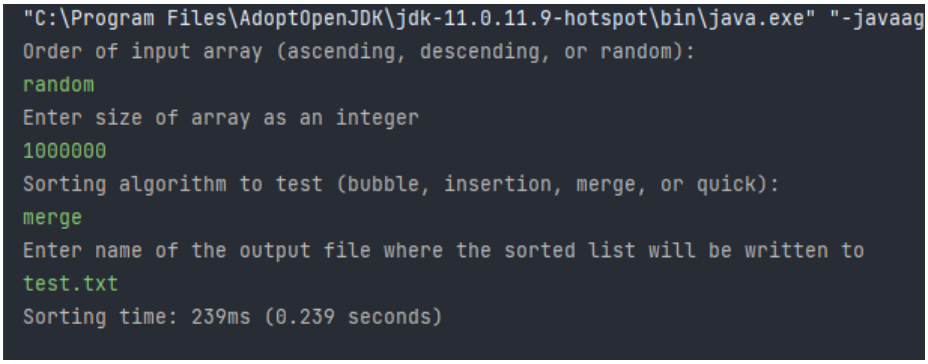
ASSIGNMENT 2

Comparison of Sorting Algorithms

Graydon Hall (30142310)
gwhall@ualberta.ca

Screenshot of User Interface

A sample screenshot of using this terminal-based application is provided as a reference.



Data Collection

For this lab, the sorting algorithms analyzed were Bubble, Insertion, Merge, and Quick Sort. These algorithms were each used to sort arrays of 10, 100, 1,000, 10,000, 100,000, and 1,000,000 arrays filled with random digits. Trials were done with the input arrays beginning in ascending, descending, and random order. The results are summarized in the following table.

Table 1 Time Required to Sort Input Arrays, while varying Input Array Order and Size

Time required to sort array (ms)								
Input Array Initial Sorting	Sort Type	Rank (based on input size = 1,000,000)	Input Array Size					
			10	100	1000	10000	100000	1000000
Ascending	Insertion	1	0	0	0	1	4	5
	Bubble	4	0	0	4	24	2447	205601
	Merge	2	0	0	1	31	29	115
	Quick	3	0	0	4	39	2165	185666
Random	Insertion	3	0	0	3	25	1527	117941
	Bubble	4	0	0	1	118	13376	2504571
	Merge	2	0	0	0	1	20	228
	Quick	1	0	0	0	0	7	81
Descending	Insertion	2	0	0	3	22	3128	229828
	Bubble	4	0	0	3	87	7766	775125
	Merge	1	0	0	0	3	12	189
	Quick	3	0	0	1	35	2973	307747

These results are then summarized in the following graphs.

Complexity Analysis

Bubble Sort

- Number of comparisons:
  - O(n^2) no matter what
- Number of swaps:
  - Best case happens when array is already sorted, has 0 swaps
  - Worst and Average case are both O(n^2) though

Insertion Sort

- Best case occurs when array is already sorted
  - Comparisons: O(n)
  - Data moves: O(n)
- Worst and average case:
  - Comparisons: O(n^2)
  - Data moves: O(n^2)

Merge Sort

- Number of comparisons is O(n lg n) regardless of initial order of data
- It has a space complexity of O(n), because it needs to create a temporary array.

Quick Sort

- Worst case occurs when the largest or the smallest element is always chosen for the pivot
  - Has a complexity of O(n^2) in that case
- However, as long as the data is not already sorted, the average case for Quick Sort is O(n lg n)
- Quick sort generally out-performs quick sort (except for in the case of nearly sorted arrays, that cause the pivot to be the largest or smallest value each time)

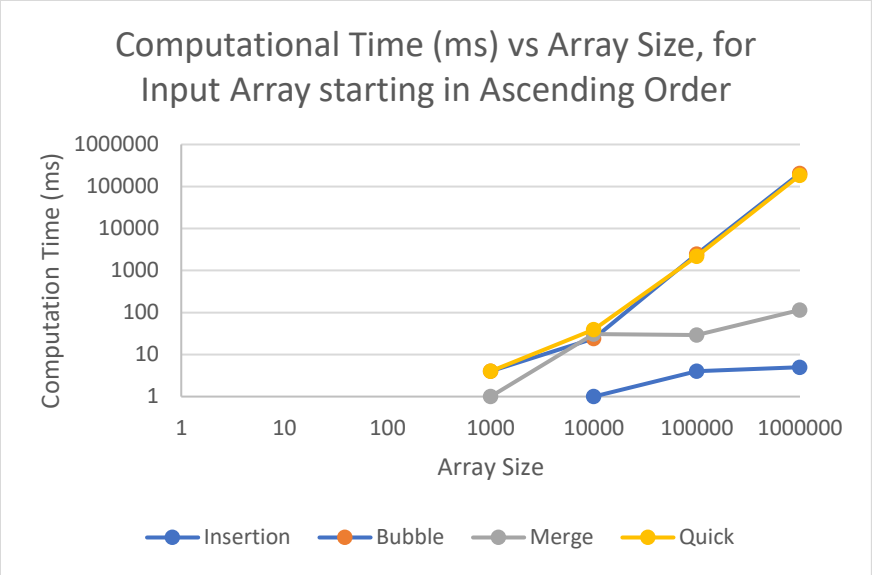


Figure 1 Computational Time (ms) vs Array Size, for Input Array starting in Ascending Order

Based on this figure, we see that when the input array is in Ascending order:

- All algorithms performed equally and took 0ms for arrays of 100 elements or less.
- Quick Sort and Bubble sort both perform significantly worse than the other two algorithms.
- Quick Sort has its worst-case complexity in this instance which is O(n^2). This is because, the worst case for quick sort occurs when the largest or smallest element is always chosen for the pivot, which happens in the case of a sorted input array.
- Despite performing poorly, this scenario actually gives the best case performance for bubble sort.
    - While the number of comparisons are constant for bubble sort, no swaps will be required, making this the best possible case for the performance of bubble sort.
- Insertion sort performed the best out of all the algorithms here. This is because based on how the insertion sort algorithm works, it will only have to complete n-1 comparisons, and 2(n-1) data moves, (see class notes). This means comparisons and data moves will both be O(n), making it the most efficient out of all the algorithms for this scenario.
- Merge sort performs second best here. Merge sort performs quite well regardless of the configuration of the input array. This is because the number of comparisons is O(n lg(n)), no matter the order of the input array.

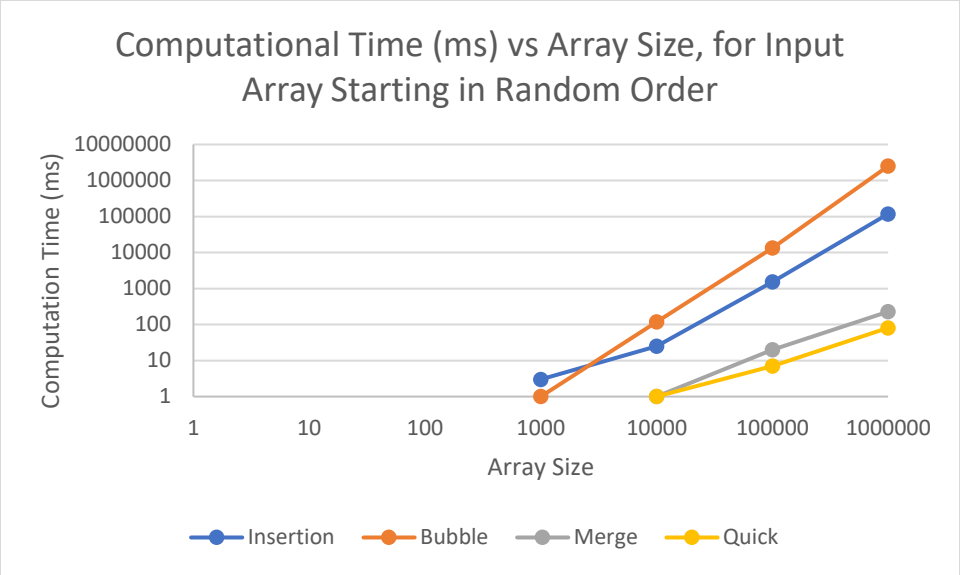


Figure 2 Computational Time (ms) vs Array Size, for Input Array Starting in Random Order

Based on this figure, we see that when the input array is in Random order:

- All algorithms performed equally and took 0ms for arrays of 100 elements or less.
- Bubble sort performs the worst out of the 4 algorithms. It has a complexity of O(n^2)
- Insertion sort performs second worst here. This is because the average case for insertion sort for both comparisons and data moves is O(n^2), therefore unless the array is already almost sorted, insertion sort generally performs poorly.
- Bubble and Insertion sort both belong to the same Big O here (O(n^2)), but insertion sort performs slightly better.
- Merge sort performs second best here. Merge sort performs quite well regardless of the configuration of the input array. This is because the number of comparisons is O(n lg(n)), no matter the order of the input array.
- Quick sort performs the best here. Quick sort is normally the quickest algorithm (see class notes). The best case for quick sort occurs when the algorithm always creates equal sized subarrays. Since the array is in random order, it is more likely to be able to do this since it always chooses the pivot as being the last element of the array, and the last element will be a random number.
- Quick sort and merge sort both belong to the same big O here (O(n lg n)), but quick sort outperforms merge sort.

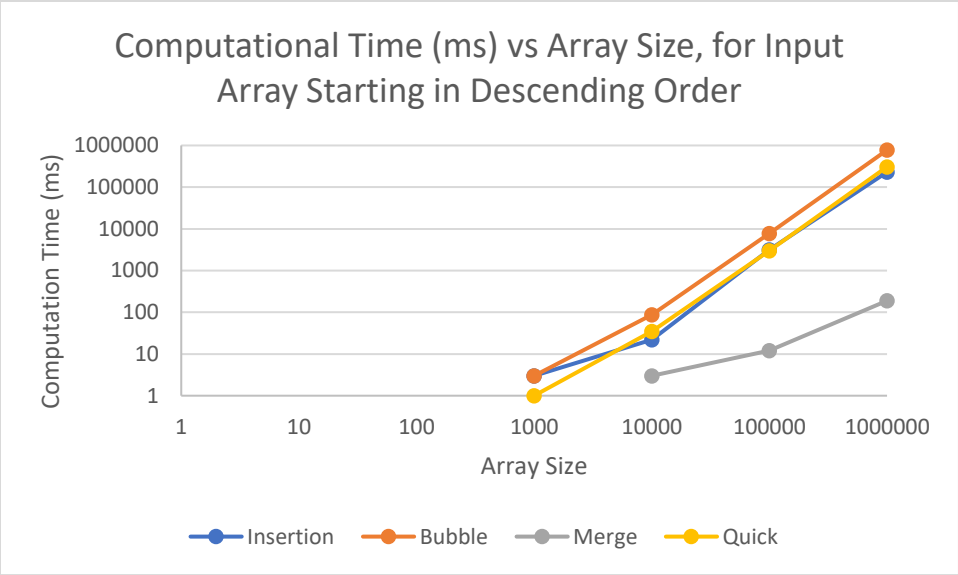


Figure 3 Computational Time (ms) vs Array Size, for Input Array Starting in Descending Order

Based on this figure, we see that when the input array is in Descending order:

- All algorithms performed equally and took 0ms for arrays of 100 elements or less.
- Quick, Bubble, and Insertion Sort all perform poorly here.
- Bubble and Insertion sort both belong to the same Big O here (O(n^2)), but insertion sort performs slightly better.
- Quick Sort actually has it's worst case performance in this instance, with a complexity of O(n^2). This is because, the worst case for quick sort occurs when the largest or smallest element is always chosen for the pivot.
    - Since the algorithm functions by choosing the last element of the array as a pivot, in this case, the smallest value will always be taken as the pivot, and a worst case complexity of O(n^2) will occur.
- Bubble and Insertion sort also both face their worst case complexities of O(n^2). They will both have to make the maximum number of comparisons and swaps required for each algorithm, causing them to perform very poorly.
- Merge sort performs best here. Merge sort performs quite well regardless of the configuration of the input array. This is because the number of comparisons is O(n lg(n)), no matter the order of the input array.

Conclusions

- If you have an array that is almost already sorted, Insertion sort will perform best. However, it is risky to use Insertion sort, because if the array is not sorted or almost sorted, it has a complexity of O(n^2) and will perform very badly.
- Merge sort is the safest to use. It will perform with excellent efficiency regardless of the order of the input array.
- Quick sort is the fastest algorithm for if the data is in random order. However, the worst case for quick sort occurs when the pivot selected is always the largest of the smallest value. In this implementation of quicksort, since the pivot selected is always the last element of the array, if the array is in ascending or descending order, the algorithm will perform very poorly.
- Bubble sort should never be used unless it is for an array with not many elements.
    - If it is a small array, and you want to be able to code the sorting algorithm by yourself for some reason, bubble sort may actually be good, as it is very easy to implement on your own.
    - However, if there's any chance your array will grow in size later down the line, this sorting algorithm scales very poorly.
- To avoid stack overflow occurring, I avoided using recursive solutions, and used iterative implementations of the algorithms instead.
- If you array is 100 elements or less, all algorithms perform equally, so the choice of algorithm you use may be arbitrary.