Winning Algorithms and Analysis Report

### Description

The experiment tested 6 sorting algorithms with 4 list types of length 10,000, taking the average of 50 trials for each sort-list pair. The values in the list range from 0 to the length. I used nanoseconds as the unit for the time for greater precision. A challenge I faced initially was a Stack Overflow error caused by insufficient memory allocated for my IDE, which I fixed with the help of the project description’s student Q/A.

The hardest part of this project involved counting the comparisons and movements for each sorting algorithm. The code itself is simple, but a deep understanding of each algorithm was required to correctly place increment counters. I did not count checks for loop bounds as comparisons because if I did, then Radix sort would be considered a comparison sort, which it is not. For movements, I only counted list value manipulations and not index or counter manipulation and other similar operations because the focus of movements in this project is on values in the list. The overhead cost of extra memory manipulation is included in the total execution time.

Another challenge I faced was my computer running each test varyingly for a given set of trials. The first trial would take some time, then the next would take less, and so on until the time of the last trial was a fraction of the time of the first. To fix this, I took the average of a set number of trials for each test for reliable results.

Finally, I implemented the following features:

* Option to specify how sorted an almost-order list is through a ratio.
* Percent differences in time between the winning sort and other sorts.
* A GUI in Java with the ability to toggle whether specific sorting algorithms are tested.

# Sorting In-Order Lists

### Wining algorithm: Insertion sort (5.0 ms)

### Analysis

Insertion sort did very well, having the fewest comparisons of the comparison sorts. It did well here because it never had to swap within its sorted sublist, meaning it never entered its second loop within the first loop. With no other complex loops or operations, it had a linear time complexity.

Merge sort (123.2%) never had to re-arrange elements when merging, but still had to check each element and manage the extra storage it uses.

Radix sort (142.7%) using count sort has the same number of buckets and elements within each bucket for each list type in this experiment. There was minimal error in time performance for other data types (+/- 4.5 ms), so I will omit Radix sort from the rest of the analyses.

Heap sort (186.4%) with a min heap had to swap k-1 times down the tree to add an element k, with removal from the heap and copying to the array taking an additional O(N) time.

Selection sort (197.2%) searches for a new minimum in the rest of the array n times, performing equally terrible in each data type. I will be omitting selection sort for the rest of the analyses similarly because of the same reason I am omitting Radix sort, only because it consistently performs the worst in the other data types besides in-order lists.

Quick sort (197.9%) achieved its worst case time complexity because every pivot was the first/least element in the partition in this implementation.

# Sorting Reverse-Order Lists

### Wining algorithm: Merge sort (24.7 ms)

### Analysis

Merge sort and Radix sort have a tie here (only 6.7% difference). Merge sort did the best because of the same reason it was second in in-order lists, only now insertion sort did poorly.

Heap sort (121.9%) had its best case scenario where it never had to swap when adding to the heap.

Quick sort (189.6%) performed poorly because of a similar reason in in-order lists – its pivot was always the first/greatest in the partition.

Insertion sort (192.8%) had to swap all the way to the beginning for each new element in the sublist.

# Sorting Almost-Order Lists (80% sorted)

### Wining algorithm: Radix sort (21.5 ms)

### Analysis

Quick sort (58.3%) was at the almost optimal case where most of the pivots are the middle value in the partition and the partition is mostly sorted.

Merge sort (52.5%) was at its average case but did a bit worse than in-order and reverse-order lists because it had to re-arrange some elements while merging.

Heap sort (142.1%) struggled because most of the list was already sorted, meaning swapping to the end for every new element.

Insertion sort (187.6%) did badly because although the list was mostly sorted, some unsorted elements had to be swapped through the majority of the sublist.

# Sorting Random-Order Lists

### Wining algorithm: Radix sort (21.3 ms)

### Analysis

Every sort performed according to their average case here: O(N^2) for insertion and selection sort, O(nlogn) for quick, merge, and heap sort, and O(N) for radix sort.