[
\begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
12 & 9 & 3 & 7 & 14 & 11 & 6 & 2 & 10 & 5
\end{array}
\]

The initial call is MERGE-SORT(A, 1, 10). Step 2A computes \( q \) to be 5, so that the recursive calls in steps 2B and 2C are MERGE-SORT(A, 1, 5) and MERGE-SORT(A, 6, 10):

\[
\begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
12 & 9 & 3 & 7 & 14 & 11 & 6 & 2 & 10 & 5
\end{array}
\]

After the two recursive calls return, these two subarrays are sorted:

\[
\begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
3 & 7 & 9 & 12 & 14 & 6 & 2 & 10 & 5
\end{array}
\]

Finally, the call MERGE(A, 1, 5, 10) in step 2D merges the two sorted subarrays into a single sorted subarray, which is the entire array in this case:

\[
\begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
2 & 3 & 5 & 6 & 7 & 9 & 10 & 11 & 12 & 14
\end{array}
\]

If we unfold the recursion, we get the figure on the next page. Diverging arrows indicate divide steps, and converging arrows indicate merge steps. The variables \( p \), \( q \), and \( r \) appearing above each subarray are located at the indices to which they correspond in each recursive call. The italicized numbers give the order in which the procedure calls occur after the initial call MERGE-SORT(A, 1, 10). For example, the call MERGE(A, 1, 3, 5) is the 13th procedure call after the initial call, and the call MERGE-SORT(A, 6, 7) is the 16th call.

The real work happens in the MERGE procedure. Therefore, not only must the MERGE procedure work correctly, but it must also be fast. If we are merging a total of \( n \) elements, the best we can hope for is \( \Theta(n) \) time, since each element has to be merged into its proper place, and indeed we can achieve linear-time merging.
Returning to our book example, let's look at just the portion of the bookshelf from slot 9 through slot 14. Suppose that we have sorted the books in slots 9–11 and that we have sorted the books in slots 12–14:

![Bookshelf diagram with sorted books from slot 9 to slot 14.]

We pull out the books in slots 9–11 and make a pile of them, with the book whose author is alphabetically first on top, and we do the same with the books in slots 12–14, making a separate pile:

![Pile of books with authors alphabetically ordered.]

Because the two piles are already sorted, the book that should go back into slot 9 must be one of the books atop its pile: either the book by Gustave Flaubert or the book by Charles Dickens. Indeed, we see that the book by Dickens comes before the book by Flaubert, and so we move it into slot 9:

![Book moved to slot 9.]

Next, we compare the books now atop their piles, which are by Jonathan Swift and London, and we move the book by London into slot 11. That leaves the book by Sir Walter Scott atop the right pile, and when we compare it with the book by Swift, we move the book by Scott into slot 12. At this point, the right pile is empty:

![Book moved to slot 11 and 12.]

All that remains is to move the books in the left pile into the remaining slots, in order. Now all books in slots 9–14 are sorted:

![Bookshelf diagram with all books sorted.]

How efficient is this merge procedure? We move each book exactly twice: once to pull it off the shelf and put it into a pile, and once to move it from the top of a pile back onto the shelf. Furthermore, whenever we are deciding which book to put back onto the shelf, we need to compare only two books: those atop their piles. To merge $n$ books, therefore, we move books $2n$ times and compare pairs of books at most $n$ times.

Why pull the books off the shelf? What if we had left the books on the shelf and just kept track of which books we had put into their correct
slots on the shelf and which books we hadn't? That could turn out to be a lot more work. For example, suppose that every book in the right half should come before every book in the left half. Before we could move the first book from the right half into the first slot of the left half, we would have to shift every book that started in the left half to the right by one slot in order to make room. And then we'd have to do the same to put the next book that started in the right half into the second slot of the left half. And the same for all the other books that started in the right half. We would have to shift half the books—all the books that started in the left half—each time we wanted to put a book that started in the right half into its correct slot.

That argument explains why we do not merge in place.3 Returning to how we merge the sorted subarrays \( A[p .. q] \) and \( A[q + 1 .. r] \) into the subarray \( A[p .. r] \), we start by copying the elements to be merged from array \( A \) into temporary arrays and then merge them back into \( A \). Let \( n_1 = q - p + 1 \) be the number of elements in \( A[p .. q] \) and \( n_2 = r - q \) be the number of elements in \( A[q + 1 .. r] \). We create temporary arrays \( B \) with \( n_1 \) elements and \( C \) with \( n_2 \) elements, and we copy the elements in \( A[p .. q] \), in order, into \( B \), and likewise the elements in \( A[q + 1 .. r] \), in order, into \( C \). Now we can merge these elements back into \( A[p .. q] \) without fear of overwriting our only copies of them.

We merge the array elements the same way we merge books. We copy elements from the arrays \( B \) and \( C \) back into the subarray \( A[p .. r] \), maintaining indices to keep track of the smallest element not yet copied back in both \( B \) and \( C \), and copying back the smaller of the two. In constant time, we can determine which element is smaller, copy it back into the correct position of \( A[p .. r] \), and update indices into the arrays.

Eventually, one of the two arrays will have all its elements copied back to \( A[p .. r] \). This moment corresponds to the moment when only one pile of books remains. But we use a trick to avoid having to check each time whether one of the arrays has been exhausted: we place at the right end of each of the arrays \( B \) and \( C \) an extra element that is greater than any other element. Do you recall the sentinel trick that we used in SENTINEL-LINEAR-SEARCH in Chapter 2? This idea is similar. Here, we use \( \infty \) (infinity) as the sentinel's sort key, so that whenever an element with a sort key of \( \infty \) is the smallest remaining element in its array, it is guaranteed to "lose" the contest to see which array has the smaller remaining element.4 Once all elements from both arrays \( B \) and \( C \) have been copied back, both arrays have their sentinels as their smallest remaining element. But there's no need to compare the sentinels at this point, because by then we have copied all the "real" elements (the non-sentinels) back to \( A[p .. r] \). Since we know in advance that we'll be copying elements back into \( A[p] \) through \( A[r] \), we can stop once we have copied an element back into \( A[r] \). We can just run a loop with an index into \( A \) running from \( p \) to \( r \).

Here is the MERGE procedure. It looks long, but it just follows the method above.

---

### Procedure MERGE\((A, p, q, r)\)

**Inputs:**

- \( A \): an array.
- \( p, q, r \): indices into \( A \). Each of the subarrays \( A[p .. q] \) and \( A[q + 1 .. r] \) is assumed to be already sorted.

**Result:** The subarray \( A[p .. r] \) contains the elements originally in \( A[p .. q] \) and \( A[q + 1 .. r] \), but now the entire subarray \( A[p .. r] \) is sorted.

1. Set \( n_1 \) to \( q - p + 1 \), and set \( n_2 \) to \( r - q \).
2. Let \( B[1 .. n_1 + 1] \) and \( C[1 .. n_2 + 1] \) be new arrays.
3. Copy \( A[p .. q] \) into \( B[1 .. n_1] \), and copy \( A[q + 1 .. r] \) into \( C[1 .. n_2] \).
4. Set both \( B[n_1 + 1] \) and \( C[n_2 + 1] \) to \( \infty \).
5. Set both \( i \) and \( j \) to 1.
6. For \( k = p \) to \( r \):
   
   **A.** If \( B[i] \leq C[j] \), then set \( A[k] \) to \( B[i] \) and increment \( i \).
   
   **B.** Otherwise (\( B[i] > C[j] \)), set \( A[k] \) to \( C[j] \) and increment \( j \).

After steps 1–4 allocate the arrays \( B \) and \( C \), copy \( A[p .. q] \) into \( B \) and \( A[q + 1 .. r] \) into \( C \), and insert the sentinels into these arrays, each

---

3 Actually, it is possible to merge in place in linear time, but the procedure to do so is pretty complicated.

4 In practice, we represent \( \infty \) by a value that compares as greater than any sort key. For example, if the sort keys are author names, \( \infty \) could be ZZZZ—assuming, of course, that no real author has that name.
iteration of the main loop in step 6 copies back the smallest remaining element to the next position in \(A[p \ldots r]\), terminating once it has copied back all the elements in \(B\) and \(C\). In this loop, \(i\) indexes the smallest remaining element in \(B\), \(j\) indexes the smallest remaining element in \(C\), and \(k\) indexes the location in \(A\) where the element will be copied back into.

If we are merging \(n\) elements altogether (so that \(n = n_1 + n_2\)), it takes \(\Theta(n)\) time to copy the elements into arrays \(B\) and \(C\), and constant time per element to copy it back into \(A[p \ldots r]\), for a total merging time of only \(\Theta(n)\).

We claimed earlier that the entire merge-sort algorithm takes time \(\Theta(n \lg n)\). We will make the simplifying assumption that the array size \(n\) is a power of 2, so that every time we divide the array, the subarray sizes are equal. (In general, \(n\) might not be a power of 2 and so the subarray sizes might not be equal in a given recursive call. A rigorous analysis can take care of this technicality, but let's not concern ourselves with it.)

Here is how we analyze merge sort. Let's say that sorting a subarray of \(n\) elements takes time \(T(n)\), which is a function that increases with \(n\) (since, presumably, it takes longer to sort more elements). The time \(T(n)\) comes from the three components of the divide-and-conquer paradigm, whose times we add together:

1. Dividing takes constant time, because it amounts to just computing the index \(q\).
2. Conquering consists of the two recursive calls on subarrays, each with \(n/2\) elements. By how we defined the time to sort a subarray, each of the two recursive calls takes time \(T(n/2)\).
3. Combining the results of the two recursive calls by merging the sorted subarrays takes \(\Theta(n)\) time.

Because the constant time for dividing is a low-order term compared with the \(\Theta(n)\) time for combining, we can absorb the dividing time into the combining time and say that dividing and combining, together, take \(\Theta(n)\) time. The conquer step costs \(T(n/2) + T(n/2)\), or \(2T(n/2)\). Now we can write an equation for \(T(n)\):

\[
T(n) = 2T(n/2) + f(n),
\]

where \(f(n)\) represents the time for dividing and combining which, as we just noted, is \(\Theta(n)\). A common practice in the study of algorithms is to just put the asymptotic notation right into the equation and let it stand for some function that we don't care to give a name to, and so we rewrite this equation as

\[
T(n) = 2T(n/2) + \Theta(n).
\]

Wait! There seems to be something amiss here. We have defined the function \(T\) that describes the running time of merge sort in terms of the very same function! We call such an equation a recurrence equation, or just a recurrence. The problem is that we want to express \(T(n)\) in a non-recursive manner, that is, not in terms of itself. It can be a real pain in the neck to convert a function expressed as a recurrence into non-recursive form, but for a broad class of recurrence equations we can apply a cookbook method known as the "master method." The master method applies to many (but not all) recurrences of the form

\[
T(n) = aT(n/b) + f(n),
\]

where \(a\) and \(b\) are positive integer constants. Fortunately, it applies to our merge-sort recurrence, and it gives the result that \(T(n)\) is \(\Theta(n \lg n)\).

This \(\Theta(n \lg n)\) running time applies to all cases of merge sort—best case, worst case, and all cases in between. Each element is copied \(\Theta(n \lg n)\) times. As you can see from examining the MERGE method, when it is called with \(p = 1\) and \(r = n\), it makes copies of all \(n\) elements, and so merge sort definitely does not run in place.

**Quicksort**

Like merge sort, quicksort uses the divide-and-conquer paradigm (and hence uses recursion). Quicksort uses divide-and-conquer in a slightly different way than merge sort, however. It has a couple of other significant differences from merge sort:

- Quicksort works in place.
- Quicksort's asymptotic running time differs between the worst case and the average case. In particular, quicksort's worst-case running time is \(\Theta(n^2)\), but its average-case running time is better: \(\Theta(n \lg n)\).

Quicksort also has good constant factors (better than merge sort's), and it is often a good sorting algorithm to use in practice.

Here is how quicksort uses divide-and-conquer. Again let us think about sorting books on a bookshelf. As with merge sort, we initially want to sort all \(n\) books in slots 1 through \(n\), and we'll consider the general problem of sorting books in slots \(p\) through \(r\).
Chapter 3: Algorithms for Sorting and Searching

1. **Divide** by first choosing any one book that is in slots \( p \) through \( r \). Call this book the **pivot**. Rearrange the books on the shelf so that all other books with author names that come before the pivot’s author or are written by the same author are to the left of the pivot, and all books with author names that come after the pivot’s author are to the right of the pivot.

In this example, we choose the rightmost book, by Jack London, as the pivot when rearranging the books in slots 9 through 15:

![Bookshelf rearrangement](image)

After rearranging—which we call **partitioning** in quicksort—the books by Flaubert and Dickens, who come before London alphabetically, are to the left of the book by London, and all other books, by authors who come after London alphabetically, are to the right. Notice that after partitioning, the books to the left of the book by London are in no particular order, and the same is true for the books to the right.

2. **Conquer** by recursively sorting the books to the left of the pivot and to the right of the pivot. That is, if the divide step moves the pivot to slot \( q \) (slot 11 in the above example), then recursively sort the books in slots \( p \) through \( q - 1 \) and recursively sort the books in slots \( q + 1 \) through \( r \).

3. **Combine**—by doing nothing! Once the conquer step recursively sorts, we are done. Why? All the books to the left of the pivot (in slots \( p \) through \( q - 1 \)) come before the pivot or have the same author as the pivot and are sorted, and all the books to the right of the pivot (in slots \( q + 1 \) through \( r \)) come after the pivot and are sorted. The books in slots \( p \) through \( r \) can’t help but be sorted!

If you change the bookshelf to the array and the books to array elements, you have the strategy for quicksort. Like merge sort, the base case occurs when the subarray to be sorted has fewer than two elements.

The procedure for quicksort assumes that we can call a procedure \( \text{PARTITION}(A, p, r) \) that partitions the subarray \( A[p...r] \), returning the index \( q \) where it has placed the pivot.

**Procedure QUICKSORT**\( (A, p, r) \)

*Inputs and Result:* Same as \( \text{MERGE-SORT} \).

1. If \( p \geq r \), then just return without doing anything.
2. Otherwise, do the following:
   A. Call \( \text{PARTITION}(A, p, r) \), and set \( q \) to its result.
   B. Recursively call \( \text{QUICKSORT}(A, p, q - 1) \).
   C. Recursively call \( \text{QUICKSORT}(A, q + 1, r) \).

The initial call is \( \text{QUICKSORT}(A, 1, n) \), similar to the \( \text{MERGE-SORT} \) procedure. Here's an example of how the recursion unfolds, with the indices \( p, q, \) and \( r \) shown for each subarray in which \( p \leq r \):
The bottommost value in each array position gives the final element stored there. When you read the array from left to right, looking at the bottommost value in each position, you see that the array is indeed sorted.

The key to quicksort is partitioning. Just as we were able to merge \( n \) elements in \( \Theta(n) \) time, we can partition \( n \) elements in \( \Theta(n) \) time. Here’s how we’ll partition the books that are in slots \( p \) through \( r \) on the shelf. We choose the rightmost book of the set—the book in slot \( r \)—as the pivot. At any time, each book will be in exactly one of four groups, and these groups will be in slots \( p \) through \( r \), from left to right:

- **group L (left group):** books with authors known to come before the pivot’s author alphabetically or written by the pivot’s author, followed by
- **group R (right group):** books with authors known to come after the pivot’s author alphabetically, followed by
- **group U (unknown group):** books that we have not yet examined, so we don’t know how their authors compare with the pivot’s author, followed by
- **group P (pivot):** just one book, the pivot.

We go through the books in group \( U \) from left to right, comparing each with the pivot and moving it into either group \( L \) or group \( R \), stopping once we get to the pivot. The book we compare with the pivot is always the leftmost book in group \( U \).

- If the book’s author comes after the pivot’s author, then the book becomes the rightmost book in group \( R \). Since the book was the leftmost book in group \( U \), and group \( U \) immediately follows group \( R \), we just have to move the dividing line between groups \( R \) and \( U \) one slot to the right, without moving any books:

Once we get to the pivot, we swap it with the leftmost book in group \( R \). In our example, we end up with the arrangement of books shown on page 50.

We compare each book with the pivot once, and each book whose author comes before the pivot’s author or is the pivot’s author causes one swap to occur. To partition \( n \) books, therefore, we make at most \( n - 1 \) comparisons (since we don’t have to compare the pivot with itself) and at most \( n \) swaps. Notice that, unlike merging, we can partition the books without removing them all from the shelf. That is, we can partition in place.

To convert how we partition books to how we partition a subarray \( A[p \ldots r] \), we first choose \( A[r] \) (the rightmost element) as the pivot. Then we go through the subarray from left to right, comparing each element with the pivot. We maintain indices \( q \) and \( u \) into the subarray that divide it up as follows:

- The subarray \( A[p \ldots q - 1] \) corresponds to group \( L \): each element is less than or equal to the pivot.
- The subarray \( A[q \ldots u - 1] \) corresponds to group \( R \): each element is greater than the pivot.
- The subarray \( A[u \ldots r - 1] \) corresponds to group \( U \): we don’t yet know how they compare with the pivot.
- The element \( A[r] \) corresponds to group \( P \): it holds the pivot.

These divisions, in fact, are loop invariants. (But we won’t prove them.)
At each step, we compare $A[u]$, the leftmost element in group U, with the pivot. If $A[u]$ is greater than the pivot, then we increment $u$ to move the dividing line between groups R and U to the right. If instead $A[u]$ is less than or equal to the pivot, then we swap the elements in $A[q]$ (the leftmost element in group R) and $A[u]$ and then increment both $q$ and $u$ to move the dividing lines between groups L and R and groups R and U to the right. Here’s the PARTITION procedure:

Procedure PARTITION(A, p, r)
Inputs: Same as MERGE-SORT.
Result: Rearranges the elements of $A[p .. r]$ so that every element in $A[p .. q]$ is less than or equal to $A[q]$ and every element in $A[q + 1 .. r]$ is greater than $q$. Returns the index $q$ to the caller.

1. Set $q$ to $p$.
2. For $u = p$ to $r - 1$ do:

By starting both of the indices $q$ and $u$ at $p$, groups L ($A[p .. q - 1]$) and R ($A[q .. u - 1]$) are initially empty and group U ($A[u .. r - 1]$) contains every element except the pivot. In some instances, such as if $A[p] \leq A[r]$, an element might be swapped with itself, resulting in no change to the array. Step 3 finishes up by swapping the pivot element with the leftmost element in group R, thereby moving the pivot into its correct place in the partitioned array, and then returning the pivot’s new index $q$.

Here is how the PARTITION procedure operates, step by step, on the subarray $A[5 .. 10]$ created by the first partitioning in the quicksort example on page 52. Group U is shown in white, group L has light shading, group R has darker shading, and the darkest element is the pivot, group P. The first part of the figure shows the initial array and indices, the next five parts show the array and indices after each iteration of the loop of step 2 (including incrementing index $u$ at the end of each iteration), and the last part shows the final partitioned array:

As when we partitioned books, we compare each element with the pivot once and perform at most one swap for each element that we compare with the pivot. Since each comparison takes constant time and each swap takes constant time, the total time for PARTITION on an $n$-element subarray is $\Theta(n)$.

So how long does the QUICKSORT procedure take? As with merge sort, let’s say that sorting a subarray of $n$ elements takes time $T(n)$, a function that increases with $n$. Dividing, done by the PARTITION procedure, takes $\Theta(n)$ time. But the time for QUICKSORT depends on how even the partitioning turns out to be.

In the worst case, the partition sizes are really unbalanced. If every element other than the pivot is less than it, then PARTITION ends up leaving the pivot in $A[r]$ and returns the index $r$ to QUICKSORT, which QUICKSORT stores in the variable $q$. In this case, the partition $A[q + 1 .. r]$ is empty and the partition $A[p .. q - 1]$ is only one element smaller than $A[p .. r]$. The recursive call on the empty subarray takes $\Theta(1)$ time (the time to make the call and determine that the subarray is empty in step 1). We can just roll this $\Theta(1)$ into the $\Theta(n)$ time for partitioning. But if $A[p .. r]$ has $n$ elements, then $A[p .. q - 1]$ has $n - 1$ elements, and so the recursive call on $A[p .. q - 1]$ takes $T(n - 1)$ time. We get the recurrence

$$T(n) = T(n - 1) + \Theta(n).$$

We can’t solve this recurrence using the master method, but it has the solution that $T(n) = \Theta(n^2)$. That’s no better than selection sort! How can we get such an uneven split? If every pivot is greater than all other elements, then the array must have started out already sorted. It also
turns out that we get an uneven split every time if the array starts out in reverse sorted order.

On the other hand, if we got an even split every time, each of the subarrays would have at most \( n/2 \) elements. The recurrence would be the same as the recurrence on page 49 for merge sort,

\[
T(n) = 2T(n/2) + \Theta(n),
\]

with the same solution, \( T(n) = \Theta(n \log n) \). Of course, we’d have to be really lucky, or the input array would have to be contrived, to get a perfectly even split every time.

The usual case is somewhere between the best and worst cases. The technical analysis is messy and I won’t put you through it, but if the elements of the input array come in a random order, then on average we get splits that are close enough to even that QUICKSORT takes \( \Theta(n \log n) \) time.

Now let’s get paranoid. Suppose that your worst enemy has given you an array to sort, knowing that you always pick the last element in each subarray as the pivot, and has arranged the array so that you always get the worst-case split. How can you foil your enemy? You could first check to see whether the array starts out sorted or reverse sorted, and do something special in these cases. Then again, your enemy could contrive the array so that the splits are always bad, but not maximally bad. You wouldn’t want to check for every possible bad case.

Fortunately, there’s a much simpler solution: don’t always pick the last element as the pivot. But then the lovely PARTITION procedure won’t work, because the groups aren’t where they’re supposed to be. That’s not a problem, either: before running the PARTITION procedure, swap \( A[r] \) with a randomly chosen element in \( A[p \ldots r] \). Now you’ve chosen your pivot randomly and you can run the PARTITION procedure.

In fact, with a little more effort, you can improve your chance of getting a split that’s close to even. Instead of choosing one element in \( A[p \ldots r] \) at random, choose three elements at random and swap the median of the three with \( A[r] \). By the median of the three, we mean the one whose value is between the other two. (If two or more of the randomly chosen elements are equal, break ties arbitrarily.) Again, I won’t make you endure the analysis, but you would have to be really unlucky in how you chose the random elements each time in order for QUICKSORT to take longer than \( \Theta(n \log n) \) time. Moreover, unless your enemy had access to your random number generator, your enemy would have no control over how even the splits turn out to be.

How many times does QUICKSORT swap elements? That depends on whether you count “swapping” an element to the same position it started in as a swap. You could certainly check to see whether this is the case and avoid the swap if it is. So let’s call it a swap only when an element really moves in the array as a result of swapping, that is, when \( q \neq r \) in step 2A or when \( q \neq r \) in step 3 of PARTITION. The best case for minimizing swaps is also one of the worst cases for asymptotic running time: when the array is already sorted. Then no swaps occur. The most swaps occur when \( n \) is even and the input array looks like \( n, n-2, n-4, \ldots, 4, 2, 1, 3, 5, \ldots, n-3, n-1 \). Then \( n^2/4 \) swaps occur, and the asymptotic running time is still the worst case \( \Theta(n^2) \).

Recap

In this chapter and the previous one, we have seen four algorithms for searching and four for sorting. Let’s summarize their properties in a couple of tables. Because the three searching algorithms from Chapter 2 were just variations on a theme, we can consider either BETTER-LINEAR-SEARCH or SENTINEL-LINEAR-SEARCH as the representatives for linear search.

**Searching algorithms**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Worst-case running time</th>
<th>Best-case running time</th>
<th>Requires sorted array?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear search</td>
<td>( \Theta(n) )</td>
<td>( \Theta(1) )</td>
<td>no</td>
</tr>
<tr>
<td>Binary search</td>
<td>( \Theta(n \log n) )</td>
<td>( \Theta(1) )</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Sorting algorithms**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Worst-case running time</th>
<th>Best-case running time</th>
<th>Worst-case swaps</th>
<th>In-place?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection sort</td>
<td>( \Theta(n^2) )</td>
<td>( \Theta(n^2) )</td>
<td>( \Theta(n) )</td>
<td>yes</td>
</tr>
<tr>
<td>Insertion sort</td>
<td>( \Theta(n^2) )</td>
<td>( \Theta(n) )</td>
<td>( \Theta(n^2) )</td>
<td>yes</td>
</tr>
<tr>
<td>Merge sort</td>
<td>( \Theta(n \log n) )</td>
<td>( \Theta(n \log n) )</td>
<td>( \Theta(n \log n) )</td>
<td>no</td>
</tr>
<tr>
<td>Quick sort</td>
<td>( \Theta(n^2) )</td>
<td>( \Theta(n \log n) )</td>
<td>( \Theta(n^2) )</td>
<td>yes</td>
</tr>
</tbody>
</table>

These tables do not show average-case running times, because with the notable exception of quicksort, they match the worst-case running times. As we saw, quicksort’s average-case running time, assuming that the array starts in a random order, is only \( \Theta(n \log n) \).
How do these sorting algorithms compare in practice? I coded them up in C++ and ran them on arrays of 4-byte integers on two different machines: my MacBook Pro (on which I wrote this book), with a 2.4-GHz Intel Core 2 Duo processor and 4 GB of RAM running Mac OS 10.6.8, and a Dell PC (my website server) with a 3.2-GHz Intel Pentium 4 processor and 1 GB of RAM running Linux version 2.6.22.14. I compiled the code with g++ and optimization level -03. I ran each algorithm on array sizes ranging up to 50,000, with each array initially in reverse sorted order. I averaged the running times for 20 runs of each algorithm on each array size.

By starting each array in reverse sorted order, I elicited the worst-case asymptotic running times of both insertion sort and quicksort. Therefore, I ran two versions of quicksort: "regular" quicksort, which always chooses the pivot as the last element $A[r]$ of the subarray $A[p..r]$ being partitioned, and randomized quicksort, which swaps a randomly chosen element in $A[p..r]$ with $A[r]$ before partitioning. (I did not run the median-of-three method.) The "regular" version of quicksort is also known as deterministic because it does not randomize; everything it does is predetermined once it's given an input array to sort.

Randomized quicksort was the champion for $n \geq 64$ on both computers. Here are the ratios of the running times of the other algorithms to randomized quicksort's running times on various input sizes:

<table>
<thead>
<tr>
<th>Machine</th>
<th>50</th>
<th>100</th>
<th>500</th>
<th>1000</th>
<th>5000</th>
<th>10,000</th>
<th>50,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>MacBook Pro</td>
<td>0.55</td>
<td>0.56</td>
<td>0.60</td>
<td>0.60</td>
<td>0.62</td>
<td>0.63</td>
<td>0.66</td>
</tr>
<tr>
<td>Dell PC</td>
<td>0.53</td>
<td>0.58</td>
<td>0.60</td>
<td>0.58</td>
<td>0.60</td>
<td>0.64</td>
<td>0.64</td>
</tr>
</tbody>
</table>

Is it possible to beat $\Theta(n \lg n)$ time for sorting? It depends. We'll see in Chapter 4 that if the only way that we can determine where to place elements is by comparing elements, doing different things based on the results of the comparisons, then no, we cannot beat $\Theta(n \lg n)$ time. If we know something about the elements that we can take advantage of, however, we can do better.

Further reading

CLRS [CLRS09] covers insertion sort, merge sort, and both deterministic and randomized quicksort. But the granddaddy of books about sorting and searching remains Volume 3 of Knuth's *The Art of Computer Programming* [Knu98b]; the advice from Chapter 1 applies—TAOCP is deep and intense.