3.1 The Selection Problem

Suppose $L$ is an array containing $n$ 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 the $k$th smallest key in $L$. 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 (as well as copying or moving keys).

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 $k = \lfloor n/2 \rfloor$, that is, where we want to find the middle, or median, element.

Of course the selection problem can be solved in general by sorting $L$; then $L[k]$ 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 $\Omega(n \log n)$? We will answer these questions in this chapter.

3.1.2 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 that the leaves represent the outputs. (For the search problem in Section 1.5, the internal nodes also represented outputs.) The number of comparisons done in the worst case is the depth of the tree; the depth is at least $\lceil \lg n \rceil$, where $l$ is the number of leaves.

In Chapter 1 we used decision trees to get the (worst-case) lower bound of $\lceil \lg n \rceil + 1$ 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 2 we used decision trees to get a lower bound of $\lceil \lg n \rceil$, or roughly $\lceil \log_2 n - 1.5 \rceil$, 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 list may be the output, i.e., the $k$th smallest. Thus we can conclude that the depth 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 is 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, draw the decision tree for $\text{FindMax}$ (Algorithm 1.3) with $n = 4$. The decision tree argument fails to give a good lower bound because we do not have an easy way to determine how many leaves will contain duplicates of a particular outcome. Instead of a decision tree, we will use a technique called an adversary argument to establish better lower bounds for the selection problem.

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 did not 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, i.e., 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 in the worst case.

We want to find a lower bound on the complexity of a problem, not just a particular algorithm. Therefore, 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.

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 will use the terminology of contests, or tournaments, to describe the results of comparisons. The comparison that is found to be larger will be called the winner; the other will be called the loser.

3.2 Finding max and min

Throughout this section we will use the names max and min to refer to the largest and smallest keys, respectively, in a list of $n$ keys.
We can find \textit{max} and \textit{min} by using Algorithm 3 to find \textit{max}, eliminating \textit{max} from the list, and then using the appropriate variant of the algorithm to find \textit{min} among the remaining \(n-1\) keys. Thus \textit{max} and \textit{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 \textit{max} or \textit{min} independently, when finding both, some of the work can be “shared.” Exercise 1.12 asks for an algorithm to find \textit{max} and \textit{min} with only about \(3n/2\) key comparisons. The 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 this section we give an adversary argument to show that this solution is optimal. Specifically, we will prove

\textbf{Theorem 3.1} Any algorithm to find \textit{max} and \textit{min} of \(n\) keys by comparison of keys must do at least \(3n/2-2\) key comparisons in the worst case.

To establish the lower bound we may assume that the keys are distinct. To know that a key \(x\) is \textit{max} and that a key \(y\) is \textit{min}, an algorithm must know that every key other than \(x\) has lost some comparison and that every key other than \(y\) has won some comparison. If we count each win as one unit of information and each loss as one unit of information, then an algorithm must have (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 units of new information as possible with each comparison. Imagine the adversary constructing a specific input list 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:

<table>
<thead>
<tr>
<th>Key status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(W)</td>
<td>Has won at least one comparison and never lost</td>
</tr>
<tr>
<td>(L)</td>
<td>Has lost at least one comparison and never won</td>
</tr>
<tr>
<td>(WL)</td>
<td>Has won and lost at least one comparison</td>
</tr>
<tr>
<td>(N)</td>
<td>Has not yet participated in a comparison</td>
</tr>
</tbody>
</table>

The adversary strategy is described in Table 3.1. 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 3.1 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 \(y\), that the adversary chooses \(x\) as the winner, and that \(x\) has not yet

\begin{table}[h]
\centering
\caption{The adversary strategy for the \textit{min} and \textit{max} problem.}
\begin{tabular}{|c|c|c|c|}
\hline
Status of keys \(x\) and \(y\) compared by an algorithm & Adversary response & New status & Units of new information \\
\hline
\(N, N\) & \(x>y\) & \(W, L\) & 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 & & 0 \\
\hline
\end{tabular}
\end{table}

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 charge \(x\)'s value to make it beat \(y\) without contradicting any of the responses it gave earlier. The other situation, where the key chosen as the loser has never won, can be handled similarly — by reducing the value of the key if necessary. So the 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.

\textbf{Example 3.1} Constructing an input using the adversary’s rules

The first column in Table 3.2 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 \(x_4\) 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 \(x_6\) and \(x_4\) consistent with its rules. After

\begin{table}[h]
\centering
\caption{An example of the adversary strategy.}
\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline
Comparison & \(x_1\) & \(x_2\) & \(x_3\) & \(x_4\) & \(x_5\) & \(x_6\) \\
\hline
\(x_1, x_2\) & \(W\) & 20 & \(L\) & 10 & \(N\) & * & \(N\) & * & \(N\) & * & \(N\) & * & \(L\) & 5 & \(L\) & 12 \\
\(x_1, x_3\) & \(W\) & 20 & \(W\) & 15 & \(L\) & 8 & \(L\) & 5 & \(L\) & 12 \\
\(x_3, x_4\) & \(W\) & 15 & \(W\) & 15 & \(L\) & 8 & \(L\) & 5 & \(L\) & 12 \\
\(x_3, x_5\) & \(W\) & 15 & \(W\) & 25 & \(L\) & 8 & \(L\) & 5 & \(L\) & 12 \\
\(x_3, x_1\) & \(WL\) & 20 & \(WL\) & 10 & \(L\) & 8 & \(WL\) & 5 & \(L\) & 3 & \(WL\) & 3 \\
\hline
\end{tabular}
\end{table}
the first five comparisons, every key except \( x_3 \) has lost at least once, so \( x_3 \) is \( \text{max} \). After the last comparison \( x_4 \) is the only key that has never won, so it is \( \text{min} \). 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 3.1, we need only 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. Thus to get \( 2n - 2 \) units of information, an algorithm 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 3.1.

3.3 Finding the Second-Largest Key

3.3.1 Introduction

Throughout this section we will use \( \text{max} \) and \( \text{secondLargest} \) 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 \( \text{FindMax} \) (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 \( \text{max} \) can be used to decrease the number of comparisons performed in finding \( \text{secondLargest} \). Specifically, any key that loses to a key other than \( \text{max} \) cannot possibly be \( \text{secondLargest} \). All such keys discovered while finding \( \text{max} \) may be ignored during the second pass through the list. (The problem of keeping track of them will be considered later.)

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

<table>
<thead>
<tr>
<th>Comparands</th>
<th>Winner</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L[1], L[2] )</td>
<td>( L[1] )</td>
</tr>
<tr>
<td>( L[1], L[3] )</td>
<td>( L[1] )</td>
</tr>
</tbody>
</table>

Then \( \text{max} = L[4] \) and \( \text{secondLargest} \) is either \( L[5] \) or \( L[1] \) because both \( L[2] \) and \( L[3] \) lost to \( L[1] \). Thus only one more comparison is needed to find \( \text{secondLargest} \) in this example.

It may happen, however, that during the first pass through the list to find \( \text{max} \) we do not obtain any information useful for finding \( \text{secondLargest} \). If \( \text{max} \) were \( L[1] \), then each other key would be compared only with \( \text{max} \). Does this mean that in the worst case \( 2n-3 \) comparisons must be done to find \( \text{secondLargest} \)? Not necessarily. In the preceding discussion we used a specific algorithm. No algorithm can find \( \text{max} \) by doing fewer than \( n-1 \) comparisons, but another algorithm may provide more information useful for eliminating some keys during the second pass through the list. The tournament method, described next, provides such information.

3.3.2 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 sits for the next round.) A tournament can be described by a tree diagram as shown in Fig. 3.1. Each leaf contains a key, and at each subsequent level the parent of each pair contains the winner. The root will contain the largest key. As in Algorithm 1.3, \( n-1 \) comparisons are done to find \( \text{max} \).

In the process of finding \( \text{max} \), every key except \( \text{max} \) loses in one comparison. How many lose directly to \( \text{max} \)? Roughly half the keys in one round will be losers and will not appear in the next round. If \( n \) is a power of 2, there are exactly \( \log_2 n \) rounds; in general, the number of rounds is \( \lceil \log_2 n \rceil \). Since \( \text{max} \) is involved in at most

![An example of a tournament; \( \text{max} = L[6]; \text{secondLargest} \) may be \( L[4], L[5], \) or \( L[7] \).](image)

Figure 3.1
one comparison in each round, there are at most $\lceil \lg n \rceil$ keys that lost only to $\max$, and thus could possibly be $\text{secondLargest}$. The method of Algorithm 1.3 can be used to find the largest of these $\lceil \lg n \rceil$ keys by doing $\lceil \lg n \rceil - 1$ comparisons. Thus the tournament finds $\max$ and $\text{secondLargest}$ by doing a total of $n + \lceil \lg n \rceil - 2$ comparisons. This is an improvement over our first result of $2n - 3$. Can we do better?

### 3.3.3 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 $\text{secondLargest}$ 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 $\text{secondLargest}$ loses must, of course, be $\max$. This argument gives a lower bound on the number of comparisons needed to find $\text{secondLargest}$, 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 $\text{secondLargest}$ 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 3.2.** Any algorithm (that works by comparing keys) to find the second largest in a list 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 $\text{secondLargest}$ 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 $\text{secondLargest}$ to compare $\max$ to $\lceil \lg n \rceil$ distinct keys.

The adversary assigns a "weight" $w(x)$ to each key $x$ in the list. 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.

<table>
<thead>
<tr>
<th>Case</th>
<th>Adversary reply</th>
<th>Updating of weights</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w(x) &gt; w(y)$</td>
<td>$x &gt; y$</td>
<td>$w(x) := w(x) + w(y)$; $w(y) := 0$</td>
</tr>
<tr>
<td>$w(x) = w(y)$</td>
<td>Same as above</td>
<td>Same as above</td>
</tr>
<tr>
<td>$w(x) &gt; w(y)$</td>
<td>$y &gt; x$</td>
<td>$w(y) := w(y) + w(x)$; $w(x) := 0$</td>
</tr>
<tr>
<td>$w(x) = w(y)$</td>
<td>Consistent with previous replies</td>
<td>No change</td>
</tr>
</tbody>
</table>

We need to verify that if the adversary follows this strategy, its replies are consistent with some input, and that $\max$ 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 $\text{secondLargest}$ incorrect.

**Lemma 3.3** Let $x$ be the key that has nonzero weight when the algorithm stops. Then $x = \max$, and $x$ has directly won against at least $\lceil \lg n \rceil$ distinct keys.

**Proof.** By facts 1, 3, and 4, when the algorithm stops, $w(x) = n$. Let $w_i = w(x)$ just after the $k$th comparison won by $x$ against a previously undefeated key. Then by the adversary’s rules,

$$w_i \leq 2w_{i-1}.$$ 

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 $K \geq \lceil \lg n \rceil$, 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 will not be counted again (even if an algorithm foolishly compares it to $x$ again).

Another way of looking at the adversary’s activity is that it 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 3.2 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.

**Example 3.2** 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 list that have not yet been specified are denoted by asterisks. Thus initially the keys are $*, *, *, *, *$. Note that values assigned to some keys may be changed at a later time. See Table 3.3, which shows just the first few comparisons (those that find $\max$, but not enough to find $\text{secondLargest}$). The weights and the values assigned to the keys will not be changed by any subsequent comparisons.

$$
\begin{array}{c|c|c}
\text{Case} & \text{Adversary reply} & \text{Updating of weights} \\
\hline
w(x) > w(y) & x > y & w(x) := w(x) + w(y); w(y) := 0 \\
w(x) = w(y) & \text{Same as above} & \text{Same as above} \\
w(x) > w(y) & y > x & w(y) := w(y) + w(x); w(x) := 0 \\
w(x) = w(y) & \text{Consistent with previous replies} & \text{No change} \\
\end{array}
$$
Table 3.3
An example of the adversary strategy.

<table>
<thead>
<tr>
<th>Comparands</th>
<th>Weights</th>
<th>Winner</th>
<th>New weights</th>
<th>Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>L[1], L[2]</td>
<td>w(L[1]) = w(L[2])</td>
<td>L[1]</td>
<td>2,0,1,1,1</td>
<td>20,10,<em>,</em>,*</td>
</tr>
<tr>
<td>L[1], L[3]</td>
<td>w(L[1]) &gt; w(L[3])</td>
<td>L[1]</td>
<td>3,0,0,1,1</td>
<td>20,10,15,<em>,</em></td>
</tr>
<tr>
<td>L[4], L[5]</td>
<td>w(L[4]) = w(L[5])</td>
<td>L[5]</td>
<td>3,0,0,0,2</td>
<td>20,10,15,30,40</td>
</tr>
<tr>
<td>L[1], L[5]</td>
<td>w(L[1]) &gt; w(L[5])</td>
<td>L[1]</td>
<td>5,0,0,0,0</td>
<td>41,10,15,30,40</td>
</tr>
</tbody>
</table>

3.3.4 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. This can be done by using an extra array of pointers or by careful indexing if the keys may be moved so that the winner is always placed in, say, the higher-indexed cell of the two being compared. We leave the choice and the details to the reader.

After max has been found by the tournament, only those keys that lose to it are to be compared to find secondLargest. How can we keep track of the elements that lose to max when we do not know in advance which key is max? One way is to maintain linked lists of keys that lose to each undefeated key. This can be done by allocating an array for links indexed to correspond to the keys. Initially all links are zero. After each comparison in the tournament, the key that lost would be added to the winner’s loser list (at the beginning of the list); see Fig. 3.3 for an example. When the tournament is complete and max has been found, it is easy to find secondLargest by traversing max’s loser list.

Time and Space

The tournament method for finding max and secondLargest uses Θ(n) extra space for links. The running time of the algorithm is in Θ(n + [lg n]−2) = Θ(n), since the number of operations for the links is roughly proportional to the number of comparisons done.

We can find the largest and second-largest keys in a list by using FindMax twice, doing 2n−3 comparisons, or we can use the more complicated tournament method, doing n+[lg n]−2 comparisons at most. Which method is better? The results of Exercise 3.8 should be instructive. Both algorithms are in Θ(n). Since the tournament method does more instructions per comparison while finding max, it may well be slower. It is also more complicated. The main point of considering this problem was not to find an algorithm that beats the straightforward one in practice, but to illustrate the adversary argument for the lower bound and, by exhibiting both the adversary argument and the tournament algorithm, to determine the optimal number of comparisons.