Project 4: Sorting Algorithms Report and Analysis

Baines Blanton, Caleb Cort

University of Georgia

CSCI 2720: Data Structures

Professor Jack Bass

20 November 2025

Introduction

In this report, we demonstrate the efficiency of Selection Sort, Quick Sort (Quick Sort Last, Quick Sort Random, Quick Sort Iterative), Merge Sort, and Heap Sort. Our experiments come with visual data to show how fast each algorithm performs under different types of inputs and input sizes as we test their effect on each algorithm's time complexity. The results tell us which sorting algorithms are faster holistically, as well as which ones are preferable for certain kinds of data.

Description of Sorting Algorithms

Selection Sort:

O(n^2) should not be used unless it is for educational purposes. This algorithm splits an array into unsorted and sorted segments. Every time it finds the smallest integer in the unsorted segment, it puts that integer at the end of the sorted segment, thereby increasing the sorted section. This process is repeated recursively until the whole array is a sorted segment.

Comparisons = N * (N – 1) / 2

Quick Sort Last:

O(nlogn), chooses an integer, the last integer in an array, as a pivot and partitions the array so that all elements left of the pivot will be smaller than the pivot, and every element that is greater is moved to the right of it. This repeats recursively on each segment of the array until it's fully sorted.

Quick Sort Random:

O(nlogn), same as Quick Sort Last, except the pivot is a random index within the array. The reason is that randomness reduces the chances of the worst-case time complexity from occurring, which would be O(n^2). At that point, we might as well use Selection Sort.

Iterative Quick Sort:

O(nlogn), Performs quick sort using an iterative loop. This way of doing it produces the same number of comparisons and time complexity as Quick Sort Last, but this algorithm demonstrates that what changes the time complexity is the pivot integer, not whether it was implemented recursively or iteratively

Merge Sort:

O(nlogn), known for its consistent time complexity; good for large databases that need reliable sorting. This algorithm divides the array into segments recursively until they are individual elements. Then it sorts and merges those elements back until it's one sorted array.

Comparisons = $T(n) = 2T(n/2) + n - 1$ (for $n > 1$) | $T(1) = 0$

Heap Sort:

O(nlogn), builds the max heap from an array so that every parent is greater than its children. Then, it recursively swaps the max element (root) with the last element of the unsorted segment, heapifying to maintain the max root property. Opposite selection sort, this algorithm puts the largest elements in their spot (on the right end) first and will end with the first index having the smallest integer.

Experiment 1: Procedure

Experiment 1's objective was to compare the number of comparisons used in the different sorting algorithms across three provided text files. The three files contained an ordered list, a reversed list, and a random list of 10,000 elements. After implementing each of the five

algorithms, we built a driver file to take the input of the selected text file and desired sorting algorithm then returning the number of comparisons for that pair.

Experiment 1: Results (and Tables)

After running each of the tests on the algorithm/input pairs, the results were recorded in the chart below.

Algorithm	Input Type	Comparisons	Time Complexity
Selection	Ordered	49995000	O(n^2)
Selection	Reverse	49995000	O(n^2)
Selection	Random	49995000	O(n^2)
Merge	Ordered	69008	O(nlogn)
Merge	Reverse	64608	O(nlogn)
Merge	Random	120414	O(nlogn)
Heap	Ordered	244460	O(nlogn)
Heap	Reverse	226682	O(nlogn)
Heap	Random	235430	O(nlogn)
Quick Sort Last	Ordered	49995000	O(n^2)
Quick Sort Last	Reverse	49995000	O(n^2)
Quick Sort Last	Random	148343	O(nlogn)
Quick Sort Random	Ordered	162042	O(nlogn)
Quick Sort Random	Reverse	64608	O(nlogn)
Quick Sort Random	Random	120414	O(nlogn)
Quick Sort Iterative	Ordered	49995000	O(n^2)
Quick Sort Iterative	Reverse	49995000	O(n^2)
Quick Sort Iterative	Random	148343	O(nlogn)

With these results, we can see the behavior of each algorithm. Selection sort always behaves in O(n^2) regardless of the ordering of the items. Both Merge Sort and Heap Sort are mostly unaffected by the ordering of the items as they stay within O(nlogn). Quick Sort Last, both recursive and iterative) produces the worst-case behavior in both the Ordered and Reverse lists as the last item in both cases are bad pivot points thereby producing O(n^2). Utilizing the Random

file prevents the poor pivot points of the other cases producing O(nlogn)). Quick Sort Random avods the worst-case poor pivots of the previous algorithm by selecting a random pivot point leading to O(nlogn) for all three files. Overall, all the algorithms besides Selection sort all perform similarly in their average case.

Experiment 1: Additional Questions:

i. For selection sort, the number of comparisons is exactly the same for ordered, reverse, and random input files (49,995,000 comparisons). This aligns with the theoretical action of the algorithm as the algorithm always scans the entire array to find the lowest element, regardless of input order, so the number of comparisons does not change. Therefore, all three input types are expected to produce the same number of comparisons, and random input should not be higher or lower than sorted inputs.

ii. In merge sort, the number of comparisons for the random input file is much higher than for the ordered and reverse files. This is because merge sort's comparison count depends on how the two halves interact during merging. With ordered or reverse data, one half tends to empty quickly, which simplifies the merge process and reduces comparisons. With random data, the values from each half are more interconnected, causing merge sort to compare many more pairs of elements before either half runs out. This leads to almost double the number of comparisons for the random input file.

iii. In the heap sort experiment, the number of comparisons for all three input types is very similar. This is because heap sort depends on how the heap is built, not on how the elements are ordered. The heapify process follows the same steps regardless of input pattern, so the comparison counts do not changelike they do in merge sort or quick sort. The ordered input file has slightly more comparisons than the reversed one because

ordered data places smaller values near the root of the heap, causing many violations of the heap property. These violations force elements to sift down more frequently, increasing the number of comparisons. In the reversed input, larger values start near the root, resulting in fewer violations and fewer comparisons overall.

Experiment 2: Procedure

Experiment 2 sought to discover if the growth relationship between input size and number of comparisons matched the theoretical Big O time complexity of each of the algorithms. To find these trends, we tested the different algorithms against the following input sizes: 100, 500, 1000, 5000, 10000, 20000, 25000, 30000, 50000, and 100000. In order to produce a valid average of these number of comparison results, we tested each algorithm-input size pair 30 times to set the value of n = 30. This sample size allows us to generalize these results to the population of all trials of that pair. A new array of randomly generated integers of the selected input size was created for each trial to ensure random inputs. After collecting all the data, we compiled the averaged results into a scatterplot to display the different efficiencies of each algorithm.

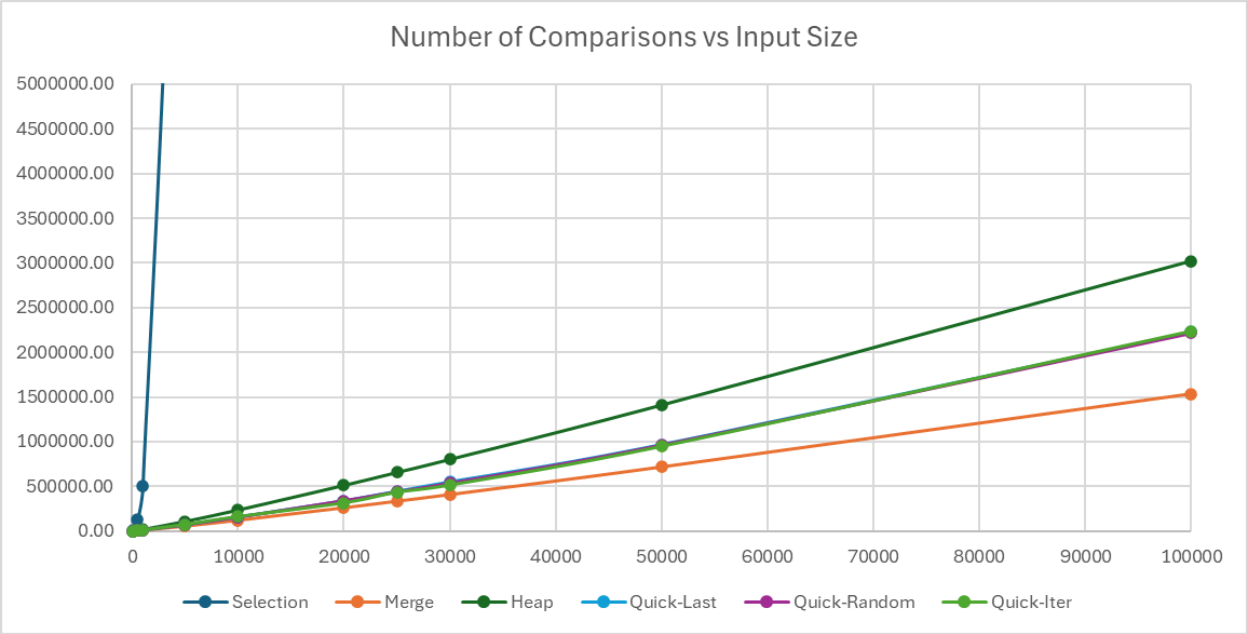
Experiment 2: Results (and Graphs)

The results of our Experiment 2 show how the number of comparisons growth in relation to input size. Below is the table containing our average number of comparisons for the 30 trials of each of the algorithm-input pairs.

Experiment 2	Selection	Merge	Heap	Quick-Last	Quick-Random	Quick-Iter
100	4950.00	542.47	1028.97	654.97	644.40	605.00
500	124750.00	3856.80	7421.83	4814.33	4765.23	4622.00
1000	499500.00	8705.57	16838.03	10813.80	10921.00	11101.00
5000	12497500.00	55213.87	107693.33	70769.60	71348.77	74056.00
10000	49995000.00	120443.57	235377.97	156619.33	155588.27	161478.00
20000	199990000.00	260860.87	510775.93	339797.43	340386.20	316313.00
25000	312487500.00	334110.87	654860.83	444969.00	439214.30	434962.00
30000	449985000.00	408589.80	800649.93	550358.77	536795.40	514682.00

50000	1249975000.00	718163.00	1409811.87	968091.83	963600.17	947487.00
100000	4999950000.00	1536365.43	3019668.50	2225763.50	2218505.87	2233718.00

After collecting this data, we built a line plot with the data from the table separated by algorithm type seen below.



As can be seen in the plot, each of the algorithms outside of Selection Sort demonstrates an nlog(n) time complexity relationship with input size. Each of the Quick Sort variations lies very close to each other, including Quick Sort Last demonstrating that when the arrays are randomized, the last item being the pivot point does not hurt the sorting speed. Selection Sort, an algorithm that always compares every item to every other item, demonstrates n^2 time complexity. Overall, the randomized nature of arrays allows for nlog(n) algorithms to perform very close to their theoretical time complexities.

Analysis (Compare results to theory)

i. Theoretically, the order of comparisons from least to greatest should go ordered, random, reverse because with reverse it is known everything will have to be moved, but with random there's a chance that some integers will be in the right place.

However, with selection sort, the number of comparisons is the same, because it's programmed to scan through the whole array no matter what to keep track of the smallest value, so it compares everything no matter what.

ii. Merge Sort's number of comparisons depends on how fast one side of the array empties during the merge. Two sorted halves merge by having their respective first elements compared, and the smaller one is removed. Ordered and reverse arrays will always have one side be bigger and one smaller, but random arrays do not have that pattern, so the algorithm is more likely to have to compare every integer.

iii. The number of comparisons is almost the same for all three input types because heap sort only looks for parent-child relationships during heapify and building the heap, while merge sort and quick sort are concerned with the overall ordering of the input. Heap sort depends on the structure of the heap rather than the array's original order. Ordered input requires a few more comparisons than the other two because the smallest integers are at the top of the heap, meaning just about every number violates the heap property, and so one node has to heapify down more.

Conclusion

These experiments have shown that empirical performance reflects theoretical expectations for the majority of the sorting algorithms we covered. The worst performance went to Selection Sort by far with a time complexity of O(n^2). The order of input had no effect since it compares everything indiscriminately. Heap and Merge Sort had O(nlogn) time complexities, with only some amount of variation because of how their heap and merge functions handle data patterns. Quick Sort's time depended the most on pivot selection, which could make the difference between an O(nlogn) time for a random pivot and an O(n^2) time for a last element

pivot. The findings of our study showcased the significance of algorithmic design choice and the difference that input characteristics can have on their sorting time. In the real world, selection sort should be avoided, while heap sort, merge sort, and quick sort carry software with their reliability and scalability.