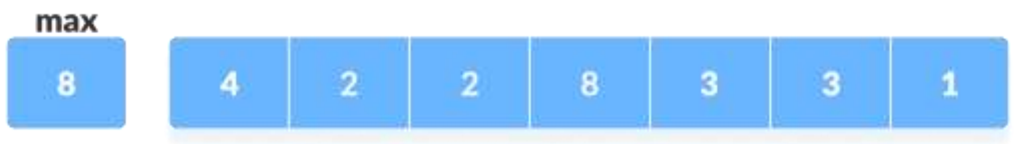
Counting Sort Algorithm

Counting sort is a sorting algorithm that sorts the elements of an array by counting the number of occurrences of each unique element in the array. The count is stored in an auxiliary array and the sorting is done by mapping the count as an index of the auxiliary array.

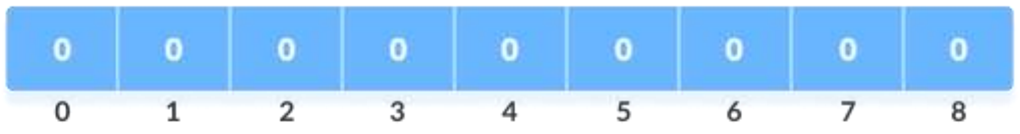
Working of Counting Sort

1. Find out the maximum element (let it be max) from the given array.



Given array

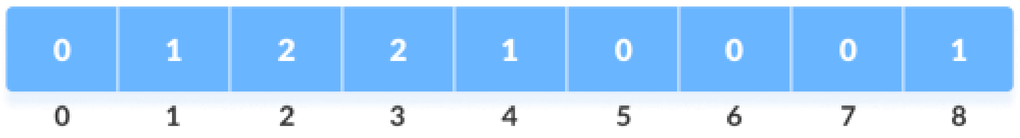
2. Initialize an array of length max+1 with all elements 0. This array is used for storing the count of the elements in the array.



Count array

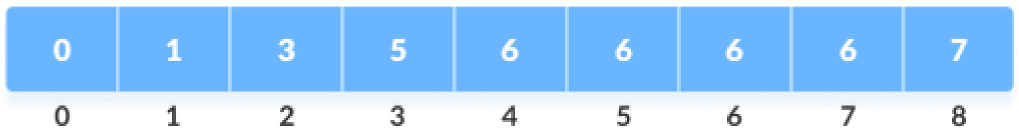
3. Store the count of each element at their respective index in count array.

   For example: if the count of element 3 is 2 then, 2 is stored in the 3rd position of count array. If element "5" is not present in the array, then 0 is stored in 5th position.



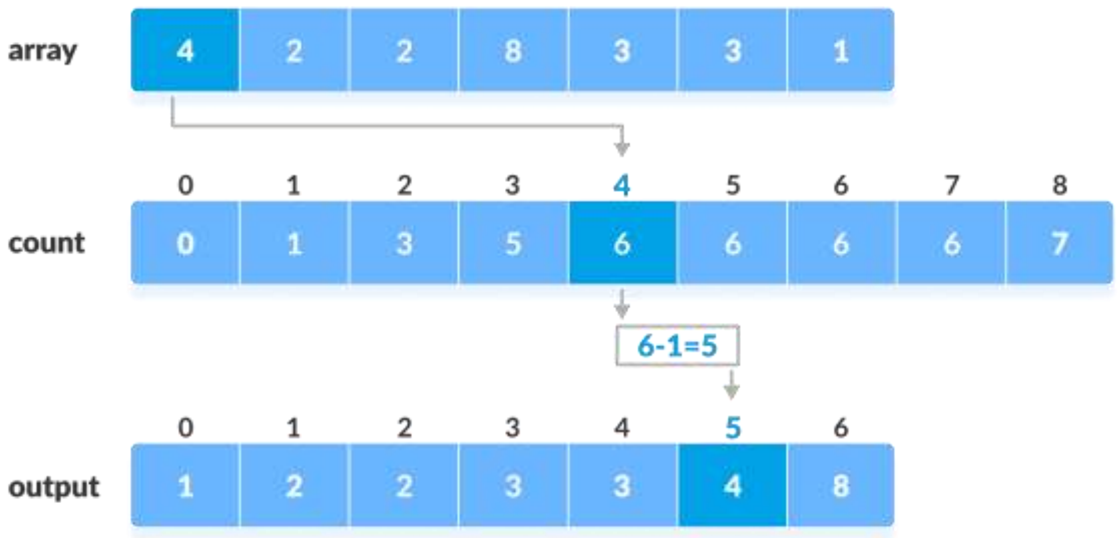
Count of each element stored

4. Store cumulative sum of the elements of the count array. It helps in placing the elements into the correct index of the sorted array.



Cumulative count

5. Find the index of each element of the original array in the count array. This gives the cumulative count. Place the element at the index calculated as shown in figure below.



Counting sort

6. After placing each element at its correct position, decrease its count by one.

Example

6,5,4,8,6,9,5,3,4,6,9

Counting Sort Algorithm

```
countingSort(array, size)
 max <- find largest element in array
 initialize count array with all zeros
 for j <- 0 to size
  find the total count of each unique element and
  store the count at jth index in count array
 for i <- 1 to max
  find the cumulative sum and store it in count array itself
 for j <- size down to 1
  restore the elements to array
```

decrease count of each element restored by 1

Time Complexity

Step	Description	Time
1. Find the max value k	Scan through array of size n	**O(n)**
2. Initialize count array	Create array of size k+1	**O(k)**
3. Count elements	Scan array again to fill counts	**O(n)**
4. Compute cumulative counts	Iterate through count array	**O(k)**
5. Build output array	Place elements in correct position	**O(n)**

Total Time Complexity = O(n + k)

Space Complexity

Counting Sort uses:

- count[] array of size k+1
- output[] array of size n

Total Space Complexity = O(n + k)

Summary Table

Case	Time Complexity	Space Complexity	Stable?
Best	O(n + k)	O(n + k)	✅ Yes
Average	O(n + k)	O(n + k)	✅ Yes
Worst	O(n + k)	O(n + k)	✅ Yes

When Counting Sort is Efficient

- When the **range of input (k) is not much larger than n** (i.e., k = O(n)).
- Works best for **integers or categorical data**.

When It's Inefficient

- When k (range of values) is **very large compared to n**, it becomes memory-inefficient.

- Not suitable for sorting floating-point or string data (unless mapped to integers).