couple of definitions will help you apply the 0-1 sorting lemma. We say that an area of an array is ***clean*** if we know that it contains either all 0s or all 1s. Otherwise, the area might contain mixed 0s and 1s, and it is ***dirty***. From here on, assume that the input array contains only 0s and 1s, and that we can treat it as an array with r rows and s columns.

d. Prove that after steps 1–3, the array consists of some clean rows of 0s at the top, some clean rows of 1s at the bottom, and at most s dirty rows between them.

e. Prove that after step 4, the array, read in column-major order, starts with a clean area of 0s, ends with a clean area of 1s, and has a dirty area of at most s^2 elements in the middle.

f. Prove that steps 5–8 produce a fully sorted 0-1 output. Conclude that columnsort correctly sorts all inputs containing arbitrary values.

g. Now suppose that s does not divide r. Prove that after steps 1–3, the array consists of some clean rows of 0s at the top, some clean rows of 1s at the bottom, and at most $2s - 1$ dirty rows between them. How large must r be, compared with s, for columnsort to correctly sort when s does not divide r?

h. Suggest a simple change to step 1 that allows us to maintain the requirement that $r \geq 2s^2$ even when s does not divide r, and prove that with your change, columnsort correctly sorts.

Chapter notes

The decision-tree model for studying comparison sorts was introduced by Ford and Johnson [110]. Knuth's comprehensive treatise on sorting [211] covers many variations on the sorting problem, including the information-theoretic lower bound on the complexity of sorting given here. Ben-Or [39] studied lower bounds for sorting using generalizations of the decision-tree model.

Knuth credits H. H. Seward with inventing counting sort in 1954, as well as with the idea of combining counting sort with radix sort. Radix sorting starting with the least significant digit appears to be a folk algorithm widely used by operators of mechanical card-sorting machines. According to Knuth, the first published reference to the method is a 1929 document by L. J. Comrie describing punched-card equipment. Bucket sorting has been in use since 1956, when the basic idea was proposed by E. J. Isaac and R. C. Singleton [188].

Munro and Raman [263] give a stable sorting algorithm that performs $O(n^{1+\epsilon})$ comparisons in the worst case, where $0 < \epsilon \leq 1$ is any fixed constant. Although

any of the $O(n \lg n)$-time algorithms make fewer comparisons, the algorithm by Munro and Raman moves data only $O(n)$ times and operates in place.

The case of sorting n b-bit integers in $o(n \lg n)$ time has been considered by many researchers. Several positive results have been obtained, each under slightly different assumptions about the model of computation and the restrictions placed on the algorithm. All the results assume that the computer memory is divided into addressable b-bit words. Fredman and Willard [115] introduced the fusion tree data structure and used it to sort n integers in $O(n \lg n / \lg \lg n)$ time. This bound was later improved to $O(n \sqrt{\lg n})$ time by Andersson [16]. These algorithms require the use of multiplication and several precomputed constants. Andersson, Hagerup, Nilsson, and Raman [17] have shown how to sort n integers in $O(n \lg \lg n)$ time without using multiplication, but their method requires storage that can be unbounded in terms of n. Using multiplicative hashing, we can reduce the storage needed to $O(n)$, but then the $O(n \lg \lg n)$ worst-case bound on the running time becomes an expected-time bound. Generalizing the exponential search trees of Andersson [16], Thorup [335] gave an $O(n(\lg \lg n)^2)$-time sorting algorithm that does not use multiplication or randomization, and it uses linear space. Combining these techniques with some new ideas, Han [158] improved the bound for sorting to $O(n \lg \lg n \lg \lg \lg n)$ time. Although these algorithms are important theoretical breakthroughs, they are all fairly complicated and at the present time seem unlikely to compete with existing sorting algorithms in practice.

The columnsort algorithm in Problem 8-7 is by Leighton [227].

9 Medians and Order Statistics

The ith ***order statistic*** of a set of n elements is the ith smallest element. For example, the ***minimum*** of a set of elements is the first order statistic ($i = 1$), and the ***maximum*** is the nth order statistic ($i = n$). A ***median***, informally, is the "halfway point" of the set. When n is odd, the median is unique, occurring at $i = (n + 1)/2$. When n is even, there are two medians, occurring at $i = n/2$ and $i = n/2+1$. Thus, regardless of the parity of n, medians occur at $i = \lfloor (n + 1)/2 \rfloor$ (the ***lower median***) and $i = \lceil (n + 1)/2 \rceil$ (the ***upper median***). For simplicity in this text, however, we consistently use the phrase "the median" to refer to the lower median.

This chapter addresses the problem of selecting the ith order statistic from a set of n distinct numbers. We assume for convenience that the set contains distinct numbers, although virtually everything that we do extends to the situation in which a set contains repeated values. We formally specify the ***selection problem*** as follows:

Input: A set A of n (distinct) numbers and an integer i, with $1 \le i \le n$.

Output: The element $x \in A$ that is larger than exactly $i - 1$ other elements of A.

We can solve the selection problem in $O(n \lg n)$ time, since we can sort the numbers using heapsort or merge sort and then simply index the ith element in the output array. This chapter presents faster algorithms.

In Section 9.1, we examine the problem of selecting the minimum and maximum of a set of elements. More interesting is the general selection problem, which we investigate in the subsequent two sections. Section 9.2 analyzes a practical randomized algorithm that achieves an $O(n)$ expected running time, assuming distinct elements. Section 9.3 contains an algorithm of more theoretical interest that achieves the $O(n)$ running time in the worst case.

9.1 Minimum and maximum

How many comparisons are necessary to determine the minimum of a set of n elements? We can easily obtain an upper bound of $n - 1$ comparisons: examine each element of the set in turn and keep track of the smallest element seen so far. In the following procedure, we assume that the set resides in array A, where $A.length = n$.

```
MINIMUM(A)
1  min = A[1]
2  for i = 2 to A.length
3      if min > A[i]
4          min = A[i]
5  return min
```

We can, of course, find the maximum with $n - 1$ comparisons as well.

Is this the best we can do? Yes, since we can obtain a lower bound of $n - 1$ comparisons for the problem of determining the minimum. Think of any algorithm that determines the minimum as a tournament among the elements. Each comparison is a match in the tournament in which the smaller of the two elements wins. Observing that every element except the winner must lose at least one match, we conclude that $n - 1$ comparisons are necessary to determine the minimum. Hence, the algorithm MINIMUM is optimal with respect to the number of comparisons performed.

Simultaneous minimum and maximum

In some applications, we must find both the minimum and the maximum of a set of n elements. For example, a graphics program may need to scale a set of (x, y) data to fit onto a rectangular display screen or other graphical output device. To do so, the program must first determine the minimum and maximum value of each coordinate.

At this point, it should be obvious how to determine both the minimum and the maximum of n elements using $\Theta(n)$ comparisons, which is asymptotically optimal: simply find the minimum and maximum independently, using $n - 1$ comparisons for each, for a total of $2n - 2$ comparisons.

In fact, we can find both the minimum and the maximum using at most $3\lfloor n/2 \rfloor$ comparisons. We do so by maintaining both the minimum and maximum elements seen thus far. Rather than processing each element of the input by comparing it against the current minimum and maximum, at a cost of 2 comparisons per element,

we process elements in pairs. We compare pairs of elements from the input first *with each other*, and then we compare the smaller with the current minimum and the larger to the current maximum, at a cost of 3 comparisons for every 2 elements.

How we set up initial values for the current minimum and maximum depends on whether n is odd or even. If n is odd, we set both the minimum and maximum to the value of the first element, and then we process the rest of the elements in pairs. If n is even, we perform 1 comparison on the first 2 elements to determine the initial values of the minimum and maximum, and then process the rest of the elements in pairs as in the case for odd n.

Let us analyze the total number of comparisons. If n is odd, then we perform $3\lfloor n/2 \rfloor$ comparisons. If n is even, we perform 1 initial comparison followed by $3(n-2)/2$ comparisons, for a total of $3n/2 - 2$. Thus, in either case, the total number of comparisons is at most $3\lfloor n/2 \rfloor$.

Exercises

9.1-1

Show that the second smallest of n elements can be found with $n + \lceil \lg n \rceil - 2$ comparisons in the worst case. (*Hint:* Also find the smallest element.)

9.1-2 ⋆

Prove the lower bound of $\lceil 3n/2 \rceil - 2$ comparisons in the worst case to find both the maximum and minimum of n numbers. (*Hint:* Consider how many numbers are potentially either the maximum or minimum, and investigate how a comparison affects these counts.)

9.2 Selection in expected linear time

The general selection problem appears more difficult than the simple problem of finding a minimum. Yet, surprisingly, the asymptotic running time for both problems is the same: $\Theta(n)$. In this section, we present a divide-and-conquer algorithm for the selection problem. The algorithm RANDOMIZED-SELECT is modeled after the quicksort algorithm of Chapter 7. As in quicksort, we partition the input array recursively. But unlike quicksort, which recursively processes both sides of the partition, RANDOMIZED-SELECT works on only one side of the partition. This difference shows up in the analysis: whereas quicksort has an expected running time of $\Theta(n \lg n)$, the expected running time of RANDOMIZED-SELECT is $\Theta(n)$, assuming that the elements are distinct.

RANDOMIZED-SELECT uses the procedure RANDOMIZED-PARTITION introduced in Section 7.3. Thus, like RANDOMIZED-QUICKSORT, it is a randomized algorithm, since its behavior is determined in part by the output of a random-number generator. The following code for RANDOMIZED-SELECT returns the ith smallest element of the array $A[p..r]$.

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p == r
2      return A[p]
3  q = RANDOMIZED-PARTITION(A, p, r)
4  k = q - p + 1
5  if i == k            // the pivot value is the answer
6      return A[q]
7  elseif i < k
8      return RANDOMIZED-SELECT(A, p, q - 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i - k)
```

The RANDOMIZED-SELECT procedure works as follows. Line 1 checks for the base case of the recursion, in which the subarray $A[p..r]$ consists of just one element. In this case, i must equal 1, and we simply return $A[p]$ in line 2 as the ith smallest element. Otherwise, the call to RANDOMIZED-PARTITION in line 3 partitions the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$ such that each element of $A[p..q-1]$ is less than or equal to $A[q]$, which in turn is less than each element of $A[q+1..r]$. As in quicksort, we will refer to $A[q]$ as the ***pivot*** element. Line 4 computes the number k of elements in the subarray $A[p..q]$, that is, the number of elements in the low side of the partition, plus one for the pivot element. Line 5 then checks whether $A[q]$ is the ith smallest element. If it is, then line 6 returns $A[q]$. Otherwise, the algorithm determines in which of the two subarrays $A[p..q-1]$ and $A[q+1..r]$ the ith smallest element lies. If $i < k$, then the desired element lies on the low side of the partition, and line 8 recursively selects it from the subarray. If $i > k$, however, then the desired element lies on the high side of the partition. Since we already know k values that are smaller than the ith smallest element of $A[p..r]$—namely, the elements of $A[p..q]$—the desired element is the $(i-k)$th smallest element of $A[q+1..r]$, which line 9 finds recursively. The code appears to allow recursive calls to subarrays with 0 elements, but Exercise 9.2-1 asks you to show that this situation cannot happen.

The worst-case running time for RANDOMIZED-SELECT is $\Theta(n^2)$, even to find the minimum, because we could be extremely unlucky and always partition around the largest remaining element, and partitioning takes $\Theta(n)$ time. We will see that

the algorithm has a linear expected running time, though, and because it is randomized, no particular input elicits the worst-case behavior.

To analyze the expected running time of RANDOMIZED-SELECT, we let the running time on an input array $A[p..r]$ of n elements be a random variable that we denote by $T(n)$, and we obtain an upper bound on $\mathrm{E}[T(n)]$ as follows. The procedure RANDOMIZED-PARTITION is equally likely to return any element as the pivot. Therefore, for each k such that $1 \le k \le n$, the subarray $A[p..q]$ has k elements (all less than or equal to the pivot) with probability $1/n$. For $k = 1, 2, \ldots, n$, we define indicator random variables X_k where

$$X_k = \mathrm{I}\{\text{the subarray } A[p..q] \text{ has exactly } k \text{ elements}\} ,$$

and so, assuming that the elements are distinct, we have

$$\mathrm{E}[X_k] = 1/n . \tag{9.1}$$

When we call RANDOMIZED-SELECT and choose $A[q]$ as the pivot element, we do not know, a priori, if we will terminate immediately with the correct answer, recurse on the subarray $A[p..q-1]$, or recurse on the subarray $A[q+1..r]$. This decision depends on where the ith smallest element falls relative to $A[q]$. Assuming that $T(n)$ is monotonically increasing, we can upper-bound the time needed for the recursive call by the time needed for the recursive call on the largest possible input. In other words, to obtain an upper bound, we assume that the ith element is always on the side of the partition with the greater number of elements. For a given call of RANDOMIZED-SELECT, the indicator random variable X_k has the value 1 for exactly one value of k, and it is 0 for all other k. When $X_k = 1$, the two subarrays on which we might recurse have sizes $k-1$ and $n-k$. Hence, we have the recurrence

$$\begin{aligned} T(n) &\le \sum_{k=1}^{n} X_k \cdot (T(\max(k-1, n-k)) + O(n)) \\ &= \sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n) . \end{aligned}$$

Taking expected values, we have

$$\begin{aligned}
&\mathrm{E}[T(n)] \\
&\le \mathrm{E}\left[\sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n)\right] \\
&= \sum_{k=1}^{n} \mathrm{E}[X_k \cdot T(\max(k-1, n-k))] + O(n) && \text{(by linearity of expectation)} \\
&= \sum_{k=1}^{n} \mathrm{E}[X_k] \cdot \mathrm{E}[T(\max(k-1, n-k))] + O(n) && \text{(by equation (C.24))} \\
&= \sum_{k=1}^{n} \frac{1}{n} \cdot \mathrm{E}[T(\max(k-1, n-k))] + O(n) && \text{(by equation (9.1))} \ .
\end{aligned}$$

In order to apply equation (C.24), we rely on X_k and $T(\max(k-1, n-k))$ being independent random variables. Exercise 9.2-2 asks you to justify this assertion.

Let us consider the expression $\max(k-1, n-k)$. We have

$$\max(k-1, n-k) = \begin{cases} k-1 & \text{if } k > \lceil n/2 \rceil \ , \\ n-k & \text{if } k \le \lceil n/2 \rceil \ . \end{cases}$$

If n is even, each term from $T(\lceil n/2 \rceil)$ up to $T(n-1)$ appears exactly twice in the summation, and if n is odd, all these terms appear twice and $T(\lfloor n/2 \rfloor)$ appears once. Thus, we have

$$\mathrm{E}[T(n)] \le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} \mathrm{E}[T(k)] + O(n) \ .$$

We show that $\mathrm{E}[T(n)] = O(n)$ by substitution. Assume that $\mathrm{E}[T(n)] \le cn$ for some constant c that satisfies the initial conditions of the recurrence. We assume that $T(n) = O(1)$ for n less than some constant; we shall pick this constant later. We also pick a constant a such that the function described by the $O(n)$ term above (which describes the non-recursive component of the running time of the algorithm) is bounded from above by an for all $n > 0$. Using this inductive hypothesis, we have

$$\begin{aligned}
\mathrm{E}[T(n)] &\le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + an \\
&= \frac{2c}{n} \left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k \right) + an
\end{aligned}$$

$$
\begin{aligned}
&= \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(\lfloor n/2\rfloor - 1)\,\lfloor n/2\rfloor}{2}\right) + an \\
&\le \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(n/2-2)(n/2-1)}{2}\right) + an \\
&= \frac{2c}{n}\left(\frac{n^2-n}{2} - \frac{n^2/4 - 3n/2 + 2}{2}\right) + an \\
&= \frac{c}{n}\left(\frac{3n^2}{4} + \frac{n}{2} - 2\right) + an \\
&= c\left(\frac{3n}{4} + \frac{1}{2} - \frac{2}{n}\right) + an \\
&\le \frac{3cn}{4} + \frac{c}{2} + an \\
&= cn - \left(\frac{cn}{4} - \frac{c}{2} - an\right) .
\end{aligned}
$$

In order to complete the proof, we need to show that for sufficiently large n, this last expression is at most cn or, equivalently, that $cn/4 - c/2 - an \ge 0$. If we add $c/2$ to both sides and factor out n, we get $n(c/4 - a) \ge c/2$. As long as we choose the constant c so that $c/4 - a > 0$, i.e., $c > 4a$, we can divide both sides by $c/4 - a$, giving

$$n \ge \frac{c/2}{c/4 - a} = \frac{2c}{c - 4a} .$$

Thus, if we assume that $T(n) = O(1)$ for $n < 2c/(c-4a)$, then $\mathrm{E}[T(n)] = O(n)$. We conclude that we can find any order statistic, and in particular the median, in expected linear time, assuming that the elements are distinct.

Exercises

9.2-1

Show that RANDOMIZED-SELECT never makes a recursive call to a 0-length array.

9.2-2

Argue that the indicator random variable X_k and the value $T(\max(k-1, n-k))$ are independent.

9.2-3

Write an iterative version of RANDOMIZED-SELECT.

9.2-4

Suppose we use RANDOMIZED-SELECT to select the minimum element of the array $A = \langle 3, 2, 9, 0, 7, 5, 4, 8, 6, 1 \rangle$. Describe a sequence of partitions that results in a worst-case performance of RANDOMIZED-SELECT.

9.3 Selection in worst-case linear time

We now examine a selection algorithm whose running time is $O(n)$ in the worst case. Like RANDOMIZED-SELECT, the algorithm SELECT finds the desired element by recursively partitioning the input array. Here, however, we *guarantee* a good split upon partitioning the array. SELECT uses the deterministic partitioning algorithm PARTITION from quicksort (see Section 7.1), but modified to take the element to partition around as an input parameter.

The SELECT algorithm determines the ith smallest of an input array of $n > 1$ distinct elements by executing the following steps. (If $n = 1$, then SELECT merely returns its only input value as the ith smallest.)

1. Divide the n elements of the input array into $\lfloor n/5 \rfloor$ groups of 5 elements each and at most one group made up of the remaining n mod 5 elements.
2. Find the median of each of the $\lceil n/5 \rceil$ groups by first insertion-sorting the elements of each group (of which there are at most 5) and then picking the median from the sorted list of group elements.
3. Use SELECT recursively to find the median x of the $\lceil n/5 \rceil$ medians found in step 2. (If there are an even number of medians, then by our convention, x is the lower median.)
4. Partition the input array around the median-of-medians x using the modified version of PARTITION. Let k be one more than the number of elements on the low side of the partition, so that x is the kth smallest element and there are $n-k$ elements on the high side of the partition.
5. If $i = k$, then return x. Otherwise, use SELECT recursively to find the ith smallest element on the low side if $i < k$, or the $(i-k)$th smallest element on the high side if $i > k$.

To analyze the running time of SELECT, we first determine a lower bound on the number of elements that are greater than the partitioning element x. Figure 9.1 helps us to visualize this bookkeeping. At least half of the medians found in

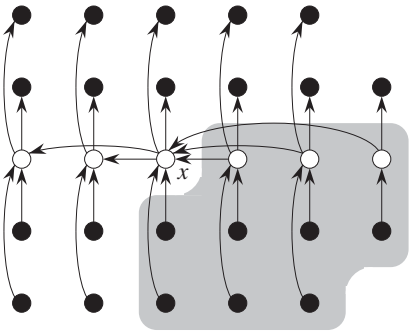

Figure 9.1 Analysis of the algorithm SELECT. The n elements are represented by small circles, and each group of 5 elements occupies a column. The medians of the groups are whitened, and the median-of-medians x is labeled. (When finding the median of an even number of elements, we use the lower median.) Arrows go from larger elements to smaller, from which we can see that 3 out of every full group of 5 elements to the right of x are greater than x, and 3 out of every group of 5 elements to the left of x are less than x. The elements known to be greater than x appear on a shaded background.

step 2 are greater than or equal to the median-of-medians x.[1] Thus, at least half of the $\lceil n/5 \rceil$ groups contribute at least 3 elements that are greater than x, except for the one group that has fewer than 5 elements if 5 does not divide n exactly, and the one group containing x itself. Discounting these two groups, it follows that the number of elements greater than x is at least

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5}\right\rceil \right\rceil - 2\right) \;\geq\; \frac{3n}{10} - 6\,.$$

Similarly, at least $3n/10 - 6$ elements are less than x. Thus, in the worst case, step 5 calls SELECT recursively on at most $7n/10 + 6$ elements.

We can now develop a recurrence for the worst-case running time $T(n)$ of the algorithm SELECT. Steps 1, 2, and 4 take $O(n)$ time. (Step 2 consists of $O(n)$ calls of insertion sort on sets of size $O(1)$.) Step 3 takes time $T(\lceil n/5 \rceil)$, and step 5 takes time at most $T(7n/10 + 6)$, assuming that T is monotonically increasing. We make the assumption, which seems unmotivated at first, that any input of fewer than 140 elements requires $O(1)$ time; the origin of the magic constant 140 will be clear shortly. We can therefore obtain the recurrence

[1] Because of our assumption that the numbers are distinct, all medians except x are either greater than or less than x.

$$T(n) \le \begin{cases} O(1) & \text{if } n < 140 , \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) & \text{if } n \ge 140 . \end{cases}$$

We show that the running time is linear by substitution. More specifically, we will show that $T(n) \le cn$ for some suitably large constant c and all $n > 0$. We begin by assuming that $T(n) \le cn$ for some suitably large constant c and all $n < 140$; this assumption holds if c is large enough. We also pick a constant a such that the function described by the $O(n)$ term above (which describes the non-recursive component of the running time of the algorithm) is bounded above by an for all $n > 0$. Substituting this inductive hypothesis into the right-hand side of the recurrence yields

$$\begin{aligned} T(n) &\le c\lceil n/5 \rceil + c(7n/10 + 6) + an \\ &\le cn/5 + c + 7cn/10 + 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + (-cn/10 + 7c + an) , \end{aligned}$$

which is at most cn if

$$-cn/10 + 7c + an \le 0 . \tag{9.2}$$

Inequality (9.2) is equivalent to the inequality $c \ge 10a(n/(n-70))$ when $n > 70$. Because we assume that $n \ge 140$, we have $n/(n-70) \le 2$, and so choosing $c \ge 20a$ will satisfy inequality (9.2). (Note that there is nothing special about the constant 140; we could replace it by any integer strictly greater than 70 and then choose c accordingly.) The worst-case running time of SELECT is therefore linear.

As in a comparison sort (see Section 8.1), SELECT and RANDOMIZED-SELECT determine information about the relative order of elements only by comparing elements. Recall from Chapter 8 that sorting requires $\Omega(n \lg n)$ time in the comparison model, even on average (see Problem 8-1). The linear-time sorting algorithms in Chapter 8 make assumptions about the input. In contrast, the linear-time selection algorithms in this chapter do not require any assumptions about the input. They are not subject to the $\Omega(n \lg n)$ lower bound because they manage to solve the selection problem without sorting. Thus, solving the selection problem by sorting and indexing, as presented in the introduction to this chapter, is asymptotically inefficient.

Exercises

9.3-1
In the algorithm SELECT, the input elements are divided into groups of 5. Will the algorithm work in linear time if they are divided into groups of 7? Argue that SELECT does not run in linear time if groups of 3 are used.

9.3-2
Analyze SELECT to show that if $n \geq 140$, then at least $\lceil n/4 \rceil$ elements are greater than the median-of-medians x and at least $\lceil n/4 \rceil$ elements are less than x.

9.3-3
Show how quicksort can be made to run in $O(n \lg n)$ time in the worst case, assuming that all elements are distinct.

9.3-4 ⋆
Suppose that an algorithm uses only comparisons to find the ith smallest element in a set of n elements. Show that it can also find the $i - 1$ smaller elements and the $n - i$ larger elements without performing any additional comparisons.

9.3-5
Suppose that you have a "black-box" worst-case linear-time median subroutine. Give a simple, linear-time algorithm that solves the selection problem for an arbitrary order statistic.

9.3-6
The kth ***quantiles*** of an n-element set are the $k - 1$ order statistics that divide the sorted set into k equal-sized sets (to within 1). Give an $O(n \lg k)$-time algorithm to list the kth quantiles of a set.

9.3-7
Describe an $O(n)$-time algorithm that, given a set S of n distinct numbers and a positive integer $k \leq n$, determines the k numbers in S that are closest to the median of S.

9.3-8
Let $X[1..n]$ and $Y[1..n]$ be two arrays, each containing n numbers already in sorted order. Give an $O(\lg n)$-time algorithm to find the median of all $2n$ elements in arrays X and Y.

9.3-9
Professor Olay is consulting for an oil company, which is planning a large pipeline running east to west through an oil field of n wells. The company wants to connect

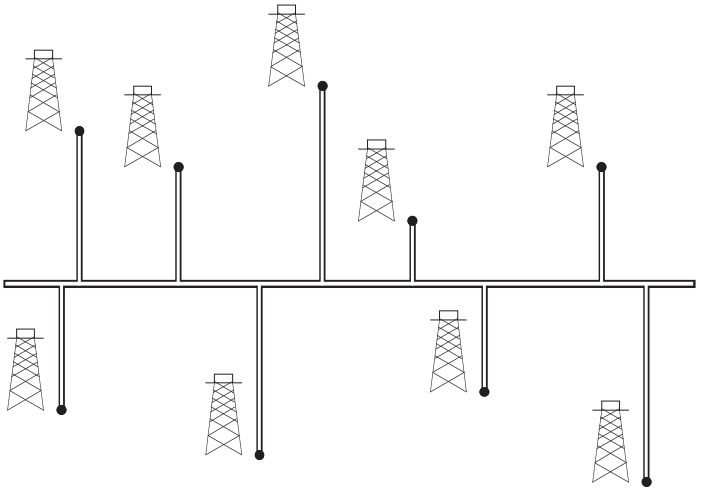

Figure 9.2 Professor Olay needs to determine the position of the east-west oil pipeline that minimizes the total length of the north-south spurs.

a spur pipeline from each well directly to the main pipeline along a shortest route (either north or south), as shown in Figure 9.2. Given the x- and y-coordinates of the wells, how should the professor pick the optimal location of the main pipeline, which would be the one that minimizes the total length of the spurs? Show how to determine the optimal location in linear time.

Problems

9-1 Largest i numbers in sorted order

Given a set of n numbers, we wish to find the i largest in sorted order using a comparison-based algorithm. Find the algorithm that implements each of the following methods with the best asymptotic worst-case running time, and analyze the running times of the algorithms in terms of n and i.

a. Sort the numbers, and list the i largest.

b. Build a max-priority queue from the numbers, and call EXTRACT-MAX i times.

c. Use an order-statistic algorithm to find the ith largest number, partition around that number, and sort the i largest numbers.

9-2 Weighted median

For n distinct elements $x_1, x_2, \ldots, x_n$ with positive weights $w_1, w_2, \ldots, w_n$ such that $\sum_{i=1}^{n} w_i = 1$, the ***weighted (lower) median*** is the element x_k satisfying

$$\sum_{x_i < x_k} w_i < \frac{1}{2}$$

and

$$\sum_{x_i > x_k} w_i \le \frac{1}{2} .$$

For example, if the elements are $0.1, 0.35, 0.05, 0.1, 0.15, 0.05, 0.2$ and each element equals its weight (that is, $w_i = x_i$ for $i = 1, 2, \ldots, 7$), then the median is 0.1, but the weighted median is 0.2.

a. Argue that the median of $x_1, x_2, \ldots, x_n$ is the weighted median of the x_i with weights $w_i = 1/n$ for $i = 1, 2, \ldots, n$.

b. Show how to compute the weighted median of n elements in $O(n \lg n)$ worst-case time using sorting.

c. Show how to compute the weighted median in $\Theta(n)$ worst-case time using a linear-time median algorithm such as SELECT from Section 9.3.

The ***post-office location problem*** is defined as follows. We are given n points $p_1, p_2, \ldots, p_n$ with associated weights $w_1, w_2, \ldots, w_n$. We wish to find a point p (not necessarily one of the input points) that minimizes the sum $\sum_{i=1}^{n} w_i \, d(p, p_i)$, where $d(a, b)$ is the distance between points a and b.

d. Argue that the weighted median is a best solution for the 1-dimensional post-office location problem, in which points are simply real numbers and the distance between points a and b is $d(a, b) = |a - b|$.

e. Find the best solution for the 2-dimensional post-office location problem, in which the points are (x, y) coordinate pairs and the distance between points $a = (x_1, y_1)$ and $b = (x_2, y_2)$ is the ***Manhattan distance*** given by $d(a, b) = |x_1 - x_2| + |y_1 - y_2|$.

9-3 Small order statistics

We showed that the worst-case number $T(n)$ of comparisons used by SELECT to select the ith order statistic from n numbers satisfies $T(n) = \Theta(n)$, but the constant hidden by the Θ-notation is rather large. When i is small relative to n, we can implement a different procedure that uses SELECT as a subroutine but makes fewer comparisons in the worst case.

a. Describe an algorithm that uses $U_i(n)$ comparisons to find the ith smallest of n elements, where

$$U_i(n) = \begin{cases} T(n) & \text{if } i \geq n/2\,, \\ \lfloor n/2 \rfloor + U_i(\lceil n/2 \rceil) + T(2i) & \text{otherwise}\,. \end{cases}$$

(*Hint:* Begin with $\lfloor n/2 \rfloor$ disjoint pairwise comparisons, and recurse on the set containing the smaller element from each pair.)

b. Show that, if $i < n/2$, then $U_i(n) = n + O(T(2i)\lg(n/i))$.

c. Show that if i is a constant less than $n/2$, then $U_i(n) = n + O(\lg n)$.

d. Show that if $i = n/k$ for $k \geq 2$, then $U_i(n) = n + O(T(2n/k)\lg k)$.

9-4 Alternative analysis of randomized selection

In this problem, we use indicator random variables to analyze the RANDOMIZED-SELECT procedure in a manner akin to our analysis of RANDOMIZED-QUICKSORT in Section 7.4.2.

As in the quicksort analysis, we assume that all elements are distinct, and we rename the elements of the input array A as $z_1, z_2, \ldots, z_n$, where z_i is the ith smallest element. Thus, the call RANDOMIZED-SELECT$(A, 1, n, k)$ returns z_k.

For $1 \leq i < j \leq n$, let

$$X_{ijk} = \mathrm{I}\{z_i \text{ is compared with } z_j \text{ sometime during the execution of the algorithm to find } z_k\}\,.$$

a. Give an exact expression for $\mathrm{E}[X_{ijk}]$. (*Hint:* Your expression may have different values, depending on the values of i, j, and k.)

b. Let X_k denote the total number of comparisons between elements of array A when finding z_k. Show that

$$\mathrm{E}[X_k] \leq 2\left(\sum_{i=1}^{k}\sum_{j=k}^{n}\frac{1}{j-i+1} + \sum_{j=k+1}^{n}\frac{j-k-1}{j-k+1} + \sum_{i=1}^{k-2}\frac{k-i-1}{k-i+1}\right)\,.$$

c. Show that $\mathrm{E}[X_k] \leq 4n$.

d. Conclude that, assuming all elements of array A are distinct, RANDOMIZED-SELECT runs in expected time $O(n)$.

Chapter notes

The worst-case linear-time median-finding algorithm was devised by Blum, Floyd, Pratt, Rivest, and Tarjan [50]. The fast randomized version is due to Hoare [169]. Floyd and Rivest [108] have developed an improved randomized version that partitions around an element recursively selected from a small sample of the elements.

It is still unknown exactly how many comparisons are needed to determine the median. Bent and John [41] gave a lower bound of $2n$ comparisons for median finding, and Schönhage, Paterson, and Pippenger [302] gave an upper bound of $3n$. Dor and Zwick have improved on both of these bounds. Their upper bound [93] is slightly less than $2.95n$, and their lower bound [94] is $(2 + \epsilon)n$, for a small positive constant ϵ, thereby improving slightly on related work by Dor et al. [92]. Paterson [272] describes some of these results along with other related work.

III Data Structures