Queue

⇒ Queue Reversal

i/p

q

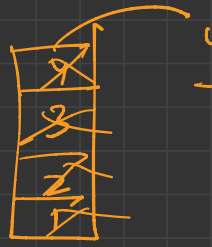
o/p →

reverse

Approach:

→ Use stack → reverse

1 2 3 4



4 3 2 1

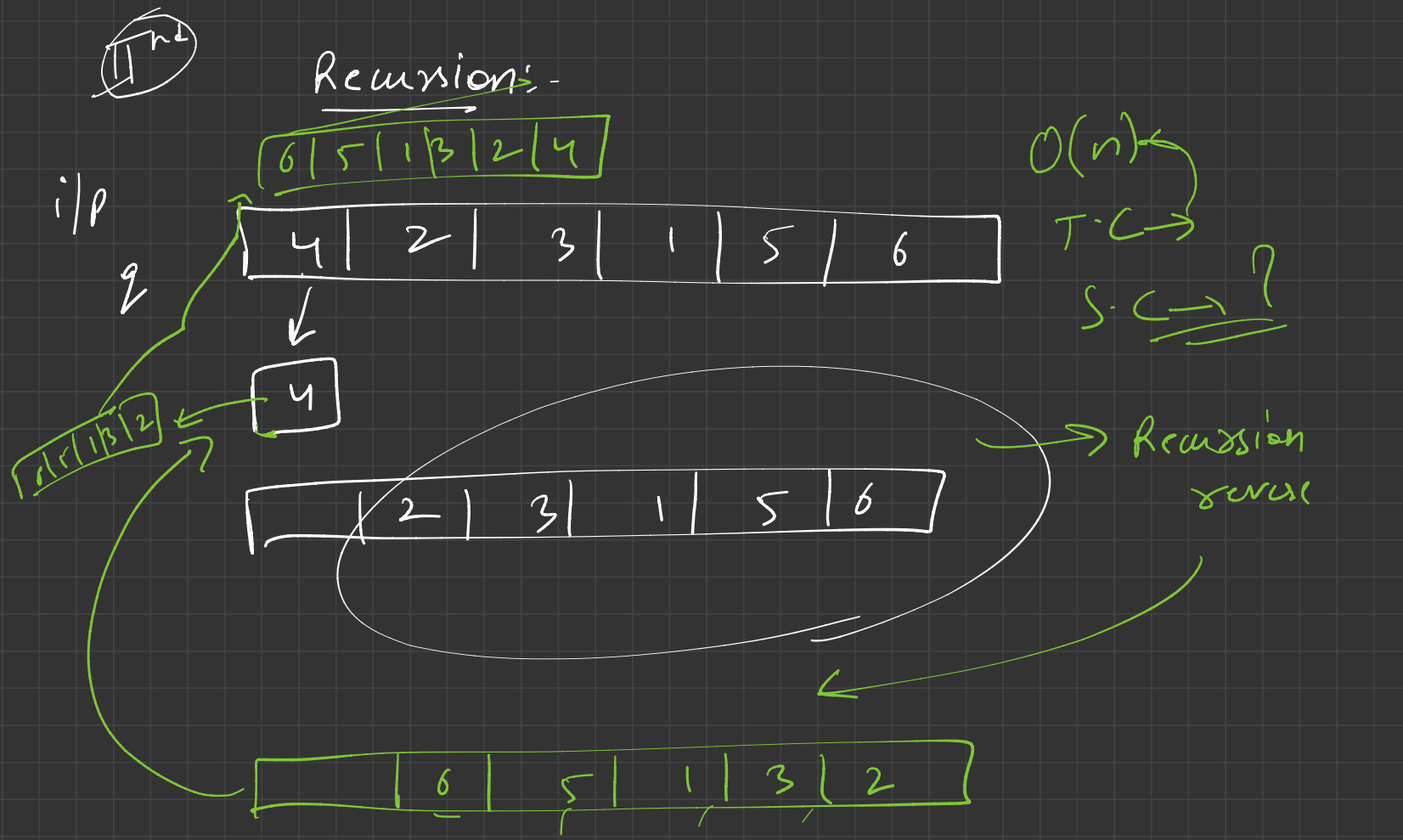
Algo:

→ queue se 1 by 1 element ↓ stack me daalo

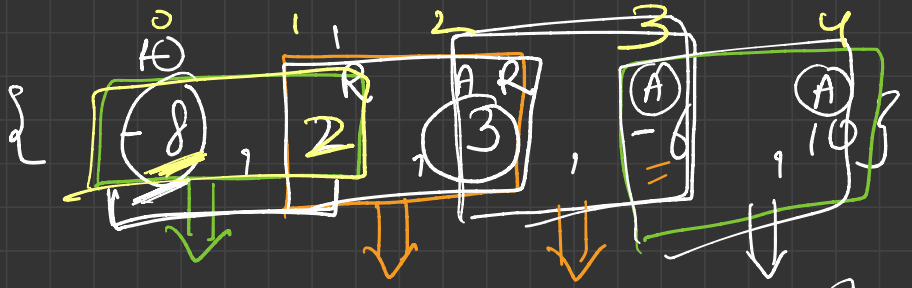
→ stack se 1 by 1 element nikal ↓ queue me daalo

T.C → O(n)

S.C → O(n)


Recursion:-
i/p
2
O(n)
T.C
S.C
?
Recursion reverse

(II)

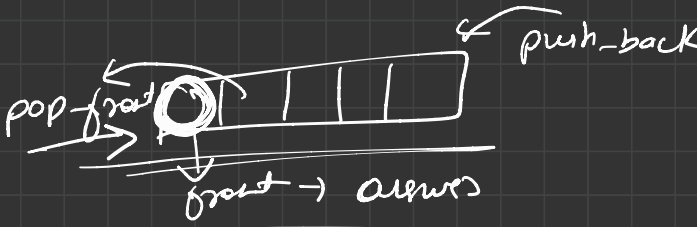


K = 2

ans ⇒ { -8 , 0 , -6 , -6 }

approach:-

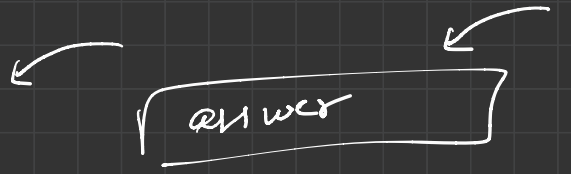
deque <int> dq;



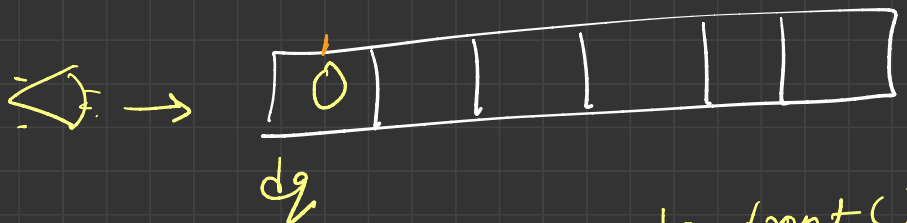
① first k element (first window)

answer ⇒

② ——> loop



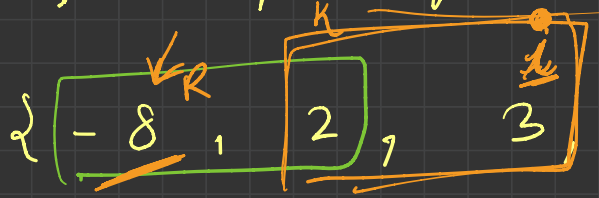
(I) → first K-size window



dq size == 0

answer = dq.front(); or 0

(II) Now processing remaining window



{ -6, 10 }

// Removal

for (i -> <n)

{

// Removal

if (dq.front() - i >= k)

dq.pop_front();

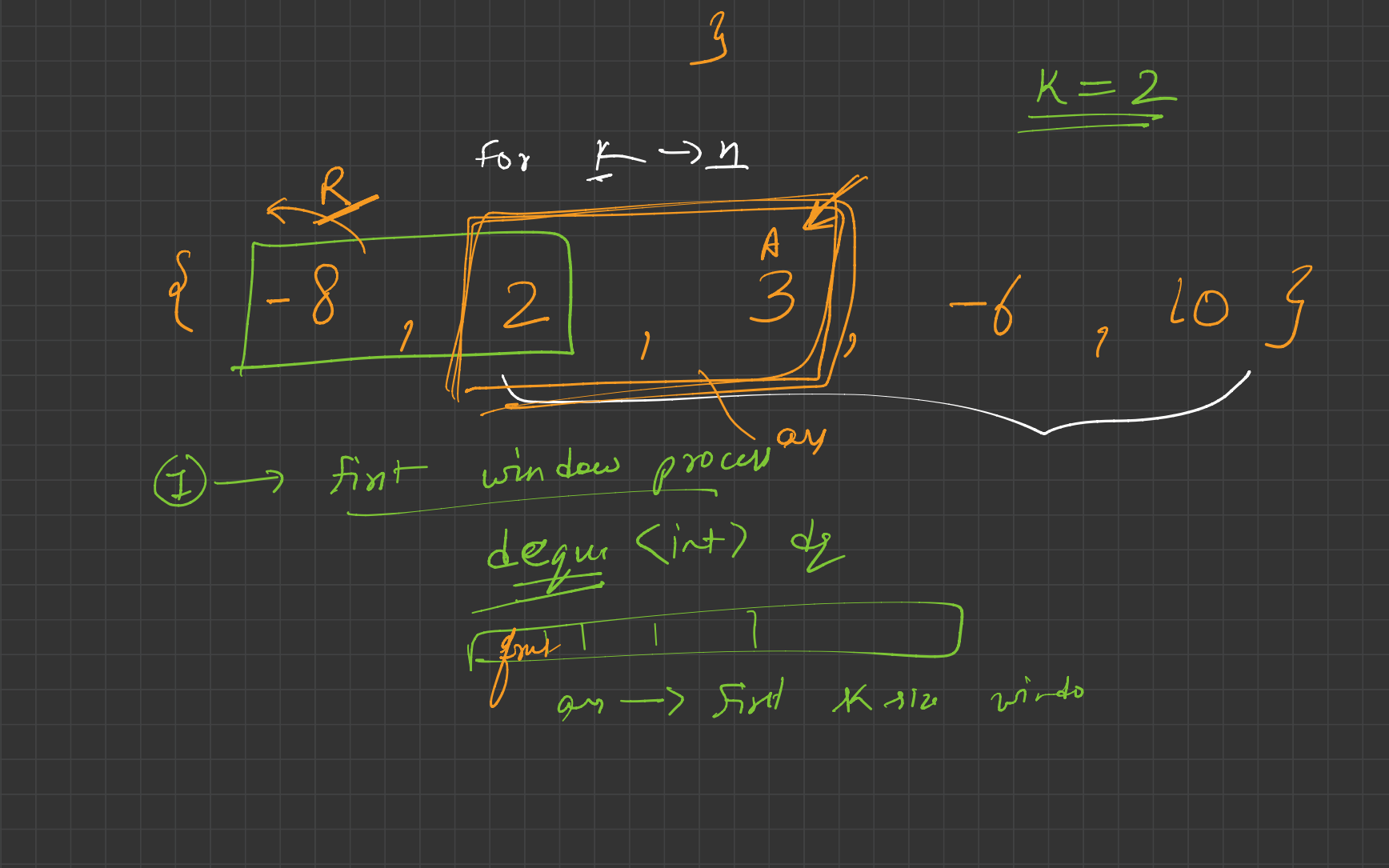
// addition

if (arr[i] < 0)

dq.push_back(i);

ans —(dq.size()>0)→ ans -> dq.front

ans —(size=0)→ 0


K = 2
for K → n
R
-8
2
3
-6
10
(I) → first window process ay
deque <int> dq
front
ans → first K size windo

→ Optimal approach → Homework

→ Reverse first K elements of Queue

stack

K = 3

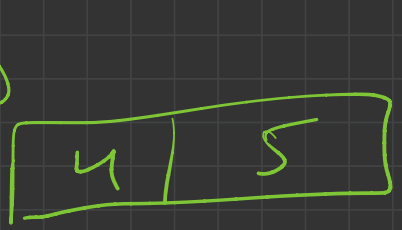
i/p {1, 2, 3, 4, 5}

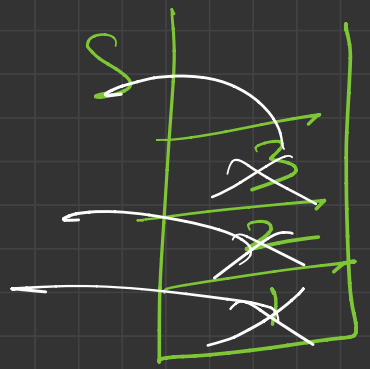
Run

o/p {3, 2, 1, 4, 5}

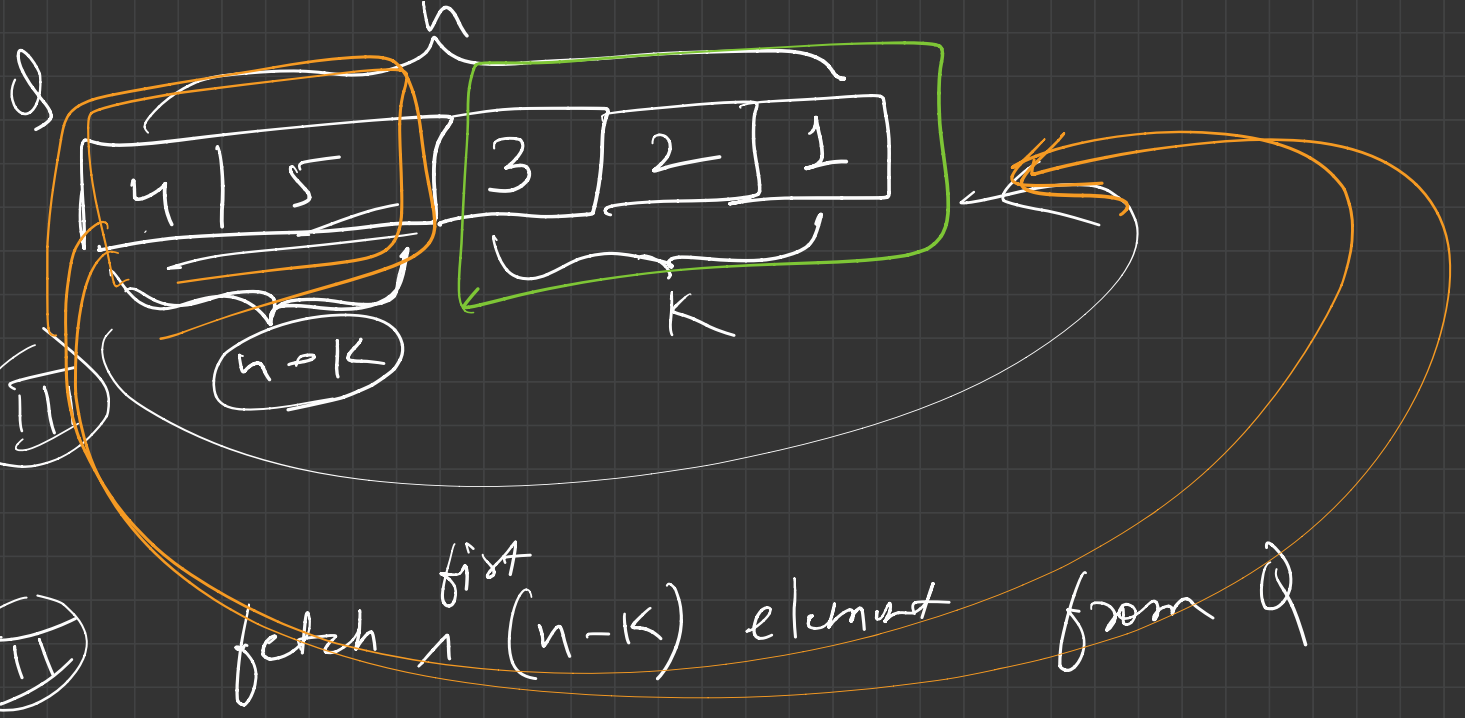
Algo:-

(I) fetch first K element from Q & put into stack

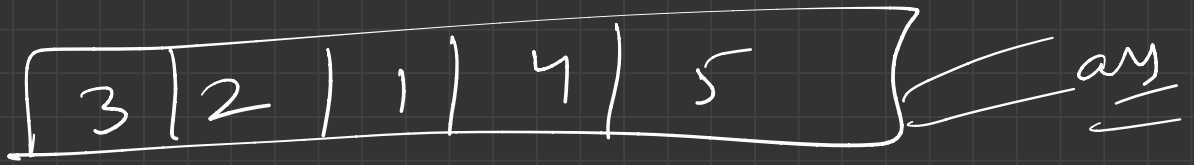




(II) fetch element from stack & put into Q



III fetch first (n-K) element from q & push_back

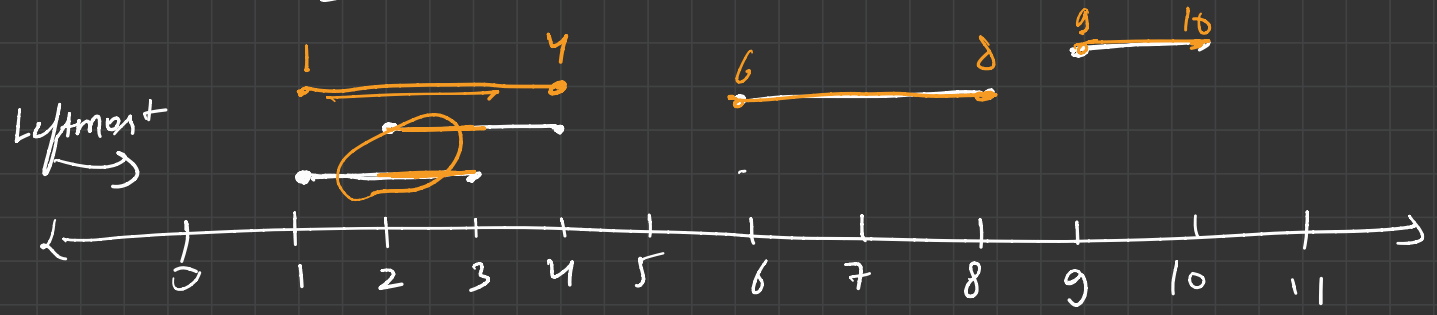


$$O(K) + O(K) + O(n-K) + O(N) + O(N) + O(n-n) = O(n)$$

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

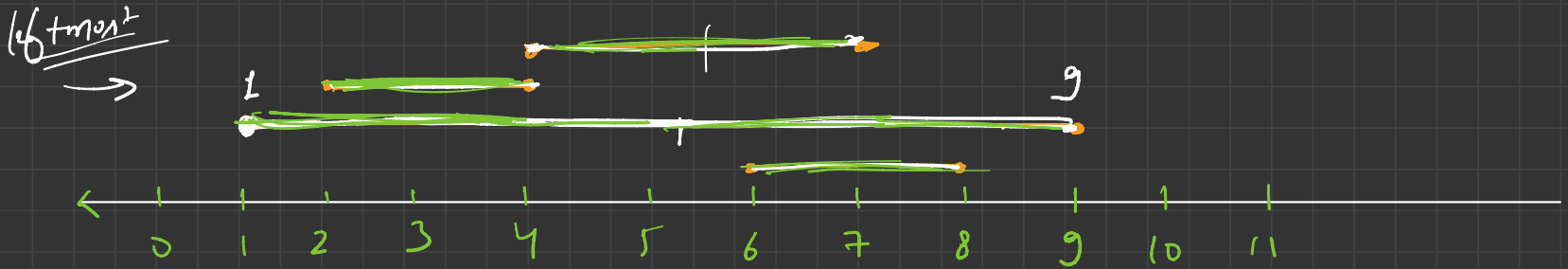
→ Merge Overlapping Intervals

i/p → [1, 3] [2, 4] [6, 8] [9, 10]



or [1, 4) [6, 8] [9, 10]

i/p → {6, 8} {1, 9} {2, 4} {4, 7}



o/p → [1, 9]

Algo:

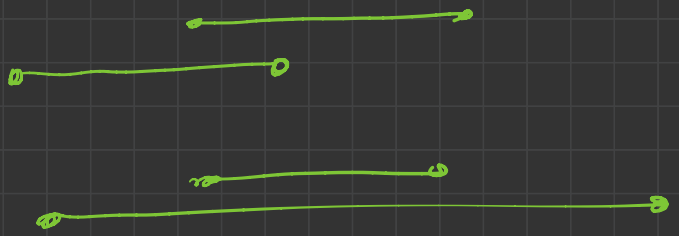
(I) sort interval → according to their starting time

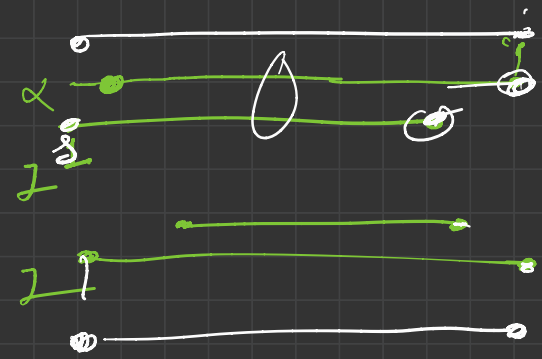
(II) iterate over all intervals

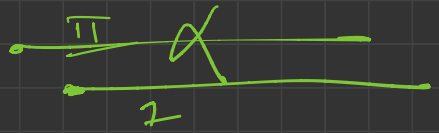
if (overlap / match) → interval update

else

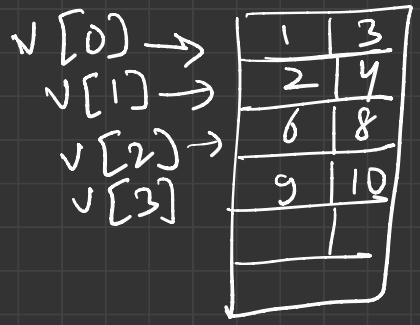
ans store

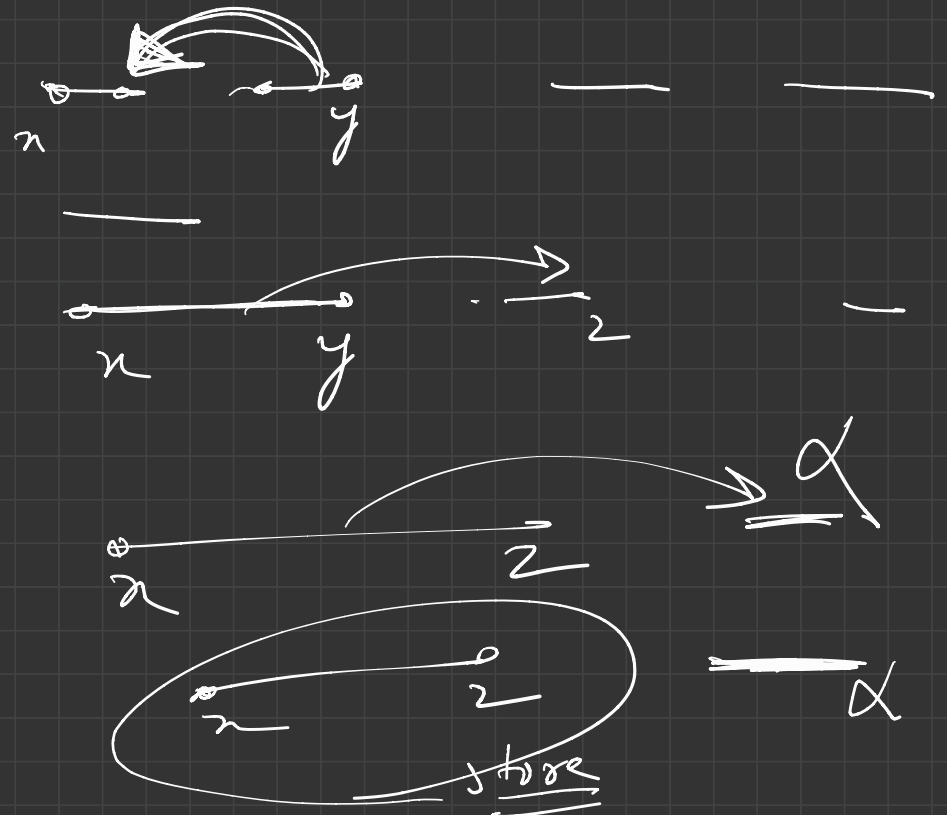


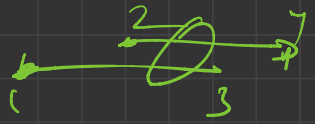


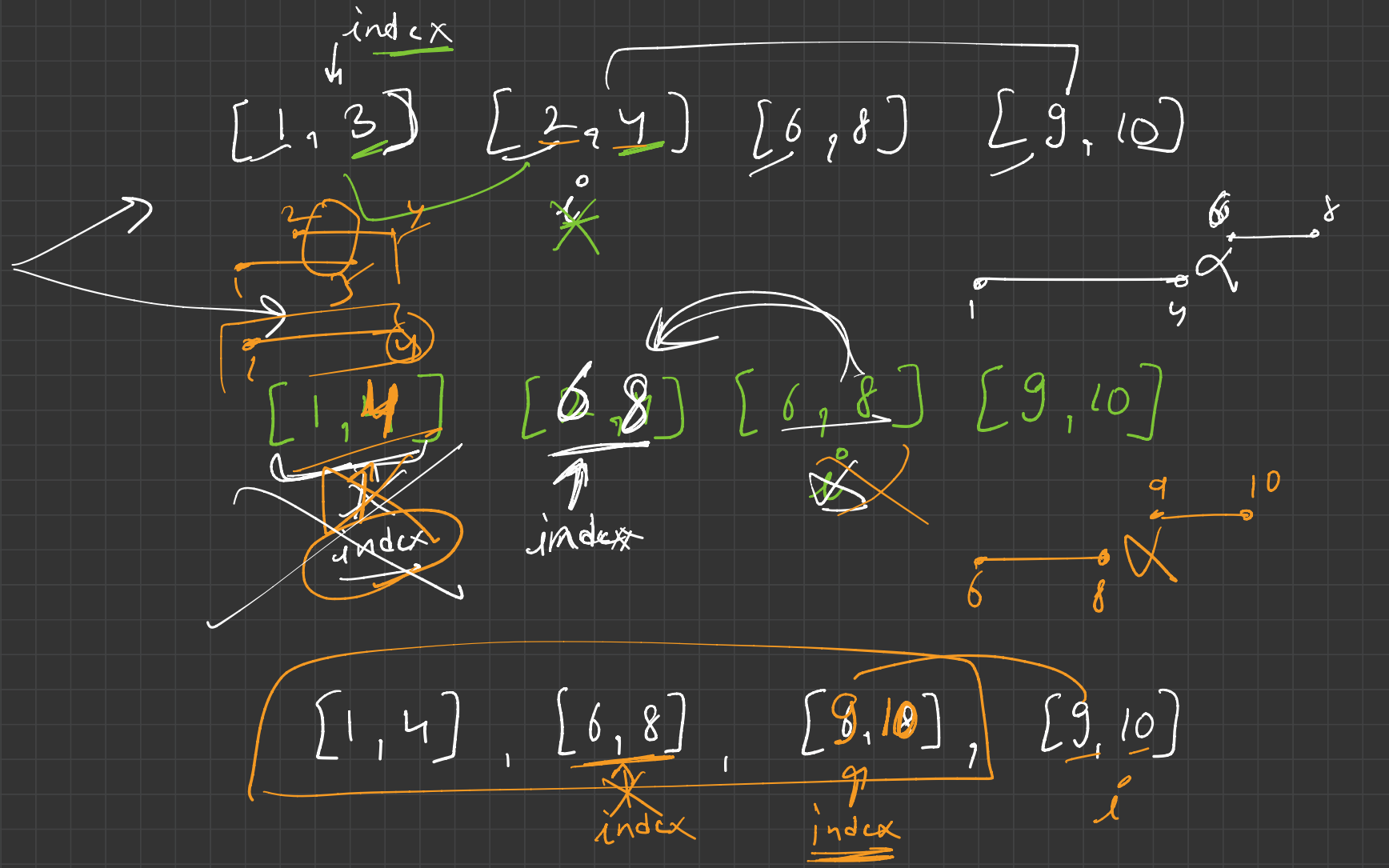


vector < vector <int> > V




x
y
y
z
α
x
z
x
z
α
more


2
3

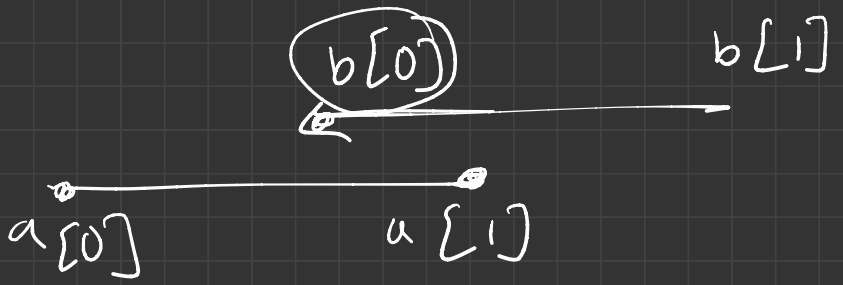

index
[1, 3] [2, 4] [6, 8] [9, 10]
[1, 4] [6, 8] [6, 8] [9, 10]
index
index
[1, 4] , [6, 8] , [9, 10] , [9, 10]
index
index
i

for(i=0 ⟶ <= index)

{ arr.push_back() 値を入れる)

}

値を返す arr



Dry Run

[6, 8] [1, 9] [2, 4] [4, 7]

(I) sort

[1, 9] [2, 4] [4, 7] [6, 8]

index

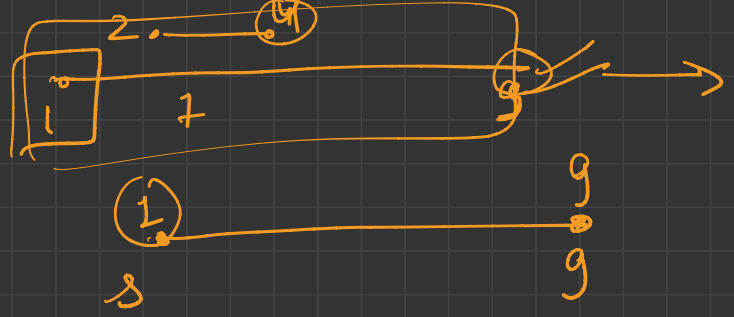
index → check current interval can be merged to other or not

(index point)

prev Interval (I)

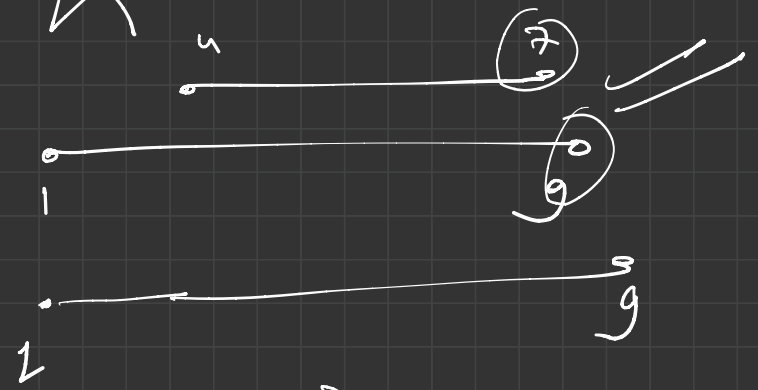
i → current Interval (II)

arr.size()

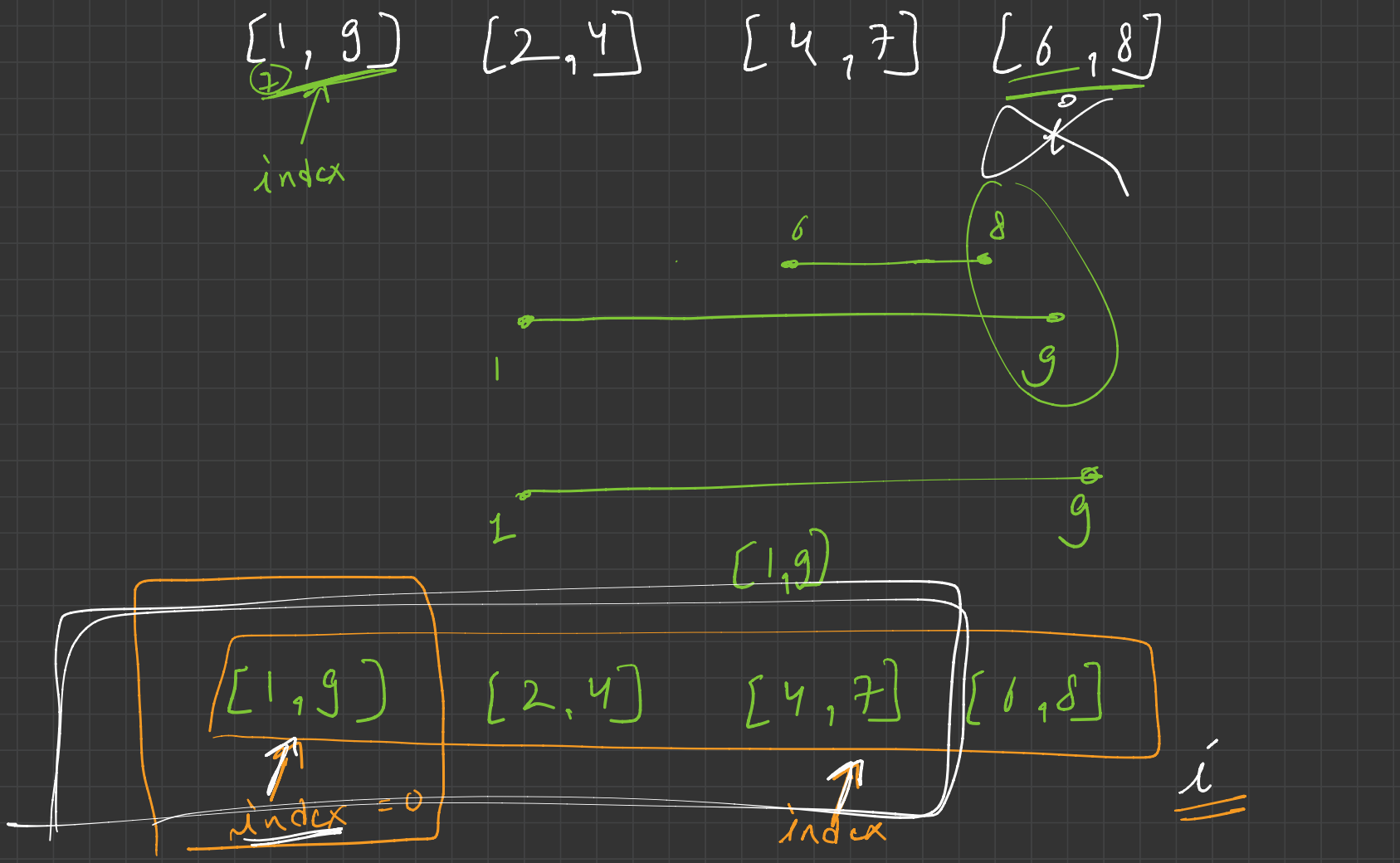


[1, 9) [2, 4] [4, 7] [6, 8]

index



[1, 2)


[1, 9]
[2, 4]
[4, 7]
[6, 8]
index
6
8
1
9
1
9
[1,9]
[1, 9]
[2, 4]
[4, 7]
[1, 8]
index = 0
index
i

ans = [1, 9]

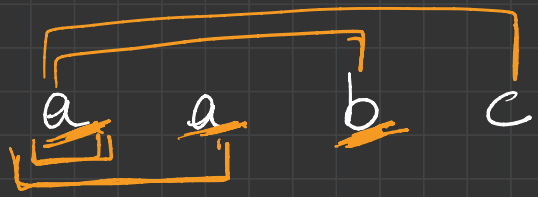
ans → [1, 9] [2, 4] [7, 7]

$O(n \log n)$
+
$O(n)$
+
$O(n)$

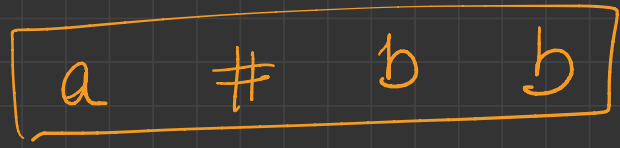
T.C → $O(n \log n)$

→ First non-repeating character in stream

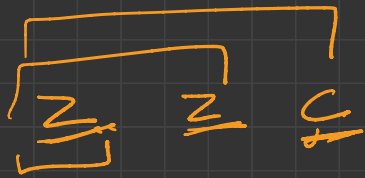
i/p →



o/p →

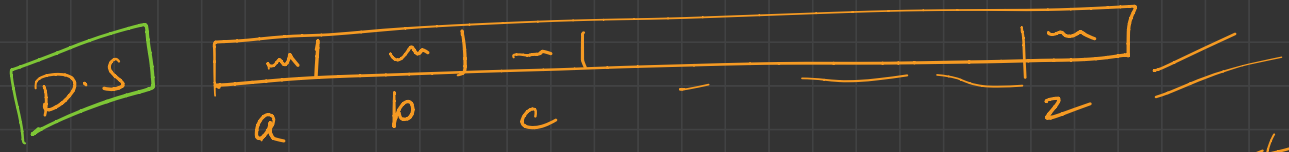


i/p →



o/p → z # c

Approach:-



Why-! → map <char, int> count;

count → N.R

Option → arr[26] → arr[26] = {0};

0 — 'a' → count

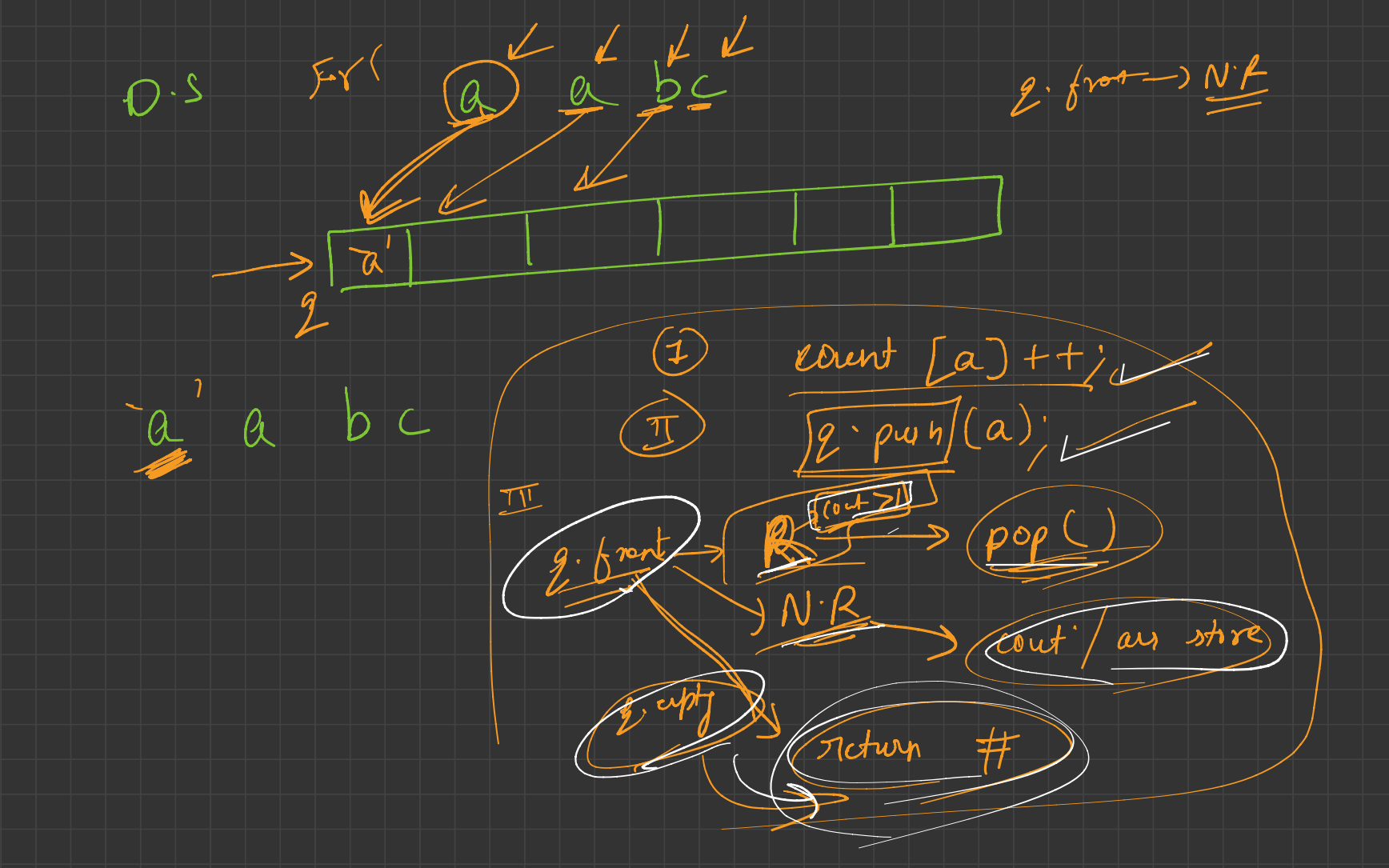
1 - b —

25 — 'z' → count

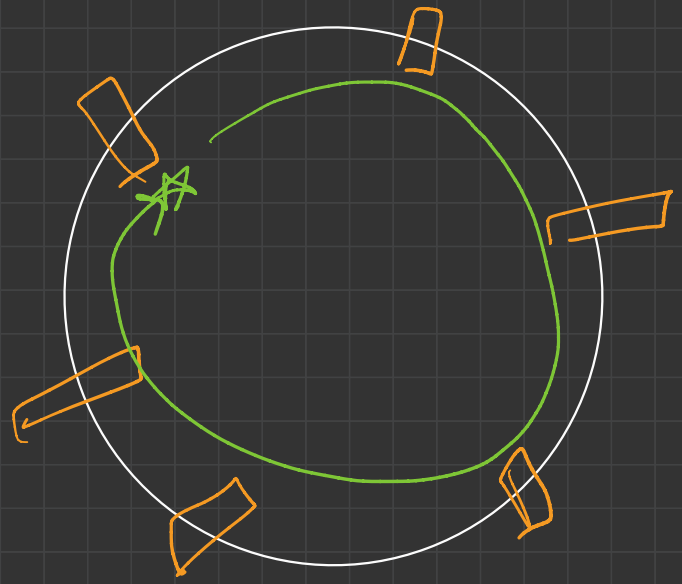
if [> 1] → repeat

else

Non-repeaty

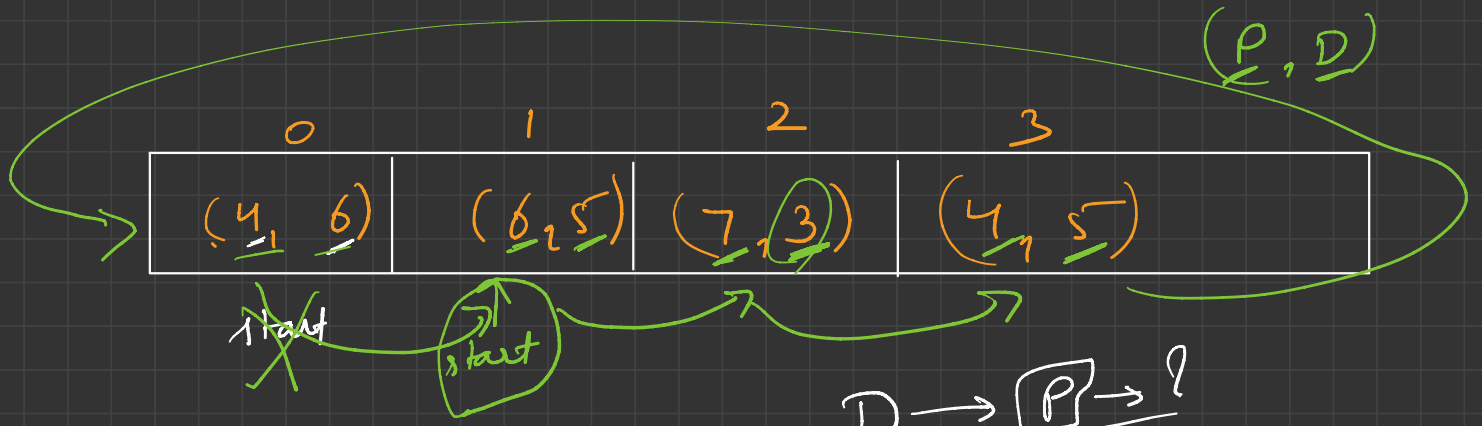

D·S
For(
a
a
b
c
a'
q
q·front → N·R
a' a b c
I
count [a] ++;
II
q·push(a);
III
q·front
R
cout >>
pop()
N·R
cout / ans store
q·empty
return #

→ Circular tour:



P.P ⟶ Petrol data (x litre)

⟶ next P.P distance (y km


(P, D)
0
1
2
3
(4, 6)
(6, 5)
(7, 3)
(4, 5)
start
start
D → P → ?
balance
= 4 - 6 = -2
< 0
>= 0
start not possible
Normal sahi logon aage jaate ho
balance = 6 - 5 = 1 >= 0
= 1 + 7 = 8 limit
8 - 3 = 5 >= 0

$= 5+4 = 9$ unit P

$= 9-4 = 5L$

$= 5+4 = 9L$

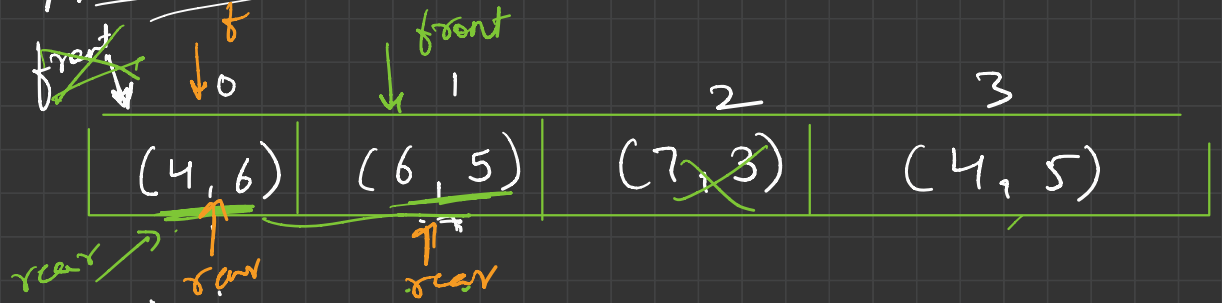
$= 9-6 >= 0$

$3 >= 0$

approach #1 :- B.F

↳ $O(n^2)$

approach #2 :-



Algo:

one block to other travel possible

P - D >= 0

if

rear ++

if (front == rear)

circle complete

else

front = rear + 1

rear = front

front ++

4 - 6 >= 0

Fol

balance >= 0

Algo:-

if (start block to its next block travel possible)

rear ++

else

front = rear + 1

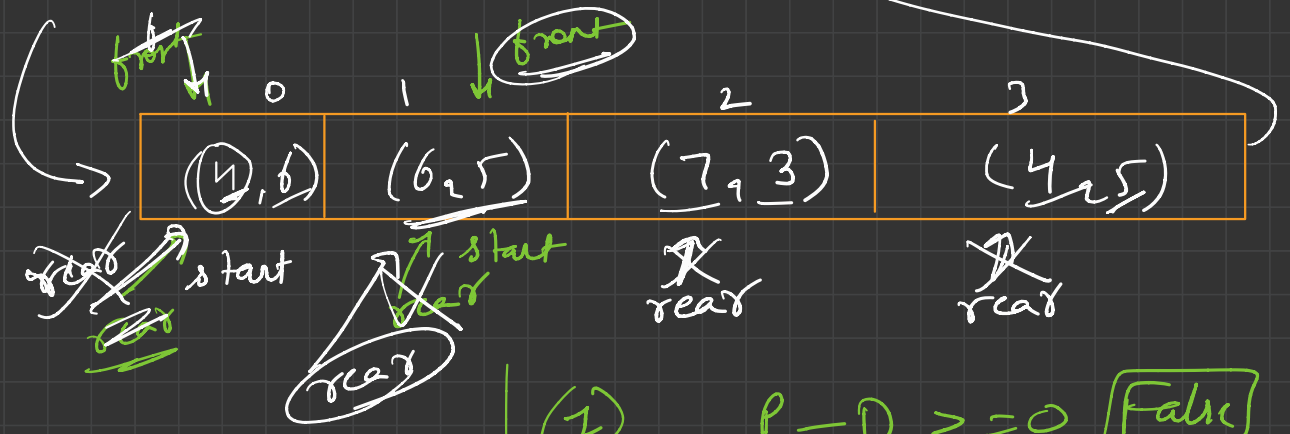
start = front

rear = front

B.C →

if (front == rear)

↳ (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 ↑

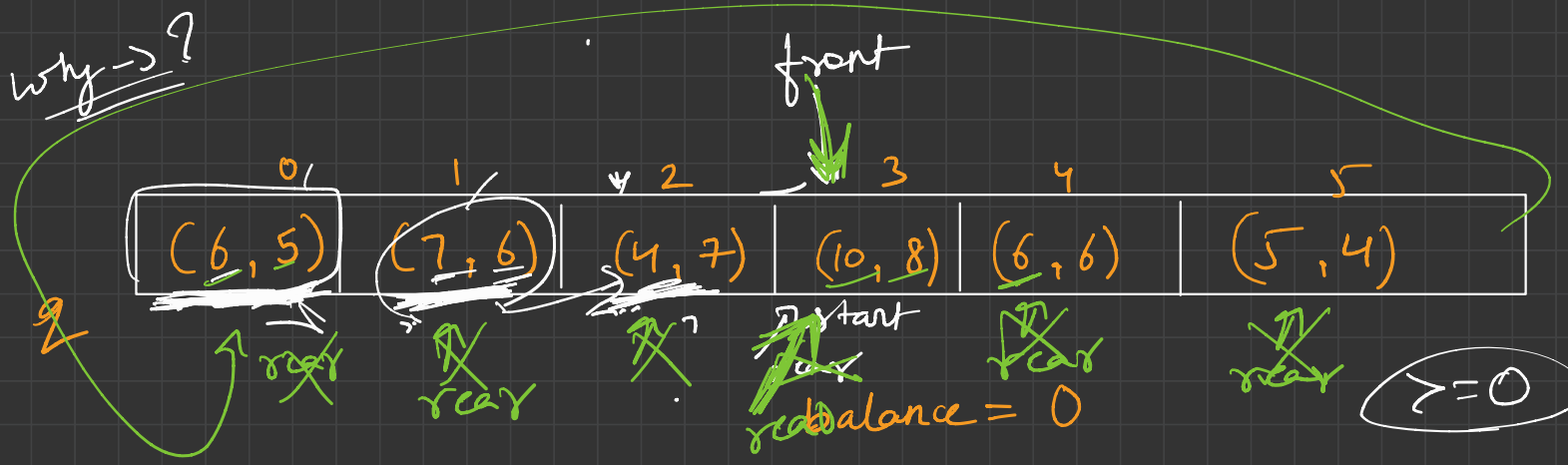
balance = 5 + 4 = 9 − 5

= 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

wont go

Block → 2 visit
&
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

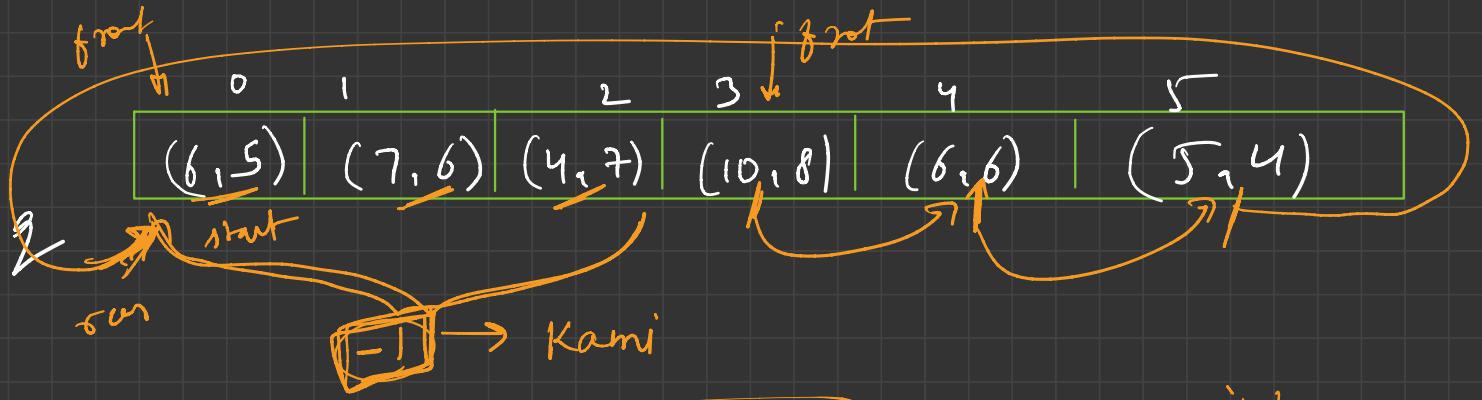
$= 4+7-6$

$= 11-6 = 5 >= 0$ TRUE

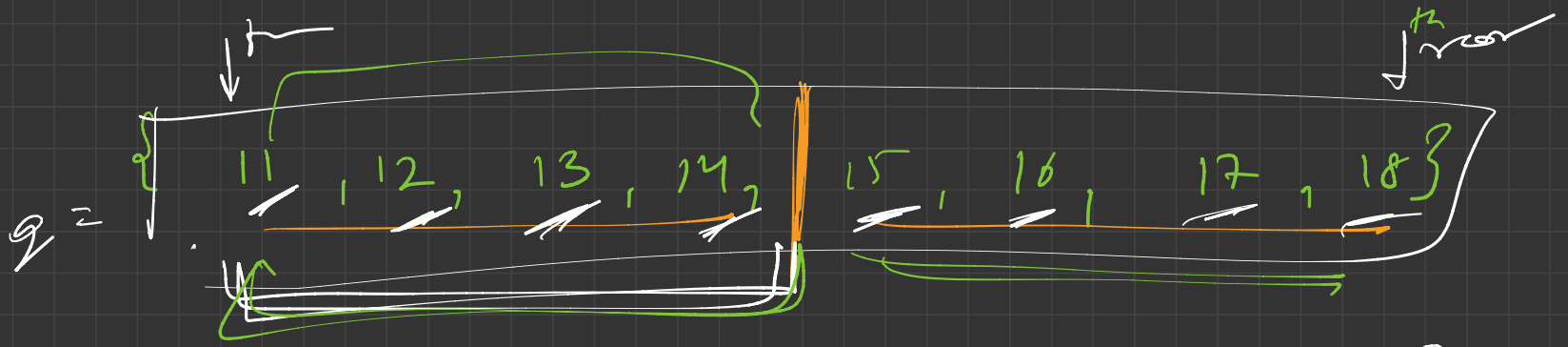
$= 5+4-7$

$= 9-7 = 2 >= 0$

TRUE



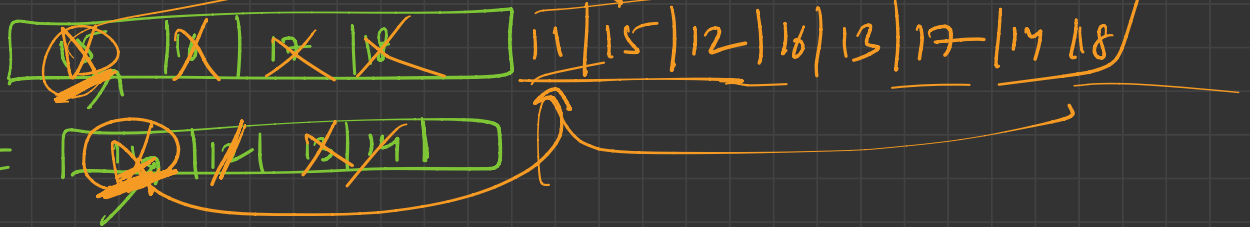
balance + Kami >= 0 → as possible



{ 11 , 15 , 12 , 16 , 13 , 17 , 14 , 18 }

#1

(I) fetch first half element from i/p qu & push into a new qu



(II)

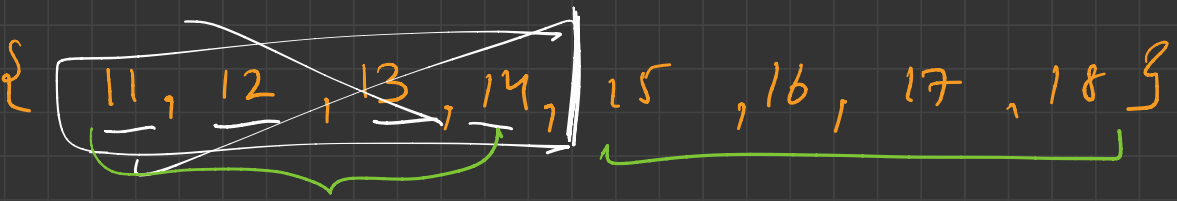
```
while (!newQ.empty())
{
    int val = newQ.front();
    newQ.pop();
    Q.push(val);
    val = Q.front();
    Q.pop();
    Q.push(val);
}
```

T.C -> O(n)

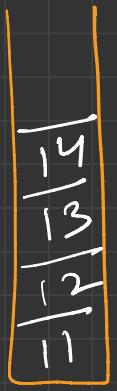
S.C -> O(n)

→ implement stack using Queue

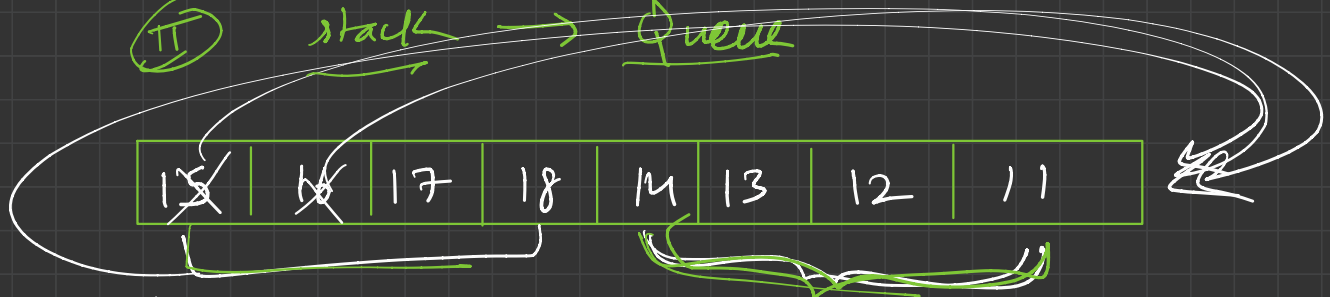
→ ——— Queue —— Stack



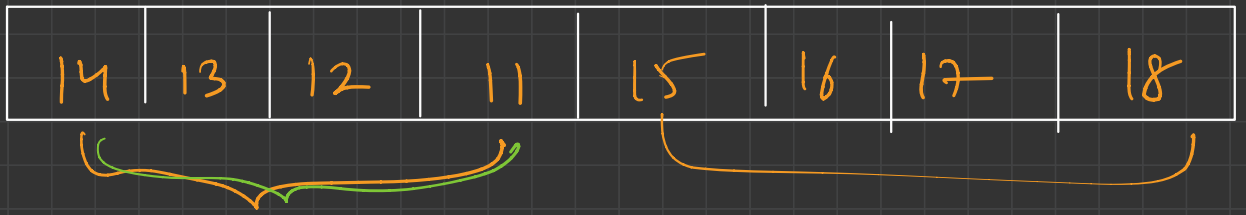
(I) first half of Q → Stack



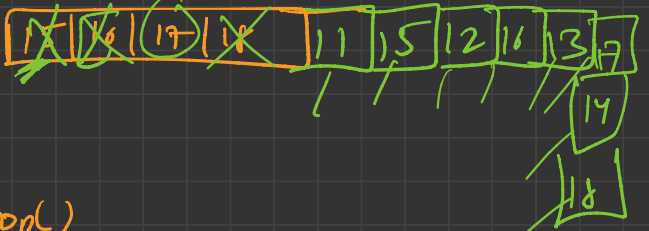
(II) stack → Queue



(III) first half of Q pop kro



(iv) first half of q → stack

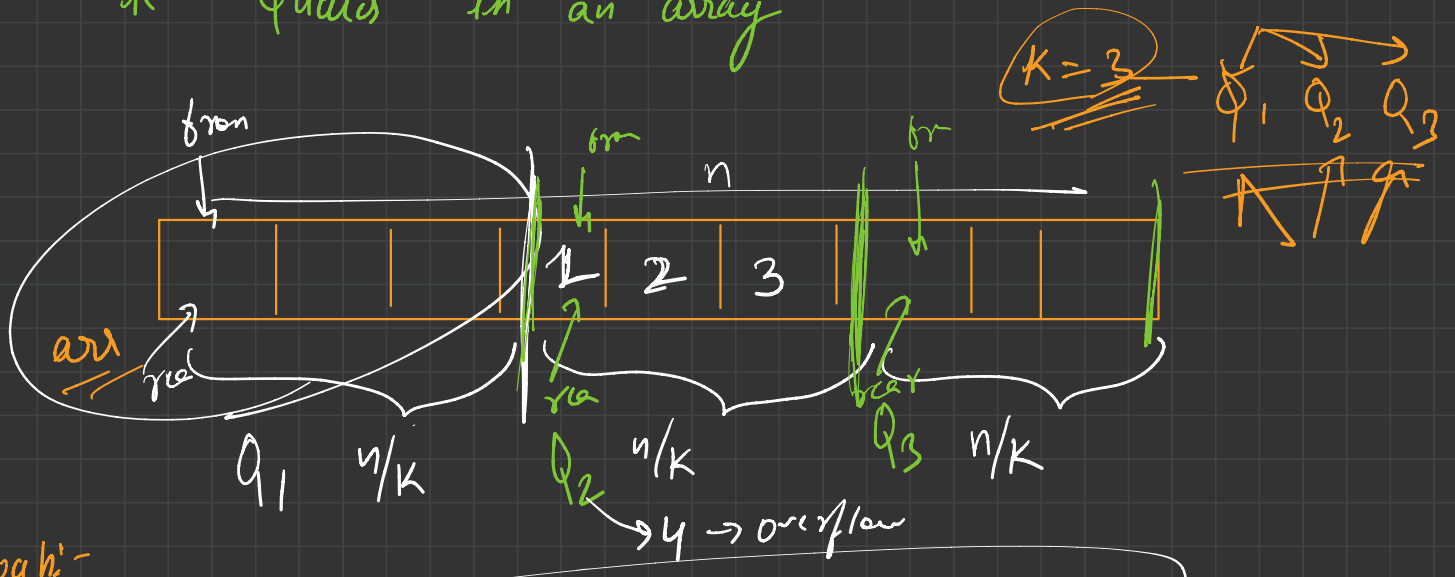



```
while (!s.empty)
{   int val = s.top()
    s.pop()
    q.push(val)
    val = q.front()
    q.pop()
    q.push(val
}
```

T.C → ?

S.C → ?

"K" Queues in an array

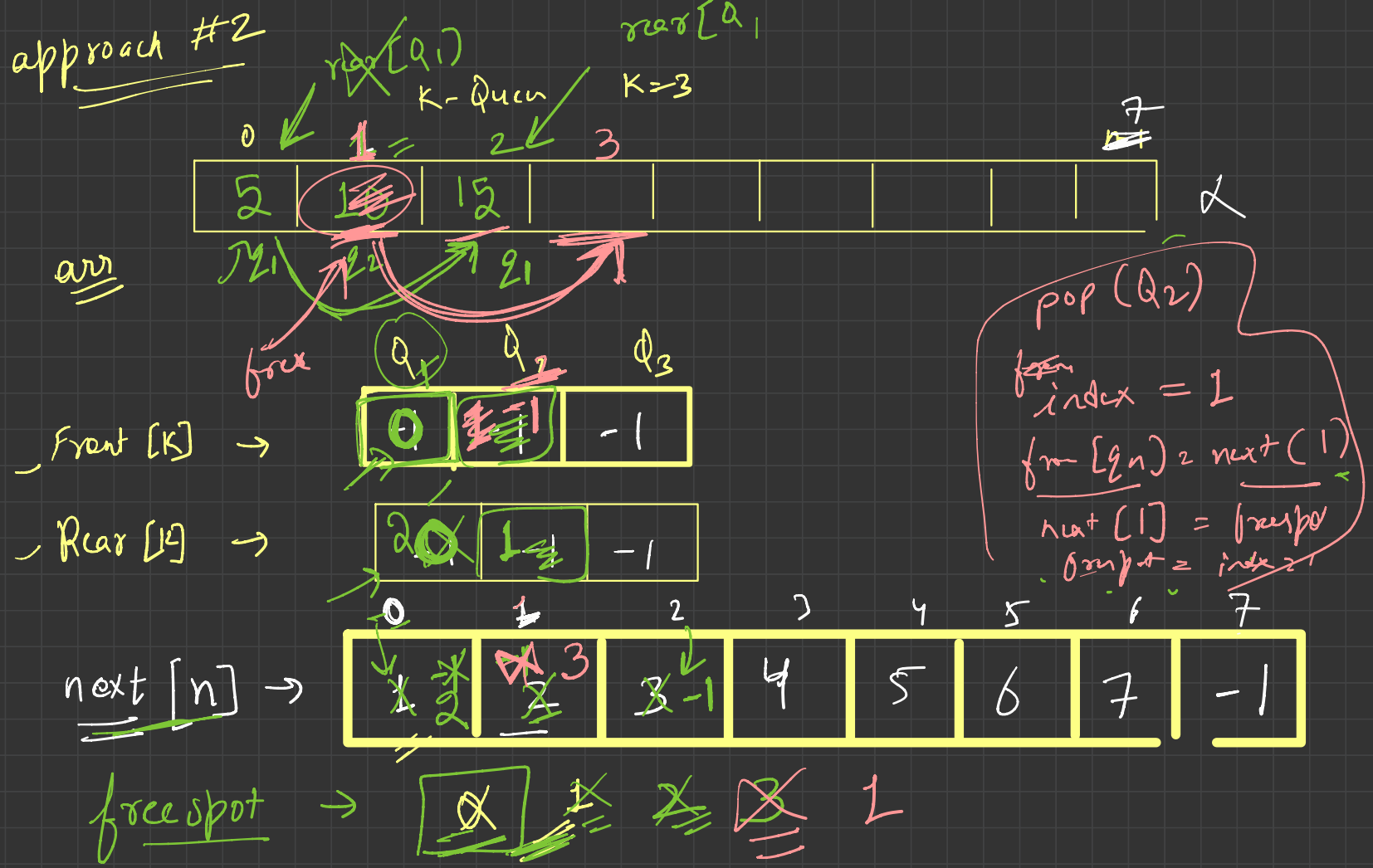


approach:-

array → k parts n → arr size

1 part → n/k

→ Space Optimally utilise


approach #2
rear[Q1]
K-Queue
rear[Q1]
K=3
0
1
2
3
7
5
10
15
arr
Q1
Q2
Q1
front
Q1
Q2
Q3
Front [K] →
0
1
-1
Rear [K] →
2
1
-1
pop(Q2)
index = 1
front[Q2] = next(1)
next[1] = freespot
freespot = index
next [n] →
0
1
2
3
4
5
6
7
1
2
3
-1
4
5
6
7
-1
freespot →
0
1
2
3
1

push:-

↳ Overflow check → free == -1

↳ // find index, where we want to insert

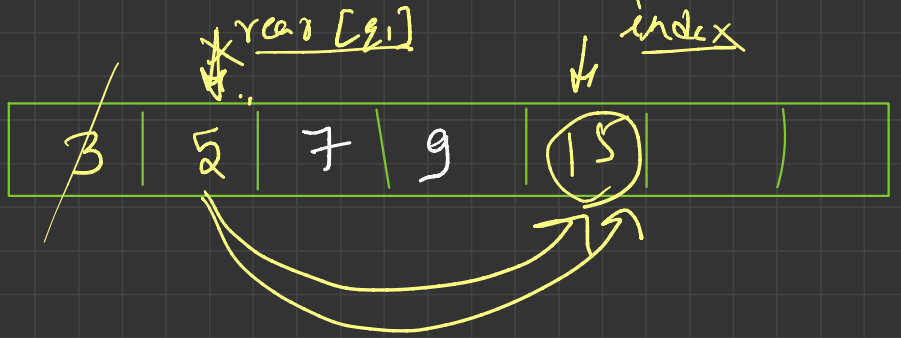
int index = freespot;

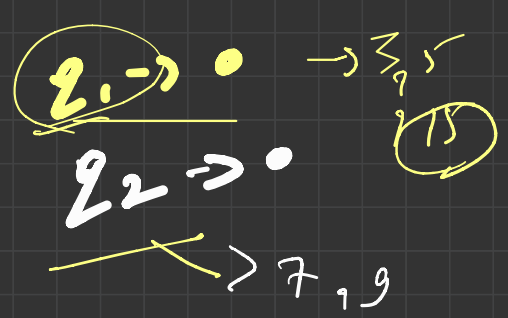
↳ //update freespot

freespot = next[index]

↳ // 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 ;

pop()

if Empty ⟶ Underflow

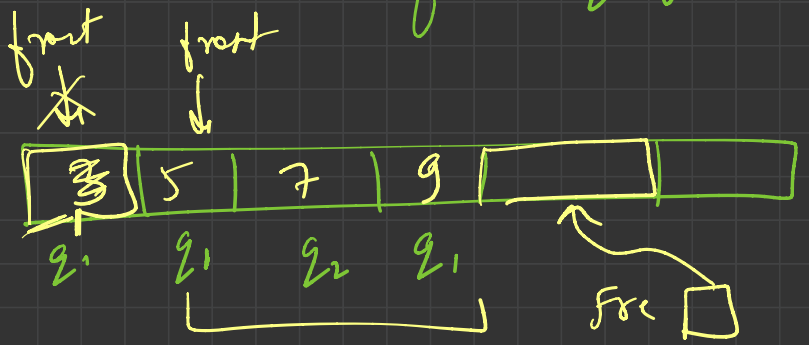
// find index

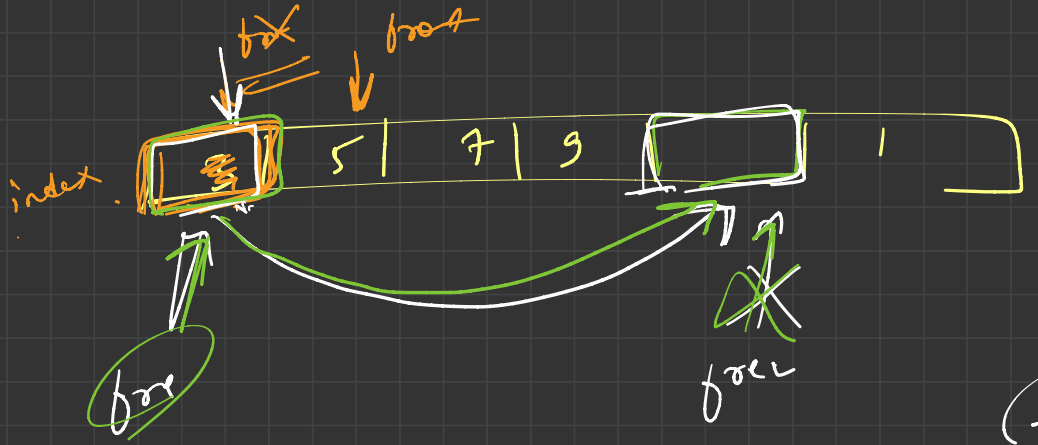
q_1 … q_{n-1}

int index = front[q_n]

// front ko aage karo

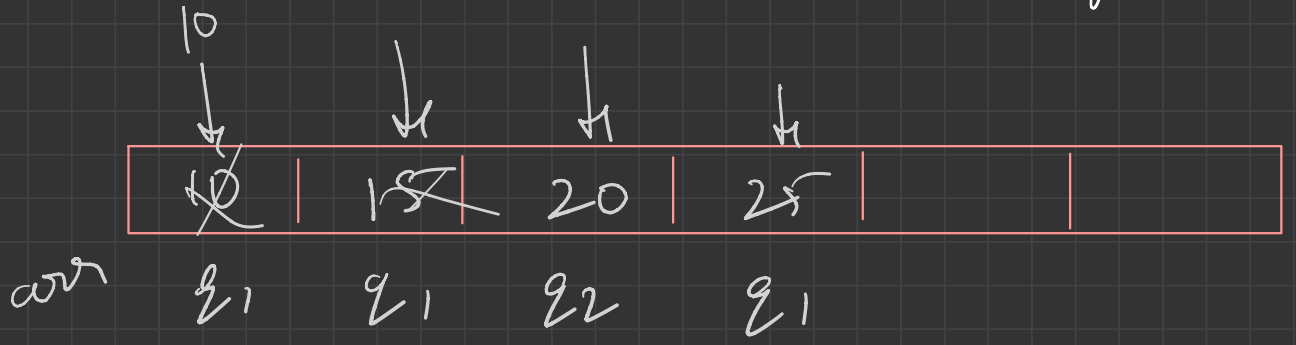
front[q_n] = next[index]

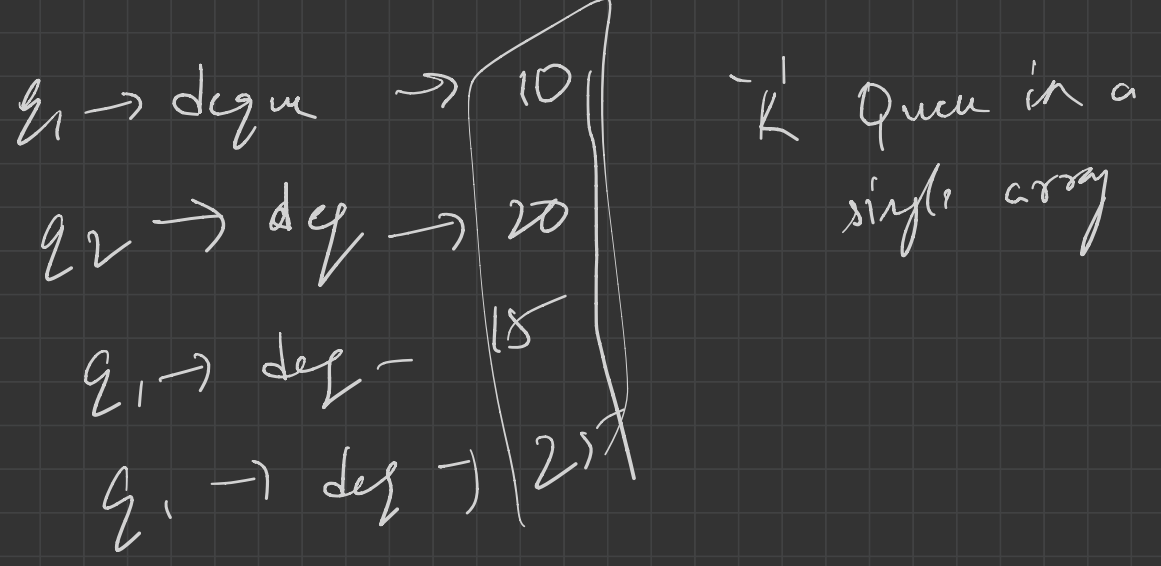




next [index] = free

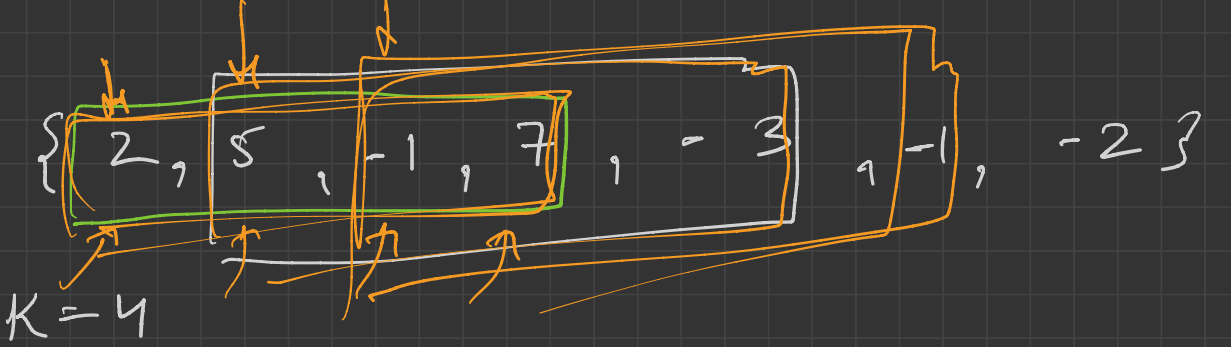
free = index





'k' Queue in a single array

DRY RUN



max = 7
min = -1 → (6)

max = 7
min = -3 → (4)

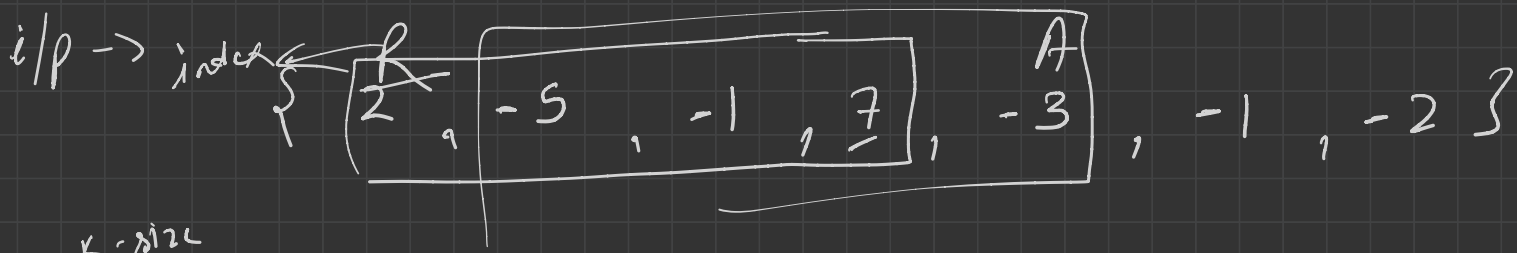
max = 7
min → -3 (4)

approach

For (0 → < n)
{
For (K times)
{ max
min
}
store sum in store

O(n * K)

Approach #2 ⇒ O(n) ≈ i



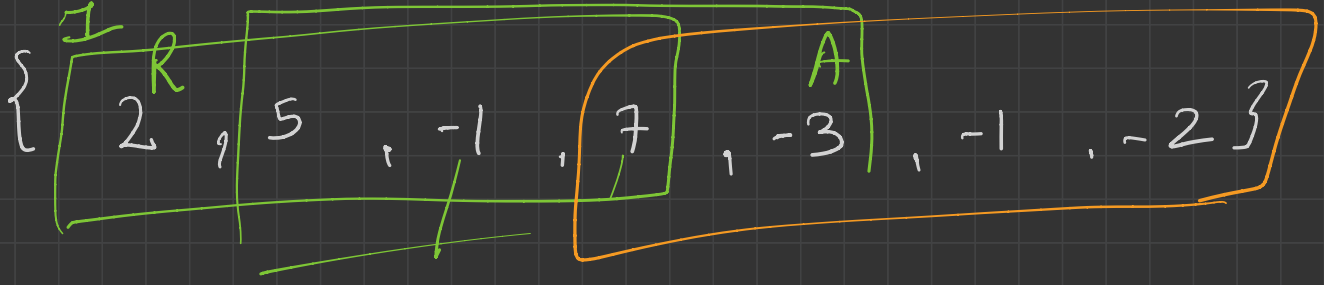
K-size

① deque → maxi → decreasing order me element honge

2 ✗ -5 ✗ -1 ✗ 7 → ~~if~~ maxi.front() → maxm element in K size window

② deque → mini → increasing order me element hoge

→ mini.front → minm in K sized window



for (i = K ; i < n ; i++)

{

max = arr[maxi.front()], min = arr[mini.front]

sum += max + min; =6

// next window

// Removal

while (maxi.front() - i >= K)

pop();

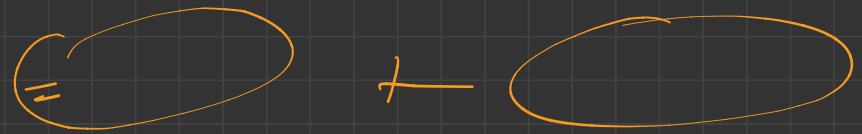
while { min

}

//Addition

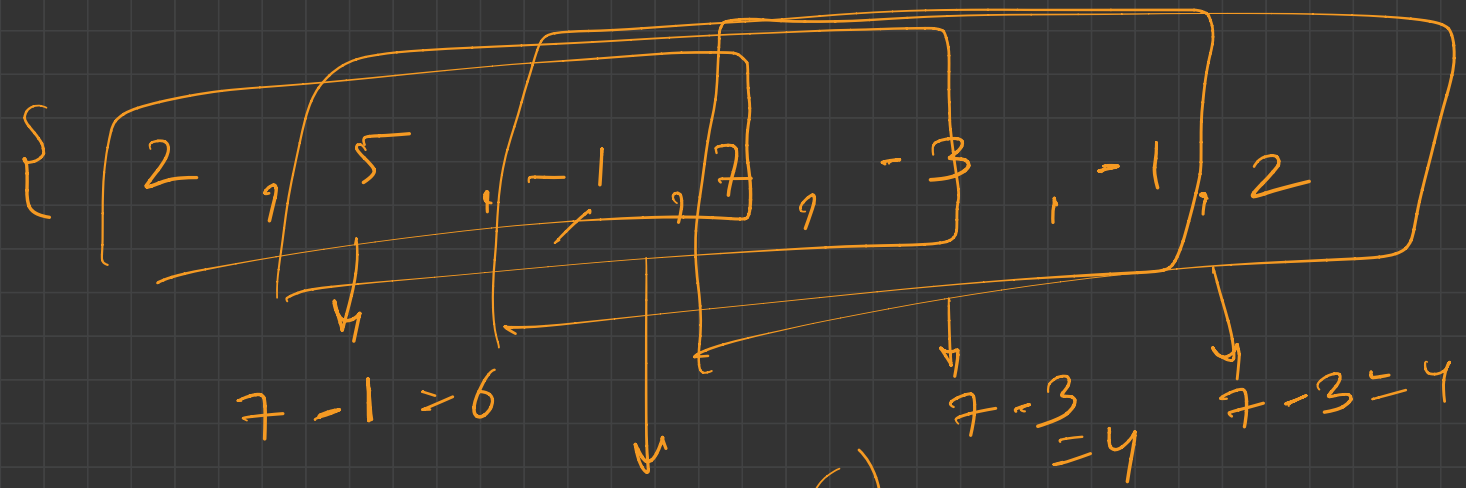
copy past

}

sum + = = + 

return sum;

}



$6 + 4 + 4 + 4 = 18$

T.C $\rightarrow O(n)$

S.C $\rightarrow$? H/W