37. $2 + 2\cdot 3^2 + 2\cdot 3^4 + \cdots + 2\cdot 3^{2n}$ is $\Theta(3^{2n})$.

38. $\dfrac{1}{5} + \dfrac{4}{5^2} + \dfrac{4^2}{5^3} + \cdots + \dfrac{4^n}{5^{n+1}}$ is $\Theta(1)$.

39. $n + \dfrac{n}{2} + \dfrac{n}{4} + \cdots + \dfrac{n}{2^n}$ is $\Theta(n)$.

40. $\dfrac{2n}{3} + \dfrac{2n}{3^2} + \dfrac{2n}{3^3} + \cdots + \dfrac{2n}{3^n}$ is $\Theta(n)$.

41. Quantities of the form

$$kn + kn\log_2 n \quad \text{for positive integers } k_1 \cdot k_2, \text{ and } n$$

arise in the analysis of the merge sort algorithm in computer science. Show that for any positive integer k,

$$k_1 n + k_2 n \log_2 n \quad \text{is} \quad \Theta(n\log_2 n).$$

42. Calculate the values of the harmonic sums

$$1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \quad \text{for } n = 2, 3, 4, \text{ and } 5.$$

43. Use part (d) of Example 11.4.7 to show that

$$n + \frac{n}{2} + \frac{n}{3} + \cdots + \frac{n}{n} \quad \text{is} \quad \Theta(n\ln n).$$

44. Use the fact that $\log_2 x = \left(\dfrac{1}{\log_e 2}\right)\log_e x$ and $\log_e x = \ln x$, for all positive numbers x, and part (c) of Example 11.4.7 to show that

$$1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \quad \text{is} \quad \Theta(\log_2 n).$$

45. a. Show that $\lfloor \log_2 n \rfloor$ is $\Theta(\log_2 n)$.
b. Show that $\lfloor \log_2 n \rfloor + 1$ is $\Theta(\log_2 n)$.

46. Prove by mathematical induction that $n \le 10^n$ for all integers $n \ge 1$.

H **47.** Prove by mathematical induction that $\log_2 n \le n$ for all integers $n \ge 1$.

H **48.** Show that if n is a variable that takes positive integer values, then 2^n is $O(n!)$.

49. Let n be a variable that takes positive integer values.
a. Show that $n!$ is $O(n^n)$.
b. Use part (a) to show that $\log_2(n!)$ is $O(n\log_2 n)$.
H **c.** Show that $n^n \le (n!)^2$ for all integers $n \ge 2$.
d. Use part (c) to show that $\log_2(n!)$ is $\Omega(n\log_2 n)$.
e. Use parts (b) and (d) to find an order for $\log_2(n!)$.

✶ **50.** a. For all positive real numbers u, $\log_2 u < u$. Use this fact to show that for any positive integer n, $\log_2 x < nx^{1/n}$ for all real numbers $x > 0$.
b. Interpret the statement of part (a) using O-notation.

51. a. For all real numbers x, $x < 2^x$. Use this fact to show that for any positive integer n, $x^n < n^n 2^x$ for all real numbers $x > 0$.
b. Interpret the statement of part (a) using O-notation.

✶ **52.** For all positive real numbers u, $\log_2 u < u$. Use this fact and the result of exercise 21 in Section 11.1 to prove the following: For all integers $n \ge 1$, $\log_2 x < x^{1/n}$ for all real numbers $x > (2n)^{2n}$.

53. Use the result of exercise 52 above to prove the following: For all integers $n \ge 1$, $x^n < 2^x$ for all real numbers $x > (2n)^{2n}$.

Exercises 54 and 55 use L'Hôpital's rule from calculus.

54. a. Let b be any real number greater than 1. Use L'Hôpital's rule and mathematical induction to prove that for all integers $n \ge 1$,

$$\lim_{x\to\infty} \frac{x^n}{b^x} = 0.$$

b. Use the result of part (a) and the definitions of limit and of O-notation to prove that x^n is $O(b^x)$ for any integer $n \ge 1$.

55. a. Let b be any real number greater than 1. Use L'Hôpital's rule to prove that for all integers $n \ge 1$,

$$\lim_{x\to\infty} \frac{\log_b x}{x^{1/n}} = 0.$$

b. Use the result of part (a) and the definitions of limit and of O-notation to prove that $\log_b x$ is $O(x^{1/n})$ for any integer $n \ge 1$.

56. Complete the proof in Example 11.4.4.

Answers for Test Yourself

1. the set of all real numbers; the set of all positive real numbers 2. the set of all positive real numbers; the set of all real numbers 3. k 4. $\log_b x < x < x\log_b x < x^2$ 5. $\ln x$ (or, equivalently, $\log_2 x$)

11.5 *Application: Analysis of Algorithm Efficiency II*

Pick a Number, Any Number — Donal O'Shea, 2007

Have you ever played the "guess my number" game? A person thinks of a number between two other numbers, say 1 and 10 or 1 and 100 for example, and you try to figure out what it is, using the least possible number of guesses. Each time you guess a number, the person tells you whether you are correct, too low, or too high.

If you have played this game, you have probably already hit upon the most efficient strategy: Begin by guessing a number as close to the middle of the two given numbers as possible. If your guess is too high, then the number is between the lower of the two given numbers and the one you first chose. If your guess is too low, then the number is between the number you first chose and the higher of the two given numbers. In either case, you take as your next guess a number as close as possible to the middle of the new range in which you now know the number lies. You repeat this process as many times as necessary until you have found the person's number.

The technique described previously is an example of a general strategy called **divide and conquer,** which works as follows: To solve a problem, reduce it to a fixed number of smaller problems of the same kind, which can themselves be reduced to the same fixed number of smaller problems of the same kind, and so forth until easily resolved problems are obtained. In this case, the problem of finding a particular number in a given range of numbers is reduced at each stage to finding a particular number in a range of numbers approximately half as long.

It turns out that algorithms using a divide-and-conquer strategy are generally quite efficient and nearly always have orders involving logarithmic functions. In this section we define the *binary search* algorithm, which is the formalization of the "guess my number" game described previously, and we compare the efficiency of binary search to the sequential search discussed in Section 11.3. Then we develop a divide-and-conquer algorithm for sorting, *merge sort,* and compare its efficiency with that of insertion sort and selection sort, which were also discussed in Section 11.3.

Binary Search

Whereas a sequential search can be performed on an array whose elements are in any order, a binary search can be performed only on an array whose elements are arranged in ascending (or descending) order. Given an array $a[1], a[2], \ldots, a[n]$ of distinct elements arranged in ascending order, consider the problem of trying to find a particular element x in the array.

To use binary search, first compare x to the "middle element" of the array. If the two are equal, the search is successful. If the two are not equal, then because the array elements are in ascending order, comparing the values of x and the middle array element narrows the search either to the lower subarray (consisting of all the array elements below the middle element) or to the upper subarray (consisting of all array elements above the middle element).

The search continues by repeating this basic process over and over on successively smaller subarrays. It terminates either when a match occurs or when the subarray to which the search has been narrowed contains no elements. The efficiency of the algorithm is a result of the fact that at each step, the length of the subarray to be searched is roughly half the length of the array of the previous step. This process is illustrated in Figure 11.5.1.

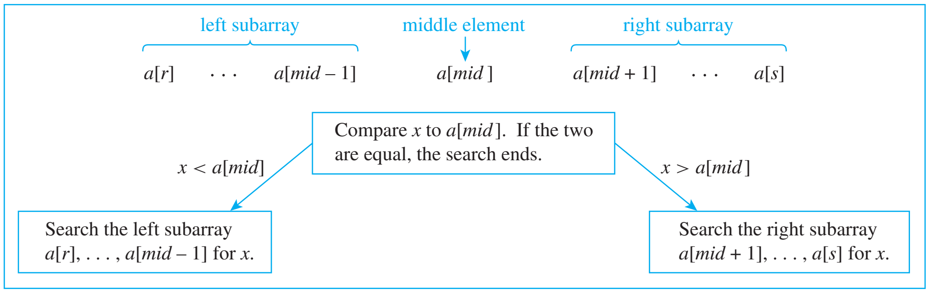
left subarray
middle element
right subarray
a[r] . . . a[mid – 1]
a[mid]
a[mid + 1] . . . a[s]
Compare x to a[mid]. If the two are equal, the search ends.
x < a[mid]
x > a[mid]
Search the left subarray a[r], . . . , a[mid – 1] for x.
Search the right subarray a[mid + 1], . . . , a[s] for x.

Figure 11.5.1 **One Iteration of the Binary Search Process**

To write down a formal algorithm for binary search, we introduce a variable *index* whose final value will tell us whether or not x is in the array and, if so, will indicate the location of x. Since the array goes from $a[1]$ to $a[n]$, we intialize *index* to be 0. If and when x is found, the value of *index* is changed to the subscript of the array element equaling x. If index still has the value 0 when the algorithm is complete, then x is not one of the elements in the array. Figure 11.5.2 shows the action of a particular binary search.

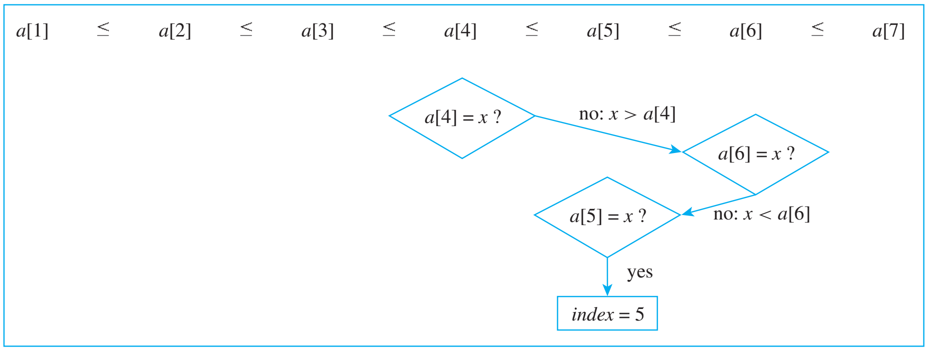

Figure 11.5.2 Binary Search of $a[1], a[2], \ldots, a[7]$ for x where $x = a[5]$

Formalizing a binary search algorithm also requires that we be more precise about the meaning of the "middle element" of an array. (This issue was side-stepped by careful choice of n in Figure 11.5.2.) If the array consists of an even number of elements, there are two elements in the middle. For instance, both $a[6]$ and $a[7]$ are equally in the middle of the following array.

$$\underbrace{a[3]\quad a[4]\quad a[5]}_{\text{three elements}}\quad \underbrace{a[6]\quad a[7]}_{\substack{\text{two middle}\\ \text{elements}}}\quad \underbrace{a[8]\quad a[9]\quad a[10]}_{\text{three elements}}$$

In a case such as this, the algorithm must choose which of the two middle elements to take, the smaller or the larger. The choice is arbitrary—either would do. We will write the algorithm to choose the smaller. The index of the smaller of the two middle elements is the floor of the average of the top and bottom indices of the array. That is, if

$$\begin{aligned} bot &= \text{the bottom index of the array,} \\ top &= \text{the top index of the array,} \quad \text{and} \\ mid &= \text{the lower of the two middle indices of the array,} \end{aligned}$$

then

$$mid = \left\lfloor \frac{bot + top}{2} \right\rfloor.$$

In this case, $bot = 3$ and $top = 10$, so the index of the "middle element" is

$$mid = \left\lfloor \frac{3 + 10}{2} \right\rfloor = \left\lfloor \frac{13}{2} \right\rfloor = \lfloor 6.5 \rfloor = 6.$$

The following is a formal algorithm for a binary search.

Algorithm 11.5.1 Binary Search

[The aim of this algorithm is to search for an element x in an ascending array of elements $a[1], a[2], \ldots, a[n]$*. If x is found, the variable index is set equal to the index of the array element where x is located. If x is not found, index is not changed from its initial value, which is* 0*. The variables bot and top denote the bottom and top indices of the array currently being examined.]*

Input: *n [a positive integer],* $a[1], a[2], \ldots, a[n]$ *[an array of data items given in ascending order], x [a data item of the same data type as the elements of the array]*

Algorithm Body:

```
$index := 0, bot := 1, top := n$
[Compute the middle index of the array, mid. Compare x to a[mid]. If the two are equal, the search is successful. If not, repeat the process either for the lower or for the upper subarray, either giving top the new value mid − 1 or giving bot the new value mid + 1. Each iteration of the loop either decreases the value of top or increases the value of bot. Thus, if the looping is not stopped by success in the search process, eventually the value of top will become less than the value of bot. This occurrence stops the looping process and indicates that x is not an element of the array.]
    while ($top \geq bot$ and $index = 0$)
        $mid := \left\lfloor \frac{bot + top}{2} \right\rfloor$
        if $a[mid] = x$ then $index := mid$
        if $a[mid] > x$
            then $top := mid - 1$
            else $bot := mid + 1$
    end while
    [If index has the value 0 at this point, then x is not in the array. Otherwise, index gives the index of the array where x is located.]
```

Output: *index [a nonnegative integer]*

Example 11.5.1 Tracing the Binary Search Algorithm

Trace the action of Algorithm 11.5.1 on the variables *index, bot, top, mid,* and the values of x given in (a) and (b) below for the input array

$$a[1] = \text{Ann}, a[2] = \text{Dawn}, a[3] = \text{Erik}, a[4] = \text{Gail}, a[5] = \text{Juan},$$
$$a[6] = \text{Matt}, a[7] = \text{Max}, a[8] = \text{Rita}, a[9] = \text{Tsuji}, a[10] = \text{Yuen}$$

where alphabetical ordering is used to compare elements of the array.

a. $x = \text{Max}$ b. $x = \text{Sara}$

Solution

a.

index	0				7
bot	1	6		7	
top	10		7		
mid		5	8	6	7

b.

index	0			
bot	1	6	9	
top	10			8
mid		5	8	9

■

The Efficiency of the Binary Search Algorithm

The idea of the derivation of the efficiency of the binary search algorithm is not difficult. Here it is in brief. At each stage of the binary search process, the length of the new subarray to be searched is approximately half that of the previous one, and in the worst case, every subarray down to a subarray with a single element must be searched. Consequently, in the worst case, the maximum number of iterations of the **while** loop in the binary search algorithm is 1 more than the number of times the original input array can be cut approximately in half. If the length n of this array is a power of 2 ($n = 2^k$ for some integer k), then n can be halved exactly $k = \log_2 n = \lfloor \log_2 n \rfloor$ times before an array of length 1 is reached. If n is not a power of 2, then $n = 2^k + m$ for some integer k (where $m < 2^k$), and so n can be split approximately in half k times also. So in this case, $k = \lfloor \log_2 n \rfloor$ also. Thus in the worst case, the number of iterations of the **while** loop in the binary search algorithm, which is proportional to the number of comparisons required to execute it, is $\lfloor \log_2 n \rfloor + 1$. The derivation is concluded by noting that $\lfloor \log_2 n \rfloor + 1$ is $O(\log_2 n)$.

The details of the derivation are developed in Examples 11.5.2–11.5.6. Throughout the derivation, for each integer $n \geq 1$, let

$$\boxed{\begin{aligned} w_n = {} & \text{the number of iterations of the \textbf{while} loop} \\ & \text{in a \textit{worst-case} execution of the binary search} \\ & \text{algorithm for an input array of length } n. \end{aligned}}$$

The first issue to consider is this. If the length of the input array for one iteration of the **while** loop is known, what is the greatest possible length of the array input to the next iteration?

Example 11.5.2 The Length of the Input Array to the Next Iteration of the Loop

Prove that if an array of length k is input to the **while** loop of the binary search algorithm, then after one unsuccessful iteration of the loop, the input to the next iteration is an array of length at most $\lfloor k/2 \rfloor$.

Solution Consider what occurs when an array of length k is input to the **while** loop in the case where $x \neq a[mid]$:

$$\underbrace{a[\text{bot}], a[\text{bot}+1], \ldots, a[\text{mid}-1]}_{\substack{\text{new input to the while} \\ \text{loop if } x < a[\text{mid}]}},\ \underset{\substack{\uparrow \\ \text{“middle} \\ \text{element”}}}{a[\text{mid}]},\ \underbrace{a[\text{mid}+1], \ldots, a[\text{top}-1], a[\text{top}]}_{\substack{\text{new input to the while} \\ \text{loop if } x > a[\text{mid}]}}.$$

Since the input array has length k, the value of *mid* depends on whether k is odd or even. In both cases we match up the array elements with the integers from 1 to k and analyze the lengths of the left and right subarrays. In case k is odd, both the left and the right subarrays have length $\lfloor k/2 \rfloor$. In case k is even, the left subarray has length $\lfloor k/2 \rfloor - 1$ and the right subarray has length $\lfloor k/2 \rfloor$. The reasoning behind these results is shown in Figure 11.5.3.

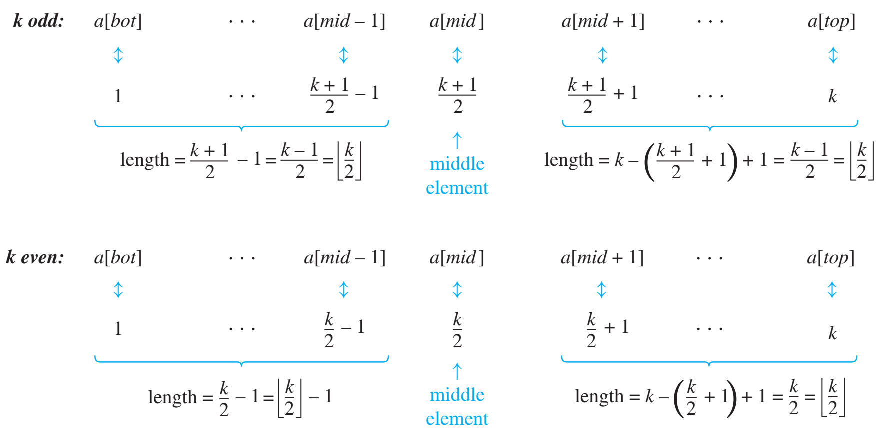
k odd:
a[bot] ... a[mid − 1] a[mid] a[mid + 1] ... a[top]
1 ... $\frac{k+1}{2} - 1$ $\frac{k+1}{2}$ $\frac{k+1}{2} + 1$ ... k
length = $\frac{k+1}{2} - 1 = \frac{k-1}{2} = \left\lfloor \frac{k}{2} \right\rfloor$
middle element
length = $k - \left(\frac{k+1}{2} + 1\right) + 1 = \frac{k-1}{2} = \left\lfloor \frac{k}{2} \right\rfloor$
k even:
a[bot] ... a[mid − 1] a[mid] a[mid + 1] ... a[top]
1 ... $\frac{k}{2} - 1$ $\frac{k}{2}$ $\frac{k}{2} + 1$ ... k
length = $\frac{k}{2} - 1 = \left\lfloor \frac{k}{2} \right\rfloor - 1$
middle element
length = $k - \left(\frac{k}{2} + 1\right) + 1 = \frac{k}{2} = \left\lfloor \frac{k}{2} \right\rfloor$

Figure 11.5.3 Lengths of the Left and Right Subarrays

Because the maximum of the numbers $\lfloor k/2 \rfloor$ and $\lfloor k/2 \rfloor - 1$ is $\lfloor k/2 \rfloor$, in the worst case this will be the length of the array input to the next iteration of the loop. ■

To find the order of the algorithm, a formula for $w_1, w_2, w_3, \ldots$ is needed. The next example derives a recurrence relation for the sequence.

Example 11.5.3 A Recurrence Relation for $w_1, w_2, w_3, \ldots$

Prove that the sequence $w_1, w_2, \ldots, w_n, \ldots$ satisfies the recurrence relation and initial condition

$$w_1 = 1,$$
$$w_k = 1 + w_{\lfloor k/2 \rfloor} \quad \text{for all integers } k > 1.$$

Solution Example 11.5.2 showed that given an input array of length k to the **while** loop, the worst that can happen is that the next iteration of the loop will have to search an array of length $\lfloor k/2 \rfloor$. Hence the maximum number of iterations of the loop is 1 more than the maximum number necessary to execute it for an input array of length $\lfloor k/2 \rfloor$. In symbols,

$$w_k = 1 + w_{\lfloor k/2 \rfloor}.$$

Also $$w_1 = 1$$

because for an input array of length 1 ($bot = top$), the **while** loop iterates only one time. ■

Now that a recurrence relation for $w_1, w_2, w_3, \ldots$ has been found, iteration can be used to come up with a good guess for an explicit formula.

Example 11.5.4 An Explicit Formula for $w_1, w_2, w_3, \ldots$

Apply iteration to the recurrence relation found in Example 11.5.3 to conjecture an explicit formula for $w_1, w_2, w_3, \ldots$.

Solution Begin by iterating to find the values of the first few terms of the sequence.

$$\begin{array}{ll}
\begin{array}{lcl} w_{\textcircled{1}} &=& \textcircled{1} \end{array} & 1 = 2^0;\ 1 = 0+1 \\
\left.\begin{array}{lcl}
w_{\textcircled{2}} &=& 1 + w_{\lfloor 2/2 \rfloor} = 1 + w_1 = 1 + 1 = \textcircled{2} \\
w_3 &=& 1 + w_{\lfloor 3/2 \rfloor} = 1 + w_1 = 1 + 1 = 2
\end{array}\right\} & 2 = 2^1;\ 2 = 1+1 \\
\left.\begin{array}{lcl}
w_{\textcircled{4}} &=& 1 + w_{\lfloor 4/2 \rfloor} = 1 + w_2 = 1 + 2 = \textcircled{3} \\
w_5 &=& 1 + w_{\lfloor 5/2 \rfloor} = 1 + w_2 = 1 + 2 = 3 \\
w_6 &=& 1 + w_{\lfloor 6/2 \rfloor} = 1 + w_3 = 1 + 2 = 3 \\
w_7 &=& 1 + w_{\lfloor 7/2 \rfloor} = 1 + w_3 = 1 + 2 = 3
\end{array}\right\} & 4 = 2^2;\ 3 = 2+1 \\
\left.\begin{array}{lcl}
w_{\textcircled{8}} &=& 1 + w_{\lfloor 8/2 \rfloor} = 1 + w_4 = 1 + 3 = \textcircled{4} \\
w_9 &=& 1 + w_{\lfloor 9/2 \rfloor} = 1 + w_4 = 1 + 3 = 4 \\
\vdots & & \qquad\qquad\qquad\qquad\qquad\quad \vdots \\
w_{15} &=& 1 + w_{\lfloor 15/2 \rfloor} = 1 + w_7 = 1 + 3 = 4
\end{array}\right\} & 8 = 2^3;\ 4 = 3+1 \\
\left.\begin{array}{lcl}
w_{\textcircled{16}} &=& 1 + w_{\lfloor 16/2 \rfloor} = 1 + w_8 = 1 + 4 = \textcircled{5} \\
\vdots & & \qquad\qquad\qquad\qquad\qquad\quad \vdots
\end{array}\right\} & 16 = 2^4;\ 5 = 4+1
\end{array}$$

Note that in each case when the subscript n is between two powers of 2, w_n is 1 more than the exponent of the lower power of 2. In other words:

$$\text{If } 2^i \le n < 2^{i+1}, \text{ then } w_n = i + 1. \qquad 11.5.1$$

But if

$$2^i \le n < 2^{i+1},$$

then *[by property (11.4.2) of Example 11.4.1]*

$$i = \lfloor \log_2 n \rfloor.$$

Substitution into statement (11.5.1) gives the conjecture that

$$w_n = \lfloor \log_2 n \rfloor + 1.$$

■

Now mathematical induction can be used to verify the correctness of the formula found in Example 11.5.4.

Example 11.5.5 Verifying the Correctness of the Formula

Use strong mathematical induction to show that if $w_1, w_2, w_3, \ldots$ is a sequence of numbers that satisfies the recurrence relation and initial condition

$$w_1 = 1 \qquad \text{and} \qquad w_k = 1 + w_{\lfloor k/2 \rfloor} \quad \text{for all integers } k > 1,$$

then $w_1, w_2, w_3, \ldots$ satisfies the formula

$$w_n = \lfloor \log_2 n \rfloor + 1 \quad \text{for all integers } n \ge 1.$$

Solution Let $w_1, w_2, w_3, \ldots$ be the sequence defined by specifying that $w_1 = 1$ and $w_k = 1 + w_{\lfloor k/2 \rfloor}$ for all integers $k \ge 2$, and let the property $P(n)$ be the equation

$$w_n = \lfloor \log_2 n \rfloor + 1. \qquad \leftarrow P(n)$$

We will use mathematical induction to prove that for all integers $n \ge 1$, $P(n)$ is true.

Show that P(1) is true: By definition of $w_1, w_2, w_3, \ldots$, we have that $w_1 = 1$. But it is also the case that $\lfloor \log_2 1 \rfloor + 1 = 0 + 1 = 1$. Thus $w_1 = \lfloor \log_2 1 \rfloor + 1$ and $P(1)$ is true.

***Show that for all integers k ≥ 1, if P(i) is true for all integers i from* 1 *through k, then P(k* + 1) *is also true*:** Let k be any integer with $k \geq 1$, and suppose that

$$w_i = \lfloor \log_2 i \rfloor + 1 \quad \text{for all integers } i \text{ with } 1 \leq i \leq k. \qquad \leftarrow \text{inductive hypothesis}$$

We must show that

$$w_{k+1} = \lfloor \log_2 (k+1) \rfloor + 1 \qquad \leftarrow P(k+1)$$

Consider the two cases: k is even and k is odd.

Case 1 (k is even): In this case, $k + 1$ is odd, and

$$\begin{aligned}
w_{k+1} &= 1 + w_{\lfloor (k+1)/2 \rfloor} && \text{by definition of } w_1, w_2, w_3, \ldots \\
&= 1 + w_{\lfloor k/2 \rfloor} && \text{because } \lfloor (k+1)/2 \rfloor = \lfloor k/2 \rfloor \text{ since } k+1 \text{ is odd} \\
&= 1 + \big(\lfloor \log_2 (k/2) \rfloor + 1\big) && \text{by inductive hypothesis because, since } k \text{ is even, } k \geq 2, \text{ and so } 1 \leq \lfloor k/2 \rfloor \leq k/2 < k \\
&= \lfloor \log_2 (k) - \log_2 2 \rfloor + 2 && \text{by substituting into the identity } \log_b (x/y) = \log_b x - \log_b y \text{ from Theorem 7.2.1} \\
&= \lfloor \log_2 (k) - 1 \rfloor + 2 && \text{since } \log_2 2 = 1 \\
&= (\lfloor \log_2 (k) \rfloor - 1) + 2 && \text{by substituting } x = \log_2 (k) \text{ into the identity } \lfloor x - 1 \rfloor = \lfloor x \rfloor - 1 \text{ derived in exercise 15 of Section 4.5} \\
&= \lfloor \log_2 (k+1) \rfloor + 1 && \text{by property (11.4.3) in Example 11.4.2}
\end{aligned}$$

Case 2 (k is odd): In this case, it can also be shown that $w_k = \lfloor \log_2 k \rfloor + 1$. The analysis is very similar to that of case 1 and is left as exercise 16 at the end of the section.

Hence regardless of whether k is even or k is odd,

$$w_{k+1} = \lfloor \log_2 (k+1) \rfloor + 1,$$

as was to be shown. *[Since both the basis and the inductive steps have been demonstrated, the proof by strong mathematical induction is complete.]* ■

The final example shows how to use the formula for $w_1, w_2, w_3, \ldots$ to find a worst-case order for the algorithm.

Example 11.5.6 The Binary Search Algorithm Is Logarithmic

Given that by Example 11.5.5, for all positive integers n,

$$w_n = \lfloor \log_2 n \rfloor + 1,$$

show that in the worst case, the binary search algorithm is $\Theta(\log_2 n)$.

Solution For any integer $n > 2$,

$$\begin{aligned}
& w_n = \lfloor \log_2 n \rfloor + 1 && \text{by Example 11.5.5} \\
\Rightarrow\ & \log_2 n \leq w_n \leq \log_2 n + 1 && \text{because } x < \lfloor x \rfloor + 1 \text{ and } \lfloor x \rfloor \leq x \text{ for all real numbers } x \\
\Rightarrow\ & \log_2 n \leq w_n \leq \log_2 n + \log_2 n && \text{since the logorithm with base 2 is increasing, if } 2 < n, \text{ then } 1 = \log_2 2 < \log_2 n \\
\Rightarrow\ & \log_2 n \leq w_n \leq 2 \log_2 n. &&
\end{aligned}$$

Both w_n and $\log_2 n$ are positive for $n > 2$. Therefore,

$$|\log_2 n| \le |w_n| \le 2|\log_2 n| \quad \text{for all integers } n > 2.$$

Let $A = 1$, $B = 2$, and $k = 2$. Then

$$A|\log_2 n| \le |w_n| \le B|\log_2 n| \quad \text{for all integers } n > k.$$

Hence by definition of Θ-notation,

$$w_n \quad \text{is} \quad \Theta(\log_2 n).$$

But w_n, the number of iterations of the **while** loop, is proportional to the number of comparisons performed when the binary search algorithm is executed. Thus the binary search algorithm is $\Theta(\log_2 n)$. ■

Examples 11.5.2–11.5.6 show that in the worst case, the binary search algorithm has order $\log_2 n$. As noted in Section 11.3, in the worst case the sequential search algorithm has order n. This difference in efficiency becomes increasingly more important as n gets larger and larger. Assuming one loop iteration is performed each nanosecond, then performing n iterations for $n = 100{,}000{,}000$ requires 0.1 second, whereas performing $\log_2 n$ iterations requires 0.000000027 second. For $n = 100{,}000{,}000{,}000$ the times are 1.67 minutes and 0.000000037 second, respectively. And for $n = 100{,}000{,}000{,}000{,}000$ the respective times are 27.78 hours and 0.000000047 second.

Merge Sort

Note that it is much easier to write a detailed algorithm for sequential search than for binary search. Yet binary search is much more efficient than sequential search. Such trade-offs often occur in computer science. Frequently, the straightforward "obvious" solution to a problem is less efficient than a clever solution that is more complicated to describe.

In the text and exercises for Section 11.3, we gave two methods for sorting, insertion sort and selection sort, both of which are formalizations of methods human beings often use in ordinary situations. Can a divide-and-conquer approach be used to find a sorting method more efficient than these? It turns out that the answer is an emphatic "yes." In fact, over the past few decades, computer scientists have developed several divide-and-conquer sorting methods all of which are somewhat more complex to describe but are significantly more efficient than either insertion sort or selection sort.

One of these methods, **merge sort,** is obtained by thinking recursively. Imagine that an efficient way for sorting arrays of length less than k is already known. How can such knowledge be used to sort an array of length k? One way is to suppose the array of length k is split into two roughly equal parts and each part is sorted using the known method. Is there an efficient way to combine the parts into a sorted array? Sure. Just "merge" them.

Figure 11.5.4 illustrates how a merge works. Imagine that the elements of two ordered subarrays, 2, 5, 6, 8 and 3, 6, 7, 9, are written on slips of paper (to make them easy to move around). Place the slips for each subarray in two columns on a tabletop, one at the left and one at the right. Along the bottom of the tabletop, set up eight positions into which the slips will be moved. Then, one-by-one, bring down the slips from the bottoms of the columns. At each stage compare the numbers on the slips currently at the column bottoms, and move the slip containing the smaller number down into the next position in the array as a whole. If at any stage the two numbers are equal, take, say, the slip on the left to move into the next position. And if one of the columns is empty at any stage, just move the slips from the other column into position one-by-one in order.

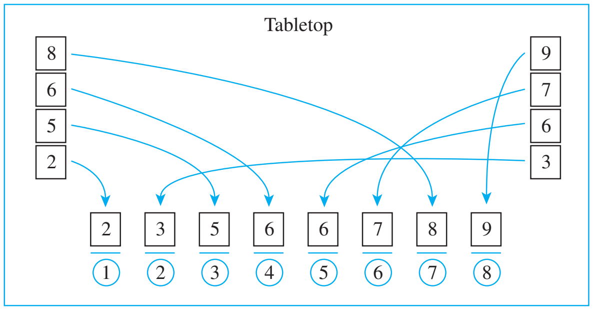

Figure 11.5.4 Merging Two Sorted Subarrays to Obtain a Sorted Array

One important observation about the merging algorithm described previously: It requires memory space to move the array elements around. A second set of array positions as long as the original one is needed into which to place the elements of the two subarrays in order. In Figure 11.5.4 this second set of array positions is represented by the positions set up at the bottom of the tabletop. Of course, once the elements of the original array have been placed into this new array, they can be moved back in order into the original array positions.

In terms of time, however, merging is efficient because the total number of comparisons needed to merge two subarrays into an array of length k is just $k - 1$. You can see why by analyzing Figure 11.5.4. Observe that at each stage, the decision about which slip to move is made by comparing the numbers on the slips currently at the bottoms of the two columns—execpt when one of the columns is empty, in which case no comparisons are made at all. Thus in the worst case there will be one comparison for each of the k positions in the final array except the very last one (because when the last slip is placed into position, the other column is sure to be empty), or a total of $k - 1$ comparisons in all.

The merge sort algorithm is recursive: Its defining statements include references to itself. The algorithm is well defined, however, because at each stage the length of the array that is input to the algorithm is shorter than at the previous stage, so that, ultimately, the algorithm has to deal only with arrays of length 1, which are already sorted. Specifically, merge sort works as follows.

Given an array of elements that can be put into order, if the array consists of a single element, leave it as it is. It is already sorted. Otherwise:

1. Divide the array into two subarrays of as nearly equal length as possible.
2. Use merge sort to sort each subarray.
3. Merge the two subarrays together.

Figure 11.5.5 illustrates a merge sort in a particular case.

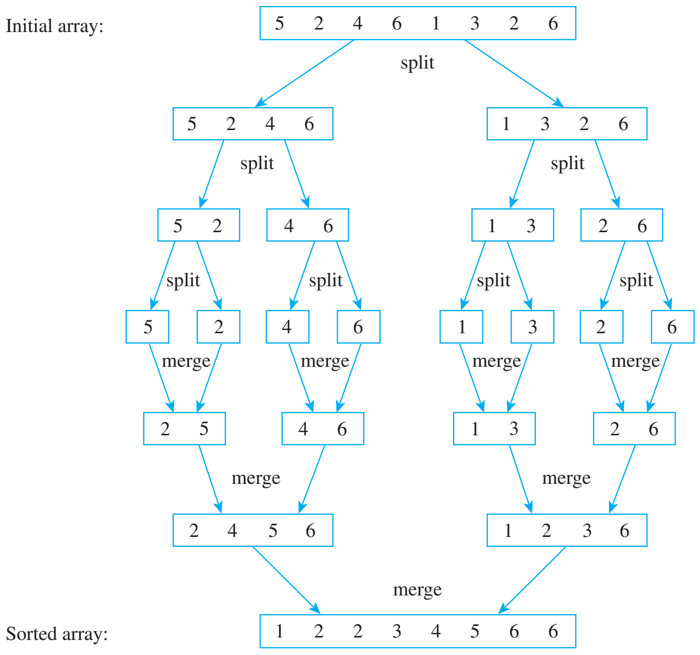

Figure 11.5.5 Applying Merge Sort to the Array 5, 2, 4, 6, 1, 3, 2, 6

As in the case of the binary search algorithm, in order to formalize merge sort we must decide at exactly what point to split each array. Given an array denoted by $a[bot]$, $a[bot + 1], \ldots, a[top]$, let $mid = \lfloor (bot + top)/2 \rfloor$. Take the left subarray to be $a[bot]$, $a[bot + 1], \ldots, a[mid]$ and the right subarray to be $a[mid + 1], a[mid + 2], \ldots, a[top]$. The following is a formal version of merge sort.

Algorithm 11.5.2 Merge Sort

[The aim of this algorithm is to take an array of elements $a[r], a[r + 1], \ldots, a[s]$ (where $r \le s$) and to order it. The output array is denoted $a[r], a[r + 1], \ldots, a[s]$ also. It has the same values as the input array, but they are in ascending order. The input array is split into two nearly equal-length subarrays, each of which is ordered using merge sort. Then the two subarrays are merged together.]

Input: r and s, *[positive integers with $r < s$]* $a[r], a[r + 1], \ldots, a[s]$ *[an array of data items that can be ordered]*

Algorithm Body:

```
$bot := r, top := s$
**while** $(bot < top)$
    $mid := \left\lfloor \dfrac{bot + top}{2} \right\rfloor$
    call **merge sort** with input $bot$, $mid$, and
        $a[bot], a[bot + 1], \ldots, a[mid]$
    call **merge sort** with input mid $+ 1$, top and
        $a[mid + 1], a[mid + 2], \ldots, a[top]$
        [After these steps are completed, the arrays $a[bot], a[bot+1], \ldots, a[mid]$ and
        $a[mid+1], a[mid+2], \ldots, a[top]$ are both in order.]
        merge $a[bot], a[bot+1], \ldots, a[mid]$ and
            $a[mid+1], a[mid+2], \ldots, a[top]$
        [This step can be done with a call to a merge algorithm. To put the final array in
        ascending order, the merge algorithm must be written so as to take two arrays in
        ascending order and merge them into an array in ascending order.]
    end while
Output: $a[r], a[r+1], \ldots, a[s]$ [an array with the same elements as the input array
but in ascending order]
```

To derive the efficiency of merge sort, let

$$\boxed{\begin{aligned} m_n = {} & \text{the maximum number of comparisons used} \\ & \text{when merge sort is applied to an array of length } n. \end{aligned}}$$

Then $m_1 = 0$ because no comparisons are used when merge sort is applied to an array of length 1. Also for any integer $k > 1$, consider an array $a[bot], a[bot+1], \ldots, a[top]$ of length k that is split into two subarrays, $a[bot], a[bot+1], \ldots, a[mid]$ and $a[mid+1], a[mid+2], \ldots, a[top]$, where $mid = \lfloor (bot + top)/2 \rfloor$. In exercise 24 you are asked to show that the right subarray has length $\lfloor k/2 \rfloor$ and the left subarray has length $\lceil k/2 \rceil$. From the previous discussion of the merge process, it is known that to merge two subarrays into an array of length k, at most $k - 1$ comparisons are needed.

Consequently,

$$\begin{bmatrix}\text{the number of comparisons} \\ \text{when merge sort is applied} \\ \text{to an array of length } k\end{bmatrix} = \begin{bmatrix}\text{the number of comparisons} \\ \text{when merge sort is applied} \\ \text{to an array of length } \lfloor k/2 \rfloor\end{bmatrix}$$
$$+ \begin{bmatrix}\text{the number of comparisons} \\ \text{when merge sort is applied} \\ \text{to an array of length } \lceil k/2 \rceil\end{bmatrix} + \begin{bmatrix}\text{the number of comparisons} \\ \text{used to merge two subarrays} \\ \text{into an array of length } k\end{bmatrix}.$$

Or, in other words,

$$m_k = m_{\lfloor k/2 \rfloor} + m_{\lceil k/2 \rceil} + (k-1) \quad \text{for all integers } k > 1.$$

In exercise 25 you are asked to use this recurrence relation to show that

$$\frac{1}{2} n \log_2 n \le m_n \le 2n \log_2 n \quad \text{for all integers } n \ge 1.$$

It follows that merge sort is $\Theta(n \log_2 n)$.

In the text and exercises for Section 11.3, we showed that insertion sort and selection sort are both $\Theta(n^2)$. How much difference can it make that merge sort is $\Theta(n \log_2 n)$? If $n = 100{,}000{,}000$ and a computer is used that performs one operation each nanosecond, the time needed to perform $n \log_2 n$ operations is about 2.7 seconds, whereas the time needed to perform n^2 operations is over 115 days.

Tractable and Intractable Problems

At an opposite extreme from an algorithm such as binary search, which has logarithmic order, is an algorithm with exponential order. For example, consider an algorithm to direct the movement of each of the 64 disks in the Tower of Hanoi puzzle as they are transferred one by one from one pole to another. In Section 5.7 we showed that such a transfer requires $2^{64} - 1$ steps. If a computer took a nanosecond to calculate each transfer step, the total time to calculate all the steps would be

$$\underset{\substack{\uparrow\\ \text{number of}\\ \text{moves}}}{(2^{64}-1)} \cdot \underset{\substack{\text{moves}\\ \text{per}\\ \text{second}}}{\left(\frac{1}{10^9}\right)} \cdot \underset{\substack{\text{seconds}\\ \text{per}\\ \text{minute}}}{\left(\frac{1}{60}\right)} \cdot \underset{\substack{\text{minutes}\\ \text{per}\\ \text{hour}}}{\left(\frac{1}{60}\right)} \cdot \underset{\substack{\text{hours}\\ \text{per}\\ \text{day}}}{\left(\frac{1}{24}\right)} \cdot \underset{\substack{\text{days}\\ \text{per}\\ \text{year}}}{\left(\frac{1}{365.25}\right)} \cong 584 \text{ years.}$$

Problems whose solutions can be found with algorithms whose worst-case order with respect to time is a polynomial are said to belong to **class P.** They are called **polynomial-time algorithms** and are said to be **tractable.** Problems that cannot be solved in polynomial time are called **intractable.** For certain problems, it is possible to check the correctness of a proposed solution with a polynomial-time algorithm, but it may not be possible to find a solution in polynomial time. Such problems are said to belong to **class NP.*** The biggest open question in theoretical computer science is whether every problem in class NP belongs to class P. This is known as the **P vs. NP** problem. The Clay Institute, in Cambridge, Massachusetts, has offered a prize of $1,000,000 to anyone who can either prove or disprove that P = NP.

In recent years, computer scientists have identified a fairly large set of problems, called **NP-complete,** that all belong to class NP but are widely believed not to belong to class P. What is known for sure is that if any one of these problems is solvable in polynomial time, then so are all the others. One of the NP-complete problems, commonly known as the *traveling salesman problem,* was discussed in Section 10.2.

A Final Remark on Algorithm Efficiency

This section and the previous one on algorithm efficiency have offered only a partial view of what is involved in analyzing a computer algorithm. For one thing, it is assumed that searches and sorts take place in the memory of the computer. Searches and sorts on disk-based files require different algorithms, though the methods for their analysis are similar. For another thing, as mentioned at the beginning of Section 11.3, time efficiency is not the only factor that matters in the decision about which algorithm to choose. The amount of memory space required is also important, and there are mathematical techniques to estimate space efficiency very similar to those used to estimate time efficiency. Furthermore, as parallel processing of data becomes increasingly prevalent, current methods of algorithm analysis are being modified and extended to apply to algorithms designed for this new technology.

Test Yourself

1. To solve a problem using a divide-and-conquer algorithm, you reduce it to a fixed number of smaller problems of the same kind, which can themselves be _____, and so forth until _____.
2. To search an array using the binary search algorithm in each step, you compare a middle element of the array to _____. If the middle element is less than _____, you _____, and if the middle element is greater than _____, you _____.
3. The worst case order of the binary search algorithm is _____.
4. To sort an array using the merge sort algorithm, in each step until the last one you split the array into approximately two equal sections and sort each section using ____. Then you _____ the two sorted sections.
5. The worst case order of the merge sort algorithm is _____.

*Technically speaking, a problem whose solution can be verified on an ordinary computer (or *deterministic sequential machine*) with a polynomial-time algorithm can be solved on a *nondeterministic sequential machine* with a polynomial-time algorithm. Such problems are called NP, which stands for *nondeterministic polynomial-time algorithm*.

Exercise Set 11.5

1. Use the facts that $\log_2 10 \cong 3.32$ and that for all real numbers a, $\log_2(10^a) = a \log_2 10$ to find $\log_2(1{,}000)$, $\log_2(1{,}000{,}000)$, and $\log_2(1{,}000{,}000{,}000{,}000)$.

2. Suppose an algorithm requires $c\lfloor \log_2 n \rfloor$ operations when performed with an input of size n (where c is a constant).
   a. By what factor will the number of operations increase when the input size is increased from m to m^2 (where m is a positive integer power of 2)?
   b. By what factor will the number of operations increase when the input size is increased from m to m^{10} (where m is a positive integer power of 2)?
   c. When n increases from $128\,(= 2^7)$ to $268{,}435{,}456$ $(= 2^{28})$, by what factor is $c\lfloor \log_2 n \rfloor$ increased?

Exercises 3 and 4 illustrate that for relatively small values of n, algorithms with larger orders can be more efficient than algorithms with smaller orders. Use a graphing calculator or computer to answer these questions.

3. For what values of n is an algorithm that requires n operations more efficient than an algorithm that requires $\lfloor 50 \log_2 n \rfloor$ operations?

4. For what values of n is an algorithm that requires $\lfloor n^2/10 \rfloor$ operations more efficient than an algorithm that requires $\lfloor n \log_2 n \rfloor$ operations?

In 5 and 6, trace the action of the binary search algorithm (Algorithm 11.5.1) on the variables *index, bot, top, mid*, and the given values of x for the input array $a[1]$ = Chia, $a[2]$ = Doug, $a[3]$ = Jan, $a[4]$ = Jim, $a[5]$ = José, $a[6]$ = Mary, $a[7]$ = Rob, $a[8]$ = Roy, $a[9]$ = Sue, $a[10]$ = Usha, where alphabetical ordering is used to compare elements of the array.

5. a. x = Chia b. x = Max

6. a. x = Amanda b. x = Roy

7. Suppose *bot* and *top* are positive integers with $bot \le top$. Consider the array

$$a[bot], a[bot+1], \ldots, a[top].$$

   a. How many elements are in this array?
   b. Show that if the number of elements in the array is odd, then the quantity $bot + top$ is even.
   c. Show that if the number of elements in the array is even, then the quantity $bot + top$ is odd.

Exercises 8–11 refer to the following algorithm segment. For each positive integer n, let a_n be the number of iterations of the **while** loop.

```
while (n > 0)
    n := n div 2
end while
```

8. Trace the action of this algorithm segment on n when the initial value of n is 27.

9. Find a recurrence relation for a_n.

10. Find an explicit formula for a_n.

11. Find an order for this algorithm segment.

Exercises 12–15 refer to the following algorithm segment. For each positive integer n, let b_n be the number of iterations of the **while** loop.

```
while (n > 0)
    n := n div 3
end while
```

12. Trace the action of this algorithm segment on n when the initial value of n is 424.

13. Find a recurrence relation for b_n.

H 14. a. Use iteration to guess an explicit formula for b_n.
   b. Prove that if k is an integer and x is a real number with $3^k \le x < 3^k$, then $\lfloor \log_3 x \rfloor = k$.
   c. Prove that for all integers $m \ge 1$,
   $$\lfloor \log_3(3m) \rfloor = \lfloor \log_3(3m+1) \rfloor = \lfloor \log_3(3m+2) \rfloor.$$
   d. Prove the correctness of the formula you found in part (a).

15. Find an order for the algorithm segment.

16. Complete the proof of case 2 of the strong induction argument in Example 11.5.5. In other words, show that if k is an odd integer and $w_i = \lfloor \log_2 i \rfloor + 1$ for all integers i with $1 \le i \le k$, then $w_{k+1} = \lfloor \log_2 k + 1 \rfloor + 1$.

For 17–19, modify the binary search algorithm (Algorithm 11.5.1) to take the upper of the two middle array elements in case the input array has even length. In other words, in Algorithm 11.5.1 replace

$$mid := \left\lfloor \frac{bot + top}{2} \right\rfloor \text{with } mid := \left\lceil \frac{bot + top}{2} \right\rceil.$$

17. Trace the modified binary search algorithm for the same input as was used in Example 11.5.1.

18. Suppose an array of length k is input to the **while** loop of the modified binary search algorithm. Show that after one iteration of the loop, if $a[mid] \neq x$, the input to the next iteration is an array of length at most $\lfloor k/2 \rfloor$.

19. Let w_n be the number of iterations of the **while** loop in a worst-case execution of the modified binary search algorithm for an input array of length n. Show that $w_k = 1 + w_{\lfloor k/2 \rfloor}$ for $k \geq 2$.

In 20 and 21, draw a diagram like Figure 11.5.4 to show how to merge the given subarrays into a single array in ascending order.

20. 3, 5, 6, 9, 12 and 2, 4, 7, 9, 11

21. F, K, L, R, U and C, E, L, P, W (alphabetical order)

In 22 and 23, draw a diagram like Figure 11.5.5 to show how merge sort works for the given input arrays.

22. R, G, B, U, C, F, H, G (alphabetical order)

23. 5, 2, 3, 9, 7, 4, 3, 2

24. Show that given an array $a[bot], a[bot+1], \ldots, a[top]$ of length k, if mid $= \lfloor (bot + top)/2 \rfloor$ then
    a. the subarray $a[mid+1], a[mid+2], \ldots, a[top]$ has length $\lfloor k/2 \rfloor$.
    b. the subarray $a[bot], a[bot+1], \ldots, a[mid]$ has length $\lceil k/2 \rceil$.

H **25.** The recurrence relation for $m_1, m_2, m_3, \ldots$, which arises in the calculation of the efficiency of merge sort, is

$$m_1 = 0$$
$$m_k = m_{\lfloor k/2 \rfloor} + m_{\lceil k/2 \rceil} + k - 1.$$

Show that for all integers $n \geq 1$,
a. $\frac{1}{2} n \log_2 n \leq m_n$ b. $m_n \leq 2n \log_2 n$

26. You might think that $n-1$ multiplications are needed to compute x^n, since

$$x^n = \underbrace{x \cdot x \cdots x}_{n-1 \text{ multiplications}}.$$

But observe that, for instance, since $6 = 4 + 2$,

$$x^6 = x^4 x^2 = (x^2)^2 x^2.$$

Thus x^6 can be computed using three multiplications: one to compute x^2, one to compute $(x^2)^2$, and one to multiply $(x^2)^2$ times x^2. Similarly, since $11 = 8 + 2 + 1$,

$$x^{11} = x^8 x^2 x^1 = ((x^2)^2)^2 x^2 x$$

and so x^{11} can be computed using five multiplications: one to compute x^2, one to compute $(x^2)^2$, one to compute $((x^2)^2)^2$, one to multiply $((x^2)^2)^2$ times x^2, and one to multiply that product by x.

a. Write an algorithm to take a real number x and a positive integer n and compute x^n by
   (i) calling Algorithm 5.1.1 to find the binary representation of n:

$$(r[k]\, r[k-1] \cdots r[0])_2,$$

   where each $r[i]$ is 0 or 1;
   (ii) computing $x^2, x^{2^2}, x^{2^3}, \ldots, x^{2^k}$ by squaring, then squaring again, and so forth,
   (iii) computing x^n using the fact that

$$\begin{aligned} x^n &= x^{r[k]2^k + \cdots + r[2]2^2 + r[1]2^1 + r[0]2^0} \\ &= x^{r[k]2^k} \cdots x^{r[2]2^2} \cdot x^{r[1]2^1} \cdot x^{r[0]2^0} \end{aligned}$$

b. Show that the number of multiplications performed by the algorithm of part (a) is less than or equal to $2\lfloor \log_2 n \rfloor$.

Answers for Test Yourself

1. reduced to the same finite number of smaller problems of the same kind; easily resolved problems are obtained 2. the element you are looking for; the element you are looking for; apply the binary search algorithm to the lower half of the array; the element you are looking for; apply the binary search algorithm to the upper half of the array 3. $\log_2 n$, where n is the length of the array 4. merge sort; merge 5. $n \log_2 n$