University of Science

Viet Nam National University Ho Chi Minh

PROJECT REPORT
LAB 3: SORTING

DATA STRUCTURE AND ALGORITHM

Tutors: Nguyen Ngoc Thao
Bui Huy Thong

Group: 6

Members:

21127267 - Phan Van Ba Hai
21127556 - Do Quoc Tri
21127118 - Lam Thanh Ngoc
21127382 - Phu Thanh Nhan

Class: 21CLC02

Contents

1 Information

Set 2 (11 algorithms): Selection Sort, Insertion Sort, Bubble Sort, Heap Sort, Merge Sort, QuickSort, Radix Sort, Shaker Sort, Shell Sort, Flash Sort and Counting Sort.
The specifications of the computer used to run these algorithms are as follow:

- **CPU**: AMD RYZEN^{TM} 5 2600 6 Cores 12 Threads (3.4 Ghz 3.9 Ghz Turbo)
- **GPU**: HIS RX 460 2GB GDDR5
- **RAM**: 16GB 2400 MHz in dual-channel configuration (2x8)

1.1 Some small notes

The CPU while running the sorting algorithms always boost to 3.9GHz without any throttling to output a consistent result.

2 Introduction

There are 4 members in our group.

- 21127556 - Do Quoc Tri
- 21127118 - Lam Thanh Ngoc
- 21127382 - Phu Thanh Nhan
- 21127267 - Phan Van Ba Hai

Our work assignments are the following:

- Hai and Tri will take care of the coding part
- Nhan and Ngoc will in charge of the report

3 Algorithm presentation

3.1 Selection Sort

3.1.1 Ideas

The array is divided into two parts (sorted and unsorted). The smallest value selected from the unsorted part is swapped with the leftmost element.

The leftmost element is marked so that the sorted part is extended to the right, and the sort ends when the unsorted array contains one element.

3.1.2 Step-by-step descriptions

Step 1: Assign the INDEX as 0

Step 2: Find the element with minimum in the array

Step 3: Swap the founded minimum with the element at INDEX

Step 4: Increment INDEX to the next location

Step 5: Continue until the array is sorted

3.1.3 Complexity evaluations

Time complexity: In all cases, selection sort has to iterate over each of the 'n' elements to find the smallest number. One has to repeat 'n' times to sort out the array. Therefore, the overall time complexity is a quadratic running time ($O(n^2)$).

Space complexity: Selection sort is an in-place sorting algorithm, which means it does not need any auxiliary space. Hence, the space complexity is $O(1)$.

3.1.4 Variants

Double Selection Sort (or Cocktail Sort)

Ideas

- Both maximum and minimum are found after every pass and placed at their correct positions. The array is sorted from both ends.

Complexity evaluation

- The amount of main iterations are decreased by half, but the inner ones are doubled. Therefore, the complexity of the variant can be considered the same as the original selection sort ($O(n^2)$).

3.2 Insertion Sort

3.2.1 Ideas

Similar to selection sort, the array is also divided into two parts (sorted and unsorted).

Each element of the unsorted part is chosen and placed to the sorted one at the right position.

3.2.2 Step-by-step descriptions

Step 1: Assume that the first element is sorted.

Step 2: Store the second element into a separated variable KEY.

Step 3: Compare KEY with the first element. If KEY is smaller, swap them.

Step 4: Assign KEY as the next element.

Step 5: Continue until the array is sorted.

3.2.3 Complexity evaluations

Time complexity:

- Best case: When the array is already sorted. Then, this algorithm has a linear running time ($\Omega(n)$).
- Worst case: When the array is in the reversed order. Then, this algorithm has a quadratic running time ($O(n^2)$).
- Average case: In case the size of the array is too big to be executed, this algorithm is impractical. Nevertheless, insertion sort is one of the best choices for sorting the small-size array. Then, the running time is considered the same as the worst case ($\Theta(n^2)$).

Space complexity: Insertion sort is an in-place sorting algorithm, which means it does not need any auxiliary space. Hence, the space complexity is $O(1)$.

3.2.4 Variants

Shell Sort Shell sorting algorithm is presented in 3.9

3.3 Bubble Sort

3.3.1 Ideas

Bubble sort is the simplest sorting algorithm starting with the first element and swapping the two adjacent elements if they are in the wrong order.

3.3.2 Step-by-step descriptions

Step 1: Compare the first element with the others in the array. If the first one is smaller than any element, swap them so that they are rearranged to the right order.

Step 2: Move to the next element and continue until reaching the last one.

3.3.3 Complexity evaluations

Time complexity:

- Best case: when the array is already sorted. Then, this algorithm has a linear running time ($\Omega(n)$).
- Worst case: when the array is in a reversed order. Then, this algorithm has a quadratic running time ($O(n^2)$).
- Average case: the time complexity of the average case is considered being equal to the one of the worst case ($\Theta(n^2)$).

Space complexity: Bubble sort is an in-place sorting algorithm, which means it does not need any auxiliary space. Hence, the space complexity is $O(1)$.

3.3.4 Variants

One of the variants for bubble sorting algorithm is similar to the selection's one (which is also known as cocktail sort).

Odd - even sort (Brick sort)

Ideas

- Instead of comparing and swapping all pairs of adjacent elements as bubble sort, this algorithm compares and swaps adjacent odd (or even) elements if they are not in the right order.

Complexity evaluation

- The time complexity of this algorithm is considered similar to the original bubble sort's one.

3.4 Heap Sort

3.4.1 Ideas

Heap sort is an efficient algorithm for sorting using a data structure called binary heap (a specially ordered complete binary tree, in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible and stored values in a parent node is greater (or smaller) than the values in its two children nodes).

3.4.2 Step-by-step descriptions

Step 1: Build a max (or min) heap (stored values in a parent node is greater (or smaller) than the values in its two children nodes).

Step 2: Replace the root storing the largest (or smallest) element with the last one of the heap. The size of the heap then is reduced by 1.

Step 3: Reshape the binary tree into the heap data structure (also known as ”heapify”).

Step 4: Iterate step two and three until the heap has one 1 element.

3.4.3 Complexity evaluations

Time complexity:

- Similar to selection sort algorithm, the running time of heap sort algorithm does not depend on distribution of data, so the running time is the same for all cases (best case, worst case and average case).
- For heap's initialization, the process called ”heapify” has a logarithmic running time ($O(logn)$).
- In heap sort algorithm, ”heapify” is called for each parents' node reversedly (starting with the last node and ending with the root of the binary tree).
- Hence, heap sort algorithm has a linearithmic time complexity in overall ($O(nlogn)$).

Space complexity: Heap sort is an in-place sorting algorithm, which means it does not need any auxiliary space. Hence, the space complexity is $O(1)$

3.4.4 Variants

Bottom-up heapsort

Ideas

- The comparison are made between the two children and the parent follows the larger one downward the tree. There, the algorithm goes backward to the root and investigate for the first element which is larger than the root and the root is placed in a free field.

Complexity evaluation

- Best case:$n * log(n) + O(n)$
- Worst case: bounded from above by $\frac{3}{2n}logn + 0(n)$
- Average case: $n * log(n) + O(n)$

3.5 Merge Sort

3.5.1 Ideas

Merge sort is considered to be one of the most efficient sorting algorithms by many. the whole concept of merge sort is established on the strategy of divide and conquer. The algorithm divide the array in two halves, sorts each of them and then merge them together like the name suggest. The most important part is in the process of merging two halves together.

3.5.2 Step-by-step descriptions

Step 1: Find the middle index of the array
Step 2: Split the array from the middle index
Step 3: Merge sort the second half of the array
Step 4: Merge sort the first half of the array
Step 5: Merge the two sorted arrays into one

3.5.3 Complexity evaluations

Time complexity Merge sort time complexity is the same for not only the best and worst case but also average case, which is $(O(nlogn))$

Space complexity The space complexity is $(O(n))$

3.5.4 Variants

There are a lot of variants of merge sort, but some of the most common variations are: bottom-up and top-down

3.6 Quick Sort

3.6.1 Ideas

Quick sort is a divide and conquer sorting algorithm (for it is also called part sort). The array is divided into two parts in comparison with an element called pivot continuously till the subarrays has one element. The array is sorted quickly by executing recursively.

3.6.2 Step-by-step descriptions

Step 1: Choose the pivot from the array (by partition process).

Step 2: Divide the array into two parts in comparison with the pivot (the array consists of elements smaller than the pivot and the other contains elements larger than the pivot).

Step 3: Execute step 2 repeatedly until the subarray has only one element.

Step 4: Recursively execute till the array is sorted.

3.6.3 Complexity evaluations

Time complexity

- Best case: when the middle element is chosen as a pivot. Then, the algorithm has a linearithmic running time complexity ($\Omega(nlogn)$).
- Worst case: when the element with largest (or smallest) value is chosen as a pivot. Then, the algorithm has a linear running time complexity ($O(n)$).
- Average case: when all possible permutations of the array are considered. In this case, the running time is also linerithmic ($\Theta(nlogn)$).

Space complexity

- In-place partitioning is used. This unstable partition requires $O(1)$ space.
- After partitioning, the partition with the fewest elements is (recursively) sorted first, requiring at most $O(logn)$ space.

3.6.4 Variants

The variants for quick sorting algorithm depends on the way pivot is chosen.

3.7 Radix Sort

3.7.1 Ideas

Radix sort is non-comparison based soritng algorithm. The elements in the array is divided into buckets digit by digit until all the digits are considered.

3.7.2 Step-by-step descriptions

Step 1: Find the number of digits of the maximum element.

Step 2: Sort the elements based on the digit present at the current significant place using counting sort.

Step 3: Continue until the array is sorted.

3.7.3 Complexity evaluations

Time complexity

- Let's look at an array with d amount of integers digits. The Radix Sorting algorithm time to finish is calculate using the following formula: O(d*(n+b)), with b stands for the base for representing numbers

Space complexity

- Assume n is the number of elements in the array and k is the largest element among the d^{th} place elements.
- The auxiliary arrays are used. Hence, the space complexity of radix sort is $O(n + k)$

3.7.4 Variants

Instead of sorting the 2D array, using radix sort with linked list can save more memory space.

3.8 Shaker Sort

Also known as cocktail shaker sort, bidirectional bubble sort but here we will use shaker sort for clarity

3.8.1 Ideas

The algorithm is based on bubble sort and improve on it by traversing the whole array in both directions. Which in turn eliminate the uneccessary iterations in large arrays

3.8.2 Step-by-step descriptions

The steps can be divided into 2 phases

The first stage iterate from left to right

Step 1: Elements standing next to each other are compared.

Step 2: if the value on the left is greater than the one the right then the two are swapped.

The second stage iterate from right to left

Step 3: Elements standing next to each other are compared.

Step 4: if the value on the left is smaller than the one the right then the two are swapped.

3.8.3 Complexity evaluations

Time complexity:

- Best case: Just like bubble sort, the best possible case for shaker sort is when the array is nearly or already sorted ($\Omega(n)$).
- Average case: The average case for shaker sort is quite similar to its worst case which is ($O(n^2)$).
- Worst case: if the array is in reversed to the direction of the shaker sort then the time complexity algorithm will be ($\Theta(n^2)$).

Space complexity: Shaker sort space complexity is $O(1)$ because we use the same amount of extra memory not including the input array.

3.8.4 Variants

There are up until now no variants of shaker sort.

3.9 Shell Sort

3.9.1 Ideas

Just like bubble sort and its improved algorithm, shaker sort. Insertion sort can also be improved, the result is shell sort.

In insertion sort, we can only move elements one position away. To improve on this, shell sort move elements far from each other first and then gradually redcues the interval between the elements to be sorted.

3.9.2 Step-by-step descriptions

Step 1: Initialize the value for the interval.

Step 2: Divide the array into smaller ones with the appropriate interval.

Step 3: Rearrange subarrays using insertion sort algorithm.

Step 4: Continue until the array is sorted.

3.9.3 Complexity evaluations

Time complexity:

- Best case: when the array is already sorted. Then, the running time complexity is linearithmic ($\Omega(nlogn)$).
- Average case: when the array is randomized which is in jumbled order. Then, the running time complexity is equivalent to the best case ($O(nlogn)$).
- Worst case: when the array is in reversed order. Then, the running time complexity is quadratic ($\Theta(n^2)$).

Space complexity: Shell sort space complexity is $O(1)$

3.9.4 Variants

There are two variants of shell sort can be considered (Dobosiewicz sort and Shaker sort).

3.10 Counting Sort

3.10.1 Ideas

Counting sort sorts elements of an array by **counting** the occurances of each of them. The elements are sorted by mapping as the index of the count stored in an auxiliary array.

3.10.2 Step-by-step descriptions

Step 1: Find the maximum element in the array and its size.

Step 2: Initialize an auxiliary array with the size of the max added 1 and fill it with 0.

Step 3: At each index of the auxiliary array, the corresponding value of the element in the source array is calculated.

Step 4: Calculate the cumulative sum by adding the current and previous frequency to the auxiliary array, which presents the exact position of the element in the sorted array.

Step 5: Iterate the auxiliary array from 0, if the element is existed in the source array, minus 1 at the equivalent index in the auxiliary one.

Step 6: Continue until the array is sorted.

3.10.3 Complexity evaluations

Time complexity

- Best case: when all elements are in the same range, or when the times to preserve the auxiliary array is equal to 1. Then, the running time is linearithmic ($\Omega(n)$).
- Average case: times to preserve the auxiliary array (also known as k) is taken variously from 1. Fixed number of elements of the source array presents that both n and k are equally dominating. Then, the running time is $O(n + k)$.
- Worst case: when the maximum element is much larger than the others. Since there is no boundary for times to preserve the auxiliary array, the higher it is, the worse the complexity is.

Space complexity

- The size of the auxiliary array is the size of the maximum element in the source array. Therefore, the space complexity is $O(k)$.

3.10.4 Variants

Counting sort is not an in-place algorithm but it is stable. However we can modify the algorithm in order that it places the items into sorted order within the same array that was given to it as the input, but will result in being unstable

3.11 Flash Sort

3.11.1 Ideas

Flash sort is an efficient in-place sorting algorithm which assigns elements in the array to the appropriate initialized buckets, and sorts them within each bucket. The performance of any buckets sort heavily depends on the **balance** of the buckets.

3.11.2 Step-by-step descriptions

Step 1: Find the minimum and maximum elements in the array.

Step 2: Divide the array into buckets.

Step 3: Calculate the number of elements for each bucket.

Step 4: Use permutation to classify the elements to their right bucket.

Step 5: Sort elements in each bucket using insertion sort.

3.11.3 Complexity evaluations

Time complexity

- Best case: The best case for flash sort occurs when we have an ideal case of a balanced data set, which means each buckets have roughly the same size. Then the run time is $(O(n))$.
- Average case: The average case for this algorithm is partly balanced which is also $(O(n))$.
- Worst case: the complexity of the algorithm is limited by the performance of the final bucket-sorting method. Therefore, the running time is quadratic $(O(n^2))$.

Space complexity

- The space complexity is $O(n)$ for only an auxiliary array is used. If the stability is required, another sub-array is possible to be used.

3.11.4 Variants

From what we've research, there aren't any variants of this algo

4 Experimental results and comments

4.1 Experimental results

4.1.1 Sorted data (in ascending order)

	Sorted data (in ascending order)											
Data size	10000		30000		50000		100000		300000		500000	
Resulting statics	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection Sort	119	99999999	1068	899999999	2944	2499999999	11731	9999999999	105411	89999999999	292593	249999999999
Insertion Sort	0	9999	0	29999	0	49999	0	99999	1	299999	2	499999
Bubble Sort	127	99999999	1139	899999999	3163	2499999999	12572	9999999999	113939	89999999999	315823	249999999999
Heap Sort	2	504005	9	1697716	16	22985913	34	6375398	113	21011202	196	36433362
Merge Sort	6	506652	20	1694748	34	2947020	69	6244044	213	20367948	380	35438188
Quick Sort	1	163613	2	537229	3	934461	7	1968925	21	6375709	35	10975709
Radix Sort	2	140038	7	510048	13	850048	26	1700048	93	6000058	161	10000058
Shaker Sort	0	19999	0	59999	0	99999	0	199999	1	599999	1	999999
Shell Sort	0	120018	1	390021	2	700021	4	1500022	14	5100026	22	8500025
Counting Sort	0	89995	0	269995	1	449995	2	899995	8	2699995	15	4499995
Flash Sort	0	20999	1	62999	2	104999	4	209999	12	629999	21	1049999

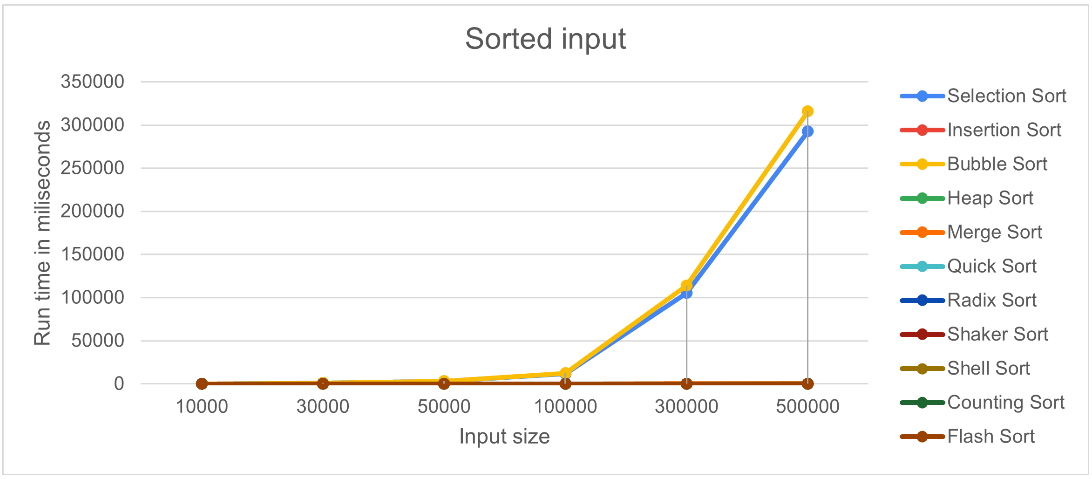

Figure 1: A line graph for visualizing the algorithms' running times on sorted data.

From what we can see, bubble sort and selection sort take the most amount of time out of all other algorithms when we gradually increases the size of the input data. The time it takes increase exponentially. This is due to the nature of these two sorting algorithms still trying to sort the input while the input itself is already sorted.

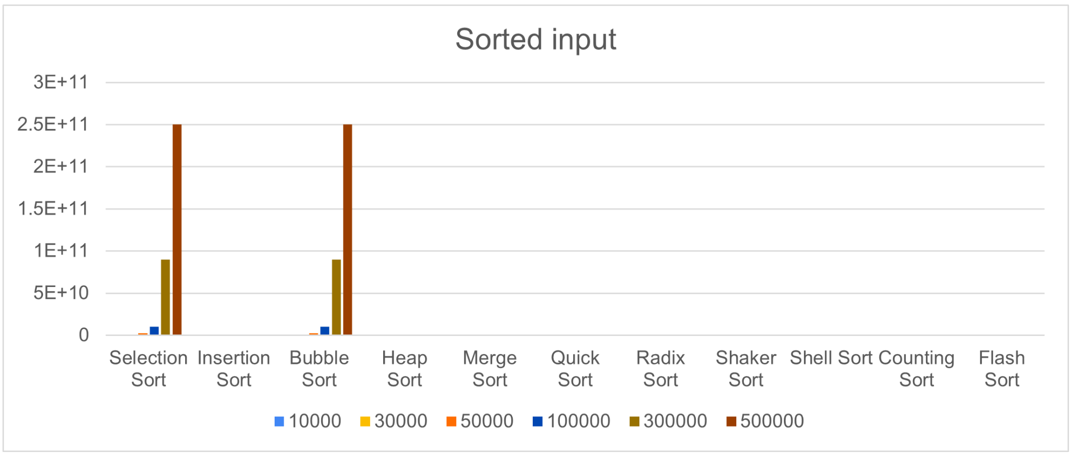

Figure 2: A bar graph for visualizing the algorithms' comparisons on sorted data.

We can see a familiar trend when comparing comparisons, bubble sort and selection sort still take the lead in the most amount of the comparisons. The reason for this is that bubble and selection are comparison-based sorting algorithms, but insertion sort is also comparison-based then why does it take very little comparisons. When the algorithm is already sorted, insertion sort doesn't need to move elements around very much.

4.1.2 Nearly sorted data

	Nearly sorted data											
Data size	10000		30000		50000		100000		300000		500000	
Resulting statics	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection Sort	119	99999999	1072	899999999	2946	2499999999	11668	9999999999	105220	89999999999	293361	249999999999
Insertion Sort	1	178607	1	403703	1	503547	1	448567	2	780479	2	1090751
Bubble Sort	128	99999999	1150	899999999	3177	2499999999	12637	9999999999	113604	89999999999	314298	249999999999
Heap Sort	3	504035	9	1697754	16	2985884	34	6375400	122	21011315	208	36433444
Merge Sort	7	572353	20	1878876	34	3167875	69	6411694	209	20554939	358	35631995
Quick Sort	1	163613	2	537229	3	934461	6	1968925	22	6375709	36	10975709
Radix Sort	2	140038	7	510048	13	850048	27	1700048	92	6000058	157	10000058
Shaker Sort	1	212133	2	461240	2	680143	3	916171	6	1555945	6	2291317
Shell Sort	1	137996	2	468005	3	768917	5	1581068	15	5186402	25	8564843
Counting Sort	0	89995	1	269995	1	449995	2	899995	9	2699995	14	4499995
Flash Sort	0	20983	1	62989	2	104985	5	209985	13	629987	22	1049987

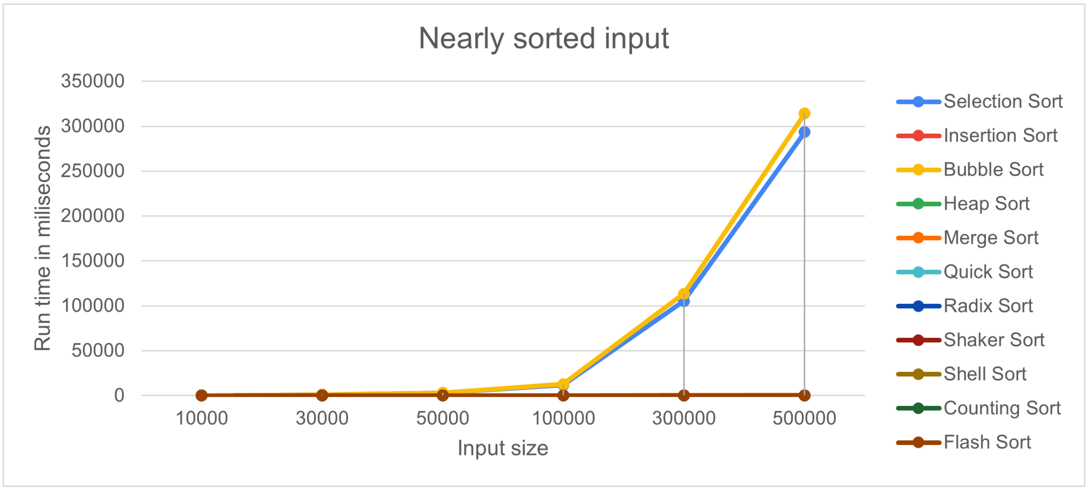

Figure 3: A line graph for visualizing the algorithms' running times on nearly sorted data.

Nearly sorted data is almost the same as sorted data with a little change is that insertion sort needs a bit more comparison due to the nature of the nearly sorted data.

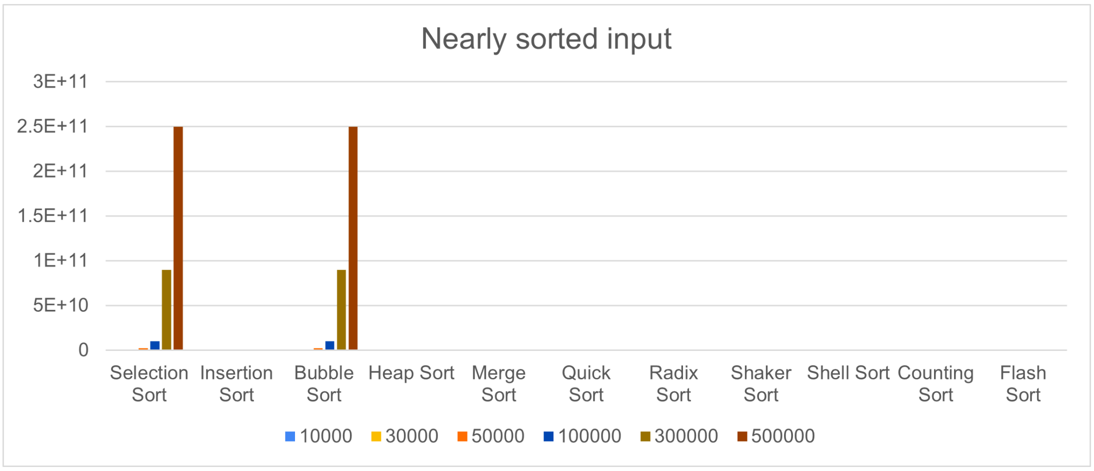

Figure 4: A bar graph for visualizing the algorithms' comparisons on nearly sorted data.

Although the graph does not portray the changes in insertion sort, looking at the table provided, we can see that there is a difference comparing sorted and unsorted data from insertion sort.

4.1.3 Reversed sorted data

	reversedd sorted data											
Data size	10000		30000		50000		100000		300000		500000	
Resulting statics	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection Sort	125	99999999	1086	899999999	3062	2499999999	12161	9999999999	108994	89999999999	302903	249999999999
Insertion Sort	136	99999999	1230	899999999	3387	2499999999	15163	9999999999	168939	89999999999	276609	249999999999
Bubble Sort	353	99999999	1799	899999999	5028	2499999999	20089	9999999999	181257	89999999999	741422	249999999999
Heap Sort	3	468401	9	1596468	15	2807198	32	6003393	111	19931637	190	34706560
Merge Sort	7	534251	20	1757643	35	3074779	70	6499563	212	21143211	362	36301067
Quick Sort	1	163612	2	537228	4	934460	6	1968924	20	6375708	35	10975708
Radix Sort	2	140038	7	510048	13	850048	26	1700048	94	6000058	156	10000058
Shaker Sort	276	100004999	1409	900014999	3937	2500024999	15731	10000049999	141924	90000149999	580533	250000249999
Shell Sort	1	182578	4	597013	7	1097301	16	2344582	49	7600906	87	12928777
Counting Sort	0	89995	1	269995	1	449995	3	899995	7	2699995	12	4499995
Flash Sort	0	17503	1	52503	2	87503	4	175003	13	525003	21	875003

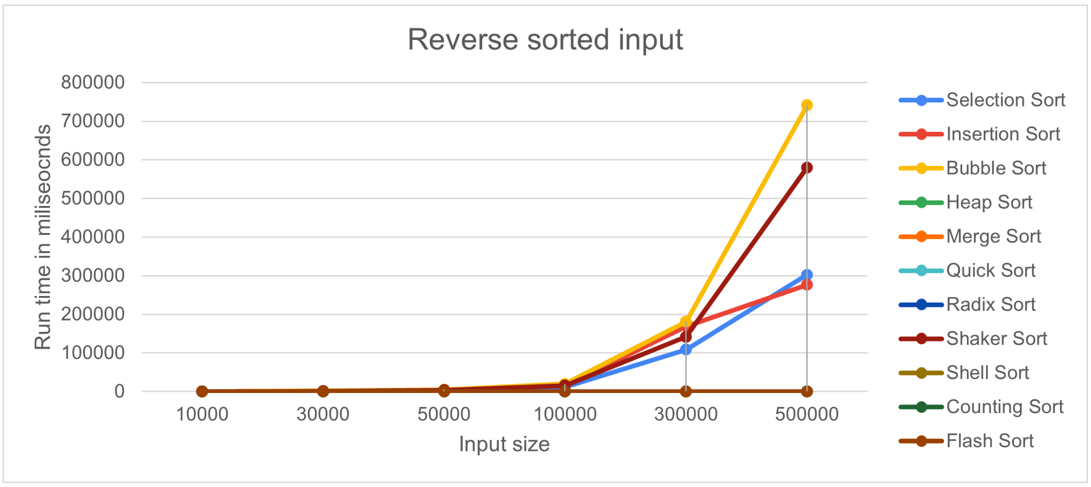

Figure 5: A line graph for visualizing the algorithms' running times reversed on sorted data.

Looking at the line graph, we can see a lot of changes from the previous two graph. Bubble sort is still the longest one. However shaker sort now takes the second place from selection sort. Most notably, selection sort is now almost as fast merge sort. We can understand why shaker sort takes a sudden change, if every element is at a position that differs by at most k (k 1) from the position it is going to end up in, the complexity of cocktail shaker sort becomes ($O(kn)$).

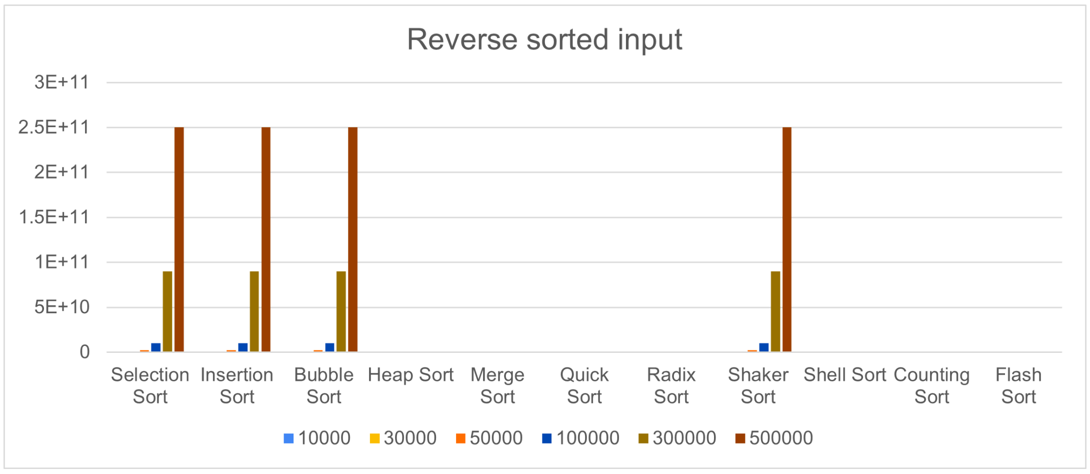

Figure 6: A bar graph for visualizing the algorithms' comparisons on reversed sorted data.

Here we can see insertion and shaker sort in their worst-case scenario with a high number of comparisons.

4.1.4 Randomized data

	Randomized data											
Data size	10000		30000		50000		100000		300000		500000	
Resulting statics	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection Sort	121	99999999	1064	899999999	2948	2499999999	11740	9999999999	105523	89999999999	292372	249999999999
Insertion Sort	66	49836639	607	448748189	1697	1248310235	6763	4986397423	95985	45016486265	337577	125071744873
Bubble Sort	293	99999999	2165	899999999	6298	2499999999	26354	9999999999	224869	89999999999	793164	249999999999
Heap Sort	3	485564	10	1647028	18	2894853	37	6189279	124	20451193	217	35542359
Merge Sort	7	734487	23	2465192	38	4312281	76	9173712	231	30129475	397	52260910
Quick Sort	2	203030	4	692944	7	1213732	14	2473158	42	7807295	71	13352871
Radix Sort	2	140038	7	510048	13	850048	25	1700048	74	6000058	122	8500048
Shaker Sort	229	67132541	1695	597243042	4931	1662711596	20635	6676026886	176082	60045803655	621047	166766760719
Shell Sort	3	270047	11	980977	22	1843153	53	4310589	186	14978000	329	26869564
Counting Sort	0	89993	0	269995	1	415525	2	765531	7	2165531	11	3565531
Flash Sort	0	16661	1	51305	3	74375	5	99999	17	299999	28	499999

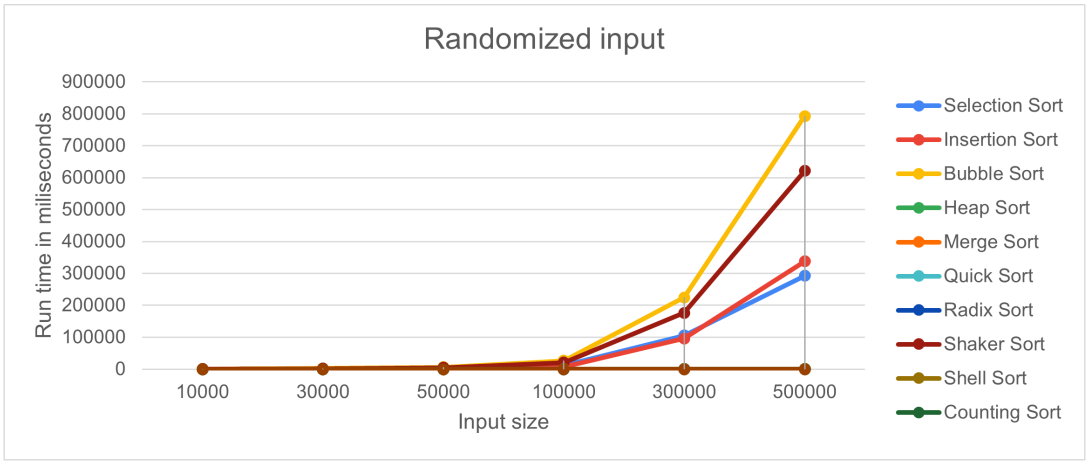

Figure 7: A line graph for visualizing the algorithms' running times on random data.

The same trend from figure 5 can be seen here with a slight deviation as merge sort takes a longer amount of time than selection sort. We can't explain this change in order, it may due to the randomized nature of the array. Because we've only run through two passes so there will be some differences compared to running a lot of passes.

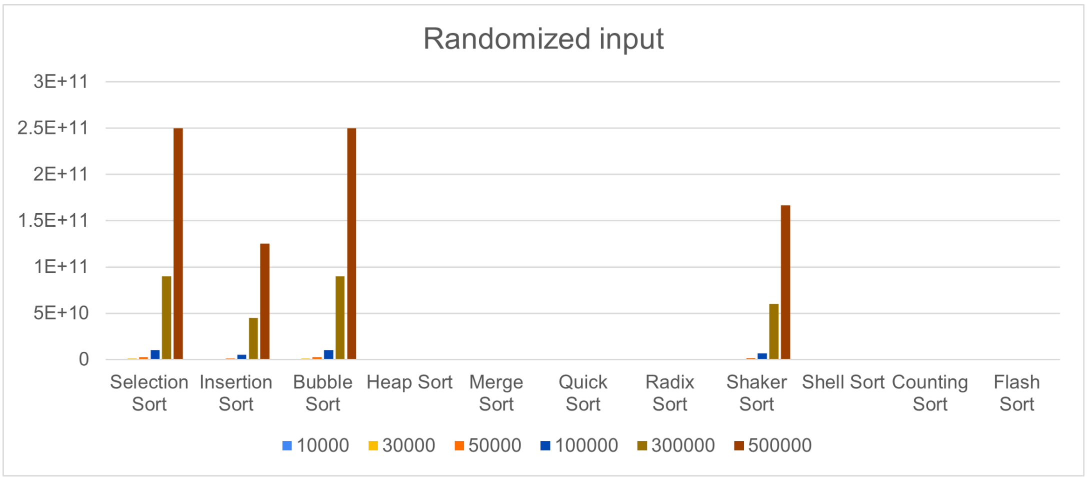

Figure 8: A bar graph for visualizing the algorithms' comparisons on sorted data.

Shaker and insertion sort number of comparisons have now reduced by quite a significant amount of time. This turnaround is because the data order is now randomized.

4.2 Comments

Overall, from all four data orders, counting sort and flash sort both is the fastest sorting algorithm based on running time and comparisons. With bubble sort being the slowest
Stable sorting algorithms:

- Insertion sort
- Bubble sort
- Merge sort
- Radix sort
- Counting sort

Unstable sorting algorithms:

- Selection sort
- Heap sort
- Quick sort
- Shaker sort
- Shell sort
- Flash sort

5 Project organization and Programming notes

5.1 Project organization

Library used (Allowed library for course):

- We use time.h to measure the time instead of chrono.h
- fstream is used for file input and ouput
- iomanip is for setprecision()
- string and cstring both for string data type manipulation
- iostream for outputing to the console

Main file and subordinates:

- DataGenerator.cpp: Data generation supplied by Mr. Bui Huy Thong
- header.h: Used to store all of our headers
- main.cpp: The program main source code
- Hai 21127267.cpp: Merge(), quick(), heap(), flash(), shaker() and shell() sort functions
- Nhan 21127382.cpp: Swap(), bubble(), insertion(), selection() sort functions
- Tri 21127556.cpp: The heart of our program, contains, file in out operations, printing to screen and measuring time and comparisons

5.2 Programming notes

We've included a -help function to show all functions of the program for ease of use!

This source code should be compiled in Visual Studio 2019 on Windows 10 for compatibily reasons.

6 References

Information on Selection Sort
Selection Sort complexity
Double Selection Sort complexity Information of Shell Sort
Ideas of Shell Sort
Buble Sort and Shaker Sort complexity
More on bottom up heapsort time complexity
Information of heap sort
Idea of Bottom Up Heap Sort
Information of Bottom Up Heap Sort
Binary Tree
Heap Sort complexity
Heap Sort Variants
Time and space complexity for all sorting algorithms and replacing best and worst notation with theta and omega respectively
Explanation for line graph of reversed sorted data
Information of Quick Sort
Variants of Quick Sort
Information of Radix Sort
Heap Sort complexity
Variants of Radix Sort
Information of Shell Sort
Variants of Counting Sort

We appreciate your reading!