1. Implement and perform time analysis of following algorithms

1.1 Find factorial number of given integer using iterative and recursive method.

Theory:

Factorial of a non-negative integer, is multiplication of all integers smaller than or equal to n. For example factorial of 6 is 6*5*4*3*2*1 which is 720.

Iterative Algorithm:

```
n! = n * (n-1) * (n-2) * (n-3) * .... * 1
```

Recursive Algorithm:

```
n! = n * (n-1)!
n! = 1 if n = 0 or n = 1
```

Implementation

```java
import java.util.Scanner;
class FactorialI
{
    public static void main(String args[])
    {
        int fact=1,n,i;
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter the number : ");
        n = sc.nextInt();

        long sT = System.nanoTime();
        for(i=1;i<=n;i++)
        {
            fact=(fact*i);
        }
        long eT = System.nanoTime();
        long RunningTime = (eT - sT);

        System.out.println("Answer is : "+ fact);
        System.out.print("Total Running Time : "+ RunningTime);
    }

}
```

```java
import java.util.Scanner;
class FactorialR
{
    public static int ftl(int n)
    {
        if(n == 0)
        {
            return 1;
        }
        else
        {
            return (n*ftl(n-1));
        }
    }
    public static void main(String args[])
    {
        int fact=1,num;
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter the number : ");
        num = sc.nextInt();

        long sT = System.nanoTime();
        fact=ftl(num);
        long eT = System.nanoTime();
        long RunningTime = (eT - sT);

        System.out.println("Answer is " + fact);
        System.out.print("Total Running Time : "+ RunningTime);

    }

}
```

Time analysis of Factorial Algorithm

Size of input	Iterative	Recursive
	Running Time (nanoseconds)	Running Time (nanoseconds)
4	600	6400
5	900	4400
6	600	4200
7	1000	4500
8	900	4400
9	700	5700
10	800	5700
11	1100	6600
12	800	4300
13	1100	5700
14	900	4100

1.2 Linear Search algorithm.

Theory:

Search is a process of finding a value in a list of values. In other words, searching is the process of locating given value position in a list of values.

Linear search algorithm finds a given key in a list of elements. This search process starts comparing key with the first element in the list. If both are matched then result is key is found otherwise key is compared with the next element in the list. Repeat the same until key is compared with the last element in the list, if that last element also doesn't match, then the result is "Key not found in the list". That means, the key is compared with element by element in the list. It is also known as sequential search.

Algorithm: Linear Search (a[1...n], n, key)

```
{
        found=0;
        i=1;
        while( i <= n)
        {
           if(key==a[i])
          {
                found=1;
                break;
          }
          i++
        }
}
```

Implementation:

```java
import java.util.Scanner;
class LinearSearch
{
    public static void main(String args[])
    {
        int key,i,n;


        Scanner sc = new Scanner(System.in);
        System.out.print("Enter the n : ");
        n = sc.nextInt();

        int[] A = new int[n];

        System.out.print("Enter the value of key : ");
        key = sc.nextInt();
```

```
        //System.out.println("Enter element of array");
        for(i=0 ; i<n ; i++)
        {
            A[i] = i;
        }

        long sT = System.nanoTime();
        for(i=0;i<n;i++)
        {
            if(A[i] == key)
            {
                break;
            }
        }
        long eT = System.nanoTime();
        long RunningTime = (eT - sT);

        if (i == n)
        {
            System.out.println(key + " is not found & Total running time is :
" + RunningTime);
        }
        else
        {
            System.out.println("The index of " + key + " is : " + i + " & " +
"Total running time is : "
                    + RunningTime);
        }

    }
}
```

Time analysis of Linear Search Algorithm

Size of input	Best Case Running Time (ns)	Worst Case Running Time (ns)
1000	300	34900
2000	500	34500
3000	200	90300
4000	600	68800
5000	900	96100
6000	400	180600
7000	1200	245700
8000	600	226800
9000	500	155300
10000	700	219200

1.3 Bubble Sort, Selection Sort and Insertion Sort algorithms.

A sorting algorithm is an algorithm that puts elements of a list in a certain order either ascending or descending.

1) Bubble sort:

Theory:

Bubble sort is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.

Algorithm Bubble_Sort (a[1…n], n)

```
{
        for i = 1 to n-1
        {
             swapped = false;
             for j = 1 to n-i
            {
                 if ( a[j] > a[j+1] )
                {
                        swap ( a[j] ←→a[j+1])
                        swapped = true
                 }
            }
            if(swapped= = false)
                  break
        }

 }
```

Implementation:

```java
import java.util.*;
class Bubble
{
    public static void main(String args[])
    {
        int i,j,temp,n,swap=0,cmp=0;

        Scanner sc = new Scanner(System.in);

        System.out.print("Enter the size of array : ");
         n = sc.nextInt();
        int[] A = new int[n];

        Random rd = new Random();

        for(i=0;i<=A.length-1;i++)
        {
            //A[i] = i;
            A[i]=  n-i;
```

```
        }

        long sT = System.nanoTime();
        for(i=0;i<A.length-1;i++)
        {
            for(j=0;j<A.length-1;j++)
            {
                cmp++;
                if(A[j] > A[j+1])
                {
                    temp = A[j];
                    A[j] = A[j+1];
                    A[j+1] = temp;
                    swap++;
                }

            }
        }
        long eT = System.nanoTime();
        long RunningTime = (eT - sT);

        System.out.print("Total running time is : " + RunningTime + " , Total
Swaps : " + swap + " , Total comparisons : " + cmp);
    }
}
```

Time analysis of Bubble Sort Algorithm

Size of Input	Best Case			Worst Case		
	#Comparisons	#Swaps	Running Time (ns)	#Comparisons	#Swaps	Running Time (ns)
100	9801	0	387000	9801	4950	234100
200	39601	0	1648700	39601	19900	1380300
300	89401	0	4063600	89401	44850	2218700
400	159201	0	4121100	159201	79800	4932200
500	249001	0	4381700	249001	124750	7112400
600	358801	0	4278500	358801	179700	8894200
700	488601	0	7820500	488601	244650	6966200
800	638401	0	7098300	638401	319600	9685600
900	808201	0	21092200	808201	404550	10087000
1000	998001	0	10323300	998001	499500	167688000

2) Selection sort

Theory :

In selection sort algorithm the input array 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.

In selection sort, the first smallest element is selected from the unsorted part and placed at the first position. After that second smallest element is selected and placed in the second position. The process continues until the array is entirely sorted.

Algorithm Selection_Sort (a[1..n], n)

```
{
        for i = 1 to n
        {
            min = i
            for j = i+1 to n
            {
                if (a[j] < a[min])
                        min = j
            }
            swap ( a[i] ←→a[min])
        }
}
```

Implementation:

```java
import java.util.*;

public class SelectionSort {
    public static void main(String args[])
    {
        int temp,cmp = 0, swap = 0;

        Scanner sc = new Scanner(System.in);
        Random rd = new Random();
        System.out.print("Enter the size of array : ");
        int n = sc.nextInt();
        int[] A = new int[n];

        for (int i = 0; i < n; i++)
        {
            //A[i]=i;
            A[i]=n-i;
        }

        long sT = System.nanoTime();
        for (int i = 0; i < n; i++)
        {
            int min = i;
            for (int j = i + 1; j < n; j++)
            {
                if (A[min] > A[j])
```

```
                {
                    cmp++;

                    min = j;
                }
            }
            swap++;

            temp = A[min];
            A[min] = A[i];
            A[i] = temp;
        }
        long eT = System.nanoTime();
        long RunningTime = (eT - sT);
        System.out.println("Total running time is : " + RunningTime + " Total
Swaps : " + swap + " , Total Comparitions : " + cmp);
    }
}
```

Time analysis of Selection Sort Algorithm

Size of Input	Best Case			Worst Case		
	#Comparisons	#Swaps	Running Time (ns)	#Comparisons	#Swaps	Running Time (ns)
100	0	100	79100	2500	100	72400
200	0	200	357100	10000	200	718600
300	0	300	907100	22500	300	931200
400	0	400	1118400	40000	400	1990500
500	0	500	2794700	62500	500	3787800
600	0	600	2363500	90000	600	2765000
700	0	700	3666800	122500	700	1939200
800	0	800	6061300	160000	800	4152000
900	0	900	3319300	202500	900	4153500
1000	0	1000	4337700	250000	1000	4584700

3) Insertion sort:

Theory:

In an insertion sort, the first element in the array is considered as sorted, even if it is an unsorted array. In an insertion sort, each element in the array is checked with the previous elements, resulting in a growing sorted output list. With each iteration, the sorting algorithm removes one element at a time and finds the appropriate location within the sorted array and inserts it there. The iteration continues until the whole list is sorted.

Algorithm Insertion_Sort (a[1...n], n)

{

```
for j=2 to n
{
    key = a[j]
    i = j-1
    while (i>=1 and key<a[i])
    {
        a[i+1] = a[i]
        i = i-1
    }
    a[i+1] = key
}
```

}

Implementation:

```java
import java.util.*;

public class InsertionSort
{
    public static void main(String args[])
    {
       int i,j,temp, cmp = 0, shift = 0;

        Scanner sc = new Scanner(System.in);
        Random rd = new Random();
        System.out.print("Enter the size of array : ");
        int n = sc.nextInt();

        int[] A = new int[n];
        for (i = 0; i < n; i++)
        {
            //A[i]=i;
            A[i]=n-i;
        }

        long sT = System.nanoTime();
        for (i = 0; i < n; i++)
        {
            temp = A[i];
```

```
            for (j = i - 1; j >= 0 && temp < A[j]; j--)
            {
                ++cmp;

                A[j + 1] = A[j];
            }
            shift++;
            A[j + 1] = temp;
        }
        long eT = System.nanoTime();
        long RunningTime = (eT - sT);

        System.out.println("Total running time is : " + RunningTime +  " ,
Total Shifts : " + shift + " , Total Comparisions : " + (cmp));
    }
}
```

Time analysis of Insertion Sort Algorithm

Size of Input	Best Case			Worst Case		
	#Comparisons	#Shifts	Running Time (ns)	#Comparisons	#Shifts	Running Time (ns)
100	0	100	4400	4950	100	94500
200	0	200	8000	19900	200	538700
300	0	300	11500	44850	300	734800
400	0	400	16100	79800	400	1467200
500	0	500	19900	124750	500	2297900
600	0	600	17200	179700	600	2906200
700	0	700	31000	244650	700	3158500
800	0	800	23400	319600	800	4575900
900	0	900	23900	404550	900	2664300
1000	0	1000	42400	499500	1000	5323700

1.4 Max Heap Sort Algorithm

Theory:

Heap sort is a comparison based sorting technique based on Binary Heap data structure. In heap sort we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

Let us first define a Complete Binary Tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible

A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array.

Since a Binary Heap is a Complete Binary Tree, it can be easily represented as array and array based representation is space efficient. If the parent node is stored at index I, the left child can be calculated by 2 * I + 1 and right child by 2 * I + 2 (assuming the indexing starts at 0).

Heap Sort Algorithm for sorting in increasing order:

1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
3. Repeat above steps while size of heap is greater than 1.

Implementation:

Time analysis of Heap Sort Algorithm

Size of input	Running Time (ns)
10	
15	
20	
25	
30	
35	
40	
45	
50	