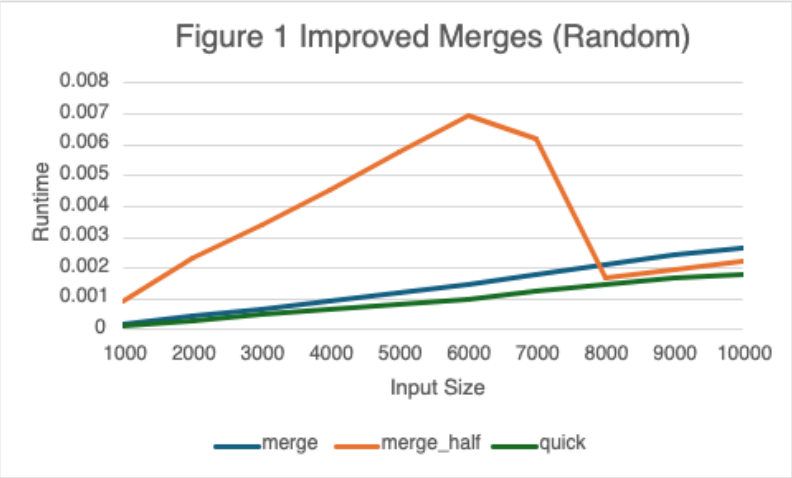
Aiden Smith
10/24
Dr. Wang
CS240

Part 4 Experimental Analysis

Figure 1

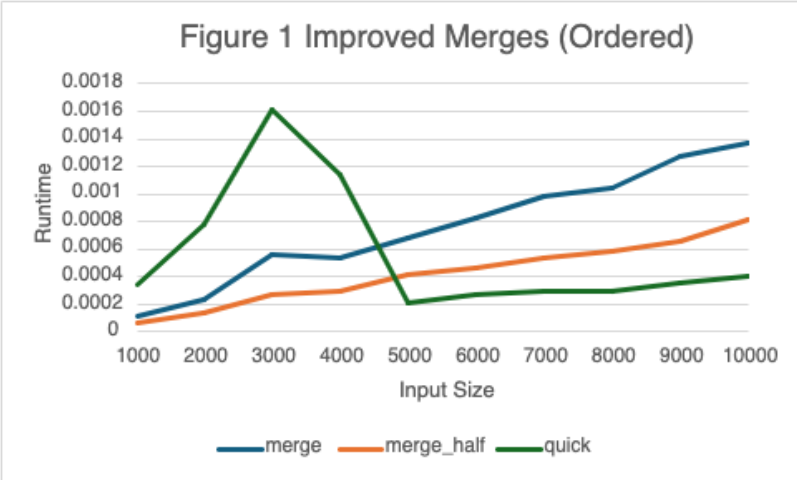
Improved Merges. This figure should compare the performance of mergeSortHalfSpace to the original merge sort implementation as well as quicksort. Results should be shown for both random and ordered inputs. To avoid clutter, it may be helpful to plot the results for random inputs and ordered inputs separately.



Random:

Quick sort consistently outperforms both versions of merge sort for all input sizes. The execution time for quick sort increases linearly. This demonstrates quick sort's efficiency on randomly distributed data due to its average case time complexity. The standard merge sort algorithm has better performance than the improved merge sort for sizes up to 7,000. However, at N=8,000 merge_half shows a increase in efficiency. This improvement in merge_half means the better utilization of memory works better with larger input sizes. However, quick sort remains the fastest algorithm in this range.

Aiden Smith
10/24
Dr. Wang
CS240



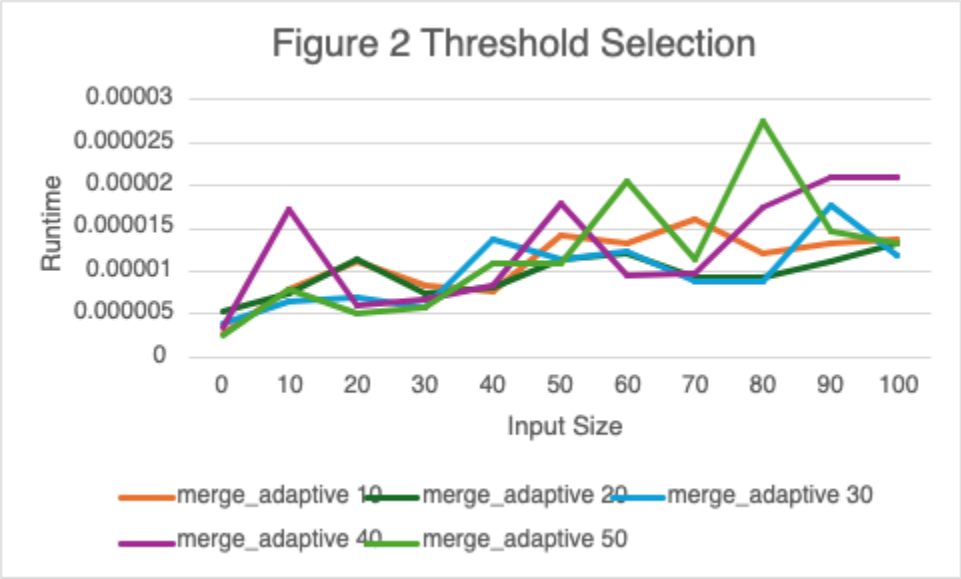
Ordered:

For these inputs the merge_half algorithm outperforms the standard merge sort for all input sizes. Quick sort has variable output on ordered data. At smaller input sizes, quick sort is slower than merge. This is due to quick sort's worst case time complexity when a poor pivot selection is in used. At N=5,000 quick sort's performance improves and remains good afterwards. This is a result from the algorithm switching to insertion sort.

Figure 2

This figure should show the data you used to select an appropriate threshold for switching strategies in mergeSortAdaptive.

Aiden Smith
10/24
Dr. Wang
CS240

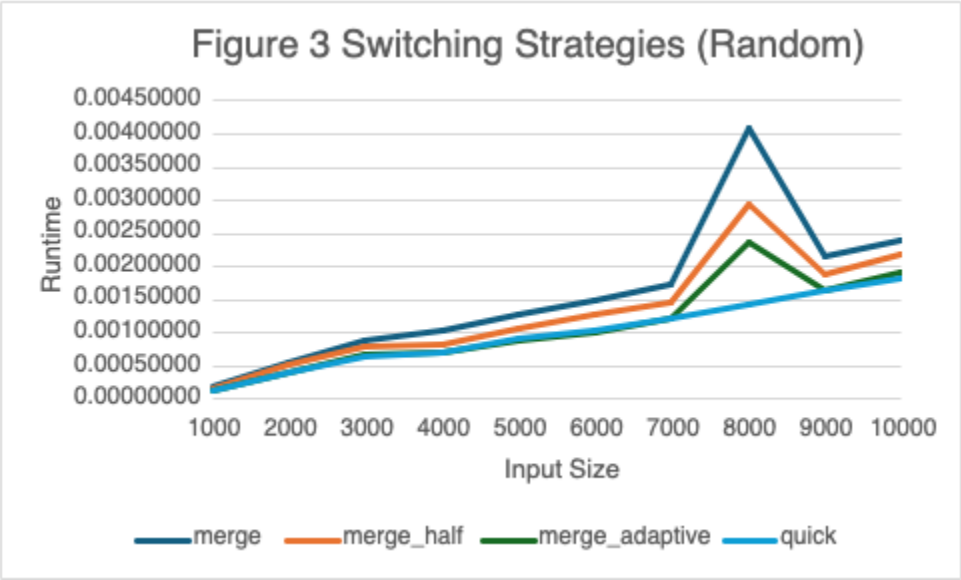


This figure shows the execution times of the mergeSortAdaptive algorithm with threshold values ranging from 10 to 50. This experiment is to determine the optimal threshold to switch from the use of merge sort to insertion sort for the sake of efficacy. When analyzing the performance of the algorithm on small input sizes, I found the threshold that gives the shortest execution time. The data backs up a threshold of ***30*** for the. This threshold provided the lowest execution times. By choosing a threshold of 30, the algorithm effectively uses the speed of insertion sort on small arrays while reducing the need for recursive merge sort calls. This improves in the performance.

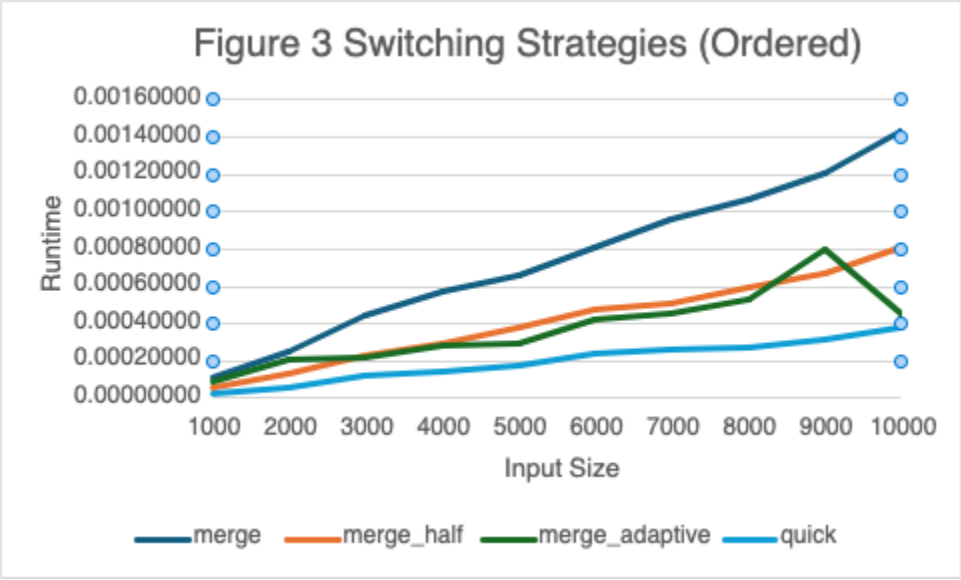
Figure 3

This figure should compare your tuned mergeSortAdaptive to the previous two merge sort implementations as well as quicksort. Results should be shown for both random and ordered inputs.

Aiden Smith
10/24
Dr. Wang
CS240



Random:
The adaptive merge sort does better than both the standard merge sort and the improved merge sort. This trend of performance consists as input size increases. Quick sort has similar efficiency as adaptive. However, when the input size grows quick sort outperforms merge_adaptive. At N=8,000 quick sort has better efficiency with larger datasets on random inputs. Both merge and merge_half show an increase in execution time at N=8,000. The spike may be due to the algorithms increased recursion calls or more memory use. In this scenario, adaptive, still maintains a better execution time showing the superiority.



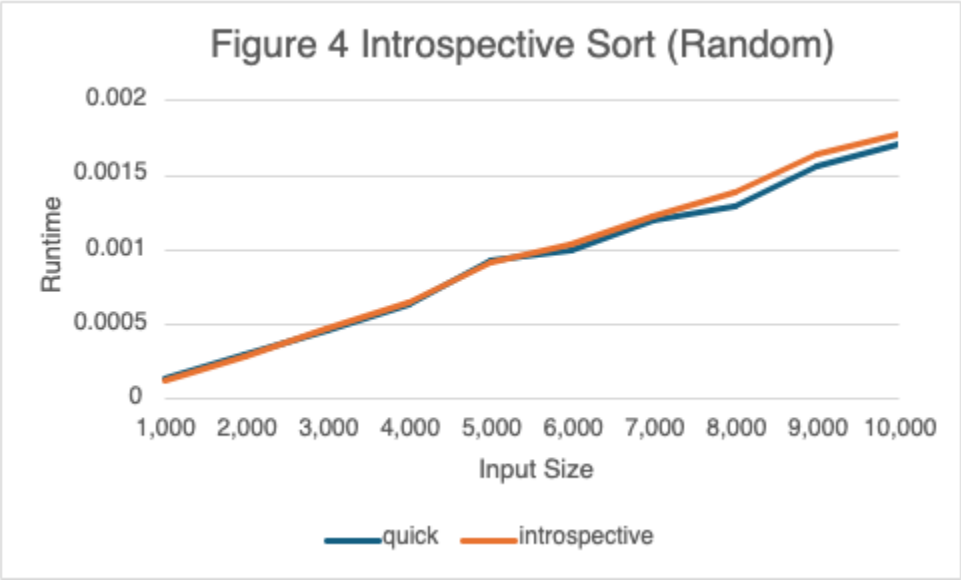
Ordered:

Quick sort has the best performance over all the merge algorithms. This trend is valid for all inputs, with the lowest execution times. The merge sort algorithms were varied. Merge_half outperformed both the standard merge sort and merge_adaptive on ordered inputs. However, at some input sizes, merge_adaptive performs better than merge_half. At N=5,000 merge_adaptive is faster than merge_half, making the case that it still has benefits on ordered data. The performance of quick sort on ordered inputs is odd. The algorithm is known to go toward O(n^2) time complexity on sorted data. However, this version is optimized to negate that.

Figure 4

This figure should compare introspective sort to quicksort for randomly generated inputs.

Aiden Smith
10/24
Dr. Wang
CS240

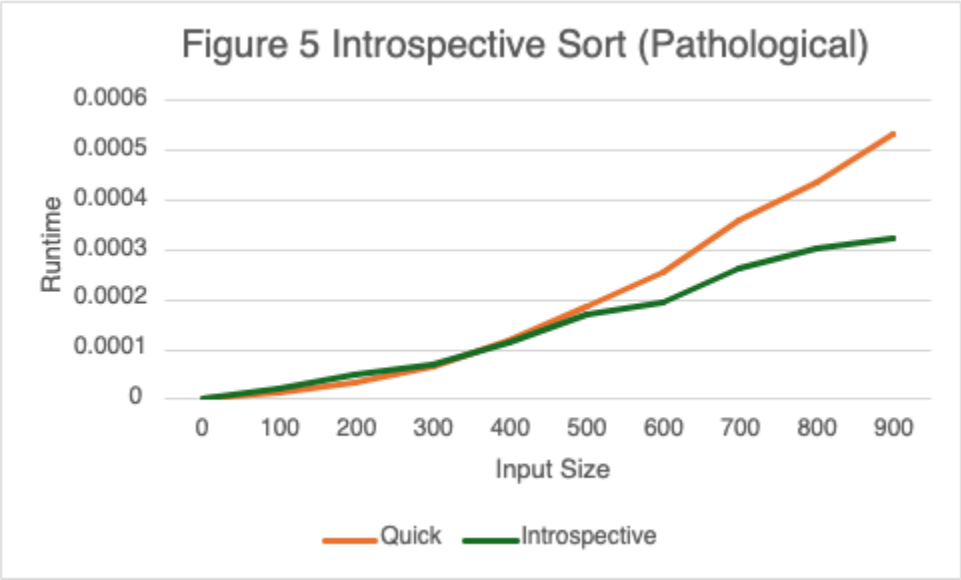


The experiment suggests that Introspective Sort matches the performance of Quick Sort with random inputs. Providing good case by case sorting without too much overhead memory use. This demonstrates that introspective is a good replacement for quick. This confirms that introspective does fine on average (random) case data.

Figure 5

This figure should compare introspective sort to quicksort for worst-case quicksort inputs.

Aiden Smith
10/24
Dr. Wang
CS240



This figure is a great demonstration that introspective negates the worst-case performance problems with quick sort given the “evil” inputs. By having O(nlog$_{10}$n) preformance, introspective outperforms quick on inputs that would catastrophic. This shows the greatness that introspective sort has. For sizes from N=400 and upwards, introspective not only surpasses quick in speed but also has a slower rate of increase in runtime. This shows that it can avoid the quadratic time complexity increase that comes with quick sort.