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 algorithm 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 truely understand our problem in a theoretical sense. It also saves us the effort of attempting to discover more 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 was the subject of Chapter 17. However, this approach does not help us when we cannot find a suitable "similar problem." We focus on this chapter with discovering and proving lower bounds from first principles. A significant example of a lower bounds argument 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 a 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 search for an 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*.[1] This can be difficult bar, given that we cannot possibly know all algorithms for any problem, since 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. Since 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.

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. The problem comes when 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? However, 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 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 fallicy 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 any 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 recurrance $\mathbf{T}(n) = 2\mathbf{T}(n-1)+1 = 2^n - 1$. 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

[2] I give credit to Gregory J.E. Rawlins for presenting this formulation in his book "Compared to What?"

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 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 example 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 affect 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 imporant 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 exist. 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 all defined), 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.

[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 alternative to this particular sequence of events.