Sorting

SORTING

In real life scenario we knowingly and unknowingly use sorting extensively.

1. In playing cards, a player gets a bunch of card during play. We need to arrange the cards of a particular player in ascending or descending order.
2. When we want to call someone we use only few characters of name because contacts in our phone are arranged in lexicographically (sorted).
3. We usually arrange student marks in decreasing order to get top three students name and roll number.

Sort means arranging items into an order either alphabetical or, numerical.

Sorting is the process of arranging data into meaningful sequence so that we can consider it with more ease and convenience. Order may be ascending or descending.

Types of Sorting Algorithm

In-place sorting and not-in-place sorting
Stable sorting and Unstable sorting
Internal sorting and External sorting
Adaptive Sorting and Non-Adaptive Sorting
Comparison based and non-comparison based

Types of Sorting Algorithm

In-place sorting and not-in-place sorting

In-place sorting

In in-place sorting algorithm we use fixed additional space for producing the output.
It sorts the list only by modifying the order of the elements within the list

eg.
Bubble sort, Comb sort, Selection sort, Insertion sort, Heap sort, Quick Sort and Shell sort.

Types of Sorting Algorithm

In not-in-place sorting (Out Place sorting)

In not-in-place sorting, we use equal or more additional space for arranging the items in the list. The extra space used by an algorithm depends on the input size.

Merge-sort is an example of not-in-place sorting.

Types of Sorting Algorithm

Stable sorting and Unstable sorting

Stable sorting

– In stable sorting algorithm the order of two objects with equal keys in output remains same after sorting as they appear in the input array to be sorted.

e.g. Merge Sort, Insertion Sort, Counting Sort, Bubble Sort, and Binary Tree Sort

Types of Sorting Algorithm

Unstable sorting

If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called unstable Sorting.
e.g. Quick Sort, Heap Sort, and Selection sort.

Types of Sorting Algorithm

Internal sorting and External sorting

Internal sorting

If the input data is such that it can be adjusted in the main memory at once or, when all data is placed in-memory it is called internal sorting .
e.g. Bubble Sort, Insertion Sort, Quick Sort.

Types of Sorting Algorithm

External sorting

If the input data is such that it cannot be adjusted in the memory entirely at once, it needs to be stored in a hard disk, floppy disk, or any other storage device.

External sorting typically uses a hybrid sort merge strategy.

In sorting phase, chunks of data small enough to fit in main memory. In the merge phase, the sorted sub-files are combined into a single larger file.

Types of Sorting Algorithm

Adaptive Sorting and Non-Adaptive Sorting-

Adaptive Sorting

If it takes advantage of already 'sorted' elements in the list that is to be sorted. It benefits from the presortedness in the input sequence. Adaptive sorting will try not to re-order them.
eg. Bubble sort, Insertion sort, quick sort

Types of Sorting Algorithm

Adaptive Sorting and Non-Adaptive Sorting-

Non- Adaptive Sorting

Which does not take into account the elements which are already sorted. They try to force every single element to be reordered to confirm their sortedness.

Merge sort is an Non adaptive because the order of the elements in the input array doesn’t matter. Time complexity will always be O(nlogn)

eg. Selection sort, Merge sort, heap sort.

Types of Sorting Algorithm

Comparison based and non-comparison based

Comparison based

- Algorithms, which sort a list, or an array, based only on the comparison of pairs of numbers, and not any other information (like what is being sorted, the frequency of numbers etc.), fall under this category. Elements of an array are compared with each other to find the sorted array.
- e.g. Bubble Sort, Selection Sort, Quick Sort, Merge Sort, Insertion Sort

Types of Sorting Algorithm

Non-Comparison based

Elements of array are not compared with each other to find the sorted array.
e.g. Radix Sort, Bucket Sort, Counting Sort.

Types of Sorting Algorithm

1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Heap Sort
5. Merge Sort
6. Quick Sort
7. Radix Sort

Types of Sorting Algorithm

The Sorting Techniques can broadly be categorised (based on time complexity) into

- Order of n^2 (Bubble Sort, Selection Sort, Insertion Sorts),
- Order of nlog n (Heap Sort, Quick Sort, Merge Sort)
- Order of n (Counting Sort, Bucket Sort and Radix Sort)

Bubble Sort:

Bubble sort, which is also known as **sinking sort,** is a simple **Brute force Sorting technique** that repeatedly iterates through the item list to be sorted.

The algorithm runs as follows:

1.Start at the beginning of the list.

2.Compare the first value in the list with the next one up. If the first value is bigger, swap the positions of the two values.

3.Move to the second value in the list. ...

4.Keep going until the there are no more items to compare.

5.Go back to the start of the list.

Bubble Sort:

In first pass

```
FOR j=1 TO N-1 DO
        IF A[j] >A[j+1] THEN
                Exchange (A[j], A[j+1])
```

Total N-1 passes of similar nature

Bubble Sort:

ALGORITHM Bubble Sort(A[ ],N)

```
BEGIN:
        FOR i=1 TO N-1 DO
                FOR j= 1 To N-i DO
                        IF A[j] >A[j+1] THEN
                                Exchange (A[j], A[j+1])
END;
```

Bubble Sort:

Complexity:

Analysis:

N-1comparisons in first iteration

N-2 comparison in second iteration

N-3 comparison in third iteration …

1 Comparisons in (N-1)st iteration

Total comparisons = (N-1) + (N-2) + (N-3) + … +1

= N(N-1)/2

= N2 /2 – N/2

Bubble Sort:

Complexity:

Analysis:

If exchanges take place with each comparison, Total number of statements to be executed are (N2 /2 – N/2)*3 as 3 statements are required for exchange. The Time Complexity can be written as θ(N2)

There are 2 extra variables used in the logic. Hence space Complexity is θ(1)

Bubble Sort Algorithm

a: array of items
n: size of list

n:	
i:	
j:	
rounds:	

```
start
1. declare variables - i, j
2. loop :(i = 0 to i < n)   //outer loop
    2.1 loop: (j = 0 to j < n-i-1)    // inner loop
        2.1.1  if(a[j] > a[j+1])  then
                2.1.1.1 swap(a[j] , a[j+1])
               end if
      end loop        // inner loop
  end loop      // outer loop
stop
```

```
int arr[5] = {5, 2, 4 , 3 , 6 }
```

0	1	2	3	4
5	2	4	3	6

Bubble Sort:

Optimized Bubble Sort

There are two scenarios possible –

Case 1 - When elements are already sorted: Bubble Sort does not perform any swapping.

Case 2 - When elements are sorted after k-1 passes: In the kth pass no swapping occurs

If no swap happens in some iteration means elements are sorted and we should stop comparisons. This can be done by the use of a flag variable.

OPTIMIZED Bubble Sort Algorithm

n:	
i:	
j:	
rounds:	
flag:	

```
a: array of items
n: size of list
start
1. declare variables - i, j
2. loop :(i = 0 to i < n)   //outer loop
    2.1     flag = false
     2.2 loop: (j = 0 to j < n-i-1)     // inner loop
           2.2.2  if(a[j] > a[j+1])  then
                    2.2.2.1 flag = true
                     2.2.2.2 swap(a[j] , a[j+1])
                    end if
          end loop          // inner loop
    2.3 if (flag == false) then BREAK
    end loop      // outer loop
 stop
```

```
int arr[5] = {5, 2, 4 , 3 , 6 }
```

0	1	2	3	4
5	2	4	3	6

Bubble Sort:

Optimized Bubble Sort

If elements are already sorted then it takes Ω(N) time because after the first pass flag remains unchanged, meaning that no swapping occurs. Subsequent passes are not performed. A total of N-1 comparisons are performed in the first pass and that is all.
If elements become sorted after k-1 passes, kth pass finds no exchanges. It takes θ(N*k) effort.

Selection Sort:

Selection sort is nothing but a variation of Bubble Sort because in selection sort swapping occurs only one time per pass

In every pass, choose largest or smallest element and swap it with last or first element. If the smallest element was taken for swap, position of first element gets fixed.
The second pass starts with the second element and smallest element is found out of remaining N-1 elements and exchanged with the second element.
In the third pass the smallest element is found out of N-2 elements (3rd to Nth element) and exchanged with the third element and so on so forth.
The same is performed for N-1 passes.

Selection Sort:

Number of comparisons in this Algorithm is just the same as that of Bubble Sort,
but number of swaps in this ‘N’ (as compared to N*(N-1)/2 Swaps in Bubble Sort.

Selection Sort Algorithm -

>> Selection sort is a sorting algorithm, specifically an in-place comparison sort.

>> It has $O(n^2)$ time complexity, making it inefficient on large lists.

>> The algorithm divides the input list into two parts: the sublist of items already sorted, which is built up from left to right at the front (left) of the list, and the sublist of items remaining to be sorted that occupy the rest of the list.

>> Initially, the sorted sublist is empty and the unsorted sublist is the entire input list.

>> The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging (swapping) it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.

```
int arr[5] = {64, 25, 12, 22, 11}
```

0	1	2	3	4
64	25	12	22	11

Selection Sort Algorithm

```
arr    : array of items
n      : size of list

for(i = 0; i<n - 1;i++)
/* set current element as minimum */
   min = i

   /* check the element to be minimum */

   for(j = i+1; j<n ;j++)
      if arr[j] < arr[min] then
         min = j;
      end if
   end for

   /* swap the minimum element & current element*/
   if(min != i) then
      swap arr[min] and arr[i]
   end if
end for
```

n:	5
i:	
min:	
j:	

```
int arr[5] = {64, 25, 12, 22, 11}
```

0	1	2	3	4
64	25	12	22	11

Applying Selection Sort -

1. Find minimum element & place it at postion 0

2. Find minimum element & place it at postion 1

3. Find minimum element & place it at postion 2

4. Find minimum element & place it at postion 3

Selection Sort:


```
BEGIN:
FOR i=1 TO N-1 DO
Min=i;

 FOR j=i+1 TO N DO
            IF A[j] <A[min] THEN
                        Min=i
Exchange (A[i], A[Min])
END;
```

Selection Sort:

Time Analysis (Total (N-1) iterations

N-1 comparisons in first iteration
N-2 comparison in second iteration
N-3 comparison in third iteration …
1 Comparison in (N-1)st iteration
Total = (N-1) + (N-2) + (N-3) + … +1
= N(N-1)/2
= N2 /2 – N/2
1 Exchange per iteration

Selection Sort:

Time Analysis (Total (N-1) iterations

1 Exchange per iteration
hence total exchanges are N-1.
Total effort for N-1 iterations =3 *(N-1)
Total Effort = 3*(N–1) + N2 /2 – N/2
= N2 /2 + 5/2*N - 3
As there is no best case possible as all iterations are compulsory,
Complexity should be written in Theta notation i.e. $\theta(N^2)$

Selection Sort:

Space Complexity

Space Complexity remains θ(1) as only 3 variables (i, j, Min) are used in the logic.

As selection sort takes only O(N) swaps in worst case, it is the best suitable where we need minimum number of writes on disk.

Insertion Sort Algorithm -

Theory -

>> Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands.

>> Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time.

>> Efficient for (quite) small data sets, much like other quadratic sorting algorithms

>> More efficient in practice than most other simple quadratic (i.e., $O(n^2)$) algorithms such as selection sort or bubble sort

>> Time Complexity: $O(n^2)$

```
int arr[5] = {9, 7, 3, 6, 2}
```

Insertion Sort Algorithm -

Working -

Step 1 – Pick next element

Step 2 – Compare with all elements in the sorted sub-list on the left

Step 3 – Shift all the elements in the sorted sub-list that is greater than the value to be sorted

Step 4 – Insert the value

Step 5 – Repeat until list is sorted

Insertion Sort Algorithm

n:	5
i:	
key:	
j:	

```
a: array of items
n: size of list

start
1. declare variables - i, key, j
2. loop : 1 to n-1        //outer loop
    2.1 key = a[i];
    2.2 j = i - 1;

    2.3 loop: (j >= 0 && arr[j] > key) // inner loop
          2.3.1 arr[j + 1] = arr[j];
          2.3.2 j = j - 1;
        end loop          // inner loop

    2.4 arr[j + 1] = key;
    end loop       // outer loop
stop
```

```
int arr[5] = {9, 7, 3, 6, 2}
```

0	1	2	3	4
9	7	3	6	2

Merge Sort Algorithm -

Theory -

>> Merge Sort is a Divide and Conquer algorithm.

>> It divides input array in two halves, calls itself for the two halves(recursively) and then merges the two sorted halves.

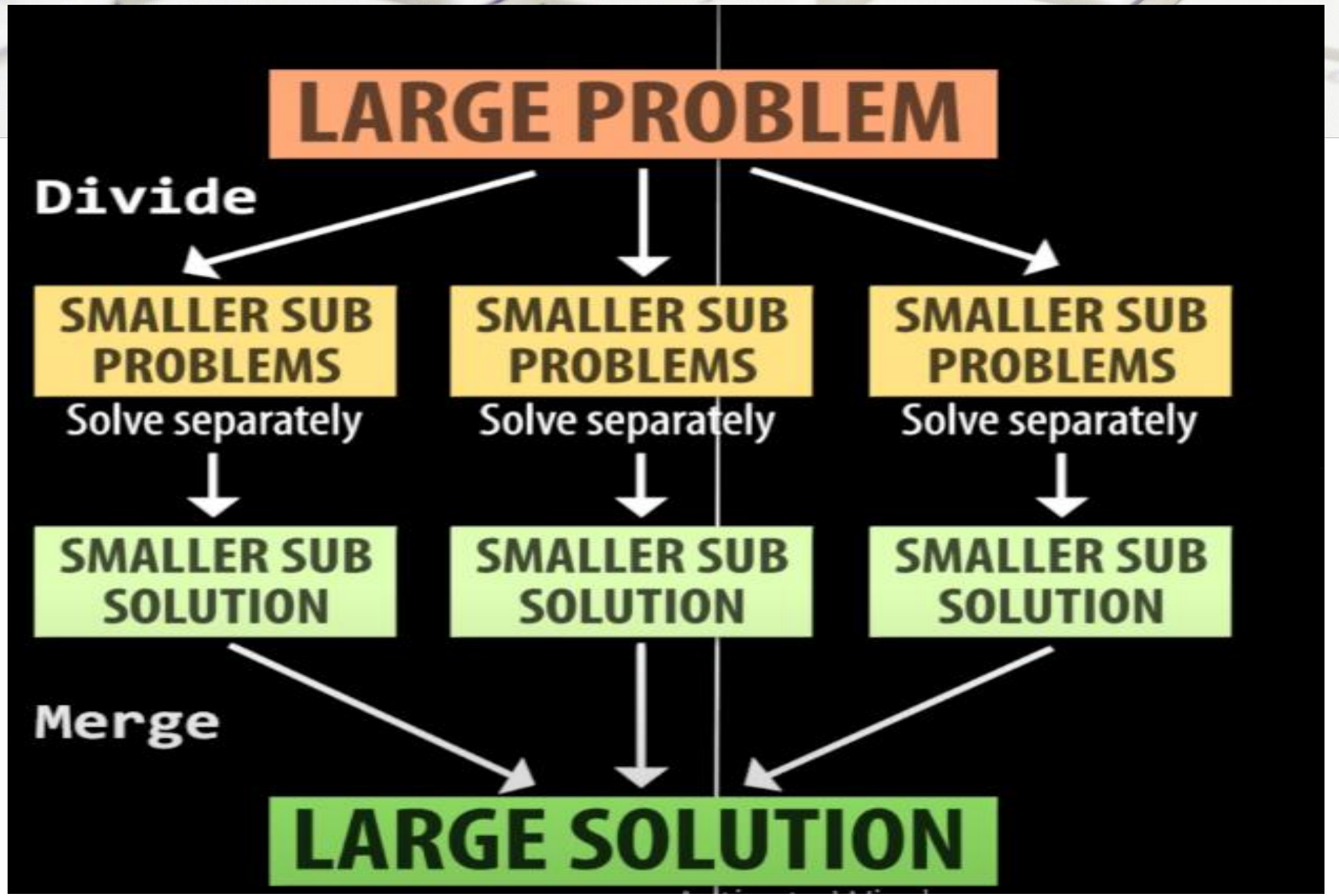
>> A separate merge() function is used for merging two halves.

>> Merge sort is one of the most efficient sorting algorithms.

>> Time Complexity: O(nlog(n))

```
int arr[5] = {9, 7, 3, 6, 2}
```

0	1	2	3	4
9	7	3	6	2



Merge Sort Algorithm (Working)

```
mergeSort(arr[], l,  r)
{
```

Step 1 - Find the middle point to divide the array into two halves:

middle m = (l+r)/2

Division

Step 2 - Call mergeSort for first half :

mergeSort(arr, l, m)

Step 3 - Call mergeSort for second half:

mergeSort(arr, m+1, r)

Recursion

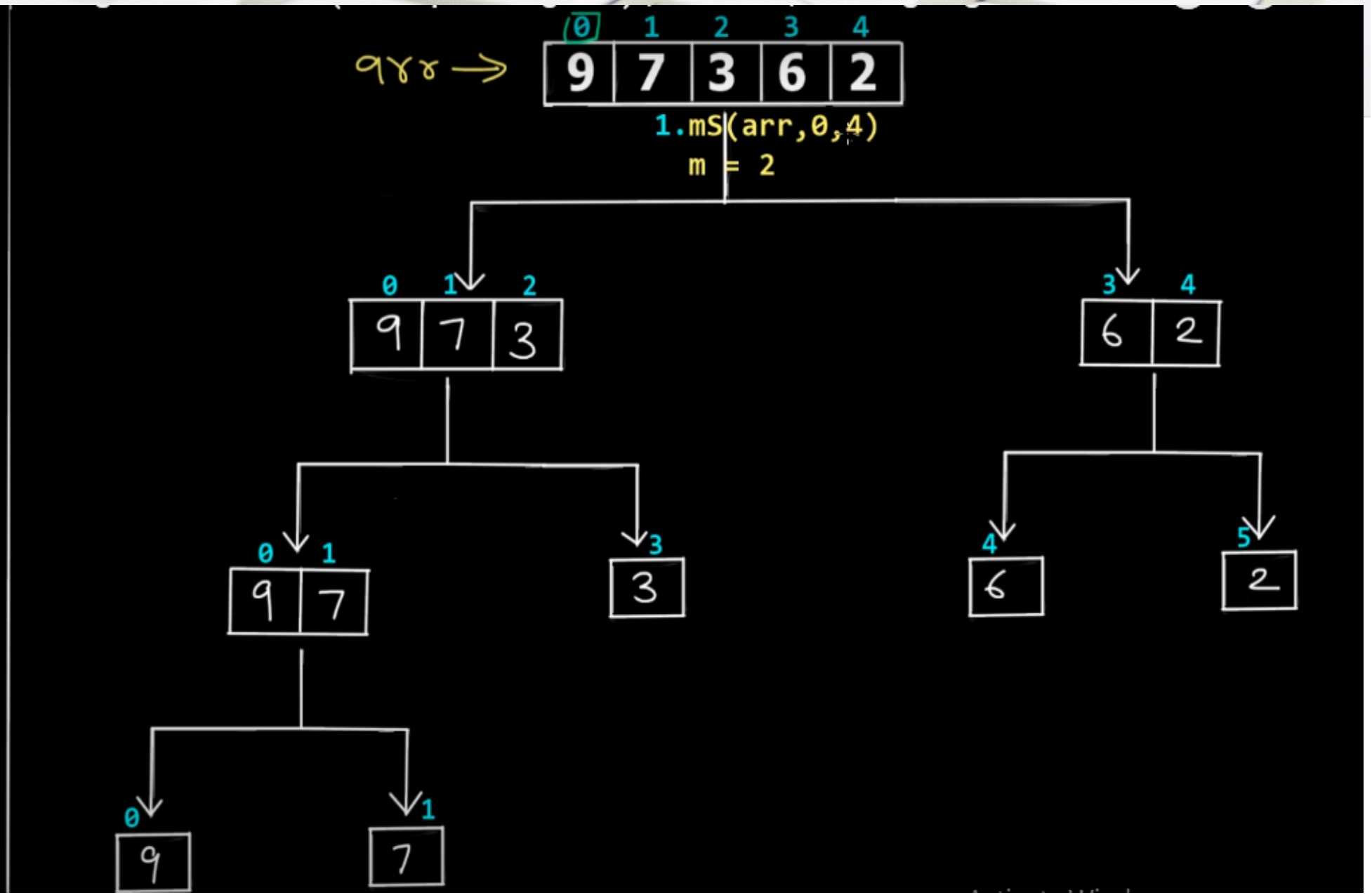
Step 4 - Merge the two halves sorted in step 2 and 3:

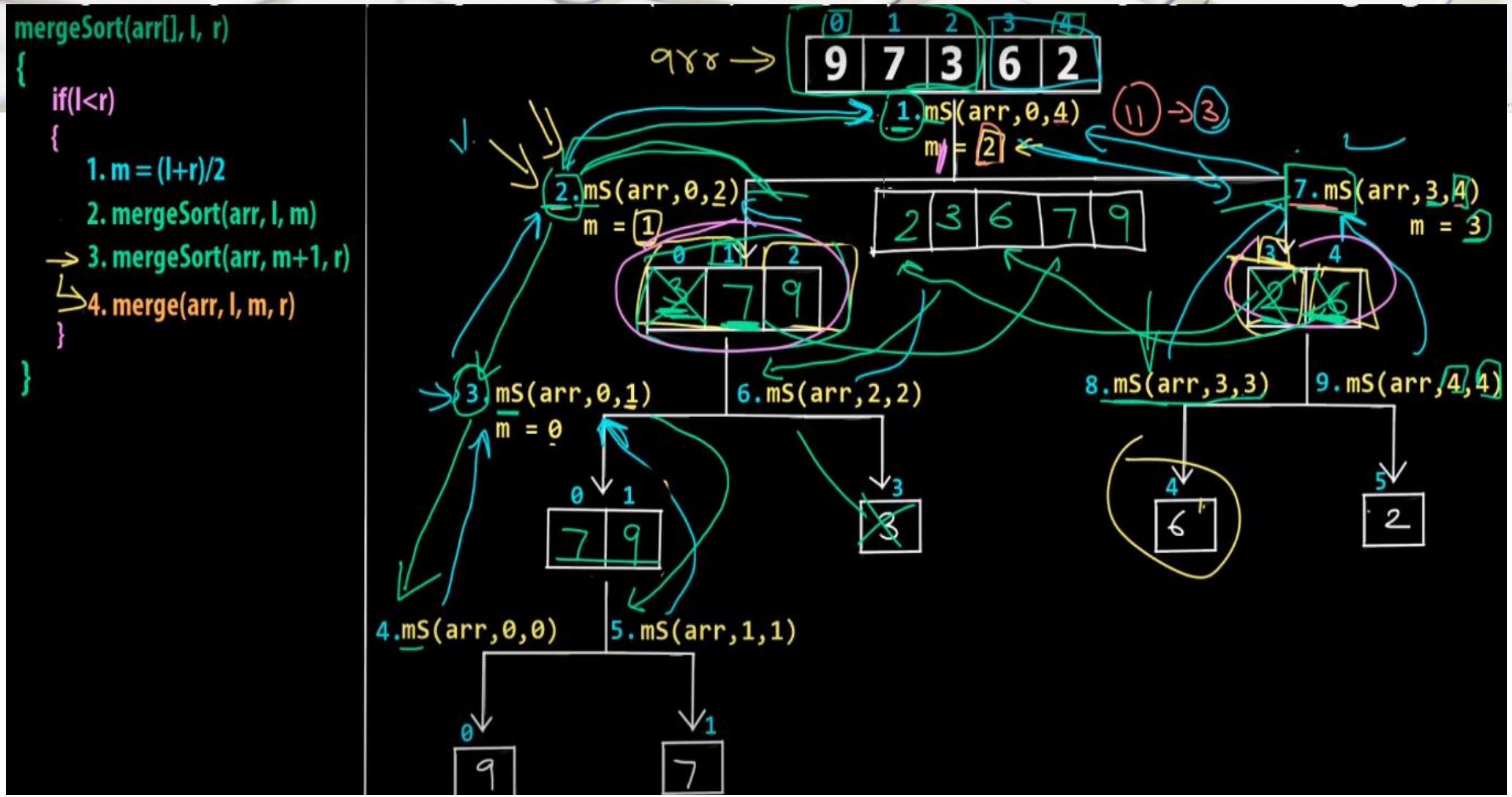
merge(arr, l, m, r)

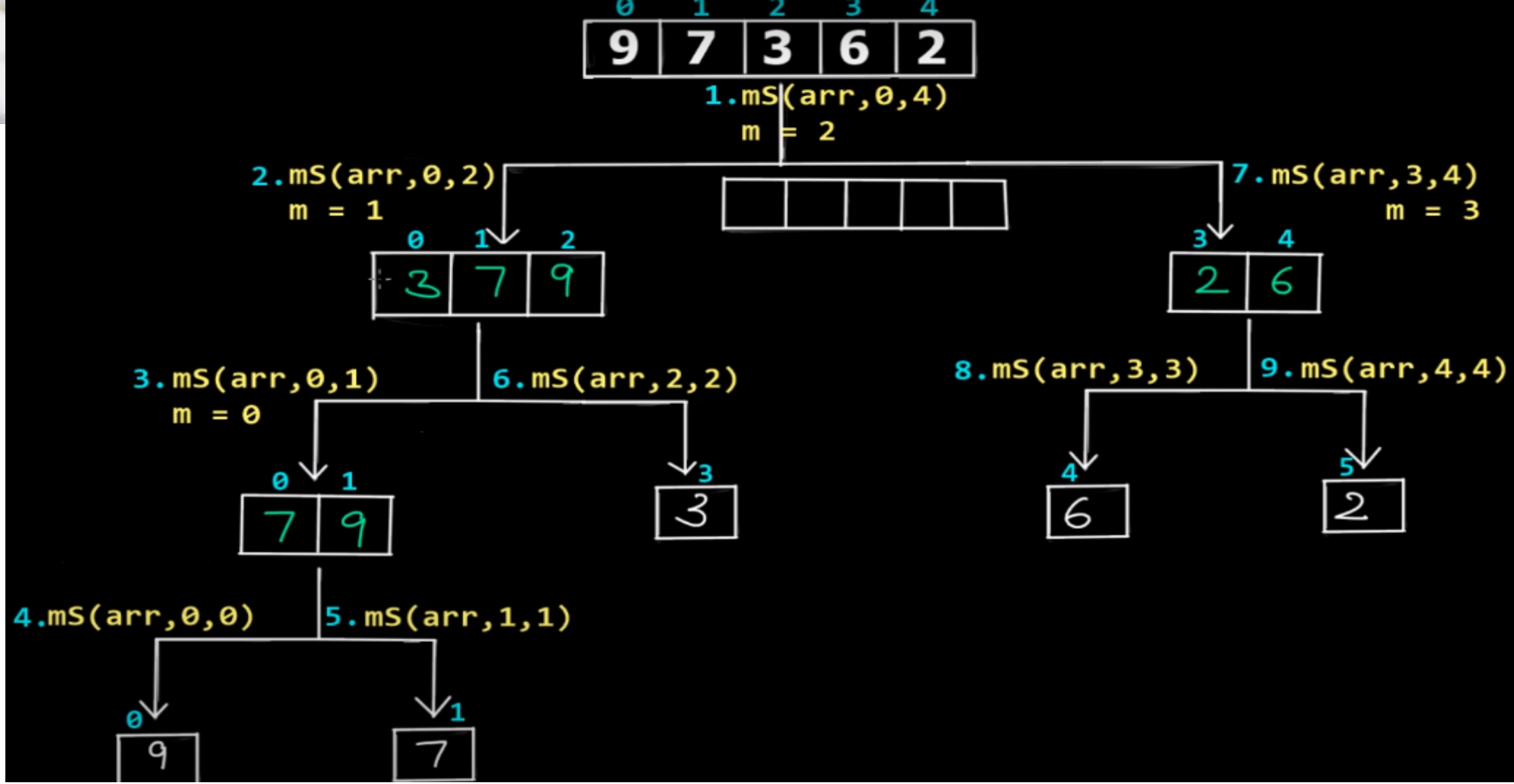
Merging

```
}
```

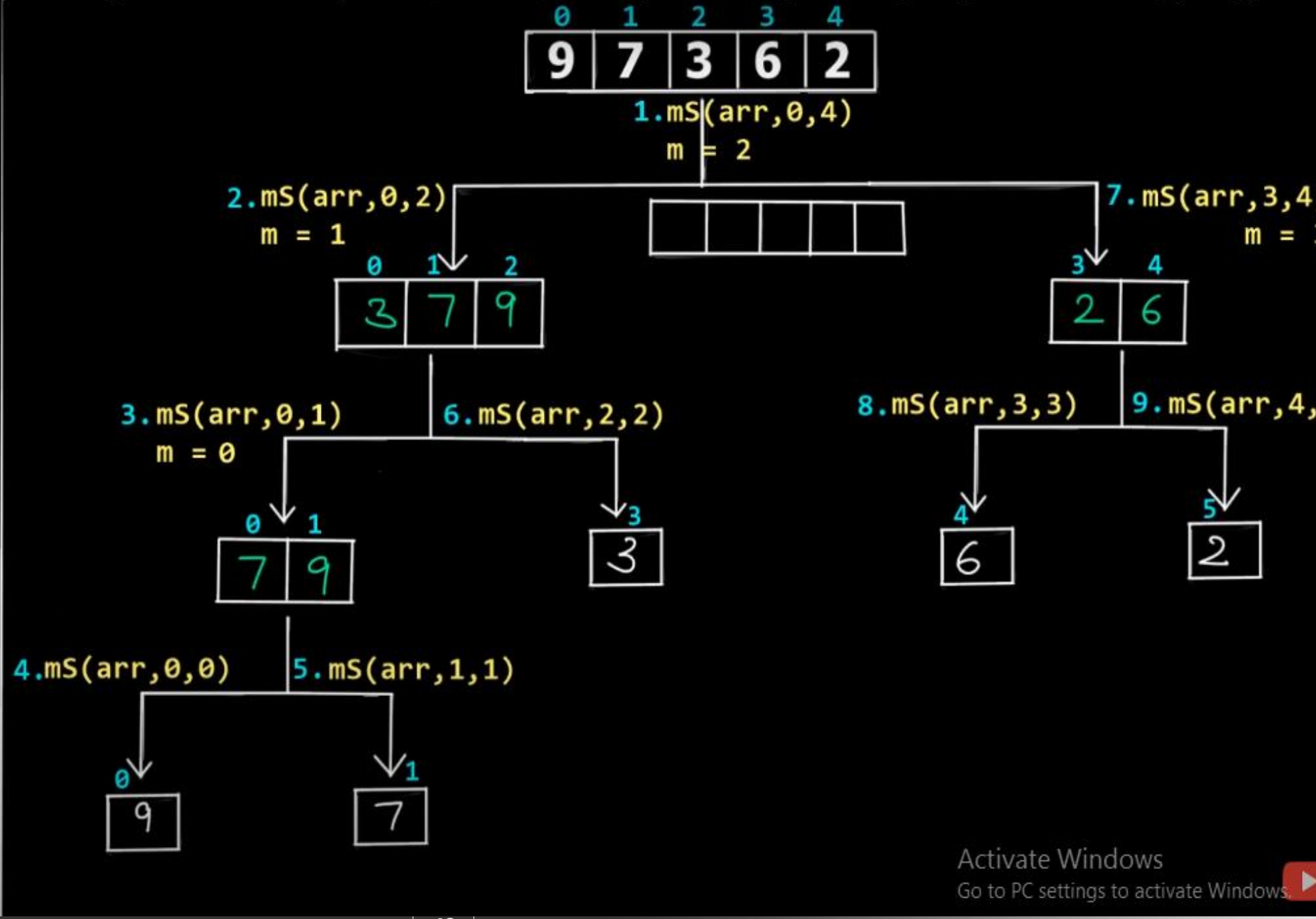
```
mergeSort(arr[], l, r)
{
    if(l<r)
    {
        1. m = (l+r)/2
        2. mergeSort(arr, l, m)
        3. mergeSort(arr, m+1, r)
        4. merge(arr, l, m, r)
    }
}
```




mergeSort(arr[], l, r)
{
if(l<r)
{
1. m = (l+r)/2
2. mergeSort(arr, l, m)
3. mergeSort(arr, m+1, r)
4. merge(arr, l, m, r)
}
}
9 7 3 6 2
1.mS(arr,0,4)
m = 2
2.mS(arr,0,2)
m = 1
7.mS(arr,3,4)
m = 3
2 3 6 7 9
3.mS(arr,0,1)
m = 0
6.mS(arr,2,2)
8.mS(arr,3,3)
9.mS(arr,4,4)
4.mS(arr,0,0)
5.mS(arr,1,1)


0 1 2 3 4
9 7 3 6 2
1.mS(arr,0,4)
m = 2
2.mS(arr,0,2)
m = 1
7.mS(arr,3,4)
m = 3
0 1 2
3 7 9
3 4
2 6
3.mS(arr,0,1)
m = 0
6.mS(arr,2,2)
8.mS(arr,3,3)
9.mS(arr,4,4)
0 1
7 9
3
4
6
5
2
4.mS(arr,0,0)
5.mS(arr,1,1)
0
9
1
7

```
merge(arr,l,m,r)

1. i=l, j=m+1, k=l  // 3 variables
2. temp[]  // create temp array

3. while (i <=m && j <= r)
 3.1 if (arr[i] <= arr[j])
      temp[k] = arr[i]
      i++, k++
 3.2 else
      temp[k] = arr[j]
      j++, k++

 4. while (i <= m)
      temp[k] = arr[i]
      i++, k++
 5. while (j <= r)
      temp[k] = arr[j]
      j++, k++

 6. for(int p=l;p<=r;p++)
      arr[p]=temp[p];
}
```



Counting Sort Algorithm -

>> 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.

>> The count is stored in an auxiliary array and the sorting is done by mapping the count as an index of the auxiliary array. This mapping is done by performing arithmetic calculations on those counts to determine the positions of each key value(unique element) in the output sequence.

>> It is often used as a subroutine in another sorting algorithm, radix sort, that can handle larger keys more efficiently.

>> It is NOT a comparison sort.

>> Time Complexity : O(n+k)

>> Space Complexity : O(n+k)

where n is the number of elements in input array and k is the range of input.

Counting Sort Algorithm (Working)

Working -

Step 1 - Take input array & range(no of unique integer values involved)

Step 2 - Create the output array of size same as input array.
Create count array with size equal to the range & initialize values to 0.

Step 3 - Count each element in the input array and place the count at the appropriate index of the count array

Step 4 - Modify the count array by adding the previous counts(cummulative). The modified count array indicates the position of each object/element in the output array.

Step 5 - Output each object from the input array into the sorted output array followed by decreasing its count by 1.

Step 6 - Print the sorted output array.

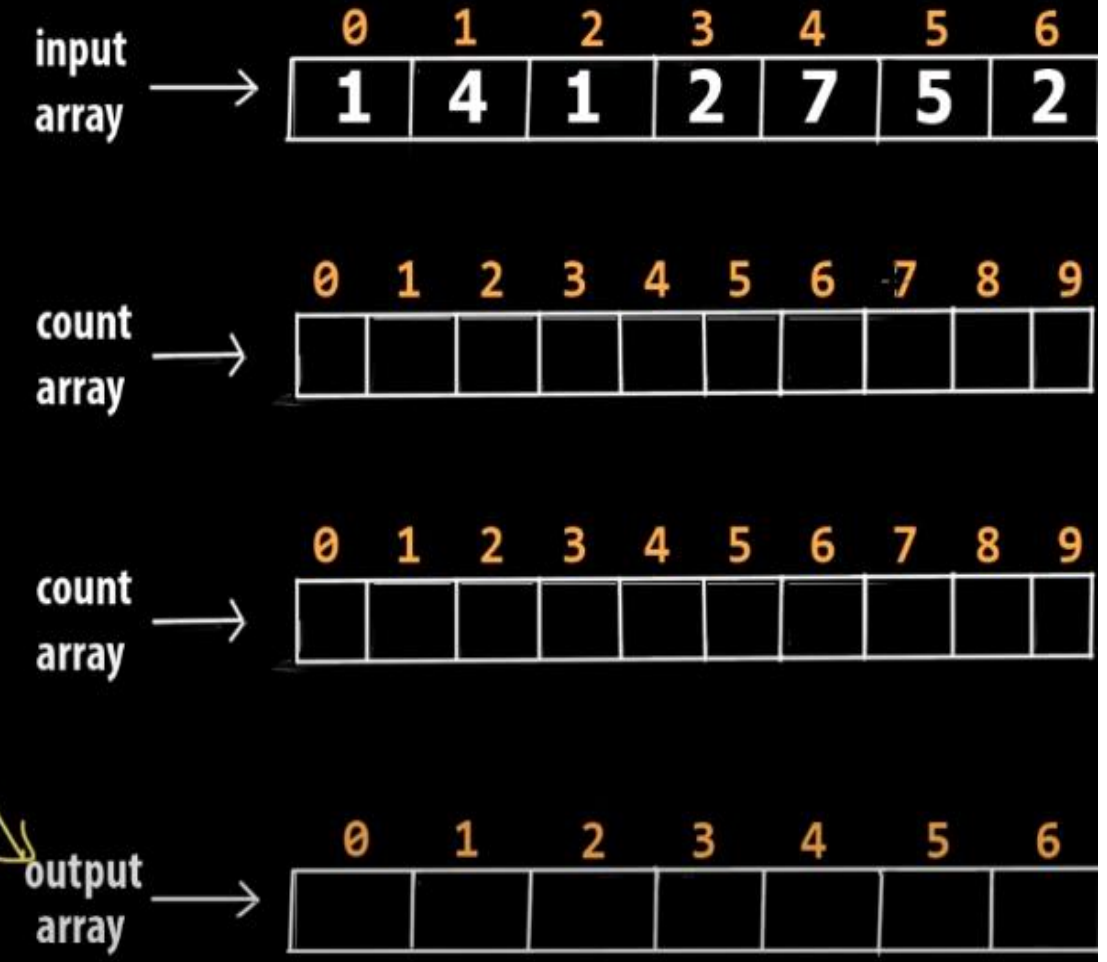
```
int arr[7] = {1, 4, 1, 2, 7, 5, 2}  range - 0-9
```

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

Pseudocode -

```
CountingSort()
{
  1. take input_array[size]
  2. create output_array[size]
  3. take range (or no of unique elements)

  4. for(int i=0 to i<range)          // create count_array[range] &
     4.1 count_array[i] = 0           // initialize all values to 0

  5. for(int i = 0 to i<size)         // Count each element &
     5.1 ++count_array[input_array[i]] // place it in the count_array

                                      // Modify count_array[] to store
  6. for(int i = 1 to i < range)      // previous counts (cummulative)
     6.1 count_array[i]=count_array[i]+count_array[i-1]

  7. for(int i=0 to i<size)           // Place elements from input_array[] to output_array[] using
                                      // this count_array[] that has the actual positions of elements
     7.1 output_array[--count_array[input_array[i]]] = input_array[i]

  8. for(i=0 to i<size)   // Transfer sorted values from output_array[] to input_array[]
     8.1 input_array[i]=output_array[i]
}
```

```
int arr[7] = {1, 4, 1, 2, 7, 5, 2}
range - 0-9        size =
      = 10
```



Radix Sort Algorithm -

>> Radix sort is a non-comparative sorting algorithm.

>> It avoids comparison by creating and distributing elements into buckets according to their radix.

>> For elements with more than one significant digit, this bucketing process is repeated for each digit, while preserving the ordering of the prior step, until all digits have been considered. For this reason, radix sort has also been called bucket sort and digital sort.

>> Typically Radix sort uses counting sort as a subroutine to sort.

>> Radix sort has linear time complexity which is better than O(nlog n) of comparative sorting algorithms.

>> Time Complexity : O d(n+k)

>> Space Complexity : O(n+k)

where n is the number of elements in input array and k is the range of input.

```
int arr[8] = {170, 45, 75, 90, 802, 24, 2, 66}
```

0	1	2	3	4	5	6	7
170	45	75	90	802	24	2	66

Radix Sort Algorithm - (Working)

Working -

0 - 9 = 10 unique element

Step 1 - Take input array & find the MAX number in the array

Step 2 - Define 10 queues each representing a bucket for each digit from 0 to 9.

Step 3 - Consider the least significant digit of each number in the list which is to be sorted.

Step 4 - Insert each number into their respective queue based on the least significant digit.

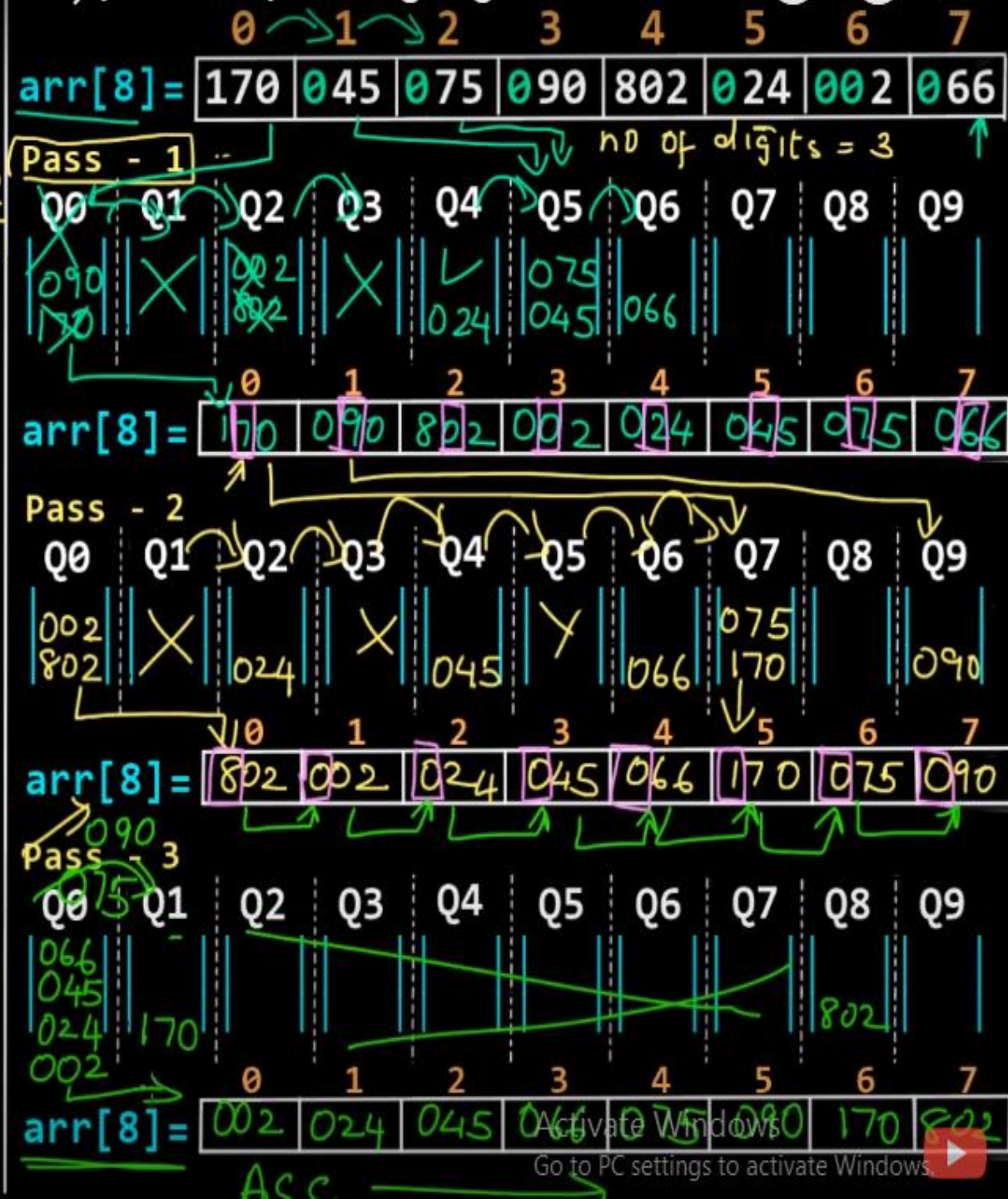
Step 5 - Group all the numbers from queue 0 to queue 9 in the order they have inserted into their respective queues.

Step 6 - Repeat from step 3 based on the next least significant digit.

Step 7 - Repeat from step 2 until all the numbers are grouped based on the most significant digit.

```
int arr[8] = {170, 45, 75, 90, 802, 24, 2, 66}
```

0	1	2	3	4	5	6	7
170	45	75	90	802	24	2	66



Pseudocode -

```
RadixSort(arr, size)
{
  1. take arr[size]
  2. Get max from the arr
     2.1 m = GetMax(arr,size)
  3. Do counting sort for every digit.
     3.1 for (int div = 1; m/div > 0; div *= 10)
         3.1.1 CountingSort( arr, size, div )
 }

GetMax(arr, size)
{
  1. max = arr[0]
  2. for (int i = 1; i <size; i++)
     2.1 if (arr[i] > max)
         2.1.1 max = arr[i]
  3. return max
}
```

Pseudocode -

div = 1, 10, 100

```
CountingSort(arr,size,div)
{
   1. create output[size]
   2. take range (or no of unique elements) // in our case range = 10
   3. for(int i=0 to i<range)
      3.1 count[i] = 0
   4. for(int i = 0 to i<size)
      4.1 count[ ( arr[i] /div)%10] ++
   5. for(int i = 1 to i < range)
      5.1 count[i]=count[i]+count[i-1]
   6. for(int i= size-1 to i>=0)
      6.1 output[ count[ ( arr[i] /div)%10] - 1 ] = arr[i]
   7. for(i=0 to i<size)
      7.1 arr[i]=output[i]
```

arr[8] =

0	1	2	3	4	5	6	7
170	45	75	90	802	24	2	66