Counting Sort

Counting sort is a sorting algorithm that sorts the elements of the array by obtaining the frequencies of each unique element in the array. The frequency of each such unique element of the array is maintained with the help of an auxiliary array and sorting is done by mapping the count(frequency) as an index of the auxiliary array.

Algorithm:
Let us say, we need to sort an array A[] of size N, we define the auxiliary array Aux[] of size max(A[i]) where 0<=i<=N-1 i.e the size of the auxiliary array should be equal to the - maximum element of the array A[]+1.

- Initialize the auxiliary array Aux[] as 0 as it is responsible for keeping the count of the elements of the array.
- Now traverse the array A[], and store the count of occurrence of the elements of the array at the appropriate index of the auxiliary array Aux[], in other words, increment Aux[A[i]] by 1, for each i from 0 to N-1.
- Now store the cumulative sum of the elements of the Aux[], i.e Aux[i] = Aux[i] + Aux[i-1] for each i from 1 to size(Aux[]) -1. This will help in determining the index of the element of the given array A[] when sorted.
  The above approach works because we are basically keeping track of the positions of the element represented by the index of the auxiliary array Aux[] will be present when A[] is sorted.
  For example:
  Let A[] = [1, 3, 1, 3, 2]

  Defining an auxiliary array Aux[] of size 4.
  Aux[] = [0, 2, 1, 2]
  Index 0 represents the number of elements in the array having value 0.
  Index 1 represents the number of elements in the array having value 1.
  Index 2 represents the number of elements in the array having value 2.
  Index 3 represents the number of elements in the array having value 3.

  Doing a cumulative sum(prefix sum) of auxiliary array Aux[]:
  Aux[] = [0, 2, 3, 5]
  Now as we iterate over the array A[] again, as we encounter an element A[i], the auxiliary array tells the number of A[i]'s remaining to be explored and because of the prefix sum elements at lower indices will always be present before the elements at higher indices
- Now, iterate over the array A[] again, and for an element, A[i], obtain the index of the element from Aux[A[i]] and place the element A[i] in the output array at index Aux[A[i]] and decrement Aux[A[i]] by 1. The resulting output array will be the required sorted array.
- **Note:**

- The limitation of the algorithm is that it is not a comparison-based sorting algorithm, with space proportional to the maximum element of the array. So if the maximum element of the array is quite large, then counting sort may not work because of memory limits.
- Counting sort is efficient if the range of input data is not much larger than the number of elements in the array.
  For example, The number of elements to be sorted are 5 but lie within the range of 0 to 125, so this essentially makes our algorithm to be of the order of 5^3. This performance can become even worse if the order of difference between the number of elements to be sorted and their range is high.
- Counting sort can be extended to work for negative numbers too.

Pseudocode:

```
/*
    array of size N from 0 to N-1 is considered
*/
function countingSort(arr, N)

        /*
                Obtaining the max element of the array arr to get the size of the
                auxiliary array.
        */
        aux_size = 0
        for idx = 0 to N-1
                if aux_size < arr[i]
                        aux_size = arr[i]
        /*
                Declaring an auxiliary array of size aux_size and an output array of
                size N
        */
        aux[aux_size+1]
        output[N]

        //  Initializing aux array to 0
        for idx = 0 to aux_size
                aux[idx] = 0

        //  Storing the frequencies of the elements of the array arr
        for idx = 0 to N-1
                aux[arr[i]] = aux[arr[i]]+1

        //  Cumulative sum (prefix sum) on auxiliary array
        for idx = 1 to aux_size
                aux[idx] = aux[idx] + aux[idx-1]

        //   Building the output array
        for idx = 0 to N-1
                output[aux[arr[idx]] - 1] = arr[idx]
                aux[arr[idx]] = aux[arr[idx]] - 1

        return output
```

Time complexity: O(N+K), in the worst case.
Traversing the given array of size N and an auxiliary array of size K i.e the range of the input.

Space complexity: O(N+K)
We need an extra auxiliary array of size K (range of input) and an output array of size N.