14-11-2023

Sorting

Prof . Shweta Dhawan Chachra

Sorting

- Sorting refers to **arranging data in a particular format.**
- Sorting algorithm **specifies the way to arrange** data in a particular order.
- Most common orders **are in numerical or lexicographical order.**

Sorting

- The importance of sorting lies in the fact that **data searching can be optimized to a very high level, if data is stored in a sorted manner.**

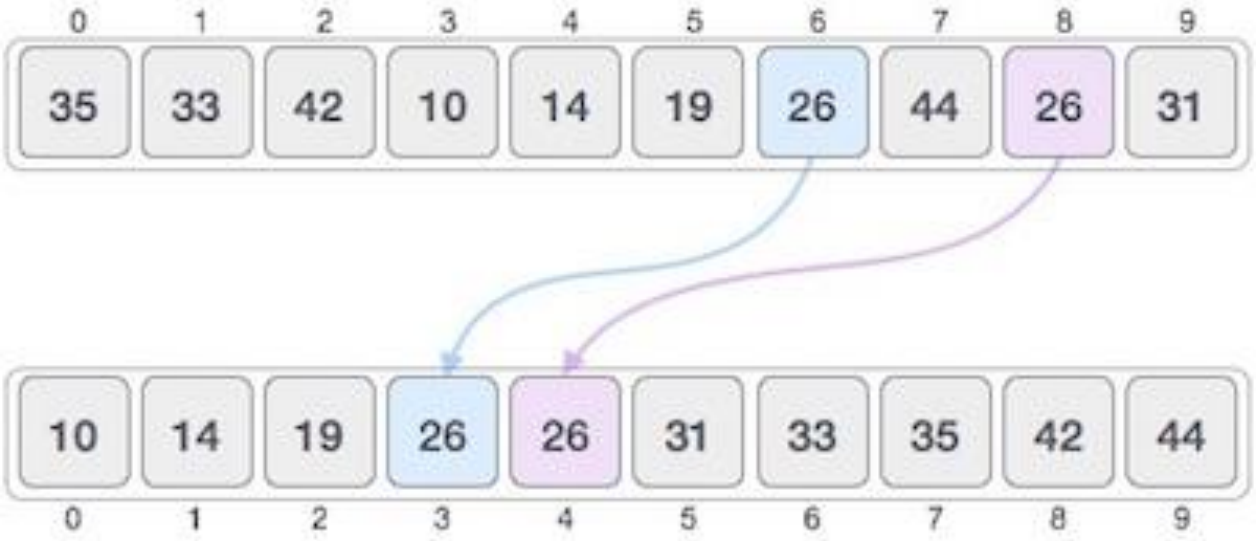
- Sorting is also used to **represent data in more readable formats**

Sorting

- Some real-life scenarios −
- **Telephone Directory** − The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.
- **Dictionary** − The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

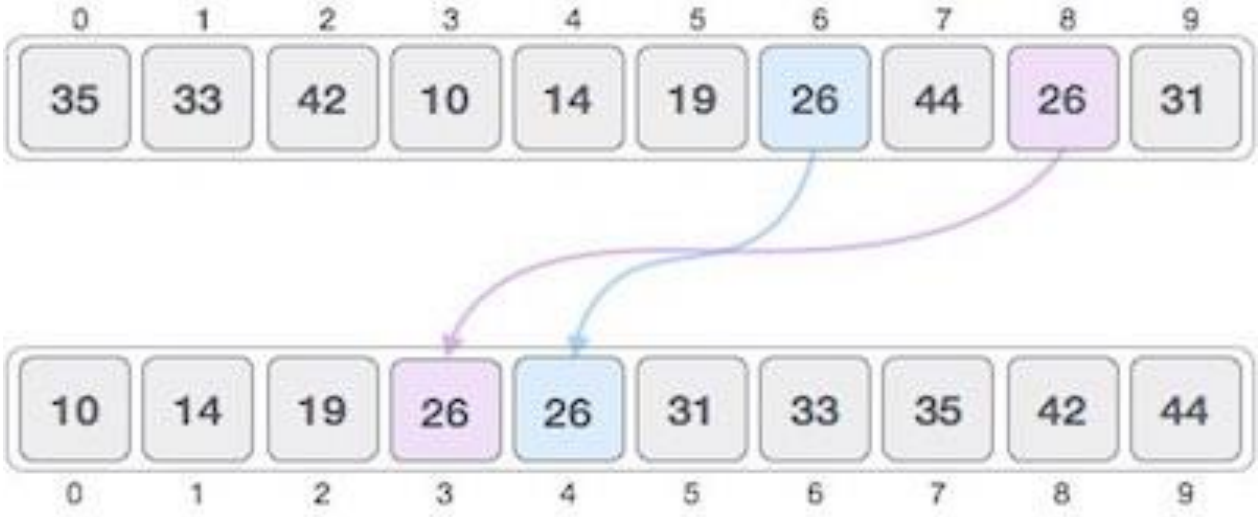
Stable and Not Stable Sorting

- If a sorting algorithm, after sorting the contents, does not change the sequence of similar content in which they appear, it is called **stable sorting**.



Stable and Not Stable Sorting

- If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called **unstable sorting**.



Stable and Not Stable Sorting

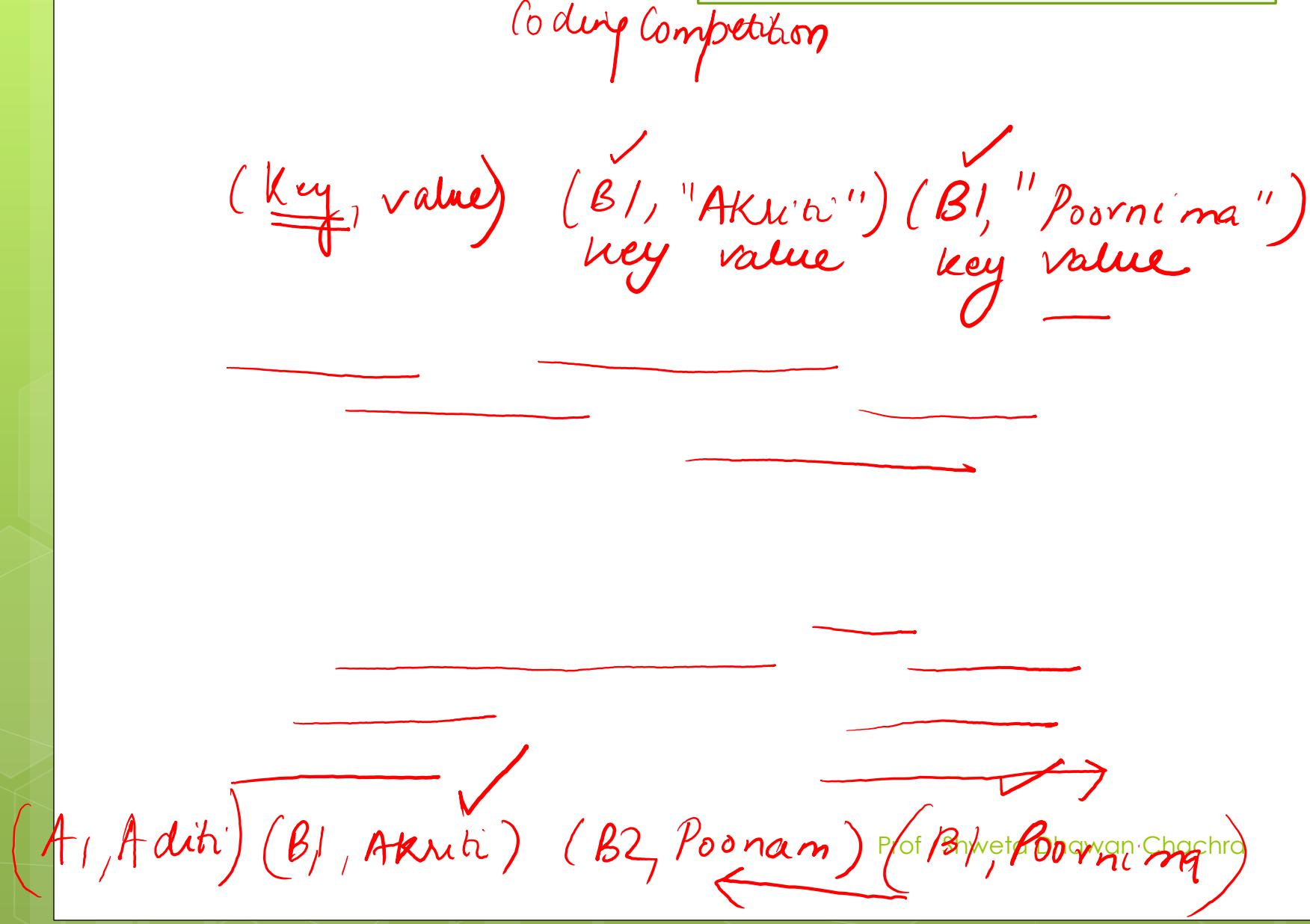
- Stability of an algorithm matters when we wish to maintain the sequence of original elements, like in a tuple for example.

Stable and Not Stable Sorting

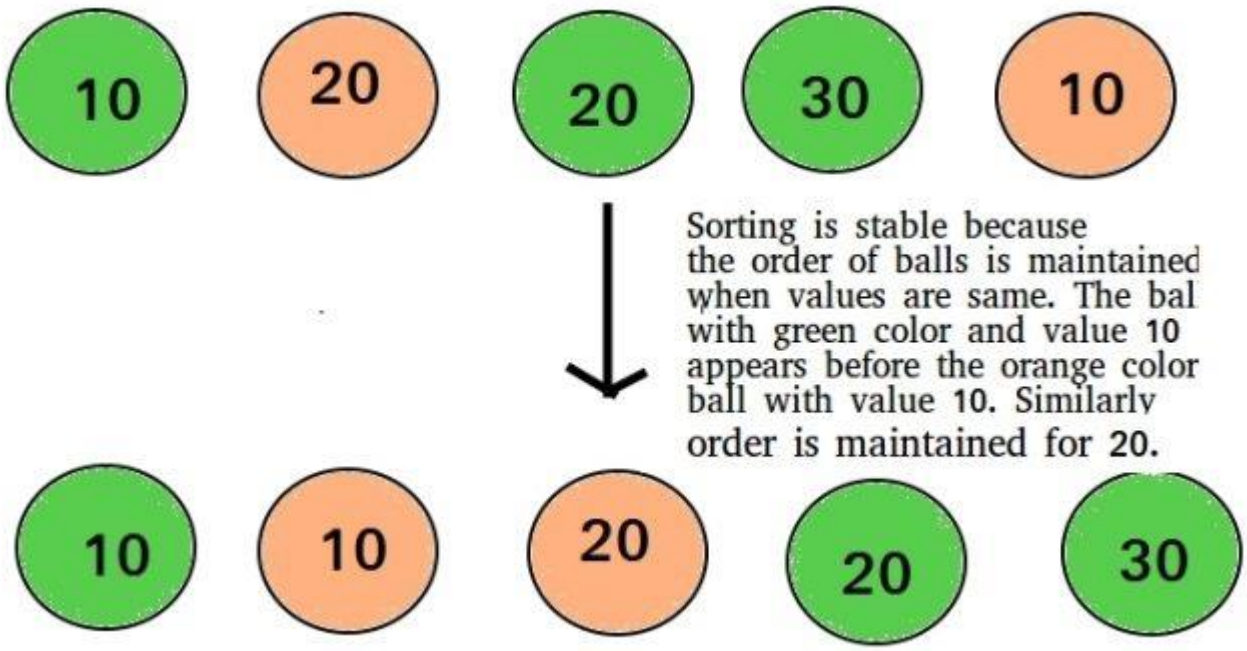
- Stability is mainly important when we have key value pairs with duplicate keys possible (like people names as keys and their details as values). And we wish to sort these objects by keys.

- **What is it?**
  A sorting algorithm is said to be **stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.**

Stable and Not Stable Sorting



Stable and Not Stable Sorting



Bubble Sort

- Bubble sort is an algorithm that **compares the adjacent elements and swaps their positions if they are not in the intended order.**

Bubble Sort

- For sorting N elements, The steps are-
- **Pass 1**
  - First Compare 1st element with 2nd element of array, If 1st >2nd , swap the elements else If 1st<2nd then compare 2nd with 3rd
  - If 2nd>3rd then interchange value of 2nd and 3rd
  - Now compare 3rd value with 4th
  - Similarly compare until N-1th element is compared with Nth element
  - **Now the highest value is reached at Nth place**
- **Pass 2**
  - Now elements will be compared until N-1 elements

Bubble Sort

- **In each iteration, the comparison takes place up to the last unsorted element.**
- The array is sorted when all the unsorted elements are placed at their correct positions.

Bubble Sort

- Let the elements be-

13,32,20,62,68,52,38,46

Pass1

- Compare 1st ,2nd element, 13<32 No change
- Compare 2nd,3rd element, 32>20,Interchange

13,20,32,62,68,52,38,46

Bubble Sort

13,20,32,62,68,52,38,46

- Compare 3rd,4th element, 32<62 No change
- Compare 4th,5th element, 62<68 No change
- Compare 5th,6th element, 68>52 Interchange swap

13,20,32,62,52,68,38,46 swap

- Compare 6th,7th element, 68>38 Interchange

13,20,32,62,52,38,68,46

- Compare 7th ,8th element, 68>46 Interchange

13,20,32,62,52,38,46,68 swap

Biggest element goes at the last position

Pass 1 completed,

The Biggest element has reached the last position

Bubble Sort

13,20,32,62,52,38,46,68

Pass 2

- Now showing only interchange
- 62>52, so 13,20,32,52,62,38,46,68
- 62>38, so 13,20,32,52,38,62,46,68
- 62>46 so 13,20,32,52,38,46,**62,68**
- After Pass 2, second largest element reaches 2nd last position

Bubble Sort

13,20,32,52,38,46,62,68

Pass 3

- Now showing only interchange
- 52>38 so 13,20,32,38,52,46,62,68
- 52>46 so 13,20,32,38,46,**52,62,68**
- After Pass 3, 3rd largest element reaches 3rd last position

13,20,32,38,46,52,62,68

Pass 4

- No of Exchanges=0, so list is sorted

Bubble Sort

Analysis-

This algorithm is not suitable for large data sets as its **average and worst case complexity are of $O(n^2)$ where n is the number of items.**

It behaves as O(n) for Sorted array

Bubble Sort

- **How many Passes are needed to sort the following list of numbers?**
  5 1 4 2 8

Bubble Sort

- **First Pass:**
  (**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
  (1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4
  (1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2
  (1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.
- **Second Pass:**
  (**1 4** 2 5 8) –> (**1 4** 2 5 8)
  (1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2
  (1 2 **4 5** 8) –> (1 2 **4 5** 8)
  (1 2 4 **5 8**) –> (1 2 4 **5 8**)
  Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.
- **Third Pass:**
  (**1 2** 4 5 8) –> (**1 2** 4 5 8)
  (1 **2 4** 5 8) –> (1 **2 4** 5 8)
  (1 2 **4 5** 8) –> (1 2 **4 5** 8)
  (1 2 4 **5 8**) –> (1 2 4 **5 8**)

Bubble Sort

- Algorithm-

```
BUBBLE_SORT(A, N)

Step 1: Repeat Step 2 For I = 0 to N-1
Step 2:     Repeat For J = 0 to N - I - 1
Step 3:                 IF A[J] > A[J + 1]
                        SWAP A[J] and A[J+1]
            [END OF INNER LOOP]
        [END OF OUTER LOOP]
Step 4: EXIT
```

Bubble Sort

```
// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
  int i, j;
  for (i = 0; i < n-1; i++)

    // Last i elements are already in place
    for (j = 0; j < n-i-1; j++)
      if (arr[j] > arr[j+1])
        swap(&arr[j], &arr[j+1]);
}
```

Bubble Sort

```
// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
   int i, j;
   for (i = 0; i < n-1; i++)

       // Last i elements are already in place
       for (j = 0; j < n-i-1; j++)
           if (arr[j] > arr[j+1])
              swap(&arr[j], &arr[j+1]);
}
```

Bubble Sort

		0	1	2	3	4	5	6	7
i = 0	j								
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
i =1	0	3	1	5	8	2	4	7	9
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i = 2	0	1	3	5	2	4	7	8	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i = 3	0	1	3	2	4	5	7		
	1	1	3	2	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			
i = 4	0	1	2	3	4	5			
	1	1	2	3	4				
	2	1	2	3	4				
i = 5	0	1	2	3	4				
	1	1	2	3					
i = 6	0	1	2	3					
		1	2						

Counting Sort

- Counting sort is a sorting algorithm that sorts the elements of an array by counting the number of occurrences of each unique element in the array.

Counting Sort

Database needed-

- Original Array/Input Array
- Count Array/Auxillary Array
  - Take a count array to store the count of each unique object.
  - Store the count of each element at their respective index in count array

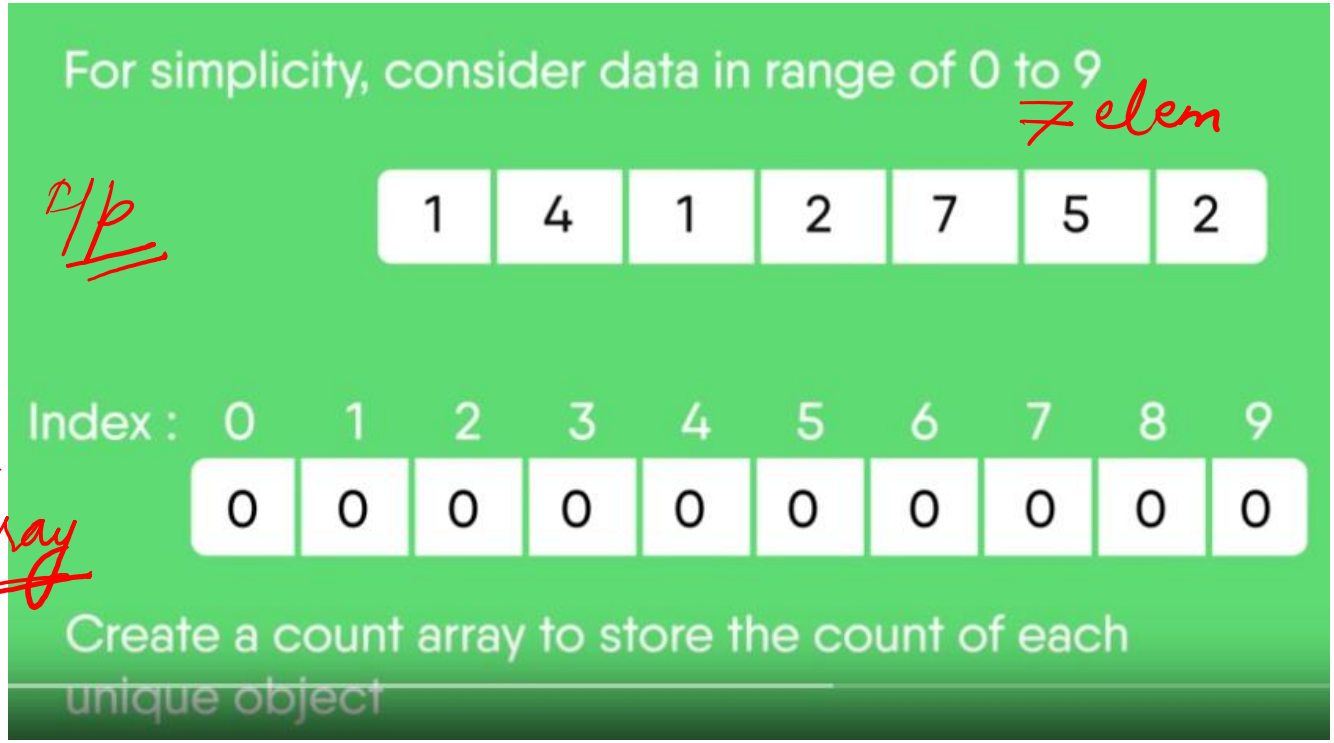
- For example: if the count of element 3 is 2 then, 2 is stored in the 3rd position of *count* array.
- If element "5" is not present in the array, then 0 is stored in 5th position.

Counting Sort

- Store cumulative sum of the elements of the count array. It helps in placing the elements into the correct index of the sorted array.
- The modified count array indicates the position of each object in the output sequence.
- After placing each element at its correct position, decrease its count by one.

Counting Sort

Let us take an **auxiliary array/count array** from 0 to 9 for simplicity



Counting Sort

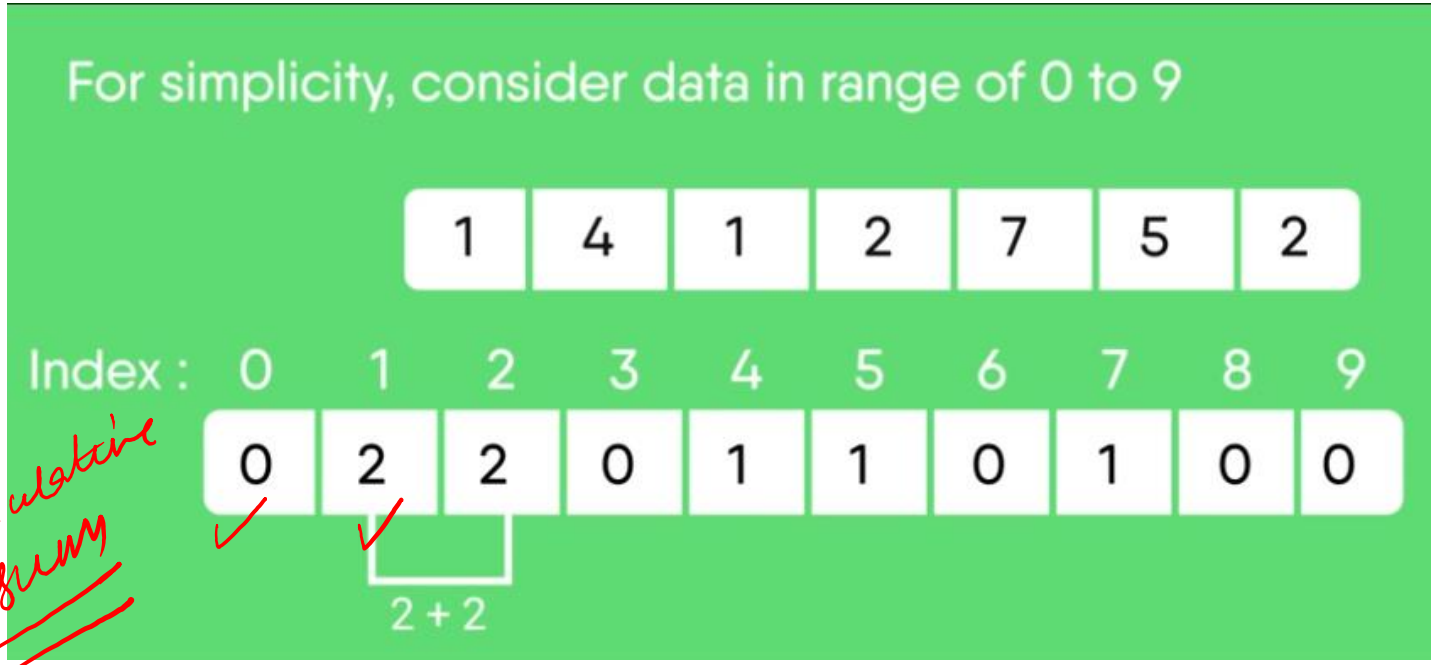
For simplicity, consider data in range of 0 to 9

1	4	1	2	7	5	2

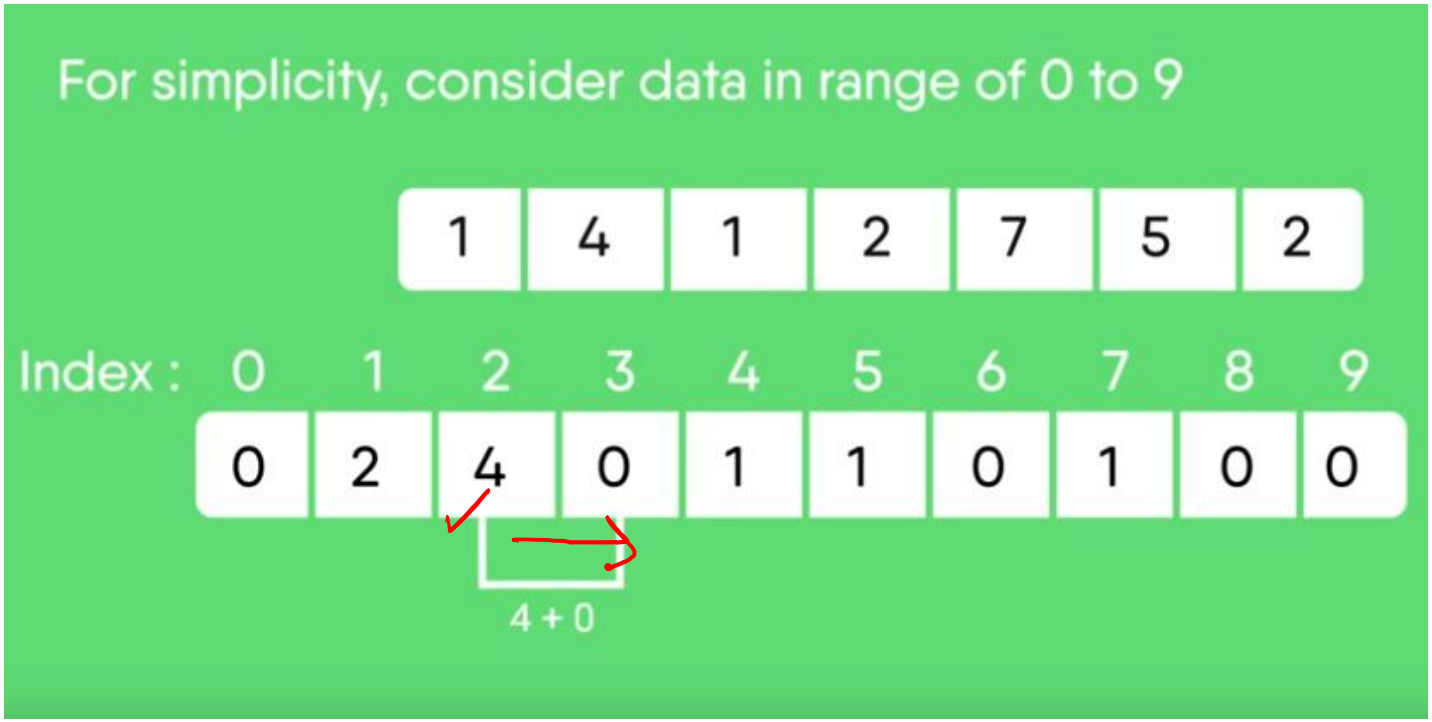
Index :	0	1	2	3	4	5	6	7	8	9
	0	2	2	0	1	1	0	1	0	0

Modify the count array by adding the previous counts.

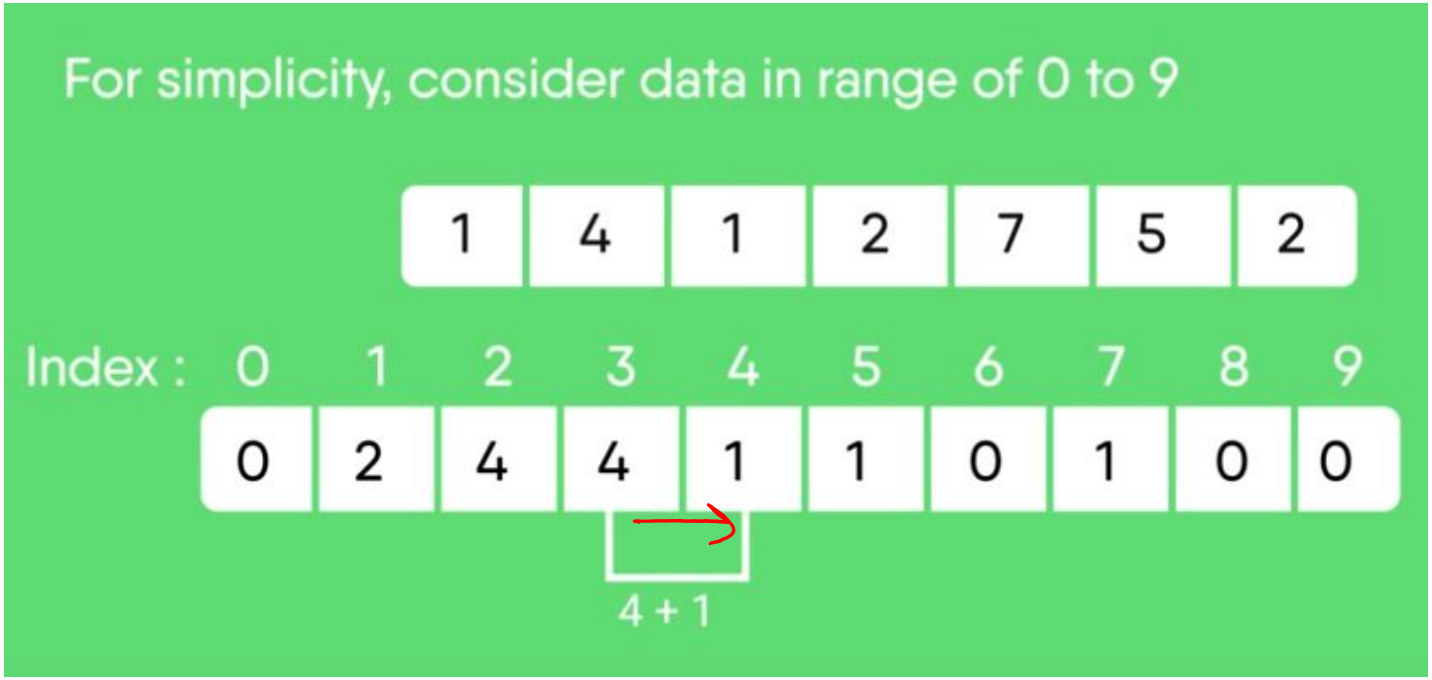
Counting Sort



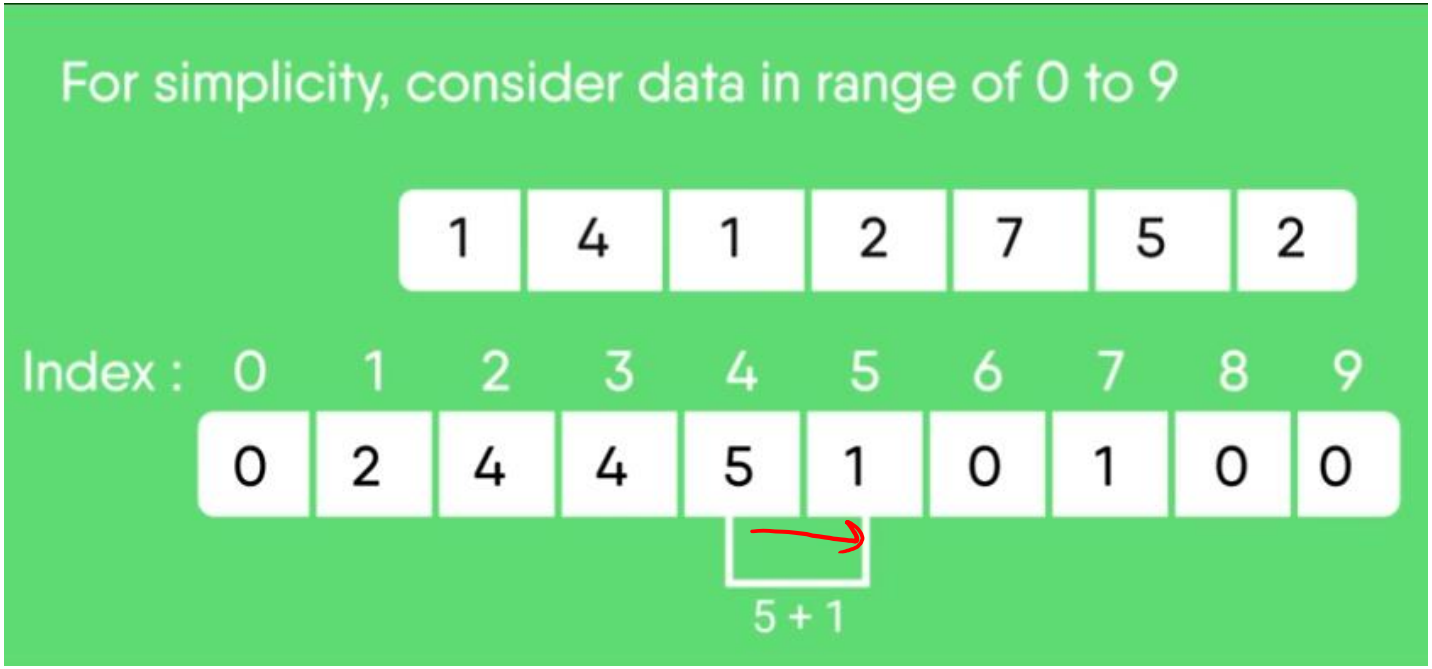
Counting Sort



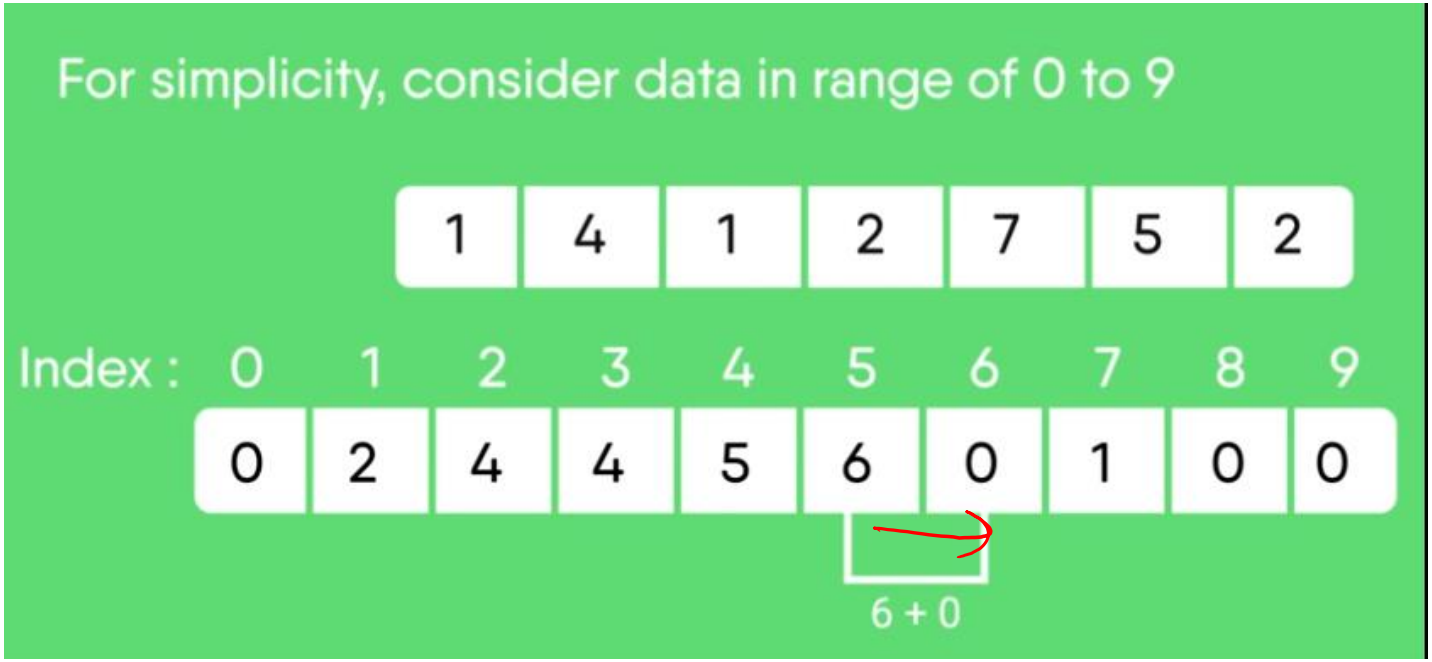
Counting Sort



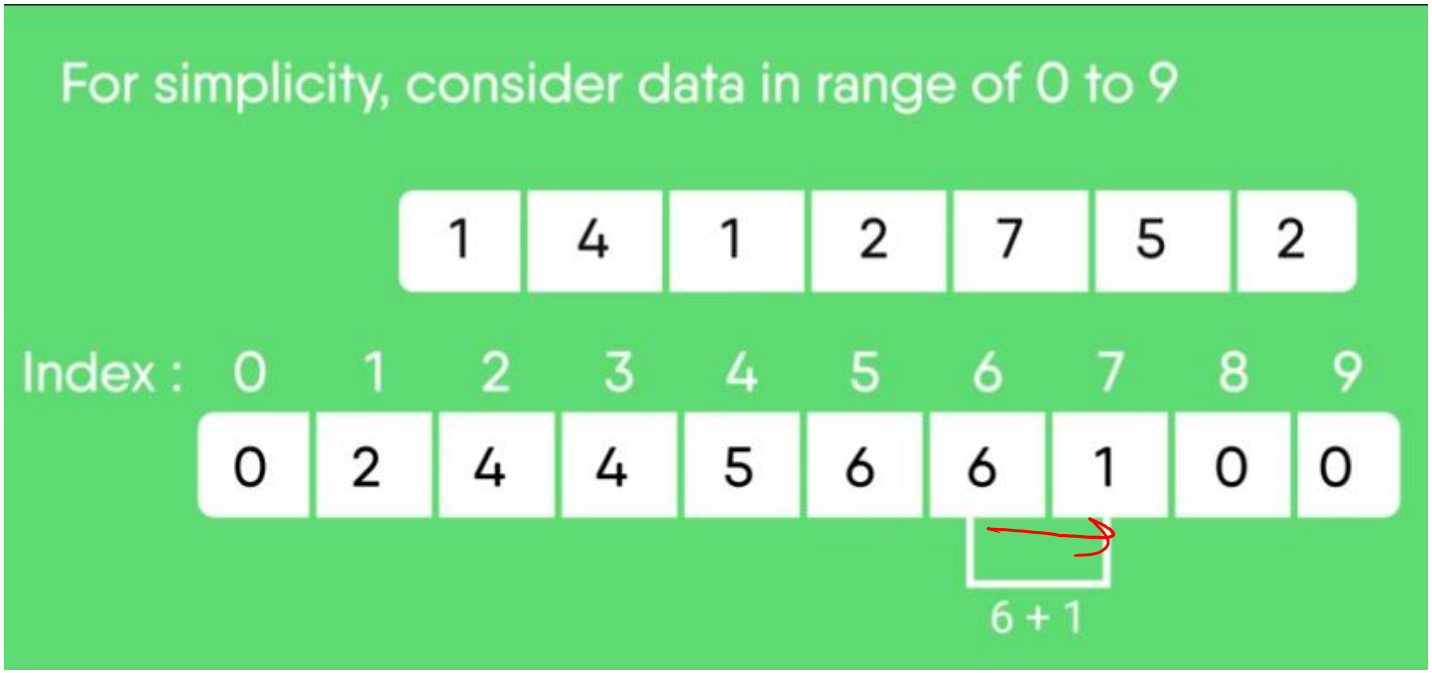
Counting Sort



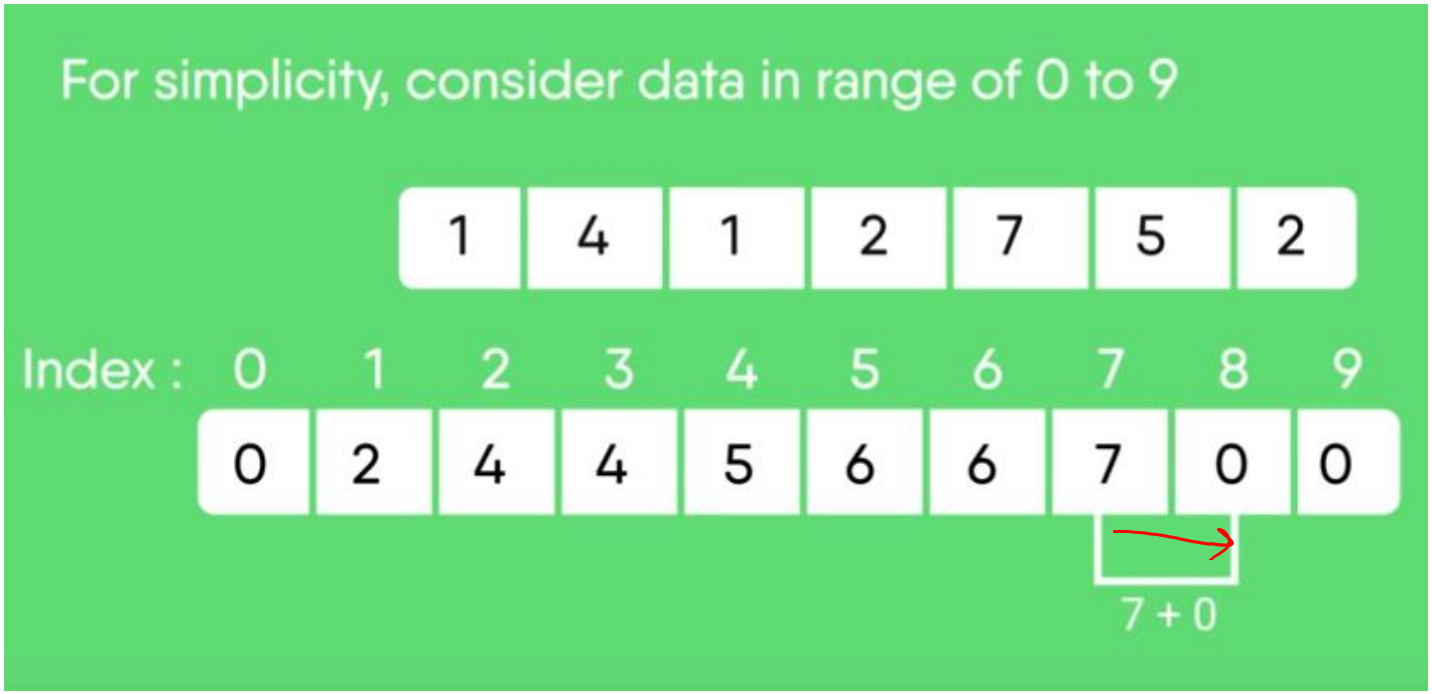
Counting Sort



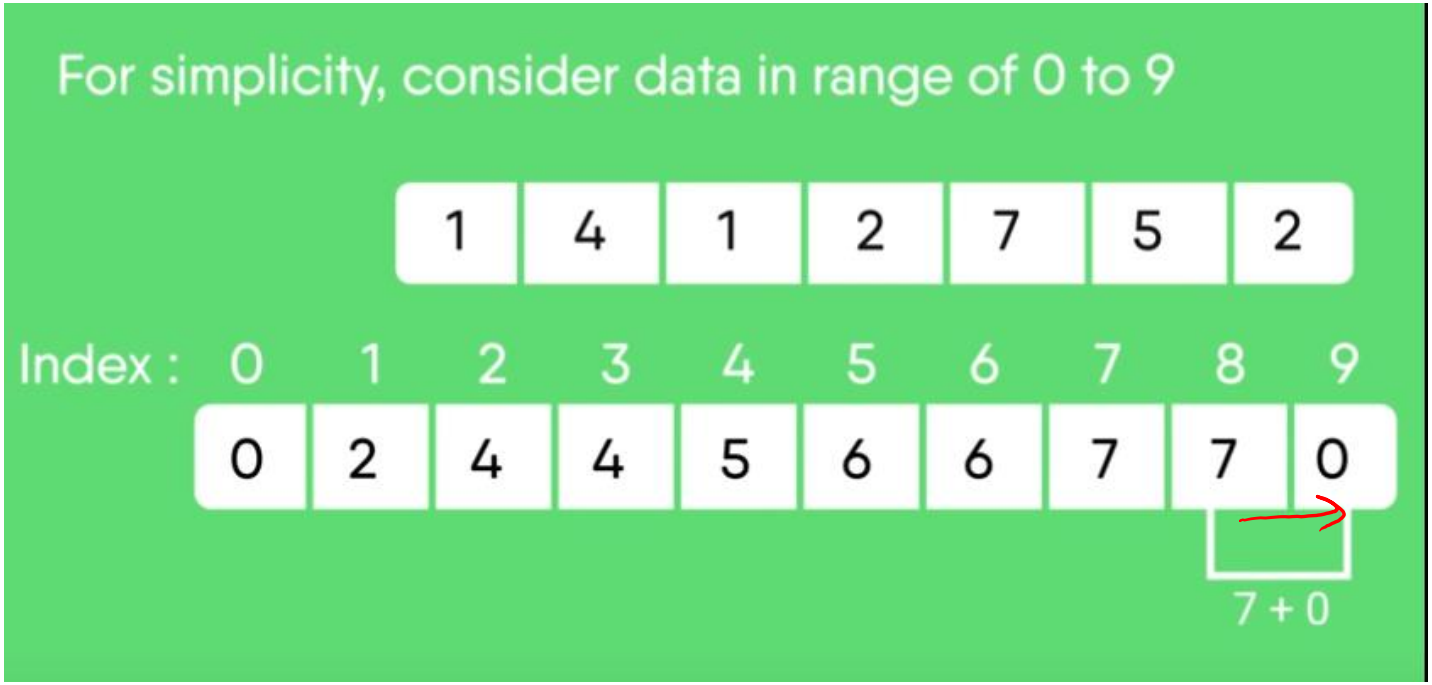
Counting Sort



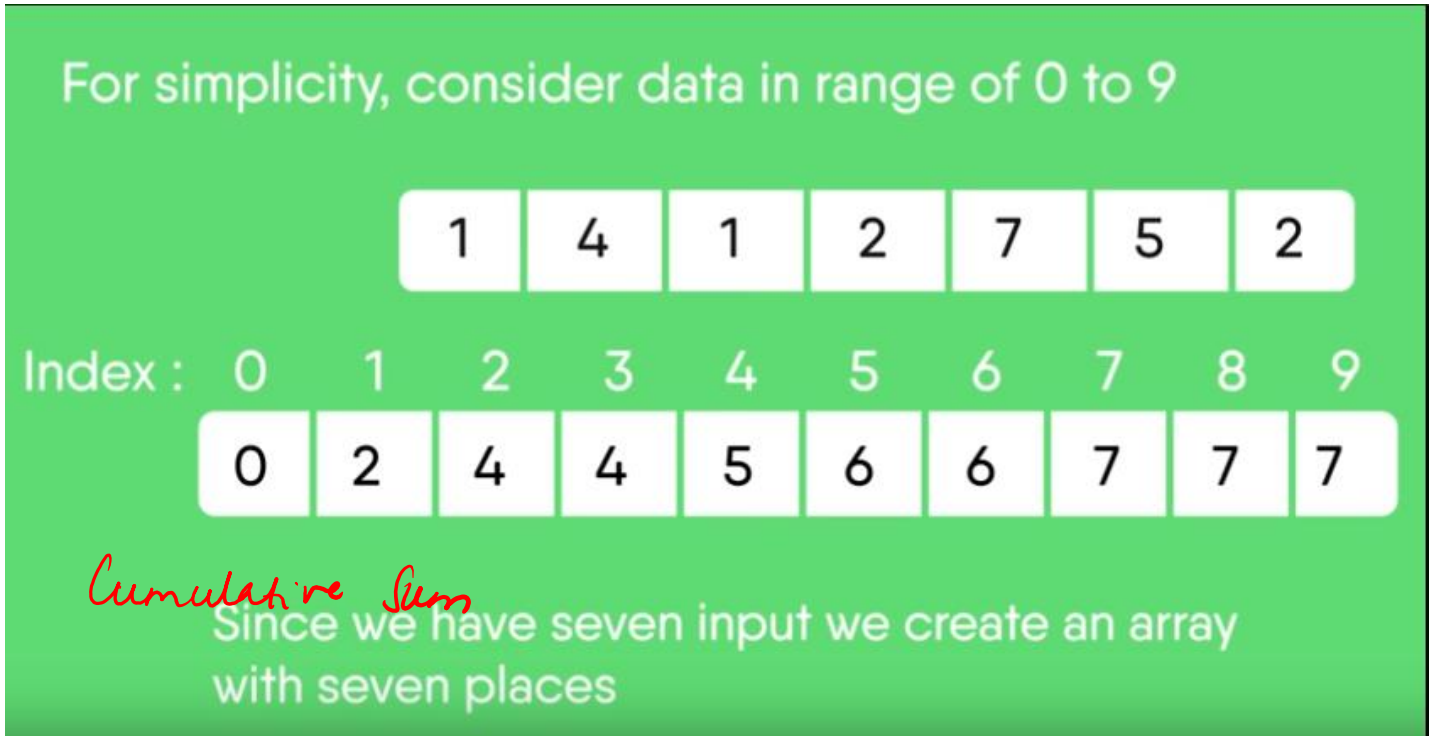
Counting Sort



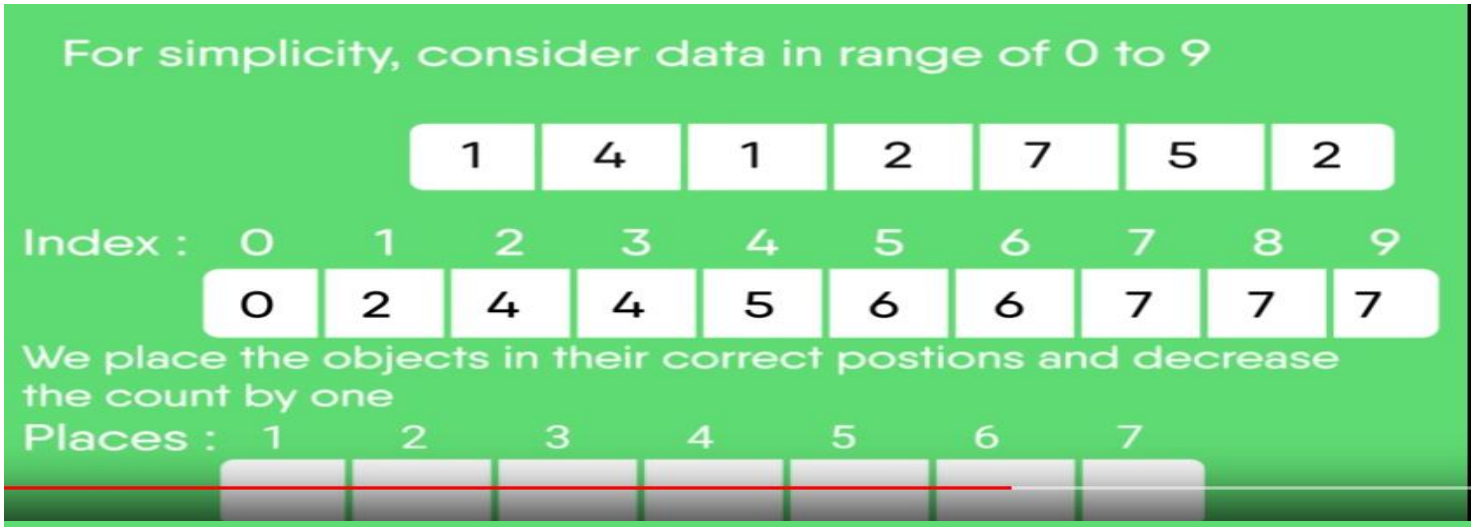
Counting Sort



Counting Sort



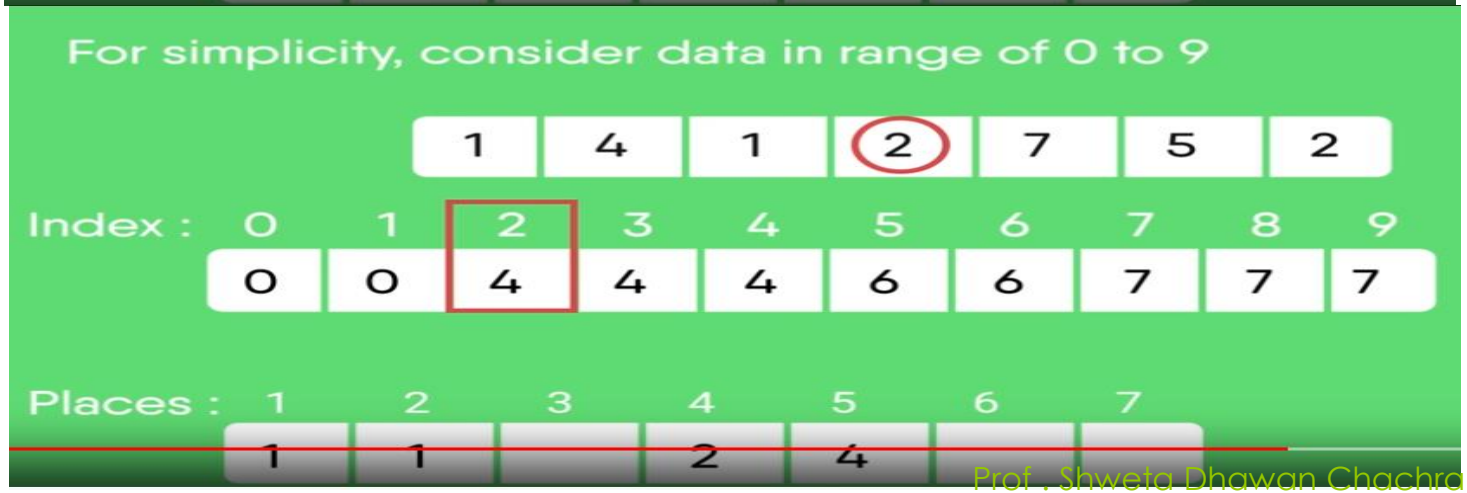
Counting Sort



Counting Sort

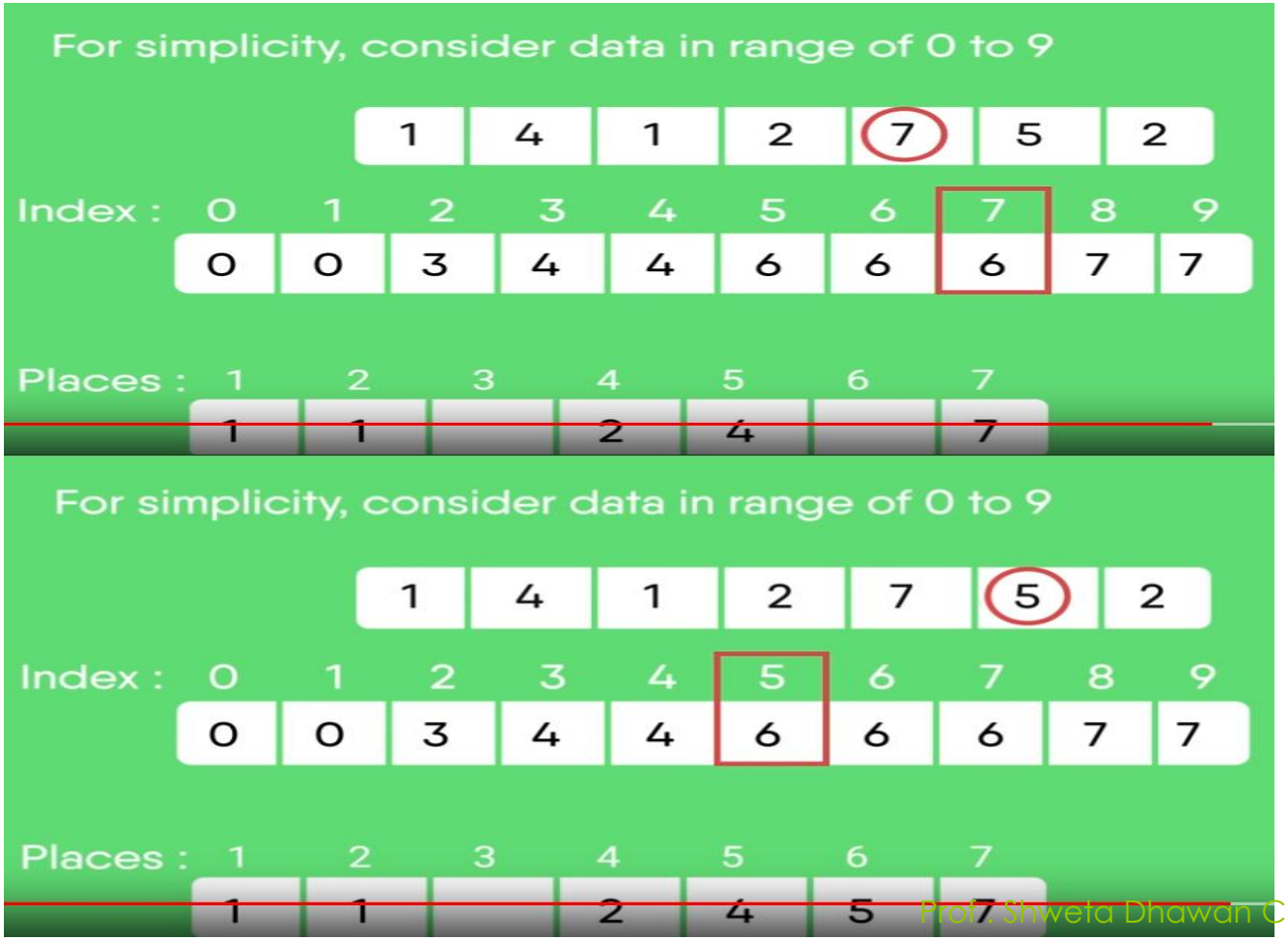
Counting Sort

Counting Sort

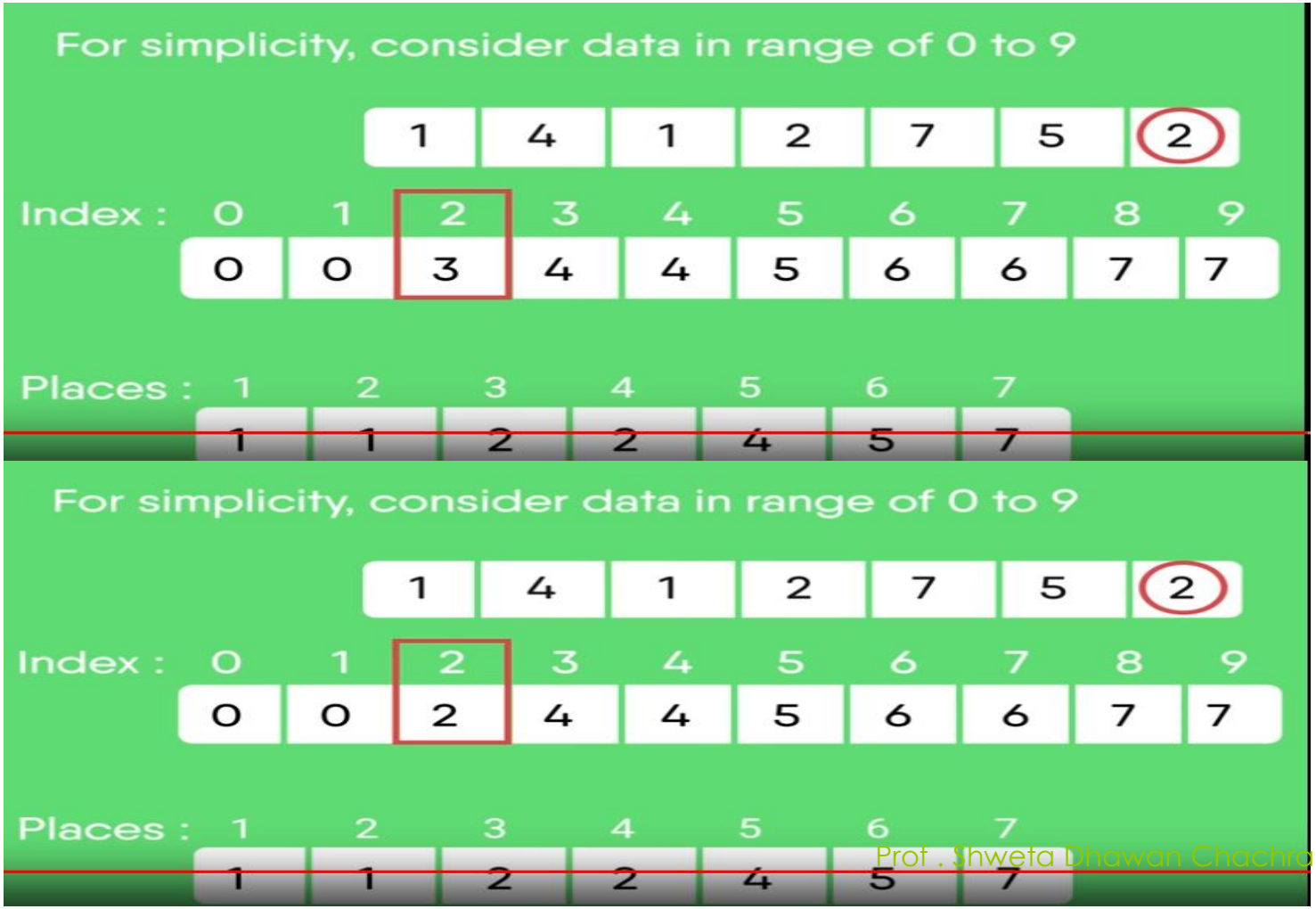


Counting Sort

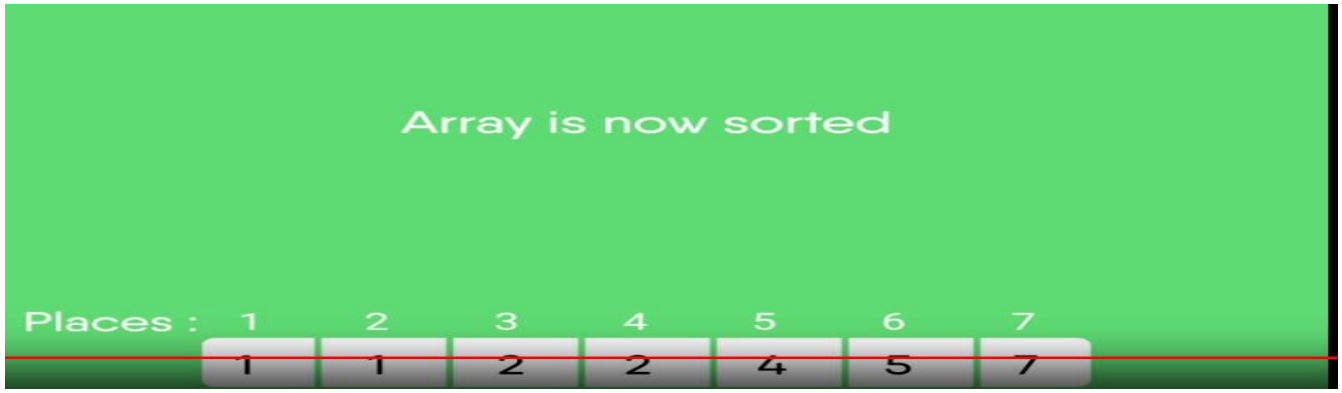
Counting Sort



Counting Sort



Counting Sort



Example 2

8	4	2	2	8	3	3	1

Sort the array of 8 elements using Counting Sort

Example 2

8	4	2	2	8	3	3	1

Let us take an auxiliary array/count array from 0 to 9 for simplicity

Example 2

8	4	2	2	8	3	3	1

Sort the array of 8 elements using Counting Sort

0	1	2	3	4	5	6	7	8	9
0	1	2	2	1	0	0	0	2	0

Example 2

8	4	2	2	8	3	3	1

Sort the array of 8 elements using Counting Sort

0	1	2	3	4	5	6	7	8	9
0	1	2	2	1	0	0	0	2	0

Cumulative Count array –

0	1	2	3	4	5	6	7	8	9
0	1	3	5	6	6	6	6	8	8

Example 2

8	4	2	2	8	3	3	1

Cumulative Count array –

0	1	2	3	4	5	6	7	8	9
0	1	3	5	6	6	6	6	8	8

- Place the objects in the correct place and decrease the count by 1
- Check for Object 8, Count=8,
- Place at 8th Position and decrement count by 1

1	2	3	4	5	6	7	8
							8

- Cumulative Count Array-

0	1	2	3	4	5	6	7	8	9
0	1	3	5	6	6	6	6	7	8

Example 2

8	4	2	2	8	3	3	1

Cumulative Count array –

0	1	2	3	4	5	6	7	8
0	1	3	5	6	6	6	6	7

- Place the objects in the correct place and decrease the count by 1
- Check for Object 4, Count=6,
- Place at 6th Position and decrement count by 1

1	2	3	4	5	6	7	8
					4		8

- Cumulative Count Array-

0	1	2	3	4	5	6	7	8
0	1	3	5	5	6	6	6	7

Example 2

8	4	2	2	8	3	3	1

Cumulative Count array –

0	1	2	3	4	5	6	7	8
0	1	3	5	5	6	6	6	7

- Place the objects in the correct place and decrease the count by 1
- Check for Object 2, Count=3,
- Place object at 3th Position and decrement count by 1

1	2	3	4	5	6	7	8
		2			4		8

- Cumulative Count Array-

0	1	2	3	4	5	6	7	8
0	1	2	5	5	6	6	6	7

Example 2

8	4	2	2	8	3	3	1

Cumulative Count array –

0	1	2	3	4	5	6	7	8
0	1	2	5	5	6	6	6	7

- Place the objects in the correct place and decrease the count by 1
- Check for Object 2, Count=2,
- Place object at 2nd Position and decrement count by 1

1	2	3	4	5	6	7	8
	2	2			4		8

- Cumulative Count Array-

0	1	2	3	4	5	6	7	8
0	1	1	5	5	6	6	6	7

Example 2

8	4	2	2	8	3	3	1

Cumulative Count array –

0	1	2	3	4	5	6	7	8
0	1	1	5	5	6	6	6	7

- Place the objects in the correct place and decrease the count by 1
- Check for Object 8, Count=7,
- Place object at 7th Position and decrement count by 1

1	2	3	4	5	6	7	8
	2	2			4	8	8

- Cumulative Count Array-

0	1	2	3	4	5	6	7	8
0	1	1	5	5	6	6	6	6

Example 2

8	4	2	2	8	3	3	1

Cumulative Count array –

0	1	2	3	4	5	6	7	8
0	1	1	5	5	6	6	6	7

- Place the objects in the correct place and decrease the count by 1
- Check for Object 3, Count=5,
- Place object at 5th Position and decrement count by 1

1	2	3	4	5	6	7	8
	2	2		3	4	8	8

- Cumulative Count Array-

0	1	2	3	4	5	6	7	8
0	1	1	4	5	6	6	6	6

Example 2

8	4	2	2	8	3	3	1

Cumulative Count array –

0	1	2	3	4	5	6	7	8
0	1	1	4	5	6	6	6	7

- Place the objects in the correct place and decrease the count by 1
- Check for Object 3, Count=4,
- Place object at 5th Position and decrement count by 1

1	2	3	4	5	6	7	8
	2	2	3	3	4	8	8

- Cumulative Count Array-

0	1	2	3	4	5	6	7	8
0	1	1	3	5	6	6	6	6

Example 2

8	4	2	2	8	3	3	1

Cumulative Count array –

0	1	2	3	4	5	6	7	8
0	1	1	4	5	6	6	6	7

- Place the objects in the correct place and decrease the count by 1
- Check for Object 1, Count=1,
- Place object at 1st Position and decrement count by 1

1	2	3	4	5	6	7	8
1	2	2	3	3	4	8	8

- Cumulative Count Array-

0	1	2	3	4	5	6	7	8
0	0	1	3	5	6	6	6	6

Example 2

Input Array-

8	4	2	2	8	3	3	1

Sorted array-

Position	1	2	3	4	5	6	7	8
Value	1	2	2	3	3	4	8	8

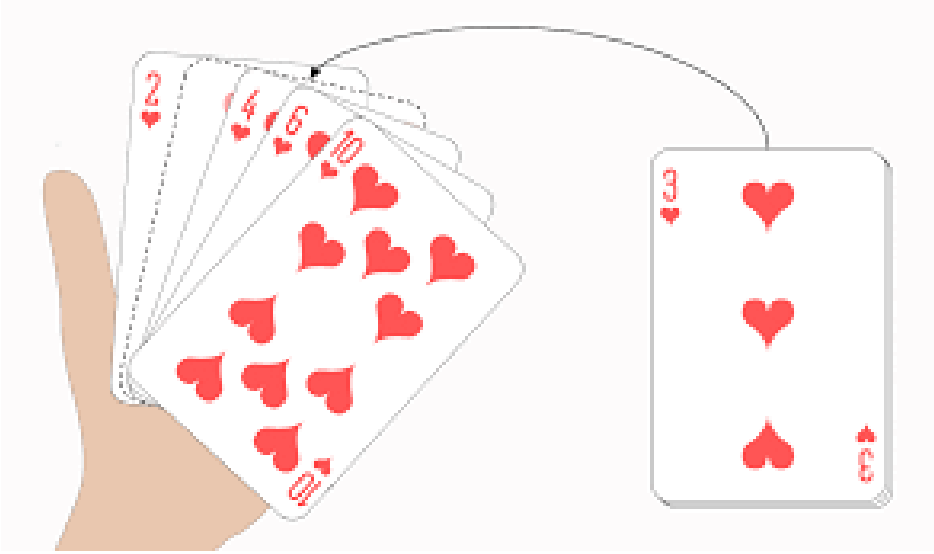
Insertion Sort-

Insertion Sort-

- Incremental algorithm.
- Considers the elements one at a time,
- Inserting each in its suitable place among those already considered (keeping them sorted).
- It builds the sorted sequence one number at a time.

Insertion Sort-

- Similar to the way you sort playing cards in your hands.
- The array is virtually split into a sorted and an unsorted part.
- Values from the unsorted part are picked and
- Placed at the correct position in the sorted part.



Insertion Sort-

Suppose a[0], a[1], a[2], ……,a[n-1] are n elements in memory, insertion sort works as follow:

Pass 1: a[0] by itself is trivially sorted.

Pass 2: a[1] is inserted either
before or after a[0] so that a[0], a[1] is sorted.

Pass 3: a[2] is inserted into its proper place in a[0],a[1],
that is before a[0],
between a[0] and a[1],
or after a[1],
so that: a[0],a[1], a[2] is sorted.

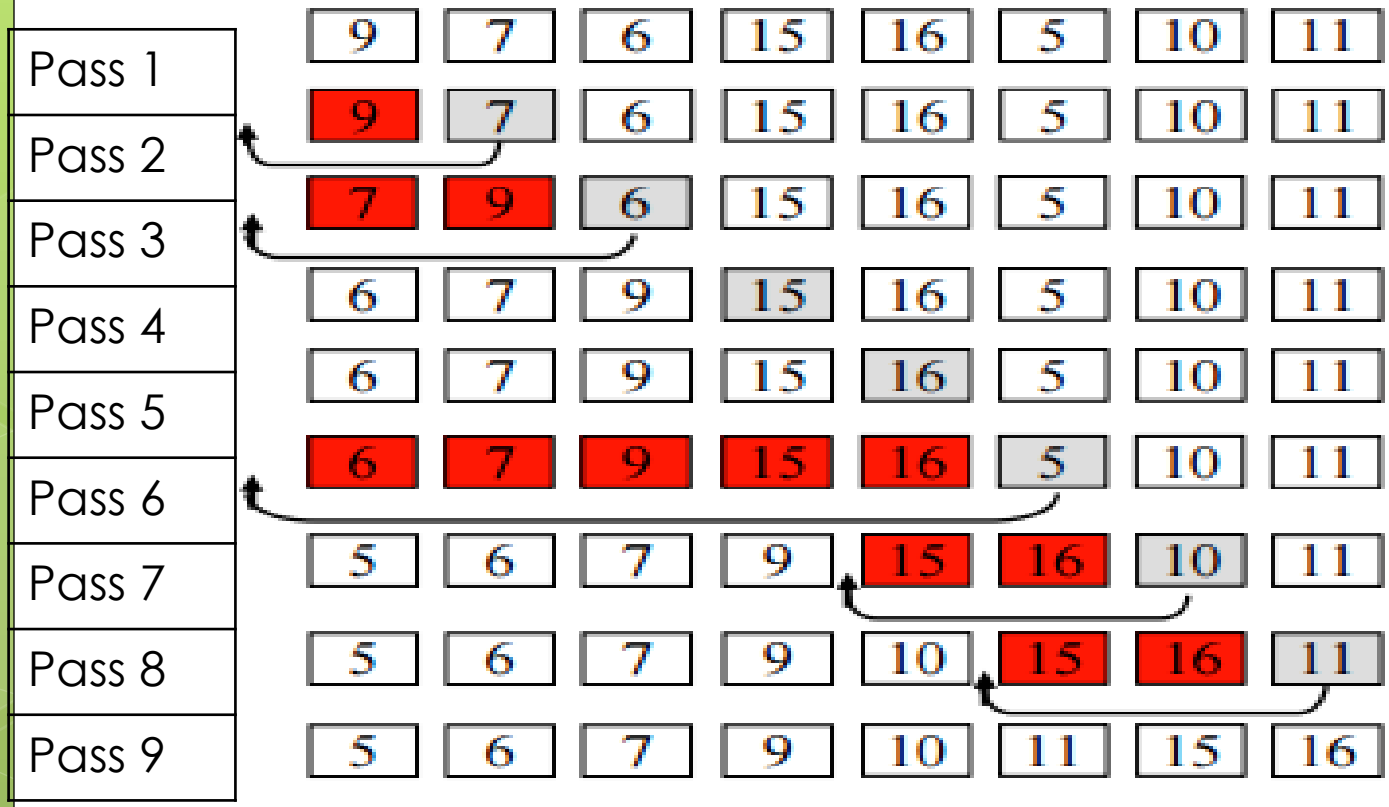
……

……

Pass n: a[n-1] is inserted into its proper place in a[0], a[1], a[2], ……a[n-2] so that a[0], a[1], a[2], ……a[n-1] is sorted array with 'n' elements

Insertion Sort-

A list of unsorted elements are: 9, 7, 6, 15, 16, 5, 10, 11



Algorithm for Insertion Sort-

- Step 1 – If it is the first element, it is already sorted. return 1;
- Step 2 – Iterate from arr[1] to arr[n-1] over the array.
- Pick the current element and store it separately in a **key.**
- Step 3 – Compare the key with its predecessors i.e. all elements in the sorted sub-list
- Step 4 – Shift all the elements in the sorted sub-list that are greater than the key towards the right by one position to make space for the key
- Step 5 – Insert the key
- Step 6 – Repeat until list is sorted