CS 151
Sorting!!
Announcements

- prelab 3 is due now
- no class on Wednesday (it’s Yom Kippur)
Recall the sorting problem, from last week:
  input: a list \( L \) of \( n \) integers
  goal: return \( L \) sorted in increasing (or, non-decreasing) order

Sorting is a very important problem, because it needs to be done a LOT.
  - often input needs to be sorted before an algorithm can run on it
  - if you can keep your data sorted, searching it becomes easier
mergeSort( list L ) // returns L sorted in non-decreasing order
    // base case:
    If the list L has 0 or 1 element, return it as sorted

    // divide:
    Otherwise, let split L down the middle into two halves Q & R
    recursively sort Q by calling mergeSort( Q )
    recursively sort R by calling mergeSort( R )

    // conquer: how do we produce sorted L out of sorted Q, R?
    reset L to be empty
    while( both Q and R have elements left )
        compare the front (smallest) elements of Q and R
        remove the smallest of the 2 from its list and add it to L
        (why is this element the next-smallest element of L?)
    add the rest of the remaining list to the end of L
    return L, the sorted list

We saw last week that the running time for Mergesort is $O(n \log n)$
Insertion Sort

idea: keep a growing “sorted list so far” in front portion of array
Initially, the sorted portion contains only the first element (it’s sorted)
On each iteration, take first element of unsorted portion, and
insert it into the sorted portion at the correct position
Selection Sort

idea: keep the $p$ smallest elements at front, in order
Start with $p=0$ (the smallest element may not be at the front)
On each iteration, find the next smallest element (i.e. increment $p$),
and move it to the end of the sorted portion
Theorem: every (comparison-based) algorithm for sorting is at least $n \log n$.

Proof: Sorting can be thought of as determining one permutation of its input. (The values themselves don’t matter, just what their relative order is.) Since the input can be in any order, the number of possible inputs is the number of permutations on $n$ elements, of which there are $n!$.

We model the decisions of the algorithm with a “binary tree”, i.e. a branching diagram that encodes the information we have so far. At the very “root” of our tree, we have all possible permutations, because we haven’t made any comparisons and have no information as to which permutation is the real sorted one.

After each comparison of some element $a$ to some element $b$, we either know we are in a permutation where $a$ is before $b$ (because $a < b$), or $b$ is before $a$ (because $b \leq a$).

That is, each comparison allows us to “branch” into one of two groups, each of half the size. We are narrowing down our search space by a half. The algorithm ends when our group is of size 1.
**Theorem:** every (comparison-based) algorithm for sorting is at least $n \log n$.

**Proof:** So, we start with $n!$ possible orderings / permutations. With each comparison, we divide the number of possibilities in two. The algorithm ends when we are down to one possibility. The number of comparisons is thus the number of times we can divide $n!$ by two before we get down to 1. I.e. we need at least $\log(n!)$ comparisons. Using Stirling’s approximation (don’t worry ‘bout it), we know that $\log(n!)$ is $n \log n$. 