COMPSCI 2XC3 Lab Report 1

Prepared by

Group 64

Luca Mawyin

Anderson Ray

Theo Pham

COMPSCI 2ME3

McMaster University

February 1, 2026

Experiment 1

In our experiment comparing Bubble Sort, Insertion Sort, and Selection Sort. We ran 100 tests for each sorting algorithm going from a list length of 0, to a list length of 1000 each list length being 10 elements longer then the last.

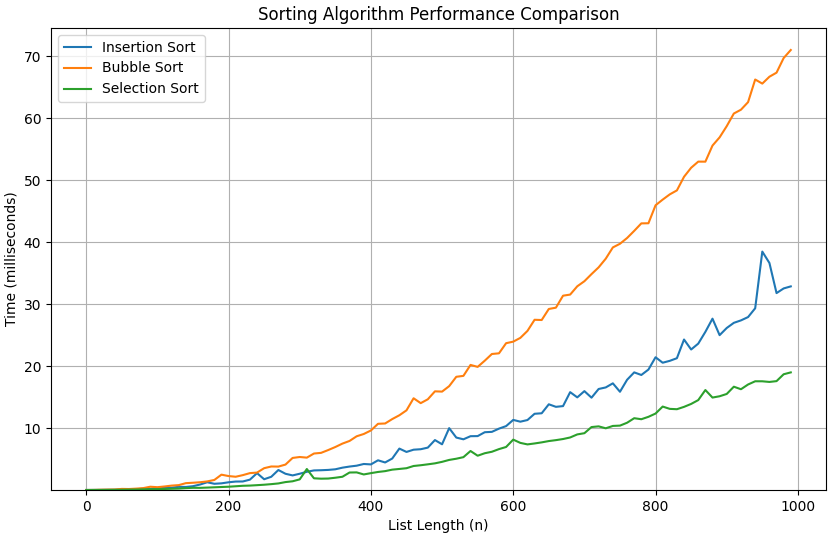

Figure 1: List Length vs. Time (ms) for Bubble, Selection, and Insertion Sort

Looking at the slopes, each algorithm looks to have a parabolic shape which makes sense as we know that the algorithms used are $\mathcal{O}(n^2)$.

Bubble sort is the slowest as the inner loop, loops through the entire list every iteration of the outer loop. Selection sort is faster then Insertion sort as swaps elements in the list smarter using less memory reads and writes making it faster.

Experiment 2

In this experiment, we compared the original Bubble Sort and Selection Sort with their optimized variations. The Bubble Sort variation reduces the number of swaps by shifting elements, while the Selection Sort variation places both the minimum and maximum elements in each iteration.

The experiment was performed on randomly generated lists with lengths ranging from 0 to 1000, increasing by 10 each time. All lists contained integers between 0 and 100000, and runtimes were measured in milliseconds.

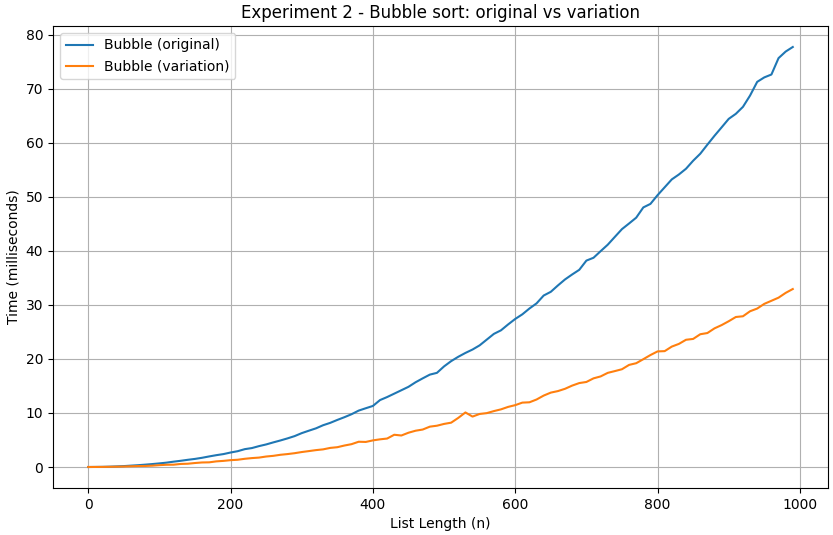

Figure 2: List Length vs. Time (ms) for original Bubble Sort and its optimized variation

Figure 2 shows that the optimized Bubble Sort consistently outperforms the original version as the list size increases. This improvement is due to the reduced number of swap operations, which lowers the cost of memory accesses. Although both implementations exhibit quadratic growth, the optimization significantly improves performance in practice.

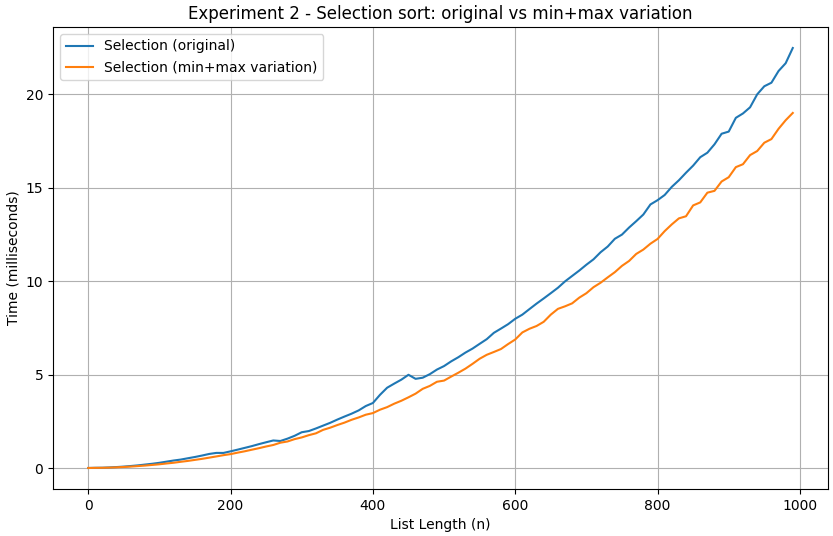

Figure 3: List Length vs. Time (ms) for original Selection Sort and the min+max variation

As shown in Figure 3, the min max variation of Selection Sort provides a modest but consistent speedup over the original implementation. By placing two elements per iteration, the number of outer loop passes is reduced. However, the overall time complexity remains quadratic, limiting the performance improvement.

In summary, both optimizations reduce constant overhead rather than asymptotic complexity. The Bubble Sort variation yields a more noticeable improvement, while the Selection Sort variation offers a smaller but stable performance gain.

Experiment 3

In our experiment comparing Bubble Sort, Insertion Sort, and Selection Sort on sorted lists with varying numbers of swaps made. We ran 109 tests for each sorting algorithm on lists of size 2000 with the number of swaps ranging from 0 - 10965. Each test would increase the number of swaps made by 100.

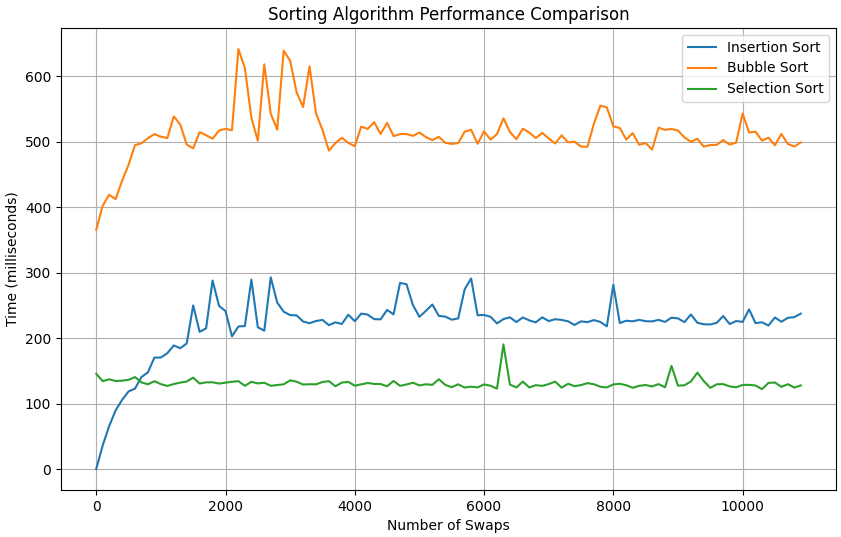

Figure 4: List Length vs. Number of Swaps on a Sorted List for Bubble, Selection, and Insertion Sort.

Looking at the graph, Bubble Sort and Insertion Sort preform better with less swaps. Bubble Sort preforms better with less swaps, as its doing much less memory reads and writes in the innerloop. Insertion swap preforms better with less swaps as it exits out the second loop if it runs into sorted pairs of elements which are more common with less swaps.

Selection Sort doesn't perform better with less swaps as the number of checks and swaps are independent of whether the list is sorted or not. Even if the innerloop can't find a min index past L[i] it still swaps L[i] with itself.

Experiment 4

For this experiment, we compared the runtime of Quick Sort, Merge Sort, and Heap Sort. There were 100 tests run for each algorithm, with increasing list sizes from 0 to 1000, incrementing the list length by ten each iteration.

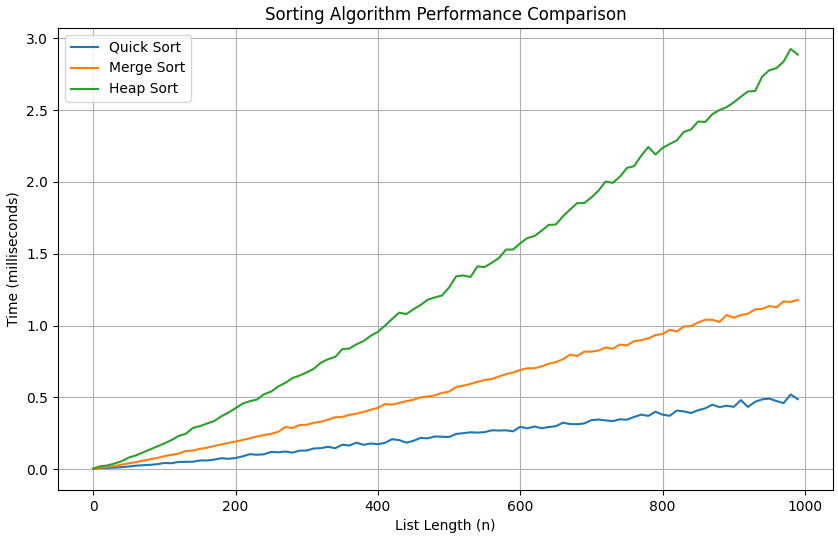

Figure 5: List Length vs. Time (ms) for Quick, Merge, and Heap Sort

Observing the resulting runtimes of each algorithm, they appear to be near-linear with a slight curve. This is an accurate depiction as each of the sorting algorithms represented have a runtime of $\mathcal{O}(n \log n)$

Heap Sort is the slowest sorting algorithm due to the constant overhead required in heap maintenance. Merge Sort is slower than Quick Sort due to the increased overhead of copying elements, along with having to zip sorted partitions of lists together, as opposed to centering the sort around a pivot and copying elements less.

Experiment 5

In this experiment, Quick Sort, Merge Sort, and Heap Sort were compared using near-sorted lists of length 1000. There were 100 tests run for each algorithm, with an increasing number of swaps on an already-sorted list. The number of swaps ranged from 0 to 500, with an incrementation of five. Furthermore, each increment was executed five times, with the runtime averaged out in order to guarantee a higher accuracy.

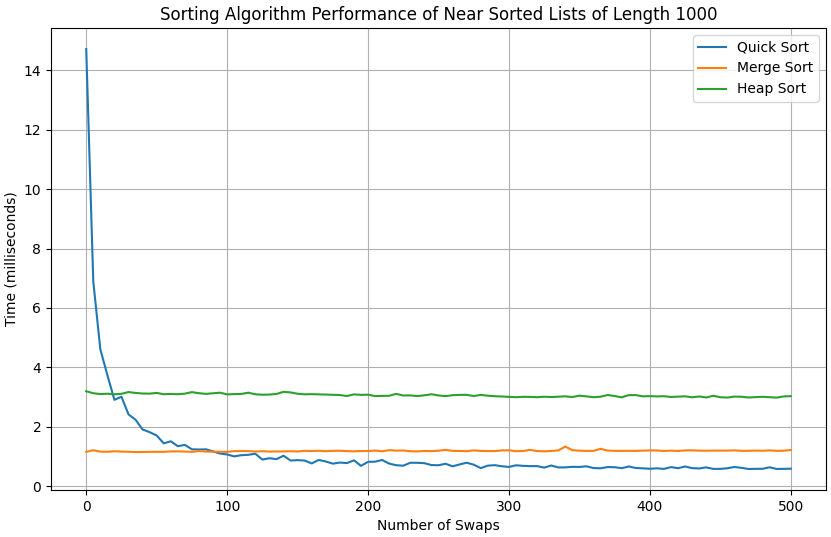

Figure 6: Swaps vs. Time (ms) for Quick, Merge, and Heap Sort

Observing the runtime of each algorithm on sorted lists relative to the number of swaps performed, Quick Sort began to outperform the other sorting algorithms when the number of swaps reached 100. As a more general observation, the performance of Quick Sort was more appealing when the number of swaps was 10% of the length of the list. This is likely due to the fact that having a swap rate of 10% introduces enough randomness into the list to nearly guarantee that the first element in the list is substantially closer to the median value, or at the very least further from the least value of the list.

Experiment 6

This experiment compares a typical Quick Sort to a Quick Sort variant that has two pivot points. The experiment was performed on lists of increasing sizes from 0 to 1000, incrementing the length by ten each iteration. Each iteration was performed five times, with the result being the average of the runtimes for improved accuracy.

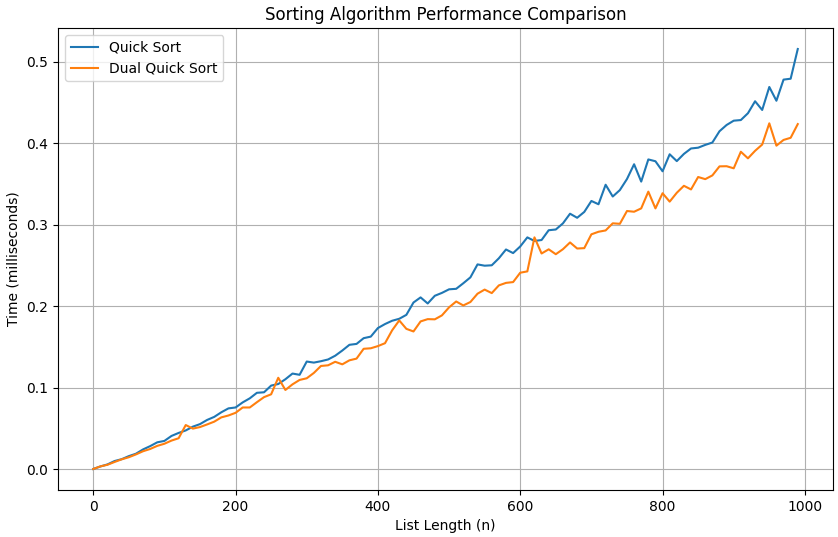

Figure 7: Swaps vs. Time (ms) for Single and Dual Quick Sort

Observing the runtime, the difference between Quick Sort with a single pivot versus with a dual pivot does not seem substantial enough to warrant implementation. However, as the list lengths used in the experiment reached 1000 elements, even though the absolute values of runtimes between Quick Sort and Dual Quick Sort were approximately 0.1ms, this denotes a 20% improvement in performance. Given the substantial improvement in relative performance, this implementation may prove useful with data sets of lengths that substantially exceed those used in the experiment.

Experiment 7

In this experiment, we compared the traditional recursive implementation of Merge Sort with a bottom-up iterative version. The bottom-up approach eliminates recursion by iteratively merging sublists using an increasing window size.

The experiment was conducted on randomly generated lists with lengths ranging from 0 to 5000, increasing by 50 each iteration. Each configuration was executed five times, and the minimum runtime was recorded to reduce noise from system interference.

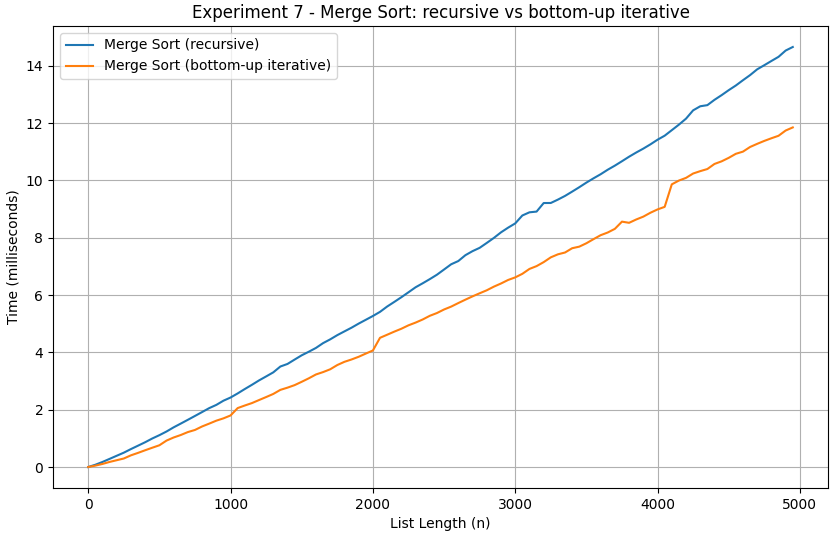

Figure 8: List Length vs. Time (ms) for recursive and bottom-up Merge Sort

As shown in Figure 8, the bottom-up Merge Sort consistently outperforms the recursive version across all tested list sizes. The improvement is modest, as both implementations share the same $\mathcal{O}(n \log n)$ time complexity. Most of the runtime cost comes from the linear merging process rather than recursion overhead, which limits the overall performance gain of the iterative approach.

Experiment 8