Jonathan Padilla

Dr. Esteban Parra Rodriguez

CSC 3130-01

February 8, 2025

Assignment 2 – Sorting Algorithms

5(Text):

1. Merge Sort
2. Quick Sort
3. Shell Sort
4. Insertion Sort, Selection Sort, Bubble Sort

I ranked Merge Sort as the fastest due to its best-case scenario being O(n log n), which is also the worst-case scenario. This gave it an edge over quicksort since the worst-case scenario for that algorithm had a time complexity of O(n^2), which has a faster growth rate regarding n-input when compared to O(n log n). Now, shell-sort should have at least a faster average time when compared to insertion sort, since it is known as a generalization/ optimization of insertion sort. And for selection sort and bubble sort, I hypothesize that their results should be similar, with bubble sort maybe being a bit worse, due to them having the same worst-case time complexity.

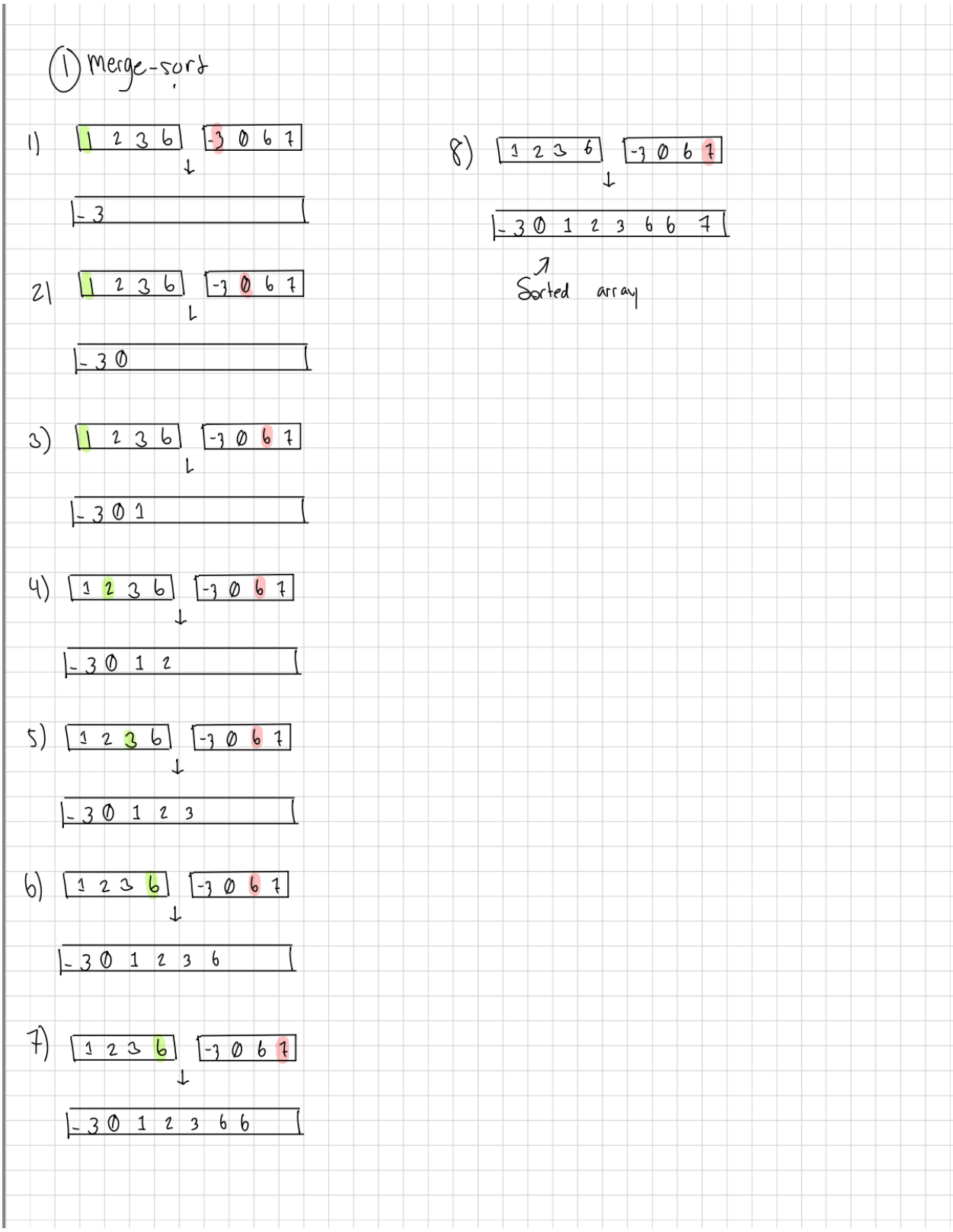

Figure 1 Problem 1(Text)

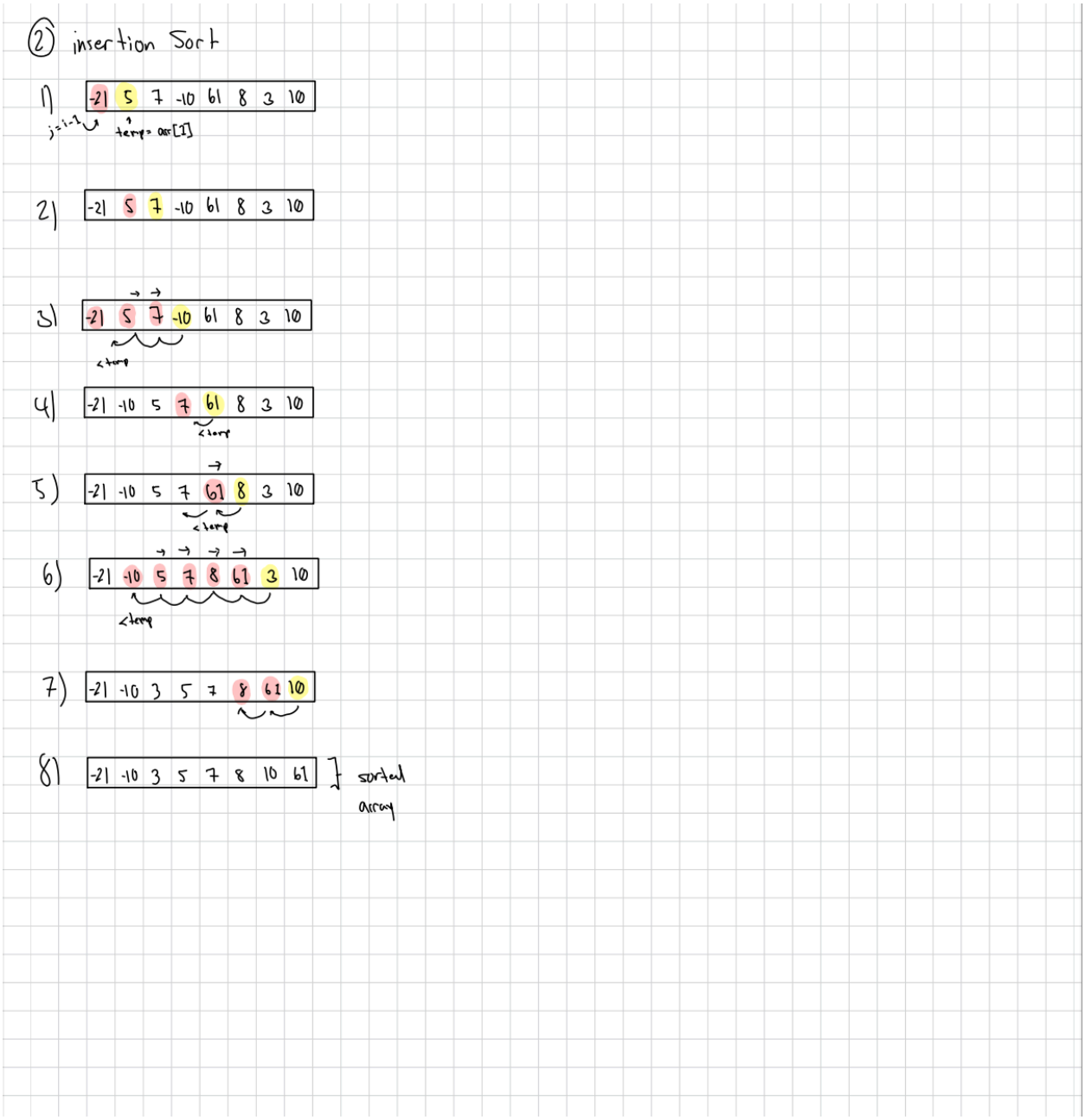

Figure 2: Problem 2(Text)

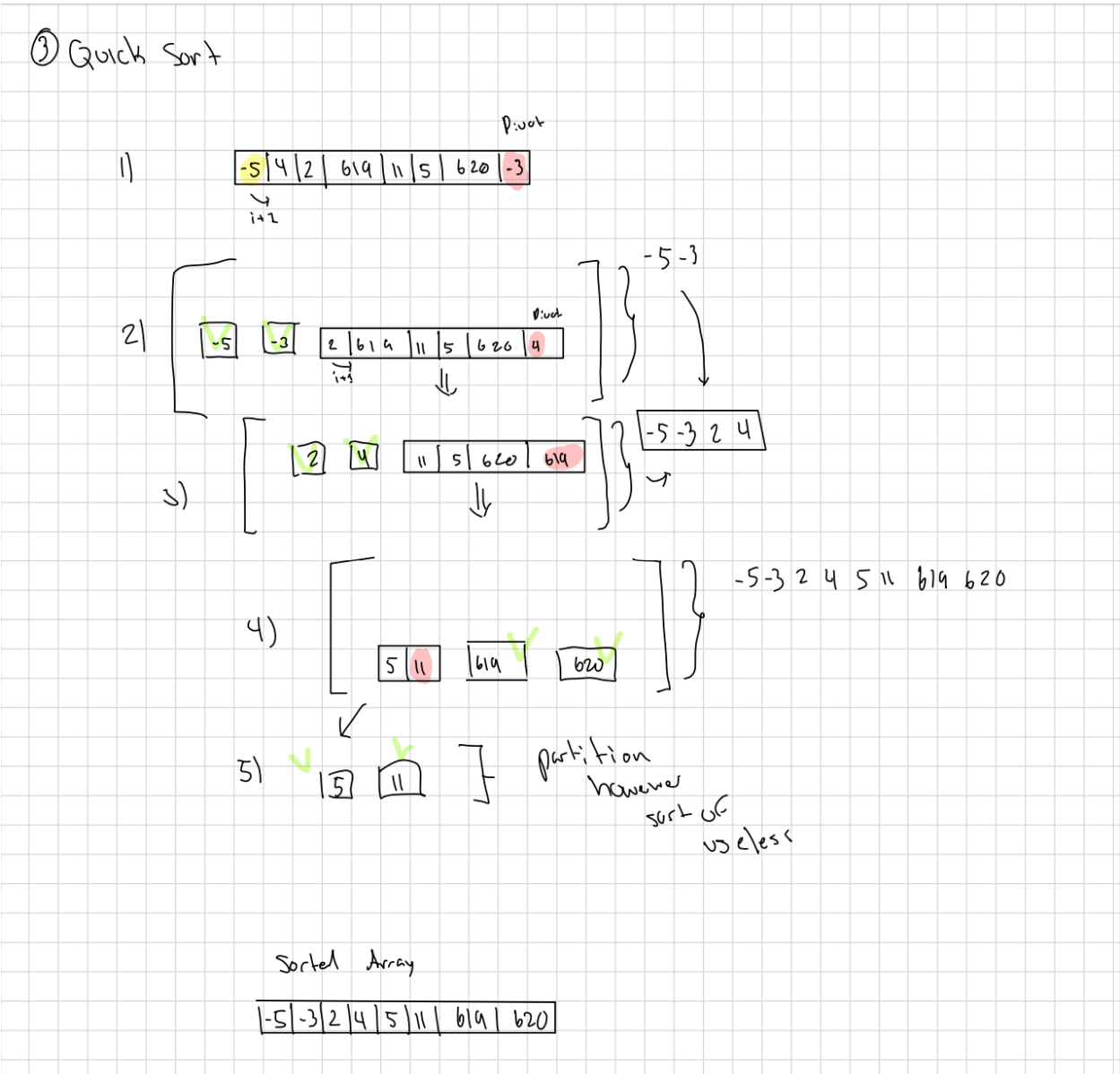

Figure 3: Problem 2 (Text)

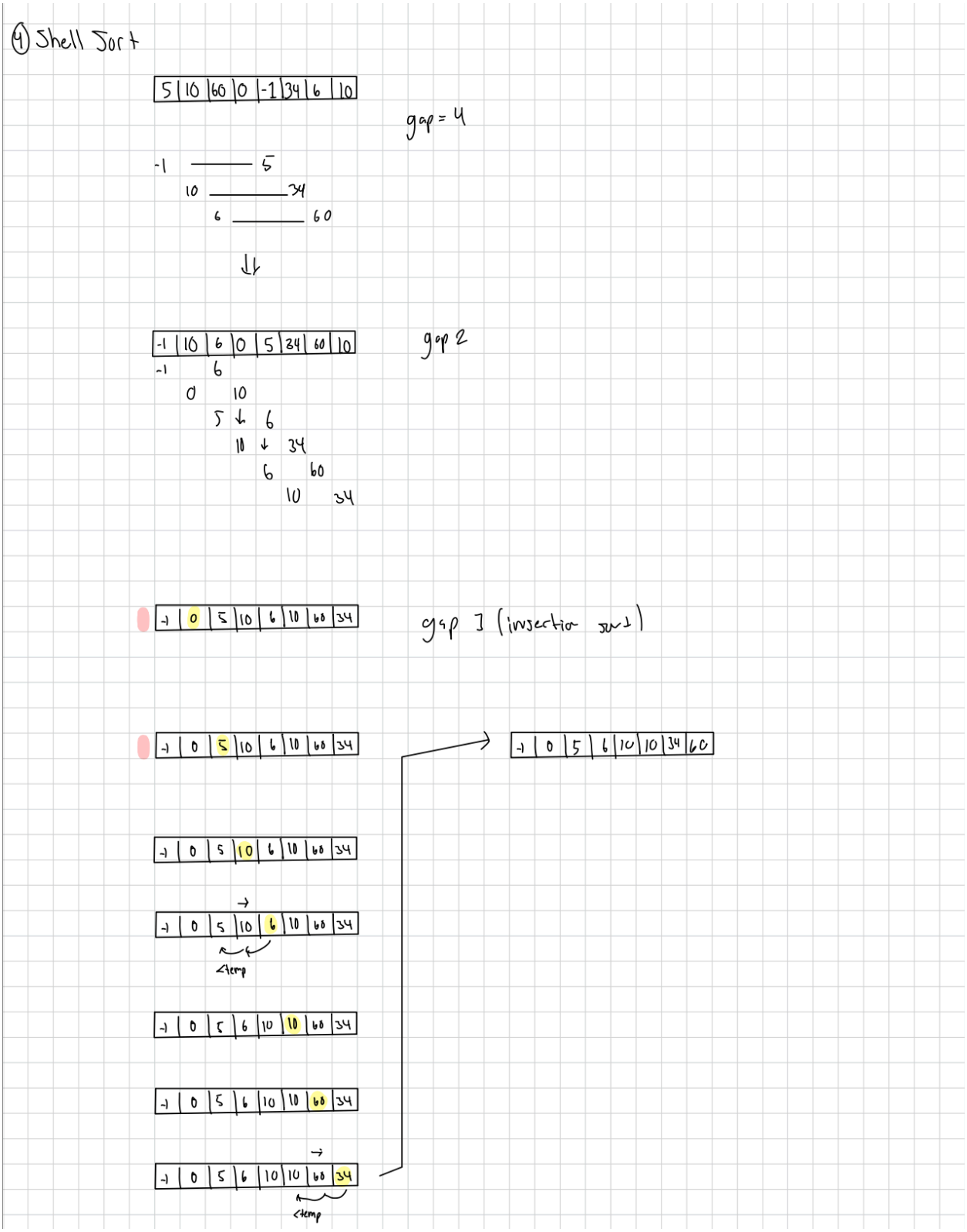

Figure 4: Problem 4 (Text)

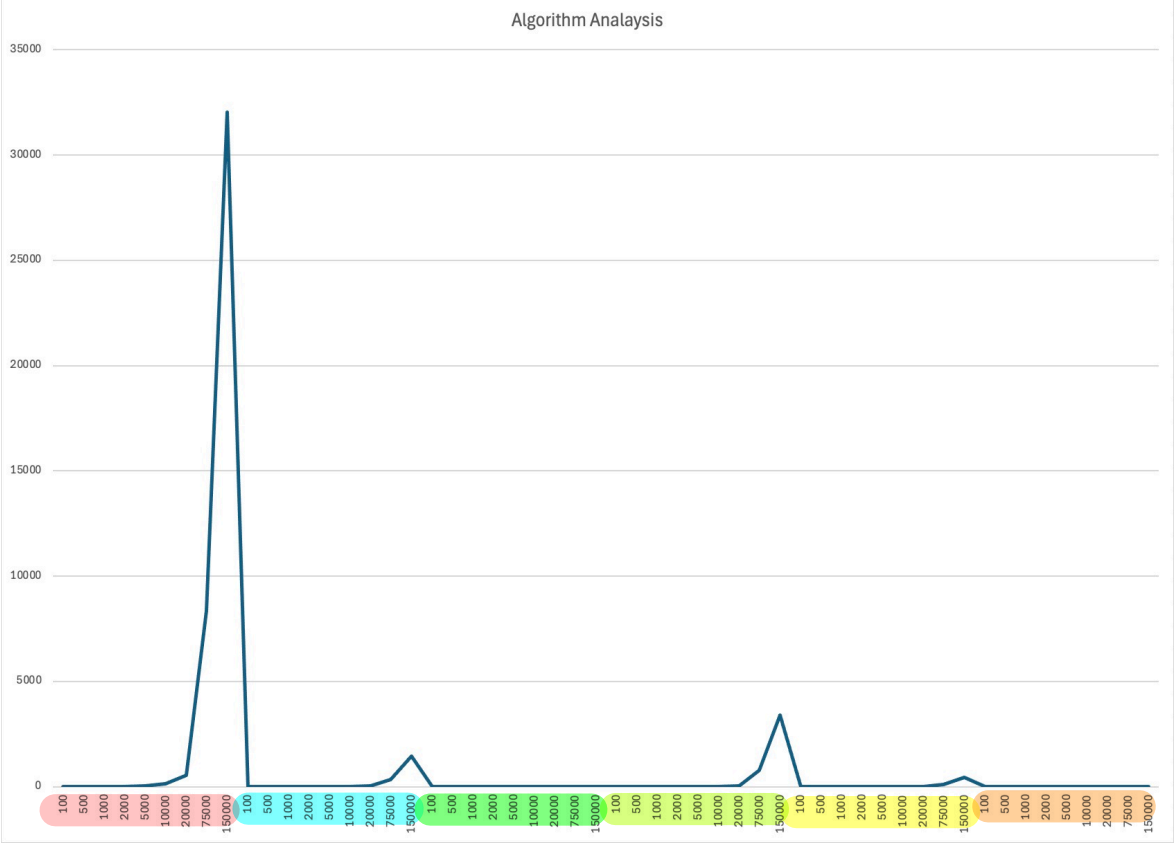

Figure 5 - Algorithm Analysis Chart

- Red – Bubble Sort
- Cyan – Insertion Sort
- Green – Shell Sort
- Light Green – Selection Sort
- Yellow – Quick Sort
- Orange – Merge Sort

Problem 9 (Text):

An issue that I found when plotting this data was that the numbers were so drastically large when it came to the bubble sort algorithm that every other change in data could not be seen. I then changed the maximum value to a smaller one that allowed me to see the growth spikes more clearly. However, after doing more research, I was able to find out that, for graphs with exponential growth, it is much more practical to use the logarithmic graph setting in Excel.

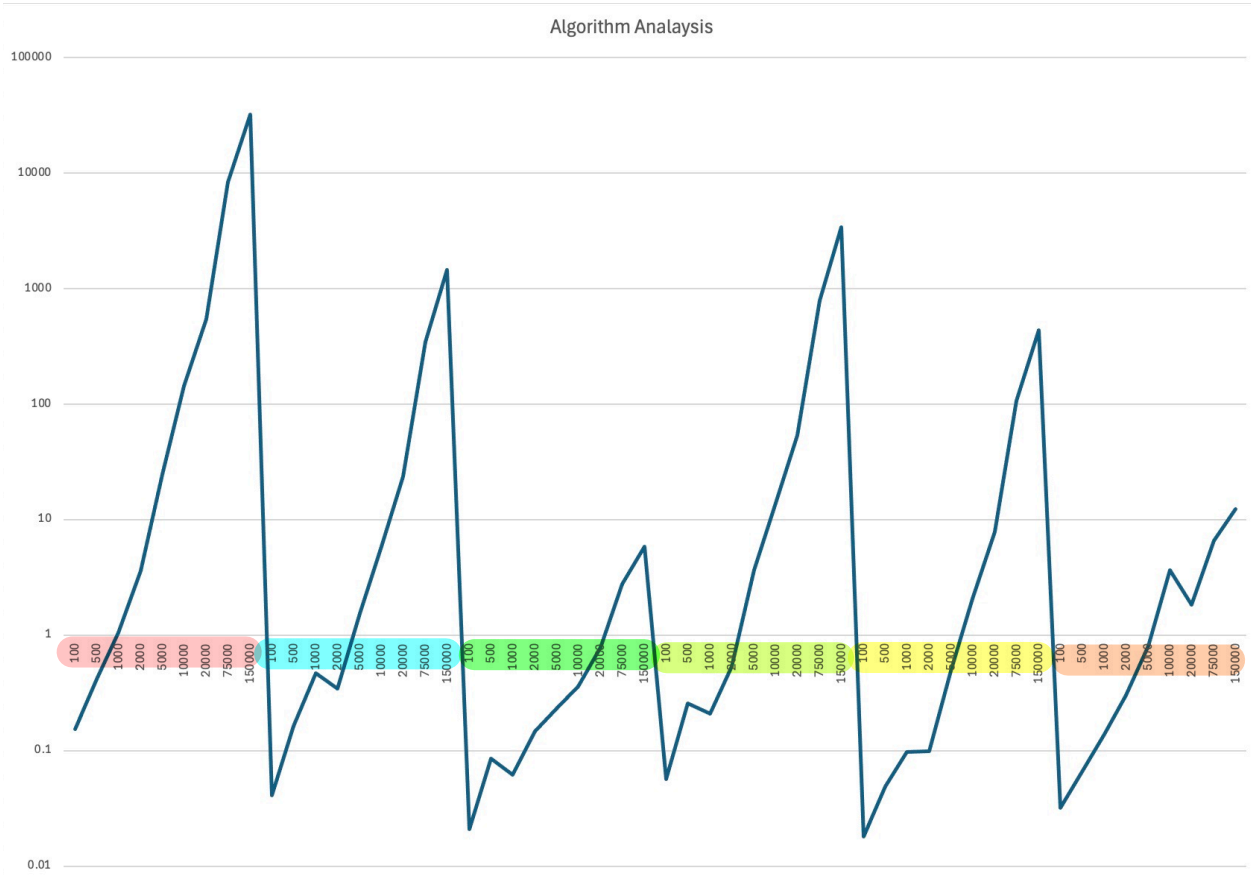

Figure 6 - Logarithmic Graph

Problem 10 (Text):

According to the graph, most of my predictions based on the time complexities were correct. Bubble sort, as I predicted, had the most time spent sorting an array of the highest n. One of the things that did surprise me, however, was that merge sort also had a spike during the higher input of n. I predicted that it would be generally better than all the other algorithms, but it ended up having a worse performance than shell sort at some points. I believe this is because merge sort has to create many new arrays, which leads to the degradation of the quality of how the code runs once the input becomes extremely large. Now I also believe that shell sort did way better than I predicted because it is an in-place algorithm and doesn't have to create constants like merge sort.

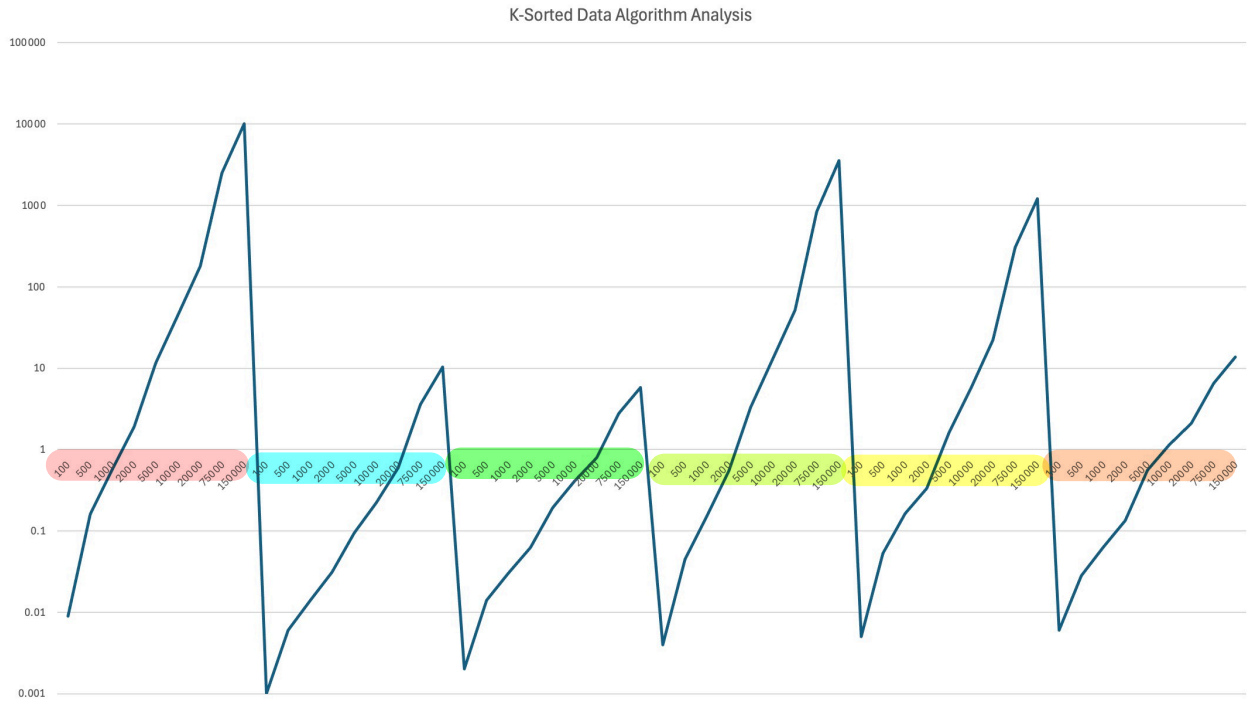

Figure 7 - K-Sorted Data Graph

- Red – Bubble Sort
- Cyan – Insertion Sort
- Green – Shell Sort
- Light Green – Selection Sort
- Yellow – Quick Sort
- Orange – Merge Sort

Problem 12(Text):

When the data was k-sorted, I observed some interesting differences. Insertion sort improved out of all the algorithms, by going from ~120ms to ~10ms according to the graph. I think this

because less elements are being shifted since it is nearly sorted data. Another drastic change was in the quicksort algorithm from ~430ms to ~1,200ms. I believe this has to do with something about the partitions that are made, since a nearly sorted array can cause unbalanced partitions, sort of like the one's in *Figure 2.*