ARRAYS

A] KADANE'S ALGORITHM:

↳ Idea is to maintain max subarray sum at every idx i & update it with max found so far.

↳ At every idx i we have 2 choices:

i) Add that ele to our curr max (Helpful when sum is non-ve)

OR

ii) Start new subarray from idx i

Pseudo Code:

```
Kadane(arr, n)
    max_sum, curr_sum = -1e9, 0

    for ele in arr:
        curr_sum = curr_sum + ele
        max_sum = max(max_sum, curr_sum)
        curr_sum = max(curr_sum, 0)

    return max_sum.
```

T.C. O(n)
S.C. O(1)

Kadane's algo needs atleast 1 +ve element, if not we return 0,
↳ for this max_sum = 0

else we return greatest -ve ele present.
↳ max_sum = -1e9

Q.1 Max Subarray Sum:

→ Direct appln of kadane's algo.

→ Thing to note, for all -ve array we return 0, instead of greatest -ve ele.

∴ Initialize max_sum = 0

T.C = O(n)
S.C = O(1)

In Few lines:

```
max_sum, curr_sum = 0, 0
For ele in arr:
    curr_sum = max(curr_sum+ele, ele)
    max_sum = max(max_sum, curr_sum)
return max_sum
```

Q.2 Flip Bits:

↳ Flip a subset of bits such that overall number of 1's in the array increases, if it is possible. finally return count of 1's

e.g:-

① 1 1 0 0 1 → 1 1 1 1 1 ⇒ 5

② 1 1 1 0 ⇒ 1 1 1 1 ⇒ 4

③ 0 0 1 0 0 ⇒ 1 1 0 1 1 ⇒ 4

④ 0 0 0 0 ⇒ 1 1 1 1 ⇒ 4

⑤ 1 1 1 1 ⇒ No flip ⇒ 4

⑥ 1 0 1 0

↳ 1 0 1 0 ⇒ 1 0 1 1 ⇒ 4

1 0 1 0 ⇒ 1 1 1 0 ⇒ 4

1 0 1 0 ⇒ 1 1 0 1 ⇒ 4

⇒ If you closely observe which sections we are choosing as a sub-array to flip, you will see that we are choosing only that section where we have more no. of 0's & less no. of 1's (e.g 1, 2, 4) or in other words those section where the diff. bet count of 0's & count of 1's is maximum (e.g: 3)

⇒ Thus we are trying to find a subarray which has largest diff bet 0's & 1's (#0 - #1) is max.

⇒ Largest diff. is nothing but a value which we want to inc. this can be considered similar to a problem where we are trying to maximize subarray sum ⇒ kadanes!

Can we somehow map the diff. finding → max sum?

⇒ we cannot directly convert/apply Kadane's algo as for e.g (4) summing up would result in only 0 so no length would be found. so, can we try substitution?

⇒ What if we convert all 0's to 1's ? :- That would work for e.g (4) but for other ex. there would be no diff betn real 1's & substituted 1's.

⇒ what if we replace 0's with 1's & 1's with 0's? That won't help either since we won't be able to get exact diff. betn #0's & 1's as any_no + 0 ⇒ same no.

⇒ What if we replace 0's with 1's & 1's with -1 ? This might work, as considering any arbitary subarray of 0's & 1's, they will be converted to 1's & -1 so more the zeros in subarray (more 1's) here our sum will inc & with each 1 found (-1) our sum will reduce. The max sum found so far will tell us exactly how many bits to replace.

⇒ This added to our original 1 count ⇒ will give total #1's

Thus this problem is basically,

Replacement + Kadane's Algo.

Ex:

(1) 1 1 0 0 1 → -1 -1 [1 1] -1
↳ flip this

max_sum = 2 (sum in original array)
orig_one = 3
∴ Ans = 3 + 2 = 5 (1 1 1 1 1)

(2) 0 0 1 0 0 ⇒ 1 1 -1 1 1 → flip this section

max_sum = 3 (3 1's & one -1)

one_count = 1

∴ Ans = 3+1 = 4

Pseudocode

```
FlipBits(arr)
one_ct, max_diff, curr_diff = 0, 0, 0

for ele in arr:
    if ele == 1:
        one_ct += 1
    replaced_value = 1 if ele == 0 else -1
    curr_diff = max(curr_diff + replaced_value, replaced_value)
    max_diff = max(max_diff, curr_diff)
return max_diff + one_ct
```

(replacement; Kadane's; curr_sum; max_sum)

Q3 max subarray after k concat

⇒ Concat array k times & find max sum in it.

Soln: we could just create new array & apply kadane's algo.

But do we really need to create a new array?

All we need is when we reach the last ele we should get back to first element. So when ever we need to move in cycles i.e. 0 → 1 → 2 → 3 the best way is to use "mod".

⇒ Let's say we have array of 4 ele & we concat it 3 times, if we really concat then there would be 12 elements.

⇒ But we really don't need to concat elements as elements are repeated after a certain interval & they all will map to some idx. lets see now!

e.g:- [15, 21, 27, 35] we concat this 3 times

0	1	2	3	4	5	6	7	8	9	10	11
15	21	27	35	15	21	27	35	15	21	27	35

If we try and arrange them one below another

0	1	2	3
15	21	27	35
4	5	6	7
15	21	27	35
8	9	10	11
15	21	27	35

we see idx 0, 4, 8 map to idx 0 (% 4 = 0)

idx 1, 5, 9 map to idx 1 (% 4 = 1)
idx 2, 6, 10 map to idx 2 (% 4 = 2)
idx 3, 7, 11 map to idx 3 (% 4 = 3)

Thus all we need to access all ele. is to run a loop n*k times where n → len. of array & k is # times to concat & each idx will be mod with len. of array n.

Thus this problem is MOD + kadane's.

Pseudocode:

```
max_sum, curr_sum = -float('inf'), 0
for i in range(n*k):
    curr_sum += arr[i%n]
    max_sum = max(max_sum, curr_sum)
    curr_sum = max(curr_sum, 0)
return max_sum.
```

Q.4 max sum Rectangle:

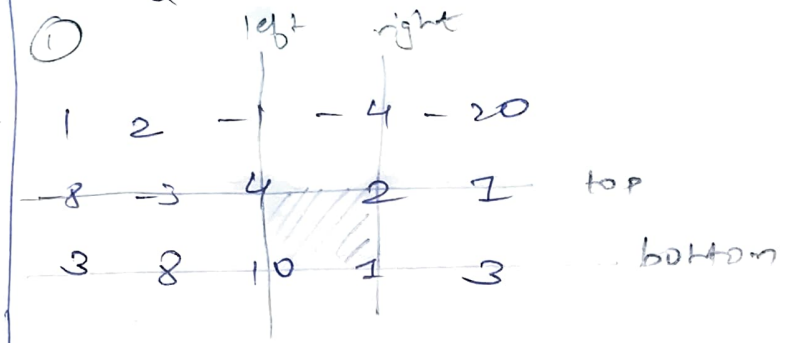
↳ Given a matrix with 'N' rows & 'M' cols. find a rectangle which has max sum.

⇒ For detailed analysis you can watch video on Back2back SWE, link will be in code also.

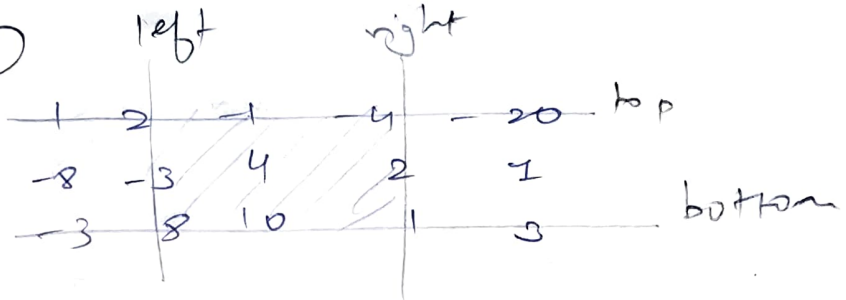
⇒ Since we need to find max sum, it is clear that we need to use Kadane's algo, but Kadane's algo work only in 1D & rectangle is 2D, so how can we convert 2D values to 1D
↳ (or maps)
so that we can use Kadane's?

⇒ Before that let's see how we can generate all possible rectangles. To generate a rect. we need two vertical & two horizontal lines.

①



②



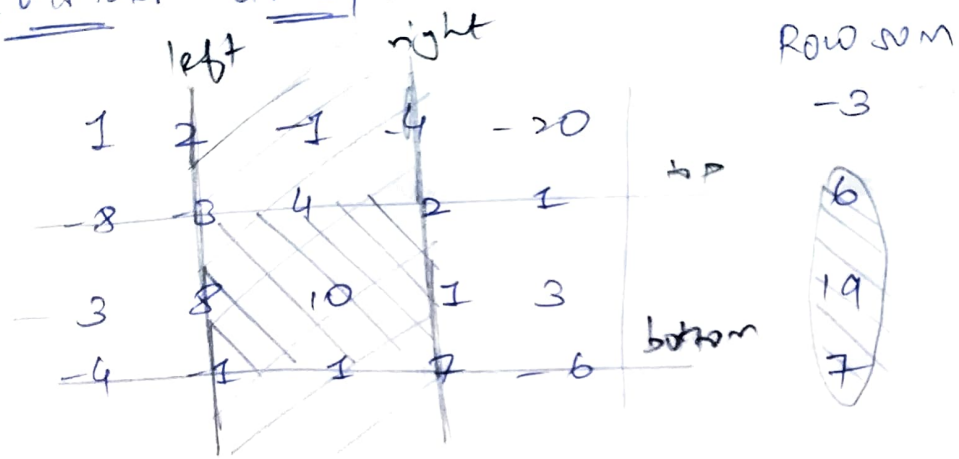
Thus we see, if we move the vertical bars from left end of array to right end we will scan the matrix col. wise

If we move horiz. bars from top to bottom we scan matrix row wise. Thus comb? of 4 movements we can generate all rectangles.

Note! Single ele is also rect. Square is also a rectangle.

⇒ Now we have found a way to generate all possible rectangles, only way is to map 2D → 1D.

Consider array:



If we consider sum of ele. for each row bet left & right, in a way we have spanned that entire section. we also get a 1D array denoting row_sum.

Now we can apply Kadanes algo on row sum to get max value

for rectangle. only modifn will be to check with max sum found so far after applying kadane's multiple times to get ans. we don't need to know top & bottom positions as we only need max value.

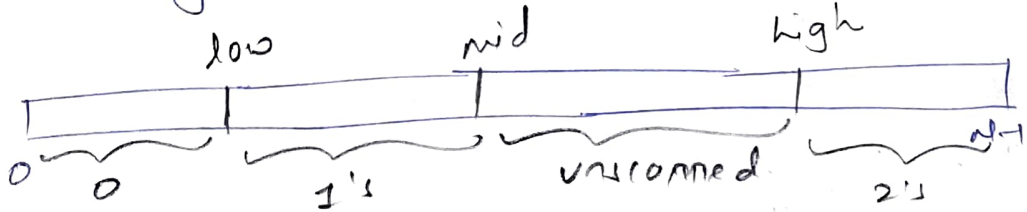
Pseudocode:

```
max sum Rect (om, rows, cols)
    max_rect_sum = -float('inf')
    for left in range(cols):
        row_sum = [0] * rows
        for right in range(left, cols):
            // comm. sum for each row span
            for i in range(rows):
                row_sum[i] += arr[i][right]
        max_row_sum = kadanes(row_sum)
        max_rect_sum = max( , )
return max_rect_sum.
```

3] DUTCH National Flag ALGO:

↳ Also known as 3-way partitioning algorithm.

↳ Image:



mid is a traverser which scans entire array.

if A[mid] = 0, swap low, mid,
low++
mid++

if A[mid] = 1, mid++

if A[mid] = 2, swap mid, high
high--

Q.1 Sort 0, 1, 2

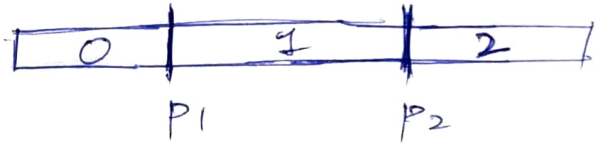
↳ simple appln of Dutch flag algo

```
sort(arr, n):
    low, mid, high = 0, 0, n-1
    while mid <= high
        if arr[mid] == 0:
            swap low, mid
            low++
            mid++
        else if arr[mid] == 1:
            mid++
        else if arr[mid] == 2:
            swap mid, high
            high--
    return arr.
```

Q. 2 Quicksort using Dutch Flag:

↳ Simple quicksort does only one part i.e, divides array into 2 equal parts, each time partition is called.

↳ If we look at the way dutch flag functions it has 2 parts in the array.



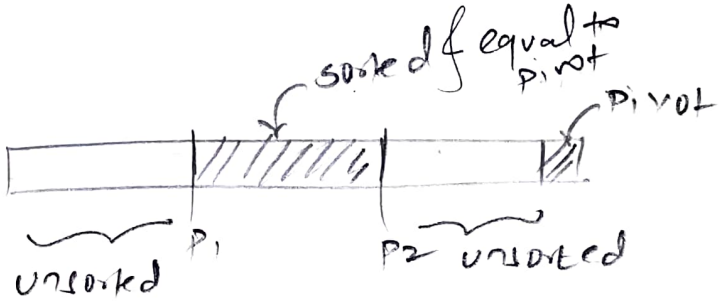
So if we have to use this as a modified part method in quicksort we are going to get 2 partition & we must be able to sort the array around it.

↳ Why do we need a 3 way quicksort? Rather why dutch flag algo to be implemented in quicksort?

If we observe the nos. bet p1 & p2 we see those are equal to a same no. let's say 1. So if we have arrays where majority elements are repeating, we can use this method.

→ The idea will be whatever pivot we choose, ele bet p1 & p2 will be equal to that pivot & we only need to sort ele left of p1 & right of p2.

→



Pseudo code:

```
modified_quicksort (arr, low, high)
    if low >= high:
        return

    p1, p2 = partition (arr, low, high)
    modified_quicksort (arr, low, p1)
                       (arr, p2, high)

partition (arr, low, high)
    pivot = arr[high]
    mid = low
    while mid <= high:
        if arr[mid] < pivot:
            swap arr[mid], arr[low]
            low++
            mid++
        elif arr[mid] == pivot
            mid++
        else:
            swap arr[mid], arr[high]
            high--
// while ends
// arr bet low, high is sorted.
// sort to left & low & right of high
p1 = low - 1
p2 = high + 1
return p1, p2
```

c) SEARCHING & SORTING

Q.1 search in rotated & sorted array

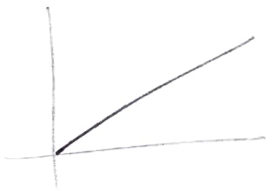
↳ Here we are given a sorted but rotated array

↳ Since array is sorted, to search first thing that must come to our mind is to use binary search. But the array is rotated too, so does this have an effect? Let's check

↳ simple sorted array:

1 2 3 4 5 6 7

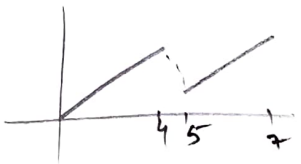
Graph:



Rotated array: Order

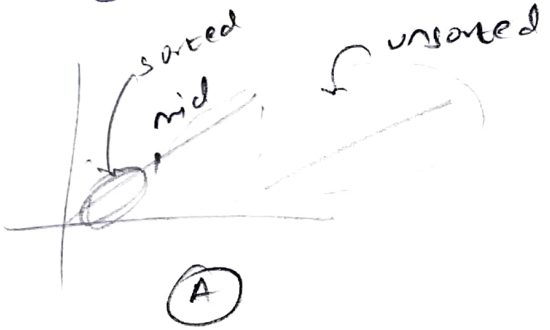
~~5~~ ~~6~~ 5 6 ~~7~~ 1 2 3 4

Graph:

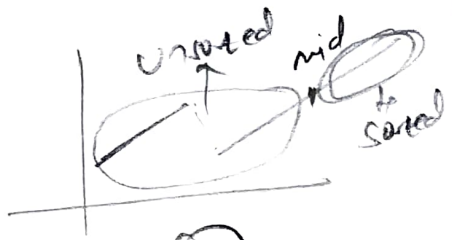


Thus we see the graph for sorted array is complete one but for rotated one it has breaks i.e, the order of growth suddenly decreases while it was increasing.

↳ So to apply binary search we must check whatever mid we have found w.r.t. to that where is sorted sec. There are 2 cases:



(A)



(B)

Find the sorted sect is it at left/right to mid

1) arr[low] <= arr[mid] ⇒ LHS is sorted

else

RHS

2) search accordingly in corr. sect.

Pseudocode:

```
rotated_bsearch(arr, target, low, high)
    mid = low + (high-low)//2
    if arr[mid] == target:
            return mid

if low <= high:  // LHS sorted
    if target >= arr[low] &
        target < arr[mid]:
        return rot_bsearch(arr, target, low,
                           mid-1)
    else
        ——— ———(arr, target, mid+1, high)

else:
    if target > arr[mid] & target <= arr[high]
        return rot_bsearch(arr, target, mid+1,
                           high)
    else:
        ——— ———(arr, target, low, mid-1)

return -1
```

Q. 2 form a triangle.

↳ Use property: Sum of 2 smallest sides must be greater than 3rd side. if so Δ poss.
else, not possible.

↳ make sure array is sorted at first.

Pseudocode:

```
def triangle(arr):
    arr.sort()
    n = len(arr)
    for i in range(n-2):
        for j in range(i+1, n-1):
            for k in range(j+1, n):
                if arr[i]+arr[j] > arr[k]
                    return True
                break
            break
    return False
```

Q-3 1st & last Occurence of ele

↳ Simple modif^n to binary search since array is sorted.

Pseudocode: mid = low + (high - low) // 2

```
first_occurence(arr, target, low, high)
  if arr[mid] == target &
     (mid == 0 or target > arr[mid-1]):
     return mid

  if low <= high:
     if target > arr[mid]:
        return f_c(arr, target, mid+1, high)
     else
        ____ ____(__, __, low, mid-1)

  return -1
```

```
last_occurence(arr, target, low, high, N)
  if arr[mid] == target &
     (mid == N-1 or target < arr[mid+1]
     return mid

  if low <= high:
     if target < arr[mid]
        ret l_c(arr, target, low, mid-1, N)
     else
        ____ (____, mid+1, high, N)

  return -1
```

Q.4 Count smaller or equal ele. in array.

↳ Here we are given 2 arrays, for each ele in arr 1 we have to find #ele <= that element.

→ Since we have to search we can think of applying binary search but arrays are not sorted, also we need to search in arr B. ∴ sort array B first.

Ex:-

A = [3]

B = [-1 1 0 4 2 5]

↳ sort B

↳ B = −1 0 1 2 4 5

0 1 2 3 4 5

1) mid_idx = 0 + (5-0)/2 = 2, low = 0, high = 5

∴ mid_ele = B[2] = 1

Now 1 < 3, so we have scope to look for more ele which are <3

∴ low = mid_idx+1 = 3

2) mid_idx = 3 + (5-3)/2 = 4, low = 3, high = 5

mid_ele = B[4] = 4

4 > 3 ∴ It means from 4 onwards no ele. are present which are ≤ 3

∴ we need to check bet low & mid_idx-1

∴ high = mid_idx-1 = 3

3) ∴ mid_idx = 3 + (3-3)/2 = 3, low = 3, high = 3

∴ mid_ele = B[3] = 2

since 2 < 3, we have scope so look ahead. ∴ low = mid_idx+1 = 4

But, low = 4, high = 3 ∴ low & high have crossed boundry

∴ we stop.

∴ we observe #ele, the index till where ele ≤ 3 are present is pointed by high.

∴ #ele = high+1 (0' based idx)

Pseudocode:

For each ele. in arr A apply foll. func?:

```
modified_bsearch(arr, low, high, target):
    while low <= high:
        mid = low + (high-low)/2

        if arr[mid] <= target:
            low = mid+1

        else:
            high = mid-1
    return high+1
```

Q.5 Find best insert posn in sorted array.

→ Here we have to find given an ele. at what posn it can be inserted in sorted array. If ele is present ~~also~~ return it's idx coz that is the posn.

→ Since array is sorted we can use bsearch to look if ele is present or not. Also we can use the algo in previous ques, to get idx of insertion, where `high+1` used to denote # ele < given ele, here it will denote the idx of insertion.

Pseudocode:

```
InsertPosn(arr, n, target)
    low = 0, high = n-1
    while low <= high:
        mid = low + (high-low)/2

        if arr[mid] == target:
            return mid
        elseif arr[mid] < target:
            low = mid+1
        else:
            high = mid-1
    return high+1
```

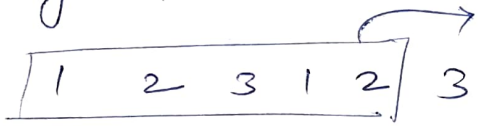
D] PREFIX AND SUFFIX

Q:1 sum of infinite array

↳ Here we are given an array, which will keep on repeating itself your task is to find the sum bet^n 2 given pos^n L & R. (1 based indx)

e.g.:- A=[1, 2, 3]

Range: 1 5

∴ 

Soln: For time being let's assume we have a finite array. & we want to find sum bet^n any 2 arbitary positions L & R multiple times.

Always looping over L & R will not be an optimal soln. what we can do is we cummulative sums (prefix sums) to get sum bet^n ranges.

A = [2, 5, 7, 9, 10, 15, 13] (indices 0 1 2 3 4 5 6)

prefix-A =

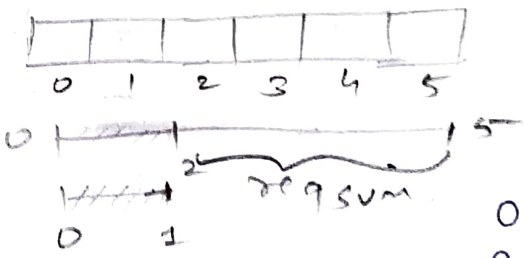
2	7	14	23	33	48	61
0	1	2	3	4	5	6

→ Let's say we have to find sum of entire array we directly get it at last idx = 6 ⇒ sum = 61

→ Let's say we want sum bet^n idx 2 to 5: sum is 7 + 9 + 10 + 15 = 41

How can we get this from prefix array?

Since prefix array contains cummulative sums, ∴ At idx = 5, we have sum from 0 to 5, at idx = 1 we have sum from 0 to 1. ∴ If we subtract them we get sum 2 to 5.



0 - 5 ⇒ 48
0 - 1 ⇒ −7
∴ 41

we saw how we can have sum between ranges using prefix sum. How can we now scale this up for infinite array?

→ If you remember we had a question in Kadane's Algo where we need to concatenate array k times & find max sum over it. What we did there was we needed to get to start of array so we just used mod to roll over to start, we shall use similar concept here.

→ for each given range L & R, we will check how many times we need to completely traverse the array using division. So those many times we will add complete sum of array.

→ There might be case we need to traverse some additional elements too, that number we will find using mod, sum till those element can already be found using prefix sum.

e.g:- Suppose we have array of length 5 & we want sum in range 7 to 19

⇒ left_range = 7-1 = 6 [prefix sum method to cal. sum at 1 less idx]

Calculate full traversals
↳ 6//5 ⇒ 1 ∴ for 6 ele we need to scan entire array 1 time, as the array has 5 ele. Now, we have additional 1 ele left that can be found as: 6%5 = 1

∴ Total we need 1 full traversal & 1 ele. extra. ⇒ 1 * sum of array + prefixsum[1]

⇒ Right_range = 19

full_traver = 19//5 = 3

remain_traverse = 19%5 = 4

∴ 3 * sum of array + prefixsum[4]

sub ⇒ ans

Pseudo Code:

```
SumInRanges(arr, n, queries, q):
1) Calculate prefix sum for entire array.
2) For each query:
    left = query[0]-1   // 0based idx
    right = ---- -1      in array
  if left == 0: // sum from start
    // we need to see only rightend
       then.
    full_trav_right = right//n
    remain_trav_right = right%n
    sum_right = prefix_sum[-1] * full_trav_
                right + prefix_sum[rem_trav_
                                   right]
    append this sum.
  else:
    left = left-1 // cal. prefix sum till
                     prev. idx.
    Similarly cal. left & right as
    above.
      sum = sum_right - sum_left
      append this sum.
```

* Question requires you to MOD 10^9+7 each sum.

Q.2 XOR Query:

Here we have given 2 types of queries one to insert integer at end of array, another to XOR all elements of array with given value. After all queries we return the array.

→ Lets remember 2 properties of XOR: $a \text{ XOR } a = 0$, $a \text{ XOR } 0 = a$

→ Here what we do is, we initialize xor_val = 0. Whenever we get new XOR value from query 2, we XOR this with xor_val.

→ While adding ele. to array we XOR them with xor_val and add them.

→ finally after processing all queries, we XOR all the array values with our xor_val, as there are some ele. in array the one inserted before which are not XORED with cum xor_val

Pseudocode:

```
xorQuery(queries):
    ans = []
    xor_val = 0

    for q_type, q_value in queries:
        if q_type == 1:
            ans.append(q_value ^ xor_val)
        else:
            xor_val ^= q_value

    for i in range(len(ans)):
        ans[i] ^= xor_val

    return ans
```

Q-3 Product of array except itself:

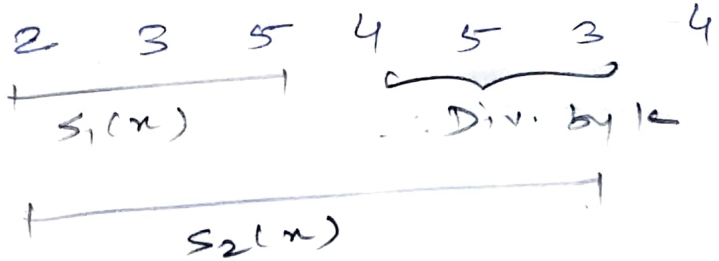
Return an array such that each ele is product of all other elements except itself.

Pseudocode:


```
MOD = 1000000007

productArray(arr, n):
    flag = False
    product = 1

    for ele in arr:
        if ele == 0:  // Zero case
            if flag == False:
                flag = True // found a zero
                continue
            else:  // multiple zeros → prod = 0
                product = 0
                break
        product = (product * ele) % MOD

    if flag = True:  // Zero exist
        if product == 0:  // mult zero.
            return [0] * n
        else:  // one zero
            for i in range(n):
                if arr[i] == 0:
                    arr[i] = product
                else:
                    arr[i] = 0
    else:
        for i in range(n):
            arr[i] = (product // arr[i])
    ret arr
```

Q.4 Count all subarrays whose sum is divisible by k.

↳ k = 7



let S_1 be sum of x be rem

S_2 " " "

$S_1 = k*n + x$

$S_2 = k*m + x$

— $S_2 - S_1 = k(m-n)$

↳ Divisible by k

So using this concept we shall find all subarrays. we shall look for repeated remainders which we store in hashmap, if those rem are found we will add their count to our answer.

Pseudocode:

```
SubArrayCount(arr, k):
    rem_hash = { }
    rem_hash[0] = 1
    sub_arr_ct = 0
    runningsum = 0

    for ele in arr:
        running_sum += ele
        rem = running_sum % k
        value = rem_hash.get(rem, None)
        if value:   // rem already present
            sub_arr_ct += val  // inc. by no of times val. present

            rem_hash[rem] += 1
        else: // 1st time rem found
            rem_hash[rem] = 1

    return sub_arr_ct.
```

Q.5 Paint the fence.

Here you are given sections painted by painters(q) you need to select q-2 painters which can paint the fence optimally.

Brute force:

We will consider each pair of painters using two nested loops & count all sections that can be painted by removing each pair of painters.

Out of all pairs, we will pick one which will result in maximum painted sections.

Pseudocode:


```
PaintFence(ranges, n, q):
    section = [0] * (n+1) // Ranges from 1 based indexing

    for rng in ranges:
        left, right = rng
        // mark all secn painted by this rng.
        for j in range(left, right+1):
            section[j] += 1

// consider every possible pair & remove them:
max_painted = -1e9
painted = None

for i in range(q):
    left, right = ranges[i][0], ranges[i][1]
    painted = section[:] // copy

    // Remove ith painter
    for k in range(left, right+1):
        painted[k] -= 1

    // consider jth painter
    for j in range(i+1, q):
        left_second, right_second = ranges[j][0], ranges[j][1]
```

4. In for loop

```
// Remove sections of ith painter
for k in range(left_record, right_record+1):
    painted[k] -= 1
paint_count = 0
// count sections painted remaing
for paint in painted:
    if paint > 0:
        paint_count += 1
max_painted = max(max_painted, paint_count)
```

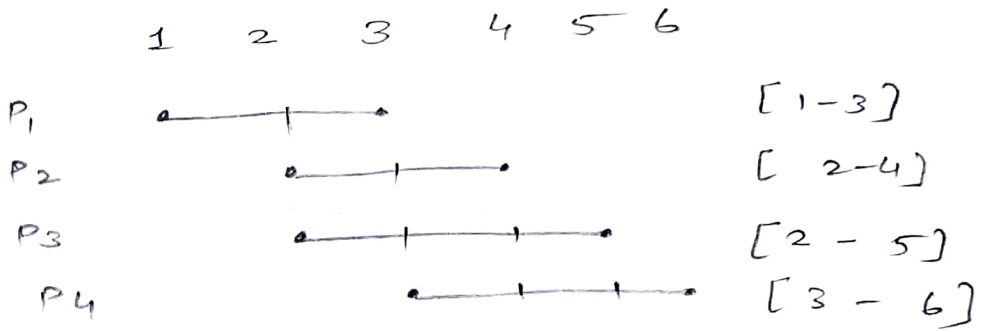
return max_painted.

Optimized Approach:

The approch to solve problem will be some like take pairs of painters try removing them & see which pair keeps maximum painted sections. We shall try and optimize each section of code.

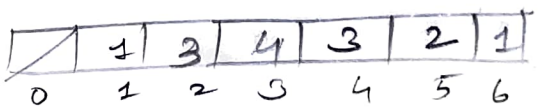
⇒ First we need to create a section array which will contain count of painters that can paint each section.

Lets say we have 6 sections



we can create for each painter we can traverse the section array and add 1 to section being painted

∴ Finally we will get,



But this will req. O(n*k) time complexity can we do in linear scan?

Actually, yes! For each ~~each~~ painter who paints ~~bet~~ section [l, r] we can mark [l] as 1 & [r+1] as -1

Finally take cumulative sum of the array. We add [r+1] as -1 as we want to nullify the effect of addition of +1. Let's see how:

Suppose we to mark idx 2 to 4 with 1.

∴

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

Now if I go on adding the array as arr[i] += arr[i-1] we get

0	0	1	1	1	0
0	1	2	3	4	5

Thus -1 at [5] nullifies the addn effect of placing 1 at [r] & prevents it from moving ahead. We can do this ~~Be~~ something for all painters and get section array.

P1: [1-3]

0	1	2	3	4	5	6	7
0	1	0	0	-1	0	0	0

P2: [2-4]

0	1	2	3	4	5	6	7
0	1	1	0	-1	-1	0	0

P3: [2-5]

0	1	2	3	4	5	6	7
0	1	2	0	-1	-2	-1	

P4: [3-6]

0	1	2	3	4	5	6	7
0	1	2	1	-1	-1	-1	-1

Now if we sum as: arr[i] += arr[i-1] we get

0	1	2	3	4	5	6	7
0	1	3	4	3	2	1	0

Thus we get array as we have seen ~~pri~~ previously. TC = O(n+k).

Here each idx represents how many painters paint that idx. Like idx = 3 is painted by all 4 painters. idx = 5 is painted by 2 painters P3, P4 & so on. - -

Now we shall see in general how we can proceed with problem.

⇒ We shall now create 2 arrays singlePainter & doublePainter.

singlePainter[i] will represent #sections from starting till ith that will be painted by only 1 painter.

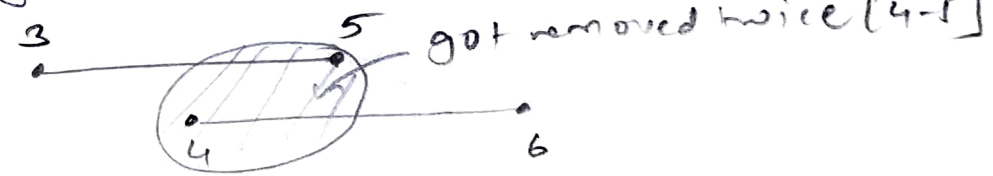
doublePainter[i] ——— painted by only 2 painters.

Why only 2 kinds of arrays? because we are considering to remove only 2 painters from q available painters. Also these 2 arrays will be commulative sum array so that to remove a section painted by painter we only need diff bet right range & (left-1) range of that painter, (Prefix method).

- we will also have total that will contain all sections that can be painted by all painters together.

- Now we shall consider each pair of painters & try removing sections painted by them using prefix method.

- While removing the sections it is possible that there might be intersection bet 2 painters & that section got subtracted twice during removal hence we need to add it again.



So find common indexes.

left_common = max(3,4) = 4

right_common = min(5,6) = 5

MIXED PROBLEMS:

⇒ Add the common section from single pointer array to ans.

⇒ Since we are removing 2 pointers the sections must also be removed from double pointers array, & subsequently from ans.

⇒ Finally calculate total painted sections available & update the maximum.

Pseudocode:

```
// Cal. section array
section = [0] * (n+2)
for i in range(q):
    section[ranges[i][0]] += 1
    section[ranges[i][1]+1] -= 1
// outside for
// prefix sum
for i in range(1, n+1):
    section[i] += section[i-1]

// creating single & double pointers
// array
SP = [0] * (n+1)
DP = [0] * (n+1)
total = 0
for i in range(1, n+1):
    if section[i] == 0:
        total --
    if section[i] == 1:
        SP[i] += 1
    if section[i] == 2:
        DP[i] += 1

    SP[i] += SP[i-1]
    DP[i] += DP[i-1]
```

```
// make painters pair & try removing
//sections painted by them

max_painted = 0

for i in range(q):   # queries
  for j in range(i+1, q):

    curr_painted = total

    // remove ith painter
    left_idx_i = ranges[i][0] - 1
    right_idx_i = ranges[i][1]

    section_i = sp[right_idx_i] -
                sp[left_idx_i]

    curr_painted -= section_i

    // remove similarly for jth
    :
    :
    curr_painted -= section_j

    // find intersecting section betn i&j

    left_common = max(left_idx_i+1,
                      left_idx_j+1)
    right_common = min(right_idx_i, right_idx_j)

    // Remove from double & add a section
    // to single (due to 2 subtraction) if
    // overlapp available.

    if right_common >= left_common:
        curr_painted += sp[right_common] -
                        sp[left_common-1]

        curr_painted -= dp[right_common] -
                        dp[left_common-1]

    max_painted = max(max_painted,
                      curr_painted)

return max_painted.
```

E] MIXED PROBLEMS:

Q.1 Pair Sum:

↳ Find all pairs that sum equal to S.

⇒ This is simple hash map based problem. For each ele, we shall check wheather it's counter part (sum - ele) is present in hash map or not. If yes we shall add it to our ans.

⇒ we shall also sort pairs as required.

Pseudocode:

```
hash_map = defaultdict(int)
ans = []
for ele in arr:
    other = s - ele
    present = hash_map.get(other, None)
    if present:
        if ele < other:
            for i in range(present):
            // add pairs equal to # times the
            // value is present
                ans.append([ele, other])
        else:
            //same for loop
    hash_map[ele] += 1
ans.sort(key=lambda x: x[0])
return ans.
```

Q.2 Valid Pairs:

↳ Given an array you have to form pairs such that sum of these pairs when divided by k gives remainder M.

→ One way is to first check if #ele are odd, for odd ele we cannot form pairs

→ To form pairs we can do using 2 for loops but that will take lot of TC. An optimization we can do is store all remainders in hashmap & for each remainder found, check if it's counterpart is present.

→ The logic is similar to pair sum. If other remainder is -ve just add k to it to make +ve

Pseudo Code:

```
IsValidPair(arr, n, k, m):
    if n % 2 != 0:  // odd
        return False

    remainder_map = defaultdict(int)
    for ele in arr:
        rem = ele % k
        remainder_map[rem] += 1

    for rem, count in remainder_map.items():
        // If rem divides m into 2 halves
        // There must be even occurances
        // of rem to form pair
        if rem * 2 == m and count % 2 != 0:
            return False

        // find counterpart
        other_rem = (m - rem + k) % k

        count_other_rem = remainder_map
                          [other_rem]

        // This count must be equal to rem
        count.
        If count != count_other_rem:
            return False
    return True.
```

Q3 ~~Valid Pairs~~ Max Product

↳ Here we have to find a product with max freq. & count max possible pairs s.t. pq = rs. If 2 candidates have max freq. choose lowest one.

→ Algorithm is simple:
without worring for what the product might be we count freq. of all possible products.

→ Choose product that has maxfreq if there are more than one prod, choose one with min. value.

e.g:- [1, 2, 3, 4, 6, 8, 12, 24]

Here prod 24 will have highest freq as it has 4 pairs contributing to prod 24: (1,24) (2,12) (3,8) (4,6)

Now to form a pair we can choose any 2 $4C_2 = 6$ ∴ 6 pairs possible.

Pseudocode:

```
maxProductCount(arr, n):
    prod_freq_map = defaultdict(int)
    for i in range(n):
        for j in range(i+1, n):
            prod_freq_map[arr[i] * arr[j]] += 1

    max_prod, max_freq = 0, 0

    for prod, freq in prod_freq_map.items():
        if freq == max_freq:
            max_prod = min(max_prod, prod)
        elif freq == max_freq:
            max_prod = min(max_prod, prod)
        else:
            max_prod = prod
            max_freq = freq

    freq_ct = prod_freq_map.get(max_prod, None)
    # If no pair having freq > 1
    if freq_ct == None or <1:
        return [0]
    return max_prod, (freq_ct * (freq_ct - 1)) // 2
```

Q.4 Selling stock:

Logic: we shall find local minima & local maxima throught the stocks as we are allowed to buy & sell multiple times. we shall buy at local minima & sell at local maxima.

Pseudo Code:

```
GetMaxProfit(values, n):
    profit = 0
    i = 0
    while i < n-1:
        // local minima
        while i < n-1 & values[i+1] <= values[i]:
            i++
        if i == n-1:  // End reached
            Break
        buy = i
        i++
        // local maxima
        while i < n-1 & values[i+1] >= values[i]:
            i++
        sell = i
        profit += (values[sell] - values[buy])
```

Q.5 Non Decreasing Array:

→ we have to check if given array could be converted into non-decreasing array (ascending) by doing at most one modification.

→ First we need to find where an array decreases also known as inversion.

If we look at growth pattern, inversion will look as follows:

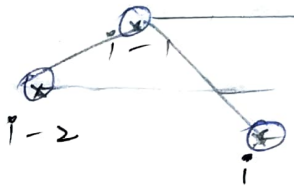
If we consider only 3 ele, the pattern would be like follows:

1)

i < 2 or arr[i-1] <= arr[i]
⇒ Bring i-1 to level of i

2) arr[i] < arr[i-1]

↳ make i at level of i-1



Pseudocode:

```
modified = 0
for i in range(1,n):
    if arr[i] <= arr[i-1]: // Inversion
        modified += 1
        if modified > 1:
            return false

        if i < 2 or arr[i-2] <= arr[i]
            arr[i-1] = arr[i]
        else
            arr[i] = arr[i-1]

return True
```

Q.6 Longest Consecutive Sequence (37)

Here we have to find len. of cons. sequence which is longest.

→ Since, here nos can be repeating so we fst make a set out of them.

→ for each ele. found, we search if previous ele. present, if so it signifies that this is not the starting of consecutive sequence.

→ If no prev. ele is found this can be start of the a seq. so we increment i+ and check till how long it's seq. inc. & keep count & update the maximum.

Pseudocode:

```
unique_nos = set(arr)
longest_seq = 0
for ele in unique_nos:
    if ele-1 not in unique_nos:
        curr = ele
        while curr in unique_nos:
            curr += 1
        longest_seq = max(longest_seq,
                          curr-ele)
return longest_seq
```

Q.7 Second largest Ele. in Array:

↳ Here we need to find second largest ele in array.

Algo: we keep 2 pointers max & second_max.

→ If max ele is found, we take previous value of max to second_max

→ Also there can be ele which is > second_max but < max hence that case must be noted too.

Pseudocode:

```
unique_nos = set(numbers)
if len(unique_nos) < 2:
    return -1
max_no, second_max = -1eq, -1eq
for ele in unique_nos:
    if ele > max_no:
        second_max = max_no
        max_no = ele
    elif ele > second_max and
            ele != max_no:
        second_max = ele
if second_max == -1eq:
    return -1
return second_max.
```

Q.8 Array after k operations:

↳ Here we are given an arr of ~~aa~~ a ~~array~~ numbers k. where you have to subtract array from it's max. ele k times & return the array.

e.g: [20, 15, 10, 5] k = 4

⇒ k=1, max=20 [0, 5, 10, 15]

k=2, max=15 [15, 10, 5, 0]

k=3, max=15 [0 5 10 15]

k=4, max=15 [15 10 5 0]

→ Thus, we see array oscillates only betn 2 values depending on k being odd/even.

→ If k is odd, we sub each ele from it's max & return the array

If k is even, we sub minimum from each ele & return the array

Pseudocode:

```
max_ele = max(arr)
min_ele = min(arr)

If k != 0:
    If k%2 != 0:  //odd
        return [max_ele - ele for ele in arr]
    else
        return [ele - min_ele for ele in arr]

return arr.
```

Q.9 minimum #Platforms:

↳ Here we are given arrival & departure times of trains, we are required to calculate min. no. of platforms required.

→ Let's first understand why an additional platform is required?

→ When we already have a train arrived on platform & not departed

yet, within that time interval if another train arrives, we shall need a platform.

→ How shall we know that a train has ~~already~~ arived when another train is already present? we compare arrival time of $i+1^{th}$ train with departure time of i^{th} train. If A.T. < D.T[i], we need a platform.

→ Also whenever a train leaves we shall reduce the count of curr_platform to denote availability of platform.

→ we also need to sort AT, DT so that we can compare the times, the procedure will be kind of similar to merge procedure of merge sort.

Pseudocode:

```
at.sort()  // SOA arrival time
dt.sort()  //  "  departure time

max_platforms = 1   // Atleast 1 platform
curr_platforms = 1  //  is required

i, j = 1, 0  // Begin by checking AT of
             //  train 1 & DT of train 0

while i<n and j<n:
    if at[i] <= dt[j]:
        curr_platforms ++
        i++
    else:
        curr_platforms --
        j++

    max_platforms = max(max_platforms,
                        curr_platforms)

return max_platforms.
```

Q.10 Majority Ele - 2

↳ In this problem, we need to find elements that occur more than $\lfloor N/3 \rfloor$ times in given array.

⇒ Before solving this problem we need to understand, how to find a majority ele (element that occurs more than $\lfloor N/2 \rfloor$ times) in an unsorted array.

⇒ one way would be to sort the list. If there is a majority ele then it must be middle value. we run another pass to count it's frequency. $O(n \log n)$

⇒ There is another algo suggested by 'Boyer-Moore' that finds majority ele in $O(n)$ time & $O(1)$ space. It is also called as 'Boyer-Moore Majority Voting' Algorithm.

⇒ In first pass of this algo, we generate single candidate value, which is the majority value if it exists. Second pass simply counts the frequency of that value to confirm.

In first pass we need 2 values:

1) A candidate value, initially set to any value.
2) Count initially set to 0.

- For each ele in our input list, we first examine the count value. If count is = 0, we set candidate to the value of curr ele. If we find ele which is equal to candidate we inc. the count else we decrement the count.

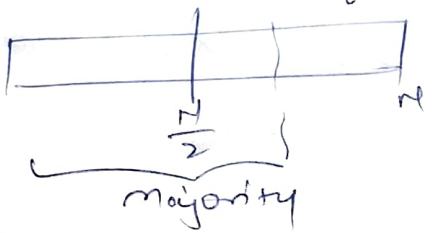
At end of all inputs, the candidate will hold the majority ele if it exist.

A second $O(N)$ pass can verify that the candidate is majority ele.

Algo:

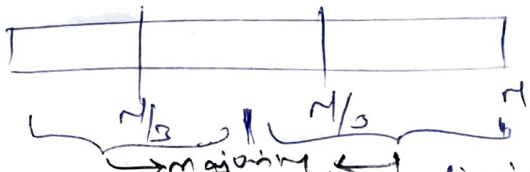
```
candidate = 0
count = 0
for value in input:
    if count == 0:
        candidate = value
        count += 1
    elif candidate == value:
        count += 1
    else:
        count -= 1

freq = 0
for value in input:
    if value == candidate:
        freq += 1

If freq > ⌊N/2⌋:
    return candidate
else:
    return -1
```

- In earlier problem we have to find ele whose freq. is $> N/2$



∴ If was sure there would be at most 1 majority ele.

Now we need to find ele that have freq $> N/3$



So array can be divided into 3 parts & if we ought to have ele $> N/3$ we can have atmost 2 majority elements.

So what we shall do, use the Boyer-moore's Voting Algo but calculate 2 majority instead of 1. ∴ we shall have 2 candidates. The working will be similar.

Pseudocode:

```
ct1, ct2, candidate1, candidate2 = 0,0,0,0

for ele in arr:
    if ele == candidate1:
        ct1 += 1
    elif ele == candidate2:
        ct2 += 1
    elif ct1 == 0:
        candidate1 = ele
        ct1 = 1
    elif ct2 == 0:
        candidate2 = ele
        ct2 = 1
    else:
        ct1 --
        ct2 --

l = len(arr) //3
return [n for n in (candidate1, candidate2)
            if arr.count(n) > l]
```

(code gives TLE in python for 1 test case
try same in cpp)