**Programming Assignment 1 Report**

Matthew Starr and Jed Pulley

**1. How do the three sorting algorithms fare against each other?**

Although all three sorting algorithms try, and eventually do, produce a sorted array, each one is different in complexity, and therefore the time it takes to complete as well.On the latest run of TestTime.java, where length is 43,252,049, the algorithms ranked from fastest to slowest are:

1. RadixSort: 7,235.0 milliseconds
2. MergeSort: 12,731.0 milliseconds
3. QuickSort (Median of 3): 17,988.0 milliseconds
4. QuickSort (Randomized): 18,340.0 milliseconds

The results produced here were consistent in the complexity derived from each method. In general, Quicksort has a complexity of Θ(n2), Mergesort has a complexity of Θ(nlog(n)), and Radix sort is an impressive complexity of Θ(n). In terms of sorting, one can not get faster than Θ(n) time, so an algorithm in Θ(n) time is likely to perform better than ones with other complexities, which in this case, is always worse.

**2. How does the randomized selection algorithm fare against a radix sort based selection? Why do you think that although both are linear time algorithms, the latter turns out to be much slower in practice?**

The randomized selection algorithm and radix sort methods are both efficient use of linear sorting, the randomized selection algorithm finds an arbitrary number for its pivot. These types of algorithms have a complexity of Θ(n). For radix sort, this is the best case, and in a scenario with large numbers, the complexity only increases to Θ((number of digits)\*n). Randomized quicksort can be quite effective, though the reality of it choosing a random number as a pivot guarantees that some random numbers will be chosen that are not ideal, such as those close to the maximum or minimum values, where nearly all values would go one one side of the pivot. If this happens too frequently, the algorithm becomes not efficient in the method used. This ensures that the worst case of the Randomized quicksort is Θ(n2).The median of three approach aims to fix this by choosing a value towards the middle, yet this is still only an estimate, and in the worst-case is still equal to the randomized sorting algorithm. Radix sorting moves from right to left, checking each value and sorting based on the single digit only. By ordering each digit sequentially, each number will be sorted once you get to the end of the left-most value.

**3. How does brute-force inversion counting fare against the merge-sort approach?**

Right off the bat, BruteForce Inversion Counting takes a beating when compared to MergeSort Inversion Counting. And it only gets worse as time goes on. On the latest run of TestTime.java, the average times that I got for both algorithms are:

1. BruteForce Average Time: 2,894.62 milliseconds
2. MergeSort Average Time: 13.85 milliseconds

It’s not even a contest with how much more efficient MergeSort is. Running TestTime.java is by far the longest I’ve had to wait for any single program to finish since I’ve been coding, and seeing the numbers scroll by, it’s no shocker as to why. The merge sorting algorithm is a standard sorting algorithm that completes in Θ(nlog(n)) time to complete. Although this is not a merge sort, the lookup process is identical to one that is, and therefore, the time complexity is the same. This approach is much faster than a brute-force search that looks at every possibility in a nested for loop. By checking each value in this manner, the process becomes slow, coming in at Θ(n2) time. For small arrays where only a few possibilities exist, the time requirement is not significant, yet at longer length arrays, the difference is unmistakable. For an extended run of the algorithm, doing a brute-force search took 2,894.62 milliseconds compared to merge sort which found the same answer in 13.85 milliseconds.