CSE 347 Lecture 7

Lower bounds in limited models

1. Lower bounds
2. Comparison lower bound for sorting
3. Radix Sort

Divide-and-conquer algorithms allowed us to solve several problems faster than a natural algorithm that seemed like it might be optimal. This raises the question of when we should stop searching for a more clever algorithm. Could it be that we can always find a better algorithm if we keep looking? A very subtle question... such problems exist, but you are unlikely to encounter them! What we would like to know is, how much time is necessary to solve a problem. This is a "lower bound" on the running time of any algorithm for the problem; if you have an algorithm that takes only this much time, then your algorithm is optimal.

We'll use the following notation, which is a counterpart to big-O: we say $f(n) = \Omega(g(n))$ if there is a positive constant C such that for all sufficiently large n, $f(n) \geq C \cdot g(n)$.

For many natural problems $g(n)=n$ is a lower bound — for example, I claim that computing $x_1+x_2+\cdots+x_n$ requires n steps. Why? If we do not read some x_i, then the algorithm's output does not depend on x_i. So, for $y \neq z$, when $x_i=y$ or $x_i=z$, as long as the rest of the inputs are the same, the output is the same. But, $(x_1+x_2+\cdots+y+\cdots+x_n)-(x_1+\cdots+z+\cdots x_n)$

$$= y-z \neq 0$$

$$\Rightarrow x_1+x_2+\cdots+y+\cdots+x_n \neq x_1+\cdots+z+\cdots+x_n$$

So in at least one of the cases, the output is wrong.

So in this case, the natural $O(n)$-time algorithm is optimal. Similarly, computing the matrix-vector product of an $n \times n$ matrix and an n-dimensional vector requires n^2 steps — if the vector is $(1,1,\cdots,1)$, then we must read every entry in the matrix, by the above argument. (Since the matrix has size n^2, this is again an input-size lower bound.) Note that we don't get that "because Matrix Multiplication computes n matrix-vector products, it takes $\Omega(n^3)$ steps" — this is false, since Strassen's algorithm runs in $O(n^{2.81})$ steps.

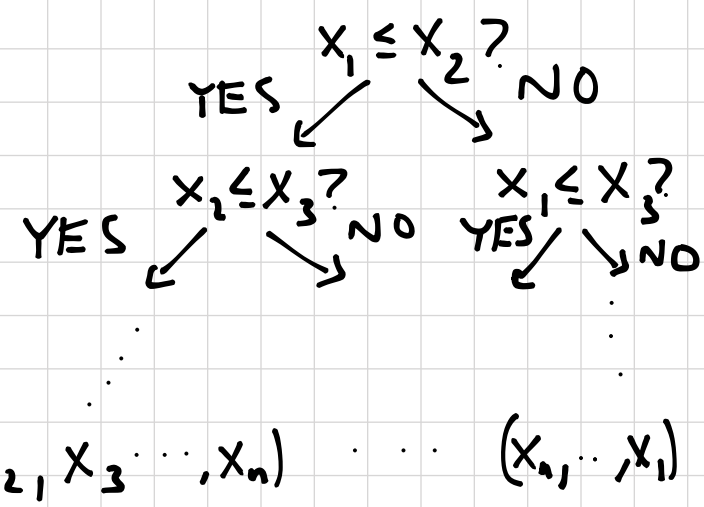
It is very hard to establish lower bounds that are larger than the input/output sizes.

Comparison lower bound for sorting

It becomes easier to establish lower bounds if we

restrict what our algorithm can do. For example, let's suppose we want to write a sorting algorithm that must treat the input as a list of objects that we can compare, but not examine otherwise. We can model the algorithm as a rooted tree of nodes labeled by pairs of elements (x_i, x_j) corresponding to a query "$x_i \leq x_j$?" The node has one branch labeled YES and another labeled NO, meaning that if $x_i \leq x_j$, the algorithm's next comparison (or output) is the label of the node on the YES branch. Otherwise, it is the label of the node on the NO branch. Again, we stress that the algorithm's next step is entirely determined by the result of the comparisons it has queried. The leaves of the tree are labeled by some reordering of the input which, if the algorithm is correct, must be a permutation π s.t. $x_{\pi(i)} \leq x_{\pi(i+1)}$ for all i.

Ex:



The start of the tree for Bubble sort.

The length of a branch traversed on some input is then the number of comparisons the algorithm makes on that input. The algorithm runs for at least that many steps.

Theorem A sorting algorithm in the comparison model

Must make $\Omega(n \log n)$ queries in the worst case.

Merge Sort runs in time $O(n \log n)$, so it is an optimal algorithm in this model.

Proof: Consider the set of inputs corresponding to all permutations of $\{1, 2, \ldots, n\}$. There are $n!$ such permutations. Consider the tree corresponding to any sorting algorithm. We'll choose a path in the tree as follows: starting with S_0 as the set of all $n!$ inputs, given a set of "current" inputs S_i, we examine how many of these inputs would take the algorithm on the YES branch, and how many would take the NO branch. Whichever is larger, we follow that branch, and put S_{i+1} equal to those inputs. Notice, then $|S_{i+1}| \geq \frac{1}{2}|S_i|$. Now, as long as $|S_i| > 1$, the algorithm cannot stop, (why?) different input permutations are sorted by different permutations — simply consider what happens to any ith element that is in different positions. Therefore, this path has length ℓ s.t. $2^\ell \geq n!$

i.e., $\ell \geq \log_2 n!$ Since $n! \geq \underbrace{\frac{n}{2} \cdots \frac{n}{2}}_{n/2}$, $\ell \geq \frac{n}{2} \log \frac{n}{2} = \Omega(n \log n)$ ☒

For example, if $n=3$, $S_0 = \{(123), (132), (213), (231), (312), (321)\}$ and if the algorithm first compares $x_1 \nless x_2$, half have $x_1 \leq x_2$ and half have $x_1 > x_2$. Arbitrarily, we might take the first branch, so $S_1 = \{(123), (132), (231)\}$. If the algorithm next queries $x_2 \leq x_3$, then only (123) satisfies $x_2 \leq x_3$, so we take the other branch, with $S_2 = \{(132), (213), (231)\}$. The next comparison determines

the order (and our final input that takes this long path).

Radix Sort

So, is Merge Sort optimal? Unsatisfyingly, we don't really know. But, if we know more about our input, we may be able to do better—for example, if our input consists of strings.

Let's first consider the simple case where our input consists of a single character. We then have a simple algorithm, "Bucket Sort":

Input: a list of characters from an alphabet of size r

```
Initialize r empty lists.
Scan the input list:
  | Insert the ith character into the corresponding list
Concatenate the r lists in lexicographic (sorted) order
```

The algorithm runs in $O(n+r)$ steps, and — it's easily verified — outputs a sorted list with the same number of copies of each character as the input. It's therefore the sorted list. If $r = O(n)$, this is $O(n)$ time beating our lower bound.

We can extend this idea to short strings: for strings of length L over an alphabet of size r, we call the following algorithm, "Radix Sort."

Input: list of strings of length L over an alphabet of size r $(x_1, \ldots, x_n)$

```
For i = L, L-1, ..., 2, 1
    Create r empty lists
    For j = 1, ..., n
        Append x_j to the end of the list corresponding
          to the ith character.
    (x_1, ..., x_n) ← the r lists concatenated in order
                      (of the ith character)
```

By inspection, we see that the algorithm runs in time $O(L(n+r)) = O(Ln)$ if $r = O(n)$.

Theorem: At the end of the ith iteration, if we drop the initial $L-i$ characters of each element, the list of these final i characters is in sorted order

Proof by induction on i. Base: $i=0$ is trivial.

Induction Step: Observe that we add characters to the r lists in the inner loop in the same order that they originally appeared. Therefore, on iteration $i+1$, each of these r lists is, by IH, in sorted order on the final i characters. Now, when we concatenate the r buckets, observe that this list is in sorted order on the final $i+1$ characters — all of the strings with a smaller character in position $L-(i+1)$ appear before those with a larger character, and within these lists, we had that the strings were in lexicographic order on the final i characters. ▨

So, at the end of the algorithm, the list is sorted.

If L is small compared to $\log_2 n$, this may be much better than MergeSort. This does not tell us how to beat MergeSort in general, but it does illustrate what a sorting algorithm that does not fit in the comparison model could look like. The comparison lower bound thus only tells us that a particular strategy for sorting requires $\Omega(n \log n)$ steps, which might not be the only kind of algorithm.