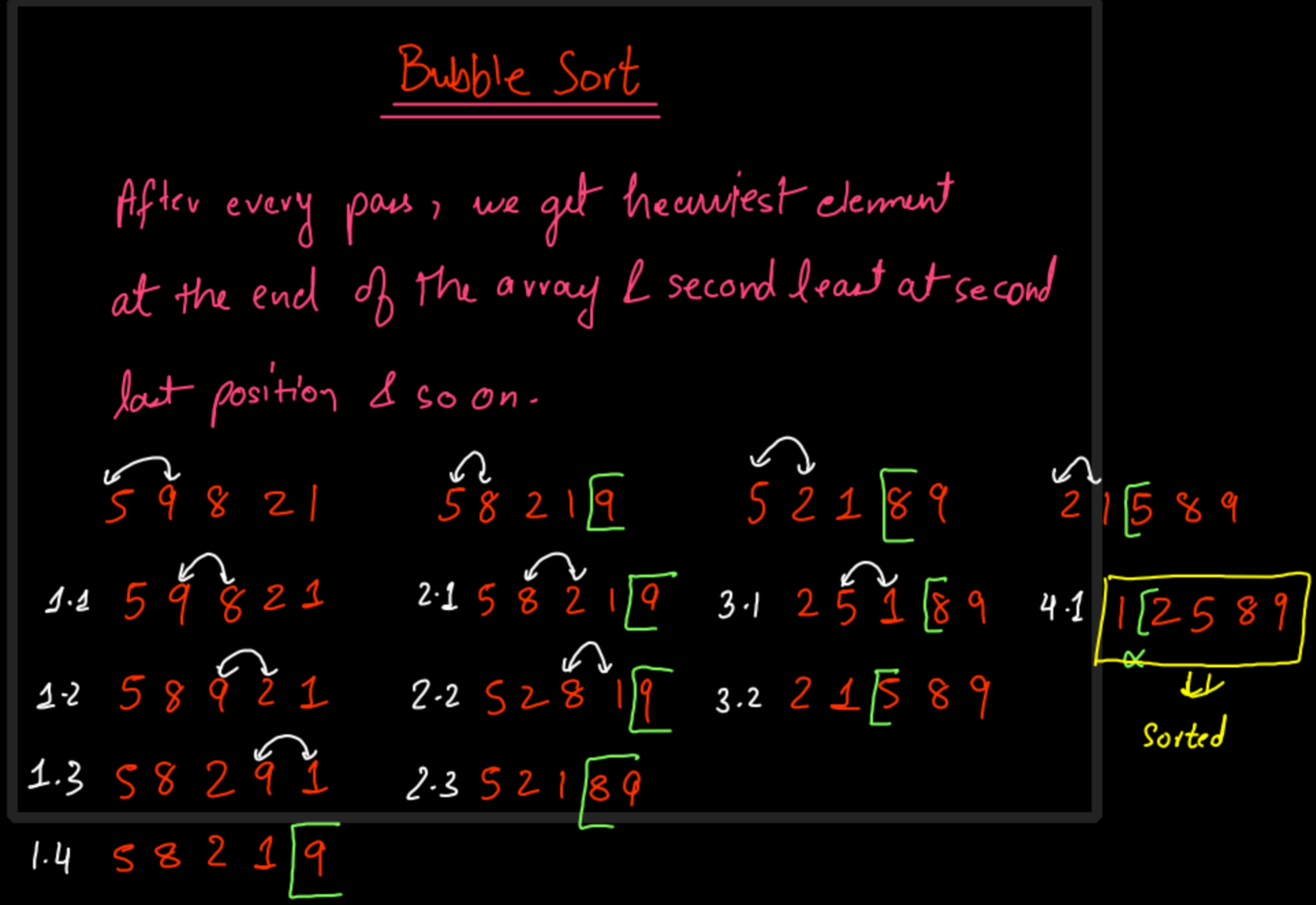
Bubble Sort

After every pass, we get the heaviest element at the end of the array & second least at second last position & so on.



5 9 8 2 1

1.1 5 9 8 2 1

1.2 5 8 9 2 1

1.3 5 8 2 9 1

1.4 5 8 2 1 9

4 comparisons

(n-1) comp

5 8 2 1 9

2.1 5 8 2 1 9

2.2 5 2 8 1 9

2.3 5 2 1 8 9

3 comparisons

(n-2)

5 2 1 8 9

3.1 2 5 1 8 9

3.2 2 1 5 8 9

2 comparison

(n-3)

2 1 5 8 9

4.1 1 2 5 8 9

Sorted

1 comp

(n-4)

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

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

$TC = O(N^2)$

```java
public static void bubbleSort(int[] arr) {
  //write your code here

  for(int i=0;i<arr.length - 1;i++) {
      for(int j=0;j<arr.length-i-1;j++) {
          if(isSmaller(arr,j+1,j)) {
              swap(arr,j+1,j);
          }
      }
  }
}
```

```java
public static void bubbleSort(int[] arr) {
  //write your code here
  boolean flag = false;
  for(int i=0;i<arr.length - 1;i++) {
      for(int j=0;j<arr.length-i-1;j++) {
          if(isSmaller(arr,j+1,j)) {
              swap(arr,j+1,j);
              flag = true;
          }
      }

      if(flag == false) {
          //Array is already sorted hence no swap occured
          break;
      }
  }
}
```

The main summary of Bubble Sort is that the heaviest element (largest) will be at its sorted position after the first iteration.

So, the largest elements keep on moving at the last of the array.

Best Case : Array Already Sorted

Best Case Time : O(N)
if this code is used else $O(N^2)$ only.

Selection Sort

At i^{th} iteration, do linear search & find i^{th} smallest element to put it at i^{th} index.

Selection Sort Code

```java
public static void selectionSort(int[] arr) {
  //write your code here
  for(int i=0;i<arr.length-1;i++) {
      int minIndex = i;
      for(int j = i + 1;j<arr.length;j++) {
          if(isSmaller(arr,j,minIndex)) {
              minIndex = j;
          }
      }
      swap(arr,i,minIndex);
  }
}
```

If array is already sorted, then also $TC = O(N^2)$
i.e. the best case will always be $O(N^2)$ (No improvisation can be done)

Bubble Sort is worst than selection sort because of high no of swaps.

Data Structure Operations Cheat Sheet

Data Structure Name	Average Case Time Complexity				Worst Case Time Complexity				Space Complexity
	Accessing n^{th} element	Search	Insertion	Deletion	Accessing n^{th} element	Search	Insertion	Deletion	Worst Case
Arrays	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Stacks	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Queues	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Binary Trees	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Binary Search Trees	$O(logn)$	$O(logn)$	$O(logn)$	$O(logn)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Balanced Binary Search Trees	$O(logn)$	$O(logn)$	$O(logn)$	$O(logn)$	$O(logn)$	$O(logn)$	$O(logn)$	$O(logn)$	$O(logn)$
Hash Tables	N/A	$O(1)$	$O(1)$	$O(1)$	N/A	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Note: For best case operations, the time complexities are $O(1)$.

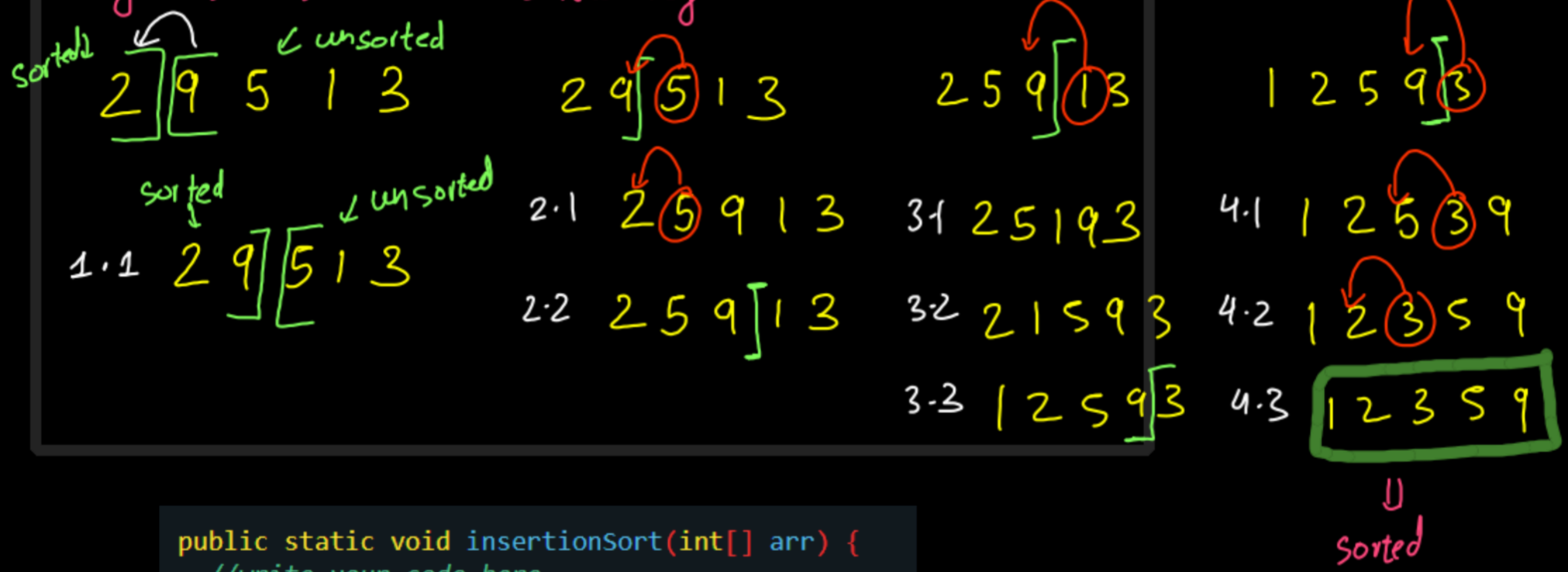
Sorting Algorithms Cheat Sheet

Sorting Algorithm Name	Time Complexity			Space Complexity	Is Stable?	Sorting Class Type	Remarks
	Best Case	Average Case	Worst Case	Worst Case			
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Comparison	Not a preferred sorting algorithm.
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Comparison	In the best case (already sorted), every insert requires constant time
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Comparison	Even a perfectly sorted array requires scanning the entire array
Merge Sort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(n)$	Yes	Comparison	On arrays, it requires $O(n)$ space; and on linked lists, it requires constant space
Heap Sort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(1)$	No	Comparison	By using input array as storage for the heap, it is possible to achieve constant space
Quick Sort	$O(nlogn)$	$O(nlogn)$	$O(n^2)$	$O(logn)$	No	Comparison	Randomly picking a pivot value can help avoid worst case scenarios such as a perfectly sorted array.
Tree Sort	$O(nlogn)$	$O(nlogn)$	$O(n^2)$	$O(n)$	Yes	Comparison	Performing inorder traversal on the balanced binary search tree.
Counting Sort	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Yes	Linear	Where k is the range of the non-negative key values.
Bucket Sort	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n)$	Yes	Linear	Bucket sort is stable, if the underlying sorting algorithm is stable.
Radix Sort	$O(dn)$	$O(dn)$	$O(dn)$	$O(d + n)$	Yes	Linear	Radix sort is stable, if the underlying sorting algorithm is stable.

In place Algorithms

Insertion Sort

Consider the given array into 2 parts one is sorted & other unsorted. One by one pick an element from unsorted region & insert it in sorted region.




```java
public static void insertionSort(int[] arr) {
  //write your code here
  for(int i=1;i<arr.length;i++) {
      for(int j=i;j>0;j--) {
          if(isGreater(arr,j-1,j)) {
              swap(arr,j-1,j);
          } else {
              break;
          }
      }
  }
}
```

$O(n^2)$

$BC = O(n)$

check yaha lagaenge.

Merge 2 Sorted Array (Two pointer technique)

(M) A: 2 3 6 7 9 10

(N) B: 1 4 5 7 10 12 13 14

(M+N) C =

0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	2	3	4	5	6	7	7	9	10	10	12	13	14

Compare A[i] & B[j] and add the smaller element into C.

After one array is completed, copy all the elements of the second array as it is.

```java
public static int[] mergeTwoSortedArrays(int[] a, int[] b){
  //write your code here
    int[] c = new int[a.length + b.length];

    int i = 0, j = 0, k = 0;

    while(i < a.length && j < b.length) {

        if(a[i] <= b[j]) {
            c[k] = a[i];
            i++;
            k++;
        } else {
            c[k] = b[j];
            j++;
            k++;
        }
    }

    while(j<b.length) {
        c[k] = b[j];
        j++;
        k++;
    }

    while(i<a.length) {
        c[k] = a[i];
        i++;
        k++;
    }
    return c;
```

A.length = n

B.length = m

TC = O(n+m)

Merge Sort (Divide & Conquer)

0 1 2 3 4 5 6 7 8 9 10

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

faith that both get sorted using MS

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

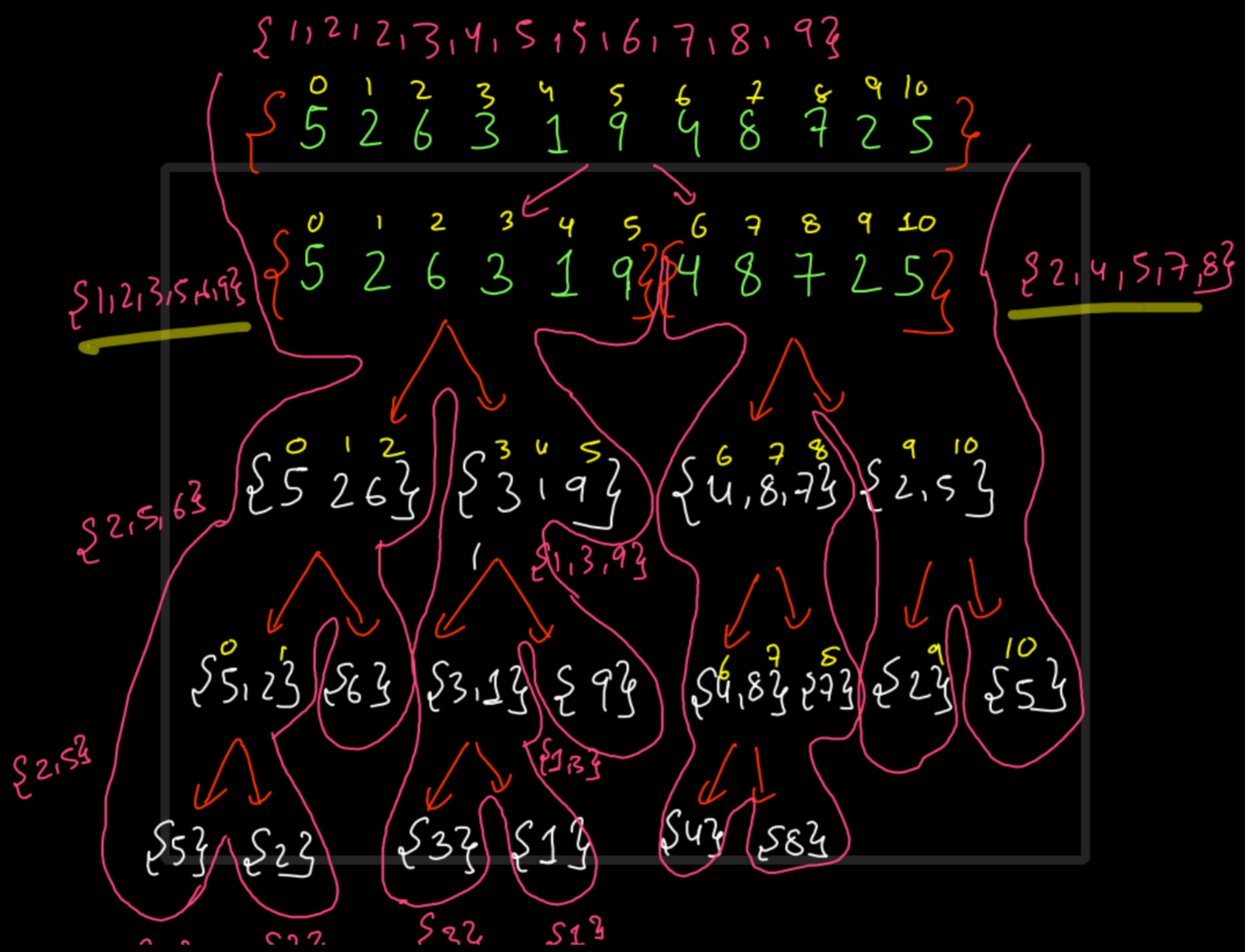
Merge them

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

Divide & Conquer: Divide big problem into 2 almost equally sized sub problems. (Recursion faith)

Now Conquer (Merge in this case)

Not Mandatory that conquer will be after divide only.


{1,2,2,3,4,5,5,6,7,8,9}
0 1 2 3 4 5 6 7 8 9 10
5 2 6 3 1 9 4 8 7 2 5
{1,2,3,5,4,9}
{2,4,5,7,8}
{2,5,6}
{1,3,9}
{2,5,4}
{1,3}

```java
public static int[] mergeSort(int[] arr, int lo, int hi) {
  //write your code here

  if(lo == hi) {
      int[] baseArr = new int[1];
      baseArr[0] = arr[lo];
      return baseArr;
  }

  int mid = (lo + hi)/2;

  int[] leftSortedArray = mergeSort(arr,lo,mid);
  int[] rightSortedArray = mergeSort(arr,mid+1,hi);

  int[] sortedArray = mergeTwoSortedArrays(leftSortedArray,rightSortedArray);

  return sortedArray;
}
```

Calls = 2

Height = $\log_2 N$

$(calls)^{h} + (pre+post)*h = (2)^{\log_2 n} + (0+N)*\log_2 N$ ↳ merge

$= n + n\log_2 N$

$O(\log_2 N)$

$$T(n) = 2T(n/2) + N$$

$$2\,T(n/2) = (2T(n/4) + N/2)\,2$$

$$2^2\,T(n/4) = (2T(n/8) + N/4)\,2^2$$

$$\vdots$$

$$2^{k-1}(T(1)) = (2T(0) + 1)\,2^{k-1}$$

$$T(n) = 2^k + N + \frac{N}{2} \times 2 + \frac{N}{2^2} \times 2^2 \cdots k \text{ times}$$

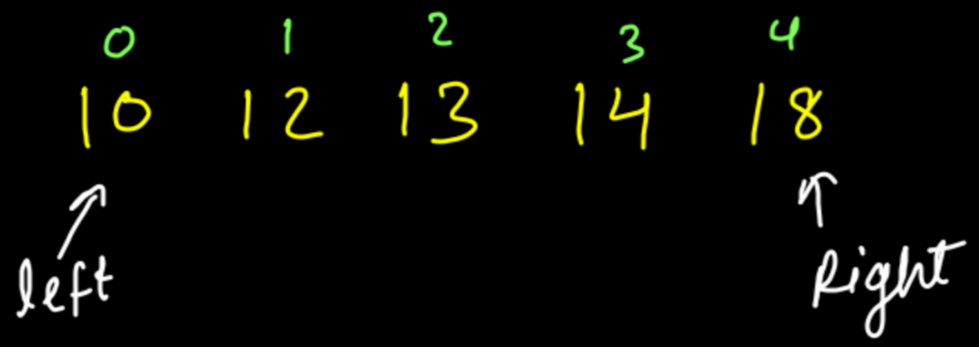
$$T(n) = 2^{\log_2 N} + N\log_2 N \;(k = \log_2 N) = O(N\log_2 N)$$

① Extra/Auxi Space: $O(N)$ {at merging}

② Input/output : $O(N)$

③ Recursion (all Stack Space: $O(\log_2 N)$
(Height of Stack/Tree)

Target Sum Pair




```
//brute force
for(int i=0;i<nums.length;i++) {
    for(int j=i+1;j<nums.length;j++) {
        if(nums[i] + nums[j] == target) {
            return new int[]{i=1,j+1};
        }
    }
}

return null;
```

2 pointer solution

```
int lo = 0;
int hi = nums.length-1;

while(lo < hi) {
    if(nums[lo] + nums[hi] < target) {
        lo++;
    } else if(nums[lo] + nums[hi] > target) {
        hi--;
    } else {
        return new int[]{lo+1,hi+1};
    }
}

return null;
```

nums[left] + nums[right] > target

right--;

bada kuch zyada he
bada reh gaya

nums[left] + num[right] < target

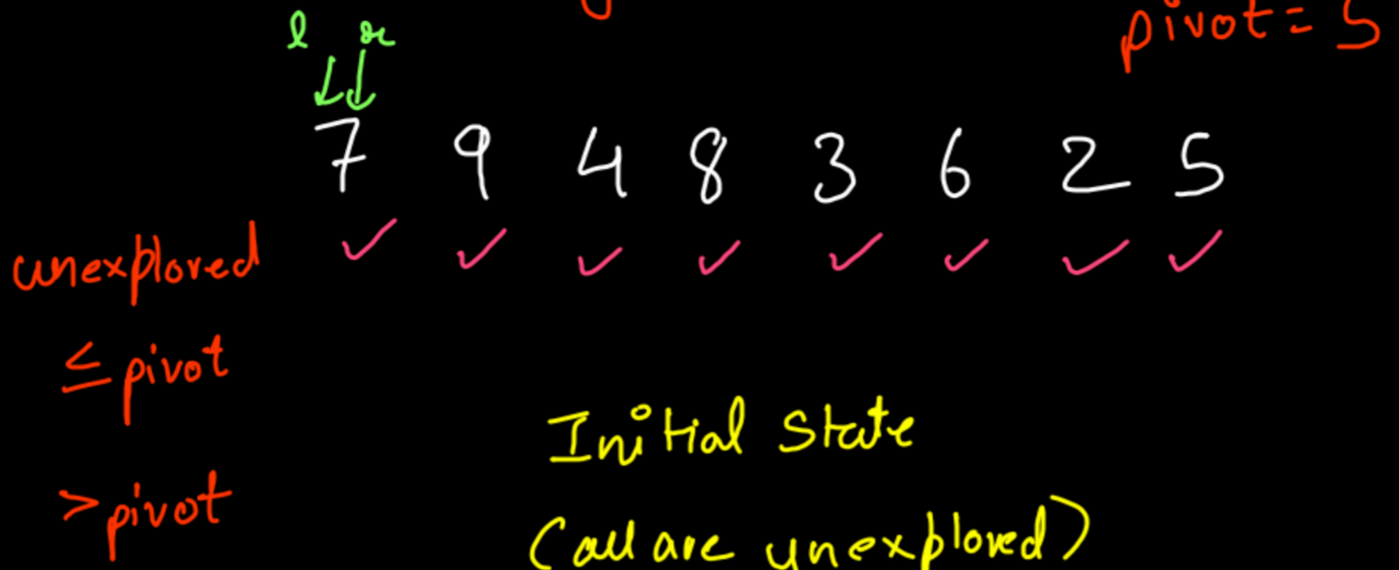
left++;

chota kuch < yada he
chota reh gaya

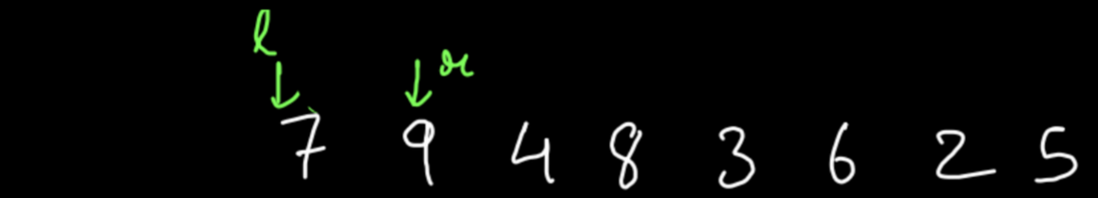
$T(n) = T(n-1) + O(1)$

$TC = O(N)$

Partition an Array



Initial State
(all are unexplored)

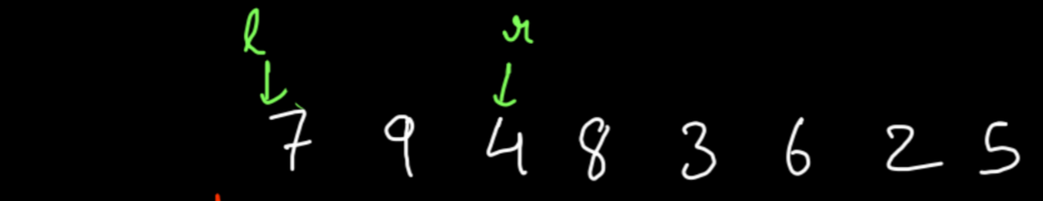


unexplored
(r to n)

≤ pivot
(0 to l-1)

> pivot ✓
(l to r-1)

```
if (arr[right] > pivot) rt++;
```

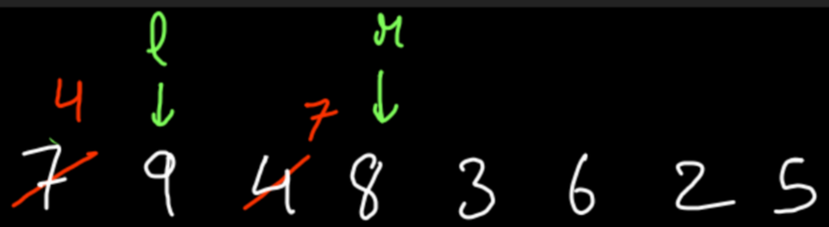


unexplored
(r to n)

≤ pivot
(0 to l-1)

> pivot
(l to r-1)

```
if (arr[right] ≤ pivot) {
    swap(arr[l], arr[r]);
    l++; r++;
}
```



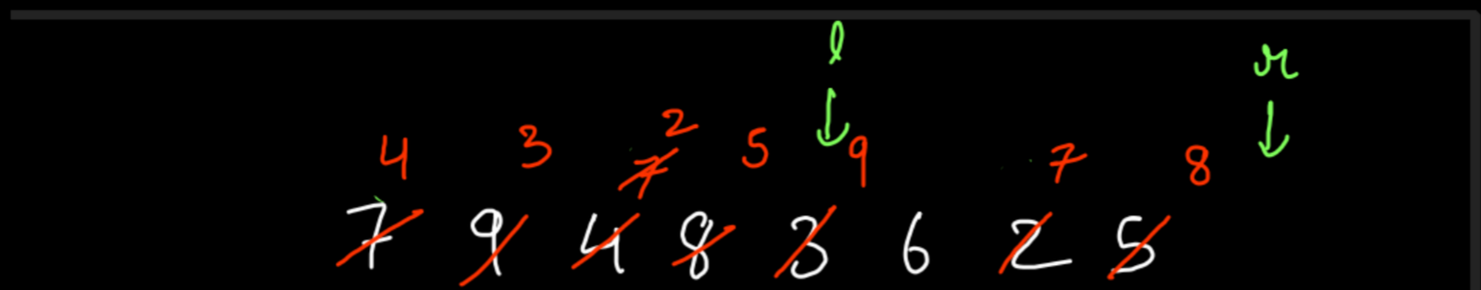
unexplored
(r to n)

≤ pivot

(0 to l-1)

> pivot

(l to r-1)



unexplored
(r to n)

≤ pivot
(0 to l-1)

> pivot
(l to r-1)

left = first ele of greater region

right = first ele of unexplored region

(Relative order changed)

Partitioned Array = 4 3 2 5 9 6 7 8

```java
public static void partition(int[] arr, int pivot){
  //write your code here
  int left = 0;
  int right = 0;

  while(right < arr.length) {
      if(arr[right] > pivot) {
          right++;
      } else {
          swap(arr,right,left);
          left++;
          right++;
      }
  }
}
```

Time Complexity = O(N)

Sort 01

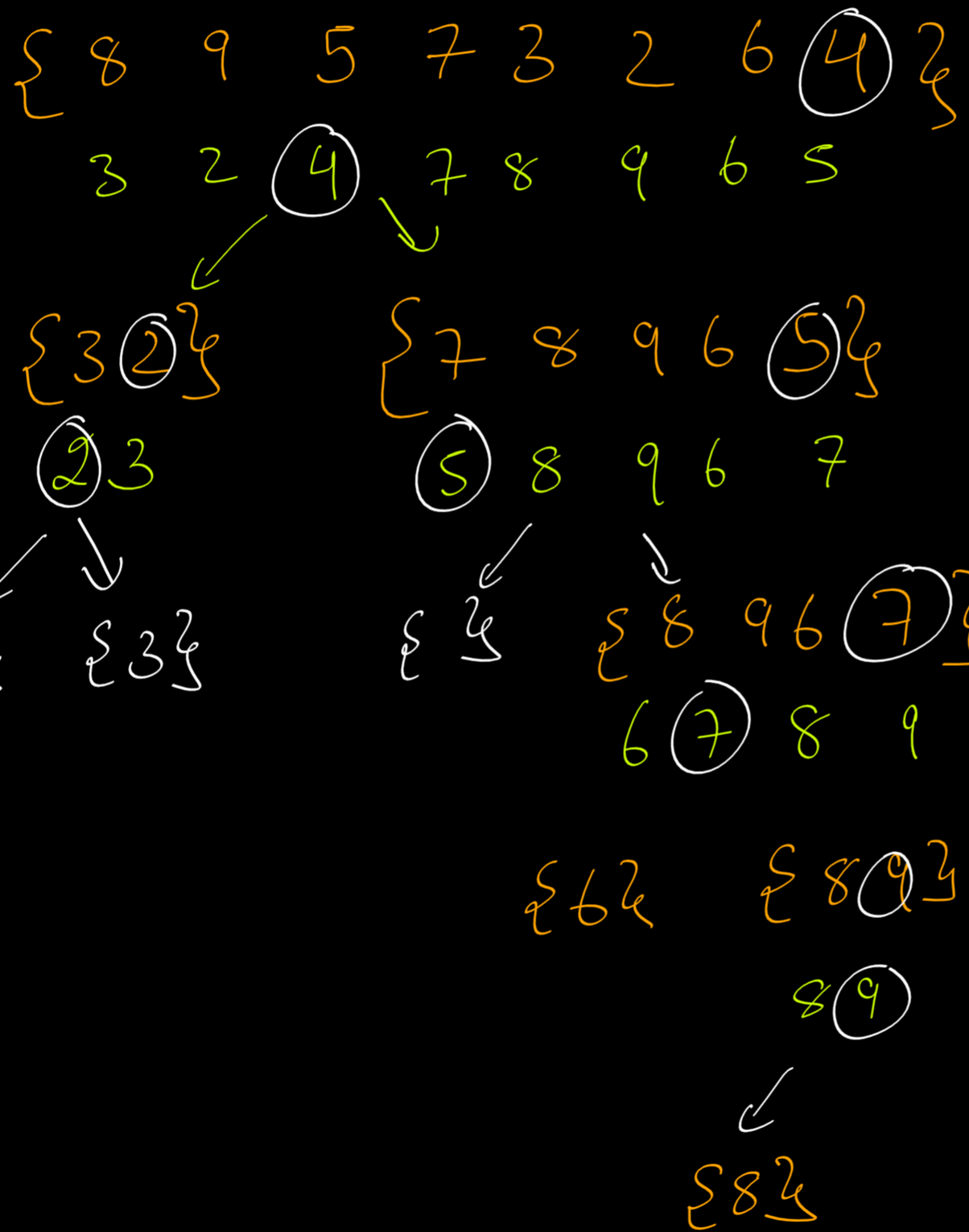
Same as the partition of array. Just make pivot = 1 or 0 and partition accordingly.

```java
public static void sort01(int[] arr){
  //write your code here
  int left = 0;
  int right = 0;

  for(int i=0;i<arr.length;i++) {
    if(arr[i] > 0) {
      right++;
    } else {
      int temp = arr[left];
      swap(arr,right,left);
      left++;right++;
    }
  }

}
```

Quick Sort



Partitions In Preorder

Subarray [l-r]

pivot = arr[right]

⇓

why?

↳ Partitioning on pivot will result in pivot element to be on its' current index (its' sorted position)

QuickSort Recursive Code

```java
public static void quickSort(int[] arr, int lo, int hi) {
  //write your code here
  if(lo > hi) {
      //no element in the array
      return;
  }

  int partitionIndex = partition(arr,arr[hi],lo,hi);

  quickSort(arr,lo,partitionIndex-1);
  quickSort(arr,partitionIndex + 1, hi);
}
```

Decrease and Conquer approach

Space

Input/output → O(N)

Extra space → O(1)

Recursion call stack → worst case O(n) (depth of Recursion tree)

Avg case O(h) = O(log n)

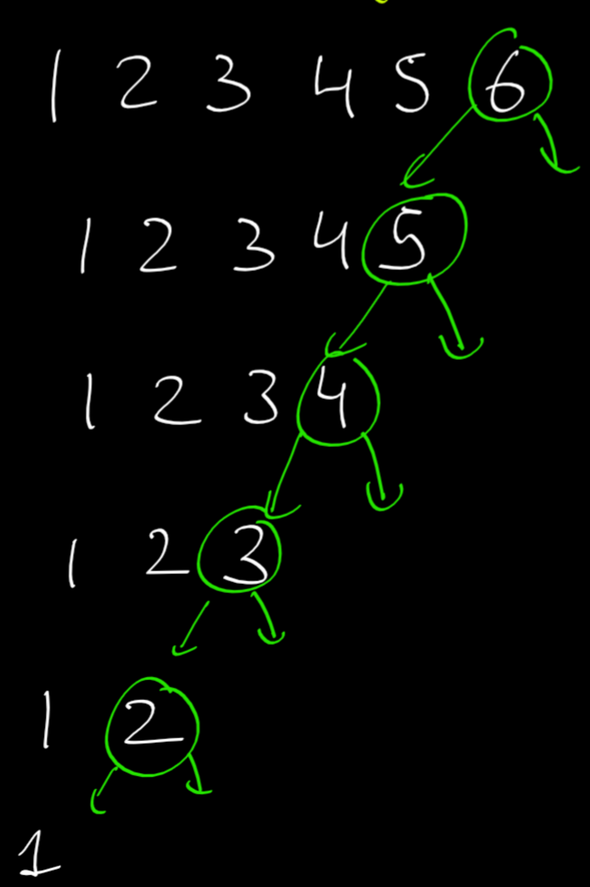
$T(n) = 2T(?) + O(n)$ → Time for partition

wc $T(n) = T(n-1) + O(n)$

↳ because partition is at one end of the array.

Avg and best case $T(n) = 2T(n/2) + O(n)$ ⇒ same recurrence relation as that of Merge Sort. So, in avg & best case, Quick Sort takes $O(n \log n)$ time.

Worst Case: (Array is already sorted or reverse sorted)



Solving RR for worst case:

$T(n) = T(n-1) + O(n)$

$T(n-1) = T(n-2) + O(n)$

$T(n-2) = T(n-3) + n$

⋮

$T(1) = T(0) + n$

} n times

$T(n) = T(0) + n^2$

↳ only 1 ele $= O(1)$

$\boxed{T(n) = O(n^2)}$

Randomized QuickSort

Even if the array is nearly sorted, QuickSort can hit worst case.

2 1 3 5 (4) 6 8 (7) ⇒ Nearly / Almost Sorted Array

Say, the Random Element is 4.

DO NOT partition about 7.

Swap 7 with a random element so that chances of partition at middle increase.

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

↳ Again, do not select 6 for partitioning. Select any random element & swap it with 6.

** Randomized QuickSort does not decrease the Worst Case time (i.e $O(n^2)$)

It just reduces the probability of hitting the worst case.

(Now the WC is not a sorted array. Infact it might be some case which works fine in Normal QS)

MergeSort vs QuickSort

MergeSort

Worst case :
Best case : } O(nlogn)
Avg case :

Extra Space : O(n) ⇒ Not in place

Stable ✓

Prefered for Non Sequential Data (Linked List)

QuickSort

Worst Case : $O(n^2)$

Best Case : O(nlogn)

Avg Case : O(nlogn)

Extra Space : O(n) ⇒ In place sorting

Stable ✗

Prefered for Arrays

Stability (Same order of output as input)

$4^a\ 3^b\ 5^c\ 1^d\ 4^e\ 5^f\ 1^g$

⇒ If after sorting

1^d comes before 1^g

and 4^a comes before 4^e

then algo will be stable

$(4^a\ 3^b)(5^c\ 1^d)(4^e\ 5^f)(1^g\ 5^h)$

$(3^b\ 4^a)(1^d\ 5^c)(4^e\ 5^f)(1^g\ 5^h)$

$(1^d\ 3^b\ 4^a\ 5^c)\ (1^g\ 4^e\ 5^f\ 5^h)$

$(1^d\ 1^g\ 3^b\ 4^a\ 4^e\ 5^c\ 5^f\ 5^h)$

Why MergeSort is Stable?

```
public static int[] mergeTwoSortedArrays(int[] a, int[] b){
  //write your code here
   int[] c = new int[a.length + b.length];

   int i = 0, j = 0, k = 0;

   while(i < a.length && j < b.length) {

      if(a[i] <= b[j]) {
         c[k] = a[i];
         i++;
         k++;
      } else {
         c[k] = b[j];
         j++;
         k++;
      }
   }

   while(j<b.length) {
      c[k] = b[j];
      j++;
      k++;
   }

   while(i<a.length) {
      c[k] = a[i];
      i++;
      k++;
   }
   return c;
```

When both are equal, we add ele of 1st array.

So, merge sort is stable because merging is stable & quicksort is unstable because partition procedure is unstable

Quick Select (K^th Smallest)

- Binary Search (Divide & Conquer)
- Partition

{8 4 5 7 1 2 6 4 3} 6^th Smallest Element