# SORTING ALGORITHMS

*Mahsa Ziraksima*
*ACIBADEM UNIVERSITY*
*Fall 2025*

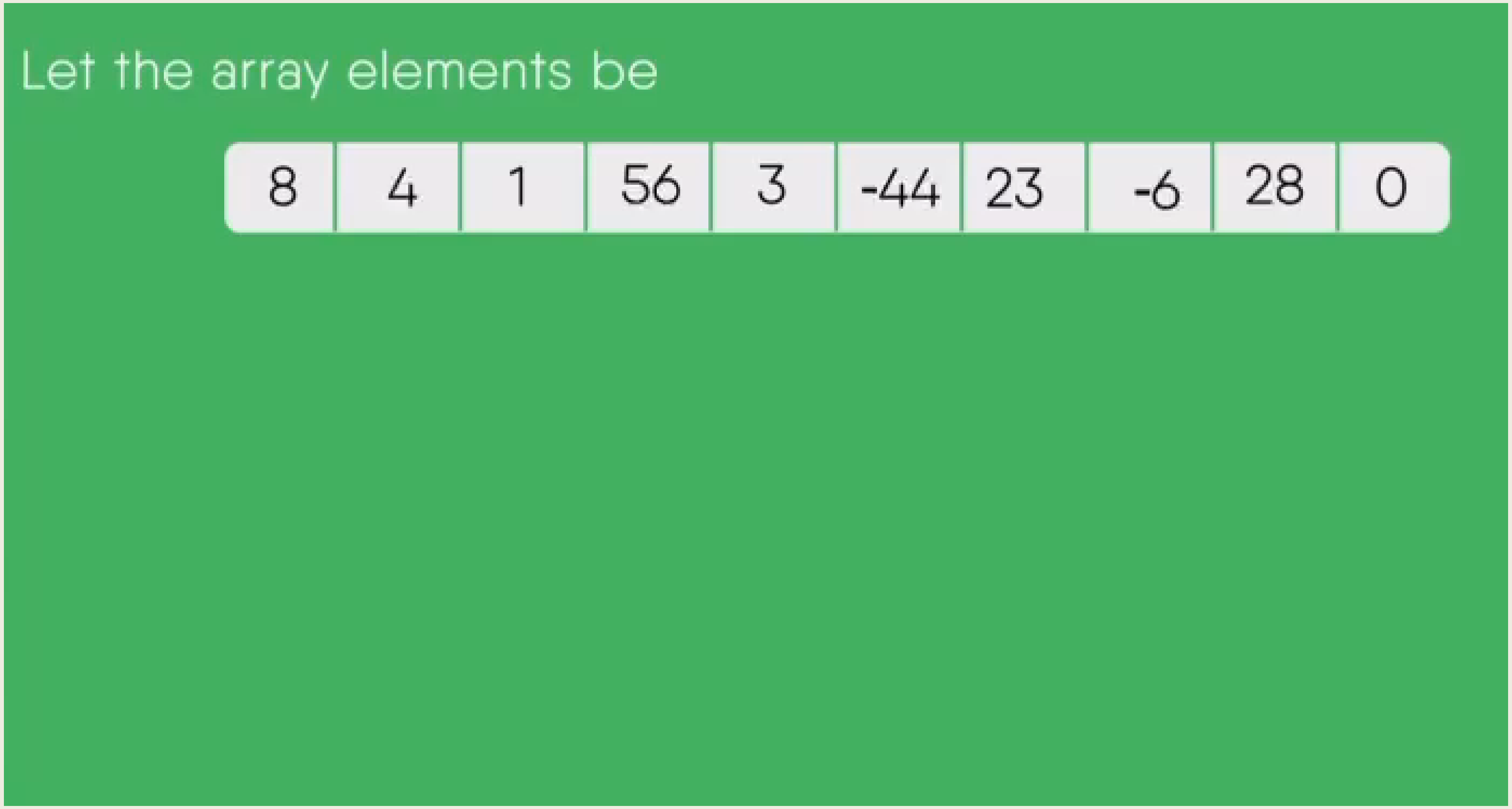
## SUMMARY

- Comb Sort
- Counting Sort
- Radix Sort
- Bucket Sort
- Tim Sort

## COMB SORT

- How It Works:
  - *Initialization: Start with a gap calculated as the length of the array divided by 1.3.*
  - *Comparison and Swapping: For each element, compare it with the element at the current gap distance. If they are out of order, swap them.*
  - *Gap Reduction: After one full pass through the array, reduce the gap and repeat until the gap is 1.*
  - *Final Pass: When the gap is 1, perform passes similar to bubble sort till you are sure the array is sorted.*

## COMB SORT



Let the array elements be

|  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|
| 8 | 4 | 1 | 56 | 3 | -44 | 23 | -6 | 28 | 0 |

## COMB SORT

- Overview: Comb sort is an efficient sorting algorithm that enhances the basic bubble sort method. By allowing exchanges of items that are far apart, it could take big steps, leading to faster sorting.

- Time Complexity: O($n^2$); it is generally faster due to the reduced number of total comparisons.

- Use Cases: It can be effectively used for sorting small to medium-sized datasets where simplicity and ease of implementation are key.

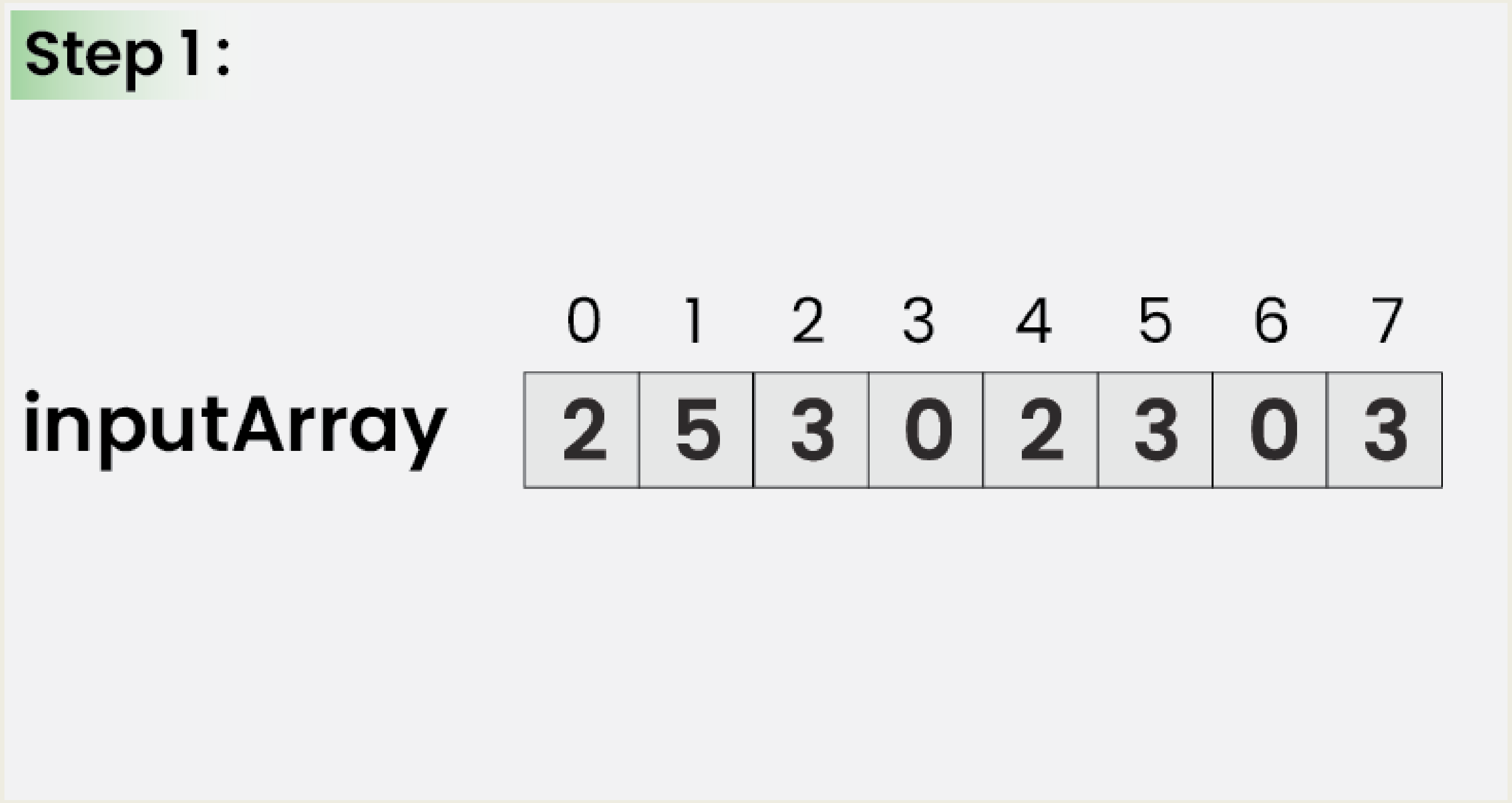
## COMB SORT

- The shrink factor has a great effect on the efficiency of comb sort. A value too small slows the algorithm down by making unnecessarily many comparisons, whereas a value too large fails to effectively deal with it, making it require many passes with a gap of 1.

- The first while should be in the form of “while gap > 1 or swapped” because swapped checks if there is any swapping in the array and if not, we can proceed to next step or finishing the sorting process. We also need to check if gap is bigger than 1 due to the possibility of not being able to swap in middle steps, but we need to check other gap sizes.

- The need for multiple iterations when gap is equal to 1 happens when shrink factor is not adjusted suitably.

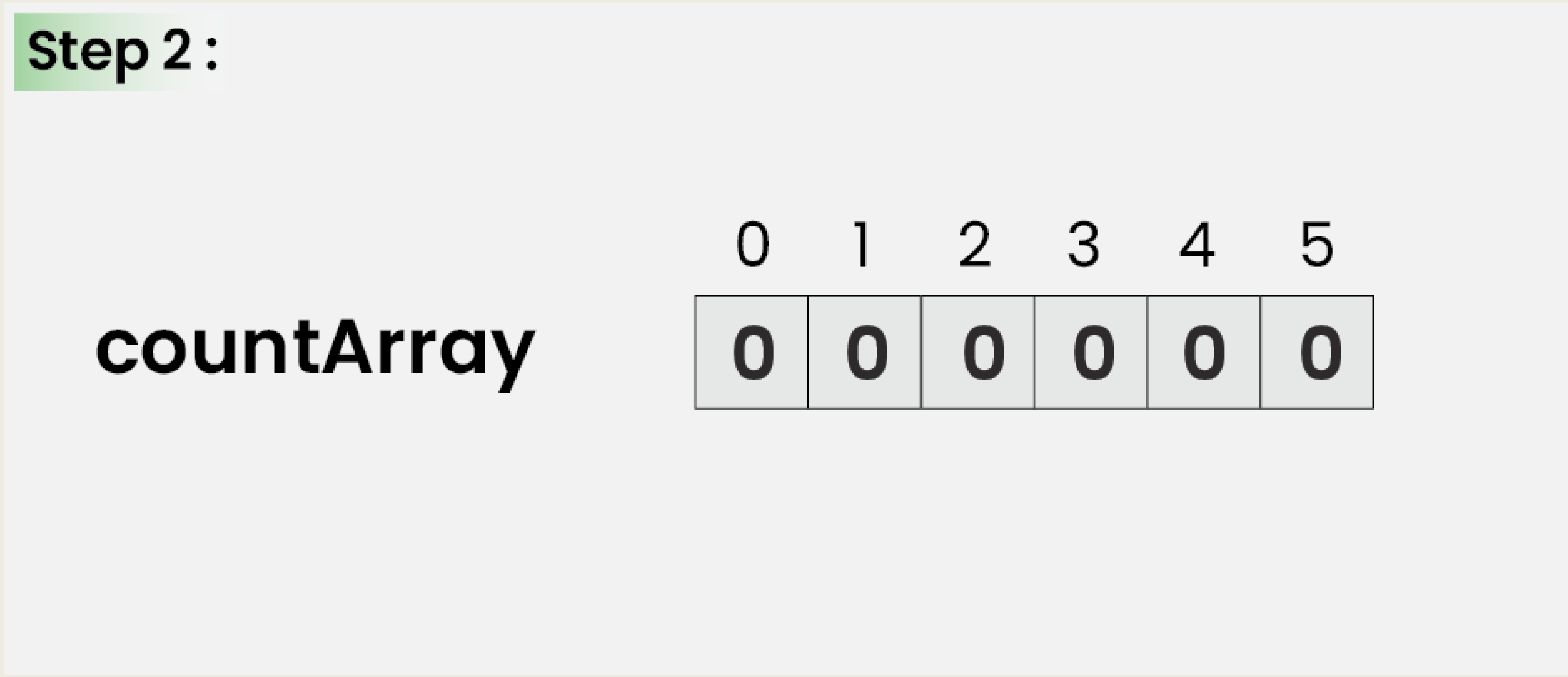
# COUNTING SORT



**Step 1:**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| inputArray | 2 | 5 | 3 | 0 | 2 | 3 | 0 | 3 |

# COUNTING SORT

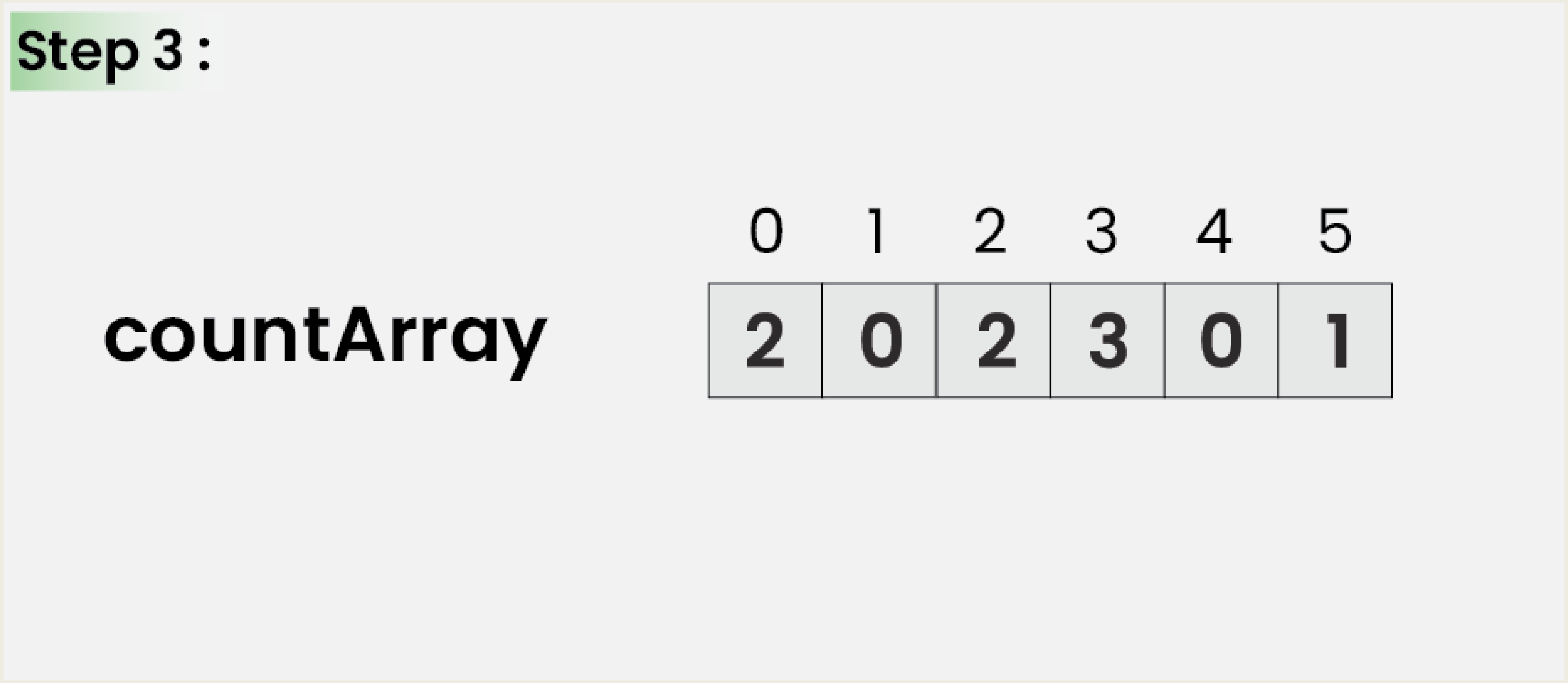


**Step 2 :**

countArray

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |

# COUNTING SORT



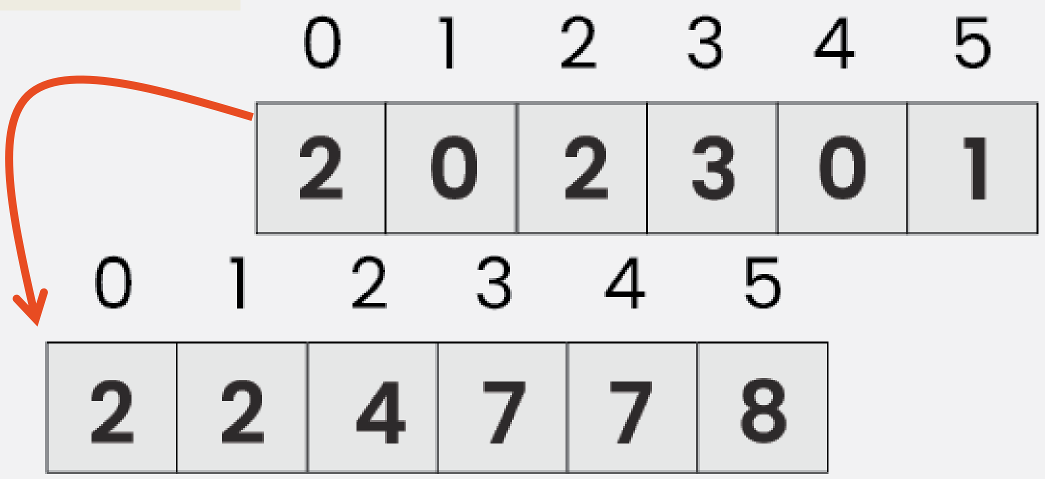
**Step 3 :**

**countArray**

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 2 | 0 | 2 | 3 | 0 | 1 |

# COUNTING SORT

**Step 4 :**

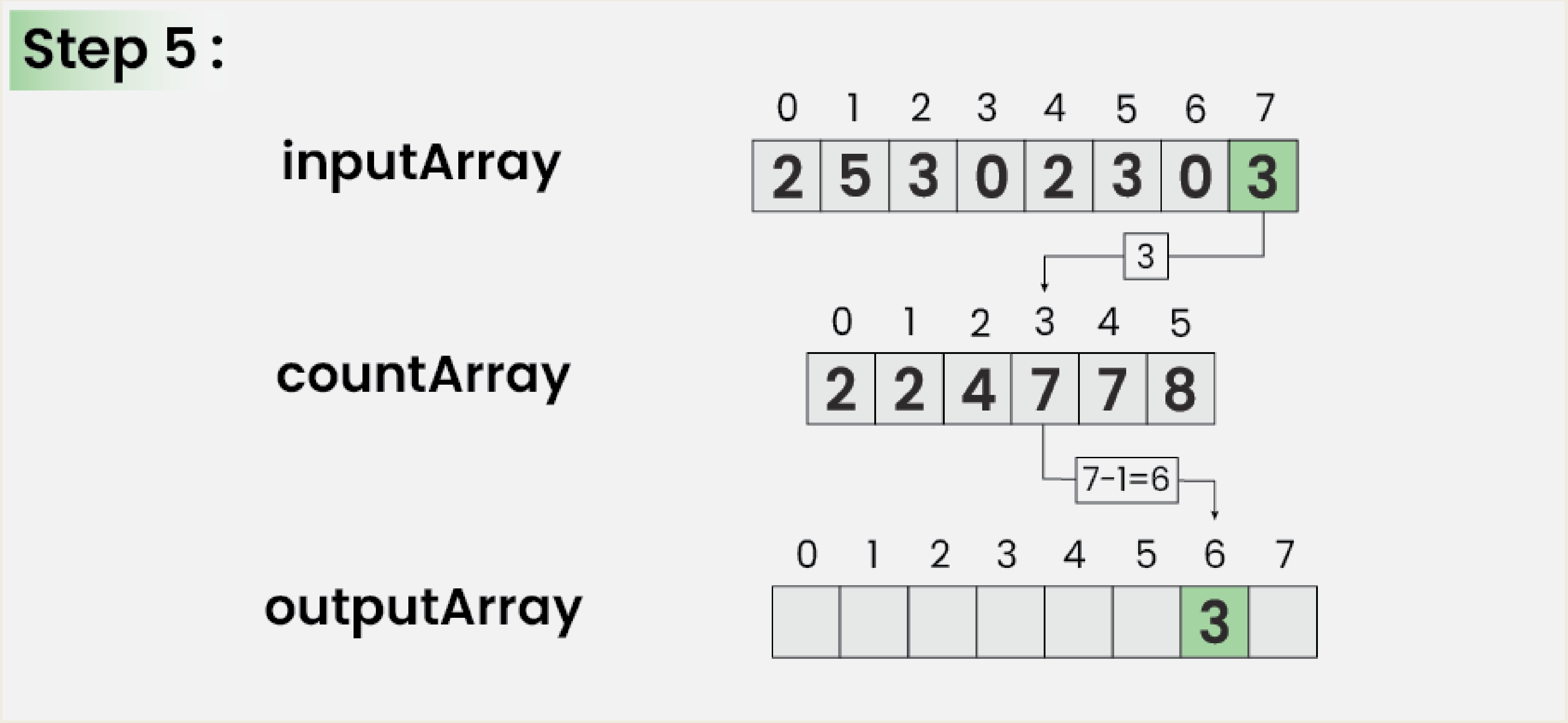


| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 2 | 0 | 2 | 3 | 0 | 1 |

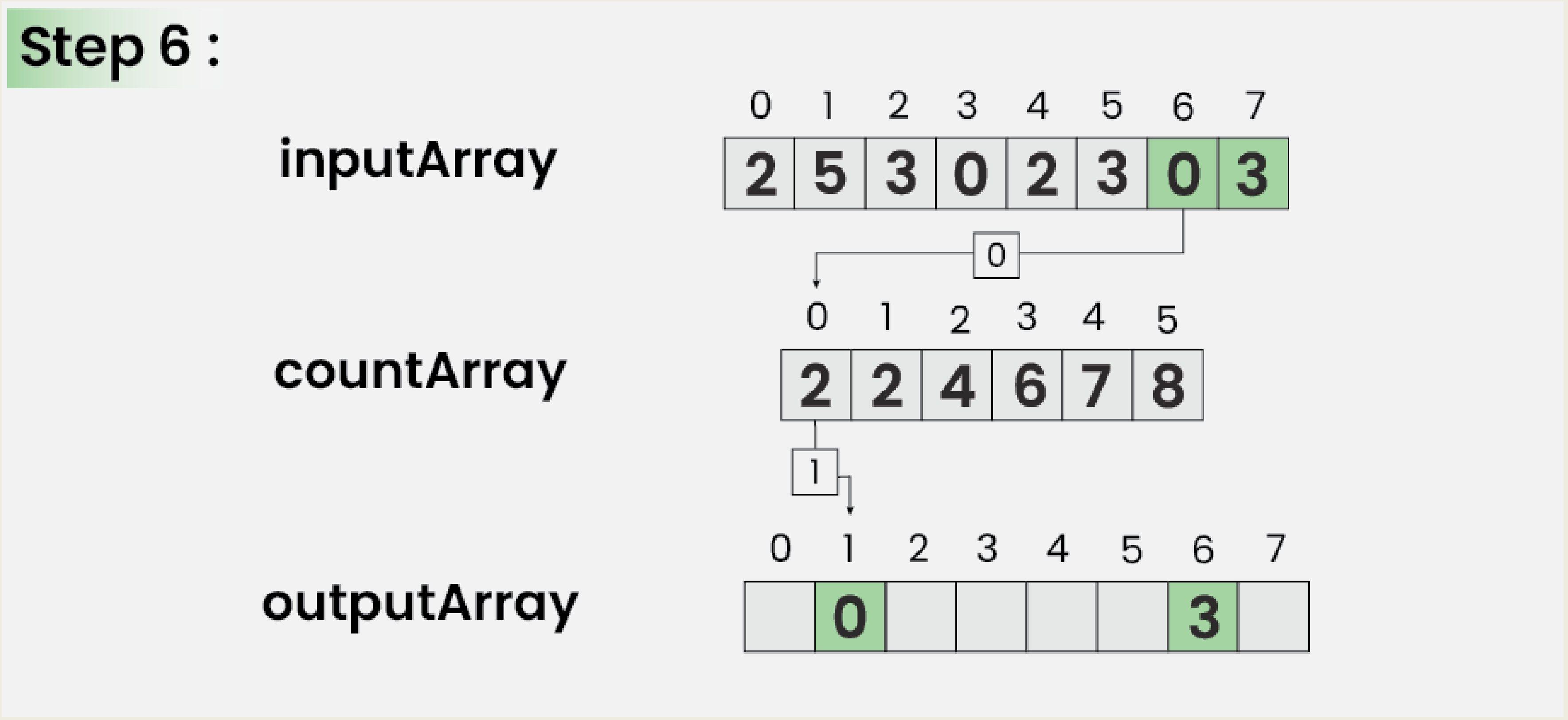
**countArray**

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 2 | 2 | 4 | 7 | 7 | 8 |

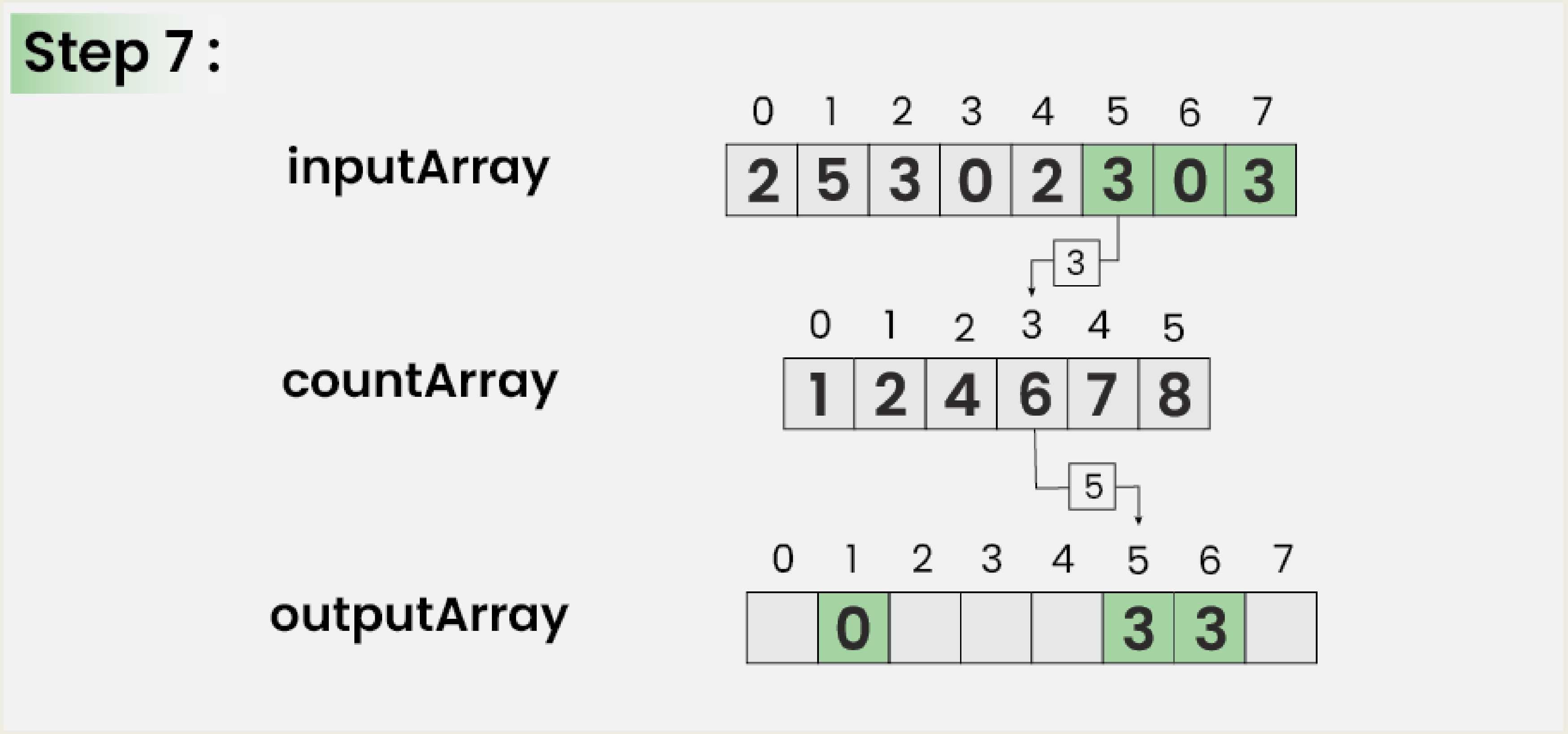
# COUNTING SORT



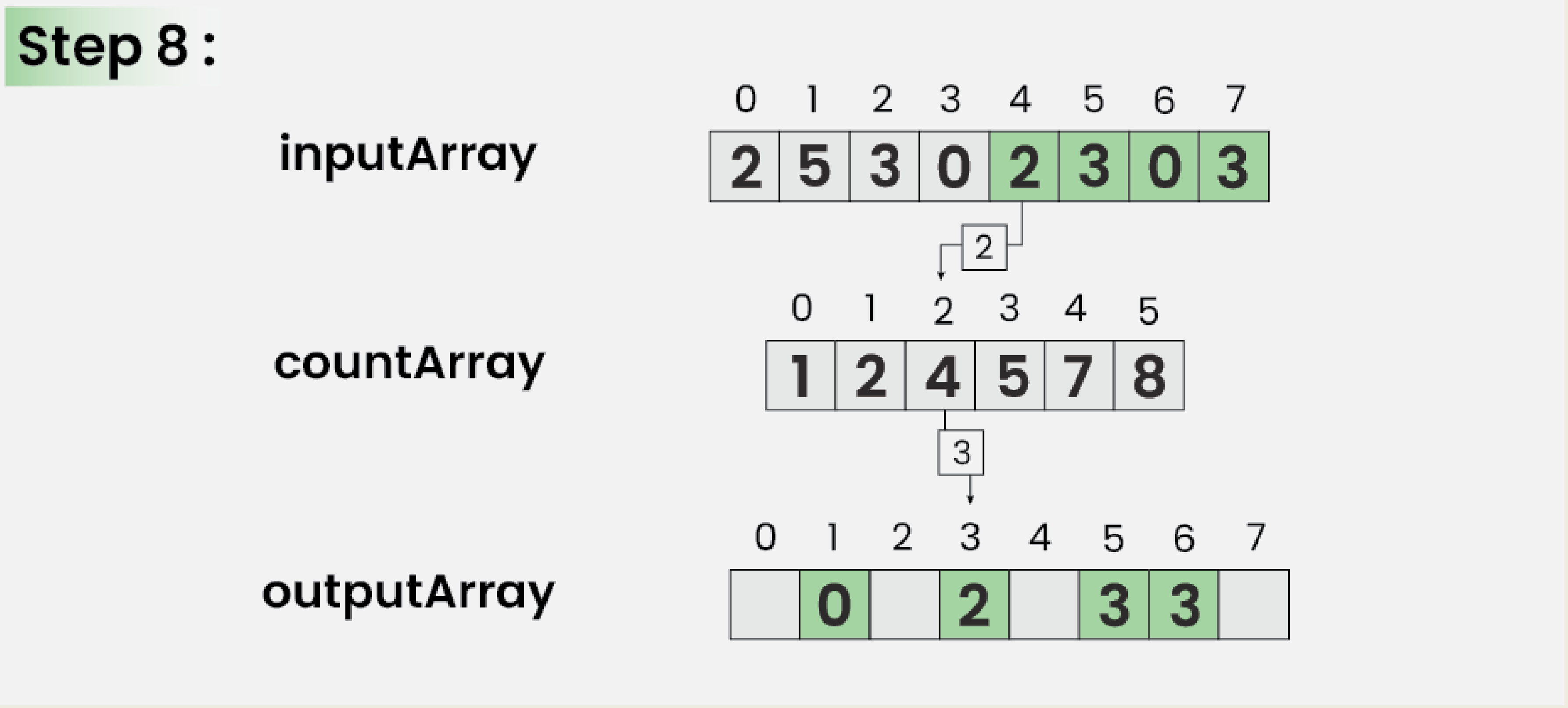
# COUNTING SORT



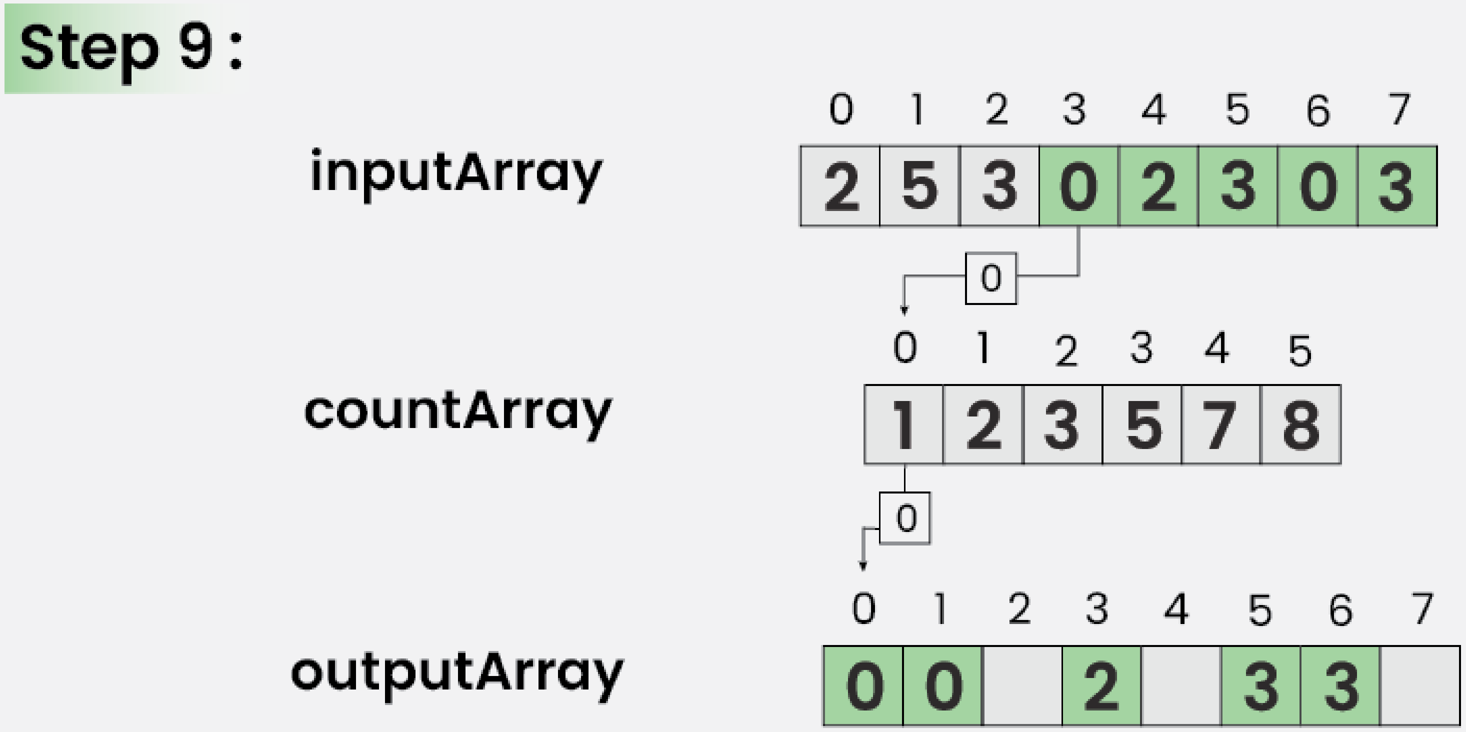
## COUNTING SORT



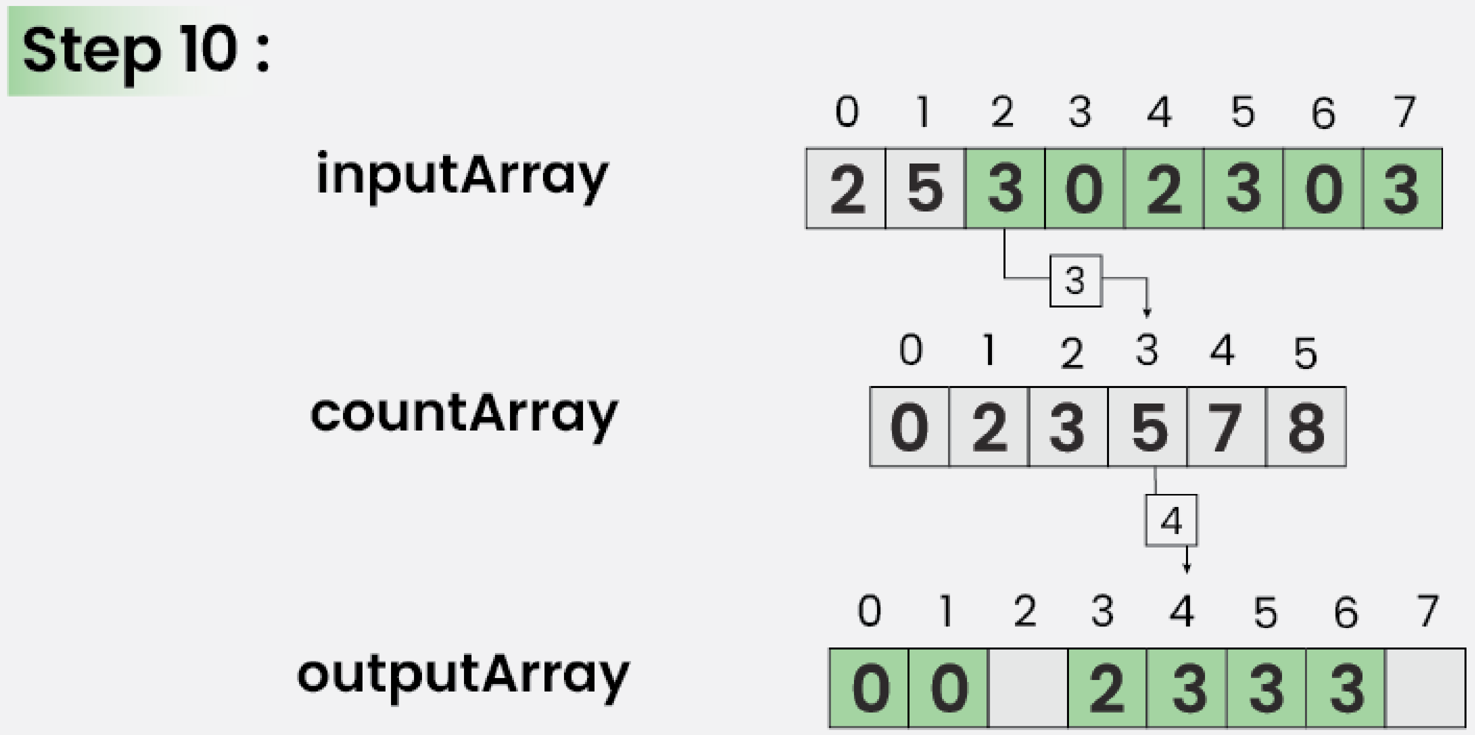
# COUNTING SORT



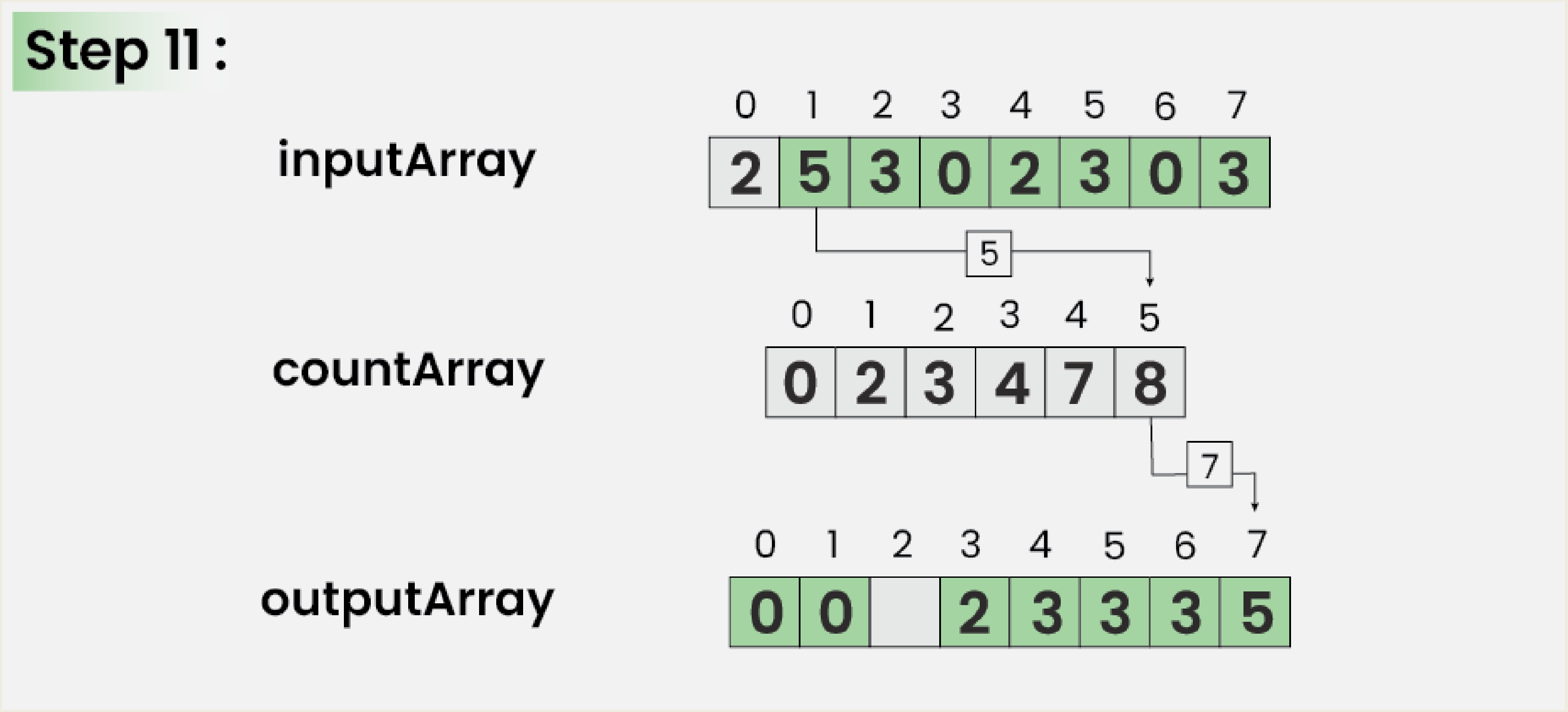
## COUNTING SORT



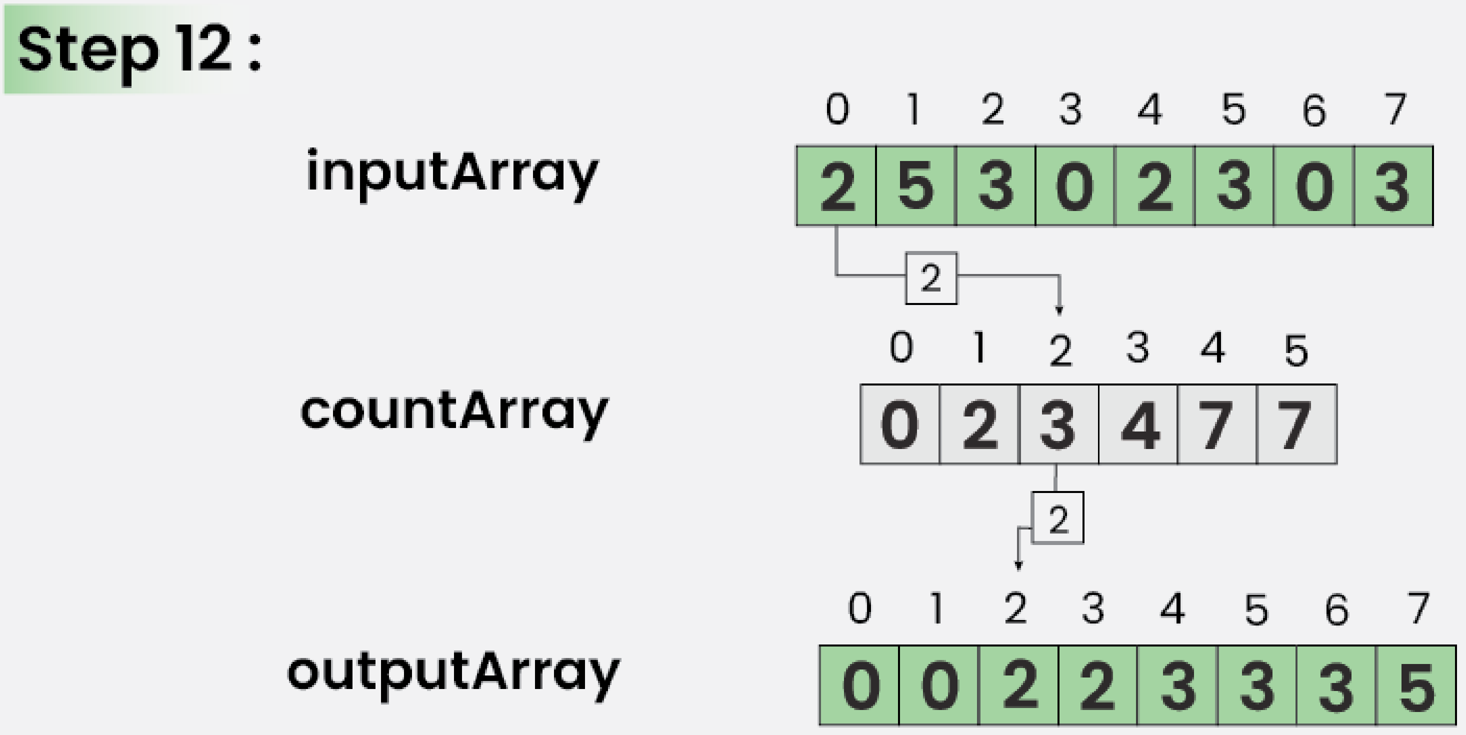
# COUNTING SORT



# COUNTING SORT



# COUNTING SORT



## COUNTING SORT

- How It Works:
  - *Counting Phase: Create an auxiliary array (the count array) where each index corresponds to the value of the elements in the input array. For each element in the input array, increment the corresponding index in the count array.*
  - *Cumulative Count: Modify the count array so that each element at each index stores the sum of previous counts. This gives the position of each element in the sorted output.*
  - *Building the Output: Iterate through the input array again, placing each element in its correct position in the output array, using the information from the count array. Decrement the count in the count array each time an element is placed.*

## COUNTING SORT

- Overview: Counting Sort is a non-comparison sorting algorithm that works by counting the occurrences of each unique element in the input array.

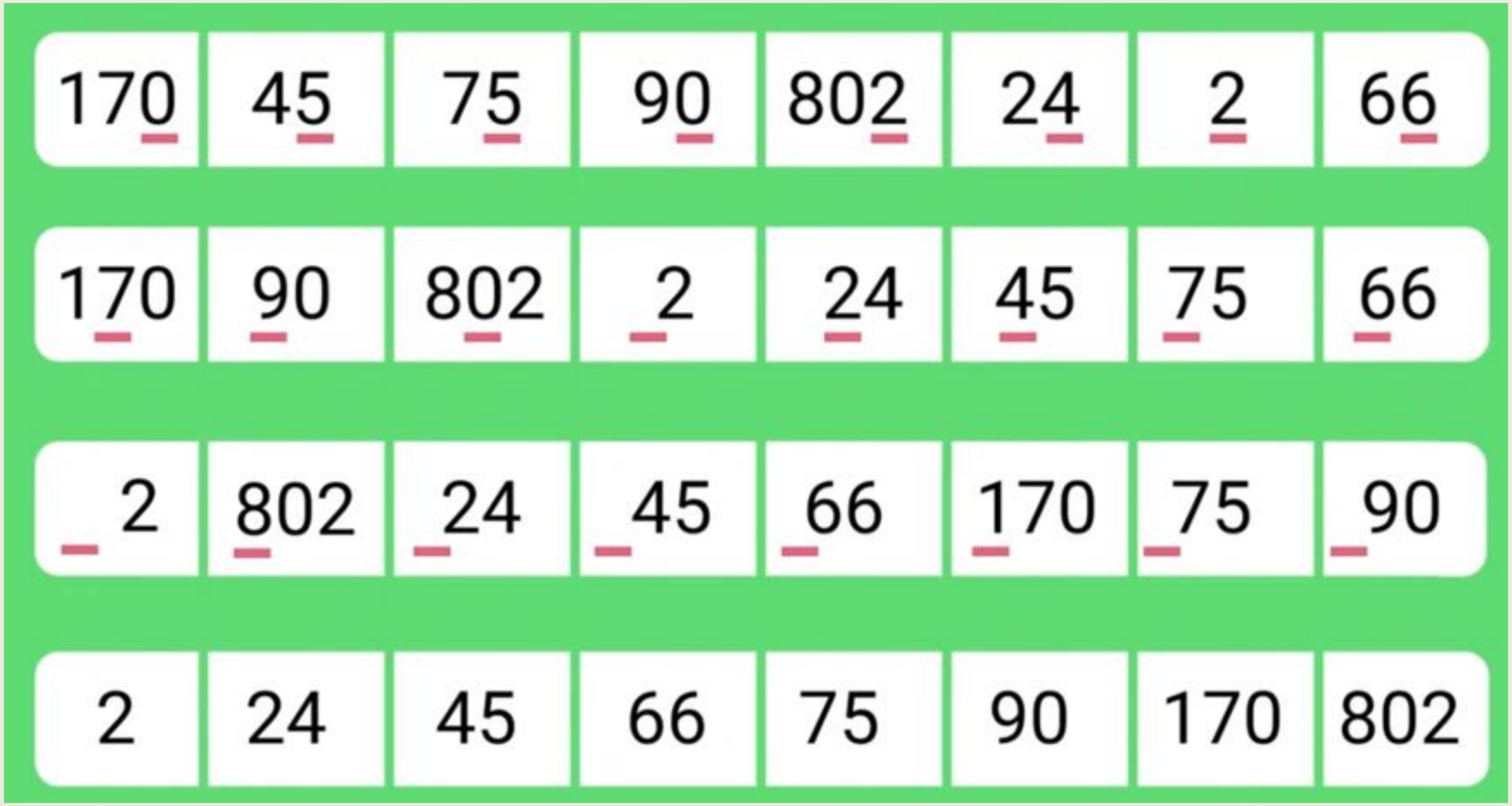
- Time Complexity: O(n+k), where n is the number of elements in the input array and k is the range of the input values.

- Use Cases: Best suited for sorting integers or categorical data with a known range.

## RADIX SORT

- How It Works:
  - *Digit Processing: Sort the input numbers by each digit, starting from the least significant digit (LSD) to the most significant digit (MSD).*
  - *Stable Sorting: Use a stable sorting algorithm (like Counting Sort) to sort the numbers based on the current digit. This ensures that numbers with the same digit maintain their relative order.*
  - *Repeat: Repeat the process for each digit until all digits have been processed.*

## RADIX SORT



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 170 | 45 | 75 | 90 | 802 | 24 | 2 | 66 |
| 170 | 90 | 802 | 2 | 24 | 45 | 75 | 66 |
| 2 | 802 | 24 | 45 | 66 | 170 | 75 | 90 |
| 2 | 24 | 45 | 66 | 75 | 90 | 170 | 802 |

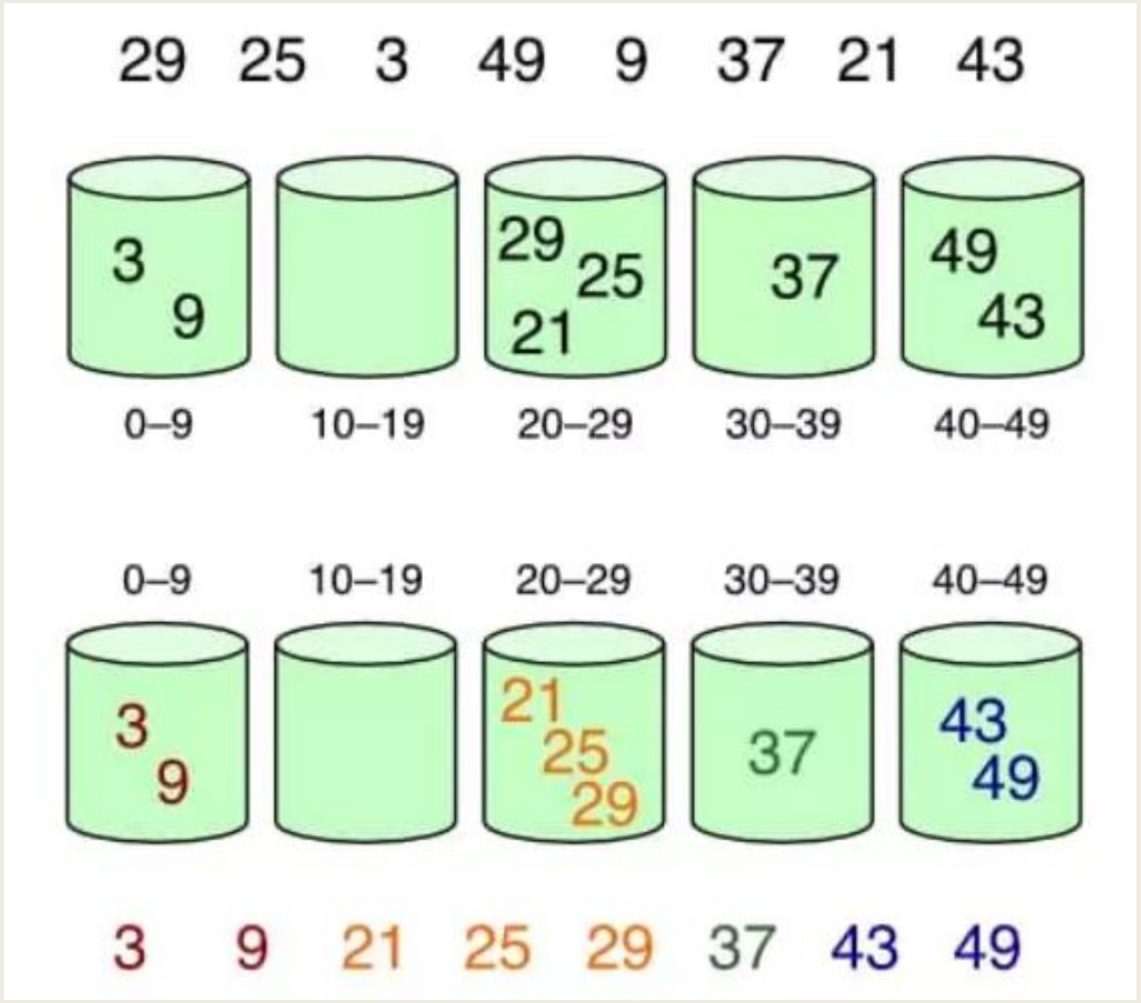
## RADIX SORT

- Overview: Radix Sort is a non-comparison sorting algorithm that sorts numbers by processing individual digits. It works on integers.

- Time Complexity: O(n·d), where n is the number of elements and d is the number of digits in the largest number.

- Use Cases: Effective for sorting large datasets of integers or strings where the number of digits is significantly less than the number of items.

## BUCKET SORT


29 25 3 49 9 37 21 43
3 9
29 25 21
37
49 43
0–9
10–19
20–29
30–39
40–49
0–9
10–19
20–29
30–39
40–49
3 9
21 25 29
37
43 49
3 9 21 25 29 37 43 49

## BUCKET SORT

- How It Works:
  - *Creating Buckets: Divide the input range into k equally spaced intervals (buckets).*
  - *Distributing Elements: Traverse the input array and place each element into its corresponding bucket based on its value.*
  - *Sorting Buckets: Sort each bucket using a different sorting algorithm (like Insertion Sort or even another Bucket Sort).*
  - *Combining: Concatenate the sorted buckets to get the final sorted array, using another algorithm, such as insertion sort.*

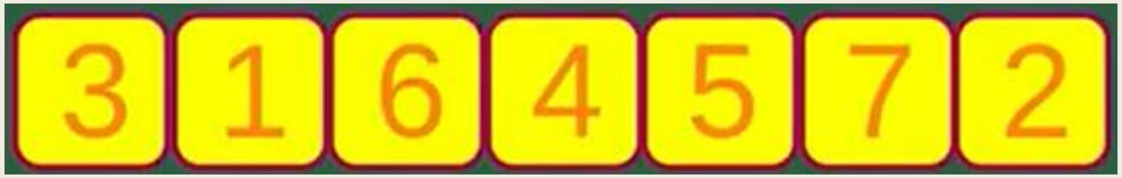
## BUCKET SORT

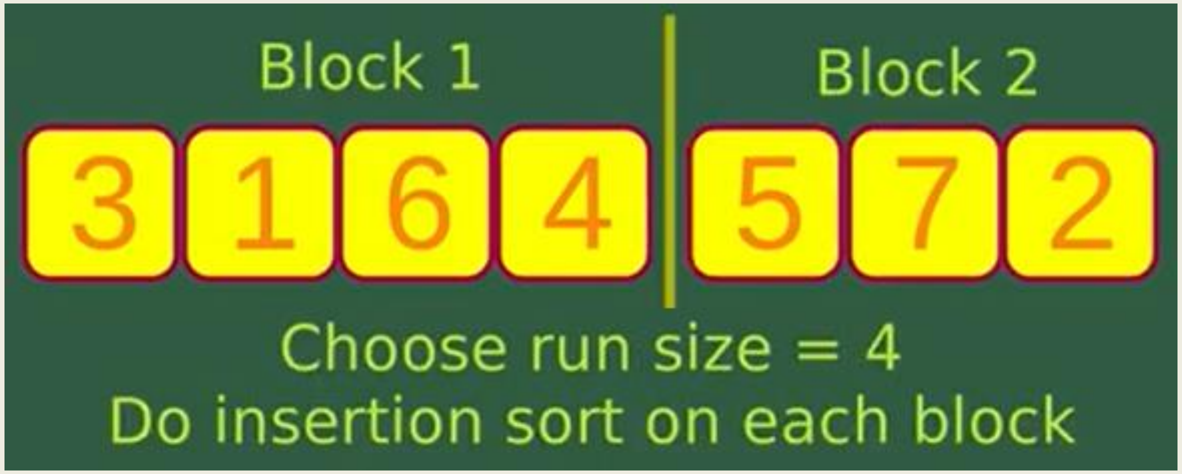
- Overview: Bucket Sort distributes elements into a number of buckets and then sorts these buckets individually. It’s useful for uniformly distributed data.

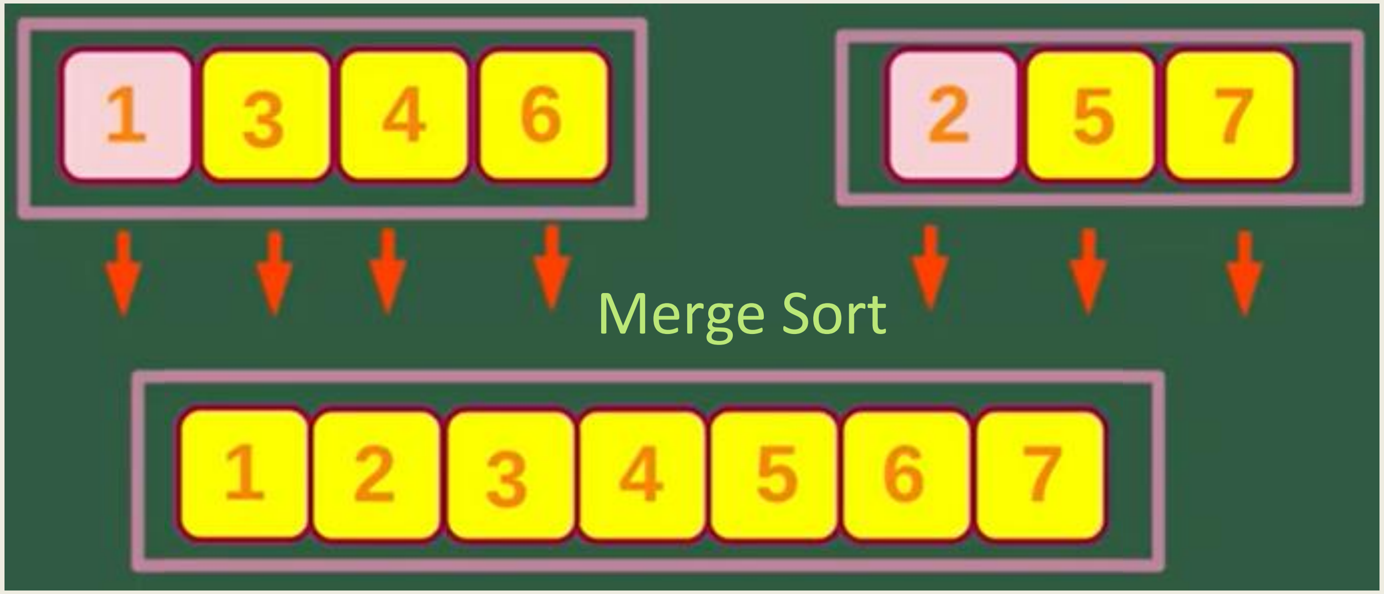
- Time Complexity: Average case O(n+k), where n in the number of elements and k is the number of buckets; the efficiency depends on the distribution of the input and the number of buckets.

- Use Cases: Works well for floating-point numbers or when the data is uniformly distributed.

## TIM SORT


3 1 6 4 5 7 2


Block 1
Block 2
3 1 6 4
5 7 2
Choose run size = 4
Do insertion sort on each block


1 3 4 6
2 5 7
Merge Sort
1 2 3 4 5 6 7

## TIM SORT

- How It Works:
  - *Divide into runsThe*

    array is scanned to find small naturally ordered sequences (called runs) — these are parts of the array that are already increasing or decreasing.

    - If a run is decreasing, it’s reversed to make it increasing.
    - If a run is shorter than a certain threshold (called MINRUN, typically between 32 and 64), insertion sort is used in the next step to extend it.
  - *Sort each run (if needed)*

    Each run is sorted individually using insertion sort (fast for small data).
  - *Merge runs*

    Once all runs are identified and sorted, Timsort merges them using a process similar to merge sort.

## TIM SORT

- Overview: Tim Sort is a hybrid sorting algorithm derived from Merge Sort and Insertion Sort. It’s the default sorting algorithm in Python and Java.

- Time Complexity: O(nlogn) in the worst case, and O(n) in the best case for nearly sorted data.

- Use Cases: Ideal for real-world data that often contains runs of sorted elements, making it very efficient in practice.