	Marwadi University **Faculty of Engineering and Technology** **Department of Information and Communication Technology**	
Subject: DAA (01CT0512)	Aim: **Implementing the Sorting Algorithms and understanding the time and space complexities**	
Lab-1		**Enrolment No**: 92301733046

Programming Language: Python

1) Insertion Sort

- **Theory:**

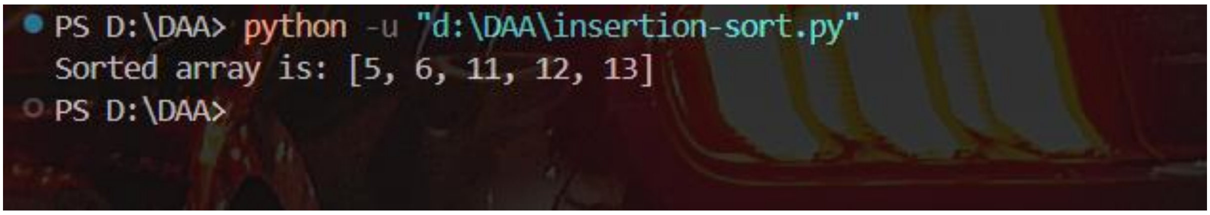
The fundamental concept behind Insertion Sort is to create a sorted list one element at a time by inserting every new element in its proper location among the already sorted elements.

- Small Data Sets
- Nearly Sorted Arrays
- Online Sorting
- Hybrid Sorting Algorithms
- Preferred in Low-Memory Environments

- **Code:**

```
import numpy as np
def insertion_sort(arr):
    n = len(arr)
    for i in range(1, n):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr
```

- **Output:**



```
PS D:\DAA> python -u "d:\DAA\insertion-sort.py"
Sorted array is: [5, 6, 11, 12, 13]
PS D:\DAA>
```

Space complexity: O(1)

Justification: Because, we have defined 3 more variables (n, I, j), which will take space.

Time complexity: O(n^2)

Best case time complexity: O(n)

Justification: Array is already sorted

Worst case time complexity: O(n^2)

Justification: Array is in descending order

2) Bubble Sort:

- **Theory:**

Bubble Sort is a basic comparison-based sorting algorithm. It operates by repeatedly swapping two adjacent elements if they are in the wrong order, moving the largest unsorted element to “bubble up” to its correct position in each pass.

- Small Inputs Sets
- Detecting Sorted Arrays
- Embedded Systems

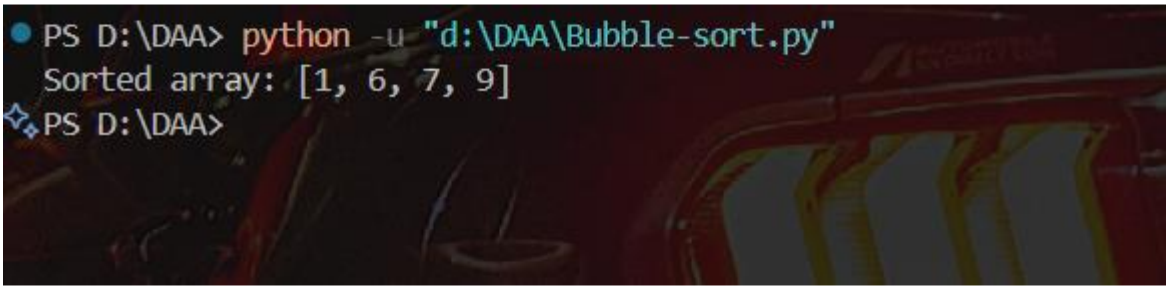
- **Code:**

```
def bubble_sort(arr):
    n = len(arr)

    for i in range(n):
        for j in range(0, n - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr

arr = [1,7,9,6]
sorted_arr = bubble_sort(arr)
print("Sorted array:", sorted_arr)
```

- **Output:**



```
PS D:\DAA> python -u "d:\DAA\Bubble-sort.py"
Sorted array: [1, 6, 7, 9]
PS D:\DAA>
```

Space complexity: O(1)

Time complexity: O(n^2)

Best case time complexity: O(n)

Justification: Array is already sorted

Worst case time complexity: O(n^2)

Justification: Array is in descending order

3) Selection Sort:

➢ Theory:

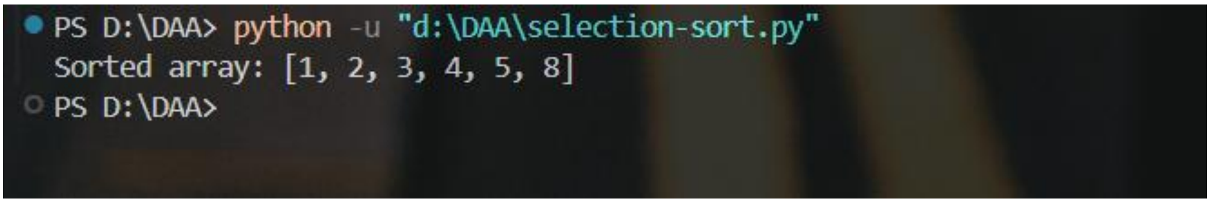
Selection Sort works by repeatedly finding the minimum element from the unsorted part of the array and moving it to the beginning.

- Small Data Sets
- Memory-Constrained Systems
- When Fewer Swaps are Needed

➢ Code:

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_index = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]
    return arr
arr = [4, 5, 3, 1, 2, 8]
sorted_arr = selection_sort(arr)
print("Sorted array:", sorted_arr)
```

➢ Output:



```
PS D:\DAA> python -u "d:\DAA\selection-sort.py"
Sorted array: [1, 2, 3, 4, 5, 8]
PS D:\DAA>
```

Space complexity: O(1)

Time complexity: O(n^2)

Best case time complexity: O(n^2)

Justification: Still scans entire unsorted array for each pass

Worst case time complexity: O(n^2)

Justification: Array is in descending order

4) Counting Sort:

➢ Theory:

Counting Sort is a non-comparison-based sorting algorithm that works efficiently for sorting integers within a known, limited range.

Counting Sort is an integer sorting algorithm that sorts elements by counting the number of occurrences of each distinct element. It does not compare elements like other sorting algorithms.

➢ Code:

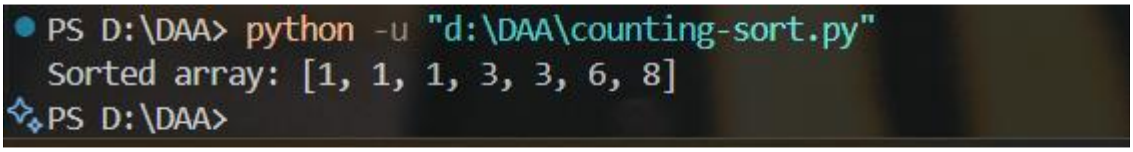
```
def counting_sort(arr):
    max_val = arr[0]
    for i in arr:
        if i > max_val:
            max_val = i

    count = [0 for _ in range(max_val + 1)]
    for i in arr:
        count[i] += 1

    sorted_arr = []
    for i in range(len(count)):
        for _ in range(count[i]):
            sorted_arr.append(i)
    return sorted_arr

arr = [6, 1, 1, 8, 3, 3, 1]
sorted_arr = counting_sort(arr)
print("Sorted array:", sorted_arr)
```

➢ **Output:**



```
PS D:\DAA> python -u "d:\DAA\counting-sort.py"
Sorted array: [1, 1, 1, 3, 3, 6, 8]
PS D:\DAA>
```

Space complexity: O(n)

Time complexity: O(n + k)

Best case time complexity: O(n + k)

Justification: All elements are within a small range, so counting and reconstructing the array is fast and linear.

Worst case time complexity: O(n + k)

Justification: If the range (k) is very large compared to the number of elements (n), time and space both increase significantly.