Subject: ITCS 6114 Algorithms and Data Structures

Objective: Comparison-based Sorting Algorithms

Submitted to: Dr. Dewan Tanvir Ahmed

Submitted By:

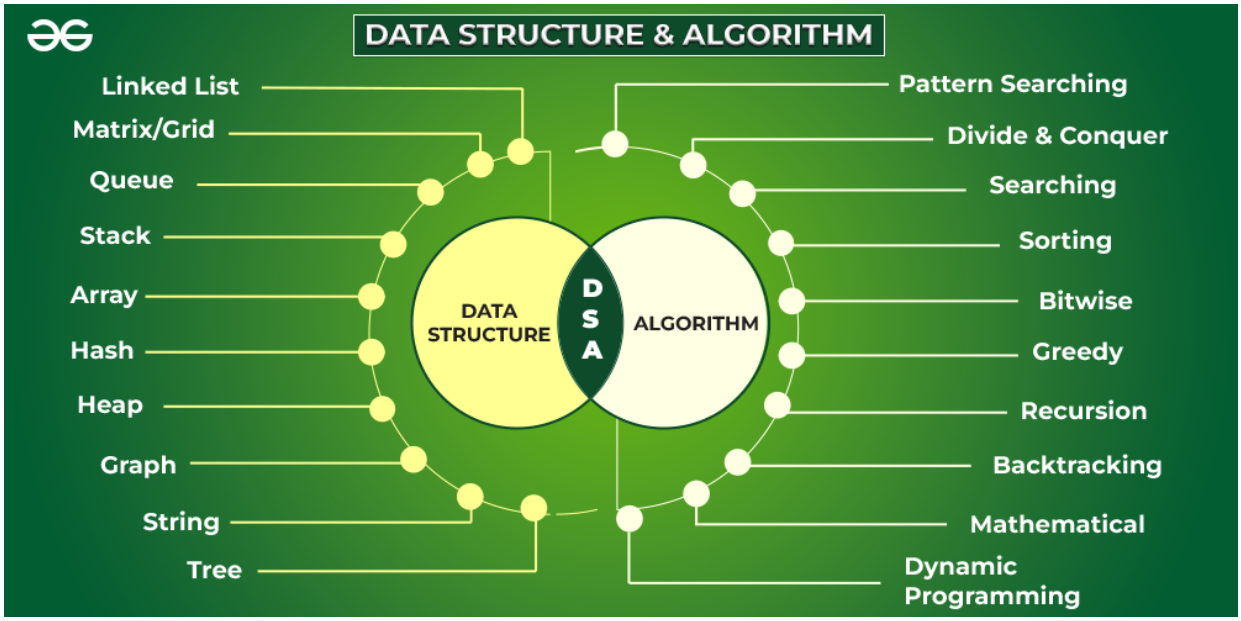
PRACHI MANOJKUMAR SHENDE (801306878)

YASH AGRAWAL (801316596)

INDEX

INTRODUCTION TO DATA STRUCTURES AND ALGORITHMS

A data structure is a collection of data pieces that offers the simplest means of storing and carrying out various operations on computer data. An effective technique to arrange data in a computer is through the use of a data structure. The goal is to simplify various chores in terms of space and time requirements.



Algorithm is defined as a process or set of well-defined instructions that are typically used to solve a particular group of problems or perform a specific type of calculation. To explain in simpler terms, it is a set of operations performed in a step-by-step manner to execute a task.

Time Complexity: The amount of time needed to run the code is determined by its time complexity.
Space Complexity: The quantity of space needed to correctly carry out the capabilities of the code is referred to as space complexity.
The phrase "auxiliary space," which refers to additional space used in the program other than the input data structure, is also used frequently in DSA.

SORTING TECHNIQUES

A Sorting Algorithm is used to rearrange a given array or list of elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.

The sorting techniques studied in this project are the following:

1. Merge Sort
2. Insertion Sort
3. Heap Sort
4. Quick Sort
5. Modified Quick Sort

The above techniques are studied within the below three cases:

1. Rearranged
2. Sorted Ascending Case
3. Reverse Descending Case

MERGE SORT

A sorting algorithm called merge sort divides a large array into smaller subarrays, sorts each subarray individually, and then merges the sorted subarrays back together to create the final sorted array.

The merge sort procedure involves splitting the array in half, sorting each half, and then joining the sorted halves back together. Up till the full array is sorted, this procedure is repeated.

Algorithm:

step 1: start

step 2: declare array and left, right, mid variable

step 3: perform merge function.

```
if left > right
        return
mid=(left+right)/2
mergesort(array,left,mid)
mergesort(array, mid+1,right)
merge(array,left,mid,right)
```

step 4: Stop

Time Complexity: O(nlogn)

Space Complexity: O(n)

INSERTION SORT

Insertion sort functions similarly to how you would arrange playing cards in your hands. In a sense, the array is divided into sorted and unsorted parts. Values are chosen and assigned to the appropriate positions in the sorted part of the data from the unsorted part.

Algorithm:

```
procedure insertionSort(A: list of sortable items)
   n = length(A)
   for i = 1 to n - 1 do
      j = i
      while j > 0 and A[j-1] > A[j] do
         swap(A[j], A[j-1])
         j = j - 1
      end while
   end for
end procedure
```

Time Complexity: O(n2)

Space Complexity: O(n)

HEAP SORT

Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to the selection sort where we first find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements.

In heap sort, basically, there are two phases involved in the sorting of elements. By using the heap sort algorithm, they are as follows -

- The first step includes the creation of a heap by adjusting the elements of the array.
- After the creation of heap, now remove the root element of the heap repeatedly by shifting it to the end of the array, and then store the heap structure with the remaining elements.

Algorithm:

1. HeapSort(arr)
2. BuildMaxHeap(arr)
3. for i = length(arr) to 2
4. swap arr[1] with arr[i]
5. heap_size[arr] = heap_size[arr] ? 1
6. MaxHeapify(arr,1)
7. End

Time Complexity: O(nlogn)

Space Complexity: O (1)

QUICK SORT

QuickSort is a Divide and Conquer algorithm. It chooses an element to act as a pivot and divides the supplied array around it. There are numerous variations of quickSort that select pivot in various ways.

As a rule, choose the first component as the pivot.

Always choose the final component as the pivot (implemented below)

Choose a pivot that is at random.

Decide on median as your pivot.

Partitioning is quickSort's primary operation (). The goal of partitions is to arrange an array with element x serving as the pivot so that all other elements smaller than x are placed before x and all other elements larger than x are placed after x in a sorted array.

Algorithm:

/* This function takes last element as pivot, places the pivot element at its correct position in sorted array, and places all smaller (smaller than pivot) to left of pivot and all greater elements to right of pivot */

```
partition(arr[],low,high)
{
// pivot (Element to be placed at right position)
pivot = arr[high];
i = (low – 1) // Index of smaller element and indicates the
// right position of pivot found so far
for (j = low; j <= high- 1; j++){
        // If current element is smaller than the pivot
        if (arr[j] < pivot) {
```

```
        i++; // increment index of smaller element
        swap arr[i] and arr[j]
        }
}
Swap arr[I + 1] and arr[high]
Return (i+1)
```

Best Case Time Complexity: O(nlogn)

Worst Case Time Complexity: $O(n^2)$

Space Complexity: O(n)

MODIFIED QUICK SORT

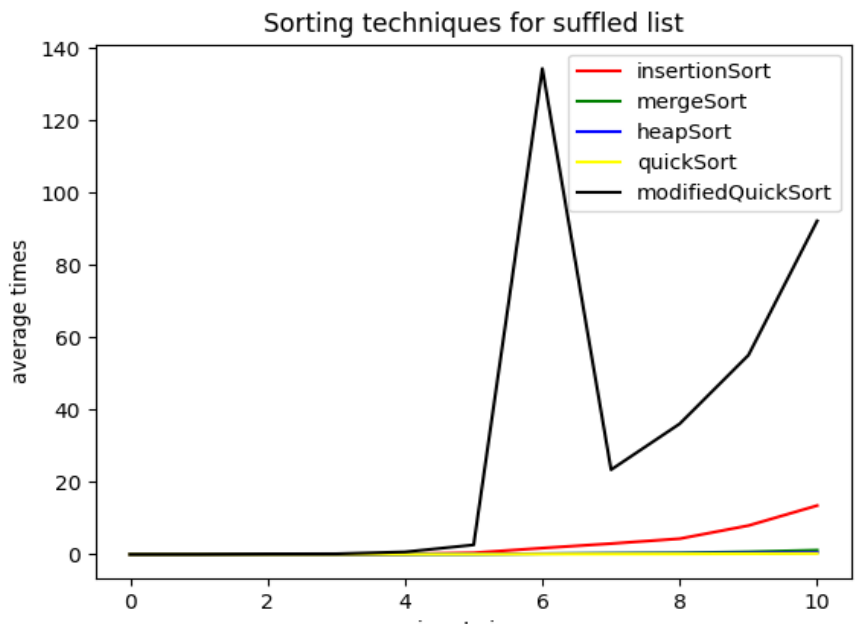
In this sorting strategy, we utilize the median of three components as a pivot. The pivot element is chosen by taking the median of the leftmost, middle, and rightmost elements from the input. Now we switch these components such that the lowest value is on the left, the median would be in the middle, and the highest value is on the right.

Best Case Time Complexity: O(nlogn)

Worst Case Time Complexity: $O(n^2)$

Space Complexity: O(n)

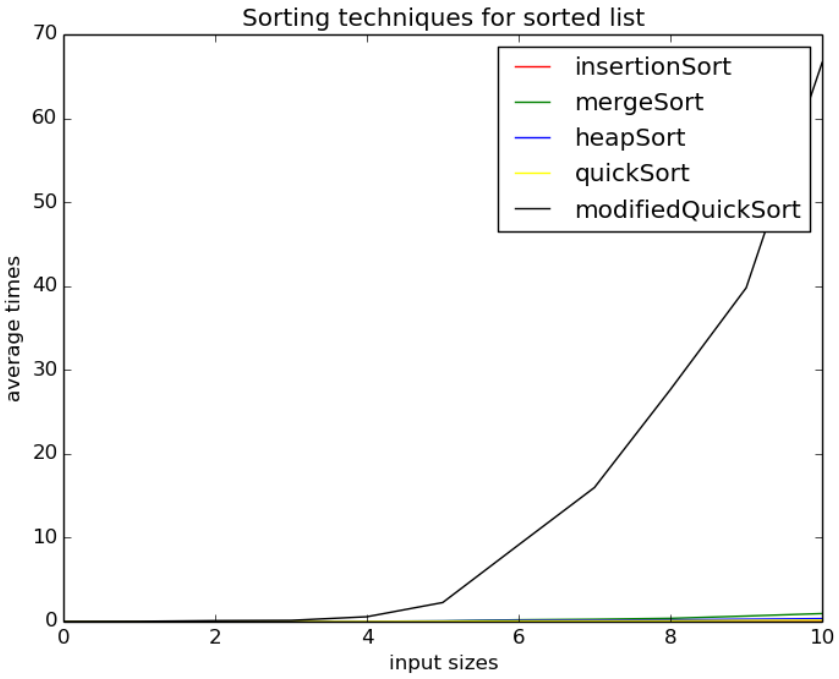
COMPARISON



Case 1: Rearranged Case

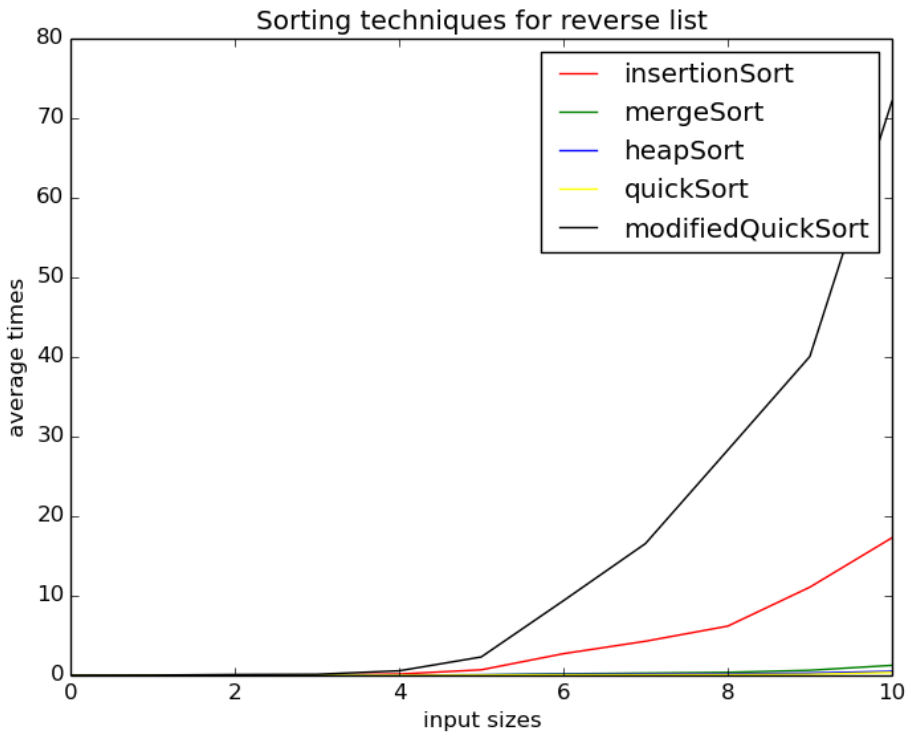
Input Size	Merge Sort	Insertion Sort	Heap Sort	Quick Sort	Modified Quick Sort
1000	0.007	0.003	0.0042	0.005	0.068
2000	0.038	0.043	0. 079	0.067	0.065
4000	0.379	1.074	0.128	0.157	0.126
6000	0.486	3.672	0.344	0.215	128.45
8000	0.592	5.763	0.487	0.306	34.534
10000	0.634	12.635	0.532	0.443	93.534

Case II: Sorted Case (Ascending)



Sr no	Merge Sort	Insertion Sort	Heap Sort	Quick Sort	Modified Quick Sort
1000	0.000	0.000	0.000	0.000	0.000
2000	0.000	0.000	0.003	0.002	0.005
4000	0.000	0.000	0.005	0.021	1.988
6000	0.0012	0.043	0.012	0.039	10.654
8000	0.0032	0.052	0.143	0.054	38.976
10000	1.231	0.065	0.497	0.067	69.865

Case III: Reverse Case (Descending)



Sr no	Merge Sort	Insertion Sort	Heap Sort	Quick Sort	Modified Quick Sort
1000	0.127	0.027	0.008	0.005	0.076
2000	0.569	0.342	0.043	0.052	0.543
4000	0.078	1.643	0.125	0.121	1.533
6000	0.098	4.986	0.612	0.434	14.764
8000	0.367	8.643	0.813	0.567	37.667
10000	0.458	18.532	0.965	0.985	74.754

CONCLUSION, OBSERVATION AND PROJECT OUTCOMES

All of the average times show that rapid sort, merge sort, and heap sort have the fastest execution times. They maintain a steady pace and do not deviate from it under any circumstances. They are excellent because they complete tasks more quickly and with reduced memory usage.

Observation:

The time of execution of insertion sort is: **220.00121402740479**

Input Size: 10000 The time of execution for merge sort is: **0.06690526008605957**

Default Array: **[8 5 5 ... 4 7 7]**

The sorted array is: **[1 1 1 ... 9 9 9]**

The time complexity for Quicksort is: **1.2773759365081787**

The time of execution of heapsort is: **0.033318281173706055**

Project Outcomes:

We have gained a practical understanding of the sorting algorithms, their time and space complexities and their comparative performances.

Gained a hands-on practice on algorithms using python as a core programming language.

Through the assignment we have also acquired technical writing skills while documenting the algorithms.

REFERENCES

To understand the data structures thoroughly we have referred to various online open-source programming portals listed below:

1. https://www.tutorialspoint.com/data_structures_algorithms/index.htm
2. https://www.geeksforgeeks.org/introduction-to-data-structures/
3. https://www.w3schools.in/data-structures/intro
4. Resources for Troubleshooting: https://stackoverflow.com/