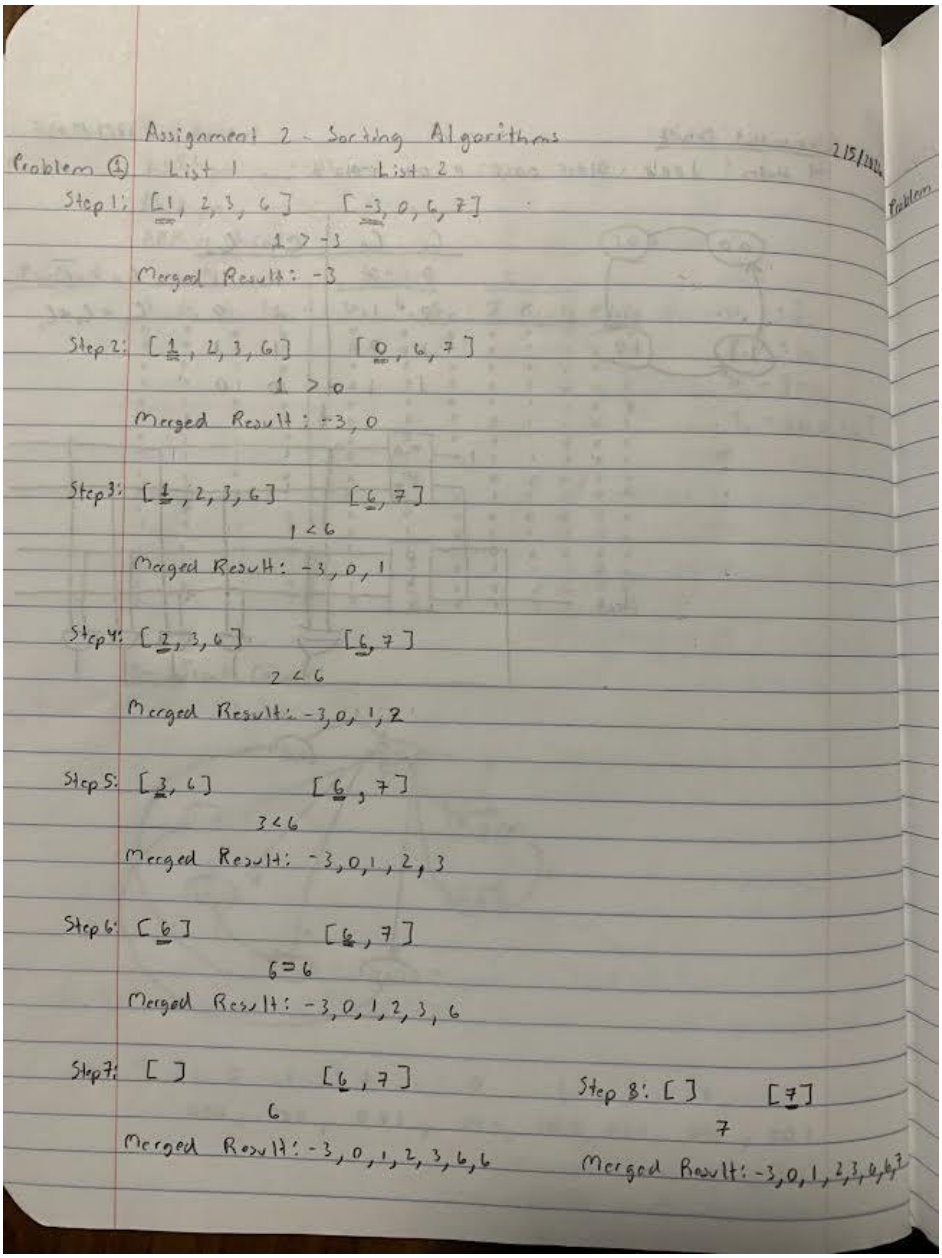
Problem 1:



Assignment 2 - Sorting Algorithms

Problem ①

List 1 [1, 2, 3, 6]

List 2 [-3, 0, 6, 7]

Step 1: 1 > -3

Merged Result: -3

Step 2: 1 > 0

Merged Result: -3, 0

Step 3: 1 < 6

Merged Result: -3, 0, 1

Step 4: 2 < 6

Merged Result: -3, 0, 1, 2

Step 5: 3 < 6

Merged Result: -3, 0, 1, 2, 3

Step 6: 6 = 6

Merged Result: -3, 0, 1, 2, 3, 6

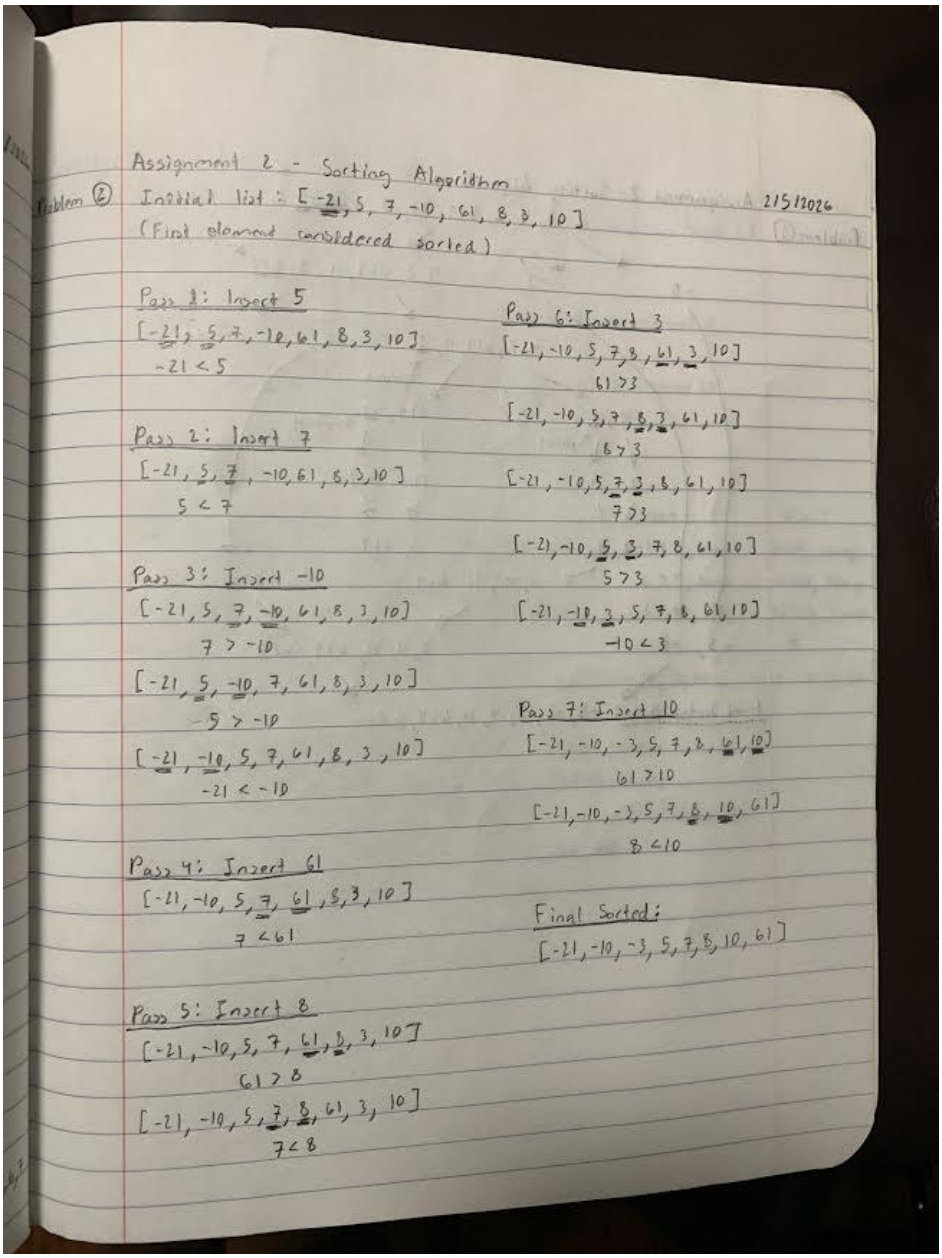
Step 7: 6

Merged Result: -3, 0, 1, 2, 3, 6, 6

Step 8: 7

Merged Result: -3, 0, 1, 2, 3, 6, 6, 7

Problem 2:



Assignment 2 - Sorting Algorithm

Problem ②

2/5/2026

Initial list : [-21, 5, 7, -10, 61, 8, 3, 10]

(First element considered sorted)

Pass 1: Insert 5

[-21, 5, 7, -10, 61, 8, 3, 10]

-21 < 5

Pass 2: Insert 7

[-21, 5, 7, -10, 61, 8, 3, 10]

5 < 7

Pass 3: Insert -10

[-21, 5, 7, -10, 61, 8, 3, 10]

7 > -10

[-21, 5, -10, 7, 61, 8, 3, 10]

5 > -10

[-21, -10, 5, 7, 61, 8, 3, 10]

-21 < -10

Pass 4: Insert 61

[-21, -10, 5, 7, 61, 8, 3, 10]

7 < 61

Pass 5: Insert 8

[-21, -10, 5, 7, 61, 8, 3, 10]

61 > 8

[-21, -10, 5, 7, 8, 61, 3, 10]

7 < 8

Pass 6: Insert 3

[-21, -10, 5, 7, 8, 61, 3, 10]

61 > 3

[-21, -10, 5, 7, 8, 3, 61, 10]

8 > 3

[-21, -10, 5, 7, 3, 8, 61, 10]

7 > 3

[-21, -10, 5, 3, 7, 8, 61, 10]

5 > 3

[-21, -10, 3, 5, 7, 8, 61, 10]

-10 < 3

Pass 7: Insert 10

[-21, -10, -3, 5, 7, 8, 61, 10]

61 > 10

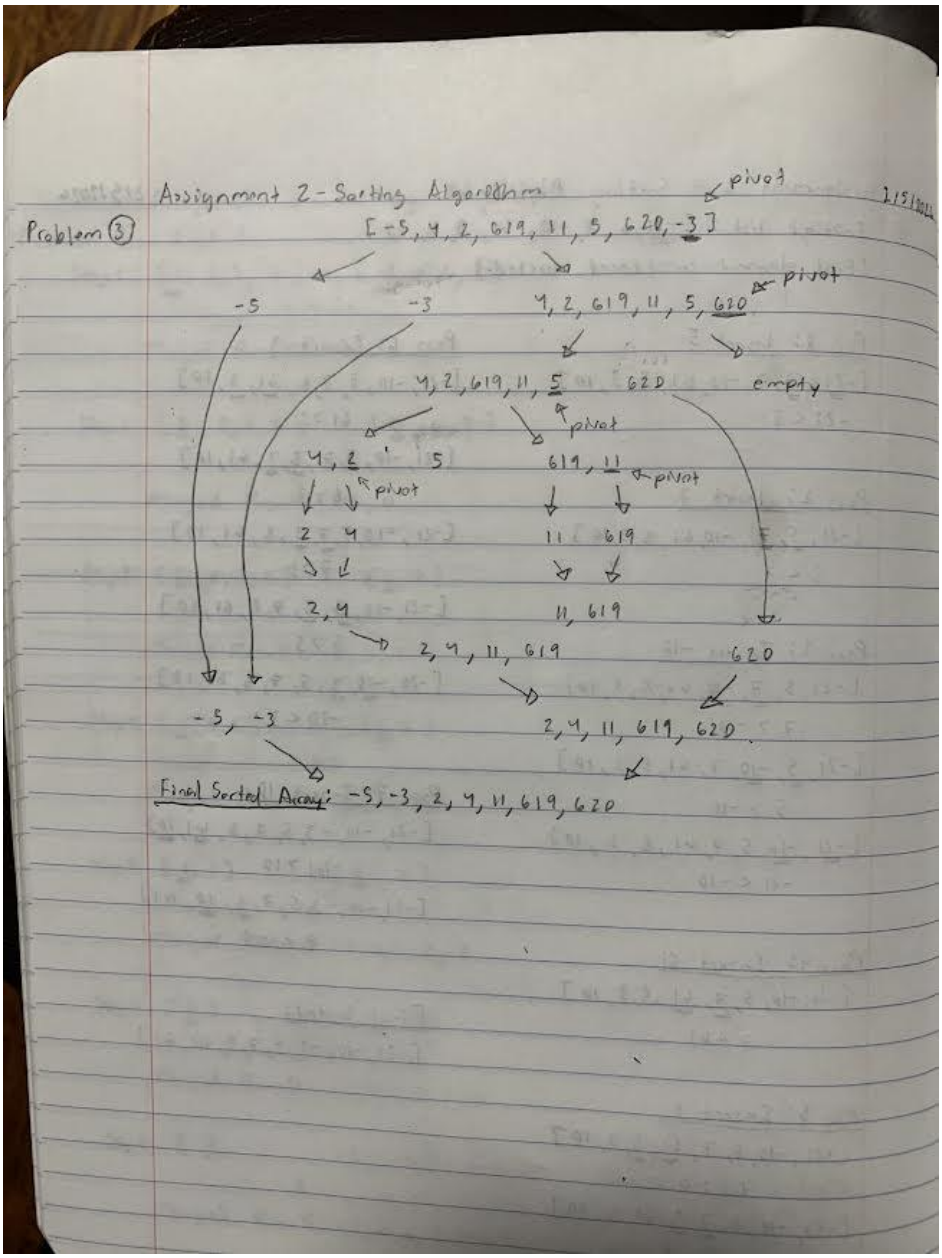
[-21, -10, -3, 5, 7, 8, 10, 61]

8 < 10

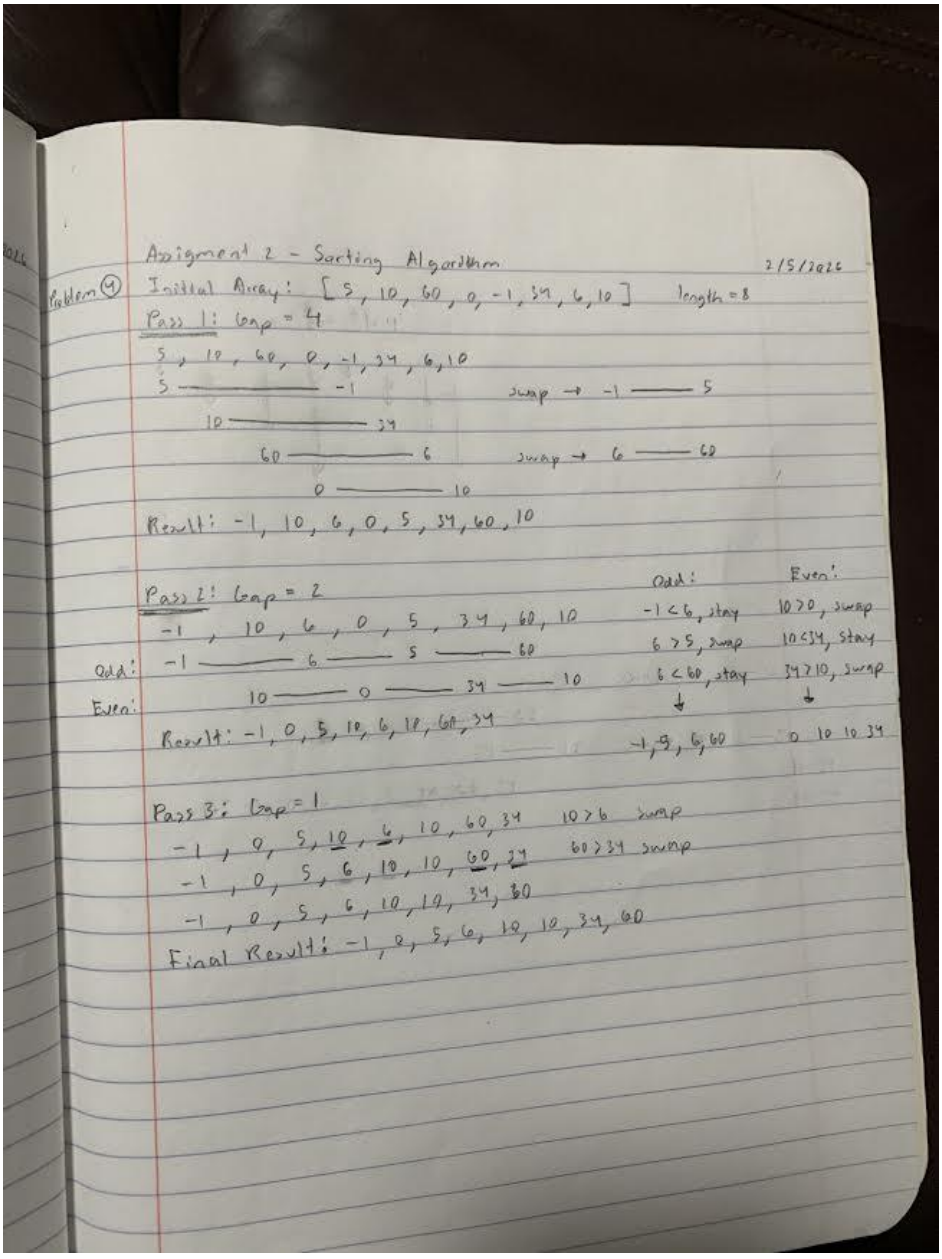
Final Sorted:

[-21, -10, -3, 5, 7, 8, 10, 61]

Problem 3:


Assignment 2 - Sorting Algorithm
Problem (3)
[-5, 4, 2, 619, 11, 5, 620, -3]
pivot
-5
-3
4, 2, 619, 11, 5, 620
pivot
4, 2, 619, 11, 5
620
empty
pivot
4, 2
5
619, 11
pivot
pivot
2
4
11
619
2, 4
11, 619
2, 4, 11, 619
620
-5, -3
2, 4, 11, 619, 620
Final Sorted Array: -5, -3, 2, 4, 11, 619, 620

Problem 4:



Assignment 2 - Sorting Algorithm 2/5/2026

Problem ④ Initial Array: [5, 10, 60, 0, -1, 34, 6, 10] length = 8

Pass 1: Gap = 4

5, 10, 60, 0, -1, 34, 6, 10

5 —— -1 swap → -1 —— 5

10 —— 34

60 —— 6 swap → 6 —— 60

0 —— 10

Result: -1, 10, 6, 0, 5, 34, 60, 10

Pass 2: Gap = 2

-1, 10, 6, 0, 5, 34, 60, 10

Odd: -1 —— 6 —— 5 —— 60

Even: 10 —— 0 —— 34 —— 10

Result: -1, 0, 5, 10, 6, 10, 60, 34

Odd:	Even:
-1 < 6, stay	10 > 0, swap
6 > 5, swap	10 < 34, stay
6 < 60, stay	34 > 10, swap
↓	↓
-1, 5, 6, 60	0 10 10 34

Pass 3: Gap = 1

-1, 0, 5, 10, 6, 10, 60, 34 10 > 6 swap

-1, 0, 5, 6, 10, 10, 60, 34 60 > 34 swap

-1, 0, 5, 6, 10, 10, 34, 60

Final Result: -1, 0, 5, 6, 10, 10, 34, 60

Problem 5:

Ranking the six sorting algorithms:

1. Merge Sort (tie with Quick Sort)

2. Quick Sort (tie with Merge Sort)

3. Shell Sort

4. Insertion Sort (tie with Selection Sort)

5. Selection Sort (tie with Insertion Sort)

6. Bubble Sort

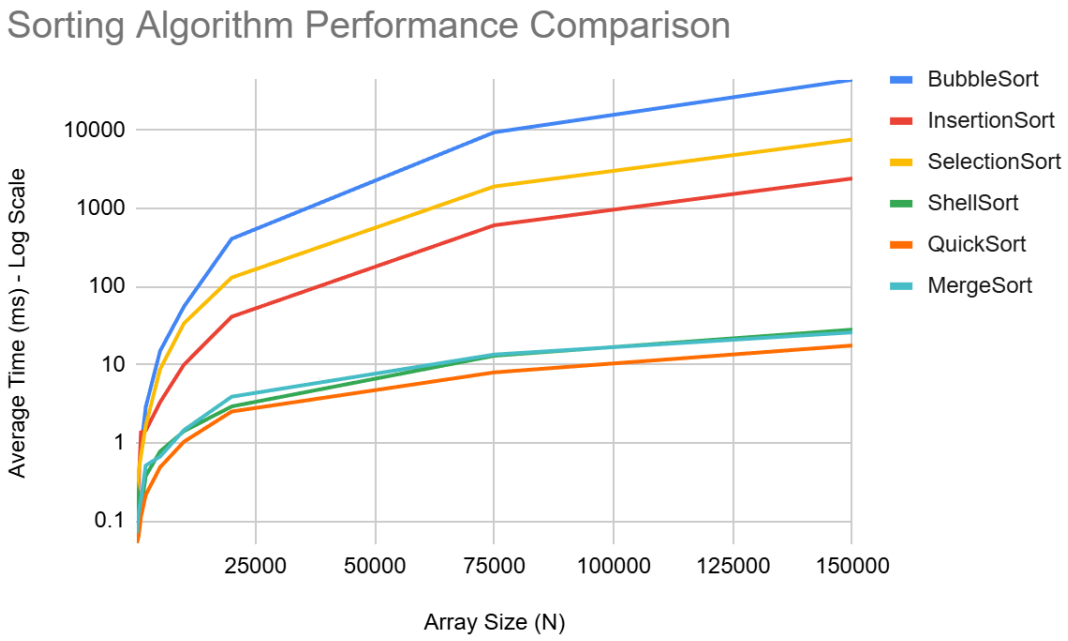
At the very top of the rank, there would be a tie between merge sort and quick sort because they both run in O(n log n) where n is the length of the list. If we count the worst cases, then merge sort would be 1st and quick sort would be 2nd since quick sort has a runtime of O(n^2). But on average, quicksort runs O(n log n) times which would make it tie with merge sort in 1st place. Both merge sort and quick sort have similar performance since they go by the strategy of divide and conquer.

The 3rd of the rank would be shell sort because it has a run time that is between O(n log n) and O(n^2). Shell sort is like an improved version of insertion sort since you can reach farther distances early, which makes it more efficient than simple quadratic but it's worse than O(n log n) because it relies on multiple insertion sort passes and the runtime depends on the how many gaps you have.

In 4th place, there is another tie between insertion sort and selection sort because they both have an average and worse case runtime of O(n^2). Insertion might have a better performance on nearly sorted lists, they both still grow at the same rate.

In the last place is bubble sort, this sortation run time is O(n^2) as well, but it does unnecessary comparisons and swaps which makes it the slowest sorting algorithm.

Problem 9:



The Issue: When I tried to plot all algorithms on a regular graph, BubbleSort, InsertionSort, and SelectionSort were so much slower than the other algorithms that QuickSort, MergeSort, and ShellSort looked like flat lines at the bottom. This made it more difficult to see the differences between the faster algorithms.

The Fix: I changed the y-axis to a logarithmic scale. This makes the graph show both very large and very small numbers clearly, so now all six algorithms can be seen and compared on the same graph.

Problem 10:

What I Observed:

The results clearly separated the algorithms into two groups. BubbleSort, InsertionSort, and SelectionSort were very slow, with BubbleSort being the worst at 43,812 ms for 150,000 elements. QuickSort, MergeSort, and ShellSort were much faster, with QuickSort being the fastest at only 17.5 ms for the same size.

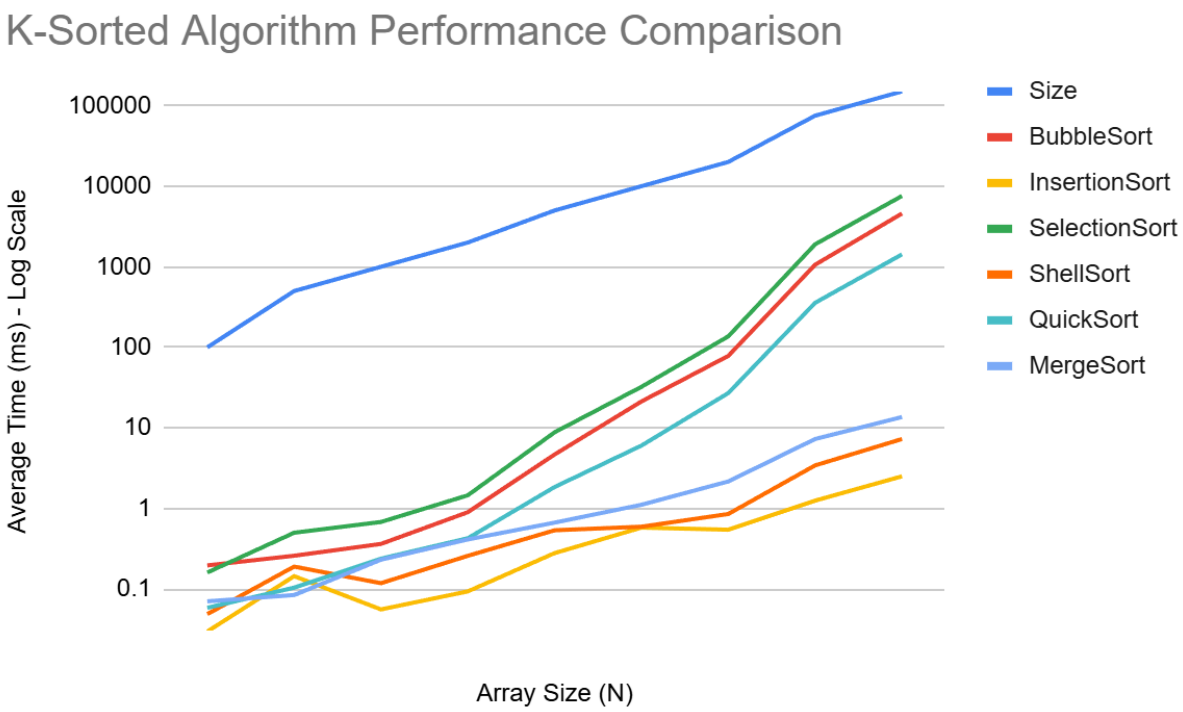
Comparison to My Predictions:

My ranking was mostly correct, it's just that QuickSort came in first place, slightly ahead of MergeSort. Interestingly, MergeSort and ShellSort look like they are nearly tied in the graph, performing very similarly throughout. BubbleSort was the slowest, exactly as I expected. The only surprise was that InsertionSort was noticeably faster than SelectionSort instead of tying with it.

Why the Results Matched:

The results matched my predictions because the math in the asymptotic analysis was mostly right. As the arrays got bigger, the difference between $O(n^2)$ and $O(n \log n)$ algorithms became huge. QuickSort beat BubbleSort by around 2,500 times at 150,000 elements. MergeSort and ShellSort performing similarly was unexpected since ShellSort isn't quite $O(n \log n)$, but ShellSort's gap sequence worked very efficiently. InsertionSort beat SelectionSort because it can skip comparisons when it finds the right spot early, while SelectionSort always does the full search.

Problem 12:



The Issue: Slow algorithms have much bigger runtimes than fast ones, so on a regular graph the fast algorithms look flat. Using a logarithmic scale fixes this so we can see all algorithms clearly.

Observations:

The rankings changed a lot on k-sorted data compared to random data. InsertionSort improved the most, going from 2,396 ms to only 2.5 ms (958x faster) because it only needs to move elements short distances. QuickSort got much slower, going from 17.5 ms to 1,424 ms (81x slower) because k-sorted data triggered its worst-case behavior. SelectionSort stayed about the same because it always does the same work no matter what.

Rankings:

- Random data: QuickSort was fastest, InsertionSort was 4th
- K-sorted data: InsertionSort was fastest, QuickSort dropped to 5th

This shows that how well an algorithm performs depends on the type of data, not just its Big-O complexity.