Question 1: Show the step-by-step process of merging two sorted lists using the merge sort algorithm. The sorted lists are: [1, 2, 3, 6] and [-3, 0, 6, 7].

The merge sort algorithm divides a given list into two halves and continues to divide (recursively) until each half contains one element (the base case). Then, it merges the lists (of size one) together until it is sorted. The merging is adding the sorted elements into a new array. The two given lists are already sorted, so we don't have to continuously halve the lists and just begin merging.

[1, 2, 3, 6] and [-3, 0, 6, 7]

Compare 1 and –3:

[-3]

[1, 2, 3, 6] and [-3, 0, 6, 7]

Compare 1 and 0:

[-3, 0]

[1, 2, 3, 6] and [-3, 0, 6, 7]

Compare 1 and 6:

[-3, 0, 1]

[1, 2, 3, 6] and [-3, 0, 6, 7]

Compare 2 and 6:

[-3, 0, 1, 2]

[1, 2, 3, 6] and [-3, 0, 6, 7]

Compare 3 and 6:

[-3, 0, 1, 2, 3]

[1, 2, 3, 6] and [-3, 0, 6, 7]

Compare 6 and 6:

[-3, 0, 1, 2, 3, 6]

[1, 2, 3, 6] and [-3, 0, 6, 7]

Lists are already sorted, so we can add the remaining elements of the second array into the new merged array.

Final sorted array: [-3, 0, 1, 2, 3, 6, 6,7]

Question 2: Show the step-by-step process of sorting a list using the insertion sort algorithm. The unsorted list is: [-21, 5, 7, -10, 61, 8, 3, 10]

The insertion sort algorithm works by starting at the second element (index 1), comparing its value to the first element, and choosing to switch the values if not sorted (value with higher index is less – choose to not swap if value with higher index is greater). This continues for the rest of the elements in the array. For example, the tenth element would be compared to the previous nine elements (which are already sorted) and switched until it is in the correct position.

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

Compare 5 and -21 (no swap)

Compare 7 and 5 (no swap)

Compare -10 and 7(swap), -10 and 5 (swap), -10 and –21 (no swap)

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

Compare 61 and 7 (no swap)

Compare 8 and 61 (swap), 8 and 7 (no swap)

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

Compare 3 and 61 (swap), compare 3 and 8 (swap), compare 3 and 7 (swap), compare 3 and 5 (swap), compare 3 and –10 (no swap)

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

Compare 10 and 61 (no swap), compare 10 and 8 (no swap)

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

Final sorted list: **[-21, -10, 3, 5, 7, 8, 10, 61]**

Question 3: Show the step-by-step process of sorting a list using the quicksort sort algorithm. The unsorted list is: [-5, 4, 2, 619, 11, 5, 620, -3]

To sort the list **[-5, 4, 2, 619, 11, 5, 620, -3]** using the quicksort algorithm, a pivot needs to be chosen to partition the list. For this example, the pivot will be the median-of-three. This means that it is the median value of the left-most, right-most, and center-most [(left + right) / 2] indices. The left-most value is –5 (index 0), the right-most value is –3 (index 7), and the center value is 619 (index 3). The median value of –5, -3, and 619 is –3; so, -3 (index 7) will be our pivot.

First, we swap the pivot with the last element.

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

The variables i and j keep track of what values we are comparing. i starts at the far left element (index 0: -5), and j starts at the far-right (index 6: 620) (excluding the pivot). i increments to the right (increases index) until a value greater than the pivot is found. j decrements (decreases index) until a value smaller than the pivot is found. If the value for j is less than the value for i, the values swap. When the index for j is less than the index for i, the value for the pivot and i swap.

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

Above, we have incremented and decremented i and j, respectively, using the quicksort rules above. Because the index of j is less than the index of i, i and the pivot swap.

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

Now the list is partitioned, and now the low end (values to the left of pivot) and high end (values to the right of pivot) portion of the list are sorted recursively using the same pattern.

LOW END (left): already sorted

HIGH END (right):

[-5, -3, **2, 619, 11, 5, 620, 4**]

Left (index 2) = 2, Right (index 7) = 4, Center (index 4) = 11 | median-of-three = 4(pivot)

Pivot and far right value the same (no change).

Increment and decrement until **i** and **j** are found:

[-5, -3, **2, 619, 11, 5, 620, 4**]

The index of j is less than the index of i. Swap values at i and pivot.

[-5, -3, **2, 4, 11, 5, 620, 619**]

LOW END: already sorted

HIGH END:

[-5, -3, 2, 4, **11, 5, 620, 619**]

Left (index 4) = 11, Right (index 7) = 619, Center (index 5) = 5| median-of-three = 11(pivot)

Swap pivot and far right value:

[-5, -3, 2, 4, **619, 5, 620, 11**]

Increment and decrement until i and j are found:

[-5, -3, 2, 4, **619, 5, 620, 11**]

Index i < j, swap values of i and j:

[-5, -3, 2, 4, **5, 619, 620, 11**]

Increment and decrement until i and j are found:

[-5, -3, 2, 4, **5, 619, 620, 11**]

Index i > j, swap pivot and value of i:

[-5, -3, 2, 4, **5, 11, 620, 619**]

LOW END: already sorted

HIGH END:

[-5, -3, 2, 4, 5, 11, **620**, **619**]

Left (index 6) = 620, Right (index 7) = 619, Center (index 6) = 620| median-of-three = 620

[-5, -3, 2, 4, 5, 11, **620, 619**]

Swap pivot and far right value (sorted):

[-5, -3, 2, 4, 5, 11, **619, 620**]

Final sorted array: **[-5, -3, 2, 4, 5, 11, 619, 620].**

Question 4: Show the step-by-step process of sorting a list using the shell sort algorithm. The unsorted list is: [5, 10, 60, 0, -1, 34, 6, 10]

The shell sort algorithm uses gaps, to sort sections of the lists that are "gap" distance apart. Usually, the first gap is n/2, where n is the length of the list. Then, the gap gets smaller (usually halved), until the gap is 1 (a shell sort of gap size 1 is the same as insertion sort). To sort the list [5, 10, 60, 0, -1, 34, 6, 10], we begin by choosing a gap of 8/2 = 4. The 4 comparisons that we sort are:

[5, -1] (swap), [10, 34] (no swap), [60, 6] (swap), and [0, 10] (no swap)

After the necessary swaps, the list is now:

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

The gap is now 4/2 = 2. Comparisons we make are

[-1, 6, 5, 60] and [10, 0, 34, 10]

With insertion sort (as seen above) the sorted lists are

[-10, 5, 6, 60] and [0, 10, 10, 34]

The interleaved list is

[-10, 0, 5, 10, 6, 10, 60, 34]

The gap is now 2/2 = 1. Because we have a gap value of 1, we use insertion sort for

[-10, 0, 5, 10, 6, 10, 60, 34]

The final sorted list is [-10, 0, 5, 6, 10, 10, 34, 60].

Question 5: Rank the six sorting algorithms in order of how you expect their runtimes to compare (fastest to slowest). Your ranking should be based on the asymptotic analysis of the algorithms. You may predict a tie if you think that multiple algorithms have the same runtime. Provide a brief justification for your ranking.

For my ranking, I will be referencing the worst case scenario for each algorithm (that was provided in the slide shows)

Fastest - **Merge Sort** worst case (O(NlogN)), best case (O(NlogN))

Out of all the algorithms, merge sort has its best case AND worst cases cenario time complexity being logarithmic, which has the fastest growth rate, behind constant. This means that merge sort would be expected to have the fastest runtime.

Quick Sort worst case (O(N^2)), best case (O(NlogN))

While quick sort has a quadratic worst case scenario, it only occurs when the pivot chosen results in an empty partition - this would result in many recursive calls (actually the max amount). However, the choice of pivot can be strategic to avoid this, and can lead to the time complexity being faster than O(N^2), and, instead, closer to O(NlogN) (closest to merge sort).

Shell Sort worst case (O(N^2)), best case (O(Nlog^2N))

In its worst case, shell sort is the exact same as insertion sort. However, that is with a bad gap choice. If an efficient gap is chosen for shell sort, it will be much faster than insertion sort and have a time complexity of O(Nlog^2N), which is faster than quadratic, but slower than logarithmic.

For both quick sort and shell sort, the algorithm's time complexity heavily relies on the choice of pivots or gaps. This means that there is a sense of control for the time complexity (when you know what the list is). When random, while control is lost, choosing the most optimal gap and

pivot choice for most scenarios helps these algorithmsm be ranked higher than the other algorithms [the best case is more common for these algorithms, then the best case being getting lucky and having a nearly sorted list]. These top 3 algorithms have a focus on dividing the list to help sort, while the following 3 have a focus on comparison (making them much slower)

Insertion Sort worst case (O(N^2)), best case (O(N))

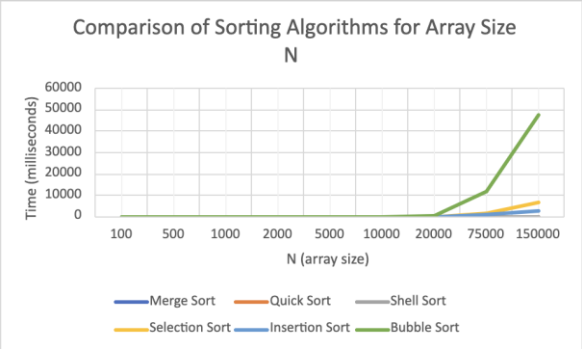
Bubble Sort worst case (O(N^2)), best case (O(N))

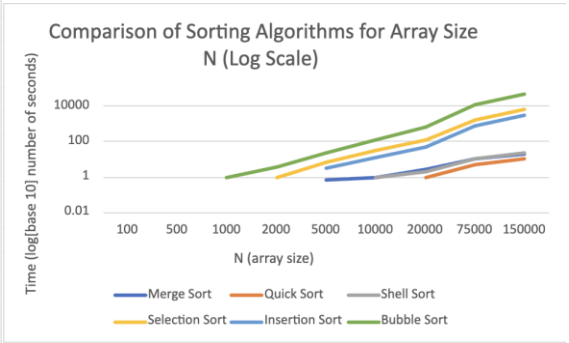
Insertion sort and bubble sort could be tied because they have the same asymptotic complexities. They both use an element and compare until the correct position is found. I could see insertion sort being faster because it has a sorted and unsorted portion, so it doesn't need to compare the before the element is in the correct position, while bubble sort does make the unnecessary check (to the elements next to the target element) to ensure the element is in the right place. But overall, I think they would be pretty similar.

Slowest - **Selection Sort** worst case (O(N^2)), best case (O(N^2))

No matter what, selection sort always has a quadratic time complexity. This is the slowest of all the other algorithms. In any case, the algorithm looks through the entire list, regardless of how sorted it is. Thus, making it the slowest algorithm in the ranking.

9 (text) [7 points] Make a graph comparing the runtimes of the six algorithms. Your x-axis should be N (the number of elements you are sorting) and your y-axis should be the average time to sort an array of size N, in milliseconds. You may create this graph using Excel or any other tool of your choice. However, making the chart directly from your java code will award you extra credit. Note that: You will find an issue when trying to plot very large numbers (This is on purpose). Once you get this issue, explain what it means and try to fix it to the best of your ability





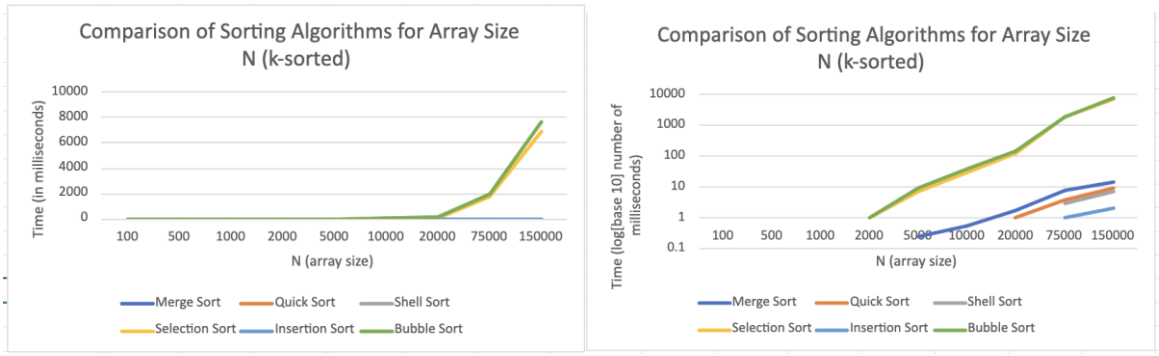
The large numbers makes it hard to see the relationship between array size and time, relative to each sorting algorithm, because it stretches out the y axis so large (due to the algorithms with higher time complexity – these are not realistic to use for such high N values). In an attempt to fix the problem, I tried to use a logarithmic-based y-axis to better scale the time. I was running into issues because I had values of 0 for my time, which does not exist for logs. Instead, maybe having a maximum time value would work better; however, that would not capture all N values for each algorithm.

10 (text) [7 points] Write a short summary of what you observed in your experiments. What did you notice about the relative performance of the different algorithms? Did the actual performance always match up to the ranking you predicted on question 6 based on the asymptotic analysis? Why or why not?

For the analysis of the results, I will be focusing on the time it took for the algorithm to sort a list of size 150000. The rankings (from fastest to slowest) were Quick Sort, Merge Sort, Shell Sort, Insertion sort, Selection sort, and Bubble Sort. This did not exactly match up with my predictions. There was a clear distinction between Quick, Merge, and Shell Sort vs. Insertion, selection, and bubble sort (as stated above, relating to dividing vs. comparing), which is reflected in both the asymptotic complexities and the experiment results. However, Merge and Quick Sort were different from my predictions; this could be due to other factors that are not accounted for by asymptotic complexity. For example, perhaps the creation of a new array led to merge sort being slower. Additionally, bubble sort was significantly slower than the other algorithms. This could be due to the repeated checking of adjacent elements to ensure elements are in the correct position.

12 (text) [5 points] Make a graph comparing the runtimes of the six algorithms.

Write a short summary of what you observed in your experiments. Did the algorithms have the same ranking on 10-sorted data as on random data? Which algorithm(s) performed significantly differently? Why? Note that: You will find the same issue as in problem 9. Once you get this issue, explain what it means.



For the k-sorted lists, the ranking (for N = 150000) is insertion sort, shell sort, quick sort, merge sort, selection sort, and bubble sort. The k-sorted lists made insertion sort and shell sort jump to the fastest algorithms. Insertion sort being the fastest makes sense because insertion sort has time complexity of O(N) when the list is nearly/already sorted (so less swaps are necessary). Seeing as shell sort seems to be a variant of insertion sort, it makes sense that it performs similarly with k-sorted lists. The partially sorted lists allow for elements to be in the correct positions more quickly. The values of merge sort and quick sort were relatively similar to the non-k sorted list, but was still faster. Because these algorithms don't considerably depend on the sortedness of the list, the algorithm behaves similarly to the unsorted list. Selection sort and bubble sort are again, not suitable for high N values and have the slowest time. However, selection sort performed the same, while bubble sort performed significantly better. This is because selection sort will always do the same amount of comparisons, while bubble sort can take advantage of the partially sorted list.

Running into the similar issue as the last tests, the algorithms with greater time complexity had very large time values that made it difficult to display the relationship between time and N value for the algorithms (even with the log based axis). Again, this means that the algorithms with such high time complexities are not suitable for larger lists.