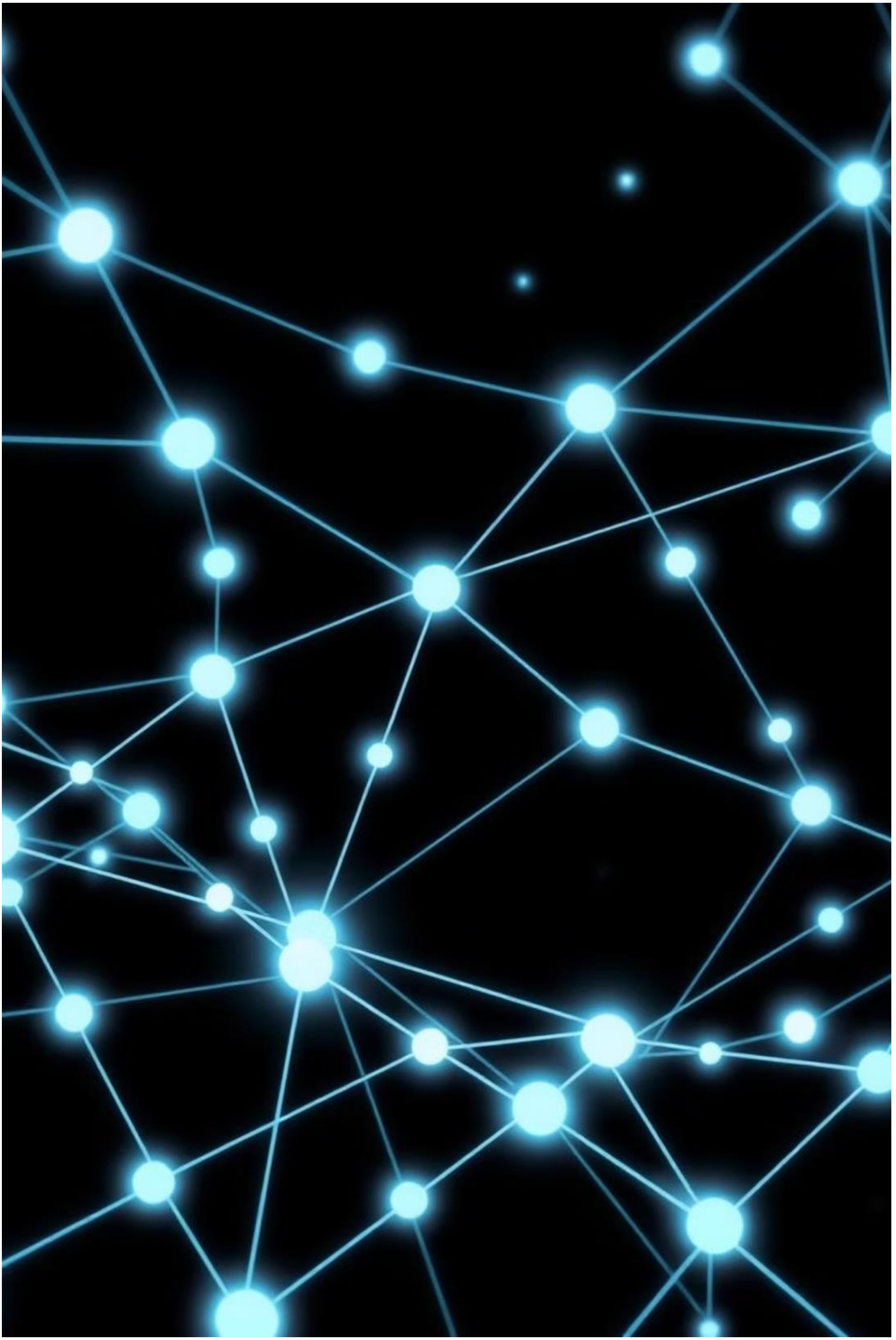


Data Structures

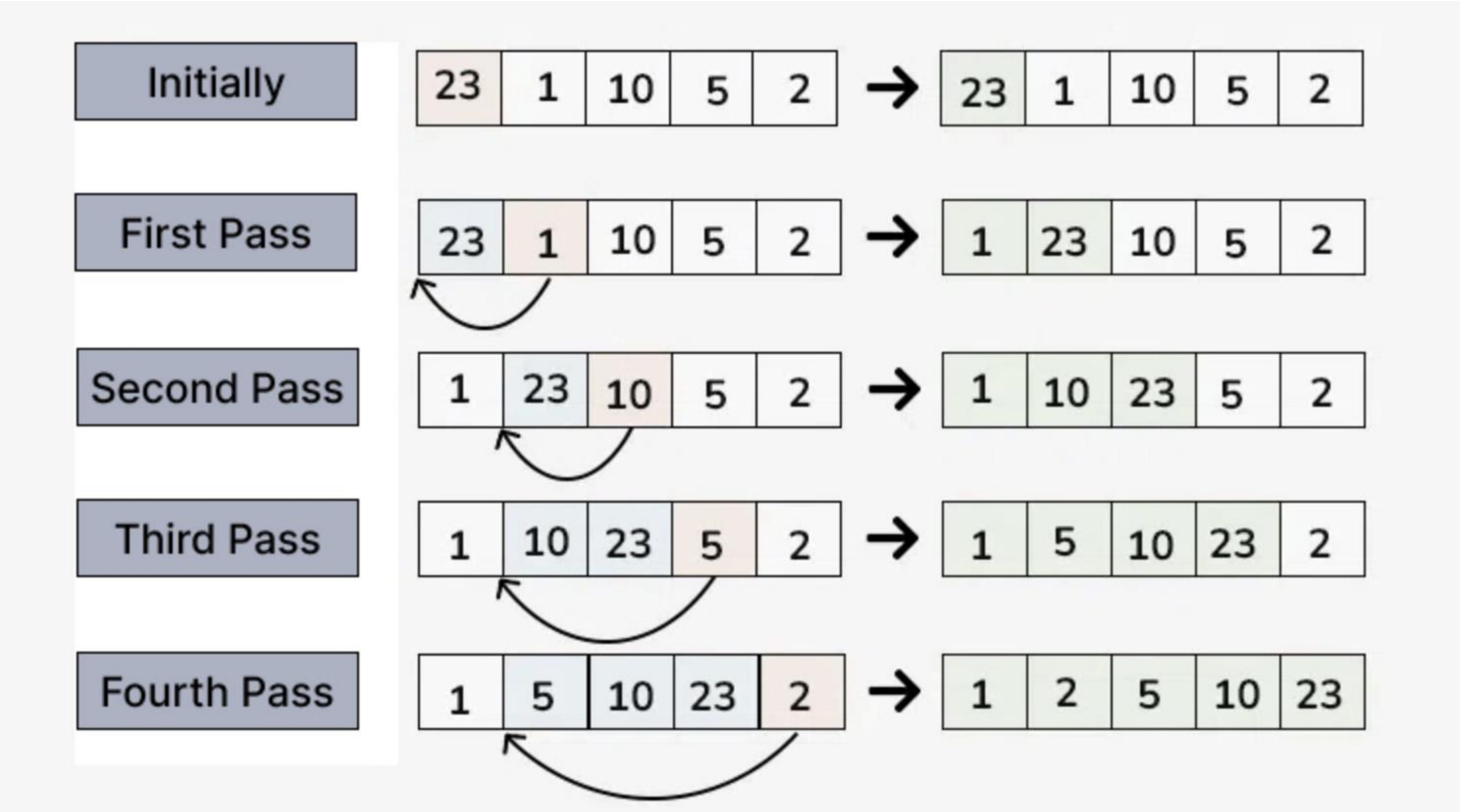
Lecture 8: Sorting

Instructor:
Md Samsuddoha
Assistant Professor
Dept of CSE, BU

Contents

Insertion Sort

- **Insertion sort** is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list.
- Process of Sorting:
  - We start with the ***second element*** of the array as the first element is assumed to be sorted.
  - Compare the second element with the first element if the second element is smaller then swap them.
  - Move to the third element, compare it with the first two elements, and put it in its correct position
  - Repeat until the entire array is sorted.


Initially
23
1
10
5
2
23
1
10
5
2
First Pass
23
1
10
5
2
1
23
10
5
2
Second Pass
1
23
10
5
2
1
10
23
5
2
Third Pass
1
10
23
5
2
1
5
10
23
2
Fourth Pass
1
5
10
23
2
1
2
5
10
23

Complexity

- Worst case: array is in **reverse order**.
- Number of Iterations: N-1
- Number of comparisons in each pass:
  - 1st pass → 1 comparison
  - 2nd pass → 2 comparisons
  - …
  - Last pass → n-1 comparisons
- Total comparisons = 1+2+3 …..+(n-1) = n(n-1)/2
- **Worst-case time complexity:** $O(n^2)$

Pseudocode

```
InsertionSort(A, n):
    for i from 1 to n-1:
        key = A[i]
        j = i - 1

        while j >= 0 and A[j] > key:
            A[j + 1] = A[j]
            j = j - 1

        A[j + 1] = key
```

Coding

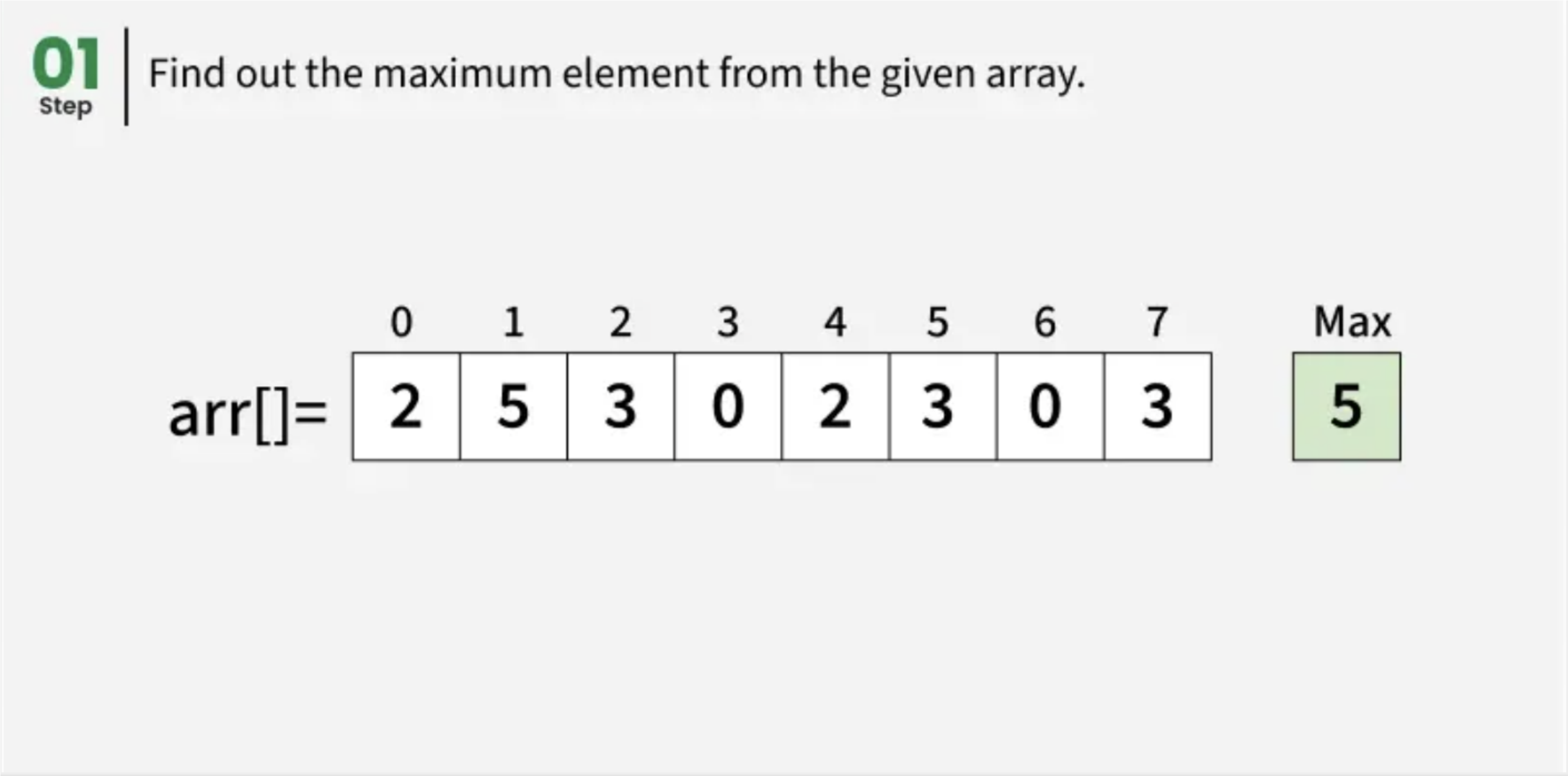
- Write a C program for Insertion sort.

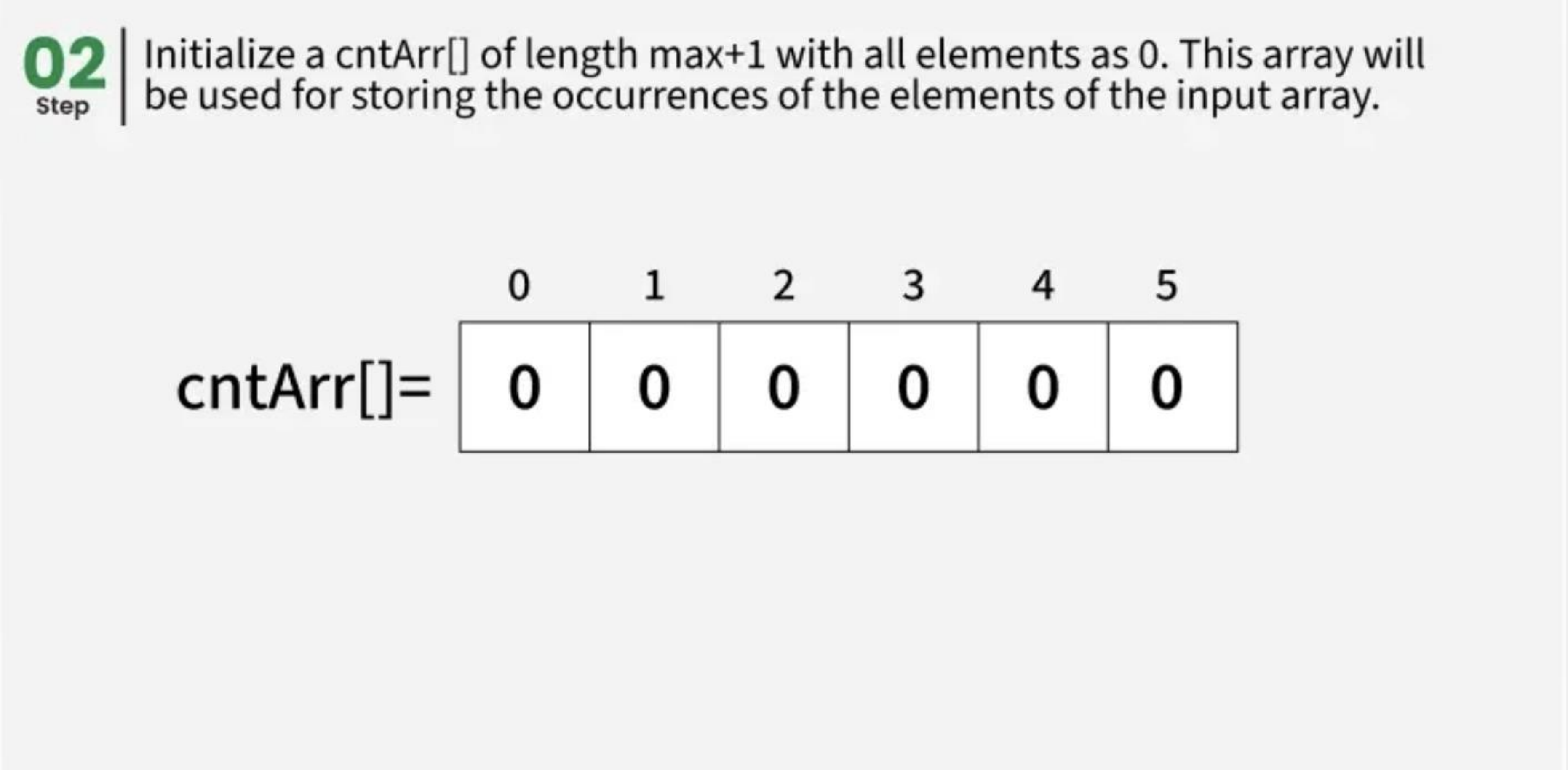
Counting Sort

- Counting Sort is a ***non-comparison-based*** sorting algorithm.
- Instead of comparing elements (like ***bubble, merge, quick***), Counting Sort counts how many times each value appears.
- The basic idea behind Counting Sort is to count the **frequency** of each distinct element in the input array and use that information to place the elements in their correct sorted positions.
- It is super fast when:
  - The range of values is small
  - Data contains integers or integer-like values (grades, ages, IDs, frequencies)

Counting Sort Algorithm

- Declare a count array **cntArr[]** of size **max(arr[])+1** and initialize it with **0**.
- Traverse input array **arr[]** and map each element of **arr[]** as an index of **cntArr[]** array, i.e., execute **cntArr[arr[i]]++** for **0 <= i < N**.
- Calculate the prefix sum at every index of **cntArr[]**.
- Create an array **ans[]** of size **N**.
- Traverse array **arr[]** from end and update **ans[cntArr[arr[i]] - 1] = arr[i]**. Also, update cntArr**[arr[i]] = cntArr[arr[i]]- -** .


01
Step
Find out the maximum element from the given array.
0 1 2 3 4 5 6 7
arr[]= 2 5 3 0 2 3 0 3
Max
5


02
Step
Initialize a cntArr[] of length max+1 with all elements as 0. This array will be used for storing the occurrences of the elements of the input array.
0 1 2 3 4 5
cntArr[]=
0 0 0 0 0 0

03 Step

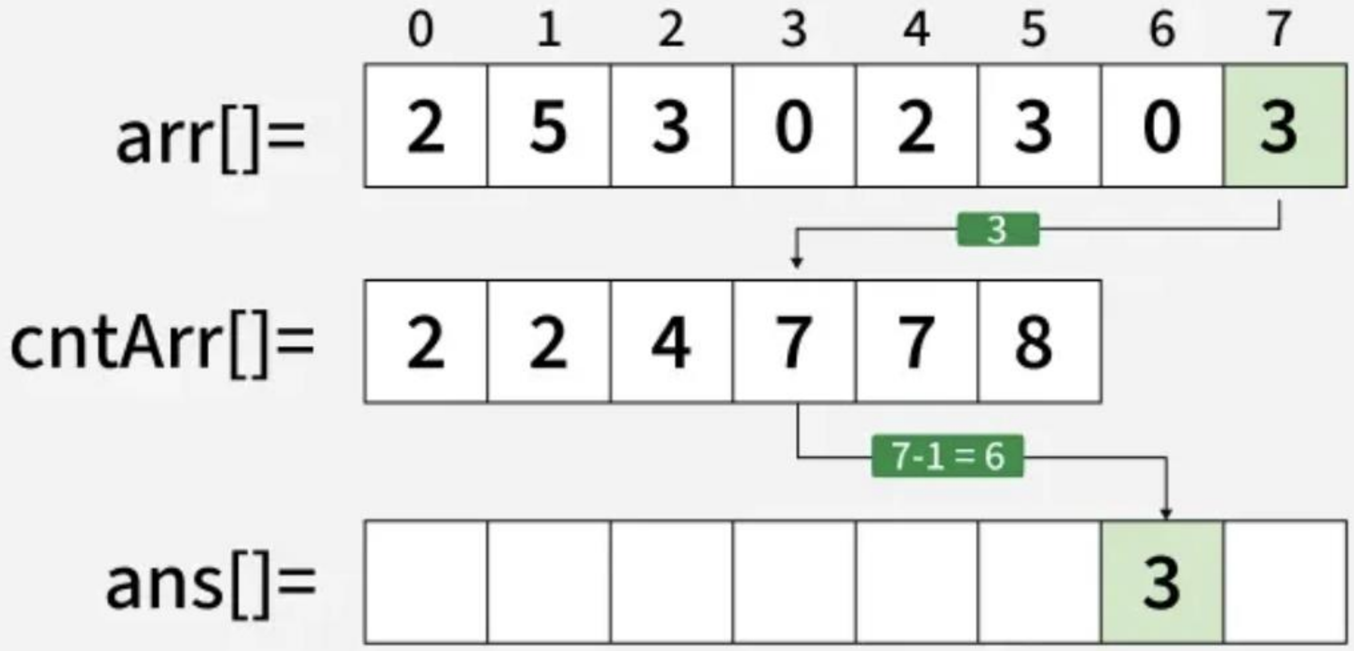
In the cntArr[], store the count of each unique element of the input array at their respective indices.

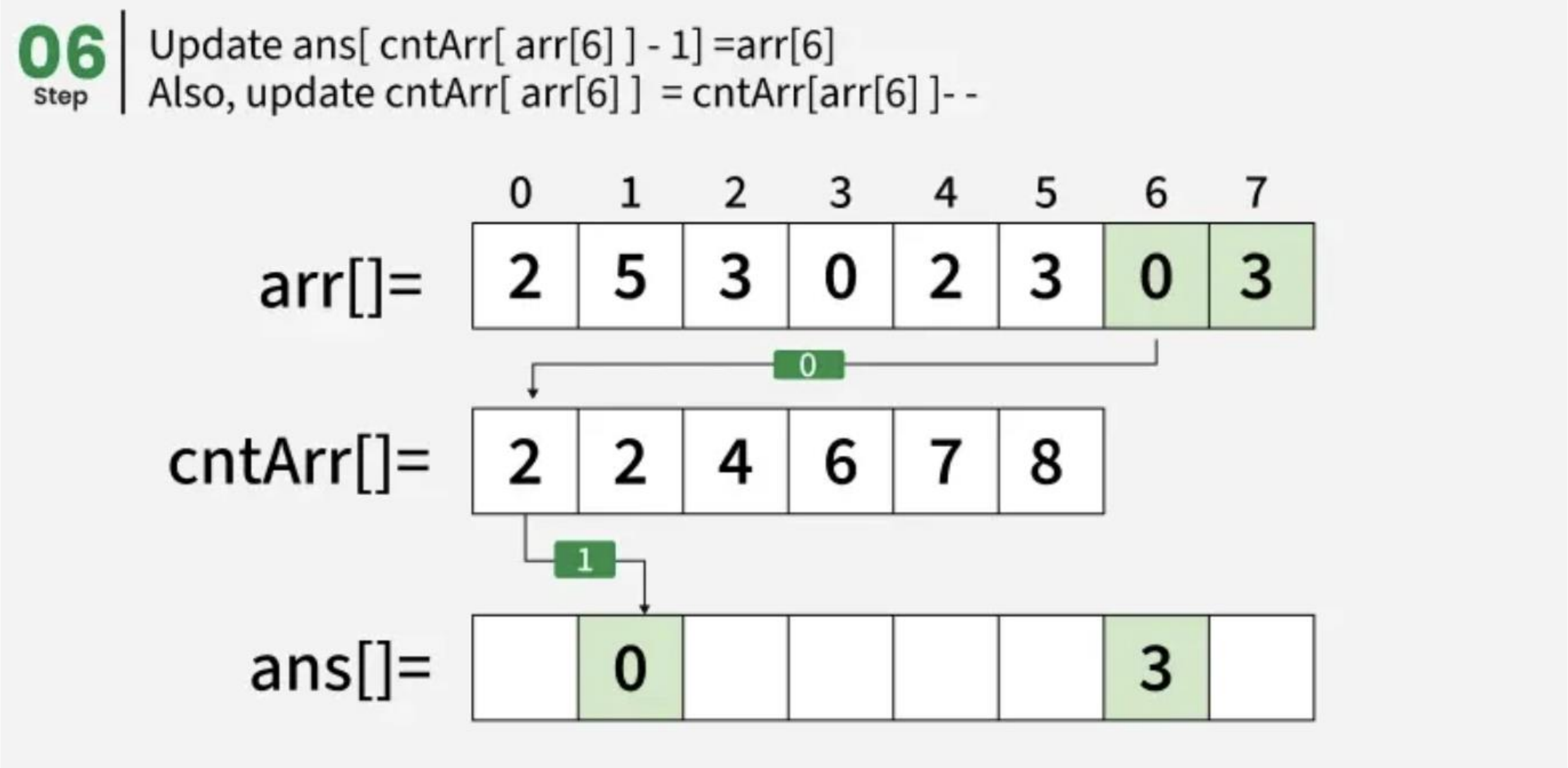
	0	1	2	3	4	5
cntArr[]=	2	0	2	3	0	1

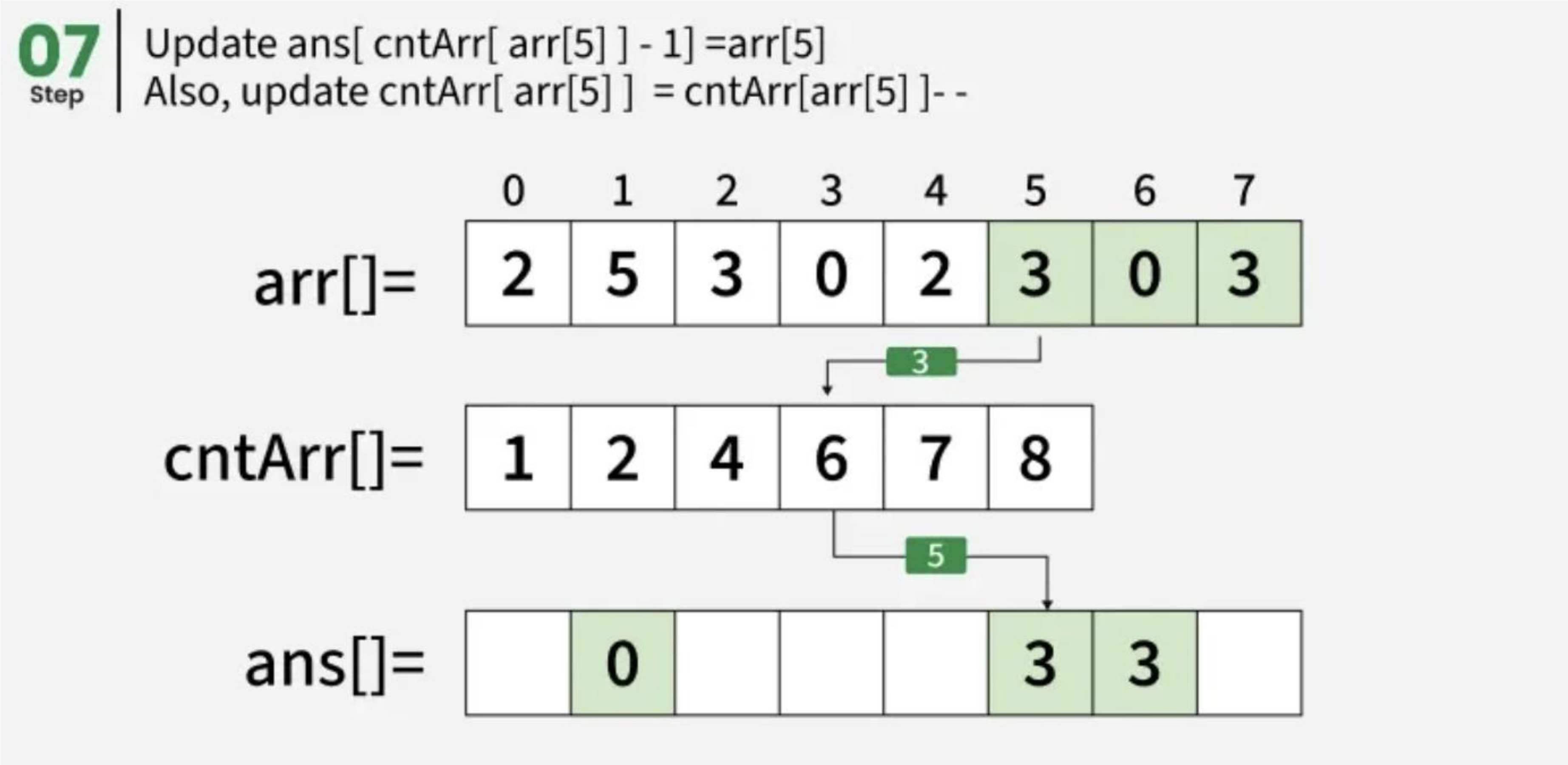
04 Step | Store the cumulative sum or prefix sum of the elements of the cntArr[] by doing cntArr[i] = cntArr[i - 1] + cntArr[i].

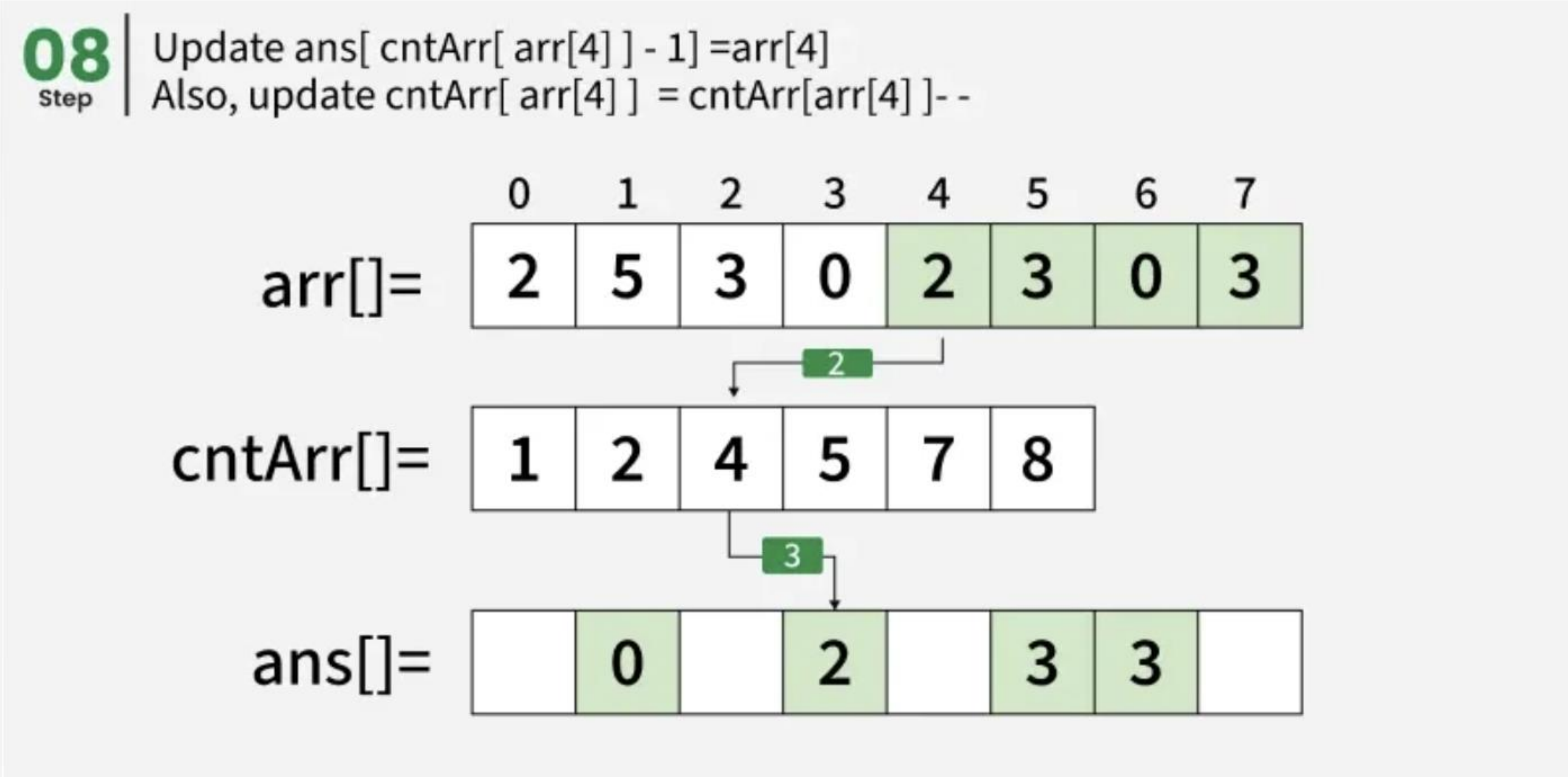
05 Step

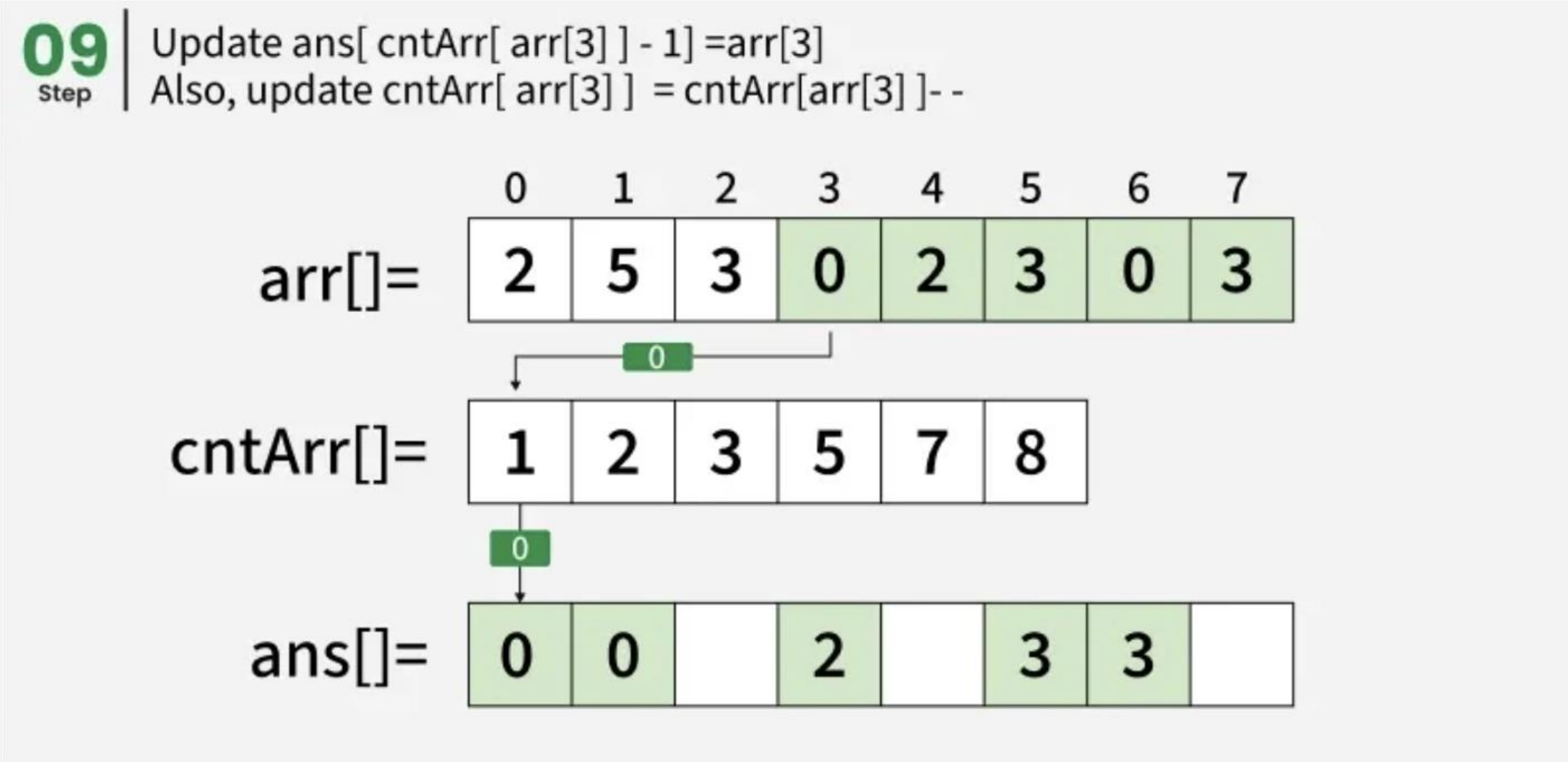
Update ans[cntArr[arr[i]] - 1] = arr[i] and decrement cntArr[arr[i]]. Traverse the input array in reverse to maintain the order of equal elements, ensuring the sort remains stable.

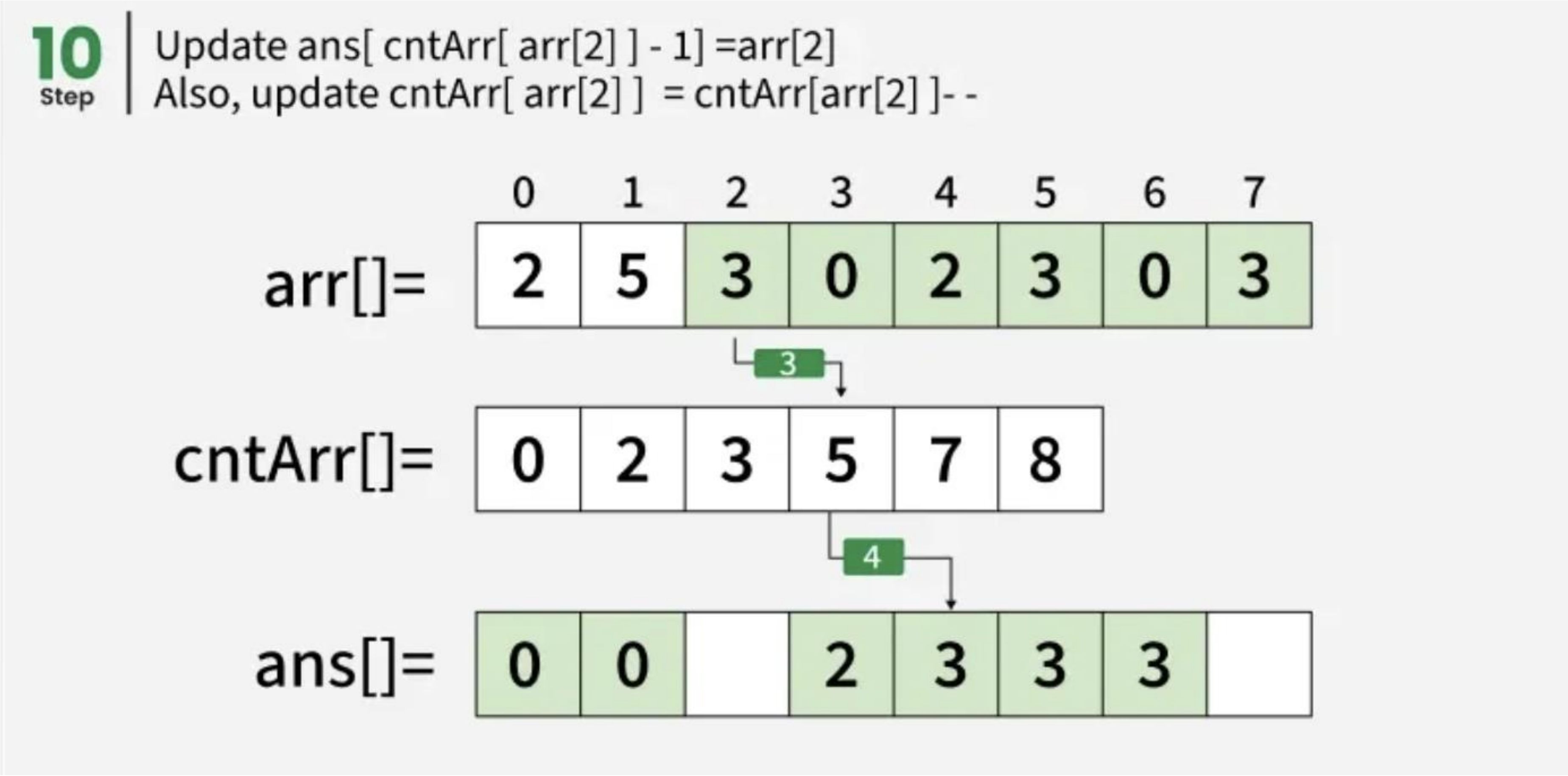


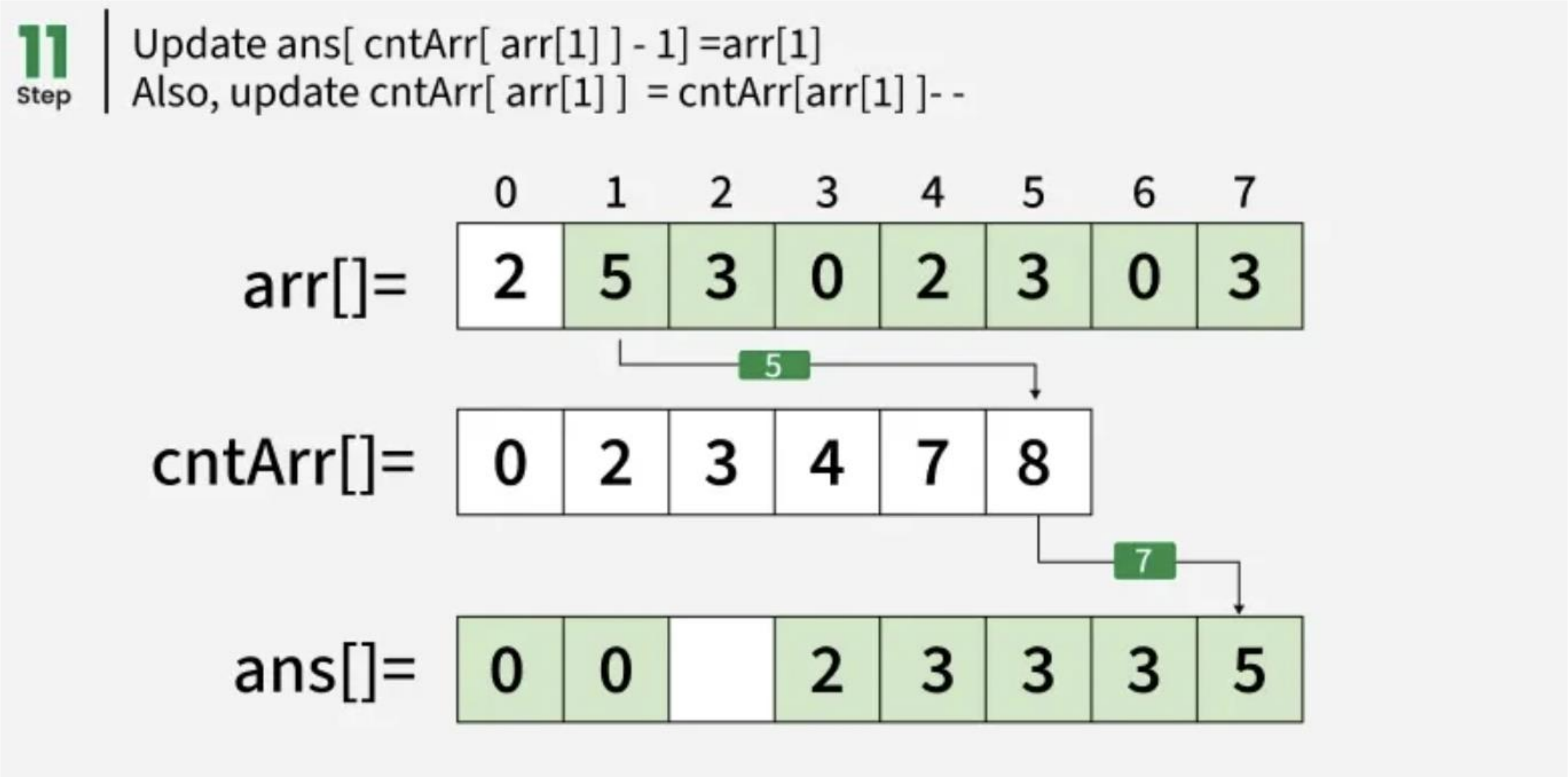

06
Step
Update ans[cntArr[arr[6]] - 1] =arr[6]
Also, update cntArr[arr[6]] = cntArr[arr[6]]- -
0 1 2 3 4 5 6 7
arr[]=
2 5 3 0 2 3 0 3
0
cntArr[]=
2 2 4 6 7 8
1
ans[]=
0 3

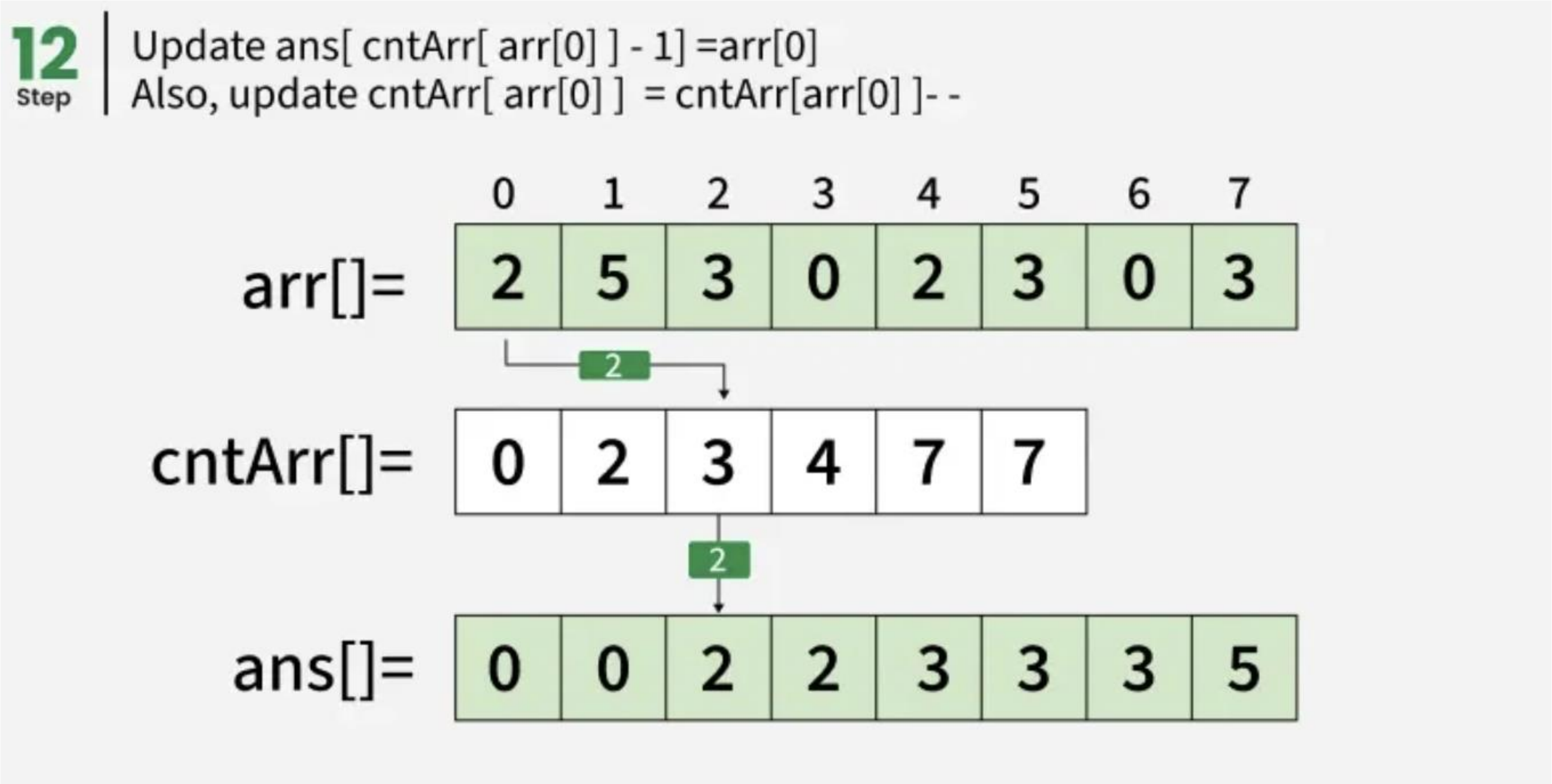

07
Step
Update ans[cntArr[arr[5]] - 1] =arr[5]
Also, update cntArr[arr[5]] = cntArr[arr[5]]- -
0 1 2 3 4 5 6 7
arr[]=
2 5 3 0 2 3 0 3
3
cntArr[]=
1 2 4 6 7 8
5
ans[]=
0 3 3


08
Step
Update ans[cntArr[arr[4]] - 1] =arr[4]
Also, update cntArr[arr[4]] = cntArr[arr[4]]- -
0 1 2 3 4 5 6 7
arr[]= 2 5 3 0 2 3 0 3
2
cntArr[]= 1 2 4 5 7 8
3
ans[]= 0 2 3 3


09
Step
Update ans[cntArr[arr[3]] - 1] =arr[3]
Also, update cntArr[arr[3]] = cntArr[arr[3]]- -
0 1 2 3 4 5 6 7
arr[]=
2 5 3 0 2 3 0 3
0
cntArr[]=
1 2 3 5 7 8
0
ans[]=
0 0 2 3 3


10
Step
Update ans[cntArr[arr[2]] - 1] =arr[2]
Also, update cntArr[arr[2]] = cntArr[arr[2]]- -
0 1 2 3 4 5 6 7
arr[]= 2 5 3 0 2 3 0 3
3
cntArr[]= 0 2 3 5 7 8
4
ans[]= 0 0 2 3 3 3


11
Step
Update ans[cntArr[arr[1]] - 1] =arr[1]
Also, update cntArr[arr[1]] = cntArr[arr[1]]- -
0 1 2 3 4 5 6 7
arr[]=
2 5 3 0 2 3 0 3
5
cntArr[]=
0 2 3 4 7 8
7
ans[]=
0 0 2 3 3 3 5


12
Step
Update ans[cntArr[arr[0]] - 1] =arr[0]
Also, update cntArr[arr[0]] = cntArr[arr[0]]- -
0 1 2 3 4 5 6 7
arr[]=
2 5 3 0 2 3 0 3
2
cntArr[]=
0 2 3 4 7 7
2
ans[]=
0 0 2 2 3 3 3 5

Complexity

- **Time Complexity**: O(N+M) in all cases, where **N** and **M** are the size of **inputArray[]** and **countArray[]** respectively.
- **Auxiliary Space:** O(N+M), where **N** and **M** are the space taken by **outputArray[]** and **countArray[]** respectively.

```
COUNTING-SORT(A, k)
    // A is the input array
    // k is the maximum value in A

    Create array count[0..k] initialized to 0
    Create array output of same length as A

    // Step 1: Count occurrences
    for i = 0 to length(A)-1
        count[A[i]] = count[A[i]] + 1

    // Step 2: Cumulative count
    for i = 1 to k
        count[i] = count[i] + count[i - 1]

    // Step 3: Build output array (stable)
    for i = length(A)-1 downto 0
        output[count[A[i]] - 1] = A[i]
        count[A[i]] = count[A[i]] - 1

    return output
```

References

- **Chapter 10: Data Structures using C** by E. Balagurusamy
- Visit the site for live visualization: https://visualgo.net/

Thank You