Algorithms Analysis and Design

Fall 2022/2023

Assignment#3:

Empirical Analysis of Sorting Algorithms.

Yara Jehad Rabaya

202110451

Introduction:

Sorting algorithms serve as foundational tools within the realm of computer science, facilitating the efficient organization and arrangement of data.

This assignment delves into the captivating domain of sorting algorithms, aiming to explore their empirical analysis. The primary goal is to acquire a profound understanding of the efficiency and performance exhibited by various sorting algorithms. This will be achieved through the systematic conduction of experiments on datasets of varying sizes, allowing for a comprehensive comparison of their respective execution times.

Commencing our empirical analysis involves an exploration of a curated selection of popular sorting algorithms. These include, but are not limited to:

1. **Bubble Sort:** An algorithm characterized by its simplicity and intuitiveness, wherein adjacent elements are repeatedly compared and swapped if they are in the incorrect order.
2. **Insertion Sort:** A straightforward algorithm that constructs the final sorted array incrementally, inserting each element into its correct position one at a time.
3. **Selection Sort:** An algorithm that iteratively identifies the minimum element within the unsorted portion of the array, placing it at the beginning.
4. **Merge Sort:** A divide-and-conquer algorithm that bifurcates the array into two halves, independently sorts them, and subsequently merges them to yield a sorted output.
5. **Quick Sort:** Another divide-and-conquer algorithm that designates an element as a "pivot" and partitions the array around this pivot, iteratively sorting the resulting sub-arrays.

Work environment and implementation:

This experiment will be performed in a computer with following specifications:

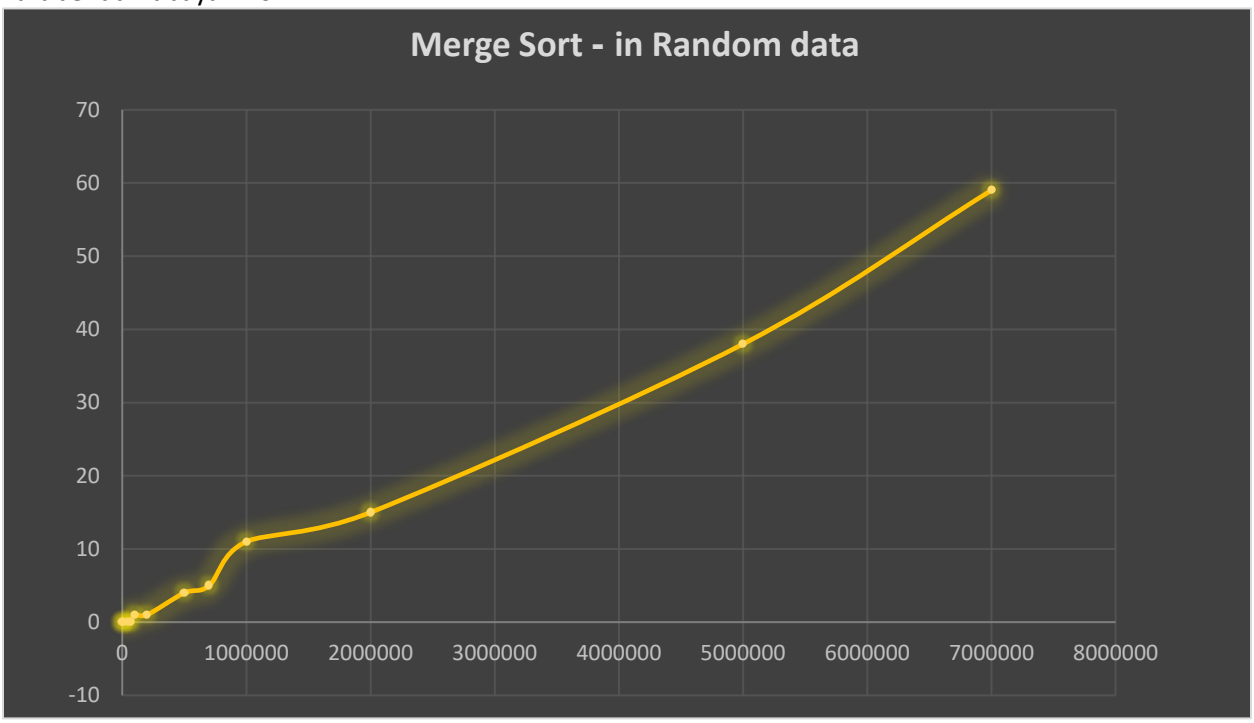
- Processor: Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz 2.11 GHz
- RAM: 8.0 GB (7.81 GB usable).
- Operating system: Windows 10 pro.
- Compiler: (Code Blocks IDE) x86_64-w64-mingw32-g++.

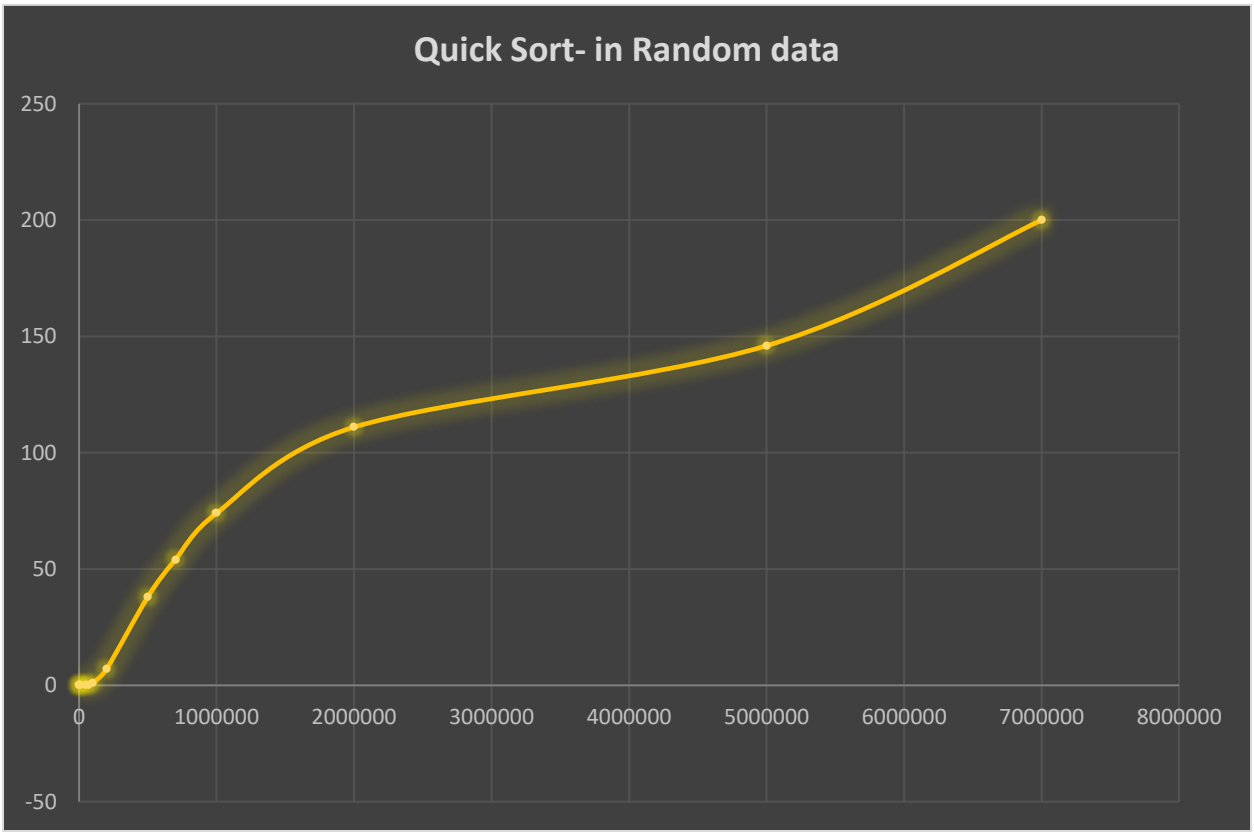
I filled a vectors of integers of different sizes, every sorting algorithm sorts the same values to make the compression fair as I can by make a five copies of the same vector.

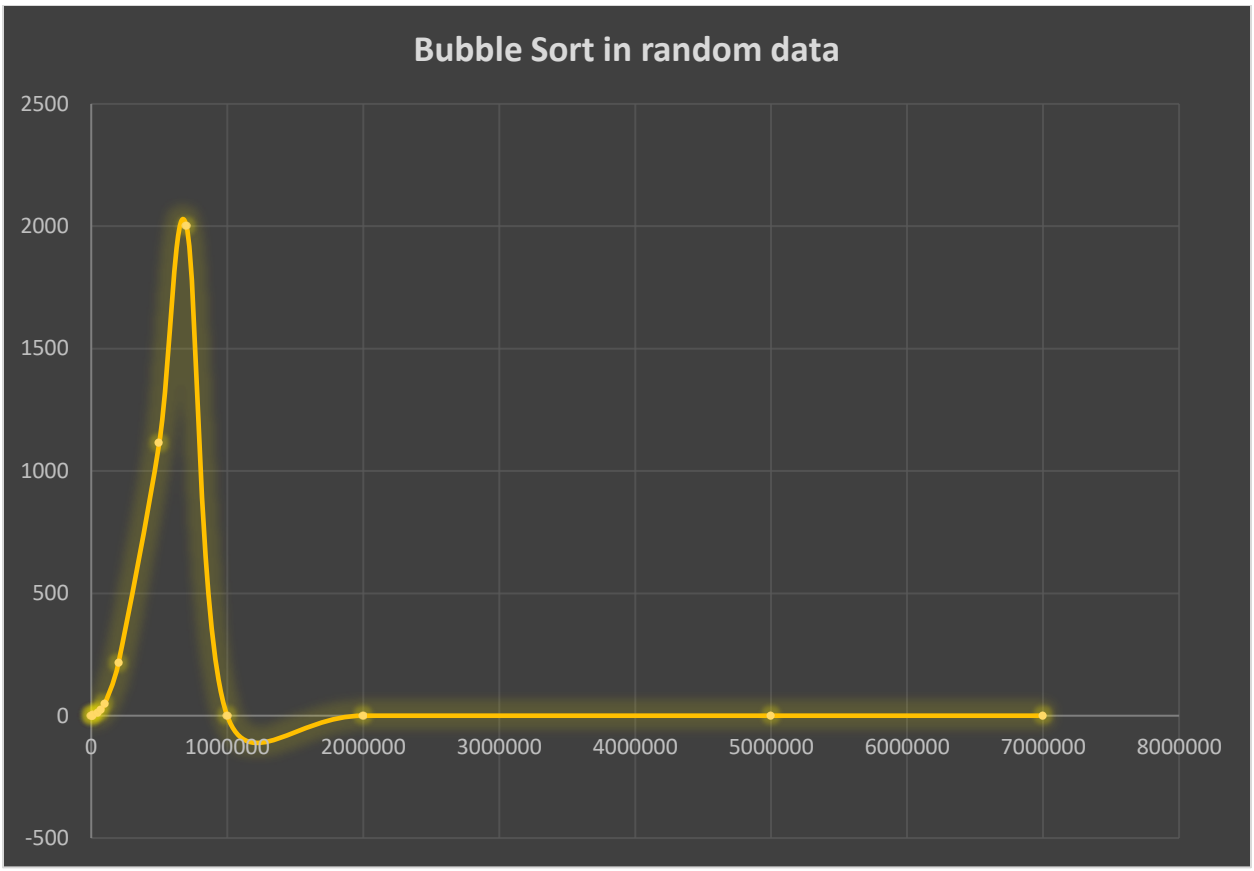
The input data divides into three sections:

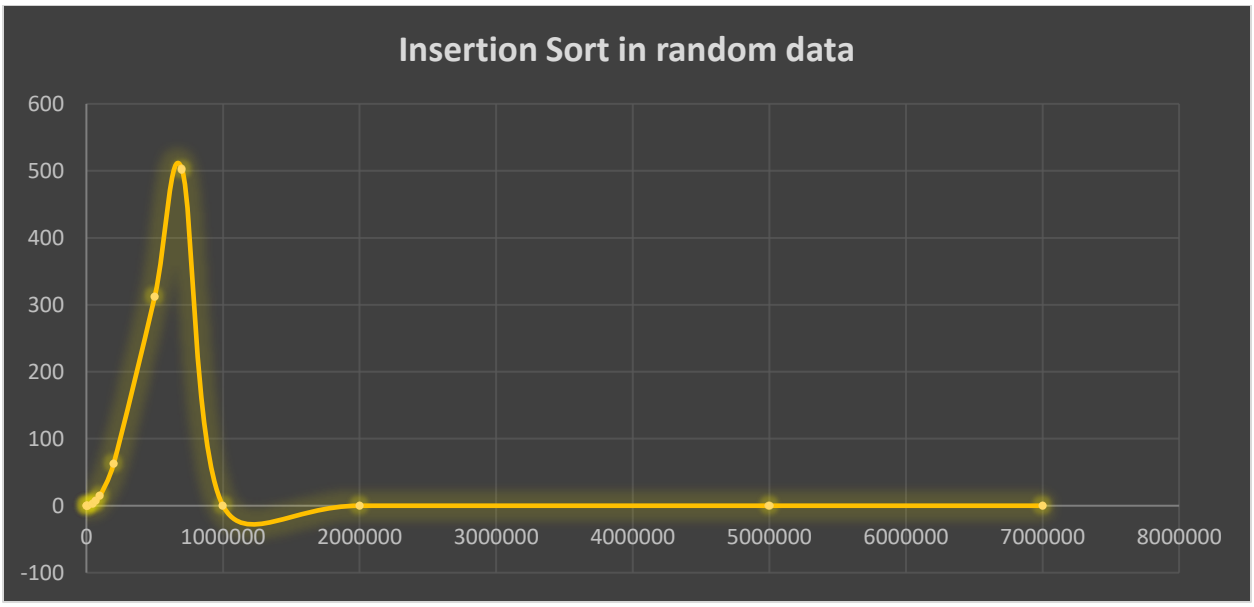
First section: Random data from 0 to 5000.

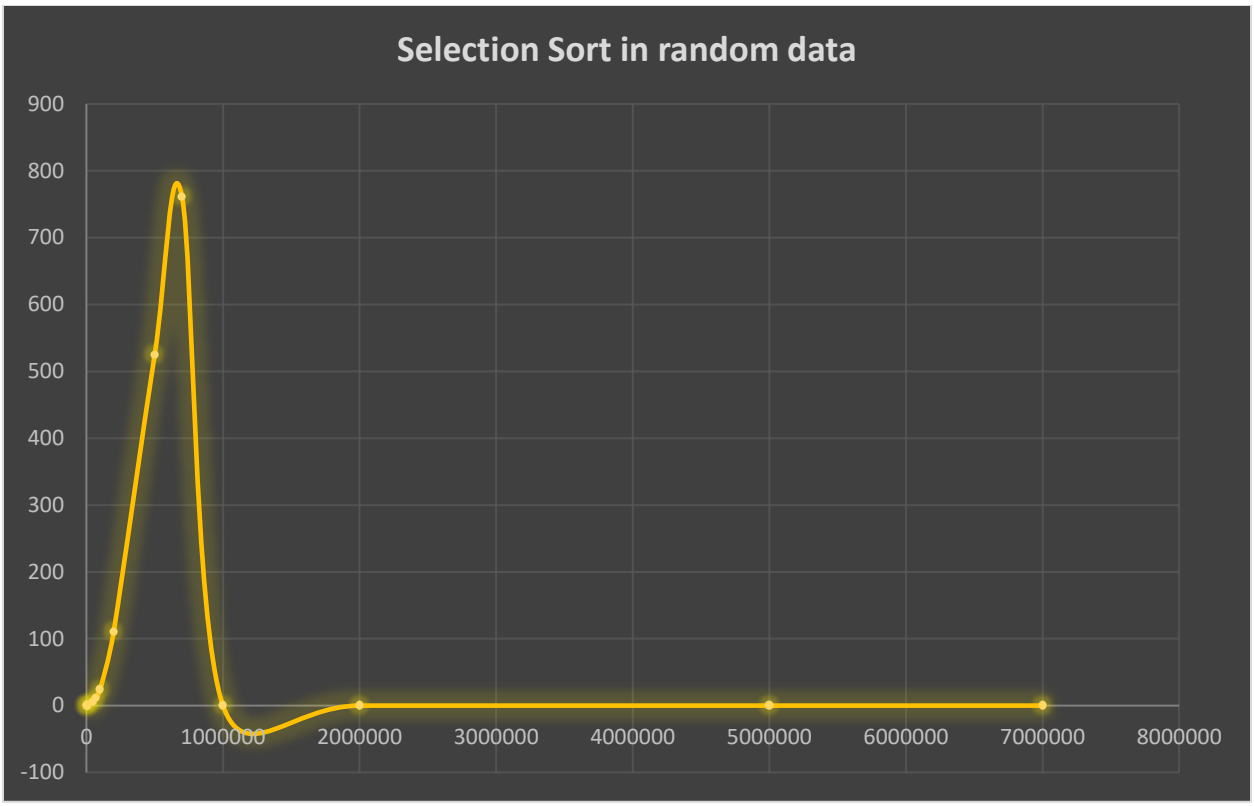
Input size	Merge Sort	Quick Sort	Bubble Sort	Insertion Sort	Selection Sort
1000	0	0	0	0	0
5000	0	0	0	0	0
10000	0	0	1	0	0
50000	0	0	12	3	5
70000	0	0	24	7	11
100000	1	1	50	15	24
200000	1	7	215	63	111
500000	4	38	1114	312	525
700000	5	54	2001	502	761
1000000	11	74	-	-	-
2000000	15	111	-	-	-
5000000	38	146	-	-	-
7000000	59	200	-	-	-

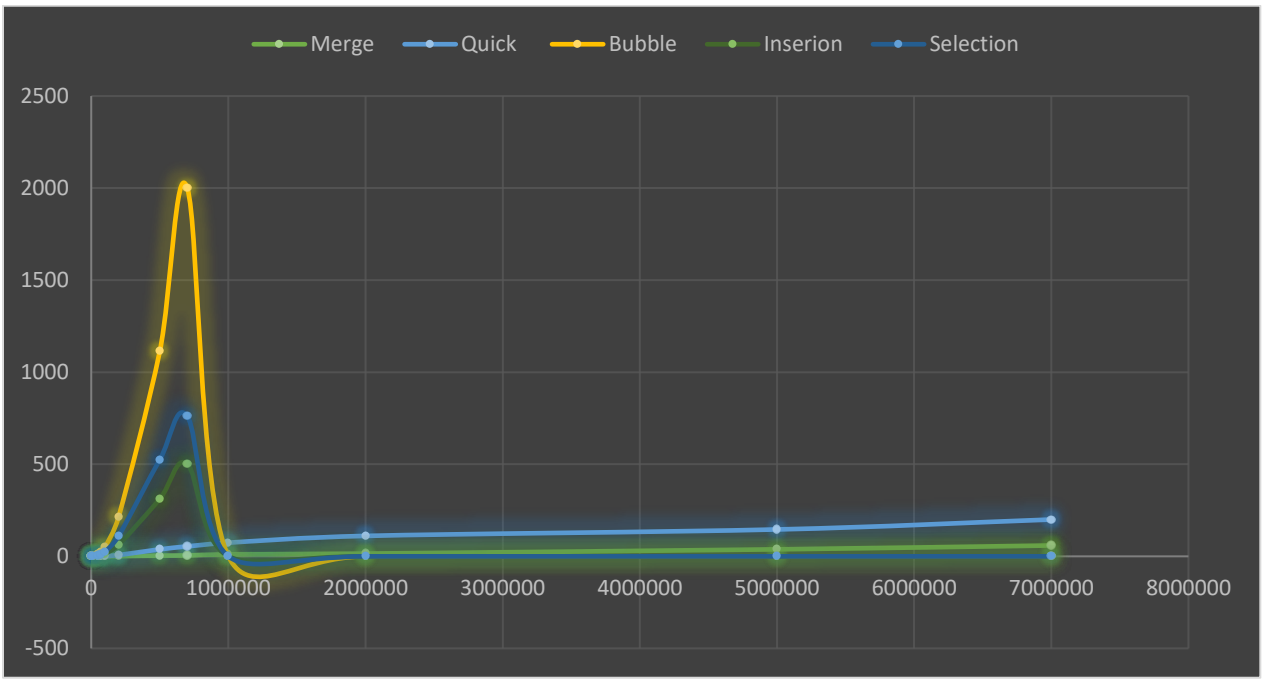

Merge Sort - in Random data
70
60
50
40
30
20
10
0
-10
0
1000000
2000000
3000000
4000000
5000000
6000000
7000000
8000000


Quick Sort- in Random data
250
200
150
100
50
0
-50
0
1000000
2000000
3000000
4000000
5000000
6000000
7000000
8000000


Bubble Sort in random data
2500
2000
1500
1000
500
0
-500
0
1000000
2000000
3000000
4000000
5000000
6000000
7000000
8000000

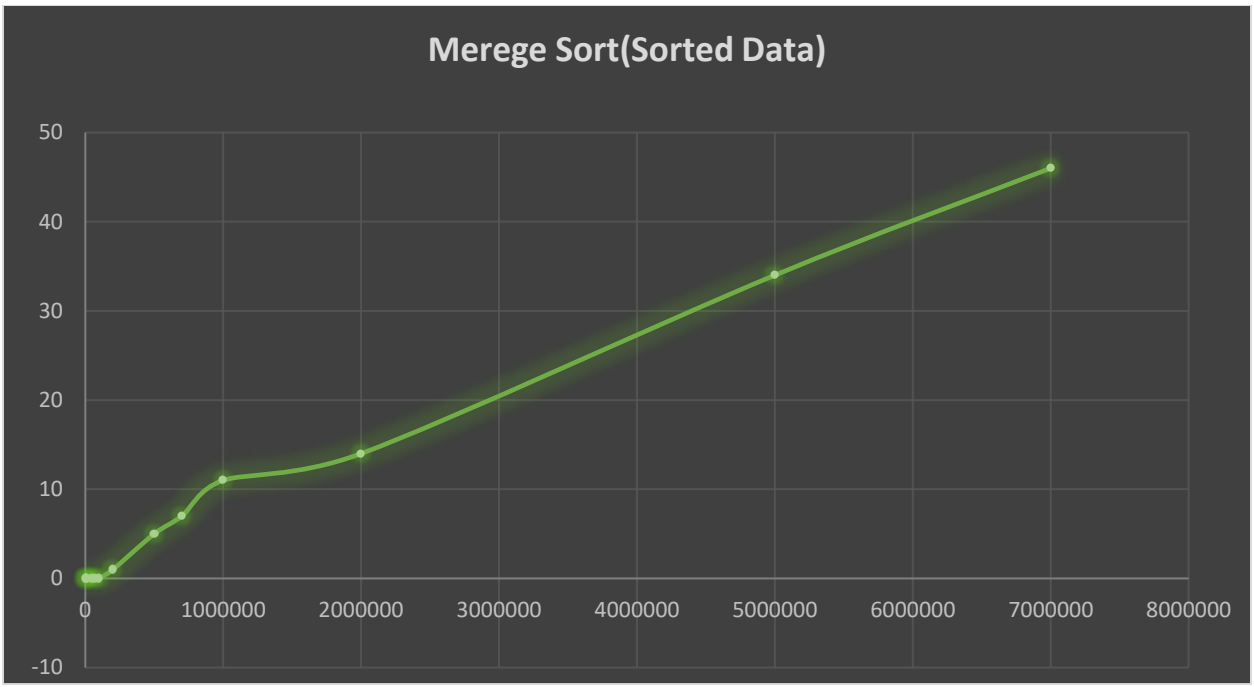

Insertion Sort in random data
600
500
400
300
200
100
0
-100
0
1000000
2000000
3000000
4000000
5000000
6000000
7000000
8000000

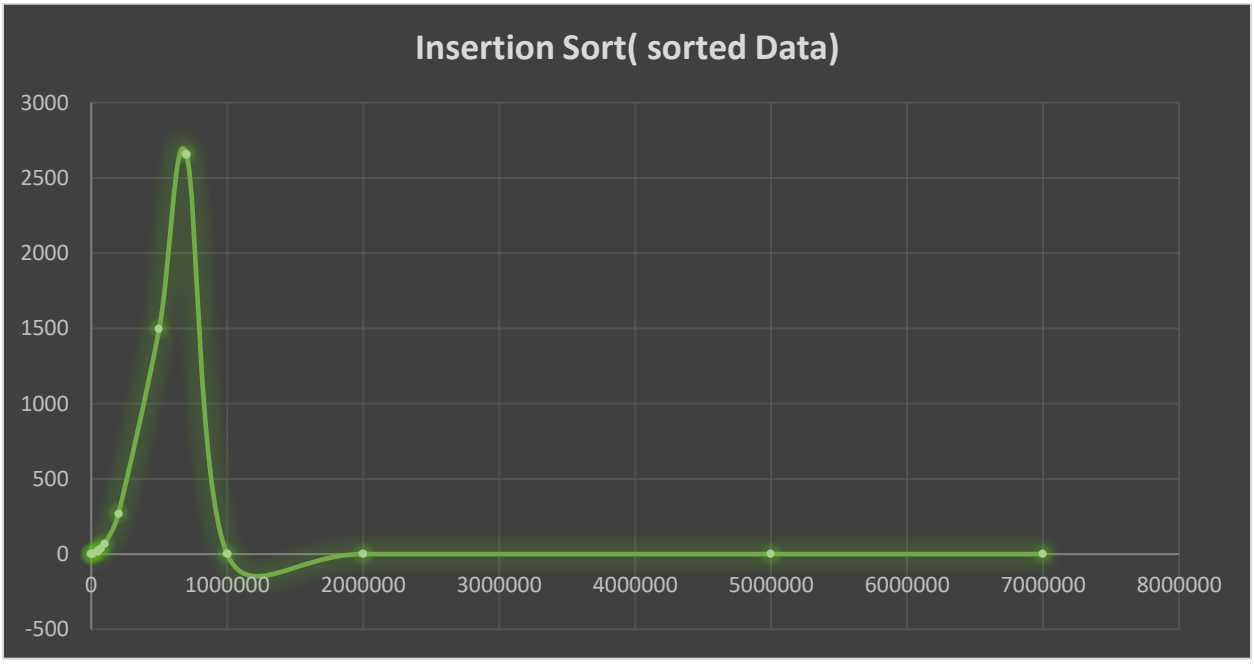

Selection Sort in random data
900
800
700
600
500
400
300
200
100
0
-100
0
1000000
2000000
3000000
4000000
5000000
6000000
7000000
8000000

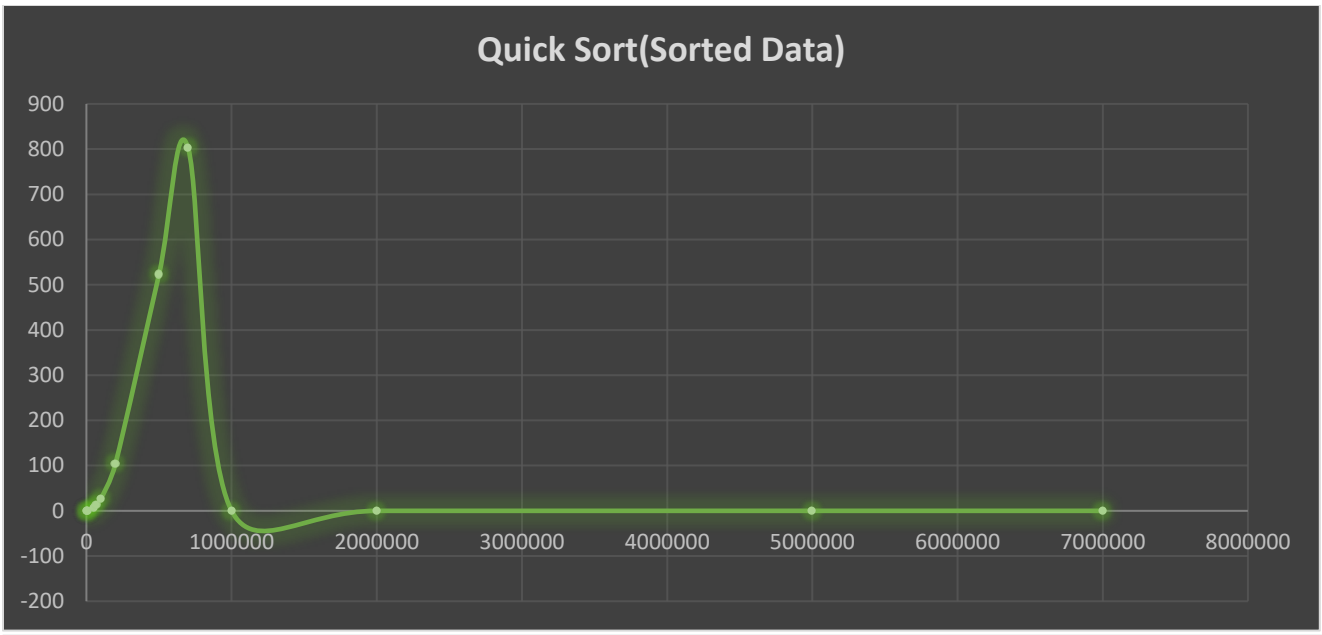

Merge
Quick
Bubble
Inserion
Selection
2500
2000
1500
1000
500
0
-500
0
1000000
2000000
3000000
4000000
5000000
6000000
7000000
8000000

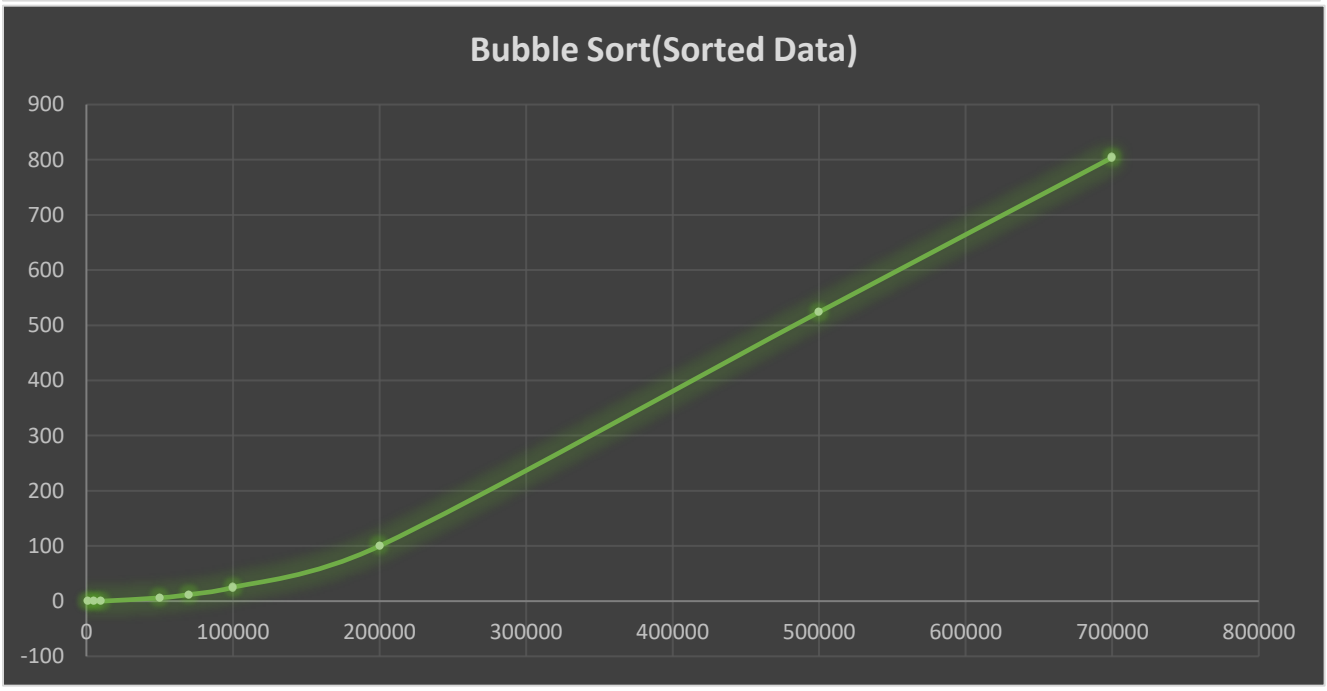
Second section: Sorted Data in the Vector.

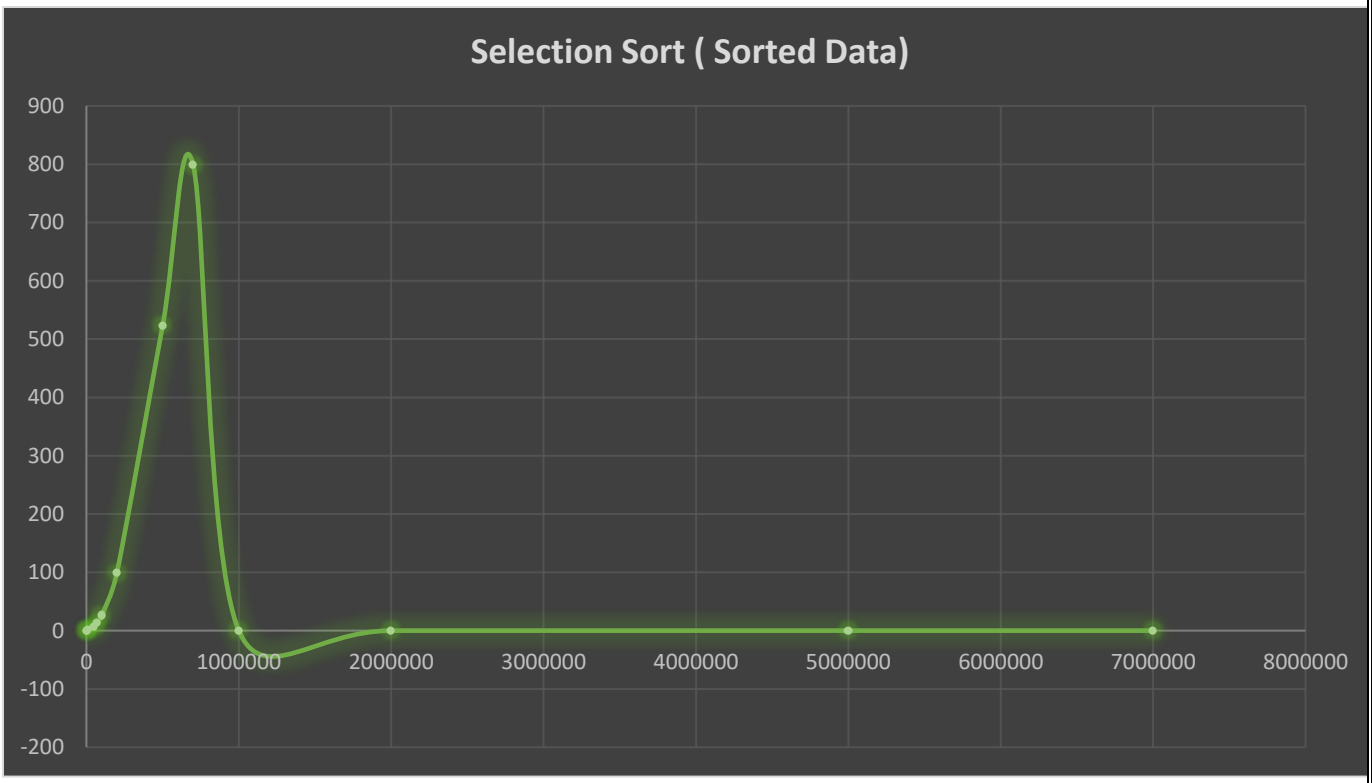
Input size	Merge Sort	Quick Sort	Bubble Sort	Insertion Sort	Selection Sort
1000	0	0	0	0	0
5000	0	1	0	0	0
10000	0	1	0	0	1
50000	0	18	7	0	6
70000	0	33	13	0	13
100000	0	67	27	0	26
200000	1	269	104	0	99
500000	5	1498	523	0	523
700000	7	2656	803	1	799
1000000	11	-	-	1	-
2000000	14	-	-	3	-
5000000	34	-	-	4	-
7000000	46	-	-	6	-

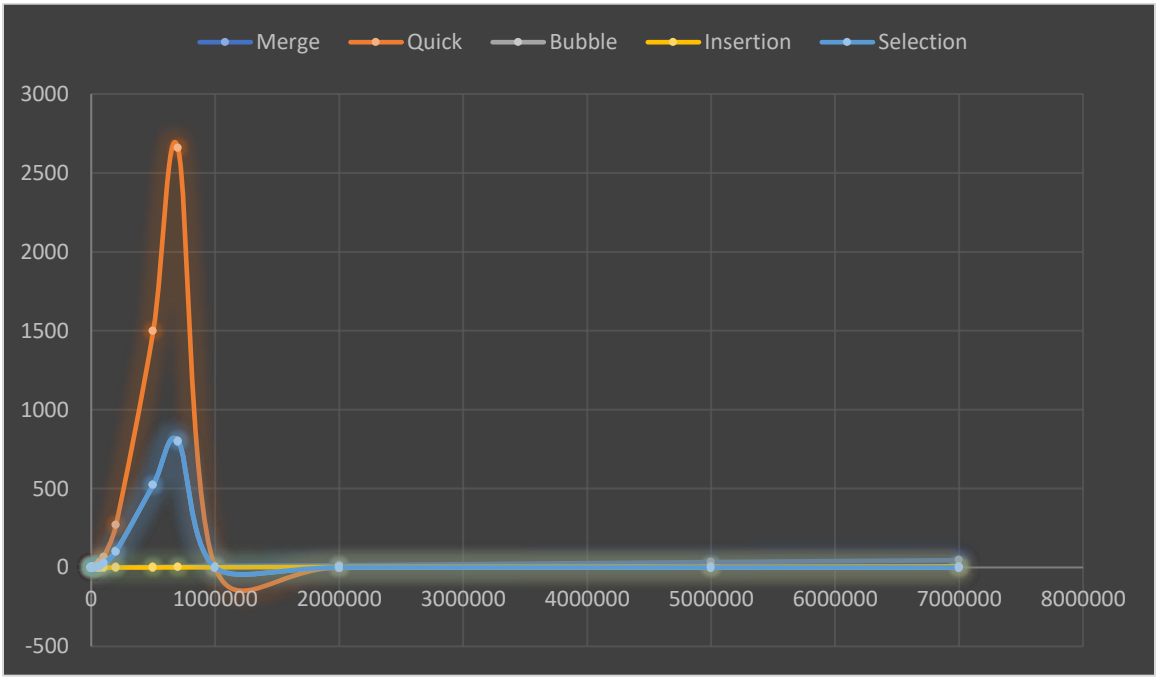

Merege Sort(Sorted Data)
50
40
30
20
10
0
-10
0
1000000
2000000
3000000
4000000
5000000
6000000
7000000
8000000


Insertion Sort(sorted Data)
3000
2500
2000
1500
1000
500
0
-500
0
1000000
2000000
3000000
4000000
5000000
6000000
7000000
8000000


Quick Sort(Sorted Data)
900
800
700
600
500
400
300
200
100
0
-100
-200
0
1000000
2000000
3000000
4000000
5000000
6000000
7000000
8000000

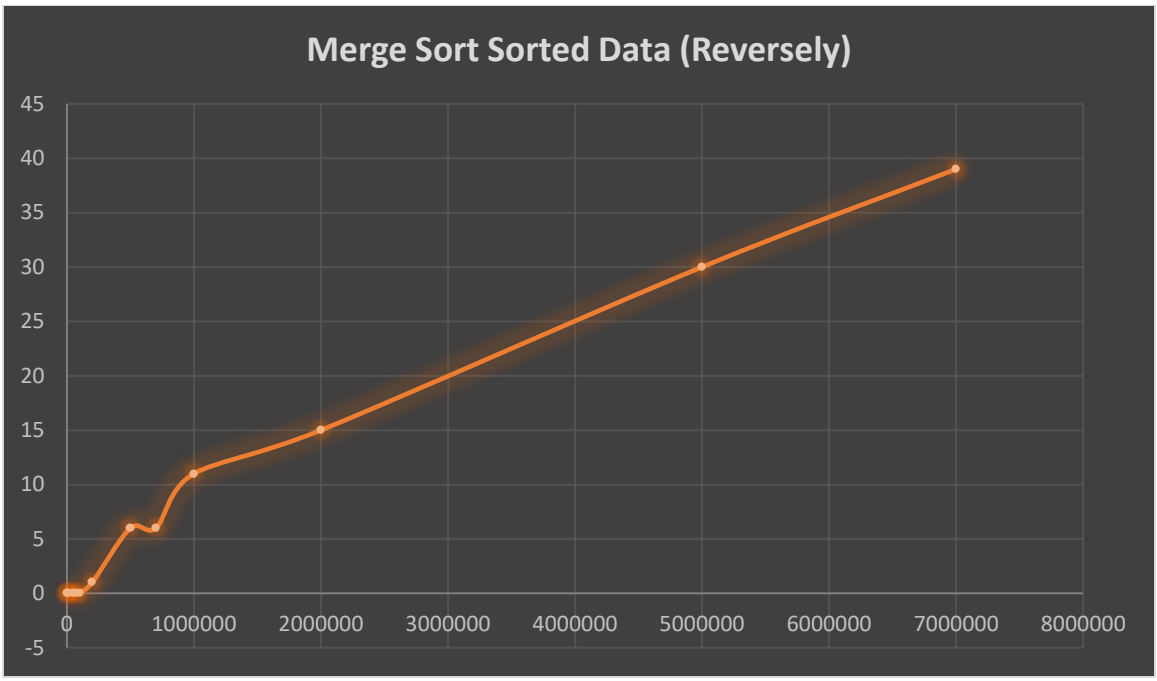

Bubble Sort(Sorted Data)
900
800
700
600
500
400
300
200
100
0
-100
0
100000
200000
300000
400000
500000
600000
700000
800000

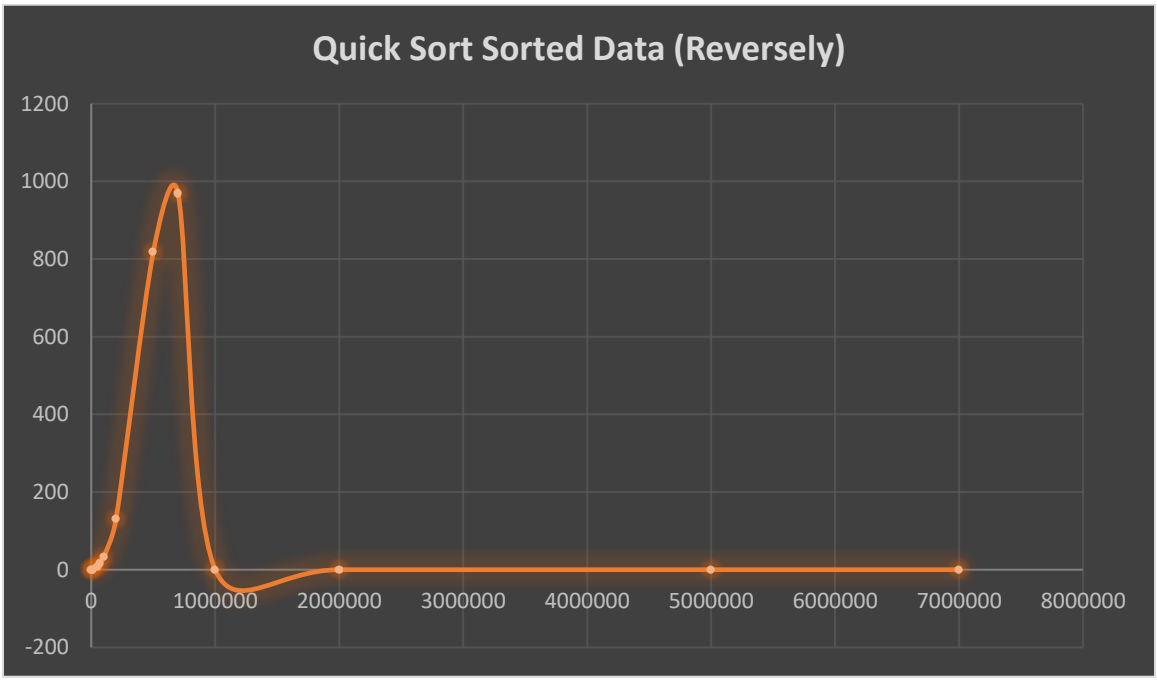

Selection Sort (Sorted Data)
900
800
700
600
500
400
300
200
100
0
-100
-200
0
1000000
2000000
3000000
4000000
5000000
6000000
7000000
8000000

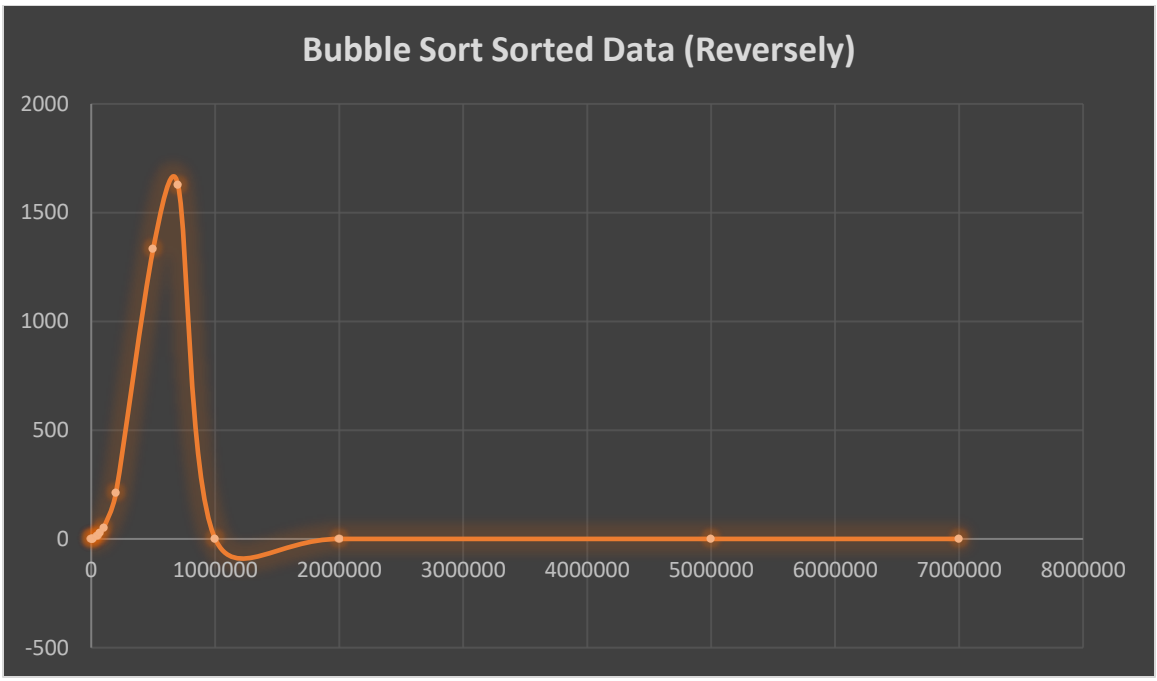

Merge
Quick
Bubble
Insertion
Selection
3000
2500
2000
1500
1000
500
0
-500
0
1000000
2000000
3000000
4000000
5000000
6000000
7000000
8000000

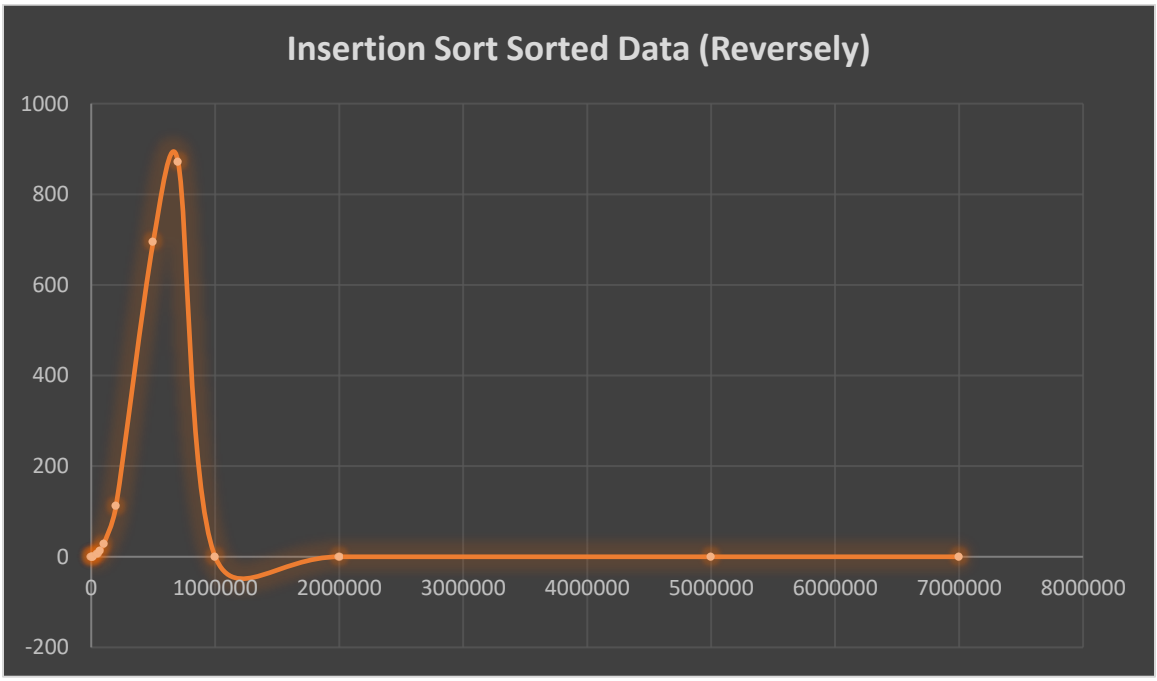
Third Section: Sorted Data (Reversely) In The Vector.

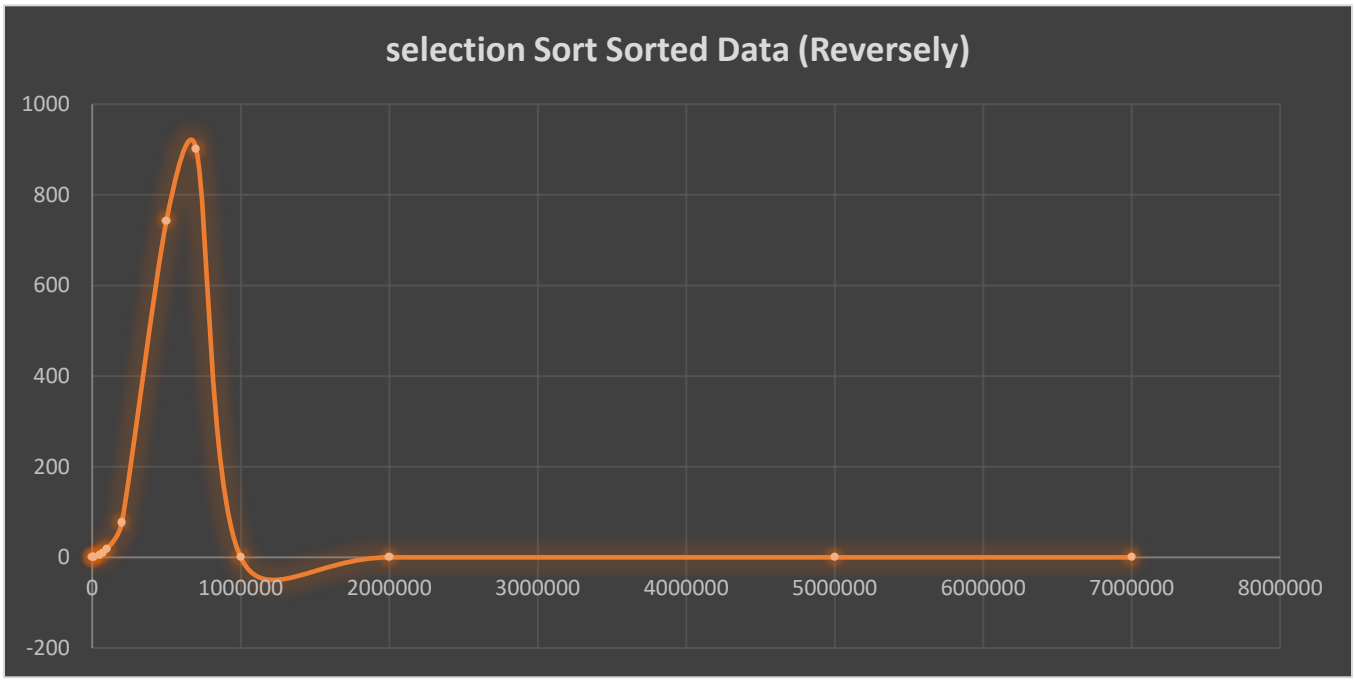
Input size	Merge Sort	Quick Sort	Bubble Sort	Insertion Sort	Selection Sort
1000	0	0	0	0	0
5000	0	0	0	0	0
10000	0	0	1	0	0
50000	0	8	13	6	5
70000	0	16	25	13	10
100000	0	32	52	28	19
200000	1	130	213	112	77
500000	6	817	1334	695	743
700000	6	969	1627	872	901
1000000	11	-	-	-	-
2000000	15	-	-	-	-
5000000	30	-	-	-	-
7000000	39	-	-	-	-

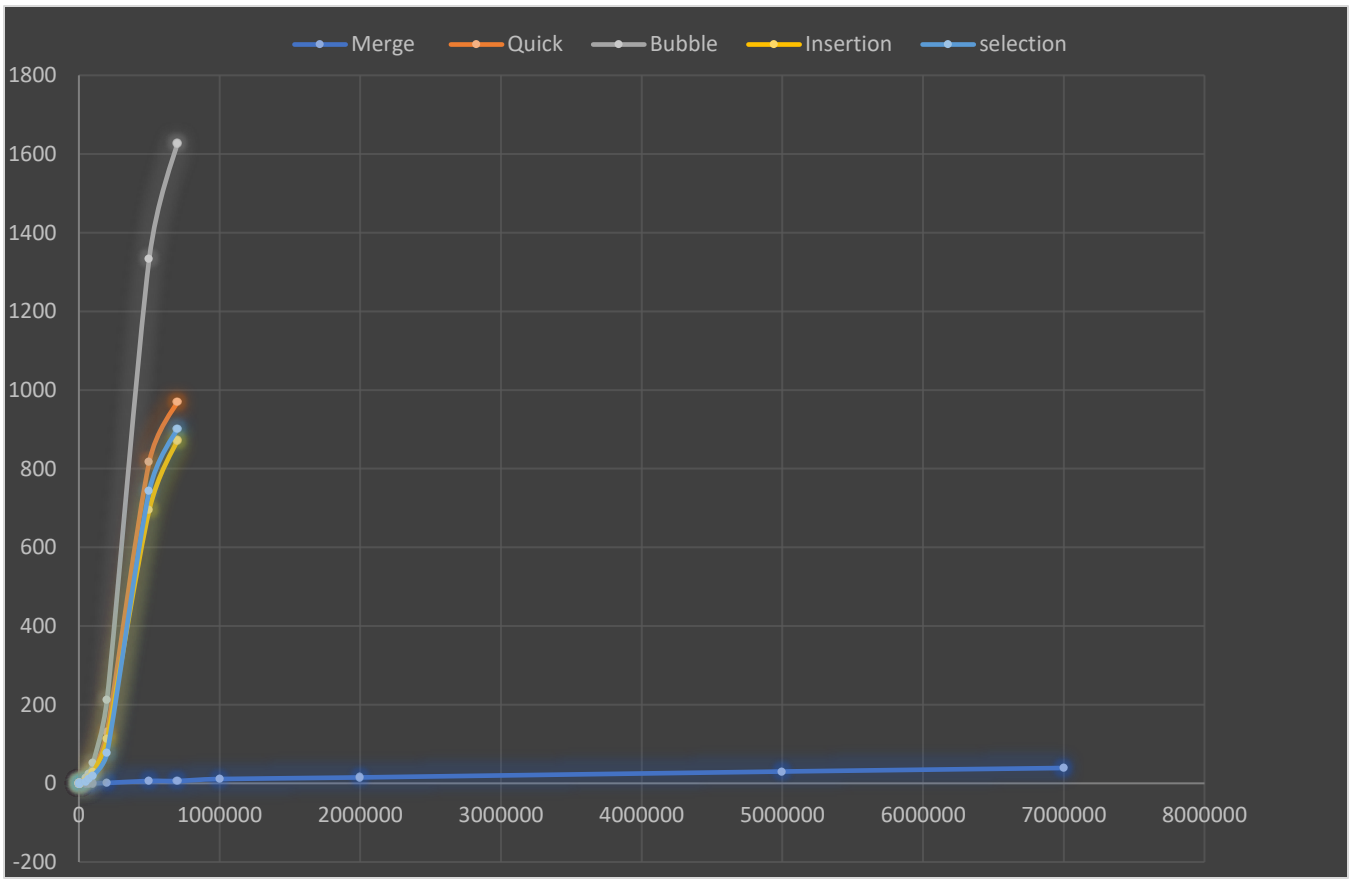

Merge Sort Sorted Data (Reversely)
45
40
35
30
25
20
15
10
5
0
-5
0
1000000
2000000
3000000
4000000
5000000
6000000
7000000
8000000


Quick Sort Sorted Data (Reversely)
1200
1000
800
600
400
200
0
-200
0
1000000
2000000
3000000
4000000
5000000
6000000
7000000
8000000


Bubble Sort Sorted Data (Reversely)
2000
1500
1000
500
0
-500
0
1000000
2000000
3000000
4000000
5000000
6000000
7000000
8000000


Insertion Sort Sorted Data (Reversely)
1000
800
600
400
200
0
-200
0
1000000
2000000
3000000
4000000
5000000
6000000
7000000
8000000


selection Sort Sorted Data (Reversely)
1000
800
600
400
200
0
-200
0
1000000
2000000
3000000
4000000
5000000
6000000
7000000
8000000


Merge
Quick
Bubble
Insertion
selection
1800
1600
1400
1200
1000
800
600
400
200
0
-200
0
1000000
2000000
3000000
4000000
5000000
6000000
7000000
8000000

Discussion of Results across Sections:

First Section:

In this section, the input data consisted of random values ranging from 0 to 5000. The results for each sorting algorithm are presented through charts. Starting with Merge Sort, it demonstrates an efficient (n log n) relation for sorting. However, when implemented using recursion, it encounters a stack overflow issue, prompting the decision to build it iteratively, resulting in favorable outcomes. Quick Sort exhibits a similar (n log n) relation to Merge Sort but with an advantage. On the other hand, Selection, Insertion, and Bubble Sorts exhibit a less desirable (n^2) relation. Among these, Bubble Sort performs the worst, yielding significantly higher execution times compared to other sorting methods. The outcomes in this section align with logical expectations.

Second Section:

In this section, the input data was already sorted. Merge, Bubble, and Selection Sorts maintain their characteristics from the first section. Notably, Quick Sort, which previously performed well, now displays an unexpectedly poor performance, demonstrating an (n^2) relation across all dataset sizes. In contrast, Insertion Sort stands out by showcasing a linear relation, proving to be the most efficient when dealing with pre-sorted data. The results in this section diverge from expectations, presenting a surprising and unexpected shift in Quick Sort's performance.

Third Section:

Similar to the first section, the input data in this section was sorted but in reverse order. The results for all sorting algorithms align with those of the first section, except for Quick Sort, which once again exhibits unexpectedly poor results, consistent with the second section. This repetition of unexpected outcomes emphasizes the sensitivity of Quick Sort to the specific characteristics of the input data. The results in this section echo the surprises encountered in the second section.

In summary, while the performance of sorting algorithms is generally in line with expectations in the first and third sections, the second section presents unexpected challenges, particularly with Quick Sort when dealing with sorted input data. This underscores the importance of considering the nature of the input data when selecting sorting algorithms for optimal performance.

Stability:

Sorting algorithms exhibit different levels of stability, a quality that refers to how well they preserve the relative order of elements with equal keys during the sorting procedure. A sorting algorithm is deemed stable if it ensures that if two elements possess the same key and are in a specific order in the input, they will maintain that order in the resulting sorted output.

For clarity, a comparison illustrating the stability of various sorting algorithms will be presented in the following table.

	Merge	**Quick**	**Bubble**	**Insertion**	**Selection**
Time complexity in first case	n logn	n logn	n^2	n^2	n^2
Time complexity in second case	n logn	n^2	n^2	n	n^2
Time complexity in third case	n logn	n^2	n^2	n^2	n^2
Stability	Stable	Non-Stable	Stable	Stable	Non-Stable

Conclusion:

In conclusion, the empirical analysis of sorting algorithms provides valuable insights into their practical performance characteristics, enabling us to make informed decisions when choosing the most suitable algorithm for specific sorting tasks. Throughout this assignment, we have explored various sorting algorithms, ranging from traditional ones like Bubble Sort, Insertion Sort, and Selection Sort to more advanced algorithms like Merge Sort and Quick Sort.

source code:

https://drive.google.com/drive/u/1/folders/1xAlExWRyK1iDV9cLUS2nPKCASRUDYEAF

resources and references:

1- https://www.geeksforgeeks.org/cpp-program-for-quicksort/

2- https://www.geeksforgeeks.org/merge-sort/

3- https://www.geeksforgeeks.org/bubble-sort/

4- https://www.geeksforgeeks.org/cpp-program-for-insertion-sort/

5- https://www.geeksforgeeks.org/selection-sort/

6- https://www.digitalocean.com/community/tutorials/random-number-generator-c-plus-plus