**Sorting Lecture:**

Examples:

Constant time O(1): Accessing an element of an array

O(log(n)): halving things over and over again. (binary search trees).

Linear time: O(n). P dece.

Log-Linear time: not terrible. Annoying for big enough probs, but not bad. (a tree for each element).

O(n^2): terrible

O(2^n): terribler

O(n!): terriblest. Traveling salesman problem is actually this. Can’t brute-force it. Need to approximate for n>10ish.

Check out:

www.bigocheatsheet.com

Big O can be time- or space-complexity.

When we say linked-list insertion is constant time, there’s a caveat! We must already know where it’s being inserted (have a pointer on that node), so that all that’s being done is switching up where the pointers point!

**Bubble Sort:**

Start with first pair. Are these in order? Nope? Swap. Move over frame of reference.

After the first pass, the end of the list is sorted because the largest value has bubbled to the end, so on the subsequent passes, we can go one step less. *O(1) in terms of space-complexity.* Once we go through the list and make no swaps, we know the list is sorted. However, this is still O(n^2) even though the list gets 1 unit shorter every iteration. Haven’t changed the overall shape.

**Merge Sort:**

Significantly better performance than Bubble Sort. We do “divide and conquer”, usually through recursion. Take the array and split in halves. Then we split those into halves… until these are all singletons or empty arrays. NOW, merge pairs of sorted arrays. Check first items. As soon as one array is empty, push the rest of the second on. Now do with neighboring arrays. Time is O(n\*log(n) for time, and O(n) for space.