Practical 1: Sorting Arrays-I

Aim

Implement the **Insertion sort** algorithm for an array. Measure the execution time and the number of steps required to execute each algorithm in best case, worst case and average case.

Theory

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.

Algorithm

To sort an array of size N in ascending order:

- Iterate from arr[1] to arr[n] over the array.
- Compare the current element (key) to its predecessor.
- If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Program

```
import java.util.*;

public class InsertionSort {

   static void insertionSort(int array[]) {

      for (int i = 1; i < array.length; i++) {

         int key = array[i];

         for (int j=i-1 ; j>-1 ; j--){

            if ( array [j] > key ){

               array [j+1] = array [j];

               array[j] = key;

            }

         }

      }

   }


   public static void main(String[] args) {
```

```
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter the number of elements: ");
        int length = sc.nextInt();

        int[] array = new int[length];

        for(int k = 0; k < length; k++) {
           System.out.printf("Enter element %d: ", k+1);
           array[k] = sc.nextInt();
        }
        System.out.print("Data you entered: ");
        for(int k = 0; k < length; k ++) {
           System.out.print(array[k] + " ");
        }
        insertionSort(array);
        System.out.print("\nAfter Sorting array: ");
        for(int k = 0; k < length; k ++) {
           System.out.print(array[k] + " ");
        }
    }
}
```

Output

```
Enter the number of elements: 7
Enter element 1: 43
Enter element 2: 67
Enter element 3: 55
Enter element 4: 12
Enter element 5: 98
Enter element 6: 5
Enter element 7: 23
Data you entered: 43 67 55 12 98 5 23
After Sorting array: 5 12 23 43 55 67 98
```

Analysis of Algorithm

i. Worst Case Complexity

Suppose, an array is in descending order, and you want to sort it in ascending order. In this case, worst case complexity occurs. Each element has to be compared with each of the other elements so, for every nth element, (n-1) number of comparisons are made.

Thus, the total number of comparisons = n(n-1) = $\mathbf{n^2}$

ii. Best case Complexity

When the array is already sorted, the outer loop runs for n number of times whereas the inner loop does not run at all. So, there are only **n** number of comparisons. Thus, complexity is linear.

iii. Average case Complexity

It occurs when the elements of an array are in jumbled order. So, we can't say how much comparisons will be made. Hence the average case complexity would be of order on $\mathbf{n^2}$,

Description	**Best Case**	**Average case**	**Worst case**
Occurrence	It occurs when Array is sorted.	It occurs when Array is randomly sorted.	It occurs when Array is reverse sorted, and tj =j
Complexity	O(n)	$O(n^2)$	$O(n^2)$

Aim

Implement the **Selection sort** algorithm for an array. Measure the execution time and the number of steps required to execute each algorithm in best case, worst case and average case.

Theory

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from the unsorted part and putting it at the beginning.

The algorithm maintains two subarrays in a given array.

- The subarray which already sorted.
- The remaining subarray was unsorted.

In every iteration of the selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the beginning of unsorted subarray.

After every iteration sorted subarray size increase by one and unsorted subarray size decrease by one.

After N (size of array) iteration we will get sorted array.

Algorithm

- selectionSort(array, size)
- repeat (size - 1) times
- set the first unsorted element as the minimum
- for each of the unsorted elements
- if element < currentMinimum
- set element as new minimum
- swap minimum with first unsorted position
- end selectionSort

Program

```
import java.util.*;

public class SelectionSort {

   static void selectionSort(int array[]) {

      for (int i = 0 ; i < array.length ; i++) {

         int min = Integer.MAX_VALUE;

         int index = 0;

         for (int j=i ; j<array.length ; j++){

            if (min > array[j]){

               min = array[j];

               index = j;

            }
```

```
            }
            array[index] = array[i];
            array[i] = min;
        }
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter the number of elements: ");
        int length = sc.nextInt();

        int[] array = new int[length];

        for(int k = 0; k < length; k++) {
            System.out.printf("Enter element %d: ", k+1);
            array[k] = sc.nextInt();
        }
        System.out.print("Data you entered: ");
        for(int k = 0; k < length; k ++) {
            System.out.print(array[k] + " ");
        }

        selectionSort(array);

        System.out.print("\nAfter Sorting array: ");
        for(int k = 0; k < length; k ++) {
            System.out.print(array[k] + " ");
        }
    }
}
```

Output:

```
Enter the number of elements: 7
Enter element 1: 67
Enter element 2: 536
Enter element 3: 254
Enter element 4: 68
Enter element 5: 42
Enter element 6: 342
Enter element 7: 54
Data you entered: 67 536 254 68 42 342 54
After Sorting array: 42 54 67 68 254 342 536
```

Analysis of Algorithm

Cycle	Number of Comparison
1st	(n-1)
2nd	(n-2)
3rd	(n-3)
4th	(n-4)
5th	(n-5)
...	...
last	1

Number of comparisons: (n - 1) + (n - 2) + (n - 3) + ... + 1 = n (n - 1) / 2 nearly equals to $\mathbf{n^2}$.

Whether it is Best/Worst/Average case both loops would execute. Hence, the complexity would be of order of $\mathbf{n^2}$.

Description	Best Case	Average case	Worst case
Occurrence	It occurs when Array is sorted.	It occurs when Array is randomly sorted.	It occurs when Array is reverse sorted, and $t_j = j$
Complexity	$O(n^2)$	$O(n^2)$	$O(n^2)$