Sara Baase

Computer Algorithms

Introduction to Design and Analysis

SECOND EDITION

This is a clear, concise text on algorithm design and analysis, introducing the principles and techniques of computational complexity. It is the second edition of a popular text, featuring new chapters on adversary arguments and selection, dynamic programming, and parallel algorithms.

HIGHLIGHTS

- All algorithms are written in a modern language; using features of Pascal and Modula-2.
- Clear, accessible presentation of advanced topics such as lower bounds, NP-completeness, and parallel algorithms.
- Emphasizes the development of algorithms through a step-by-step process rather than merely presenting the end result.
- Over 300 exercises, including 100 exercises new to the second edition.

Computer Algorithms: Introduction to Design and Analysis, Second Edition teaches algorithms for solving real problems that arise frequently in computer applications. It emphasizes the importance of the algorithm process — continuously re-evaluating, modifying, and perhaps rejecting algorithms until a satisfactory solution is attained.

ABOUT THE AUTHOR

Sara Baase is Professor of Computer Science in the Mathematical Sciences Department at San Diego State University. She received a Ph.D. from the University of California, Berkeley, where she recently spent two years as a visiting professor. Dr. Baase is also the author of a VAX-II assembly language programming book. She received the San Diego State University Alumni Association's Outstanding Faculty Award in both 1982 and 1984.

ISBN 0-201-06035-3

Addison-Wesley Publishing Company

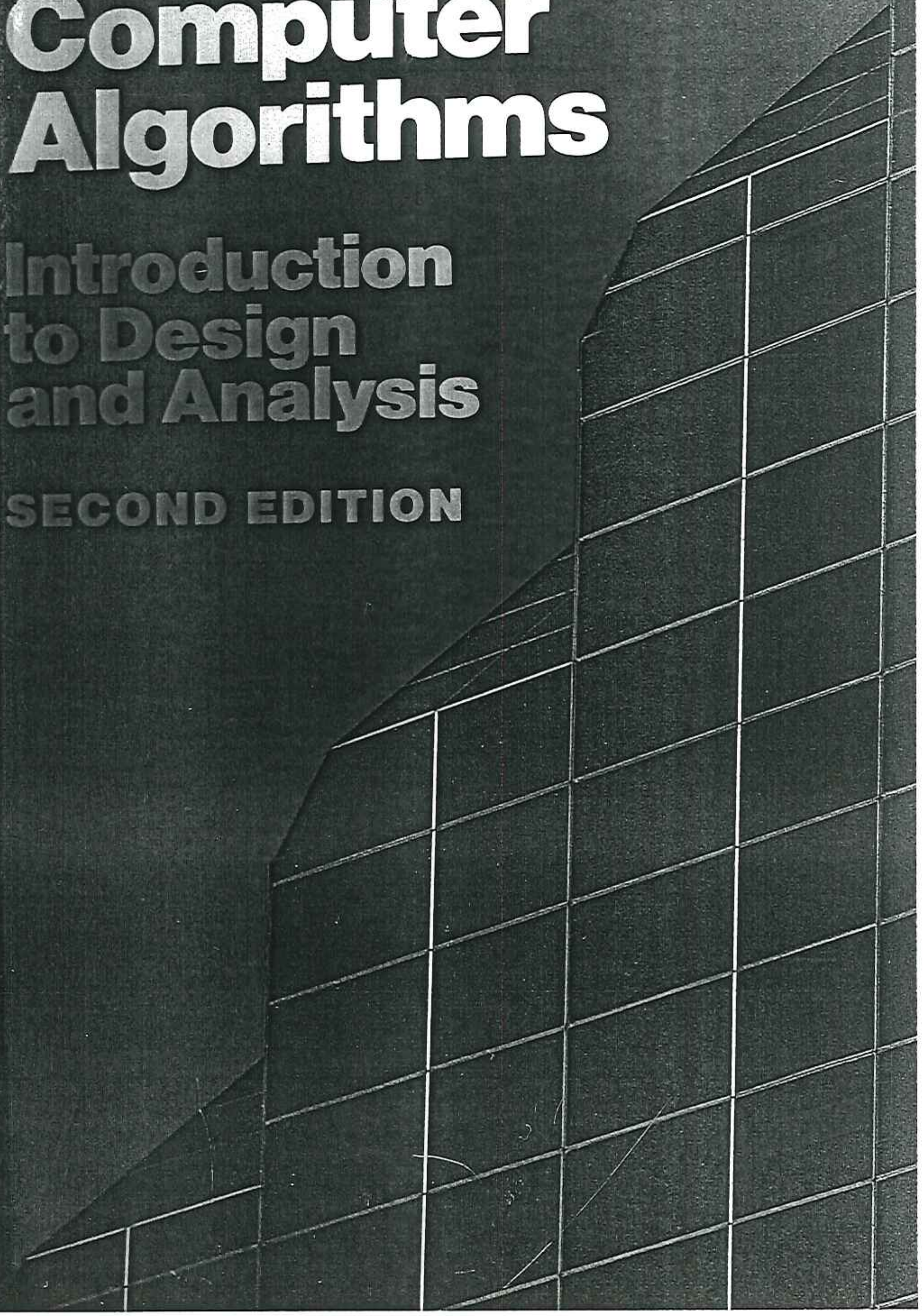
Ba Introduction to Design and Analysis

Sara Baase

Computer Algorithms

Introduction to Design and Analysis

SECOND EDITION



2

$$= k\,2^{k+1} - (2^{k+1} - 2) = (k-1)2^{k+1} + 2. \qquad \square$$

Summations Using Integration

Several summations that arise often in the analysis of algorithms can be approximated (or bounded from above or below) using integration. Suppose f is a continuous, decreasing function and that a and b are integers. Then, as Fig. 1.2 illustrates,

$$\int_a^{b+1} f(x)dx \le \sum_{i=a}^{b} f(i) \le \int_{a-1}^{b} f(x)dx. \tag{1.6}$$

Similarly, if f is an increasing function,

$$\int_{a-1}^{b} f(x)dx \le \sum_{i=a}^{b} f(i) \le \int_a^{b+1} f(x)dx. \tag{1.7}$$

Here are two examples that are used later in the text.

Example 1.1 An estimate for $\sum_{i=2}^{n} \frac{1}{i}$.

$$\sum_{i=2}^{n} \frac{1}{i} \le \int_1^n \frac{dx}{x} = \ln x \Big|_1^n = \ln n - \ln 1 = \ln n$$

Similarly,

$$\sum_{i=2}^{n} \frac{1}{i} \ge \ln(n+1) - \ln 2.$$

Thus

$$\sum_{i=2}^{n} \frac{1}{i} \approx \ln n. \tag{1.8}$$

■

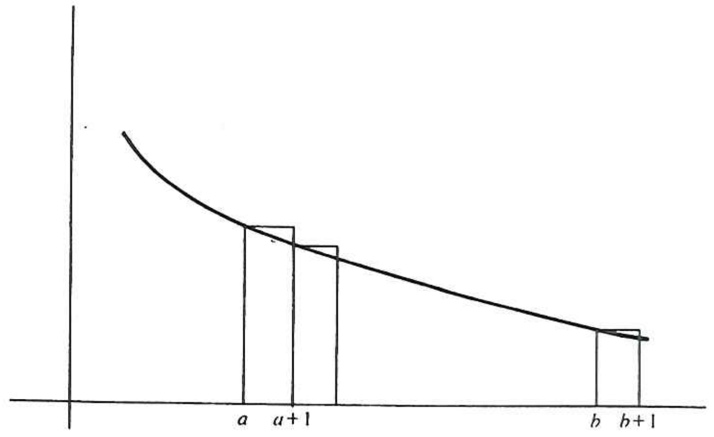
Example 1.2 A lower bound for $\sum_{i=1}^{n} \lg i$.

$$\sum_{i=1}^{n} \lg i \ge n \lg n - 1.5n \tag{1.9}$$

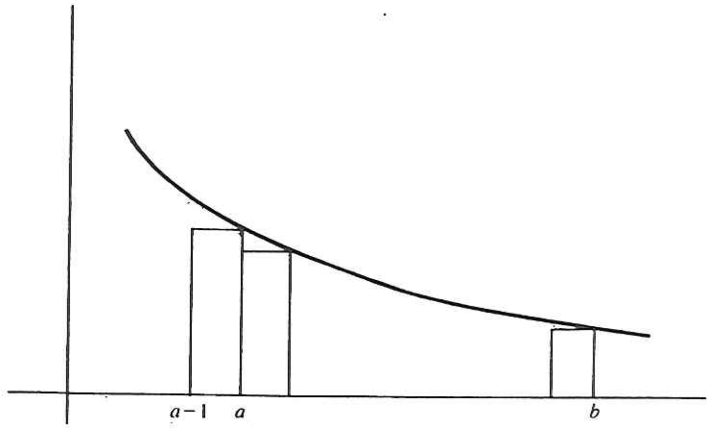
Proof. From Eq. 1.7 (and the observation that $\lg 1 = 0$) we have

$$\sum_{i=1}^{n} \lg i \ge \int_1^n \lg x\, dx.$$

$$\int_1^n \lg x\, dx = \int_1^n \lg e \ln x\, dx = \lg e \int_1^n \ln x\, dx$$



(a) $\int_a^{b+1} f(x)dx \leqslant \sum_{i=a}^{b} f(i)$



(b) $\sum_{i=a}^{b} f(i) \leqslant \int_{a-1}^{b} f(x)dx$

Figure 1.2 Approximating a sum of values of a decreasing function.

$$= \lg e\,[x \ln x - x]\Big|_1^n = \lg e\,(n \ln n - n + 1)$$

$$= n \lg n - n \lg e + \lg e \ge n \lg n - n \lg e.$$

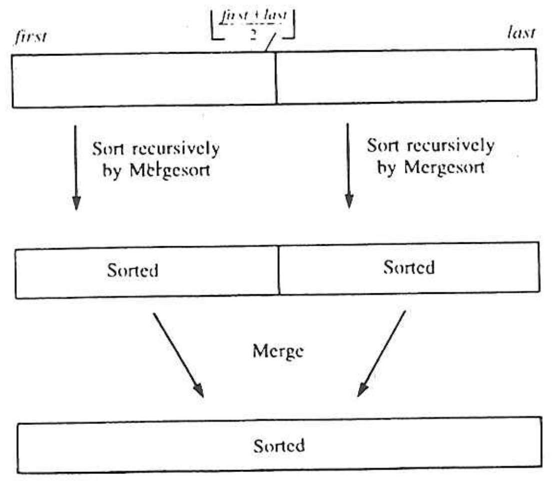

first
$\lfloor \frac{first+last}{2} \rfloor$
last
Sort recursively by Mergesort
Sort recursively by Mergesort
Sorted
Sorted
Merge
Sorted

Figure 2.10 Mergesort strategy.

The recurrence relation for the worst-case behavior of Mergesort is

$$W(n) = W(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil) + n - 1$$

$$W(1) = 0.$$

This is very similar to the informal analysis we did for the best case for Quicksort. Thus $W(n) \approx n\lg n - n$. So we finally have a sorting algorithm whose worst-case behavior is in $\Theta(n\lg n)$, but because of the extra space used for the merging, Mergesort is not an in-place sort.

2.4 Lower Bounds for Sorting by Comparison of Keys

In this section we derive lower bounds for the number of comparisons that must be done in the worst case and on the average by any algorithm that sorts by comparison of keys. To derive the lower bounds we will assume that the keys in the list to be sorted are distinct.

2.4.1 Decision Trees for Sorting Algorithms

Let n be fixed and suppose that the keys are $x_1, x_2, \ldots, x_n$. We will associate with each algorithm and positive integer n a (binary) decision tree that describes the sequence of comparisons carried out by the algorithm on any input of size n. Let Sort be any algorithm that sorts by comparison of keys. Each comparison has a two-way branch (since the keys are distinct), and we assume that Sort has an output

instruction that outputs the rearranged list of keys. The decision tree for Sort is defined inductively by associating a tree with each compare and output instruction as follows. The tree associated with an output instruction consists of one node labeled with the rearrangement of the keys. The tree associated with an instruction that compares keys x_i and x_j consists of a root labeled $(i:j)$, a left subtree that is the tree associated with the instruction executed next if $x_i < x_j$, and a right subtree that is the tree associated with the instruction executed next if $x_i > x_j$. The decision tree for Sort is the tree associated with the first compare instruction it executes. An example of a decision tree for $n=3$ is shown in Fig. 2.11.

The action of Sort on a particular input corresponds to following one path in its decision tree from the root to a leaf. The tree must have at least $n!$ leaves because there are $n!$ ways in which the keys may be permuted. Since the unique path followed for each input depends only on the ordering of the keys and not on their particular values, exactly $n!$ leaves can be reached from the root by actually executing Sort. We will assume that any paths in the tree that are never followed are removed. We also assume that comparison nodes with only one child are removed and replaced by the child, and that this "pruning" is repeated until all internal nodes have degree 2. The pruned tree represents an algorithm that is at least as efficient as the original one, so the lower bounds we derive using trees with exactly $n!$ leaves and all internal nodes of degree 2 will be valid lower bounds for all algorithms that sort by comparison of keys. From now on we assume Sort is described by such a tree.

The number of comparisons done by Sort on a particular input is the number of internal nodes on the path followed for that input. Thus the number of comparisons done in the worst case is the number of internal nodes on the longest path, and that is the depth of the tree. The average number of comparisons done is the average of the lengths of all paths from the root to a leaf. (For example, for $n=3$,

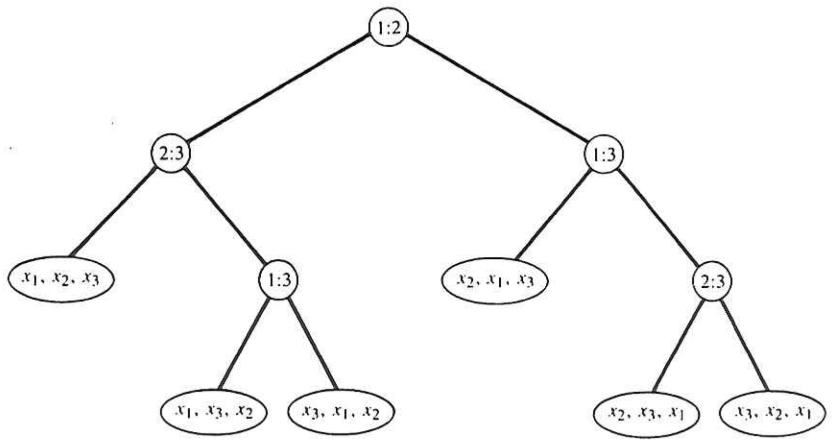


Figure 2.11 Decision tree for a sorting algorithm, $n=3$.

4

the algorithm whose decision tree is shown in Fig. 2.11 does three comparisons in the worst case and two and two-thirds on the average.)

2.4.2 Lower Bound for Worst Case

To get a worst-case lower bound for sorting by comparison, we derive a lower bound for the depth of a binary tree in terms of the number of leaves, since the only quantitative information we have about the decision trees is the number of leaves.

Lemma 2.4 Let l be the number of leaves in a binary tree and let d be its depth. Then $l \le 2^d$.

Proof. A straightforward induction on d. □

Lemma 2.5 Let l and d be as in Lemma 2.4. Then $d \ge \lceil \lg l \rceil$.

Proof. Taking logs of both sides of the inequality in Lemma 2.4 gives $\lg l \le d$. Since d is an integer, $d \ge \lceil \lg l \rceil$. □

Lemma 2.6 For a given n, the decision tree for any algorithm that sorts by comparison of keys has depth at least $\lceil \lg n! \rceil$.

Proof. Let $l = n!$ in Lemma 2.5. □

So the number of comparisons needed to sort in the worst case is at least $\lceil \lg n! \rceil$. Our best sort so far is Mergesort, but how close is $\lceil \lg n! \rceil$ to $n \lg n$? There are several ways to estimate or get a lower bound for $\lg n!$. Perhaps the simplest is to observe that

$$n! \ge n(n-1) \cdots (\lceil n/2 \rceil) \ge \left(\frac{n}{2}\right)^{n/2}.$$

So

$$\lg n! \ge \frac{n}{2} \lg \frac{n}{2},$$

which is in $\Theta(n \lg n)$. Thus we see already that Mergesort is of optimal order. To get a closer lower bound, we use the fact that

$$\lg n! = \sum_{j=1}^{n} \lg j.$$

Using Eq. 1.9 we get

$$\lg n! \ge n \lg n - 1.5n.$$

Thus the depth of the decision tree is at least $\lceil n \lg n - 1.5n \rceil$.

Theorem 2.7 Any algorithm to sort n items by comparisons of keys must do at least $\lceil \lg n! \rceil$, or approximately $\lceil n \lg n - 1.5n \rceil$, key comparisons in the worst case.

So Mergesort is very close to optimal. There is some difference between the exact behavior of Mergesort and the lower bound. Consider the case where $n = 5$. Insertion Sort does 10 comparisons in the worst case, and Mergesort does 8, but the lower bound is $\lceil \lg 5! \rceil = \lceil \lg 120 \rceil = 7$. Is the lower bound simply not good enough, or can we do better than Mergesort? The reader is encouraged to try to find a way to sort five keys with only seven comparisons in the worst case.

2.4.3 Lower Bound for Average Behavior

We need a lower bound on the average of the lengths of all paths from the root to a leaf. The *external path length* of a tree is the sum of the lengths of all paths from the root to a leaf; it will be denoted by *epl*.

A binary tree in which every node has degree 0 or 2 is called a *2-tree*. Our decision trees are 2-trees, so the next two lemmas give us a lower bound on their epl.

Lemma 2.8 Among 2-trees with l leaves, the epl is minimized only if all the leaves are on at most two adjacent levels.

Proof. Suppose we have a 2-tree with depth d that has a leaf X at level k, where $k \le d-2$. We will exhibit a 2-tree with the same number of leaves and lower epl. Choose a node Y at level $d-1$ that is not a leaf, remove its children, and attach two children to X. (See Fig. 2.12 for an illustration.) The total number of leaves has not

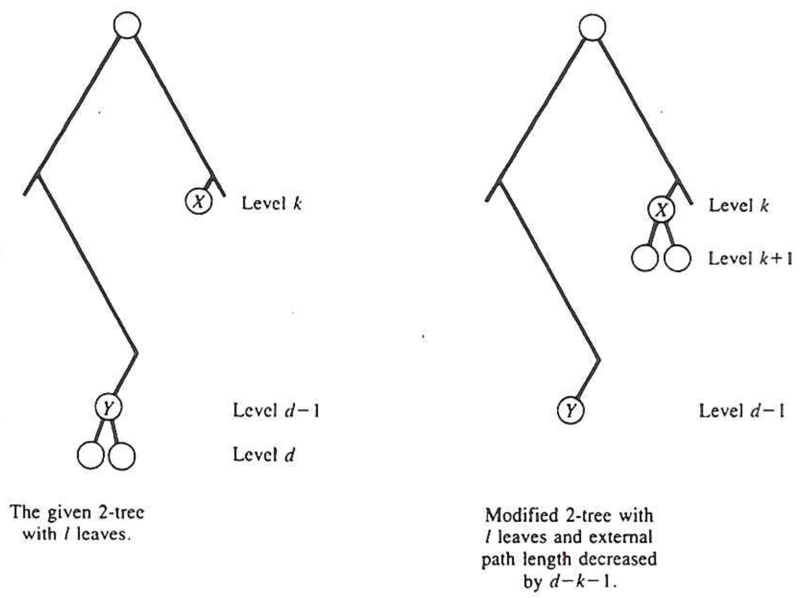

X
Level k
Y
Level $d-1$
Level d
The given 2-tree with l leaves.
X
Level k
Level $k+1$
Y
Level $d-1$
Modified 2-tree with l leaves and external path length decreased by $d-k-1$.

Figure 2.12 Decreasing external path length.

S

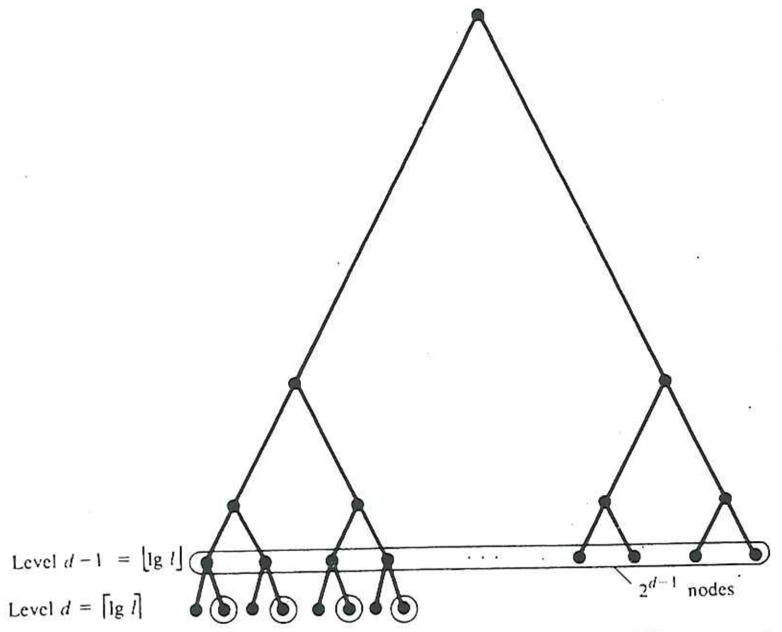
changed. The epl has been decreased by $2d+k$, because the paths to the children of Y and the path to X are no longer counted, and increased by $2(k+1)+d-1 = 2k+d+1$, the sum of the lengths of the paths to Y and the new children of X. There is a net decrease in the epl of $2d+k-(2k+d+1) = d-k-1 > 0$ since $k \le d-2$. □

Lemma 2.9 The minimum epl for 2-trees with l leaves is $l\lfloor \lg l \rfloor + 2(l-2^{\lfloor \lg l \rfloor})$.

Proof. If l is a power of 2, all the leaves are at level $\lg l$. (This statement depends on both the facts that the tree is a 2-tree and that all leaves are on at most two levels. The reader should verify it.) The epl is $l \lg l$, which is the value of the expression in the lemma in this case.

If l is not a power of 2, the depth of the tree is $d = \lceil \lg l \rceil$ and all the leaves are at levels $d-1$ and d. The sum of the path lengths (for all leaves) down to level $d-1$ is $l(d-1)$. For each leaf at level d, 1 must be added to get the total epl. The number of leaves at level d is $2(l-2^{d-1})$, since for each node at level $d-1$ that is not a leaf, there are two leaves at level d. (See Fig. 2.13.) Thus the sum is $l(d-1)+2(l-2^{d-1})$ $= l\lfloor \lg l \rfloor + 2(l-2^{\lfloor \lg l \rfloor})$. □

Lemma 2.10 The average path length in a 2-tree with l leaves is at least $\lfloor \lg l \rfloor$.



Level $d-1 = \lfloor \lg l \rfloor$
Level $d = \lceil \lg l \rceil$
2^{d-1} nodes

Figure 2.13 Computing external path length for Lemma 2.9. $l = 2^{d-1}$ + the number of nodes at level $d-1$ that are not leaves.

Proof. The minimum average path length is

$$\frac{l\lfloor \lg l \rfloor + 2(l-2^{\lfloor \lg l \rfloor})}{l} = \lfloor \lg l \rfloor + \varepsilon,$$

where $0 \le \varepsilon < 1$ since $l-2^{\lfloor \lg l \rfloor}$ is always less than $l/2$. □

Theorem 2.11 The average number of comparisons done by an algorithm to sort n items by comparison of keys is at least $\lfloor \lg n! \rfloor \approx \lfloor n \lg n - 1.5n \rfloor$.

Thus no algorithm can do substantially better than Quicksort and Mergesort on the average.

2.5 Heapsort

2.5.1 Heaps

The Heapsort algorithm uses a data structure called a *heap*, which is a binary tree with some special properties. The definition of a heap includes a description of the structure and a condition on the data in the nodes. Informally, a heap structure is a complete binary tree with some of the rightmost leaves removed. (See Fig. 2.14 for illustrations.) Let S be a set of keys with a linear ordering and let T be a binary tree with depth d whose nodes contain elements of S. T is a heap if and only if it satisfies the following conditions:[2]

1. There are 2^l nodes at level l, for $1 \le l \le d-1$. At level $d-1$ the leaves are all to the right of the internal nodes. The rightmost internal node at level $d-1$ may have degree 1 (with no right child); the others all have degree 2.
2. The key at any node is greater than or equal to the keys at each of its children (if it has any).

We will use the term *heap structure* to describe a binary tree that satisfies condition (1). Observe that a complete binary tree is a heap structure. When new nodes are added to a heap, they must be added left to right at the bottom level, and if a node is removed, it must be the rightmost node at the bottom level if the resulting structure is still to be a heap. Note that the root must contain the largest key in the heap.

2.5.2 The Heapsort Strategy

If the keys to be sorted are arranged in a heap, then we can build a sorted list in reverse order by repeatedly removing the key from the root (the largest remaining key), filling the output array from the end, and rearranging the keys still in the heap to reestablish the heap property, thus bringing the next largest key to the root. Since

[2] Warning: Some texts use slightly different definitions. Frequently heaps are defined so that the key at a node is less than or equal to the keys at its children.

9

3.1 Introduction

3.1.1 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 kth 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 $k = \lceil n/2 \rceil$, 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 $\Theta(n\lg n)$ 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 $\Theta(n\lg n)$? 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 $\lceil \lg l \rceil$, where l is the number of leaves.

In Chapter 1 we used decision trees to get the (worst-case) lower bound of $\lfloor \lg n \rfloor + 1$ 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 $\lceil n\lg n - 1.5n \rceil$, 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 list may be the output, i.e., the kth 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 *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 comparand 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 *max* and *min* by using Algorithm 1.3 to find *max*, eliminating *max* from the list, and then using the appropriate variant of the algorithm 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 1.12 asks for an algorithm to find *max* and *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

Theorem 3.1 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.

To establish the lower bound we may assume that the keys are distinct. To know that a key x is *max* and that a key y is *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:

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

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 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 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.

Table 3.1
The adversary strategy for the min and max problem.

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

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_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_6 and x_4 consistent with its rules. After

Table 3.2
An example of the adversary strategy.

Comparison	x_1 Status	x_1 Value	x_2 Status	x_2 Value	x_3 Status	x_3 Value	x_4 Status	x_4 Value	x_5 Status	x_5 Value	x_6 Status	x_6 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

8

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 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 *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* 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:

Comparands	*Winner*
$L[1], L[2]$	$L[1]$
$L[1], L[3]$	$L[1]$
$L[1], L[4]$	$L[4]$
$L[4], L[5]$	$L[4]$

Then *max* = $L[4]$ and *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 *secondLargest* in this example.

It may happen, however, that during the first pass through the list to find *max* we do not obtain any information useful for finding *secondLargest*. If *max* were $L[1]$, then each other key would be compared only with *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. No algorithm can find *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 waits 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 *max*.

In the process of finding *max*, every key except *max* loses in one comparison. How many lose directly to *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 $\lg n$ rounds; in general, the number of rounds is $\lceil \lg n \rceil$. Since *max* is involved in at most

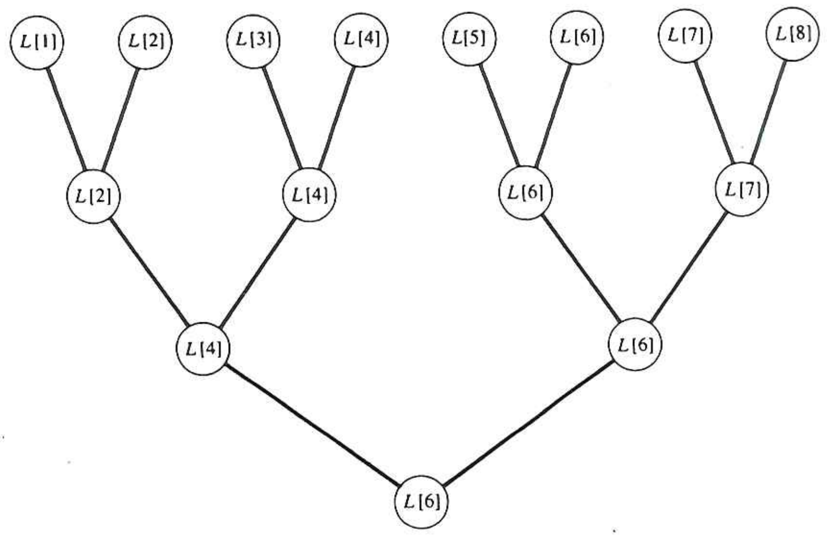


Figure 3.1 An example of a tournament; *max* = $L[6]$; *secondLargest* may be $L[4]$, $L[5]$, or $L[7]$.

6

one comparison in each round, there are at most $\lceil \lg n \rceil$ keys that lost only to *max*, and thus 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 $\lceil \lg n \rceil-1$ comparisons. Thus the tournament finds *max* and *secondLargest* by doing a total of $n+\lceil \lg n \rceil-2$ comparisons. 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 *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 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 *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 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.

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

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 *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 $w_k = w(x)$ just after the kth comparison won by x against a previously undefeated key. Then by the adversary's rules,

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

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

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

Thus $K \ge \lg n$, and since K is an integer, $K \ge \lceil \lg n \rceil$. 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 *secondLargest*). The weights and the values assigned to the keys will not be changed by any subsequent comparisons. ■

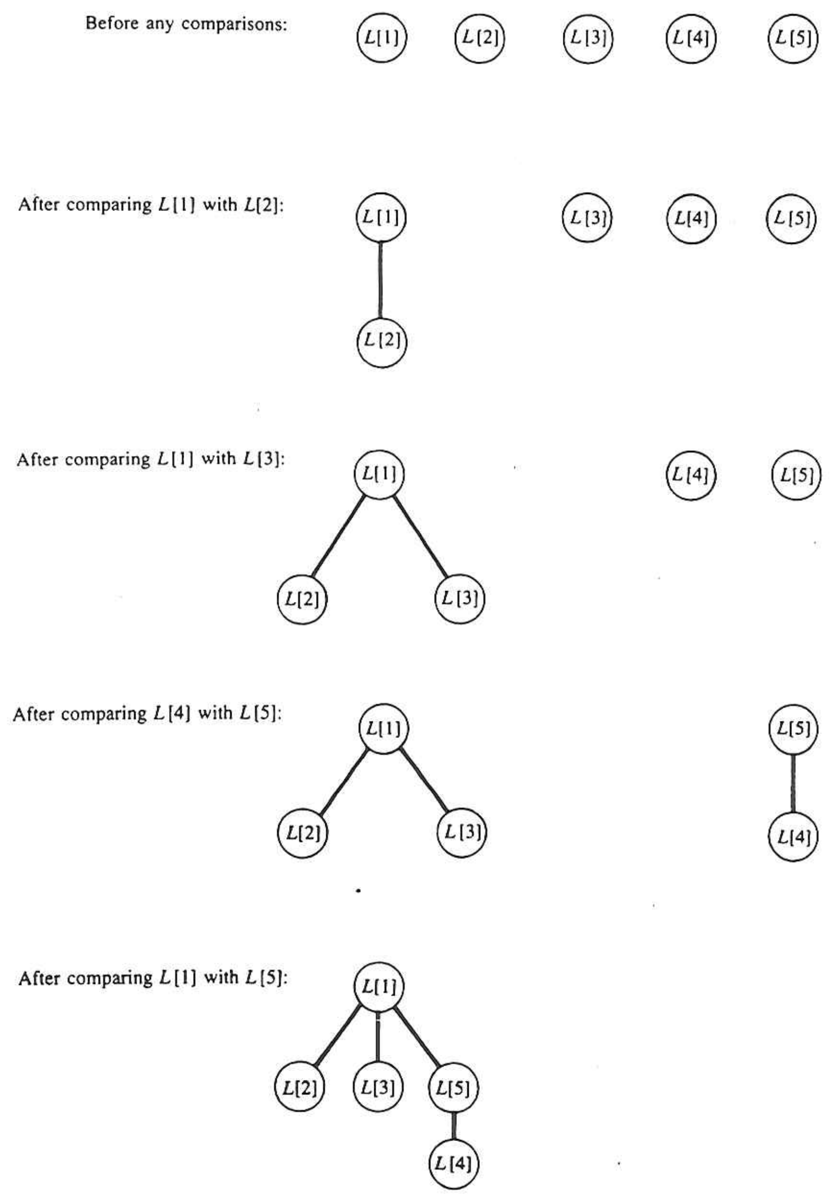

Before any comparisons:
After comparing $L[1]$ with $L[2]$:
After comparing $L[1]$ with $L[3]$:
After comparing $L[4]$ with $L[5]$:
After comparing $L[1]$ with $L[5]$:
L[1]
L[2]
L[3]
L[4]
L[5]

Figure 3.2 Trees for the adversary decisions in Example 3.2.

Table 3.3
An example of the adversary strategy.

Comparands	Weights	Winner	New weights	Keys
$L[1], L[2]$	$w(L[1]) = w(L[2])$	$L[1]$	2,0,1,1,1	20,10,*,*,*
$L[1], L[3]$	$w(L[1]) > w(L[3])$	$L[1]$	3,0,0,1,1	20,10,15,*,*
$L[5], L[4]$	$w(L[5]) = w(L[4])$	$L[5]$	3,0,0,0,2	20,10,15,30,40
$L[1], L[5]$	$w(L[1]) > w(L[5])$	$L[1]$	5,0,0,0,0	41,10,15,30,40

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 $\Theta(n)$ extra space for links. The running time of the algorithm is in $\Theta(n+\lceil\lg n\rceil-2) = \Theta(n)$, 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 $n+\lceil\lg n\rceil-2$ comparisons at most. Which method is better? The results of Exercise 3.8 should be instructive. Both algorithms are in $\Theta(n)$. 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.

11

3.5 A Lower Bound for Finding the Median

We are assuming that L is a list 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 Fig. 3.6. 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 Fig. 3.7(a) for an example.) An adversary could change the value of y, moving it to the opposite side of *median*, as in Fig. 3.7(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 Fig. 3.6, there are $n-1$ branches, so at least $n-1$ comparisons must be done. This is neither a surprising nor an 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 Fig. 3.6.

Definition 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 $y \leq median$. (This is the comparison that 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.)

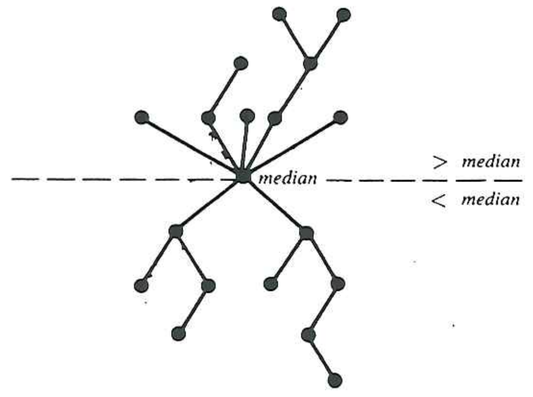
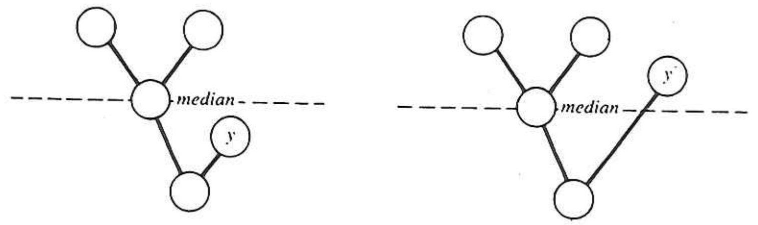


Figure 3.6 Comparisons relating each key to *median*.



(a) $y < median$.

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

Figure 3.7 An adversary conquers a bad algorithm.

Comparisons of x and y where $x > median$ and $y < median$ are noncrucial. We will exhibit an adversary that forces an algorithm to perform such 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 Larger than *median*.
S	Has been assigned a value Smaller than *median*.
N	Has not yet been in a comparison.

The adversary's strategy is summed up in Table 3.4. 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

Table 3.4
The adversary strategy for the median-finding problem.

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.

12

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 3.4 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 cannot 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) $+ (n-1)/2$ (the noncrucial comparisons). We sum up the result in the following theorem.

Theorem 3.4 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.

Exercises

Section 3.1: Introduction

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

*3.2. Use a decision tree argument to get a lower bound on the number of comparisons needed to merge two sorted lists each containing n keys. How does your result compare with the lower bound derived in Section 2.3.6?

Section 3.2: Finding max *and* min

3.3. 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 3.3: Finding the Second-Largest Key

3.4. Write an algorithm in detail for the tournament method to find *max* and *secondLargest*.

3.5. How many comparisons are done by the tournament method to find *secondLargest* on the average if n is a power of 2?

3.6. The following algorithm finds the largest and second largest keys in an array L 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 $L[1]>L[2]$ then
   $max := L[1]$;
   $second := L[2]$
else
   $max := L[2]$;
   $second := L[1]$
end { if };
for $i := 3$ to $n$ do
   if $L[i]>second$ then
      if $L[i]>max$ then
         $second := max$;
         $max := L[i]$
      else $second := L[i]$
      end { if $L[i]>max$ }
   end { if $L[i]>second$ }
end { for }
```

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?

*3.7. 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*?

3.8. Write assembly language routines to find *max* and *secondLargest* by the tournament method and by using Algorithm 1.3 twice. How many instructions are executed in the worst case by each routine?

3.9. The Replacement Selection algorithm of Section 2.8.3 can be adapted to find *max* and *secondLargest*. What are the pros and cons of using Replacement Selection instead of the tournament method?

Section 3.4: The Selection Problem

3.10. Show that the median of five keys can be found with only six comparisons in the worst case. (Recall that at least seven comparisons are needed to sort five keys.)

3.11. 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 list completely. Write a modified Quicksort algorithm called *FindKth* for this purpose. Analyze your algorithm.

3.12. Suppose we use the following algorithm to find the k largest keys in a list of n keys.

```
Build a heap out of the $n$ keys;
for $i := 1$ to $k$ do
   remove and output the key at the root;
   Use the FixHeap procedure to rearrange the remaining
      keys as needed to reestablish the heap property
end
```

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