**Comparison-Based Sorts**

* Sort items in a list based on comparing things to each other
* In contrast to another type of sort that we’ll see: counting sort (Thursday’s lab)

**Insertion Sort**

* Maintain a sorted portion and unsorted portion of the list (in the beginning the sorted portion will be nothing, and the unsorted portion will be the entire list)
* For the first item i of the unsorted portion, “insert” it into the correct position in the sorted list
  + Do this by comparing i to the number that comes before it (on the first iteration, this will be the last number of the sorted portion), if i is smaller than this value, swap the two
  + Continue the above step until i is greater than the number that comes before it or you have reached the front of the list
* Runtime: will be based on the number of swaps that we have to do!
  + Best case: Theta(N), everything is in sorted order, we will compare each element of the unsorted portion with the number that comes before it and no additional swaps will be necessary
  + Worst case: Theta(N^2), reverse sorted order and will have to do the maximum number of swaps every time

**Selection Sort**

* Maintain a sorted portion and unsorted portion
* Scan through the entire unsorted portion at every iteration, “select”-ing the minimum value and placing it in the sorted portion
* Runtime:
  + Best/worst case: Theta(N^2), each iteration will require us to look through the entire list

**Merge Sort**

* Divide and conquer algorithm, recursive
* Split the list in half and recursively call mergeSort on each half (now the halves are sorted)
* For the sorted halves, merge them together
  + Can do this by maintaining a pointer to the first elements of each half, see which one is less, and then add that to the sorted portion, moving the pointers up as necessary
* Runtime:
  + We will split the list in half every time, resulting in logN splits
  + We will do a linear amount of work on each level merging the lists together
  + This will total in Theta(NlogN) amount of work done in the best and worst case!

**Quicksort**

* We will pick a pivot value and then for each of the elements in the list, put them in a list representing the lesser elements and the greater elements
* Call quicksort on each of the partitions
* Runtime:
  + Depends on how well the pivot works out for us!
  + If the pivot splits everything roughly in half, then we have something similar to mergesort, Theta(NlogN)
  + What if the pivot always partitions to a list with 0 elements and the other list containing all the other elements? Theta(N^2) (looks kind of like selection sort!)