TODOs:

1. Try to solve the “Product Of Array Except Self” problem using prefProd[] & not suffProd[]
2. Triplet sum problem: code both the brute-force as well as using the pair-sum technique in O(n^2) time.
3. Identify the edge-cases and do a dry-run the “Remove duplicate” problem. Very important (from both the learning and interview perspective) to be able to recognize the edge-cases and being able to do a dry-run on those.

3	30	38
36	43	60
40	51	69
41	**62**	70

X = 62

N x M

Binary search on each row: O(N * Log(M))

Algo:

1. Start from the top-right
2. If arr[i][j] > X: discard the column
3. If arr[i][j] < X: discard the row

TC: O(N + M)
Aux Space: O(1)

Prefix/Suffix Sum

Arr: [1, 3, 7, 4, 10, 12, 11]
prefSum: [1, 4, 11, 15, 25, 37, 48]
suffSum: [48, 47, 44, 37, 33, 23, 11]

Product Of Array Except Self

i/p : [1, 2, 3, 4]
o/p: [24, 12, 8, 6]

PrefProd: prefProd[i] = arr[0] * arr[1] * arr[2] … arr[i]
SuffProd: suffProd[i] = arr[i] * arr[i + 1] * arr[i + 2] … * arr[n - 1]

Result[0] = suffProd[1]
Result[n - 1] = prefProd[n - 2]
Result[i] = prefProd[i - 1] * suffProd[i + 1] :-> i > 0 and i < n - 1

Maximum Occurred Integer In N Ranges

L[]: [1, 4, 3, 1, 19]
R[]: [15, 8, 5, 4, 20]

(1, 15): 1, 2, 3, **4**, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
(4, 8): **4**, 5, 6, 7, 8
(3, 5): 3, **4**, 5
(1, 4): 1, 2, 3, **4**
(19, 20): 19, 20

O/P: 4

Approach-1:

Frequency Map to track the freq of all the elements
TC: O(n * maxx) where maxx is Max of R[i] - L[i]

Worst-case: O(10^6 * 10^6) = 10^12 ~ 1000 seconds

Approach-2:

Use Prefix Sum!!

L[]: [1, 4, 3, 1, 19]
R[]: [15, 8, 5, 4, 20]

Frequency-map:

Idx	freq
0	
1	2
2	
3	1
4	1
5	-1
6	-1
7	
8	
9	-1
10	
11	
12	
13	
14	
15	
16	-1
17	
18	
19	1
20	
21	-1

Idx	freq
1	2
2	2
3	3
4	4
5	3
6	2
7	2
8	2
9	1
10	1
11	1
12	1
13	1
14	1
15	1
16	0
17	0
18	0
19	1
20	1
21	0

(4, 8) (3, 7)

1	0	0
2	0	0
3	+1	+1
4	+1	+2
5	0	+2
6	0	+2
7	0	+2
8	-1	+1
9	-1	0

For (L[i], R[i])
L[i] -> +1
R[i] + 1 -> -1

To calculate the freq array: O(n)
To calculate the pref sum: O(maxx) where maxx is the max of R[]

Total TC: O(n + maxx)

Pair with a given sum in a sorted array

Arr: [1, 2, 5, 6, 10], target = 8
o/p: true

Arr: [1, 2, 5, 6, 10], target = 20
o/p: false

Arr: [1, 2, 5, 6, 10, 10], target = 20
o/p: true

Approach-1:

At every i-th element, do a binary search for (target - arr[i]) in arr[i + 1 … n-1]

TC: O(n * log(n))

Approach-2:

Two-pointer based

[1, 2, 5, 6, 10] -> sum = 1 + 10 = 11 > target(8) -> discard 10

[1, 2, 5, 6] -> sum = 1 + 6 = 7 < target(8) -> discard 1

[2, 5, 6] -> sum = 2 + 6 = 8 == target -> return true

Sum > target -> discard the highest element -> right – –
Sum < target -> discard the lowest element -> left ++

TC: O(n)
AS: O(1)

Triplet Sum In Array

A[] = {1, 4, 45, 6, 10, 8} X = 13
Output: True

Approach-1:

Brute-force -> 3-nested for-loops to find the sum of **ALL** the triplets
i -> 0 … n-1
j -> i + 1 …
k -> j + 1 …

TC: O(n^3)

Approach-2: (TODO)

1. Sort the array.
   [1, 4, 45, 6, 10, 8] -> [1, 4, 6, 8, 10, 45]
2. Fix the first element and find the pair sum of target - arr[i] in the remaining array.
   Left = i + 1
   Right = n - 1

   TC: O(nlog(n) + n^2) = O(n^2)
   AS: O(1)

Remove Duplicates From Sorted Array

i/p: [1, 2, 2, 3, 3]
o/p: 3

Return the count of the elements in the resultant array.

[1, 2, 3, 3, 3]
[1, 2, 3, 100, 1000]
[1, 2, 3, -100, 999]
…

Approach-1:

Find all the distinct elements using an ordered-set and build the output.

TC: O(n*log(n))
AS: O(n)

Approach-2:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9
[1, 2, 2, 2, 2, 3, 3, 4, 4, 4]
[1, 2, 3, 4, 2, 3, 3, 4, 4, 4]

first = 3, second = 7

“i” is the current pointer accepting the new elements,
“j” is at the beginning of the subarray and skips to the first element which is not equal to the initial element of the subarray.

In the first iteration, subarray under consideration should be:
[1, 2, 2, 2, 2, 3, 3, 4, 4, 4]

In the second iteration:
[2, 2, 2, 2, 3, 3, 4, 4, 4]

In the third iteration:
[3, 3, 4, 4, 4]

In the fourth iteration:
[4, 4, 4]

In the fifth iteration:
[]