**Performance Test Report**

**Michael Placzek and Greg Scheidler**

**November 14, 2021**

In this assignment, we created rankings of multiple sorting algorithms. They all did what they were supposed to do, sort Comparable items in an array based on what its compareTo() method would output. If it were a 1 or a 0, the item would be sorted after the previous one. If it were less than 0, it would be sorted before the previous item.

Our group's testing involved Bubble Sort, Heap Sort, Insertion Sort, Merge Sort, Selection Sort, Shell Sort, and Quick Sort. We have deduced that Quick Sort was the winner in general. However, there may be certain situations where a different sorting algorithm would perform more efficiently. For example, performing an Insertion Sort is the most efficient in case the Comparable target array is already sorted. Because, an Insertion Sort would not change the items in the target array if it’s sorted. Therefore its execution time would be less than 1 millisecond.

Devices that had slower RAM usually had a slower recursive execution. Also, devices with a "beefy" CPU could process an iterative algorithm effortlessly. Therefore, we have concluded that picking the most efficient algorithm could rely on multiple factors. We have found that the size of the sorting and device specifications all matter. For instance, a recursive algorithm may perform well on one machine and suffer on another.

Our group initially tried to perform testing on Gradescope. Therefore, we have concluded that one machine operating as the server could perform timing tests on our sorting algorithms. However, our group noticed one key detail. No one knew the background processes of that machine. For example, if background processes were running, the system would have to share its resources with different programs. Therefore, the code execution would be slower. A theoretical situation is; no significant background processes could be running simultaneously while running someone’s code, but some processes could be running simultaneously while running someone else's code. That was a big problem that added variables for execution time. A key recommendation for testers in the future would be that the tester would have to ensure that no significant background processes are running while the code executes. Our group has concluded that eliminating background processes majorly helps any system run smoother, and therefore fewer variables are present during the code execution.