C22 - LAB ASSIGNMENT. Counting sort

Counting Sort is a non-comparative sorting algorithm that works well when the range of input values is not significantly greater than the number of items to be sorted. It operates by counting the number of occurrences of each unique element in the input array and then uses these counts to determine the sorted position of each element.

Here's how it works step by step:

1. **Find the Range:** Determine the maximum value (max) in the input array. This helps in creating a counting array of the appropriate size.
2. **Create a Counting Array:** Initialize an array (let's call it count) of size max + 1 with all elements set to zero. This array will store the frequency of each element in the input array.
3. **Count Occurrences:** Iterate through the input array. For each element, increment the count at the index corresponding to that element's value in the count array. For example, if the input array contains the number 3, you would increment count[3].
4. **Create a Cumulative Count Array:** Modify the count array so that each element at index i stores the sum of counts of all elements less than or equal to i. This cumulative count tells us the final sorted position of elements. To do this, iterate through the count array starting from the second element (index 1) and update each element by adding it to the value of the previous element (count[i] = count[i] + count[i-1]).
5. **Create the Output Array:** Initialize an output array of the same size as the input array.
6. **Place Elements in Sorted Order:** Iterate through the input array in reverse order. For each element in the input array:
    - Find its cumulative count in the count array. Let's say the element is x, and count[x] is c. This means that x should be placed at the c-1 index in the output array (because array indices are zero-based).
    - Place the element x at output[count[x] - 1].
    - Decrement count[x] by 1. This is crucial for handling duplicate elements correctly, ensuring that they are placed in the correct consecutive positions in the output array.

7. **Copy to Original Array:** Finally, copy the elements from the output array back to the original input array.

Example: Let's sort the array [4, 2, 2, 8, 3, 3, 1] using Counting Sort:

1. **Find the Range:** The maximum value is 8.
2. **Create a Counting Array:** count = [0, 0, 0, 0, 0, 0, 0, 0, 0] (size 8 + 1)
3. **Count Occurrences:**
    - count[4] becomes 1
    - count[2] becomes 1
    - count[2] becomes 2
    - count[8] becomes 1
    - count[3] becomes 1
    - count[3] becomes 2
    - count[1] becomes 1
    - count is now [0, 1, 2, 2, 1, 0, 0, 0, 1]
4. **Create a Cumulative Count Array:**
    - count[1] = count[1] + count[0] = 1 + 0 = 1
    - count[2] = count[2] + count[1] = 2 + 1 = 3
    - count[3] = count[3] + count[2] = 2 + 3 = 5
    - count[4] = count[4] + count[3] = 1 + 5 = 6
    - count[5] = count[5] + count[4] = 0 + 6 = 6
    - count[6] = count[6] + count[5] = 0 + 6 = 6
    - count[7] = count[7] + count[6] = 0 + 6 = 6
    - count[8] = count[8] + count[7] = 1 + 6 = 7
    - count is now [0, 1, 3, 5, 6, 6, 6, 6, 7]
5. **Create the Output Array:** output = [0, 0, 0, 0, 0, 0, 0]
6. **Place Elements in Sorted Order:**
    - Input: 1, count[1] is 1, output[1-1] = output[0] = 1, count[1] becomes 0
    - Input: 3, count[3] is 5, output[5-1] = output[4] = 3, count[3] becomes 4
    - Input: 3, count[3] is 4, output[4-1] = output[3] = 3, count[3] becomes 3
    - Input: 8, count[8] is 7, output[7-1] = output[6] = 8, count[8] becomes 6
    - Input: 2, count[2] is 3, output[3-1] = output[2] = 2, count[2] becomes 2
    - Input: 2, count[2] is 2, output[2-1] = output[1] = 2, count[2] becomes 1
    - Input: 4, count[4] is 6, output[6-1] = output[5] = 4, count[4] becomes 5
    - output is now [1, 2, 2, 3, 3, 4, 8]

7. **Copy to Original Array:** The original array becomes [1, 2, 2, 3, 3, 4, 8].

Key Characteristics of Counting Sort:

- **Non-comparative:** It doesn't compare elements to sort them.
- **Stable:** It preserves the relative order of elements with equal values (due to the reverse iteration in step 6).
- **Efficiency:** Its time complexity is linear, O(n + k), where n is the number of elements in the input array and k is the range of input values (max - min + 1). If k is significantly larger than n, the performance degrades.
- **Space Complexity:** It requires extra space for the count array (O(k)) and the output array (O(n)).
- **Limited Range:** It is most effective when the range of input values is relatively small compared to the number of elements.