After solving , reasonable number of problems I have realized that , whenever you need something like complement for a portion of a sum , you can immediately go for the method of hashing.

# Minimum operations to make all elements equal

Given an array consisting of positive integers, return the minimum number of operations to make all the elements of the array equal. The operations can be addition, multiplication, division, or subtraction.

Brute-Force

To find the element with the highest frequency, we will run two loops. The outer loop picks all the elements one by one and the inner loop finds the frequency of the picked element and compares it with the element with the highest frequency so far. If it is greater, it replaces to be the highest frequent element. After this, we will simply return the "number of elements  - frequency of the most frequent element".

Time complexity: O(n2)

Space complexity: O(1)

Better Approach (Hashing)

Maintain a map that consists of the frequency of all the elements in the array. The highest frequent element in the hash table will be our target element and thus, our answer will be "number of elements - frequency of the target element".

Time complexity: O(n)

Space complexity: O(n)

# Check if the given array contains duplicate elements within k distance

Given an unsorted array that may contain duplicates. Also given a number k which is smaller than the size of the array, returns true if the array contains duplicates within k distance.

Example-1:

Input: k = 3, arr[] = {1, 2, 3, 4, 1, 2, 3, 4}

Output: false

All duplicates are more than k(3) distance away.

1...1(has a distance of 4)

2...2(has a distance of 4)

3...3 and 4...4 are similar

Brute-Force

A Naive solution is to run two loops. The outer loop picks every element ‘arr[i]’ as a starting element, and the inner loop compares all elements which are within k distance of ‘arr[i]’.

Time complexity: O(k\*n)

Space complexity: O(1)

Better Approach (Hashing)

If the array element is already present in our map then get its position from a map and calculate the difference between the current index and the position that we have fetched from the map.

If the difference is greater than k then update the position of the current element in the map

If the difference is less than k then simply return true.

# Max distance between same elements

Given an array with repeated elements, the task is to find the maximum distance between two occurrences of an element.

Input : arr[] = {3, 2, 1, 2, 1, 4, 5, 8, 6, 7, 4, 2}

Output: 10

// maximum distance for 2 is 11-1 = 10

// maximum distance for 1 is 4-2 = 2

// maximum distance for 4 is 10-5 = 5

Brute-Force

A simple solution for this problem is to, one by one, pick each element from the array and find its first and last occurrence in the array and take the difference between the first and last occurrence for maximum distance.

Time complexity: O(n^2)

Space complexity: O(1)

Better Approach (Hashing)

The idea is to traverse the input array and store the index of the first occurrence in a hash map. For every other occurrence, find the difference between the index and the first index stored in the hash map. If the difference is more than the result so far, then update the result.

Time complexity : O(n)

Space complexity: O(n)

# Count pairs with given sum

Given an array of N integers, and an integer K, find the number of pairs of elements in the array whose sum is equal to K.

Input:

N = 4, K = 6

arr[] = {1, 5, 7, 1}

Output: 2

Explanation:

arr[0] + arr[1] = 1 + 5 = 6

and arr[1] + arr[3] = 5 + 1 = 6.

Input:

N = 4, K = 2

arr[] = {1, 1, 1, 1}

Output: 6

Explanation:

Each 1 will produce sum 2 with any 1.

Brute-Force

A simple solution is to traverse each element and check if there’s another number in the array which can be added to it to give sum.

This can be achieved by nested loops.

Time Complexity: O(n2), traversing the array for each element

Auxiliary Space: O(1)

Better Approach (Hashing)

i)HashMap on the go

Check the frequency of sum – arr[i] in the arr , by using a hash map.

If the complement exists , we can form that many pairs with the current element to complement , so we increment the count.

After that , increase the frequency of arr[i] which can be used by the future elements.

Time complexity : O(n)

Space complexity: O(n)

ii) HashMap post construct

Here we will first build the hashmap with the all frequencies of the values , in the array.

Then we shall find the frequencies of element and it’s complement to sum and mutiply them , and add it to the count variable.

If the element = complement , we might get the repeated pairs , like take

[1, 1, 1, 1] and target = 2 , in such a case we should do (nc2) i.e n(n-1)/2.

Time complexity : O(n) + O(n) as we are traversing the array twice.

Space complexity: O(n)

# First Unique Character in a String

Given a string s, find the first non-repeating character in it and return its index. If it does not exist, return -1.

Input: s = "loveleetcode"

Output: 2

Brute-Force

Idea is to loop over the string and for every character check the occurrence of the same character in the string. If the count of its occurrence is 1 then return that character. Otherwise, search for the remaining characters.

Time Complexity: O(N2), Traverse over the string for every character in the string of size N.

Auxiliary Space: O(1)

Better Approach (Hashing)

Idea is to find the frequency of all characters in the string and check which character has a unit frequency.

Time Complexity: O(N) + O(N) ~ O(N)

Auxiliary Space: O(26) , At most the hashmap keys will be covered with this much space only.

# Find Common Characters

Given a string array words, return an array of all characters that show up in all strings within the words (including duplicates). You may return the answer in any order.

Example :

Input: words = ["bella","label","roller"]

Output: ["e","l","l"]

Better Approach (Hashing)

This solution uses an array count of size 26 to keep track of the minimum count of each character that appears in all the words. It initializes the count array with a large value and updates it by finding the minimum count for each character in each word.

Time Complexity: O(26 x N )

Space Complexity : O(len of min word in worst case)

# Longest Consecutive Sequence in an Array

You are given an array of ‘N’ integers. You need to find the length of the longest sequence which contains the consecutive elements.

Example 1:

Input: nums = [100,4,200,1,3,2]

Output: 4

Explanation: The longest consecutive elements sequence is [1, 2, 3, 4]. Therefore, its length is 4.

BruteForce:

If they are asking to find this in the un-sorted array only , we can sort the array then

There are 3 cases that may need to be handled,

1. prev and curr elements are consecutive
2. prev and curr are non-consecutive but equal.
3. prev and curr are non-consecutive and different

We can keep a counter and increase it as long as the case i is valid , and case ii we can skip , as we are choosing to consider only strictly increasing sequence length , finally case iii is where we should break the sequence.

**Time Complexity:** We are first sorting the array which will take O(N \* log(N)) time and then we are running a for loop which will take O(N) time. Hence, the overall time complexity will be O(N \* log(N)).

**Space Complexity:** The space complexity for the above approach is O(1) because we are not using any auxiliary space.

Best Approach:

If sorting were to be used in the above approach, we can follow this approach.

We will first push all are elements in the HashSet. Then we will run a for loop and check for any number(x) if it is the starting number of the consecutive sequence by checking if the HashSet contains (x-1) or not. If ‘x’ is the starting number of the consecutive sequence we will keep searching for the numbers y = x+1, x+2, x+3, ….. And stop at the first ‘y’ which is not present in the HashSet. Using this we can calculate the length of the longest consecutive subsequence.

**Time Complexity:** The time complexity of the above approach is O(N) because we traverse each consecutive subsequence only once. (assuming hashset takes O(1) to search)

**Space Complexity:** The space complexity of the above approach is O(N) because we are maintaining a HashSet.

# Count Number of Pairs With Absolute Difference K

Given an integer array nums and an integer k, return the number of pairs (i, j) where i < j such that |nums[i] - nums[j]| == k.

The value of |x| is defined as:

x if x >= 0.

-x if x < 0.

Brute-Force

Explore each pair using a nested for loop.

Time Complexity: O(n2), traversing the array for each element

Auxiliary Space: O(1)

Better Approach (Hashing)

Iterate through each number num in the input array nums.

If num - k or num + k exists in the frequency\_map, increment the count by the frequency of num - k or num + k in the frequency\_map.

Increment the frequency of num in the frequency\_map.

Time complexity : O(n)

Space complexity: O(n)

Question : Why do we have to make the hash map on the go and not before hand ?

Answer : If we make that hash map before hand , we will be taking the same pair twice , like (i , j) and (j , i) ,so we need to make the hashmap on the go only which ensures only one pair is counted.

# Subarray Sum Equals K

Given an unsorted array of integers, find the number of subarrays having a sum exactly equal to a given number k.

Brute-Force

Explore each pair subarray using a nested for loop.

Time Complexity: O(n2), traversing the array for each element

Auxiliary Space: O(1)

Better Approach (Hashing)

While traversing the array, storing sum so far in currsum.

Also, maintain the count of different values of currsum in a map.

If the value of currsum is equal to the desired sum at any instance increment count of subarrays by one.

The value of currsum exceeds (in case of positive nums , but we should generalize this condition ) the desired sum by currsum – k. If this value is removed from currsum then the desired sum can be obtained.

From the map, find the number of subarrays previously found having sum equal to currsum-k.

Excluding all those subarrays from the current subarray, gives new subarrays having the desired sum.

Note that when currsum is equal to the desired sum then also check the number of subarrays previously having a sum equal to 0.

Time complexity : O(n)

Space complexity: O(n)

# Count the number of subarrays having a given XOR

# Find the Prefix Common Array of Two Arrays

You are given two 0-indexed integer permutations A and B of length n.

A prefix common array of A and B is an array C such that C[i] is equal to the count of numbers that are present at or before the index i in both A and B.

Return the prefix common array of A and B.

A sequence of n integers is called a permutation if it contains all integers from 1 to n exactly once.

Example 1:

Input: A = [1,3,2,4], B = [3,1,2,4]

Output: [0,2,3,4]

Explanation: At i = 0: no number is common, so C[0] = 0.

At i = 1: 1 and 3 are common in A and B, so C[1] = 2.

At i = 2: 1, 2, and 3 are common in A and B, so C[2] = 3.

At i = 3: 1, 2, 3, and 4 are common in A and B, so C[3] = 4.

Brute Force Approach :

Run an outer for loop for keeping track of positions , as it moves ahead and then run a nested for loop at each postion to find the no.of common elements till that position.

TC : O(n3)

SC : O(1)

Better Approach (Hashing)

We will iterate the both arrays simultaneosly and check if elements from both the arrays at a particular index are equal if so we will increase the count , We shall maintain a hashset and keep pushing the seen elements from both arrays sofar , in case there is a mismatch we see if the elemets already exists in the hashset , meaning there is a match before so we will increase the count for that case.

Time complexity : O(n)

Space complexity: O(n)

# Longest Sub-Array with Sum K

Given an array containing **N** integers and an integer **K**., Your task is to find the length of the longest Sub-Array with the sum of the elements equal to the given value **K**.

**Brute Force:** Run a nested for loop

Time Complexity: O(N^2) as we have two loops for traversal

Space Complexity: O (1) as we aren’t using any extra space.

very similar question : Length of the longest subarray with zero Sum in striver sheet.

we do not update the index of a sum if it’s seen again because we require the length of the longest subarray.

# Longest Palindrome by Concatenating Two Letter Words

You are given an array of strings words. Each element of words consists of two lowercase English letters.

Create the longest possible palindrome by selecting some elements from words and concatenating them in any order. Each element can be selected at most once.

Return the length of the longest palindrome that you can create. If it is impossible to create any palindrome, return 0.

A palindrome is a string that reads the same forward and backward.

Input: words = ["ab","ty","yt","lc","cl","ab"]

Output: 8

Explanation: One longest palindrome is "ty" + "lc" + "cl" + "yt" = "tylcclyt", of length 8.

Note that "lcyttycl" is another longest palindrome that can be created.

Brute Force

Run a nested loop and find every possible combination by concatinating and check if it is palindrome or not.

Time complexity : O(n^3)

Space complexity: O(1)

Better Approach

The idea is form a palindrome of format L M R ,

L -> left, M -> middle , R-> right

Reversing L will give R , and M only consists of 1 character.

Construct a hash-map before hand with frequencies (as there may be multiple occurences of the words like “ab” , “ab” etc )

For each word check ,

If it is diffent characters “ab” , then find the occurences of it’s reverse also “ba” and take the min(freq , freq\_rev) , this is because we have to ensure that we are placing same amount of words and it’s reverse in L and R.

And remove both of them , otherwise they will again appear in the counts.

If it is same characters “gg” , we need to consider only 1 odd frequency of that kind and rest we should be considering even .

Let’s say we have

‘aa’ -> 7

‘bb’ -> 3

‘cc’ -> 5

Here , we should have any one of the odd occurences and remaining we need to take even only , like ‘aa’ 🡪 7 , ‘bb’ 🡪 2 , ‘cc’ 🡪 2 (does not matter which one we are taking odd , as all will add up into the length only).

Other wise we can not partition them to L and R properly.

Time complexity : O(n)

Space complexity: O(n)

# Max Sum of a Pair With Equal Sum of Digits

Given an array A consisting of N integers, return the maximum sum of two numbers whose digits add up to an equal sum.

if there are not two numbers whose digits have an equal sum, the function should return -1.

Constraints: N is integer within the range [1, 200000]

each element of array A is an integer within the range [1, 1000000000]

Example :

Input:

A = [51, 71, 17, 42]

Output: 93

Explanation: There are two pairs of numbers whose digits add up to an equal sum: (51, 42) and (17, 71), The first pair sums up to 93

Brute Force

Run a nested loop for every pair of numbers .

Time complexity : O(n x n x 9 ) -🡪 O(9 ) for calculating the digit sum as the max num given limit is 10^9. 🡪 O(n^2)

Space complexity: O(1)

Best Approach

Why hash table ? : We can avoid the inner loop , if we have some how stored the digit sum of previous visited nums.

Iterate through each number in the given array.

For each number, calculate the sum of its digits.

Check if the sum of digits exists as a key in the digitSums dictionary.

If the sum of digits does exist in the dictionary, update the maxSum variable with the maximum value between the current maxSum and the sum of the current number and the corresponding value in the digitSums hashtable.

We should update the key digitSum , only if the current num if greater than previous number with same digitSum.(as we are looking to maximize the sum)

If the sum of digits doesn't exist in the dictionary, add it as a new key with the value equal to the current number.

Time Complexity : O(N x 9 ) 🡪 O(N) as 9 is constant.

Space Complexity :- how many keys can there be at maximum in hashmap —> only digitsum can be there as they key so the answer is O(maximum digitsum) == O(81)[practical manner] which is constant ; O(1) space complexity = constant space complexity.

# Find number of ‘t’ in ‘s’

<https://leetcode.com/discuss/interview-question/3114099/AMAZON-OA-(INTERN-2024)>

Given two strings ‘t’ and ‘s’ , find the number of occurences of ‘t’ in ‘s’.

We would only match the characters of ‘t’ in ‘s’ , their order does matter.

Once the complete ‘t’ is found in ‘s’ , we would need to remove those corresponding charaters in ‘s’ for further consideration.

Example :

s = mononom

t = mon

We can see that 2 times we can see ‘t’ in ‘s’

1)~~mon~~onom mon

2)~~mon~~o~~nom~~ mon

S= ‘bcaacb’

T= ‘abc’

We can see that ‘abc’ is found 2 times i)’bca’ ii)’acb’

Brute Force

Start with an infinite loop to check cnt of string t in string s until we are getting string t in string s; and start removing the characters from s once matched to t.

if any char of t doesn’t present in s at any moment then we will break & return the cnt;

if string s is empty at any moment then again we will break from the loop & return the cnt.

Time Complexity : O(len(t)) x O(len(s)) x O( len(s) / len(t) )

Space Complexity : O(1)

Best Approach :

Take 2 hashmaps. One will store the cnt of char present in ‘s’ & another will store the cnt of char present in ‘t’.

Now traverse ‘t’ ( or the 2nd HashMap ) and ans = min(ans, mp1[t[i]] / mp2[t2[i]] )

In essence , we are checking for each character in ‘t’ , how many times it has appeared in ‘s’ , and we need to take the minimum of such value as the whole of ‘t’ needs to be present in ‘s’.

Edge cases :

If ‘s’ is empty , we would have ans = 0

If ‘t’ is empty , we would have ans = inf , ( need to return -1 in such a cases)

Time Complexity : O(len(t) + len(s))

Space Complexity : O(len(t) + len(s))

# Convert an Array Into a 2D Array With Conditions

You are given an integer array nums. You need to create a 2D array from nums satisfying the following conditions:

The 2D array should contain only the elements of the array nums.

Each row in the 2D array contains distinct integers.

The number of rows in the 2D array should be minimal.

Return the resulting array. If there are multiple answers, return any of them.

Note that the 2D array can have a different number of elements on each row.

Example 1:

Input: nums = [1,3,4,1,2,3,1]

Output: [[1,3,4,2],[1,3],[1]]

Explanation: We can create a 2D array that contains the following rows:

- 1,3,4,2

- 1,3

- 1

All elements of nums were used, and each row of the 2D array contains distinct integers, so it is a valid answer.

It can be shown that we cannot have less than 3 rows in a valid array.

Hashing Approach 1:

Find the freq of every element in the array and start filling the rows , until the frequency dried out for each element.

Time Complexity : O(Size of the map x highest freq )

Space Complexity : O(N) , at worst case all the elements in the given array are unique.

Hashing Approach 2:

Make a frequency map to store number of occurance of all elements.

The most appearing elements count will decide the number of row in final answer.

For Ex : If we have a element appearing twice then will run a loop to push it in 2 rows, similarly done for all.

# First Completely Painted Row or Column

You are given a 0-indexed integer array arr, and an m x n integer matrix mat. arr and mat both contain all the integers in the range [1, m \* n].

Go through each index i in arr starting from index 0 and paint the cell in mat containing the integer arr[i].

Return the smallest index i at which either a row or a column will be completely painted in mat.



Input: arr = [1,3,4,2], mat = [[1,4],[2,3]]

Output: 2

Explanation: The moves are shown in order, and both the first row and second column of the matrix become fully painted at arr[2].

Brute Force

For every move we need to check if any of the row or col is completely filled or not.

Time Complexity : O(m x n )^2

Space Complexity : O(1)

Best Approach

We can create a map for each value , index of the array.

Since we are traversing from index 0 in the **arr** to paint the **matt**.We need to consider in the same order if we can fill any row / col completely.

For each matrix row / col to be painted, we need to find the maximum index in that particular row / column.

Max index in that row / col ensures that all the remaining indices in that row / col are visited (painted ) already.

For example if we have row like [ 5 , 2 , 4 ] **as indices** for some values which were to be painted , index 5 will ensure that all remaining 2 , 4 are covered already while painting in the order.

Out of all such indices for every row / col , we need to find the minimum index which make sure that our answer is min possible.

Time Complexity : O(m x n )

Space Complexity : O(m x n ) , those many number of keys present.

# Range Sum Query 2D – Immutable

Given a 2D matrix matrix, handle multiple queries of the following type:

Calculate the sum of the elements of matrix inside the rectangle defined by its upper left corner (row1, col1) and lower right corner (row2, col2).

Implement the NumMatrix class:

NumMatrix(int[][] matrix) Initializes the object with the integer matrix matrix.

int sumRegion(int row1, int col1, int row2, int col2) Returns the sum of the elements of matrix inside the rectangle defined by its upper left corner (row1, col1) and lower right corner (row2, col2).You must design an algorithm where sumRegion works on O(1) time complexity.

Brute Force

Iterate through all sub matrix.

TC : O(R x C)

Optimal Approach

Build a preprocessing prefix sum of all the rows , so that now we just have to iterate through the all the last column of the rectangle to the sum of that particular row.

Preprocessing Row



If we take that green colour box , we could see that in the preprocessing prefix sum matrix , in the last row 11 – 3(which is outside of the box) gives 8 , likewise if we could sum all such numbers (8 + 9 + 8 = 25 ) gives you the answer 25.

T.C : O(R )

Best Approach

Building Area Matrix

We have to do the preprocessing such that , we can fill all the cells of matrix with rectangular sum area from (0 , 0 ) to that cell.

Just take R+1 , C+1 size matrix to make the formulation work always , otherwise we need to have some if else conditions to build it.

Area({0 , 0} {r ,c }) = Area({0 , 0} {r-1 ,c }) (top)+ Area({0 , 0} {r ,c-1 }) (left)

- Area({0 , 0} {r-1 ,c-1 }) ( common overlapping portion ) +

curr\_element {matrix[r-1][c-1]} 🡪 {As when we map to matrix we need to do -1}

Top + Left – Top\_Left\_Diagonal + current element.

here , **[r , c] should start from [1,1] and end with [R,C]** as the first row and column already filled with 0.

|  |  |  |
| --- | --- | --- |
| 8 | 5 | 4 |
| 2 | 1 | 3 |
| 6 | 7 | 9 |

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 0 | 0 | 0 |
| 0 | 8 | 13 | 17 |
| 0 | 10 | 16 | 23 |
| 0 | 16 | 29 | 45 |

Calculating Sum

Now after building that area matrix , we can have

Sum ({r1 , c1} {r2 ,c2 }) = Area({r2+1 , c2+1 }) - Area({r1 , c2+1 }) (top)

- Area({r2+1 , c1 }) (left) + Area({r1 , c1 }) (diagonal )

Here what we are doing is from the bottom corner {r2+1 , c2+1 } , subtractiong top {r1 , c2+1 } 🡪 [ *r1+1 is the top corner and we just need to exclude that cell also* ] and bottom {r2+1 , c1 } 🡪 [ *c1+1 is the bottom corner* *and we just need to exclude that cell also*  ] and adding diagonal once as it was subtracted twice.



As from the figure it is evident that we should have the below formula

sum ([r1 , r2] , [ c1 , c2] ) {green portion} =

sum ( [0,0] [r2,c2] ) {sum till the bottom corner }

– sum([0,0] , [r1-1 , c2]) { Top ( red portion ) }

- sum([0,0] , [r2 , c1-1] ) { Left ( orange portion ) }

+sum ([0,0] , [r1-1 , c1-1] ) {Diagonal (overlapping portion) }

When expressing in terms of area , we should add the +1 to each coordinate.

# Range Update

You are given an array of size “N”; all elements are 0 in it :)

You are given Q queries :-> [L,R] ; you have to add +1 to all the elements from [L,R].

After all queries are performed you have to print the final answer.

BruteForce Approach :

For every query iterate through the array and from L to R range and do the update.

Time Complexity : O(N x Q ) , since we have Q queries and N as the length of the range on worst case.

Space Complexity : O(1)

Better Approach :

Array : [0 , 0, 0 , 0 , 0 , 0]

Index :[ 0 , 1 , 2 , 3 , 4, 5 ]

Q1 : [1 , 4]

Brute : [0 , 1, 1 , 1 , 1 , 0]

Better: [0 , 1, 0 , 0 , 0 , -1]

Q2 : [2 , 5]

Brute : [0 , 1, 2 , 2 , 2 , 1]

Better: [0 , 1, 1 , 0 , 0 , -1]

Q3 : [0 , 3]

Brute : [1 , 2, 3 , 3 , 2 , 1]

Better: [1 , 1, 1 , 0 , -1 , -1]

The trick here in the better approach of the range update is that , we are updating the

values of b[L] = +1 and b[R+1] = -1 for each of the query , which means that at the end

when we take the prefix sum of the resultant array , we should be getting the updated.

range at one shot , here for the question why we should update b[R+1] = -1 is that during the

prefix sum till R , it will get added up and after that , it should get reduced.

Time Complexity : O(Q) for running down all queries and updating the sum at L and R+1 + O(N) for taking the prefix sum at the end. O(N + Q)

Space Complexity : O(N) / O(N+1) for maintaing the array of size N.

# Maximize equal numbers

<https://www.geeksforgeeks.org/google-interview-experience-for-girl-hackathon-2023/>

You are given the following:

• An array a consisting of n elements

• Integer k

For each (1≤i≤n), perform the following operation exactly one time:

Replace ai by ai +x, where x Є [-k, k] which denotes x should lie in the range of -k and k, both inclusive.

Determine the maximum length of the subsequence of array a, such that all numbers in that subsequence are equal after applying the given operation.

Notes:

A subsequence of array ‘a’ is an array that can be derived from array a by removing some elements from it. (maybe none or all of them)

Assuming 1 based indexing.

**Constraints: (IMP)**

\* 1<=N<=10^5

\* 1<=k<=10^5

\* k+1<=a[i]<=10^5 // Meaning that if you subtract k from any a[i] , it should be still greater than or equal to 1.

Example :

[5 , 8 , 10] k =2

You may generate the below sets possibly.

[7 7 10] [6 6 10] [5 9 9 ] [5 10 10] [5 8 8]

so , here the answer is 2 as we can make not more than 2 numbers equal.

Best Approach :

If you explore all of the possible ranges of the each number and do a range update over the big array having such all possible numbers , the maximum of all the range updated array should indicate the number which can be formed by making this update +/- x Є [-k, k].



From the above we can see that , [6 , 7 , 8, 9, 10 ] can be made of maximum frequency with the update.

Time Complexity : O(N ) + O( max(a[i]) +k )

# No Of Lamps Overlapping Visa OA

Imagine that there are several lamps placed on a number line, each of which illuminates some segment of the line. Specifically, the lamps are represented in a two-dimensional array lamps where the ith lamp covers the segment from lamps [i][0] to lamps [i][1], inclusive.

Additionally, you are given a list of control points on this number line, represented by an array points. **Your task is to find the number of lamps that illuminate each control point.** Specifically, for each control point points[j] in the array, your task is to find the number of lamps lamps [i] which include this point within its covered segment - when points[j] lies inside the segment [lamps [i][0], lamps [i][1]].

Best Approach :

Just run the range update for all the lamp light ranges and get the result array at the end.

Now it is just about returning a specific index point as asked.

Time Complexity : O(N + Q )

Space Complexity : O(1)

A screenshot of a computer

Description automatically generated

# Meeting Assistant Hacker Rank OA

Implement a simple meeting assistant. A list of strings, events/n) in the form "<person name <action> <start> <end>" is provided

where person\_name performs action from start to end, both inclusive. Times are formatted HH:MM. Find the earliest time in the day, from "00:00" to "23:59", when all people mentioned in at least one event are available for a meeting of k minutes. Report the answer as "HH MM" or the string -1" if it is not possible.

Example:

events["Alex sleep 00:00 08:00", "Sam sleep 07:00 13:00", "Alex lunch 12:30 13:59

k=60

Alex is not available until 8:00. After that, Sam is not available until 13:00. Then Alex is busy until 13:59. Return the earliest time they are both available, "14:00"

Function Description

Complete the function getEarliest Meet Time in the editor below

getEarliest Meet Time takes the following arguments

string events[n]: event descriptors

int k: meeting duration

string: the earliest time for the meeting or "-1" if it is not possible

Understanding the Question :

There are n events from timing [start.end] ; in each event a particular person is doing some action in that time bracket ; we have to find the timing when everyone mentioned is free so they can have party for k minutes non stop.

Analysis :

We are given n intervals ; we have to find the first point where there is no interval such that k length distance is possible from that point.

Observation1 :- It doesn't matter what is the name of person and what is he doing all matters is time interval of work.

Observation2 :- Convert all the time to minutes ( for the ease of representing on the timeline of minutes 0 🡪 1440 ).

Approach :

Just run the range update for all the minute ranges and get the result array at the end.

TC :- O(N + 1440 + 1440)

Space.-> O(1440) = O(1)

# Count Of Subarrays Of Shortest Length

Find the count. of subarray of shortest length whose sum==k

A = {10,5,2,7,1,9,8,7} k = 15

Eg. subarrays having sum =15 , {10,5} , {5,2,7,1} , {8,7}

Smallest length of subarray having sum 15 is 2 and there are two such subarrays :

{10,5} | {8,7}

Hence the count and output is 2.

Brute Force: Run a nested for loop.

TC :- O(n^2)

Best Approach : Based On Prefix Sum / Hashing

Now let’s say we know that the sum of subarray(i, j) = S, and we also know that sum of subarray(i, x) = S-k where i < x < j. We can conclude that the sum of subarray (x+1, j) = k.

This way we should be able to find the shortest length of the subarray , then using the sliding window algorithm , we should be able to find the number of such sub arrays.

TC :- O(n)

SC :- O(n)

# Count Of Subarrays Of Longest Length

Find the count of subarray of longest length whose sum==k

A = {10,5,2,7,1,9,8,7}   k = 15

O/p: 1

Eg. subarrays having sum =15 , {10,5} , {5,2,7,1} , {8,7}

Longest length is 4

And there is only 1 such subarray present in the array A. {5,2,7,1}

Approach : Same as the above question , except we have to find the longest one here.

# Google OA Pair Search

You are given an array A of N integers , you have to find the number of special pairs in array A.

A pair of indices i and j are called special , if the following 2 conditions are satisfied.

* + 1. i< j
    2. A[A[A[i]]] = A[A[A[j]]]

Find out the number of special pairs present in array A.

Note : 1 based indexing is used.

BruteForce :

Run a nested for loop and check for all the pairs.

TC : O(n^2)

SC : O(1)

Best Approach :

First,let us not bother about this nested level of checking and consider only A[i] == A[j] and i<j only for now ,the way that we would be able to check is that to grow an frequency hashmap, and check if the current element has any such frequency before in the map , which should give the counts of such matches till before that index.

TC : O(n)

SC : O(1)

# Maximum Sum Selection Media Net OA

You are given an integer array A of size N. Your task is to select exactly B elements from either the left end or the right end of the array A in order to maximize the sum of the selected elements.

Here are the details of the problem:

You are given an integer array A of size N where 1 <= N <= 10^5.

You need to select exactly B elements from either the left end or the right end of the array A.

The goal is to find and return the maximum possible sum of elements you can pick.

Input:

An integer array A of size N where each element A[i] satisfies -103 <= A[i] <= 103.

An integer B where 1 <= B <= N.

Output:

An integer representing the maximum possible sum of selected elements.

[5, -2, 3, 1, 2], 3 -> 8

From the below picture , out of all as we see here that {5} , {1,2}

A close-up of a number

Description automatically generated

is the best option available.

Brute Force Approach :

You can take the k values varing from [0 , k] , [1 , k-1 ] , [2 , k-2] , [k , 0] in the left and right sides and find out the sum in all such a cases , then we can find the best / max answer from that.

N = 5 0 1 2 3 4 (indices)

k = 3

i-1 , N - k + i

i Left Right

0 -1 2

1 0 3

2 1 4

3 2 5

TC :- O( k x k )

SC :- O(1)

Best Approach :

In the brute force approach , we are calculating the sum every time , instead if we could just store it in the prefix and suffix sum arrays , it should work quickly with the time complexity of O(k).

TC :- O( k )

SC :- O( N + N )

# Arcesium OA Even and Odd Swap

Two strings, a and b, are said to be twins only if they can be made equivalent by performing some number of operations on one or both strings. There are two possible operations:

SwapEven: Swap a character at an even-numbered index with a character at another even-numbered index.

SwapOdd: Swap a character at an odd-numbered index with a character at another odd-numbered index.

For example, a = "abcd" and b = "cdab" are twins.

Intuition :

Let’s try to solve an easy version of the same problem. Let’s say you can swap any character in any string with any other character of the same string.

This means here that the order doesn't matter;only the quantity matters If “a” is coming for same numbers of times in A and B ; similarly if frequency of “b” is same in both A and B.

Both the strings will be equal if all the characters from “a” to “z” are having the same frequency in both the strings.

Approach :

If we could follow the same approach as above , we could break down the problem into the following.

A 🡪 A0 A1 A2 A3 A4 A5 A6

B 🡪 B0 B1 B2 B3 B4 B5 B6

We can check for the all the even and odd indices separately by breaking them into 2 sets of strings.

TC : - O(Sum of length of both the strings)

SC :- O(26) ~ O(constant) ~ O(1)

# 4Sum II LC

Given four integer arrays nums1, nums2, nums3, and nums4 all of length n, return the number of tuples (i, j, k, l) such that:

0 <= i, j, k, l < n

nums1[i] + nums2[j] + nums3[k] + nums4[l] == 0

Example :

Input: nums1 = [1,2], nums2 = [-2,-1], nums3 = [-1,2], nums4 = [0,2]

Output: 2

Explanation:

The two tuples are:

1. (0, 0, 0, 1) -> nums1[0] + nums2[0] + nums3[0] + nums4[1] = 1 + (-2) + (-1) + 2 = 0

2. (1, 1, 0, 0) -> nums1[1] + nums2[1] + nums3[0] + nums4[0] = 2 + (-1) + (-1) + 0 = 0

Brute Force :

Run 4 level nested for loop

TC : O(n^4)

SC : O(1)

Optimal Approach :

We can eliminate the 4th nested loop and use the hashmap as the look up for the complementary sum for the first 3 elements of the tuple.

a[i] + a[j] + a[k] = - a[l] .

TC : O(n^3)

SC : O(n)

Best Approach :

We can extend the above approach of hashing to 2 level nested loop also where we could first find the sum of all possible pairs of num3 , num4 with their count in a map , then again use a nested for loop on the first and second array , then leverage this map.

TC :- O(n^2) + O(n^2)

SC :- O(n^2)

# Find No Of Triplets Google OA

Given an array of size “N”; find the number of triplets; such that

A[i] >A[j]< A[k] such that i < j < k;

1<=N<=1000

Example: If we take [8 1 2 3 4 5]

We would get the following possible 10 triplets.

8 1 2 8 1 3 8 1 4 8 1 5

8 2 3 8 2 4 8 2 5

8 3 4 8 3 5

8 4 5

Brute Force: Run 3 level nested for loop.

TC : O( n^3 )

SC : O(1)

Best Approach :

In this condition A[i] >A[j]< A[k] , we can split it into 2 levels

i) A[i] >A[j] , we can maintain a prefix array from the point of view of j ,

where , Prefix[j] = number of pairs such that a[i] > a[j] ; j is fixed; all such i are infront of j.

ii) A[j]< A[k] , we can maintain a suffix array from the point of view of j ,

where , Suffix[j] = number of pairs such that a[j] < a[k] ; j is fixed; all such i comes after to j.

Now , we should be able to find the total number of pairs at a specific j , by taking the product of Prefix[j] , Suffix[j] , as for each pair of A[i] >A[j] , we could add each such pair A[j]< A[k] by merging A[j] in the middle.

TC : O( n^2 + n^2 ) ~ O( n ^ 2)

SC : O( n + n ) ~ O(n)

# Find Number Of Quadraplets Google OA

Given an array of size “N”; find the number of quadruplets; such that A[i] > A[j] < A[k] >A[l] such that i < j < k < l ;

1<=N<=1000

Example :

Let's say we have an array of size N = 6:

Array: [5, 2, 8, 4, 6, 1]

Now, let's find the quadruplets where A[i] > A[j] < A[k] > A[l]:

Quadruplet 1: (5, 2, 8, 1) - Here, i = 0, j = 1, k = 2, l = 5

Quadruplet 2: (5, 4, 6, 1) - Here, i = 0, j = 3, k = 4, l = 5

Quadruplet 3: (8, 4, 6, 1) - Here, i = 2, j = 3, k = 4, l = 5

Quadruplet 4: (5, 2, 6, 1) - Here, i = 0, j = 1, k = 4, l = 5

Quadruplet 5: (5, 4, 6, 1) - Here, i = 0, j = 3, k = 4, l = 5

Quadruplet 6: (8, 2, 6, 1) - Here, i = 2, j = 1, k = 4, l = 5

Quadruplet 7: (8, 4, 6, 1) - Here, i = 2, j = 3, k = 4, l = 5

So, for the given array, there are 7 quadruplets that satisfy the condition.

Approach:

This is a follow up question of the above one, we can find prefix[j] , for

A[i] > A[j] and suffix[k] for A[k] >A[l] like above.

Now for each pair of j , k where j < k , we need to check for the condition in the middle A[j] < A[k] , so that if this statisfies , we can find the total number of pairs through merging the pairs by multiplying prefix[j] and suffix[k].

TC : O( n^2 + n^2 + n^2 ) ~ O( n ^ 2)

SC : O( n + n ) ~ O(n)

# Count quadruplets with sum K

Given an array arr[] of size N and an integer S, the task is to find the count of quadruplets present in the given array having sum S.

Brute Force: Run 4 level nested for loop.

TC : O( n^4 )

SC : O(1)

Better Approach :

Run a 2 level nested loop for i , j and at the third level k , maintain a hashmap for k until j changes , then find the frequencies of all the elements for such k.

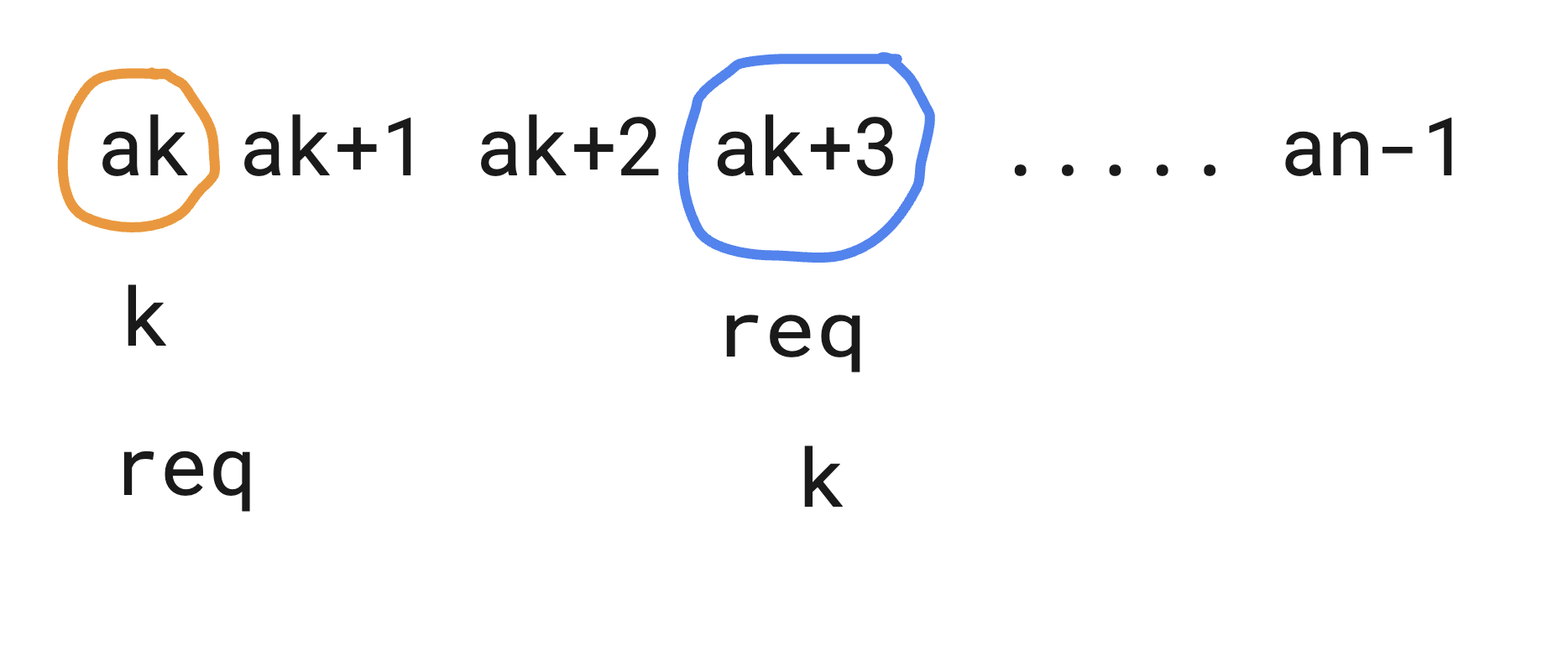
Then , we can check of such k , req = sum – ( A[i] + A[j] + A[k] ) exists in the Map or not and add it’s frequency to count.

Note :

i) When iterating for k and checking for req , it is possible that if ak+3 is req , then when k goes to ak+3 , ak will be it’s req , so in this way we are counting the same quadraple twice , hence we should do count/2 at the end when returning.

TC : O( n^3 )

SC : O(n)



ii) At times it is possible that req = a[k] , which is the case where the element we are looking in the map is the same as the one which is at k ( 3rd and 4th elements in the quadraple are the same. ) In such cases after adding it’s frequencies for req ( it will be a[k] ) , we have to subtract 1 if this condition req = a[k] is true which disregards a[k] it self from the frequency , as it will be coming again ( for example in the below case

1 1 it’s frequency is always taken 2 so we have to remove a[k] as it should not be counted here for req )

Ex:

[ 5 3 1 1 ] S = 10

i j k req = S – ( A[i] + A[j] + A[k] ) = 10 – 9 = 1

here req = 1 = A[k]

Best Approach :

Here , we would follow the same approach as the 2 sum problem , we shall divide the array virtuall into the two halves ,

i) From i to n , we shall reserve it for the 3rd and 4th elements using which we can find req = sum – ( A[i] + A[j] ).

ii) We can then use this ‘req’ to look up for first 2 elements in the range 0 to i , through (A[i] + A[j] ) , and update the frequency of it accordingly in the map.

Question :

1)Why do we need to split the array through 0 to i and i to n ?

We have to do this , as we do not want to overlap the ranges of first 2 , and last 2 elements in the array.

2)Why should we first check for the ‘req’ first in the map , then add it to the map ?

We are following the 2 sum problem , We have to check if the complement exists for last 2 elements , we can form that many pairs with the first 2 elements to complement.

After that only , increase the frequency of first 2 elements sum which can be used by the future elements.

Otherwise , this gives a wrong count of answer as we would get more count than the expected.

TC : O( n^2 )

SC : O(n)

# Shortest Sub Array Length With Sum K

Given an array of size “N”; find the shortest subarray such that the sum of first and last element of the subarray is “k” (Subarray should at least be of size>=2).

Example : [5 6 7 8 10 4 3 2 1] K = 8

Ans : 7

Brute Force: Run 2 level nested for loop.

TC :- O(n^2)

SC :- O(n)

Better Approach :

Use index hashmap and look for the complement in the array , similar to the two sum problem.

TC :- O(n)

SC :- O(n)

Follow Up : How would you find the longest length in the array for the same question ?

We shall not update the hashmap indices , if the same element appeared again also , so that we shall always maintain the length longest index length.

TC :- O(n)

SC :- O(n)

# Minimum Rehabiliation Cost Microsoft OA

We are given an array of size “N”. We have to pick “x” elements at exactly “y” distance; sum of all them should be minimum.

Example :

[1 5 8 1 4 5 8] X = 2; Y =2;

(5+1=6 index 1 and 3 summed up here.)

BruteForce :

We can traverse the array using the nested for loop , where at the each index , we could traverse x times front / back to make the sum of elements which are at y distance.

TC :- O(N\*x ) ~ O(N\*N ) when x is nearly equal to N at worst case

SC :- O(1)

Best Approach :

We can precalculate the prefix sum array at each index which is at a distance y back from that.

prefix[i] = b[i] + b[i-y] + b[i-2y] + b[i-3y] + …………+b[>=0]

prefix[i] = b[i] + prefix[i-y]

This can help us to determine quickly the cost of x elements back at from that index , which are separated by y each.

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

i = 20

x = 4

y = 2

required sum = sum of x numbers from index i at a distance of y

required sum = A[20] + A[18] + A[16] + A[14]

prefix[20] = A[20] + A[18] + A[16] + A[14] + A[12] + ….. + A[0]

prefix[12] = A[12] + A[10] + ….. + A[0]

required sum = prefix[20] - prefix[12] ==> prefix[i] - prefix[i - x\*y]

TC :- O( N )

SC :- O( 1 )

# Find Sum Of All Concatenation

Given an array of non-negative integers, we need to find the sum of concatenation of elements in the array.

Example

For a [10, 2], the output should be solution(a) = 1344

a[0] a[0] = 10.10 = 1010,

a[0] a[1] = 10.2 = 102,

a[1] a[0]=2.10 = 210,

a[1] a[1]=2.2 = 22

So the sum is equal to 1010 + 102 + 210 + 22 = 1344.

BruteForce : Run a nested for loop to find all the possible concatenations including the self , then covert that string to int and find sum.

TC :-> O(N^2\* (2\*|S|) ) = O(N^2\*2\*6) = O(N^2\*12) = O(N^2)

|S| -> will take for converting the string to int

S.-> O(2\*|S|) = O(12) = O(1)

Best Approach :

10 , 2

=>10.10 + 10.2

=>1010 + 102

=>1000 + 10 + 100 + 2

=>1000 + 100 + (10 + 2)

=>( 10 x 10^2 + 10 x 10^1 ) + (10 + 2)

So in a single pass , we could actually break the problem down into finding the length of all the numbers in the array (the max length will be 6 as per constraints )

and raising it to the power of 10 to the number under consideration in the current pass.

TC :- O(N \* 6 + N ) = O(N)

SC :- O(6) = O(1)

# Max Number of K-Sum Pairs

You are given an integer array nums and an integer k.

In one operation, you can pick two numbers from the array whose sum equals k and remove them from the array.

Return the maximum number of operations you can perform on the array.

Input: nums = [3,1,3,4,3], k = 6

Output: 1

Explanation: Starting with nums = [3,1,3,4,3]:

- Remove the first two 3's, then nums = [1,4,3]

There are no more pairs that sum up to 6, hence a total of 1 operation.

Approach :

Since we have to consider the pairs which should not overlap with otherpair elements , we can divide the whole array into 2 groups , one first number and second for it’s complement , and if we know the frequencies of all these numbers , for a pair to be formed from a group , at max we can consider min(freq\_num , freq\_compl) , and if num == compl , then freq\_num /2 only , as we would have repetitions otherwise.

We can take hashmap for finding all the frequencies of the numbers and then traverse the map and put a hashset for marking those and corresponding complements to be visited.

TC :- O(N)

SC :- O(N)

# Max Number Of Sum Pairs

You are given an array of N integers, you want to group the numbers into pairs such that the sum of the elements of each pair is the same. Each element can be a part of at most 1 pair. It is not necessary for all the elements to be part of some pair.

A = [1, 9, 8, 100, 2], should return 2: the pairs are [1, 9] and [8, 2] for a sum of 10.

A = [2, 2, 2, 3], should return 1: [2, 2] (sum 4) OR [2, 3] (sum 5). Notice we can only form sum 4 once since there is overlap between the elements.

A = [2, 2, 2, 2, 2], should return 2: [2, 2] and [2, 2] for a sum for 4. The fifth 2 is not used, while the first four 2's are used.

Understanding :-> You are given an array of size “N”; try to find the maximum number of pairs whose sum is “K”; but “K” is unknown. Fix such a K ; that the number of pairs comes as maximum!

Approach 1 :

You can run the algorithm for K = 1 , K =2 … K = sum of two largest numbers ( maximum pair sum. )

Out of all these choices pick the maximum answer and your job is done.

Approach 2 :

In the above approach , we are actually considering a loop of size

1 to 2 x 109 , as 0<= arr[i] <= 109 , but instead of that , since the possible values of k are only the sums of pairs in array , we can stick with them only

TC : O(N^3) , since N = 100 ; O(1000000) , it will be 10^6 which is still less than 10^8 , so should be good.

# Maximum Value of an Ordered Triplet I

You are given a 0-indexed integer array nums.

Return the maximum value over all triplets of indices (i, j, k) such that i < j < k. If all such triplets have a negative value, return 0.

The value of a triplet of indices (i, j, k) is equal to (nums[i] - nums[j]) \* nums[k].

BruteForce :

Run a three level nested for loop , and check for every possible triplet.

TC : O(N\*N\*N)

SC : O(1)

Better Approach:

For each index k from 2 to n-1; Let’s try to calculate the maximum value of (nums[i]-nums[j])\*nums[k] ( i < j < k ); maximum of all of them will be answer.

So for a given k , We can pre-calculate/build it on the go, the maximum value of (nums[i]-nums[j]) over the range 0 to k-1 for i , j.

Prefix[index] = maximum value of nums[i] - nums[j] from 0 to index.

Prefix[0] can not be determined , as we need two values each for i and j.

Prefix[1] = nums[0] – nums[1] , only this pair is possible.

Prefix[2] = max(nums[0] – nums[1] , nums[0] – nums[2] , nums[1] – nums[2])

🡪 max(Prefix[1] , nums[0] – nums[2] , nums[1] – nums[2] )

That means for every Prefix[x] = max( prefix[x-1] ,

For every y from 0 to x-1 => nums[y] – nums[x] )

After this prefix array is calculated , for every given k from 2 to n , we would have ,

ans = max (pref[k-1]\*nums[k])

TC : O(N^2)

SC : O(1)

Best Approach:

In the above approach , when we are calculating Prefix[x] , if we see we are looping y through 0 to x-1 , to find the max of nums[y] – nums[x] , but here nums[x] is a constant, so which means we can just find maximum nums[y] , then we should be good.

In otherwords , we can maintain a variable which tells the max value till 0 to k-1.

TC : O(N)

SC : O(N) for maintaining the prefix array.

Best Approach with O(1) SC :

In the above approach we need not to maintain the prefix array of size N , we can calculate it’s value on the fly , by maintaining only a single variable.

TC : O(N)

SC : O(1)

# Count the number of mountains

Count the number of mountain subarrays ( length>=3 ) in the array.

What is a mountain subarray ?

Initially the array is strictly increasing and then it is strictly decreasing.

1<=N<=100000

1<=A[i]<=100000000

[1 2 4 2 1]

O/p -> 4

There can be a total of 4 such valid sub arrays are present , hence the answer is 4.

[1 2 4 2]

[2 4 2]

[2 4 2 1]

[1 2 4 2 1]



Best Approach :

Let us maintain 2 arrays

Pref[i] = length of longest increasing subarray which ends at index ‘i’ . (strictly)

Suf[i] = length of longest decreasing subarray which starts at index “i” (strictly)

Now after finding these arrays , now we can easily deduct and answer the following questions

1)To find the length of largest mountain in the array

Just iterate through all indices and find **max(pref[i] + suf[i] -1)** .

-1 is for avoiding the double counting.

2)To find the number of mountains whose peak is at index “i” ,

**(suf[i]-1)\*(pref[i]-1)**

Let’s take [1 2 4 2 1] , and now if peak index is pointing 4



We can see that left part 1 2 4 and right part 4 2 1.

Now if we see from the perceptive of peak i.e 4 , we can see that each of the positions 1 , 2 in left can pair with each of 2 , 1 on the right via the peak , hence we can have the number of mountains at this point of peak 4 will be 2 x 2 = 4.

so total number of mountains can be deducted by adding all at each index as a peak.

TC : O(N)

SC : O(N)

# Count Number Of Valleys

Given an array of N positive integers; find the number of valleys of size >=3.

B = [5 3 4 8]

Number of V-Shapes: - 2

(5,3,4) and (5,3,4,8)

B = [5 4 3 6 7 ]

Number of V-Shapes: - 4

( 4 3 6 )

( 4 3 6 7 )

( 5 4 3 6 )

( 5 4 3 6 7 )

**Brute Force:**

For every possible subarray we need to check if it is a valley or not. The structure for valley should be first it is strictly decreasing then it should be strictly increasing after the breaking point.

For every subarray, find the smallest element in that and check the left and right parts, if there are two same elements, then it will not be a valley.

Time Complexity: O (N^3)

Space Complexity: O (1)

**Best Approach :**

This is very similar to finding number of maintains question

Let’s build 2 arrays first.

left[i] = length of longest decreasing subarray which ends at index ‘i’ . (strictly) from the left side.

right[i] = length of longest decreasing subarray which ends at index ‘i’ . (strictly) from the right side.

TC : O(N)

SC : O(N)

# Largest Subarray Sum At Each Index i

We are given an array of size “N”; for each index “i” ; output the subarray sum which is largest and ends at index “i”.

For example, let’s say we are given the below array

[5 -8 1 1 5]

Now, if we look at the end of each index the largest will be below

[5 -3 1 2 7]

**Naive Approach:**

We can maintain a prefix array which will tell us that at each index the maximum possible sum possible ending to that index, when we traverse forward.

At every index, we would have 2 possible options

* + 1. To see if including that index better the sum to continue streak
    2. Start sum with that index alone.

TC: O(N)

SC: O(N)

**Better Approach:**

We could solve this problem in O(1) space ( by a variable alone ) as well avoiding the need to maintain a separate array.

# Maximum Sum Of Two Non Overlapping SubArray

Find the maximum possible sum of any two non-overlapping subarrays in an array.

**Naive Approach:**

We can try to solve this problem by tracking 2 subarrays sum

For 1st subarray we will find the maximum sum ending at each index i, ‘ ‘i’ will traverse from the 0 to n-2 ( if we include n-1 also , it will be completely one array all alone ).

For 2nd subarray, we will find the maximum sum starting at each index j till the end, j is always greater than i and starts from 1.

We would calculate the 2 arrays

P[i] 🡪 tells us best sum possible of subarray ending at index “i” ;

S[j] 🡪 tells us best possible sum of subarray starting at index “j” ;

We iterate through all pairs (i,j) , to find the maximum non-overlapping subarray sum.

But here is a tricky part, as we iterate through all possible P[i] for all i , we should be good with first subarray sum part , but for the 2nd part , for each of i , we have to look for all j , as we may not immediately conclude that for first ‘j’ alone , S[j] itself is largest , for example

Let’s take [ 6 , -5 , 2 , 1]

P= [6 , 1 , 3 , 4 ]

S= [ 6, -2 , 3 , 1 ]

And when i=0 and j= 1 , we may not take S[1] = -2 value directly as we see the better subarray sum ‘3’ later j = 2.

So, for every given ‘i’, we possibly need to iterate through every ‘j’ after that.

TC O(N^2)

SC O(N)

**Best Approach:**

Instead of iterating through every value of ‘j’ to find the max for a given ‘i’, we could construct a precomputed array which takes care of this.

max\_suff[j] 🡪 provide you the value that is greatest of subarray sum starting from ‘j’.

For a given ‘i’ , we could readily find the max of all possible ‘j’ suffixes through max\_suff[i+1].

TC: O(N)

SC: O(N)

# Maximum Sum Of Two Non Overlapping SubArray with Strictly Increasing 2nd subarray

Find the maximum possible sum of any two non-overlapping subarrays in an array, where the 2nd subarray should be strictly increasing.

Example:

If the input array is [8, -800, 5 , 10 , 1 ] ,

Then, the output will be sum of [8] and [ 5, 10] 🡺 23.

**Approach:**

This question is a follow up of the above questions, here we see that for the suffix array, we need to maintain the suffix array for strictly increasing second subarray sum starting at j.

While computing we need to understand the below 3 scenarios

Case 1:

Let’s take the example [6, 2, 4, 7], here if we assume j at 1 index, we could see the rest of the array is strictly increasing, i.e [2, 4 , 7].

So, if the numbers are positive and strictly increasing, the streak will be giving better sum, so we need to consider all the sum of streak.

Case 2:

But let’s take [6, -5, -3, -1], here in case of negative numbers, we should not consider the whole streak whose sum will be lesser as we add the numbers so forth, so in this case here the sum will be better only if we take -1 alone at the last.

Case 3:

Also, let’s take another case like [6, -5, -3, -1, 100, 125], here we will have the better sum if we take [100, 125] alone.

So, to summarize here, we would have to just check if the current longest increasing streak sum at ‘j’ is better or just consider ‘j’ element alone.

Therefore, while computing suffix array for strictly increasing sum, parallelly, we need to maintain another array also which should store the max of streak sum, current element.

And finally, after that we need to find the max suffix array also, which should readily give me the value of all greatest suffix arrays starting from ‘j’ till the end.

TC: O(N)

SC: O(N)

# Find Number of Sub Arrays

We are given an array of size “N”; find the number of subarrays of length >=3 which are having the same element at start as well as end; and the sum of that subarray (excluding the first and last number) = first number = last number.

Constraints:

1<=N<=300000

1<=c[i]<=1000000000.

Example [9 3 3 3 9]

valid sub arrays

[3 3 3]

[9 3 3 3 9]

Brute Force:

We can traverse all possible subarrays starting at i and ending at j , and check the condition manually .

TC: O(N^3) 🡪 for traversing using 2 loops and then finding the sum after that.

SC: O(N)

Better Approach:

We can avoid the 3rd loop for calculating sum between [i+1, j-1] , if we could pre calculate it using prefix sum array (or) maintain the sum variable also.

TC: O(N^2)

SC: O(N) (if we use prefix array) or O(1) (if we use sum variable alone)

Best Approach:

In the subarray problems with hashmap , analyze the subarray starting at index ‘i’ and ending at index ‘j’ , then deduce a property and derive a equation and use hashmap to solve it.

Let’s first compute the prefix sum array P

P[j] 🡪 denotes the prefix sum till ‘j’

P[i] 🡪 denotes the prefix sum till ‘i’

The sum of the valid sub array ‘g’ will be

g = P[j] – P[i] – arr[j]

A white board with red writing

Description automatically generated

Here, from the reference point of ‘j’, we know the values of P[j] and arr[j] and we also know that, g = arr[i] = arr[j].

g = P[j] – P[i] – arr[j]

P[i] = P[j] – g - arr[j]

P[i] = P[j] – arr[j] - arr[j]

P[i] = P[j] – 2 \* arr[j]

So , when we are at index ‘j’ , if we could find the number of values for P[i] , i.e how many i indices exist before j which satisfies this condition, then we need to check for the values also which are at that i ( it is possible that value at i may be different , but still matches the condition ) , therefore we have to construct the hashmap such that , it would store the values also along with the prefix sum for frequencies.

Map << pref\_sum , value > , freq >

(Or) since the pref\_sum can be always different each time (as arr[i] >= 1), the frequency will be always 1, so we simply maintain it in a set instead of map.

TC: O(N)

SC: O(N)

# Sub Strings in Substring

You are given a string S [1…. n] of length n having only lowercase Eng letters. You are also given q queries each having 2 space separated integers l , r (1 based indexed ). For each query, you need to print the number of substrings of the string S [l … r] which starts and end with the same character.

Brute Force:

We need to check all the possible substrings within the l , r range , if the length is n = r – l + 1 , then we would be going through n\*(n+1) number of substrings

TC : O(N^2) x Q

SC : O(1)

Best Approach:

Let us consider a string ‘arabbc’

[a , r , a , b , b , c , ara , bb]

We could see that the number of substrings with that matching criteria will not change even though we change the order of the characters in the string.

Let’s say even if we have ‘acbrab’

[a , c , b , r , a , b , acbra , brab ]

Hence, the order does not matter really.

So, if we are able to find the frequency of each character in that query range , we will be able to find number of possible such substrings , which will be sum of each frequency of char formula f\*(f+1)/2 ( the trick we are treating each character together as a string and finding num of strings possible out of it and adding all together as the order does not matter here).

But, here we would need to know the frequency of each characters within the specific range [l … r] , for this we need to maintain a prefix array which will be able to tell the frequency of a specific character till a given index of string S.

Prefix[n+1][26]

* The first dimension (n + 1) allows for indexing from 0 to n, where pre[i] will store the count of each character from the beginning of the string up to the i-th character (does not include i).
* The second dimension (26) corresponds to the 26 lowercase English letters ('a' to 'z').
* **pre[i][j]** will contain the count of the **j-th** letter (where **j** ranges from 0 to 25) in the substring **s** to **s[i-1]**.
* We could use size n also, but we need to take special care of **pre[i][j] = (e == j) + (i > 0 ? pre[i - 1][j] : 0);**

TC: O (N x 26) x Q ==> O(N) x Q

SC: O (N x 26)

# Get Max Score of Subsequence

Given an array of size “N”; find a subsequence from an array with a particular property and the sum of that subsequence should be maximum possible.

A[p[i]] - A[p[j]] = p[i] - p[j] , here p —> refers to the position of the element in the array.

Note: A subsequence of length 1 is always balanced.

Example: [1 5 3 7 8]

There are 2 subsequences which may satisfy the above property

[1 3], for the indices [0 2]

[5 7 8], for the indices [1 3 4]

We can take the second subsequence whose sum is the maximum here

Ans 🡪 5 + 7 + 8 = 20

Naive Approach:

We can traverse all the possible subarrays between i , j of the array for this kind of property satisfaction A[i] – A[j] = i – j , tracking the maximum sum of all of them.

T.C: O(N^2)

S.C: O (1)

Best Approach:

If we analyze the property here, we can deduce the 2nd property

A[i] – A[j] = i – j

A[i] – i = A[j] – j

Meaning that at any specific index we need to consider the differences of that index with that element, then if that difference is equal, it will make a subsequence of that elements.

So, first calculate the ‘diff’, then consider a hash map for storing the sum of elements of all equal differences, find the maximum of all that.

TC: O(N)

SC: O(N)

# Good Binary Strings

<https://www.codechef.com/START66B/problems/GOODBINSTR>

Approach (Observation based)

Here , we need to understand one point , by flipping means , we are just flipping only one index at a time and finding the number of such possible flips , so each way corresponds to one flip at a particular index.





Ans :

int goodString(string s)

{

int n = s.length();

**if**(s[0] != s[n-1])

{

**return** 2;

}

**else**{

**return** n-2;

}

}