From: Computer Algorithms, Introduction to Design & Analysis, 3rd Edition
Sara Baase and Allen Van Gelder

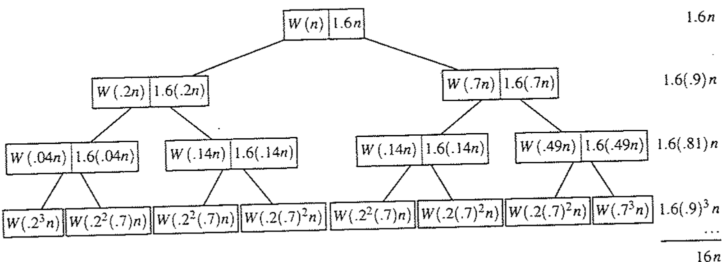

Figure 5.4 Recursion tree for select

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.

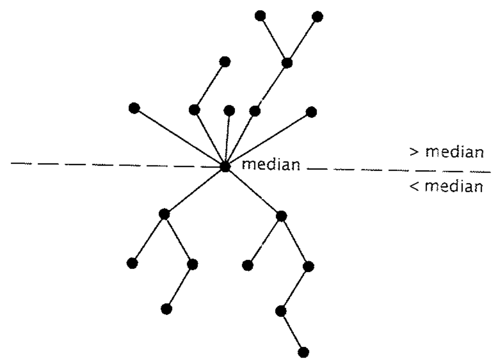

Figure 5.5 Comparisons relating each key to median

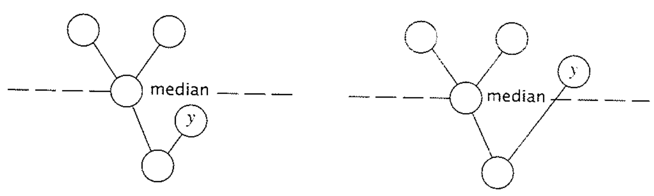

Figure 5.6 An adversary conquers a bad algorithm

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.