Lab 11 Sorting

Name: _____Tanadon Santisan_____

ID: _____6538068821_____

No coding in this lab! :-D

The objective of this lab is to analyze time complexity of sorting algorithms by experiments.

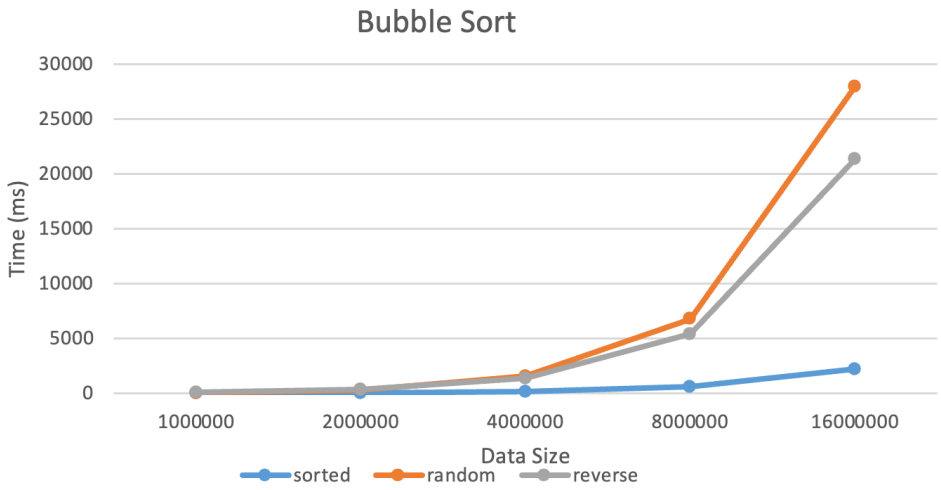
The provided program contains all sorting algorithms. The code in the program may look a little different from the code in lecture slides but they uses the same principles.

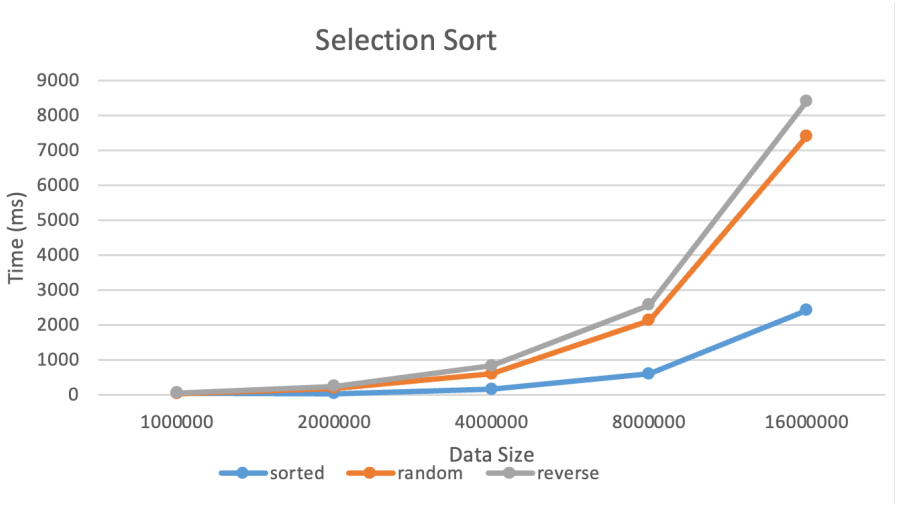
To help you understand how each algorithm works:

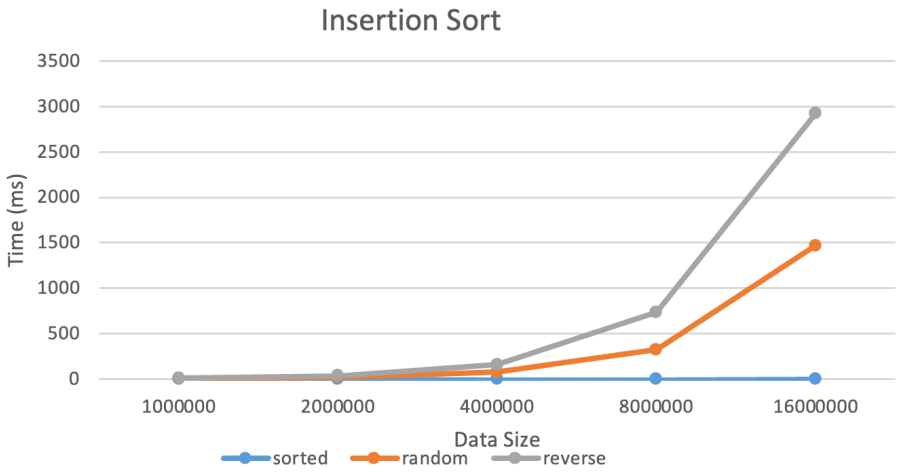
- Watch this video http://img-9gag-fun.9cache.com/photo/aPyoG4P_460sv_v1.mp4

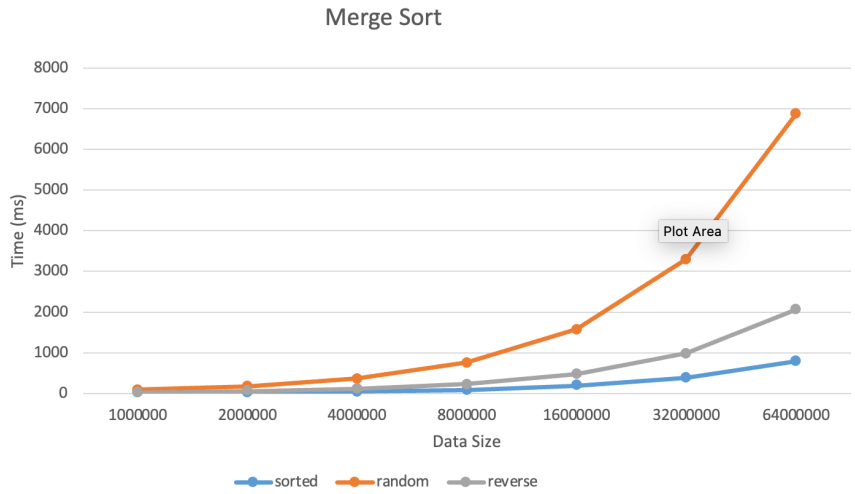
Follow the instructions and answer question 5-8:

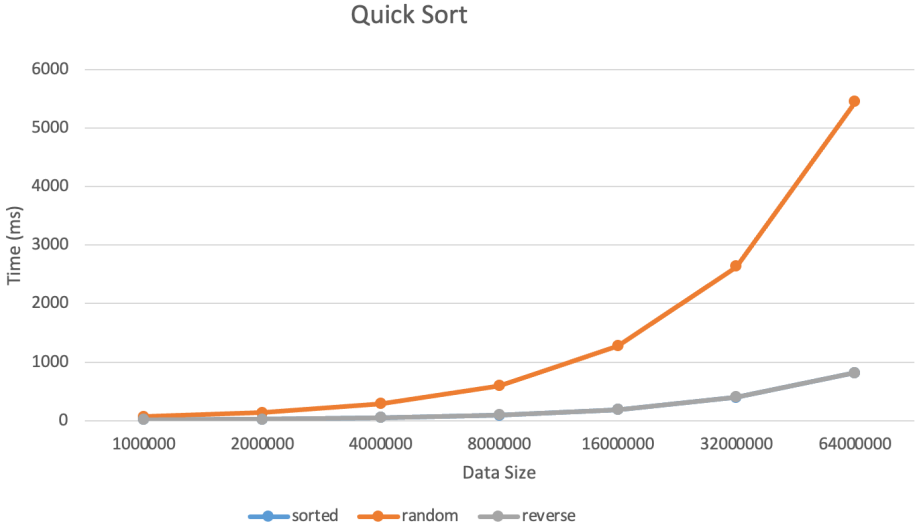
1. Select an algorithm from the list in line 34-40 of the given code by putting // in front of other algorithms. The algorithms to be used in this lab are bubble sort, selection sort, insertion sort, merge sort and quicksort.
2. Select one of the for loops in line 15 and 16 based on the selected algorithm.
3. Run the Sorting program. Do not run other applications while the Sorting program is running.
4. The program will create an array of size n, populate the array with data, sort the array, check the results, and print out the execution time for sorting. It will vary the size of the array and also vary the initial order of data (sorted, random, and reversed order).
5. For each algorithm and each initial order, create a line graph between data size and execution time. There will be 15 lines in total but you can put the graphs of the same algorithm in one plot. You can copy the output from Eclipse into the Excel file to create graphs.




Selection Sort
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
Time (ms)
1000000
2000000
4000000
8000000
16000000
Data Size
sorted
random
reverse


Insertion Sort
3500
3000
2500
2000
1500
1000
500
0
Time (ms)
1000000
2000000
4000000
8000000
16000000
Data Size
sorted
random
reverse


Merge Sort
8000
7000
6000
5000
4000
3000
2000
1000
0
Time (ms)
Plot Area
1000000
2000000
4000000
8000000
16000000
32000000
64000000
Data Size
sorted
random
reverse


Quick Sort
6000
5000
4000
3000
2000
1000
0
Time (ms)
1000000
2000000
4000000
8000000
16000000
32000000
64000000
Data Size
sorted
random
reverse

6. Based on the experimental result, determine the time complexity of each algorithm in terms of Big O and fill in the table.

Time Complexity

	Ordered	Random	Reverse
Bubble Sort	O(n)	O(n^2)	O(n^2)
Selection Sort	O(n^2)	O(n^2)	O(n^2)
Insertion Sort	O(n)	O(n^2)	O(n^2)
Merge Sort	O(n log n)	O(n log n)	O(n log n)
Quicksort	O(n log n)	O(n log n)	O(n^2)

7. Which algorithm in each group is the fastest? What is the reason?

7.1) Bubble, Selection, Insertion

Insertion Sort is fastest

Because among these three, Insertion Sort tends to perform the best in practice, particularly for small arrays or nearly sorted data. Its main advantage comes from its ability to minimize the number of operations in scenarios where the data is already almost sorted. Insertion Sort efficiently places each new element into its proper place in the already sorted section of the array with minimal shifts, making it significantly faster than Bubble Sort and Selection Sort in these cases. While Bubble and Selection Sorts have fixed $O(n^2)$ time complexities in their best, average, and worst cases (with minor variations for Bubble Sort), Insertion Sort shines in its best case with a time complexity of $O(n)$, making it exceptionally efficient for nearly sorted or small datasets.

7.2) Merge, Quick

Quick Sort is fastest

Because Quick Sort can often outperform Merge Sort in practical scenarios, despite both having an average time complexity of $O(n \log n)$. The reason for Quick Sort's superior performance is its lower constant factors and its in-place sorting capability (not requiring additional storage proportional to the size of the input array, unlike Merge Sort). Quick Sort's efficiency comes from its ability to partition the data such that each element is placed in its final position in the sorted array, with all subsequent operations only refining this order. However, it's crucial to note that Quick Sort's performance heavily depends on the choice of the pivot. In its best and average cases, with a good pivot selection strategy, Quick Sort will generally be faster than Merge Sort due to these reasons. However, in the worst case (such as when the smallest or largest element is consistently chosen as the pivot), Quick Sort's performance degrades to $O(n^2)$, although this is rare with modern pivot selection techniques like the median-of-three method.

8. For each algorithm, how is it sensitive to the initial order of data? (Does it run much faster or slower when the data is initially sorted, random, or reversed?) Why?

   Bubble Sort:

   Sensitivity to Initial Order: Highly sensitive.
   Explanation: In its basic form, Bubble Sort performs better if the data is already sorted (or nearly sorted), operating at its best-case time complexity of $O(n)$ because it only needs to make one pass through the data to confirm it's sorted, with no swaps needed. Conversely, when the data is in reverse order, it exhibits its worst-case behavior, $O(n^2)$, as each element needs to be "bubbled" all the way to its correct position.

   Selection Sort:

   Sensitivity to Initial Order: Not sensitive.
   Explanation: Selection Sort has a time complexity of $O(n^2)$ in all cases because it always scans the remaining items to find the minimum and places it at the current position, regardless of the initial order of the array. The number of comparisons remains constant, although slight variations in swap operations might not significantly affect the overall performance.

   Insertion Sort:

   Sensitivity to Initial Order: Highly sensitive.
   Explanation: Insertion Sort benefits significantly from an initially sorted (or nearly sorted) list, achieving a best-case performance of $O(n)$ because each new element to be inserted typically doesn't have to move far. However, in the case of a reverse-ordered list, it exhibits its worst-case scenario, $O(n^2)$, as every element needs to be moved to the beginning of the array.

   Merge Sort:

   Sensitivity to Initial Order: Not sensitive.
   Explanation: Merge Sort's time complexity is $O(n\log n)$ regardless of the initial order of the data. This stability comes from its divide-and-conquer approach, which systematically divides the array into smaller pieces to sort and merge. However, the merge step might be slightly more efficient if the data is already sorted or nearly sorted, but this doesn't change the overall time complexity.

   Quicksort :

   Sensitivity to Initial Order: Sensitive, depending on pivot selection.
   Explanation: The efficiency of Quicksort greatly depends on the choice of the pivot. With a good pivot selection strategy, Quicksort performs well on random data, achieving $O(n\log n)$ on average. However, in its worst-case scenario, such as when the smallest or largest element is always chosen as the pivot (which might happen with sorted or reverse-sorted data if the pivot selection is poor), its performance degrades to $O(n^2)$. Various strategies, like choosing a random pivot or the median-of-three method, are used to mitigate this sensitivity and ensure that Quicksort remains efficient across different data orders.

9. Submit this file. Name it YourID_Lab08_Sorting, where YourID is your student ID.