Vietnam National University – Ho Chi Minh city
University of Science
Faculty of Computer Science

COURSE PROJECT

DATA STRUCTURES AND ALGORITHMS

PROJECT of SEMESTER 1
2020 – 2021

SORTING ALGORITHMS ANALYSIS

Students:	**Trương Gia Đạt - 19127017**
Class:	**19CLC10**
Instructor:	**Bùi Huy Thông**

Ho Chi Minh, 2020

CONTENTS

INTRODUCTION

WHAT IS SORTING?

In computer science, a **sorting algorithm** is an algorithm that puts elements of a list in a certain order. The most frequently used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the efficiency of other algorithms (such as search and merge algorithms) that require input data to be in sorted lists. Sorting is also often useful for canonicalizing data and for producing human-readable output. More formally, the output of any sorting algorithm must satisfy two conditions:

1. The output is in nondecreasing order (each element is no smaller than the previous element according to the desired total order);
2. The output is a permutation (a reordering, yet retaining all of the original elements) of the input.

Further, the input data is often stored in an array, which allows random access, rather than a list, which only allows sequential access; though many algorithms can be applied to either type of data after suitable modification.

SORTING ALGORITHMS

1. Selection Sort

Basic idea: Selection Sort finds the minimum value, swaps it with the value in the first position of unsorted array, and repeats these steps for the remainder of the list.

Algorithm:

_Step 1: Set the first element as minimum.

_Step 2: Compare minimum with the second element. If the second element is smaller than minimum, assign the second element as minimum. The progress goes on until the last element.

_Step 3: After each iteration, minimum is placed in the front of the unsorted list.

_Step 4: For each iteration, indexing starts from the first unsorted element. Step 1 to 3 are repeated until all the elements are placed at their correct positions.

Review:

_Selection Sort is easily implemented.

_Comparisons taken: (n - 1) + (n - 2) + (n - 3) + … + 1 = n(n - 1) / 2 nearly equals to n^2.

_Time complexities:

+Worst case complexity: $\boldsymbol{O}(\boldsymbol{n}^2)$**.**

+Best case complexity: $\boldsymbol{O}(\boldsymbol{n}^2)$. When the array is already sorted.

+Average case complexity: $\boldsymbol{O}(\boldsymbol{n}^2)$.

_Space complexity is $\boldsymbol{O}(\mathbf{1})$ because an extra variable temp is used.

_Selection Sort has performance advantages over more complicated algorithms in certain situations. **(*)**

2. Insertion Sort

Basic idea: Insertion Sort which places an unsorted element at its suitable place in each iteration.

Algorithm:

_Step 1: Assuming the first element in the array is already sorted. Take the second element and store it in key. Next, make a comparison between first element and key, if the first element is greater than key, then key is placed in front of the first element.

_Step 2: Now, the first two elements are sorted. We take the third element in unsorted array and compare it with all elements of its left – hand side. Placed it after the element smaller than it. If there doesn’t exist the element smaller than it, then it will be placed at the beginning of the array.

_Step 3: Similarly, do the same work till the last element.

Review:

_Insertion Sort is easily implemented.

_For every n^{th} element, (n – 1) number of comparisons are required. Thus, there will be $n * (n - 1) \sim \boldsymbol{n^2}$ comparisons in total.

_Time complexities:

+Worst case complexity: $\boldsymbol{O(n^2)}$**.**

+Best case complexity: $\boldsymbol{O(n)}$. When the array is already sorted.

+Average case complexity: $\boldsymbol{O(n^2)}$.

_Space complexity is $\boldsymbol{O(1)}$ because an extra variable key is used.

_In the best case (already sorted), every insert requires constant time. **(*)**

3. Bubble Sort

Basic idea: Bubble Sort that compares the adjacent elements and swaps their positions if they are in wrong order.

Algorithm:

_Step 1: Starting at the beginning of the array. Make a comparison between the first two elements, if the 1^{st} element is greater than 2^{nd}element, then swaps them. It continues doing this for each pair of adjacent elements to the last element.

_Step 2: Repeating stages for the remaining iterations. After each iteration, the biggest element amongst the unsorted array is placed at the end.

Review:

_Bubble is a simple sorting algorithm. Two loops are required for this algorithm.

_Comparisons taken: (n - 1) + (n - 2) + (n - 3) + … + 1 = n(n - 1) / 2 nearly equals to n^2.

_Time complexities:

+Worst case complexity: $\boldsymbol{O(n^2)}$**.** When we want to sort in ascending order and the array is in descending order.

+Best case complexity: $\boldsymbol{O(n)}$. When the array is already sorted.

+Average case complexity: $\boldsymbol{O(n^2)}$.

_Space complexity is $\boldsymbol{O(1)}$ because an extra variable temp is used for swapping.

_In ***nearly sorted*** case, Bubble sort can also be used **efficiently** on a list of any length. (*)

4. Heap Sort

Basic idea: Heap Sort works by visualizing the elements of the array as a special kind of **complete binary tree** called a heap.

Algorithm:

_Step 1: Using the **Max – Heap** property, then the largest element will be the root of tree.

_Step 2: Swapped the root (first) element to the end of the array and the last element becomes the new root. Reduce the size of heap by 1.

_Step 3: Repeating Step 1 until all elements of the array are sorted.

Review:

_Heap Sort algorithm can be built in either iterative or recursive version. Recursive version is highly recommended.

_Time complexity of heapify is $\boldsymbol{O(logn)}$**.** During the build **Max – Heap** stage, we do that for n/2 elements so the time complexity of the build heap step is $\boldsymbol{nlogn}$.

_Time complexities: $\boldsymbol{O(nlogn)}$ for 3 cases.

_Heap Sort algorithm performs sorting in $\boldsymbol{O(1)}$ space complexity.

_No matter what cases, Heap Sort algorithm does the same scenario. Hence, there is no huge influence on time complexity. **(*)**

5. Merge Sort

Basic idea: Merge Sort using a kind of **Divide and Conquer** technique in computer programming. It is one of the most popular sorting algorithms and a great way to develop confidence in building recursive algorithms.

Algorithm: There are two main functions required for Merge Sort algorithm.

_MergeSort function repeatedly divides the array into two halves until we reach a stage in which we try to perform MergeSort on a subarray of size 1.

After that, the **merge** function comes into play and combines the sorted arrays into larger arrays unti; the whole array is merged.

Ex: {4, 13, 21, 10} → {4, 13}, {10, 9} → {4}, {13}, {10}, {9}.

_Merge function is in the opposite of **MergeSort** function, it is used for merging two halves in right order.

+Step 1: Create duplicate copies of subarrays to be sorted.

+Step 2: Maintain the current index of subarrays and main array.

+Step 3: Until we reach the end of either L or R, pick larger among elements L and R and place them in the correct position at A[p.. r].

+Step 4: When we run out of elements in either L or R, pick up the remaining elements and put in A[p...r].

Review:

_Merge Sort is one of the fastest sorting algorithms because it uses the **Divide and Conquer** algorithm. Moreover, Heap sort is hardly implemented.

_Every time each subarray is split into a half. Hence, time complexity of this stage is $\boldsymbol{O(logn)}$. Additionally, we need to sort the whole array in expected order so it should be $\boldsymbol{O(n)}$**.** Therefore, the time complexity of Merge Sort algorithm is $\boldsymbol{O(nlogn)}$ for 3 cases.

_The space complexity of merge sort is $\boldsymbol{O(n)}$**.**

_Merge sort takes advantage of the ease of merging **<u>already sorted</u>** lists into a new sorted list. **(*)**

6. Quick Sort

Basic idea: Quick Sort is also an algorithm based on **Divide and Conquer** approach in which the array is split into subarrays and these subarrays are recursively called to sorted the array.

Algorithm:

_Step 1: Choosing the pivot. There are several ways to pick the pivot of the array:

+The first element.

+The last element.

+**The middle element**. (Easy, Common and Recommended).

+Randomly choosing.

_Step 2: Create indices called **Left** and **Right. Left** will be the first element and **Right** will be the last element.

_Step 3: Moving **Left** forward until **Left** is greater than pivot and **Right** backward until **Right** is smaller than pivot. Then, Swapped **Left** and **Right** and return when **Left** and **Right** crossed.

_Step 4: Pivot elements are again chosen for the left and the right sub-parts separately. Within these sub-parts, the pivot elements are placed at their right position. Then, step 2 is repeated.

Review:

_Quick Sort is among the fastest sorting algorithms in practice.

_Time complexities:

+Worst case complexity: $\boldsymbol{O(n^2)}$**.** It occurs when the pivot element picked is either the greatest or the smallest element.

+Best case complexity: $\boldsymbol{O(nlogn)}$**.** The pivot element is always the middle element or near to the middle element.

+Average case complexity: $\boldsymbol{O(nlogn)}$**.**

_ The space complexity for quick sort is $\boldsymbol{O(logn)}$**.**

_ The most complex issue in quicksort is thus choosing a good pivot element, as consistently poor choices of pivots can result in drastically slower $\boldsymbol{O(n^2)}$ performance, but good choice of pivots yields $\boldsymbol{O(nlogn)}$ performance, which is **asymptotically optimal**.

7. Radix Sort

Basic idea: Radix sort is a sorting technique that sorts the elements by first grouping the individual digits of the same **place value**. Then, sort the elements according to their increasing/decreasing order.

Algorithm: Assuming we an array whose elements have 3 – digit.

_Step 1: Find the largest element in the array, **Max**. Let X be the number of digits in **Max**. X is calculated because we have to go through all the significant places of all elements.

_Step 2: Go through each significant place one by one. Use any **stable sorting** technique to sort the digits at each significant place. (Counting sort is recommended).

_Step 3: Sort the elements based on digits at tens place.

_Step 4: Finally, sort the elements based on the digits at hundreds place.

Review:

_ Since radix sort is a non – comparative algorithm, it has advantages over comparative sorting algorithms.

_Time complexity: $\boldsymbol{O(d*(n+k))}$**.** Where $\boldsymbol{O(n+k)}$ is time complexity of Counting Sort, d is the number of digits in each element.

_ The space complexity for radix sort also comes from counting sort $\boldsymbol{O(n+k)}$**.**

_Radix Sort is mostly used to place where there are numbers in **large ranges.**

TABLE OF TIME AND SPACE COMPLEXITY

Algorithm	Data Structure	Best case	Worst case	Average Case	Space complexity
Selection Sort	Array	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	Array	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Bubble Sort	Array	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Heap Sort	Heap	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(1)$
Merge Sort	Array	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(n)$
Quick Sort	Array	$O(nlogn)$	$O(n^2)$	$O(nlogn)$	$O(logn)$
Radix Sort	Array	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Conclusion:

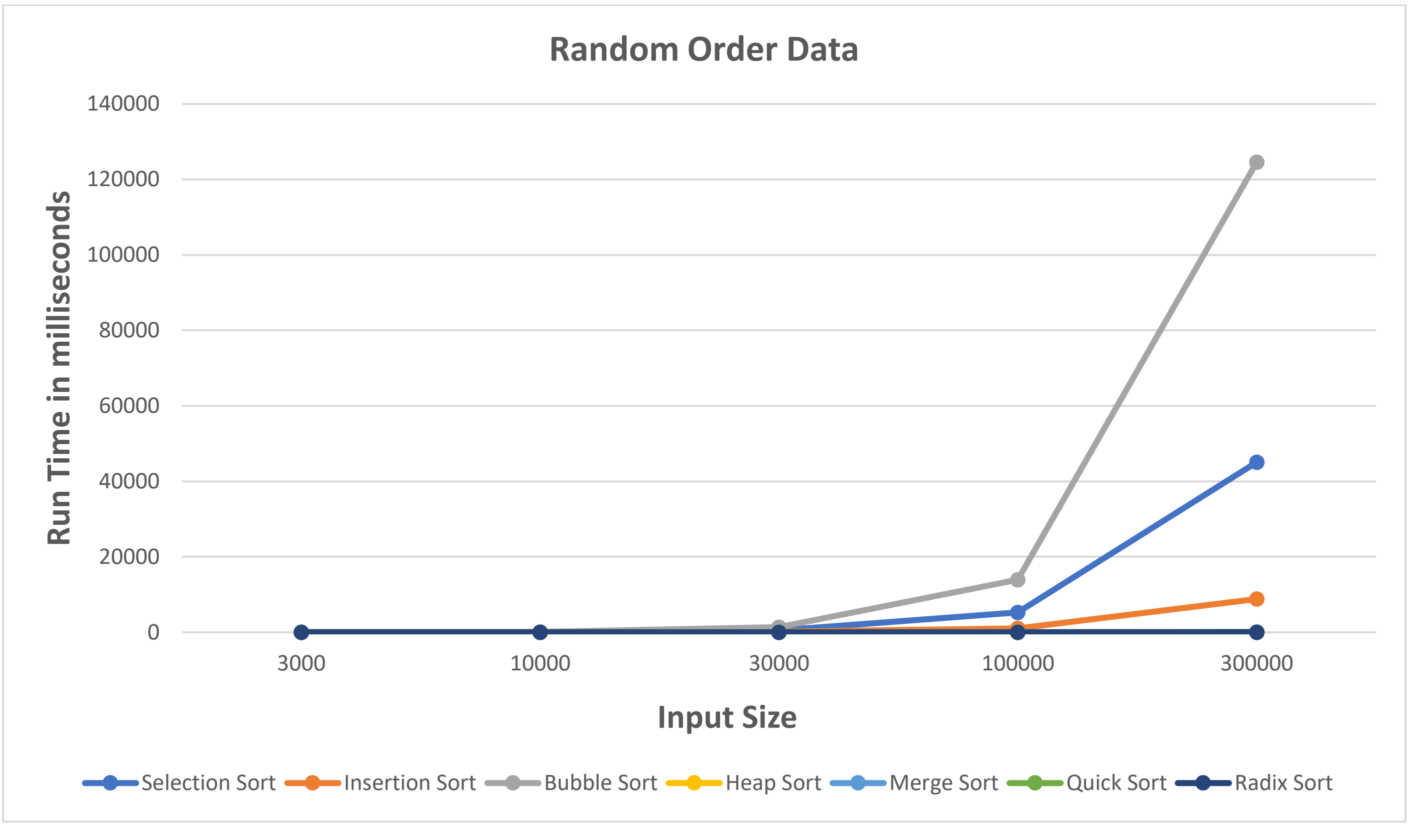
_Selection Sort, Insertion Sort and Bubble Sort are quite straightforward algorithms. They are relatively inefficient in large lists.

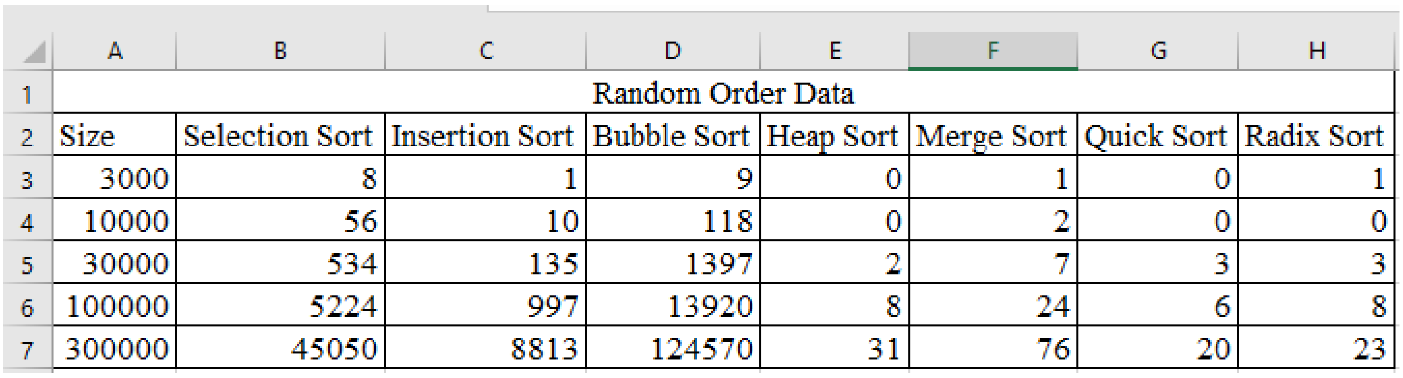
_Heap Sort and Merge Sort are more sophisticated, and work effectively in large lists. The time complexity of two algorithms is the same in all three cases no matter how the array looks like. Merge Sort is one of the stable sorting algorithms.

_Quick Sort is not easy to implement. Even though time complexity of Quick Sort can be $O(n^2)$ for some cases, in practice, Quick Sort is faster than Merge Sort and Heap. Moreover, this algorithm is very efficient when using linked list to implement.

_Radix Sort, a non – comparative algorithm, takes a linear time to sort n integers with a fixed number of bits. Radix Sort cooperates with Counting Sort which is a stable sorting technique to sort the array more effectively. However, Radix sort only works when sorting numbers with a fixed number of digits. It won't work for arbitrarily - large numbers.

THE STATISTICS OF RUNNING TIME

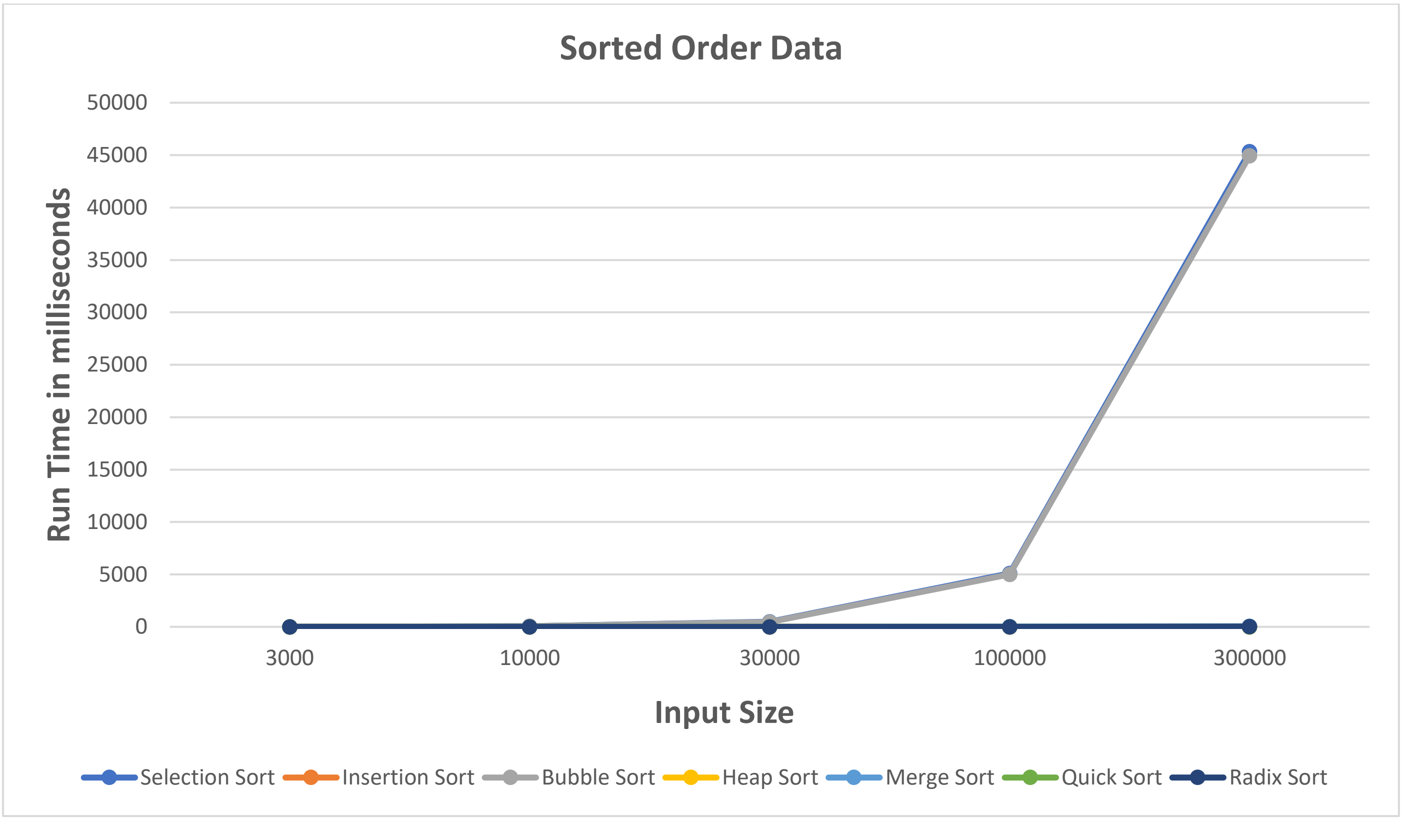


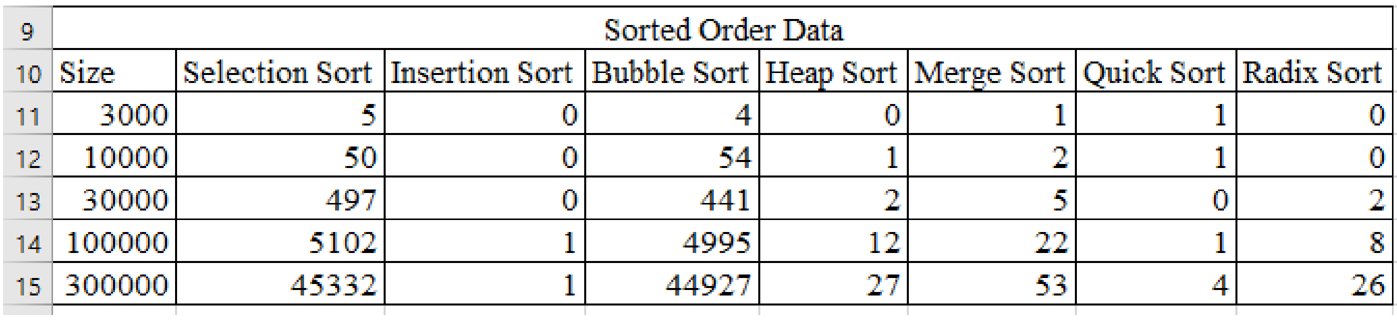


	A	B	C	D	E	F	G	H
1	Random Order Data							
2	Size	Selection Sort	Insertion Sort	Bubble Sort	Heap Sort	Merge Sort	Quick Sort	Radix Sort
3	3000	8	1	9	0	1	0	1
4	10000	56	10	118	0	2	0	0
5	30000	534	135	1397	2	7	3	3
6	100000	5224	997	13920	8	24	6	8
7	300000	45050	8813	124570	31	76	20	23

_In arbitrary data case, Bubble Sort is the slowest sorting algorithm whose time complexity is $\boldsymbol{O}(\boldsymbol{n}^2)$ and a group of four fastest sorting algorithms (Heap Sort, Merge Sort, Quick Sort and Radix Sort).

_The running time of Bubble Sort experienced a dramatic increase from 9 milliseconds in size 3.000 to around 125.000 milliseconds in size 300.000.

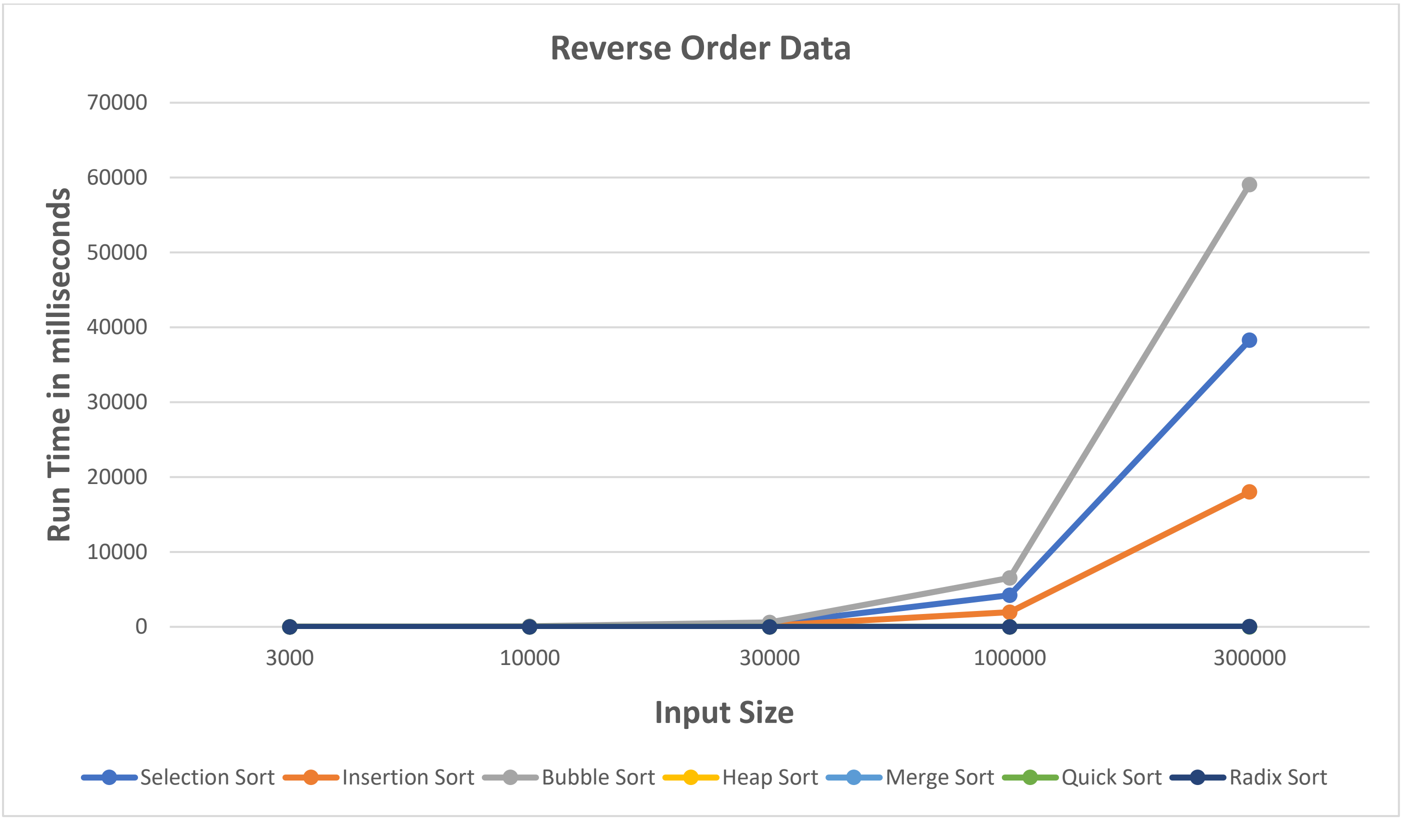


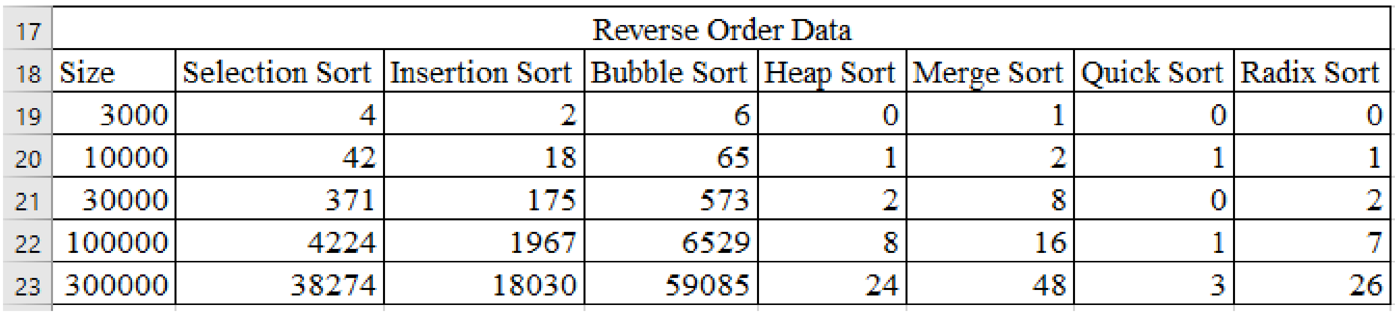


9	Sorted Order Data							
10	Size	Selection Sort	Insertion Sort	Bubble Sort	Heap Sort	Merge Sort	Quick Sort	Radix Sort
11	3000	5	0	4	0	1	1	0
12	10000	50	0	54	1	2	1	0
13	30000	497	0	441	2	5	0	2
14	100000	5102	1	4995	12	22	1	8
15	300000	45332	1	44927	27	53	4	26

_In sorted data case, Selection Sort and Bubble Sort are the slowest sorting algorithms and Insertion Sort is the fastest sorting algorithm compared to the others.

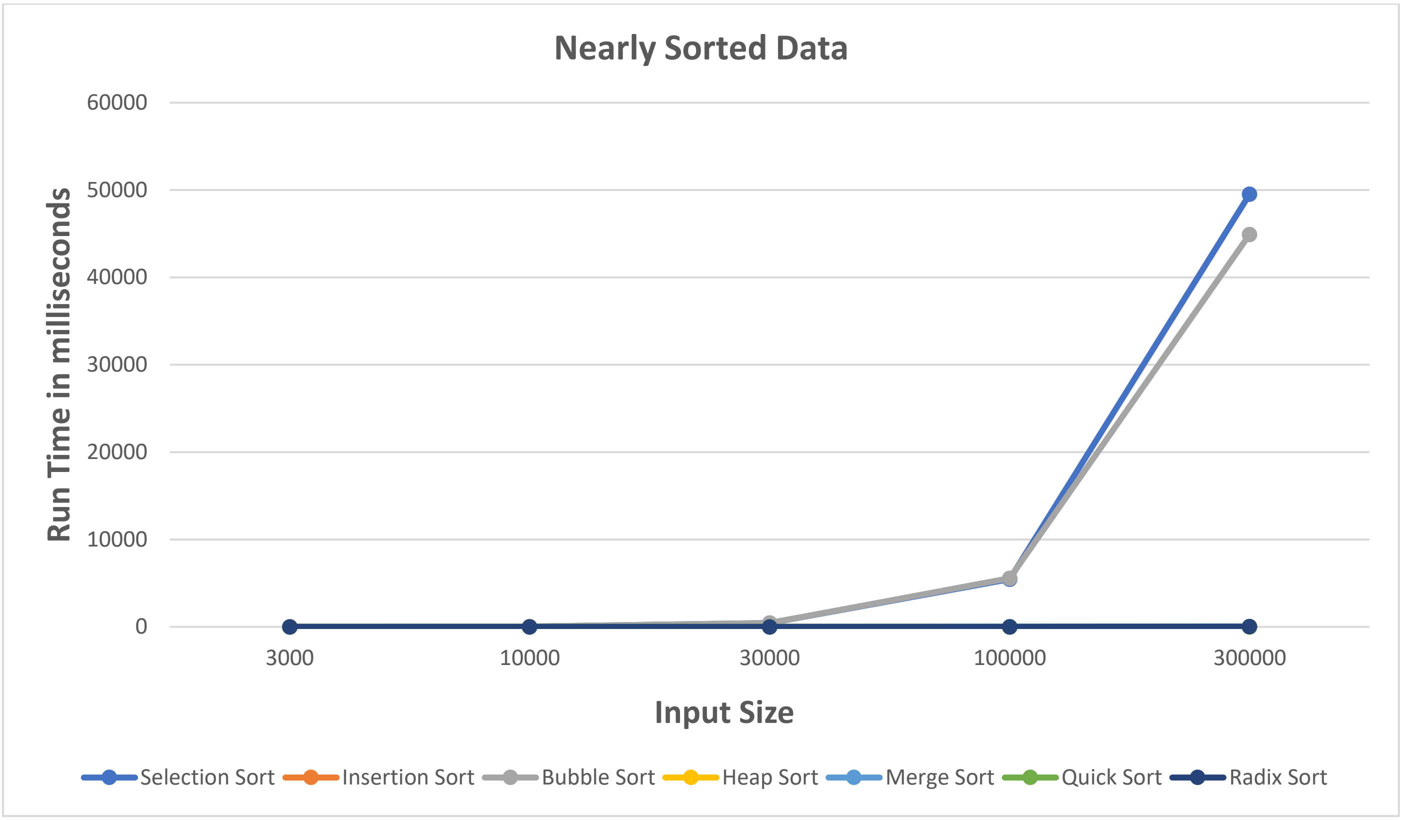
_Even though lists are already sorted, both selection sort and bubble sort still do the same work. Thus, time complexity of two algorithms is $\boldsymbol{O(n^2)}$.

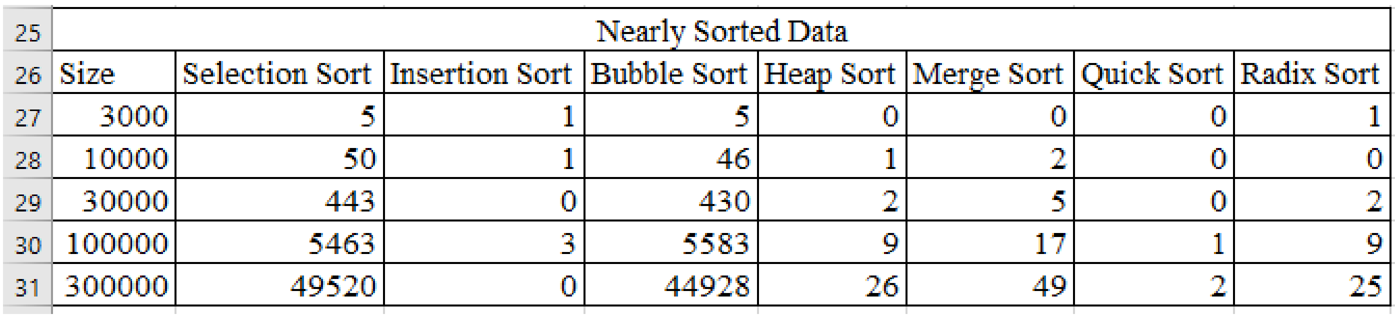




17	Reverse Order Data							
18	Size	Selection Sort	Insertion Sort	Bubble Sort	Heap Sort	Merge Sort	Quick Sort	Radix Sort
19	3000	4	2	6	0	1	0	0
20	10000	42	18	65	1	2	1	1
21	30000	371	175	573	2	8	0	2
22	100000	4224	1967	6529	8	16	1	7
23	300000	38274	18030	59085	24	48	3	26

_In reverse data case, Bubble Sort is the slowest sorting algorithm and Quick Sort is the fastest sorting algorithm.





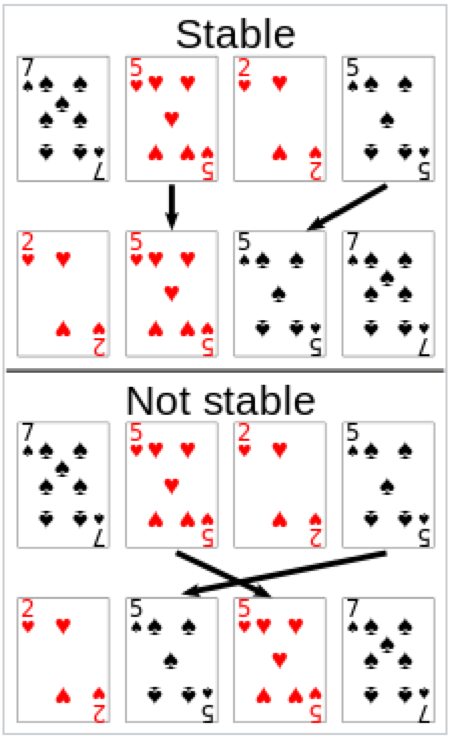
Nearly Sorted Data							
Size	Selection Sort	Insertion Sort	Bubble Sort	Heap Sort	Merge Sort	Quick Sort	Radix Sort
3000	5	1	5	0	0	0	1
10000	50	1	46	1	2	0	0
30000	443	0	430	2	5	0	2
100000	5463	3	5583	9	17	1	9
300000	49520	0	44928	26	49	2	25

_Nearly sorted data case is similar to sorted case.

_Overall, seven sorting algorithms are broken down into 2 groups:

+Stable Sorting Algorithm.

+Unstable Sorting Algorithm.



Stable Algorithms	Unstable Algorithms
Merge Sort, Insertion Sort, Bubble Sort, Radix Sort.	Quick Sort, Heap Sort, Selection sort.

_Ranking Sorting Algorithms based on running time (milliseconds).

1. Quick Sort
2. Radix Sort
3. Merge Sort
4. Heap Sort
5. Insertion Sort
6. Selection Sort
7. Bubble Sort

REFERENCES

1. https://en.wikipedia.org/wiki/Sorting_algorithm#Bubble_sort.
2. https://www.geeksforgeeks.org/sorting-algorithms/.
3. Visualization of 9 Sorting Algorithms: https://www.youtube.com/watch?v=ZZuD6iUe3Pc.
4. Visualization of 24 Sorting Algorithms: https://www.youtube.com/watch?v=BeoCbJPuvSE.