Counting Sort Algorithm

Counting Sort Algorithm is an efficient sorting algorithm that can be used for sorting elements within a specific range. This sorting technique is based on the frequency/count of each element to be sorted and works using the following algorithm-

- **Input**: Unsorted array A[] of n elements
- **Output**: Sorted arrayB[]

Step 1: Consider an input array A having n elements in the range of 0 to k, where n and k are positive integer numbers. These n elements have to be sorted in ascending order using the counting sort technique. Also note that A[] can have distinct or duplicate elements

Step 2: The count/frequency of each distinct element in A is computed and stored in another array, say count, of size k+1. Let u be an element in A such that its frequency is stored at count[u].

Step 3: Update the count array so that element at each index, say i, is equal to -

$$count[i]= \sum count[u] \quad \text{where } 0<= u<= i$$

Step 4: The updated count array gives the index of each element of array A in the sorted sequence. Assume that the sorted sequence is stored in an output array, say B, of size n.

Step 5: Add each element from input array A to B as follows:

a. Set i=0 and $t = A[i]$

b. Add t to B[v] such that $v = (count[t]-1)$.

c. Decrement count[t] by 1

d. Increment i by 1

Repeat steps **(a)** to **(d)** till **i = n-1**

Step 6: Display B since this is the sorted array

Pictorial Representation of Counting Sort with an Example

Let us trace the above algorithm using an example:

Consider the following input array A to be sorted. All the elements are in range 0 to 9

```
A[]= {1, 3, 2, 8, 5, 1, 5, 1, 2, 7}
```

Copy

Step 1: Initialize an auxiliary array, say count and store the frequency of every distinct element. Size of count is 10 (k+1, such that range of elements in A is 0 to k)

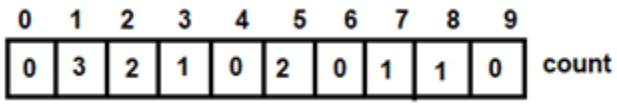

Figure 1: count array

Step 2: Using the formula, updated count array is -

$$count[i]= \sum count[u] \quad \text{where } 0<= u<= i$$

Figure 2: Formula for updating count array

0	1	2	3	4	5	6	7	8	9	
0	3	5	6	6	8	8	9	10	10	count

Figure 3 : Updated count array

Step 3: Add elements of array A to resultant array B using the following steps:

- For, i=0, t=1, count[1]=3, v=2. After adding 1 to B[2], count[1]=2 and i=1

0	1	2	3	4	5	6	7	8	9	
0	0	1	0	0	0	0	0	0	0	B
0	2	5	6	6	8	8	9	10	10	count

Figure 4: For i=0

- For i=1, t=3, count[3]=6, v=5. After adding 3 to B[5], count[3]=5 and i=2

0	1	2	3	4	5	6	7	8	9	
0	0	1	0	0	3	0	0	0	0	B
0	2	5	5	6	8	8	9	10	10	count

Figure 5: For i=1

- For i=2, t=2, count[2]= 5, v=4. After adding 2 to B[4], count[2]=4 and i=3

0	1	2	3	4	5	6	7	8	9	
0	0	1	0	2	3	0	0	0	0	B
0	2	4	5	6	8	8	9	10	10	count

Figure 6: For i=2

- For i=3, t=8, count[8]= 10, v=9. After adding 8 to B[9], count[8]=9 and i=4

0	1	2	3	4	5	6	7	8	9	
0	0	1	0	2	3	0	0	0	8	B
0	2	4	5	6	8	8	9	9	10	count

Figure 7: For i=3

- On similar lines, we have the following:

0	1	2	3	4	5	6	7	8	9	
0	0	1	0	2	3	0	5	0	8	B
0	2	4	5	6	7	8	9	9	10	count

Figure 8: For i=4

0	1	2	3	4	5	6	7	8	9	
0	1	1	0	2	3	0	5	0	8	B
0	1	4	5	6	7	8	9	9	10	count

Figure 9: For i=5

0	1	2	3	4	5	6	7	8	9	
0	1	1	0	2	3	5	5	0	8	B
0	1	4	5	6	6	8	9	9	10	count

Figure 10: For i=6

0	1	2	3	4	5	6	7	8	9	
1	1	1	0	2	3	5	5	0	8	B
0	0	4	5	6	6	8	9	9	10	count

Figure 11: For i=7

0	1	2	3	4	5	6	7	8	9	
1	1	1	2	2	3	5	5	0	8	B
0	0	3	5	6	6	8	9	9	10	count

Figure 12: For i=8

Figure 13: For i=9

Thus, array B has the sorted list of elements.

Program for Counting Sort Algorithm

Below we have a simple program in C++ implementing the counting sort algorithm:

```
#include<iostream>

using namespace std;



int k=0;  // for storing the maximum element of input array



/* Method to sort the array */

void sort_func(int A[],int B[],int n)

{

     int count[k+1],t;

     for(int i=0;i<=k;i++)

     {

          //Initialize array count

          count[i] = 0;

     }

     for(int i=0;i<n;i++)

     {

         // count the occurrence of elements u of A

         // & increment count[u] where u=A[i]

          t = A[i];

          count[t]++;

     }

     for(int i=1;i<=k;i++)

     {
```

```
        // Updating elements of count array
        count[i] = count[i]+count[i-1];
    }
    for(int i=0;i<n;i++)
    {
        // Add elements of array A to array B
        t = A[i];
        B[count[t]] = t;
        // Decrement the count value by 1
        count[t]=count[t]-1;
        }
}
int main()
{
    int n;
    cout<<"Enter the size of the array :";
    cin>>n;

    // A is the input array and will store elements entered by the
user
    // B is the output array having the sorted elements
    int A[n],B[n];
    cout<<"Enter the array elements: ";
    for(int i=0;i<n;i++)
    {
        cin>>A[i];
        if(A[i]>k)
        {
            // k will have the maximum element of A[]
            k = A[i];
        }
```

```
    }


    sort_func(A,B,n);


    // Printing the elements of array B
    for(int i=1;i<=n;i++)
    {
         cout<<B[i]<<" ";
    }

         cout<<"\n";
         return 0;
}
```

Copy

The input array is the same as that used in the example:

```
Enter the size of the array :10
Enter the array elements: 1 3 2 8 5 1 5 1 2 7
1 1 1 2 2 3 5 5 7 8


...Program finished with exit code 0
Press ENTER to exit console.
```

Figure 14: Output of Program

Note: The algorithm can be mapped to any programming language as per the requirement.

Time Complexity Analysis

For scanning the input array elements, the loop iterates n times, thus taking $O(n)$ running time. The sorted array B[] also gets computed in n iterations, thus requiring O(n) running time. The count array also uses k iterations, thus has a running time of O(k). Thus the total running time for counting sort algorithm is $O(n+k)$.

Key Points:

- The above implementation of Counting Sort can also be extended to sort negative input numbers
- Since counting sort is suitable for sorting numbers that belong to a well-defined, finite and small range, it can be used asa subprogram in other sorting algorithms like radix sort which can be used for sorting numbers having a large range
- Counting Sort algorithm is efficient if the range of input data (k) is not much greater than the number of elements in the input array (n). It will not work if we have 5 elements to sort in the range of 0 to 10,000

- It is an integer-based sorting algorithm unlike others which are usually comparison-based. A comparison-based sorting algorithm sorts numbers only by comparing pairs of numbers. Few examples of comparison based sorting algorithms are **quick sort, merge sort, bubble sort, selection sort, heap sort, insertion sort**, whereas algorithms like radix sort, bucket sort and comparison sort fall into the category of non-comparison based sorting algorithms.

Advantages of Counting Sort:

- It is quite fast
- It is a stable algorithm

Note: For a sorting algorithm to be stable, the order of elements with equal keys (values) in the sorted array should be the same as that of the input array.

Disadvantages of Counting Sort:

- It is not suitable for sorting large data sets
- It is not suitable for sorting string values
