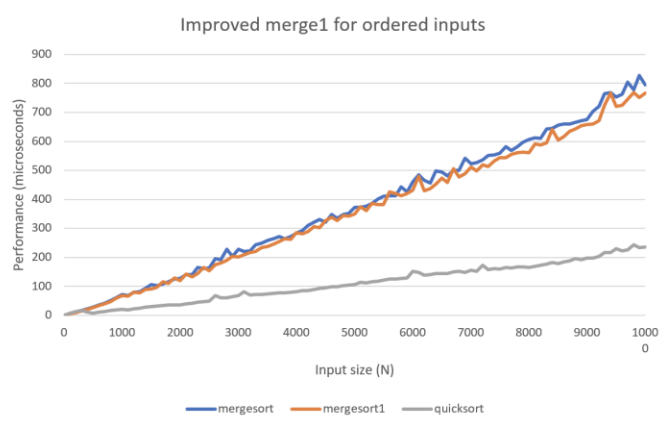
Experimental Results Document - PA3

Jackson Burns

Figure 1: The graphs below show the improvements in the performance of the mergeSort1 algorithm. This implementation of the improved merge sort (mergesort1) cut the size of the temporary array in half in order to reduce the space required by the algorithm. The difference in the run time between the original merge sort and mergesort1 for both random and ordered inputs is minimal, but it is due to the decrease in the amount of array assignments. Quick sort is significantly faster for both types of inputs since it is proven to be the fastest sorting algorithm aside from its worst-case performance.



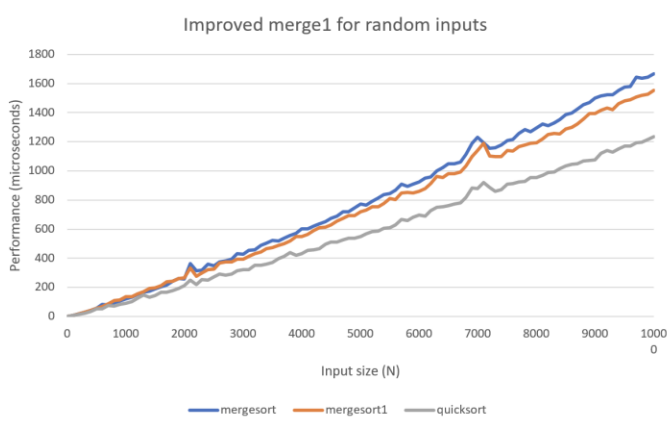


Figure 2: To determine an appropriate threshold value for the mergesort2 algorithm, I narrowed my search to values of 11 +- 3. For each possible threshold value, I tested the run time with input sizes starting with 50 and 100 and increasing each by multiples of 10 until reaching 1,000,000. Once the data was collected, I outlined the quickest runtimes for each input size across all threshold values. I found that the threshold value of 10 had 5/10 fastest run times, which is how I concluded it to be my optimal threshold value.

	Varrying threshold values						
Input Size	8	9	10	11	12	13	14
50	0.00000346	0.00000506	0.0000042	0.00000383	0.00000294	0.00000257	0.00000271
100	0.00001057	0.00000642	0.00000885	0.00000696	0.00000845	0.00000595	0.00000639
500	0.00004921	0.00004551	0.00004279	0.00004964	0.00004563	0.00004508	0.00004949
1000	0.00011047	0.00012102	0.00010847	0.00011825	0.00010341	0.00011049	0.00024942
5000	0.00070489	0.00064811	0.00065099	0.00070026	0.0006564	0.00065372	0.00065998
10000	0.00139285	0.00147882	0.00137719	0.00155167	0.00138782	0.00138027	0.00139814
50000	0.0088674	0.01011586	0.00963234	0.00913399	0.01086371	0.00957674	0.00949681
100000	0.02003282	0.02050097	0.01997797	0.01988233	0.01965595	0.01977737	0.01983051
500000	0.20666675	0.20731546	0.17859708	0.18065571	0.20662179	0.19143847	0.19326685
1000000	0.49034954	0.48619525	0.41554209	0.42757918	0.4825796	0.44540913	0.45165543

Figure 3: The graphs below demonstrate the performance improvement of the mergesort2 algorithm, which uses the improved merge operation from mergesort1 and switches to insertion sort when it reaches small enough sub-arrays. For both ordered and random inputs, we can see a

dramatic decrease in the run-time compared to the previous merge sorts while quicksort is still fastest when pathological inputs are avoided.

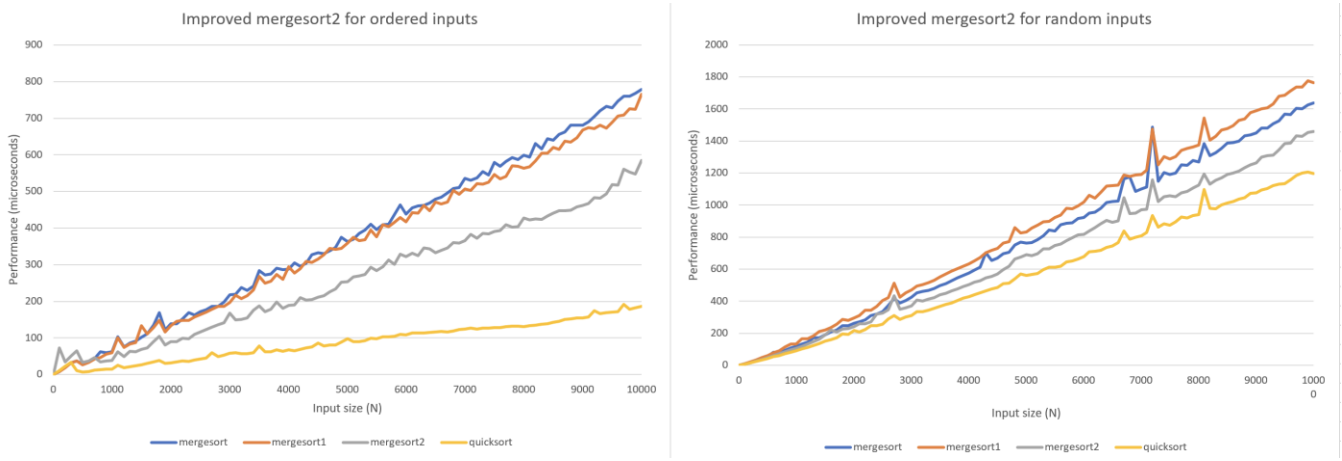


Figure 4: The following graph shows the performance of introspective sort which, when passed an array, performs quicksort if no pathological inputs are passed. If pathological inputs are passed it will merge sort the remaining values given that the worst-case analysis of merge sort (O(nlogn)) is far less taxing than quick sorts O(n^2). As seen in the figure, there is no noticeable difference between run-time performance between quicksort and introspective sort when given random inputs.

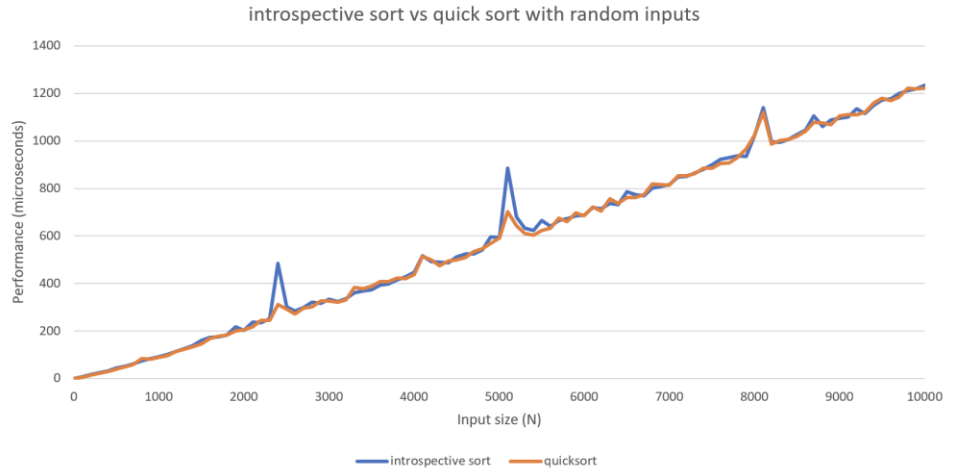
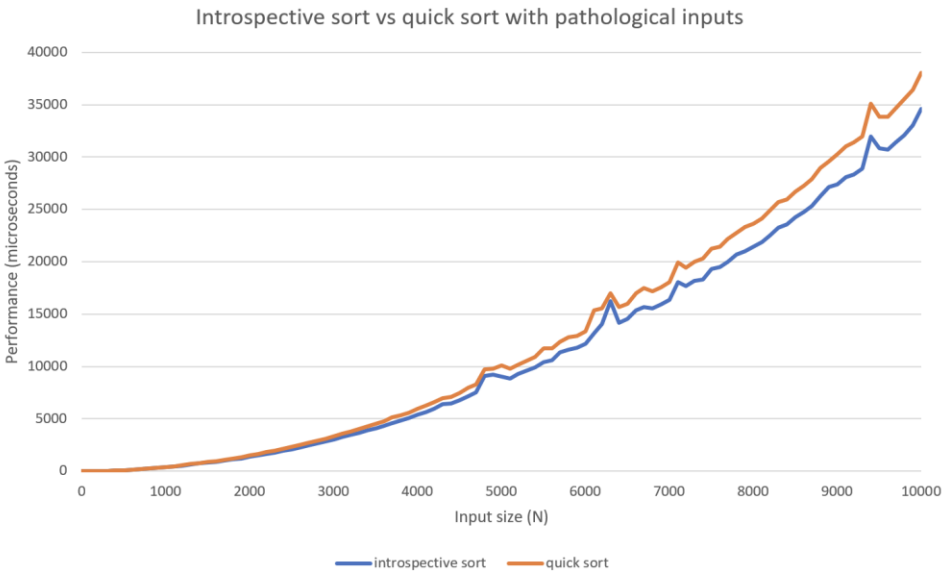


Figure 5: In the graph below, it is displayed that introspective sort has a better performance than quicksort when given pathological inputs. This conclusion makes sense given that introspective sort switches strategies when a conditional statement proving pathological inputs is presented. Introspective sort takes advantage of quick sorts run time in the average and best-case as well as merge sorts consistent O(nlogn) performance to provide better performance when given pathological inputs.


Introspective sort vs quick sort with pathological inputs
Performance (microseconds)
40000
35000
30000
25000
20000
15000
10000
5000
0
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
Input size (N)
introspective sort
quick sort