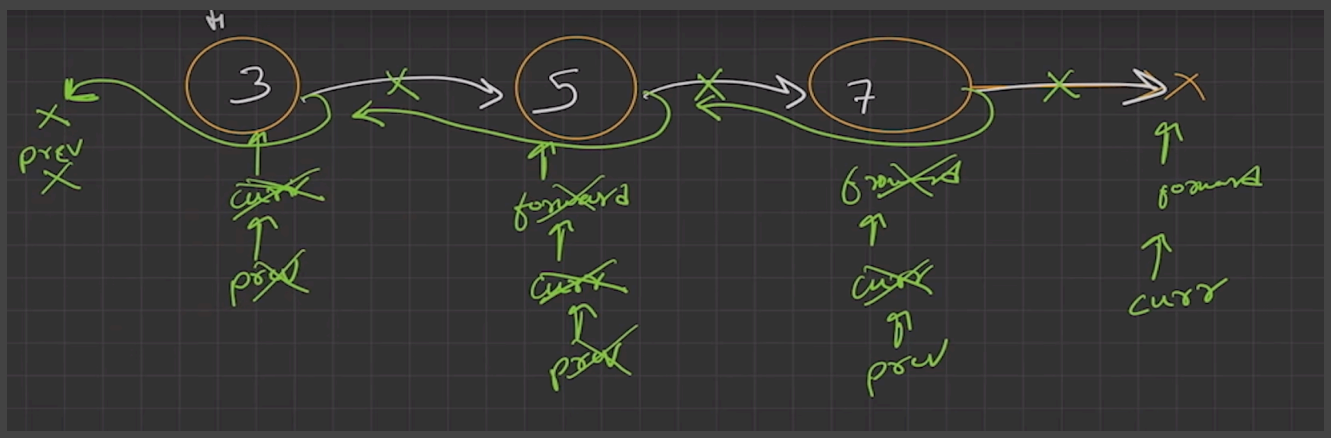
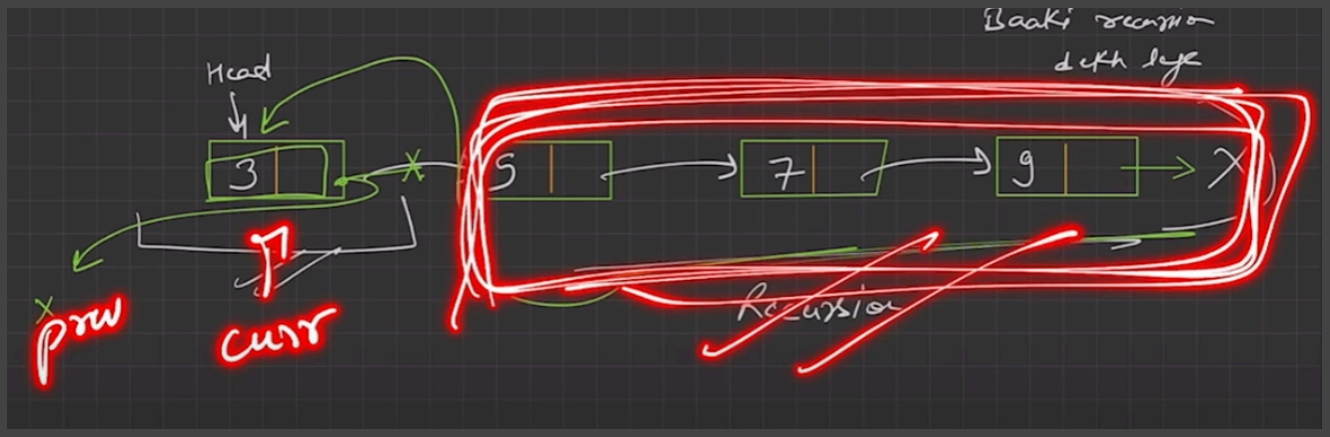
Linked-List

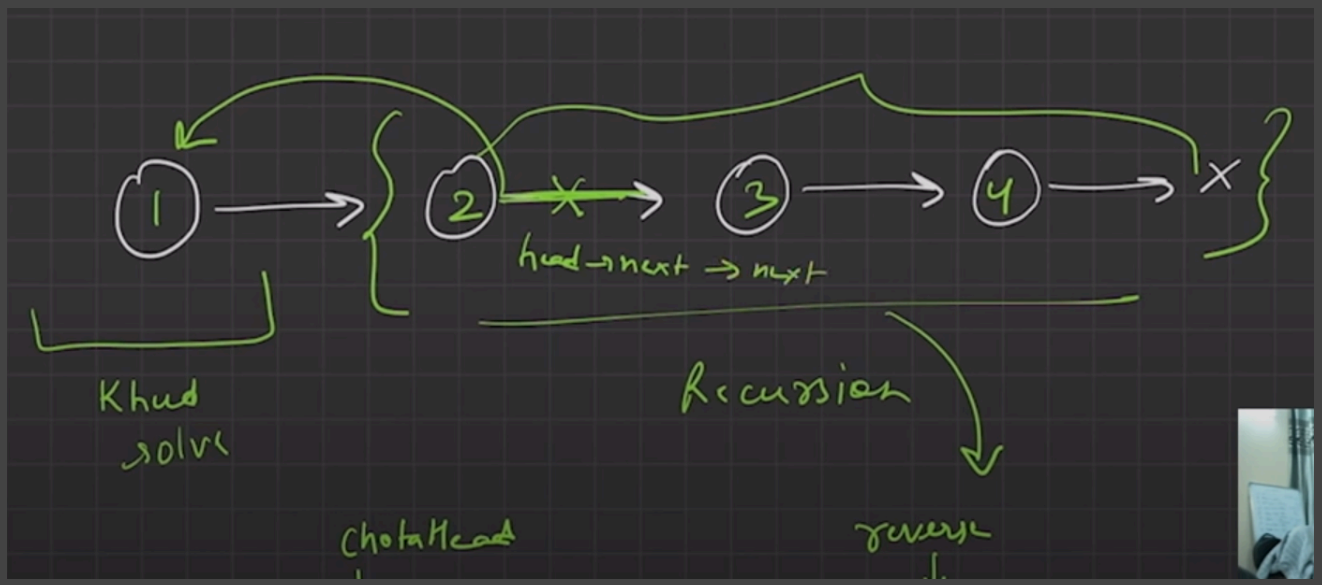
Reverse A Singly Linked List Approach-I:



Reverse A Singly Linked List Approach-II:

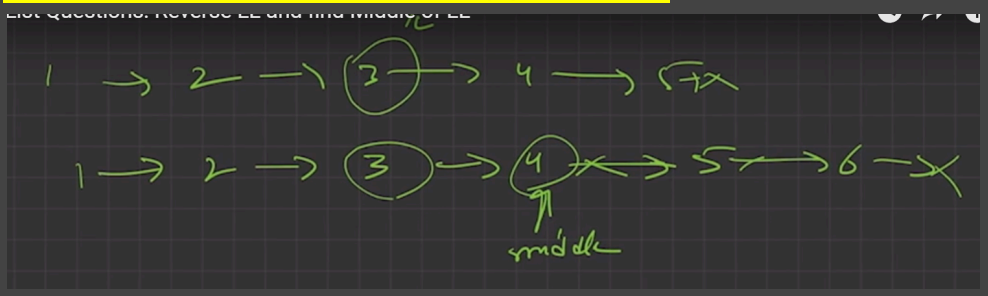


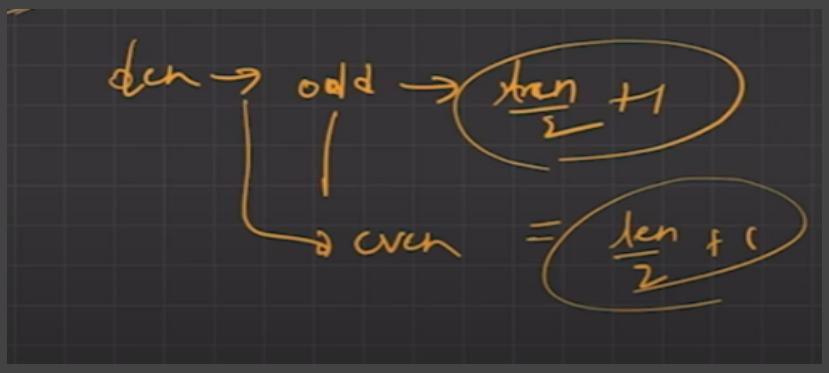
Reverse A Singly Linked List Approach-III:



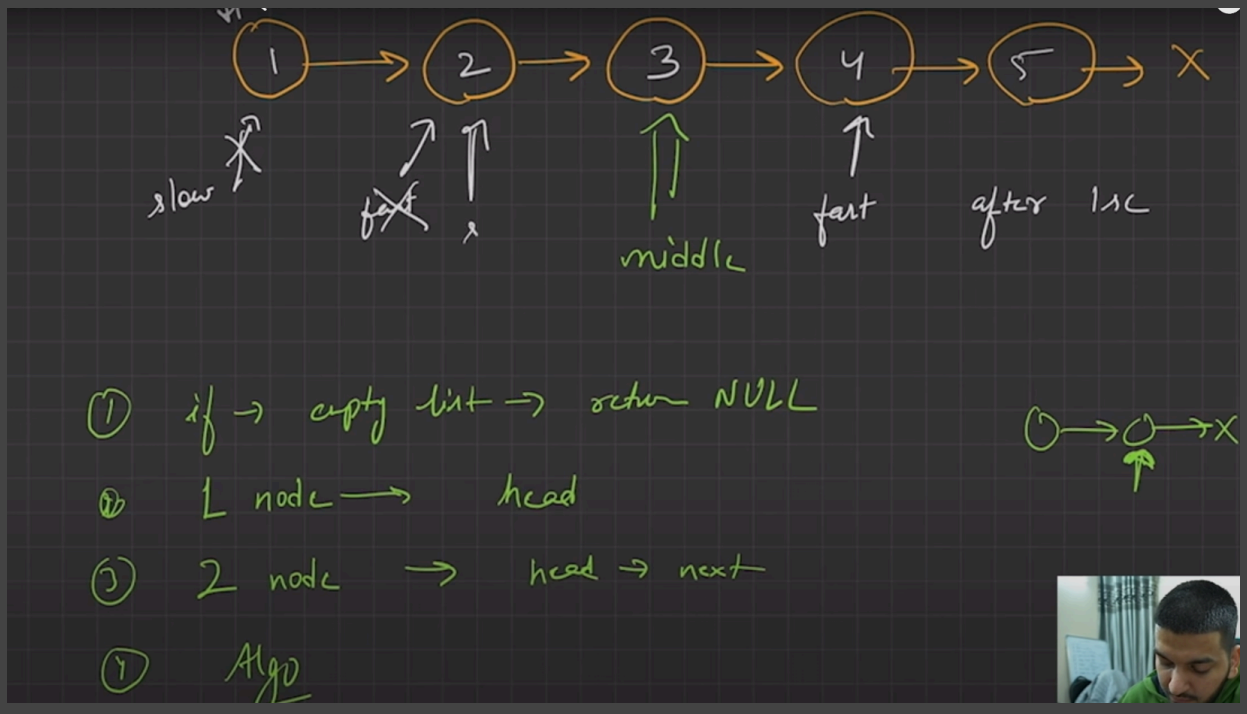
Reverse A Doubly Linked List

Find Middle In A Linked List Approach-I:

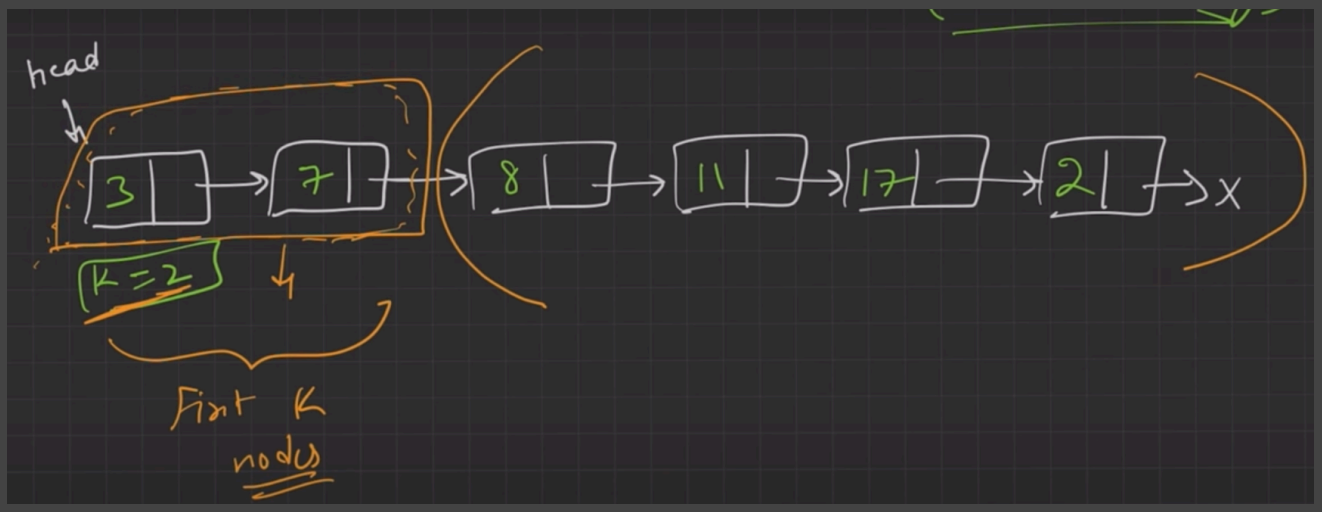


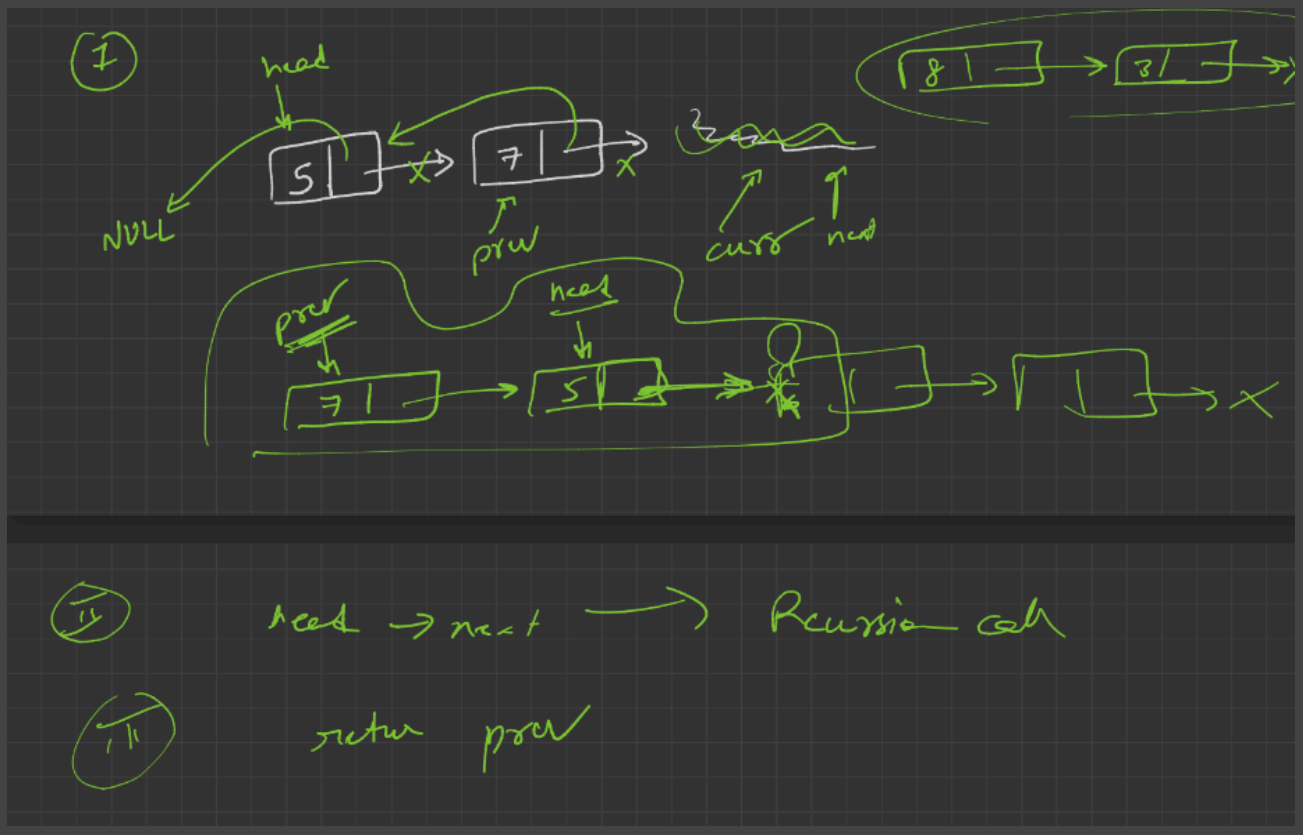


Find Middle In A Linked List Approach-II:

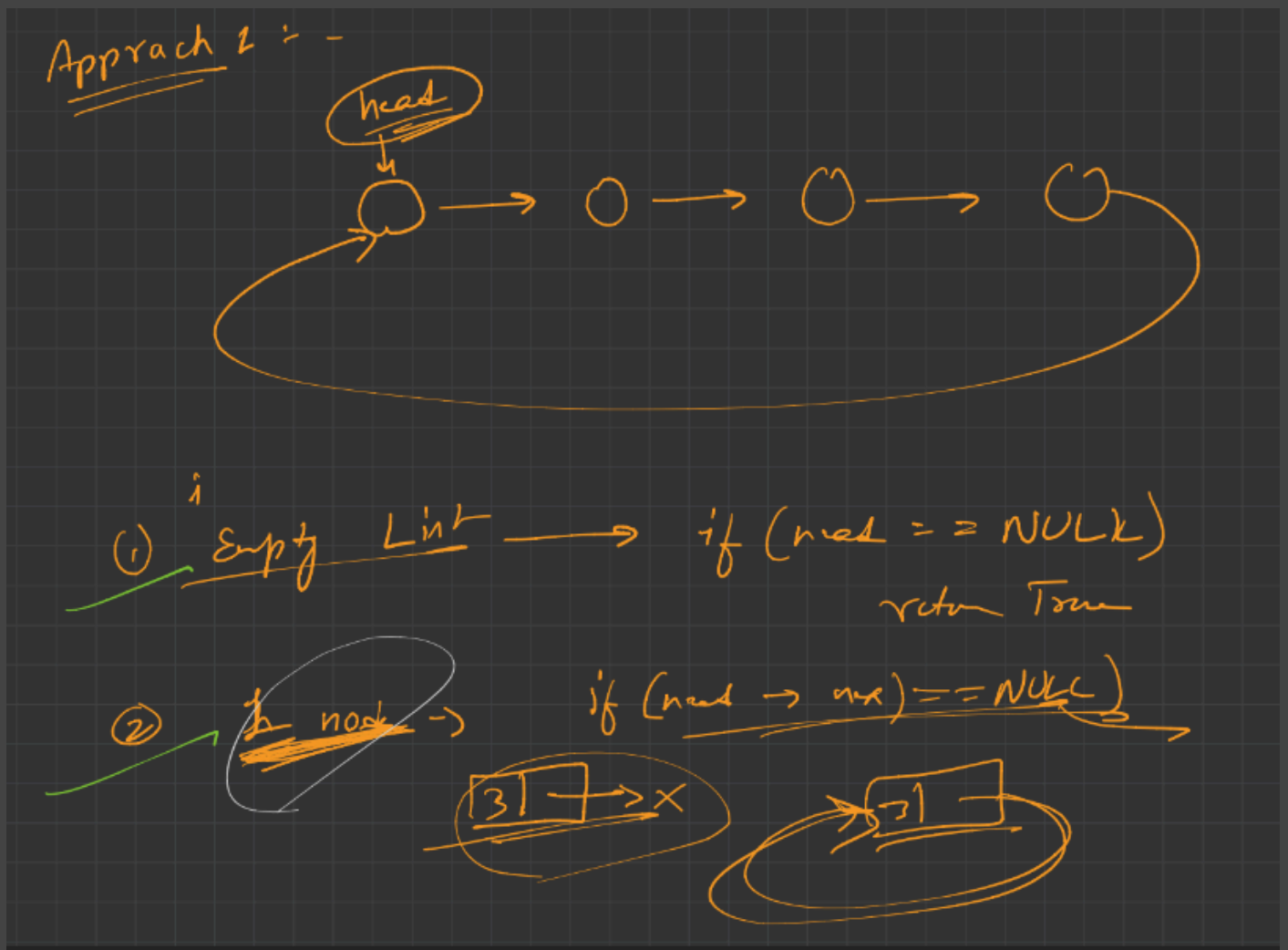


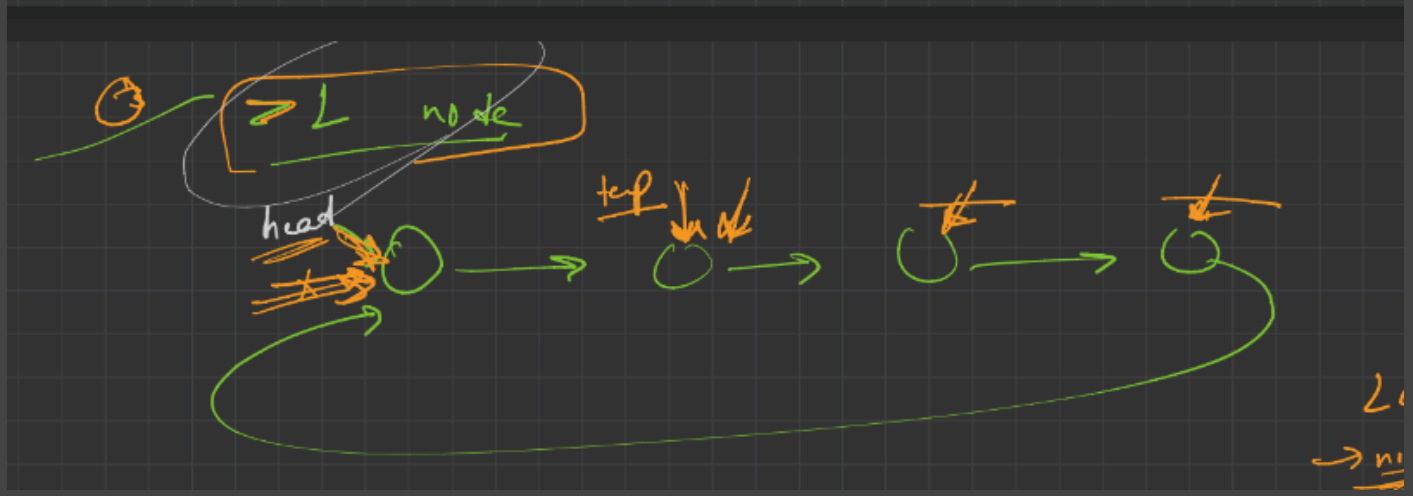
Reverse Linked List In 'K' groups:



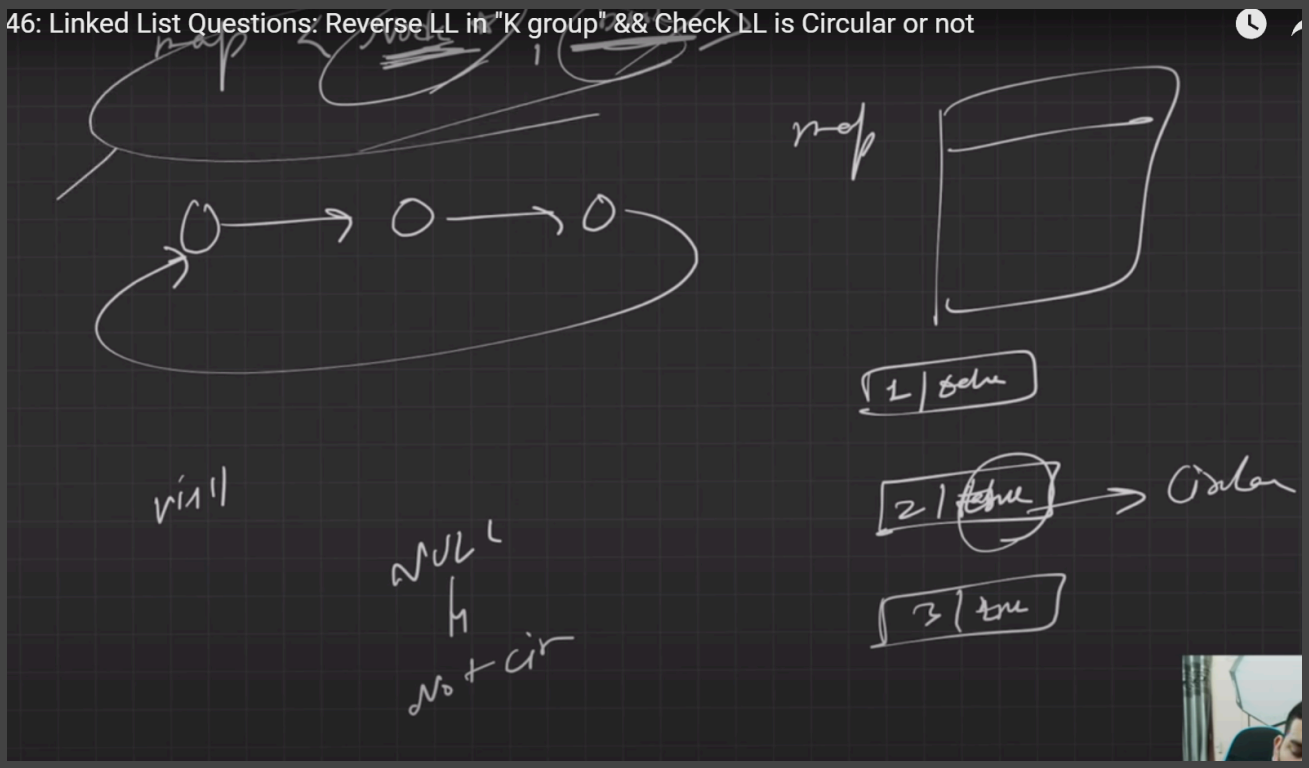


Check Whether A Linked List is Circular or Not Approach-I:

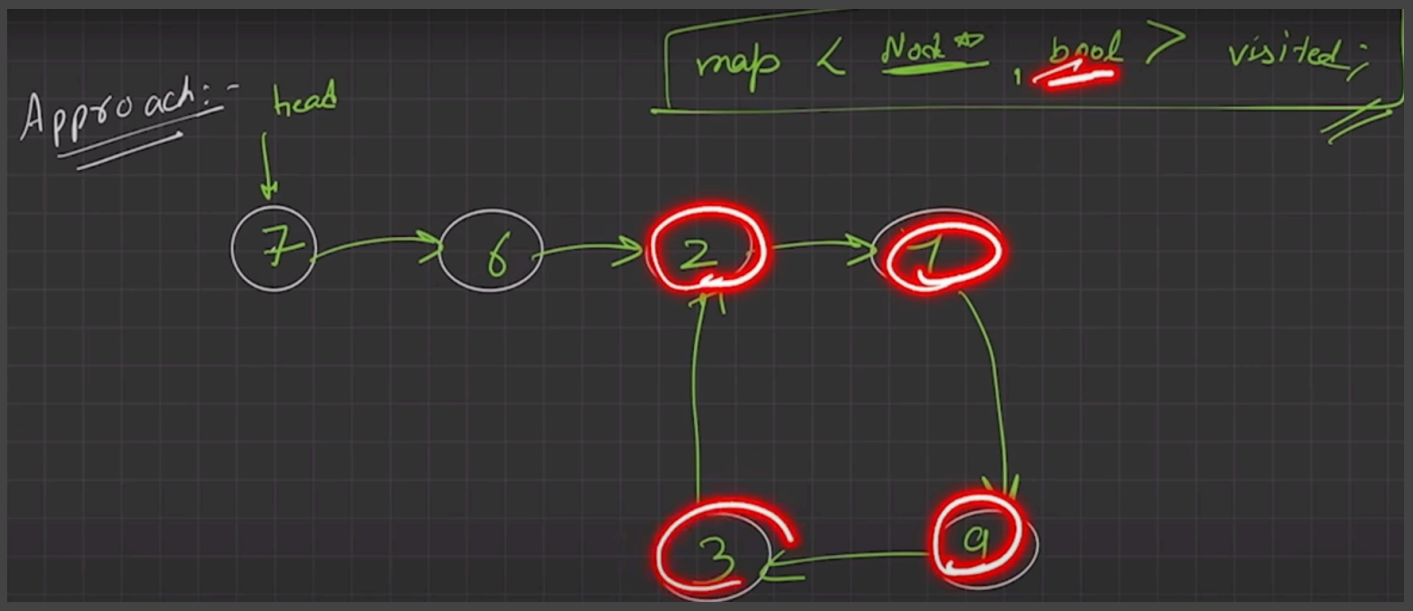




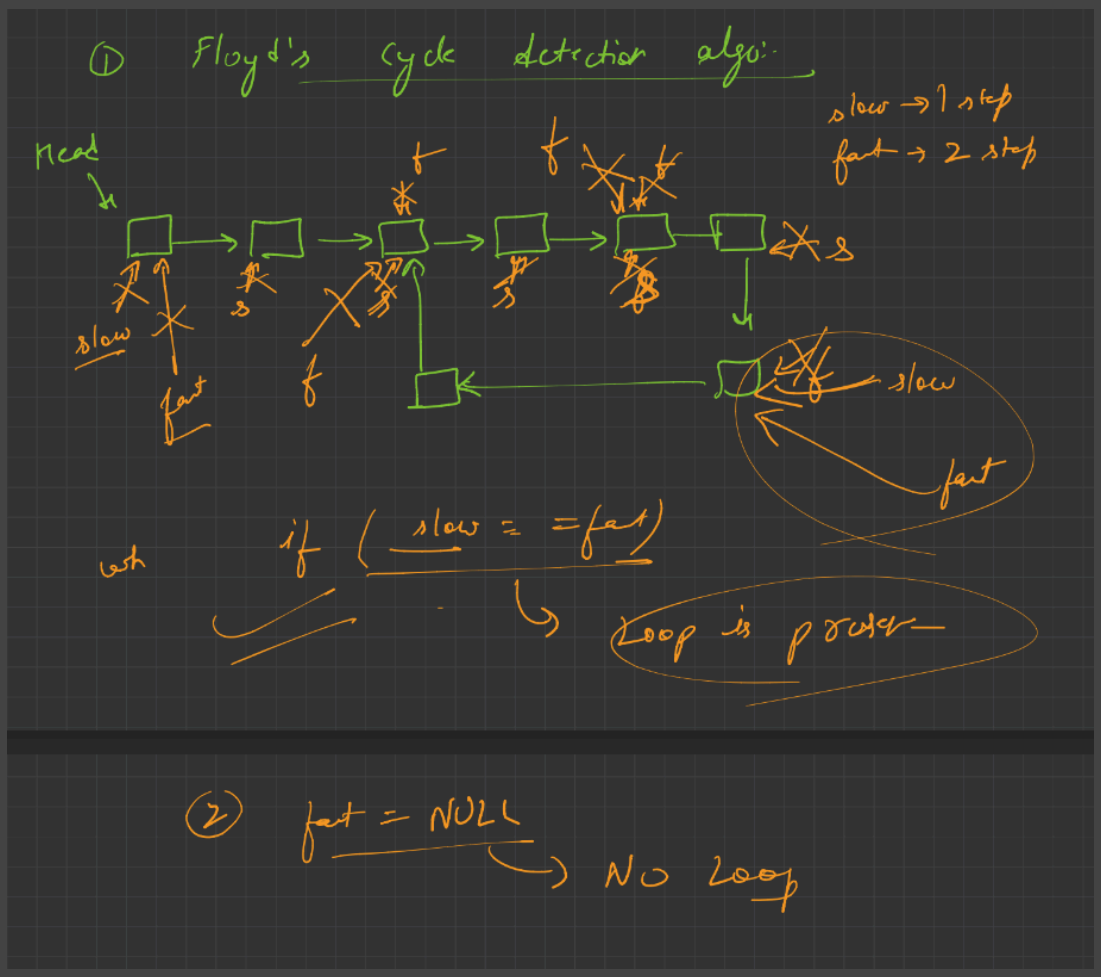
Check Whether A Linked List is Circular or Not Approach-II:

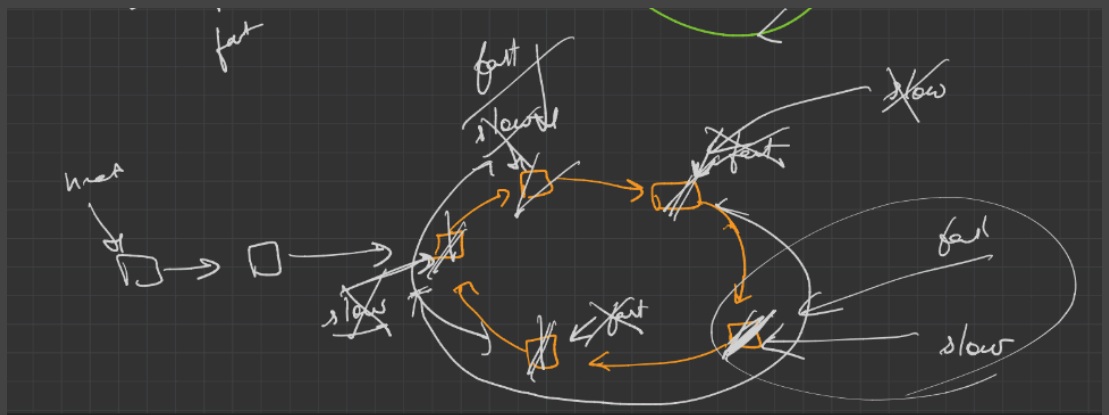


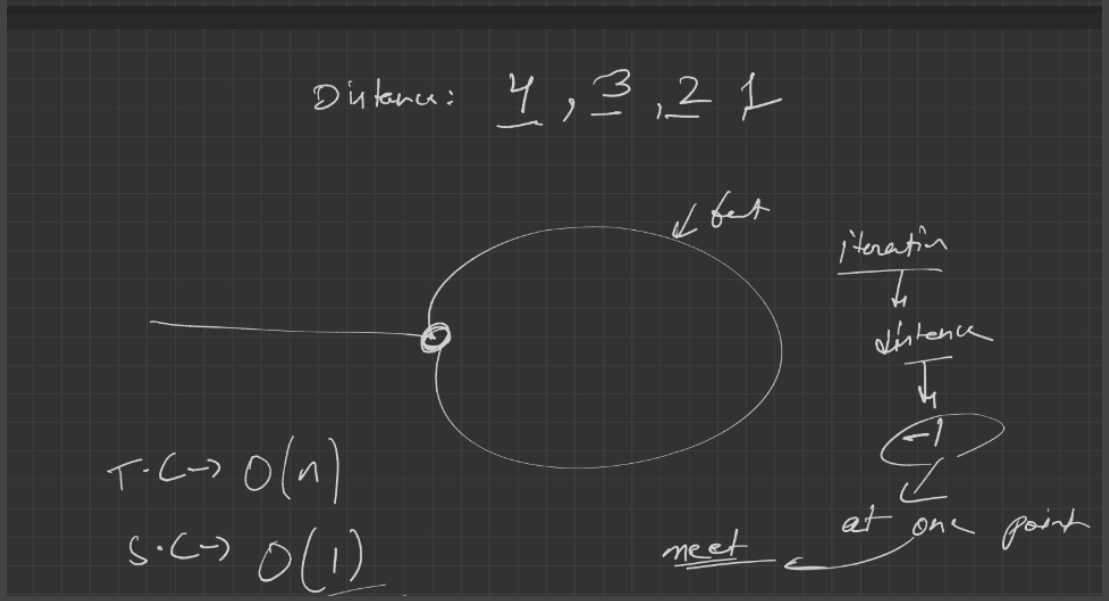
Detect Whether A Linked List Forms a Loop or Not:



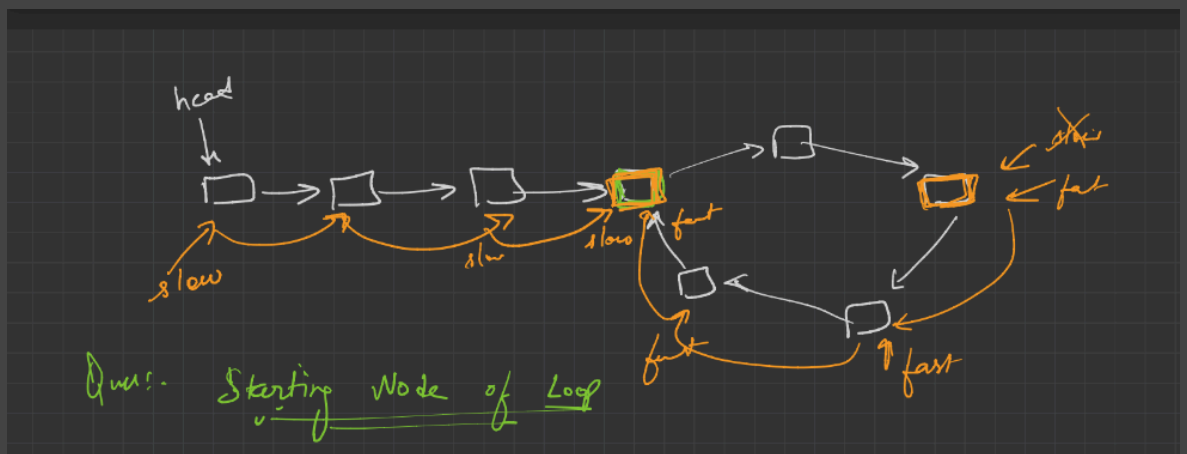
Floyd's Cycle Detection:







Find The Starting Node Of The Loop:



Ques: Starting Node of Loop

Approach:-

I → FCD Ifo → Point of Intersection

II → slow = head

↳ slow, fast → same pace
1, 1/1

↙

while(slow == fast)

↳ starting point of Loop

Distance by fast pointer = 2 * Distance by slow pointer

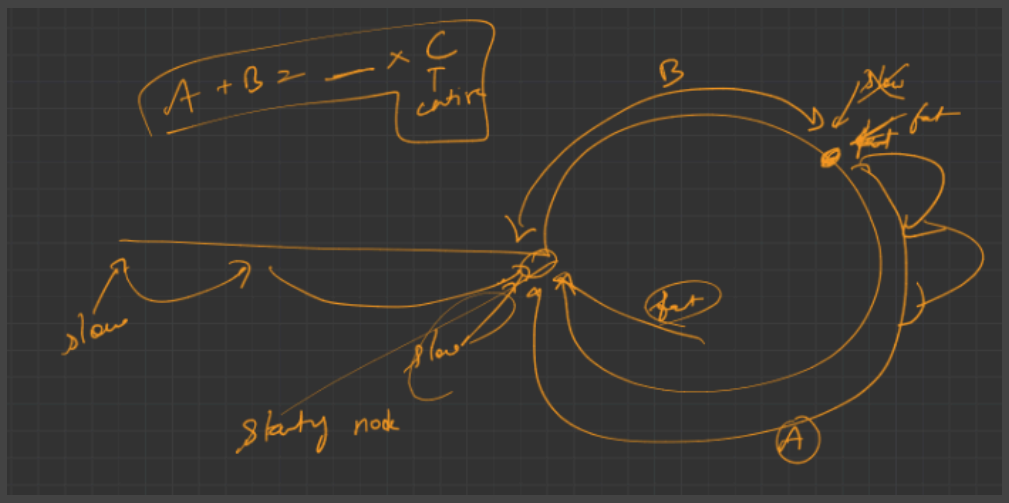
$$(A + x * C + B) = 2 * (A + y * C + B)$$

$$A + xC + B = 2A + 2yC + 2B$$

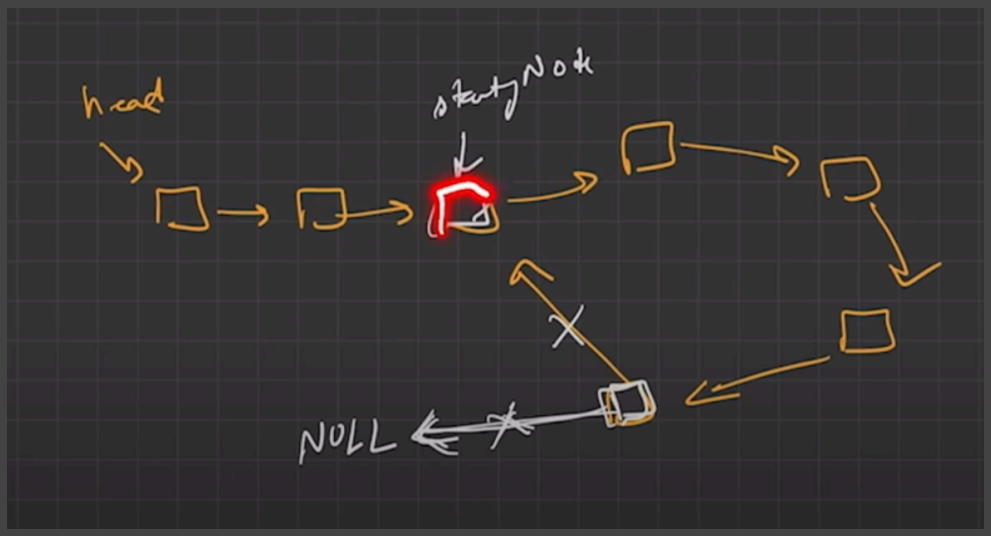
$$C(x - 2y) = A + B$$

$$\frac{C(x-2y)}{K} = A + B$$

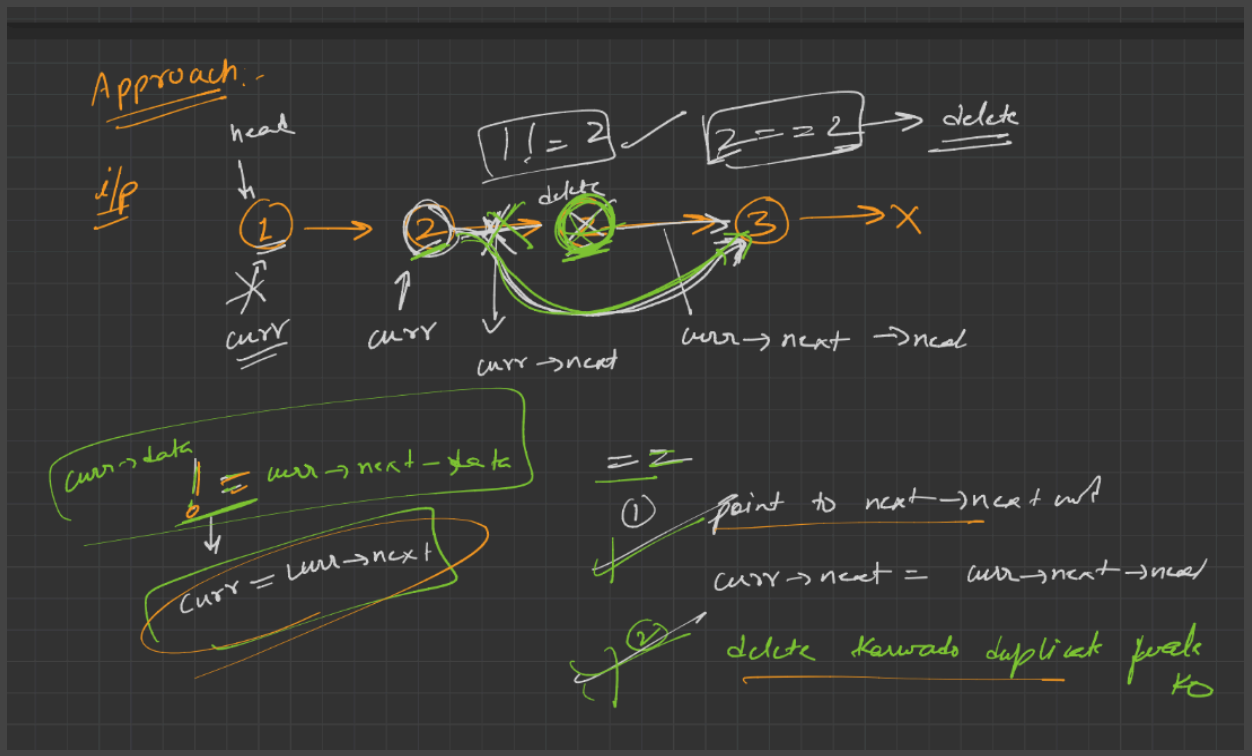
A + B = K times C



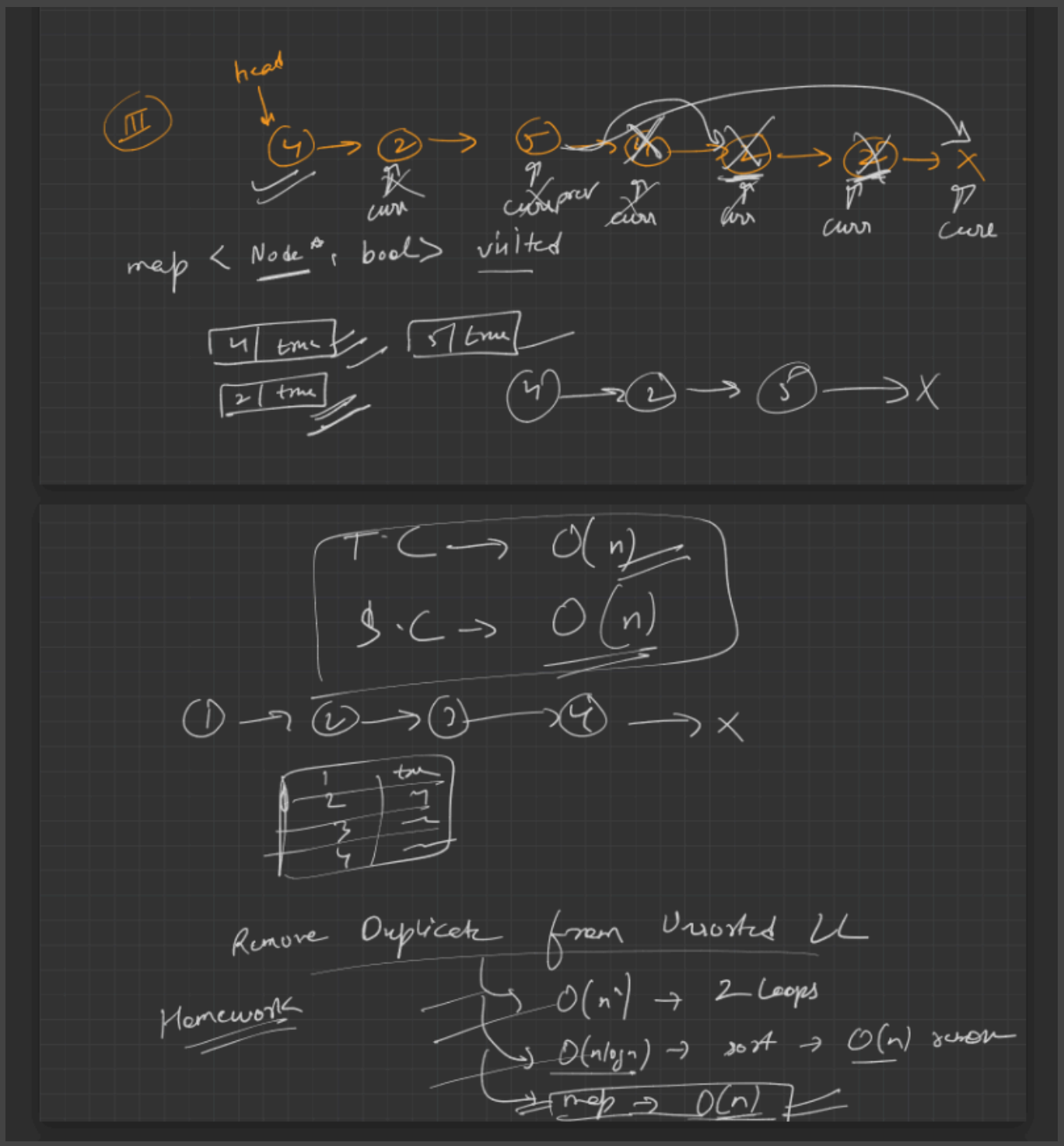
Remove The Loop:



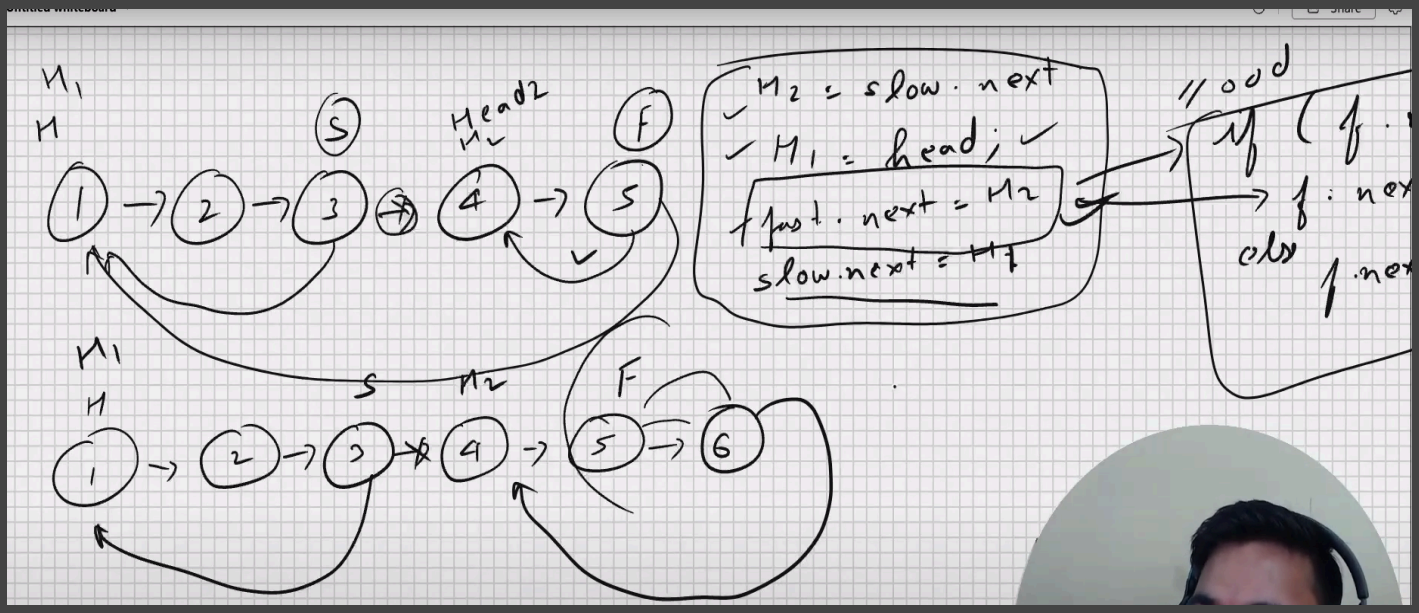
Remove Duplicates From A Sorted Linked List:



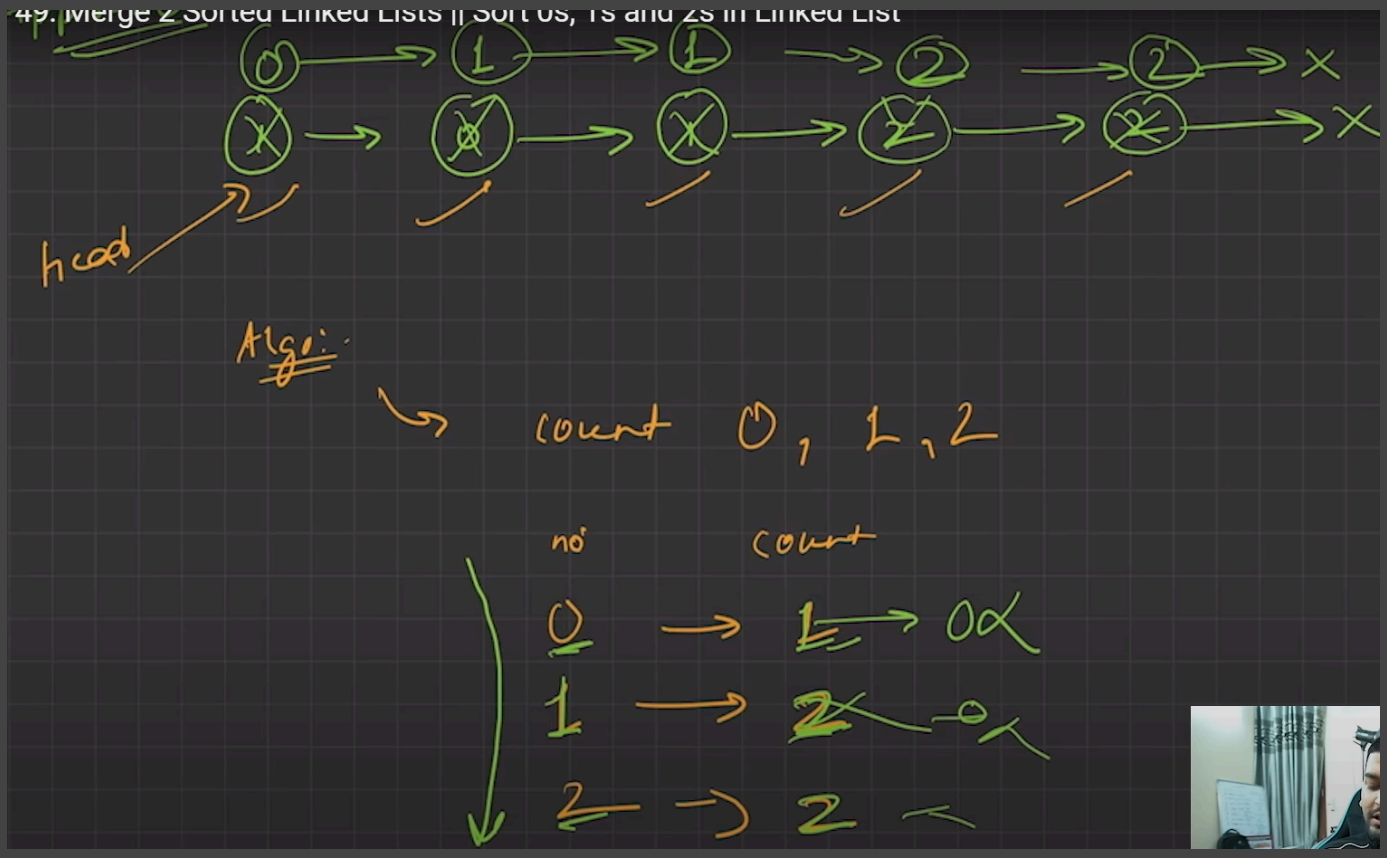
Remove Duplicates From A Unsorted Linked List:



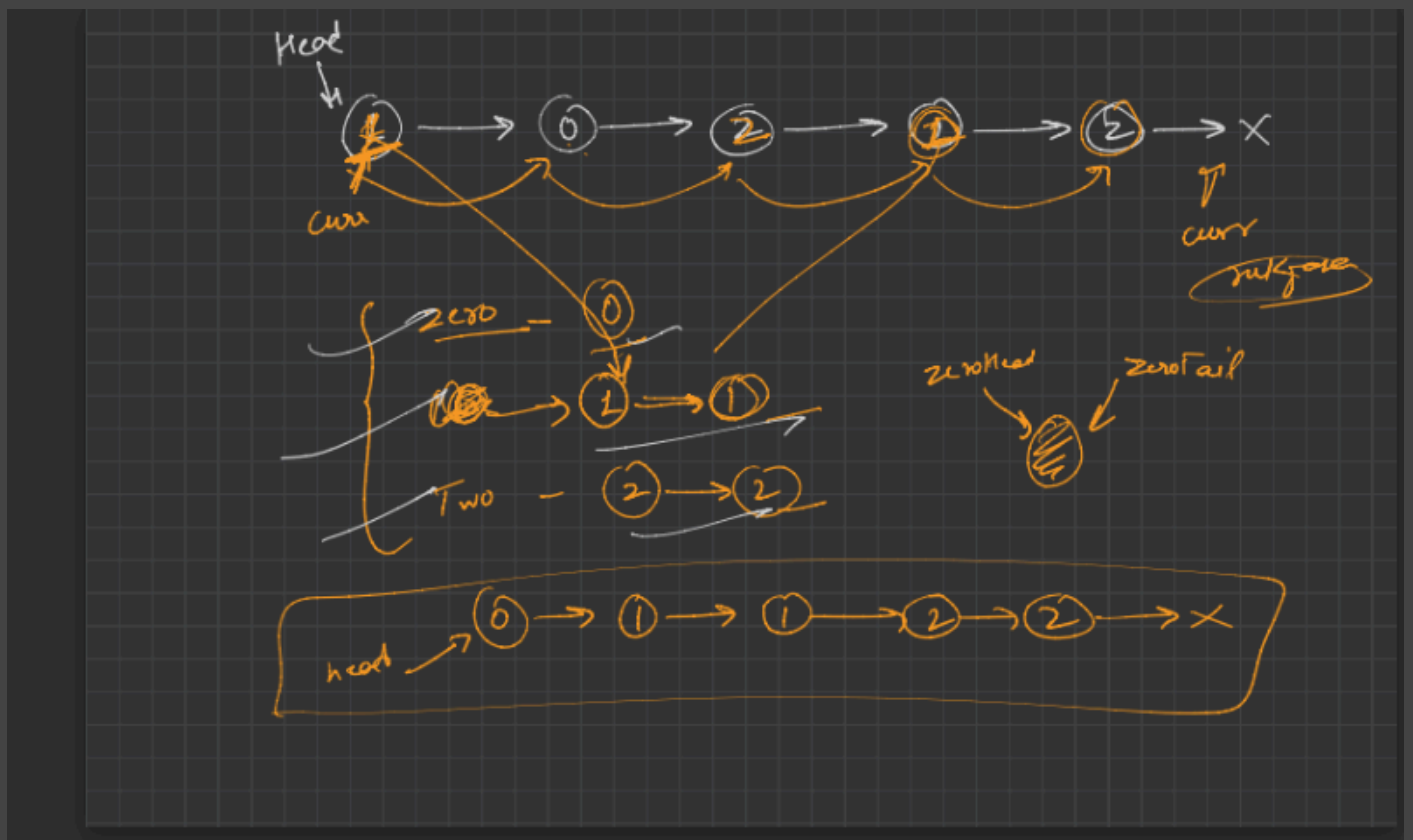
Split A Circular LL Into Two Halves:

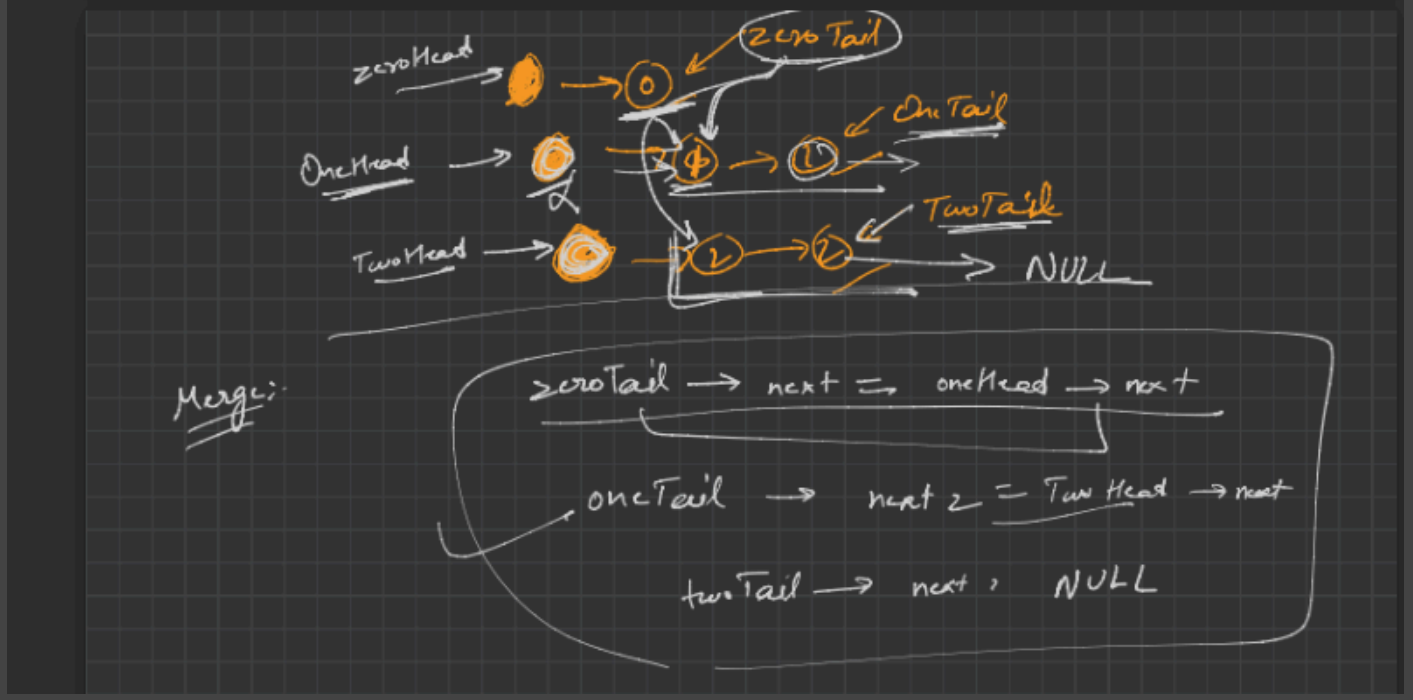


Sort 0s,1s & 2s In A LL Approach-I:

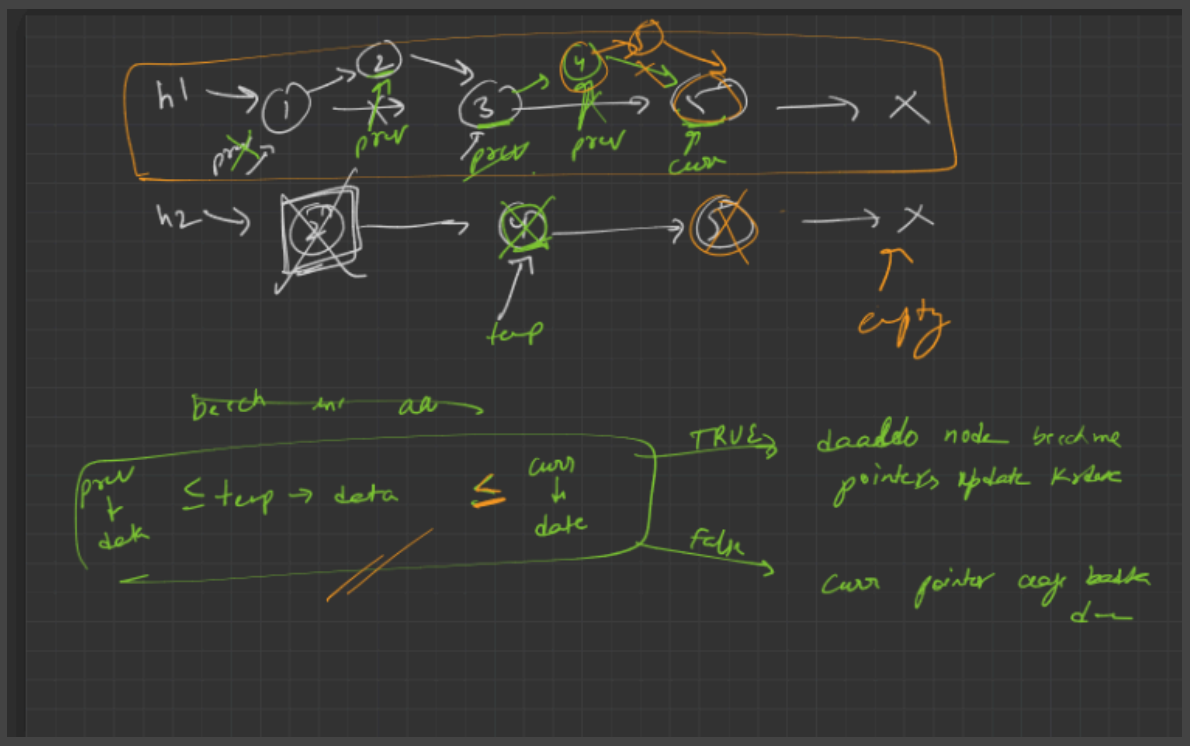


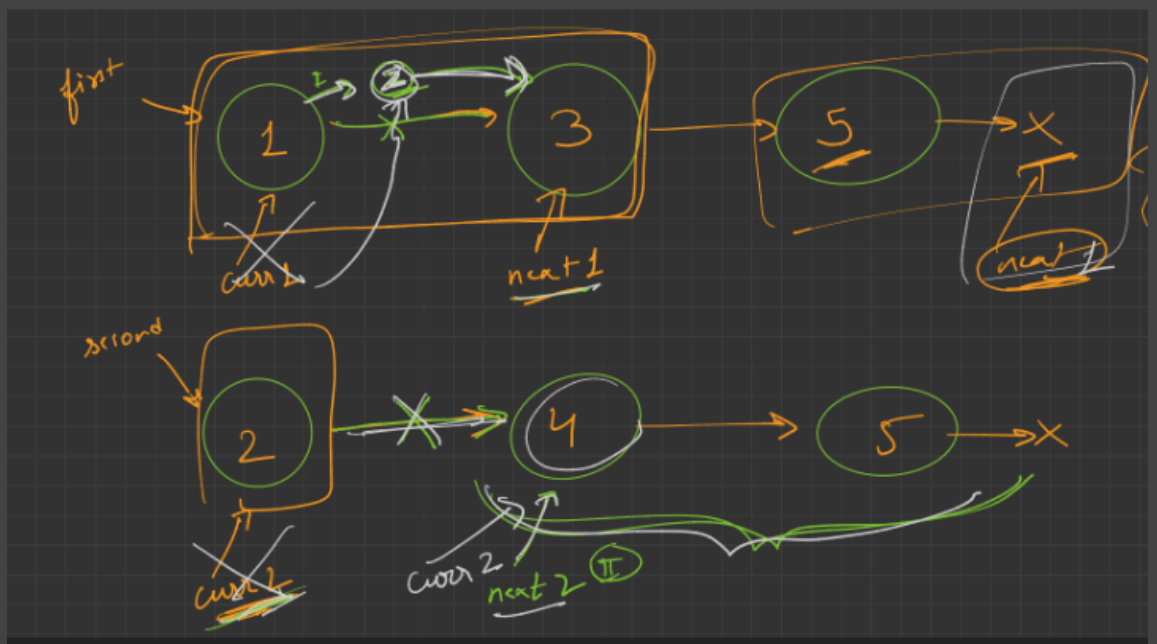
Sort 0s,1s & 2s In A LL Approach-II:





Merge Two Sorted LL:





I → curr1 → next = curr2

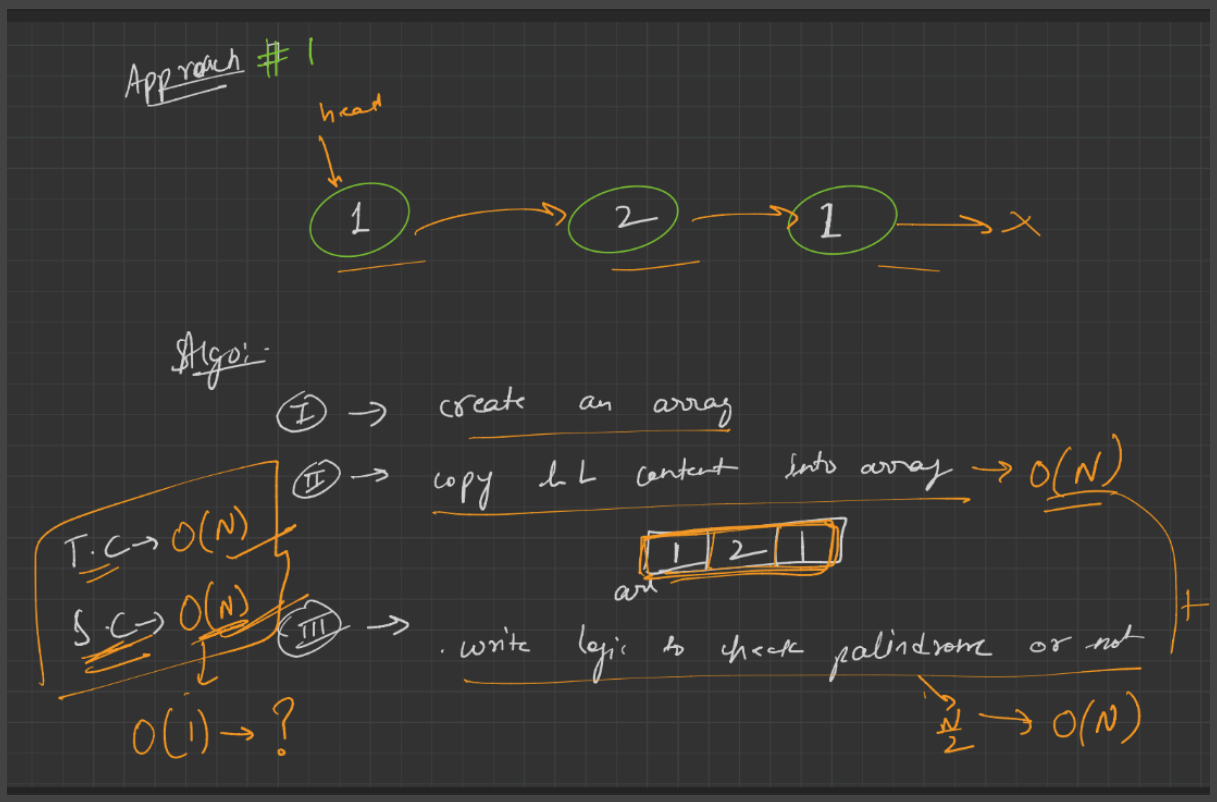
II → next2 = curr2 → next

III → curr2 → next = next 1

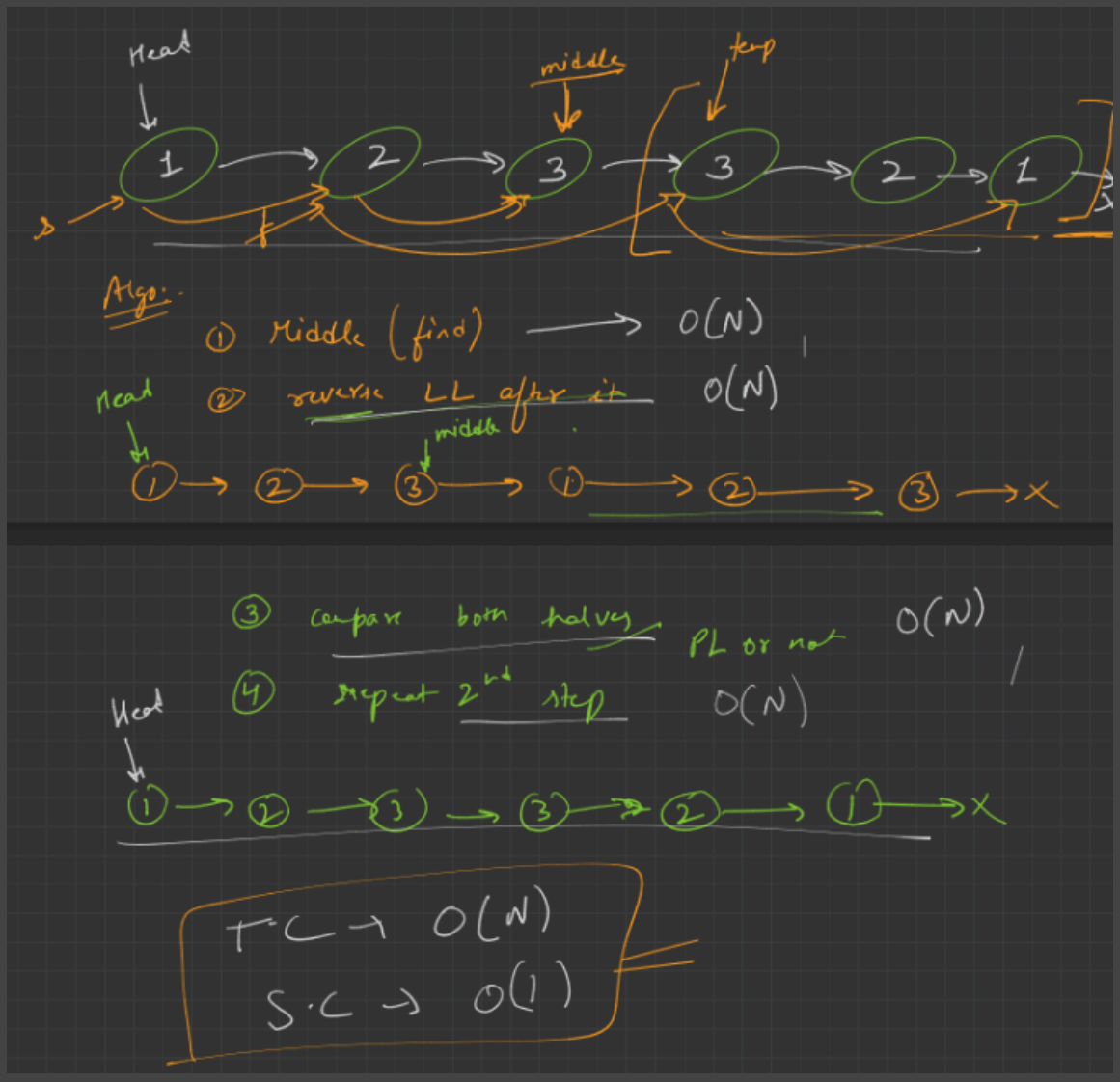
IV ⟹ curr 1 = curr 2

V = Curr 2 = ne

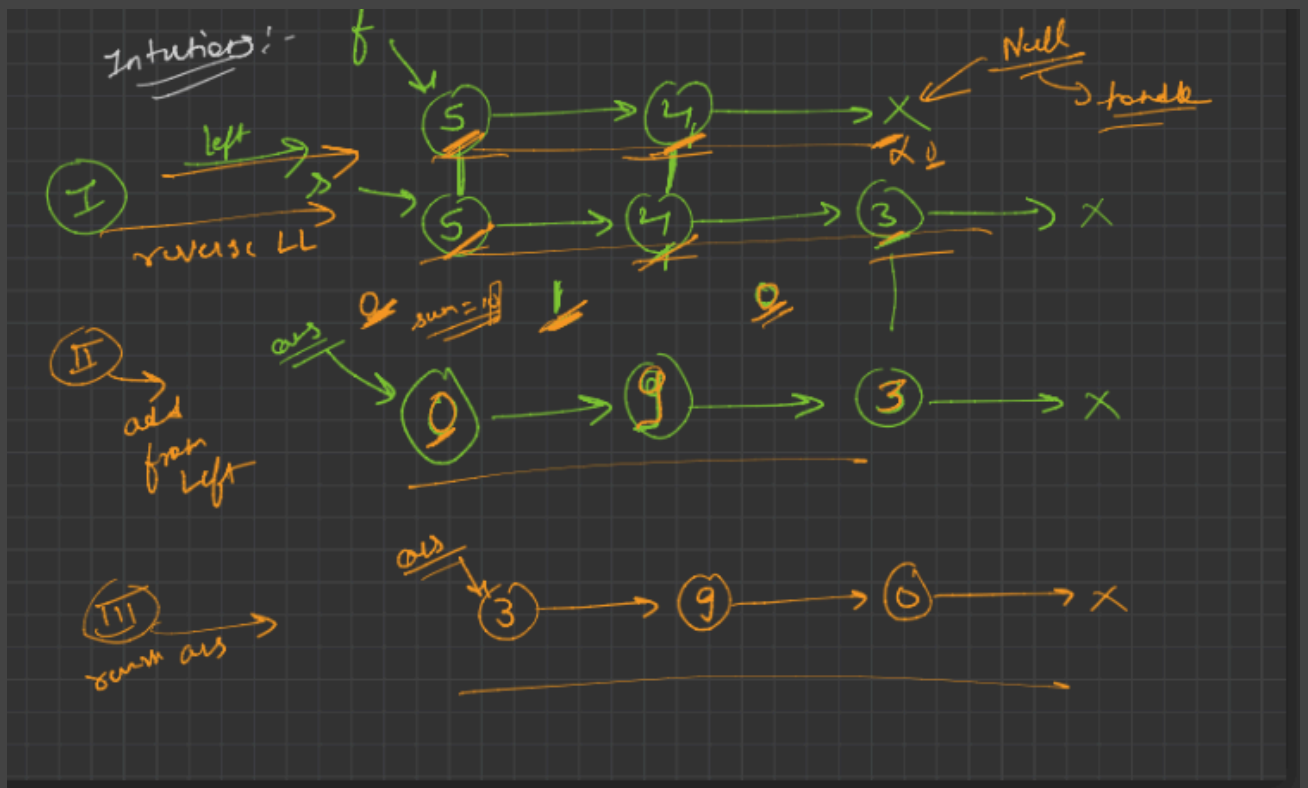
Check Palindrome In LL Approach-I:



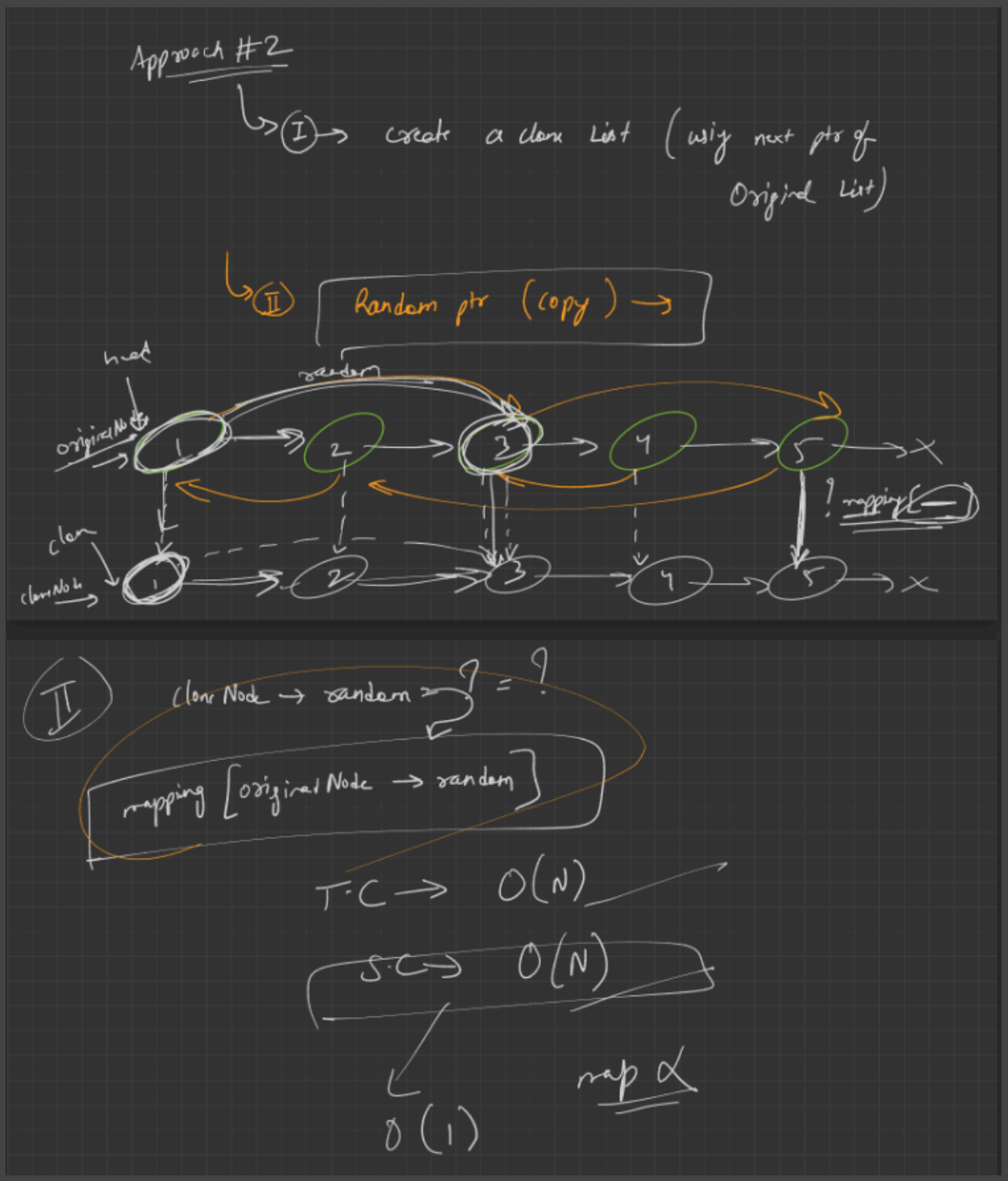
Check Palindrome In LL Approach-II:



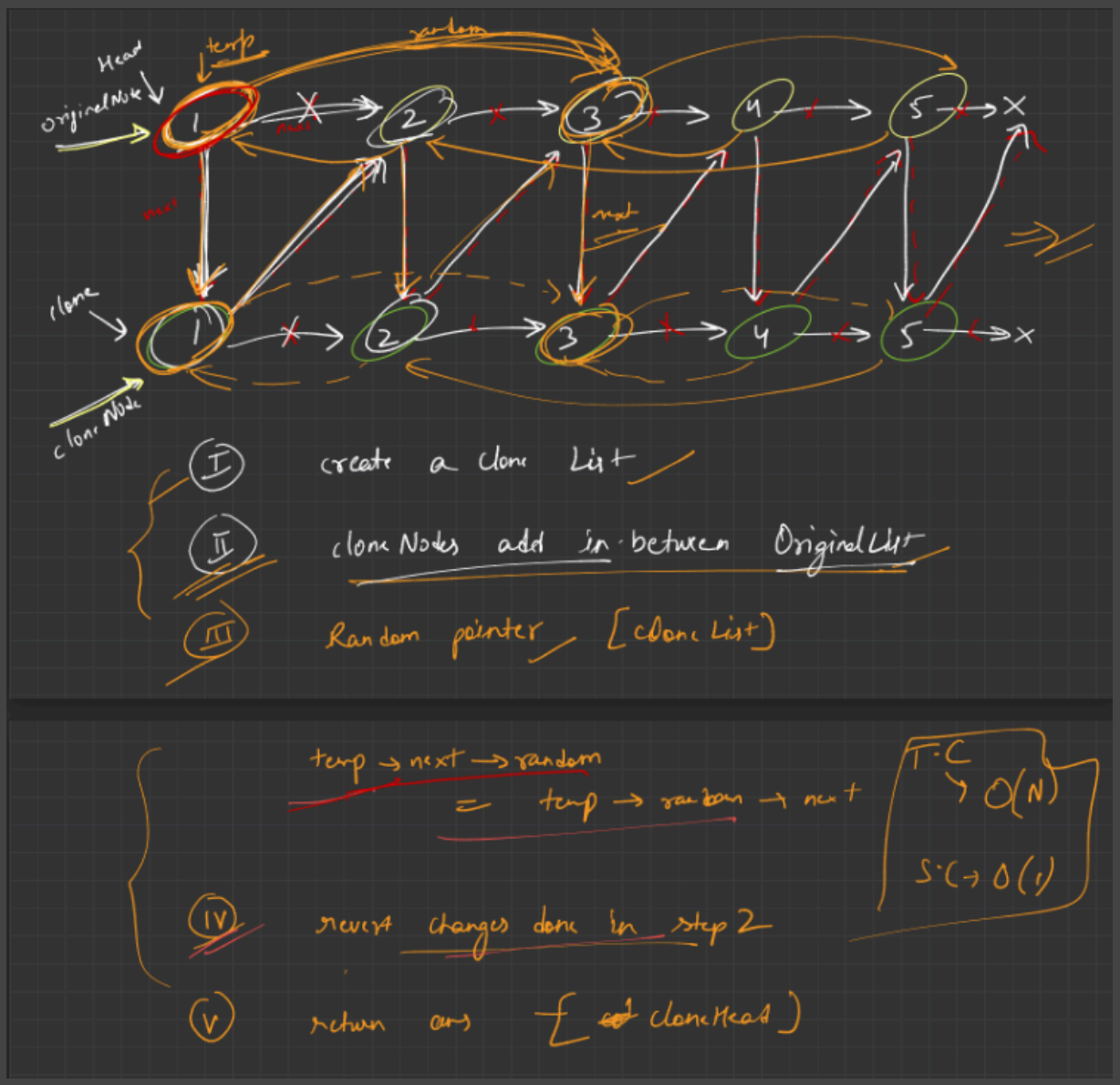
Add Two Numbers Represented By Linked List:

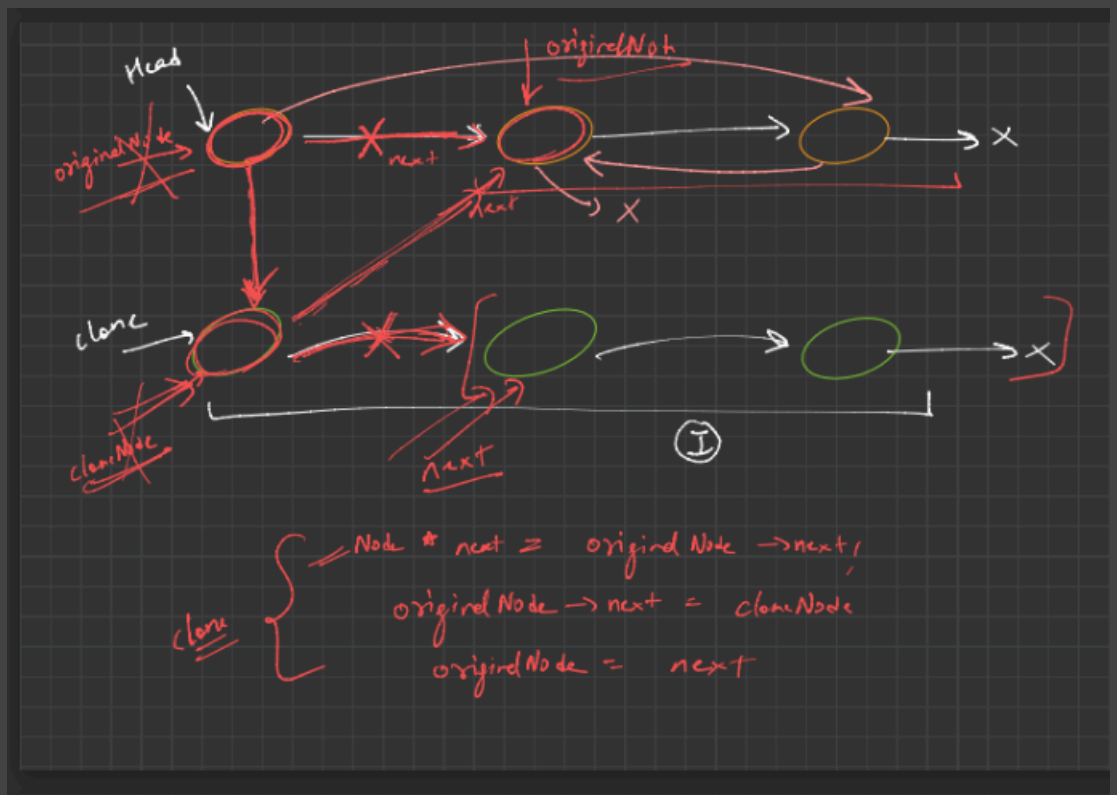


Clone A Linked List Approach-I:



Clone A Linked List Approach-II:

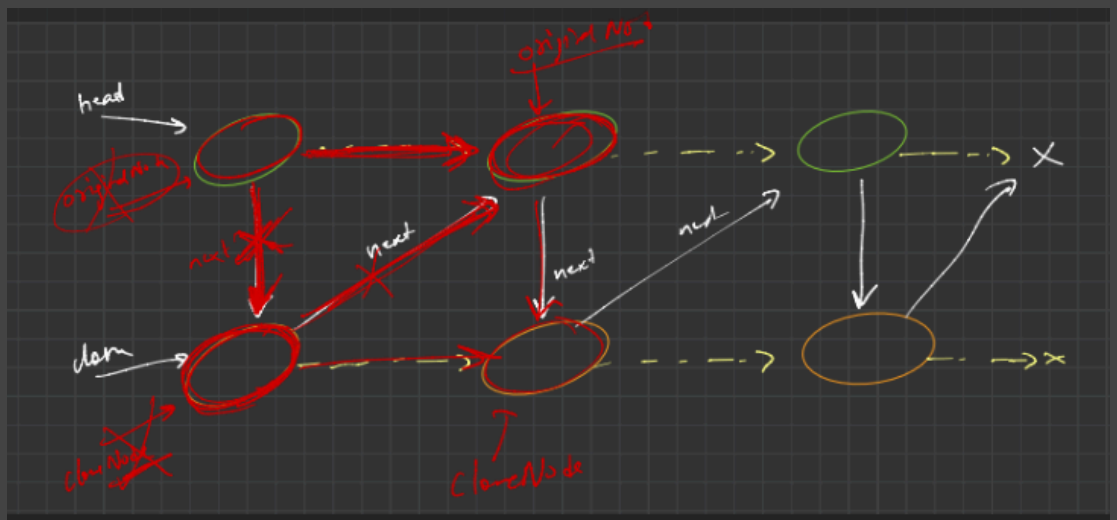





```
clone {
    Node * next = originalNode->next;
    originalNode->next = cloneNode
    originalNode = next
```

(I)

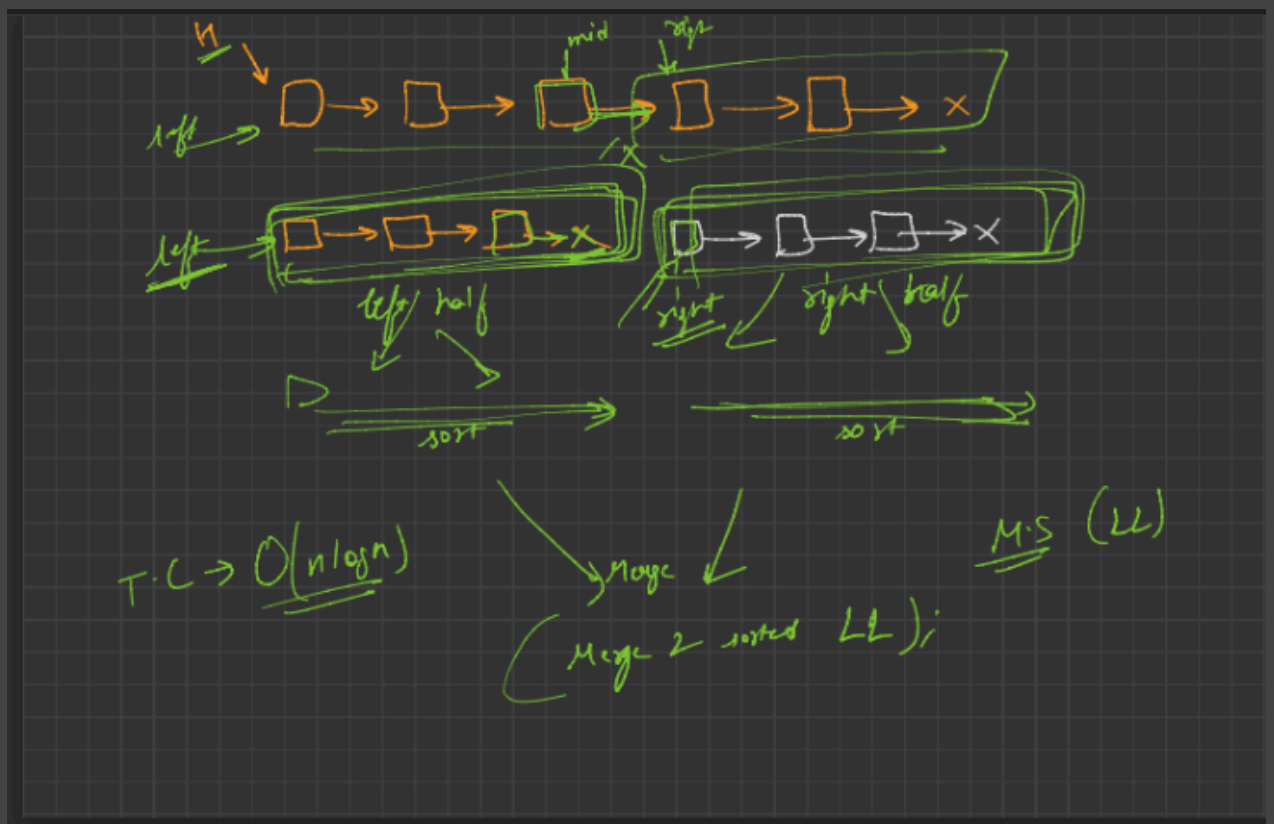
```
    next = cloneNode->next;
    cloneNode->next = originalNode
    cloneNode = next
```

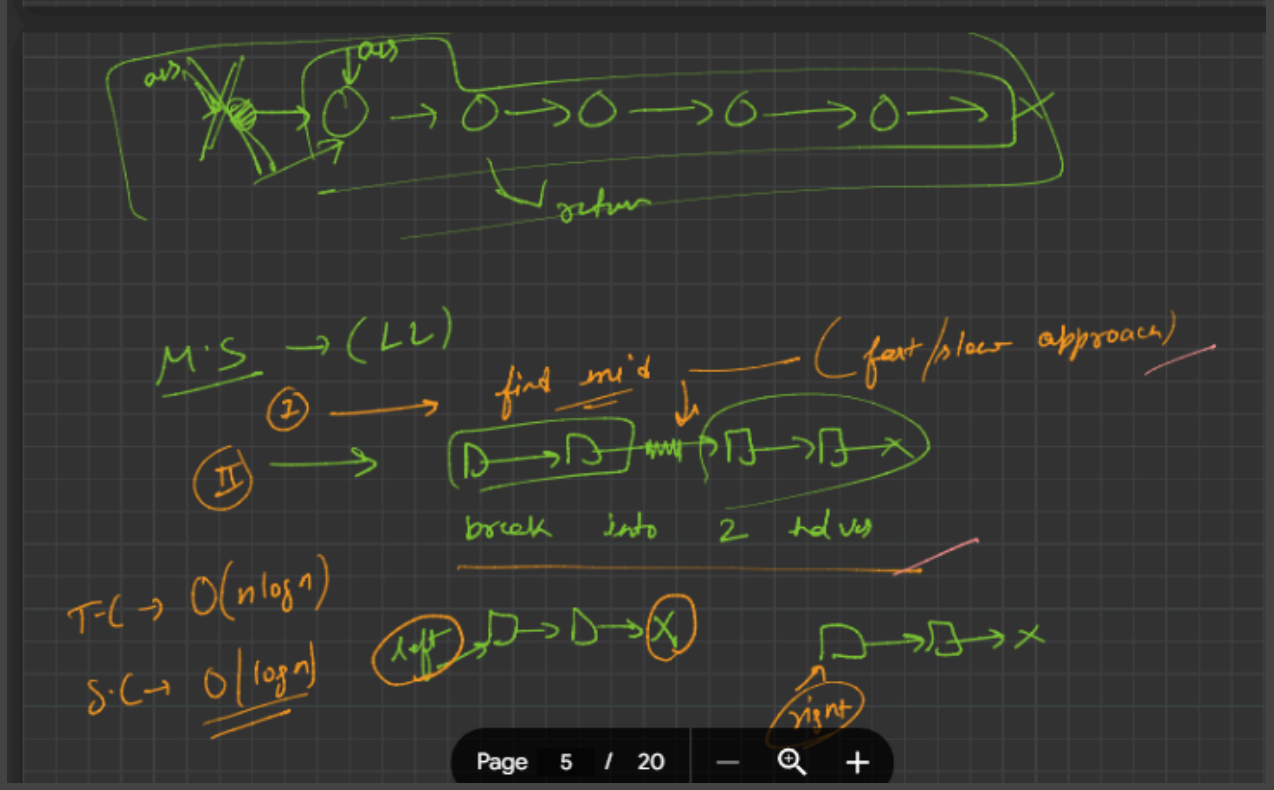



```
clone {
    originalNode->next = cloneNode->next
    originalNode = originalNode->next

    cloneNode->next = originalNode->next
    cloneNode = cloneNode->next
```

(II)

Merge Sort In Linked List:



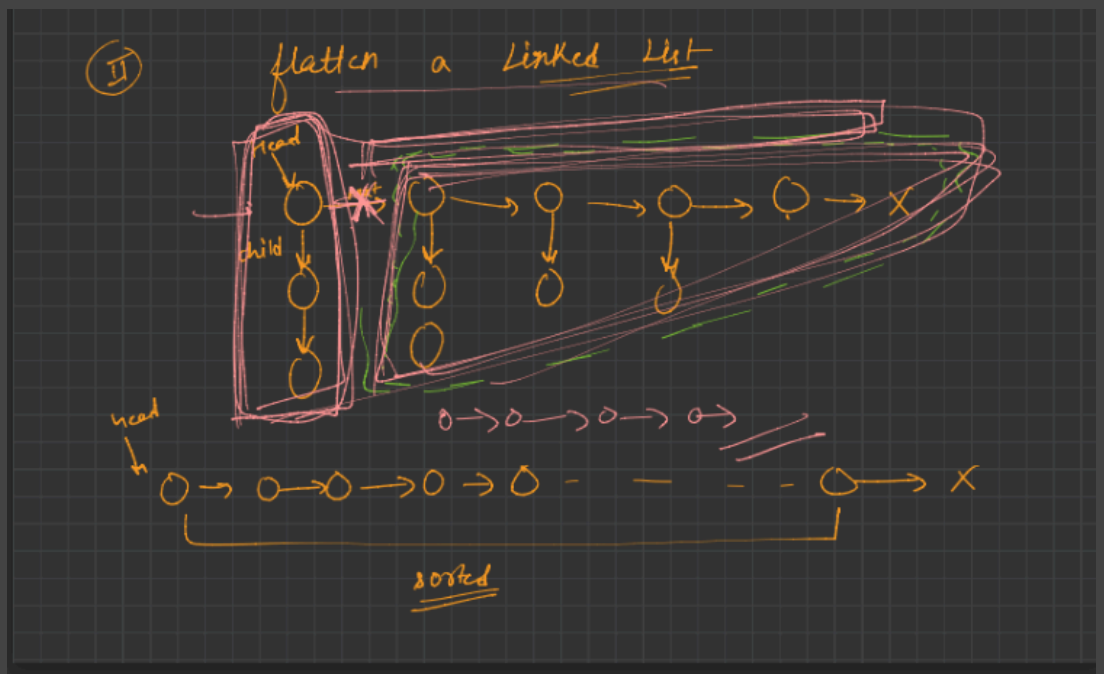


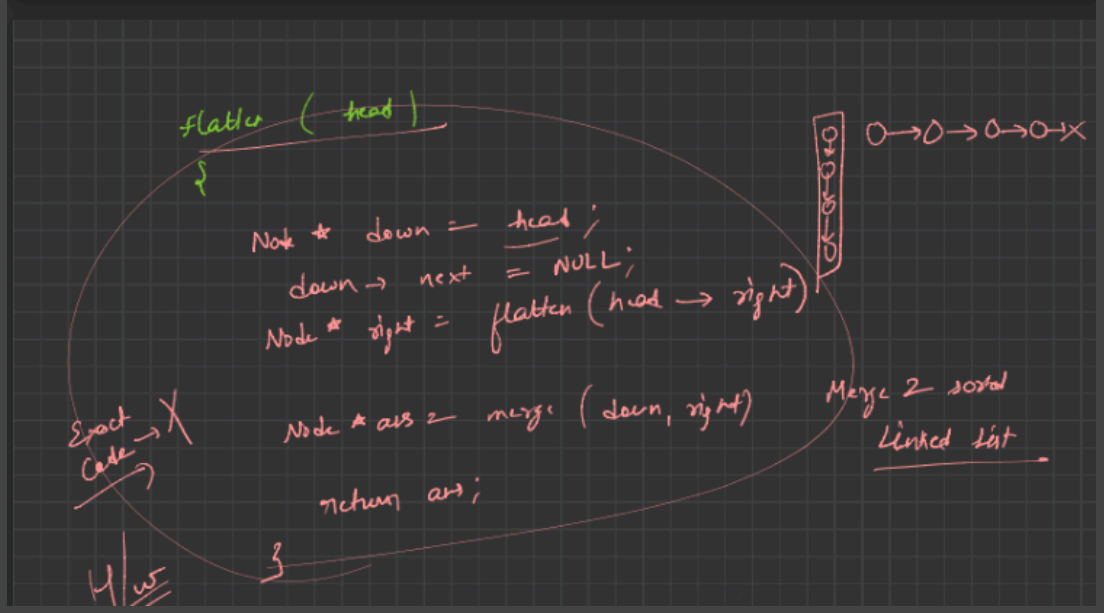
III Recursively sort left/right halves

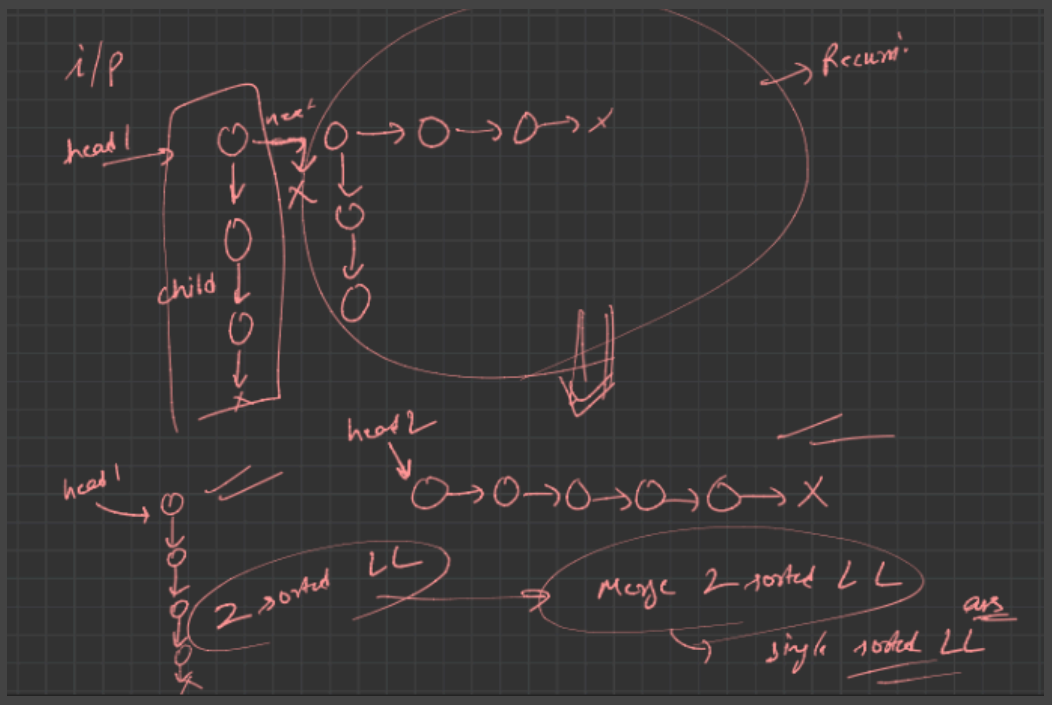
IV merge left/right sorted halves → (Merge 2 sorted LL)

V return merged list

Flatten Linked List:

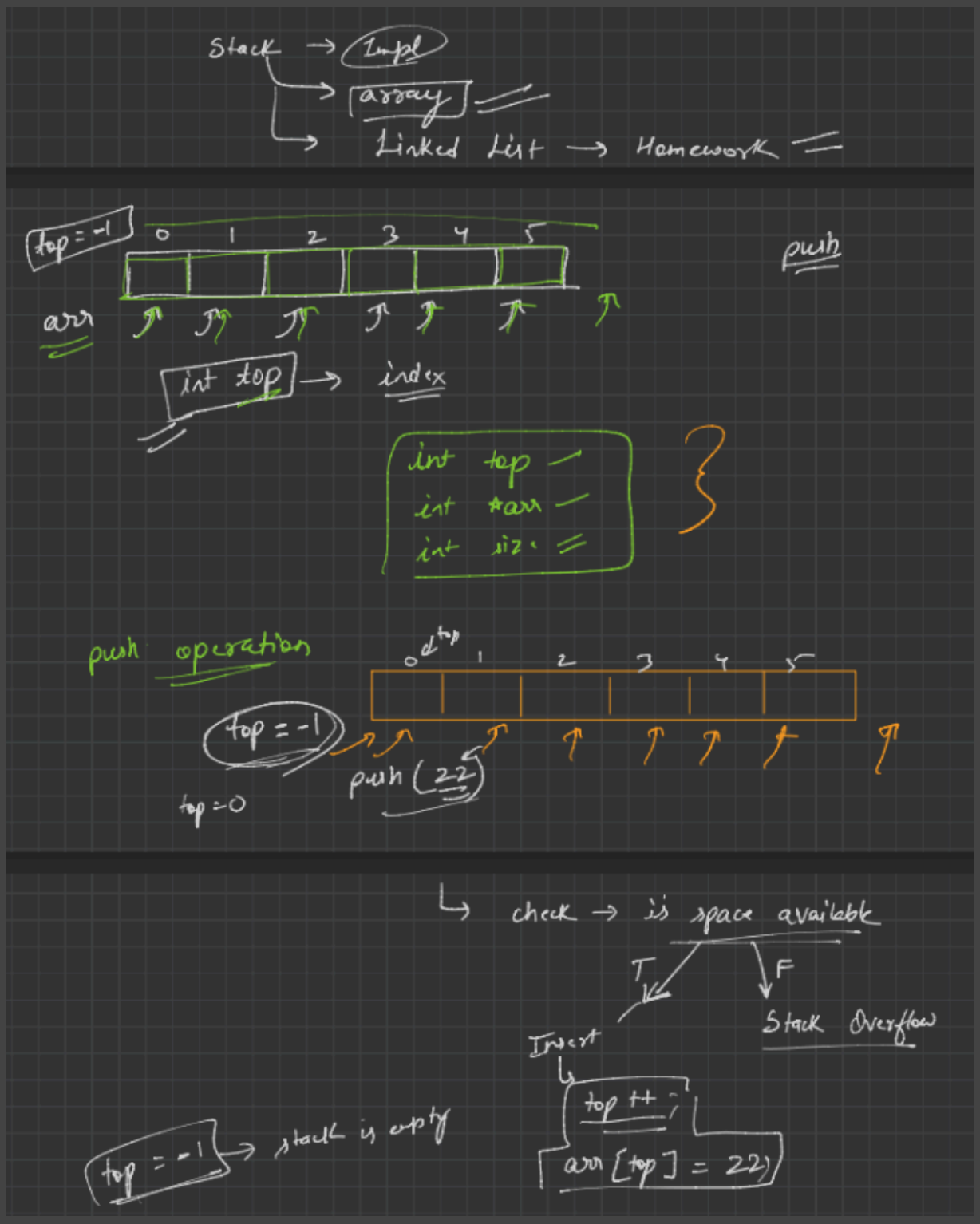




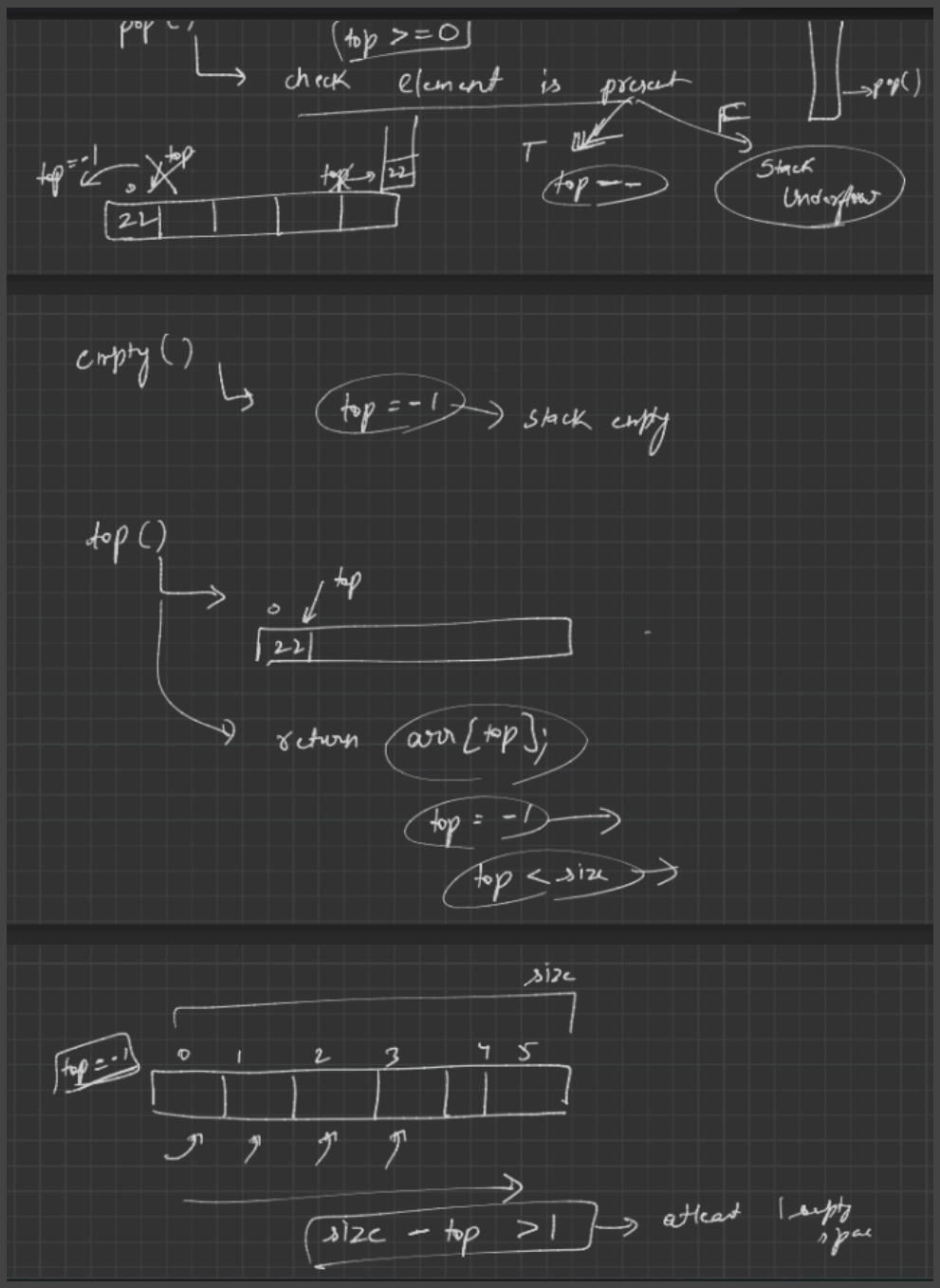


Stack

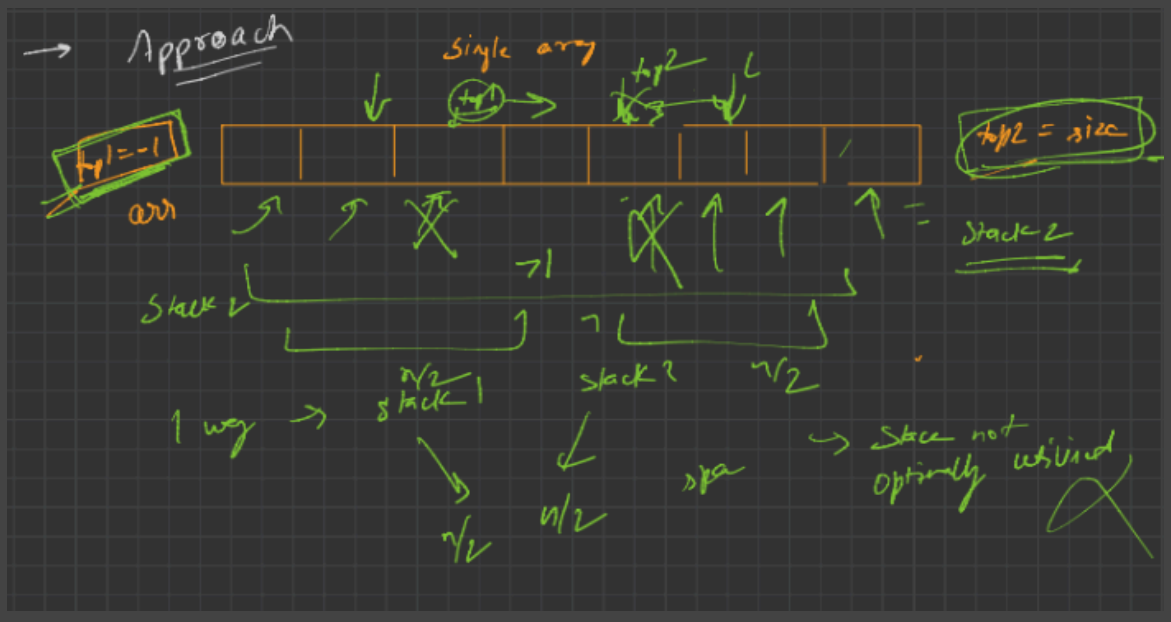
Create A Stack Using An Array:



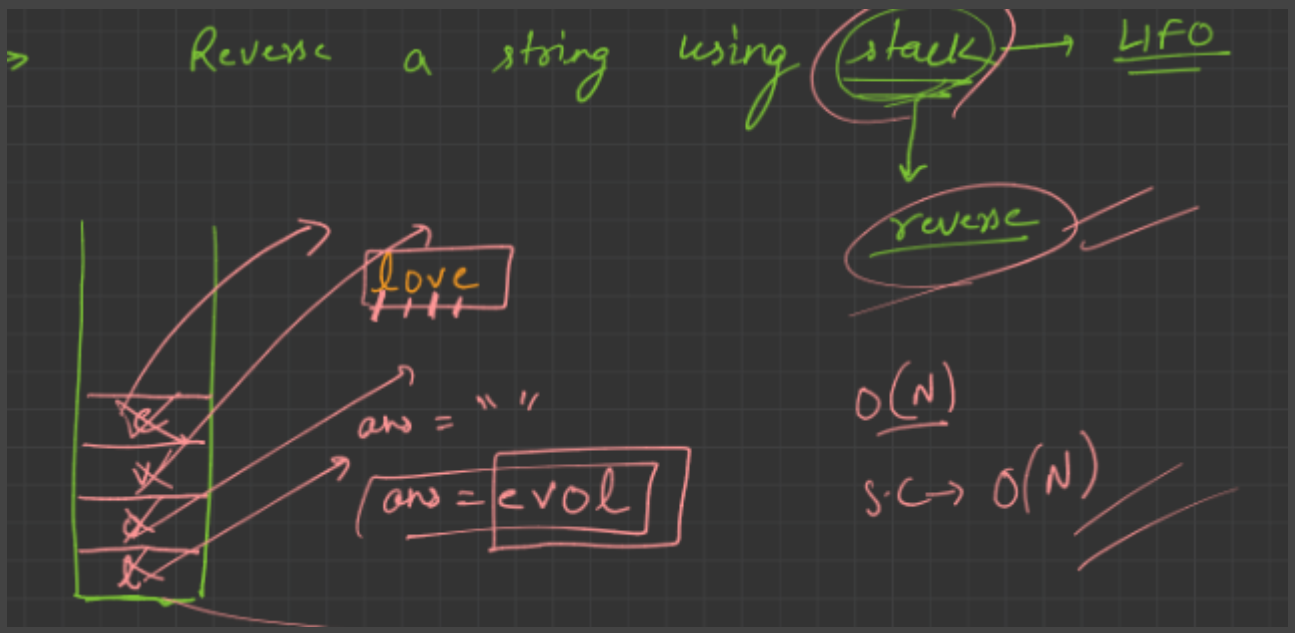
Create A Stack Using A Linked List:



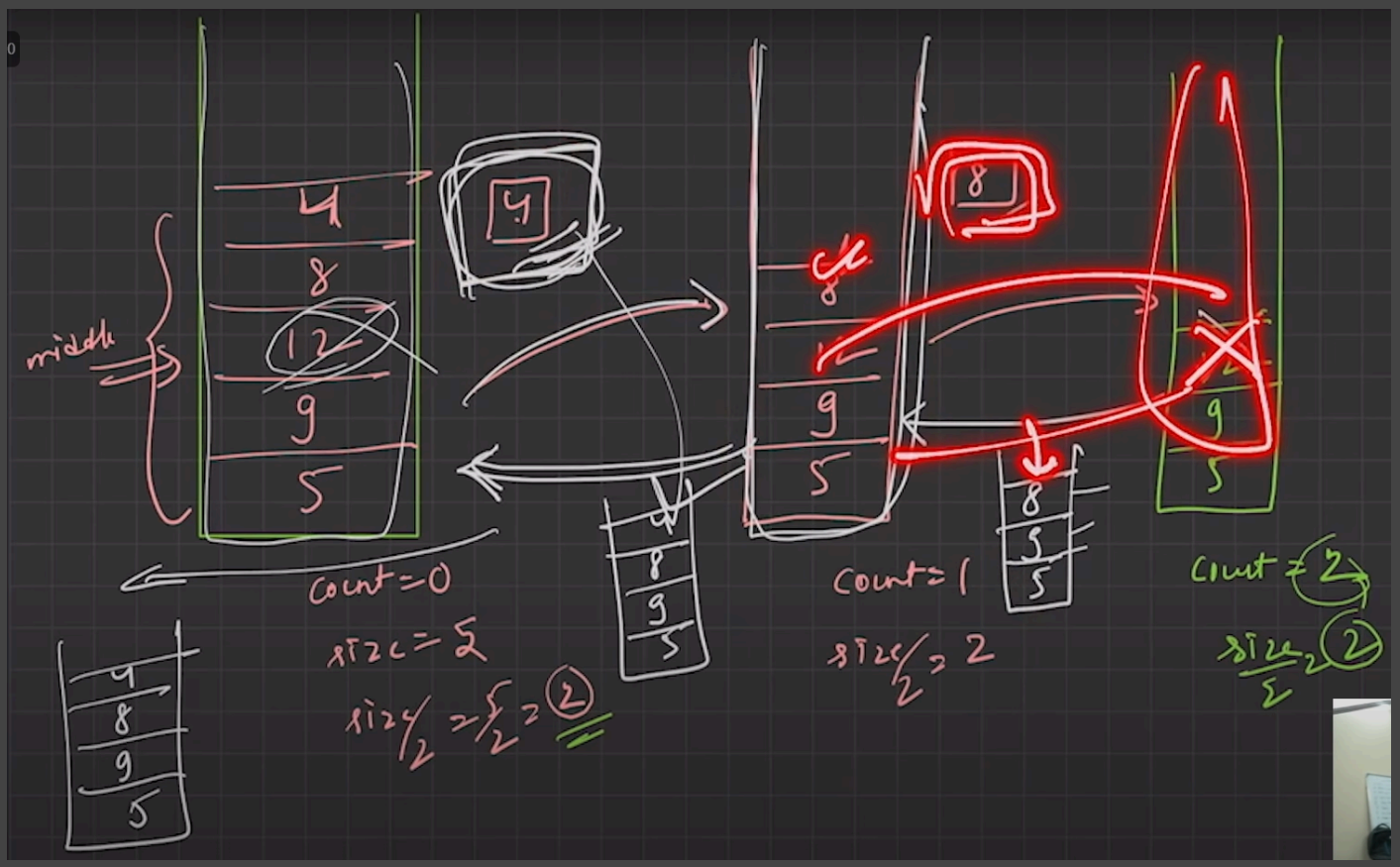
Create Two Stacks Using An Array:



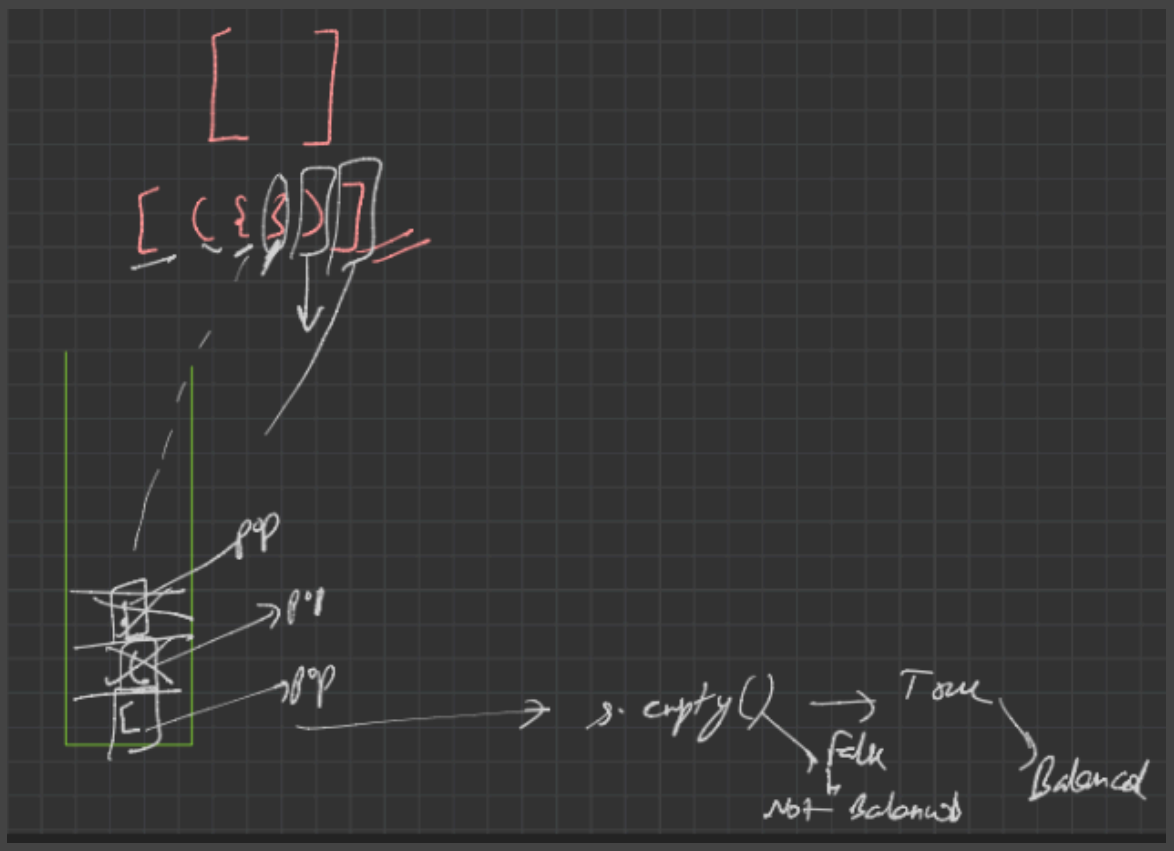
Reverse A String Using Stack:



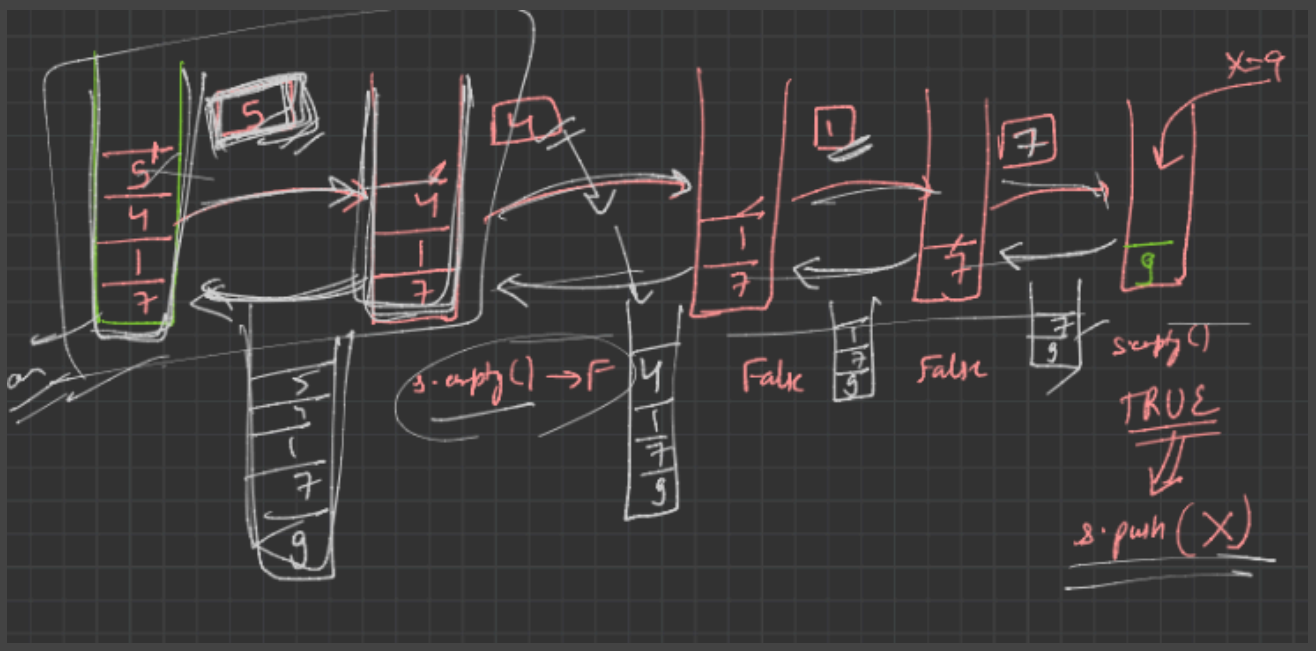
Delete MIddle Element In A Stack:



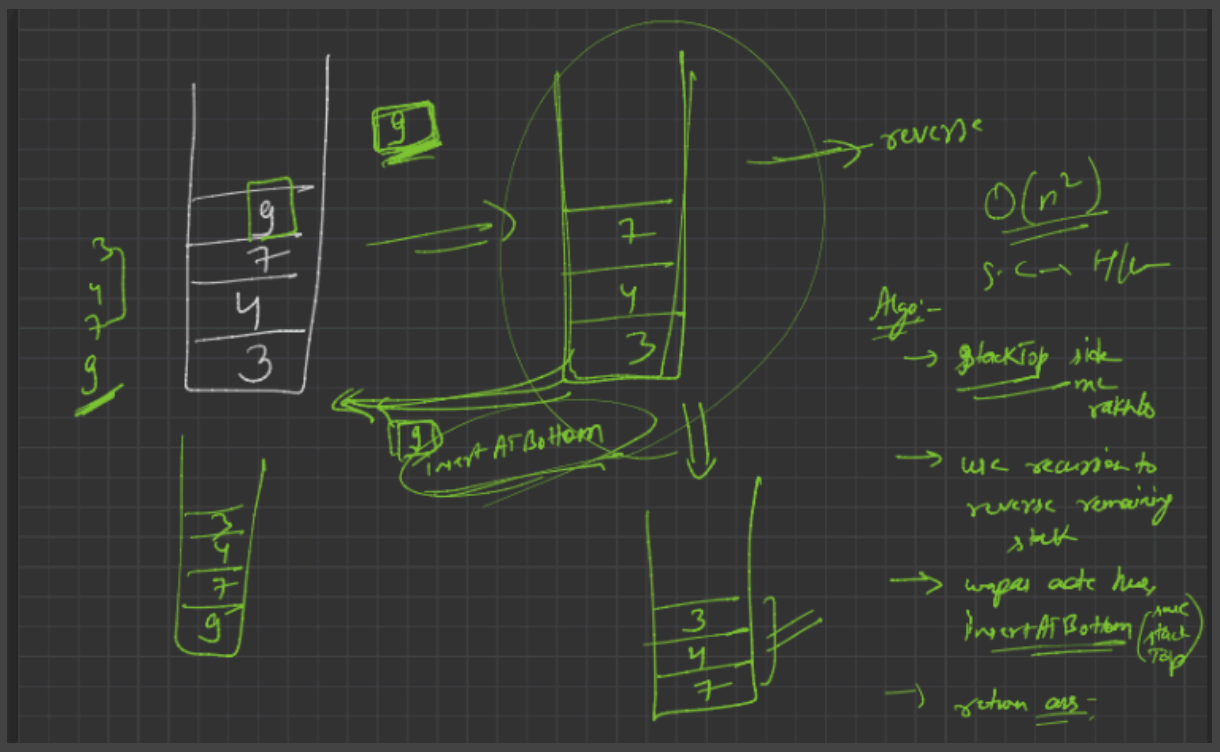
Check Whether the Parenthesis Are Valid:



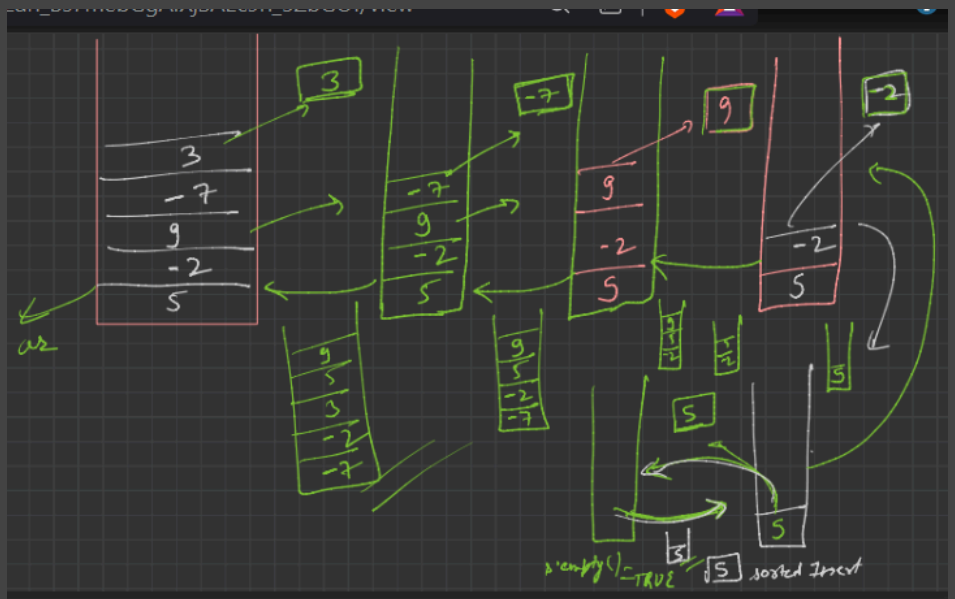
Insert An Element At Its Bottom In A Given Stack:

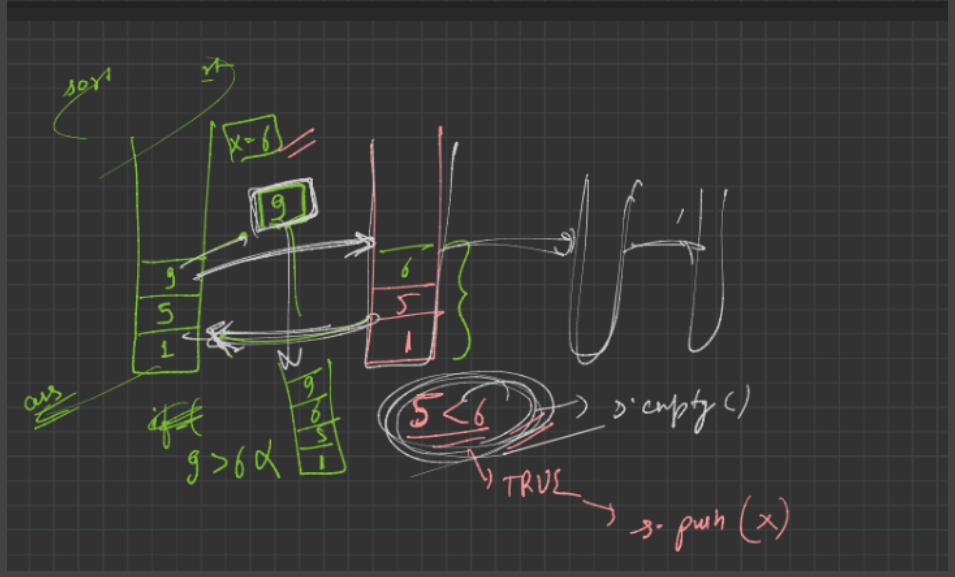


Reverse A Stack:

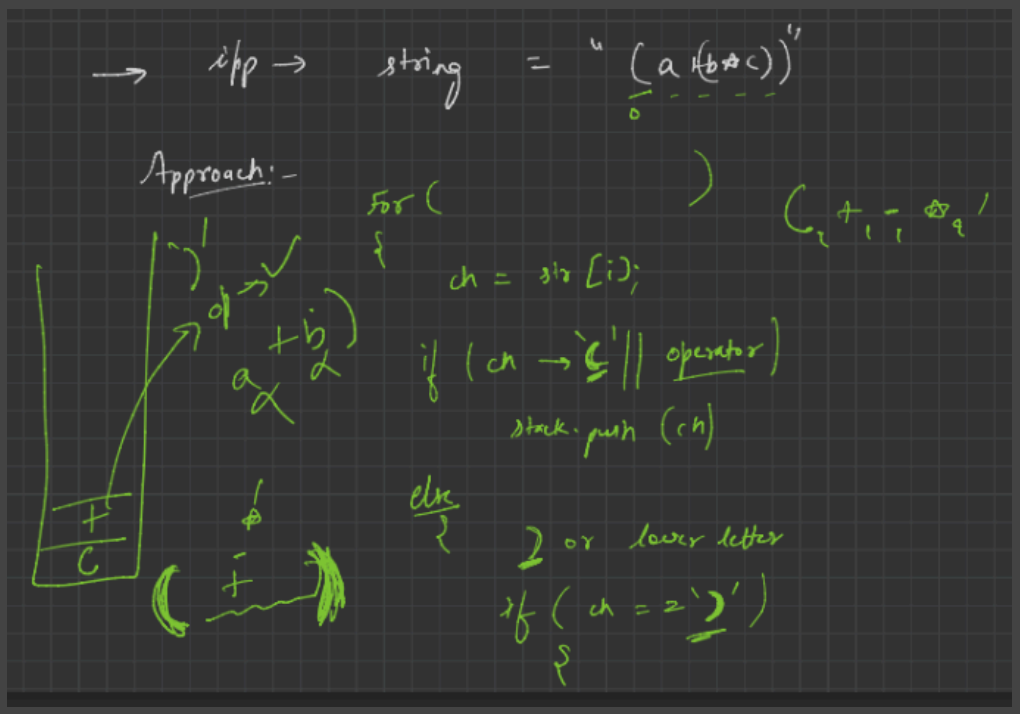


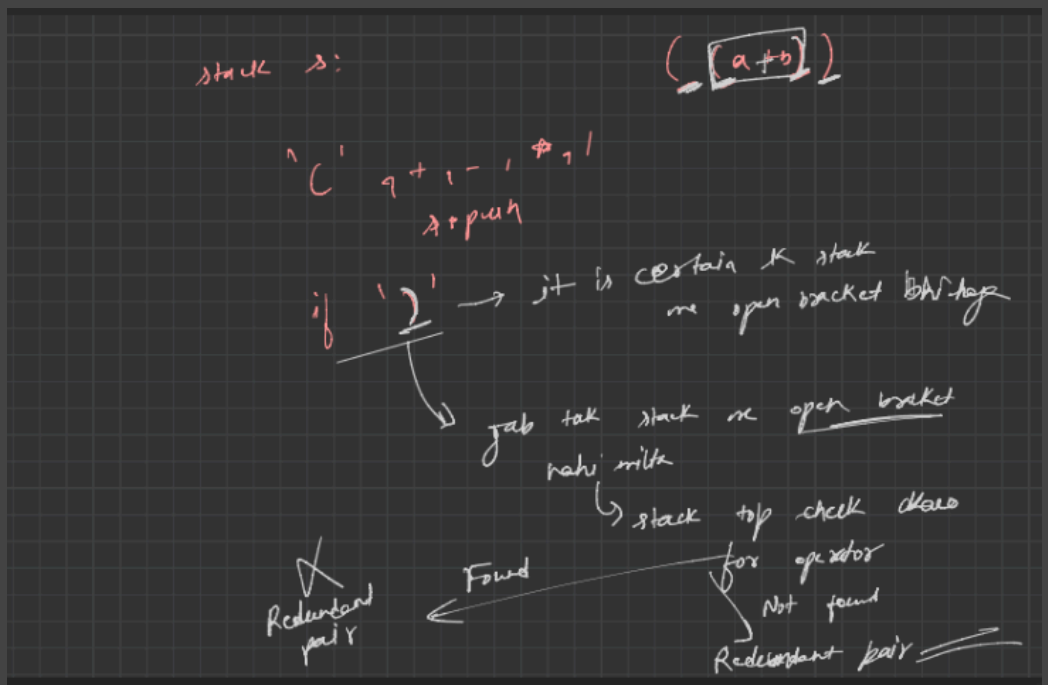
Sort A Stack:

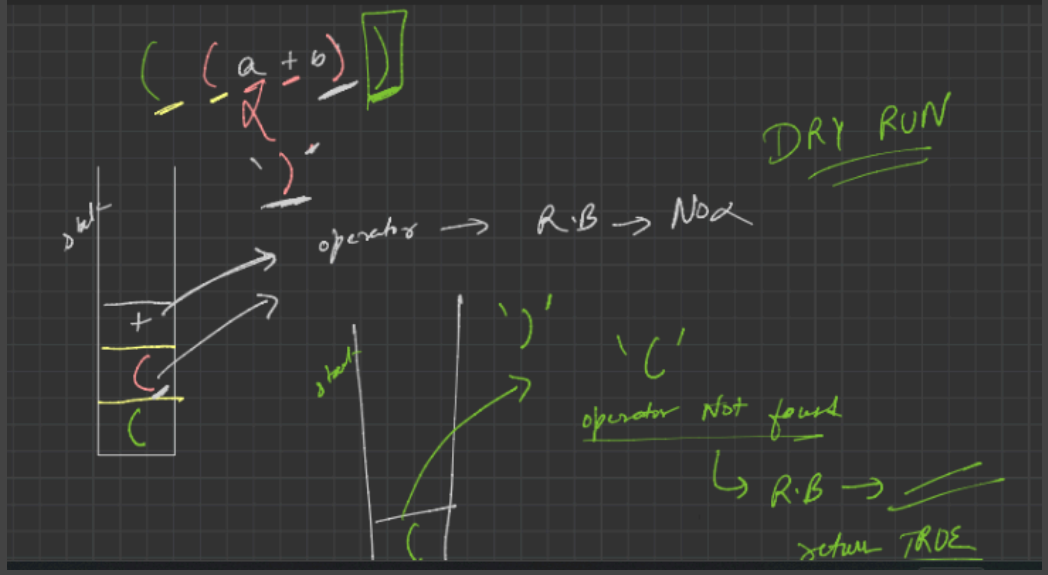




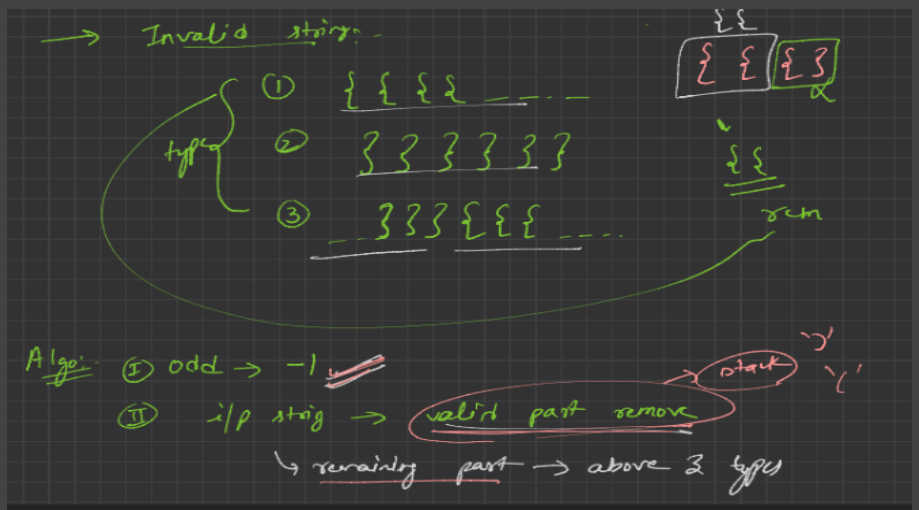
Check Whether There Are Redundant Brackets Or Not:

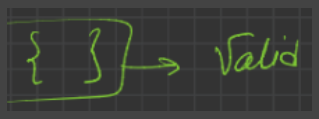


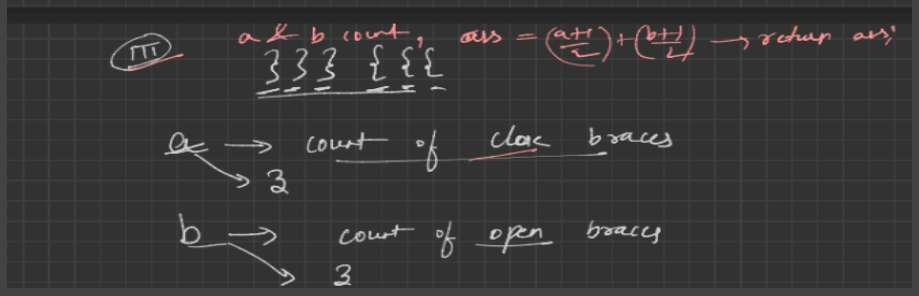


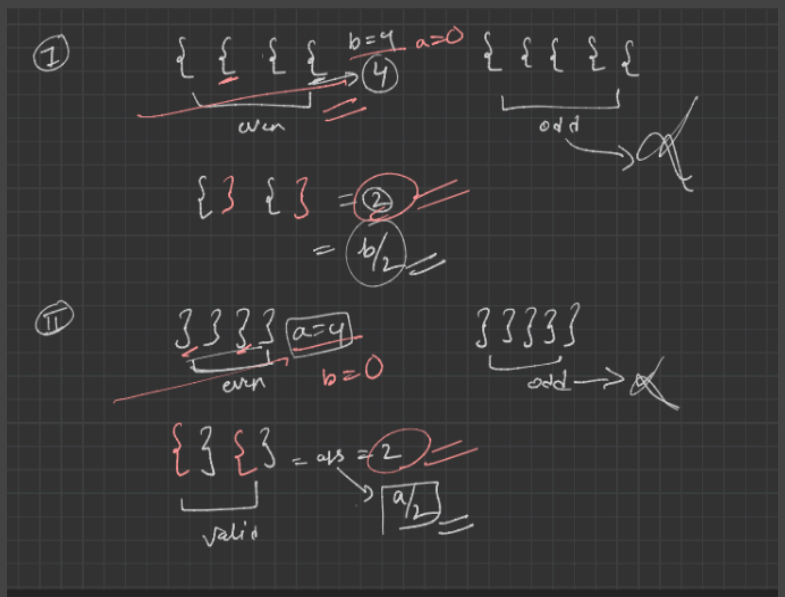


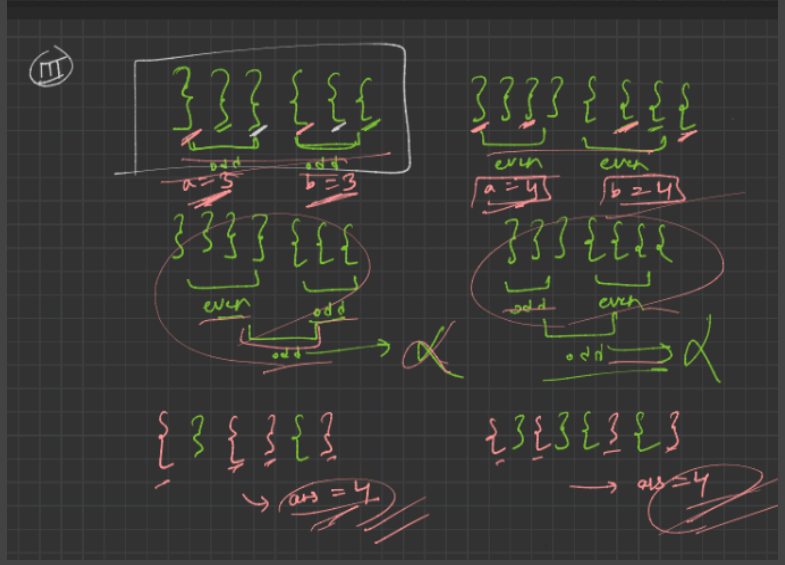
Minimum Number Of Operations To Make A Valid Bracket String:

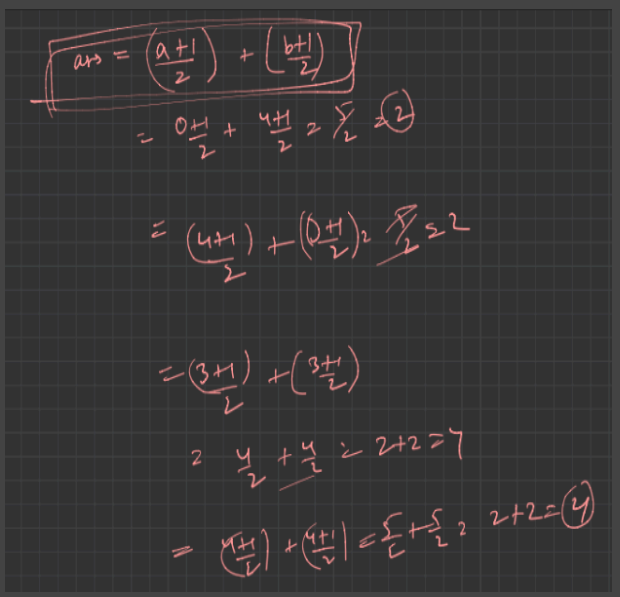




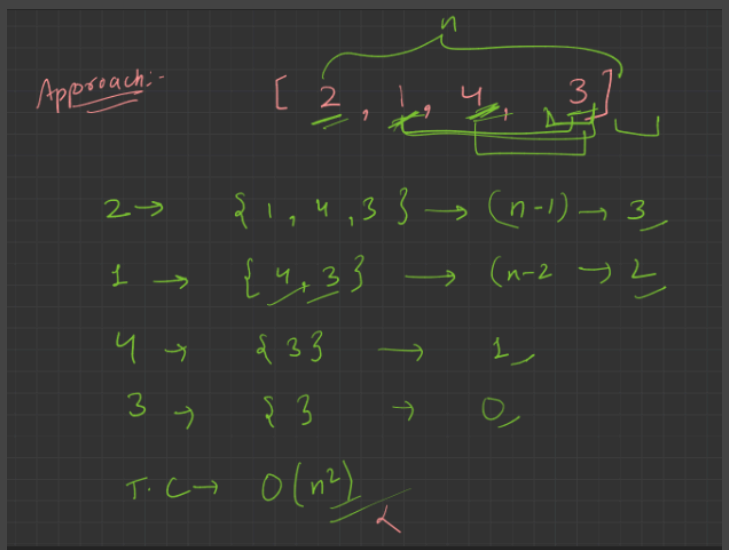


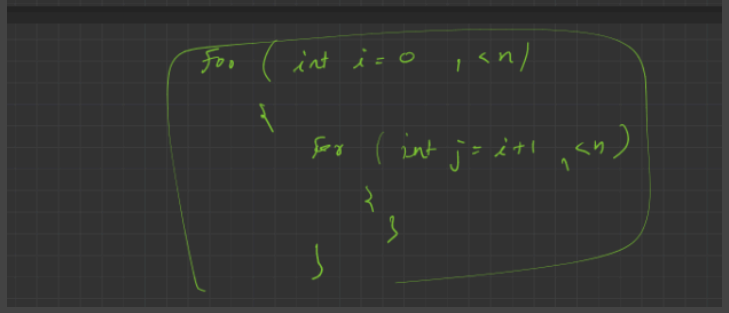




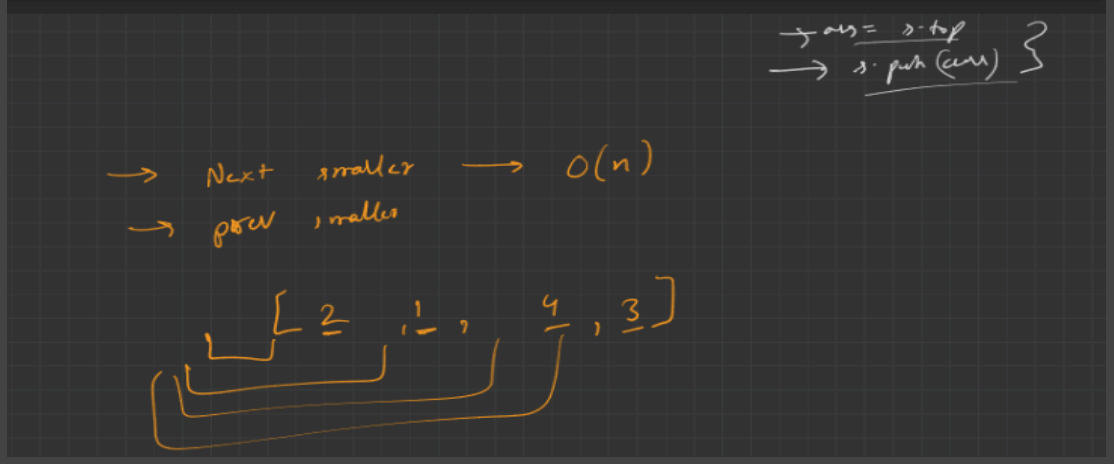


Next Smaller Element Approach-I:





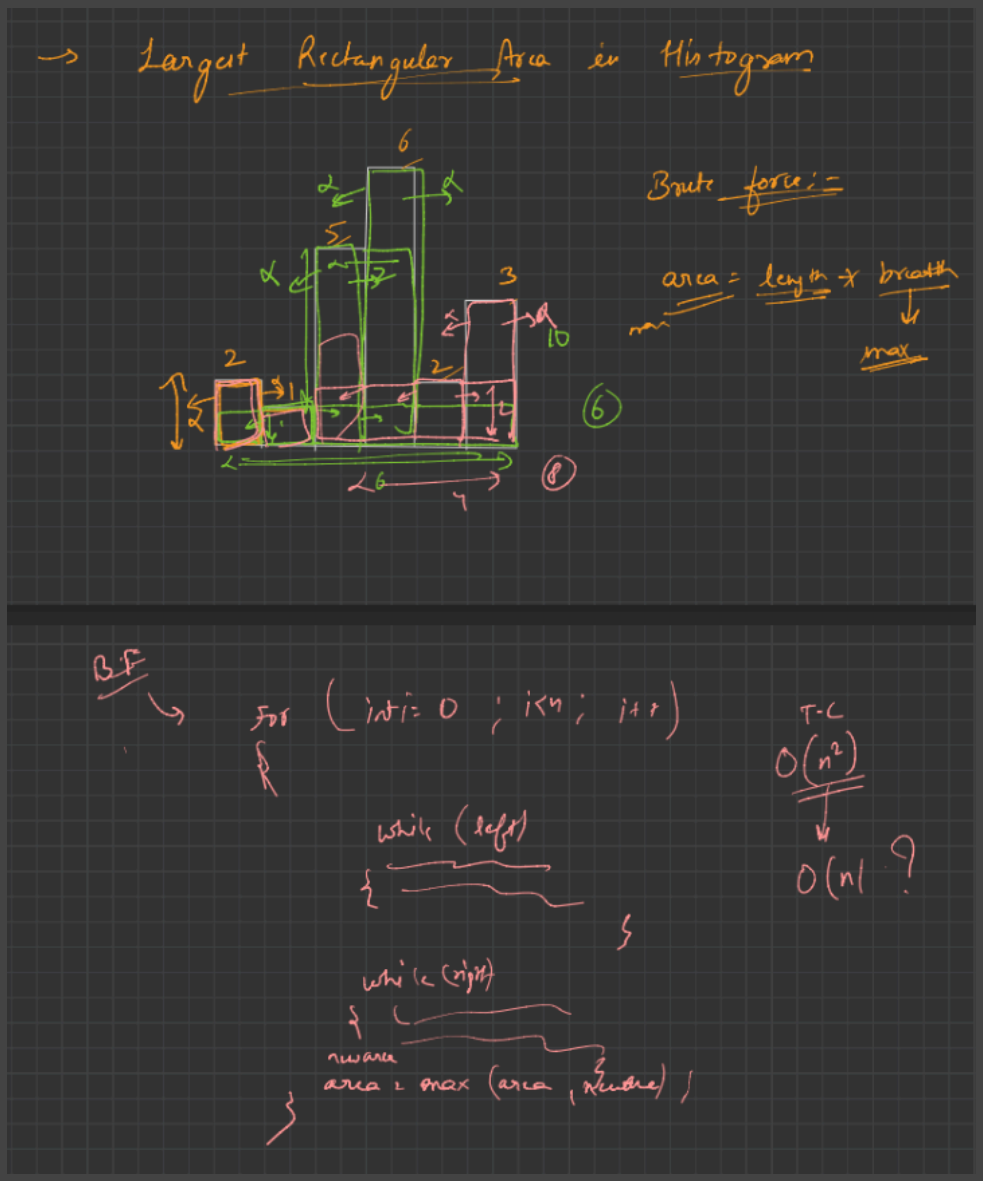
Next Smaller Element Approach-II:



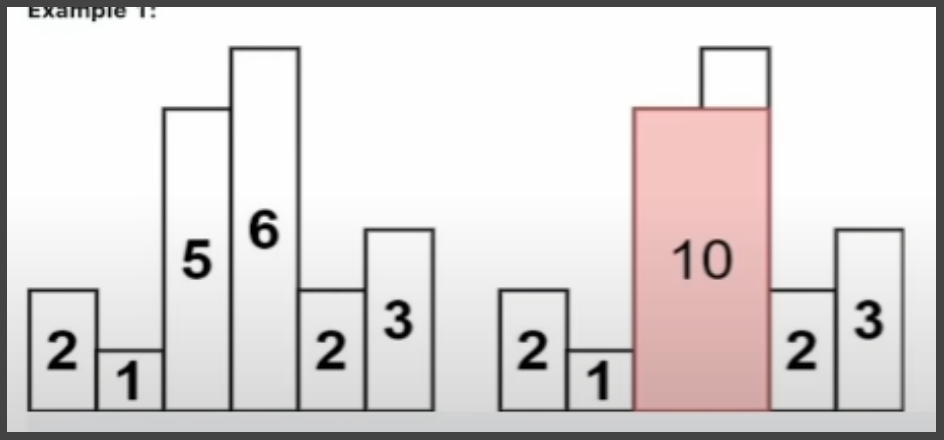
Previous Smaller Element:

If you want to find the previous smaller element for each element in the array, you would need to modify the code to iterate from left to right instead of from right to left.

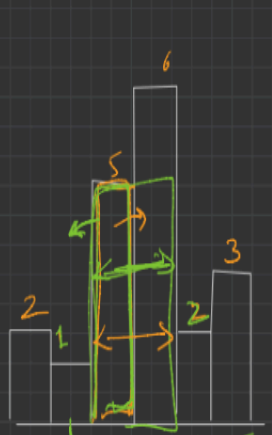
Largest Rectangle In Histogram Approach-I:



Largest Rectangle In Histogram Approach-II:



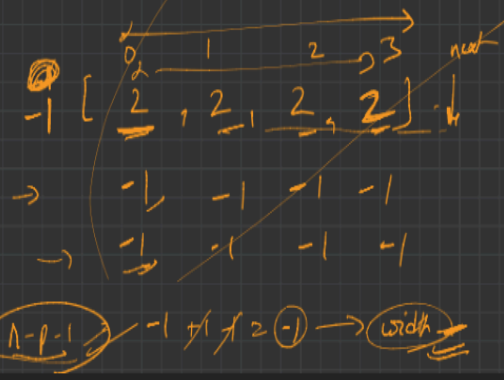
O(n) → Approach #2



width = n-p-1

area = $\frac{\text{base}}{3} \times h$ = 4-1-1 = 2

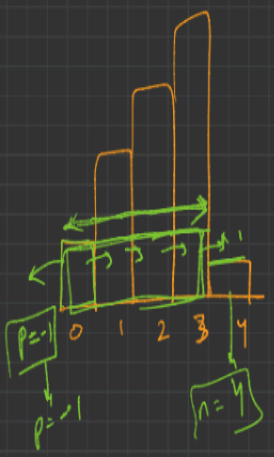
2 2 2 2



next → -1 -1 -1 -1

prev → -1 -1 -1 -1

n-p-1 = -1 ≠ 1 ≠ 2 → -1 → width



width = n-p-1

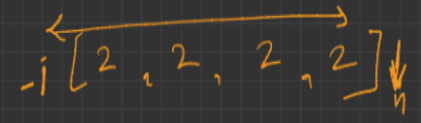
= 4-(-1)-1

= 4 ≠ 1 ≠ 1

= 4

next → next smaller element index

prev → prev



next = -1 -1 -1 -1 if(next == -1

prev = -1 -1 -1 1

4

n-p-1

= -1 ≠ 1 ≠ -1 → break/width = -1

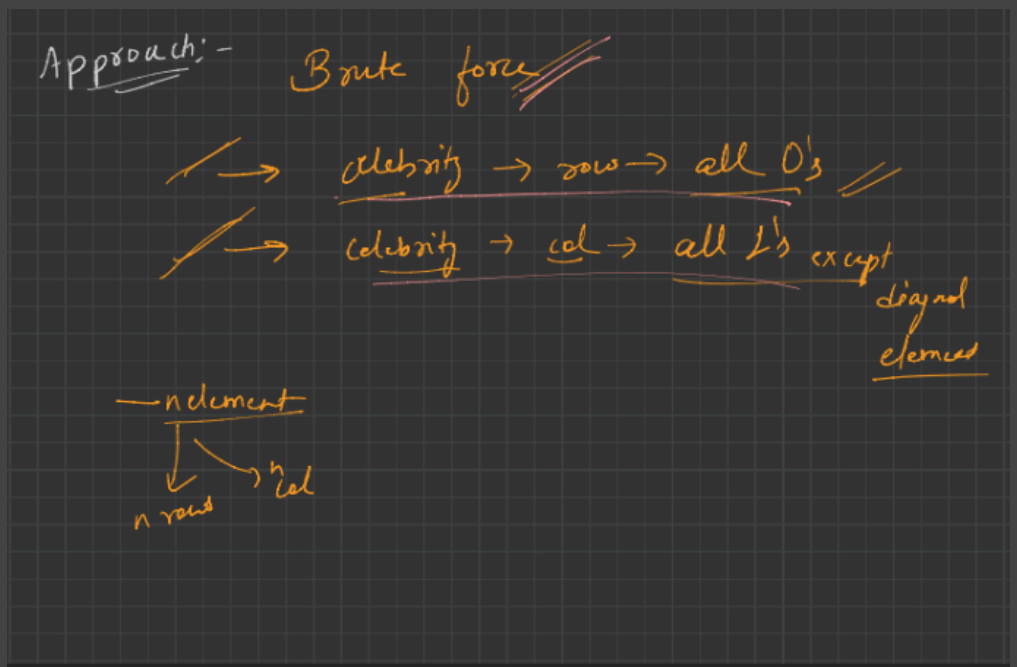
area = -ve

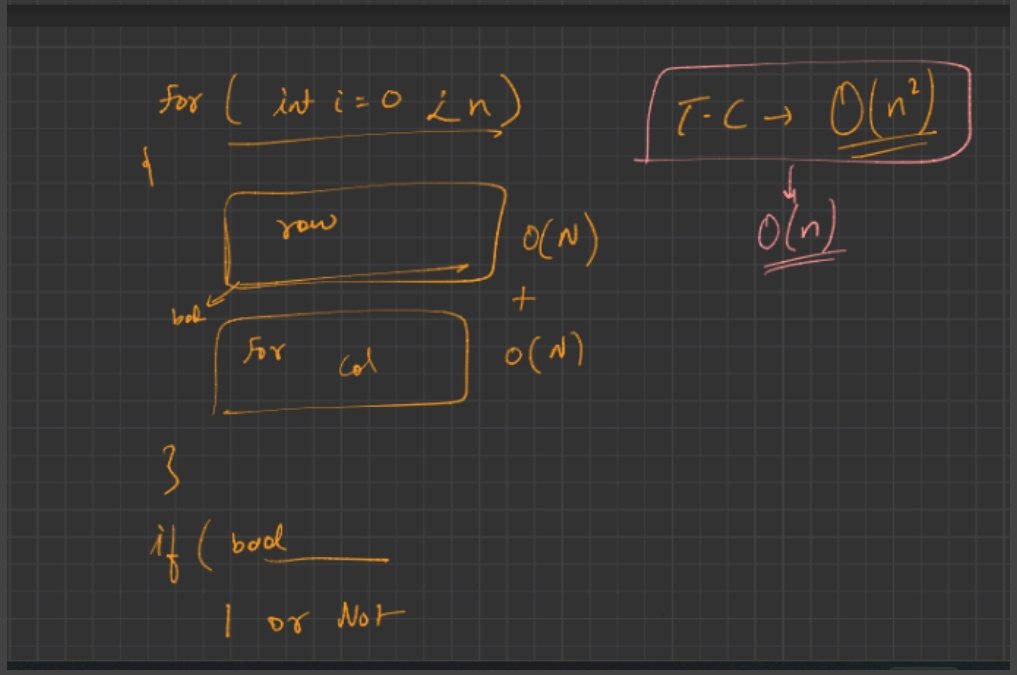
T.C → O(N)

S.C → O(N)

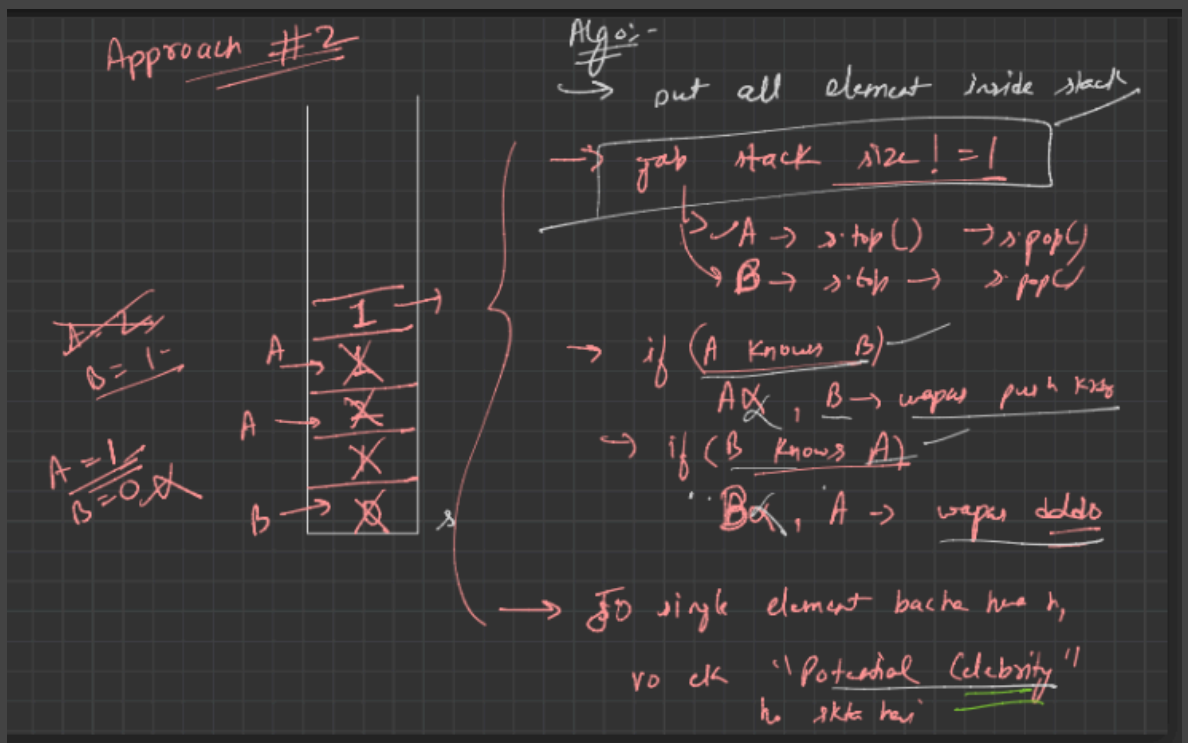
Hoza aagya

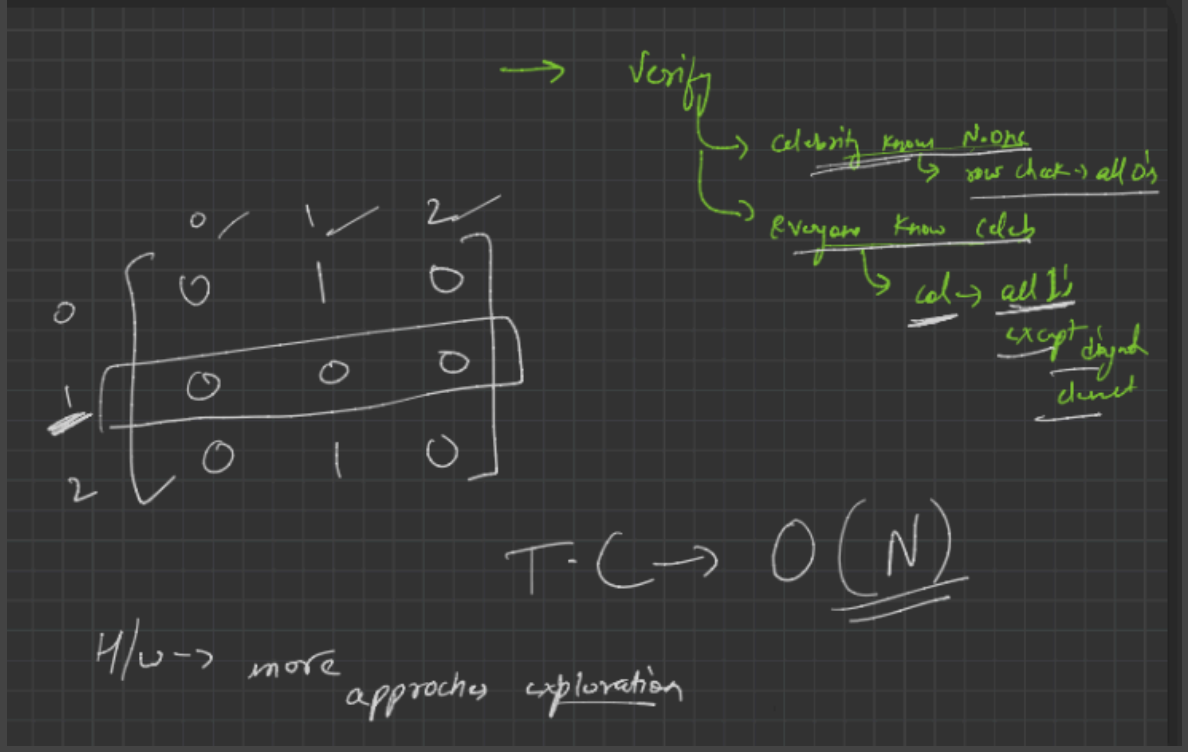
Celebrity Problem Approach-I:



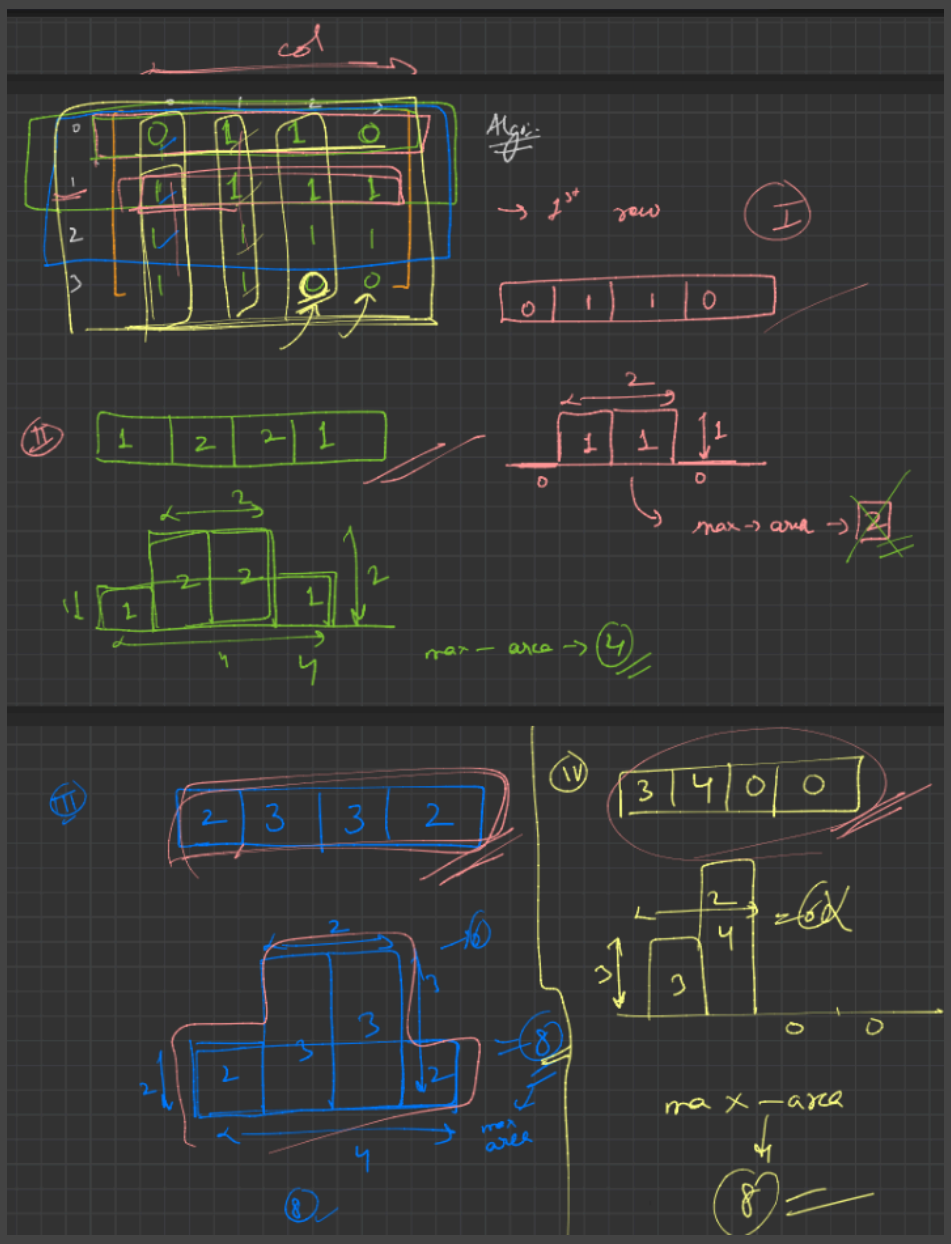


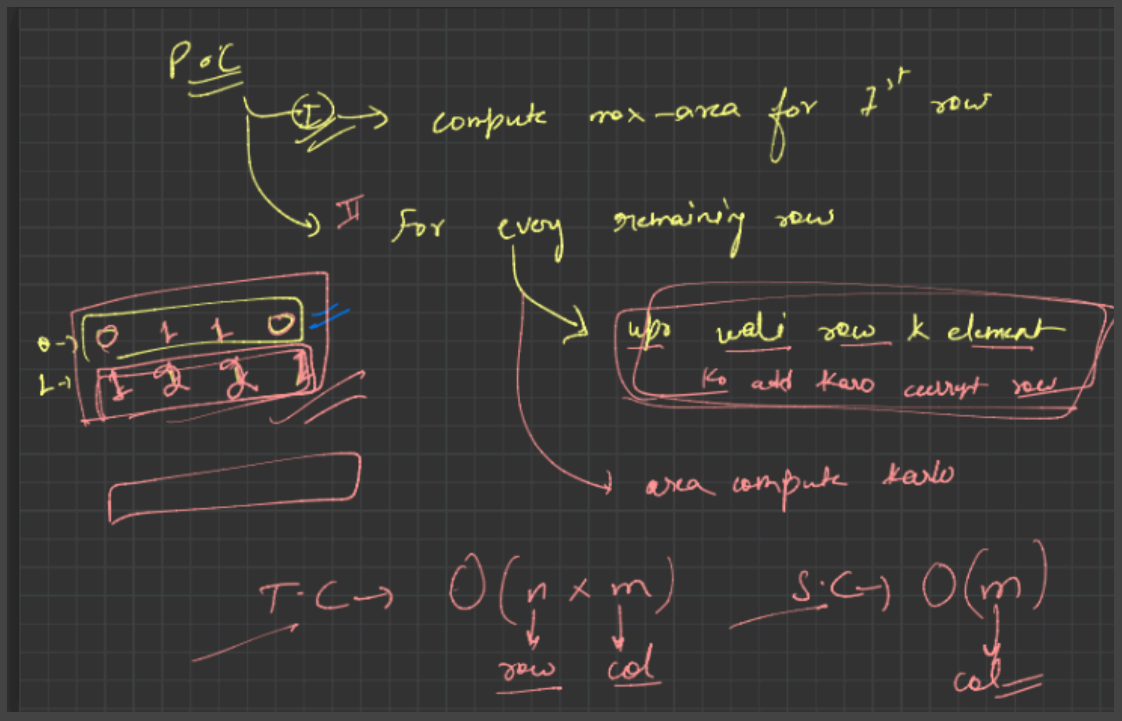
Celebrity Problem Approach-II:





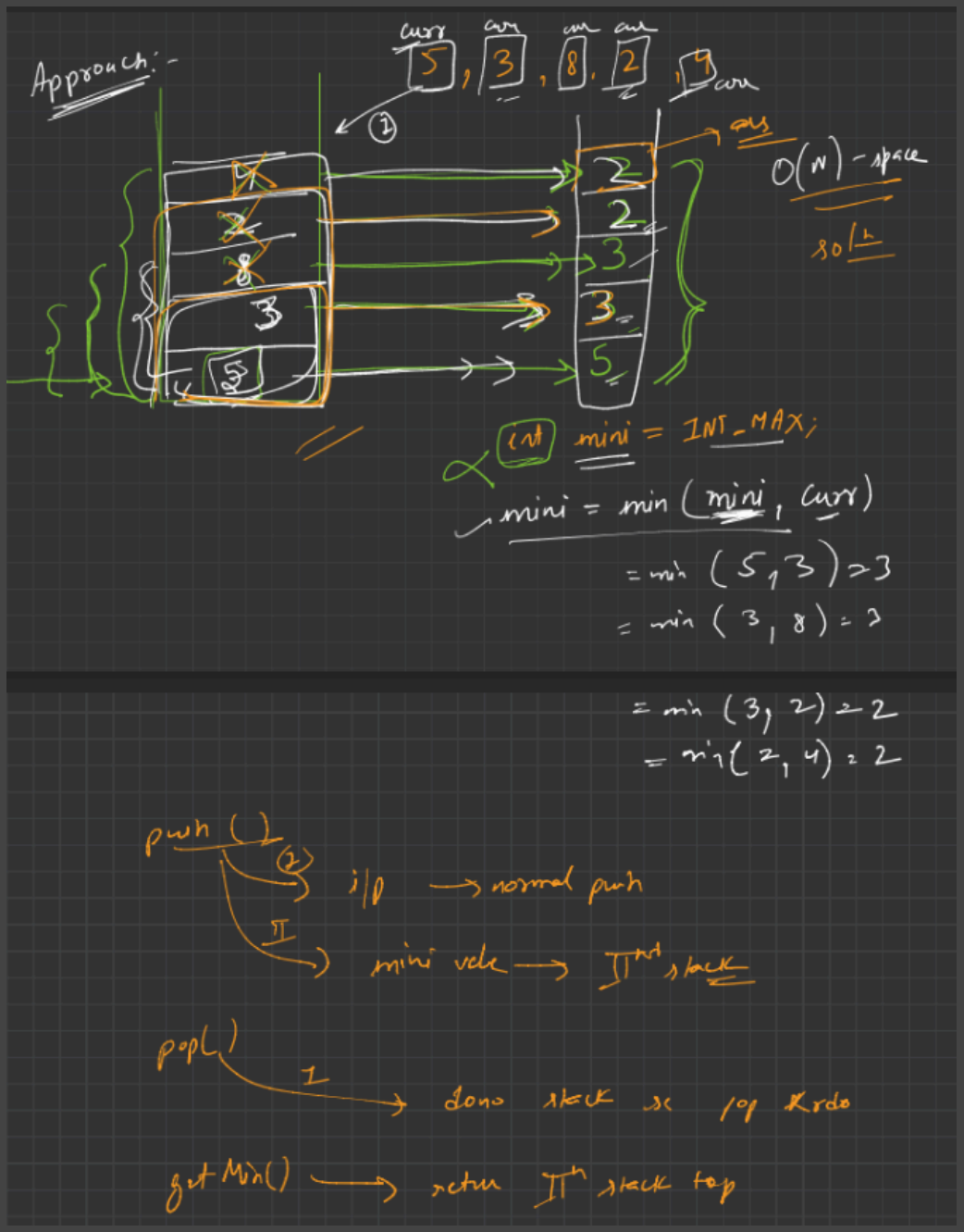
Maximum Rectangle In Binary Matrix:

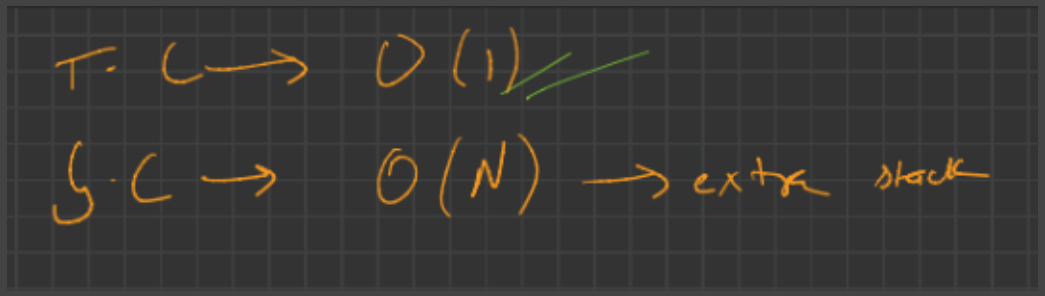




Design A Stack That Supports getMin in O(1) Time Complexity and O(1) Space Complexity:

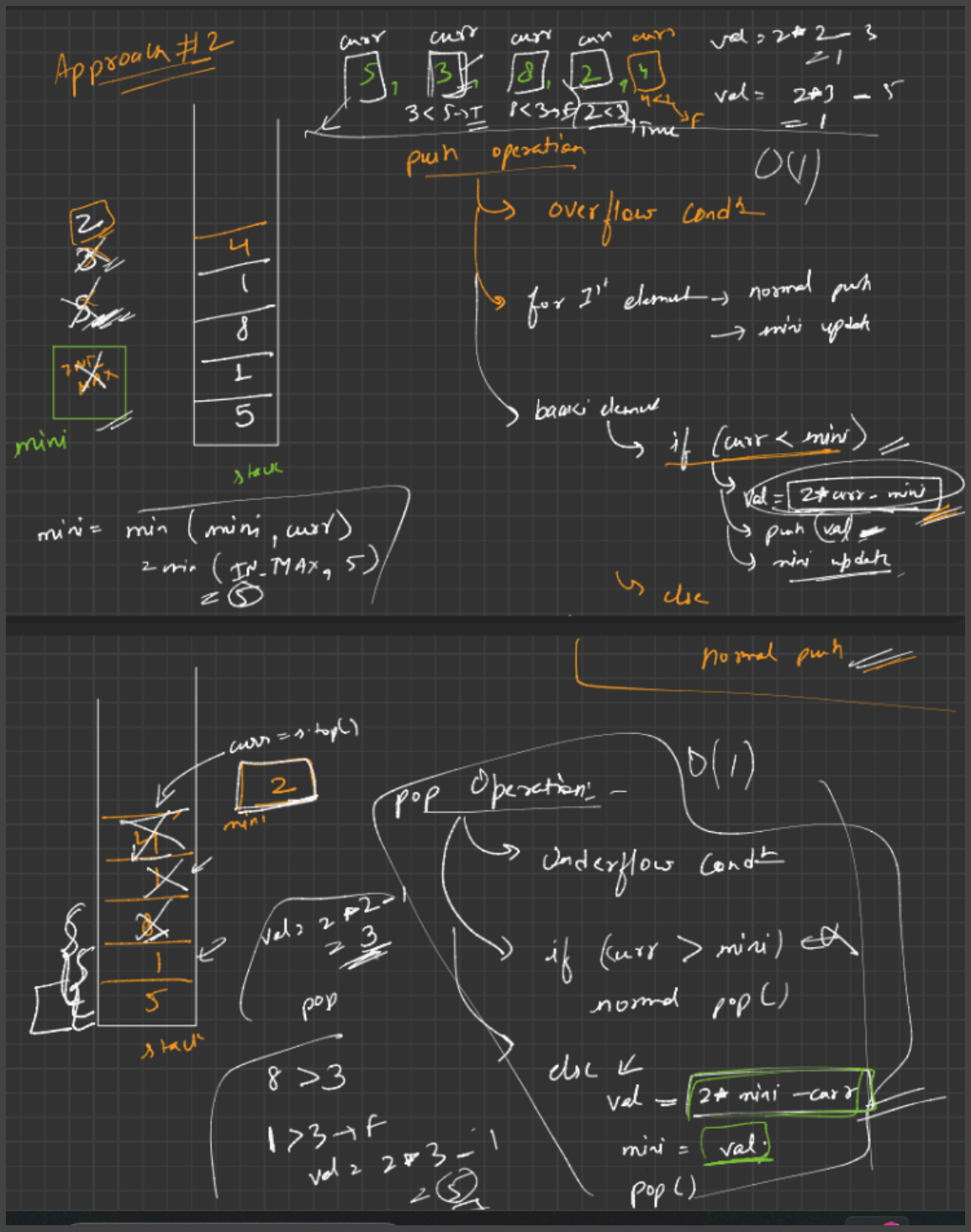
Approach -I:





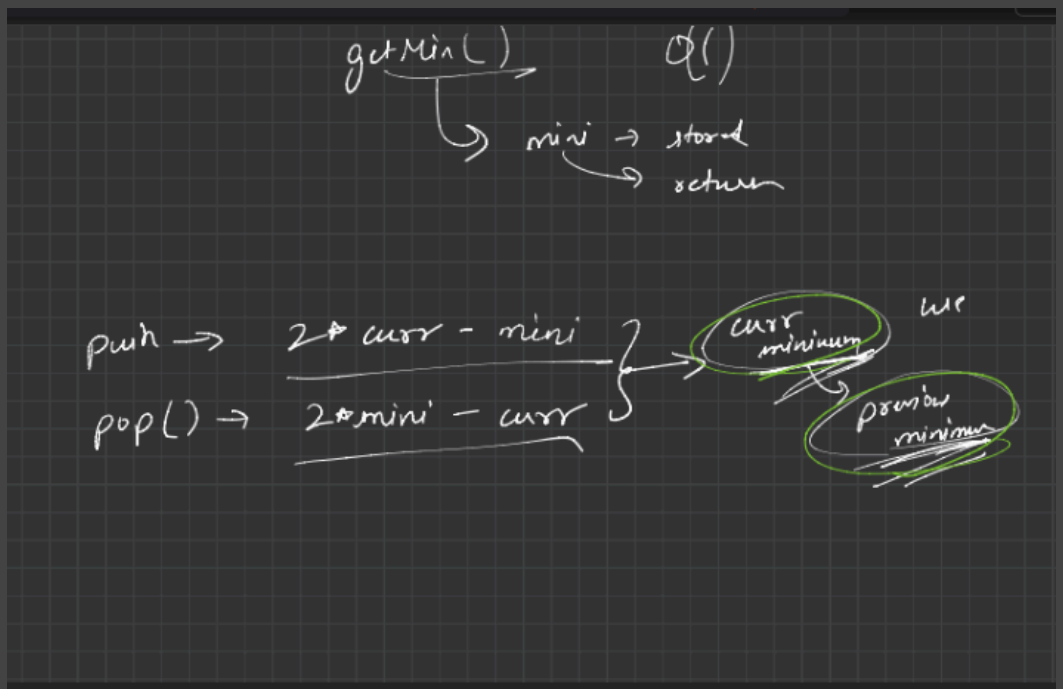
Design A Stack That Supports getMin in O(1) Time Complexity and O(1) Space Complexity:

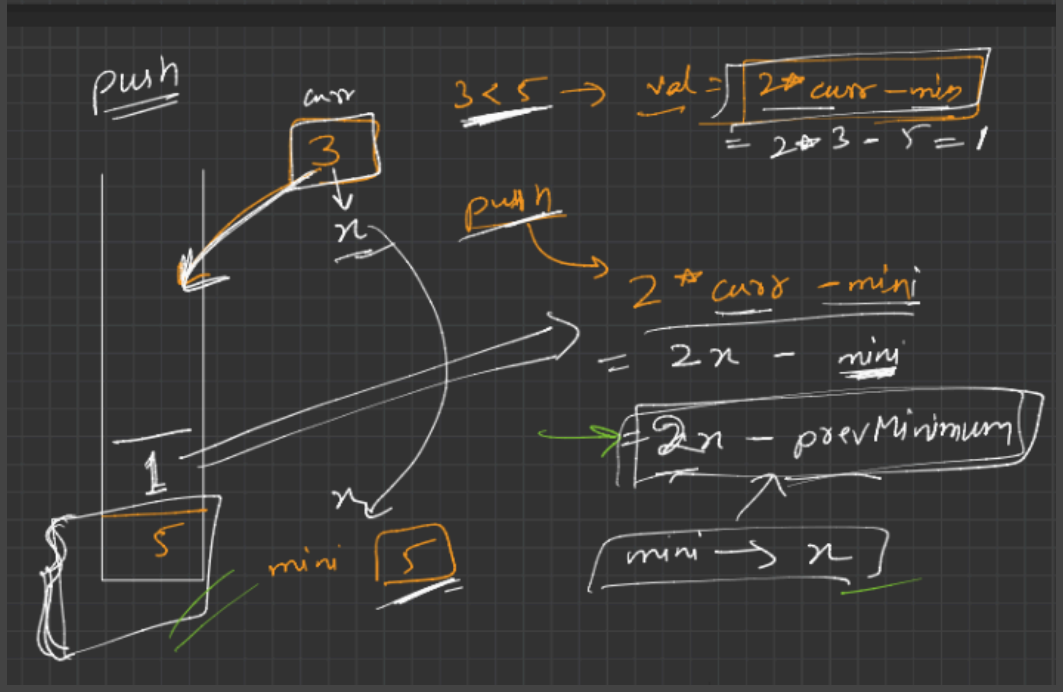
Approach -II:

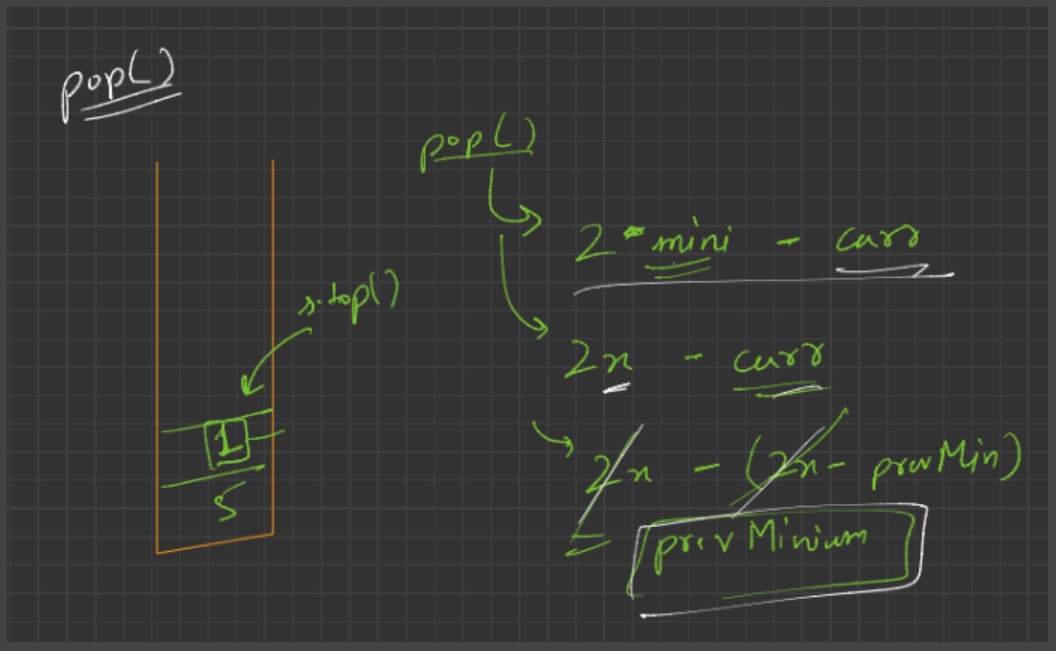


The SpecialStack class uses a single stack s to store the elements. The mini variable is used to keep track of the minimum element. When a new element is pushed onto the stack, if it is less than the current minimum, the stack stores a value derived from the new element and the current minimum (specifically, 2*data - mini). This derived value is not an additional element, but a transformation of the new element being pushed. Therefore, it does not count as additional space.

When an element is popped from the stack, if it is less than the current minimum, the minimum is updated using the derived value and the current minimum (specifically, 2*mini - curr). Again, this does not use additional space, as it's a calculation performed on existing values.

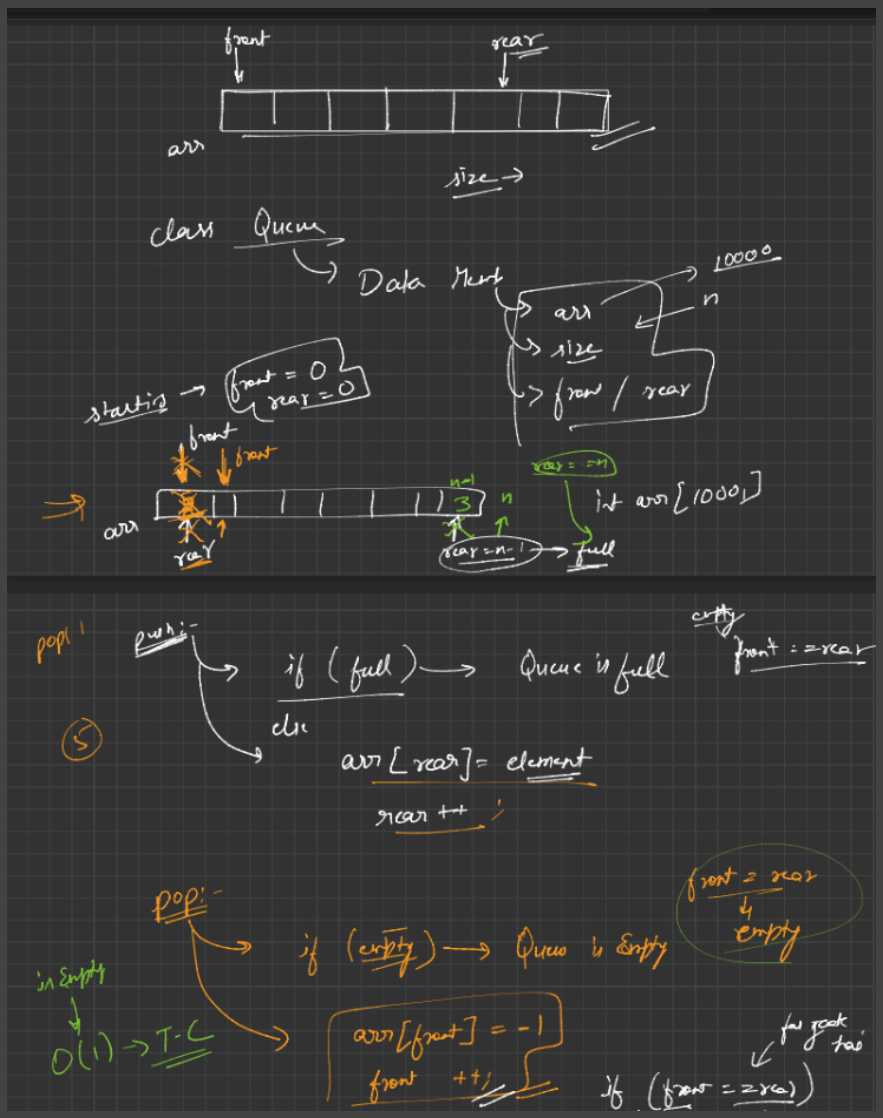

getMin()
pop()
mini → stored
actual
push → 2*curr - mini
pop() → 2*mini - curr
curr minimum
we
previous minimum


push
curr
3
3 < 5 → val = 2*curr - mini
= 2*3 - 5 = 1
push n
2*curr - mini
= 2n - mini
= 2n - prevMinimum
mini → n
1
5
mini 5


pop()
s.top()
1
5
pop()
2*mini - curr
2n - curr
2n - (2n - prevMin)
prevMinimum

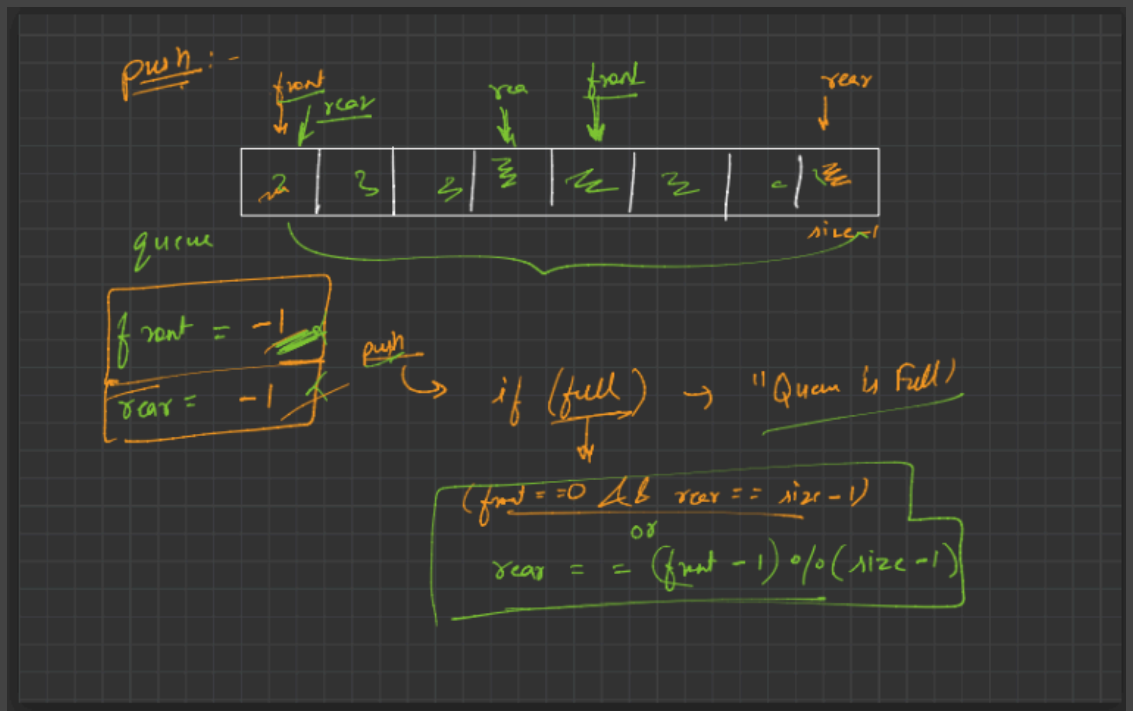
Queue

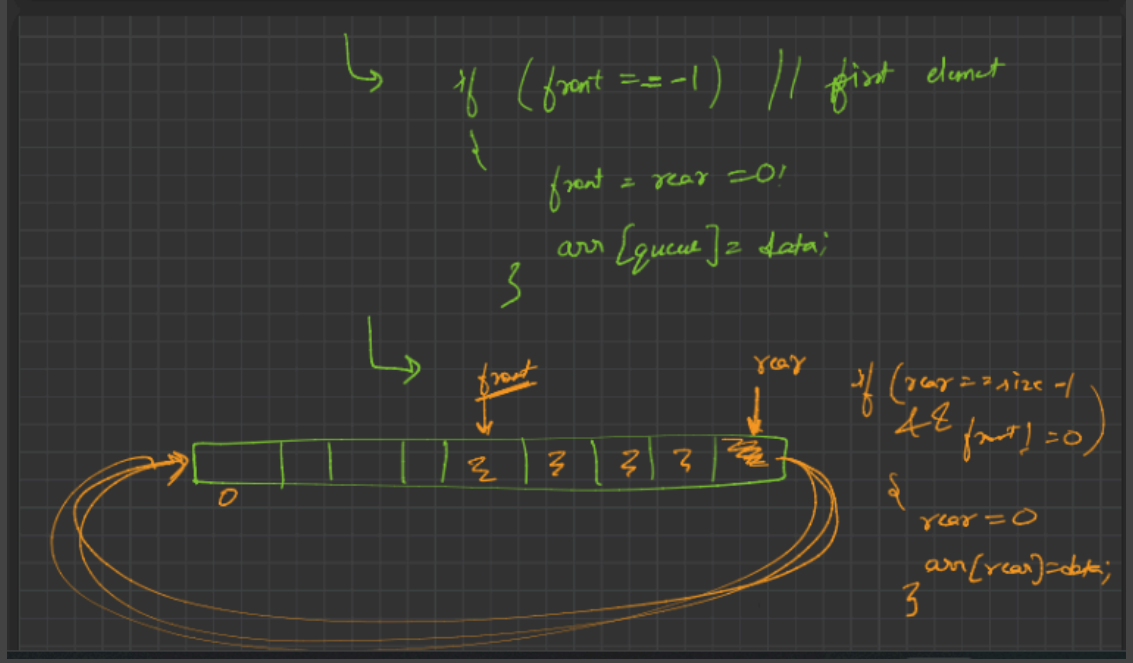
Implementing Queue Using Arrays:

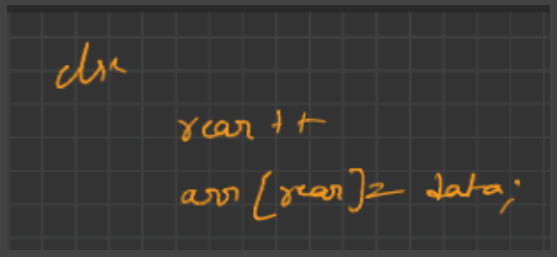


Implementing Queue Using Linked-List:

Circular Queue:







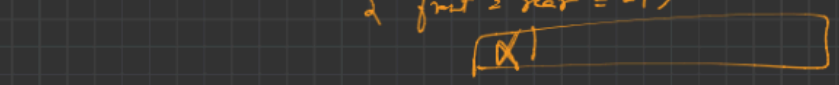
Dequeue / pop :-

→ if (empty) → Queue is Empty

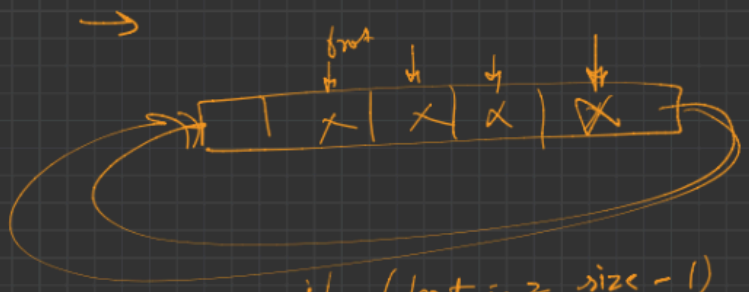
front == -1

→ // single element in queue

if (front == rear)
{ front = rear = -1;



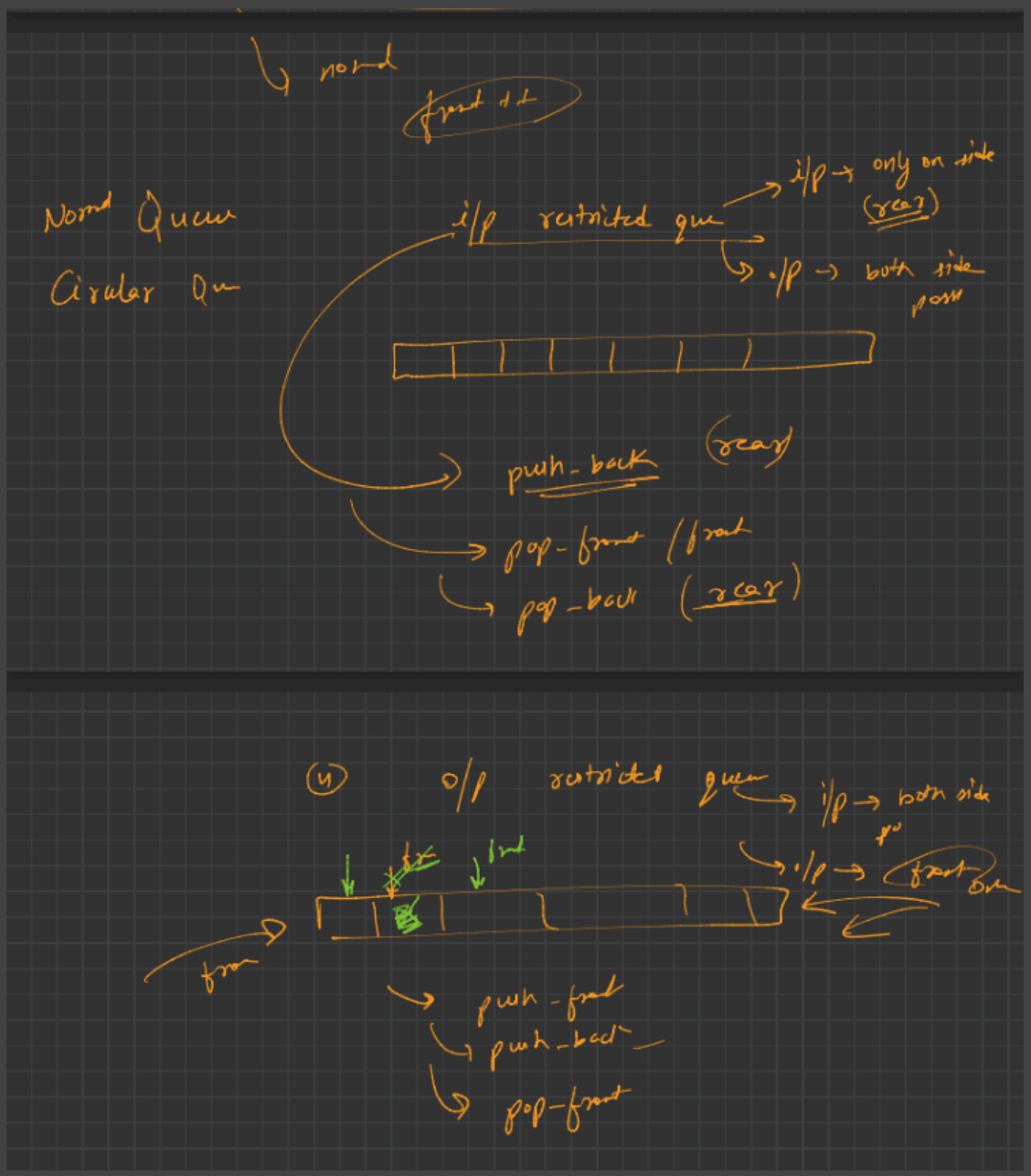
→



if (front == size - 1)
{ front = 0;

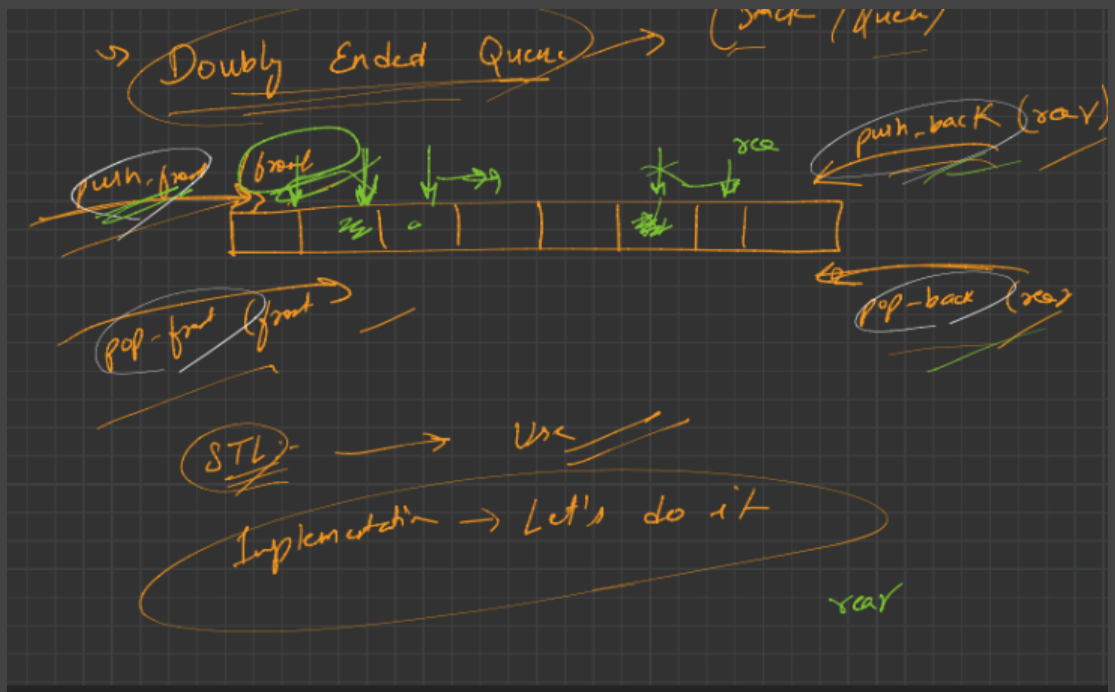
→ front ++;

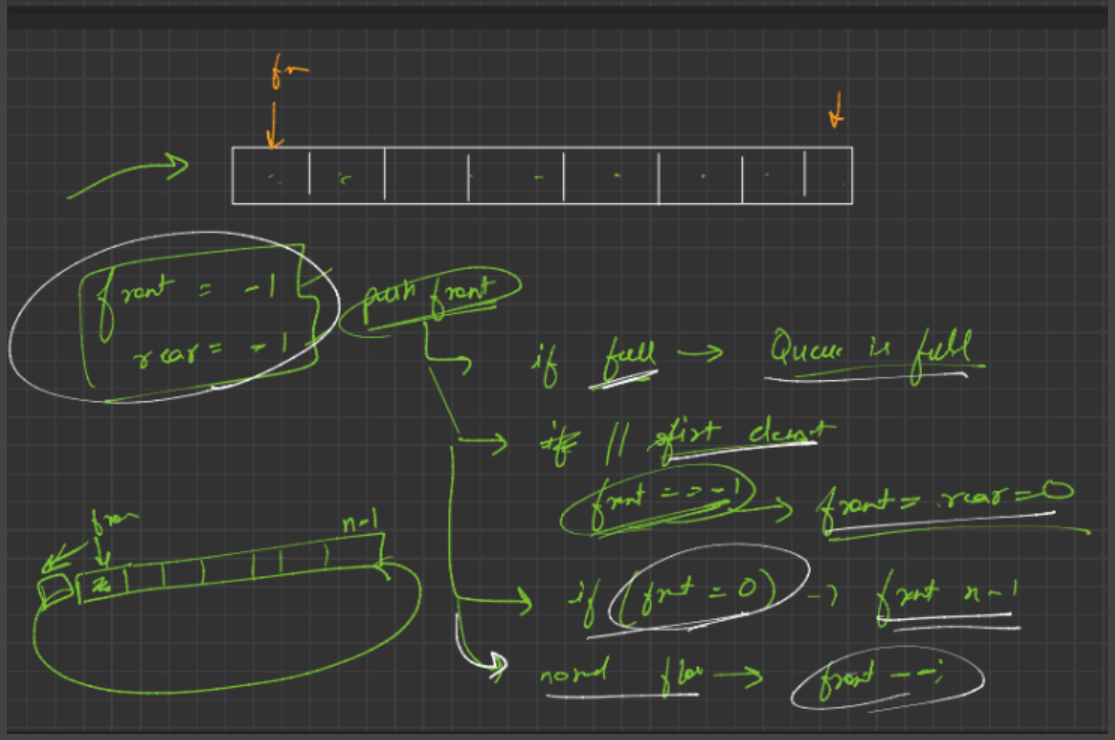
Input Restricted And Output Restricted Queue:

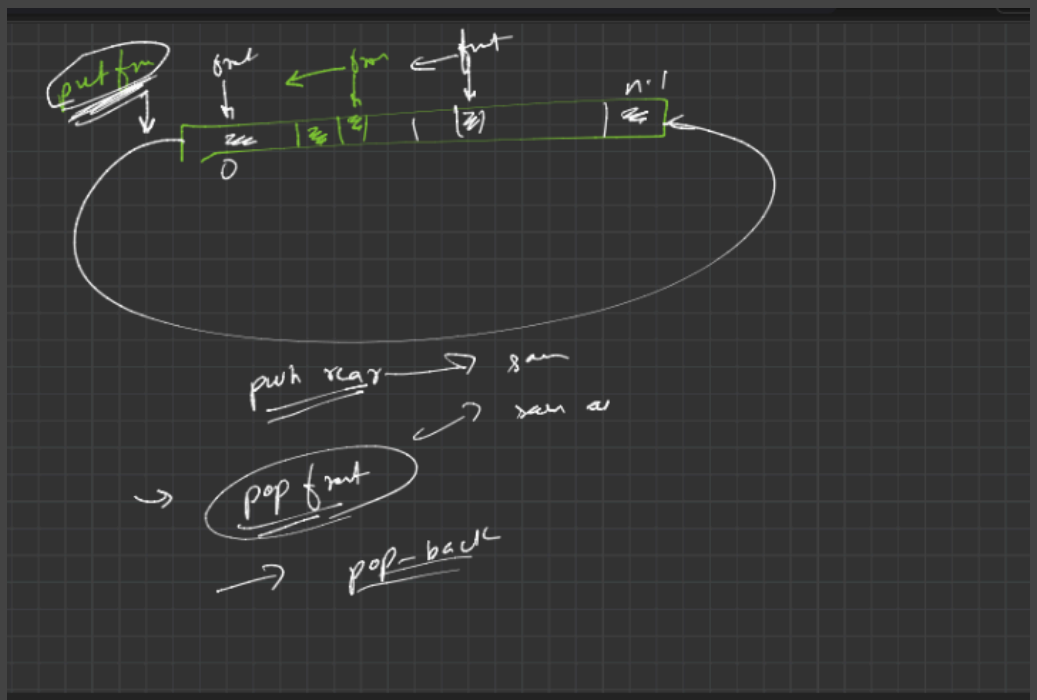


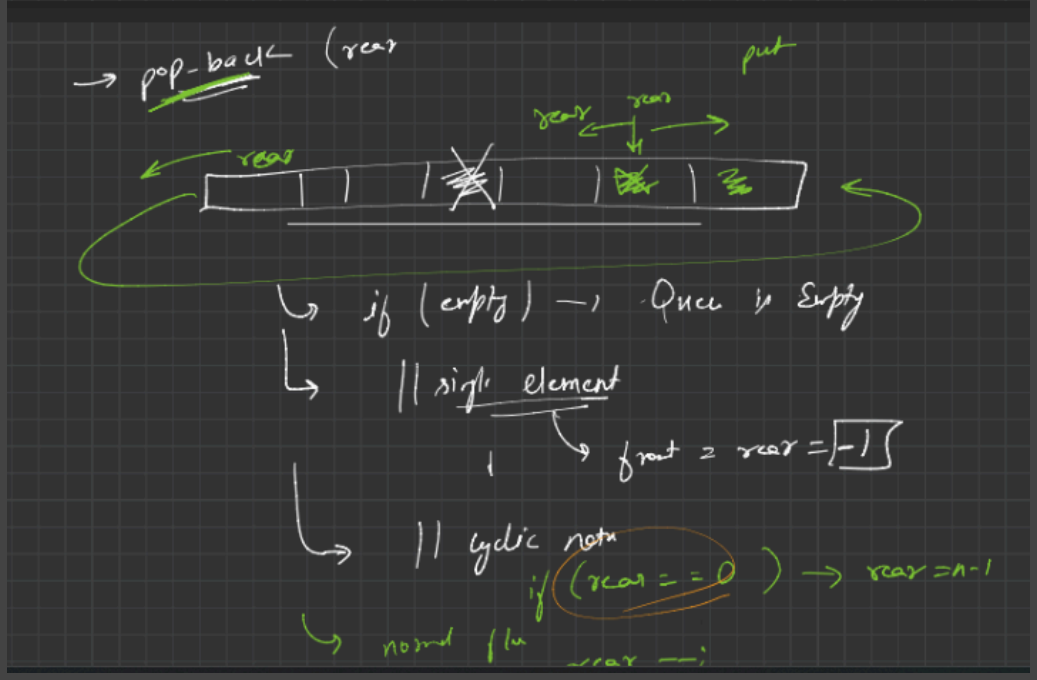
Doubly Ended Queue(Deque) STL Code:

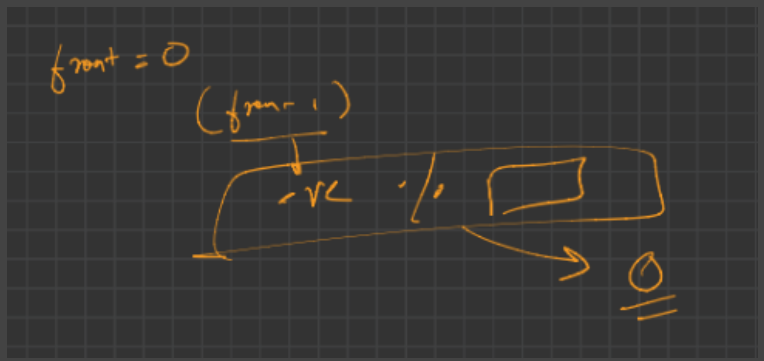
Doubly Ended Queue(Deque) Implementation:



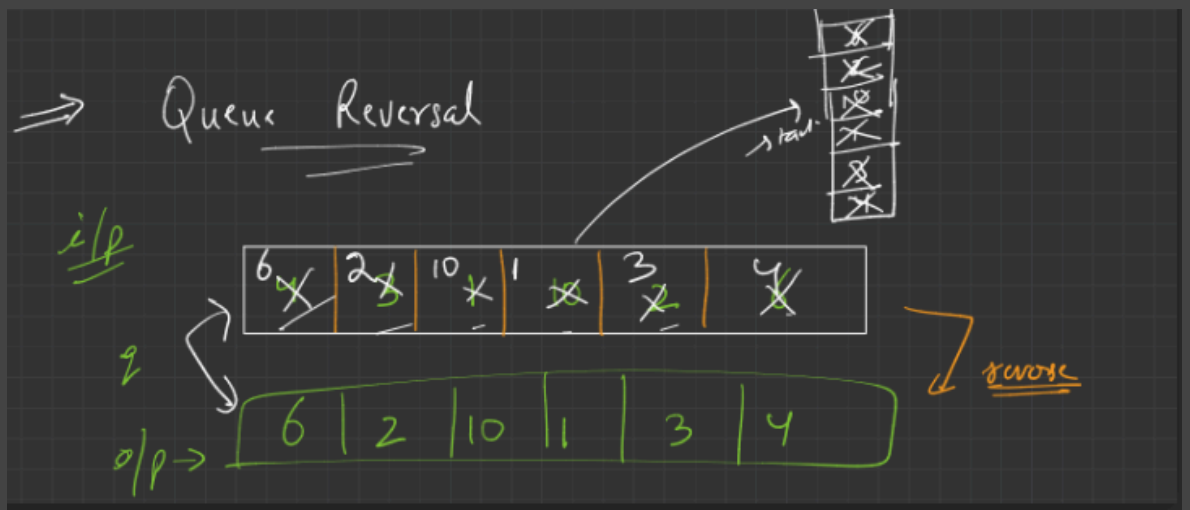


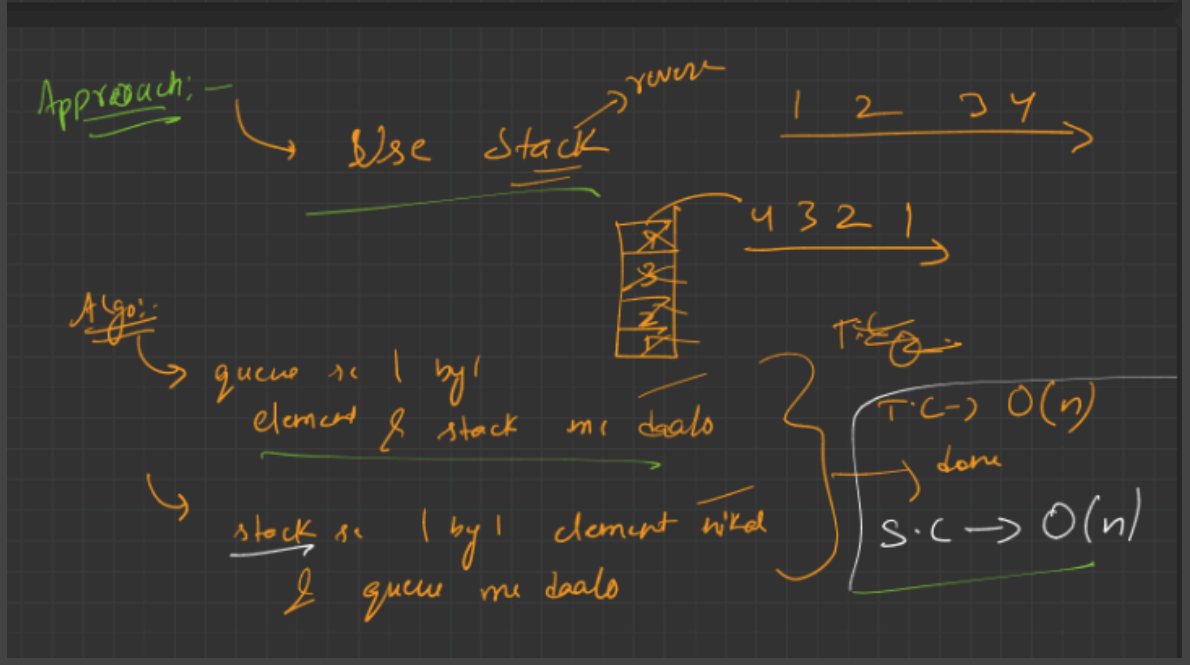

front
front
front
n-1
0
push rear → same
→ pop front
→ pop-back


→ pop-back (rear
rear
rear
rear
front
→ if (empty) → Queue is Empty
→ || single element
front = rear = -1
→ || cyclic note
if (rear == 0) → rear = n-1
→ normal flow
rear --;

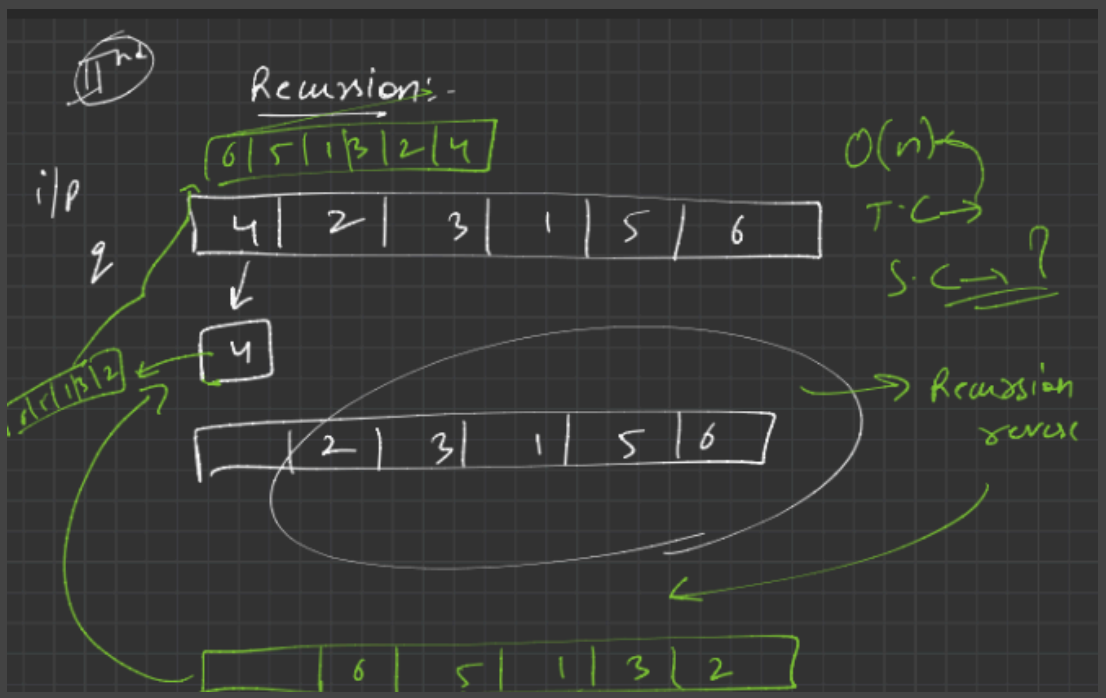

front = 0
(front - 1)
-ve
0

Reverse A Queue Approach-I:

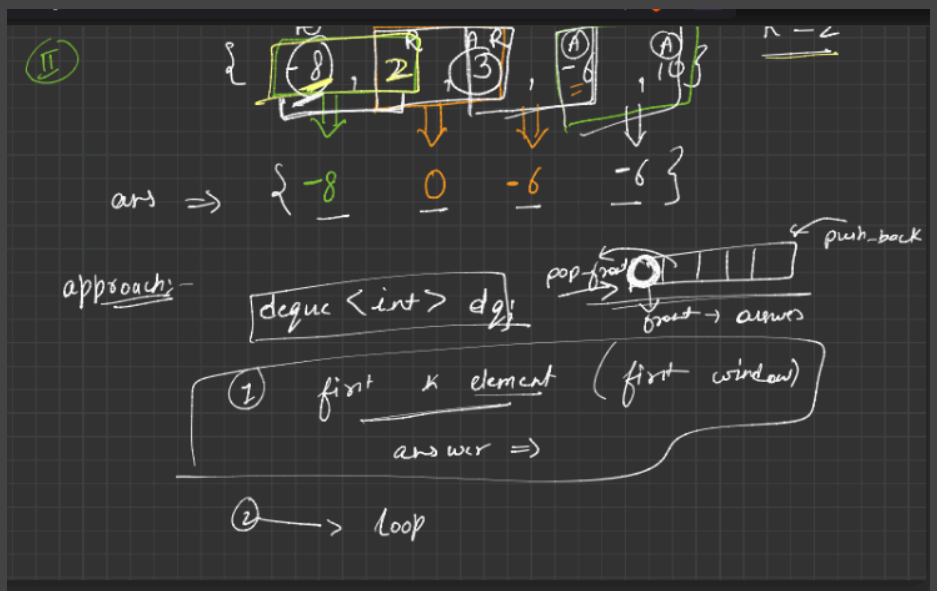


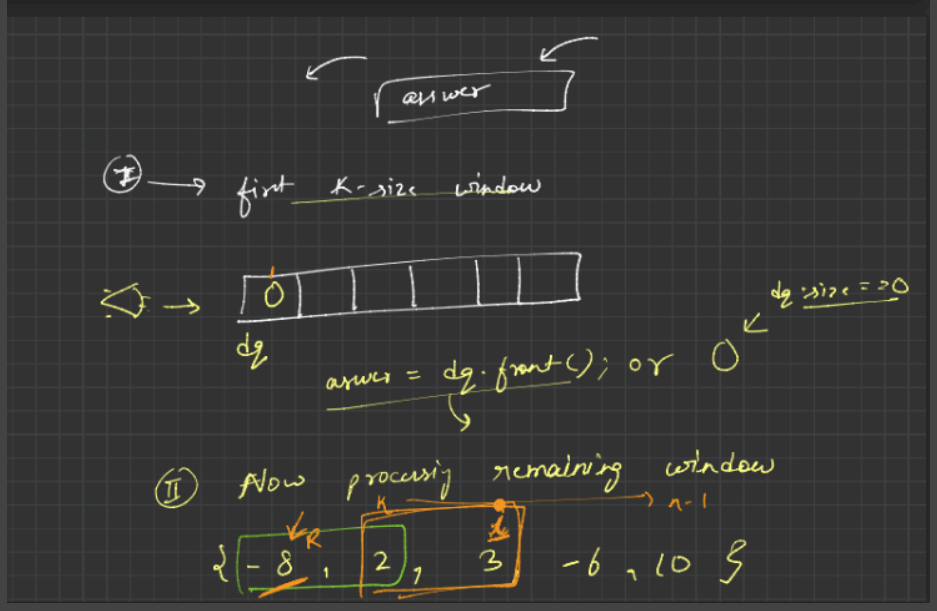


Reverse A Queue Approach-II:



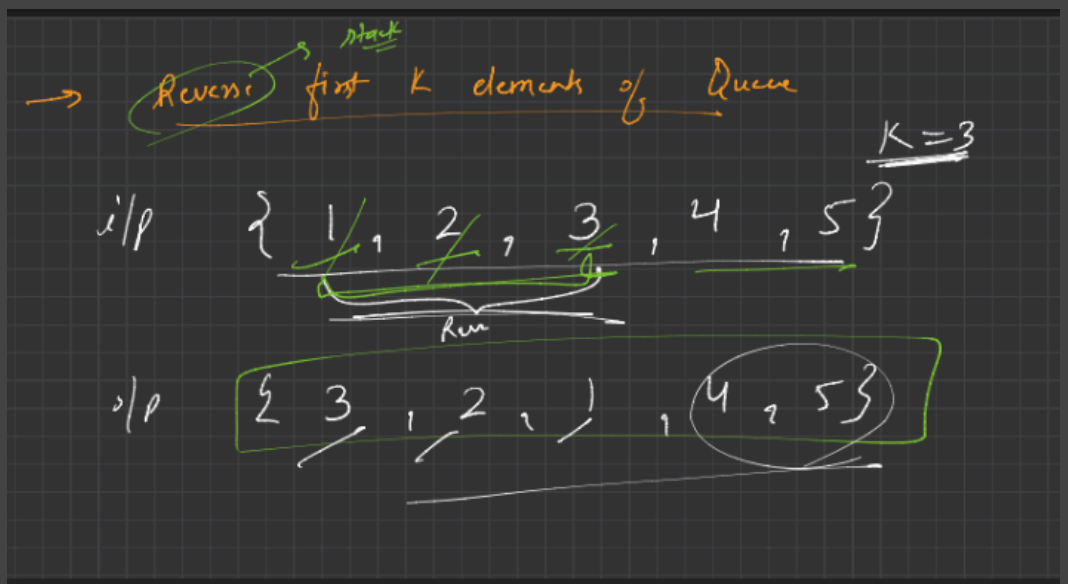
First Negative Integer In Every Window Of Size K:

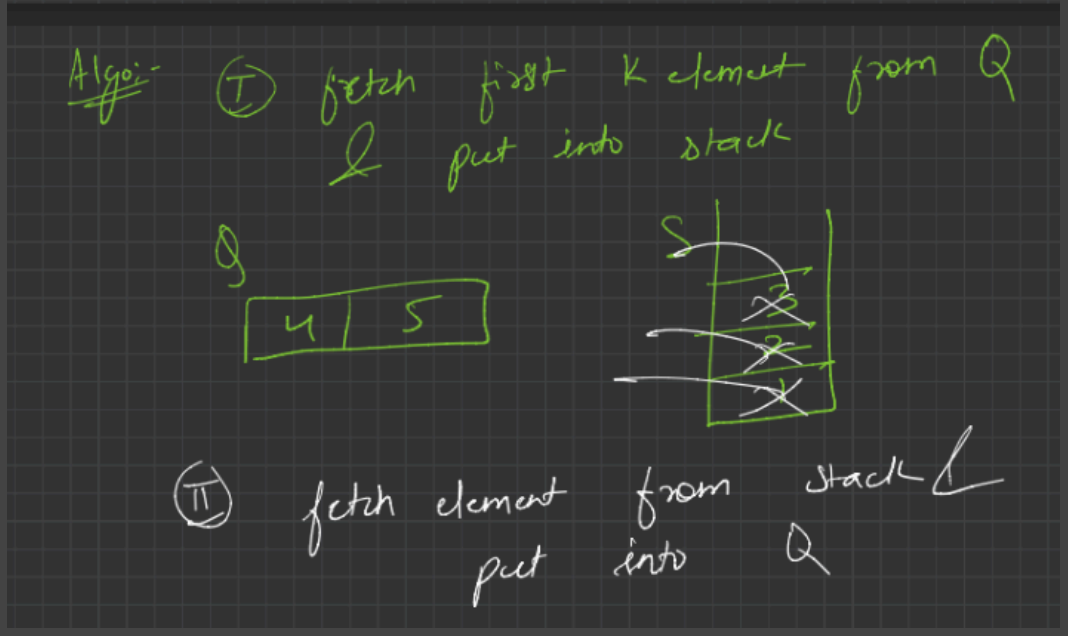


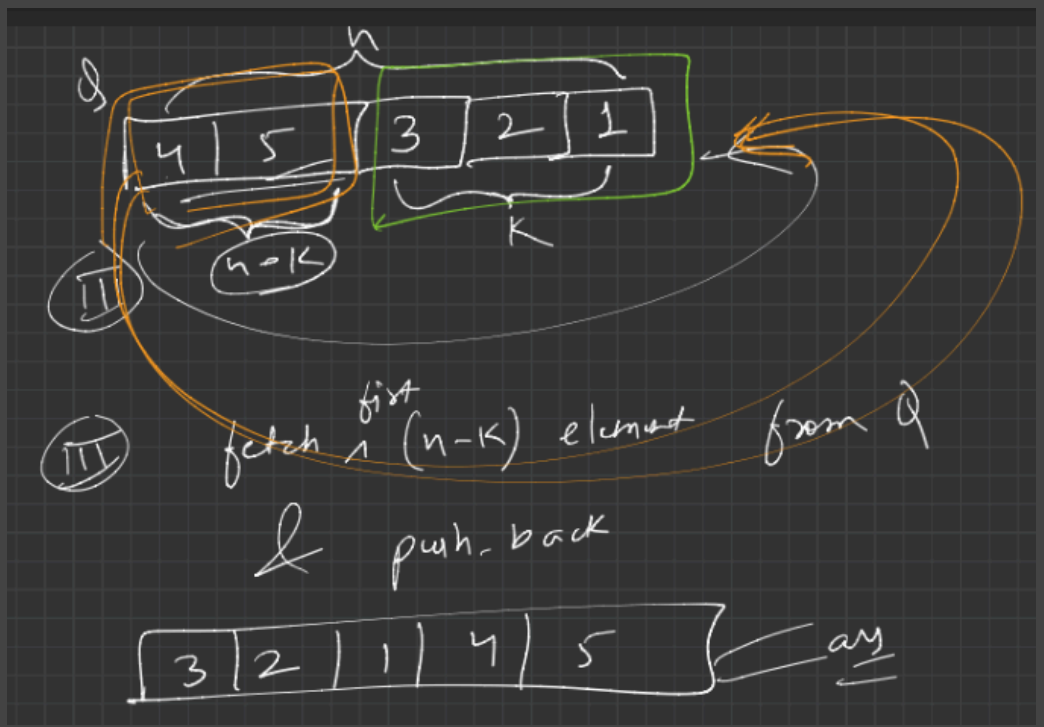



```
// Removal
for ( i -> <n )
{
    // Removal
    if ( dq.front() - i >= k)
        dq.pop_front();
    // addition
    if (arr[i] < 0)
        dq.push_back(i);
ans  dq.size()>0  ans -> dq.front
     size=0       0
```

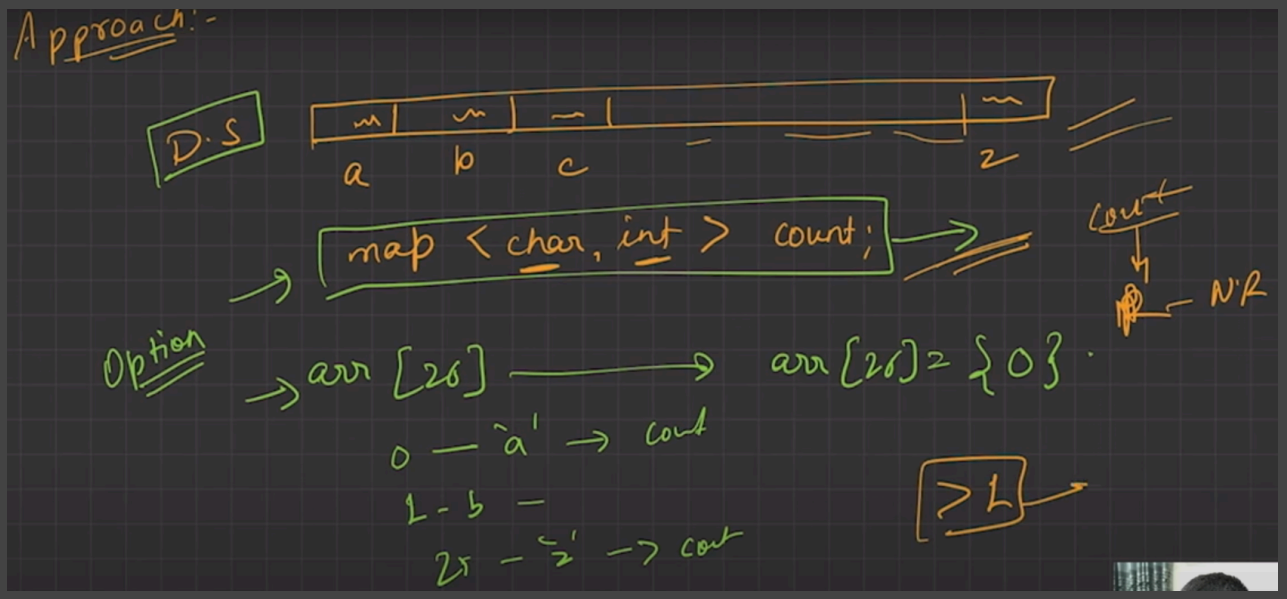
Reverse First K Elements In A Queue:

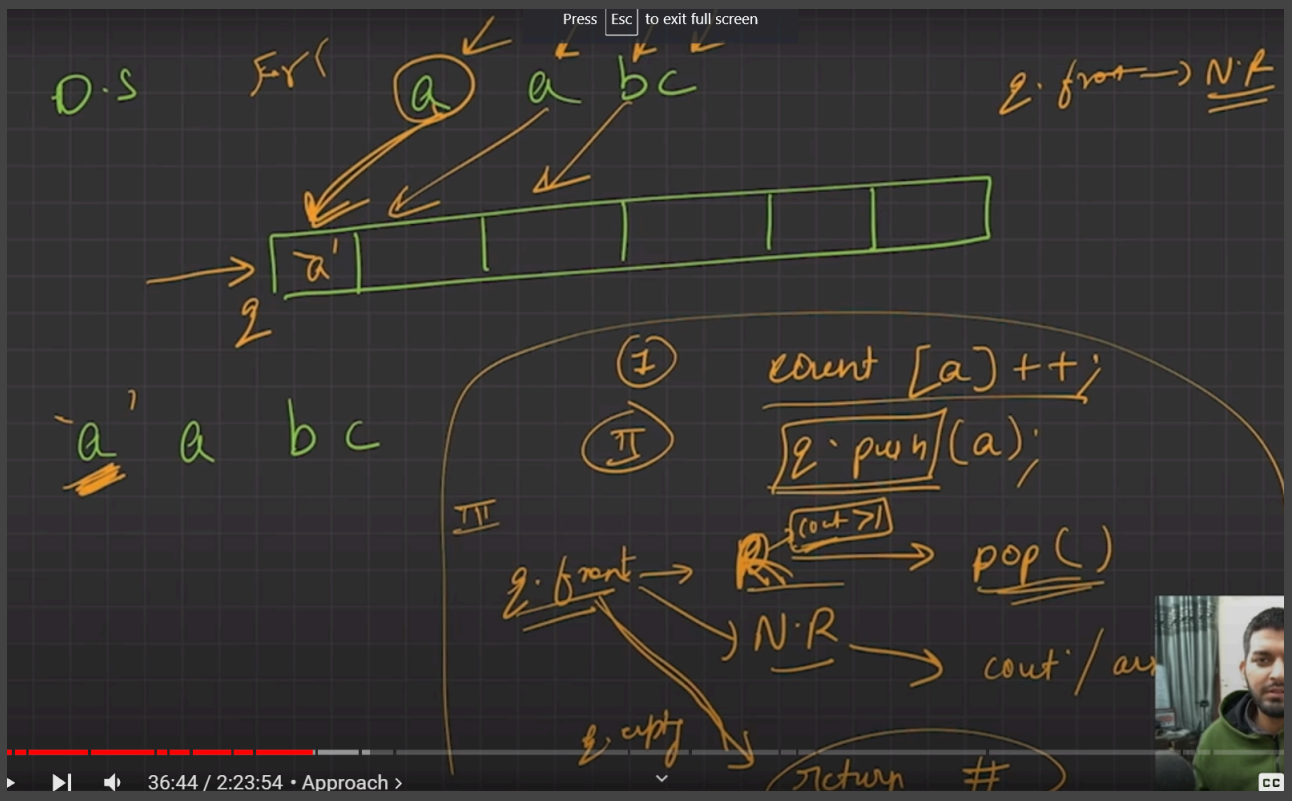




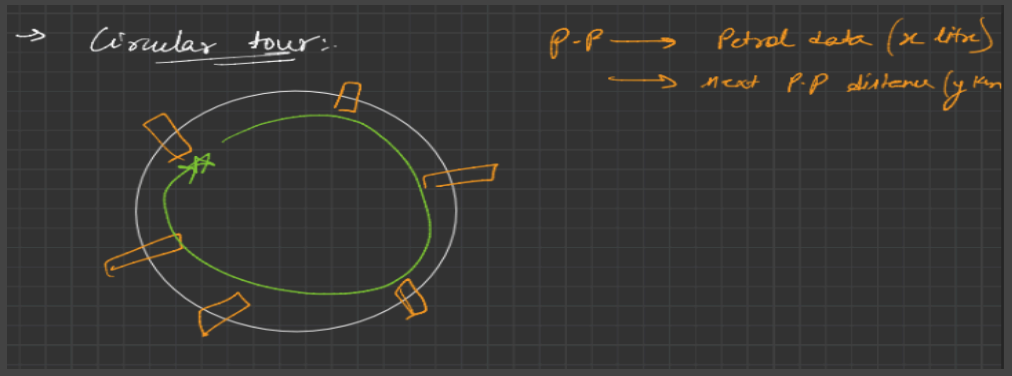


First Non-Repeating Character In A Stream:





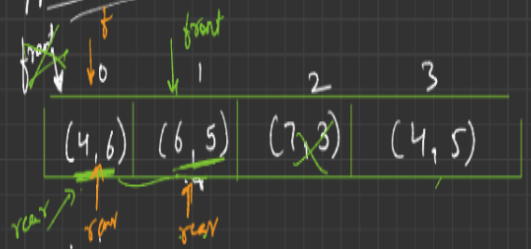
Circular Tour:



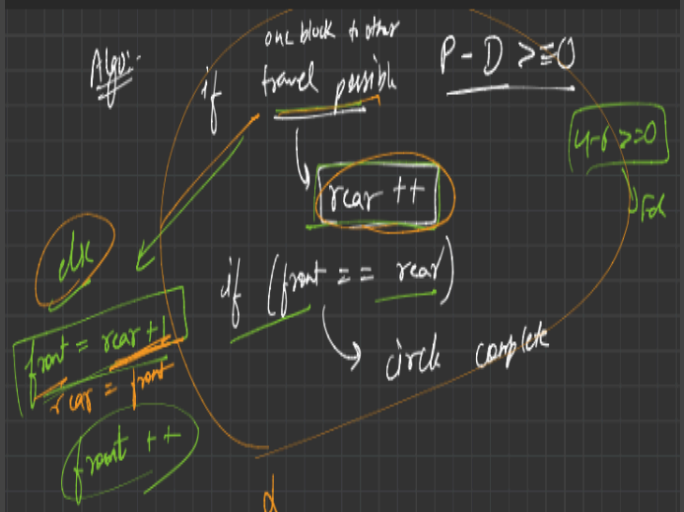
approach #1 :- B.F

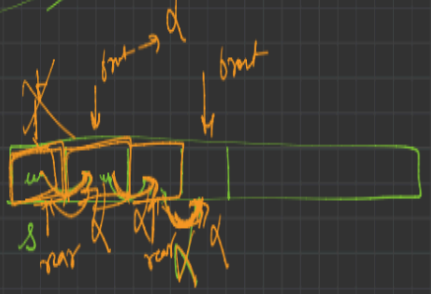
$\rightarrow O(n^2)$

approach #2 :-



Algo :-





balance >=0

Algo :- start block to next block

if (travel possible)

rear ++

else

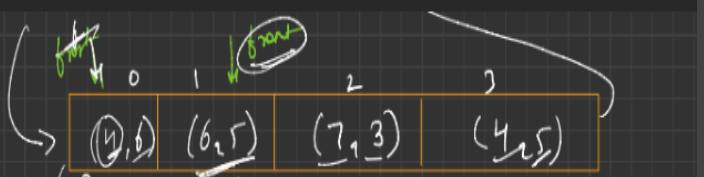
front = rear + 1

start = front

rear = front

B.C $\rightarrow$ if (front == rear)

$\rightarrow$ cycle complete



(1) P - D >= 0 False

4 - 6 = -2 < 0

front = rear + 1

(I) P - D >= 0 !

8 - 5 = 3 >= 0

balance = 4

balance = 1 + 7 - 8 - 3 = 5 >= 0 T

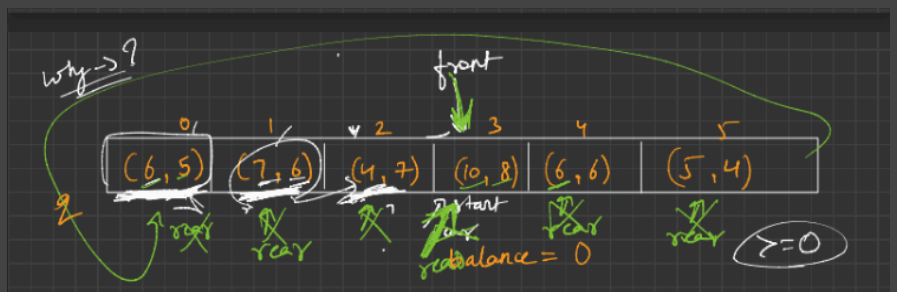
balance = 5 + 4 = 9 - 5

= 4 >= 0 T

balance = 4 + 4 - 6

= 8 - 6

= 2 >= 0 T



why -> front ++ ∝

front = rear + 1

balance = balance + P - D

= 0 + 6 - 5 = 1 >=0 -> TRUE

balance = 1 + 7 - 6

= 8 - 6 = 2 >=0 -> TRUE

balance = 2 + 4 - 7

= 6 - 7 = -1 >=0 -> false

worst case

Block -> 2 visit

at

single visit

balance = 0

balance = 0 + 10 - 8

= 2 >=0 TRUE

balance = 2 + 6 ≠ 6

2 (2) >=0 TRUE

= 2 + 5 - 4

= 7 - 4

= 3 >=0 -> TRUE

= 3 + 6 - 5

= 9 - 5 = 4 >=0 T

= 3 + 6 - 5

= 9 - 5 = 4 >=0 T

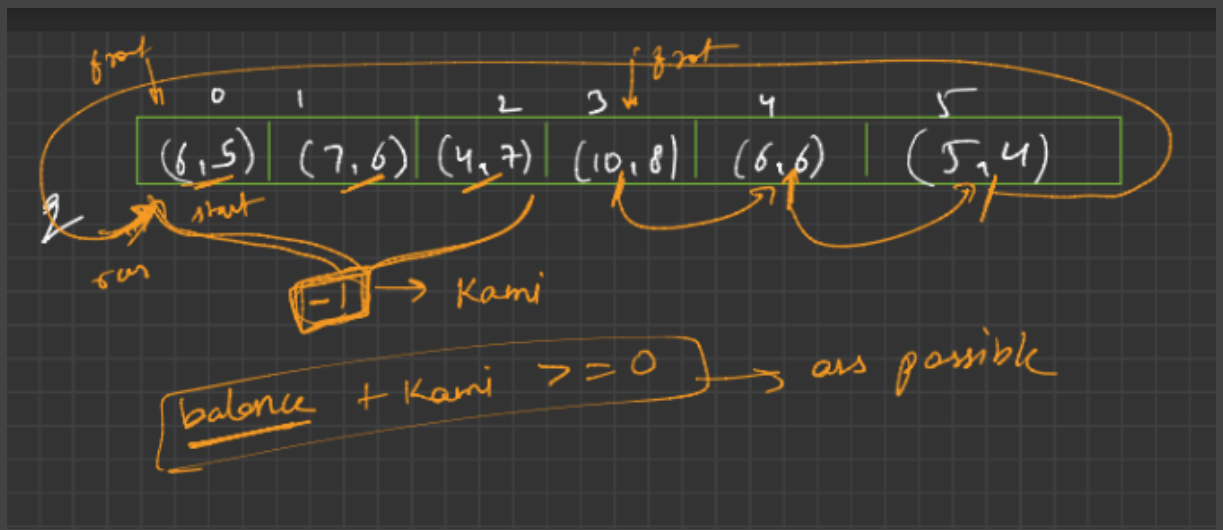
= 4 + 7 - 6

= 11 - 6 = 5 >=0 TRUE

= 5 + 4 - 7

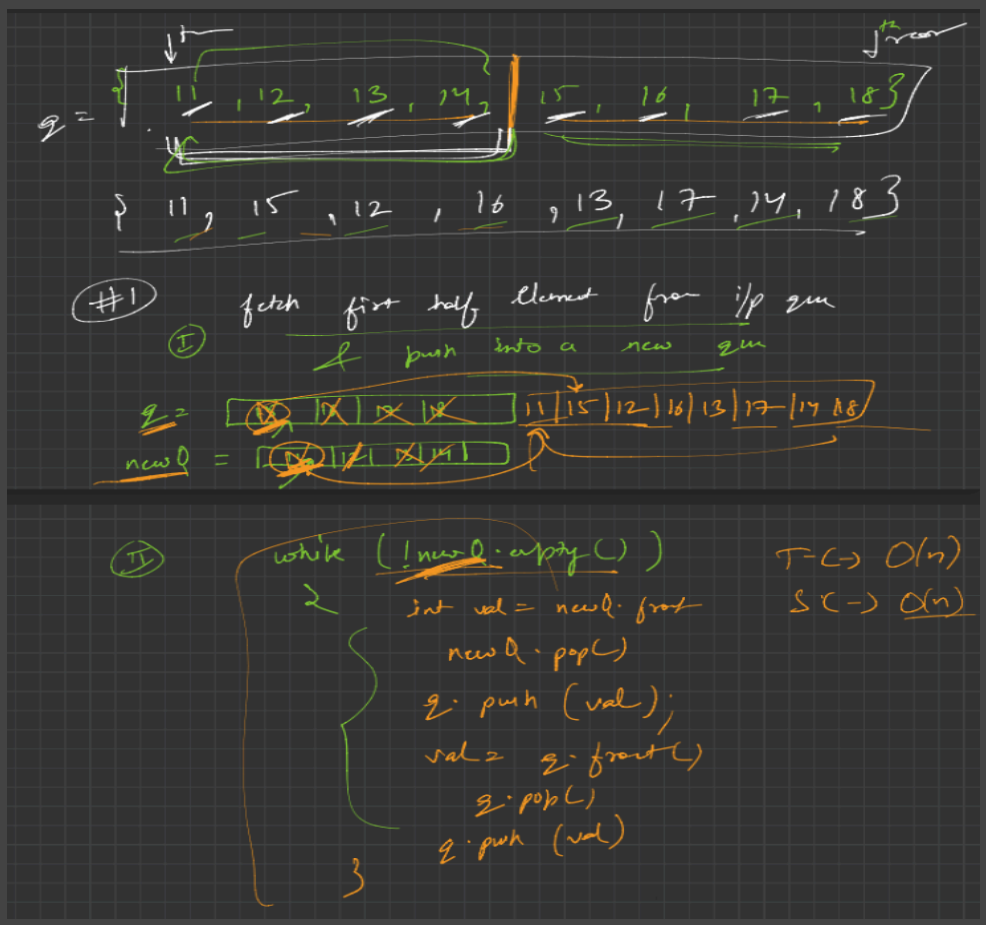
= 9 - 7 = 2 >=0

TRUE

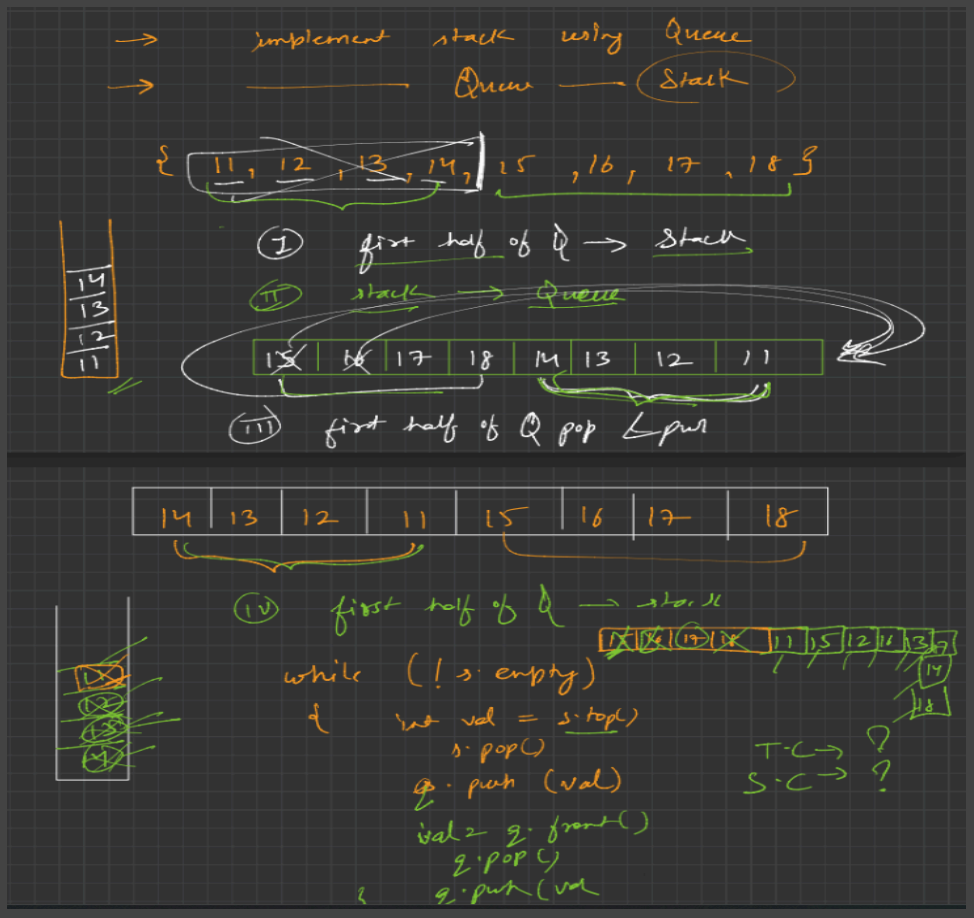


balance + Kami >=0 -> as possible

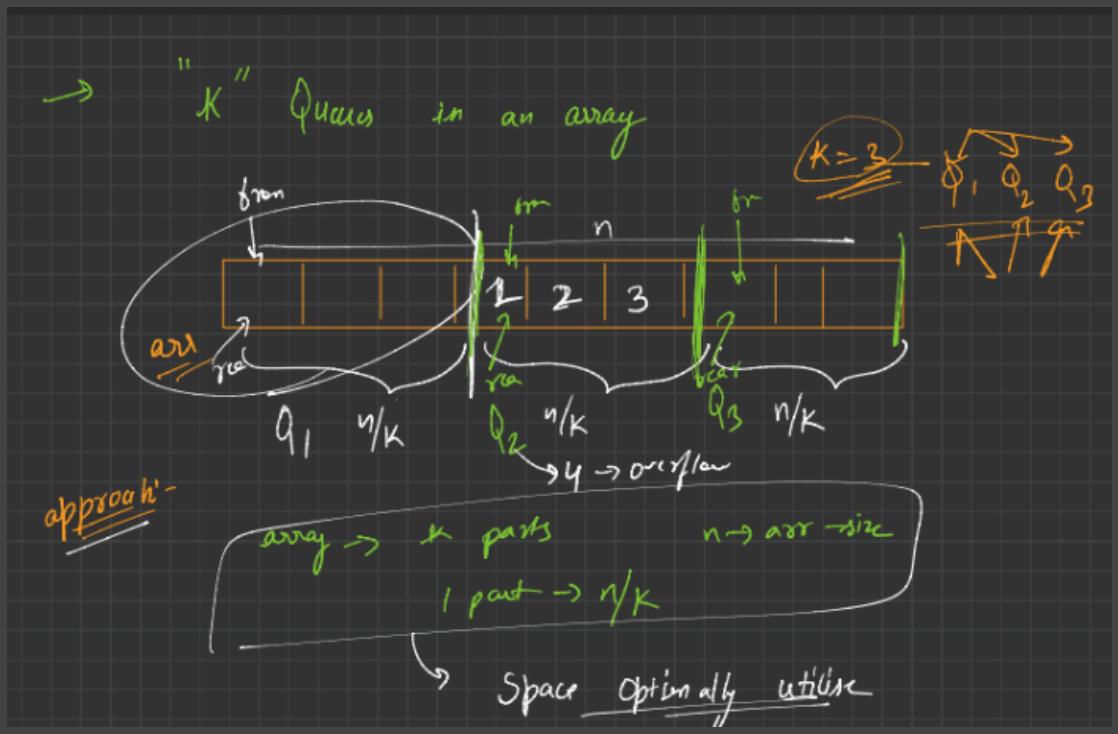
Interleave First and Second Halves Of A Queue:



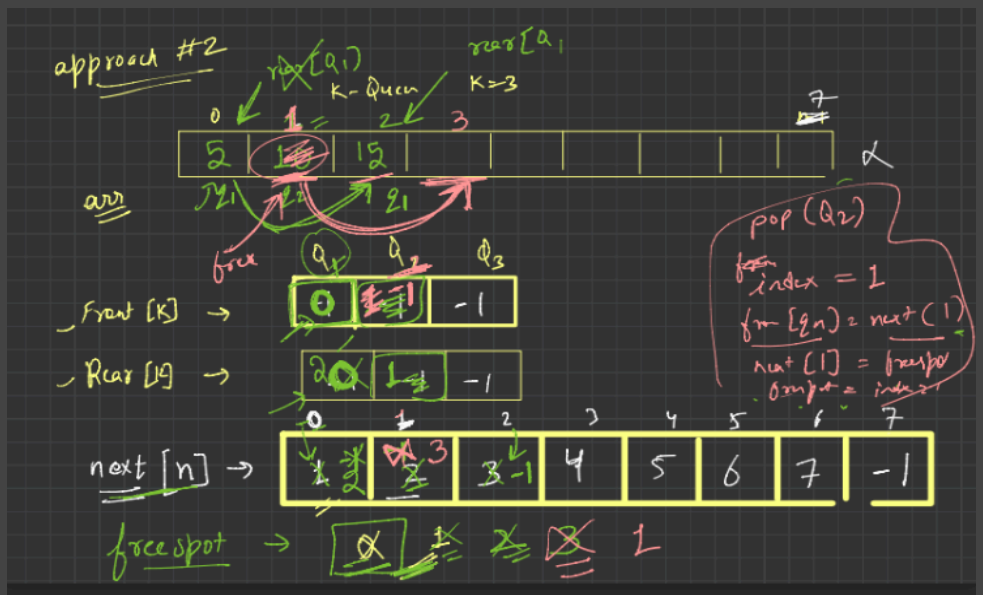
Interleave First and Second Halves Of A Queue Using Stack:

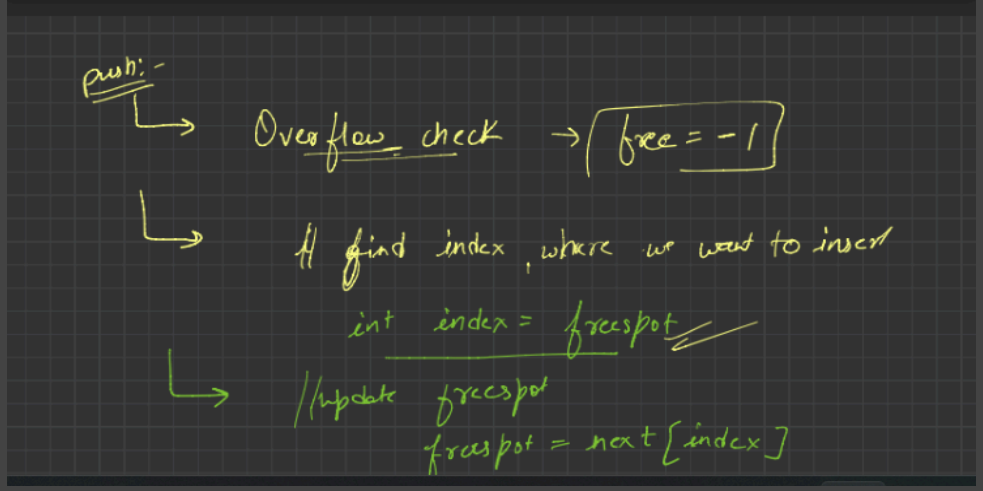


Insert K Queues In An Array Approach-I:

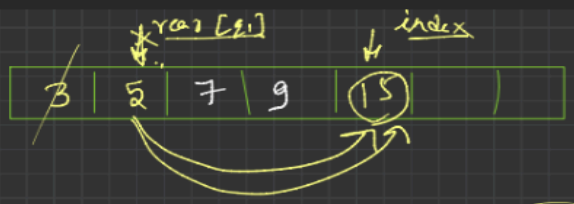


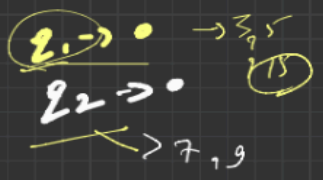
Insert K Queues In An Array Approach-II:





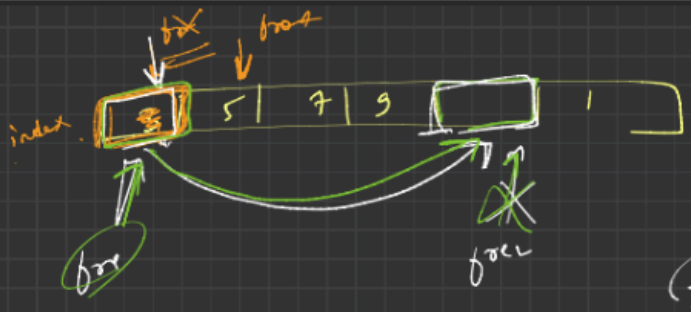
```
↳ // if first element
   if (front[qn] == -1)
       front[qn] = index;
   else
   {
       next[rear[qn]] = index;
   }
```





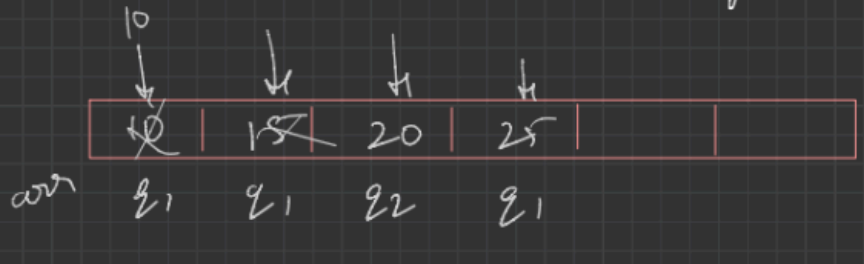
next[rear[qn]] = index;

```
↳ next[index] = -1;
   // point rear to index
↳ rear[qn] = index;
↳ // push element
   arr[index] = x;
```

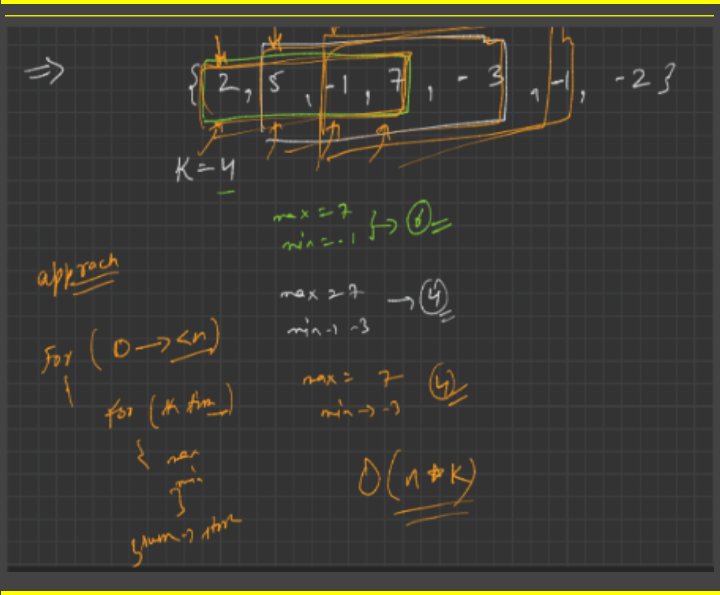


next[index] = free

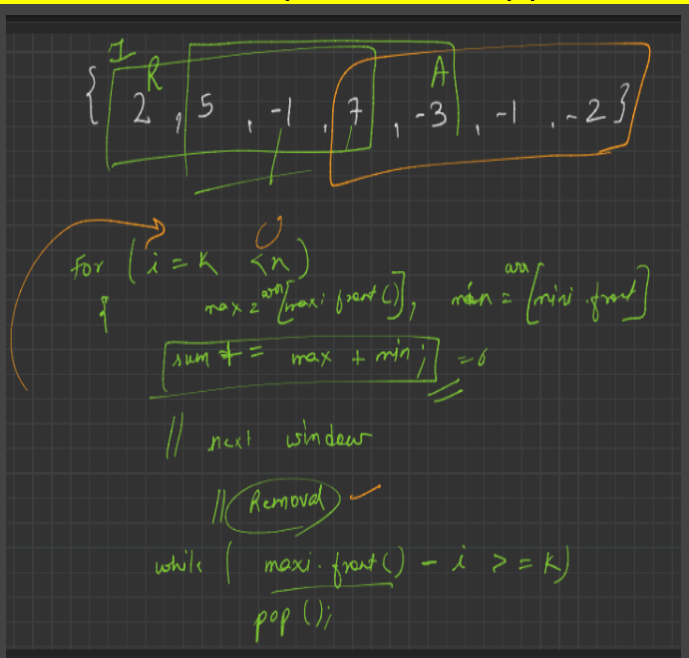
free = index

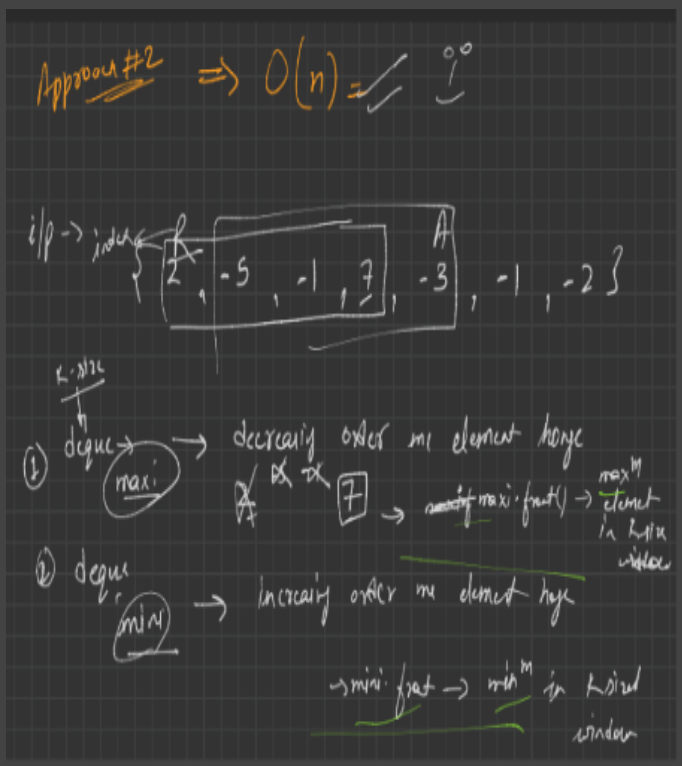


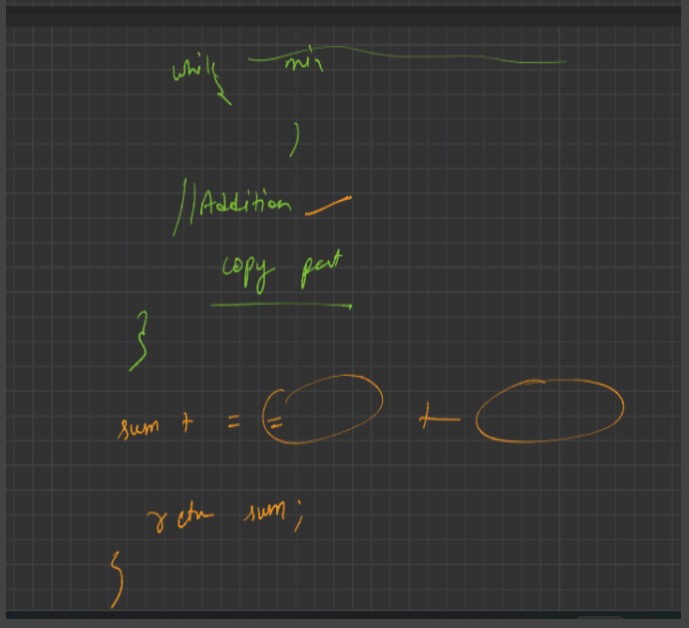
Sum Of Minimum And Maximum Elements Of All Subarrays Of Size K Approach-I:

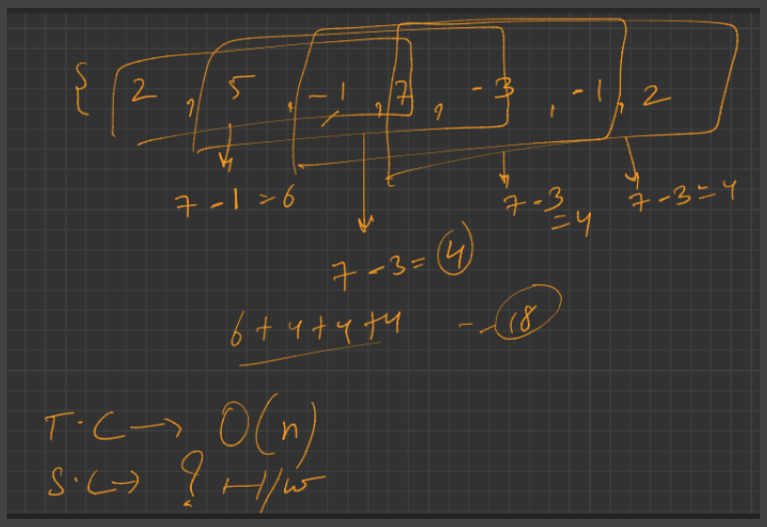


Sum Of Minimum And Maximum Elements Of All Subarrays Of Size K Approach-II:









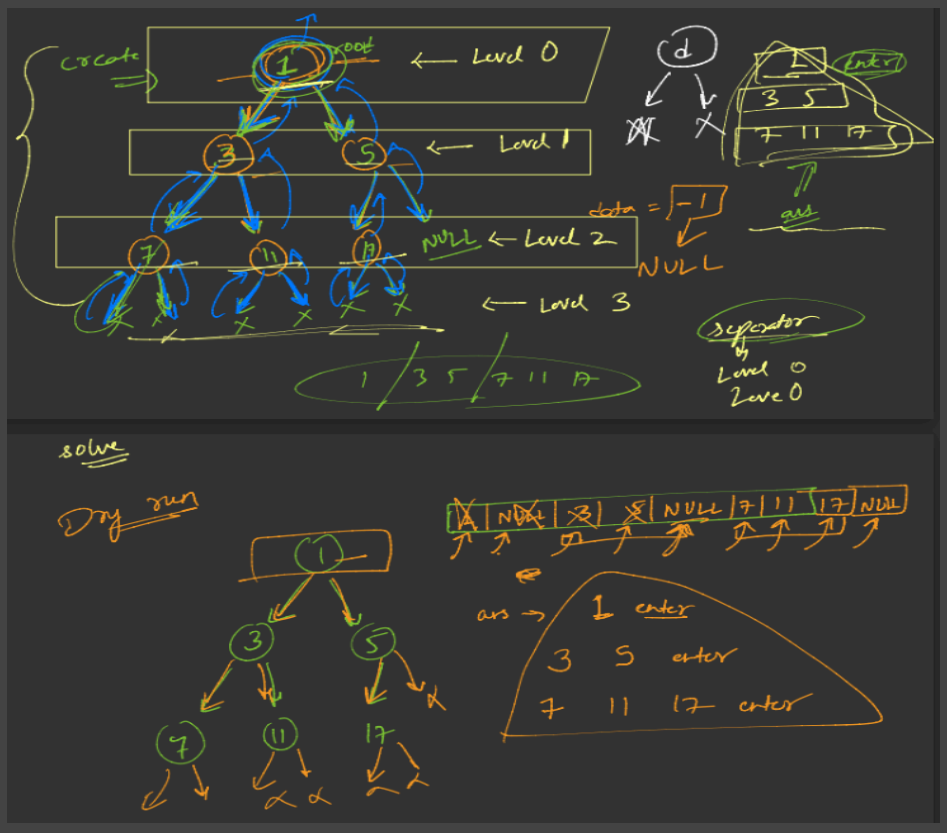
1. For the first k elements: The code enters the first for loop and processes the first k elements of the array. Each element is added to the deques maxi and mini after removing elements from the back of the deques that are not needed. This process takes constant time for each element, so for k elements, it takes O(k) time.
2. For the remaining elements: The code enters the second for loop and processes the remaining n-k elements of the array. For each element, it first removes elements that are out of the current window from the front of the deques. Then it adds the current element to the deques after removing elements from the back of the deques that are not needed. This process also takes constant time for each element, so for n-k elements, it takes O(n-k) time.

Since the code processes k elements in the first loop and n-k elements in the second loop, the total time complexity is O(k) + O(n-k) = O(n), where n is the total number of elements in the array. This is why we say each element in the array is processed only once in a single pass. The operations inside the loops are all constant time operations, so they do not change the overall time complexity.

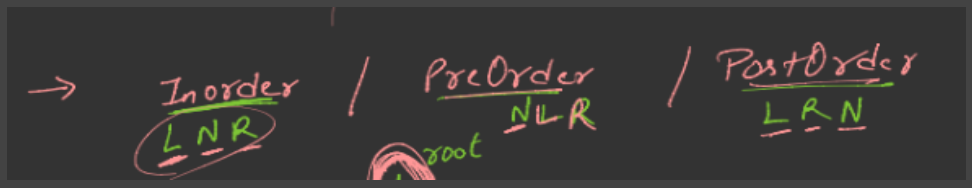
Binary Tree

Basic Terms Of Binary Tree:

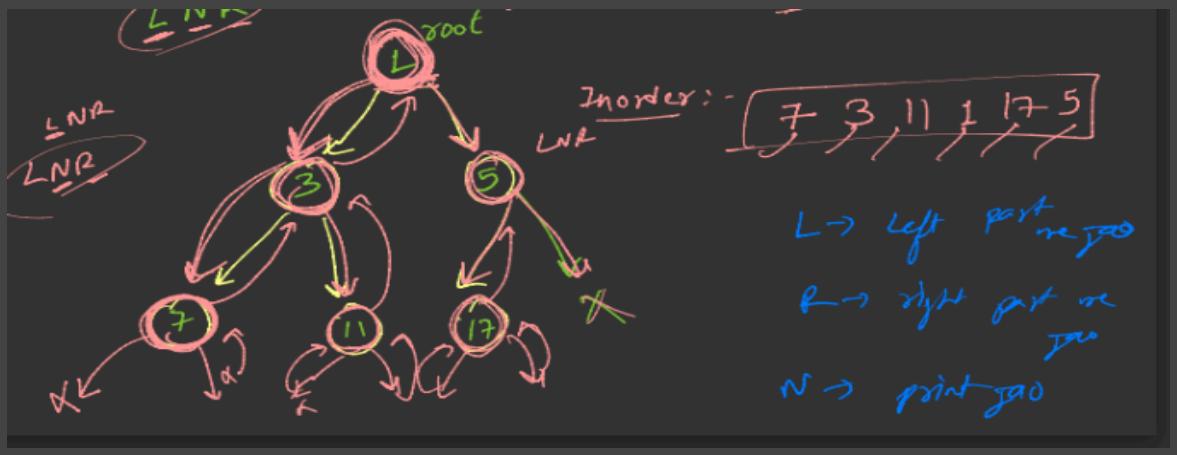
Creation Of Binary Tree & Level Order Traversal(LOT):



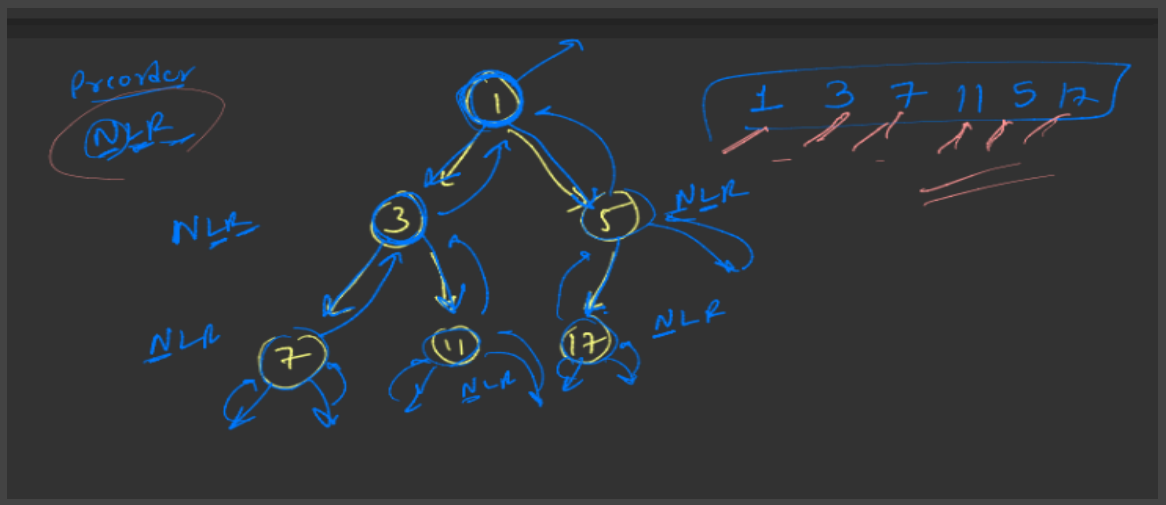
Reverse Level Order Traversal:



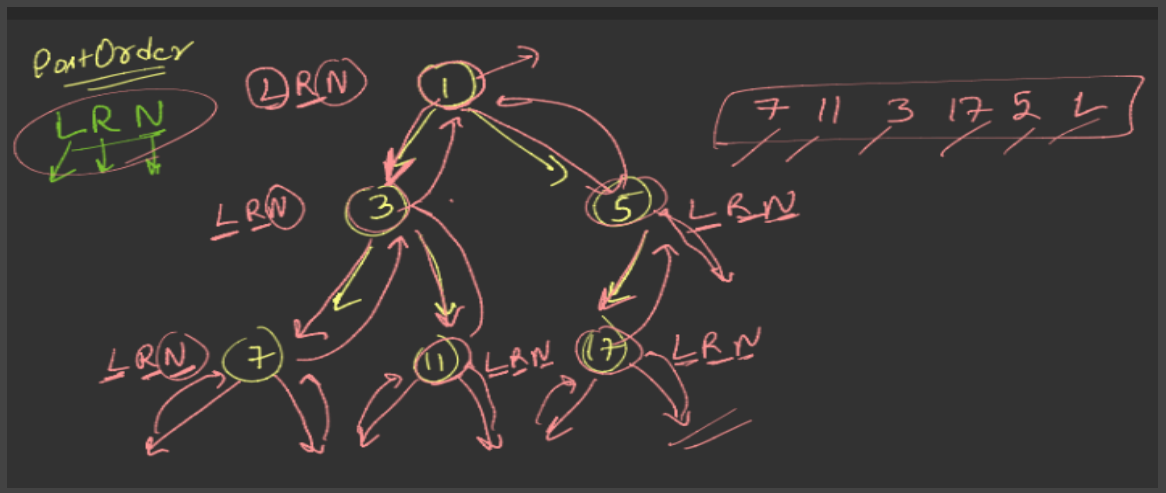
Inorder Traversal:



Preorder Traversal:



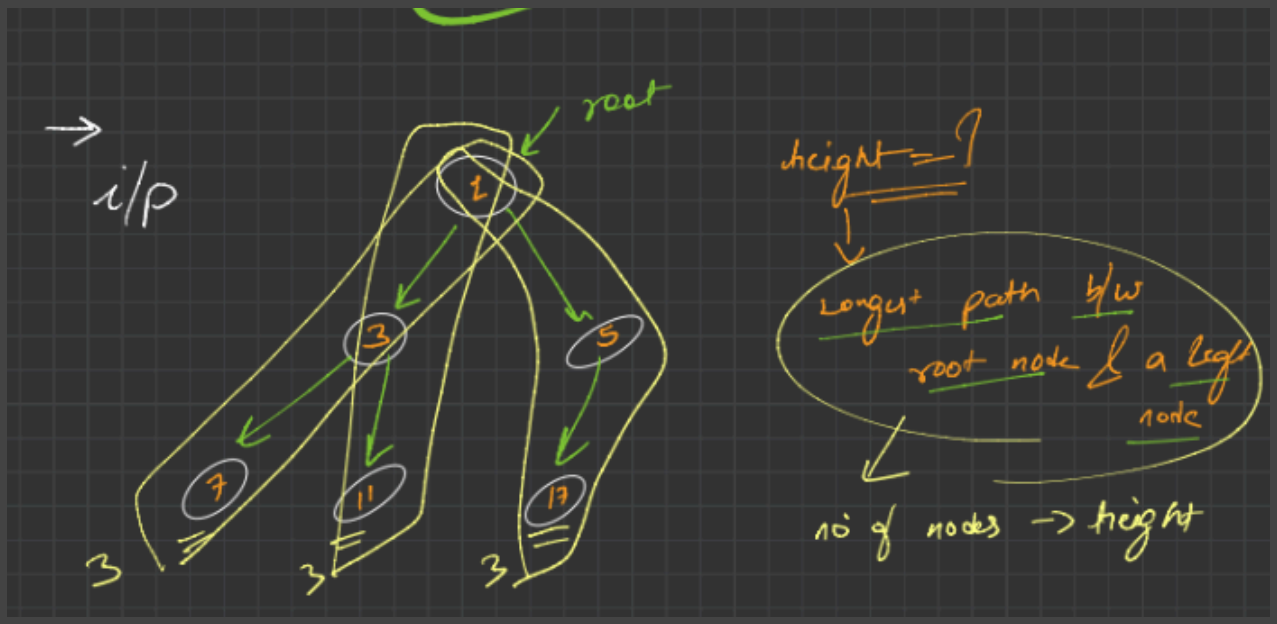
Postorder Traversal:

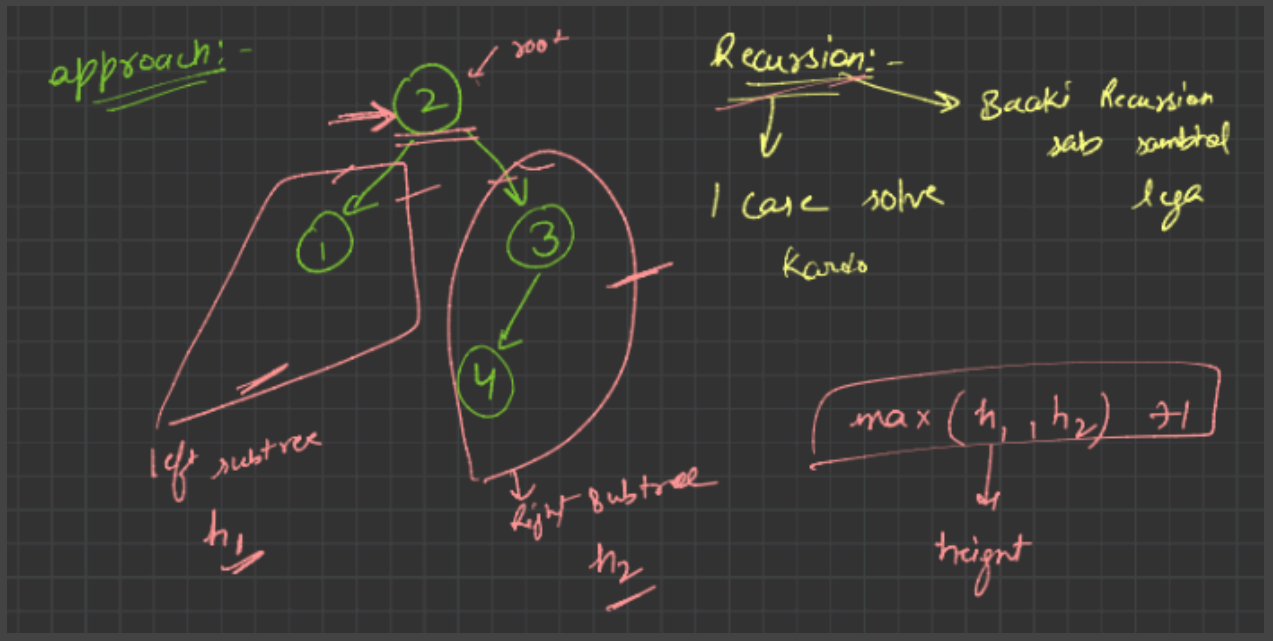


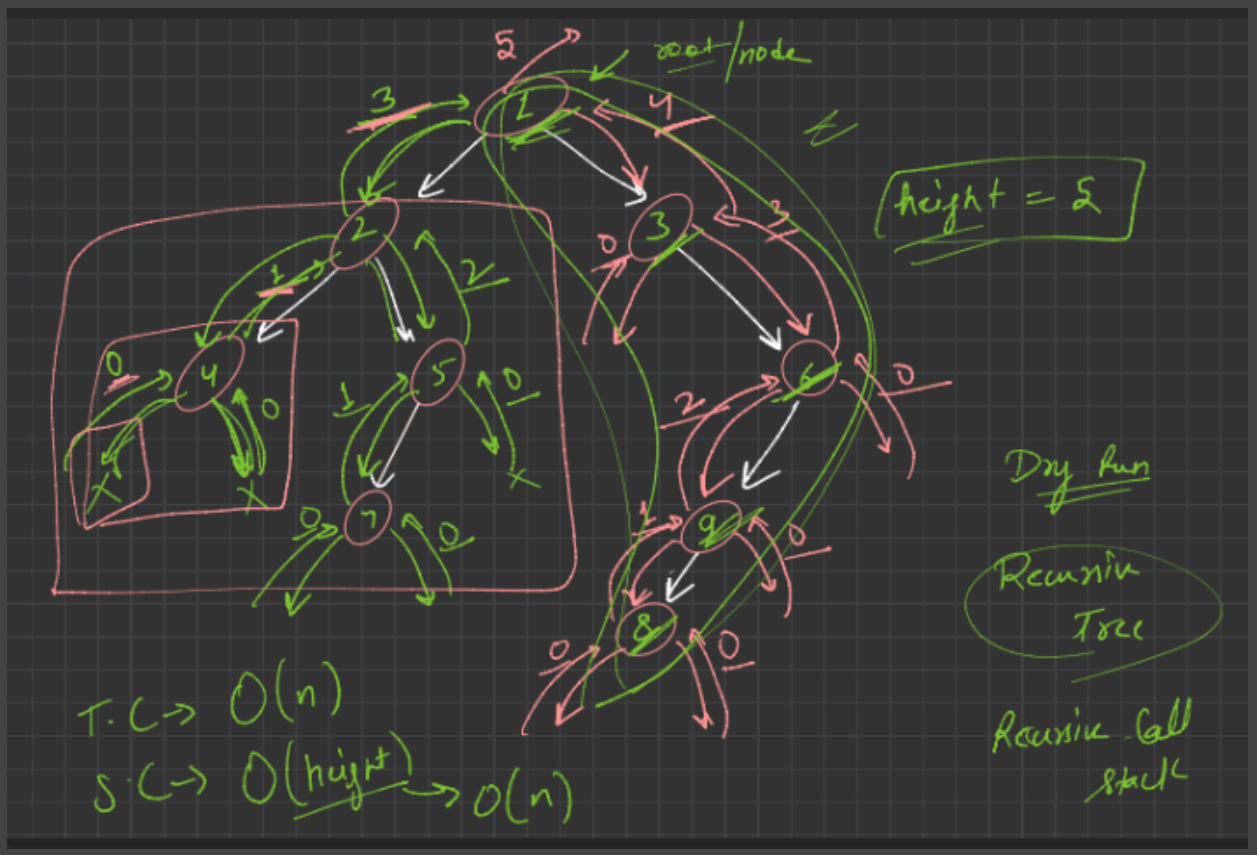
Build Binary Tree From Level Order:

Count Leaf Nodes:

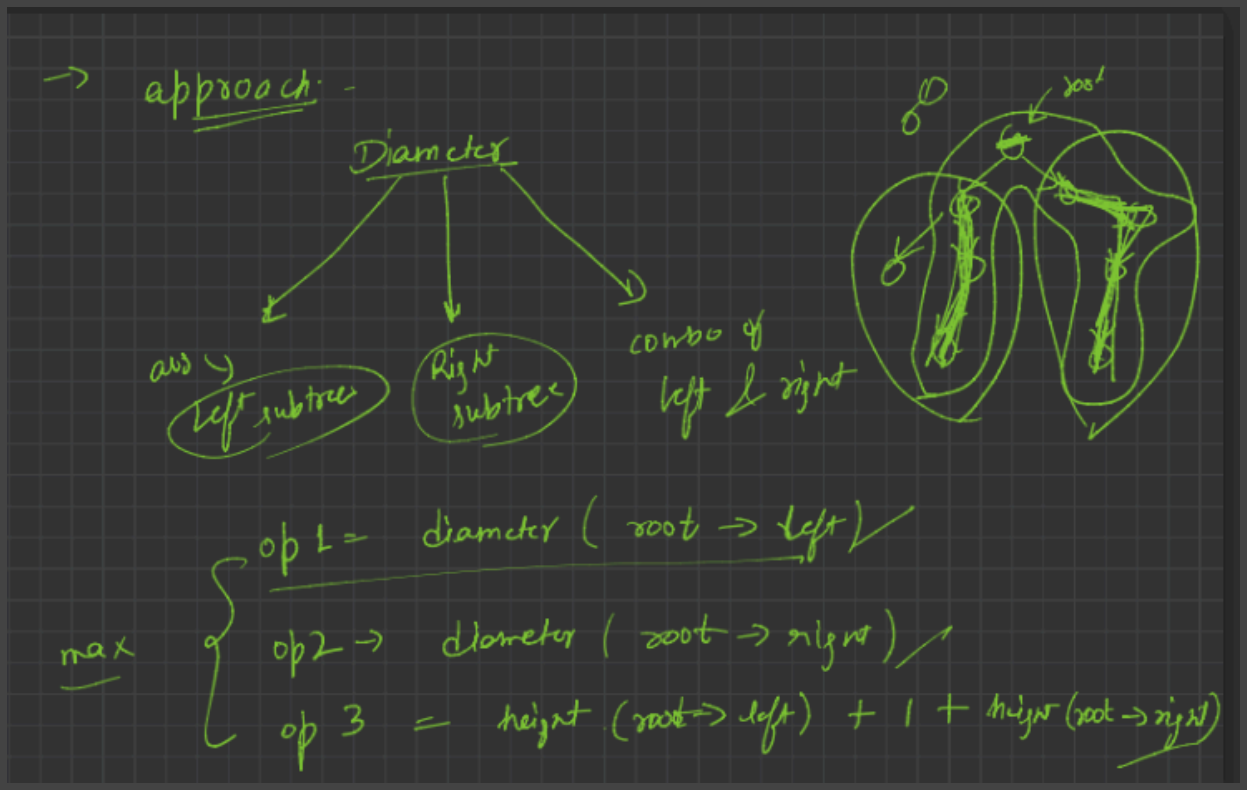
Height Of A Binary Tree:



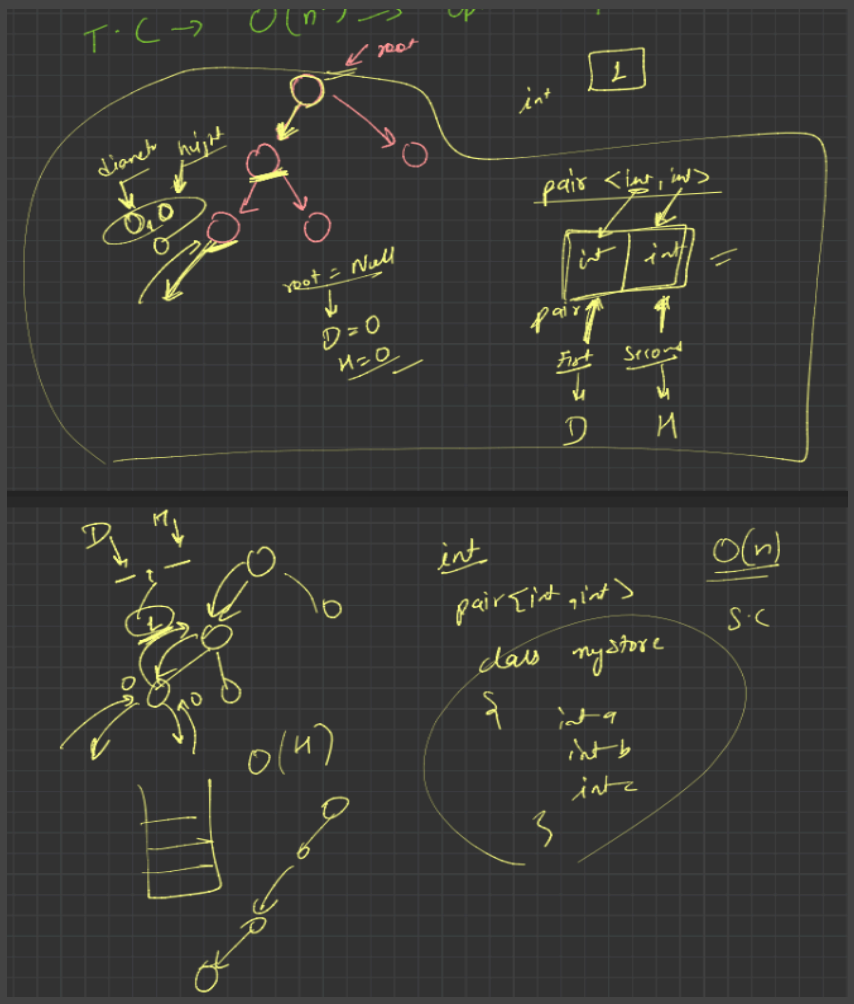




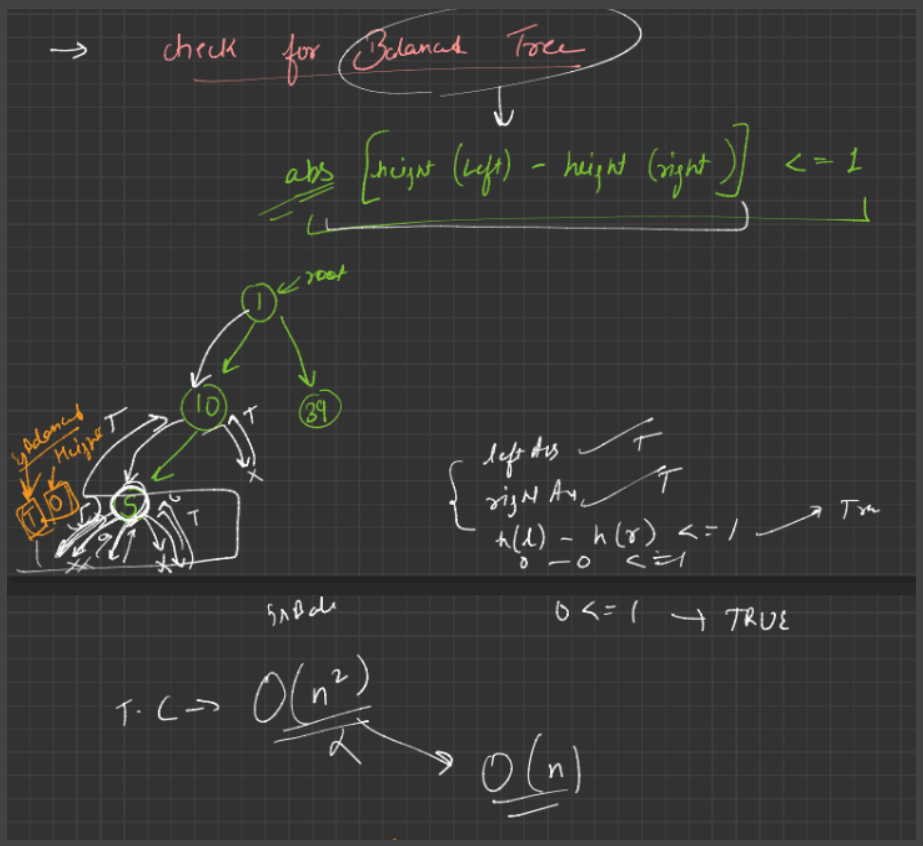
Diameter Of Binary Tree Approach-I:



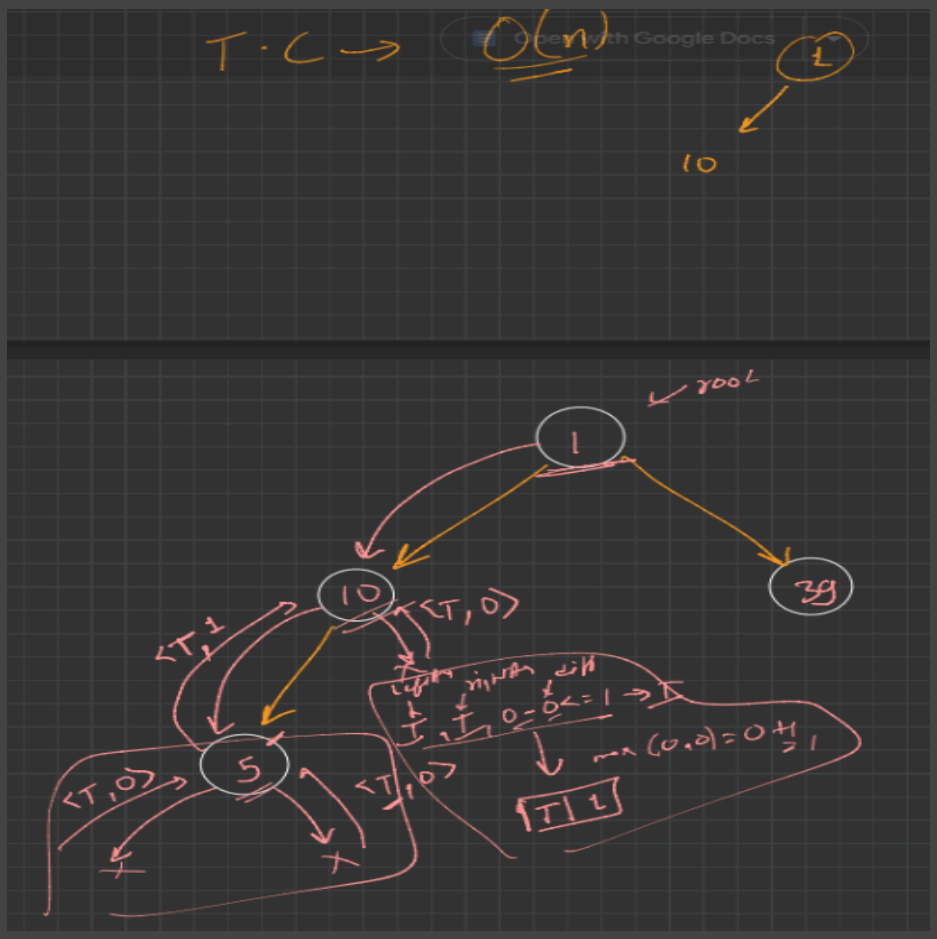
Diameter Of Binary Tree Approach-II:



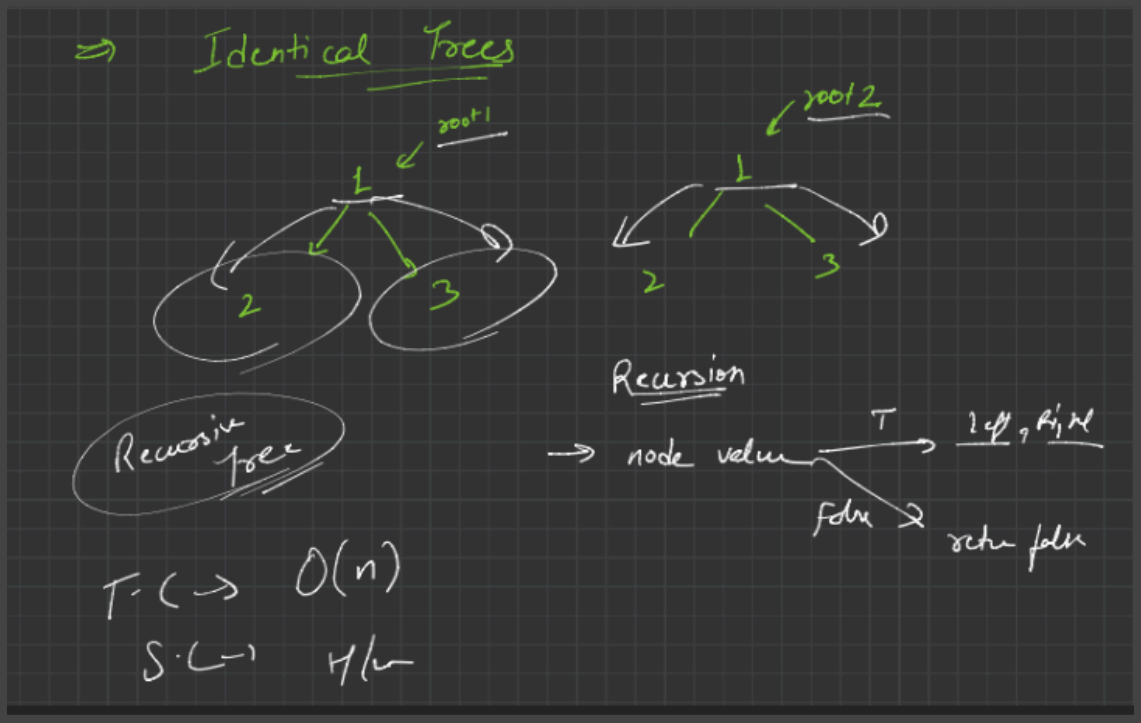
Check If The Tree Is Balanced Or Not Approach-I:



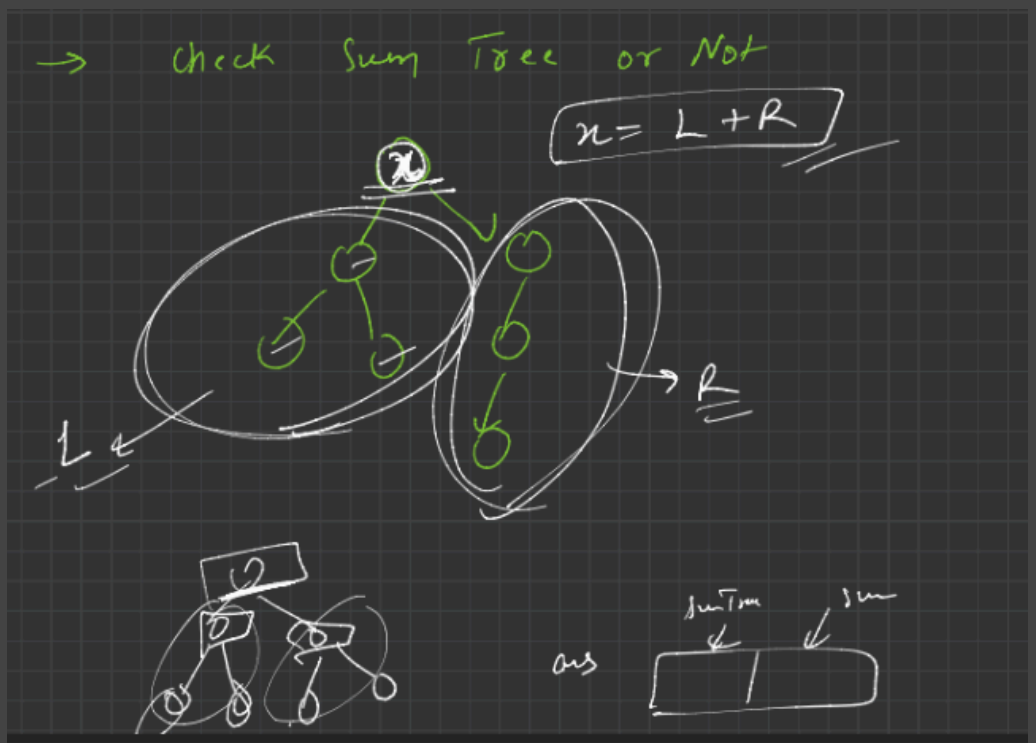
Check If The Tree Is Balanced Or Not Approach-II:

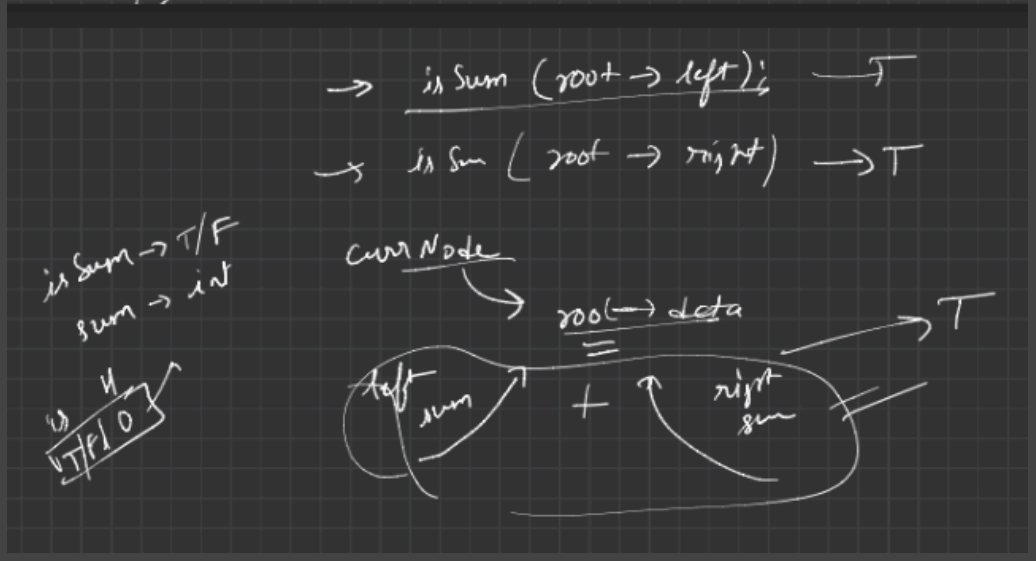


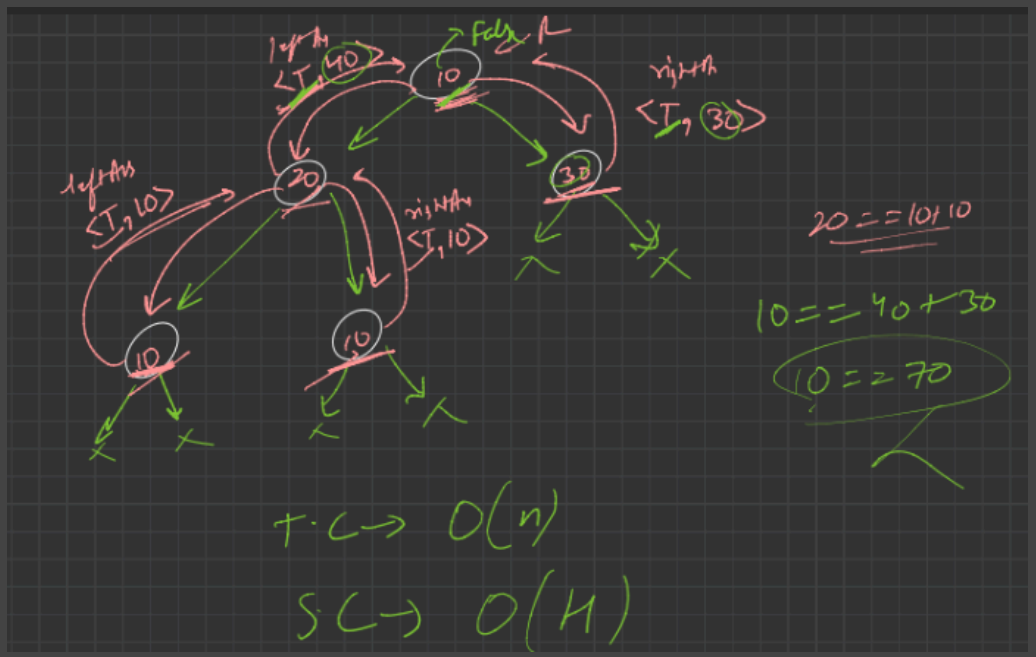
Check Whether The Two Trees Are Identical Or Not:



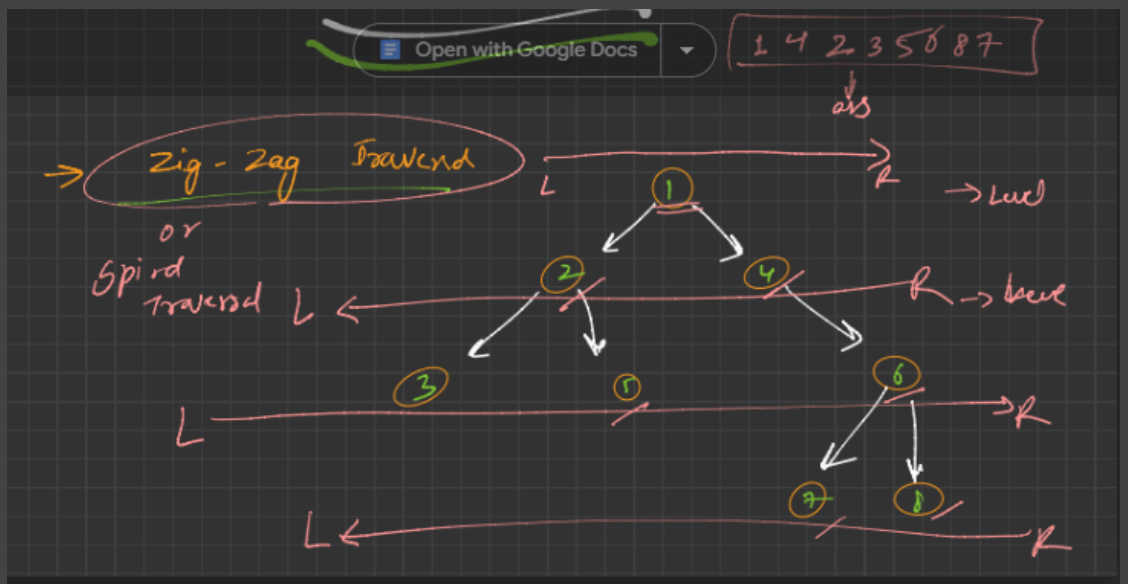
Sum Tree:

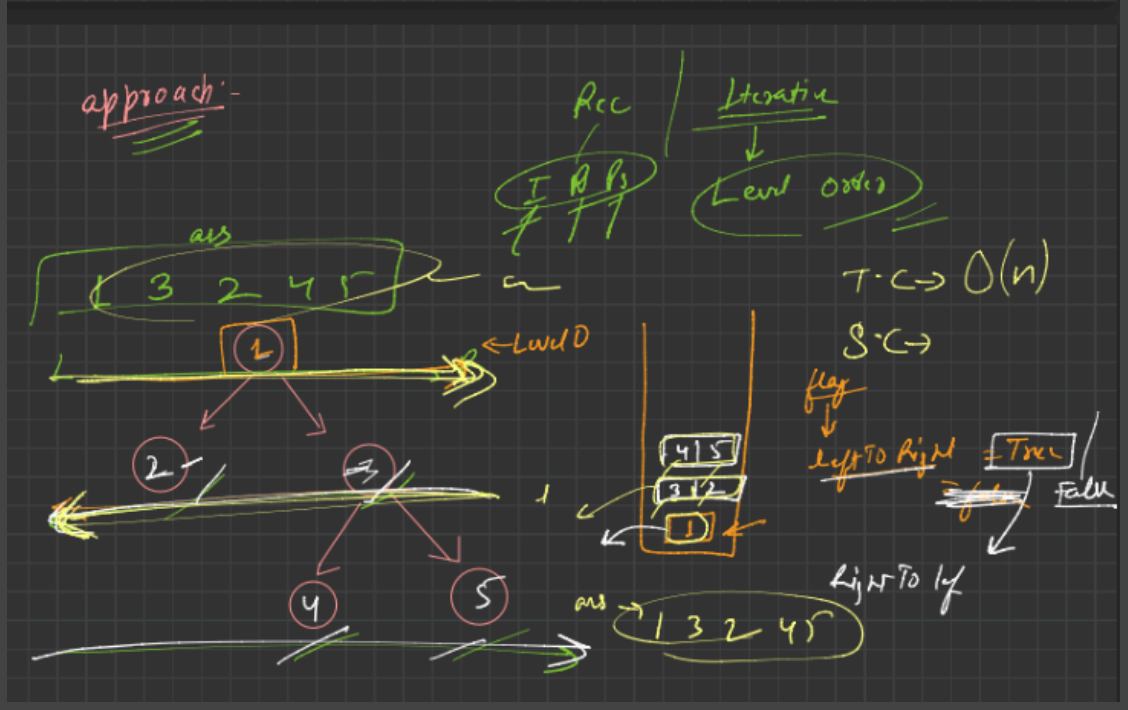

Check Sum Tree or Not
x = L + R
L
R
sumTree
sum
or

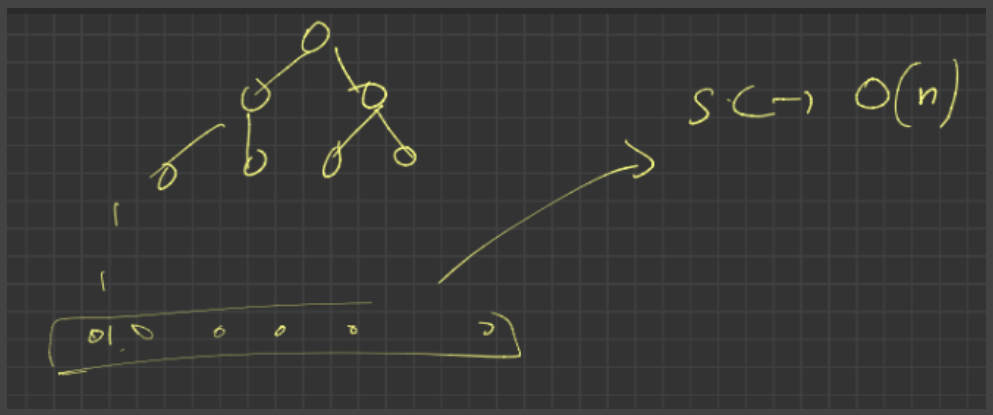

isSum (root → left);
isSum (root → right)
T
isSum → T/F
sum → int
currNode
root → data
left sum
+
right sum


leftNN
rightNN
20 == 10 + 10
10 == 40 + 30
10 == 70
T.C → O(n)
S.C → O(H)

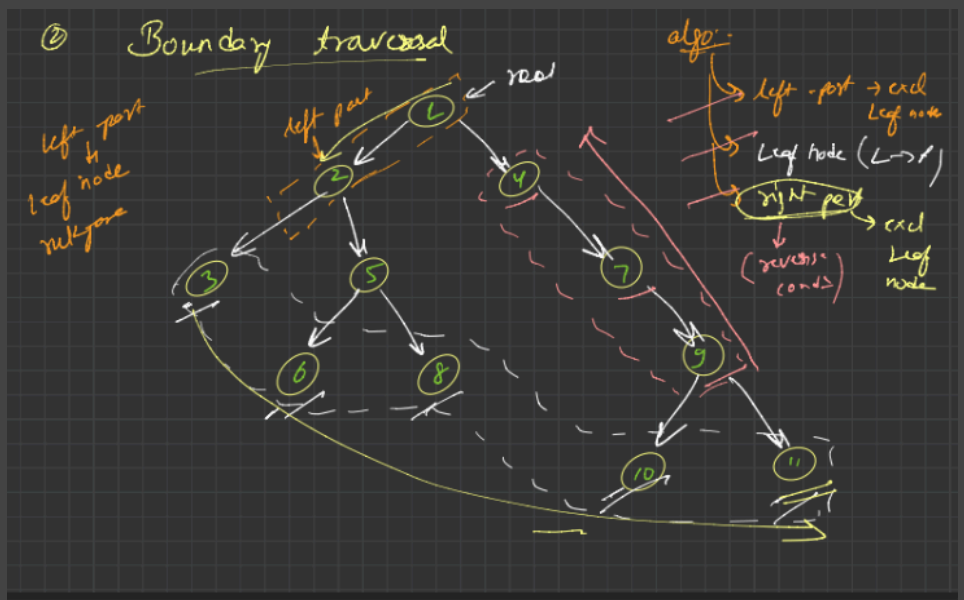
Zigzag Traversal:

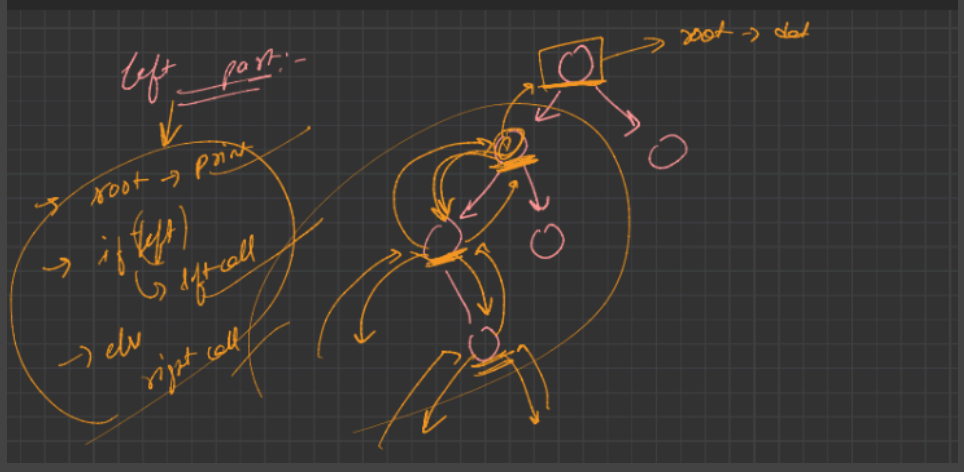


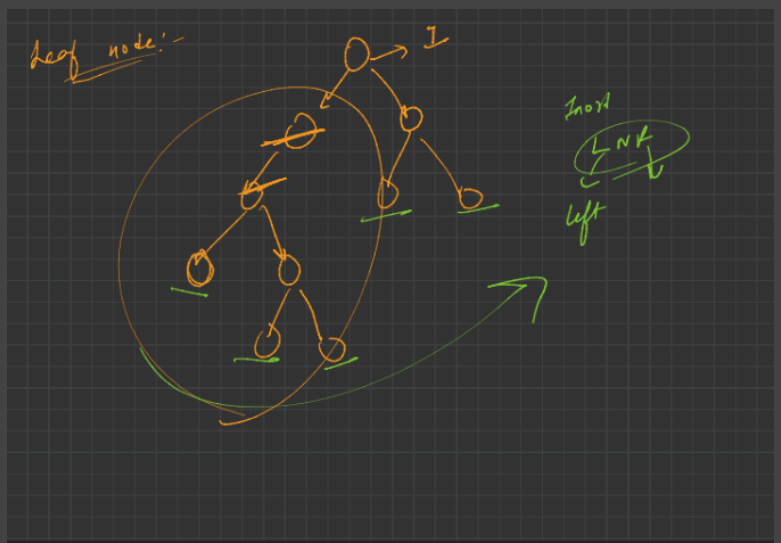


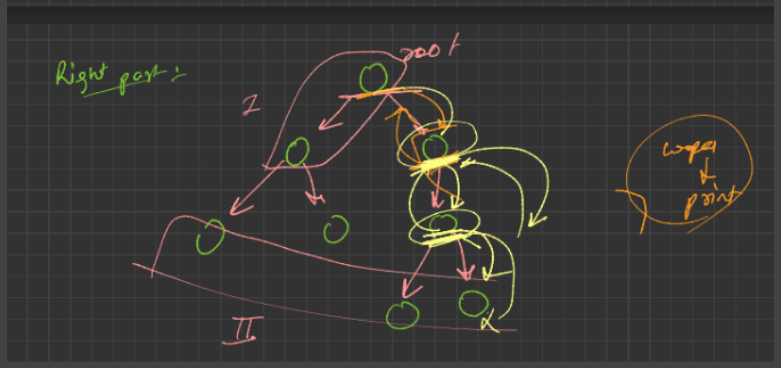


Boundary Traversal:

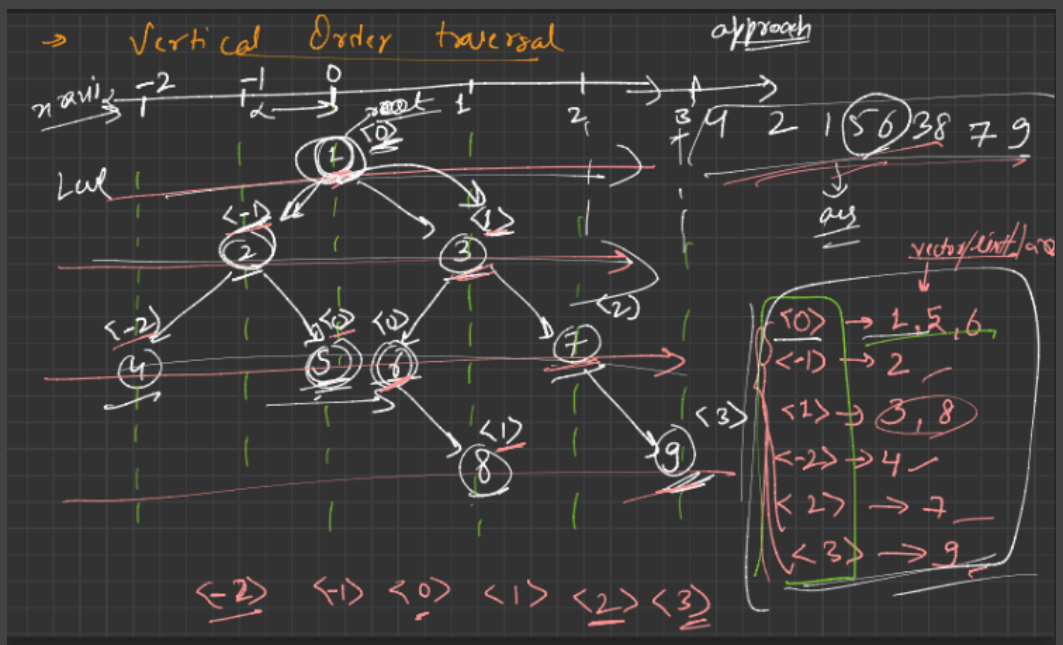


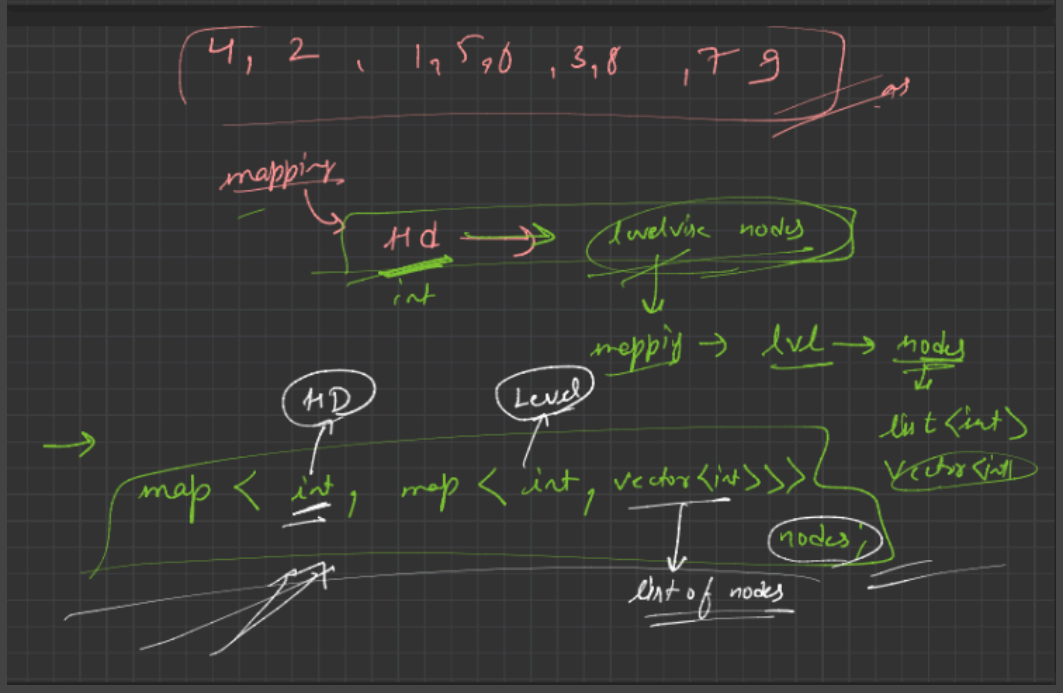


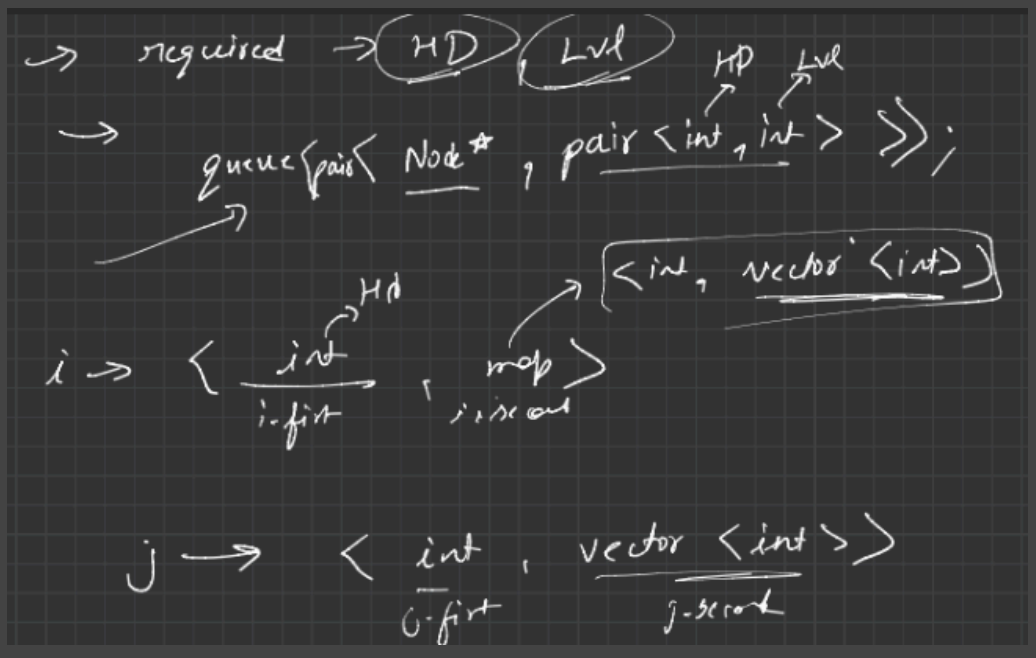




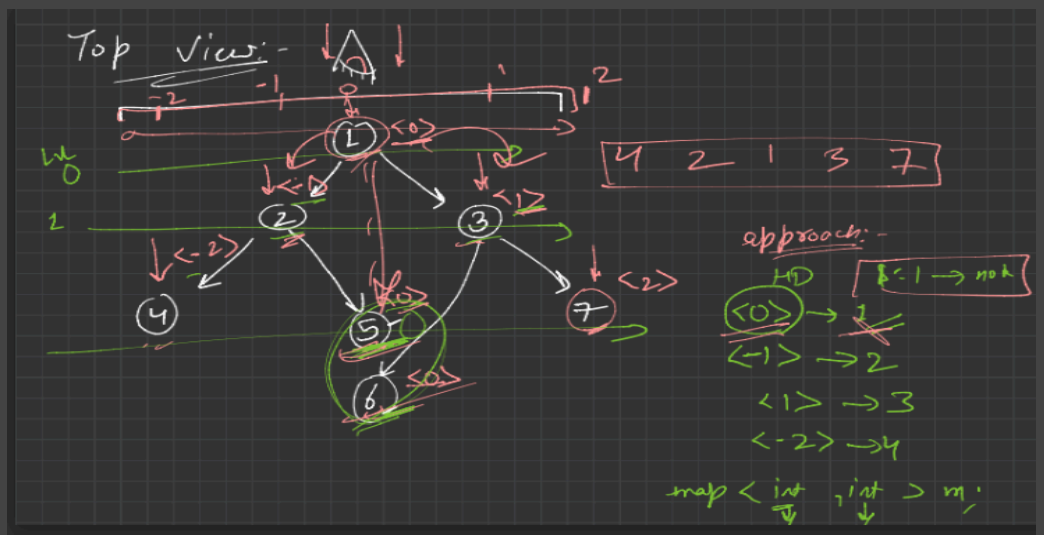
Vertical Traversal:

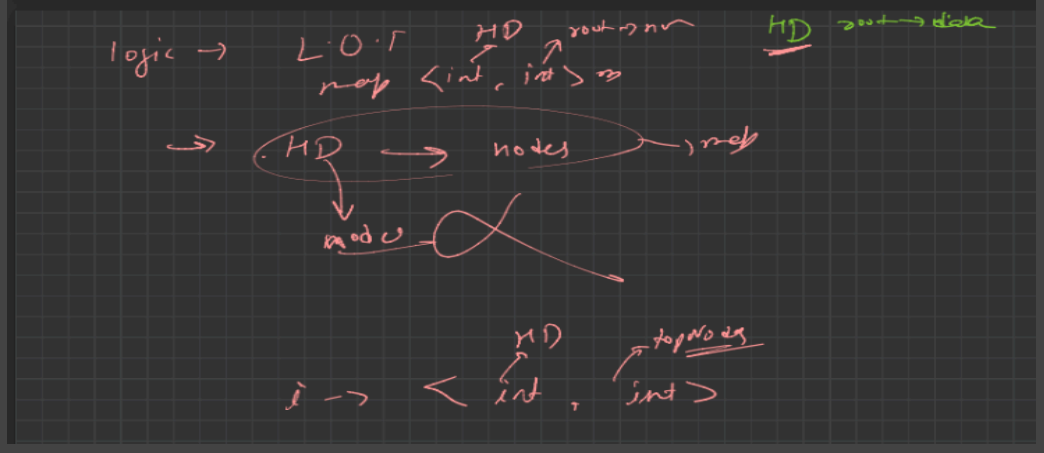




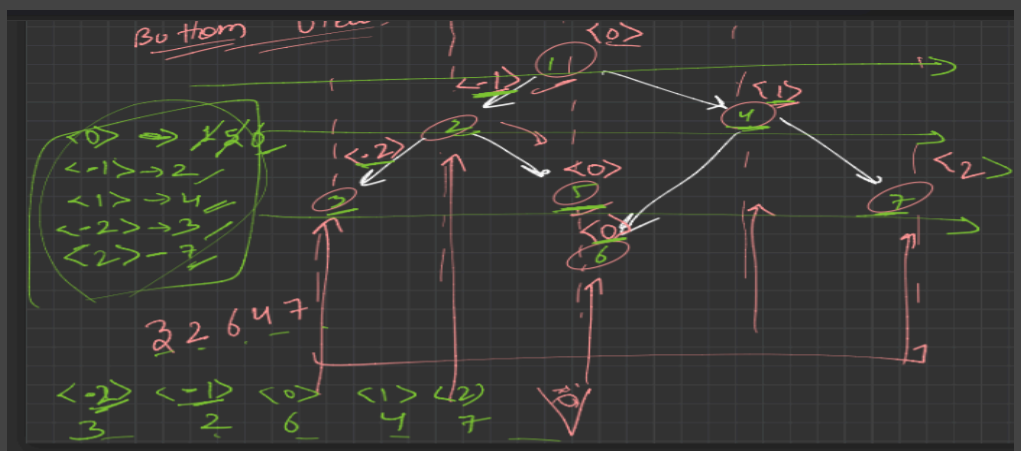


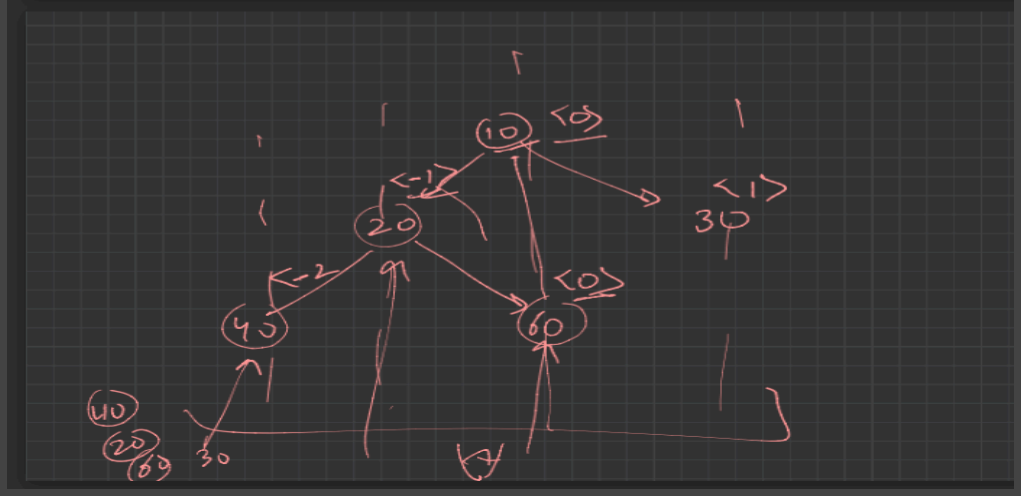
Top View:



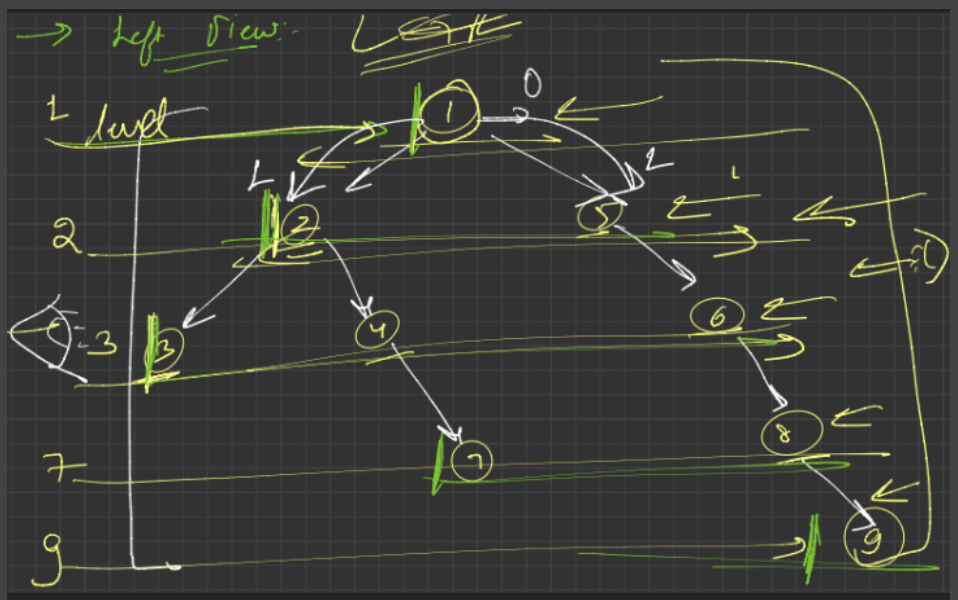


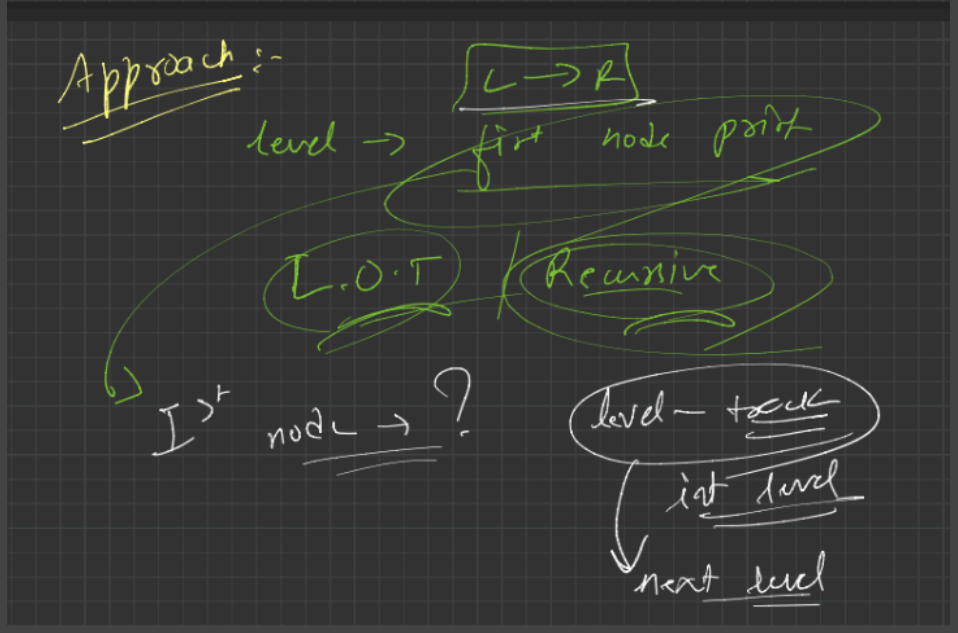
Bottom View:

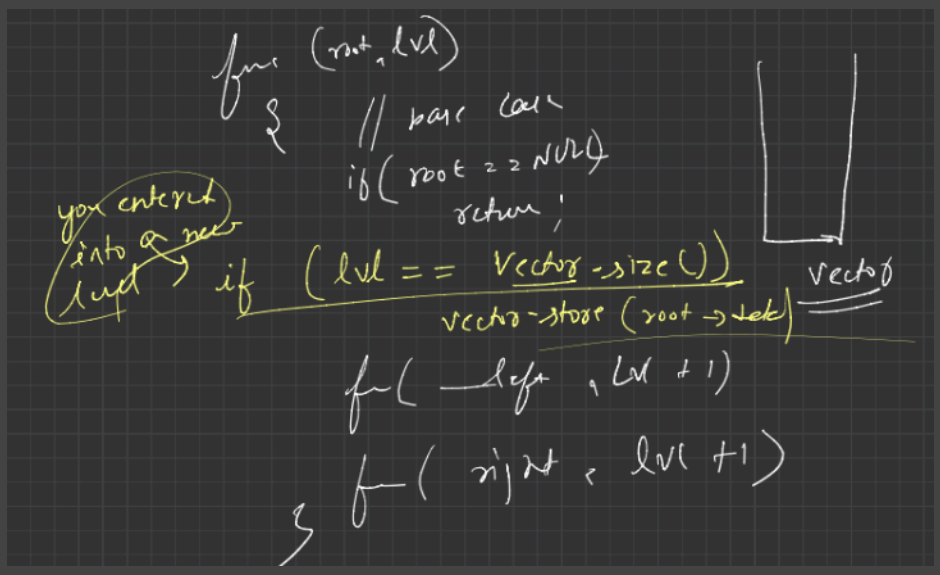




Left View:





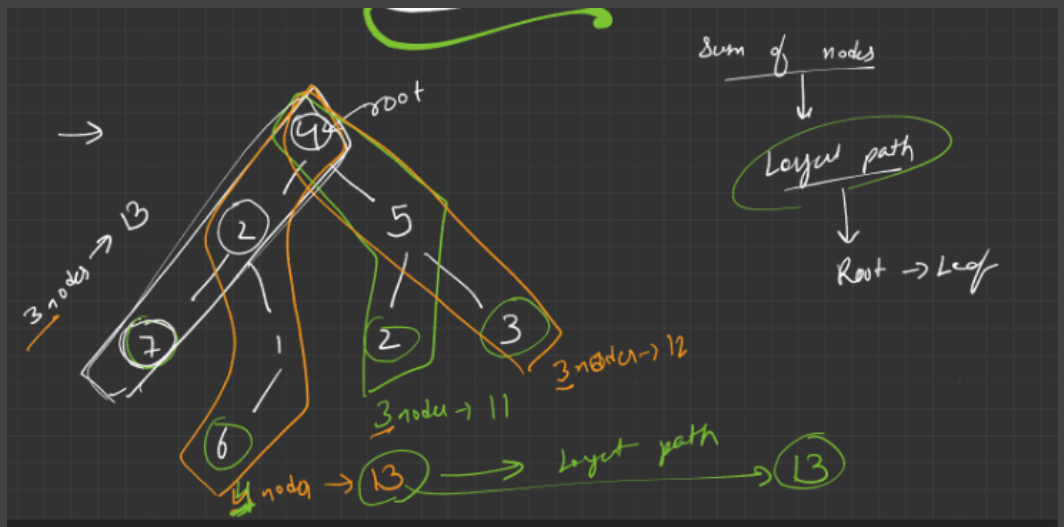


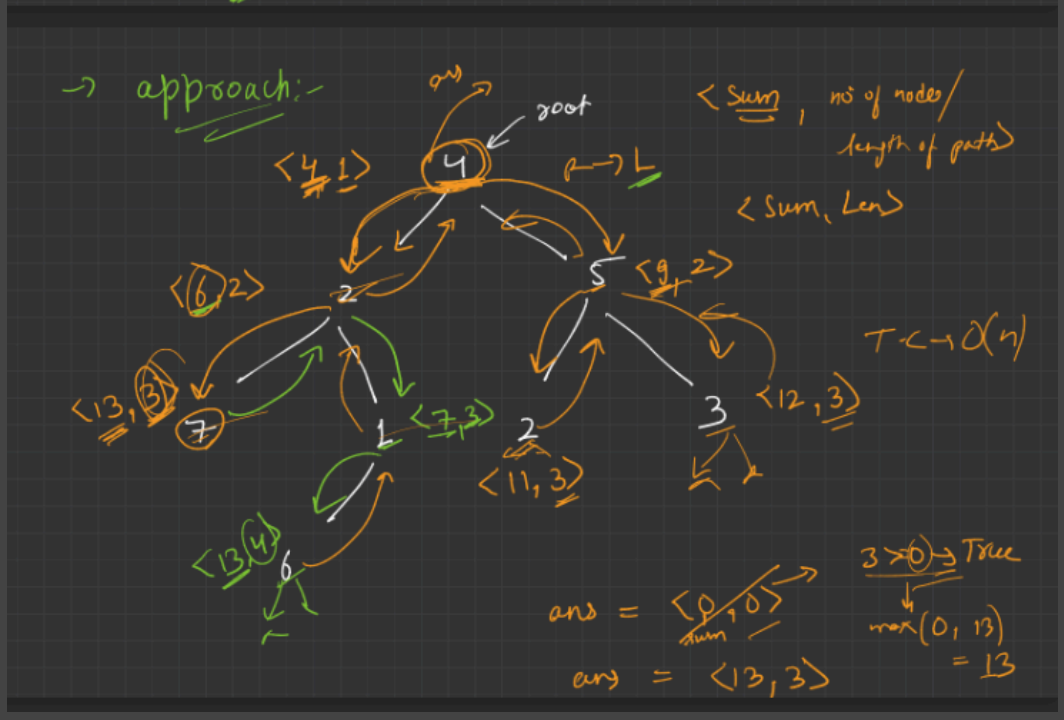
Right View:

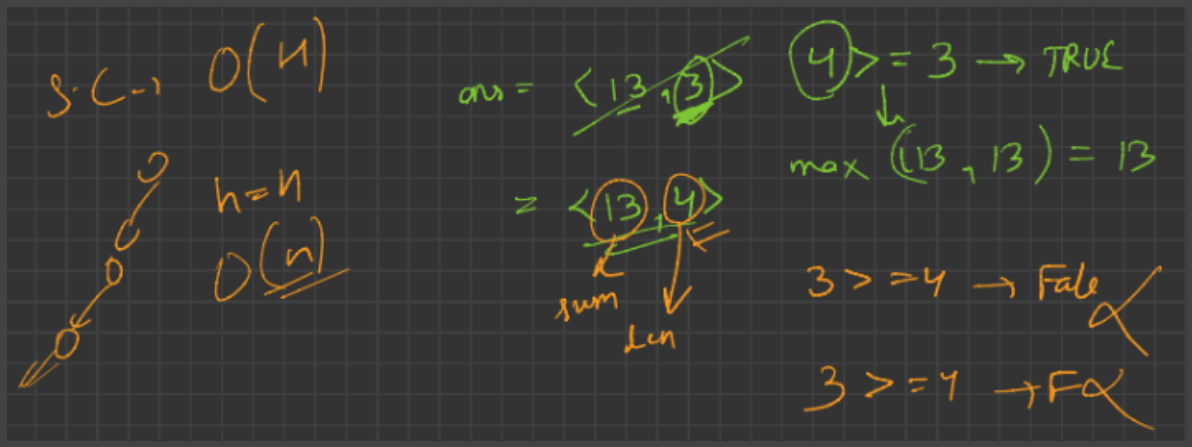
Just make the recursive call for right first and then left.

Diagonal Traversal:

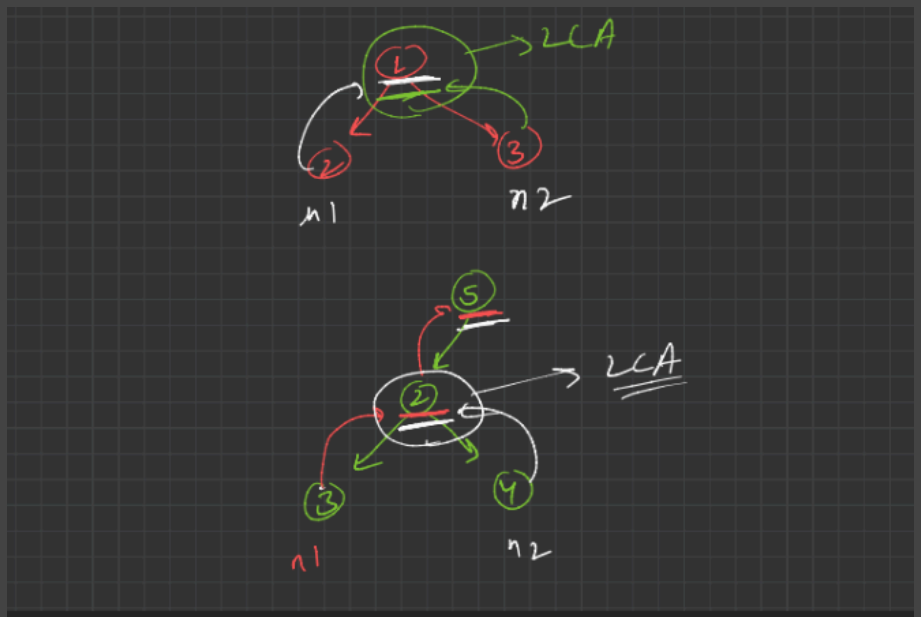
Sum Of The Largest Bloodline Of A Tree(Sum Of Nodes On The Longest Path From Roof To Leaf Node):

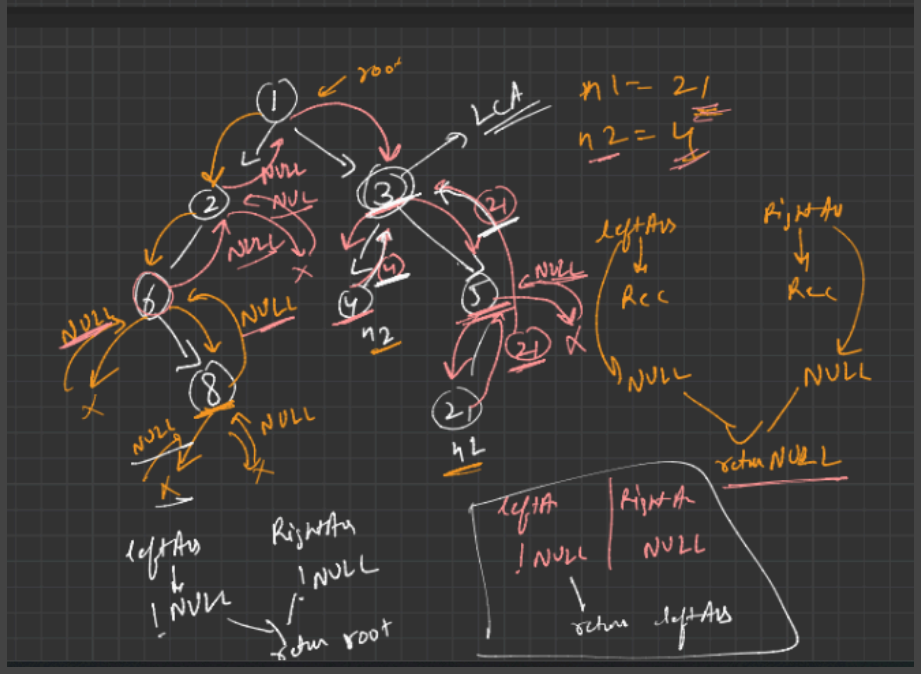


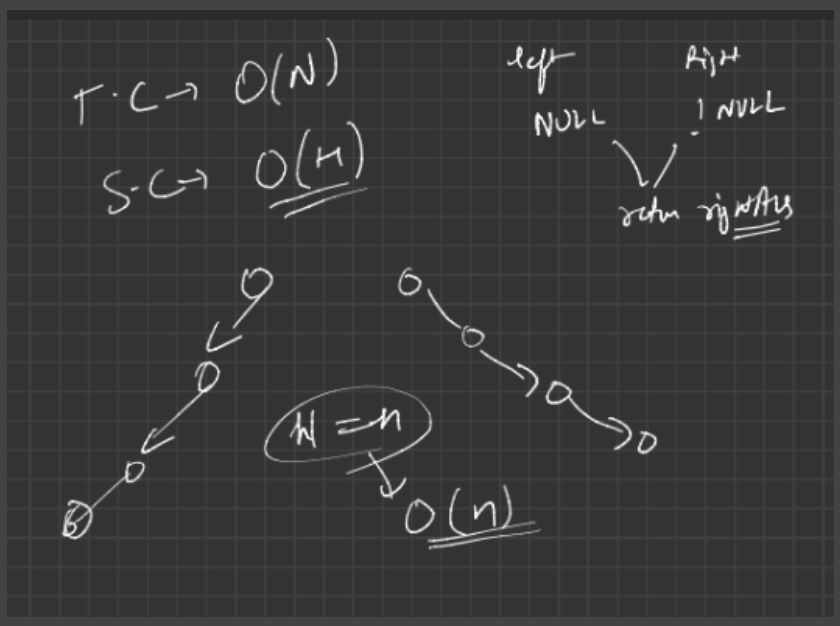




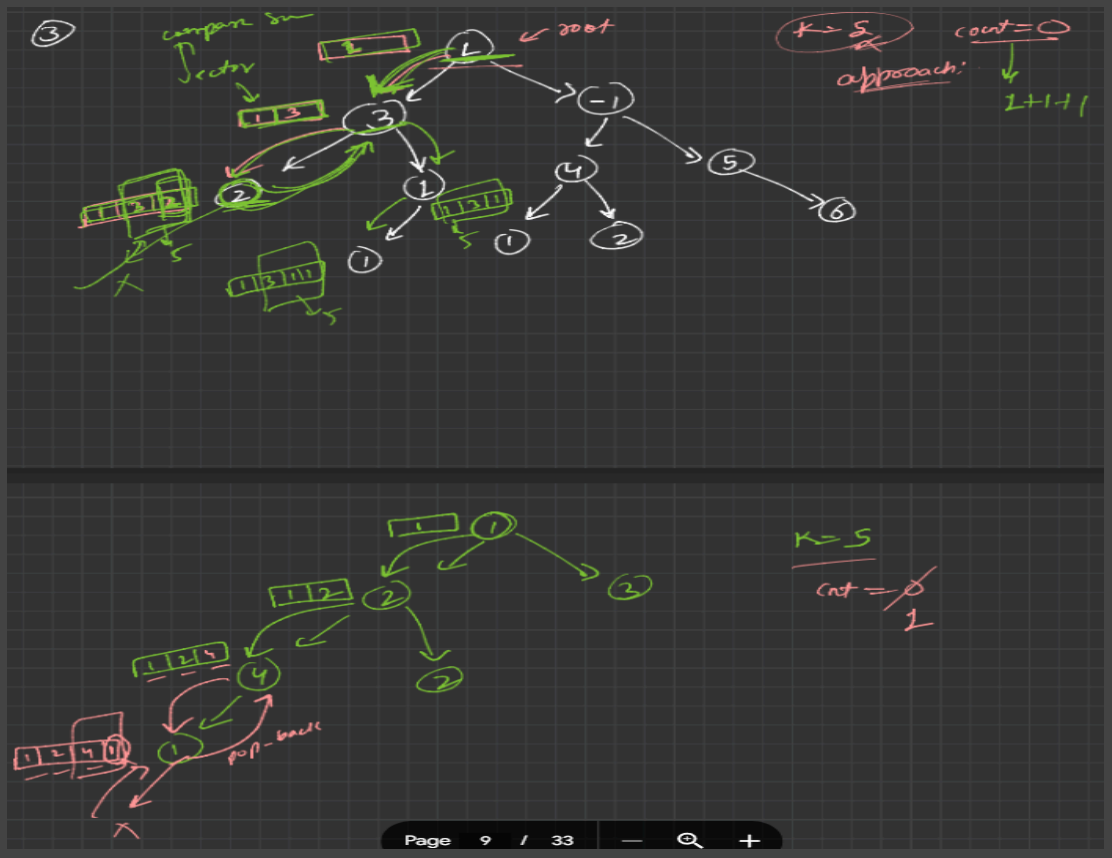
Lowest Common Ancestor:



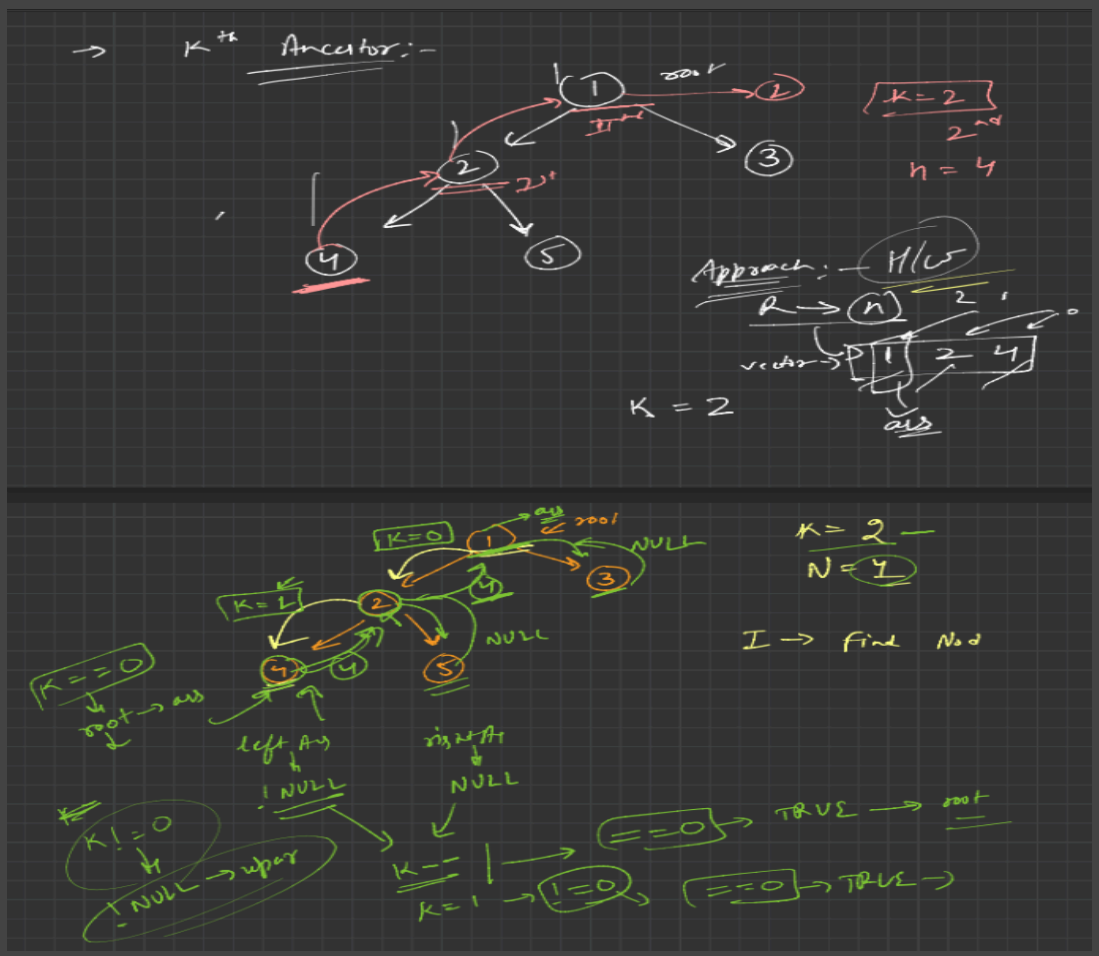


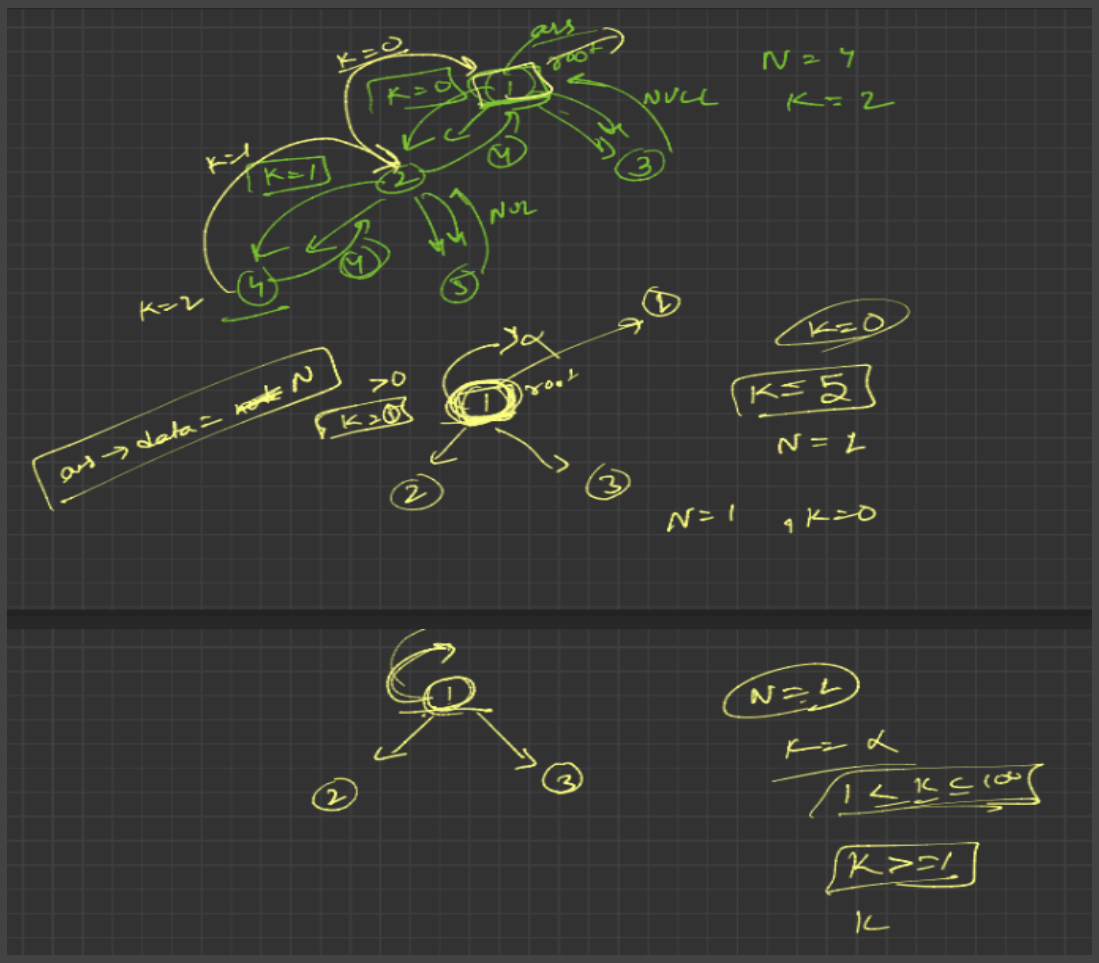


K Sum Paths:

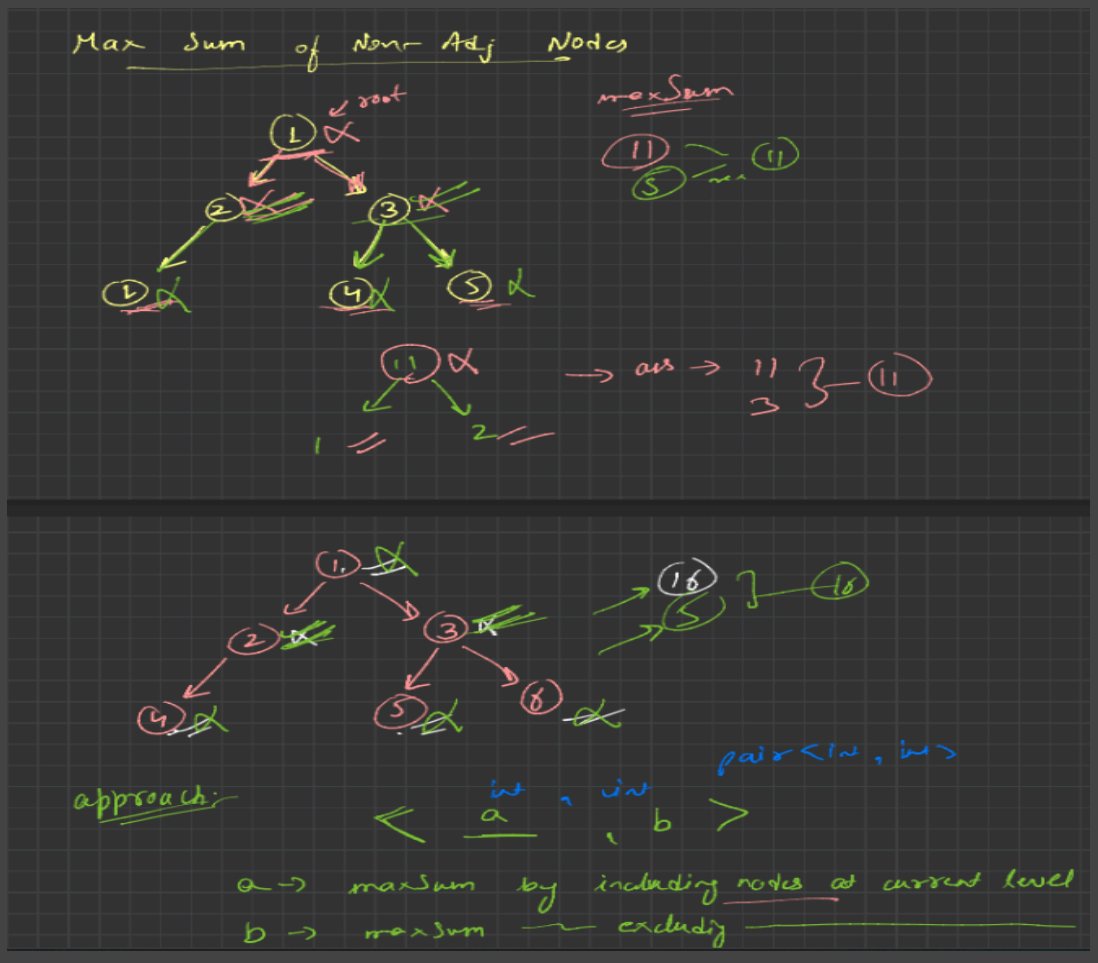


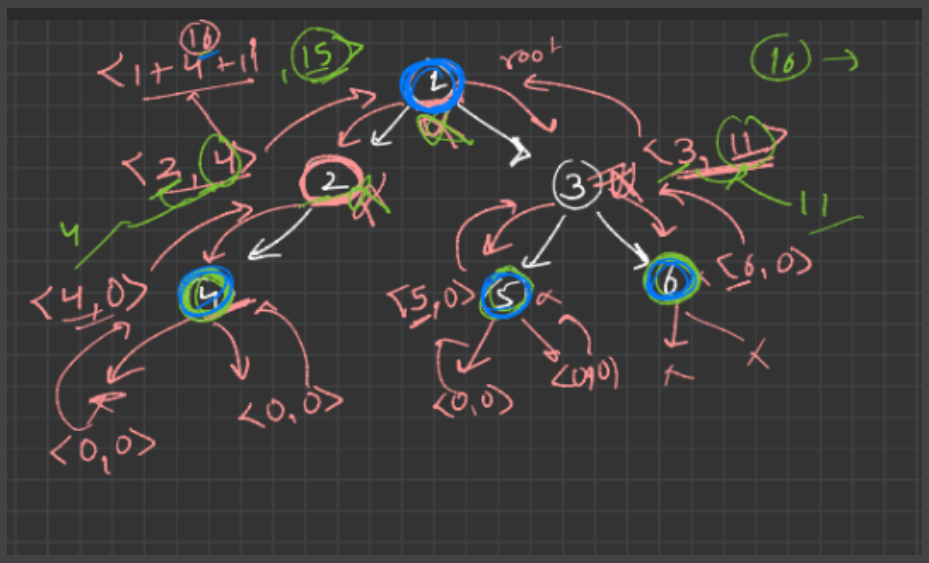
Kth Ancestor In A Tree:



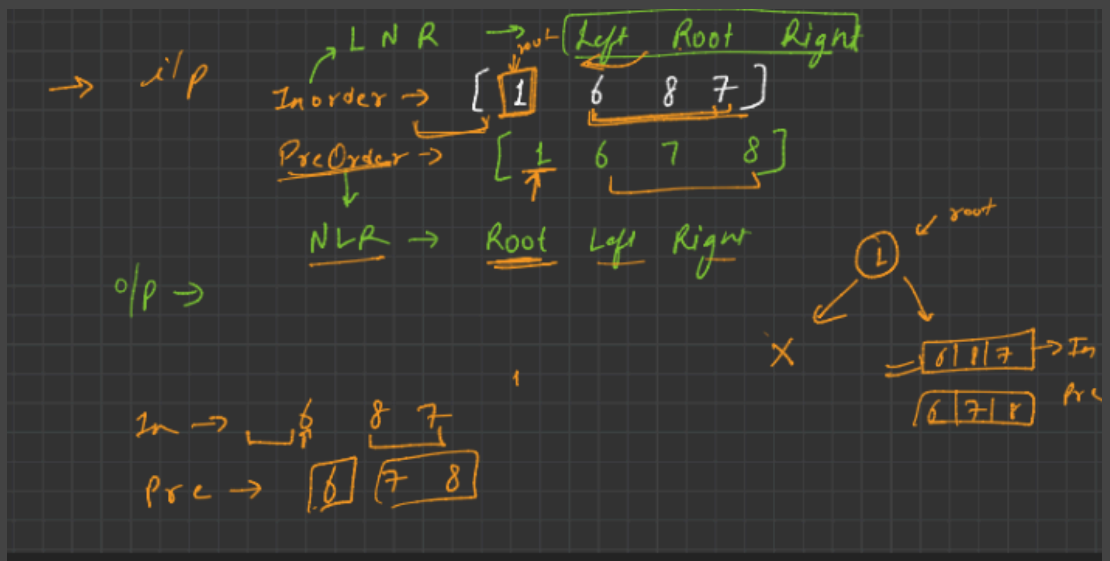


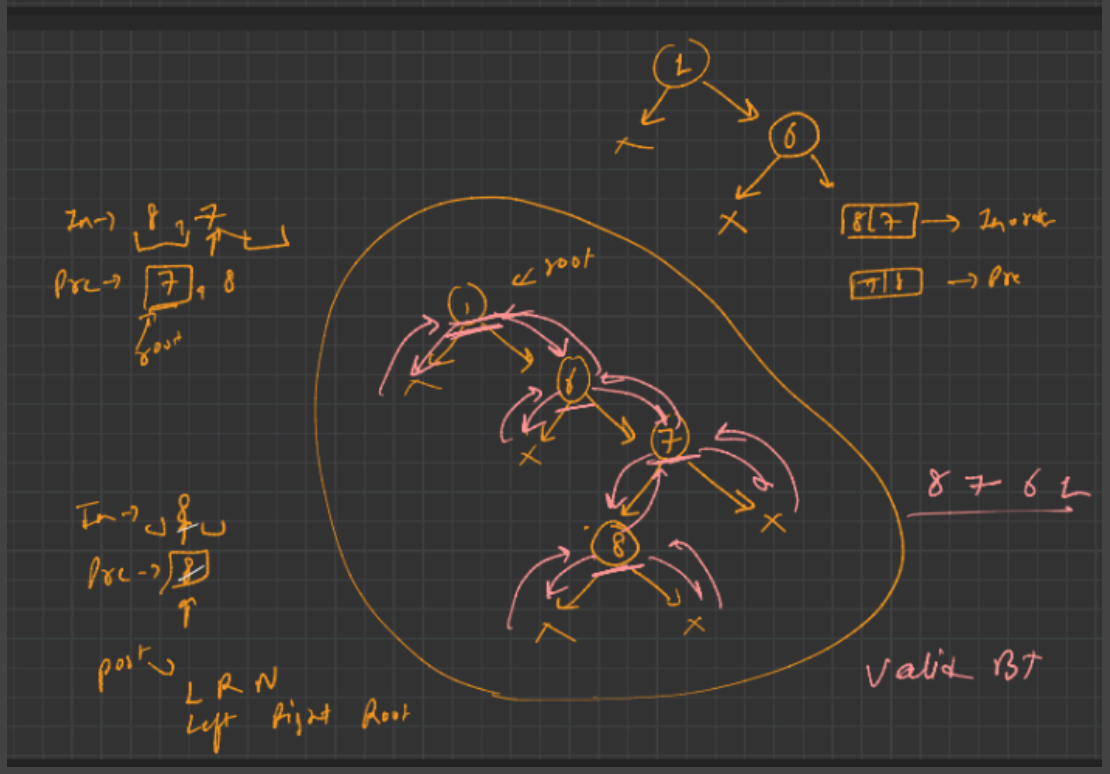
Maximum Sum Of Non-Adjacent Nodes:

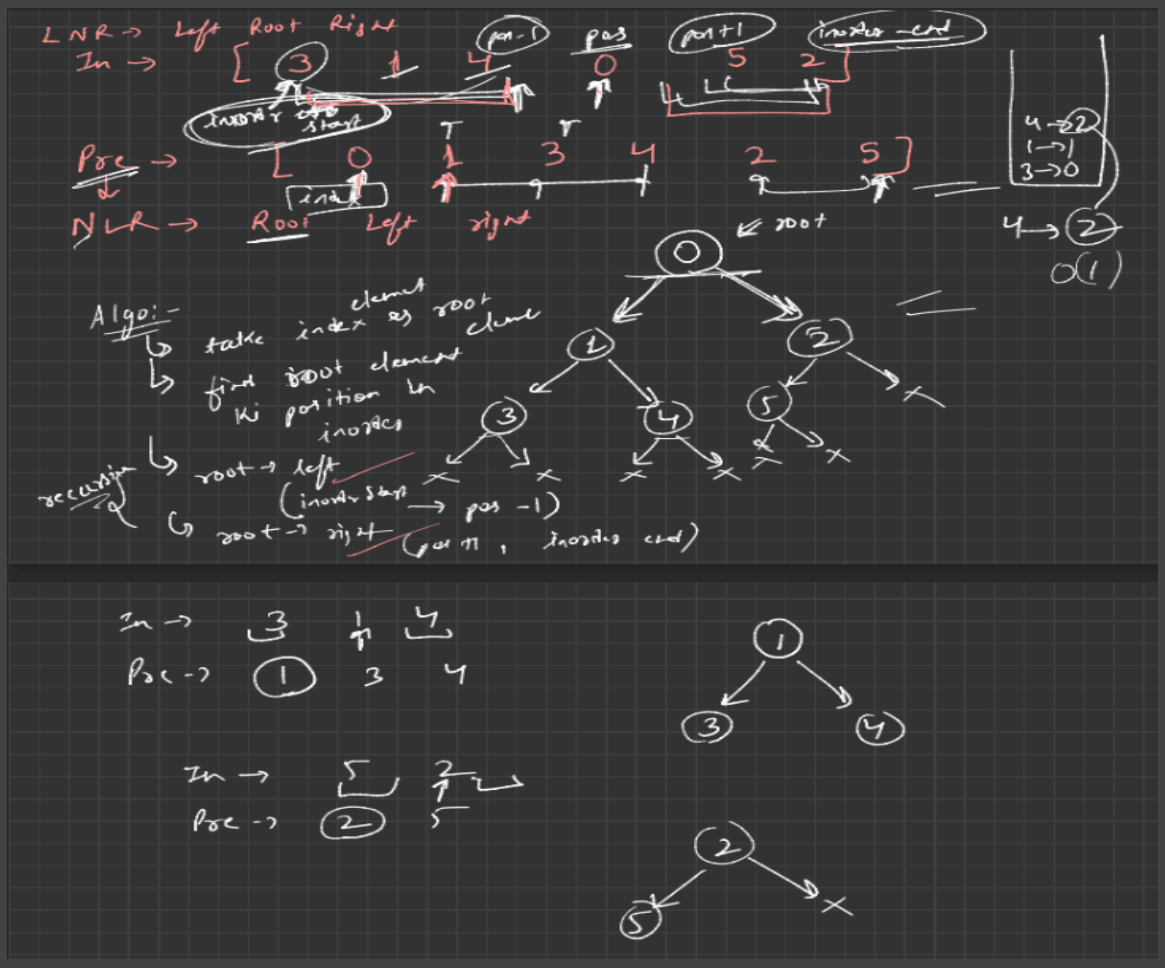




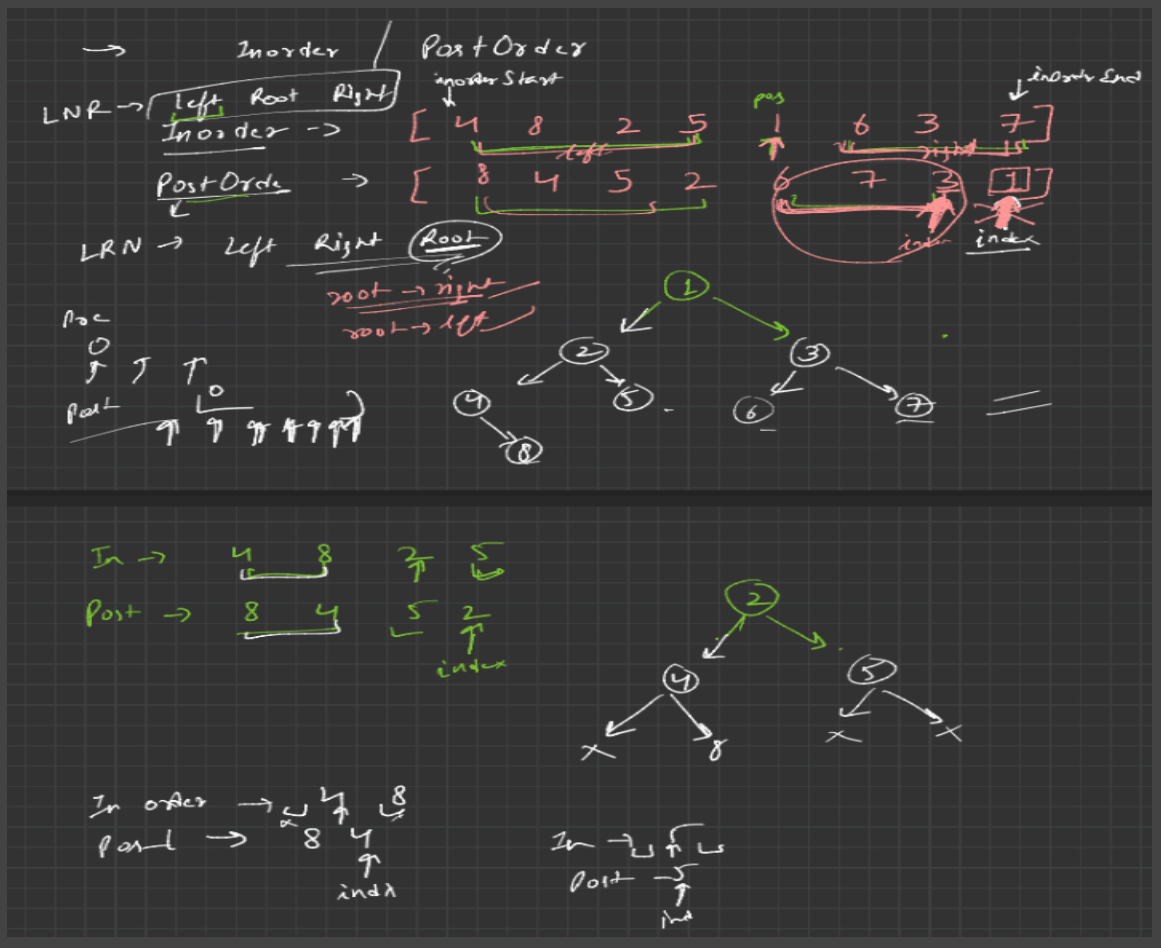
Construct Tree From InOrder And PreOrder:

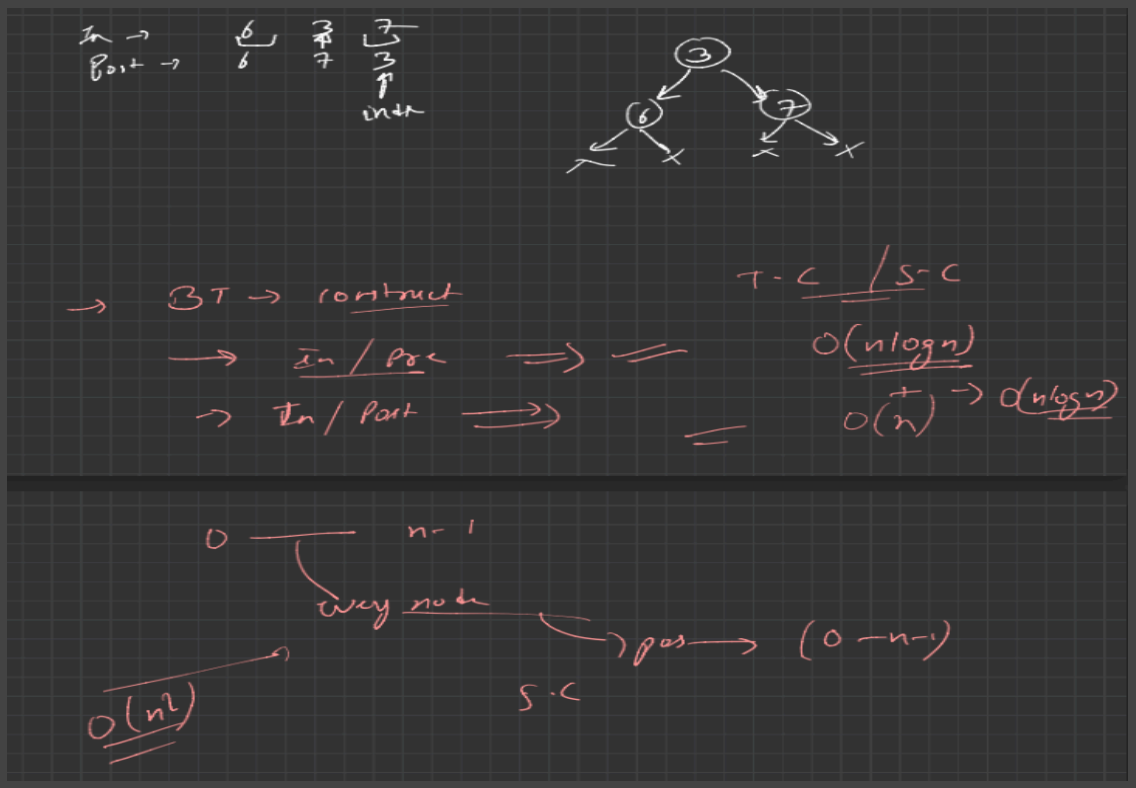




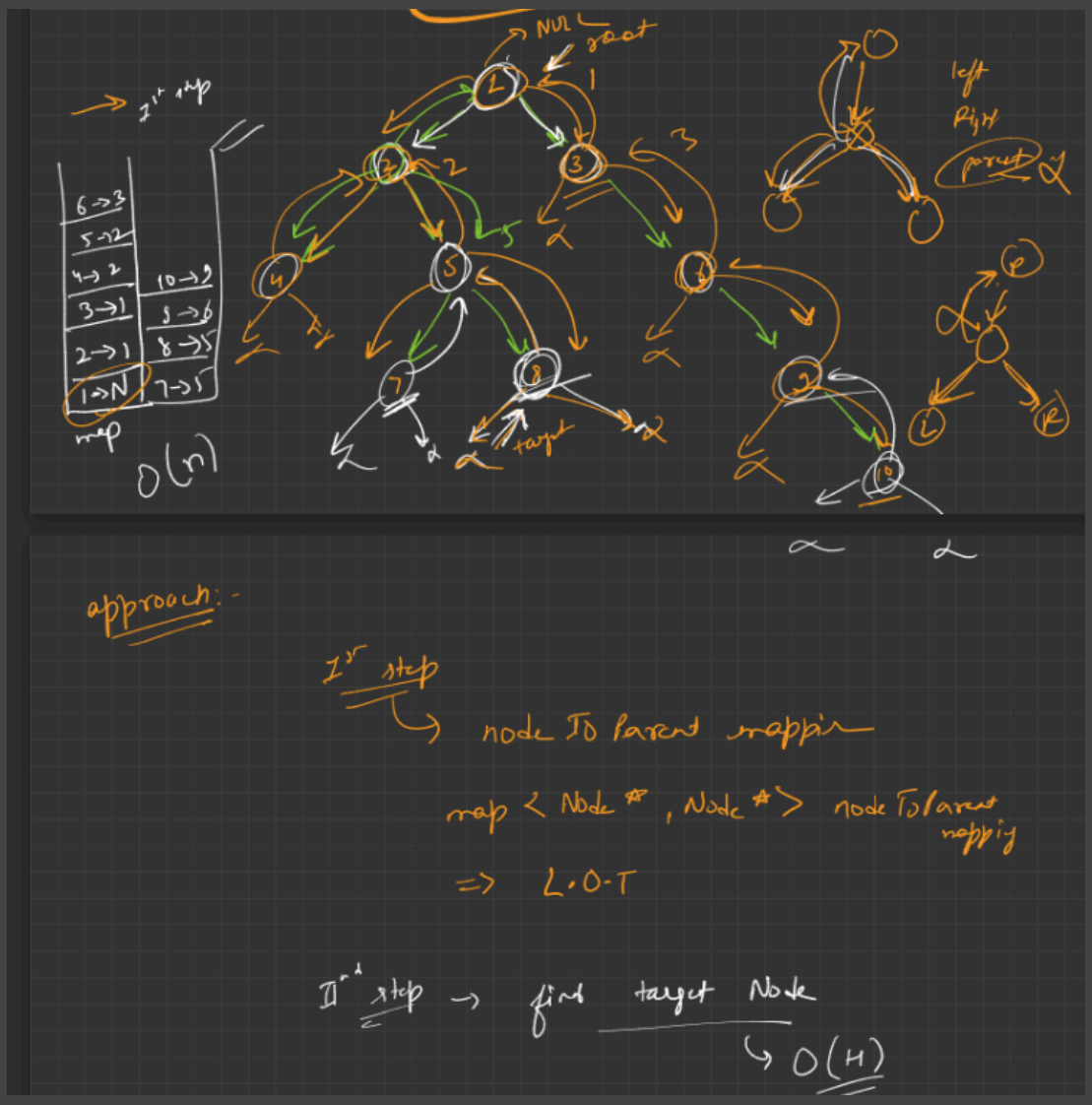


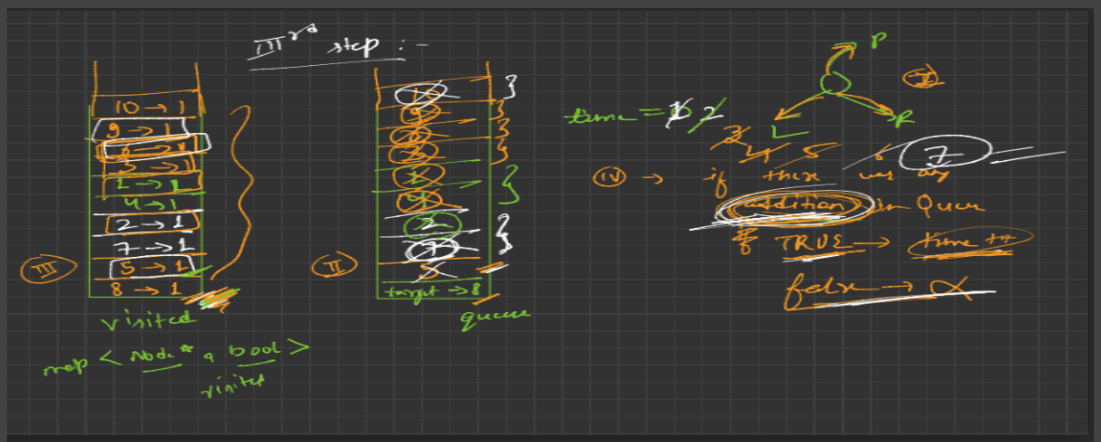
Construct Tree From InOrder And PostOrder:

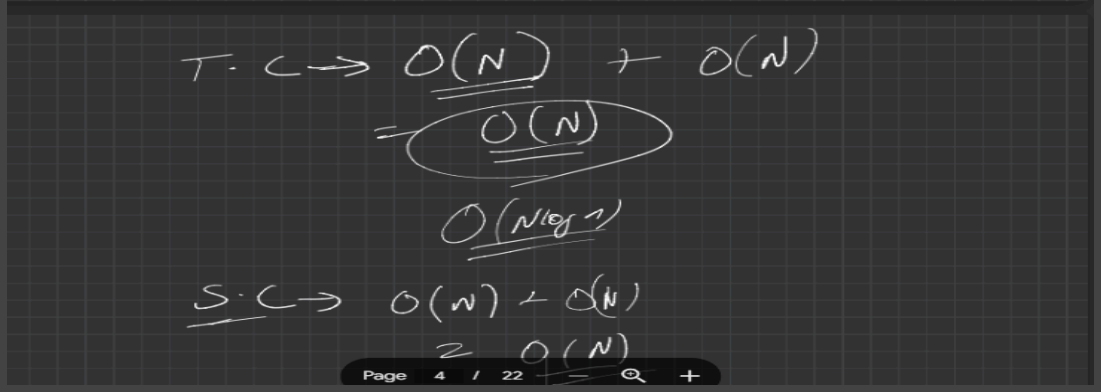




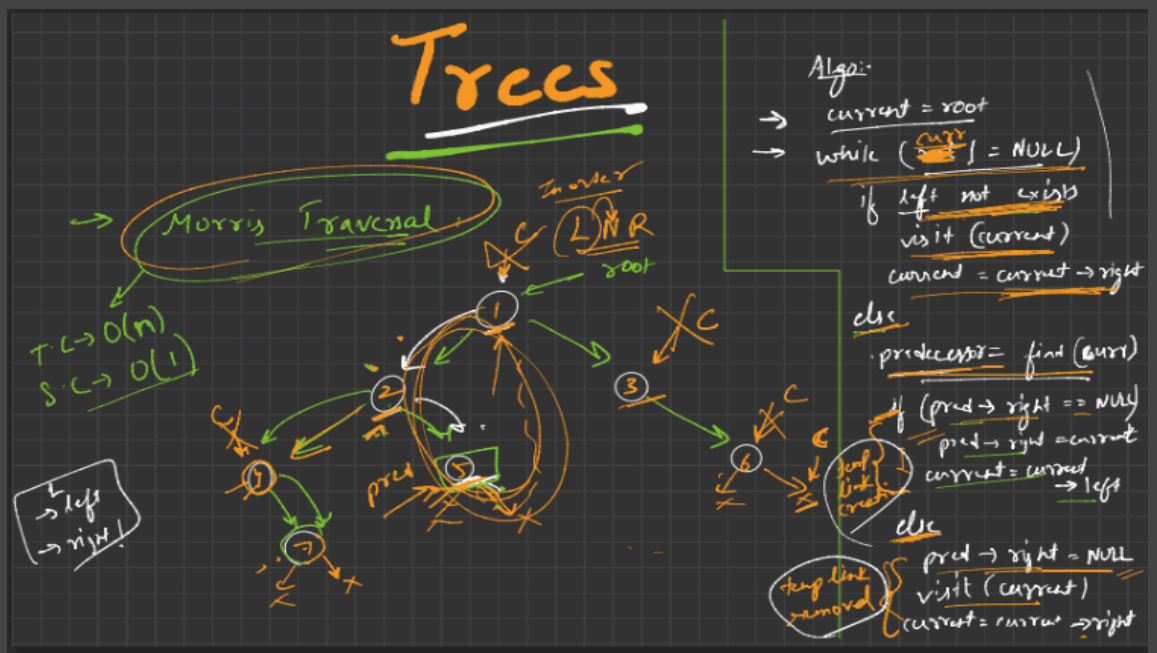
Minimum Tree To Burn The Entire Binary Tree:

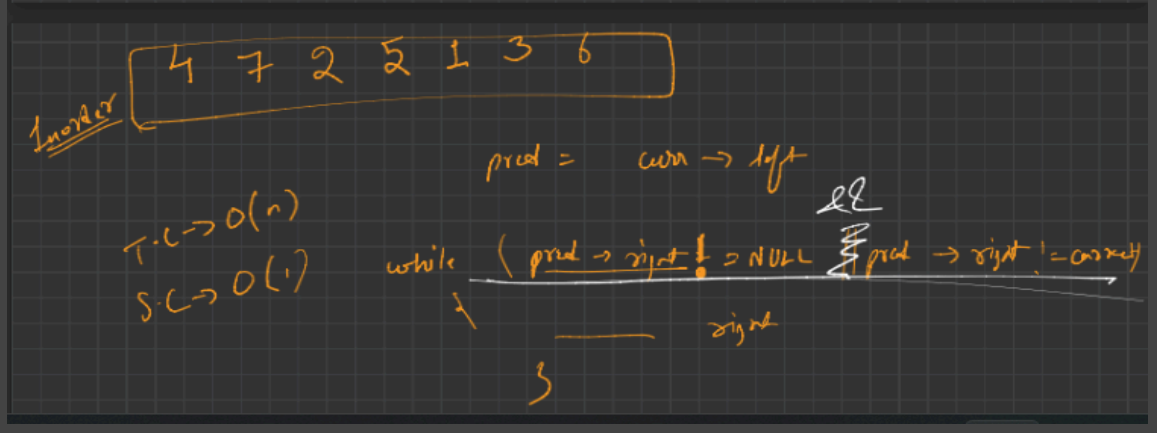




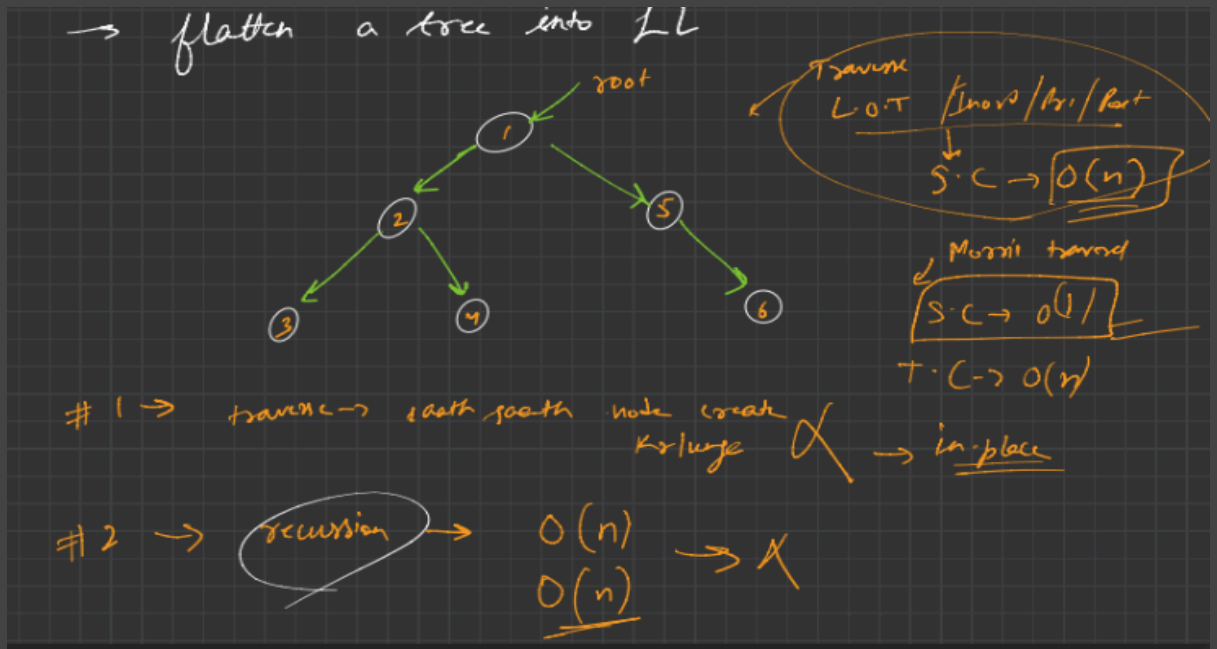


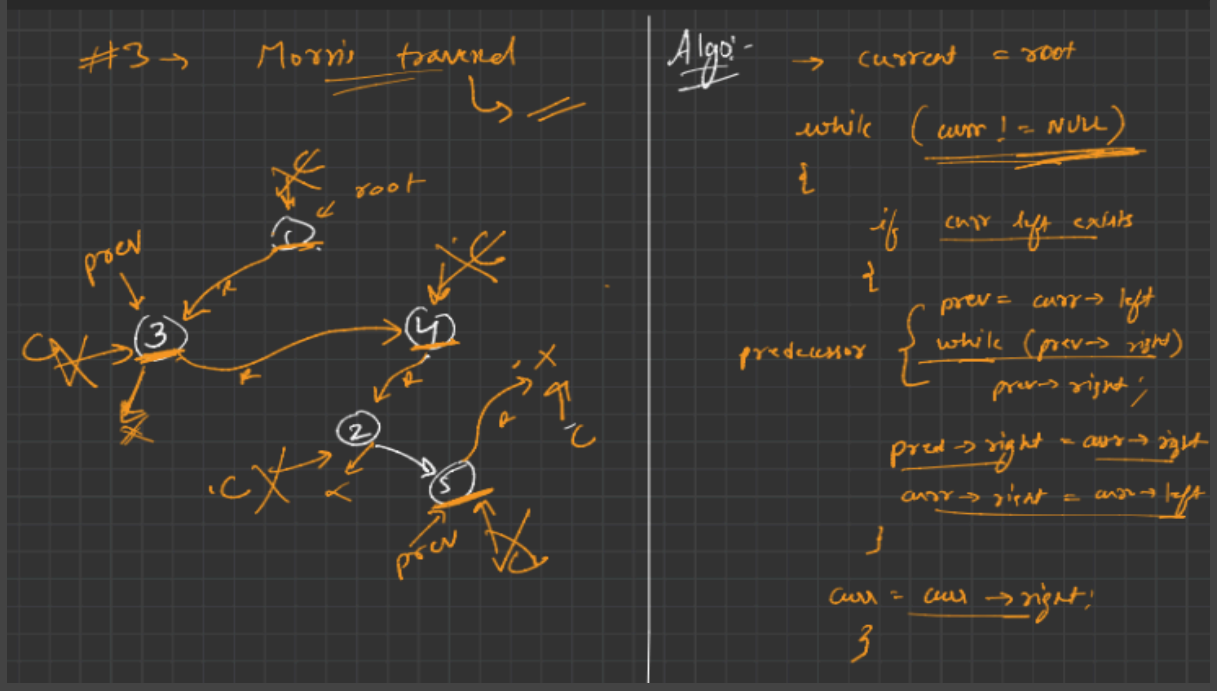
Morris Traversal:

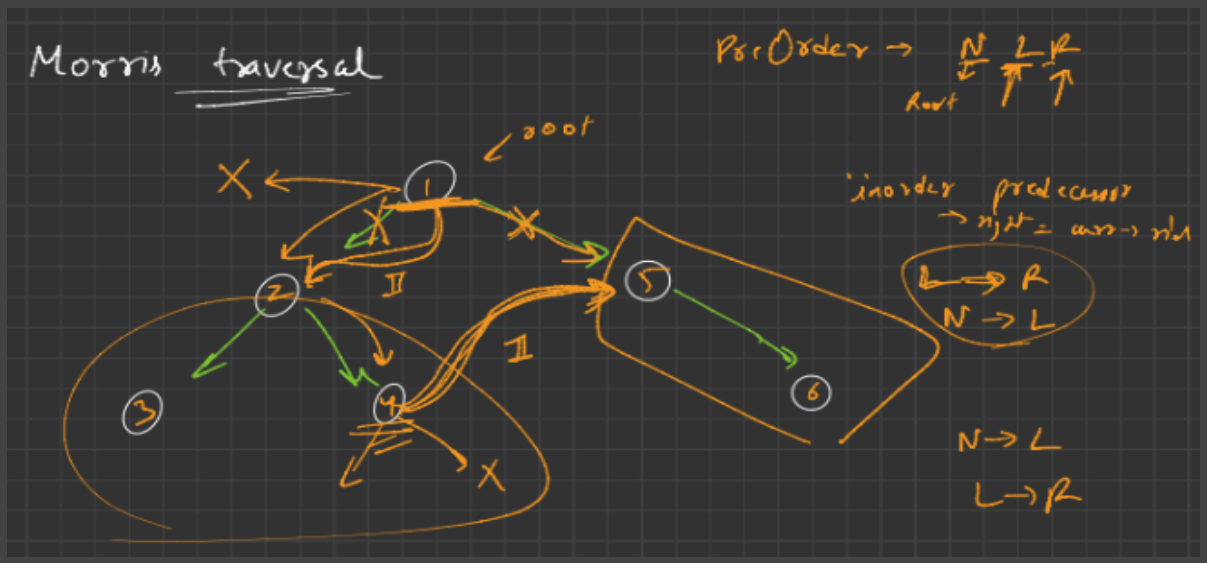




Flatten A Binary Tree To Linked List:

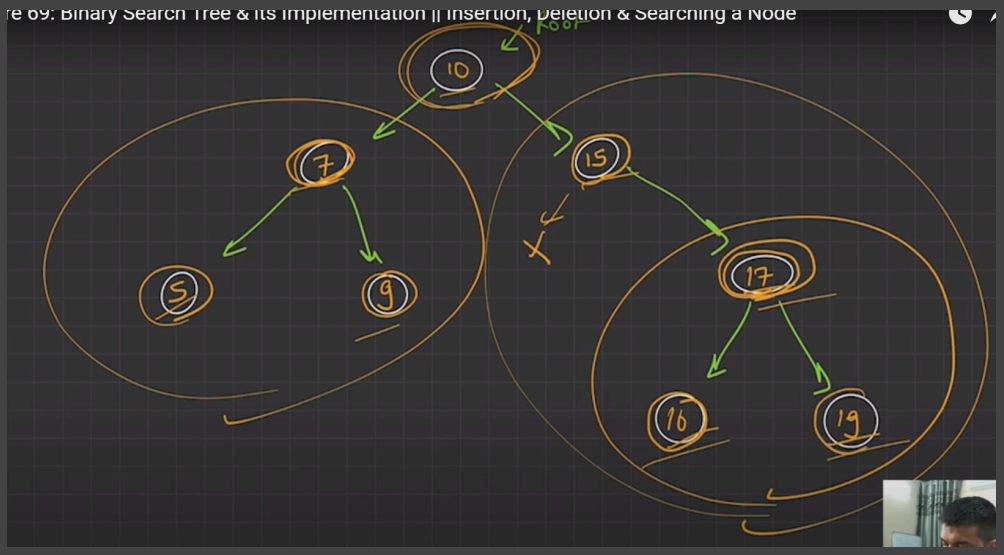


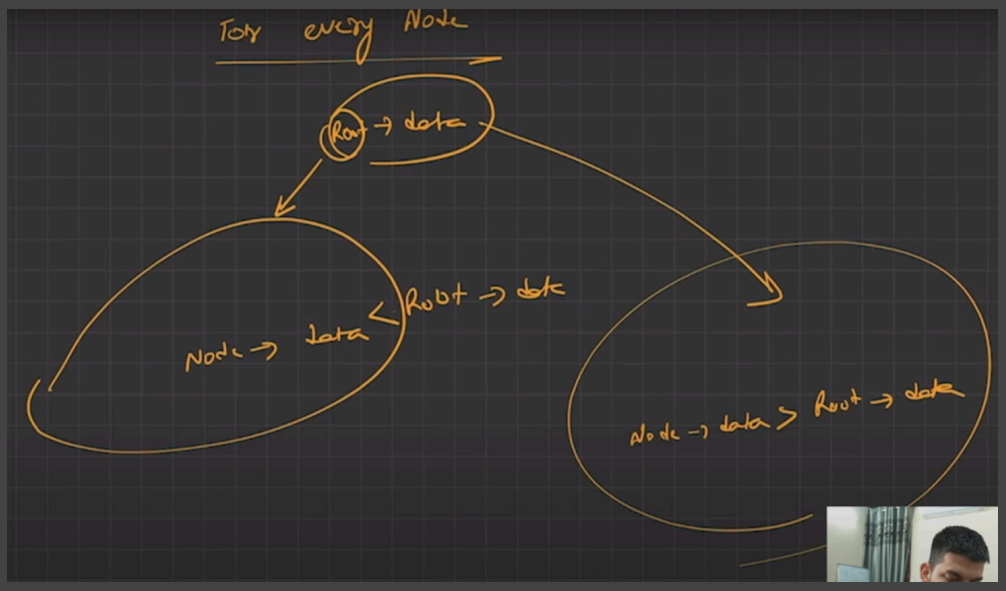


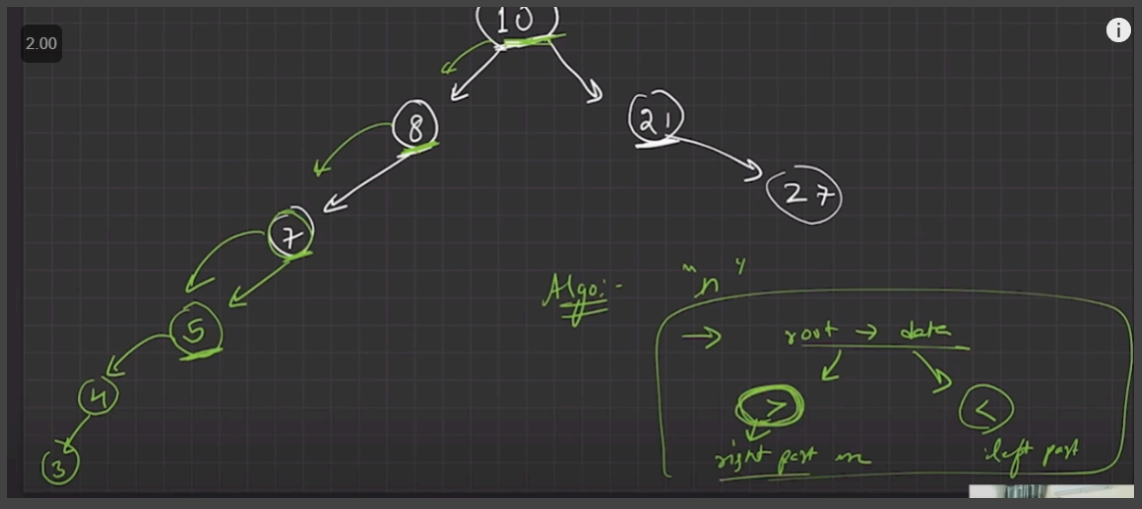


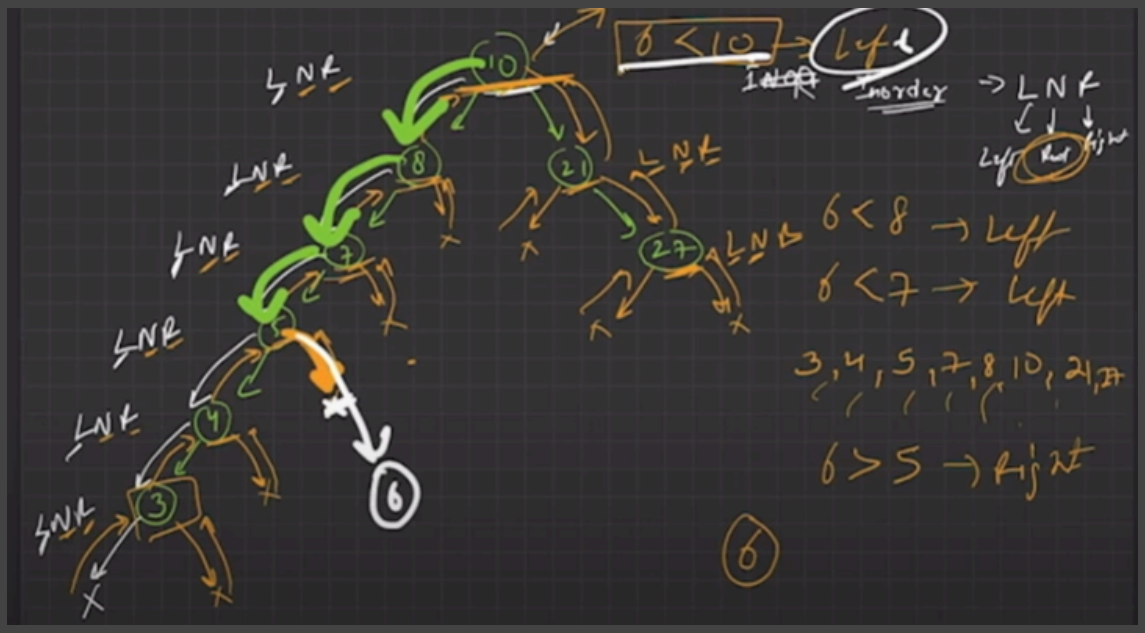
Binary Search Tree(BST)

Insert A Node In BST:

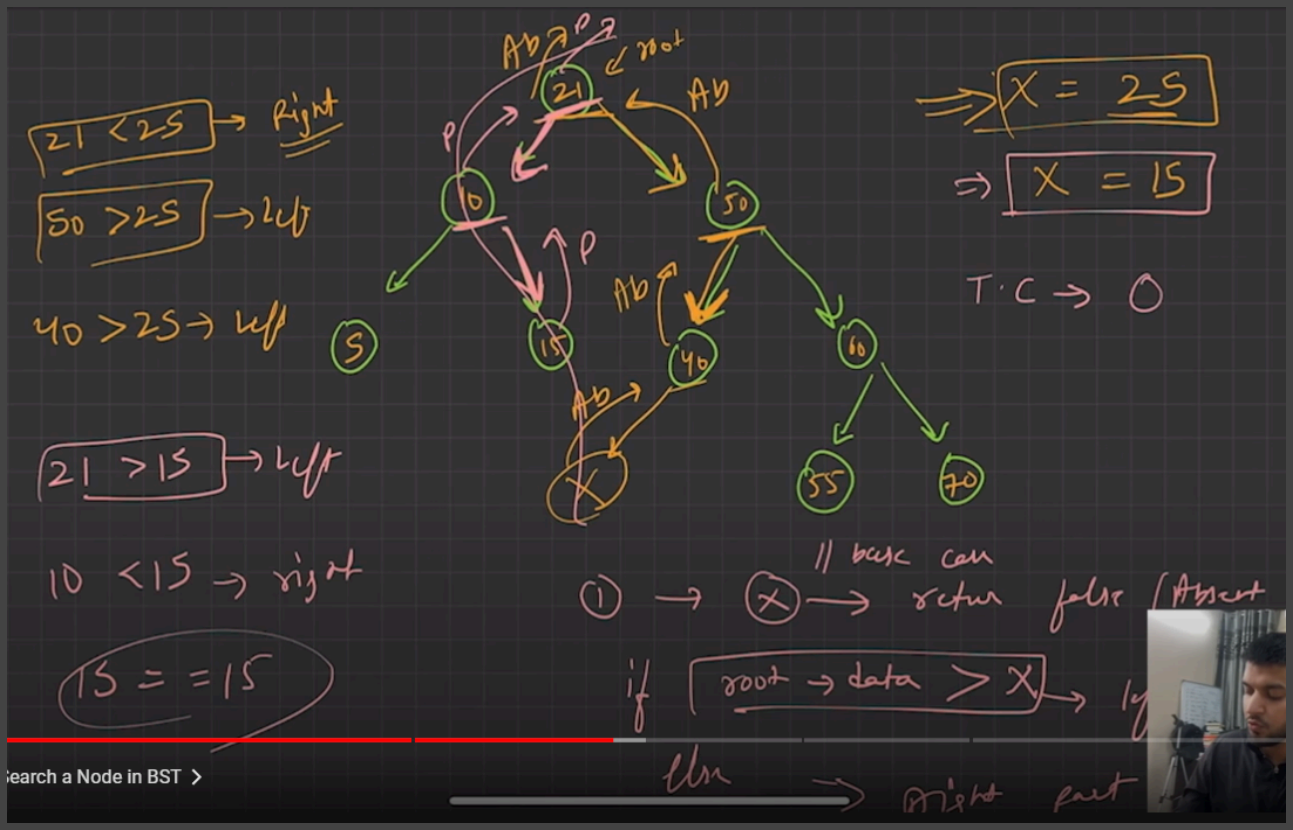




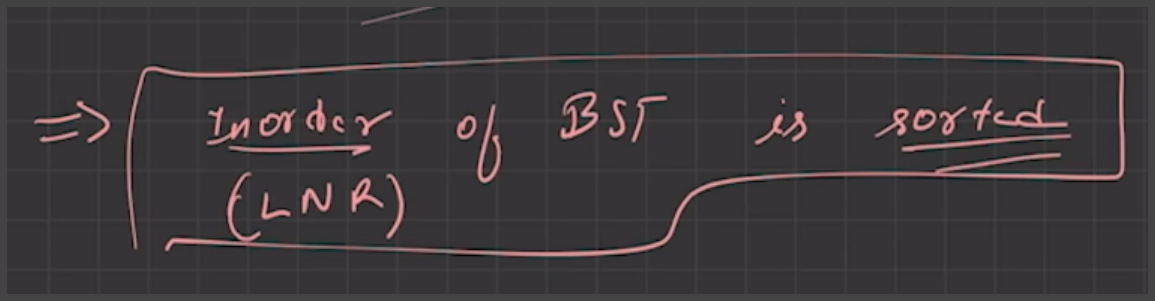


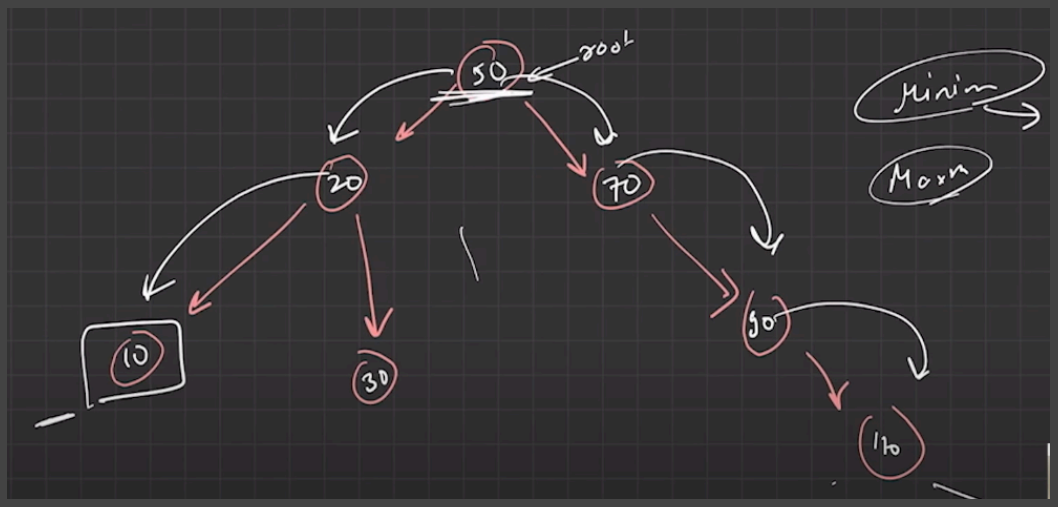


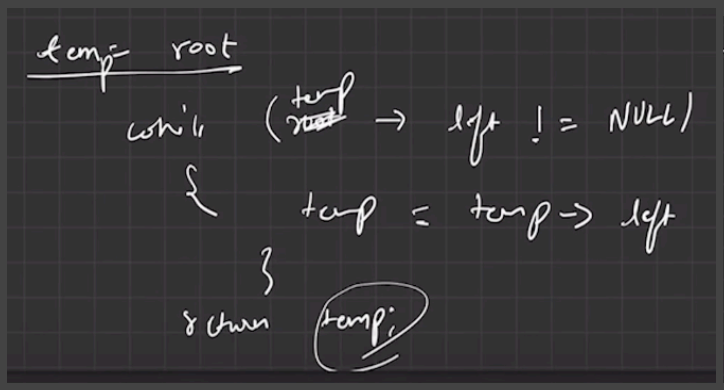
Search A Node In BST:

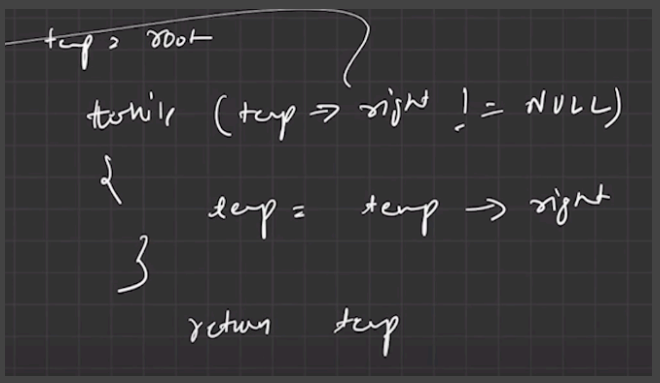


Minimum/Maximum Value In BST:

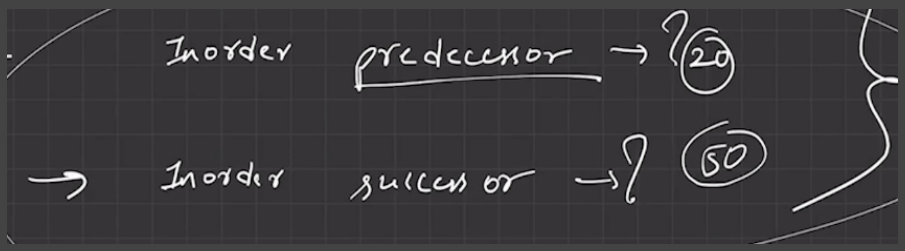








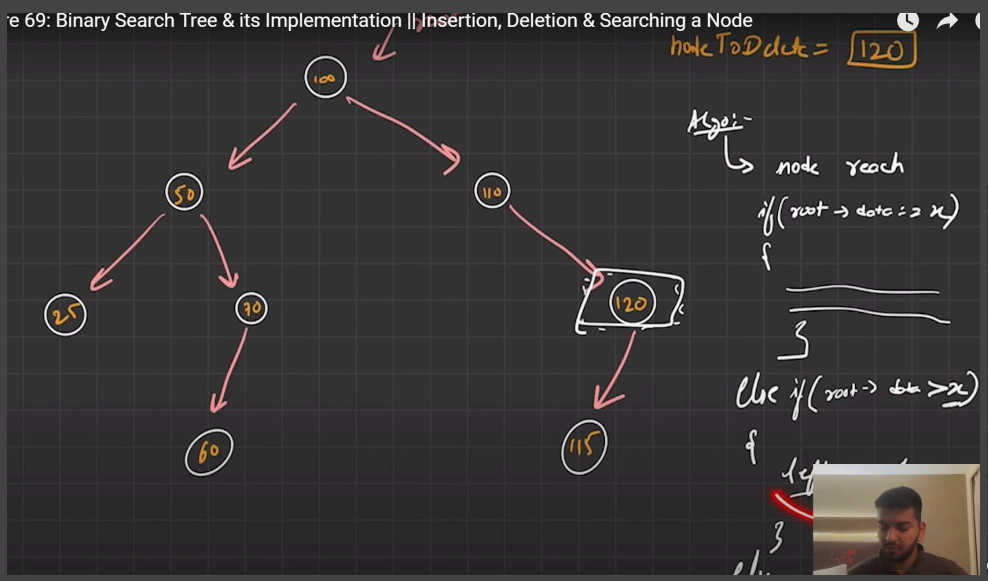
Inorder Predecessor In BST:

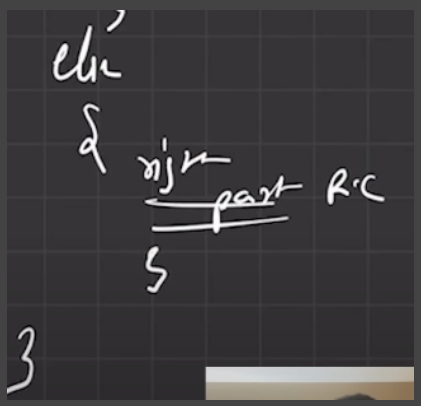


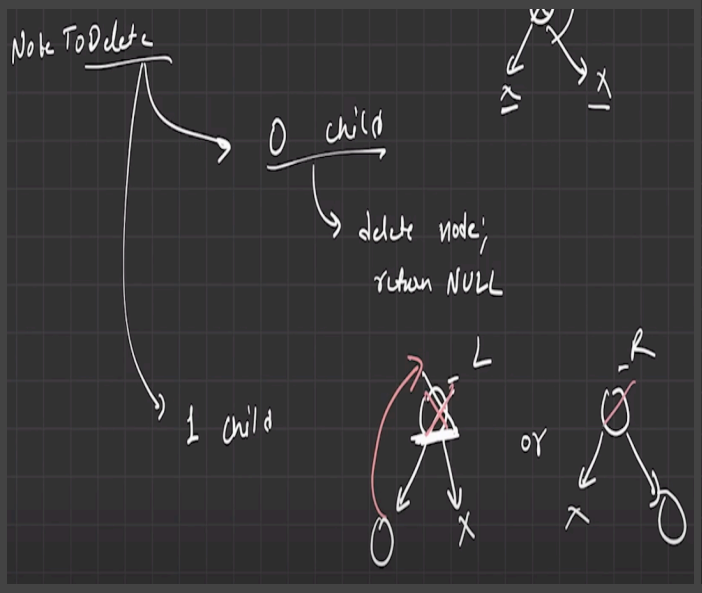
Inorder Successor In BST:

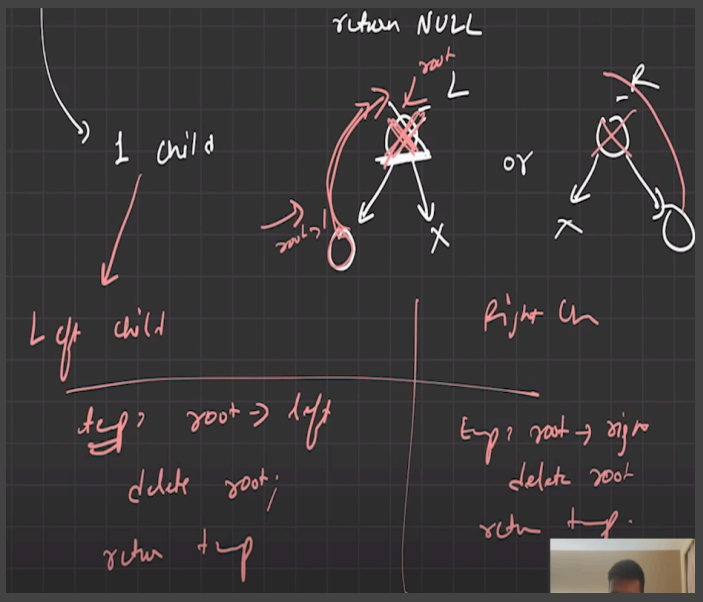
```
110
Printing Inorder
10 20 30 50 70 90 110
Printing Preorder
```

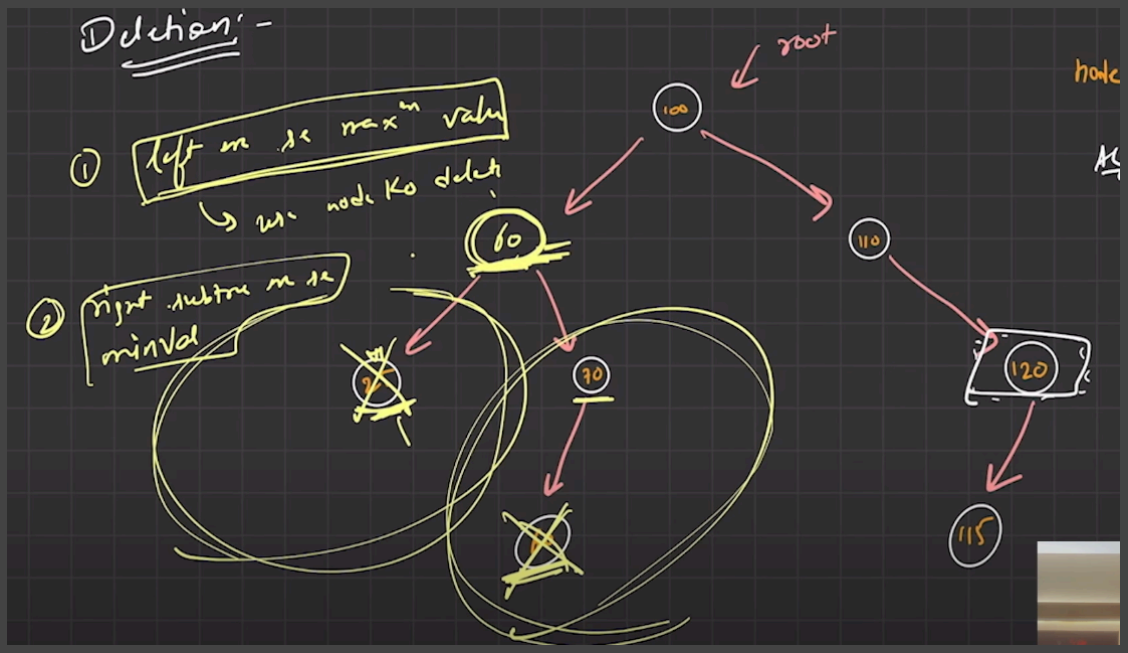
Delete A Node In BST:



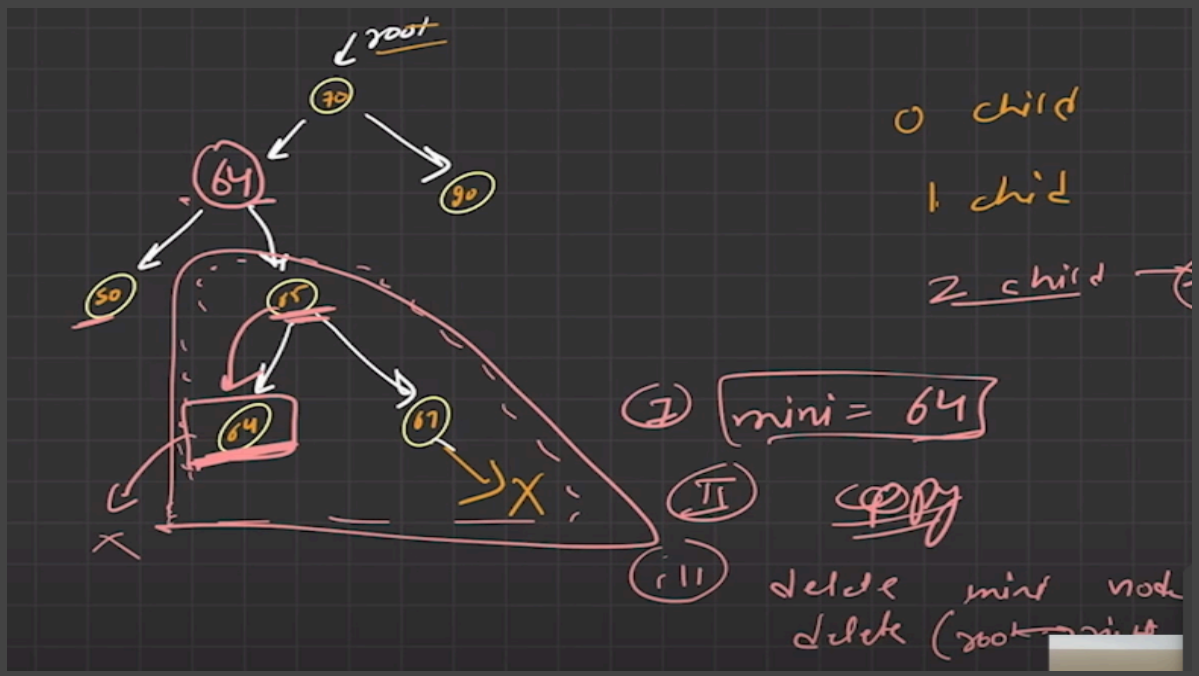



Note ToDelete
0 child
delete node;
return NULL
1 child
L
R
or
X

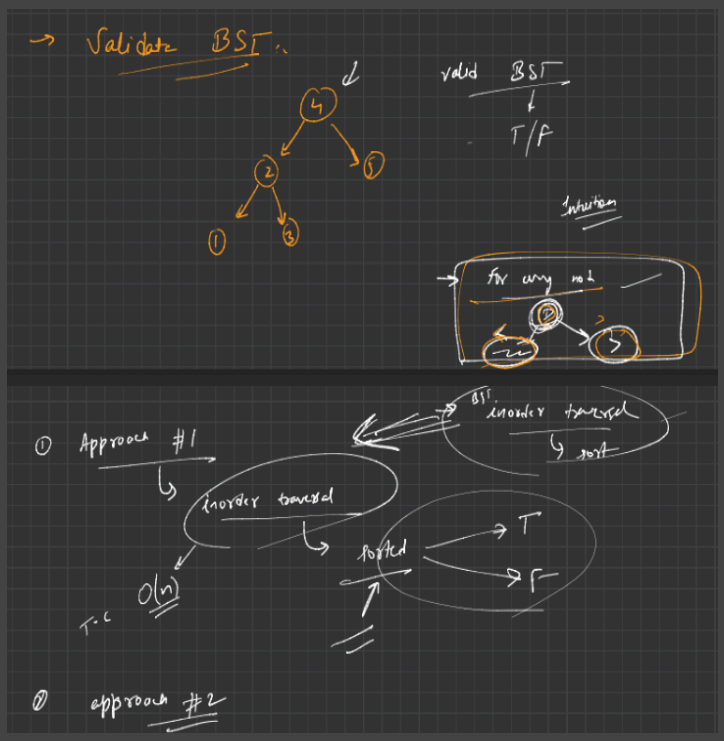

return NULL
root
L
R
1 child
or
X
Left child
Right Ch
temp = root -> left
delete root;
return temp
temp = root -> right
delete root
return temp.

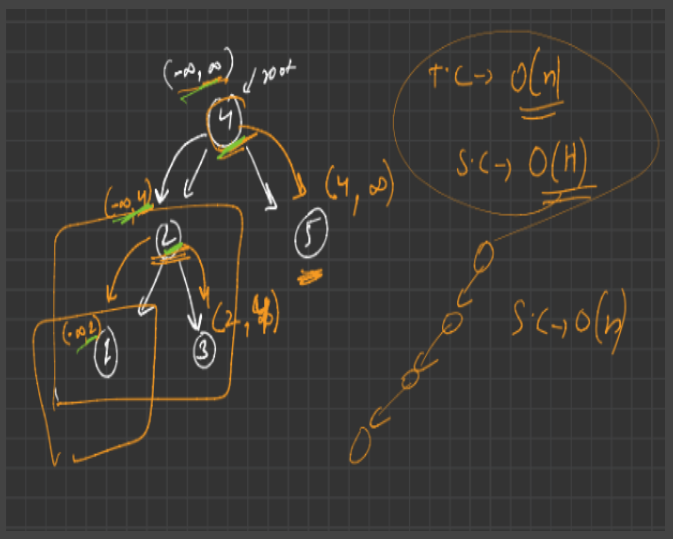

Deletion:-
root
node
1 left se max^m value
2 right subtree se
minVal
100
60
110
70
120
115

Any one 1 or 2

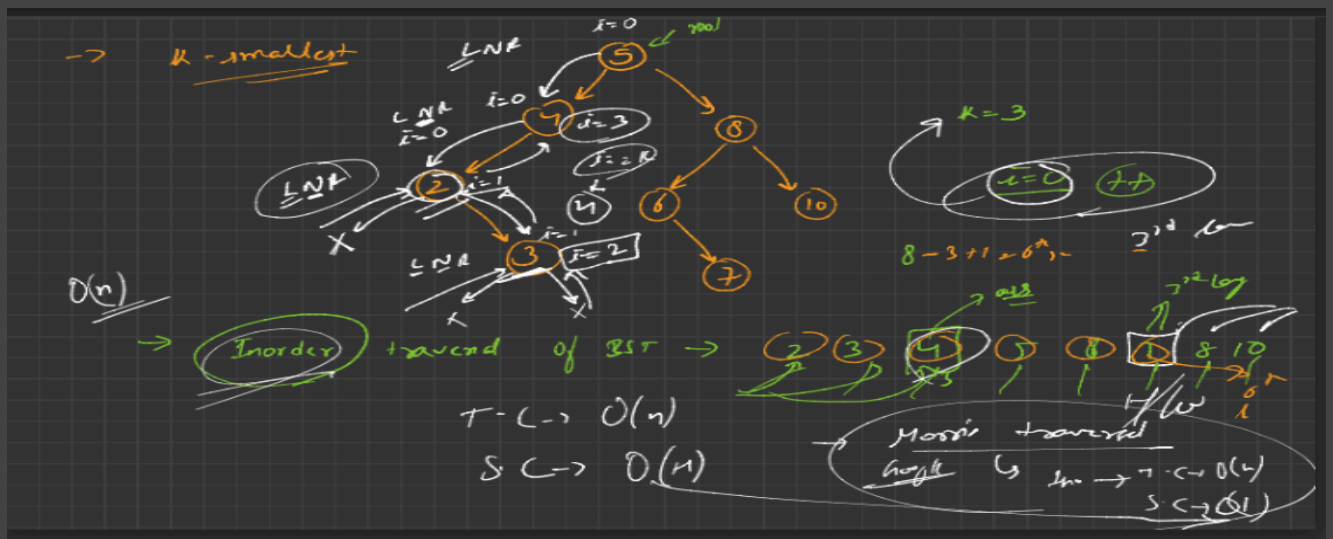

root
70
64
90
50
65
64
67
X
0 child
1 child
2 child
2 mini = 64
delete mini node
delete (root

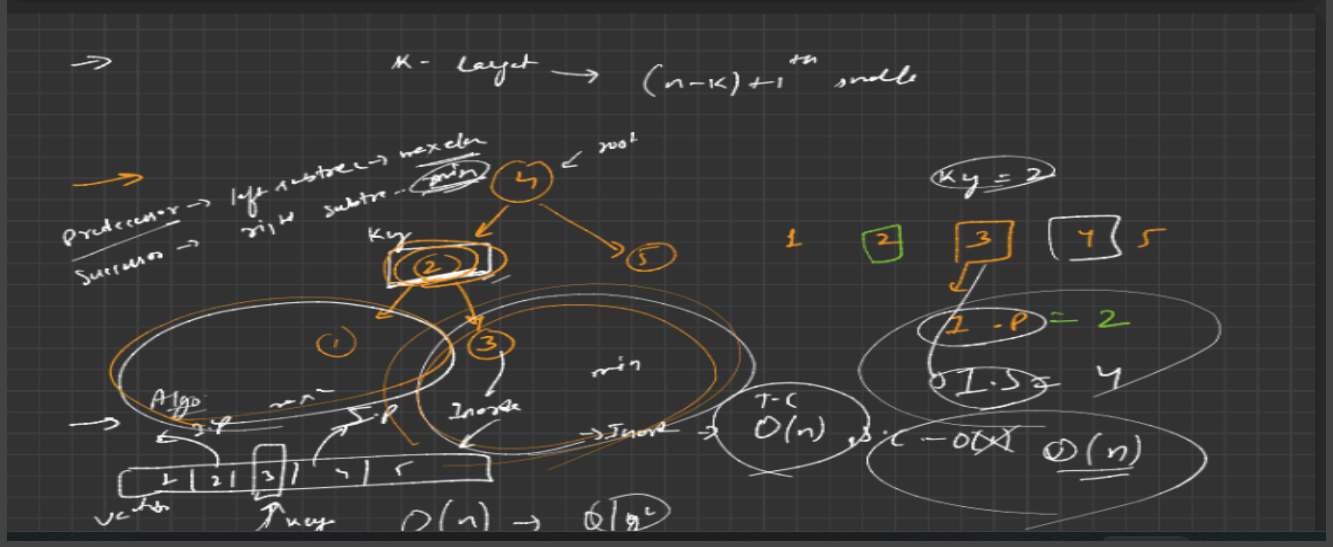
Validate BST:

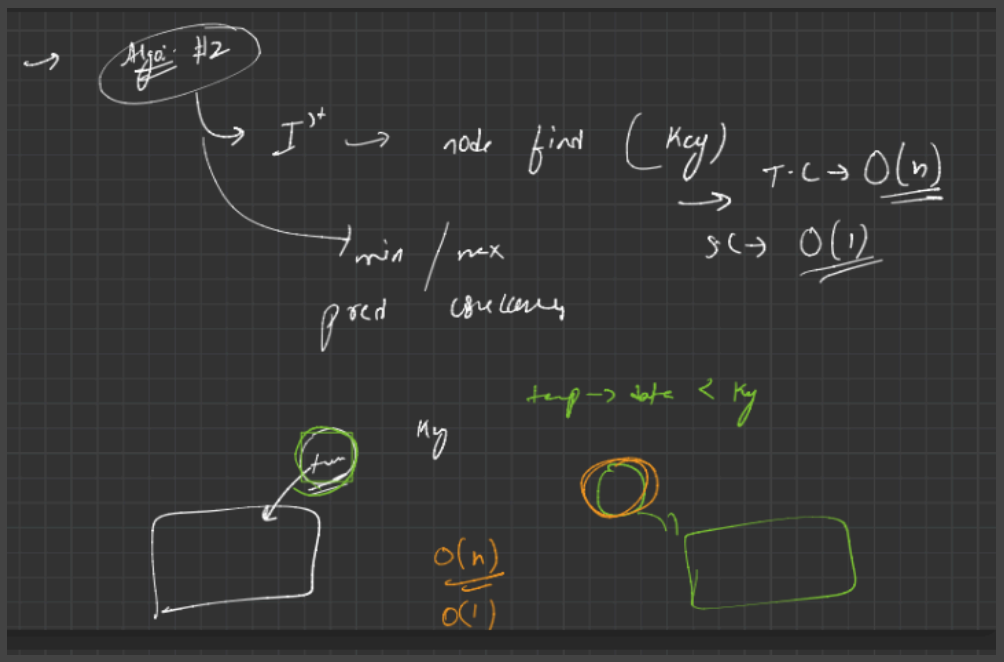




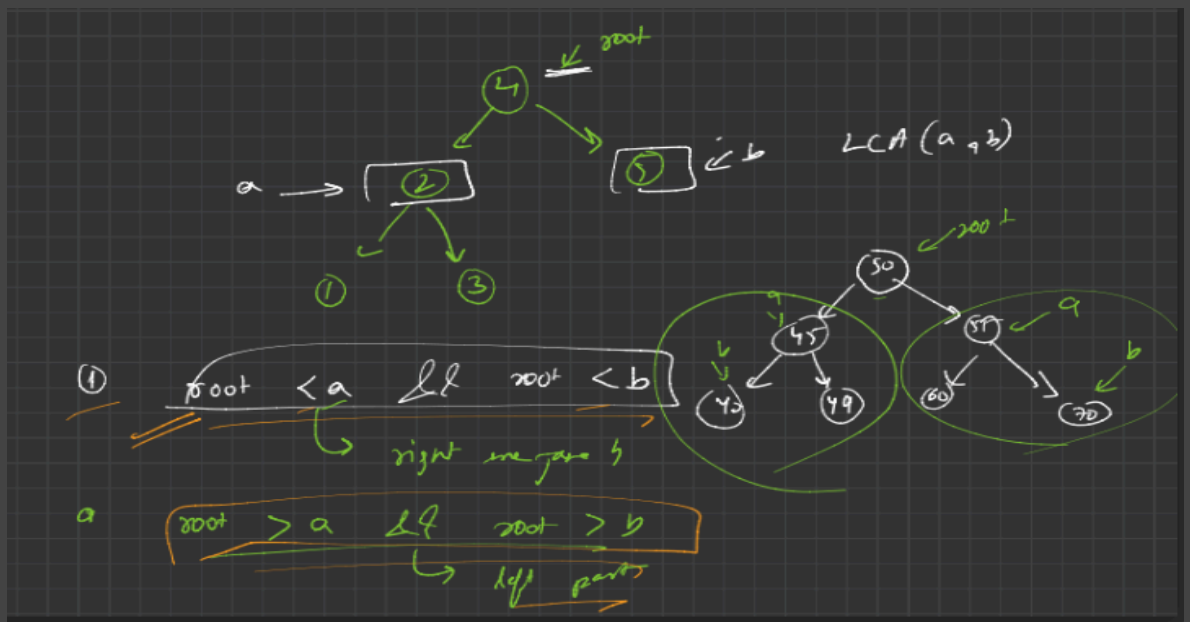
Kth Smallest/Largest Element In BST:

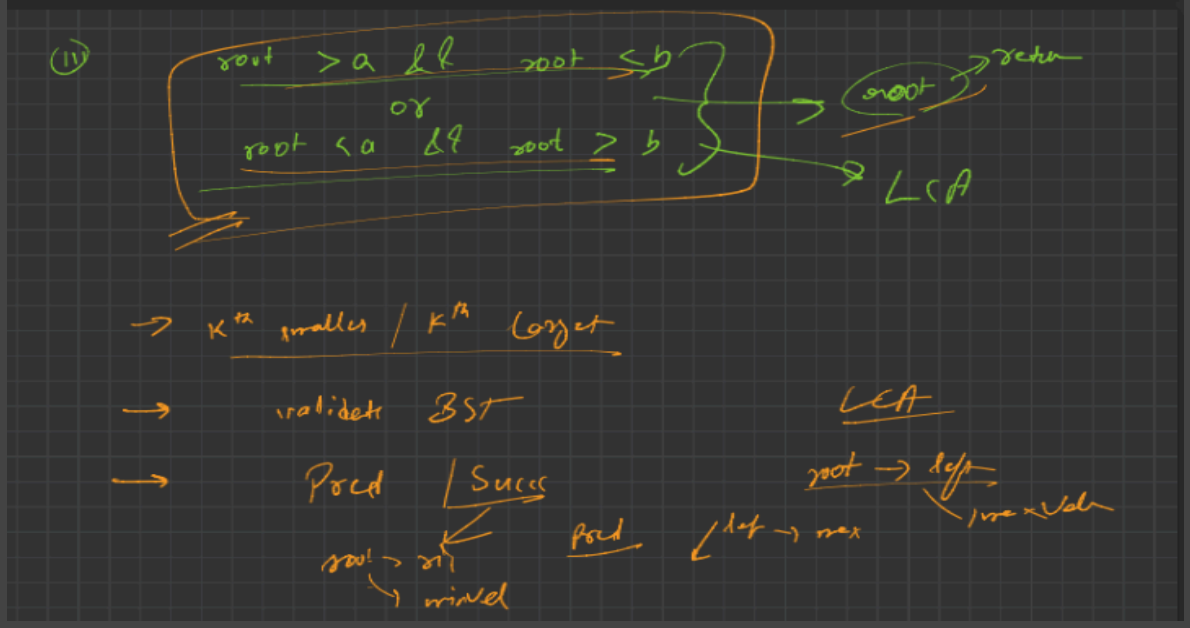




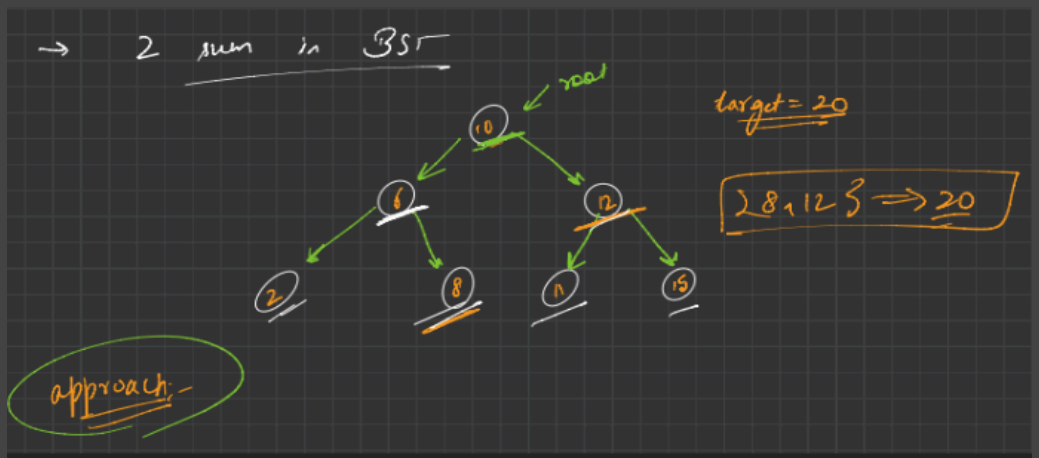


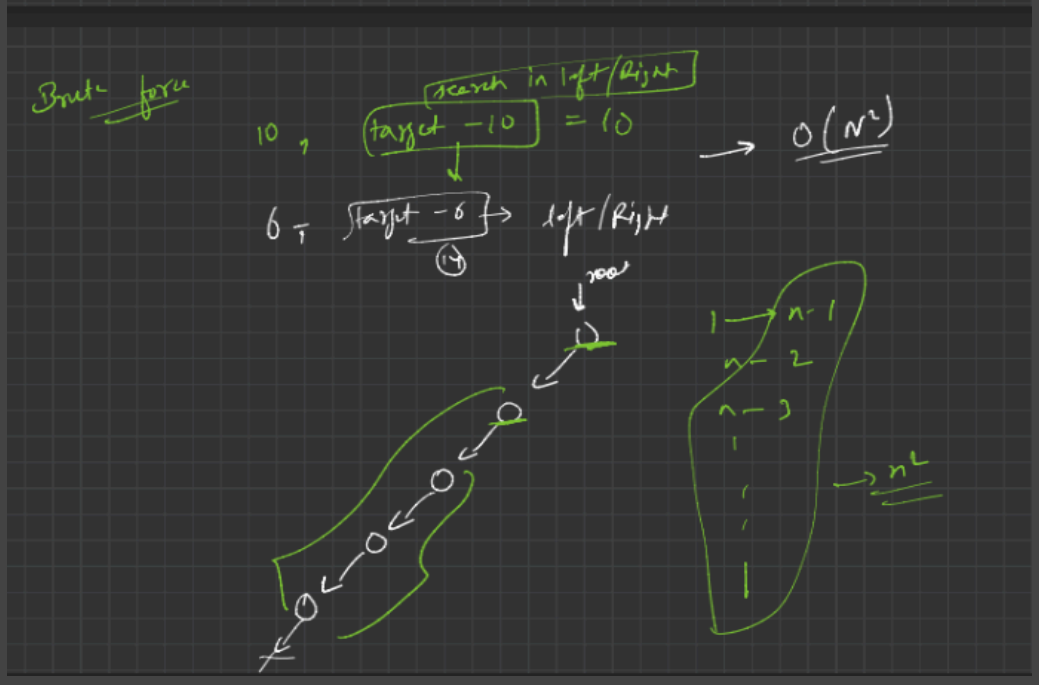
Lowest Common Ancestor In BST:

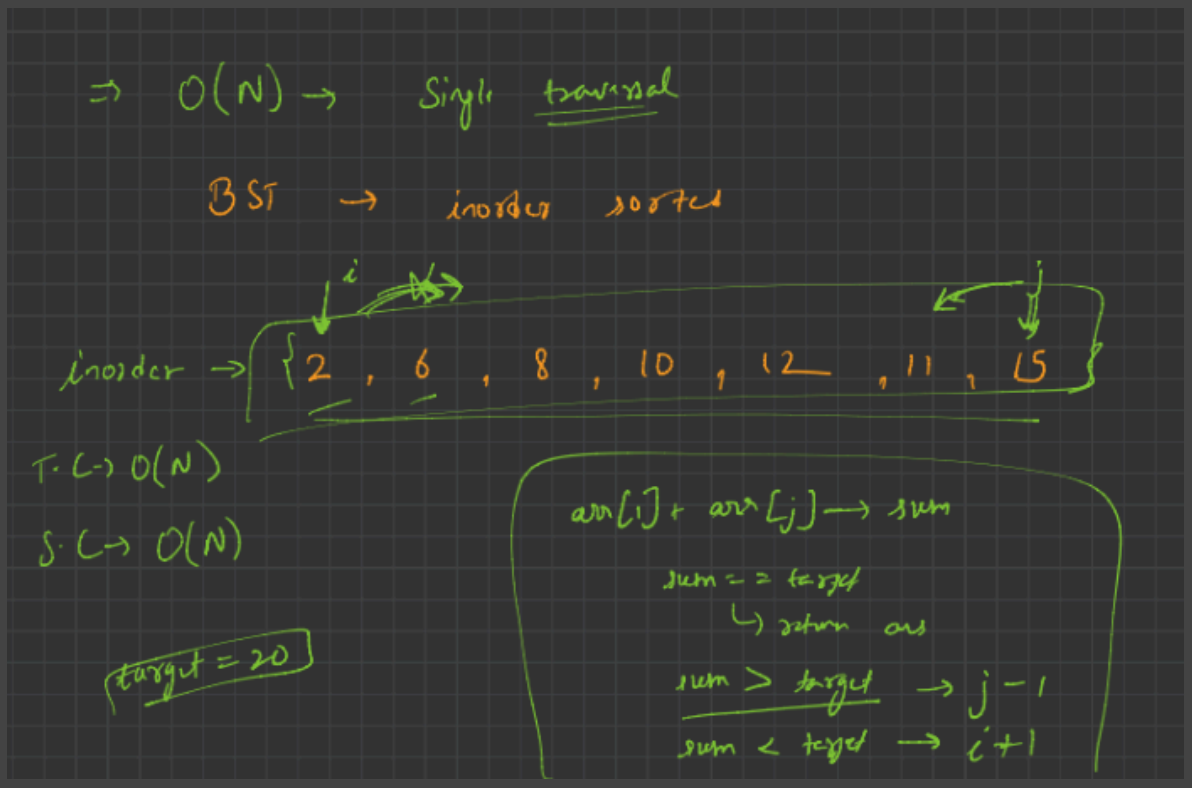




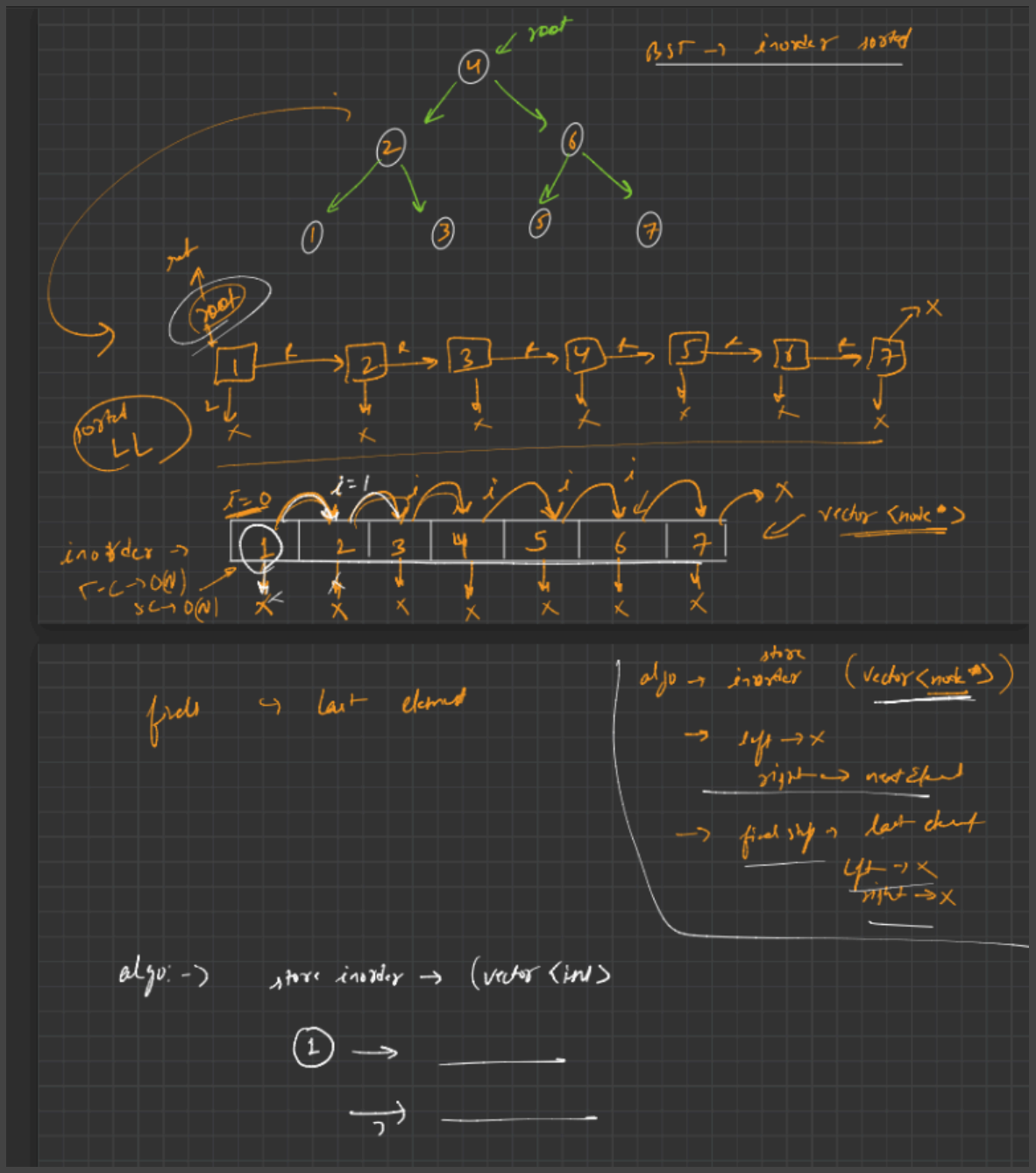
Two Sum In BST:



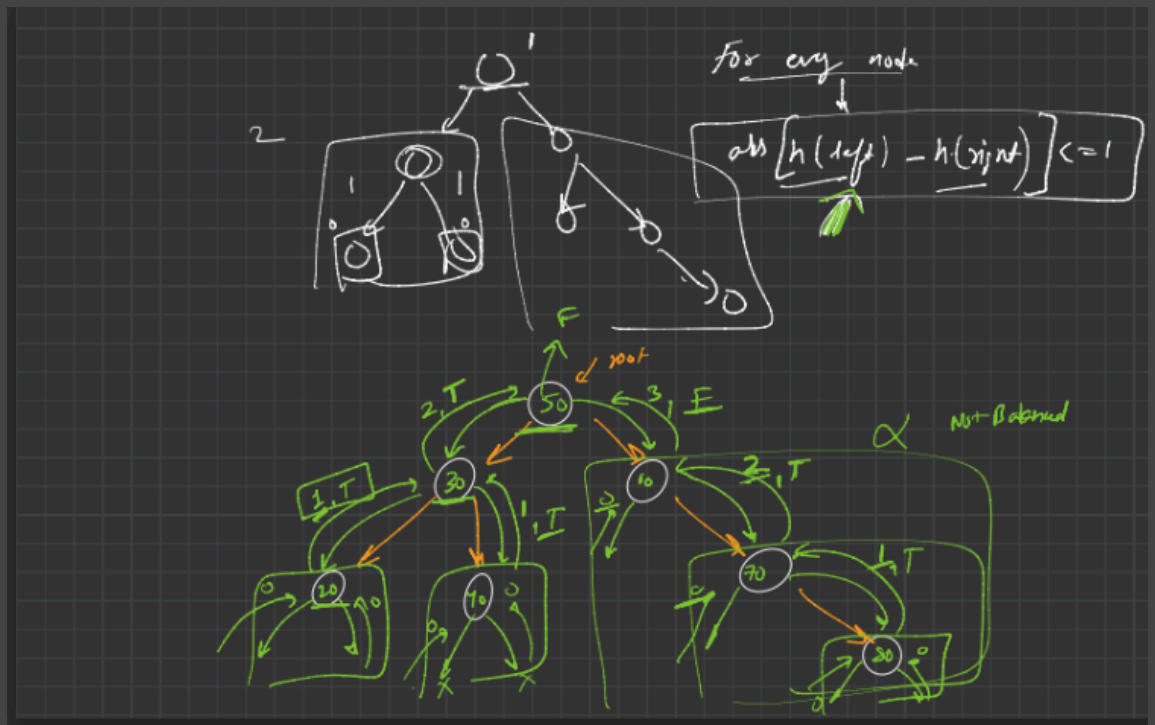


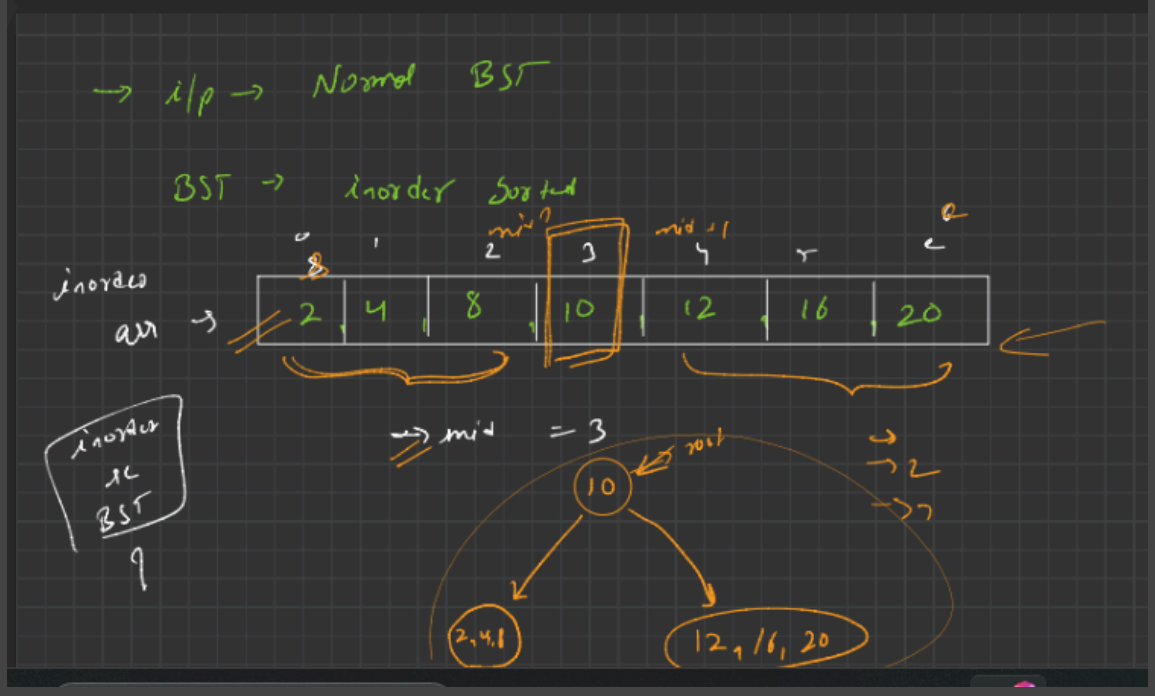


Flatten BST To A Sorted List:

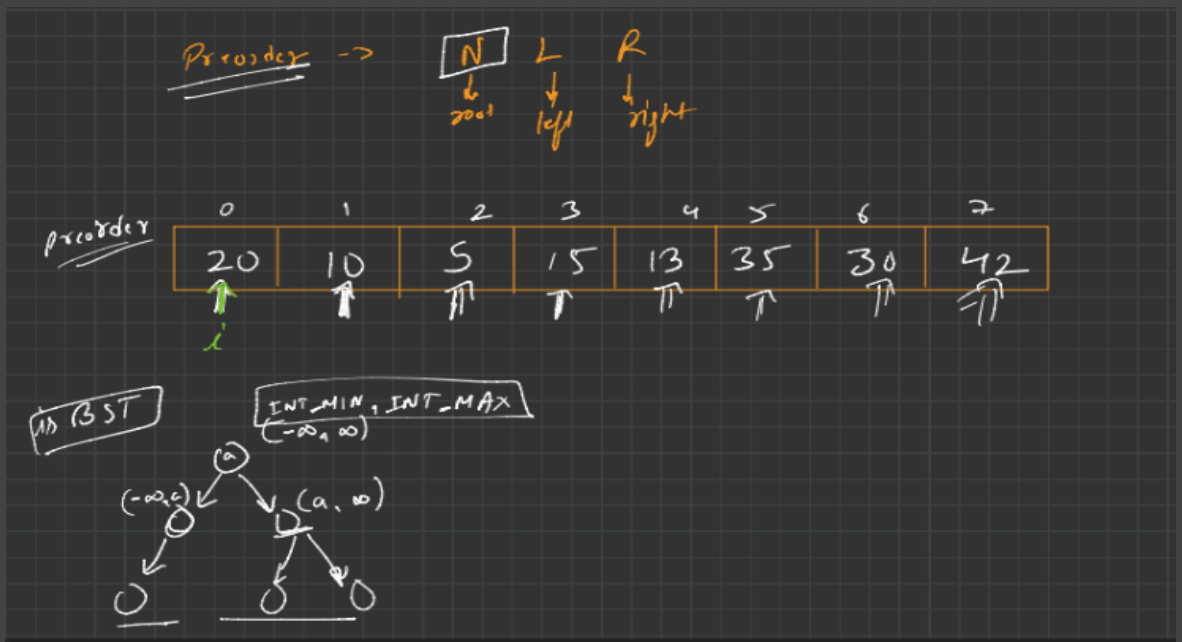


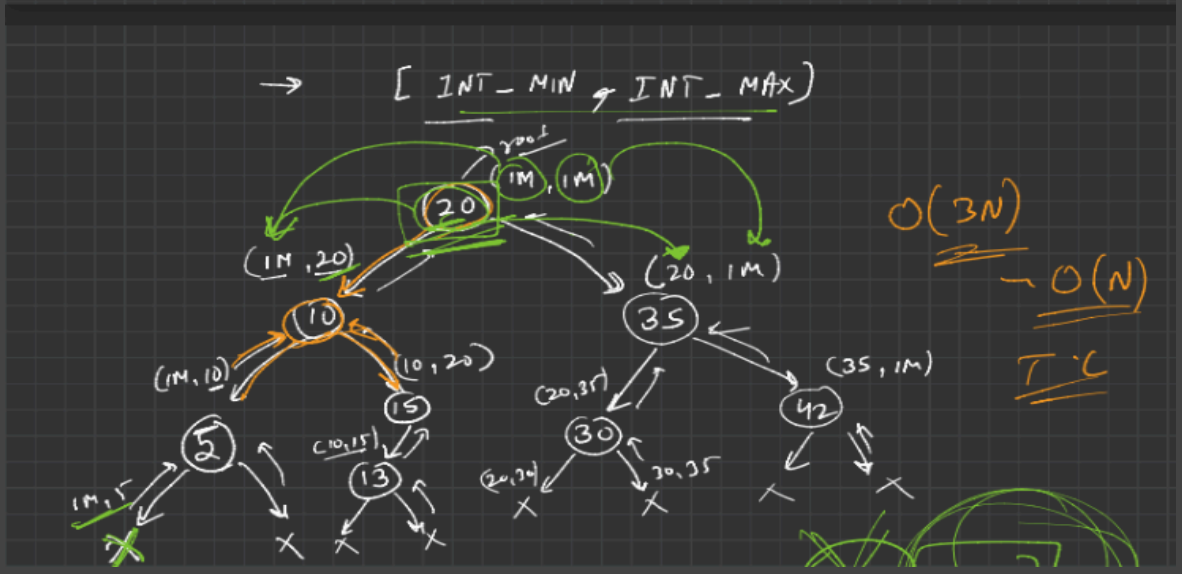
Normal BST To Balanced BST:



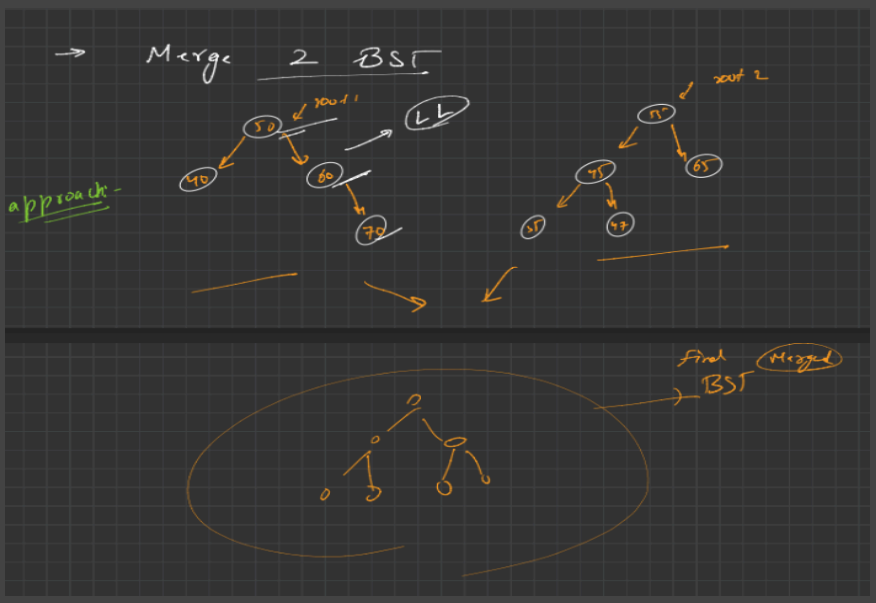


BST From Preorder Traversal:





Merge Two BST:



approach #1

Inorder To BST

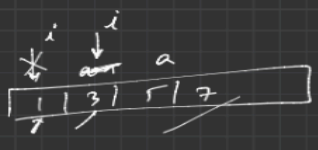
↳ i/p → BST 1 , BST 2

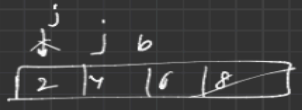
step 1 → vector → inorder 1 (m) , inorder 2 (n) [store inorder]

step II → Merge 2 sorted arrays → arr3 = [sorted, m+n]

step III → inorder to BST → ans

T.C → O(m+n)
S.C → O(m+n)





ans (a+b)

⇒ O(m+n) ✗

S.C →

S.C → O(h1 + h2)

Approach #2

↳ i/p → BST 1 Root 1 , BST 2 Root 2

convert BST into sorted LL step 1

sorted LL , sorted LL

T.C → O(m) O(n)
S.C → O(h1) O(h2)

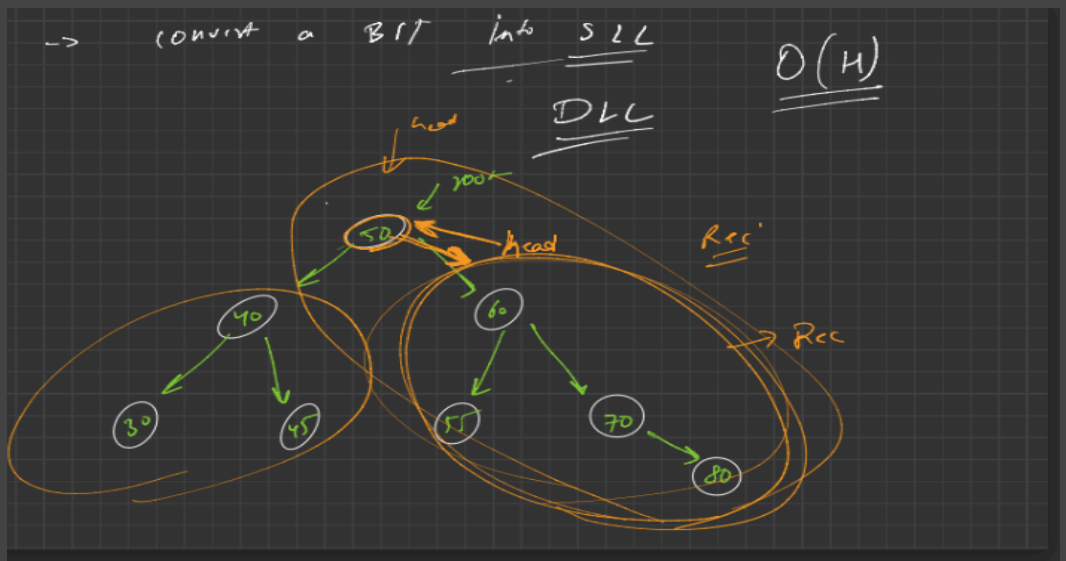
step 2 → merge 2 sorted Linked List → T.C → O(m+n), S.C → O(1)

1 sorted LL

step 3 → sorted LL → BST → S.C → O(h1+h2)

Balanced BST

① convert a BST into sorted LL → O(n) → O(h)
② Merge 2 sorted Linked List
③ sorted LL → BST

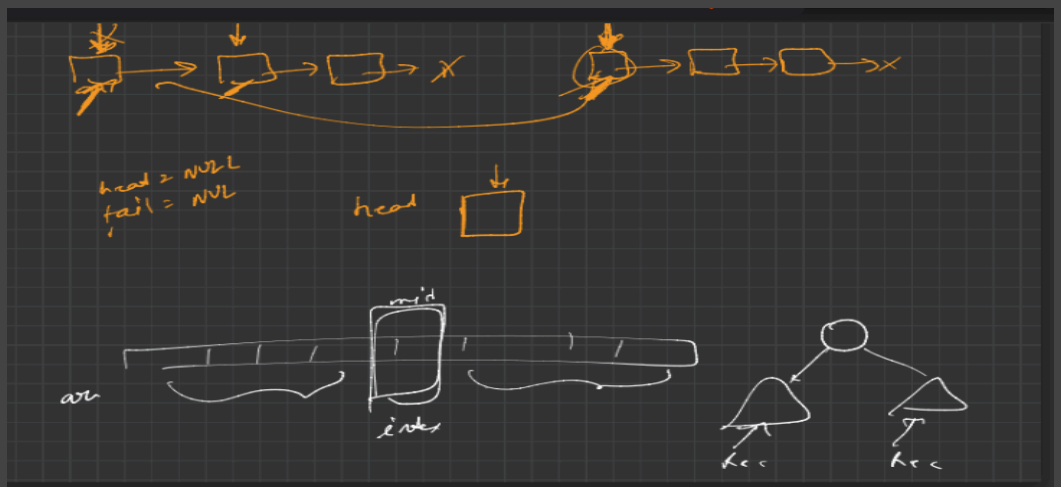


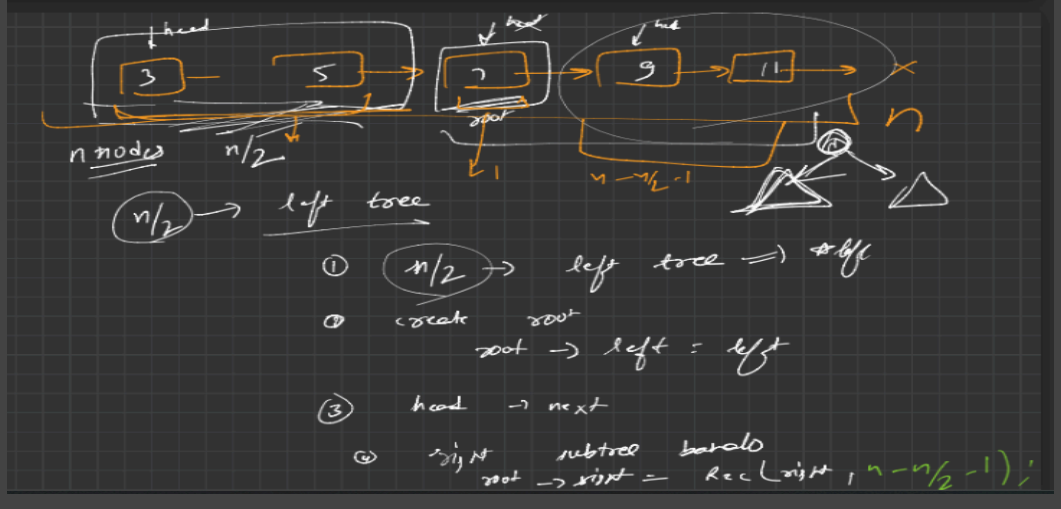
algo -> convert right subtree into LL -> (head of LL)

-> root -> right = head
head -> left = root

-> head = root

-> convert left subtree into LL -> head of LL



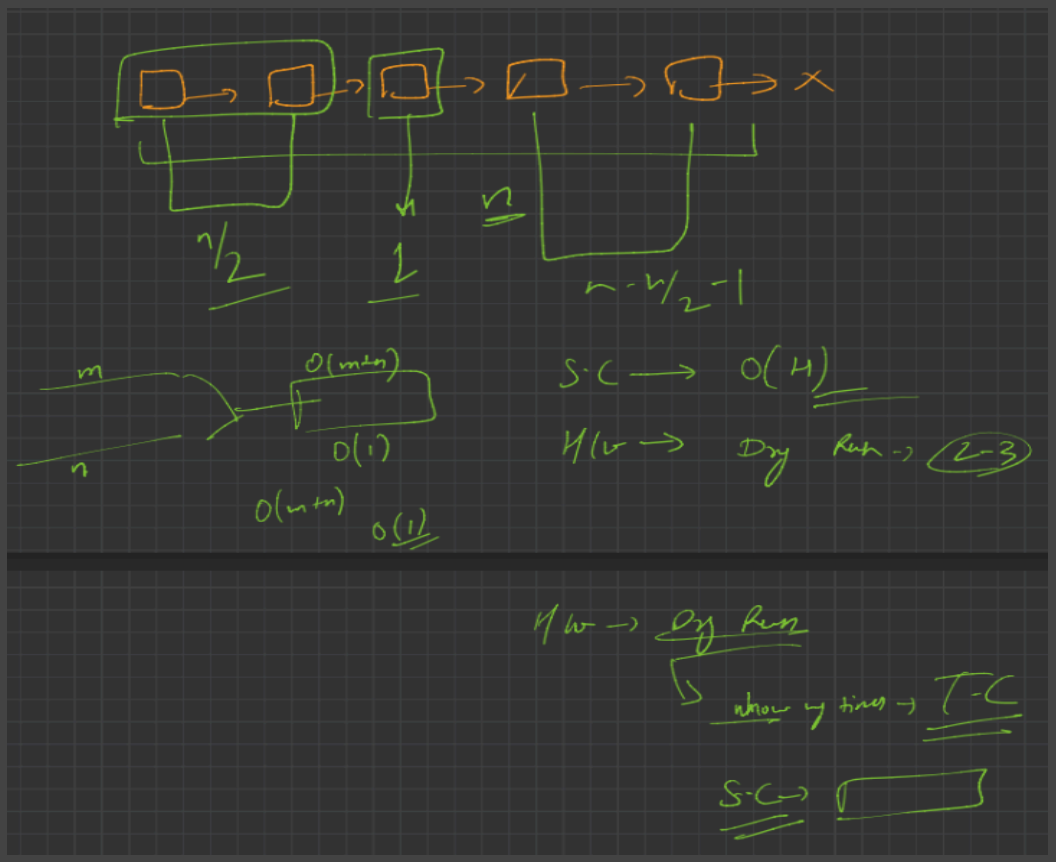


① n/2 -> left tree => left

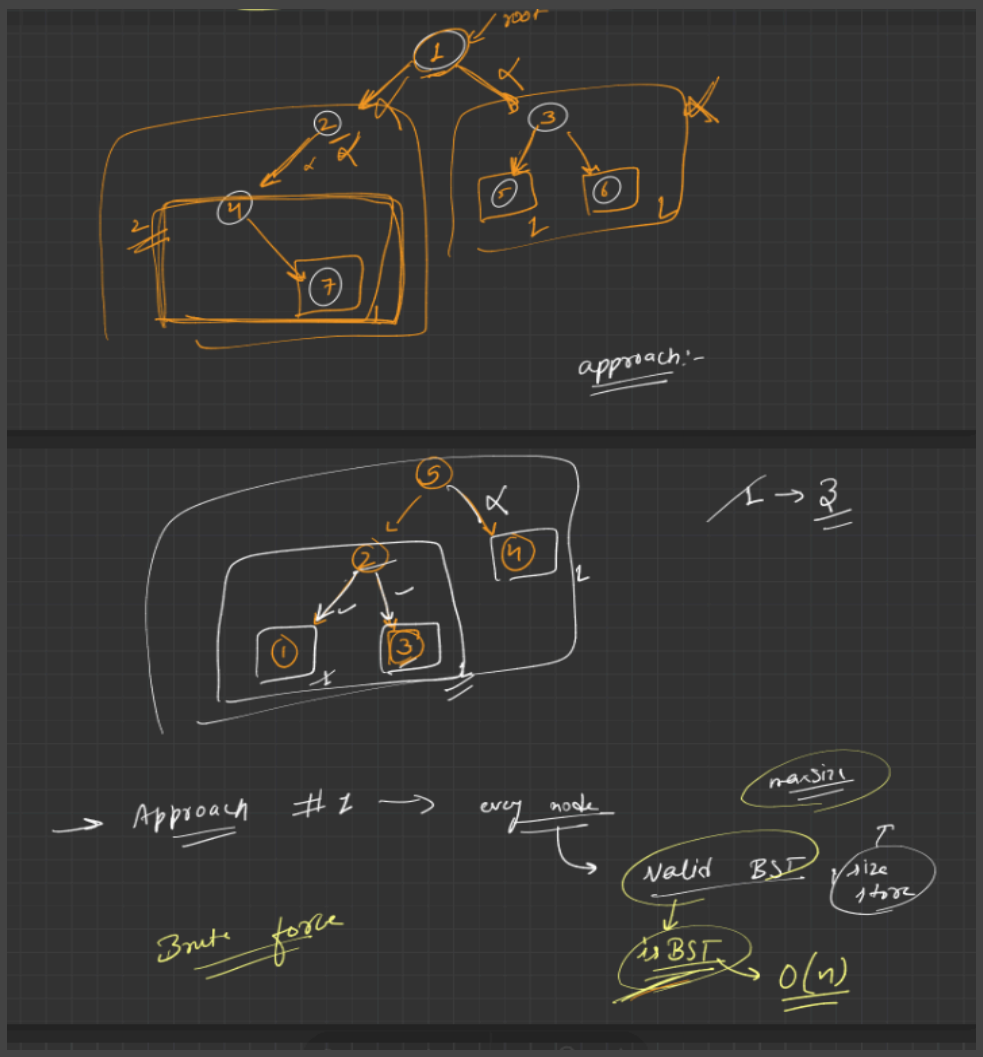
② create root
root -> left = left

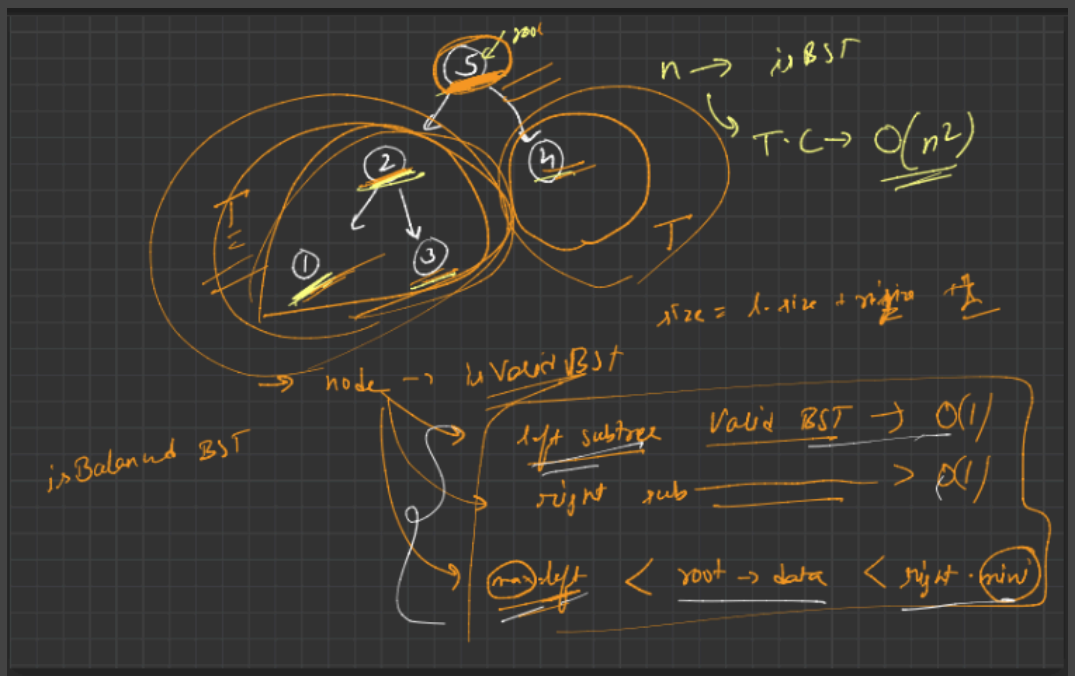
③ head -> next

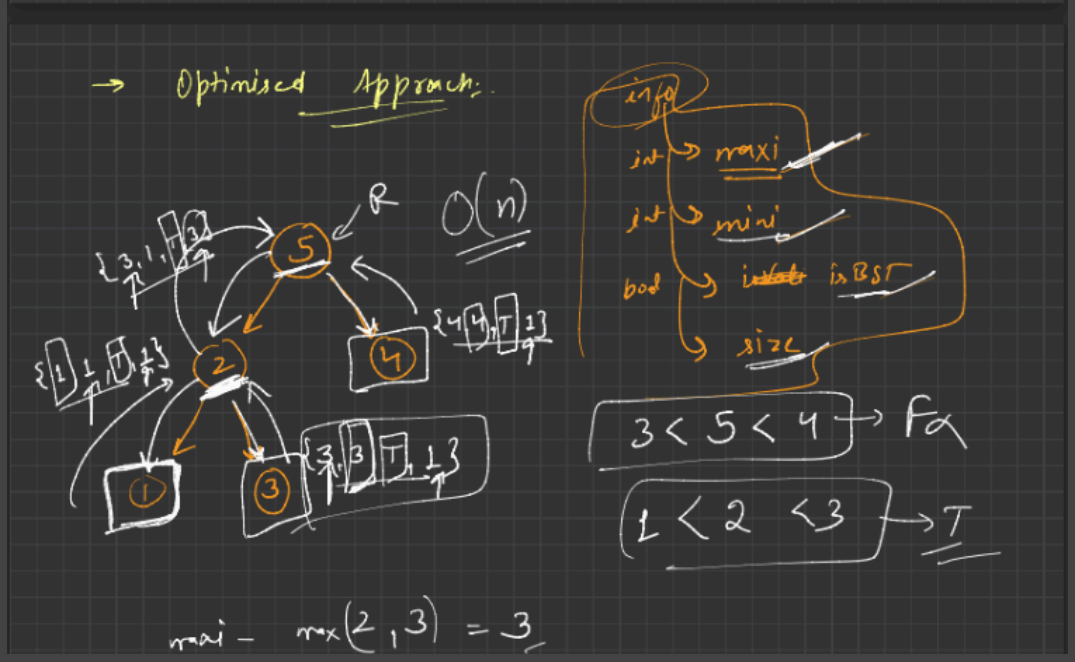
④ right subtree banaLO
root -> right = Rec(right, n - n/2 - 1);

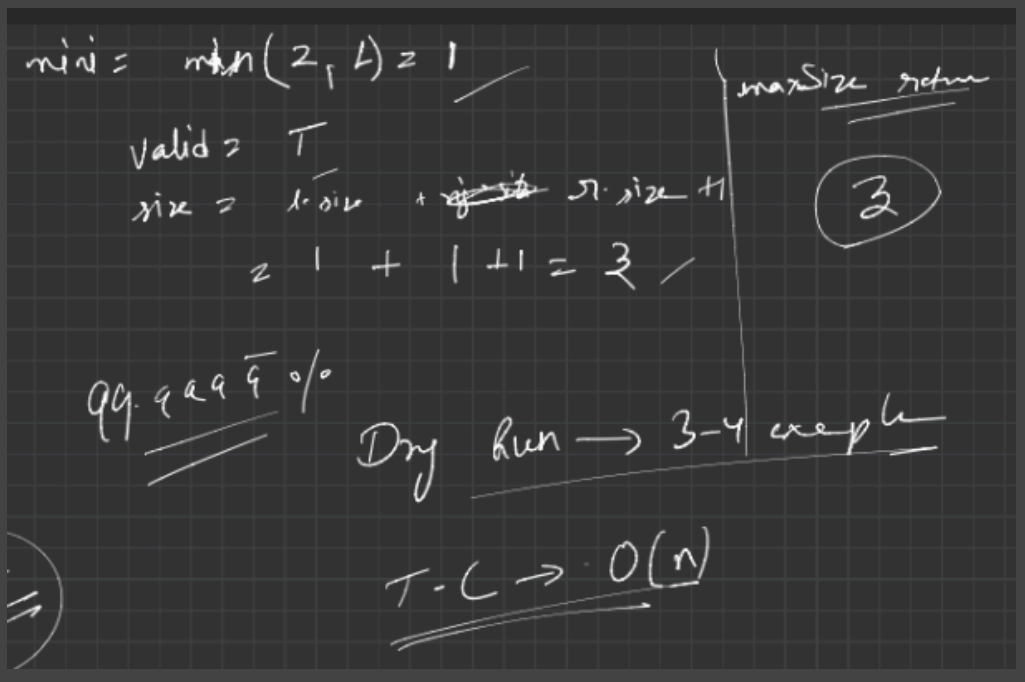


Largest BST In A Binary Tree:




root
5
2
4
1
3
T
n → isBST
T.C → O(n²)
size = l.size + r.size + 1
node → isValidBST
isBalanced BST
left subtree Valid BST → O(1)
right sub → O(1)
root → data

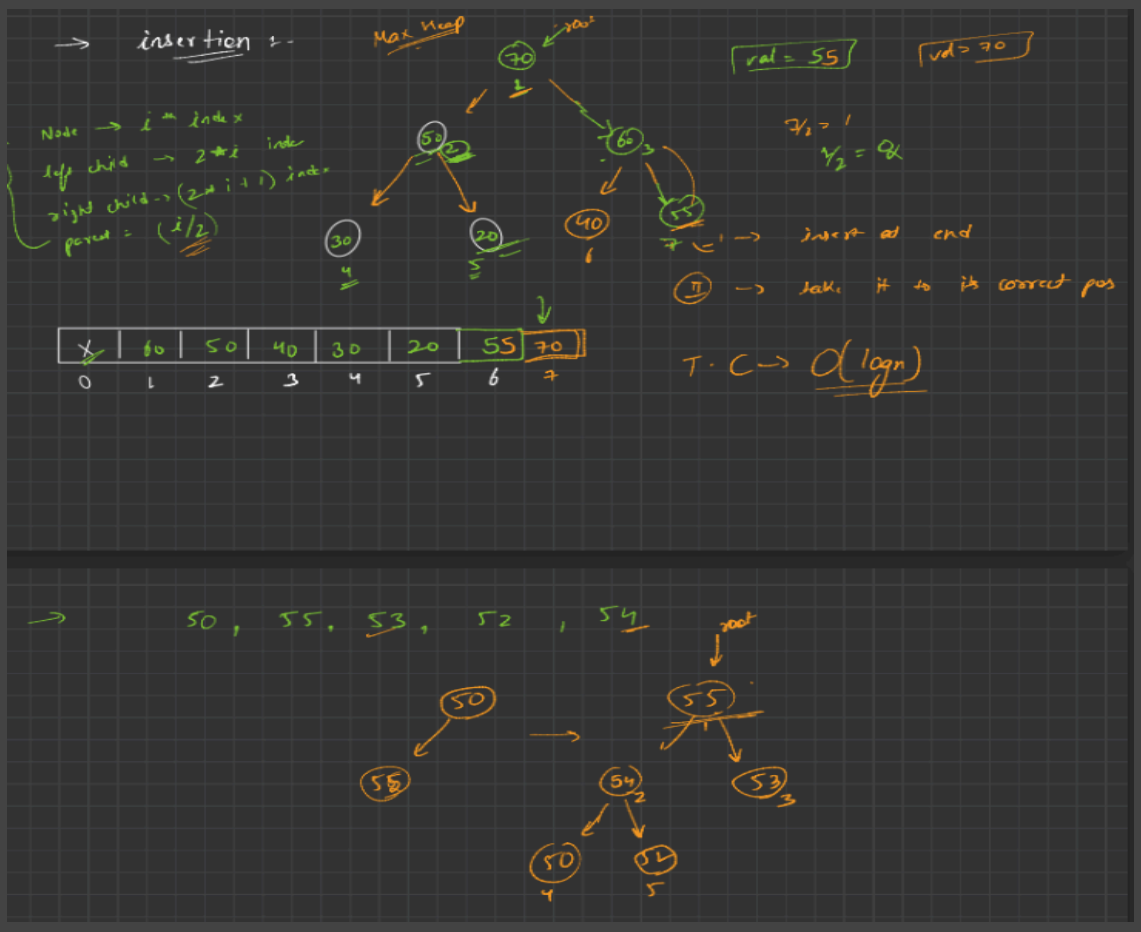

→ Optimised Approach:
info
int → maxi
int → mini
bool → isBST
size
R
O(n)
3 < 5 < 4 → F
1 < 2 < 3 → T
maxi = max(2, 3) = 3


mini = min(2, 1) = 1
Valid = T
size = l.size + r.size + 1
= 1 + 1 + 1 = 3
maxSize return
3
Dry Run → 3-4 example
T.C → O(n)

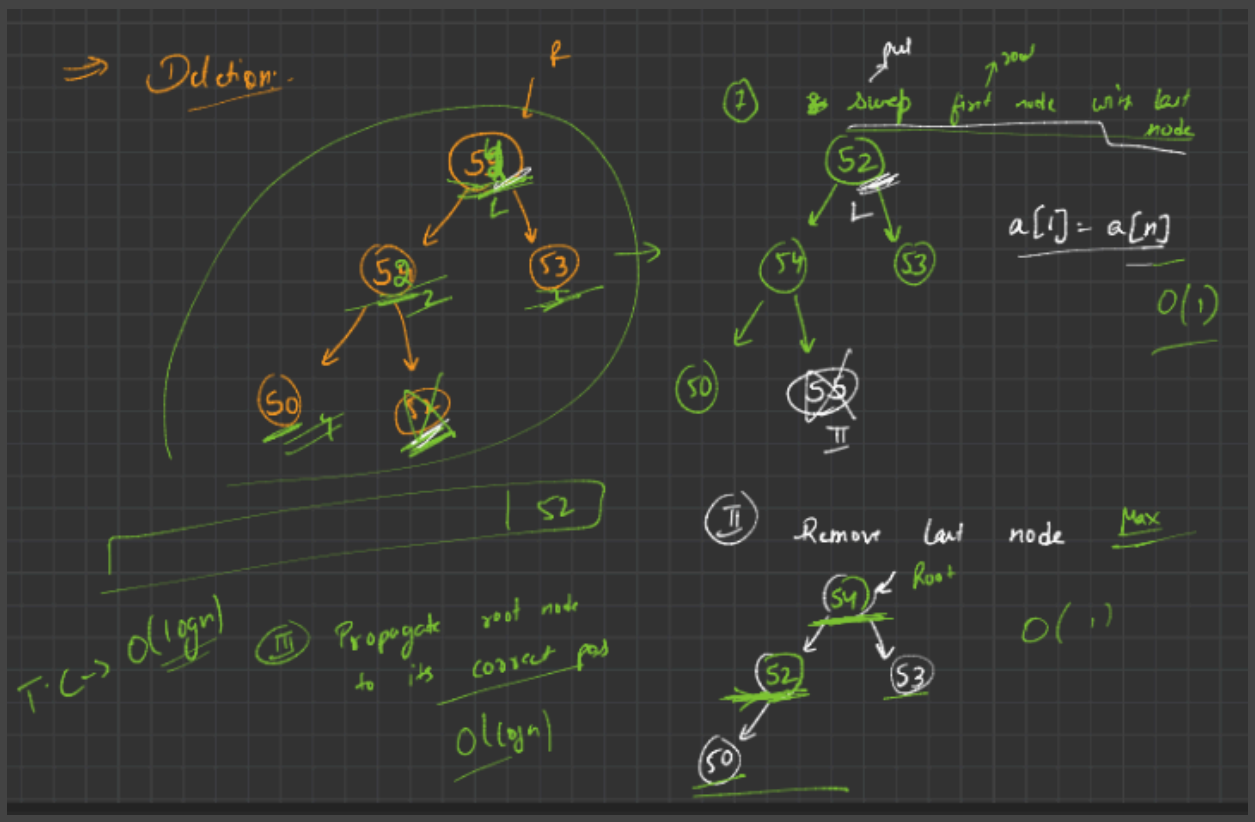
Heap


Heap
what is Heap?
is a CBT
that comes with a
Heap Order Property
Max Heap
Min Heap
what is CBT?
every level is completely filled except the last level
nodes always added from the left
lean towards the left
almost 2 child
BT
CBT
Heap → array form

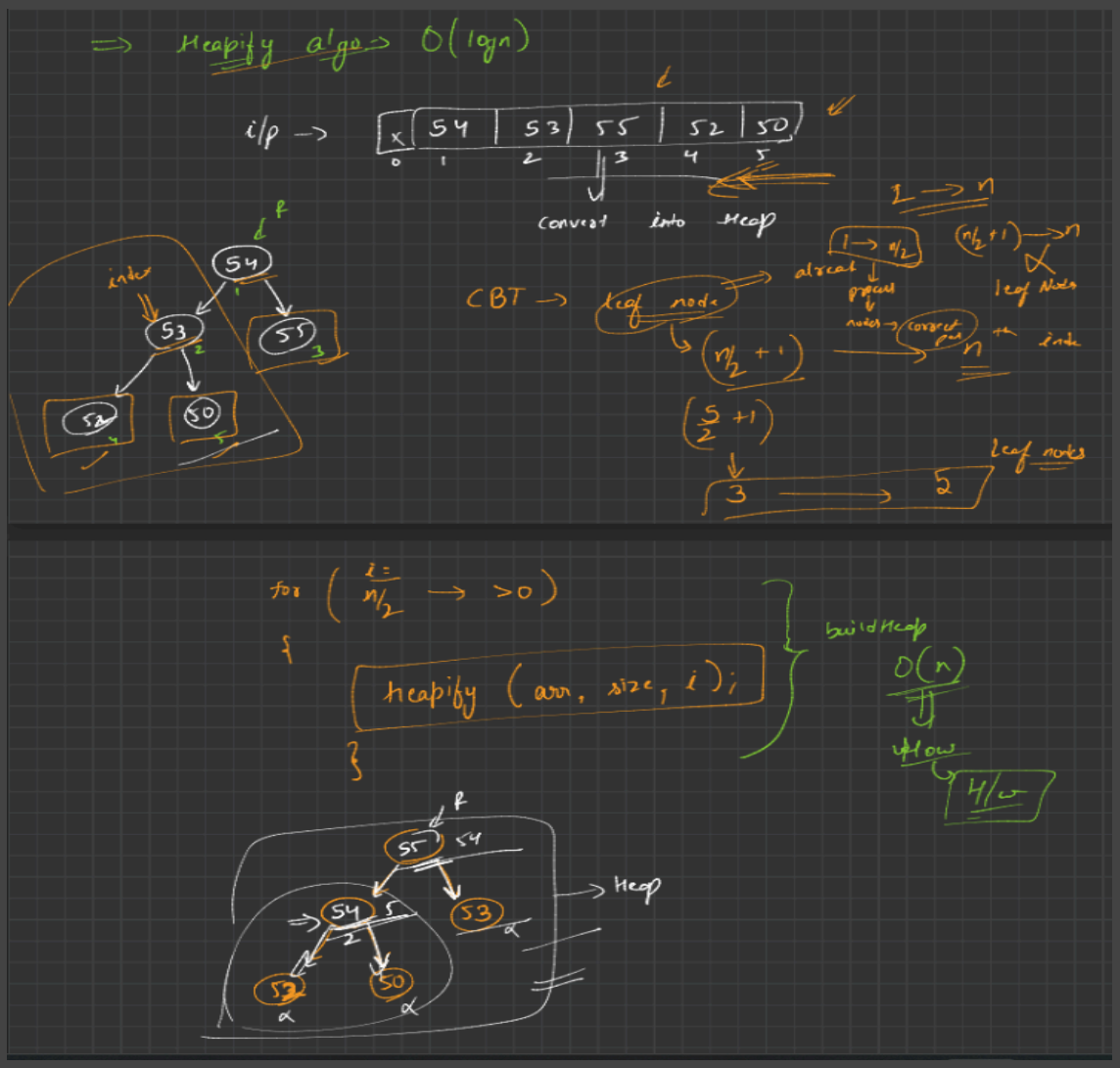
Insert:


insertion :-
Max Heap
root
Node → i index
left child → 2*i index
right child → (2*i+1) index
parent = (i/2)
val = 55
val = 70
i/2 = 1
i/2 = 2
i → insert at end
π → take it to its correct pos
T.C → O(logn)
50, 55, 53, 52, 54
root

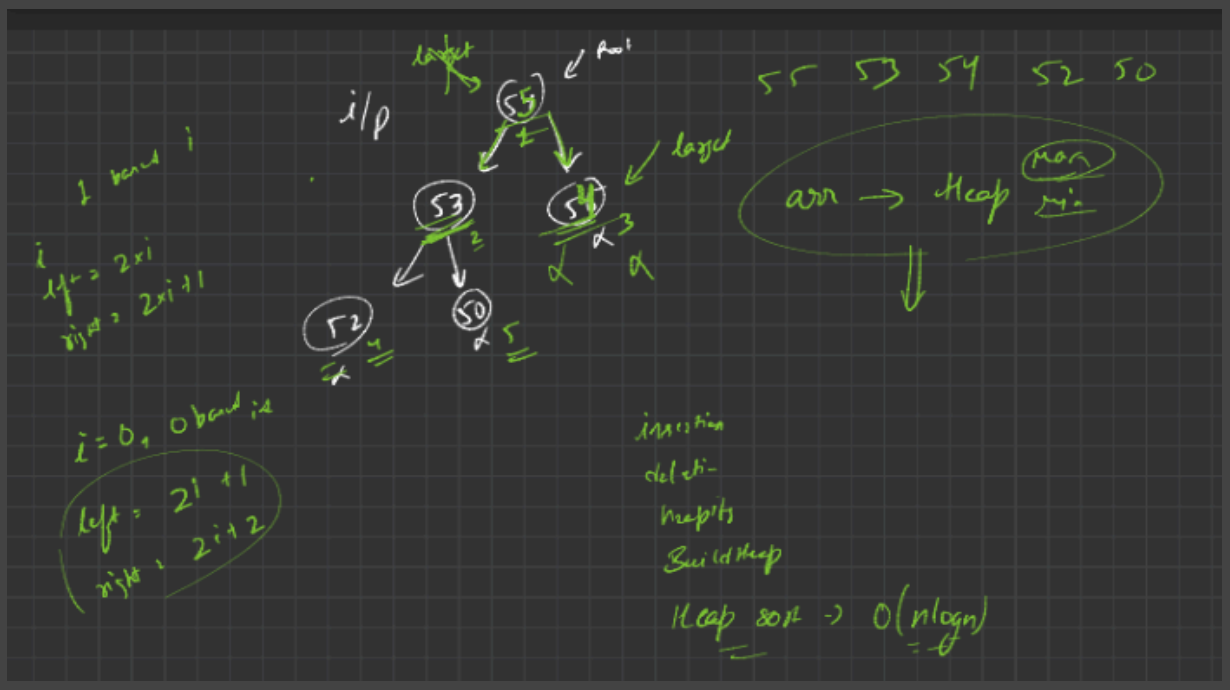
Delete:



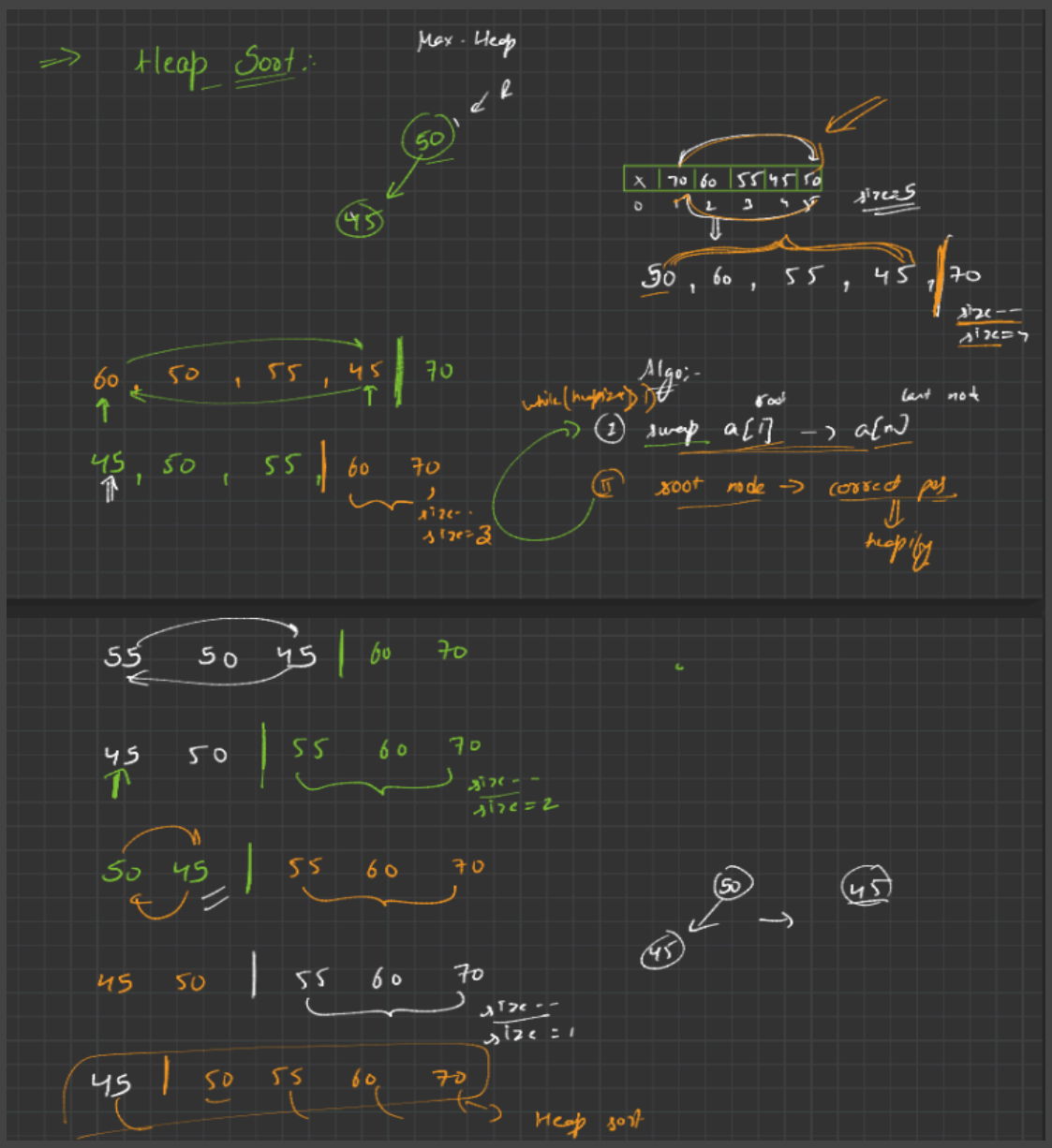
Heapify:

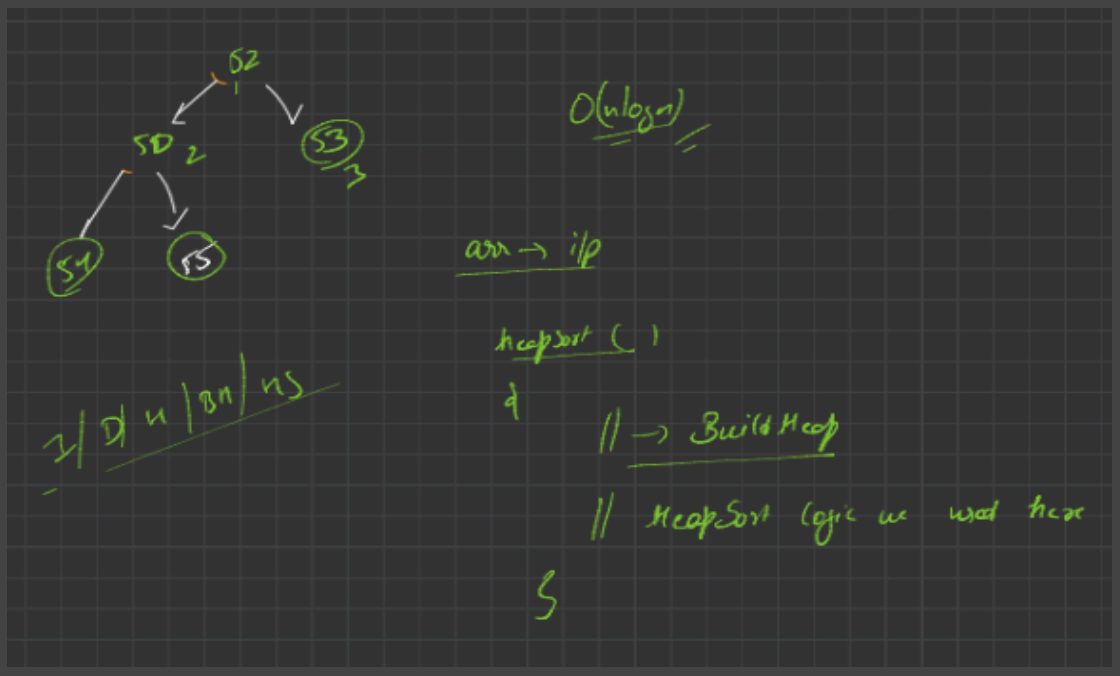


Build Min Heap:

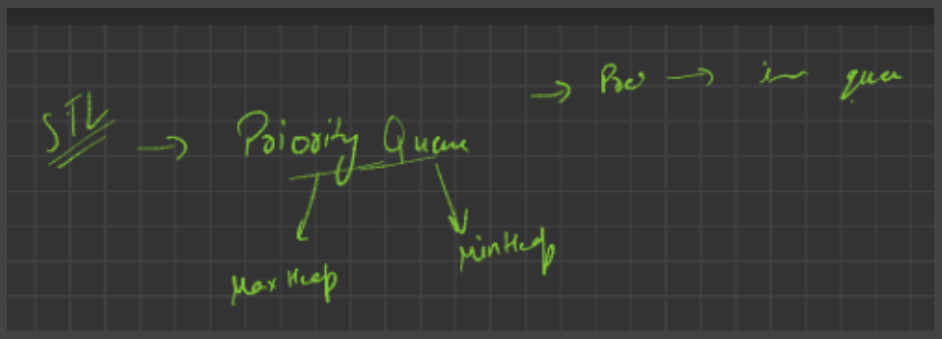


Heap Sort:

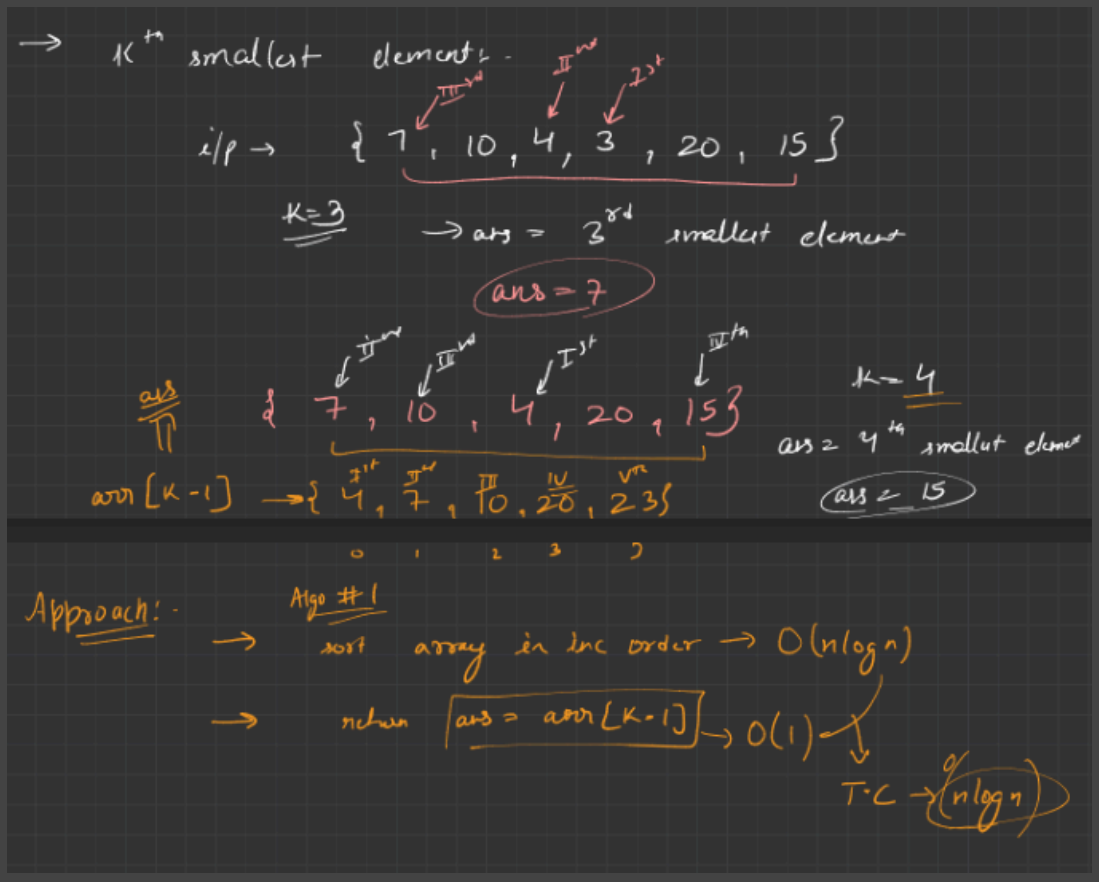




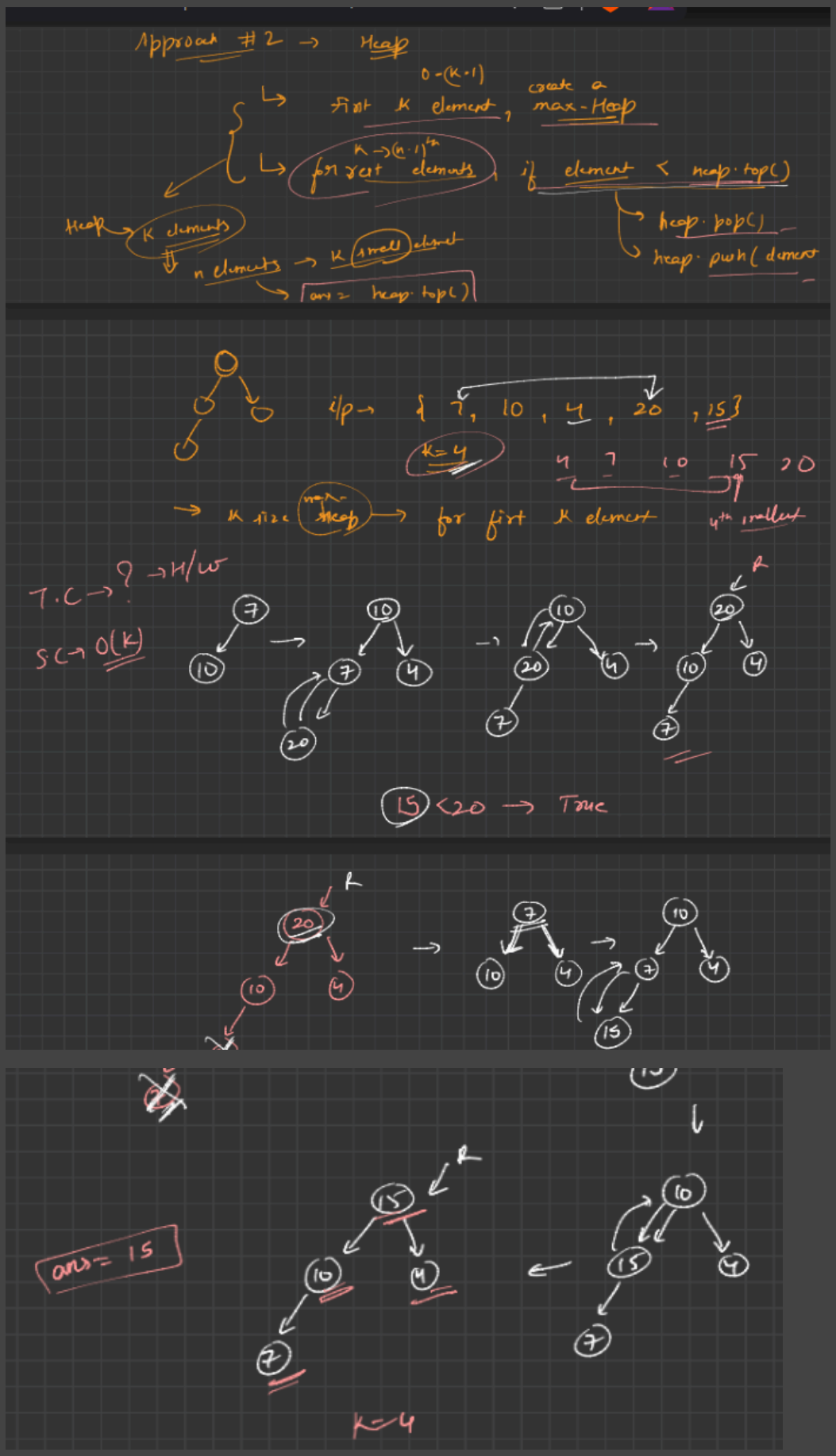
Priority Queue:



Kth Smallest Element Approach-I:



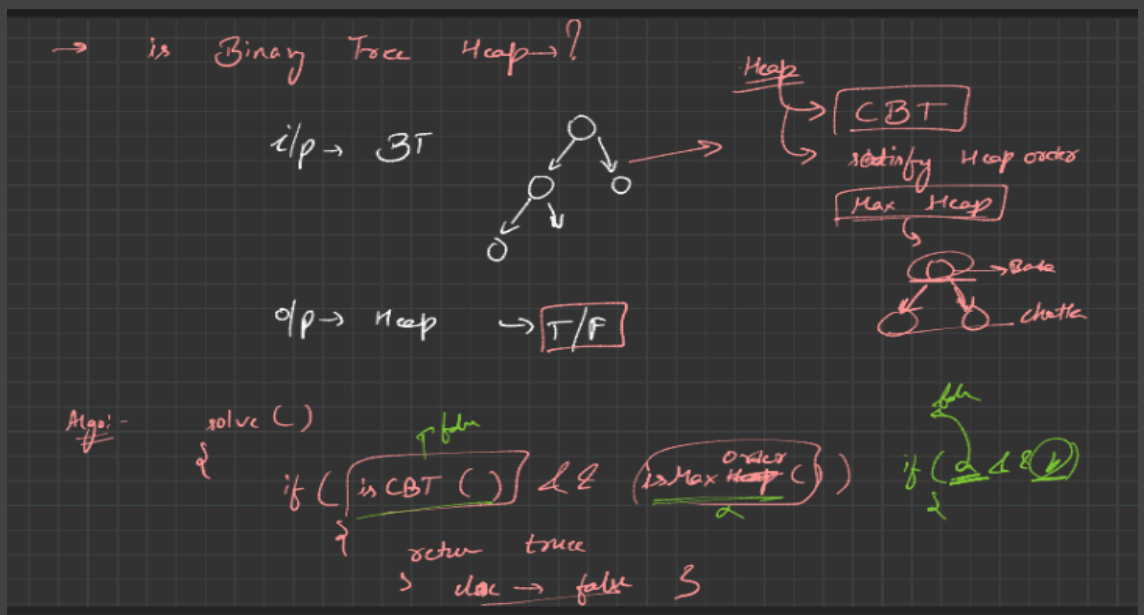
Kth Smallest Element Approach-II:

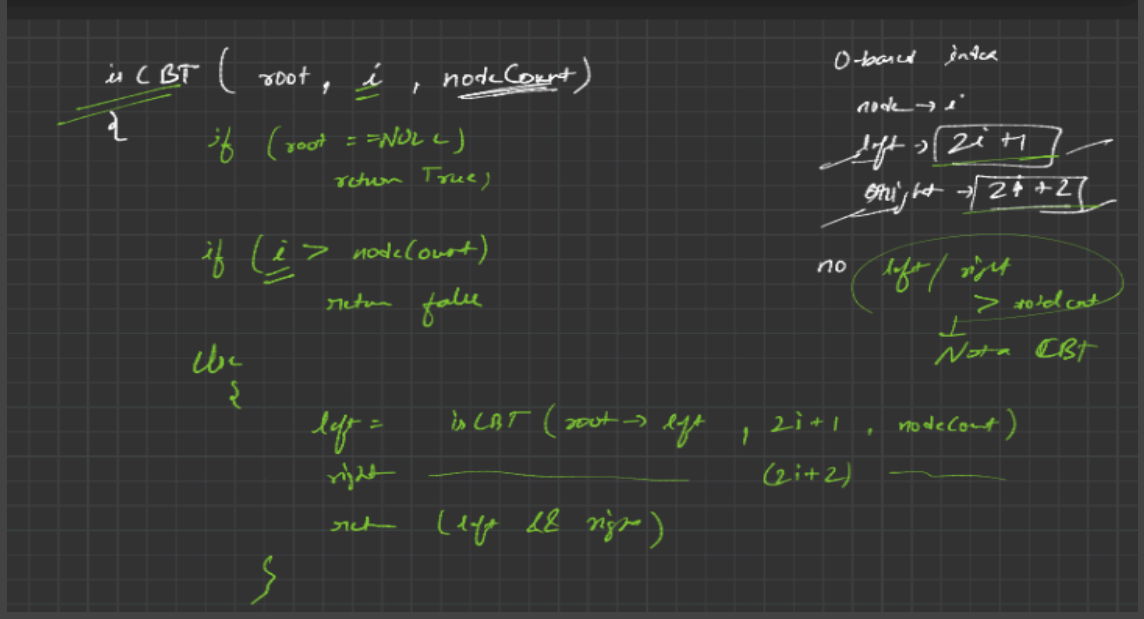


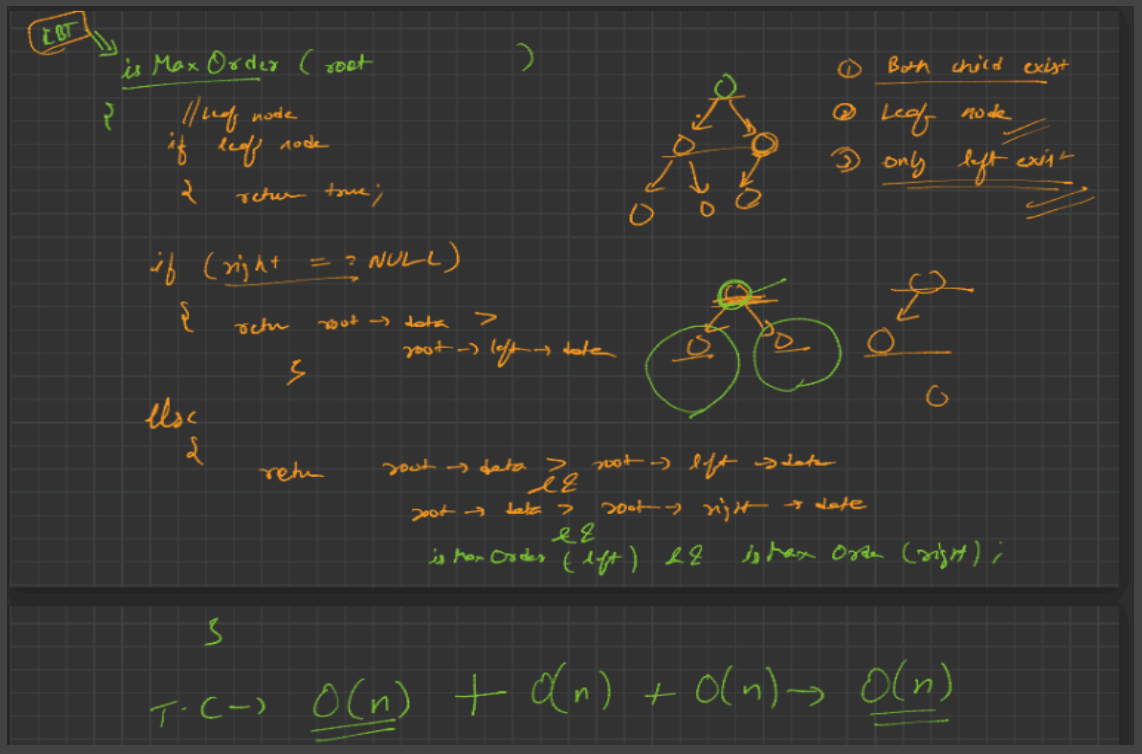
Kth Largest Element:

Use a min heap instead of a max heap to find the Kth largest element in an array.

Is The Given Binary Tree A Heap:

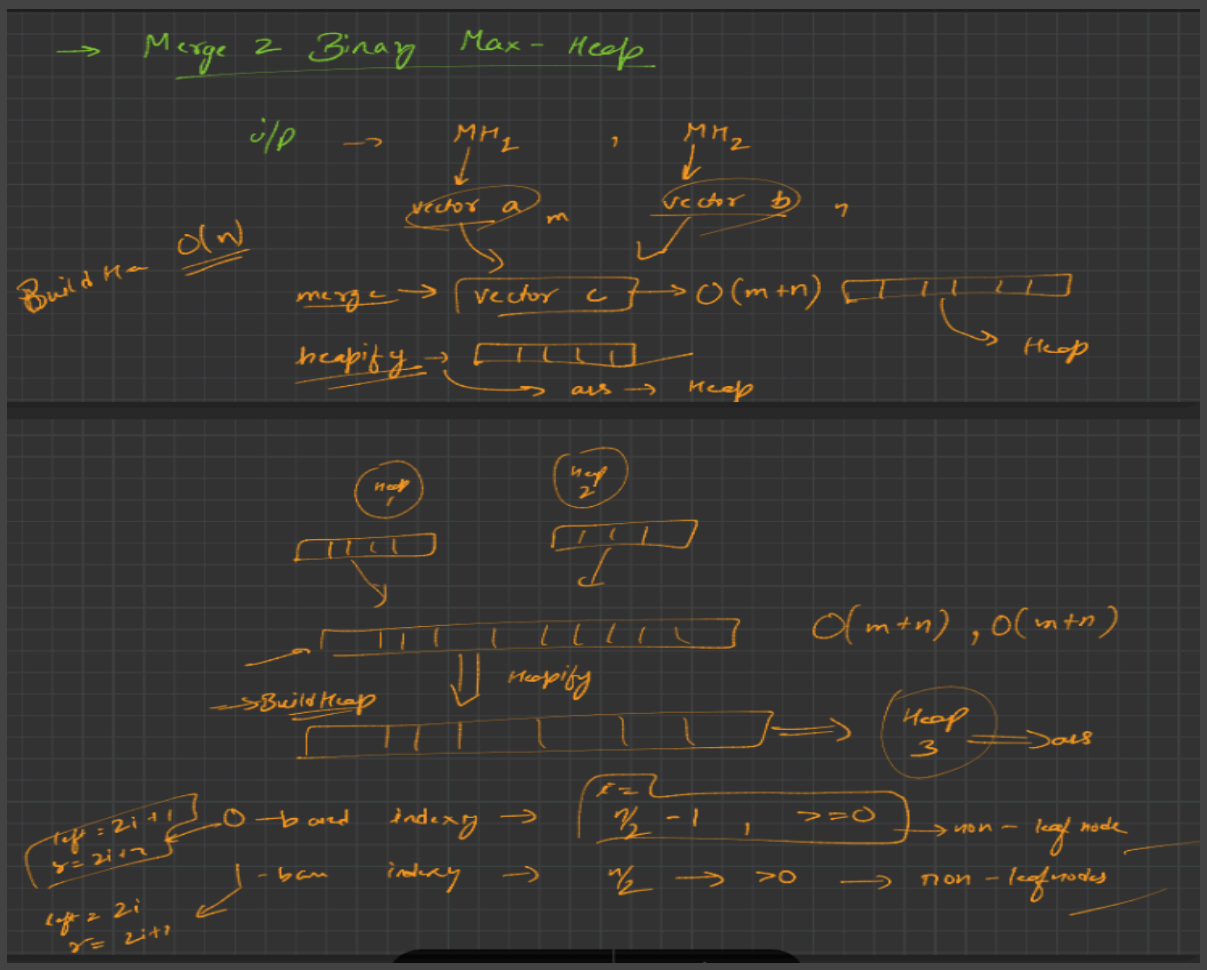




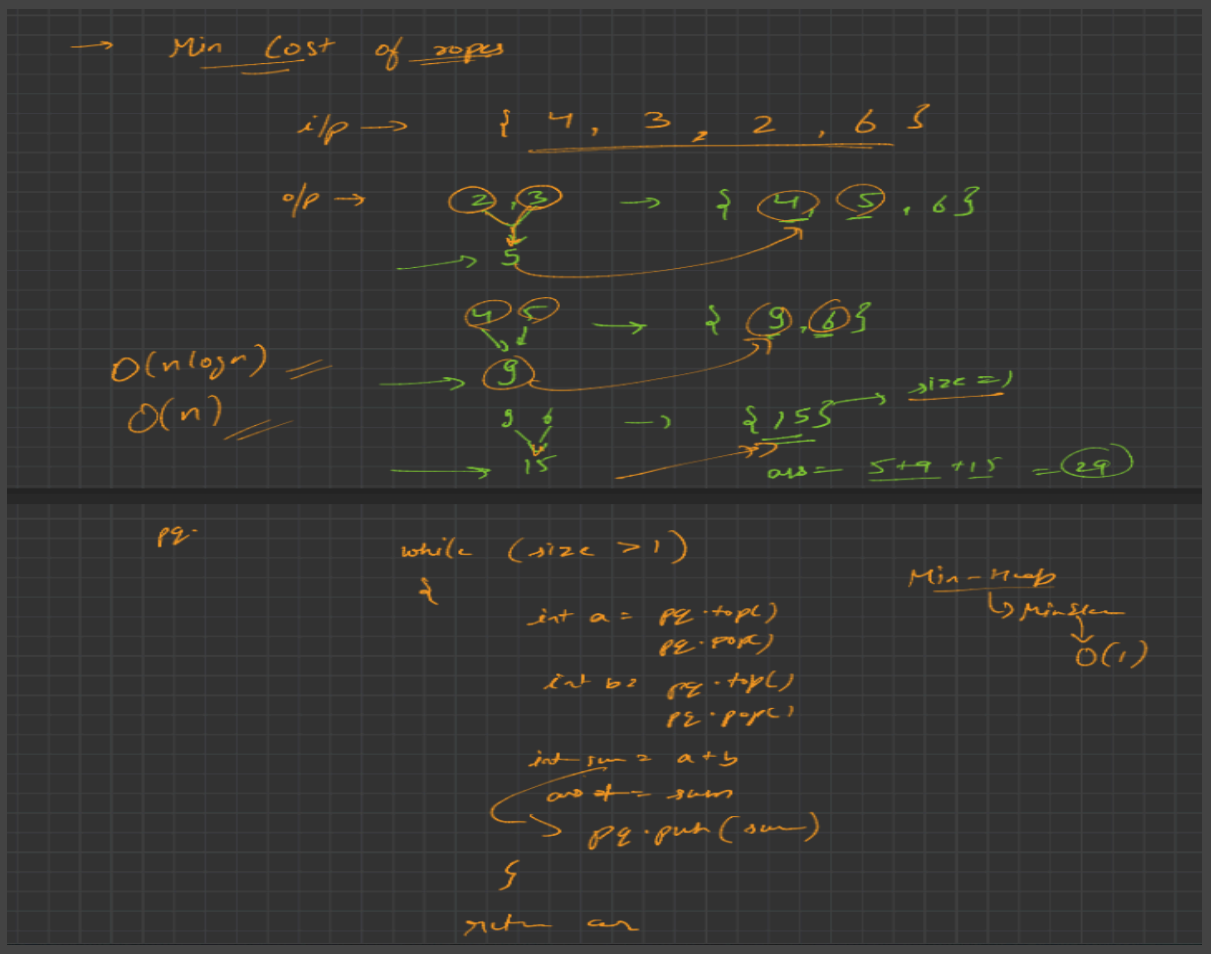


T.C → O(n) + O(n) + O(n) → O(n)

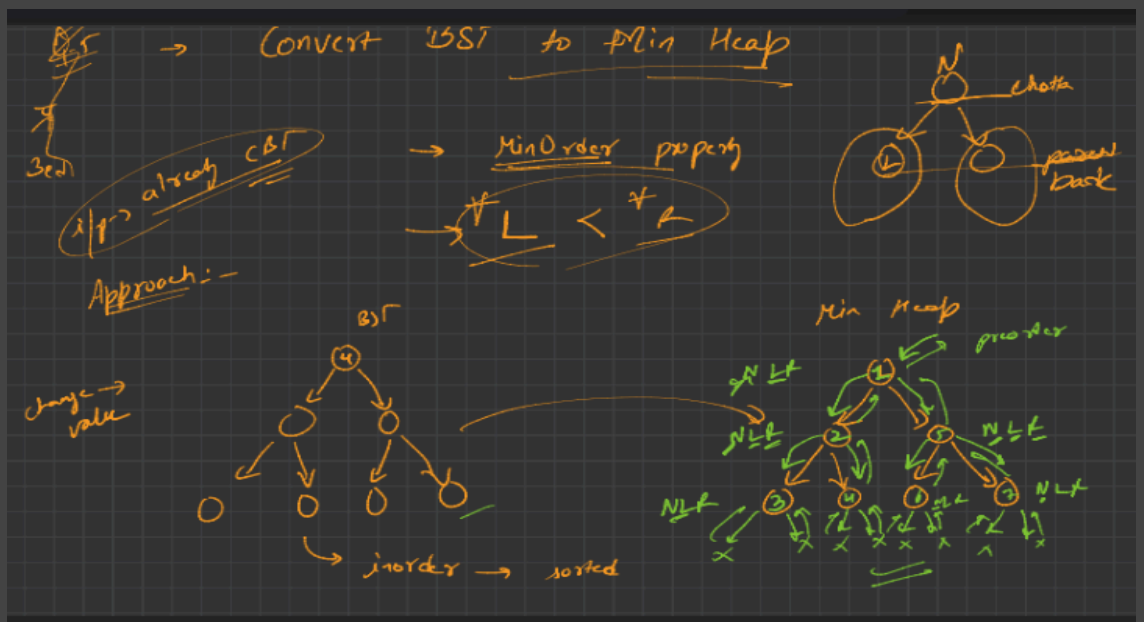
Merge Two Binary Max Heaps:

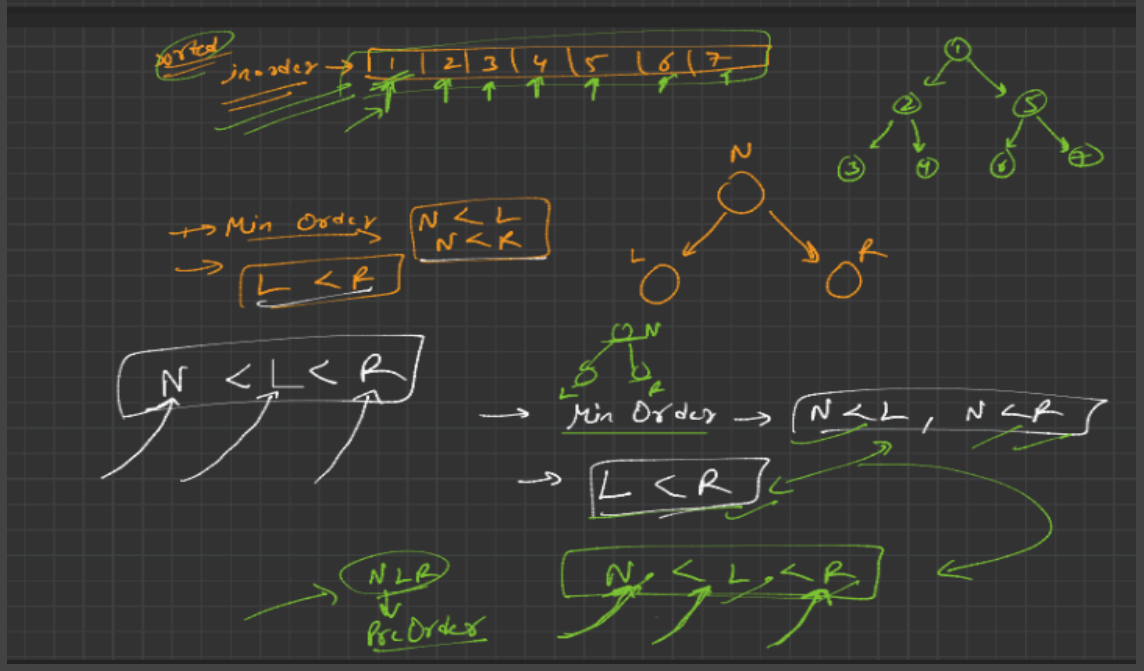


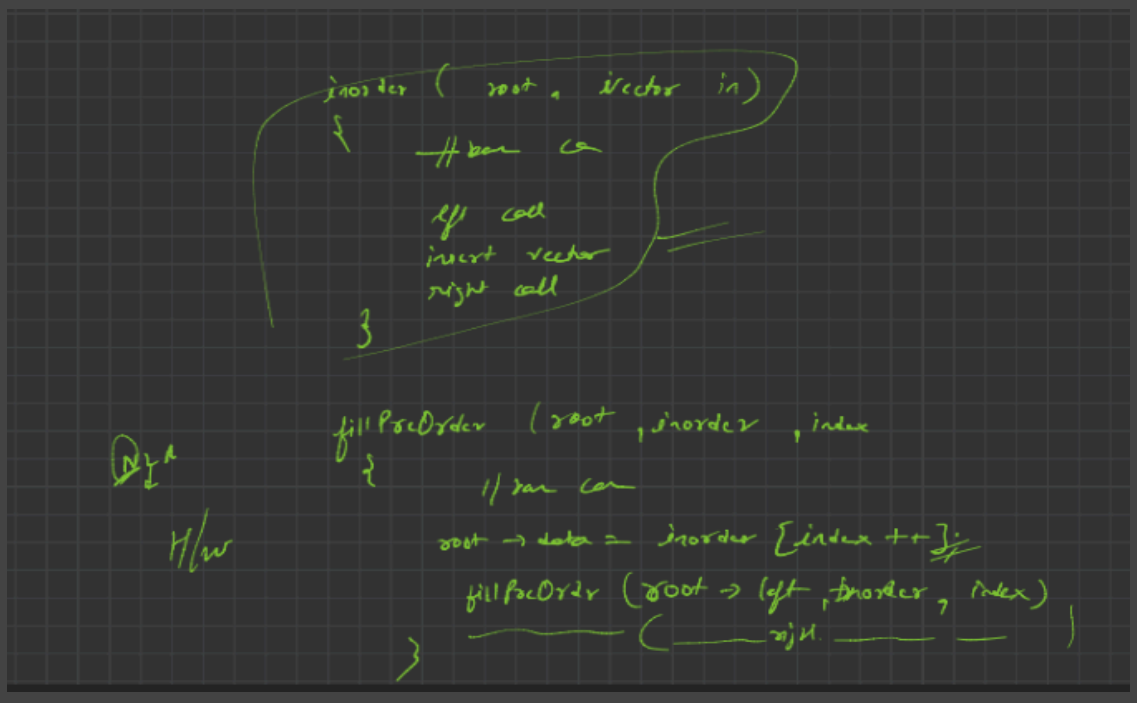
Minimum Cost Of Ropes:



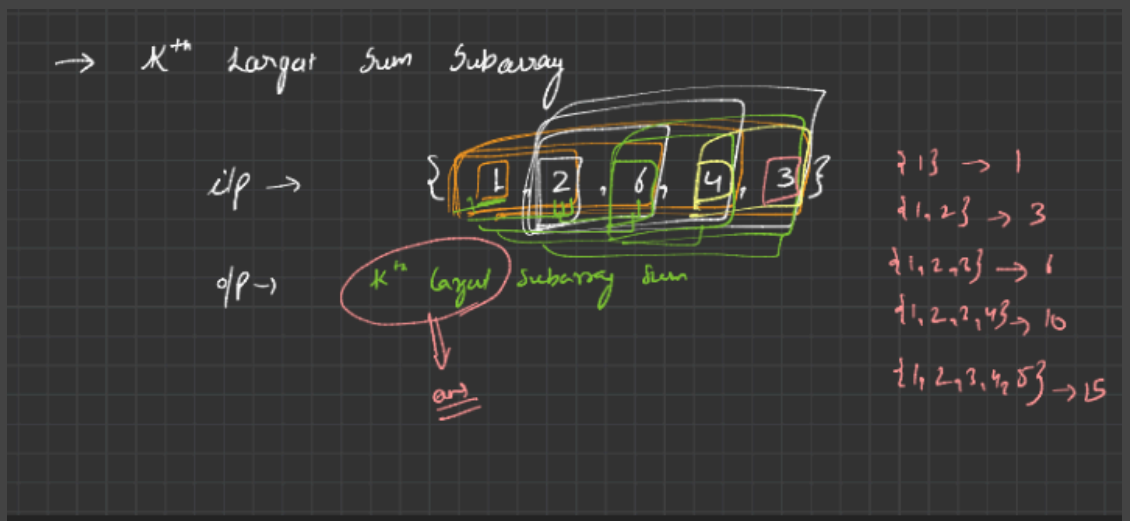
Convert BST To MinHeap:

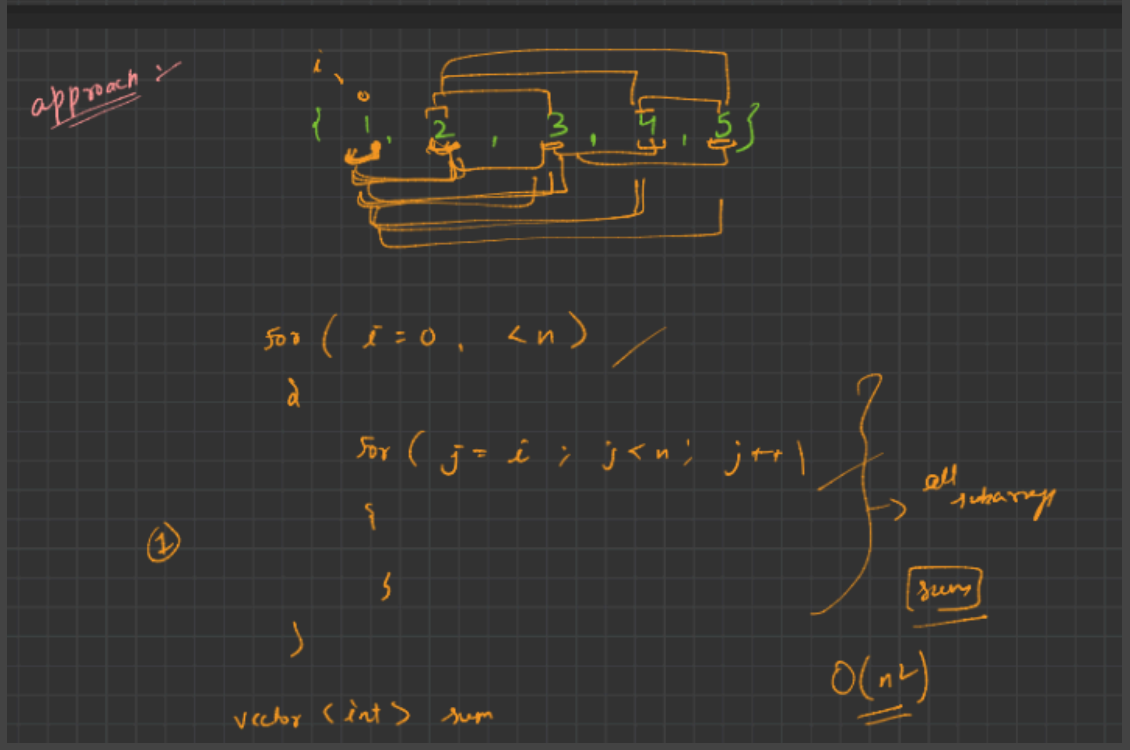


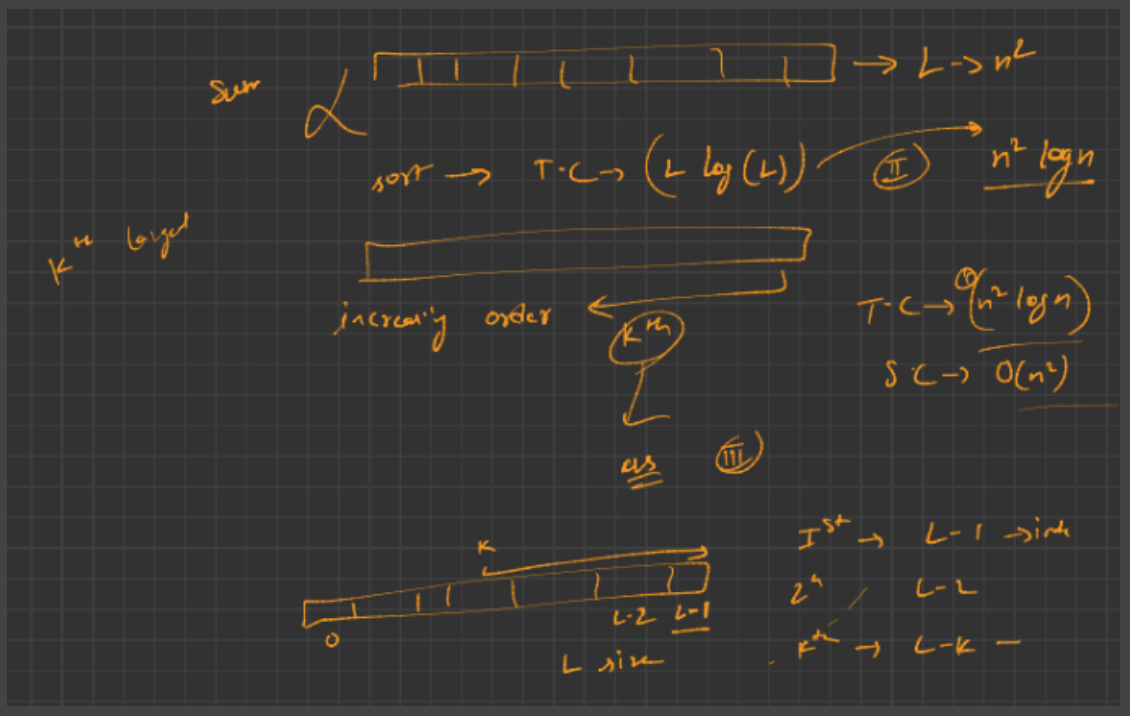




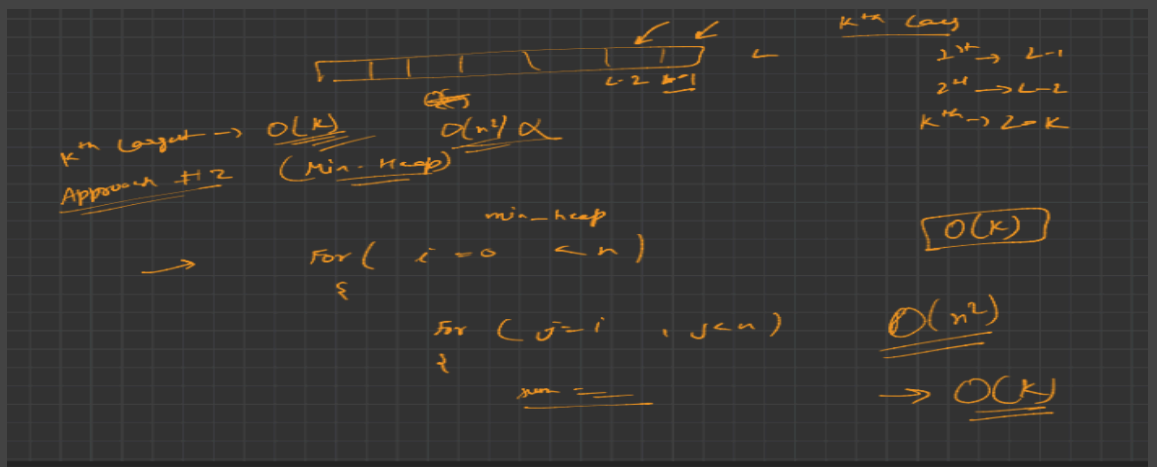
Kth Largest Sum SubArray Approach-I:

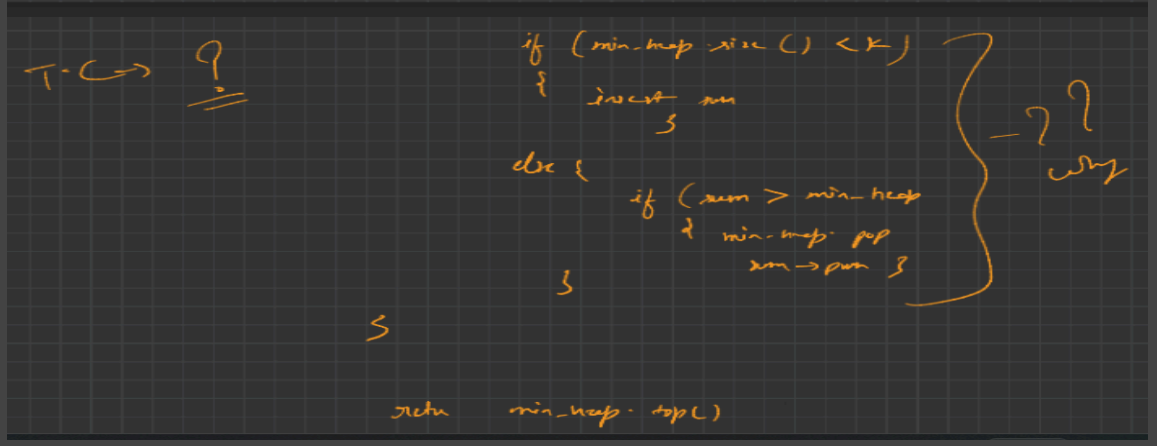




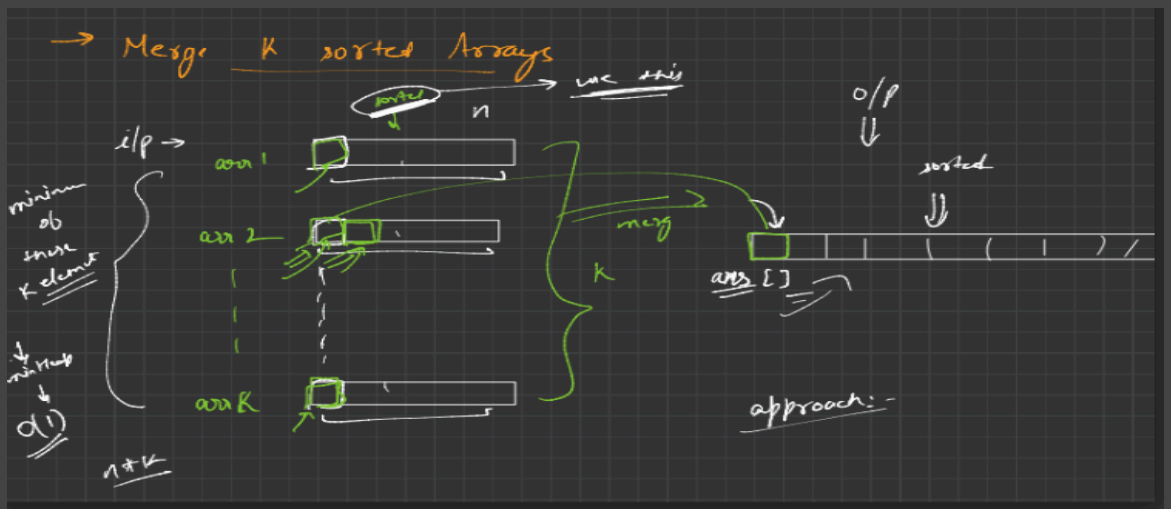


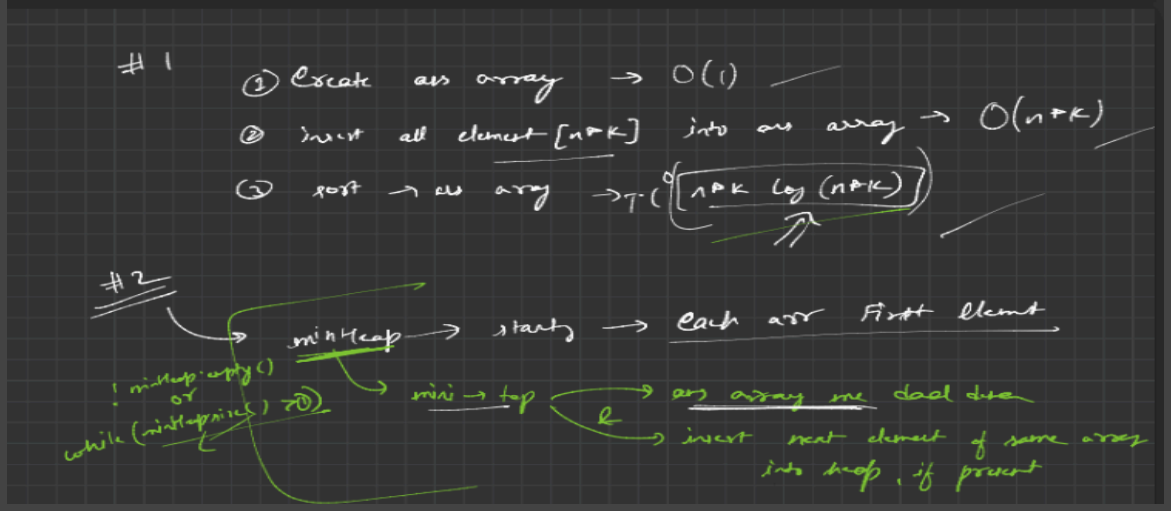
Kth Largest Sum SubArray Approach-II:

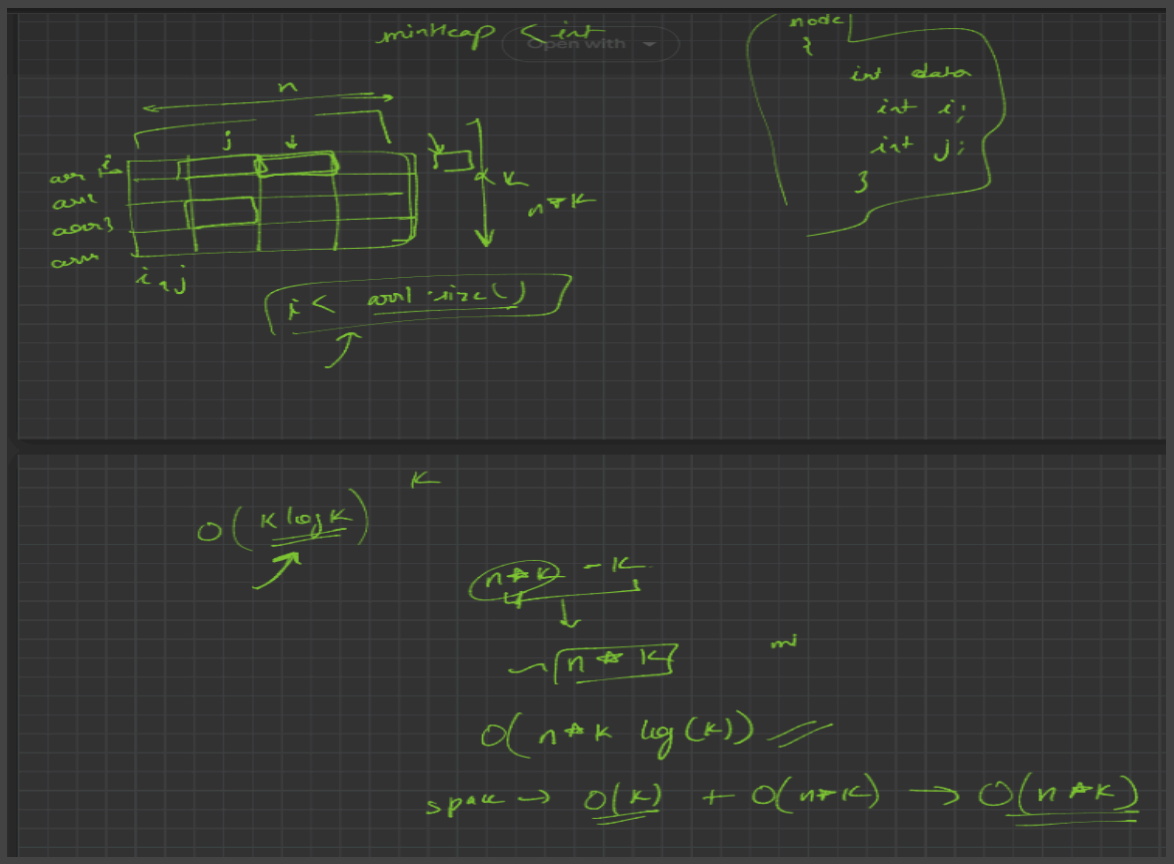




Merge K Sorted Arrays:



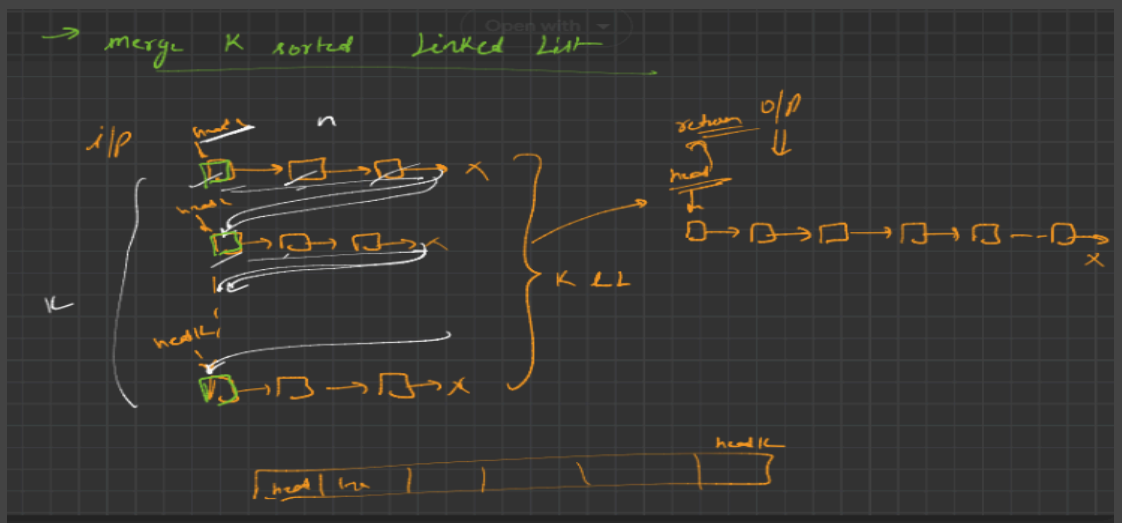


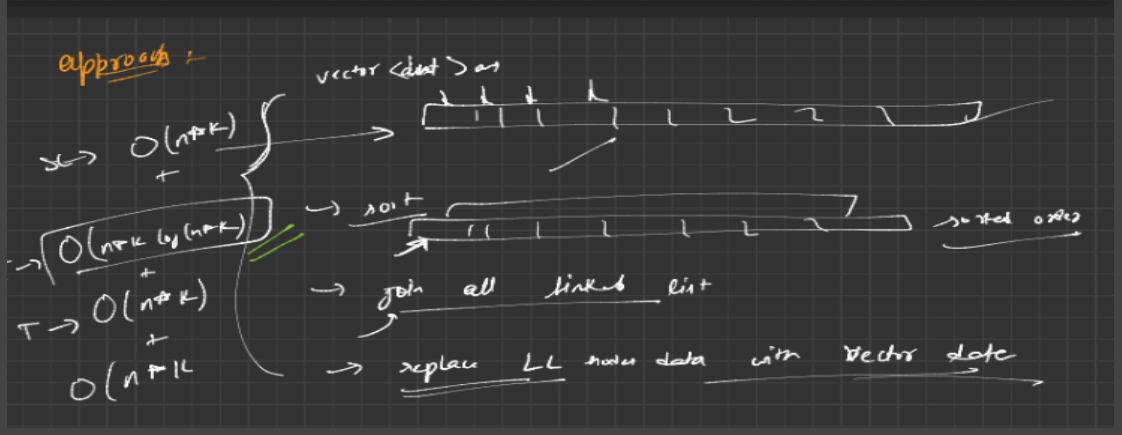


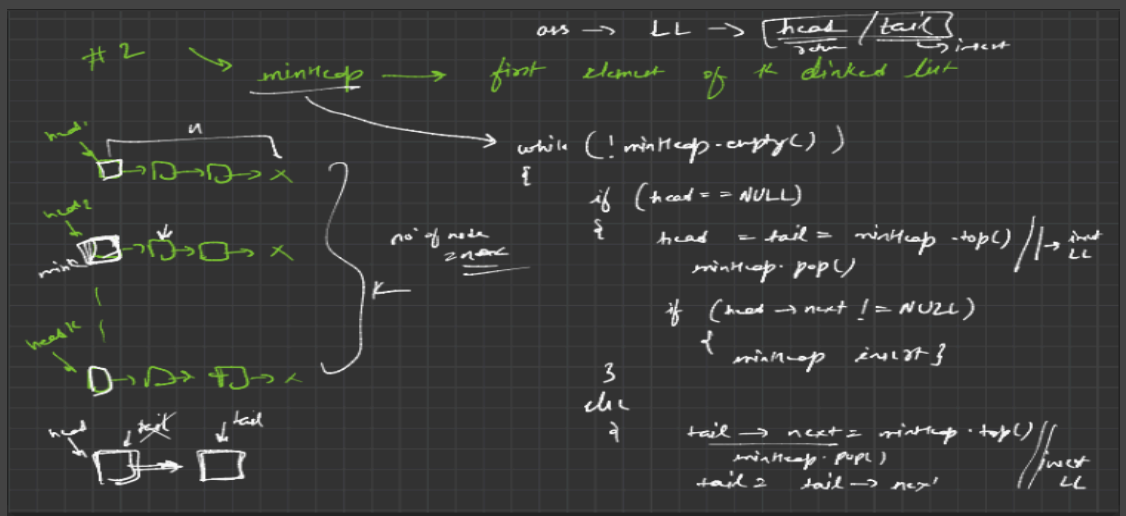
Dry Run:

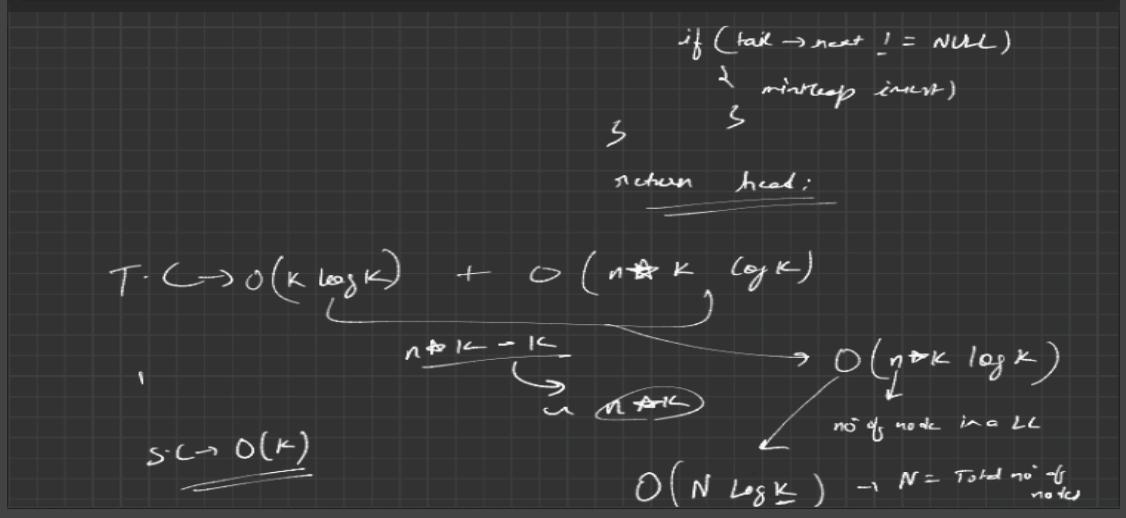
1. A min heap `minHeap` is created, which stores pointers to `node` objects. The heap is ordered by the `data` member of the nodes, in ascending order.
2. The first element of each array in `kArrays` is inserted into `minHeap`. After this step, `minHeap` contains the nodes `(2, 0, 0)`, `(1, 1, 0)`, and `(3, 2, 0)`.
3. The `while` loop begins. In each iteration, the node at the top of `minHeap` is removed, its `data` is added to the `ans` vector, and if there is a next element in the same array, a new node is created for that element and pushed into `minHeap`.
    - Iteration 1: The top node is `(1, 1, 0)`. 1 is added to `ans`, and the node is removed from `minHeap`. The next element in the same array is 3, so the node `(3, 1, 1)` is pushed into `minHeap`.
    - Iteration 2: The top node is `(2, 0, 0)`. 2 is added to `ans`, and the node is removed from `minHeap`. The next element in the same array is 6, so the node `(6, 0, 1)` is pushed into `minHeap`.
    - Iteration 3: The top node is `(3, 1, 1)`. 3 is added to `ans`, and the node is removed from `minHeap`. The next element in the same array is 14, so the node `(14, 1, 2)` is pushed into `minHeap`.
    - Iteration 4: The top node is `(3, 2, 0)`. 3 is added to `ans`, and the node is removed from `minHeap`. The next element in the same array is 5, so the node `(5, 2, 1)` is pushed into `minHeap`.
    - Iteration 5: The top node is `(5, 2, 1)`. 5 is added to `ans`, and the node is removed from `minHeap`. The next element in the same array is 8, so the node `(8, 2, 2)` is pushed into `minHeap`.
    - Iteration 6: The top node is `(6, 0, 1)`. 6 is added to `ans`, and the node is removed from `minHeap`. The next element in the same array is 12, so the node `(12, 0, 2)` is pushed into `minHeap`.
    - Iteration 7: The top node is `(8, 2, 2)`. 8 is added to `ans`, and the node is removed from `minHeap`. The next element in the same array is 10, so the node `(10, 2, 3)` is pushed into `minHeap`.
    - Iteration 8: The top node is `(10, 2, 3)`. 10 is added to `ans`, and the node is removed from `minHeap`. There is no next element in the same array.
    - Iteration 9: The top node is `(12, 0, 2)`. 12 is added to `ans`, and the node is removed from `minHeap`. The next element in the same array is 15, so the node `(15, 0, 3)` is pushed into `minHeap`.
    - Iteration 10: The top node is `(14, 1, 2)`. 14 is added to `ans`, and the node is removed from `minHeap`. The next element in the same array is 20, so the node `(20, 1, 3)` is pushed into `minHeap`.
    - Iteration 11: The top node is `(15, 0, 3)`. 15 is added to `ans`, and the node is removed from `minHeap`. There is no next element in the same array.
    - Iteration 12: The top node is `(20, 1, 3)`. 20 is added to `ans`, and the node is removed from `minHeap`. There is no next element in the same array.
4. The `while` loop ends because `minHeap` is empty. The `ans` vector is returned.

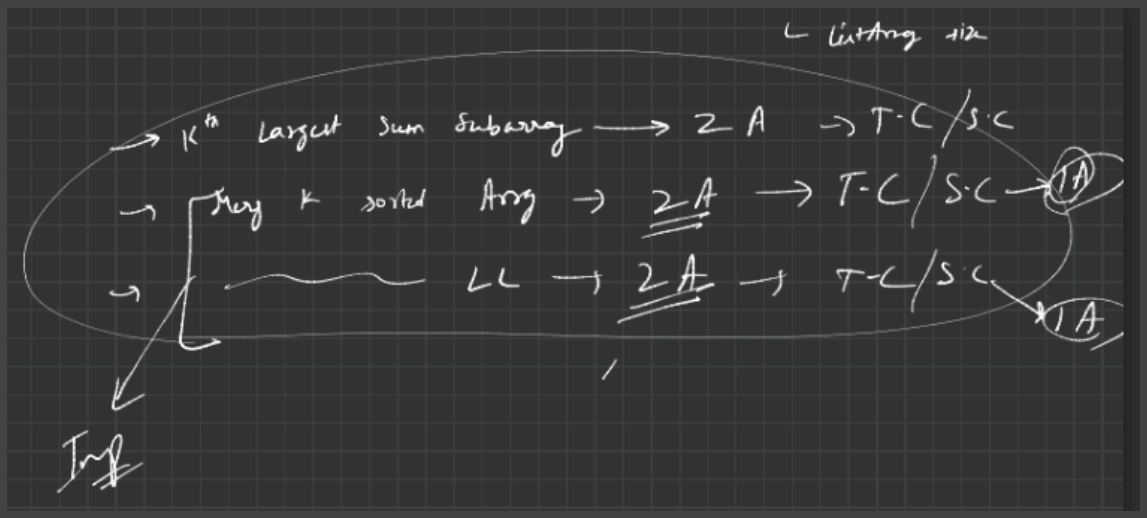
Merge K Sorted Linked List:



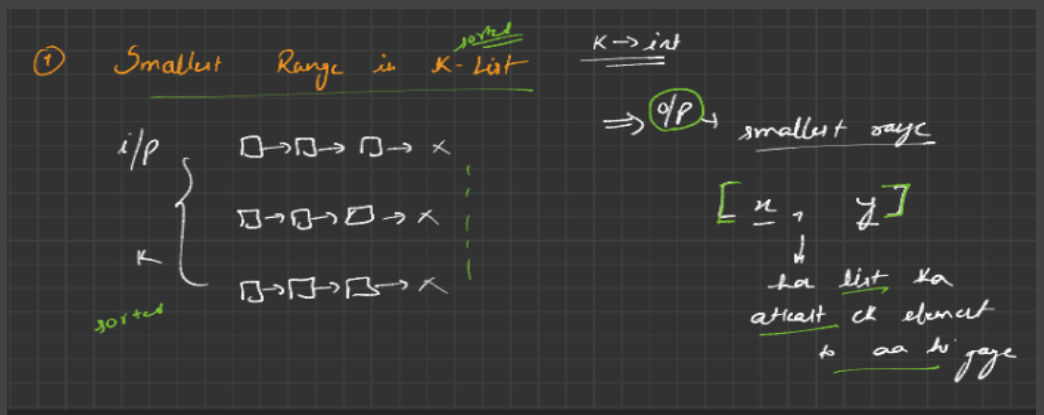


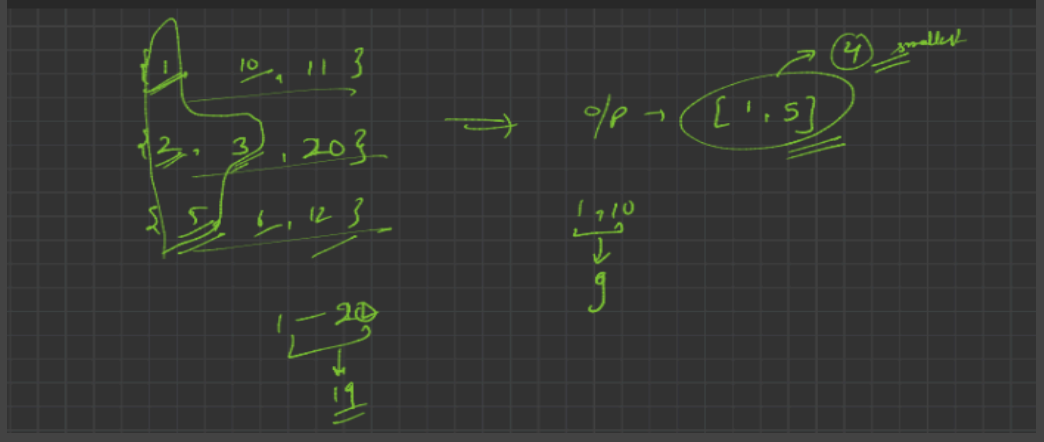


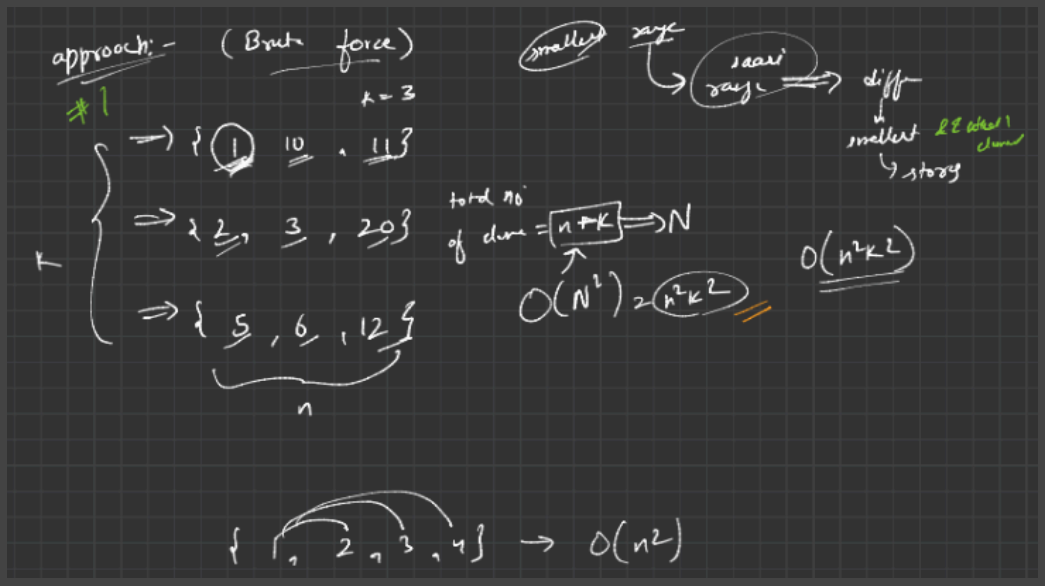


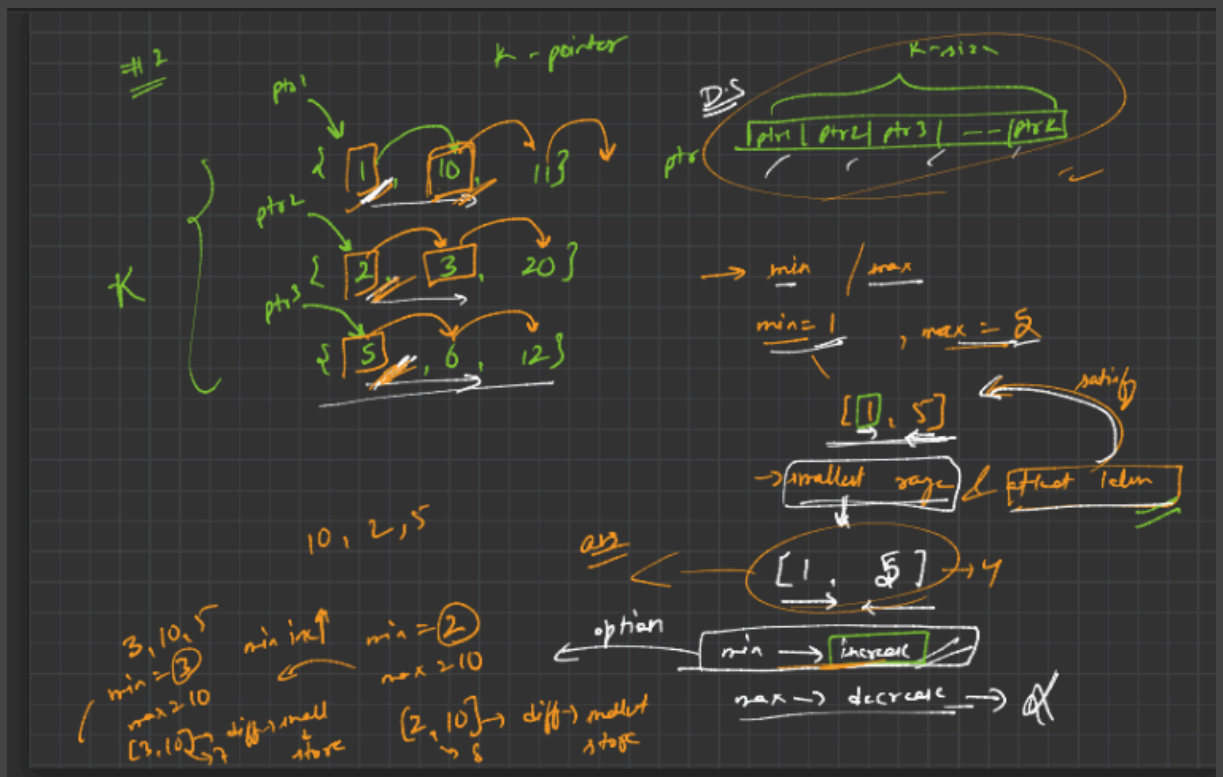


Smallest Range From K Sorted List:



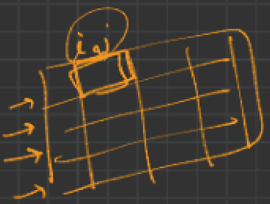





#2
k - pointer
ptr1
ptr2
ptr3
K
1, 10, 11
2, 3, 20
5, 6, 12
DS
K-size
ptr1 | ptr2 | ptr3 | --- | ptrK
ptr
→ min / max
min = 1 , max = 5
[1, 5]
satisfy
→ smallest range & efficient return
[1, 5] → 4
ans
option
min → increase
max → decrease → ✗
10, 2, 5
3, 10, 5
min = 3
max = 10
[3, 10] → 7
diff → small → store
min inc
min = 2
max = 10
[2, 10] → 8
diff → small → store
20, 10, 5 → min = 5 max = 20 → 15 → [5, 20] → 15
diff
small
ans update
6, 10, 20 → [6, 20] → 14
12, 10, 20 → [10, 20] → 10
12, 11, 20 → [11, 20] → 9
while ( ) → (n*K) → how element ✗
min
max
→ O(K)
log K
✗
T.C = n*K*K*log K → O(n*K*K)
S.C → O(K)

2 Heaps

i/p → { [1], 10, 11 }
→ { [2], 3, 20 }
→ { [5], 6, 12 }



min

Algo:
0 → min / max →
1 → min-Heap → k size
↳ list → starting element
2 → daalte hue
↳ aapne max track krenge

int ansStart, ansEnd;
range = ansEnd - ansStart

```
while (!pq.empty())
{
    min = pq.top()      → element, row, col
    pq.pop();
```

ans update →

```
    if (max - min < ansEnd - ansStart)
    {
        update ans
    }
```

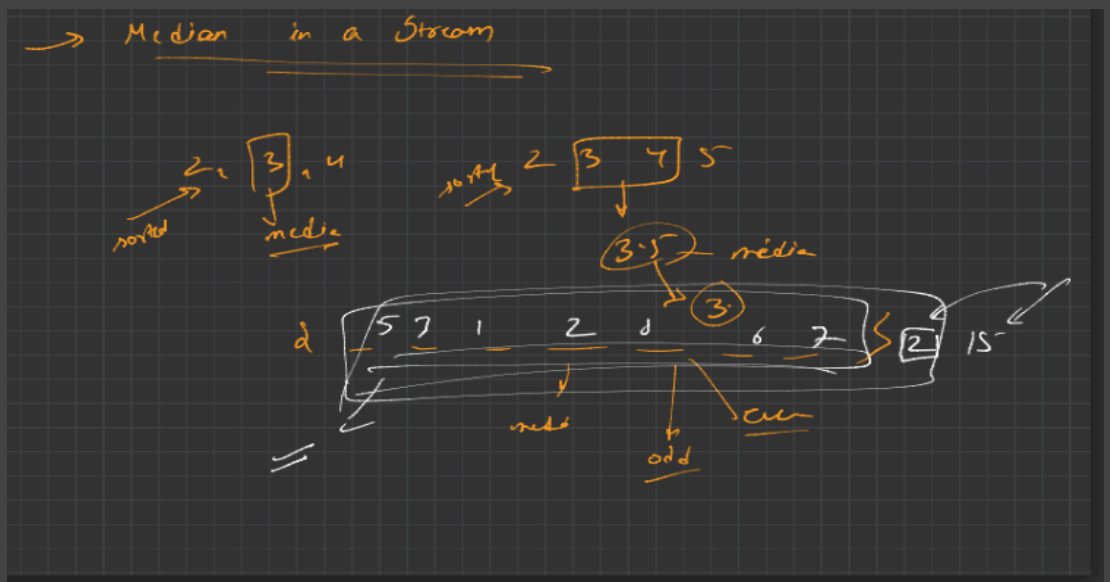
max - update →

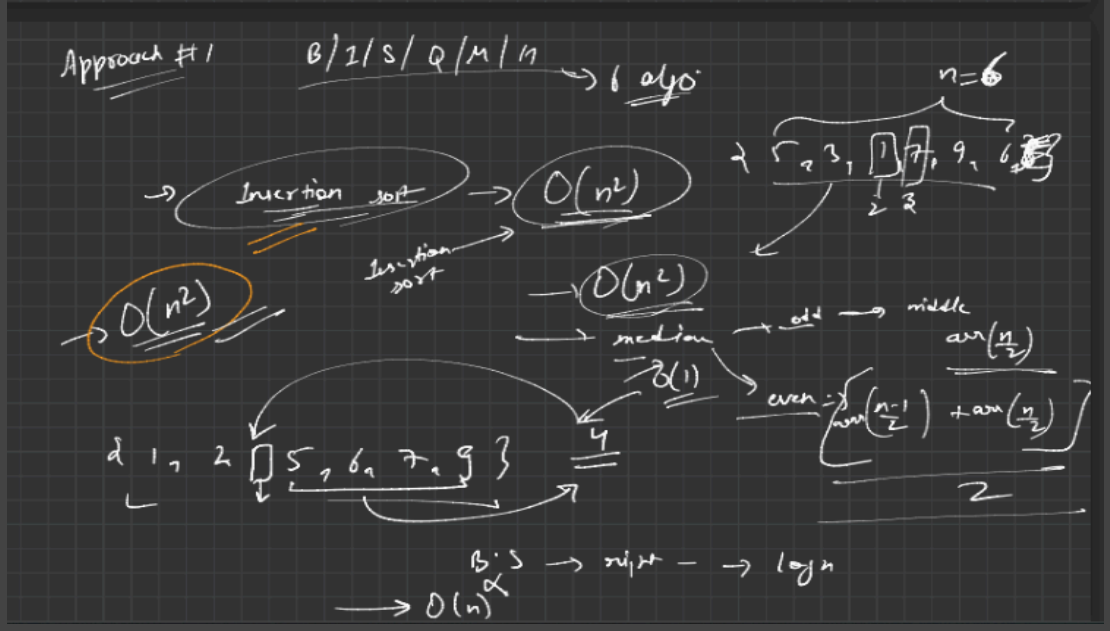
```
    if (min.col < n)      // next element exist
    {
        maxi = max(maxi, arr[min.row][min.col+1]
        pq.push(new node( ____
                          min row
                          ____ ))
    }
    else
    {
        break;
    }
```

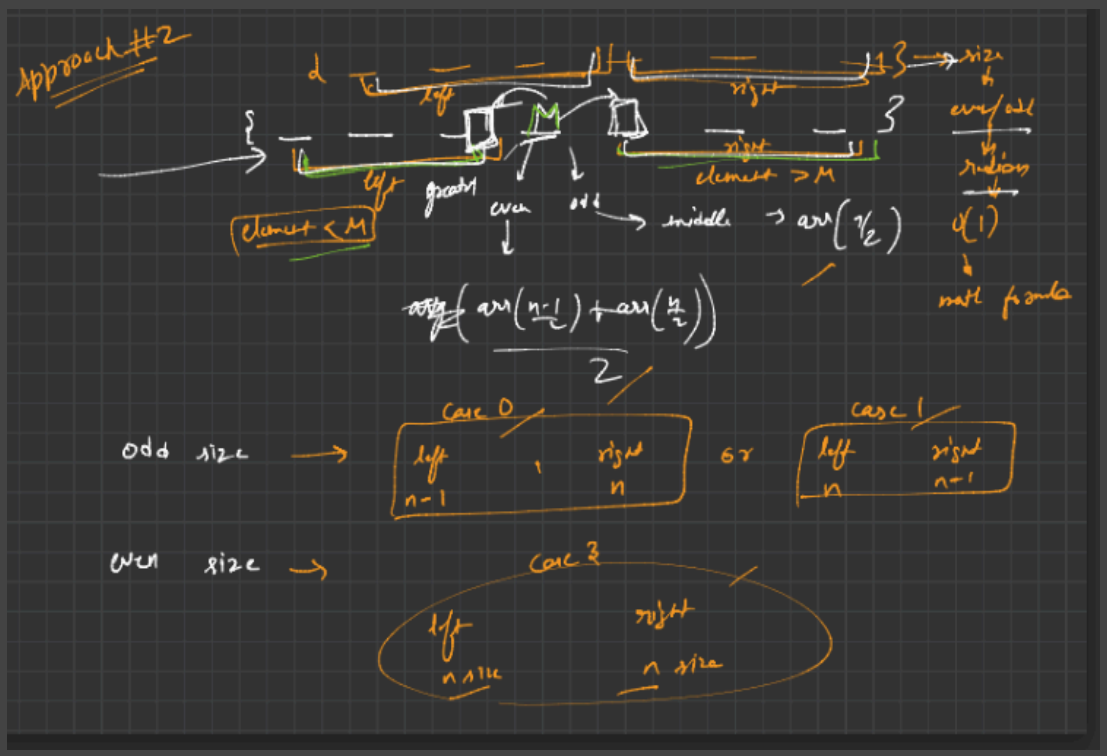
(5, 5) → 1
5 - 5 + 1 = 1

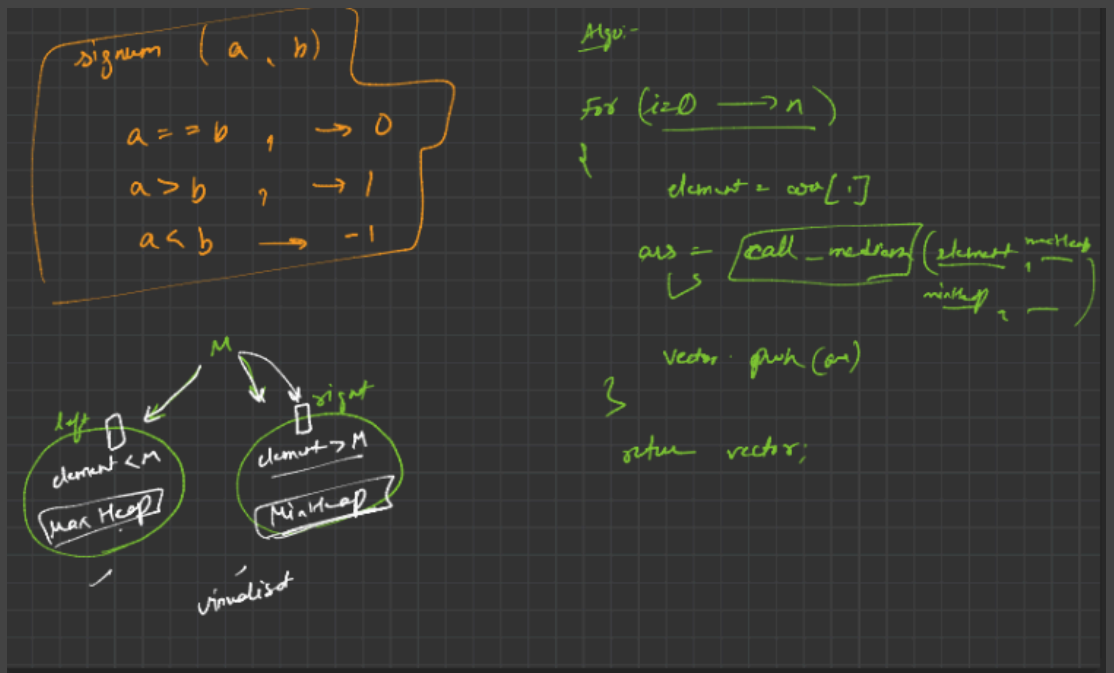
[3 - 9] → 9 - 3 + 1
7 7

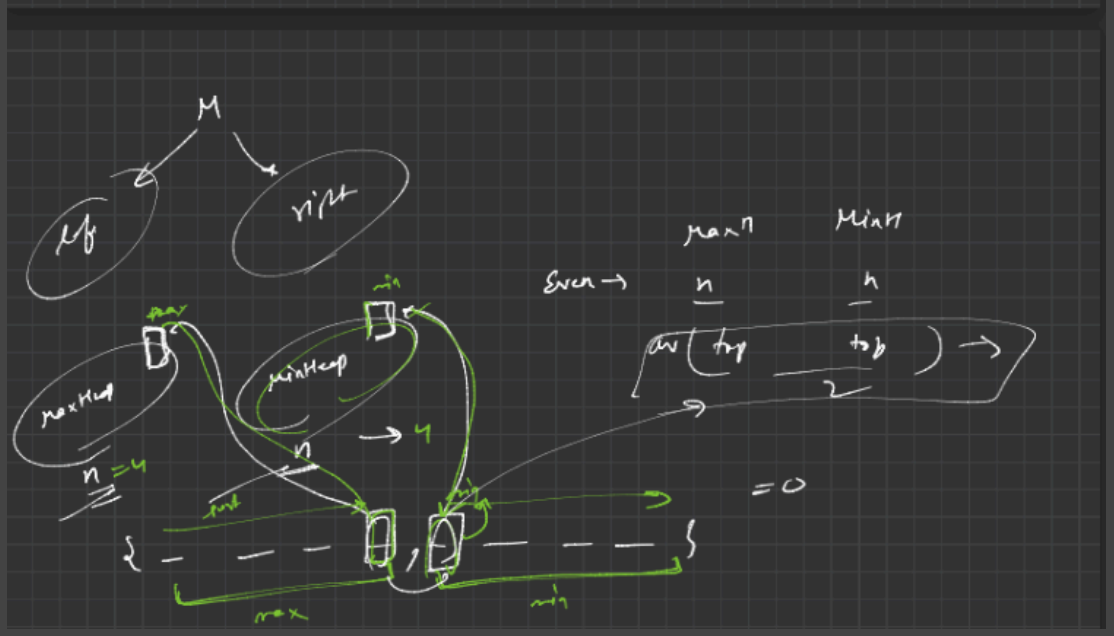
Median In A Stream:

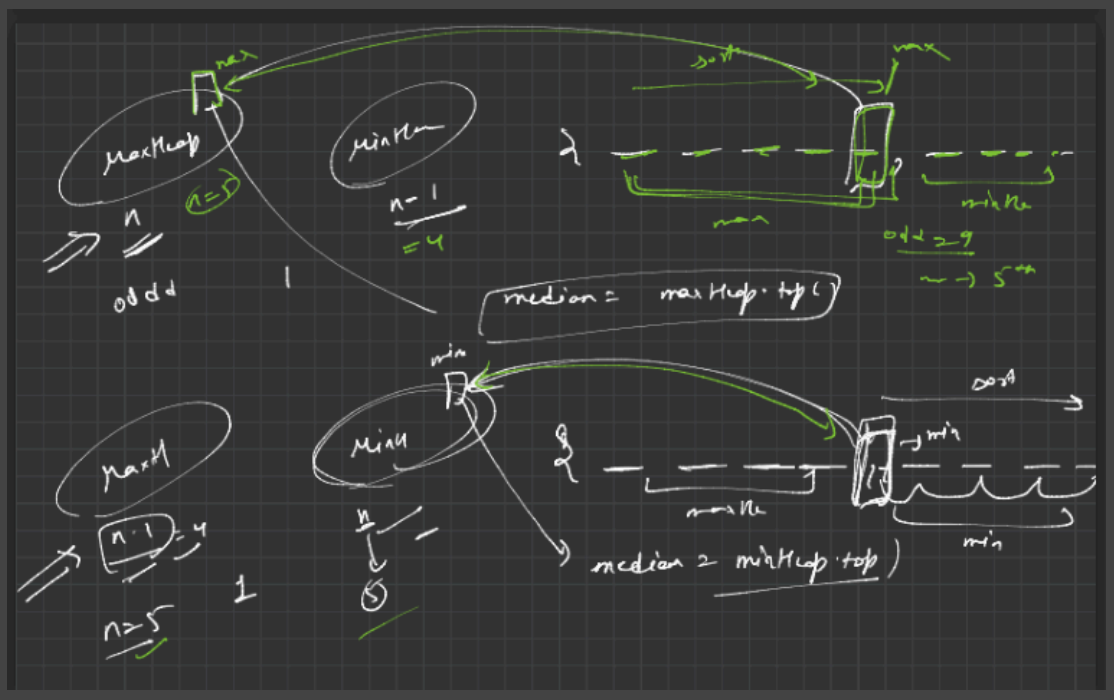







signum (a , b)
a == b , → 0
a > b , → 1
a < b → -1
Algo:-
for (i=0 → n)
{
element = arr[i]
ans = call_median (element, maxHeap, minHeap)
vector . push (ans)
}
return vector;
M
left
right
element < M
element > M
maxHeap
minHeap
visualise


M
left
right
maxH
minH
Even →
n
h
top
top
2
n = 4
n → 4
=0
max
min


maxHeap
minHeap
n = 5
n
odd
1
n - 1
= 4
odd = 9
→ 5th
median = maxHeap . top ()
maxH
minH
n - 1 = 4
n = 5
1
median = minHeap . top)

call _median (elem, _maxH, minH, &median) by ref

{

switch (signum (maxHeap.size(), minHeap.size()))

{

case 0: if (element > M)

{ minHeap.push (element)

median = minHeap.top() }

else

{ maxHeap.push (element

median = maxHeap.top()

}

case 1: if (element > M)

{ minHeap.push (element)

median = arg { maxH.top() , minHeap.top() }

else

{

minHeap.push(maxHeap.top())

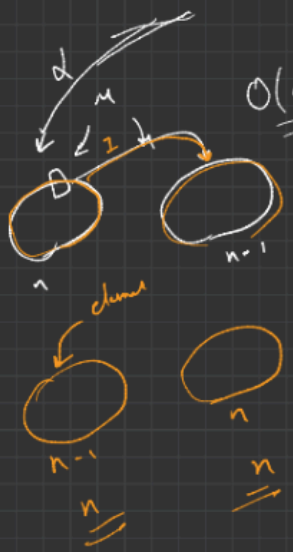
maxHeap.pop()

maxHeap.push (element

} // copy

}

O(n log n)



a < b

case -1 :

if (element > M)

{

maxHeap.push (minHeap.top)

minHeap.pop()

minHeap.push (element)

median = avg (minTop , maxTop) }

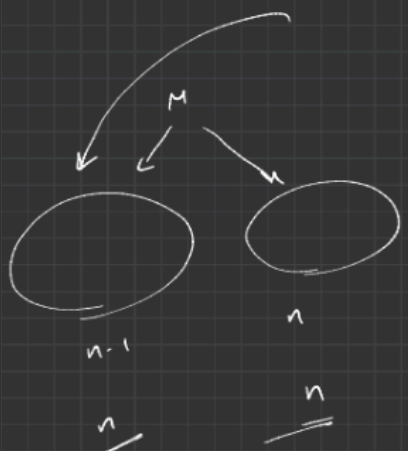
else

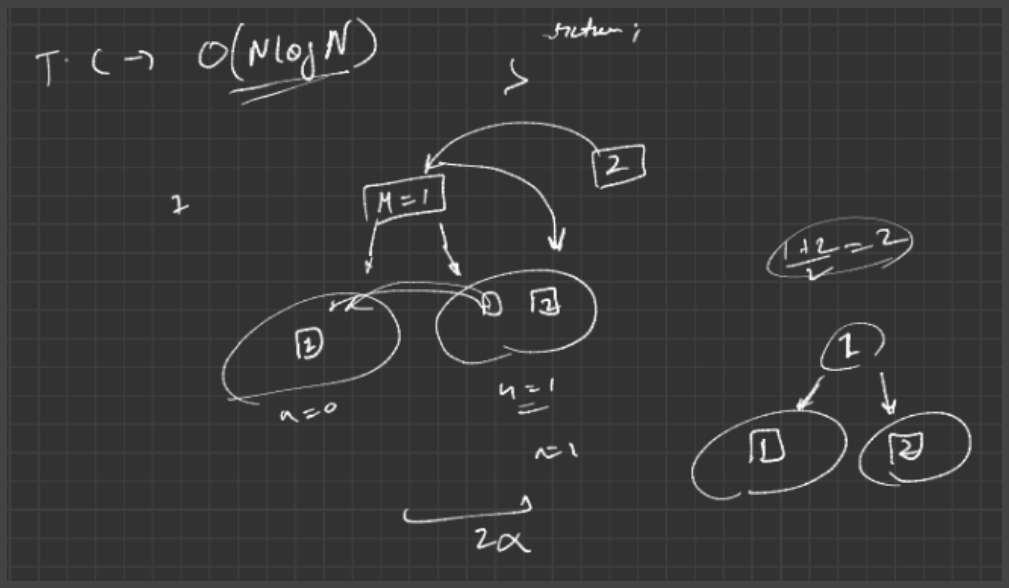
{

maxHeap.push (element)

// copy

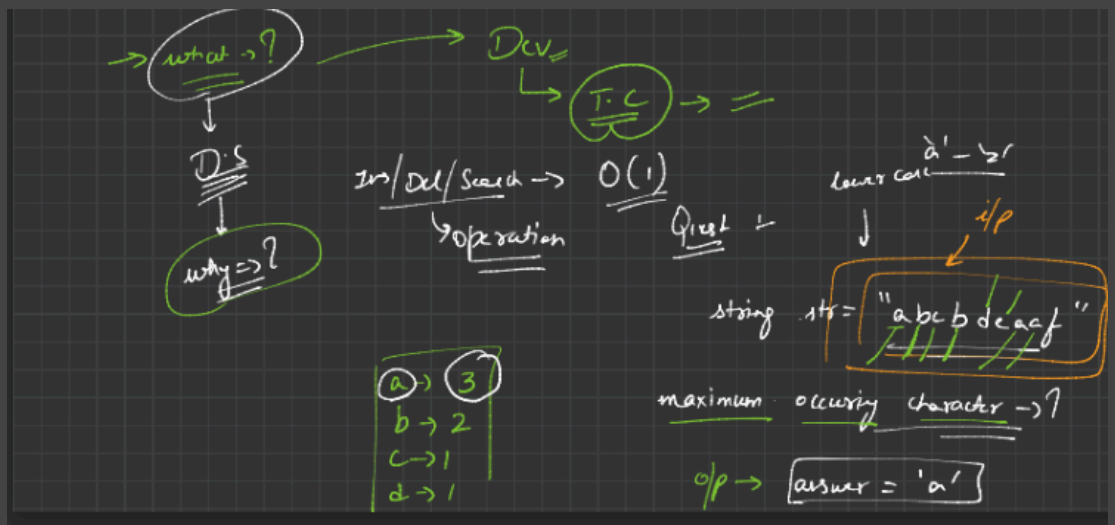
}

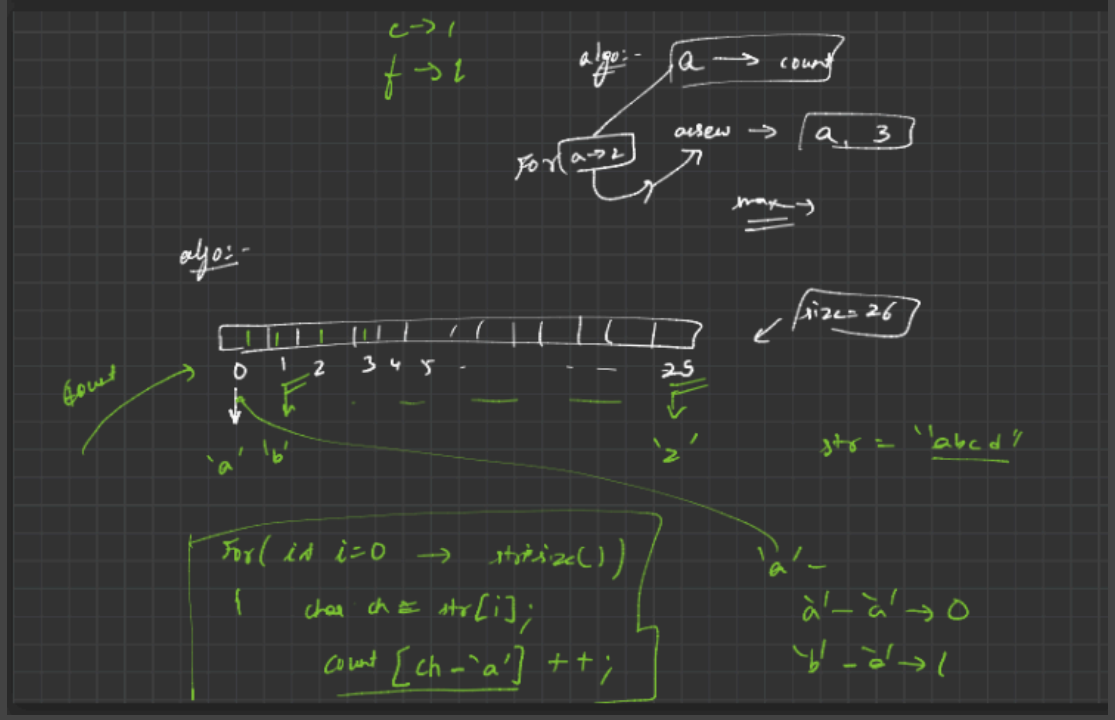


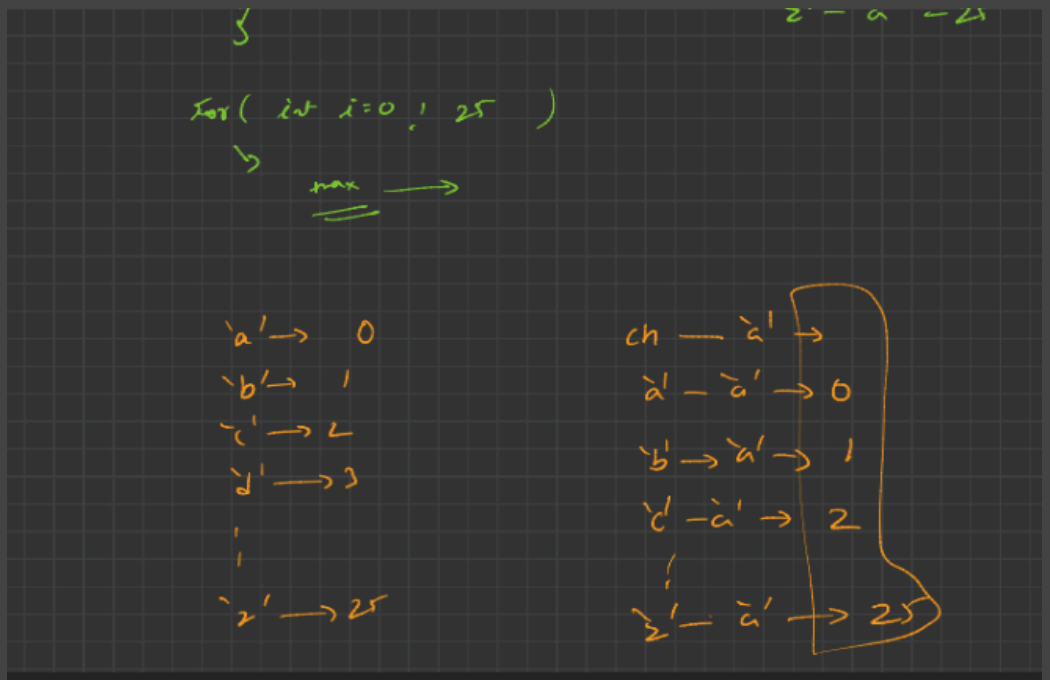


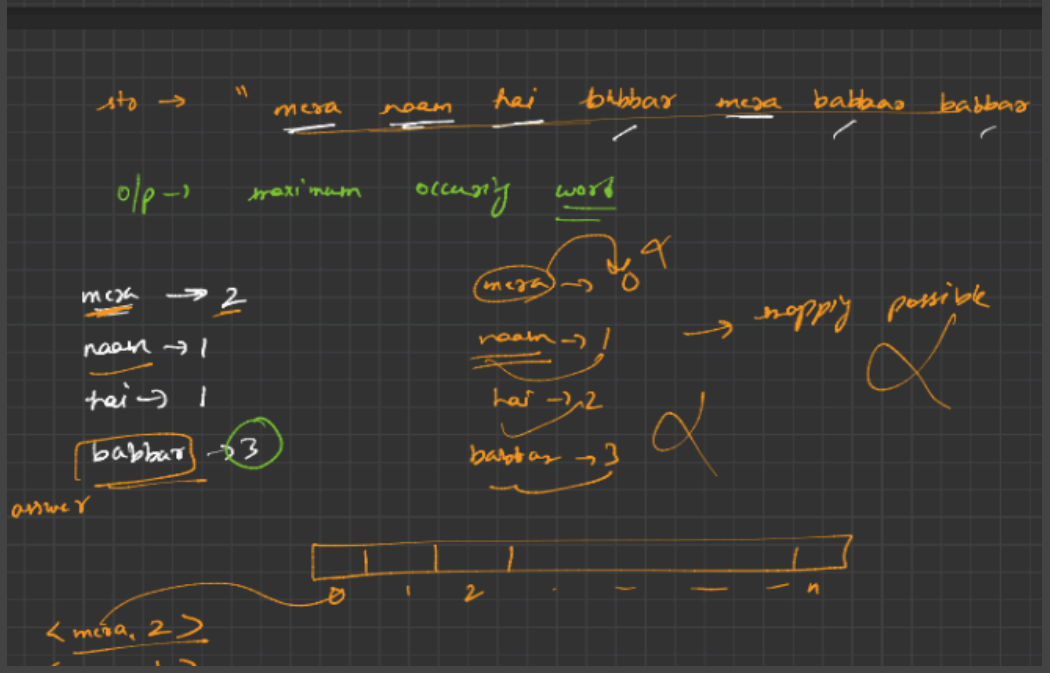
HashMaps

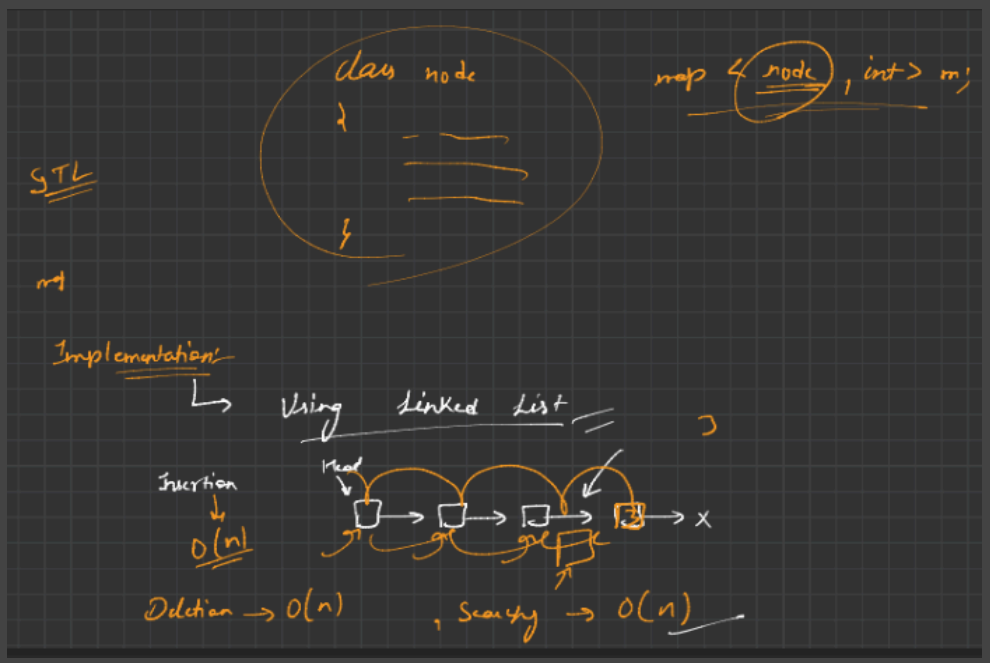
Intro:

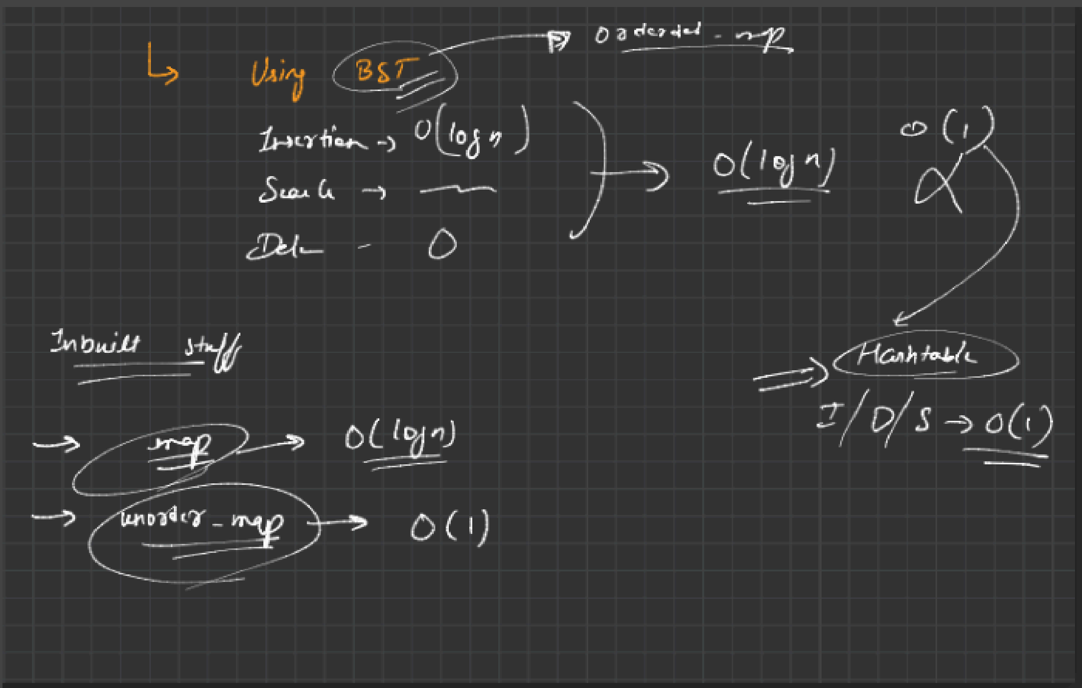




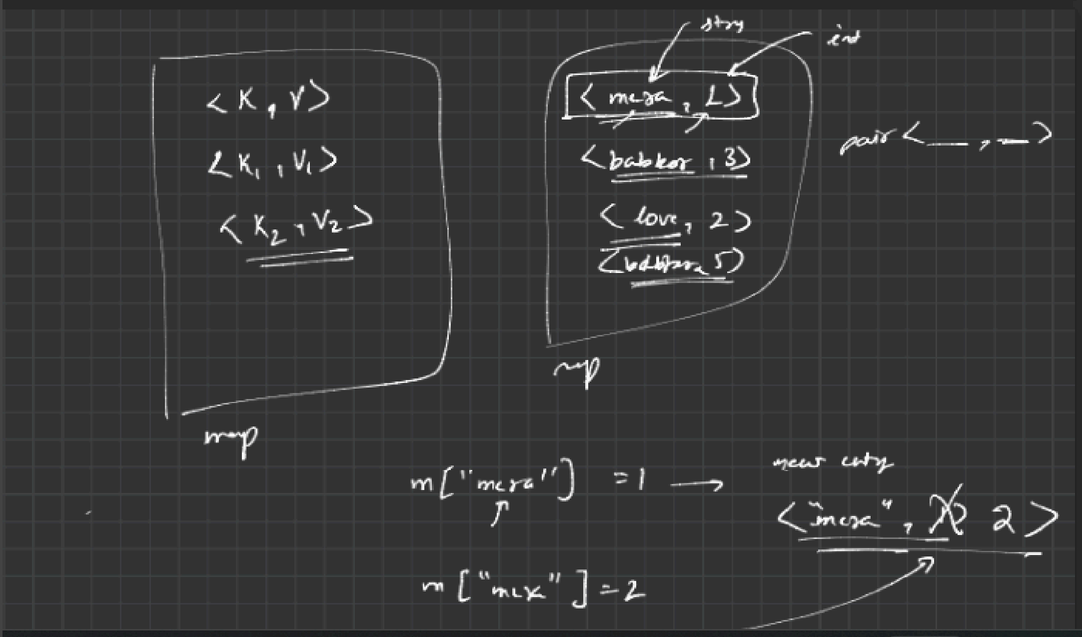

for (int i=0 ; 25)
max
'a' → 0
'b' → 1
'c' → 2
'd' → 3
'z' → 25
ch – 'a' →
'a' – 'a' → 0
'b' → 'a' → 1
'c' – 'a' → 2
'z' – 'a' → 25


sto → " mera naam hai babbar mera babbar babbar
o/p → maximum occuring word
mera → 2
naam → 1
hai → 1
babbar → 3
answer
mera → 0
naam → 1
hai → 2
babbar → 3
mapping possible
0 1 2 n
< mera, 2 >

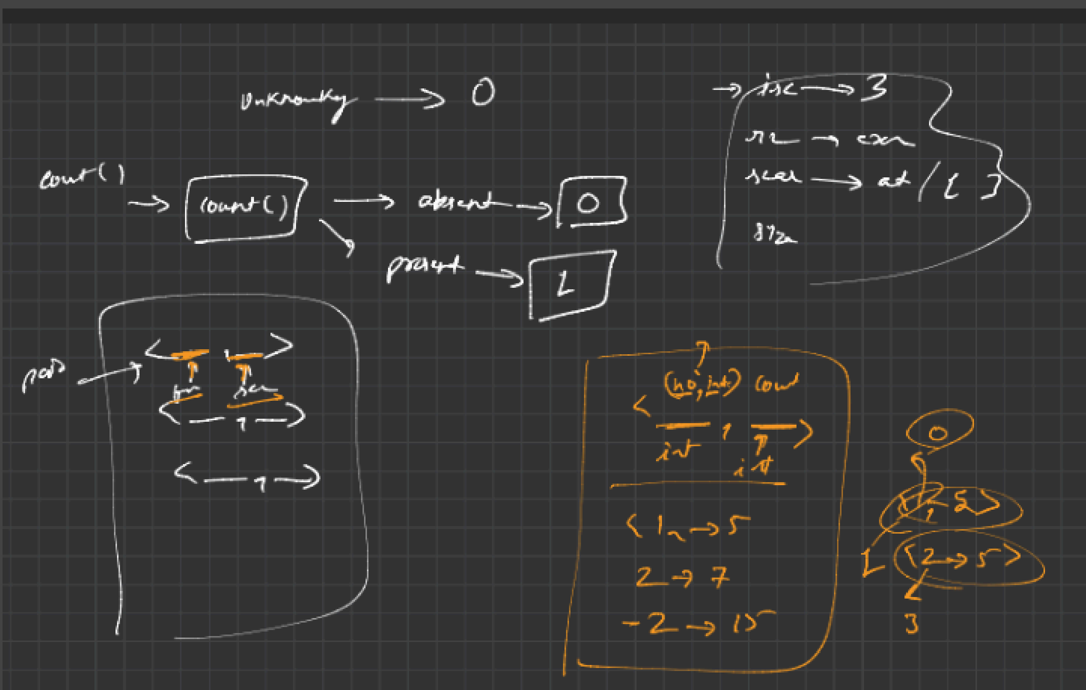

class node
{
}
map < node , int > m;
STL
map
Implementation:
Using linked list
Insertion
O(n)
Head
x
Deletion → O(n)
, Searching → O(n)



Using BST
ordered_map
Insertion → O(log n)
Search →
Delete
O(log n)
O(1)
Inbuilt stuff
→ map → O(log n)
→ unorder_map → O(1)
Hashtable
I/D/S → O(1)

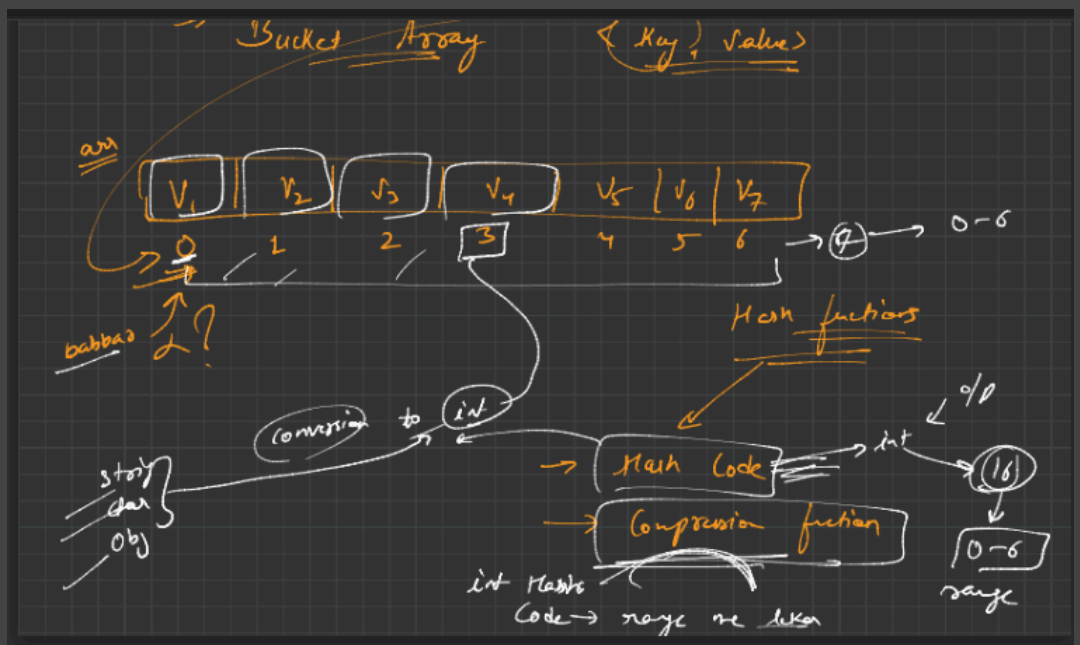


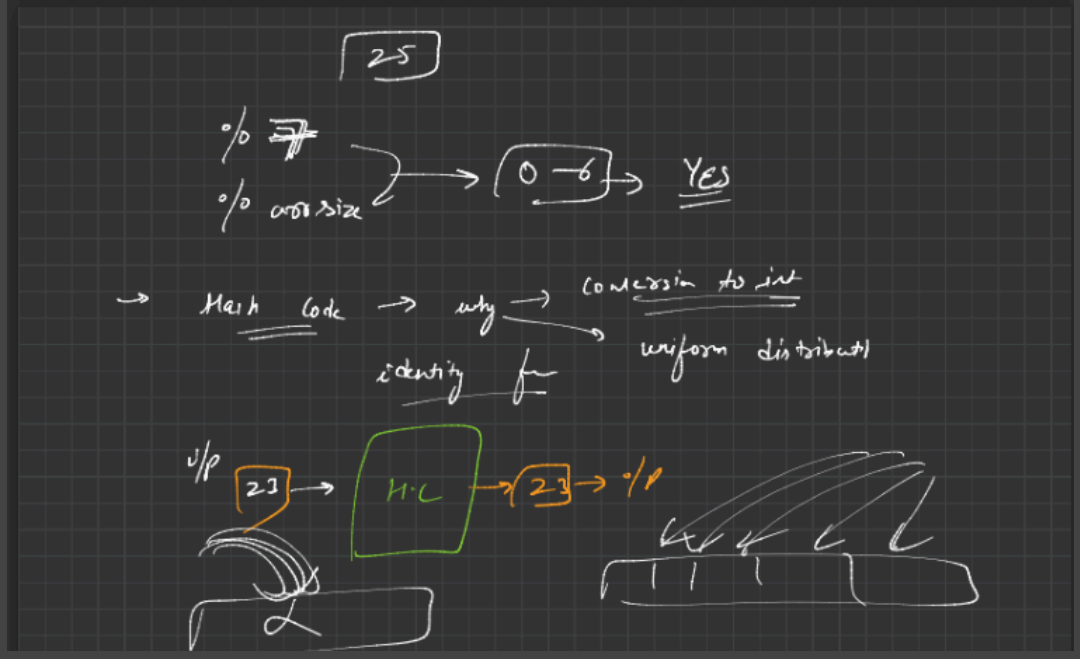
< K , V >
< K1 , V1 >
< K2 , V2 >
map
string
int
< mera , 1 >
< babbar , 3 >
< love , 2 >
map
pair < __ , __ >
m["mera"] = 1
m["mera"] = 2

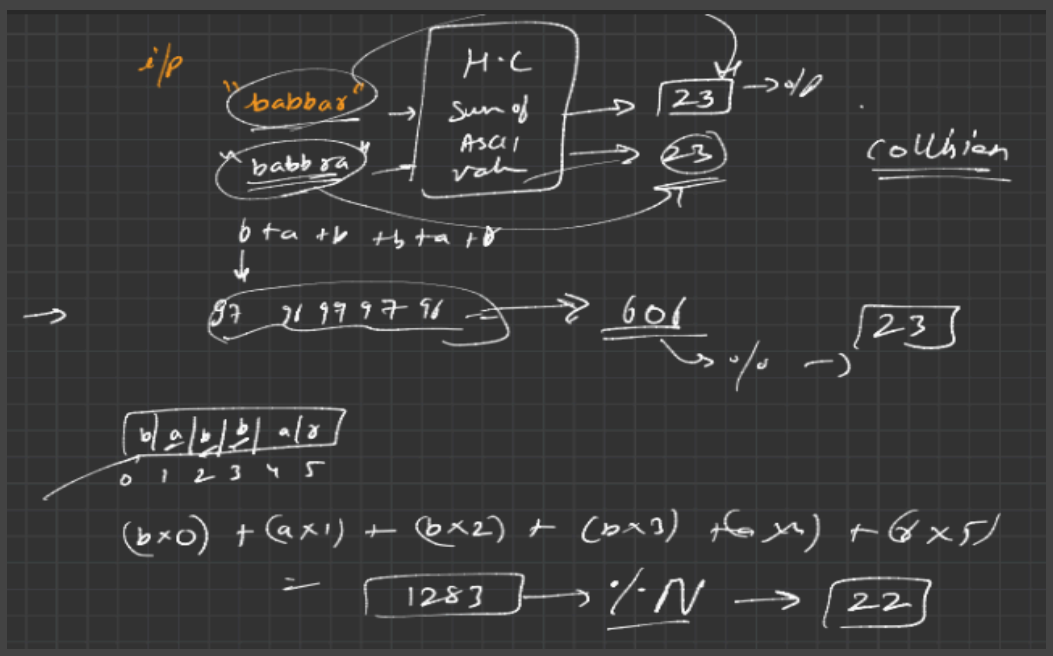


UnknownKey → 0
count()
count() → absent → 0
present → 1
pair
int
< 1 → 5
2 → 7
-2 → 15

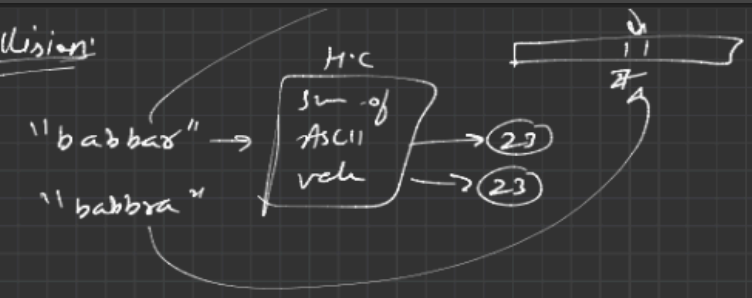
Bucket Array:







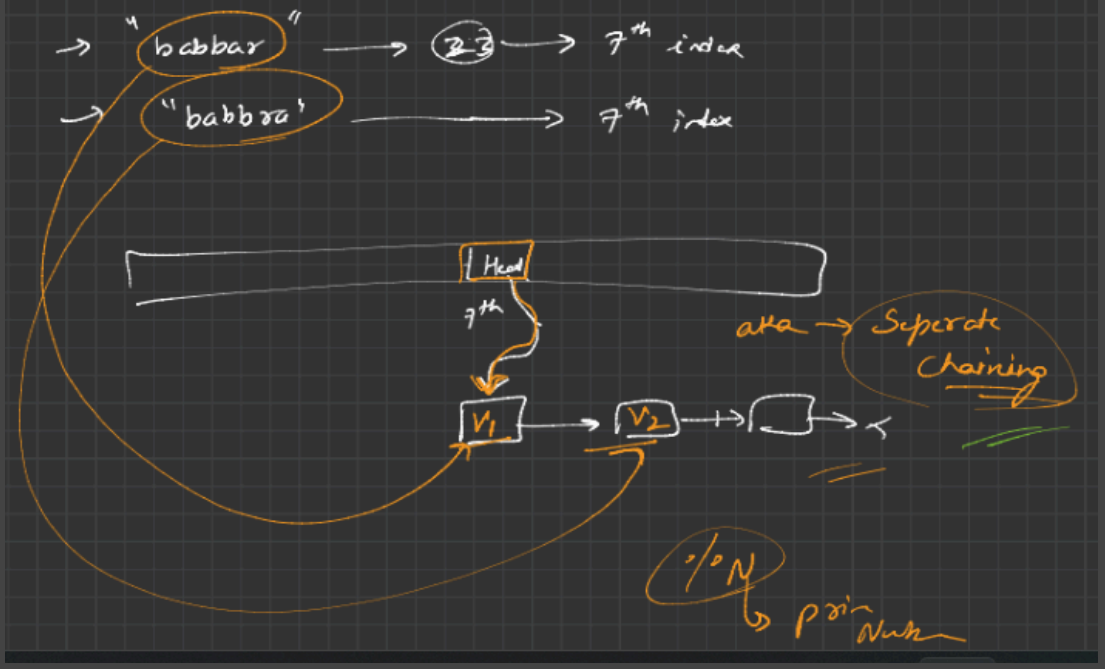
→ Collision:



Collision Handling :-

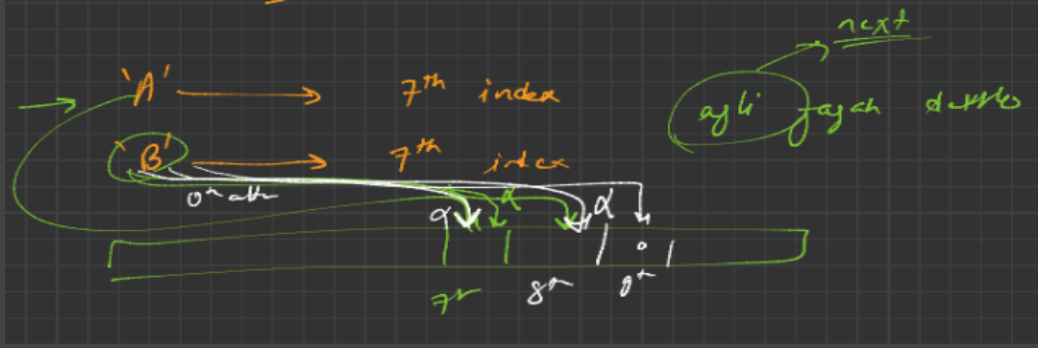
→ Open Hashing → same place pr tri jaa

→ Closed Addressing



100 word → index → 100 LL α

→ Open Addressing



→ $H_i(a) = h(a) + f_i(a)$ (7^{th} +1)

i^{th} attempt me kaha pr jaye

→ Linear probing

$f(i) \rightarrow i$

$F(1) \rightarrow 1$

$f(2) \rightarrow 2$

→ Quadratic Probing

$f(i) \rightarrow i^2$

$F(0) \rightarrow 0$

"babbar" → 7th index

$F(1) \rightarrow (1)^2 \rightarrow 1$

$F(2) = 2^2 \rightarrow 4$

$h_i(a) = h(a) + f_i(a) = 8$

$= 7 + 4 = 11$

S.C

$i^3 + 2$

$i^2 + 13$

$i^2 + 9$

=> Complexity analysis

"mera bhai love babbar bhai kya had"

Insertion

n → total no of word

k → word length

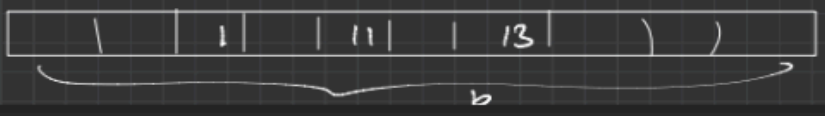
if $n >> k$

H.C → $O(k) \rightarrow \sim O(1)$

→ traverse → O(n)

n → no of entries in map

b → no of boxes available n = 3



no of entries in a box → n/b → Load factor

we always ensure n/b < 0.7

→ H.C → acha → O(1)

L.F ≥ n/b < 0.7

it is very ✗ to use → O(1)

n/b → n↑

n/b < 0.7 →

b↑

↳ no of available boxes ↑

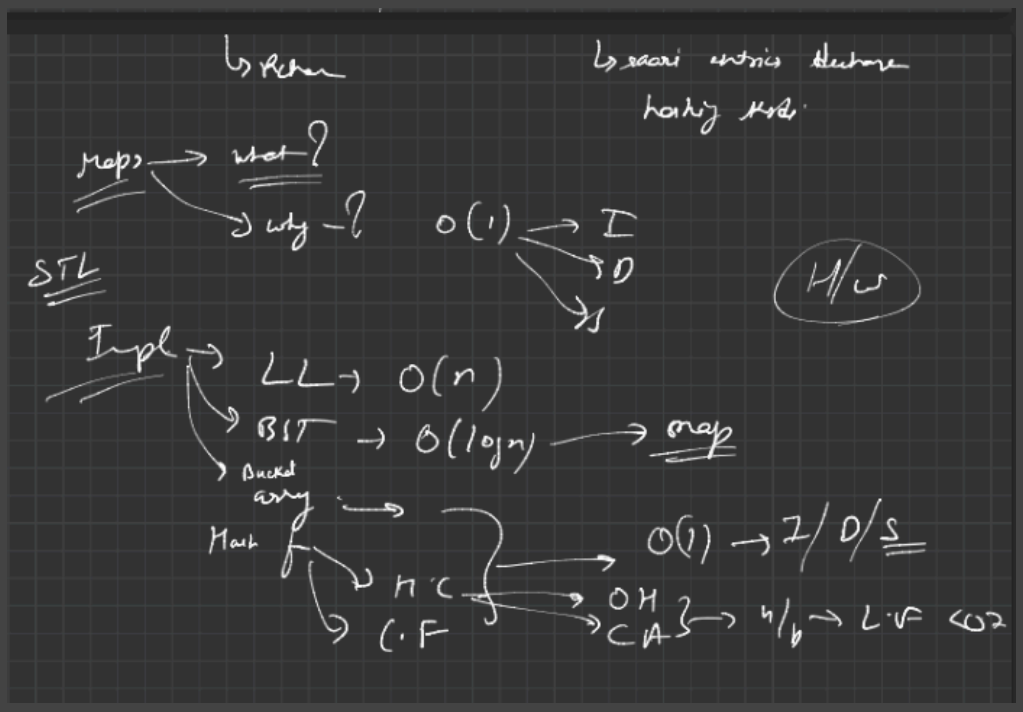
↳ Rehashing

↳ increase bucket size (2×)

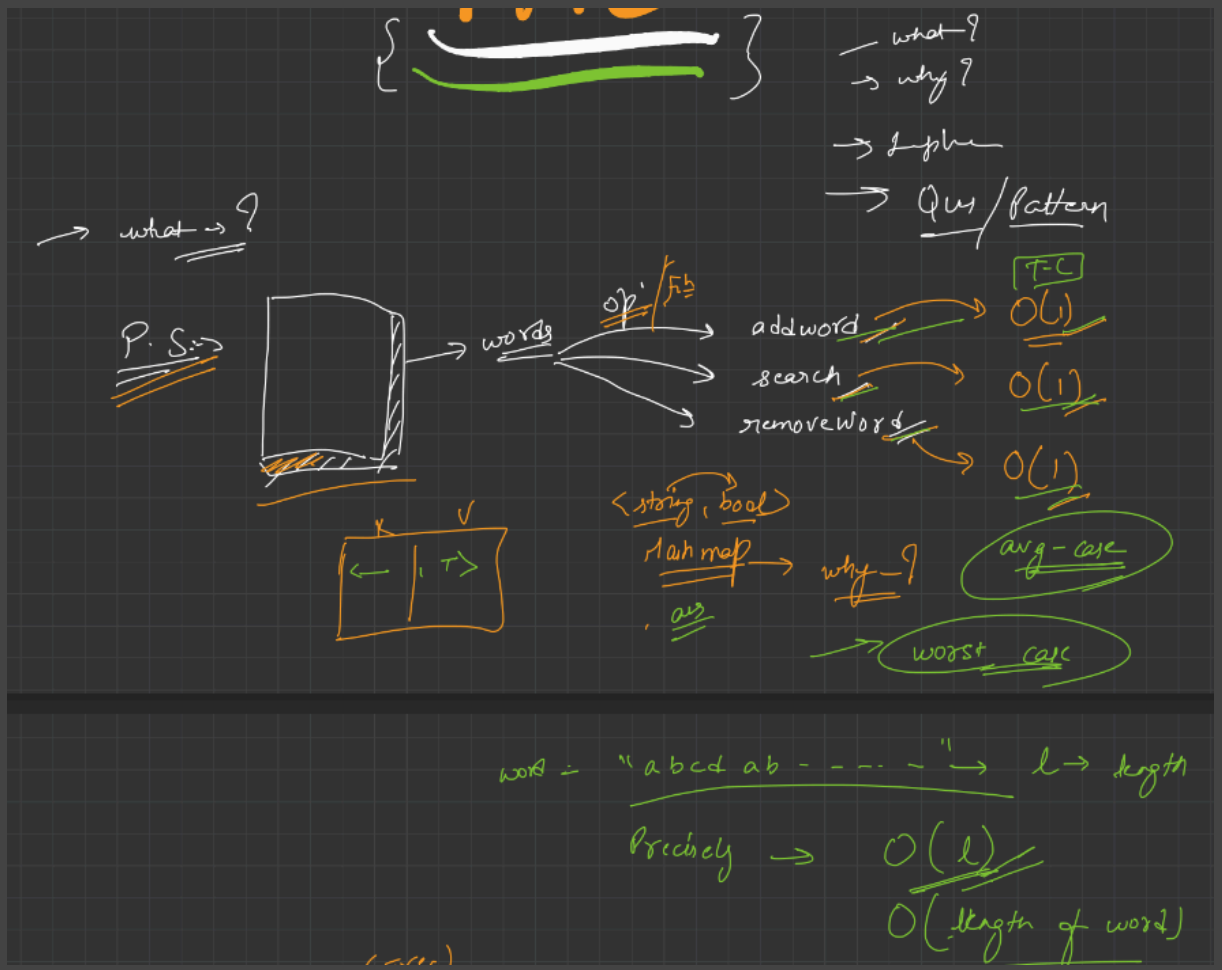
insertion — Delete/search → n/b → half

H.C → O(1)

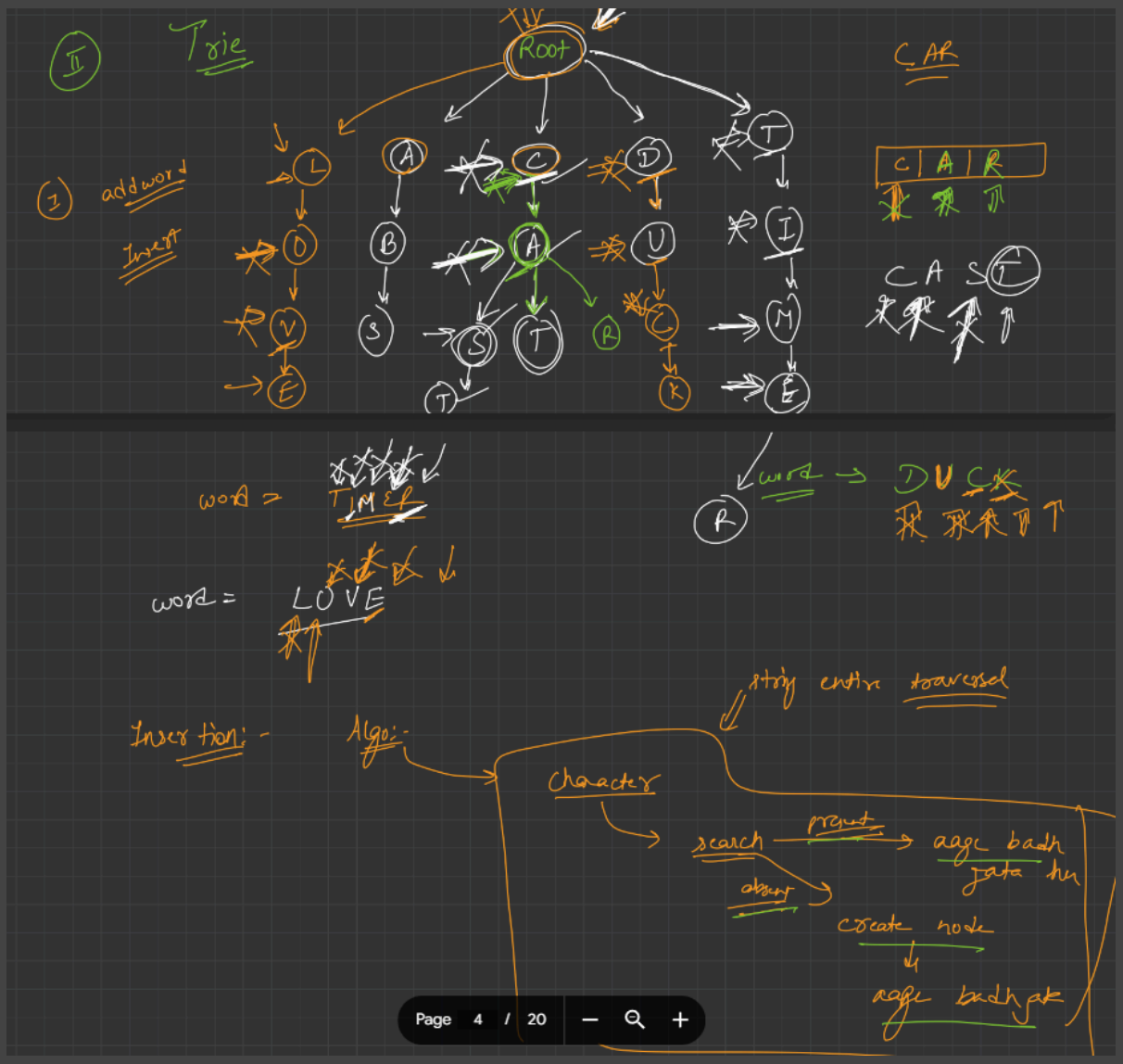
→ L.F < 0.7


Maps → what?
why -?
O(1) → I
D
S
STL
H/W
Impl → LL → O(n)
BIT → O(log n) → map
Bucket array
Hash
O(1) → I/D/S
H.C
C.F
OH
CA
H/b → L.F

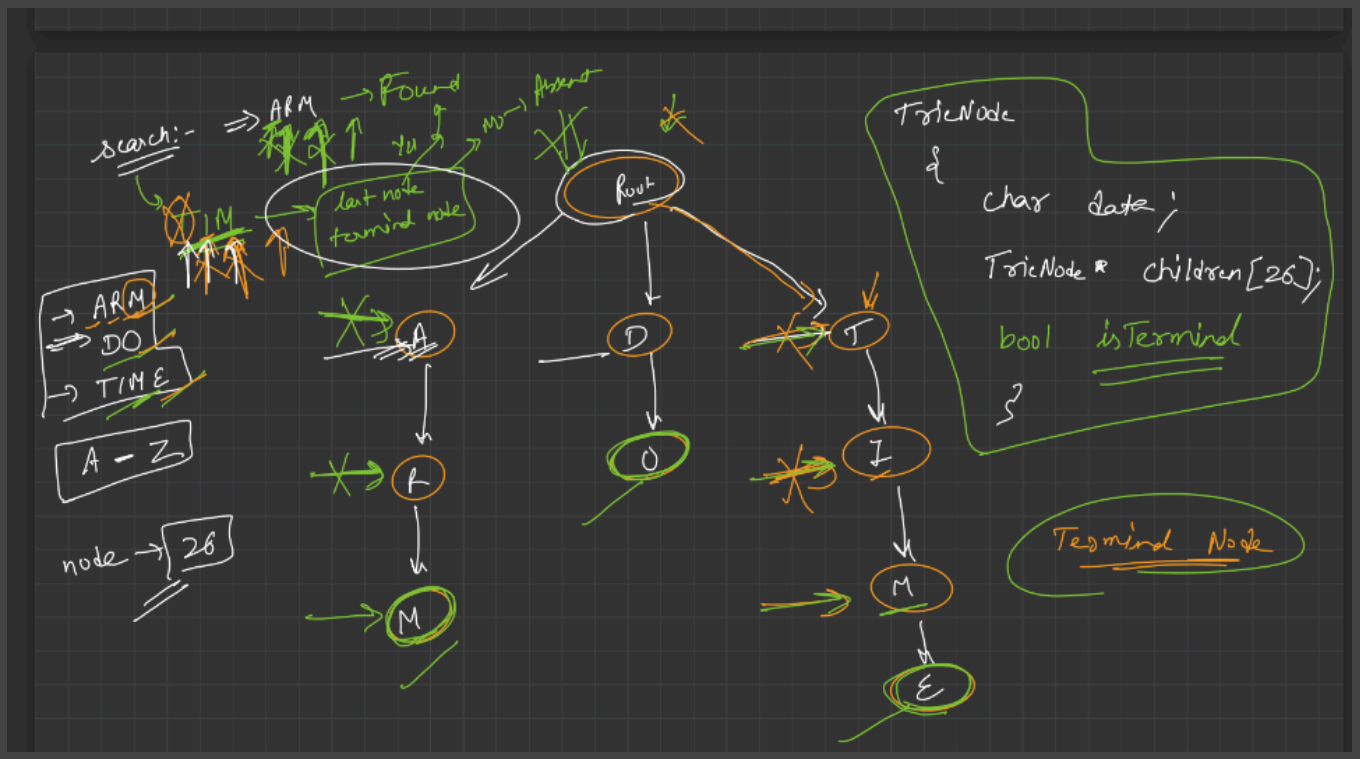
Trie

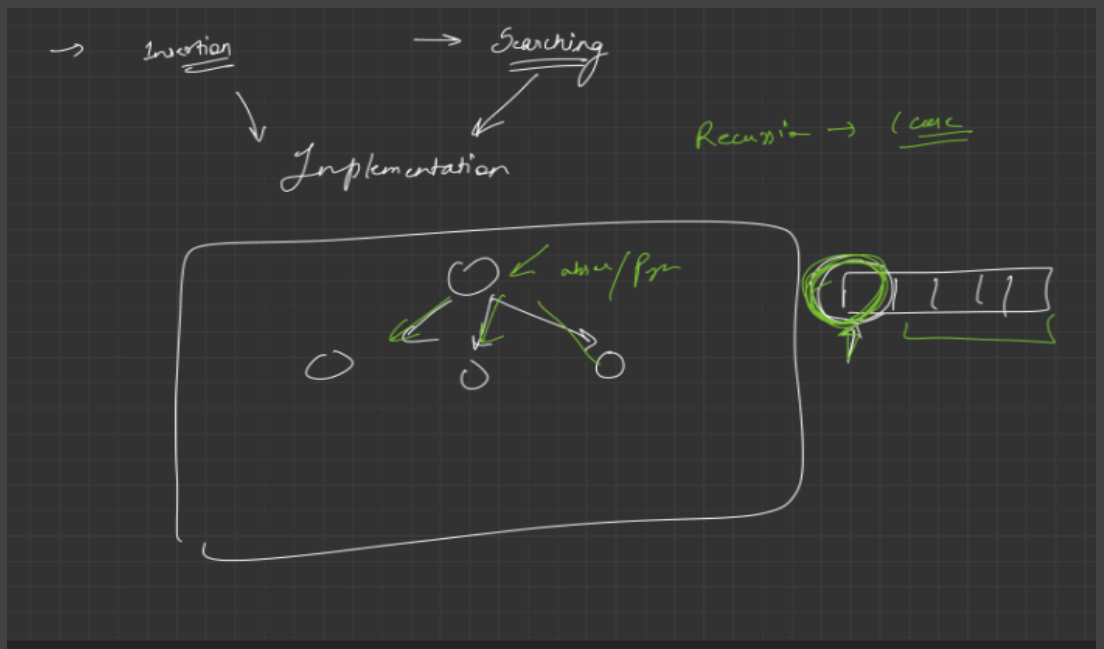

what?
why?
Applications
Ques/Pattern
what →?
P.S.:→
words
addword
search
removeWord
T.C
O(1)
O(1)
O(1)
<string, bool>
Hash map
why -?
avg - case
worst case
word = "abcd ab - - - - - -" → l → length
Precisely → O(l)
O(length of word)

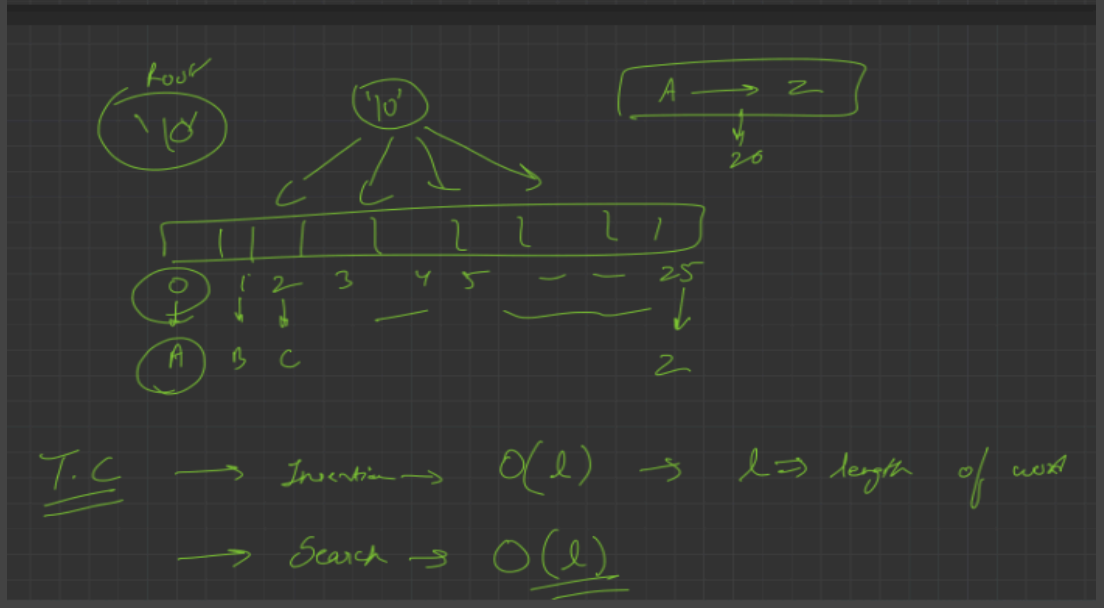
Insert Word:



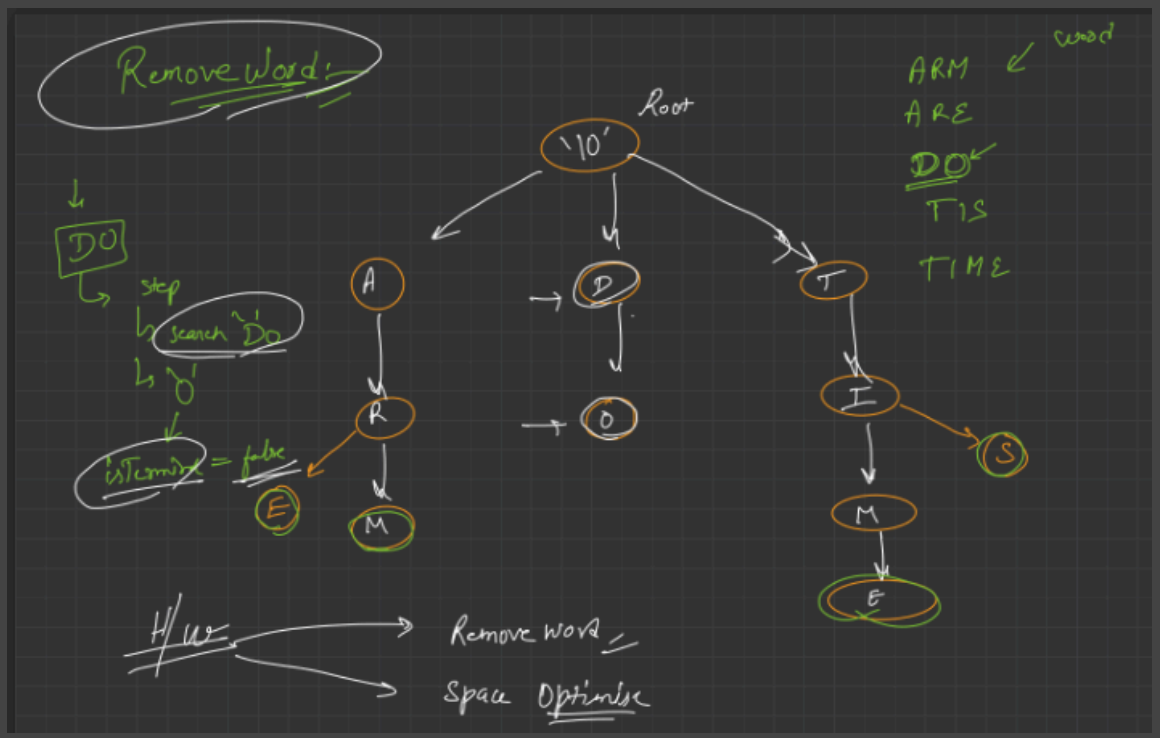
Search Word:



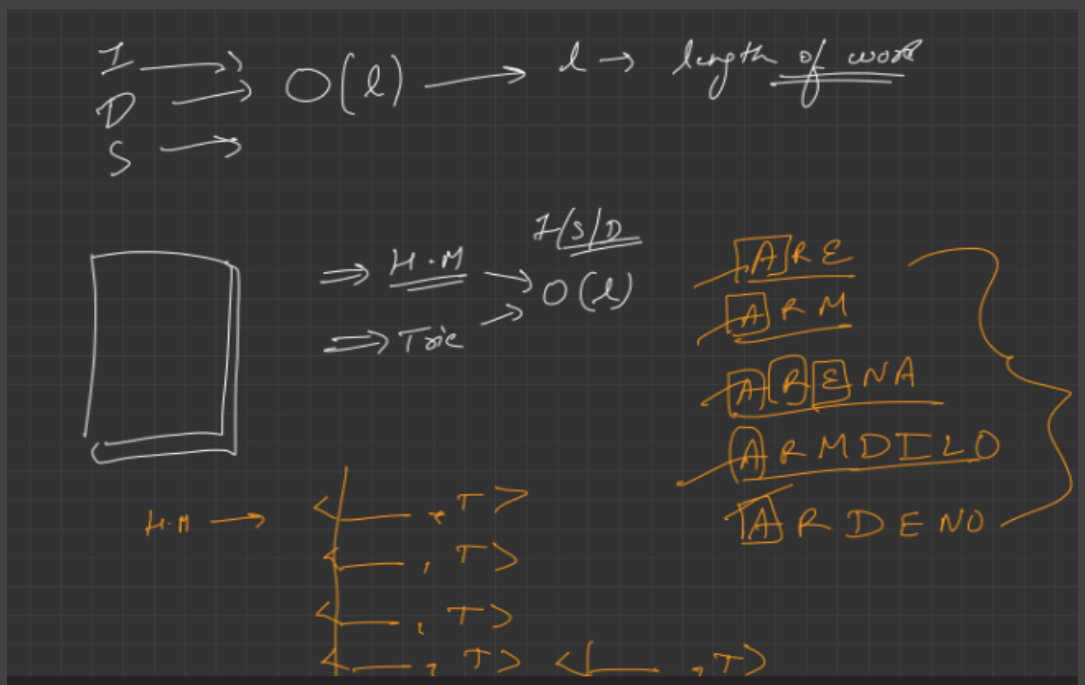

Insertion
Searching
Implementation
Recursion → (conc

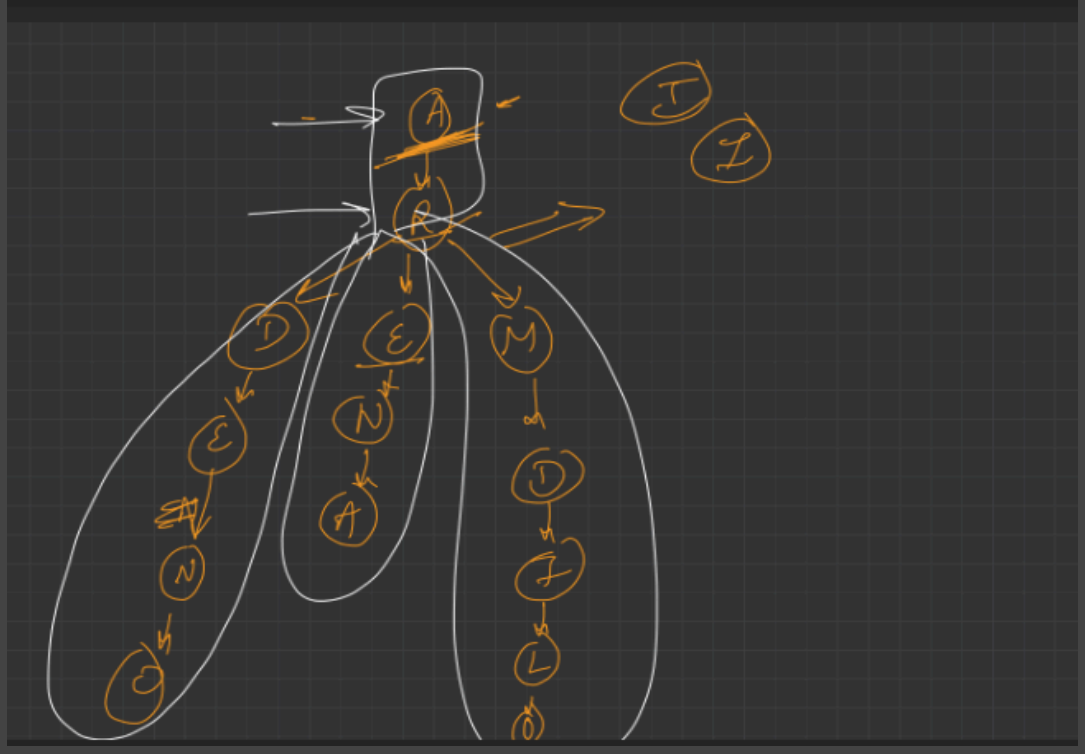

Root
'10'
A → Z
26
0 1 2 3 4 5 25
A B C Z
T.C → Insertion → O(l) → l ⇒ length of word
→ Search → O(l)

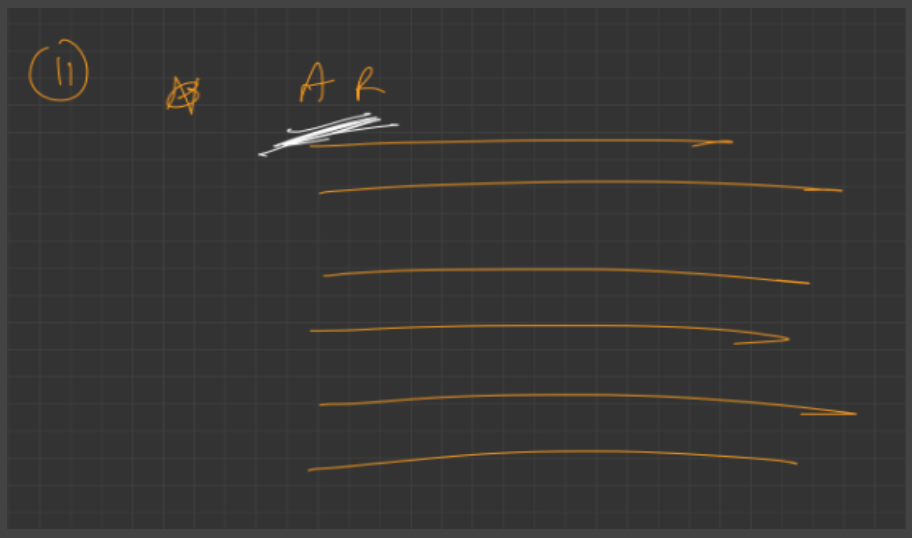
Remove Word:


Remove Word:
Root
'10'
ARM
ARE
DO
TIS
TIME
word
DO
step
search 'D'
'D'
isTerminal = false
A
D
T
R
O
I
E
M
S
M
E
H/W
Remove word
Space Optimise

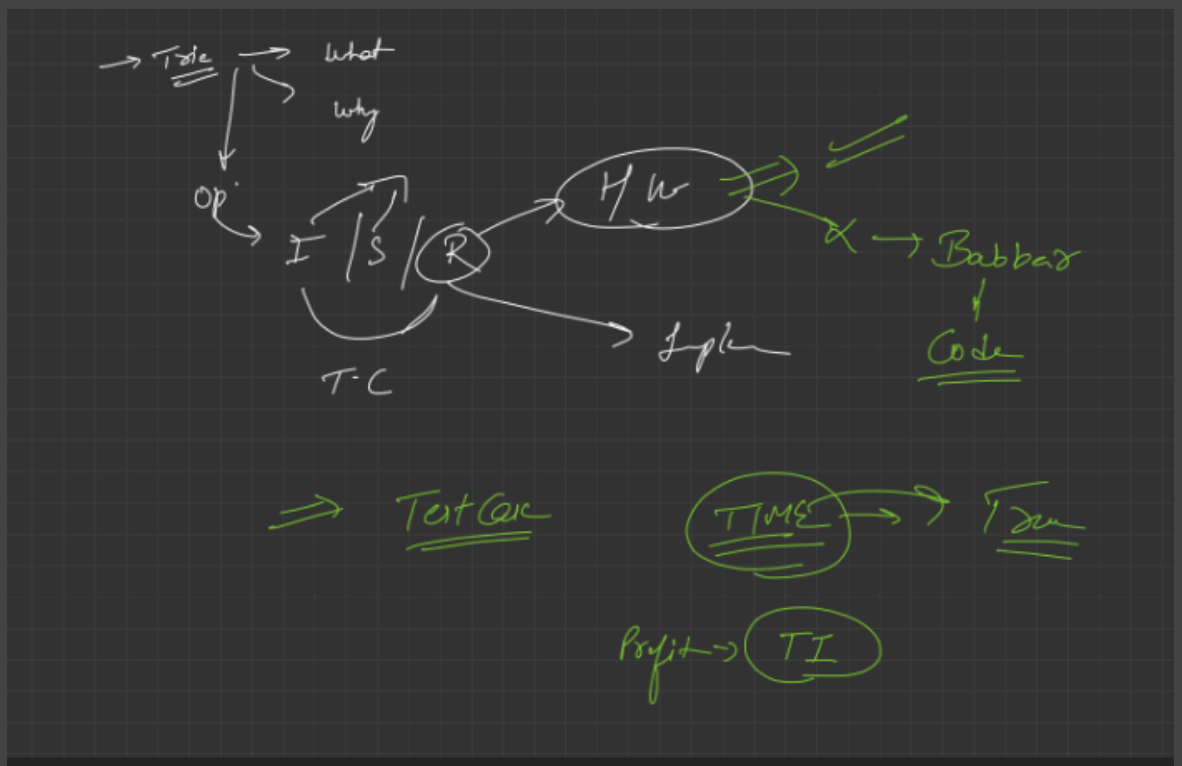
Hashmap v/s Trie:

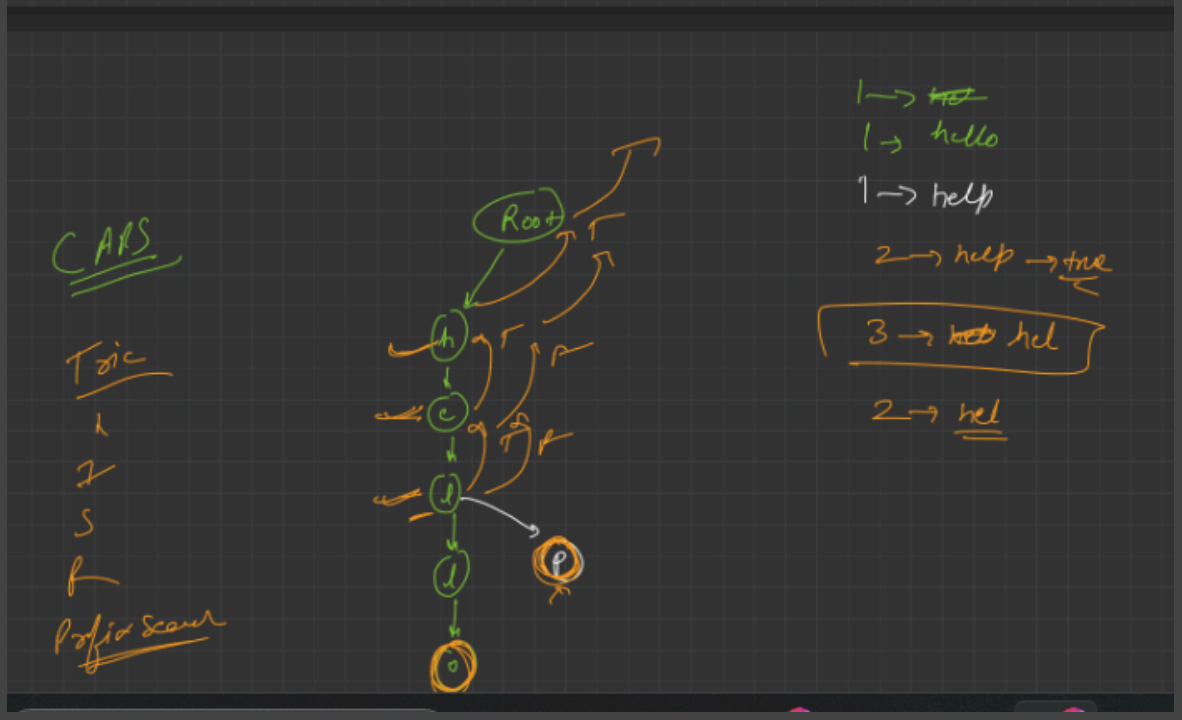




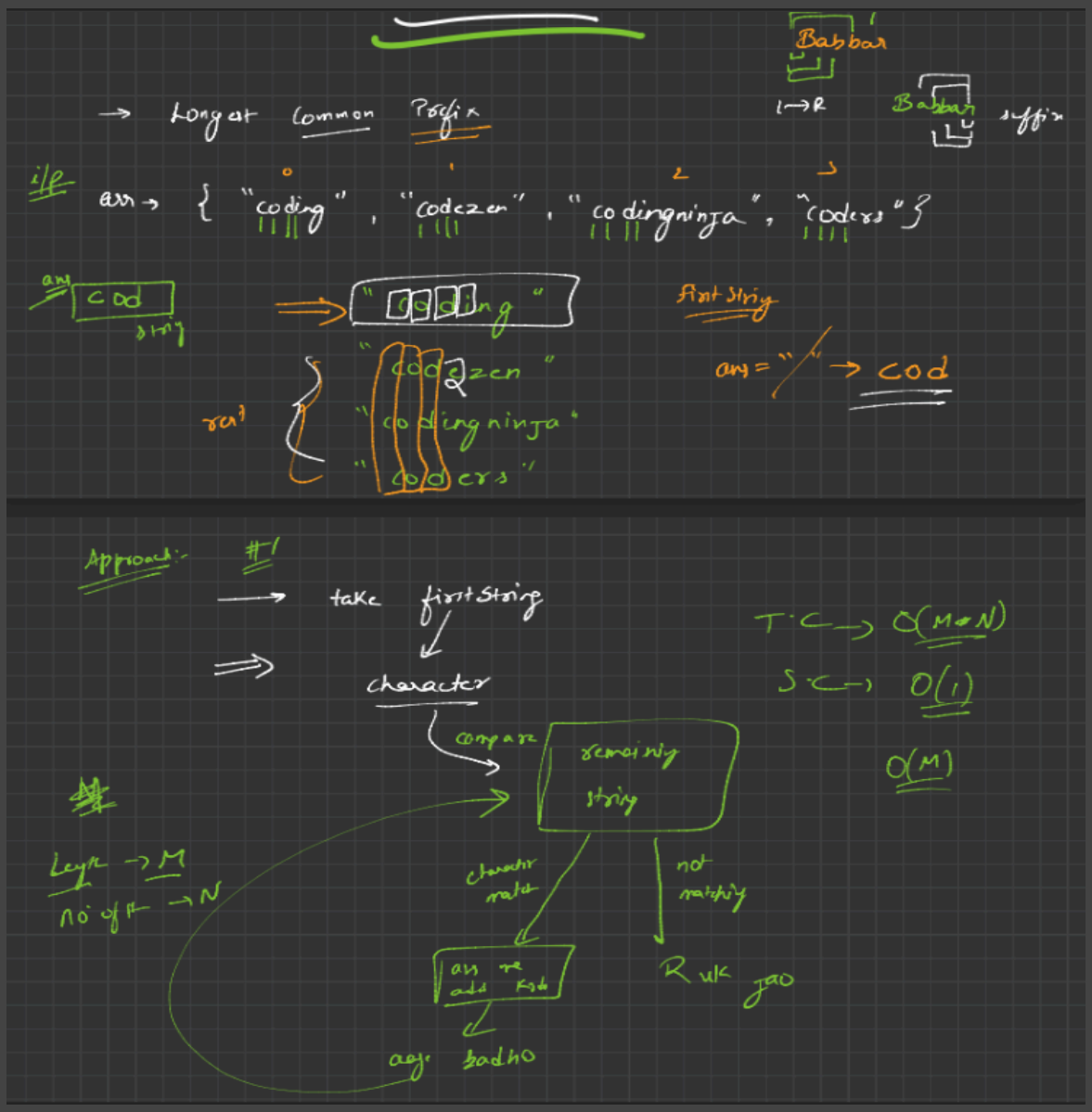


Implement Trie Problem:

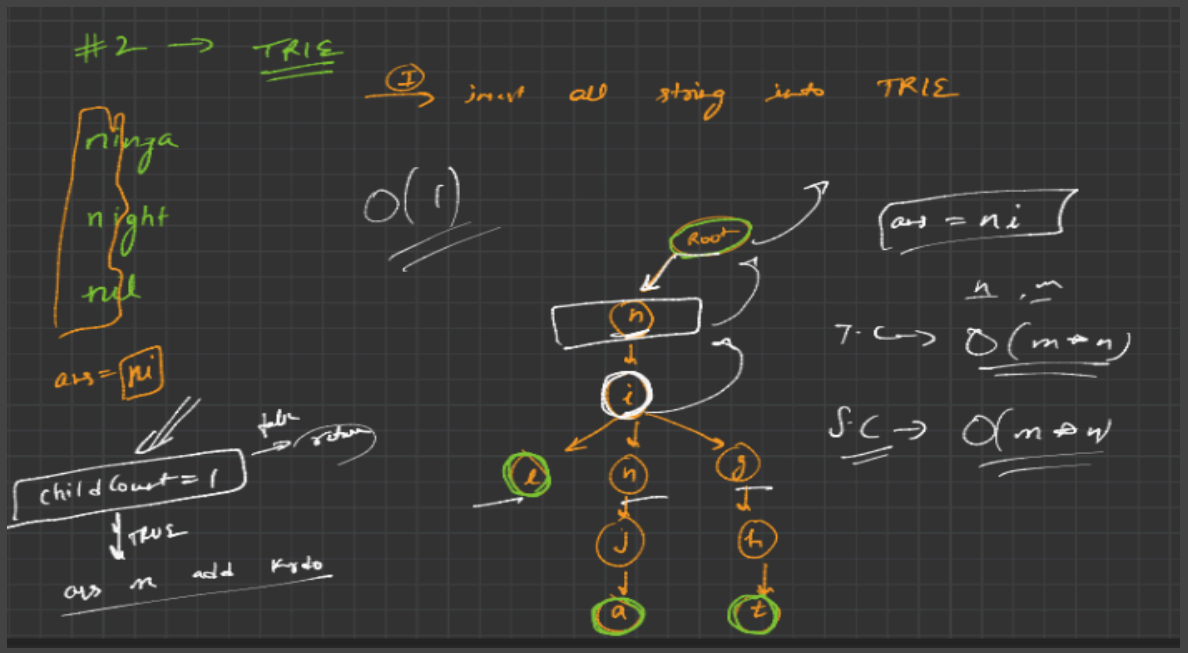




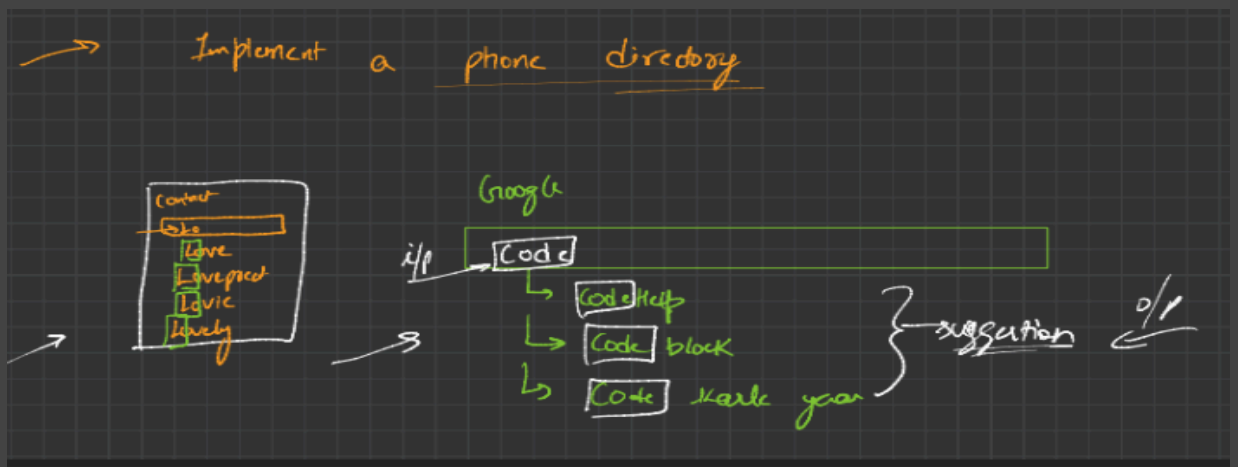
Longest Common Prefix Approach-I:

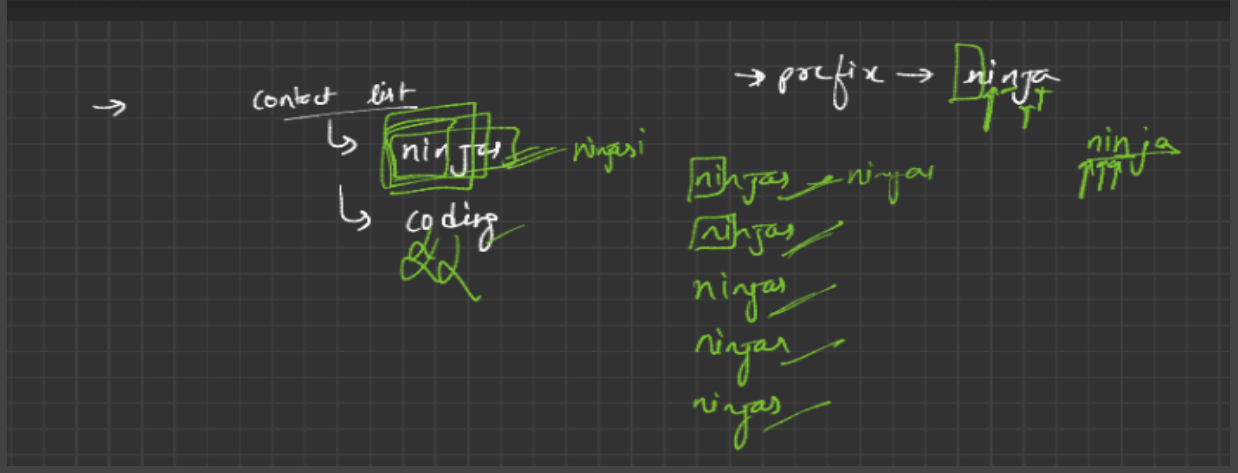


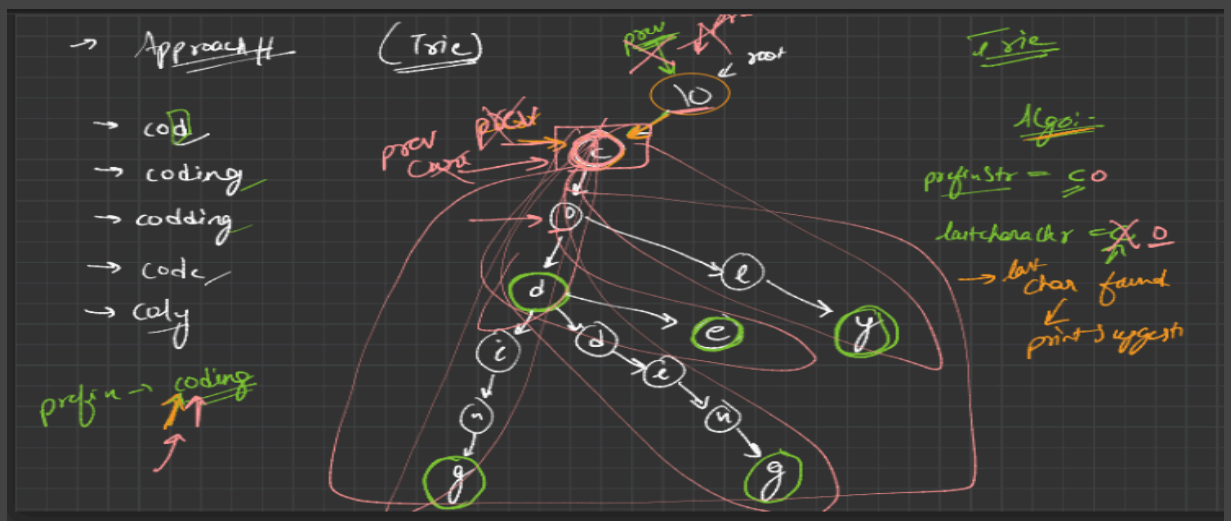
Longest Common Prefix Approach-II:

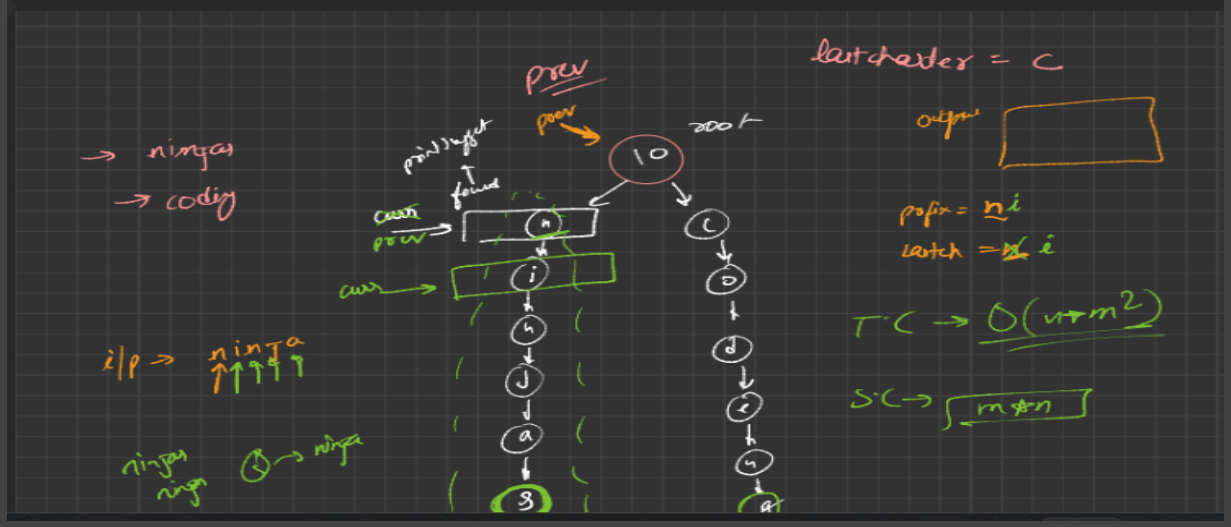


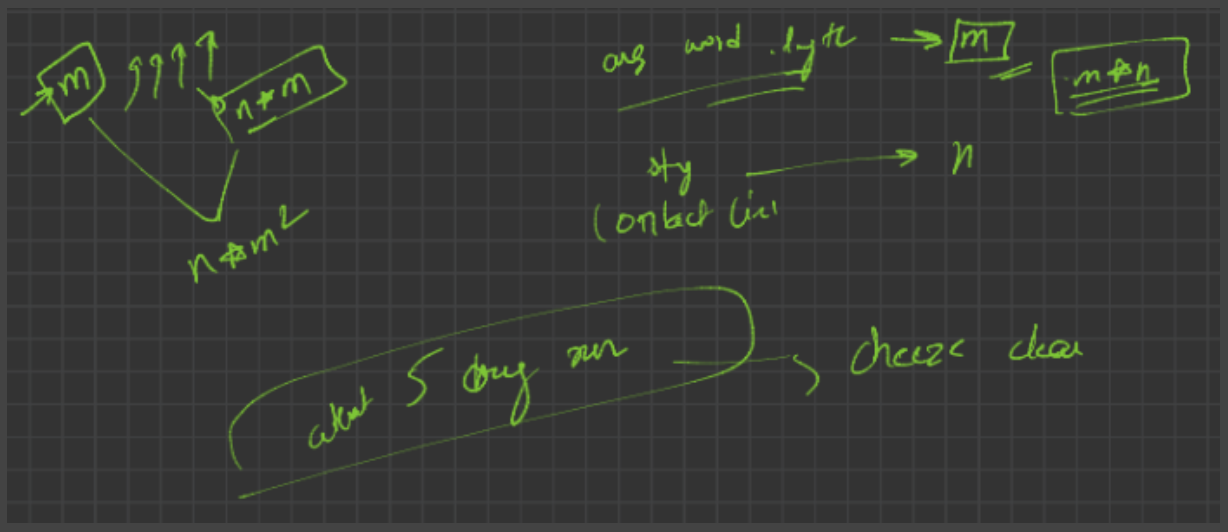
Implement A Phone Directory:



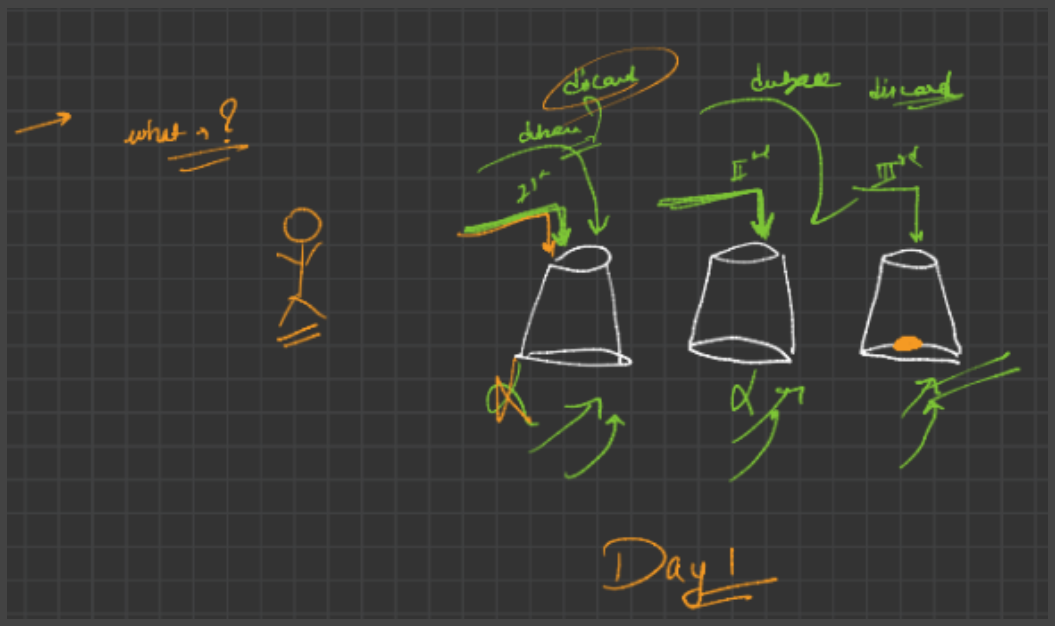




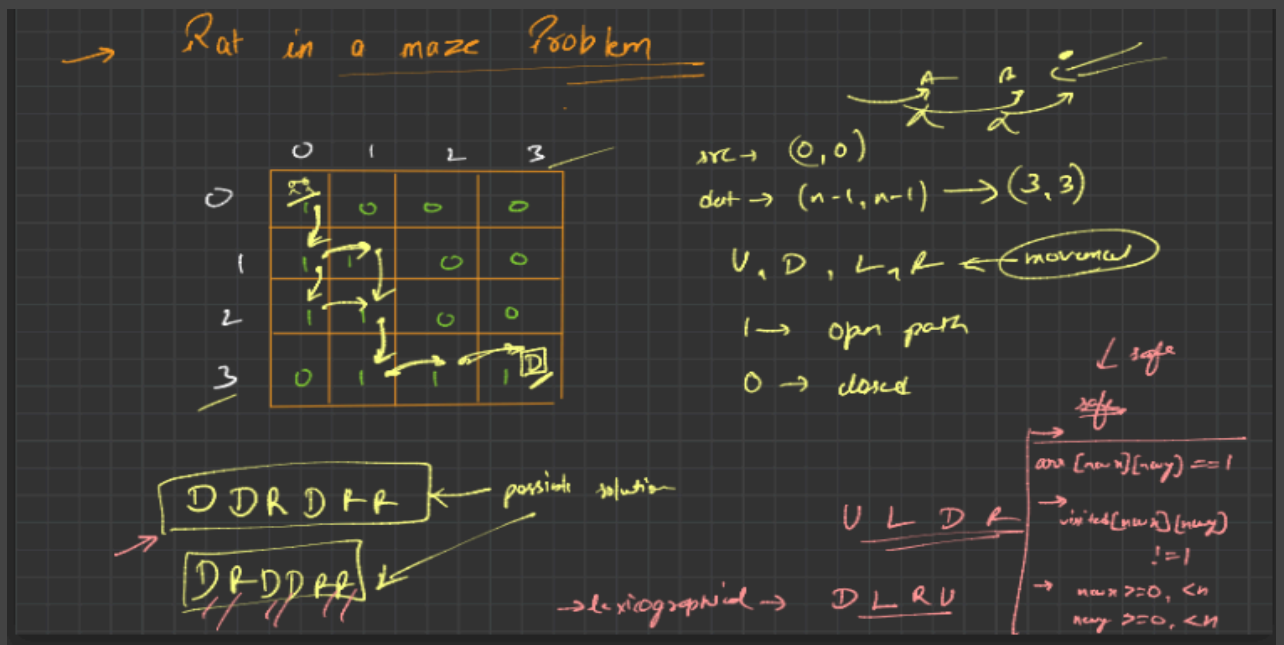


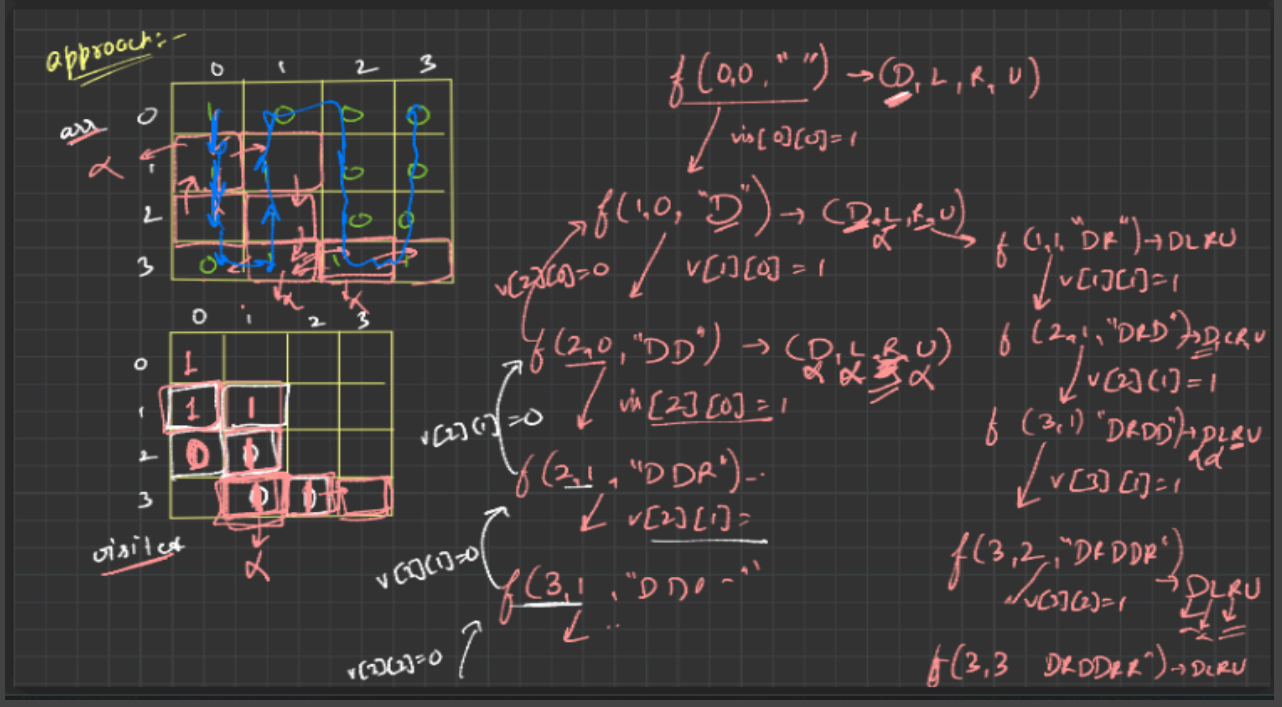


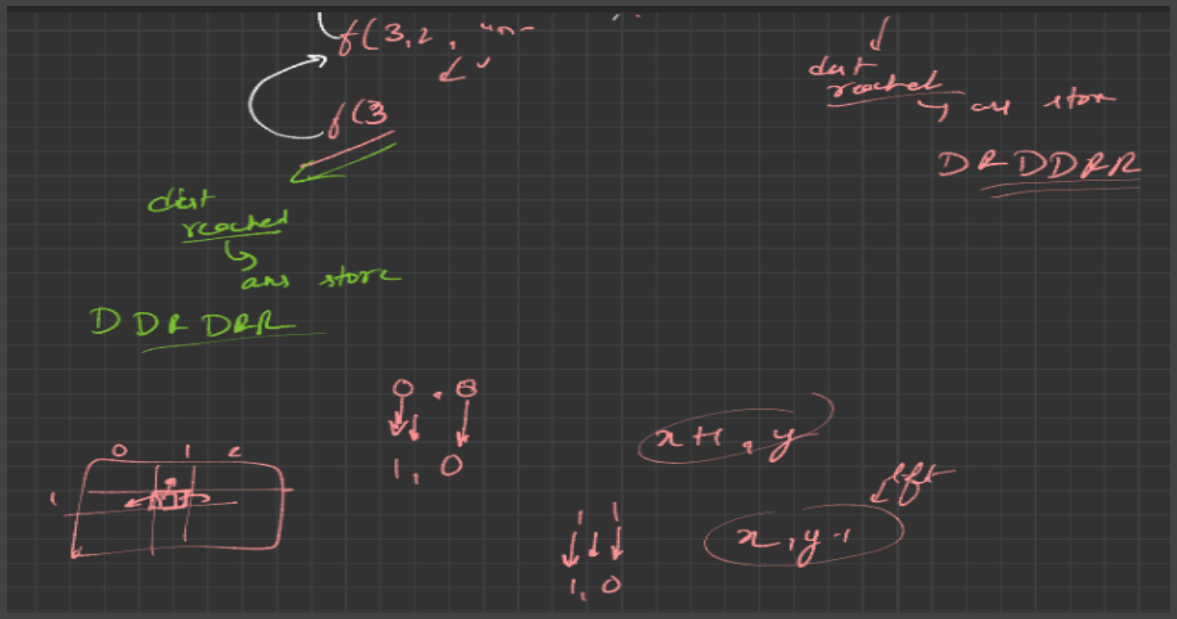
BackTracking

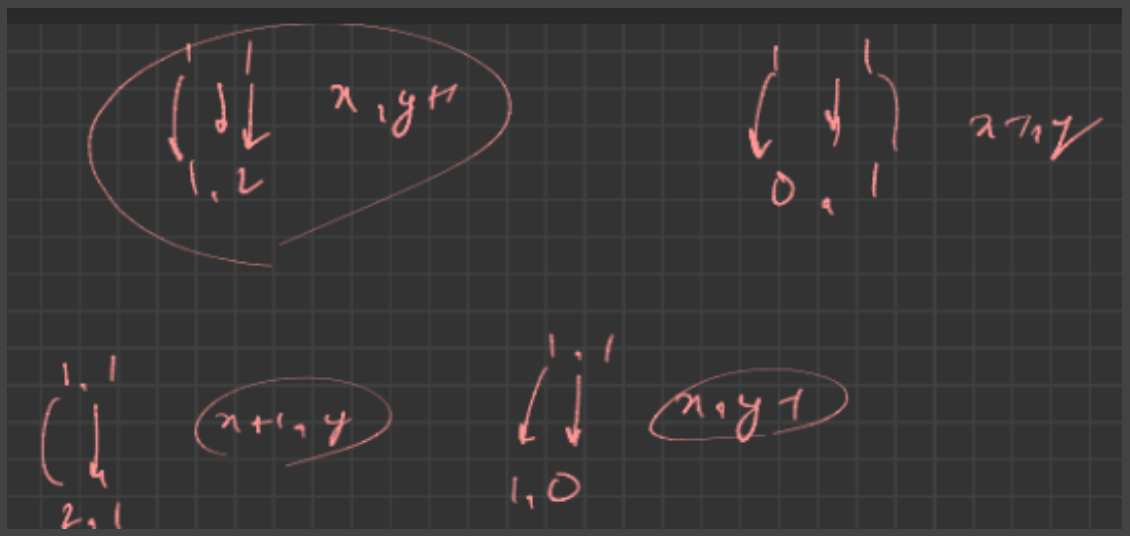


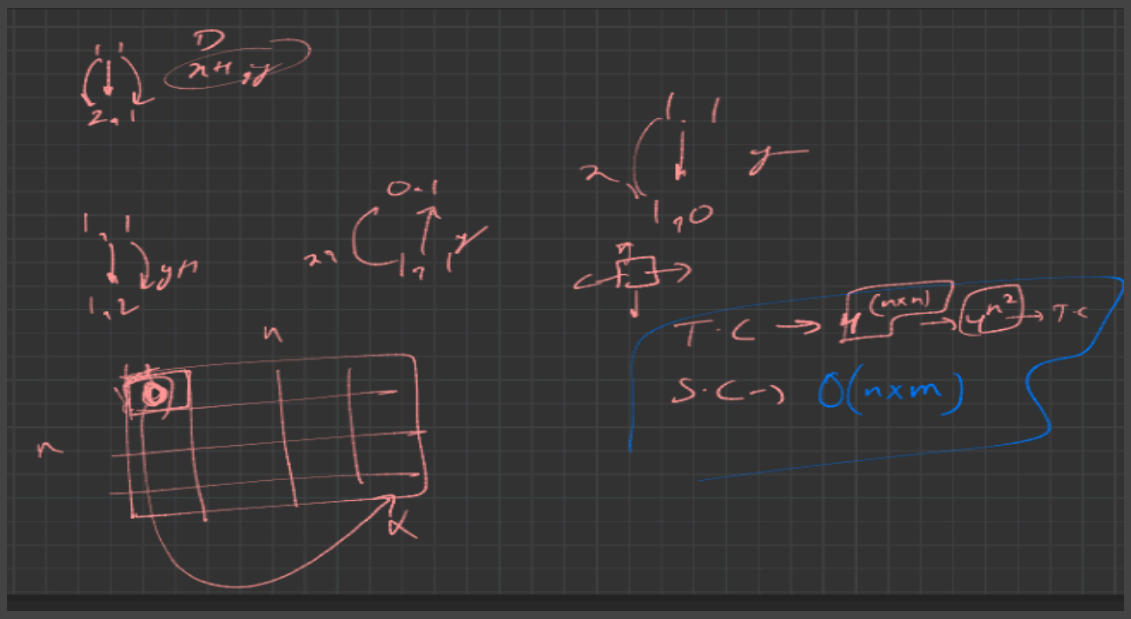
Rat In A Maze Problem:



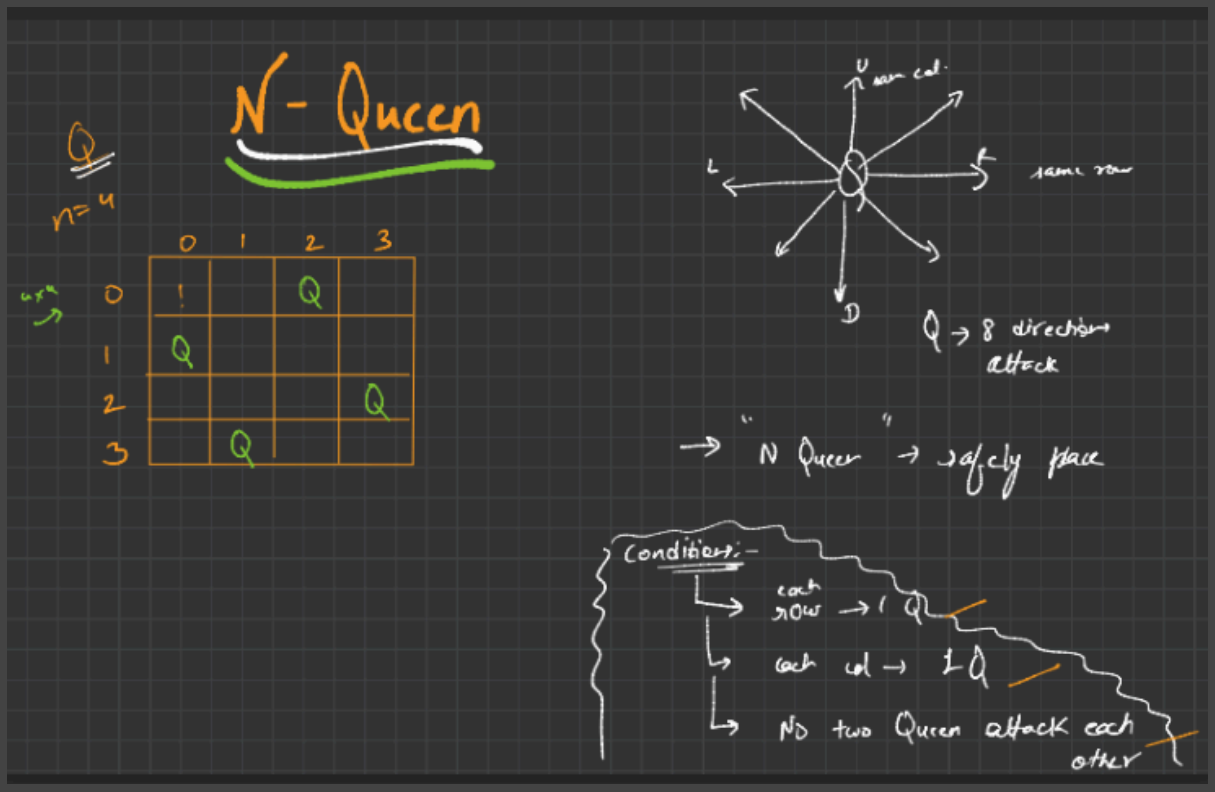


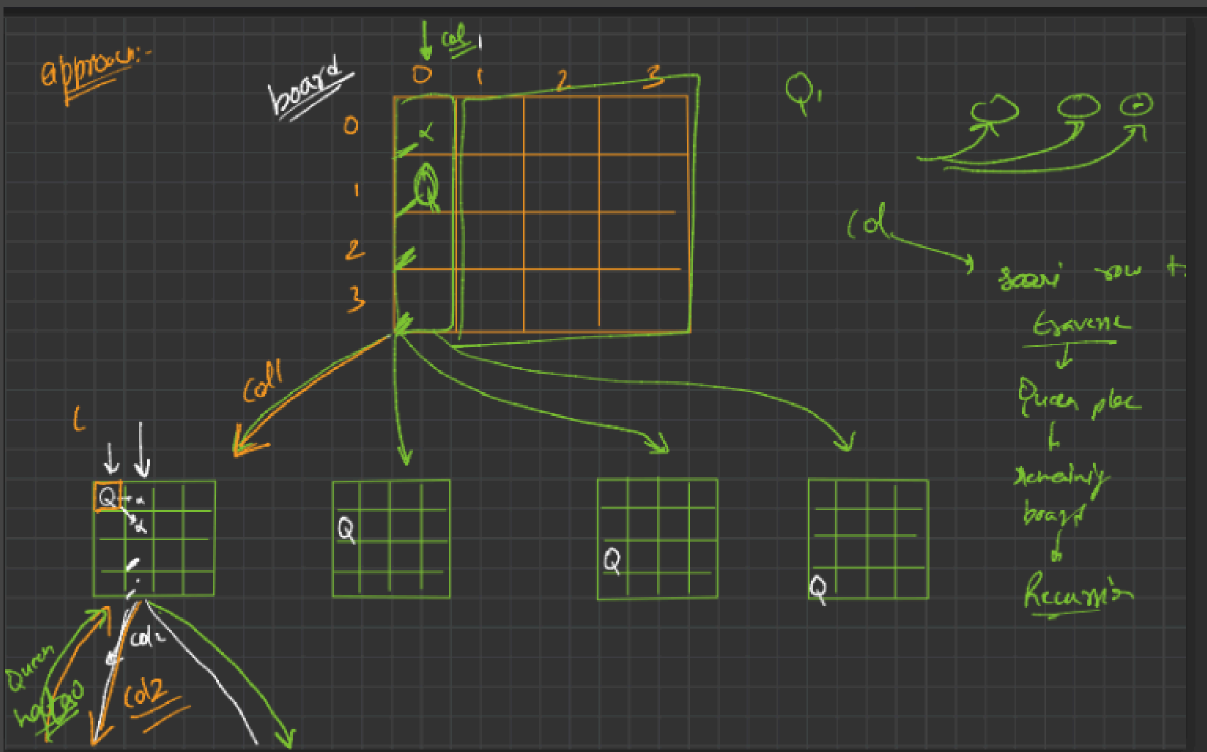


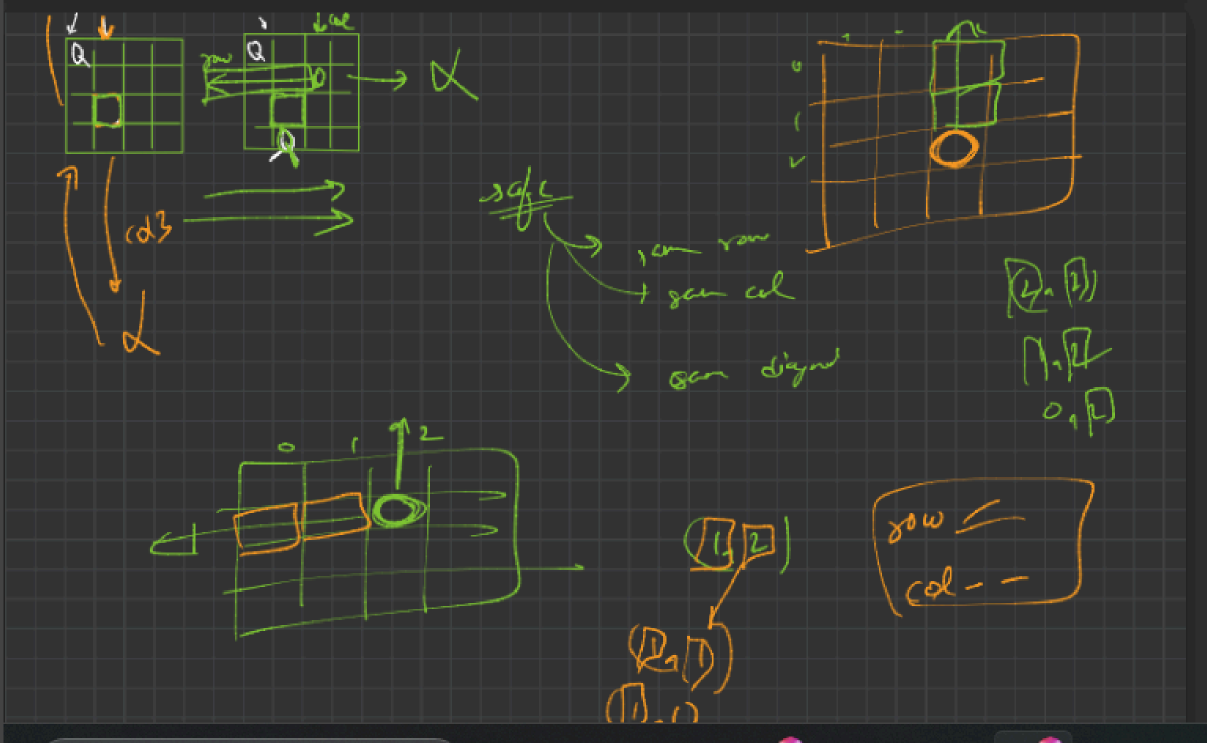


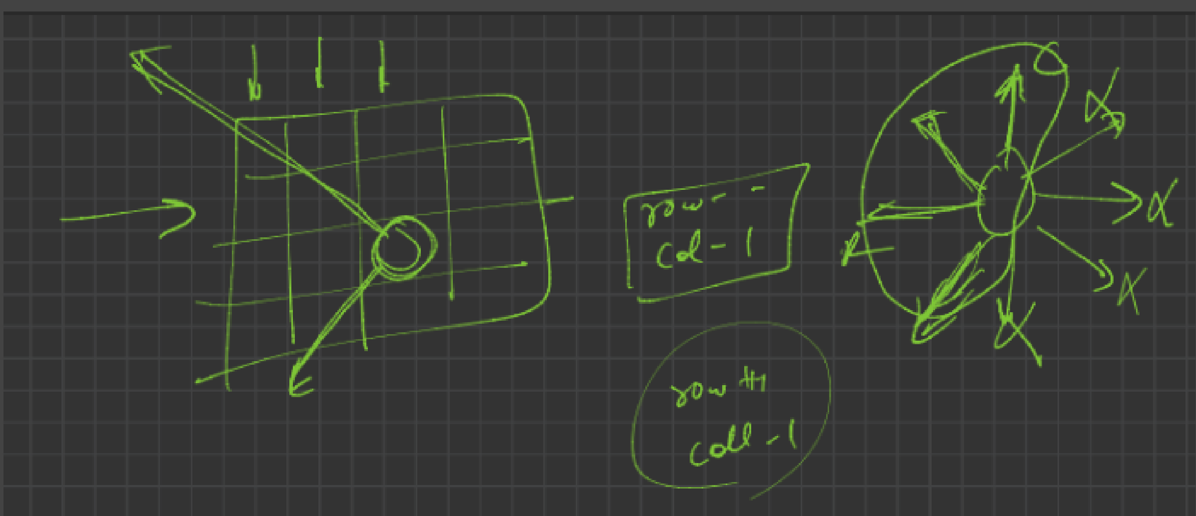


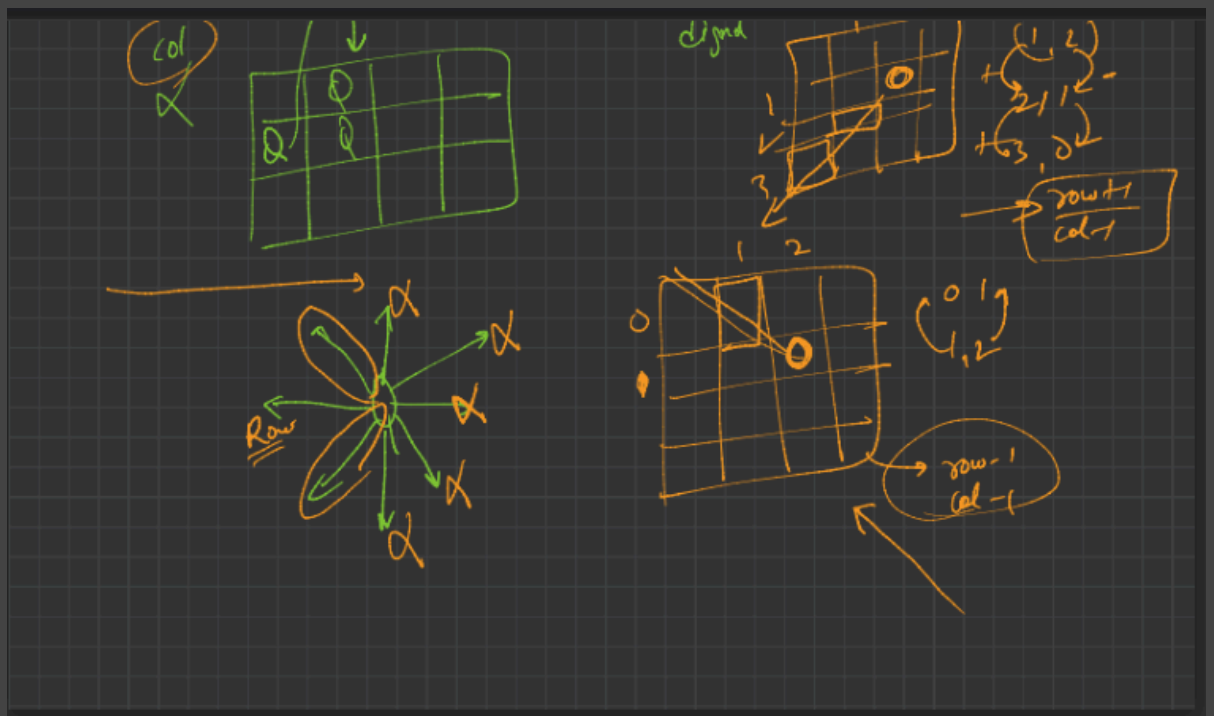
N Queen Problem:

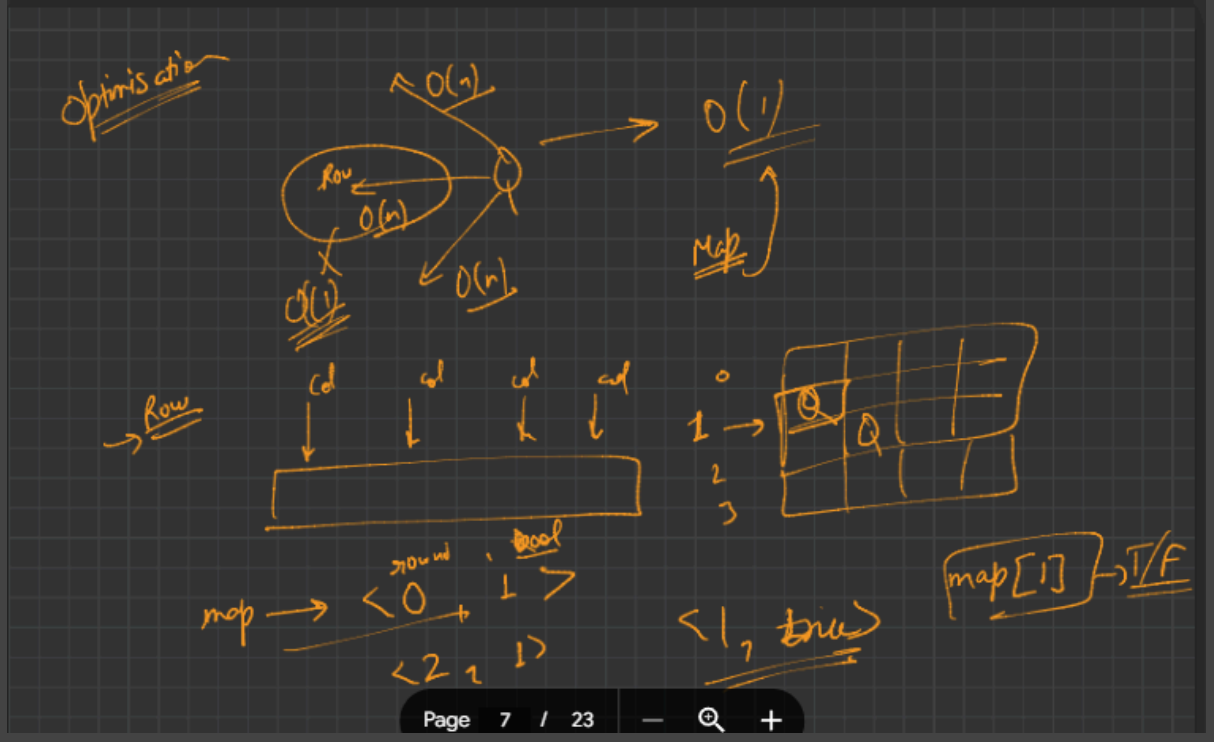



approach:-
board
col
Q1
Recursion

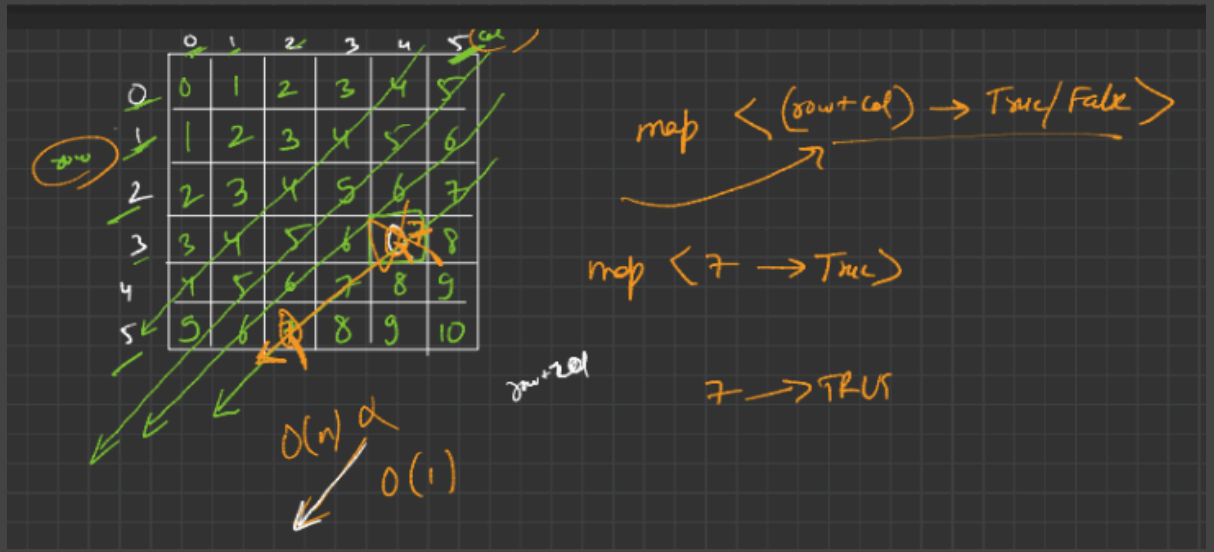

safe
same col
row
col

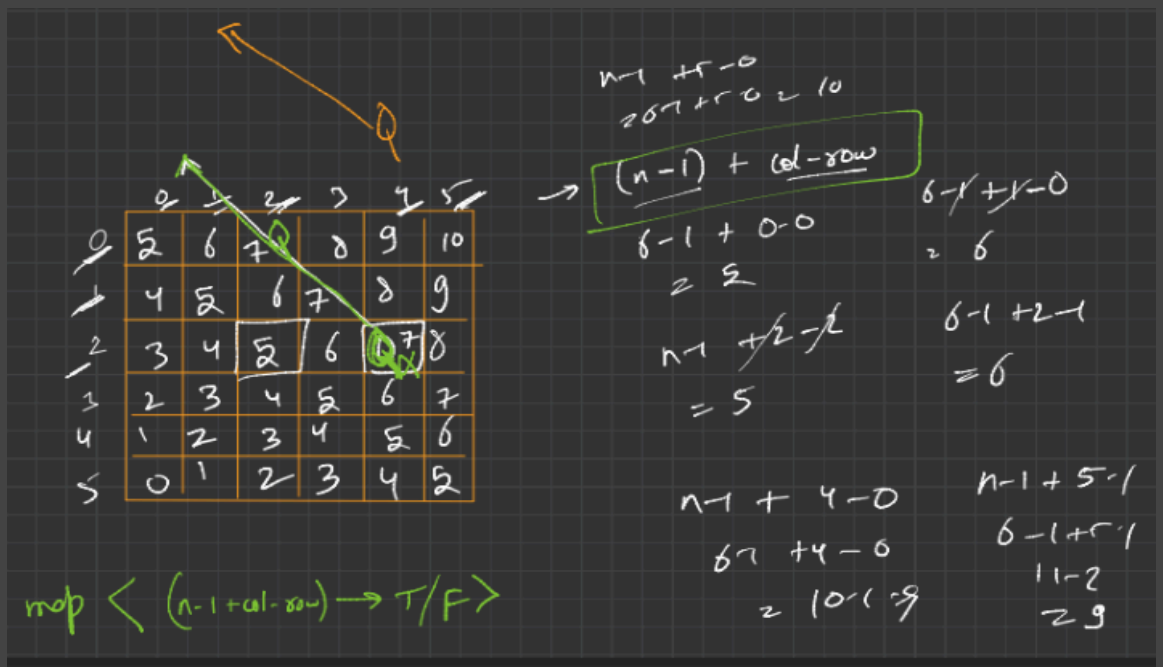

row--
col-1
row ++
col -1

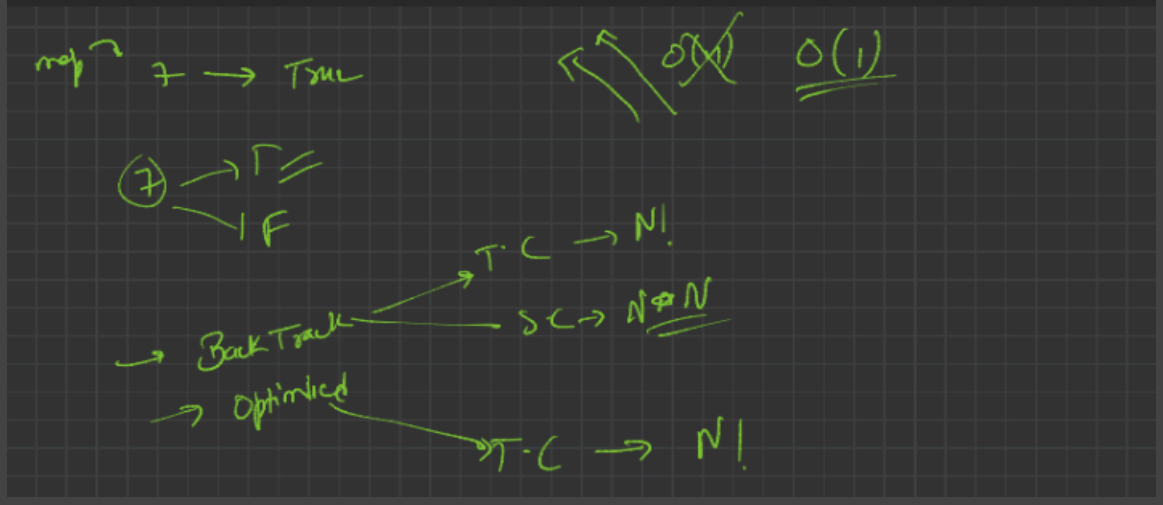

col
diagnol
row+1
col+1
Row
row-1
col-1


Optimisation
O(n)
O(1)
Row
O(n)
O(n)
O(1)
Map
col
Row
map → <row val, bool>
<0, 1>
<2, 1>
<1, true>
map[1] → T/F

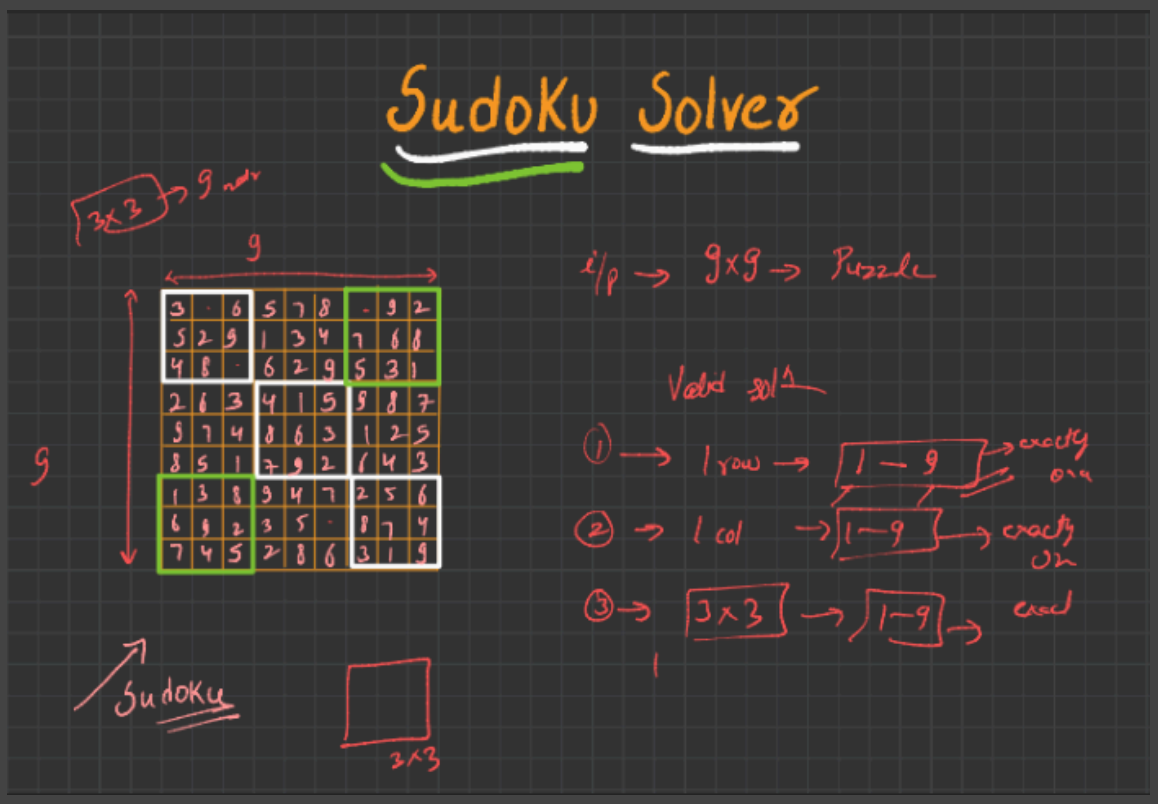
Page 7 / 23

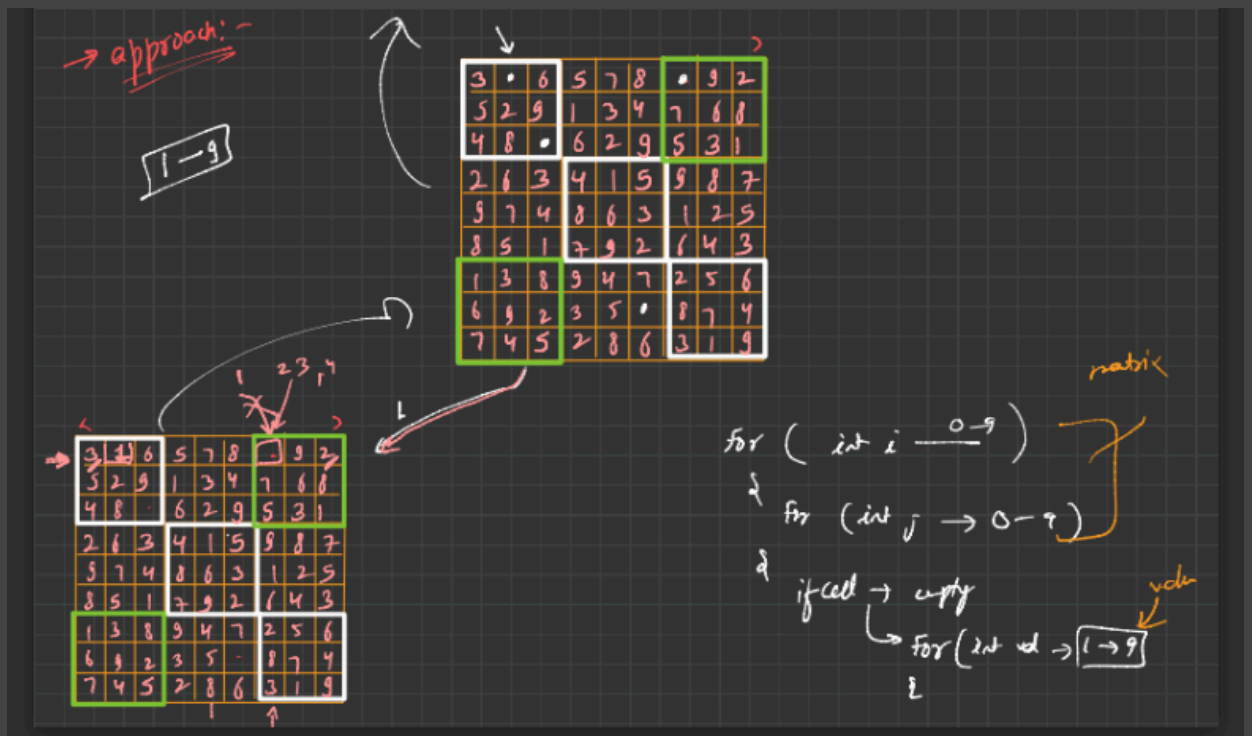

row
col
map <(row+col) → True/False>
map <7 → True>
7 → TRUE
O(n)
O(1)

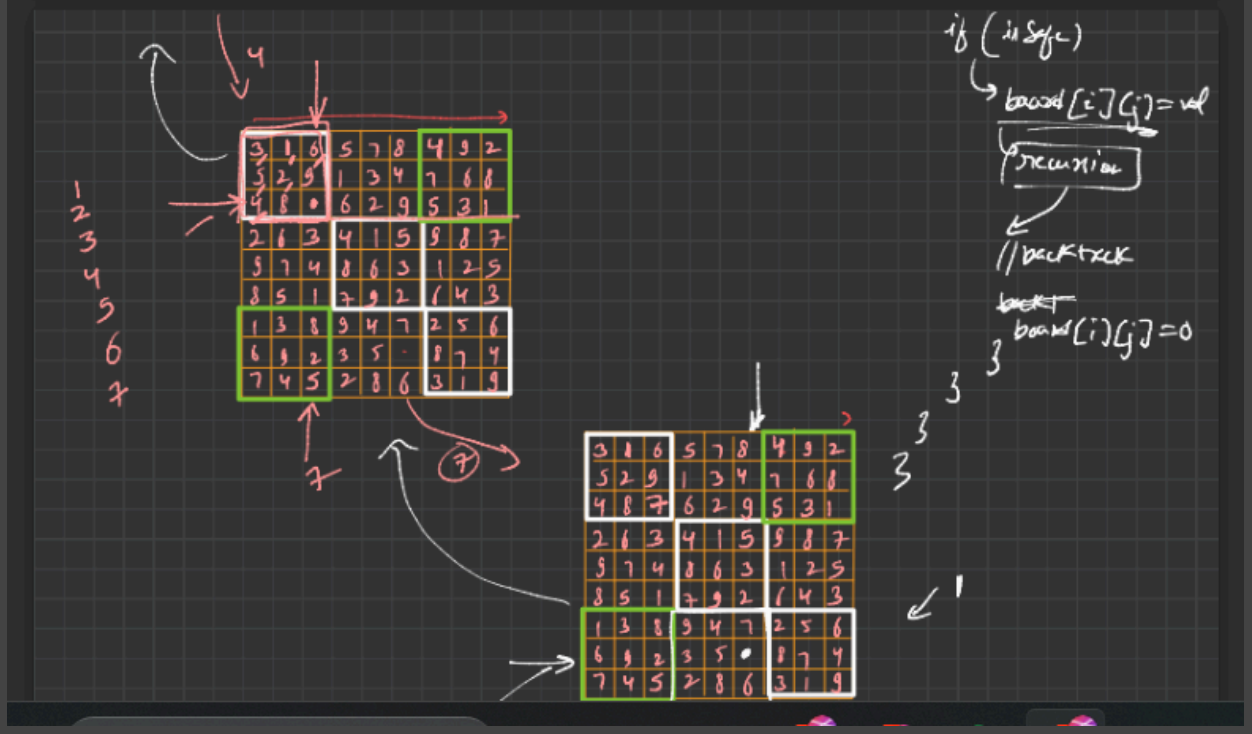


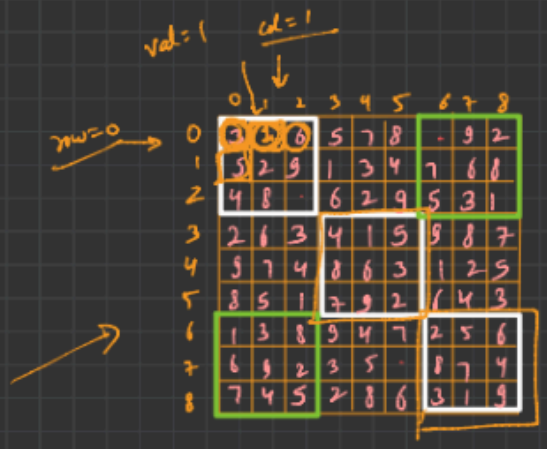


Sudoku Solver Problem:




approach:-
1-9
matrix
for (int i → 0-9)
for (int j → 0-9)
if cell → empty
for (int val → 1 → 9


if (isSafe)
board[i][j] = val
recursion
//backtrack
board[i][j] = 0



for(i → 0 → <9)

Dry run

3*(row/3) + i/3

3*(col/3) + i%3

i=0 → 3×0/3 + 0/3 = 0 (0,0)

3×(1/3) + 0%3 = 0

i→1 3*(0/3) + 1/3 = 0 (0,1)

3×(1/3) + 1%3 = 0+1 = 1

i→3 → 3*0/3 + 3/3 = 0+1 = 1 (1,0

3×(1/3) + 3%3 = 0+0 = 0

i=2 → 3×(0/3) + 2/3 = 0 (0,2)

3×(1/3) + 2%3 = 0+2 = 2

T.C → 9^m m → no of empty cells

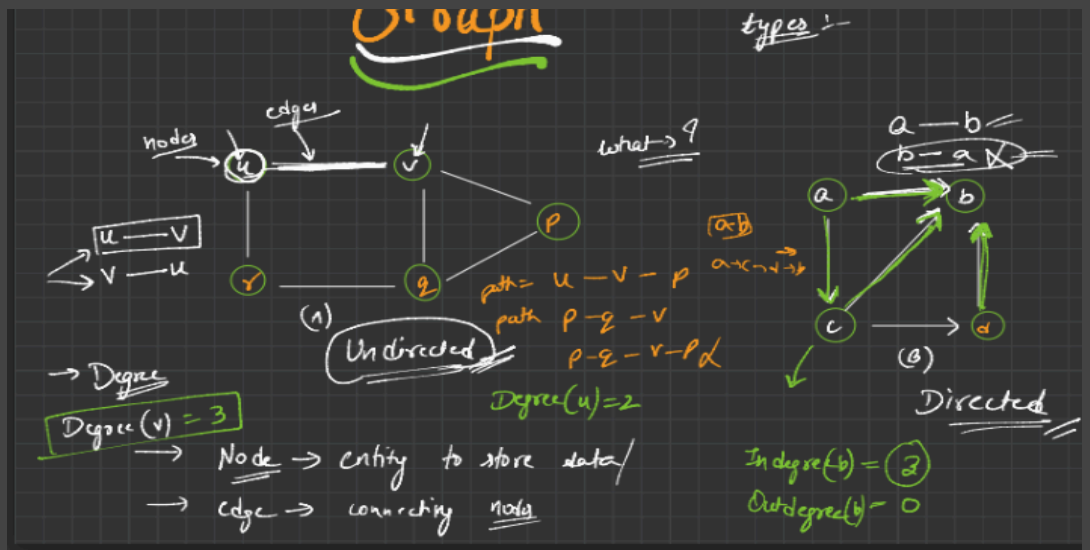
S.C → O(1)

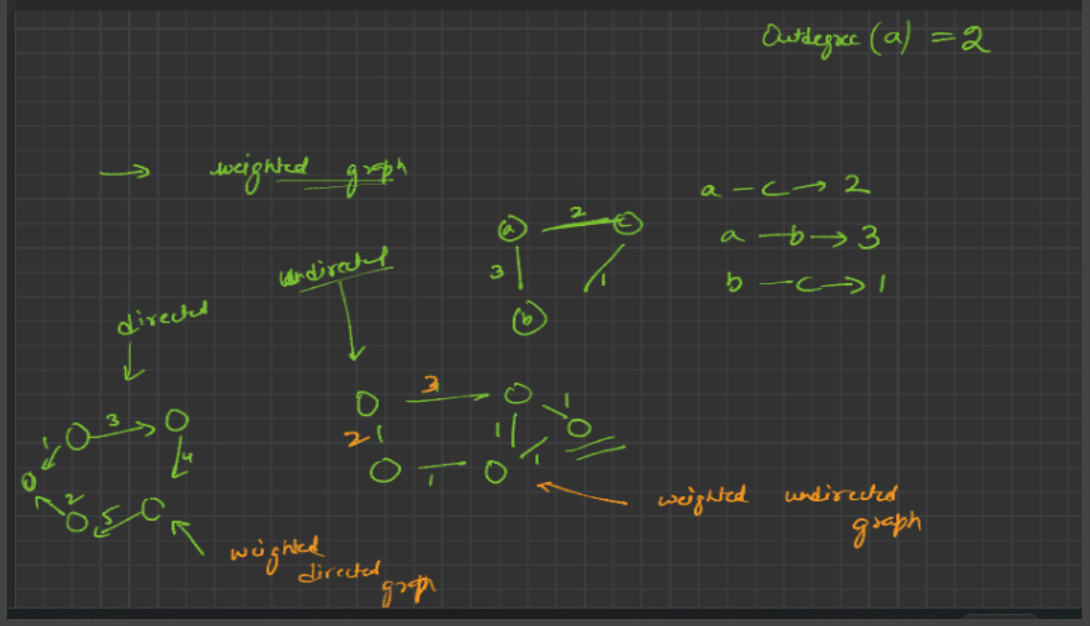
9×9

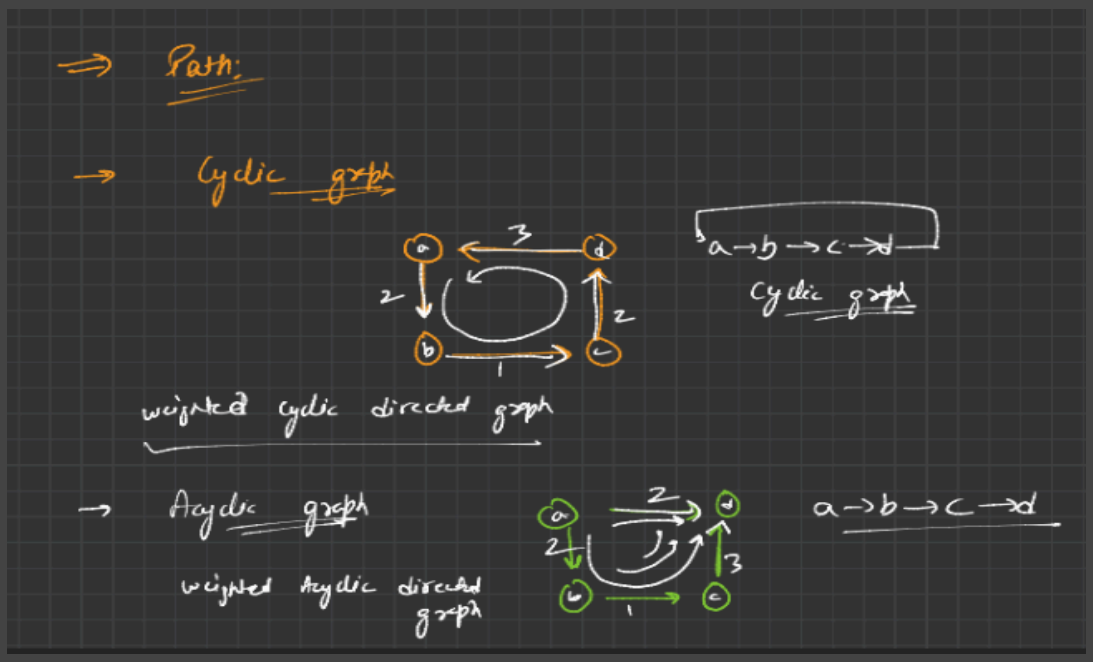
9×9 = 81

Graph

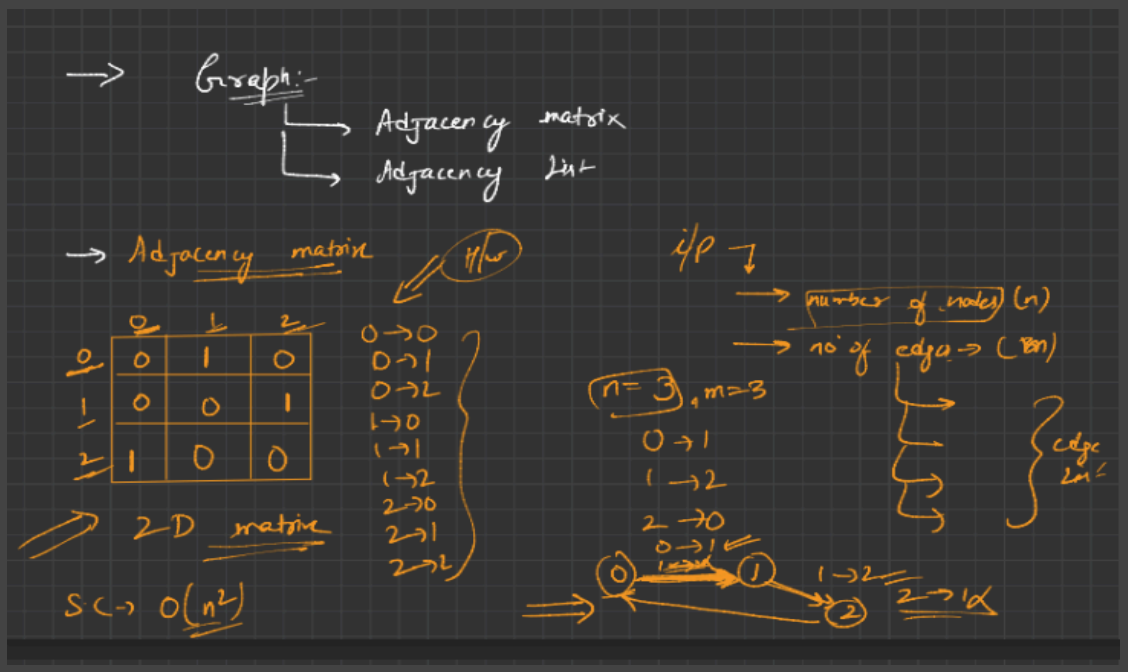
Introduction:



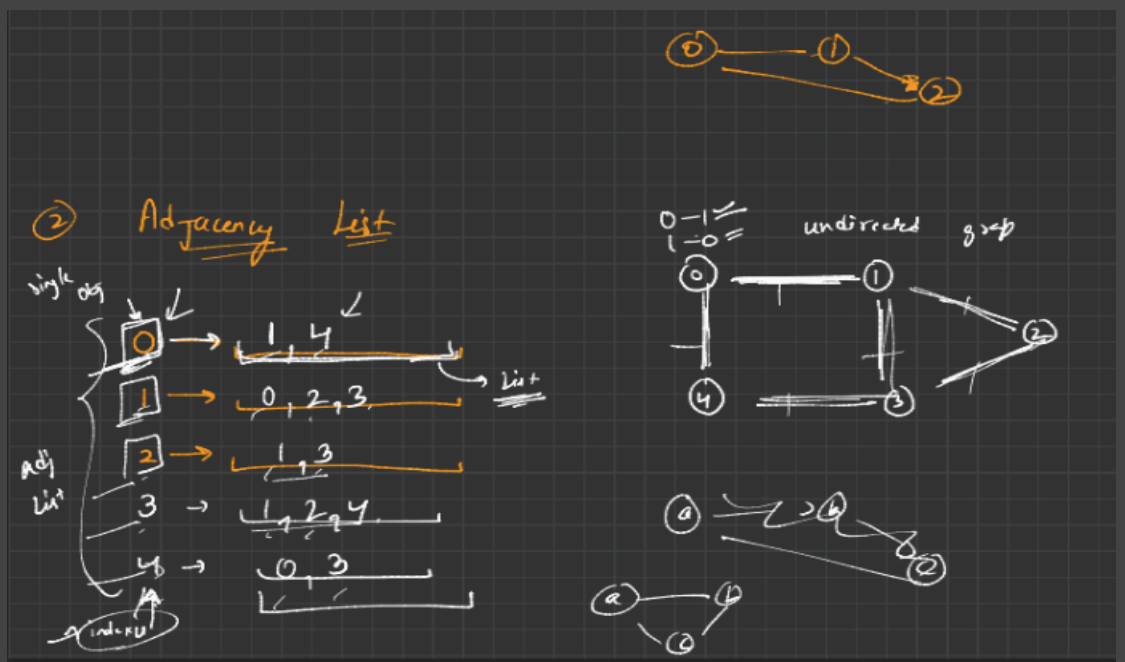


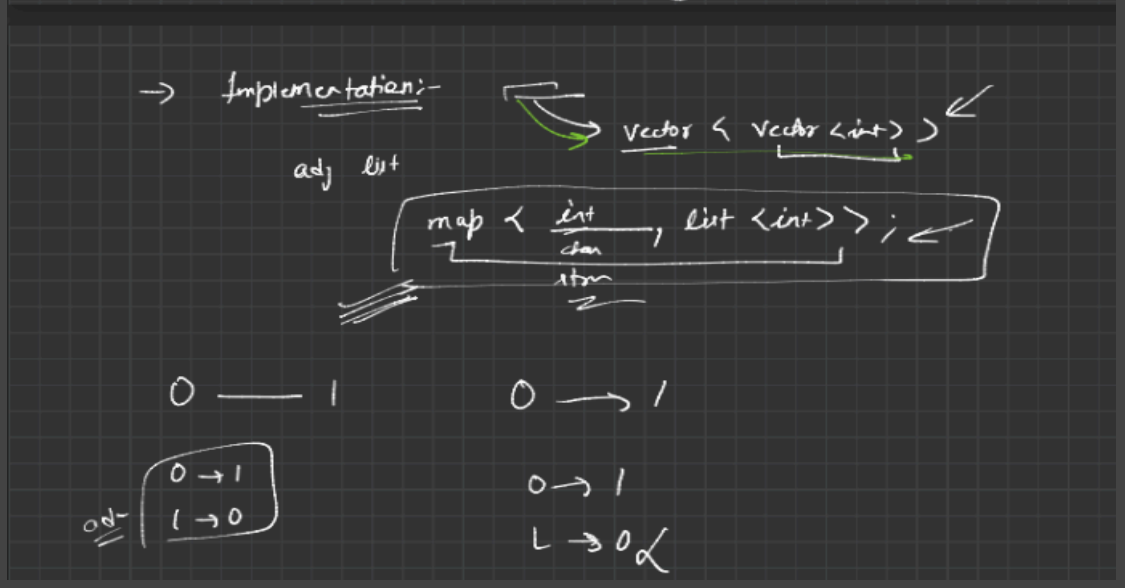


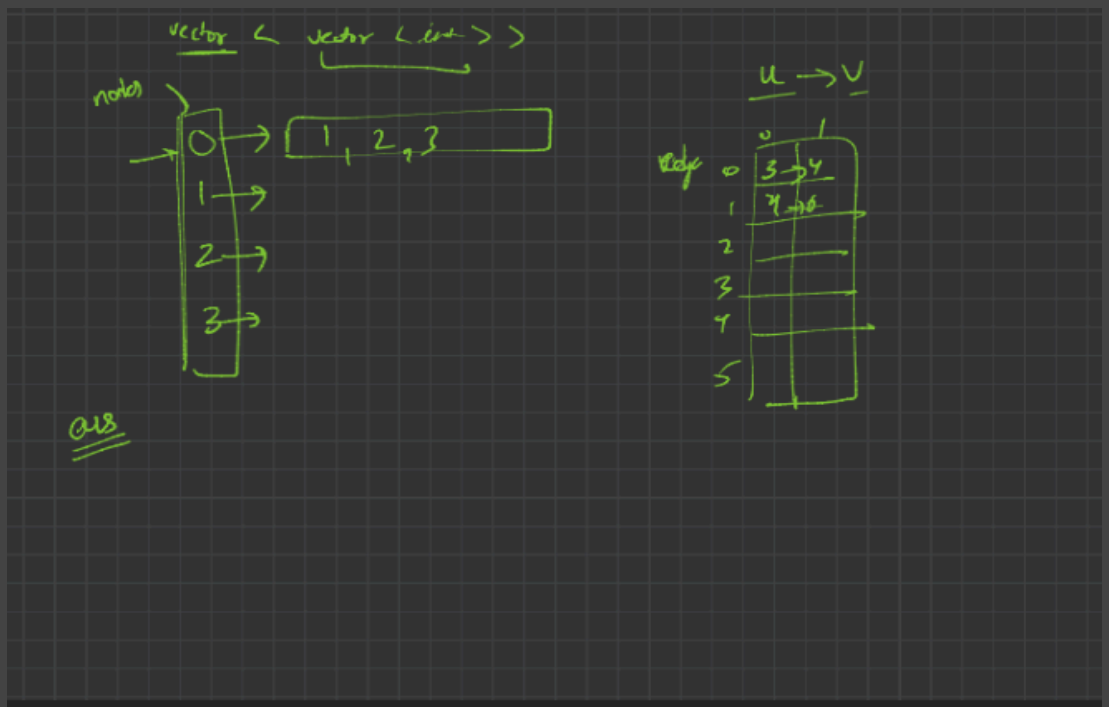
Adjacency Matrix:

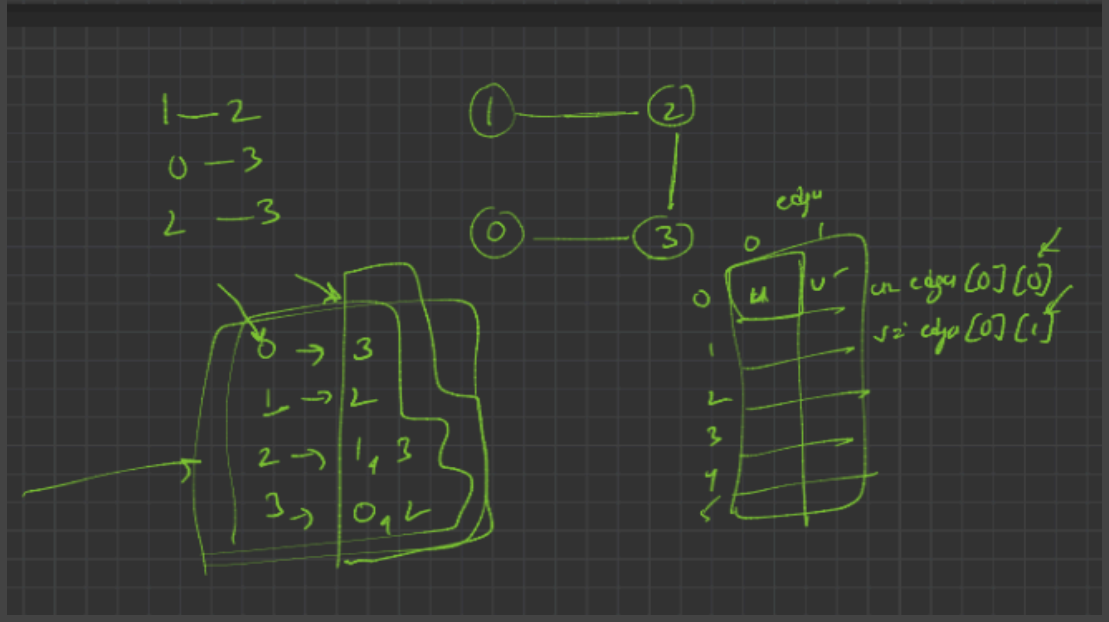


Adjacency List:

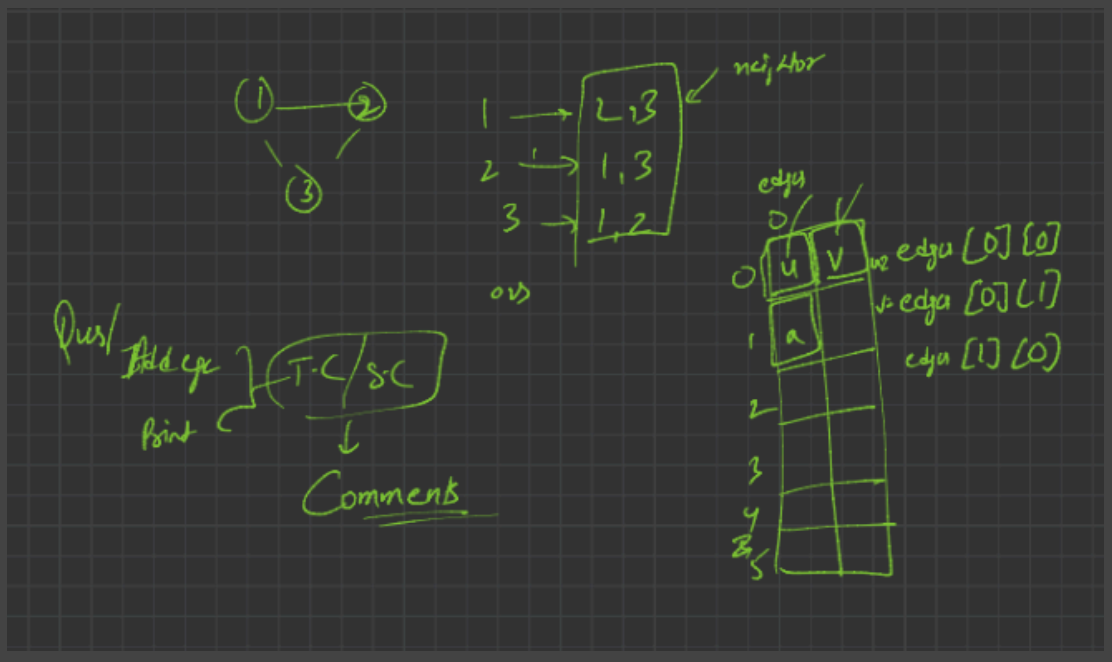




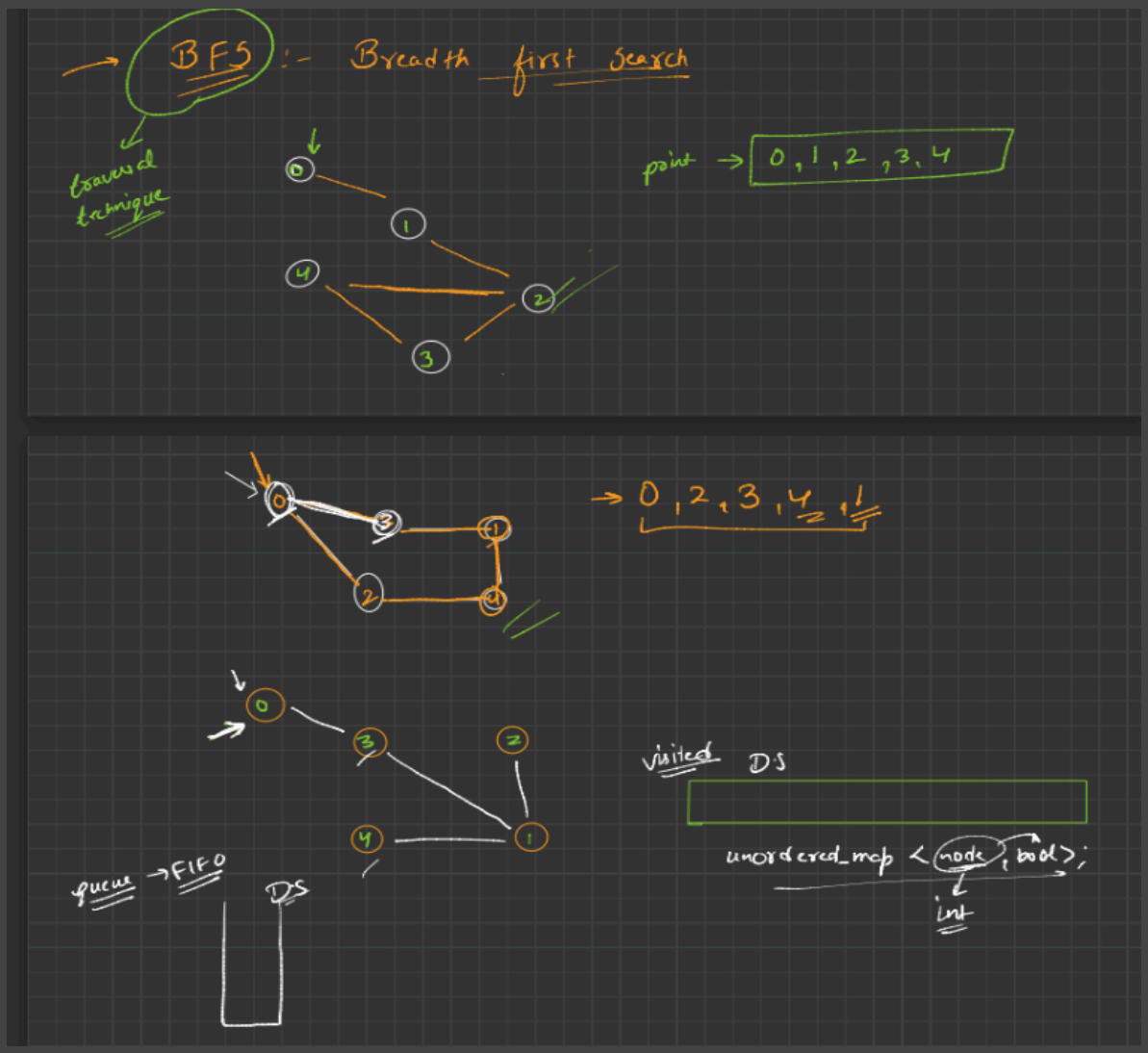


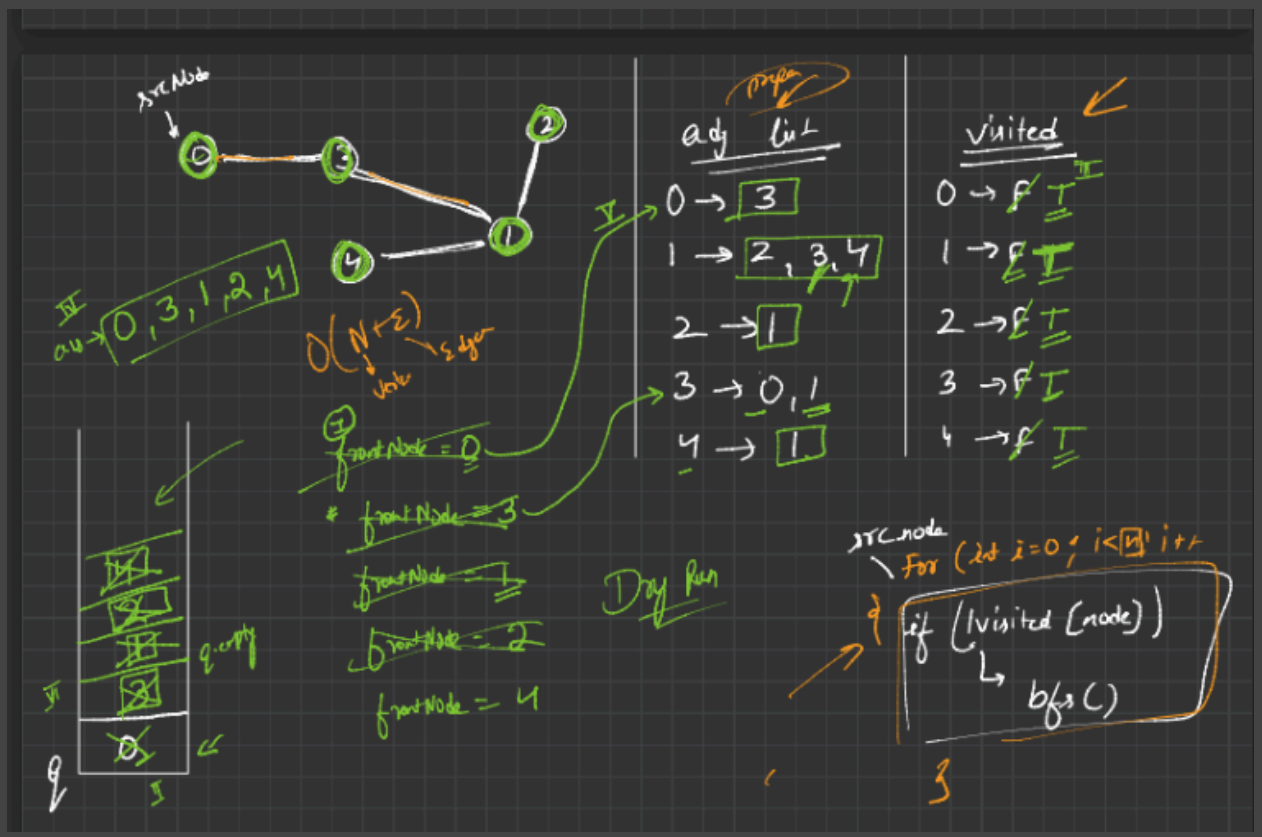


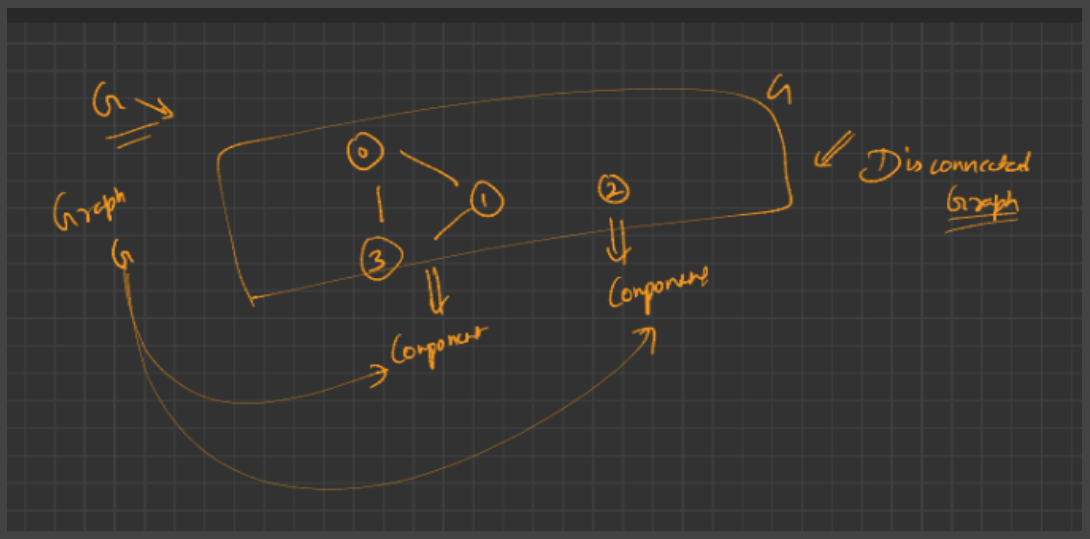
Creating And Printing A Graph:



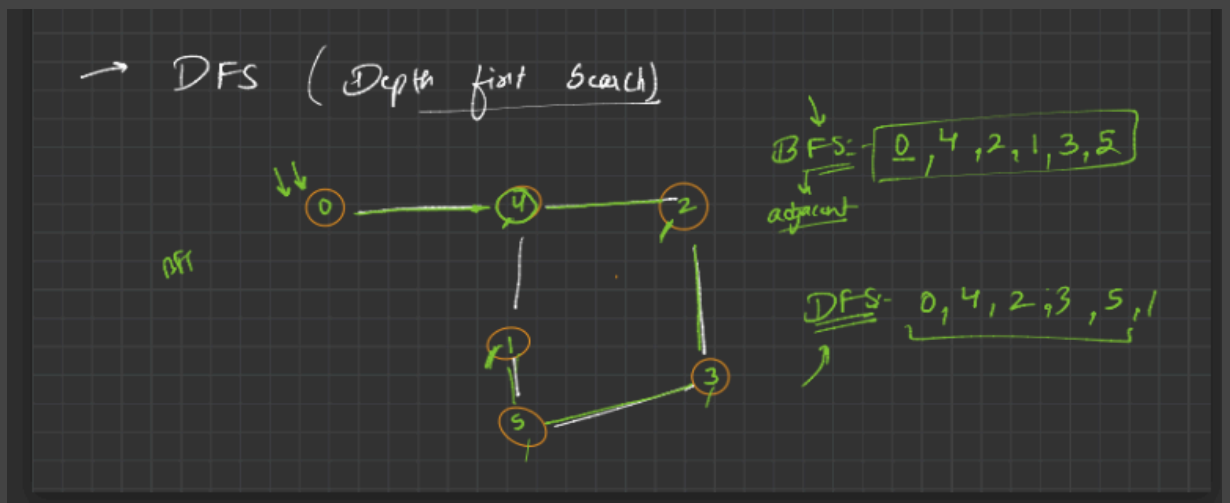
Breadth First Search(BFS) Traversal:

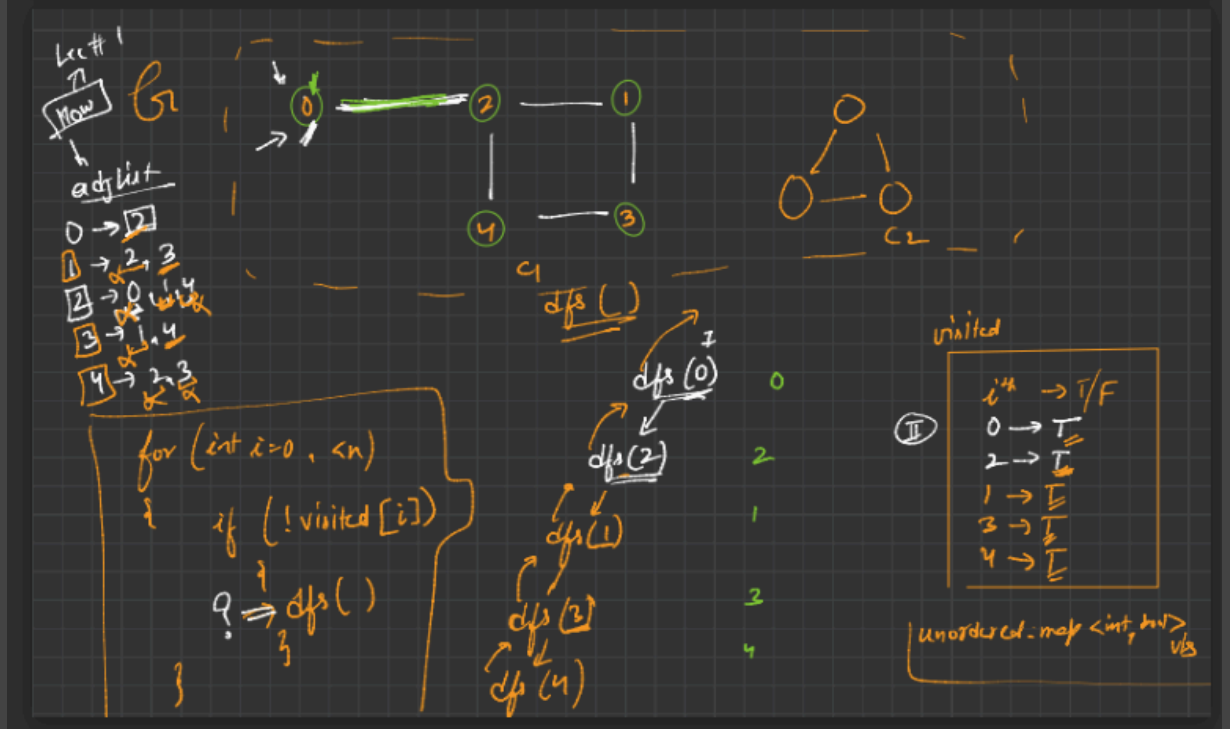


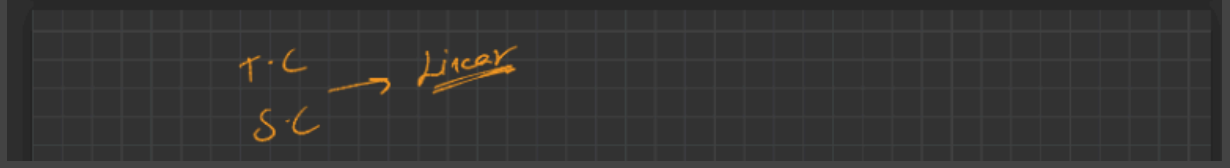




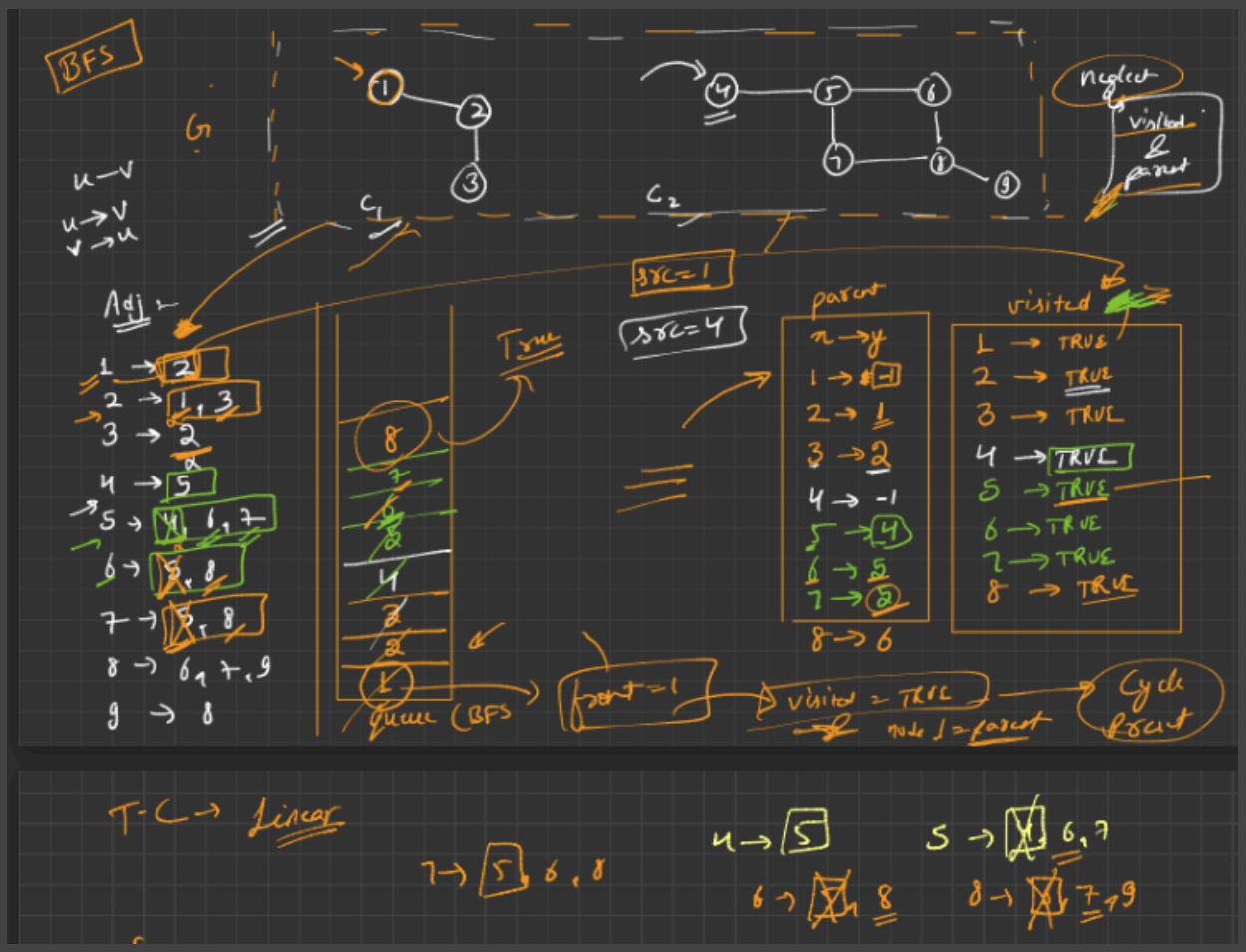
Depth First Search(DFS) Traversal:



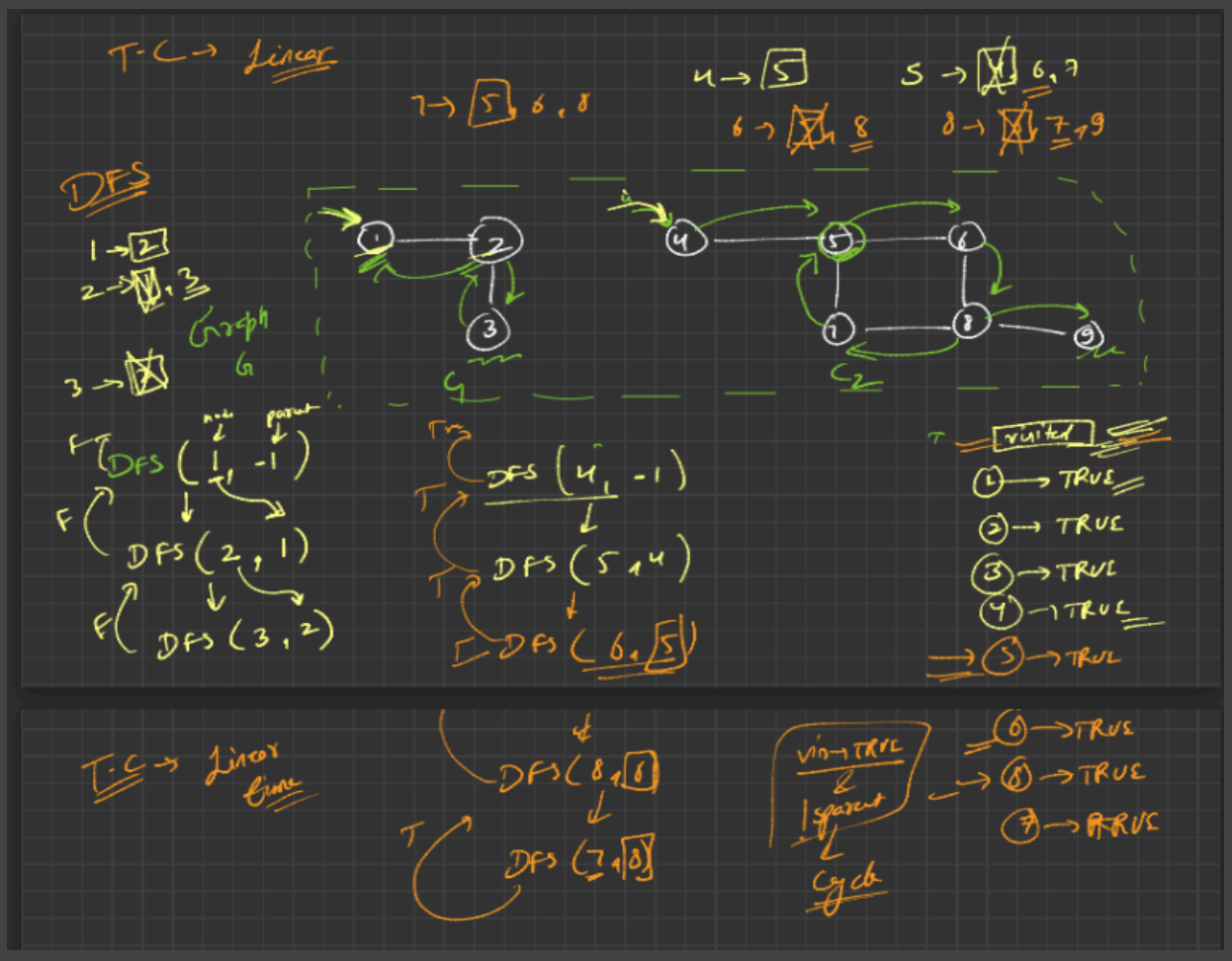




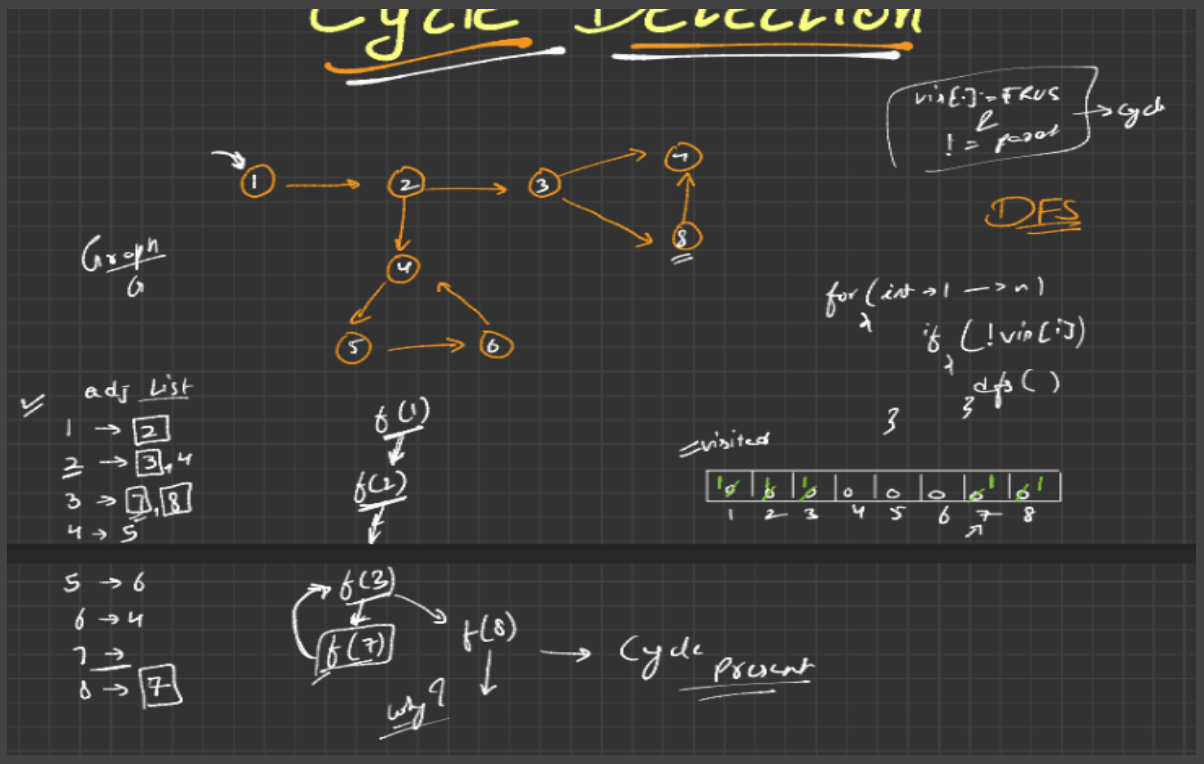
Cycle Detection In Undirected Graphs(BFS):



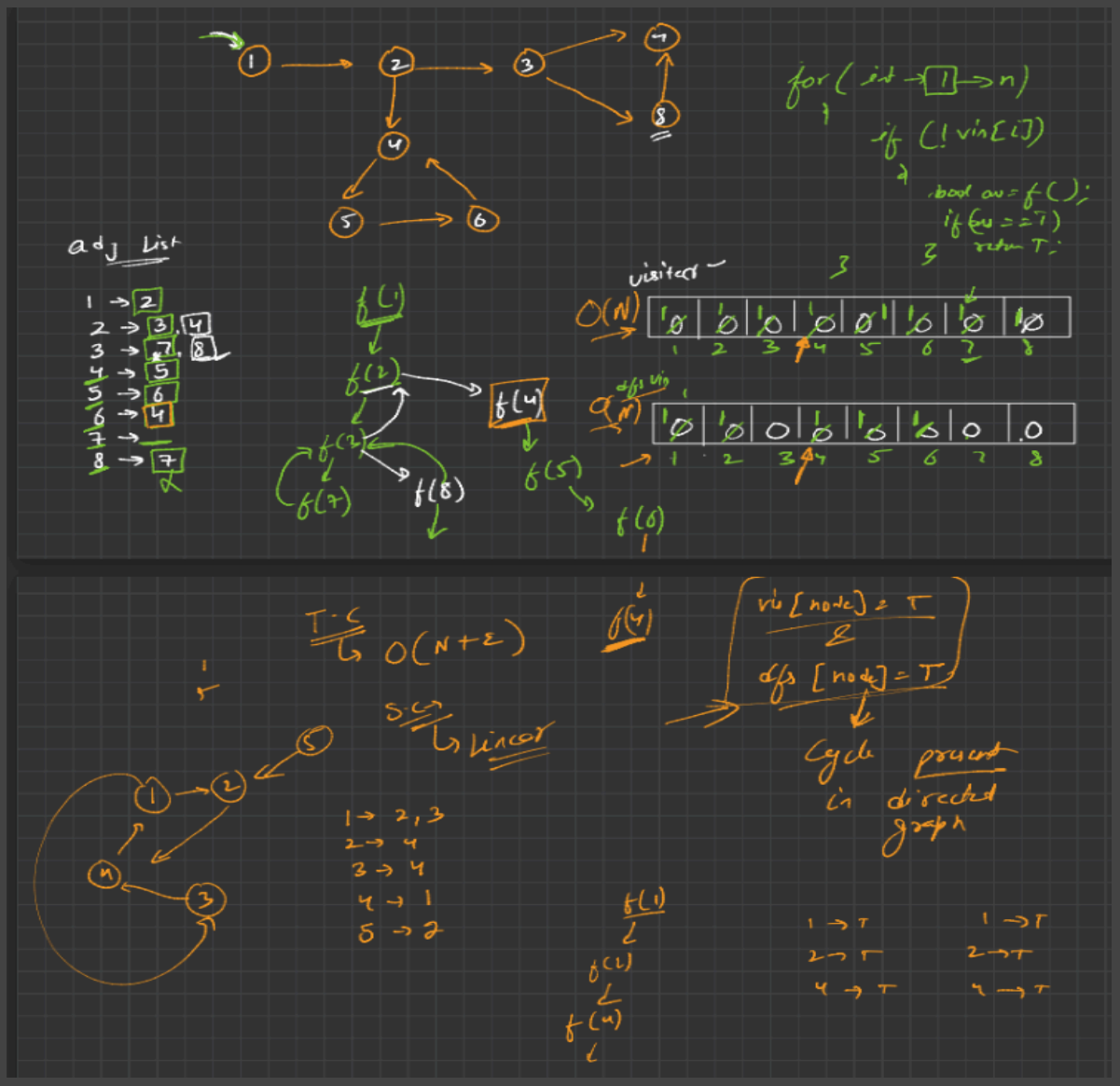
Cycle Detection In Undirected Graphs(DFS):



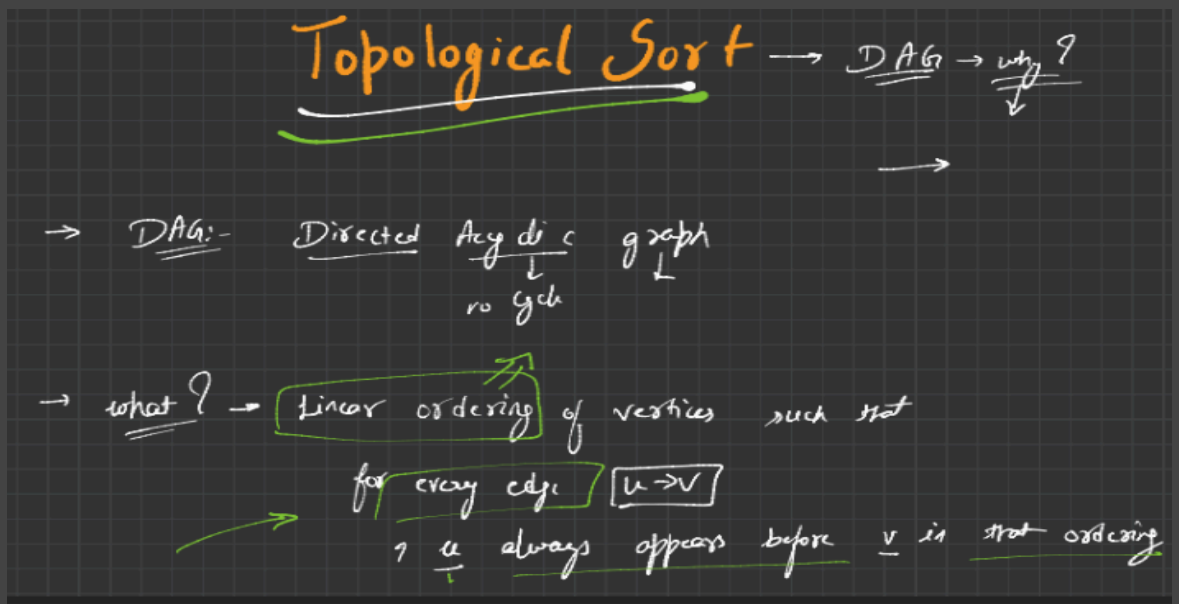
Cycle Detection In Directed Graphs(DFS):

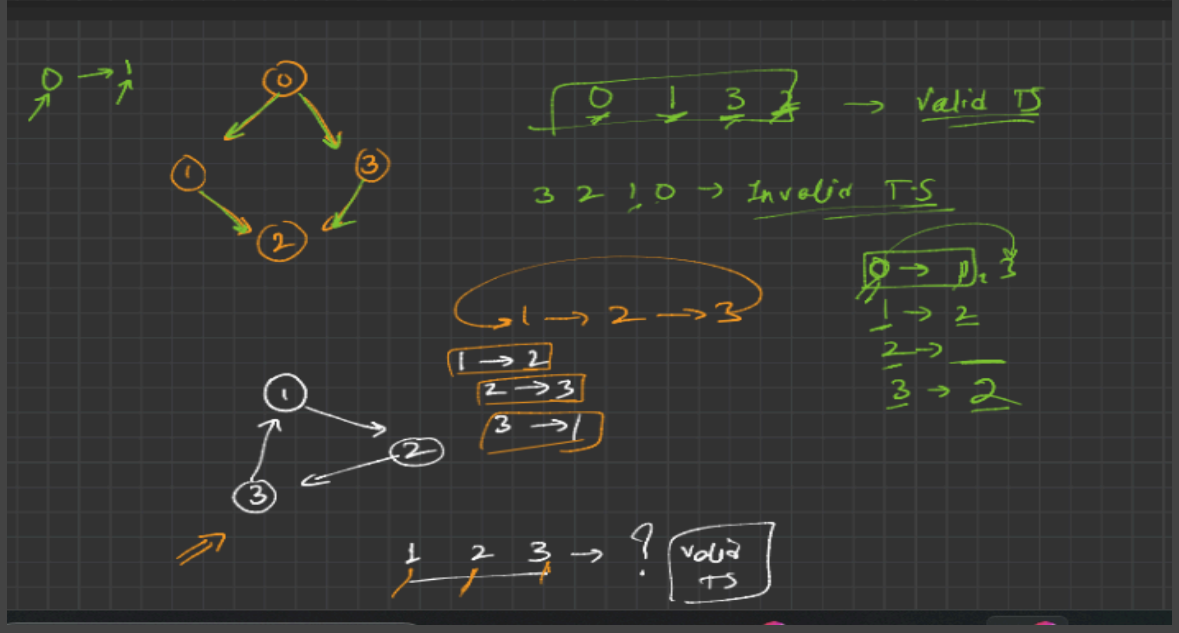


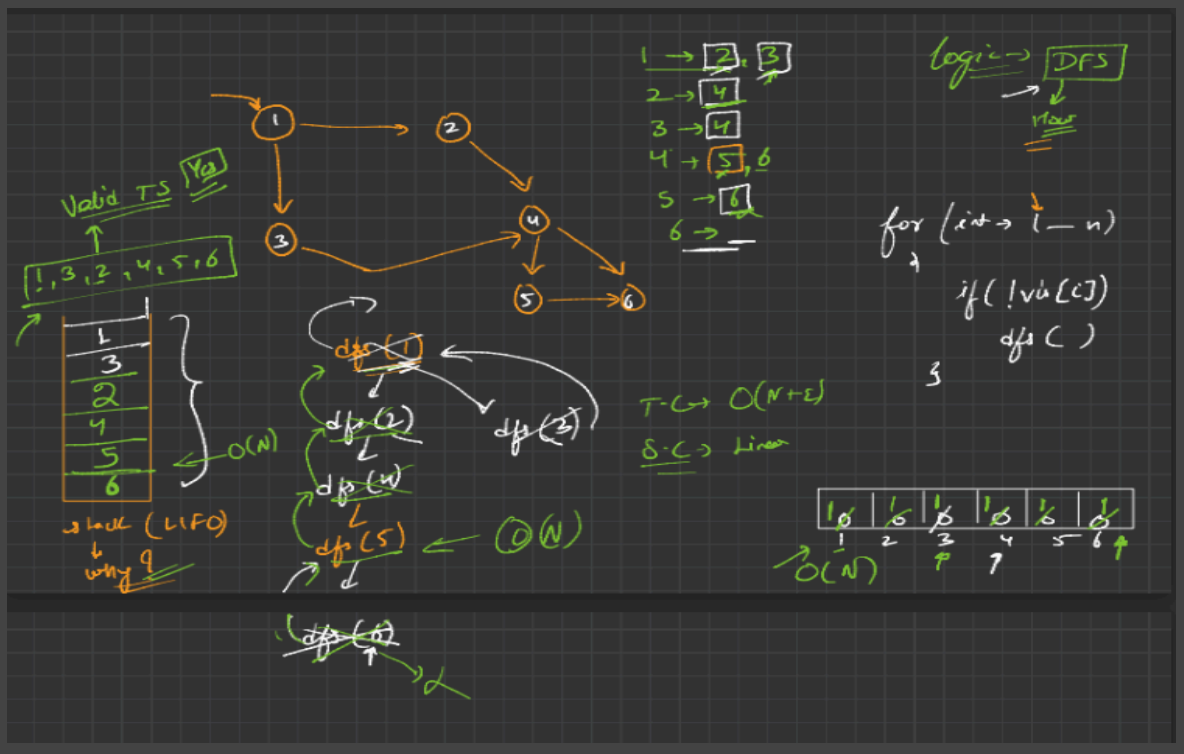
The above method will not work so, we go with the below method



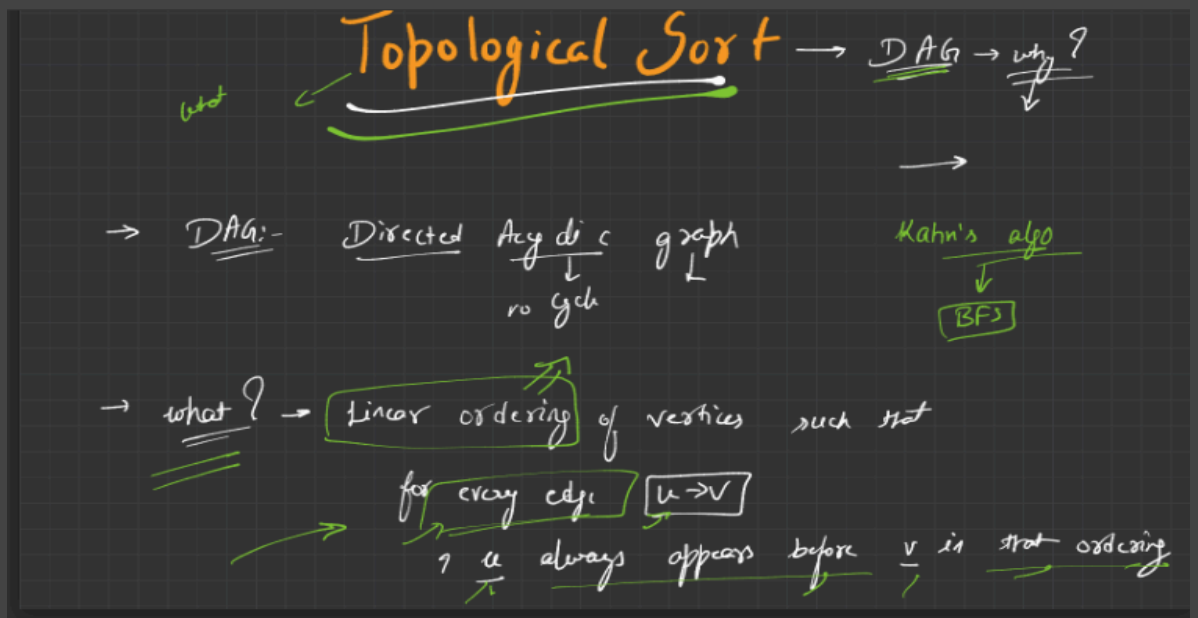
Topological Sort Using DFS:

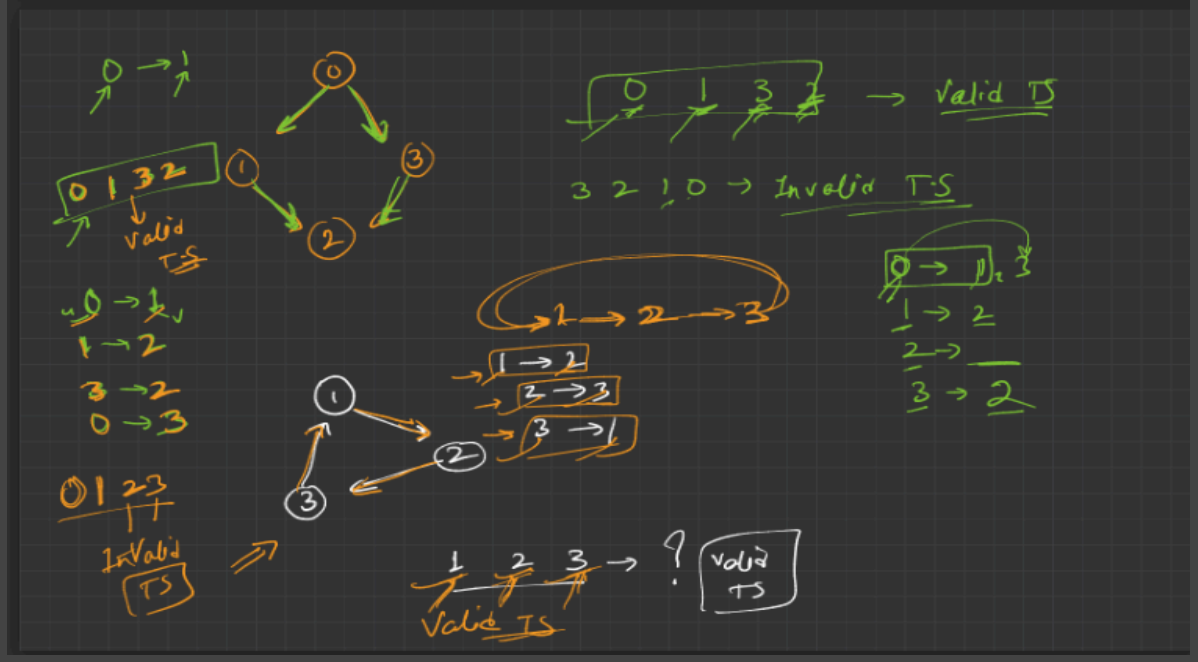


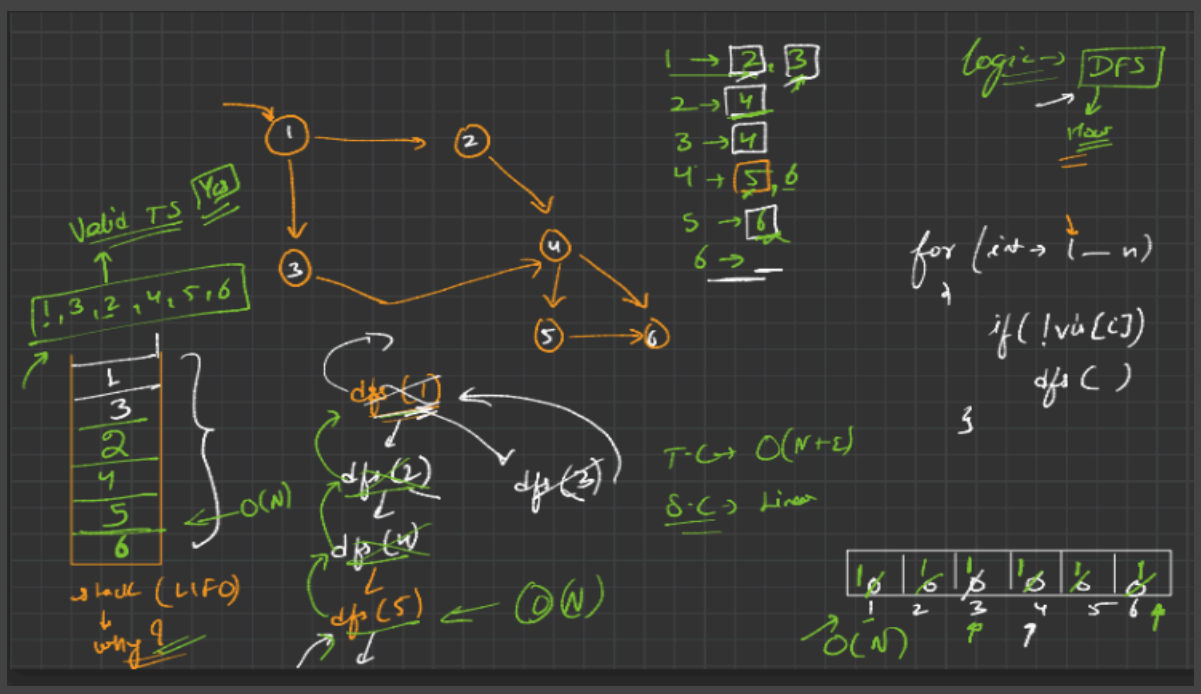


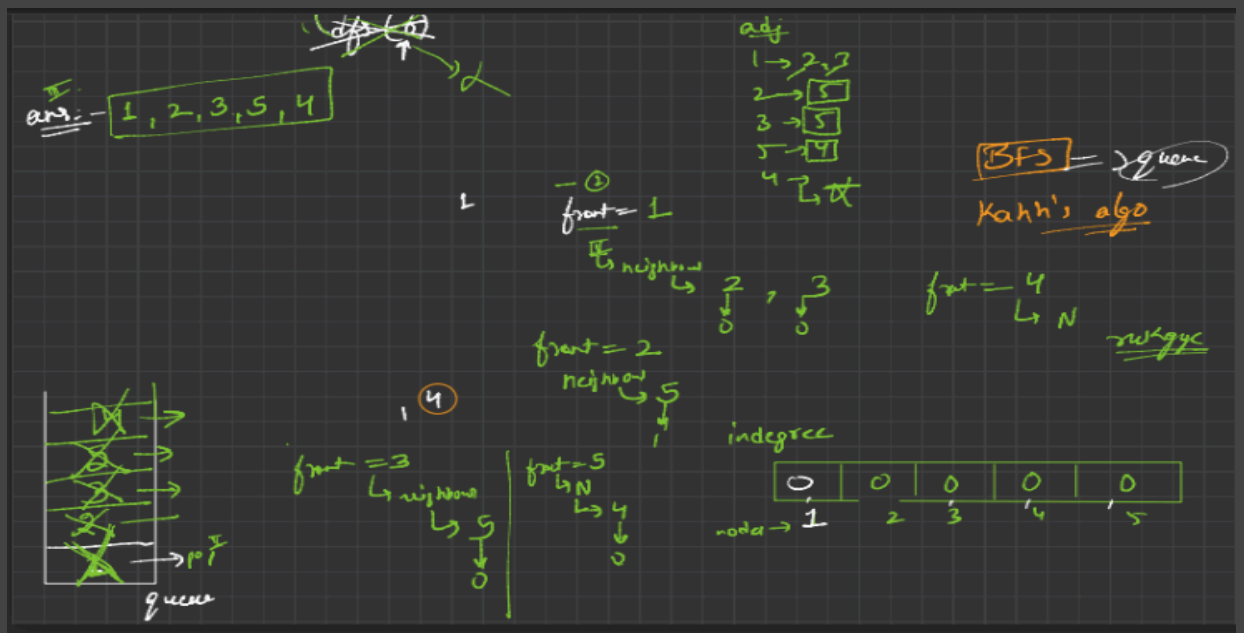


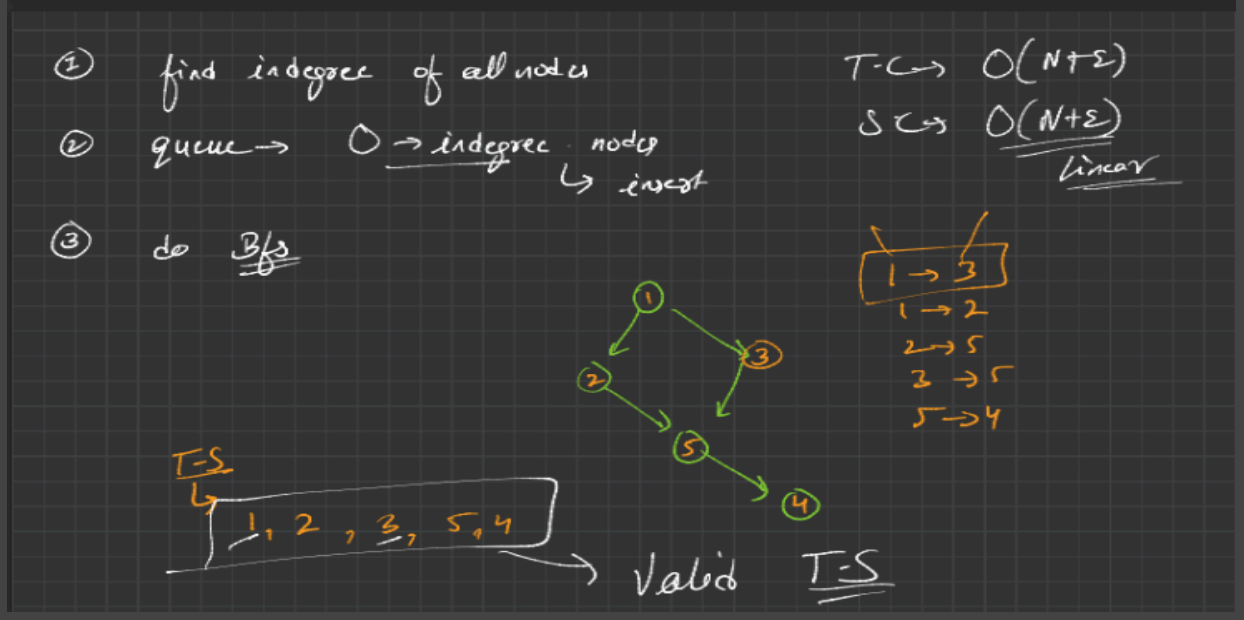
Topological Sort Using Kahn's Algorithm(BFS):



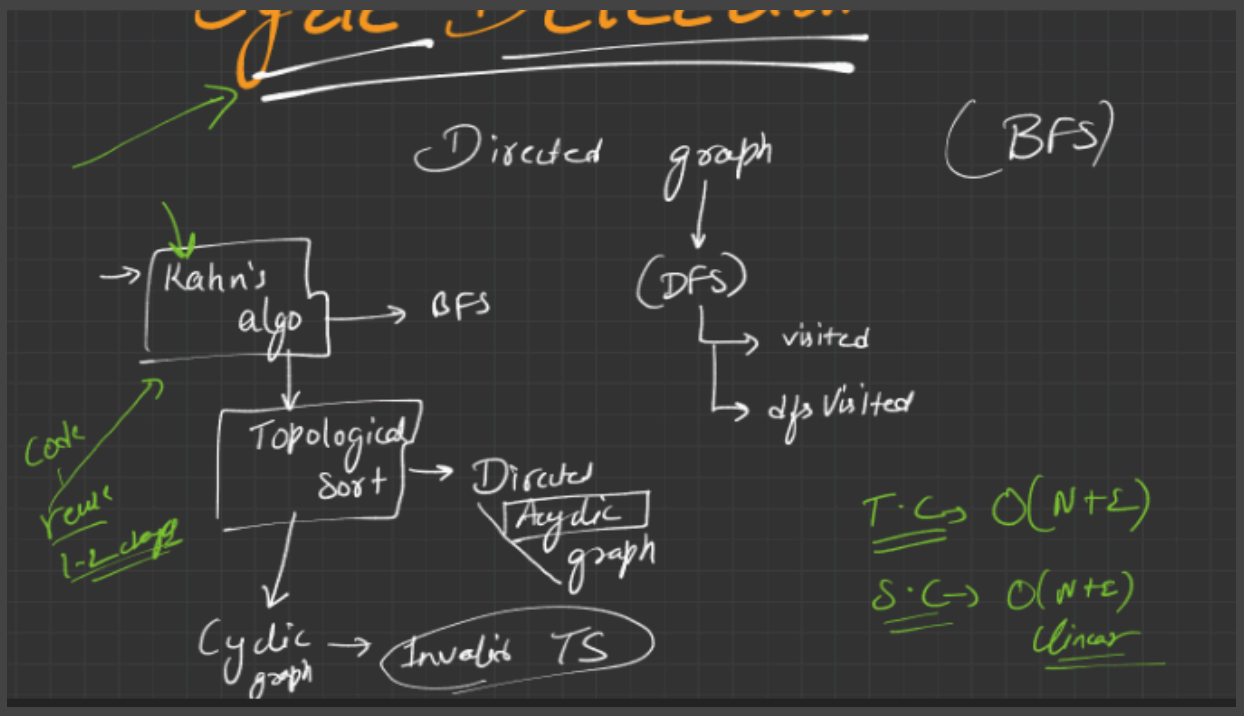




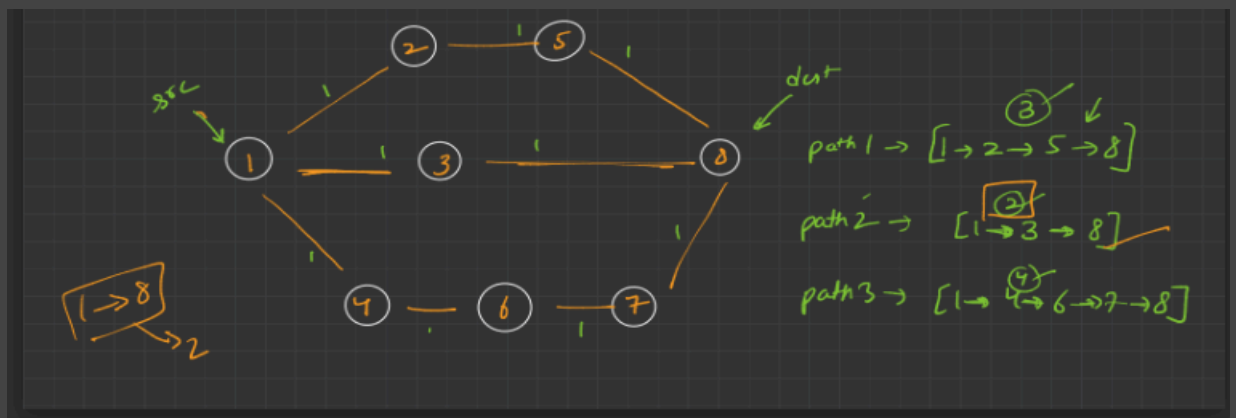


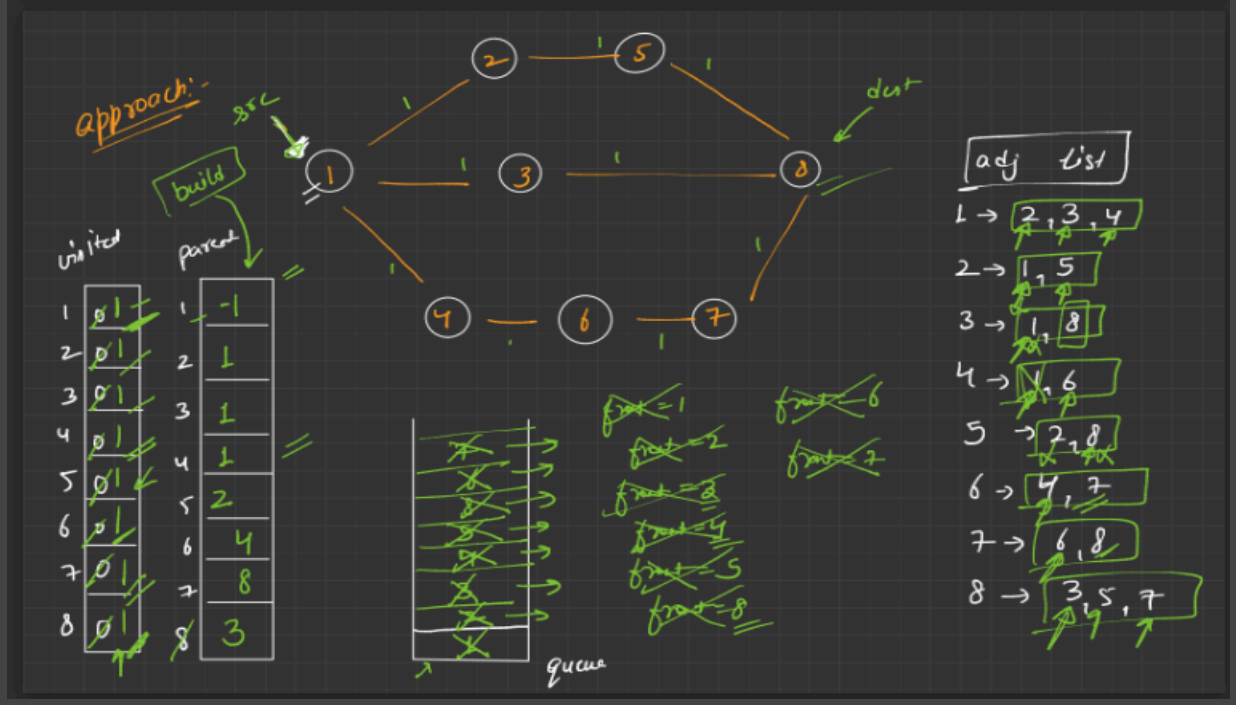


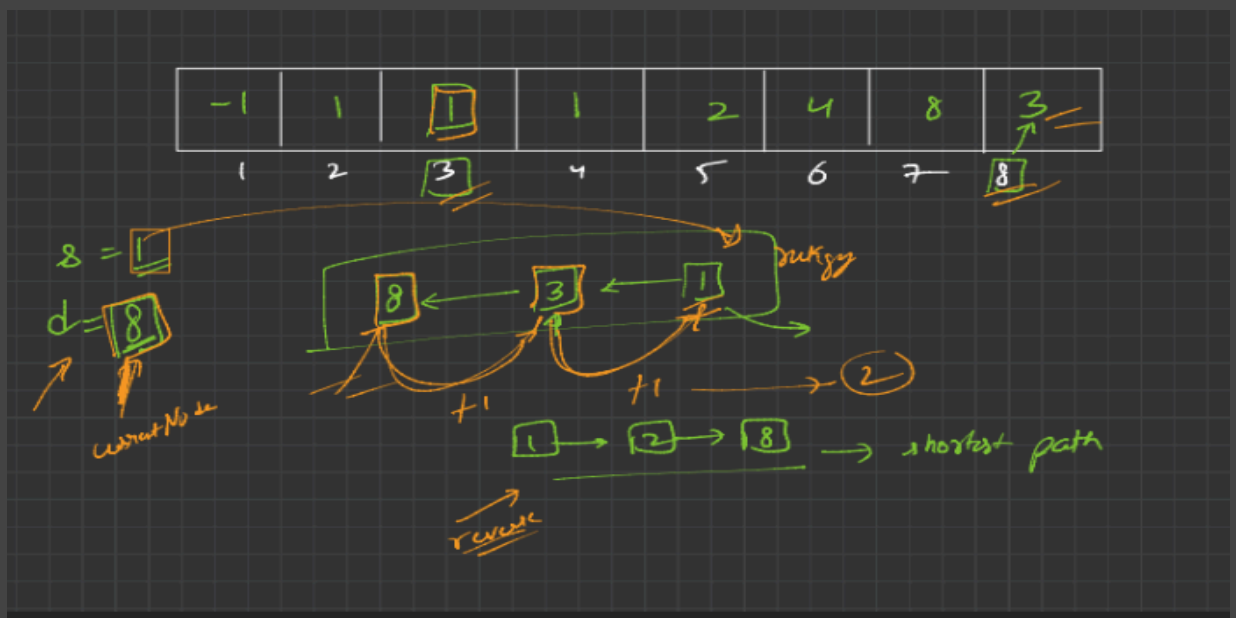
Cycle Detection Using BFS:



Shortest Path In Undirected Unweighted Graph Using Modified BFS:



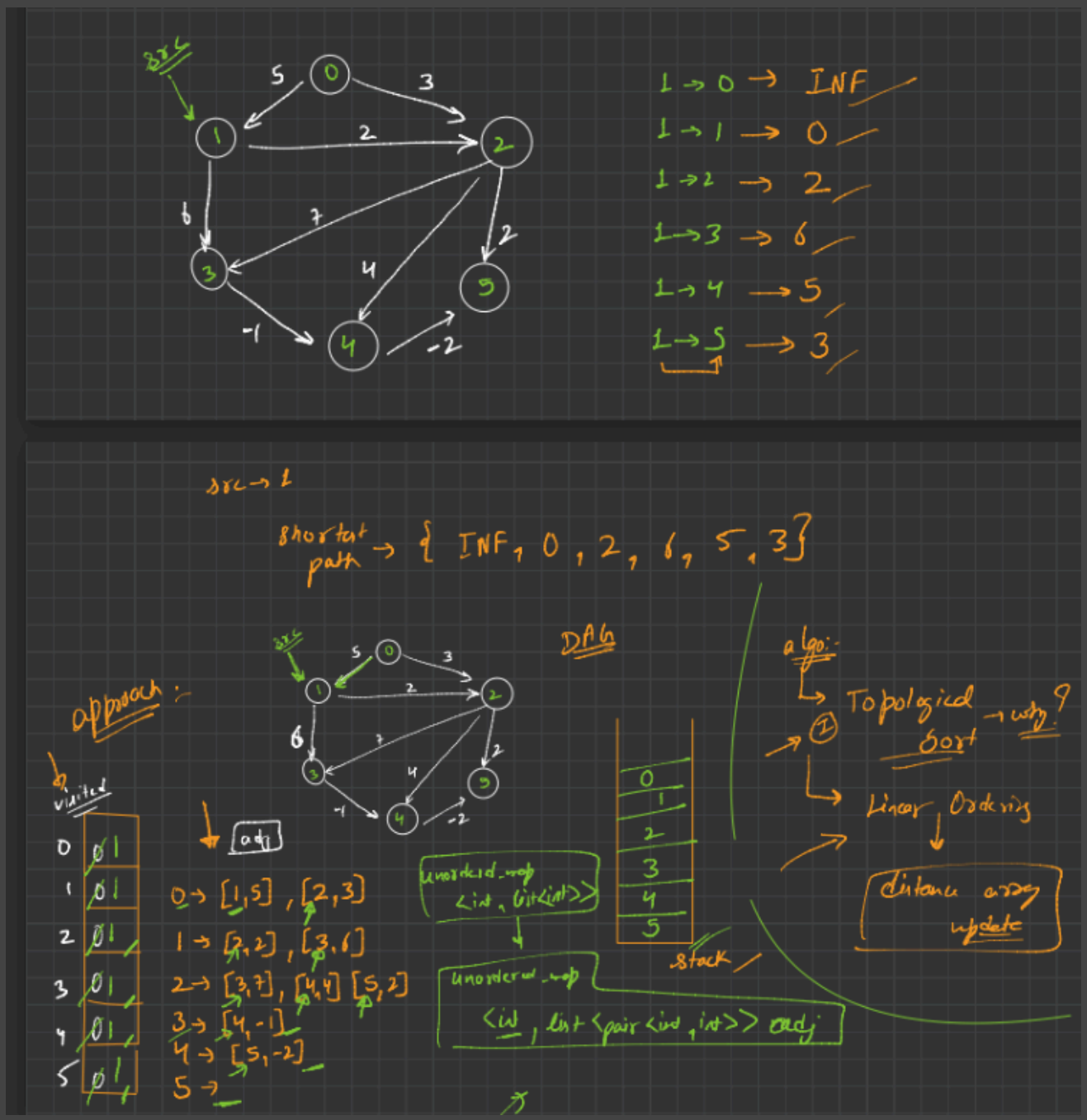


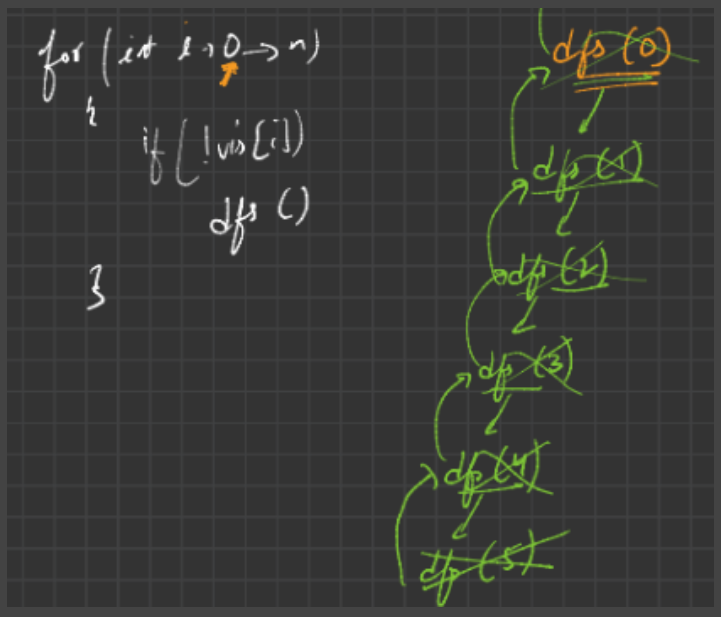


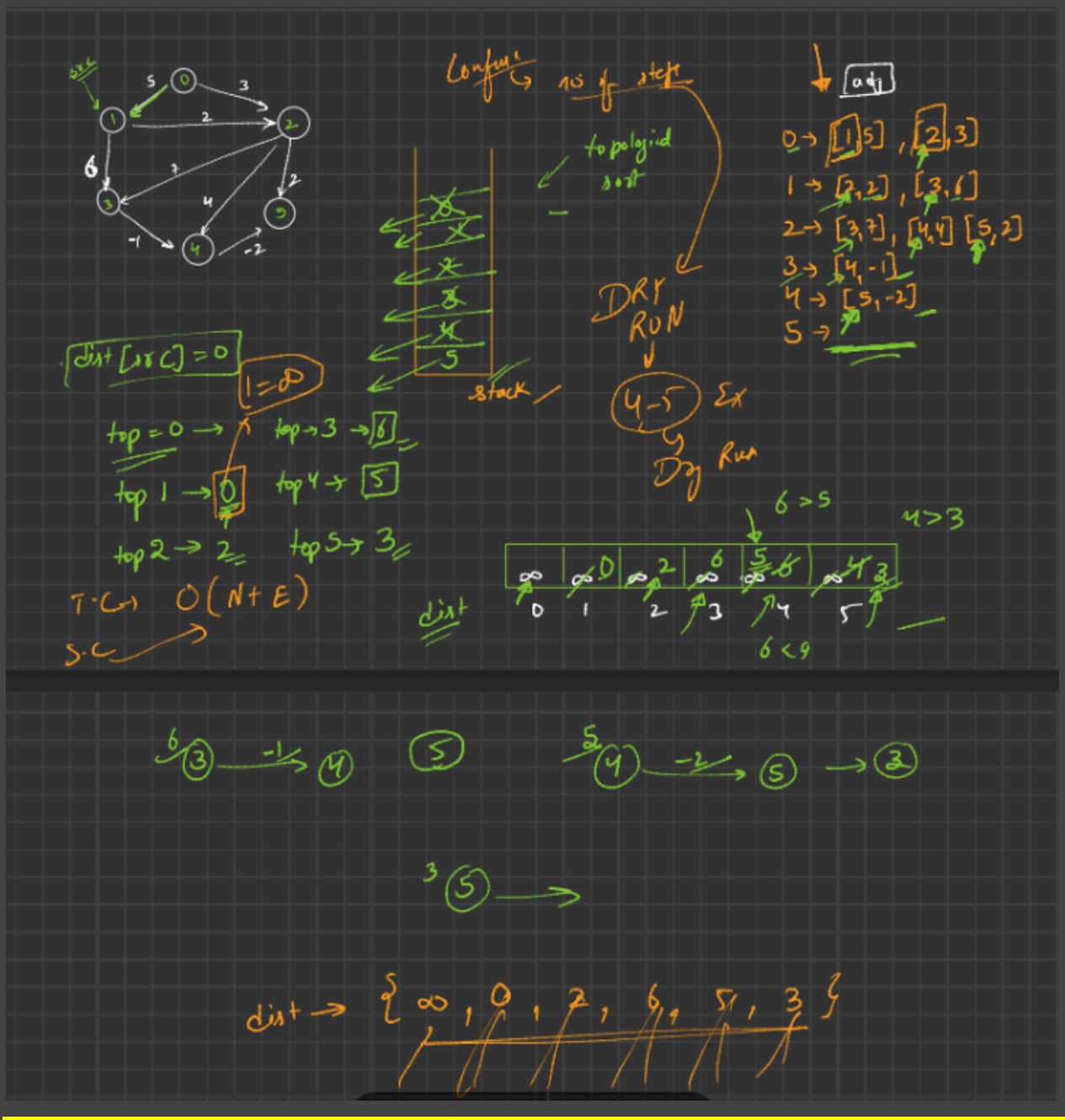
T.C → O(N+E)

S.C → O(N+E)

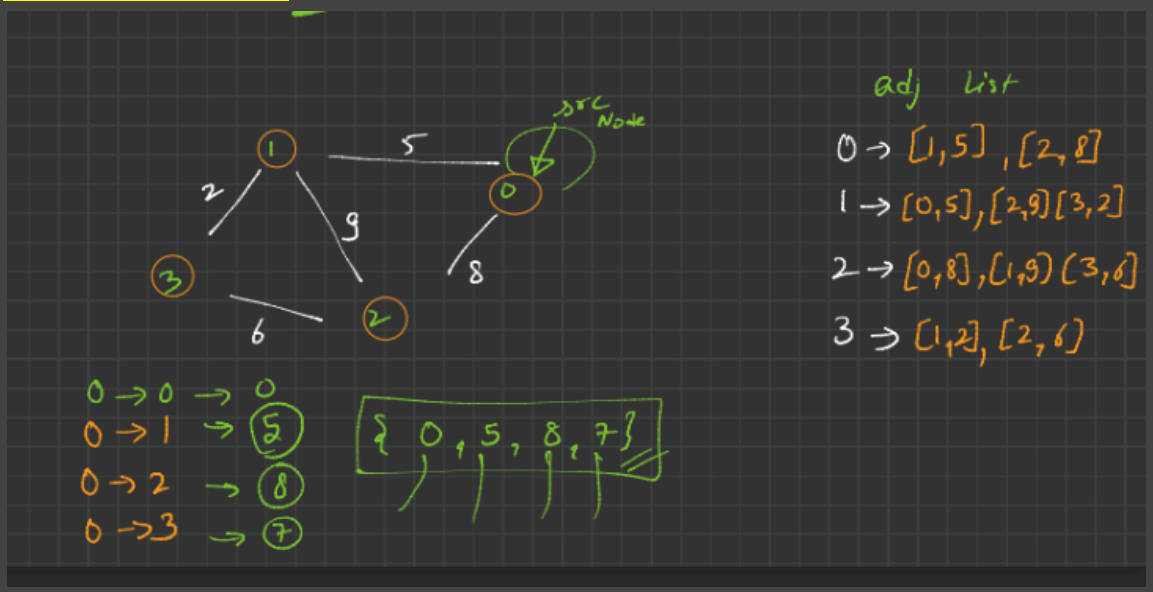
Shortest Path In Directed Acyclic Graph(DAG) Using DFS(Modified):







Dijkstra's Algorithm / Shortest Path In Undirected Weighted(excluding negative weights) Graph:



approach:

src -> 0

si.insert (make_pair(0,0));

DS -> priority queue, set

top -> (0, 0) -> topNode

NodeDistance

top -> (5, 1) -> topNode

NodeDistance

(dist, node)

pair<int, int> set

adj list

0 -> [1,5], [2,8]

1 -> [0,5], [2,9] [3,7]

2 -> [0,8], [1,9] [3,6]

3 -> [1,7], [2,6]

INT_MAX

dist

0	5	8	7
0	1	2	3

5 < INT_MAX

7 < 14

node

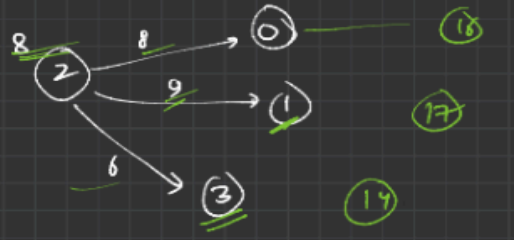
top -> (7, 3) -> topNode

NodeDistance

top -> (8, 2) -> topNode

NodeDistance

0	5	8	7
0	1	2	3



adj

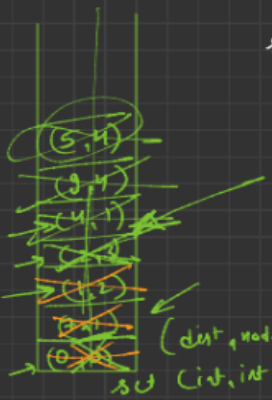
0 -> [1,7], [2,1] [3,2]

1 -> [0,7], [2,3] [3,5] [4,1]

2 -> [0,1], [1,3]

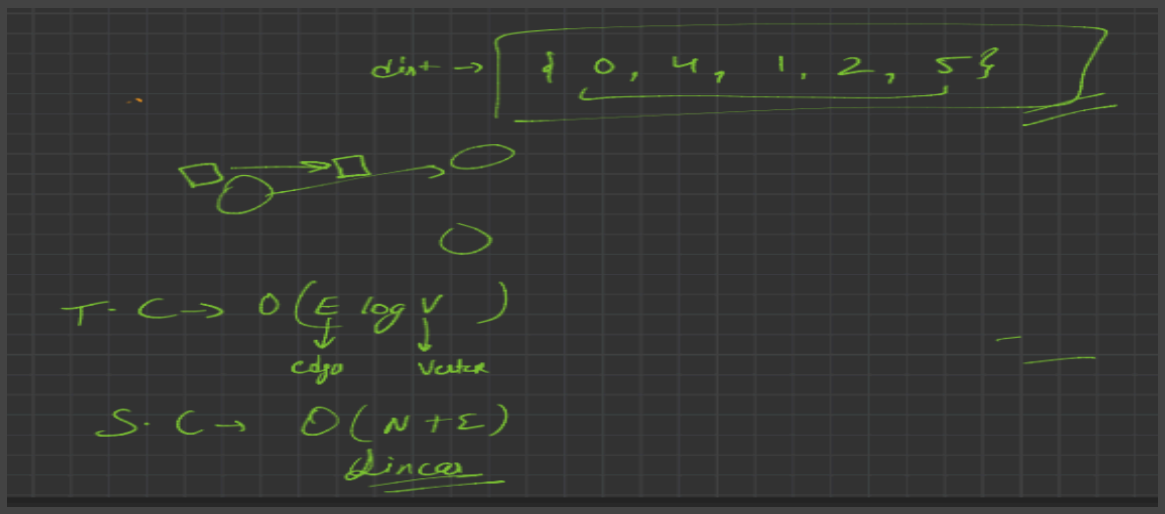
3 -> [0,2] [1,5] [4,7]

4 -> [1,1] [3,7]

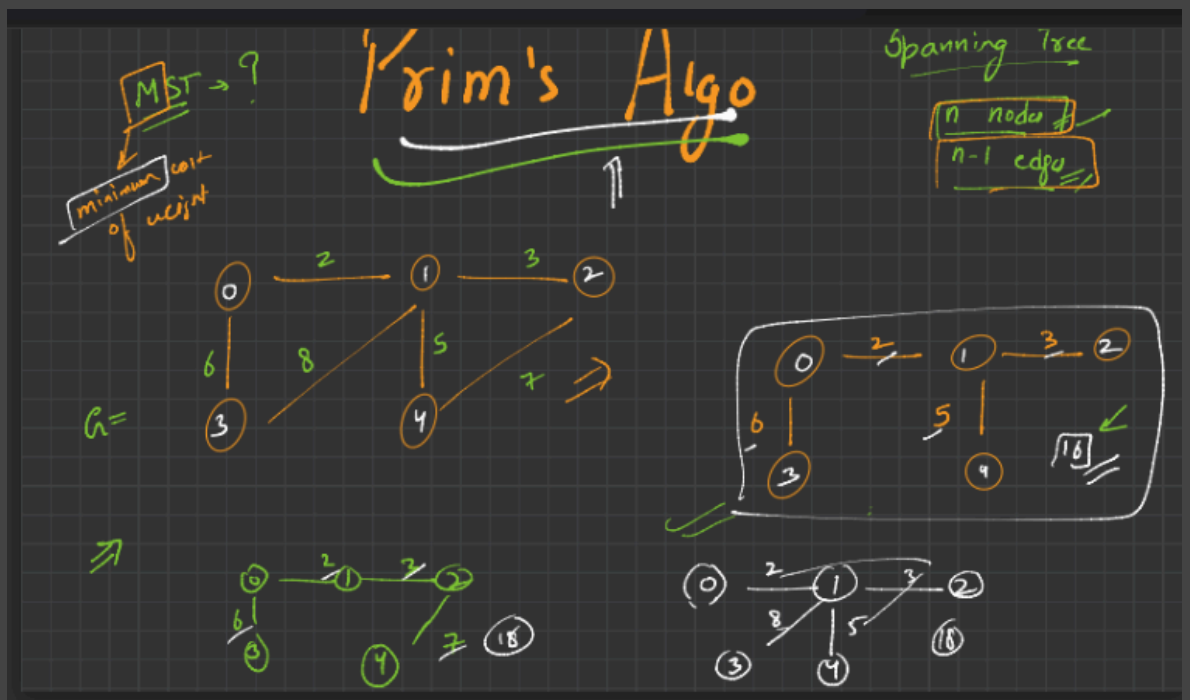


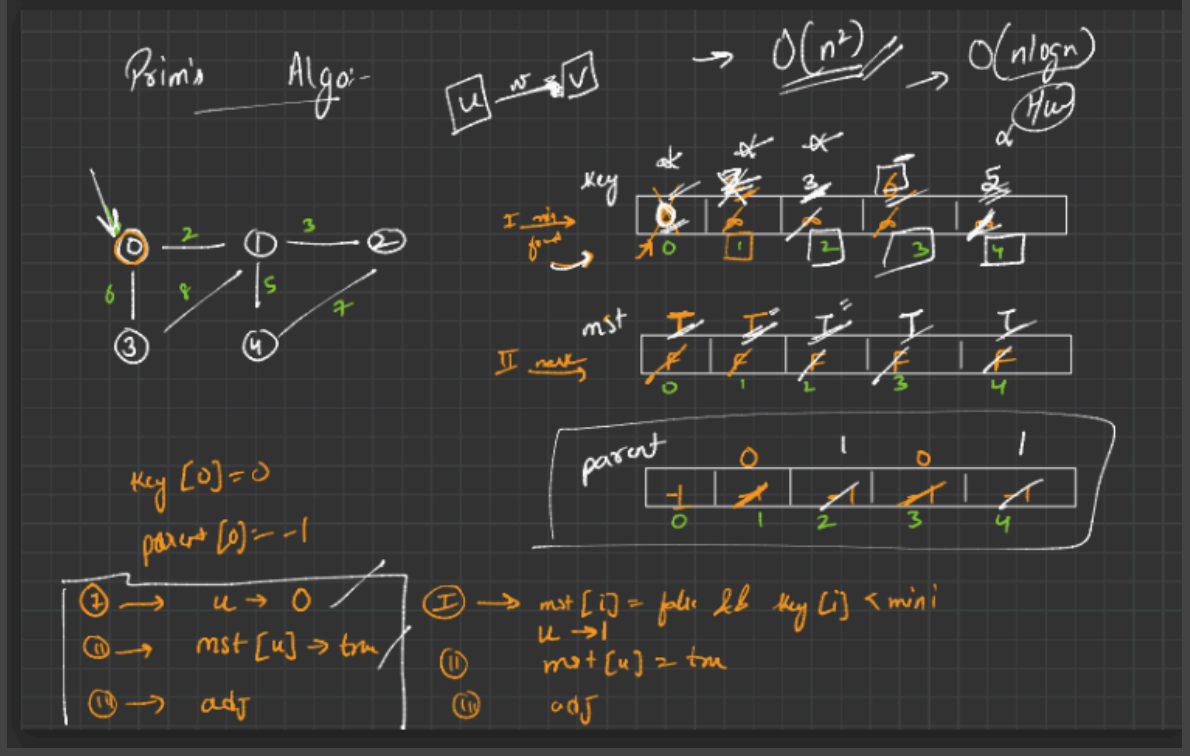
dist

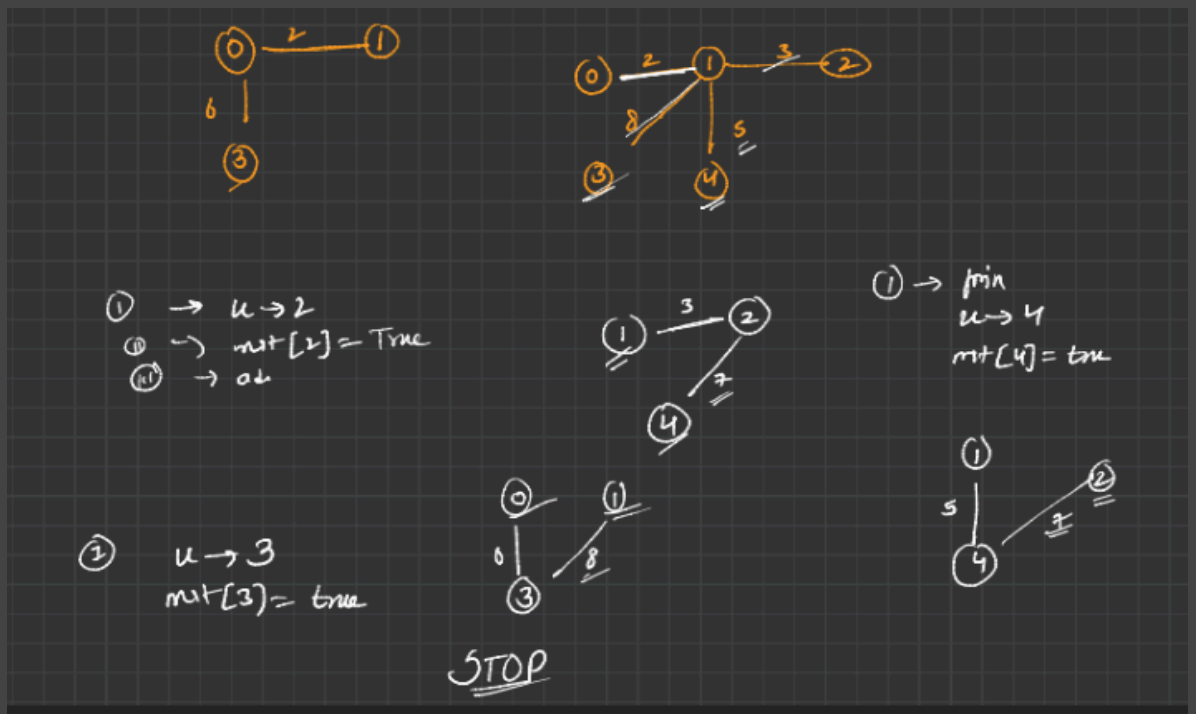
0	1	2	3	4
0	4	1	2	5

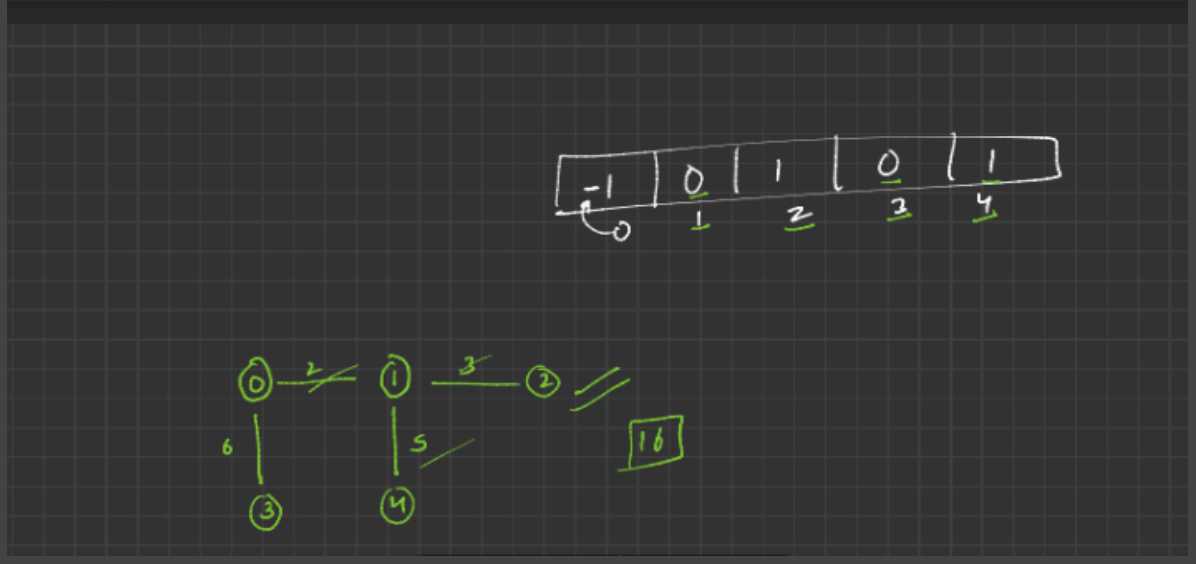


Prim's Algorithm / Minimum Spanning Tree:

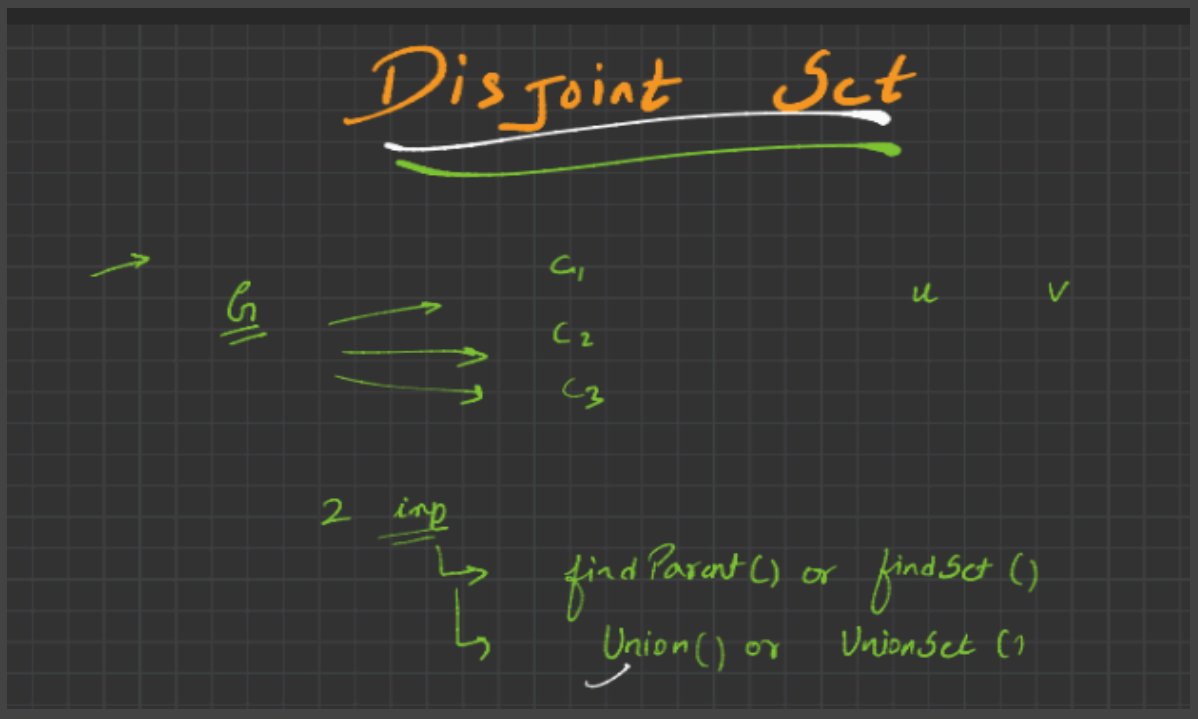




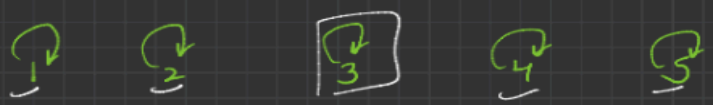




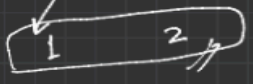
Disjoint Set & Kruskal's Algorithm:



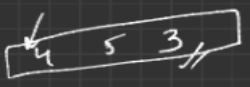
5 disconnected comp



Union (1,2)

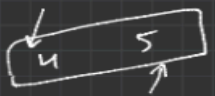


Union (3,5)

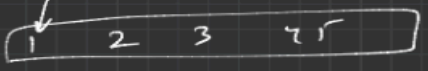


find parent (1) → 1
(2) → 2
(3) → 3
(4) → 4
(5) → 5

Union (4,5)

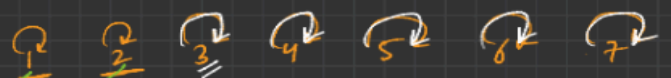


Union (1,3)

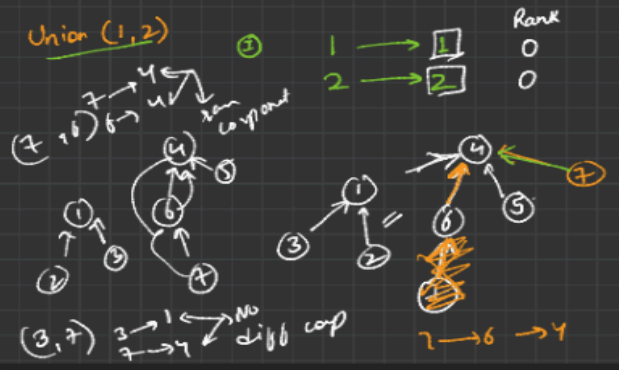


↳ Union by Rank & Path Compression

(7)



Union (1,2)



Rank

1 → 1 — 0
2 → 2 — 0

same comp

(7,6) 7→4, 6→4

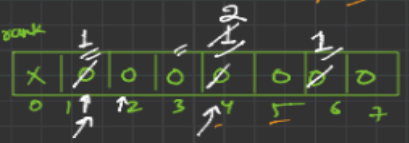
(3,7) 3→1, 7→4 — No diff comp

7→6→4

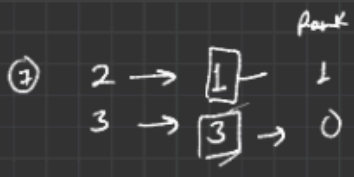
same rank
① parent[2] = 1
② rank[1] ++
③ parent[5] = 4
rank[4] ++

find Parent (1) = 1

2 — 2
3 — 3
4 — 4
5 — 5
6 — 6
7 — 7



Union (2,3)

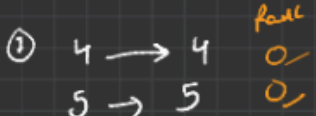


Rank

2 → 1 — 1
3 → 3 → 0

rank[3] < rank[1] !
parent[3] = 1

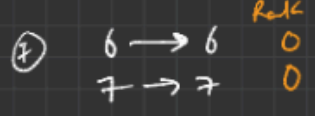
Union (4,5)



Rank

4 → 4 — 0
5 → 5 — 0

Union (6,7)

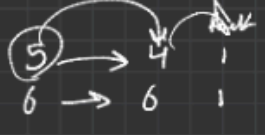


Rank

6 → 6 — 0
7 → 7 — 0

parent[7] = 6
rank[6] ++

Union (5,6)



Rank

5 → 4 — 1
6 → 6 — 1

parent[6] = 4
rank[4] ++

Union (3, 7)

① 3 → 1 — Rank 1

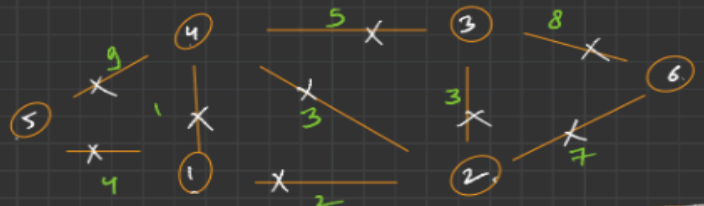
7 → 4 — Rank 2

rank[1] < rank[4]

parent[1] = 4

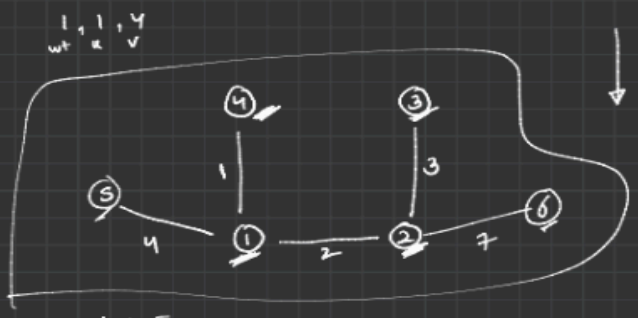
Kruskal's algo:-

Prim's



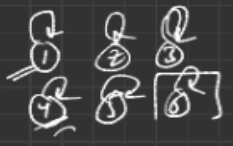
① adj list ✗

② linear D.S



sorted

wt	u	v
1	1	4
2	1	2
3	2	3
3	2	4
4	1	5
5	3	4
7	2	6
8	3	6
9	4	5



1, 1

2, 6

M.S.T

36 α 45 α

3, 4 α

1 2 3

2, 4 α

u v

1 4

① 1 → 1, 4 → 4 — diff → merge / Union

same → α ignore

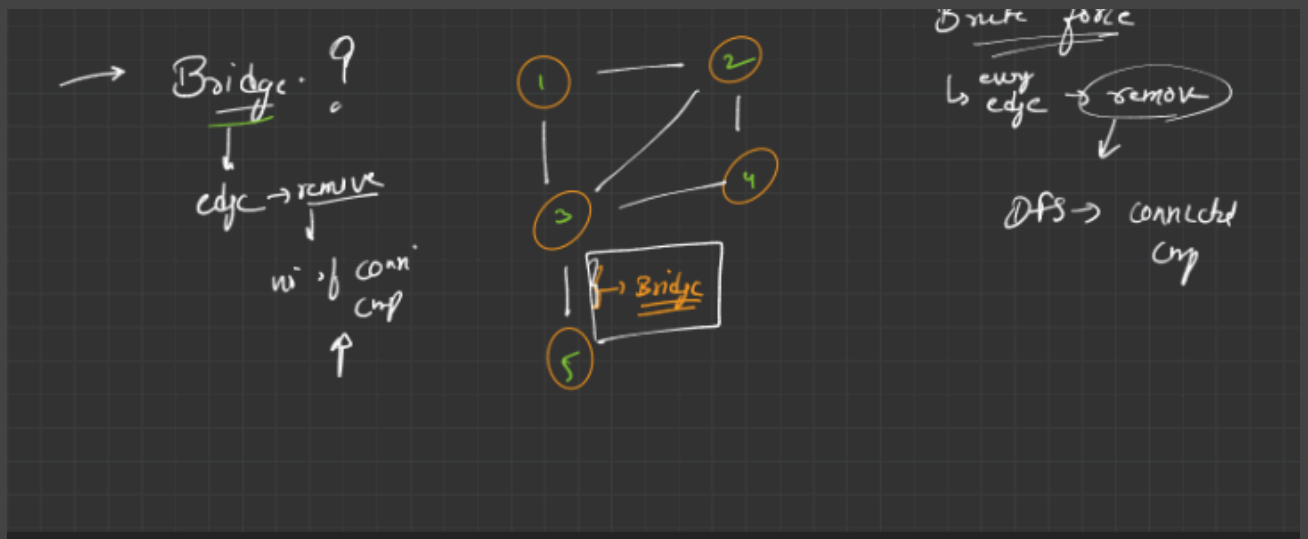
O(4α) = O(4) → O(1)

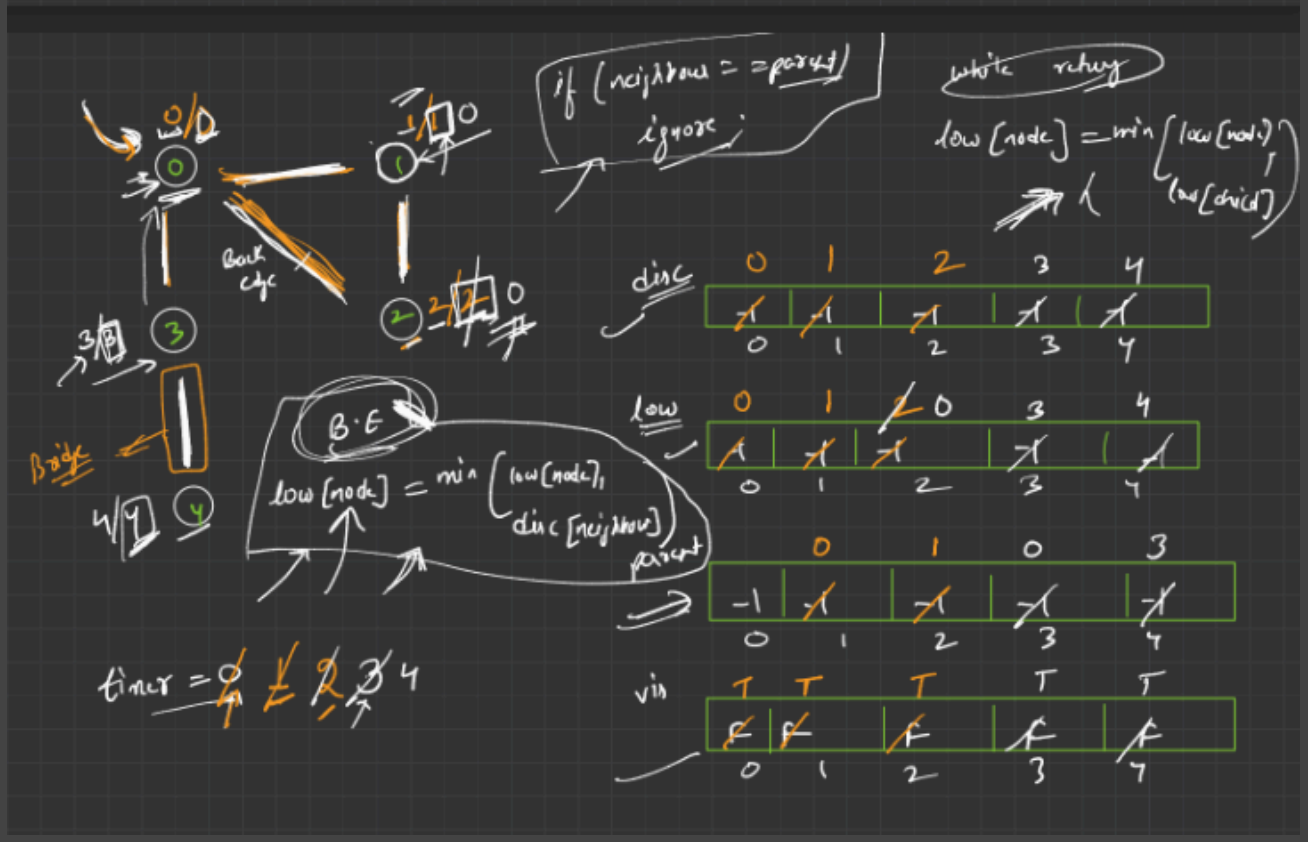
find par, Union set

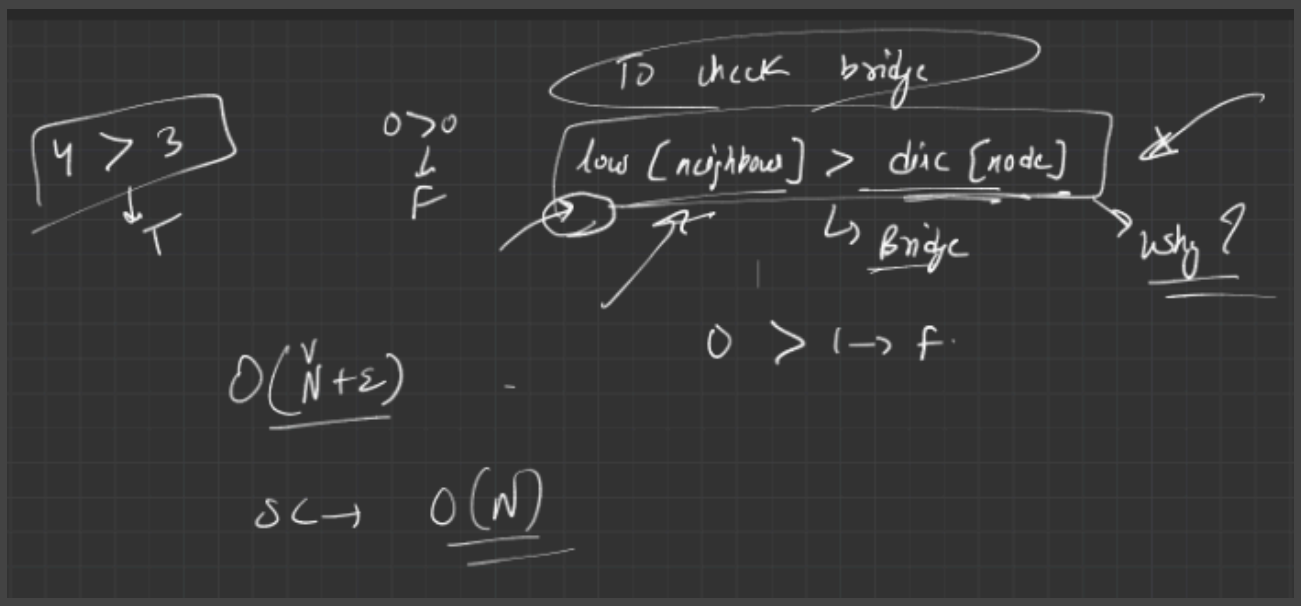
T.C → O(m log m)

S.C → O(n)

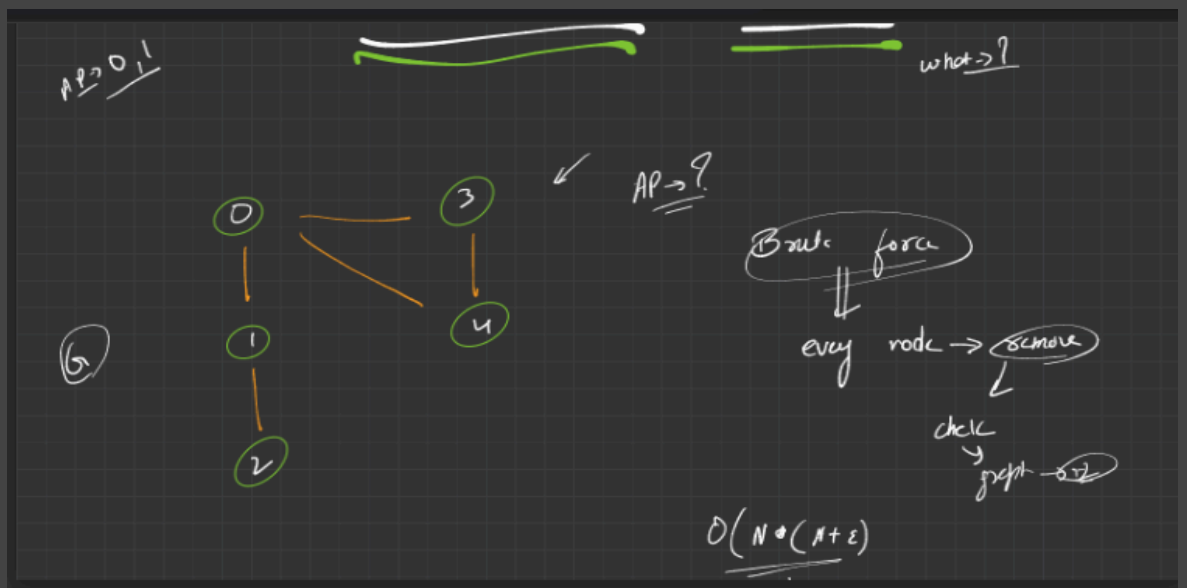
Find Bridges In A Graph:

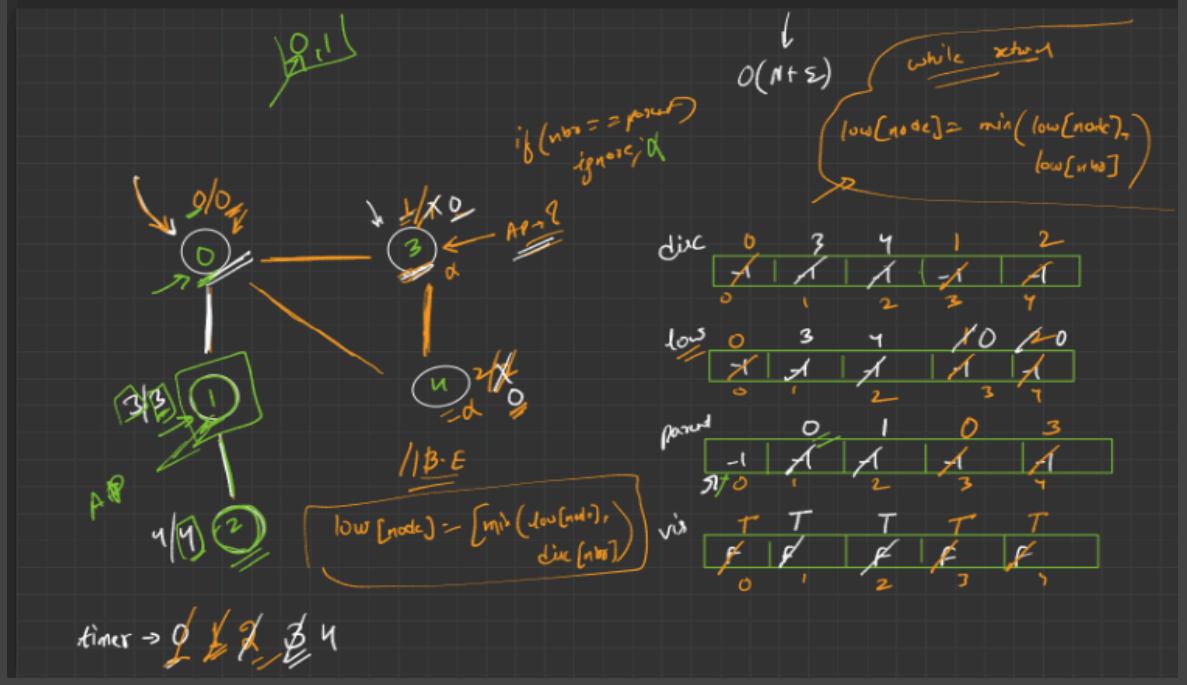


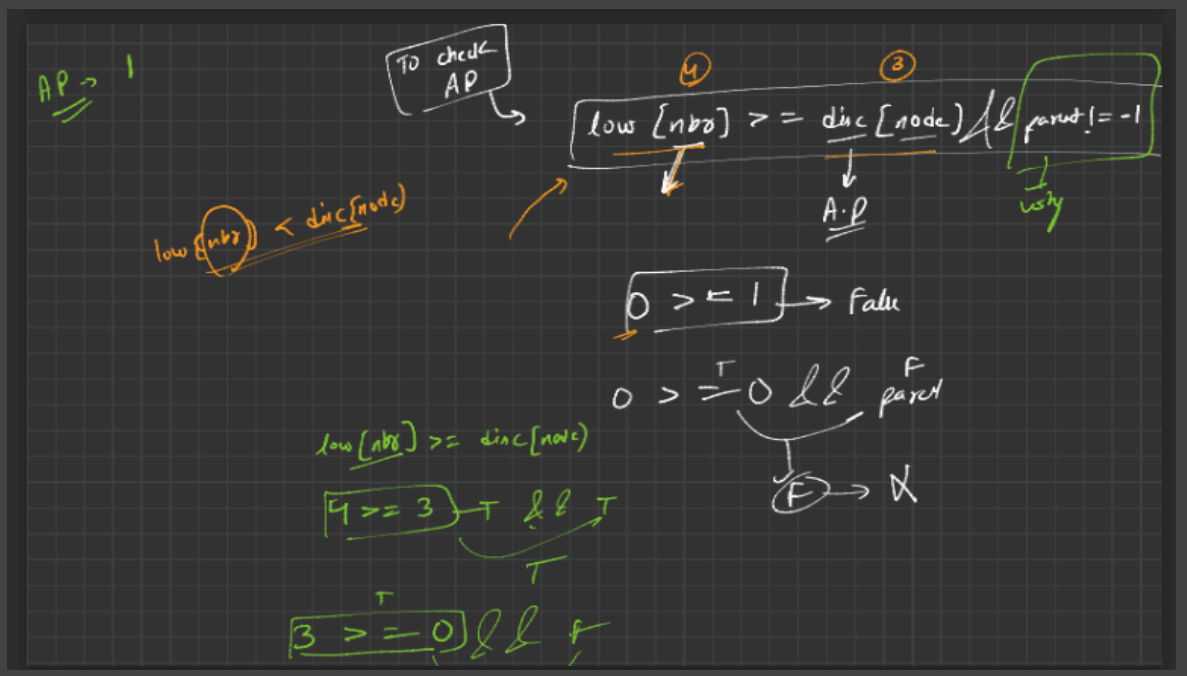


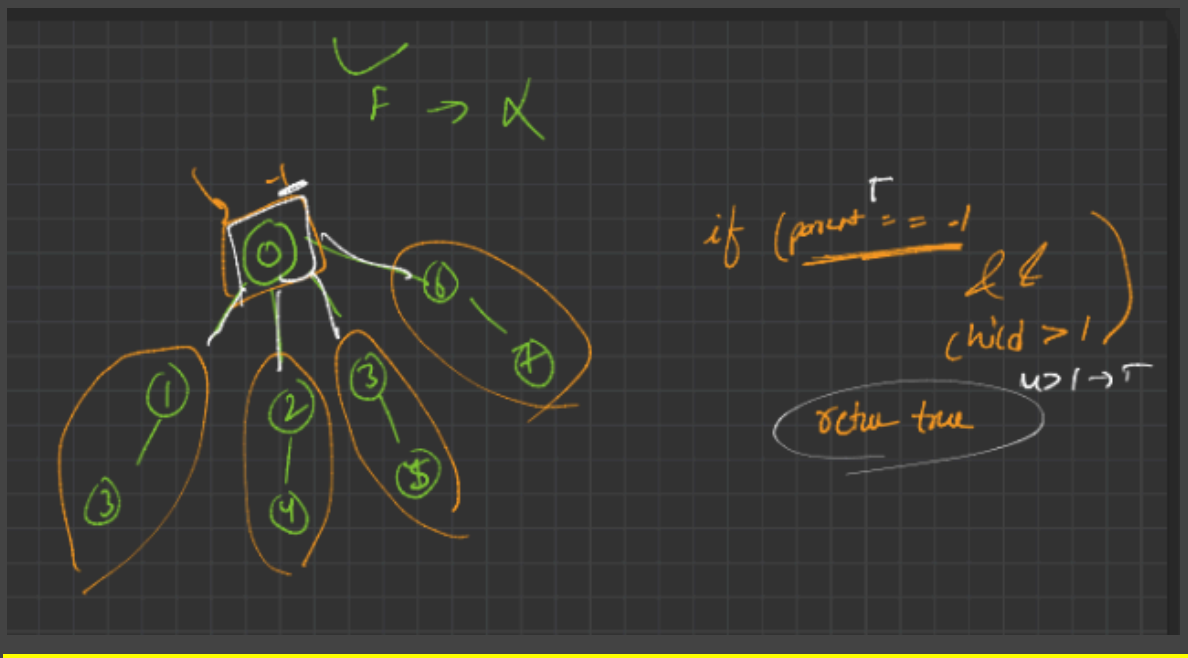


Find Articulation Points In A Graph:

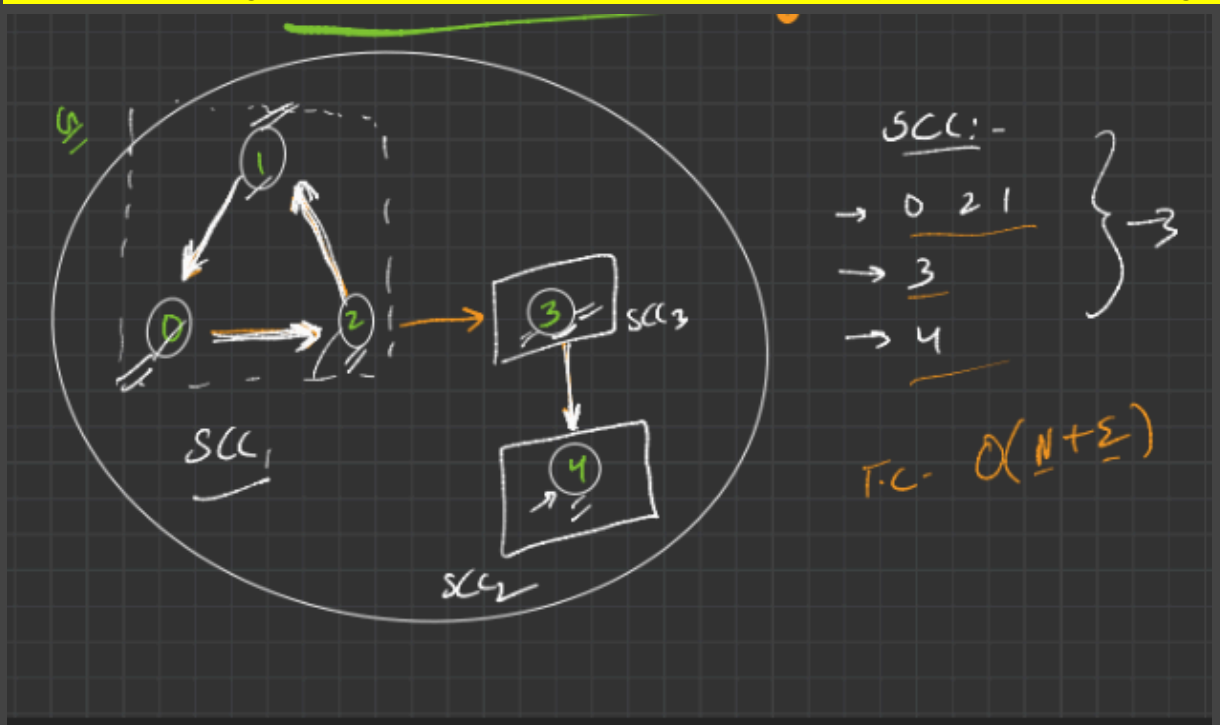


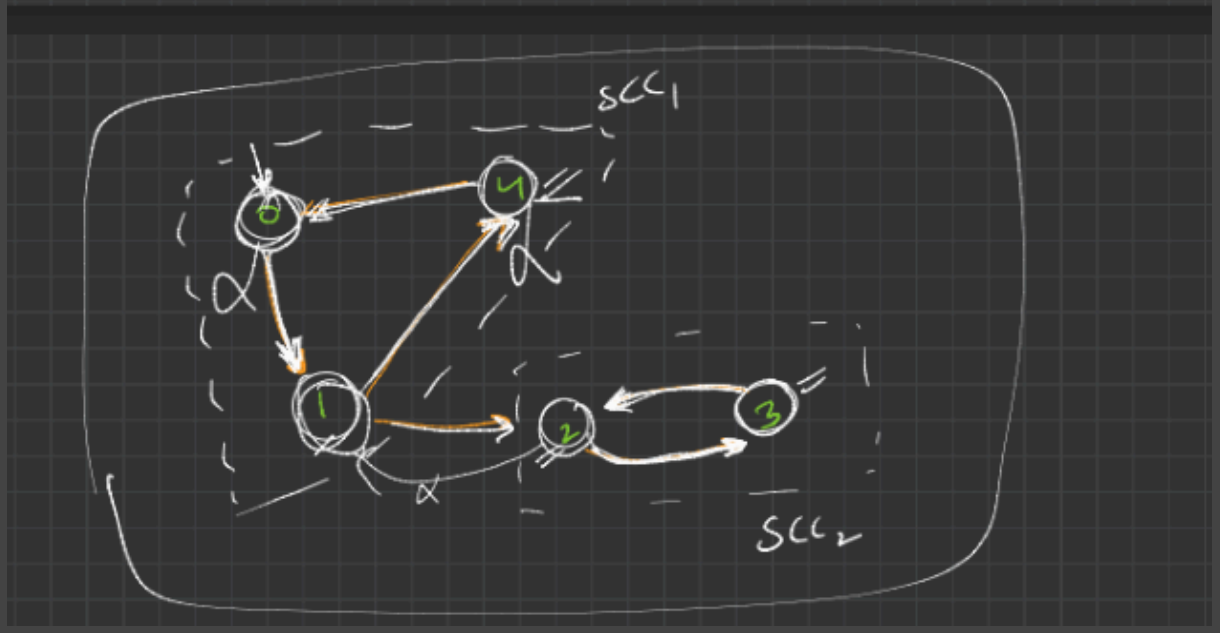


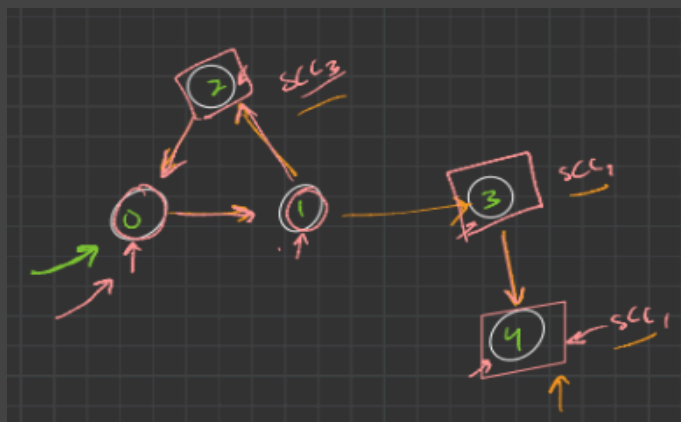


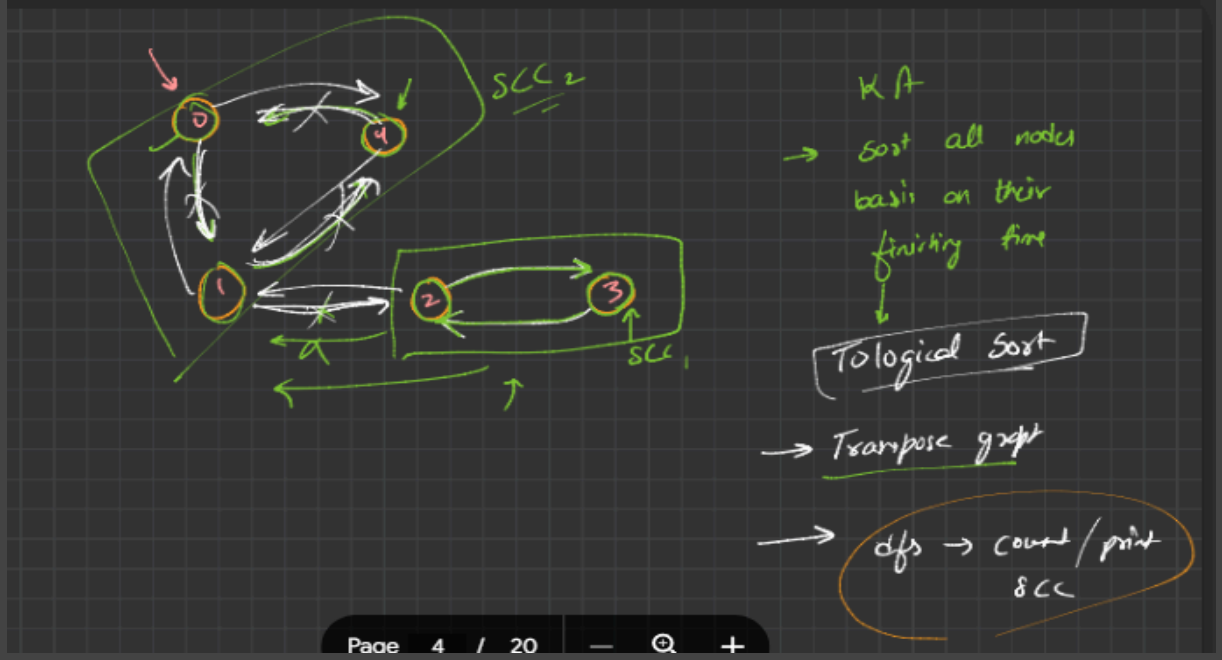


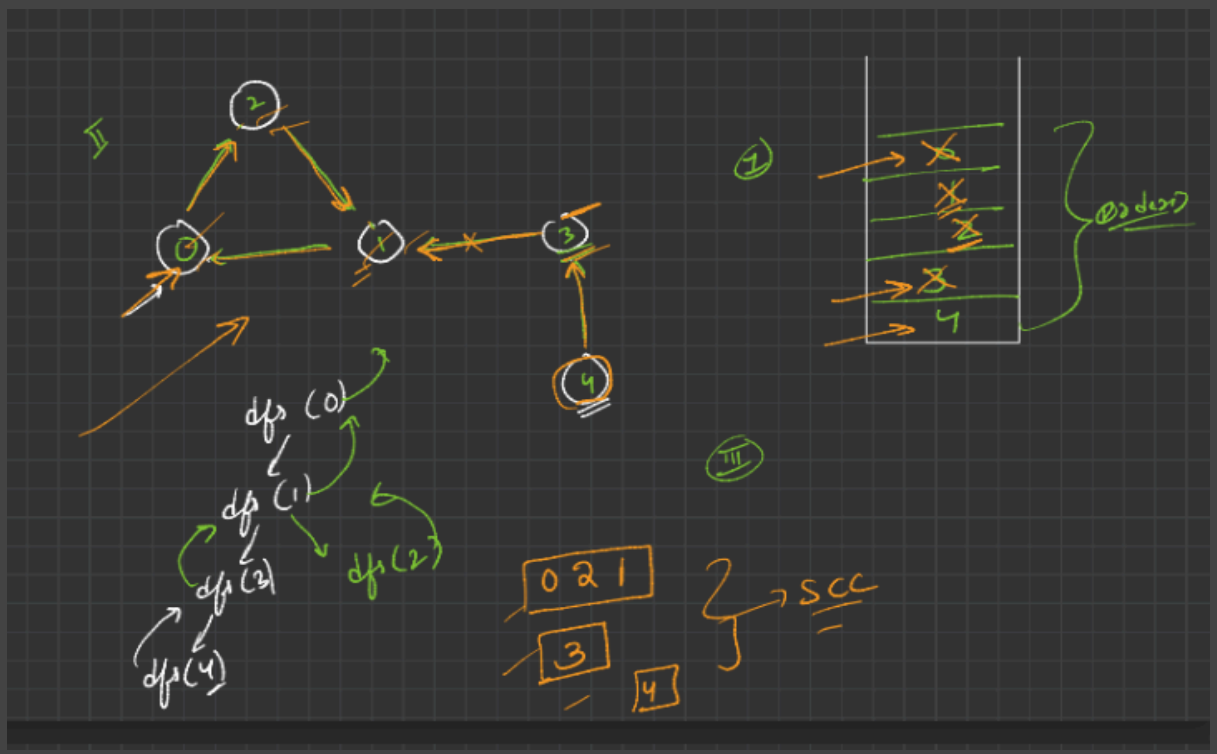
Find the Strongly Connected Components In A Graph / Kosaraju's Algorithm:



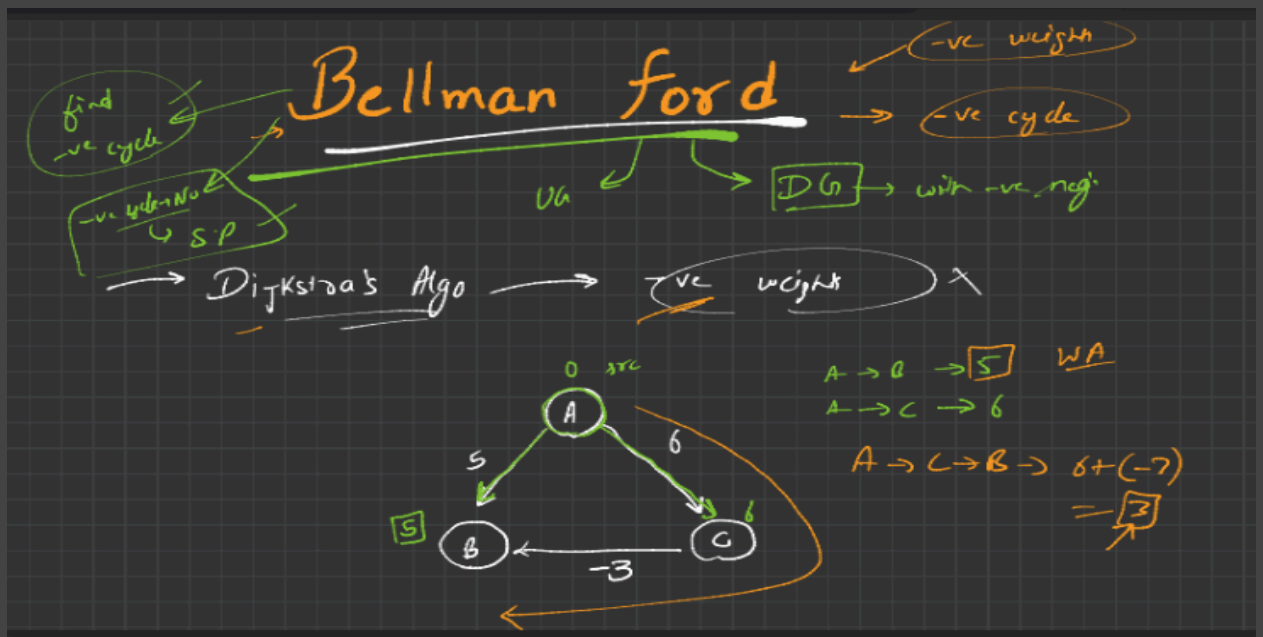


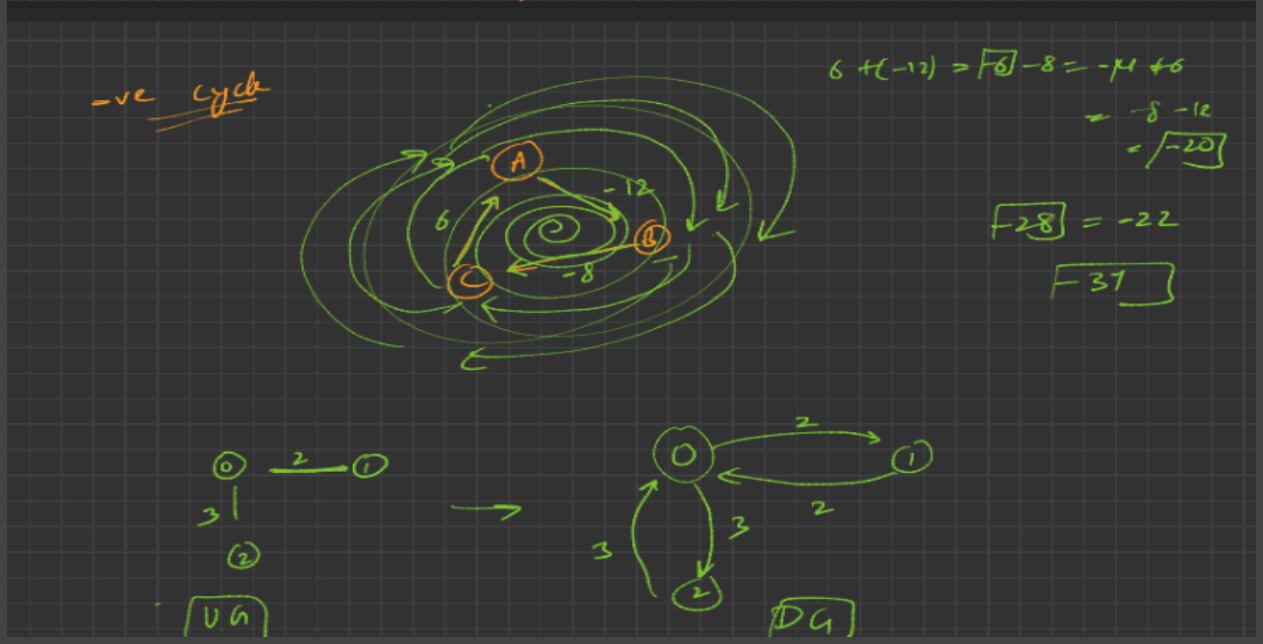

SCC3
SCC2
SCC1

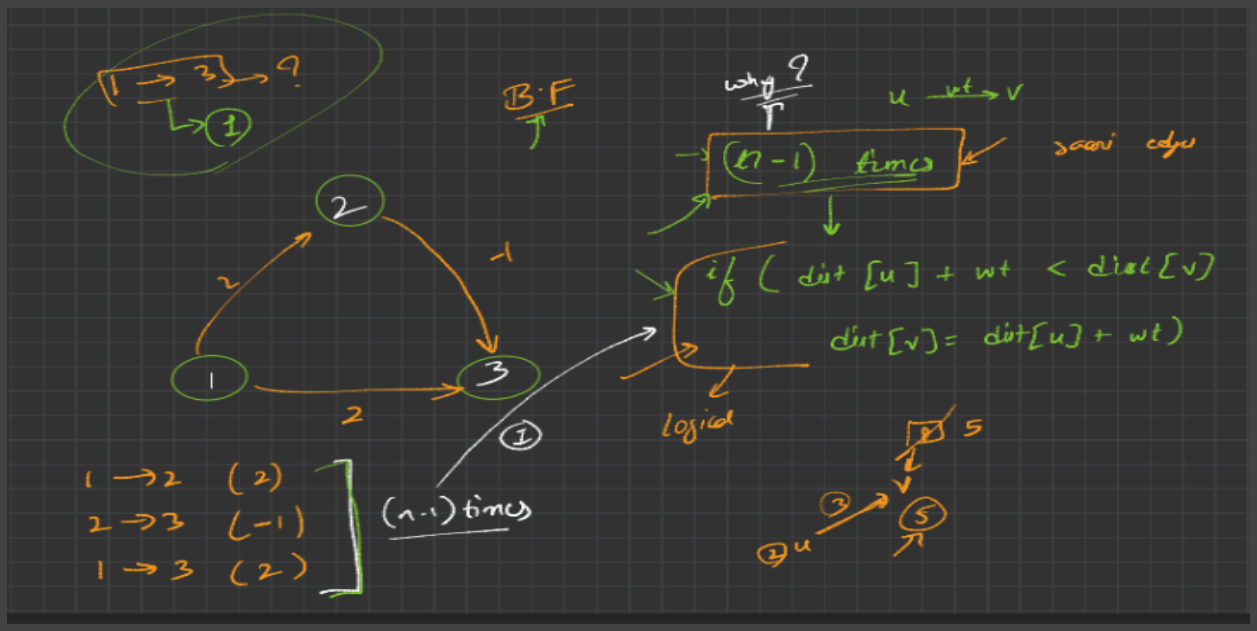

SCC2
SCC1
KA
→ Sort all nodes basis on their finishing time
Tological Sort
→ Transpose graph
→ dfs → count / print SCC
Page 4 / 20


dfs (0)
dfs (1)
dfs (3)
dfs (4)
dfs (2)
0 2 1
3
4
SCC

Shortest Path In Undirected/Directed Weighted(including negative weights) Graph: / Bellman Ford's Algorithm:







(11) 1 more time → same formula

dist → update → -ve cycle present

(yes) (No)

return dist[dest];

SP ✗

n = 3

1st 2nd

edges)

u → v (wt)

2 → 3 (-1)

1 → 3 (2)

src

T.C

$(n-1) \times E_m$

$O(n \times m)$

check cycle -

-ve cycle → not present

dist[2] + (-1) < dist[3]

2 + (-1) < 1

1 < 1 → F

1 → 2 (2)

2 → 3 (-1)

1 → 3 (2)

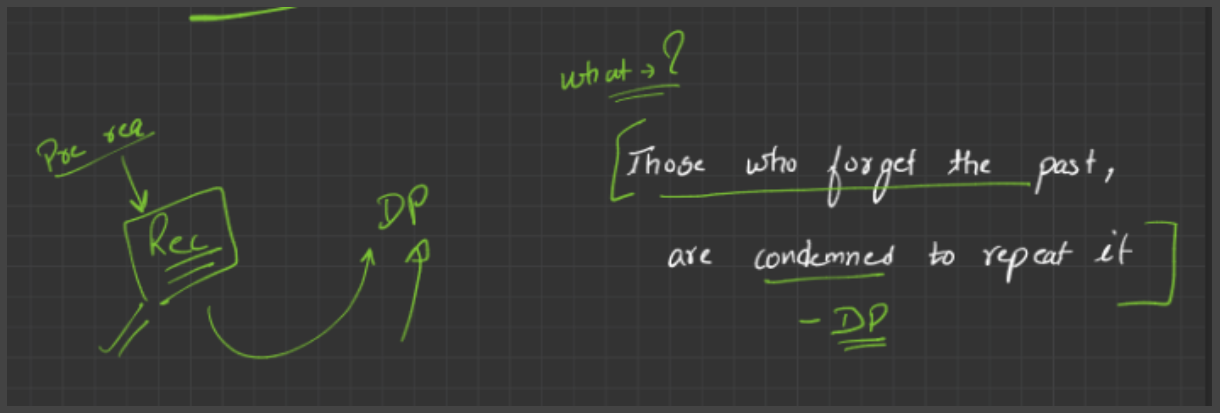
dist[1] + 2 < dist[3]

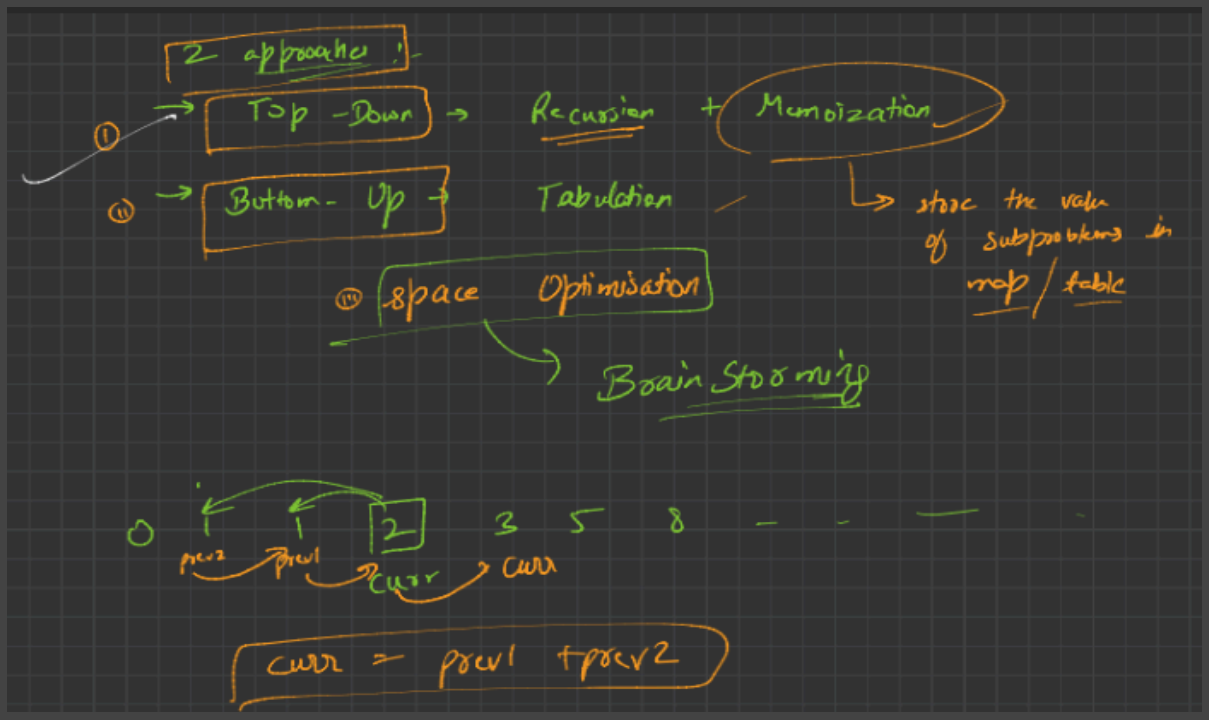
0 + 2 < 1

2 < 1

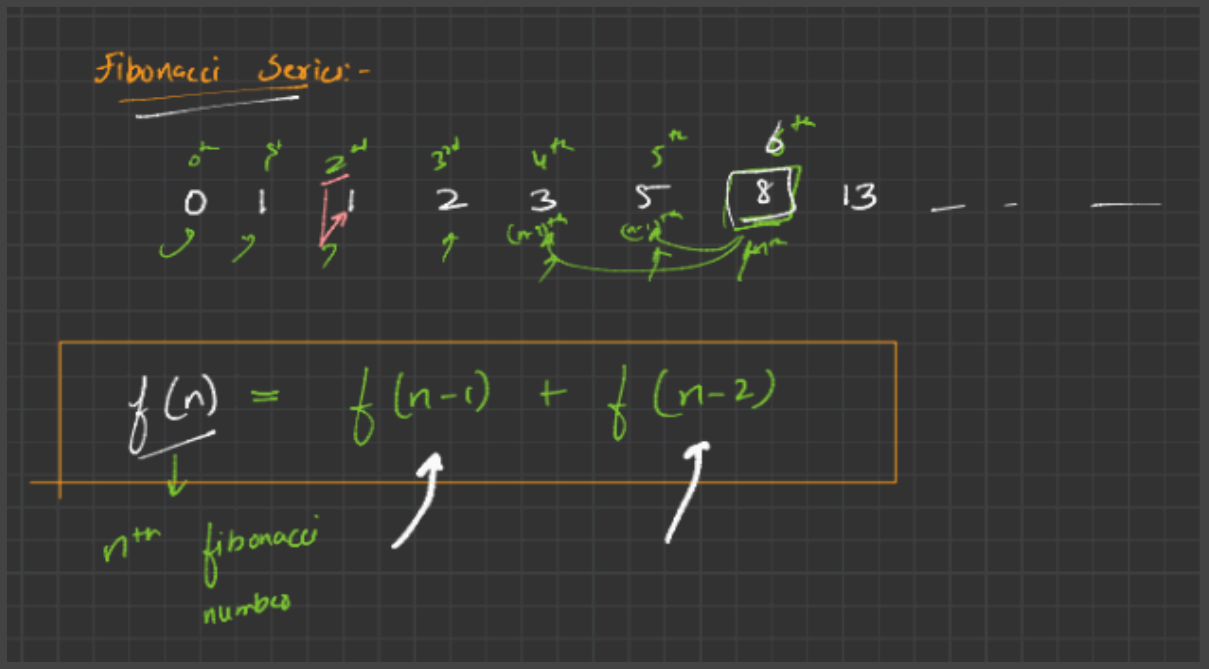
F

Dynamic Programming





Fibonacci By Top-Down Approach:



code

Recursive Implementation

```
func ( int n)
{
    // B.C
    if (n==1 || n==0)
        return n;
    return dp[n] = f(n-1) + f(n-2);
}
```

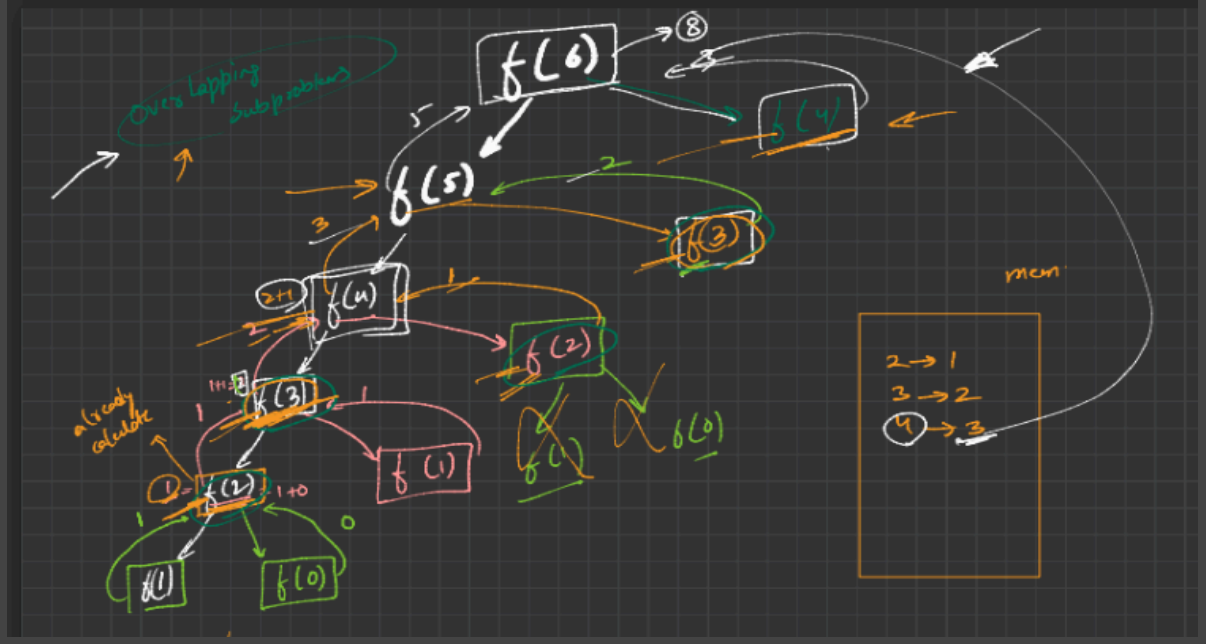
Recursive soln → DP sol → Top Down → Rec + Memoization

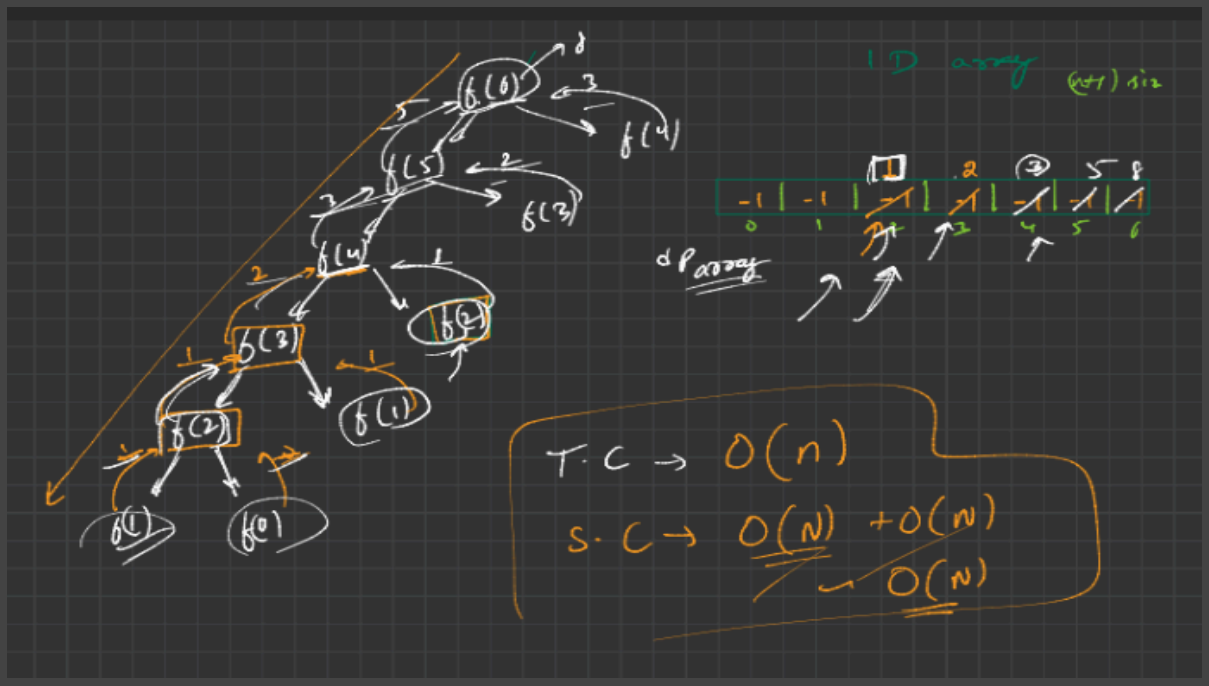
① dp[n+1];

② if (dp[n] != -1) return dp[n]; (III)

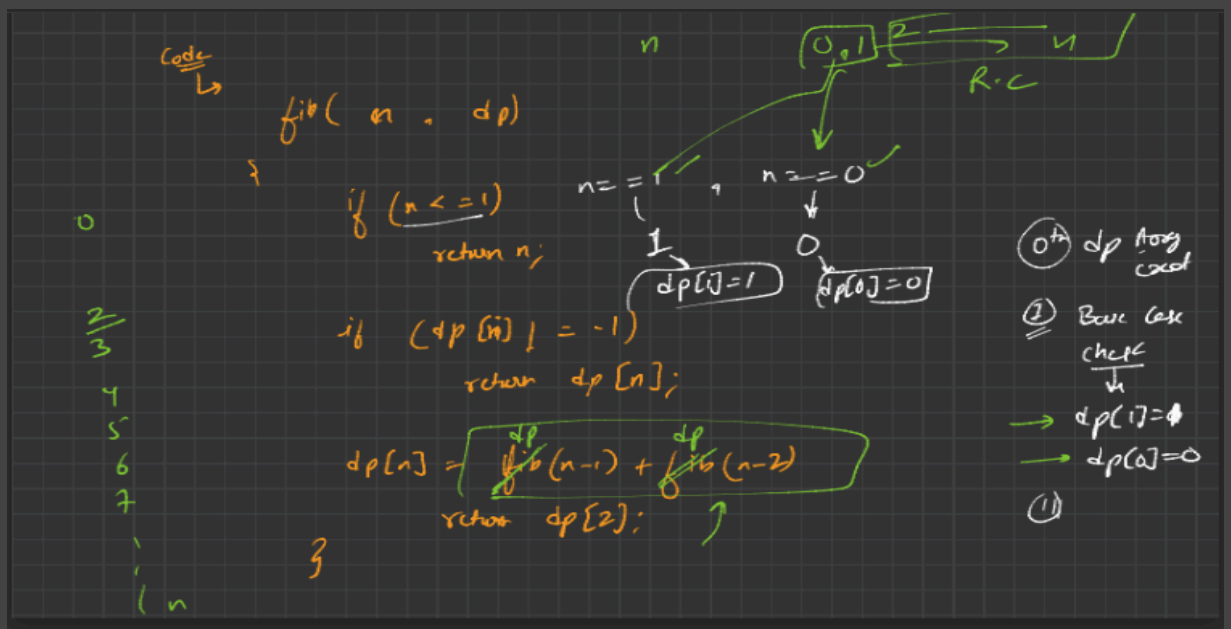
(II)

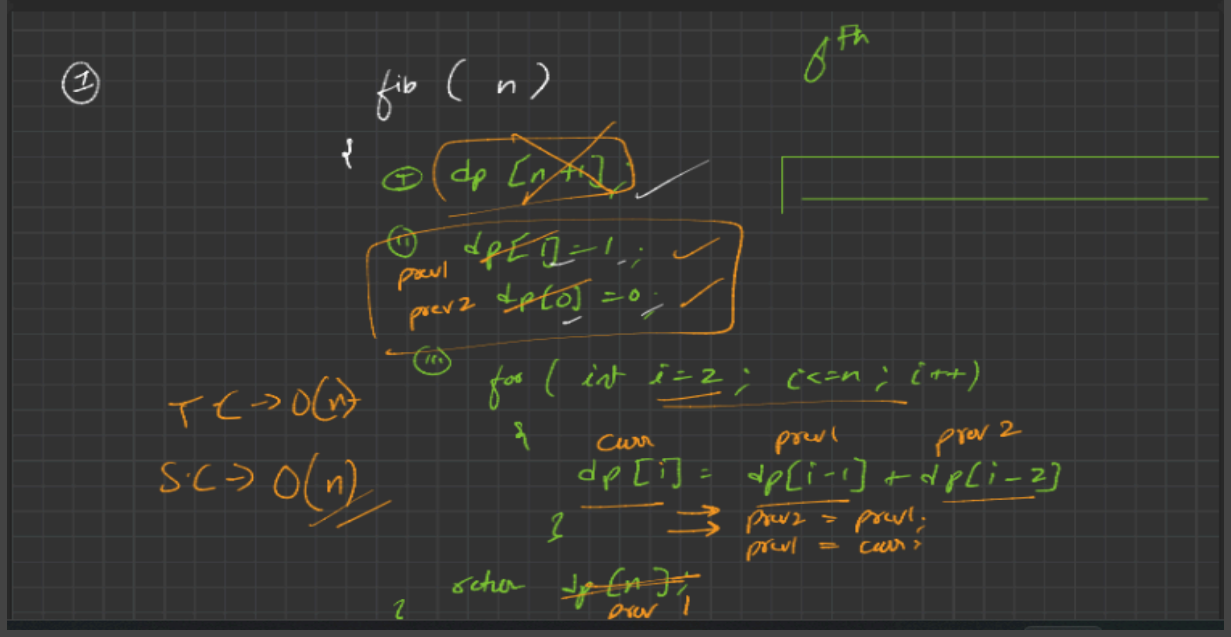
DP
- I
- II



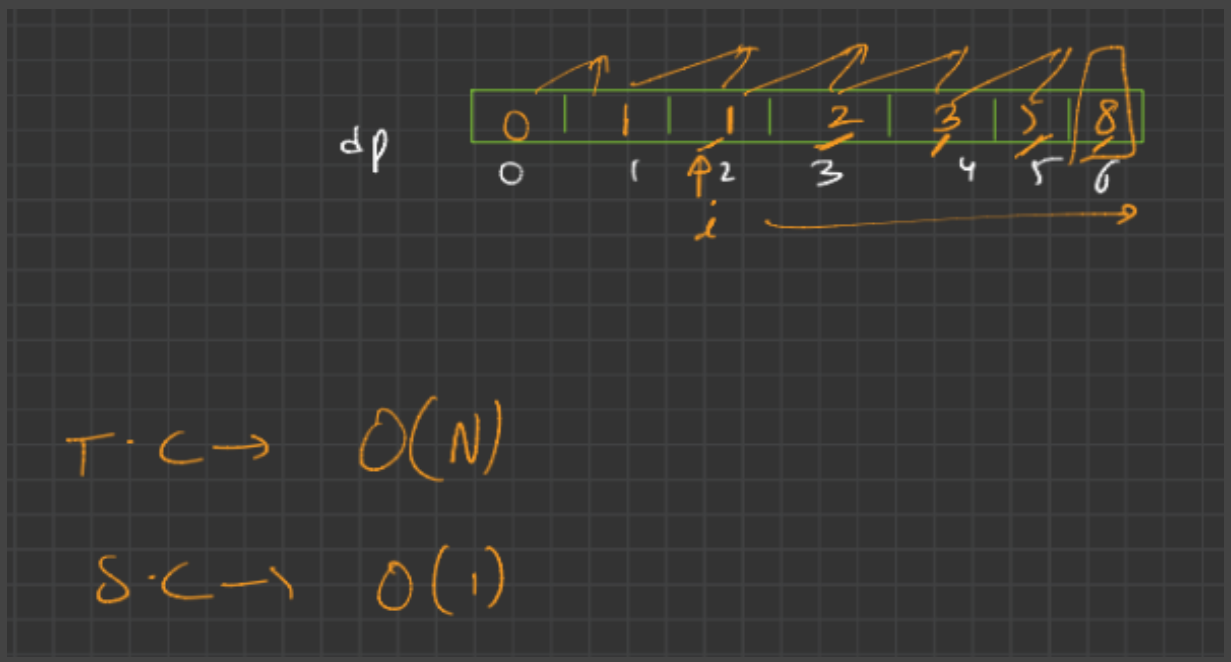


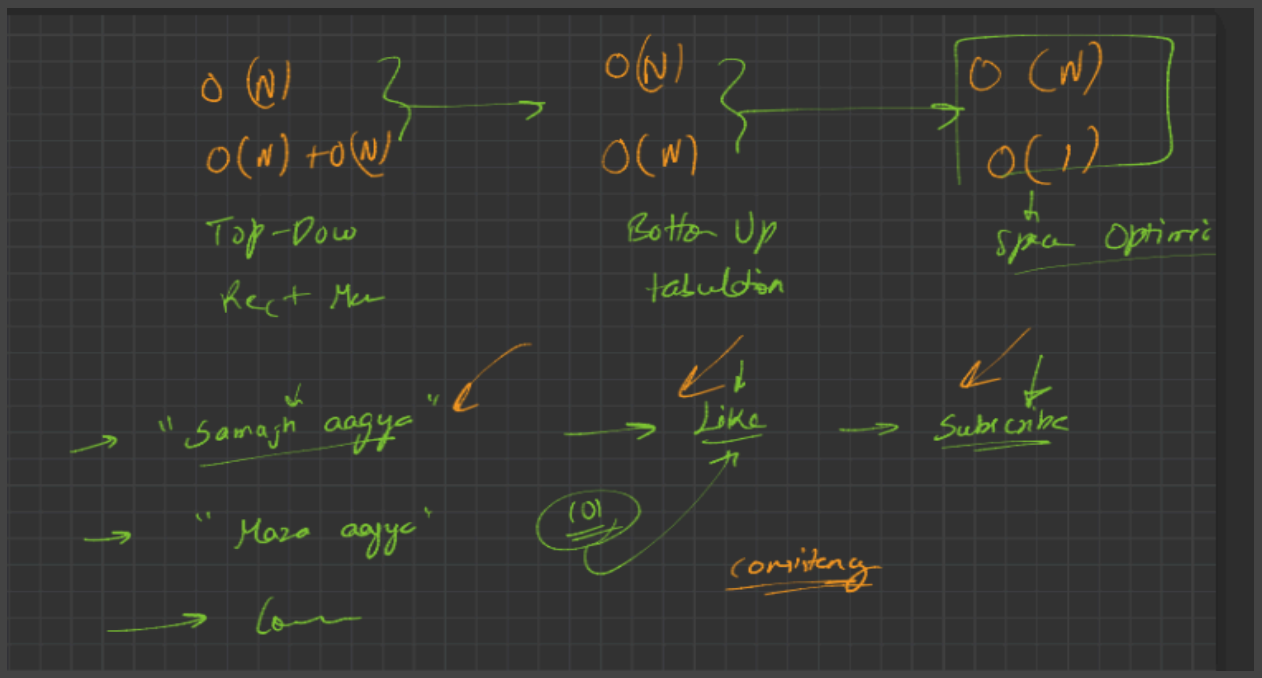
Fibonacci By Bottom-Up Approach:



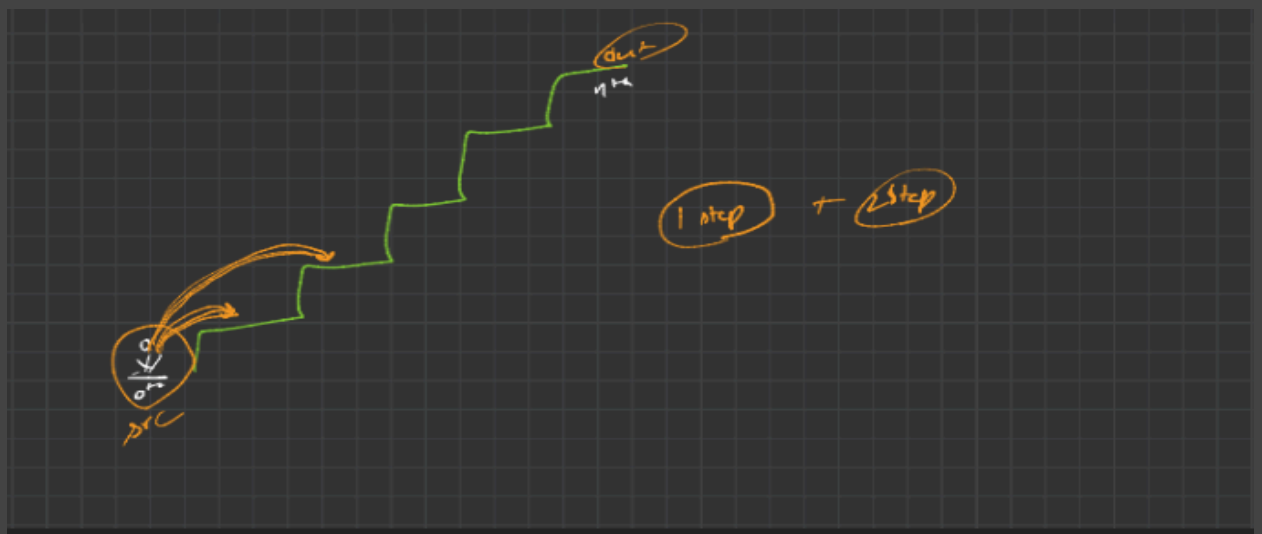


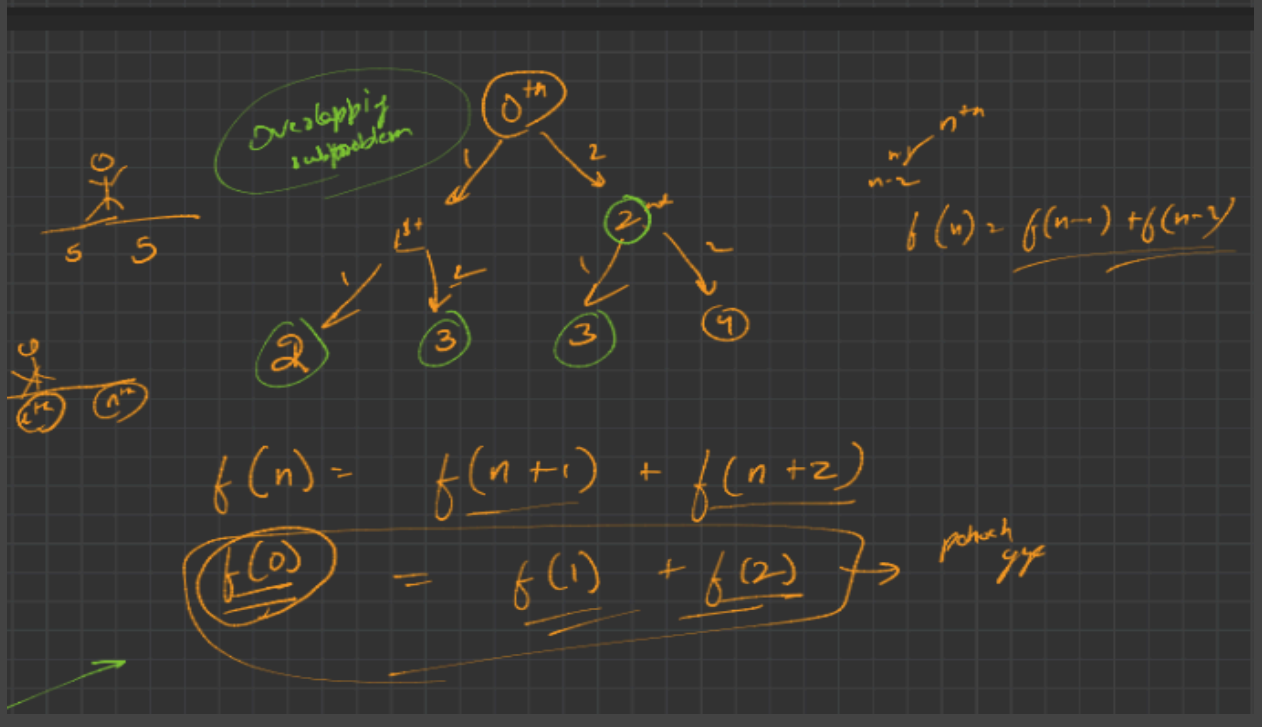
Fibonacci By Space Optimization Approach:

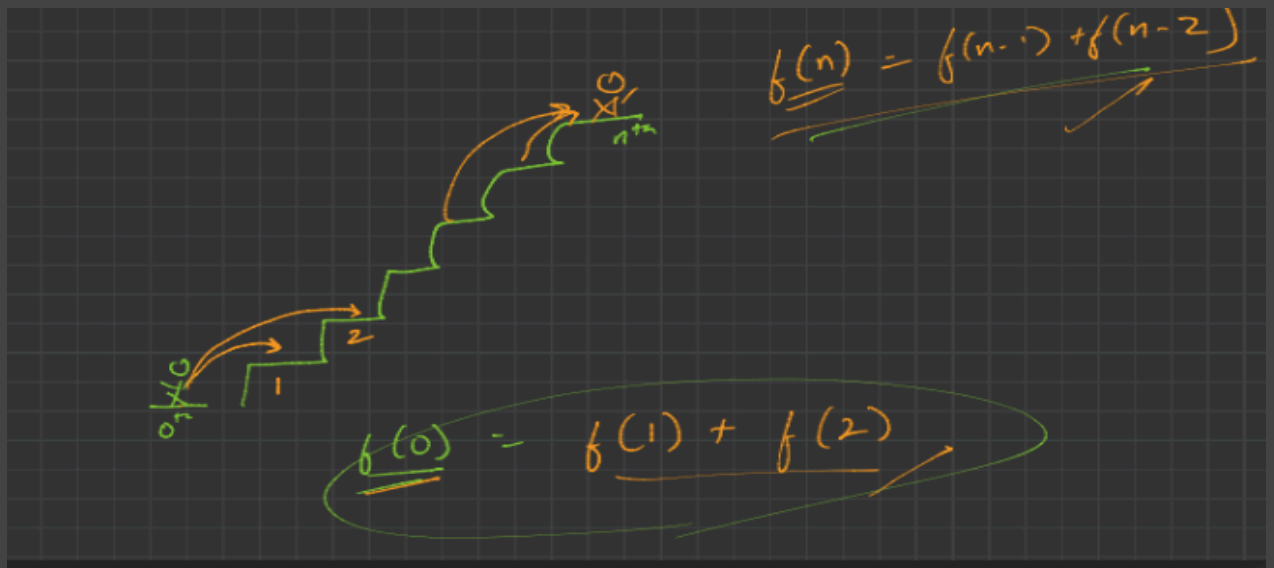




Minimum Cost Of Climbing Stairs By Recursion / Top-Down Approach / Bottom-Up Approach / Space Optimization Approach:





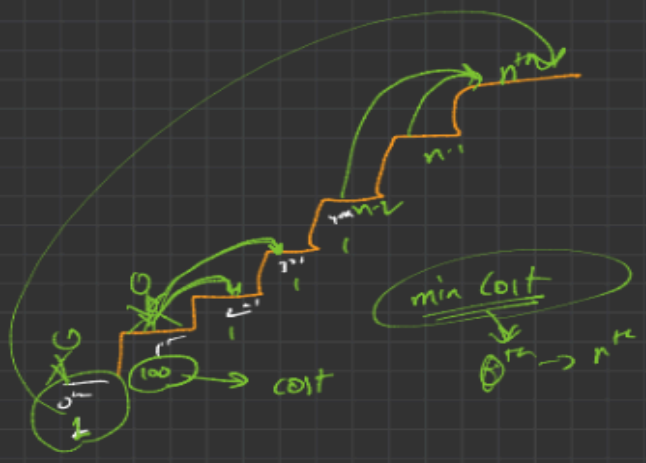


Min Cost Climbing Stair

① we can start from 0th or 1st stair

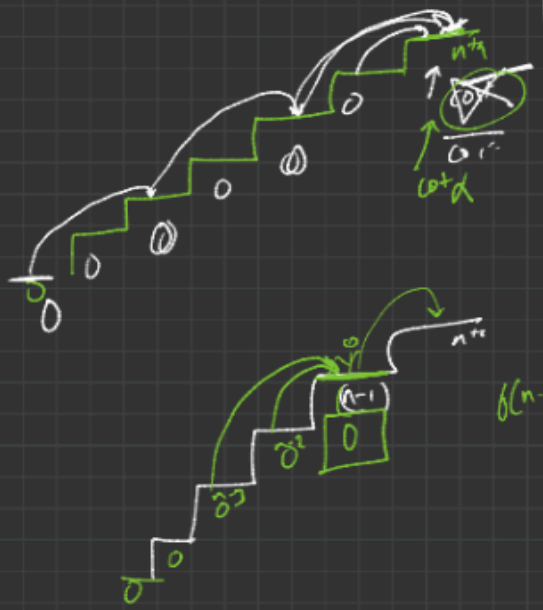
② we have to pay cost, only then we can move 1 or 2 steps ahead

③ Aim: min cost to reach top floor



Recursi

$$f(n) = \min(f(n-1), f(n-2))$$



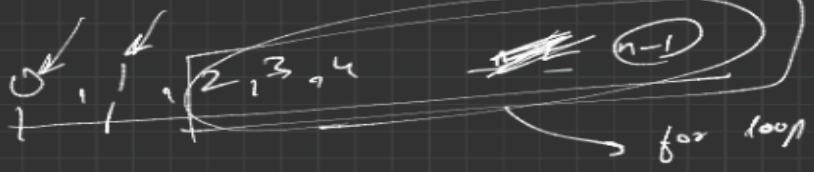
$$f(n-1) = \min(f(n-2), f(n-3)) + cost[i]$$

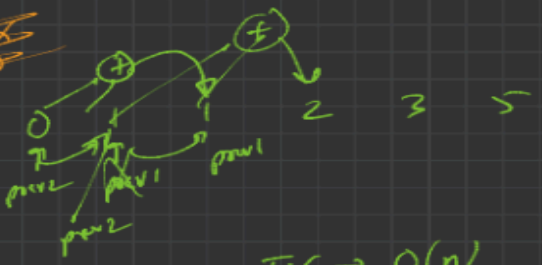
$f(K) = \min(f(K-1), f(K-2)) + cost[K]$

0 — n-1

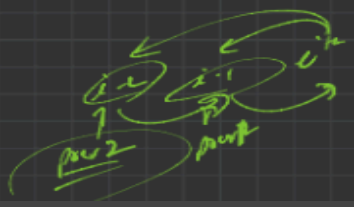
① Recursion + Memoization → O(n)
O(n)+O(n)

③ Tabulation → Bottom - Up T.C → O(n)
S.C → O(n) → O(1)





T.C → O(n)
S.C → O(1)

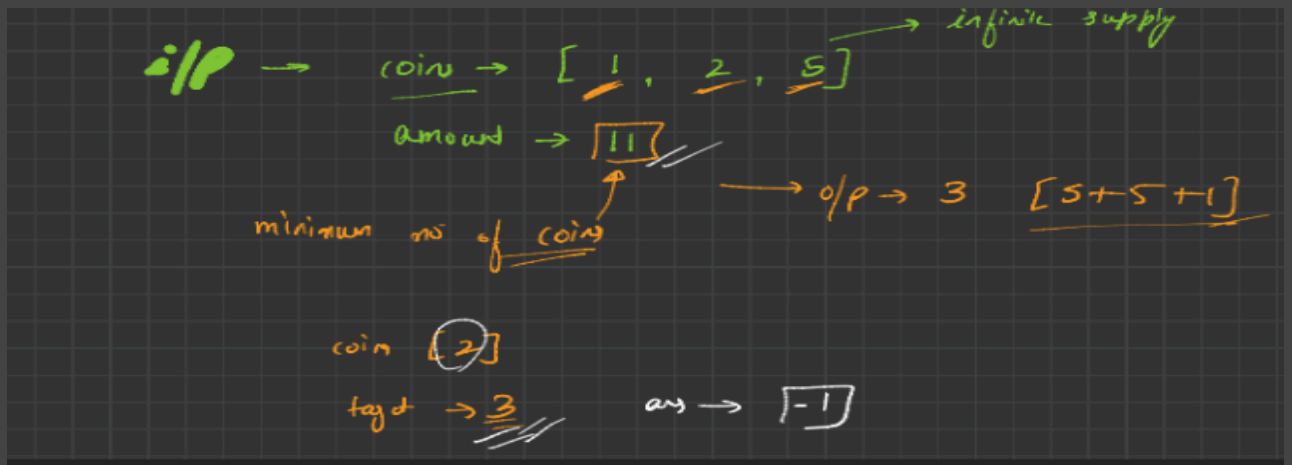


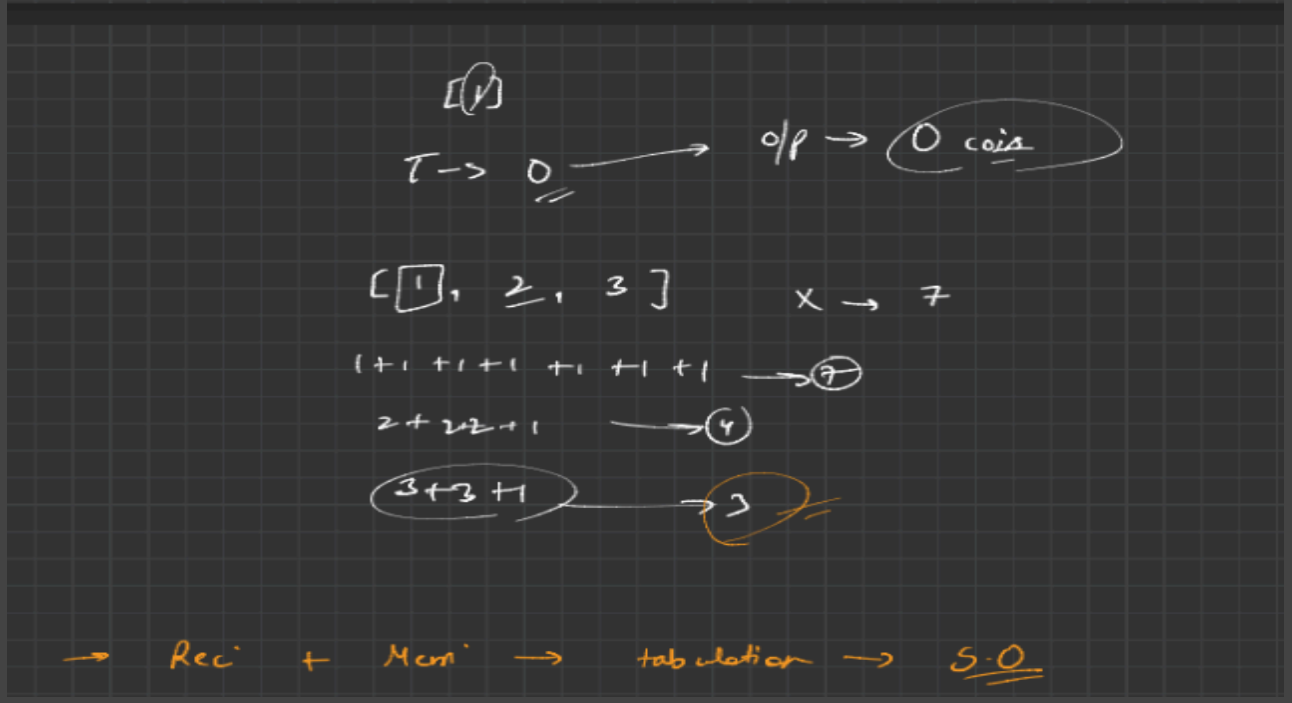
→ Recursion + Memoization → T.C → O(n)
S.C → O(n)+O(n)

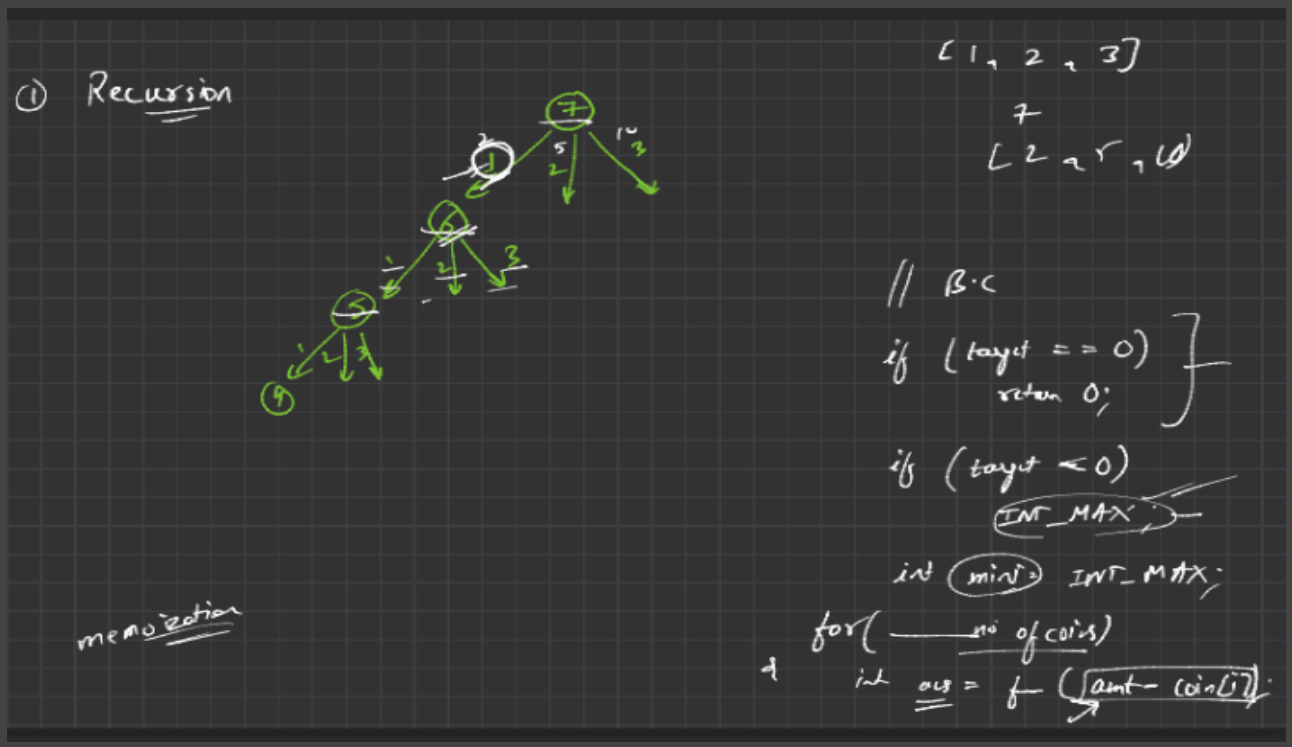
→ Tabulation → T.C → O(n)
S.C → O(n)

→ Space Optimisation → T.C → O(n)
S.C → O(1)

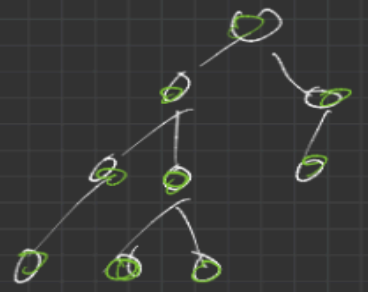
Minimum Number Of Coins By Recursion / Top-Down Approach / Bottom-Up Approach:







if (ans != INT_MAX)
mini = min (mini, 1+ans)
}
return ans;



dp[i]

n

k → min no of coins

dp[] → dp[—]

for (i → <=amt)

dp[1]
dp[2]
dp[3]

dp[amount]

① Recursion: - T.C → exp0
+
Mem

T.C → x * n
amount no of coins

S.C → O(x)

② Tabulation → T.C → x*n

S.C → O(x) → O(1)

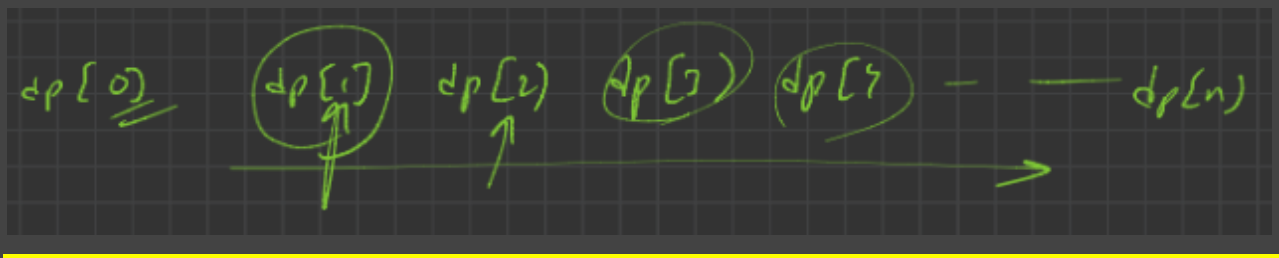
dp[i] → dp[i-1] → prev
→ dp[i-2] → prev2

Space Optimisation
?

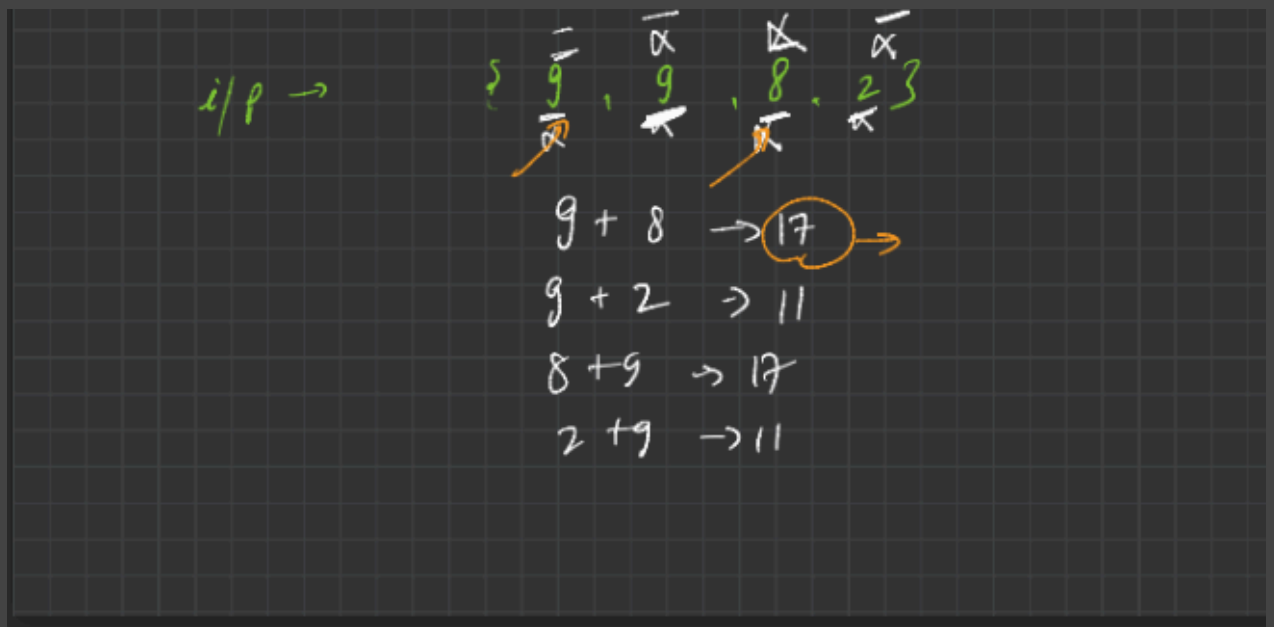
dp[i] → dp[i - num[i]];

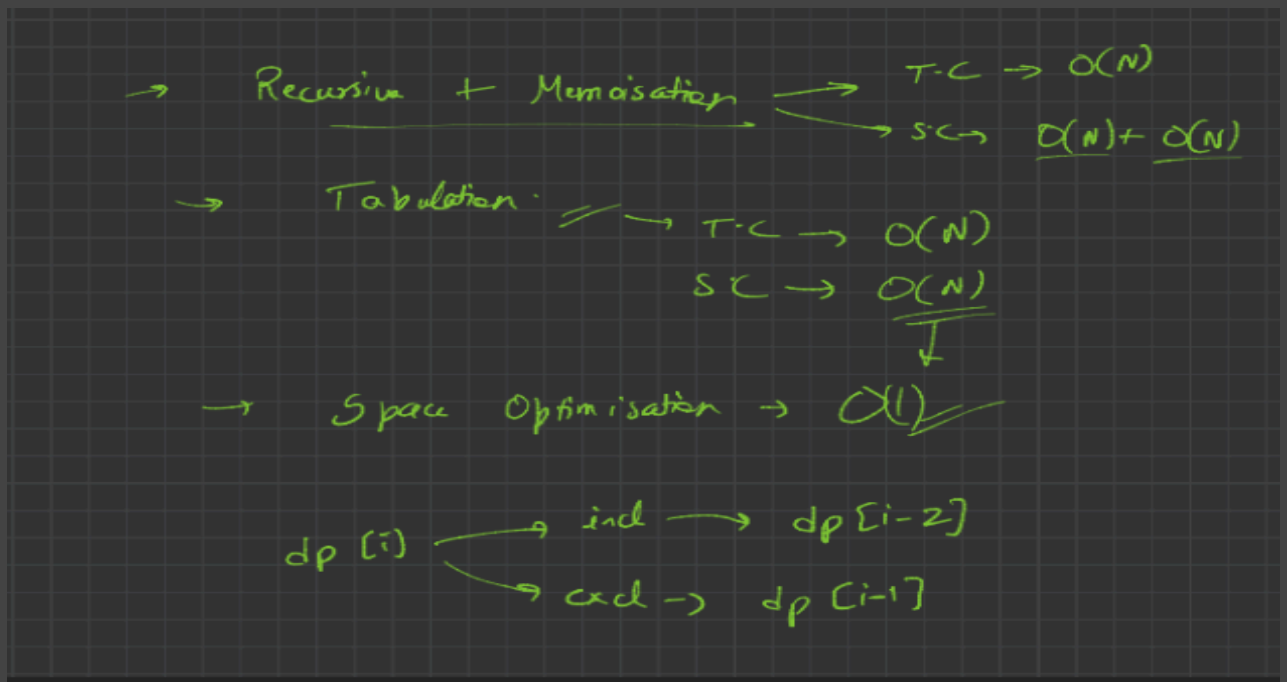
i - 1
i - 2
i - 3
i - 4
i - 5
:
0

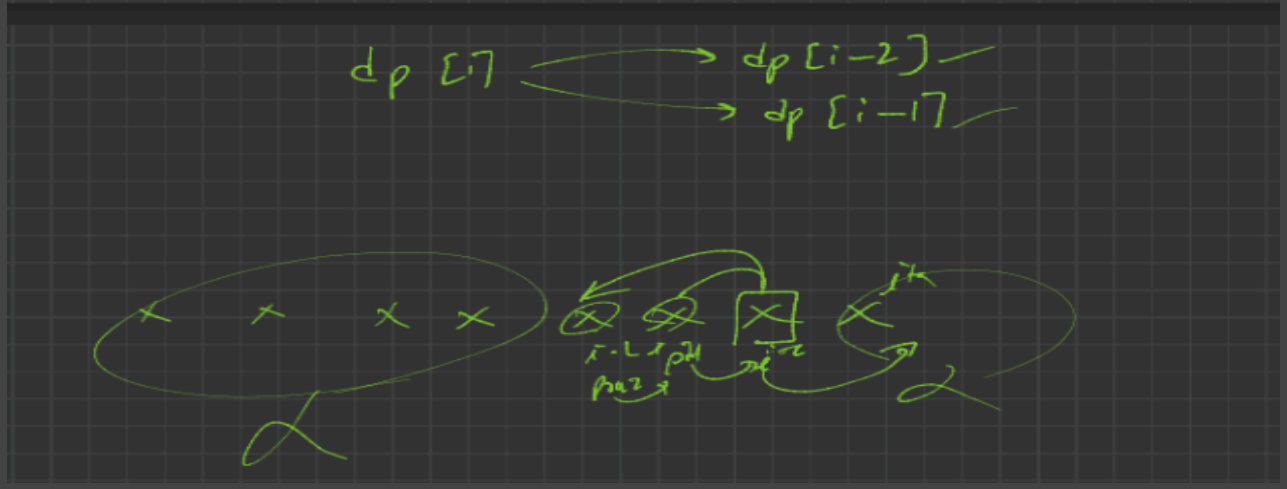
5
1 2 3 4 5
dp x → variable
O(x)



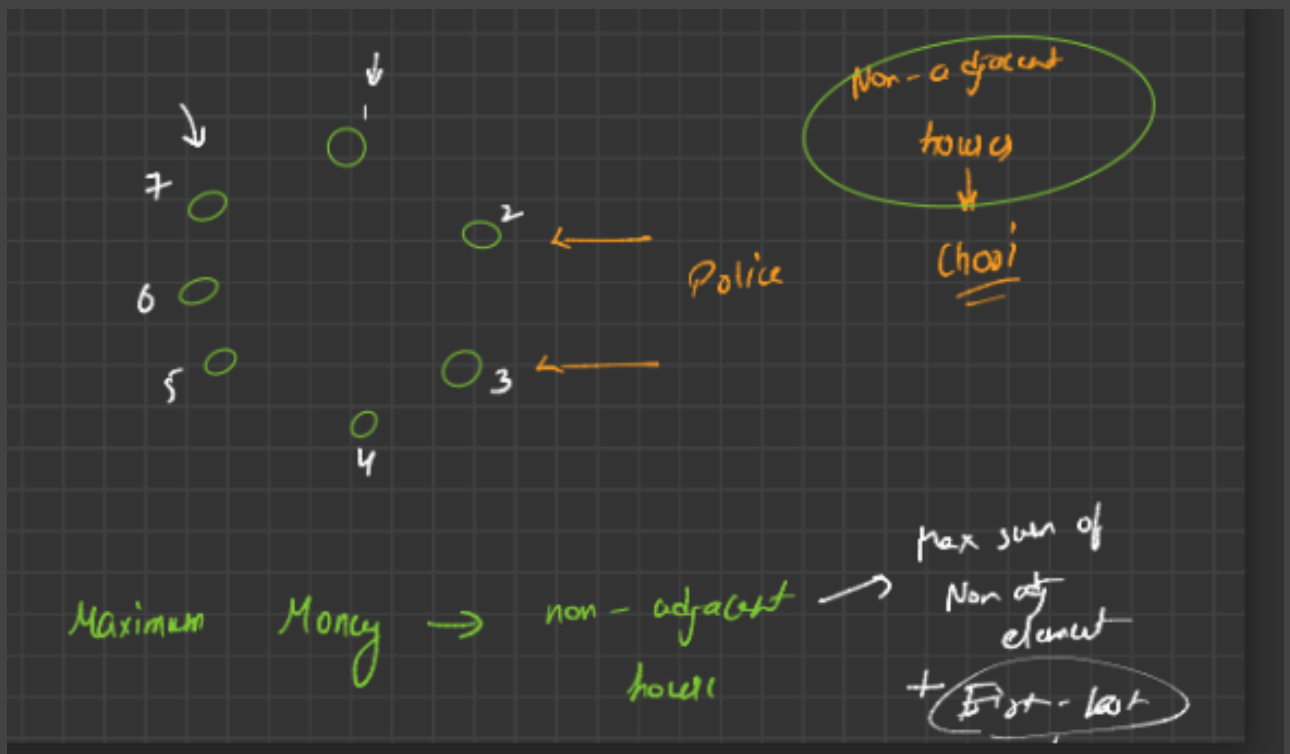
Maximum Sum Of Non-Adjacent Elements By Recursion / Top-Down Approach / Bottom-Up Approach / Space Optimization Approach:

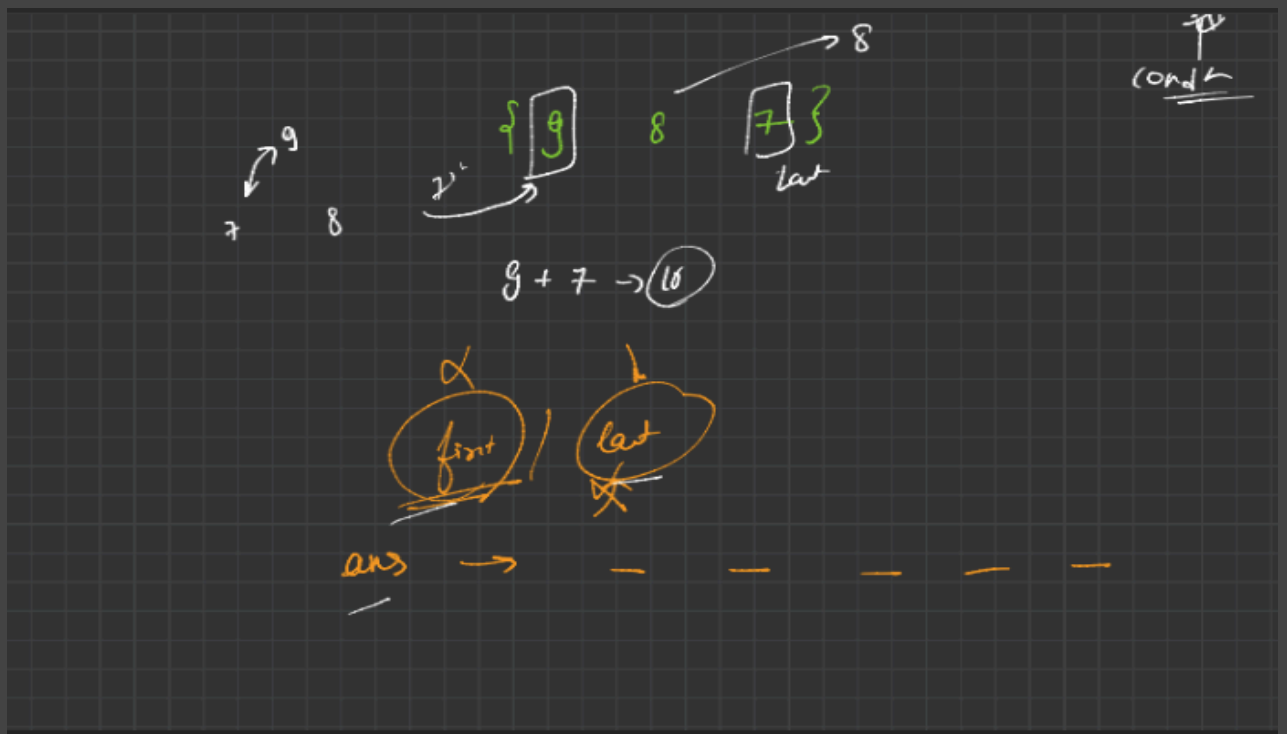


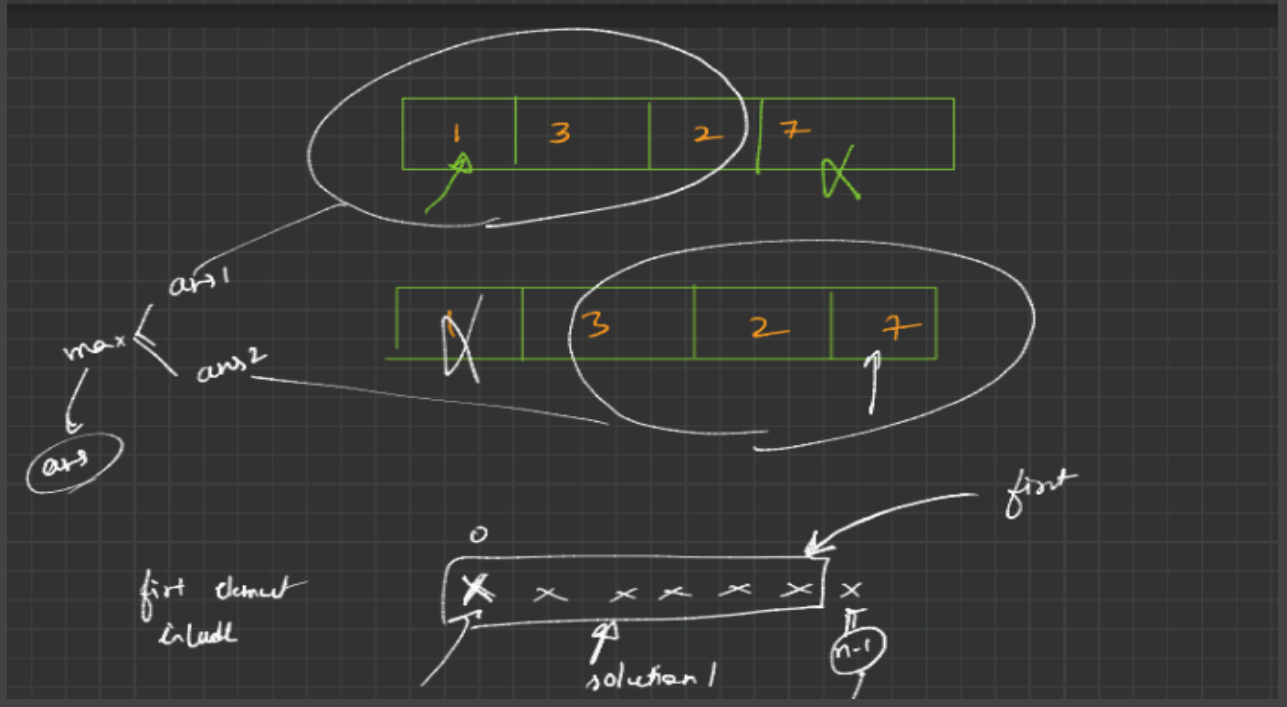


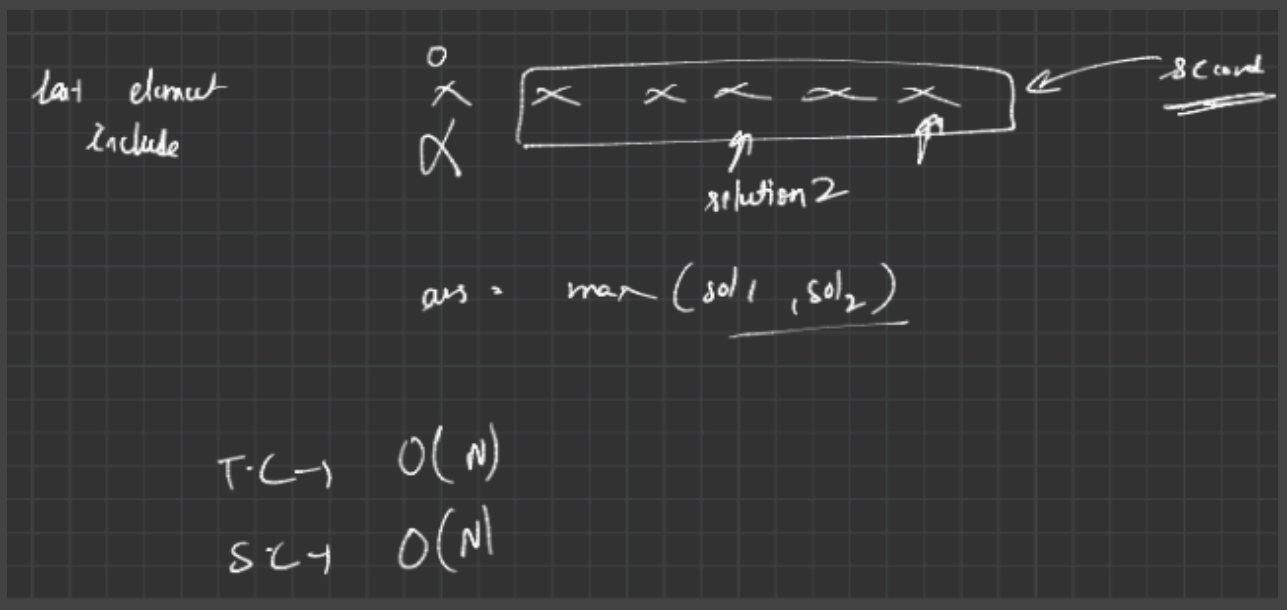


House Robbery Problem By Tabulation With Space Optimized Approach:

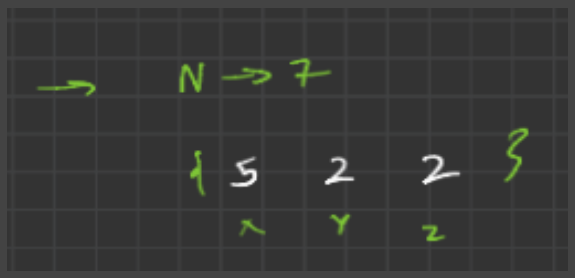


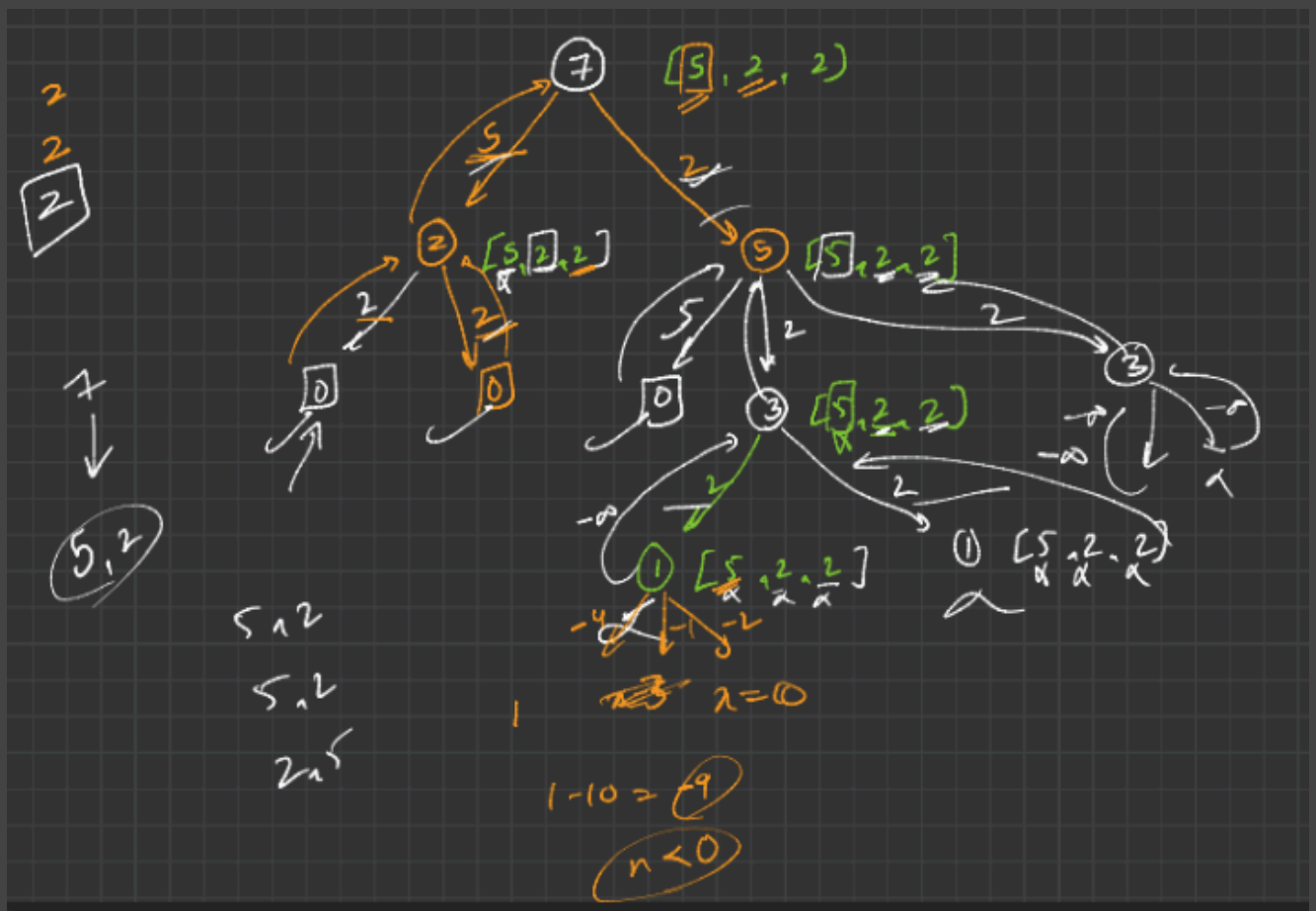

8
cond
9
7
8
9
8
7
last
9 + 7 → 16
first
last
ans →

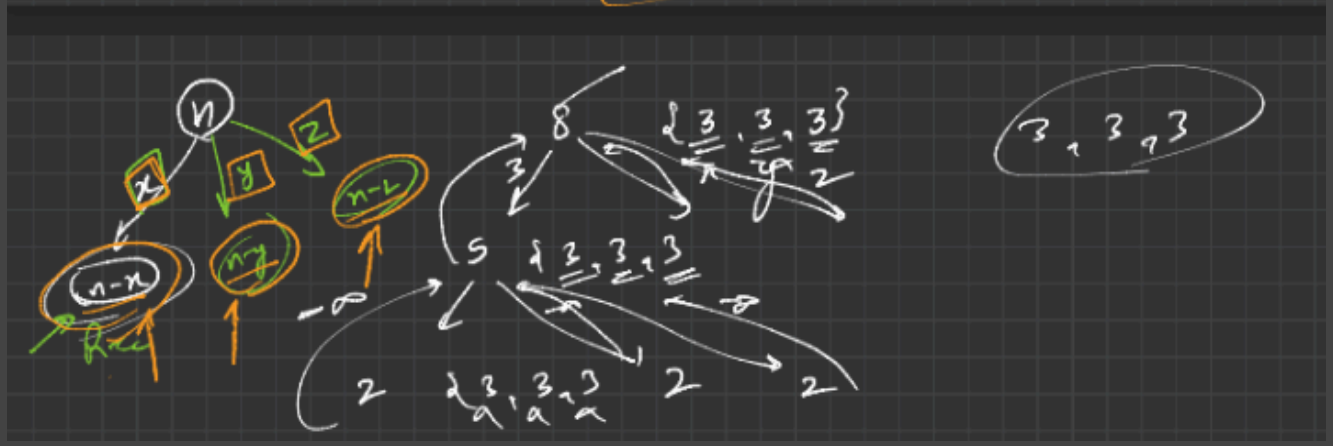

1
3
2
7
ans1
max
ans2
ans
3
2
7
first
0
first element include
solution 1
n-1

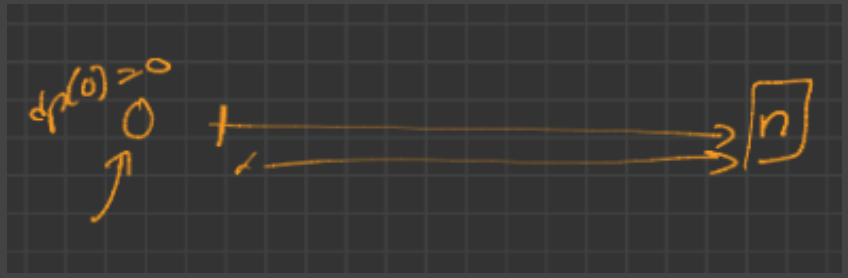

last element include
0
second
solution 2
ans = max (sol1 , sol2)
T.C → O(N)
S.C → O(N)

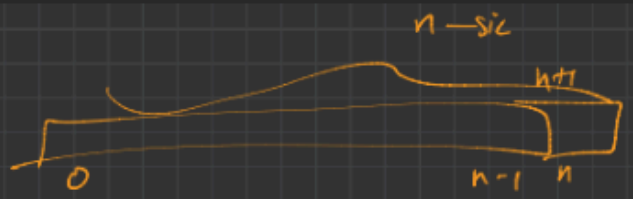
Cut Rod Into Segments X,Y,Z…. By Recursion / Top-Down Approach / Bottom-Up Approach:





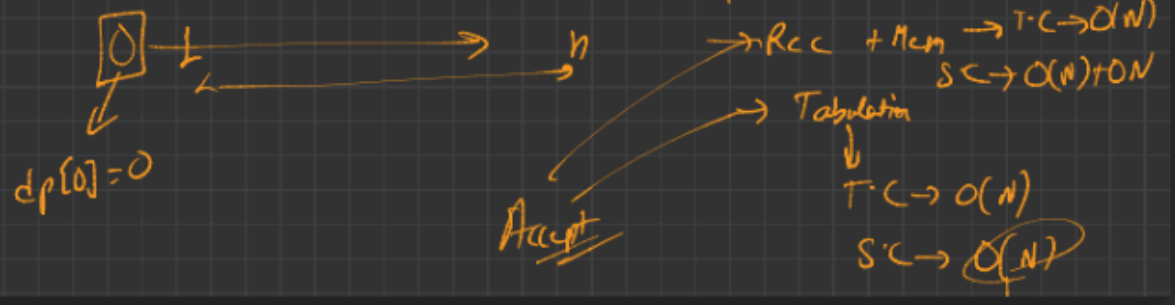




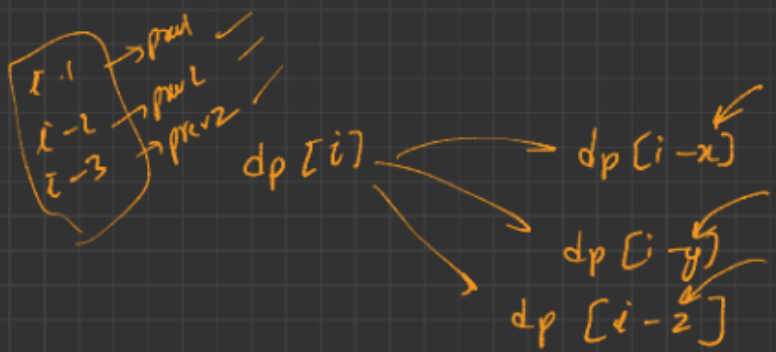


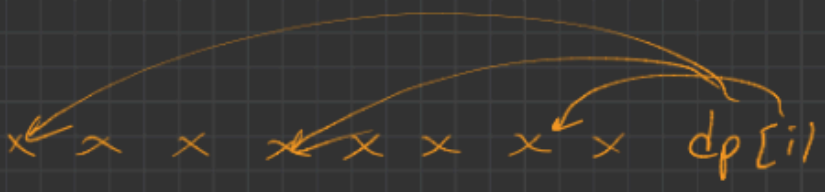
dp[n-1]

dp[n]

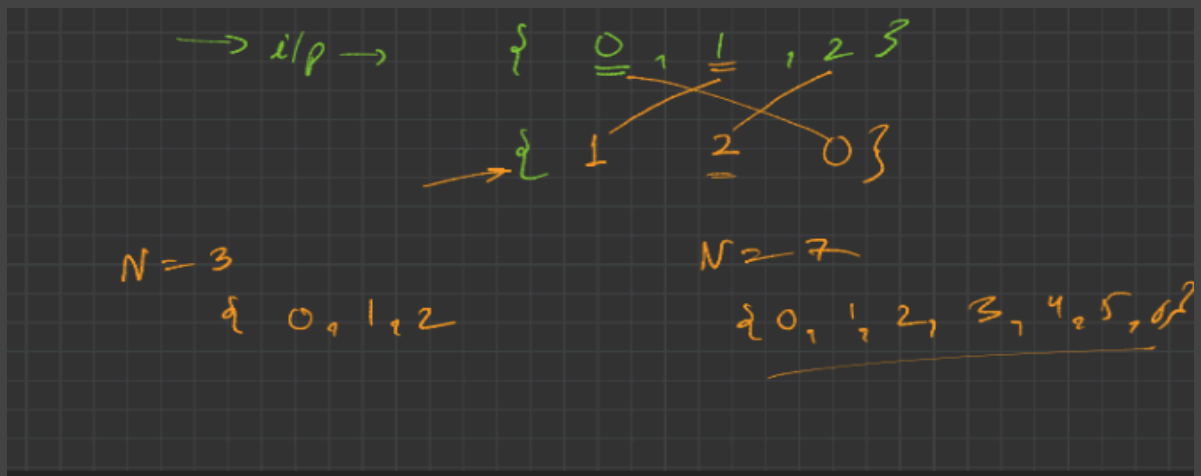


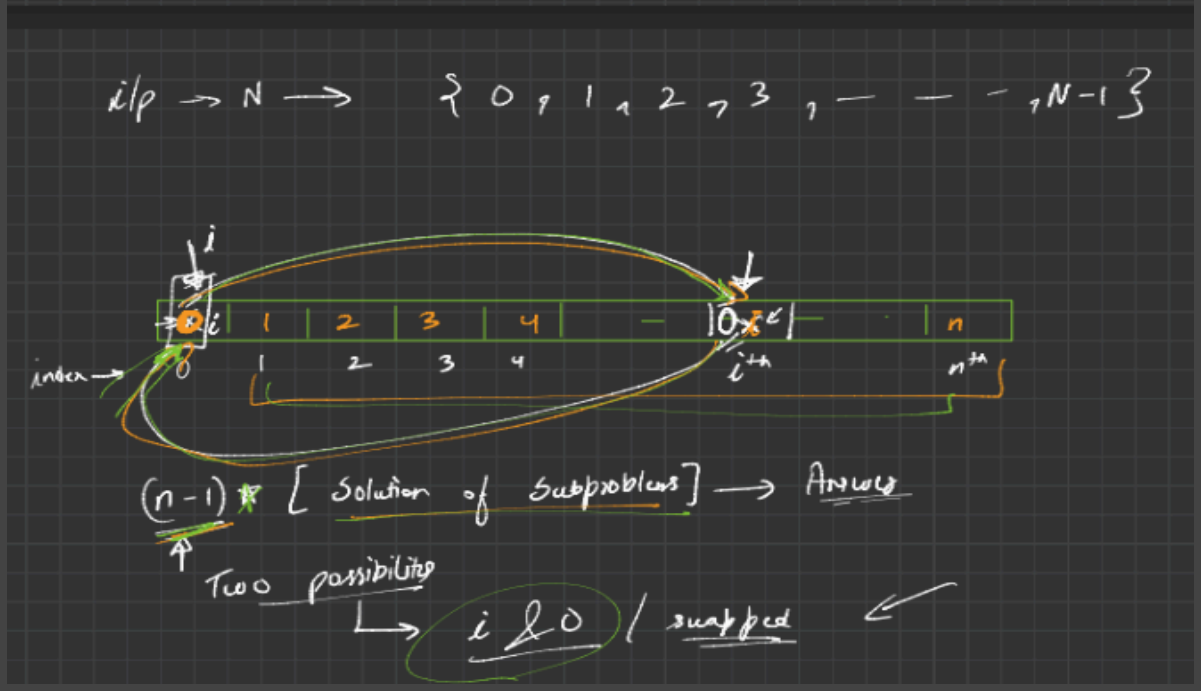
S·O → O(1) or less

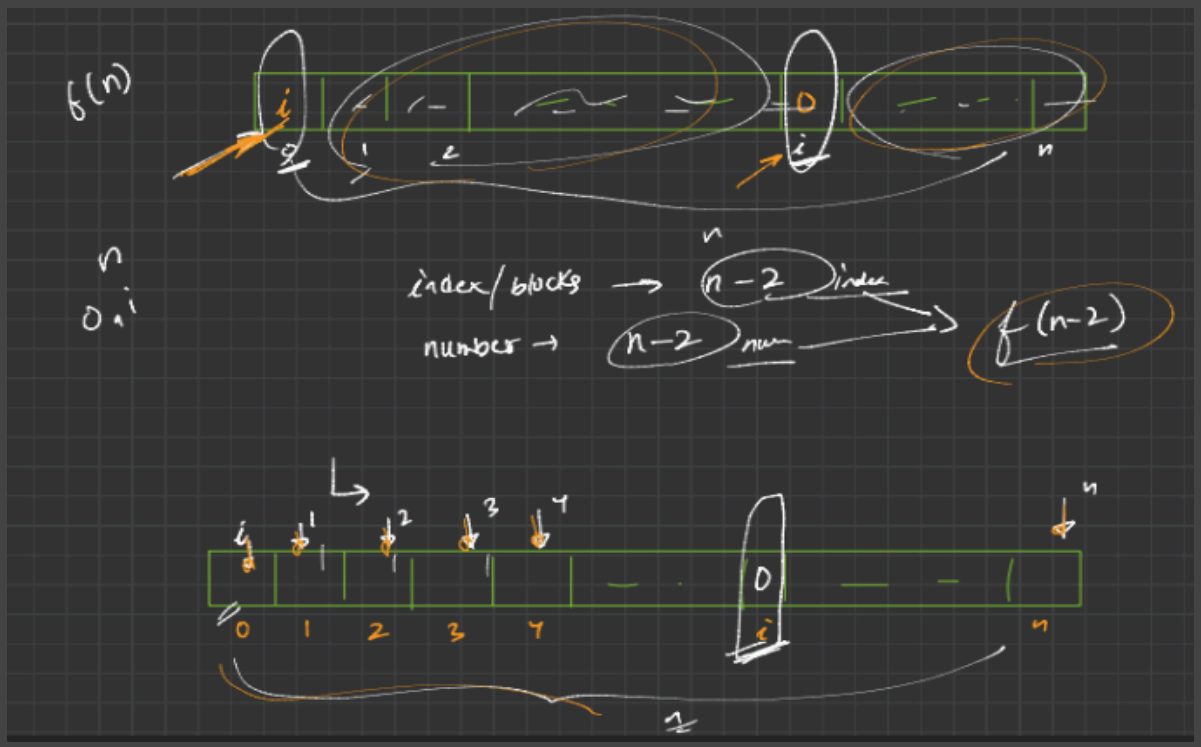


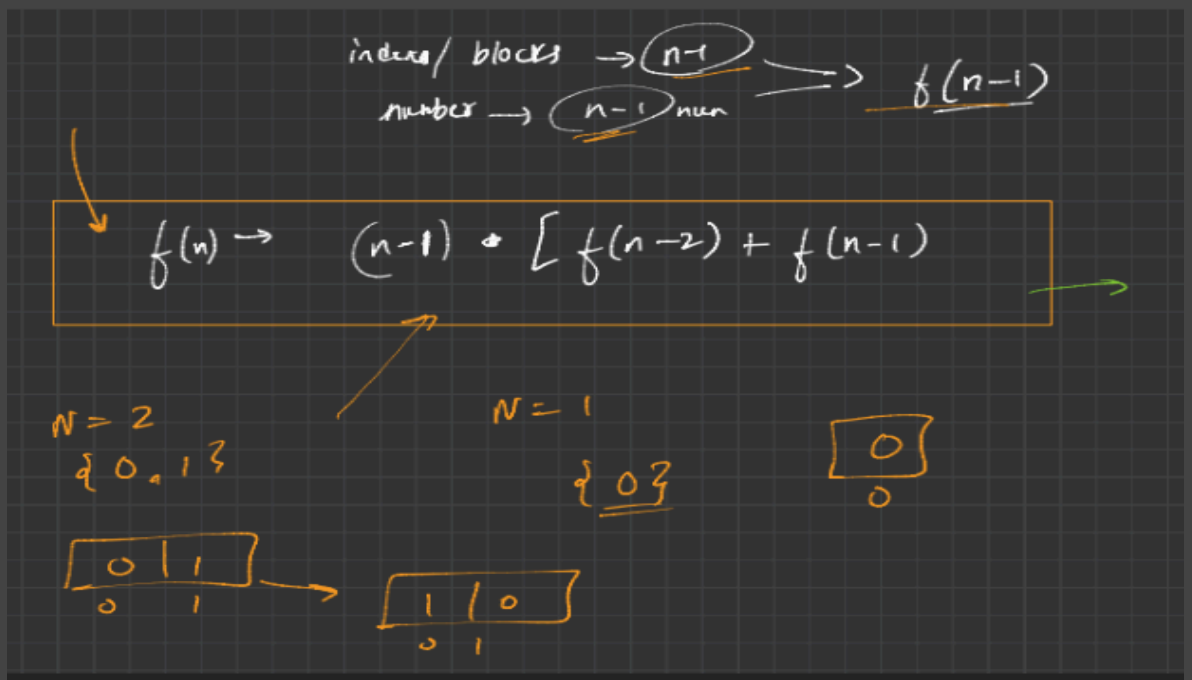


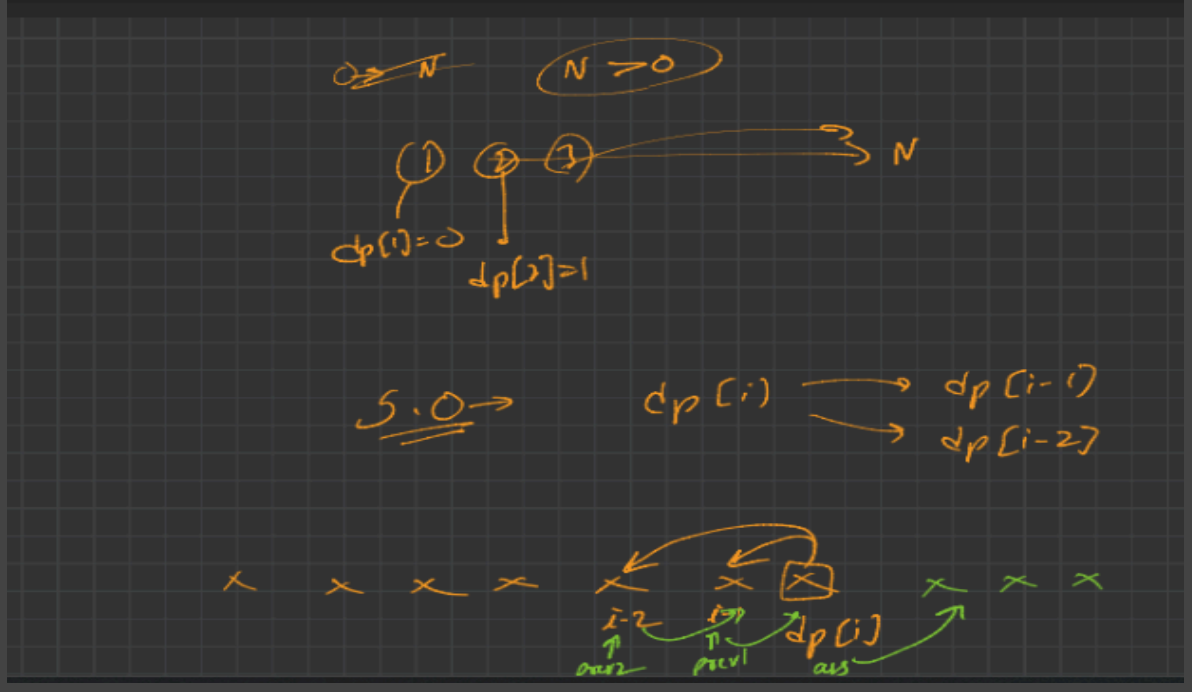
Count Derangements By Recursion / Top-Down Approach / Bottom-Up Approach / Space Optimization Approach:

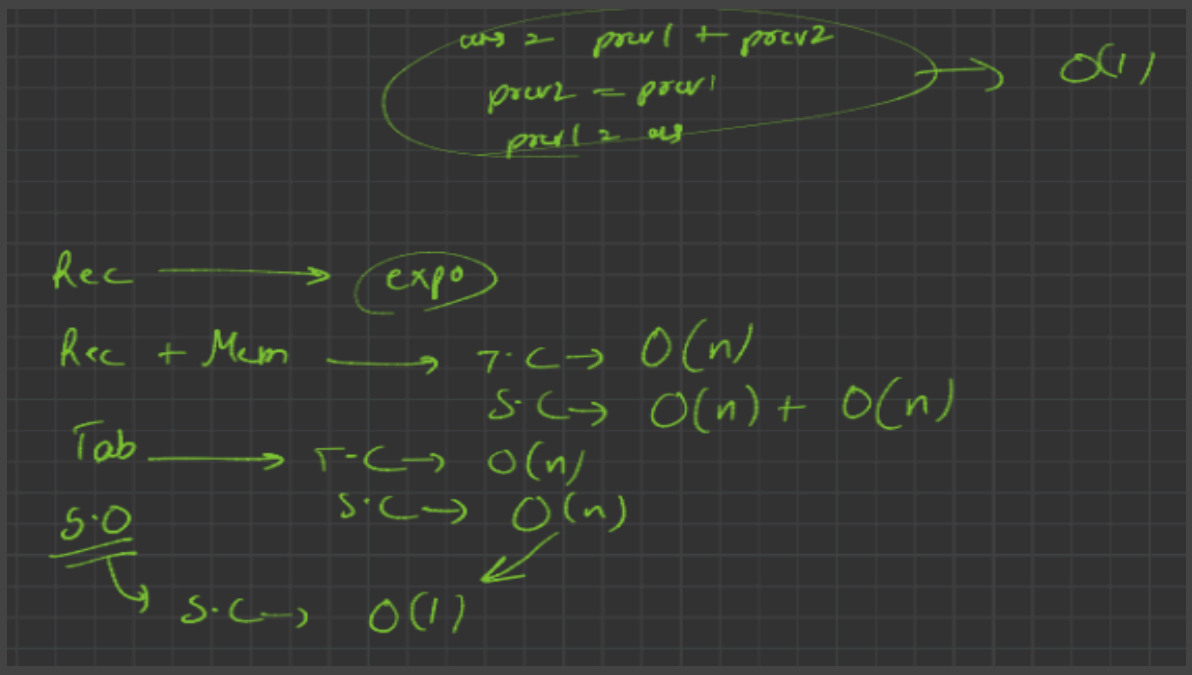




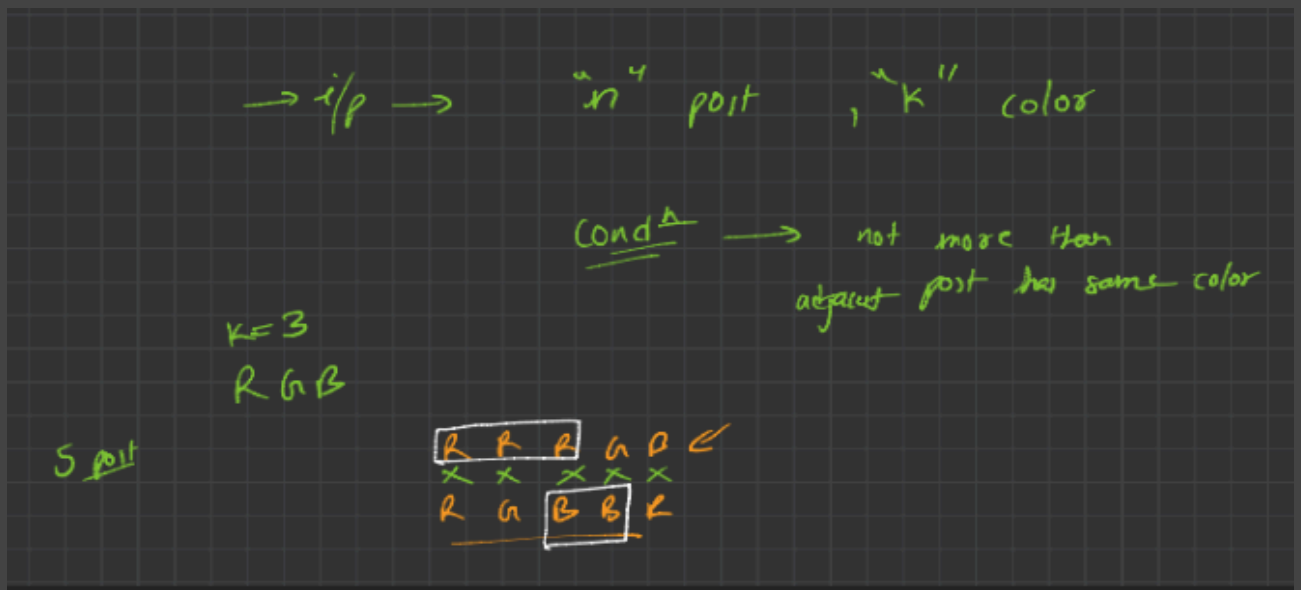


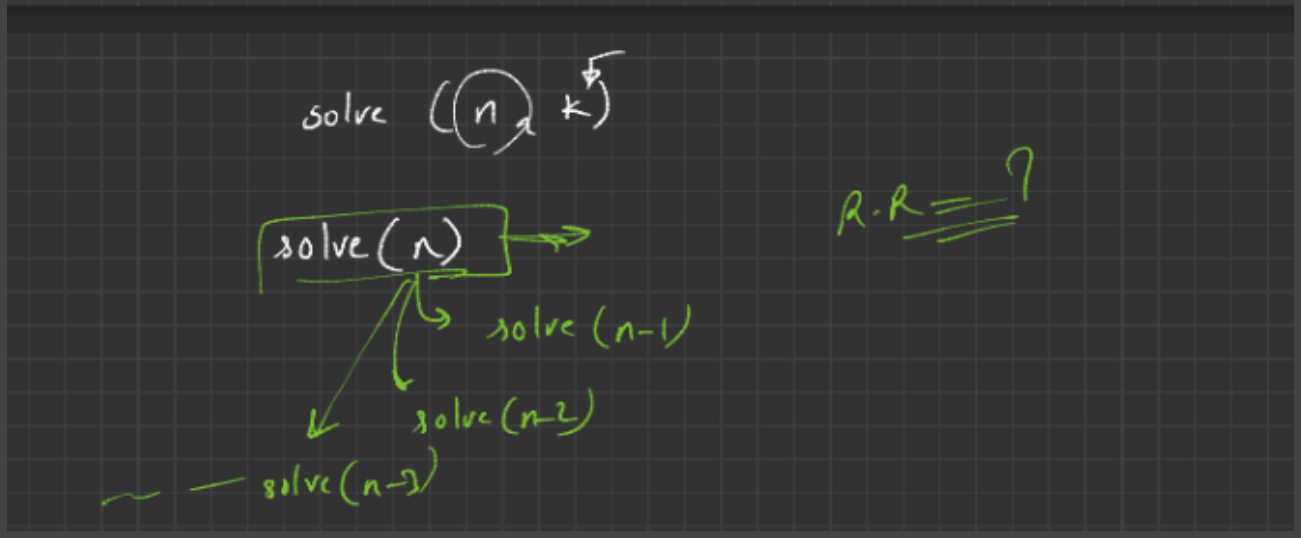

indexes / blocks → n-1
number → n-1 more
f(n-1)
f(n) → (n-1) * [f(n-2) + f(n-1)
N = 2
{0, 1}
N = 1
{0}
0
0
0 1
0 1
1 0
0 1

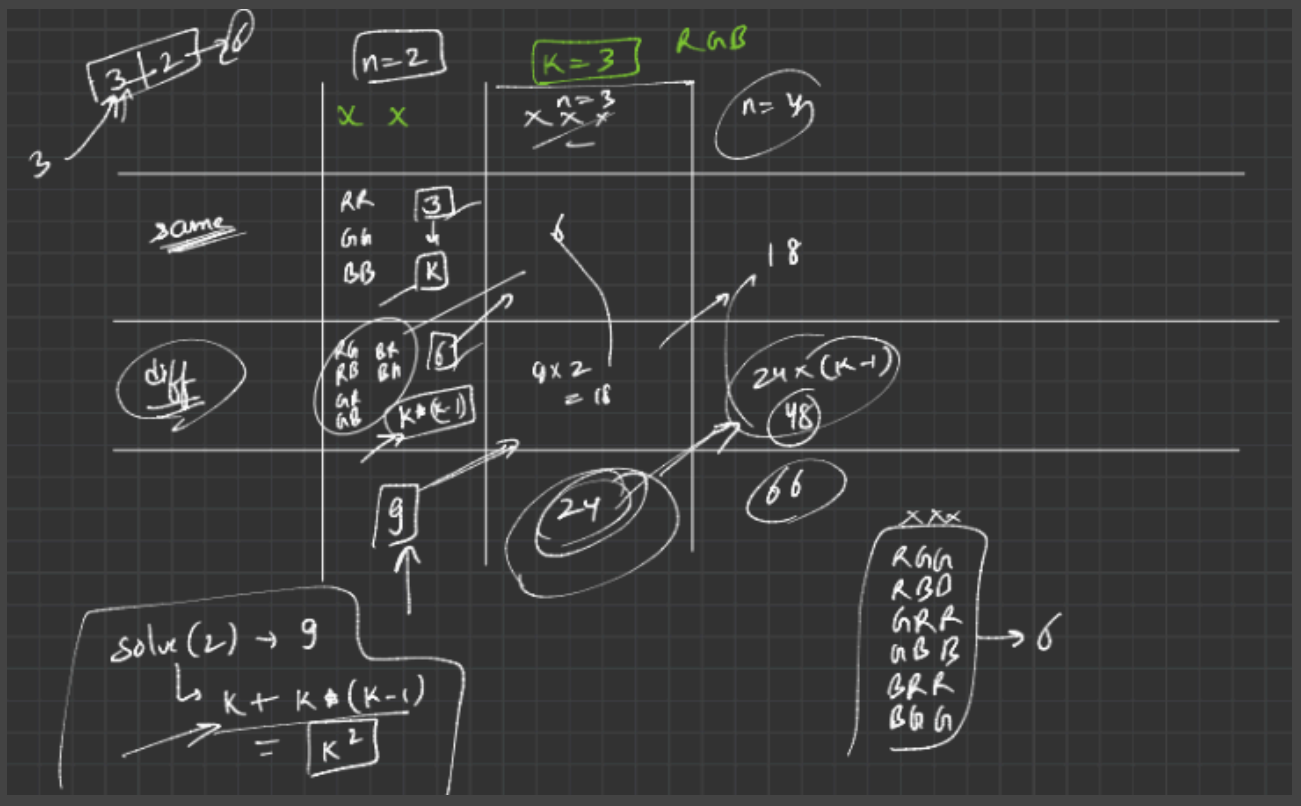

N
N >= 0
1
2
3
N
dp[1]=0
dp[2]=1
S.O →
dp[i]
dp[i-1]
dp[i-2]
i-2
i-1
i
dp[i]
prev2
prev1
ans

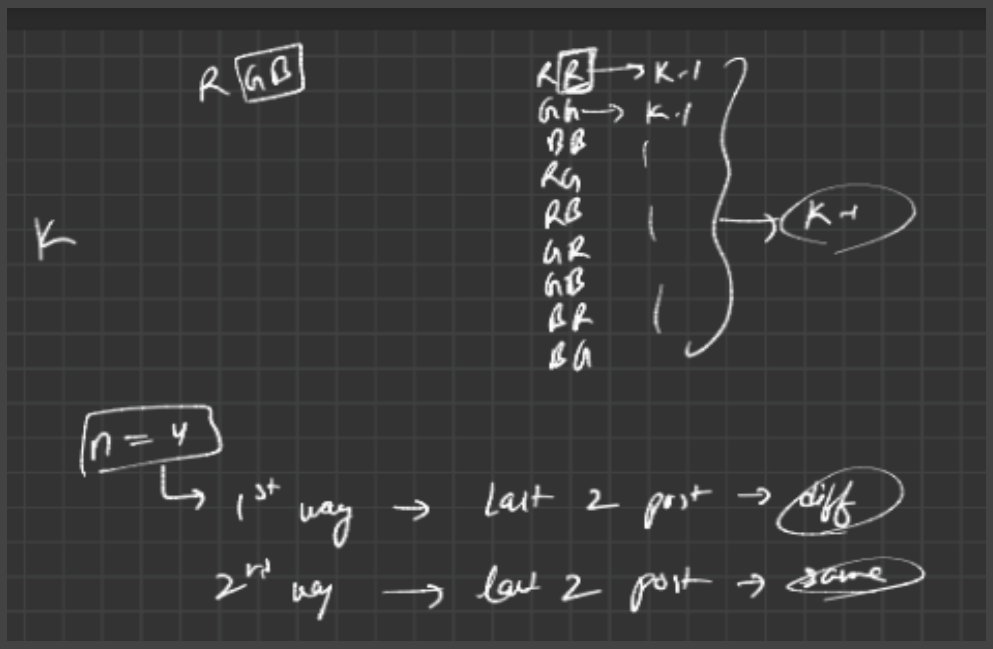

ans = prev1 + prev2
prev2 = prev1
prev1 = ans
O(1)
Rec → expo
Rec + Mem → T.C → O(n)
S.C → O(n) + O(n)
Tab → T.C → O(n)
S.C → O(n)
S.O
S.C → O(1)

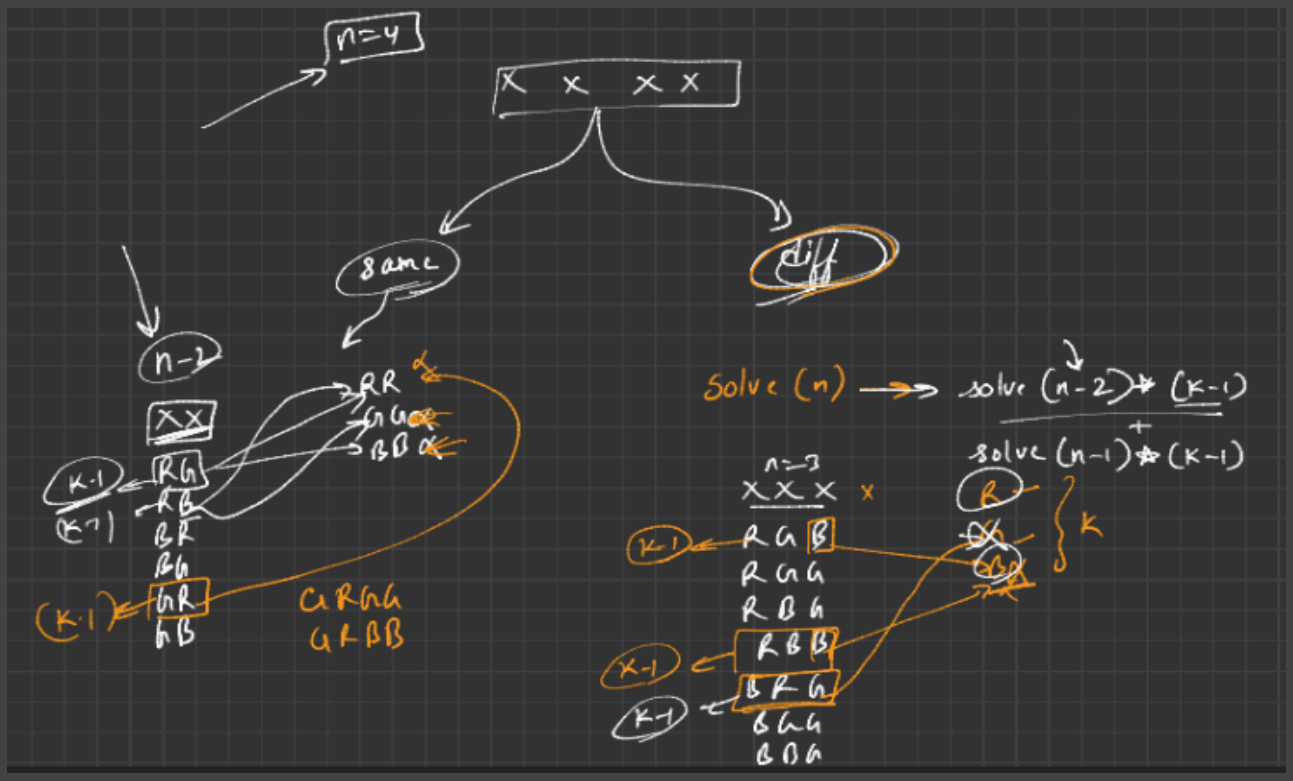
Painting Fence Problem By Recursion / Top-Down Approach / Bottom-Up Approach / Space Optimization Approach:

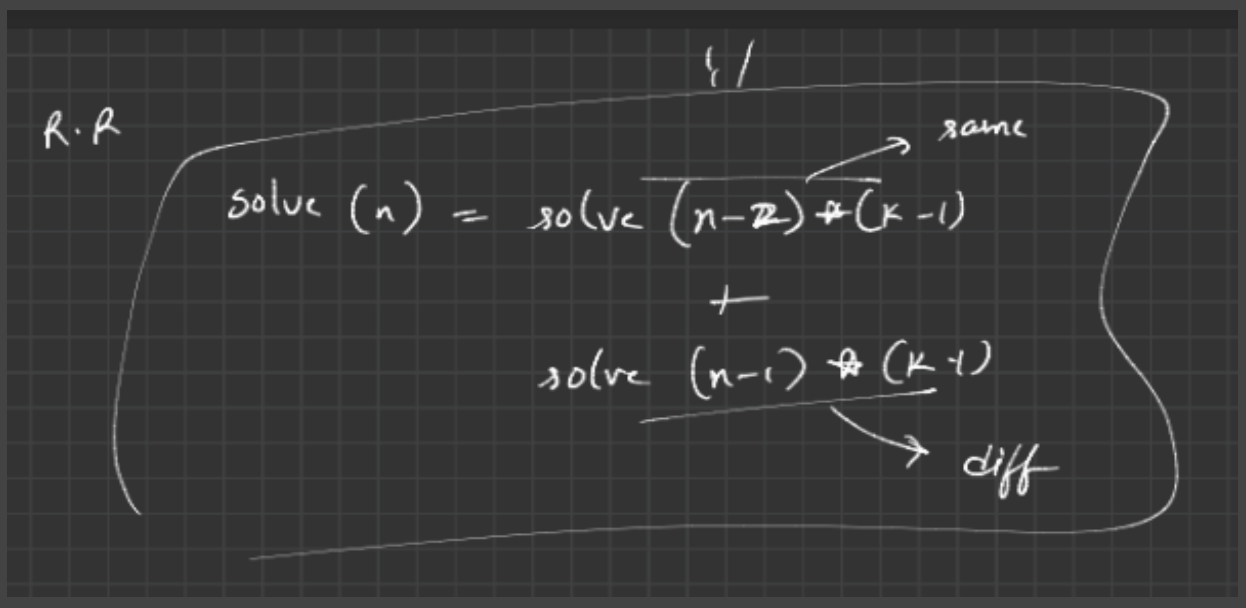


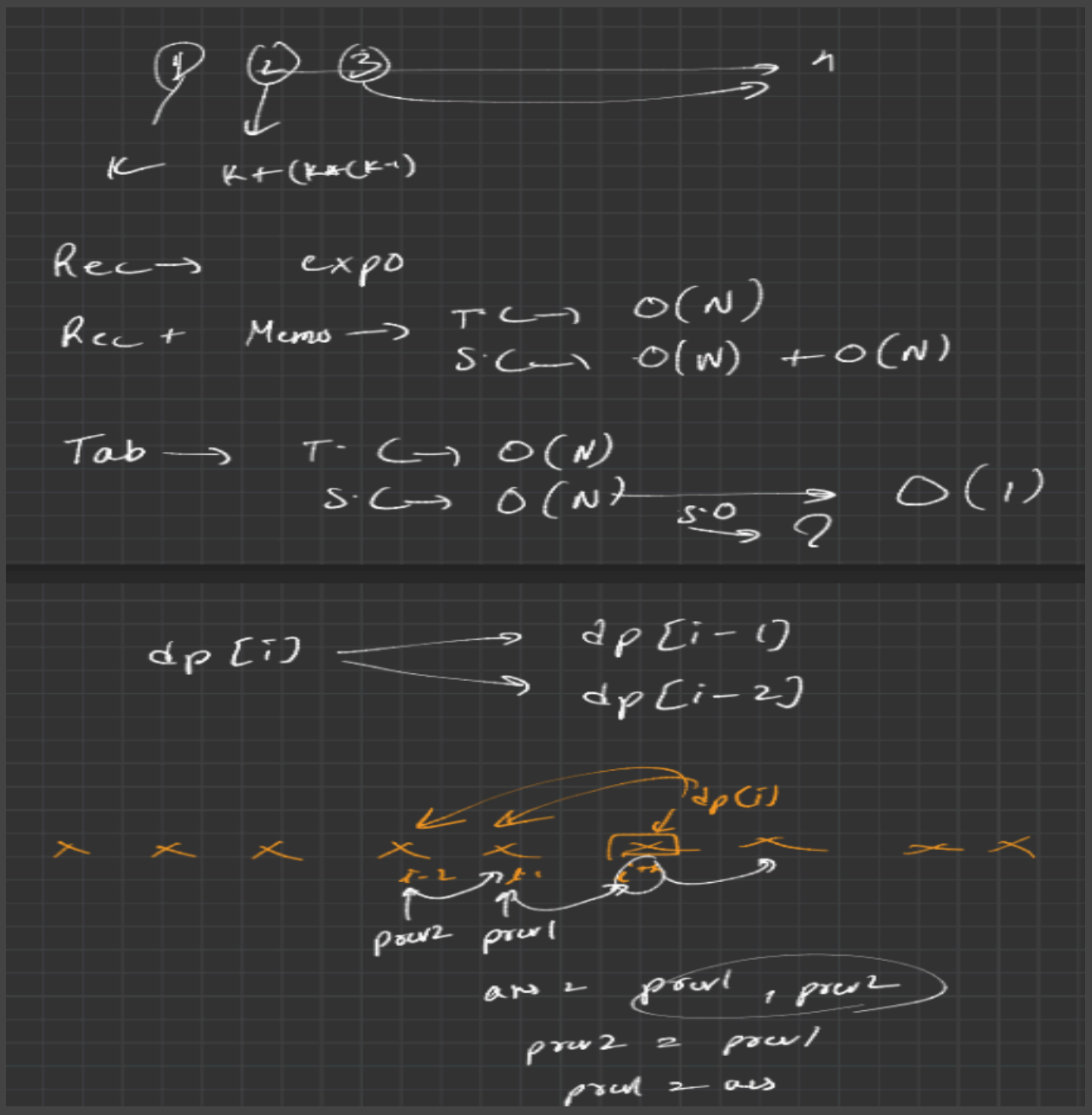




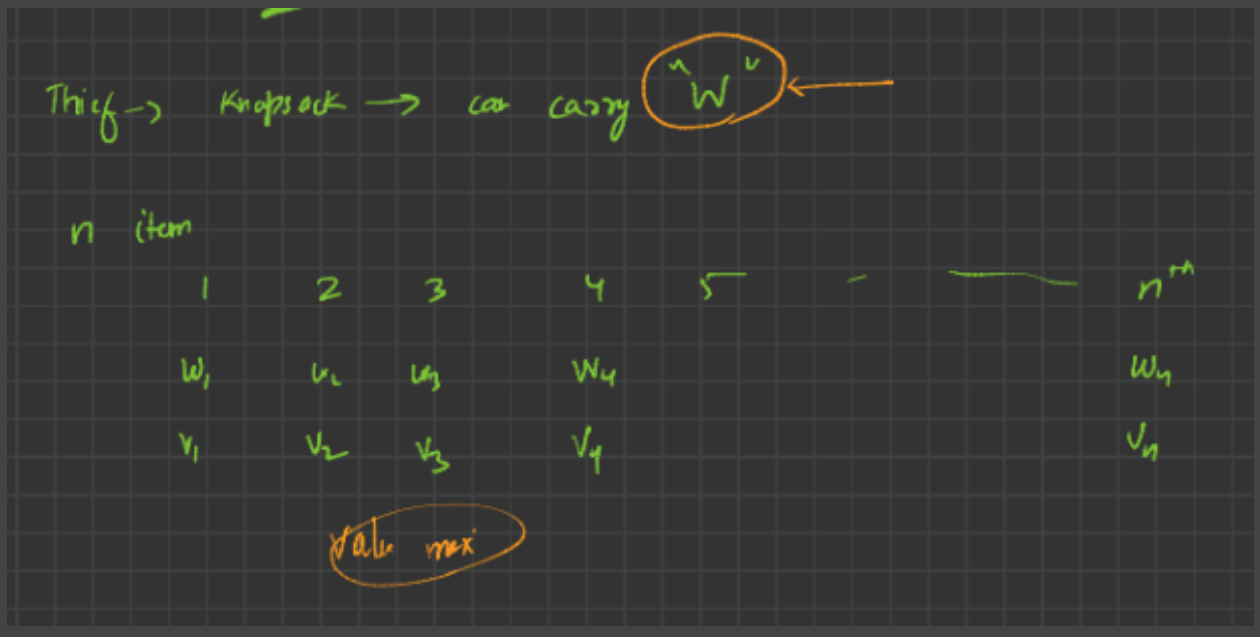

R G B
K
R B → K-1
G G → K-1
B B
R G
R B
G R
G B
B R
B G
K-1
n = 4
1st way → last 2 post → diff
2nd way → last 2 post → same

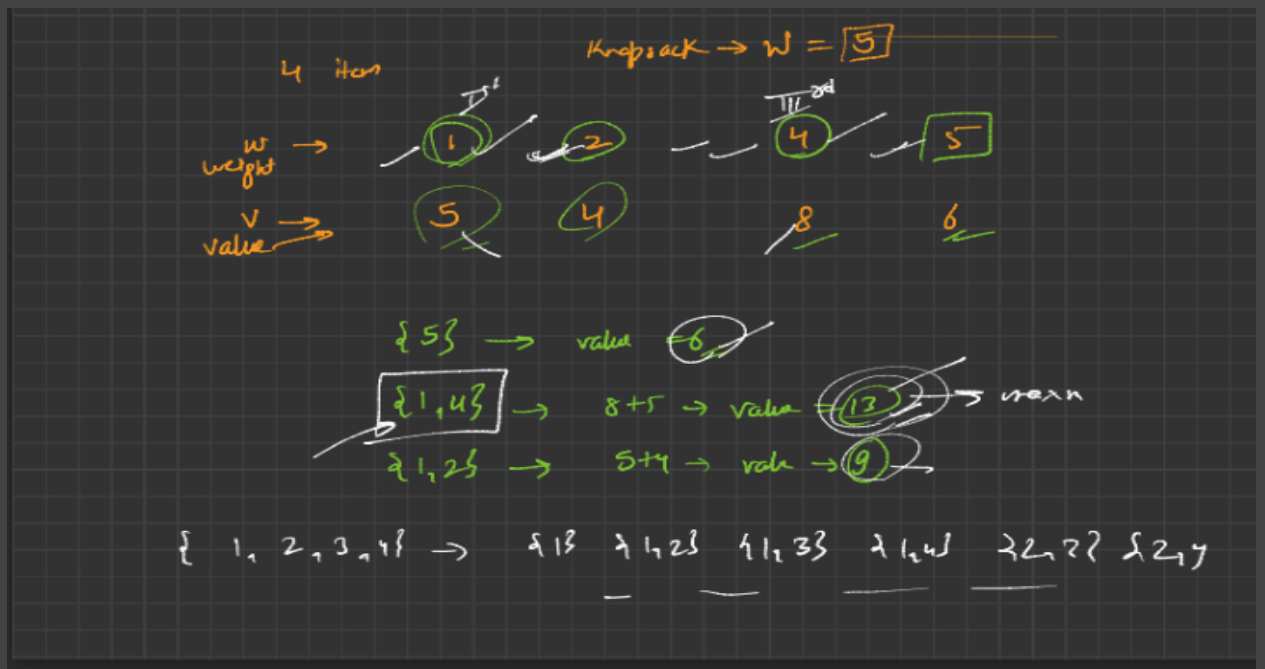

n=4
X X X X
same
diff
n-2
XX
RG
RB
BR
BG
GR
GB
K-1
(K-1)
RR
GG
BB
GRGG
GRBB
Solve (n) → solve (n-2) * (K-1)
+
solve (n-1) * (K-1)
n=3
XXX X
RGB
RGG
RBG
RBB
BRG
BGG
BBG
R
K

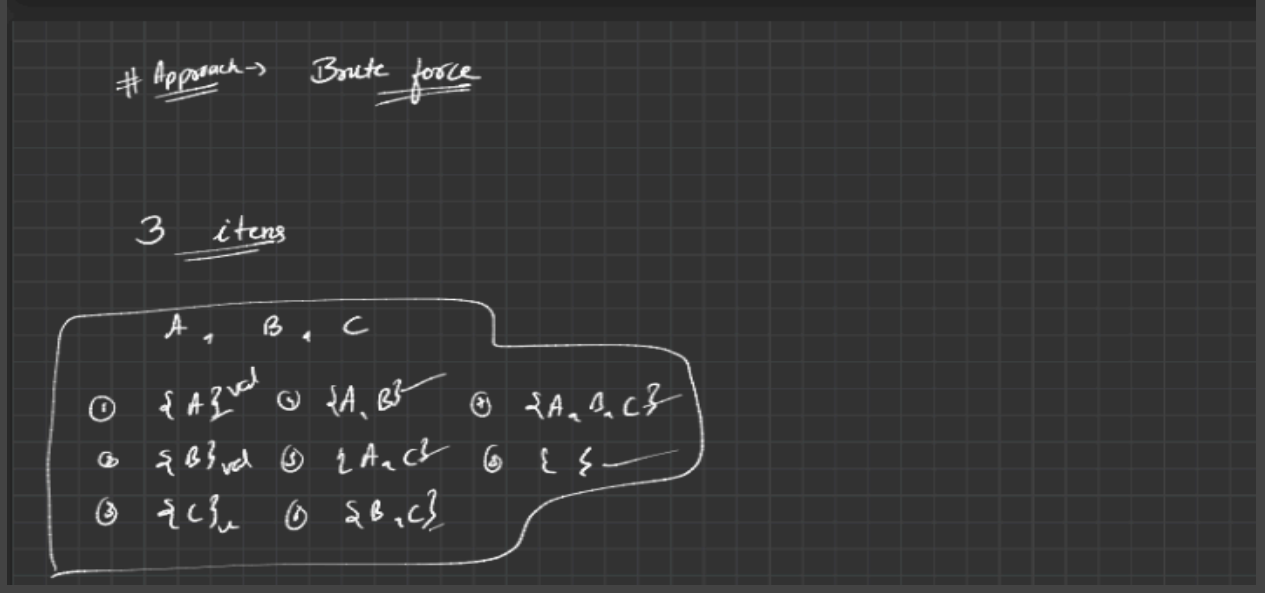

R.R
Solve (n) = solve (n-2) * (K-1)
same
+
solve (n-1) * (K-1)
diff

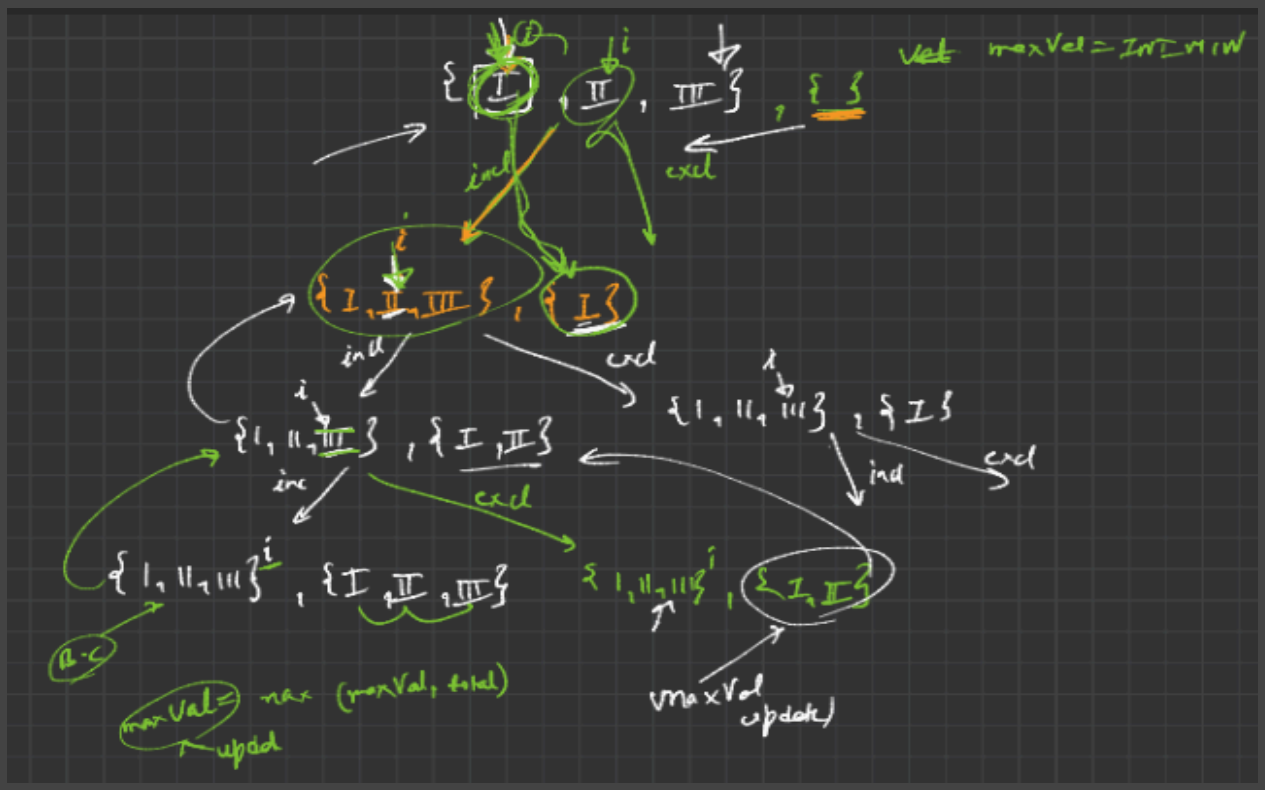


0/1 KnapSack Problem By Recursion / Top-Down Approach / Bottom-Up Approach / Space Optimization Approach:




4 item
Knapsack → W = 5
weight → W
value → V
1
2
4
5
5
4
8
6
{5} → value = 6
{1,4} → 8+5 → value = 13 → max
{1,2} → 5+4 → val → 9
{1, 2, 3, 4} → {1} {1,2} {1,3} {1,4} {2,?} {2,4}


Approach → Brute force
3 items
A, B, C
① {A} val
② {A, B}
③ {A, B, C}
④ {B} val
⑤ {A, C}
⑥ { }
⑦ {C}
⑧ {B, C}


Val maxVal = INT_MIN
{I, II, III}, { }
incl
excl
{I, II, III}, {I}
{I, II, III}, {I, III}
{I, II, III}, {I}
{I, II, III}, {I, II, III}
{I, II, III}, {I, II}
B-C
maxVal = max (maxVal, total)
update
maxVal update

W

B.C

1 item

Inc

exc

W

```
if (wt[0] <= maxWeight)
    return val[0]
else
    return 0;
```

① Recursive + Memoization (Top-Down)

Step ① → dp array → (-1)

Step 2 → Rec call → dp → ans store

return

step 3 → B.C → if dp array contain ans

then return

② Bottom-up Approach (Tabulation)

nrows → maxWeight+1

✗ → n-1

1 → n-1

rows

col=0 <= maxWeight

0

rows → n

col → maxWeight+1

0 → maxWg

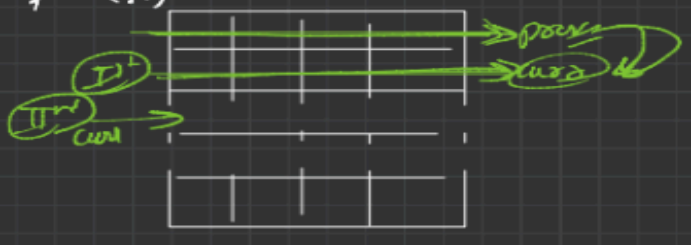
for(=0 <=maxWeight)

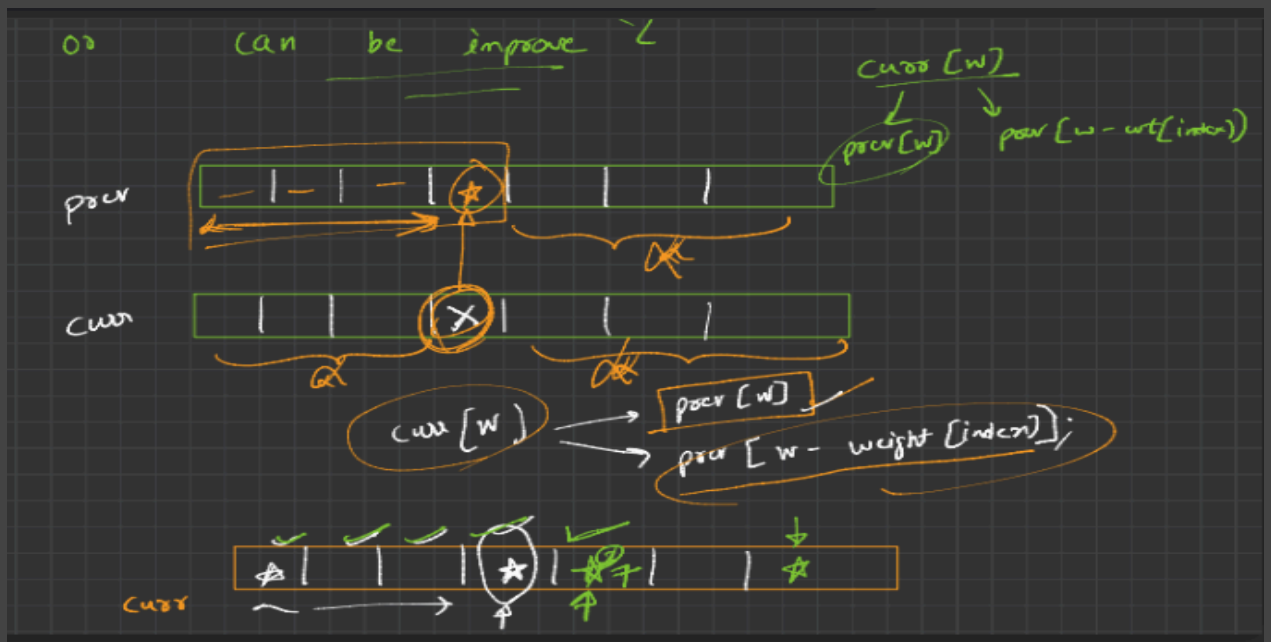
✗ → n-1

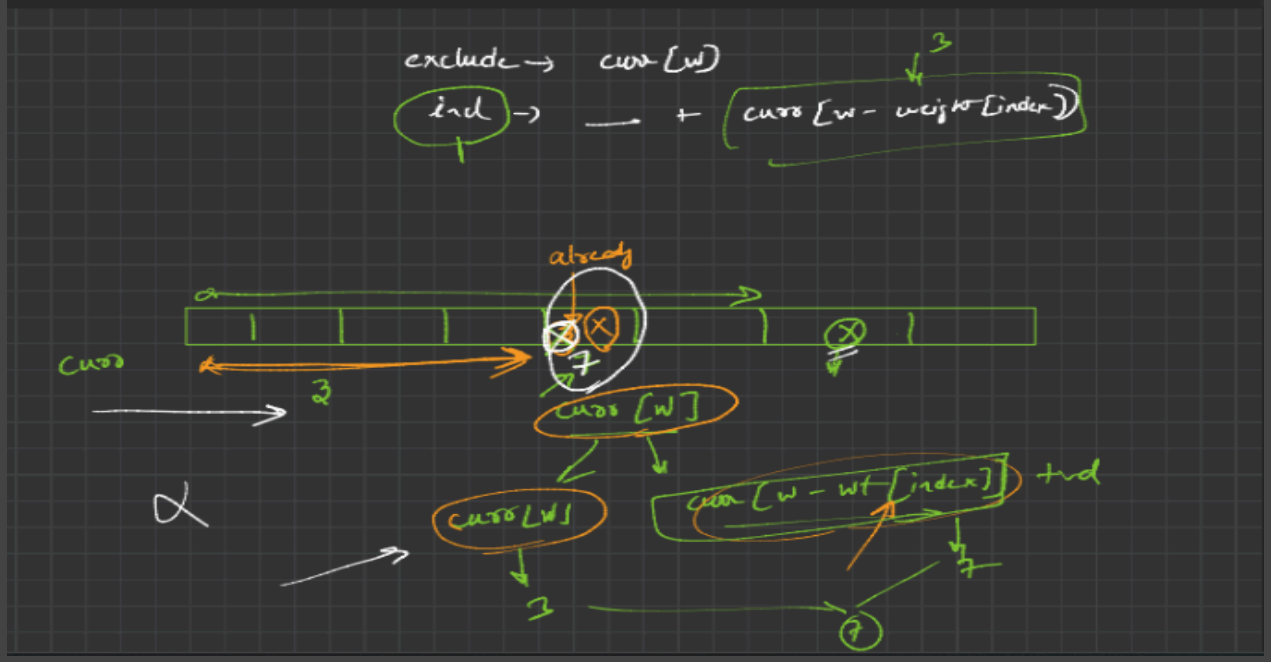
1 → n-1

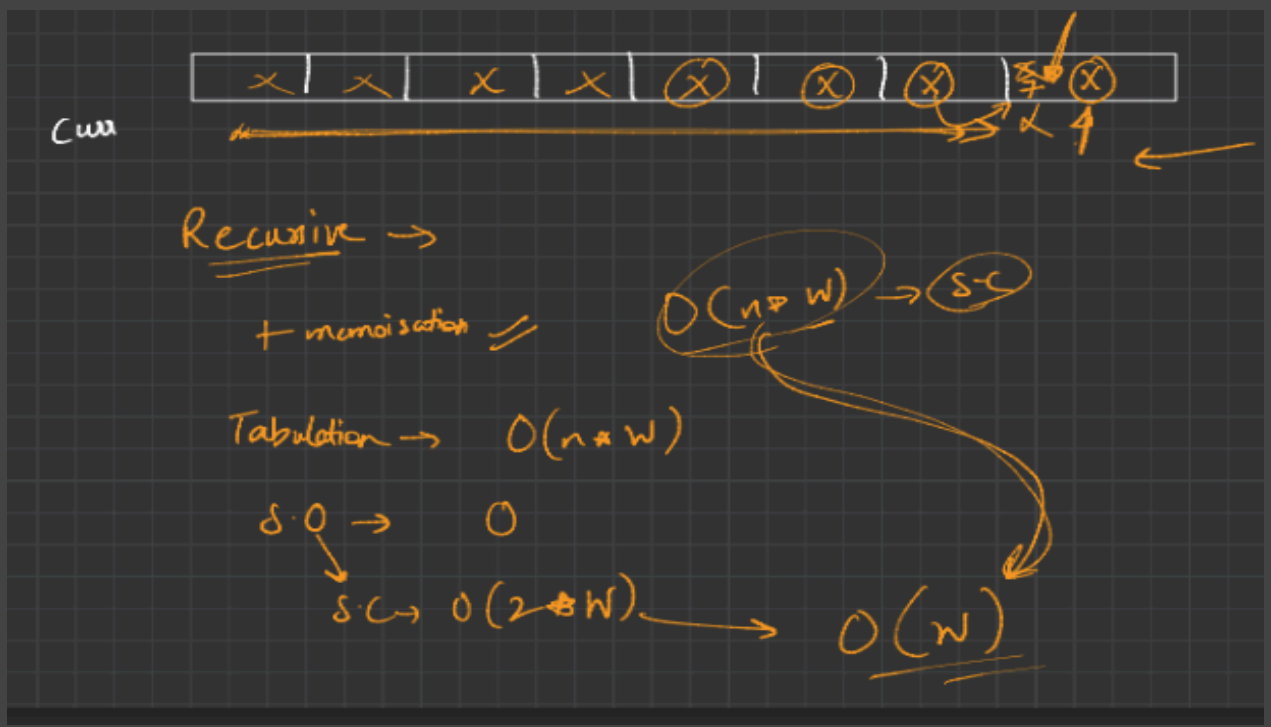
for(=1 , <n)

S.O →

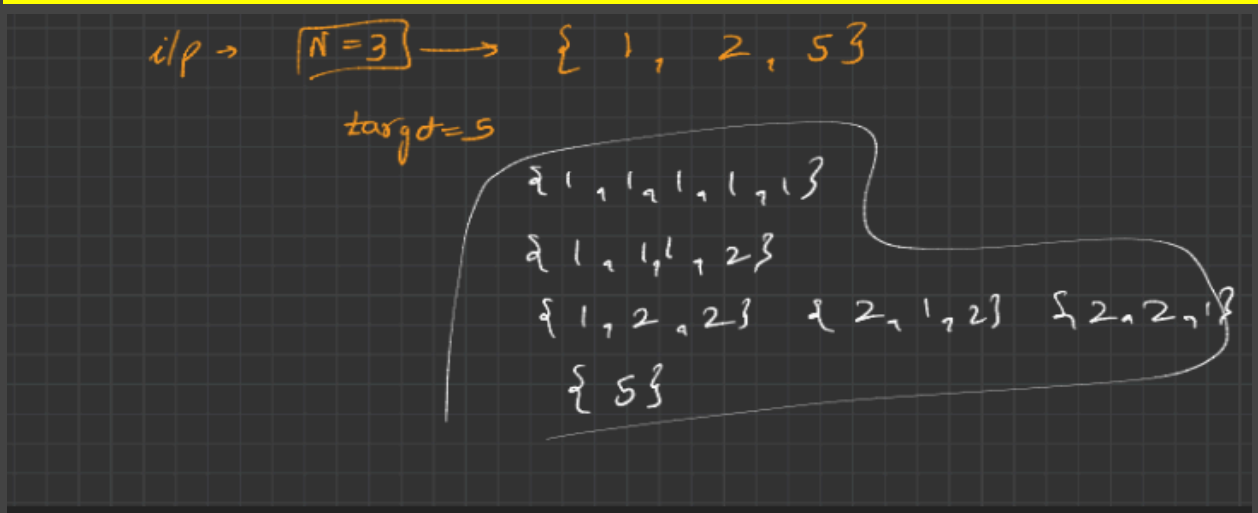


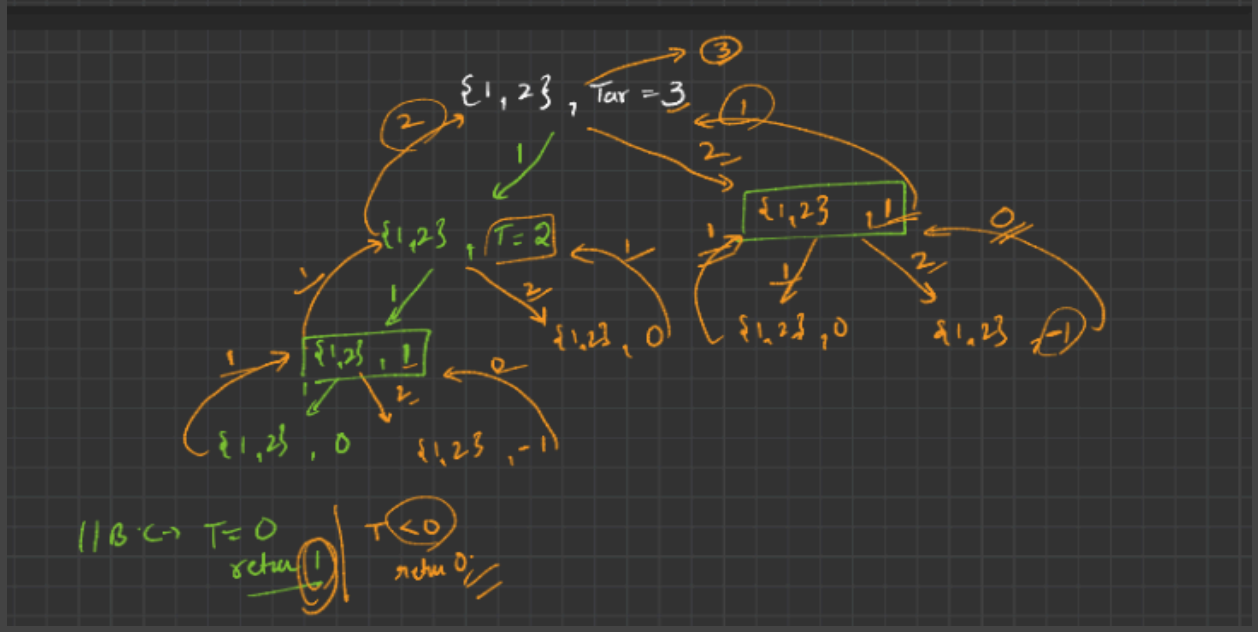

O3 can be improve
curr[w]
prev[w]
prev[w - wt[index]]
prev
curr
curr[w] → prev[w]
prev[w - weight[index]];
curr

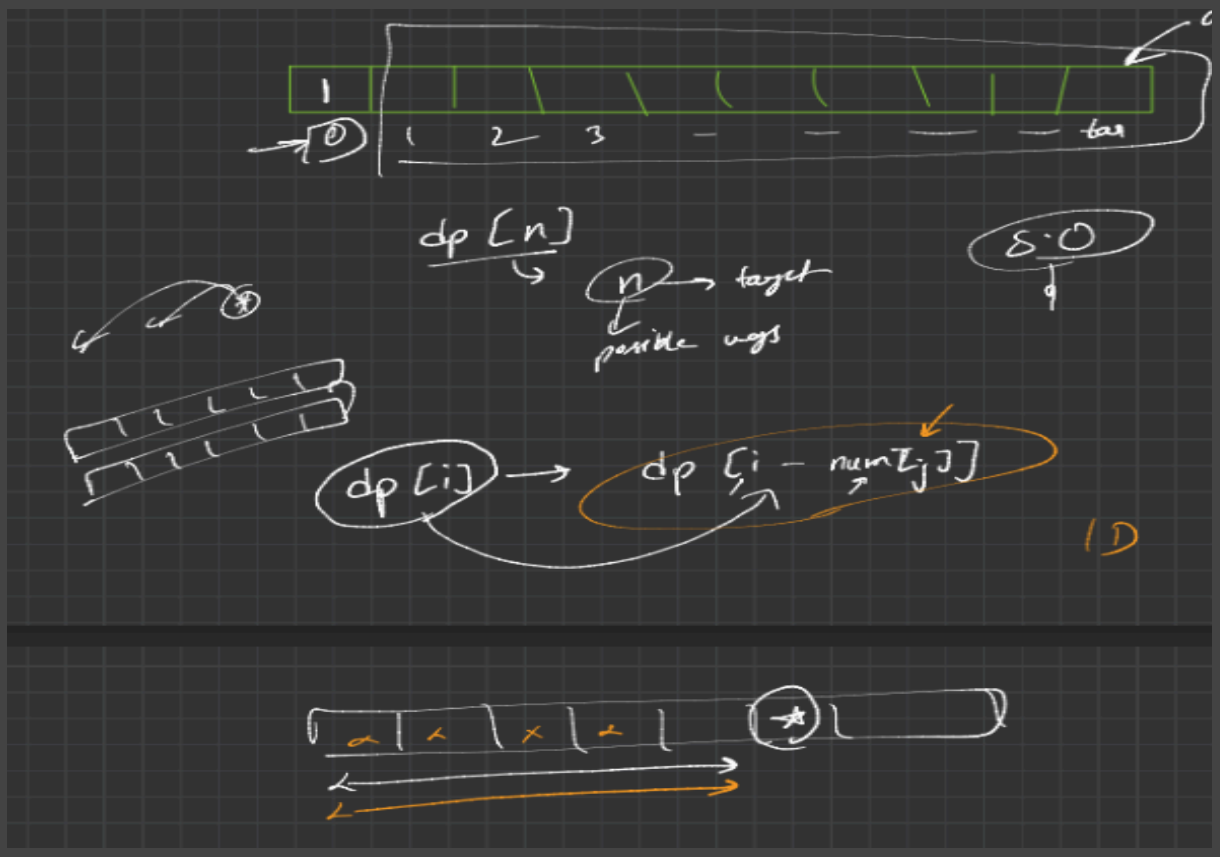

exclude → curr[w]
incl → + curr[w - weight[index]]
already
curr
curr[w]
curr[w]
curr[w - wt[index]]
incl


curr
Recursive →
+ memoisation
O(n * W) → S.C
Tabulation → O(n * W)
S.C → O(2 * W) → O(W)

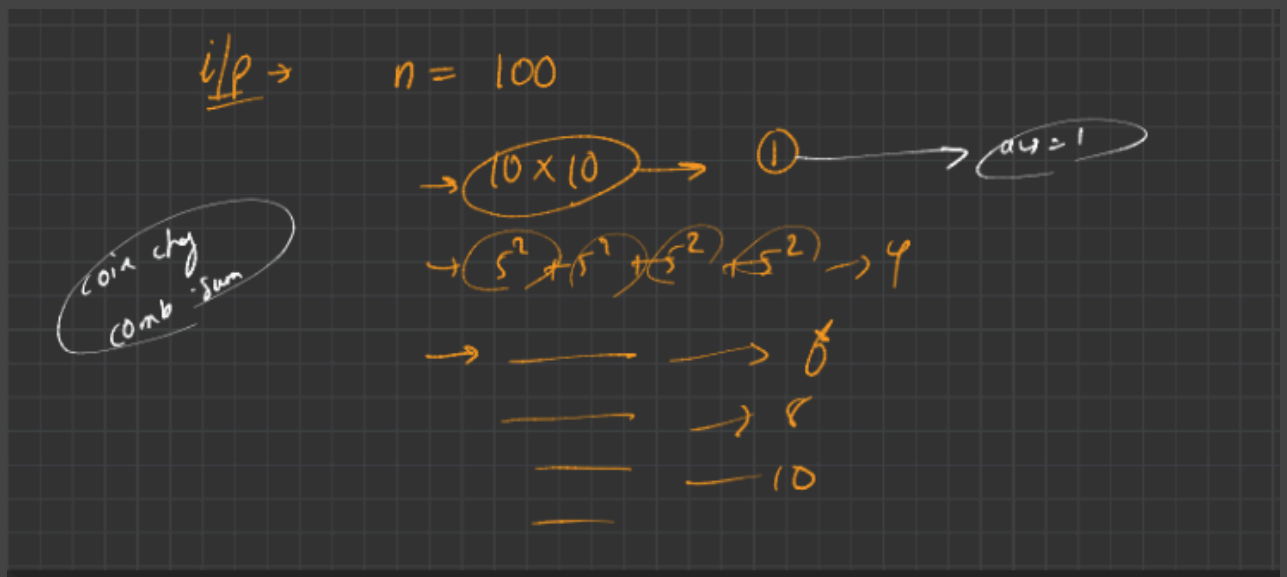
Combination Sum Problem By Recursion / Top-Down / Bottom-Up Approach:

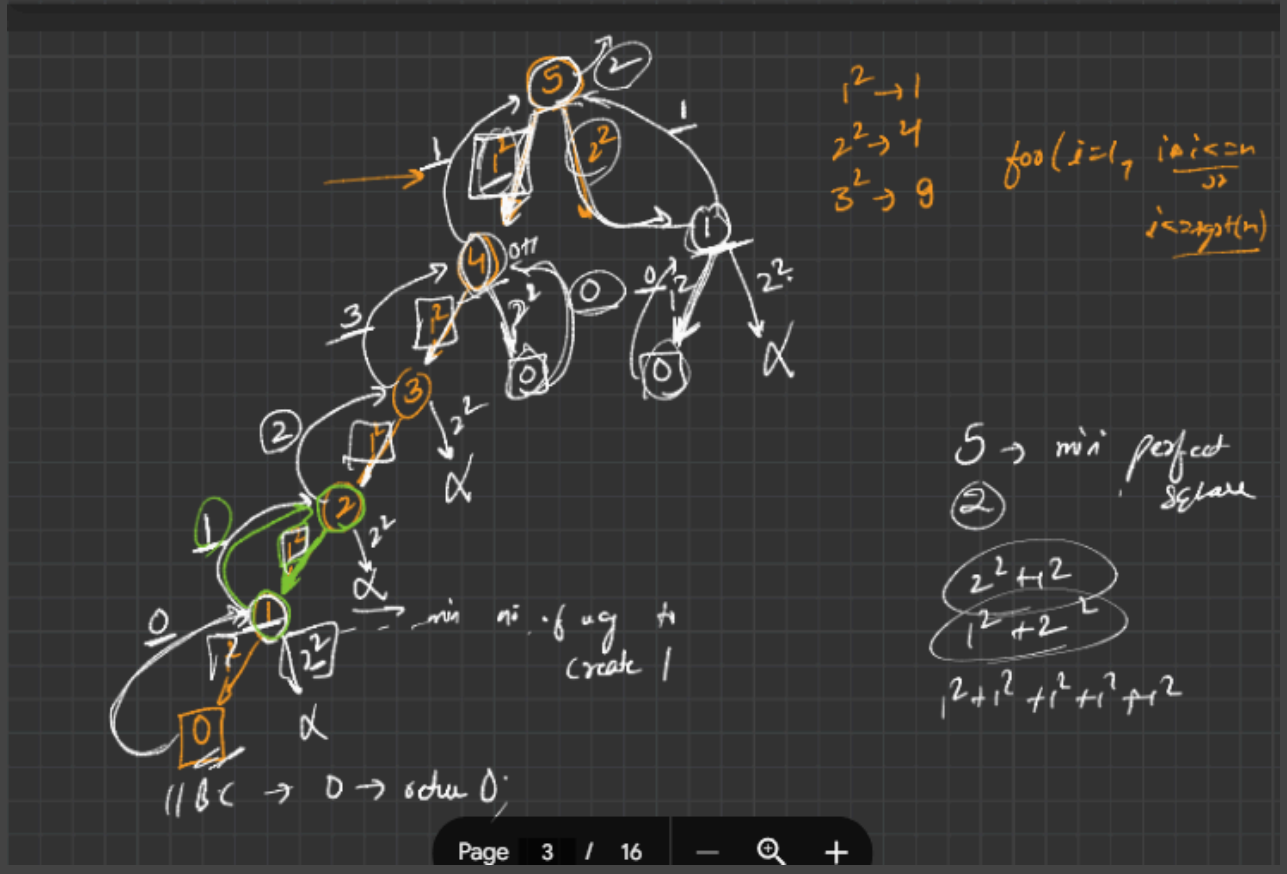


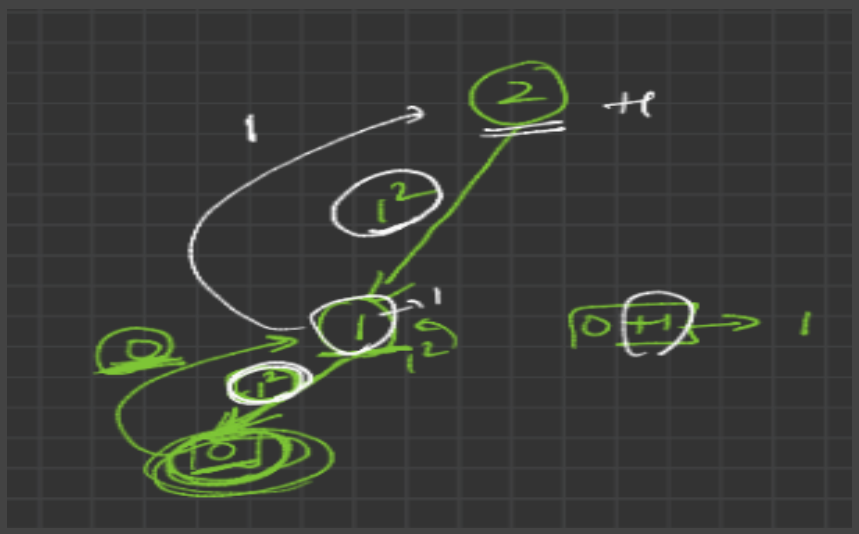


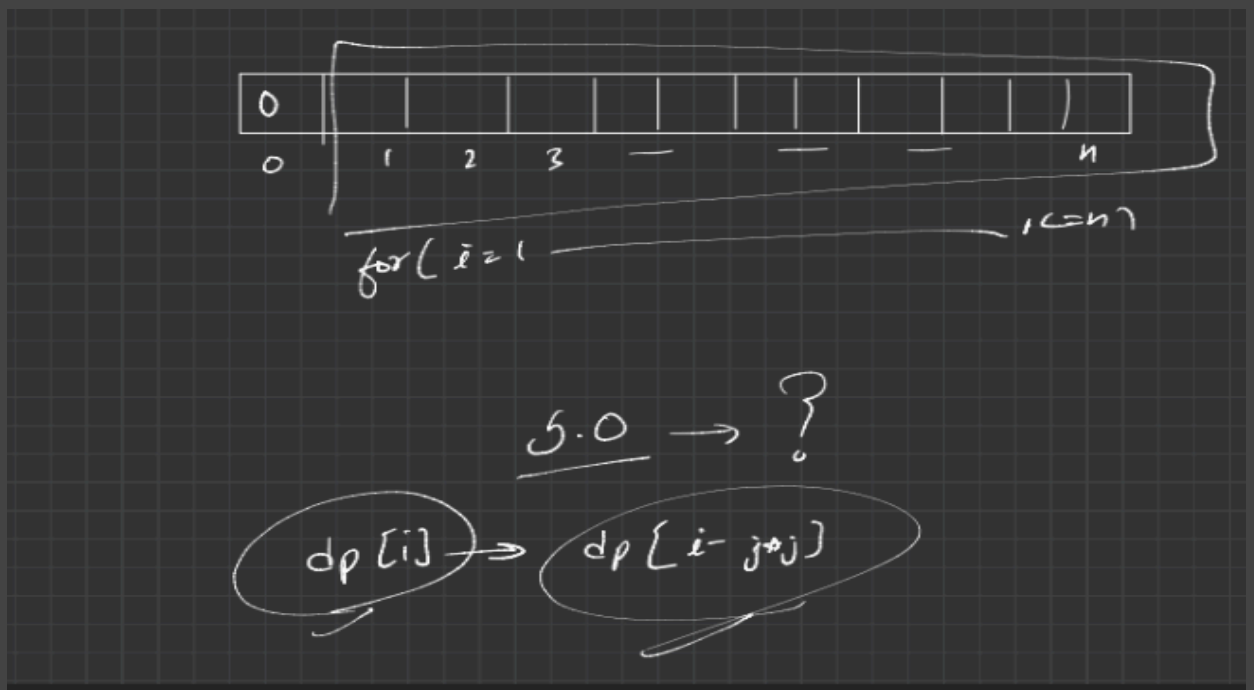


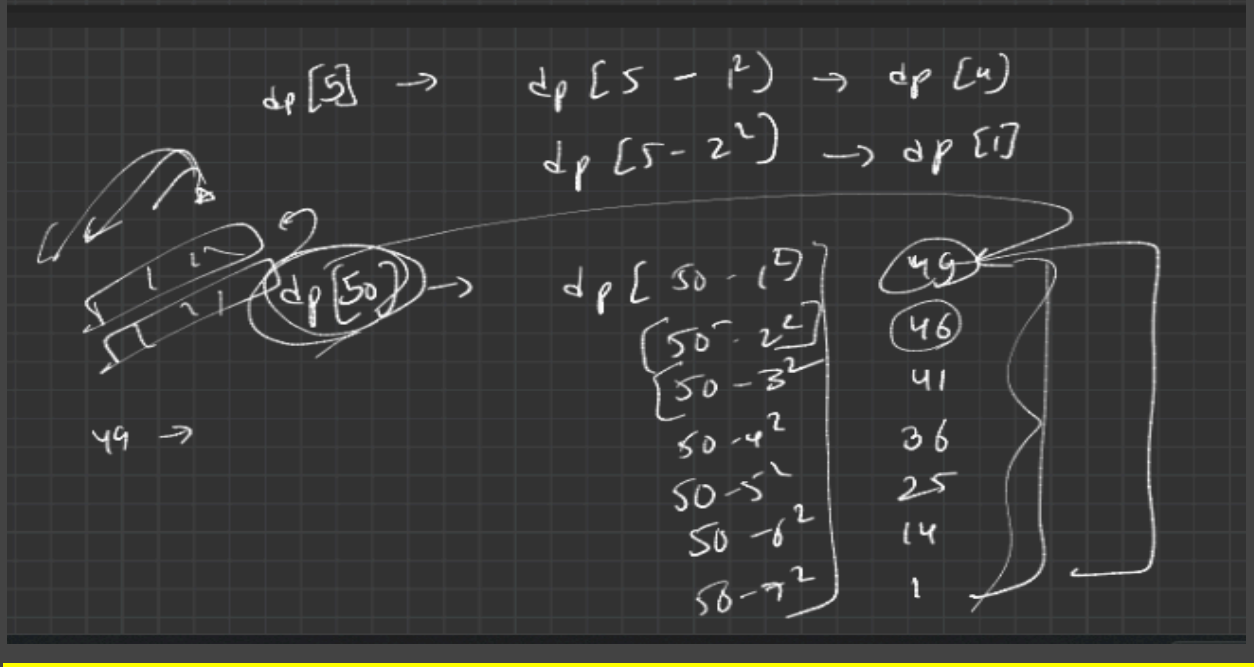
Get Minimum Squares By Recursion / Top-Down / Bottom-Up Approach:



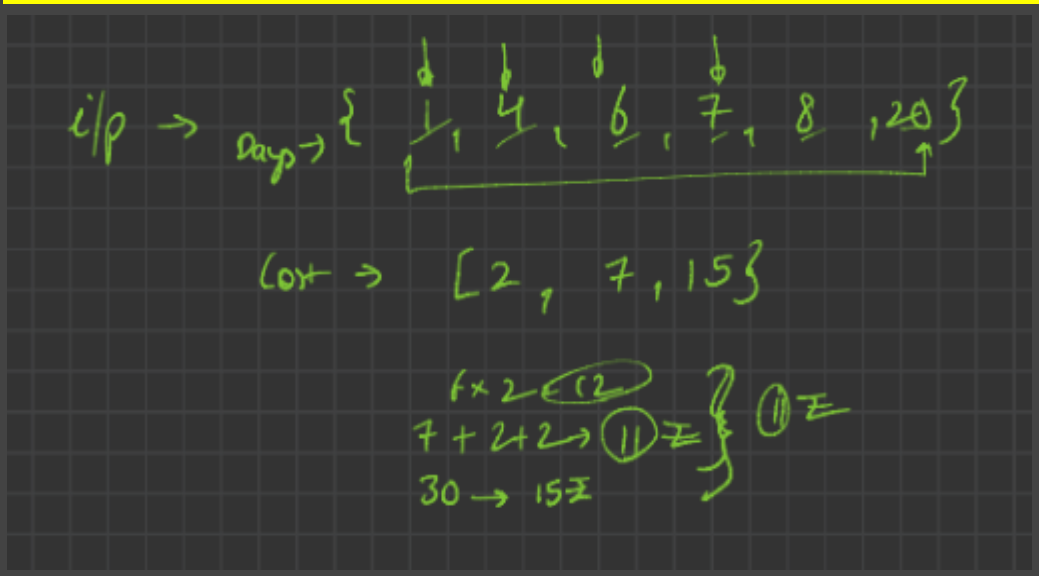


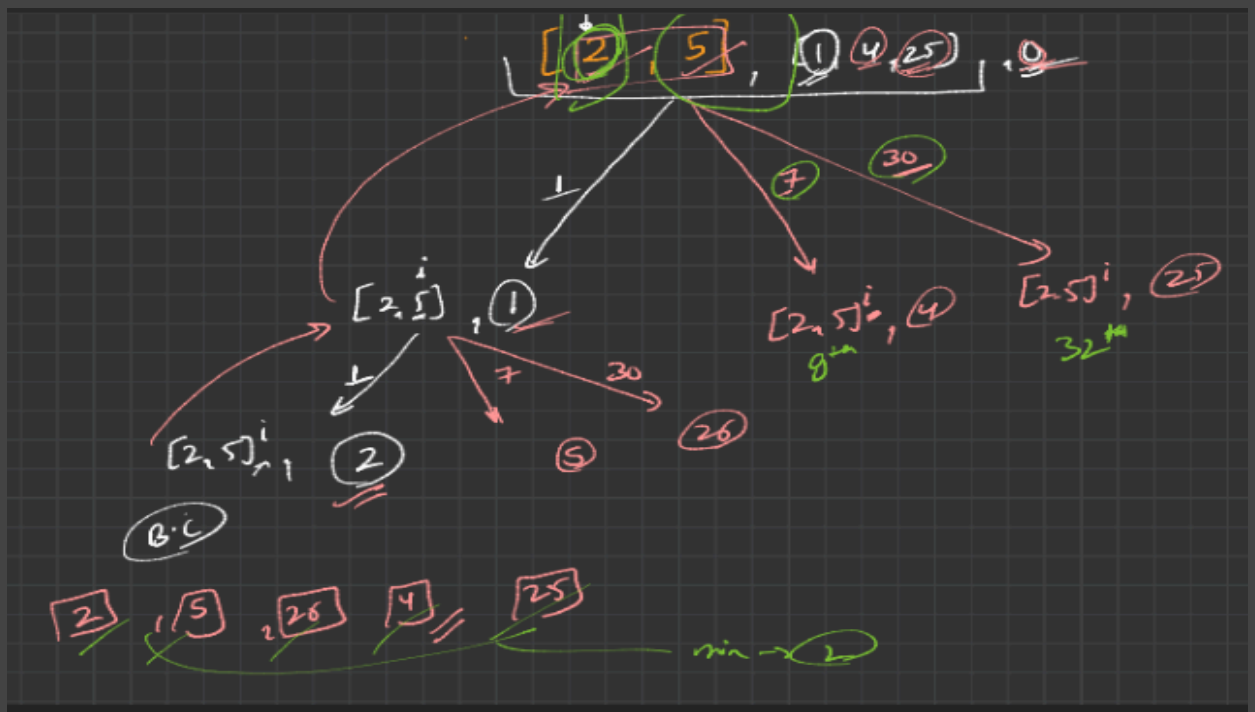


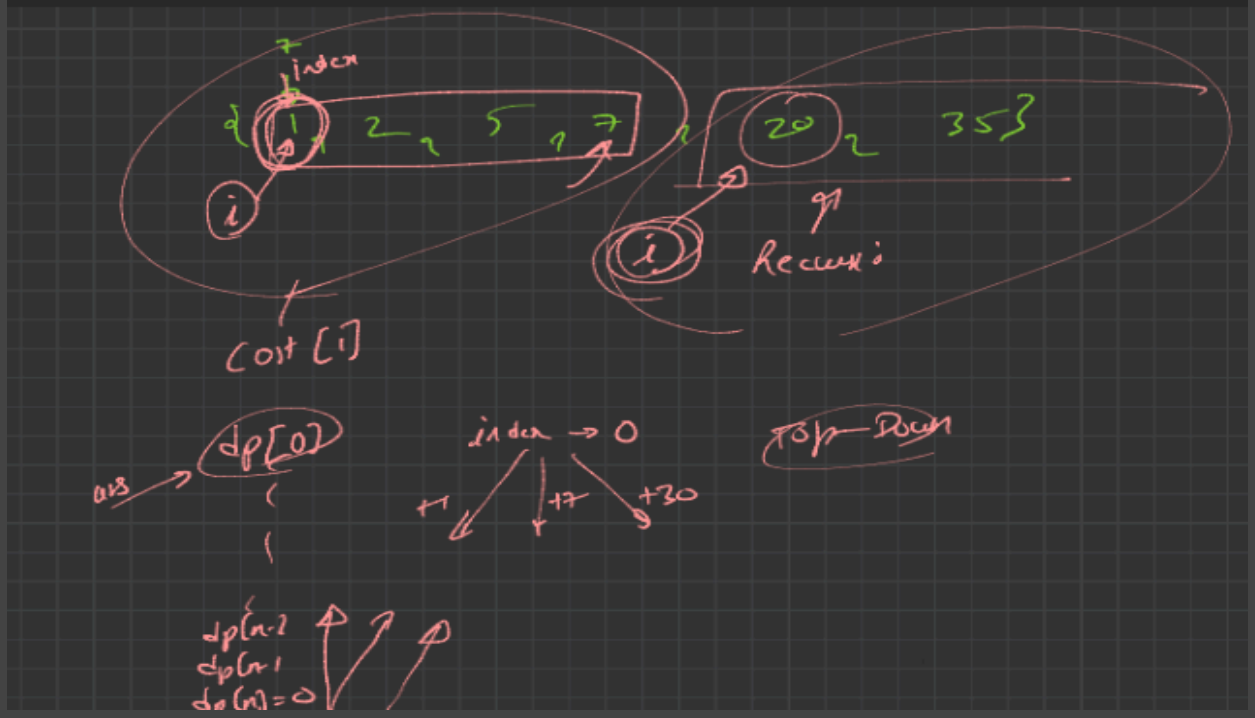


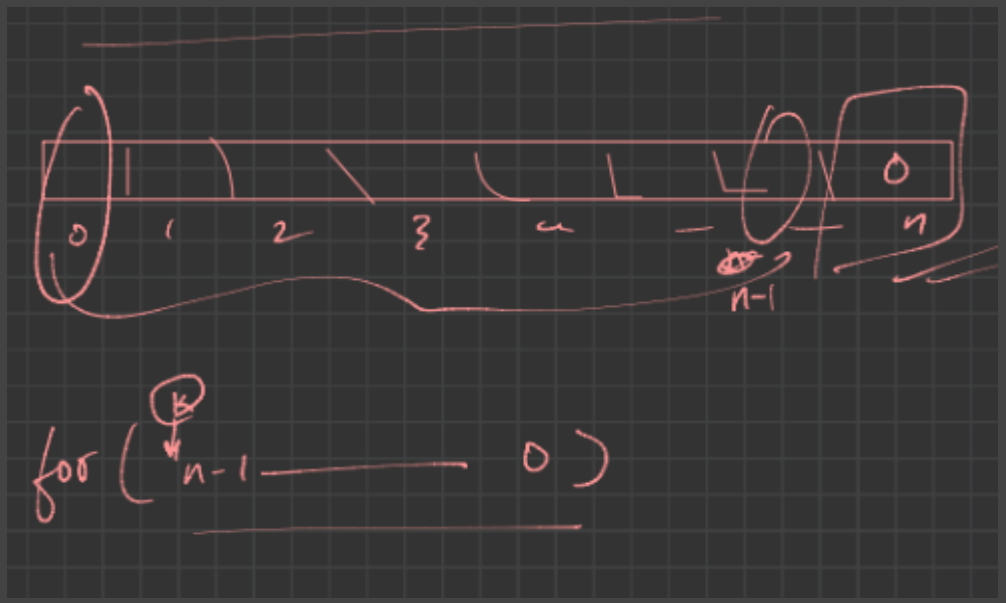


Minimum Cost For Tickets By Recursion / Top-Down / Bottom-Up Approach:




[2, 5, 1, 4, 25, 0]
1
7
30
[2,5]i, 1
[2,5]i, 4
[2,5]i, 25
[2,5]i, 1
2
5
26
B.C
2
5
26
4
25
min → 2


index
1
2
5
7
i
20
35
i
Recursion
Cost [i]
dp[0]
0th
index → 0
+1
+7
+30
Top-Down
dp[n-1]
dp[n-1]
dp[n]=0


0
1
2
3
n
n-1
for (n-1 ——— 0)

6:0

Index → 365

dp[365]

Days arr

365

[1 ——— 365]

dp[↖

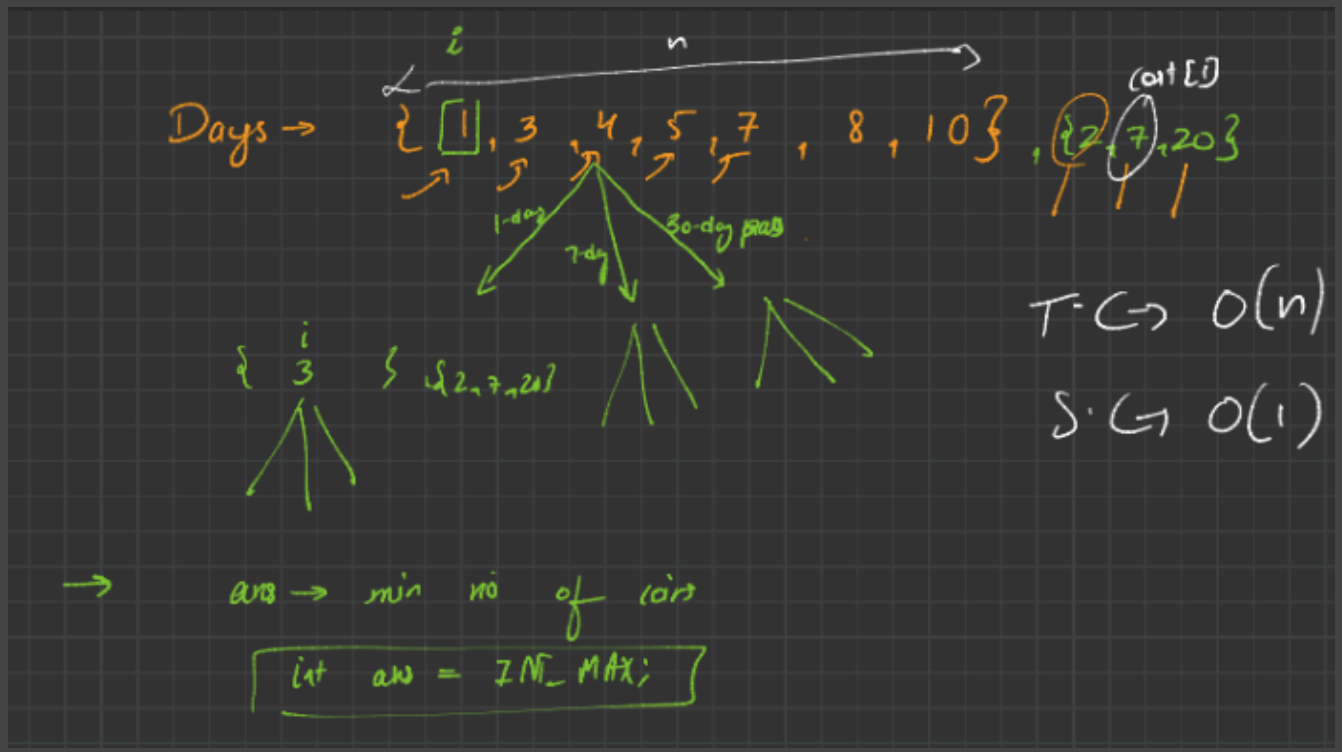
O(1)

#4 → Space Optimisation]→ next Episode (20 min)

→ Bottom - Up Approach :

T.C → O(n)

S.C → O(n) → O(1)



T.C → O(n)

S.C → O(1)

→ ans → min no of coins

int ans = INT_MAX;

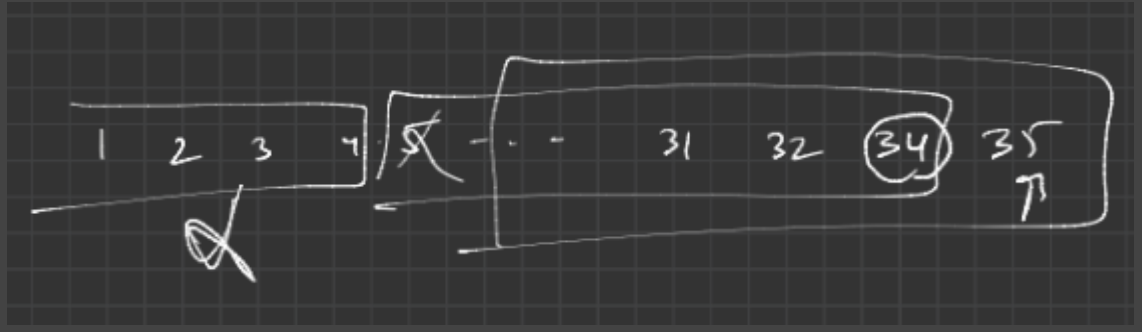
ans + cost[0]

queue < pair < int (day), int (cost till that day) > > monthly; → 30 entries

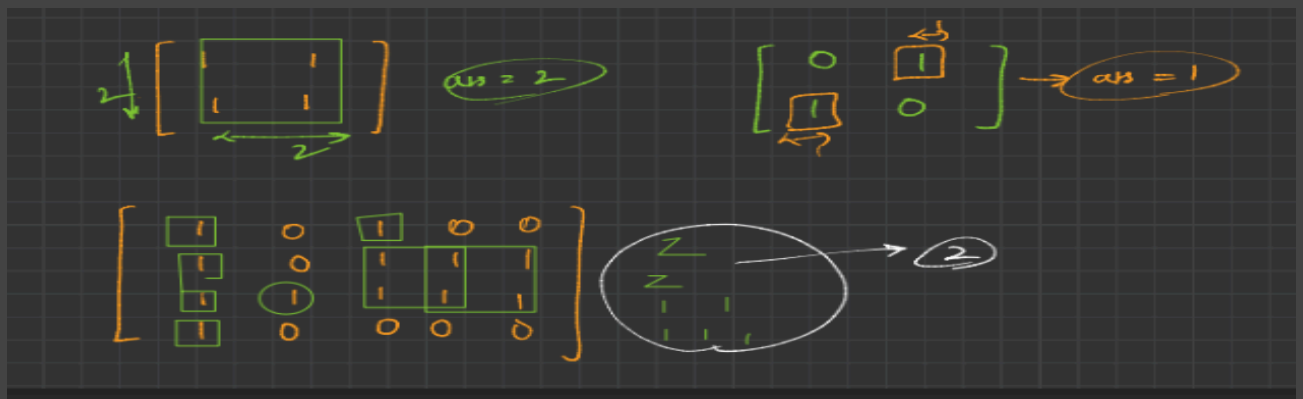
weekly; → 7 entries

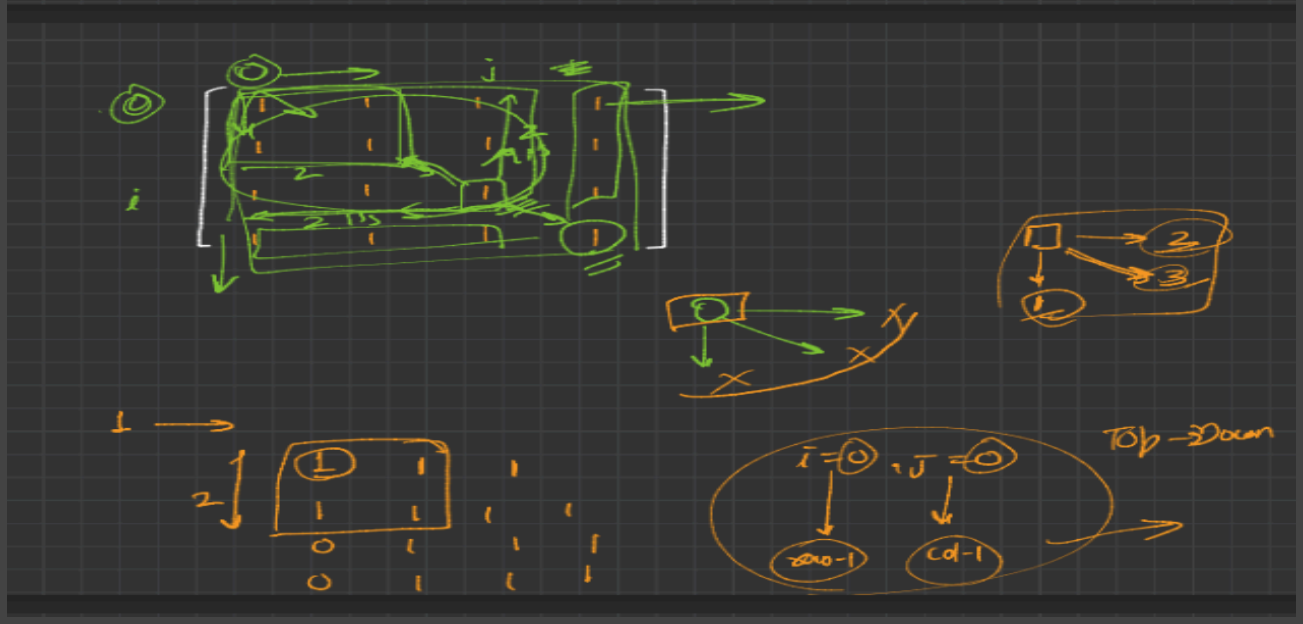
S.C → O(1)

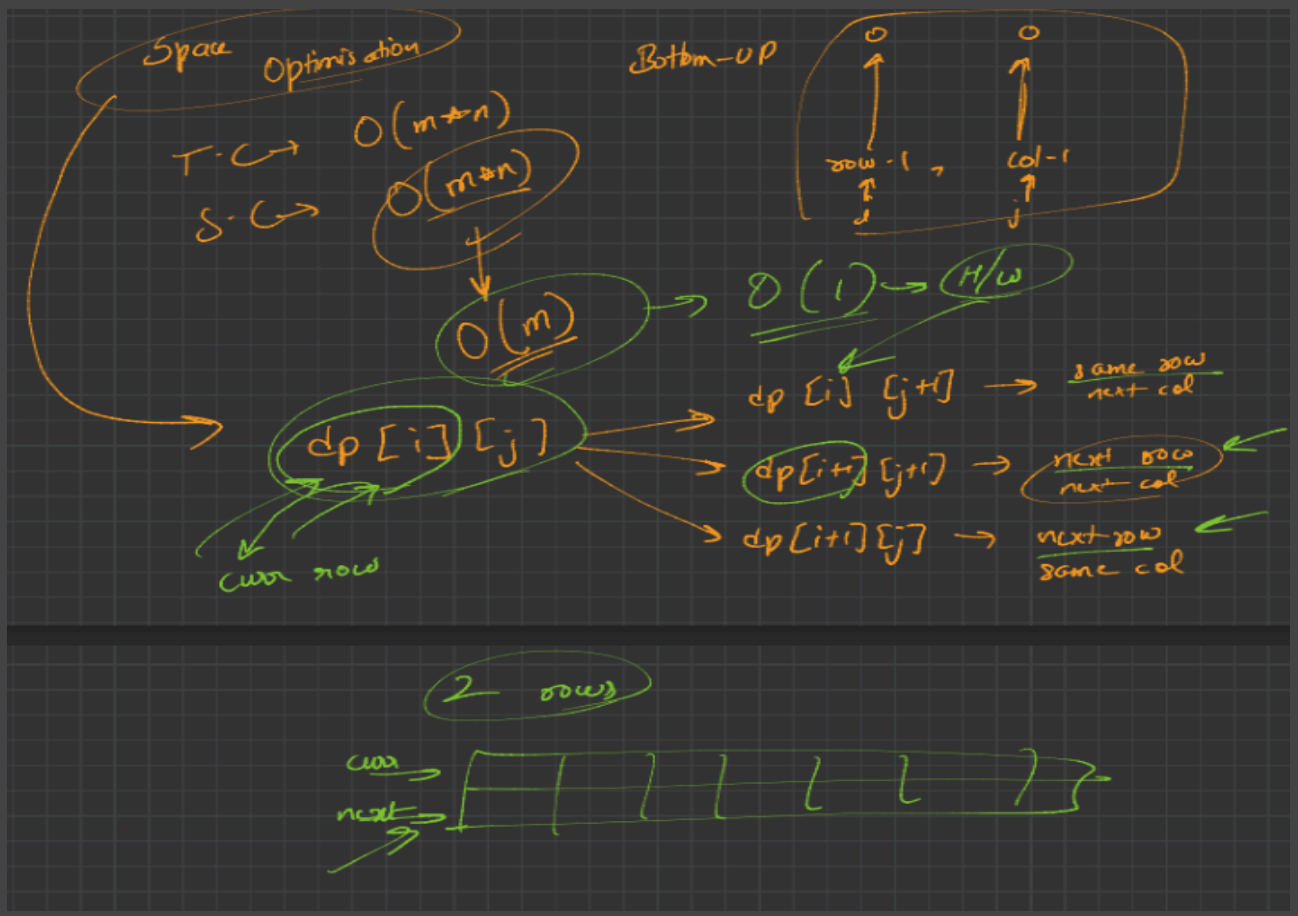
```
int ans = INT_MAX;

for ( day ; days)
{
    // 1 step → remove expired days from queue
    while (monthly.size() && monthly.front().first + 30 <= day)
        monthly.pop()
    while (weekly.size() && weekly.front().first + 7 <= day)
        weekly.pop()

    // Step2 → push current day's cost
    weekly.push (make_pair (day, ans + cost[1]);
    monthly.push (make_pair (day, ans + cost[2]);

    // Step 3 → ans update
    ans = min ( ans + cost[0] , min ( monthly.front().second,
                                      weekly.
    }
    return ans;
}
```

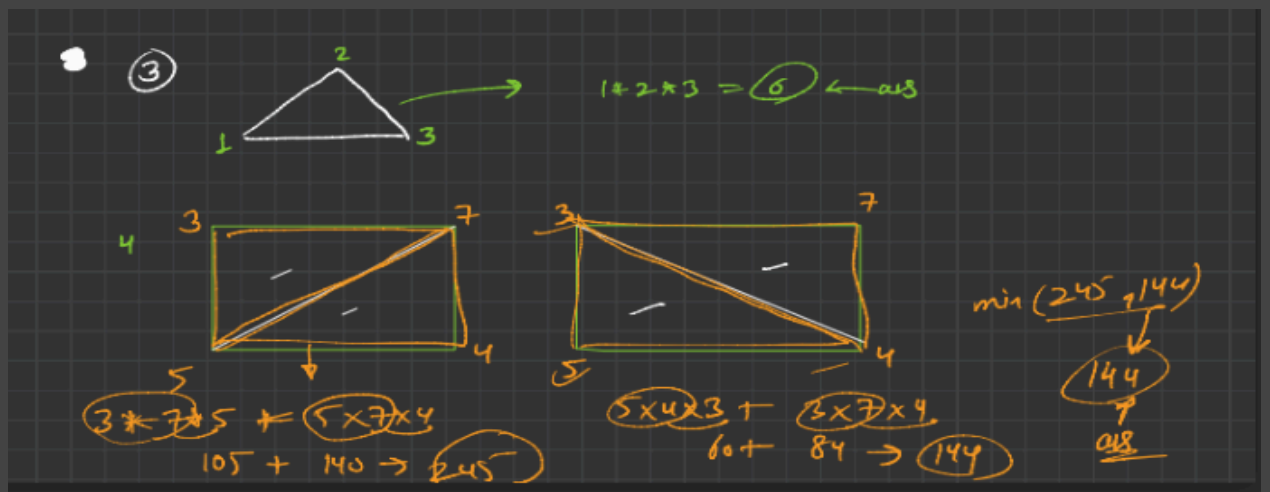
Largest Square Area In Matrix By Recursion / Top-Down Approach / Bottom-Up Approach / Space Optimization Approach:

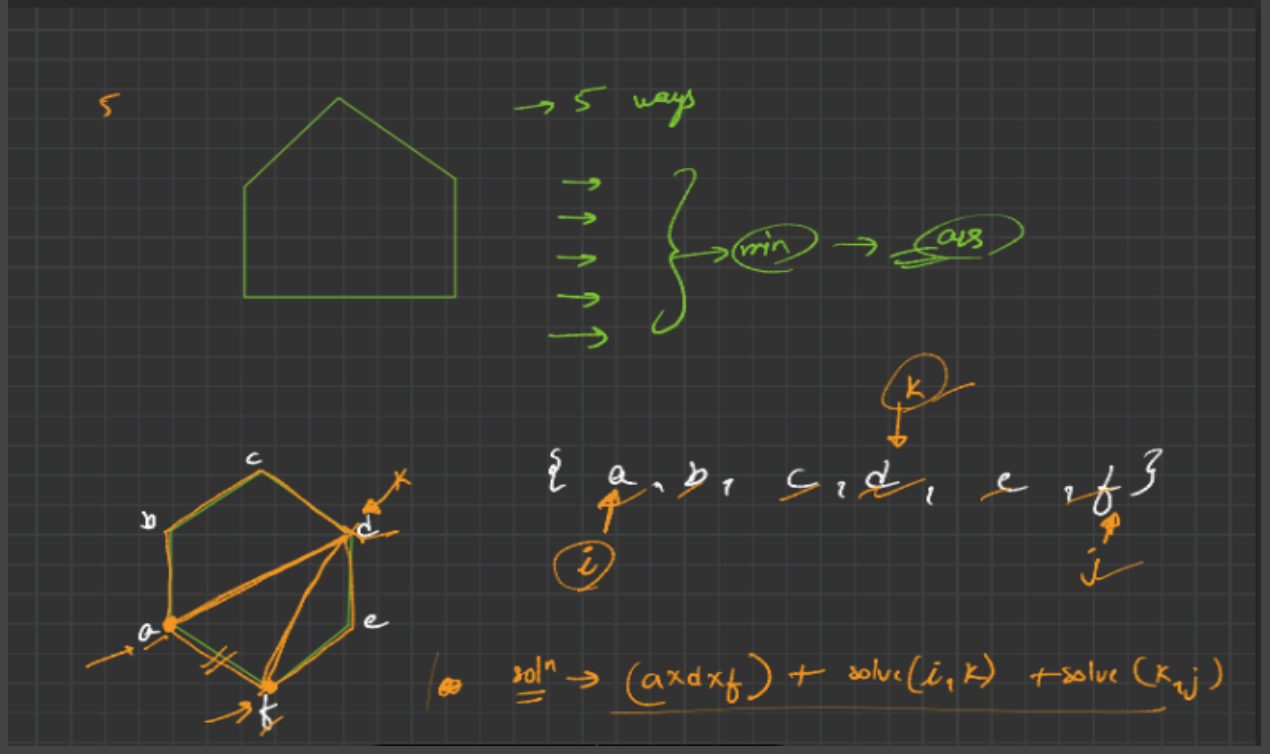


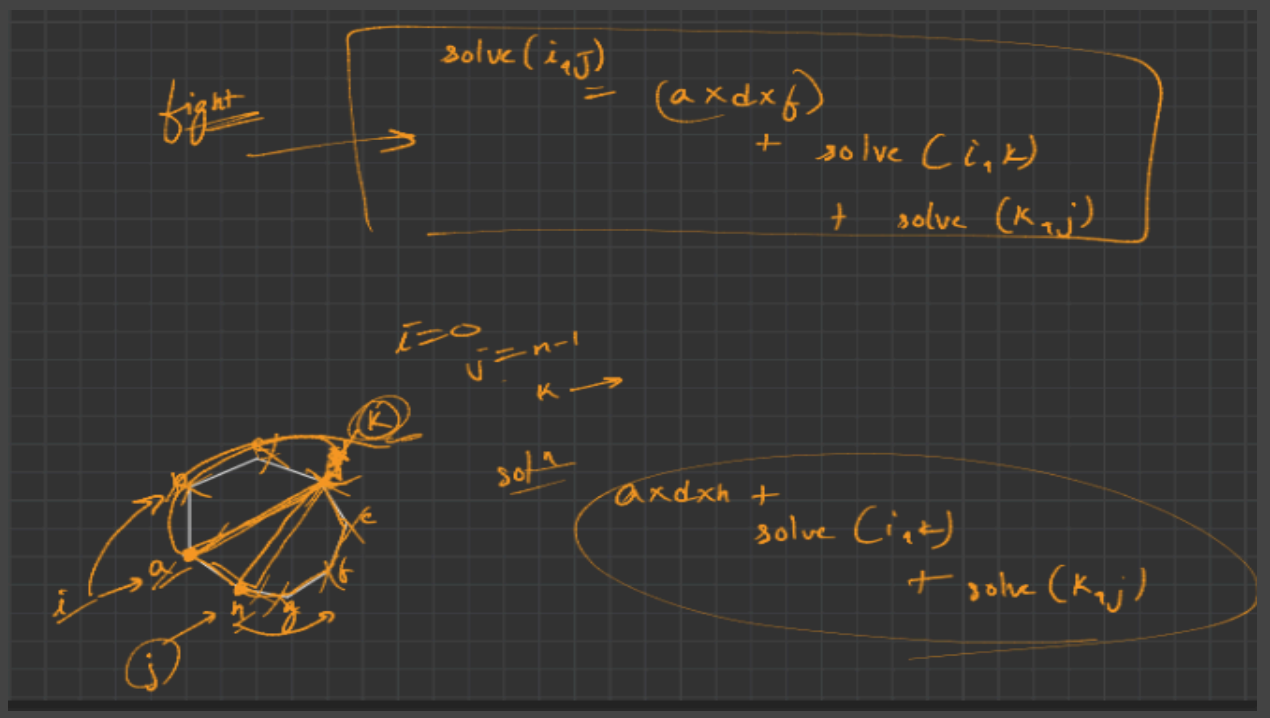


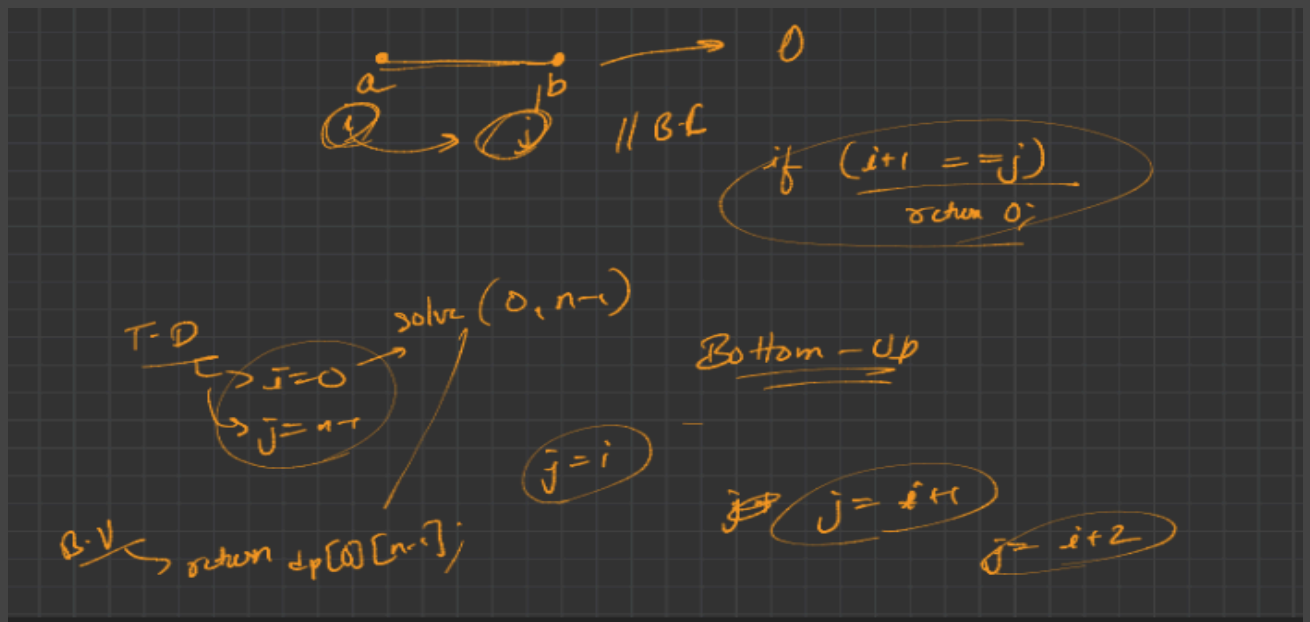


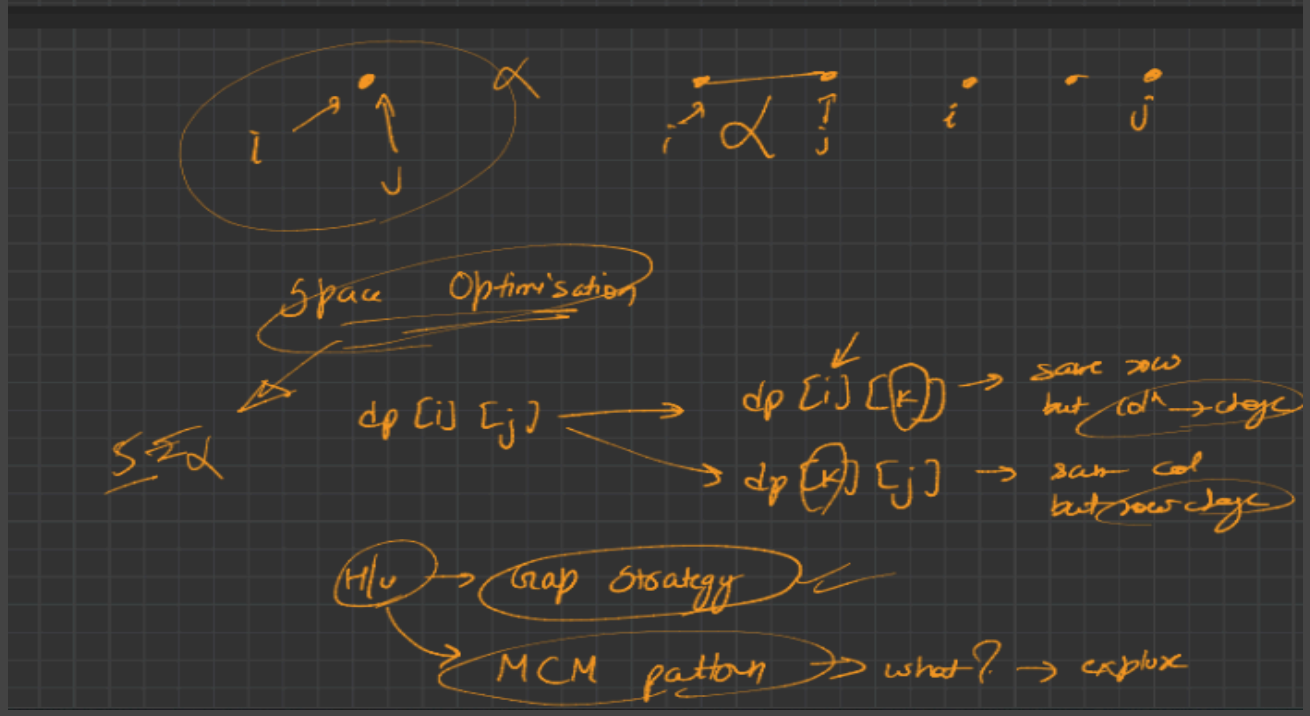
Minimum Score Triangulation By Recursion / Top-Down / Bottom-Up Approach:

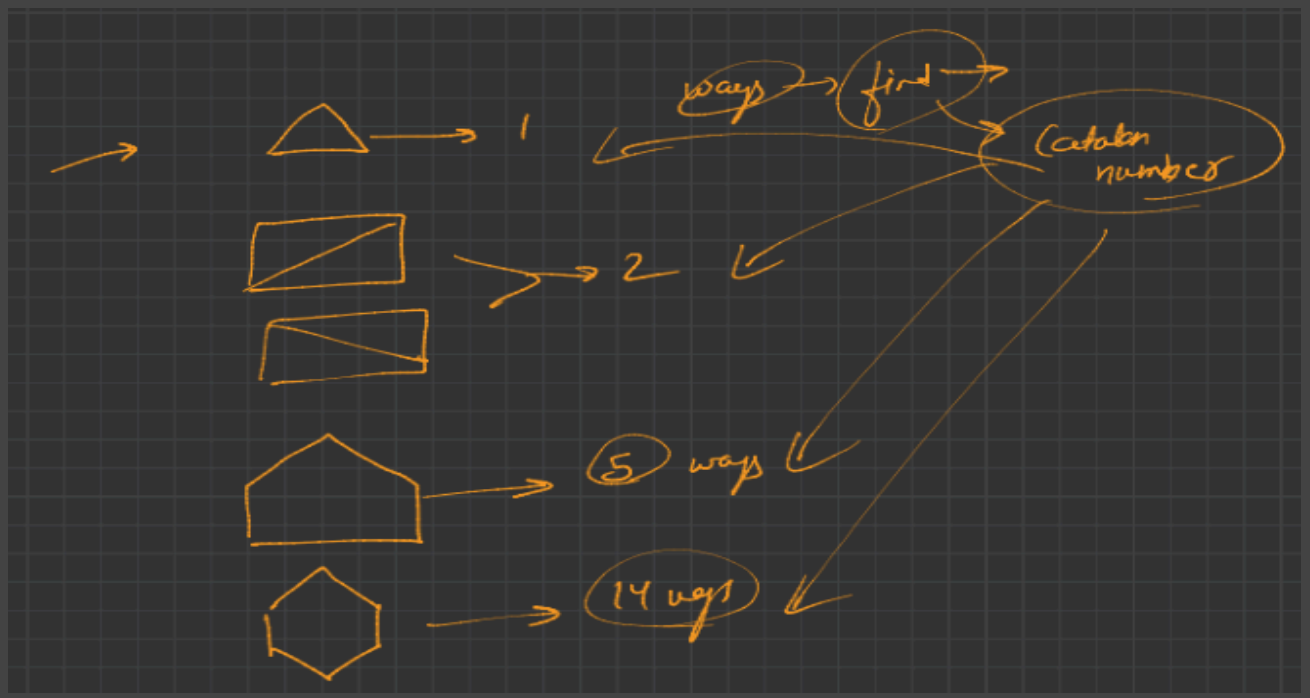




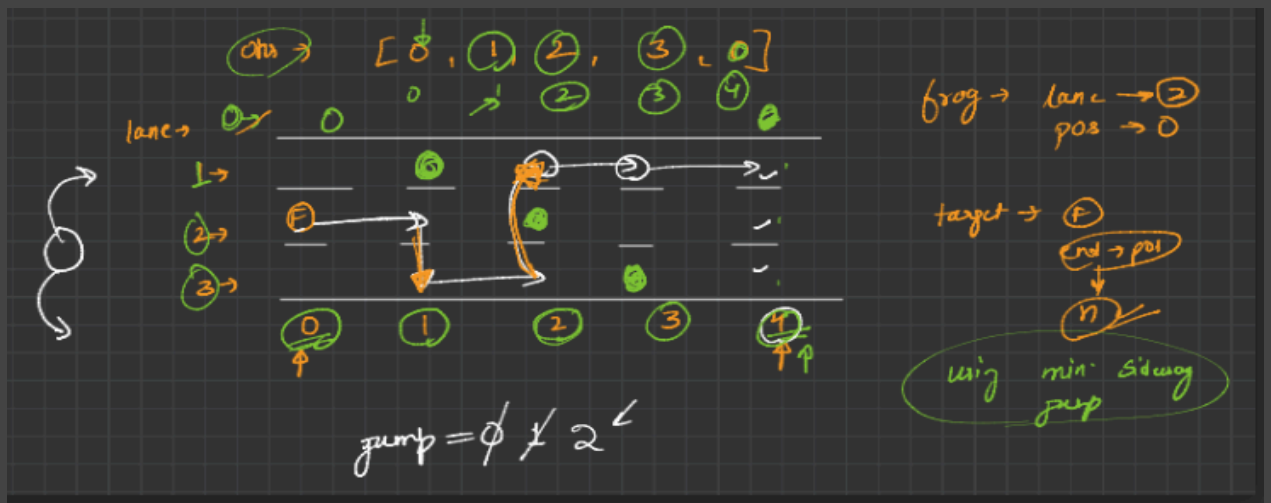


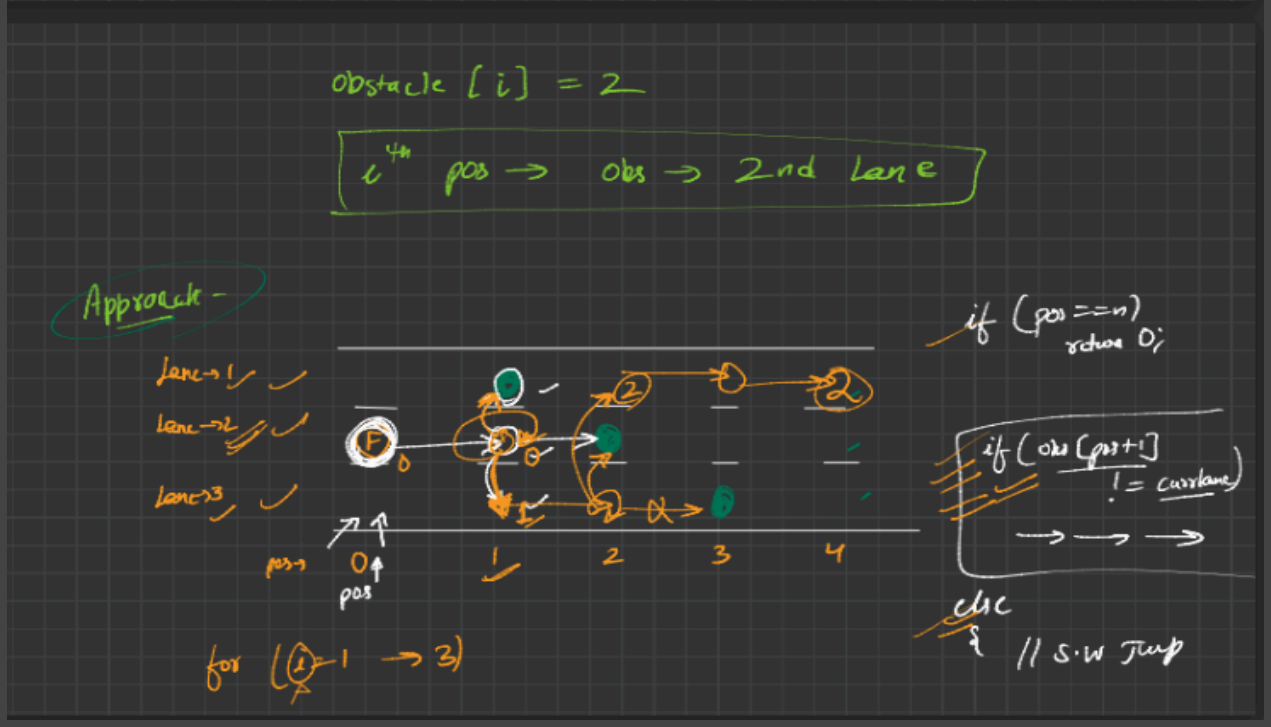

a
b
0
// B-L
if (i+1 == j)
return 0;
T-D
solve (0, n-1)
j=0
j=n-1
B.V
return dp[0][n-1];
Bottom-Up
j=i
j=i+1
j=i+2

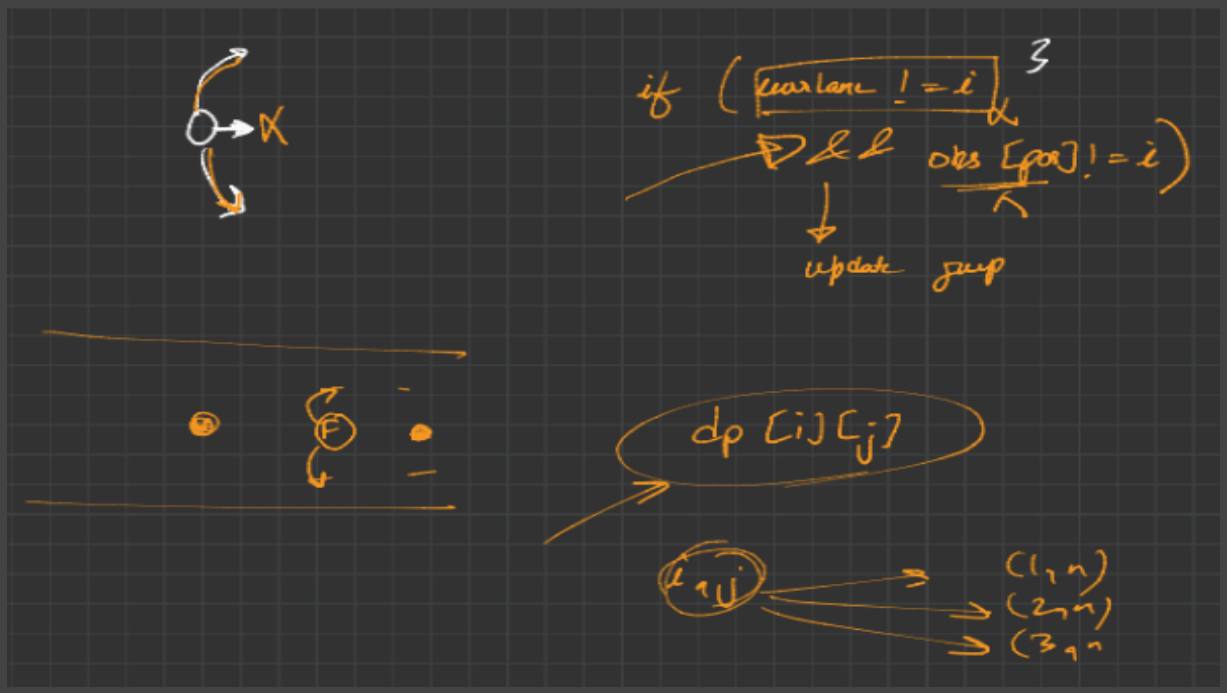

i
j
Space Optimisation
dp[i][j]
dp[i][k]
dp[k][j]
save row but col → days
save col but row days
Hu
Gap Strategy
MCM pattern
what?
complex

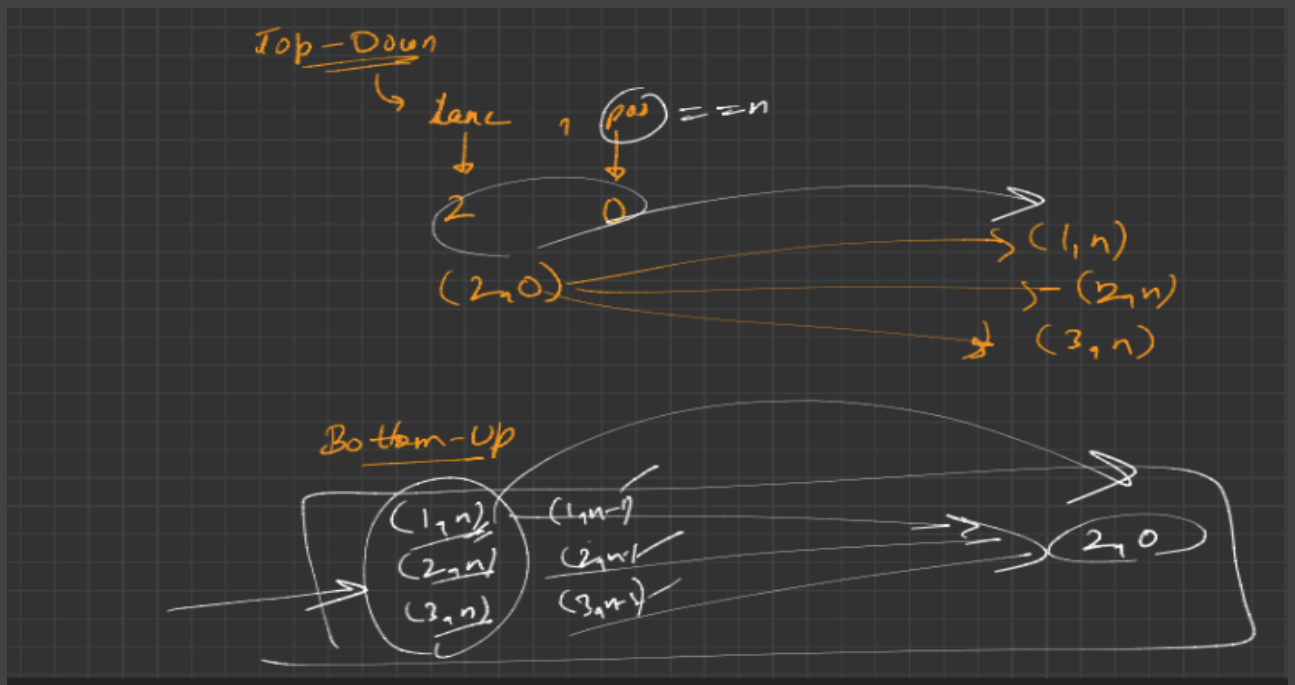

ways
find
Catalan numbers
1
2
5 ways
14 ways

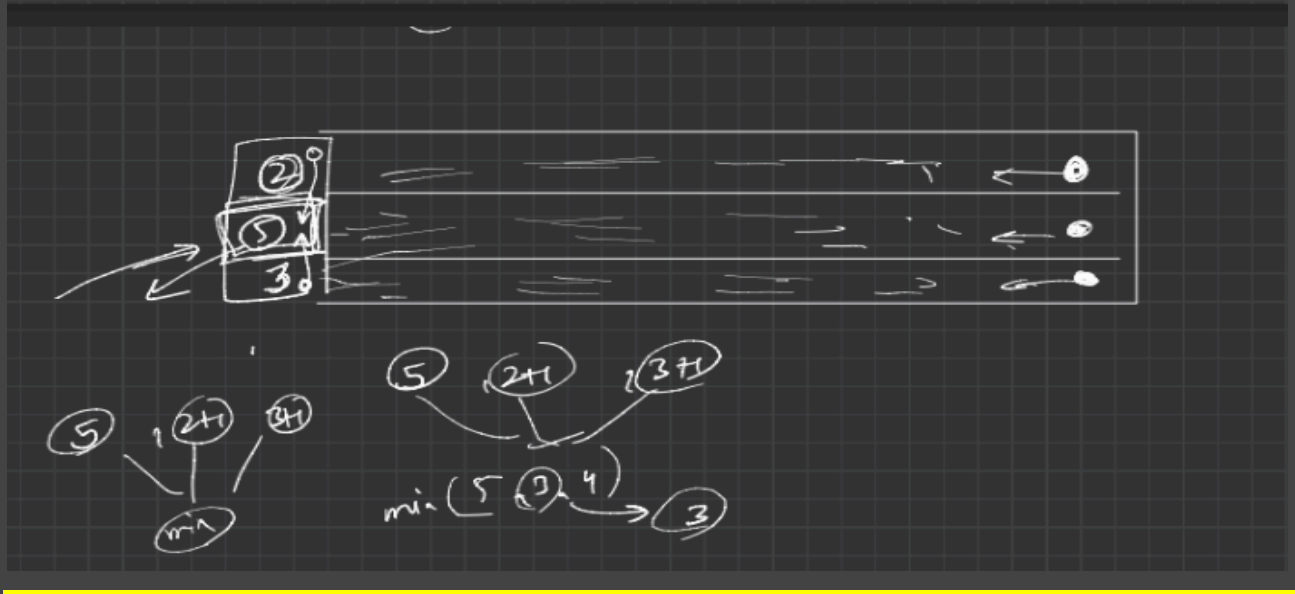
Minimum Sideways Jumps By Recursion / Top-Down Approach / Bottom-Up Approach / Space Optimization Approach:



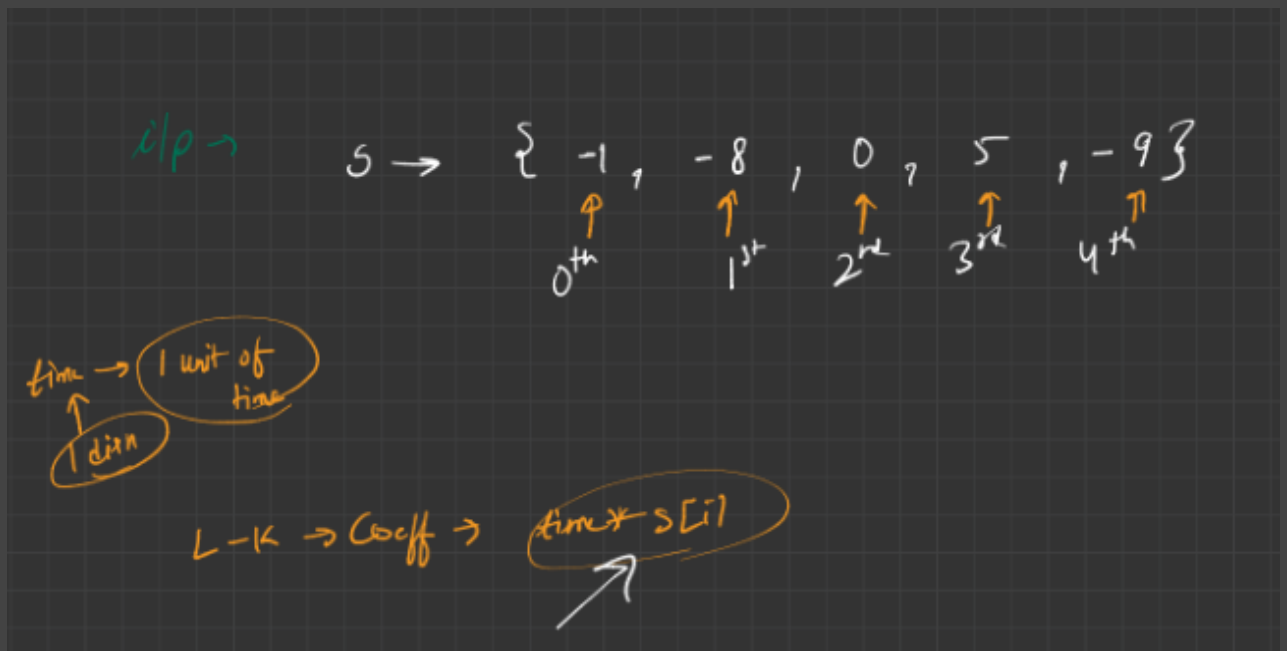








Reducing Dishes By Recursion / Top-Down Approach / Bottom-Up Approach / Space Optimization Approach:



[-1 , 0 , 5]

neglect include

[-1 , 0 , 5]

-1 x 1 + 0 x 2 + 5 x 3

-1 + 0 + 15 → 14

[4 , 3 , 2]

sort → [2 , 3 , 4]

2x1 + 3x2 + 4x3 = 20

[ ]

→ 0

discard include

a , b , c , d

x1 x2 x3 x4

0/1 Knapsack

time time time time

arr → [4 , 0 , -1]

sort → [-1 , 0 , 4]

Rec

arr = (-1 x 1)

= -1

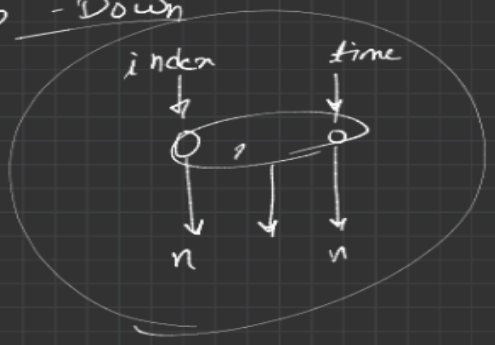
arr = -1

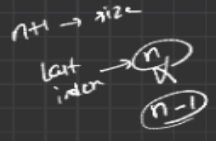
0 x 2

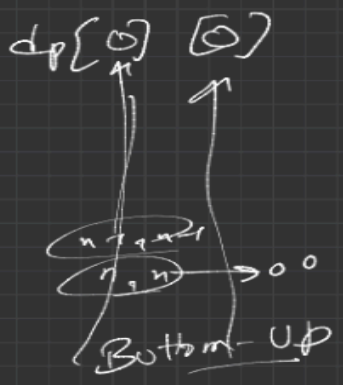
4 x 3

BC

Top - Down

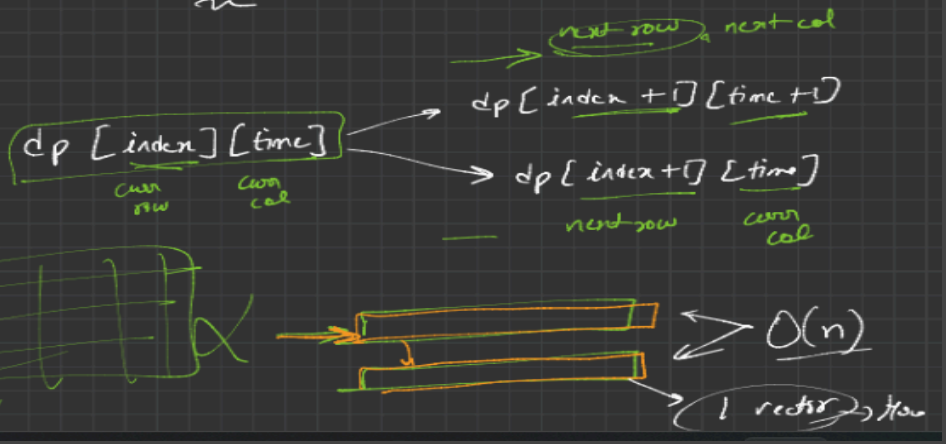


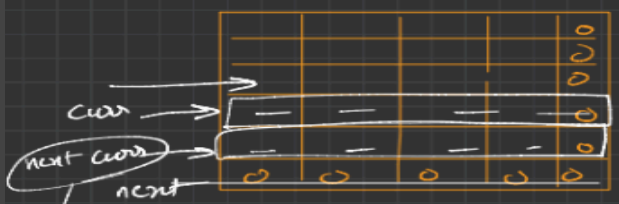




Bottom - Up

n^2 → ? S.O → ?





$O(n^2) \rightarrow O(n)$

0/1 Knap DP

Rec → TLE

T-C

T-D → $O(n^2) \rightarrow O(n^2)$

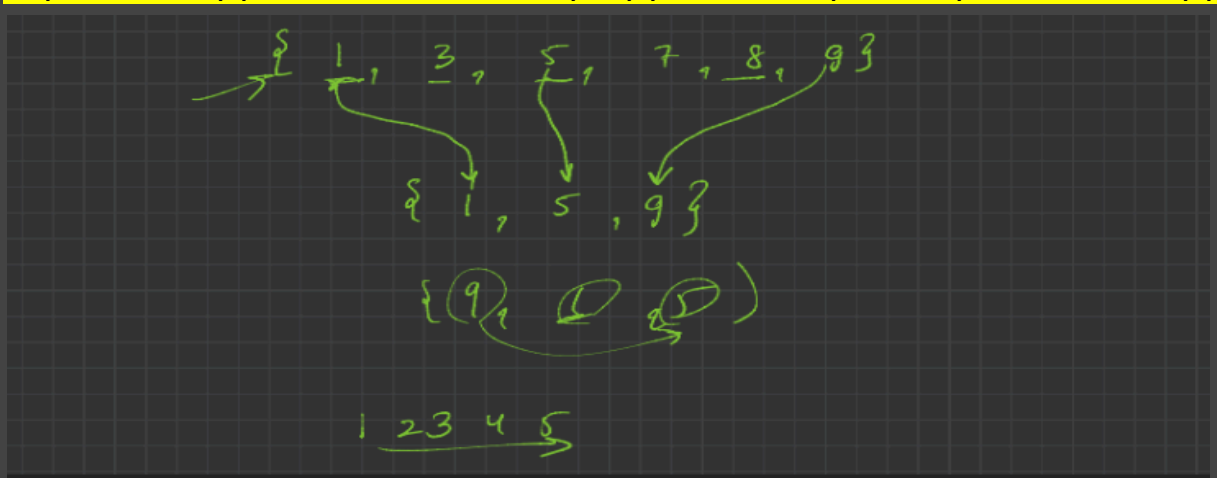
BU → $O(n^2) \rightarrow O(n^2)$

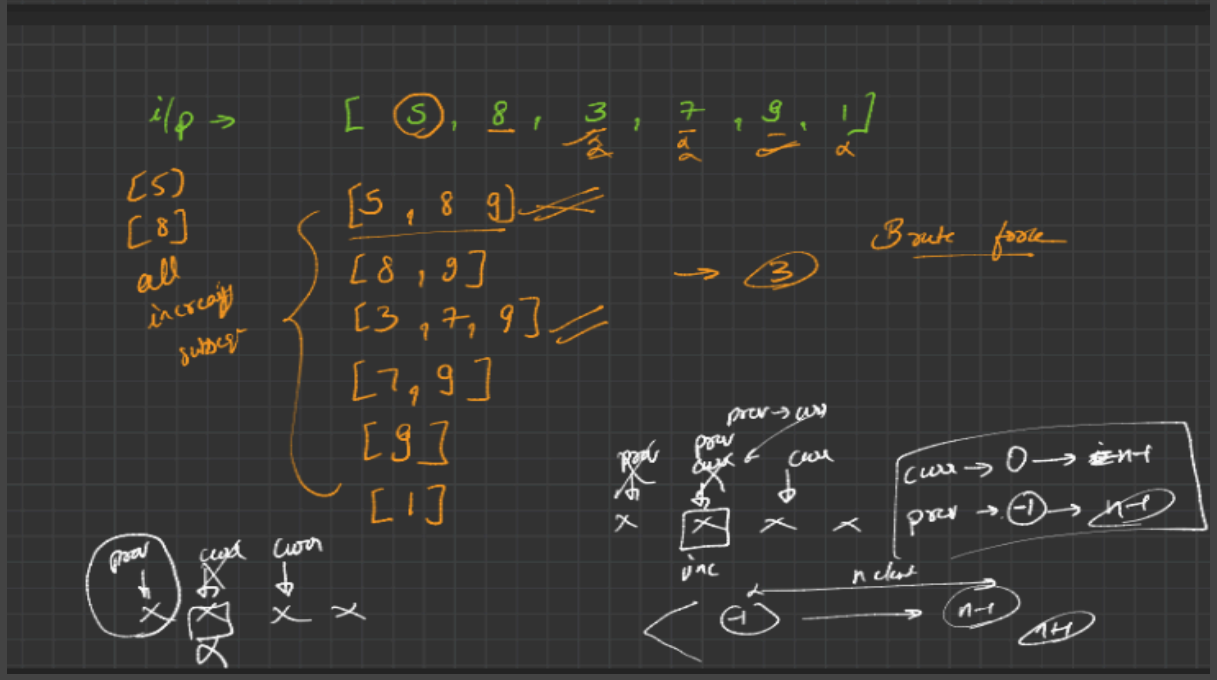
S.O → $O(n)$

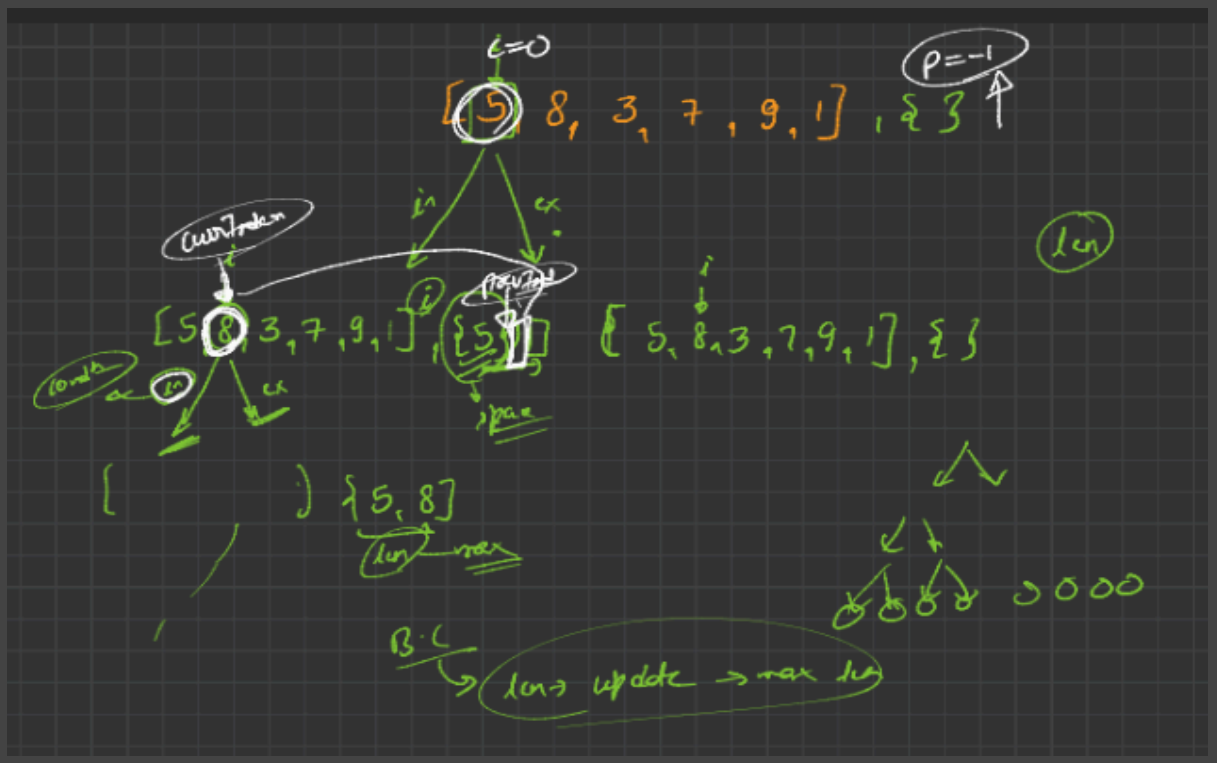
$O(1) \rightarrow$?

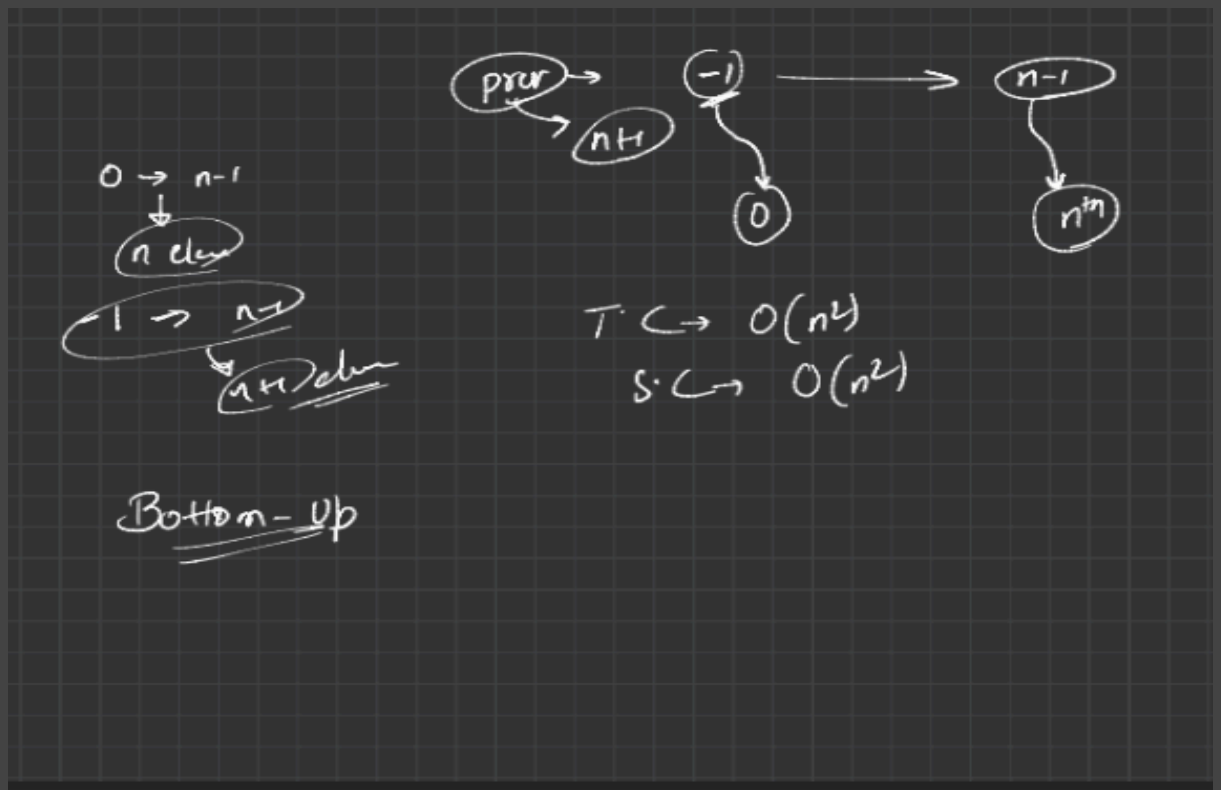
Possible → Greedy

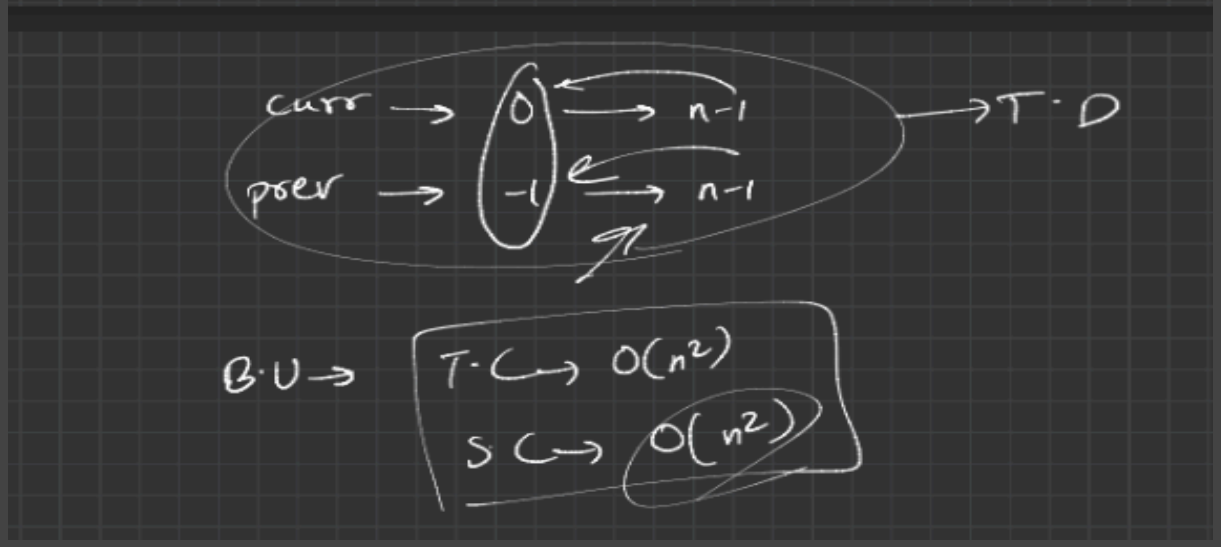
Longest Increasing Subsequence And Russian Doll Envelopes By Recursion / Top-Down Approach / Bottom-Up Approach / Space Optimization Approach:

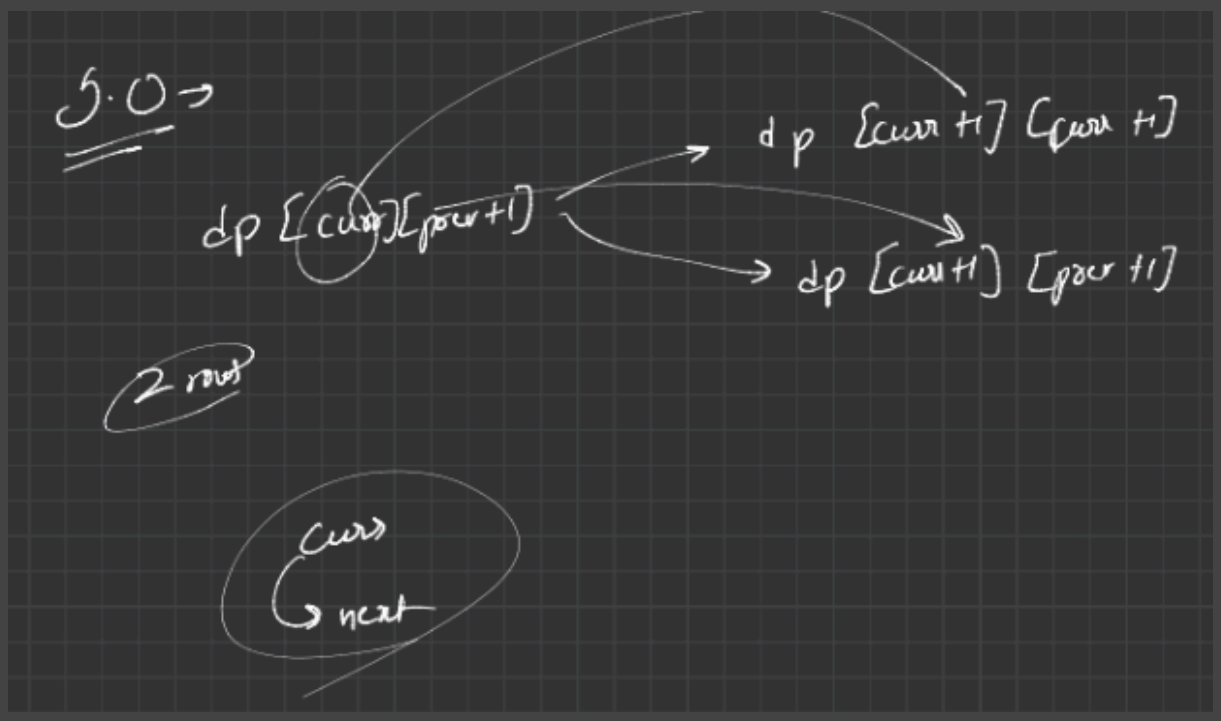


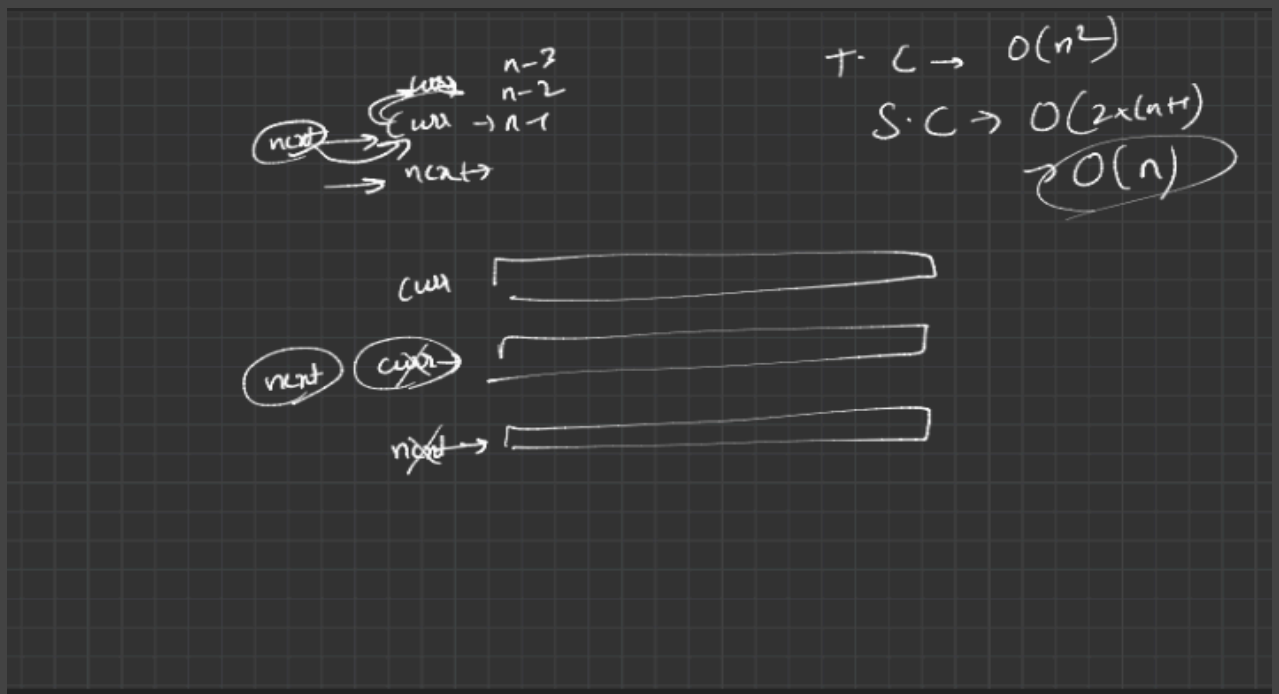


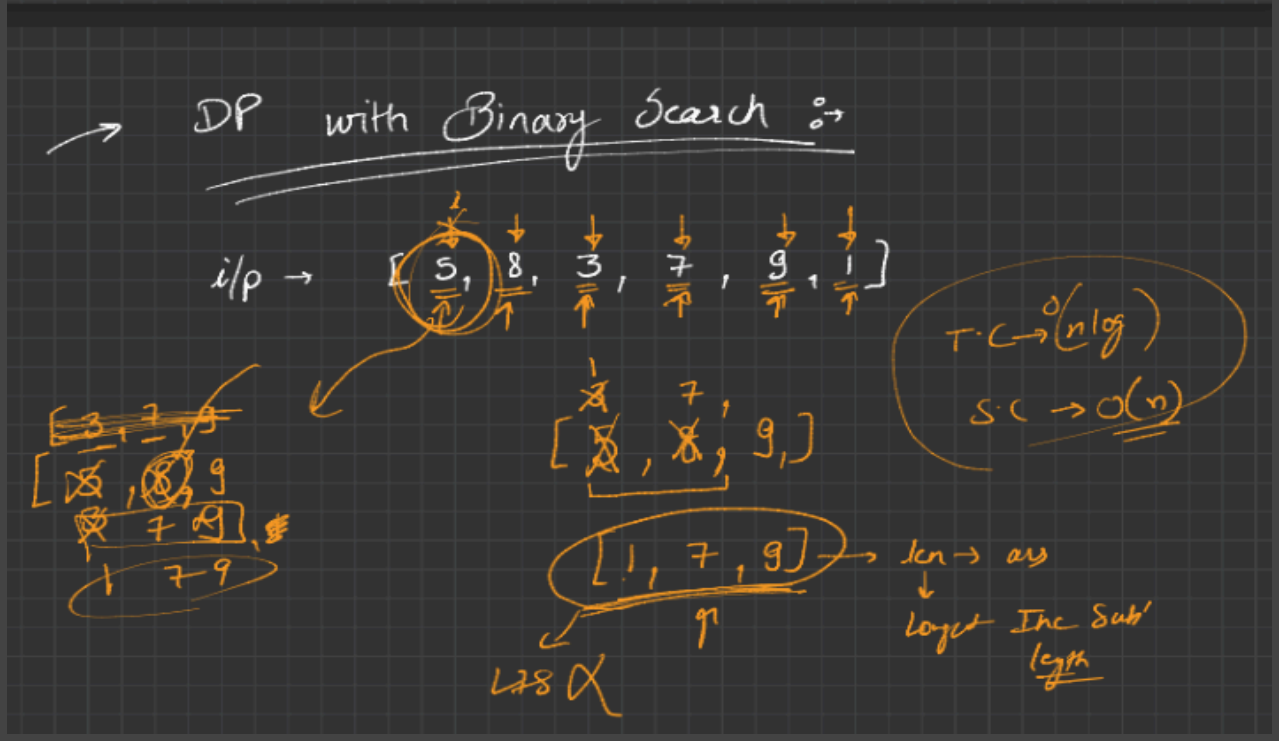



prev
-1
n-1
n+1
0
n-1
0 → n-1
n elem
1 → n-1
(n+1) elem
T.C → O(n²)
S.C → O(n²)
Bottom-Up

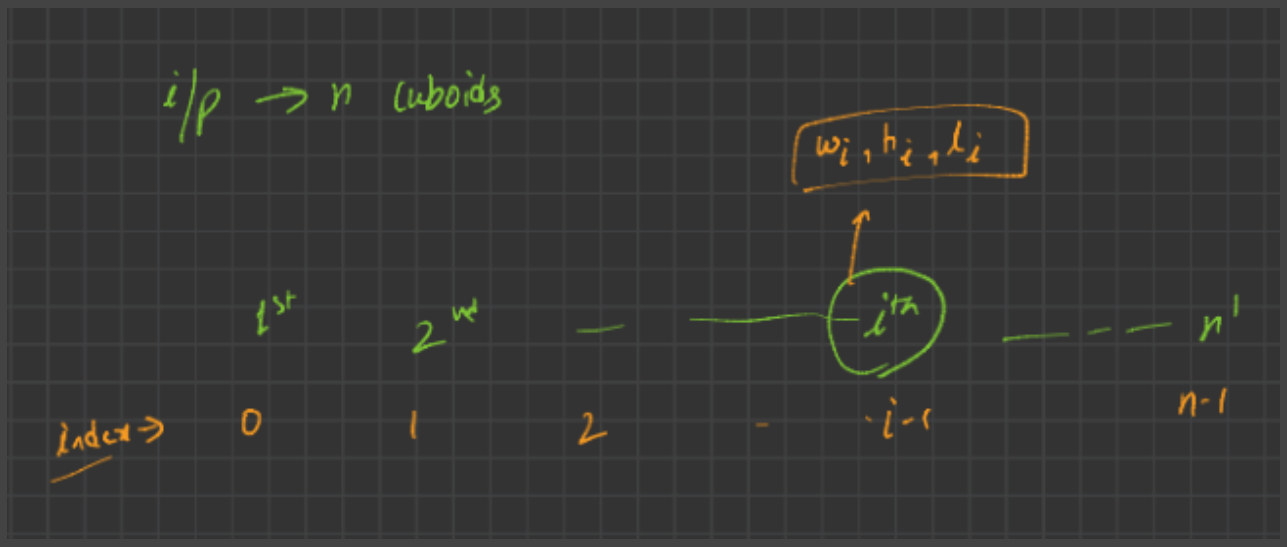

curr
prev
0
-1
n-1
n-1
T.D
B.U →
T.C → O(n²)
S.C → O(n²)

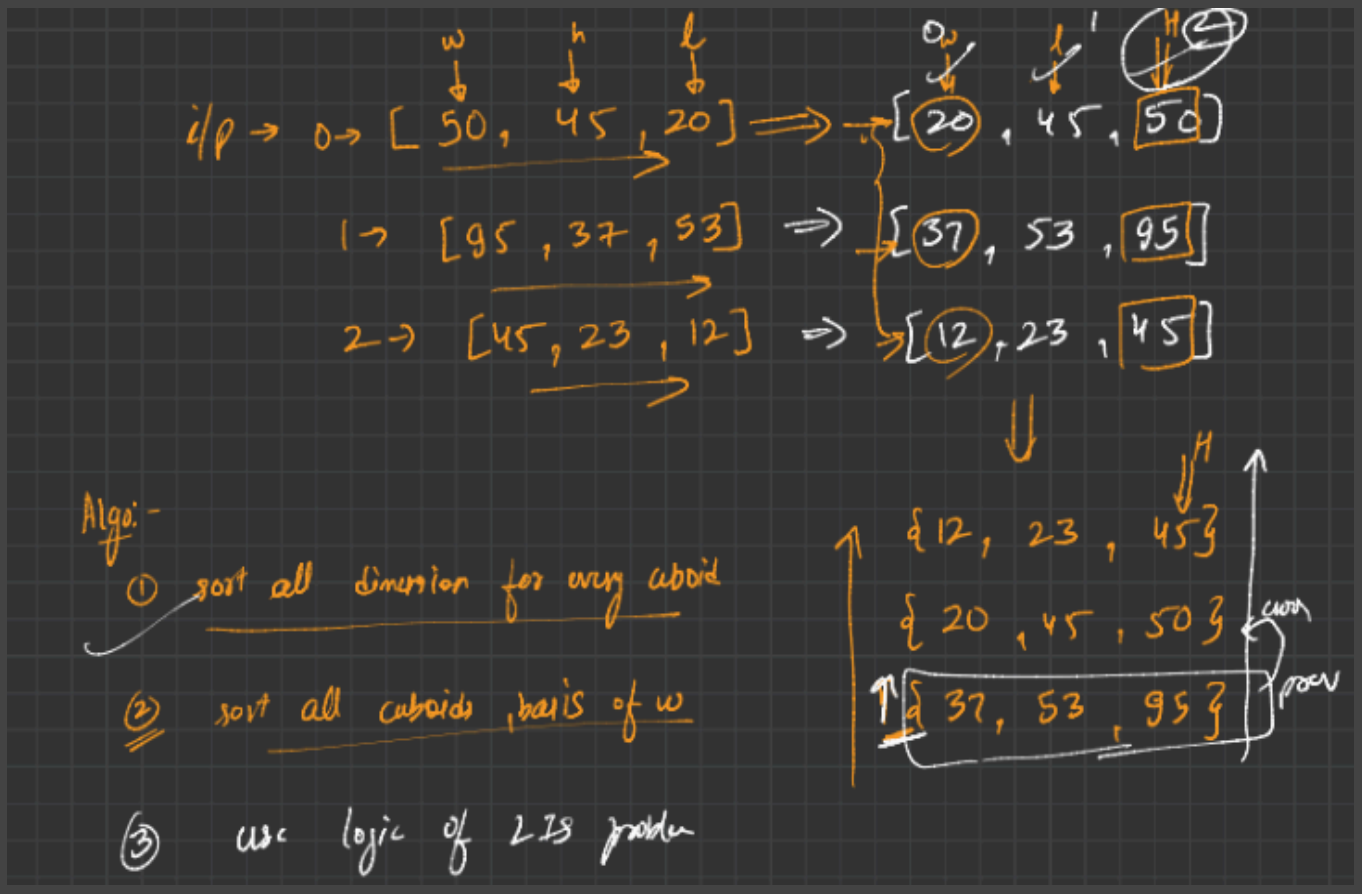

S.O →
dp [curr][prev+1]
dp [curr+1][curr+1]
dp [curr+1][prev+1]
2 rows
curr
next



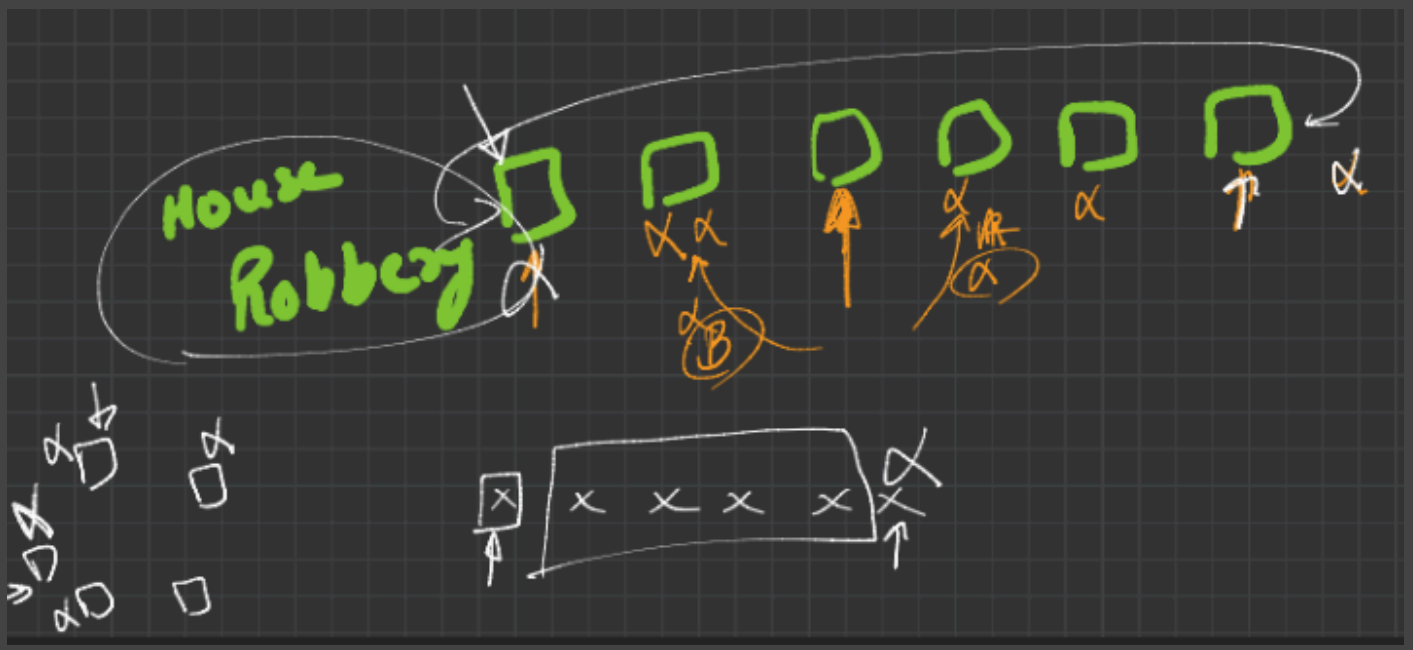


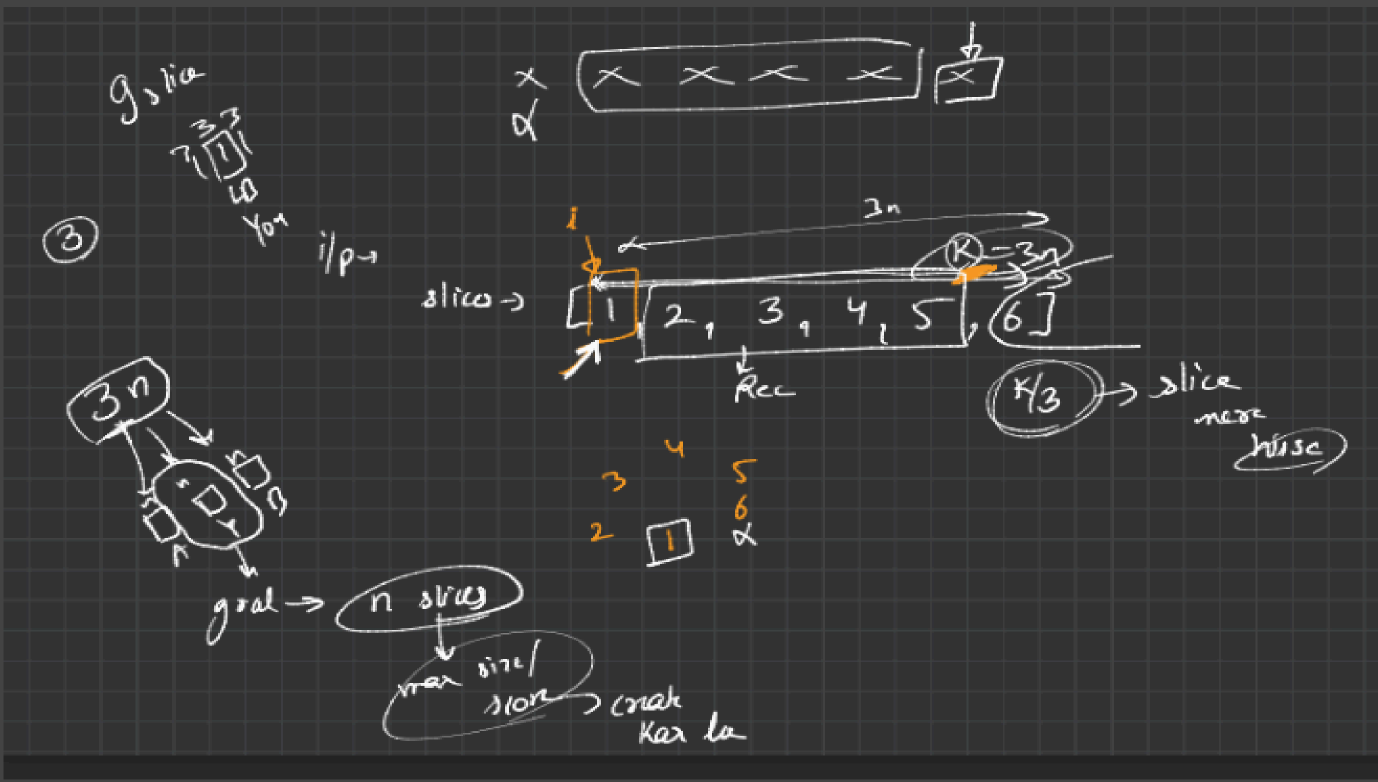
Maximum Height By Stacking Cuboids Space Optimization Approach:



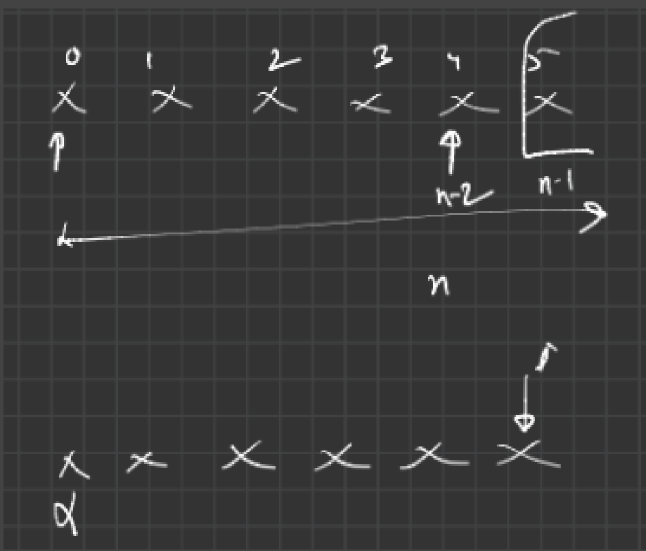


Pizza With 3n Slices Recursion / Top-Down Approach / Bottom-Up Approach / Space Optimization Approach:

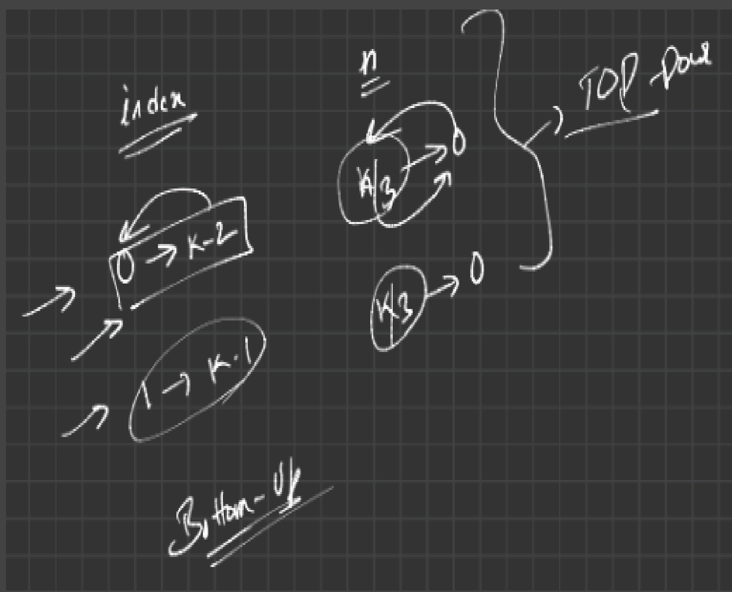




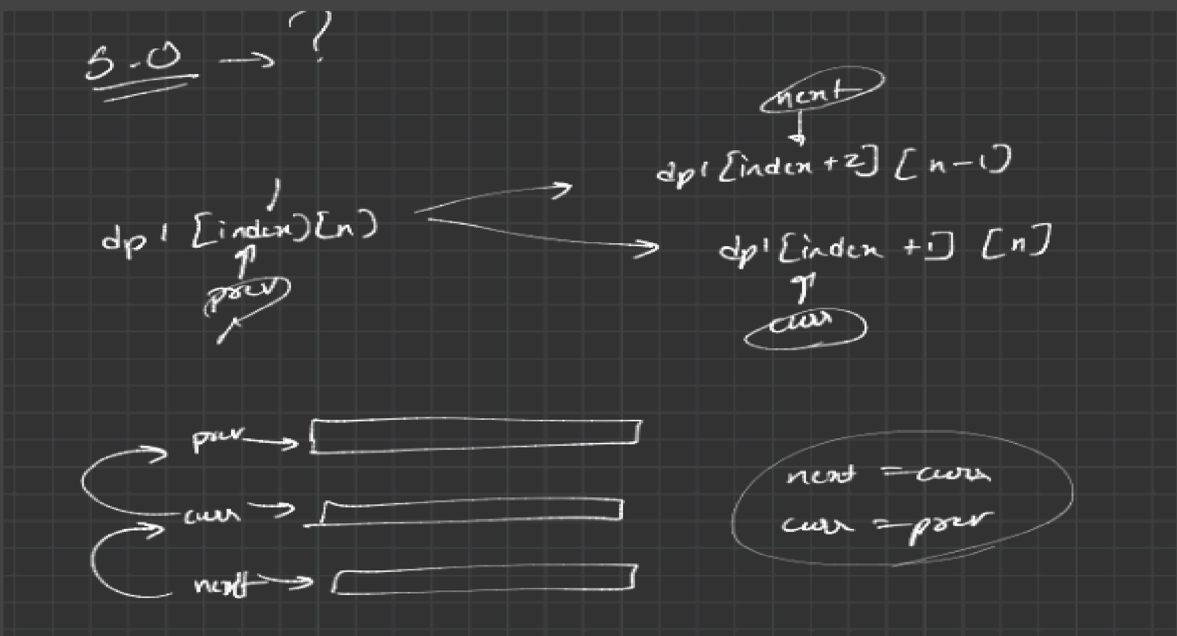
3
slice →
i/p →
3n
k = 3n
Rec
K/3 → slice
goal → n slices
max size/slice



0 1 2 3 4 5
n-2 n-1
n

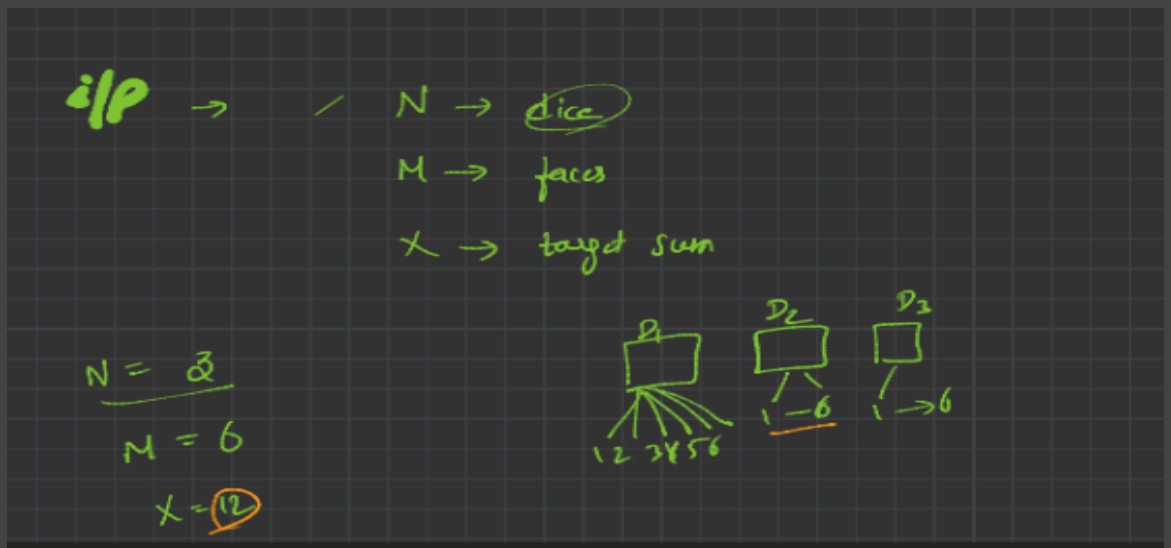


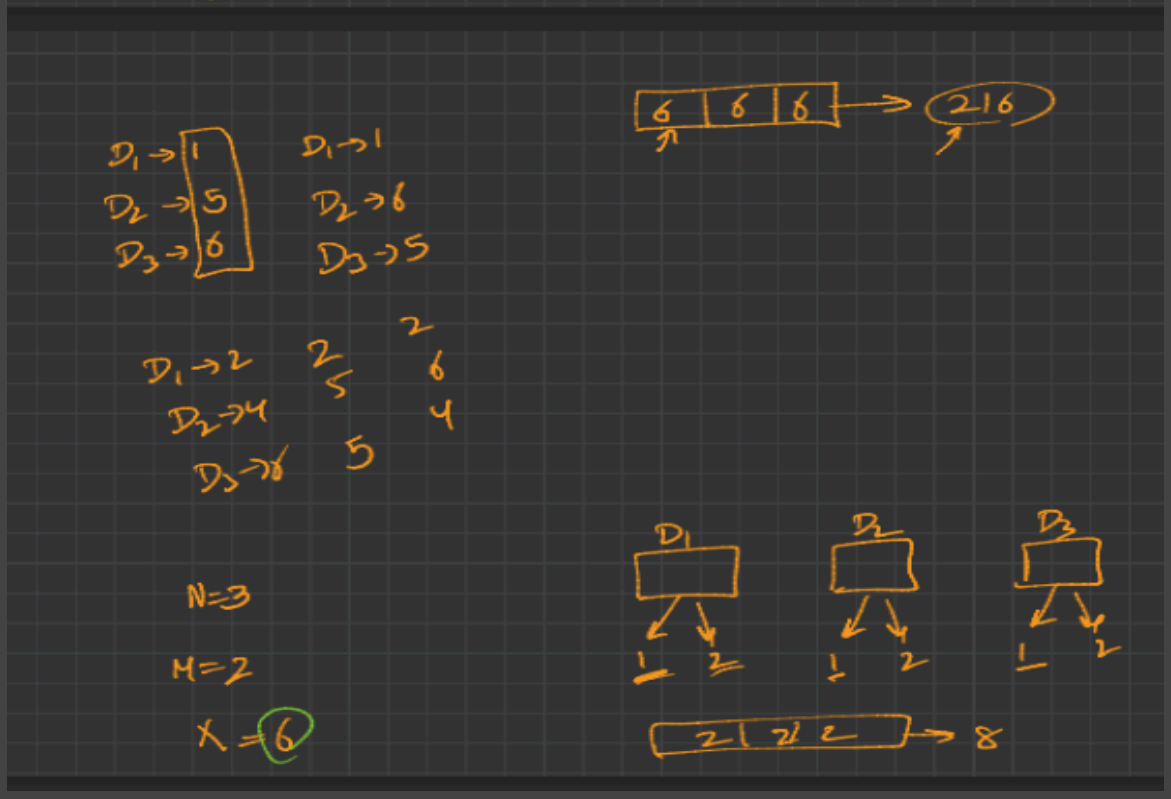
index
n
0 → k-2
1 → k-1
K/3 → 0
TOP Down
Bottom-Up

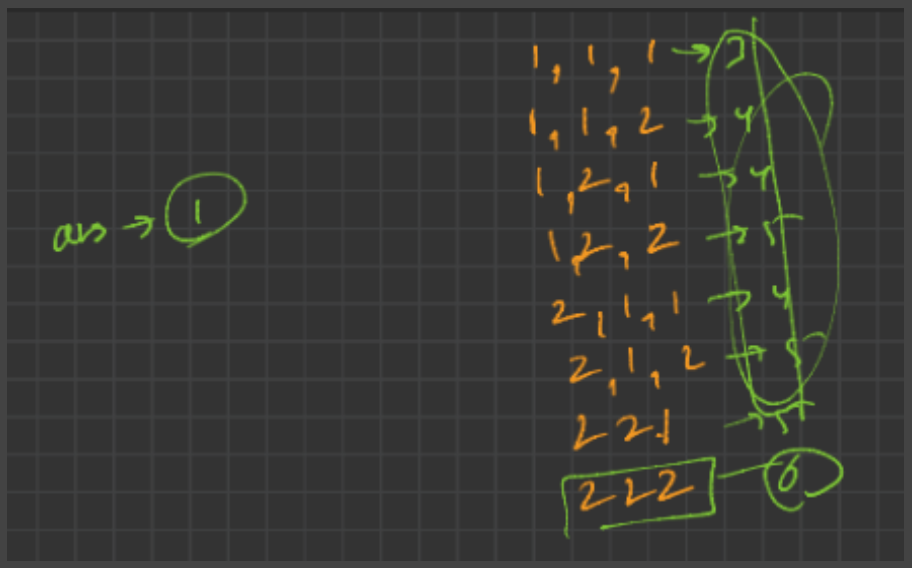


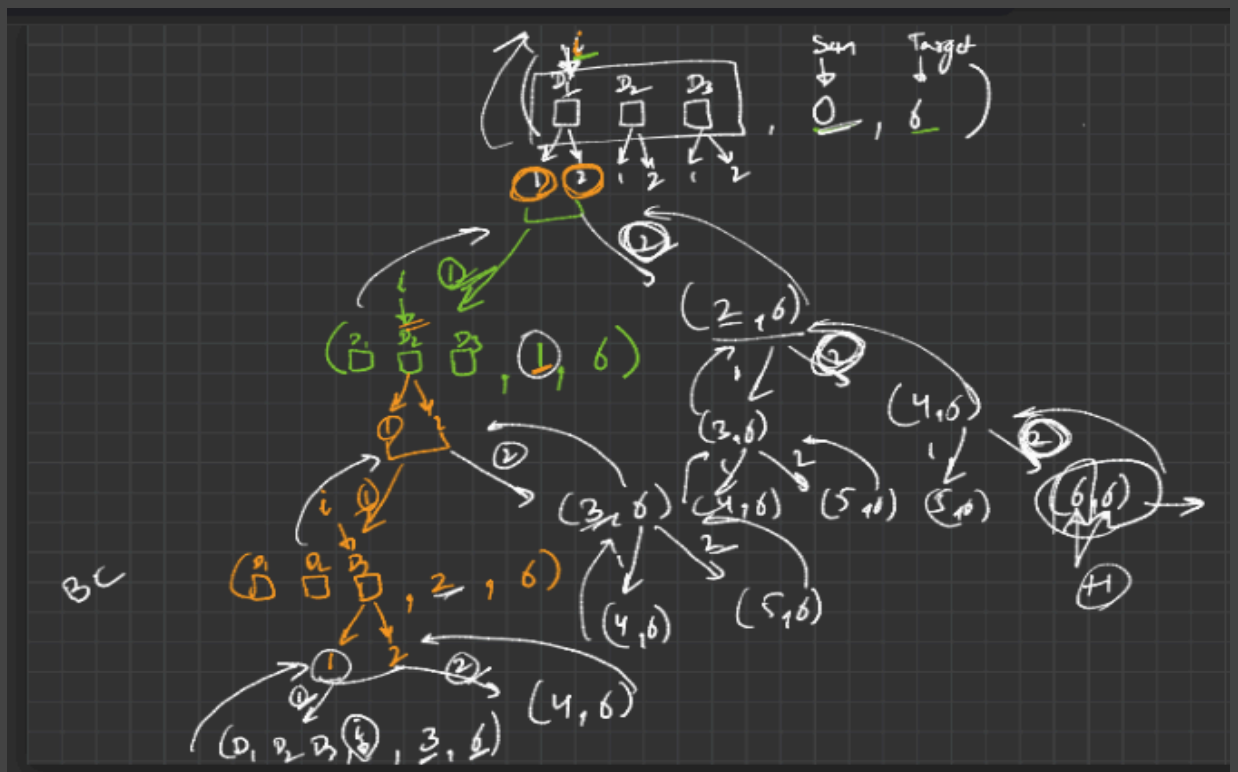
S-O → ?
next
dp1[index+2][n-1]
dp1[index][n]
dp1[index+1][n]
prev
curr
prev →
curr →
next →
next = curr
curr = prev

Number Of Dice Rolls With Target Sum Recursion / Top-Down Approach / Bottom-Up Approach / Space Optimization Approach:









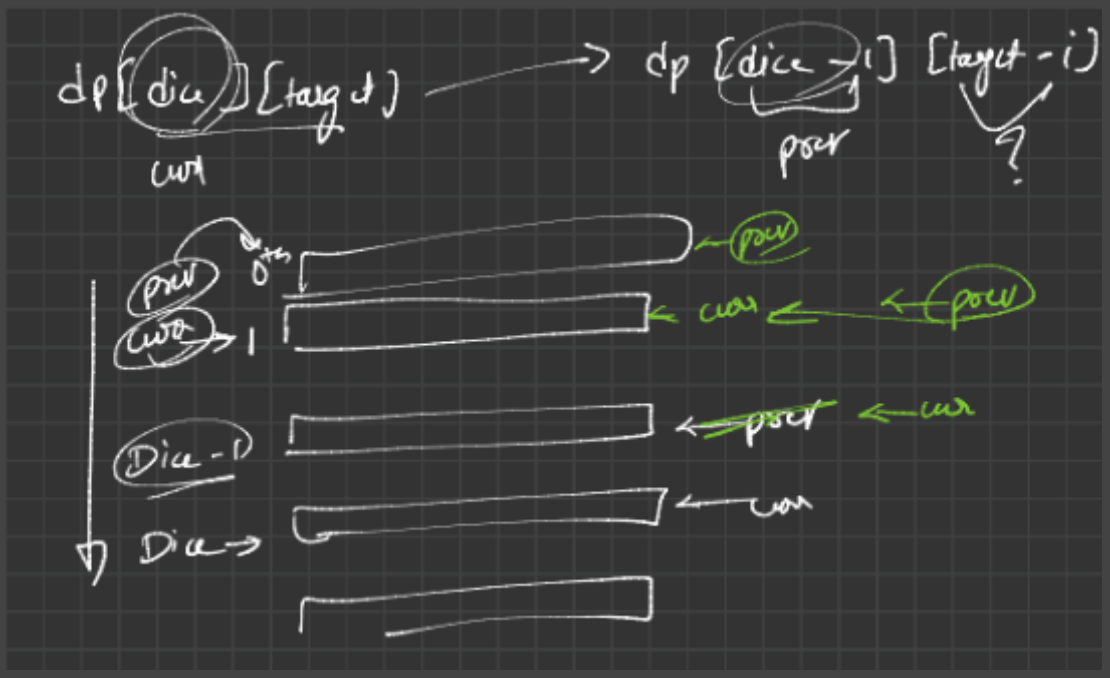
Top Down

5:0

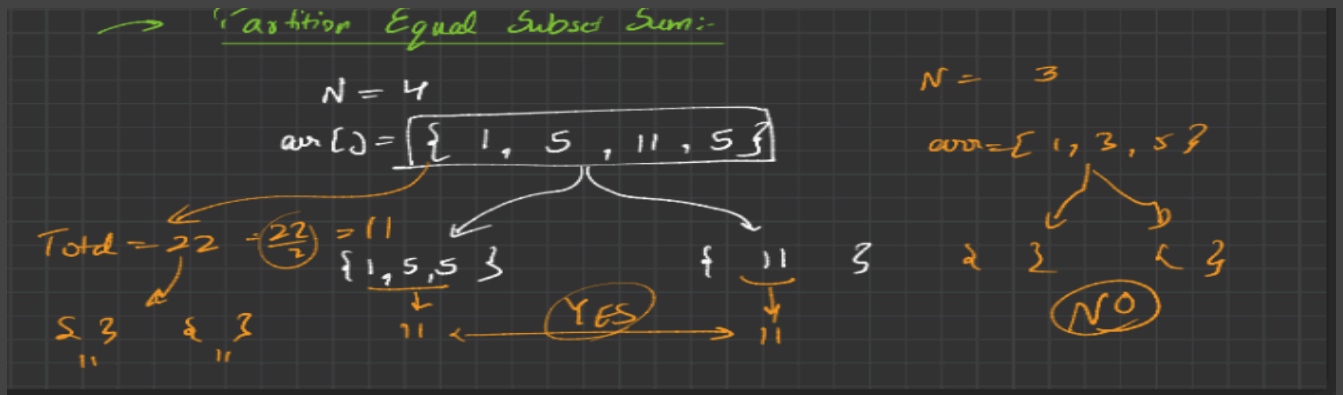
dice →

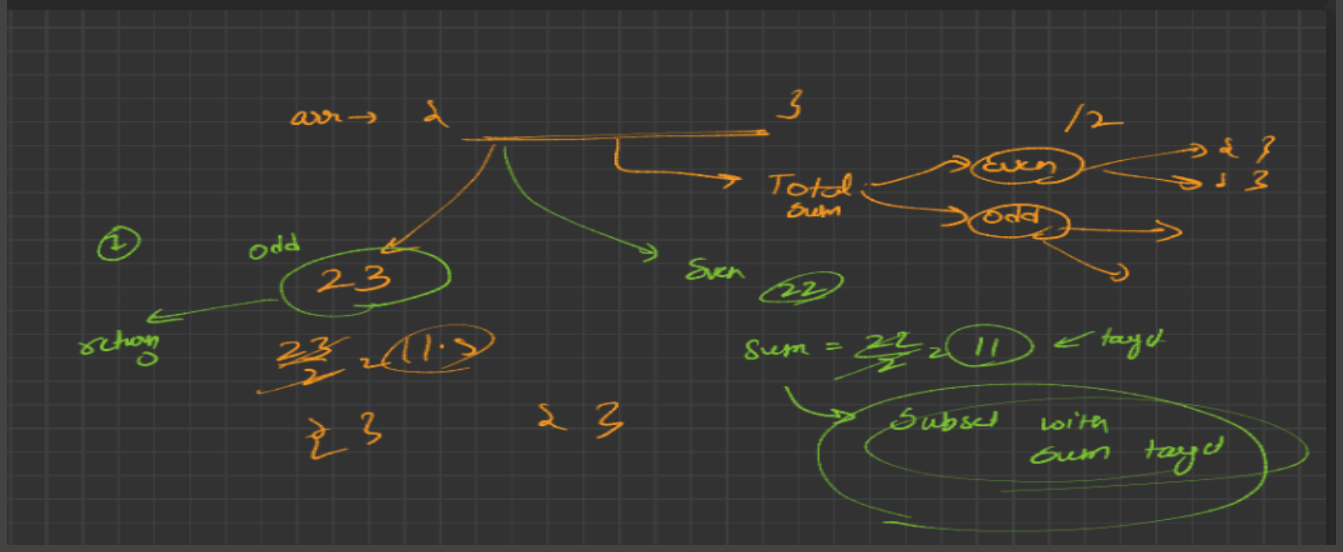
target →

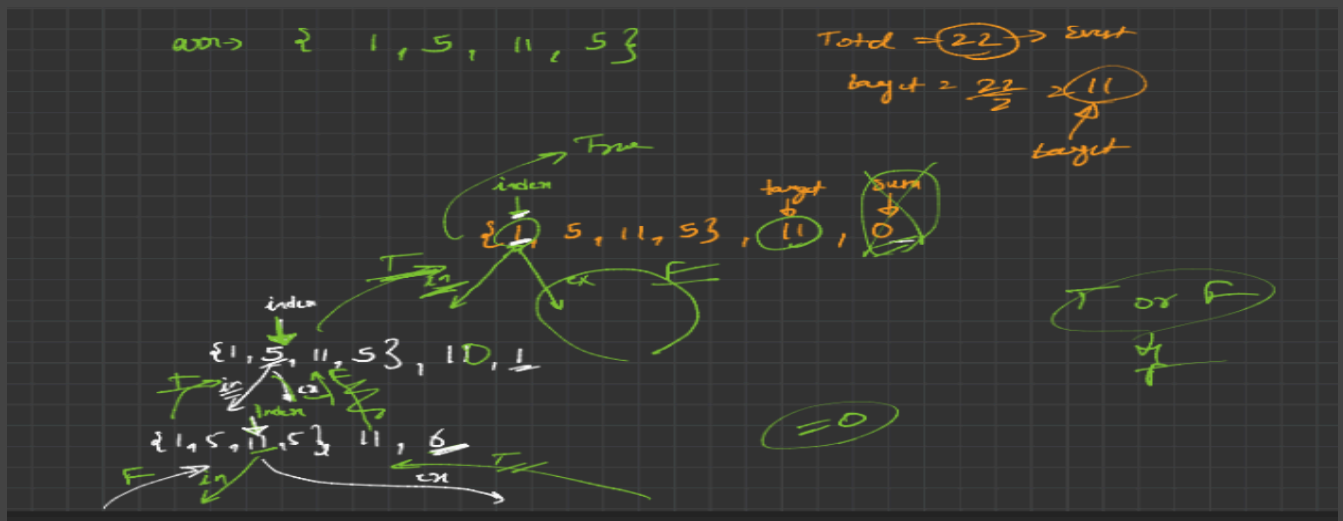
Bottom-Up

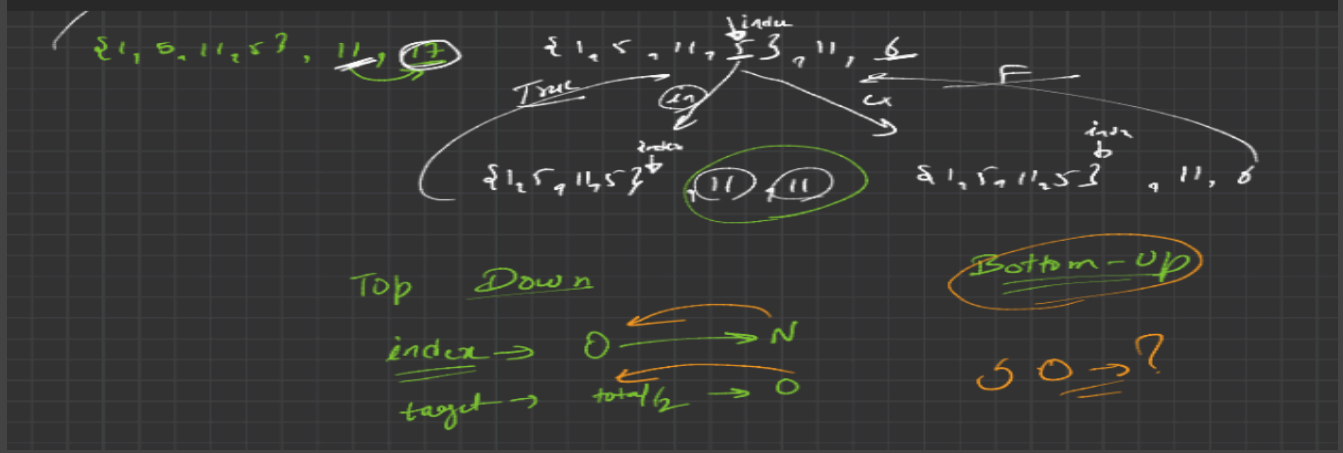


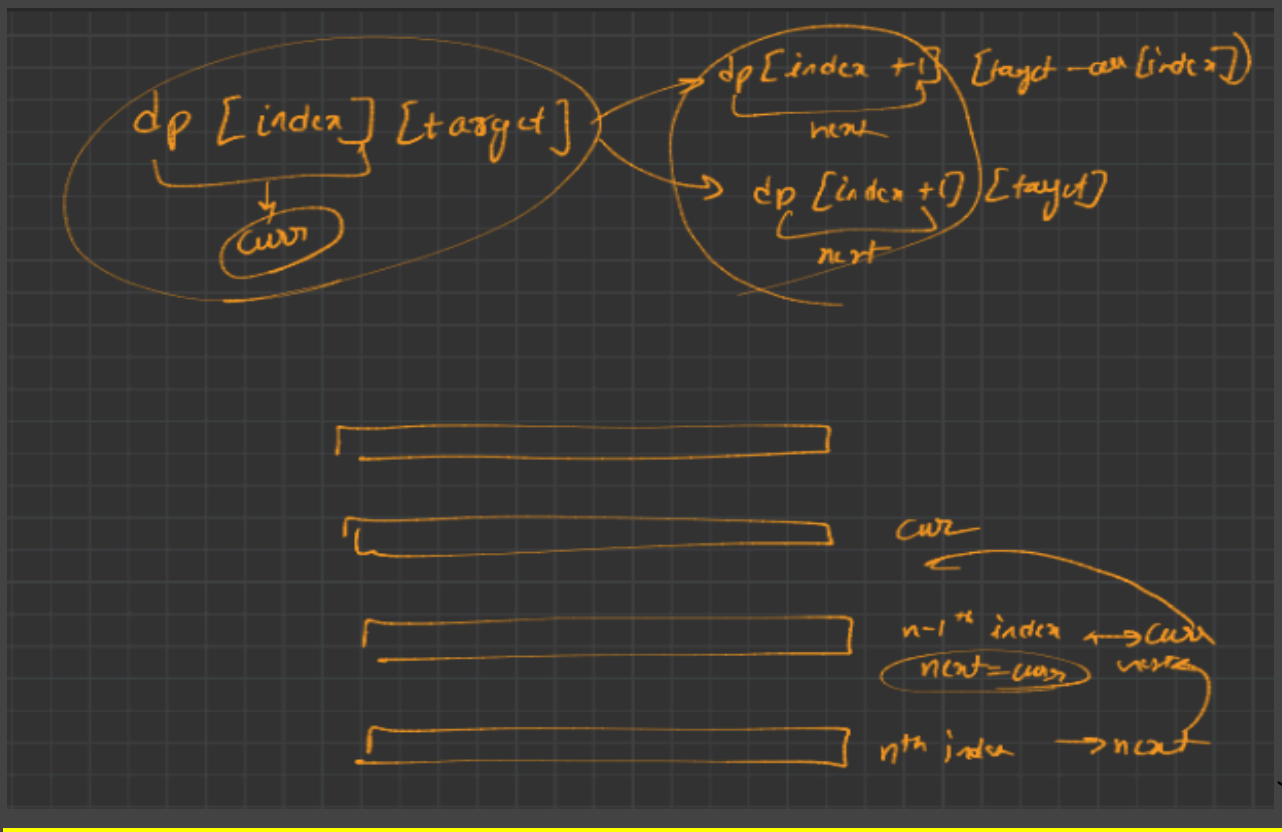
Partition Equal Subset Sum By Recursion / Top-Down Approach / Bottom-Up Approach / Space Optimization Approach:



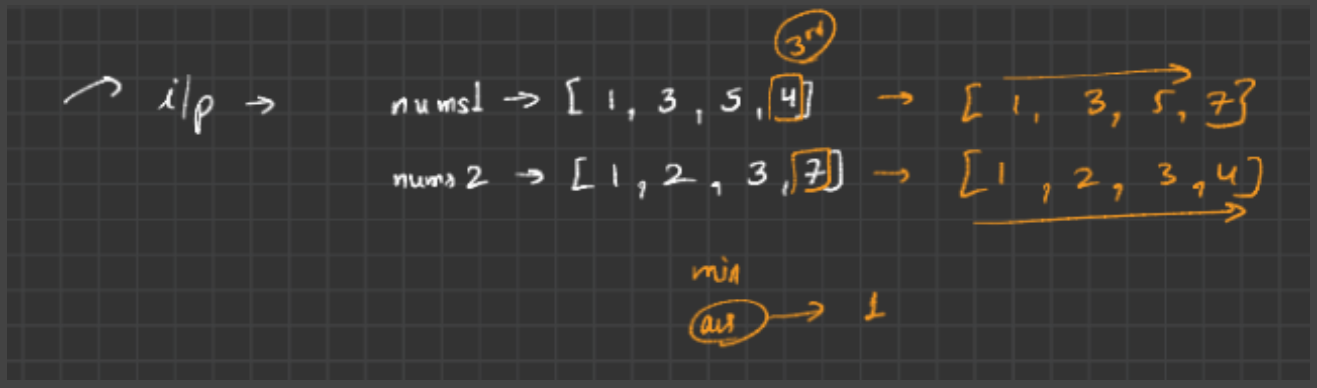


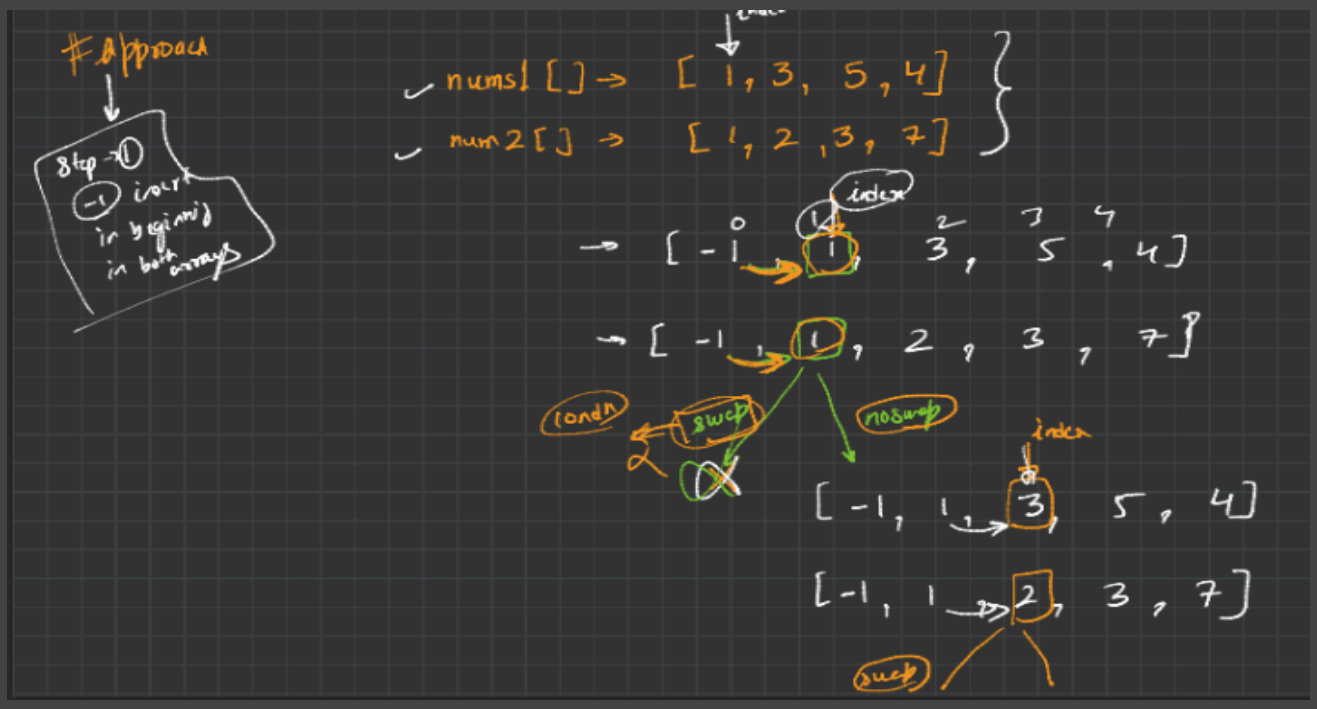


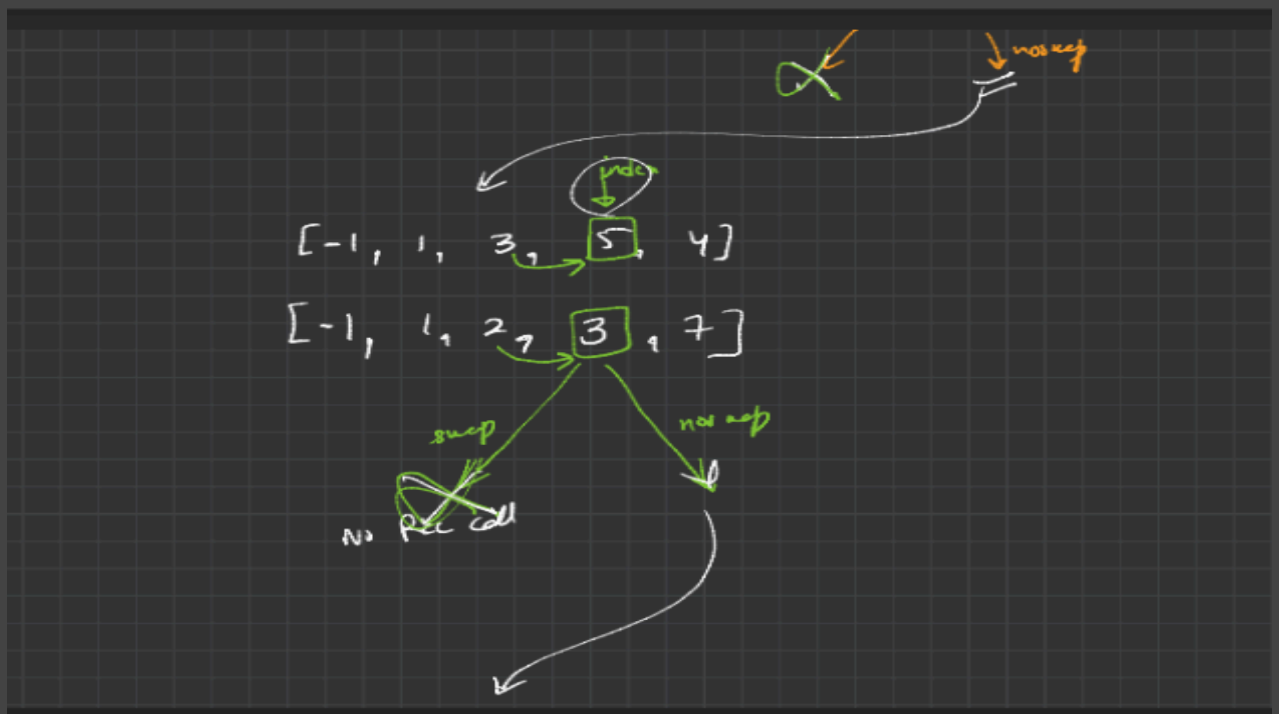


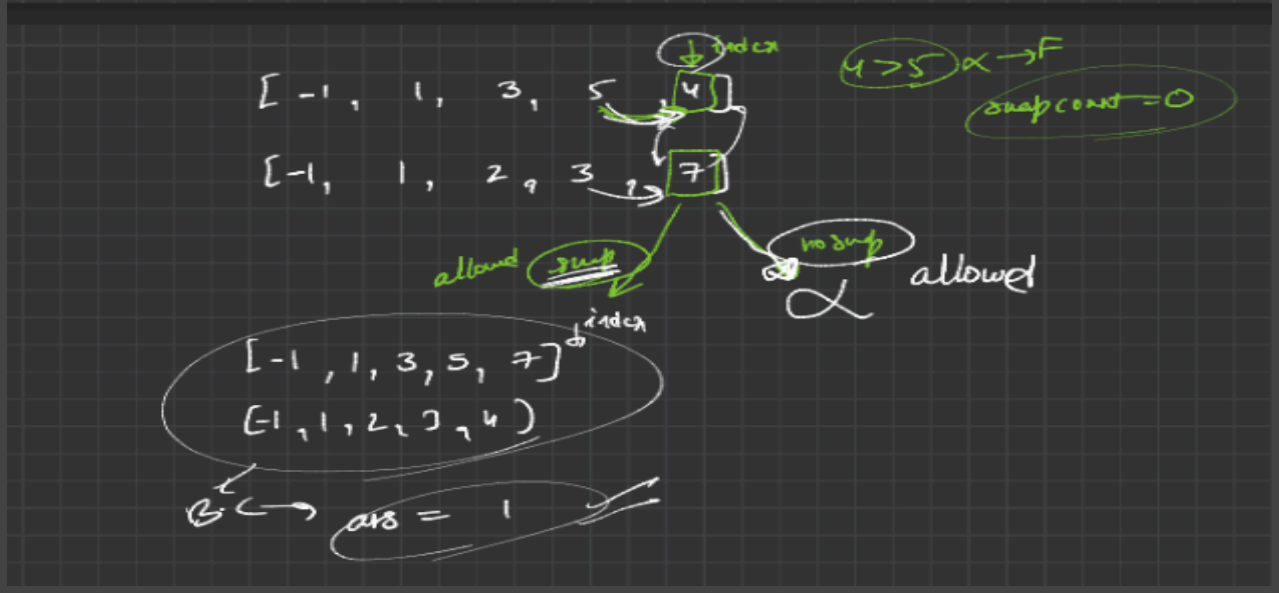


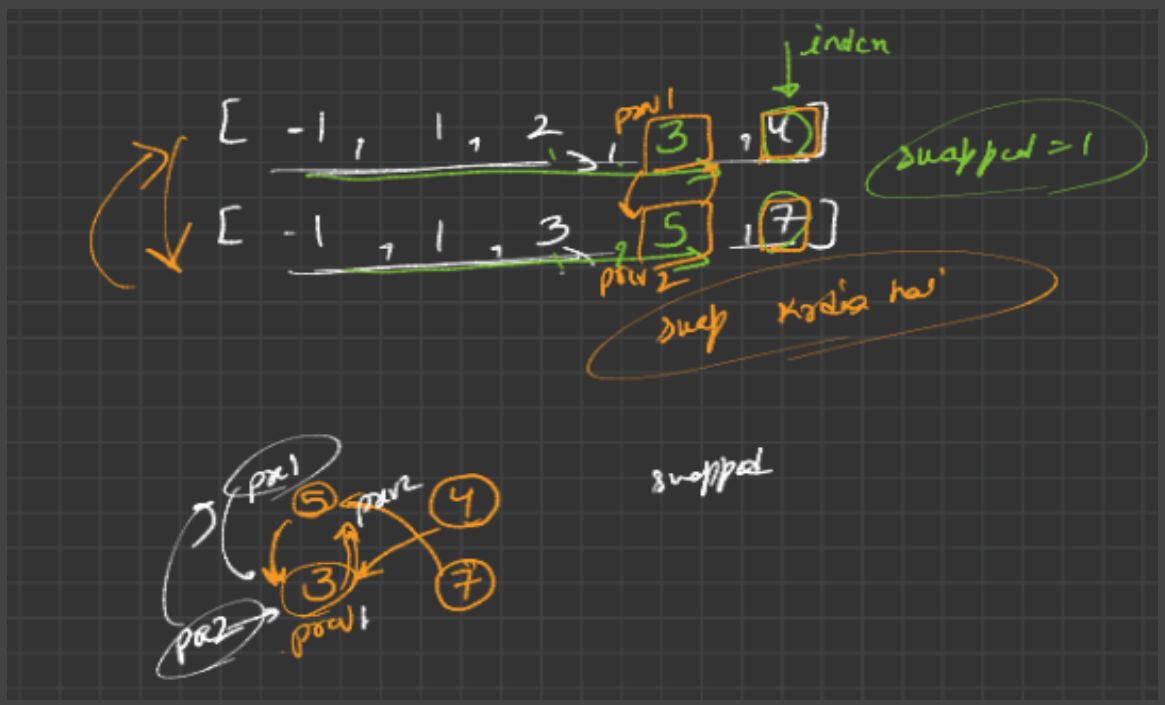
Minimum Swaps To Make Subsequence Increasing By Recursion / Top-Down Approach / Bottom-Up Approach / Space Optimization Approach:

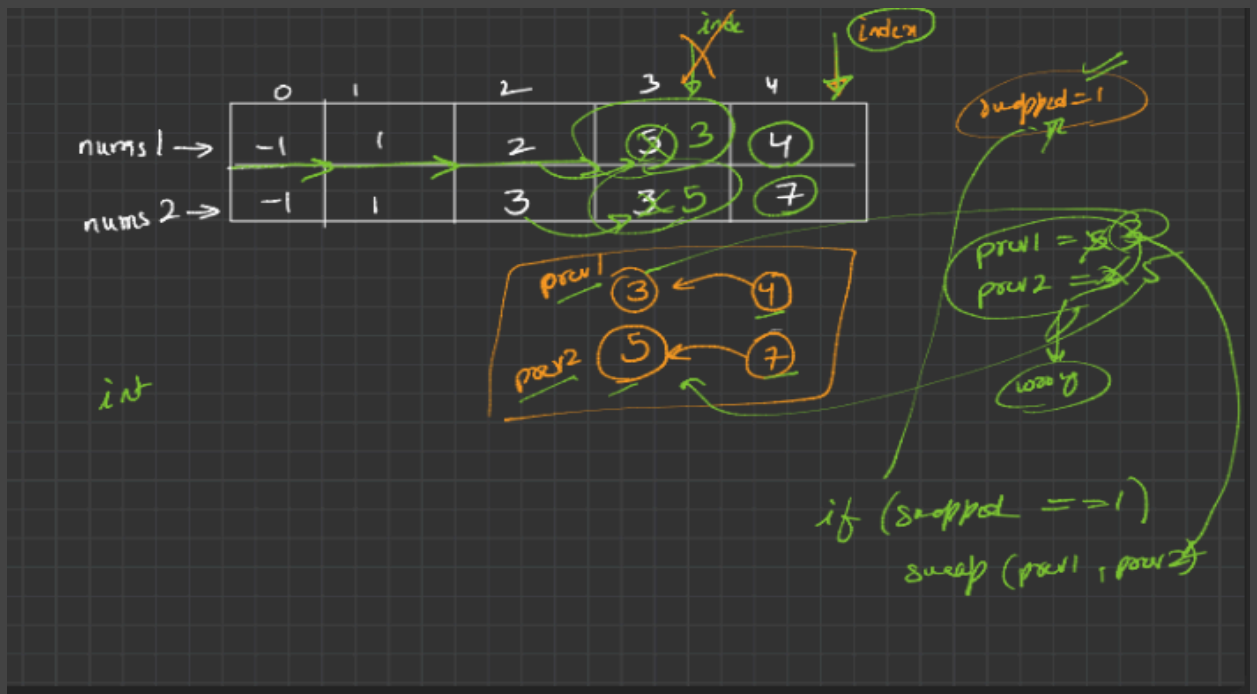


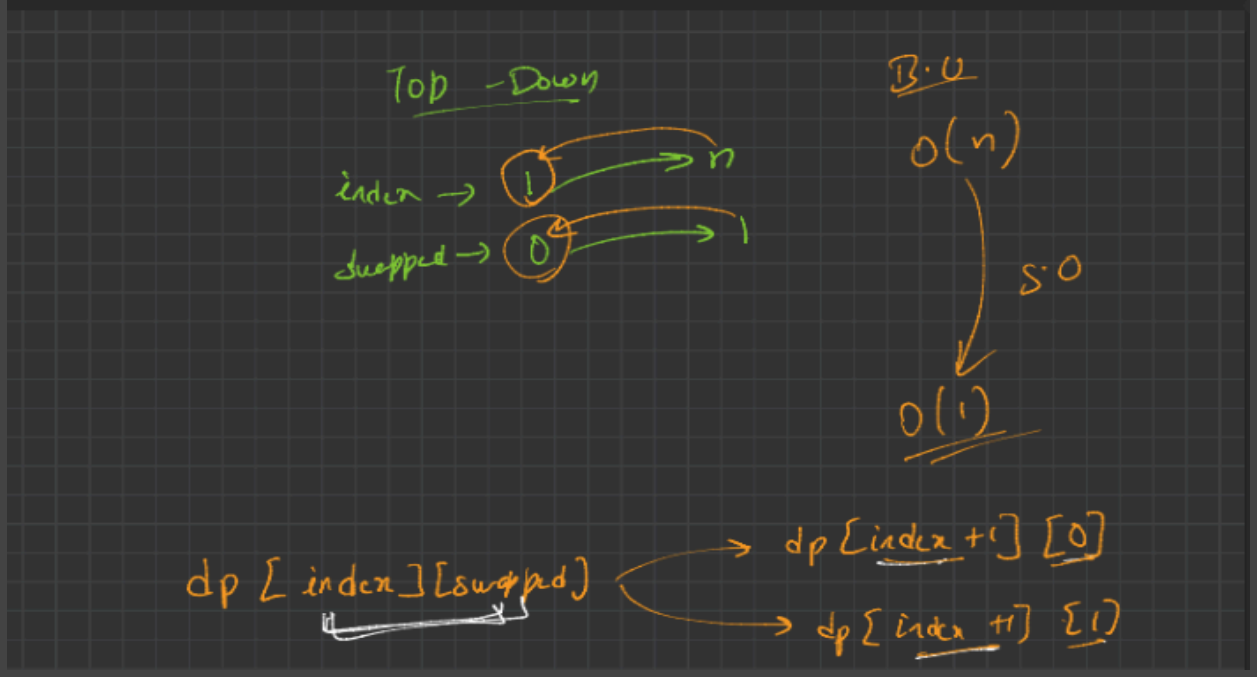


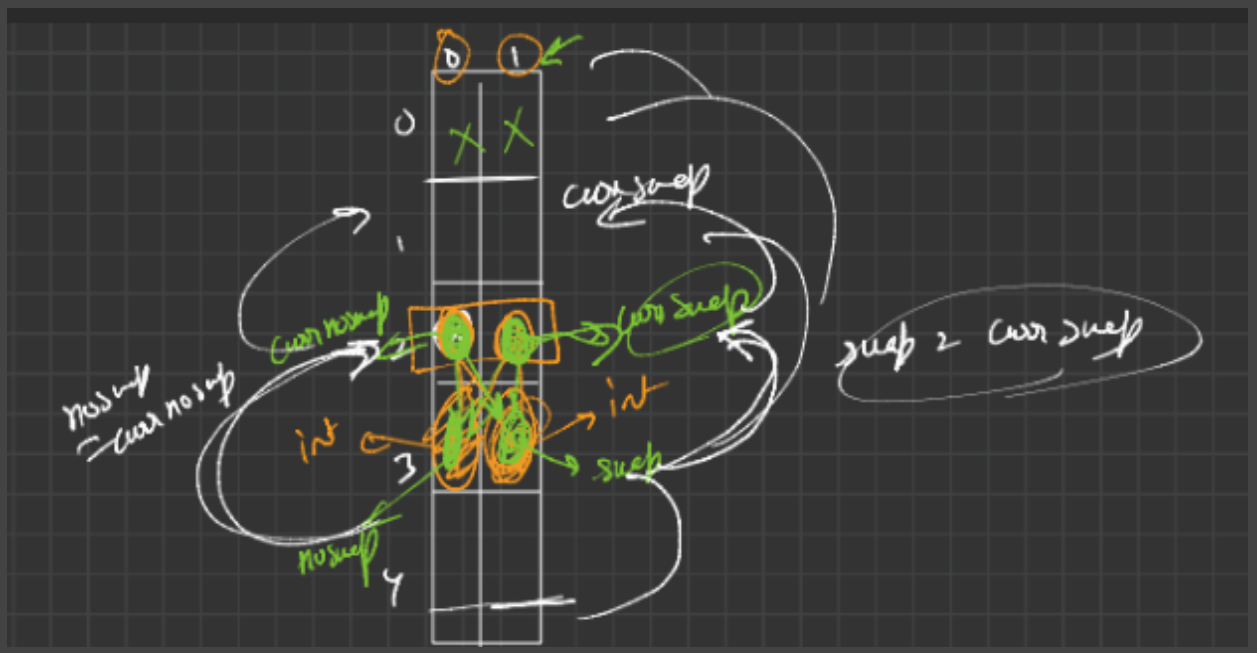

index
[-1, 1, 3, 5, 4]
[-1, 1, 2, 3, 7]
swap
no swap
No Rec call


index
[-1, 1, 3, 5, 4]
[-1, 1, 2, 3, 7]
4>5 → F
swapCount = 0
allowed
swap
no swap
allowed
index
[-1, 1, 3, 5, 7]
[-1, 1, 2, 3, 4]
ans = 1

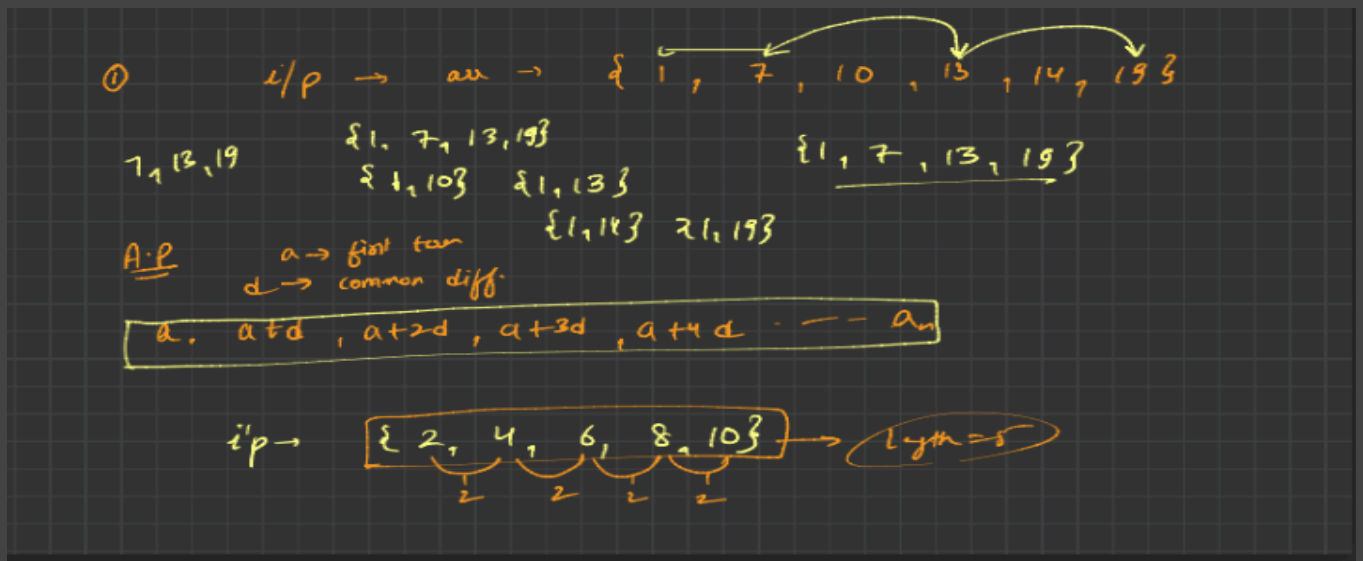

index
[-1, 1, 2, 3, 4]
prev
swapped = 1
[-1, 1, 3, 5, 7]
prev
swap karna hai
swapped
prev
prev2
prev1

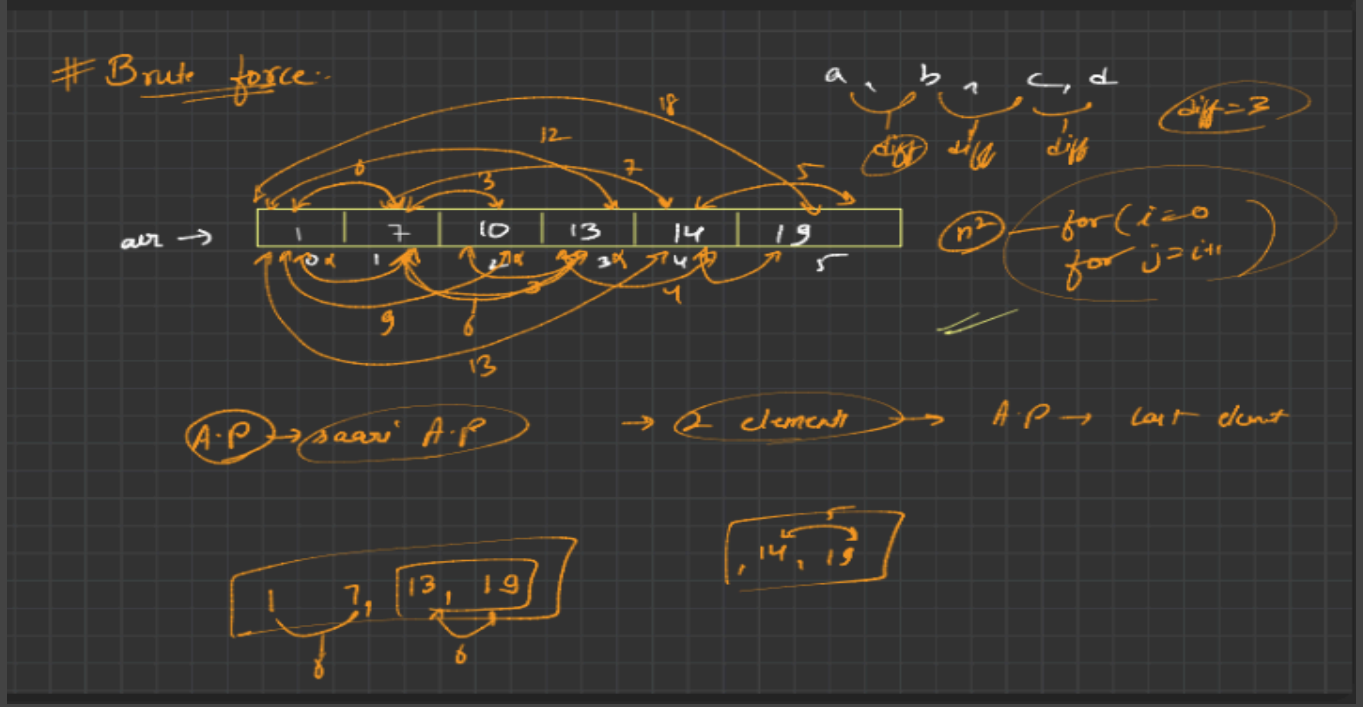

index
0 1 2 3 4
nums1 → -1 1 2 3 3 4
nums2 → -1 1 3 3 5 7
swapped = 1
prev1 = 3
prev2 = 5
prev1 3 4
prev2 5 7
int
if (swapped == 1)
swap (prev1 , prev2)

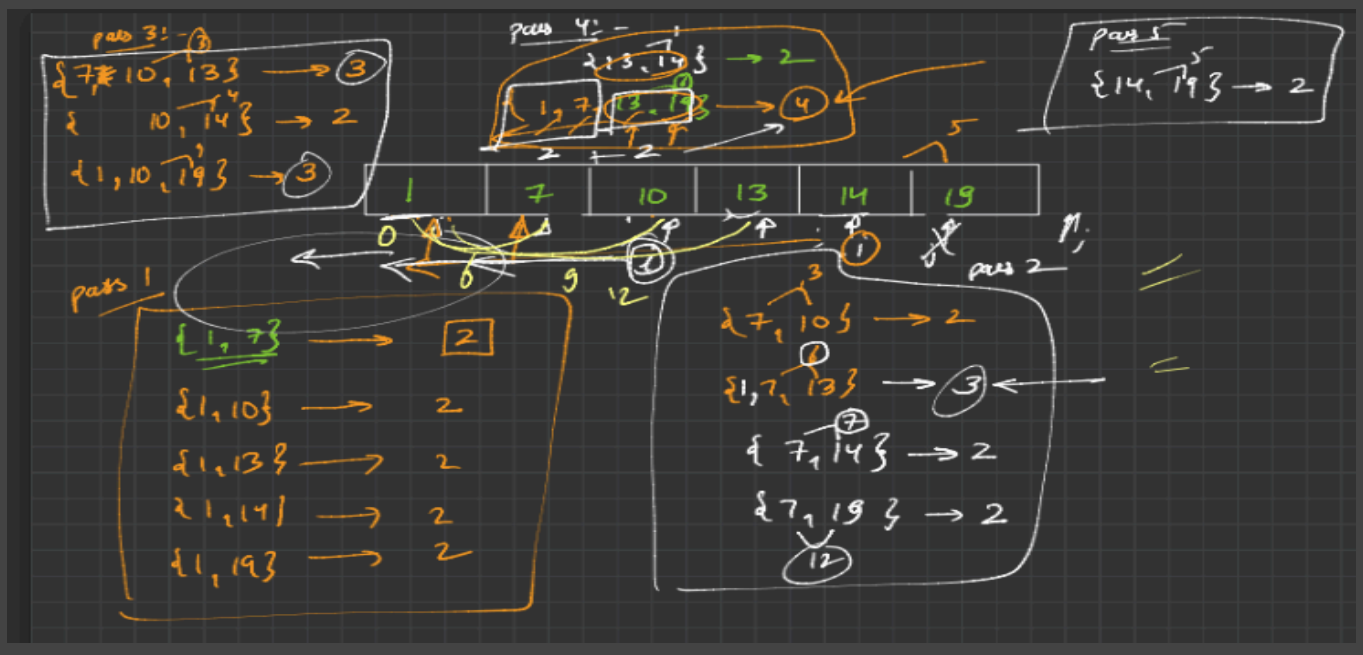

Top - Down
index → 1 → n
swapped → 0 → 1
B.U
O(n)
S.O
O(1)
dp [index] [swapped]
dp [index + 1] [0]
dp [index + 1] [1]

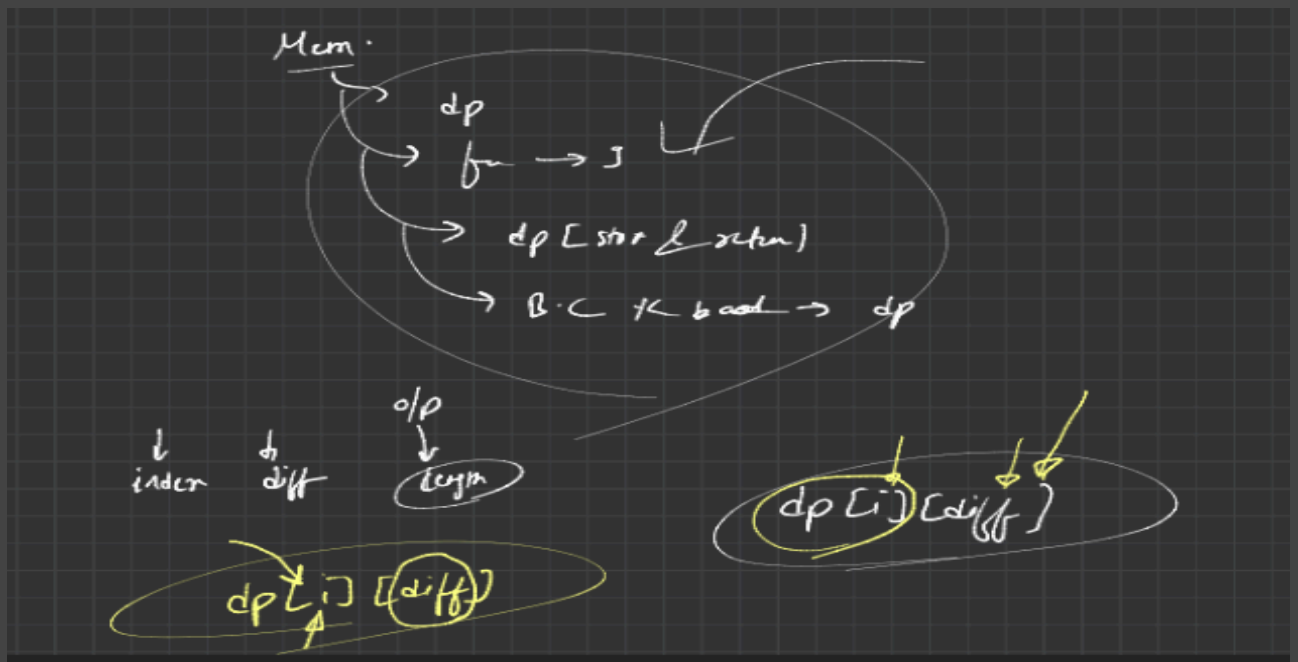

0 1
0
1
curr swap
curr noswap
curr swap
noswap = curr noswap
int
3
int
swap
noswap
4
swap = curr swap

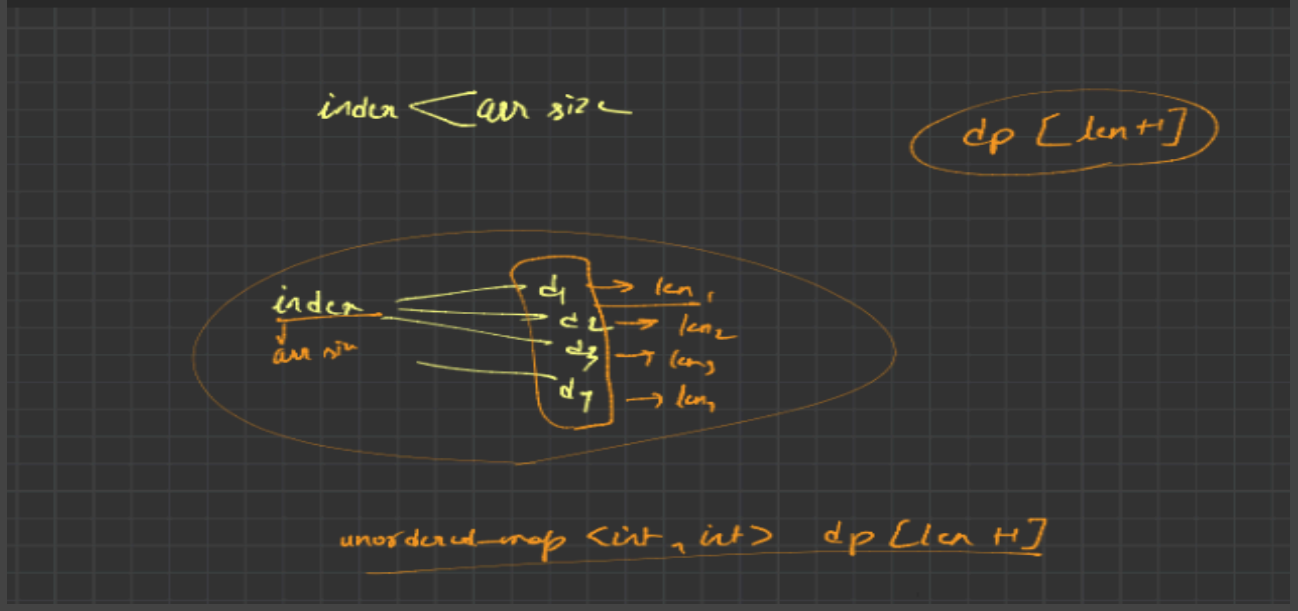
Longest Arithmetic Subsequence And Russian Dolls Increasing By Recursion / Top-Down Approach / Bottom-Up Approach:

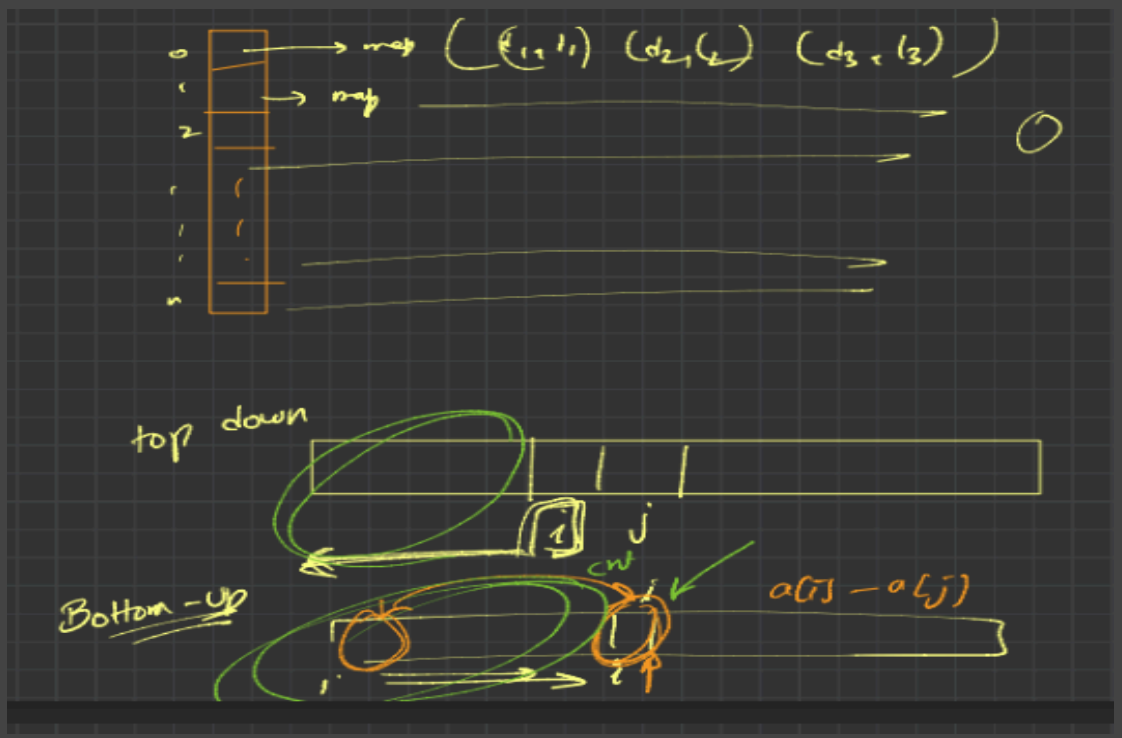


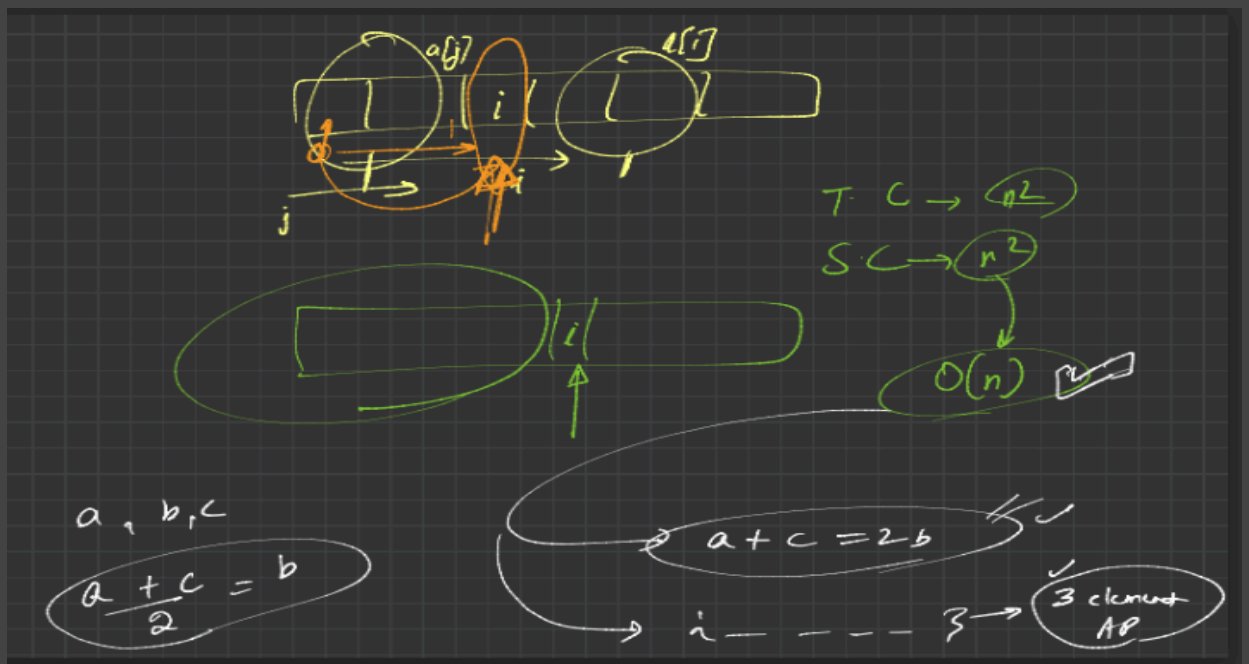


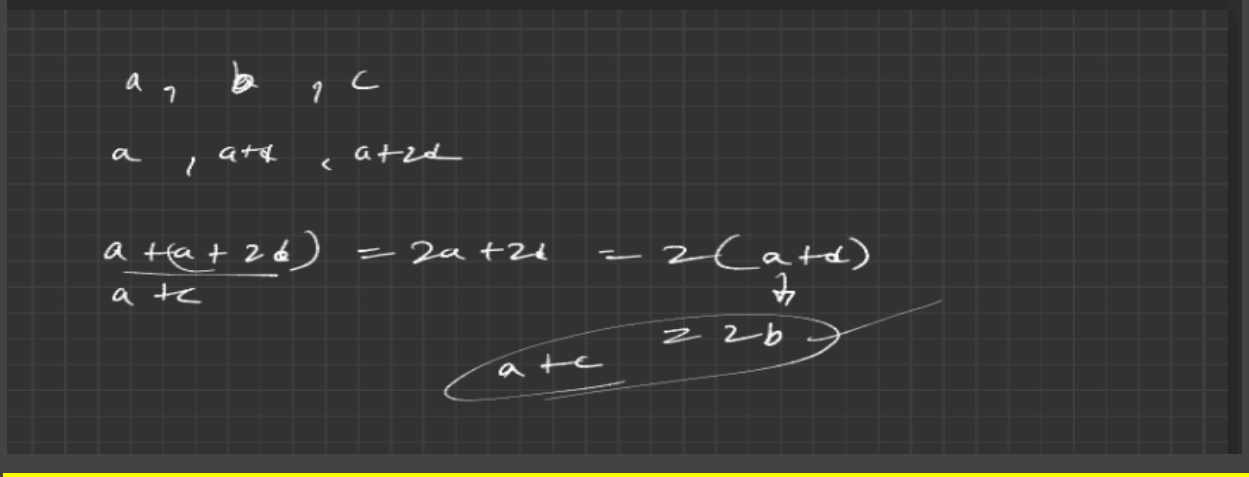



Mem.
dp
for → J
dp [start & return]
B.C if key found → dp
o/p
index
diff
length
dp [i] [diff]
dp [i] [diff]

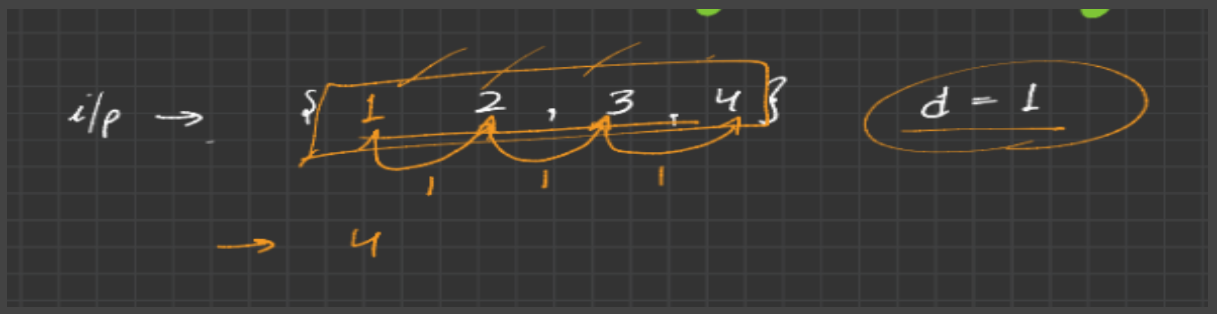

index < arr size
dp [len+1]
index
arr size
d1 → len1
d2 → len2
d3 → len3
d7 → len7
unordered_map <int, int> dp [len+1]

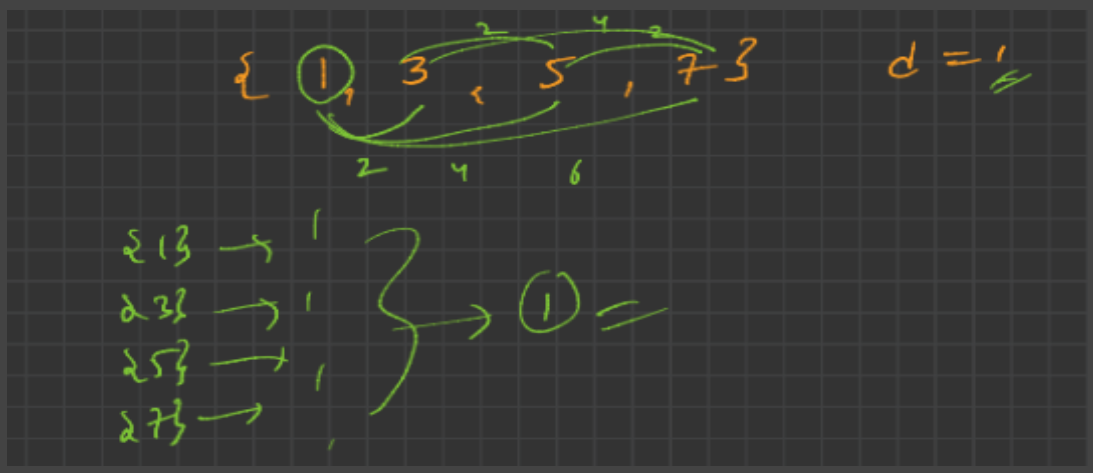

0
1
2
n
map
map
(d1, l1) (d2, l2) (d3, l3)
0
top down
i
j
Bottom-Up
a[i] - a[j]

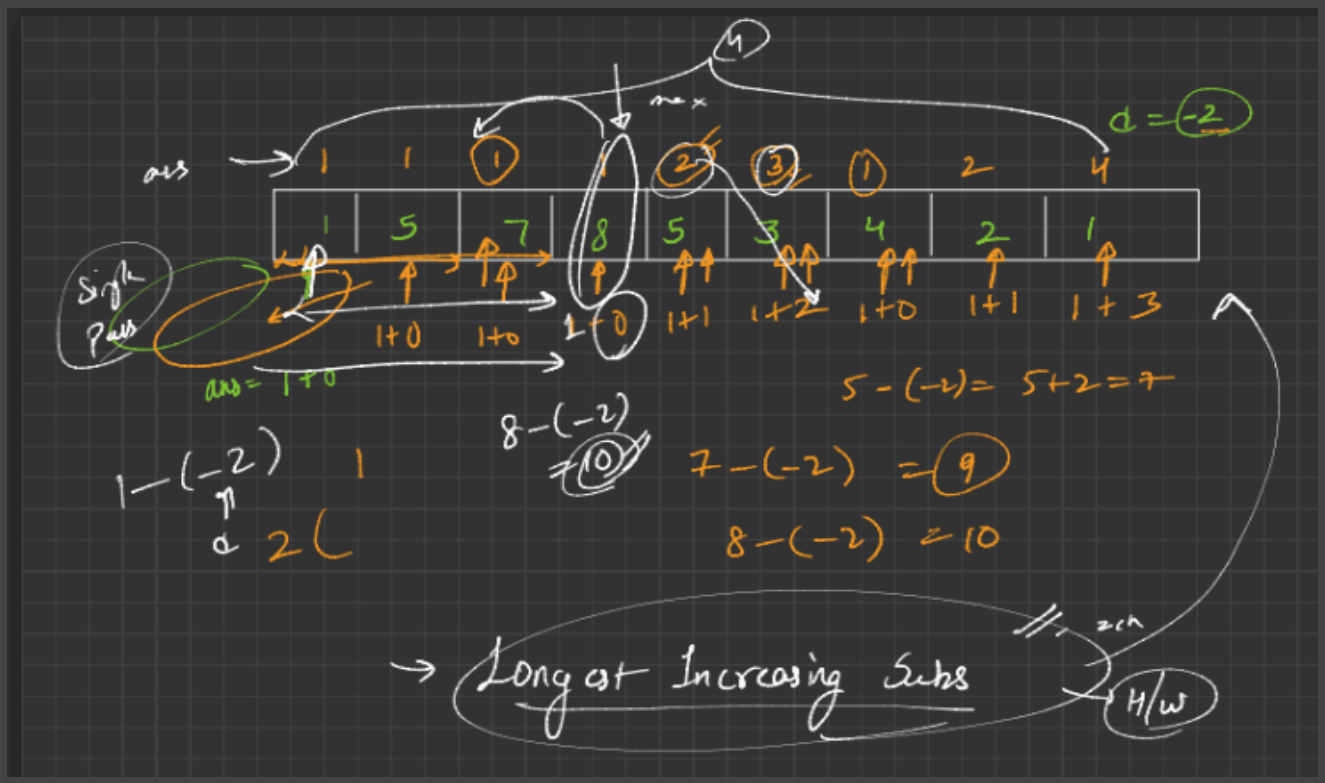




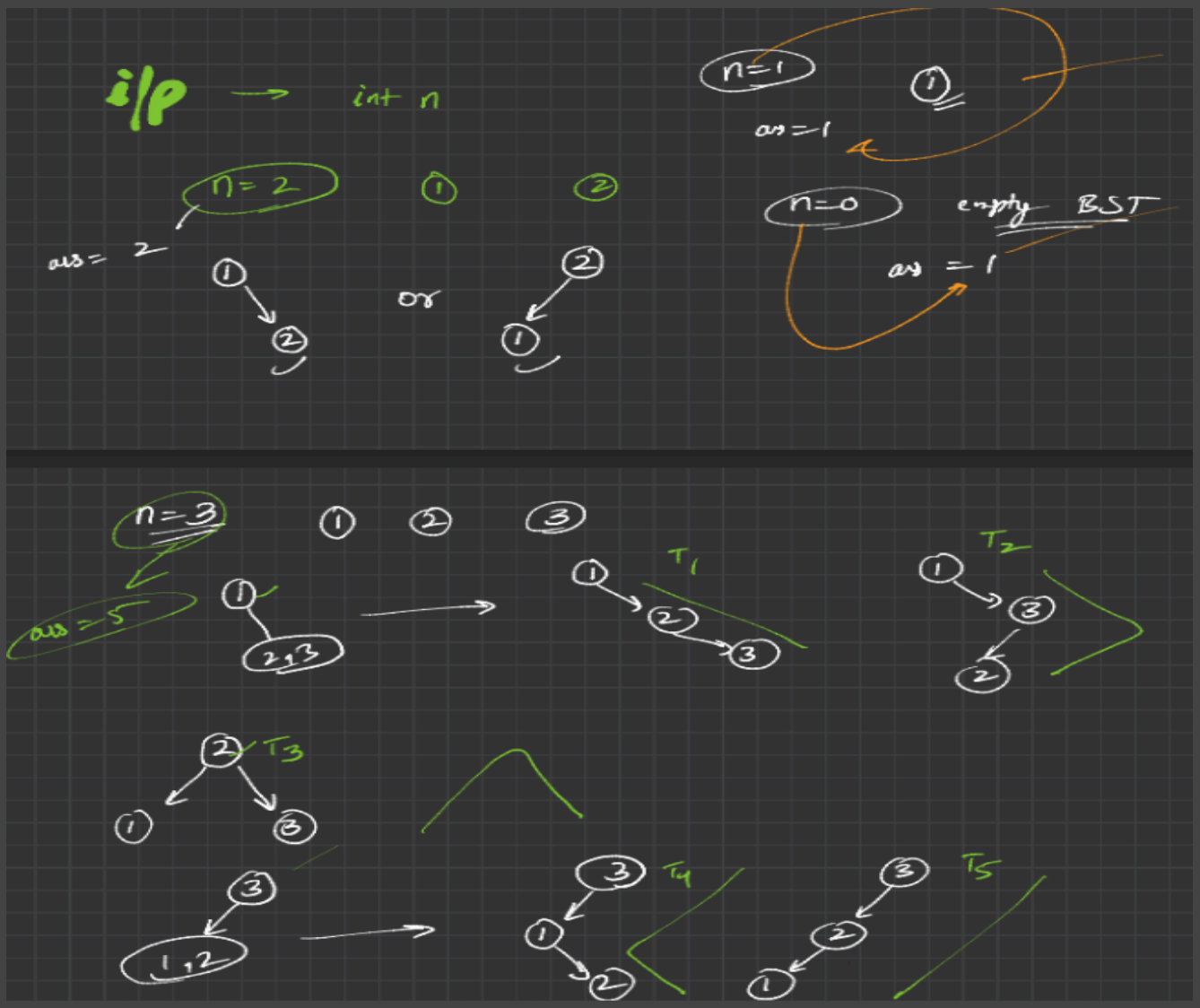
Longest AP With Given Difference(d) By Recursion / Top-Down Approach / Bottom-Up Approach / Space Optimization Approach:

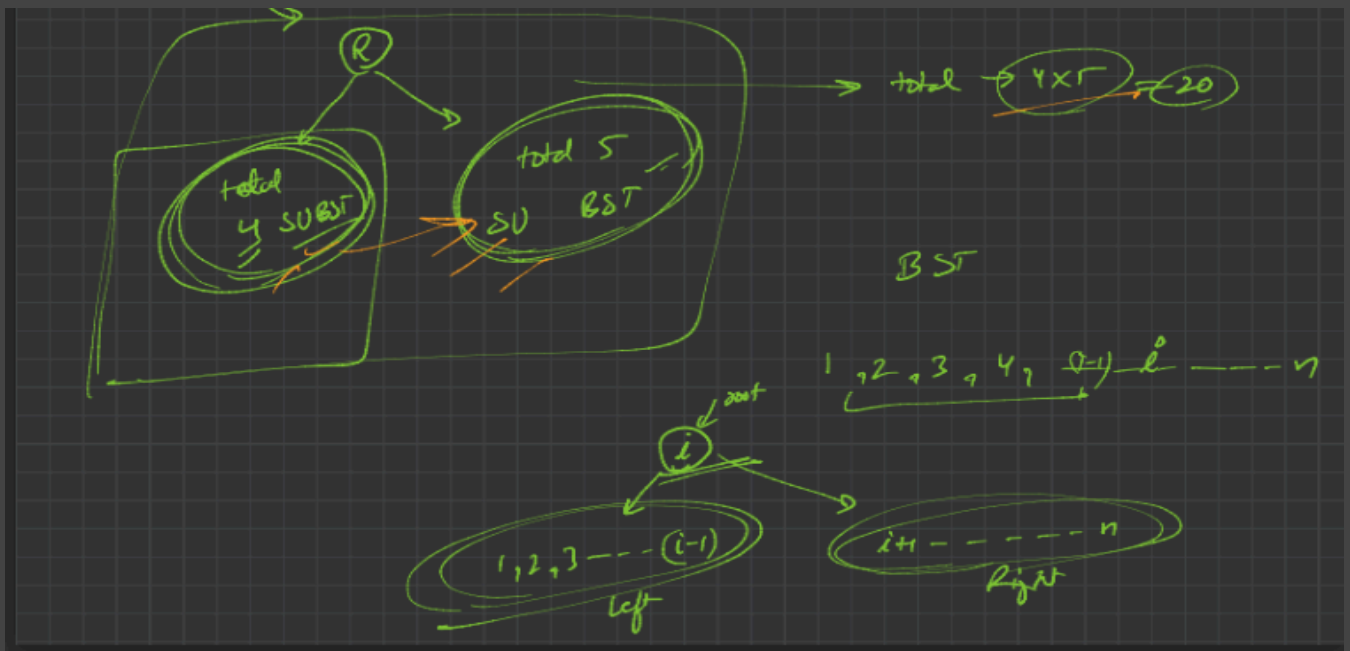


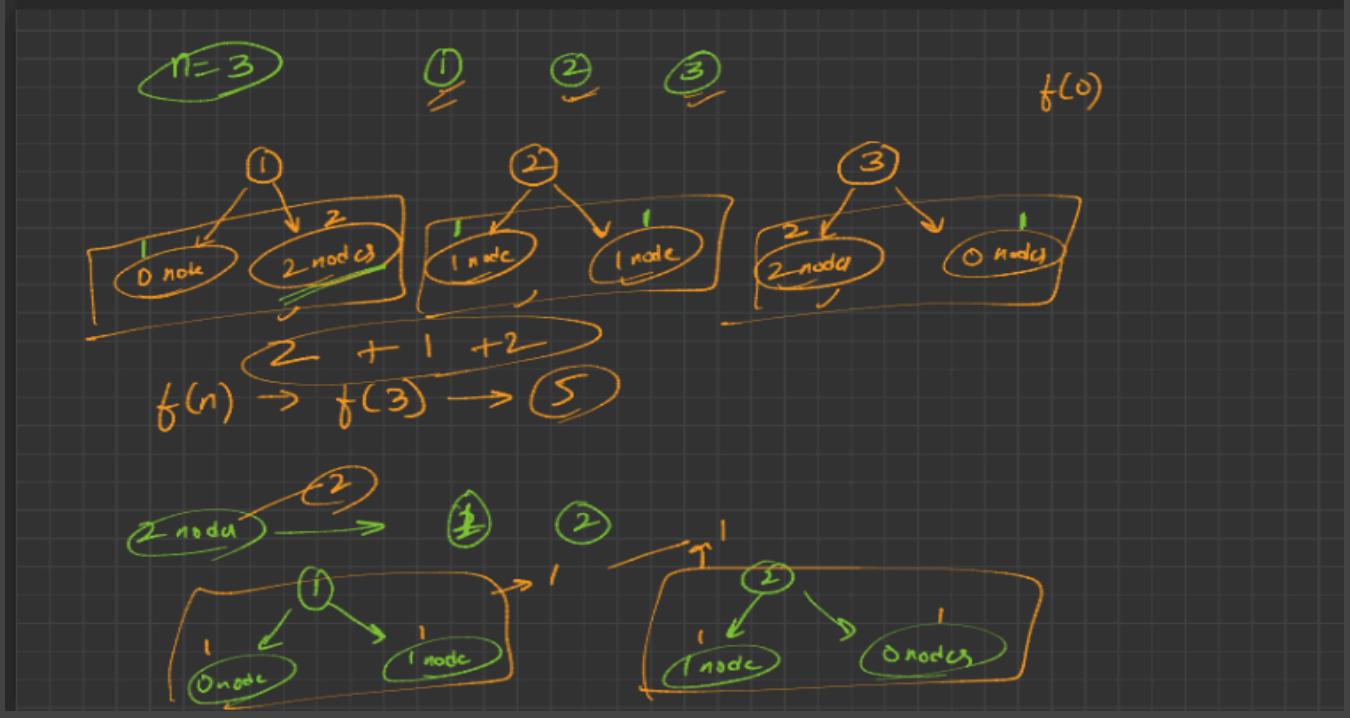


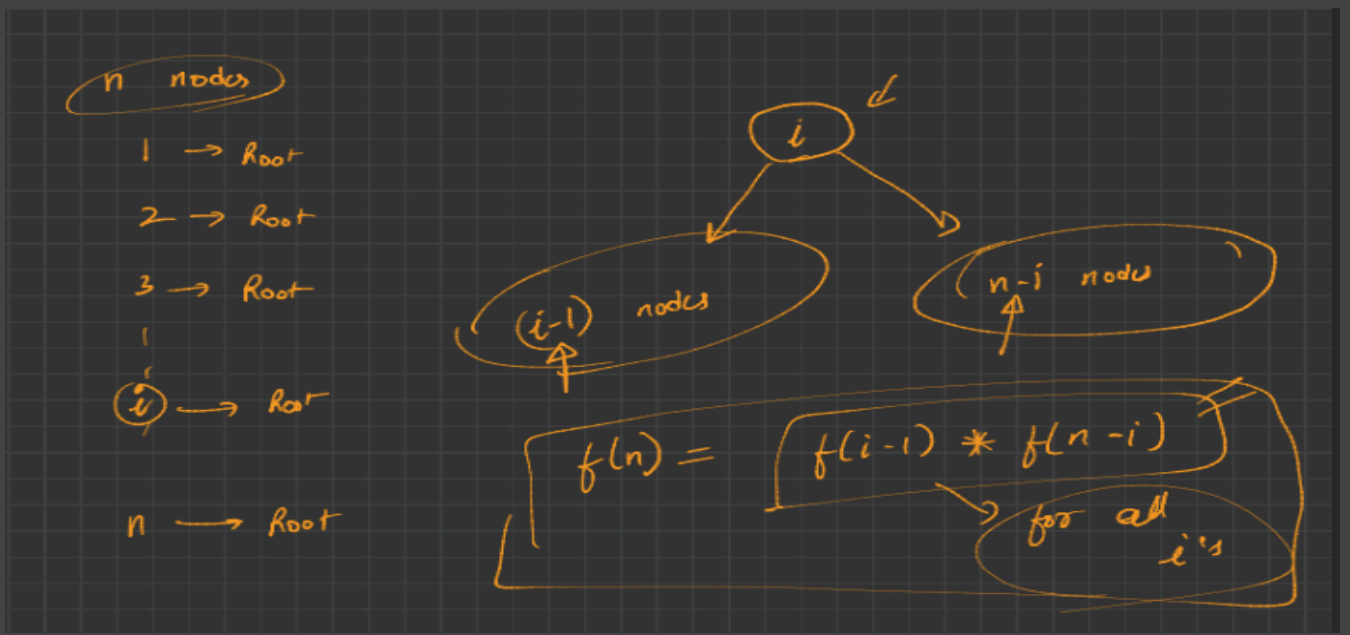


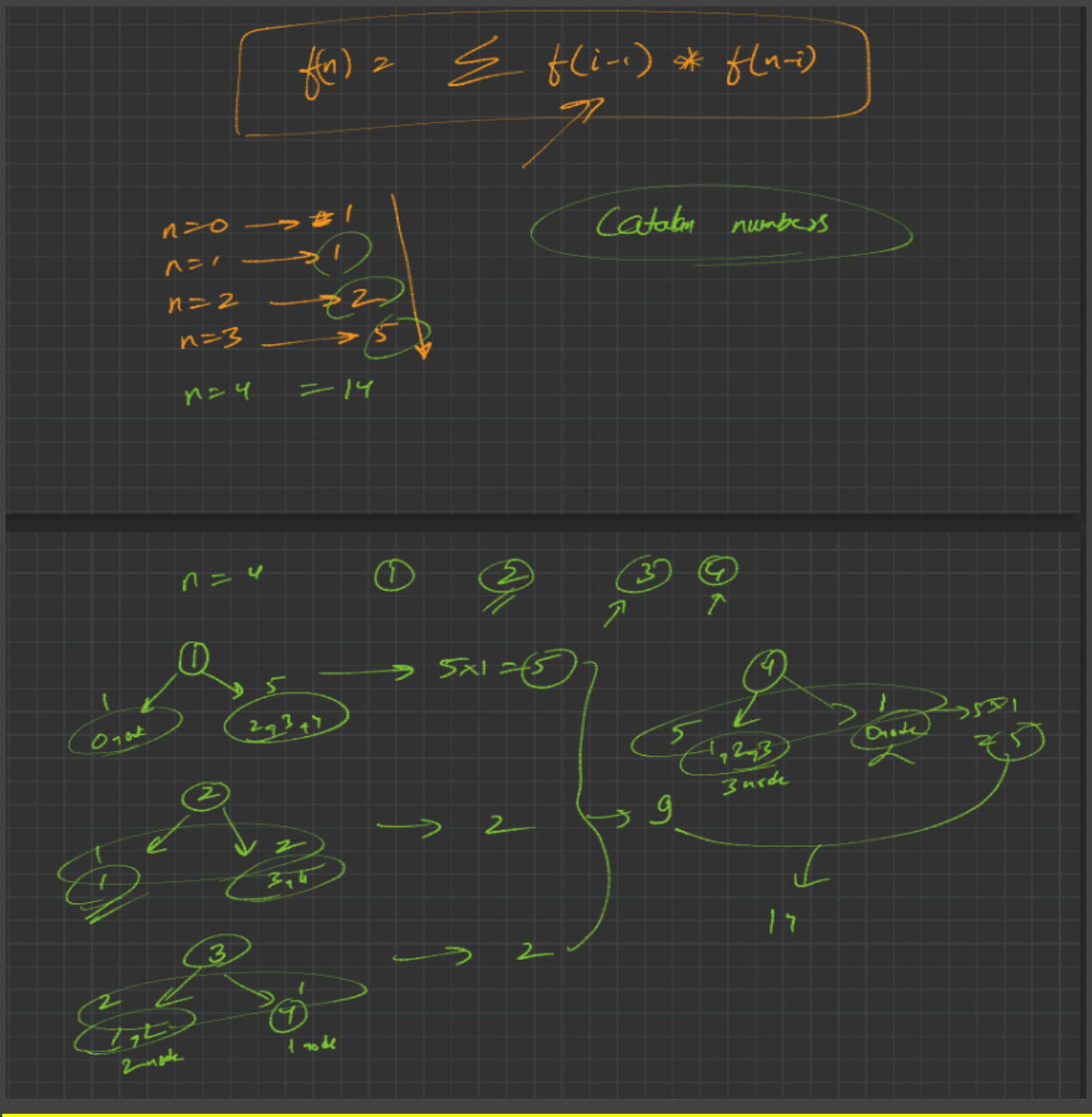
Unique Binary Search Tree By Recursion / Top-Down Approach / Bottom-Up Approach:



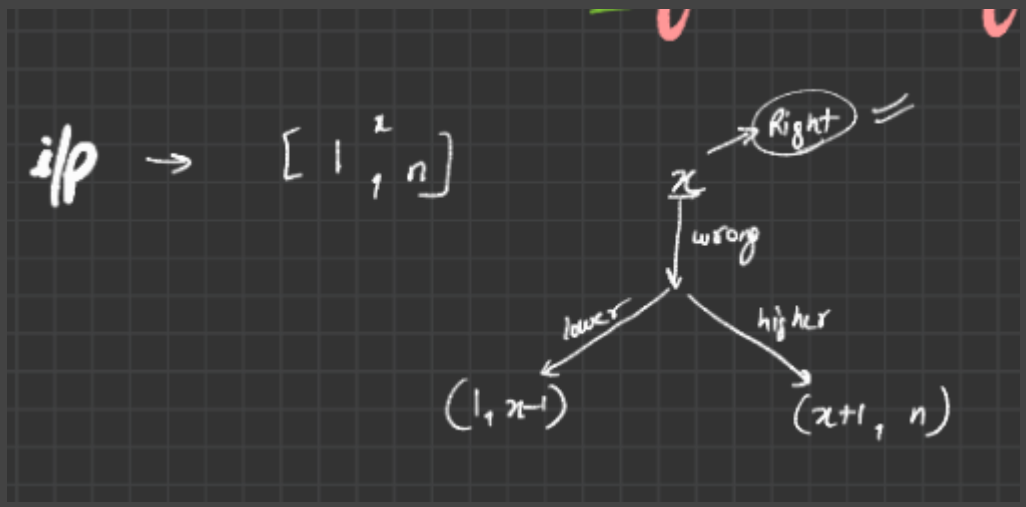

R
total
4 SUBST
total 5
SU BST
total → 4 x 5 = 20
BST
1, 2, 3, 4, (i-1) i ---- n
root
i
1,2,3---(i-1)
Left
i+1 ------ n
Right

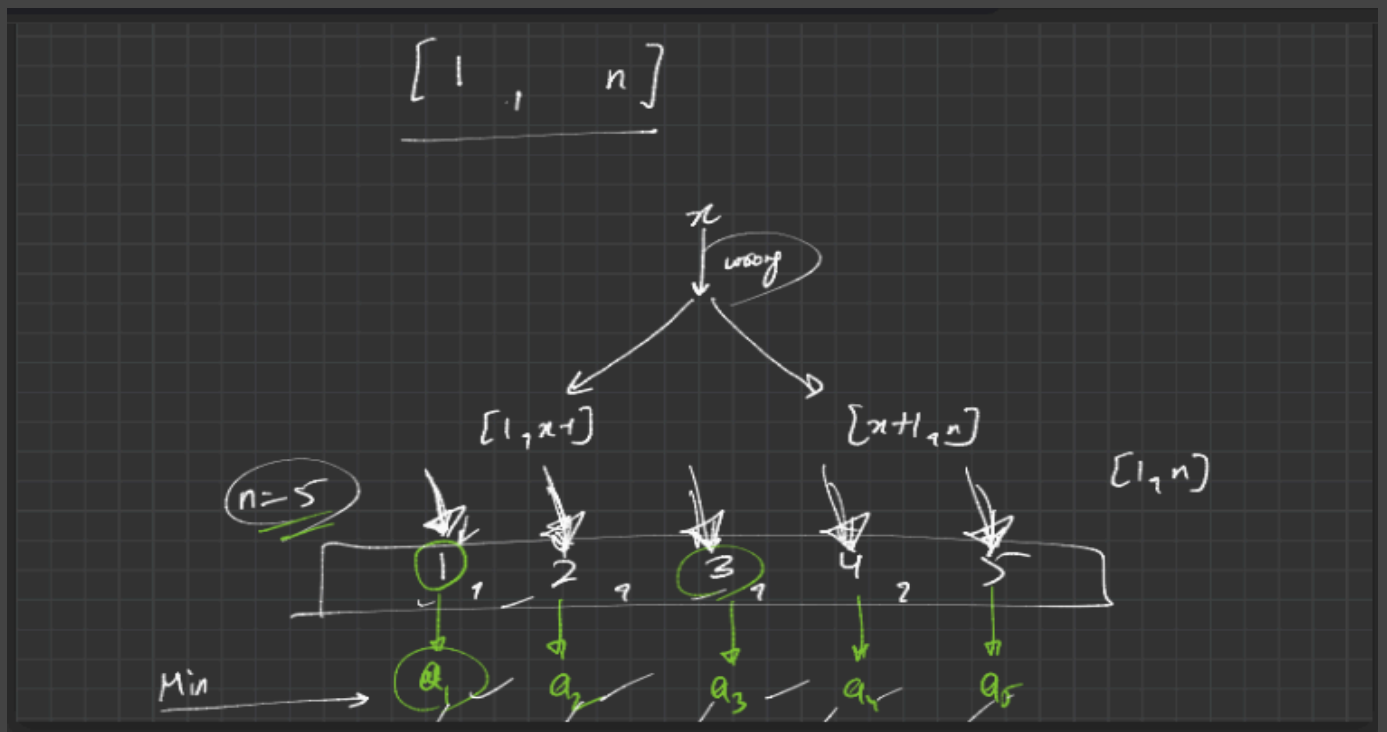

n=3
1
2
3
f(0)
0 node
2 nodes
1 node
1 node
2 nodes
0 nodes
2 + 1 + 2
f(n) → f(3) → 5
2 nodes
2
1
2
1 node
0 node
1 node
0 nodes


n nodes
1 → Root
2 → Root
3 → Root
i → Root
n → Root
i
(i-1) nodes
n-i nodes
f(n) = f(i-1) * f(n-i)
for all i's



Guess Number Higher Or Lower By Recursion / Top-Down Approach / Bottom-Up Approach:

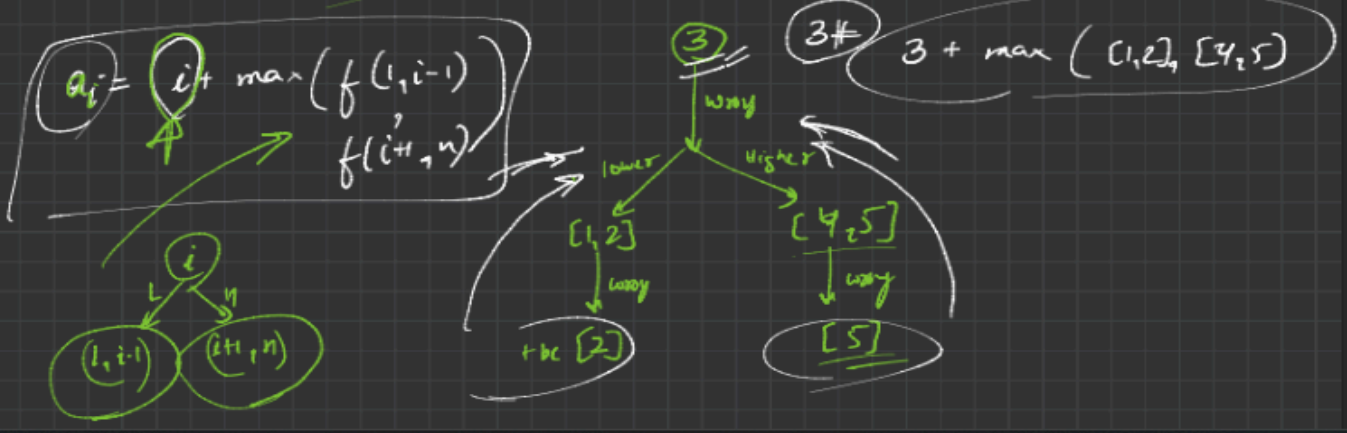


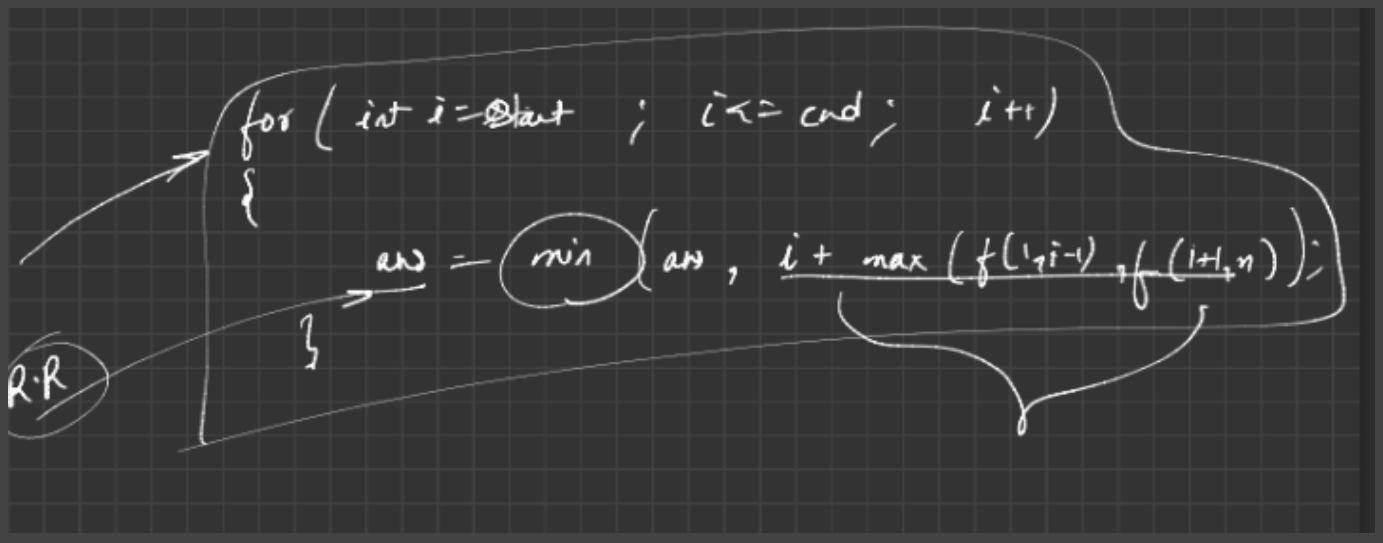


ans = min(a1, a2, a3, a4, a5);

$a_i \rightarrow ?$

a_i = i + max(f(1,i-1), f(i+1,n))





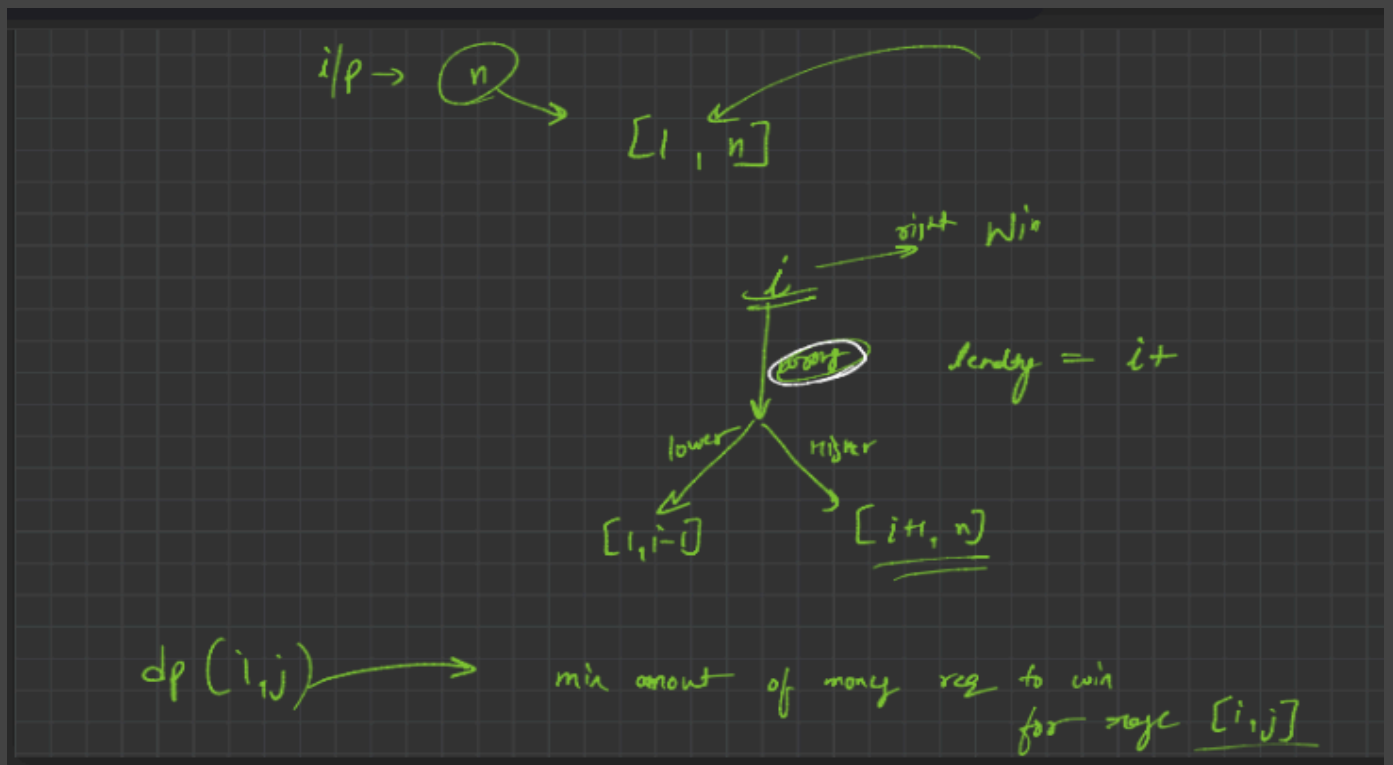
Top - Down

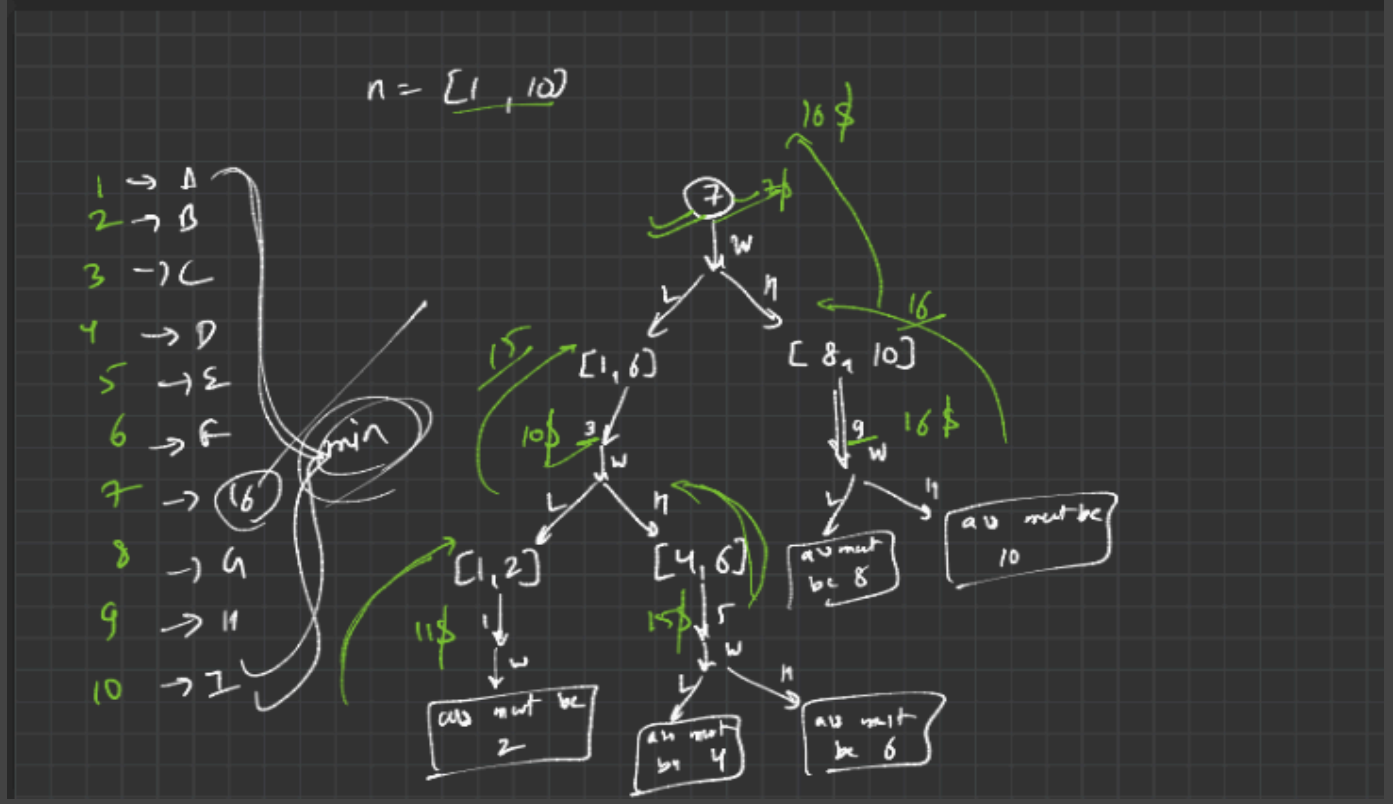
↳ start → 1 ⟶ n

↳ end → n ⟶ 1

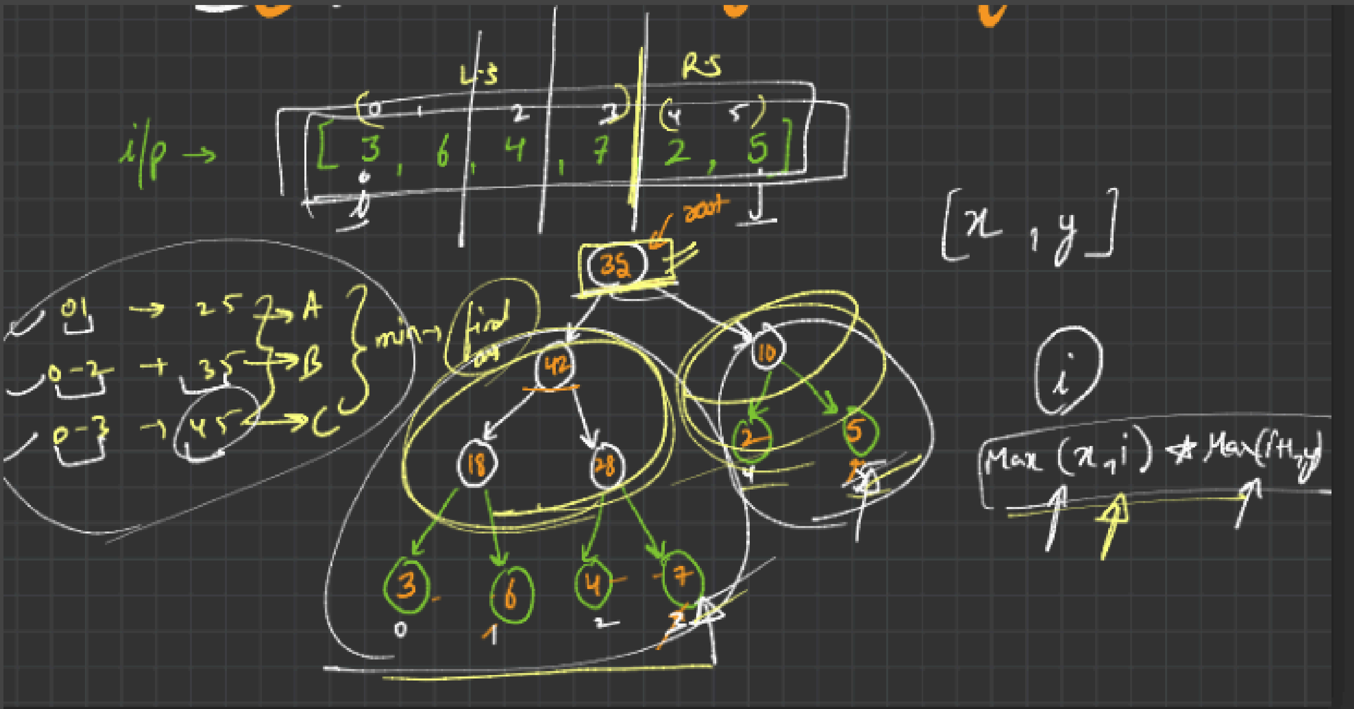
S.O

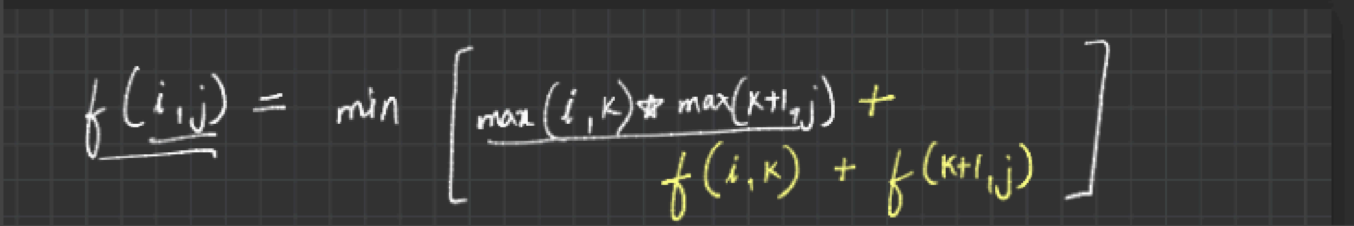
Bottom Up

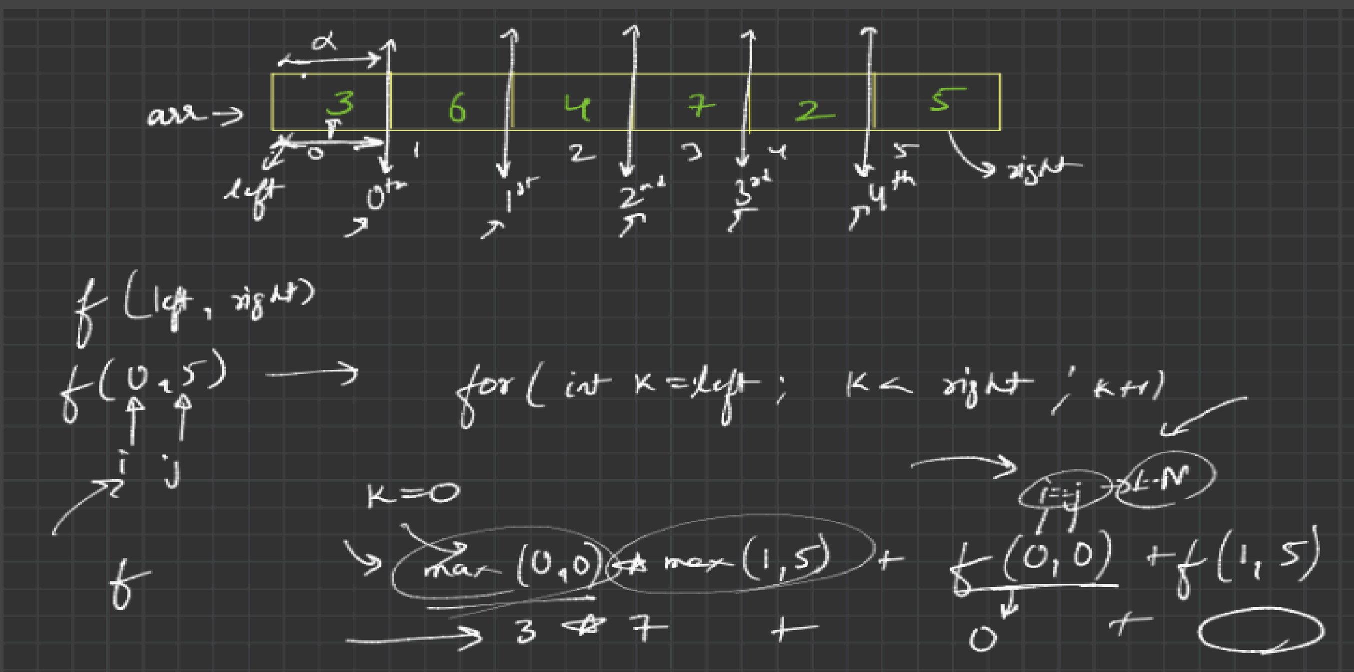


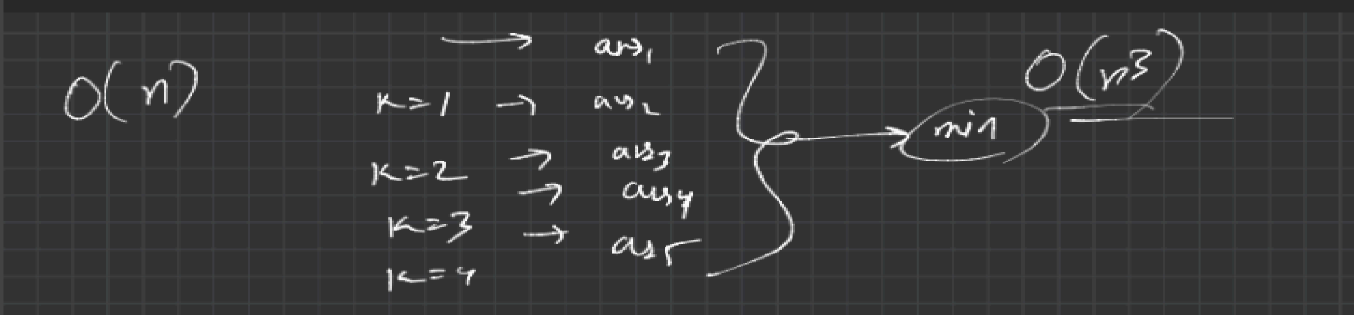


Minimum Cost Tree From Leaf Values By Recursion / Top-Down Approach:

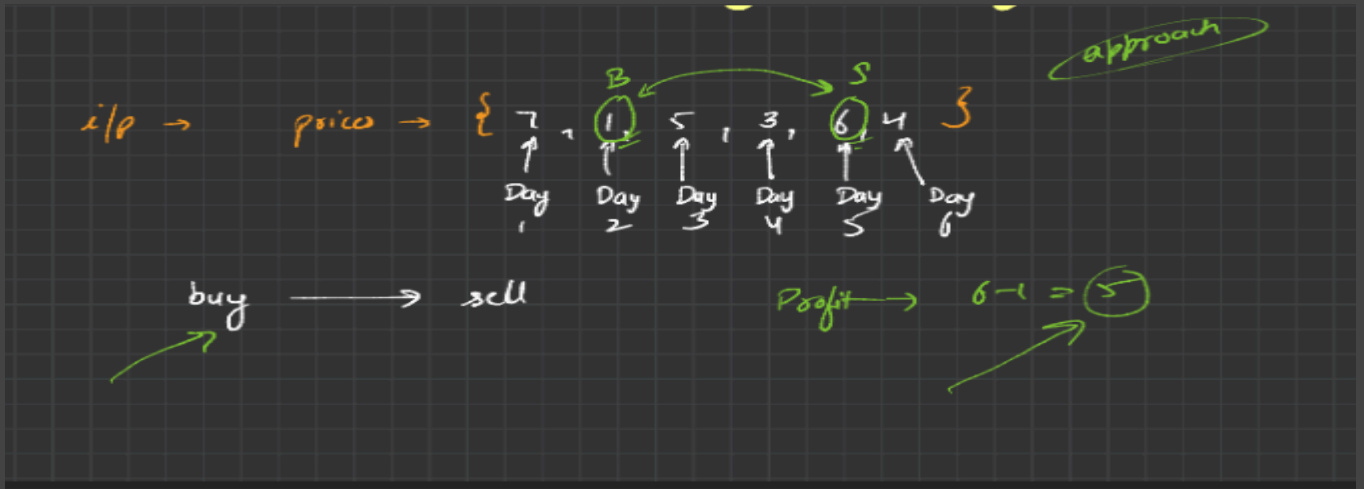


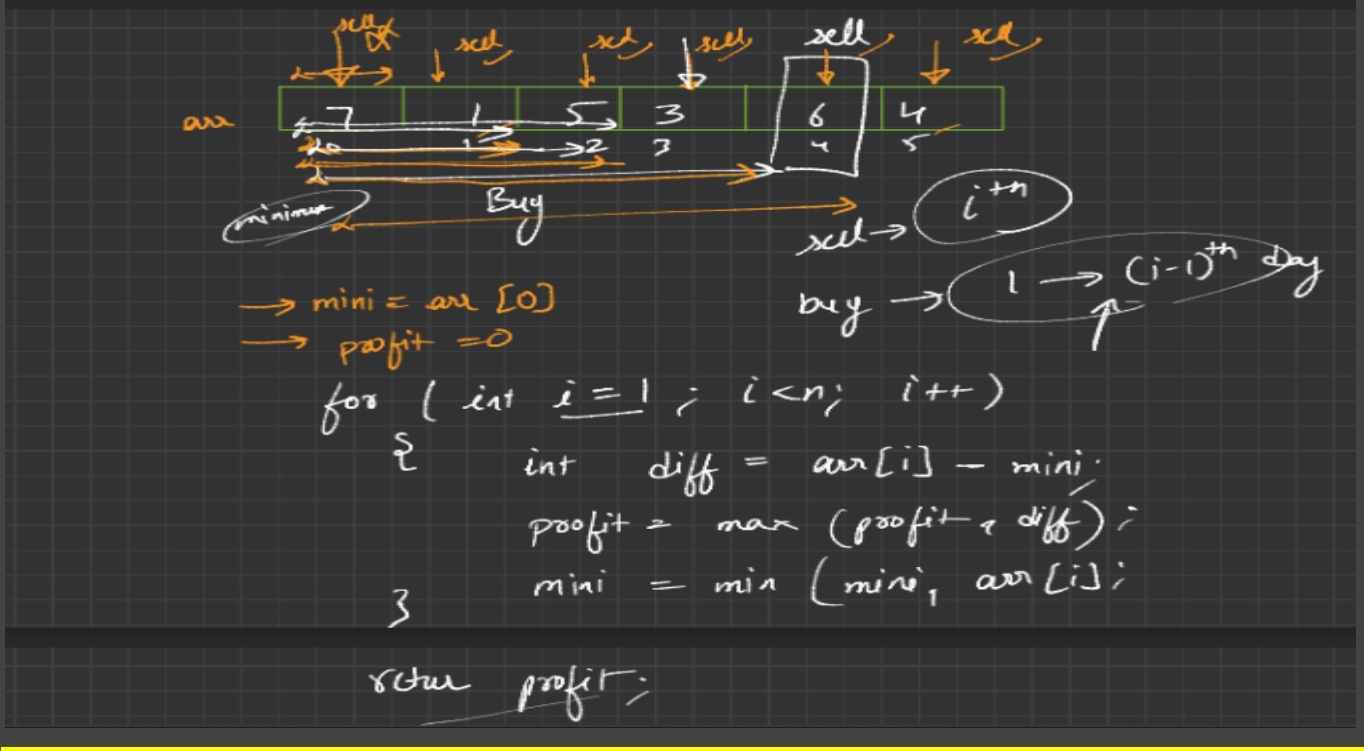




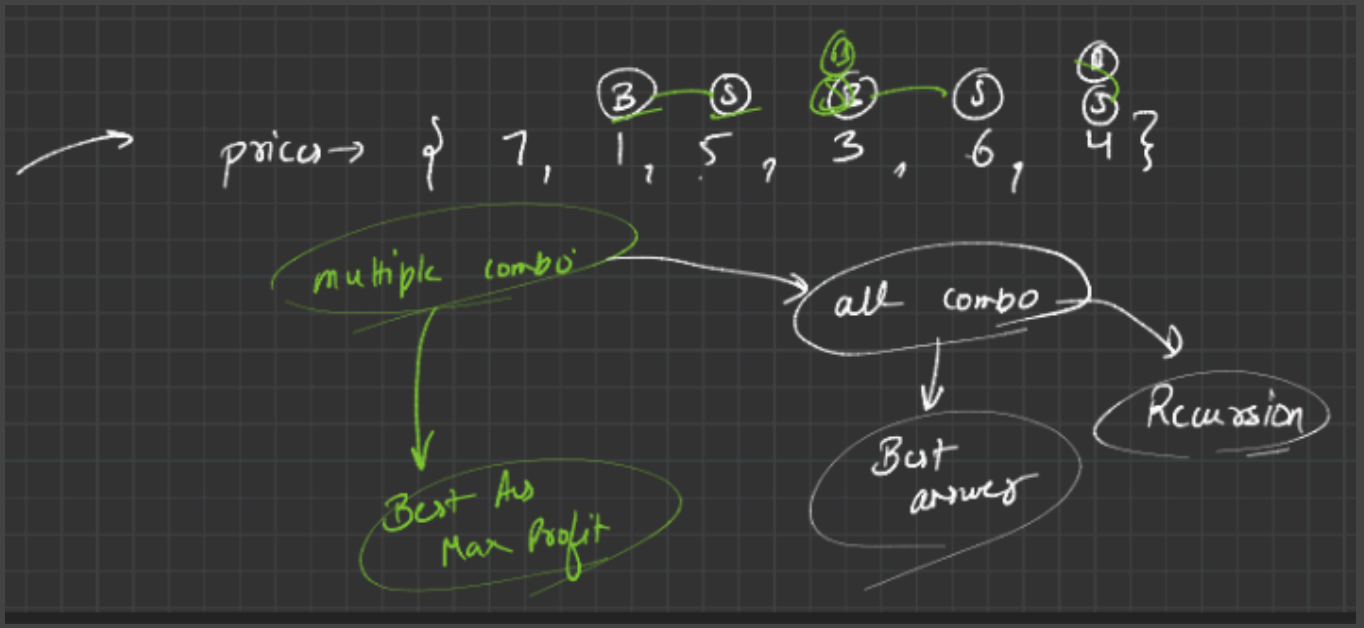


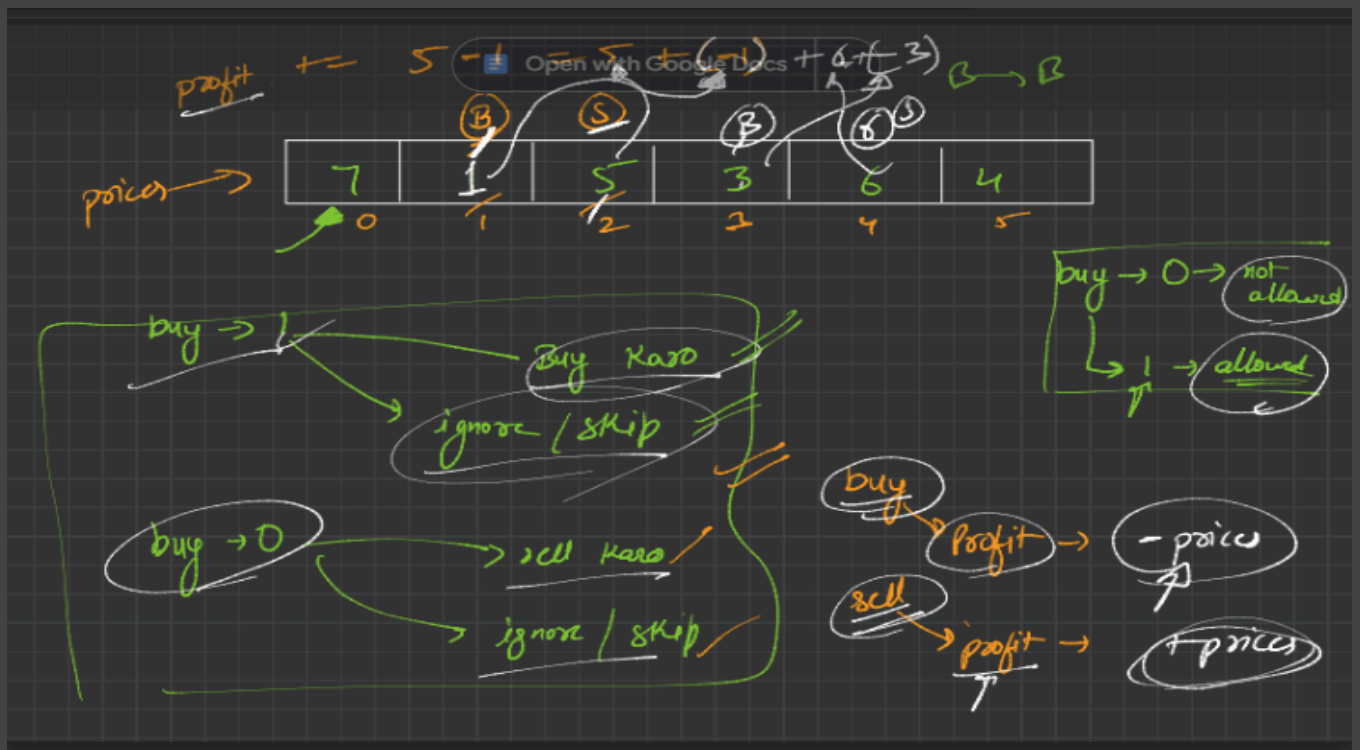
Buy And Sell Part-I:

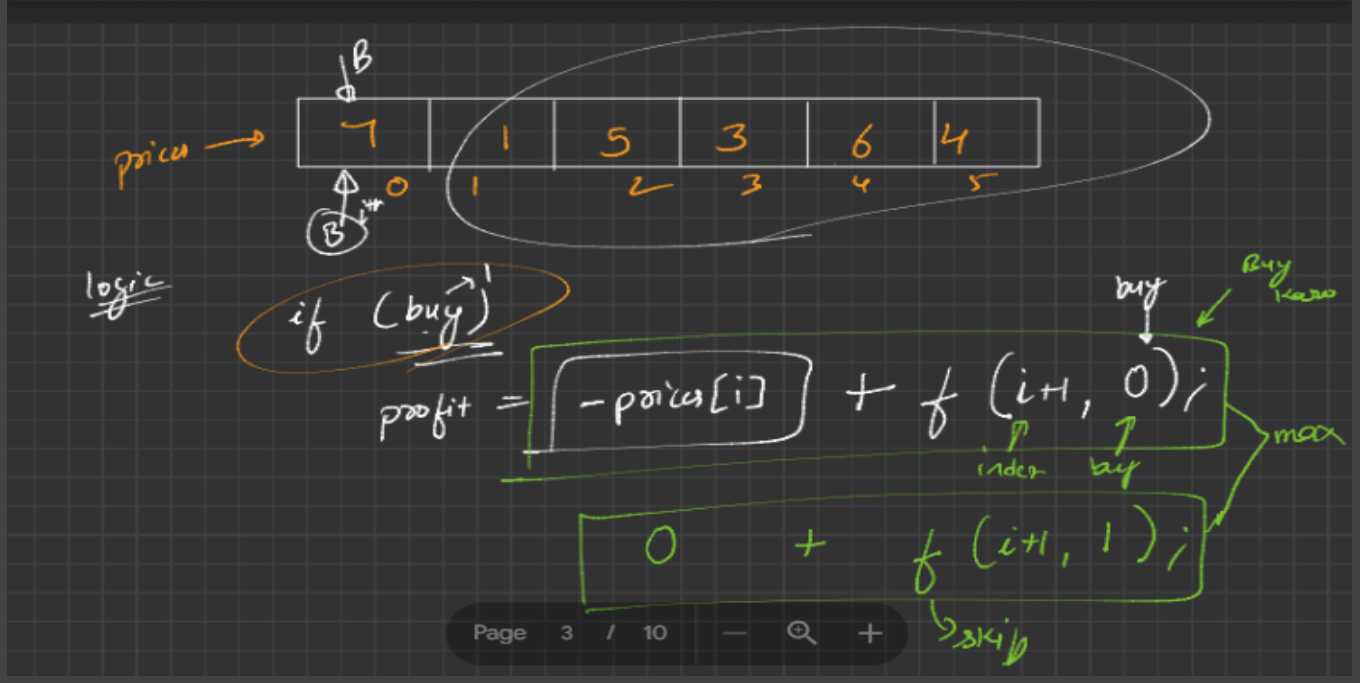


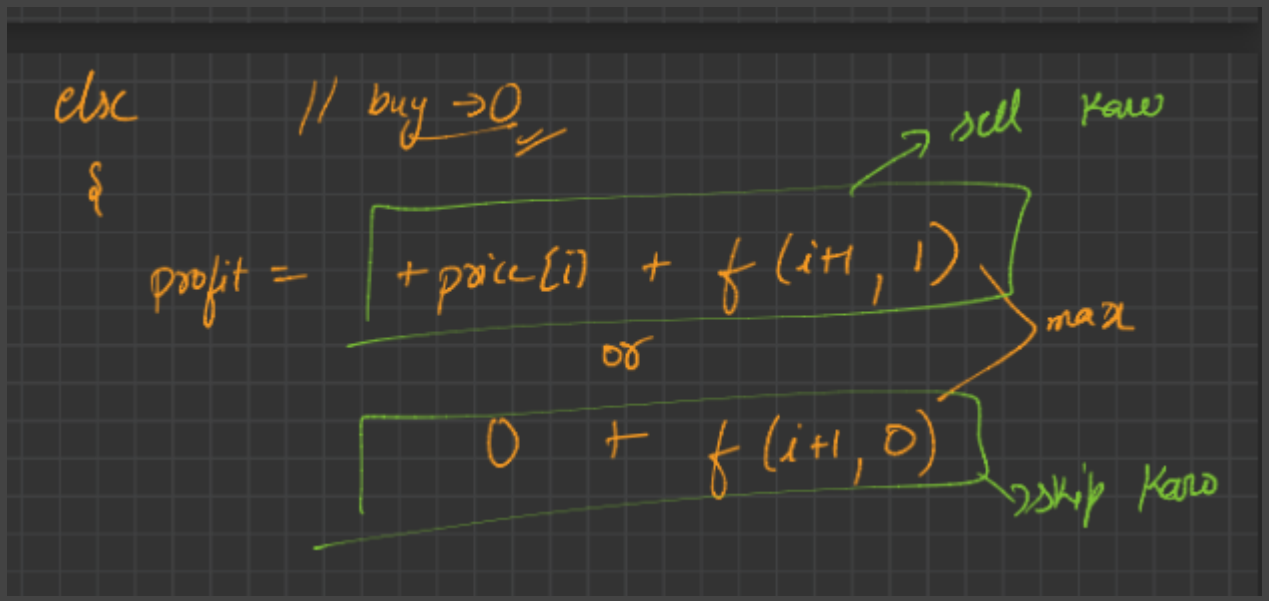


Buy And Sell Part-II By Recursion / Top-Down Approach / Bottom-Up Approach / Space Optimization Approach:




profit += 5
prices
7
1
5
3
6
4
buy → 1
Buy Karo
ignore / skip
buy → 0
sell karo
ignore / skip
buy → 0 → not allowed
1 → allowed
buy → profit → -price
sell → profit → +price


price
7
1
5
3
6
4
logic
if (buy)
profit = -price[i] + f(i+1, 0);
index
buy
Buy Karo
max
0 + f(i+1, 1);
skip
Page 3 / 10


else // buy → 0
{
profit = +price[i] + f(i+1, 1)
sell Karo
or
0 + f(i+1, 0)
skip Karo
max

Open with Google Docs

+ Mem

① dp array create
func call add

② ans return
↳ dp me store

③ B.C → K base
dp check karo → answer already present
↓
return karo

Top - Down

index → 0 ⟶ n

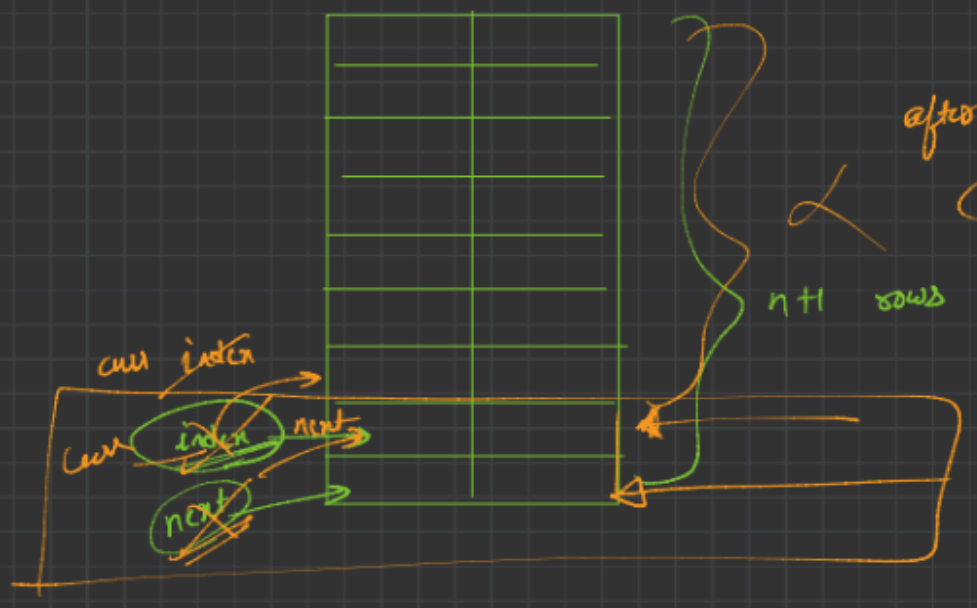
Bottom - Up

n-1 ——— 0

S.O → ? O(n) → O(1)

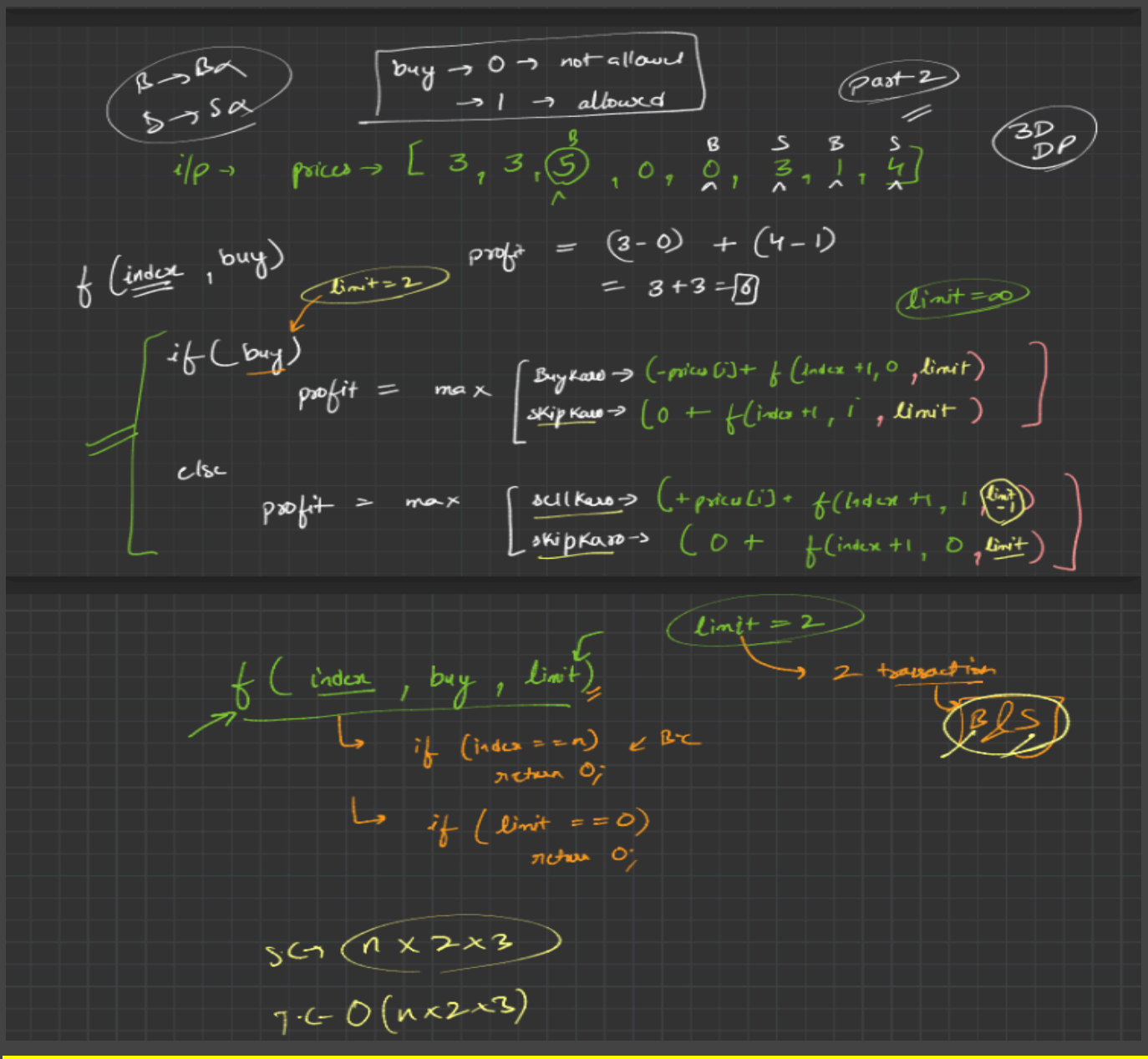
Page 7 / 10

dp [index] [buy]

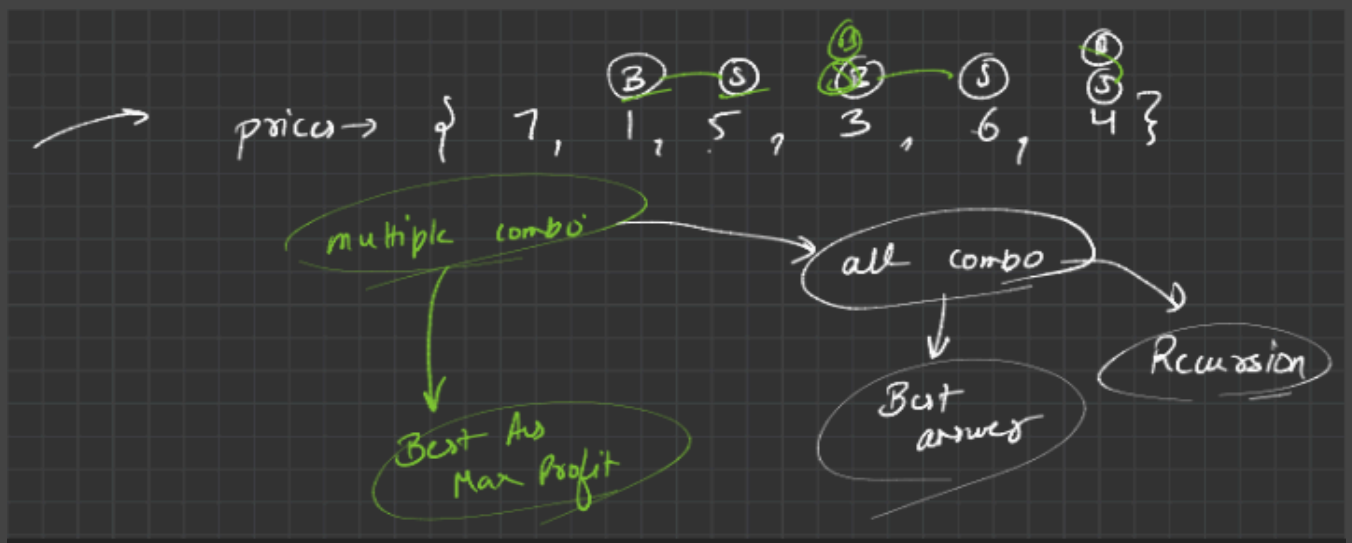


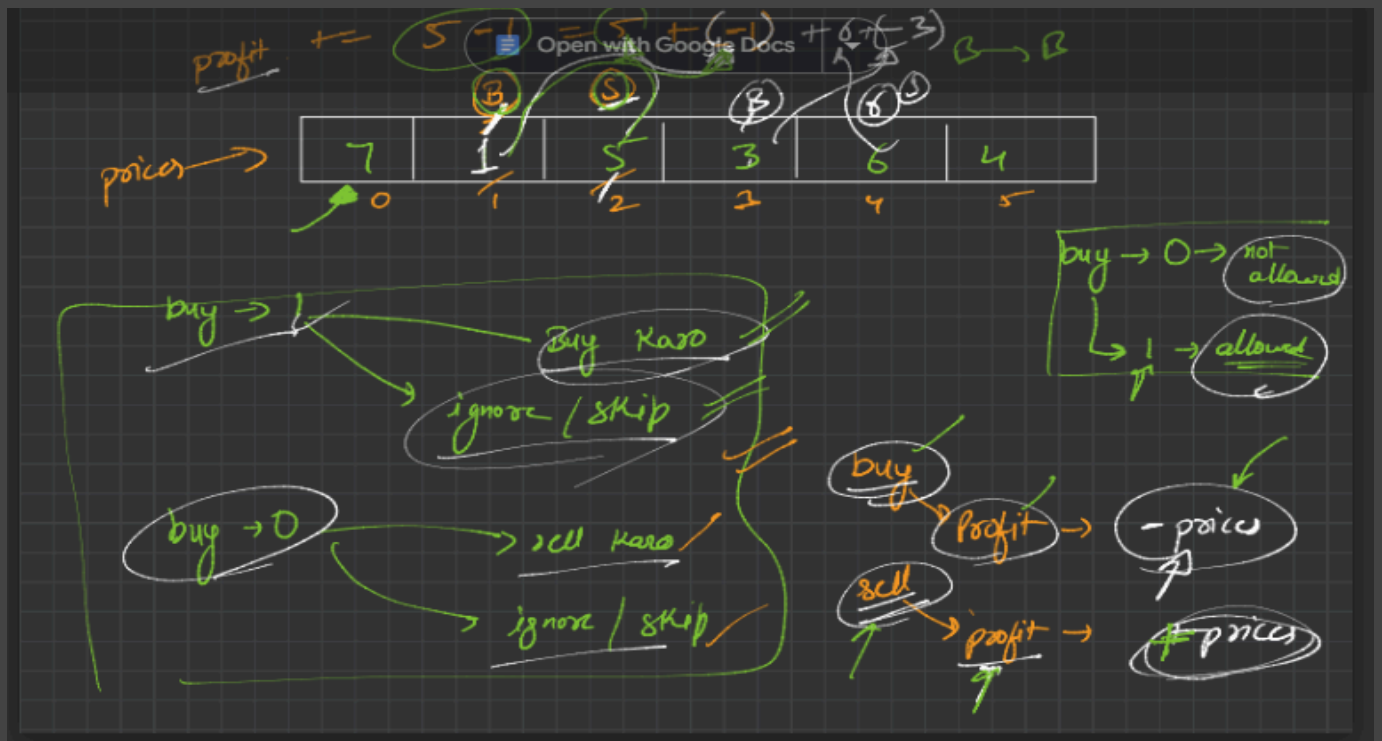
after each iteration
next = index
curr

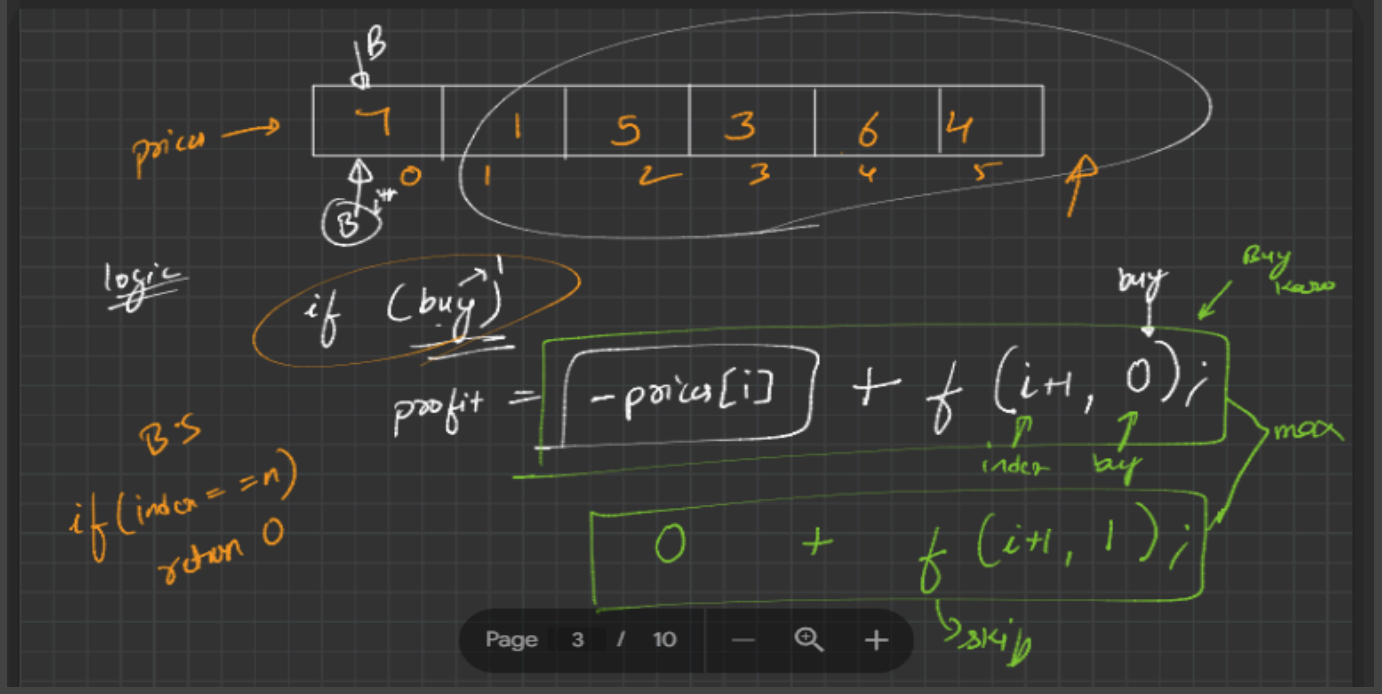
Buy And Sell Part-III By Recursion / Top-Down Approach / Bottom-Up Approach / Space Optimization Approach:

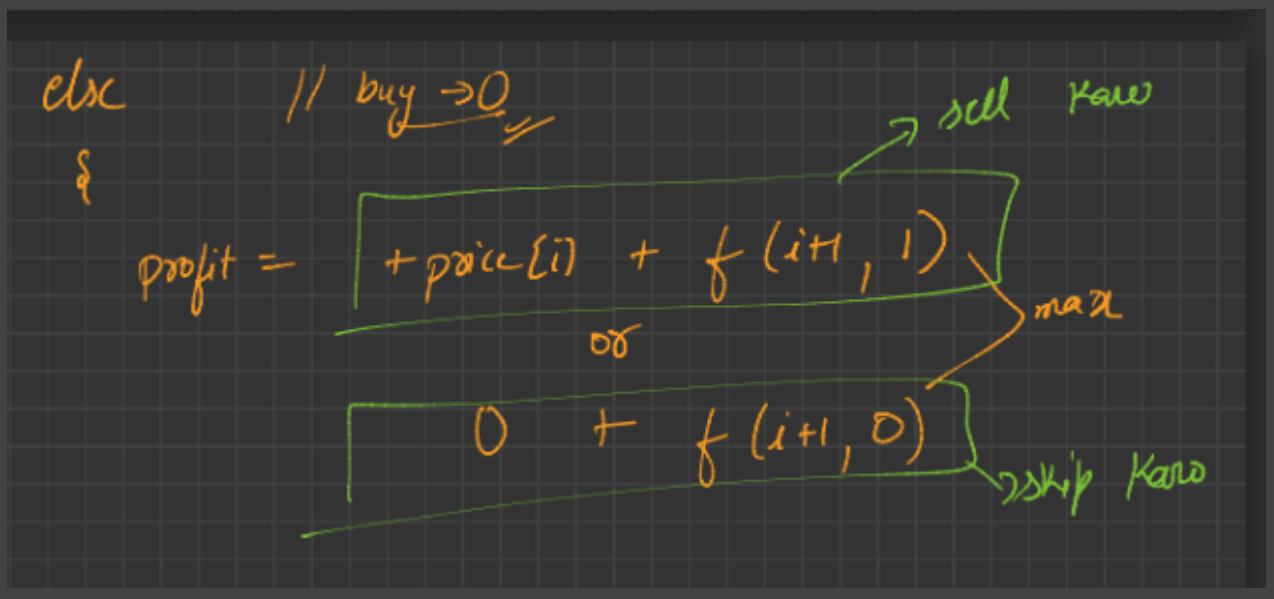


Buy And Sell Part-IV By Recursion / Top-Down Approach / Bottom-Up Approach / Space Optimization By Previous Approach / Space Optimization Approach:




Open with Google Docs
profit += 5 - 1 = 5 + (-1) + 6 + (-3)
B → B
prices → 7 1 5 3 6 4
0 1 2 3 4 5
buy → 1
Buy Karo
ignore / skip
buy → 0
sell karo
ignore / skip
buy → 0 → not allowed
1 → allowed
buy
Profit → -price
sell
profit → +prices


prices → 7 1 5 3 6 4
0 1 2 3 4 5
logic
if (buy)
profit = -prices[i] + f(i+1, 0); → buy
0 + f(i+1, 1); → skip
max
index
buy
Buy karo
B.S
if(index == n)
return 0
Page 3 / 10


else // buy → 0
{
profit = +price[i] + f(i+1, 1) → sell karo
OR
0 + f(i+1, 0) → skip karo
max

→ + Mem

① dp array create
func calls add

② ans return
↳ dp me store

③ B.C → K back
dp check karo → answer already present
↓
return karo

Top - Down

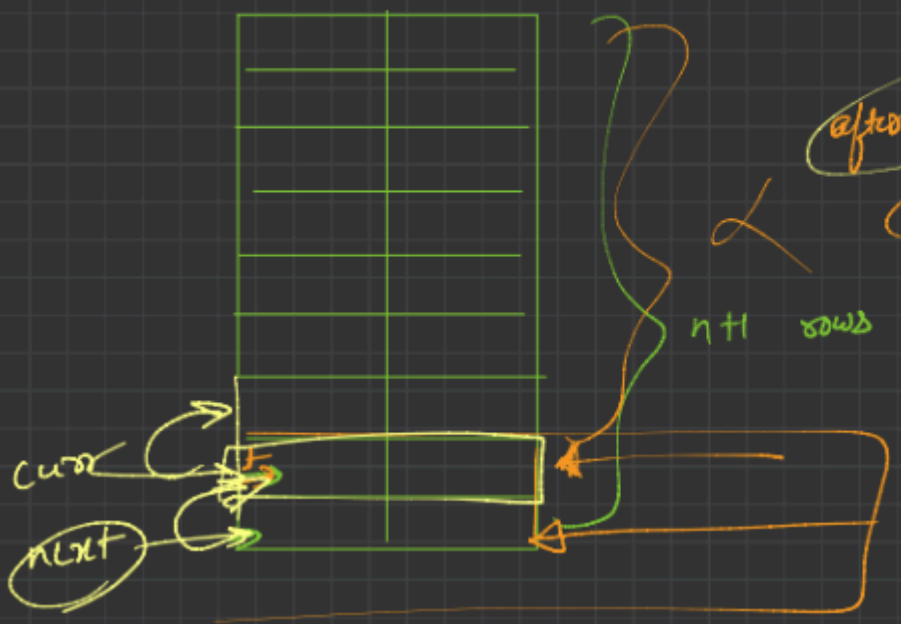
index → 0 ⟶ n

Bottom - Up

n-1 —— 0

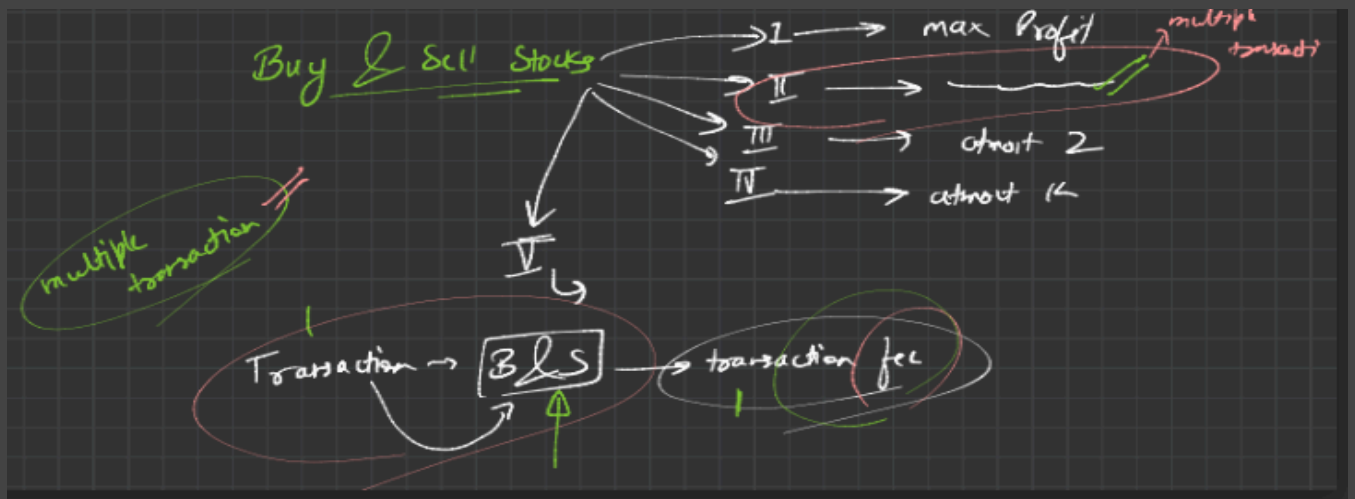
S.O → ? O(n) → O(1)=

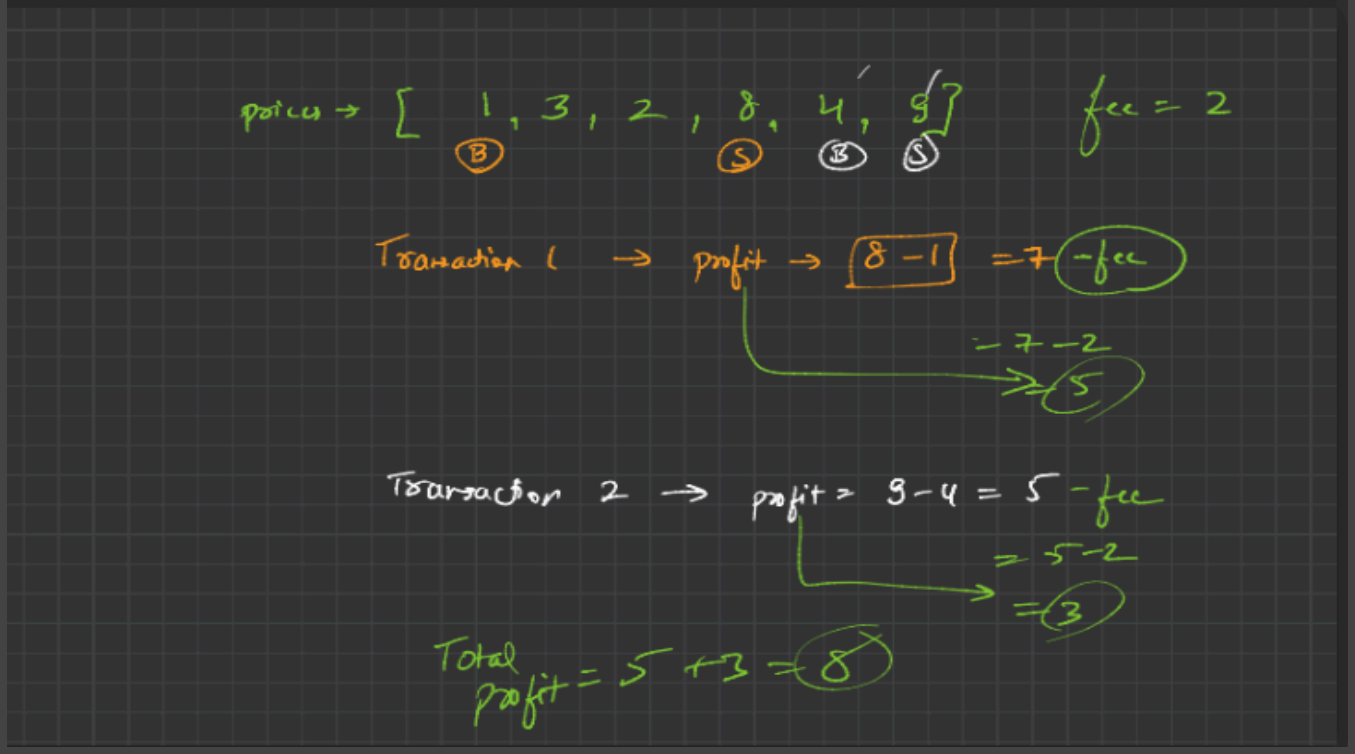
dp [index] [bag]

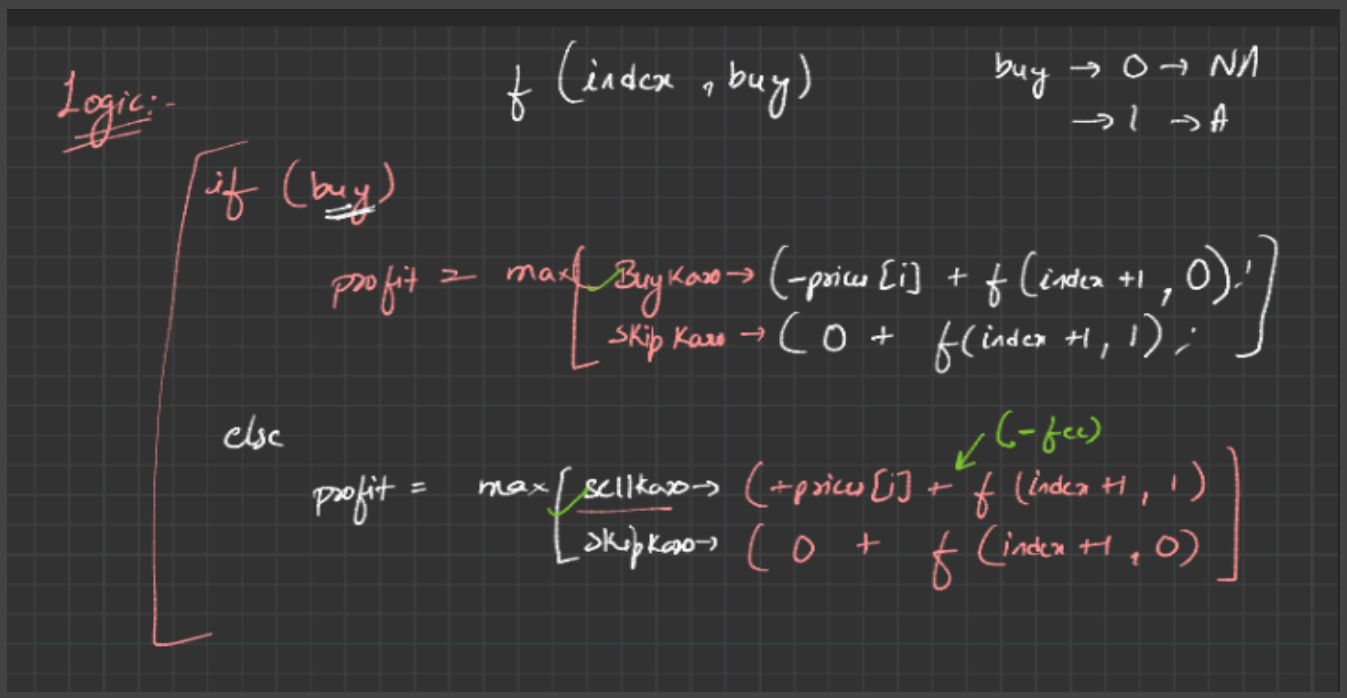


after each iteration
next = index curr

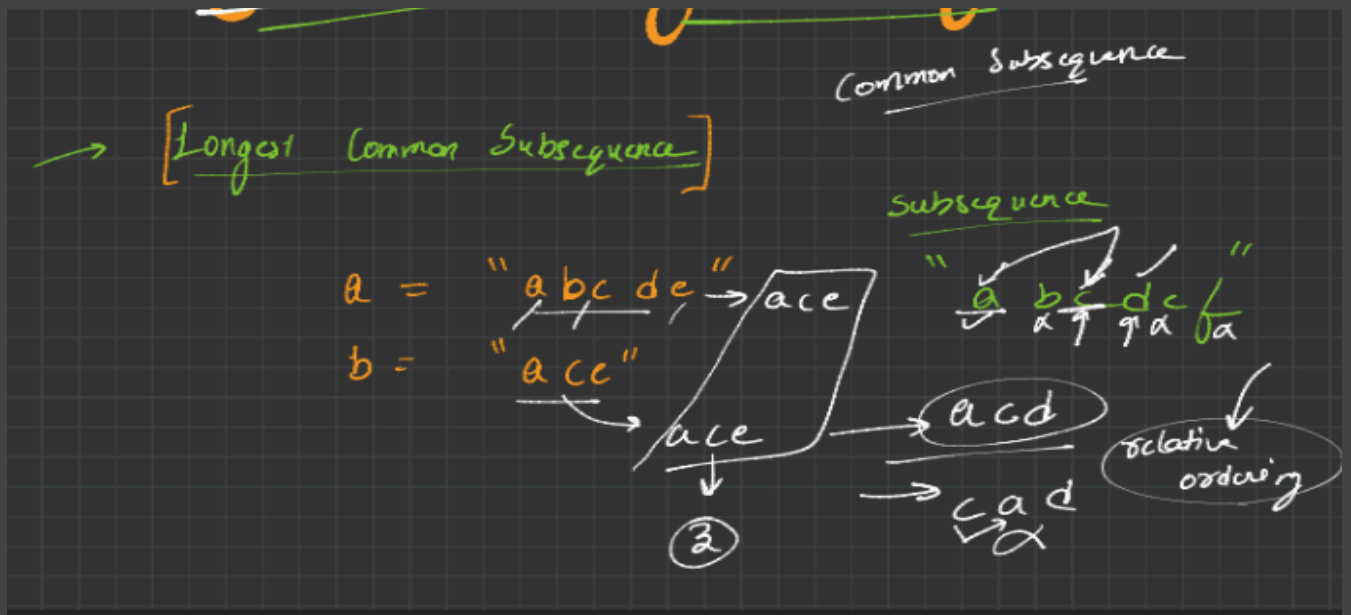
Buy And Sell With Transaction Fee:

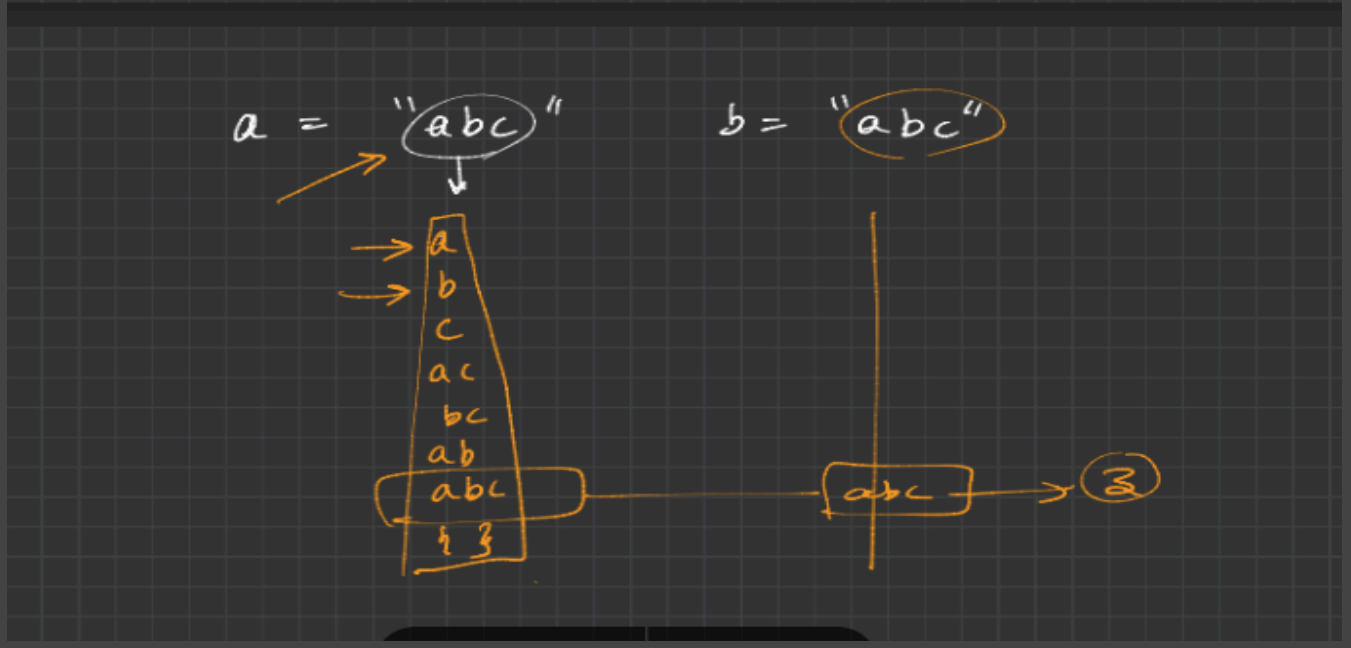


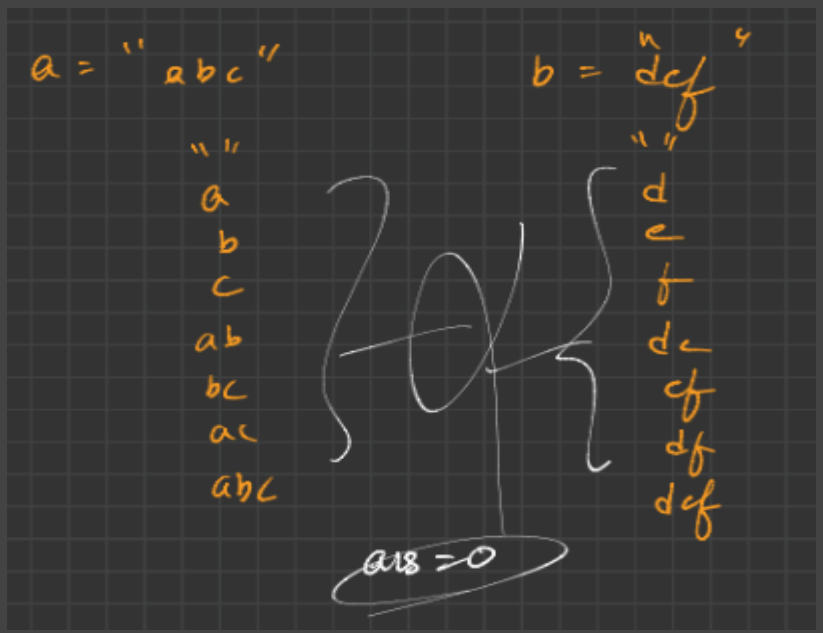




Longest Common Subsequence By Recursion / Top-Down Approach / Bottom-Up Approach / Space Optimization Approach:







a = "a b c d c"

b = "a c e"

f(a, b, i, j)

```
if (a[i] == b[j]) // match
    return 1 + f(a, b, i+1, j+1)
else
    return max [ f(a, b, i, j+1)
                 f(a, b, i+1, j) ];
```

Top-Down

i → 0 → n — a's left

j → 0 → m → b's left

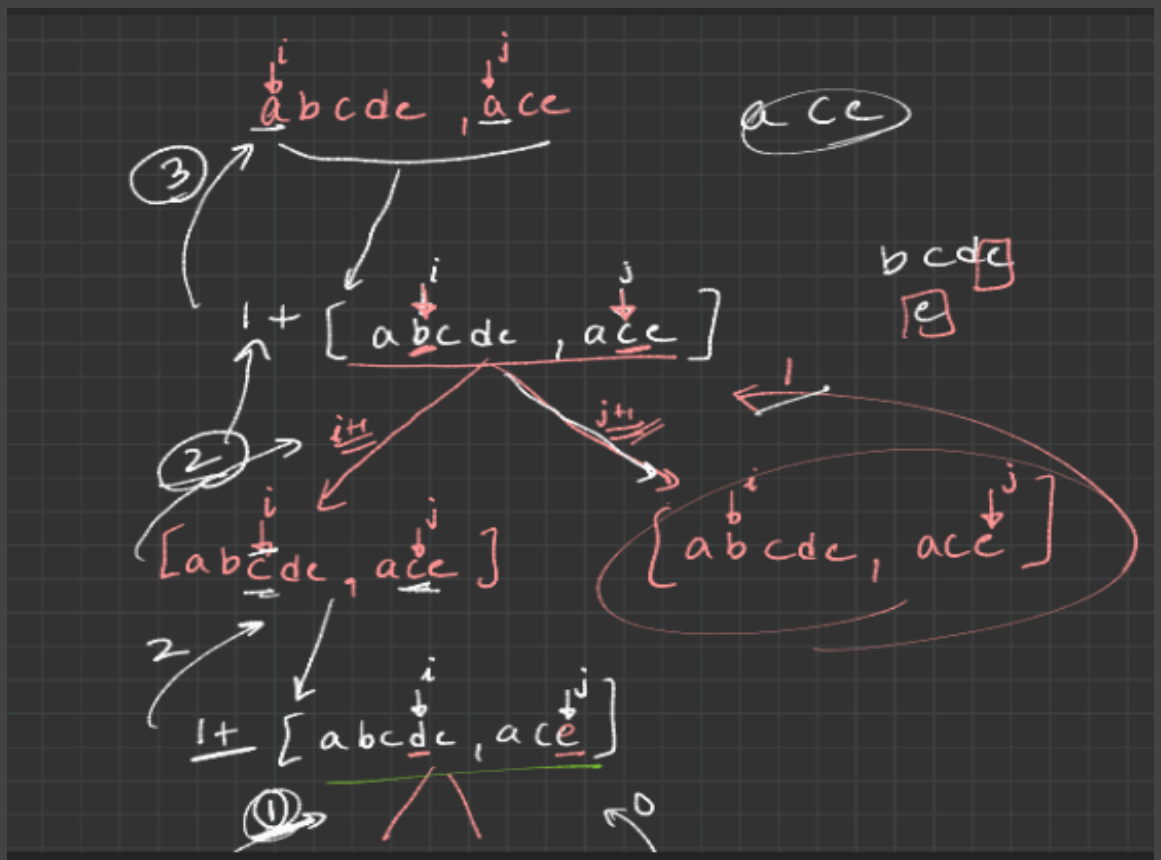
curr

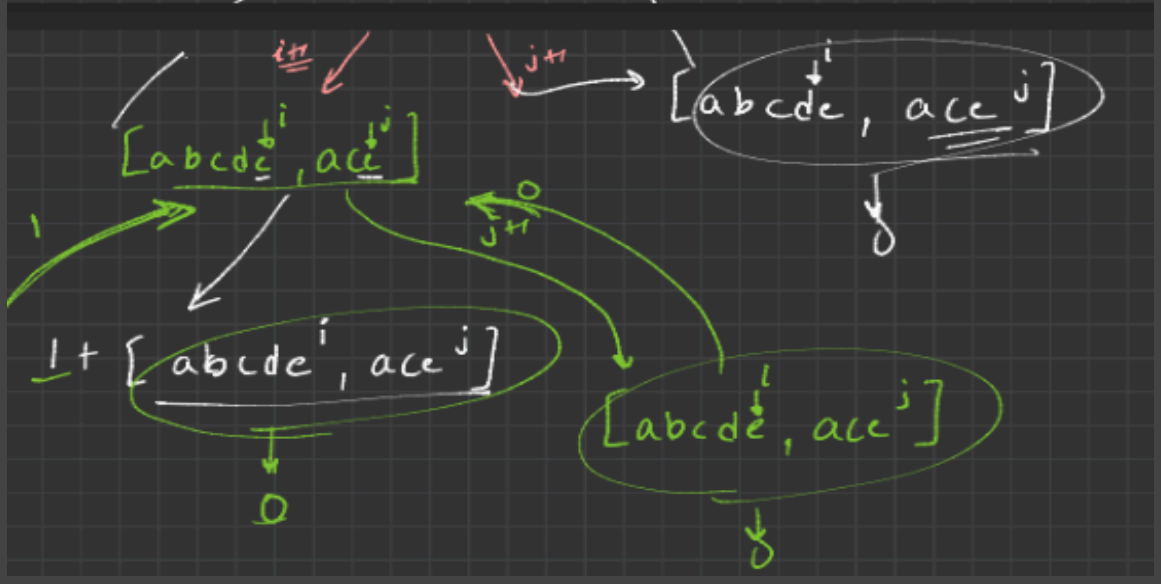
next

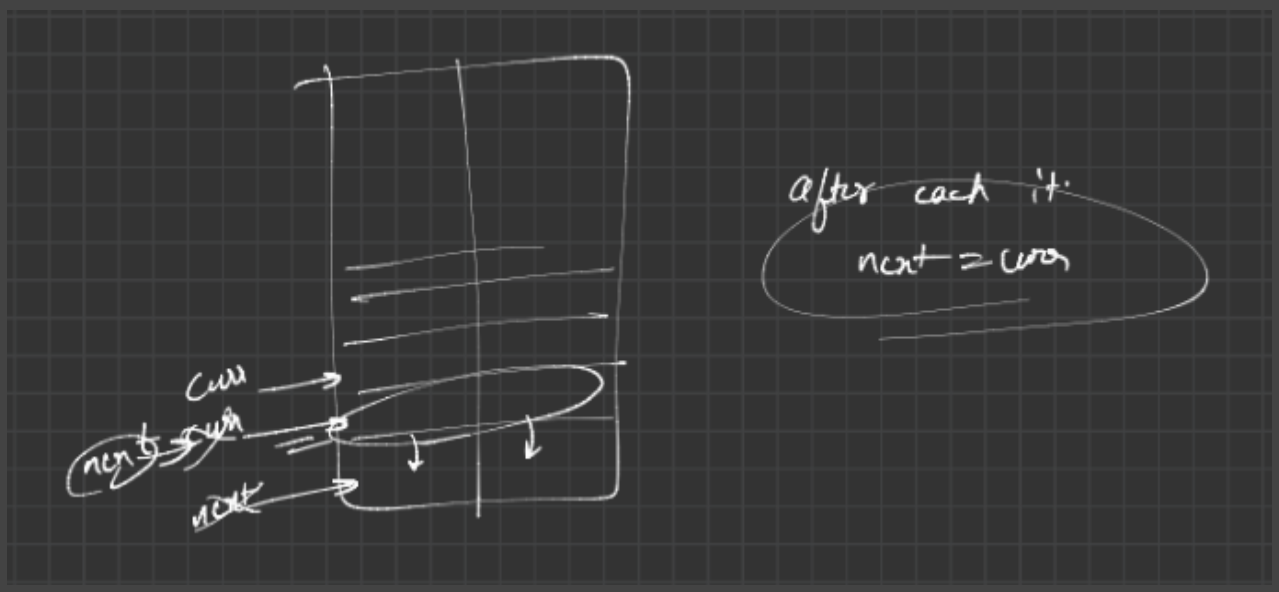
B.U

i → n → 0

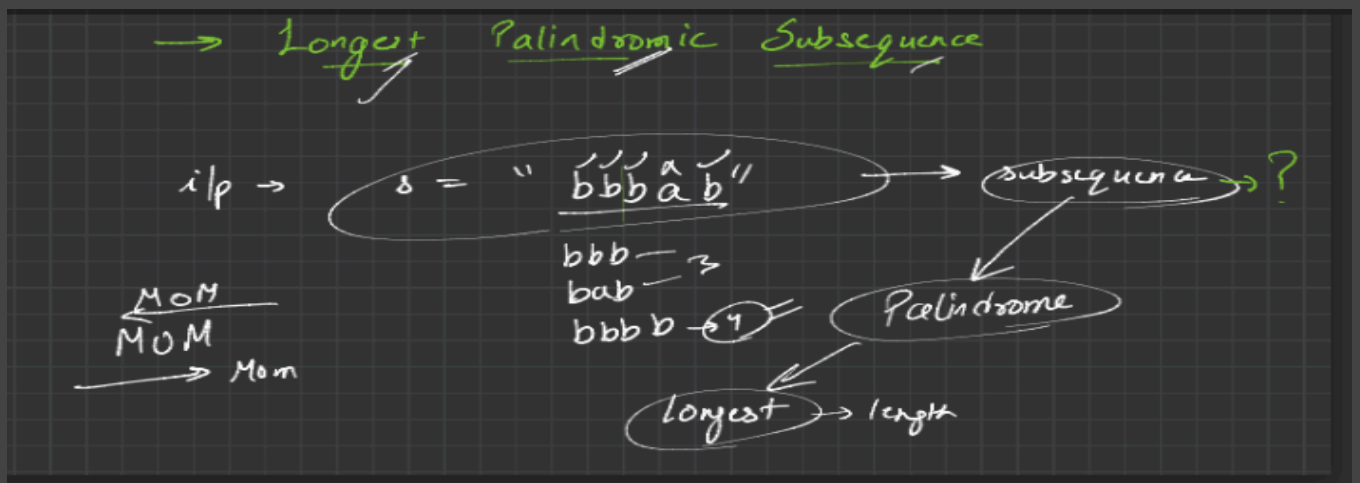
j → m → 0

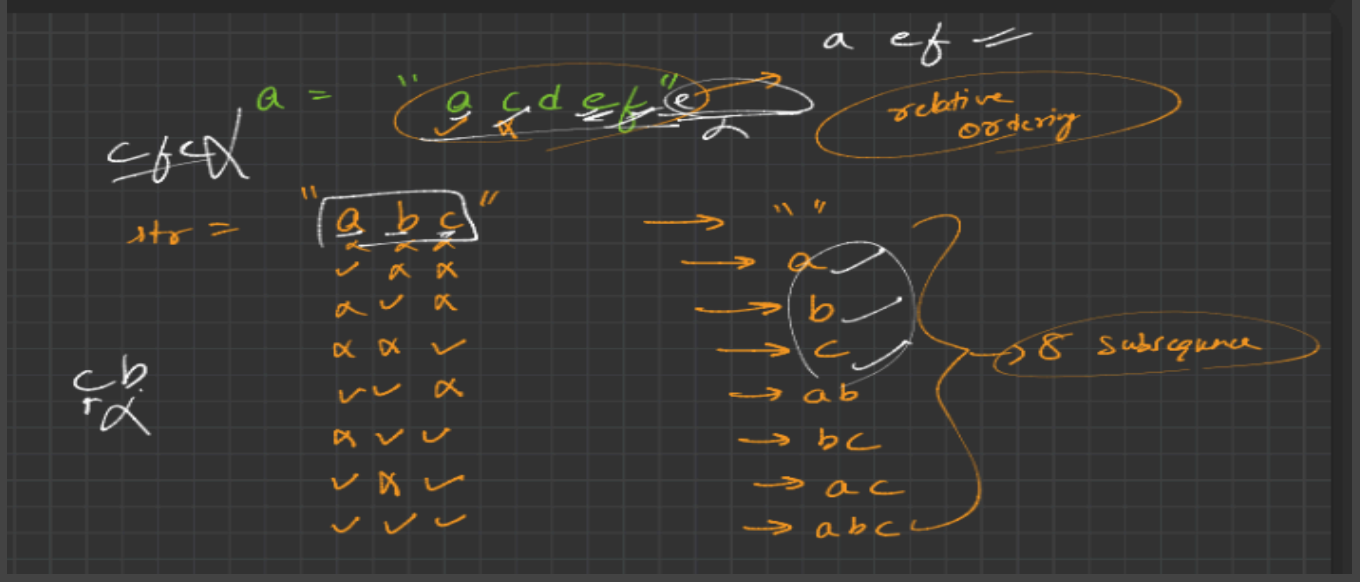

abcde, ace
ace
bcdg
e
1+ [abcde, ace]
i+
j+
1
2
[abcde, ace]
[abcde, ace]
2
1+ [abcde, ace]
1
0

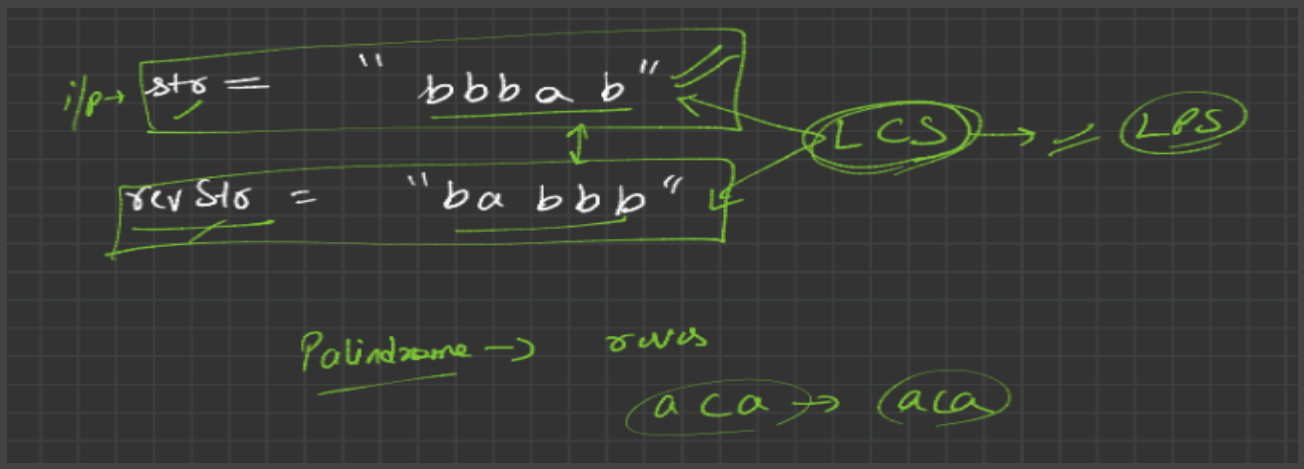

i+
j+
[abcde, ace]
[abcde, ace]
1
0
j+
1+ [abcde, ace]
[abcde, ace]
0

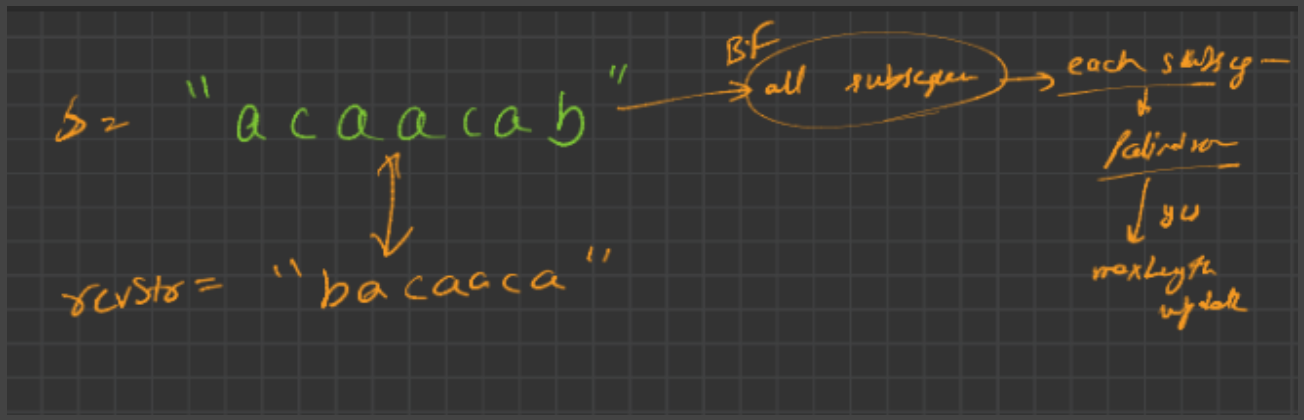

cur
next = cur
next
after each it:
next = cur

Longest Palindromic Subsequence:

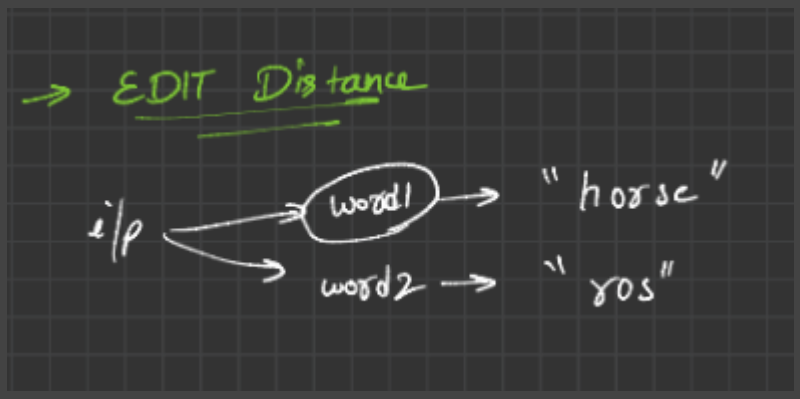


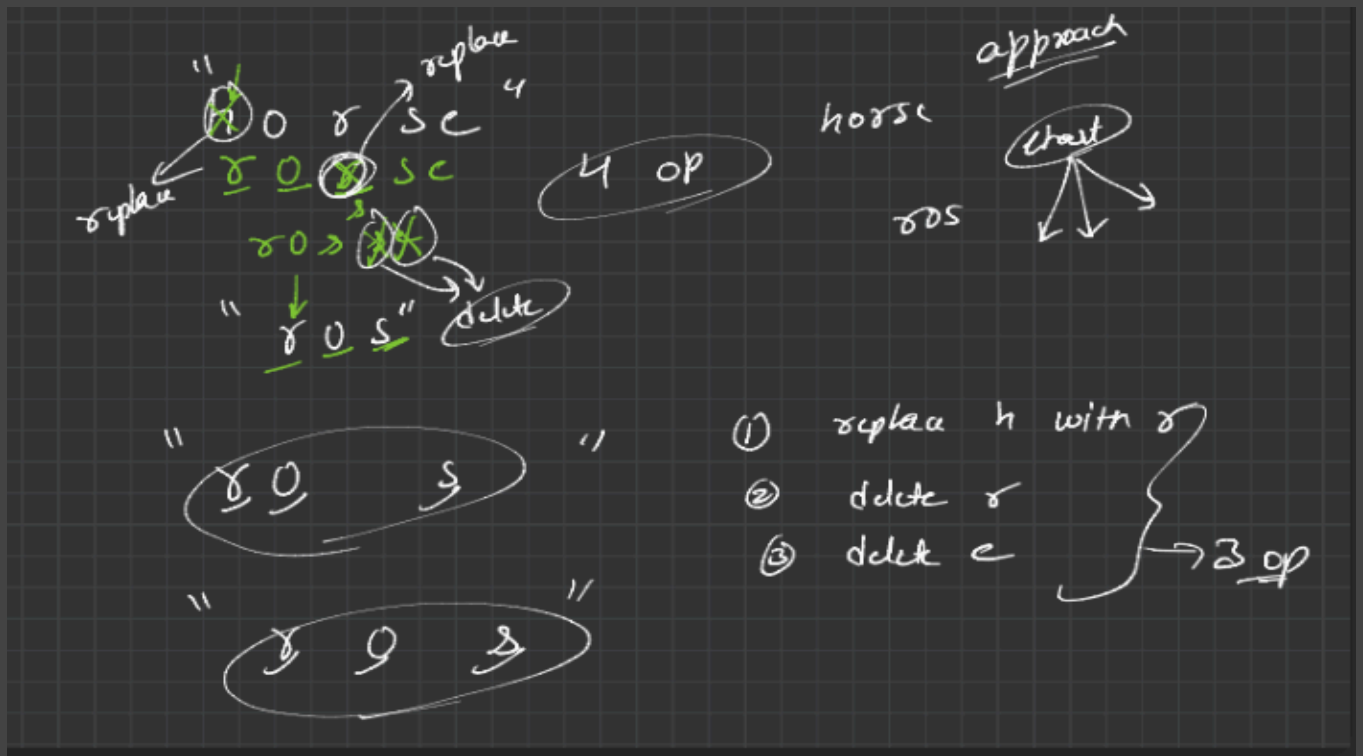


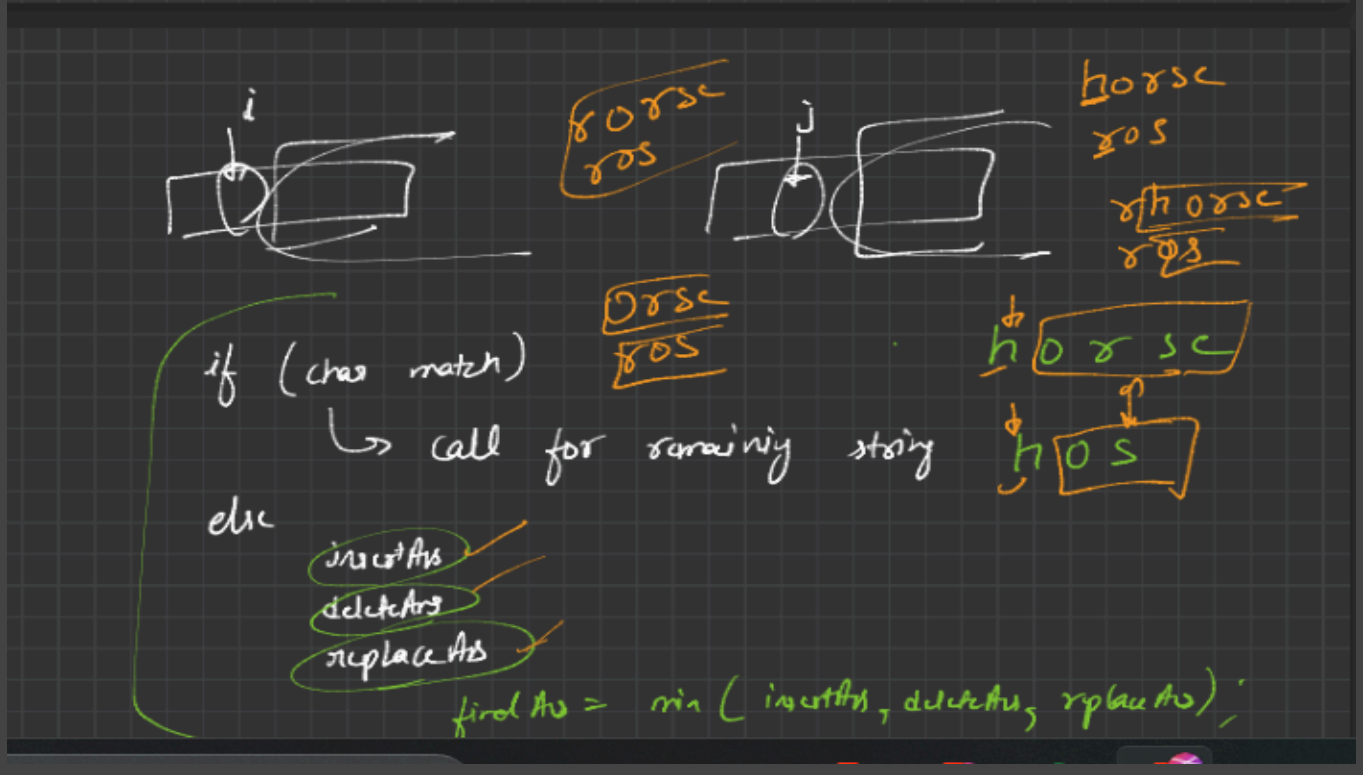




Edit Distance By Recursion / Top-Down Approach / Bottom-Up Approach / Space Optimization Approach:





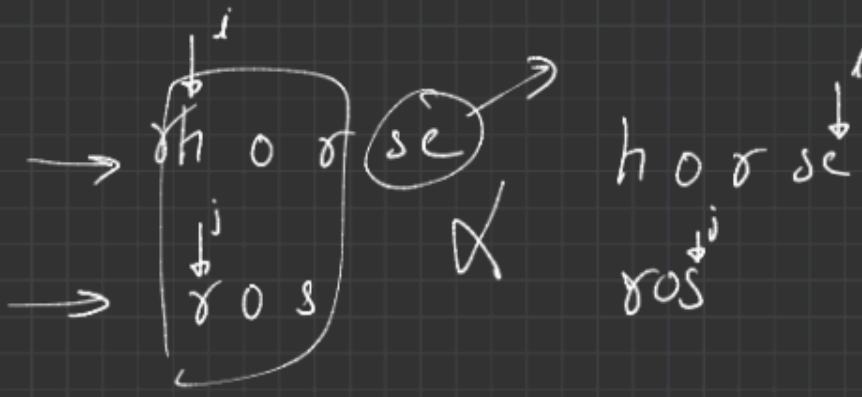


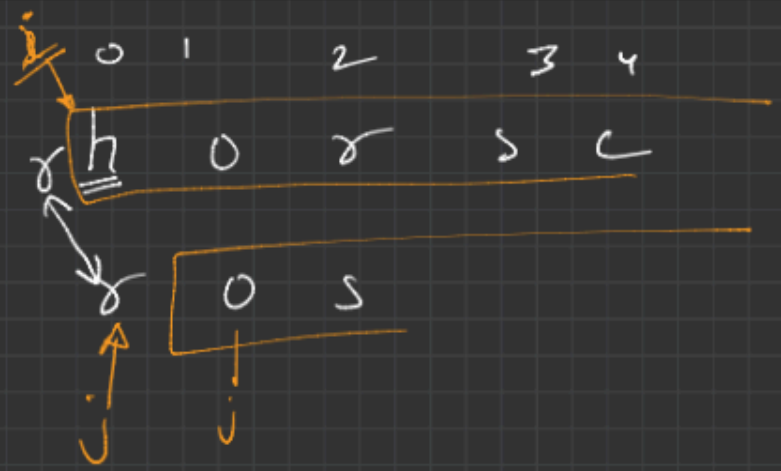
word1 → ros | hos | hor (i) sc

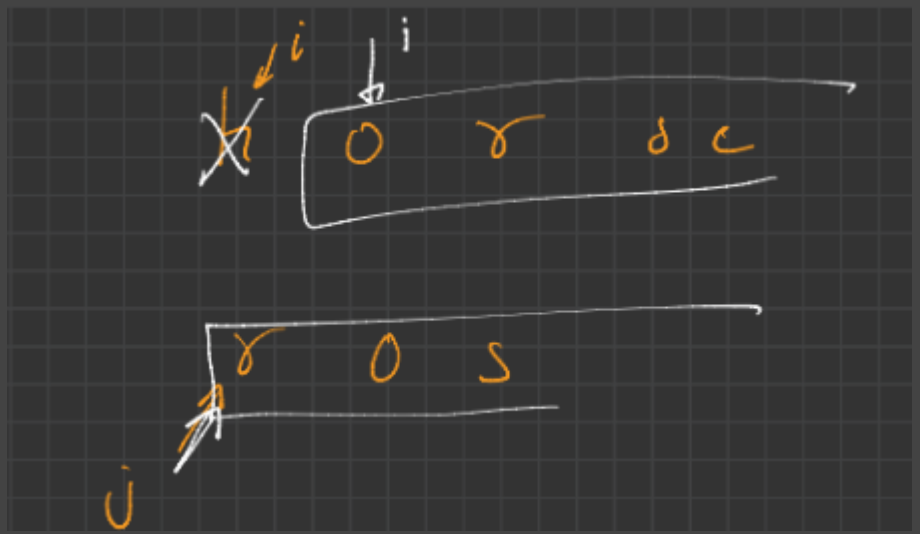
word2 → horse | horse | hors (i) e

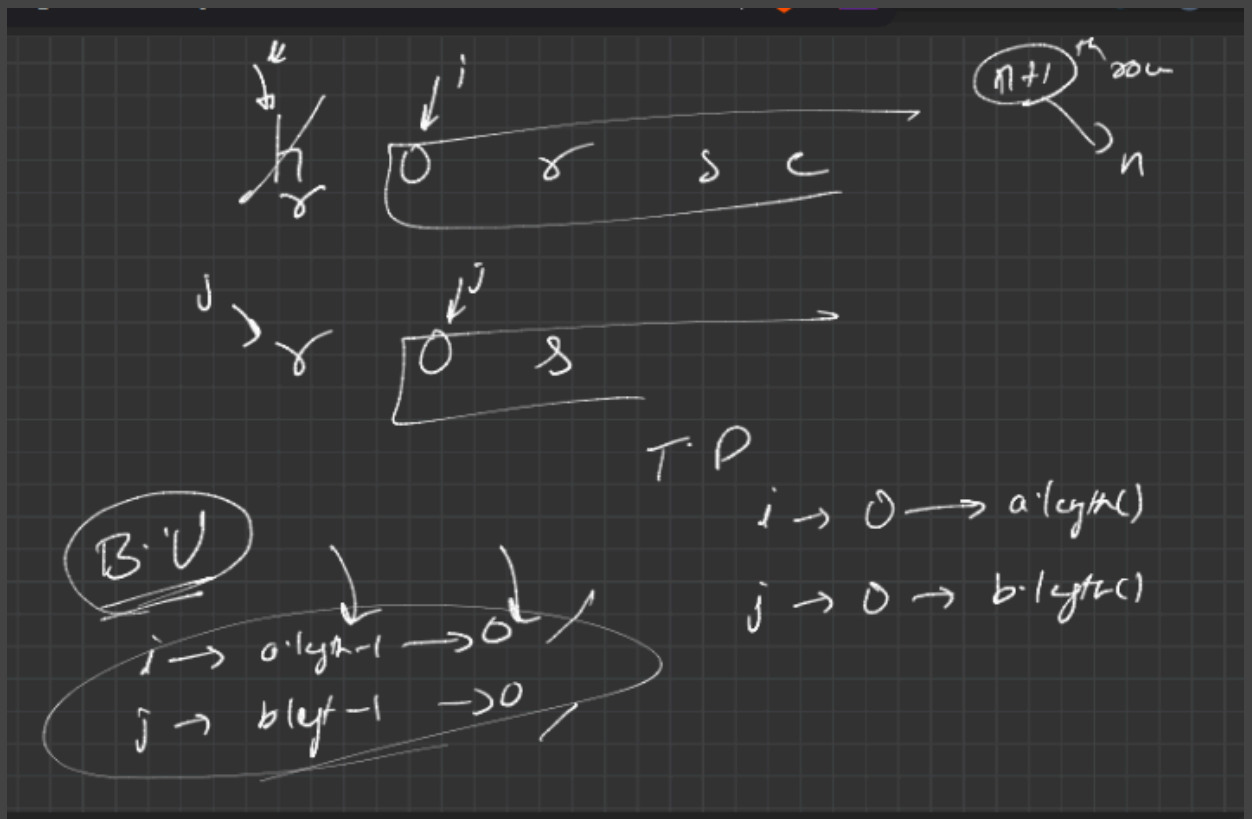
b.length() − j

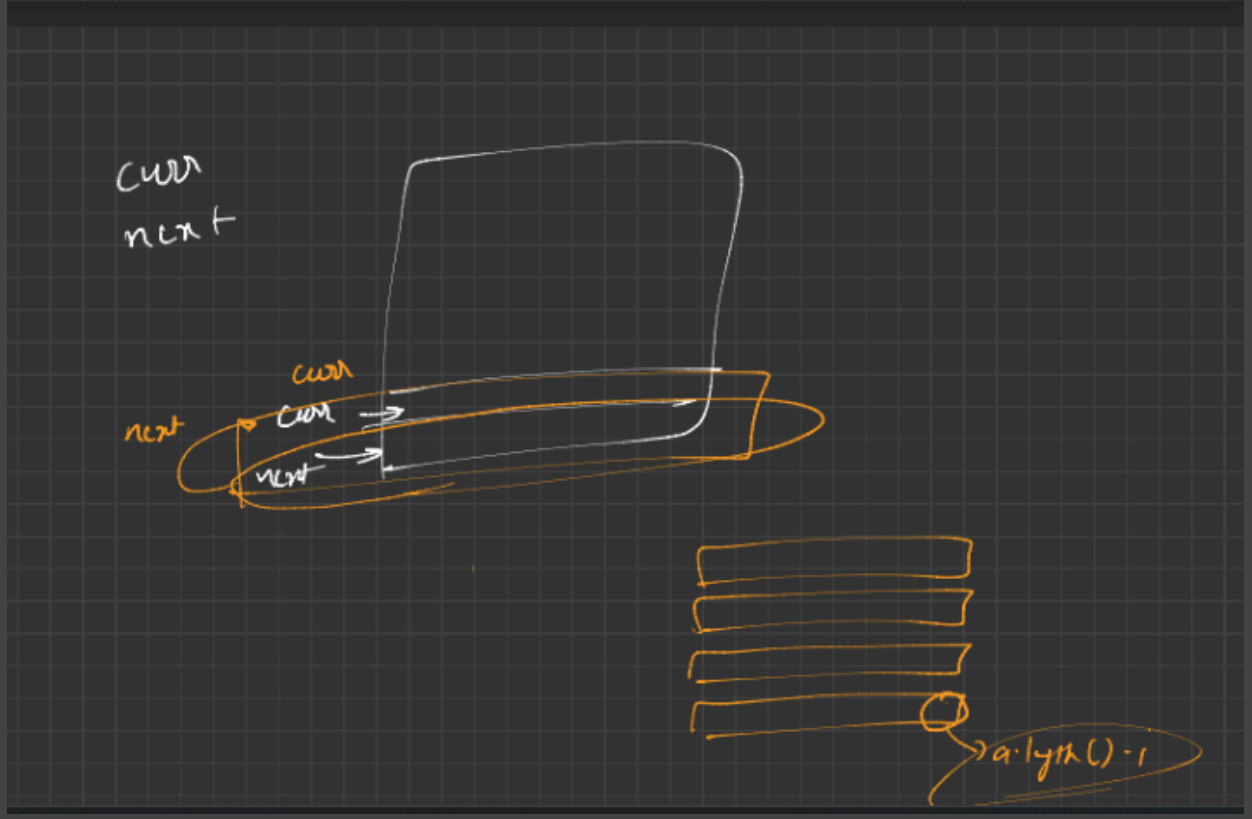
5 − 3 = 2

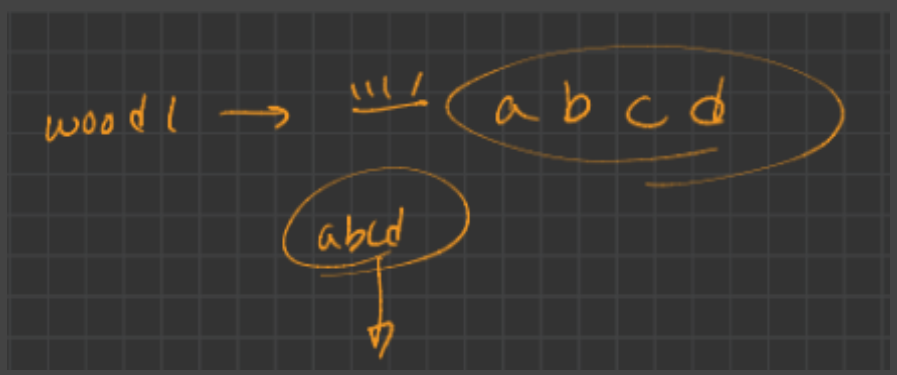




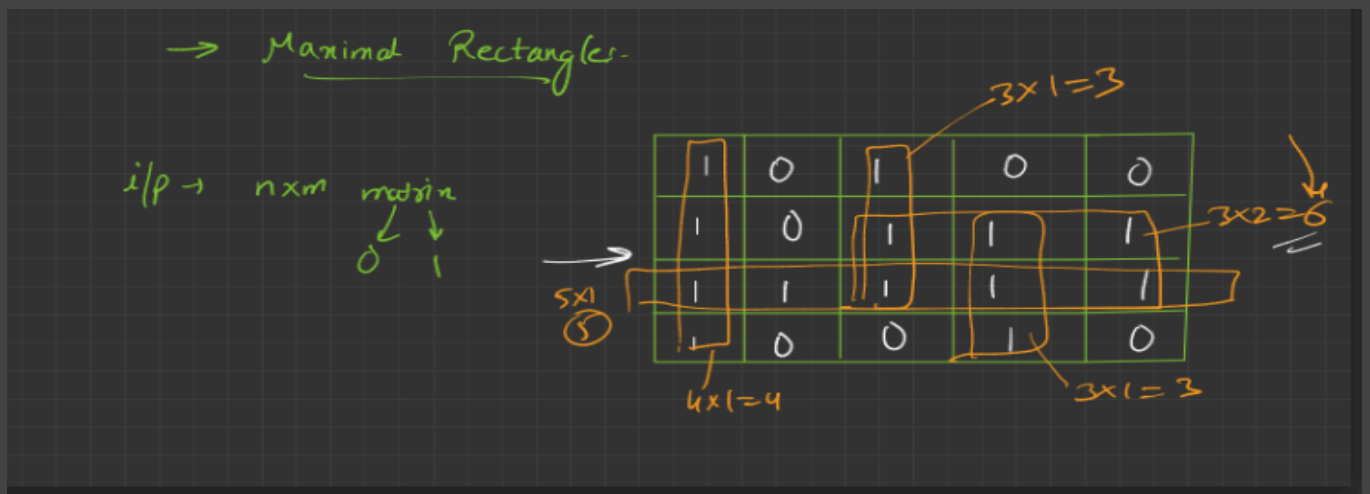


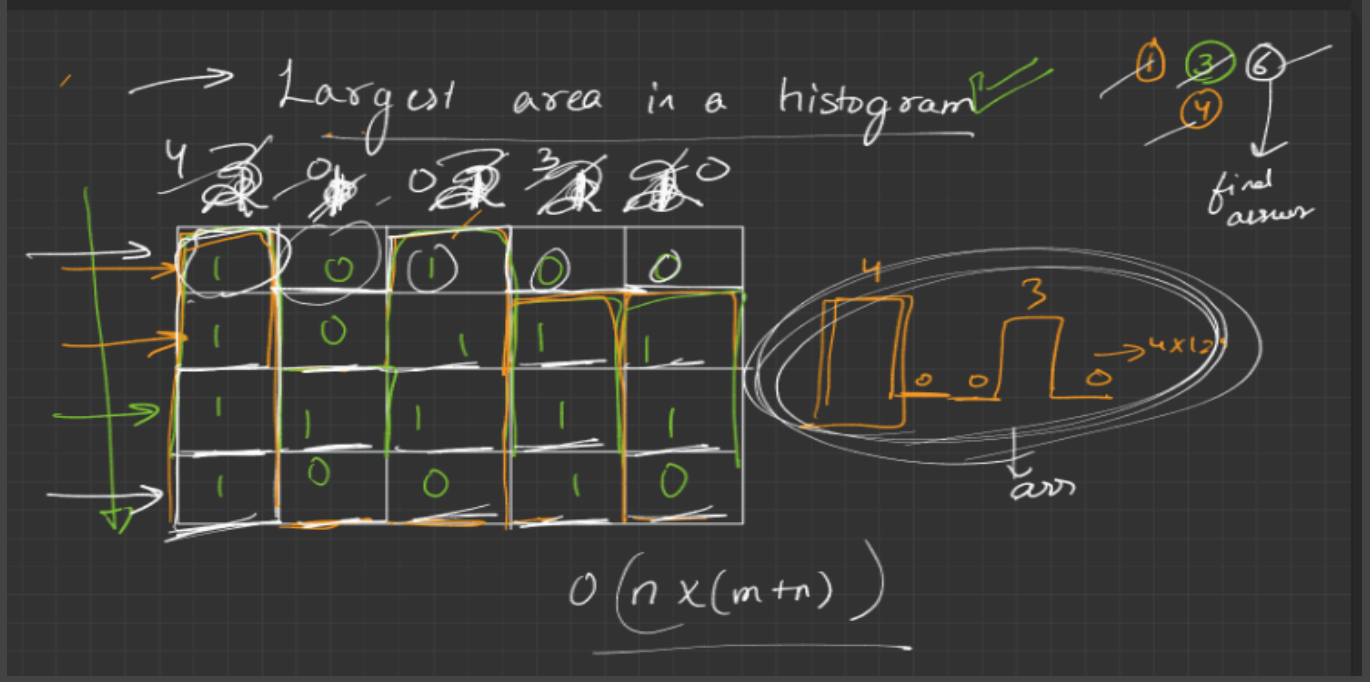

i
n+1
n
j
T.P
i → 0 → a.length()
j → 0 → b.length()
B.V
i → a.length-1 → 0
j → b.length-1 → 0


curr
next
curr
curr
next
next
a.length() - 1

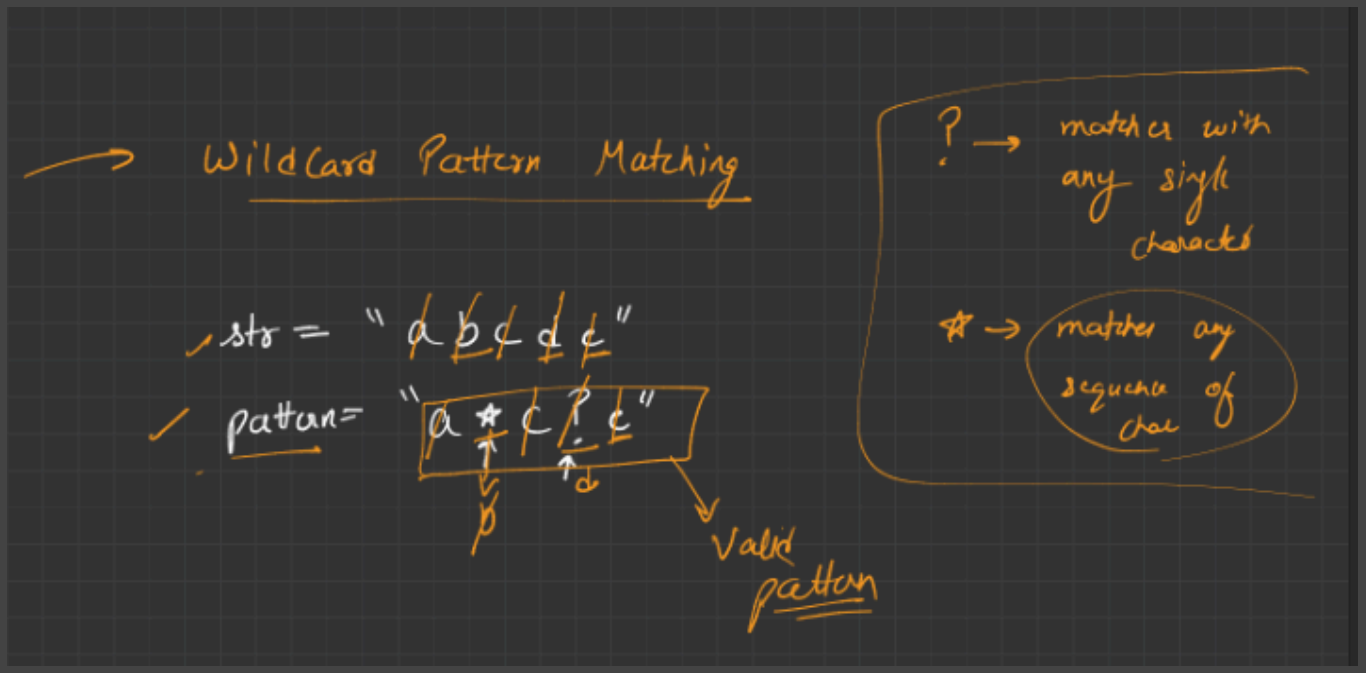

woodl → !!!
a b c d
abcd

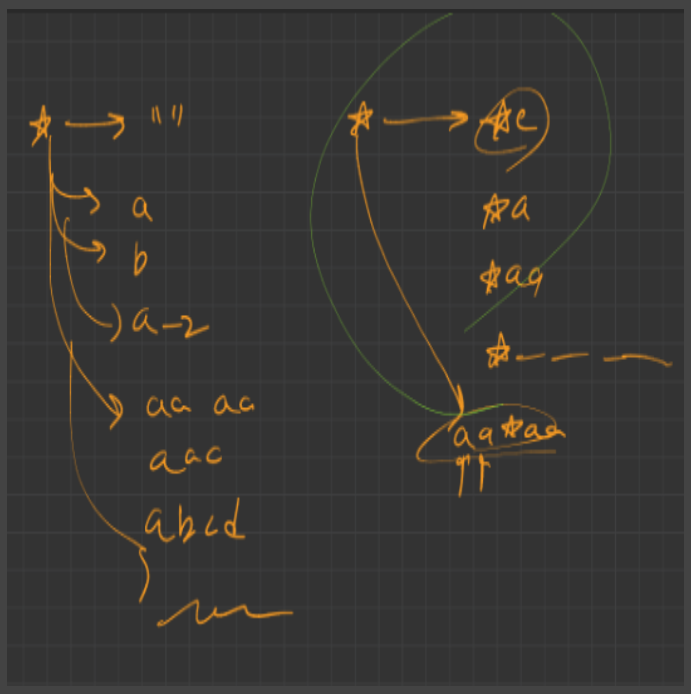
Maximum Rectangle With All 1s:

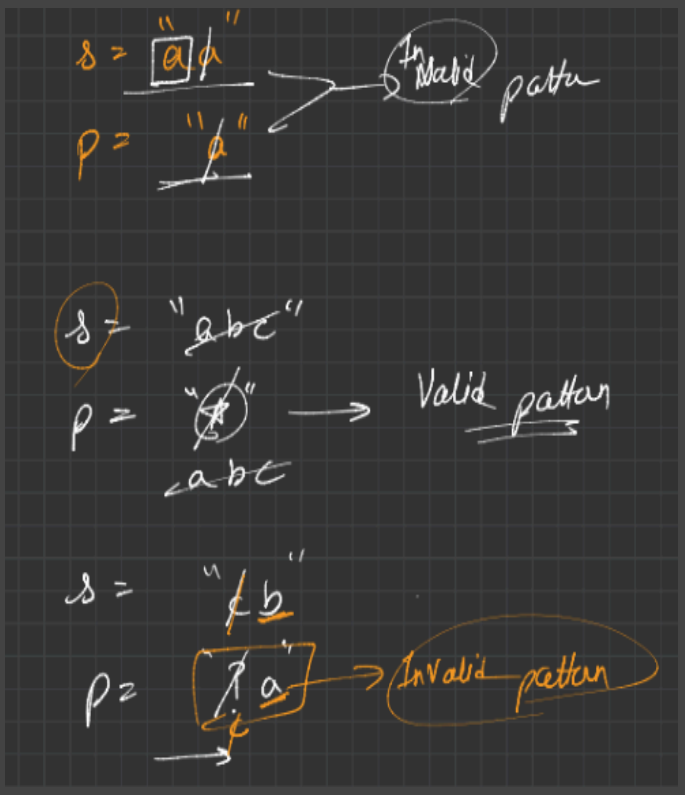


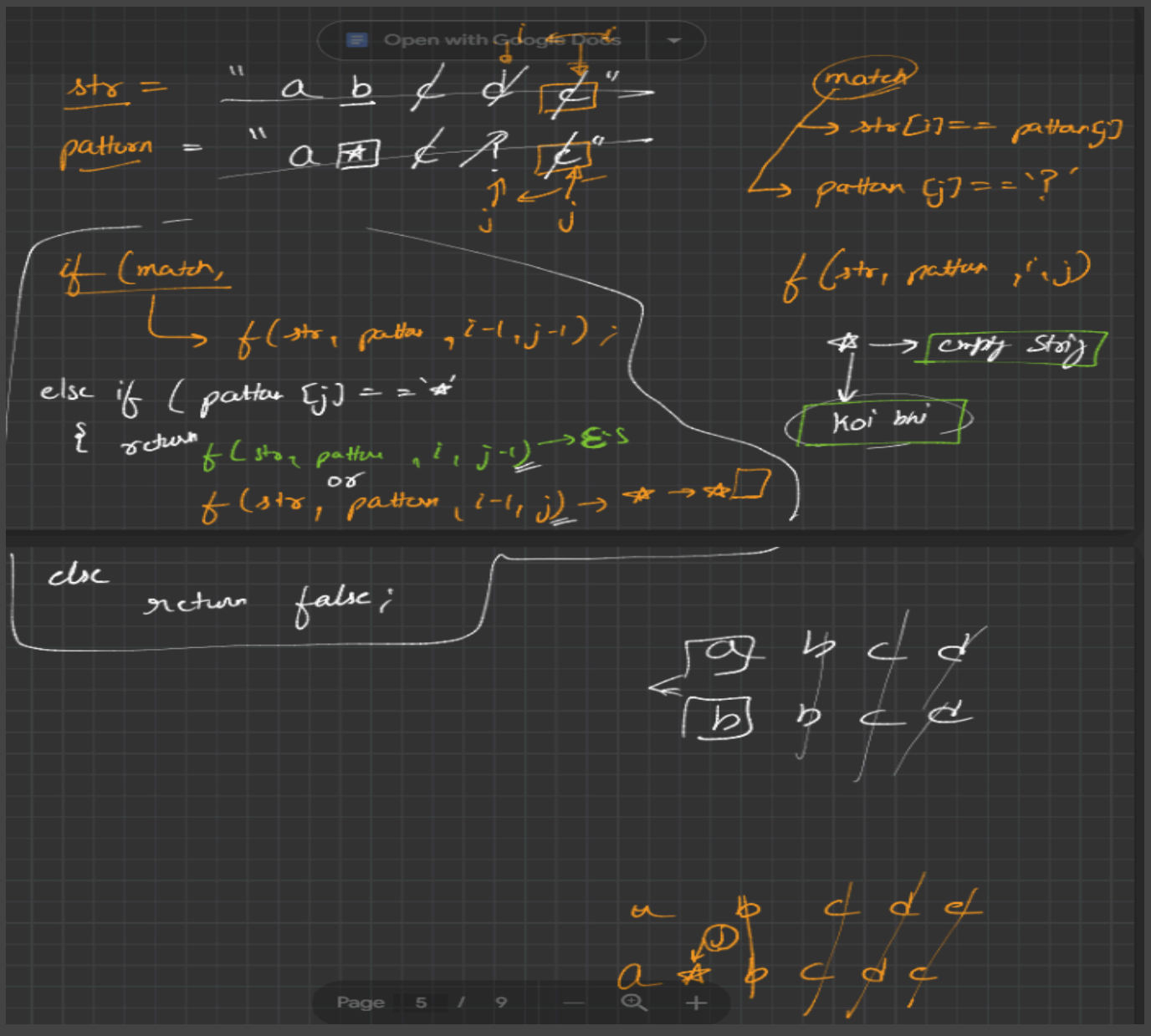


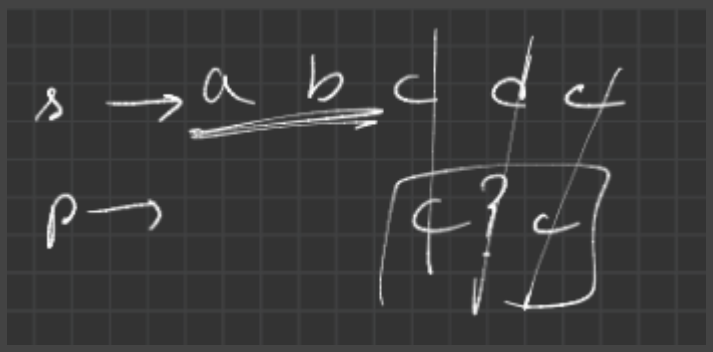
WildCard Pattern Matching By Recursion / Top-Down Approach / Bottom-Up Approach / Space Optimization Approach:

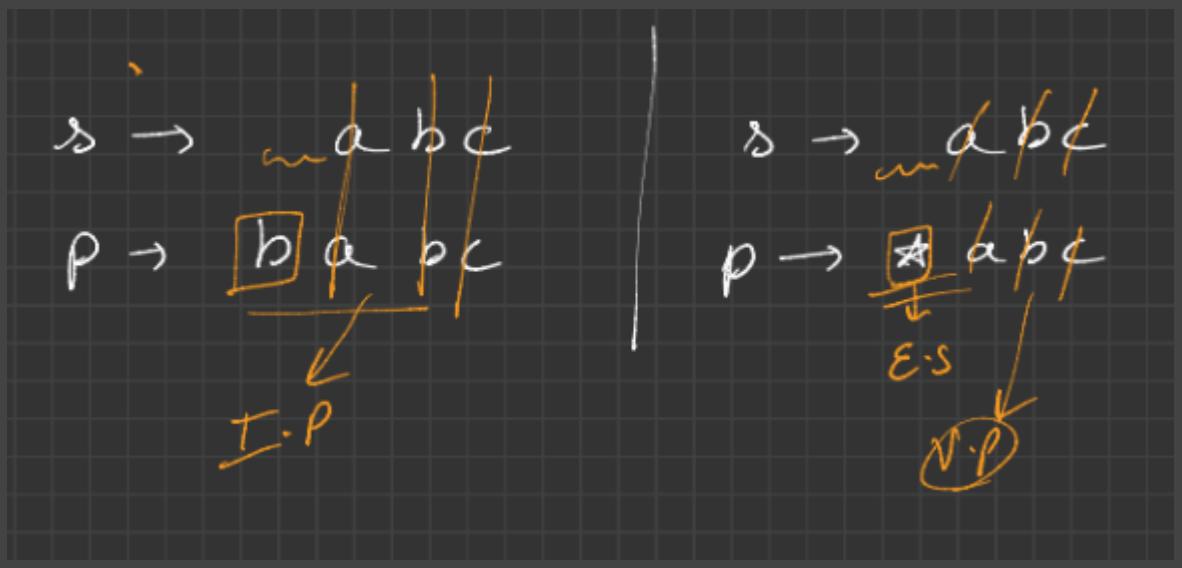


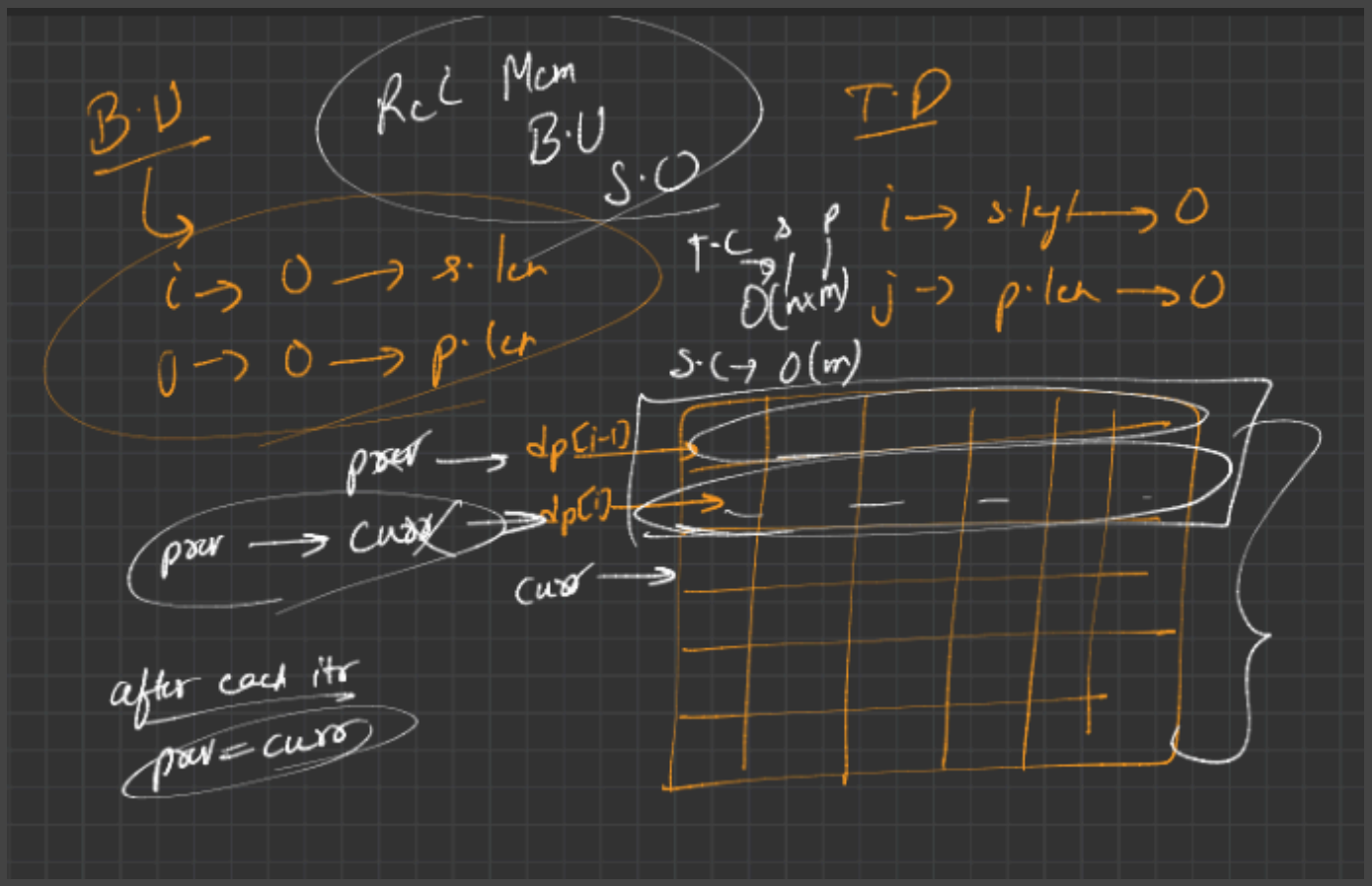

""
a
b
a-z
aa ac
aac
abcd
c
a
aa
aa
aac


s = "a"
p = "a"
Invalid pattern
s = "abc"
p = "*"
abc
Valid pattern
s = "cb"
p = "?a"
Invalid pattern

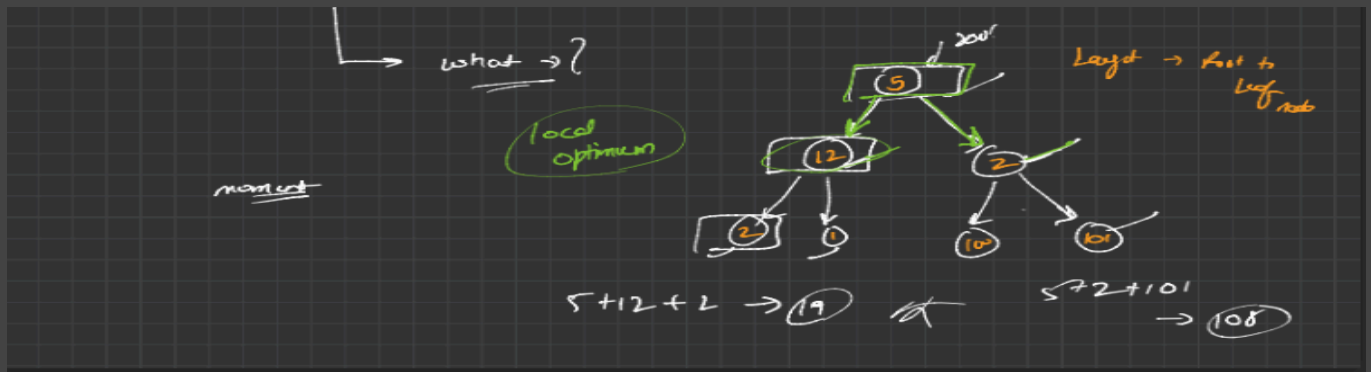

Open with Google Docs
str = "a b c d c"
pattern = "a * c ? c"
match
str[i] == pattern[j]
pattern[j] == '?'
if (match)
f(str, pattern, i-1, j-1);
else if (pattern[j] == '*')
{ return
f(str, pattern, i, j-1)
or
f(str, pattern, i-1, j)
f(str, pattern, i, j)
copy str[j]
Koi bhi
else
return false;
Page 5 / 9

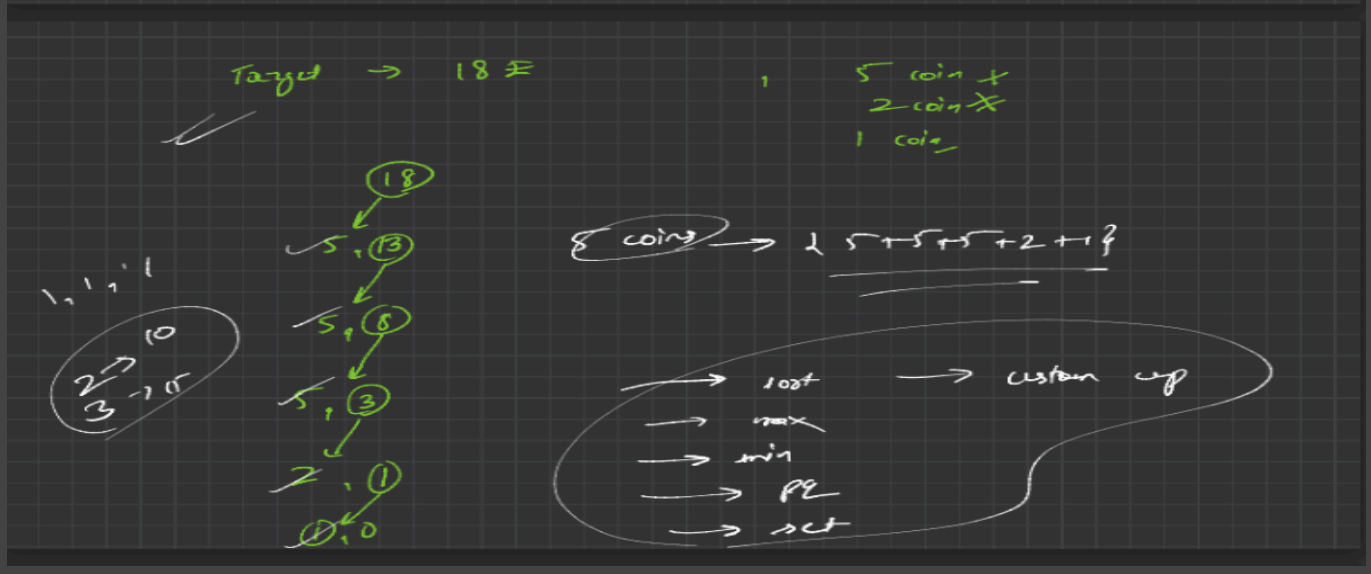

s → a b c d c
p →
c ? c


s → a b c
p → b a b c
I·P
s → a b c
p → a b c
ε·s
N·P

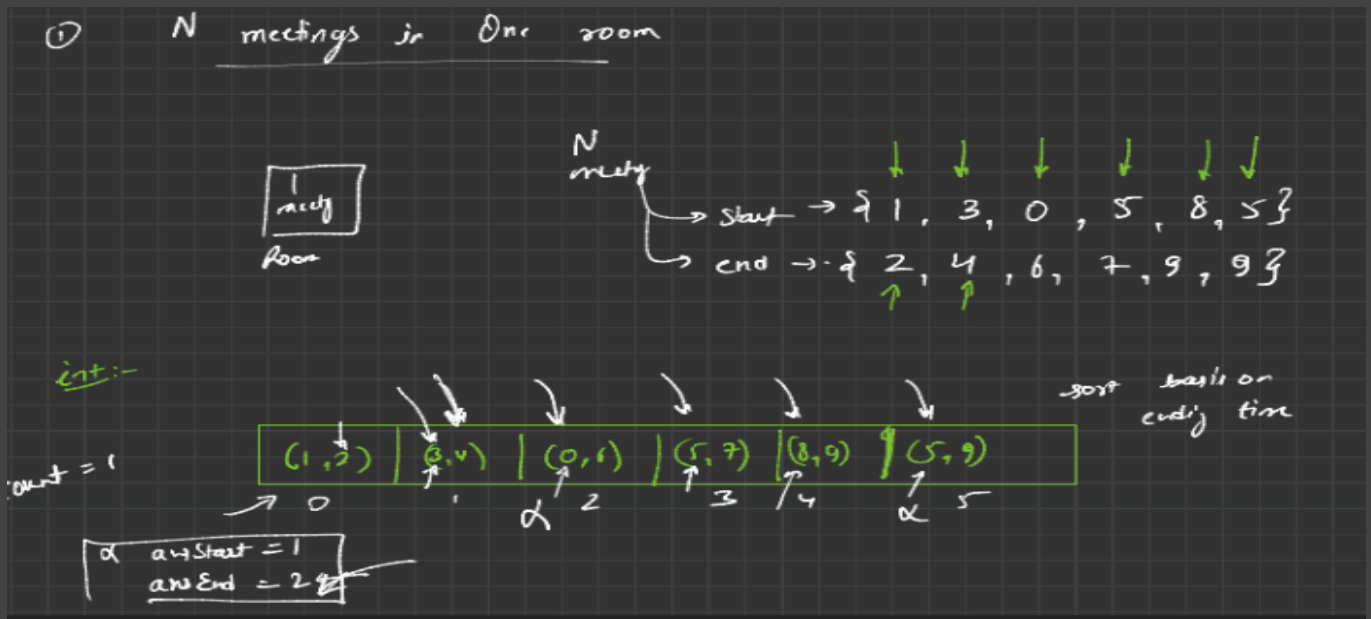

B·U
Rec Mem
B·U
S·O
i → 0 → s.len
j → 0 → p.len
T·D
T·C
O(n×m)
i → s.len → 0
j → p.len → 0
S·C → O(m)
prev → dp[i-1]
curr → dp[i]
curr →
after each itr
prev = curr

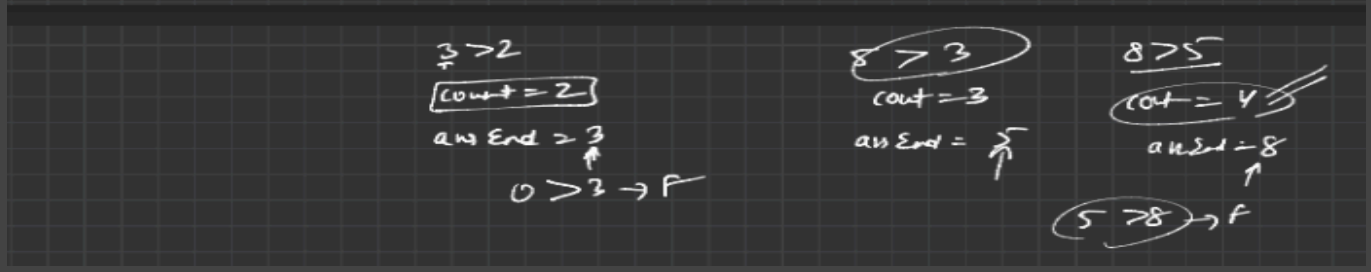
Greedy Algorithm



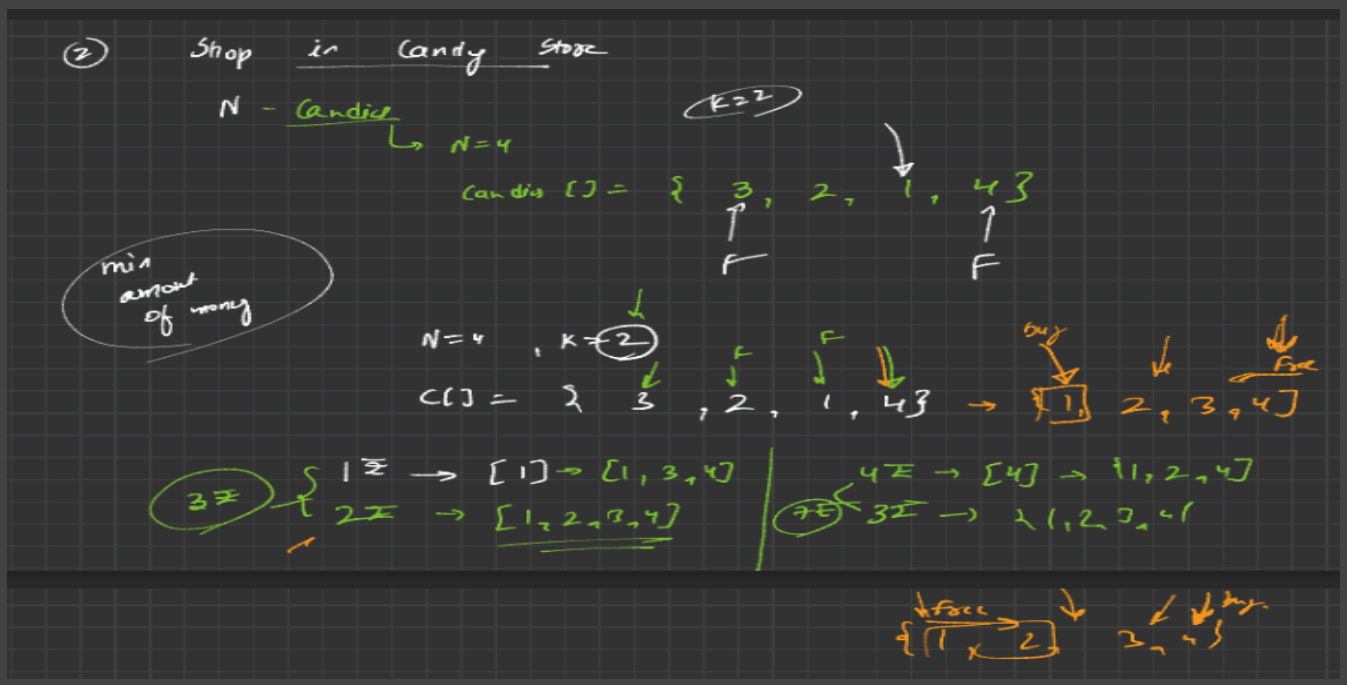


N Meetings In One Room:

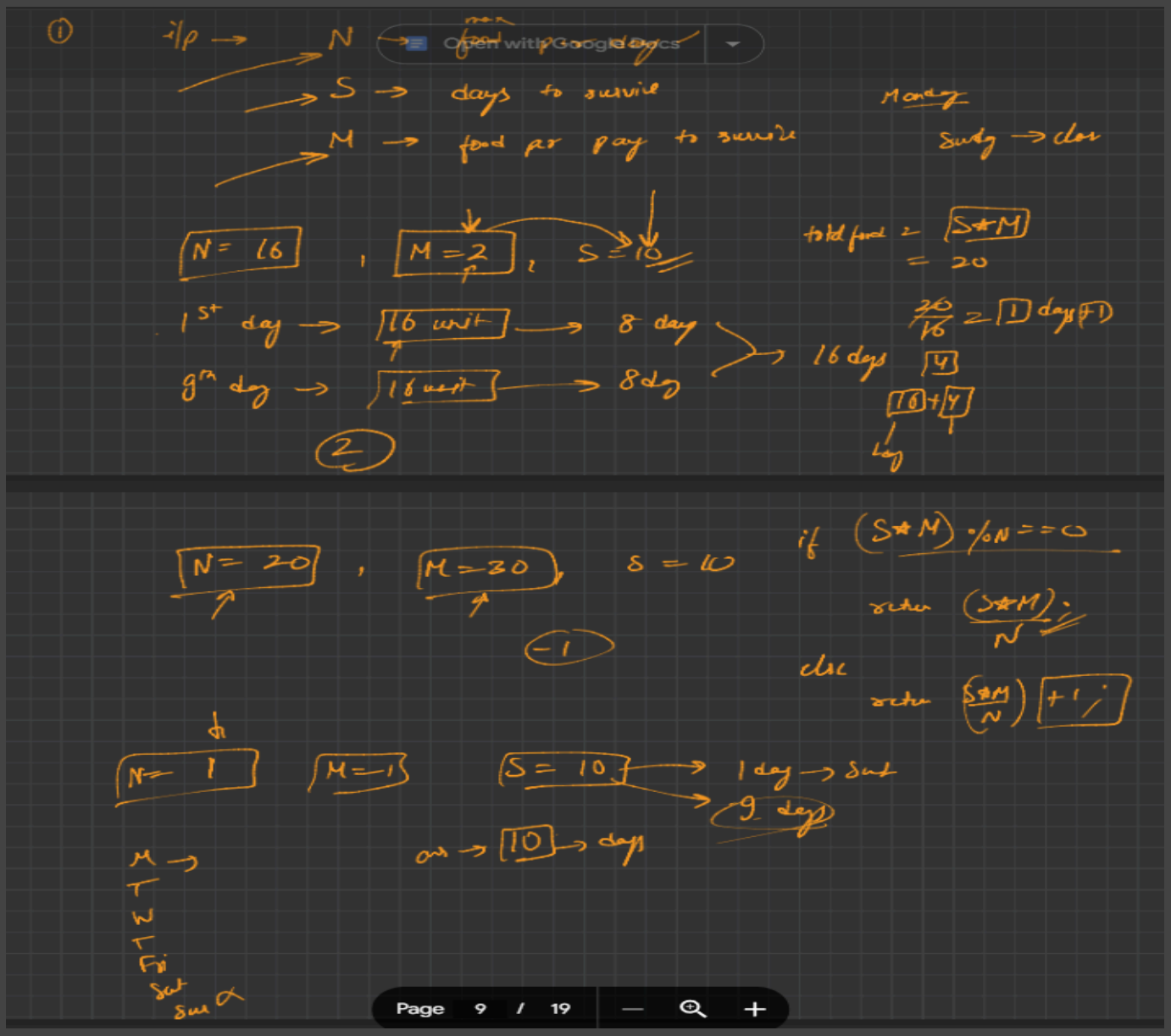




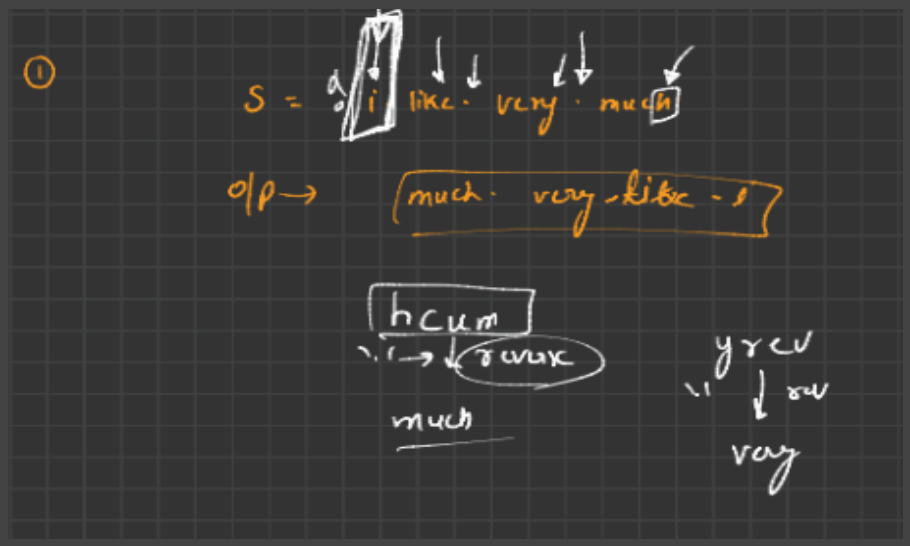
Shop In Candy Store:



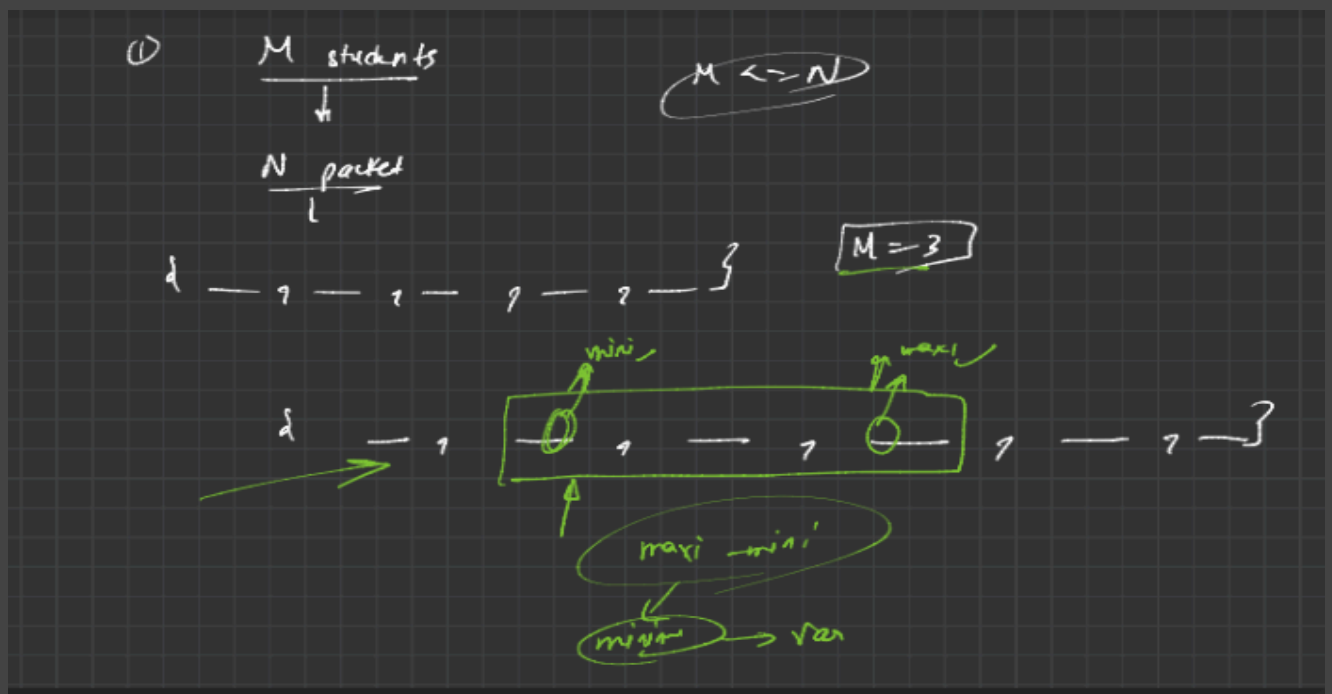
Survival In The Island:

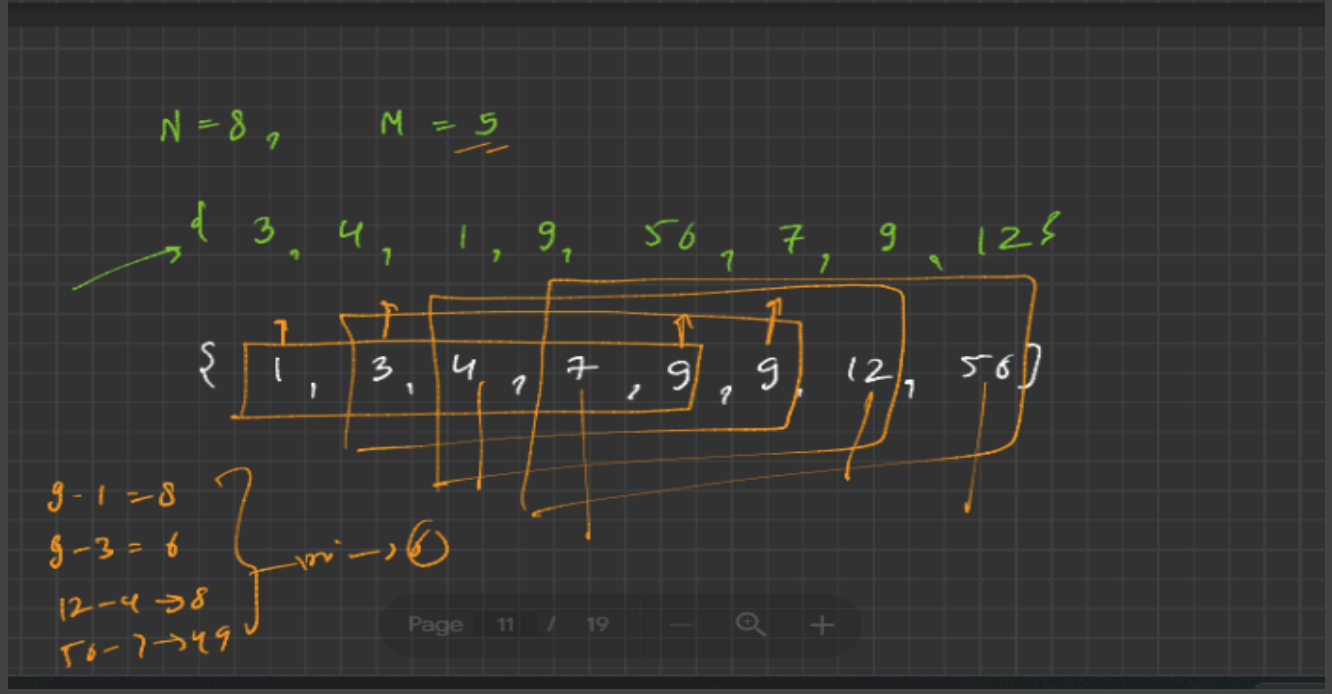


Reverse Words In A Given String:

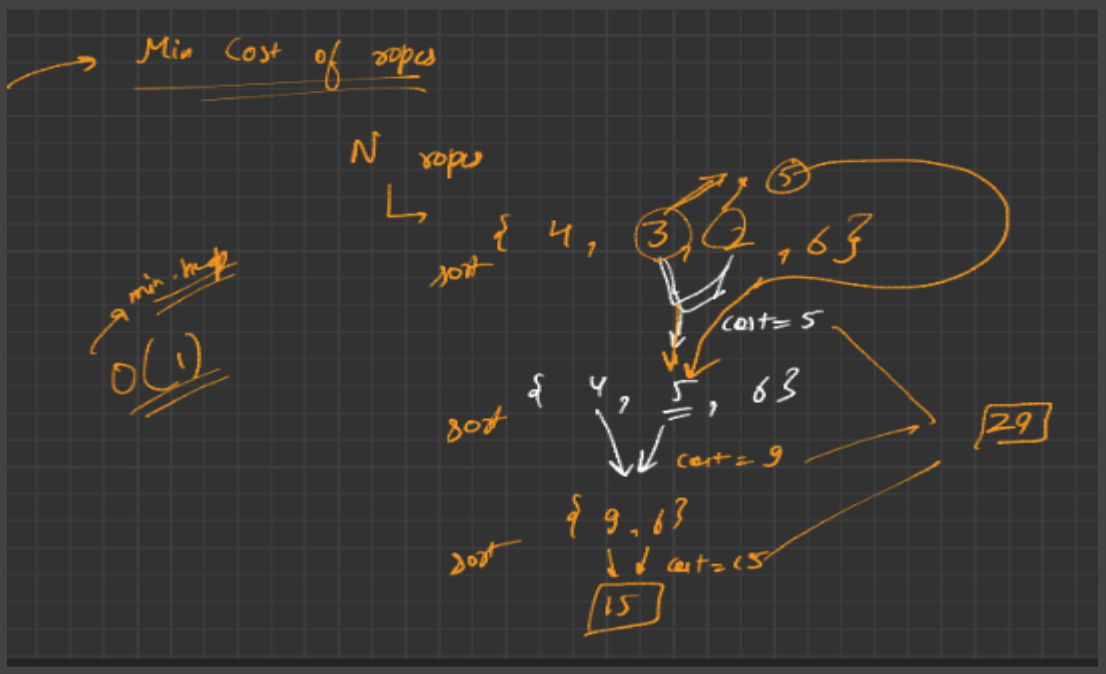


Chocolate Distribution Problem:

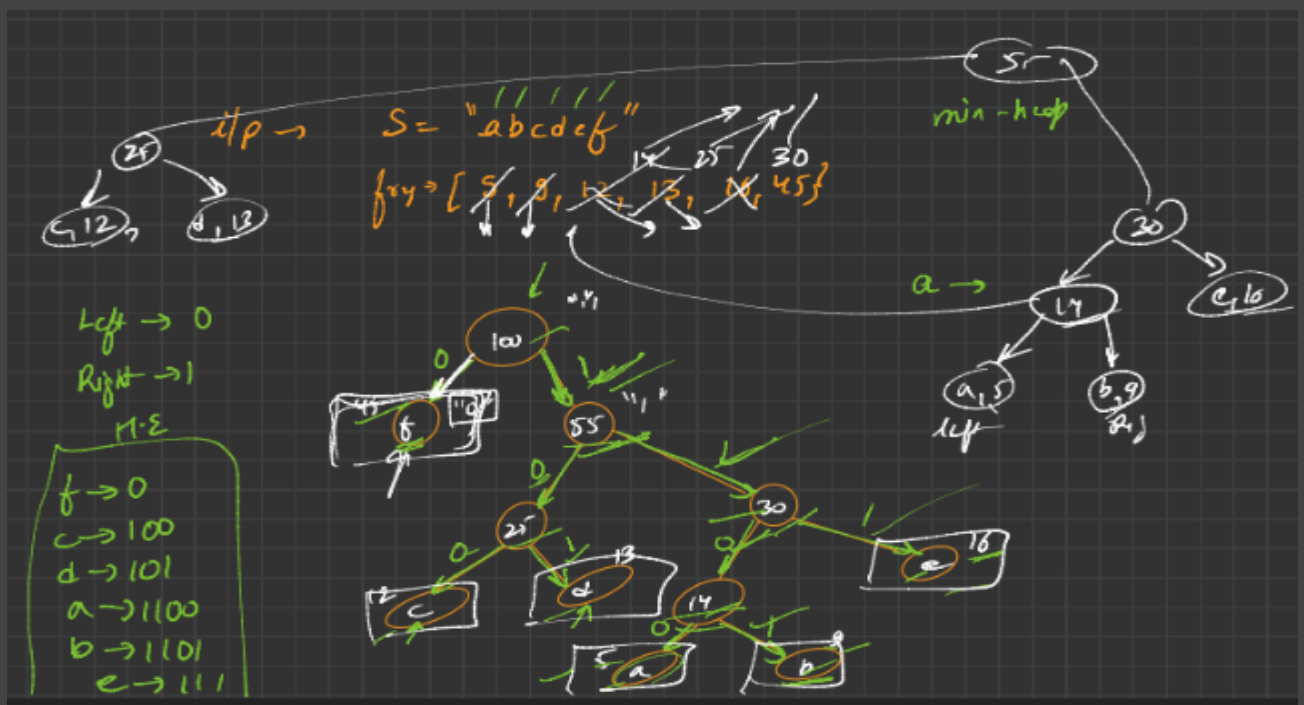


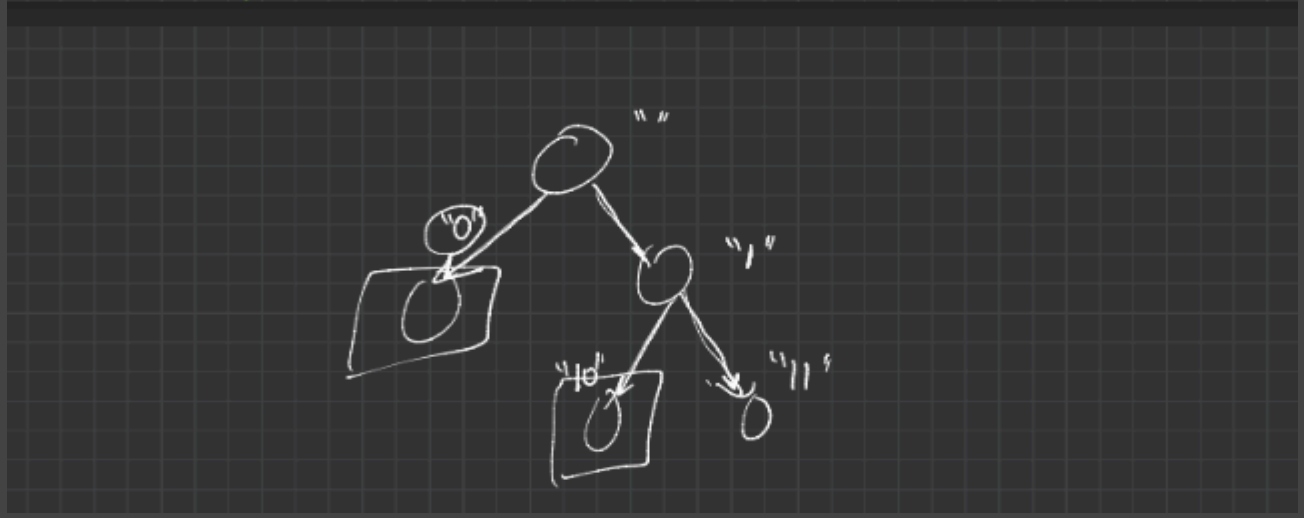


Minimum Cost Of Ropes:

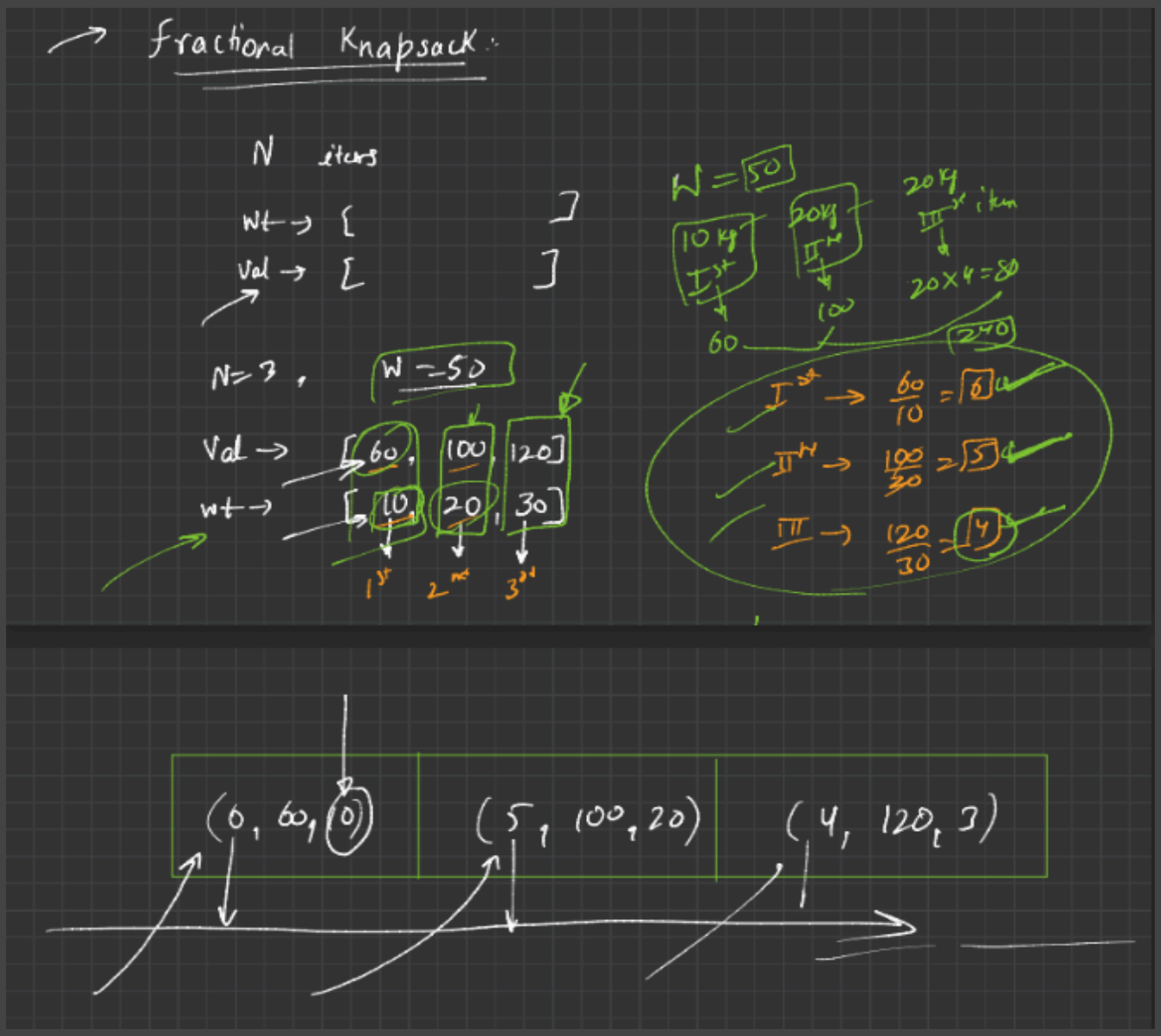


Huffman Encoding:

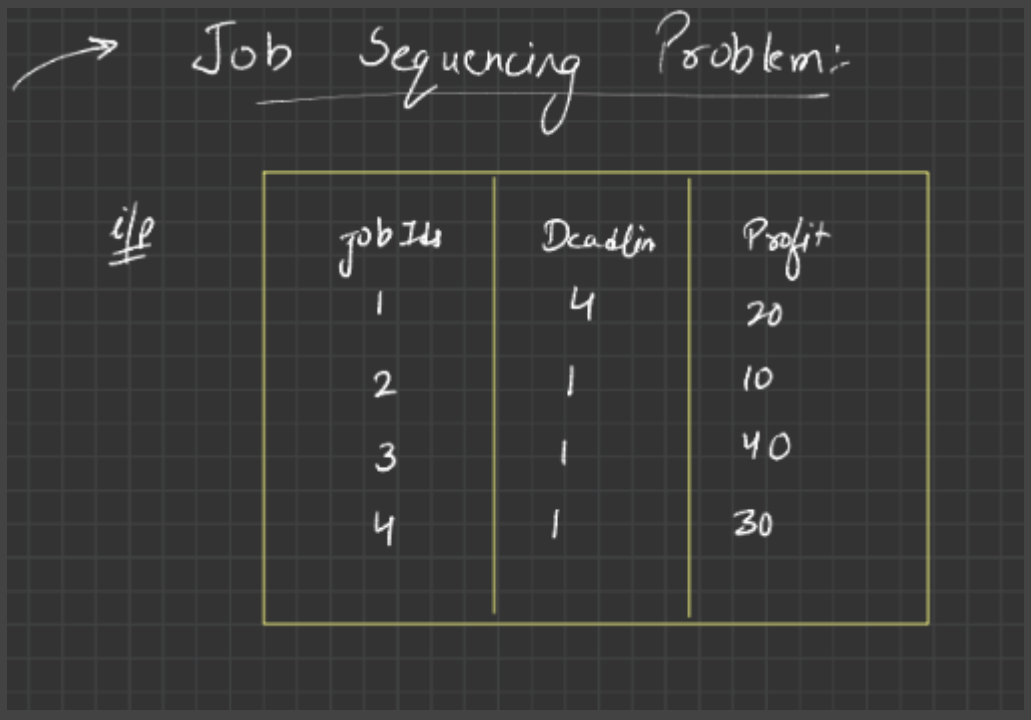




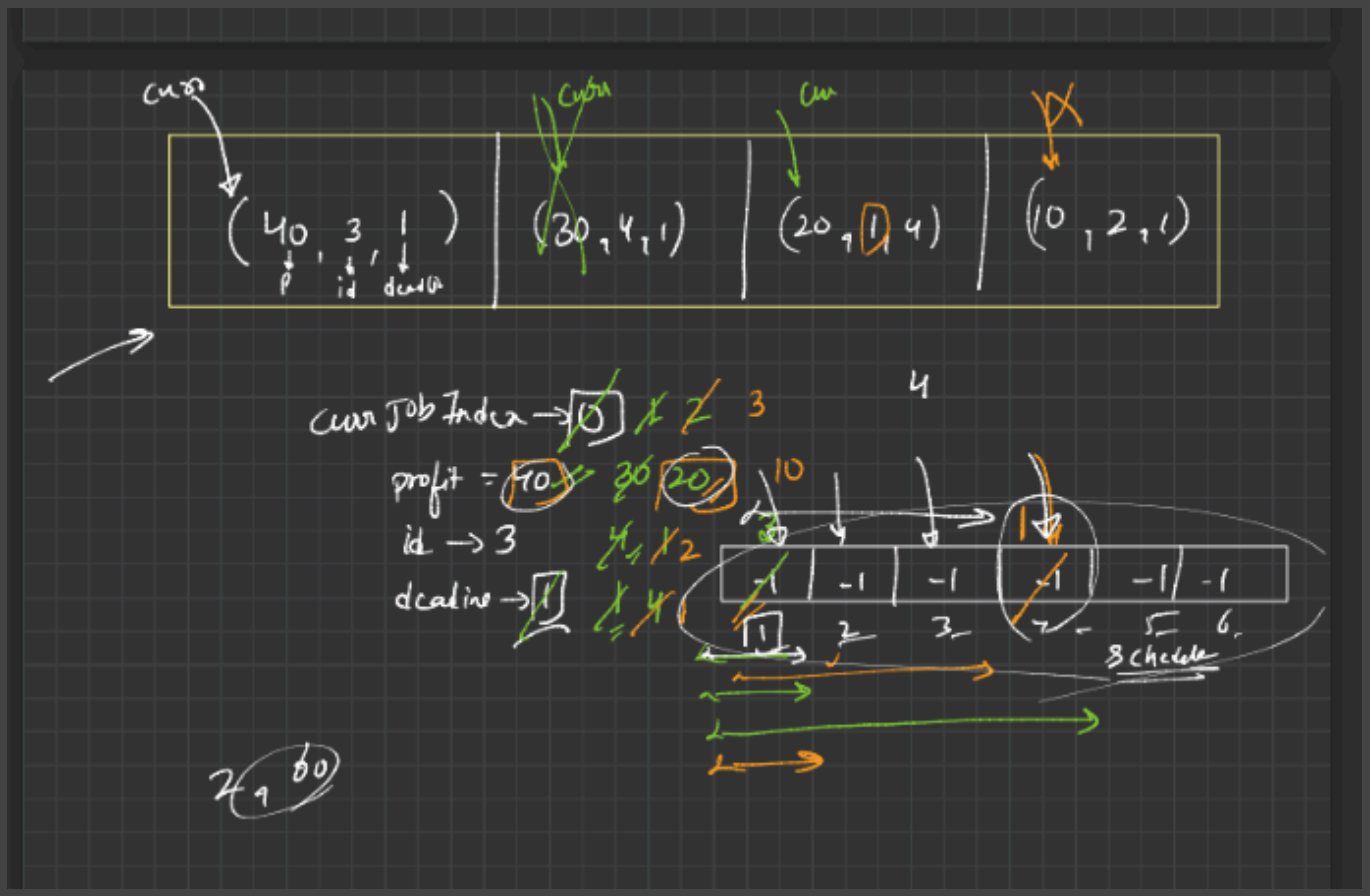
Fractional KnapSack:



Job Sequencing:



job Ids	Deadline	Profit
1	4	20
2	1	10
3	1	40
4	1	30


curr
curr
curr
(40, 3, 1)
p
id
deadline
(30, 4, 1)
(20, 1, 4)
(10, 2, 1)
currJobIndex → 0
profit = 40 30 20
id → 3
deadline → 1
4
10
-1 -1 -1 -1 -1 -1
1 2 3 4 5 6
3 schedule
2, 60