

## EXPERIMENT NO 1

**Title:** To implement a program for Insertion sort and find its complexity.

**Aim:** - Write a program to sort elements of an array in ascending order using insertion sort algorithm and finding its complexity.

**Lab Outcomes : CSL401.1:** Demonstrate the ability to develop and implement algorithms using the divide and conquer approach, and analyze the complexity of various sorting algorithms.

**Theory:** -

Insertion sort is a faster and more improved sorting algorithm than selection sort. In selection sort the algorithm iterates through all of the data through every pass whether it is already sorted or not. However, insertion sort works differently, instead of iterating through all of the data after every pass the algorithm only traverses the data it needs to until the segment that is being sorted is sorted.

Pseudo Code for insertion sort:-

INSERTION-SORT (A)

```
1 for j = 2 to A. length
2   key = A[j]
3   // Insert A[j] into the sorted sequence A [1...j-1]
4   i = j - 1
5   while i > 0 and A[i] > key
6     A [i+1] = A[i]
7     i = i - 1
8   A [i+1] = key
```

How Insertions Sort Algorithm works?

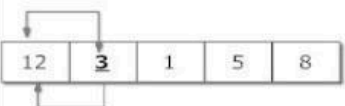
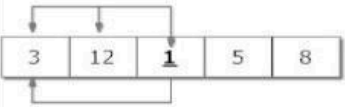
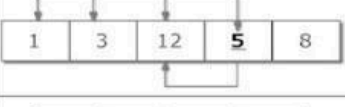


Step 1		Checking second element of array with element before it and inserting it in proper position. In this case, 3 is inserted in position of 12.
Step 2		Checking third element of array with elements before it and inserting it in proper position. In this case, 1 is inserted in position of 3.
Step 3		Checking fourth element of array with elements before it and inserting it in proper position. In this case, 5 is inserted in position of 12.
Step 4		Checking fifth element of array with elements before it and inserting it in proper position. In this case, 8 is inserted in position of 12.
		Sorted Array in Ascending Order

Figure: Sorting Array in Ascending Order Using Insertion Sort Algorithm

Suppose, you want to sort elements in ascending as in above figure, then,

**Step 1:** The second element of an array is compared with the elements that appear before it (only first element in this case). If the second element is smaller than the first element, the second element is inserted in the position of the first element. After the first step, the first two elements of an array will be sorted.

**Step 2:** The third element of an array is compared with the elements that appears before it (first and second element). If the third element is smaller than the first element, it is inserted in the position of the first element. If the third element is larger than the first element but smaller than the second element, it is inserted in the position of the second element. If the third element is larger than both the elements, it is kept in the position as it is. After the second step, the first three elements of an array will be sorted.

**Step 3:** Similarly, the fourth element of an array is compared with the elements that appear before it (first, second and third element) and the same procedure is applied and that element is inserted in the proper position. After the third step, the first four elements of an array will be sorted.

If there are  $n$  elements to be sorted, then, this procedure is repeated  $n-1$  times to get a sorted list of arrays.

## General Run Time

The run time for insertion sort can then be written as

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n-1)$$

Where the  $c_i$ 's are the cost of each line (noting that  $c_3 = 0$  since line 3 is a comment).

### Case 1: Best Case

The best case for insertion sort is when the input array is already sorted, in which case the while loop never executes (but the condition must be checked once). Thus  $t_j = 1$  for all  $j = 2, 3, \dots, n$  (i.e.  $t_j - 1 = 0$ ) and the run time reduces to:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^n 1 + c_8(n-1) \\ &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \\ &= an + b \end{aligned}$$

which is a *linear* function of  $n$ .

### Case 2: Worst Case

The worst case for insertion sort is when the input array is in reverse (decreasing) sorted order, in which case the while loop executes the maximum number of times. Thus  $t_j = j$  for  $j = 2, 3, \dots, n$ . Using Appendix A of CLRS, the summation terms can be reduced as follows:

$$\sum_{j=2}^n j = \left( \sum_{j=1}^n j \right) - 1 = \frac{n(n+1)}{2} - 1$$

and

$$\sum_{j=2}^n (j-1) = \left( \sum_{k=1}^{n-1} k \right) = \frac{(n-1)n}{2}$$

Thus the run time becomes

$$\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \left( \frac{n(n+1)}{2} - 1 \right) + c_6 \left( \frac{(n-1)n}{2} \right) + c_7 \left( \frac{(n-1)n}{2} \right) + c_8(n-1) \\
&= \left( \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + \left( c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 \right) n - (c_2 + c_4 + c_5 + c_8) \\
&= an^2 + bn + c
\end{aligned}$$

which is a *quadratic* function of  $n$ .

**Conclusion:** - Thus we have implemented the program for insertion sort algorithm. The program was successfully compiled & executed, and the output was verified. We have also analyzed the algorithm with its worst case and best case complexities.