Experiment No. : 1

Title: Basic Sorting algorithm and its analysis

Batch: A2 **Roll No.: 16010421073** **Experiment No.: 1**

Aim: To implement and analyse time complexity of insertion sort & Heap sort.

Explanation and Working of insertion sort & Heap sort:

1) 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.

Characteristics of Insertion Sort:

- This algorithm is one of the simplest algorithm with simple implementation
- Basically, Insertion sort is efficient for small data values
- Insertion sort is adaptive in nature, i.e. it is appropriate for data sets which are already partially sorted.

Working of Insertion Sort algorithm:

Consider an example: arr[]: {12, 11, 13, 5, 6}

12 11 13 5 6

First Pass:

- Initially, the first two elements of the array are compared in insertion sort.

12 **11** 13 5 6

- Here, 12 is greater than 11 hence they are not in the ascending order and 12 is not at its correct position. Thus, swap 11 and 12.
- So, for now 11 is stored in a sorted sub-array.

11 **12** 13 5 6

Second Pass:

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.

Characteristics of Insertion Sort:

- This algorithm is one of the simplest algorithm with simple implementation
- Basically, Insertion sort is efficient for small data values
- Insertion sort is adaptive in nature, i.e. it is appropriate for data sets which are already partially sorted.

Working of Insertion Sort algorithm:

Consider an example: arr[]: {12, 11, 13, 5, 6}

12 11 13 5 6

First Pass:

- Initially, the first two elements of the array are compared in insertion sort.

12 **11** 13 5 6

- Here, 12 is greater than 11 hence they are not in the ascending order and 12 is not at its correct position. Thus, swap 11 and 12.
- So, for now 11 is stored in a sorted sub-array.

11 **12** 13 5 6

Second Pass:

- Now, move to the next two elements and compare them

11 **12** **13** 5 6

- Here, 13 is greater than 12, thus both elements seems to be in ascending order, hence, no swapping will occur. 12 also stored in a sorted sub-array along with 11

Third Pass:

- Now, two elements are present in the sorted sub-array which are **11** and **12**
- Moving forward to the next two elements which are 13 and 5

11 12 **13** **5** 6

- Both 5 and 13 are not present at their correct place so swap them

11 12 **5** **13** 6

- After swapping, elements 12 and 5 are not sorted, thus swap again

11 **5** **12** 13 6

- Here, again 11 and 5 are not sorted, hence swap again

5 **11** 12 13 6

- here, it is at its correct position

Fourth Pass:

- Now, the elements which are present in the sorted sub-array are **5, 11** and **12**
- Moving to the next two elements 13 and 6

5 11 12 **13** **6**

- Clearly, they are not sorted, thus perform swap between both

5 11 12 **6** **13**

- Now, 6 is smaller than 12, hence, swap again

5 11 **6** **12** 13

- Here, also swapping makes 11 and 6 unsorted hence, swap again

5 **6** **11** 12 13

- Finally, the array is completely sorted.

2) Heap sort:

What is Heap Sort?

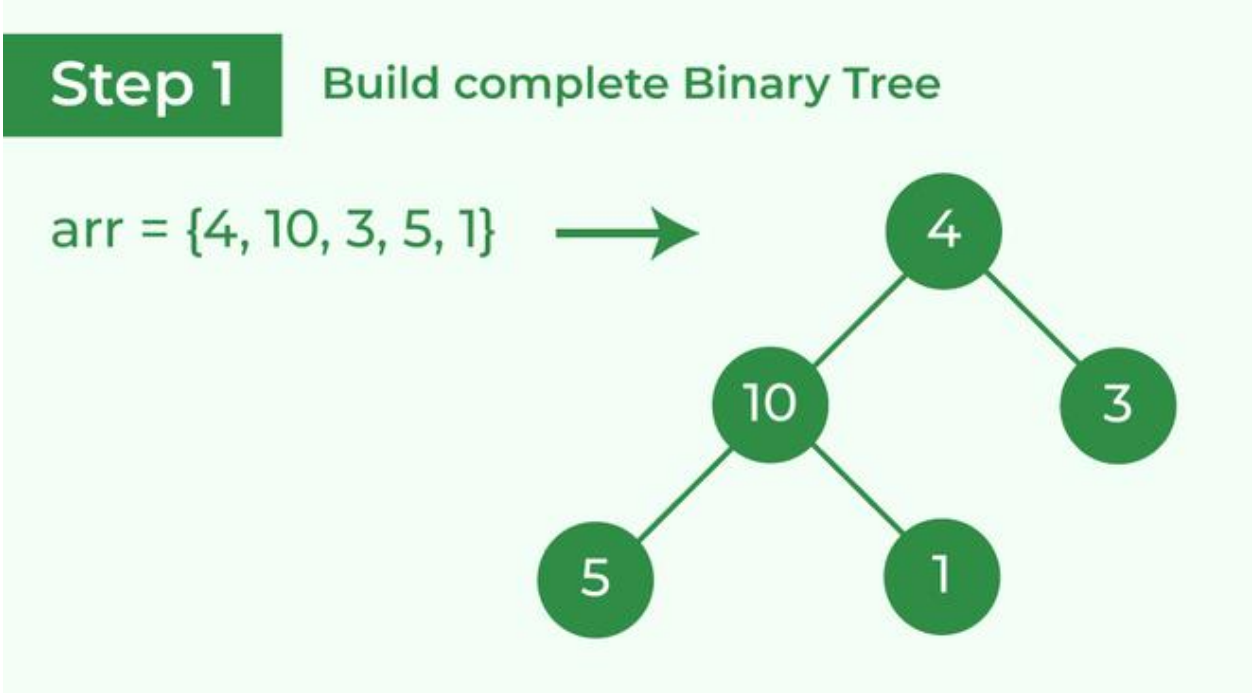
Heap sort is a comparison-based sorting technique based on Binary heap data structure. It is similar to the selection sort where we first find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements.

- Heap sort is an in-place algorithm.
- Its typical implementation is not stable, but can be made stable.
- Typically 2-3 times slower than well-implemented QuickSort. The reason for slowness is a lack of locality of reference.

Working of Heap Sort algorithm:

To understand heap sort more clearly, let's take an unsorted array and try to sort it using heap sort.
Consider the array: arr[] = {4, 10, 3, 5, 1}.

Build Complete Binary Tree: Build a complete binary tree from the array.



Build complete binary tree from the array

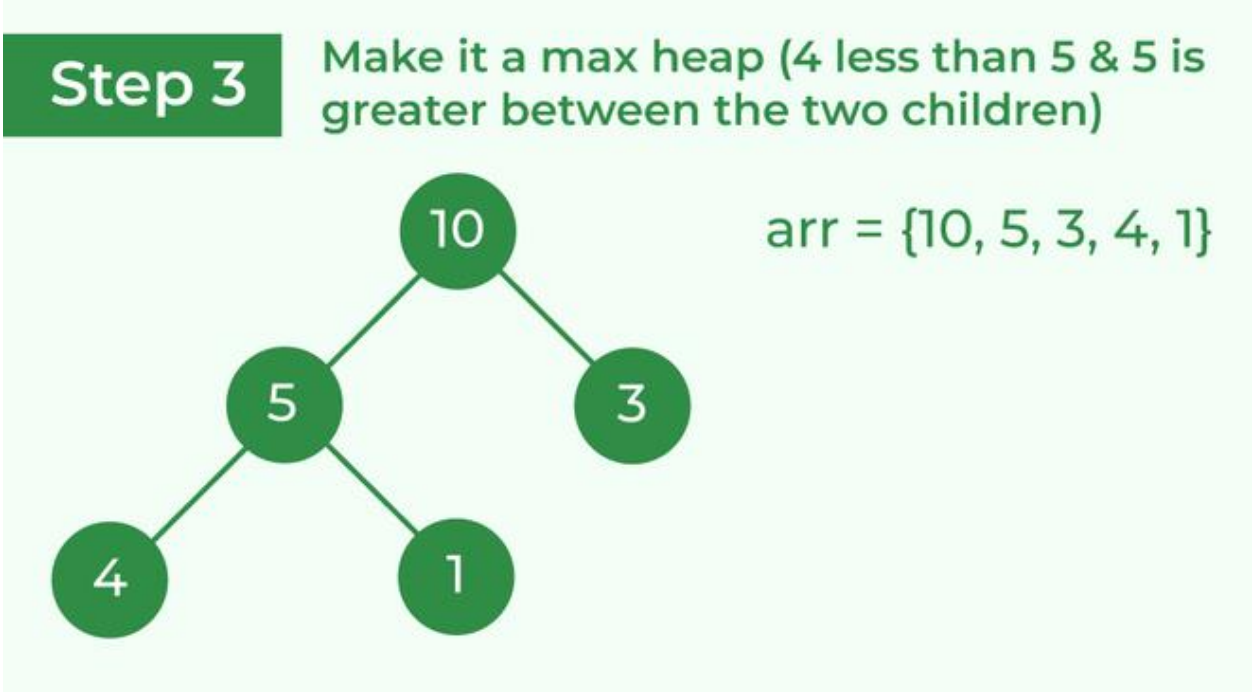
Transform into max heap: After that, the task is to construct a tree from that unsorted array and try to convert it into max heap.

- To transform a heap into a max-heap, the parent node should always be greater than or equal to the child nodes

- Here, in this example, as the parent node **4** is smaller than the child node **10,** thus, swap them to build a max-heap.

Transform it into a max heap image widget

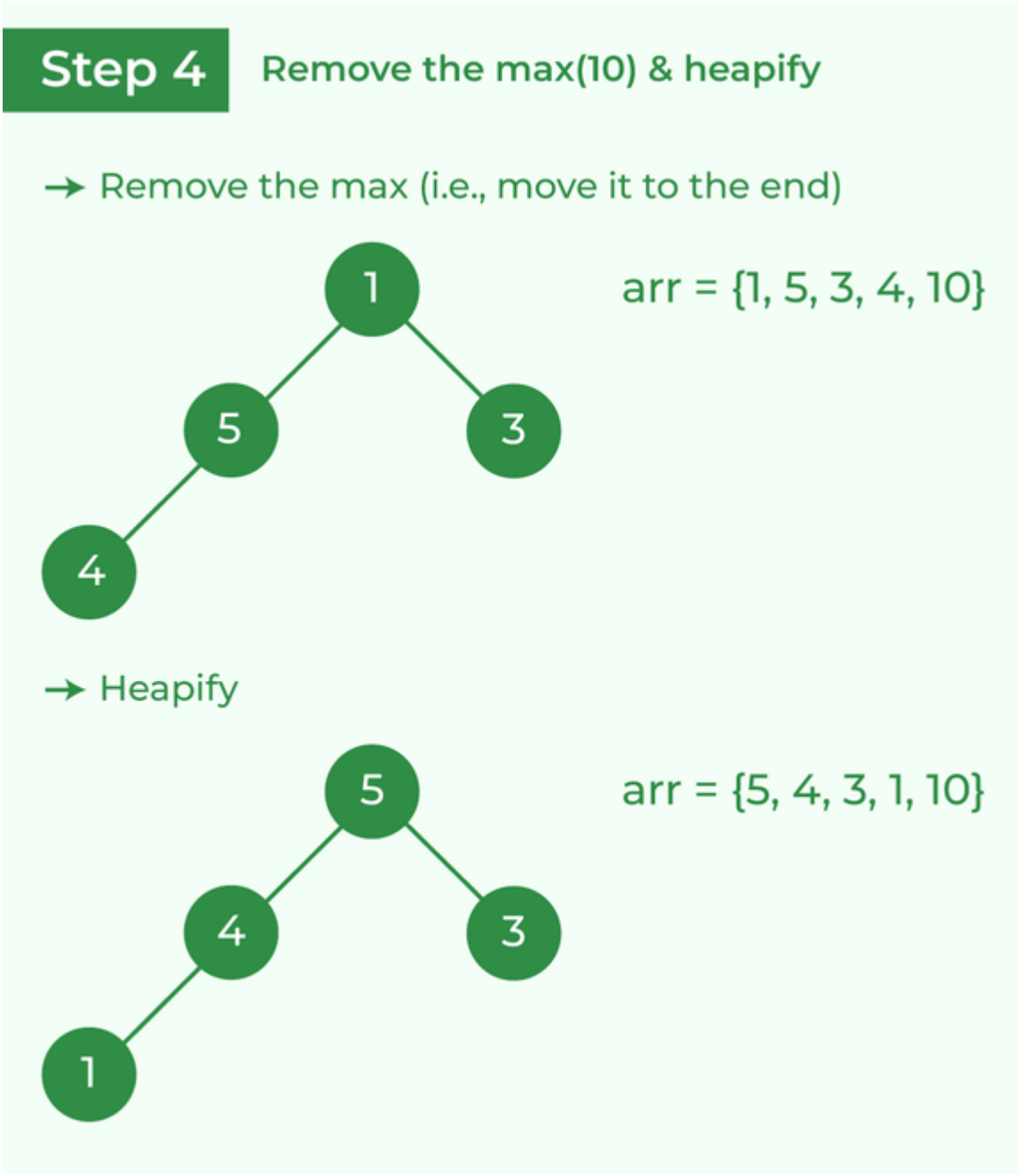
- Now, as seen, **4** as a parent is smaller than the child **5**, thus swap both of these again and the resulted heap and array should be like this:



Make the tree a max heap

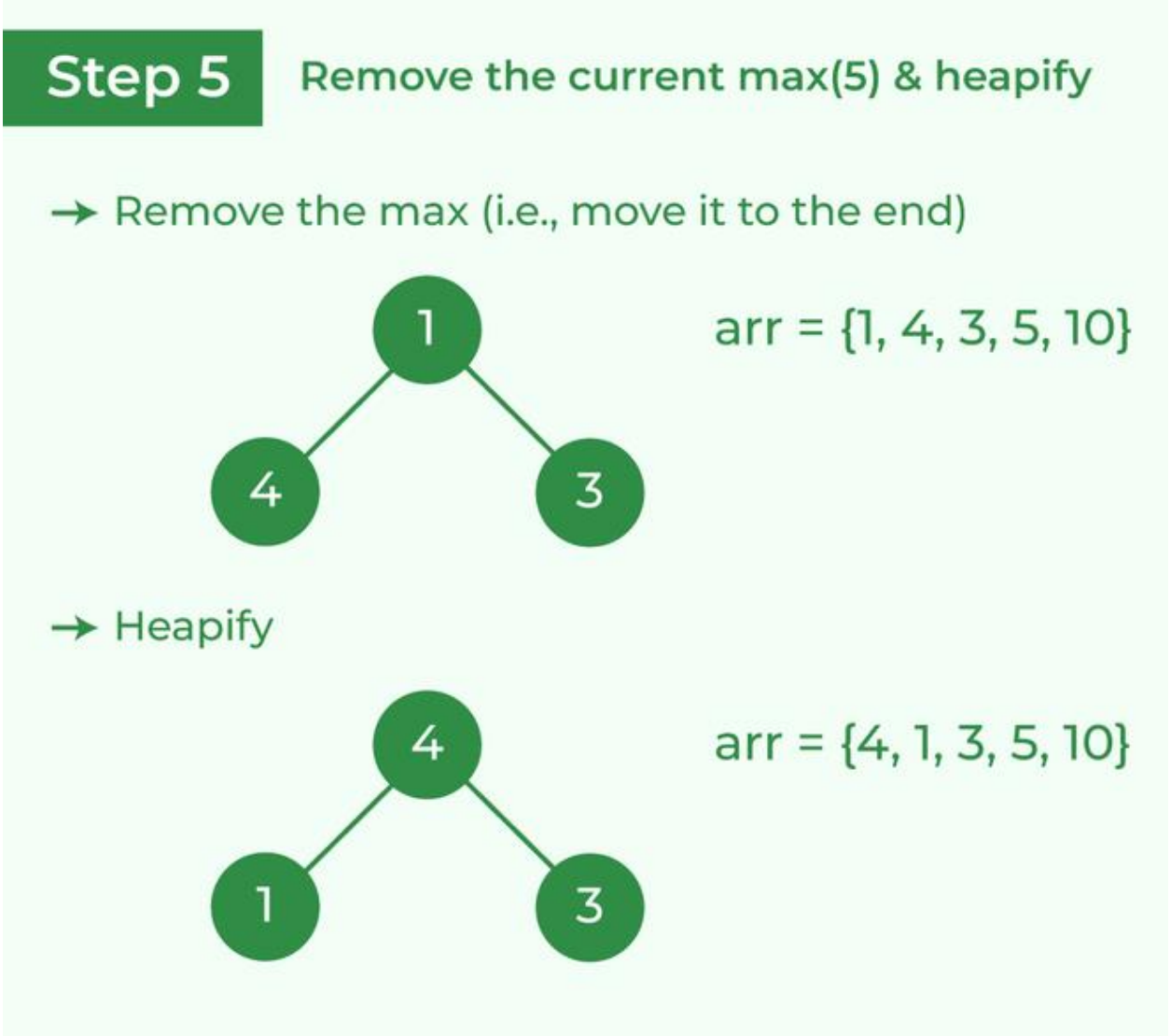
Perform heap sort: Remove the maximum element in each step (i.e., move it to the end position and remove that) and then consider the remaining elements and transform it into a max heap.

- Delete the root element **(10)** from the max heap. In order to delete this node, try to swap it with the last node, i.e. **(1).** After removing the root element, again heapify it to convert it into max heap.
    - Resulted heap and array should look like this:



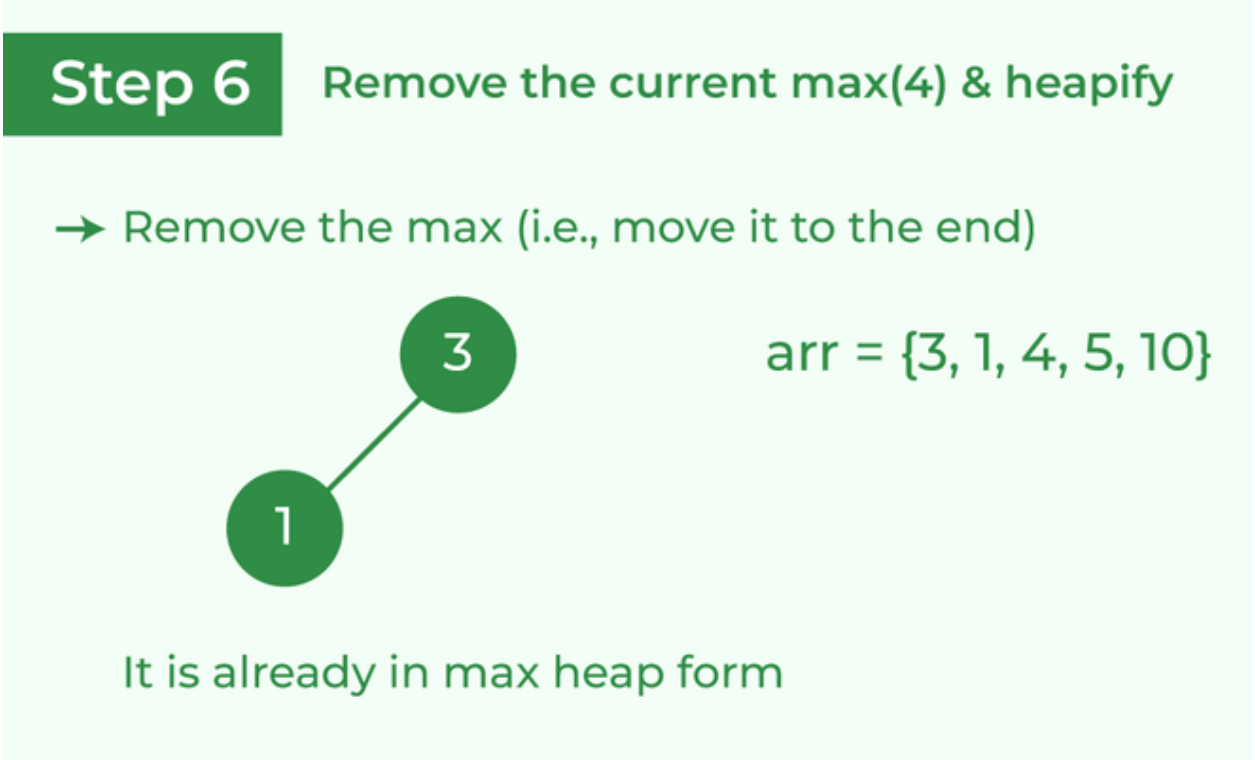
Remove 10 and perform heapify

- Repeat the above steps and it will look like the following:



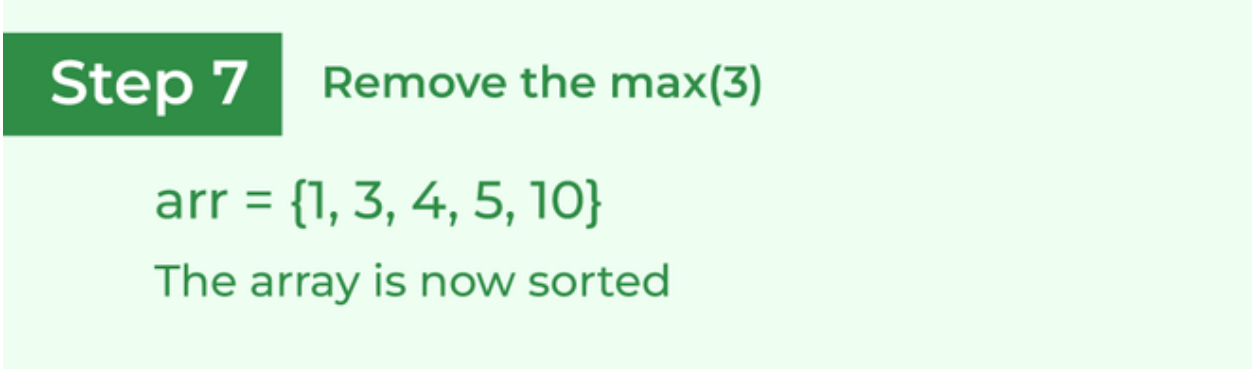
Remove 5 and perform heapify

- Now remove the root (i.e. 3) again and perform heapify.



Remove 4 and perform heapify

- Now when the root is removed once again it is sorted. and the sorted array will be like **arr[] = {1, 3, 4, 5, 10}**.



Algorithm of insertion sort & Heap sort:

1) Insertion sort:

```
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
```

2) Heap sort:

Derivation of Analysis insertion sort & Heap sort:

1)Insertion sort:

COST OF LINE	NO. OF TIMES IT IS RUN
C_1	n
C_2	n - 1
C_3	n - 1
C_4	$\Sigma^{n-1}_{j=1}(t_j)$
C_5	$\Sigma^{n-1}_{j=1}(t_j - 1)$
C_6	$\Sigma^{n-1}_{j=1}(t_j - 1)$
C_8	n - 1

Then Total Running Time of Insertion sort (T(n)) = $C_1 * n + (C_2 + C_3) * (n - 1) + C_4 * \Sigma^{n-1}_{j=1}(t_j) + (C_5 + C_6) * \Sigma^{n-1}_{j=1}(t_j) + C_8 * (n - 1)$

Worst Case Analysis:

In Worst Case i.e., when the array is reversly sorted (in descending order), $\mathbf{t_j = j}$
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 $\mathbf{n^2}$ and gives $T(n) = C * (n^2) \text{ or } O(n^2)$

Best Case Analysis:

In Best Case i.e., when the array is already sorted, $\mathbf{t_j = 1}$
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 $T(n) = C * (n) \text{ or } O(n)$

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 $\mathbf{n^2}$ and gives $T(n) = C * (n^2) \text{ or } O(n^2)$

2) <u>Heap sort:</u>

Worst Case Analysis

O(n*log(n))

Best Case Analysis

O(n*log(n))

Average Case Analysis
O(n*log(n))

Program(s) of insertion sort & Heap sort:

1) Insertion sort:

```c
#include <stdio.h>
#include<stdlib.h>


int insertion_sort(int arr[],int n);
void main();

int insertion_sort(int arr[],int n)
{
    int i, key , j,swap;
    swap=0;
    for(i=0;i<n;i++){
        key= arr[i];
        j=i-1;

        while(j>= 0 && arr[j]>key)
        {
            arr[j+1]=arr[j];
            j=j-1;
            swap++;

        }
        arr[j+1]=key;
    }
    return swap;
}
void main()
{
    int n,i;
    printf("Enter the size of the array: ");
    scanf("%d",&n);
    int avg[n],best[n],worst[n];
    printf("\n");
    for(i=0;i<n;i++)
    {

        best[i]=i;
        worst[i]=n-i;
        avg[i]=rand();
    }
```

```
    int best_swap = insertion_sort(best,n);
    int worst_swap = insertion_sort(worst,n);
    int avg_swap = insertion_sort(avg,n);

    printf("Swaps in best case scenario: %d \n",best_swap);
    printf("Swaps in worst case scenario: %d \n",worst_swap);
    printf("Swaps in avg case scenario: %d \n",avg_swap);
    printf("Time complexity of the best case : %d \n",n);
    printf("Time complexity of the worst case : %d \n",n*n);
    printf("Time complexity of the average case : %d \n",n*n);
}
```

2) Heap sort:

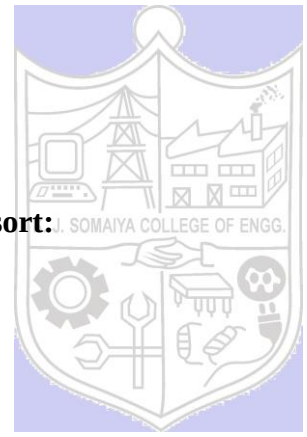
```
#include <sl'dio.h>


void buildMaxHeap(inl' arr[], inl' n)

{

        for (inl' i = 1; i < n; i++)

        {

        // if child is bigger l'han parenl'if
        (arr[i] > arr[(i - 1) / 2])

        {

        inl' j = i;



        // swap child and parenl' unl'il

        // parenl' is smaller

        while (arr[j] > arr[(j - 1) / 2])

        {

        inl' l'emp=arr[j];
        arr[j]=arr[(j-1)/2];

        arr[(j-1)/2]=l'emp; j
        = (j - 1) / 2;

        }

        }

        }

}
```

```
void heapSorl'(inl' arr[], inl' n)
{  inl' c=0;
        buildMaxHeap(arr, n);

        for (inl' i = n - 1; i > 0; i--)
        {
        // swap value of firsl' indexed
        // wil'h lasl' indexed
```

```
        inl' l'emp=arr[0];
        arr[0]=arr[i];
        arr[i]=l'emp;


        // mainl'aining heap properl'y

          // afl'er each swappinginl' j
        = 0, index;


        do

        {

        index = (2 * j + 1);



        if (arr[index] < arr[index + 1] &&

                                index < (i - 1))

        index++;


        if (arr[j] < arr[index] && index < i)

        {

        inl' l'em1=arr[j];
        arr[j]=arr[index];
        arr[index]=l'em1;
        c++;

        }



        j = index;


        } while (index < i);

        }

        prinl'f("No. of il'eral'ions:%d\n",c);

}


void prinl'Array(inl' arr[], inl' N)

{
```

```
    for (inl' i = 0; i < N; i++)
    prinl'f("%d ", arr[i]);
    prinl'f("\n");

}
```

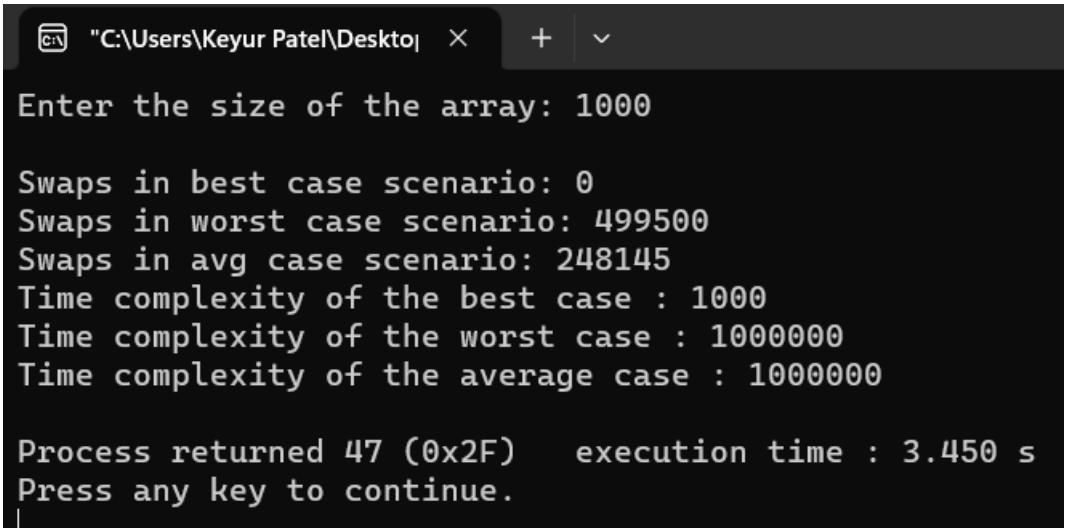
```
// Driver Code l'o l'esl' aboveinl'
main()

{

        inl' my_arr[100];
        srand(1);


        inl' i;

        for (i = 0; i < 100; i++) {
        my_arr[i] = rand();

  }

        inl' n = sizeof(my_arr) / sizeof(my_arr[0]);


        heapSorl'(my_arr, n);


        rel'urn 0;

}
```



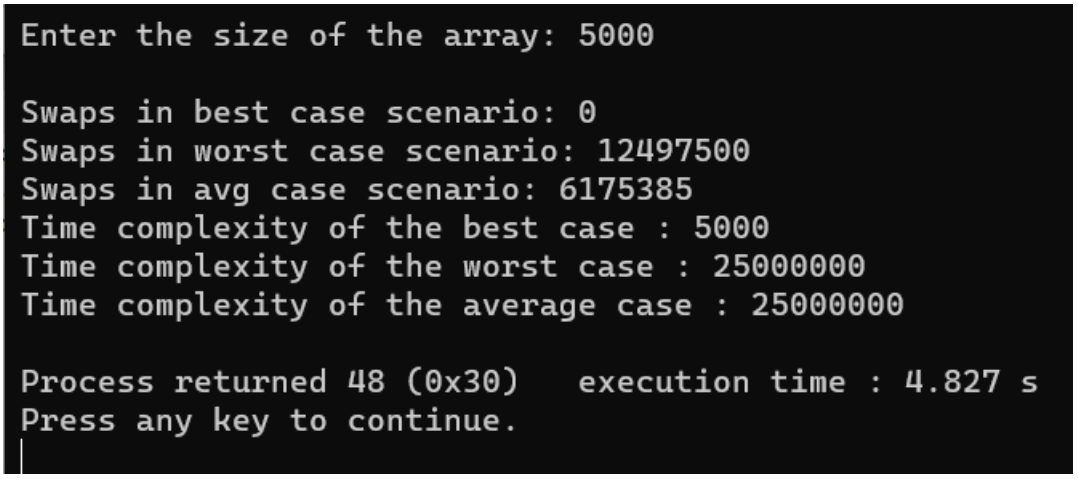
Output(o) of insertion sort & Heap sort:

1) Insertion sort:

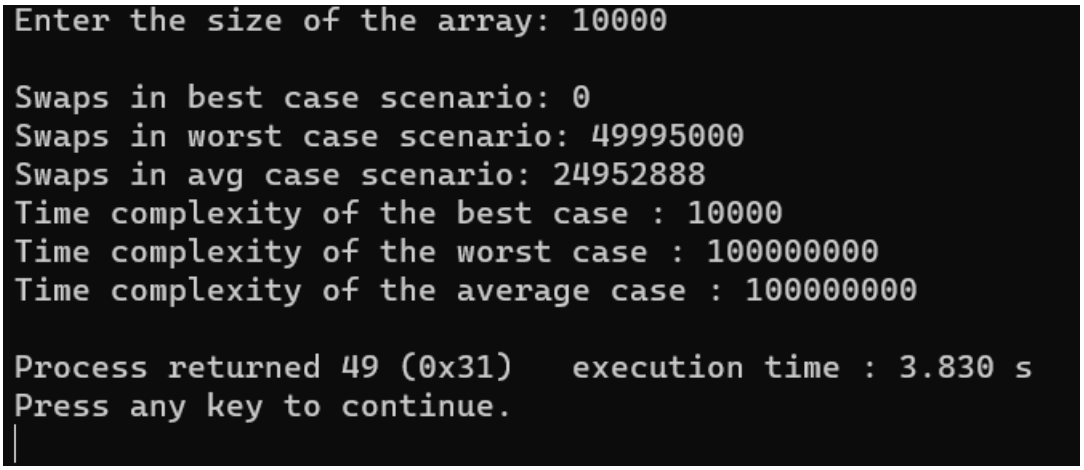
Case of 1000 enteries:



Case of 5000 enteries:



Case of 10000 enteries:



Case of 15400 enteries:

```
Enter the size of the array: 15400

Swaps in best case scenario: 0
Swaps in worst case scenario: 118572300
Swaps in avg case scenario: 59118982
Time complexity of the best case : 15400
Time complexity of the worst case : 237160000
Time complexity of the average case : 237160000

Process returned 49 (0x31)   execution time : 6.058 s
Press any key to continue.
```

2) Heap sort:

```
Enter the size of the array: 1000

Swaps in best case scenario: 1000
Swaps in worst case scenario: 1000
Swaps in avg case scenario: 1000
Time complexity of the best case : 1509334946
Time complexity of the worst case : 1509334946
Time complexity of the average case : 1509334946

Process returned 50 (0x32)   execution time : 2.468 s
Press any key to continue.
```

```
No. of iterations:200
Sorted array is
35005211 42999170 84353895 135497281 137806862 149798315 184803526
    233665123 278722862 294702567 304089172 336465782 356426808 412776091
    424238335 468703135 491705403 511702305 521595368 572660336 596516649
    608413784 610515434 628175011 635723058 709393584 719885386 749241873
    752392754 756898537 760313750 783368690 805750846 846930886 855636226
    859484421 861021530 939819582 943947739 945117276 982906996
    1025202362 1059961393 1100661313 1101513929 1102520059 1125898167
    1129566413 1131176229 1141616124 1159126505 1189641421 1264095060
    1303455736 1315634022 1350490027 1365180540 1369133069 1374344043
    1411549676 1424268980 1433925857 1469348094 1474612399 1477171087
    1540383426 1548233367 1585990364 1632621729 1649760492 1653377373
    1656478042 1681692777 1714636915 1726956429 1734575198 1749698586
    1780695788 1801979802 1804289383 1827336327 1843993368 1889947178
    1911759956 1914544919 1918502651 1937477084 1956297539 1957747793
    1967513926 1973594324 1984210012 1998898814 2001100545 2038664370
    2044897763 2053999932 2084420925 2089018456 2145174067
```

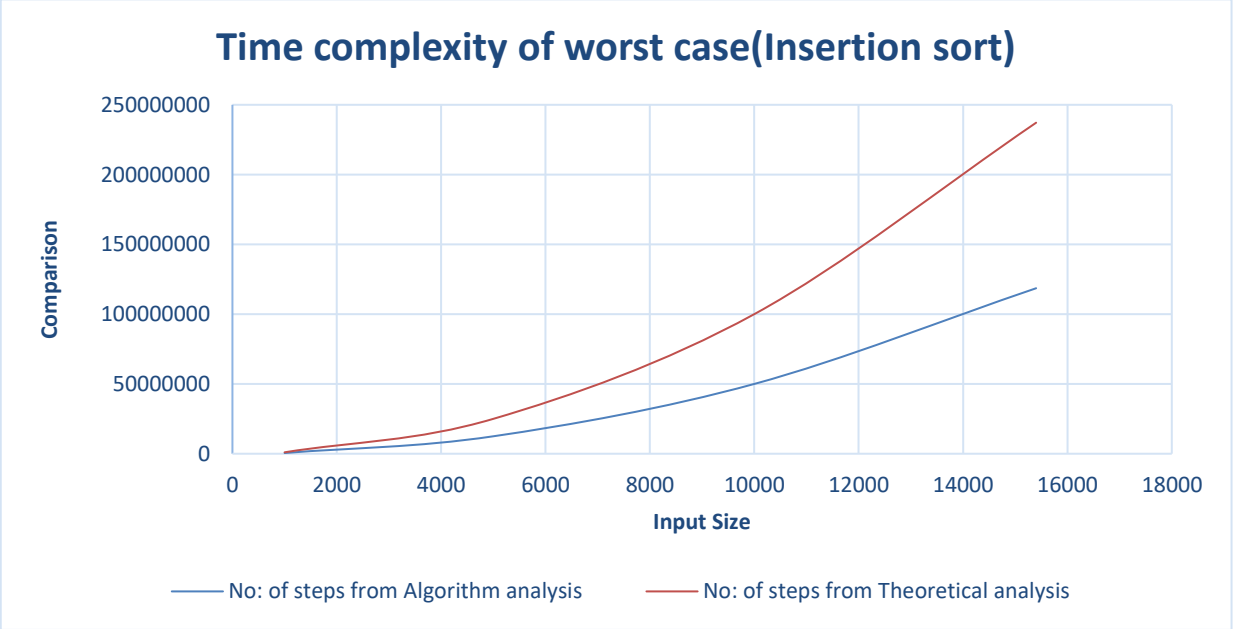
Results:

Time Complexity of Insertion sort:

Worst Case Analysis:

Sr. No.	Input size	No: of steps from Algorithm analysis	No: of steps from Theoretical analysis
1	1000	499500	1000000
2	5000	12497500	25000000
3	10000	49995000	100000000
4	15400	118572300	237,160,000

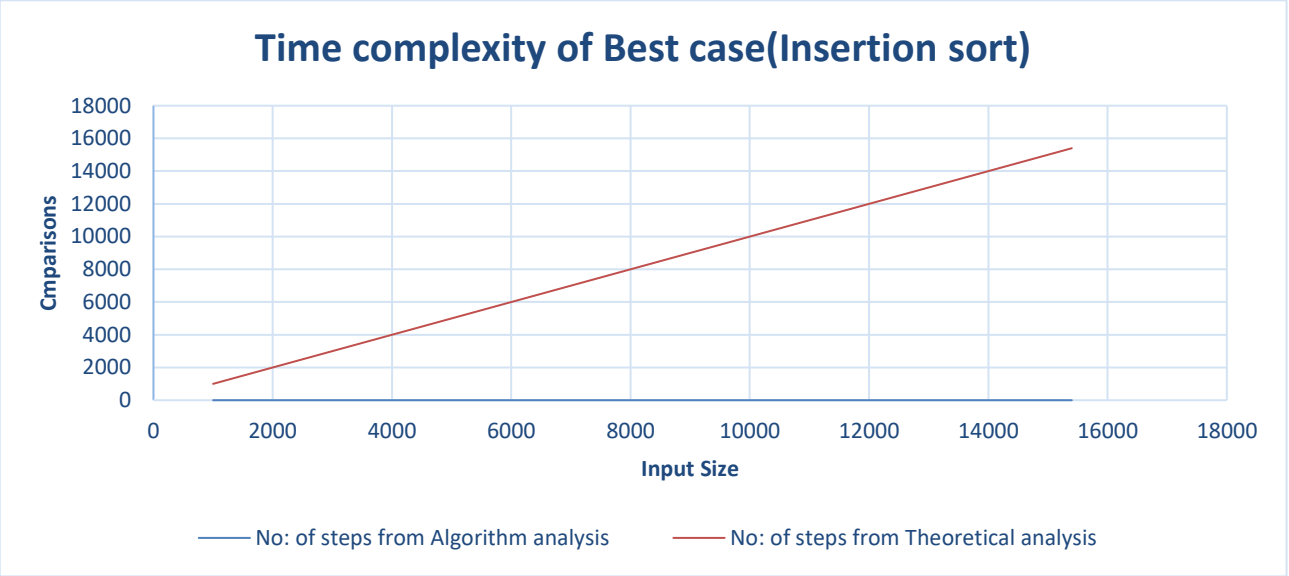
GRAPH:



Best Case Analysis:

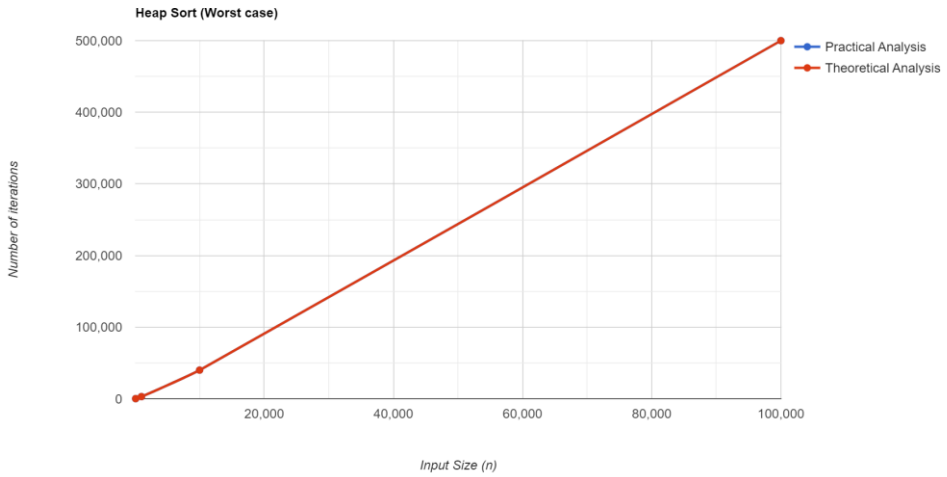
Sr. No.	Input size	No: of steps from Algorithm analysis	No: of steps from Theoretical analysis
1	1000	0	1000
2	5000	0	5000
3	10000	0	10000
4	15400	0	15400

GRAPH



Time Complexity of Heap sort:
Worst Case Analysis:

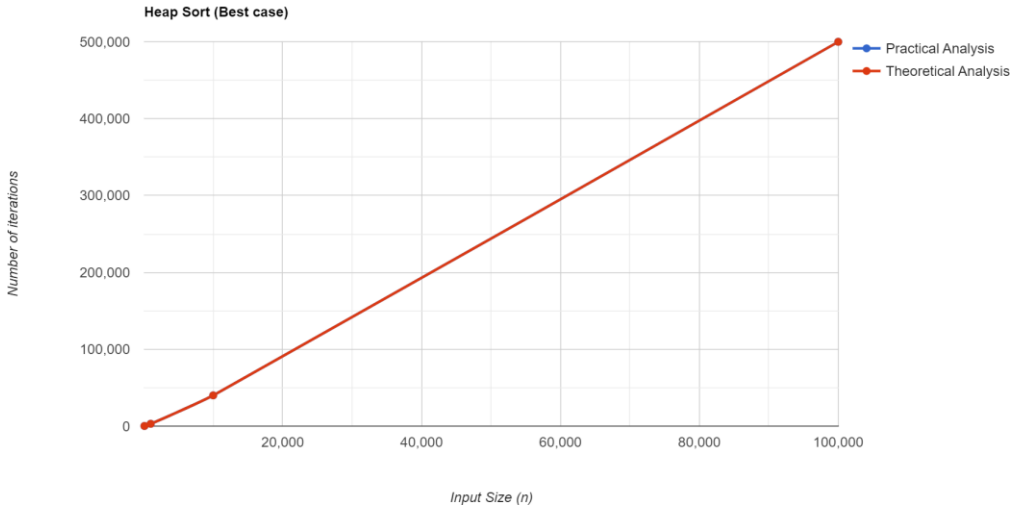
Sr. No.	Input size	No: of steps from Algorithm analysis	No: of steps from Theoretical analysis
1.	100	200	200
2.	1000	2999	3000
3.	10000	39998	40000
4.	100000	499997	500000



GRAPH:

Best Case Analysis:

Sr. No.	Input size	No: of steps from Algorithm analysis	No: of steps from Theoretical analysis
1.	100	200	200
2.	1000	2999	3000
3.	10000	39998	40000
4.	100000	499997	500000



Conclusion: (Based on the observations):
Hence we implemented insertion and heap sort

Outcome:
CO1: Analyze time and space complexity of basic algorithms.

References:

1. Richard E. Neapolitan, " Foundation of Algorithms ", 5th Edition 2016, Jones & Bartlett Students Edition
2. Harsh Bhasin , " Algorithms : Design & Analysis", 1st Edition 2013, Oxford Higher education, India
3. T.H. Coreman ,C.E. Leiserson,R.L. Rivest, and C. Stein, " Introduction to algorithms", 3rd Edition 2009, Prentice Hall India Publication
4. Jon Kleinberg, Eva Tardos, " Algorithm Design", 10th Edition 2013, Pearson India Education Services Pvt. Ltd.