Marwadi University Marwadi Chandarana Group	**Marwadi University** **Faculty of Engineering and Technology** **Department of Information and Communication Technology**	
Subject: Design & Analysis of Algorithms (01CT0512)	**Aim: Implementing the Sorting Algorithms and understanding the time and space complexities**	
Experiment No: 1	**Date: 03/08/2025**	**Enrollment No: 92301733049**

Theory:

Sorting in C++ refers to the process of rearranging the elements of an array or container in a specific order, such as ascending (smallest to largest) or descending (largest to smallest). Sorting is a fundamental operation in programming, as it helps organize data for efficient searching, comparison, and analysis.

Programming Language: C++

1. Insertion Sort

Insertion Sort is a simple and intuitive sorting algorithm used to arrange elements in a specific order (ascending or descending). It works by building a sorted portion of the array one element at a time

Code:-

```cpp
#include <iostream>
using namespace std;

void insertionSort(int arr[], int n) {
    for(int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while(j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }

        arr[j + 1] = key;
    }
}
```

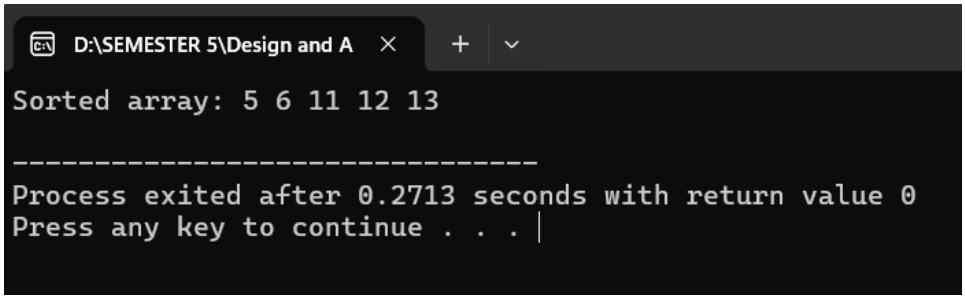
```cpp
int main() {
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr)/sizeof(arr[0]);

    insertionSort(arr, n);

    cout << "Sorted array: ";
    for(int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Output:



Space complexity: O(1)

Justification: We are not taking any extra array , we are doing the operation on the same array.

Time complexity:

– Best case time complexity: O(n)

Justification: If the array is already sorted, each element is compared only once without shifting.

– Worst case time complexity: O(n2)

Justification: If the array is in reverse order, each element is compared and shifted through all previously sorted elements.

2. Bubble Sort

Bubble Sort works by repeatedly swapping the adjacent elements if they are in the wrong order. It is often used to introduce the concept of a sorting and is particularly suitable for sorting small datasets.

Code :-

Marwadi University Marwadi Chandarana Group	**Marwadi University** **Faculty of Engineering and Technology** **Department of Information and Communication Technology**	
Subject: Design & Analysis of Algorithms (01CT0512)	**Aim: Implementing the Sorting Algorithms and understanding the time and space complexities**	
Experiment No: 1	**Date: 03/08/2025**	**Enrollment No: 92301733049**

```cpp
#include <iostream>
using namespace std;

void bubbleSort(int arr[], int n) {
    for(int i = 0; i < n - 1; i++) {
        for(int j = 0; j < n - i - 1; j++) {
            if(arr[j] > arr[j + 1]) {
                // Swap the elements
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

```cpp
int main() {
    int arr[] = {5, 1, 4, 2, 8};
    int n = sizeof(arr)/sizeof(arr[0]);

    bubbleSort(arr, n);

    cout << "Sorted array: ";
    for(int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

OUTPUT :-

```
D:\SEMESTER 5\Design and A
Sorted array: 1 2 4 5 8

--------------------------------
Process exited after 0.2635 seconds with return value 0
Press any key to continue . . .
```

Space complexity: O(1)

Justification: We are not using any extra array or data structure; all operations are done in-place on the input array.

Time complexity:

– Best case: O(n)
Justification: If the array is already sorted, the algorithm makes a single pass without any swaps. With optimization (early exit using a flag), this results in linear time.

– Worst case: $O(n^2)$
Justification: If the array is in reverse order, every element must be compared and swapped with every other element, resulting in quadratic time.

3. Selection Sort

Selection sort is a simple and intuitive sorting algorithm used for ordering arrays or lists. It works by repeatedly finding the minimum (or maximum) element from the unsorted part of the array and moving it to the beginning.

CODE :-

```cpp
#include <iostream>
using namespace std;

void selectionSort(int arr[], int n) {
    for(int i = 0; i < n - 1; i++) {
        int minIndex = i;
        for(int j = i + 1; j < n; j++) {
            if(arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        int temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }
}
```

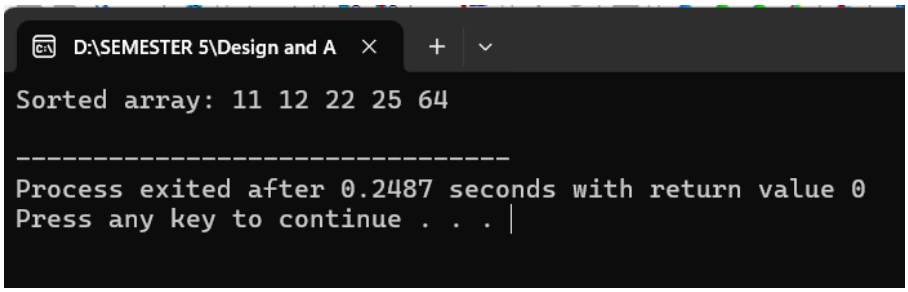
```cpp
int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);

    selectionSort(arr, n);

    cout << "Sorted array: ";
    for(int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

OUTPUT :-



Space complexity: O(1)

Justification: We are not using any extra array or data structure; all operations are done in-place on the input array.

Time complexity:

– Best case: $O(n^2)$

Justification: Selection sort always scans the unsorted part of the array for the minimum, even if the array is already sorted.

– Worst case: $O(n^2)$

Justification: It always performs (n-1) + (n-2) + ... + 1 comparisons regardless of input order, resulting in quadratic time.

4. Counting Sort

Counting Sort is a non-comparison-based sorting algorithm that is particularly efficient for sorting integers or objects with keys that are within a known, limited range. Unlike comparison-based sorts (like quicksort or merge sort), counting sort leverages the frequency of each element to determine its final position directly.

CODE :-

```cpp
#include <iostream>
#include <vector>
using namespace std;

void countingSort(int arr[], int n) {
    int maxVal = arr[0];
    for(int i = 1; i < n; i++) {
        if(arr[i] > maxVal)
            maxVal = arr[i];
    }

    vector<int> count(maxVal + 1, 0);

    for(int i = 0; i < n; i++) {
        count[arr[i]]++;
    }

    int index = 0;
    for(int i = 0; i <= maxVal; i++) {
        while(count[i] > 0) {
            arr[index++] = i;
            count[i]--;
        }
    }
}
```

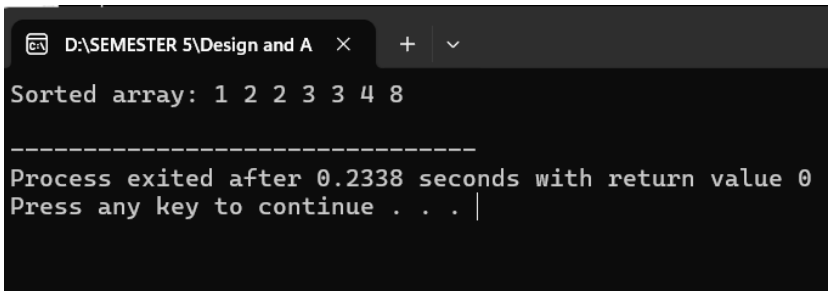
```cpp
int main() {
    int arr[] = {4, 2, 2, 8, 3, 3, 1};
    int n = sizeof(arr) / sizeof(arr[0]);

    countingSort(arr, n);

    cout << "Sorted array: ";
    for(int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;

    return 0;
}
```

OUTPUT :-



Space complexity: O(k)

Justification: We use extra space to store a count array of size k, where k is the range of input values (maximum element + 1).

Time complexity:

– Best case: O(n + k)

Justification: Counting sort goes through the original array to count elements and then builds the sorted array — both steps take linear time relative to n and k.

– Worst case: O(n + k)

Justification: Regardless of the order of input, we always count all elements and reconstruct the array from the count array, leading to linear time with respect to n and k.