Bharatiya Vidya Bhavan’s
Sardar Patel Institute of Technology
Bhavan’s Campus, Munshi Nagar, Andheri (West), Mumbai-400058-India
(Autonomous College Affiliated to University of Mumbai)

Computer Engineering Department &
Information Technology Engineering Department

Academic Year: 2021-2022

Class: S.Y.B.Tech **Sem.:** 4 **Course:** DAA

Name	**Pratik Pujari**		
UID no.	**2020300054**	**Class:**	**Comps C Batch**
ISE	Sorting Experiment		

AIM:	To sort 1million elements using different sorting algorithm to compare the time complexities of different algorithms
ISE EXPERIMENT	
PROBLEM STATEMENT:	In this assignment, the team must evaluate various sorting algorithms and determine which one is most suitable for organizing the various elements in a given file. To generate the elements, divide the file into multiple batches and sort the elements in each batch.
FILE STRUCTURE:	**Folder “unsorted**”: It contains all the files which are unsorted or shuffled and ready for sorting. **Folder “sorted**”: It contains all the files which are sorted by a particular algorithm **Folder output**: It contains the final output of the sorting program **Folder sort**: It contains all the sorting functions like selection,insertion…etc **Java Files** **Driver.java** - This files maintains all the calling and object creation of the function from different programs **fileCreator.java** - This file make and deletes all the previously generated output files and keeps track of generated numbers

	fileReader.java - This file has function that reads from files and puts into array **fileWriter.java** - This java files writes into files for unsorted and sorted arrays to be stored **numberGenerator.java** - This files generates random /unique numbers upto 1 million **fileMerger.java** - This files merges all the files output into one array and outputs into a file **sortAssign.java** - -This files generated the order/type of sort to be done on a file
CALCULATION:	In this assignment, we have used: 1) Merge Sort 2) Quick Sort 3) Heap Sort 4) Selection Sort 5) Radix Sort 6) Bubble Sort 7) Insertion Sort **MERGE SORT** Merge Sort is a divide and conquer algorithm. At every recursive stage, we divide the array into 2 subarrays, call merge sort on both halves and then merge the 2 resultant sorted subarrays into a single sorted array. The special feature of this algorithm is that the time complexity is O(n log(n)) for all 3 cases, including the worst-case which is very efficient.

Statistic	Value
Time Complexity: Best Case	Ω (n log(n))
Time Complexity: Average Case	θ (n log(n))
Time Complexity: Worst Case	O (n log(n))
Space Complexity	O(n)

The Merge Sort algorithm repeatedly divides the array into two halves until we reach a stage where we try to perform Merge Sort on a subarray of size 1 i.e. p == r.

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

Merge Sort Algorithm

MERGE_SORT(arr, beg, end)

1. START
2. If beg < end

A. set mid = (beg + end)/2
B. MERGE_SORT(arr, beg, mid)
C. MERGE_SORT(arr, mid + 1, end)
D. MERGE (arr, beg, mid, end)

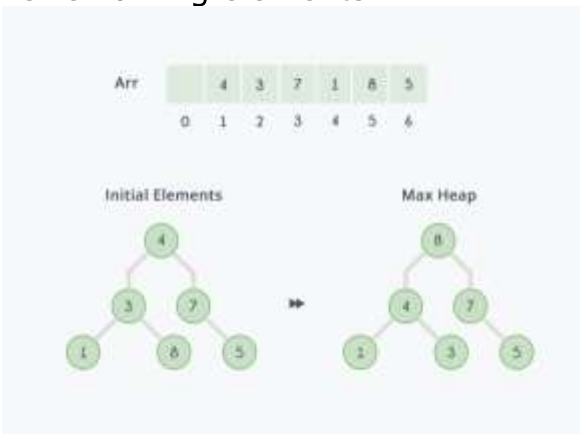
3. STOP

Reason for using merge sort

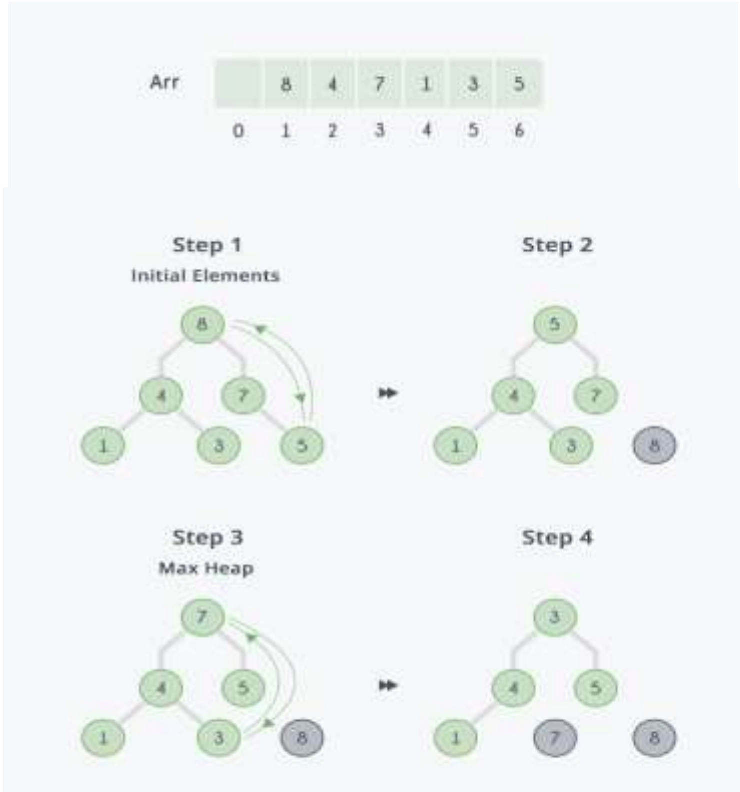
We chose this algorithm because it is fast, efficient and along with QuickSort, it is one of the most
famous divide and conquer sorting algorithms. Since the number of elements in each file is large, the n log n runtime is extremely important for us.

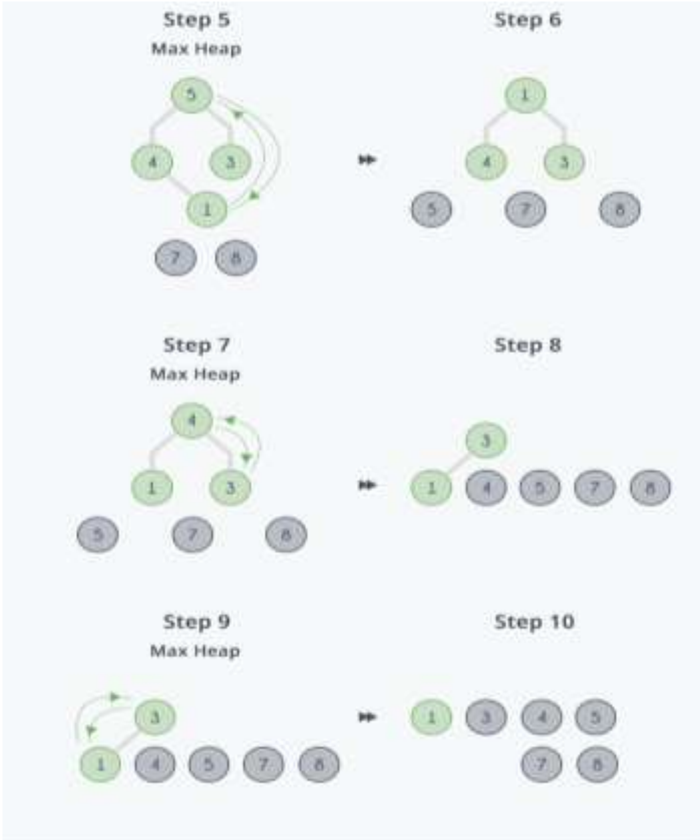
HEAP SORT

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



After building max-heap, the elements in the array Arr will be:





After all the steps, we will get a sorted array.

Algorithm

HeapSort

1) For i going from n/2 to 0
   a. Heapify(arr, n, i)

2) For i going from n-1 to 1
   a. Swap arr[0] and arr[i]
   b. Heapify(arr, i, 0)

Heapify(arr, n, i)

1) Get index of largest element between i and its left and right child
2) If i is not the largest element
   a. Swap i and the largest element
   b. heapify(arr, n, largest)

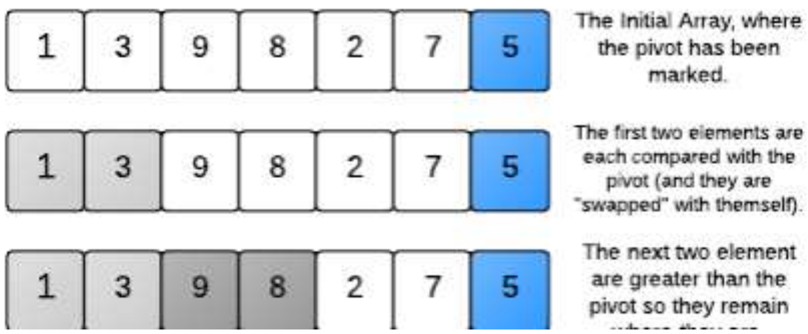
Reason for using Heap Sort

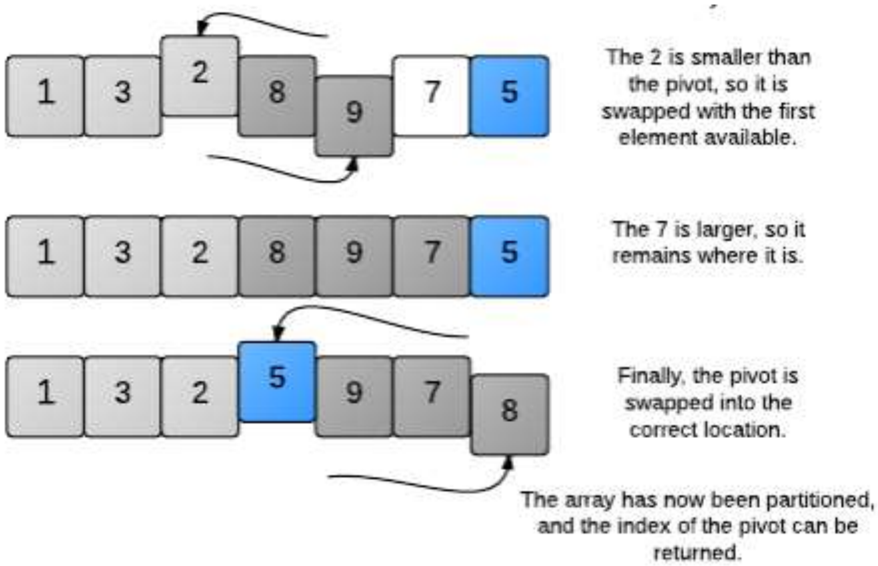
Like merge sort, Heap Sort too has an O (n log n) runtime complexity in all cases. But unlike merge sort, it has a constant space complexity which means that it does not require any extra space other than that required to store the original array. However speed-wise it is slower than quick and merge sort. Overall, it is still much more efficient than n2 algorithms and is a good choice.

QUICK SORT

Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and
partitions the given array around the picked pivot. There are many different versions of quick sort
that pick pivot in different ways. We have decided to always pick the first element as a pivot.
The most important function in quick sort is partition(). The array is partitioned in such a way such that at every step, all elements to the left of the pivot are less than it and all elements to the right of the pivot are greater than it.

Partitioning an array





Note: In the above example the last element is taken as the pivot.

Statistic	Value
Time Complexity: Best Case	Ω (n log(n))
Time Complexity: Average Case	θ (n log(n))
Time Complexity: Worst Case	O (n^2)
Space Complexity	O(1)

Reason for using Quick Sort
As seen above, Quick Sort offers nLogn time complexity in the best and average cases and n2
in the worst case. This may seem lower than merge sort and heap sort which offer nLogn time complexity in the worst case as well. However, in practice, having worst-case time complexity with Quick Sort is rare and can be avoided by choosing a random pivot. Furthermore, Quick Sort is much faster and much more efficient compared to other sorting algorithms since it uses sequential access and the inner loop can run very efficiently on modern architectures.

INSERTION SORT
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.

The first step involves the comparison of the element in question with its adjacent
element.

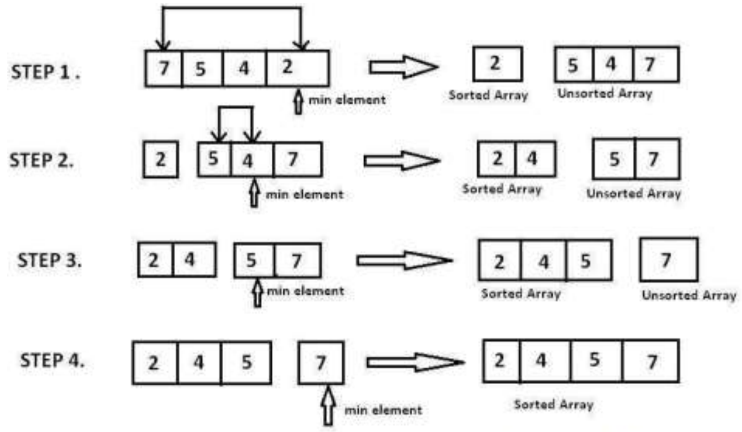
And if at every comparison reveals that the element in question can be inserted at a particular position, then space is created for it by shifting the other elements one position to the right and inserting the element at the suitable position. The above procedure is repeated until all the element in the array is at their apt position. Let us now understand working with the following example:
For example: 4,3,2,10,12,1,5,6

*credits to tutorialspoint

Reason for using Insertion Sort
Used Insertion Sort Algorithm as it keeps giving the element its proper position in array
as soon as it enters .Only applied it on 3-4 files as worst and avg case complexity are
O(n2) which isn't good for large data sets as for worst case the entered element has to
be compared with every element before it

SELECTION SORT
Selection sort is a simple sorting algorithm. This sorting algorithm is an inplace comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

Let's take a look at the implementation.
At ith iteration, elements from position 0 to i−1 will be sorted.



credits to google

Reason for using Selection Sort
Used Selection sort because of its basic algorithm but only on two -three files of
considerably small data sets as time complexity being O(n2) it is not suitable for large
amount of data as for each iteration minimum element is being compared with every
element

Radix Sort:

Now, let's see the working of Radix sort Algorithm.

The steps used in the sorting of radix sort are listed as follows -

- First, we have to find the largest element (suppose max) from the given array. Suppose 'x' be the number of digits in max. The 'x' is calculated because we need to go through the significant places of all elements.
- After that, go through one by one each significant place. Here, we have to use any stable sorting algorithm to sort the digits of each significant place.

Now let's see the working of radix sort in detail by using an example. To understand it more clearly, let's take an unsorted array and try to sort it using radix sort. It will make the explanation clearer and easier.

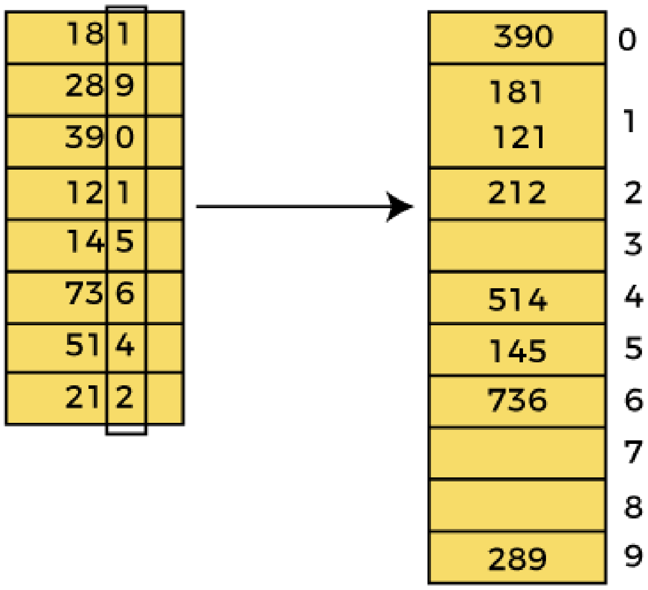
181	289	390	121	145	736	514	212

In the given array, the largest element is 736 that have 3 digits in it. So, the loop will run up to three times (i.e., to the hundreds place). That means three passes are required to sort the array.

Now, first sort the elements on the basis of unit place digits (i.e., x = 0). Here, we are using the counting sort algorithm to sort the elements.

Pass 1:

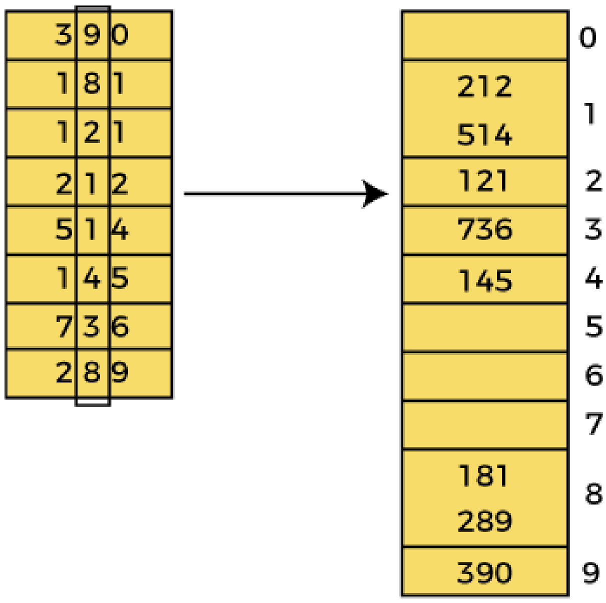
In the first pass, the list is sorted on the basis of the digits at 0's place.



After the first pass, the array elements are -

390	181	121	212	514	145	736	289

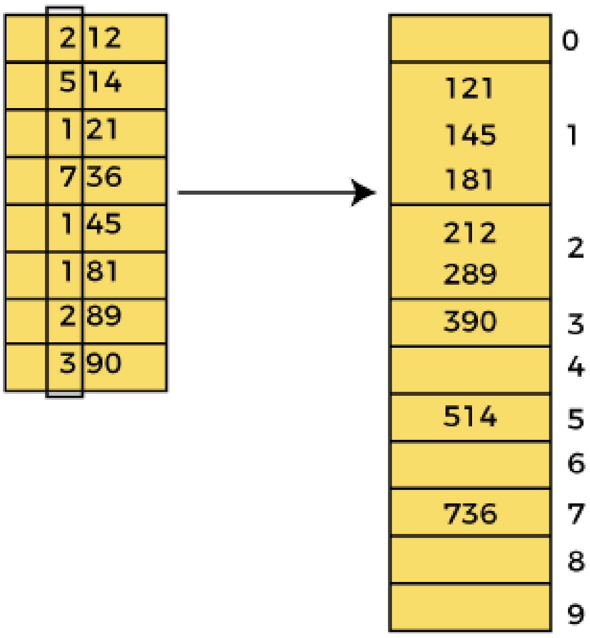
Pass 2:
In this pass, the list is sorted on the basis of the next significant digits (i.e., digits at 10^{th} place).



After the second pass, the array elements are -

212	514	121	736	145	181	289	390

Pass 3:
In this pass, the list is sorted on the basis of the next significant digits (i.e., digits at 100^{th} place).



After the third pass, the array elements are -

121	145	181	212	289	390	514	736

Now, the array is sorted in ascending order.

Why Radix Sort should be used?

- Radix sort only applies to integers, fixed size strings, floating points and to "less than", "greater than" or "lexicographic order" comparison predicates, whereas comparison sorts can accommodate different orders.
- k can be greater than log N.

Time Complexity

Best Case Complexity - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of Radix sort is Ω(n+k).

- Average Case Complexity - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of Radix sort is θ(nk).
- Worst Case Complexity - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of Radix sort is O(nk).

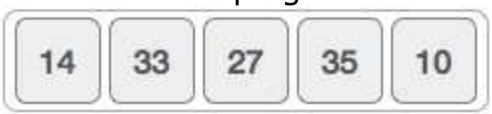
Radix sort is a non-comparative sorting algorithm that is better than the comparative sorting algorithms. It has linear time complexity that is better than the comparative algorithms with complexity O(n logn).

BUBBLE SORT

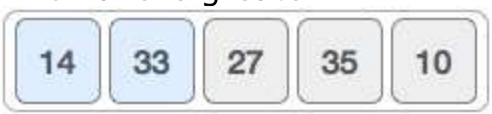
Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where **n** is the number of items.

How Bubble Sort Works?

We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.

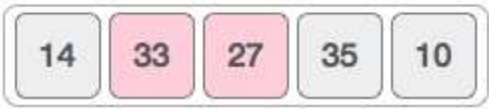


Bubble sort starts with very first two elements, comparing them to check which one is greater.



In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

We find that 27 is smaller than 33 and these two values must be swapped.



The new array should look like this −

Next we compare 33 and 35. We find that both are in already sorted positions.

Then we move to the next two values, 35 and 10.

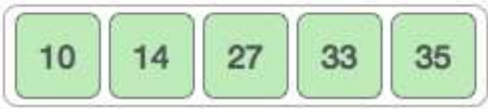
We know then that 10 is smaller 35. Hence they are not sorted.

We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −

Notice that after each iteration, at least one value moves at the end.

And when there's no swap required, bubble sorts learns that an array is completely sorted.



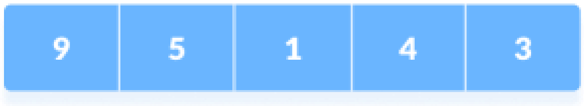
Now we should look into some practical aspects of bubble sort.

PSEUDOCODE:
```
begin BubbleSort(int arr[],int n):
    for i=0 to i< (n-1)
         if arr[i] is greater than arr[i+1]
             swap arr[i] and arr[i+1]

     if n-1 is greater than 1 return BubbleSort(arr,n-1)
```

Worst Case Time Complexity
Θ(N^2) is the Worst Case Time Complexity of Bubble Sort.
This is the case when the array is reversely sort
The number of swaps of two elements is equal to the number of comparisons in this case as every element is out of place.
Best Case Time Complexity
Θ(N) is the Best Case Time Complexity of Bubble Sort.
This case occurs when the given array is already sorted.
T(N)=C(N)=NT(N)=C(N)=N
S(N)=0S(N)=0

Average Case Time Complexity
Θ(N^2) is the Average Case Time Complexity of Bubble Sort.
The number of comparisons is constant in Bubble Sort so in average case, there is O(N^2) comparisons. This is because irrespective of the arrangement of elements, the number of comparisons C(N) is same.

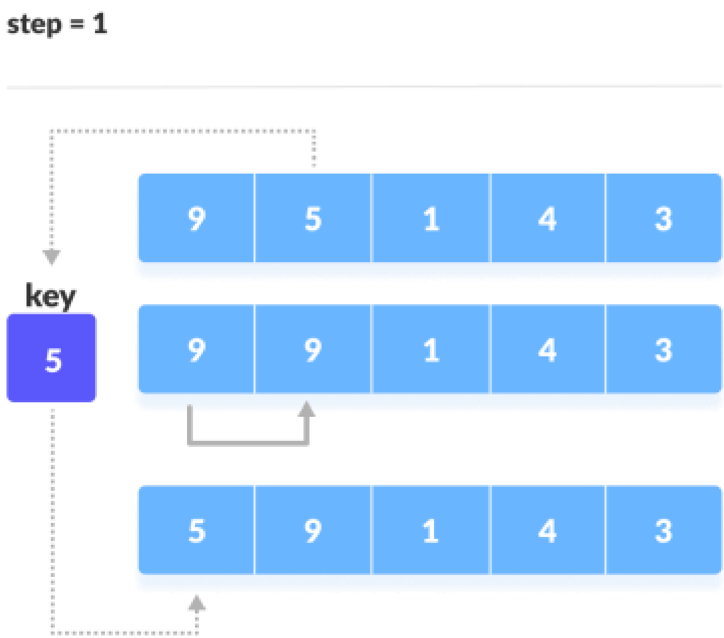
Insertion sort:
Working of Insertion Sort
Suppose we need to sort the following array.



Initial array

1. The first element in the array is assumed to be sorted. Take the second element and store it separately in key.

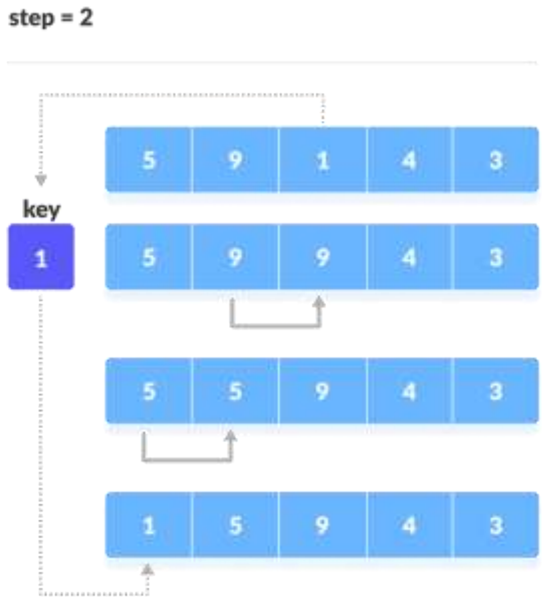
   Compare key with the first element. If the first element is greater than key, then key is placed in front of the first element.



If the first element is greater than key, then key is placed in front of the first element.

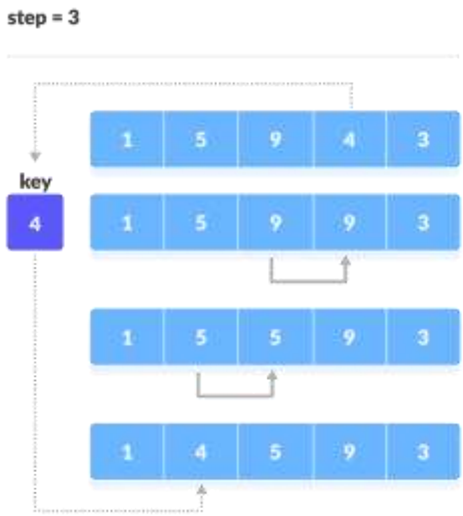
2. Now, the first two elements are sorted.

   Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array.

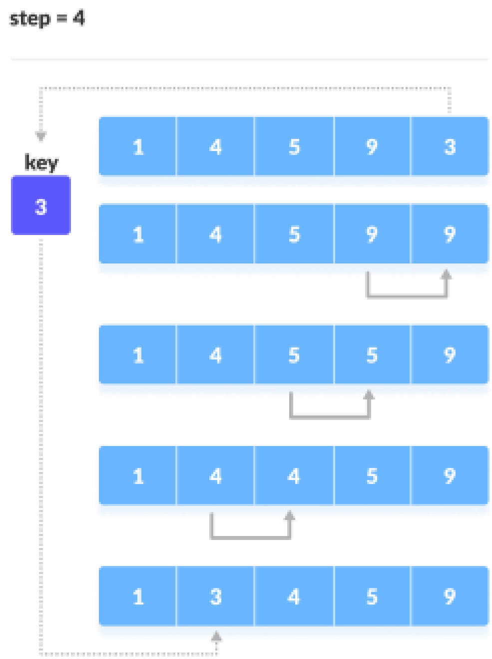


Place 1 at the beginning

3. Similarly, place every unsorted element at its correct position.



Place 4 behind 1



Place 3 behind 1 and the array is sorted

Insertion Sort
Insertion sort is a simple sorting algorithm that is relatively efficient for small lists and mostly sorted lists, and is often used as part of more sophisticated algorithms. It works by taking elements from the list one by one and inserting them in their correct position into a new sorted list similar to how we put money in our wallet. In arrays, the new list and the remaining elements can share the array's space, but insertion is expensive, requiring shifting all following elements over by one. Shellsort (see below) is a variant of insertion sort that is more efficient for larger lists.

Time Complexity
The worst case time complexity of Insertion sort is O(N^2)
The average case time complexity of Insertion sort is O(N^2)
The time complexity of the best case is O(N).

	The space complexity is O(1) Working Principle • Compare the element with its adjacent element. • If at every comparison, we could find a position in sorted array where the element can be inserted, then create space by shifting the elements to right and insert the element at the appropriate position. • Repeat the above steps until you place the last element of unsorted array to its correct position. **Best Case Analysis** In Best Case i.e., when the array is already sorted, $t_j = 1$ Therefore, $T(n) = C_1 * n + (C_2 + C_3) * (n - 1) + C_4 * (n - 1) + (C_5 + C_6) * (n - 2) + C_8 * (n - 1)$ which when further simplified has dominating factor of n and gives $T(n) = C * (n)$ or $O(n)$ **Worst Case Analysis** In Worst Case i.e., when the array is reversly sorted (in descending order), $t_j = j$ Therefore, $T(n) = C_1 * n + (C_2 + C_3) * (n - 1) + C_4 * (n - 1) (n) / 2 + (C_5 + C_6) * ((n - 1) (n) / 2 - 1) + C_8 * (n - 1)$ which when further simplified has dominating factor of n^2 and gives $T(n) = C * (n^2)$ or $O(n^2)$ **Average Case Analysis** Let's assume that $t_j = (j-1)/2$ to calculate the average case Therefore, $T(n) = C_1 * n + (C_2 + C_3) * (n - 1) + C_4/2 * (n - 1) (n) / 2 + (C_5 + C_6)/2 * ((n - 1) (n) / 2 - 1) + C_8 * (n - 1)$ which when further simplified has dominating factor of n^2 and gives $T(n) = C * (n^2)$ or $O(n^2)$ 1. ALGO: 1. Find the smallest card. Swap it with the first card. 2. Find the second-smallest card. Swap it with the second card.

3. Find the third-smallest card. Swap it with the third card.
4. Repeat finding the next-smallest card, and swapping it into the correct position until the array is sorted.

Sorting Algorithm	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$
Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(1)$
Insertion Sort	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$

EXECUTION:

- Initially when the Driver File is executed User is promted with the question of number of elements and number of files
- User then is can see the files being created with random / unique numbers in it.
- If there are existing sorted files. It is deleted in advance to avoid any misunderstanding
- User then chooses the type of algorithm to be executed on files
- The files invidually generated are sorted and then mergesort at the last to generate the last ouput file of sorted elements
- Finally the user can see all the sorted/unsorted files along with the long list of output.txt which has all sorted elements in it
- This program has the capacity to alter the sorting methods, add new method without any major changes.
- File creation is also variable and number of elements is also changeable according to the user too.
- Error handling part where the user enters more files than elements is also implement along with switch case default section.

<table>
<tr>
<td>OUTPUT TABLE:</td>
<td>
<pre>
Enter the number of elements to be sorted: 1000000

Enter the number of files to be created: 13

1.Unique Sorting
2.Random Wise Sorting   :1

Generating Numbers...
Finished Generating numbers Time=0.109375s
Shuffled Finished Time =0.34375s
----------FILE MAKING----------

----------WRITING TO FILES----------

Finished writing to file numbers-1.txt
Finished writing to file numbers-2.txt
Finished writing to file numbers-3.txt
Finished writing to file numbers-4.txt
Finished writing to file numbers-5.txt
Finished writing to file numbers-6.txt
Finished writing to file numbers-7.txt
Finished writing to file numbers-8.txt
Finished writing to file numbers-9.txt
Finished writing to file numbers-10.txt
Finished writing to file numbers-11.txt
Finished writing to file numbers-12.txt
Finished writing to file numbers-13.txt

Time taken for writing files = 11677.0ms
</pre>
Time taken for the files is shown
User has the ability to choose between Math.random or generation of 1-(user number)
Shows the time duration of generating number and shuffling time too.
</td>
</tr>
</table>

```
----------SORTING----------

1.Manual Sorting
2.Automatic Sorting      :1

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


Enter the sorting method: 4

|      Sort-Method      |    Time Taken          |       File Name      |
|       Quick Sort      |  94.0 ms               |       numbers-1.txt  |
|       Quick Sort      |  60.0 ms               |       numbers-2.txt  |
|       Quick Sort      |  25.0 ms               |       numbers-3.txt  |
|       Quick Sort      |  67.0 ms               |       numbers-4.txt  |
|       Quick Sort      |  17.0 ms               |       numbers-5.txt  |
|       Quick Sort      |  59.0 ms               |       numbers-6.txt  |
|       Quick Sort      |  18.0 ms               |       numbers-7.txt  |
|       Quick Sort      |  28.0 ms               |       numbers-8.txt  |
|       Quick Sort      |  20.0 ms               |       numbers-9.txt  |
|       Quick Sort      |  34.0 ms               |       numbers-10.txt |
|       Quick Sort      |  43.0 ms               |       numbers-11.txt |
|       Quick Sort      |  34.0 ms               |       numbers-12.txt |
|       Quick Sort      |  53.0 ms               |       numbers-13.txt |
Total time taken for all sorts = 552.0ms
```

Auto sorting is also an option where the user can see different sorting algorithms done on various files at the same time

User can also see the different duration of the sorting of each files

This varies as the number of files varies

```
----------MERGING----------


Merge Sort:
Merging Files 1-2 : 0.0 ms
Merging Files 1-3 : 0.0 ms
Merging Files 1-4 : 0.0 ms
Merging Files 1-5 : 0.0 ms
Merging Files 1-6 : 0.0 ms
Merging Files 1-7 : 0.0 ms
Merging Files 1-8 : 0.0 ms
Merging Files 1-9 : 0.0 ms
Merging Files 1-10 : 0.0 ms
Merging Files 1-11 : 0.0 ms
Merging Files 1-12 : 0.0 ms
Merging Files 1-13 : 0.0 ms

Total Time taken for Merging Files = 0ms
----------TOTAL TIME----------
TOTAL TIME FOR ALL THE SORTING IS : 12229.453125ms
```

Total time execution time excluding the user input time is shown

Merge sort takes very less time when merging the files and writing into output.txt

For number of files -15

```
Enter the number of elements to be sorted: 1000000

Enter the number of files to be created: 15

1.Unique Sorting
2.Random Wise Sorting     :1

Generating Numbers...
Finished Generating numbers Time=0.1171875s
Shuffled Finished Time =0.3515625s
----------FILE MAKING----------
```

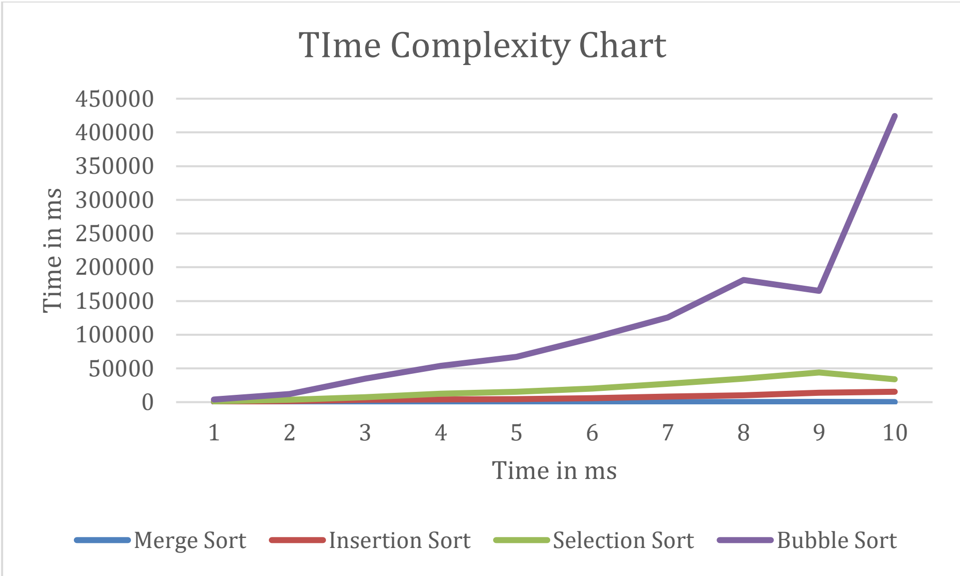
```
----------WRITING TO FILES----------

Finished writing to file numbers-1.txt
Finished writing to file numbers-2.txt
Finished writing to file numbers-3.txt
Finished writing to file numbers-4.txt
Finished writing to file numbers-5.txt
Finished writing to file numbers-6.txt
Finished writing to file numbers-7.txt
Finished writing to file numbers-8.txt
Finished writing to file numbers-9.txt
Finished writing to file numbers-10.txt
Finished writing to file numbers-11.txt
Finished writing to file numbers-12.txt
Finished writing to file numbers-13.txt
Finished writing to file numbers-14.txt
Finished writing to file numbers-15.txt
```

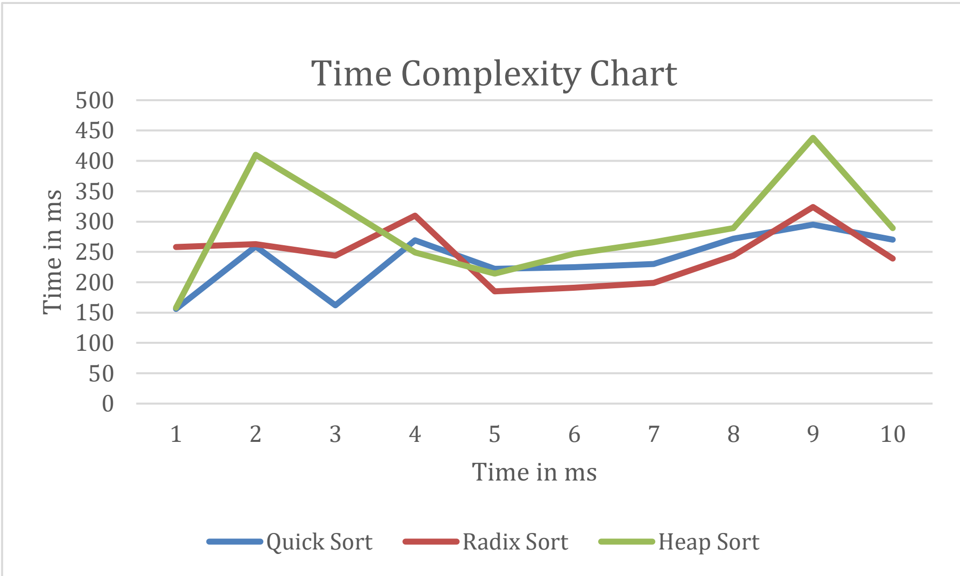
```
----------SORTING----------

1.Manual Sorting
2.Automatic Sorting      :1

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


Enter the sorting method: 4

|      Sort-Method      |   Time Taken        |       File Name
|       Quick Sort      |  81.0 ms            |       numbers-1.txt
|       Quick Sort      |  26.0 ms            |       numbers-2.txt
|       Quick Sort      |  20.0 ms            |       numbers-3.txt
|       Quick Sort      |  59.0 ms            |       numbers-4.txt
|       Quick Sort      |  39.0 ms            |       numbers-5.txt
|       Quick Sort      |  35.0 ms            |       numbers-6.txt
|       Quick Sort      |  27.0 ms            |       numbers-7.txt
|       Quick Sort      |  21.0 ms            |       numbers-8.txt
|       Quick Sort      |  20.0 ms            |       numbers-9.txt
|       Quick Sort      |  60.0 ms            |       numbers-10.txt
|       Quick Sort      |  25.0 ms            |       numbers-11.txt
|       Quick Sort      |  43.0 ms            |       numbers-12.txt
|       Quick Sort      |  18.0 ms            |       numbers-13.txt
|       Quick Sort      |  33.0 ms            |       numbers-14.txt
|       Quick Sort      |  23.0 ms            |       numbers-15.txt
Total time taken for all sorts = 530.0ms
```

```
----------MERGING----------


Merge Sort:
Merging Files 1-2 : 0.0 ms
Merging Files 1-3 : 0.0 ms
Merging Files 1-4 : 0.0 ms
Merging Files 1-5 : 0.0 ms
Merging Files 1-6 : 0.0 ms
Merging Files 1-7 : 0.0 ms
Merging Files 1-8 : 0.0 ms
Merging Files 1-9 : 0.0 ms
Merging Files 1-10 : 0.0 ms
Merging Files 1-11 : 0.0 ms
Merging Files 1-12 : 0.0 ms
Merging Files 1-13 : 0.0 ms
Merging Files 1-14 : 0.0 ms
Merging Files 1-15 : 0.0 ms

Total Time taken for Merging Files = 0ms
----------TOTAL TIME----------
TOTAL TIME FOR ALL THE SORTING IS : 14047.46875ms
```

Total time is 14s which is greater than 13 files

TIME COMPLEXITY CHART
Following algorithms were tested for time
1) Merge Sort
2) Quick Sort
3) Heap Sort
4) Selection Sort
5) Radix Sort
6) Bubble Sort
7) Insertion Sort





Bubble Sort	3893	12261	34878	53790	67320	95164	125730	181319	164909	424319
Insertion Sort	617	1592	3382	4153	4426	5931	8026	10331	14013	15391
Selection Sort	1170	3435	7278	12697	15413	20116	27330	34747	43993	33804
Quick Sort	156	259	162	269	222	225	230	272	295	270
Heap Sort	158	410	331	249	214	247	266	289	438	289
Radix Sort	258	263	244	310	185	191	199	244	324	239
Merge Sort	127	331	272	371	256	264	385	297	420	325

Bharatiya Vidya Bhavan's

Sardar Patel Institute of Technology

Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai-400058-India
(Autonomous College Affiliated to University of Mumbai)

Computer Engineering Department & Information Technology Engineering Department

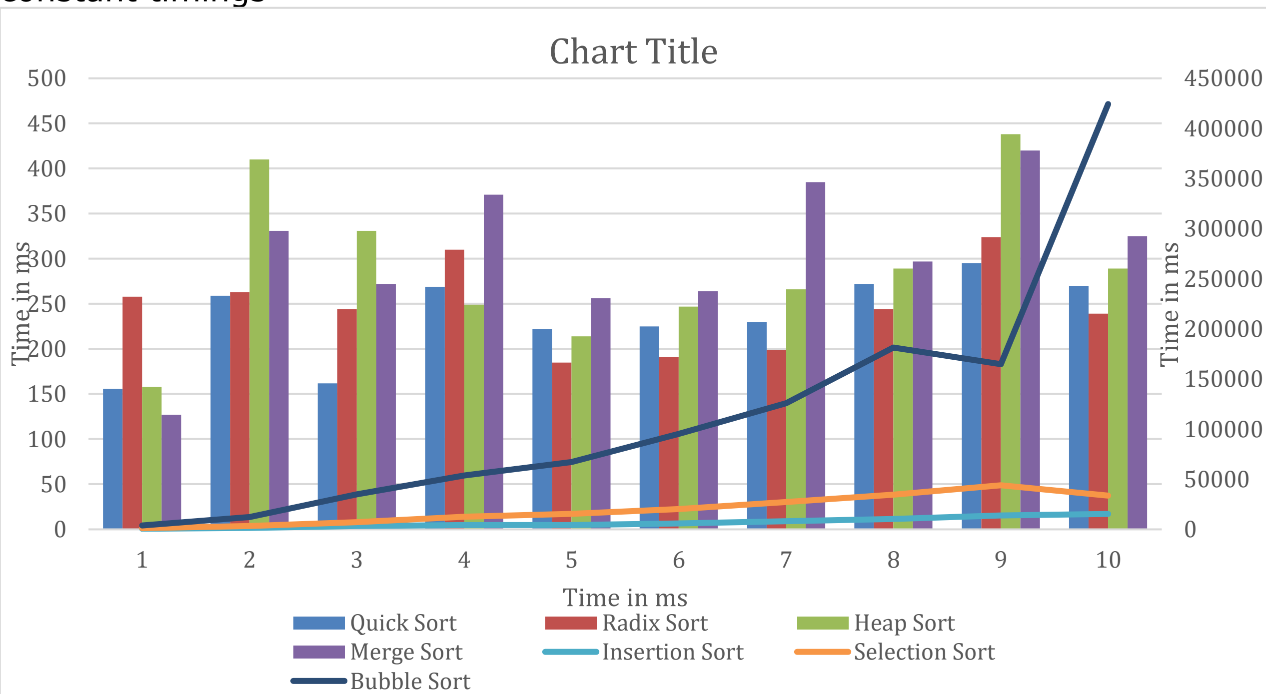
Academic Year: 2021-2022

Class: S.Y.B.Tech **Sem.:** 4 **Course:** DAA

RESULT:

	1	2	3	4	5	6	7	8	9	10	Average
Quick Sort	156	259	162	269	222	225	230	272	295	270	236
Radix Sort	258	263	244	310	185	191	199	244	324	239	245.7
Heap Sort	158	410	331	249	214	247	266	289	438	289	289.1
Merge Sort	127	331	272	371	256	264	385	297	420	325	304.8
Insertion Sort	617	1592	3382	4153	4426	5931	8026	10331	14013	15391	6786.2
Selection Sort	1170	3435	7278	12697	15413	20116	27330	34747	43993	33804	19998.3
Bubble Sort	3893	12261	34878	53790	67320	95164	125730	181319	164909	424319	116358.3

From the above image quick sort takes the least amount of time and bubble sort takes the most amount of time
Radix, Merge and Heap sort were quite close to the quick sort
Comparing the other timings Insertion, Selection and Bubble sort took a lot of time as the time complexity increases with number of elements rapidly whereas the other having lesser time complexity where getting almost constant timings



Here is the combined graph of all the time complexity of the algoritms