Experiment No. : 1

Title: Basic Sorting algorithm and its analysis

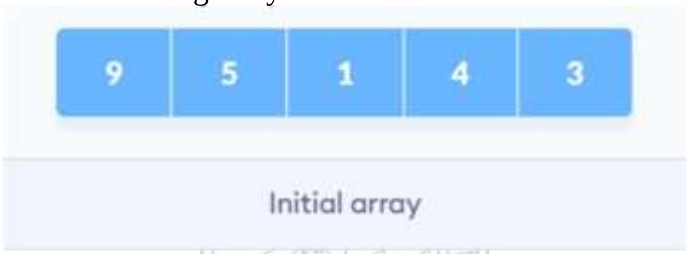
Batch: A2 **Roll No.: 16010421059** **Experiment No.: 1**

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

Explanation and Working of insertion sort & Heap sort:

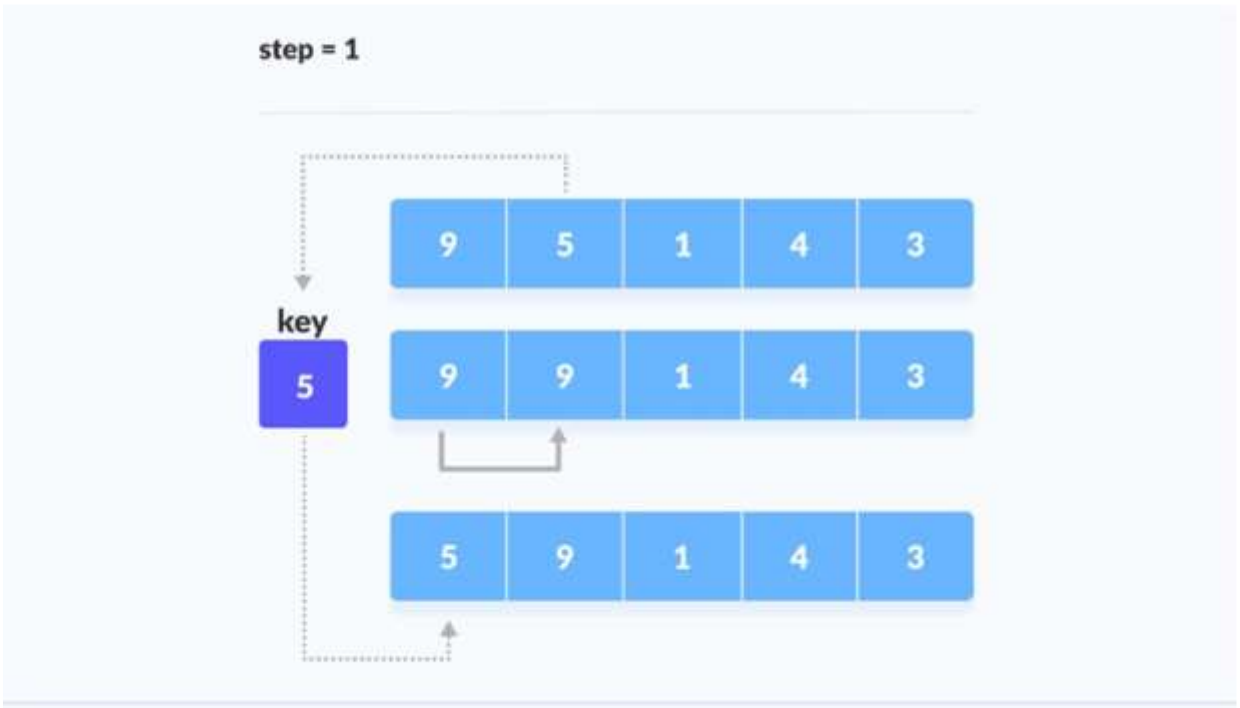
Insertion Sort:
Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.
Insertion sort works similarly as we sort cards in our hand in a card game.
We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put in their right place.
A similar approach is used by insertion sort.

Suppose we need to sort the following array.



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

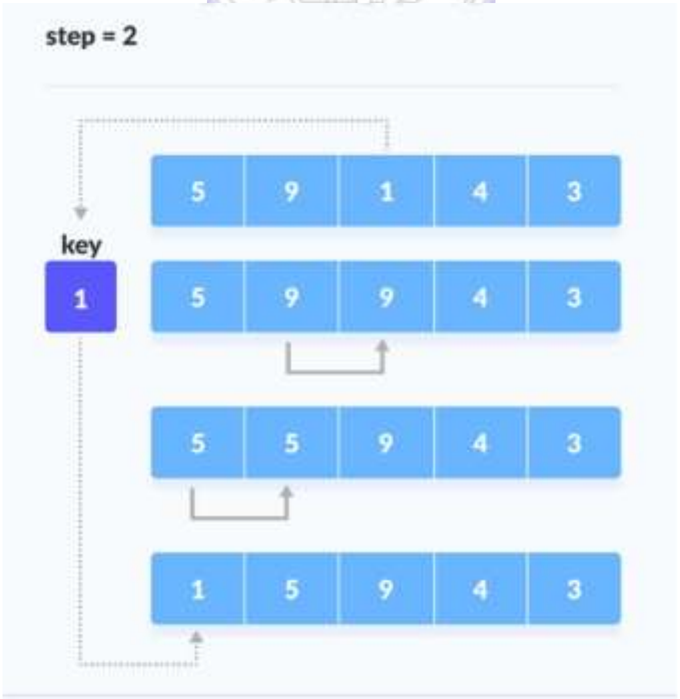
   Compare key with the first element. If the first element is greater than key, then key is placed in front of the first element.
   If the first element is greater than key, then key is placed in front of the first element.



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

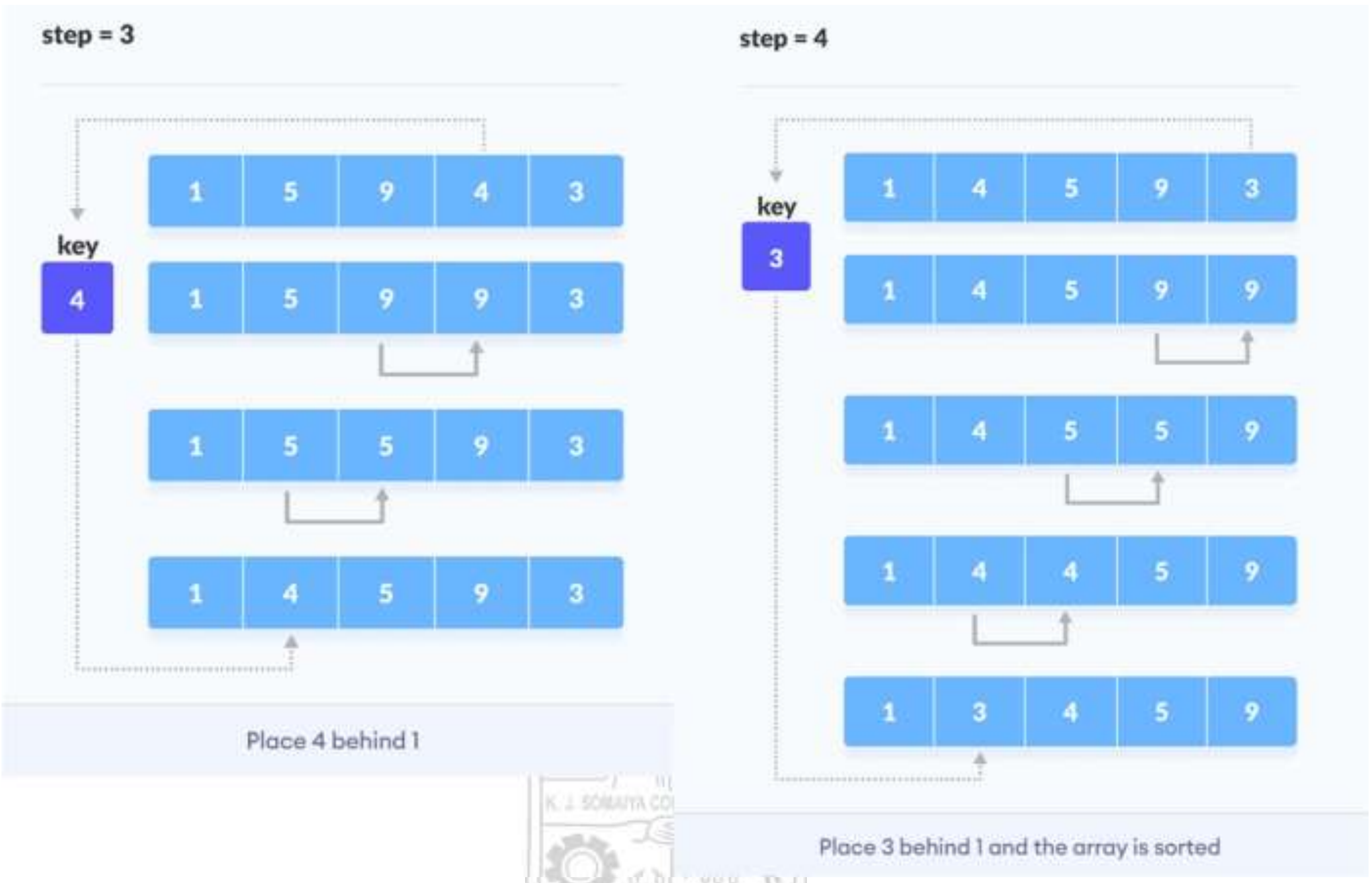
2. Now, the first two elements are sorted.

   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.



Heap Sort:

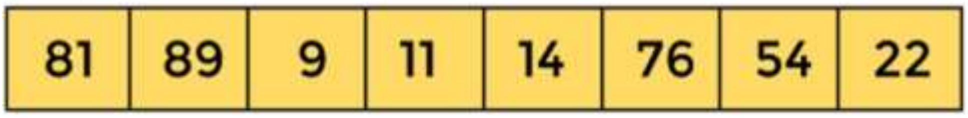
Heap sort processes the elements by creating the min-heap or max-heap using the elements of the given array. Min-heap or max-heap represents the ordering of array in which the root element represents the minimum or maximum element of the array.

Heap sort basically recursively performs two main operations -

- Build a heap H, using the elements of array.
- Repeatedly delete the root element of the heap formed in 1st phase.

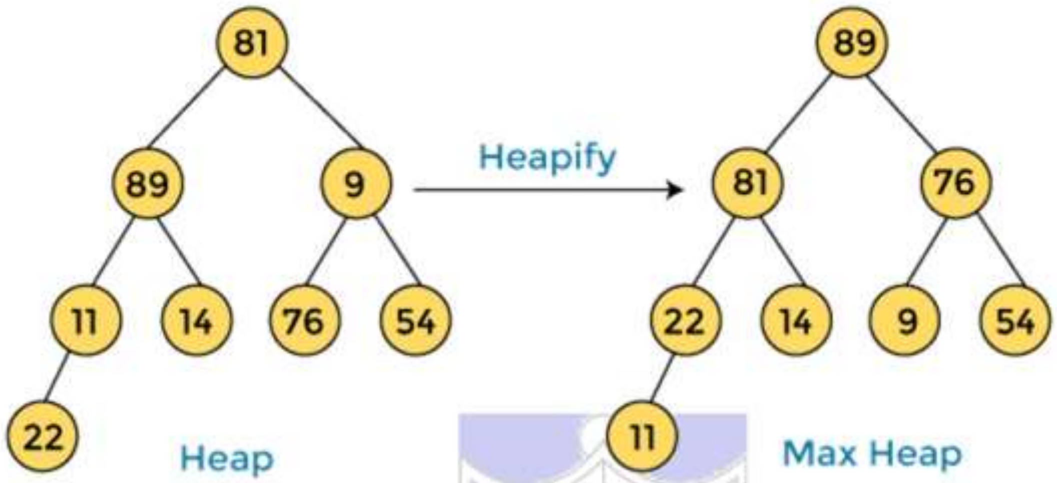
Heapsort is a popular and efficient sorting algorithm. The concept of heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list.

Example:

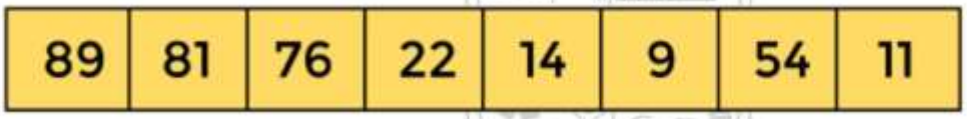


81	89	9	11	14	76	54	22

First, we have to construct a heap from the given array and convert it into max heap.

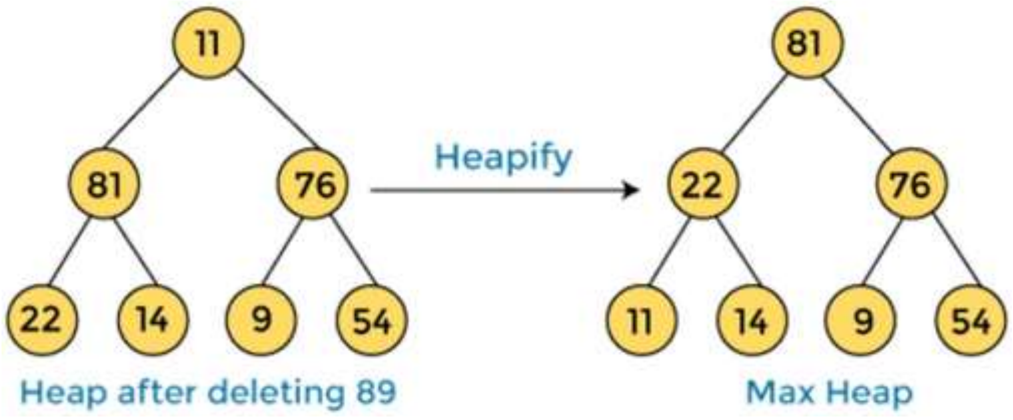


After converting the given heap into max heap, the array elements are -

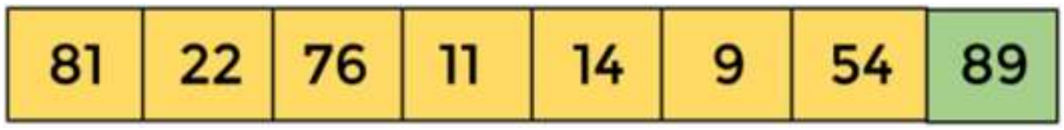


89	81	76	22	14	9	54	11

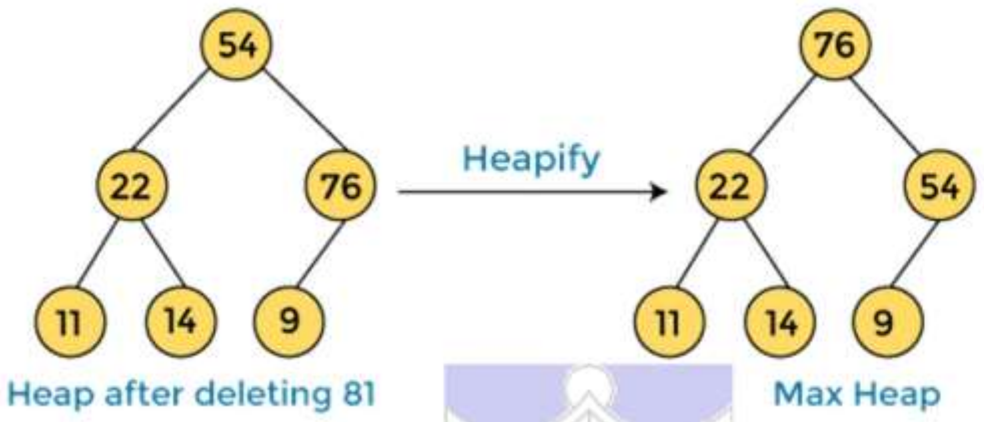
Next, we have to delete the root element **(89)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.



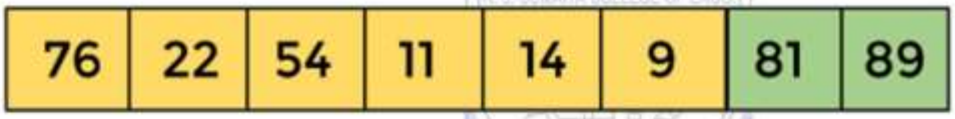
After swapping the array element **89** with **11,** and converting the heap into max-heap, the elements of array are -



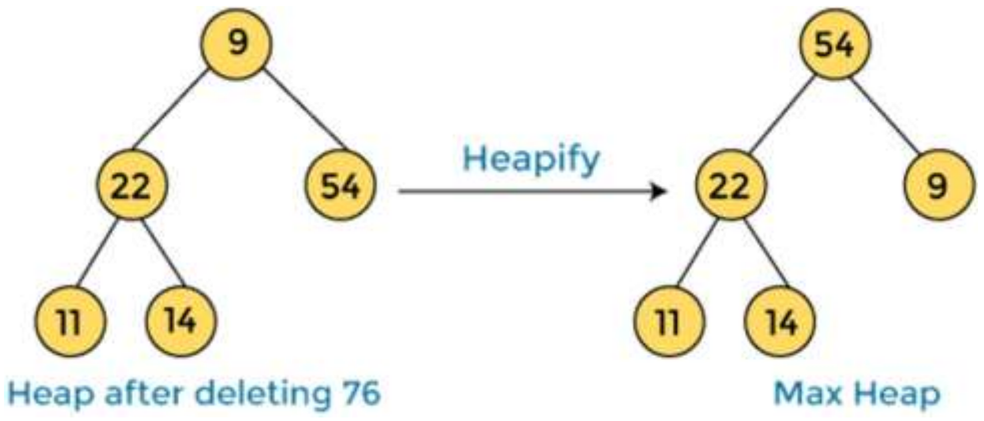
In the next step, again, we have to delete the root element **(81)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(54).** After deleting the root element, we again have to heapify it to convert it into max heap.



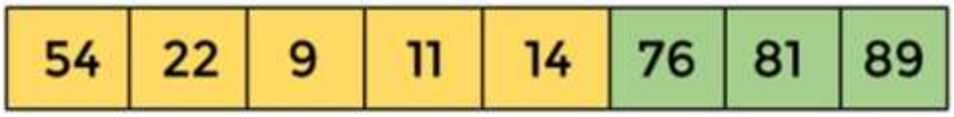
After swapping the array element **81** with **54** and converting the heap into max-heap, the elements of array are -



In the next step, we have to delete the root element **(76)** from the max heap again. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.

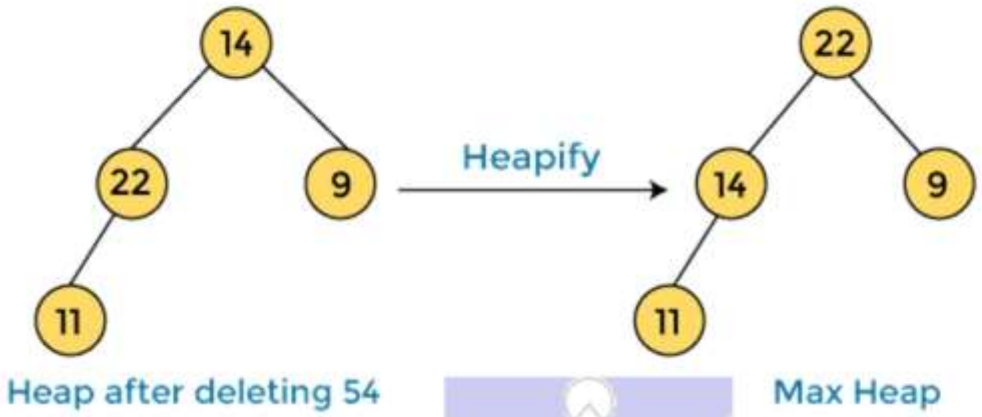


After swapping the array element **76** with **9** and converting the heap into max-heap, the elements of array are -

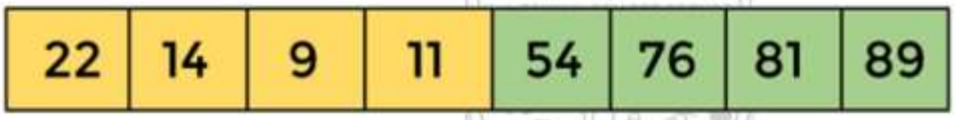


54	22	9	11	14	76	81	89

In the next step, again we have to delete the root element **(54)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(14).** After deleting the root element, we again have to heapify it to convert it into max heap.

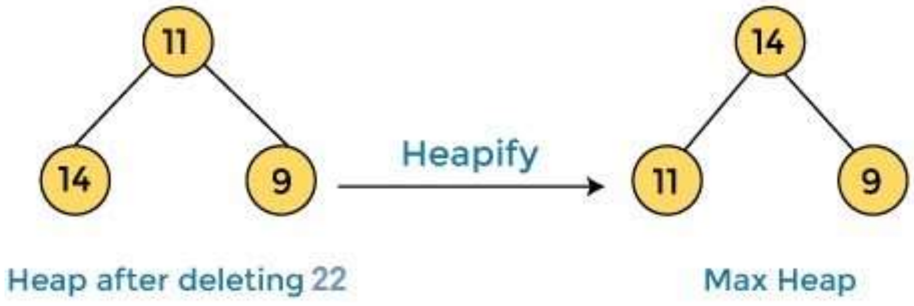


After swapping the array element **54** with **14** and converting the heap into max-heap, the elements of array are -

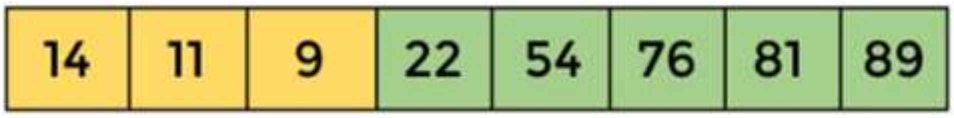


22	14	9	11	54	76	81	89

In the next step, again we have to delete the root element **(22)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.

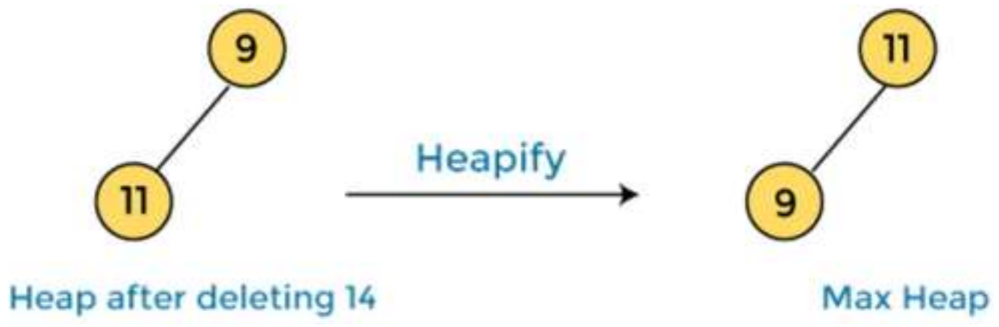


After swapping the array element **22** with **11** and converting the heap into max-heap, the elements of array are -

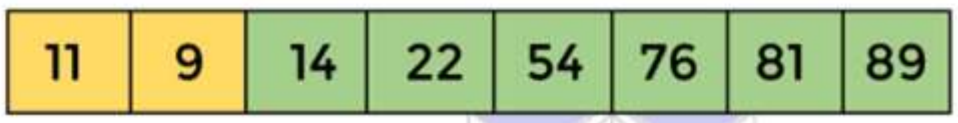


14	11	9	22	54	76	81	89

In the next step, again we have to delete the root element **(14)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.

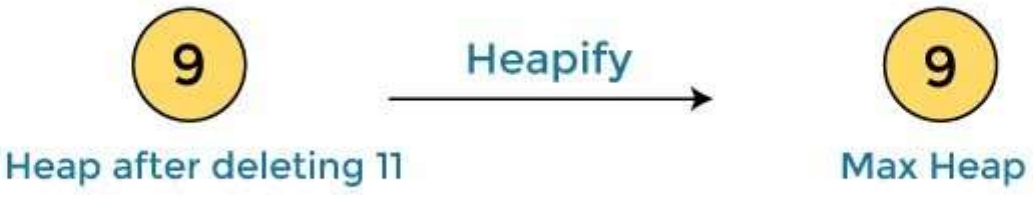


After swapping the array element **14** with **9** and converting the heap into max-heap, the elements of array are -

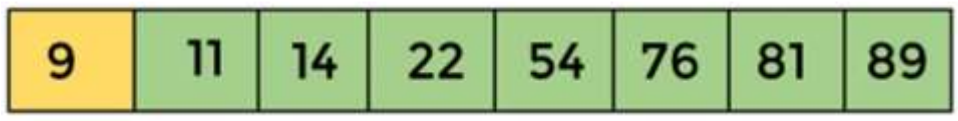


11	9	14	22	54	76	81	89

In the next step, again we have to delete the root element **(11)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **11** with **9,** the elements of array are -

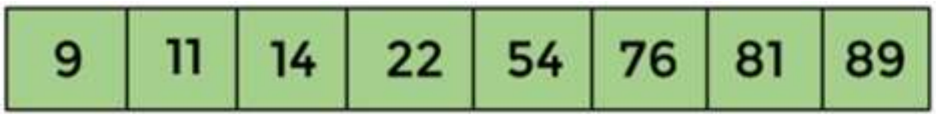


9	11	14	22	54	76	81	89

Now, heap has only one element left. After deleting it, heap will be empty.

9
Remove 9
Empty

After completion of sorting, the array elements are -



Now, the array is completely sorted.

Algorithm of insertion sort & Heap sort:

Insertion Sort:

```
for(i=1; i<=n-1; i++)
{
    x = a[i];
    j = i;

    while(a[j-1] > x && j>0)
    {
        a[j] = a[j-1];
        j--;
    }
    a[j] = x;
}
```

Heap Sort:

```
adjust(int a[], int i, int n)
{
    x = a[i];
    j= i*2;
    while(j<=n)
    {
        if(a[j] < a[j+1] && j<n)
        {
            j = j+1;
        }
        if (x>a[j])
        break;
        a[j/2] = a[j];
        j= j*2;
    }
    a[j/2] = x;
}

int main() {

    for(i=n/2; i>=1; i++)
    {
        adjust(a, i, n);
    }

    for(i=n/2; i>=1; i++)
    {
        adjust(a, i, n);
    }
    for(i=n; i>=1; i++)
    {
        t = a[i];
        a[1] = a[i];
        a[i] = t;
        adjust(a, 1, i-1);
    }
}
```

Derivation of Analysis insertion sort & Heap sort:

Insertion Sort:

Input: Given n input elements.

Output: Number of steps incurred to sort a list.

Logic: We are comparing 2 things – if (a[j-1] > x) and if (j > 0)

Worst Case Analysis
The worst-case time complexity is also **O(n^2)**, which occurs when we sort the ascending order of an array into the descending order.
In this algorithm, we divide the array into an unsorted and sorted array and pick an element from the unsorted array, compare it with the elements in the sorted array and then place it in its correct position. Therefore, for the nth pass, we compare n+1 times.

$2+3+4+\dots + n = n(n+1)/2 - 1 = n^2/2 + n/2 - 1$

Sum = i.e., O(n^2)

Best Case Analysis
The insertion sort algorithm has a best-case time complexity of **Ω(n)** for the already sorted array because here, only the outer loop is running n times, and the inner loop is kept still. Therefore in each pass, only 1 comparison is occurring.
Total comparisons = Number of passes = n – 1
Therefore, the time complexity will be Ω(n).

Average Case Analysis
The average-case time complexity for the insertion sort algorithm is **O(n^2)**, which is incurred when the existing elements are in jumbled order, i.e., neither in the ascending order nor in the descending order.
Therefore, for the nth pass, we compare it n+1 times.

$2+3+4+\ldots + n = n(n+1)/2 - 1 = n^2/2 + n/2 - 1$

Sum = i.e., $O(n^2)$

Heap Sort:

```
1   heapSort
2   {
3    BuildMaxHeap---------------O(?)
4    {
5      for (K=N to K>=1)------------------ O(?)
6     {
         6.1 swap;
         6.2 reheapify;------------------O(?)
7     }
8    }
9   }
```

The time complexity for the **BuildMaxHeap** function will be equal to the sum of the time complexities of the **for loop** and the **reheapify** function.

The for loop is going from K=N to K>=1, therefore, it runs for n times. Therefore, the time complexity will be O(n).

- Heapification Analysis –

  Maximum number of swaps = height of the node

  Maximum distance the node can cover = height of the tree

  Where, height of the tree = $\log_2 n$

- BuildMaxHeap

```
{
  for (i=n/2 to i>=1)------------------ O(?)
 {
    adjust;...........................O(?)
 }
}
```

  The time complexity of the for loop will be O(n) and for the adjust function it will be equal to the height of tree i.e O ($\log_2 n$)

  Therefore, technically the total time complexity to build the max heap will be O (n $\log_2$ n) but in reality, it is actually O(n).

  Total time at height h = Total nodes at height h * $\log_2 n$
  Where, the total nodes at height h = $[\frac{n}{2^{h+1}}]$

Hence, total time at height h = $[\frac{n}{2^{h+1}}] * O(h)$ ---- (for 1 level)

Total time at height h (for all the levels) = $\sum_{i=0}^{\log n} [\frac{n}{2^{h+1}}] * O(h)$

$$= \sum_{i=0}^{\log n} [\frac{n}{2^{h+1}}] * (c * h)$$

$$= (\frac{c*n}{2}) * \sum_{i=0}^{\log n} [\frac{h}{2^h}]$$

$$< (\frac{c*n}{2}) * \sum_{i=0}^{\infty} [\frac{h}{2^h}]$$

$$< (\frac{c*n}{2}) * 2$$

$$< (c * n)$$

$$< O(n)$$

Hence, proved that the time complexity of BuildMaxHeap function will be O(n).

- Reheapify

  The time complexity = $O(n \log_2 n)$

Total Time complexity = $O(n) + O(n \log_2 n)$

$= O(n \log_2 n)$

Worst Case Analysis
The worst case for heap sort might happen when all elements in the list are distinct. Therefore, we would need to call **BuildMaxHeap** every time we remove an element. In such a case, considering there are 'n' number of nodes-

The number of swaps to remove every element would be log(n), as that is the max height of the heap
Considering we do this for every node, the total number of moves would be n * (log(n)).
Therefore, the runtime in the worst case will be O(n(log(n)).

Best Case Analysis
The best case for heapsort would happen when all elements in the list to be sorted are identical. In such a case, for 'n' number of nodes-

Removing each node from the heap would take only a constant runtime, O(1). There would be no need to bring any node down or bring max valued node up, as all items are identical.
Since we do this for every node, the total number of moves would be n * O(1).
Therefore, the runtime in the best-case would-be O(n).

Average Case Analysis

In terms of total complexity, we already know that we can create a heap in O(n) time and do insertion/removal of nodes in O(log(n)) time. In terms of average time, we need to take into account all possible inputs, distinct elements or otherwise. If the total number of nodes is 'n', in such a case, the **BuildMaxHeap** function would need to perform:

log(n)/2 comparisons in the first iteration (since we only compare two values at a time to build max-heap)
log(n-1)/2 in the second iteration
log(n-2)/2 in the third iteration
and so on
So mathematically, the total sum would turn out to be-

(log(n))/2 + (log(n-1))/2 + (log(n-2))/2 + (log(n-3))/2 + ...
Upon approximation, the final result would be
=1/2(log(n!))
=1/2(n∗log(n)−n+O(log(n)))

Considering the highest ordered term, the average runtime of **BuildMaxHeap** would then be O(n(log(n)).

Since we call this function for all nodes in the final heapsort function, the runtime would be (n * O(n(log(n))). Calculating the average, upon dividing by n, we'd get a final average runtime of O(n(log(n)).

Program(s) of insertion sort & Heap sort:

Insertion Sort:
```
#include<stdio.h>
#include<stdlib.h>
void user()
 {
    int n;
    printf("Enter the size of array: ");
    scanf("%d",&n);
    int arr[n];

    int i,j,x=0;

    for(i=0; i<=n-1; i++){
       printf("Enter the Element %d: ",i+1);
       scanf("%d",&arr[i]);
    }

   for(i=1; i<=n-1; i++)
   {
      x = arr[i];
      j=i;
      while(arr[j-1]>x && j>0)
```

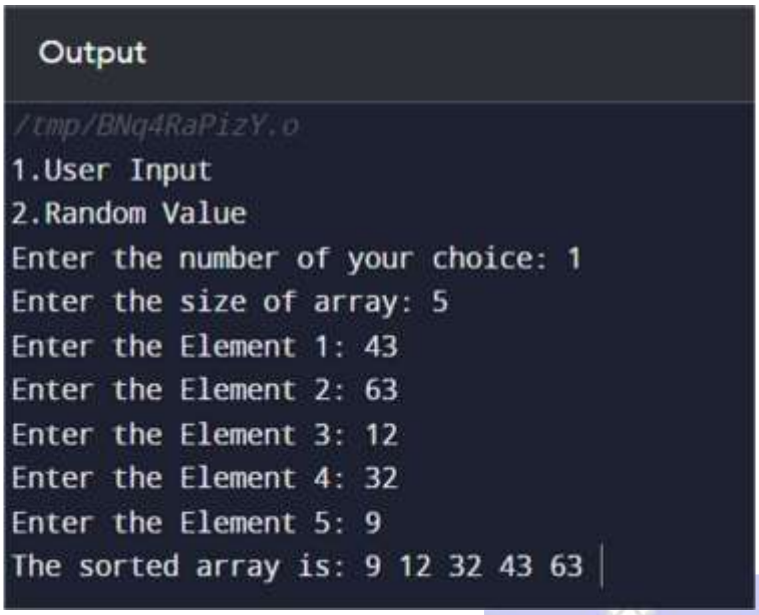
```
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = x;
        }
    printf("The sorted array is: ");
    for (i=0; i<n; i++)
    {
        printf("%d ", arr[i]);
    }

 }
int 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 R(int n)
{
printf("Enter the size of the array: ");
scanf("%d",&n);
int i, best[n], worst[n], avg[n];
for(i=0;i<n;i++)
    {
        best[i]=i;
        worst[i]=n-i;
        avg[i]=rand();
    }

    int best_swap = sort(best,n);
    int worst_swap = sort(worst,n);
    int avg_swap = 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);
}
void main(){
    int C, n;
    printf("1.User Input\n2.Random Value\n");
    printf("Enter the number of your choice: ");
    scanf("%d", &C);

    switch(C)
    {
        case 1:
            user();
            break;

        case 2:
            R(n);
            break;
        default:
            printf("Error! operator is not correct");
}
}
```

Heap Sort:

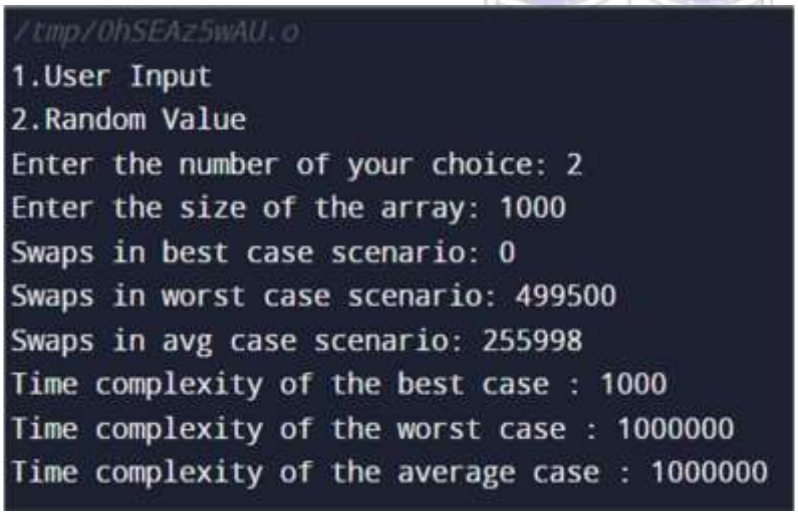
```
#include  <stdio.h>
#include <stdlib.h>
int counter = 0;

void adjust(int arr[], int i, int n) {
  int j, temp, count;
  temp = arr[i];
  j = 2 * i;

  while (j <= n) {
    if (j < n && arr[j+1] > arr[j])
      j = j + 1;
    if (temp > arr[j])
      break;
    else if (temp <= arr[j]) {
      arr[j/2] = arr[j];
      j = 2 * j;
    }
    counter++;
  }
  arr[j/2] = temp;
}

void heapsort(int arr[], int n) {
  int i, temp;
  for (i = n; i >= 2; i--) {
    temp = arr[i];
    arr[i] = arr[1];
    arr[1] = temp;
    adjust(arr, 1, i - 1);
    counter++;
  }
}
int main(){
    int i, n, arr[100];
  printf("Enter the number of elements in the array: ");
  scanf("%d", &n);

  printf("Enter the elements in the array:\n");
  for (i = 1; i <= n; i++)
    scanf("%d", &arr[i]);

  for (i = n/2; i >= 1; i--)
    adjust(arr, i, n);
  heapsort(arr, n);

  printf("The sorted array is:\n");
  for (i = 1; i <= n; i++)
  printf("%d ", arr[i]);

  printf("\nValue of counter is %d", counter);
  return 0;
}
```

Output(o) of insertion sort & Heap sort:

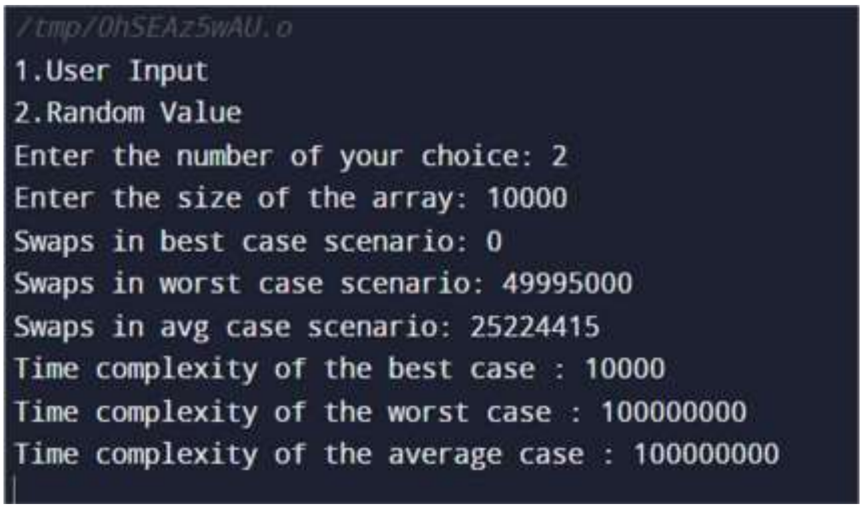
Insertion Sort:



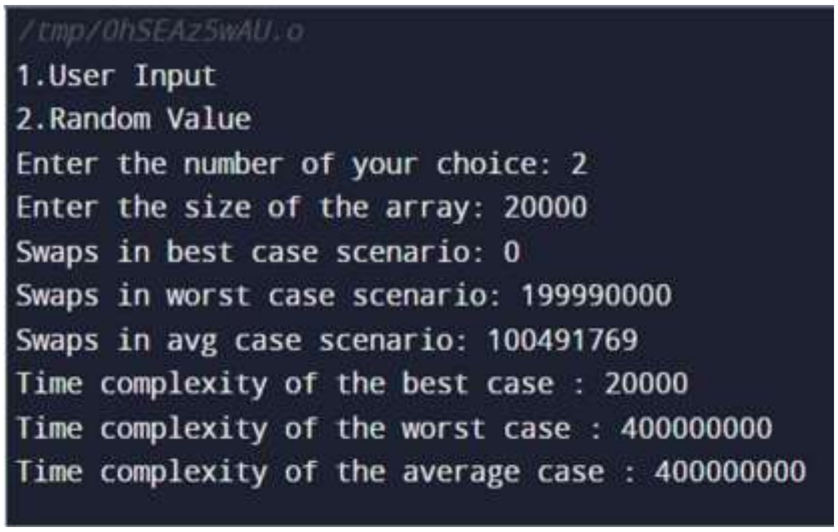
```
Output
/tmp/BNq4RaPizY.o
1.User Input
2.Random Value
Enter the number of your choice: 1
Enter the size of array: 5
Enter the Element 1: 43
Enter the Element 2: 63
Enter the Element 3: 12
Enter the Element 4: 32
Enter the Element 5: 9
The sorted array is: 9 12 32 43 63
```



```
/tmp/0hSEAz5wAU.o
1.User Input
2.Random Value
Enter the number of your choice: 2
Enter the size of the array: 1000
Swaps in best case scenario: 0
Swaps in worst case scenario: 499500
Swaps in avg case scenario: 255998
Time complexity of the best case : 1000
Time complexity of the worst case : 1000000
Time complexity of the average case : 1000000
```

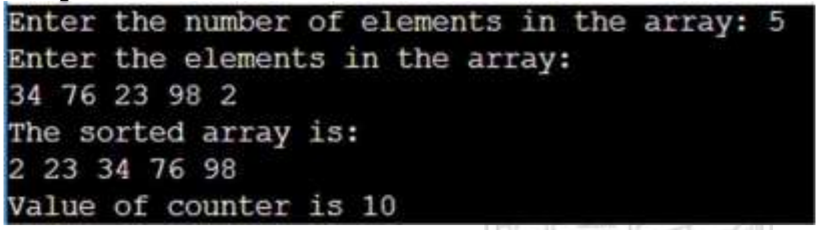


```
/tmp/0hSEAz5wAU.o
1.User Input
2.Random Value
Enter the number of your choice: 2
Enter the size of the array: 10000
Swaps in best case scenario: 0
Swaps in worst case scenario: 49995000
Swaps in avg case scenario: 25224415
Time complexity of the best case : 10000
Time complexity of the worst case : 100000000
Time complexity of the average case : 100000000
```

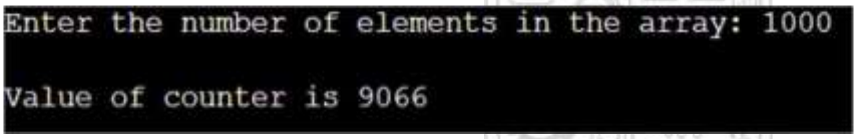


```
/tmp/0hSEAz5wAU.o
1.User Input
2.Random Value
Enter the number of your choice: 2
Enter the size of the array: 20000
Swaps in best case scenario: 0
Swaps in worst case scenario: 199990000
Swaps in avg case scenario: 100491769
Time complexity of the best case : 20000
Time complexity of the worst case : 400000000
Time complexity of the average case : 400000000
```

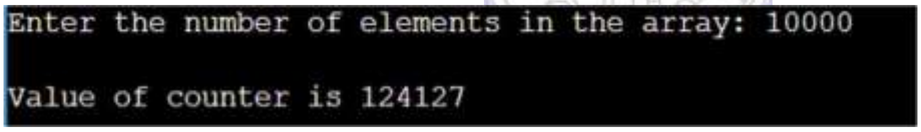
Heap Sort:



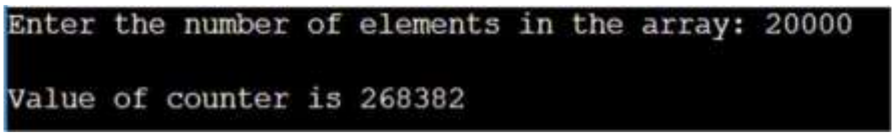
```
Enter the number of elements in the array: 5
Enter the elements in the array:
34 76 23 98 2
The sorted array is:
2 23 34 76 98
Value of counter is 10
```



```
Enter the number of elements in the array: 1000

Value of counter is 9066
```



```
Enter the number of elements in the array: 10000

Value of counter is 124127
```



```
Enter the number of elements in the array: 20000

Value of counter is 268382
```

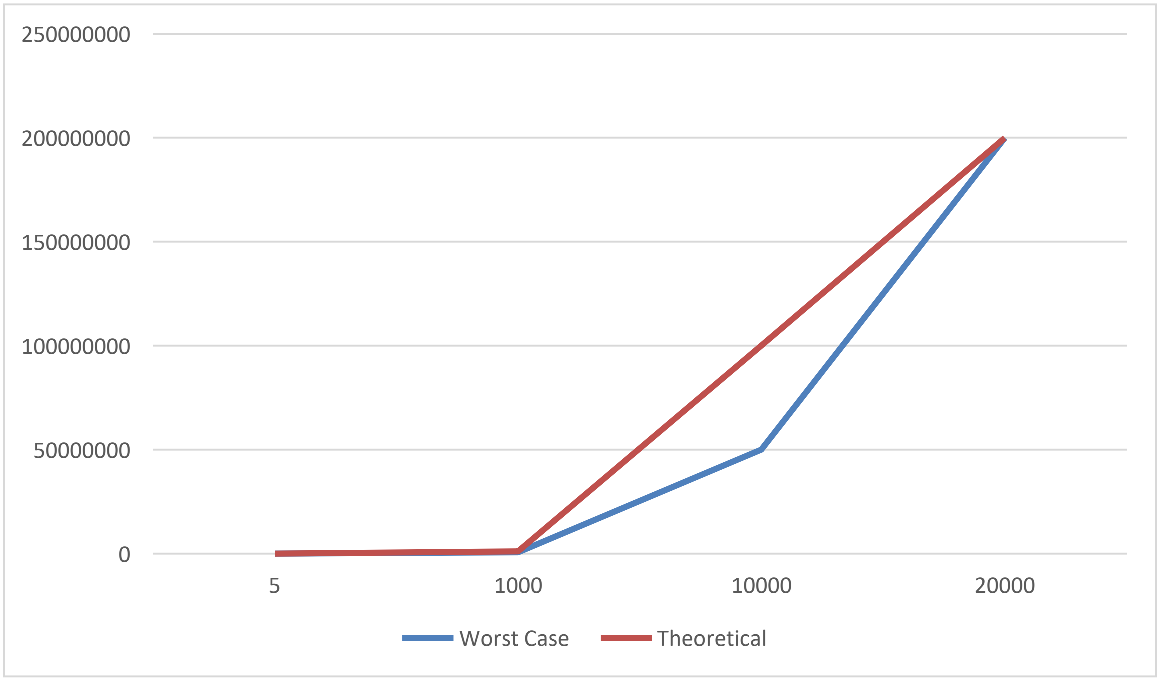
Results:

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	5	10	25
2	1000	499500	$(1000)^2$
3	10000	49995000	$(10000)^2$
4	20000	199990000	$(20000)^2$

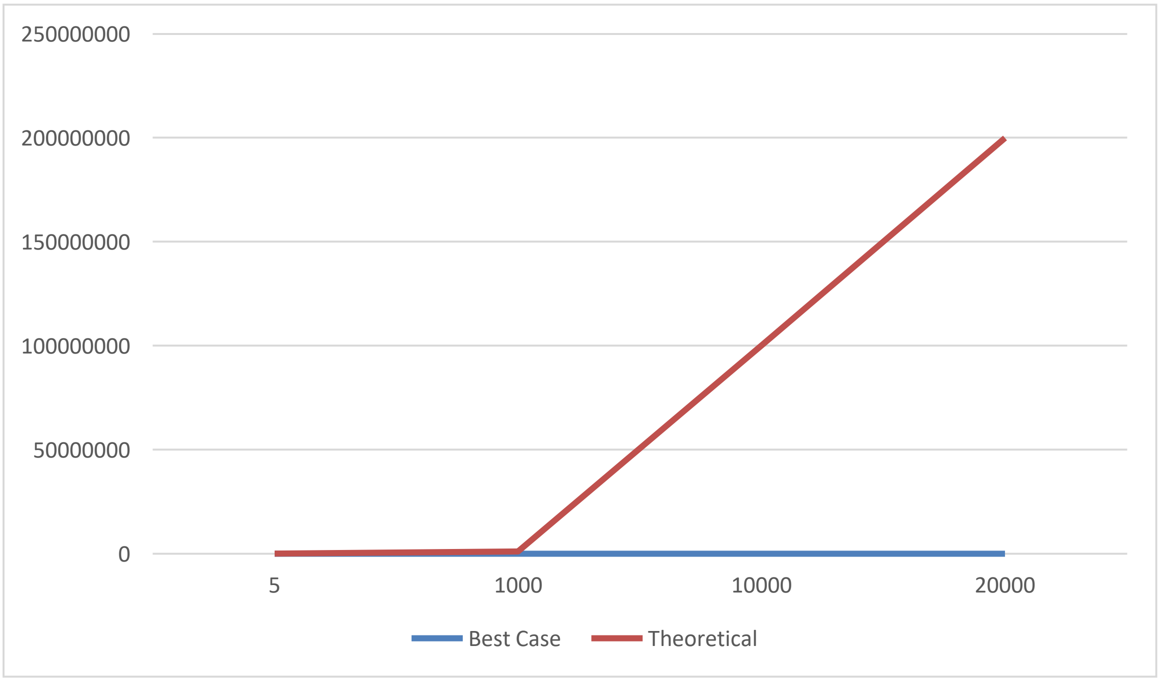
GRAPH:



Best Case Analysis:

Sr. No.	Input size	No: of steps from Algorithm analysis	No: of steps from Theoretical analysis
1	5	0	25
2	1000	0	$(1000)^2$
3	10000	0	$(10000)^2$
4	20000	0	$(20000)^2$

GRAPH:

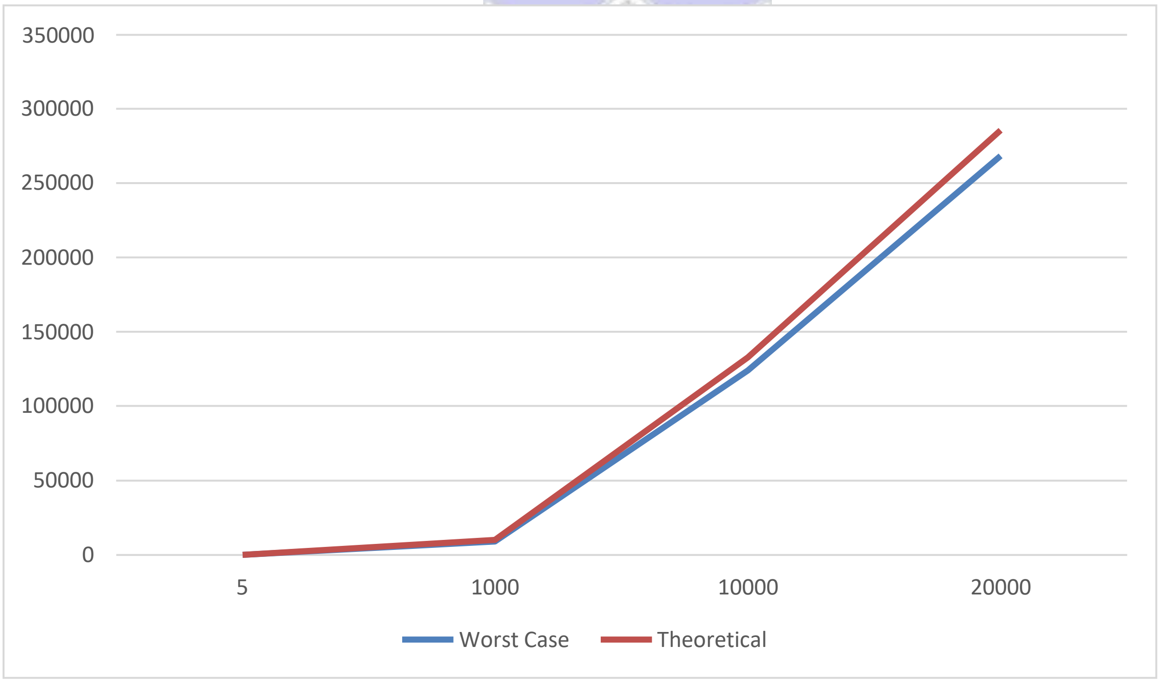


Time Complexity of Heap sort:

Worst Case Analysis:

Sr. No.	Input size	No: of steps from Algorithm analysis	No: of steps from Theoretical analysis
1	5	10	5 log(5)
2	1000	9066	1000 log(1000)
3	10000	124127	10000 log(10000)
4	20000	268382	20000 log(20000)

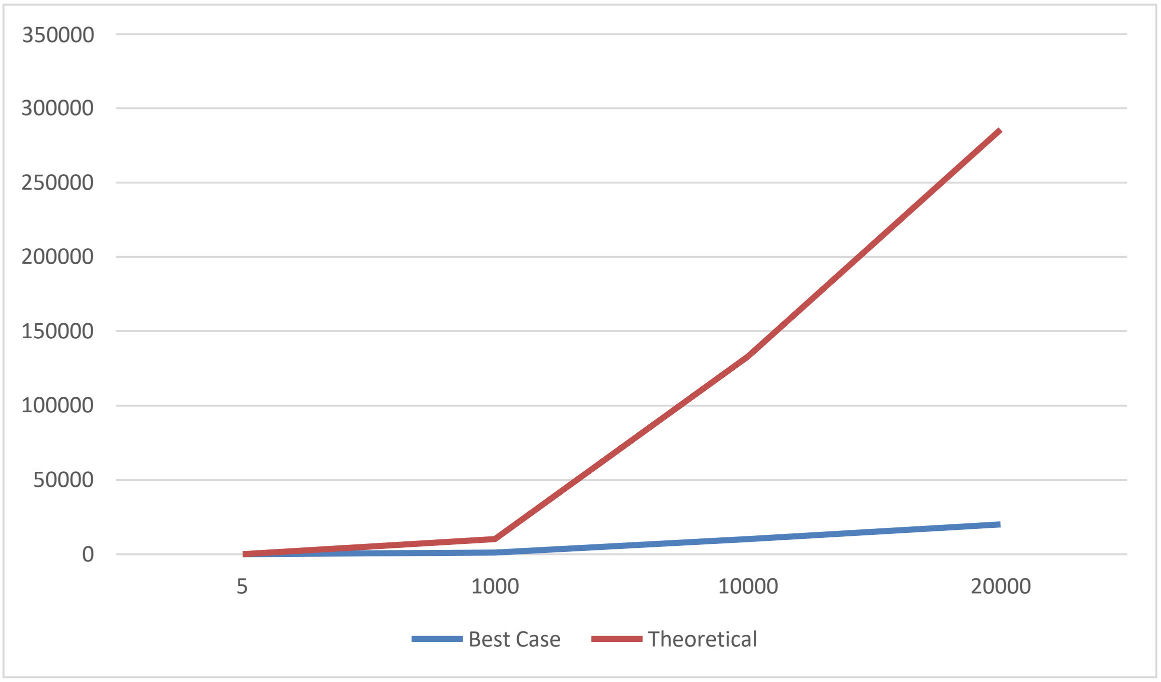
GRAPH:



Best Case Analysis:

Sr. No.	Input size	No: of steps from Algorithm analysis	No: of steps from Theoretical analysis
1	5	5	5 log(5)
2	1000	1000	1000 log(1000)
3	10000	10000	10000 log(10000)
4	20000	20000	20000 log(20000)

GRAPH:



Conclusion: (Based on the observations): We successfully implemented Insertion sort and Heap sort and then analysed their time complexity. We observed that the actual values are usually lesser than compared to the theoretical values.

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.