By Theorem 2.1.21(d),
$$\lg 2^{k-1} = (k-1)\lg 2.$$
Since
$$\lg 2 = \log_2 2 = 1,$$
we have
$$\lg 2^{k-1} = (k-1)\lg 2 = k-1.$$
Similarly,
$$\lg 2^k = k.$$
The given inequality now follows. □

Inequalities

In this subsection, we derive some inequalities that will be useful later.

Theorem 2.1.28. *Let a and b be numbers such that* $0 \le a < b$. *Then*
$$\frac{b^{n+1} - a^{n+1}}{b-a} < (n+1)b^n.$$

Proof. If $0 \le a < b$, then
$$\frac{b^{n+1} - a^{n+1}}{b-a} = \sum_{i=0}^{n} a^i b^{n-i} < \sum_{i=0}^{n} b^i b^{n-i} = (n+1)b^n. \qquad \blacksquare$$

We may use Theorem 2.1.28 to show that the sequence $\{(1+1/n)^n\}$ is increasing and bounded above by 4. This fact will be used to derive another useful inequality.

Theorem 2.1.29. *The sequence* $\{(1+1/n)^n\}$ *is increasing and bounded above by* 4.

Proof. We first rewrite the inequality of Theorem 2.1.28 as
$$b^n[b-(n+1)(b-a)] < a^{n+1}.$$
If we set $a = 1 + 1/(n+1)$ and $b = 1 + 1/n$, the term in brackets reduces to 1 and we have
$$\left(1+\frac{1}{n}\right)^n < \left(1+\frac{1}{n+1}\right)^{n+1}.$$
Therefore, the sequence $\{(1+1/n)^n\}$ is increasing.

Next, we set $a = 1$ and $b = 1 + 1/(2n)$. This time the term in brackets reduces to $\frac{1}{2}$, and we have
$$\left(1+\frac{1}{2n}\right)^n < 2.$$

Squaring both sides gives

$$\left(1 + \frac{1}{2n}\right)^{2n} < 4.$$

Since $\{(1 + 1/n)^n\}$ is increasing,

$$\left(1 + \frac{1}{n}\right)^n < \left(1 + \frac{1}{2n}\right)^{2n} < 4.$$

Therefore, the sequence $\{(1 + 1/n)^n\}$ is bounded above by 4. ■

A calculus theorem states that an increasing sequence that is bounded above converges; thus, $\{(1 + 1/n)^n\}$ converges. The limit is $e = 2.71828\ldots$, the base of the natural logarithm function.

Since $\{(1 + 1/n)^n\}$ is increasing and $(1 + 1/n)^n = 2$ if $n = 1$, we have

$$2 \le \left(1 + \frac{1}{n}\right)^n < 4$$

for all n. Taking the logarithm to the base 2 gives our next inequality.

Theorem 2.1.30.

$$\frac{1}{n} \le \lg(n+1) - \lg n < \frac{2}{n}$$

Proof. We have already noted that

$$2 \le \left(1 + \frac{1}{n}\right)^n < 4$$

for all n. Taking the logarithm to the base 2, we obtain

$$1 = \lg 2 \le \lg\left(1 + \frac{1}{n}\right)^n < \lg 4 = 2.$$

Since

$$\lg\left(1 + \frac{1}{n}\right)^n = \lg\left(\frac{n+1}{n}\right)^n = n \lg\left(\frac{n+1}{n}\right) = n[\lg(n+1) - \lg n],$$

the preceding inequality becomes

$$1 \le n[\lg(n+1) - \lg n] < 2.$$

Dividing by n yields the desired result. ■

Upper Bounds, Lower Bounds, Supremum, Infimum

The maximum value in a nonempty finite set of real numbers is simply the largest element in the set. If the set is infinite, the set may fail to have a

By properties of logarithms, we have

$$\lg n! = \lg n + \lg(n-1) + \cdots + \lg 2 + \lg 1.$$

Since lg is an increasing function,

$$\begin{aligned}\lg n! &= \lg n + \lg(n-1) + \cdots + \lg 2 + \lg 1\\ &\le \lg n + \lg n + \cdots + \lg n + \lg n = n \lg n.\end{aligned}$$

We conclude that

$$\lg n! = O(n \lg n).$$

For $n \ge 4$, we have

$$\begin{aligned}&\lg n + \lg(n-1) + \cdots + \lg 2 + \lg 1\\ &\quad\ge \lg n + \lg(n-1) + \cdots + \lg\lceil n/2\rceil\\ &\quad\ge \lg\lceil n/2\rceil + \lg\lceil n/2\rceil + \cdots + \lg\lceil n/2\rceil\\ &\quad= \lceil (n+1)/2\rceil \lg\lceil n/2\rceil\\ &\quad\ge (n/2)\lg(n/2)\\ &\quad= (n/2)\lg n - n/2\\ &\quad\ge n\lg n/4\end{aligned}$$

(since $\lg n \ge 2$ for $n \ge 4$). Therefore,

$$\lg n! = \Omega(n \lg n).$$

It follows that

$$\lg n! = \Theta(n \lg n).$$

□

The number H_n, defined by

$$H_n = \sum_{i=1}^{n} \frac{1}{i},$$

is called the nth **harmonic number**. It occurs frequently in the analysis of algorithms. Our next theorem gives a theta notation for the nth harmonic number.

Theorem 2.3.9.

$$\sum_{i=1}^{n} \frac{1}{i} = \Theta(\lg n)$$

Proof. By Theorem 2.1.30,

$$\frac{1}{n} \le \lg(n+1) - \lg n < \frac{2}{n}.$$

Therefore,

$$\sum_{i=1}^{n} \frac{1}{i} \le \sum_{i=1}^{n} [\lg(i+1) - \lg i] = \lg(n+1) \le \lg 2n = 1 + \lg n \le 2 \lg n,$$

if $n \ge 2$. Thus,

$$\sum_{i=1}^{n} \frac{1}{i} = O(\lg n).$$

Also,

$$\sum_{i=1}^{n} \frac{1}{i} > \frac{1}{2} \sum_{i=1}^{n} [\lg(i+1) - \lg i] = \frac{1}{2} \lg(n+1) \ge \frac{1}{2} \lg n.$$

Thus,

$$\sum_{i=1}^{n} \frac{1}{i} = \Omega(\lg n).$$

We conclude that

$$\sum_{i=1}^{n} \frac{1}{i} = \Theta(\lg n).$$ ∎

The asymptotic notation can be extended to functions of two or more variables as follows. If f and g are nonnegative functions of two variables on the positive integers, we write

$$f(m,n) = O(g(m,n))$$

if there exist constants C and N such that

$$f(m,n) \le Cg(m,n) \quad \text{for all } m \ge N \text{ and } n \ge N.$$

For example,

$$(1 + 2 + \cdots + m)(1 + 2 + \cdots + n) = O(m^2 n^2).$$

We may also extend the definition of the omega and theta notations to functions of two or more variables in a similar way.

We next define what it means for the worst-case or average-case time of an algorithm to be of order at most $g(n)$.

Definition 2.3.10. If an algorithm requires $t(n)$ units of time to terminate in the worst case for an input of size n and

$$t(n) = O(g(n)),$$

we say that the *worst-case time required by the algorithm is of order at most* $g(n)$ or that the *worst-case time required by the algorithm is* $O(g(n))$.

wonder whether a second array is necessary. The answer is "No." Several versions of merge have been designed so that the merging is done in place. By "in place," we mean that only one extra cell (in addition to the input array) is allowed, together with $O(1)$ storage for handling array indexing. One of the most efficient in-place merge algorithms is due to Huang and Langston (see Huang, 1988). In practice, in-place merging is much slower than merging using an extra array.

In-place merging gives rise to an in-place version of mergesort. However, since the merging is slower than merging using an extra array, in practice, the resulting version of mergesort runs much slower than Algorithm 5.2.3.

An alternative to using an in-place version of merge in mergesort is to obtain an in-place version of mergesort directly (see, e.g., Katajainen, 1996). The key observation that makes a direct in-place version of mergesort possible is that if we have an array a consisting of a sorted subarray of size m followed by a sorted subarray of size n, then the subarrays can be merged into a using extra storage, say an array b, of size $\min\{m, n\}$. For example, if $m \le n$, we copy the first m elements of a into b and then merge b and the last n elements of a into a using the standard merge algorithm (see Exercise 13).

Now suppose that we have an array of size n. The idea of in-place mergesort is to sort the first $n/2$ elements using the last part of the array as the extra storage. (When we do so, we have to be careful to *swap* data rather than overwrite data; for details, see Katajainen, 1996.) Next, we sort the $n/4$ elements following the first $n/2$ elements, using the last fourth of the array as the extra storage. We then merge the two sorted subarrays using the last fourth of the array as the extra storage. We then repeatedly sort half of the unsorted end of the array and merge it with the already sorted first part of the array, using the end of the array as the extra storage until the entire array is sorted. In practice, an optimized version of this algorithm runs much faster than in-place mergesort using in-place merging; however, experiments by Katajainen, et al., show that the optimized version is about 50 percent slower than mergesort implemented with a version of merge that uses a second array (as in Algorithm 5.2.3).

Exercises

Show how merge *merges the arrays in Exercises 1–4.*

1S. 14 24 27
17 26 54

2. 14 24 27 28 31 45 47 51
7 10 11 29 31

3. 7 10 11 29 31
47 50 71 79 101

4S. 7 10 11 29 31
9 28 71 79 101

Show how mergesort sorts each array in Exercises 5–8.

5S. 14 40 31 28 3 15 17 51

6. 3 14 15 17 28 31 40 51

7. 51 40 31 28 17 15 14 3

8S. 23 23 23 23 23 23 23 23

9S. Write a nonrecursive version of mergesort. Make your algorithm as efficient as you can.

10. Give a formal proof using mathematical induction that mergesort is stable.

11. Suppose we merge two sorted *linked* lists of sizes m and n by changing reference fields in the list. Show that the best-case time is $\Theta(\min\{m, n\})$ and the worst-case time is $\Theta(m + n)$. Give examples of input that produce the best-case time and the worst-case time.

12S. Let t_n be the worst-case time of mergesort. Show that t_n is nondecreasing.

13. Write an algorithm that, given an array a consisting of a sorted subarray of size m followed by a sorted subarray of size n, merges them into a using an extra array b of size $\min\{m, n\}$.

5.3 Finding a Closest Pair of Points

A lumber store stocks several different kinds of plywood. Each is classified according to several features (e.g., number of knotholes per square foot, smoothness). In Figure 5.3.1, the different kinds of plywood are plotted as points in the plane, where the x-coordinate measures the number of knotholes per square foot, and the y-coordinate measures the smoothness. Notice that the distance between two points gives a dissimilarity measure: If the distance is small, the types of plywood represented by the points are similar; if the distance is large, the types of plywood represented by the points are dissimilar. The store decides to reduce the number of different kinds of plywood in stock by identifying two types of plywood that are most similar and eliminating one of them. The problem then is to identify a closest pair of points and eliminate a type of plywood that corresponds to one of the points. (We say *a* closest pair since it is possible that several pairs achieve the same minimum distance.) Our distance measure is ordinary Euclidean distance.

WWW The **closest-pair problem** furnishes an example of a problem from computational geometry. **Computational geometry** is concerned with the design is already sorted. If *val* $< a[i-1]$, we move $a[i-1]$ one cell to the right (i.e., we copy $a[i-1]$ to $a[i]$), and we then compare *val* with $a[i-2]$. If *val* $\ge a[i-2]$, we copy *val* to $a[i-1]$ and stop since now

$$a[1],\ldots,a[i]$$

is sorted. If *val* $< a[i-2]$, we move $a[i-2]$ one cell to the right. We continue shifting data to the right until we obtain the correct position for *val* after which we insert *val*. The algorithm terminates when $a[n]$ is inserted into the proper position in the array

$$a[1],\ldots,a[n-1].$$

Example 6.1.1. Suppose that the array to sort is

$$\begin{array}{|c|c|c|c|c|}\hline 36 & 14 & 27 & 40 & 31 \\ \hline\end{array}\ .$$

Insertion sort first inserts 14 into the one-element array 36. Since $14 < 36$, 36 moves one cell to the right

$$\begin{array}{|c|c|}\hline & 36 \\ \hline\end{array}\ ,$$

after which 14 is inserted into the first cell

$$\begin{array}{|c|c|}\hline 14 & 36 \\ \hline\end{array}\ .$$

Insertion sort next inserts 27 into the array $14, 36$. Since $27 < 36$, 36 moves one cell to the right

$$\begin{array}{|c|c|c|}\hline 14 & & 36 \\ \hline\end{array}\ .$$

Since $27 > 14$, 14 does not move, and 27 is inserted into the second cell

$$\begin{array}{|c|c|c|}\hline 14 & 27 & 36 \\ \hline\end{array}\ .$$

Insertion sort next inserts 40 into the array $14, 27, 36$. Since $40 > 36$, 36 does not move; the array is already sorted

$$\begin{array}{|c|c|c|c|}\hline 14 & 27 & 36 & 40 \\ \hline\end{array}\ .$$

Finally, insertion sort inserts 31 into the array $14, 27, 36, 40$. Since $31 < 40$, 40 moves one cell to the right

$$\begin{array}{|c|c|c|c|c|}\hline 14 & 27 & 36 & & 40 \\ \hline\end{array}\ .$$

Since $31 < 36$, 36 moves one cell to the right

14	27		36	40

.

Since $31 > 27$, 27 does not move, and 31 is inserted into the third cell

14	27	31	36	40

.

The array is sorted. □

We state insertion sort as Algorithm 6.1.2.

Algorithm 6.1.2 Insertion Sort. This algorithm sorts the array a by first inserting $a[2]$ into the sorted array $a[1]$; next inserting $a[3]$ into the sorted array $a[1], a[2]$; and so on; and finally inserting $a[n]$ into the sorted array $a[1], \ldots, a[n-1]$.

Input Parameter: a
Output Parameter: a

```
insertion_sort(a) {
  n = a.last
  for i = 2 to n {
    val = a[i] // save a[i] so it can be inserted into the correct place
    j = i - 1
    // if val < a[j], move a[j] right to make room for a[i]
    while (j ≥ 1 && val < a[j]) {
      a[j + 1] = a[j]
      j = j - 1
    }
    a[j + 1] = val // insert val
  }
}
```

In the worst case in Algorithm 6.1.2, the while loop executes the maximum number of times, which occurs when *val* is always inserted at index one. Such a situation occurs when the input is sorted in *decreasing* order. In this case, when $i = 2$ the while loop body executes once. When $i = 3$, the while loop body executes twice, and so on. Thus, in the worst case, the total number of times that the while loop body executes is

$$1 + 2 + \cdots + (n-1) = \frac{(n-1)n}{2} = \Theta(n^2).$$

Thus, the worst-case time of Algorithm 6.1.2 is $\Theta(n^2)$.

Exercises

Show how insertion sort sorts each array in Exercises 1–4.

18. Answer Exercise 17 with "array" replaced by "linked list."

19. If the size of an array is small, say less than some number s, whose exact value depends on the implementation, system, and so on, insertion sort can run faster than mergesort (see Section 5.2). Modify mergesort to take this fact into account, and, thereby, make mergesort run faster.

6.2 Quicksort

WWW Like mergesort (see Section 5.2), **quicksort** first divides the array to be sorted into two parts and then sorts each part. In quicksort, this division process is called **partition**. Unlike mergesort, which divides the array into two nearly equal parts, the sizes of the two parts into which partition divides the array can range from nearly equal to highly unequal. The division depends on a particular element, called the *partition element*, that is chosen. Partition rearranges the data so that the partition element is in the correct position (i.e., the partition element is in the cell where it would be *if* the entire array was sorted). In addition, the particular version of partition that we discuss places all data less than the partition element to the left of the partition element (but not necessarily in sorted order) and all data greater than or equal to the partition element to the right of the partition element (again not necessarily in sorted order). Because the data are on the correct sides of the partition element, once quicksort sorts each side, the entire array is sorted.

Example 6.2.1. Quicksort begins by partitioning the array to be sorted. If the array

12	30	21	8	6	9	1	7

is partitioned using 12 as the partition element, the result could be

7	8	6	9	1	12	21	30

.

(The arrangement of the data to the left and right of the partition element is determined by the particular version of partition that is used.) The partition element, 12, is in the cell where it would be if the entire array was sorted. The data to the left of 12 are less than 12, and the data to the right of 12 are greater than or equal to 12. The quicksort algorithm concludes by sorting the part to the left of 12 and the part to the right of 12

1	6	7	8	9	12	21	30

after which the array is sorted. □

Jon Bentley (see Bentley, 2000, page 117) attributes our first version of partition to Nico Lomuto. We use the first element *val* as the partition element. At the kth iteration of partition, there is a section of the array a following *val* containing data less than *val* and an index h marking the end of this

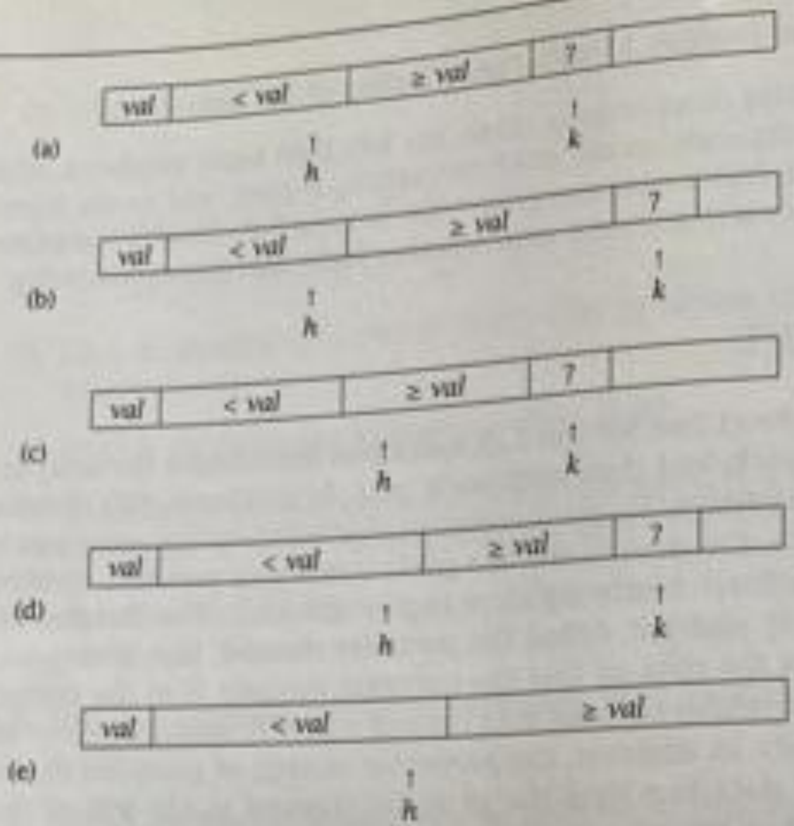


Figure 6.2.1 Partitioning an array a. The situation just before another iteration of the algorithm is shown in (a). If $a[k] \geq val$, we may extend the second section of the array to include $a[k]$ by simply incrementing k [see (b)]. If $a[k] < val$, we may increment h and then swap $a[h]$ and $a[k]$ [see (c)], thereby extending the first section and shifting the second section's collection of values one to the right. We then increment k [see (d)]. We are now ready for another iteration of the algorithm. After iterating through the array, we have the situation shown in (e), so if we then swap val and $a[h]$, the partitioning is complete.

section of the array. This section of the array is followed by another section containing data greater than or equal to *val*. The cell following this section has index k [see Figure 6.2.1(a)]. Either or both of these sections might be empty. If $a[k] \geq val$, we may extend the second section of the array to include $a[k]$ by simply incrementing k [see Figure 6.2.1(b)]. If $a[k] < val$, we may increment h and then swap $a[h]$ and $a[k]$ [see Figure 6.2.1(c)], thereby extending the first section and shifting the second section's collection of values one to the right. We then increment k [see Figure 6.2.1(d)]. We are now ready for another iteration of partition. After iterating through the array, we have the situation shown in Figure 6.2.1(e), so if we then swap *val* and $a[h]$, the partitioning is complete. We conclude with *val* at index h, all data less than *val* to the left of *val*'s cell, and all data greater than or equal to *val* to the right of *val*'s cell.

We state the partition algorithm as Algorithm 6.2.2.

Algorithm 6.2.2 Partition. This algorithm partitions the array

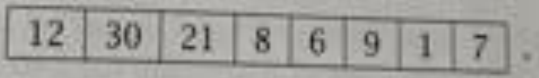
$$a[i], \ldots, a[j]$$

by inserting $val = a[i]$ at the index h where it would be if the array was sorted. When the algorithm concludes, values at indexes less than h are less than val, and values at indexes greater than h are greater than or equal to val. The algorithm returns the index h.

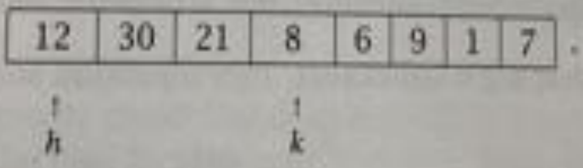
Input Parameters: a, i, j
Output Parameter: a

```
partition(a, i, j) {
  val = a[i]
  h = i
  for k = i + 1 to j
    if (a[k] < val) {
      h = h + 1
      swap(a[h], a[k])
    }
  swap(a[i], a[h])
  return h
}
```

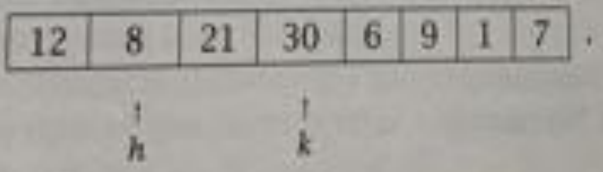
Example 6.2.3. We show how Algorithm 6.2.2 partitions the array



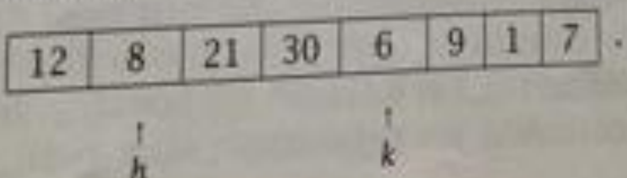
Since 30 and 21 are greater than 12, k simply increments giving



Since $8 < 12$, h increments and we swap $a[h]$ and $a[k]$ giving



Next k increments giving



Since $6 < 12$, h increments and we swap $a[h]$ and $a[k]$ giving

$$\begin{array}{|c|c|c|c|c|c|c|c|}\hline 12 & 8 & 6 & 30 & 21 & 9 & 1 & 7 \\\hline \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{\uparrow} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{\uparrow} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} \\ \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{h} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{k} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} \end{array} .$$

Next k increments giving

$$\begin{array}{|c|c|c|c|c|c|c|c|}\hline 12 & 8 & 6 & 30 & 21 & 9 & 1 & 7 \\\hline \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{\uparrow} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{\uparrow} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} \\ \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{h} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{k} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} \end{array} .$$

Each of 9, 1, and 7 is less than 12, so for each of these values, h increments and $a[h]$ and $a[k]$ are swapped, giving

$$\begin{array}{|c|c|c|c|c|c|c|c|}\hline 12 & 8 & 6 & 9 & 1 & 7 & 21 & 30 \\\hline \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{\uparrow} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} \\ \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{h} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} \end{array} .$$

After swapping $a[h]$ and the partition element, 12,

$$\begin{array}{|c|c|c|c|c|c|c|c|}\hline 7 & 8 & 6 & 9 & 1 & 12 & 21 & 30 \\\hline \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{\uparrow} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} \\ \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{h} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} \end{array} ,$$

all values to the left of 12 are less than 12, and all values to the right of 12 are greater than or equal to 12. The partitioning is complete. □

The proof that Algorithm 6.2.2 is correct is given in Figure 6.2.1.

If we define the time of Algorithm 6.2.2 to be the number of comparisons of array elements, the time for an array of size n is $n - 1$.

Having written partition, it is straightforward to write quicksort.

Algorithm 6.2.4 Quicksort. This algorithm sorts the array

$$a[i], \ldots, a[j]$$

by using the partition algorithm (Algorithm 6.2.2).

```
Input Parameters:  a, i, j
Output Parameter:  a

quicksort(a, i, j) {
  if (i < j) {
    p = partition(a, i, j)
    quicksort(a, i, p - 1)
    quicksort(a, p + 1, j)
  }
}
```

Because of all of the swapping in partition, it is not surprising that quicksort is *not* stable. We leave as an exercise (Exercise 12) the problem of constructing a concrete example.

In the following subsection, we discuss the worst-case time of quicksort.

Worst-Case Time for Quicksort

As we remarked earlier, in general a divide-and-conquer algorithm is most efficient when the division is as even as possible. Thus, we suspect that worst-case time for quicksort occurs when the division is as *uneven* as possible. We show that this is indeed the case.

Suppose we input an n-element array a to quicksort that has the property that every time partition is called, the partition element is placed at either the beginning or the end of the array (resulting in the worst possible division). Such a situation can occur. If a is sorted in increasing order, the partition element is always placed at the beginning of the array (see Exercise 9). When quicksort is first called, partition takes time $n - 1$. Quicksort is then called twice, once with an empty array and again with an array of size $n - 1$. The next call to partition takes time $n - 2$. Quicksort is again called twice, once with an empty array and again with an array of size $n - 2$. The next call to partition takes time $n - 3$. The process continues. The total time for all of the calls to partition is

$$(n-1) + (n-2) + (n-3) + \cdots + 1 = \frac{(n-1)n}{2} = \Theta(n^2).$$

Therefore, the worst-case time to execute quicksort is $\Omega(n^2)$. We show that the time for quicksort is always $O(n^2)$, so that the worst-case time to execute quicksort is $\Theta(n^2)$.

Theorem 6.2.5. *The worst-case time for quicksort for an array of size n is $\Theta(n^2)$.*

Proof. We have already noted that if an n-element input array is sorted in increasing order, the time for quicksort is $\Theta(n^2)$. Thus, the worst-case time of quicksort is $\Omega(n^2)$. We show that the worst-case time of quicksort is $O(n^2)$, thus proving the theorem.

Let c_n be the worst-case time for quicksort for an array of size n. Then $c_1 \geq 1$. We use mathematical induction to show that $c_n \leq c_1 n^2$, thus proving that the worst-case time of quicksort is $O(n^2)$. The basis step is immediate.

Suppose that $n > 1$ and $c_p \leq c_1 p^2$ for all $p < n$. We show that $c_n \leq c_1 n^2$. Suppose that the input is the array $a[1],\ldots,a[n]$. Quicksort first partitions a, which takes time (i.e., the number of comparisons) at most n. If the partition element is at index p, the recursive call to sort the left side takes time at most c_{p-1}, and the recursive call to sort the right side takes time at most c_{n-p}. Thus, if the partition element is at index p, the worst-case time is at most

$$n + c_{p-1} + c_{n-p}.$$

Since the partition element could be at any index between 1 and n,

$$c_n \le n + \max_{1 \le p \le n} c_{p-1} + c_{n-p}.$$

By the inductive assumption, $c_{p-1} \le c_1(p-1)^2$ and $c_{n-p} \le c_1(n-p)^2$. Therefore,

$$c_n \le n + \max_{1 \le p \le n} c_{p-1} + c_{n-p} \le n + c_1 \max_{1 \le p \le n} (p-1)^2 + (n-p)^2.$$

The function $y = (p-1)^2 + (n-p)^2$ on $[1, n]$ is a parabola symmetric about the line $p = (n+1)/2$ that opens up (see Figure 6.2.2). Thus, the maximum occurs at the points $p = 1$ and $p = n$; and the value of this maximum is $(n-1)^2$. It follows that

$$c_n \le n + c_1 \max_{1 \le p \le n} (p-1)^2 + (n-p)^2 \le n + c_1(n-1)^2.$$

Now

$$n + c_1(n-1)^2 = c_1 n^2 + (1 - 2c_1)n + c_1.$$

Since $n > 1$ and $c_1 \ge 1$, the expression $(1-2c_1)n + c_1$ is negative. Therefore,

$$n + c_1(n-1)^2 = c_1 n^2 + (1-2c_1)n + c_1 \le c_1 n^2,$$

and

$$c_n \le c_1 n^2.$$

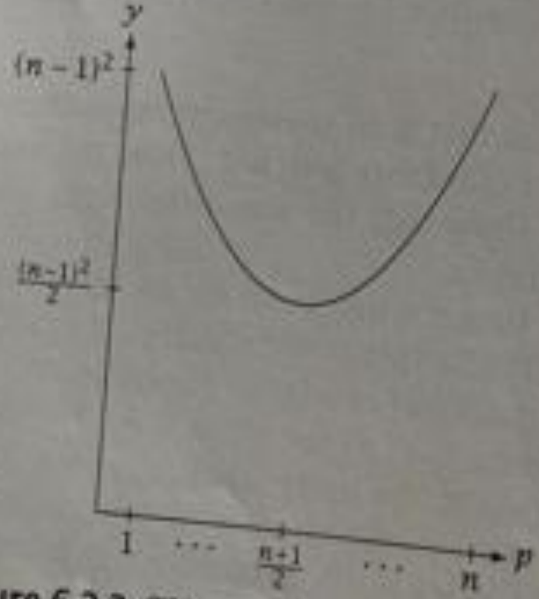


Figure 6.2.2 The graph of $y = (p-1)^2 + (n-p)^2$ on $[1, n]$. The maximum occurs at the points $p = 1$ and $p = n$, and the value of this maximum is $(n-1)^2$.

■

46. We have

$$1^k + 2^k + \cdots + n^k \le n^k + n^k + \cdots + n^k = n \cdot n^k = n^{k+1}$$

for $n \ge 1$; hence

$$1^k + 2^k + \cdots + n^k = O(n^{k+1}).$$

Also,

$$\begin{aligned} 1^k + 2^k + \cdots + n^k &\ge \lceil n/2 \rceil^k + \cdots + (n-1)^k + n^k \\ &\ge \lceil n/2 \rceil^k + \cdots + \lceil n/2 \rceil^k + \lceil n/2 \rceil^k \\ &= \lceil (n+1)/2 \rceil \lceil n/2 \rceil^k \ge (n/2)(n/2)^k = n^{k+1}/2^{k+1}. \end{aligned}$$

Therefore,

$$1^k + 2^k + \cdots + n^k = \Omega(n^{k+1}).$$

We conclude that

$$1^k + 2^k + \cdots + n^k = \Theta(n^{k+1}).$$

Section 2.4

1. After one month, there is still just one pair because a pair does not become productive until after one month. Therefore, $a_1 = 1$. After two months, the pair alive in the beginning becomes productive and adds one additional pair. Therefore, $a_2 = 2$. The increase in pairs of rabbits $a_n - a_{n-1}$ from month $n-1$ to month n is due to each pair alive in month $n-2$ producing an additional pair. That is, $a_n - a_{n-1} = a_{n-2}$. Since $a_1 = f_2$ and $a_2 = f_3$ and $\{a_n\}$ satisfies the same recurrence relation as $\{f_n\}$, we conclude that $a_n = f_{n+1}$, $n \ge 1$.

4. Basis Step ($n = 2$): $f_2^2 = 1^2 = 1 = 1 \cdot 2 - 1 = f_1 f_3 + (-1)^3$

Inductive Step:

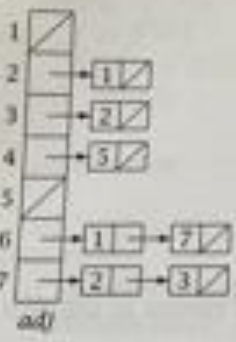
$$\begin{aligned} f_n f_{n+2} + (-1)^n &= f_n(f_n + f_{n+1}) + (-1)^n \\ &= f_n^2 + f_n f_{n+1} + (-1)^n \\ &= f_{n-1} f_{n+1} + (-1)^{n+1} + f_n f_{n+1} + (-1)^n \\ &= f_{n+1}(f_{n-1} + f_n) = f_{n+1}^2 \end{aligned}$$

7. Basis Steps ($n = 1, 2$): $f_1 = 1$ is odd, and the subscript, 1, is not divisible by 3. $f_2 = 1$ is odd, and the subscript, 2, is not divisible by 3.

Inductive Step: First, assume that n is divisible by 3. Then, $n-1$ and $n-2$ are not divisible by 3; so, by the inductive assumption, f_{n-1} and f_{n-2} are odd. Therefore, $f_n = f_{n-1} + f_{n-2}$ is even.

Now assume that n is not divisible by 3. Then, exactly one of $n-1$, $n-2$ is divisible by 3; so, by the inductive assumption, one of f_{n-1}, f_{n-2} is odd and the other is even. Therefore, $f_n = f_{n-1} + f_{n-2}$ is odd.

15.



Section 3.4

1. We use induction to prove that if a binary tree has n nodes, the conditional line,

```
if (root != null) {
```

is executed $2n + 1$ times.

The proof is by induction on n. The basis step is $n = 0$. In this case, the conditional line is executed $1 = 2 \cdot 0 + 1$ time, which verifies the basis step.

Now assume that $n > 0$ and if a binary tree has $m < n$ nodes, the conditional line is executed $2m + 1$ times.

Let T be an n-node binary tree with root *root*. Suppose that T's left subtree contains n_l nodes and T's right subtree contains n_r nodes. When inorder is called with input *root*, first the line

```
if (root != null) {
```

is executed, which accounts for one execution of the conditional line.

Next, inorder is called with input *root.left*. The tree rooted at *root.left* has $n_l < n$ nodes; so, by the inductive assumption, the conditional line is executed a total of $2n_l + 1$ times while processing T's left subtree. Similarly, the tree rooted at *root.right* has $n_r < n$ nodes; so, by the inductive assumption, the conditional line is executed a total of $2n_r + 1$ times while processing T's right subtree. The total number of times the conditional line is executed is, therefore,

$$1 + (2n_l + 1) + (2n_r + 1) = 2(n_l + n_r + 1) + 1 = 2n + 1.$$

The inductive step is complete.

4.
```
terminals(root) {
  if (root == null) {
    return 0
  if (root.left == null && root.right == null)
    return 1
  return terminals(root.left) + terminals(root.right)
}
```

10 comparisons in the worst case. Thus the algorithm does not use the minimum number of comparisons when $n = 6$.

Section 6.4

1. After initializing the array c to zero, the next for loop sets $c[k]$ to the number of occurrences of value k in the array input array a:

$$c[2] = c[3] = c[8] = c[9] = c[10] = c[15] = c[17] = 1.$$

The next for loop modifies c so that $c[k]$ is equal to the number of elements in a less than or equal to k:

$$c[2] = 1,\quad c[3] = 2,\quad c[8] = 3,\quad c[9] = 4,\quad c[10] = 5,\quad c[15] = 6,\quad c[17] = 7.$$

The next for loop copies the items from a into the array b, which will be sorted. Copying begins with the last element in a.

The last element in a is 8. Since $c[8] = 3$, 8 is copied to the third cell of b:

		8				

.

The predecessor of 8 in a is 10 and $c[10] = 5$, so 10 is copied to the fifth cell of b:

		8		10		

.

The predecessor of 10 in a is 9 and $c[9] = 4$, so 9 is copied to the fourth cell of b:

		8	9	10		

.

The predecessor of 9 in a is 2 and $c[2] = 1$, so 2 is copied to the first cell of b:

2		8	9	10		

.

The predecessor of 2 in a is 17 and $c[17] = 7$, so 17 is copied to the seventh cell of b:

2		8	9	10		17

.

The predecessor of 17 in a is 3 and $c[3] = 2$, so 3 is copied to the second cell of b:

2	3	8	9	10		17

.

The predecessor of 3 in a is 15 and $c[15] = 6$, so 15 is copied to the sixth cell of b:

2	3	8	9	10	15	17

.

The sort is complete when the last for loop copies b to a.