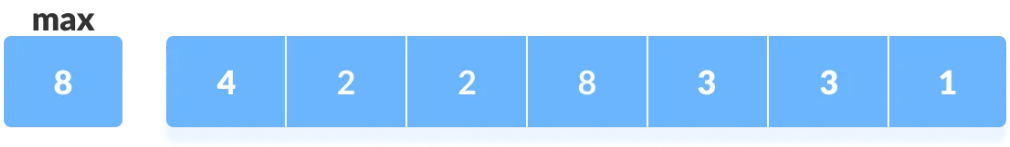
Counting Sort Algorithm

In this tutorial, you will learn about the counting sort algorithm and its implementation in Python, Java, C, and C++.

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.

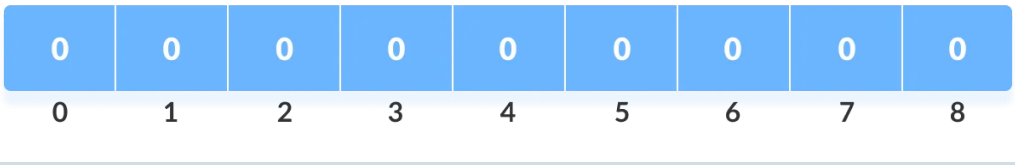
Working of Counting Sort

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



Given array

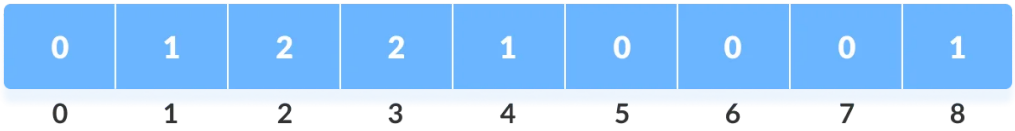
2. Initialize an array of length `max+1` with all elements 0. This array is used for storing the count of the elements in the array.



Count array

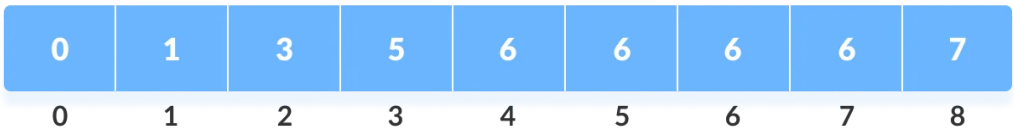
3. Store the count of each element at their respective index in `count` array

   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.



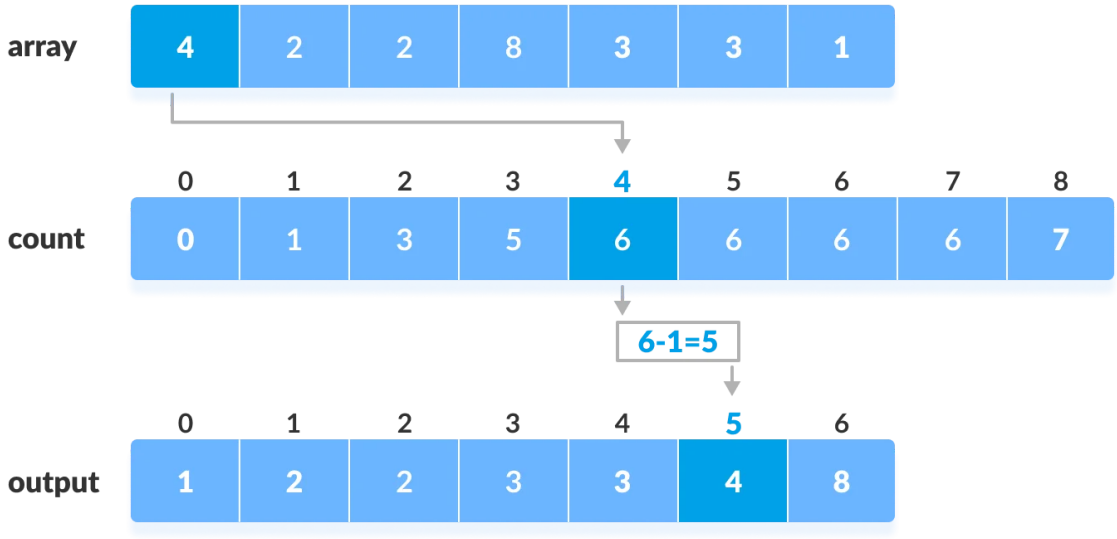
Count of each element stored

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.

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
```

Counting Sort Code in Python, Java, and C/C++

Python Java C C++

```c
// Counting sort in C programming

#include <stdio.h>

void countingSort(int array[], int size) {
  int output[10];

  // Find the largest element of the array
  int max = array[0];
  for (int i = 1; i < size; i++) {
    if (array[i] > max)
      max = array[i];
  }

  // The size of count must be at least (max+1) but
  // we cannot declare it as int count(max+1) in C as
  // it does not support dynamic memory allocation.
  // So, its size is provided statically.
  int count[10];

  // Initialize count array with all zeros.
  for (int i = 0; i <= max; ++i) {
    count[i] = 0;
  }

  // Store the count of each element
  for (int i = 0; i < size; i++) {
    count[array[i]]++;
```

Complexity

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

Time Complexities

There are mainly four main loops. (Finding the greatest value can be done outside the function.)

for-loop	time of counting
1st	O(max)
2nd	O(size)
3rd	O(max)
4th	O(size)

Overall complexity = `O(max)+O(size)+O(max)+O(size)` = `O(max+size)`

- **Worst Case Complexity:** `O(n+k)`
- **Best Case Complexity:** `O(n+k)`
- **Average Case Complexity:** `O(n+k)`

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+k` 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.

Space Complexity

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

Counting Sort Applications

Counting sort is used when:

- there are smaller integers with multiple counts.
- linear complexity is the need.