5

Selection and Adversary Arguments

5.1 Introduction

In this chapter we study several problems that can be grouped under the general name, *selection*. Finding the median element of a set is a well-known example. Besides finding algorithms to solve the problems efficiently, we also explore *lower bounds* for the problems. We introduce a widely applicable technique known as *adversary arguments* for establishing lower bounds.

5.1.1 The Selection Problem

Suppose E is an array containing n elements with keys from some linearly ordered set, and let k be an integer such that $1 \leq k \leq n$. The *selection* problem is the problem of finding an element with the kth smallest key in E. Such an element is said to *have rank* k. As with most of the sorting algorithms we studied, we will assume that the only operations that may be performed on the keys are comparisons of pairs of keys (and copying or moving elements). In this chapter keys and elements are considered identical, because our focus is on the number of key comparisons, and we are usually not concerned with element movement. Also, when storing keys in an array, we will use positions $1, \ldots, n$, to agree with common ranking terminology, rather than $0, \ldots, n-1$. Position 0 of the array is simply left unused.

In Chapter 1 we solved the selection problem for the case $k = n$, for that problem is simply to find the largest key. We considered a straightforward algorithm that did $n-1$ key comparisons, and we proved that no algorithm could do fewer. The dual case for $k = 1$, that is, finding the smallest key, can be solved similarly.

Another very common instance of the selection problem is the case where $k = \lceil n/2 \rceil$, that is, where we want to find the middle, or *median*, element. The median is helpful for interpreting very large sets of data, such as the income of all people in a particular country, or in a particular profession, the price of houses, or scores on college entrance tests. Instead of including the whole set of data, news reports, for example, summarize by giving us the mean (average) or median. It is easy to compute the average of n numbers in $\Theta(n)$ time. How can we compute the median efficiently?

Of course, all instances of the selection problem can be solved by sorting E; then, for whatever rank k we are interested in, $E[k]$ would be the answer. Sorting requires $\Theta(n \log n)$ key comparisons, and we have just observed that for some values of k, the selection problem can be solved in linear time. Finding the median seems, intuitively, to be the hardest instance of the selection problem. Can we find the median in linear time? Or can we establish a lower bound for median finding that is more than linear, maybe $\Theta(n \log n)$? We answer these questions in this chapter, and we sketch an algorithm for the general selection problem.

5.1.2 Lower Bounds

So far we have used the decision tree as our main technique to establish lower bounds. Recall that the internal nodes of the decision tree for an algorithm represent the comparisons the algorithm performs, and the leaves represent the outputs. (For the search problem in Section 1.6, the internal nodes also represented outputs.) The number of comparisons

done in the worst case is the height of the tree; the height is at least $\lceil \lg L \rceil$, where L is the number of leaves.

In Section 1.6 we used decision trees to get the worst-case lower bound of $\lceil \lg(n+1) \rceil$ for the search problem. That is exactly the number of comparisons done by Binary Search, so a decision tree argument gave us the best possible lower bound. In Chapter 4 we used decision trees to get a lower bound of $\lceil \lg n! \rceil$, or roughly $\lceil n \lg n - 1.5n \rceil$, for sorting. There are algorithms whose performance is very close to this lower bound, so once again, a decision tree argument gave a very strong result. However, decision tree arguments do not work very well for the selection problem.

A decision tree for the selection problem must have at least n leaves because any one of the n keys in the set may be the output, that is, the kth smallest. Thus we can conclude that the height of the tree (and the number of comparisons done in the worst case) is at least $\lceil \lg n \rceil$. But this is not a good lower bound; we already know that even the easy case of finding the largest key requires at least $n - 1$ comparisons. What's wrong with the decision tree argument? In a decision tree for an algorithm that finds the largest key, some outputs appear at more than one leaf, and there will in fact be more than n leaves. To see this, Exercise 5.1 asks you to draw the decision tree for FindMax (Algorithm 1.3) with $n = 4$. The decision tree argument fails to give a good lower bound because we don't have an easy way of determining how many leaves will contain duplicates of a particular outcome.

Instead of a decision tree, we use a technique called an *adversary argument* for establishing better lower bounds for the selection problem. This technique is described next.

5.1.3 Adversary Arguments

Suppose you are playing a guessing game with a friend. You are to pick a date (a month and day), and the friend will try to guess the date by asking yes/no questions. You want to force your friend to ask as many questions as possible. If the first question is, "Is it in the winter?" and you are a good adversary, you will answer "No," because there are more dates in the three other seasons. To the question, "Is the first letter of the month's name in the first half of the alphabet?" you should answer "Yes." But is this cheating? You didn't really pick a date at all! In fact, you will not pick a specific month and day until the need for consistency in your answers pins you down. This may not be a friendly way to play a guessing game, but it is just right for finding lower bounds for the behavior of an algorithm.

Suppose we have an algorithm that we think is efficient. Imagine an adversary who wants to prove otherwise. At each point in the algorithm where a decision (a key comparison, for example) is made, the adversary tells us the result of the decision. The adversary chooses its answers to try to force the algorithm to work hard, that is, to make a lot of decisions. You may think of the adversary as gradually constructing a "bad" input for the algorithm while it answers the questions. The only constraint on the adversary's answers is that they must be internally consistent; there must be *some* input for the problem for which its answers would be correct. If the adversary can force the algorithm to perform $f(n)$ steps, then $f(n)$ is a lower bound for the number of steps done in the worst case. This approach is explored in Exercise 5.2 for sorting and merging by key comparisons.

In fact, "designing against an adversary" is often a good technique for solving a comparison-based problem efficiently. In thinking about what comparison to make in a given situation, imagine that the adversary will give the least favorable answer—then choose a comparison where both outcomes are about equally favorable. This technique is discussed in more detail in Section 5.6. However, here we are primarily interested in the role of adversary arguments in lower-bound arguments.

We want to find a lower bound on the complexity of a *problem*, not just a particular algorithm. When we use adversary arguments, we will assume that the algorithm is any algorithm whatsoever from the class being studied, just as we did with the decision tree arguments. To get a good lower bound we need to construct a clever adversary that can thwart any algorithm.

5.1.4 Tournaments

In the rest of this chapter we present algorithms for selection problems and adversary arguments for lower bounds for several cases, including the median. In most of the algorithms and arguments, we use the terminology of contests, or tournaments, to describe the results of comparisons. The comparand that is found to be larger will be called the *winner*; the other will be called the *loser*.

5.2 Finding max and min

Throughout this section we use the names max and min to refer to the largest and smallest keys, respectively, in a set of n keys.

We can find max and min by using Algorithm 1.3 to find max, eliminating max from the set, and then using the appropriate variant of Algorithm 1.3 to find min among the remaining $n-1$ keys. Thus max and min can be found by doing $(n-1)+(n-2)$, or $2n-3$, comparisons. This is not optimal. Although we know (from Chapter 1) that $n-1$ key comparisons are needed to find max or min independently, when finding both, some of the work can be "shared." Exercise 1.25 asked for an algorithm to find max and min with only about $3n/2$ key comparisons. A solution (for even n) is to pair up the keys and do $n/2$ comparisons, then find the largest of the winners, and, separately, find the smallest of the losers. If n is odd, the last key may have to be considered among the winners and the losers. In either case, the total number of comparisons is $\lceil 3n/2 \rceil - 2$. In this section we give an adversary argument to show that this solution is optimal. Specifically, in the remainder of this section we prove:

Theorem 5.1 Any algorithm to find max and min of n keys by comparison of keys must do at least $3n/2 - 2$ key comparisons in the worst case.

Proof To establish the lower bound we may assume that the keys are distinct. To know that a key x is max and a key y is min, an algorithm must know that every key other than x has lost some comparison and every key other than y has won some comparison. If we count each win and loss as one unit of information, then an algorithm must have

Status of keys x and y compared by an algorithm	Adversary response	New status	Units of new information
N, N	$x > y$	W, L	2
W, N or WL, N	$x > y$	W, L or WL, L	1
L, N	$x < y$	L, W	1
W, W	$x > y$	W, WL	1
L, L	$x > y$	WL, L	1
W, L or WL, L or W, WL	$x > y$	No change	0
WL, WL	Consistent with assigned values	No change	0

Table 5.1 The adversary strategy for the min and max problem

(at least) $2n - 2$ units of information to be sure of giving the correct answer. We give a strategy for an adversary to use in responding to the comparisons so that it gives away as few as possible units of new information with each comparison. Imagine the adversary constructing a specific input set as it responds to the algorithm's comparisons.

We denote the status of each key at any time during the course of the algorithm as follows:

Key status	Meaning
W	Has won at least one comparison and never lost
L	Has lost at least one comparison and never won
WL	Has won and lost at least one comparison
N	Has not yet participated in a comparison

Each W or L is one unit of information. A status of N conveys no information. The adversary strategy is described in Table 5.1. The main point is that, except in the case where both keys have not yet been in any comparison, the adversary can give a response that provides at most one unit of new information. We need to verify that if the adversary follows these rules, its replies are consistent with some input. Then we need to show that this strategy forces any algorithm to do as many comparisons as the theorem claims.

Observe that in all cases in Table 5.1 except the last, either the key chosen by the adversary as the winner has not yet lost any comparison, or the key chosen as the loser has not yet won any. Consider the first possibility; suppose that the algorithm compares x and y, the adversary chooses x as the winner, and x has not yet lost any comparison. Even if the value already assigned by the adversary to x is smaller than the value it has assigned to y, the adversary can change x's value to make it beat y without contradicting any of the responses it gave earlier. The other situation, where the key to be the loser has never won, can be handled similarly—by reducing the value of the key if necessary. So the

Comparison	x_1		x_2		x_3		x_4		x_5		x_6	
	Status	Value	Status	Value	Status	Value	Status	Value	Status	Value	Status	Value
x_1, x_2	W	20	L	10	N	*	N	*	N	*	N	*
x_1, x_5	W	20							L	5		
x_3, x_4					W	15	L	8				
x_3, x_6					W	15					L	12
x_3, x_1	WL	20			W	25						
x_2, x_4			WL	10			L	8				
x_5, x_6									WL	5	L	3
x_6, x_4							L	2			WL	3

Table 5.2 An example of the adversary strategy for max and min

adversary can construct an input consistent with the rules in the table for responding to the algorithm's comparisons. This is illustrated in the following example.

Example 5.1 Constructing an input using the adversary's rules

The first column in Table 5.2 shows a sequence of comparisons that might be carried out by some algorithm. The remaining columns show the status and value assigned to the keys by the adversary. (Keys that have not yet been assigned a value are denoted by asterisks.) Each row after the first contains only the entries relevant to the current comparison. Note that when x_3 and x_1 are compared (in the fifth comparison), the adversary increases the value of x_3 because x_3 is supposed to win. Later, the adversary changes the values of x_4 and x_6 consistent with its rules. After the first five comparisons, every key except x_3 has lost at least once, so x_3 is max. After the last comparison, x_4 is the only key that has never won, so it is min. In this example the algorithm did eight comparisons; the worst-case lower bound for six keys (still to be proved) is $3/2 \times 6 - 2 = 7$. ■

To complete the proof of Theorem 5.1, we just have to show that the adversary rules will force any algorithm to do at least $3n/2 - 2$ comparisons to get the $2n - 2$ units of information it needs. The only case where an algorithm can get two units of information from one comparison is the case where the two keys have not been included in any previous comparisons. Suppose for the moment that n is even. An algorithm can do at most $n/2$ comparisons of previously unseen keys, so it can get at most n units of information this way. From each other comparison, it gets at most one unit of information. The algorithm needs $n - 2$ additional units of information, so it must do at least $n - 2$ more comparisons. Thus to get $2n - 2$ units of information, it must do at least $n/2 + n - 2 = 3n/2 - 2$ comparisons in total. The reader can easily check that for odd n, at least $3n/2 - 3/2$ comparisons are needed. This completes the proof of Theorem 5.1. □

5.3 Finding the Second-Largest Key

We can find the second-largest element of a set by finding and eliminating the largest, then finding the largest remaining element. Is there a more efficient method? Can we prove a certain method is optimal? This section answers these questions.

5.3.1 Introduction

Throughout this section we use max and secondLargest to refer to the largest and second-largest keys, respectively. For simplicity in describing the problem and algorithms, we will assume that the keys are distinct.

The second-largest key can be found with $2n - 3$ comparisons by using FindMax (Algorithm 1.3) twice, but this is not likely to be optimal. We should expect that some of the information discovered by the algorithm while finding max can be used to decrease the number of comparisons performed in finding secondLargest. Specifically, any key that loses to a key other than max cannot possibly be secondLargest. All such keys discovered while finding max can be ignored during the second pass through the set. (The problem of keeping track of them will be considered later.)

Using Algorithm 1.3 on a set with five keys, the results might be as follows:

Comparands	Winner
x_1, x_2	x_1
x_1, x_3	x_1
x_1, x_4	x_4
x_4, x_5	x_4

Then max = x_4 and secondLargest is either x_5 or x_1 because both x_2 and x_3 lost to x_1. Thus only one more comparison is needed to find secondLargest in this example.

It may happen, however, that during the first pass through the set to find max we don't obtain any information useful for finding secondLargest. If max were x_1, then each other key would be compared only to max. Does this mean that in the worst case $2n - 3$ comparisons must be done to find secondLargest? Not necessarily. In the preceding discussion we used a specific algorithm, Algorithm 1.3. No algorithm can find max by doing fewer than $n - 1$ comparisons, but another algorithm may provide more information that can be used to eliminate some keys from the second pass through the set. The tournament method, described next, provides such information.

5.3.2 The Tournament Method

The tournament method is so named because it performs comparisons in the same way that tournaments are played. Keys are paired off and compared in "rounds." In each round after the first one, the winners from the preceding round are paired off and compared. (If at any round the number of keys is odd, one of them simply waits for the next round.) A tournament can be described by a binary-tree diagram as shown in Figure 5.1. Each leaf contains a key, and at each subsequent level the parent of each pair contains the winner.

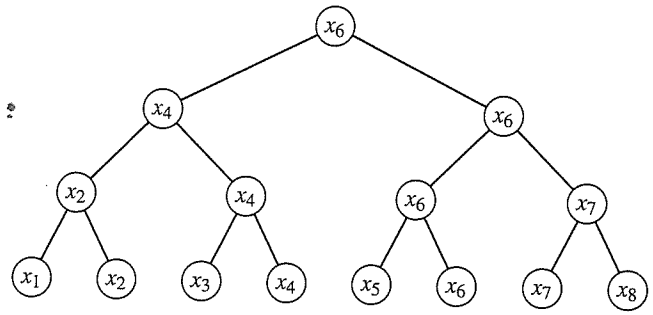


Figure 5.1 An example of a tournament; max = x_6; secondLargest may be x_4, x_5, or x_7.

The root contains the largest key. As in Algorithm 1.3, $n - 1$ comparisons are done to find max.

In the process of finding max, every key except max loses in one comparison. How many lose directly to max? If n is a power of 2, there are exactly $\lg n$ rounds; in general, the number of rounds is $\lceil \lg n \rceil$. Since max is involved in at most one comparison in each round, there are at most $\lceil \lg n \rceil$ keys that lost only to max. These are the only keys that could possibly be secondLargest. The method of Algorithm 1.3 can be used to find the largest of these $\lceil \lg n \rceil$ keys by doing at most $\lceil \lg n \rceil - 1$ comparisons. Thus the tournament finds max and secondLargest by doing a total of $n + \lceil \lg n \rceil - 2$ comparisons in the worst case. This is an improvement over our first result of $2n - 3$. Can we do better?

5.3.3 An Adversary Lower-Bound Argument

Both methods we considered for finding the second-largest key first found the largest key. This is not wasted effort. Any algorithm that finds secondLargest must also find max because, to know that a key is the second largest, one must know that it is not the largest; that is, it must have lost in one comparison. The winner of the comparison in which secondLargest loses must, of course, be max. This argument gives a lower bound on the number of comparisons needed to find secondLargest, namely $n - 1$, because we already know that $n - 1$ comparisons are needed to find max. But one would expect that this lower bound could be improved because an algorithm to find secondLargest should have to do more work than an algorithm to find max. We will prove the following theorem, which has as a corollary that the tournament method is optimal.

Theorem 5.2 Any algorithm (that works by comparing keys) to find the second largest in a set of n keys must do at least $n + \lceil \lg n \rceil - 2$ comparisons in the worst case.

Proof For the worst case, we may assume that the keys are distinct. We have already observed that there must be $n - 1$ comparisons with distinct losers. If max was a comparand in $\lceil \lg n \rceil$ of these comparisons, then all but one of the $\lceil \lg n \rceil$ keys that lost to max must lose again for secondLargest to be correctly determined. Then a total of at least $n + \lceil \lg n \rceil - 2$

comparisons would be done. Therefore we will show that there is an adversary strategy that can force any algorithm that finds secondLargest to compare max to $\lceil \lg n \rceil$ distinct keys.

The adversary assigns a "weight" $w(x)$ to each key x in the set. Initially $w(x) = 1$ for all x. When the algorithm compares two keys x and y, the adversary determines its reply and modifies the weights as follows.

Case	Adversary reply	Updating of weights
$w(x) > w(y)$	$x > y$	New $w(x)$ = prior $(w(x) + w(y))$; new $w(y) = 0$.
$w(x) = w(y) > 0$	Same as above.	Same as above.
$w(y) > w(x)$	$y > x$	New $w(y)$ = prior $(w(x) + w(y))$; new $w(x) = 0$.
$w(x) = w(y) = 0$	Consistent with previous replies.	No change.

To interpret the weights and adversary rules, imagine that the adversary builds trees to represent the ordering relations between the keys. If x is the parent of y, then x beat y in a comparison. Figure 5.2 shows an example. The adversary combines two trees only when their roots are compared. If the algorithm compares nonroots, no change is made in the trees. The weight of a key is simply the number of nodes in that key's tree, if it is a root, and zero otherwise.

We need to verify that if the adversary follows this strategy, its replies are consistent with some input, and max will be compared to at least $\lceil \lg n \rceil$ distinct keys. These conclusions follow from a sequence of easy observations:

1. A key has lost a comparison if and only if its weight is zero.
2. In the first three cases, the key chosen as the winner has nonzero weight, so it has not yet lost. The adversary can give it an arbitrarily high value to make sure it wins without contradicting any of its earlier replies.
3. The sum of the weights is always n. This is true initially, and the sum is preserved by the updating of the weights.
4. When the algorithm stops, only one key can have nonzero weight. Otherwise there would be at least two keys that never lost a comparison, and the adversary could choose values to make the algorithm's choice of secondLargest incorrect.

Let x be the key that has nonzero weight when the algorithm stops. By facts 1 and 4, $x = \max$. Using fact 3, $w(x) = n$ when the algorithm stops.

To complete the proof of the theorem, we need to show that x has directly won against at least $\lceil \lg n \rceil$ distinct keys. Let $w_k = w(x)$ just after the kth comparison won by x against a previously undefeated key. Then by the adversary's rules,

$$w_k \leq 2w_{k-1}.$$

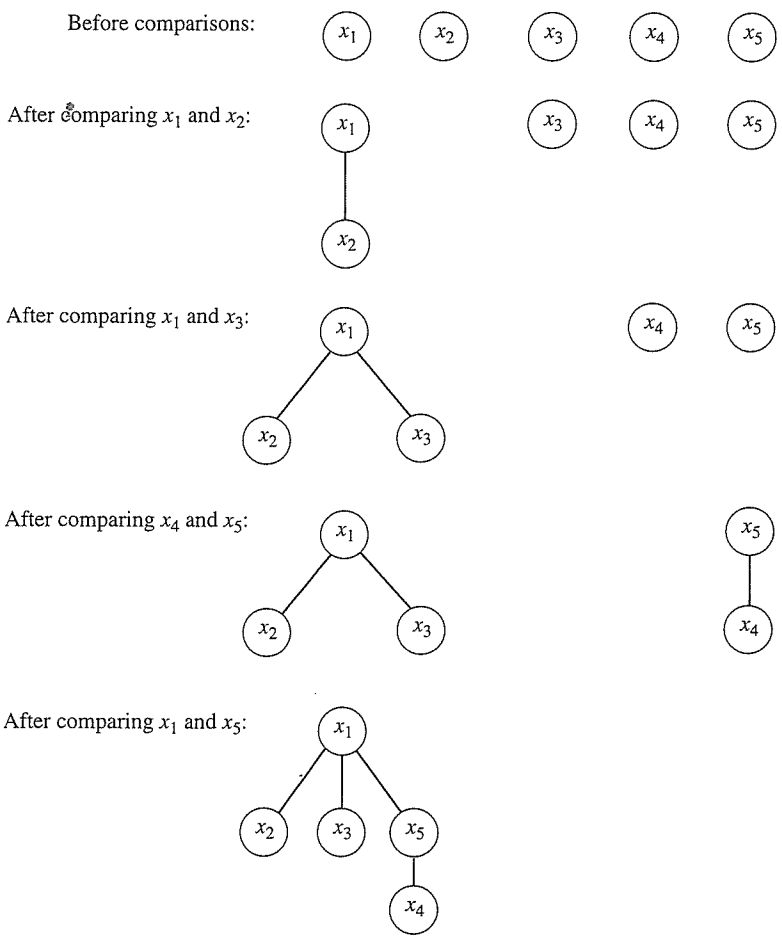


Figure 5.2 Trees for the adversary decisions in Example 5.2

Now let K be the number of comparisons x wins against previously undefeated keys. Then

$$n = w_K \leq 2^K w_0 = 2^K.$$

Thus $K \geq \lg n$, and since K is an integer, $K \geq \lceil \lg n \rceil$. The K keys counted here are of course distinct, since once beaten by x, a key is no longer "previously undefeated" and won't be counted again (even if an algorithm foolishly compares it to x again). □

Example 5.2 The adversary strategy in action

To illustrate the adversary's action and show how its decisions correspond to the step-by-step construction of an input, we show an example for $n = 5$. Keys in the set that have not yet been specified are denoted by asterisks. Thus initially the keys are *, *, *, *, *. Note that

Comparands	Weights	Winner	New weights	Keys
x_1, x_2	$w(x_1) = w(x_2)$	x_1	2, 0, 1, 1, 1	20, 10, *, *, *
x_1, x_3	$w(x_1) > w(x_3)$	x_1	3, 0, 0, 1, 1	20, 10, 15, *, *
x_5, x_4	$w(x_5) = w(x_4)$	x_5	3, 0, 0, 0, 2	20, 10, 15, 30, 40
x_1, x_5	$w(x_1) > w(x_5)$	x_1	5, 0, 0, 0, 0	41, 10, 15, 30, 40

Table 5.3 Example of the adversary strategy for the Second-Largest Key problem

values assigned to some keys may be changed at a later time. See Table 5.3, which shows just the first few comparisons (those that find max, but not enough to find secondLargest). The weights and the values assigned to the keys will not be changed by any subsequent comparisons. ■

5.3.4 Implementation of the Tournament Method for Finding max and secondLargest

To conduct the tournament to find max we need a way to keep track of the winners in each round. After max has been found by the tournament, only those keys that lost to it are to be compared to find secondLargest. How can we keep track of the elements that lose to max when we don't know in advance which key is max? Since the tournament is conceptually a binary tree that is as balanced as possible, the *heap structure* of Section 4.8.1 suggests itself. For a set of n elements, a heap structure of $2n - 1$ nodes is used; that is, an array E[1], . . ., E[2*n-1]. Initially, place the elements in positions $n, \ldots, 2n - 1$. As the tournament progresses, positions $1, \ldots, n - 1$ will be filled (in reverse order) with winners. Exercise 5.4 covers the additional details. This algorithm uses linear extra space and runs in linear time.

5.4 The Selection Problem

Suppose we want to find the median of n elements in an array E in positions $1, \ldots, n$. (That is, we want the element of rank $\lceil n/2 \rceil$.) In earlier sections we found efficient methods for finding ranks near one extreme or the other, such as the maximum, the minimum, both maximum *and* minimum, and the second-largest key. The exercises explore more variations, but all of the techniques for these problems lose efficiency as we move away from the extremes, and are not useful for finding the median. If we are to find a solution that is more efficient than simply sorting the whole set, then a new idea is needed.

5.4.1 A Divide-and-Conquer Approach

Suppose we can partition the keys into two sets, S_1 and S_2, such that all keys in S_1 are smaller than all keys in S_2. Then the median is in the larger of the two sets (that is, the set with more keys, not necessarily the set with larger keys). We can ignore the other set and restrict our search to the larger set.

But what key do we look for in the larger set? Its median is not the median of the original set of keys.

Example 5.3 Partitioning in search of the median

Suppose $n = 255$. We are seeking the median element (whose rank is $k = 128$). Suppose, after partitioning, that S_1 has 96 elements and S_2 has 159 elements. Then the median of the whole set is in S_2, and it is the 32nd-smallest element in S_2. Thus the problem reduces to finding the element of rank 32 in S_2, which has 159 elements. ■

The example shows that this approach to solving the median problem naturally suggests that we solve the general selection problem.

Thus we are developing a divide-and-conquer solution for the general selection problem that, like Binary Search and FixHeap, divides the problem to be solved into *two* smaller problems, but needs to solve only *one* of the smaller problems. Quicksort uses Partition to divide the elements into subranges of elements "small" and "large" relative to a pivot element (see Algorithm 4.2). We can use a modified version of Quicksort for the selection problem, called findKth, in which only one recursive subproblem needs to be solved. The details are worked out in Exercise 5.8.

In the analysis parts of Exercise 5.8 we learn that the same pattern emerges that we found when we analyzed Quicksort. Although findKth works well on average, the worst case is plagued by the same problem that confronts Quicksort: The pivot element may give a very uneven division of the elements into S_1 and S_2. To develop a better solution, consider what we learned from Quicksort.

Seeing that the crux of the problem is to choose a "good" pivot element, we can review the suggestions of Section 4.4.4, but none of them guarantees that the pivot will divide the set of elements into subsets of equal, or almost equal, size. In the next section we will see that, by investing more effort, it is possible to choose a pivot that is guaranteed to be "good." It guarantees that each set will have at least $0.3\,n$ and at most $0.7\,n$ elements. With this "high-quality" pivot element, the divide-and-conquer method works efficiently in the worst case, as well as the average case.

⋆ 5.4.2 A Linear-Time Selection Algorithm

The algorithm we present in this section is a simplification of the first linear algorithm discovered for solving the selection problem. The simplification makes the general strategy easier to understand (though the details are complicated and tricky to implement), but it is less efficient than the original. The algorithm is important and interesting because it solves the selection problem in general, not just for the median, because it *is* linear, and because it opened the way for improvements.

As usual, to simplify the description of the algorithm, we assume the keys are distinct. It is not hard to modify if there are duplicate keys.

Algorithm 5.1 Selection

Input: S, a set of n keys; and k, an integer such that $1 \le k \le n$.

Output: The kth smallest key in S.

Remark: Recall that $|S|$ denotes the number of elements in S.

Element select(SetOfElements S, **int** k)

0. **if** ($|S| \leq 5$)
   return direct solution for kth element of S.

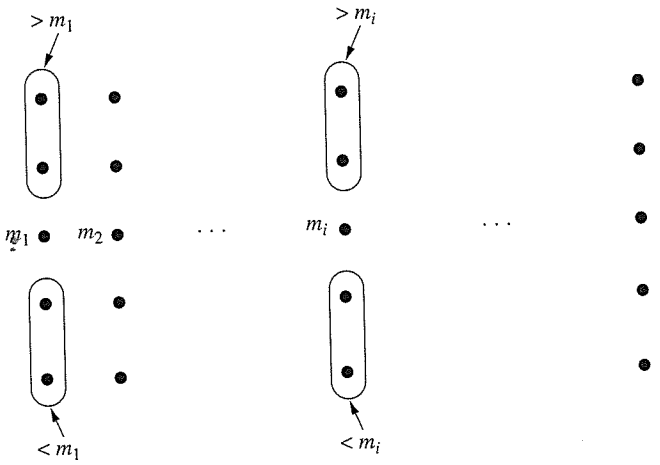
1. Divide the keys into sets of five each, and find the median of each set. (The last set may have fewer than five keys; however, later references to "set of five keys" include this set, too.) Call the set of medians M. Let $n_M = |M| = \lceil n/5 \rceil$.
   At this point we can imagine the keys arranged as shown in Figure 5.3(a).
   In each set of five keys, the two larger than the median appear above the median, and the smaller two appear below the median.

2. m^* = select(M, $\lceil |M|/2 \rceil$);
   (m^* is now the median of M, i.e., the median of medians.)
   Now imagine the keys as in Figure 5.3(b), where the sets of five keys have been arranged so that the sets whose medians are larger than m^* appear to the right of the set containing m^*, and the sets with smaller medians appear to the left of the set containing m^*. Observe that, by transitivity, all keys in the section labeled B are larger than m^*, and all keys in the section labeled C are smaller than m^*.

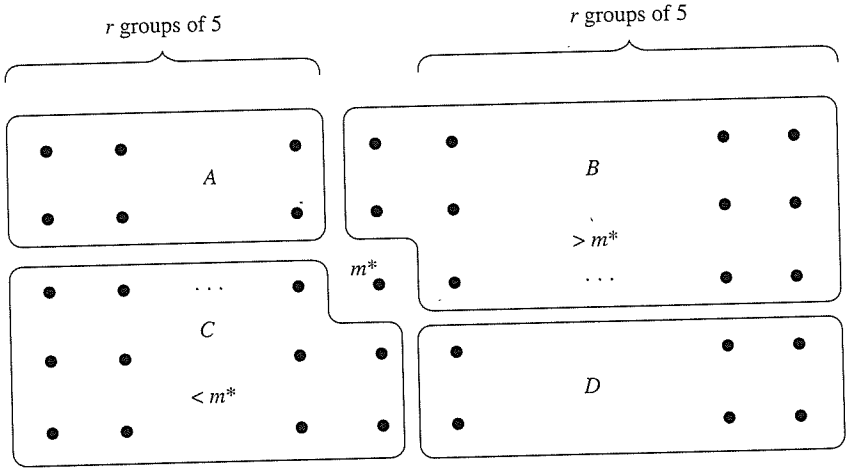
3. Compare each key in the sections labeled A and D in Figure 5.3(b) to m^*.
   Let $S_1 = C \cup \{\text{keys from } A \cup D \text{ that are smaller than } m^*\}$.
   Let $S_2 = B \cup \{\text{keys from } A \cup D \text{ that are larger than } m^*\}$.
   This completes the partitioning process with m^* as pivot.

4. Divide and conquer:
   if ($k = |S_1| + 1$)
   m^* is the kth-smallest key, so:
   return m^*;
   else if ($k \leq |S_1|$)
   the kth-smallest key is in S_1, so:
   return select(S_1, k);
   else
   the kth-smallest key is in S_2, so:
   return select(S_2, $k - |S_1| - 1$);

Algorithm 5.1 is expressed in terms of a set S and rank k. Here we briefly discuss implementation for an array E, using indexes 1 through n, rather than 0 through $n - 1$. Finding an element with rank k is equivalent to answering the question: If this array were sorted, which element would be in E[k]? If S_1 has n_1 elements, then we rearrange E so that



(a) m_i is the median of the ith group of five keys. There are $n_M = \lceil n/5 \rceil$ groups. This diagram assumes n is a multiple of 5 for simplicity.



(b) Medians less than m^* are to its left; medians greater than m^* are to its right. This diagram assumes n is an odd multiple of 5 for simplicity. Therefore $n = 5(2r + 1)$ for some r.

Figure 5.3 Steps 1 and 2 for the linear-time selection algorithm

all elements of S_1 are in positions $1, \ldots, n_1$, m^* is in position $n_1 + 1$, and all elements of S_2 are in positions $n_1 + 2, \ldots, n$.

First we observe that if $k = n_1 + 1$, then m^* is the desired element. If $k \leq n_1$, then for the next call to select, the question is: If the segment E[1], . . ., E[n_1] were sorted, which element would be in E[k]? If $k \geq n_1 + 2$, then for the next call to select, the question is:

If the segment E[$n_1 + 2$], . . ., E[n] were sorted, which element would be in E[k]? (This is equivalent to the problem of finding an element of rank $k - n_1 - 1$ in set S_2 by itself.) The point here is that the variable k will be the same for all recursive calls. However, some changes are needed in the details of the tests to determine which subrange to search recursively. These are left as an exercise (Exercise 5.9).

⋆ 5.4.3 Analysis of the Selection Algorithm

We next show that select is a linear algorithm. We will not completely prove this claim, but we will give the structure of the argument assuming that n is an odd multiple of 5 to simplify the counting.

Let $W(n)$ be the number of key comparisons done by select in the worst case on inputs with n keys. Assuming $n = 5(2r + 1)$ for some nonnegative integer r (and ignoring the problem that this might not be true of the sizes of the inputs for the recursive calls), we count the comparisons done by each step of select. Brief explanations of the computation are included after some of the steps.

1. Find the medians of sets of five keys: $6(n/5)$ comparisons.
   The median of five keys can be found using six comparisons (Exercise 5.14). There are $n/5$ such sets.
2. Recursively find the median of the medians: $W(n/5)$ comparisons.
3. Compare all keys in sections A and D to m^* (see Figure 5.3b): $4r$ comparisons.
4. Call select recursively: $W(7r + 2)$ comparisons.
   In the worst case, all $4r$ keys in sections A and D will be on the same side of m^* (i.e., all smaller than m^* or all greater than m^*). B and C each have $3r + 2$ elements. So the size of the largest possible input for the recursive call to select is $7r + 2$.

Since $n = 5(2r + 1)$, r is approximately $n/10$. So

$$W(n) \leq 1.2n + W(0.2n) + 0.4n + W(0.7n) = 1.6n + W(0.2n) + W(0.7n). \quad (5.1)$$

Although this recurrence equation (actually inequality) is of the divide-and-conquer type, the two subproblems are not of equal size, so we cannot simply apply the Master Theorem (Theorem 3.17). However, we can develop a recursion tree (Section 3.7), as shown in Figure 5.4. Since the row-sums form a decreasing geometric series, whose ratio is 0.9, the total is Θ of the largest term, which is $\Theta(n)$. Equation (1.10) gives the exact expression for the geometric series, which is $16n$ minus some very small term. This result can also be verified by induction. Thus the selection algorithm is a linear algorithm.

The original presentation of the algorithm in the literature included improvements to cut the number of key comparisons down to approximately $5.4\,n$. The best currently known algorithm for finding the median does $2.95\,n$ comparisons in the worst case. (It, too, is complicated.)

Since select is recursive, it uses space on a stack; it is not an in-place algorithm. However, the depth of recursion is in $O(\log n)$, so it is unlikely to cause a problem.

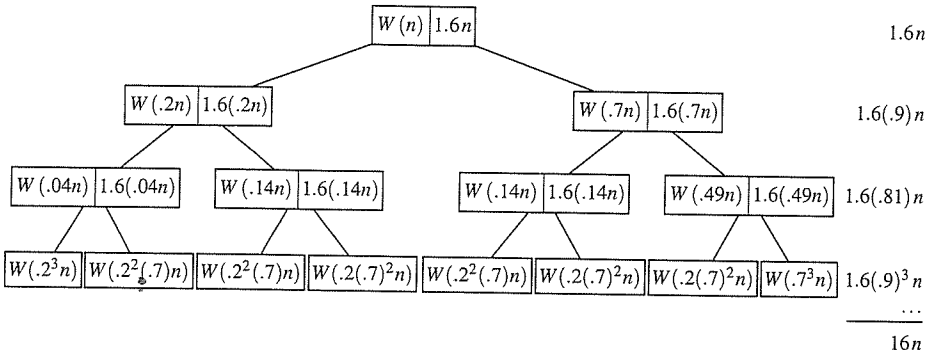


Figure 5.4 Recursion tree for select

5.5 A Lower Bound for Finding the Median

We are assuming that E is a set of n keys and that n is odd. We will establish a lower bound on the number of key comparisons that must be done by any key-comparison algorithm to find median, the $(n + 1)/2$-th key. Since we are establishing a lower bound, we may, without loss of generality, assume that the keys are distinct.

We claim first that to know median, an algorithm must know the relation of every other key to median. That is, for each other key, x, the algorithm must know that $x >$ median or $x <$ median. In other words, it must establish relations as illustrated by the tree in Figure 5.5. Each node represents a key, and each branch represents a comparison. The key at the higher end of the branch is the larger key. Suppose there were some key, say y, whose relation to median was not known. (See Figure 5.6(a) for an example.) An adversary could change the value of y, moving it to the opposite side of median, as in Figure 5.6(b), without contradicting the results of any of the comparisons done. Then median would not be the median; the algorithm's answer would be wrong.

Since there are n nodes in the tree in Figure 5.5, there are $n - 1$ branches, so at least $n - 1$ comparisons must be done. This is neither a surprising nor exciting lower bound. We will show that an adversary can force an algorithm to do other "useless" comparisons before it performs the $n - 1$ comparisons it needs to establish the tree of Figure 5.5.

Definition 5.1 Crucial comparison

A comparison involving a key x is a *crucial comparison for* x if it is the first comparison where $x > y$, for some $y \geq$ median, or $x < y$ for some $y \leq$ median. Comparisons of x and y where $x >$ median and $y <$ median are *noncrucial*. ■

A crucial comparison establishes the relation of x to median. Note that the definition does not require that the relation of y to median be already known at the time the crucial comparison for x is done.

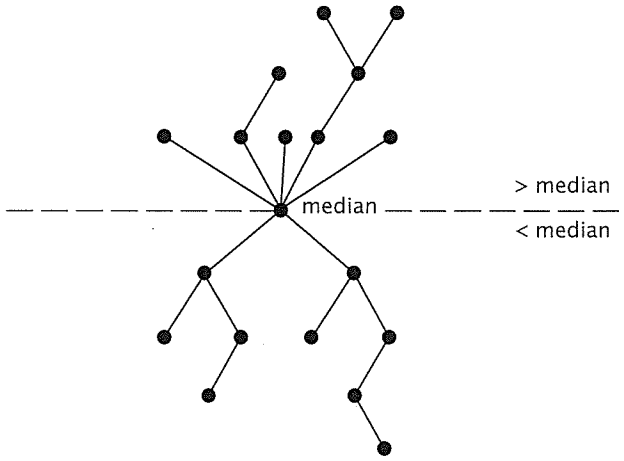
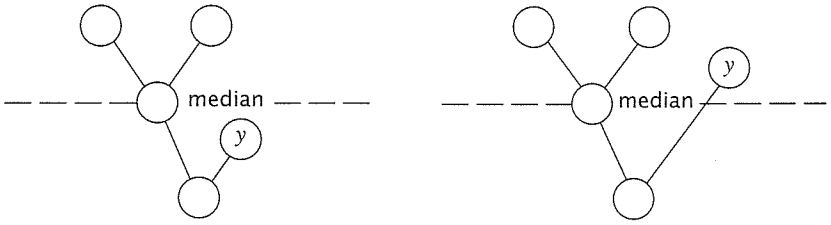


Figure 5.5 Comparisons relating each key to median



(a) $y <$ median. (b) $y >$ median; median is not the median.

Figure 5.6 An adversary conquers a bad algorithm

We will exhibit an adversary that forces an algorithm to perform *noncrucial* comparisons. The adversary chooses some value (but not a particular key) to be median. It will assign a value to a key when the algorithm first uses that key in a comparison. So long as it can do so, the adversary will assign values to new keys involved in a comparison so as to put the keys on opposite sides of median. The adversary may not assign values larger than median to more than $(n-1)/2$ keys, nor values smaller than median to more than $(n-1)/2$ keys. It keeps track of the assignments it has made to be sure not to violate these restrictions. We indicate the status of a key during the running of the algorithm as follows:

- L Has been assigned a value *L*arger than median.
- S Has been assigned a value *S*maller than median.
- N Has not yet been in a comparison.

Comparands	Adversary's action
N, N	Make one key larger than median, the other smaller.
L, N or N, L	Assign a value smaller than median to the key with status N.
S, N or N, S	Assign a value larger than median to the key with status N.

Table 5.4 The adversary strategy for the median-finding problem

The adversary's strategy is summed up in Table 5.4. In all cases, if there are already $(n-1)/2$ keys with status S (or L), the adversary ignores the rule in the table and assigns value(s) larger (or smaller) than median to the new key(s). When only one key without a value remains, the adversary assigns the value median to that key. Whenever the algorithm compares two keys with statuses L and L, S and S, or L and S, the adversary simply gives the correct response based on the values it has already assigned to the keys.

All of the comparisons described in Table 5.4 are noncrucial. How many can the adversary make any algorithm do? Each of these comparisons creates at most one L-key, and each creates at most one S-key. Since the adversary is free to make the indicated assignments until there are $(n-1)/2$ L-keys or $(n-1)/2$ S-keys, it can force any algorithm to do at least $(n-1)/2$ noncrucial comparisons. (Since an algorithm could start out by doing $(n-1)/2$ comparisons involving two N-keys, this adversary can't guarantee any more than $(n-1)/2$ noncrucial comparisons.)

We can now conclude that the total number of comparisons must be at least $n-1$ (the crucial comparisons) + $(n-1)/2$ (noncrucial comparisons). We sum up the result in the following theorem.

Theorem 5.3 Any algorithm to find the median of n keys (for odd n) by comparison of keys must do at least $3n/2 - 3/2$ comparisons in the worst case. □

Our adversary was not as clever as it could have been in its attempt to force an algorithm to do noncrucial comparisons. In the past several years the lower bound for the median problem has crept up to roughly $1.75n - \log n$, then roughly $1.8n$, then a little higher. The best lower bound currently known is slightly above $2n$ (for large n). There is still a small gap between the best known lower bound and the best known algorithm for finding the median.

5.6 Designing Against an Adversary

Designing against an adversary can be a powerful technique for developing an algorithm with operations like comparisons, which elicit information about the input elements. The main idea is to anticipate that any "question" (i.e., comparison or other test performed by the algorithm) is going to receive an answer chosen by an adversary to be as unfavorable as possible for the algorithm, usually by giving the least information. To counter this,

the algorithm should choose comparisons (or whatever the operation is) for which both answers give the same amount of information, as far as possible.

The idea that a good algorithm uses some notion of balance has come up before, when we studied decision trees. The number of comparisons done in the worst case is the height of a decision tree for the algorithm. To keep the height small, for a fixed problem size, means keeping the tree as balanced as possible. A good algorithm chooses comparisons such that the number of possible outcomes (outputs) for one result of the comparison is roughly equal to the number of outcomes for the other result.

We have seen several examples of this technique already: Mergesort, finding both max and min, and finding the second-largest element. The first phase of the tournament method for finding the second-largest element, that is, the tournament that finds the maximum element, is the clearest example. In the first round each key comparison is between two elements about which nothing is known, so an adversary has no basis for favoring one answer over another. In subsequent rounds, to the extent possible, elements that have equal win–loss records are compared, so the adversary never can give one answer that is less informative than the other. In contrast, the straightforward algorithm to find the maximum first compares x_1 with x_2, then compares the winner (say x_2) with x_3. In this case, the adversary *can* give one answer that is less informative than the other. (Which?)

In general, for comparison-based problems, the complete status of an element includes more than the number of prior wins and losses. Rather, an element's status includes the number of elements known to be smaller and the number of elements known to be larger by transitivity. Tree structures like those in Figure 5.2 can be used to represent the status information graphically.

To further illustrate the technique of designing against an adversary, we consider two problems whose optimum solution is difficult: finding the median of five elements and sorting five elements (Exercises 5.14 and 5.15). The median can be found with six comparisons, and five elements can be sorted with seven comparisons. Many students (and instructors) have spent hours trying various strategies, looking unsuccessfully for the solutions. The optimal algorithms squeeze the most information possible out of each comparison. The technique we are describing in this section gives a big boost in getting started right. The first comparison is arbitrary; it is necessarily between two keys about which we have no information. Should the second comparison include either of these keys? No; comparing two new keys, which have equal status, gives more information. Now we have two keys that (we know) are each larger than one other, two keys that (we know) are each smaller than one other, and one unexamined key. Which two will you compare next?

Are you beginning to wonder which problem we are working on? The technique of designing against an adversary suggests the same first three comparisons for both the median problem and the sorting problem. Finishing the algorithms is still tricky and makes instructive exercises.

Exercises

Section 5.1 Introduction

5.1 Draw the decision tree for FindMax (Algorithm 1.3) with $n = 4$.

5.2 Consider the problem of sorting n elements. Essentially, there are only $n!$ distinct outcomes, one for each permutation. Adversaries are not limited in how much computation they are allowed to do in deciding the outcome, or answer, for a comparison that is "asked" by the algorithm. In principle, an adversary for the sorting problem could look at all permutations before arriving at a decision.

a. Use the above idea to develop an adversary strategy for comparison-based sorting. Find a lower bound based on your strategy. How does your result compare with the lower bound of Theorem 4.10?

⋆ **b.** Develop an adversary strategy for the problem of merging two sorted sequences, each containing $n/2$ keys. It should be a simple modification of your strategy for part (a). Find a lower bound for the worst case of comparison-based algorithms for this problem, based on your strategy. How does your result compare with the lower bound of Theorem 4.4? *Hint*: Look at Exercise 4.25.

Section 5.2 Finding max *and* min

5.3 We used an adversary argument to establish the lower bound for finding the minimum and maximum of n keys. What lower bound do we get from a decision tree argument?

Section 5.3 Finding the Second-Largest Key

5.4 In this exercise you will write an algorithm based on the heap structure (Section 4.8.1) for the tournament method to find max and secondLargest.

a. Show that the following procedure places the max in E[1]. Array E is allocated for indexes $1, \ldots, 2n - 1$. (Recall that "last −= 2" subtracts 2 from last.)

```
heapFindMax(E, n)
    int last;
    Load n elements into E[n], . . ., E[2*n−1].
    for (last = 2*n − 2; last ≥ 2; last −= 2)
        E[last/2] = max(E[last], E[last+1]);
```

b. Explain how to determine which elements lost to the winner.

c. Complete the code to find secondLargest after heapFindMax finishes.

5.5 How many comparisons are done by the tournament method to find secondLargest on the average,

a. if n is a power of 2?

b. if n is not a power of 2?

Hint: Consider Exercise 5.4.

5.6 The following algorithm finds the largest and second-largest keys in an array E of n keys by sequentially scanning the array and keeping track of the two largest keys seen so far. (It assumes $n \geq 2$.)

```
if (E[1] > E[2])
    max = E[1];
    second = E[2];
else
    max = E[2];
    second = E[1];
for (i = 3; i ≤ n; i ++)
    if (E[i] > second)
        if (E[i] > max)
            second = max;
            max = E[i];
        else
            second = E[i];
```

a. How many key comparisons does this algorithm do in the worst case? Give a worst-case input for $n = 6$ using integers for keys.

⋆ **b.** How many key comparisons does this algorithm do on the average for n keys assuming any permutation of the keys (from their proper ordering) is equally likely?

⋆ **5.7** Write an efficient algorithm to find the third-largest key from among n keys. How many key comparisons does your algorithm do in the worst case? Is it necessary for such an algorithm to determine which key is max and which is secondLargest?

Section 5.4 The Selection Problem

5.8 Quicksort can be modified to find the kth-smallest key among n keys so that in most cases it does much less work than is needed to sort the set completely.

a. Write a modified Quicksort algorithm called findKth for this purpose.

b. Show that when this algorithm is used to find the median, the worst case is in $\Theta(n^2)$.

c. Develop a recurrence equation for the average running time of this algorithm.

⋆ **d.** Analyze your algorithm's average running time. What is the asymptotic order?

5.9 Following the pseudocode outline for the Selection algorithm (Algorithm 5.1), we briefly discussed implementation in an array. Finding an element with rank k in an array E with n elements is equivalent to answering the question: If this array were sorted, which element would be in E[k]? The point was that the parameter k will be the same for all

recursive calls. Rewrite the test conditions in the two **if** statements in step 4 to work properly with this implementation.

5.10 Suppose we use the following algorithm to find the k largest keys in a set of n keys. (See Section 4.8 for heap algorithms.)

```
Build a heap H out of the n keys;
for (i = 1; i ≤ k; i ++)
    output(getMax(H));
    deleteMax(H);
```

How large can k be (as a function of n) for this algorithm to be linear in n?

⋆ **5.11** Generalize the tournament method to find the k largest of n keys (where $1 \leq k \leq n$). Work out any implementation details that affect the order of the running time. How fast is your algorithm as a function of n and k?

Section 5.5 A Lower Bound for Finding the Median

5.12 Suppose n is even and we define the median to be the $n/2$th-smallest key. Make the necessary modifications in the lower bound argument and in Theorem 5.3 (where we assumed n was odd).

Section 5.6 Designing Against an Adversary

5.13 How well do the sorting algorithms below meet the criterion of performing comparisons for which either outcome is about equally informative? How would an adversary respond to the comparisons using a "least new information" strategy? Does this push the algorithms into their worst cases?

a. Insertion Sort?
b. Quicksort?
c. Mergesort?
d. Heapsort?
e. Accelerated Heapsort?

⋆ **5.14** Give an algorithm to find the median of five keys with only six comparisons in the worst case. Describe the steps, but don't write code. Using tree diagrams like those in Figure 5.2 may be helpful in explaining what your algorithm does. *Hint*: A useful strategy and the first few steps were partly sketched in Section 5.6.

⋆ **5.15** Give an algorithm to sort five keys with only seven comparisons in the worst case. Describe the steps, but don't write code. Using tree diagrams like those in Figure 5.2 may be helpful in explaining what your algorithm does. *Hint*: A useful strategy and the first few steps were partly sketched in Section 5.6.

Additional Problems

5.16 Prove Theorem 1.16 (the lower bound for searching an ordered array) by means of an adversary argument. *Hint*: Define an *active range* consisting of the minimum and maximum indexes of the array that might contain K, the key being searched for.

5.17 Let E be an array with elements defined for indexes 0, . . ., n (thus there are $n + 1$ elements). Suppose it is known that E is *unimodal*, which means that E[i] is strictly increasing up to some index M, and is strictly decreasing for indexes $i > M$. Thus E[M] is the maximum. (Note that M may be 0 or n.) The problem is to find M.

a. As a warm-up, show that for $n = 2$, two comparisons are necessary and sufficient.

b. Write an algorithm to find M, by comparing various keys in E.

c. How many comparisons does your algorithm do in the worst case? (You should be able to devise an algorithm that is in $o(n)$.)

⋆ **d.** Suppose that $n = F_k$, the kth Fibonacci number, as defined in Equation (1.13), where $k \geq 2$. Describe an algorithm to find M with $k - 1$ comparisons. Describe the ideas, but don't write code.

⋆ **e.** Devise an adversary strategy that forces any comparison-based algorithm to do at least $\lg n + 2$ comparisons to find M, for $n \geq 4$. This shows that the problem is at least a little harder than searching an ordered array. *Hint*: Try a more elaborate version of the adversary strategy suggested for Exercise 5.16.

⋆ **5.18** Suppose E1 and E2 are arrays, each with n keys sorted in ascending order.

a. Devise an $O(\log n)$ algorithm to find the nth smallest of the $2n$ keys. (This is the median of the combined set.) For simplicity, you may assume the keys are distinct.

b. Give a lower bound for this problem.

5.19

a. Give an algorithm to determine if the n keys in an array are all distinct. Assume three-way comparisons; that is, the result of a comparison of two keys is $<$, $=$, or $>$. How many key comparisons does your algorithm do?

⋆ **b.** Give a lower bound on the number of (three-way) key comparisons needed. (Try for $\Omega(n \log n)$.)

5.20 Consider the problem of determining if a bit string of length n contains two consecutive zeroes. The basic operation is to examine a position in the string to see if it is a 0 or a 1. For each n = 2, 3, 4, 5 either give an adversary strategy to force any algorithm to examine every bit, or give an algorithm that solves the problem by examining fewer than n bits.

5.21 Suppose you have a computer with a small memory and you are given a sequence of keys in an external file (on a disk or tape). Keys can be read into memory for processing, but no key can be read more than once.

a. What is the minimum number of storage cells needed for keys in memory to find the largest key in the file? Justify your answer.

b. What is the minimum number of cells needed for keys in memory to find the median? Justify your answer.

5.22

a. You are given n keys and an integer k such that $1 \leq k \leq n$. Give an efficient algorithm to find *any one* of the k smallest keys. (For example, if $k = 3$, the algorithm may provide the first-, second-, or third-smallest key. It need not know the exact rank of the key it outputs.) How many key comparisons does your algorithm do? *Hint*: Don't look for something complicated. One insight gives a short, simple algorithm.

b. Give a lower bound, as a function of n and k, on the number of comparisons needed to solve this problem.

⋆ **5.23** Let E be an n-element array of positive integers. A *majority element* in E is an element that occurs *more than* $n/2$ times in the array. The *majority element problem* is to find the majority element in an array if it has one, or return -1 if it does not have one. The only operations you may perform on the elements are to compare them to each other and move or copy them.

Write an algorithm for the majority element problem. Analyze the time and space used by your algorithm in the worst case. (There are easy $\Theta(n^2)$ algorithms, but there is a linear solution. *Hint*: Use a variation of the technique in Section 5.3.2.)

⋆ **5.24** M is an $n \times n$ integer matrix in which the keys in each row are in increasing order (reading left to right) and the keys in each column are in increasing order (reading top to bottom). Consider the problem of finding the position of an integer x in M, or determining that x is not there. Give an adversary argument to establish a lower bound on the number of comparisons of x with matrix entries needed to solve this problem. The algorithm is allowed to use three-way comparisons; that is, a comparison of x with M[i][j] tells if $x <$ M[i][j], $x =$ M[i][j], or $x >$ M[i][j].

Note: Finding an efficient algorithm for the problem was Exercise 4.58 in Chapter 4. If you did a good job on both your algorithm and your adversary argument, the number of comparisons done by the algorithm should be the same as your lower bound.

Notes and References

Knuth (1998) is an excellent reference for the material in this chapter. It contains some history of the selection problem, including the attempt by Charles Dodgson (Lewis Carroll), in 1883, to work out a correct algorithm so that second prize in lawn tennis tournaments could be awarded fairly. The tournament algorithm for finding the second-largest key appeared

in a 1932 paper by J. Schreier (in Polish). It was proved optimal in 1964 by S. S. Kislitsin (in Russian). The lower bound argument given here is based on Knuth (1998).

The algorithm and lower bound for finding min and max and Exercise 5.21 are attributed to I. Pohl by Knuth.

The first linear selection algorithm is in Blum, Floyd, Pratt, Rivest, and Tarjan (1973). Other selection algorithms and lower bounds appear in Hyafil (1976), Schönhage, Paterson, and Pippenger (1976), and Dor and Zwick (1995, 1996a, 1996b).