Counting Sort: Introduction

- A non-comparison-based sorting algorithm.
- Uses element counts instead of comparisons.
- Runs in O(n + k) time, faster than O(n log n) algorithms for small ranges.
- Important for sorting integers, characters, or categorical data efficiently.

Counting Sort is most suitable for:

- **Small Range of Input Values**
- Works best when the maximum element (max) is not much larger than the number of elements (n), i.e., max = O(n).
- Example: sorting exam marks in the range 0–100 for thousands of students.

- **Integer or Discrete Keys**
- Only suitable for integers or data that can be mapped to integer keys (like characters).
- Cannot directly handle floating-point numbers or very large ranges efficiently.

- **Stability is Required**
- Counting Sort is stable (preserves the order of equal elements).
- Useful when sorting records based on keys, e.g., sorting students by roll number while maintaining order of names.

Steps of Counting Sort

1. Find the maximum element (k).
2. Initialize a count array of size (k+1).
3. Count each element’s frequency.
4. Compute cumulative sum (prefix sum) of counts.
5. Place each element in its correct sorted position.
6. Build and return the sorted output array.

Steps of Counting Sort (in detail)

1.Find the maximum element

- Let the maximum value in the array be max.
- This determines the size of the count array.

2. Initialize the count array

- Create an array count of size max + 1.
- Fill it with 0s initially.
- Purpose: to store the frequency of each element.

3. Store the count of each element

- For each element arr[i] in the input array, increment count[arr[i]].
- Example: if arr[i] = 3, then increase count[3] by 1.
- After this step, count[x] tells how many times x appears in the input.

Steps of Counting Sort (in detail)

4. Store the cumulative count

- Modify the count array so that each position stores the cumulative sum:

  count[i]=count[i]+count[i−1]

- This tells us the position of each element in the sorted output.

5. Place the elements into the output array

- Traverse the input array **from right to left** (to maintain stability).
- For each element arr[i]:
  - Look up count[arr[i]].
  - Place arr[i] into the correct index in the output array (index = count[arr[i]] - 1).
  - Decrease count[arr[i]] by 1.

6. Copy back the sorted array

- Copy the elements from the output array back to the input array (if needed).

Counting Sort Example

Step 1: Input Array = [4, 2, 2, 8, 3, 3, 1]

Step 2: Count Array after counting → [0,1,2,2,1,0,0,0,1]

Step 3: Cumulative Count Array → [0,1,3,5,6,6,6,6,7]

Step 4: Place elements in output array → [1,2,2,3,3,4,8]

Final Sorted Output = [1,2,2,3,3,4,8]

C++ Implementation of Counting Sort

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

    int count[maxVal+1] = {0};
    for (int i = 0; i < n; i++)
        count[arr[i]]++;

    for (int i = 1; i <= maxVal; i++)
        count[i] += count[i-1];

 int output[n];
    for (int i = n-1; i >= 0; i--) {
        output[count[arr[i]] - 1] = arr[i];
        count[arr[i]]--;
    }

 for (int i = 0; i < n; i++)
        arr[i] = output[i];
}
```

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

int main() {
    int arr[] = {4, 2, 2, 8, 3, 3, 1};
    int n = sizeof(arr)/sizeof(arr[0]);
    countingSort(arr, n);
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    return 0;
}
```

Complexity of Counting Sort

Space Complexity:

The space complexity of Counting Sort is O(max). Larger the range of elements, larger is the space complexity.

Time Complexity:

Worst Case Complexity: O(n+max)
Best Case Complexity: O(n+max)
Average Case Complexity: O(n+max)

Here, n=size of the array to be sorted, max= range of input values (maximum element + 1)
In all the above cases, the complexity is the same because no matter how the elements are placed in the array, the algorithm goes through n+max times.

There is no comparison between any elements, so it is better than comparison based sorting techniques. But, it is bad if the integers are very large because the array of that size should be made.

At a glance:

Time Complexity	
Best	O(n+max)
Worst	O(n+max)
Average	O(n+max)
Space Complexity	O(max)
Stability	Yes

Additionally:

The time complexity of Counting Sort is generally:

$$O(n+k)$$

Where:
n = number of elements in the input array
k = range of input values (maximum element + 1)

When does it become O(n)?
Counting Sort runs in O(n) time when the range of numbers k is proportional to n, i.e.,k=O(n)
This means the maximum element is not much larger than the size of the array.

Additionally:

When does it become O(n)?
Counting Sort runs in O(n) time when the range of numbers k is proportional to n, i.e.,k=O(n)
This means the maximum element is not much larger than the size of the array.

Example:

- Input: n = 1000 elements
- Range of values: 0 to 999 (k=1000k)
- Time = O(n+k)=O(1000+1000)=O(n)
- But if the range is very large, say 0 to 1,000,000, then:
- O(n+k)=O(n+1,000,000) That's closer to **O(k)**, not O(n), and becomes inefficient.

So Counting Sort is O(n) when the maximum element (range) is not much larger than the number of elements.