Lower bounds for Comparison Based Algorithms

Professor: David Kirkpatrick

Scribe: Sarah Huber

October 7th, 2013

1. Introduction

Upper bounds for growth rates of algorithms are likely the largest concern in algorithm development and analysis- and rightly so. Knowing how complex a problem may become is an essential part of solving that problem. However, the lower bound of a method's growth rate is a subject that is commonly overlooked. Often, this is because finding a good lower bound can be very difficult. Looking into the subject of finding good lower bounds for algorithms gives rise to some beautiful and effective proof methods, such as adversary arguments. Our lecture will focus on finding lower bounds for the deterministic selection problem and other related comparison based problems while illustrating some general tactics for finding lower bounds. By comparison of our lower bounds with the running time of the randomized version of the selection problem, we will illustrate the effectiveness of randomization. We will also introduce direct access tables as a higher model of computation.

2. Methods for Finding K^{th} Element in a List

2a. A Review of Randomized Selection

Previous classes have discussed the selection problem; how to find the k^{th} element in a list of size n. A randomized algorithm for this problem was illustrated, and this algorithm was proven to have a running time of $T(n)=2n+o(n)$. However, there exist other randomized algorithms with a running times of down to $1.5n+o(n)$.

This running time is quite good, and one must ask what makes randomization so effective. Several explanations exist. By selecting random elements, and discarding others, we reduce the size of the original problem. By using recursion, we focus on a problem that is distinctly smaller than the original problem. Effective algorithms, such as the ones discussed previously, exploit the properties of sampling- we choose our sample size large enough such that the probability of a poor choice of sample is low, but small enough that our problem is significantly reduced.

2b. A Deterministic Algorithm for Median Finding

Let us restrict our focus to the sub problem of finding the median element in our list. First, we will establish a general scheme for this problem. A good deterministic algorithm will "estimate" where the median is, find the median of random groups of elements, and partition the problem accordingly.

A more specific algorithm is:

“Median of Medians”

1. Break elements of S into groups of five
2. Find the median of each group, call it x[i]
3. Using a recursive call to the algorithm, find the “median of medians”- M
4. Use M to partition the elements of S, and call the algorithm again on the partitioned problem

An initial group size of five has proven to be the most efficient, but is not required for a functional algorithm.

How long does this algorithm, or another deterministic algorithm for this problem, take to run? We search for a lower bound for the running time. An initial lower bound for a problem can be given by the input or output size of the problem. For example, to find the median of a list of size n, we must observe all n elements in order to know we have obtained the median. However, to find a tighter lower bound, we must use a more sophisticated argument. We will now discuss some these arguments in a general fashion.

3. General Lower Bound Strategies and Examples

3a. Comparison of Lower Bounds

In comparing algorithms lower bounds, we must first establish our limitations in making these comparisons. A given algorithms might have higher level operations, or a more advanced model of computation, that make it difficult to compare with a more simplistic algorithm. In this situation, a comparison of lower bounds is less meaningful because the former algorithm has a “head start” of sorts. We can observe that in order to meaningfully compare algorithms, we must choose a model of computation to fit our problem, and restrict our comparisons to algorithms of that model. In the selection problem, we could consider a model of computation that allowed us to perform averaging to find the median. However, that is less meaningful than comparing algorithms following only comparison based models of computation. It is to this comparison based model we will restrict our focus. Now we illustrate some examples of lower bound finding in simple problems

3b. Example: Top at Tennis

We consider the simpler problem of finding the best tennis player in a tournament. We wish to find out how many games need to be played in order to find this player. We can observe that at least (n-1) comparisons, or games, must be made or played. There are several different explanations for this. The most intuitive might be that in a tournament, every player must be either a “non-winner” or a “winner.” To find the winner, we must find all the “non-winners”, of which there are n-1. We can also explain this in terms of the output graph of comparisons. In order to have discovered our winner, we must have a connected output graph, which means that we need at least n-1 edges, or games.

3c. Example: Dictionary problem

Another good example for finding a lower bound on is the dictionary problem. Consider a dictionary with n keys. We want to know whether a given query, q, belongs in our dictionary. Our lower bound is given by finding how many yes/no questions, or comparisons, must be made to determine this. Since every questions halves our uncertainty, it's obvious that by using binary search, we can solve our problem in with a minimum lg(n) comparisons, or questions.

However, it's not clear that this is a good lower bound. An adversary argument, as discussed in the following section, allows us to determine a tighter lower bound.

3d. Adversary Arguments

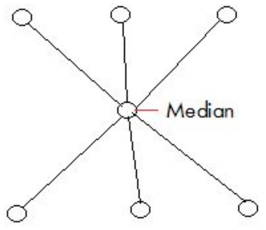
A useful tactic for determining lower bounds is that of adversary arguments. A comparison based algorithm will proceed through a series of questions or comparisons. We give an "adversary" control over the answer to these questions, with his objective being to make the algorithm make as many comparisons as possible, while the algorithm attempts to find the solution in as few steps as possible. The adversary cannot violate any of the previous statements it has made. It's important to realize that the adversary makes no assumptions about the algorithm- it cannot choose the answer to questions based on future questions the algorithm might ask.

We will now attempt to find a non-trivial lower bound for the (median) element finding problem

4. A Lower Bound for Median Finding via Connectivity

Now that we have established some general strategies for finding an algorithms lower bound, we return to the median finding problem, and attempt to find a non-trivial lower bound.

Let's choose n=2k+1 for some integer k. We find our median via comparisons, as demonstrated in the image below. Note that not all comparisons are necessarily direct- via transitivity, we can construct an ordering of n.



Our problem is now reduced to the problem of constructing partial orders, and using this to find the median.

It's easy to observe that we must have n-1 comparisons. It may be possible to show that more comparisons are required. We can observe that a comparison that take place between an element above and an element below the median is "wasted," as it does not add any meaning to our partial order. The total number of comparisons will then be the sum of the n-1 required comparisons and all wasted comparisons.

<u>4a. Adversary Argument</u>

Let us perform our algorithm on a set A_k for k={1:n}. Designate the median as A_m.

We define a crucial comparison for A_i, $i \neq m$, to be one of:

1. $A_i<A_m$ or $A_i<A_j$ for $A_j<A_m$ (A_i less than median)
2. $A_i>A_m$ or $A_i>A_j$ for $A_j>A_m$ (A_i greater than median)

These crucial comparisons are the n-1 comparisons required to find the median, since they determine that A_i cannot be the median. We can compare this to finding all our "non-winners" in a tennis tournament.

To increase the total number of comparisons, the adversary will attempt to maximize the number of non-crucial or wasted comparisons.

Let us place elements can be in either "top," "middle" or "bottom," where all elements start in "middle." When an element is placed in the "top" or "bottom," it cannot be the median. Thus, our algorithm will terminate when n-1 elements have been placed in the top or bottom. The adversary's strategy for answering "is $A_i<A_j$?" is as follows, where the adversary updates the partial order after each step.

<u>Adversary Strategy</u>

(a) If A_i and A_j are in the middle then respond "yes." Then A_i is moved to the bottom and A_j is moved to the top.
(b) If A_i is in the middle and A_j is in the top then reply "yes"
(c) If A_i is in the middle and A_j is in the bottom then reply "no"
(d) In all remaining cases (both A_i and A_j in the top or bottom) answer consistently with the existing partial order.

We observe that (a) never results in a crucial comparison, since it is the first comparison for each element- we cannot know how A_i, A_j compare to the median. We can also observe that n-1 crucial comparisons are required to find the median.

By answering as we did in (b) and (c), we keep as many elements in the middle as possible. Elements can only leave the middle via (a). Since we have an initial 2k+1 elements in the middle, of which 2k must leave in the middle in pairs, there must be k cases of (a), where n=2k+1. So we have (n-1)/2 "wasted" comparisons and n-1 crucial comparisons are produced via this argument. This gives us a lower bound of ~1.5n comparisons.

A more sophisticated adversary can give us a better lower bound. For example, it might include more levels than the three described above, and a more complicated algorithm for moving between levels. These further arguments can give us increasingly tighter lower bounds of Ω=1.75n and Ω=2n.

4b. Comparison with Randomized Algorithm

We now recall that our randomized algorithm had a worst-case run time of ~1.5n. This is an example of how randomization can be more effective than a deterministic method.

5. Direct Access Tables with Random Access Machines

5a. Direct Access Tables

A direct access table (DAT) is the most elementary form of hashing. We assume that each possible object, or key, maps to a positive integer [0,..., m], where m is the size of our table. We can access any key by knowing its location in table, so a search is fast. However, if we want to include many potential keys (if m is large), then our table is very expensive. This is especially true if we only want to hold a few keys out of a large set of potential keys. We also require some method of mapping keys to a unique integer in [0,..., m].

5b. Dictionary Problem with RAM

We established previously a lower bound of Ω=lg(n) as our lower bound for querying a dictionary S, on a general comparison-based model. Now we consider more sophisticated models of computation.

We consider a direct access table over the universe U that contains S. Using a random access machine (RAM), we can access any element of the DAT. However, U could be very large, and our DAT would require space proportional to |U|. We must consider how to reduce this initialization cost, as well as the cost of predecessor and successor queries.

6. Conclusion

We have discussed general methods for finding tight lower bounds for comparison based problems. Our methodology is particularly focused on adversary arguments. We have found lower bounds for several different problems, including the deterministic median selection problem. Our lower bound for this problem is tight enough that it reveals the superiority of the randomized algorithm. We also considered the dictionary problem, both in a comparison based model, and in a model using a random access machine. In the latter case, we can use a direct access table. Future classes will discuss the effective use and limitations of these tables, and their extension to hashing.

References

[1] http://www.ics.uci.edu/~eppstein/161/960130.html
[2] http://www.cs.wustl.edu/~sg/CS441_FL01/adv-lb.pdf
[3] http://faculty.simpson.edu/lydia.sinapova/www/cmsc250/LN250_Weiss/L09-Hashing09.pdf
[4] http://www.comp.nus.edu.sg/~ooiwt/tp/cs1102-0203-s1/lecture/10-hash.pdf
[5] http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/notes/27-lowerbounds.pdf