# 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.

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)

# 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)

# 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’

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

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

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 :

Cases :-

S = abcbacbbac t = abc → unique

S = abcbacbbac t = abbc → duplicate

S = abcbacbbac t = abcp

By analyzing the above 3 cases, the approach is →

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 and ans = min(ans, mp1[t[i]] / mp2[t2[i]] )

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

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

# Good Binary Strings

Approach (Observation based)





# 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 :

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.

# 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.

For a matrix row / col to be filled , we need to find the maximum index in that particular row / column , so that it make sure that all the remaining indices in that row / col are visited (painted ) already .

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

T.C : O(R )

Best Approach

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}

Now after building that prefix sum 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.