Lecture 2 (08-31) - Computational Models and Lower Bounds

Computational Models

Differing computational models produce different efficient algorithms based on what operations are charged (and likewise which are free) and certain limitations of the algorithm (disallowed operations). For example, the following are examples of some of the models we may use:

- **Word RAM model** (Von Neuman Machines) - Generally the closest approximation to how modern computers operate. We have some memory with M total bits divided into words of order $lg(M)$. We can then perform unit-time operations on words and such.
- **Transdichotomous model** - Similar to the Word RAM model, but word size scales with problem size to some n^k degree. In this model, it has been shown that search can be completed in $o(\log n)$ time and sorting can be done in $o(n \log n)$ time.
- **Comparison model** - Comparison operations cost time while other operations do not. Data does not have inherent meaning besides some natural ordering that can be determined through comparisons.
- **Exchange Model** - Exchanging elements in memory costs time while other operations do not. Algorithms that operate in this case are efficient based on a "fewest moves" principle.
- **Query Model** - Accessing information is costly and other operations are free. Generally, we want to determine an outcome by accessing the lowest units of information.

Upper and Lower Bounds

We say the **upper bound** is $f(n)$ when $\exists\ \mathcal{A}$ that outputs a correct solution to every input in $f(n)$ time. The faster $f(n)$, the tighter the bound.

Likewise, the **lower bound** is $g(n)$ when $\forall\ \mathcal{A}$, the problem takes at least $g(n)$ time. The slower $g(n)$, the tighter the bound.

> Note that by definition, both of these operate off of the worst-case runtime.

The optimal algorithm runtime $p(n)$ must fall between $g(n) \leq p(n) \leq f(n)$.
Showing a lower bound is essentially showing necessity while an upper bound shows sufficiency. Showing both for a tight bound shows necessary and sufficient runtime.

To show a valid upper bound, it is generally sufficient to provide an algorithm $\mathcal{A}$ that runs in $f(n)$ time and prove its efficiency.

To show a lower bound, it can be a bit more difficult as we need to demonstrate the bound for arbitrary algorithms. However, there are two general ideas we can try:

1. **Decision Tree Argument** (good for asymptotic bounds) - We show that given the M permutations of the input, we must require a decision tree of height $lg(M)$ in order to obtain enough information to provide the correct output.
2. **By Contradiction** - This can be further be split into two ideas:
   - **Adversary Argument** (better for tighter bounds) - We assume that we can solve the problem in at least $g(n) - 1$ steps. We then provide a deterministic approach to generating an input that contradicts this.
   - **Prior Result Argument** - We take some prior known result and show that if the algorithm is possible in $o(g(n))$ time, then the prior result is contradicted. Useful for related algorithms (i.e. search vs. sort in comparison model).

Lower bound for comparison-based sorting

It's commonly-known that comparison-based sorting of some array A cannot be faster by $O(lg\, n)$ time; how do we prove it?

By Decision Tree Argument

Consider the number of inputs for an array A of n elements. There would be $n!$ different permutations of the elements of the array assuming a strictly increasing comparison order is possible.

Then, suppose our algorithm $\mathcal{A}$ is able to output the exchanges necessary to return A in sorted order for each permutation as the leaf of the decision tree, and we can assume that there exists a permutation in the tree where only a single leaf node contains the answer.

- For the latter point, we can argue that the algorithm returns the exchanges in some defined order. Then in order to find which permutation the current input is, we must take traverse $lg(M)$ layers of the decision tree as we can eliminate at most half of the remaining permutations as the result of a comparison.

Thus, we need $lg(n!)$ bits of information no matter the algorithm.
Simplifying this to asymptotic form, we get:

$$lg(n!) = lg(n) + lg(n-1) + lg(n-2) \cdots + lg(1) \leq O(nlg(n))$$

by taking the largest term.

We also get

$$lg(n!) = lg(n) + lg(n-1) + lg(n-2) \cdots + lg\left(\frac{n}{2}\right) \cdots + lg(1) \geq \frac{n}{2} lg\left(\frac{n}{2}\right) = \Omega(nlg(n))$$

by taking the $\frac{n}{2}$ largest terms as a lower bound.
We can also use Sterling's Formula but it is unnecessary for this summation.

Exchange-based sorting

For the exchange model, recall that only exchanges are charged in unit time. Comparisons and other operations are free.

Upper Bound

For the upper bound, we can show that $n-1$ is sufficient to sort all elements. Consider an algorithm that knows the sorted order of the array A. Then, we simply swap the ordered element 1 into index 1, element 2 into index 2, element 3 in index 3, as long as the element is not already in its specified index. When element $n-1$ is swapped, we get element n for free and the array is sorted.
This algorithm takes $n-1$ exchanges.

Lower Bound

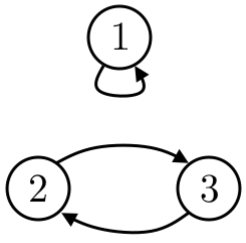
We can get a naïve lower bound of $\frac{n}{2}$ exchanges pretty easily by stating that there are such cases where all elements may not be in their respective indices and therefore, we must exchange each element at least once. As we exchange the places of 2 elements per operation, we take $\frac{n}{2}$ operations.

However, this bound isn't exactly tight and we can indeed do better:

Consider a directed graph with the indices of A. Each index has a single edge that points to the index of its comparison order. This directed graph has the property of being a permutation of a set of cycles.

For example, an element in its sorted position would look like the following:

Likewise, the array $A = [1, 3, 2]$ would be represented as follows:



The argument then is that how sorted an array can be encoded in how many cycles are in the representative graph. An array of size n will have n cycles in its sorted position.

Then, we we perform an exchange of two elements (swapping the indices while keeping the pointers), we notice the following:

1. If the elements were of distinct cycles, then the number of cycles decreases by 1.
2. If the elements were of the same cycle, then the number of cycles increases by 1.

From these points, it is clear that if we have a worst-case scenario of a graph that begins with only 1 cycle. Then if we were to increase the number of cycles to n, we need at least $n - 1$ exchanges.

Comparison-based maximum

Suppose we have an array A of comparable elements and we wish to find the maximum element of A. How many comparisons does this take?

Upper Bound

One way we can calculate the maximum is by initially setting a max equal to $A[0]$. Then, starting from 2 to n, compare each element to the max. Then, if $max > A[i]$. leave the max as-is. If not, $max \leftarrow A[i]$.

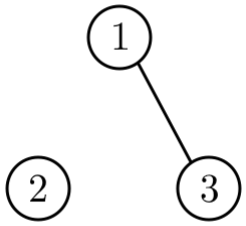
We then return the max at the end.
This will take $n-1$ total comparisons where we compare each element once besides the first element.

Lower Bound

Once again, we can obtain a naïve lower bound by saying we must look at each element once. At 2 "look-ups" per comparison, we obtain a lower bound of $\frac{n}{2}$ comparisons. This, however is once again not sufficiently tight.

For this problem, let us take the adversary argument:
Suppose we could find the maximum in $n-2$ comparisons.

Let us construct a graph where the vertices are the indices of the array and edges represent elements that have been compared. For example, suppose we compare $A[1], A[3]$ in a 1-indexed array of 3 elements. This would be represented by:



Now, for simplicity, we will assume A is an array of positive integers (if there exist negative integers, we add $|min(A)|$ to each element). Suppose now if 1 or 3 was the maximum. We can add $max(A)$ to the element in index 2 as a new input and the algorithm would still output 1 or 3.

Likewise, if 2 was the maximum, we can add $A[2]$ to either index 1 or 3 and the algorithm would still output 2 based off the comparisons. Therefore, the lower bound $> n-2$, making $n-1$ a tight bound.

Upper Bound for Comparison-based 2^{nd} maximum

Consider how we would get the 2^{nd} largest element in the array.
One way we could solve this is by running the algorithm for the maximum twice like so:

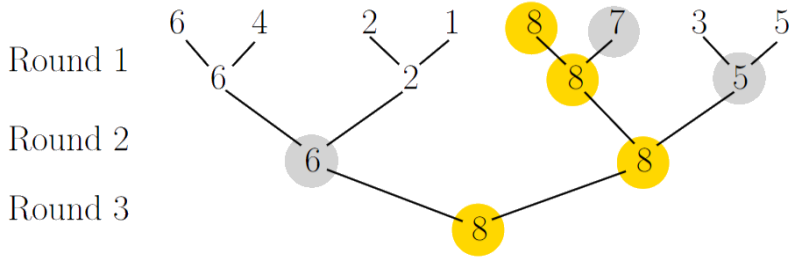
1. Find the maximum of the array in $n-1$ comparisons.
2. Remove the maximum from the array.
3. Find and return the maximum of the new array in $n-2$ comparisons.

This gives a new upper bound of $n-1+n-2=2n-3$ comparisons.

Once again, we can do better.

Consider the fact that the 2^{nd} largest must have been greater than every element except for the maximum, which means it must have been compared to the maximum at some point. If we organize the way we compare our elements cleverly, we can take advantage of this and find the maximum of the elements that have been compared to the true max.

Consider the following "tournament-like" structure:



Notice that we only have one candidate for the 2^{nd} max every round. Given the binary tree of $lg(n)$ height, we can therefore determine the 2^{nd} largest element in $n-1+\lceil lg(n) \rceil$.

#decision-trees #algo #comp-models #comparison #exchange #bounds #sorting #maximum