Sorting Algorithms:

1) MERGESORT:
    1) List 1: [1,25,31,16]
        i. [1,25] & [31,16]
        ii. [1] [25] [31] [16]
        iii. [1,25] [16,31]
        iv. [1,16,25,31]
    2) List 2: [-3,0,16,27]
        i. [-3,0] & [16,27]
        ii. [-3] [0] [16] [27]
        iii. [-3,0] [16,27]
        iv. [-3,0,16,27]
    3) Final List: [1,16,25,31] & [-3,0,16,27]
        i. [-3,0,1,16,16,25,27,31]

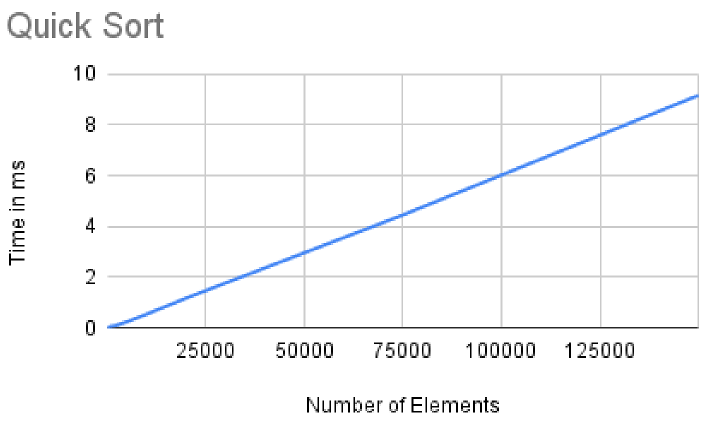
2) INSERTION SORT:
    1) List: [-1,-5,67,-10,21,8,4,1] (steps start below)
        i. [-5,-1,67,-10,21,8,4,1]
        ii. [-5,-1,67,-10,21,8,4,1]
        iii. [-10,-5,-1,67,21,8,4,1]
        iv. [-10,-5,-1,21,67,8,4,1]
        v. [-10,-5,-1,8,21,67,4,1]
        vi. [-10.-5,-1,4,8,21,67,1]
        vii. [-10,-5,-1,1,4,8,21,67] (SORTED)

3) QUICKSORT:
    1) List: [-5,42,6,19,11,25,26,-3] PERSONALLY CHOSEN PIVOT: 11
        i. [-5,6,-3] p=11[42,19,25,26]
        ii. Left sorted: P =-3: [-5,-3,6]
        iii. Right sorted: P = 25: [19,25,42,26]
            1. P = 26: [26,42]
            2. [19,25,26,42]
        iv. [-5,-3,6,11,19,25,26,42] (SORTED)

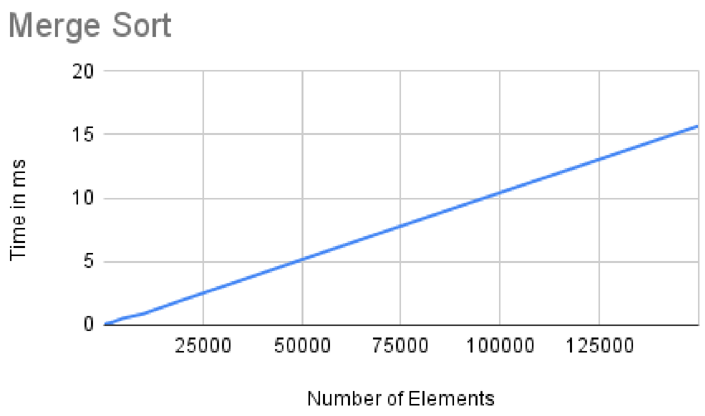
4) SHELL SORT:
    1) List: [15,14,-6,10,1,15,-6,0] GAP: 4
        i. [1,14,-6,10,15,15,-6,0]
        ii. [1,14,-6,10,15,15,-6,0]
        iii. [1,14,-6,10,15,15,-6,0]
        iv. [1,14,-6,0,15,15,-6,10] -> GAP: 2
        v. [-6,14,1,0,15,15,-6,10]
        vi. [-6,0,1,14,15,15,-6,10]
        vii. [-6,0,1,14,15,15,-6,10]
        viii. [-6,0,1,14,15,15,-6,10]
        ix. [-6,0,1,14,-6,10,15,15] -> GAP: 1
        x. [-6,0,1,14,-6,10,15,15]

xi. [-6,0,1,14,-6,10,15,15]
xii. [-6,0,1,14,-6,10,15,15]
xiii. [-6,0,1,-6,14,10,15,15]
xiv. [-6,0,1,-6,10,14,15,15] continues with gap: 1 until sort
xv. [-6,-6,0,1,10,14,15,15] (SORTED)

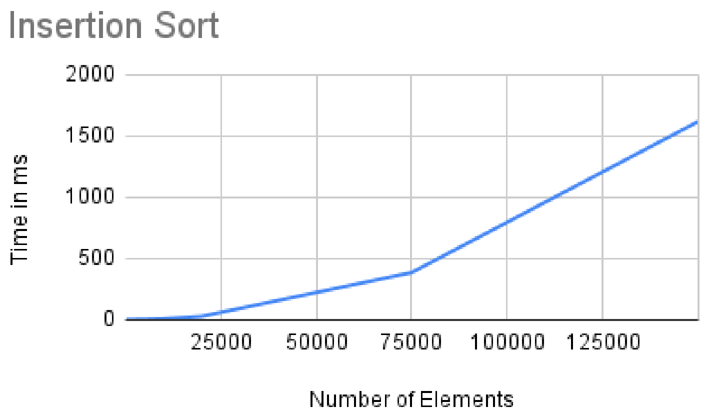
5) RANKING SORTING ALGORITHMS:
   1) Merge Sort: O(n log n) – O(n log n)
      i. In the best- and worst-case scenario with any list, merge sort will split into two and recursively sort. Then the arrays would merge by add the smallest element among the two arrays to a new array.
   2) Quicksort: O(n^2)-O(n log n)
      i. The worst case is O(n^2) when the pivot value results in empty left or right lists and has the maximum possible number of recursive calls
      ii. However, when left and right lists are of equal size or only requires 1 pass through the array, you can achieve O(n log n) time
      iii. This means if the programmer is able to find a good formula to determine the pivot value, then it can be one of the fastest sorting algorithms.
   3) Shell Sort: O(n^2)-O(nlog^2n)
      i. Shell sort uses gaps to sort and swap values, selection of gap increments is critical to performance of shell sort
      ii. Good gap increments: O(nlog^2n)
      iii. Bad gap increments: O(n^2)
   4) Insertion Sort: O(n^2)-O(n)
      i. Worst case scenario when the max possible number of swaps is reached since insertion sort removes element from unsorted
      ii. It is better than selection sort because if there is a partially sorted input the best case would be around O(n)
      iii. Reverse order list is a worst-case scenario
   5) Selection Sort: O(n^2) always
      i. Worst case scenario is that the algorithms always does two passes through the array since it has to find the smallest element in the array and swap it to the index with the first unsorted element
   6) Bubble Sort: O(n^2)-O(n)
      i. Worst case scenario is when all numbers must be swapped because bubble sort compares neighboring elements to find which elements to swap
      ii. Reverse order list is a worst-case scenario
6) Code: Implement sorting algorithms
7) Code: Performance testing framework
8) Code: Performance comparison
9) GRAPH OF RUNTIME OF SIX ALGORTHIMS:


Quick Sort
Time in ms
10
8
6
4
2
0
25000
50000
75000
100000
125000
Number of Elements

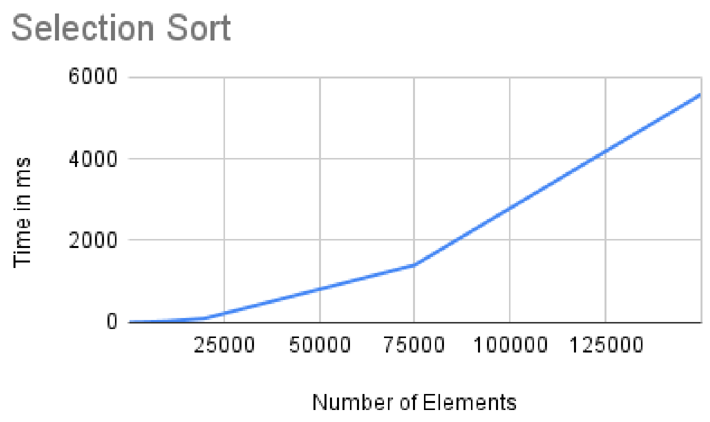
1)


Merge Sort
Time in ms
20
15
10
5
0
25000
50000
75000
100000
125000
Number of Elements

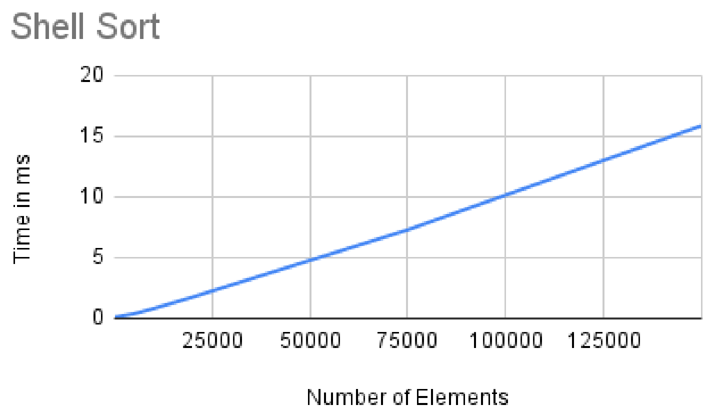
2)


Insertion Sort
Time in ms
2000
1500
1000
500
0
25000
50000
75000
100000
125000
Number of Elements

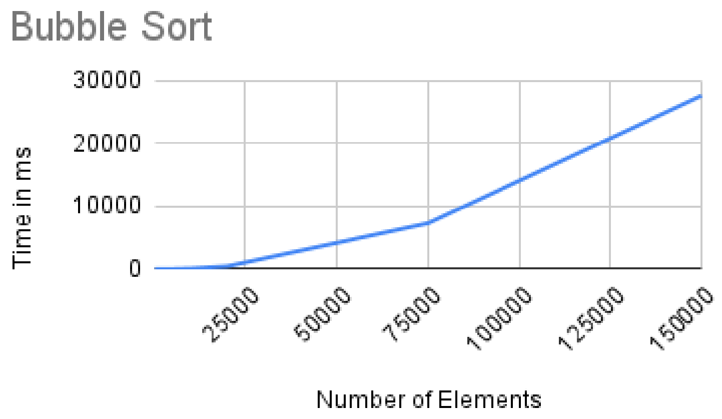
3)



4)



5)



6)

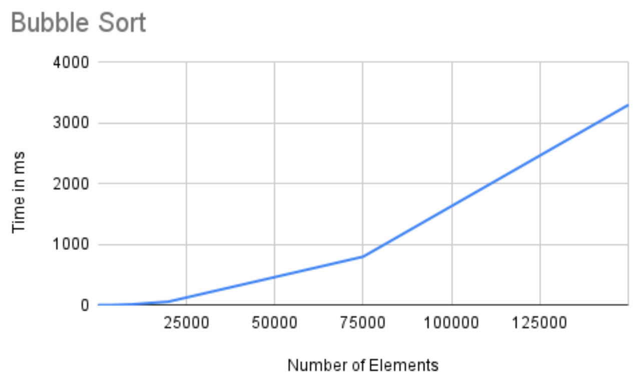
10) SUMMARY OF EXPERIMENTS:

1) Order of fastest to slowest with 150000 elements:
    i. Quick sort
    ii. Merge sort
    iii. Shell sort
    iv. Insertion sort
    v. Selection sort
    vi. Bubble sort

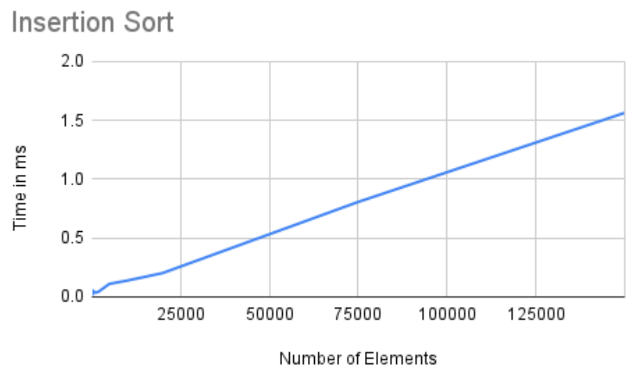
2) Quick sort and merge sort were switched in my original theories of time complexity. I put merge sort as the fastest since it is consistently O(nlogn) no matter the list. However, quick sort performed noticeably quicker. I noticed that bubble sort was extremely slow as predicted since it does not perform well with large datasets. I noticed that the algorithms with best case with O(nlogn) performed faster than the algorithms with the best case of O(n) or O(n^2). The rest of my predictions matched the results of the experiment.

11) Code: k-sorting

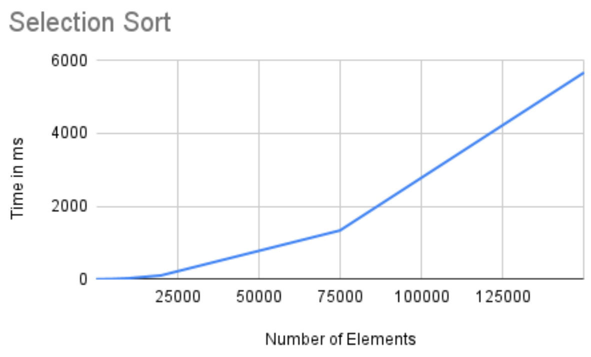
12) GRAPH OF RUNTIME OF SIX ALGORITHMS:



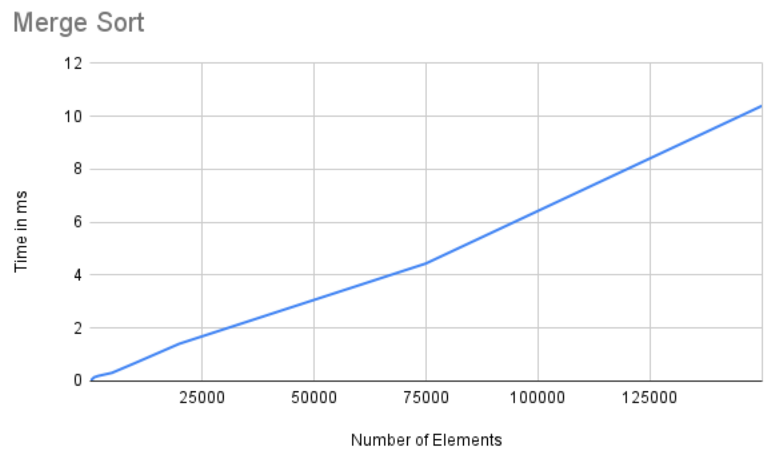
1)



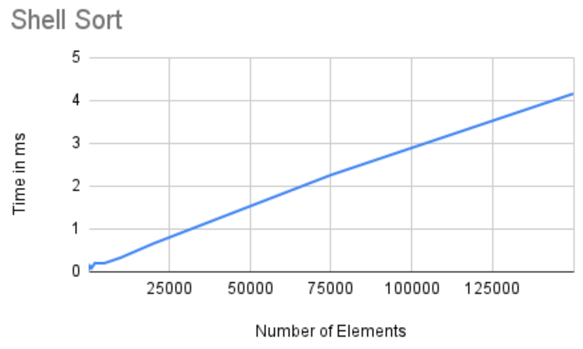
2)



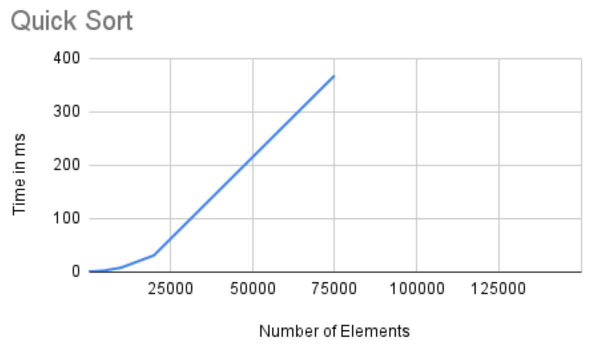
3)



4)



5)



6)

I found that all the algorithms performed similarly to the initial tests even with the 10. There were some that performed quicker than original by a few seconds. I had an issues with stack overflow issues with quicksort.