Counting Sort Algorithm

Counting Sort is an integer sorting algorithm that works by counting the number of occurrences of each distinct value in the input array. It uses this information to place the elements in the correct sorted order.

Algorithm:

Input:

- An array A[]A[]A[] with nnn integers to be sorted.
- The maximum value in the array, kkk, where all elements of A[]A[]A[] are in the range [0,k][0, k][0,k].

Steps:

Initialize a Counting Array:

- Create an auxiliary array count[] of size k+1k+1k+1 to store the frequency of each element in A[]A[]A[].
- Initialize all elements of count[] to 0.

Count the Occurrences:

- Iterate through the array A[]A[]A[], and for each element xxx in A[]A[]A[], increment count[x] by 1.

Cumulative Count:

- Modify the count[] array by taking cumulative sums. This step will ensure that each position in count[] will represent the number of elements less than or equal to the current index.

Sort the Array:

- Create a result array B[]B[]B[] (of the same size as A[]A[]A[]).
- Traverse the array A[]A[]A[] in reverse order and place each element in its correct position in the result array B[]B[]B[] based on the counts in count[].
- Decrease the count for each element placed in B[]B[]B[].

Copy the Sorted Elements:

- Copy the sorted elements from B[]B[]B[] back to A[]A[]A[].

Output:

- The sorted array A[]A[]A[].

Counting Sort Algorithm (Pseudocode):

```
plaintext

Copy code

CountingSort(A, n, k)
  // A is the input array, n is the size of A, k is the maximum value in A
  Initialize count[0..k] to 0   // Initialize count array to store frequencies
  Initialize output[0..n-1] to 0  // Create output array to store sorted values

  // Step 1: Count the frequency of each element in A[]
  for i = 0 to n - 1
    count[A[i]] = count[A[i]] + 1

  // Step 2: Update count array to hold cumulative sum
  for i = 1 to k
    count[i] = count[i] + count[i - 1]

  // Step 3: Build the output array by placing elements at the correct position
  for i = n - 1 down to 0
    output[count[A[i]] - 1] = A[i]
    count[A[i]] = count[A[i]] - 1

  // Step 4: Copy the sorted elements from output[] to A[]
  for i = 0 to n - 1
    A[i] = output[i]
```

Part (b): Analyze in Detail Your Written Algorithms

Time Complexity Analysis:

Counting the Occurrences:

- We iterate through the input array A[]A[]A[] once, and for each element A[i]A[i]A[i], we increment the corresponding index in the count[] array.

- This step takes O(n)O(n)O(n) time, where nnn is the number of elements in the input array.

Cumulative Count:

- We iterate through the count[] array of size k+1k+1k+1, where kkk is the maximum value in the input array.
- This step takes O(k)O(k)O(k) time.

Placing Elements in the Output Array:

- We traverse the input array A[]A[]A[] once again in reverse order to place each element in its sorted position in the output array B[]B[]B[].
- This step also takes O(n)O(n)O(n) time.

Copying Sorted Elements Back to A[]A[]A[]:

- We copy the elements from the output array B[]B[]B[] back into the original array A[]A[]A[].
- This step takes O(n)O(n)O(n) time.

Overall Time Complexity:

The total time complexity of the Counting Sort algorithm is:

$T(n)=O(n)+O(k)+O(n)+O(n)=O(n+k)T(n) = O(n) + O(k) + O(n) + O(n) = O(n + k)T(n)=O(n)+O(k)+O(n)+O(n)=O(n+k)$

So, the **time complexity** of Counting Sort is **O(n + k)**, where:

- nnn is the number of elements in the input array.
- kkk is the range of the input values (i.e., the maximum value).

Space Complexity:

- The algorithm requires extra space for the count[] array, which has size k+1k+1k+1.
- Additionally, the output[] array has size nnn to store the sorted elements.

Thus, the **space complexity** is:

$S(n)=O(n+k)S(n) = O(n + k)S(n)=O(n+k)$

Advantages of Counting Sort:

- **Efficient for small ranges of integers**: When kkk is small relative to nnn, Counting Sort is very efficient.
- **Stable Sort**: Counting Sort maintains the relative order of equal elements in the input array.

- **Non-Comparative Sort**: It does not compare elements directly, unlike other algorithms like Merge Sort or Quick Sort.

Disadvantages of Counting Sort:

- **Large Range of Values**: If kkk is large (the range of input values is large), Counting Sort becomes inefficient due to the high space and time complexities.
- **Not suitable for non-integer data**: Counting Sort only works for non-negative integers (or values that can be mapped to non-negative integers).

Part (c): Implement Your Written Algorithms in Part (a)

Below is the Python implementation of the Counting Sort algorithm based on the pseudocode provided in Part (a):

python

Copy code

```
def counting_sort(A):
    n = len(A)
    if n == 0:
        return A

    # Step 1: Find the maximum value in the array
    k = max(A)

    # Step 2: Initialize count array and output array
    count = [0] * (k + 1)
    output = [0] * n

    # Step 3: Count the occurrences of each number in the input array
    for num in A:
        count[num] += 1

    # Step 4: Modify count array to store the cumulative sum
```

```
    for i in range(1, len(count)):
        count[i] += count[i - 1]

    # Step 5: Build the output array by placing elements at correct positions
    for i in range(n - 1, -1, -1):
        output[count[A[i]] - 1] = A[i]
        count[A[i]] -= 1

    # Step 6: Copy the sorted elements from output[] back to A[]
    for i in range(n):
        A[i] = output[i]

    return A

# Example usage
A = [4, 2, 2, 8, 3, 3, 1]
sorted_A = counting_sort(A)
print("Sorted Array:", sorted_A)
```

Explanation of the Implementation:

1. **Input Array (A)**: The input array is provided to the function.
2. **Find Maximum Value (k)**: The maximum value in the array is found to determine the size of the counting array.
3. **Count Occurrences**: The counting array is populated based on the frequency of each element in the input array.
4. **Cumulative Count**: The cumulative sum is calculated in the count[] array.
5. **Place Elements in Output Array**: The elements are placed in their correct positions in the output array.
6. **Copy Sorted Array Back**: The sorted values are copied back into the original array.

Example Output:

plaintext

Copy code

Sorted Array: [1, 2, 2, 3, 3, 4, 8]