Arrays

Q5] Union of 2 arrays.
return size of 2 arrays after union.
unordered_set<int> s;
insert first array elements, second array elements
return s.size().

Q6] Cyclically rotate array by one.
1 2 3 4 5 → 5 1 2 3 4.
OR can use deque. also.

```
t = arr[n-1];
for(int i=n-1; i>=0; i--){
  arr[i] = arr[i-1];
}
arr[0] = temp;
```

V.V. IMP.

Q7] Maximum sum of contiguous subarray [Kadane's Algo]

Method I → 3 nested for loops.

```
for(i=0; i<n; i++){
    for(j=i; j<n; j++){
        for(k=i; k<=j; k++){
            s = s + a[k];
        }
        if(s>max){
            max=s;
        }
```

TIME: $O(n^3)$

Method II → Store sum in array till that element

To get ans we do

	7	5	-13	5	10	-2	5
pre[] → 0	7	12	-1	4	14	12	17.

```
for(i=0; i<n; i++){
  for(j=i; j<n; j++){
  s = pre[j] - pre[i-1];
  if(s>ma)
    ma = s;
  }
}
```

TIME: $O(n^2)$

IMP. Q8

Meth
→ We
keep
and
if
if
in
in
fo

Page No.
Date

Method III → Kadane's Algorithm, → Similarly Max Product Subarray (Q 22)

We create int cursum & max sum.
keep adding elements in cursum
and in each iteration compare with maxsum
if it is more update maxsum
if cursum < 0 then make it 0

```
int max = INT_MIN.
int cursum = 0;
for (i = 0, i<n, i++) {
    cursum += a[i];
    if (cursum > max) {
        max = cursum
    }
    if (cursum < 0) {
        cursum = 0
    }
}
```

TIME: O(n)
SPACE: O(1)

Q8] Minimize the height [Minimize Maximum difference between heights]

k = 2, N = 4.
1 5 8 10 → 3 3 6 8 : diff bet^n smallest & largest
= 8 - 3 = 5.

k = 3, N = 5
3 9 12 16 20 → 6 12 9 13 17.
17 - 6 = 11

Sort array.
maxi = a[n-1] - k
mini = a[0] + k.
ans = a[n-1] - a[0].

```
for (i=0; i<n; i++) {
    ma = max(maxi, a[i]+k);
    mi = min(mini, a[i+1]-k);
    if (mi<0) {continue;}
    ans = min(ans, ma-mi);
```

Page No.
Date

Q9] Minimum number of jumps.

1 4 3 2 6 7

Iterate through array.
find max of maximum reach till where you can go
or i + arr[i].
set a int curr to know we have reached at our jump point. and count ++ & update curr = maxreach
if at end curr < n-1; we could not reach end of array ∴ return -1.

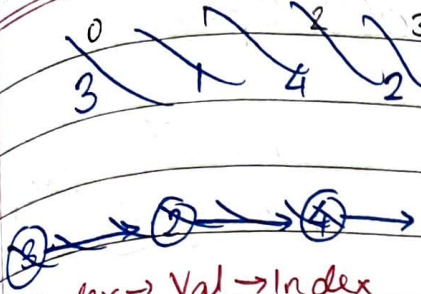
```
for (i=0; i<n-1; i++){
  maxreach = max (maxreach, i+arr[i]);
  if (i==curr){
    count++
    curr = maxreach;
  }
}
if(curr<n-1) return -1;
return count;
```

Q] Find duplicate Element in O(n) Time & O(1) Space.
array contains n+1 integers in range [1, n].
There is only 1 repeated no.
Solve w/o modifying array & use O(1) space.

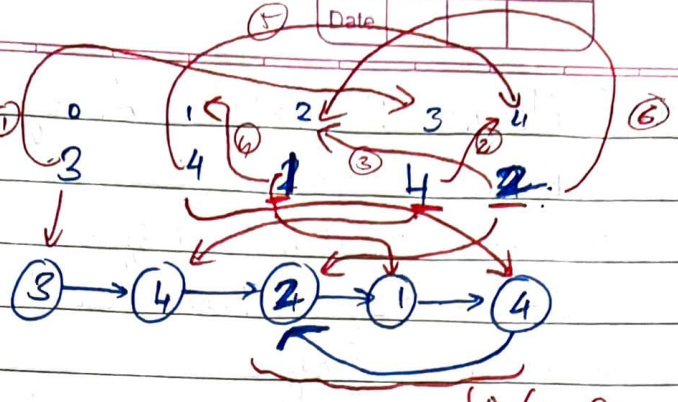
1 3 4 2 2.

Ans → 2.

Method ① → 2 nested for loops Time: $O(n^2)$
Method ② → Hash Map → Time O(n) Space → O(n).
Method ③ → Treat array like linked list.



Index → Val → Index
(always start from idx 0)



→ Why we always have cycle?
↳ we have (n+1) size array in 0 to N.
↳ elements are in range of 1 to N.

Eg n = 4

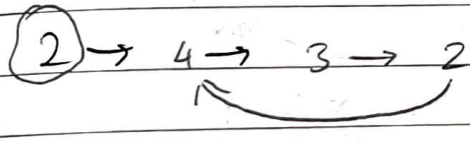
0	1	2	3	4
1	2	3	4	_

↳ This will always be repeating element (You can put 1/2/3/4).

→ First node will always be in cycle component.
↳ There is no value 0 element in array ∴ index 0 will have 0 incoming edge.

Eg

0	1	2	3	4
2	1	4	2	3.



→ Repeating element will always have outgoing edge to cycle intersection pt as they pt to same index.
→ To Find repeating element → Floyd's cycle detection
Fast & slow ptrs.
↳ Find intersection of cycle.

```
Fast, slow = a[0]
do{
  slow = a[slow];
  fast = a[a[fast]];
}while(slow != fast);
slow = a[0];
while(slow != fast){
  slow = a[slow]
  fast = a[fast]
}
return a[fast];
```

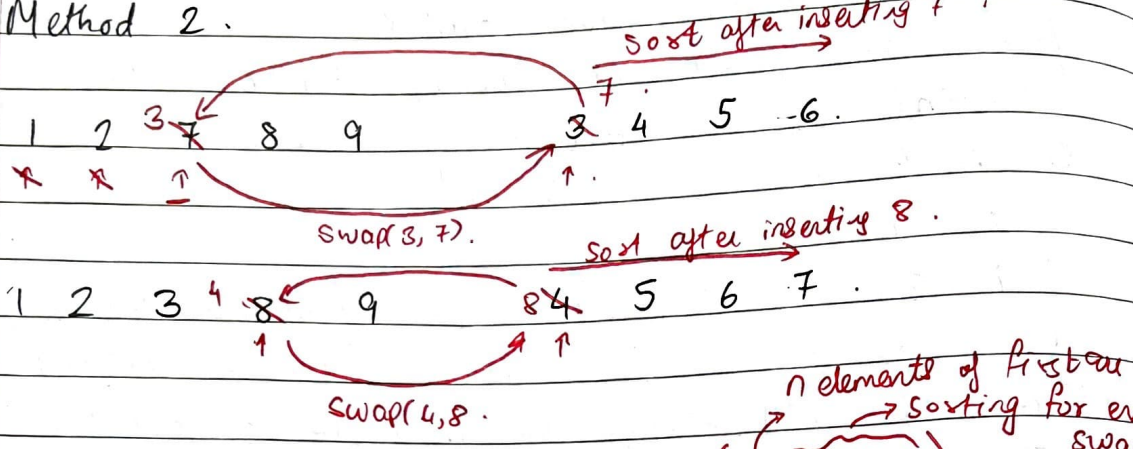
Q[2] Merge 2 sorted Arrays in O(1) Space.

→ Brute Force.
→ Create another array.
→ Store array 1 then arr 2 in it
→ Sort (new array)
→ put n elements in arr 1 & m in arr 2.

Time: O((m+n) log(m+n))
Space: O(n)

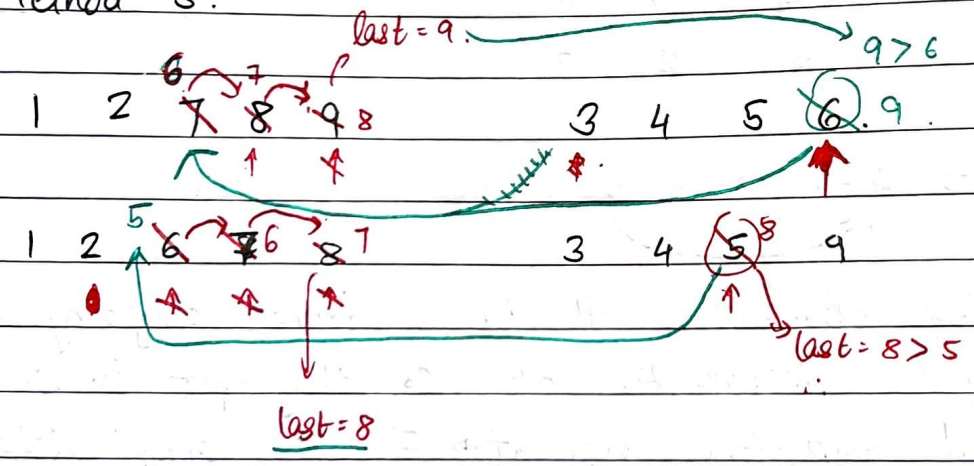
Method 2.

→



Continue above process: TIME: O(nm log m)
n elements of first arr → sorting for every swap!
SPACE: O(n)

→ Method 3.



Continue above process TIME: O(nm)
SPACE: O(1)

Method 4: GAP Method.

→ Take 2 ptrs with gap = $\frac{n+m}{2}$ [if odd take upper ans]
→ Compare both values if first ptr value > second ptr value, swap.
→ after first iteration is completed; gap = gap/2.
→ Continue the same process
→ if gap == 1 & process is completed arrays are sorted.

gap = 5

1 2 5 8 9 | 3 4 5 6 8 — gap = 5+4/2 = 4.5 = 5

gap = 3

1 2 3 4 7 | 5 3 6 9 8 — gap = 5/2 = 2.5 = 3

gap = 3

1 2 3 4 5 | 5 7 6 9 8 — gap = 3/2 = 1.5 = 2

1 2 3 4 5 | 6 5 7 8 9 — gap = 2/2 = 1

1 2 3 4 5 6 7 8 9. ⇒ SORTED.

TIME: $(n+m)\log(m+n)$

SPACE: $O(1)$

Q13] Merge Intervals

[(1,3), (2,6), (8,10), (15,18)]

Ans → [(1,6), (8,10), (15,18)]

Create stack<pair<int, int>>

Cost input vector

Push first element in stack (st.push{a[0][0], a[0][1]})

Compare next elements second & st.top().second

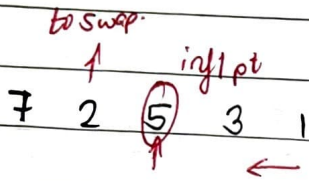
```
if (st.top().second < a[i][1]){
    x = max(st.top.first, x)
    y = (st.top.second, y)
    st.push({x, y})
}
```

Empty stack & Store in vector

reverse and vector.

Page No.
Date

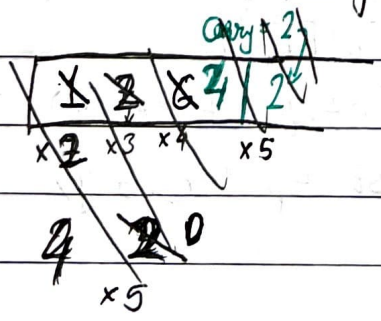
Q Next Permutations → like in dictionary.
find next lexographical permutation
1 2 3 → 1 3 2 → 2 1 3 → 2 3 1 → 3 1 2 → 3 2 1.
if input is 3 2 1 [last permutation] return 1 2 3 (sorted).



Start iterating from behind till a[j] > a[j-1] i.e you find first element greater than its prev element.
i.e here 5 is > 2 this is inflection pt
a[j-1] i.e (2) will be number to be swaped
we will check if inflpt is 0 i.e input is last permutation ∴ sort the array & done.
if not
swap 2 with the next minimum greater element
i.e 3
∴ 7 3 5 2 1 → sort rest of elements

∴ 7 3 1 2 5 → ANS.

*Q] large factorials
create vector & push 1.
& find it. See code you'll understand.



Page No.
Date

Median of 2 sorted arrays of same size.

1 2 3 6
4 6 8 10

count = 4 → size/2.
m1 = 4
m2 = 6
$\frac{4+6}{2} = 5$.
TIME: O(n) SPACE: O(1).

Method 2 [Optimal]

Recursive

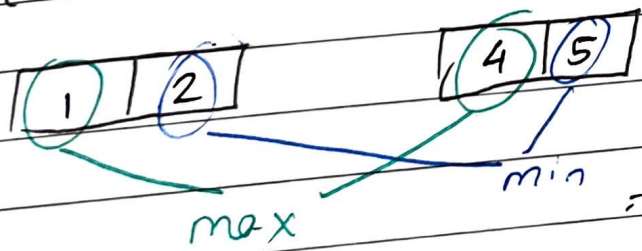
1 2 3 6 → Median = m1
4 6 8 10 → Median = m2

if m1 == m2 return m1 or m2 as that same will be median

→ if m1 > m2
m1 ke right ke no use
m2 ke left ke no use.

→ if m2 > m1
m1 ke left ke no use.
m2 ke right ke no use

Base Condition:
→ if (Ea - Sa ≤ 1) i.e size = 2.



$= \frac{2+4}{2} = 3$.

Sa code you'll understand!

Median of 2 Sorted Arrays of different size. Good sum.
See video of Striver