If you have any questions or suggestions regarding the notes, please feel free to reach out to me on WhatsApp: **Sanjay Yadav Phone: 8310206130**

Notes -> DSA: Hashing: 1 - 20 - May 2023

- Need of HashMap
- Syntax Of HashMap
- Count Frequency
- Non Repeating Element
- HashSet
- distinct element

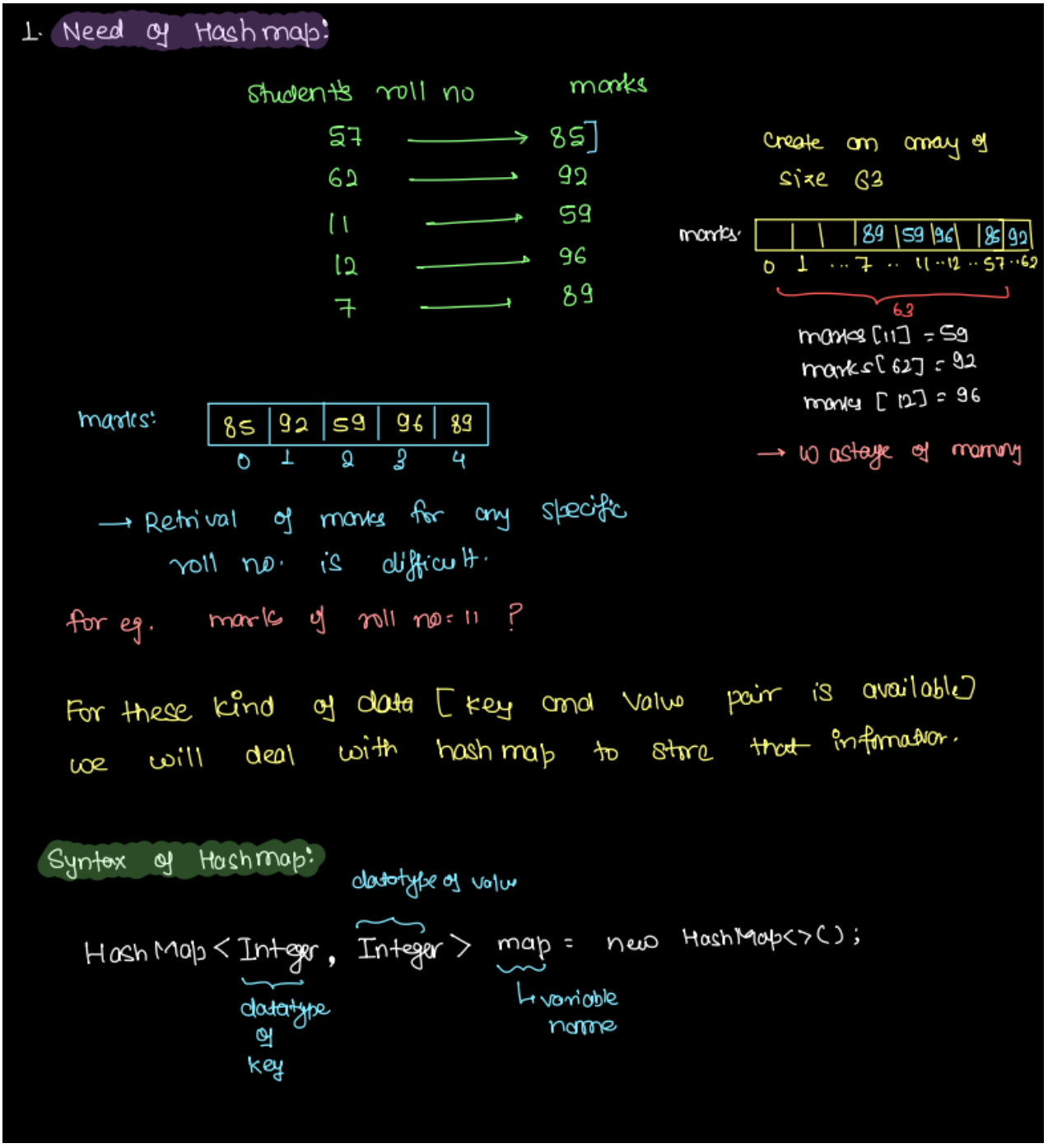
Note - Whenever we solve problems using prefix sum with HashMap or HashSet, we should include 0 as the first value. If using a HashSet, add 0 as a key; if using a HashMap, add a key-value pair of (0, -1). This approach covers the edge case where the prefix sum is 0. By following this method, we ensure that if the prefix sum becomes 0 at any point, it will be present in the map; otherwise, we may not get the desired result.

2. To find the longest length subarray with a sum of 0, we'll employ a HashSet and prefix sum. In this approach, the length of the subarray is typically calculated as right - left + 1. However, if two prefix sums in the range are equal, it means the sum of elements between those positions is 0, so in such cases, we adjust the length formula to be right - left to include all elements in the range without counting the element at the "right" position twice.

foreach loop ->

```
for(int ele:array){
        System.out.print(ele);
}
```

- **Need of HashMap**



Syntax Of HashMap -> Whenever we have to store key value pair data we will use HasMap.

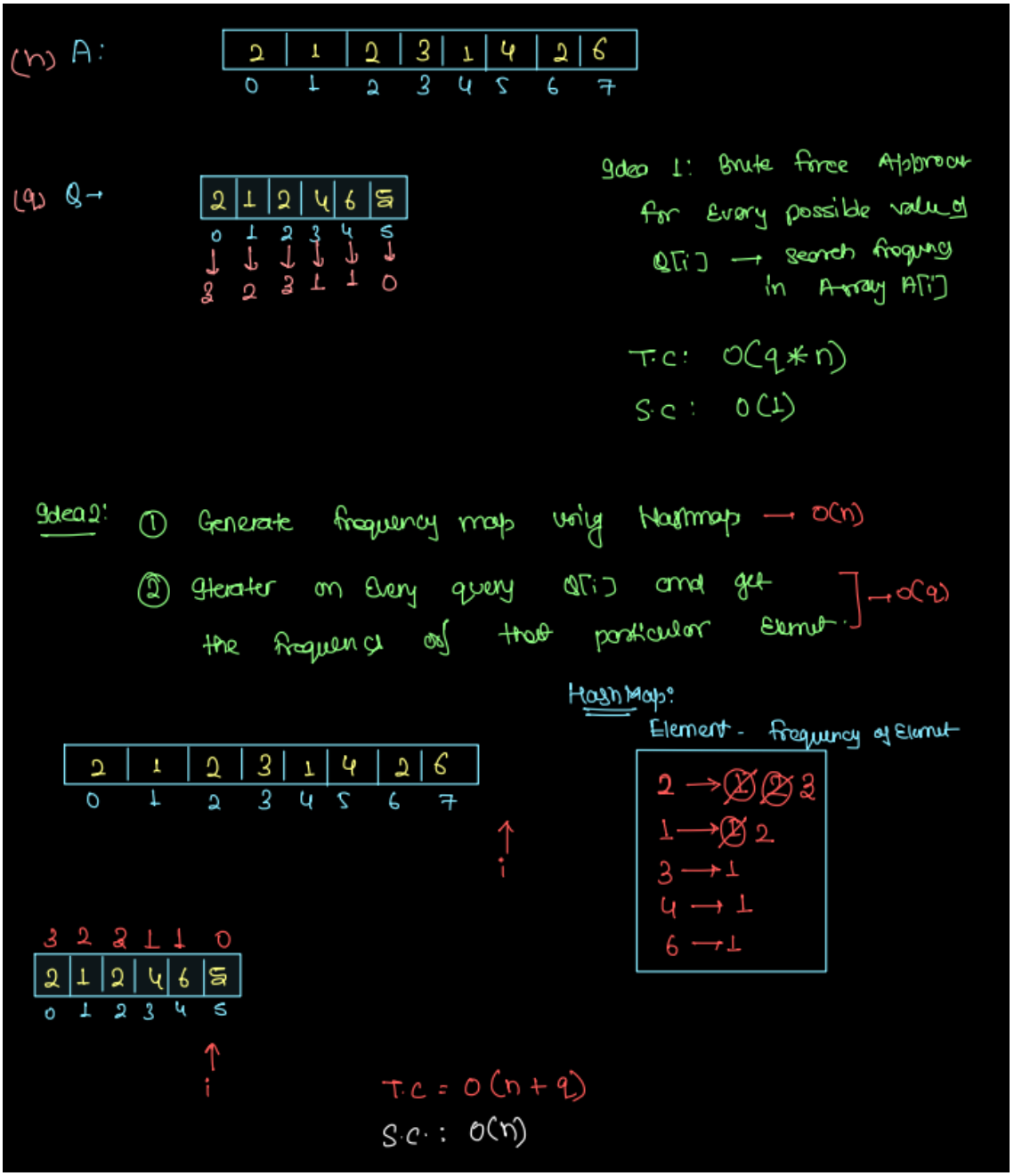
HasMap<key datatype, value datatype> variableName = new HasMap<>();

HasMap<Integer, Integer> map = new HasMap<>();

- Add values**, map.put(key, value);** -> If value is not there it will add if it is there it will update exiting value.
- Get Values **map.get(key);** -> If value is present it will return **value** else **null**

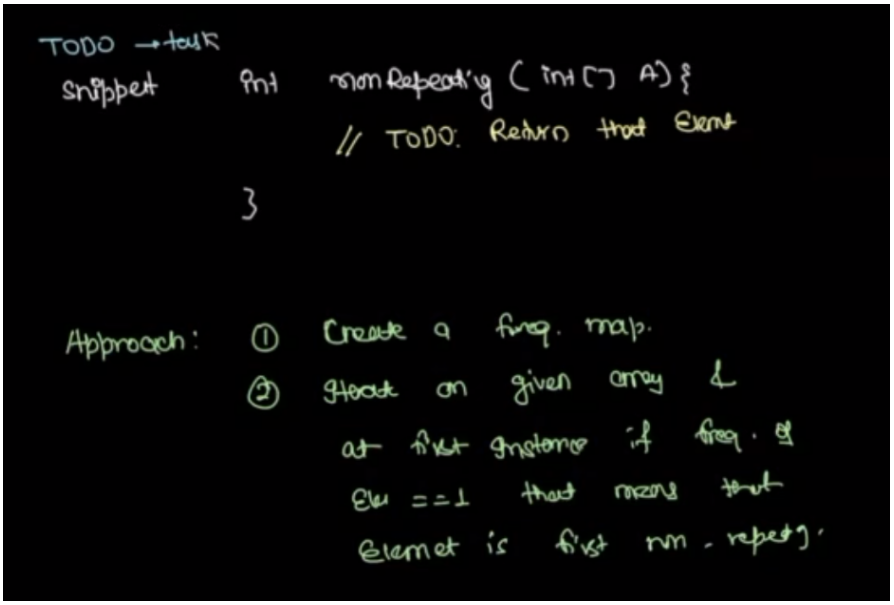
- Check Key avialable **map.containsKey(key);** -> If key is present it will return **true** else **false**
- Remove key value **map.remove(key);** -> If key is present it will remove key value pair and return **value** associated with that key. If not present then it will return **null**
- Count keys **map.size() ->** count keys in map Hashmap
- get all keys **map.keySet() ->** It woll give you all keys as array

Problem 1: Count Frequency
Given an array A and query Q. find how many time Q[i] is avaialble in array.



```java
public static void frequencyOfEle(int[] A, int[] Q) {
    // make a frquency map
    HashMap<Integer, Integer> fmap = new HashMap<>();
    /*
     * Key -> element of array A
     * Value -> frquency of that element
     */
    for(int ele : A) {
        if(fmap.containsKey(ele) == false) {
            // insert new pair of ele with freq = 1
            fmap.put(ele, 1);
        } else {
            // increase old freq by 1
            int oldFreq = fmap.get(ele);
            fmap.put(ele, oldFreq + 1);
        }
    }
    // Iterate on every query and just get the freq. from fmap
    for(int val : Q) {
        if(fmap.containsKey(val) == true) {
            System.out.println(val + " : " + fmap.get(val));
        } else {
            System.out.println(val + " : " + 0);
        }
    }
}
```

Probelm2 - Non Repeating Element
Given an array A first non repeating element
A: [2,5,4,5,2,6]
Non repeating -> 4,6
First Not repeating -> 4



Syntax Of HashSet -> Whenever we have to store only unique values we will use it.

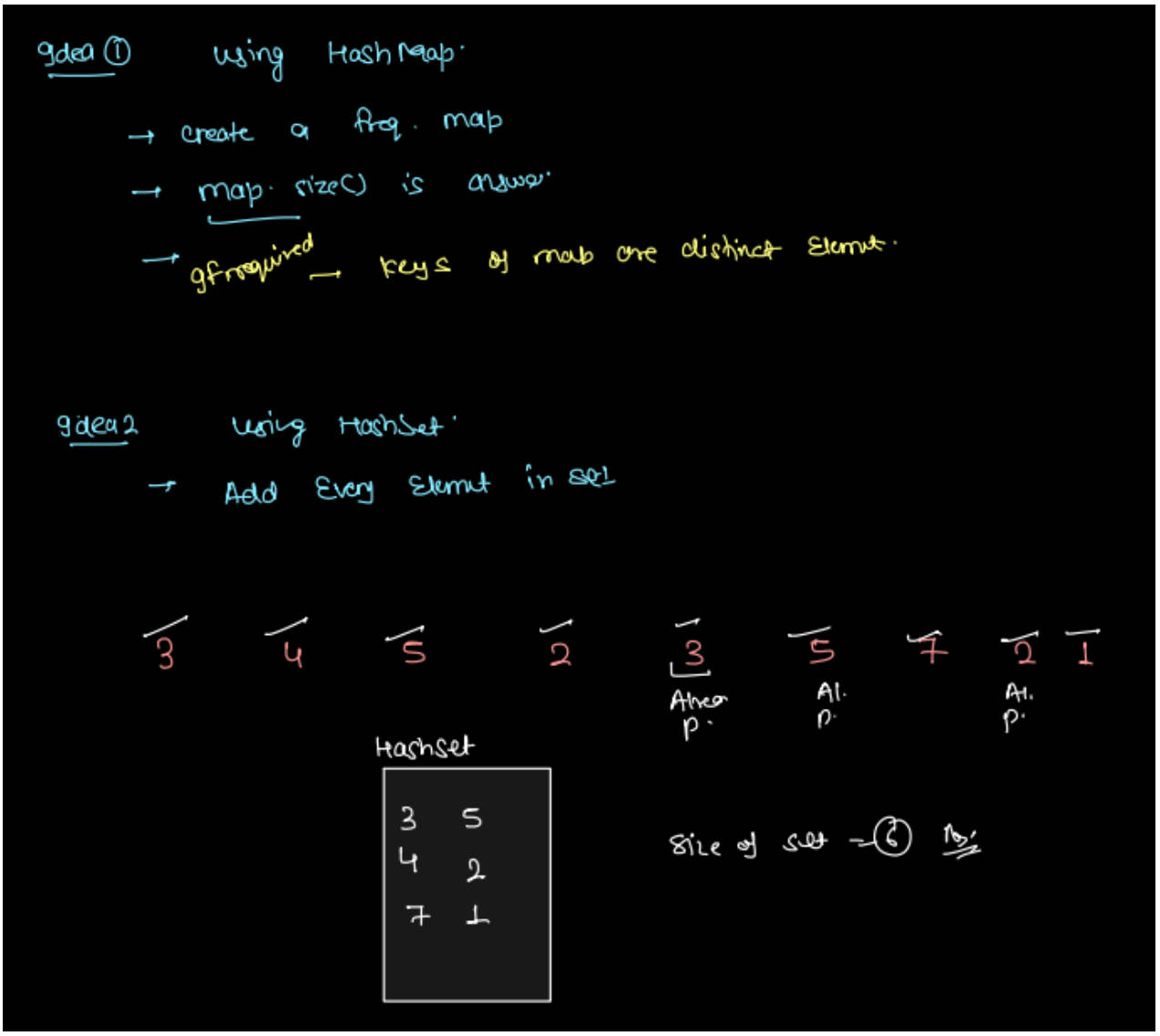
HasSet<key datatype> variableName = new HasSet<>();

HasSet<Integer> set = new HasSet<>();

- Add values**, set.add(value);**
- **If we print it directly it will print value not address**
- Get Values **It has no get function**
- Check value available **set.contains(value);** -> If value is present it will return **true** else **false**
- Remove value **set.remove(value);** -> If value is present it will remove value and return **true**. If not present then it will return **false**
- Count element **set.size() ->** count keys in map Hashmap

Probelm3 - Given an array A, find distinct element available in array.

A: 3,9,3,4,5



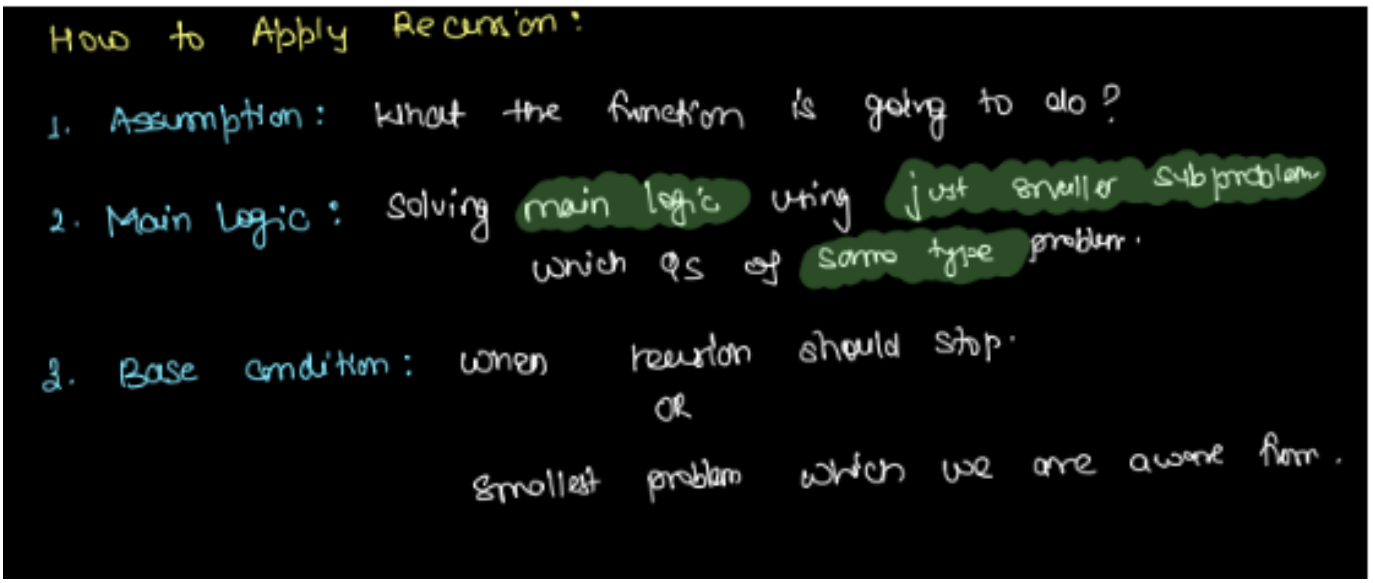
DSA: Hashing-2 - 22 - May 2023

- Target Sum Pair
- count of pair sum
- Subarray with Sum K
- Count of subarray having sum K

Problem1 - Given an array A[]. Find if there exist a pair such that A[i]+A[j] == k and i!=j.
Problem2 : Given an array A[] and value k. Find count of pairs such that sum of A[i] + A[j] == k and where i!=j.
Problem3 : Given an array. check if there exist a subaaray with sum k or not
Problem4 - Given an array, count total number of subarray having sum = k.

DSA: Recursion-1- 27 - May 2023

- Call Stack
- Recursion - Start
- Factorial using recursion
- nth fibbonacci using recursion
- Euler Diagram OR tree diagram
- Print Increasing number
- Check is subarray of character is palindromic or not
- Recursion -> function make a call to itself is know as recursion.



Assumption : Sum from 1 to n

```
int sum (int n) {
    // Base case
    if(n==1) {
        return 1;
    }
    // Main logic
    int temp = sum(n-1);
    return temp + n;
}
void main () {
    int n= scn.nextInt();
    sop(sum(n));
}
```

Assumption: Given 'n', return sum for n - natural number.

Main logic:

sum(n) = sum(n-1) + n
(sum(n-1) = temp)

Base case: Smallest problem natural number, smallest problem, n=1

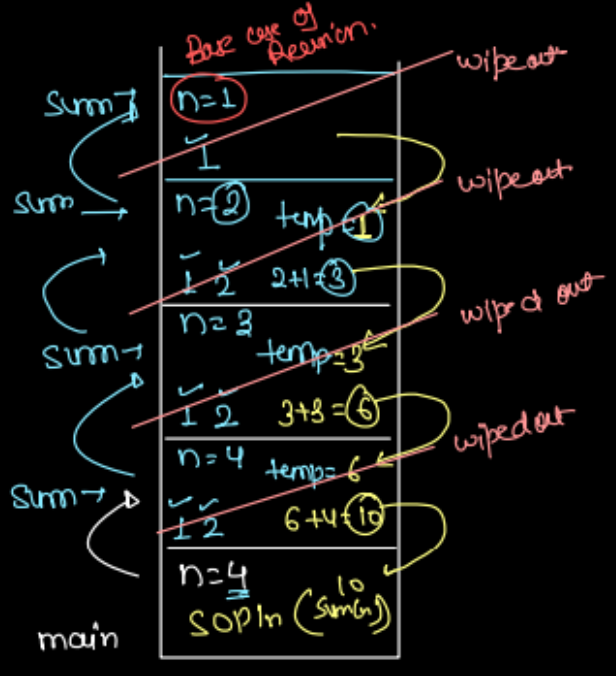
sum(1) = 1

DRY RUN 1:

Assumption : Sum from 1 to n

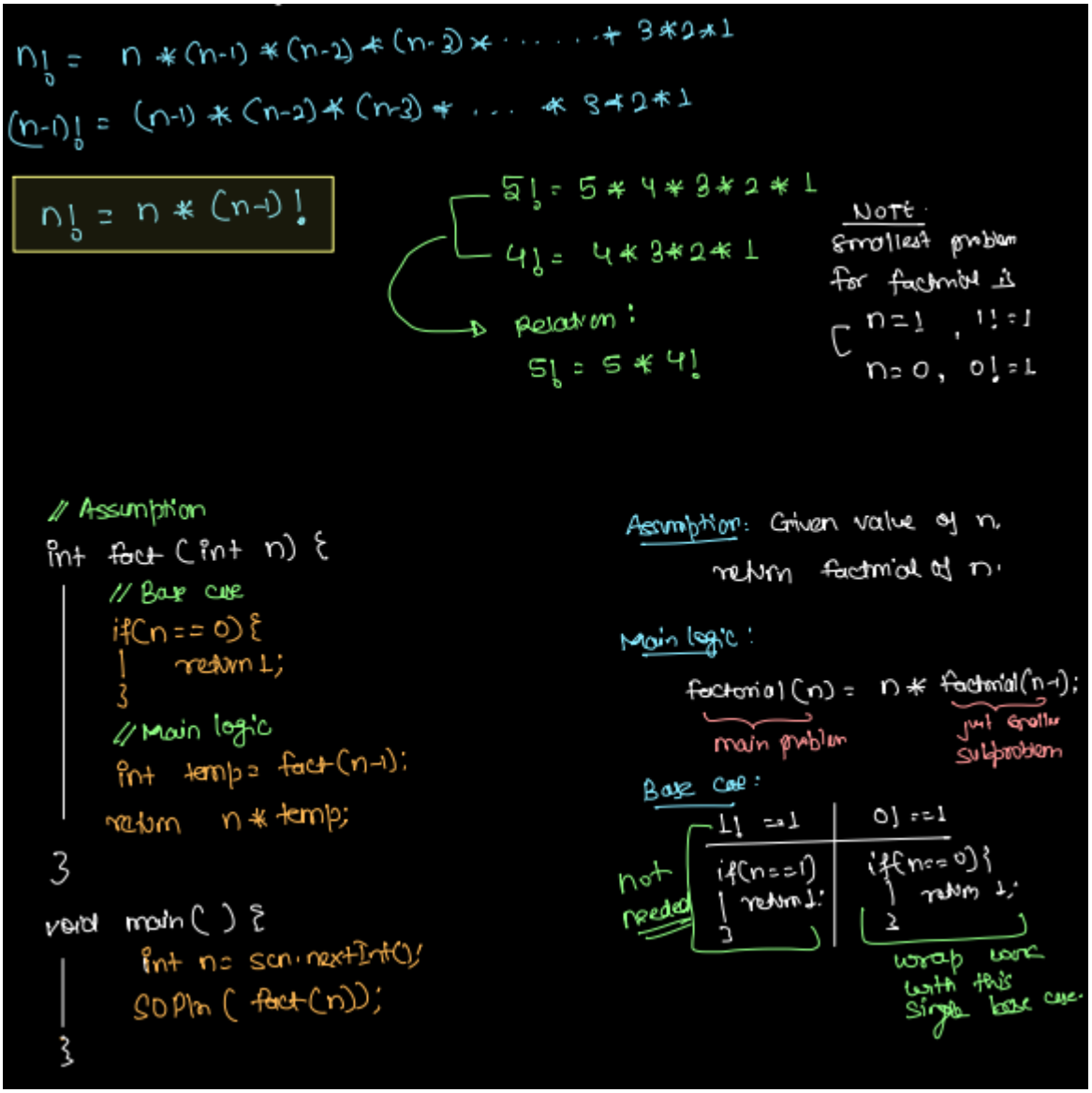
```
int sum (int n) {
    1 // Base case
      if(n==1) {
          return 1;
      }
    2 // Main logic
      int temp = sum(n-1);
      return temp + n
}
void main () {
    int n= scn.nextInt();
    sop(sum(n));
}
```

n = 10



op: 10

Problem2: Given the value of n, caclculate n factorial using recursion.



Problem3: Given n, find nth fibbonacci number (Using recurssion)

- in this first 2 number are fixed and for next number we will plus first and second and create 3 same for next we will take second and third and create 4th number.

0 1 2 3 4 5 6 7 8 9

0 1 1 (2) + (3) = [5] 8 13 21 34

```
// Assumption
int fib(int n){
    // Base Case
    if(n==0 || n==1){
        return n;
    }
    //main logic
    int temp1 = fib(n-1);
    int temp2 = fib(n-2);
    return temp1 + temp2;
}
```

Assumption: Given n, find fibbonacci of n^{th} index

Main logic: $fib(5) = fib(4) + fib(3)$

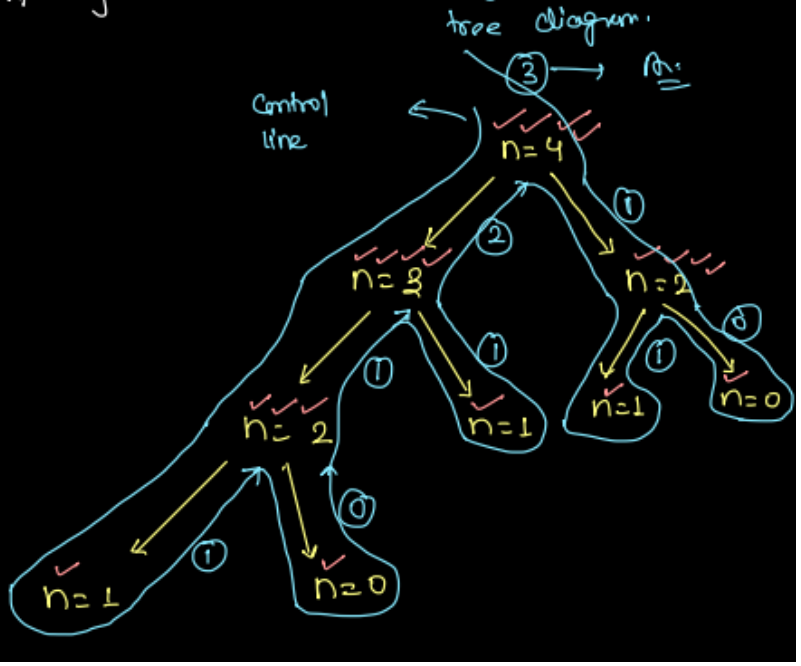
$fib(n) = fib(n-1) + fib(n-2)$

Base case:

```
if(n==0 || n==1){
    return n;
}
```

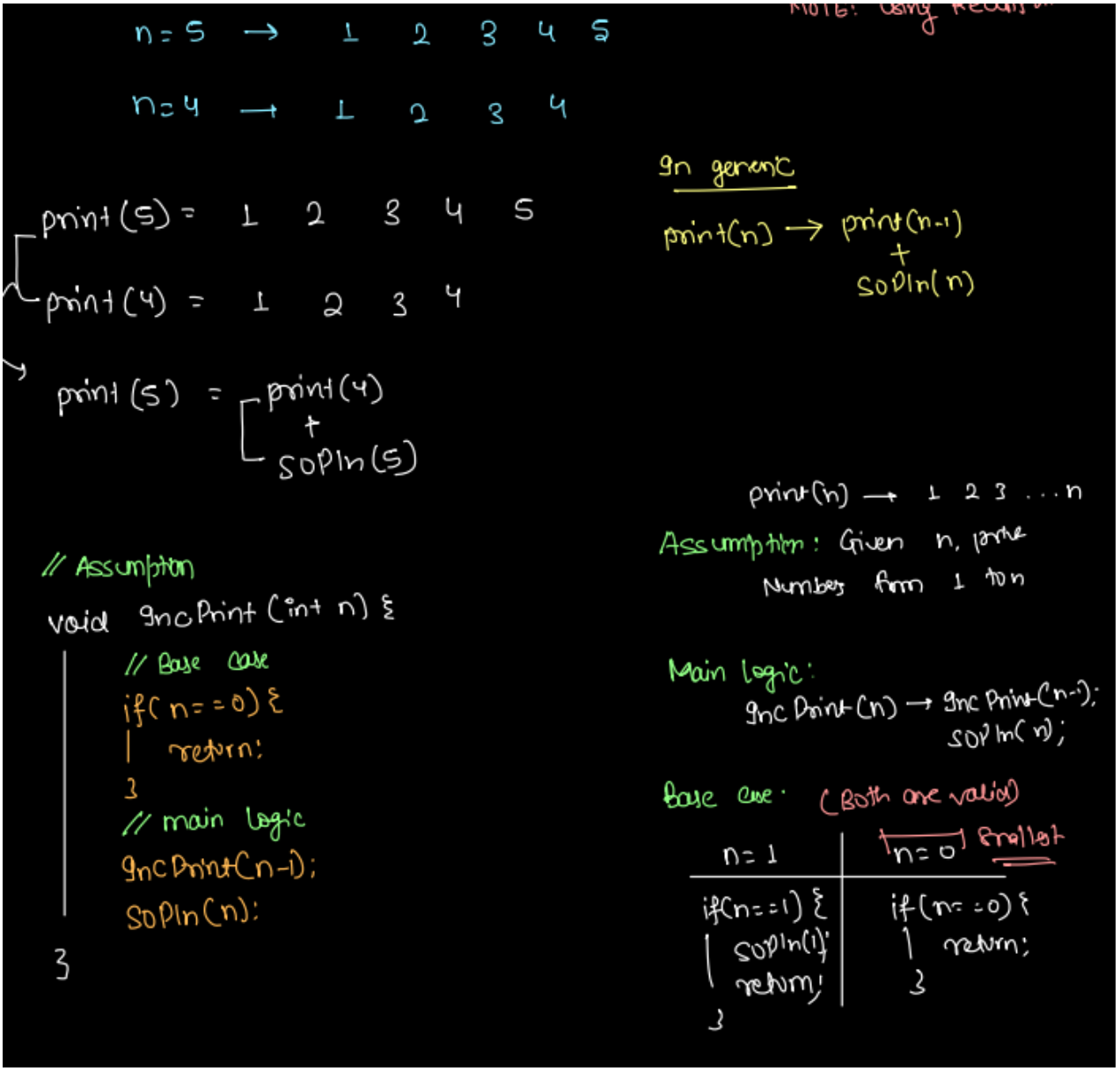
If more than one call is available, dry Run with stack diagram is slightly complex.

→ to avoid that complexity we have Euler Diagram.
OR
tree diagram.

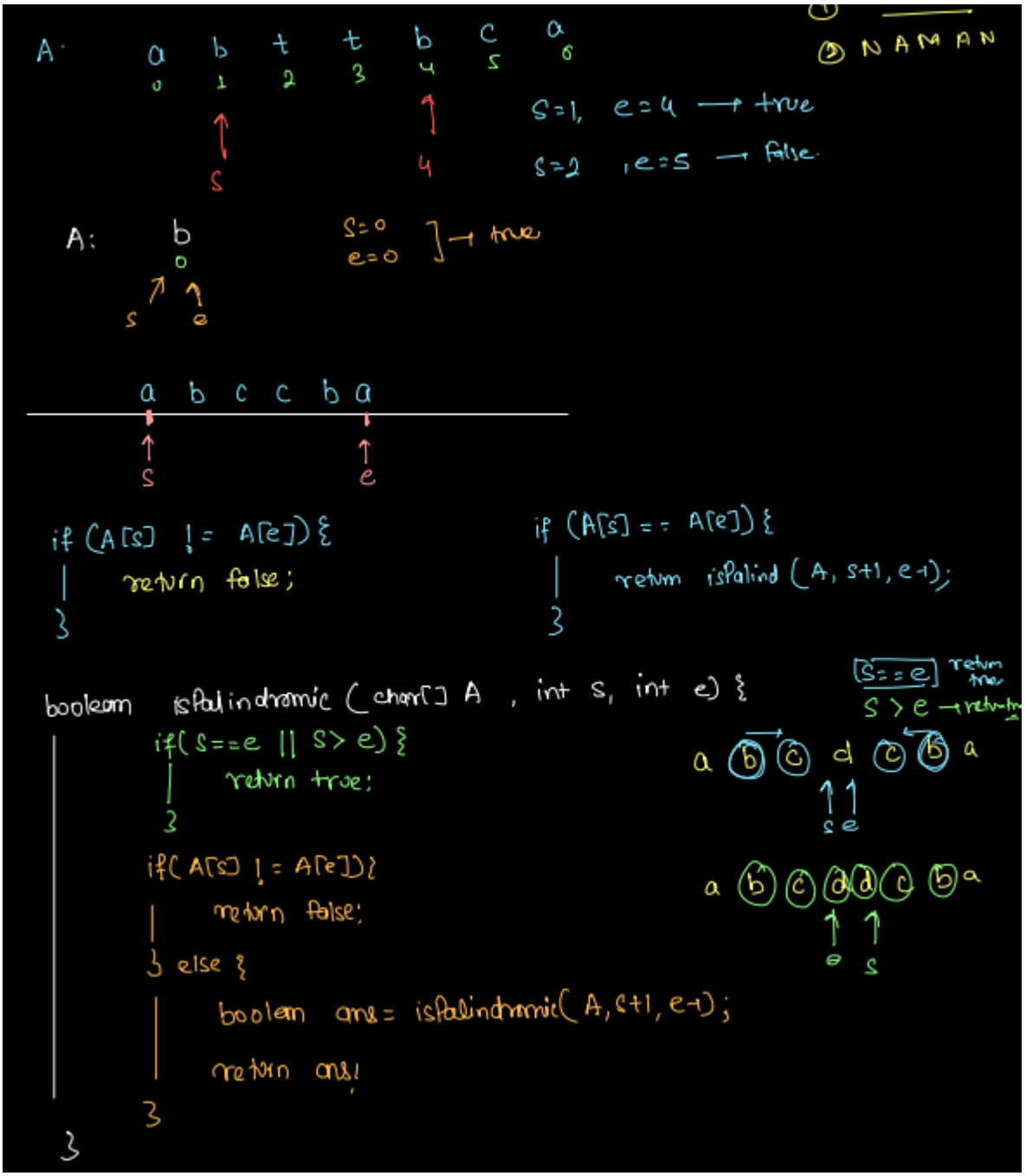



```
// Assumption
int fib(int n){
    // Base Case
1   if(n==0 || n==1){
        return n;
    }
    //main logic
2   int temp1 = fib(n-1);
3   int temp2 = fib(n-2);
4   return temp1 + temp2;
}
```

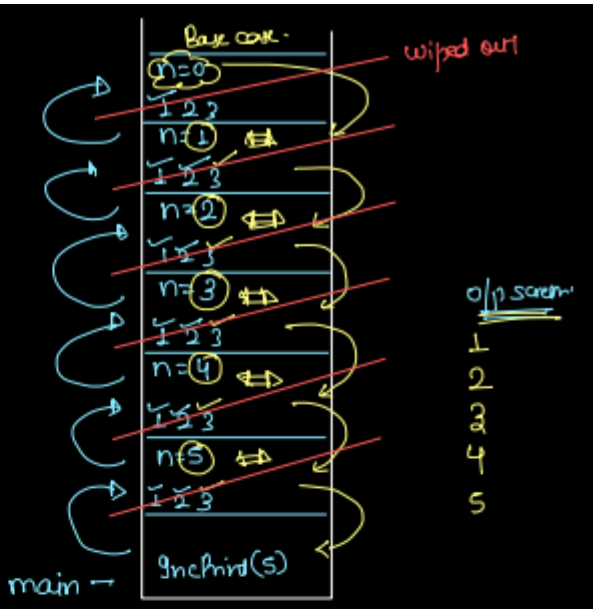
Problem4: Given n, print values from 1 to n [Incresing order] solve using recurssion



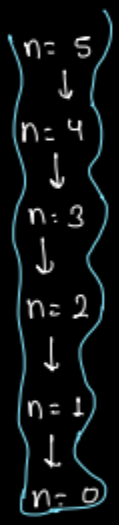
Prolem5: Given a char[] A and start index 's' and end index 'e' find if that subarray (from s to e) is palindromic or not



```
// Assumption
void incPrint(int n) {
    // Base case
    if(n==0) {
1.      return;
    }
    // main logic
2.  incPrint(n-1);
3.  SOPln(n);
}
```



tree Diagram.



o/p screen.

1
2
3
4
5

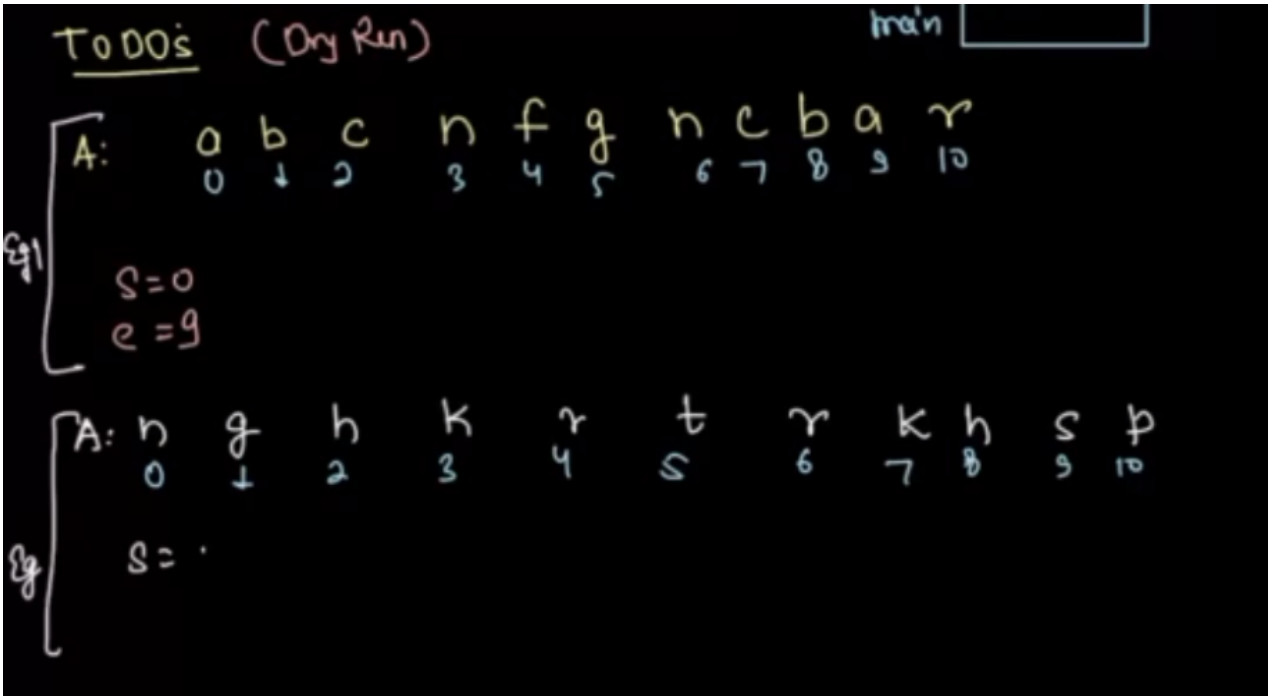
NOTE: Beese of single call, Euler Diagram is linear

TODO: Dry Run it & observe it & figure out some conclusion from problem (4).

```
void fun(int n) {
    if(n==0) {
        return;
    }
    SOPln(n);
    fun(n-1);
}
```

use Recursion

gist → loop



Notes -> DSA: Recursion 2 - 30 - May 2023

Agenda =>

1. Stack Overflow
2. sum of digit using recursion
3. calculate a^n
4. calculate a^n%p
5. Time Complexity of recursive code

1. Stack Overflow —----------

Resson behind stack overflow

- **If base case is not there**
- **If base case is present but value is high in such a way that we are causing the limit it will throw stack overflow.**
- **Stack range**
  in C++ -> 10^5 levels aprrox more then 2.4 lac
  in Java -> 10^4 level approx 10K

2. Sum of digit using recursion ----------

Problem 1: Given n, (n>0), find sum of all digits.

N = 1104 ⟶ 6 sum = 1+1+0+4 = 6

N = 128 ⟶ 11 , sum = 1+2+8 = 11

1248%10

sum(1248) = sum(124) + 8

1248/10

Assumption: Given n, return sum of digits of n.

Main Logic:

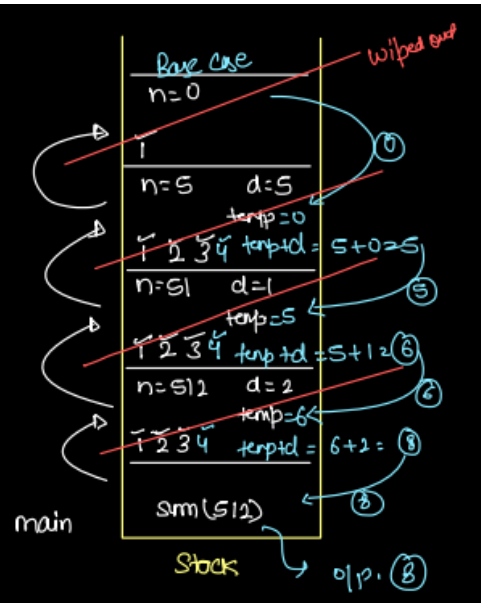
sum(n) = sum(n/10) + (n%10)

base case:

```
if(n=0){
    return 0;
}
```

```
int sumofDigit(int n){
    if(n==0){
        return 0;
    }
    int d = n%10;
    int temp = sumofDigit(n/10);
    return temp + d;
}
```

```
int sumofDigit(int n){
1{  if(n==0){
        return 0;
    }
2{  int d = n%10;
3{  int temp = sumofDigit(n/10);
4{  return temp + d;
}
```



512 = 5+1+2

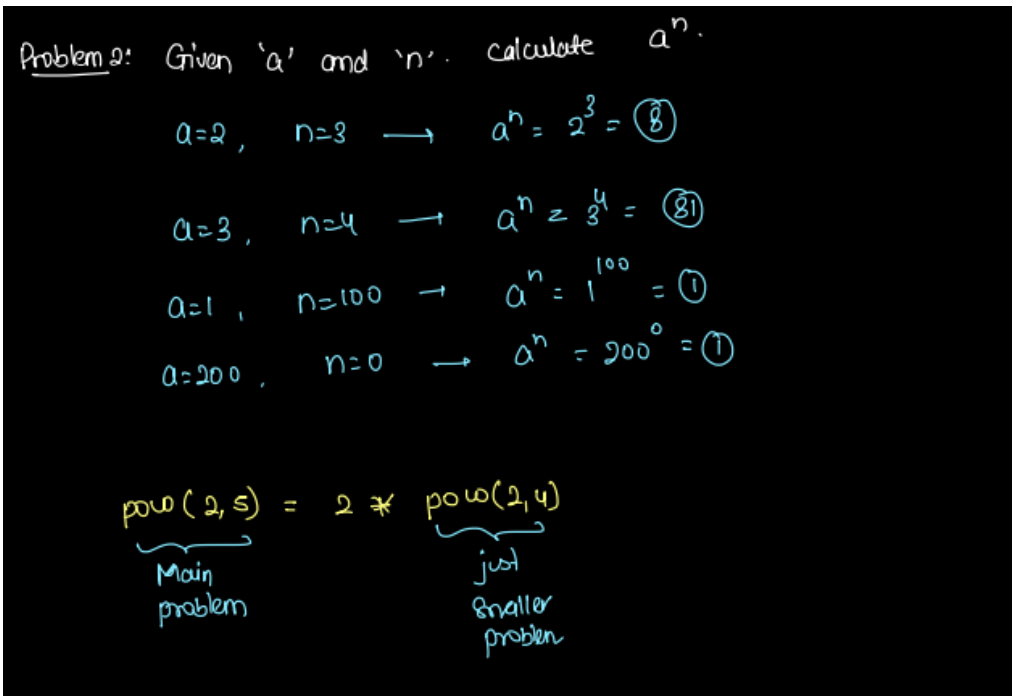
= 8 Ans

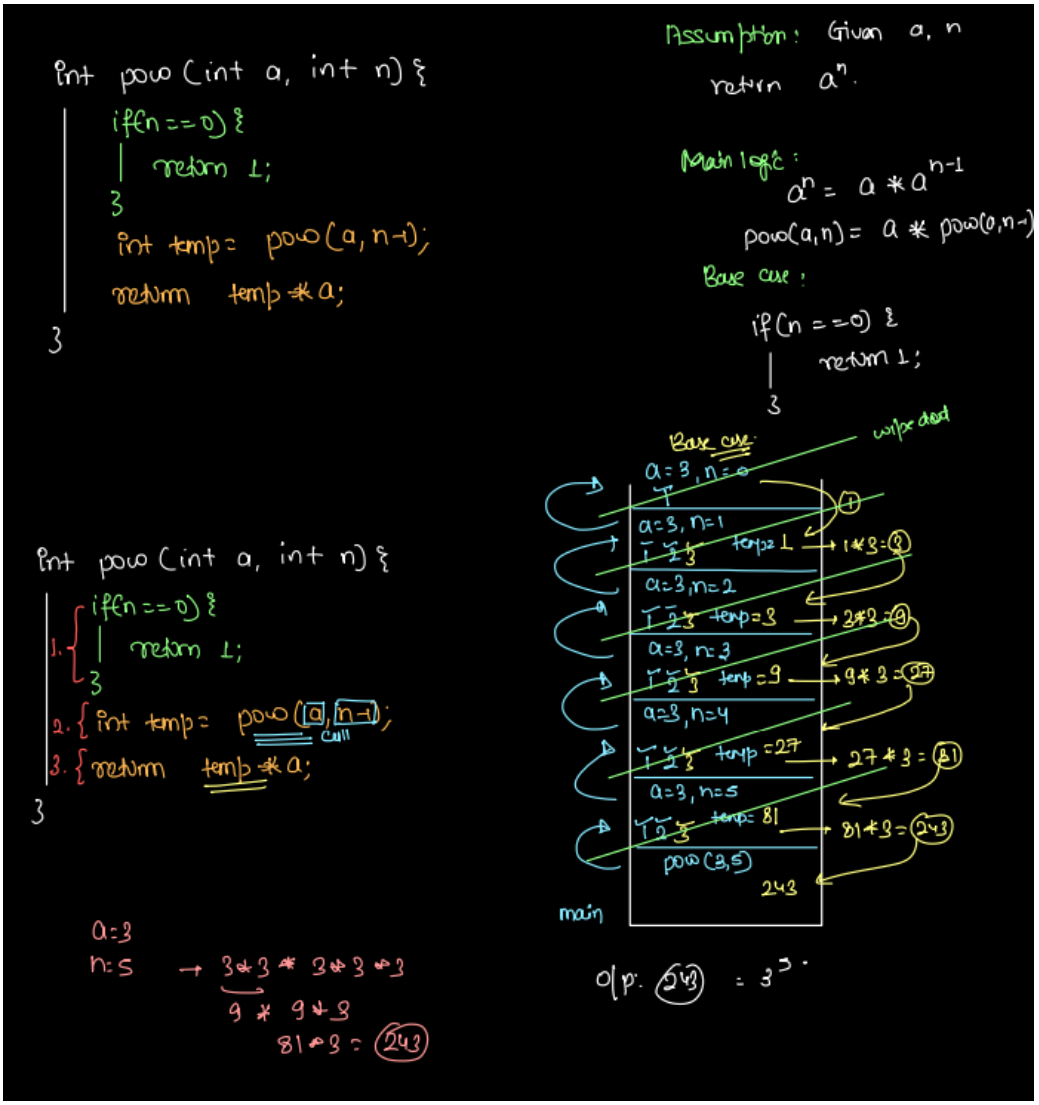
TODO: Dry Run

① n = 418 ③ n = 3142

② n = 926 ④ n = 10034

3. calculate a^n ----------





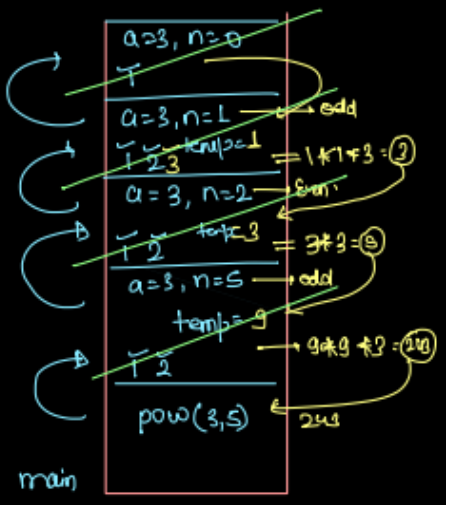
Observation:

$5^8 \Rightarrow 5^4 * 5^4$

$5^9 \Rightarrow 5^4 * 5^4 * 5$

$$a^n \rightarrow \begin{cases} a^{n/2} * a^{n/2} & \text{[if n is Even]} \\ a^{n/2} * a^{n/2} * a & \text{[if n is odd]} \end{cases}$$

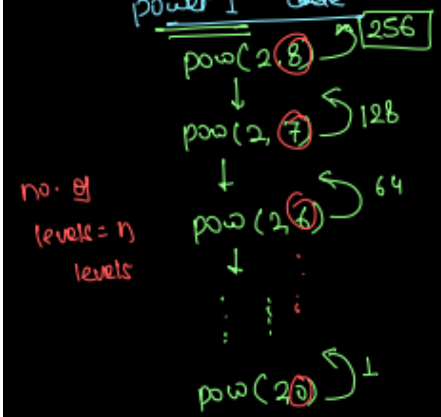
```
int pow(int a, int n){
1.  if(n==0){
        return 1;
    }
2.  int temp = pow(a, n/2);
3.  if(n%2==0){          // Even
        return temp * temp;
    } else {
        return temp * temp * a;
    }
}
```



$3^5 = 243$

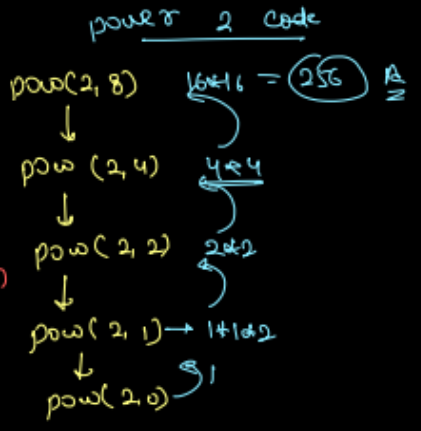
a=2, n=8 [Analysis of Depth of Recursion]

power 1 code



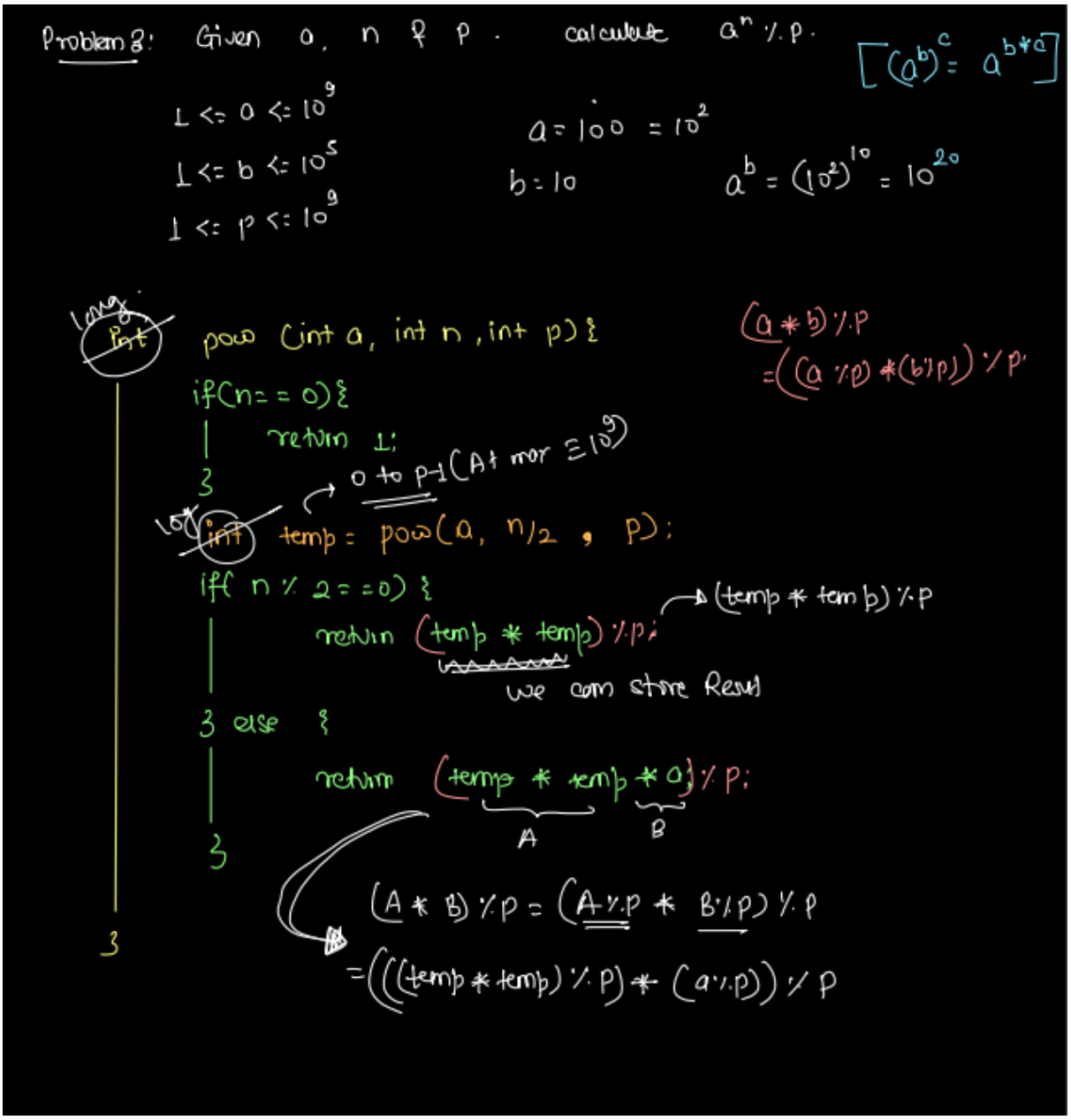
no. of levels = n levels

power 2 code

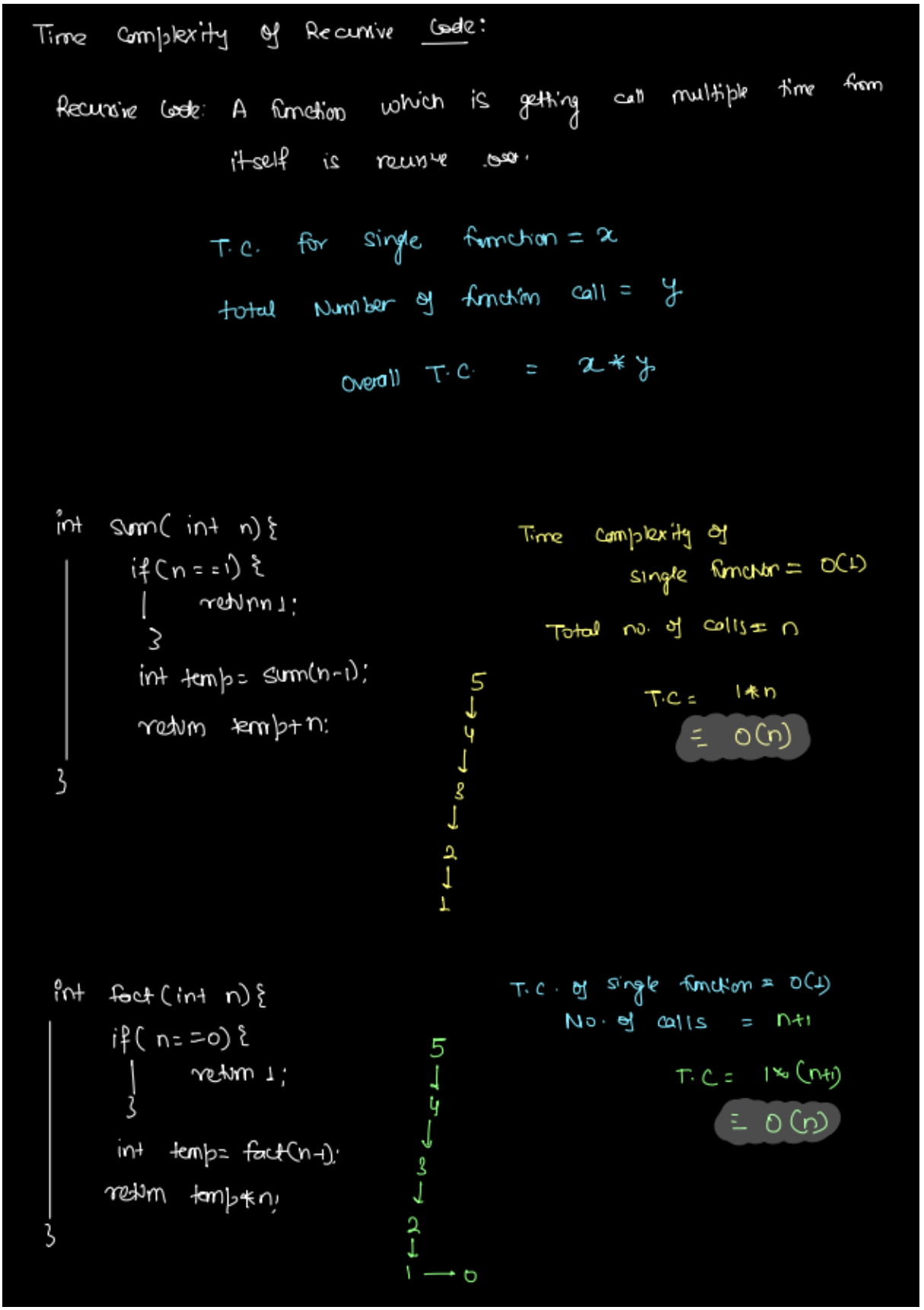


no. of levels = log n

4. calculate a^n%p ——----------



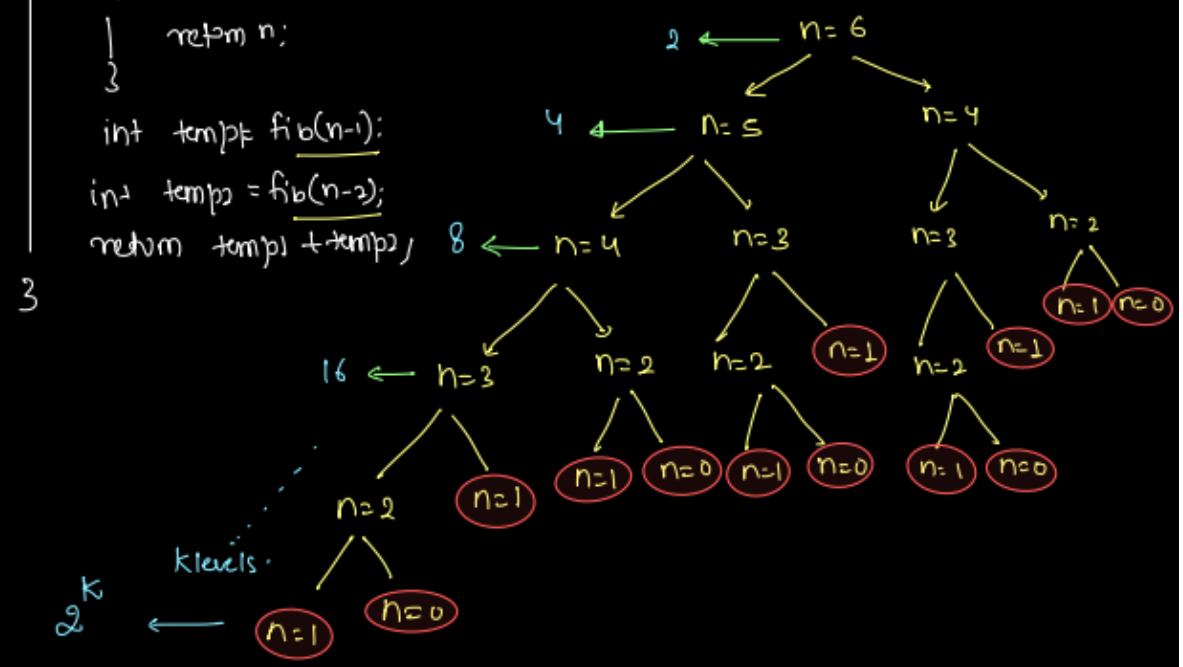
5. Time Complexity of recursive code —----------



```
int fib(int n) {
    if(n==0 || n==1){
        return n;
    }
    int temp1 = fib(n-1);
    int temp2 = fib(n-2);
    return temp1 + temp2;
}
```

T.C. for single function = O(1)

No. of call = ? $= 2^n$



Value of $k = ?$

Value of n is equal to number of level.

$k = n$.

total number of call $= 2^1 + 2^2 + 2^3 + \cdots + 2^n$

Sum of G.P (Geometric Progression) with $a = 2$, $r = 2$, $t = n$

$$\text{Sum} = \frac{a(r^t - 1)}{r - 1}$$

=] equal

≡] equivalent

≈ Approximate ⟹

$$\text{total call} = \frac{2(2^n - 1)}{2 - 1} = 2^{n+1} - 2 \approx 2^n]$$

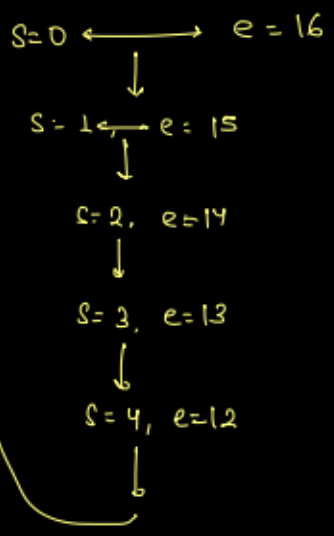
T.C. $= 1 * 2^n$

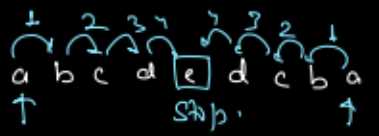
$= O(2^n)$

```
boolean isPalindrome(char[] A, int s, int e){
    if(s==e || s>e){
        return true;
    }
    if(A[s] != A[e]){
        return false;
    } else {
        return isPalindrome(A, s+1, e-1);
    }
}
```

T.C for single fn = O(1)

No. of calls = ? n/2





$T.C = 1 * \frac{n}{2} = \frac{n}{2}$

$T.C. = O(n)$

Power [Linear Power]

T.C for single function = O(1)

No. of call = n calls

```
int pow(int a, int n){
    if(n==0){
        return 1;
    }
    int temp = pow(a, n-1);
    return temp * a;
}
```

3,5
↓
3,4
↓
3,3
↓
3,2
↓
3,1
↓
3,0

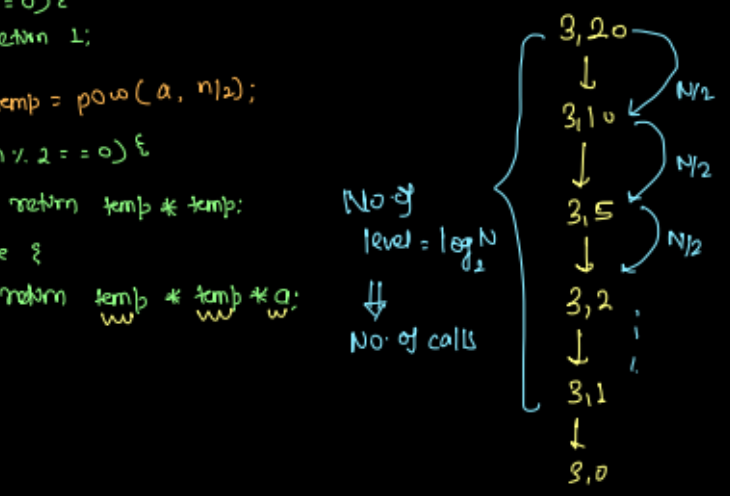
T.C = 1+n

= O(n) Ans

T.C. for single function = O(1)

No. of calls = $\log_2 N$

```
int pow(int a, int n){
    if(n==0){
        return 1;
    }
    int temp = pow(a, n/2);
    if(n%2==0){
        return temp * temp;
    } else {
        return temp * temp * a;
    }
}
```



No of level = $\log_2 N$

⇓

No. of calls

T.C : $1 * \log_2 N$

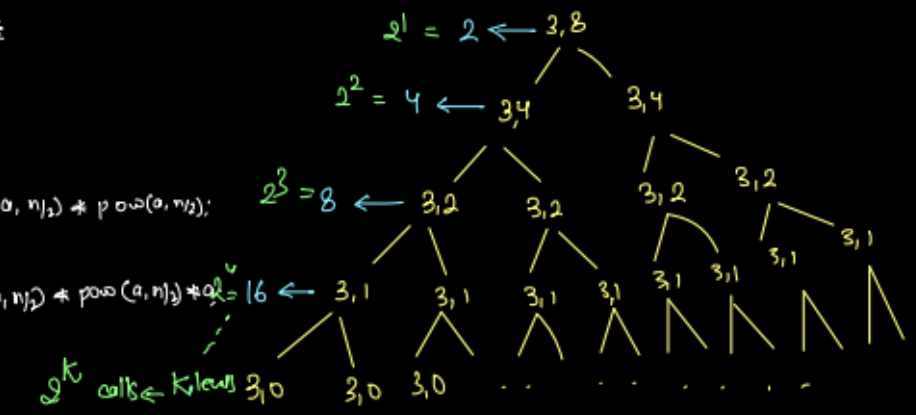
$= O(\log_2 N)$

Fake Smart / pseudo power :

T.C for single function = O(1)

No. of calls ?

```
int pow(int a, int n){
    if(n==0){
        return 1;
    }
    if(n%2==0){
        return pow(a, n/2) * pow(a, n/2);
    } else {
        return pow(a, n/2) * pow(a, n/2) * a;
    }
}
```



Value of K → No. of levels → $\log n$ ⇒ $K = \log K$

total call = $2^1 + 2^2 + 2^3 + \cdots + 2^k$

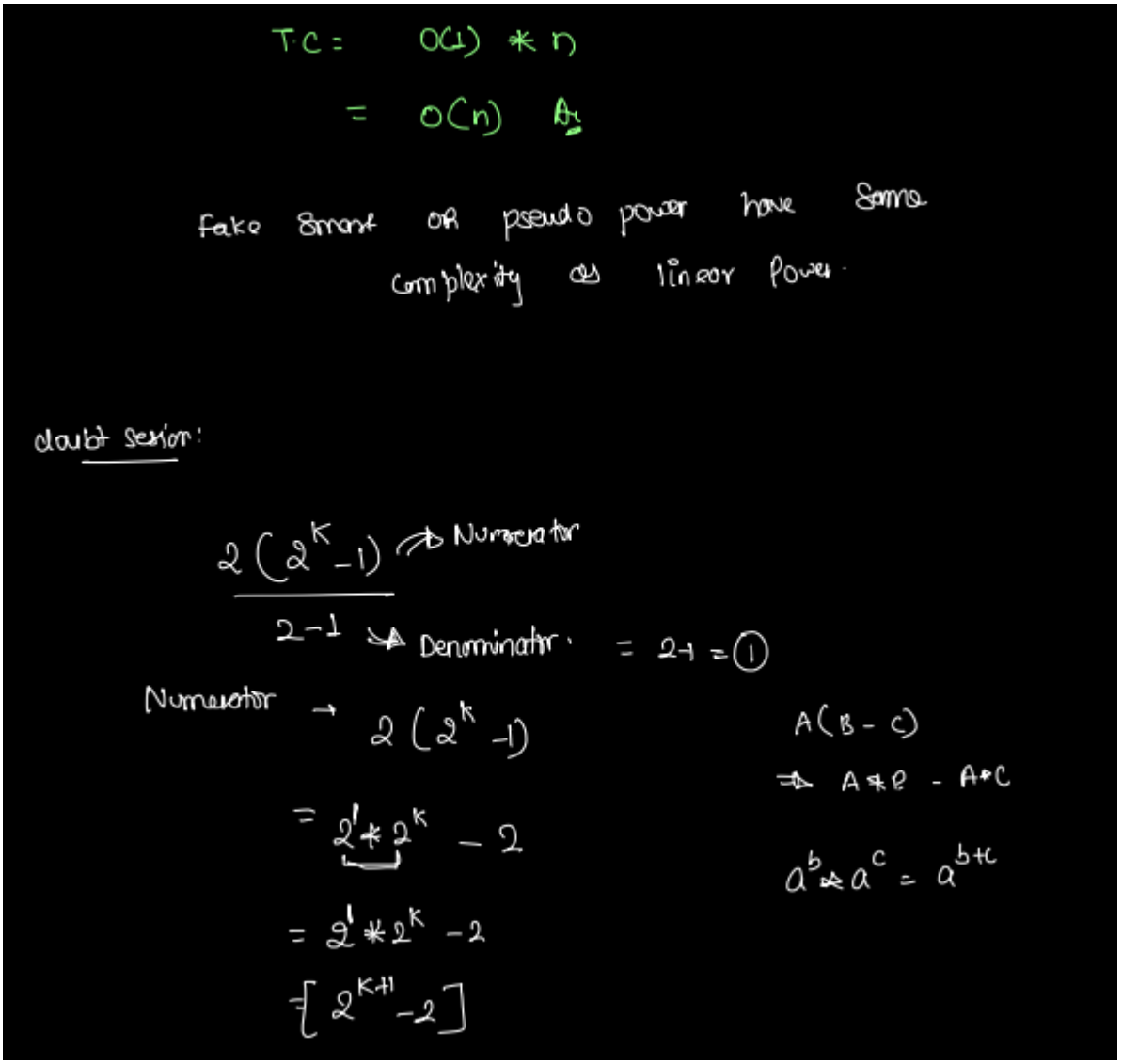
$= \frac{2(2^K - 1)}{2-1} = 2^{K+1} - 2$

$\equiv 2^{\log n}$

total call = n

$\left[a^{\log_a b} = b\right]$

$\left[a^{\log_a b} = b\right]$



Problem1: Given n, (n>0), find sum of all digits
https://www.interviewbit.com/snippet/703833adb2b2f622fcec/

```
import java.util.*;
// Create MinStack using stack
class Main{
public static int printNumSum(int n){
if(n == 0){
return 0;
}
int num = n % 10;
count = printNum(n/10);
return count + num;
}
public static void main(String args[]) {
```

```
// Problem1: Given n, (n>0), find sum of all digits
System.out.print(printNumSum(1104));
}
}
```

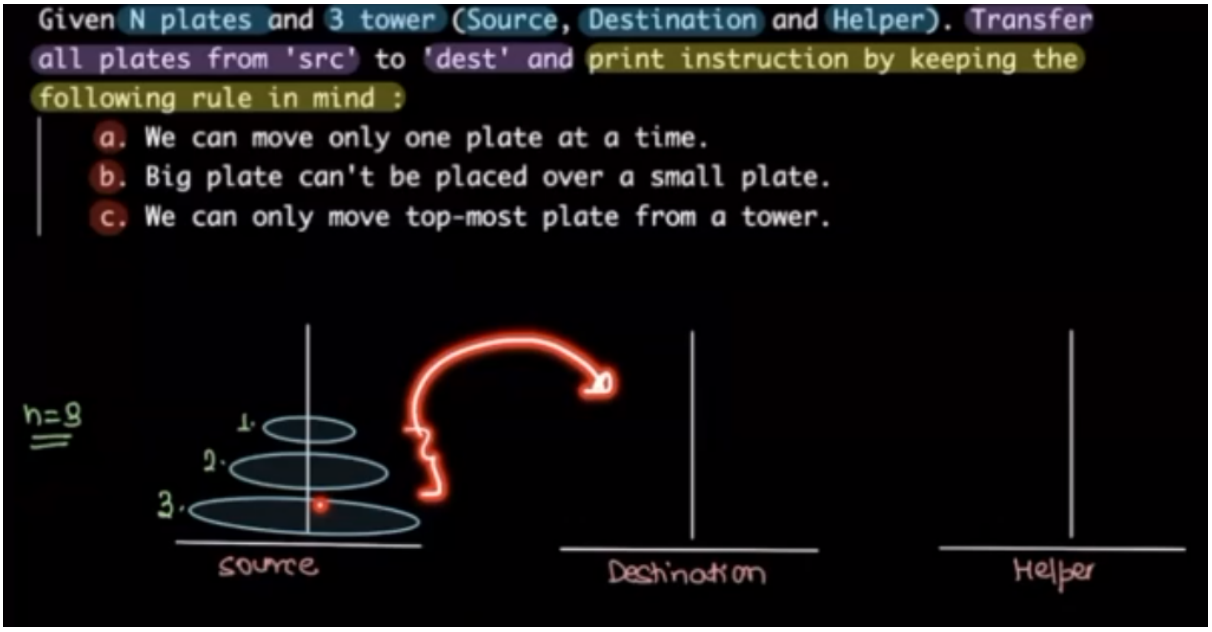
Problem2: Given ‘a’ and ‘n’. calculate a^n;

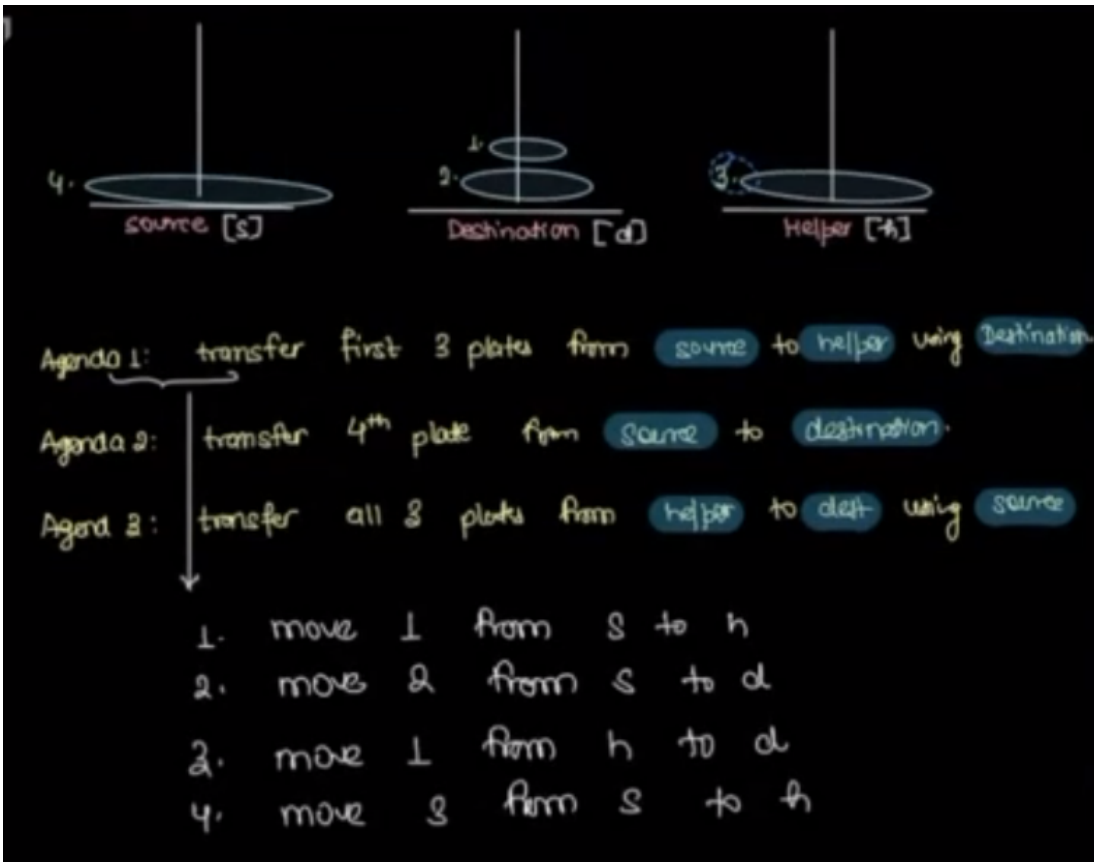
https://www.interviewbit.com/snippet/13461d4eb767e69ee29b/

```
import java.util.*;
// Create MinStack using stack
class Main{
public static int powerOfNum(int a, int n){
if(n == 0){
return 1;
}
int count = powerOfNum(a, n-1);
return a * count;
}
// this is optimized version
public static int powerOfNumOpt(int a, int n){
if(n == 0){
return 1;
}
int temp = powerOfNum(a, n/2);
if(n % 2 == 0){
return temp * temp;
}
return temp * temp * a;

}
public static void main(String args[]) {
// Problem2: Given a and n. calculate a^n
int a = 2;
int n = 5;
System.out.print(powerOfNumOpt(a,n));
}
}
```

Problem3: Given o, n & p. calcualte a^n % p.

Notes -> DSA: Recursion 3 - 1 - Jun 2023
Tower of Hanoi





Assumption: toh(n, S, d, h); (source, dest, hlp.)

↳ 'toh' function will print all the instruction to move all plates from source to destination using helper by following all rules.

Main logic:

① transfer (n-1) plates from source to hlp using dest.
toh(n-1, source, dest, hlp);

② transfer n^{th} plate from source to dest

③ transfer (n-1) plates from hlp to dest using source.

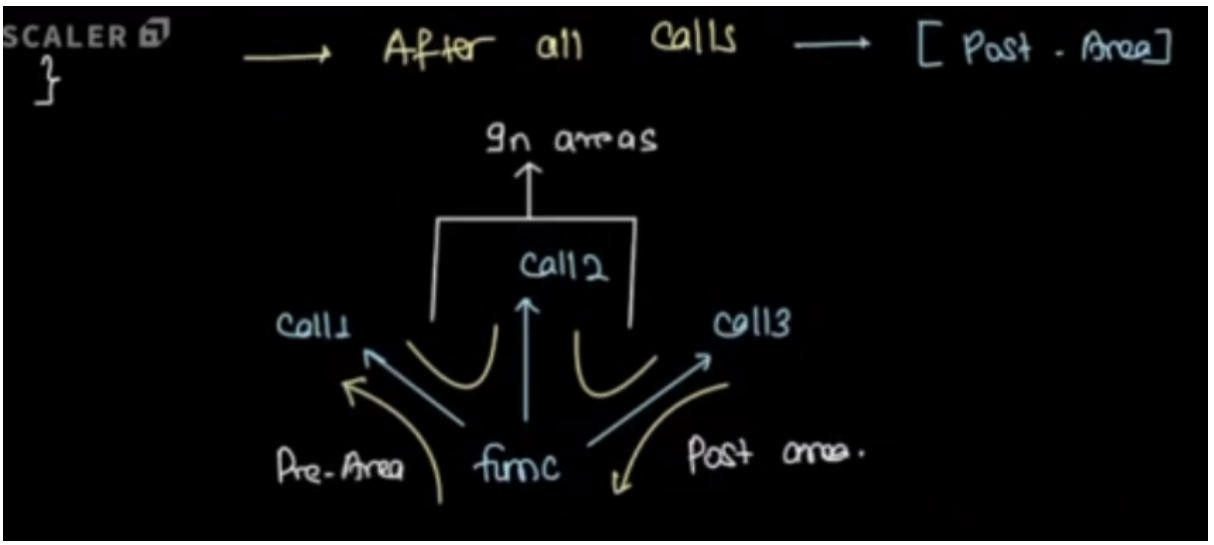
SCALER

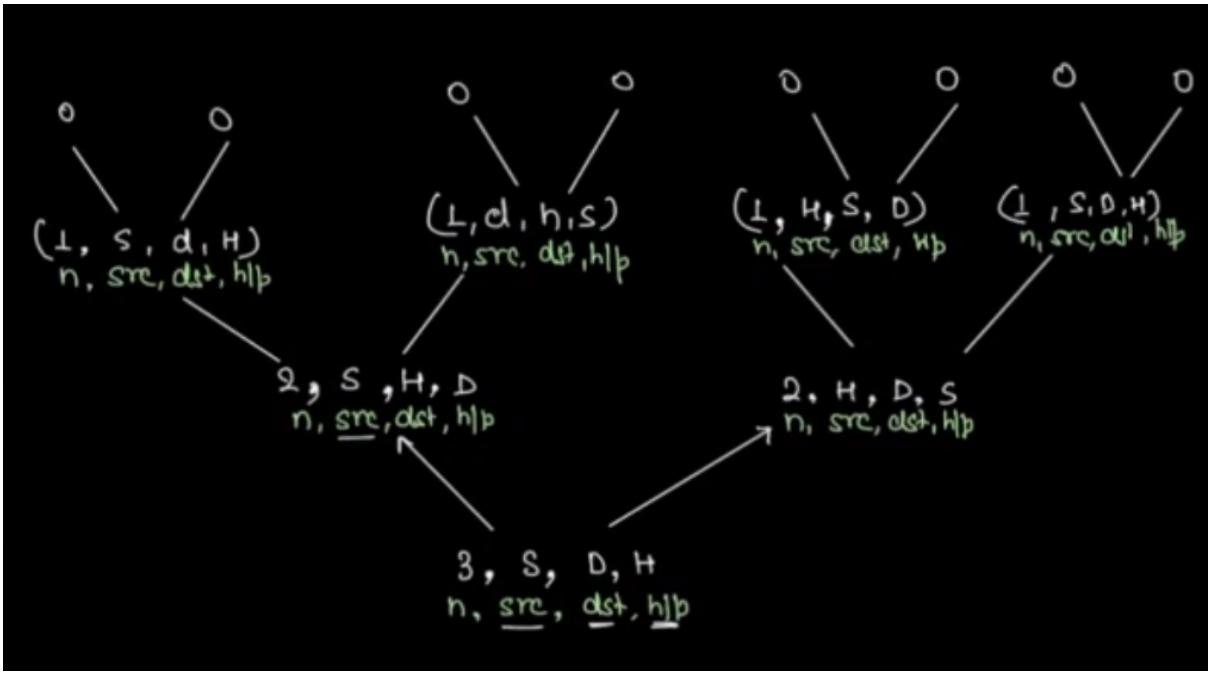
```
class Main {

    public static void toh(int n, char src, char dst, char hlp) {
        // smallest problem, if n = 0 -> don't do anything
        // base case
        if(n == 0) {
            return;
        }
        // move n-1 plates from src to help using destination
        toh(n-1, src, hlp, dst);
        // move nth plate from src to dest
        System.out.println("Move " + n + " plate from " + src + " to " +
        // move n-1 plates from hlp to dest using src
        toh(n-1, hlp, dst, src);
    }


    public static void main(String args[]) {
        // Your code goes here
        int n = 3;
        toh(n, 'S', 'D', 'H');
    }
}
```

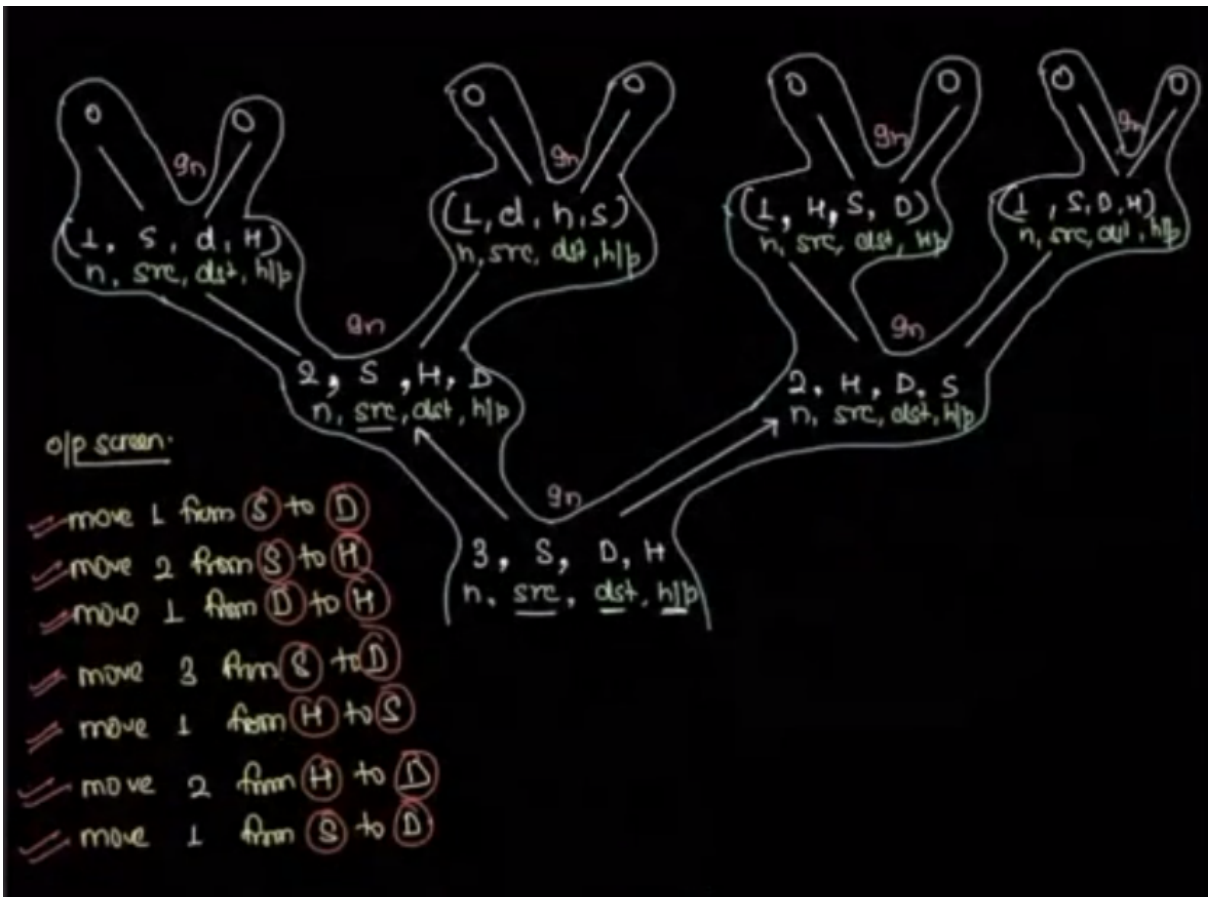
Custom Input

Output

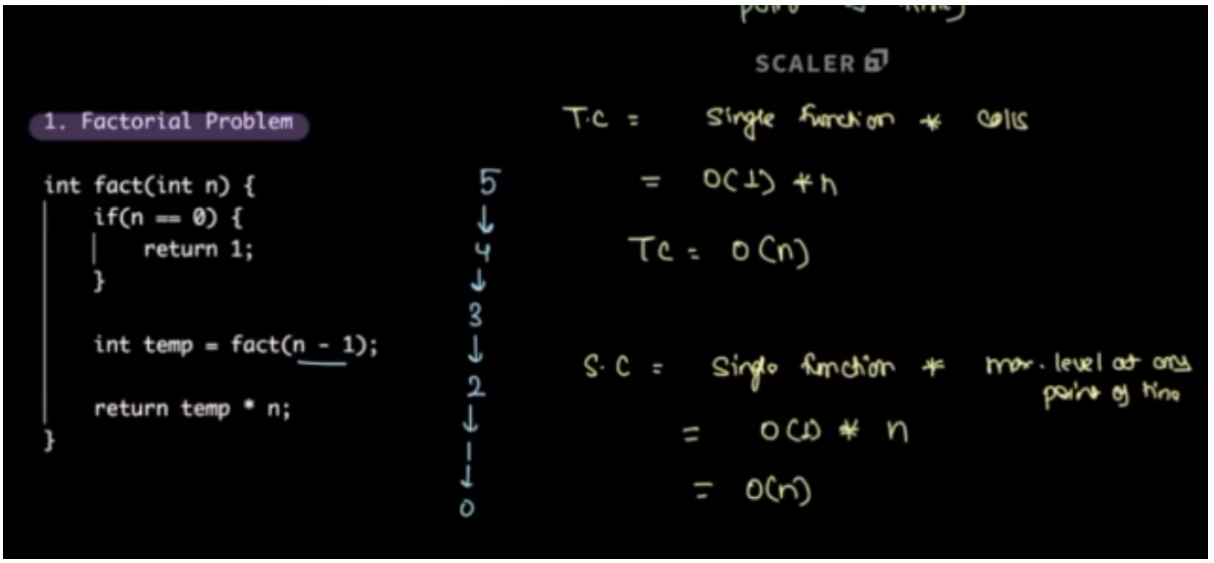
```
[Success] Your code was executed
successfully

Move 1 plate from S to D
Move 2 plate from S to H
Move 1 plate from D to H
Move 3 plate from S to D
Move 1 plate from H to S
Move 2 plate from H to D
Move 1 plate from S to D
```


SCALER
After all calls
[Post - Area]
9n areas
call2
call1
call3
Pre-Area
func
Post area.


(1, S, d, H)
n, src, dst, hlp
(1, d, h, S)
n, src, dst, hlp
(1, H, S, D)
n, src, dst, hlp
(1, S, D, H)
n, src, dst, hlp
2, S, H, D
n, src, dst, hlp
2, H, D, S
n, src, dst, hlp
3, S, D, H
n, src, dst, hlp



Space Complexity in recursion
T = Single function * No. of calls
S = Single function * Max level of any point of time

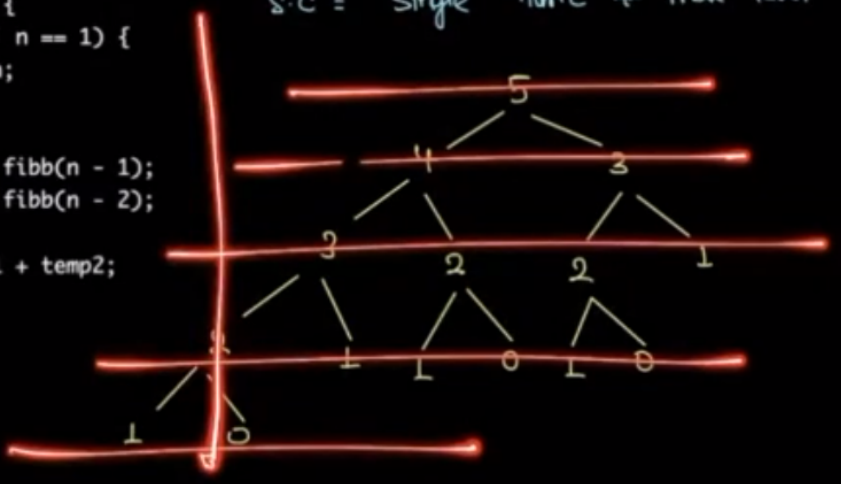


2. Fibbonacci Problem

T.C: $O(2^n)$

S.C = Single func * max level

```
int fibb(int n) {
    if(n == 0 || n == 1) {
        return n;
    }

    int temp1 = fibb(n - 1);
    int temp2 = fibb(n - 2);

    return temp1 + temp2;
}
```



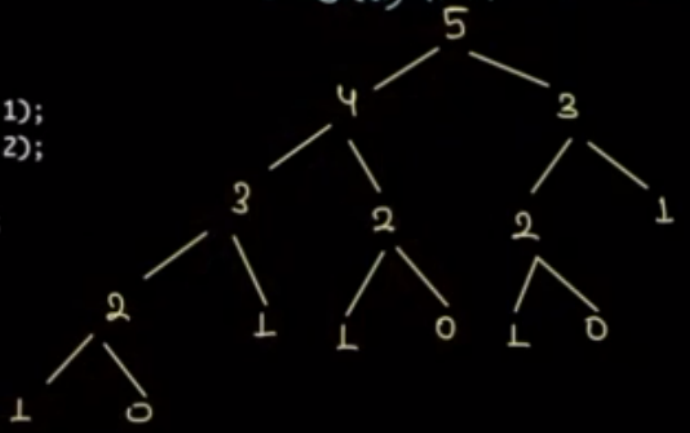
2. Fibbonacci Problem

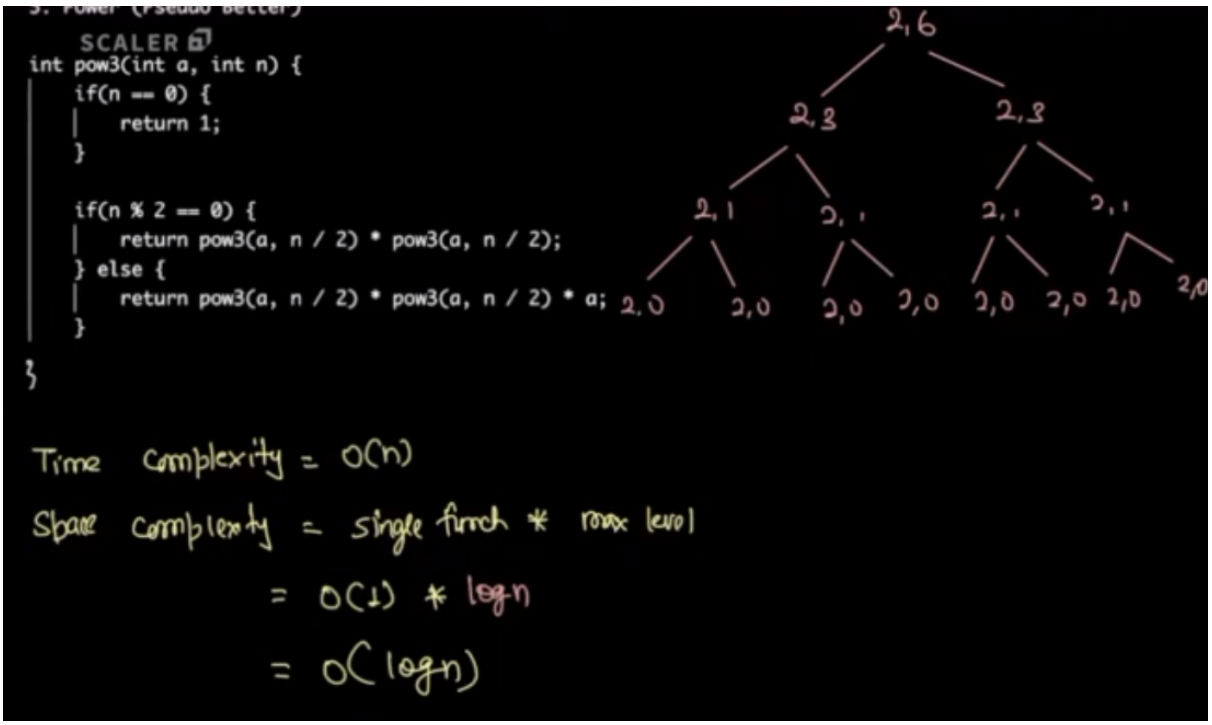
T.C: $O(2^n)$

S.C = Single func * max level

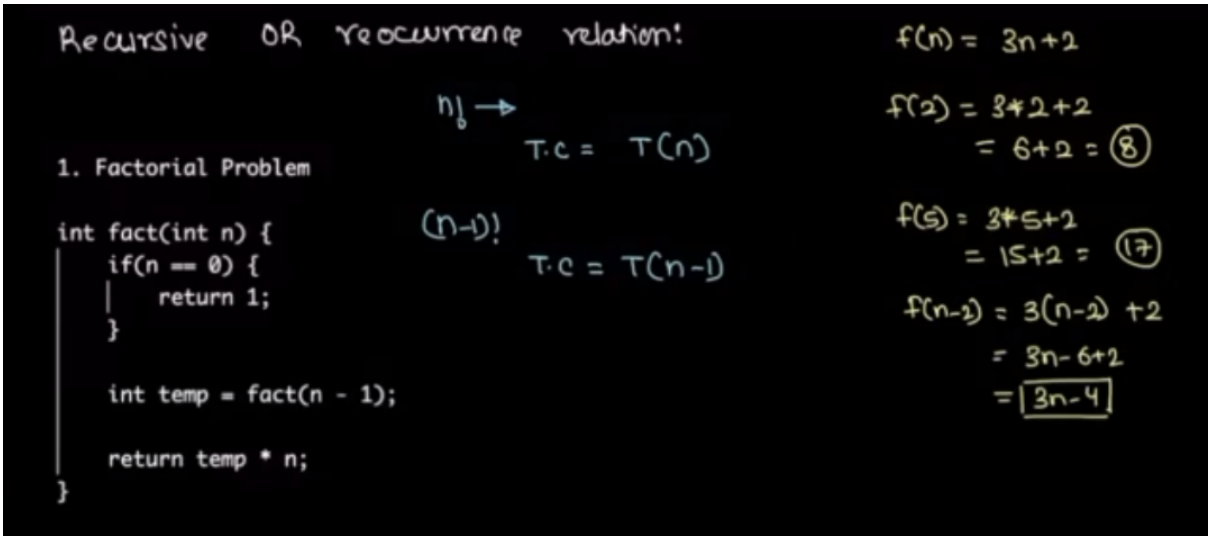
```
int fibb(int n) {
    if(n == 0 || n == 1) {
        return n;
    }

    int temp1 = fibb(n - 1);
    int temp2 = fibb(n - 2);

    return temp1 + temp2;
}
```

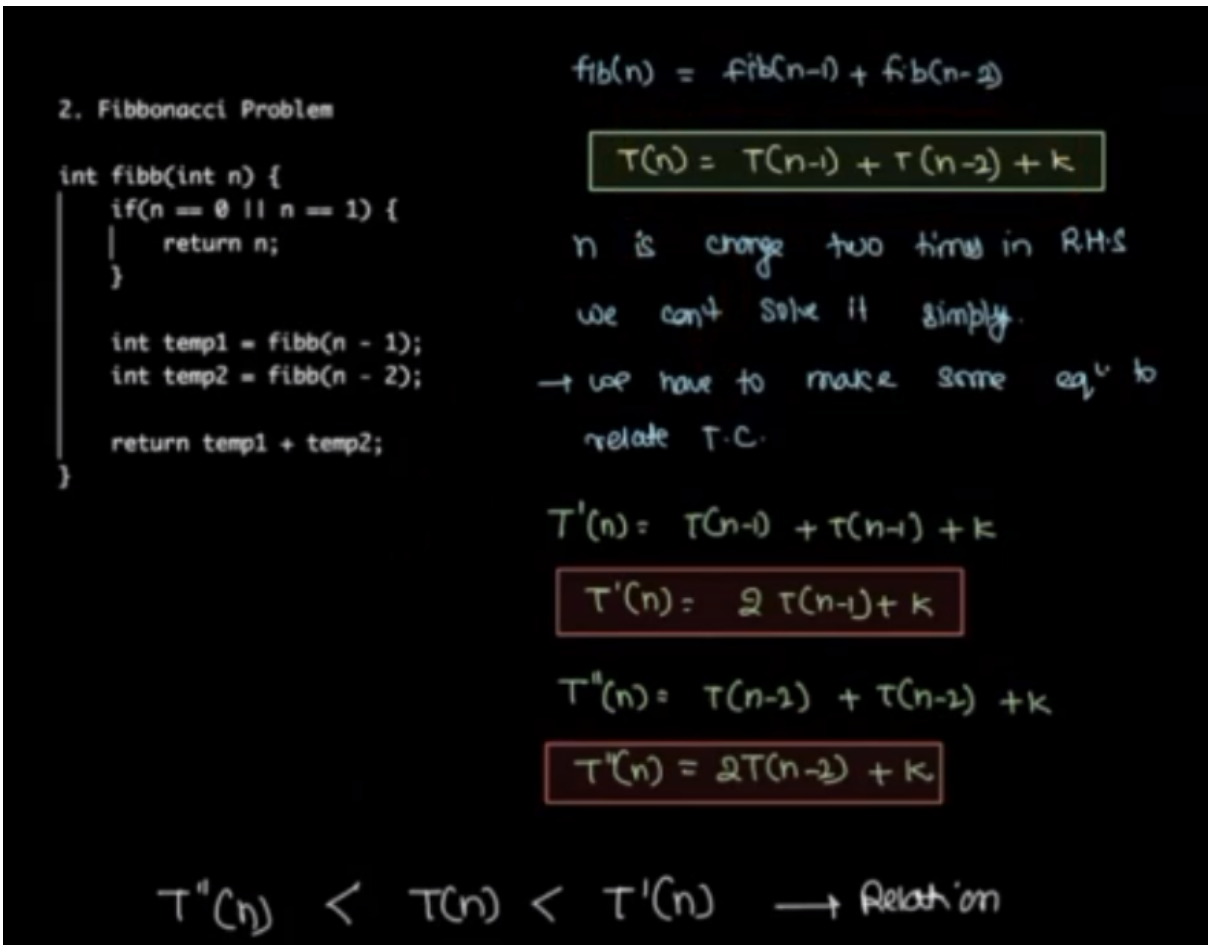
$= O(1) * n = O(n)$





Recursive OR reoccurrence relation





DSA: Sorting 1 - 6 - June 2023

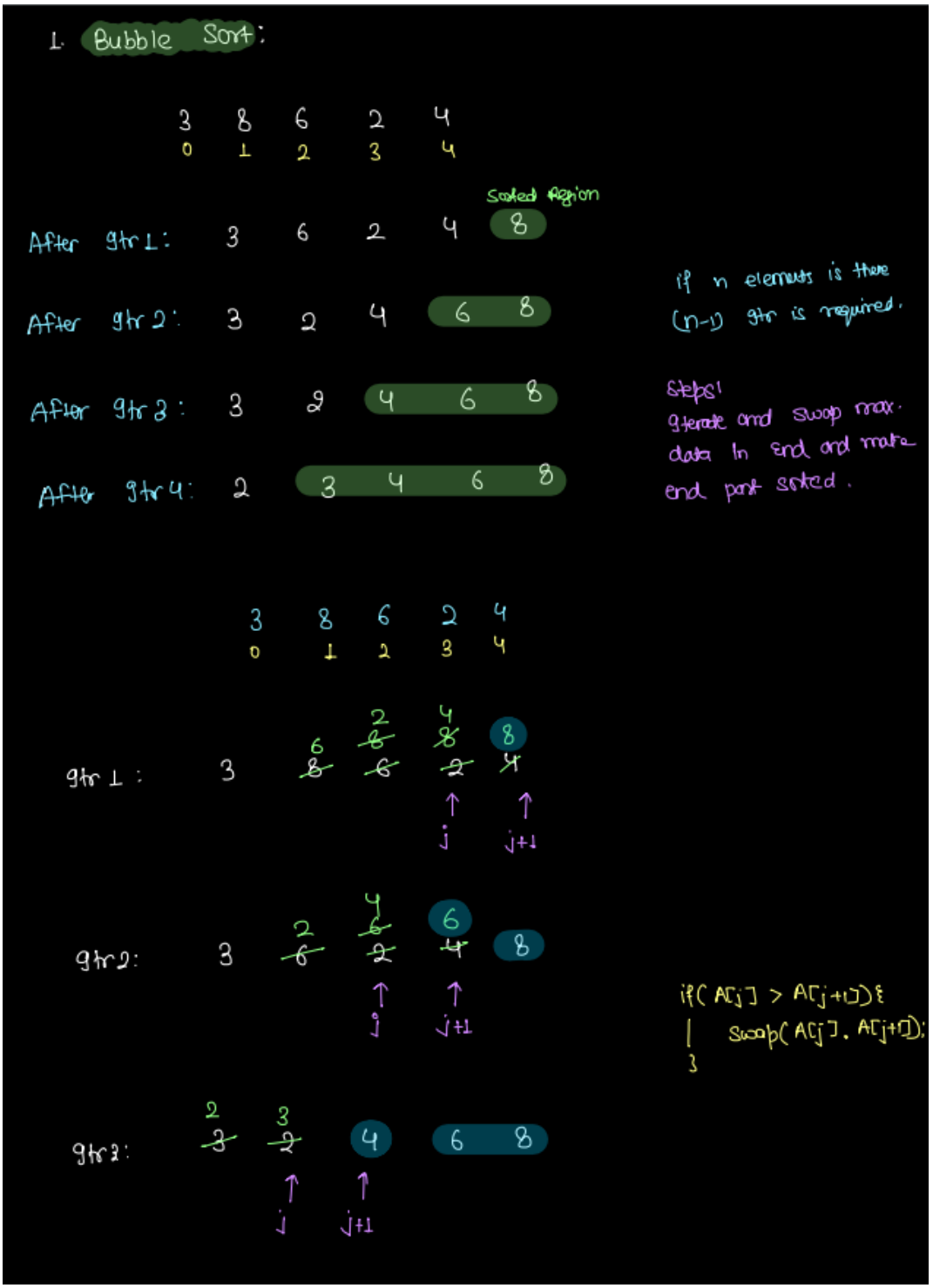
- **What is sorting**
- **Bubble Sort**
- **Insertion Sort**
- **Merge Two Sorted Array**
- **Merge Sort**

What is sorting -> Arranging data to some specific order is sorting. Arrays.sort(arr), Collections.sort(ArrayList) this is the in build function which has complexity O(NlogN).

Bubble Sort ->
In this method, we need to find the maximum element and place it at the end of the array. This process is called a "sorted region." In each step, we compare the current element with the next element and swap them if the current element is greater. We repeat this process for every element using two nested loops.
- First loop will run 0 to i < n - 1 // we are assuming first one is already sorted if we reach end
- Second loop run 0 to j < n - 1 - i

The time complexity of this method is O(n^2), where n is the number of elements in the array. In the best case scenario, when the array is already sorted, the time complexity is O(n) as there are no swaps needed.

Please note that this method is not the most efficient for sorting large arrays, but it can be useful for smaller arrays or educational purposes.



```
i = 0  to  i < n-1
    j = 0  to  j < n-1-i
        if ( A[j] > A[j+1] ) {
            swap( A[j], A[j+1] );
        }
```

SCALER

```java
// bubble sort algorithm
public static void bubbleSort(int[] A) {
    int n = A.length;
    for(int i = 0; i < n - 1; i++) {
        for(int j = 0; j < n - 1 - i; j++) {
            if(A[j] > A[j + 1]) {
                // swap the data of A[j] and A[j+1]
                int temp = A[j];
                A[j] = A[j + 1];
                A[j + 1] = temp;
            }
        }
    }
}
```

A: 8 1 1 3
0 1 2 3

i	j
0	0, 1, 2
1	0, 1
2	0

T.C.: $O(n^2)$

S.C: $O(1)$

1o 2o 3o 4o5o

Bubble Sort:

Best Case T.C. ⇒ Bubble sort not depend on order of data.

But if no. of swapping is 0
that means data is already sorted.
→ If data is already sorted stop the algorithm.

```java
// bubble sort algorithm
public static void bubbleSort(int[] A) {
    int n = A.length;
    for(int i = 0; i < n - 1; i++) {
        boolean swapping = false;
        for(int j = 0; j < n - 1 - i; j++) {
            if(A[j] > A[j + 1]) {
                // swap the data of A[j] and A[j+1]
                int temp = A[j];
                A[j] = A[j + 1];
                A[j + 1] = temp;
                // mark swapping as true for
                // indentification of step
                swapping = true;
            }
        }
        if(swapping == false) {
            return;
        }
    }
}
```

Best case: → data is sorted
T.C. → $O(n)$

Worst case → data is not sorted
T.C. → $O(n^2)$

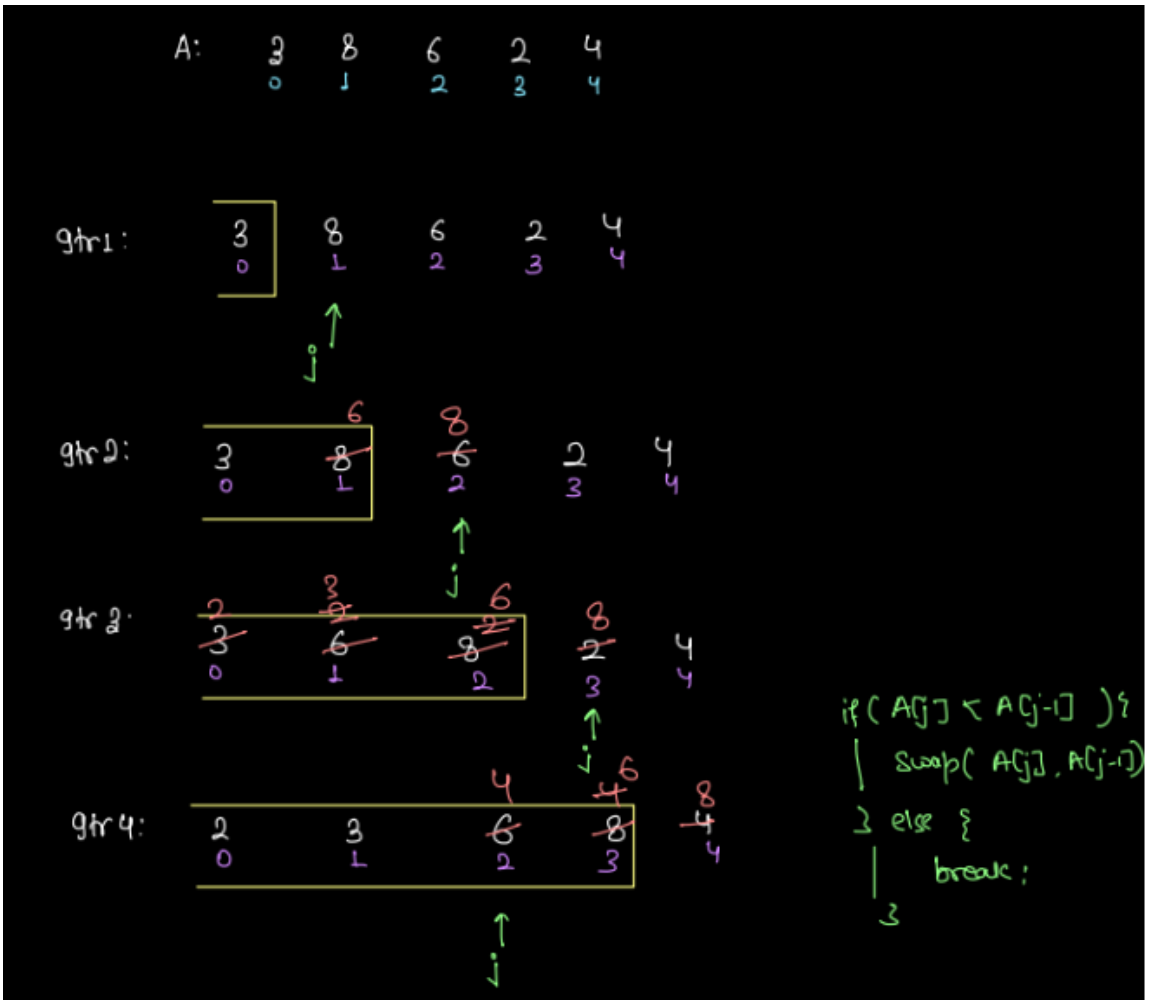
S.C → $O(1)$

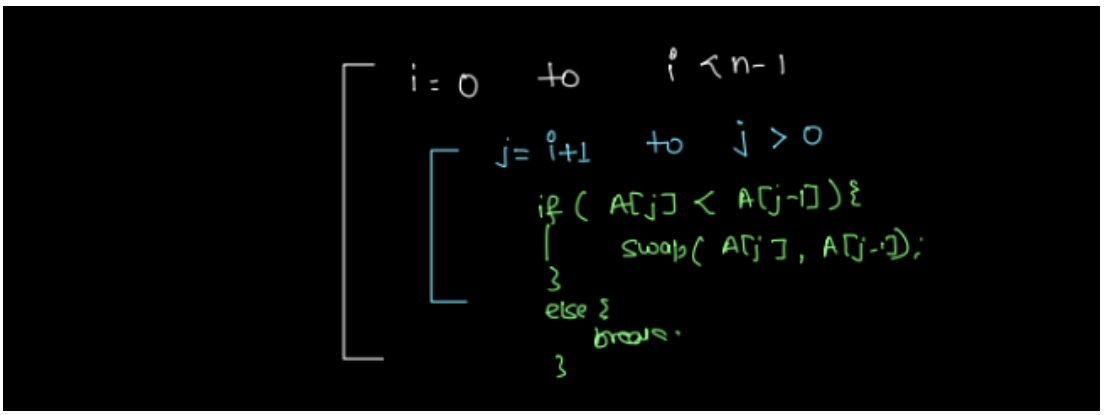
8:19 – 8:30 → Break Time

Insertion Sort -> In this algorithm, we start by assuming that the first element is already in its sorted position. We then iterate through the remaining elements, comparing each element with the previous one. If the previous element is greater, we swap them. This process continues until we reach the end of the array or until no more swaps are needed.

To implement this, we use two loops. The first loop starts from the beginning of the array and goes up to the second-to-last element (n - 1), as we assume the first element is already in place. The second loop starts from the element next to the current element (i + 1) and continues as long as j is greater than 0. Within the second loop, we compare the current element with its previous element, and if a swap is necessary, we perform it.

The algorithm breaks the inner loop if no swap is made in an iteration, indicating that the array is already sorted. This optimization helps improve the efficiency of the algorithm for partially sorted or already sorted arrays.
- First loop
i = 0; i < n - 1; i++
- Second loop
j = i + 1; j > 0; j –
This algorythem called insertation sort
It has TC = n^2






```java
// insertion sort
public static void insertionSort(int[] A) {
    int n = A.length;
    for(int i = 0; i < n - 1; i++) {
        for(int j = i+1; j > 0; j--) {
            if(A[j] < A[j-1]) {
                // swap curr i.e. A[j] with prev. i.e. A[j-1]
                int temp = A[j];
                A[j] = A[j - 1];
                A[j - 1] = temp;
            } else {
                // if curr is greater than prev.
                //that means curr ele is already placed at
                // correct position
                break;
            }
        }
    }
}
```

T.C: $O(n^2)$

S.C. $O(1)$

array : 7 2 4 6 (0 1 2 3)

i	j	state of array
0	1 → 0	2 7 4 6
1	2 → 1 (break)	2 4 7 6
2	3 → 2 (break)	2 4 6 7

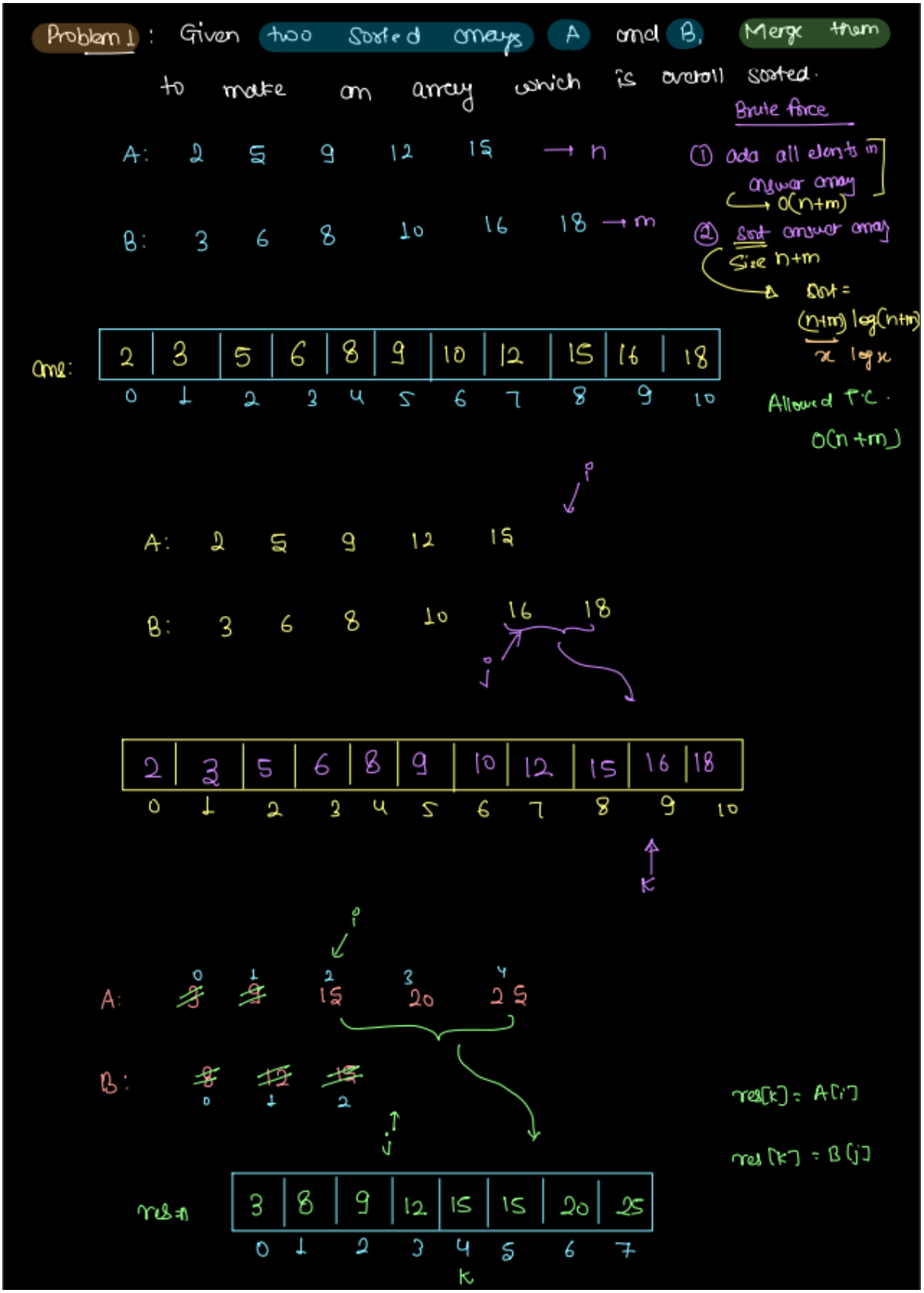
Eg. 10 20 30 40 50

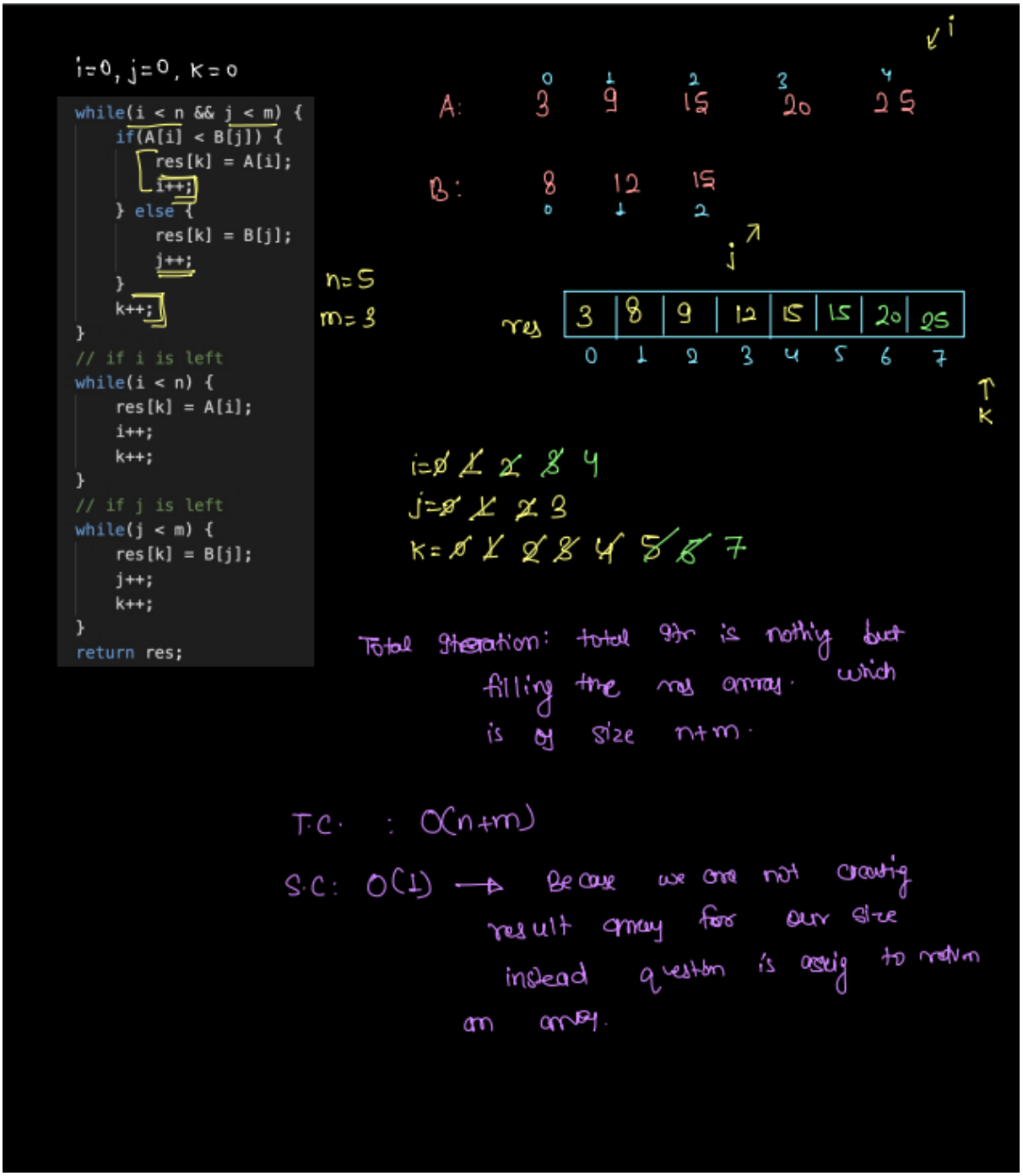
final array 2 4 6 7 Sorted.

i	j	state of array
0	1 → break	10 20 30 40 50
1	2 → break	10 20 30 40 50
2	3 → break	10 20 30 40 50
3	4 → break	10 20 30 40 50

T.C: O(n) → Best case in insertion sort i.e. → Array is already sorted.

Merge Two Sorted Array





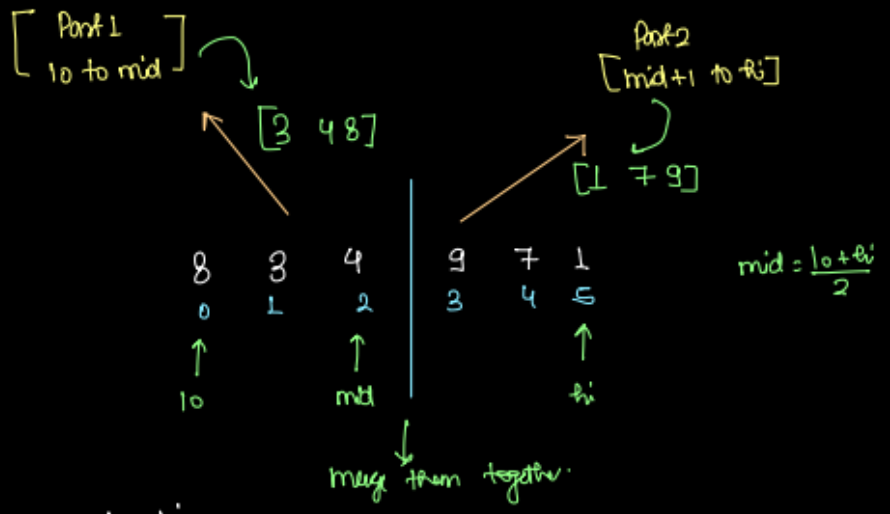
Merge Sort -> Divide and conquer algorithm (TC: NlogN)

This algorithm based on recursion. In this algorithm We will get the mide point using lo+hi/2 like if array has 1,2,3,4,5,6 then lo is 0 index and hi is 5th index and mid point is 2. now we will make it two half of array and then we will merge both the array till we will not reach to base case where lo == hi.

Merge Sort : T.C. O(nlogn)

divide and conquer algorithm

↳ this algorithm is based on Recursion.



Assumption → arr, lo, hi

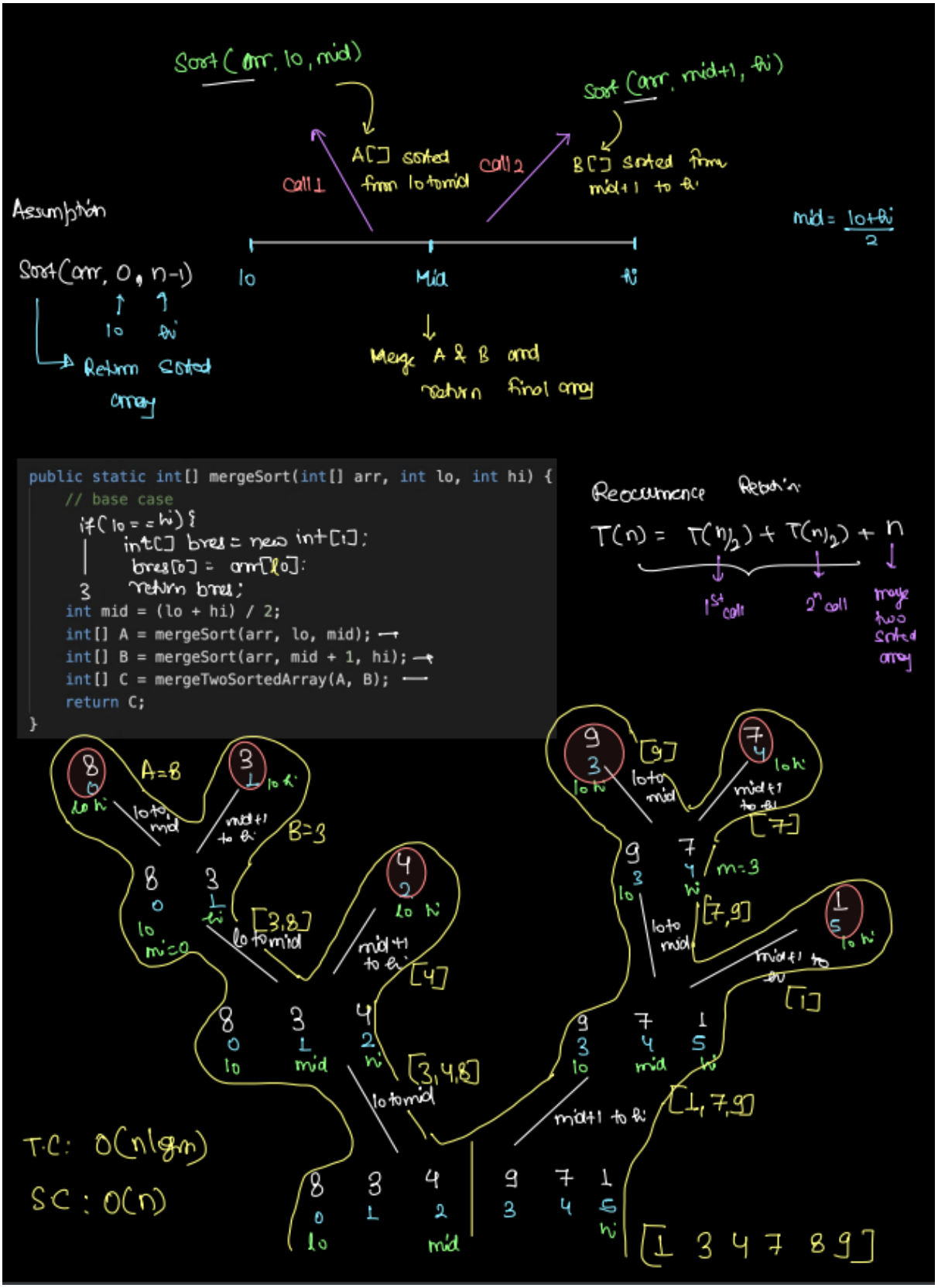
Sort array from lo to hi

Main logic: →

Sort (arr, lo, mid) → A

Sort (arr, mid+1, hi) → B

My task → Merge them together.

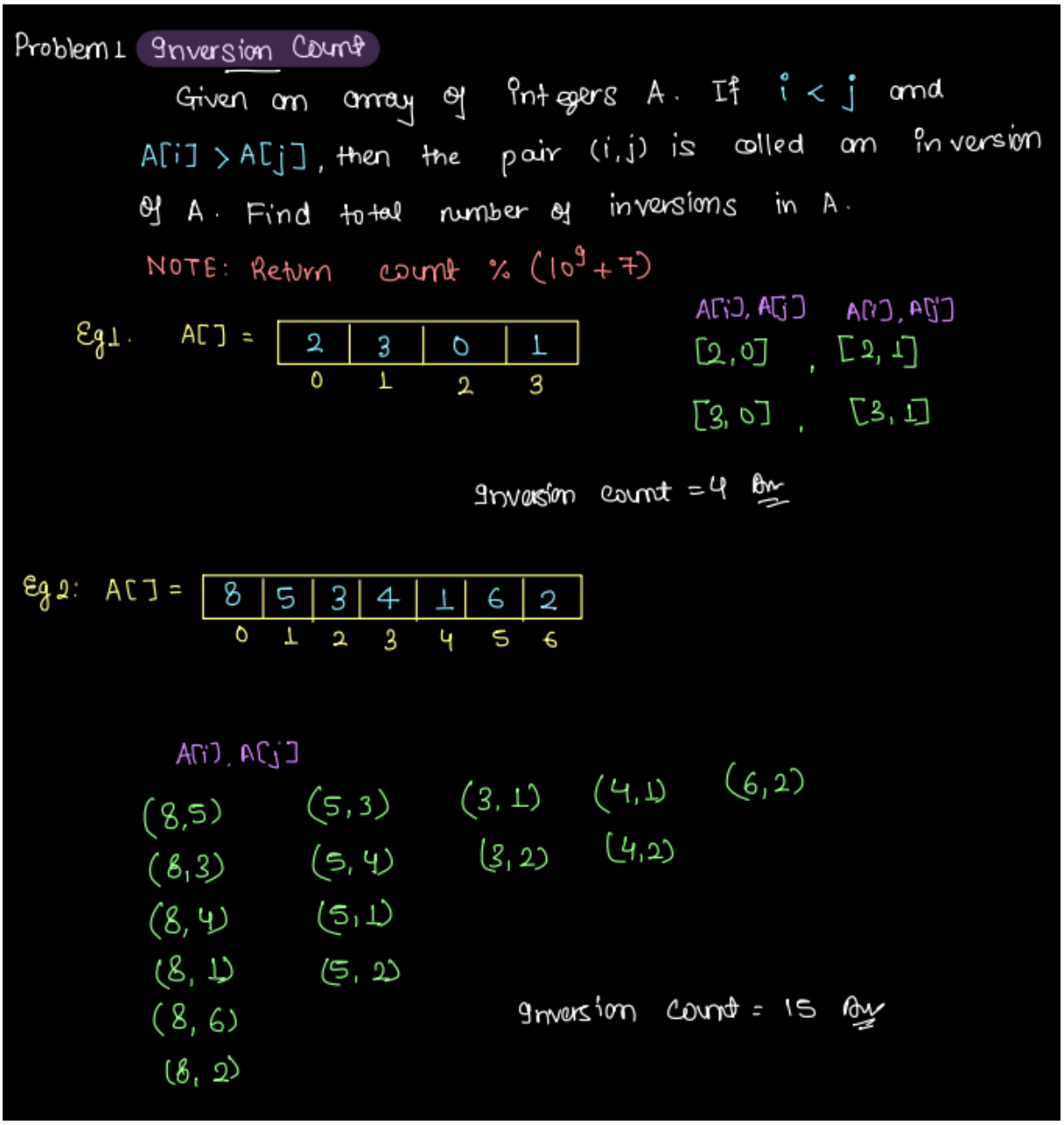


DSA: Sorting 2 - 8 - June 2023

- **Inversion Count**
- **Custom Comparison**

- **Factor Sort**
- **Sort Coordinates**
- **Largest Number**

Inversion Count -> if A[i] > A[j] then all pairs with A[i] called inversions like below. It will solved using recursion shorting function like merge short where we will divid array in 2 parts and then call same function using mergeSort(A, lo, mid), merge Sort (A, mid+1, hi) and mid will calculate n+m/2



Brute force Idea:

for value of A[i], we will count value of A[j] in which A[i] > A[j] & j = i+1 to n-1.

```
int inversionCount(int[] A) {
    int n = A.length;
    long count = 0;
    for(int i=0; i< n; i++) {
        for(int j=i+1; j<n; j++) {
            if( A[i] > A[j] ){
                count = (count + 1)% mod
            }
        }
    }
    return (int)(count);
}
```

T.C : $O(n^2)$

S.C : $O(1)$

Time complexity

i=0 , j=1 to n-1 ⟶ n-1

i=1 , j=2 to n-1 ⟶ n-2

i=2 , j=3 to n-1 ⟶ n-3

⋮

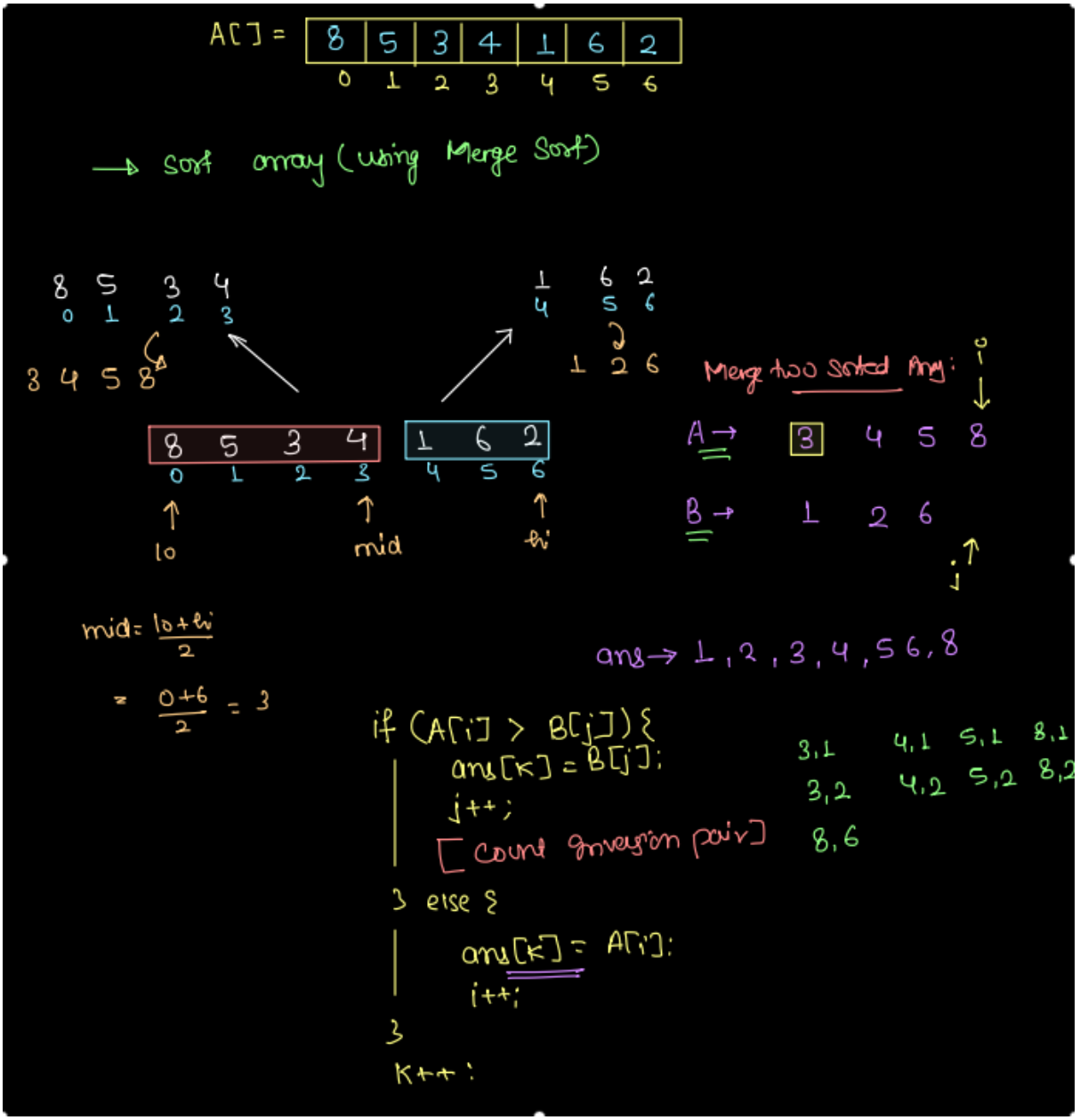
i=n-2 , j=n-1 to n-1 ⟶ 1

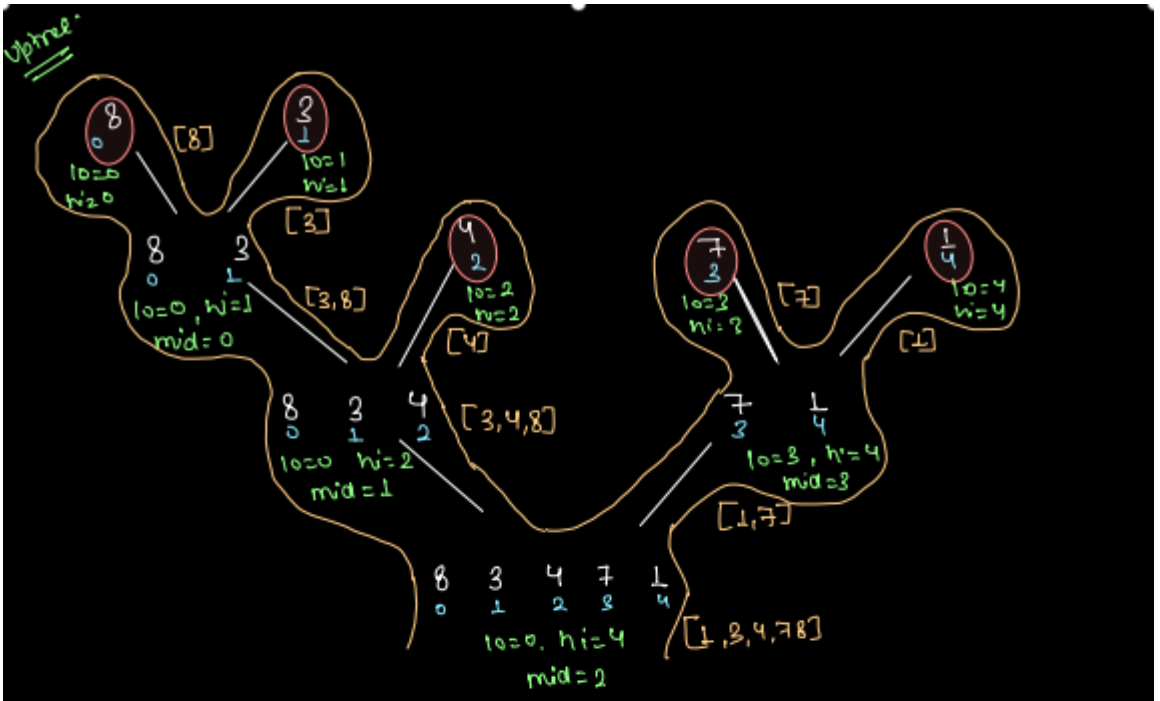
Total str = $1+2+3+\cdots+n-2+n-1$

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

T.C = $O(n^2)$

Expected T.C : $O(n \log n)$


A[] = 8 5 3 4 1 6 2
0 1 2 3 4 5 6
→ Sort array (using Merge Sort)
8 5 3 4
0 1 2 3
3 4 5 8
1 6 2
4 5 6
1 2 6
8 5 3 4 | 1 6 2
0 1 2 3 4 5 6
lo
mid
hi
Merge two Sorted Arr:
A → 3 4 5 8
B → 1 2 6
i
j
mid = lo+hi / 2
= 0+6 / 2 = 3
ans → 1, 2, 3, 4, 5 6, 8
if (A[i] > B[j]) {
ans[k] = B[j];
j++;
[Count inversion pair]
} else {
ans[k] = A[i];
i++;
}
k++;
3,1 4,1 5,1 8,1
3,2 4,2 5,2 8,2
8,6



A[] → 3 4 8 ↙ i

B[] → 1 7 ↑ j

C[] → 1, 3, 4, 7, 8

A[i] > B[j]

Count += (n-i)

count = 0 + 1 + 1 + 1 + 3 + 1

= 7 Ans

(8,3), (8,4), (7,1)

(3,1) (4,1) (8,1)

(8,7)

We are just adding one single line in merge two sorted array

8:32 – 8:45

Break time

→ Rest Every code is same

T.C : O(nlogn)

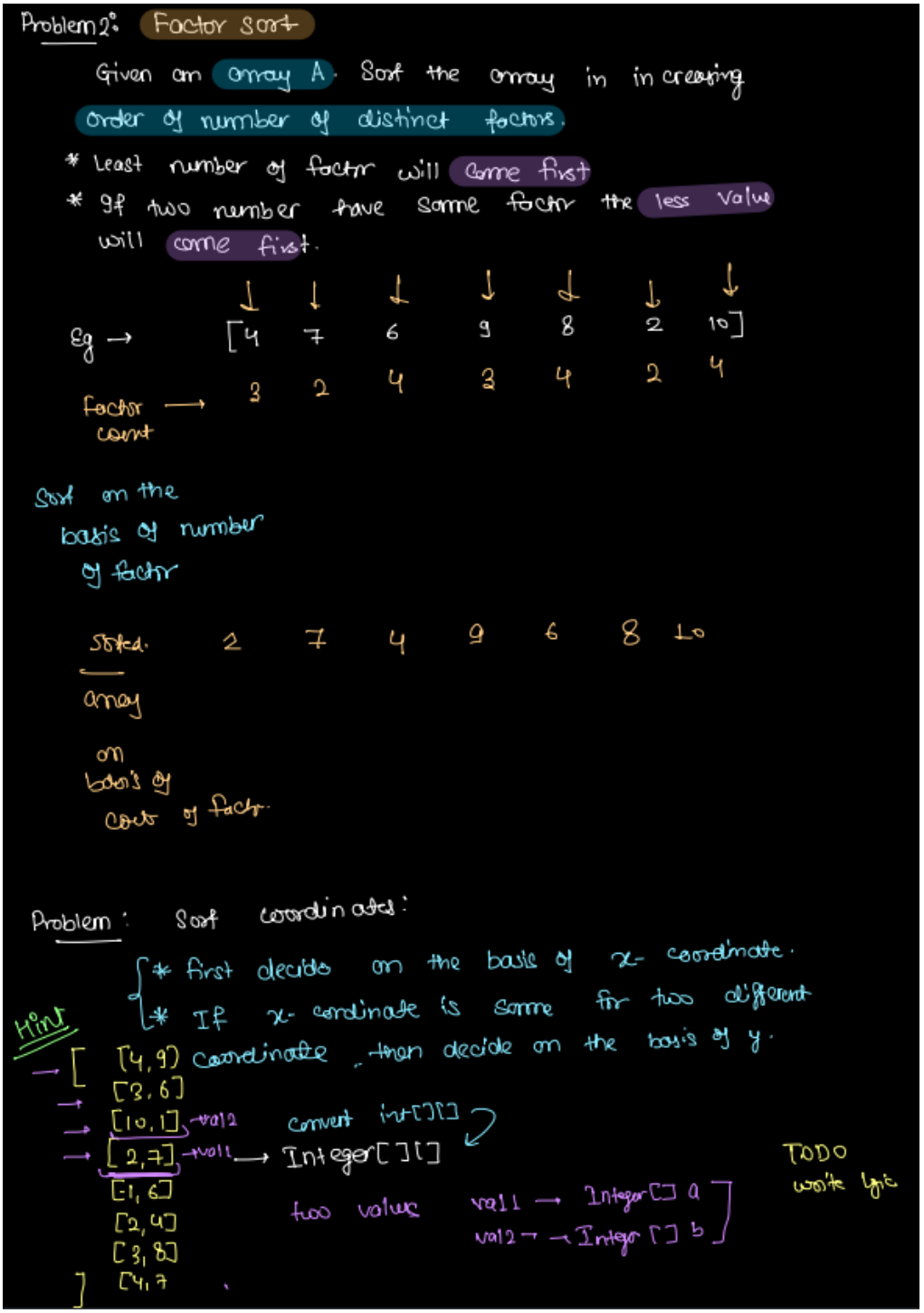
S.C. : O(n)

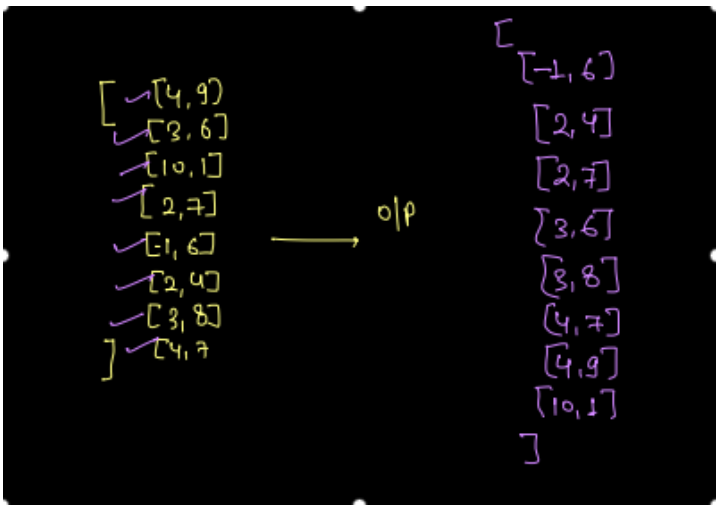
Custom Comparison -> We will write custom short function for custom comparison like if we want to sort car by price or by feature then we have to write it because default sort feature only provide chat, int sorting. for custom sorting function we will use sort method and pass second argument as comparator function like below.
We will use Interger class reference type not primitive type int and array also we have to convert in int to Integer. We will use @Override it is optional. If we want to return first parameter

```
Arrays.sort(arr, new Comparator<Integer>() {
@Override
public int compare(Integer a, Integer b) {
// Possibility 1 : want to keep a before b
// return negative value
// Possibility 2 : want to keep b before a
// return positive value
// Possibility 3 : Doesnot matter
// return 0

// noraml Order
/*
if(a < b) {
// want to keep a before b
return -1;
} else if(a > b) {
// want to keep b before a
return 1;
} else {
// Doesnot matter
return 0;
}
*/
// reverse the order
if(a < b) {
// want to keep b before a
return 1;
} else if(a > b) {
// want to keep a before b
return -1;
} else {
// Doesnot matter
return 0;
}
```

```
// sorter only use this instead above
return a - b;
// for order reversal
// return b - a;
}
});
```

Factor Sort

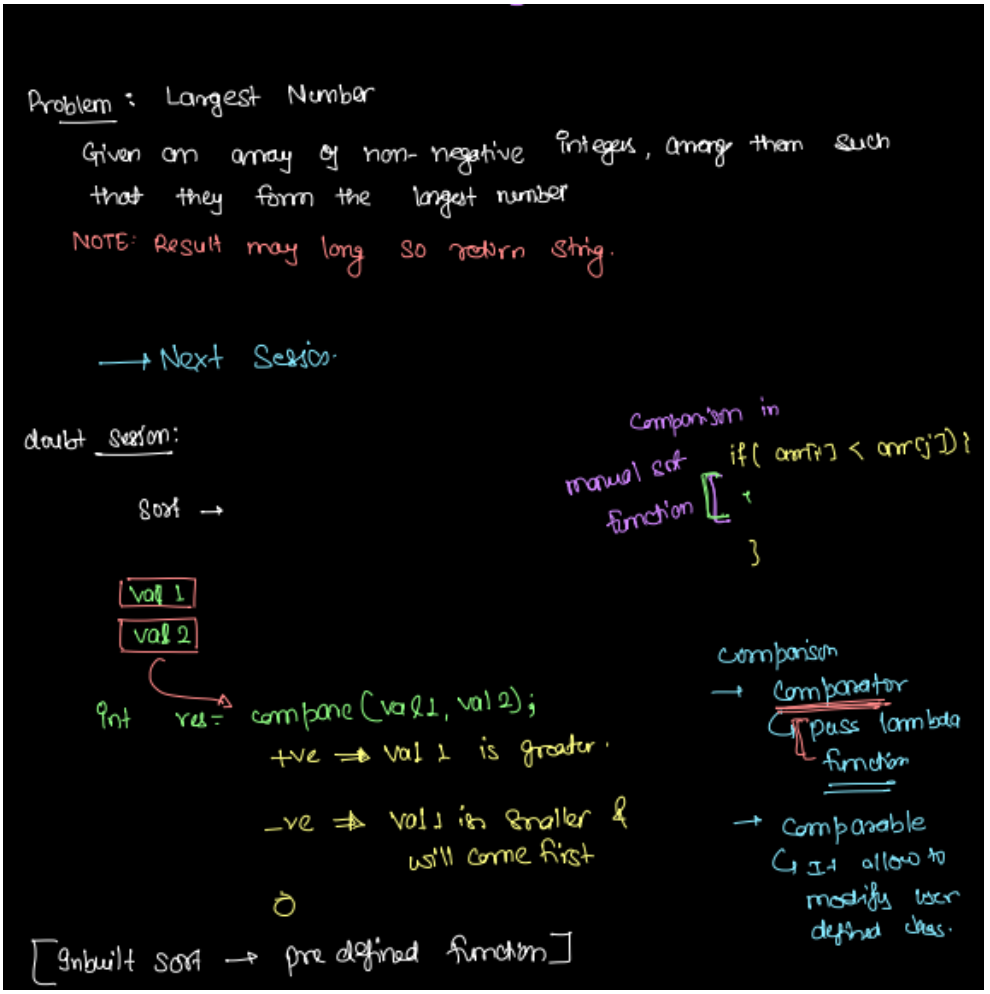





```
class Main {
/* Problem : factor Sort */
public static int factorCount(int val) {
// todo : write optimise code
int fcount = 0;
for(int i = 1; i <= val; i++) {
if(val % i == 0) {
fcount++;
}}
return fcount;
}
public static int[] factorSort(int[] arr) {
int n = arr.length;
// convert int array into Integer array
Integer[] A = new Integer[n];
for(int i = 0; i < n; i++) {
A[i] = arr[i];
}
Arrays.sort(A, new Comparator<Integer>() {
@Override
public int compare(Integer a, Integer b) {
int fa = factorCount(a);
int fb = factorCount(b);
if(fa < fb) {
// a come first
return -1;
} else if(fa > fb) {
// b compe first
return 1;
} else {
```

```
// if both are equal
// decide according to value of a and b
return a - b;
}}
});
// fill arr from A
for(int i = 0; i < n; i++) {
arr[i] = A[i];
}
return arr;
}
public static void main(String args[]) {
// Your code goes here
int[] arr = {4, 7, 6, 9, 8, 2, 10};
System.out.println(Arrays.toString(arr));
arr = factorSort(arr);
System.out.println(Arrays.toString(arr));
}}
```

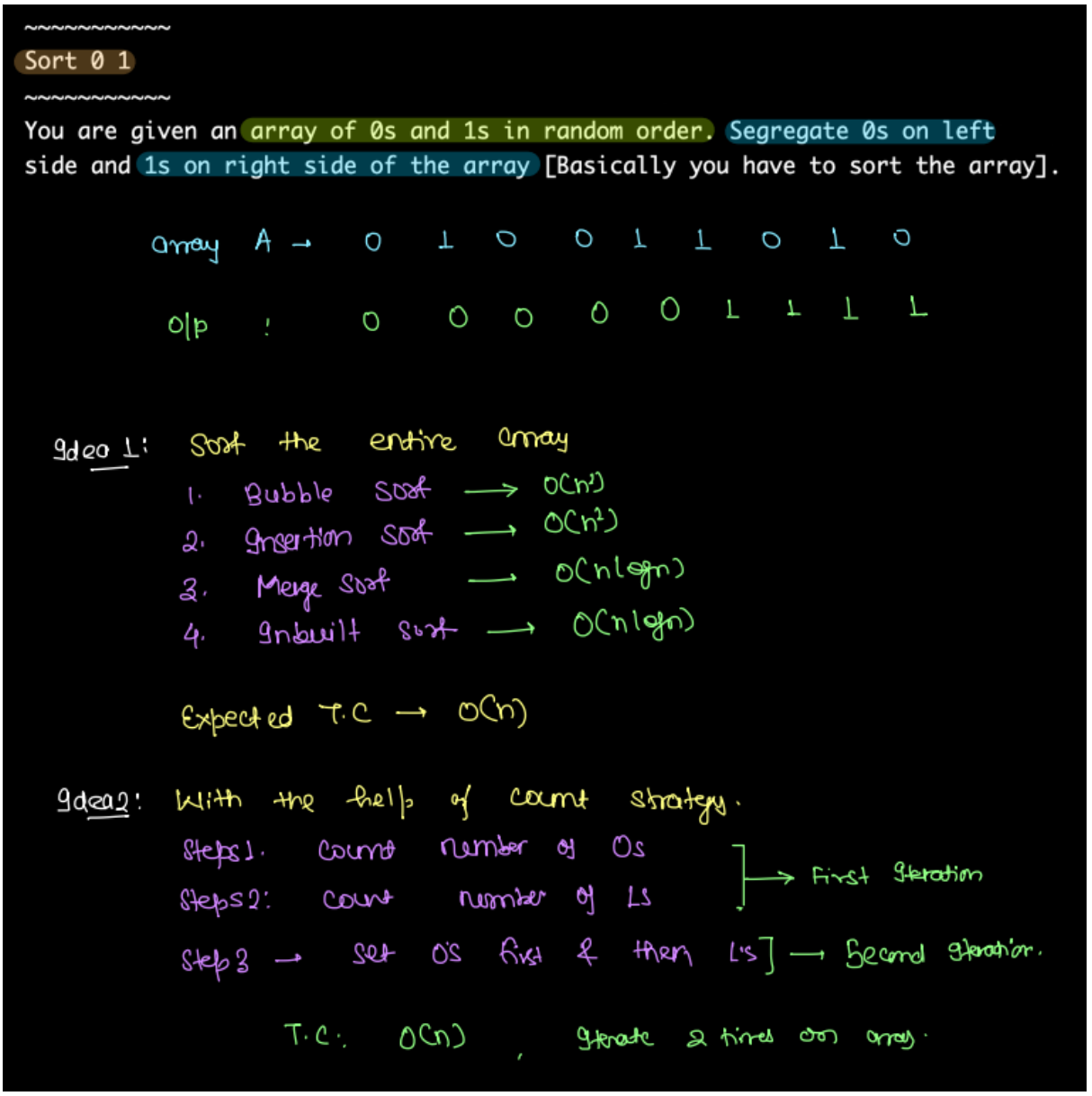
Largest Number

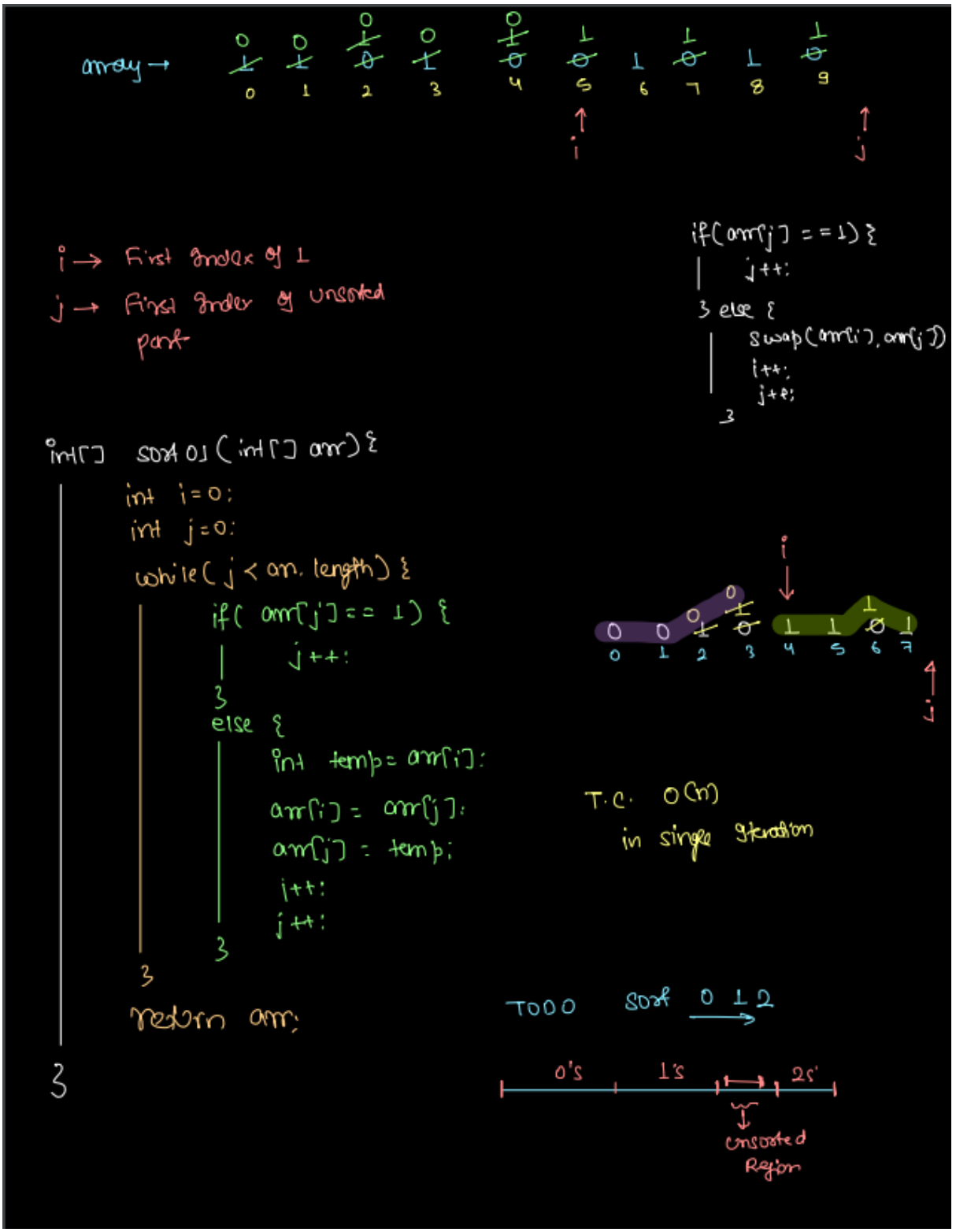


DSA: Sorting 3 - 10 - June 2023

- **Sort 0 1**
- **Partition of Array**
- **Quick Sort Algorithm**
- **TC and SC of quickSort**
- **Worst case of quickSort**
- **Largest Number**
- **Sort Coordinates**

Sort 0 1 -> sort all 0's first then 1's after. We can use any sorting algorithm but we have to sort this in O(n). we will use Aproch swap in single loop. We will define 2 initial variable i and j to 0 and then we will apply condition if j == 1 then j++ go to next element and if j !== 1 then we will swap in same array j to i and i to j.





i → First index of 1

j → First index of unsorted part

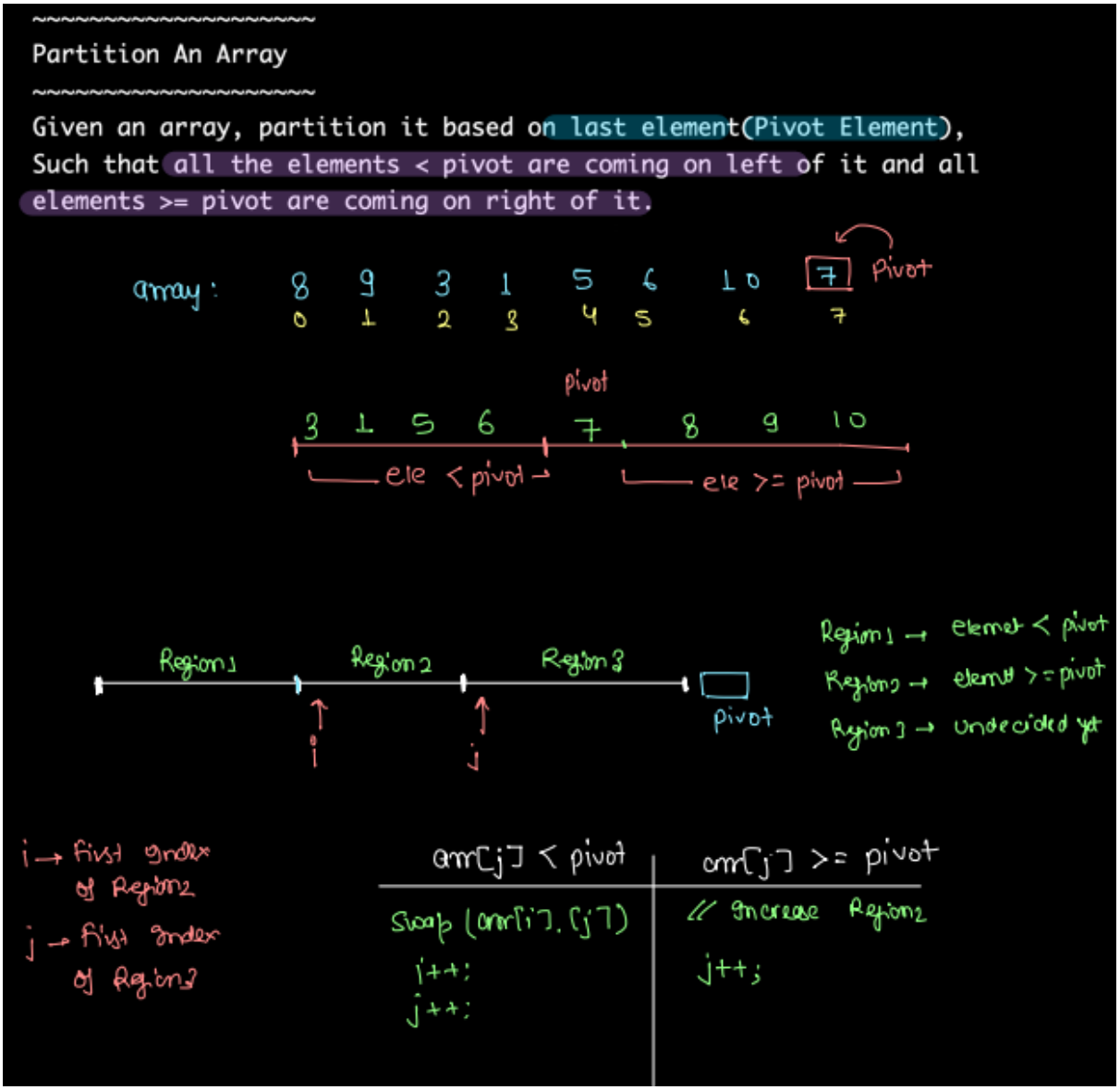
```
if(arr[j] == 1) {
    j++;
} else {
    swap(arr[i], arr[j])
    i++;
    j++;
}
```

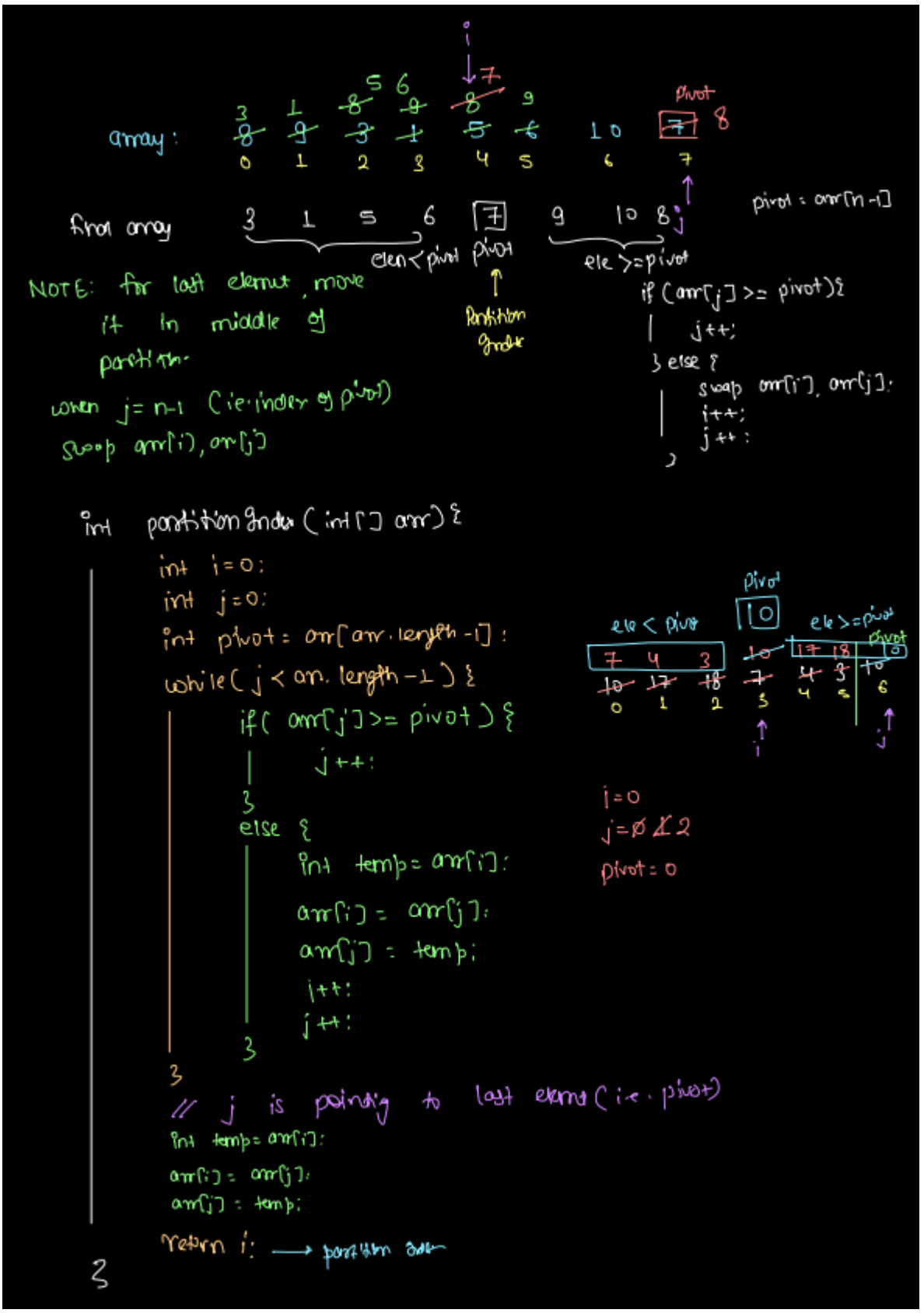
```
int[] sortOI(int[] arr) {
    int i=0;
    int j=0;
    while(j < arr.length) {
        if(arr[j] == 1) {
            j++;
        }
        else {
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
            i++;
            j++;
        }
    }
    return arr;
}
```


T.C. O(n)
in single iteration

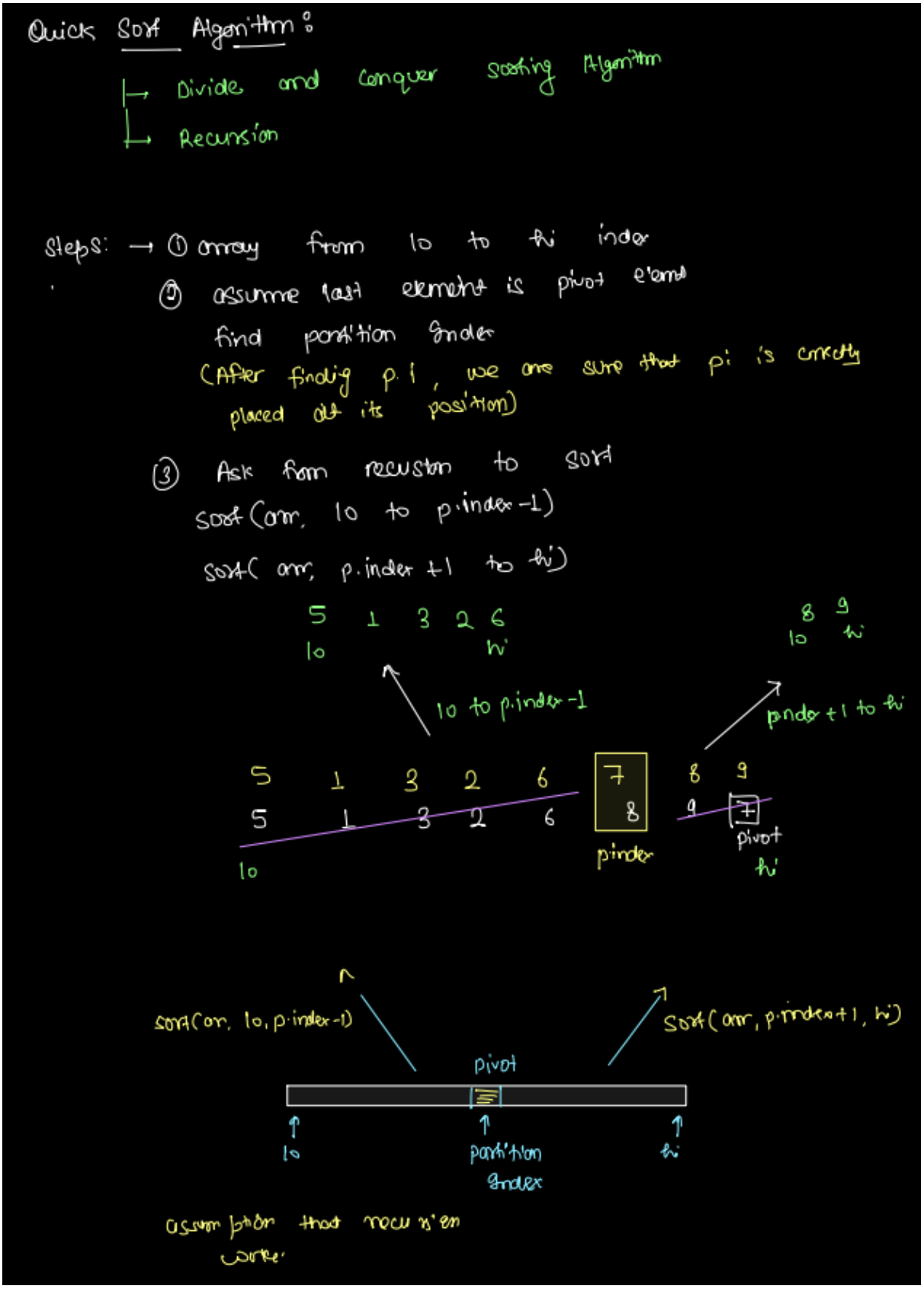
TODO Sort 0 1 2 →

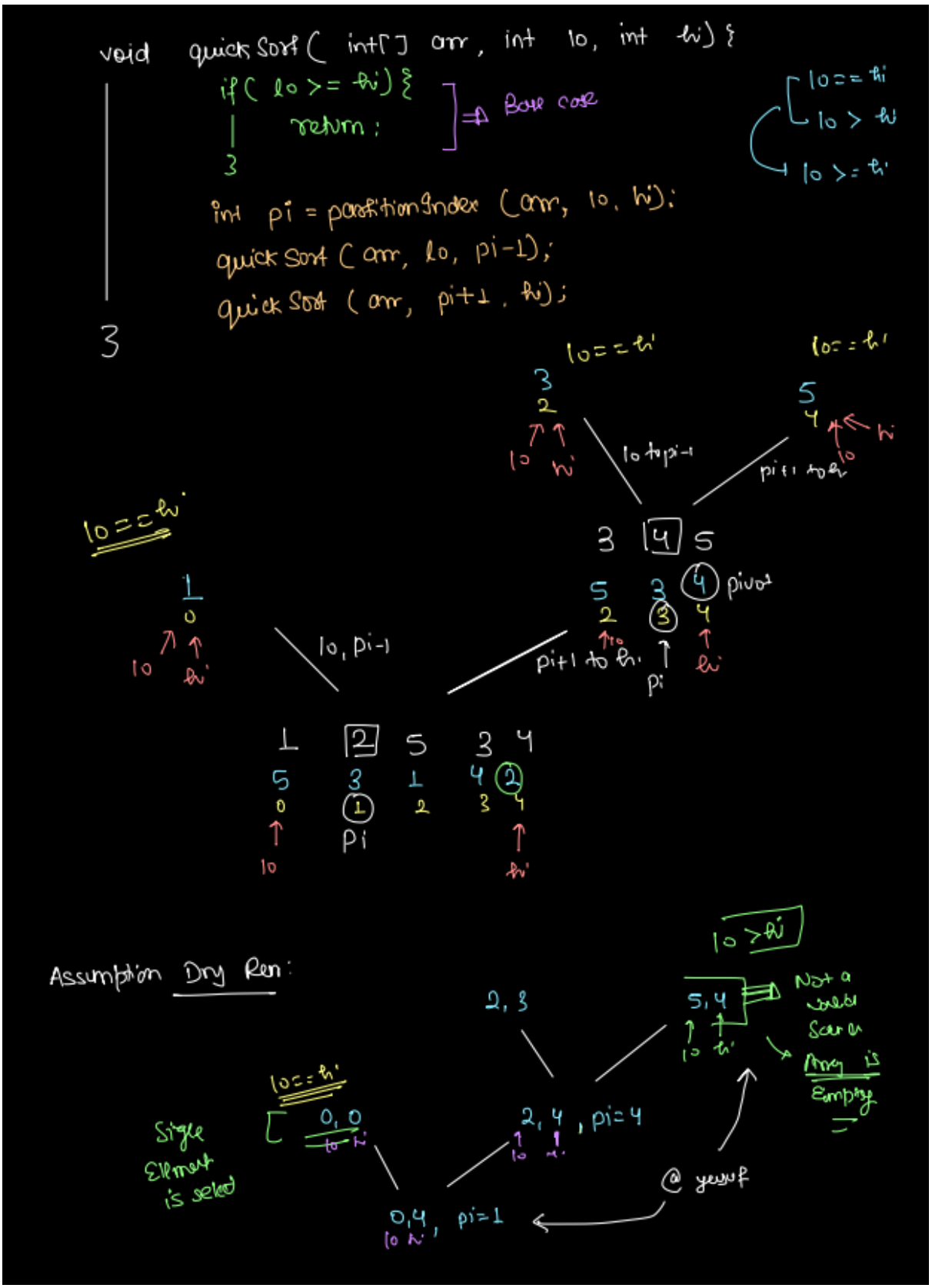
Partition of Array




array :
pivot
final array
pivot = arr[n-1]
ele < pivot
pivot
Partition index
ele >= pivot
NOTE: for last element, move it in middle of partition.
Given j = n-1 (ie. index of pivot)
Swap arr[i], arr[j]
if (arr[j] >= pivot){
j++;
} else {
swap arr[i], arr[j];
i++;
j++;
}
int partitionIndex (int[] arr) {
int i = 0;
int j = 0;
int pivot = arr[arr.length -1];
while(j < arr.length -1) {
if(arr[j] >= pivot) {
j++;
}
else {
int temp = arr[i];
arr[i] = arr[j];
arr[j] = temp;
i++;
j++;
}
}
// j is pointing to last element (i.e. pivot)
int temp = arr[i];
arr[i] = arr[j];
arr[j] = temp;
return i; → partition index
}
ele < pivot
pivot
ele >= pivot
pivot
i
j
i = 0
j = 0 & 2
pivot = 0

Quick Sort Algorithm



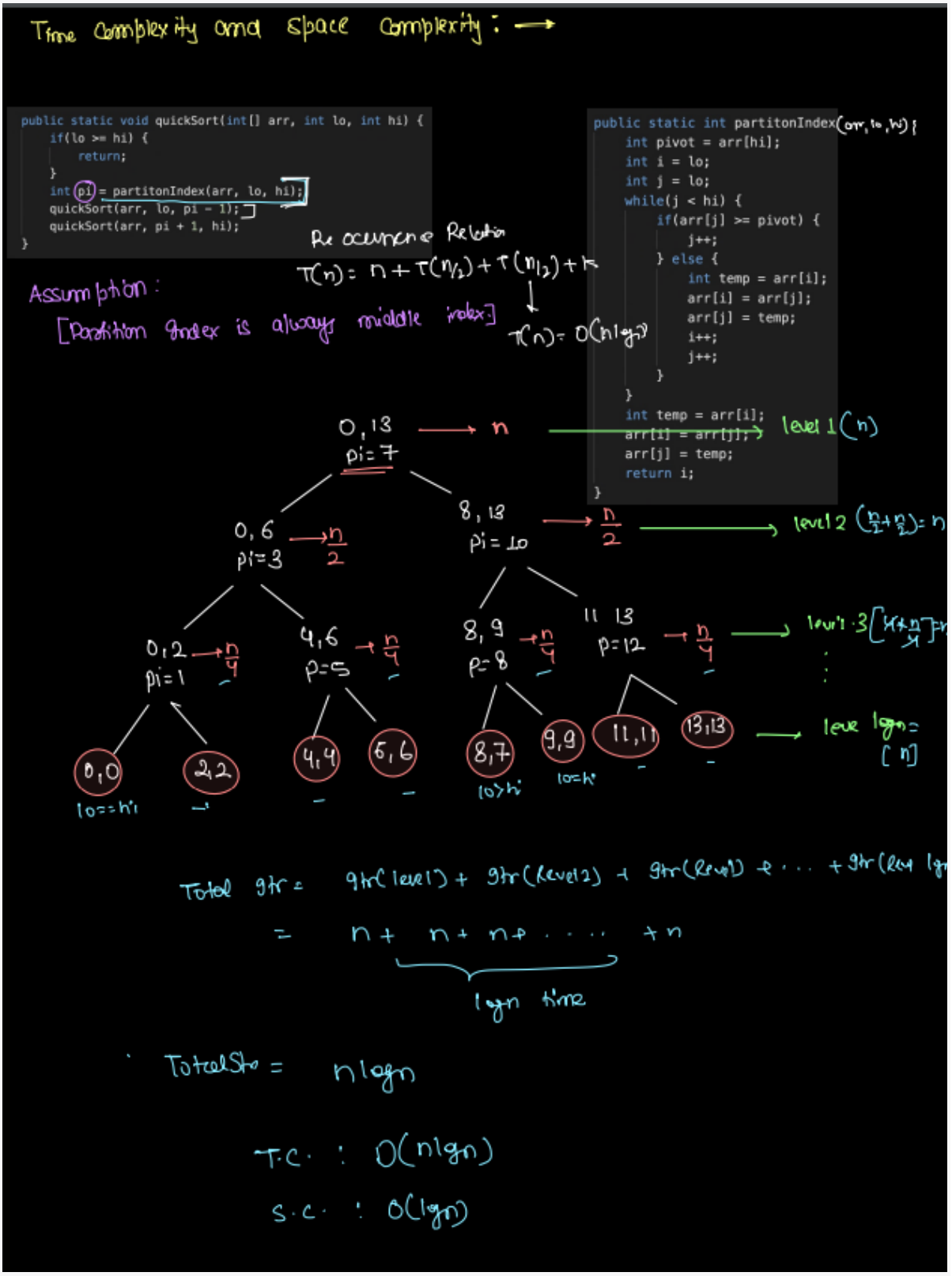


In this particular scenario, the initial array is [0, 1, 2, 3, 4], which has been sorted. Now, we are examining the second condition. The first base condition is when the "lo" index is equal to the "hi" index. However, upon further consideration of different scenarios, including the one

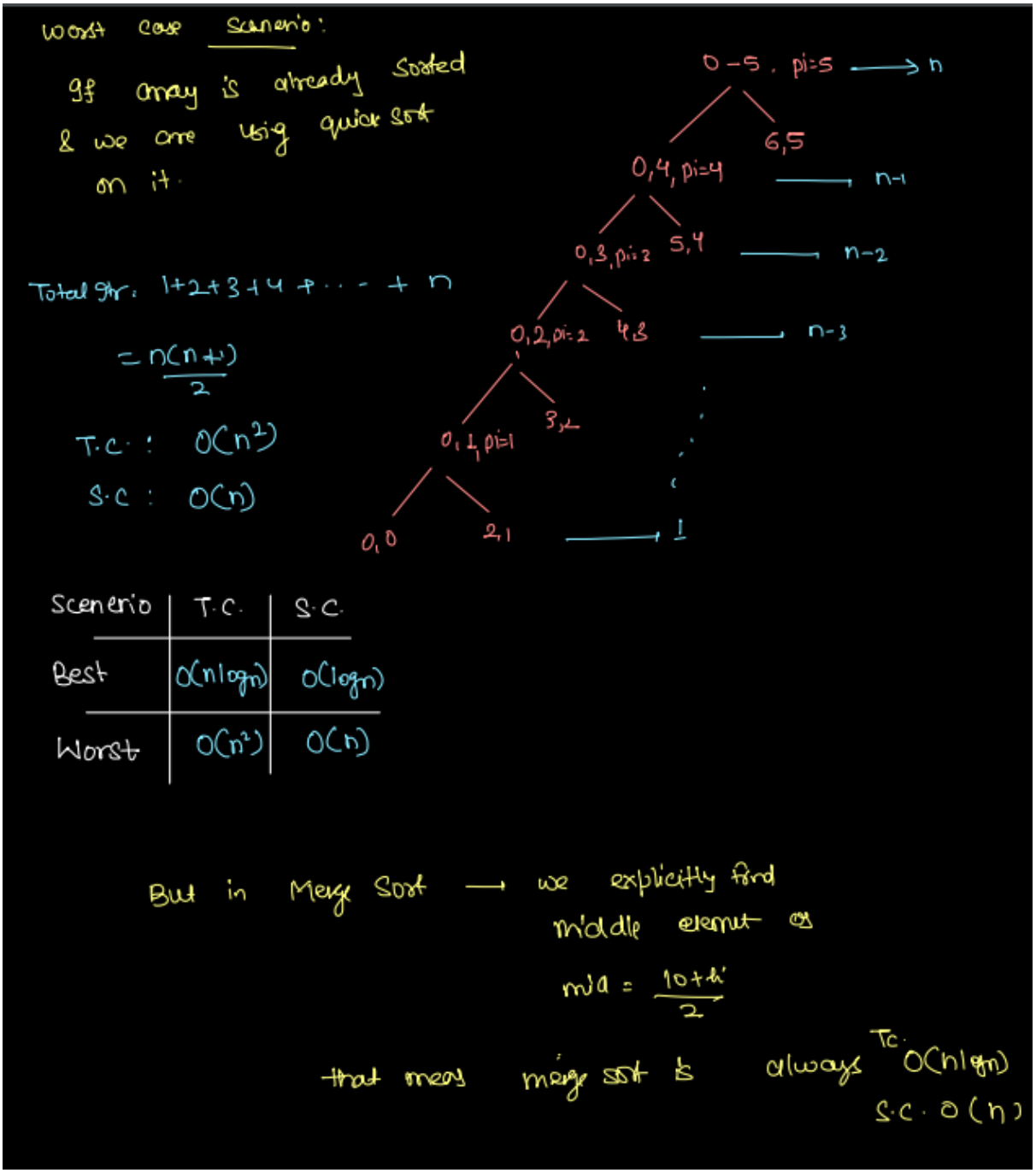
mentioned earlier, we observe that when we encounter a pivot (in this case, 4), we increment the pivot value by 1 and compare it to the elements in the array. Since there is no element equivalent to "pivot + 1" (i.e., 5) in the array, we can conclude that if we don't have an element greater than our pivot element, we should return from the function. This leads to the introduction of a new condition: "lo > hi". Consequently, we combine both conditions and establish a new base case: "lo >= hi".

```java
public class Solution {
    public static int partitionIndex(int[] arr, int lo, int hi){
        int pivot = arr[hi];
        int i = lo;
        int j = lo;

        while(j < hi){
            if(arr[j] >= pivot){
                j++;
            }else{
                int temp = arr[i];
                    arr[i] = arr[j];
                    arr[j] = temp;
            }
        }

        int temp = arr[i];
                arr[i] = arr[j];
                    arr[j] = temp;

        return i;

    }
    public static void quickSort(int[] arr, int lo, int hi){
        if(lo >= hi){
            return ;
        }

        int pi = partitionIndex(arr, lo, hi);
        quickSort(arr, lo, pi-1);
        quickSort(arr, pi + 1, hi);

    }
    public int[] solve(int[] A) {
        quickSort(A, 0, A.length-1);
        return A;
    }
}
```

TC and SC of quickSort



Worst case of quickSort



Largest Number

long.parseLong('str') is a function which convert string to long. Why it required because we can not compire to string directly then we have to convert it in long.

String.valueOf -> It is used for convert int to String.

left over:

Problem : Largest Number

Given an array of non-negative integers, arrange them such that they form the largest number

NOTE: Result may long so return String.

arr[] → {2 3 9 0} → {9 3 2 0} → 9320

Sort it in decr. order - ? ✗ Not work

arr[] → {99 90 98} → {99 98 90} ⇒ 999890

{99 90 98} ⇒ 999098

arr[] → {998, 9} → {998, 9} ⇒ 9989

{9, 9983} → 9998

Sort on the basis of digits → lexicographical order. → Not work

arr[] → {30, 3} → {30, 3} → 303

{3, 30} → 330

arr[] {34, 3} → {34, 3} → 343

{3, 34} → 334

Conclusion:

① Sorting on decreasing order will not work

② Sorting on digit will also not work

num1 → a

num2 → b

possibility ⇒ → ab

└ ba

ab > ba → a first in array

ab < ba → b come first

Steps → ① Convert Integer array into String array.

② Arrays.sort (str_Array, new Comparator<String>(){

```
public int compare ( String a , String b ) {
    String n1 = a+b;
    String n2 = b+a;
    Long val1 = Long.parseLong(n1);
    Long val2 = Long.parseLong(n2);
    if (val1 > val2 {
        // ab is best → a comes first
        return -1;
    } else if (val1 < val2) {
        // ba is best → b come first
        return 1;
    } else {
        return 0;
    }
}
```

Coordinates

```
import java.util.*;

class Main {
public static void sortCordinate() {
int[][] arr = {
{1, 1},
{4, 9},
{-1, 5},
{4, 4},
{3, 0},
{4, 1},
{3, 0}
};
// convert it into Integer
Integer[][] A = new Integer[arr.length][2];
for(int i = 0; i < arr.length; i++) {
A[i][0] = arr[i][0];
A[i][1] = arr[i][1];
}
for(Integer[] ele : A) {
System.out.println(ele[0] + " " + ele[1]);
}
// sort
Arrays.sort(A, new Comparator<Integer[]>() {
public int compare(Integer[] a, Integer[] b) {
int x1 = a[0];
int y1 = a[1];
int x2 = b[0];
int y2 = b[1];
// first compare on the basis of x
if(x1 != x2) {
return x1 - x2;
}
// if they are same compare on the basis of y
return y1 - y2;
}
});
System.out.println("~~~~~~~");
for(Integer[] ele : A) {
System.out.println(ele[0] + " " + ele[1]);
}
```

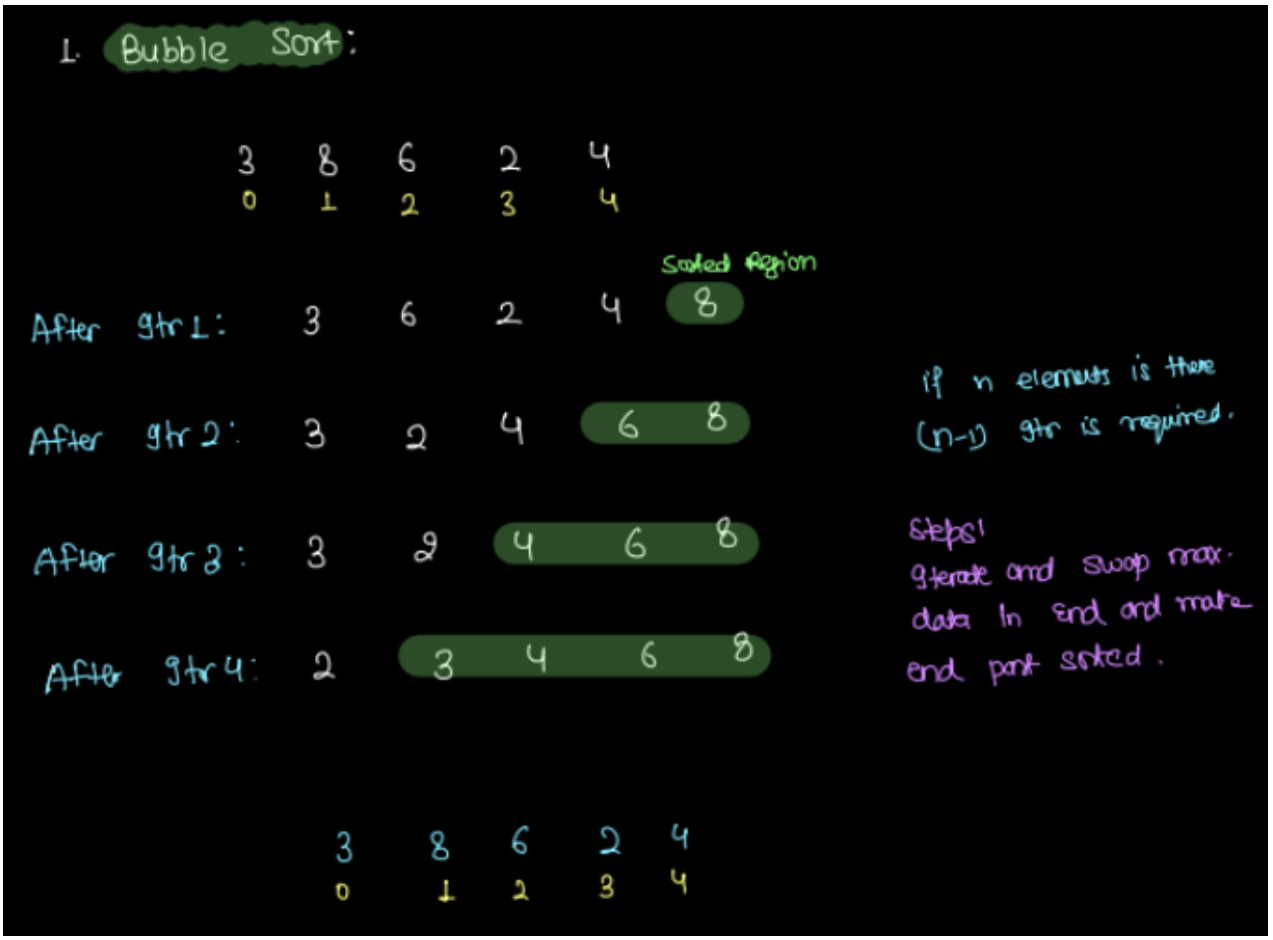
```
}
public static void main(String args[]) {
sortCordinate();
}
}
```

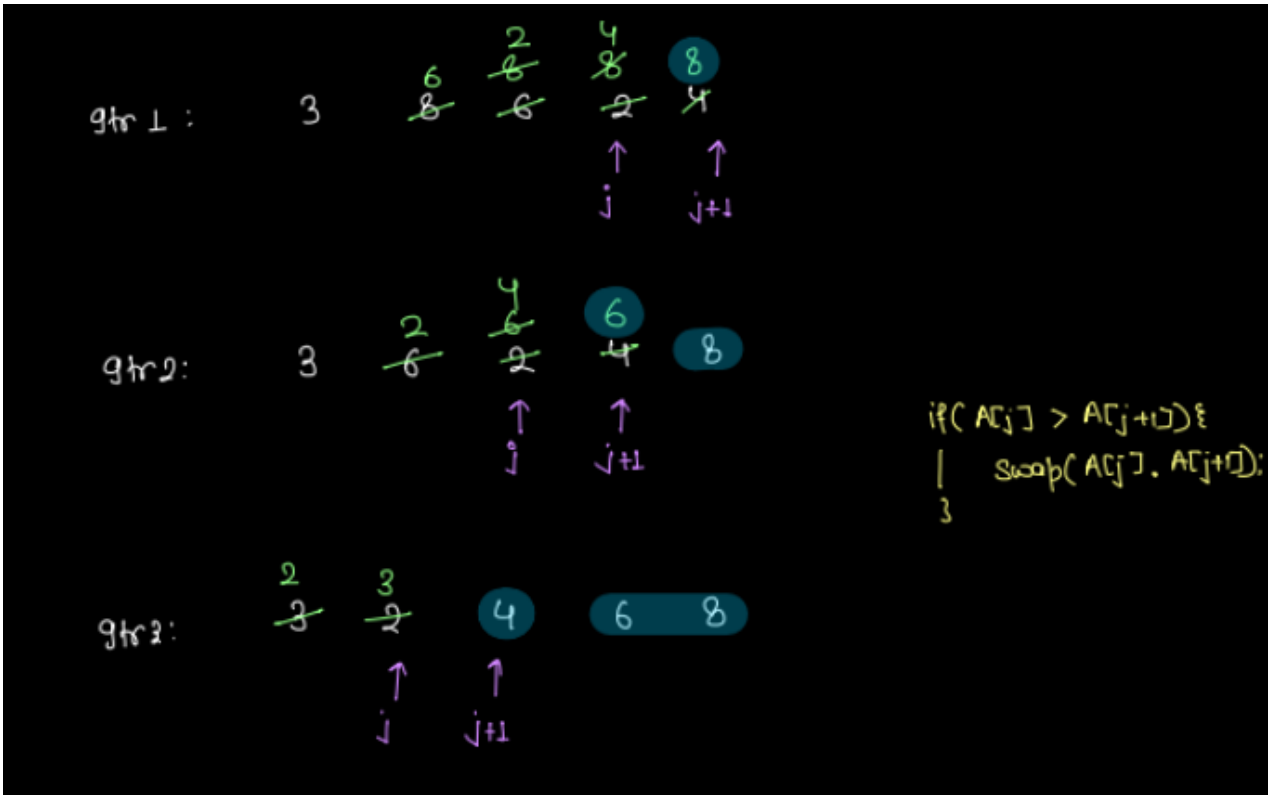
DSA: Sorting 1 - 6 - June 2023

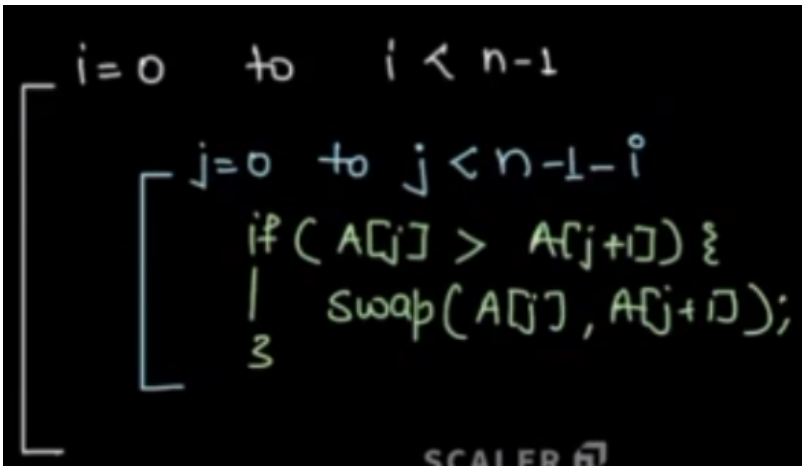
- **What is sorting**
- **Bubble Sort**
- **Insertion Sort**
- **Merge Two Sorted Array**
- **Merge Sort**

What is sorting -> Arranging data to some specific order is sorting. Arrays.sort(arr), Collections.sort(ArrayList) this is the in build function which has complexity O(NlogN).

Bubble Sort ->
In this method we have to check the max and set it to end. Set end part called sorted reason then on every steps we will check if current > next element then we will swap it. We will use 2 loop to inrate and check every element. It has time complexity O(n^2) and best time complexity is o(n) if no. of swapping 0. It means it is already ordered.



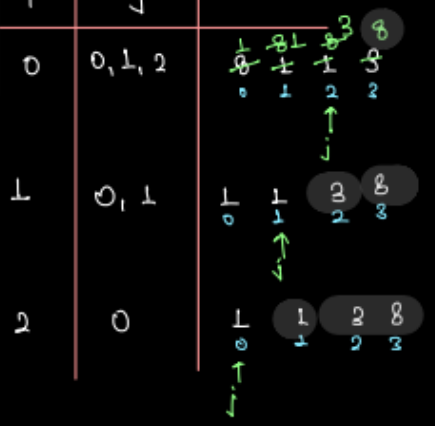

itr 1 :
3
6
2
4
8
i
j+1
itr 2:
3
2
4
6
8
itr 3:
2
3
4
6
8
if(A[j] > A[j+1]){
swap(A[j], A[j+1]);
}


i=0 to i < n-1
j=0 to j < n-1-i
if (A[j] > A[j+1]) {
swap(A[j], A[j+1]);
}
SCALER

```java
// bubble sort algorithm
public static void bubbleSort(int[] A) {
    int n = A.length;
    for(int i = 0; i < n - 1; i++) {
        for(int j = 0; j < n - 1 - i; j++) {
            if(A[j] > A[j + 1]) {
                // swap the data of A[j] and A[j+1]
                int temp = A[j];
                A[j] = A[j + 1];
                A[j + 1] = temp;
            }
        }
    }
}
```

A: 8 1 1 2
0 1 2 3

i	j
0	0, 1, 2
1	0, 1
2	0



T.C.: $O(n^2)$

S.C: $O(1)$

10 20 30 40 50

Bubble Sort:

Best Case T.C. ⇒ Bubble Sort not depend on order of data.

But if no. of swapping is 0
that means data is already sorted.
→ If data is already sorted stop the algorithm.

```java
// bubble sort algorithm
public static void bubbleSort(int[] A) {
    int n = A.length;
    for(int i = 0; i < n - 1; i++) {
        boolean swapping = false;
        for(int j = 0; j < n - 1 - i; j++) {
            if(A[j] > A[j + 1]) {
                // swap the data of A[j] and A[j+1]
                int temp = A[j];
                A[j] = A[j + 1];
                A[j + 1] = temp;
                // mark swapping as true for
                // indentification of step
                swapping = true;
            }
        }
        if(swapping == false) {
            return;
        }
    }
}
```

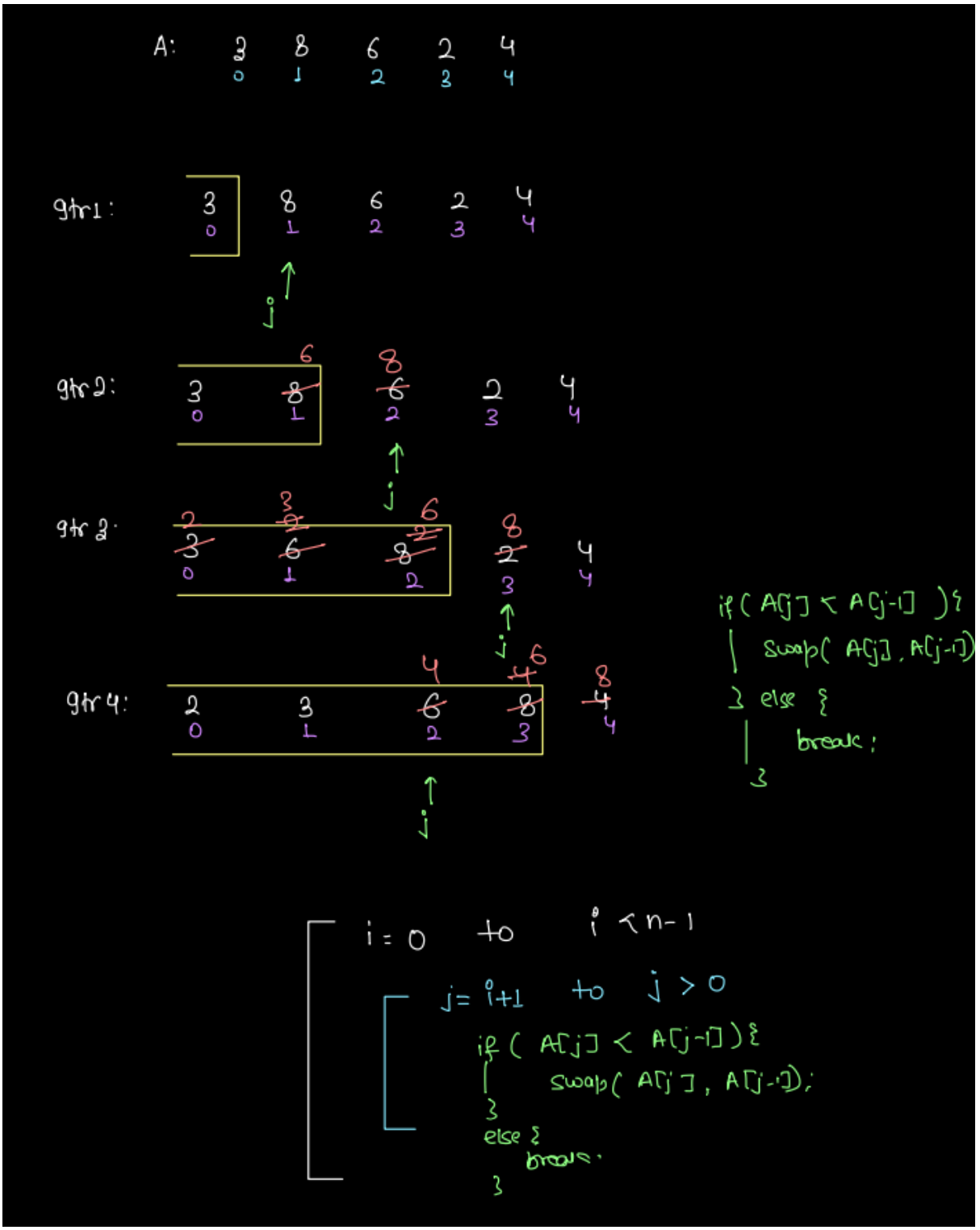
Best case: → data is sorted
T.C. → $O(n)$

worst case → data is not sorted
T.C. → $O(n^2)$

S.C → $O(1)$

8:19 – 8:30 → Break Time

Insertion Sort -> In this algo we will keep first unchanged and asum it is max and from next element we will check is previous element > to current Element if condition true then swap it either break the loop. We will use 2 loop for this. first loop will start from 0 and second from i+1;



```java
// insertion sort
public static void insertionSort(int[] A) {
    int n = A.length;
    for(int i = 0; i < n - 1; i++) {
        for(int j = i+1; j > 0; j--) {
            if(A[j] < A[j-1]) {
                // swap curr i.e. A[j] with prev. i.e. A[j-1]
                int temp = A[j];
                A[j] = A[j - 1];
                A[j - 1] = temp;
            } else {
                // if curr is greater than prev.
                //that means curr ele is already placed at
                // correct position
                break;
            }
        }
    }
}
```

T.C: $\theta(n^2)$

S.C. $O(1)$

array: 7 2 4 6 (0 1 2 3)

i	j	state of array
0	1 → 0	2 7 4 6
1	2 → 1 (break)	2 4 7 6
2	3 → 2 (break)	2 4 6 7

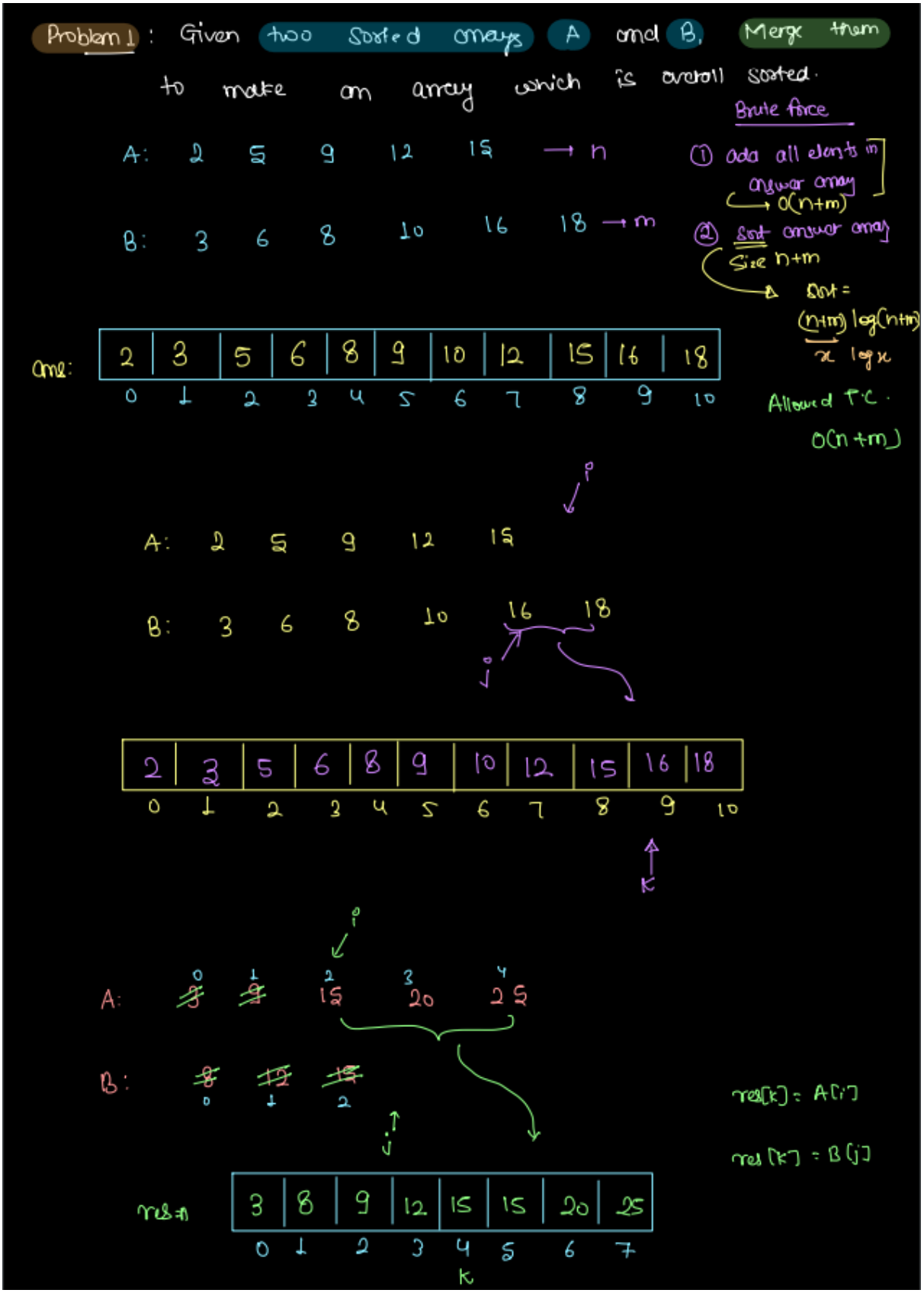
Eg. 10 20 30 40 50

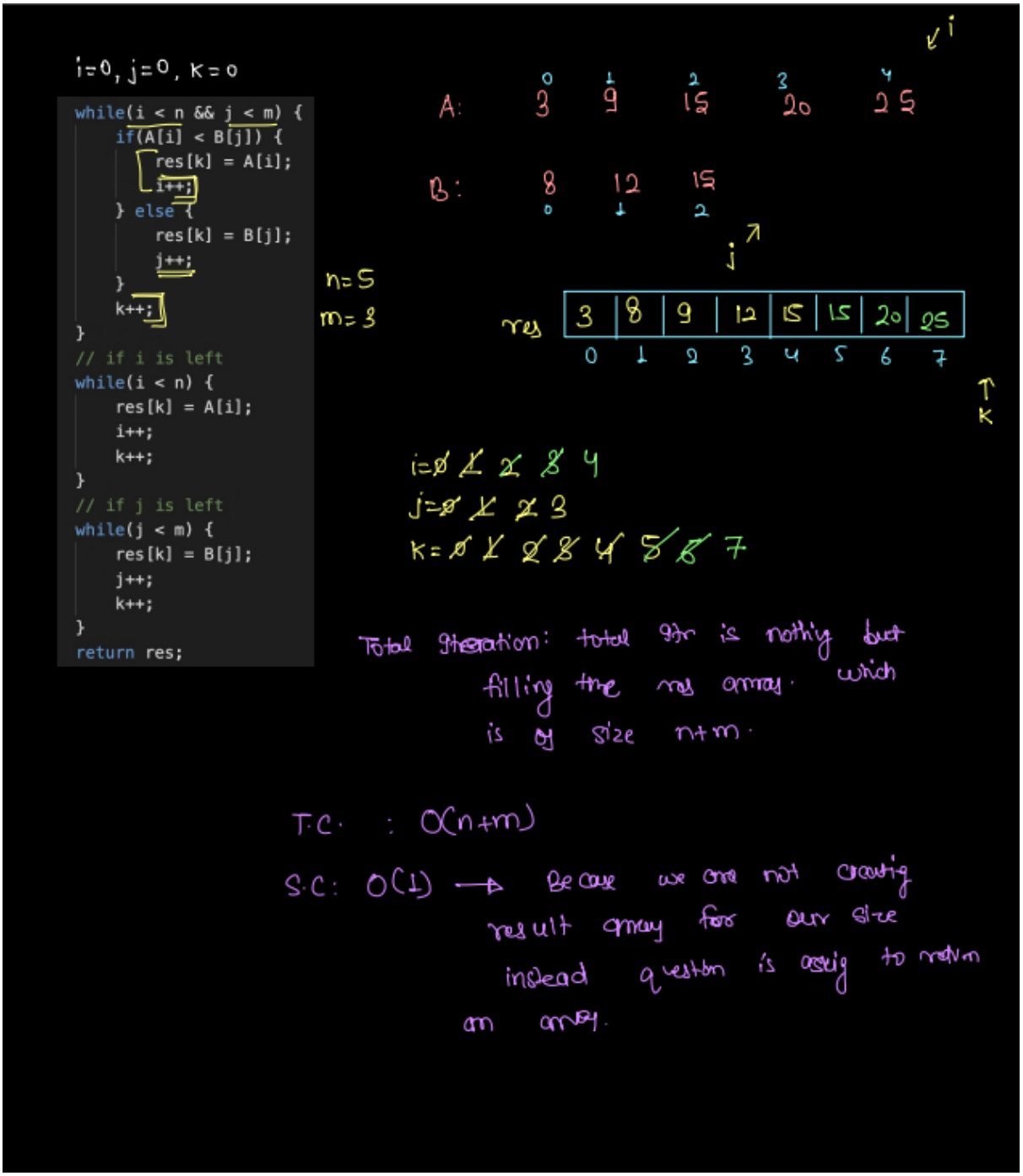
final answer 2 4 6 7 (Sorted)

i	j	state of array
0	1 → break	10 20 30 40 50
1	2 → break	10 20 30 40 50
2	3 → break	10 20 30 40 50
3	4 → break	10 20 30 40 50

T.C: $O(n)$ → Best case in insertion sort i.e. → array is already sorted.

Merge Two Sorted Array





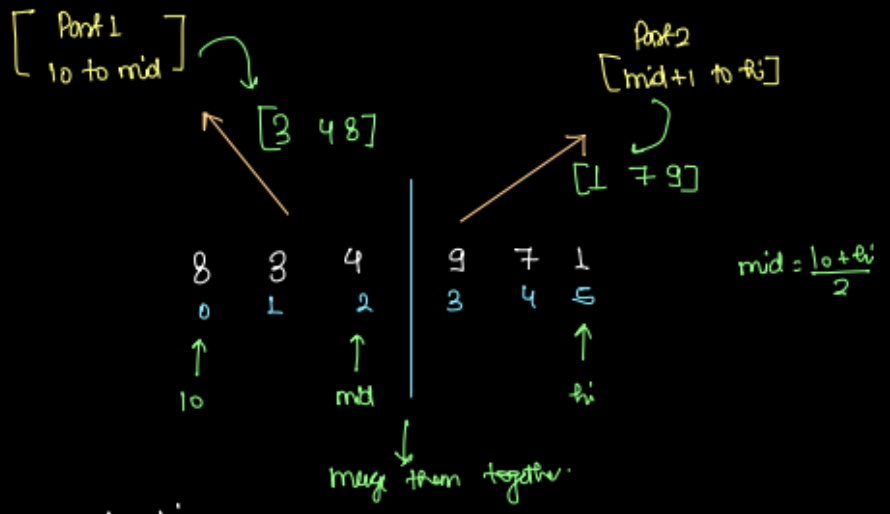
Merge Sort -> Divide and conquer algorithm (TC: NlogN)

This algorithm based on recursion. In this algorithm We will get the mide point using lo+hi/2 like if array has 1,2,3,4,5,6 then lo is 0 index and hi is 5th index and mid point is 2. now we will make it two half of array and then we will merge both the array till we will not reach to base case where lo == hi.

Merge Sort : T.C. O(nlogn)

divide and conquer algorithm

↳ this algorithm is based on Recursion.



Assumption → arr, lo, hi

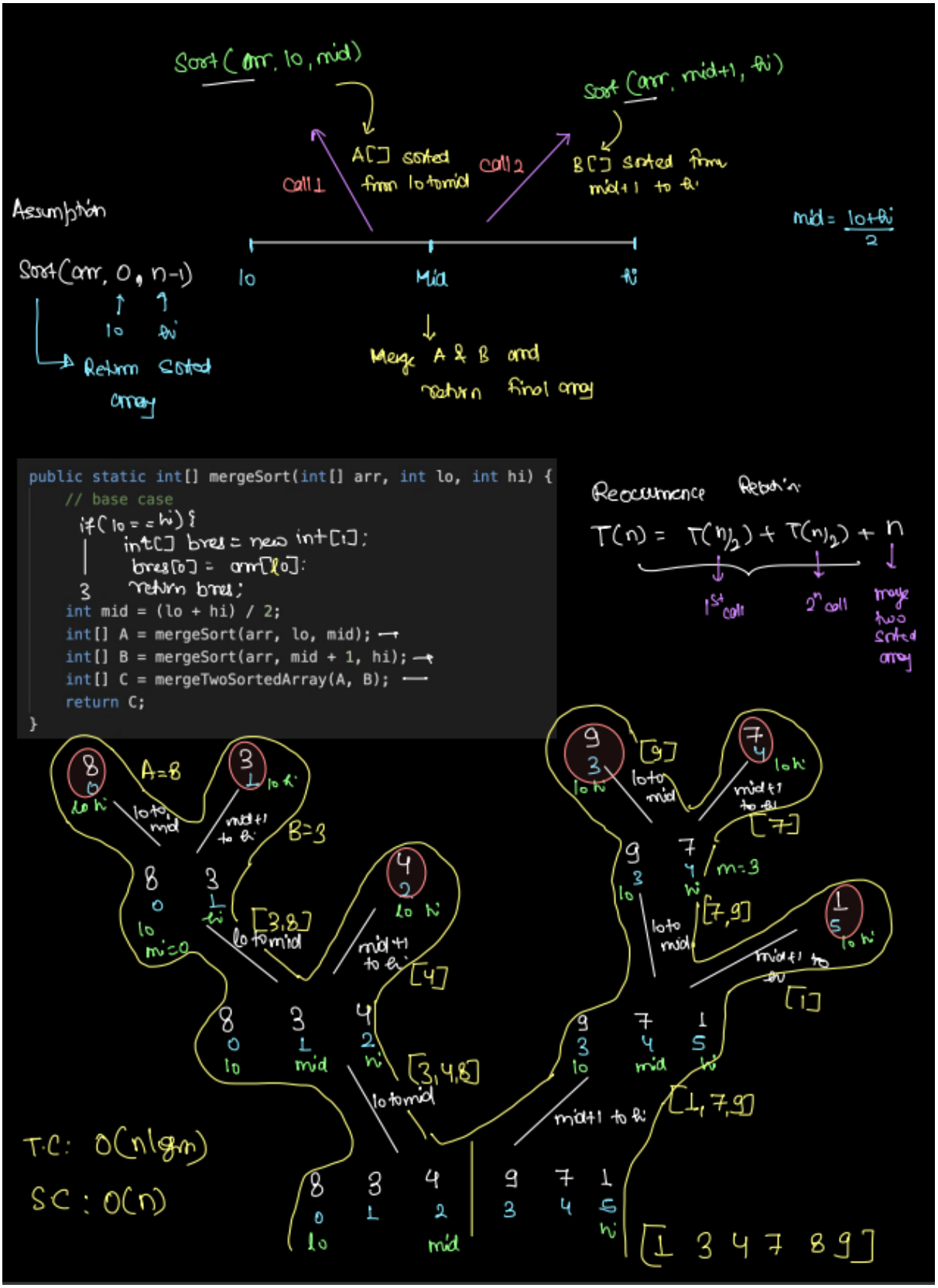
Sort array from lo to hi

Main logic: →

Sort (arr, lo, mid) → A

Sort (arr, mid+1, hi) → B

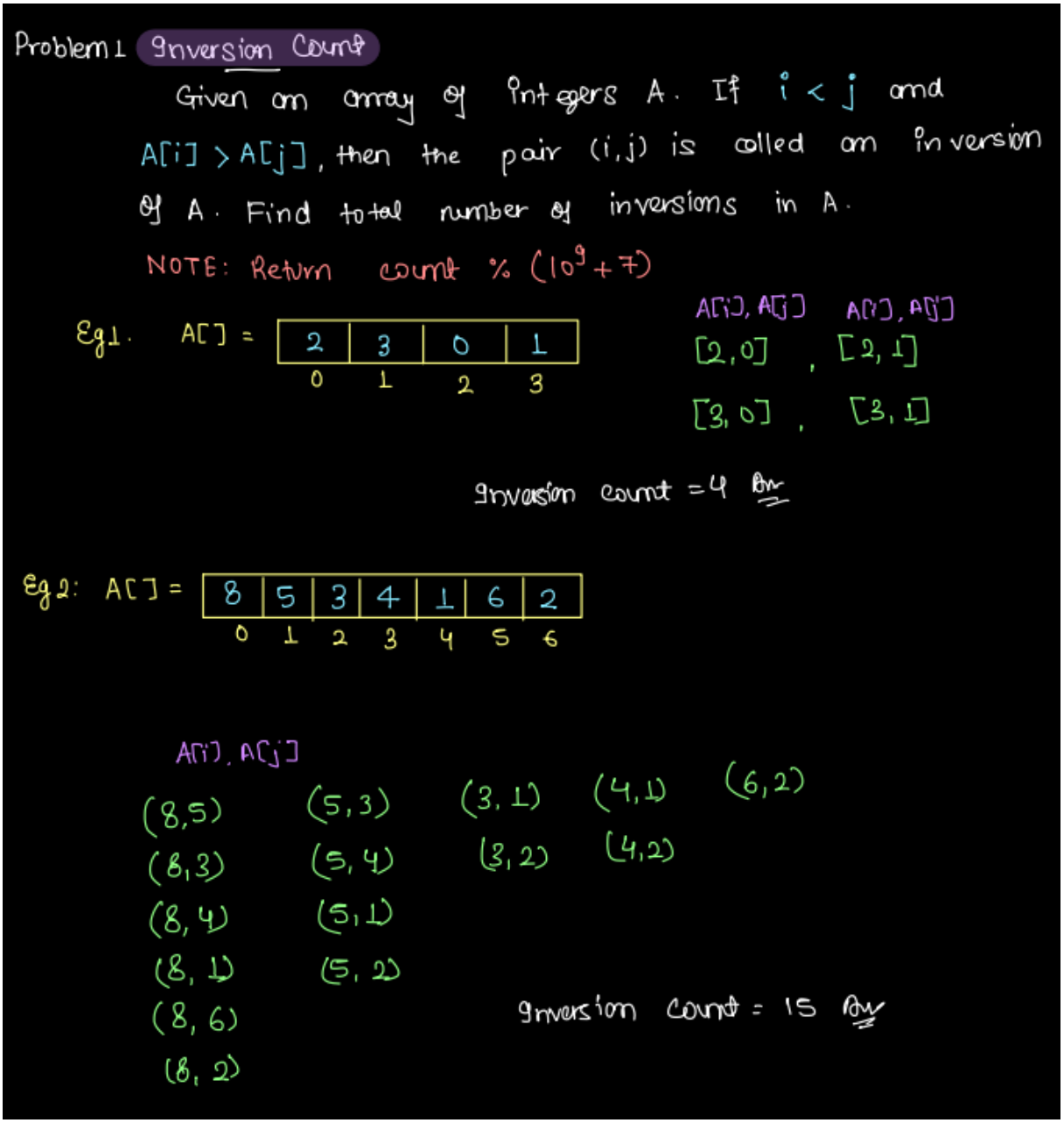
My task → Merge them together.


Sort(arr, lo, mid)
Sort(arr, mid+1, hi)
A[] sorted from lo to mid
B[] sorted from mid+1 to hi
Call 1
Call 2
Assumption
Sort(arr, 0, n-1)
lo
hi
Return sorted array
mid = lo+hi / 2
lo
Mid
hi
Merge A & B and return final array
public static int[] mergeSort(int[] arr, int lo, int hi) {
// base case
if(lo == hi) {
int[] bres = new int[1];
bres[0] = arr[lo];
return bres;
}
int mid = (lo + hi) / 2;
int[] A = mergeSort(arr, lo, mid);
int[] B = mergeSort(arr, mid + 1, hi);
int[] C = mergeTwoSortedArray(A, B);
return C;
}
Recurrence Relation:
T(n) = T(n/2) + T(n/2) + n
1st call
2nd call
merge two sorted array
A=8
B=3
[3,8]
[4]
[3,4,8]
[9]
[7]
m=3
[7,9]
[1]
[1,7,9]
[1 3 4 7 8 9]
T.C: O(nlogn)
S.C: O(n)

DSA: Sorting 1 - 8 - June 2023

- **Inversion Count**
- **Custom Comparison**
- **Factor Sort**
- **Sort Coordinates**
- **Largest Number**

Inversion Count -> if A[i] > A[j] then all pairs with A[i] called inversions like below. It will solved using recursion shorting function like merge short where we will divid array in 2 parts and then call same function using mergeSort(A, lo, mid), merge Sort (A, mid+1, hi) and mid will calculate n+m/2



Brute force Idea:

for value of A[i], we will count value of A[j] in which A[i] > A[j] & j = i+1 to n-1.

```
int inversionCount(int[] A) {
    int n = A.length;
    long count = 0;
    for(int i=0; i< n; i++) {
        for(int j=i+1; j<n; j++) {
            if( A[i] > A[j] ){
                count = (count + 1)% mod
            }
        }
    }
    return (int)(count);
}
```

T.C : $O(n^2)$

S.C : $O(1)$

Time complexity

i=0 , j=1 to n-1 ⟶ n-1

i=1 , j=2 to n-1 ⟶ n-2

i=2 , j=3 to n-1 ⟶ n-3

⋮

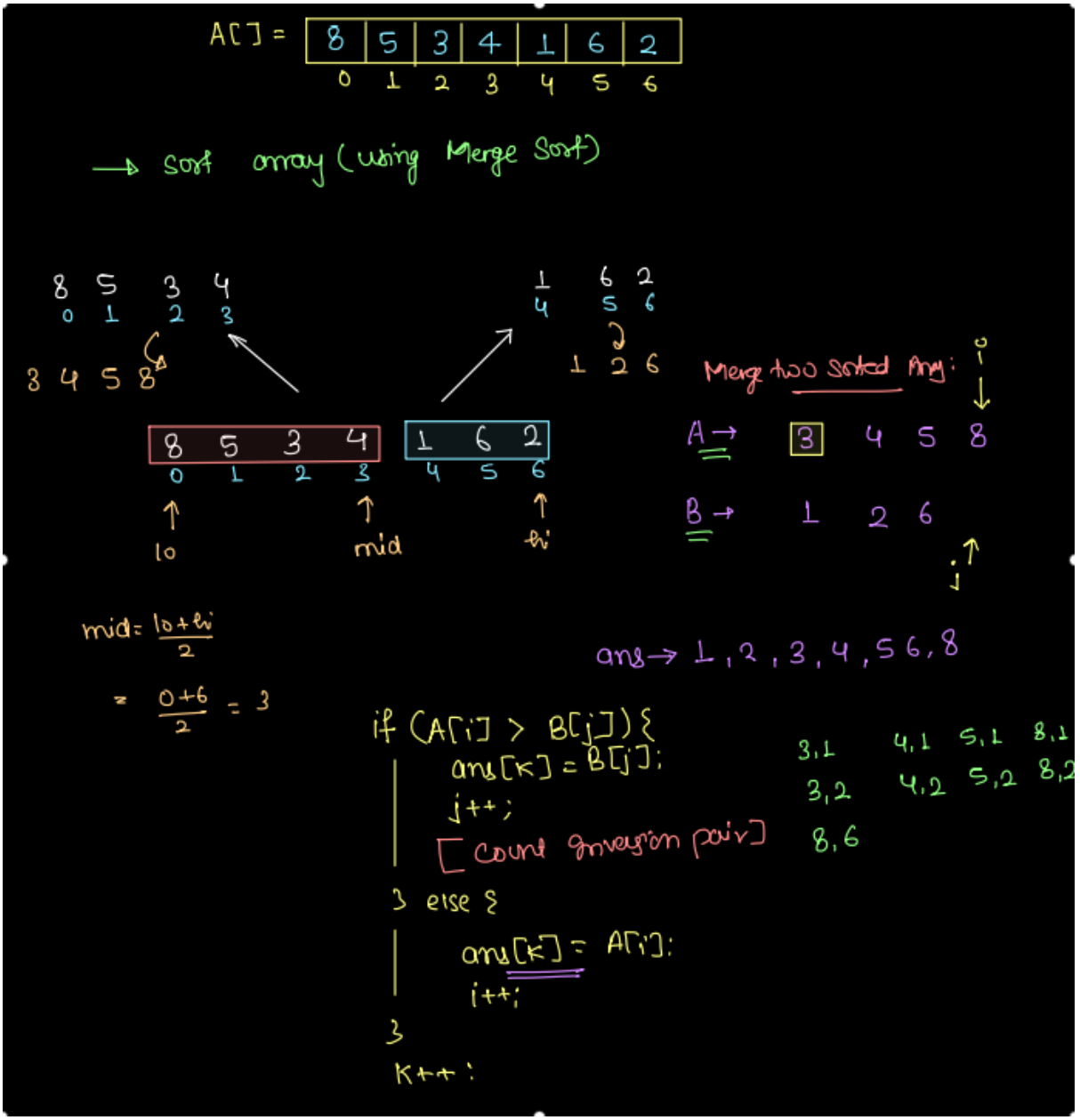
i=n-2 , j=n-1 to n-1 ⟶ 1

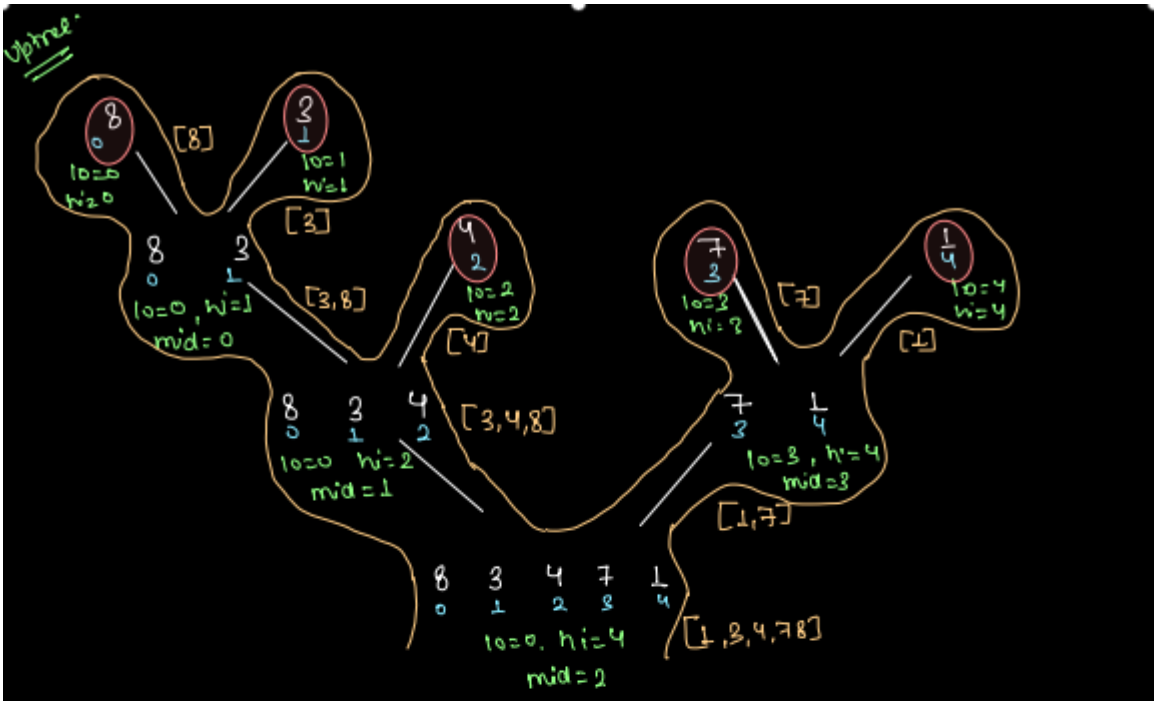
Total str = $1+2+3+\cdots+n-2+n-1$

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

T.C = $O(n^2)$

Expected T.C : $O(n \log n)$


A[] = 8 5 3 4 1 6 2
0 1 2 3 4 5 6
→ Sort array (using Merge Sort)
8 5 3 4
0 1 2 3
3 4 5 8
1 6 2
4 5 6
1 2 6
8 5 3 4 | 1 6 2
0 1 2 3 4 5 6
lo
mid
hi
Merge two Sorted Arr:
A → 3 4 5 8
B → 1 2 6
i
j
mid = lo+hi / 2
= 0+6 / 2 = 3
ans → 1, 2, 3, 4, 5 6, 8
if (A[i] > B[j]) {
ans[k] = B[j];
j++;
[Count inversion pair]
} else {
ans[k] = A[i];
i++;
}
k++;
3,1 4,1 5,1 8,1
3,2 4,2 5,2 8,2
8,6



A[] → 3 4 8 ↙ i

B[] → 1 7 ↑ j

C[] → 1, 3, 4, 7, 8

A[i] > B[j]

Count += (n-i)

count = 0 + 1 + 1 + 1 + 3 + 1

= 7 Ans

(8,3), (8,4), (7,1)

(3,1) (4,1) (8,1)

(8,7)

We are just adding one single line in merge two sorted array

8:32 – 8:45

Break time

→ Rest Every code is same

T.C : O(nlogn)

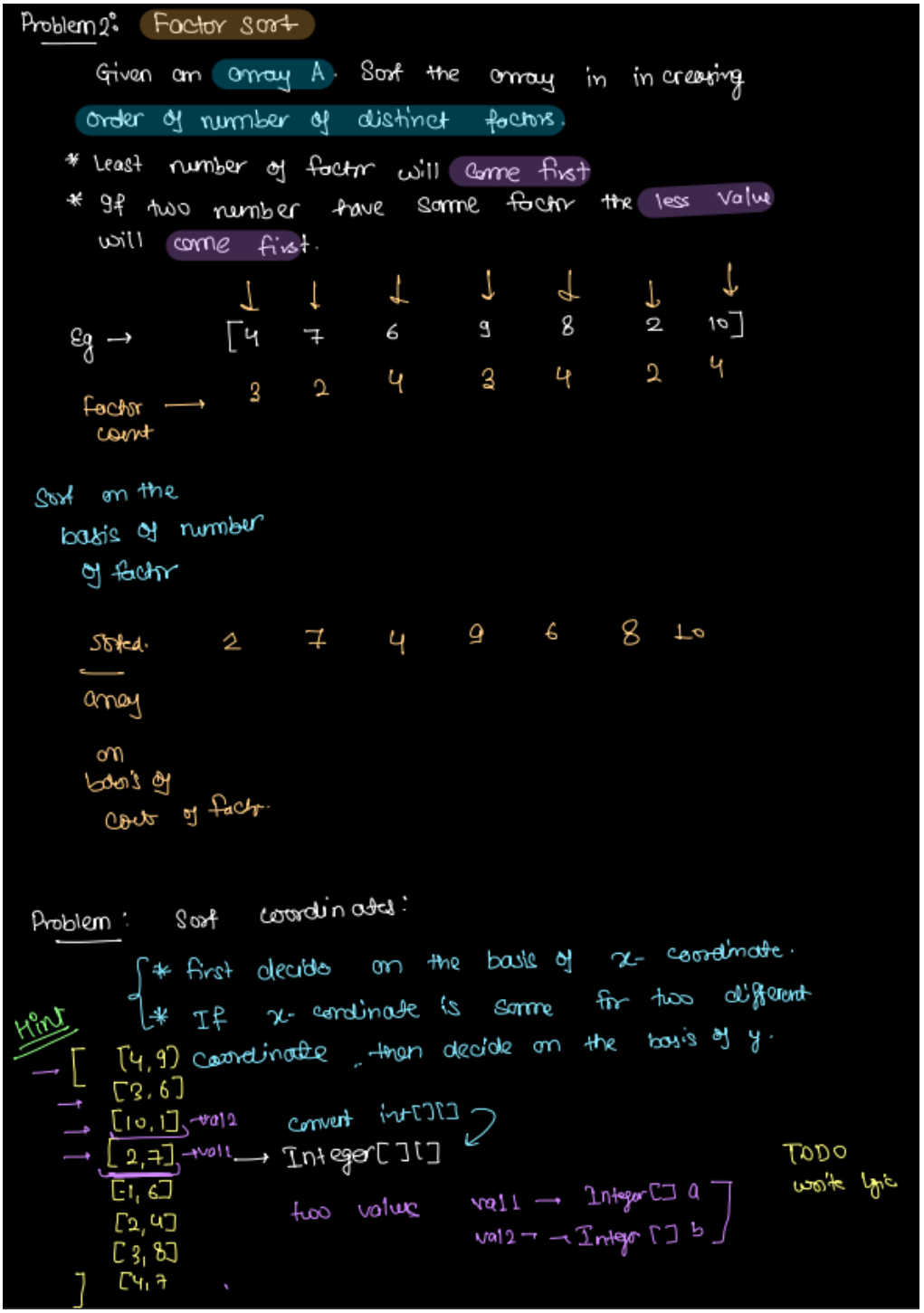
S.C. : O(n)

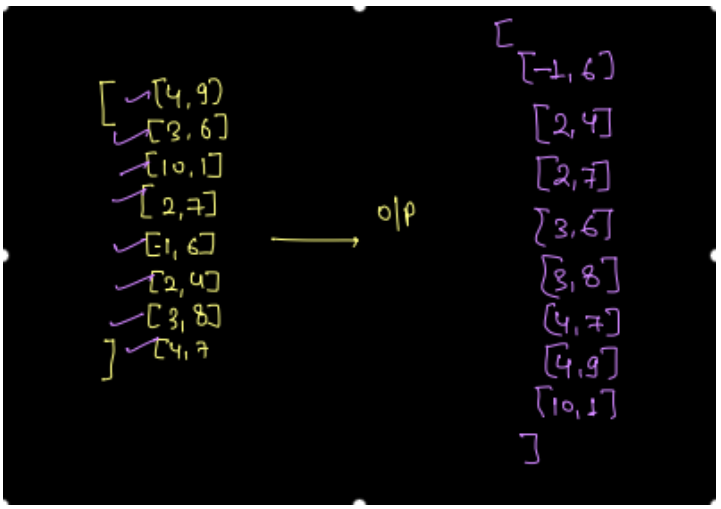
Custom Comparison -> We will write custom short function for custom comparison like if we want to sort car by price or by feature then we have to write it because default sort feature only provide chat, int sorting. for custom sorting function we will use sort method and pass second argument as comparator function like below.
We will use Interger class reference type not primitive type int and array also we have to convert in int to Integer. We will use @Override it is optional. If we want to return first parameter

```
Arrays.sort(arr, new Comparator<Integer>() {
@Override
public int compare(Integer a, Integer b) {
// Possibility 1 : want to keep a before b
// return negative value
// Possibility 2 : want to keep b before a
// return positive value
// Possibility 3 : Doesnot matter
// return 0

// noraml Order
/*
if(a < b) {
// want to keep a before b
return -1;
} else if(a > b) {
// want to keep b before a
return 1;
} else {
// Doesnot matter
return 0;
}
*/
// reverse the order
if(a < b) {
// want to keep b before a
return 1;
} else if(a > b) {
// want to keep a before b
return -1;
} else {
// Doesnot matter
return 0;
}
```

```
// sorter only use this instead above
return a - b;
// for order reversal
// return b - a;
}
});
```

Factor Sort

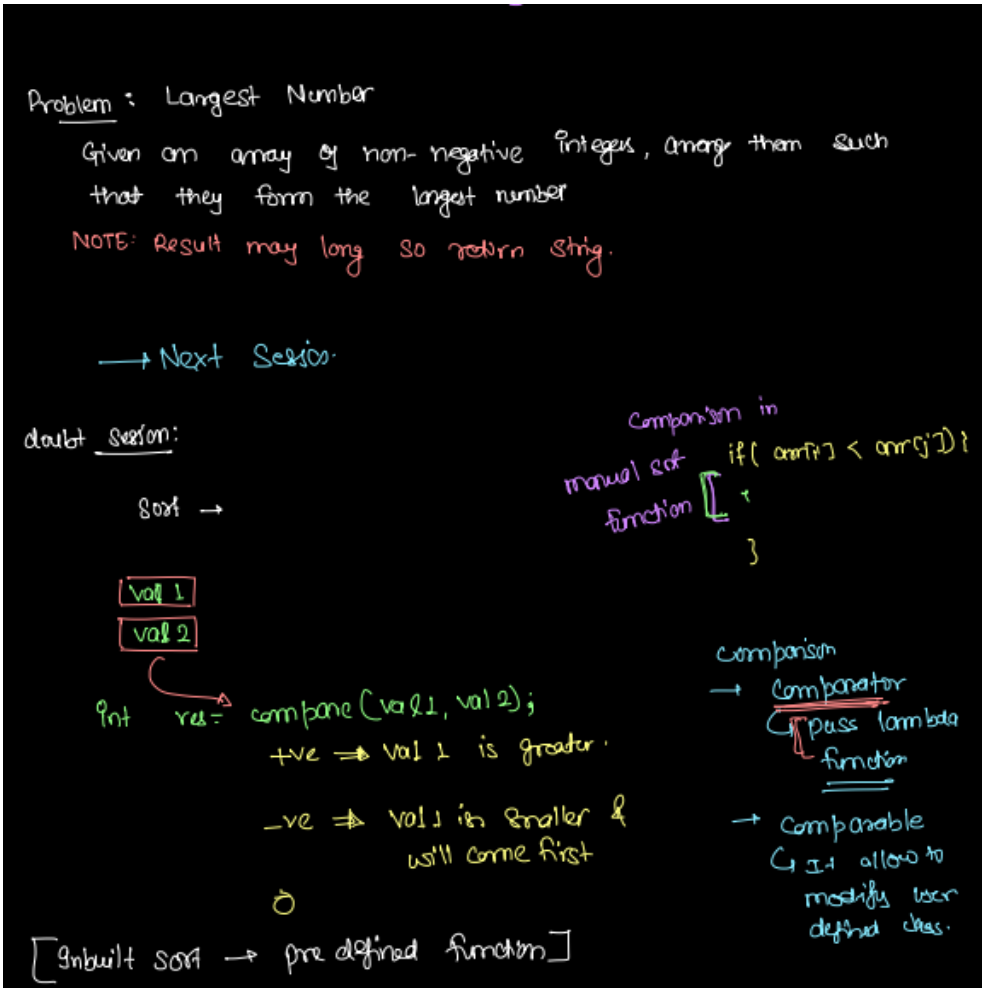





```
class Main {
/* Problem : factor Sort */
public static int factorCount(int val) {
// todo : write optimise code
int fcount = 0;
for(int i = 1; i <= val; i++) {
if(val % i == 0) {
fcount++;
}}
return fcount;
}
public static int[] factorSort(int[] arr) {
int n = arr.length;
// convert int array into Integer array
Integer[] A = new Integer[n];
for(int i = 0; i < n; i++) {
A[i] = arr[i];
}
Arrays.sort(A, new Comparator<Integer>() {
@Override
public int compare(Integer a, Integer b) {
int fa = factorCount(a);
int fb = factorCount(b);
if(fa < fb) {
// a come first
return -1;
} else if(fa > fb) {
// b compe first
return 1;
} else {
```

```
// if both are equal
// decide according to value of a and b
return a - b;
}}
});
// fill arr from A
for(int i = 0; i < n; i++) {
arr[i] = A[i];
}
return arr;
}
public static void main(String args[]) {
// Your code goes here
int[] arr = {4, 7, 6, 9, 8, 2, 10};
System.out.println(Arrays.toString(arr));
arr = factorSort(arr);
System.out.println(Arrays.toString(arr));
}}
```

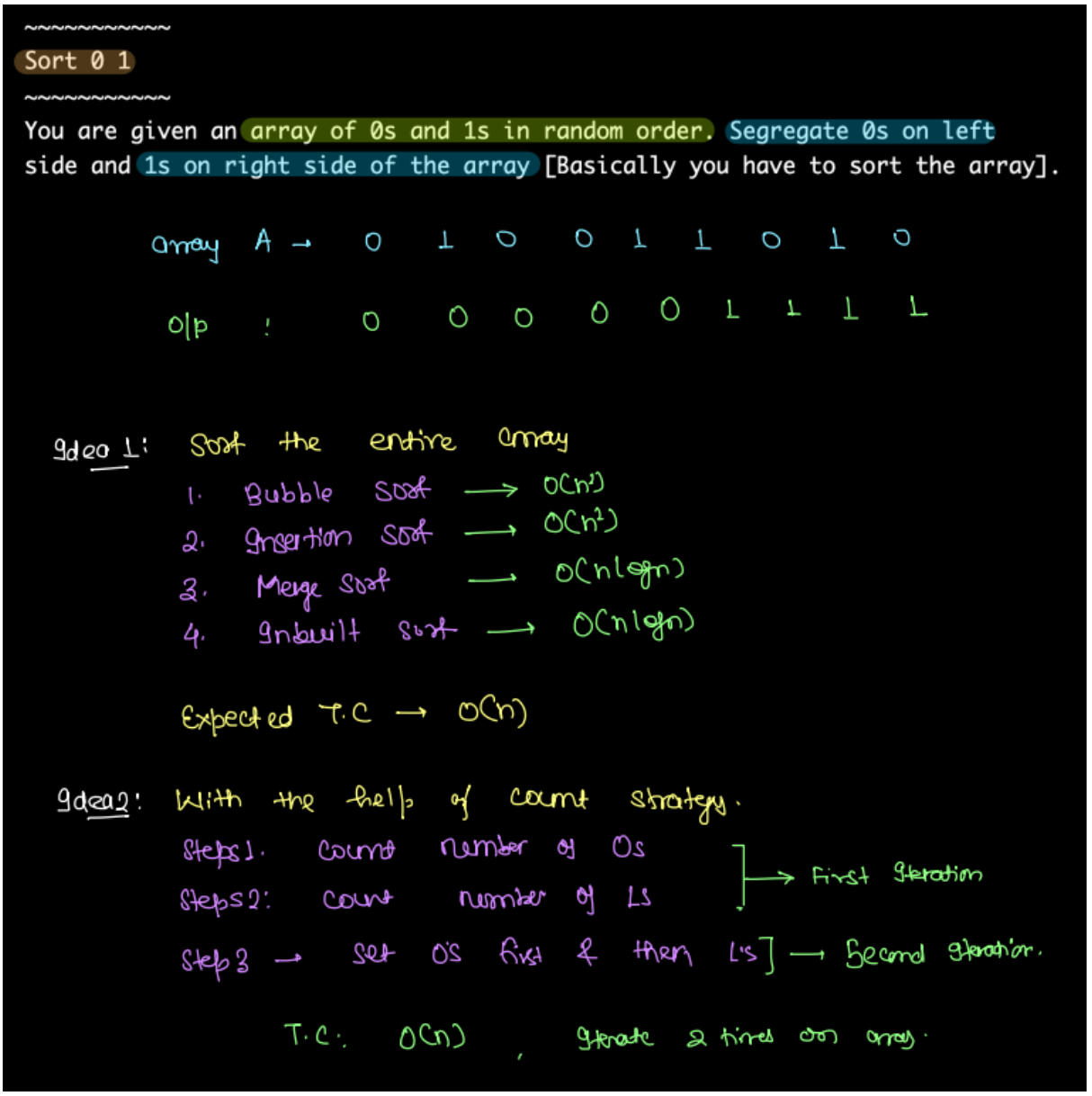
Largest Number

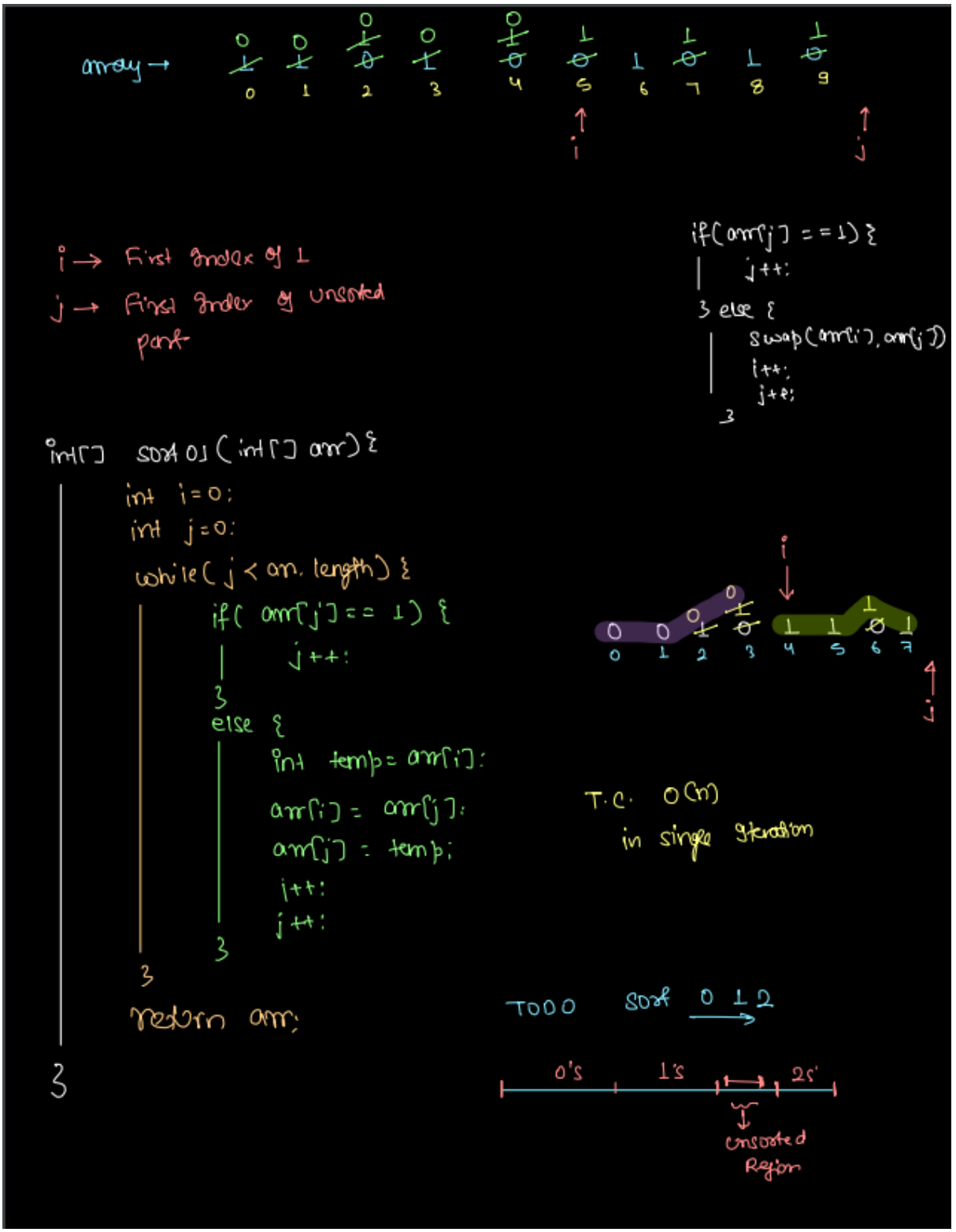


DSA: Sorting 3 - 10 - June 2023

- **Sort 0 1**
- **Partition of Array**
- **Quick Sort Algorithm**
- **TC and SC of quickSort**
- **Worst case of quickSort**
- **Largest Number**
- **Sort Coordinates**

Sort 0 1 -> sort all 0's first then 1's after. We can use any sorting algorithm but we have to sort this in O(n). we will use Aproch swap in single loop. We will define 2 initial variable i and j to 0 and then we will apply condition if j == 1 then j++ go to next element and if j !== 1 then we will swap in same array j to i and i to j.





i → First index of 1

j → First index of unsorted part

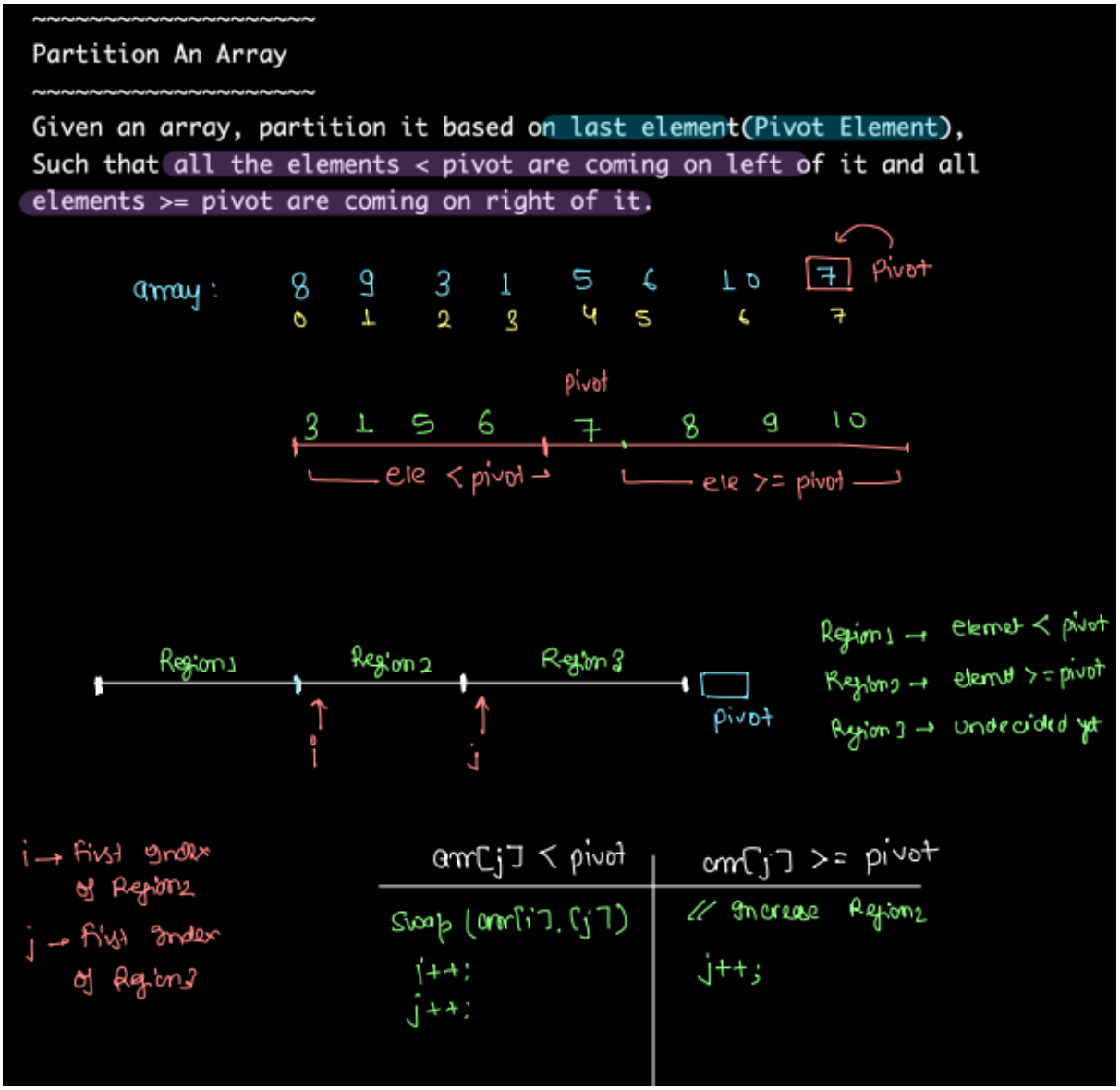
```
if(arr[j] == 1) {
    j++;
} else {
    swap(arr[i], arr[j])
    i++;
    j++;
}
```

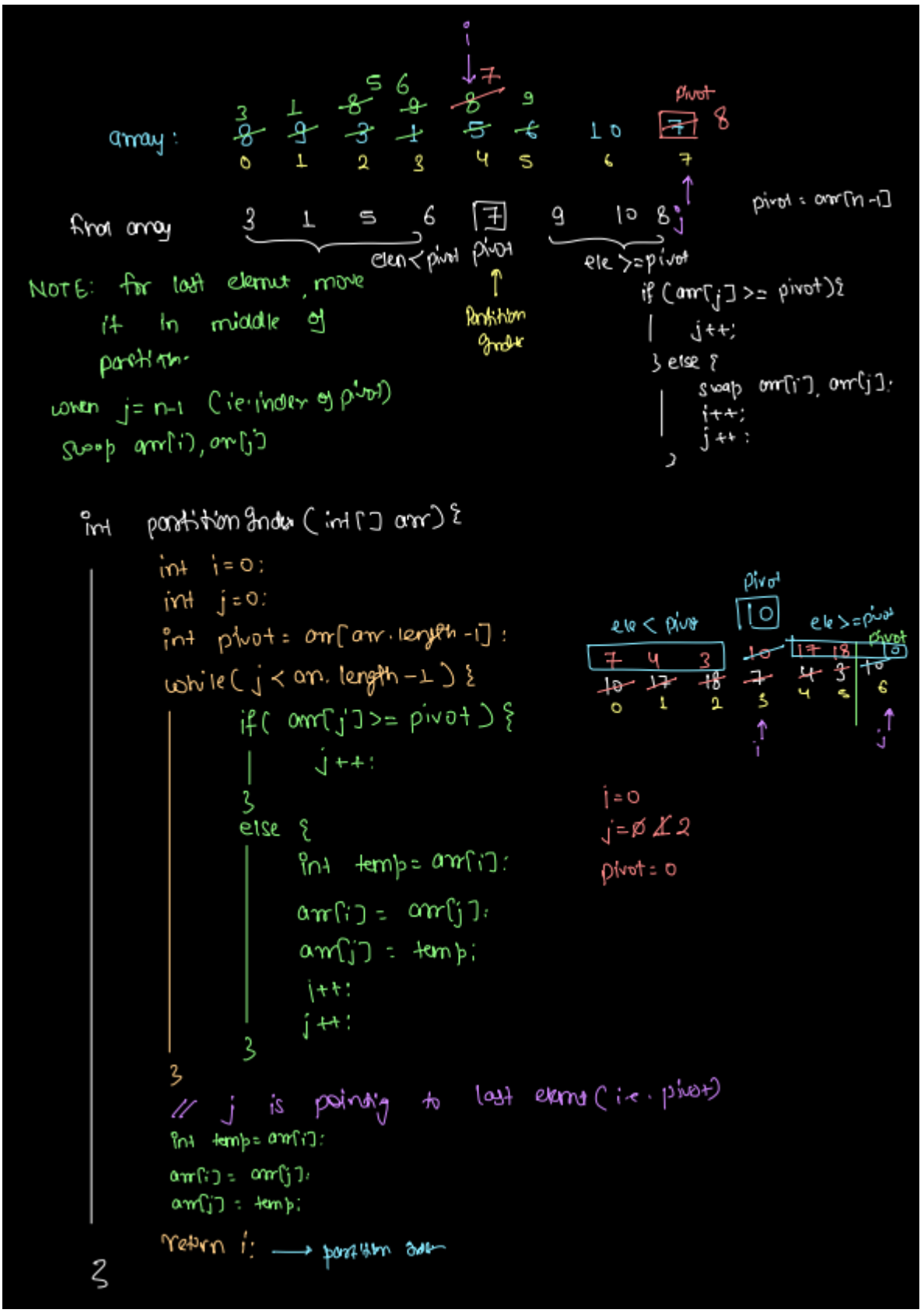
```
int[] sortOI(int[] arr) {
    int i=0;
    int j=0;
    while(j < arr.length) {
        if(arr[j] == 1) {
            j++;
        }
        else {
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
            i++;
            j++;
        }
    }
    return arr;
}
```


T.C. O(n)
in single iteration

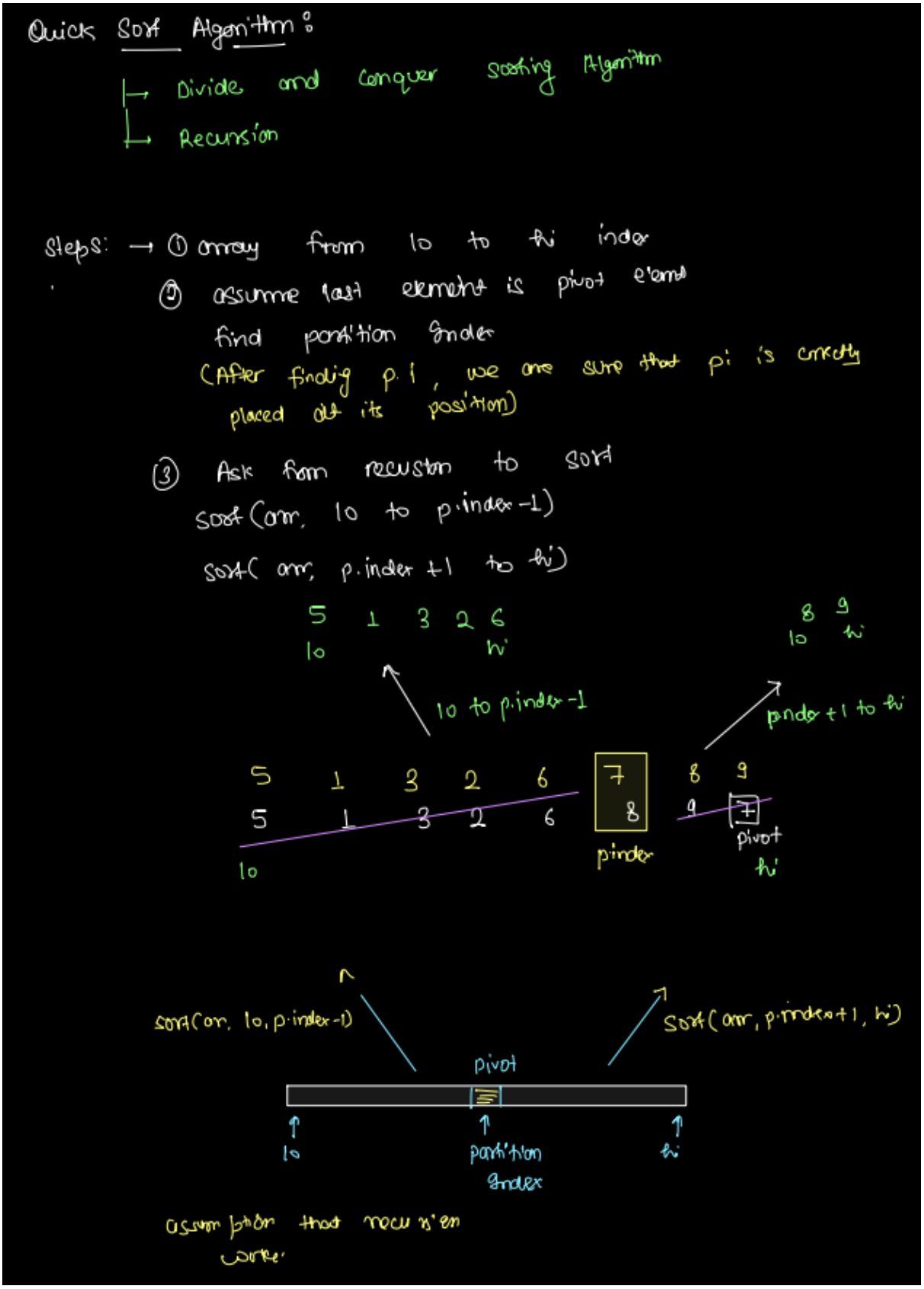
TODO Sort 0 1 2 →

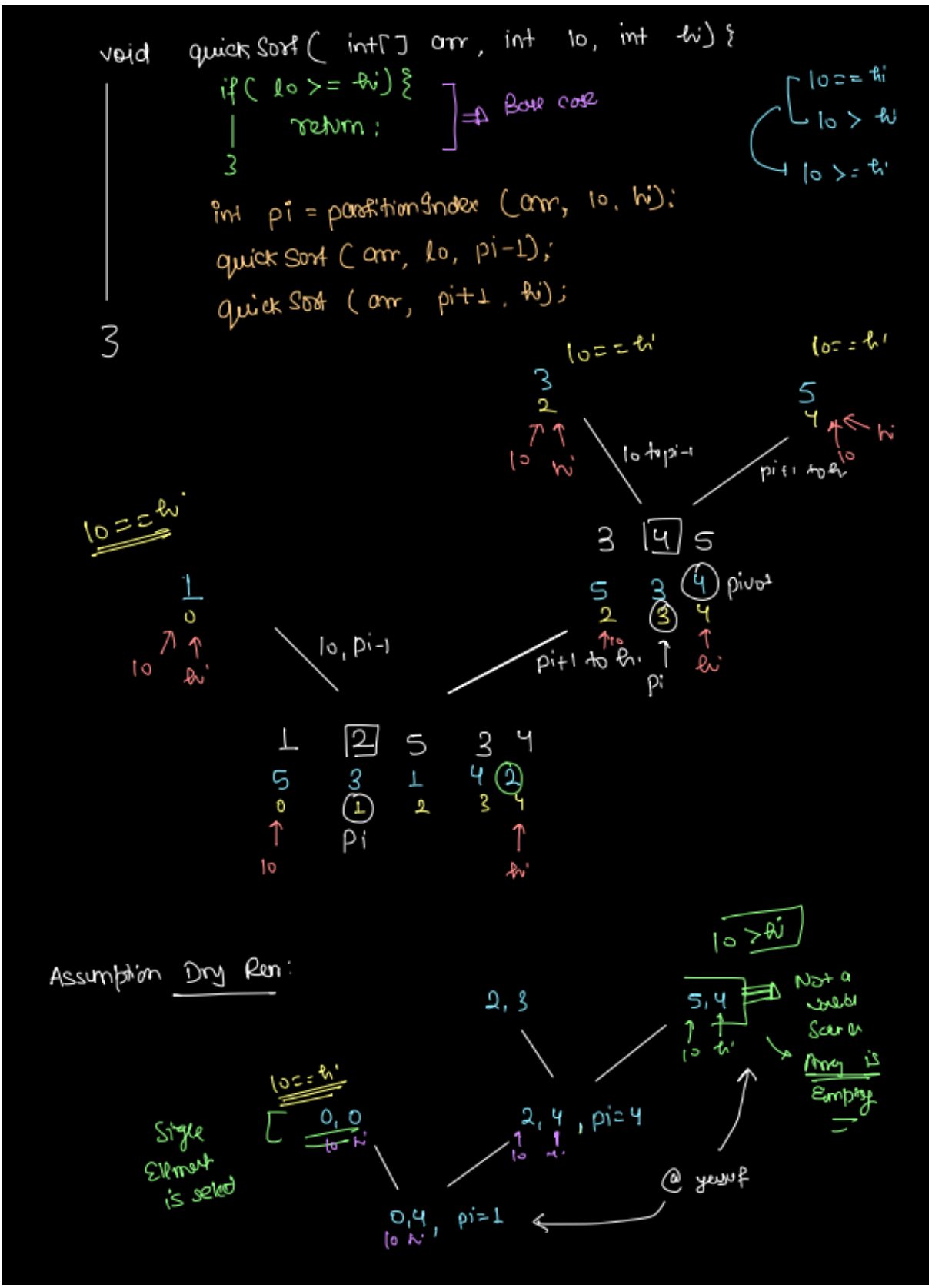
Partition of Array




array :
pivot
final array
pivot = arr[n-1]
ele < pivot
pivot
Partition index
ele >= pivot
NOTE: for last element, move it in middle of partition.
Given j = n-1 (ie. index of pivot)
Swap arr[i], arr[j]
if (arr[j] >= pivot){
j++;
} else {
swap arr[i], arr[j];
i++;
j++;
}
int partitionIndex (int[] arr) {
int i = 0;
int j = 0;
int pivot = arr[arr.length -1];
while(j < arr.length -1) {
if(arr[j] >= pivot) {
j++;
}
else {
int temp = arr[i];
arr[i] = arr[j];
arr[j] = temp;
i++;
j++;
}
}
// j is pointing to last element (i.e. pivot)
int temp = arr[i];
arr[i] = arr[j];
arr[j] = temp;
return i; → partition index
}
ele < pivot
pivot
ele >= pivot
pivot
i
j
i = 0
j = 0 & 2
pivot = 0

Quick Sort Algorithm



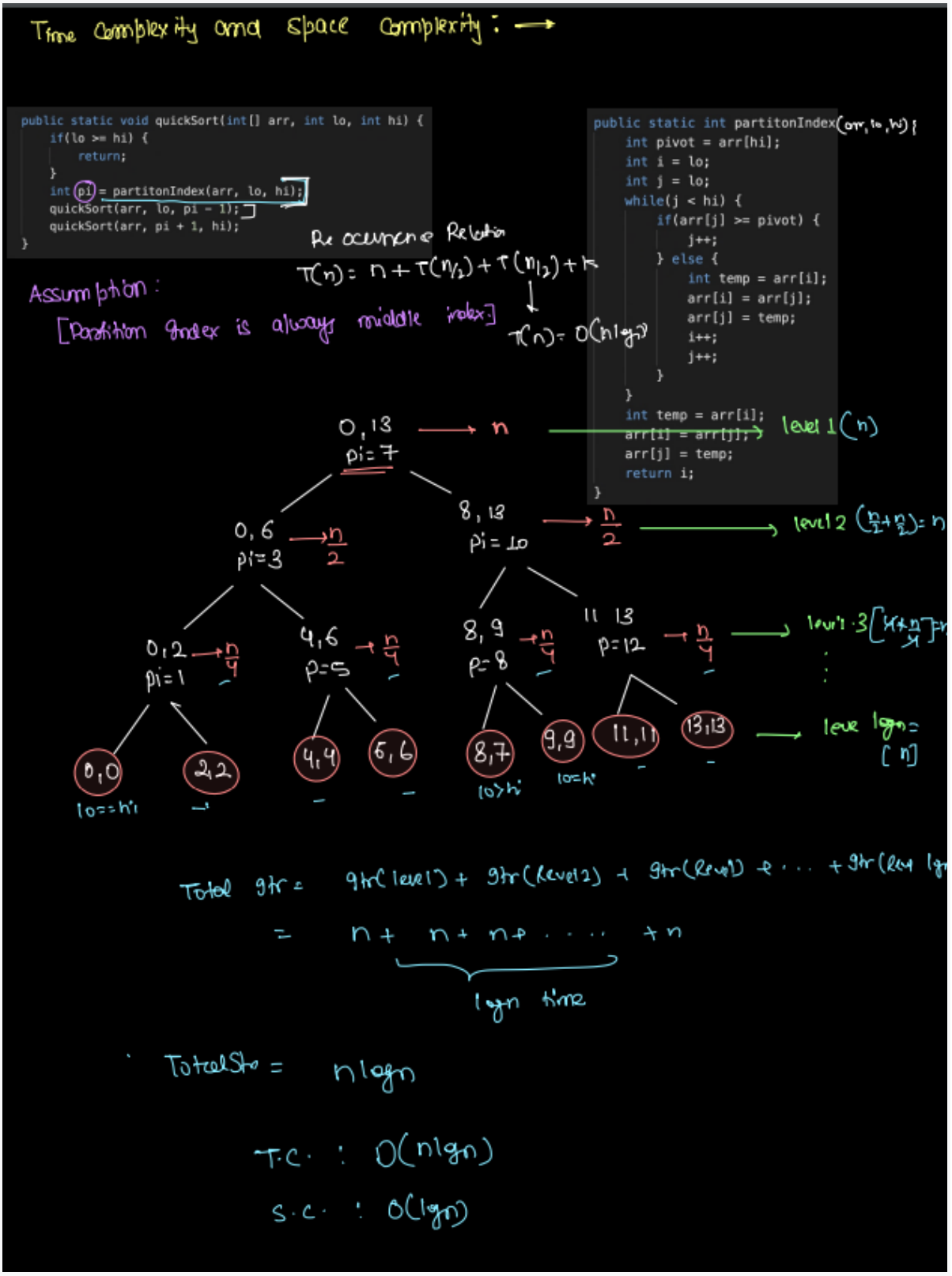


In this particular scenario, the initial array is [0, 1, 2, 3, 4], which has been sorted. Now, we are examining the second condition. The first base condition is when the "lo" index is equal to the "hi" index. However, upon further consideration of different scenarios, including the one

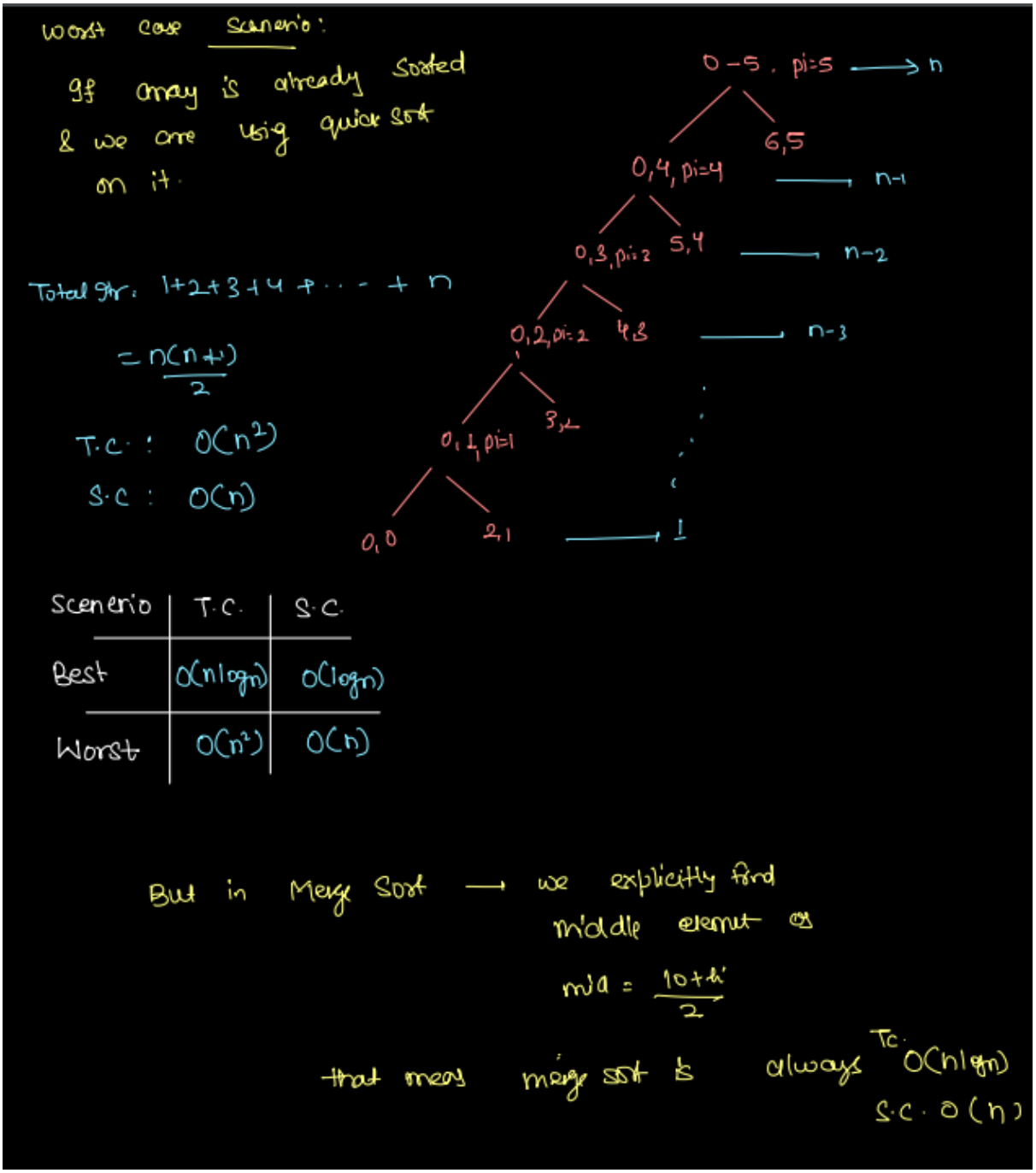
mentioned earlier, we observe that when we encounter a pivot (in this case, 4), we increment the pivot value by 1 and compare it to the elements in the array. Since there is no element equivalent to "pivot + 1" (i.e., 5) in the array, we can conclude that if we don't have an element greater than our pivot element, we should return from the function. This leads to the introduction of a new condition: "lo > hi". Consequently, we combine both conditions and establish a new base case: "lo >= hi".

```java
public class Solution {
    public static int partitionIndex(int[] arr, int lo, int hi){
        int pivot = arr[hi];
        int i = lo;
        int j = lo;

        while(j < hi){
            if(arr[j] >= pivot){
                j++;
            }else{
                int temp = arr[i];
                    arr[i] = arr[j];
                    arr[j] = temp;
            }
        }

        int temp = arr[i];
                arr[i] = arr[j];
                    arr[j] = temp;

        return i;

    }
    public static void quickSort(int[] arr, int lo, int hi){
        if(lo >= hi){
            return ;
        }

        int pi = partitionIndex(arr, lo, hi);
        quickSort(arr, lo, pi-1);
        quickSort(arr, pi + 1, hi);

    }
    public int[] solve(int[] A) {
        quickSort(A, 0, A.length-1);
        return A;
    }
}
```

TC and SC of quickSort



Worst case of quickSort



Largest Number

long.parseLong('str') is a function which convert string to long. Why it required because we can not compire to string directly then we have to convert it in long.

String.valueOf -> It is used for convert int to String.

left over:

Problem : Largest Number

Given an array of non-negative integers, arrange them such that they form the largest number

NOTE: Result may long so return String.

arr[] → {2 3 9 0} → {9 3 2 0} → 9320

Sort it in decr. order - ? ✗ Not work

arr[] → {99 90 98} → {99 98 90} ⇒ 999890

{99 90 98} ⇒ 999098

arr[] → {998, 9} → {998, 9} ⇒ 9989

{9, 9983} → 9998

Sort on the basis of digits → lexicographical order. → Not work

arr[] → {30, 3} → {30, 3} → 303

{3, 30} → 330

arr[] {34, 3} → {34, 3} → 343

{3, 34} → 334

Conclusion:

① Sorting on decreasing order will not work

② Sorting on digit will also not work

num1 → a

num2 → b

possibility ⇒ → ab

└ ba

ab > ba → a first in array

ab < ba → b come first

Steps → ① Convert Integer array into String array.

② Arrays.sort (str_Array, new Comparator<String>(){

```
public int compare ( String a , String b ) {
    String n1 = a+b;
    String n2 = b+a;
    Long val1 = Long.parseLong(n1);
    Long val2 = Long.parseLong(n2);
    if (val1 > val2 {
        // ab is best → a comes first
        return -1;
    } else if (val1 < val2) {
        // ba is best → b come first
        return 1;
    } else {
        return 0;
    }
}
```

Coordinates

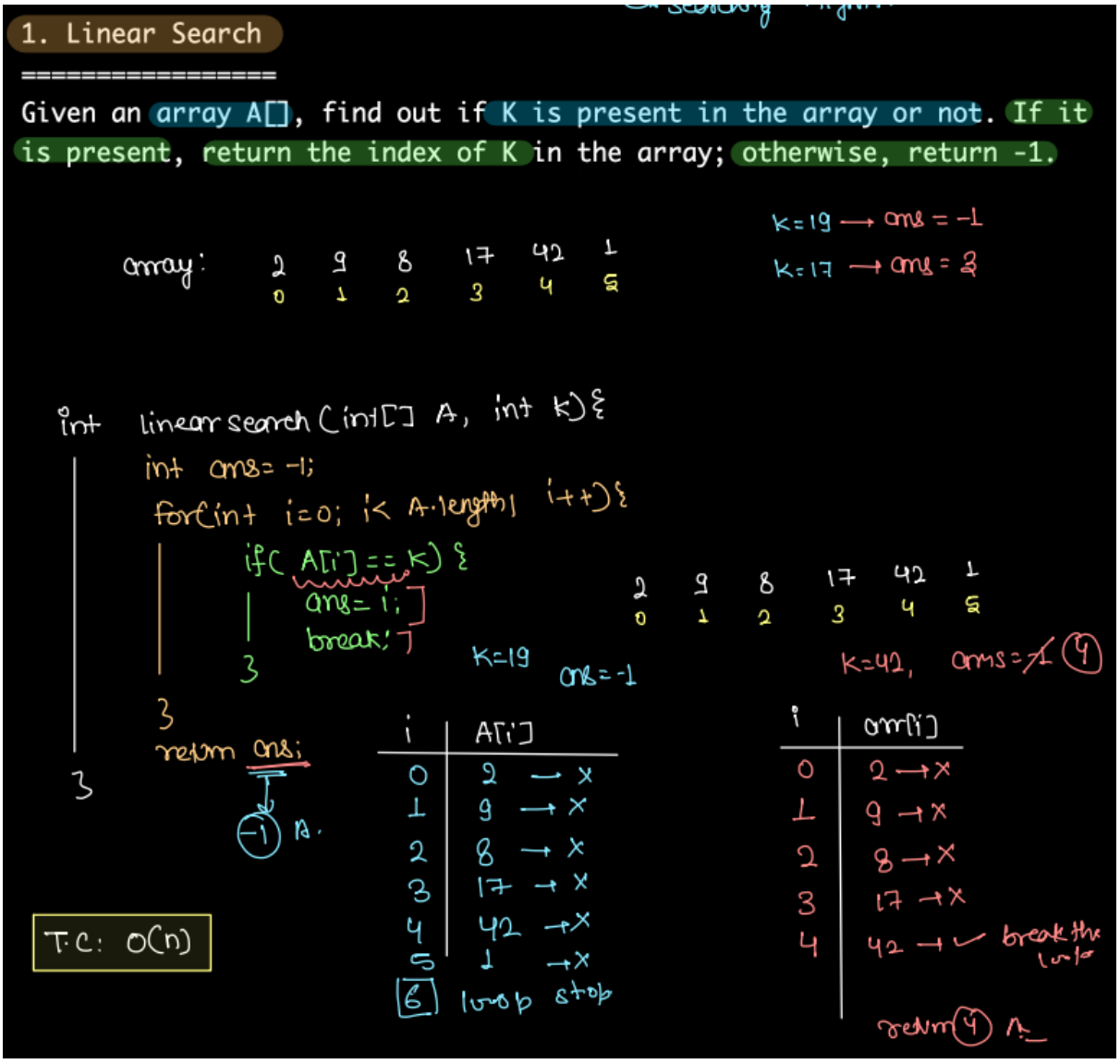
```
import java.util.*;

class Main {
public static void sortCordinate() {
int[][] arr = {
{1, 1},
{4, 9},
{-1, 5},
{4, 4},
{3, 0},
{4, 1},
{3, 0}
};
// convert it into Integer
Integer[][] A = new Integer[arr.length][2];
for(int i = 0; i < arr.length; i++) {
A[i][0] = arr[i][0];
A[i][1] = arr[i][1];
}
for(Integer[] ele : A) {
System.out.println(ele[0] + " " + ele[1]);
}
// sort
Arrays.sort(A, new Comparator<Integer[]>() {
public int compare(Integer[] a, Integer[] b) {
int x1 = a[0];
int y1 = a[1];
int x2 = b[0];
int y2 = b[1];
// first compare on the basis of x
if(x1 != x2) {
return x1 - x2;
}
// if they are same compare on the basis of y
return y1 - y2;
}
});
System.out.println("~~~~~~~");
for(Integer[] ele : A) {
System.out.println(ele[0] + " " + ele[1]);
}
```

```
}
public static void main(String args[]) {
sortCordinate();
}
}
```

DSA: Searching - 13 - June 2023

- **Linear Search**
- **Binary Search**
- **First Occurrence of K**
- **Floor Of K in Array**
- **Local Minima**

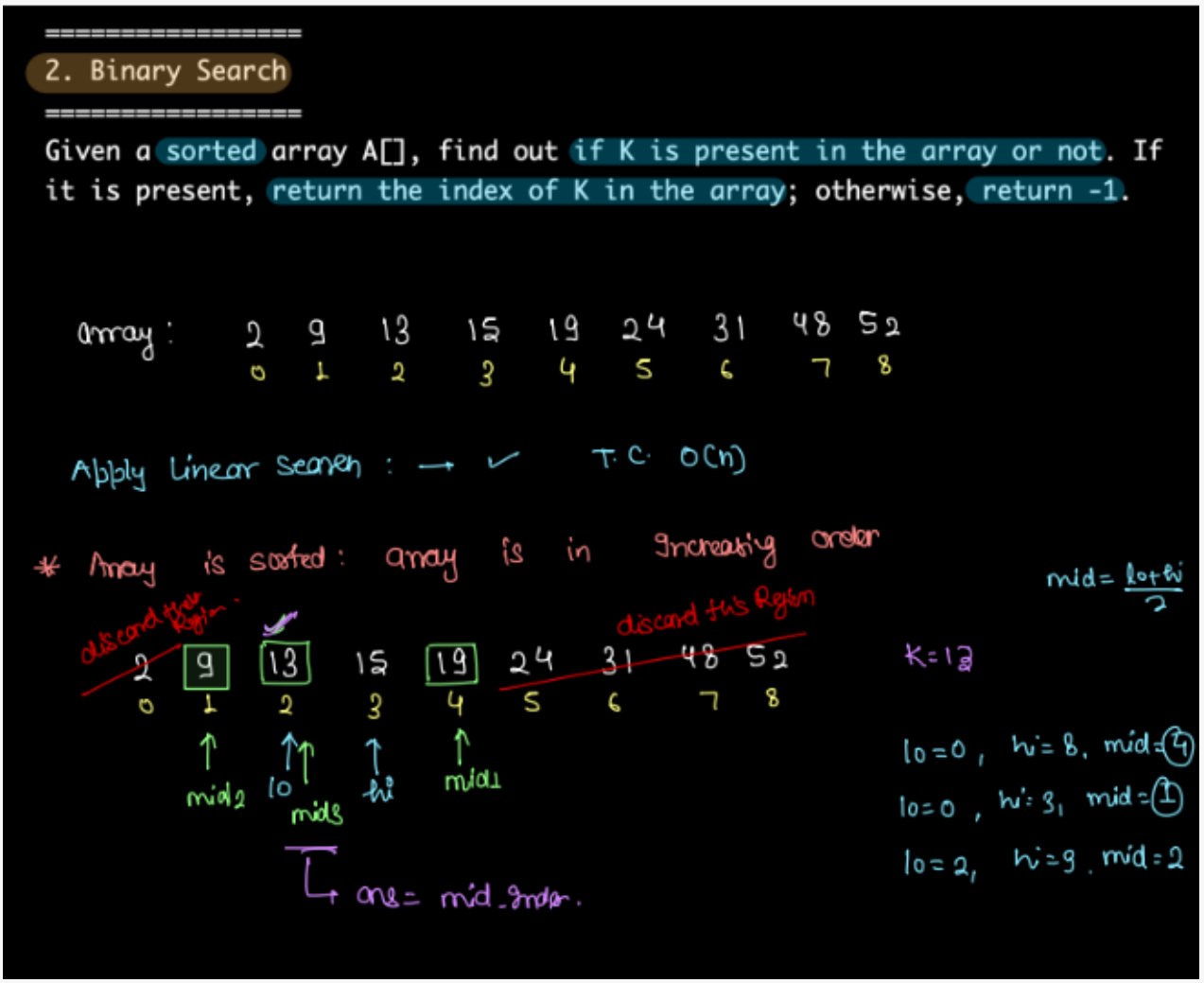
Linear Search -> It is simple search using loop



Binary Search ->In our search algorithm, we are using the Binary Search method, which has a time complexity of O(logN). This will work if array is already sorted in ascending order. If it in random order then we will use linior search.
To avoid integer overflow we can get middle index like
hi - (hi - lo) /2 ,
hi - hi / 2 + lo/2
Here's how the algorithm works:

1. We initialize two indices, "lo" and "hi," representing the lower and upper bounds of the search range. We calculate the midIndex as (hi + lo) / 2.
2. Inside a loop, we have three conditions to check:
    - If arr[midIndex] is equal to the userValue, we have found a match, so we return true and exit the loop.
    - If arr[midIndex] is greater than the userValue, we discard the right portion of the range by updating hi to midIndex - 1.
    - If arr[midIndex] is less than the userValue, we discard the left portion of the range by updating lo to midIndex + 1.
3. We continue the loop as long as the condition lo <= hi is true.

This algorithm allows us to efficiently search for a value in a sorted array by continually dividing the search range in half.



mid = $\frac{lo + hi}{2}$

① arr[mid] == K : found the answer, ans = mid, break ;

② arr[mid] > K : discard Right Region
hi = mid-1

③ arr[mid] < K : discard left Region
lo = mid+1

$(lo+hi)/2 \Rightarrow \frac{lo}{2} + \frac{hi}{2}$

$hi - \frac{(hi-lo)}{2}$

```
int Searching(int[] A, int K){
    int lo = 0;
    int hi = A.length-1;
    while(          ){
        int mid = (lo+hi)/2;
        if( A[mid] == K) {
            //data found
            return mid;
        } else if ( A[mid] > K) {
            //discard Right Region]
            hi = mid -1;
        } else if (A[mid] < K) {
            //discard left Region
            lo = mid +1;
        }
    }
    return -1;
}
```

K = 8

	0	1	2	3	4	5	6	7	8
A	2	4	8	13	19	29	38	42	49

mid2 lo mid3 hi mid1

lo	hi	mid	Action
0	8	4	arr[mid] > K ⇒ hi = mid-1, hi=3
0	3	1	arr[mid] < K lo = mid+1, lo=2
2	3	2	arr[mid] == K 2 -1 is ans.

First Occurrence of K

Given a sorted array A[], find out if K is present in the array or not. If it is present return index of first occurrence of K in the array; otherwise, return -1.

Expected T.C : O(logn)

2	2	3	4	5	5	5	7	7	8	9	12	19	20
0	1	2	3	4	5	6	7	8	9	10	11	12	13

K=3 , → ans=2

K=5 , → ans = 4

K=10 , → ans = -1

K → data to find

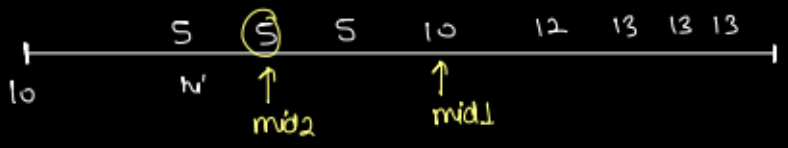
arr[mid] == K

ans = mid;

keep looking for K in left side.

hi = mid-1

K=5



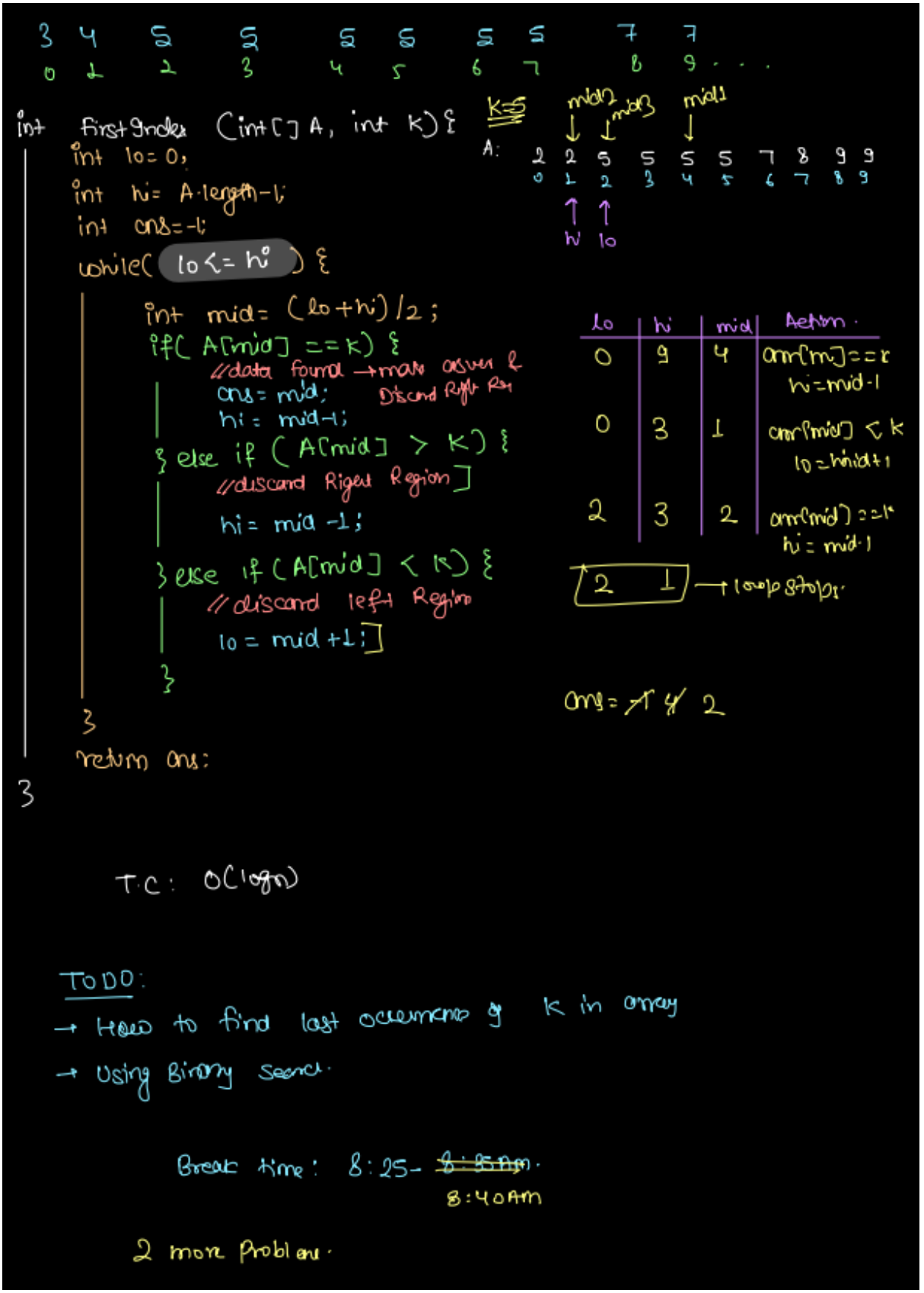
ans = mid2

mid =

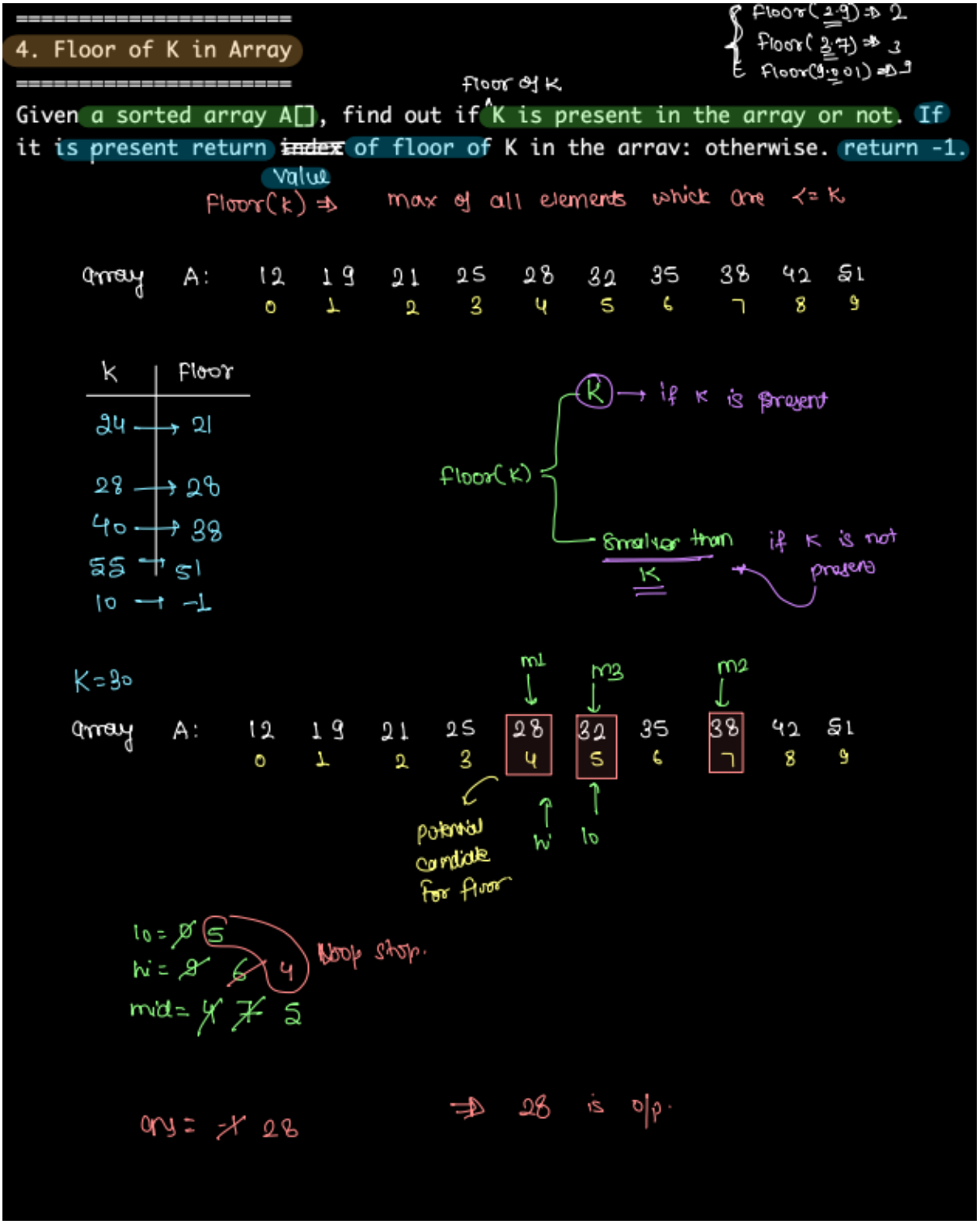
arr[mid] == k → ans = mid. Right Region Discard

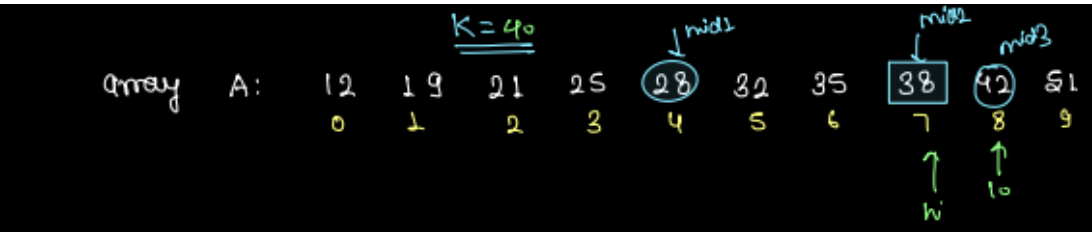
arr[mid] > K → Right Reg. Discard

arr[mid] < K → left Region Discard


3 4 5 5 5 5 5 5 7 7
0 1 2 3 4 5 6 7 8 9 . . .
int firstIndex (int[] A, int K){
int lo = 0;
int hi = A.length-1;
int ans=-1;
while(lo <= hi) {
int mid = (lo+hi)/2;
if(A[mid] == K) {
//data found → may be answer &
ans = mid; Discard Right Reg
hi = mid-1;
} else if (A[mid] > K) {
//discard Right Region
hi = mid -1;
} else if (A[mid] < K) {
//discard left Region
lo = mid +1;
}
}
return ans;
}
K=5
A:
mid2 mid3 mid1
2 2 5 5 5 5 7 8 9 9
0 1 2 3 4 5 6 7 8 9
hi lo
lo hi mid Action
0 9 4 arr[mid]==k hi=mid-1
0 3 1 arr[mid] < k lo=hi+1
2 3 2 arr[mid] > k hi=mid-1
2 1 → loop stops
ans = ~~1~~ 2
T.C: O(logn)
ToDO:
→ How to find last occurrence of K in array
→ Using Binary Search.
Break time: 8:25- ~~8:35Am~~.
8:40Am
2 more Problems.

Floor Of K in Array






```
if ( arr[mid] == k) {
|     ans= arr[mid];
|     break;
} else if ( arr[mid] < k) {
|     // potential candidate → arr[mid]
|     ans = arr[mid]
|     lo = mid +1;
} else {
|     hi=mid-1;
}
```

lo = ~~0~~ 8
hi = ~~9~~ 7
] loop stop.

mid = ~~4~~ 8

ans = ~~28~~ 38 → o/p:

TODO: ① Dry Run for floor

array A:

12	19	21	25	28	32	35	38	42	51
0	1	2	3	4	5	6	7	8	9

K = 12

K = 50

K = 28

K = 55

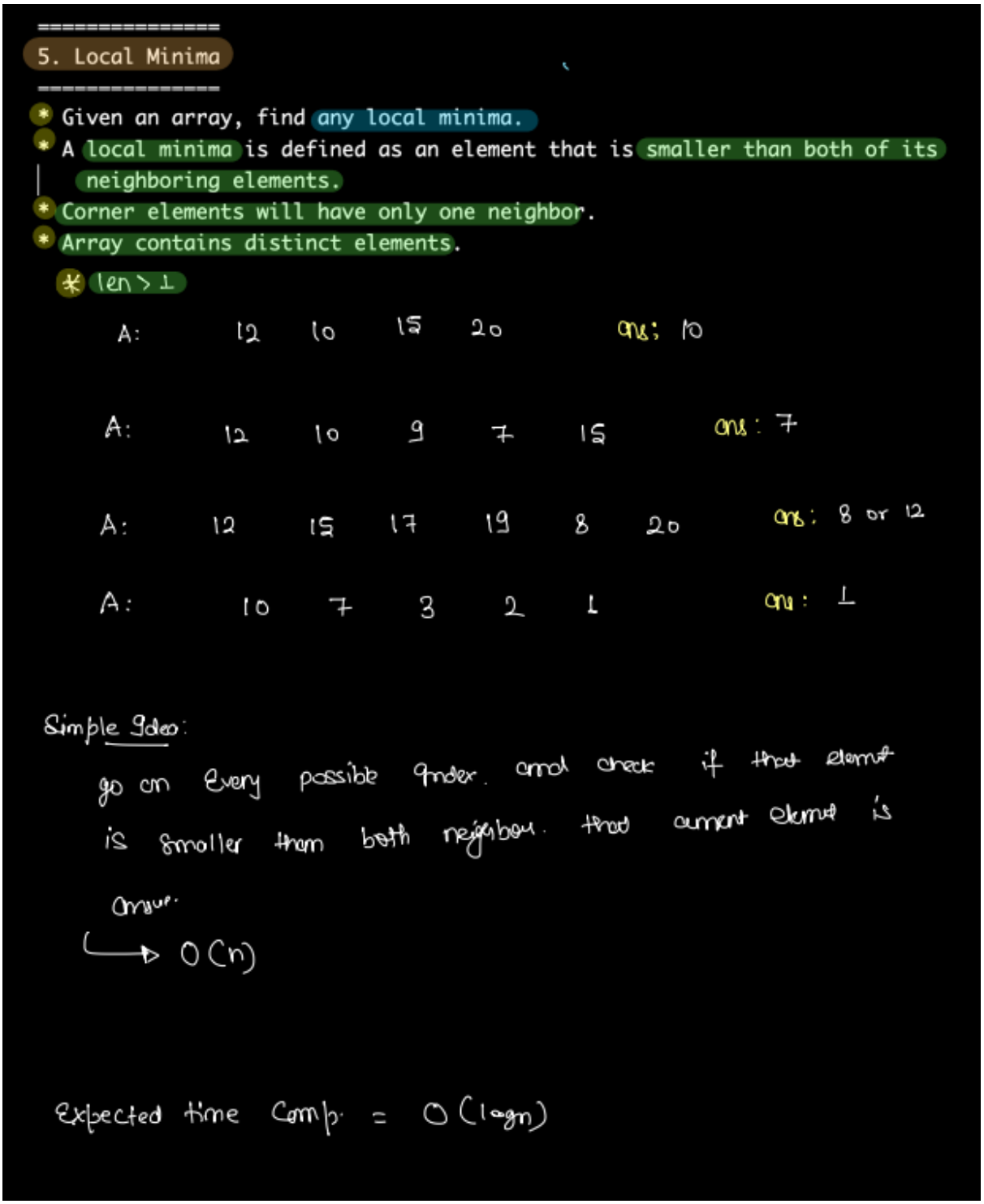
② Try to solve for ceil value of K

ceil value of K → min from all max.

```
int floorOfK (int[] A, int K){
    int lo= 0;
    int hi= A.length-1;
    int ans=-1;
    while( lo<= hi ) {
        int mid= (lo+hi)/2;
        if( A[mid] == K) {
            ans= A[mid];
            break;
        } else if ( A[mid] > K) {
            //discard Right Region
            hi= mid -1;
        } else if (A[mid] < K) {
            ans= A[mid];
            lo= mid +1;
        }
    }
    return ans;
}
```

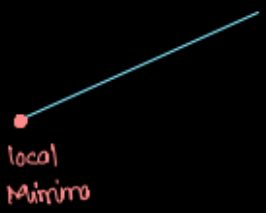
T.C: O(logn)

Local Minima

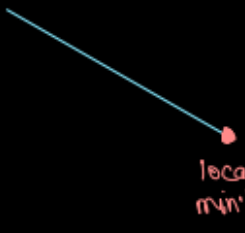


→ local minima will always present in the array :

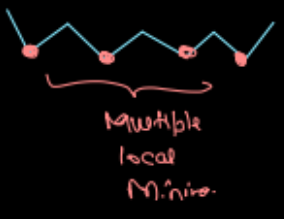
1. Increasing order



2. Decreasing order



3. Random Arrayst



arr[mid-1] < arr[mid] ||
arr[mid] < arr[mid+1]

→ arr[mid] => ans.

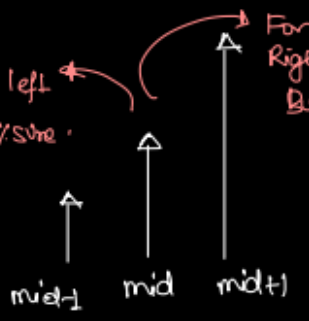
this middle element itself is answer.

arr[mid-1] < arr[mid]
|| arr[mid] < arr[mid+1]

→ left side

For left 100% sure.

For Right side (May be there is ans)
But we are not 100% sure.

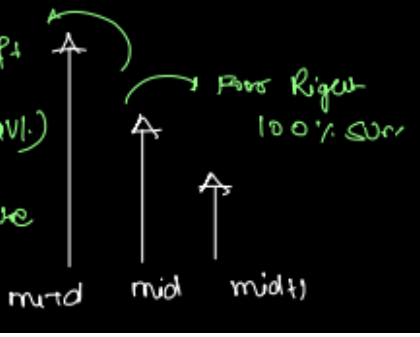


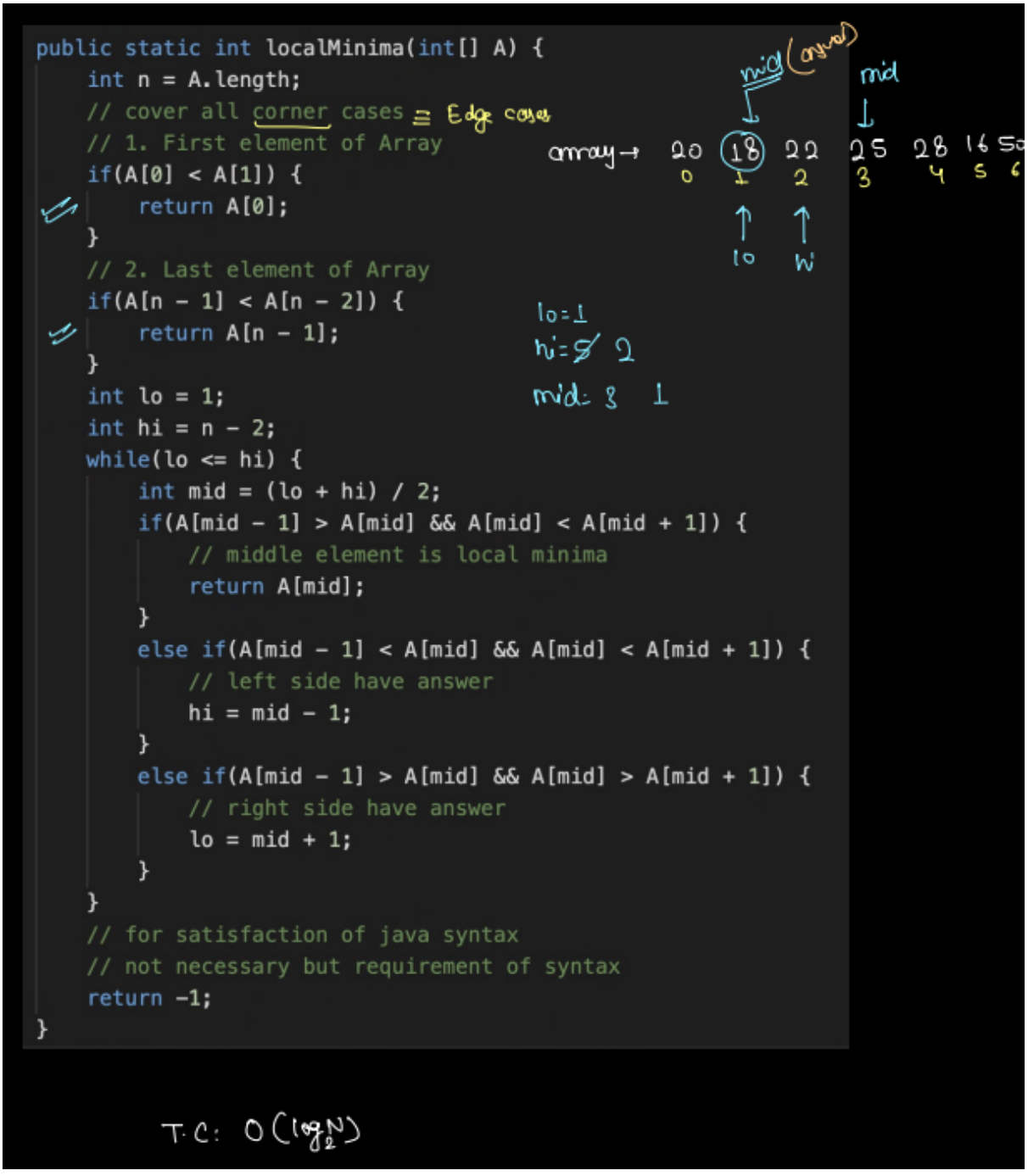
arr[mid-1] > arr[mid]
|| arr[mid] > arr[mid+1]

→ Right side

For left side
(May ans. avl.)
But not 100% sure

For Right
100% sure





DSA: Two Pointers - 22 - June 2023