15

Lower Bounds

How do I know if I have a good algorithm to solve a problem? If my algorithm runs in $\Theta(n \log n)$ time, is that good? It would be if I were sorting the records stored in an array. But it would be terrible if I were searching the array for the largest element. The value of an algorithm must be determined in relation to the inherent complexity of the problem at hand.

In Section 3.6 we defined the upper bound for a problem to be the upper bound of the best algorithm we know for that problem, and the lower bound to be the tightest lower bound that we can prove over all algorithms for that problem. While we usually can recognize the upper bound for a given algorithm, finding the tightest lower bound for all possible algorithms is often difficult, especially if that lower bound is more than the "trivial" lower bound determined by measuring the amount of input that must be processed.

The benefits of being able to discover a strong lower bound are significant. In particular, when we can make the upper and lower bounds for a problem meet, this means that we truly understand our problem in a theoretical sense. It also saves us the effort of attempting to discover more (asymptotically) efficient algorithms when no such algorithm can exist.

Often the most effective way to determine the lower bound for a problem is to find a reduction to another problem whose lower bound is already known. This is the subject of Chapter 17. However, this approach does not help us when we cannot find a suitable "similar problem." Our focus in this chapter is discovering and proving lower bounds from first principles. Our most significant example of a lower bounds argument so far is the proof from Section 7.9 that the problem of sorting is $\mathrm{O}(n \log n)$ in the worst case.

Section 15.1 reviews the concept of a lower bound for a problem and presents the basic "algorithm" for finding a good algorithm. Section 15.2 discusses lower bounds on searching in lists, both those that are unordered and those that are ordered. Section 15.3 deals with finding the maximum value in a list, and presents a model for selection based on building a partially ordered set. Section 15.4 presents

the concept of an adversarial lower bounds proof. Section 15.5 illustrates the concept of a state space lower bound. Section 15.6 presents a linear time worst-case algorithm for finding the ith biggest element on a list. Section 15.7 continues our discussion of sorting with a quest for the algorithm that requires the absolute fewest number of comparisons needed to sort a list.

15.1 Introduction to Lower Bounds Proofs

The lower bound for the problem is the tightest (highest) lower bound that we can prove *for all possible algorithms* that solve the problem.[1] This can be a difficult bar, given that we cannot possibly know all algorithms for any problem, because there are theoretically an infinite number. However, we can often recognize a simple lower bound based on the amount of input that must be examined. For example, we can argue that the lower bound for any algorithm to find the maximum-valued element in an unsorted list must be $\Omega(n)$ because any algorithm must examine all of the inputs to be sure that it actually finds the maximum value.

In the case of maximum finding, the fact that we know of a simple algorithm that runs in $\mathrm{O}(n)$ time, combined with the fact that any algorithm needs $\Omega(n)$ time, is significant. Because our upper and lower bounds meet (within a constant factor), we know that we do have a "good" algorithm for solving the problem. It is possible that someone can develop an implementation that is a "little" faster than an existing one, by a constant factor. But we know that its not possible to develop one that is asymptotically better.

We must be careful about how we interpret this last statement, however. The world is certainly better off for the invention of Quicksort, even though Mergesort was available at the time. Quicksort is not asymptotically faster than Mergesort, yet is not merely a "tuning" of Mergesort either. Quicksort is a substantially different approach to sorting. So even when our upper and lower bounds for a problem meet, there are still benefits to be gained from a new, clever algorithm.

So now we have an answer to the question "How do I know if I have a good algorithm to solve a problem?" An algorithm is good (asymptotically speaking) if its upper bound matches the problem's lower bound. If they match, we know to stop trying to find an (asymptotically) faster algorithm. What if the (known) upper bound for our algorithm does not match the (known) lower bound for the problem? In this case, we might not know what to do. Is our upper bound flawed, and the algorithm is really faster than we can prove? Is our lower bound weak, and the true lower bound for the problem is greater? Or is our algorithm simply not the best?

[1] Throughout this discussion, it should be understood that any mention of bounds must specify what class of inputs are being considered. Do we mean the bound for the worst case input? The average cost over all inputs? Regardless of which class of inputs we consider, all of the issues raised apply equally.

Now we know precisely what we are aiming for when designing an algorithm: We want to find an algorithm who's upper bound matches the lower bound of the problem. Putting together all that we know so far about algorithms, we can organize our thinking into the following "algorithm for designing algorithms."[2]

If the upper and lower bounds match,
then stop,
else if the bounds are close or the problem isn't important,
  then stop,
  else if the problem definition focuses on the wrong thing,
    then restate it,
    else if the algorithm is too slow,
      then find a faster algorithm,
      else if lower bound is too weak,
        then generate a stronger bound.

We can repeat this process until we are satisfied or exhausted.

This brings us smack up against one of the toughest tasks in analysis. Lower bounds proofs are notoriously difficult to construct. The problem is coming up with arguments that truly cover all of the things that *any* algorithm possibly *could* do. The most common fallacy is to argue from the point of view of what some good algorithm actually *does* do, and claim that any algorithm must do the same. This simply is not true, and any lower bounds proof that refers to specific behavior that must take place should be viewed with some suspicion.

Let us consider the Towers of Hanoi problem again. Recall from Section 2.5 that our basic algorithm is to move $n-1$ disks (recursively) to the middle pole, move the bottom disk to the third pole, and then move $n-1$ disks (again recursively) from the middle to the third pole. This algorithm generates the recurrence $\mathbf{T}(n) = 2\mathbf{T}(n-1)+1 = 2^n - 1$. So, the upper bound for our algorithm is $2^n - 1$. But is this the best algorithm for the problem? What is the lower bound for the problem?

For our first try at a lower bounds proof, the "trivial" lower bound is that we must move every disk at least once, for a minimum cost of n. Slightly better is to observe that to get the bottom disk to the third pole, we must move every other disk at least twice (once to get them off the bottom disk, and once to get them over to the third pole). This yields a cost of $2n-1$, which still is not a good match for our algorithm. Is the problem in the algorithm or in the lower bound?

We can get to the correct lower bound by the following reasoning: To move the biggest disk from first to the last pole, we must first have all of the other $n-1$ disks out of the way, and the only way to do that is to move them all to the middle pole (for a cost of at least $\mathbf{T}(n-1)$). We then must move the bottom disk (for a cost of

[2]This is a minor reformulation of the "algorithm" given by Gregory J.E. Rawlins in his book "Compared to What?"

at least one). After that, we must move the $n-1$ remaining disks from the middle pole to the third pole (for a cost of at least $\mathbf{T}(n-1)$). Thus, no possible algorithm can solve the problem in less than $2^n - 1$ steps. Thus, our algorithm is optimal.[3]

Of course, there are variations to a given problem. Changes in the problem definition might or might not lead to changes in the lower bound. Two possible changes to the standard Towers of Hanoi problem are:

- Not all disks need to start on the first pole.
- Multiple disks can be moved at one time.

The first variation does not change the lower bound (at least not asymptotically). The second one does.

15.2 Lower Bounds on Searching Lists

In Section 7.9 we presented an important lower bounds proof to show that the problem of sorting is $\Theta(n \log n)$ in the worst case. In Chapter 9 we discussed a number of algorithms to search in sorted and unsorted lists, but we did not provide any lower bounds proofs to this important problem. We will extend our pool of techniques for lower bounds proofs in this section by studying lower bounds for searching unsorted and sorted lists.

15.2.1 Searching in Unsorted Lists

Given an (unsorted) list $\mathbf{L}$ of n elements and a search key K, we seek to identify one element in $\mathbf{L}$ which has key value K, if any exists. For the rest of this discussion, we will assume that the key values for the elements in $\mathbf{L}$ are unique, that the set of all possible keys is totally ordered (that is, the operations $<$, $=$, and $>$ are defined for all pairs of key values), and that comparison is our only way to find the relative ordering of two keys. Our goal is to solve the problem using the minimum number of comparisons.

Given this definition for searching, we can easily come up with the standard sequential search algorithm, and we can also see that the lower bound for this problem is "obviously" n comparisons. (Keep in mind that the key K might not actually appear in the list.) However, lower bounds proofs are a bit slippery, and it is instructive to see how they can go wrong.

Theorem 15.1 *The lower bound for the problem of searching in an unsorted list is n comparisons.*

[3] Recalling the advice to be suspicious of any lower bounds proof that argues a given behavior "must" happen, this proof should be raising red flags. However, in this particular case the problem is so constrained that there really is no (better) alternative to this particular sequence of events.

Here is our first attempt at proving the theorem.

Proof 1: We will try a proof by contradiction. Assume an algorithm A exists that requires only $n-1$ (or less) comparisons of K with elements of $\mathbf{L}$. Because there are n elements of $\mathbf{L}$, A must have avoided comparing K with `L[i]` for some value i. We can feed the algorithm an input with K in position i. Such an input is legal in our model, so the algorithm is incorrect. □

Is this proof correct? Unfortunately no. First of all, any given algorithm need not necessarily consistently skip any given position i in its $n-1$ searches. For example, it is not necessary that all algorithms search the list from left to right. It is not even necessary that all algorithms search the same $n-1$ positions first each time through the list.

We can try to dress up the proof as follows: **Proof 2:** On any given run of the algorithm, if $n-1$ elements are compared against K, then *some* element position (call it position i) gets skipped. It is possible that K is in position i at that time, and will not be found. Therefore, n comparisons are required. □

Unfortunately, there is another error that needs to be fixed. It is not true that all algorithms for solving the problem must work by comparing elements of $\mathbf{L}$ against K. An algorithm might make useful progress by comparing elements of $\mathbf{L}$ against each other. For example, if we compare two elements of $\mathbf{L}$, then compare the greater against K and find that this element is less than K, we know that the other element is also less than K. It seems intuitively obvious that such comparisons won't actually lead to a faster algorithm, but how do we know for sure? We somehow need to generalize the proof to account for this approach.

We will now present a useful abstraction for expressing the state of knowledge for the value relationships among a set of objects. A **total order** defines relationships within a collection of objects such that for every pair of objects, one is greater than the other. A **partially ordered set** or **poset** is a set on which only a partial order is defined. That is, there can be pairs of elements for which we cannot decide which is "greater". For our purpose here, the partial order is the state of our current knowledge about the objects, such that zero or more of the order relations between pairs of elements are known. We can represent this knowledge by drawing directed acyclic graphs (DAGs) showing the known relationships, as illustrated by Figure 15.1.

Proof 3: Initially, we know nothing about the relative order of the elements in $\mathbf{L}$, or their relationship to K. So initially, we can view the n elements in $\mathbf{L}$ as being in n separate partial orders. Any comparison between two elements in $\mathbf{L}$ can affect the structure of the partial orders. This is somewhat similar to the UNION/FIND algorithm implemented using parent pointer trees, described in Section 6.2.

Now, every comparison between elements in $\mathbf{L}$ can at best combine two of the partial orders together. Any comparison between K and an element, say A, in $\mathbf{L}$ can at best eliminate the partial order that contains A. Thus, if we spend m comparisons

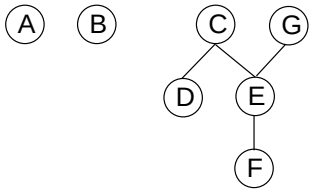

Figure 15.1 Illustration of using a poset to model our current knowledge of the relationships among a collection of objects. A directed acyclic graph (DAG) is used to draw the poset (assume all edges are directed downward). In this example, our knowledge is such that we don't know how *A* or *B* relate to any of the other objects. However, we know that both *C* and *G* are greater than *E* and *F*. Further, we know that *C* is greater than *D*, and that *E* is greater than *F*.

comparing elements in **L** we have at least $n - m$ partial orders. Every such partial order needs at least one comparison against *K* to make sure that *K* is not somewhere in that partial order. Thus, any algorithm must make at least n comparisons in the worst case. □

15.2.2 Searching in Sorted Lists

We will now assume that list **L** is sorted. In this case, is linear search still optimal? Clearly no, but why not? Because we have additional information to work with that we do not have when the list is unsorted. We know that the standard binary search algorithm has a worst case cost of $\mathrm{O}(\log n)$. Can we do better than this? We can prove that this is the best possible in the worst case with a proof similar to that used to show the lower bound on sorting.

Again we use the decision tree to model our algorithm. Unlike when searching an unsorted list, comparisons between elements of **L** tell us nothing new about their relative order, so we consider only comparisons between *K* and an element in **L**. At the root of the decision tree, our knowledge rules out no positions in **L**, so all are potential candidates. As we take branches in the decision tree based on the result of comparing *K* to an element in **L**, we gradually rule out potential candidates. Eventually we reach a leaf node in the tree representing the single position in **L** that can contain *K*. There must be at least $n + 1$ nodes in the tree because we have $n + 1$ distinct positions that *K* can be in (any position in **L**, plus not in **L** at all). Some path in the tree must be at least $\log n$ levels deep, and the deepest node in the tree represents the worst case for that algorithm. Thus, any algorithm on a sorted array requires at least $\Omega(\log n)$ comparisons in the worst case.

We can modify this proof to find the average cost lower bound. Again, we model algorithms using decision trees. Except now we are interested not in the depth of the deepest node (the worst case) and therefore the tree with the least-deepest node. Instead, we are interested in knowing what the minimum possible is

for the "average depth" of the leaf nodes. Define the **total path length** as the sum of the levels for each node. The cost of an outcome is the level of the corresponding node plus 1. The average cost of the algorithm is the average cost of the outcomes (total path length$/n$). What is the tree with the least average depth? This is equivalent to the tree that corresponds to binary search. Thus, binary search is optimal in the average case.

While binary search is indeed an optimal algorithm for a sorted list in the worst and average cases when searching a sorted array, there are a number of circumstances that might lead us to select another algorithm instead. One possibility is that we know something about the distribution of the data in the array. We saw in Section 9.1 that if each position in **L** is equally likely to hold X (equivalently, the data are well distributed along the full key range), then an interpolation search is $\Theta(\log \log n)$ in the average case. If the data are not sorted, then using binary search requires us to pay the cost of sorting the list in advance, which is only worthwhile if many (at least $\mathrm{O}(\log n)$) searches will be performed on the list. Binary search also requires that the list (even if sorted) be implemented using an array or some other structure that supports random access to all elements with equal cost. Finally, if we know all search requests in advance, we might prefer to sort the list by frequency and do linear search in extreme search distributions, as discussed in Section 9.2.

15.3 Finding the Maximum Value

How can we find the ith largest value in a sorted list? Obviously we just go to the ith position. But what if we have an unsorted list? Can we do better than to sort it? If we are looking for the minimum or maximum value, certainly we can do better than sorting the list. Is this true for the second biggest value? For the median value? In later sections we will examine those questions. For this section, we will continue our examination of lower bounds proofs by reconsidering the simple problem of finding the maximum value in an unsorted list.

Here is a simple algorithm for finding the largest value.

```
/** @return Position of largest value in  array A */
static int largest(int[] A) {
  int currlarge = 0; // Holds largest element position
  for (int i=1; i<A.length; i++)   // For each element
    if (A[currlarge] < A[i])      // if A[i] is larger
       currlarge = i;       //   remember its position
  return currlarge;         // Return largest position
}
```

Obviously this algorithm requires n comparisons. Is this optimal? It should be intuitively obvious that it is, but let us try to prove it. (Before reading further you might try writing down your own proof.)

Proof 1: The winner must compare against all other elements, so there must be $n-1$ comparisons. □

This proof is clearly wrong, because the winner does not need to explicitly compare against all other elements to be recognized. For example, a standard single-elimination playoff sports tournament requires only $n-1$ comparisons, and the winner does not play every opponent. So let's try again.

Proof 2: Only the winner does not lose. There are $n-1$ losers. A single comparison generates (at most) one (new) loser. Therefore, there must be $n-1$ comparisons. □

This proof is sound. However, it will be useful later to abstract this by introducing the concept of posets as we did in Section 15.2.1. We can view the maximum-finding problem as starting with a poset where there are no known relationships, so every member of the collection is in its own separate DAG of one element.

Proof 2a: To find the largest value, we start with a poset of n DAGs each with a single element, and we must build a poset having all elements in one DAG such that there is one maximum value (and by implication, $n-1$ losers). We wish to connect the elements of the poset into a single DAG with the minimum number of links. This requires at least $n-1$ links. A comparison provides at most one new link. Thus, a minimum of $n-1$ comparisons must be made. □

What is the average cost of **`largest`**? Because it always does the same number of comparisons, clearly it must cost $n-1$ comparisons. We can also consider the number of assignments that **`largest`** must do. Function **`largest`** might do an assignment on any iteration of the **`for`** loop.

Because this event does happen, or does not happen, if we are given no information about distribution we could guess that an assignment is made after each comparison with a probability of one half. But this is clearly wrong. In fact, **`largest`** does an assignment on the ith iteration if and only if **A**$[i]$ is the biggest of the the first i elements. Assuming all permutations are equally likely, the probability of this being true is $1/i$. Thus, the average number of assignments done is

$$1 + \sum_{i=2}^{n} \frac{1}{i} = \sum_{i=1}^{n} \frac{1}{i}$$

which is the Harmonic Series $\mathcal{H}_n$. $\mathcal{H}_n = \Theta(\log n)$. More exactly, $\mathcal{H}_n$ is close to $\log_e n$.

How "reliable" is this average? That is, how much will a given run of the program deviate from the mean cost? According to Čebyšev's Inequality, an observation will fall within two standard deviations of the mean at least 75% of the time. For **`Largest`**, the variance is

$$\mathcal{H}_n - \frac{\pi^2}{6} = \log_e n - \frac{\pi^2}{6}.$$

The standard deviation is thus about $\sqrt{\log_e n}$. So, 75% of the observations are between $\log_e n - 2\sqrt{\log_e n}$ and $\log_e n + 2\sqrt{\log_e n}$. Is this a narrow spread or a wide spread? Compared to the mean value, this spread is pretty wide, meaning that the number of assignments varies widely from run to run of the program.

15.4 Adversarial Lower Bounds Proofs

Our next problem will be finding the second largest in a collection of objects. Consider what happens in a standard single-elimination tournament. Even if we assume that the "best" team wins in every game, is the second best the one that loses in the finals? Not necessarily. We might expect that the second best must lose to the best, but they might meet at any time.

Let us go through our standard "algorithm for finding algorithms" by first proposing an algorithm, then a lower bound, and seeing if they match. Unlike our analysis for most problems, this time we are going to count the exact number of comparisons involved and attempt to minimize this count. A simple algorithm for finding the second largest is to first find the maximum (in $n-1$ comparisons), discard it, and then find the maximum of the remaining elements (in $n-2$ comparisons) for a total cost of $2n-3$ comparisons. Is this optimal? That seems doubtful, but let us now proceed to the step of attempting to prove a lower bound.

Theorem 15.2 *The lower bound for finding the second largest value is* $2n-3$.

Proof: Any element that loses to anything other than the maximum cannot be second. So, the only candidates for second place are those that lost to the maximum. Function **`largest`** might compare the maximum element to $n-1$ others. Thus, we might need $n-2$ additional comparisons to find the second largest. □

This proof is wrong. It exhibits the **necessity fallacy**: "Our algorithm does something, therefore all algorithms solving the problem must do the same."

This leaves us with our best lower bounds argument at the moment being that finding the second largest must cost at least as much as finding the largest, or $n-1$. Let us take another try at finding a better algorithm by adopting a strategy of divide and conquer. What if we break the list into halves, and run **`largest`** on each half? We can then compare the two winners (we have now used a total of $n-1$ comparisons), and remove the winner from its half. Another call to **`largest`** on the winner's half yields its second best. A final comparison against the winner of the other half gives us the true second place winner. The total cost is $\lceil 3n/2 \rceil - 2$. Is this optimal? What if we break the list into four pieces? The best would be $\lceil 5n/4 \rceil$. What if we break the list into eight pieces? Then the cost would be about $\lceil 9n/8 \rceil$. Notice that as we break the list into more parts, comparisons among the winners of the parts becomes a larger concern.

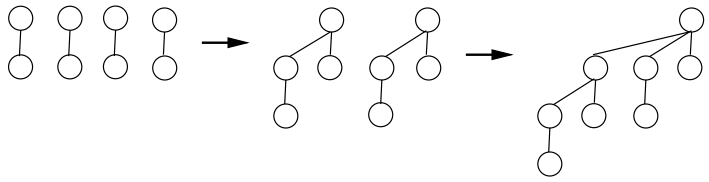

Figure 15.2 An example of building a binomial tree. Pairs of elements are combined by choosing one of the parents to be the root of the entire tree. Given two trees of size four, one of the roots is chosen to be the root for the combined tree of eight nodes.

Looking at this another way, the only candidates for second place are losers to the eventual winner, and our goal is to have as few of these as possible. So we need to keep track of the set of elements that have lost in direct comparison to the (eventual) winner. We also observe that we learn the most from a comparison when both competitors are known to be larger than the same number of other values. So we would like to arrange our comparisons to be against "equally strong" competitors. We can do all of this with a **binomial tree**. A binomial tree of height m has 2^m nodes. Either it is a single node (if $m = 0$), or else it is two height $m - 1$ binomial trees with one tree's root becoming a child of the other. Figure 15.2 illustrates how a binomial tree with eight nodes would be constructed.

The resulting algorithm is simple in principle: Build the binomial tree for all n elements, and then compare the $\lceil \log n \rceil$ children of the root to find second place. We could store the binomial tree as an explicit tree structure, and easily build it in time linear on the number of comparisons as each comparison requires one link be added. Because the shape of a binomial tree is heavily constrained, we can also store the binomial tree implicitly in an array, much as we do for a heap. Assume that two trees, each with 2^k nodes, are in the array. The first tree is in positions 1 to 2^k. The second tree is in positions $2^k + 1$ to 2^{k+1}. The root of each subtree is in the final array position for that subtree.

To join two trees, we simply compare the roots of the subtrees. If necessary, swap the subtrees so that tree with the the larger root element becomes the second subtree. This trades space (we only need space for the data values, no node pointers) for time (in the worst case, all of the data swapping might cost $\mathrm{O}(n \log n)$, though this does not affect the number of comparisons required). Note that for some applications, this is an important observation that the array's data swapping requires no comparisons. If a comparison is simply a check between two integers, then of course moving half the values within the array is too expensive. But if a comparison requires that a competition be held between two sports teams, then the cost of a little bit (or even a lot) of book keeping becomes irrelevent.

Because the binomial tree's root has $\log n$ children, and building the tree requires $n - 1$ comparisons, the number of comparisons required by this algorithm is $n + \lceil \log n \rceil - 2$. This is clearly better than our previous algorithm. Is it optimal?

We now go back to trying to improve the lower bounds proof. To do this, we introduce the concept of an **adversary**. The adversary's job is to make an algorithm's cost as high as possible. Imagine that the adversary keeps a list of all possible inputs. We view the algorithm as asking the adversary for information about the algorithm's input. The adversary may never lie, in that its answer must be consistent with the previous answers. But it is permitted to "rearrange" the input as it sees fit in order to drive the total cost for the algorithm as high as possible. In particular, when the algorithm asks a question, the adversary must answer in a way that is consistent with at least one remaining input. The adversary then crosses out all remaining inputs inconsistent with that answer. Keep in mind that there is not really an entity within the computer program that is the adversary, and we don't actually modify the program. The adversary operates merely as an analysis device, to help us reason about the program.

As an example of the adversary concept, consider the standard game of Hangman. Player *A* picks a word and tells player *B* how many letters the word has. Player *B* guesses various letters. If *B* guesses a letter in the word, then *A* will indicate which position(s) in the word have the letter. Player *B* is permitted to make only so many guesses of letters not in the word before losing.

In the Hangman game example, the adversary is imagined to hold a dictionary of words of some selected length. Each time the player guesses a letter, the adversary consults the dictionary and decides if more words will be eliminated by accepting the letter (and indicating which positions it holds) or saying that its not in the word. The adversary can make any decision it chooses, so long as at least one word in the dictionary is consistent with all of the decisions. In this way, the adversary can hope to make the player guess as many letters as possible.

Before explaining how the adversary plays a role in our lower bounds proof, first observe that at least $n-1$ values must lose at least once. This requires at least $n-1$ compares. In addition, at least $k-1$ values must lose to the second largest value. That is, k direct losers to the winner must be compared. There must be at least $n+k-2$ comparisons. The question is: How low can we make k?

Call the **strength** of element $\mathbf{A}[i]$ the number of elements that $\mathbf{A}[i]$ is (known to be) bigger than. If $\mathbf{A}[i]$ has strength a, and $\mathbf{A}[j]$ has strength b, then the winner has strength $a+b+1$. The algorithm gets to know the (current) strengths for each element, and it gets to pick which two elements are compared next. The adversary gets to decide who wins any given comparison. What strategy by the adversary would cause the algorithm to learn the least from any given comparison? It should minimize the rate at which any element improves it strength. It can do this by making the element with the greater strength win at every comparison. This is a "fair" use of an adversary in that it represents the results of providing a worst-case input for that given algorithm.

To minimize the effects of worst-case behavior, the algorithm's best strategy is to maximize the minimum improvement in strength by balancing the strengths of any two competitors. From the algorithm's point of view, the best outcome is that an element doubles in strength. This happens whenever $a = b$, where a and b are the strengths of the two elements being compared. All strengths begin at zero, so the winner must make at least k comparisons when $2^{k-1} < n \leq 2^k$. Thus, there must be at least $n + \lceil \log n \rceil - 2$ comparisons. So our algorithm is optimal.

15.5 State Space Lower Bounds Proofs

We now consider the problem of finding both the minimum and the maximum from an (unsorted) list of values. This might be useful if we want to know the range of a collection of values to be plotted, for the purpose of drawing the plot's scales. Of course we could find them independently in $2n - 2$ comparisons. A slight modification is to find the maximum in $n - 1$ comparisons, remove it from the list, and then find the minimum in $n - 2$ further comparisons for a total of $2n - 3$ comparisons. Can we do better than this?

Before continuing, think a moment about how this problem of finding the minimum and the maximum compares to the problem of the last section, that of finding the second biggest value (and by implication, the maximum). Which of these two problems do you think is harder? It is probably not at all obvious to you that one problem is harder or easier than the other. There is intuition that argues for either case. On the one hand intuition might argue that the process of finding the maximum should tell you something about the second biggest value, more than that process should tell you about the minimum value. On the other hand, any given comparison tells you something about which of two can be a candidate for maximum value, and which can be a candidate for minimum value, thus making progress in both directions.

We will start by considering a simple divide-and-conquer approach to finding the minimum and maximum. Split the list into two parts and find the minimum and maximum elements in each part. Then compare the two minimums and maximums to each other with a further two comparisons to get the final result. The algorithm is shown in Figure 15.3.

The cost of this algorithm can be modeled by the following recurrence.

$$\mathbf{T}(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ \mathbf{T}(\lfloor n/2 \rfloor) + \mathbf{T}(\lceil n/2 \rceil) + 2 & n > 2 \end{cases}$$

This is a rather interesting recurrence, and its solution ranges between $3n/2 - 2$ (when $n = 2^i$ or $n = 2^1 \pm 1$) and $5n/3 - 2$ (when $n = 3 \times 2^i$). We can infer from this behavior that how we divide the list affects the performance of the algorithm.

```
/** @return The minimum and maximum values in A
    between positions l and r */
static void MinMax(int A[], int l, int r, int Out[]) {
  if (l == r) {        // n=1
    Out[0] = A[r];
    Out[1] = A[r];
  }
  else if (l+1 == r) { // n=2
    Out[0] = Math.min(A[l], A[r]);
    Out[1] = Math.max(A[l], A[r]);
  }
  else {               // n>2
    int[] Out1 = new int[2];
    int[] Out2 = new int[2];
    int mid = (l + r)/2;
    MinMax(A, l, mid, Out1);
    MinMax(A, mid+1, r, Out2);
    Out[0] = Math.min(Out1[0], Out2[0]);
    Out[1] = Math.max(Out1[1], Out2[1]);
  }
}
```

Figure 15.3 Recursive algorithm for finding the minimum and maximum values in an array.

For example, what if we have six items in the list? If we break the list into two sublists of three elements, the cost would be 8. If we break the list into a sublist of size two and another of size four, then the cost would only be 7.

With divide and conquer, the best algorithm is the one that minimizes the work, not necessarily the one that balances the input sizes. One lesson to learn from this example is that it can be important to pay attention to what happens for small sizes of n, because any division of the list will eventually produce many small lists.

We can model all possible divide-and-conquer strategies for this problem with the following recurrence.

$$\mathbf{T}(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ \min_{1 \leq k \leq n-1}\{\mathbf{T}(k) + \mathbf{T}(n-k)\} + 2 & n > 2 \end{cases}$$

That is, we want to find a way to break up the list that will minimize the total work. If we examine various ways of breaking up small lists, we will eventually recognize that breaking the list into a sublist of size 2 and a sublist of size $n-2$ will always produce results as good as any other division. This strategy yields the following recurrence.

$$\mathbf{T}(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ \mathbf{T}(n-2) + 3 & n > 2 \end{cases}$$

This recurrence (and the corresponding algorithm) yields $\mathbf{T}(n) = \lceil 3n/2 \rceil - 2$ comparisons. Is this optimal? We now introduce yet another tool to our collection of lower bounds proof techniques: The state space proof.

We will model our algorithm by defining a **state** that the algorithm must be in at any given instant. We can then define the start state, the end state, and the transitions between states that any algorithm can support. From this, we will reason about the minimum number of states that the algorithm must go through to get from the start to the end, to reach a state space lower bound.

At any given instant, we can track the following four categories of elements:

- Untested: Elements that have not been tested.
- Winners: Elements that have won at least once, and never lost.
- Losers: Elements that have lost at least once, and never won.
- Middle: Elements that have both won and lost at least once.

We define the current state to be a vector of four values, (U, W, L, M) for untested, winners, losers, and middles, respectively. For a set of n elements, the initial state of the algorithm is $(n, 0, 0, 0)$ and the end state is $(0, 1, 1, n-2)$. Thus, every run for any algorithm must go from state $(n, 0, 0, 0)$ to state $(0, 1, 1, n-2)$. We also observe that once an element is identified to be a middle, it can then be ignored because it can neither be the minimum nor the maximum.

Given that there are four types of elements, there are 10 types of comparison. Comparing with a middle cannot be more efficient than other comparisons, so we should ignore those, leaving six comparisons of interest. We can enumerate the effects of each comparison type as follows. If we are in state (i, j, k, l) and we have a comparison, then the state changes are as follows.

$$
\begin{array}{lllll}
U:U & (i-2, & j+1, & k+1, & l) \\
W:W & (i, & j-1, & k, & l+1) \\
L:L & (i, & j, & k-1, & l+1) \\
L:U & (i-1, & j+1, & k, & l) \\
\quad or & (i-1, & j, & k, & l+1) \\
W:U & (i-1, & j, & k+1, & l) \\
\quad or & (i-1, & j, & k, & l+1) \\
W:L & (i, & j, & k, & l) \\
\quad or & (i, & j-1, & k-1, & l+2)
\end{array}
$$

Now, let us consider what an adversary will do for the various comparisons. The adversary will make sure that each comparison does the least possible amount of work in taking the algorithm toward the goal state. For example, comparing a winner to a loser is of no value because the worst case result is always to learn nothing new (the winner remains a winner and the loser remains a loser). Thus, only the following five transitions are of interest:

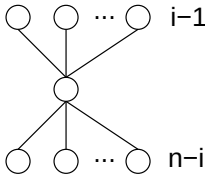

Figure 15.4 The poset that represents the minimum information necessary to determine the ith element in a list. We need to know which element has $i-1$ values less and $n-i$ values more, but we do not need to know the relationships among the elements with values less or greater than the ith element.

$U:U$	$(i-2,$	$j+1,$	$k+1,$	$l)$
$L:U$	$(i-1,$	$j+1,$	$k,$	$l)$
$W:U$	$(i-1,$	$j,$	$k+1,$	$l)$
$W:W$	$(i,$	$j-1,$	$k,$	$l+1)$
$L:L$	$(i,$	$j,$	$k-1,$	$l+1)$

Only the last two transition types increase the number of middles, so there must be $n-2$ of these. The number of untested elements must go to 0, and the first transition is the most efficient way to do this. Thus, $\lceil n/2 \rceil$ of these are required. Our conclusion is that the minimum possible number of transitions (comparisons) is $n + \lceil n/2 \rceil - 2$. Thus, our algorithm is optimal.

15.6 Finding the ith Best Element

We now tackle the problem of finding the ith best element in a list. As observed earlier, one solution is to sort the list and simply look in the ith position. However, this process provides considerably more information than we need to solve the problem. The minimum amount of information that we actually need to know can be visualized as shown in Figure 15.4. That is, all we need to know is the $i-1$ items less than our desired value, and the $n-i$ items greater. We do not care about the relative order within the upper and lower groups. So can we find the required information faster than by first sorting? Looking at the lower bound, can we tighten that beyond the trivial lower bound of n comparisons? We will focus on the specific question of finding the median element (i.e., the element with rank $n/2$), because the resulting algorithm can easily be modified to find the ith largest value for any i.

Looking at the Quicksort algorithm might give us some insight into solving the median problem. Recall that Quicksort works by selecting a pivot value, partitioning the array into those elements less than the pivot and those greater than the pivot, and moving the pivot to its proper location in the array. If the pivot is in position i, then we are done. If not, we can solve the subproblem recursively by only considering one of the sublists. That is, if the pivot ends up in position $k > i$, then we

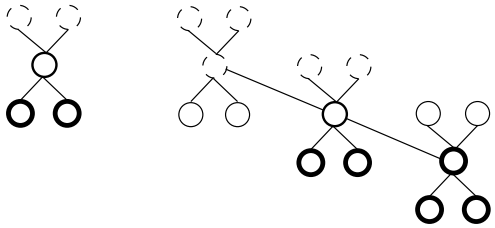

Figure 15.5 A method for finding a pivot for partitioning a list that guarantees at least a fixed fraction of the list will be in each partition. We divide the list into groups of five elements, and find the median for each group. We then recursively find the median of these $n/5$ medians. The median of five elements is guaranteed to have at least two in each partition. The median of three medians from a collection of 15 elements is guaranteed to have at least five elements in each partition.

simply solve by finding the ith best element in the left partition. If the pivot is at position $k < i$, then we wish to find the $i - k$th element in the right partition.

What is the worst case cost of this algorithm? As with Quicksort, we get bad performance if the pivot is the first or last element in the array. This would lead to possibly $\mathrm{O}(n^2)$ performance. However, if the pivot were to always cut the array in half, then our cost would be modeled by the recurrence $\mathbf{T}(n) = \mathbf{T}(n/2) + n = 2n$ or $\mathrm{O}(n)$ cost.

Finding the average cost requires us to use a recurrence with full history, similar to the one we used to model the cost of Quicksort. If we do this, we will find that $\mathbf{T}(n)$ is in $\mathrm{O}(n)$ in the average case.

Is it possible to modify our algorithm to get worst-case linear time? To do this, we need to pick a pivot that is guaranteed to discard a fixed fraction of the elements. We cannot just choose a pivot at random, because doing so will not meet this guarantee. The ideal situation would be if we could pick the median value for the pivot each time. But that is essentially the same problem that we are trying to solve to begin with.

Notice, however, that if we choose any constant c, and then if we pick the median from a sample of size n/c, then we can guarantee that we will discard at least $n/2c$ elements. Actually, we can do better than this by selecting small subsets of a constant size (so we can find the median of each in constant time), and then taking the median of these medians. Figure 15.5 illustrates this idea. This observation leads directly to the following algorithm.

- Choose the $n/5$ medians for groups of five elements from the list. Choosing the median of five items can be done in constant time.
- Recursively, select *M*, the median of the $n/5$ medians-of-fives.
- Partition the list into those elements larger and smaller than *M*.

While selecting the median in this way is guaranteed to eliminate a fraction of the elements (leaving at most $\lceil (7n-5)/10 \rceil$ elements left), we still need to be sure that our recursion yields a linear-time algorithm. We model the algorithm by the following recurrence.

$$\mathbf{T}(n) \leq \mathbf{T}(\lceil n/5 \rceil) + \mathbf{T}(\lceil (7n-5)/10 \rceil) + 6\lceil n/5 \rceil + n - 1.$$

The $\mathbf{T}(\lceil n/5 \rceil)$ term comes from computing the median of the medians-of-fives, the $6\lceil n/5 \rceil$ term comes from the cost to calculate the median-of-fives (exactly six comparisons for each group of five element), and the $\mathbf{T}(\lceil (7n-5)/10 \rceil)$ term comes from the recursive call of the remaining (up to) 70% of the elements that might be left.

We will prove that this recurrence is linear by assuming that it is true for some constant r, and then show that $\mathbf{T}(n) \leq rn$ for all n greater than some bound.

$$\begin{aligned}
\mathbf{T}(n) &\leq \mathbf{T}(\lceil \frac{n}{5} \rceil) + \mathbf{T}(\lceil \frac{7n-5}{10} \rceil) + 6\lceil \frac{n}{5} \rceil + n - 1 \\
&\leq r(\frac{n}{5} + 1) + r(\frac{7n-5}{10} + 1) + 6(\frac{n}{5} + 1) + n - 1 \\
&\leq (\frac{r}{5} + \frac{7r}{10} + \frac{11}{5})n + \frac{3r}{2} + 5 \\
&\leq \frac{9r+22}{10}n + \frac{3r+10}{2}.
\end{aligned}$$

This is true for $r \geq 23$ and $n \geq 380$. This provides a base case that allows us to use induction to prove that $\forall n \geq 380, \mathbf{T}(n) \leq 23n$.

In reality, this algorithm is not practical because its constant factor costs are so high. So much work is being done to guarantee linear time performance that it is more efficient on average to rely on chance to select the pivot, perhaps by picking it at random or picking the middle value out of the current subarray.

15.7 Optimal Sorting

We conclude this section with an effort to find the sorting algorithm with the absolute fewest possible comparisons. It might well be that the result will not be practical for a general-purpose sorting algorithm. But recall our analogy earlier to sports tournaments. In sports, a "comparison" between two teams or individuals means doing a competition between the two. This is fairly expensive (at least compared to some minor book keeping in a computer), and it might be worth trading a fair amount of book keeping to cut down on the number of games that need to be played. What if we want to figure out how to hold a tournament that will give us the exact ordering for all teams in the fewest number of total games? Of course, we are assuming that the results of each game will be "accurate" in that we assume

not only that the outcome of *A* playing *B* would always be the same (at least over the time period of the tournament), but that transitivity in the results also holds. In practice these are unrealistic assumptions, but such assumptions are implicitly part of many tournament organizations. Like most tournament organizers, we can simply accept these assumptions and come up with an algorithm for playing the games that gives us some rank ordering based on the results we obtain.

Recall Insertion Sort, where we put element i into a sorted sublist of the first $i-1$ elements. What if we modify the standard Insertion Sort algorithm to use binary search to locate where the ith element goes in the sorted sublist? This algorithm is called **binary insert sort**. As a general-purpose sorting algorithm, this is not practical because we then have to (on average) move about $i/2$ elements to make room for the newly inserted element in the sorted sublist. But if we count *only* comparisons, binary insert sort is pretty good. And we can use some ideas from binary insert sort to get closer to an algorithm that uses the absolute minimum number of comparisons needed to sort.

Consider what happens when we run binary insert sort on five elements. How many comparisons do we need to do? We can insert the second element with one comparison, the third with two comparisons, and the fourth with 2 comparisons. When we insert the fifth element into the sorted list of four elements, we need to do three comparisons in the worst case. Notice exactly what happens when we attempt to do this insertion. We compare the fifth element against the second. If the fifth is bigger, we have to compare it against the third, and if it is bigger we have to compare it against the fourth. In general, when is binary search most efficient? When we have $2^i - 1$ elements in the list. It is least efficient when we have 2^i elements in the list. So, we can do a bit better if we arrange our insertions to avoid inserting an element into a list of size 2^i if possible.

Figure 15.6 illustrates a different organization for the comparisons that we might do. First we compare the first and second element, and the third and fourth elements. The two winners are then compared, yielding a binomial tree. We can view this as a (sorted) chain of three elements, with element *A* hanging off from the root. If we then insert element *B* into the sorted chain of three elements, we will end up with one of the two posets shown on the right side of Figure 15.6, at a cost of 2 comparisons. We can then merge *A* into the chain, for a cost of two comparisons (because we already know that it is smaller then either one or two elements, we are actually merging it into a list of two or three elements). Thus, the total number of comparisons needed to sort the five elements is at most seven instead of eight.

If we have ten elements to sort, we can first make five pairs of elements (using five compares) and then sort the five winners using the algorithm just described (using seven more compares). Now all we need to do is to deal with the original losers. We can generalize this process for any number of elements as:

- Pair up all the nodes with $\lfloor \frac{n}{2} \rfloor$ comparisons.

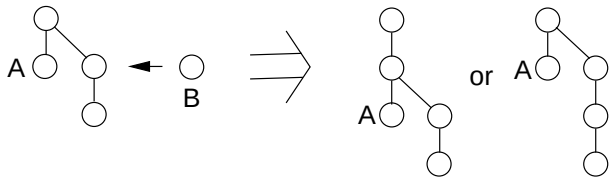

Figure 15.6 Organizing comparisons for sorting five elements. First we order two pairs of elements, and then compare the two winners to form a binomial tree of four elements. The original loser to the root is labeled *A*, and the remaining three elements form a sorted chain. We then insert element *B* into the sorted chain. Finally, we put *A* into the resulting chain to yield a final sorted list.

- Recursively sort the winners.
- Fold in the losers.

We use binary insert to place the losers. However, we are free to choose the best ordering for inserting, keeping in mind the fact that binary search has the same cost for 2^i through $2^{i+1} - 1$ items. For example, binary search requires three comparisons in the worst case for lists of size 4, 5, 6, or 7. So we pick the order of inserts to optimize the binary searches, which means picking an order that avoids growing a sublist size such that it crosses the boundary on list size to require an additional comparison. This sort is called **merge insert sort**, and also known as the Ford and Johnson sort.

For ten elements, given the poset shown in Figure 15.7 we fold in the last four elements (labeled 1 to 4) in the order Element 3, Element 4, Element 1, and finally Element 2. Element 3 will be inserted into a list of size three, costing two comparisons. Depending on where Element 3 then ends up in the list, Element 4 will now be inserted into a list of size 2 or 3, costing two comparisons in either case. Depending on where Elements 3 and 4 are in the list, Element 1 will now be inserted into a list of size 5, 6, or 7, all of which requires three comparisons to place in sort order. Finally, Element 2 will be inserted into a list of size 5, 6, or 7.

Merge insert sort is pretty good, but is it optimal? Recall from Section 7.9 that no sorting algorithm can be faster than $\Omega(n \log n)$. To be precise, the **information theoretic lower bound** for sorting can be proved to be $\lceil \log n! \rceil$. That is, we can prove a lower bound of exactly $\lceil \log n! \rceil$ comparisons. Merge insert sort gives us a number of comparisons equal to this information theoretic lower bound for all values up to $n = 12$. At $n = 12$, merge insert sort requires 30 comparisons while the information theoretic lower bound is only 29 comparisons. However, for such a small number of elements, it is possible to do an exhaustive study of every possible arrangement of comparisons. It turns out that there is in fact no possible arrangement of comparisons that makes the lower bound less than 30 comparisons when $n = 12$. Thus, the information theoretic lower bound is an underestimate in this case, because 30 really is the best that can be done.

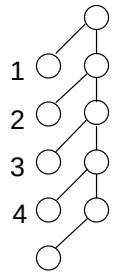

Figure 15.7 Merge insert sort for ten elements. First five pairs of elements are compared. The five winners are then sorted. This leaves the elements labeled 1-4 to be sorted into the chain made by the remaining six elements.

Call the optimal worst cost for n elements $S(n)$. We know that $S(n+1) \leq S(n) + \lceil \log(n+1) \rceil$ because we could sort n elements and use binary insert for the last one. For all n and m, $S(n+m) \leq S(n) + S(m) + M(m,n)$ where $M(m,n)$ is the best time to merge two sorted lists. For $n = 47$, it turns out that we can do better by splitting the list into pieces of size 5 and 42, and then merging. Thus, merge sort is not quite optimal. But it is extremely good, and nearly optimal for smallish numbers of elements.

15.8 Further Reading

Much of the material in this book is also covered in many other textbooks on data structures and algorithms. The biggest exception is that not many other textbooks cover lower bounds proofs in any significant detail, as is done in this chapter. Those that do focus on the same example problems (search and selection) because it tells such a tight and compelling story regarding related topics, while showing off the major techniques for lower bounds proofs. Two examples of such textbooks are "Computer Algorithms" by Baase and Van Gelder [BG00], and "Compared to What?" by Gregory J.E. Rawlins [Raw92]. "Fundamentals of Algorithmics" by Brassard and Bratley [BB96] also covers lower bounds proofs.

15.9 Exercises

15.1 Consider the so-called "algorithm for algorithms" in Section 15.1. Is this really an algorithm? Review the definition of an algorithm from Section 1.4. Which parts of the definition apply, and which do not? Is the "algorithm for algorithms" a heuristic for finding a good algorithm? Why or why not?

15.2 Single-elimination tournaments are notorious for their scheduling difficulties. Imagine that you are organizing a tournament for n basketball teams (you may assume that $n = 2^i$ for some integer i). We will further simplify

things by assuming that each game takes less than an hour, and that each team can be scheduled for a game every hour if necessary. (Note that everything said here about basketball courts is also true about processors in a parallel algorithm to solve the maximum-finding problem).

(a) How many basketball courts do we need to insure that every team can play whenever we want to minimize the total tournament time?
(b) How long will the tournament be in this case?
(c) What is the total number of "court-hours" available? How many total hours are courts being used? How many total court-hours are unused?
(d) Modify the algorithm in such a way as to reduce the total number of courts needed, by perhaps not letting every team play whenever possible. This will increase the total hours of the tournament, but try to keep the increase as low as possible. For your new algorithm, how long is the tournament, how many courts are needed, how many total court-hours are available, how many court-hours are used, and how many unused?

15.3 Explain why the cost of splitting a list of six into two lists of three to find the minimum and maximum elements requires eight comparisons, while splitting the list into a list of two and a list of four costs only seven comparisons.

15.4 Write out a table showing the number of comparisons required to find the minimum and maximum for all divisions for all values of $n \leq 13$.

15.5 Present an adversary argument as a lower bounds proof to show that $n - 1$ comparisons are necessary to find the maximum of n values in the worst case.

15.6 Present an adversary argument as a lower bounds proof to show that n comparisons are necessary in the worst case when searching for an element with value X (if one exists) from among n elements.

15.7 Section 15.6 claims that by picking a pivot that always discards at least a fixed fraction c of the remaining array, the resulting algorithm will be linear. Explain why this is true. Hint: The Master Theorem (Theorem 14.1) might help you.

15.8 Show that any comparison-based algorithm for finding the median must use at least $n - 1$ comparisons.

15.9 Show that any comparison-based algorithm for finding the second-smallest of n values can be extended to find the smallest value also, without requiring any more comparisons to be performed.

15.10 Show that any comparison-based algorithm for sorting can be modified to remove all duplicates without requiring any more comparisons to be performed.

15.11 Show that any comparison-based algorithm for removing duplicates from a list of values must use $\Omega(n \log n)$ comparisons.

15.12 Given a list of n elements, an element of the list is a *majority* if it appears more than $n/2$ times.

(a) Assume that the input is a list of integers. Design an algorithm that is linear in the number of integer-integer comparisons in the worst case that will find and report the majority if one exists, and report that there is no majority if no such integer exists in the list.

(b) Assume that the input is a list of elements that have no relative ordering, such as colors or fruit. So all that you can do when you compare two elements is ask if they are the same or not. Design an algorithm that is linear in the number of element-element comparisons in the worst case that will find a majority if one exists, and report that there is no majority if no such element exists in the list.

15.13 Given an undirected graph G, the problem is to determine whether or not G is connected. Use an adversary argument to prove that it is necessary to look at all $(n^2 - n)/2$ potential edges in the worst case.

15.14 **(a)** Write an equation that describes the average cost for finding the median.

(b) Solve your equation from part (a).

15.15 **(a)** Write an equation that describes the average cost for finding the ith-smallest value in an array. This will be a function of both n and i, $\mathbf{T}(n, i)$.

(b) Solve your equation from part (a).

15.16 Suppose that you have n objects that have identical weight, except for one that is a bit heavier than the others. You have a balance scale. You can place objects on each side of the scale and see which collection is heavier. Your goal is to find the heavier object, with the minimum number of weighings. Find and prove matching upper and lower bounds for this problem.

15.17 Imagine that you are organizing a basketball tournament for 10 teams. You know that the merge insert sort will give you a full ranking of the 10 teams with the minimum number of games played. Assume that each game can be played in less than an hour, and that any team can play as many games in a row as necessary. Show a schedule for this tournament that also attempts to minimize the number of total hours for the tournament and the number of courts used. If you have to make a tradeoff between the two, then attempt to minimize the total number of hours that basketball courts are idle.

15.18 Write the complete algorithm for the merge insert sort sketched out in Section 15.7.

15.19 Here is a suggestion for what might be a truly optimal sorting algorithm. Pick the best set of comparisons for input lists of size 2. Then pick the best set of comparisons for size 3, size 4, size 5, and so on. Combine them together into one program with a big case statement. Is this an algorithm?

15.10 Projects

15.1 Implement the median-finding algorithm of Section 15.6. Then, modify this algorithm to allow finding the ith element for any value $i < n$.