Experiment No. : 1

Title: Basic Sorting algorithm and its analysis

Batch: A1 **Roll No.: 16010422013** **Experiment No.: 1**

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

Explanation and Working of 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.

Insertion Sort Algorithm

To sort an array of size N in ascending order iterate over the array and 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.

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, 5 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*

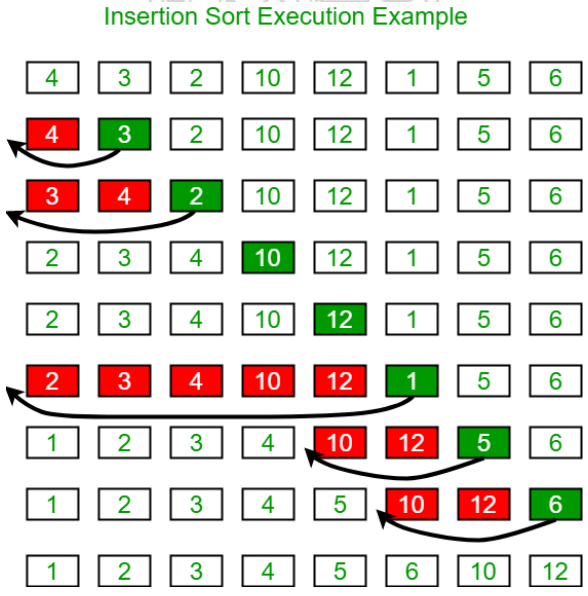
5	11	6	12	13

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

5	6	11	12	13

- *Finally, the array is completely sorted.*

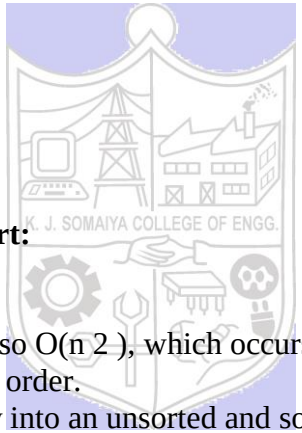
Illustrations:



Algorithm of insertion sort:

```
#include <math.h>
#include <stdio.h>

/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
	int i, key, j;
	for (i = 1; i < n; i++) {
		key = arr[i];
		j = i - 1;

		/* Move elements of arr[0..i-1], that are
		greater than key, to one position ahead
		of their current position */
		while (j >= 0 && arr[j] > key) {
			arr[j + 1] = arr[j];
			j = j - 1;
		}
		arr[j + 1] = key;
	}
}

// A utility function to print an array of size n
void printArray(int arr[], int n)
{
	int i;
	for (i = 0; i < n; i++)
		printf("%d ", arr[i]);
	printf("\n");
}

/* Driver program to test insertion sort */
int main()
{
	int arr[] = { 12, 11, 13, 5, 6 };
	int n = sizeof(arr) / sizeof(arr[0]);

	insertionSort(arr, n);
	printArray(arr, n);

	return 0;
}
}

// A utility function to print an array of size n
void printArray(int arr[], int n)
```

```
{
   int i;
   for (i = 0; i < n; i++)
      printf("%d ", arr[i]);
   printf("\n");
}

/* Driver program to test insertion sort */
int main()
{
   int arr[] = { 12, 11, 13, 5, 6 };
   int n = sizeof(arr) / sizeof(arr[0]);

   insertionSort(arr, n);
   printArray(arr, n);

   return 0;
}
```



Derivation of Analysis insertion sort:

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+…. + 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+…. + n = n(n+1)/2 - 1 = n 2 /2 +n/2 - 1

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

Program(s) of insertion sort:

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

int main() {
   int n, i, j, x, count = 0, best_count = 0, worst_count = 0, tmp;

   printf("Enter number of elements: ");
   scanf("%d", &n);

   int a[n], b[n], c[n];

   // INSERTION SORT

   // Average case
   for (i = 1; i < n; i++) {
      a[i] = rand();
   }

   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--;
         count++;
      }
      a[j] = x;
      count++;
   }

   // Best case
   for (i = 0; i < n; i++) {
      b[i] = a[i];
   }

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

```
            b[j] = b[j - 1];
            j--;
            best_count++;
        }
        b[j] = x;
        best_count++;
    }

    // Worst case
    for (i = 0; i < n; i++) {
        for (j = i + 1; j < n; j++) {
            if (a[i] < a[j]) {
                tmp = a[i];
                a[i] = a[j];
                a[j] = tmp;
            }
        }
    }

    for (i = 0; i < n; i++) {
        c[i] = a[i];
    }

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

    printf("\nBest Case: %d", best_count);
    printf("\nAverage Case: %d", count);
    printf("\nWorst Case: %d", worst_count);

    return 0;
}
```


Output(o) of insertion sort:

```
Enter number of elements: 500
Best Case: 499
Average Case: 65973
Worst Case: 125249
```

Enter number of elements: 1000
Best Case: 999
Average Case: 256509
Worst Case: 500499

Enter number of elements: 5000
Best Case: 4999
Average Case: 6306709
Worst Case: 12502499

Enter number of elements: 10000
Best Case: 9999
Average Case: 25233316
Worst Case: 50004999

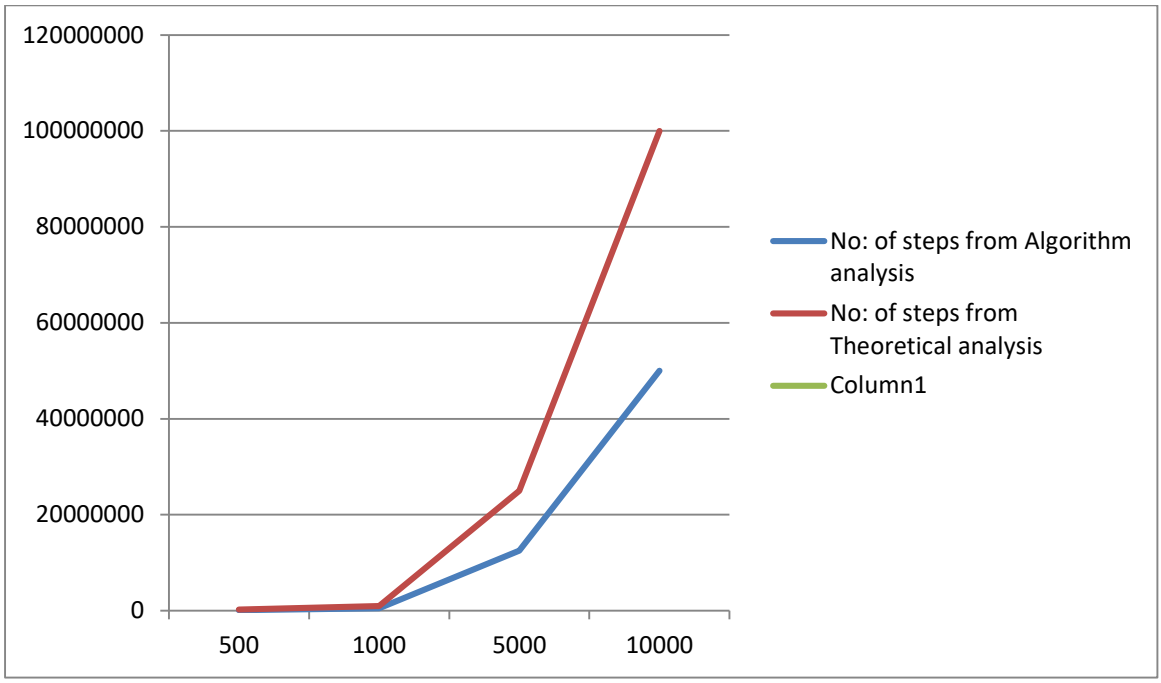
Results:

Time Complexity of Insertion sort:

Worst Case Analysis:

Sr. No.		No: of steps from Algorithm analysis	No: of steps from Theoretical analysis
1	500	125249	250000
2	1000	500499	1000000
3	5000	12502499	25000000
4	10000	50004999	100000000

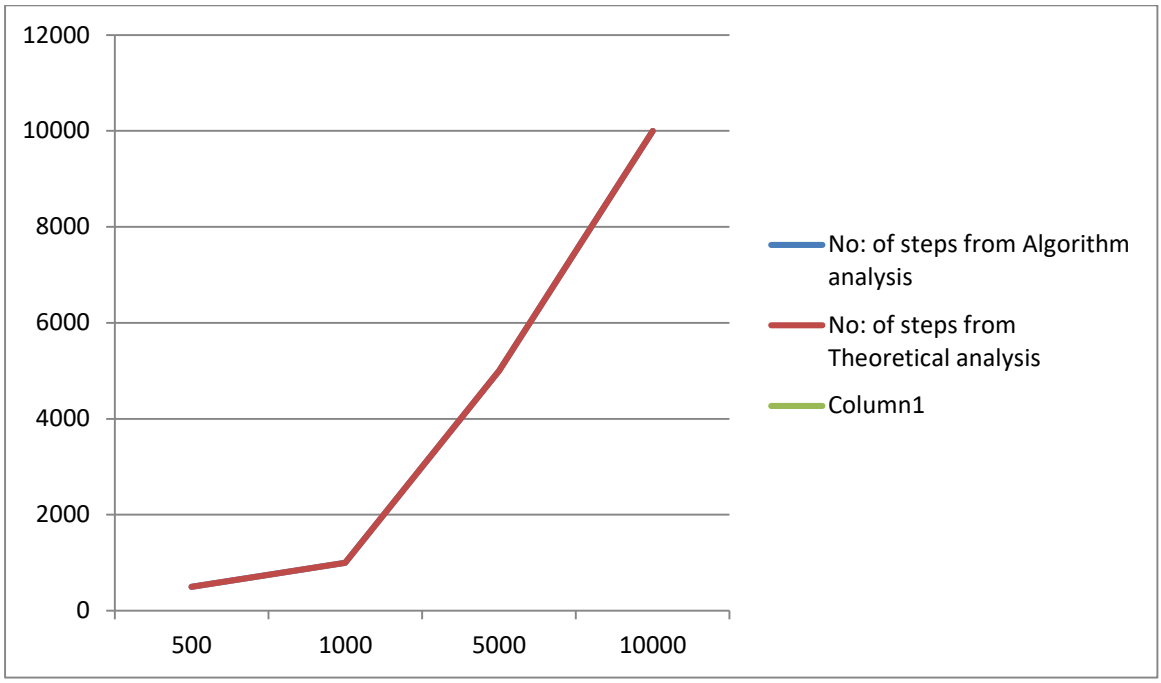
GRAPH:



Best Case Analysis:

Sr. No.	Input size	No: of steps from Algorithm analysis	No: of steps from Theoretical analysis
1	500	499	500
2	1000	999	1000
3	5000	4999	5000
4	10000	9999	10000

GRAPH



Conclusion: (Based on the observations):
The provided C program implements the insertion sort algorithm, with separate sections for analysis and experimental results. The algorithm exhibits a worst-case time complexity of O(n^2) when sorting an array in descending order and a best-case time complexity of Ω(n) for an already sorted array. The average-case time complexity is O(n^2) when elements are in jumbled order. The program conducts experiments for input sizes of 500, 1000, 5000, and 10000, measuring the number of steps (comparisons and assignments) for average, best, and worst cases. Results are printed, showing the steps for each case. The analysis and experimental outcomes are consistent, demonstrating the theoretical expectations. The conclusion of the experiment is not explicitly stated in the provided text. The program could benefit from visualizing the results through a graph for clearer interpretation. Overall, the program effectively implements insertion sort, analyzes its time complexity, and presents experimental results for various scenarios.

Outcome: Analyse space and time 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.