**Average-Case Runtimes**

* For some sorts’ best and worst case running times, they need to sort a very particular type of array
  + Ex. insertion sort on an already sorted array, quicksort and always picking the minimum item as the pivot
* Need to find how long sorts run on average!
* We did something similar to this for hashing, we tried to find the average based on multiple calls to a particular method, for sorting we will focus on random lists and expectations
* Insertion sort: need to remove all inversions
  + Inversion: a pair of elements that are in the in correct order
  + Maximum number of inversions is N(N+1)/2, every single pair is inverted
    - Reverse sorted list
  + In the best case, there are no inversions to make
    - Sorted list
  + On average, we will have (0 + (N(N+1))/2)/2 = N(N+1)/4 inversions, average case is still N^2
* Quicksort: randomize the pivot choice
  + In the worst case, pivot would only put one element in the right place (reduce the list to sort by one item), split the list into 0% and 100%
  + In the best case, pivot splits the list in two equal halves, 50% and 50%
  + On average, we will split it 25% and 75%, we will have log(4/3)N levels and each level does around N work, NlogN still average case

**Comparison-Based Sorts**

* In the worst case, all comparison-based sorts will have to do NlogN comparisons, meaning the worst case of a comparison-based sort is NlogN
* Can we do better?

**Counting Sort**

* Let’s say we had an array full of 3 values: “fish”, “cat”, “dog”
* If we were to mergesort/quicksort this, we will have to compare all values together, even “fish” and “fish”
* Instead keep an array called counts that keeps track of the count of each value (one index for “fish”, one for “cat”, one for “dog”) and go through the entire array and collect the counts
* Will know where each item will belong in the array based on this counts array! Let’s call this the starts array and this will tell us where to start for each element
* Go through all the elements of the array and place the element where it should belong based on the starts array
* Runtime:
  + N is the number of items in our list, K is the radix
  + Iterate through the list: N
  + Iterate through the counts: K
  + Iterate through the list again: N
  + N + N + K, which is roughly N + K

**Stability**

* Relative ordering of duplicate items are preserved!

**Radix Sort**

* Sorts items based on their radix and run counting sort on each radix!
* Radix: how many elements a particular value can take on, with the numbers that we have today, radix is 10 (0->9), for the alphabet, radix is 26 (a->z)
* LSD Radix sort:
  + Start with the least significant digit (rightmost element) and go towards the most significant digit (leftmost element)
  + Stable sort, iterative, easier to implement
  + Runtime:
    - Will have to go through all the digits to sort it correctly
    - (counting sort runtime) \* (length of longest value sorted)
* MSD Radix sort:
  + Start with the most significant digit (leftmost element) and group all the elements with the same digit in the same bucket, recurse on each bucket looking at the next value
  + Not stable sort if we try to do it in place and stack calls can get large if we recurse too far, bad caching (61C), usually not used in practice
  + Runtime:
    - Not necessary to go through all the digits, what if we sort based on the MSD and all of the values have different MSD?
    - Best case: (counting sort runtime)
      * We don’t go through all the digits, let’s say all of our values start with a different letter, then they all go to different buckets and now we can concatenate the buckets to get the sorted list!
    - Worst case: (counting sort runtime) \* (length of longest value sorted)
      * Must go through all the letters (like LSD radix sort)

**Visualizations (RECOMMENDED TO VIEW)**

* visualgo.net