how you might change Partition so that QuickSort can take advantage of this situation.

5. In addition to Partition, there are many other ways to partition a set. Consider modifying Partition so that i is incremented while $a[i] \leq v$ instead of $a[i] < v$. Rewrite Partition making all of the necessary changes to it and then compare the new version with the original.

6. Compare the sorting methods MergeSort1 and QuickSort2 (Algorithm 3.10 and 3.14, respectively). Devise data sets that compare both the average- and worst-case times for these two algorithms.

7. (a) On which input data does the algorithm QuickSort exhibit its worst-case behavior?

   (b) Answer part (a) for the case in which the partitioning element is selected according to the median of three rule.

8. With MergeSort we included insertion sorting to eliminate the bookkeeping for small merges. How would you use this technique to improve QuickSort?

9. Take the iterative versions of MergeSort and QuickSort and compare them for the same-size data sets as used in Section 3.5.1.

10. Let S be a sample of s elements from X. If X is partitioned into $s+1$ parts as in Algorithm 3.16, show that the size of each part is $\tilde{O}(\frac{n}{s}\log n)$.

3.6 SELECTION

The Partition algorithm of Section 3.5 can also be used to obtain an efficient solution for the selection problem. In this problem, we are given n elements $a[1:n]$ and are required to determine the kth-smallest element. If the partitioning element v is positioned at $a[j]$, then $j-1$ elements are less than or equal to $a[j]$ and $n-j$ elements are greater than or equal to $a[j]$. Hence if $k < j$, then the kth-smallest element is in $a[1:j-1]$; if $k = j$, then $a[j]$ is the kth-smallest element; and if $k > j$, then the kth-smallest element is the $(k-j)$th-smallest element in $a[j+1:n]$. The resulting algorithm is function Select1 (Algorithm 3.17). This function places the kth-smallest element into position $a[k]$ and partitions the remaining elements so that $a[i] \leq a[k]$, $1 \leq i < k$, and $a[i] \geq a[k]$, $k < i \leq n$.

Example 3.10 Let us simulate Select1 as it operates on the same array used to test Partition in Section 3.5. The array has the nine elements 65, 70,

```
1   Algorithm Select1(a, n, k)
2   // Selects the kth-smallest element in a[1 : n] and places it
3   // in the kth position of a[ ]. The remaining elements are
4   // rearranged such that a[m] ≤ a[k] for 1 ≤ m < k, and
5   // a[m] ≥ a[k] for k < m ≤ n.
6   {
7       low := 1; up := n + 1;
8       a[n + 1] := ∞; // a[n + 1] is set to infinity.
9       repeat
10      {
11          // Each time the loop is entered,
12          // 1 ≤ low ≤ k ≤ up ≤ n + 1.
13          j := Partition(a, low, up);
14                  // j is such that a[j] is the jth-smallest value in a[ ].
15          if (k = j) then return;
16          else if (k < j) then up := j; // j is the new upper limit.
17                  else low := j + 1; // j + 1 is the new lower limit.
18      } until (false);
19  }
```

Algorithm 3.17 Finding the kth-smallest element

75, 80, 85, 60, 55, 50, and 45, with $a[10] = \infty$. If $k = 5$, then the first call of Partition will be sufficient since 65 is placed into $a[5]$. Instead, assume that we are looking for the seventh-smallest element of a, that is, $k = 7$. The next invocation of Partition is Partition$(6, 10)$.

a:	(5)	(6)	(7)	(8)	(9)	(10)
	65	85	80	75	70	$+\infty$
	65	70	80	75	85	$+\infty$

This last call of Partition has uncovered the ninth-smallest element of a. The next invocation is Partition$(6, 9)$.

a:	(5)	(6)	(7)	(8)	(9)	(10)
	65	70	80	75	85	$+\infty$
	65	70	80	75	85	$+\infty$

This time, the sixth element has been found. Since $k \neq j$, another call to Partition is made, Partition$(7, 9)$.

a:	(5)	(6)	(7)	(8)	(9)	(10)
	65	70	80	75	85	$+\infty$
	65	70	75	80	85	$+\infty$

Now 80 is the partition value and is correctly placed at $a[8]$. However, Select1 has still not found the seventh-smallest element. It needs one more call to Partition, which is Partition$(7, 8)$. This performs only an interchange between $a[7]$ and $a[8]$ and returns, having found the correct value. □

In analyzing Select1, we make the same assumptions that were made for QuickSort:

1. The n elements are distinct.
2. The input distribution is such that the partition element can be the ith-smallest element of $a[m : p-1]$ with an equal probability for each i, $1 \leq i \leq p-m$.

Partition requires $O(p-m)$ time. On each successive call to Partition, either m increases by at least one or j decreases by at least one. Initially $m = 1$ and $j = n+1$. Hence, at most n calls to Partition can be made. Thus, the worst-case complexity of Select1 is $O(n^2)$. The time is $\Omega(n^2)$, for example, when the input $a[1 : n]$ is such that the partitioning element on the ith call to Partition is the ith-smallest element and $k = n$. In this case, m increases by one following each call to Partition and j remains unchanged. Hence, n calls are made for a total cost of $O(\sum_1^n i) = O(n^2)$. The average computing time of Select1 is, however, only $O(n)$. Before proving this fact, we specify more precisely what we mean by the average time.

Let $T_A^k(n)$ be the average time to find the kth-smallest element in $a[1 : n]$. This average is taken over all $n!$ different permutations of n distinct elements. Now define $T_A(n)$ and $R(n)$ as follows:

$$T_A(n) = \frac{1}{n} \sum_{1 \leq k \leq n} T_A^k(n)$$

and

$$R(n) = \max_k \{T_A^k(n)\}$$

$T_A(n)$ is the average computing time of Select1. It is easy to see that $T_A(n) \leq R(n)$. We are now ready to show that $T_A(n) = O(n)$.

Theorem 3.3 The average computing time $T_A(n)$ of Select1 is $O(n)$.

Proof: On the first call to Partition, the partitioning element v is the ith-smallest element with probability $\frac{1}{n}, 1 \leq i \leq n$ (this follows from the assumption on the input distribution). The time required by Partition and the **if** statement in Select1 is $O(n)$. Hence, there is a constant $c, c > 0$, such that

$$\begin{aligned} T_A^k(n) &\leq cn + \frac{1}{n}[\sum_{1 \leq i < k} T_A^{k-1}(n-i) + \sum_{k < i \leq n} T_A^k(i-1)], \quad n \geq 2 \\ \text{So, } R(n) &\leq cn + \frac{1}{n} \max_k \{\sum_{1 \leq i < k} R(n-i) + \sum_{k < i \leq n} R(i-1)\} \\ R(n) &\leq cn + \frac{1}{n} \max_k \{\sum_{n-k+1}^{n-1} R(i) + \sum_{k}^{n-1} R(i)\}, \quad n \geq 2 \end{aligned} \tag{3.8}$$

We assume that c is chosen such that $R(1) \leq c$ and show, by induction on n, that $R(n) \leq 4cn$.

Induction Base: For $n = 2$, (3.8) gives

$$\begin{aligned} R(n) &\leq 2c + \frac{1}{2} \max \{R(1), R(1)\} \\ &\leq 2.5c < 4cn \end{aligned}$$

Induction Hypothesis: Assume $R(n) \leq 4cn$ for all $n, 2 \leq n < m$.

Induction Step: For $n = m$, (3.8) gives

$$R(m) \leq cm + \frac{1}{m} \max_k \left\{ \sum_{m-k+1}^{m-1} R(i) + \sum_{k}^{m-1} R(i) \right\}$$

Since we know that $R(n)$ is a nondecreasing function of n, it follows that

$$\sum_{m-k+1}^{m-1} R(i) + \sum_{k}^{m-1} R(i)$$

is maximized if $k = \frac{m}{2}$ when m is even and $k = \frac{m+1}{2}$ when m is odd. Thus, if m is even, we obtain

$$R(m) \leq cm + \frac{2}{m} \sum_{m/2}^{m-1} R(i)$$

$$\begin{aligned} & \leq cm + \frac{8c}{m}\sum_{m/2}^{m-1} i \\ & < 4cm \\ \text{If } m \text{ is odd, } R(m) & \leq cm + \frac{2}{m}\sum_{(m+1)/2}^{m-1} R(i) \\ & \leq cm + \frac{8c}{m}\sum_{(m+1)/2}^{m-1} \\ & < 4cm \end{aligned}$$

Since $T_A(n) \leq R(n)$, it follows that $T_A(n) \leq 4cn$, and so $T_A(n)$ is $O(n)$. □

The space needed by Select1 is $O(1)$.

Algorithm 3.15 is a randomized version of QuickSort in which the partition element is chosen from the array elements randomly with equal probability. The same technique can be applied to Select1 and the partition element can be chosen to be a random array element. The resulting randomized Las Vegas algorithm (call it RSelect) has an expected time of $O(n)$ (where the expectation is over the space of randomizer outputs) on *any input*. The proof of this expected time is the same as in Theorem 3.3.

3.6.1 A Worst-Case Optimal Algorithm

By choosing the partitioning element v more carefully, we can obtain a selection algorithm with worst-case complexity $O(n)$. To obtain such an algorithm, v must be chosen so that at least some fraction of the elements is smaller than v and at least some (other) fraction of elements is greater than v. Such a selection of v can be made using the median of medians (mm) rule. In this rule the n elements are divided into $\lfloor n/r \rfloor$ groups of r elements each (for some $r, r > 1$). The remaining $n - r\lfloor n/r \rfloor$ elements are not used. The median m_i of each of these $\lfloor n/r \rfloor$ groups is found. Then, the median mm of the m_i's, $1 \leq i \leq \lfloor n/r \rfloor$, is found. The median mm is used as the partitioning element. Figure 3.5 illustrates the m_i's and mm when $n = 35$ and $r = 7$. The five groups of elements are $B_i, 1 \leq i \leq 5$. The seven elements in each group have been arranged into nondecreasing order down the column. The middle elements are the m_i's. The columns have been arranged in nondecreasing order of m_i. Hence, the m_i corresponding to column 3 is mm.

Since the median of r elements is the $\lceil r/2 \rceil$th-smallest element, it follows (see Figure 3.5) that at least $\lceil \lfloor n/r \rfloor /2 \rceil$ of the m_i's are less than or equal to mm and at least $\lfloor n/r \rfloor - \lceil \lfloor n/r \rfloor /2 \rceil + 1 \geq \lceil \lfloor n/r \rfloor /2 \rceil$ of the m_i's are greater than or equal to mm. Hence, at least $\lceil r/2 \rceil \lceil \lfloor n/r \rfloor /2 \rceil$ elements are less than

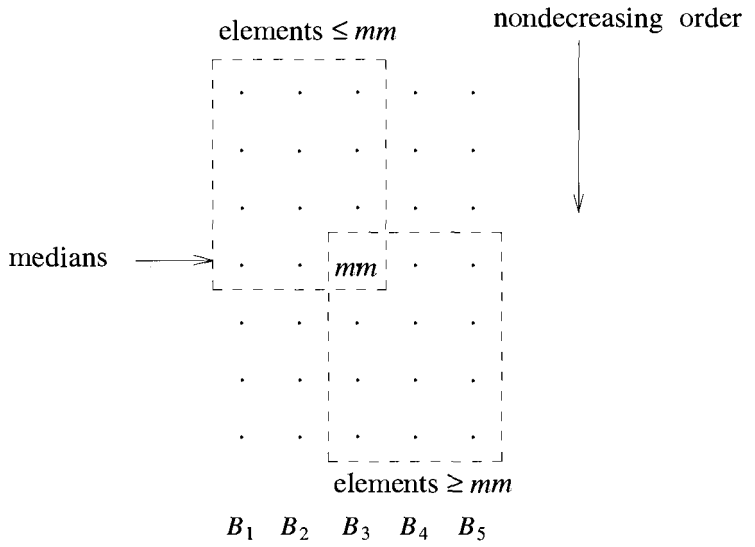

Figure 3.5 The median of medians when $r = 7$, $n = 35$

or equal to (or greater than or equal to) mm. When $r = 5$, this quantity is at least $1.5\lfloor n/5 \rfloor$. Thus, if we use the median of medians rule with $r = 5$ to select $v = mm$, we are assured that at least $1.5\lfloor n/5 \rfloor$ elements will be greater than or equal to v. This in turn implies that at most $n - 1.5\lfloor n/5 \rfloor \leq .7n + 1.2$ elements are less than v. Also, at most $.7n + 1.2$ elements are greater than v. Thus, the median of medians rule satisfies our earlier requirement on v.

The algorithm to select the kth-smallest element uses the median of medians rule to determine a partitioning element. This element is computed by a recursive application of the selection algorithm. A high-level description of the new selection algorithm appears as Select2 (Algorithm 3.18). Select2 can now be analyzed for any given r. First, let us consider the case in which $r = 5$ and all elements in $a[\ ]$ are distinct. Let $T(n)$ be the worst-case time requirement of Select2 when invoked with $up - low + 1 = n$. Lines 4 to 9 and 11 to 12 require at most $O(n)$ time (note that since $r = 5$ is fixed, each $m[i]$ (lines 8 and 9) can be found in $O(1)$ time). The time for line 10 is $T(n/5)$. Let S and R, respectively, denote the elements $a[low : j-1]$ and $a[j+1 : up]$. We see that $|S|$ and $|R|$ are at most $.7n + 1.2$, which is no more than $3n/4$ for $n \geq 24$. So, the time for lines 13 to 16 is at most $T(3n/4)$ when $n \geq 24$. Hence, for $n \geq 24$, we obtain

```
1   Algorithm Select2(a, k, low, up)
2   // Find the k-th smallest in a[low : up].
3   {
4       n := up − low + 1;
5       if (n ≤ r) then sort a[low : up] and return the k-th element;
6       Divide a[low : up] into n/r subsets of size r each;
7       Ignore excess elements;
8       Let m[i], 1 ≤ i ≤ (n/r) be the set of medians of
9       the above n/r subsets.
10      v := Select2(m, ⌈(n/r)/2⌉, 1, n/r);
11      Partition a[low : up] using v as the partition element;
12      Assume that v is at position j;
13      if (k = (j − low + 1)) then return v;
14      elseif (k < (j − low + 1)) then
15                  return Select2(a, k, low, j − 1);
16          else return Select2(a, k − (j − low + 1), j + 1, up);
17  }
```

Algorithm 3.18 Selection pseudocode using the median of medians rule

$$T(n) \leq T(n/5) + T(3n/4) + cn \tag{3.9}$$

where c is chosen sufficiently large that

$$T(n) \leq cn \quad \text{for } n \leq 24$$

A proof by induction easily establishes that $T(n) \leq 20cn$ for $n \geq 1$. Algorithm Select2 with $r = 5$ is a linear time algorithm for the selection problem on distinct elements! The exercises examine other values of r that also yield this behavior. Let us now see what happens when the elements of $a[\,]$ are not all distinct. In this case, following a use of Partition (line 11), the size of S or R may be more than $.7n + 1.2$ as some elements equal to v may appear in both S and R. One way to handle the situation is to partition $a[\,]$ into three sets U, S, and R such that U contains all elements equal to v, S has all elements smaller than v, and R has the remainder. Lines 11 to 16 become:

```
Partition a[ ] into U, S, and R as above.
if (|S| ≥ k) then return Select2(a, k, low, low + |S| − 1);
else if ((|S| + |U|) ≥ k) then return v;
     else return Select2(a, k − |S| − |U|, low + |S| + |U|, up);
```

When this is done, the recurrence (3.9) is still valid as $|S|$ and $|R|$ are $\leq .7n + 1.2$. Hence, the new Select2 will be of linear complexity even when elements are not distinct.

Another way to handle the case of nondistinct elements is to use a different r. To see why a different r is needed, let us analyze Select2 with $r = 5$ and nondistinct elements. Consider the case when $.7n+1.2$ elements are less than v and the remaining elements are equal to v. An examination of Partition reveals that at most half the remaining elements may be in S. We can verify that this is the worst case. Hence, $|S| \leq .7n + 1.2 + (.3n - 1.2)/2 = .85n + .6$. Similarly, $|R| \leq .85n + .6$. Since, the total number of elements involved in the two recursive calls (in lines 10 and 15 or 16) is now $1.05n + .6 \geq n$, the complexity of Select2 is not $O(n)$. If we try $r = 9$, then at least $2.5\lfloor n/9 \rfloor$ elements will be less than or equal to v and at least this many will be greater than or equal to v. Hence, the size of S and R will be at most $n - 2.5\lfloor n/9 \rfloor + 1/2(2.5\lfloor n/9 \rfloor) = n - 1.25\lfloor n/9 \rfloor \leq 31/36n + 1.25 \leq 63n/72$ for $n \geq 90$. Hence, we obtain the recurrence

$$T(n) \leq \begin{cases} T(n/9) + T(63n/72) + c_1 n & n \geq 90 \\ c_1 n & n < 90 \end{cases}$$

where c_1 is a suitable constant. An inductive argument shows that $T(n) \leq 72c_1 n, n \geq 1$. Other suitable values of r are obtained in the exercises.

As far as the additional space needed by Select2 is concerned, we see that space is needed for the recursion stack. The recursive call from line 15 or 16 is easily eliminated as this call is the last statement executed in Select2. Hence, stack space is needed only for the recursion from line 10. The maximum depth of recursion is log n. The recursion stack should be capable of handling this depth. In addition to this stack space, space is needed only for some simple variables.

3.6.2 Implementation of Select2

Before attempting to write a pseudocode algorithm implementing Select2, we need to decide how the median of a set of size r is to be found and where we are going to store the $\lfloor n/r \rfloor$ medians of lines 8 and 9. Since, we expect to be using a small r (say $r = 5$ or 9), an efficient way to find the median of r elements is to sort them using InsertionSort(a, i, j). This algorithm is a modification of Algorithm 3.9 to sort $a[i : j]$. The median is now the middle element in $a[i : j]$. A convenient place to store these medians is at the front of the array. Thus, if we are finding the kth-smallest element in $a[low : up]$, then the elements can be rearranged so that the medians are $a[low]$, $a[low+1]$, $a[low+2]$, and so on. This makes it easy to implement line 10 as a selection on consecutive elements of $a[\,]$. Function Select2 (Algorithm 3.19) results from the above discussion and the replacement of the recursive calls of lines 15 and 16 by equivalent code to restart the algorithm.

```
1   Algorithm Select2(a, k, low, up)
2   // Return i such that a[i] is the kth-smallest element in
3   // a[low : up]; r is a global variable as described in the text.
4   {
5       repeat
6       {
7           n := up − low + 1; // Number of elements
8           if (n ≤ r) then
9           {
10              InsertionSort(a, low, up);
11              return low + k − 1;
12          }
13          for i := 1 to ⌊n/r⌋ do
14          {
15              InsertionSort(a, low + (i − 1) ∗ r, low + i ∗ r − 1);
16              // Collect medians in the front part of a[low : up].
17              Interchange(a, low + i − 1,
18                  low + (i − 1) ∗ r + ⌈r/2⌉ − 1);
19          }
20          j := Select2(a, ⌈⌊n/r⌋/2⌉, low, low + ⌊n/r⌋ − 1); // mm
21          Interchange(a, low, j);
22          j := Partition(a, low, up + 1);
23          if (k = (j − low + 1)) then return j;
24          else if (k < (j − low + 1)) then up := j − 1;
25              else
26              {
27                  k := k − (j − low + 1); low := j + 1;
28              }
29      } until (false);
30  }
```

Algorithm 3.19 Algorithm Select2

An alternative to moving the medians to the front of the array $a[low : up]$ (as in the Interchange statement within the **for** loop) is to delete this statement and use the fact that the medians are located at $low + (i-1)r + \lceil r/2 \rceil - 1, 1 \leq i \leq \lfloor n/r \rfloor$. Hence, Select2, Partition, and InsertionSort need to be rewritten to work on arrays for which the interelement distance is $b, b \geq 1$. At the start of the algorithm, all elements are a distance of one apart, i.e., $a[1], a[2], \ldots, a[n]$. On the first call of Select2 we wish to use only elements that are r apart starting with $a[\lceil r/2 \rceil]$. At the next level of recursion, the elements will be r^2 apart and so on. This idea is developed further in the exercises. We refer to arrays with an interelement distance of b as *b-spaced arrays.*

Algorithms Select1 (Algorithm 3.17) and Select2 (Algorithm 3.19) were implemented and run on a SUN Sparcstation 10/30. Table 3.8 summarizes the experimental results obtained. Times shown are in milliseconds. These algorithms were tested on random integers in the range [0, 1000] and the average execution times (over 500 input sets) were computed. Select1 outperforms Select2 on random inputs. But if the input is already sorted (or nearly sorted), Select2 can be expected to be superior to Select1.

n	1,000	2,000	3,000	4,000	5,000
Select1	7.42	23.50	30.44	39.24	52.36
Select2	49.54	104.02	174.54	233.56	288.64
n	6,000	7,000	8,000	9,000	10,000
Select1	70.88	83.14	95.00	101.32	111.92
Select2	341.34	414.06	476.98	532.30	604.40

Table 3.8 Comparison of Select1 and Select2 on random inputs

EXERCISES

1. Rewrite Select2, Partition, and InsertionSort using b-spaced arrays.

2. (a) Assume that Select2 is to be used only when all elements in a are distinct. Which of the following values of r guarantee $O(n)$ worst-case performance: $r = 3, 5, 7, 9$, and 11? Prove your answers.

   (b) Do you expect the computing time of Select2 to increase or decrease if a larger (but still eligible) choice for r is made? Why?

3. Do Exercise 2 for the case in which a is not restricted to distinct elements. Let $r = 7, 9, 11, 13$, and 15 in part (a).

4. Section 3.6 describes an alternative way to handle the situation when $a[\,]$ is not restricted to distinct elements. Using the partitioning element v, $a[\,]$ is divided into three subsets. Write algorithms corresponding to Select1 and Select2 using this idea. Using your new version of Select2 show that the worst-case computing time is $O(n)$ even when $r = 5$.

5. Determine optimal r values for worst-case and average performances of function Select2.

6. [Shamos] Let $x[1 : n]$ and $y[1 : n]$ contain two sets of integers, each sorted in nondecreasing order. Write an algorithm that finds the median of the $2n$ combined elements. What is the time complexity of your algorithm? (*Hint*: Use binary search.)

7. Let S be a (not necessarily sorted) sequence of n keys. A key k in S is said to be an *approximate median* of S if $|\{k' \in S : \ k' < k\}| \geq \frac{n}{4}$ and $|\{k' \in S : \ k' > k\}| \geq \frac{n}{4}$. Devise an $O(n)$ time algorithm to find all the approximate medians of S.

8. Input are a sequence S of n distinct keys, not necessarily in sorted order, and two integers m_1 and m_2 $(1 \leq m_1, m_2 \leq n)$. For any x in S, we define the *rank* of x in S to be $|\{k \in S : \ k \leq x\}|$. Show how to output all the keys of S whose ranks fall in the interval $[m_1, m_2]$ in $O(n)$ time.

9. The kth *quantiles* of an n-element set are the $k - 1$ elements from the set that divide the sorted set into k equal-sized sets. Give an $O(n \log k)$ time algorithm to list the kth quantiles of a set.

10. Input is a (not necessarily sorted) sequence $S = k_1, k_2, \ldots, k_n$ of n arbitrary numbers. Consider the collection C of n^2 numbers of the form $\min\{k_i, k_j\}$, for $1 \leq i, j \leq n$. Present an $O(n)$-time and $O(n)$-space algorithm to find the median of C.

11. Given two vectors $X = (x_1, \ldots, x_n)$ and $Y = (y_1, \ldots, y_n)$, $X < Y$ if there exists an $i, 1 \leq i \leq n$, such that $x_j = y_j$ for $1 \leq j < i$ and $x_i < y_i$. Given m vectors each of size n, write an algorithm that determines the minimum vector. Analyze the time complexity of your algorithm.

12. Present an $O(1)$ time Monte Carlo algorithm to find the median of an array of n numbers. The answer output should be correct with probability $\geq \frac{1}{n}$.