FACULTY OF INFORMATION TECHNOLOGY

DATA STRUCTURES

&

ALGORITHMS

Semester 3 2022

Sorting Project Report

supervised by
Mr.Bui Huy Thong & Ms.Tran Thi Thao Nhi

Preface

This project has been done in two weeks by four of us for "Data Structure & Algorithms " course, taught by our lecturers: Mrs. Van Chi Nam, Dr. Le Thanh Tung, Mr.Bui Huy Thong, and Ms. Tran Thi Thao Nhi. Since it revealed our knowledge of this course, as a result, it may lead to misunderstanding, however it still reflects many good things from self-studying, please read it carefully.

Acknowledgements

Thank you Mr. Van Chi Nam, Dr. Le Thanh Tung, Mr. Bui Huy Thong and Ms. Tran Thi Thao Nhi for supervising us in this project, on both theoretical and practical side, with many skills such as critical thinking, writing reports and working as a team, we could not have done it without your helps.

Team Members

Mac Tuan Trung - 21127462,
Phan Lam Anh - 21127580,
Tong Gia Huy - 21127307,
Doan Nguyen Tan Sang - 21127683

Keywords

- Sorting; Data Structures and Algorithms; HCMUS; Time Complexity; Space Complexity; Data; comparison; GitHub

Programming Note

There is some sum of works through our process. We've been given a problem that we need to implement and demonstrate 11 different sorting algorithms on the same data structures. Due to the fact that it's an interesting problem since we've learned this in theory class, this is a good opportunity for us to improve both hard-skill and soft-skill. Hence, there were many things we need to set up before working :
+ We use LaTeX(mainly overleaf.com) to compose our report.
+ We use GitHub to store, manage, and control changes in our source.
+ Each day, members should fetch and pull code from GitHub.
+ Whenever conflicts appeared, we use discord to resolve and merge code.
+ When a member completed a new sorting algorithm and pushes it on GitHub, the Coding leader will check if there are any bugs.
+ All members read the checked code and free to ask if there is any question. We have 7 files in our project.
Main.cpp for int main() function, conditions for 5 commands
SortAlgorithm.cpp for source code of 11 sorting algorithms. Each algorithm we have 2 versions: origin version (for finding running time) and comparison version (for number of counting comparison).

1. *Selection Sort*
2. *Insertion Sort*
3. *Bubble Sort*
4. *Shaker Sort*
5. *Shell Sort*
6. *Heap Sort*
7. *Merge Sort*
8. *Quick Sort*
9. *Counting Sort*
10. *Radix Sort*
11. *Flash Sort*

SortAlgorithm.h for functions declarations of 11 sorting algorithms
DataGenerator.cpp for 4 data order generating functions
GenerateRandomData: *Random data*
GenerateSortedData: *Already sorted data (smallest to largest)*
GenerateReverseData: *Reverse data (largest to smallest)*
GenerateNearlySortedData: *Nearly sorted data*
Generator.h for functions declarations of 4 generating functions
Source.cpp for some other functions to discriminate 5 commands and support us debugging:
chooseGenerator: *read the command line arguments and choose a data generator in DataGenerator.cpp (-rand, -rev,...)*
chooseMode: *read the command line arguments and choose output parameters (time, comparision or both)*
chooseSort: *read the command line arguments and choose right sorting algorithms command(1,2,3,4,5) Output: print out the answers for user commands*
checkOrder: *have a function to check if an array is ordered or not*
checkBucket: *using for debug flash sort to check if all the elements have already in the right group (bucket)*
Header.h for all functions declarations:
Call in Main.cpp for declare all functions and library

Contents

1 Introduction

There are many common problems in different practical fields in the information technology industry, especially the problem of information processing is focused on the world because of its importance in terms of processing performance as well as time. execution time of an application. With the rapidly growing information processing needs in our world, to be able to quickly process the data, we need to have the most arrangement or order to be able to use the algorithms. Math needs a specific rule such as a binary search that needs the data to be sorted in ascending or descending order, or a heap data structure that helps handle sorting problems, finding k times word part according to a certain requirement (top k problem), reducing the processing time of algorithms. One of the ways to solve this problem is sorting. To determine the optimality of a sorting algorithm, several factors must be considered: time complexity, stability, and memory space. Since the sorting problem has attracted a lot of researchers, it has led to the increasing development of sorting algorithms. Therefore, we have chosen 11 basic sorting algorithms to learn, do experiments and get the measured results and compare to find out which algorithms are the most optimal. And here are our results.

2 Algorithm Presentation

We presented 11 different sorting algorithms implemented in this project, including : ideas, step-by-step descriptions, and complexity evaluations (in terms of time complexity and space complexity, if possible). Variants/improvements of an algorithm, if there is any, should be also mentioned.

2.1 Selection Sort

2.1.1 Idea[1] [2]

Selection sort goes through the data sequence and "selects" each data element that is not in its correct position(minimum if ascending or maximum if descending) It then swaps it with the data element in the leftest position of the unsorted elements.

2.1.2 Algorithm

Step 1: Start with $i = 0$ and use a loop from i to $n - 1$ to find the minimum element.

Step 2: Swap the minimum with the i th element.

Step 3: Go back to step 1 continue with $i + 1$ until the array it sorted.

2.1.3 Space-time Complexity

Time complexity: $\mathrm{O}(n^2)$ at all cases

Space complexity: $\mathrm{O}(1)$

2.2 Insertion Sort

2.2.1 Idea[3]

This algorithm go through the array, then gradually move the current value along the array to their correct position.

2.2.2 Algorithm

The algorithm will go through each element of the array and do the following:

Check if the current value is smaller than the one before it:

• If yes: "insert" the current element (a_i) before the one before it (a_{i-1}), then check again, continue the insertion if necessary.

• If no: do nothing and move on to the next element.

2.2.3 Space-time Complexity

Time complexity: $\mathrm{O}(n^2)$. Best case is $\mathrm{O}(n)$ when the array is already sorted and no insertion is needed.

Space complexity: $\mathrm{O}(1)$

2.2.4 Improvement[4]

The linear search using the for loop have to check a condition each time whether the index is still in the array. So if we store the data from $a[1]$ to $a[n]$ and let $a[0]$ be a sentinel we use the while loop and don't care about the above problem because it always stop before the index 0. With this, we can reduce many times of comparisons.

Binary insertion sort uses binary search instead of using linear search to find the position in which the element need to be inserted. This will help us reduce the comparisons

2.3 Bubble Sort

2.3.1 Idea[5][6]

Bubble sort works by comparing each pair of elements in the array and switching their place to put them in the right order (ascendingly or descendingly). The algorithm will go through each pair one by one, make any adjustments necessary, and repeat until the array is sorted. The array will be sorted from left to right. For an array of N elements, there will be N passes, after each pass, the unsorted range will be gradually decreased until there are no more unsorted elements.

2.3.2 Algorithm

The algorithm consists of 2 nested loops, with the outer one looping N times (N being the number of elements in the array). The inner loop will go through the unsorted elements, and do the following:
Step 1 : Check if the current value is bigger than the one after it, if yes then switch the places of those values, if not then do nothing, and move on to the next element.
Step 2 : Decrease the size of the unsorted part by 1.

2.3.3 Space-time Complexity

Time complexity: $O(n^2)$. Adjustments can be made to identify an already sorted array, then the time complexity for the best case can be $O(n)$.
Space complexity : $O(1)$

2.3.4 Improvement [4]

Each turn we go left-to-right, we check if there is no swaps that mean the array has already sorted then stop the process.
The last position we do the swap each turn let us know form that position to the end, all the elements is sorted, so at next turn we just need to go to that position. This help the algorithm runs faster.

2.4 Shaker Sort

2.4.1 Idea[7][3]

Shaker sort is a variation of bubble sort. Shaker sort bring the largest and the smallest element to the right position by traversing alternatively forward and backward.
In an array, we start from the beginning of the array to the end and check each adjacent pair of element if they are in the right position or not. If not, we swap them.

After taking one element from the front to the right position, we continue sorting from the back until the array is sorted.

2.4.2 Algorithm

Step 1: If the begin and the end is the same, end the process.
Step 2: Start sorting from the beginning and check each adjacent pair of element if they are in the right position or not. If not, we swap them. Decrease the end by 1.
Step 3: Start sorting from the end and and check each adjacent pair of element if they are in the right position or not. Increase the beginning by 1.
Step 4: Go to step 1.

2.4.3 Space-time Complexity

Time complexity: Best case is O(n) when the array is already sorted, (check if there are swaps in the improved version).
Average and worst case is O(n^2).
Space complexity: O(1) for all case

2.4.4 Improvement

Each turn (left to right or right to left), we mark the last position we do the swap. At the end of the left-to-right tour, we update the end. It means we don't need to check the rest after that position because they have already in the right positions. Do the same with the right-to-left tour (update the beginning instead of the end).

2.5 Shell Sort

2.5.1 Idea[8]

Shell short is mainly a variation of Insertion sort.We using an interval as a key comparison between two value, start from far interval and keep reducing until 0.

2.5.2 Algorithm [9]

Step 1 : Create a interval : $h = 1$.

Step 2 : Let $h = h * 3 + 1$ until $h > n/3$.

Step 3 : Check if the gap (h) is greater than 0. If not end the process.

Step 4 : Start at the position $i = h$

Step 5 : If i is greater than or equal to n, go to step 7. If i is smaller than n, compare two elements at index $j = i$ with the element at the position $j - h$. If these two in right position, increase i by 1 and repeat this step.

Step 6 : Swap them, update the and then repeat step 5 but with $j = j - h$.

Step 7 : Decrease the gap $h = (h - 1)/3$ and go to step 3.

2.5.3 Space-time Complexity

Time complexity :

O(n^2), the gap is reduced by half in every iteration.

The best and average case is O($nlog(n)$)

Space complexity : O(1)

2.6 Heap Sort

2.6.1 Heap Structure[10]

The structure of heap was first mentioned by J. W. J. Williams in Communication of the ACM, in which he wrote: "The elements are normally so arranged that $A[i] \leq A[j]$ for $2 \leq j \leq n, i = j/2$. Such an arrangement will be called a heap". His definition of the heap will later be referred to as the min heap, since "$A[1]$ is always the least element of the heap". The max heap is opposite to the min heap: for each parent element at position i, the value of its children at position $2i + 1$ and $2i + 2$ are no greater than the parent, thus the first element of the heap is the largest.

2.6.2 Idea

The heap sort algorithm has the same idea as selection sort. Instead of going through the array to find the maximum value, it will construct a heap and get the first element as the maximum.

2.6.3 Algorithm

Step 1 : Construct a max heap: Start at the middle of the array (let $i = n/2$ with n as the size of the array), check the children elements at position $2i + 1$ and $2i + 2$; if either children is larger than the parent, swap the position of said child with the parent, then repeat the algorithm to check the children of that child. Decrease i by 1 and repeat until i reaches the front of the array. This method is called the 'bottom-up' construction of a heap.
Step 2 : Switch the position of the first value of the heap with its corresponding position in the array (towards the end of the array).
Step 3 : Decrease the heap size by 1, rebuild the heap at position index 0 and repeat until the array is sorted (the number of elements in the heap is 0).

2.6.4 Space-time Complexity

Time complexity: $\mathrm{O}(n \log_2 n)$, where $\mathrm{O}(\log_2 n)$ is the complexity of heap construction, and $\mathrm{O}(n)$ is the complexity for putting the first heap element to its correct position.
Space complexity: $\mathrm{O}(1)$

2.7 Merge Sort

2.7.1 Idea[11]

Merge Sort use Divide and Conquer. It divides the input array into two halves into only 1 element and then it merges two halves into a sorted one.

2.7.2 Algorithm

Step 1: Use a variable to save the position of the middle element and split it into 2 halves until each halves only have 1 elements.

Step 2: Choose each 2 split halves and create a new temp array.

Step 3: Pick the smallest out of those 2 halves and put in the temp array until picked all elements in those 2 halves.

Step 4: Copy the temp array into the main array in the right position.

Step 5: Continue until temp array with n th elements copy into the main array.

2.7.3 Space-time Complexity

Time complexity: $O(n\log_2 n)$ at all cases since it either way will split until have 1 element so it no diffent at any case.

Space complexity: $O(n)$

2.7.4 Improvement

There are :

- K-way merge sort[12][13]: By splitting into k part, it time complexity instead of $n\log_2 n$ it become $n\log_k n$ which is smaller so it will run faster than normal merge sort

- External merge sort[14]: useful when data is bigger than than the main memory

- Natural merge sort[15]: In the array if there are some part that elements that are already sorted, it will only merge it with another sorted array instead of splitting. So in the best case which is the array is sorted, time complexity is $O(n)$ only

- Non-recursive merge sort: requires less space since it not using recursion.

2.8 Quick Sort

2.8.1 Idea[16]

Quick Sort use Divide and Conquer. It picks an element as pivot and split into 2 specific area in the given array and sorted it.

2.8.2 Algorithm

Step 1: Choose a pivot by choosing the middle element and split 2 areas indicate smaller (S_1) and larger (S_2) then the pivot.
Step 2: Swap the pivot with the last element.
Step 3: Use a loop from begin to $n-1$ element, use a variable (j) to save the index of the last element of S_1.
Step 4: Each time we find a element that smaller than the pivot, we increase j by 1 and swap it with j.
Step 5: After done the loop, swap the pivot with the j + 1 element. With that, we have sorted all element into the correct area (S_1 and S_2).
Step 6: Pick a pivot by choosing the middle element again in each area and continue like step 1 until the area have only 1 element.

2.8.3 Space-time Complexity

Time complexity:
O($nlog_2n$) at best cases when we always pick exactly the median of the subarray as a pivot.
O($nlog_2n$) average cases when pick other numbers as a pivot.
O(n^2) at worst cases when we always pick the min or max as a pivot
Space complexity: O($logn$)

2.8.4 Improvement

There are many version on how to pick a pivot (pick first or last or random or median). Pick the first, the middle, the last elements of the array and find out which one is the closest to the average of those 3 elements to become the pivot, and split it into 2 halves like other algorithm. With picking the median of three, We can avoid the worst case.

2.9 Counting Sort

2.9.1 Idea

It works by counting the number of objects having distinct key values . Then do some arithmetic to calculate the position of each object in the output sequence.https://www.overleaf.com/project/62a59e524128ad8f28d2

2.9.2 Algorithm

Step 1 : Find the max-value of the array.

Step 2 : Create an max-element frequency array.

Step 3 : Go through the array and count number of appearances of each value.

Step 4 : Go through the frequency array from 0 to max, frequency[i] is number of elements whose value is i. Put them sequentially to the first array.

2.9.3 Space-time Complexity

Time Complexity: O(max)

Space Complexity: O(max)

2.9.4 Improvement

In the first step, we find both the min-value and the max-value then create the frequency array with $[max - min + 1]$ elements. Let frequency[i] is the number of elements whose value is $i + min$. This help us avoid wasting memory and also use for sorting negative integers.

2.10 Flash Sort

2.10.1 Idea[17]

Flash Sort separate the array into many small groups (buckets) and move each right element into it's group (subclass arrangement). After that, we use insertion sort for the "nearly sorted" array.

We create many groups (buckets) [0.45 * size of array][18] and move elements into. In each group, elements' degree of priority in sorting is very low. And Using insertion sort in "nearly sorted" array is efficient.

2.10.2 Algorithm

Step 1 : Separate data into many buckets ($\lfloor 0.45n \rfloor$ buckets)

Step 2 : Find the min and the max value in the array.

Step 3 : Calculate number of elements in each bucket: index i element is in the bucket number (numberOfBucket$-1)(a[i] - min)/(max - min)$.

Step 4. Use prefix sum of array to get the last element in each bucket.

Step 5 : For each element index i int the array, check if it is already in the right bucket. If not go to swap it with the last elements which is in the last position of the right bucket and decrease the last position of that bucket by 1. Then recheck the index i element.

Step 6 : Use insertion sort for the array.

2.10.3 Space-time Complexity

Time complexity: O(n) in all cases, because in every case we have to check and move all elements into right buckets. And then use insertion sort with it's "nearly" best case (nearly sorted array) is also O(n)

Space complexity: O(n)

2.11 Radix Sort

2.11.1 Idea[19]

We create many buckets which present for value 0 to 9, then we compare every elements from rightmost to leftmost and sort the array.

2.11.2 Algorithm

Step 1 : Find max value on the array

Step 2 : Begin with the least significant digit, count the number of digit from 0 to 9 and then store into another array

Step 3 : We encount the first place of each elements, with $L[i] = L[i+1] - L[i]$

Step 4 : Put them into the first array sequentially: For each element, put it in the position $L[i]$ (with i is the digit we are checking) in the array and increase the value of $L[i]$ by one (for the next digit i element). Go to step 2 with the next left digit.

2.11.3 Space-time Complexity

Time Complexity: The best, worst and average case is $\mathrm{O}(n)$

Space Complexity : $\mathrm{O}(n)$.

3 Experiment

3.1 Experimentation protocol

We wrote our algorithms by C++ language and performed it on Visual Studio Community 2022. We conducted the experiment by 4 different data order : including: Sorted data (in ascending order), Nearly sorted data, Reverse sorted data and Randomized data. We also examine the algorithms by 6 different sizes : 10,000, 30,000, 50,000, 100,000, 300,000, and 500,000 elements in order to make the experiment precisely.

3.2 System Information

Operating System: Windows 11 Home Single Language 64-bit (10.0, Build 22000)

System Manufacturer: ASUSTeK COMPUTER INC.

System Model: ASUS TUF Gaming F15 FX506LI

BIOS: FX506LI.310

Processor: Intel(R) Core(TM) i7-10870H CPU @ 2.2GHz (16 CPUs), 2.2GHz

Memory: 16384MB RAM

3.3 Data tables

Running time (s)

Table 1: Data order: Randomized

Data size	10,000		30,000		50,000	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection Sort	0.106	100010001	0.84	900030001	2.375	2500050001
Insertion Sort	0.083	49732368	0.682	450061772	1.912	1249869530
Bubble Sort	0.294	99990760	2.98	900029725	8.101	2499815912
Shaker Sort	0.281	66179081	2.439	600566050	6.282	1667948201
Shell Sort	0.001	705524	0.004	2708111	0.008	5221186
Heap Sort	0.001	497281	0.005	1681784	0.007	2952597
Merge Sort	0.001	583633	0.005	1937447	0.01	3382946
Quick Sort	0.001	348754	0.004	1144605	0.007	1962613
Counting Sort	0	69996	0	210002	0	298303
Radix Sort	0	140100	0.002	510125	0.002	850125
Flash Sort	0	124060	0.001	374037	0.001	613688

Data size	100,000		300,000		500,000	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection Sort	8.801	10000100001	77.959	90000300001	224.625	250000500001
Insertion Sort	7.267	5002127647	66.435	45012353811	186.966	125038004002
Bubble Sort	31.84	9999691632	286.588	90000575665	794.017	249998833217
Shaker Sort	25.413	6665159882	213.312	60060320343	702.103	166819626165
Shell Sort	0.018	11576958	0.06	41537882	0.107	76374176
Heap Sort	0.017	6304476	0.054	20797057	0.097	36117636
Merge Sort	0.019	7166399	0.058	23381557	0.1	40381532
Quick Sort	0.016	4249944	0.046	15373766	0.079	29203599
Counting Sort	0	498306	0.001	1298306	0.003	2098306
Radix Sort	0.006	1700125	0.021	5100125	0.028	8500125
Flash Sort	0.004	1189961	0.012	3569969	0.022	5949945

Table 2: Data order: Nearly Sorted

Data size	10,000		30,000		50,000	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection Sort	0.104	100010001	0.859	900030001	2.343	2500050001
Insertion Sort	0	155214	0	558322	0	617170
Bubble Sort	0.083	93997885	0.799	897170001	1.734	1992093480
Shaker Sort	0	157196	0.001	576062	0.002	657062
Shell Sort	0	294276	0.001	1065064	0.002	1661444
Heap Sort	0.001	518555	0.003	1739603	0.006	3056581
Merge Sort	0.001	498602	0.004	1637250	0.006	2823367
Quick Sort	0	255559	0.002	864723	0.004	1518148
Counting Sort	0	70002	0	210002	0	350002
Radix Sort	0	140100	0.001	510125	0.002	850125
Flash Sort	0	99045	0	297033	0	495043

Data size	100,000		300,000		500,000	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection Sort	8.911	10000100001	80.476	90000300001	232.2	250000500001
Insertion Sort	0	697362	0	1155074	0.001	1913502
Bubble Sort	3.599	4350573105	8.588	10175138845	17.878	20872201725
Shaker Sort	0.002	647544	0.001	941278	0.002	1443758
Shell Sort	0.003	3168349	0.009	21431846	0.017	17044689
Heap Sort	0.011	6519531	0.034	11152000	0.061	37116289
Merge Sort	0.013	5845742	0.039	18715920	0.067	32106719
Quick Sort	0.007	3164330	0.021	10292617	0.036	17779187
Counting Sort	0	700002	0.001	2100002	0.003	3500002
Radix Sort	0.006	1700125	0.019	6000150	0.034	10000150
Flash Sort	0.001	990057	0.003	2970045	0.005	4950041

Table 3: Data order: Sorted

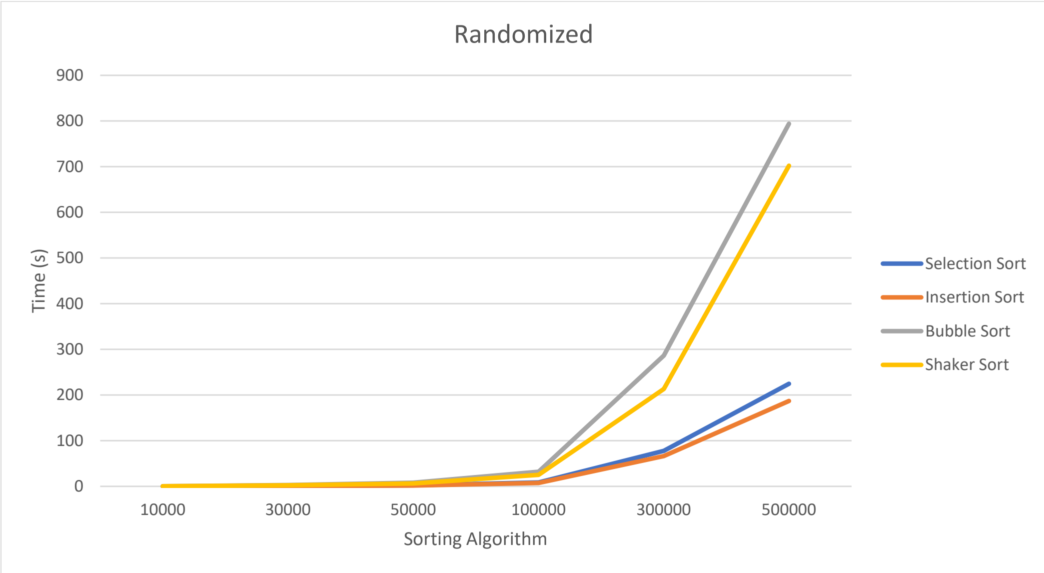
Data size	10,000		30,000		50,000	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection Sort	0.099	100010001	0.838	900030001	2.349	2500050001
Insertion Sort	0	29998	0	89998	0	149998
Bubble Sort	0	20001	0	60001	0	100001
Shaker Sort	0	20002	0	60002	0	100002
Shell Sort	0	225757	0	767188	0.001	1367188
Heap Sort	0	518705	0.003	1739633	0.006	3056481
Merge Sort	0.001	475242	0.003	559914	0.008	2722826
Quick Sort	0	244975	0.001	823646	0.003	1467264
Counting Sort	0	70002	0	210002	0	350002
Radix Sort	0	140100	0.002	510125	0.002	850125
Flash Sort	0	99001	0	297001	0	495001

Data size	100,000		300,000		500,000	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection Sort	8.982	10000100001	79.845	90000300001	222.897	250000500001
Insertion Sort	0	299998	0	899998	0.001	1499998
Bubble Sort	0	200001	0	600001	0	1000001
Shaker Sort	0	200002	0	600002	0.001	1000002
Shell Sort	0.002	2901472	0.008	21431637	0.015	16804315
Heap Sort	0.011	6519813	0.035	11151888	0.062	37116275
Merge Sort	0.012	5745658	0.038	18645946	0.066	32017850
Quick Sort	0.007	3134498	0.021	10258249	0.036	17737894
Counting Sort	0	700002	0.001	2100002	0.003	3500002
Radix Sort	0.006	1700125	0.02	6000150	0.034	10000150
Flash Sort	0.002	990001	0.003	2970001	0.005	4950001

Table 4: Data order:Reverse sorted data

Data size	10,000		30,000		50,000	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection Sort	0.108	100010001	0.894	900030001	2.497	2500050001
Insertion Sort	0.161	100009999	1.394	900029999	3.769	2500049999
Bubble Sort	0.338	100020000	2.936	900060000	8.059	2500100000
Shaker Sort	0.361	100005001	3.156	900015001	8.319	2500025001
Shell Sort	0	378112	0.001	1309440	0.002	2142223
Heap Sort	0.001	476739	0.003	1622791	0.006	2848016
Merge Sort	0.001	476441	0.004	1573465	0.006	2733945
Quick Sort	0	254764	0.002	855622	0.003	1508351
Counting Sort	0	70002	0	210002	0	35002
Radix Sort	0	140100	0.001	510125	0.002	850125
Flash Sort	0	109001	0	327001	0.001	545001

Data size	100,000		300,000		500,000	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection Sort	9.459	10000100001	86.078	90000300001	239.374	250000500001
Insertion Sort	14.341	10000099999	131.041	90000299999	374.646	250000499999
Bubble Sort	29.948	10000200000	276.697	90000600000	796.188	250001000000
Shaker Sort	32.535	10000050001	278.836	90000150001	885.81	250000250001
Shell Sort	0.004	4707128	0.013	20187386	0.037	26729067
Heap Sort	0.012	6087452	0.035	10446746	0.062	35135730
Merge Sort	0.012	5767897	0.038	18708313	0.066	32336409
Quick Sort	0.007	3219401	0.024	10567016	0.044	18332891
Counting Sort	0	700002	0.001	2100002	0.003	3500002
Radix Sort	0.006	1700125	0.019	6000150	0.035	10000150
Flash Sort	0.001	1090001	0.004	3270001	0.007	5450001



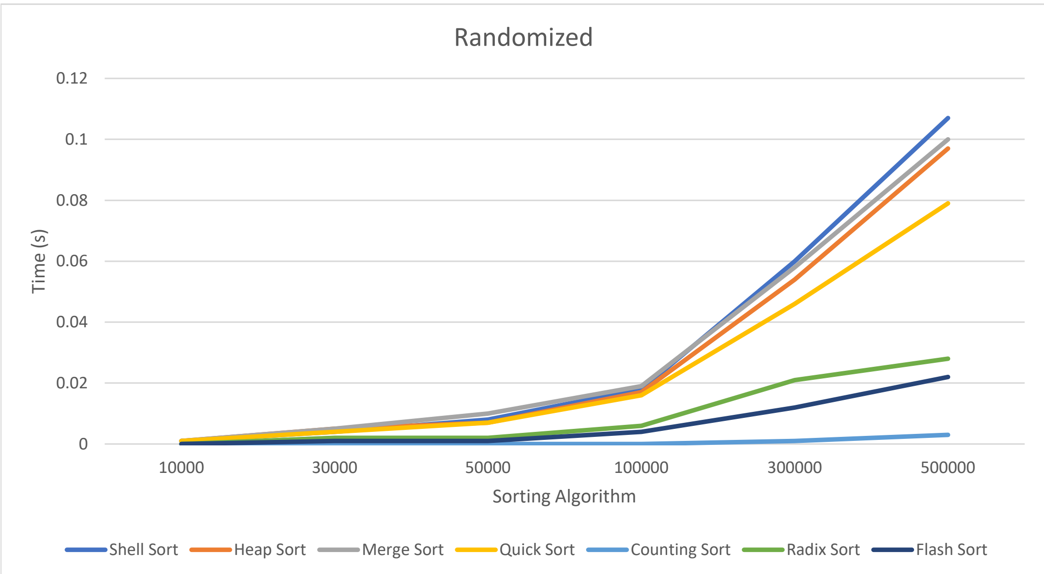
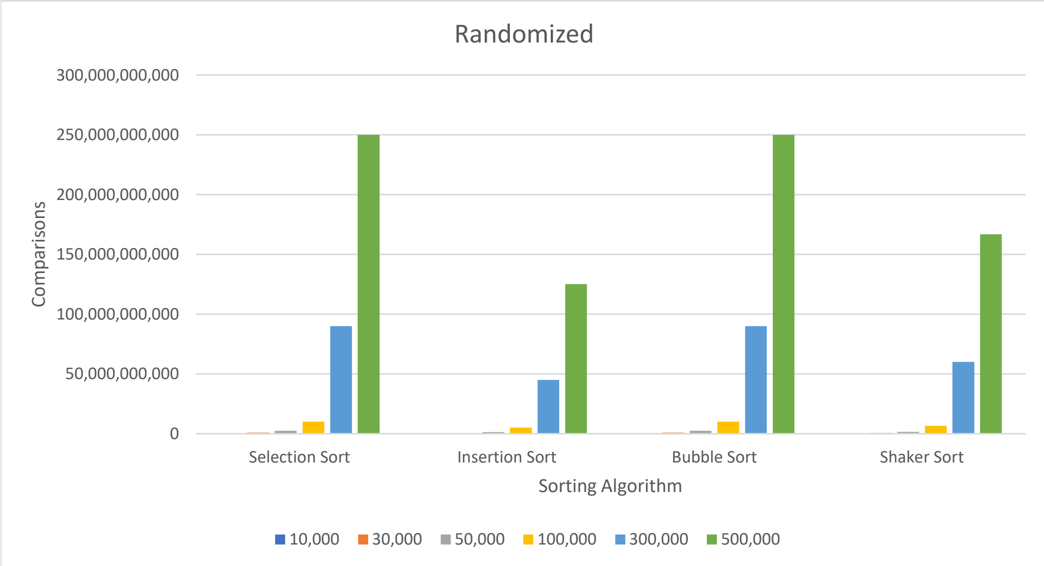


Figure 1: Time Complexity of Randomized Data



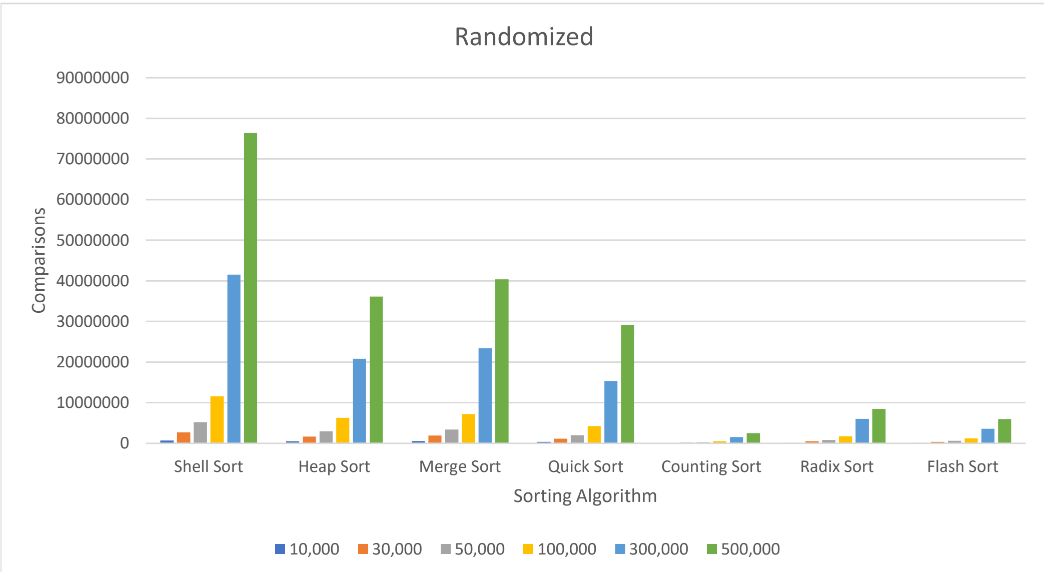
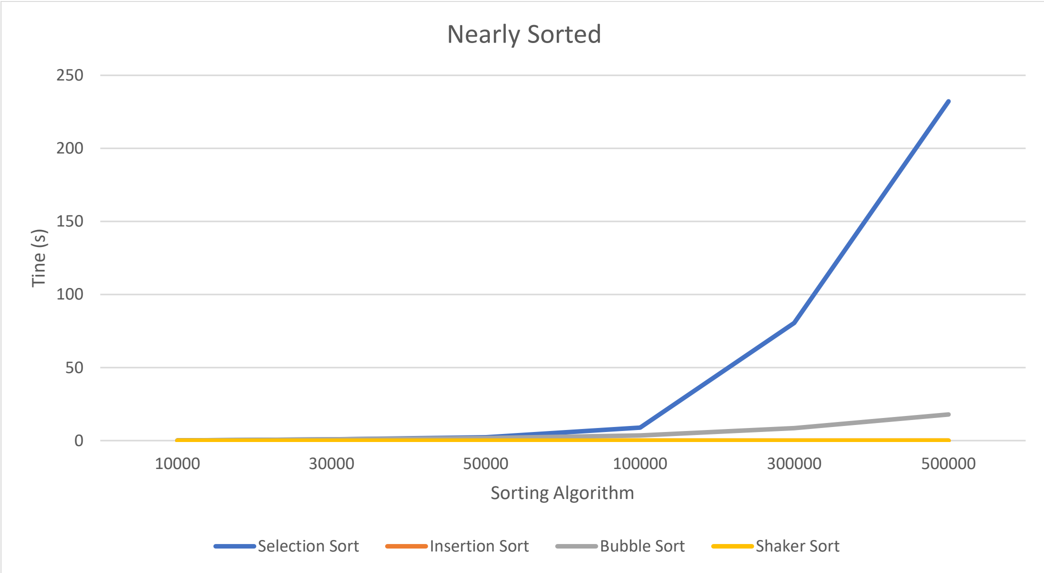


Figure 2: Comparisons of Randomized Data



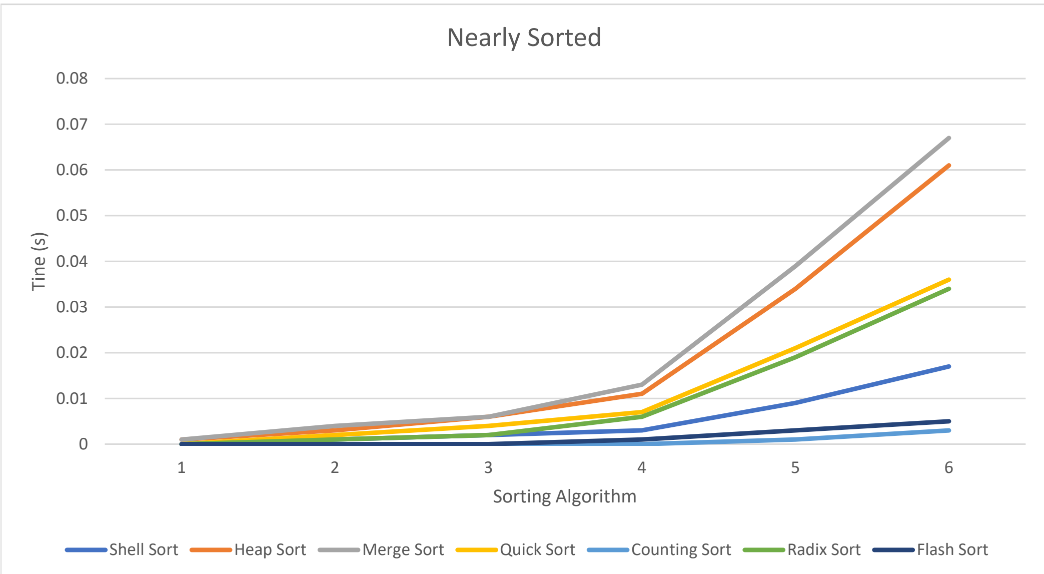
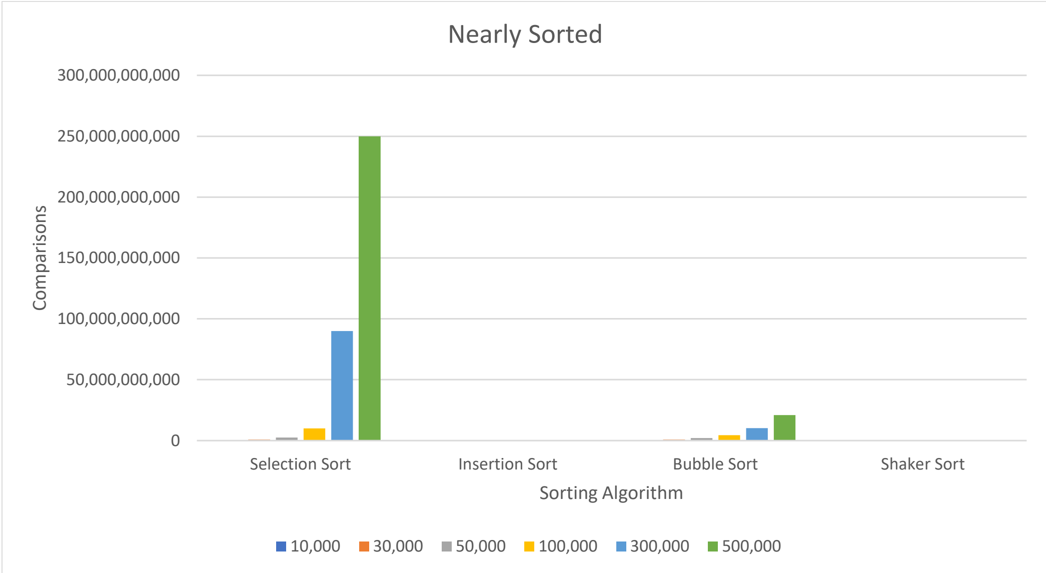


Figure 3: Time Complexity of Nearly Sorted Data



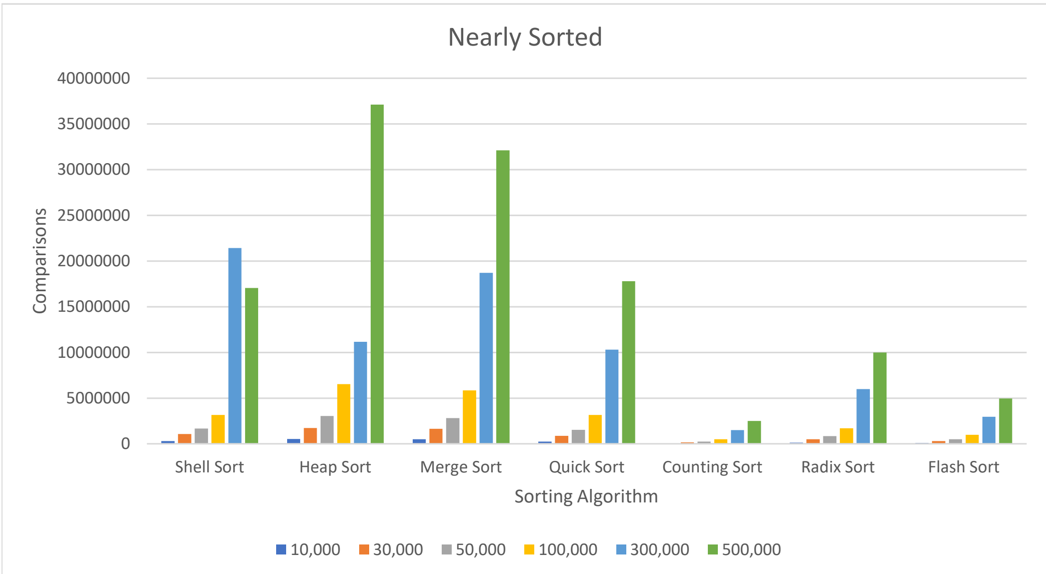
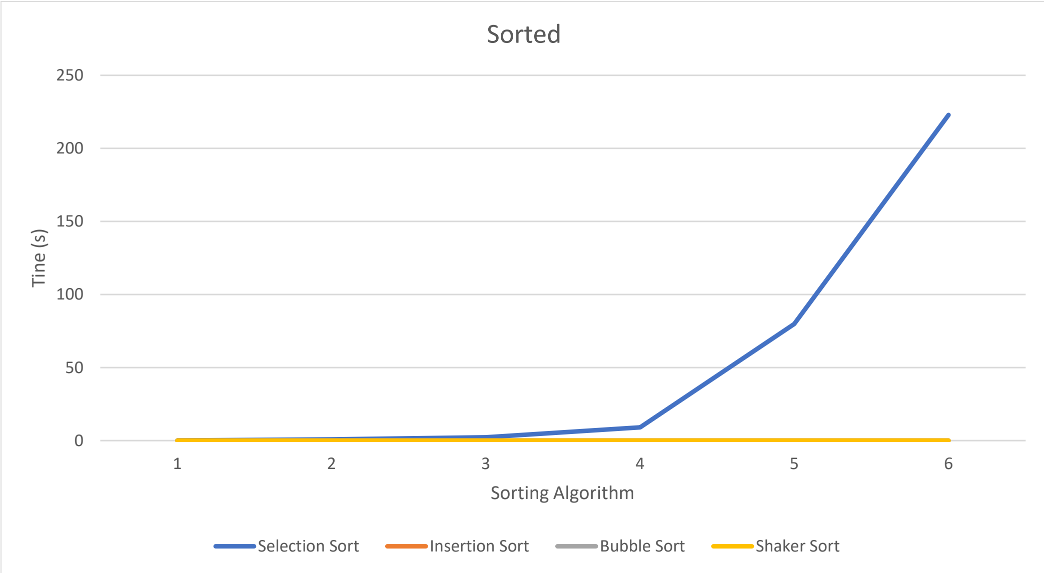


Figure 4: Comparisons of Nearly Sorted Data



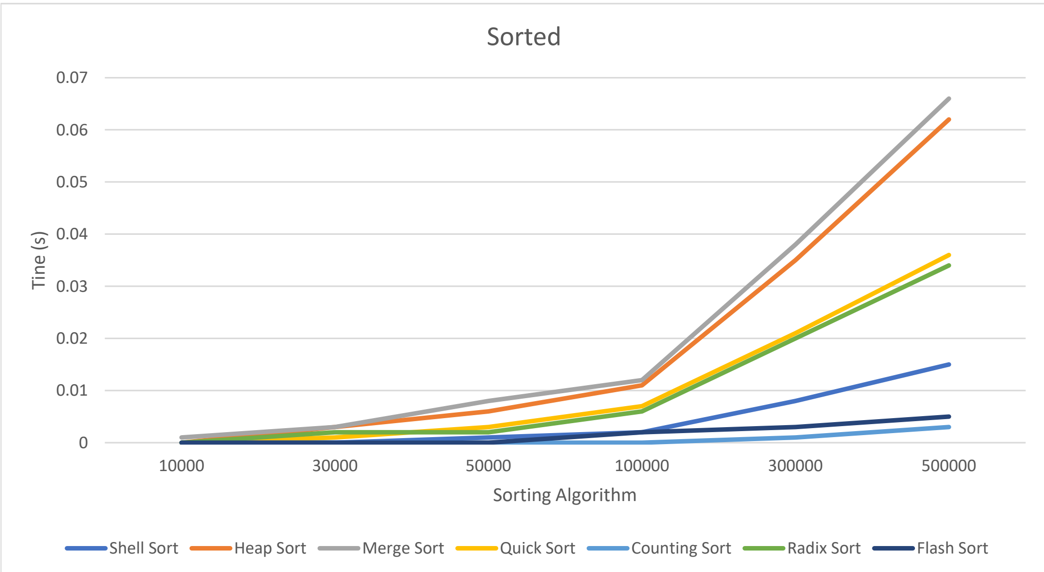
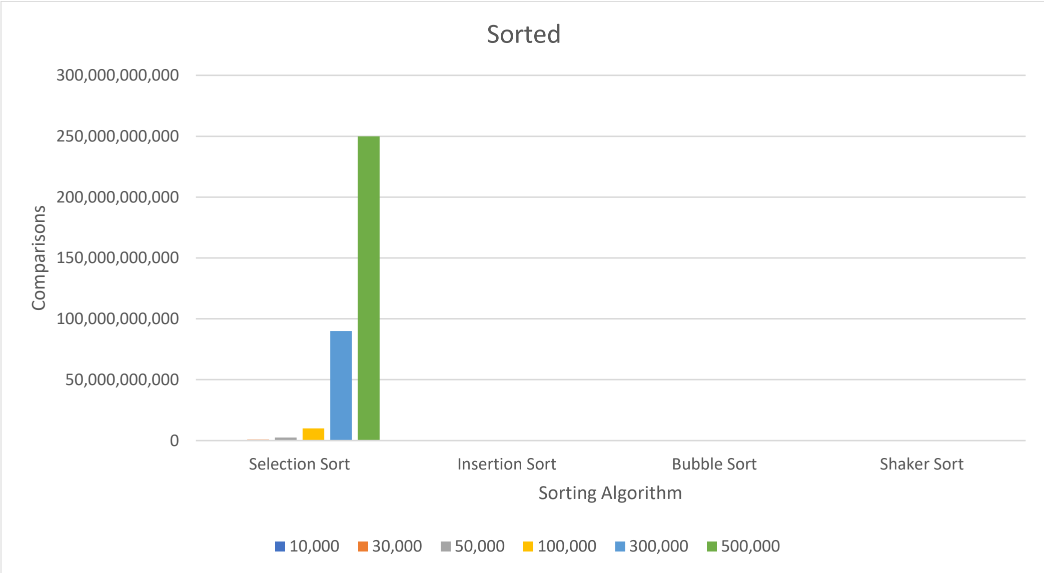


Figure 5: Time Complexity of Sorted Data



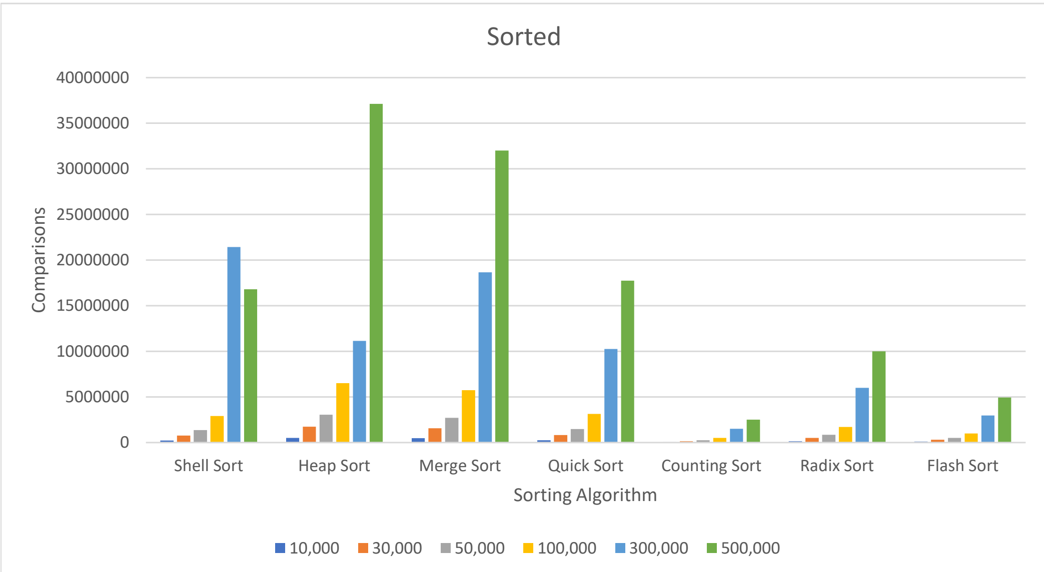
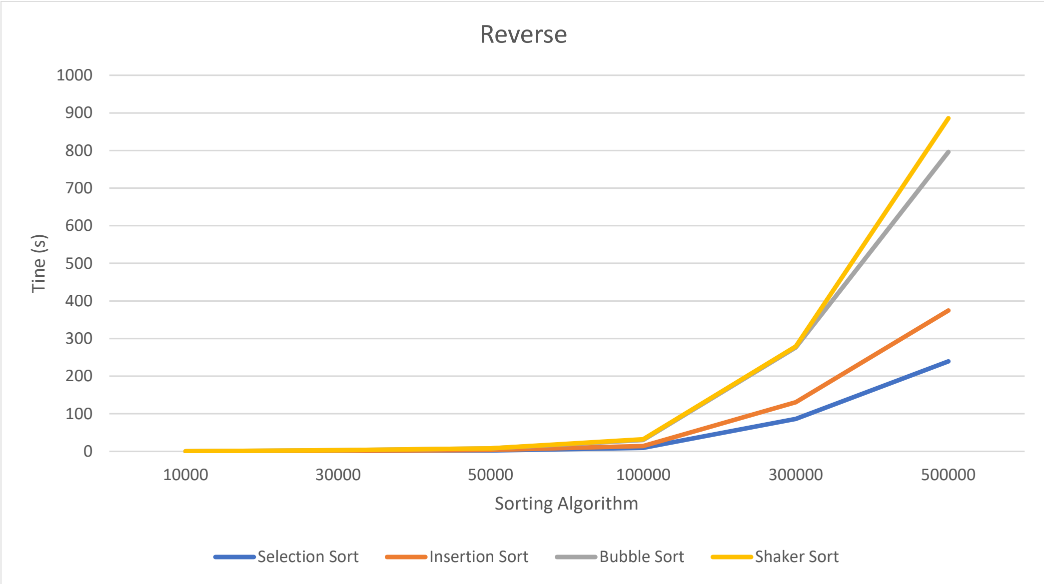


Figure 6: Comparisons of Sorted Data



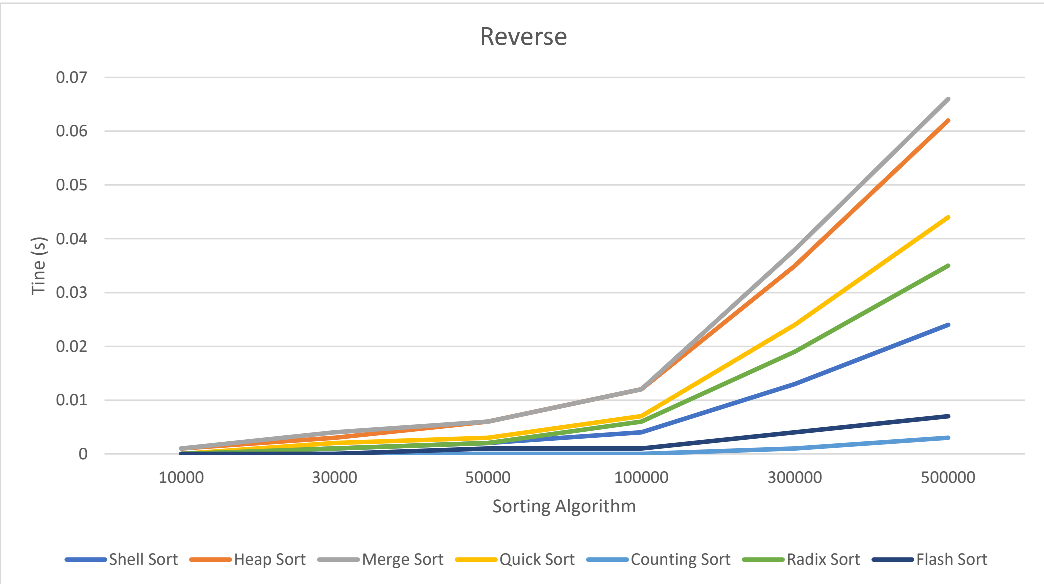
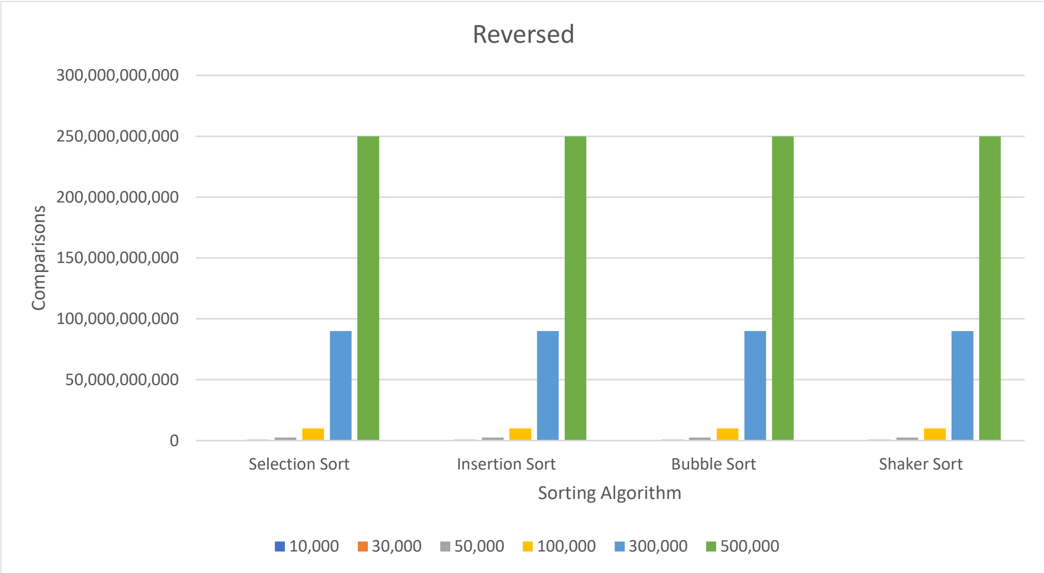


Figure 7: Time Complexity of Reverse Sorted Data



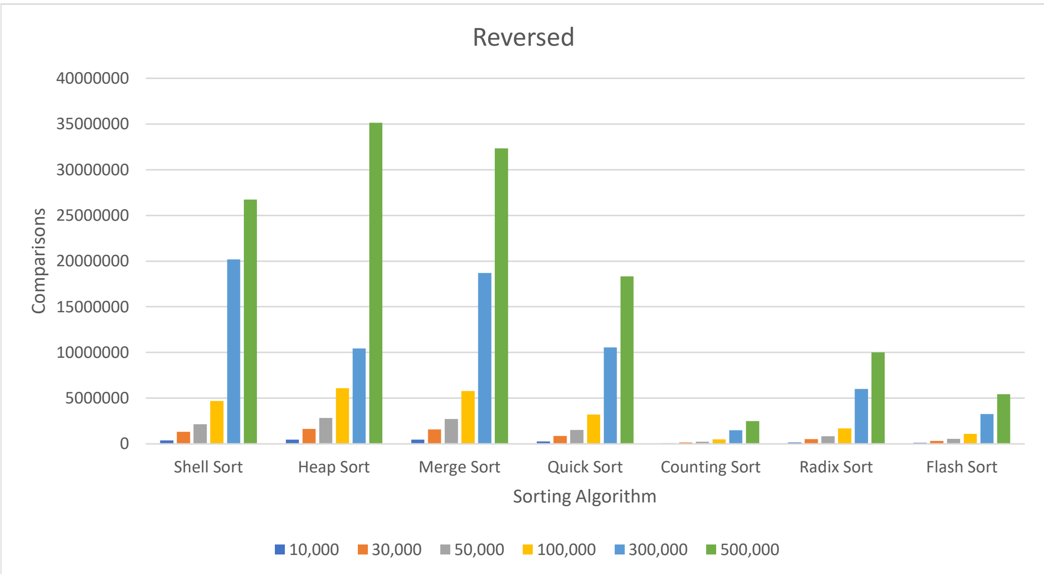


Figure 8: Comparisons of Reverse Sorted Data

4 Discussion

4.1 Randomized Data

As a result from figure 1 and 2 , we made some conclusions . The algorithms that take most of the comparisons are Selection sort and Bubble sort. On the other hand, The algorithm that takes the least comparisons is Counting Sort, it also takes the least time to sort, while Bubble sort and Shaker sort take most of the time. With more data (300k, 500k elements), the $O(n \log n)$ sorting algorithms were more efficient than the $O(n^2)$. Time is less than 1 second and comparisons are about $< 10^8$ while $O(n^2)$ algorithms, the runtime measured by minutes and comparisons is $> 10^{11}$. In conclusion, the $O(n)$ non-comparisons algorithms like Radix sort, and Counting sort are the fastest.

4.2 Nearly Sorted Data

From the illustrator of 3 and 4, it is easy to see that the algorithms which took the most comparisons are selection sort, while the least is Shaker sort and Insertion sort. On the other hand, Selection sort required the most time, contrary to Counting sort, Shaker sort, and Insertion sort. In conclusion, Insertion sort is very efficient in case the data is nearly sorted, with improvement, Shaker sort also performed well in this case.

4.3 Sorted Data

From the illustrator of 5 and 6, it is easy to see that the algorithms which took the most comparisons are Selection sort, while the least is Shaker sort and Insertion sort. On the other hand, Selection sort required the most time, contrary to Counting sort, Shaker sort, and Insertion sort. In conclusion, of course, Insertion sort is also very efficient in case the data is sorted. With improvement, Bubble sort, Shaker sort also performed well in this case (when there are no swaps and break the loop).

4.4 Reverse Sorted Data

As a result from figure 7 and 8 , we made some conclusions . The algorithms that take most of the comparisons are Selection sort, Bubble sort, Insertion sort, and Shaker sort. On the other hand, The algorithm that takes the least comparisons is Counting Sort, this methods and Flash sort also take the least time to sort, while Bubble sort and Shaker sort are the slowest ones.

5 Conclusion

- O(n) non-comparisons like Radix sort and Counting sort are the fastest in sorting non-negative integers. But if the range is large (for example the max-value is n^2) Counting sort is much slower than Radix sort.
- Flash sort gives us an advantage in sorting numbers (real numbers) over other algorithms because of the time complexity also O(n).
- Only use O(n^2) like Selection sort, Bubble sort,... for small data or simple testing because of its ease in implementation.
- If the data given is nearly sorted, Insertion sort is efficient (this is why it's used in the last part of Flash sort while the data is nearly sorted)
- Heap sort, Merge sort, Quick sort run with stable time and comparisons in all kinds of order
- Use algorithms with time complexity O($nlogn$) when the data is more complex than only numbers and the number of elements is large.
- There is no " perfect sorting algorithm ", depending on different cases and situation, we will choose the most suitable method.

References

[1] TutorialsPoint, "Data structure and algorithms selection sort." [Online]. Available: http://surl.li/cdyzc

[2] A. Zutshi and D. Goswami, "Systematic review and exploration of new avenues for sorting algorithm," *International Journal of Information Management Data Insights*, vol. 1, no. 2, p. 100042, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2667096821000355

[3] E. K. Donald *et al.*, "The art of computer programming," *Sorting and searching*, vol. 3, pp. 426–458, 1999.

[4] T.D.Thu, N.T.Phuong, D.B.Tien, T.M.Triet, D.B.Phuong, "Ky thuat lap trinh," pp. 380–382, 386–387, 2021.

[5] E. H. Friend, "Sorting on electronic computer systems," *J. ACM*, vol. 3, no. 3, p. 134–168, jul 1956. [Online]. Available: https://doi.org/10.1145/320831.320833

[6] R. C. Bose and R. J. Nelson, "A sorting problem," *J. ACM*, vol. 9, no. 2, p. 282–296, apr 1962. [Online]. Available: https://doi.org/10.1145/321119.321126

[7] S. NGUYEN, "Bubble sort và shaker sortt," https://www.stdio.vn/giai-thuat-lap-trinh/bubble-sort-va-shaker-sort-01Si3U, 2020, stdio.vn.

[8] E. K. Donald *et al.*, "The art of computer programming," *Sorting and searching*, vol. 3, pp. 84–95, 1999, shell sort.

[9] T. Hoang, "Giai thich thuat toan shell-sort." [Online]. Available: https://youtu.be/nMmWMbbGdmA

[10] G. E. Forsythe, "Algorithms," *Commun. ACM*, vol. 7, no. 6, p. 347–349, jun 1964. [Online]. Available: https://doi.org/10.1145/512274.512284

[11] T. Cormen and p. t. K. A. c. c. t. Devin Balkcom, "Analysis of merge sort." [Online]. Available: https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/analysis-of-merge-sort

[12] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, "Introduction to algorithmsm, chap. 16. greedy algorithms," pp. 28–29, 2001.

[13] J. Bentley, D. Knuth, and D. McIlroy, "Programming pearls: A literate program," *Communications of the ACM*, pp. 147–162, 1986.

[14] E. K. Donald *et al.*, "The art of computer programming," *Sorting and searching*, vol. 3, pp. 248–379, 1999.

[15] E. Emerald, "Work c++ algorithm of external natural merge sort with non-decreasing and decreasing ordered sub sequences," 2010. [Online]. Available: https://www.codeproject.com/Articles/92761/Work-C-Algorithm-of-External-Natural-Merge-Sort-wi

[16] T. Cormen and p. t. K. A. c. c. t. Devin Balkcom, "Analysis of quick sort." [Online]. Available: https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort

[17] youtuber Lap trinh cùng Eric, https://www.youtube.com/watch?v=CAaDJJUszvE, 2021, youtube.

[18] K.-D. Neubert, "The flashsort1 algorithm," *Dr. Dobb's Journal*, vol. 23, no. 2, pp. 123–129, 1998.

[19] E. K. Donald *et al.*, "The art of computer programming," *Sorting and searching*, vol. 3, pp. 168–179, 1999, shell sort.