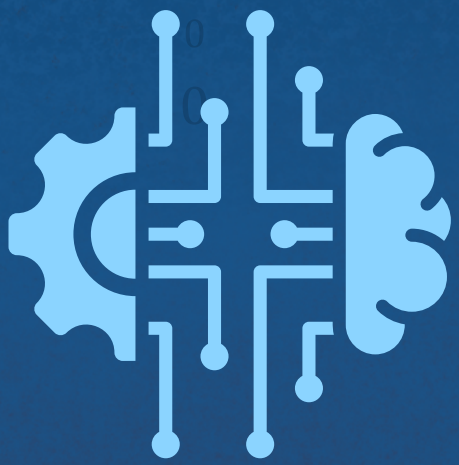
DSA NOTES

Part- 2



FOLLOW US!

www.topperworld.in

Searching Algorithm

Searching in Data Structures

Searching is a crucial operation in computer science that **involves finding a specific element within a collection of data**. It's like looking for a particular book in a library or finding a contact in your phone's address book. In the world of programming, efficient searching can make a significant difference in performance and resource utilization.

➔Types of Searching Algorithms

1. Linear Search

Linear search is a **simple** and **straightforward** algorithm. Imagine you're looking for a name in a list of names; you start from the beginning and go through each name one by one until you find the one you're looking for.

Algorithm for Linear Search:

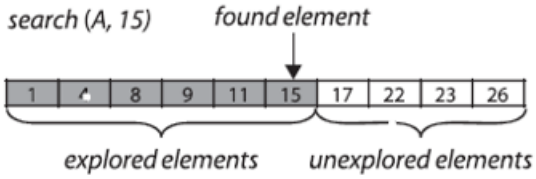
SEQUENTIAL SEARCH

Best	Average	Worst
O (1)	O (n)	O (n)

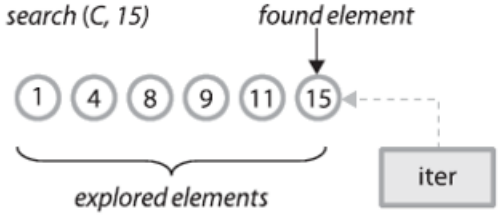
Array

Brute Force

```
search (A, t)
1.    for i = 0 to n − 1 do
2.      if (A[i] = t) then
3.          return true
4.    return false
end
```




```
search (C, t)
1.    iter = C.begin()
2.    while (iter ≠ C.end()) do
3.      e = next element from iter
4.      if (e = t) then
5.          return true
6.    return false
end
```



Steps :

1. Start at the beginning of the list.
2. Compare the current element with the target element.
3. If the current element matches the target, you've found it!
4. If not, move to the next element and repeat steps 2 and 3.
5. If you've checked all elements and haven't found the target, it's not in the list.

➔Java Implementation of Linear Search:

```
public class LinearSearch {
    public static int linearSearch(int[] arr, int target) {
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == target) {
                return i; // Element found at index i
            }
        }
        return -1; // Element not found
    }

    public static void main(String[] args) {
        int[] array = { 3, 9, 4, 7, 2, 1 };
        int targetElement = 7;
        int resultIndex = linearSearch(array, targetElement);

        if (resultIndex != -1) {
            System.out.println("Element found at index: " +
resultIndex);
        } else {
            System.out.println("Element not found.");
        }
    }
```

```
}
```

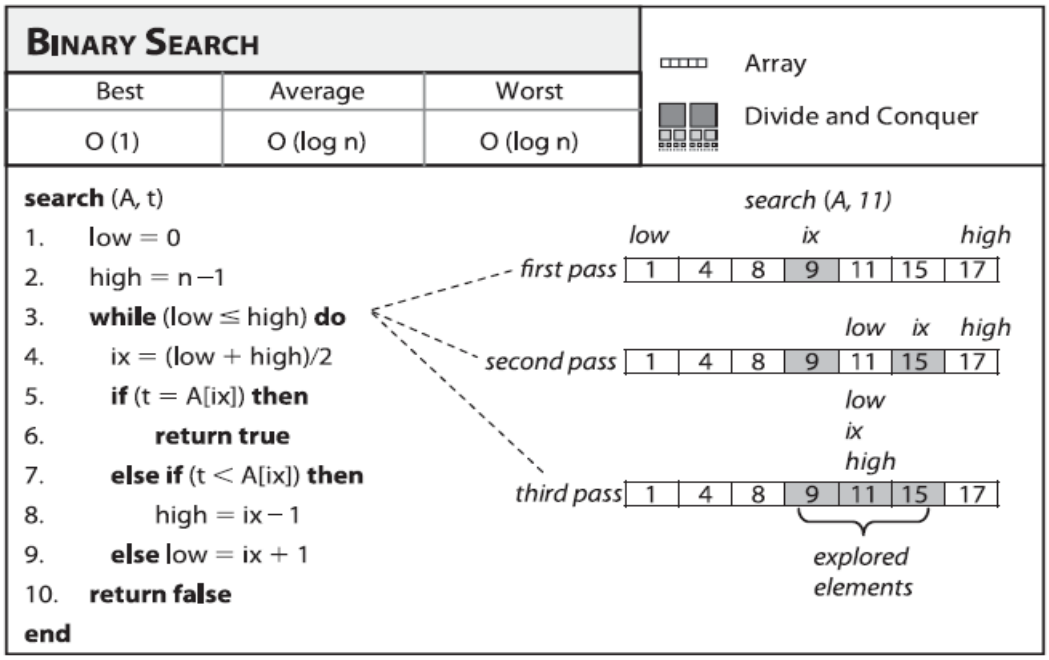
OUTPUT:

```
Element found at index: 3
```

2. Binary Search

Binary search is a more sophisticated algorithm that relies on the fact that the data is sorted. It's like finding a word in a dictionary by flipping pages in half.

Algorithm for Binary Search:



Steps:

1. Start with the entire sorted collection.
2. Calculate the middle index of the current range.
3. Compare the element at the middle index with the target element.
4. If the middle element matches the target, you've found it!
5. If the middle element is greater than the target, narrow down the search to the left half.
6. If the middle element is less than the target, narrow down the search to the right half.

7. Repeat steps 2 to 6 until the target element is found or the search range is empty.

➔Java Implementation of Binary Search:

```
public class BinarySearch {
    public static int binarySearch(int[] arr, int target) {
        int left = 0;
        int right = arr.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (arr[mid] == target) {
                return mid; // Element found at index mid
            } else if (arr[mid] < target) {
                left = mid + 1; // Search the right half
            } else {
                right = mid - 1; // Search the left half
            }
        }

        return -1; // Element not found
    }
    public static void main(String[] args) {
        int[] array = { 1, 2, 3, 4, 7, 9 };
        int targetElement = 7;
        int resultIndex = binarySearch(array, targetElement);

        if (resultIndex != -1) {
            System.out.println("Element found at index: " +
resultIndex);
        } else {
            System.out.println("Element not found.");
        }
    }
}
```

OUTPUT:-

```
Element found at index: 4
```

➔Difference between Linear Search and Binary Search

- **Search Approach:**
  - Linear Search: Sequentially checks each element in the collection.
  - Binary Search: Divides the search range in half at each step.
- **Data Requirement:**
  - Linear Search: Works on both sorted and unsorted data.
  - Binary Search: Requires sorted data.
- **Time Complexity:**
  - Linear Search: O(n) in the worst case (n is the number of elements).
  - Binary Search: O(log n) in the worst case.
- **Efficiency:**
  - Binary Search is generally faster for larger datasets.
- **Applicability:**
  - Linear Search is suitable for small datasets or when data is unsorted.
  - Binary Search shines when dealing with sorted data and larger datasets.

Remember, the choice of algorithm depends on the characteristics of your data and your performance requirements. Linear search is easy to understand but may be slower for larger data, while binary search is efficient for sorted data but requires the data to be sorted.

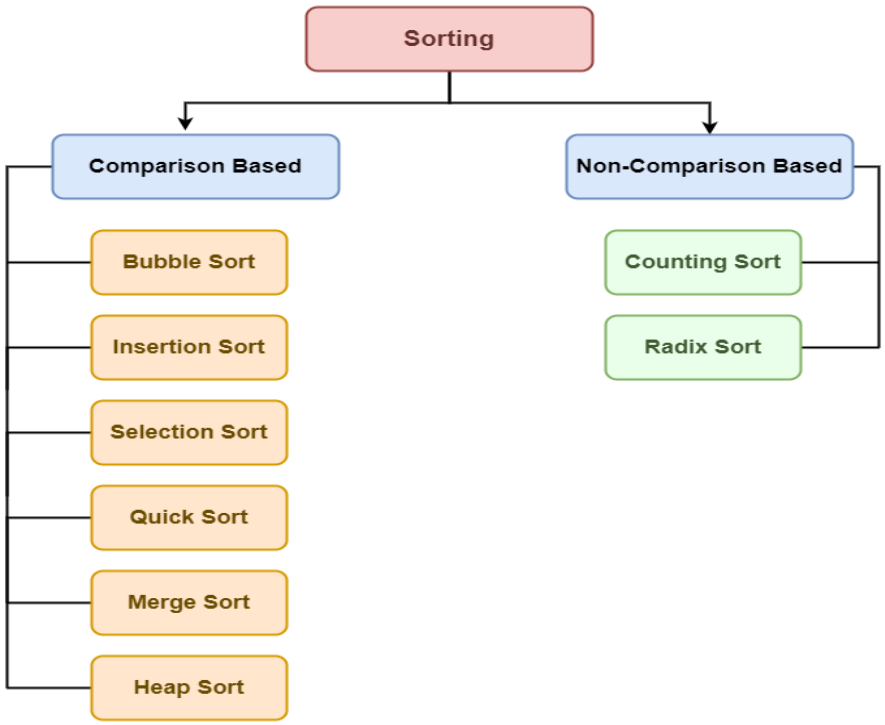
Sorting Algorithm

Sorting refers to rearrangement of a given array or list of elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.
When we have a **large amount of data**, it can be difficult to deal with it, especially when it is arranged randomly. When this happens, sorting that data becomes crucial. It is necessary to sort data in order to make searching easier.

Types of Sorting Techniques

There are various sorting algorithms are used in data structures. The following two types of sorting algorithms can be broadly classified:

1. **Comparison-based:** We compare the elements in a comparison-based sorting algorithm)
2. **Non-comparison-based:** We do not compare the elements in a non-comparison-based sorting algorithm)



Sorting algorithm

Why Sorting Algorithms are Important

The sorting algorithm is important in Computer Science because it reduces the complexity of a problem. There is a wide range of applications for these algorithms, including searching algorithms, database algorithms, divide and conquer methods, and data structure algorithms.

In the following sections, we list some important scientific applications where sorting algorithms are used

- When you have hundreds of datasets you want to print, you might want to arrange them in some way.
- Sorting algorithm is used to arrange the elements of a list in a certain order (either ascending or descending).
- Searching any element in a huge data set becomes easy. We can use Binary search method for search if we have sorted data. So, Sorting become important here.
- They can be used in software and in conceptual problems to solve more advanced problems.

Some of the most common sorting algorithms are:

Below are some of the most common sorting algorithms:

1. Selection Sort

Selection sort is another sorting technique in which we find the minimum element in every iteration and place it in the array beginning from the first index. Thus, a selection sort also gets divided into a sorted and unsorted subarray.

Advantages and Disadvantages of Selection Sort

Advantages	Disadvantages
The main advantage of the selection sort is that it performs well on a small list.	The primary disadvantage of the selection sort is its poor efficiency when dealing with a huge list of items.
Because it is an in-place sorting algorithm, no additional temporary storage is required beyond what is needed to hold the original list.	The selection sort requires n-squared number of steps for sorting n elements.
Its performance is easily influenced by the initial ordering of the items before the sorting process.	Quick Sort is much more efficient than selection sort

Working of Selection Sort algorithm:

Lets consider the following array as an example: arr[] = {64, 25, 12, 22, 11}

First pass:
For the first position in the sorted array, the whole array is traversed from index 0 to 4 sequentially. The first position where 64 is stored presently, after traversing whole array it is clear that 11 is the lowest value.

64	25	12	22	11

Thus, replace 64 with 11. After one iteration 11, which happens to be the least value in the array, tends to appear in the first position of the sorted list.

11	25	12	22	64

Second Pass:
For the second position, where 25 is present, again traverse the rest of the array in a sequential manner.

11	25	12	22	64

After traversing, we found that 12 is the second lowest value in the array and it should appear at the second place in the array, thus swap these values.

11	12	25	22	64

Third Pass:
Now, for third place, where 25 is present again traverse the rest of the array and find the third least value present in the array.

11	12	25	22	64

While traversing, 22 came out to be the third least value and it should appear at the third place in the array, thus swap 22 with element present at third position.

11	12	22	25	64

Fourth pass:

Similarly, for fourth position traverse the rest of the array and find the fourth least element in the array
As 25 is the 4th lowest value hence, it will place at the fourth position.

11	12	22	25	64

Fifth Pass:
At last the largest value present in the array automatically get placed at the last position in the array
The resulted array is the sorted array.

11	12	22	25	64

2. Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.

Advantages and Disadvantages of Bubble Sort

Advantages	Disadvantages
The primary advantage of the bubble sort is that it is popular and easy to implement.	The main disadvantage of the bubble sort is the fact that it does not deal well with a list containing a huge number of items.
In the bubble sort, elements are swapped in place without using additional temporary storage.	The bubble sort requires n-squared processing steps for every n number of elements to be sorted.
The space requirement is at a minimum	The bubble sort is mostly suitable for academic teaching but not for real-life applications.

Working of Bubble Sort algorithm:

Lets consider the following array as an example: arr[] = {5, 1, 4, 2, 8}

First Pass:

Bubble sort starts with very first two elements, comparing them to check which one is greater.
(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:

Now, during second iteration it should look like this:

(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)

Third Pass:

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.

(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)

Illustration:

		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						

Illustration of Bubble Sort

3. Insertion Sort

Insertion sort is a simple sorting algorithm that works similarly 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.

Advantages and Disadvantages of Insertion Sort

Advantages	Disadvantages
The main advantage of the insertion sort is its simplicity.	The disadvantage of the insertion sort is that it does not perform as well as other, better sorting algorithms
It also exhibits a good performance when dealing with a small list.	With n-squared steps required for every n element to be sorted, the insertion sort does not deal well with a huge list.
The insertion sort is an in-place sorting algorithm so the space requirement is minimal.	The insertion sort is particularly useful only when sorting a list of few items.

Working of Insertion Sort algorithm:

Consider an example: arr[]: {12, 11, 13, 5, 6}

12	11	13	5	6

First Pass:

- Initially, the first two elements of the array are compared in insertion sort.

12	11	13	5	6

- Here, 12 is greater than 11 hence they are not in the ascending order and 12 is not at its correct position. Thus, swap 11 and 12.
- So, for now 11 is stored in a sorted sub-array.

11	12	13	5	6

Second Pass:

- Now, move to the next two elements and compare them

11	12	13	5	6

- Here, 13 is greater than 12, thus both elements seems to be in ascending order, hence, no swapping will occur. 12 also stored in a sorted sub-array along with 11

Third Pass:

- Now, two elements are present in the sorted sub-array which are 11 and 12
- Moving forward to the next two elements which are 13 and 5

11	12	13	5	6

- Both 5 and 13 are not present at their correct place so swap them

11	12	5	13	6

- After swapping, elements 12 and 5 are not sorted, thus swap again

11	5	12	13	6

- Here, again 11 and 5 are not sorted, hence swap again

5	11	12	13	6

- here, it is at its correct position

Fourth Pass:

- Now, the elements which are present in the sorted sub-array are 5, 11 and 12
- Moving to the next two elements 13 and 6

5	11	12	13	6

- Clearly, they are not sorted, thus perform swap between both

5	11	12	6	13

- Now, 6 is smaller than 12, hence, swap again

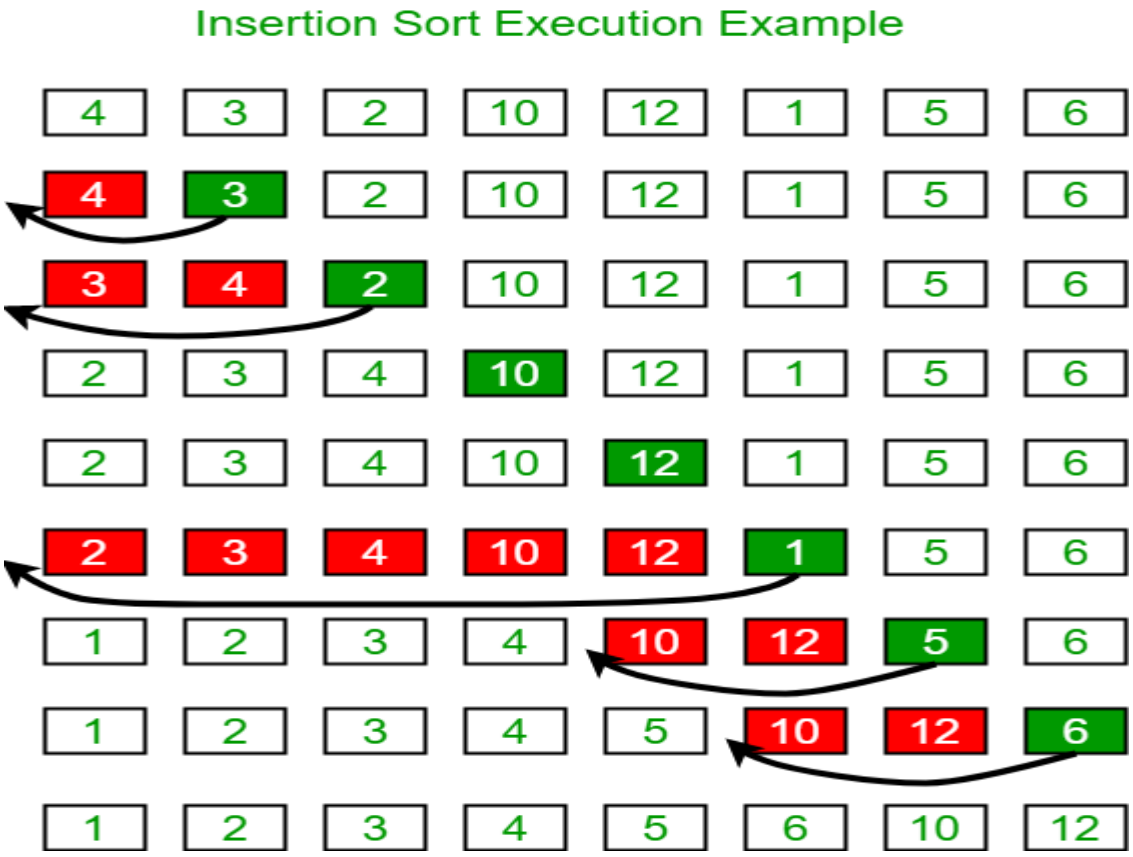
5	11	6	12	13

- Here, also swapping makes 11 and 6 unsorted hence, swap again

5	6	11	12	13

- Finally, the array is completely sorted.

Illustrations:



4. Merge Sort

The **Merge Sort** algorithm is a sorting algorithm that is based on the Divide and Conquers paradigm. In this algorithm, the array is initially divided into two equal halves and then they are combined in a sorted manner.

Let's see how Merge Sort uses Divide and Conquer:

The merge sort algorithm is an implementation of the divide and conquers technique. Thus, it gets completed in three steps:

1. Divide: In this step, the array/list divides itself recursively into sub-arrays until the base case is reached.
2. Conquer: Here, the sub-arrays are sorted using recursion.

3. Combine: This step makes use of the merge( ) function to combine the sub-arrays into the final sorted array.

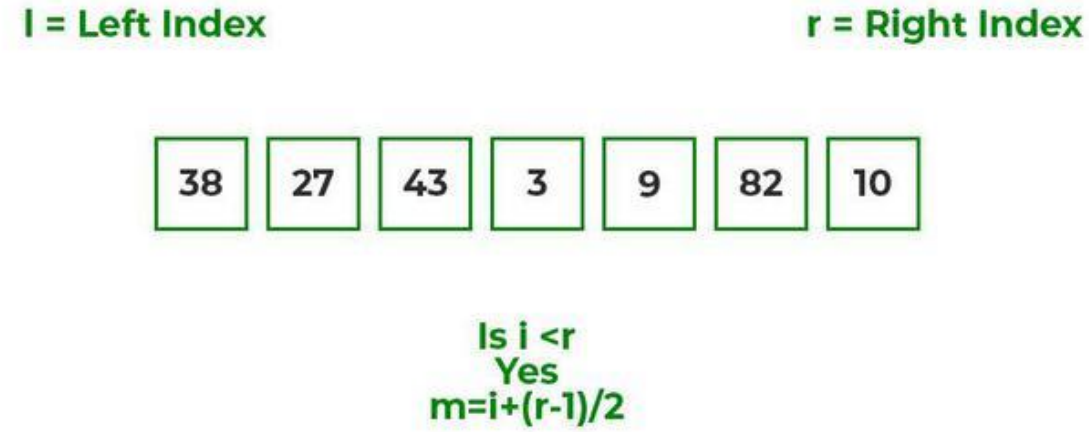
Advantages and Disadvantages of Merge Sort

Advantages	Disadvantages
It can be applied to files of any size.	Requires extra space »N
Reading of the input during the run-creation step is sequential ==> Not much seeking.	Merge Sort requires more space than other sort.
If heap sort is used for the in-memory part of the merge, its operation can be overlapped with I/O	Merge sort is less efficient than other sort

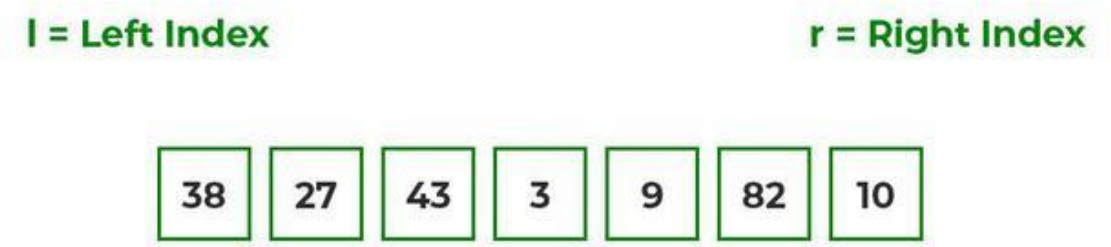
Working of Merge Sort algorithm:

To know the functioning of merge sort, lets consider an array arr[] = {38, 27, 43, 3, 9, 82, 10}

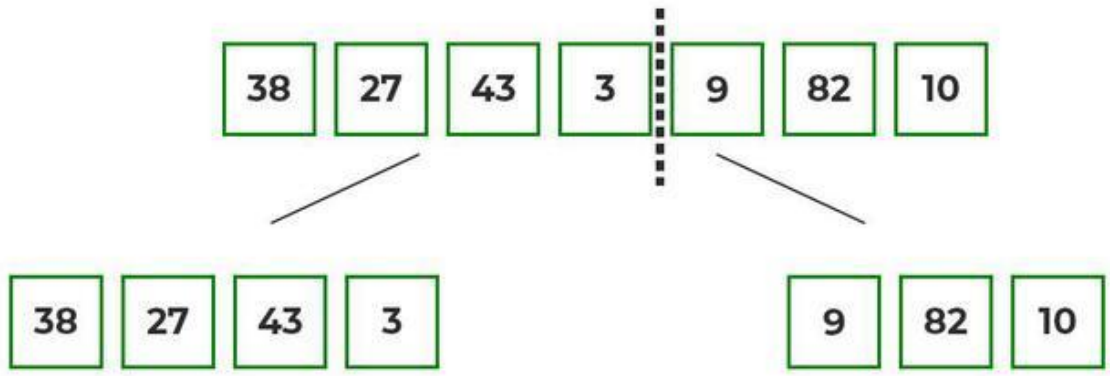
At first, check if the left index of array is less than the right index, if yes then calculate its mid point



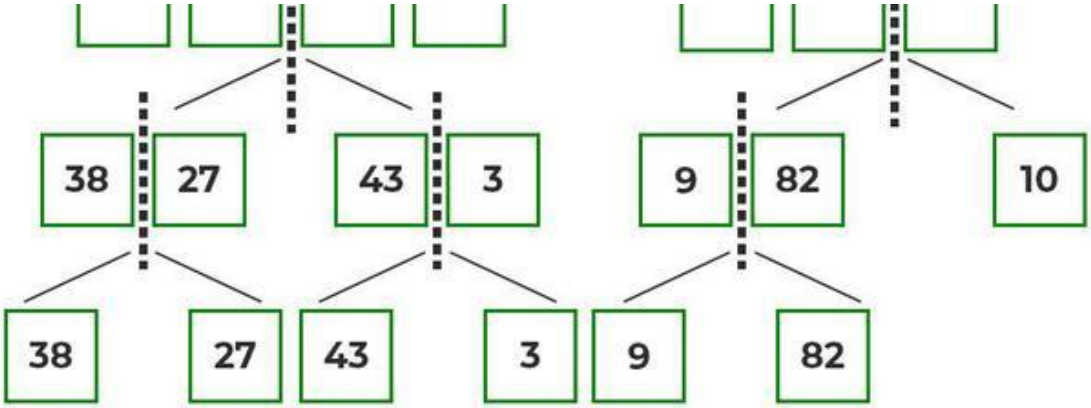
Now, as we already know that merge sort first divides the whole array iteratively into equal halves, unless the atomic values are achieved. Here, we see that an array of 7 items is divided into two arrays of size 4 and 3 respectively.



Now, again find that is left index is less than the right index for both arrays, if found yes, then again calculate mid points for both the arrays.

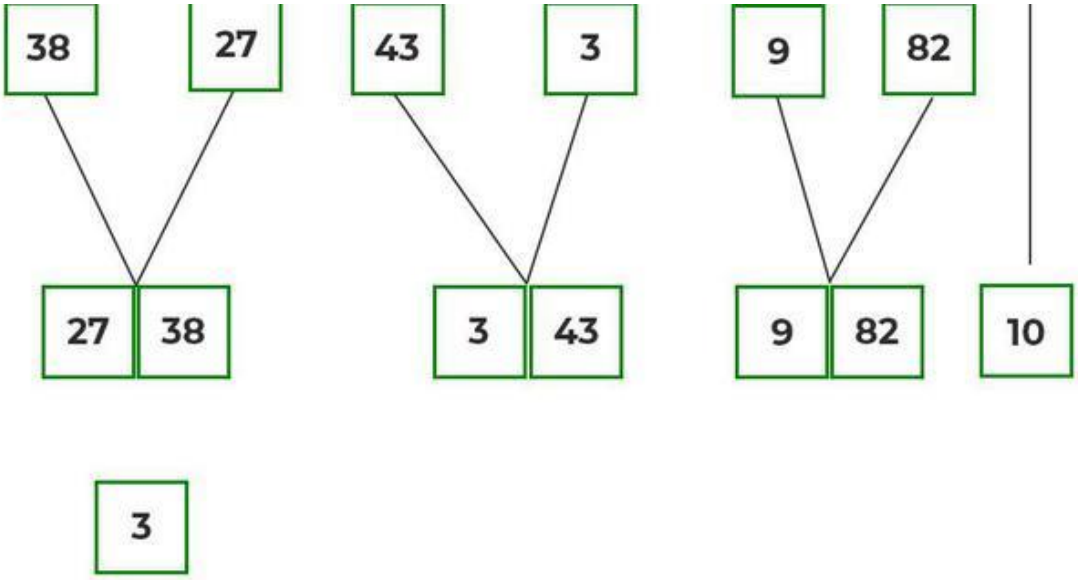


Now, further divide these two arrays into further halves, until the atomic units of the array is reached and further division is not possible.

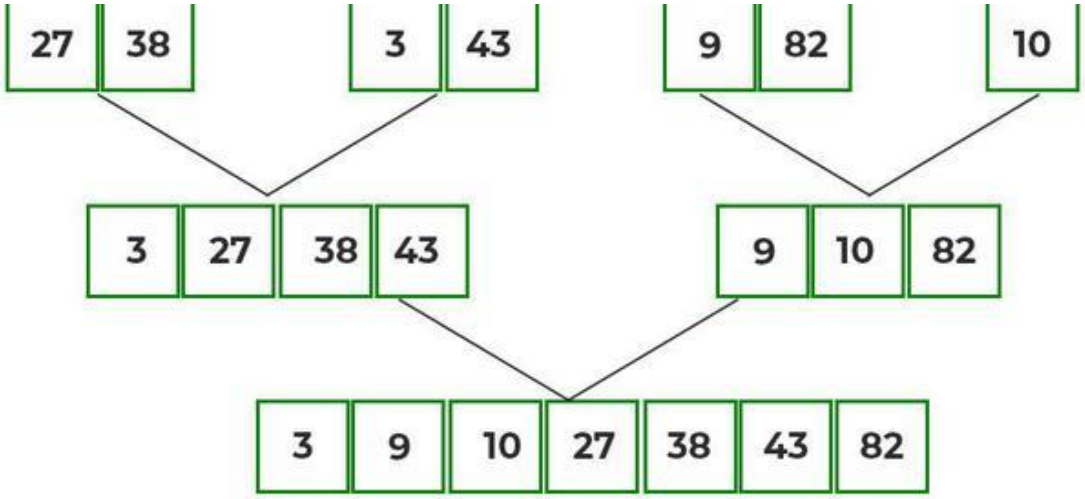


After dividing the array into smallest units merging starts, based on comparison of elements.

After dividing the array into smallest units, start merging the elements again based on comparison of size of elements Firstly, compare the element for each list and then combine them into another list in a sorted manner.



After the final merging, the list looks like this:



5 Quick Sort

Quicksort is a sorting algorithm based on the divide and conquer approach where an array is divided into subarrays by selecting a pivot element (element selected from the array).

1. While dividing the array, the pivot element should be positioned in such a way that elements less than the pivot are kept on the left side, and elements greater than the pivot is on the right side of the pivot.
2. The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.
3. At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

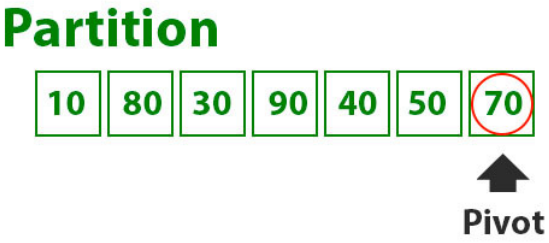
Advantages and Disadvantages of Quick Sort

Advantages	Disadvantages
The quick sort is regarded as the best sorting algorithm.	The slight disadvantage of quick sort is that its worst-case performance is similar to average performances of the bubble, insertion or selections sorts.
It is able to deal well with a huge list of items.	If the list is already sorted than bubble sort is much more efficient than quick sort
Because it sorts in place, no additional storage is required as well	If the sorting element is integers than radix sort is more efficient than quick sort.

Working of Quick Sort algorithm:

To know the functioning of Quick sort, let's consider an array arr[] = {10, 80, 30, 90, 40, 50, 70}

- Indexes: 0 1 2 3 4 5 6
- low = 0, high = 6, pivot = arr[h] = 70
- Initialize index of smaller element, i = -1



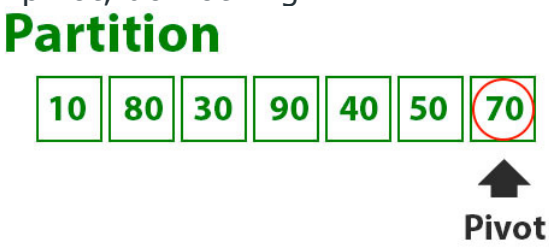
Counter variables
I: Index of smaller element
J: Loop variable

We start the loop with initial values

Test Condition	Actions	Value of variables
arr [J] <= pivot		I = -1 J = 0

Step1

- Traverse elements from j = low to high-1
- j = 0: Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
- i = 0
- arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j are same
- j = 1: Since arr[j] > pivot, do nothing



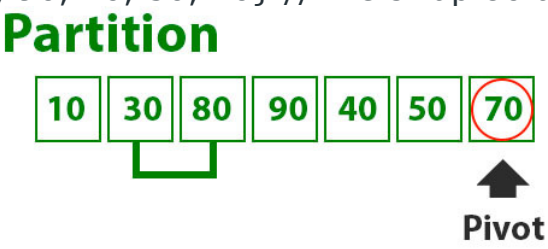
Counter variables
I: Index of smaller element
J: Loop variable

Pass 2

Test Condition	Actions	Value of variables
arr [J] <= pivot 80 < 70 false	No Action	I = 0 J = 1

Step2

- j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
- i = 1
- arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30

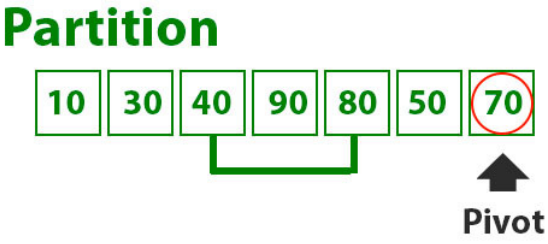


Counter variables
I: Index of smaller element
J: Loop variable

Test Condition	Actions	Value of variables
arr [J] <= pivot 30 < 70 true	i++ Swap(arr[i],arr[j])	I = 1 J = 2

Step3

- j = 3 : Since arr[j] > pivot, do nothing // No change in i and arr[]
- j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
- i = 2
- arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped



Counter variables
I: Index of smaller element
J: Loop variable

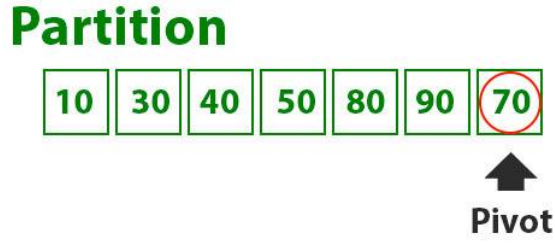
Pass 5

Test Condition	Actions	Value of variables
arr [J] <= pivot 40 < 70 true	i++ Swap(arr[i],arr[j])	I = 2 J = 4

Step 4

- j = 5 : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]
- i = 3

- arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped



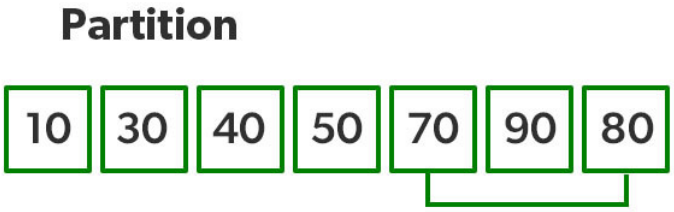
Counter variables
I: Index of smaller element
J: Loop variable

Before Pass 7, J becomes 6
so we come out of the loop

Test Condition	Actions	Value of variables
arr [J] <= pivot		I = 3 J = 6

Step 5

- We come out of loop because j is now equal to high-1.
- Finally we place pivot at correct position by swapping arr[i+1] and arr[high] (or pivot)
- arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped



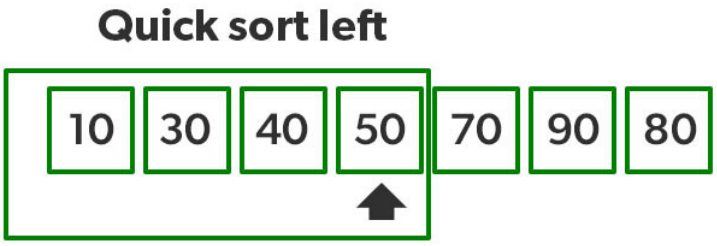
Counter Variable
I : Index of smaller element
J : Loop variable

We know swap arr[i+1] and pivot

I = 3

Step 6

- Now 70 is at its correct place. All elements smaller than 70 are before it and all elements greater than 70 are after it.
- Since quick sort is a recursive function, we call the partition function again at left and right partitions



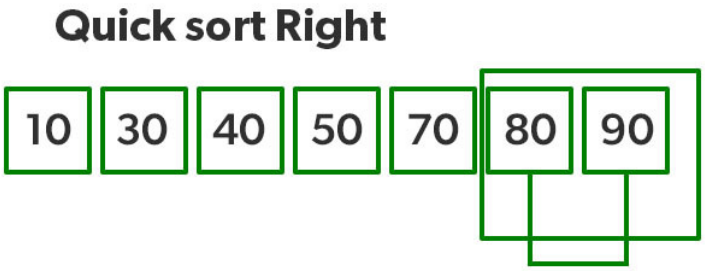
Since quick sort is a recursion function,
wecall the Partition function again

First 50 is the pivot.

As it is already at its correct position
we call the quicksort function again on the left part.

Step 7

- Again call function at right part and swap 80 and 90



80 is the Pivot

80 and 90 are swapped to bring pivot
to correct position

Step 8

6. Heap Sort

Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to the selection sort where we first find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements.

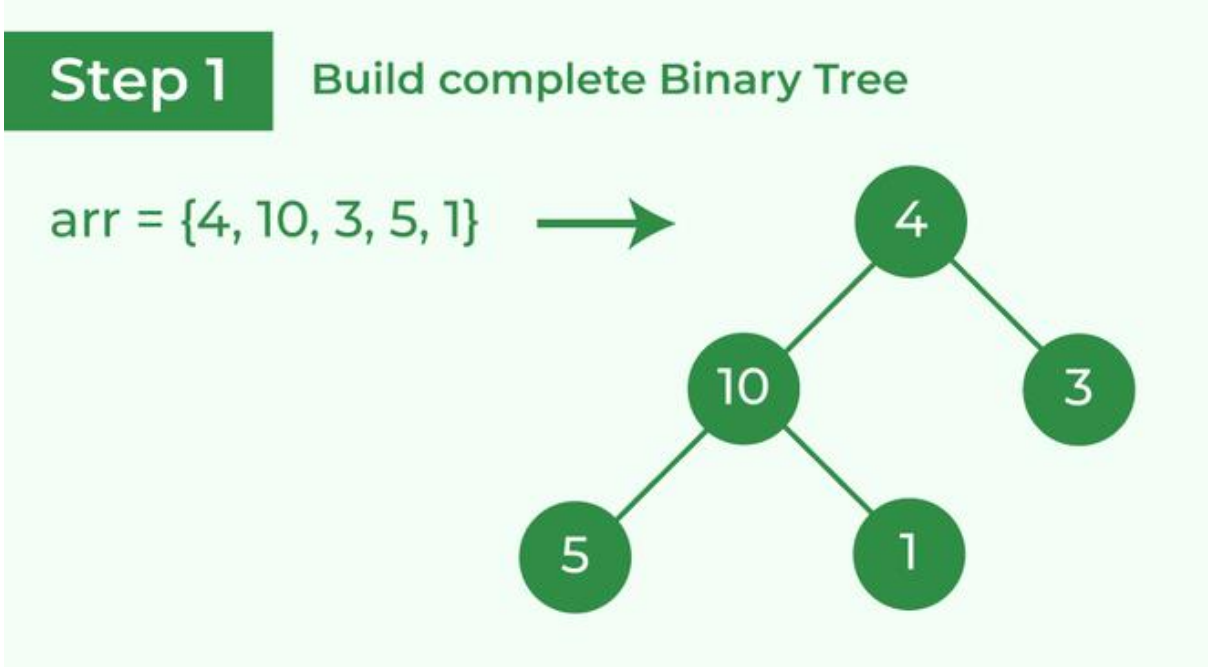
Advantages and Disadvantages of Heap Sort

Advantages	Disadvantages
The Heap sort algorithm is widely used because of its efficiency.	Heap sort requires more space for sorting
The Heap sort algorithm can be implemented as an in-place sorting algorithm	Quick sort is much more efficient than Heap in many cases
its memory usage is minimal	Heap sort make a tree of sorting elements.

Working of Heap Sort algorithm:

To understand heap sort more clearly, let's take an unsorted array and try to sort it using heap sort. Consider the array: arr[] = {4, 10, 3, 5, 1}.

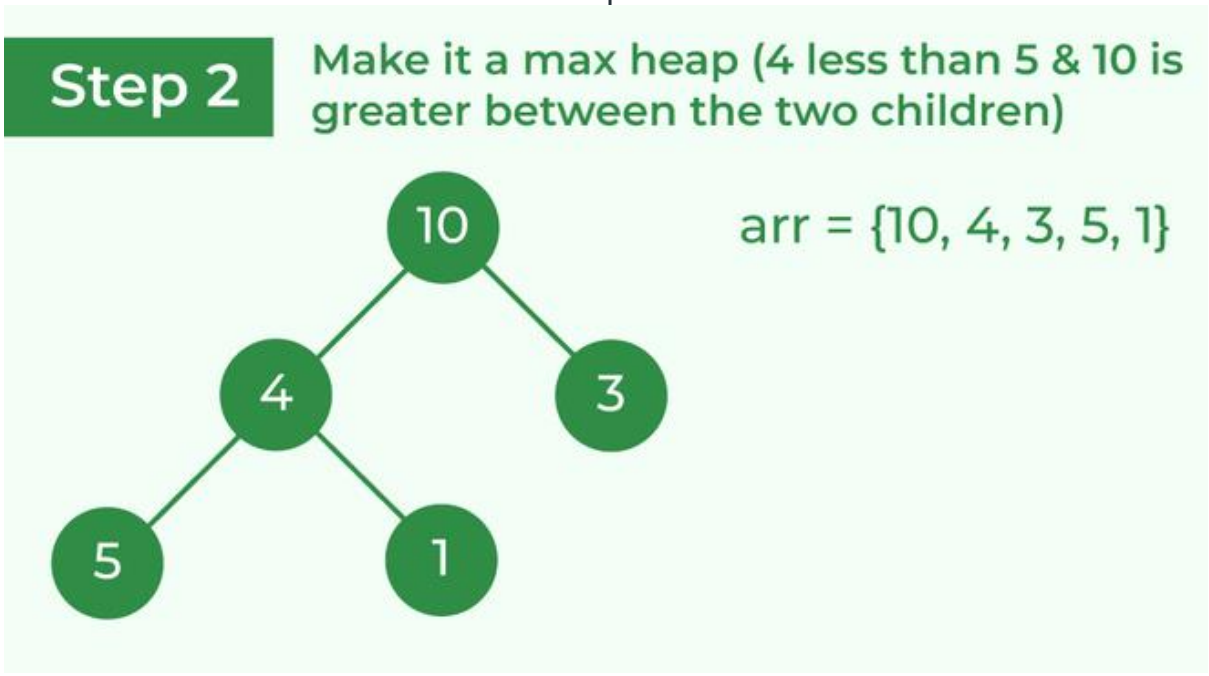
Build Complete Binary Tree: Build a complete binary tree from the array.



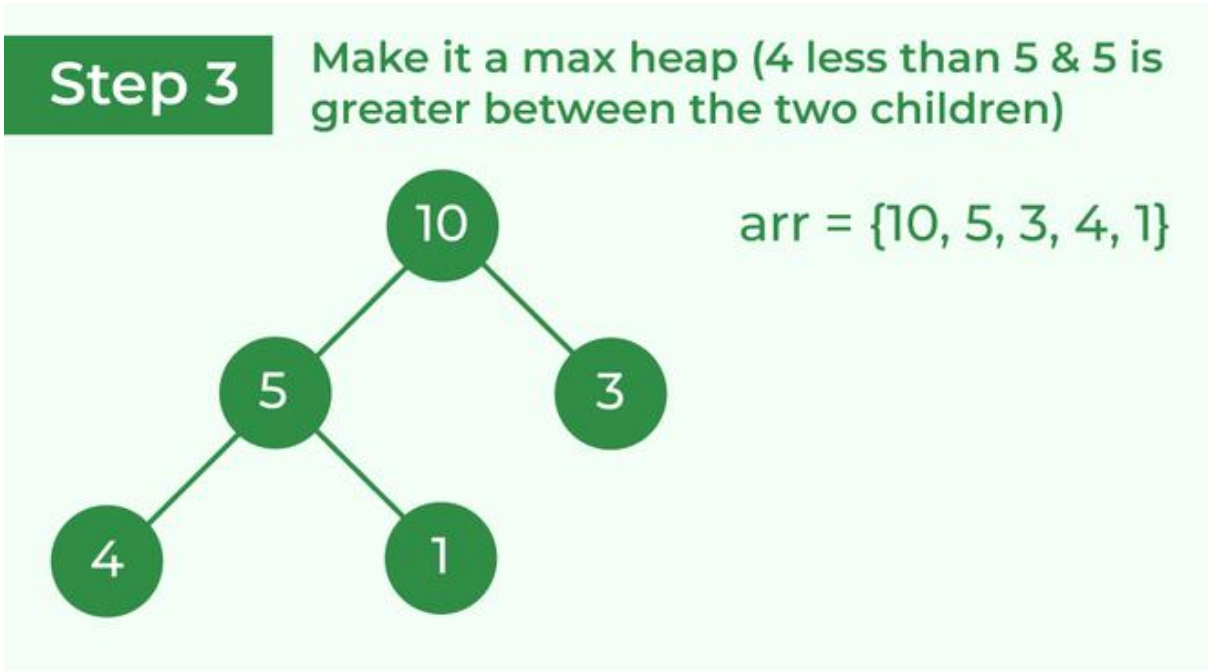
Build complete binary tree from the array

Transform into max heap: After that, the task is to construct a tree from that unsorted array and try to convert it into max heap.

- To transform a heap into a max-heap, the parent node should always be greater than or equal to the child nodes
    - Here, in this example, as the parent node 4 is smaller than the child node 10, thus, swap them to build a max-heap.
- Transform it into a max heap



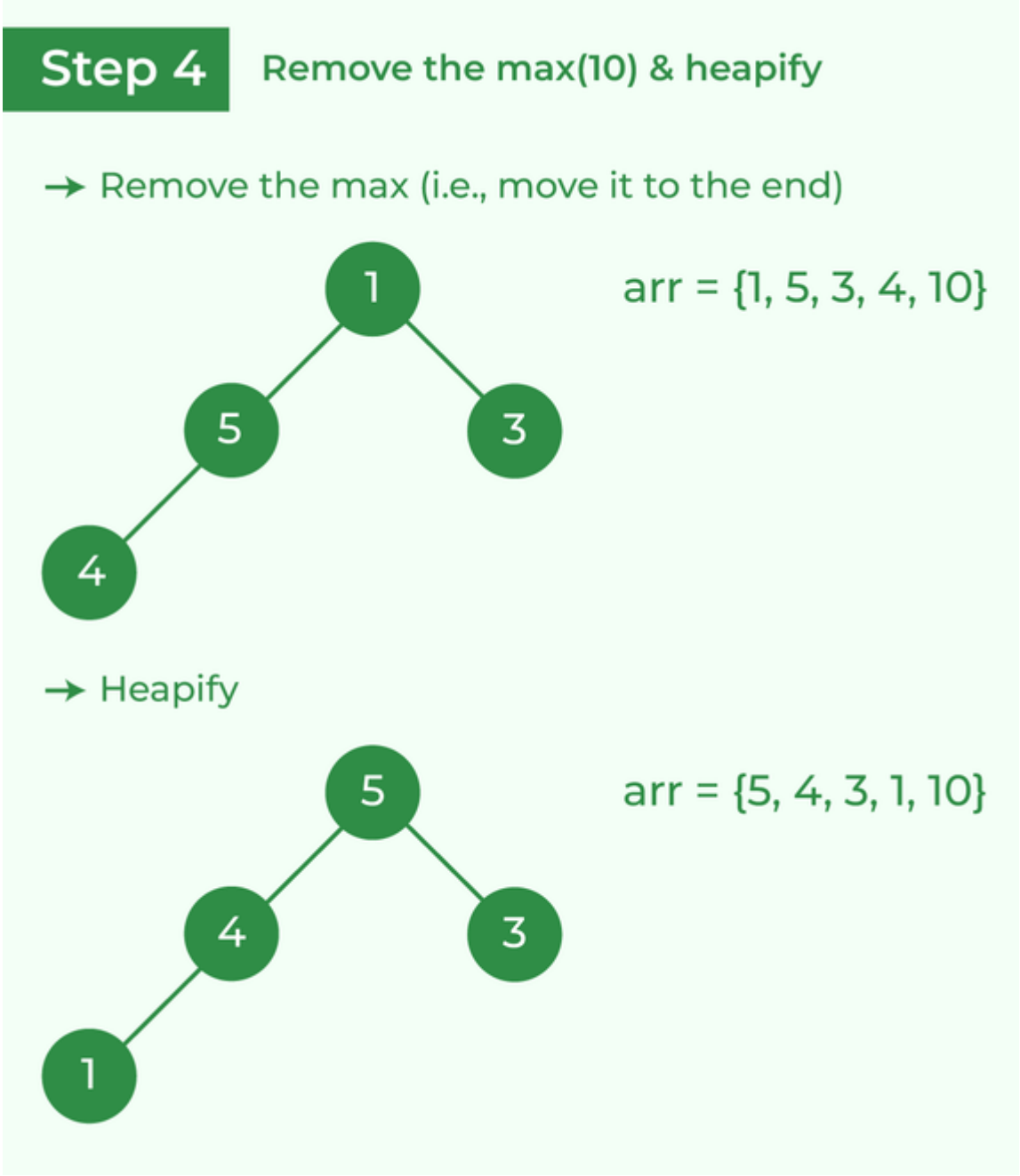
- Now, as seen, 4 as a parent is smaller than the child 5, thus swap both of these again and the resulted heap and array should be like this:



Make the tree a max heap

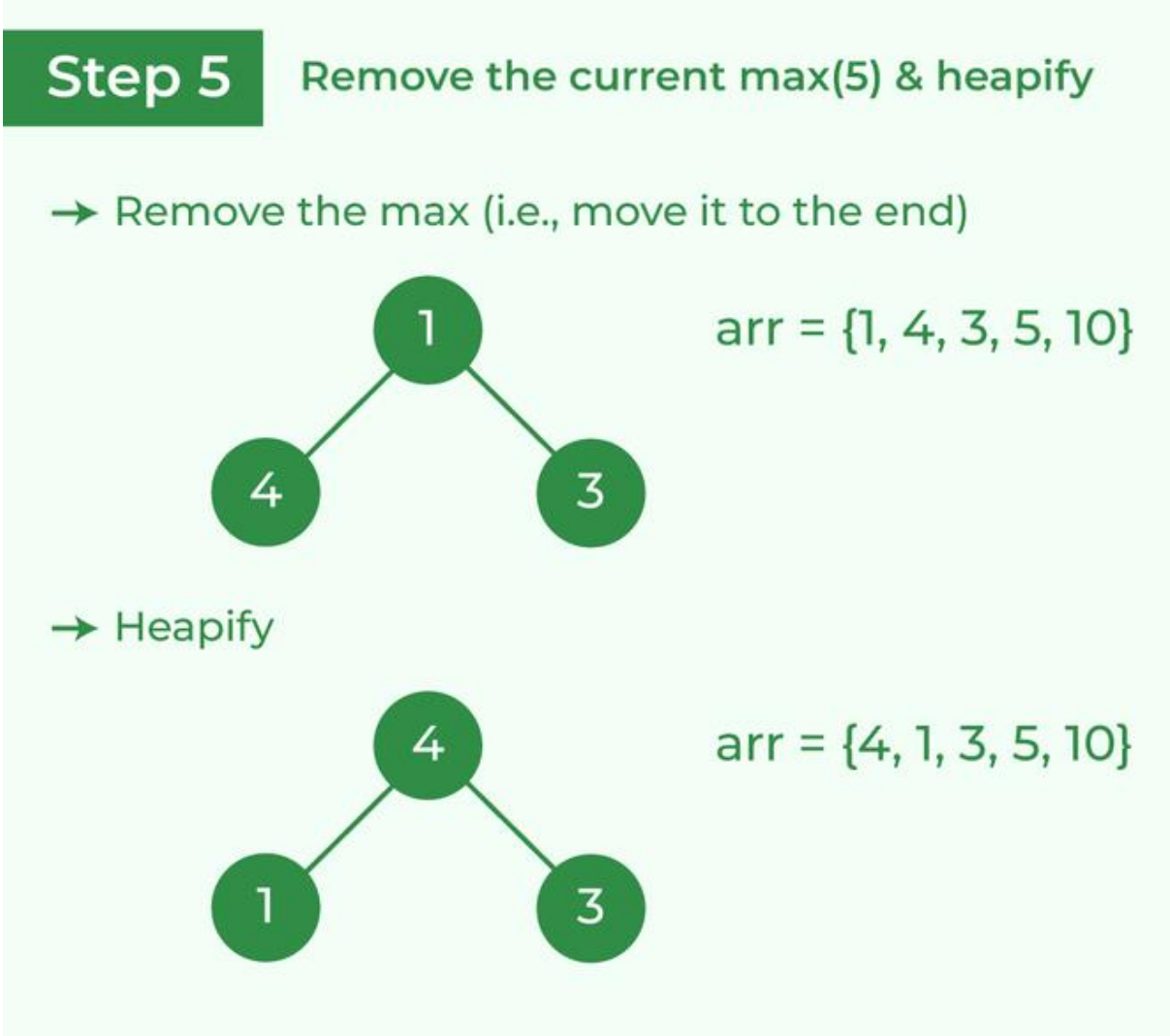
Perform heap sort: Remove the maximum element in each step (i.e., move it to the end position and remove that) and then consider the remaining elements and transform it into a max heap.

- Delete the root element (10) from the max heap. In order to delete this node, try to swap it with the last node, i.e. (1). After removing the root element, again heapify it to convert it into max heap.
    - Resulted heap and array should look like this:



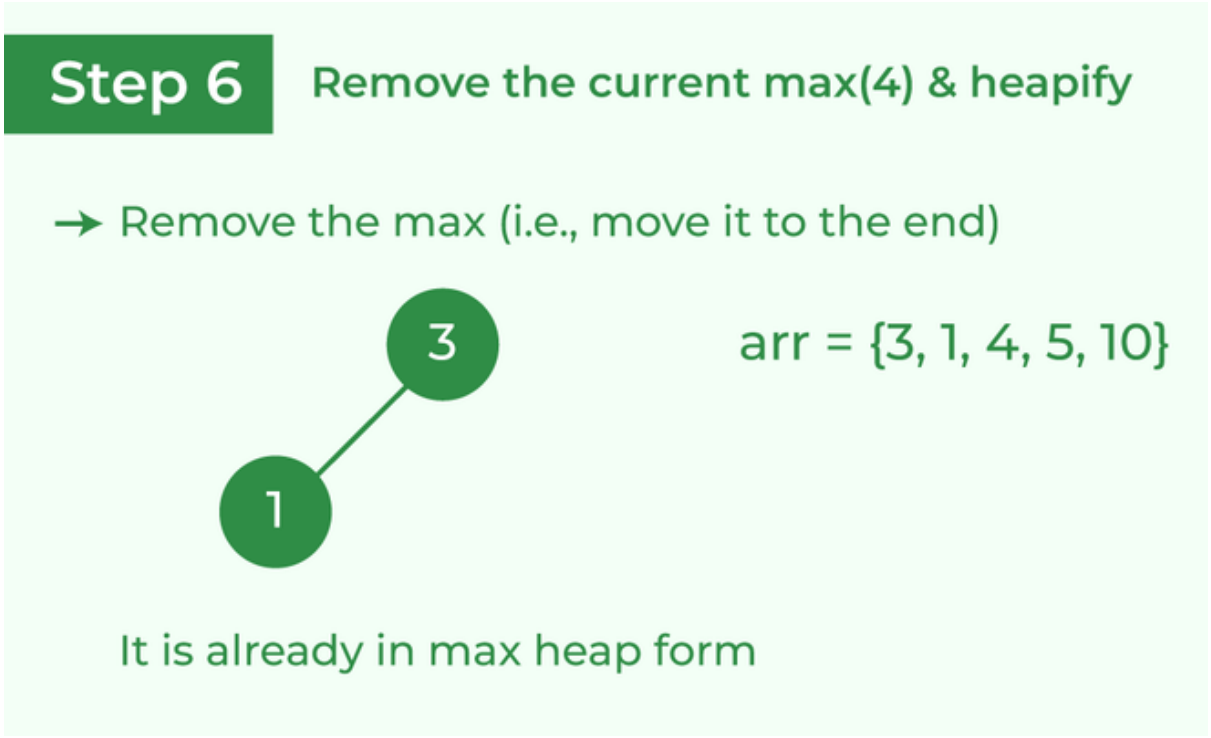
Remove 10 and perform heapify

- Repeat the above steps and it will look like the following:



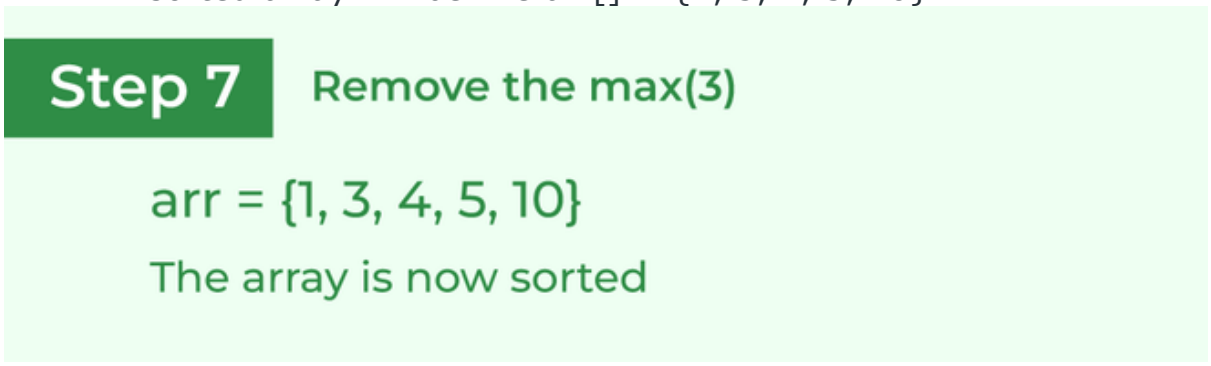
Remove 5 and perform heapify

- Now remove the root (i.e. 3) again and perform heapify.



Remove 4 and perform heapify

- Now when the root is removed once again it is sorted. and the sorted array will be like arr[] = {1, 3, 4, 5, 10}.



The sorted array

7. Counting Sort

Counting sort is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (kind of hashing). Then do some arithmetic to calculate the position of each object in the output sequence.

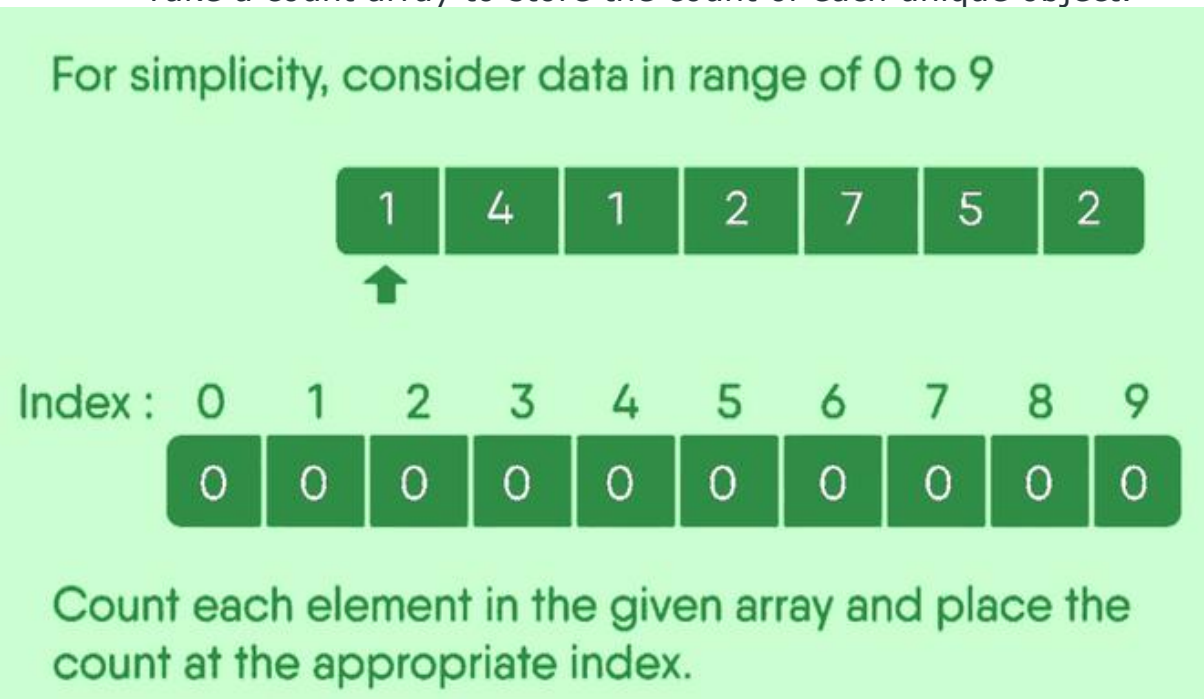
Advantages and Disadvantages of Counting Sort

Advantages	Disadvantages
Best-case and average-case: O(n + k)	Worst-case: O(n + k)
O(k)	High memory usage for large k
Stable (preserves relative order)	Limited to sorting integers/small integer keys
Efficient for small key ranges	Inefficient for large key ranges
Can outperform comparison-based sorts	Slower for general-purpose sorting

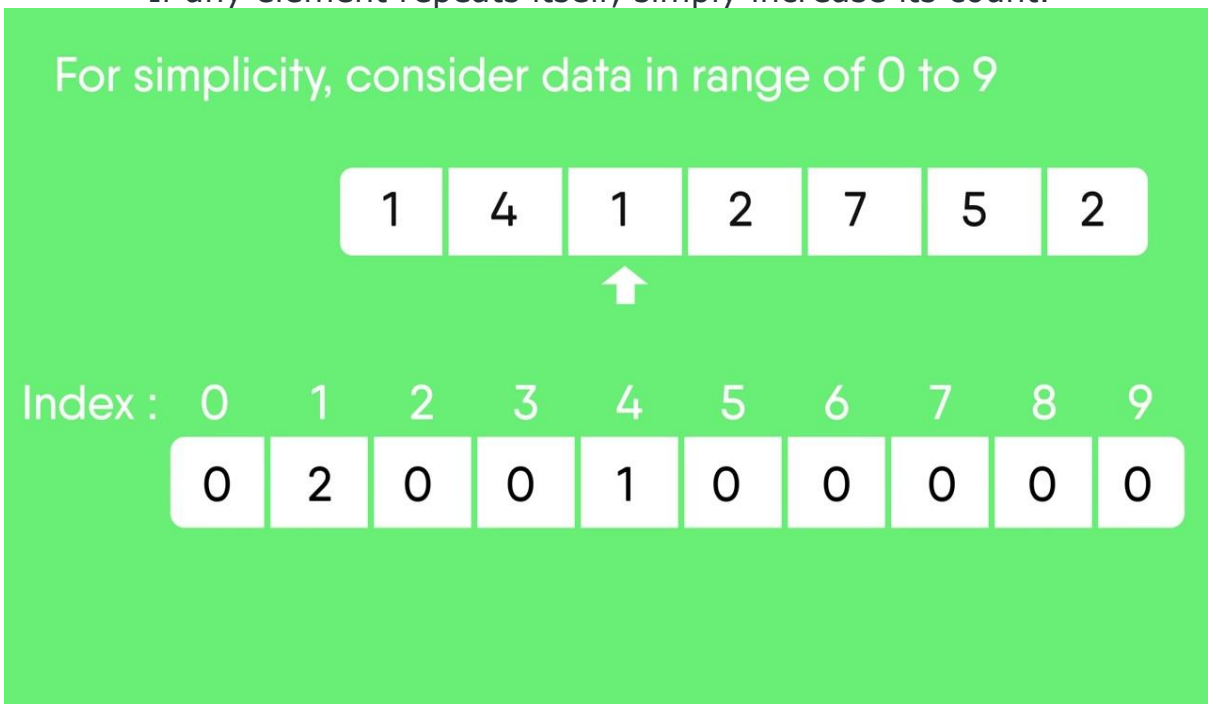
Working of Counting Sort algorithm:

Consider the array: arr[] = {1, 4, 1, 2, 7, 5, 2}.

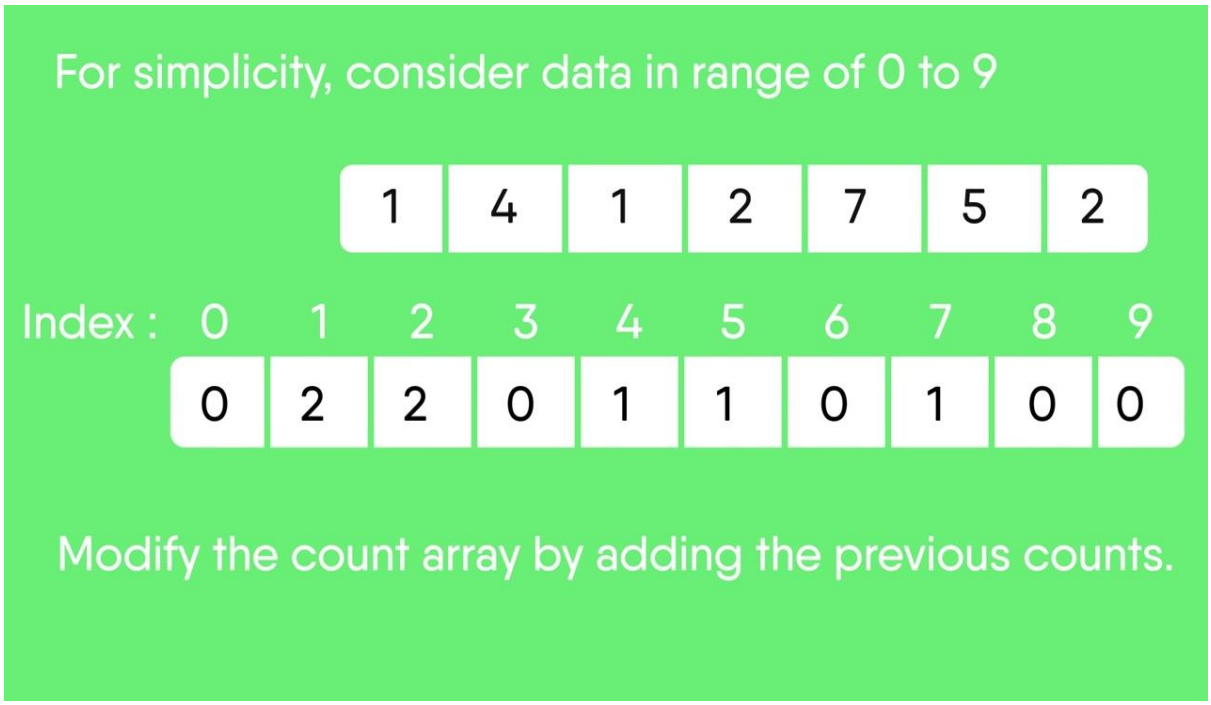
- Input data: {1, 4, 1, 2, 7, 5, 2}
- Take a count array to store the count of each unique object.



- Now, store the count of each unique element in the count array
- If any element repeats itself, simply increase its count.

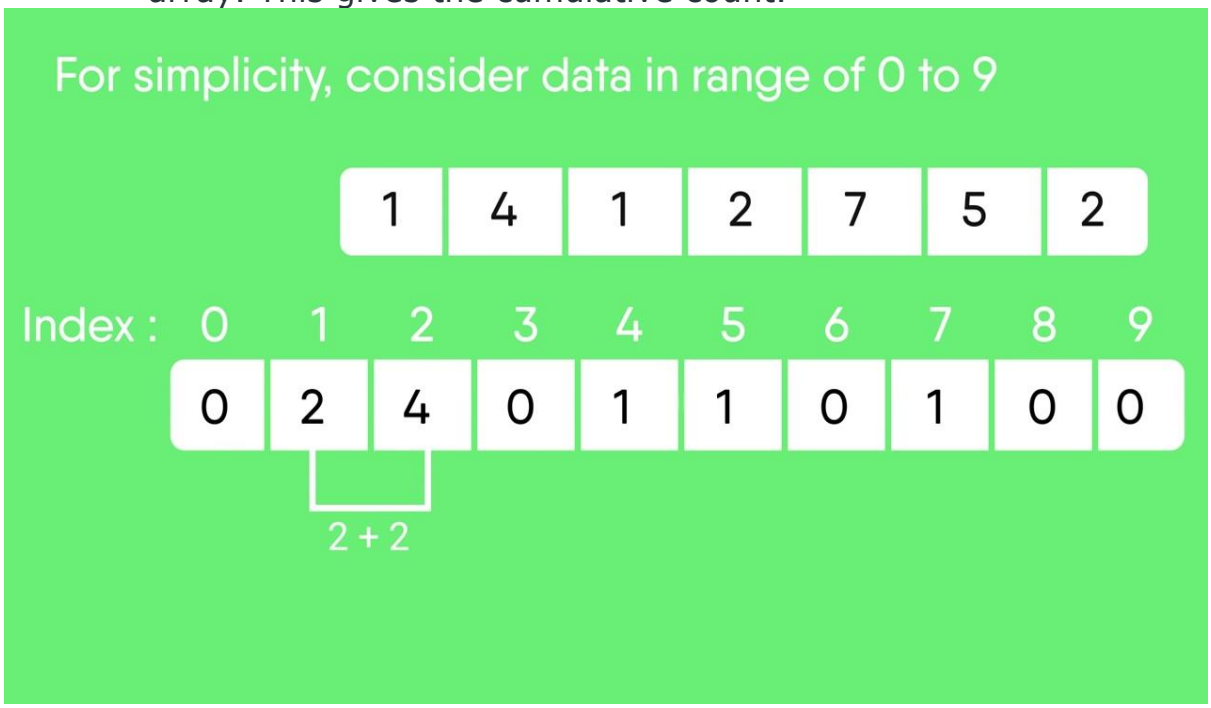


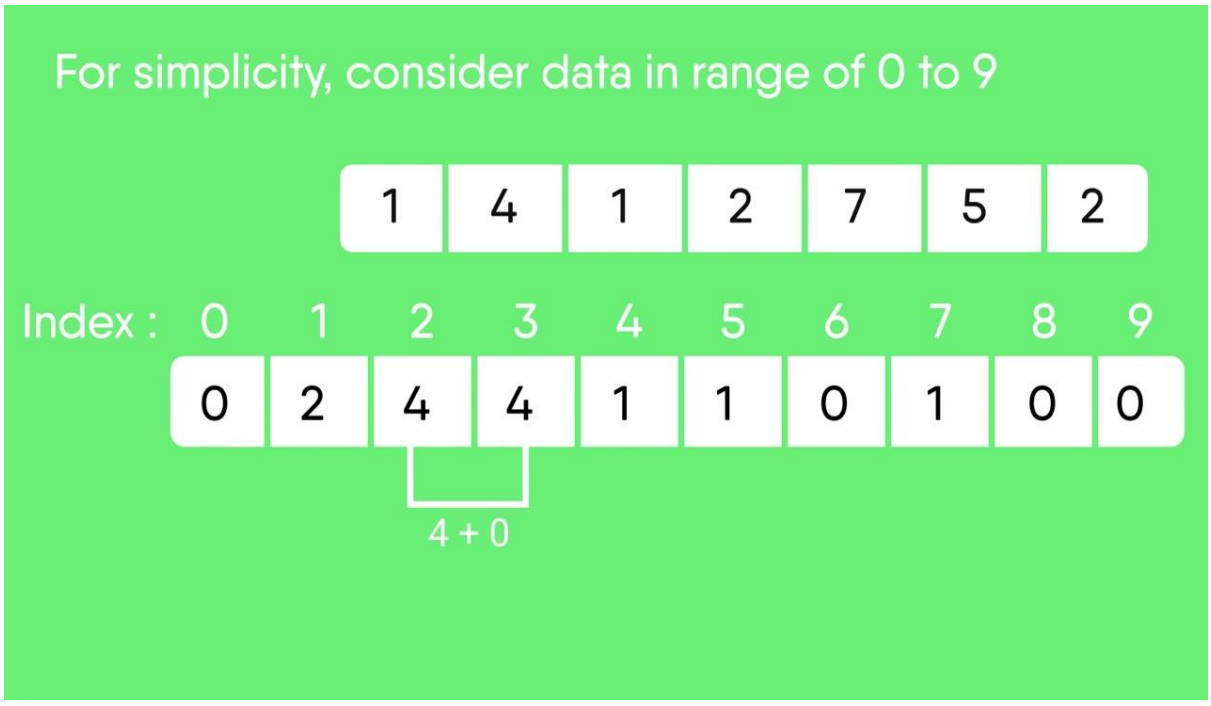
- Here, the count of each unique element in the count array is as shown below:
  - Index: 0 1 2 3 4 5 6 7 8 9
  - Count: 0 2 2 0 1 1 0 1 0 0



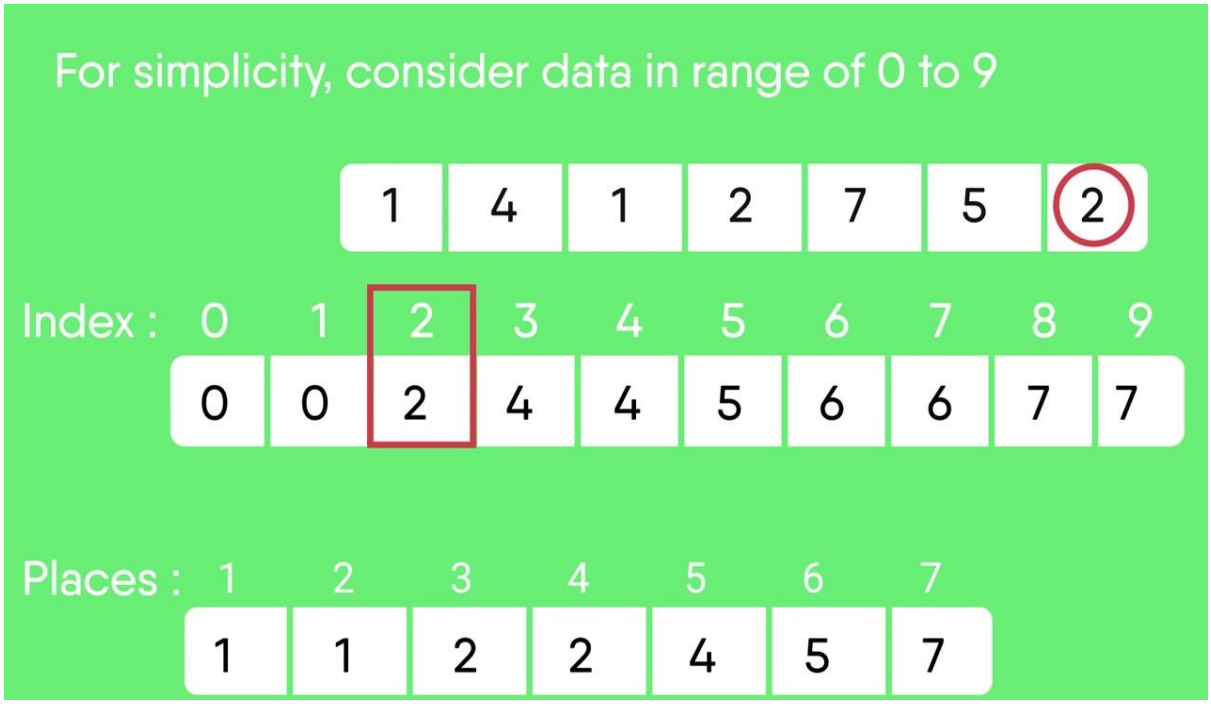
- Modify the count array such that each element at each index stores the sum of previous counts.
  - Index: 0 1 2 3 4 5 6 7 8 9
  - Count: 0 2 4 4 5 6 6 7 7 7
- The modified count array indicates the position of each object in the output sequence.

- Find the index of each element of the original array in the count array. This gives the cumulative count.

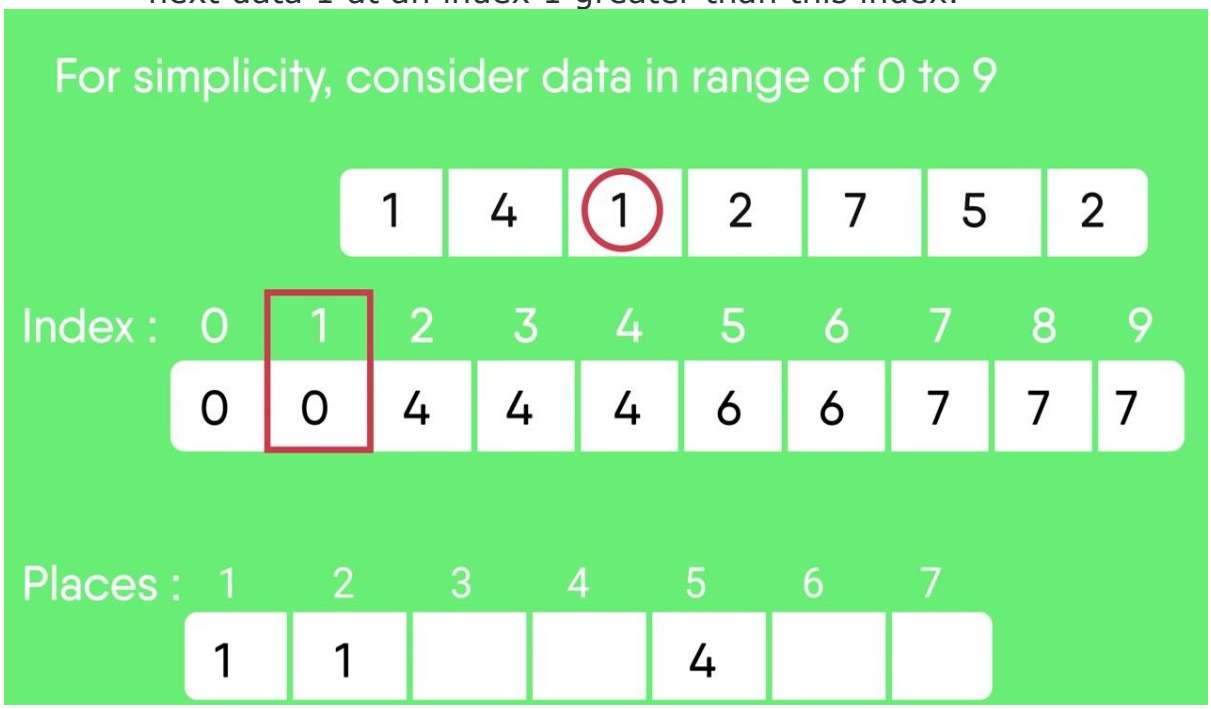




- Rotate the array clockwise for one time.
  - Index: 0 1 2 3 4 5 6 7 8 9
  - Count: 0 0 2 4 4 5 6 6 7 7



- Output each object from the input sequence followed by increasing its count by 1.
- Process the input data: 1, 4, 1, 2, 7, 5, 2. Position of 1 is 0.
- Put data 1 at index 0 in output. Increase count by 1 to place next data 1 at an index 1 greater than this index.



- After placing each element at its correct position, decrease its count by one.

Unlock Your Potential With Topperworld

LEARN MORE

www. topperworld.in

FOLLOW US!