Rawshan Sinde

Sorting Algorithms

CSC 3130-02

02/03/2026

1. Merge Sort:

   Adding the smallest element among the two arrays to a new array

   Step 1: list1[0] = 1 and list2[0] = -3, -3 < 1, outList = [-3] → list1 = [1, 2, 3, 6], list2 = [0, 6, 7]

   Step 2: list1[0] = 1 and list2[1] = 0, 0 < 1, outList = [-3, 0] → list1 = [1, 2, 3, 6], list2 = [6, 7]

   Step 3: list1[0] = 1 and list2[2] = 6, 1 < 6, outList = [-3, 0, 1] →list1 = [2, 3, 6], list2 = [6, 7]

   Step 4: list1[1] = 2 and list2[2] = 6, 2 < 6, outList = [-3, 0, 1, 2] → list1 = [3, 6], list2 = [6, 7]

   Step 5: list1[2] = 3 and list2[2] = 6, 3 < 6, outList = [-3, 0, 1, 2, 3] → list1 = [6], list2 = [6, 7]

   Step 6: list[3] = 6 and list2[2] = 6, 6 == 6, outList = [-3, 0, 1, 2, 3, 6] → list1 = empty, list2 = [6, 7]

   Step 7: since list1 is empty, add rest of list2 → outList = [-3, 0, 1, 2, 3, 6, 6, 7]

2. Insertion Sort:

   Insert key into the sorted part

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

Step 1: (i= 1, key = 5) sorted = [-21], insert 5 into [-21], 5 is already >= -21 so no shift → [-21, 5]

Step 2: (i = 2, key = 7) [-21, 5], insert 7 into [-21, 5], 7 >= 5 so no shift → [-21, 5, 7]

Step 3: (i = 3, key = -10) insert -10 into [-21, 5, 7], 7 > -10 so shift, 5 > -10 so shift, -21 < -10 so stop → [-21, -10, 5, 7]

Step 4: (i = 4, key = 61) insert 61 into [-21, -10, 5, 7], 61 > 7 so no shift → [-21, -10, 5, 7, 61]

Step 5: (i = 5, key = 8) insert 8 into [-21, -10, 5, 7, 61], 61 > 8 so shift , 7 < 8 so stop → [-21, -10, 5, 7, 8, 61]

Step 6: (i = 6, key = 3) insert 3 into [-21, -10, 5, 7, 8, 61], 61 > 3 so shift, 8 > 3 so shift, 7 > 3 so shift, 5 > 3 so shift, -10 < 3 so stop → [-21, -10, 3, 5, 7, 8, 61]

Step 7: (i = 7, key = 10) insert 10 into [-21, -10, 3, 5, 7, 8, 61], 61 > 10 so shift, 8 < 10 so stop → [-21, -10, 3, 5, 7, 8, 10, 61]

3. Quick sort:

Repeatedly partitions input into low and high parts then recursively sorts each part

Array = [-5, 4, 2, 619, 11, 5, 620, -3]

Step 1: pivot = 619, low (<= 619): [-5, 4, 2, -3, 11, 5], high(>= 619): [620, 619]→ quicksort([-5, 4, 2, -3, 11, 5]) and quicksort([620, 619]

Step 2: quicksort([-5, 4, 2, -3, 11, 5]), pivot = 2, low(<= 2): [-5, -3, 2], high(>= 2): [4, 11, 5] →quicksort([-5, -3, 2]) and quicksort([4, 11, 5])

Step 3: quicksort([-5, -3, 2]), pivot = -3, low(<= -3): [-5, -3], high(>= -3): [2]→ result:[-5, -3, 2]

Step 4: quicksort([4, 11, 5], pivot = 11, low(<= 11): [4, 5], high(>= 11): [11]→ result: [4, 5, 11]

Step 5: combine results of step 3 and 4: [-5, -3, 2, 4, 5, 11]

Step 6: quicksort([620, 619]), pivot = 620, low(<= 620): [619], high(>= 620): [620]→result: [619, 620]

Final sorted list: [-5, -3, 2, 4, 5, 11, 619, 620]

4. Shell sort:

Treat the list as interleaved arrays determined by gap value K, sort each interleaved array using insertion sort style shifting, then finish with normal insertion sort

Array = [5, 10, 60, 0, -1, 34, 6, 10]

Gap K = 4

Step 1: Compare indices (0,4): [5, -1], -1 < 5 so shift, insert -1 → [-1, 10, 60, 0, 5, 34, 6, 10]

Step 2: Compare indices (1, 5): [10, 34], already in order, no shift → [-1, 10, 60, 0, 5, 34, 6, 10]

Step 3: Compare indices (2, 6): [60, 6], 6 < 60 so shift, insert 6 → [-1, 10, 6, 0, 5, 34, 60, 10]

Step 4: Compare indices (3, 7): [0, 10], already in order, no shift → [-1, 10, 6, 0, 5, 34, 60, 10]

Gap K = 2

Step 5: Sort indices (0, 2, 4, 6): [-1, 6, 5, 60], 5 < 6 so shift, place 5 after -1 → [-1, 10, 5, 0, 6, 34, 60, 10]

Step 6: Sort indices (1, 3, 5, 7): [10, 0, 34, 10], 0 < 10 so shift, place 0 before 10 → [-1, 0, 5, 10, 6, 34, 60, 10]; 10 < 34 so shift, place 10 before 34 → [-1, 0, 5, 10, 6, 10, 60, 34]

Gap K = 1

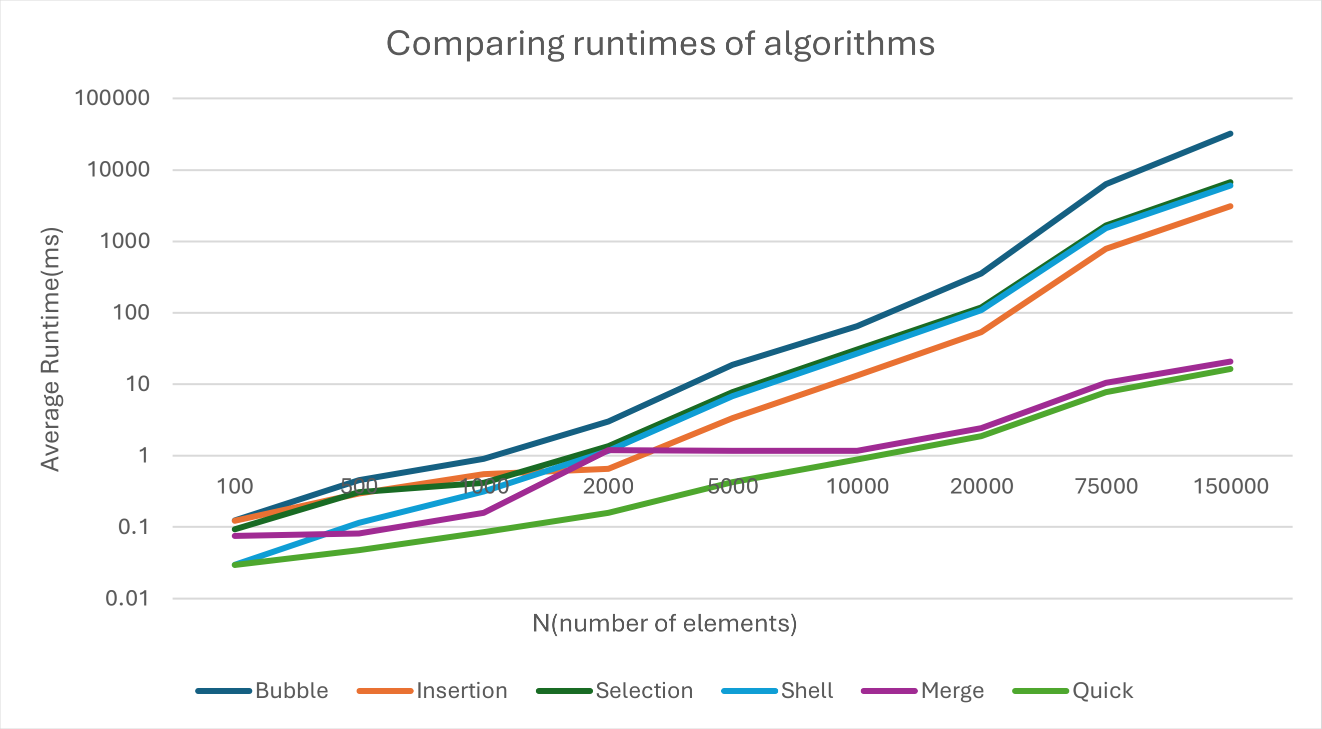
Step 7: Insert 6 into [-1, 0, 5, 10], 6 < 10 so shift, place 6 after 5 → [-1, 0, 5, 6, 10, 10, 60, 34]

Step 8: Insert 34 into [-1, 0, 5, 6, 10, 10, 60], 34 < 60 so shift, place 34 before 60 → [-1, 0, 5, 6, 10, 10, 34, 60]

Final sorted list: [-1, 0, 5, 6, 10, 10, 34, 60]

5. Ranking the six sorting algorithms based on asymptotic runtime, from fastest to slowest would be quick sort, merge sort, shell sort, insertion sort, selection sort, and bubble sort. Quick sort and merge sort are tied for the fastest because they run in (N log N) time. Shell sort is medium run time because the runtime is (N^1.5). Bubble sort, insertion sort, and selection sort are also tied for the slowest because the runtime is (N^2).

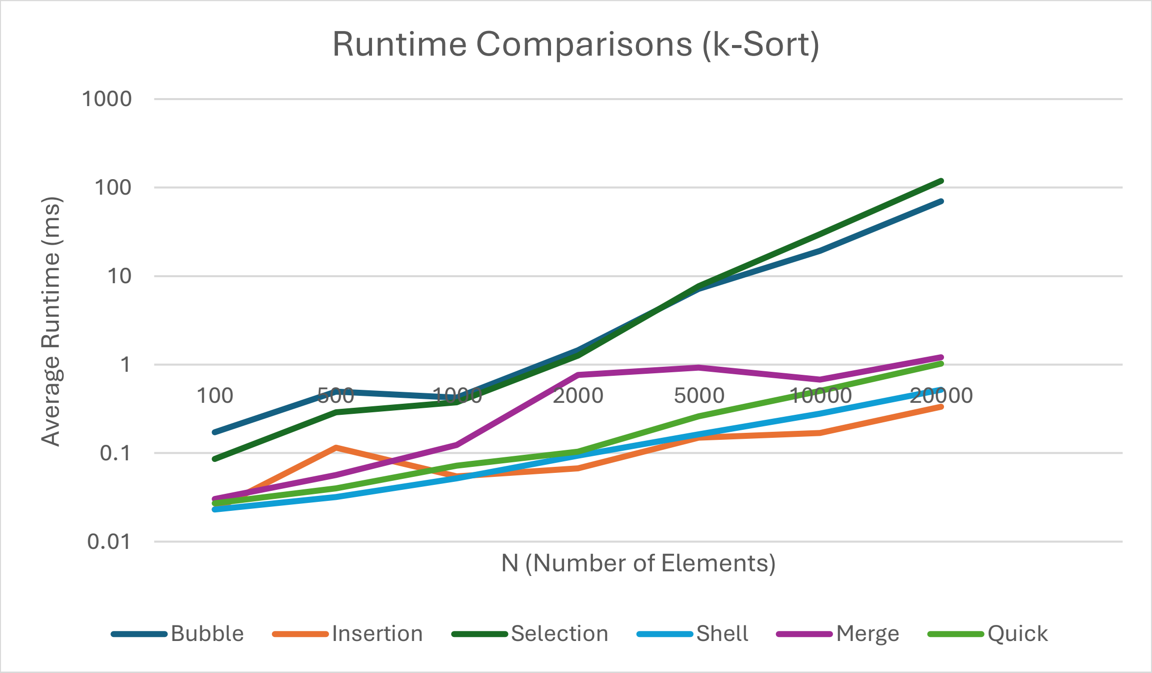
9.



10. I noticed that the difference in performance times of each sorting algorithm is seen the most clearly as input size increased. Relatively, bubble sort was the slowest while shell sort and selection sort were mostly tied for performance. Furthermore, insertion sort was relatively in the middle in terms of performance time but quick sort and merge sort were relatively the fastest. The actual performance time of quick sort and merge sort matched my predictions based on asymptotic analysis because of their time complexity which is (N log N) and that they were relatively the fastest. However, my prediction that shell sort would be in the middle was proven incorrectly because the actual performance time shows that insertion sort was relatively in the middle. Finally, based on actual performance time, bubble sort was the slowest while shell sort and selection sort were tied for performance time. The most significant thing I notice though, is that the performance time takes a pivot around 2000 elements because in one instance, insertion sort significantly slows down after 2000 elements. Another instance of this is shell sort which begins much faster than selection sort but then slows down to the relatively same runtime at 2000

elements. I also ran into the issue of large numbers and to try to deal with it, Excel/Word has an option to format the axis with logarithmic scale which I selected and it helped me view the large numbers better.

12.



For the kSorting implementation, I used a shorter sample size (only going through 20000 elements instead of 150000) due to time constraints. However, I am still able to draw a lot of conclusions from the performance run. I found that the runtimes were significantly faster under the 10-sorted data than on random data. The algorithms that performed significantly different were bubble sort, shell sort, and insertion sort. Surprisingly, selection sort maintained the same runtime as before, k-sorting didn't have any change on its runtime because selection sort still scans through the entire unsorted portion of the array to find the minimum value. Insertion sort and shell sort performed the fastest because the input was already nearly sorted. While bubble sort was still the slowest in terms of relative performance time, it did decrease significantly.

Again, the large input numbers did serve an issue when displaying the graph, but I used the logarithmic scale feature on the graph axis to help.