Video - 48

Introduction to Sorting Algorithm

Sorting

→ Method to arrange a set of elements in either increasing or decreasing order according to some basic or relationship among the elements.

1 9 8 2 7 . . . → sorting

Two way of sorting

① Ascending order
→ 12789

② Descending order
→ 98721

Ques: Why Sorting ?

Swiggy/Zomato → Restaurant → sort by Rating
→ sort by Price

① School Assembly.

↓

we stood a/ height-wise during morning assembly.

The basis of sorting ⇒ height

② Binary Search

↓

Searching in sorted array takes at most $O(\log N)$ time. if not sorted it takes $O(n)$ time.

∴ Retrieval becomes much faster.

③ Dictionary

↓

words are sorted for you to find it easily (lexicographically → sorted)

④ Swiggy, Zomato, flipkart, Amazon

↓

sort → prices
→ offer
→ new launch

⑤ social media application → email, outlooks

Video - 49

Analysis Criteria For Sorting Algorithm

(1) Time Complexity

⇒ which algorithm works efficiently for larger data sets and which algorithm works faster with smaller data sets.

⇒ Eg: 1 sorting algo sorts only 4 elements efficiently & fails to sort 1000 element

⇒ Algo 1 → $O(n^2)$

Algo 2 → $O(n \log n)$

∴ Algo 2 better than Algo 1

$O(n \log n)$ better $O(n^2)$

⇒ Lesser the time complexity, the better is the algorithm.

② Space Complexity

⇒ help us to compare the space the algorithm uses to sort any data set.

⇒ If any algorithm consumes a lot of space for larger inputs ⇒ it's considered a poor algorithm for sorting large data sets.

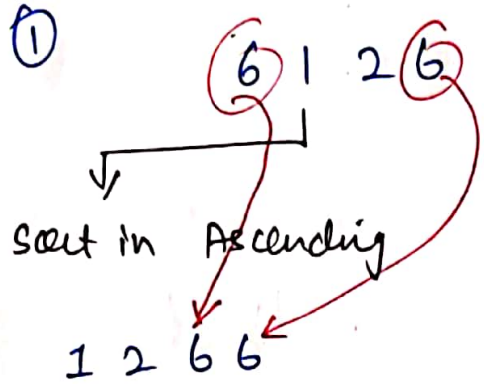
In-place sorting Algorithm

⇒ Algorithm which results in constant space complexity.

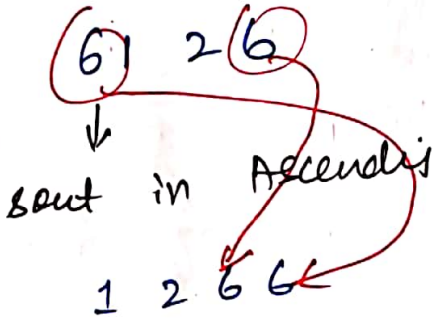
⇒ Mostly use swapping and rearrange technique to sort a data set.
eg: Bubble sort

③ Stability

⇒ whether the order of the elements having equal status when sorted on some basis is preserved or not.

① 

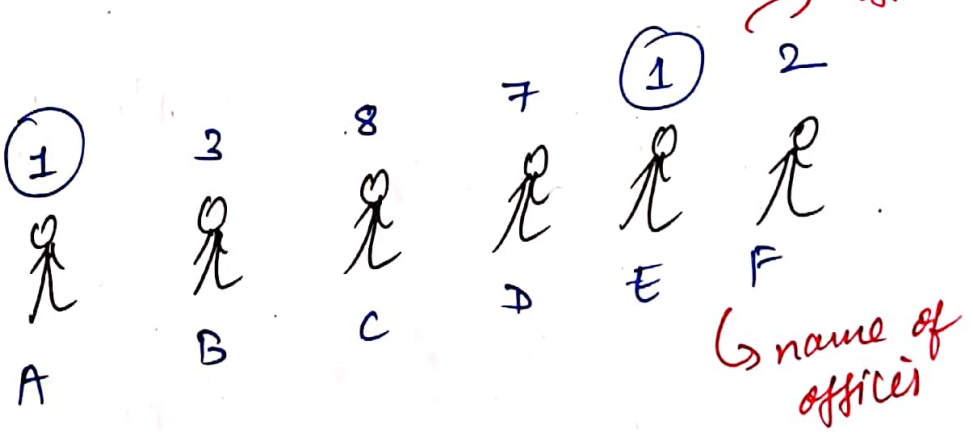
→ stable sorting algo

② 

→ not stable sorting algo

Example :-

store



Sort

A E F B D C → stable sorting Algorithm

⇒ if 2 person with same Rank, then whoever comes 1st; he/she

will be served first.

⇒ maintain the order

⇒ same value के order को maintain करती है

④ Internal / External Sorting Algorithm

Internal SA

⇒ All the data is loaded into the memory

External SA

⇒ All the data is not loaded into the memory

memory (RAM)

⑤ Adaptive

⇒ adapt the fact that if the data are already sorted and it must take less time

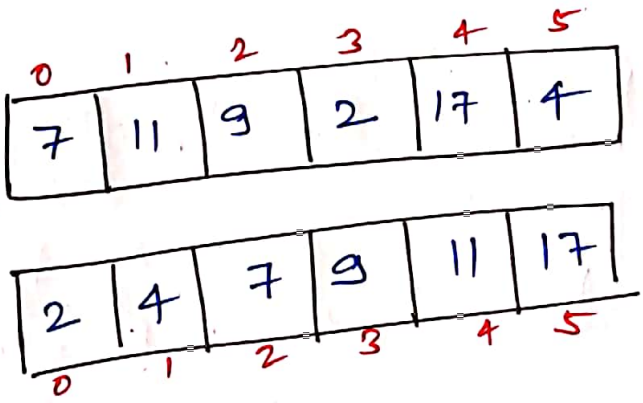
⑥ Recursive / Non-Recursive SA

⇒ Algorithm uses recursion to sort a data set ⇒ Recursive Algorithm

otherwise ⇒ Non-Recursive Algorithm

Video - 50

Bubble Sort Algorithm



⇒ input Array

⇒ output Array

⇒ Sorting in Ascending Order

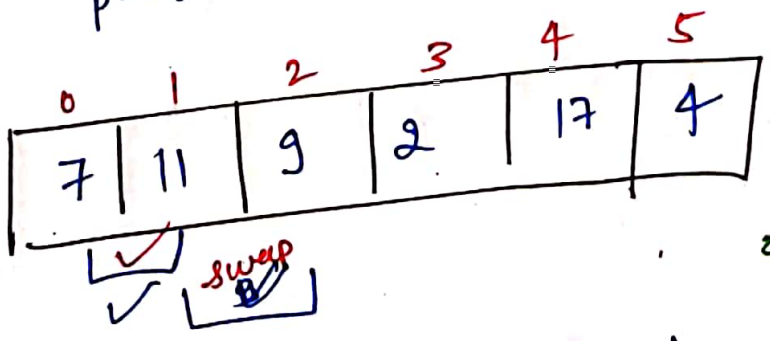
1st Pass → 5 comparison → 5 possible swap

→ whole array → unsorted

→ start by comparing each adjacent pair

→ ∵ array size = n

then pair to compare = (n-1)



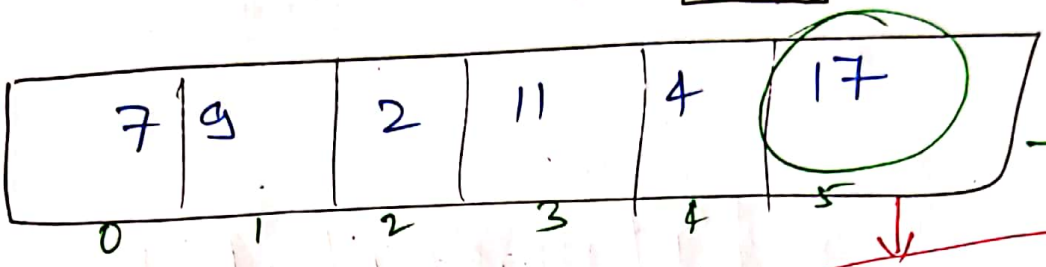
7 9 11 2 17 4 → 1 | 2

swap

7 9 2 11 17 4 → 2, 3

7 9 2 11 17 4 → 3, 4

swap



→ 4, 5

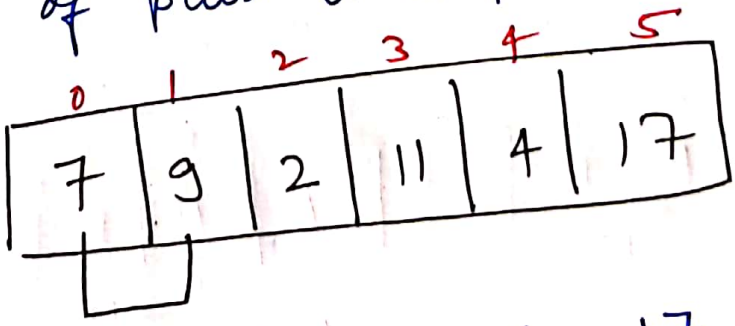
1st Pass complete here

last index of array gets sorted

→ now we have to sort only upto 4th index of array

2nd Pass → 4 comparison → 4 possible swap

→ begining se start karo again swapping & no. of pair to compare = 4

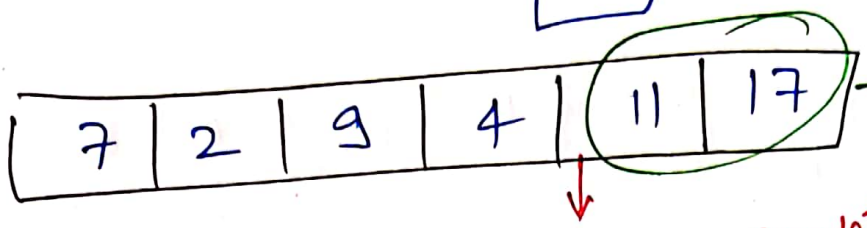


→ 4 comparison

7 9 2 11 4 17 → 0, 1

7 2 9 11 4 17 → 1, 2

7 2 9 11 4 17 → 2, 3



→ 3, 4

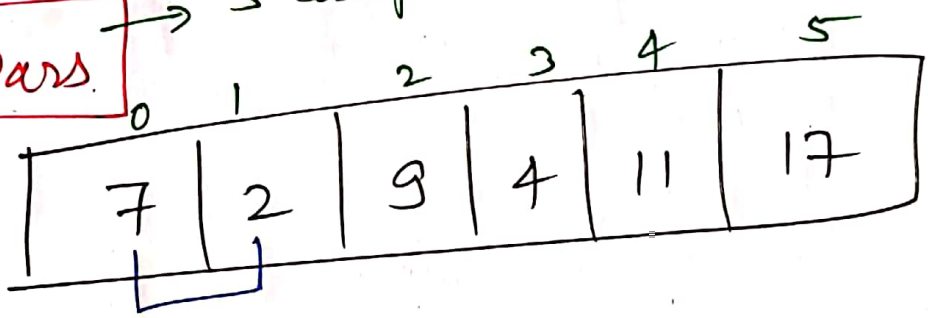
2nd pass Complete

↓

last 2 index sorted

11 17

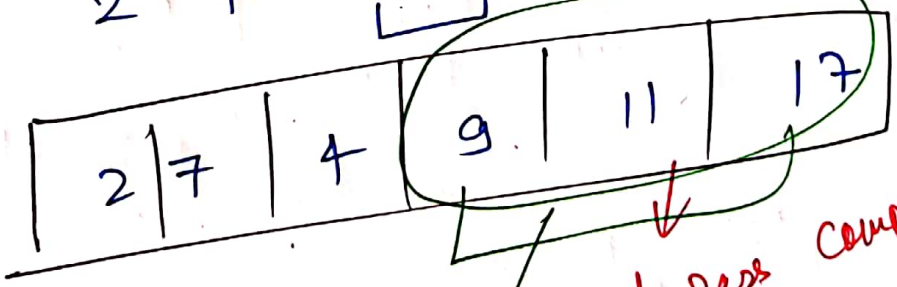
3rd pass → 3 comparisions → 3 possible swap



2 7 9 4 11 17 → 0, 1

2 7 9 4 11 17 → 1, 2

2 7 9 4 11 17 → 2, 3

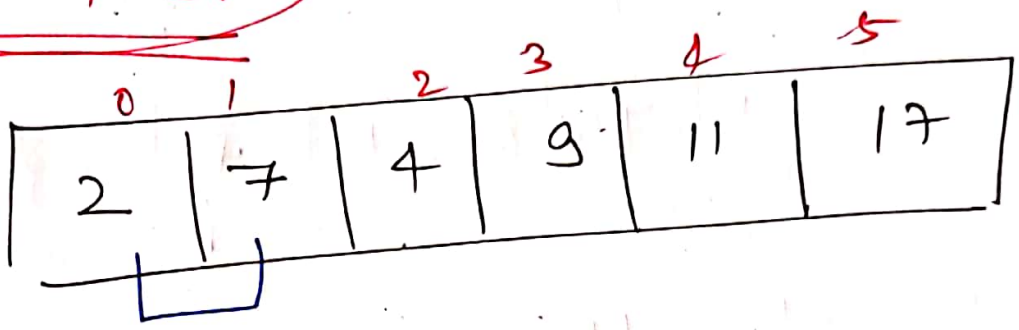


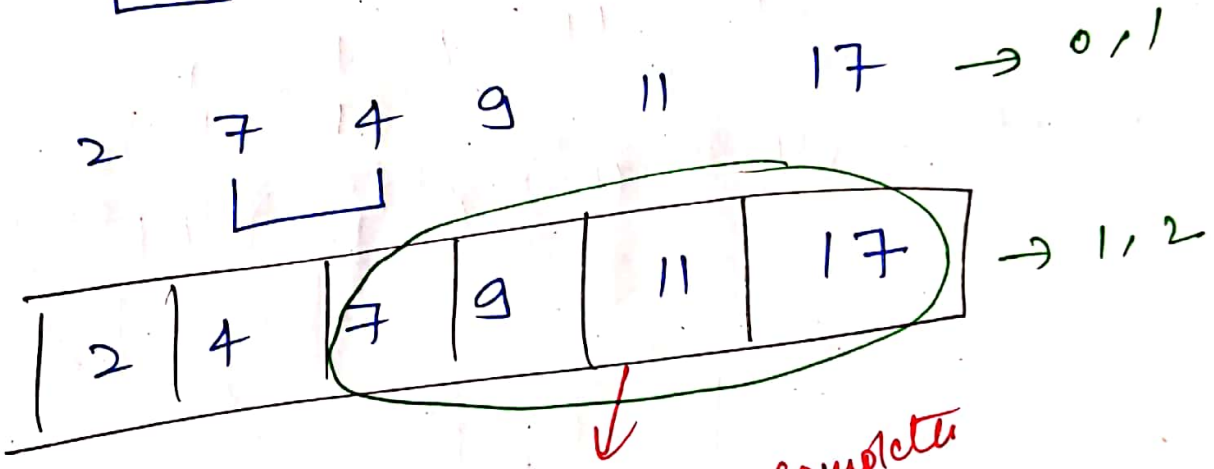
3rd pass completed

↓

Sorted (last 3 indx)

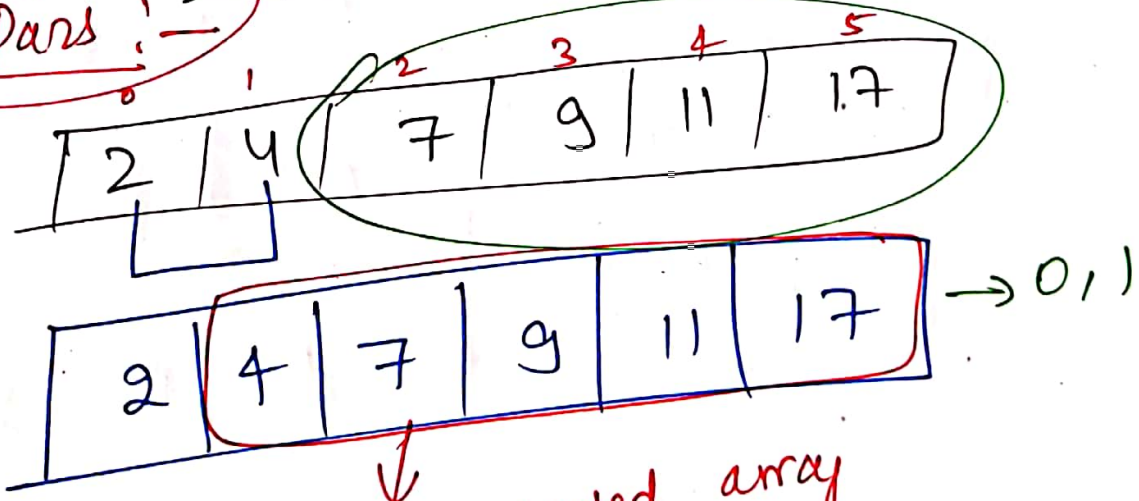
4th Pass :- → 2 Comparison → 2 pos. Swap





4th pass complete

5th Pass :- → 1 comparison → 1 possible swap



got sorted array

5th pass gives :⇒ Sorted Array

Note:-

Length of Array = 6 (n)

No. of Passes = 5 $(n-1)$

Note:- (1) Comparision Number (Total)

Total no. of comparison for Array

$= (5+4+3+2+1)$

$= 15$ comparison

for General:

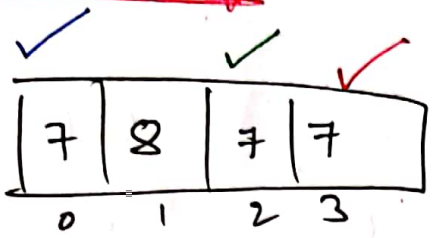
$= 1+2+3+4+\cdots+(n-1)$

$= \frac{n(n-1)}{2} = O(n^2)$ (time Complexity)

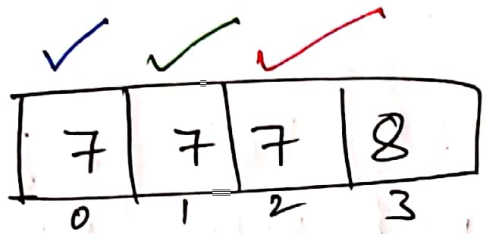
$(n-1)+(n-2)+(n-3)+\cdots+1$

$= \frac{n(n-1)}{2} = O(n^2)$

② Stability



→ Unsorted array

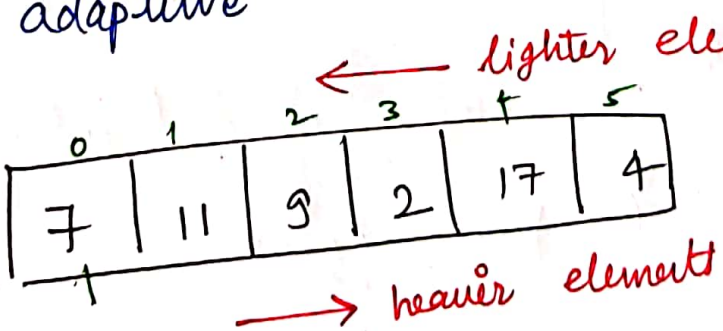


→ sorted array

⇒ order is same as the order in input array

⇒ Here, Stable → Bubble sort is stable

⇒ Bubble sort is not adaptive by defaut, But we can make it adaptive



Time Complexity

O(n) → adaptive Bubble Sort → if already sorted array is given

$O(n^2)$ → otherwise

Video - 51

Bubble Sort Coding

$i=0$ 1st Pass : 5 comp. ; $(n-1)$ comp.

$i=1$ 2nd Pass : 4 comp. ; $(n-2)$ "

$i=2$ 3rd Pass : 3 comp. ; $(n-3)$ "

$i=3$ 4th Pass : 2 comp. ; $(n-4)$ "

$i=4$ 5th Pass : 1 comp. ; $(n-5)$ "

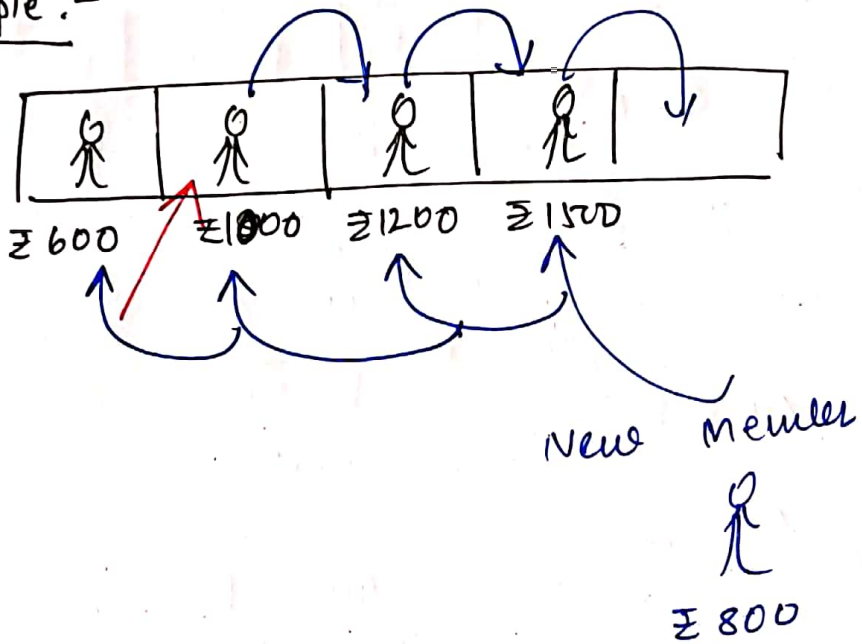
$(n-1-i)$

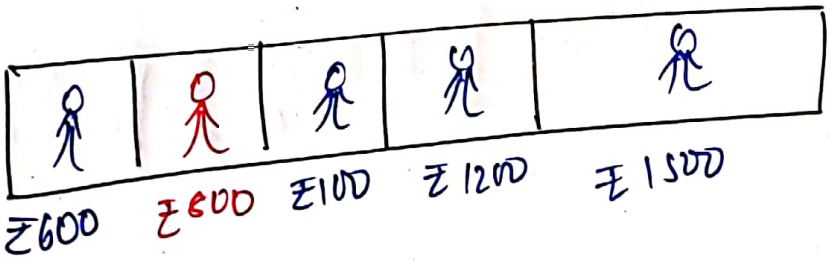
Size = 6 = n

Video - 52

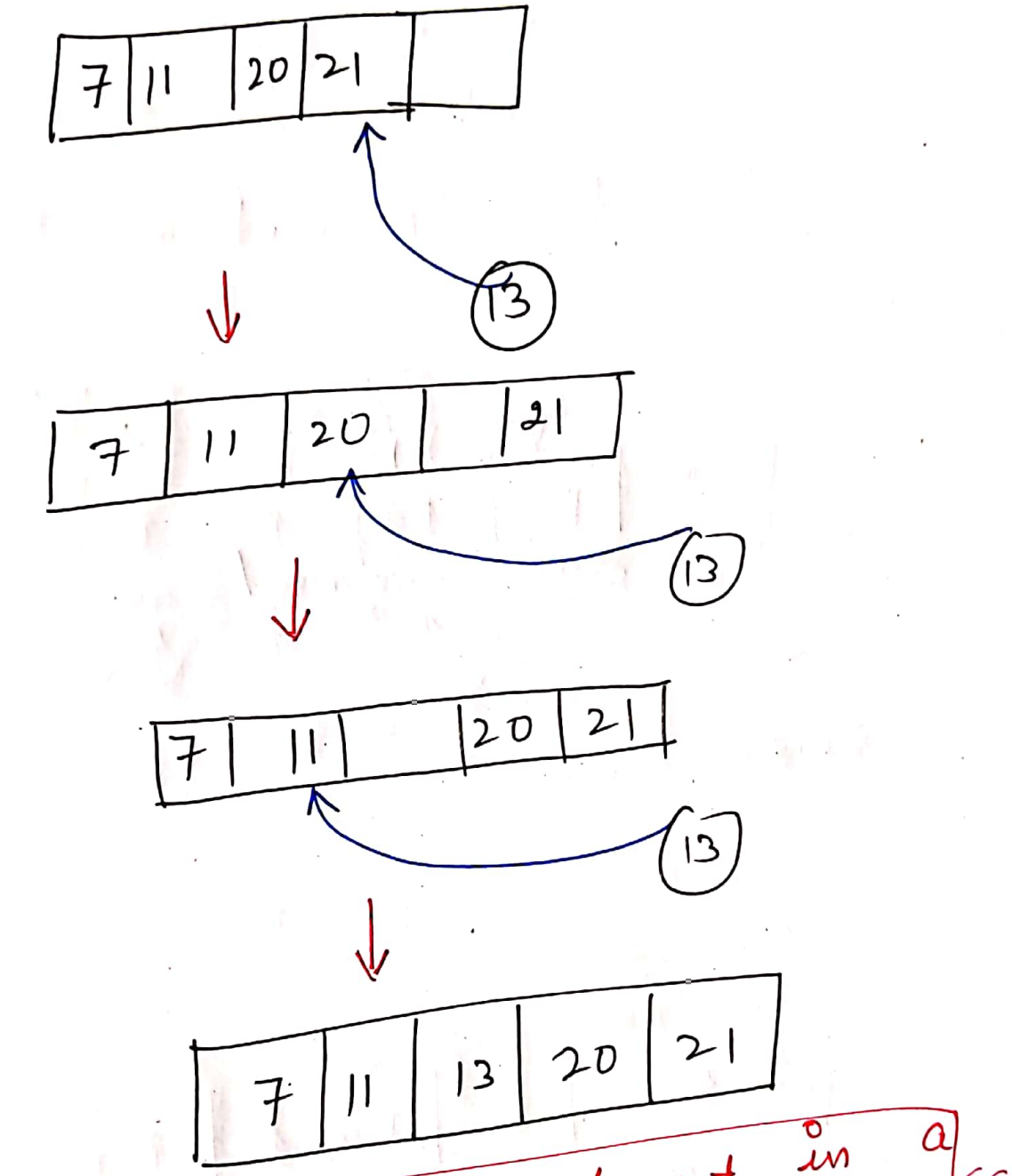
Insertation Sort Algorithm

Example :-



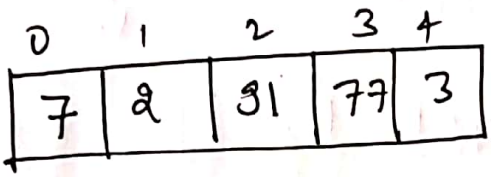


⇒ insert hua new member + sorting of array is conserved.



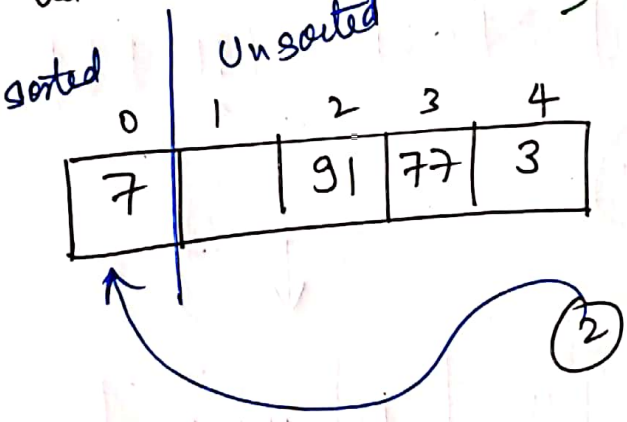
* Inserting an element in a Sorted Array *

⇒ Use the insertion sort algo to soet its elements in increasing order.



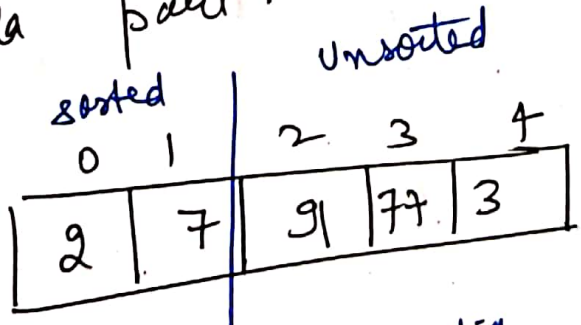
* Array of single element is always sorted.

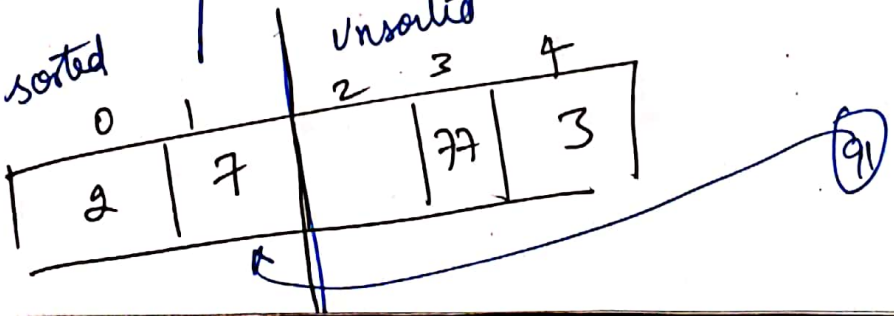
1st Pass → 1 pos. Comp.
1 possible Swap

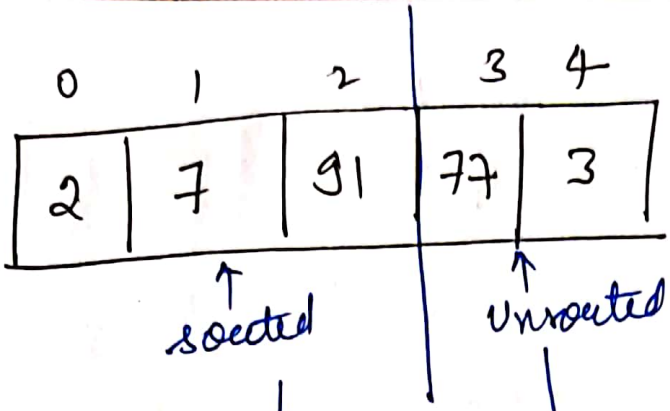


* Pluck 1st element from begging of unsorted part and insert it in sorted wala part.

2nd Pass
2 possible Comparisons
2 possible Swap

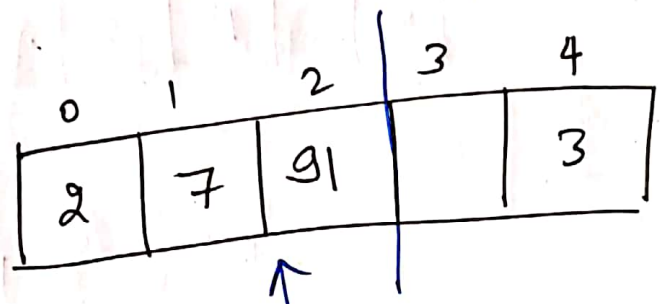


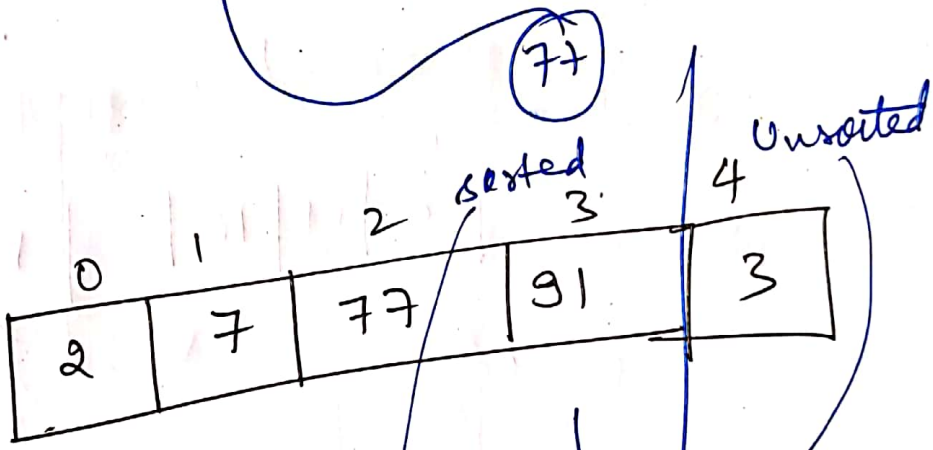




3 possible Comparisons
3 possible Swap

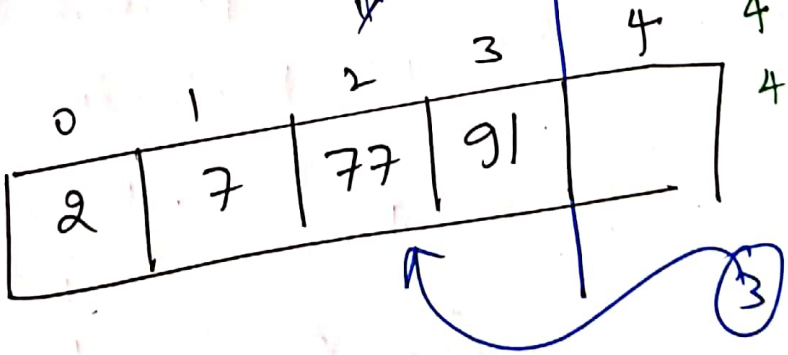
Pass-3

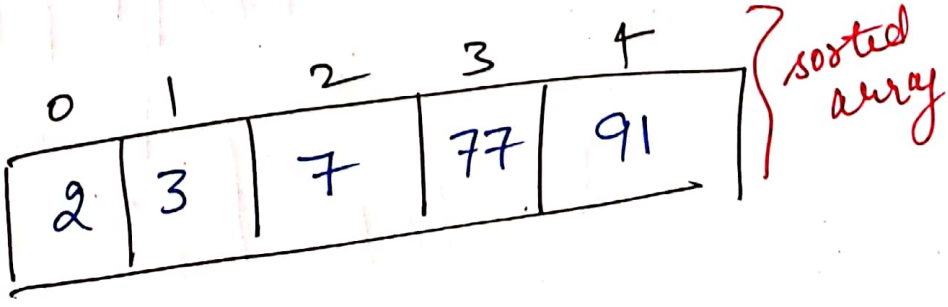




Pass-4

4 pcs. Comp
4 pcs. Swap





Time Complexity

length of array = 5 (n)

Total No. of Pass = 4 $(n-1)$

Total Pass. Comp./Swap = $1+2+\cdots+(n-1)$

$= \frac{n(n-1)}{2}$

$= O(n^2)$

↓ worst Case Complexity

Best Case Complexity

↓ Already Sorted Array

Total Comp = $(n-1)$

$= O(n)$

↓ Best Case.

Analysis

① Total passes = $(n-1)$

② Total Comp. = $1+2+\cdots+(n-1)$

$= \frac{n(n-1)}{2}$

$= O(n^2)$ → worst case

③ Best Case Total Comp $= O(n)$

④ Stability → Yes, its stable

⑤ Adaptive → Yes, it is adaptive (when the array is sorted → $O(n)$)

↑ By nature this is adaptive in nature

* No Intermediate Result

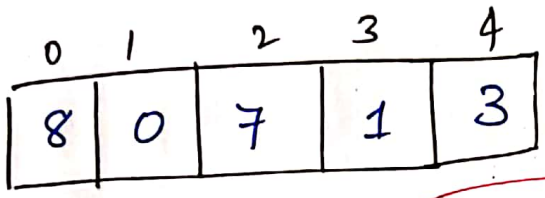
* But! Bubble sort → intermediate result → last index sorted out.

Video - 53

Insertation Sort in V.S. Code

Video - 54

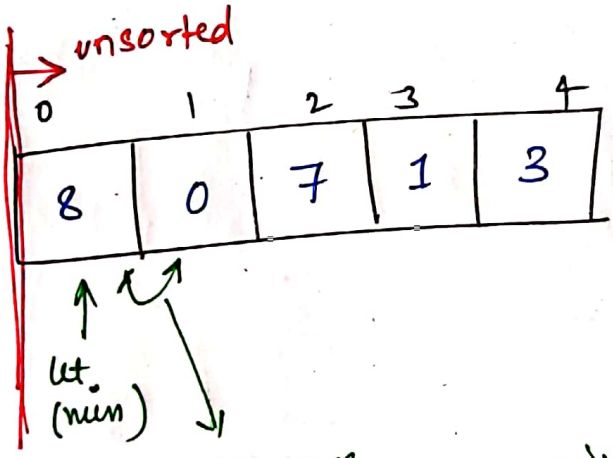
Selection Sort Algorithm

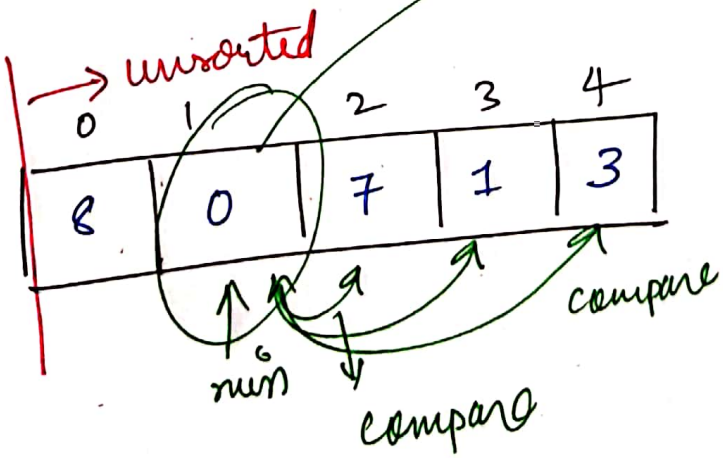


length of array = 5

Unsorted Array

Note

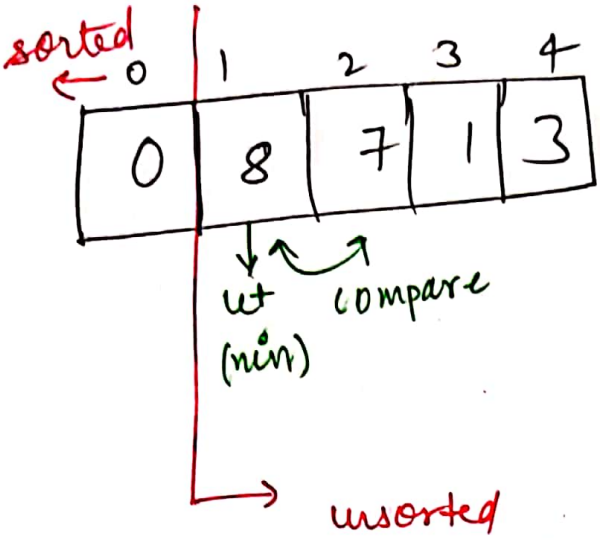


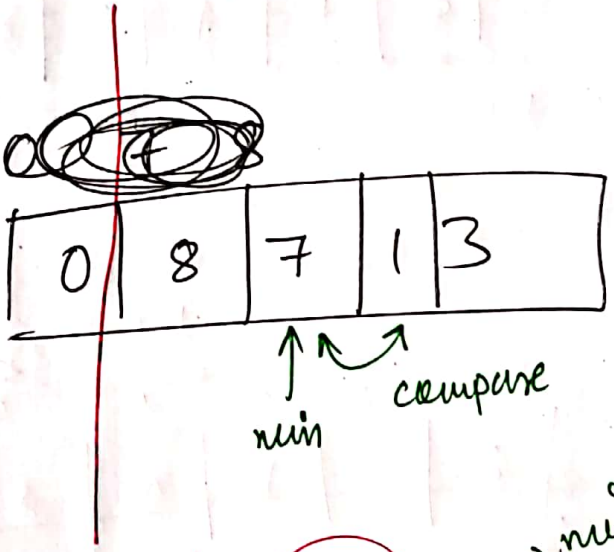


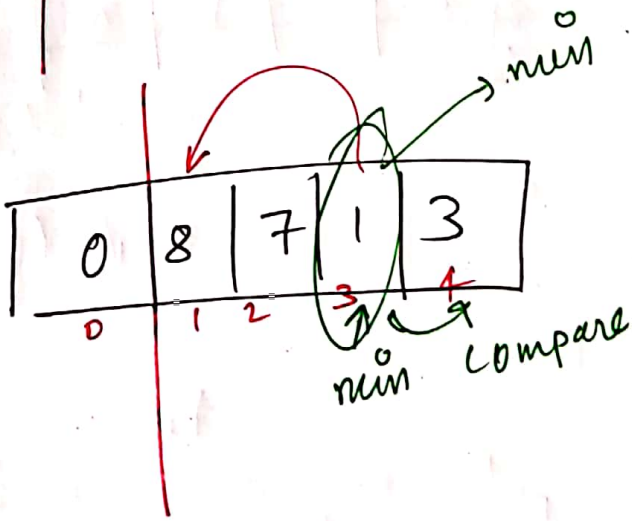
min ⇒ "0"

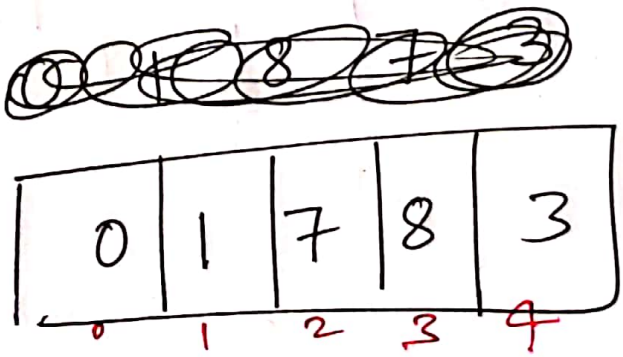
exchange min with "8" ↓ 1st position element

Pass-2

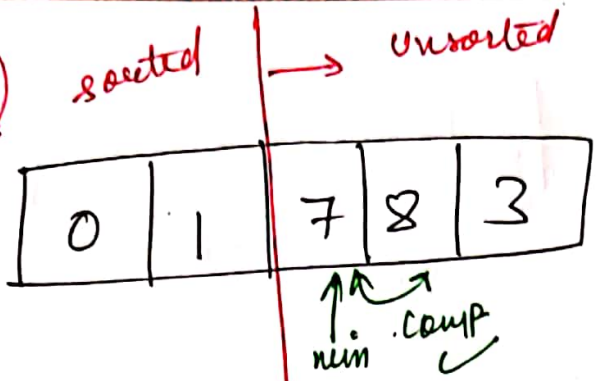


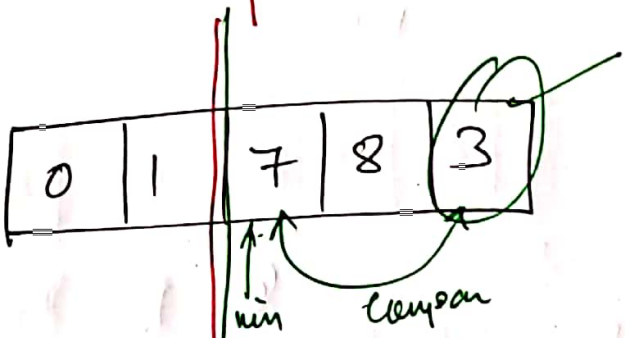




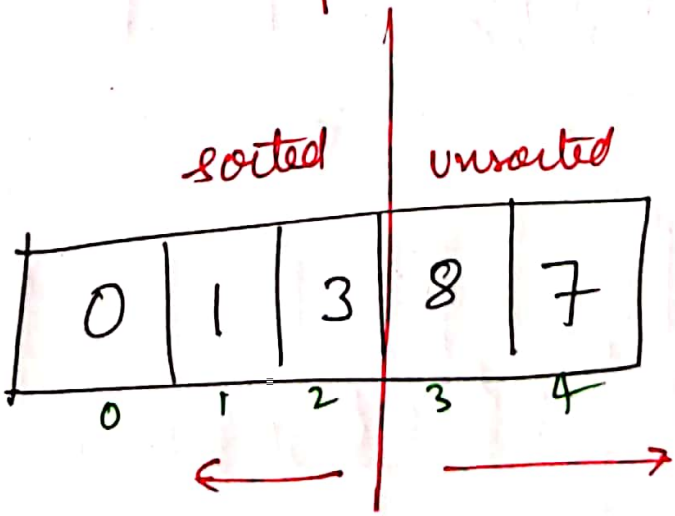


Pass 3

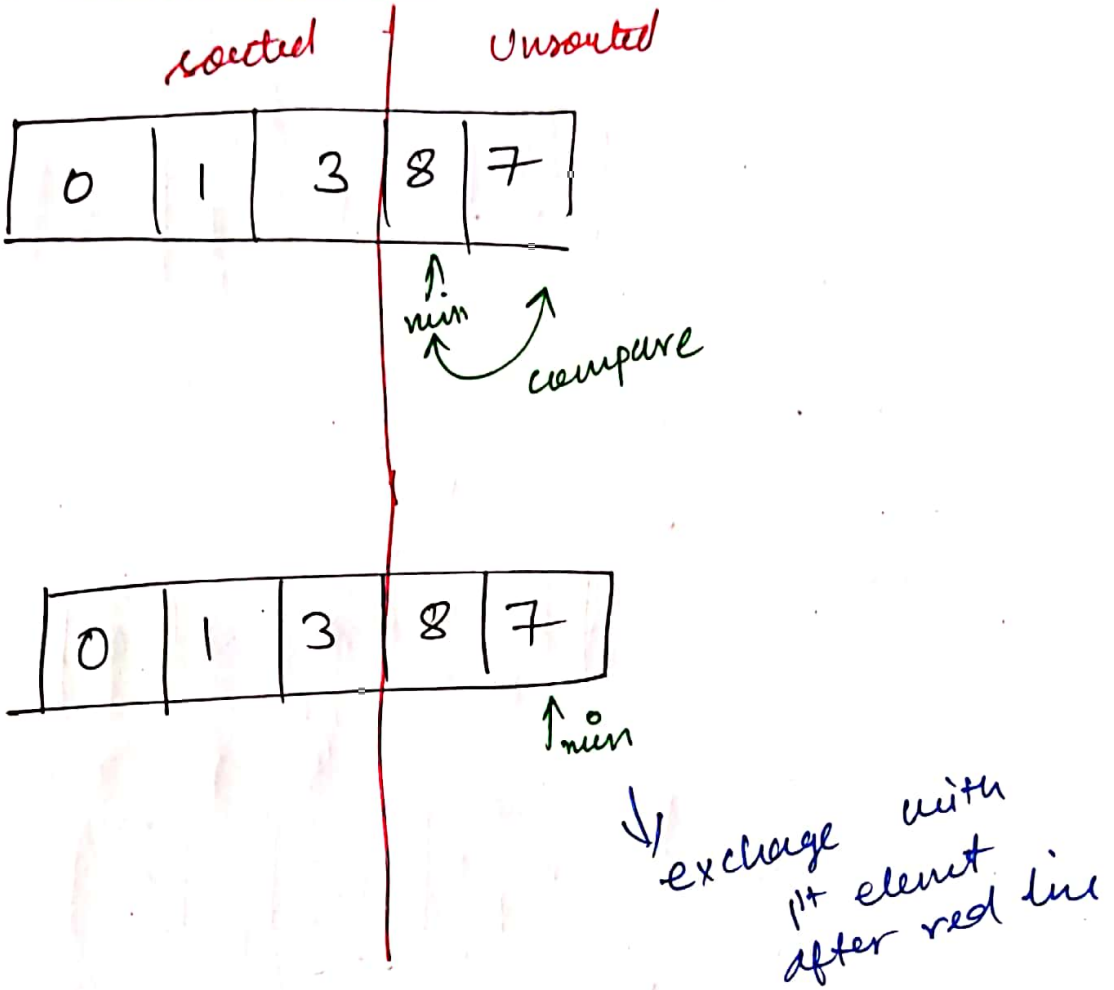


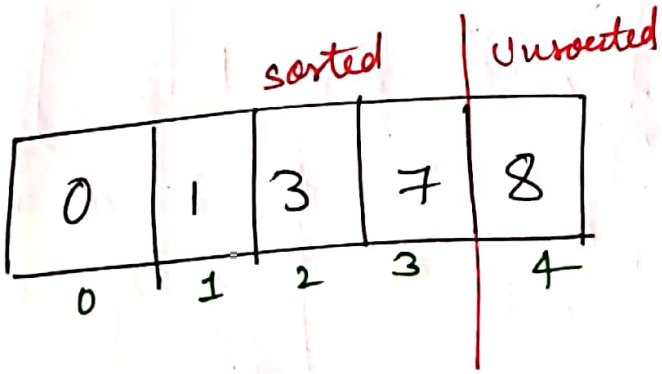


↓ exchange min with 1st elemt after red line



Pass 4





⇒ ∵ a single element is always sorted,
∴ we ignore unsorted part & make it sorted too.

Detailed Explanation

0	1	2	3	4
8	0	7	1	3

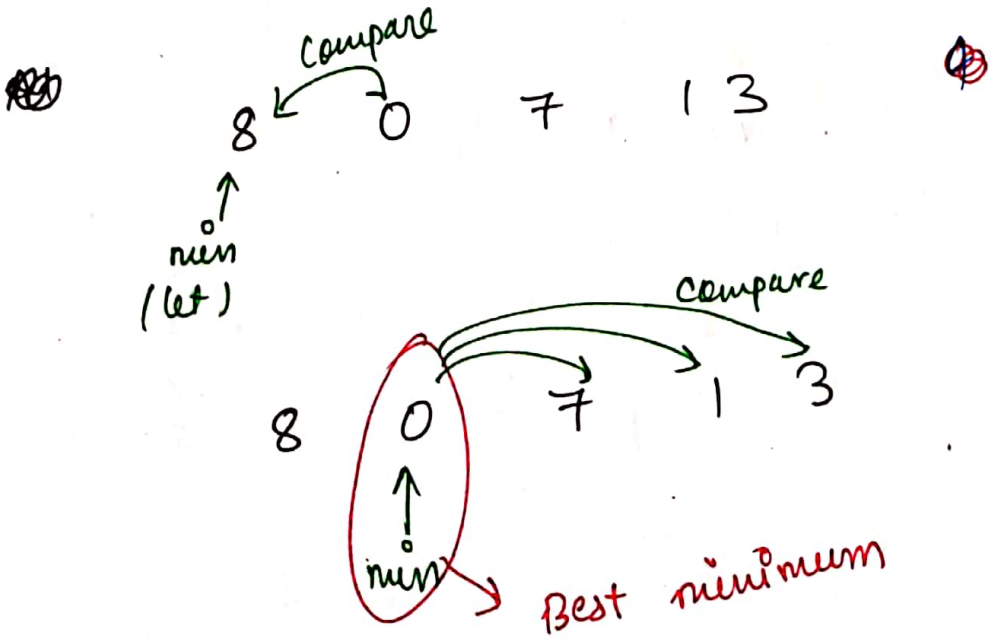
Unsorted Array

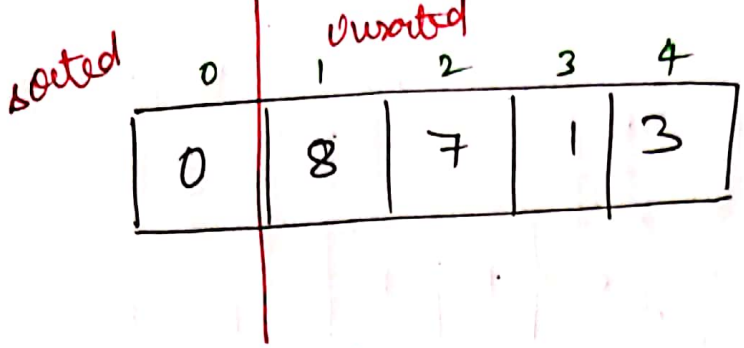
1st Pass:- (4 Comp.)

Consider element at index = 0 as minimum. and then compare it with each rest element of array; as you proceed and get other value less than that at index = 0; make it as "min" and again compare "new min" with rest element.

Check the Best minimum.

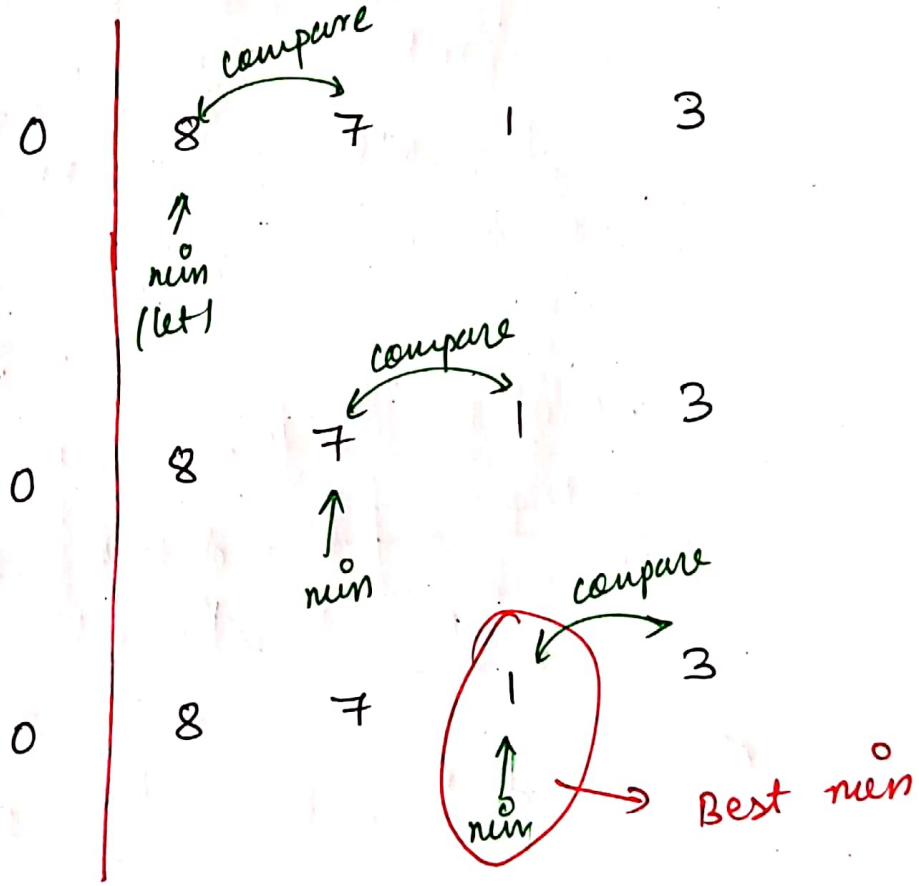
Then, Swap the Best minimum with element at index = 0.



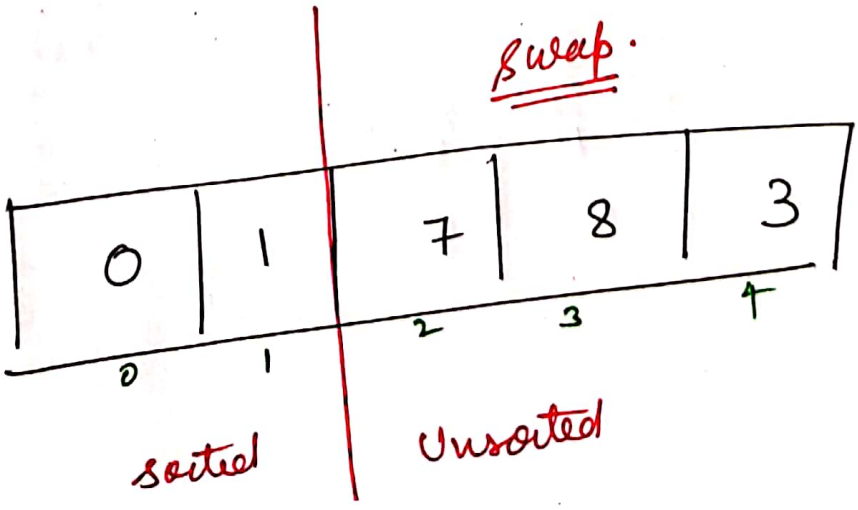


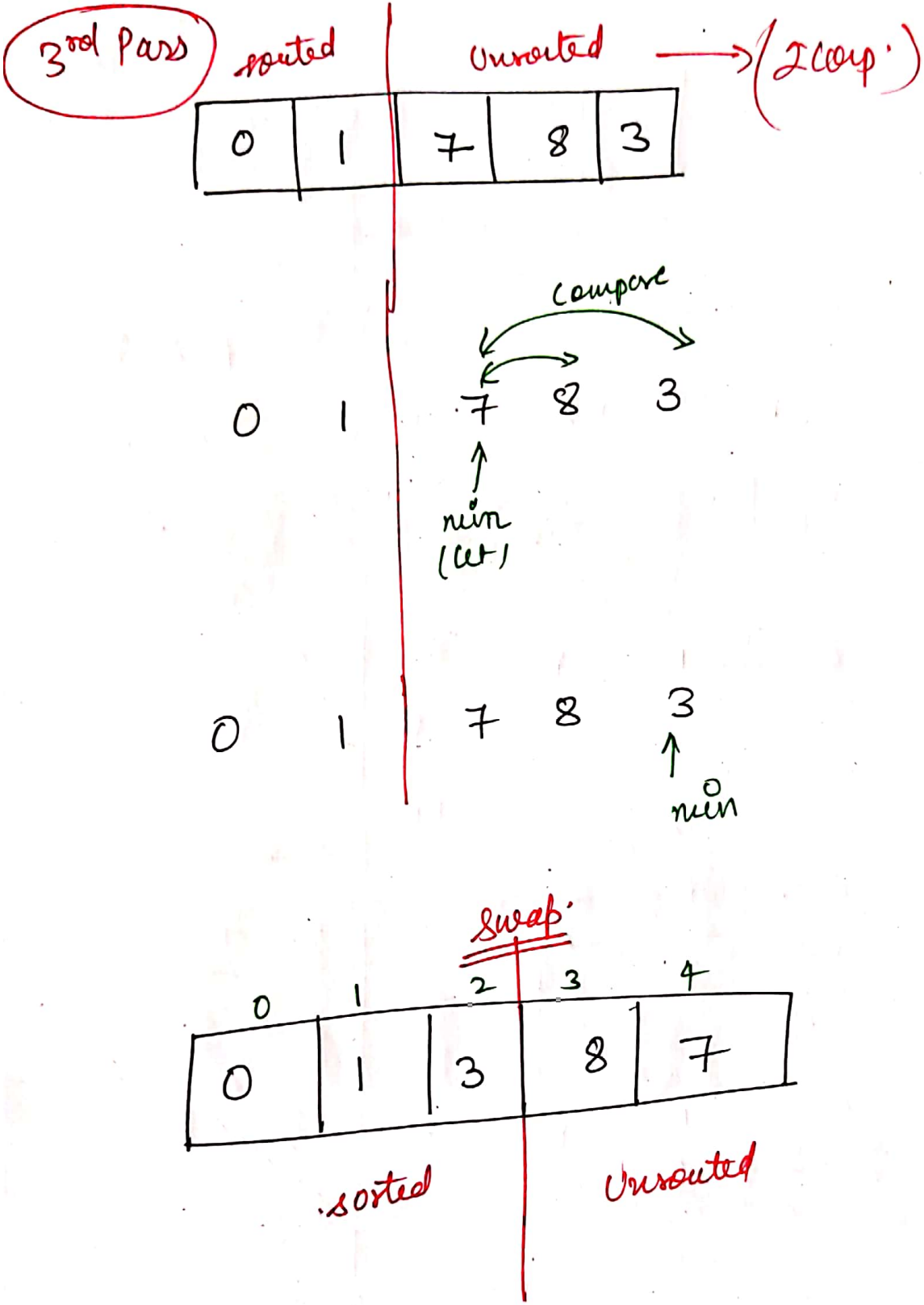
2nd Pass. → (3 Comp.)

Now repeat pass 1 from begining of unsorted array i.e., from index = 1



Swap.

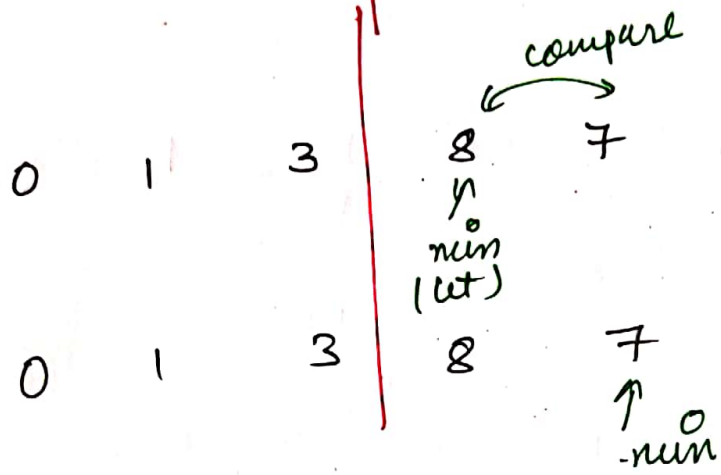




4th Pass:

sorted | Unsorted

⟶ (1 Comp.)

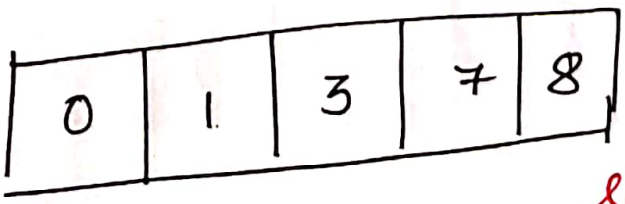


swap

sorted | Unsorted

Note:-

A single Element is always sorted we ignore unsorted part & make it sorted.



Sorted Array.

Note:-

① length of array = 5 (n)

② Total no. of pass = 4 (n-1)

③ Possible Comparison = 1 + 2 + ... + (n-1)

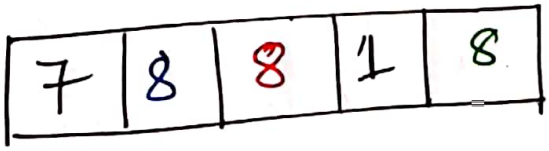
$$= \frac{n(n-1)}{2}$$

Time complexity $= O(n^2)$

④ Max. Swap = n-1

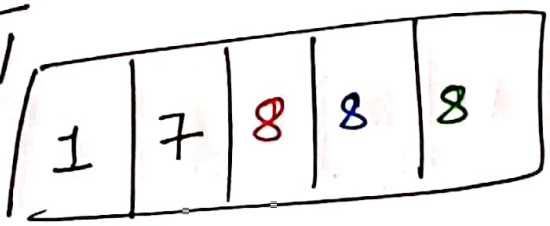
⑤ Min. Swap = 0 [for Sorted Array]

⑥ Stability & Adaptive



Unsorted array

after sorting



→ sorted array.

⇒

⇒ Selection Sort is not stable.

⇒ Selection Sort is Not ~~Adaptive~~

↓

comparison उतना ही करना पड़ता है

⇒ sorting in minimum no. of swaps.

Video - 5-5

Selection Sort in V.S. Code

Video - 56

Quick - Sort Algorithm

Procedure in short

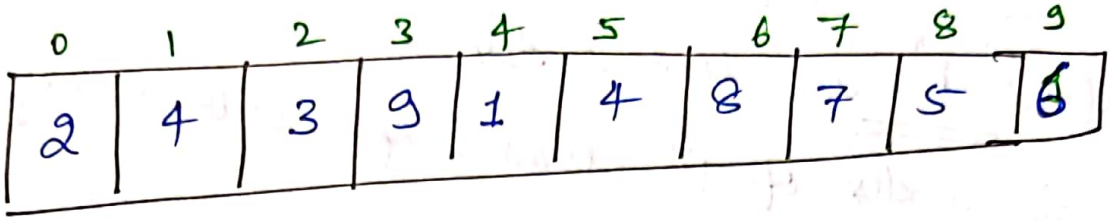
1. i = low
2. j = high
3. pivot = low
4. i++ until element > pivot is found
5. j-- until element ≤ pivot is found
6. Swap A[i] & A[j] and repeat step 4 & step 5 until (j ≤ i)
7. Swap pivot & A[j]

Example:

length of array = 10

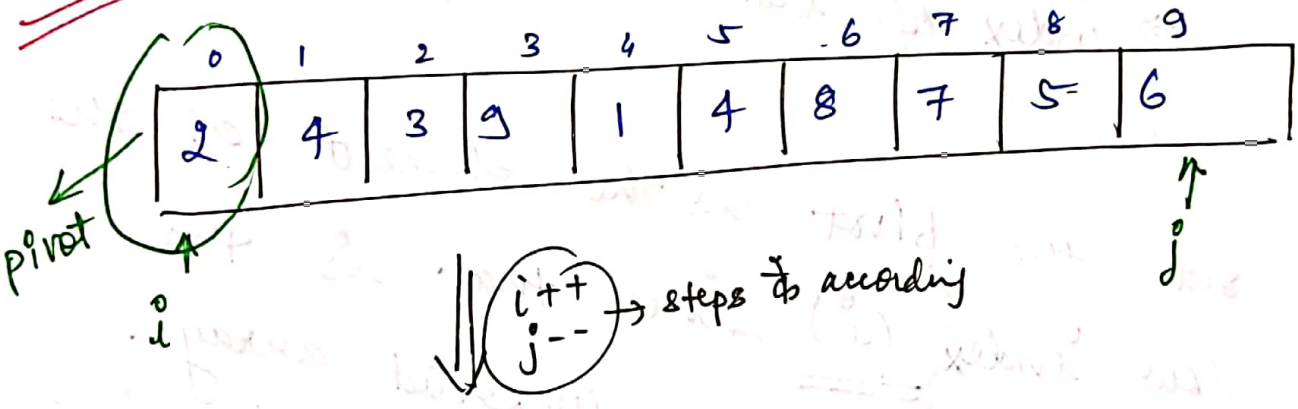
Array

Ques



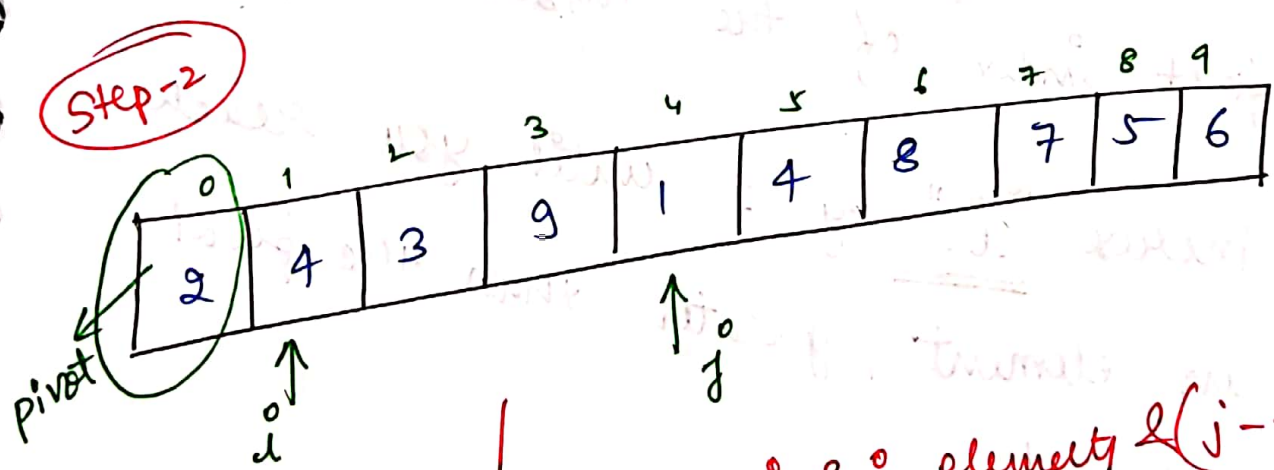
Unsorted Array

Solution Step-1

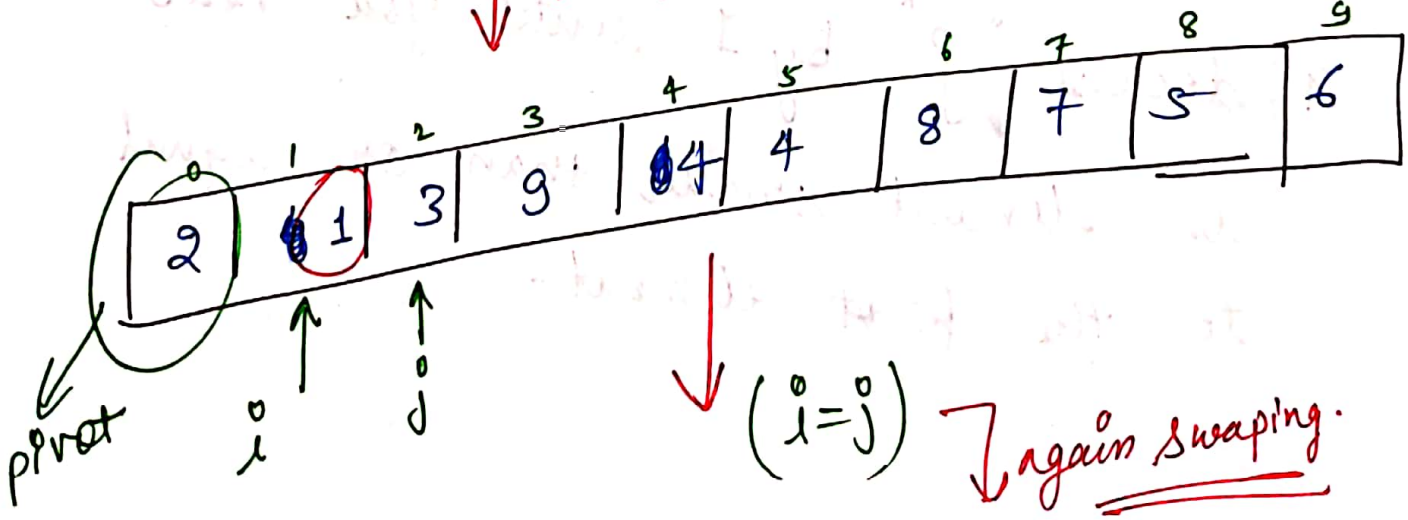


i++ j-- → steps ↑↓ according

Step-2



swap i & j elements & (j--)



(i=j) → again swaping.

Procedure

Partitioning Algorithm

① define "i" as low index

⇒ index of 1st element of sub array

define "j" as high index

⇒ index of last element of sub-array

② Set the pivot as the element at the low index (i) since that is the first index of the unsorted array.

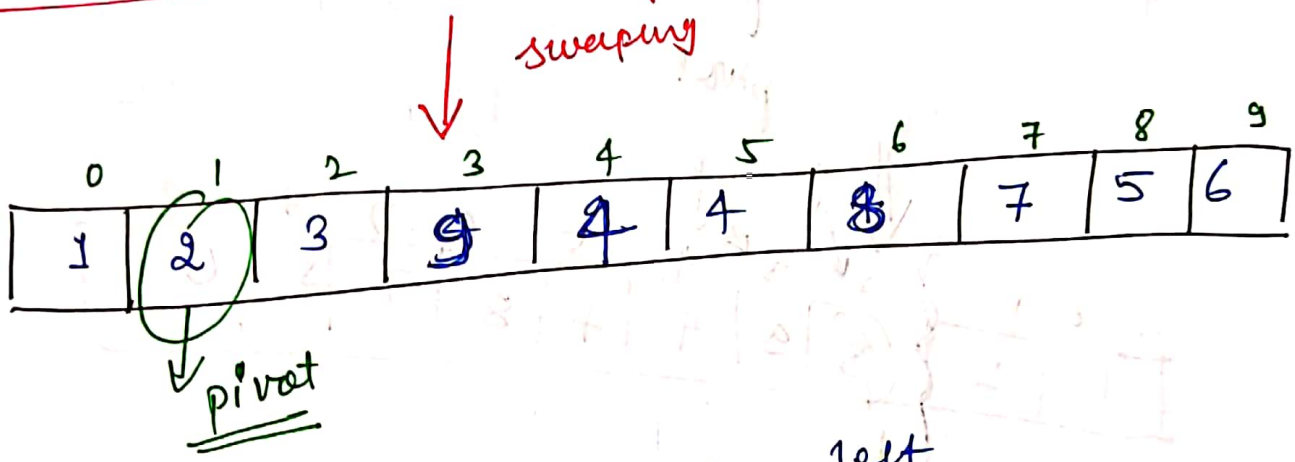
③ increase "i" by 1 until you reach an element greater than the pivot element.

④ decrease "j" by 1 until you reach an element smaller than or equal to the pivot element.

(5) Having fixed the values of "i" and "j" interchange the elements at indices "i" and "j".

(6) Repeat steps 3, 4 & 5 until $j \leq i$

(7) Swap the pivot element & the element at the index.



* element < 2 are on left
  elemt > 2 are on Right

↓

Apply: Divide & Conquer.

↓

are seprate our focus from whole array to just subarrays, which are not sorted yet!

Subarray :-

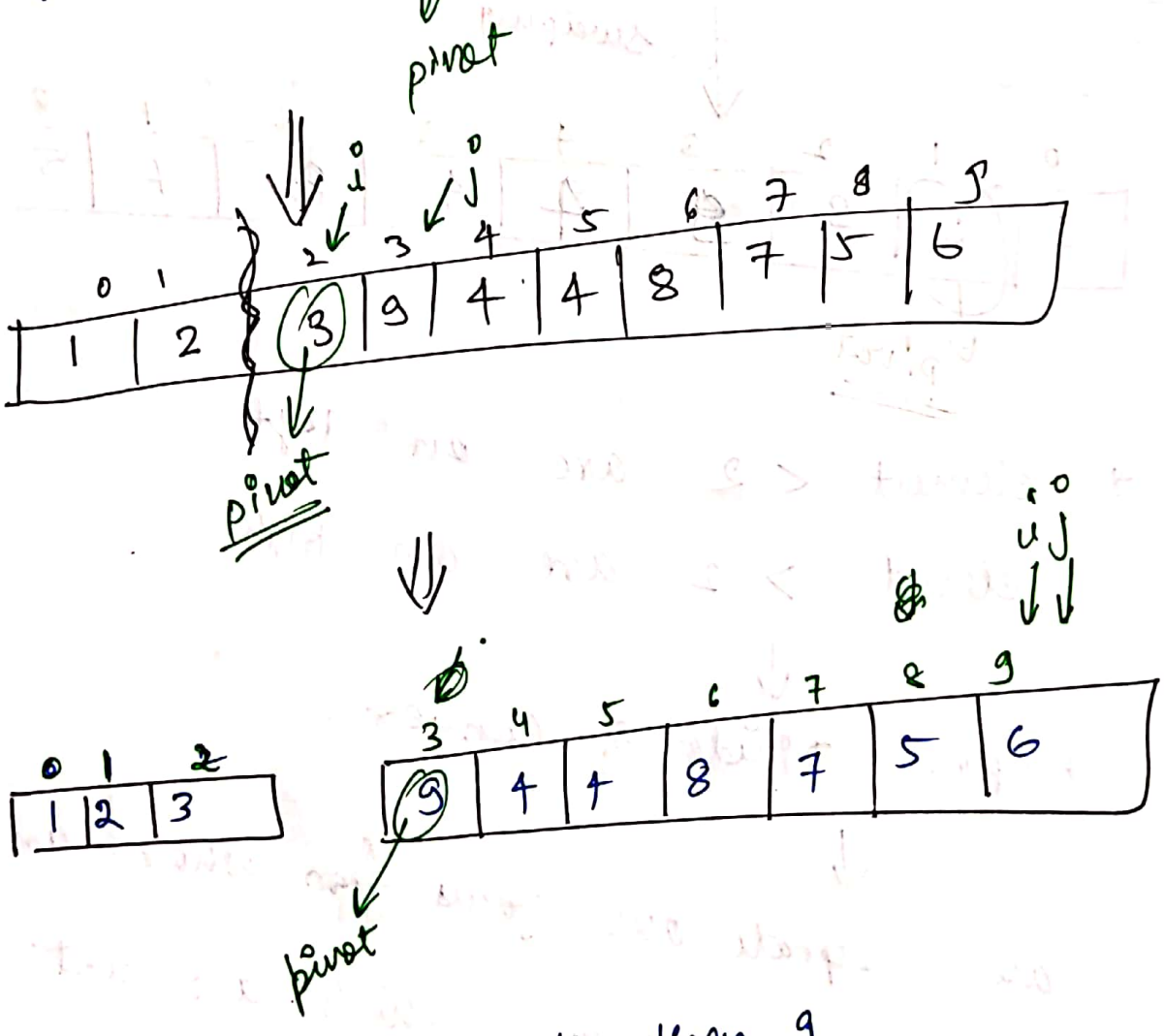
{1} & {3, 9, 4, 4, 8, 7, 5, 6}

↓

∵ 2 is also automatically sorted.

↓

{1, 2} & {3, 9, 4, 4, 8, 7, 5, 6}

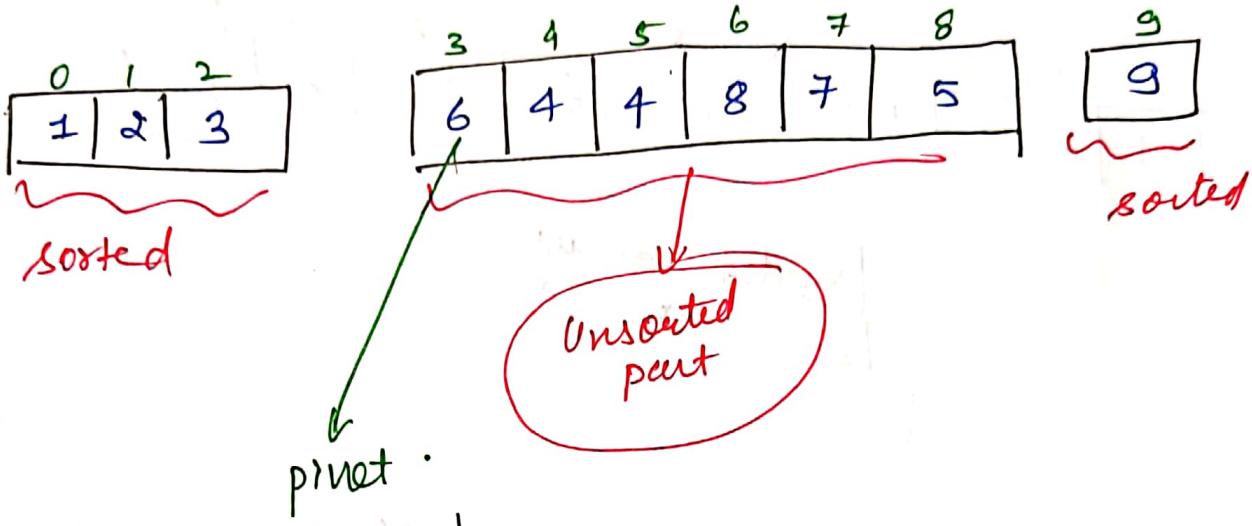


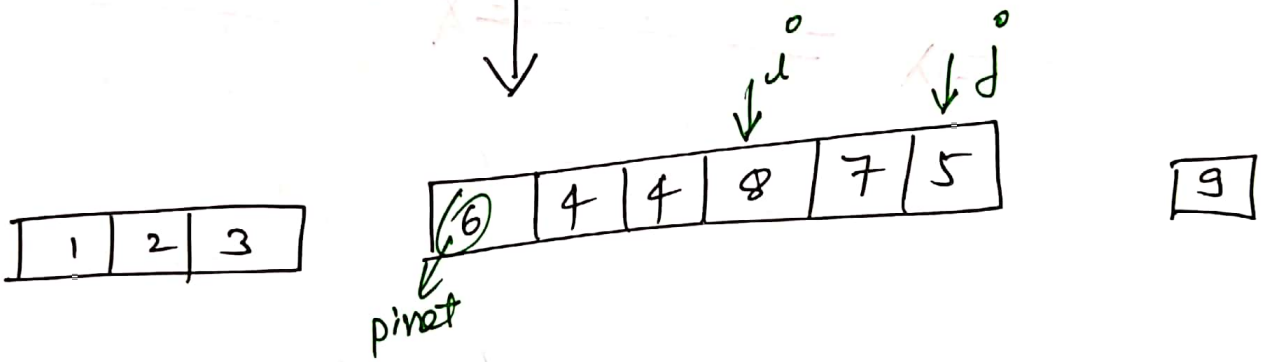
→ no element greater than 9

∴ i at last index reached

and 6 = 1st element j found to be smaller than 9 & they collided.

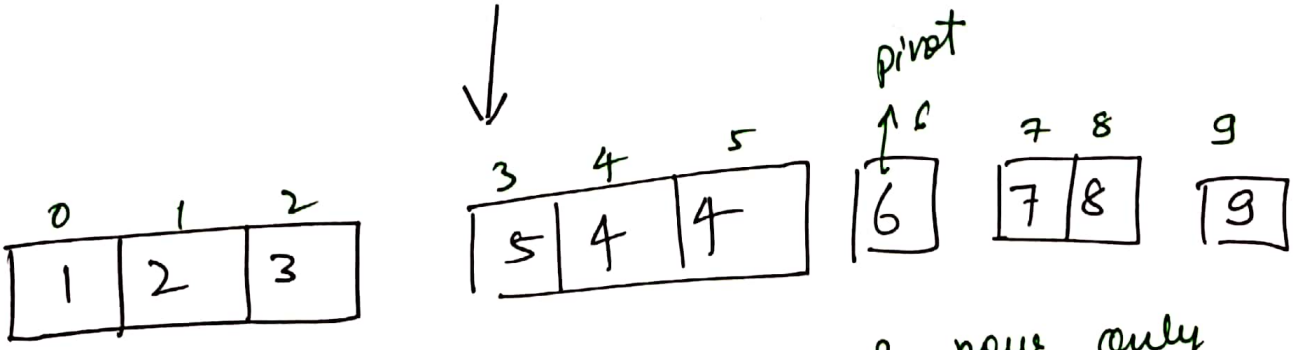
∴ swap them.





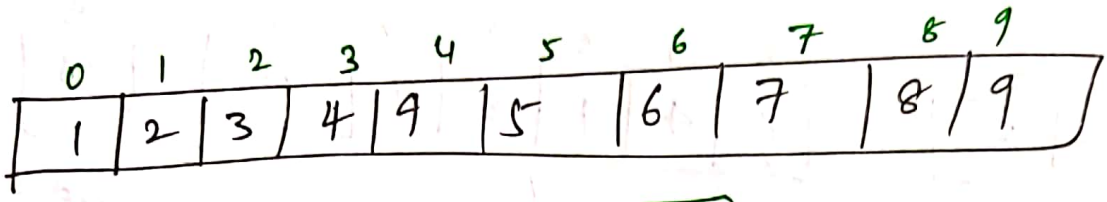
↓ again swap

(j<=i has not met, we just swap our elements & continue to search)



⇒ i & j crossed each other & now only swap our pivot elemet & elemt[j].

↓ Repeat the same for the rest



Sorted Array

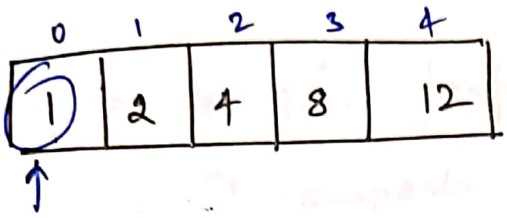
X X

Video - 57

Analysis of QuickSort Algorithm

* Worst Case :⇒(Already Sorted Elements)

eg:



⇒ Apply partiting algorithm

Take : pivot

(sub-array)

pivot partition

(apply partition concept)

(1) pivot 2 4 8 12

(1) 4 8 12 pivot

(1) 8 12 pivot

(1) 12

4 partition.

⇒ No. of partition = 4 (here) → size = 5

⇒ if n = size :

then no. of partition ⇒ (n-1)

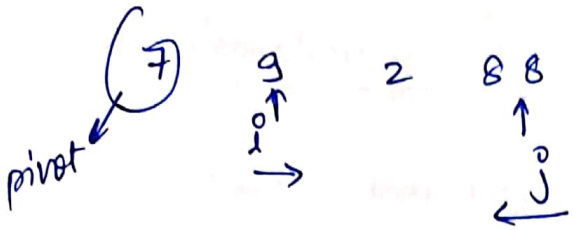
* to know QuickSort time complexity in this "worst case" ⇒ already sorted elements

⇓

QuickSort time Complexity

⇓

Time Complexity of partitioning ∝ n (size)



⇒ traverse i & j until they crosses each other

⇒ Partition performs comparison

⇒ it compares element to pivot
(i.e, 7 ko 9 se compare)

⇒ if the comparison isn't successful, it'll compare go on.
(7 with 2, then 88 and so on ...)

⇒ Partitioning is some linear f^n of 'n'.

(No. of Comparisons in Partition Algorithm) ⇒ some linear f^n of n
⇒ $k_1 n + k_2$

No. of partition happening = $(n-1)$ times
n = size of Array

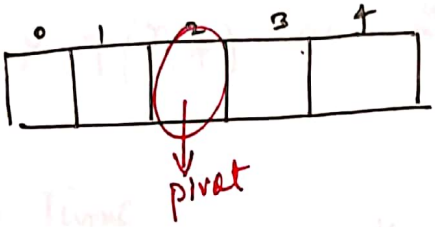
Total time for time Complexity = $(n-1) \cdot k_1 n + k_2$

$= O(n^2)$

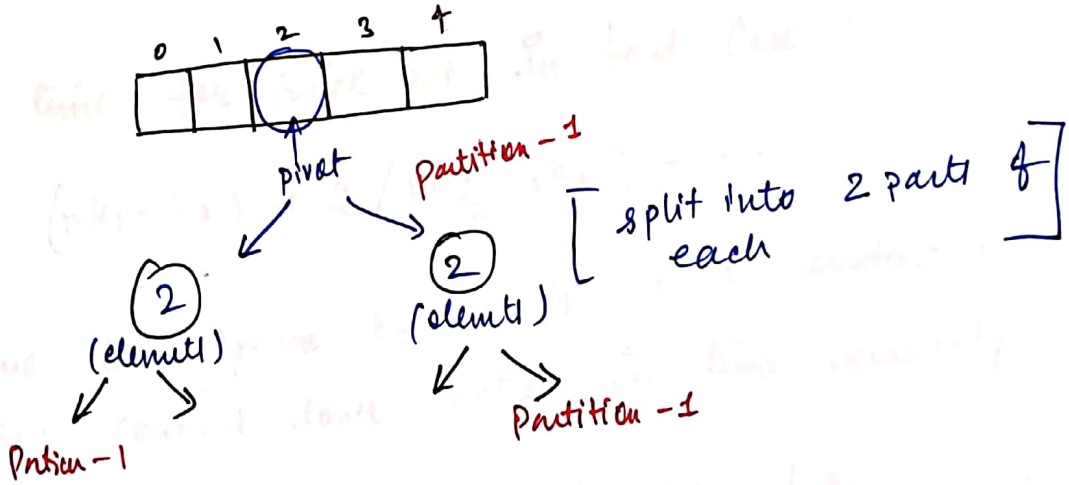
Time Complexity in Best Case Analysis !

Time complexity for Best Case for Quick Sort => $O(n \log n)$

=> In best case, we assume that the array split into 2. That means the pivot is here.

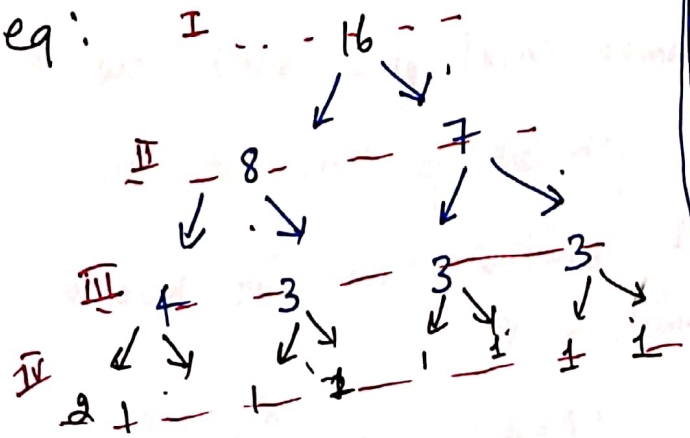


i.e., pivot comes in middle of our Array



* total 3 times partition

eg:



[note: => 1 element kaam lia hamne bzc. wo already pivot ban gaya and apne place par hi hai wo
↓
1 element automatically in its own place]

I : $1 \times T(n)$:⇒ Have to do 1 partitioning and size of partitioning ⇓ array of size (n)

no. of partitiing ↑ Size of partitiing ↑

II : $2 \times T(n/2)$

III : $4 \times T(n/4)$ or $2^2 \times T(n/2^2)$

Partitioning Time $T(n) = k_1 n + k_2$

Total time for QuickSort in best Case :

$$= (nk_1 + k_2) + 2\left(k_1 \frac{n}{2} + k_2\right) + \cdots$$

* we can ignore k_2 as it is a constant term Bzc constant don't matter in time complexity

∴ Total time $= (nk_1) + 2\left(k_1 \frac{n}{2}\right) + \left(2^2 \times k_1 \times \frac{n}{2^2}\right)$

* we have only $(k_1 n)$ remaining in each term at the end of the day

* and this will continue till 'h' (height of tree) ⇒ 'h' times

* in this case $(h=4)$ ⇒ 4 level of the tree

$\therefore$ Total Time $= (h \times k, n) + K$

h ↑ height of tree ↓ time complexity $\Rightarrow O(\log n)$

K ↑ constant

$\cong O(n \log n)$

Note:

$\log_2 16 = 4$

$\log_2 8 = 3$

* Average Time Complexity $= O(n \log n)$

* Note * → T(n)

① Best Case → $O(n \log n)$

② Avg. Case → $O(n \log n)$

③ Worst Case → $O(n^2)$

Stability

2 8 9 12 2

i → ← j

2 2 9 12 8 ⇒ 2 2 9 12 8

i → ← j

(Not Stable)

⇒ Quick sort is not stable.

⇒ it will change the order of ur elements

⇒ Quick sort is in-place algorithm bzc it's not taking up extra space.

⇒ It's not necessary that when you perform Quick sort, the pivot taken as 1st element. You can take last element as pivot too.

⇒ You can use randomized Quick sort too, in which pivot is a random element

Video- 58

Merge - Sort - Sorting - Algorithm

→ Algorithm in which we break an array into pieces, sort it, break it - sort it.

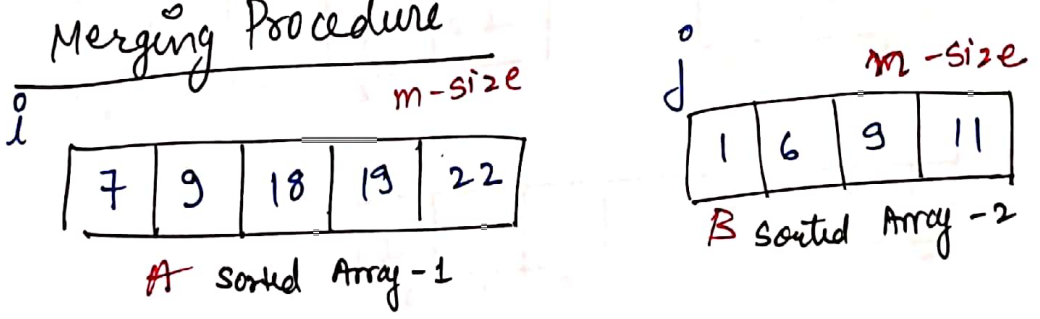
Eg: 7, 8, 22, 81 (array)

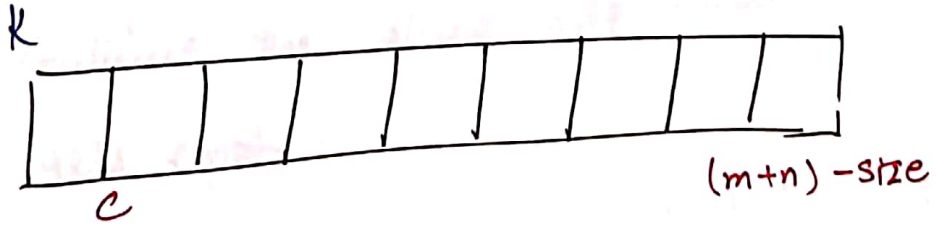
↓ break this array into pieces until every element is left in it & then when it's left with one element at a time, then we merge them again.

such that the merged array have both these elements sorted.

[initially i = j = K = 0]

Merging Procedure





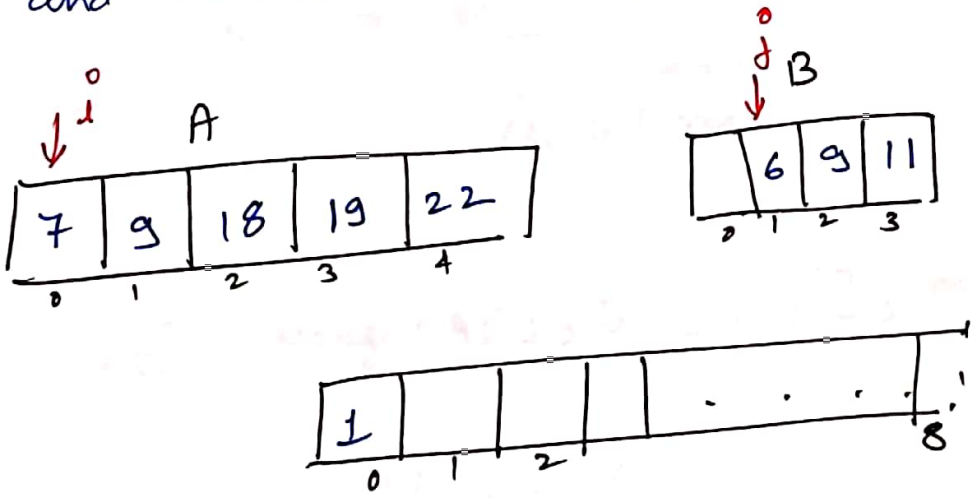
* we have to merge both the array into one array.

⇒ We compare the element at index i of 'A' and index j of 'B' & see which one is smaller. Fill the smaller element at index 'k' of array 'c' and increment k by 1. Also, Increment the index variable of the array we fetched the smaller element from.

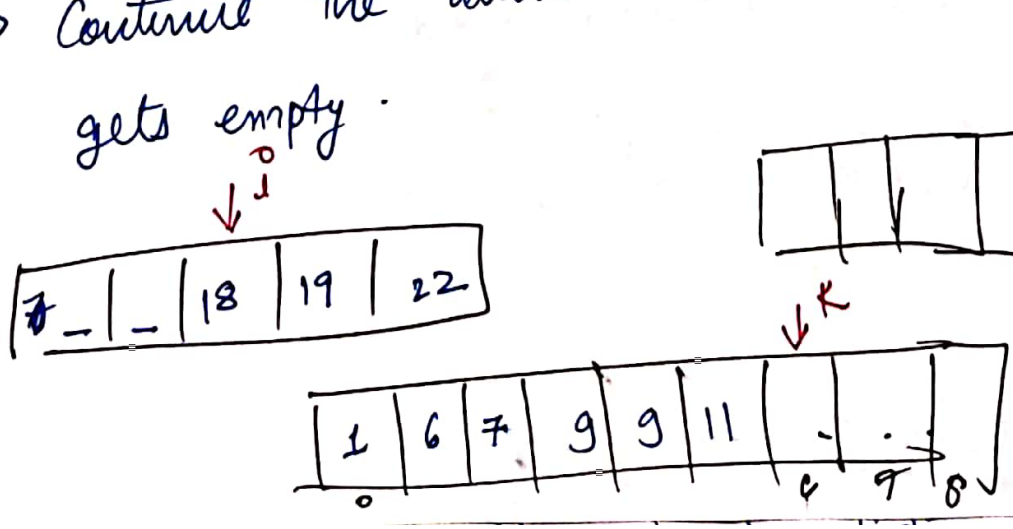
⇒ Here, $A[i] > B[j]$

∴ fill $C[k]$ with $B[j]$

and increase k & j by 1.

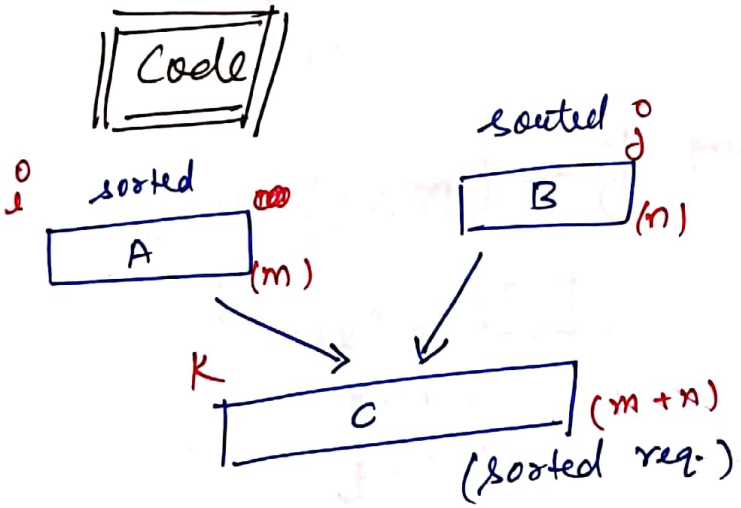


(#) ⇒ Continue the above step until A or B gets empty.



Here,

⇒ Array B inserted all its elements in the merged array C. Since we are now only left with the elements of element A, we'll simply put them in the merged array. This will result in our final merged array C.

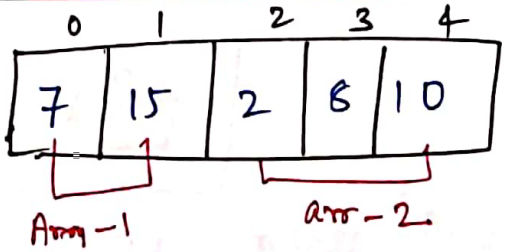



```
void Merge(A[], B[], C[], m, n)
{
    int i, j, K;
    i = j = K;
    while (i < m && i < n)
    {
        if (A[i] < B[j])
        {
            C[K] = A[i]; i++;
            K++;
        }
        else if (A[i] > B[j])
        {
            C[K] = B[j]; j++;
            K++;
        }
    }
```

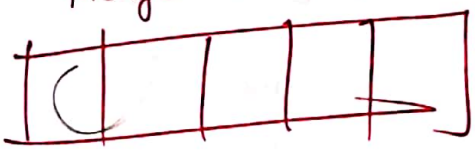
// when 1 Array fully gets copied and other array is left with data to be copied directly into the Array - C.

```
while (i < m)        → if Array A elemet left
{
    c[k] = A[i];
    k++;
    i++;
}
while (j < m)        → if Array B elemts left
{
    c[k] = B[j];
    k++;
    j++;
}
```

Part - 2 :- Merge - Sort

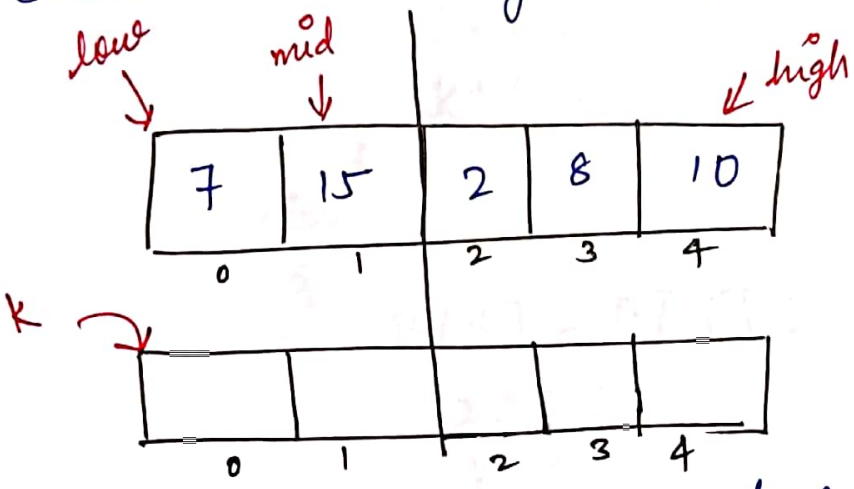


merge into diff arr



* Array of 5 elements and contains 2 sorted subarray of length 2 & 3 in itself.

→ Merge both the sorted subarray, create a auxillary array of same length.



- 1st index of 1st subarray = low
- last index of 2nd subarray = high
- separate them by line
- mark the index prior to the 1st index of 2nd subarray = mid

Coding

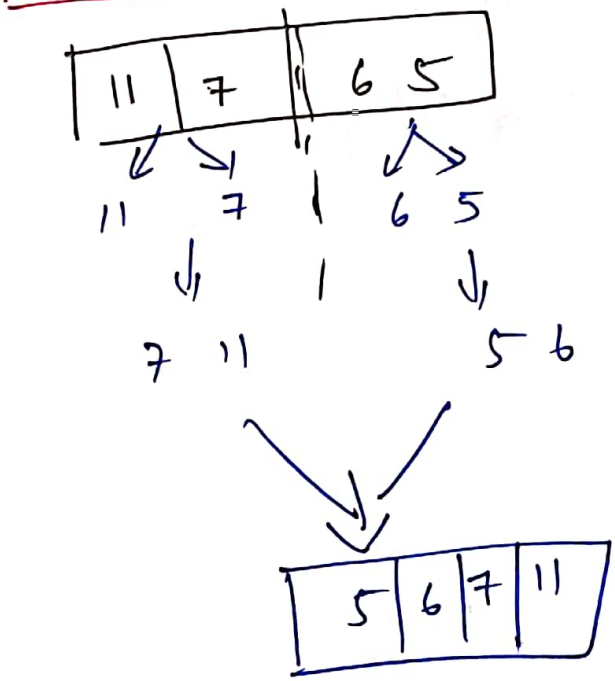
```
void merge (A[], mid, low, high
{
    int i, j, K;
    i = low;
    j = mid + 1;
    K = low;
```

```
while (i < mid && j < high)
{
    if (A[i] < A[j])
    {
        B[k] = A[i];
        i++;
        k++;
    }
    else
    {
        B[k] = A[j];
        j++;
        k++;
    }
}
if i array gets exhausted
while (i ≤ mid)
{
    B[k] = A[i];
    k++; i++;
}
while (j ≤ high)
{
    B[k] = A[j];
    k++; j++;
}
```

To copy all the elements of B back to A.

```
for(i = low ; i ≤ high ; i++)
  {
      A[i] = B[j] ;
```

Merge Sort Algo



Coding

```
void merge(A[], mid, low, high)
  {
      int i, j, intB[high+1] ;
      :
      :   similar to prev. code.
      :
  }
```

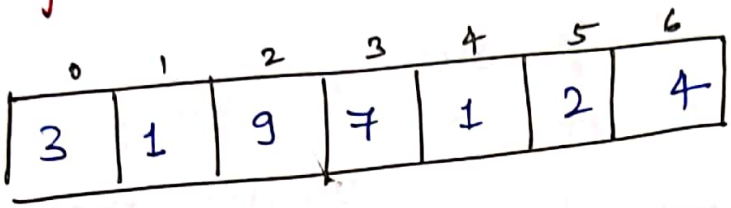
Recursive Merge Sort

```
void MS (int A[], int low, int high)
{
    if (low < high)
    {
        int mid = (l+h)/2;
        MS(A, l, mid);
        MS(A, mid+1, h);
        Merge(A, mid, low, high);
    }
}
```

Video - 60

Count - Sort Algorithm → one of the fastest algo to sort array

Array.



=> Algo says that the array which u have find out the max. element from it

=> max = 9 (above array)

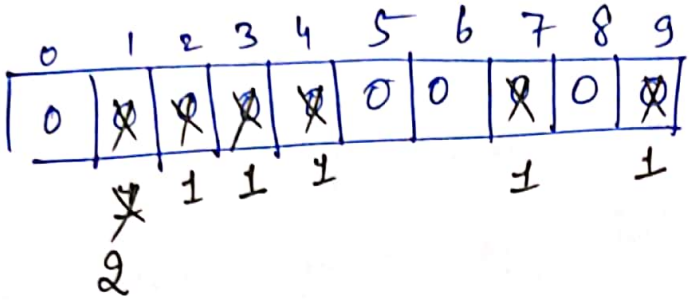
=> Make a count array of size = 9

Procedure:

1) find largest element from all the elements in the array & store it in some integer max (variable)

2) Create a 'count' array of size = (max + 1) size
   ⇓
   or auxillary array.
   => Initialise all count elements = '0'.

3) After initialising the count array, traverse through the given array, and increment the value of that element in the count by '1'.

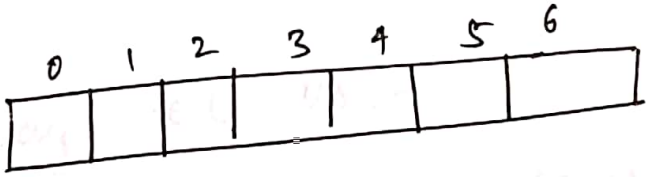


(4) By defining the size of the count array as the max. element in the array, you ensure that each element in the array has its own corresponding index in the count array. After we traverse through the whole array, we'll have the count of each element in the array.

(5) Now, traverse through the count array, and look for the non-zero elements. The moment u find an index with some value other than zero, fill in the sorted array the index of the non-zero element until it becomes zero by decrementing it by 1 everytime you fill.

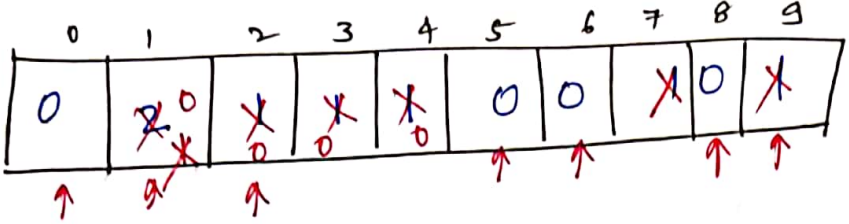
Count array of yours which came, that you will again copy into a new array.

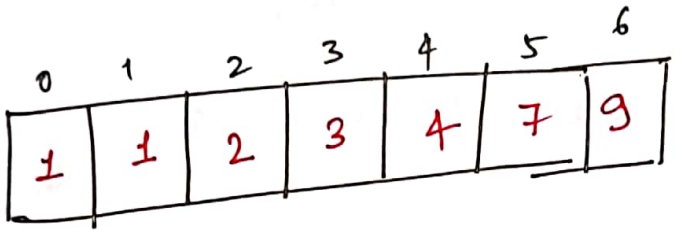
Now, traverse this array



→ when value = 0, move ahead, when I'll move forward I'll see 2 here, and as I can see 2 here, what I will do is decrement 2 and keep index of 2 here.

→ Is this a zero? No, again decrement it to make it zero(0). when it becomes zero(0) then only move ahead.
Just decrement it and copy its index.





⇒ new array (Sorted array)

⇒ Count Array tells us :-

⇓

tells us which elements comes how many times in the main array.

⇓

we use this information to sort our array

Coding

```
int maximum (int A[], int n)
{
    int max = INT_MIN ;
    for (i=0 ; i<n ; i++)
    {
        if (max < A[i])
        {
            max = A[i];
        }
    }
} return max ;
```

```
void countsort (int A[], int n)
{
    int i, j;
    int max = maximum (A, n)
    int count [] = new int[max+1];
```

to initialize all element =0 ← `for (i=0; i < max+1; i++)`

```
    {
        count [i] = 0;
    }
```

increment the corresponding index ← `for (i=0; i<n; i++)`

```
    {
        count [A[i]] = count [A[i]] + 1;
    }
    i = 0; → counts count array
    j = 0; → counts for given array
    while (i <= max)
    {
        if (count [i] > 0)
        {
            A[j] = i;
            count [i] = count [i] - 1;
            j++;
        }
        else
        {
            i++;
        }
    }
}
```

Analysis

⇒ Takes extra space

⇒ T(n)

Let; Size of array = n

max. elemt in array = m

$\therefore T_n = O(m+n)$

$\approx O(n)$

when n >>> m much larger

⇓ linear time

But at the cost of ext

Bubble Sort

```
void bubble(int A[])
{
    int temp;
    int n = A.length;
    //no of passes - for
    for(i=0; i<(n-1); i++)
    {
        //for comparison in each pass
        for(j=0; j<(n-1-i); j++)
        {
            if(A[j] > A[j+1])
            {
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }//j for
    }//i for
}
```

If
*(n)= length
then
no. of passes = (n-1)

1st pass → (n-1) comparis
2nd pass → (n-2) comparis
3rd pass → (n-3) co

Insertion Sort

n = length
passes = (n-1)
Start for index = 1 (not 0)

```
public void insertSort (int A[], int n)
{
    int n = A.length;
    //loop for passes
    for (i = 1 ; i <= (n-1) ; i++)
    {
        //loop for each pass
        int key = A[i];
        int j = i - 1;
        key se 1 value just less
        while (A[j] > key && j >= 0)
        {
            A[j+1] = A[j];
            j--;
        }
        A[j+1] = key;
    }
}//fn.
```

Selection Sort

* n = length
* no of passes = (n-1)

```
public void ssort(int A[]).
{
int n = A.length ; int temp ;
                   int min ;
// no. of passes
for (i= 0 ; i <(n-1) ;i++)
{
    min = i ;
    // for comparison
    for (j = i+1 ; j <n ; j++)
    {
        if(A[j] < A[min])
        {
            min = j ;
        }
    }
    temp = A[i];
    A[i] = A[min]:
    A[min] = temp :
}
}//fn
```

Quick - Sort

range : low - high

```
public void quickSort(int A[], int low, int high)
{
    int partition_index;   (p=i)   -> index of pivot after partition

    if (low < high)
    {
        pi = partition(A, low, high);
        quickSort(A, low, pi-1);      <- sort left subarray
        quickSort(A, pi+1, high);     <- sort right subarray
    }
}

public int partition(int A[], int low, int high)
{
    int pivot = A[low];      <- taking 1st element as pivot
    int i = low+1;
    int j = high;
    int temp;
    do{
        while(A[i] <= pivot)
        {
            i++;
        }
        while(A[j] > pivot)
        {
            j--;
        }
```

```
if (i < j)
{
    temp = A[i];
    A[i] = A[j];
    A[j] = temp;
}
while (i < j);
temp = A[low];
A[low] = A[j];
A[j] = temp;
return j;
}//fn
```

A(low) = pivot;