

**VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY**  
**UNIVERSITY OF SCIENCE**  
**FACULTY OF INFORMATION TECHNOLOGY**



# **DATA STRUCTURES & ALGORITHMS**

## **SORTING ALGORITHMS PROJECT**

**Group: 10**

**Instructors:**

Mr. Bùi Huy Thông

Mrs. Trần Thị Thảo Nhi

**Group members:**

20127169 – Phạm Huy Hoàng

20127458 – Đặng Tiến Đạt

20127566 – Hoàng Quốc Nam

20127637 – Võ Lê Anh Thôn

# CONTENTS

<b>I. INTRODUCTION .....</b>	<b>3</b>
<b>II. INFORMATION .....</b>	<b>4</b>
<b>III. ALGORITHM PRESENTATION.....</b>	<b>5</b>
1. Selection Sort .....	5
2. Insertion Sort .....	7
3. Bubble Sort .....	9
4. Heap Sort .....	12
5. Merge Sort .....	15
6. Quick Sort.....	18
7. Radix Sort .....	20
8. Shaker Sort .....	21
9. Shell Sort.....	24
10. Counting Sort.....	27
11. Flash Sort.....	30
<b>IV. EXPERIMENTAL RESULT &amp; COMMENTS.....</b>	<b>32</b>
1. Randomized data .....	32
2. Nearly sorted data.....	35
3. Sorted data.....	37
4. Reverse sorted data.....	40
<b>V. ANALYSIS .....</b>	<b>43</b>
1. Algorithms with $O(n^2)$ complexity .....	43
2. Algorithms with other complexity.....	43
3. Stable sorting algorithms.....	43
4. Unstable sorting algorithms.....	44
<b>VI. PROJECT ORGANIZATION AND PROGRAMMING .....</b>	<b>45</b>
<b>VII. REFERENCES.....</b>	<b>47</b>

# I. INTRODUCTION

First of all, Group 10 would like to send our sincere thanks to the University of Science have brought Data Structures & Algorithms to the education program. Specially, we would like to thank deeply our practice lecturers & instructors – *Mr. Bùi Huy Thông* and *Mrs. Trần Thị Thảo Nhi*, who have taught and imparted valuable knowledge to us during school time. During studying times, we have improved ourselves with many useful skills, serious and effective – learning spirit. This will definitely be valuable knowledge, plant the seed for us to go out into the wider world.

After this mid – term project, we, Group 10, have improved a lot about communication skills, time management skills and problem – solving. So precious that we are learnt how to work like a team, listen to our partner, ask for helps and share difficulties with each other. Besides, acquiring knowledge by search on the web is one of the necessary skills needed in the Faculty of Information Technology.

Data Structures & Algorithms is an interesting subject, very useful with high practicality. Guaranteed to provide enough knowledge, associated with the actual demand of students. However, because of our limited knowledge and receptive ability, even we have tried our best but certainly, our project is hard to avoid deficiencies and mistakes, we are hope that our report is considered and feedback to make our project more complete.

Sincerely yours.

## II. INFORMATION

After many days of studying and absorbing knowledge from the lecturer, group 10 understood the application and convenience of the sort technique in programming.

In this project, we were asked to implement all algorithm (for ascending order only), besides we chose **Set 2** (*11 algorithms*):

- Selection Sort.
- Insertion Sort.
- Bubble Sort.
- Shaker Sort.
- Shell Sort.
- Heap Sort.
- Merge Sort.
- Quick Sort.
- Counting Sort.
- Radix Sort.
- Flash Sort.
- Binary Insertion Sort (bonus – not in Set 2)

### • Data size

- Data size of input array:  $n$ .
- Array:  $a$  (begins at  $a[0]$  and ends at  $a[n-1]$ ).
- The element at  $i$  position of array  $a$ :  $a[i]$  ( $\forall i \in [0; n-1]$ ).
- Data size of input array:  $10000 \leq n \leq 500000$ .
- Data size of each element:  $0 \leq a[i] \leq n$ .

### • Charts and tables

- Line graph showing the running time of each function.
- The time unit in the graphs is *millisecond* (ms).
- Bar chart showing the comparisons of each function.

### • Compiling

- To compile the program, we use the compile command:  
**g++ main.cpp DataGenerator.cpp -o <file\_name>.exe**

# III. ALGORITHM PRESENTATION

## 1. Selection Sort

### a. Ideas

-The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from the unsorted part and putting it at the sorted list. The algorithm maintains two subarrays in a given array, the subarray which is already sorted and the remaining subarray which is unsorted. In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

### b. Step-by-step descriptions

- To sort an array of size n in ascending order:
  - 1: Set a marker for the unsorted section at the front of the list
  - 2: Repeat steps 3 - 5 until one number remains in the unsorted section
  - 3: Compare all unsorted numbers in order to select the smallest one
  - 4: Swap this number with the first number in the unsorted section
  - 5: Advance the marker to the right one position
- Example:  $a = \{20, 12, 10, 15, 2\}$

Step	Array a	Explain
1	{   20, 12, 10, 15, <b>2</b> }	Initially the sorted part has nothing and the smallest element of the unsorted part is <b>2</b> , we will bring the element <b>2</b> first and increase the length of the sorted part.
2	{ 2   20, 12, <b>10</b> , 15 }	Now that the smallest element of the unsorted part is <b>10</b> , we bring <b>10</b> over the sorted part.
3	{ 2, 10,   20, <b>12</b> , 15 }	Now that the smallest element of the unsorted part is <b>12</b> , we bring <b>12</b> over the sorted part.

4	{ 2 ,10 ,12 ,  20 , <b>15</b> }	Now that the smallest element of the unsorted part is <b>15</b> , we bring <b>15</b> over the sorted part.
5	{ 2 ,10 ,12 ,15 ,  20 }	At this point, the algorithm stops.

### c. Complexity evaluations

- The number of comparisons to be performed is  $O(n^2)$ . For each stage we need a permutation, because so we need  $n - 1$  permutation, or  $O(n)$  permutation.

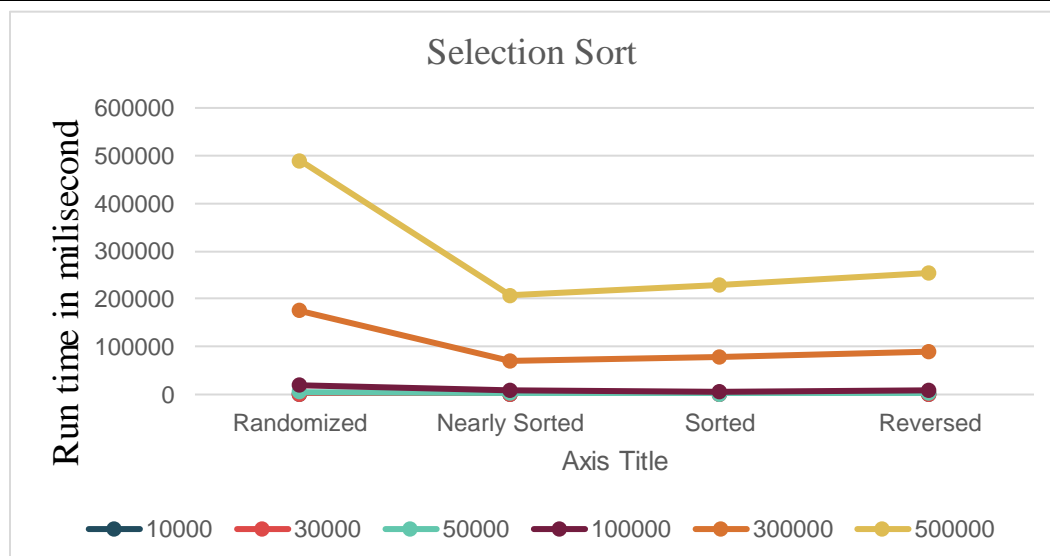
#### ➤ Time Complexities:

- Worst Case: If we want to sort in ascending order and the array is in descending order then, the worst case occurs  $O(n^2)$ .
- Best Case: It occurs when the array is already sorted  $O(n^2)$ .
- Average Case: It occurs when the elements of the array are in jumbled order (neither ascending nor descending)  $O(n^2)$ .

#### ➤ Space Complexities: $O(1)$ .

- Run time in millisecond table:

Data size	10000	30000	50000	100000	300000	500000
Randomized	189	1732	4823	19315	175109	490158
Nearly Sorted	103	897	2467	10024	70386	207837
Sorted	57	510	1415	5627	78660	228204
Reversed	194	1322	3612	9717	90676	253643



## 2. Insertion Sort



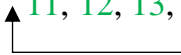
### a. Ideas

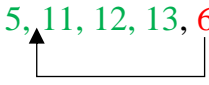
- Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part (in this case, it will be placed at the position where the element  $a[i]$  larger than  $a[i-1]$  and smaller than  $a[i+1]$ ).

### b. Step-by-step descriptions

- To sort an array of size  $n$  in ascending order:
  - ✓ 1: Iterate from  $a[1]$  to  $a[n]$  over the array.
  - ✓ 2: Compare the current element (key) to its predecessor.
  - ✓ 3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element. And all elements which from its their index to position 0 are larger than the key, swap them with the top.
- Ex: There is an unsorted array:

**$a = \{12, 11, 13, 5, 6\}$**

Stage	Array	Explanation
0	 $12, 11, 13, 5, 6$	In the beginning, the array doesn't have no change, start at $a[1] = 11$ .
1	 $11, 12, 13, 5, 6$	Now, we compare and see that $a[0] = 12 > a[1]$ , so we will swap them to place at suitable positions ( $11 < 12$ )
2	 $11, 12, 13, 5, 6$	Then, $a[1] < a[2]$ ( $12 < 13$ ), so we continue the loop.

3		We can see $a[3] < a[2]$ , so we keep compare with $a[1]$ , $a[0]$ . All of them is larger than $a[3]$ , so we place 5 at $a[0]$ .
4	5, 6, 11, 12, 13	At $a[5] = 6$ , we still compare with $a[4], a[3], \dots$ But we can see that, $5 < 6 < 11$ , so we will place 6 at position between them.
5	5, 6, 11, 12, 13	The algorithm finishes here. We will have a sorted array.

### c. Complexity evaluations

➤ **Time complexity:**

- Average complexity:  $O(n^2) \rightarrow$  Randomized data.
- Best Case:  $O(n) \rightarrow$  Sorted Data (+ Nearly Sorted Data).
- Worst Case:  $O(n^2) \rightarrow$  Reversed Data.

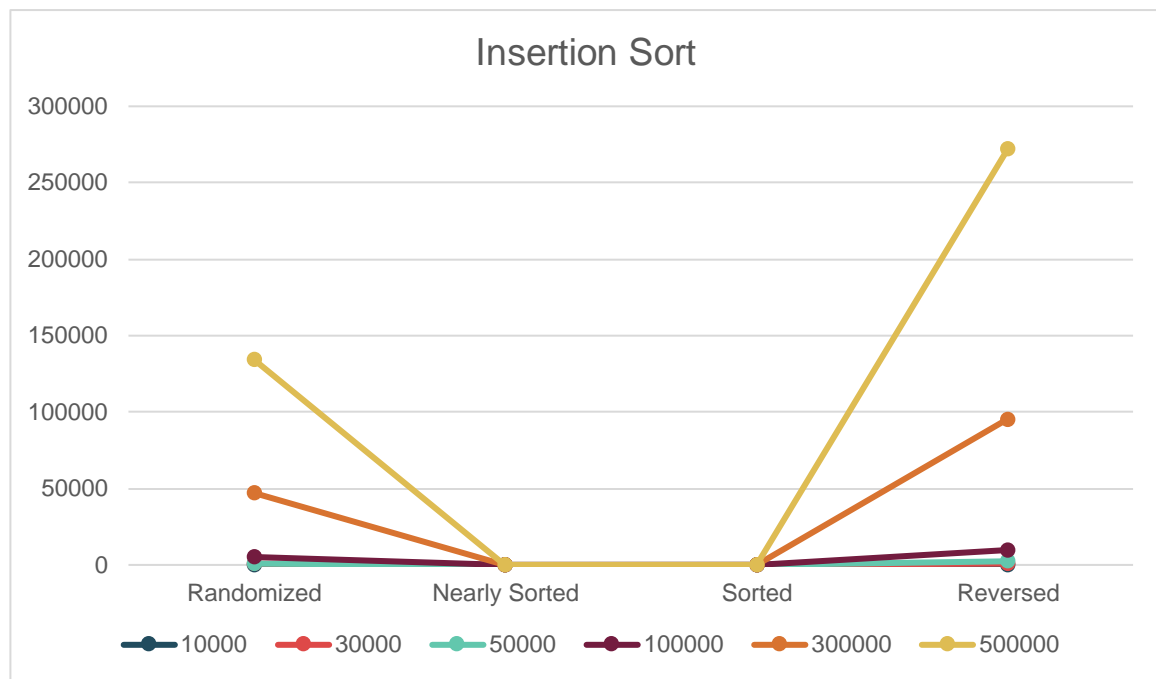
➤ **Space Complexity:**  $O(1)$ . Running time in milisecond table:

Data size	10000	30000	50000	100000	300000	500000
Randomized	48	427	1221	4869	46944	134285
Nearly Sorted	0	1	1	1	1	1
Sorted	0	~ 0	~ 1	1	1	1
Reversed	95	866	2379	9728	95357	271759

### d. Variants / Improvements

- Shell Sort.
- Binary Insertion Sort.





### 3. Bubble Sort

#### a. Ideas

- Compare each pair of adjacent elements from the beginning of an array and, if they are in reversed order, swap them.
- If at least one swap has been done, repeat step 1.

#### b. Step-by-step descriptions

- Example:  $a = \{2, 3, 4, 5, 1\}$

Step	Array a	Explain
1	{2, 3, 4, 5, 1}	Unsorted
2	{2, 3, 4, 5, 1}	We begin with the loop from left to right, we compare two adjacent number : because $2 < 3$ , ok
3	{2, 3, 4, 5, 1}	$3 < 4$ , ok

4	{2, 3, 4, 5, 1}	$4 < 5$ also ok
5	{2, 3, 4, 5, 1}	In here we have $5 > 1$ so we swap 5 and 1 ; To moving the large number to the right side of array
6	{2, 3, 4, 1, 5}	$4 > 1$ we swap 4 and 1
7	{2, 3, 1, 4, 5}	$3 > 1$ we swap 3 and 1
8	{2, 1, 3, 4, 5}	$2 > 1$ we swap 2 and 1, We continue until we don't have any swap operation.
9	{1, 2, 3, 4, 5}	We have a sorted array

### c. Complexity evaluations

- In Bubble Sort,  $n-1$  comparisons will be done in the 1st pass,  $n-2$  in 2nd pass,  $n-3$  in 3rd pass and so on.

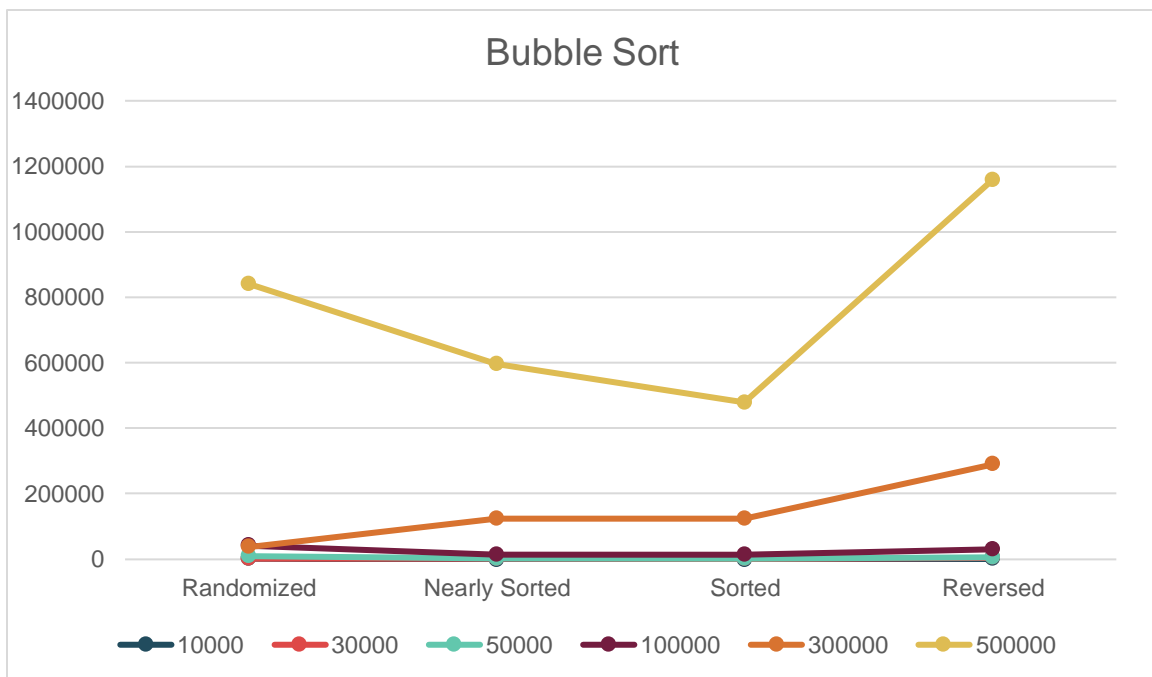
#### ➤ Time Complexity

- Worst and Average Case Time Complexity:  $O(n^2)$  Worst case occurs when the array is reverse sorted.
- Best Case Time Complexity:  $O(n)$ . Best case occurs when the array is already sorted.

#### ➤ Space Complexity: $O(1)$ .

- Running time in milisecond table:

Data size	10000	30000	50000	100000	300000	500000
Randomized	387	3348	9797	41499	37601	841542
Nearly Sorted	146	1139	3000	14261	125771	596976
Sorted	143	1134	3132	14259	124189	479370
Reversed	424	4311	7945	32280	289596	1158300



#### d. Variants/improvements

- One way to improve this algorithm is to phase it out, instead of just introducing small elements in front, we can put the largest element in back.
- Another way is shaker sort, this is an improvement of bubble sort.
- Improve this process itself is to stop browsing as soon as the array is sorted, it has a name is Optimized bubble sort.

## 4. Heap Sort

### a. Ideas

-The Heap sort algorithm has the basic idea of separating the array into two parts, one is sorted in ascending order and the other part is to maintain it as a max-heap.

-Max-heap is defined as a heap with the maximum value at the top of the heap. Max-heap is a binary tree with a parent node that is larger than the two children.

### b. Step-by-step descriptions

- To sort an array of size  $n$  in ascending order:
  - ✓ Stage 1: Adjust the original number sequence to heap
  - ✓ Stage 2: Sort the array based on the heap
    - Step 1: Bring the largest element to the position standing at the end of the sequence
      - +  $r = n - 1$
      - +  $\text{swap}(a[0], a[r])$
    - Step 2:
      - Remove the largest element from the heap:  $r = r - 1$ .
      - Edit the rest of the range from  $a[0]$ ,  $a[1]$ , ...,  $a[r]$  to a heap.
    - Step 3: If  $r > 0$  (heap has elements): Repeat step 2. Opposite: stop.
- Example:

**$a = \{20, 12, 2, 15, 10\}$  ,  $n = 5$**

- When setting heap using array, I consider element standing at position  $i$  will have left child at position  $2*i + 1$  and right child at position  $2 * i + 2$

#### Stage 1: Adjust the original number sequence to heap

(Note: the red cabinet is the element being considered, the blue element is the child of the element under consideration)

Position	Array	Explanation
1	{ 20, 12, 2, 15, 10 }	Due to the elements in the segment $[n/2; n-1]$ will have no children so we skip it without

		considering, we will consider from the element at position $n/2 - 1$ . Element 1 has two children, 15 and 10, due to stopping max-heap, we need put 15 up instead of 12
0	{ <b>20</b> , <b>15</b> , <b>2</b> , 10, 12}	We consider the element at position 0, this element has 2 subvalues, the value 15 and 2 are both less than 20, so we do nothing. Complete phase 1.

### Stage 2: sort array based on heap

(Note : character "|" to split array into 2 parts heap and sorted part)

Repeats	Array	Explanation
1	{ <b>20</b> , <b>15</b> , <b>2</b> , 10, 12,  }	Firstly, we see element 20 at the end of the array, then bring element 12 up instead of element 20
1	{ <b>12</b> , <b>15</b> , <b>2</b> , 10,   <b>20</b> }	We swap positions of values 12 and 15 to ensure max-heap.
1	{15, <b>12</b> , <b>2</b> , <b>10</b> ,   <b>20</b> }	Element 12 is at place, we stop iteration 1. Now element 12 has only 1 child, because element 20 is already outside the current management area.
2	{ <b>15</b> , <b>12</b> , <b>2</b> , 10,   <b>20</b> }	Move 15 backwards and move the element 10 up instead of
2	{ <b>10</b> , <b>12</b> , <b>2</b> ,   <b>15</b> , <b>20</b> }	Swap the position of element 10 and 12 to ensure max-heap
2	{ <b>12</b> , <b>10</b> , <b>2</b> ,   <b>15</b> , <b>20</b> }	Element 10 is already in place and has 0 children. Stop the 2nd iteration

3	{ <b>12</b> , 10, 2,   15, 20 }	Bring 12 back and bring the element 2 up instead of.
3	{ <b>2</b> , <b>10</b>   12, 15, 20 }	Swap position 2 elements 2 and 10 to ensure max-heap.
3	{ 10, <b>2</b>   12, 15, 20 }	Element 2 is already in place and has 0 children. Stop the 3rd iteration.
4	{ <b>10</b> , 2   12, 15, 20 }	Bring 10 back and bring the element 2 up instead of
4	{ <b>2</b>   10, 12, 15, 20 }	Element 0 is already in place and has 0 children. Stop the 4th iteration

### c. Complexity evaluations

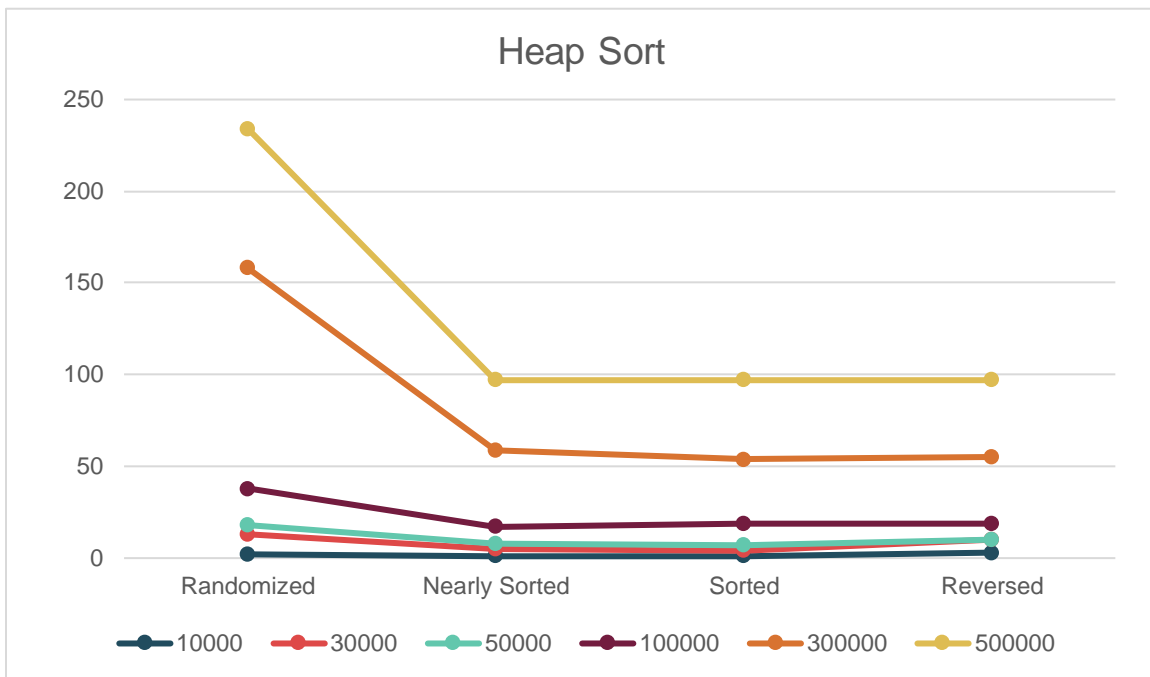
➤ **Time complexity:**

- Average complexity:  $O(n \log n)$
- Best Case:  $O(n)$ .
- Worst Case:  $O(n \log n)$

➤ **Space Complexity:**  $O(1)$

- Running time in milisecond table:

Data size	10000	30000	50000	100000	300000	500000
Randomized	2	13	18	38	158	234
Nearly Sorted	1	5	8	17	59	97
Sorted	1	4	7	19	54	97
Reversed	3	10	10	19	55	97



## 5. Merge Sort

### a. Ideas

- In Merge Sort, the given unsorted array with  $n$  elements, is divided into  $n$  subarrays, each having one element, because a single element is always sorted in itself. Then, it repeatedly merges these subarrays, to produce new sorted subarrays, and in the end, one complete sorted array is produced.
- The concept of Divide and Conquer involves three steps:
  - ✓ **Divide** the problem into multiple small problems.
  - ✓ **Conquer** the subproblems by solving them. The idea is to break down the problem into atomic subproblems, where they are actually solved.
  - ✓ **Combine** the solutions of the subproblems to find the solution of the actual problem.

### b. Step-by-step descriptions

Example:  $a = \{14, 7, 3, 12, 9, 11, 6, 2\}$

Step	Array a	Explain
1	{14,17, 3, 12} and {9,11,6,2}	We will break these subarrays into even smaller subarrays ,until we have multiple subarrays with a single element in them.
2	{14,17} and {3, 12} and {9,11} and {6,2}	Continue divide array.
3	{14} and {17} and {3} and {12} and {9} and {11} and {6} and {2}	Now we have multiple subarrays with single element in them
4	{14,17} and {3, 12} and {9,11} and {2,6}	Then we have to merge all these sorted subarrays, step by step to form one single sorted array.
5	{3,12, 14, 17} and {2,6,9,11}	We continue merge small arr until we got sorted original array
6	{2, 3, 6, 9, 11, 12, 14, 17}	Now we got a sorted array

### c. Complexity evaluations

- Merge Sort is quite fast and has a time complexity of  $O(n \cdot \log_2(n))$ . It is also a stable sort, which means the "equal" elements are ordered in the same order in the sorted list.
- We divide a number into half in every step, it can be represented using a logarithmic function, which is  $\log n$  and the number of steps can be represented by  $\log_2(n) + 1$  (at most)
- Also, we perform a single step operation to find out the middle of any subarray, i.e.  $O(1)$ .
- And to merge the subarrays, made by dividing the original array of  $n$  elements, a running time of  $O(n)$  will be required.



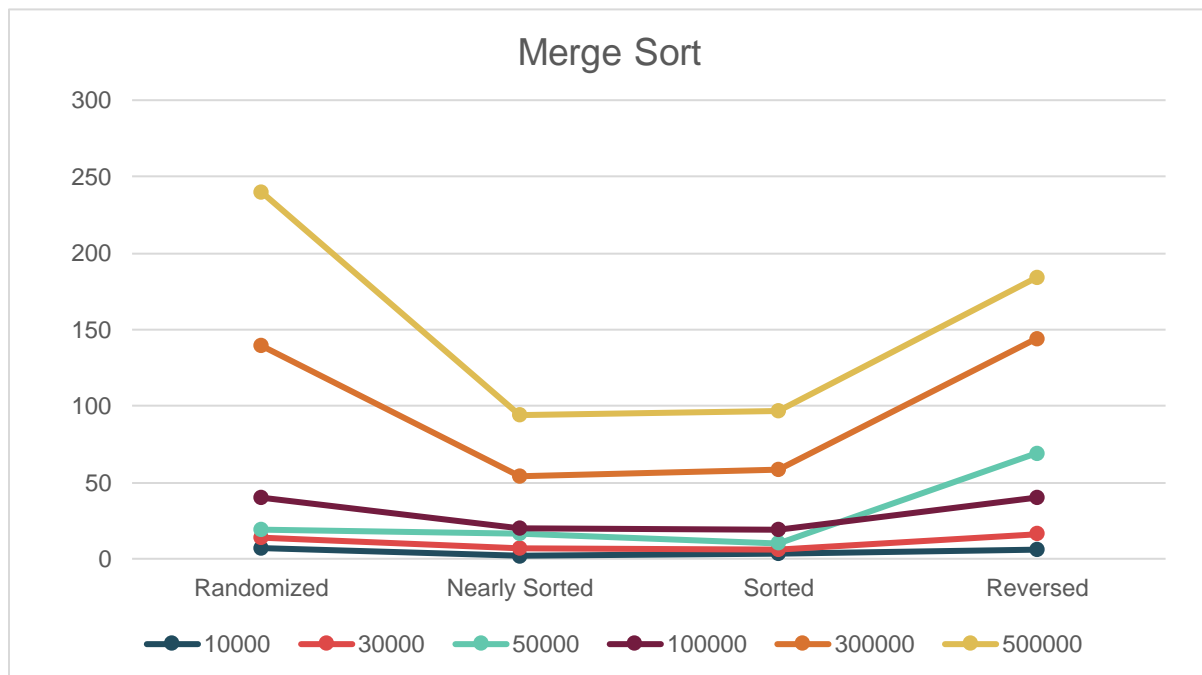
- Hence the total time for **mergeSort** function will become  $n(\log n + 1)$ , which gives us a time complexity of  $O(n \cdot \log n)$ .

➤ **Time Complexity:**

- Worst Case:  $O(n \cdot \log_2(n))$
- Best Case:  $O(n \cdot \log_2(n))$
- Average Case:  $O(n \cdot \log_2(n))$

➤ **Space Complexity:**  $O(n)$

Data size	10000	30000	50000	100000	300000	500000
Randomized	7	14	19	40	139	240
Nearly Sorted	2	7	16	20	54	94
Sorted	3	6	10	19	58	97
Reversed	6	16	69	40	144	184



## 6. Quick Sort

### a. Ideas

- Quicksort is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

### b. Step-by-step descriptions

- To sort an array of size  $n$  in ascending order:
  - 1: If the range has less than two elements, return immediately as there is nothing to do.
  - 2: Otherwise pick a value, called a pivot, that occurs in the range.
  - 3: Partition the range: reorder its elements, while determining a point of division, so that all elements with values less than the pivot come before the division, while all elements with values greater than the pivot come after it.
  - 4: Recursively apply the quicksort to the sub-range up to the point of division and to the sub-range after it, possibly excluding from both ranges the element equal to the pivot at the point of division.
- Example: There is an unsorted array:  $a = \{4, 1, 0, 3, 2\}$

Stage	Array	Explanation
1	{4, 1, 0, 3, <b>2</b> }	Select <b>2</b> as a pivot. Divide into two arrays [1, 0] and [4, 3]
2	{ [ <b>1</b> , 0], 2, [4, 3] }	In [1, 0] we select <b>1</b> as a pivot. Divide into two arrays [1] and [0].
3	{ 0, 1, 2, [ <b>4</b> , 3] }	In [4, 3] we select <b>4</b> as a pivot. Divide into two arrays [4] and [3].
4	{ 0, 1, 2, 3, 4 }	We have a sorted array

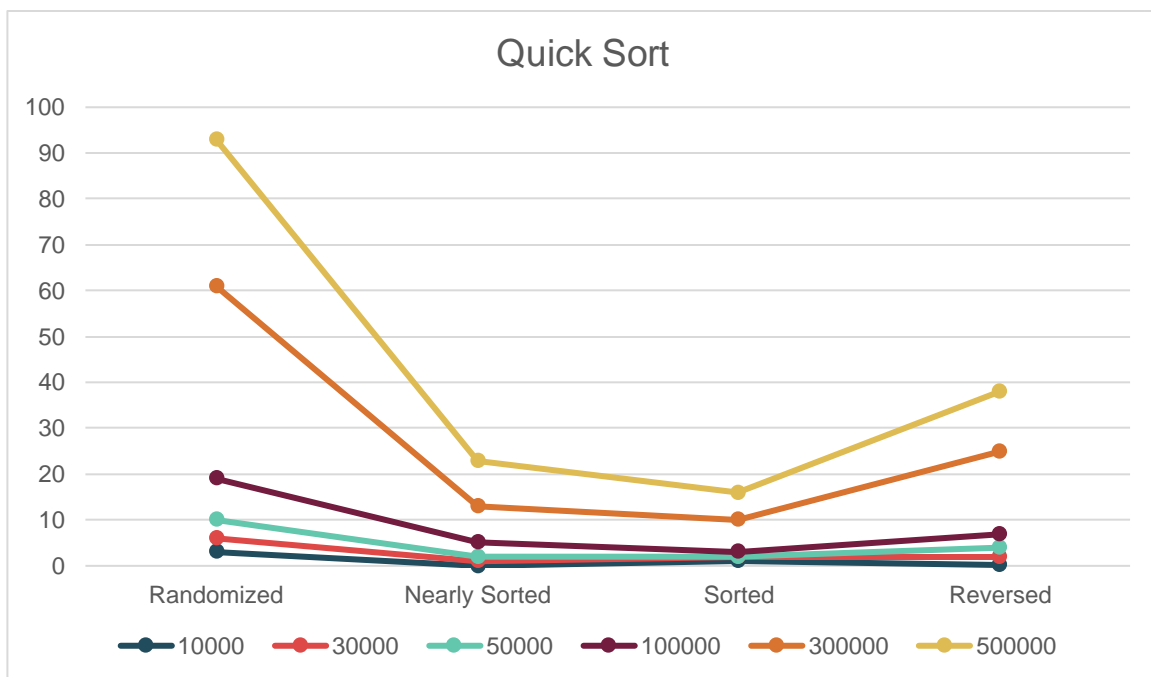
### c. Complexity evaluations

- The complexity of the quick sort algorithm depends on how we choose the pivot element:

- **Best case & Average case:  $O(n \cdot \log_2 n)$**  (In the most balanced case, each time we perform a partition we divide the list into two nearly equal pieces. This means each recursive call processes a list of half the size).
- **Worst case:  $O(n^2)$** . (This may occur if the pivot happens to be the smallest or largest element in the list, it is divided into 2 arrays of length 1 and  $(n - 1)$ ).
- **Space Complexity:  $O(\log_2 n)$** .

- Running time in milisecond table:

Data size	10000	30000	50000	100000	300000	500000
Randomized	3	6	10	19	61	93
Nearly Sorted	0	1	2	5	13	23
Sorted	1	2	2	3	10	16
Reversed	0.1	2	4	7	25	38



## 7. Radix Sort

### a. Ideas

- If in other algorithms, the basis for sorting is always comparing the value of two elements, then radix-sort is based on the postal trial taxonomy. It doesn't care at all about comparing element values, and the sorting and sorting order itself creates the ordering of the elements.

### b. Step-by-step descriptions

- Example:

$$a = \{29, 88, 52, 19, 43\}, n = 5$$

✓ Sort digits by units

0	1	2	3	4	5	6	7	8	9
									19
		52	43					88	29

→ After reviewing and comparing the unit row, the array becomes:

52, 43, 88, 29, 19

✓ Sort element by tens

0	1	2	3	4	5	6	7	8	9
	19	29		43	52			88	

→ After reviewing and comparing the ten rows, the array becomes

⇒ 19, 29, 43, 52, 88

### c. Complexity evaluations

➤ **Time complexity:**

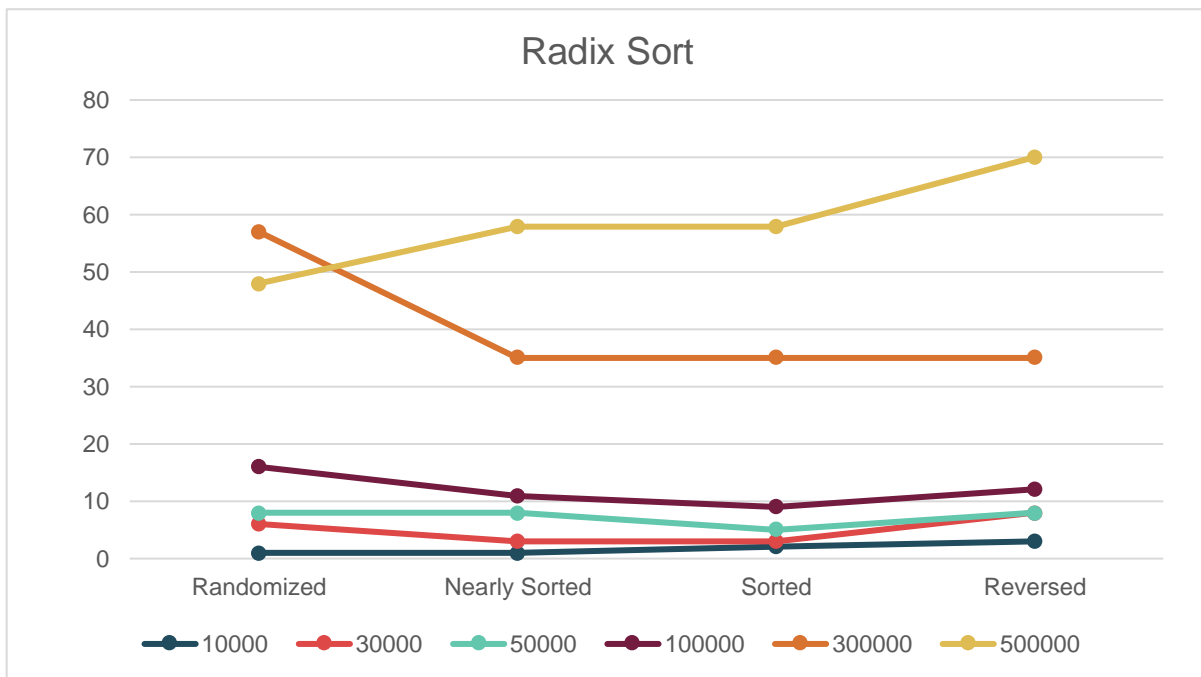
- With a sequence of  $n$  numbers, each number has a maximum of  $m$  digits, the algorithm performs  $m$  times the batching and concatenating operations. In

the batching operation, each element is considered exactly once, also when concatenating. . Thus, the cost of implementing the algorithm is obviously  $O(2mn) = O(n)$

➤ **Space Complexity:**  $O(1)$

- Running time in milisecond table:

Data size	10000	30000	50000	100000	300000	500000
Randomized	1	6	8	16	57	48
Nearly Sorted	1	3	8	11	35	58
Sorted	2	3	5	9	35	58
Reversed	3	8	8	12	35	70



## 8. Shaker Sort

### a. Ideas

- The first stage loops through the array from left to right, just like the Bubble Sort. During the loop, adjacent items are compared and if the value on the

left is greater than the value on the right, then values are swapped. At the end of the first iteration, the largest number will reside at the end of the array.

- The second stage loops through the array in the opposite direction- starting from the item just before the most recently sorted item and moving back to the start of the array. Here also, adjacent items are compared and are swapped if required.

## b. Step-by-step descriptions

- Example:  $a = \{4, 1, 0, 3, 2\}$

We use “ | ” to control the sorted and unsorted element

Step	Array a	Explain
0	{ 4, 1, 0, 3, 2 }	Unsorted
1	{ 4, 1, 0, 3, 2 }	We begin with the loop from left to right, we compare two adjacent number : because $4 < 1$ , so we swap 4 and 1
3	{ 1, 4, 0, 3, 2 }	$4 > 0$ swap 4 and 0
4	{ 1, 0, 4, 3, 2 }	$4 < 3$ swap 4 and 3
5	{ 1, 0, 3, 4, 2 }	$4 < 2$ swap 4 and 2
6	{ 1, 0,3,2  ,4}	Now we finish the first task. Next is the second task, we check the array from last to first element.
7	{ 1, 0, 3, 2  ,4}	$2 < 3$ we swap 2 and 3

8	{ 1, 0, 2, 3  ,4}	$0 < 2$ we do nothing
9	{ 1, 0, 2, 3  ,4}	$0 < 1$ we swap 0 and 1
10	{0, 1,   2,3,4}	We got a sorted array

### c. Complexity evaluations

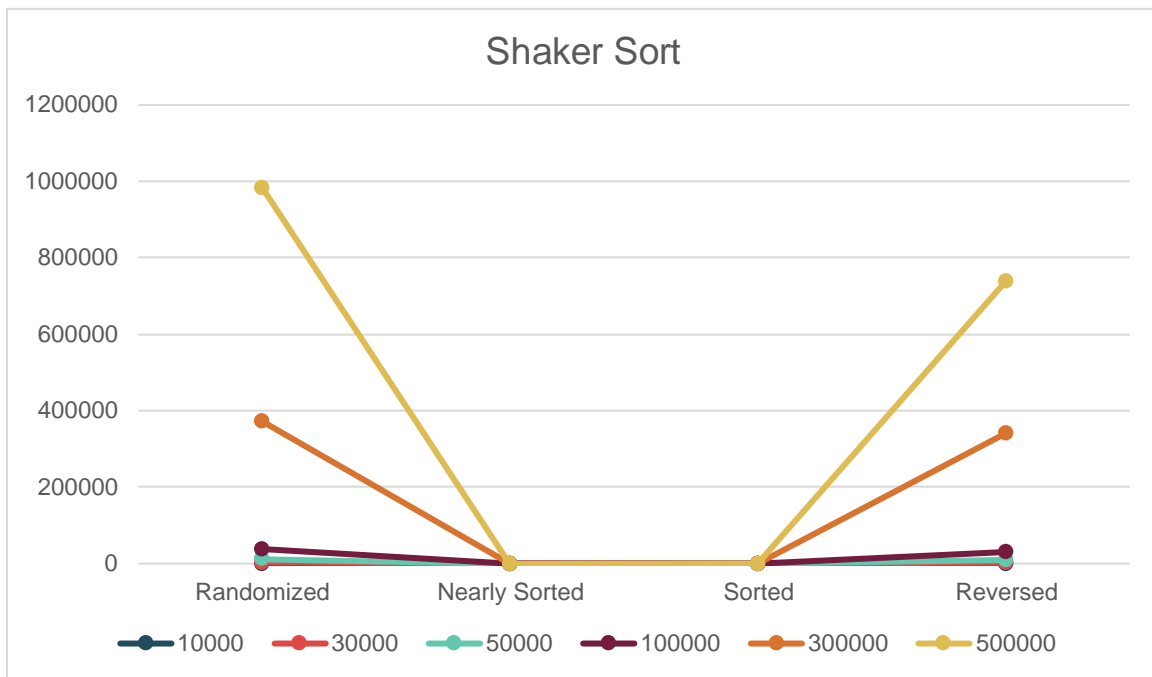
#### ➤ Time Complexity:

- Worst and Average Case Time Complexity:  $O(n^2)$ .
- Best Case Time Complexity:  $O(n)$ . Best case occurs when the array is already sorted.

#### ➤ Space Complexity: $O(1)$

- Running time in milisecond table:

Data size	10000	30000	50000	100000	300000	500000
Randomized	338	3506	11323	38166	372197	983565
Nearly Sorted	1	1	2	1	1	2
Sorted	1	1	1	1	1	1
Reversed	408	3002	8353	32113	340792	740424



## 9. Shell Sort

### a. Ideas

- Shell sort algorithm is an improved method of Insert Sort. The term Insert sort resentment compares and replaces elements that are close together. Here the idea of the sorting method is that the beginning of the original sequence into sequences of elements at a distance “h” from each other.

### b. Step-by-step descriptions

- ✓ Step 1: Initialize h values
- ✓ Step 2: Split the list into smaller sublists corresponding to h
- ✓ Step 3: Sort these sublists using insertion sort (Insert sort)
- ✓ Step 4: Iterate to when the list is sorted
- Example:

$$a = \{20, 12, 10, 15, 2\}, n = 5$$

Step	Distance	Array a	Explain
------	----------	---------	---------



1	$h = n/2 = 2$	{ <b>20</b> , 12, <b>10</b> , 15, 2}	Starting with the element $h = 2$ , we compare 20 and 10, but $20 > 10$ , so 10 should be permuted first
2	$h = 2$	{ <b>10</b> , <b>12</b> , 20, <b>15</b> , 2}	With $h = 2$ we continue to compare 12 and 15 but $12 < 15$ should stay in place. "Distance" has run out of array start creating a new step
3	$h = h/2 = 1$	{ <b>10</b> , <b>12</b> , 20, 15, 2}	With $h = 1$ we compare 10 and 12 but $10 < 12$ should stay in place.
4	$h = 1$	{ <b>10</b> , <b>12</b> , <b>20</b> , 15, 2}	With $h = 1$ we compare 12 and 20 but $12 < 20$ should stay in place.
5	$h = 1$	{ <b>10</b> , 12, <b>20</b> , <b>15</b> , 2}	With $h = 1$ we compare 20 and 15 but $20 > 15$ (not in the right place) we swap positions of 20 and 15.
6	$h = 1$	{ <b>10</b> , 12, 15, <b>20</b> , <b>2</b> }	With $h = 1$ we compare 20 and 2 but $20 > 2$ (not in the right place) we swap positions of 20 and 2. Here we end the array traversal with $h = 1$ and end the algorithm...

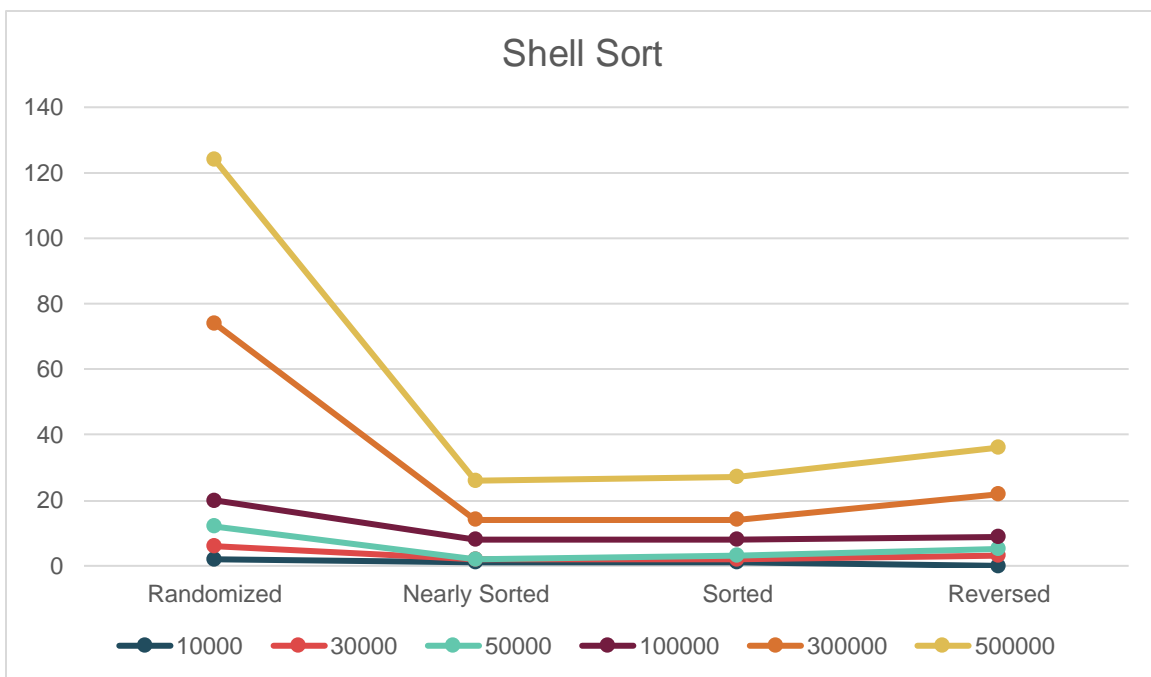
### c. Complexity evaluations

- Currently, the evaluation of Shellsort solutions leads to very complex mathematical problems, some of which have not been proven yet. However, the efficiency of the algorithm depends on the "Distance" selected. In the case of choosing the length sequence according to the formula  $h[i] = (h[i-1] - 1)/2$  and  $h[k] = 1$ ,  $k = \log_2 - 1$  (where  $k$  is the length of the sequence Distance), the algorithm has an equivalent complexity of  $\approx n^{1.2} \ll n^2$ .

- Best case:  $O(n \cdot \log_2(n))$ .
- Average case:  $O(n^{4/3})$ .
- Worst Case:  $O(n^{3/2})$ .
- Space Complexity:  $O(1)$ .

- Running time in milisecond table:

Data size	10000	30000	50000	100000	300000	500000
Randomized	2	6	12	20	74	124
Nearly Sorted	1	2	2	8	14	26
Sorted	1	2	3	8	14	27
Reversed	~0	3	5	9	22	36



## 10. Counting Sort

### a. Ideas

- Counting sort is a sorting technique based on numbers between a specific range. It works by counting the number of elements having distinct key values. Then doing some arithmetic to calculate the position of each object in the output sequence.

### b. Step-by-step descriptions

- To sort an array of size  $n$  in ascending order:

**a**

1	4	1	2	7	5	2
0	1	2	3	4	5	6

- ✓ 1: Find out the maximum and minimum elements (let then be  $max$ ,  $min$ ) from the given array. In the instance,  $max = 7$ ,  $min = 1$ ).
- ✓ 2: Initialize an array (count) of length  $max + 1$  with all elements 0. This array is used for storing the count of the elements in the array.

**count**

0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7

- ✓ 3: Store the count of each element at their respective index in count array.

**count**

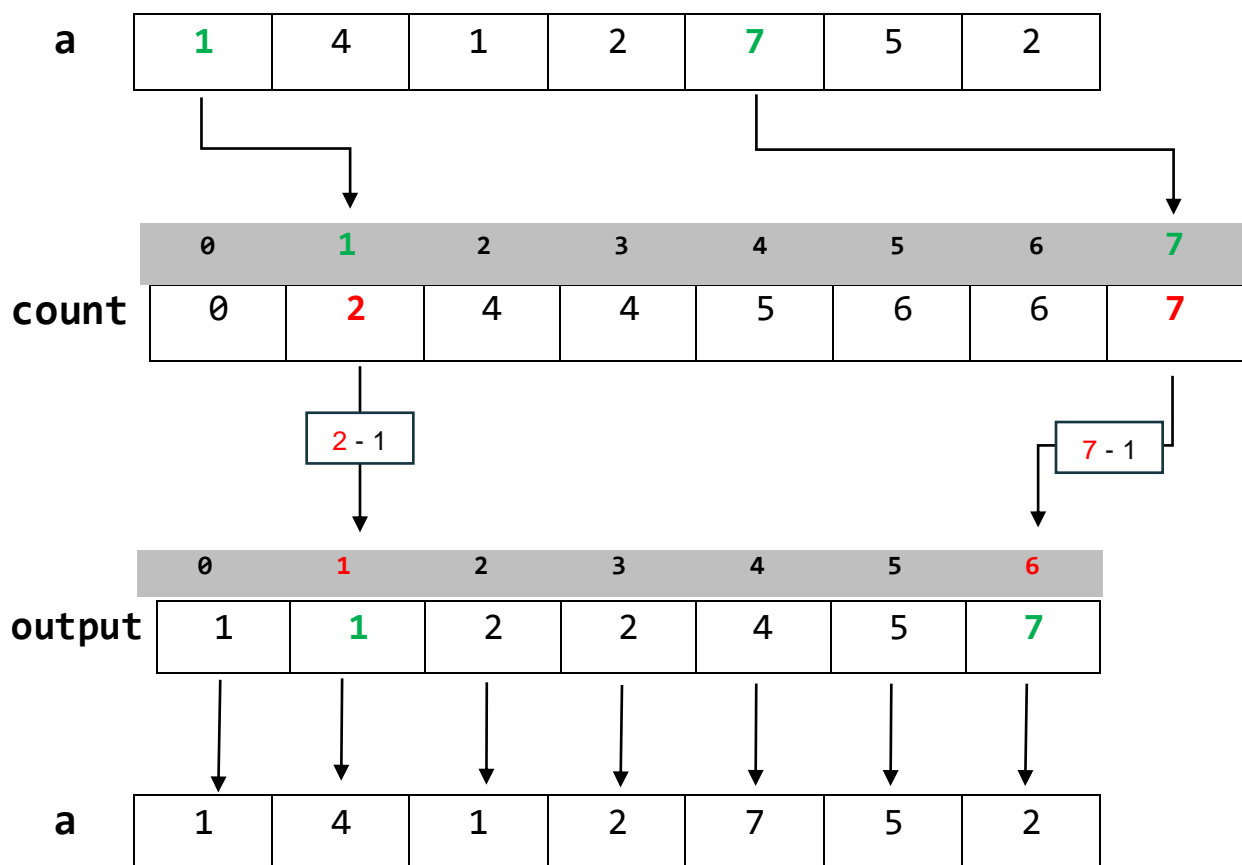
0	2	2	0	1	1	0	1
0	1	2	3	4	5	6	7

- ✓ 4: Store cumulative sum of the elements of the count array. It helps in placing the elements into the correct index of the sorted array.

<b>count</b>	0	2	4	4	5	6	6	7
	0	1	2	3	4	5	6	7

✓ 5: Find the index of each element of the original array in the count array. This gives the cumulative count. Place the element at the index calculated as shown in figure below.

✓ 6: After placing each element at its correct position, decrease its count by one.



### c. Complexity evaluations

- There are mainly four main loops.

Loop	Complexity
------	------------

1: Generate Count array	n
2: Store cumulative sum of the elements of the count array	Count.size = max – min + 1 = k
3: Find the index of each element of the original array in the count array	n
4: Copy the output array to the origin array	n

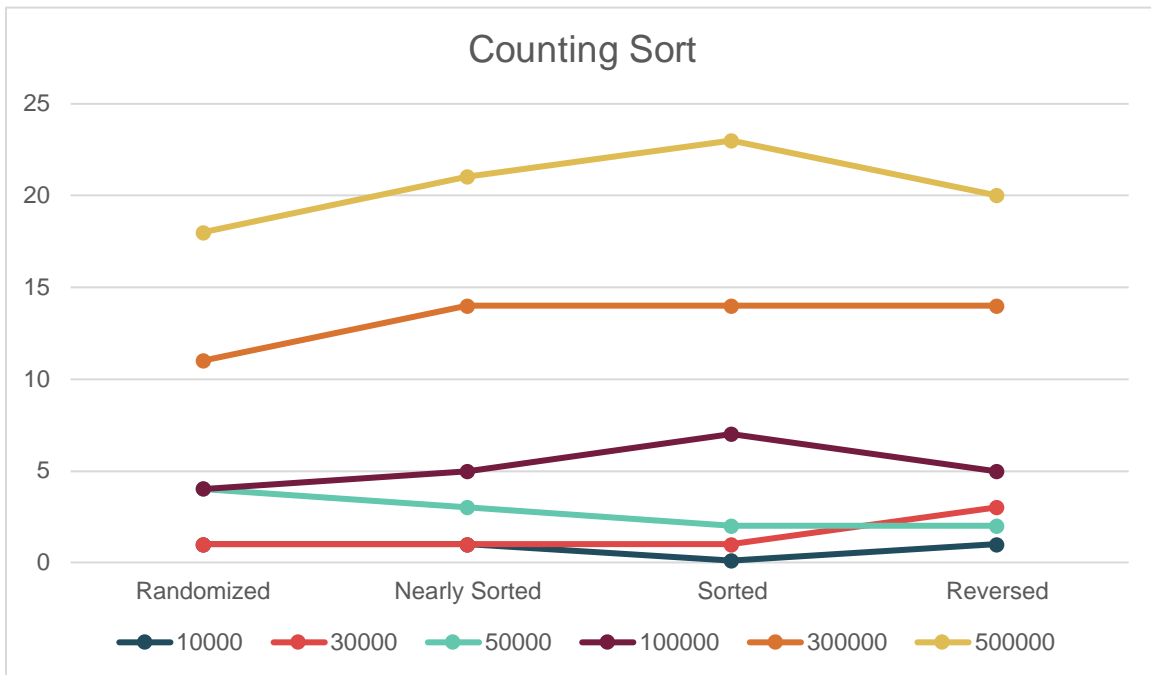
➤ **Time complexity:** Because there isn't any compare, so:

- Average complexity:  $O(n + k)$
- Best Case:  $O(n + k)$
- Worst Case:  $O(n + k)$
- ( k is the range of input )

➤ **Space Complexity:**  $O(n + k)$ .

- Running time in milisecond table:

Data size	10000	30000	50000	100000	300000	500000
Randomized	1	1	4	4	11	18
Nearly Sorted	1	1	3	5	14	21
Sorted	0.1	1	2	7	14	23
Reversed	1	3	2	5	14	20



## 11. Flash Sort

### a. Ideas

- The idea of the algorithm is to divide the elements into  $m$  different segments, the element  $a[i]$  will be in the segment:  $1 + \left\lfloor (m - 1) * \frac{a[i] - \min}{\max - \min} \right\rfloor$   
( $m = 0.43 * n$ )
- After fission is complete, the algorithm will arrange the elements in the same segment by the insertion sort algorithm.

### b. Step-by-step descriptions

$a = \{4, 1, 0, 3, 2\}$  ;  $n = 5 \rightarrow m = 3$

Element	Segment
4	3
1	1
0	1
2	2
3	2

$\rightarrow a = \{1, 0 \mid 3, 2 \mid 4\}$

- In the next step, we will use insertion sort to sort each segment:

$\rightarrow a = \{1, 0 \mid 3, 2 \mid 4\}$

- At this point, the algorithm stops.

### c. Complexity evaluations

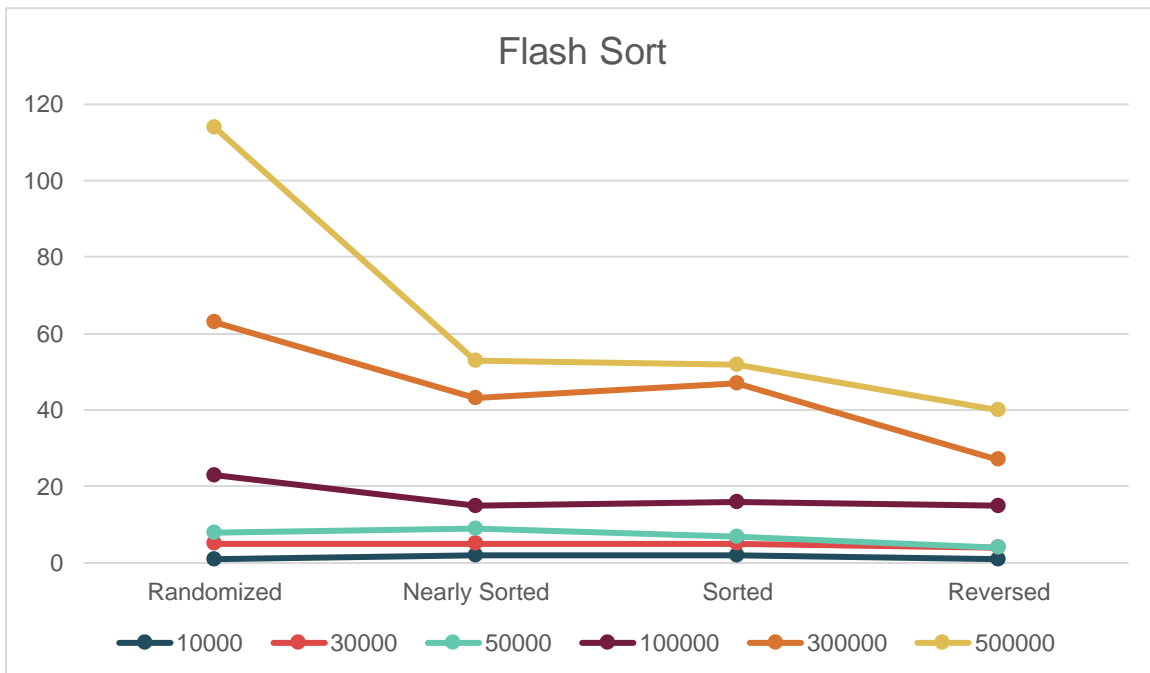
#### ➤ Time complexity:

- Average complexity:  $O(n + m)$
- Best Case:  $O(n + m)$
- Worst Case:  $O(n^2)$

#### ➤ Space Complexity: $O(m)$ .

- Running time in milisecond table:

Data size	10000	30000	50000	100000	300000	500000
Randomized	1	5	8	23	63	114
Nearly Sorted	2	5	9	15	43	53
Sorted	2	5	7	16	47	52
Reversed	1	4	4	15	27	40

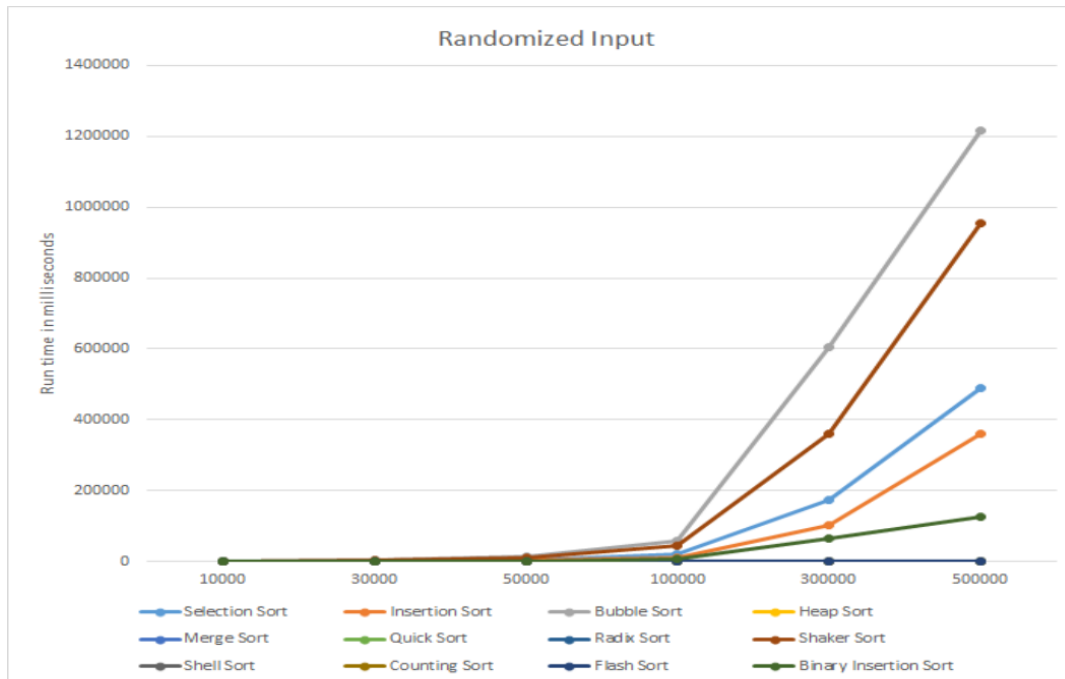


# IV. EXPERIMENTAL RESULT & COMMENTS

## 1. Randomized data

RANDOMIZED												
DATA SIZE	10000		30000		50000		100000		300000		500000	
RESULTING STATICS	RUNTIME	COMPARISONS	RUNTIME	COMPARISONS	RUNTIME	COMPARISONS	RUNTIME	COMPARISONS	RUNTIME	COMPARISONS	RUNTIME	COMPARISONS
<i>Selection Sort</i>	189	100,030,001	1,732	900,090,001	4,823	2,500,150,001	19,315	10,000,300,001	175,109	90,000,900,001	490,158	250,001,500,001
<i>Insertion Sort</i>	94	50,297,724	875	452,340,784	2253	1,244,847,522	10560	4,997,559,494	101506	44,946,744,449	361351	124,956,954,292
<i>Bubble Sort</i>	403	100,009,999	4606	900,029,999	15313	2,500,049,999	59206	10,000,099,999	604698	90,000,299,999	1216980	250,000,499,999
<i>Heap Sort</i>	2	622,481	13	2,105,181	18	3,695,975	38	7,894,611	158	26,037,819	234	45,219,773
<i>Merge Sort</i>	7	547422	14	1,841,468	19	3,202,758	40	6,804,446	139	22,341,976	240	38,826,530
<i>Quick Sort</i>	3	453272	6	1,538,216	10	2,741,701	19	5,535,706	61	18,273,127	93	31,950,819
<i>Radix Sort</i>	1	120,057	6	450,071	8	750,071	16	1,500,071	57	4,500,071	48	7,500,071
<i>Shaker Sort</i>	338	66,922,896	3.834	597,910,008	9.239	1,667,325,588	44.168	6,648,285,354	358.660	59,832,194,476	953.675	249,999,696,288
<i>Shell Sort</i>	2	693,521	6	2,399,478	12	4,399,882	20	9,875,471	74	39,171,287	124	65,834,264
<i>Counting Sort</i>	1	40,001	1	120,002	2	182,771	4	332,771	11	932,771	18	1,532,771
<i>Flash Sort</i>	1	87,094	3	268,644	6	465,283	8	830550	27	2,690,843	97	4,410,369
<i>Binary Insertion Sort</i>	71	25,450,847	487	226,590,675	1,489	626,136,993	7,765	2,506,124,357	64,989	22,531,217,090	126,673	62,519,061,043

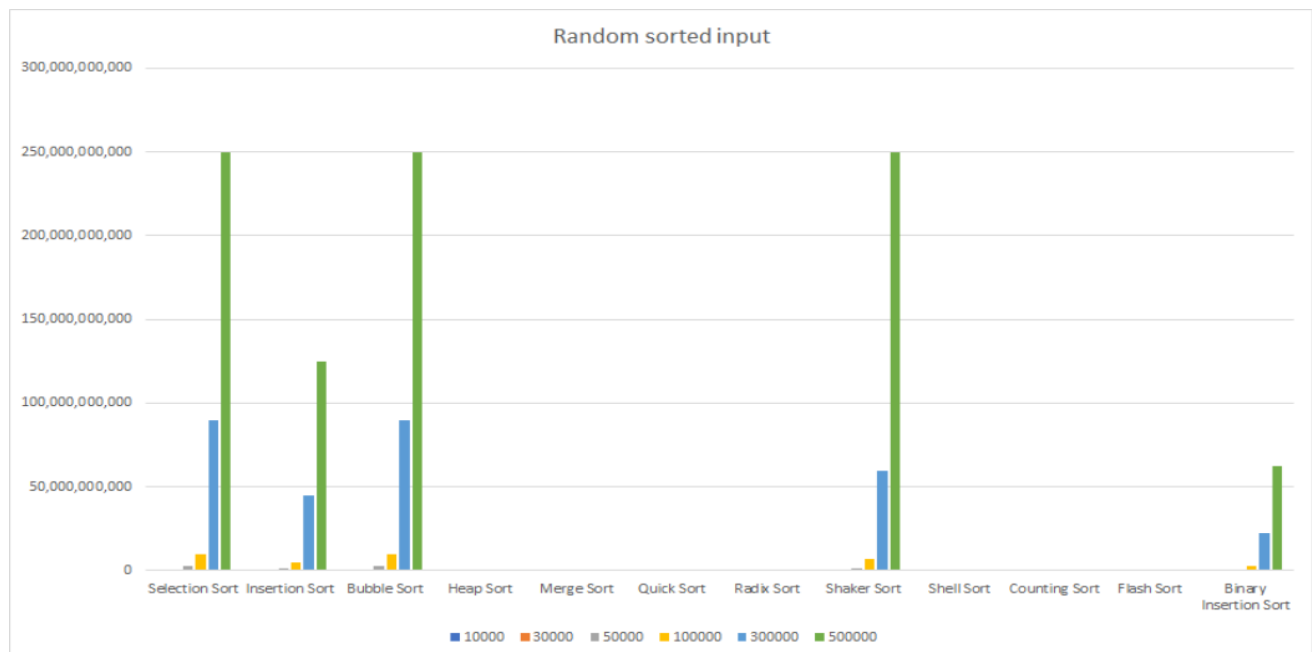
### a. Running time





- We see that in sorting algorithms with complexity  $O(n^2)$ , **Bubble Sort** is the worst algorithm, even optimizing it by **Shaker Sort** does not improve the speed.
- Next, **Selection Sort** algorithm is almost twice as slow as **Insertion Sort**. The **Binary Insertion Sort** algorithm is quite good, which can be said to be the fastest running algorithm in  $O(n^2)$  algorithms except **Shell Sort**.
- **Shell Sort** algorithm has faster speed than **Merge Sort**. The **Radix Sort** algorithm in small data sets runs slower than most other algorithms, but when  $n$  is large, it only runs slower than **Flash Sort** and **Counting Sort**.
- Of the 3 algorithms **Merge Sort**, **Heap Sort**, **Quick Sort**, **Quick Sort** has the faster speed (not too much) than **Heap Sort**, **Merge Sort**.
- The fastest algorithm is **Counting Sort**, then **Flash Sort**.

## b. Comparisons

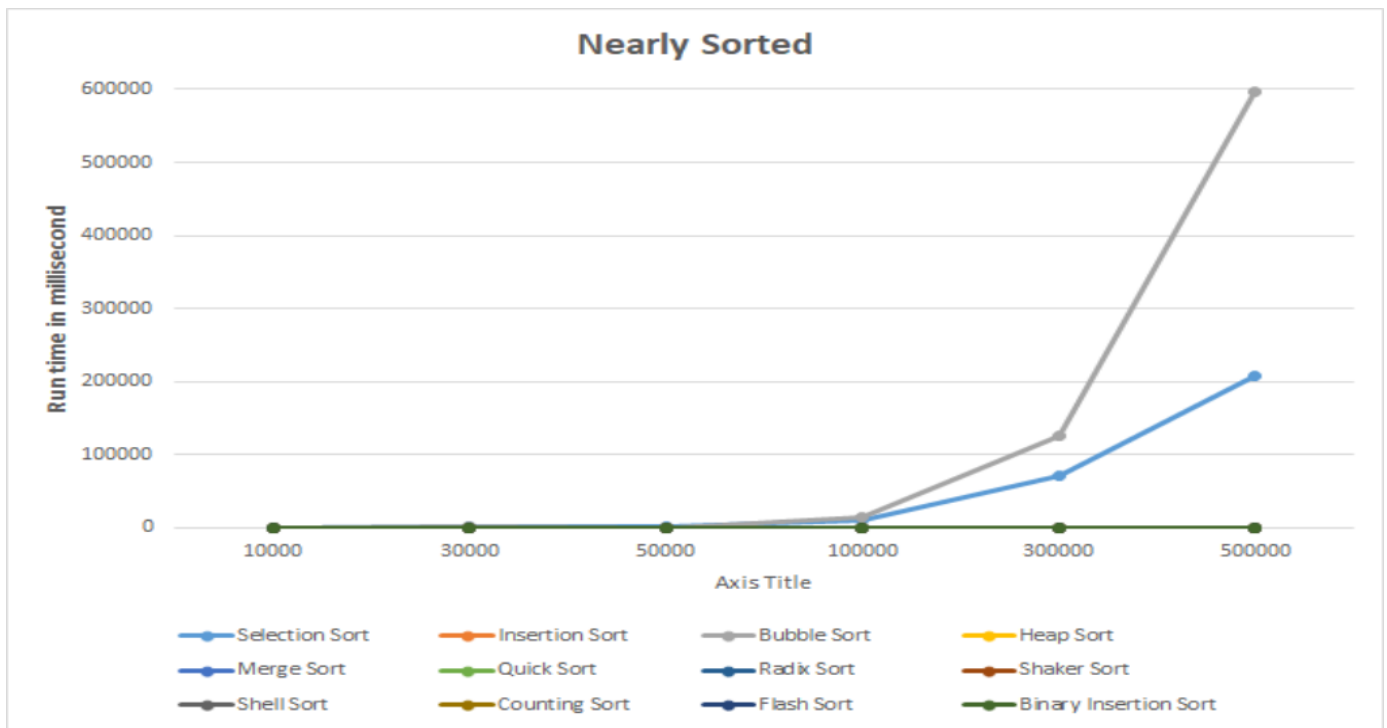


- We see that in sorting algorithms with complexity  $O(n^2)$ , *Bubble Sort*, *Selection Sort* and *Shaker Sort* have a huge number of comparisons ( $\sim n^2$ ).
- Then, *Insertion Sort* has half the number of comparisons compared to *Selection Sort*, and Binary Insertion Sort is also half that of *Insertion Sort*.
- *Heap Sort*, *Merge Sort*, *Quick Sort*, *Radix Sort*, *Shell Sort*, *Counting Sort*, *Flash Sort* have a very small number of comparisons when compared to Bubble Sort, Selection Sort and Shaker Sort.

## 2. Nearly sorted data

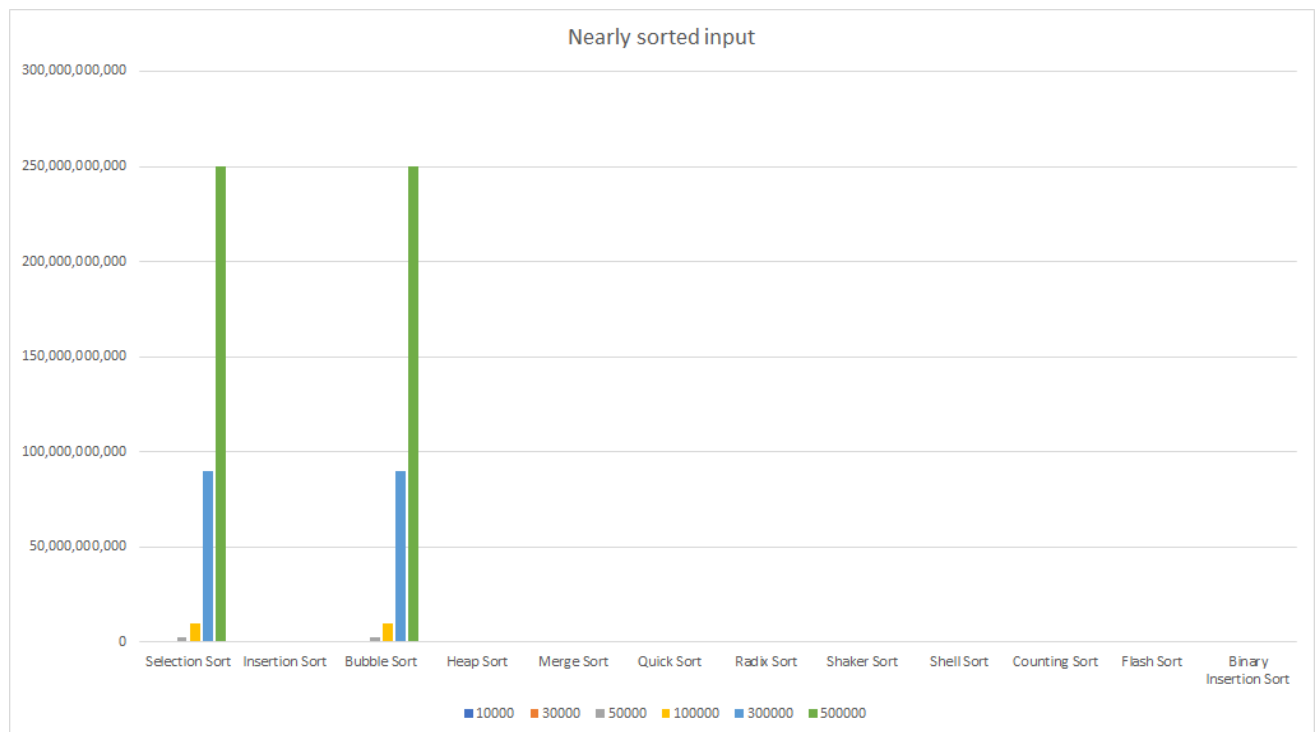
NEARLY SORTED												
DATA SIZE	10000		30000		50000		100000		300000		500000	
RESULTING STATICS	RUNTIME	COMPARISONS	RUNTIME	COMPARISONS	RUNTIME	COMPARISONS	RUNTIME	COMPARISONS	RUNTIME	COMPARISONS	RUNTIME	COMPARISONS
<i>Selection Sort</i>	103	100,030,001	897	900,090,001	2,467	2,500,150,001	10,024	10,000,300,001	70,386	90,000,900,001	207,837	250,001,500,001
<i>Insertion Sort</i>	0	120,339	1	382,583	2	647027	1	766859	3	938467	2	1,443,379
<i>Bubble Sort</i>	146	100,009,999	1,139	900,029,999	3,313	2,500,049,999	14,261	10,000,099,999	125,771	90,000,299,999	596,976	250,000,499,999
<i>Heap Sort</i>	1	655,328	5	2,191,647	8	3,850,350	17	8,215,079	59	26,963,229	97	45,215,287
<i>Merge Sort</i>	2	468,008	7	1,548,840	16	2,660,680	20	5,621,368	54	18,494,520	94	31,920,696
<i>Quick Sort</i>	0	337,225	1	1,104,457	2	1,918,921	5	4,037,849	13	13,051,417	23	22,451,417
<i>Radix Sort</i>	1	120,057	3	450,071	8	750,071	11	1,500,071	35	5,400,085	58	9,000,085
<i>Shaker Sort</i>	1	185866	1	662334	2	610,802	1	663,466	1	1,116,298	2	1,459,082
<i>Shell Sort</i>	1	340,042	3	1,200,044	4	2,150,050	5	4,600,053	15	14,700,053	26	26,000,062
<i>Counting Sort</i>	0	40,003	1	120,003	1	200,003	3	400,003	10	1,200,003	14	2,000,003
<i>Flash Sort</i>	1	112,874	2	338,672	3	564,478	5	1,128,977	15	3,386,978	26	5,644,976
<i>Binary Insertion Sort</i>	1	336,436	3	1,114,280	3	1,733,088	6	3,619,776	15	11,365,140	27	19,647,050

### a. Running time



- Leading the slowest-running algorithms are ***Bubble Sort*** and ***Selection Sort***. ***Selection Sort*** has largest time sorting almost all the sort algorithm.
- ***Merge Sort*** still runs at the same time as ***Heap Sort*** and slower a bit than ***Quick Sort***. We can see in any case, if Quick Sort is better installed, it always runs faster than Heap Sort.
- With this data we see that the ***Radix Sort*** algorithm has a slightly slower speed than the ***Flash Sort***, ***Shell Sort***, ***Counting Sort*** and Binary Insertion Sort.
- The fastest algorithm is ***Shaker Sort***.

## b. Comparisons

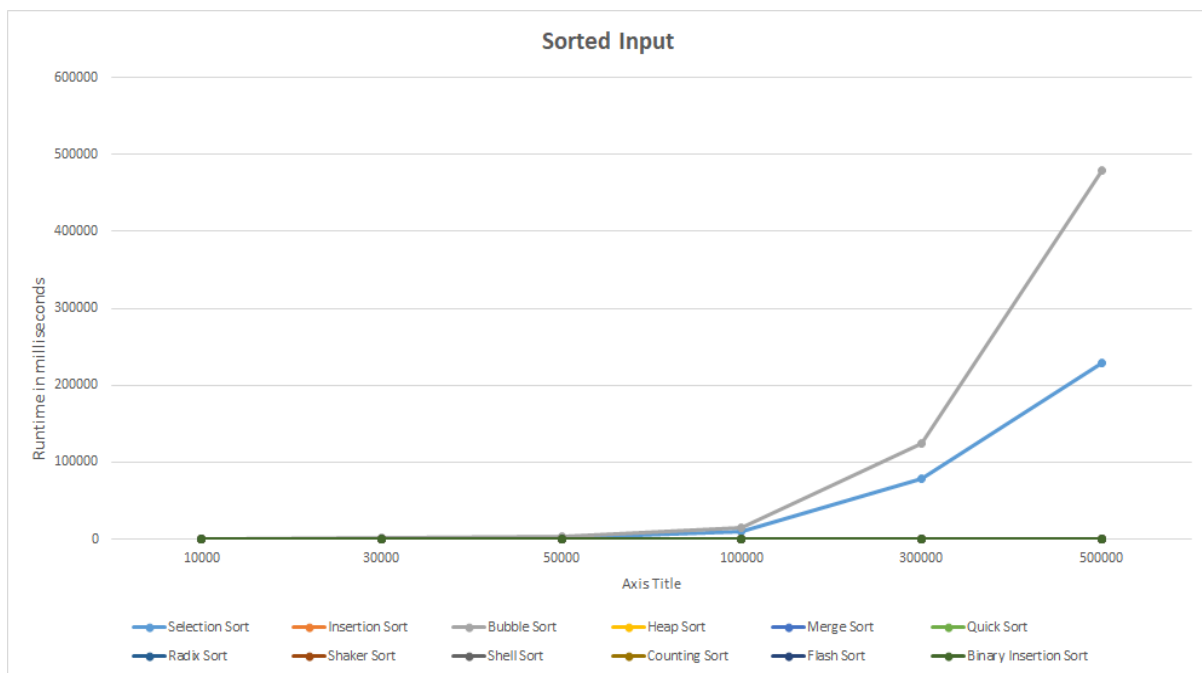


- ***Bubble Sort*** and ***Selection Sort*** have a huge number of comparisons ( $\sim n^2$ )
- ***Insertion Sort***, ***Heap Sort***, ***Merge Sort***, ***Quick Sort***, ***Radix Sort***, ***Shell Sort***, ***Counting Sort***, ***Flash Sort*** and Binary Insertion Sort have a very small number of comparisons when compared to Bubble Sort and Selection Sort.

### 3. Sorted data

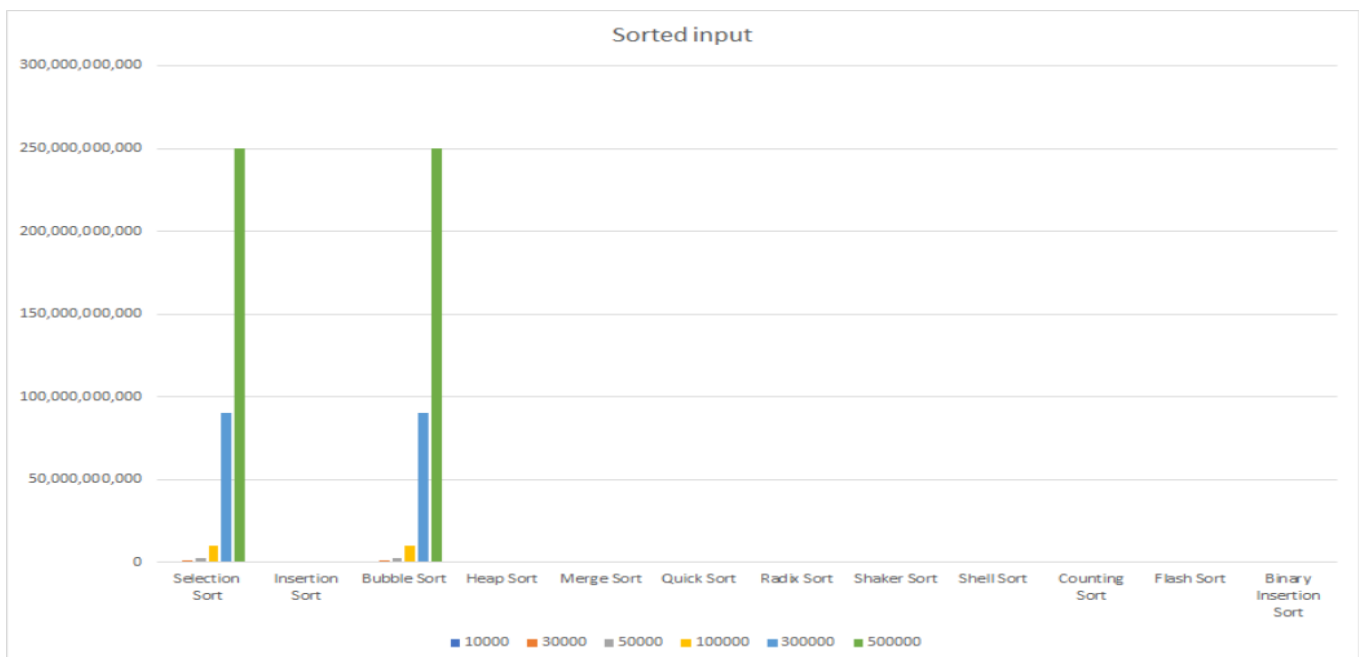
SORTED												
DATA SIZE	10000		30000		50000		100000		300000		500000	
RESULTING STATICS	RUNTIME	COMPARISONS	RUNTIME	COMPARISONS	RUNTIME	COMPARISONS	RUNTIME	COMPARISONS	RUNTIME	COMPARISONS	RUNTIME	COMPARISONS
<i>Selection Sort</i>	57	100,030,001	510	900,090,001	1,415	2,500,150,001	5,627	10,000,300,001	78660	90,000,900,001	228,204	250,001,500,001
<i>Insertion Sort</i>	0	19,999	0	59,999	0	99,999	0	199,999	1	599,999	1	999,999
<i>Bubble Sort</i>	143	100,009,999	1,134	900,029,999	3,132	2,500,049,999	14,259	10,000,099,999	124,189	90,000,299,999	479370	250,000,499,999
<i>Heap Sort</i>	1	654,957	4	2,191,635	7	3,850,238	19	8,215,111	54	26,963,109	97	46,654,885
<i>Merge Sort</i>	3	497,114	6	1,638,098	10	2,740,736	19	5,697,988	58	18,607,440	97	32,029,790
<i>Quick Sort</i>	1	337,261	2	1,104,497	2	1,918,961	3	4,037,885	10	13,051,465	16	22,451,457
<i>Radix Sort</i>	2	120,057	3	450,071	5	750,071	9	1,500,071	35	5,400,085	58	9,000,085
<i>Shaker Sort</i>	1	20000	1	60000	1	100000	1	200000	1	600.000	1	1,000,000
<i>Shell Sort</i>	1	388,437	2	1,322,455	3	2,325,676	4	4,807,664	13	14,883,814	23	26,173,470
<i>Counting Sort</i>	0	40,003	2	120,003	2	200,003	3	400,003	14	1,200,003	24	2,000,003
<i>Flash Sort</i>	1	241,754	2	665,438	2	995,534	5	1,483,278	16	3,326,294	25	5,172,958
<i>Binary Insertion Sort</i>	1	312,338	1	1,101,012	2	1,885,666	3	3,617,362	8	11,479,844	14	19,726,000

#### a. Running time



- Leading the slowest-running algorithms are ***Bubble Sort*** and ***Selection Sort***. Unlike in previous cases, Bubble Sort's running speed is not much greater than that of ***Selection Sort***. ***Selection Sort*** has largest time sorting almost all the sort algorithm  $O(n^2)$ .
- ***Merge Sort*** still runs at the same time as ***Heap Sort*** and slow than ***Quick Sort***.
- We see that the ***Radix Sort*** algorithm has a slightly slower speed than the ***Flash Sort***, ***Shell Sort***, ***Counting Sort***.
- Counting Sort, Insertion Sort, Shaker Sort, Flash Sort, Counting Sort algorithms all have not too different speeds.

## b. Comparisons



- Like above, ***Bubble Sort*** and ***Selection Sort*** still have a huge number of comparisons.

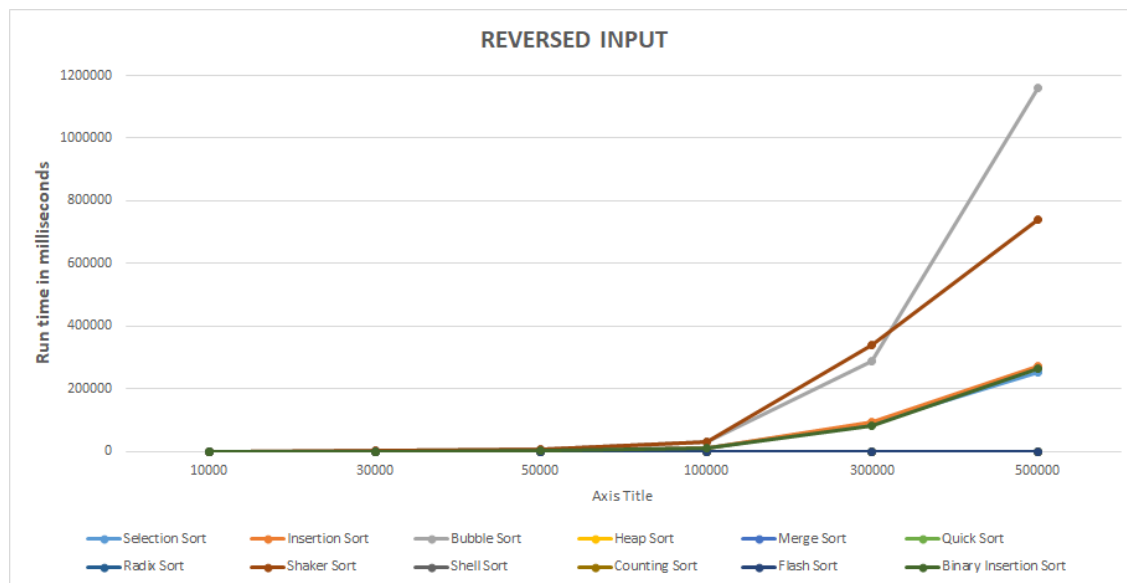
- *Insertion Sort, Heap Sort, Merge Sort, Quick Sort, Radix Sort, Shell Sort, Counting Sort, Flash Sort* and Binary Insertion Sort have a very small number of comparisons when compared to Bubble Sort and Selection Sort.

## 4. Reverse sorted data

REVERSED SORTED												
DATA SIZE	10000		30000		50000		100000		300000		500000	
RESULTING STATICS	RUNTIME	COMPARISONS	RUNTIME	COMPARISONS	RUNTIME	COMPARISONS	RUNTIME	COMPARISONS	RUNTIME	COMPARISONS	RUNTIME	COMPARISONS
<i>Selection Sort</i>	194	100,030,001	1,322	900,090,001	3,612	2,500,150,001	9,717	10,000,300,001	90,676	90,000,900,001	253,643	250,001,500,001
<i>Insertion Sort</i>	211	100,019,998	1,438	900,059,998	4,103	2,500,099,998	9,728	10,000,199,998	95,357	90,000,599,998	271,759	250,000,999,998
<i>Bubble Sort</i>	424	100,010,000	4,311	900,029,999	7,945	2,500,049,999	32280	10,000,099,999	289,596	90,000,299,999	1,158,300	250,000,499,999
<i>Heap Sort</i>	3	591770	10	2,018,323	10	3,537,723	19	7,568,942	55	25,119,378	97	43,733,347
<i>Merge Sort</i>	6	446,444	16	1,483,468	69	2,583,948	40	5,467,900	144	17,808,316	184	30,836,412
<i>Quick Sort</i>	0	347,221	2	1,134,453	4	1,968,917	7	4,137,845	25	13,351,413	38	22,951,413
<i>Radix Sort</i>	3	120,057	8	450,071	8	750,071	12	1,500,071	35	5,400,085	70	9,000,085
<i>Shaker Sort</i>	408	100,010,000	3,002	900,030,000	8,353	2,500,050,000	32,113	10,000,100,000	340,792	60,031,004,486	740,424	90,000,300,000
<i>Shell Sort</i>	1	465,388	3	1,605,417	5	2,893,451	9	6,186,870	22	20,067,432	36	34,602,560
<i>Counting Sort</i>	1	40,003	3	120,003	2	200,003	5	400,003	14	1,200,003	20	2,000,003
<i>Flash Sort</i>	1	100,014,511	4	900,043,511	4	2,500,072,511	15	10,000,145,011	27	90,000,435,011	40	250,000,725,011
<i>Binary Insertion Sort</i>	134	50,252,260	1,012	450,849,494	2,849	1,251,493,960	10,665	5,003,187,890	82,863	45,010,501,462	265,443	125,018,201,462

### a. Running time

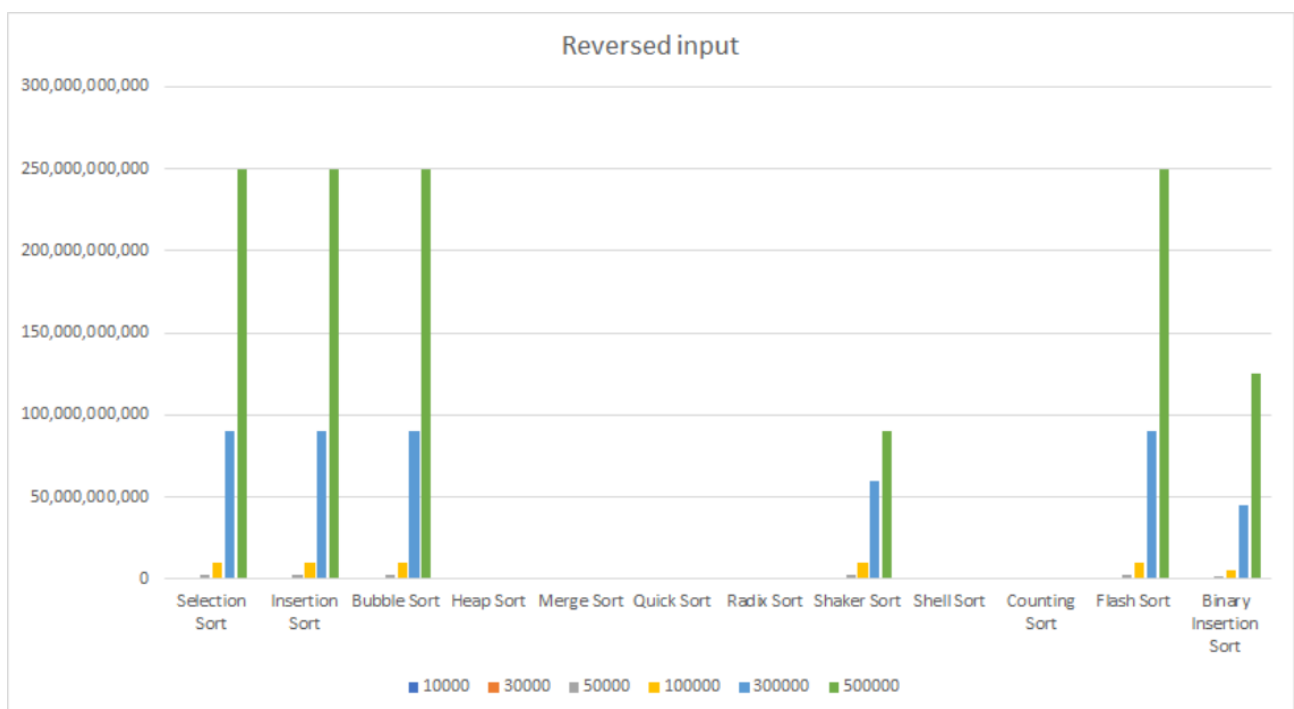
- Leading the slowest-running algorithms are *Bubble Sort* and *Shaker Sort*. The runtime of these two algorithms is very long compared to *Selection Sort*.
- Next, *Selection Sort*, *Insertion Sort* and Binary Insertion Sort have almost equal runtime.





- Like other data, **Merge Sort**, **Heap Sort** and **Quick Sort** algorithms keep the runtime order. Merge Sort runtime is still slower than Heap Sort and Quick Sort.
- With this data, we see that the Shell Sort has a slightly faster speed than **Radix Sort** and **Flash Sort**.
- The fastest algorithm is still **Counting Sort**.

## b. Comparisons



- Like above, **Bubble Sort** and **Selection Sort** still have a huge number of comparisons. This is the worst case of **Insertion Sort** and Binary Insertion Sort, so Insertion Sort has the same number of comparisons as Bubble Sort and Selection Sort. Binary Insertion Sort has half the number of comparisons compared to Insertion Sort.
- This is the worst case of **Flash Sort** but it is also not related to the runtime of the algorithm.

- Although the *Shaker Sort* has a smaller number of comparisons than *Flash Sort*, the speed of Flash Sort is still faster.
- *Heap Sort, Merge Sort, Quick Sort, Radix Sort, Shell Sort, Counting Sort* have a very small number of comparisons when compared to Bubble Sort, Selection Sort, Insertion Sort, Shaker Sort, Flash Sort, Binary Insertion Sort.

## V. ANALYSIS

### 1. Algorithms with $O(n^2)$ complexity

- With these algorithms, *Bubble Sort* has the slowest speed, much slower than other algorithms.
- With a simple installation, *Insertion Sort* and *Selection Sort* have a much better runtime than *Bubble Sort* and *Shaker Sort*.
- If we need to arrange with a small number of components, *Insertion Sort* is a reasonable choice because of its ease of installation.

### 2. Algorithms with other complexity.

- Of the three Algorithms *Merge Sort*, *Quick Sort* and *Heap Sort*, *Merge Sort* has the slowest running speed. In some cases, *Merge Sort's* runtime may be equal to *Heap Sort's* runtime, always has complexity  $O(n \cdot \log_2(n))$ . But in the worst case, the complexity of *Quick Sort* can be up to  $O(n^2)$ .
- *Shell Sort* with runtime is usually faster than *Merge Sort* but slower than *Quick Sort*. The runtime of *Radix Sort* is much faster than the above three algorithms, but it relies on the base of the component, so it will be difficult if the components are not integers.
- *Flash Sort* has complexity  $O(n + m)$  and *Counting Sort* has complexity  $O(n + k)$ . Easy-to-install *Counting Sort* algorithm with lower programming constants than *Flash Sort*.

### 3. Stable sorting algorithms

- Stable algorithms include : *Insertion Sort*, *Binary Insertion Sort*, *Bubble Sort*, *Shaker Sort*, *Merge Sort*, *Counting Sort*.

- Of all stable sorting algorithms, *Counting Sort* has the lowest runtime as well as complexity.

#### **4. Unstable sorting algorithms**

- ✓ Unstable algorithms include : *Selection Sort, Shell Sort, Heap Sort, Quick Sort, Radix Sort, Flash Sort*.
- ✓ Of all unstable sorting algorithms, *Flash Sort* has the lowest runtime.

## VI. PROJECT ORGANIZATION AND PROGRAMMING

- Command line: (how to distinguish)
- We will have 2 types of `argv[1]`: “-a” (algorithm mode: command 1 + command 2 + command 3) & “-c” (comapre mode: command 4 + command 5):
  - **Algorithm mode:**
    - ✓ **Command 1:** Run a sorting algorithm on the given input data.  
→ `argc = 5` and `argv[3]` will be a string meaning file\_name of given input.
    - ✓ We will input data from given file (`argv[3]`) into array and run the program correctly with suitable algorithm, giving result after calculating running time and comparison. (Write down the sorted array to the "output.txt" file).
  - ✓ **Command 2:** Run a sorting algorithm on the data generated automatically with specified size and order. → `argc = 6`.
  - ✓ Unlike command 1, command 2 will automatically create an array with size = `argv[3]` with the appropriate input order from `argv[4]`.
  - ✓ We also need to write down the sorted array to the "output.txt" file and write down the generated input to the "input.txt" file.
- ✓ **Command 3:** Run a sorting algorithm on ALL data arrangements of a specified size. → `argc = 5` but `argv[3]` is an integer number meaning a size of data.

- ✓ We will work through all input orders with size of data = `argv[3]`.

- **Compare mode:**

- ✓ **Command 4:** Run two sorting algorithms on the given input.  
→ `argc = 5`.
- ✓ It's kindly like command 1 but will calculating 2 different algorithms with the same data-size.
- ✓ **Command 5:** Run two sorting algorithms on the data generated automatically. → `argc = 6`.
- ✓ It's kindly like command 2 but will calculating 2 different algorithms with data-size = `argv[3]`.

- We also write some functions to support our command line:

- ✓ **Display Functions:** to display the results, some information from command lines and also write the results in required files.
- ✓ **Calculating Functions:** to generate array by input\_order from command lines, calculate running time + comparisons, import data into array from files,...

- Special libraries:

- + `<time.h>` : to calculating running time.
- + `<algorithm>`: find max and min element in array.
- + `<vector>`:

## VII. REFERENCES

### **Heap sort**

- <https://www.geeksforgeeks.org/heap-sort/>
- Sách Cấu trúc dữ liệu và giải thuật 2003 trường Đại học Khoa Học Tự Nhiên

### **Shell sort**

- <https://www.geeksforgeeks.org/shellsort/>
- Sách Cấu trúc dữ liệu và giải thuật 2003 trường Đại học Khoa Học Tự Nhiên

### **Radix sort**

- <https://www.geeksforgeeks.org/radix-sort/>

### **Quick sort**

- <https://www.geeksforgeeks.org/quick-sort/>
- <https://www.stdio.vn/giai-thuat-lap-trinh/quick-sort-hlmLf1>

### **Counting sort:**

- <https://www.geeksforgeeks.org/counting-sort/>
- <https://www.programiz.com/dsa/counting-sort>
- [https://en.wikipedia.org/wiki/Counting\\_sort](https://en.wikipedia.org/wiki/Counting_sort)

### **Flash sort:**

- [https://github.com/leduythuocs/Sorting-Algorithms/blob/master/sorting-methods/flash\\_sort.h](https://github.com/leduythuocs/Sorting-Algorithms/blob/master/sorting-methods/flash_sort.h)
- <https://youtu.be/CAaDJJUszvE>
- <https://codelearn.io/sharing/flash-sort-thuat-toan-sap-xep-than-thanh>
- <https://stackoverflow.com/questions/2912377/flashsort-algorithm>
- <http://javascript.algorithmexamples.com/web/Sorts/flashSort.html>

### **Bubble sort:**

- <https://www.studytonight.com/data-structures/bubble-sort>
- <https://www.geeksforgeeks.org/bubble-sort/>

**Merge sort:**

- <https://www.studytonight.com/data-structures/merge-sort>
- <https://www.geeksforgeeks.org/merge-sort/>
- <https://nguyenvanhieu.vn/thuat-toan-sap-xep-merge-sort/>

**Shaker sort:**

- <https://www.geeksforgeeks.org/cocktail-sort/>
- <https://www.stdio.vn/giai-thuat-lap-trinh/bubble-sort-va-shaker-sort-01Si3U>

**Selection sort:**

- <https://www.geeksforgeeks.org/selection-sort/>
- <https://www.geeksforgeeks.org/insertion-sort/>

**Binary Insertion sort:**

- <https://www.geeksforgeeks.org/binary-insertion-sort/>
- <https://www.tutorialspoint.com/binary-insertion-sort-in-cplusplus>

**Complexity (time + space + stable):**

- <https://github.com/leduythuucs/Sorting-Algorithms/blob/master/Report-VNM.pdf>
- [https://en.wikipedia.org/wiki/Sorting\\_algorithm#:~:text=In%20computer%20science%2C%20a%20sorting,numerical%20order%20and%20lexicographical%20order.](https://en.wikipedia.org/wiki/Sorting_algorithm#:~:text=In%20computer%20science%2C%20a%20sorting,numerical%20order%20and%20lexicographical%20order.)

**Calculating time:**

- <https://eitguide.net/cach-do-thoi-gian-thuc-thi-cua-mot-function-trong-cc/>