**HAPPY CODING**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

3066. Minimum Operations to Exceed Threshold Value II

Medium

You are given a 0-indexed integer array nums, and an integer k.

In one operation, you will:

Take the two smallest integers x and y in nums.

Remove x and y from nums.

Add min(x, y) \* 2 + max(x, y) anywhere in the array.

Note that you can only apply the described operation if nums contains at least two elements.

Return the minimum number of operations needed so that all elements of the array are greater than or equal to k.

Example 1:

Input: nums = [2,11,10,1,3], k = 10

Output: 2

Explanation: In the first operation, we remove elements 1 and 2, then add 1 \* 2 + 2 to nums. nums becomes equal to [4, 11, 10, 3].

In the second operation, we remove elements 3 and 4, then add 3 \* 2 + 4 to nums. nums becomes equal to [10, 11, 10].

At this stage, all the elements of nums are greater than or equal to 10 so we can stop.

It can be shown that 2 is the minimum number of operations needed so that all elements of the array are greater than or equal to 10.

Example 2:

Input: nums = [1,1,2,4,9], k = 20

Output: 4

Explanation: After one operation, nums becomes equal to [2, 4, 9, 3].

After two operations, nums becomes equal to [7, 4, 9].

After three operations, nums becomes equal to [15, 9].

After four operations, nums becomes equal to [33].

At this stage, all the elements of nums are greater than 20 so we can stop.

It can be shown that 4 is the minimum number of operations needed so that all elements of the array are greater than or equal to 20.

Constraints:

2 <= nums.length <= 2 \* 105

1 <= nums[i] <= 109

1 <= k <= 109

The input is generated such that an answer always exists. That is, there exists some sequence of operations after which all elements of the array are greater than or equal to k

Solution:

class Solution {

public int minOperations(int[] nums, int k) {

PriorityQueue<Long> pq = new PriorityQueue<>();

for (int num : nums) {

pq.offer((long) num);

}

int op = 0;

while (pq.size() >= 2) {

long x = pq.poll(); // remove minimum

if (x >= k)

break;

long y = pq.poll();

long res = Math.min(x, y) \* 2 + Math.max(x, y);

pq.offer(res);

op++;

}

return op;

}

}

53. Maximum Subarray

Medium

Given an integer array nums, find the subarray with the largest sum, and return its sum.

Example 1:

Input: nums = [-2,1,-3,4,-1,2,1,-5,4]

Output: 6

Explanation: The subarray [4,-1,2,1] has the largest sum 6.

Example 2:

Input: nums = [1]

Output: 1

Explanation: The subarray [1] has the largest sum 1.

Example 3:

Input: nums = [5,4,-1,7,8]

Output: 23

Explanation: The subarray [5,4,-1,7,8] has the largest sum 23.

Constraints:

1 <= nums.length <= 105

-104 <= nums[i] <= 104

Follow up: If you have figured out the O(n) solution, try coding another solution using the divide and conquer approach, which is more subtle.

Solution:

class Solution {

public int maxSubArray(int[] nums) {

int max = nums[0];

int currSum = nums[0];

for (int i = 1; i < nums.length; i++) { // 1 to (n-1)

if (currSum < 0) {

currSum = 0;

}

currSum = currSum + nums[i];

if (currSum > max) {

max = currSum;

}

}

return max;

}

}

27. Remove Element

Easy

Given an integer array nums and an integer val, remove all occurrences of val in nums in-place. The order of the elements may be changed. Then return the number of elements in nums which are not equal to val.

Consider the number of elements in nums which are not equal to val be k, to get accepted, you need to do the following things:

Change the array nums such that the first k elements of nums contain the elements which are not equal to val. The remaining elements of nums are not important as well as the size of nums.

Return k.

Custom Judge:

The judge will test your solution with the following code:

int[] nums = [...]; // Input array

int val = ...; // Value to remove

int[] expectedNums = [...]; // The expected answer with correct length.

// It is sorted with no values equaling val.

int k = removeElement(nums, val); // Calls your implementation

assert k == expectedNums.length;

sort(nums, 0, k); // Sort the first k elements of nums

for (int i = 0; i < actualLength; i++) {

assert nums[i] == expectedNums[i];

}

If all assertions pass, then your solution will be accepted.

Example 1:

Input: nums = [3,2,2,3], val = 3

Output: 2, nums = [2,2,\_,\_]

Explanation: Your function should return k = 2, with the first two elements of nums being 2.

It does not matter what you leave beyond the returned k (hence they are underscores).

Example 2:

Input: nums = [0,1,2,2,3,0,4,2], val = 2

Output: 5, nums = [0,1,4,0,3,\_,\_,\_]

Explanation: Your function should return k = 5, with the first five elements of nums containing 0, 0, 1, 3, and 4.

Note that the five elements can be returned in any order.

It does not matter what you leave beyond the returned k (hence they are underscores).

Constraints:

0 <= nums.length <= 100

0 <= nums[i] <= 50

0 <= val <= 100

Solution:

class Solution {

public int removeElement(int[] nums, int val) {

int n = nums.length;

int k = 0;

for (int i = 0; i < n; i++) {

if (nums[i] != val) {

nums[k] = nums[i];

k++;

}

}

return k;

}

}

2342. Max Sum of a Pair With Equal Sum of Digits

Medium

You are given a 0-indexed array nums consisting of positive integers. You can choose two indices i and j, such that i != j, and the sum of digits of the number nums[i] is equal to that of nums[j].

Return the maximum value of nums[i] + nums[j] that you can obtain over all possible indices i and j that satisfy the conditions.

Example 1:

Input: nums = [18,43,36,13,7]

Output: 54

Explanation: The pairs (i, j) that satisfy the conditions are:

- (0, 2), both numbers have a sum of digits equal to 9, and their sum is 18 + 36 = 54.

- (1, 4), both numbers have a sum of digits equal to 7, and their sum is 43 + 7 = 50.

So the maximum sum that we can obtain is 54.

Example 2:

Input: nums = [10,12,19,14]

Output: -1

Explanation: There are no two numbers that satisfy the conditions, so we return -1.

Constraints:

1 <= nums.length <= 105

1 <= nums[i] <= 109

Solution:

class Solution {

public int getDigitSum(int num) {

int sum = 0;

while(num>0) {

sum+=(num%10);

num/=10;

}

return sum;

}

public int maximumSum(int[] nums) {

int map[] = new int[82];

int ans=-1;

for(int num : nums) {

int digitSum = getDigitSum(num);

if(map[digitSum]>0) {

int prevNum = map[digitSum];

ans = Math.max(prevNum + num,ans);

map[digitSum] = Math.max(prevNum,num);

} else {

map[digitSum] = num;

}

}

return ans;

}

}

26. Remove Duplicates from Sorted Array

Easy

Given an integer array nums sorted in non-decreasing order, remove the duplicates in-place such that each unique element appears only once. The relative order of the elements should be kept the same. Then return the number of unique elements in nums.

Consider the number of unique elements of nums to be k, to get accepted, you need to do the following things:

Change the array nums such that the first k elements of nums contain the unique elements in the order they were present in nums initially. The remaining elements of nums are not important as well as the size of nums.

Return k.

Custom Judge:

The judge will test your solution with the following code:

int[] nums = [...]; // Input array

int[] expectedNums = [...]; // The expected answer with correct length

int k = removeDuplicates(nums); // Calls your implementation

assert k == expectedNums.length;

for (int i = 0; i < k; i++) {

assert nums[i] == expectedNums[i];

}

If all assertions pass, then your solution will be accepted.

Example 1:

Input: nums = [1,1,2]

Output: 2, nums = [1,2,\_]

Explanation: Your function should return k = 2, with the first two elements of nums being 1 and 2 respectively.

It does not matter what you leave beyond the returned k (hence they are underscores).

Example 2:

Input: nums = [0,0,1,1,1,2,2,3,3,4]

Output: 5, nums = [0,1,2,3,4,\_,\_,\_,\_,\_]

Explanation: Your function should return k = 5, with the first five elements of nums being 0, 1, 2, 3, and 4 respectively.

It does not matter what you leave beyond the returned k (hence they are underscores).

Constraints:

1 <= nums.length <= 3 \* 104

-100 <= nums[i] <= 100

nums is sorted in non-decreasing order.

Solution:

class Solution {

public int removeDuplicates(int[] nums) {

if (nums.length == 0) {

return 0;

}

int k = 1;

for (int i = 1; i < nums.length; i++) {

if (nums[i] != nums[i - 1]) {

nums[k] = nums[i];

k++;

}

}

return k;

}

}

1910. Remove All Occurrences of a Substring

Medium

Given two strings s and part, perform the following operation on s until all occurrences of the substring part are removed:

Find the leftmost occurrence of the substring part and remove it from s.

Return s after removing all occurrences of part.

A substring is a contiguous sequence of characters in a string.

Example 1:

Input: s = "daabcbaabcbc", part = "abc"

Output: "dab"

Explanation: The following operations are done:

- s = "daabcbaabcbc", remove "abc" starting at index 2, so s = "dabaabcbc".

- s = "dabaabcbc", remove "abc" starting at index 4, so s = "dababc".

- s = "dababc", remove "abc" starting at index 3, so s = "dab".

Now s has no occurrences of "abc".

Example 2:

Input: s = "axxxxyyyyb", part = "xy"

Output: "ab"

Explanation: The following operations are done:

- s = "axxxxyyyyb", remove "xy" starting at index 4 so s = "axxxyyyb".

- s = "axxxyyyb", remove "xy" starting at index 3 so s = "axxyyb".

- s = "axxyyb", remove "xy" starting at index 2 so s = "axyb".

- s = "axyb", remove "xy" starting at index 1 so s = "ab".

Now s has no occurrences of "xy".

Constraints:

1 <= s.length <= 1000

1 <= part.length <= 1000

s​​​​​​ and part consists of lowercase English letters.

Solution:

class Solution {

public String removeOccurrences(String s, String part) {

StringBuilder sb = new StringBuilder();

int n = s.length();

int m = part.length();

for(int i=0; i<n; i++) {

sb.append(s.charAt(i));

if(sb.length()>=m) {

String sub = sb.substring(sb.length()-m);

if(sub.equals(part)) {

sb.setLength(sb.length()-m);

}

}

}

return sb.toString();

}

}

1352. Product of the Last K Numbers

Medium

Design an algorithm that accepts a stream of integers and retrieves the product of the last k integers of the stream.

Implement the ProductOfNumbers class:

ProductOfNumbers() Initializes the object with an empty stream.

void add(int num) Appends the integer num to the stream.

int getProduct(int k) Returns the product of the last k numbers in the current list. You can assume that always the current list has at least k numbers.

The test cases are generated so that, at any time, the product of any contiguous sequence of numbers will fit into a single 32-bit integer without overflowing.

Example:

Input

["ProductOfNumbers","add","add","add","add","add","getProduct","getProduct","getProduct","add","getProduct"]

[[],[3],[0],[2],[5],[4],[2],[3],[4],[8],[2]]

Output

[null,null,null,null,null,null,20,40,0,null,32]

Explanation

ProductOfNumbers productOfNumbers = new ProductOfNumbers();

productOfNumbers.add(3); // [3]

productOfNumbers.add(0); // [3,0]

productOfNumbers.add(2); // [3,0,2]

productOfNumbers.add(5); // [3,0,2,5]

productOfNumbers.add(4); // [3,0,2,5,4]

productOfNumbers.getProduct(2); // return 20. The product of the last 2 numbers is 5 \* 4 = 20

productOfNumbers.getProduct(3); // return 40. The product of the last 3 numbers is 2 \* 5 \* 4 = 40

productOfNumbers.getProduct(4); // return 0. The product of the last 4 numbers is 0 \* 2 \* 5 \* 4 = 0

productOfNumbers.add(8); // [3,0,2,5,4,8]

productOfNumbers.getProduct(2); // return 32. The product of the last 2 numbers is 4 \* 8 = 32

Constraints:

0 <= num <= 100

1 <= k <= 4 \* 104

At most 4 \* 104 calls will be made to add and getProduct.

The product of the stream at any point in time will fit in a 32-bit integer.

Follow-up: Can you implement both GetProduct and Add to work in O(1) time complexity instead of O(k) time complexity?

Solution:

class ProductOfNumbers {

ArrayList<Integer> list = new ArrayList<>();

public ProductOfNumbers() {

// create empty stream

list.clear();

}

public void add(int num) {

// append operation

if (num == 0) {

list.clear();

return;

}

int prev = (list.size() == 0) ? 1 : list.get(list.size() - 1);

list.add(num \* prev);

}

public int getProduct(int k) {

int s = list.size();

if (s < k) {

return 0;

} else if (s == k) {

return list.get(s - 1);

} else {

return (list.get(s - 1) / list.get(s - 1 - k));

}

}

}

121. Best Time to Buy and Sell Stock

Easy

You are given an array prices where prices[i] is the price of a given stock on the ith day.

You want to maximize your profit by choosing a single day to buy one stock and choosing a different day in the future to sell that stock.

Return the maximum profit you can achieve from this transaction. If you cannot achieve any profit, return 0.

Example 1:

Input: prices = [7,1,5,3,6,4]

Output: 5

Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.

Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

Example 2:

Input: prices = [7,6,4,3,1]

Output: 0

Explanation: In this case, no transactions are done and the max profit = 0.

Constraints:

1 <= prices.length <= 105

0 <= prices[i] <= 104

Solution:

class Solution {

public int maxProfit(int[] prices) {

int n = prices.length;

int buyPrice = Integer.MAX\_VALUE;

int maxProfit = 0;

for (int i = 0; i < n; i++) {

if (buyPrice < prices[i]) {

int profit = prices[i] - buyPrice;

maxProfit = Math.max(maxProfit, profit);

} else {

buyPrice = prices[i];

}

}

return maxProfit;

}

}

2698. Find the Punishment Number of an Integer

Medium

Given a positive integer n, return the punishment number of n.

The punishment number of n is defined as the sum of the squares of all integers i such that:

1 <= i <= n

The decimal representation of i \* i can be partitioned into contiguous substrings such that the sum of the integer values of these substrings equals i.

Example 1:

Input: n = 10

Output: 182

Explanation: There are exactly 3 integers i in the range [1, 10] that satisfy the conditions in the statement:

- 1 since 1 \* 1 = 1

- 9 since 9 \* 9 = 81 and 81 can be partitioned into 8 and 1 with a sum equal to 8 + 1 == 9.

- 10 since 10 \* 10 = 100 and 100 can be partitioned into 10 and 0 with a sum equal to 10 + 0 == 10.

Hence, the punishment number of 10 is 1 + 81 + 100 = 182

Example 2:

Input: n = 37

Output: 1478

Explanation: There are exactly 4 integers i in the range [1, 37] that satisfy the conditions in the statement:

- 1 since 1 \* 1 = 1.

- 9 since 9 \* 9 = 81 and 81 can be partitioned into 8 + 1.

- 10 since 10 \* 10 = 100 and 100 can be partitioned into 10 + 0.

- 36 since 36 \* 36 = 1296 and 1296 can be partitioned into 1 + 29 + 6.

Hence, the punishment number of 37 is 1 + 81 + 100 + 1296 = 1478

Constraints:

1 <= n <= 1000

Solution:

class Solution {

public boolean isPartition(int j, String i2, int i, int currSum) {

int n = i2.length();

if (j == n) {

return (currSum == i);

}

if (currSum > i) {

return false;

}

for (int index = j; index < n; index++) {

int val = Integer.parseInt(i2.substring(j, index + 1));

if (isPartition(index + 1, i2, i, currSum + val)) {

return true;

}

}

return false;

}

public int punishmentNumber(int n) {

int res = 0;

for (int i = 1; i <= n; i++) {

String i2 = Integer.toString(i \* i);

if (isPartition(0, i2, i, 0)) {

res += (i \* i);

}

}

return res;

}

}

152. Maximum Product Subarray

Medium

Given an integer array nums, find a subarraythat has the largest product, and return the product.

The test cases are generated so that the answer will fit in a 32-bit integer.

Example 1:

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

Output: 6

Explanation: [2,3] has the largest product 6.

Example 2:

Input: nums = [-2,0,-1]

Output: 0

Explanation: The result cannot be 2, because [-2,-1] is not a subarray.

Constraints:

1 <= nums.length <= 2 \* 104

-10 <= nums[i] <= 10

The product of any subarray of nums is guaranteed to fit in a 32-bit integer.

Solution:

class Solution {

public int maxProduct(int[] nums) {

if (nums == null || nums.length == 0) {

return 0;

}

int maxProduct = nums[0];

int minProduct = nums[0];

int result = nums[0];

for (int i = 1; i < nums.length; i++) {

if (nums[i] >= 0) {

maxProduct = Math.max(nums[i], maxProduct \* nums[i]);

minProduct = Math.min(nums[i], minProduct \* nums[i]);

} else {

int temp = maxProduct;

maxProduct = Math.max(nums[i], minProduct \* nums[i]);

minProduct = Math.min(nums[i], temp \* nums[i]);

}

result = Math.max(result, maxProduct);

}

return result;

}

}

3066. Minimum Operations to Exceed Threshold Value II

Medium

You are given a 0-indexed integer array nums, and an integer k.

You are allowed to perform some operations on nums, where in a single operation, you can:

Select the two smallest integers x and y from nums.

Remove x and y from nums.

Insert (min(x, y) \* 2 + max(x, y)) at any position in the array.

Note that you can only apply the described operation if nums contains at least two elements.

Return the minimum number of operations needed so that all elements of the array are greater than or equal to k.

Example 1:

Input: nums = [2,11,10,1,3], k = 10

Output: 2

Explanation:

In the first operation, we remove elements 1 and 2, then add 1 \* 2 + 2 to nums. nums becomes equal to [4, 11, 10, 3].

In the second operation, we remove elements 3 and 4, then add 3 \* 2 + 4 to nums. nums becomes equal to [10, 11, 10].

At this stage, all the elements of nums are greater than or equal to 10 so we can stop.

It can be shown that 2 is the minimum number of operations needed so that all elements of the array are greater than or equal to 10.

Example 2:

Input: nums = [1,1,2,4,9], k = 20

Output: 4

Explanation:

After one operation, nums becomes equal to [2, 4, 9, 3].

After two operations, nums becomes equal to [7, 4, 9].

After three operations, nums becomes equal to [15, 9].

After four operations, nums becomes equal to [33].

At this stage, all the elements of nums are greater than 20 so we can stop.

It can be shown that 4 is the minimum number of operations needed so that all elements of the array are greater than or equal to 20.

Constraints:

2 <= nums.length <= 2 \* 105

1 <= nums[i] <= 109

1 <= k <= 109

The input is generated such that an answer always exists. That is, there exists some sequence of operations after which all elements of the array are greater than or equal to k.

Solution:

class Solution {

public int minOperations(int[] nums, int k) {

PriorityQueue<Long> pq = new PriorityQueue<>();

for (int num : nums) {

pq.offer((long) num);

}

int op = 0;

while (pq.size() >= 2) {

long x = pq.poll(); // remove minimum

if (x >= k)

break;

long y = pq.poll();

long res = Math.min(x, y) \* 2 + Math.max(x, y);

pq.offer(res);

op++;

}

return op;

}

}

238. Product of Array Except Self

Medium

Given an integer array nums, return an array answer such that answer[i] is equal to the product of all the elements of nums except nums[i].

The product of any prefix or suffix of nums is guaranteed to fit in a 32-bit integer.

You must write an algorithm that runs in O(n) time and without using the division operation.

Example 1:

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

Output: [24,12,8,6]

Example 2:

Input: nums = [-1,1,0,-3,3]

Output: [0,0,9,0,0]

Constraints:

2 <= nums.length <= 105

-30 <= nums[i] <= 30

The input is generated such that answer[i] is guaranteed to fit in a 32-bit integer.

Follow up: Can you solve the problem in O(1) extra space complexity? (The output array does not count as extra space for space complexity analysis.)

Solution:

class Solution {

public int[] productExceptSelf(int[] nums) {

int n = nums.length;

int right[] = new int[n];

int left[] = new int[n];

int output[] = new int[n];

left[0] = 1;

right[n - 1] = 1;

for (int i = 1; i < n; i++) {

left[i] = left[i - 1] \* nums[i - 1];

}

for (int i = n - 2; i >= 0; i--) {

right[i] = right[i + 1] \* nums[i + 1];

}

for (int i = 0; i < n; i++) {

output[i] = left[i] \* right[i];

}

return output;

}

}

1718. Construct the Lexicographically Largest Valid Sequence

Medium

Given an integer n, find a sequence that satisfies all of the following:

The integer 1 occurs once in the sequence.

Each integer between 2 and n occurs twice in the sequence.

For every integer i between 2 and n, the distance between the two occurrences of i is exactly i.

The distance between two numbers on the sequence, a[i] and a[j], is the absolute difference of their indices, |j - i|.

Return the lexicographically largest sequence. It is guaranteed that under the given constraints, there is always a solution.

A sequence a is lexicographically larger than a sequence b (of the same length) if in the first position where a and b differ, sequence a has a number greater than the corresponding number in b. For example, [0,1,9,0] is lexicographically larger than [0,1,5,6] because the first position they differ is at the third number, and 9 is greater than 5.

Example 1:

Input: n = 3

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

Explanation: [2,3,2,1,3] is also a valid sequence, but [3,1,2,3,2] is the lexicographically largest valid sequence.

Example 2:

Input: n = 5

Output: [5,3,1,4,3,5,2,4,2]

Constraints:

1 <= n <= 20

Solution:

class Solution {

public int[] constructDistancedSequence(int n) {

boolean used[] = new boolean[n+1];

int seq[] = new int[2\*n-1];

backtrack(0,used, seq, n);

return seq;

}

public boolean backtrack(int index, boolean used[], int seq[], int n){

while(index < seq.length && seq[index]!=0) index++;

if(index == seq.length) return true;

for(int i=n;i>=1;i--){

if(used[i]) continue;

if(i==1){

seq[index] = i;

used[i] = true;

if(backtrack(index+1,used,seq,n)) return true;

// if we cannot find answer

seq[index]=0;

used[i] = false;

}else if(index + i < seq.length && seq[index+i] == 0){

seq[index] = i;

seq[index + i] = i;

used[i] = true;

if(backtrack(index+1,used,seq,n)) return true;

seq[index]=0;

seq[index + i]=0;

used[i] = false;

}

}

return false;

}

}

1079. Letter Tile Possibilities

Medium

You have n tiles, where each tile has one letter tiles[i] printed on it.

Return the number of possible non-empty sequences of letters you can make using the letters printed on those tiles.

Example 1:

Input: tiles = "AAB"

Output: 8

Explanation: The possible sequences are "A", "B", "AA", "AB", "BA", "AAB", "ABA", "BAA".

Example 2:

Input: tiles = "AAABBC"

Output: 188

Example 3:

Input: tiles = "V"

Output: 1

Constraints:

1 <= tiles.length <= 7

tiles consists of uppercase English letters.

Solution:

class Solution {

int len;

public int numTilePossibilities(String tiles) {

len = tiles.length();

boolean used[] = new boolean[len];

HashSet<String> set = new HashSet<>();

backtrack(tiles, used, set, "");

return set.size()-1;

}

public void backtrack(String tiles, boolean used[], HashSet<String> set, String cur) {

if(set.contains(cur)) return;

set.add(cur);

for(int i=0; i<len; i++) {

if(used[i]) continue;

used[i] = true;

backtrack(tiles, used, set, cur + tiles.charAt(i));

used[i] = false;

}

}

}

Optimized Solution:

class Solution {

int len;

public int numTilePossibilities(String tiles) {

len = tiles.length();

int freq[] = new int[26];

for(int i=0; i<len; i++) {

freq[tiles.charAt(i) - 'A']++;

}

int count = backtrack(freq);

return count;

}

public int backtrack(int freq[]) {

int count = 0;

for (int i = 0; i < 26; i++) {

if (freq[i]==0) continue;

count++;

freq[i]--;

count += backtrack(freq);

freq[i]++;

}

return count;

}

}

2375. Construct Smallest Number From DI String

Medium

You are given a 0-indexed string pattern of length n consisting of the characters 'I' meaning increasing and 'D' meaning decreasing.

A 0-indexed string num of length n + 1 is created using the following conditions:

num consists of the digits '1' to '9', where each digit is used at most once.

If pattern[i] == 'I', then num[i] < num[i + 1].

If pattern[i] == 'D', then num[i] > num[i + 1].

Return the lexicographically smallest possible string num that meets the conditions.

Example 1:

Input: pattern = "IIIDIDDD"

Output: "123549876"

Explanation:

At indices 0, 1, 2, and 4 we must have that num[i] < num[i+1].

At indices 3, 5, 6, and 7 we must have that num[i] > num[i+1].

Some possible values of num are "245639871", "135749862", and "123849765".

It can be proven that "123549876" is the smallest possible num that meets the conditions.

Note that "123414321" is not possible because the digit '1' is used more than once.

Example 2:

Input: pattern = "DDD"

Output: "4321"

Explanation:

Some possible values of num are "9876", "7321", and "8742".

It can be proven that "4321" is the smallest possible num that meets the conditions.

Constraints:

1 <= pattern.length <= 8

pattern consists of only the letters 'I' and 'D'.

Solution:

class Solution {

public String smallestNumber(String pattern) {

int n = pattern.length();

boolean[] used = new boolean[10];

StringBuilder result = new StringBuilder();

backtrack(pattern, 0, new int[n + 1], used, result);

return result.toString();

}

private boolean backtrack(String pattern, int index, int[] num, boolean[] used, StringBuilder result) {

if (index > pattern.length()) {

for (int i = 0; i < num.length; i++) {

result.append(num[i]);

}

return true; // found valid lexicographically smallest number

}

for (int digit = 1; digit <= 9; digit++) {

if (!used[digit] && (index == 0 || isValid(num[index - 1], digit, pattern.charAt(index - 1)))) {

used[digit] = true;

num[index] = digit;

if(backtrack(pattern, index + 1, num, used, result)) {

return true;

}

num[index] = 0;

used[digit] = false; // Backtrack

}

}

return false;

}

private boolean isValid(int lastDigit, int currentDigit, char condition) {

return (condition == 'I' && lastDigit < currentDigit) || (condition == 'D' && lastDigit > currentDigit);

}

}

78. Subsets

Solved

Given an integer array nums of unique elements, return all possible

subsets (the power set).

The solution set must not contain duplicate subsets. Return the solution in any order.

Example 1:

Input: nums = [1,2,3]

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

Example 2:

Input: nums = [0]

Output: [[],[0]]

Constraints:

1 <= nums.length <= 10

-10 <= nums[i] <= 10

All the numbers of nums are unique.

Solution:

class Solution {

//this is a global list

List<List<Integer>> res = new ArrayList<>();

public List<List<Integer>> subsets(int[] nums) {

findSubsets(nums, 0, new ArrayList<>());

return res;

}

public void findSubsets(int nums[], int index, List<Integer> sublist) {

//base case

if(index == nums.length) {

res.add(new ArrayList<>(sublist));

return;

}

//Yes case

sublist.add(nums[index]);

findSubsets(nums, index+1, sublist);

//while backtracking we need to remove the last added element

sublist.remove(sublist.size()-1);

//No case

findSubsets(nums, index+1, sublist);

}

}

1415. The k-th Lexicographical String of All Happy Strings of Length n

Medium

A happy string is a string that:

consists only of letters of the set ['a', 'b', 'c'].

s[i] != s[i + 1] for all values of i from 1 to s.length - 1 (string is 1-indexed).

For example, strings "abc", "ac", "b" and "abcbabcbcb" are all happy strings and strings "aa", "baa" and "ababbc" are not happy strings.

Given two integers n and k, consider a list of all happy strings of length n sorted in lexicographical order.

Return the kth string of this list or return an empty string if there are less than k happy strings of length n.

Example 1:

Input: n = 1, k = 3

Output: "c"

Explanation: The list ["a", "b", "c"] contains all happy strings of length 1. The third string is "c".

Example 2:

Input: n = 1, k = 4

Output: ""

Explanation: There are only 3 happy strings of length 1.

Example 3:

Input: n = 3, k = 9

Output: "cab"

Explanation: There are 12 different happy string of length 3 ["aba", "abc", "aca", "acb", "bab", "bac", "bca", "bcb", "cab", "cac", "cba", "cbc"]. You will find the 9th string = "cab"

Constraints:

1 <= n <= 10

1 <= k <= 100

Solution:

class Solution {

// global variable

String res;

int count;

public String getHappyString(int n, int k) {

count = 0;

res = "";

backtrack(n, k, new StringBuilder(""));

return res;

}

public boolean backtrack(int n, int k, StringBuilder cur) {

// base case

if (cur.length() == n) {

count++;

if (count == k) {

res = cur.toString();

return true;

}

return false;

}

for (char ch = 'a'; ch <= 'c'; ch++) {

int len = cur.length();

if (len > 0 && cur.charAt(len - 1) == ch)

continue;

cur.append(ch);

if (backtrack(n, k, cur)) {

return true;

}

cur.deleteCharAt(cur.length() - 1);

}

return false;

}

}

51. N-Queens

Hard

The n-queens puzzle is the problem of placing n queens on an n x n chessboard such that no two queens attack each other.

Given an integer n, return all distinct solutions to the n-queens puzzle. You may return the answer in any order.

Each solution contains a distinct board configuration of the n-queens' placement, where 'Q' and '.' both indicate a queen and an empty space, respectively.

Example 1:

Input: n = 4

Output: [[".Q..","...Q","Q...","..Q."],["..Q.","Q...","...Q",".Q.."]]

Explanation: There exist two distinct solutions to the 4-queens puzzle as shown above

Example 2:

Input: n = 1

Output: [["Q"]]

Constraints:

1 <= n <= 9

Solution:

public class Solution {

private List<List<String>> result = new ArrayList<>();

public List<List<String>> solveNQueens(int n) {

if (n == 0) {

return result;

}

List<String> board = new ArrayList<>();

// For n = 3, board = {"...", "...", "..."} initially

for (int i = 0; i < n; i++) {

StringBuilder row = new StringBuilder();

for (int j = 0; j < n; j++) {

row.append('.');

}

board.add(row.toString());

}

solve(board, 0);

return result;

}

private boolean isValid(List<String> board, int row, int col) {

// Look for up

for (int i = row; i >= 0; i--) {

if (board.get(i).charAt(col) == 'Q')

return false;

}

// Check left diagonal upwards

for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {

if (board.get(i).charAt(j) == 'Q')

return false;

}

// Check right diagonal upwards

for (int i = row, j = col; i >= 0 && j < board.size(); i--, j++) {

if (board.get(i).charAt(j) == 'Q')

return false;

}

return true;

}

private void solve(List<String> board, int row) {

if (row == board.size()) {

result.add(new ArrayList<>(board));

return;

}

for (int i = 0; i < board.size(); i++) {

if (isValid(board, row, i)) { //this is the condition for checking is our queens are exit left & and right diagonally, up side

StringBuilder newRow = new StringBuilder(board.get(row));

newRow.setCharAt(i, 'Q');

board.set(row, newRow.toString());

solve(board, row + 1);

newRow.setCharAt(i, '.');

board.set(row, newRow.toString());

}

}

}

}

1980. Find Unique Binary String

Medium

Given an array of strings nums containing n unique binary strings each of length n, return a binary string of length n that does not appear in nums. If there are multiple answers, you may return any of them.

Example 1:

Input: nums = ["01","10"]

Output: "11"

Explanation: "11" does not appear in nums. "00" would also be correct.

Example 2:

Input: nums = ["00","01"]

Output: "11"

Explanation: "11" does not appear in nums. "10" would also be correct

Example 3:

Input: nums = ["111","011","001"]

Output: "101"

Explanation: "101" does not appear in nums. "000", "010", "100", and "110" would also be correct.

Constraints:

n == nums.length

1 <= n <= 16

nums[i].length == n

nums[i] is either '0' or '1'.

All the strings of nums are unique.

Solutions:

//O(2^N)

//This is the brute force solution

class Solution {

StringBuilder res;

public String findDifferentBinaryString(String[] nums) {

int n = nums.length;

HashSet<String> set = new HashSet<>();

for(String num : nums){

set.add(num);

}

res=new StringBuilder("");

backtrack(nums,n,set,res);

return res.toString();

}

public boolean backtrack(String nums[], int n, HashSet<String> set, StringBuilder res){

//base case

if(res.length() == n){

if(!set.contains(res.toString())){

return true;

}

return false;

}

for(char ch='0';ch<='1';ch++){

res.append(ch);

if(backtrack(nums,n,set,res)){

return true;

}

res.deleteCharAt(res.length()-1);

}

return false;

}

}

Optimized solution:

class Solution {

public String findDifferentBinaryString(String[] nums) {

int n = nums.length;

StringBuilder res = new StringBuilder("");

for(int i=0; i<n; i++) {

char ch = (nums[i].charAt(i)== '0'?'1':'0');

res.append(ch);

}

return res.toString();

}

}

1261. Find Elements in a Contaminated Binary Tree

Medium

Given a binary tree with the following rules:

root.val == 0

For any treeNode:

If treeNode.val has a value x and treeNode.left != null, then treeNode.left.val == 2 \* x + 1

If treeNode.val has a value x and treeNode.right != null, then treeNode.right.val == 2 \* x + 2

Now the binary tree is contaminated, which means all treeNode.val have been changed to -1.

Implement the FindElements class:

FindElements(TreeNode\* root) Initializes the object with a contaminated binary tree and recovers it.

bool find(int target) Returns true if the target value exists in the recovered binary tree.

Example 1:

Input

["FindElements","find","find"]

[[[-1,null,-1]],[1],[2]]

Output

[null,false,true]

Explanation

FindElements findElements = new FindElements([-1,null,-1]);

findElements.find(1); // return False

findElements.find(2); // return True

Example 2:

Input

["FindElements","find","find","find"]

[[[-1,-1,-1,-1,-1]],[1],[3],[5]]

Output

[null,true,true,false]

Explanation

FindElements findElements = new FindElements([-1,-1,-1,-1,-1]);

findElements.find(1); // return True

findElements.find(3); // return True

findElements.find(5); // return False

Example 3:

Input

["FindElements","find","find","find","find"]

[[[-1,null,-1,-1,null,-1]],[2],[3],[4],[5]]

Output

[null,true,false,false,true]

Explanation

FindElements findElements = new FindElements([-1,null,-1,-1,null,-1]);

findElements.find(2); // return True

findElements.find(3); // return False

findElements.find(4); // return False

findElements.find(5); // return True

Constraints:

TreeNode.val == -1

The height of the binary tree is less than or equal to 20

The total number of nodes is between [1, 104]

Total calls of find() is between [1, 104]

0 <= target <= 106

Solution:

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode() {}

\* TreeNode(int val) { this.val = val; }

\* TreeNode(int val, TreeNode left, TreeNode right) {

\* this.val = val;

\* this.left = left;

\* this.right = right;

\* }

\* }

\*/

class FindElements {

HashSet<Integer> set = new HashSet<>();

public FindElements(TreeNode root) {

dfs(root,0);

}

public boolean find(int target) {

return set.contains(target);

}

public void dfs(TreeNode root, int val){

if(root==null) return;

root.val = val;

set.add(val);

dfs(root.left, 2\*val+1);

dfs(root.right, 2\*val+2);

}

}

/\*\*

\* Your FindElements object will be instantiated and called as such:

\* FindElements obj = new FindElements(root);

\* boolean param\_1 = obj.find(target);

\*/

[**1028. Recover a Tree From Preorder Traversal**](https://leetcode.com/problems/recover-a-tree-from-preorder-traversal/)

Hard

We run a preorder depth-first search (DFS) on the root of a binary tree.

At each node in this traversal, we output D dashes (where D is the depth of this node), then we output the value of this node.  If the depth of a node is D, the depth of its immediate child is D + 1.  The depth of the root node is 0.

If a node has only one child, that child is guaranteed to be **the left child**.

Given the output traversal of this traversal, recover the tree and return *its* root.

**Example 1:**



**Input:** traversal = "1-2--3--4-5--6--7"

**Output:** [1,2,5,3,4,6,7]

**Example 2:**



**Input:** traversal = "1-2--3---4-5--6---7"

**Output:** [1,2,5,3,null,6,null,4,null,7]

**Example 3:**



**Input:** traversal = "1-401--349---90--88"

**Output:** [1,401,null,349,88,90]

**Constraints:**

* The number of nodes in the original tree is in the range [1, 1000].
* 1 <= Node.val <= 109

**Solution**:

/\*\*

 \* Definition for a binary tree node.

 \* public class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode() {}

 \*     TreeNode(int val) { this.val = val; }

 \*     TreeNode(int val, TreeNode left, TreeNode right) {

 \*         this.val = val;

 \*         this.left = left;

 \*         this.right = right;

 \*     }

 \* }

 \*/

class Solution {

    int index=0;

    int n=0;

    public TreeNode recoverFromPreorder(String traversal) {

        n = traversal.length();

        return recur(traversal,0);

    }

    public TreeNode recur(String traversal, int depth){

        if(index >= n) return null;

        //count the dash

        int count=0;

        int tempI=index;

        while(tempI < n && !Character.isDigit(traversal.charAt(tempI))){

            count++;

            tempI++;

        }

        if(count!=depth) return null;

        index = tempI;

        //find the number

        int val=0;

        while(index < n && Character.isDigit(traversal.charAt(index))){

            val = val \* 10 + (traversal.charAt(index) - '0');

            index++;

        }

        TreeNode node = new TreeNode(val);

        node.left = recur(traversal, depth+1);

        node.right = recur(traversal, depth+1);

        return node;

    }

}

Optimized Solution:

/\*\*

 \* Definition for a binary tree node.

 \* public class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode() {}

 \*     TreeNode(int val) { this.val = val; }

 \*     TreeNode(int val, TreeNode left, TreeNode right) {

 \*         this.val = val;

 \*         this.left = left;

 \*         this.right = right;

 \*     }

 \* }

 \*/

// iterative stack

class Solution {

    int index=0;

    int n=0;

    public TreeNode recoverFromPreorder(String traversal) {

        n = traversal.length();

        return recur(traversal);

    }

    public TreeNode recur(String traversal){

        Stack<TreeNode> stack = new Stack<>();

        // if(index >= n) return null;

        while(index < n){

             //count the dash

            int count=0;

            while(index < n && !Character.isDigit(traversal.charAt(index))){

                count++;

                index++;

            }

            int depth = stack.size();

            while(depth > count){

                stack.pop();

                depth--;

            }

            //find the number

            int val=0;

            while(index < n && Character.isDigit(traversal.charAt(index))){

                val = val \* 10 + (traversal.charAt(index) - '0');

                index++;

            }

            TreeNode node = new TreeNode(val);

            if(!stack.isEmpty()){

                if(stack.peek().left == null){

                    stack.peek().left = node;

                }else{

                    stack.peek().right = node;

                }

            }

            stack.push(node);

        }

        while(stack.size()>1){

            stack.pop();

        }

        return stack.peek();

    }

}

[**889. Construct Binary Tree from Preorder and Postorder Traversal**](https://leetcode.com/problems/construct-binary-tree-from-preorder-and-postorder-traversal/)

Medium

Given two integer arrays, preorder and postorder where preorder is the preorder traversal of a binary tree of **distinct** values and postorder is the postorder traversal of the same tree, reconstruct and return *the binary tree*.

If there exist multiple answers, you can **return any** of them.

**Example 1:**



**Input:** preorder = [1,2,4,5,3,6,7], postorder = [4,5,2,6,7,3,1]

**Output:** [1,2,3,4,5,6,7]

**Example 2:**

**Input:** preorder = [1], postorder = [1]

**Output:** [1]

**Constraints:**

* 1 <= preorder.length <= 30
* 1 <= preorder[i] <= preorder.length
* All the values of preorder are **unique**.
* postorder.length == preorder.length
* 1 <= postorder[i] <= postorder.length
* All the values of postorder are **unique**.
* It is guaranteed that preorder and postorder are the preorder traversal and postorder traversal of the same binary tree.

Solution:

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode() {}

\* TreeNode(int val) { this.val = val; }

\* TreeNode(int val, TreeNode left, TreeNode right) {

\* this.val = val;

\* this.left = left;

\* this.right = right;

\* }

\* }

\*/

// Approach-1 (brute force)

// T.C : O(n^2)

// S.C : O(n) for System Stack used for Recursion

class Solution {

public TreeNode solve(int prestart, int poststart, int preend, int[] preorder, int[] postorder) {

if (prestart > preend) {

return null;

}

TreeNode root = new TreeNode(preorder[prestart]);

if (prestart == preend) {

return root;

}

int nextNode = preorder[prestart + 1]; // root of left subtree

int j = poststart;

while (postorder[j] != nextNode) {

j++;

}

int num = j - poststart + 1;

root.left = solve(prestart + 1, poststart, prestart + num, preorder, postorder);

root.right = solve(prestart + num + 1, j + 1, preend, preorder, postorder);

return root;

}

public TreeNode constructFromPrePost(int[] preorder, int[] postorder) {

int n = preorder.length;

return solve(0, 0, n - 1, preorder, postorder);

}

}

------------------------------------------------------------------------------------------------------

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode() {}

\* TreeNode(int val) { this.val = val; }

\* TreeNode(int val, TreeNode left, TreeNode right) {

\* this.val = val;

\* this.left = left;

\* this.right = right;

\* }

\* }

\*/

// Approach-2 (using map to optimize)

// T.C : O(n)

// S.C : O(n) using map of size n (you can also include n for System Stack used for Recursion)

import java.util.HashMap;

import java.util.Map;

class Solution {

public TreeNode solve(int prestart, int poststart, int preend, int[] preorder, int[] postorder,

Map<Integer, Integer> mp) {

if (prestart > preend) {

return null;

}

TreeNode root = new TreeNode(preorder[prestart]);

if (prestart == preend) {

return root;

}

int nextNode = preorder[prestart + 1]; // root of left subtree

int j = mp.get(nextNode);

int num = j - poststart + 1;

root.left = solve(prestart + 1, poststart, prestart + num, preorder, postorder, mp);

root.right = solve(prestart + num + 1, j + 1, preend, preorder, postorder, mp);

return root;

}

public TreeNode constructFromPrePost(int[] preorder, int[] postorder) {

int n = preorder.length;

Map<Integer, Integer> mp = new HashMap<>();

for (int i = 0; i < n; i++) {

mp.put(postorder[i], i);

}

return solve(0, 0, n - 1, preorder, postorder, mp);

}

}

[**2467. Most Profitable Path in a Tree**](https://leetcode.com/problems/most-profitable-path-in-a-tree/)

Medium

There is an undirected tree with n nodes labeled from 0 to n - 1, rooted at node 0. You are given a 2D integer array edges of length n - 1 where edges[i] = [ai, bi] indicates that there is an edge between nodes ai and bi in the tree.

At every node i, there is a gate. You are also given an array of even integers amount, where amount[i] represents:

* the price needed to open the gate at node i, if amount[i] is negative, or,
* the cash reward obtained on opening the gate at node i, otherwise.

The game goes on as follows:

* Initially, Alice is at node 0 and Bob is at node bob.
* At every second, Alice and Bob **each** move to an adjacent node. Alice moves towards some **leaf node**, while Bob moves towards node 0.
* For **every** node along their path, Alice and Bob either spend money to open the gate at that node, or accept the reward. Note that:
  + If the gate is **already open**, no price will be required, nor will there be any cash reward.
  + If Alice and Bob reach the node **simultaneously**, they share the price/reward for opening the gate there. In other words, if the price to open the gate is c, then both Alice and Bob pay c / 2 each. Similarly, if the reward at the gate is c, both of them receive c / 2 each.
* If Alice reaches a leaf node, she stops moving. Similarly, if Bob reaches node 0, he stops moving. Note that these events are **independent** of each other.

Return*the****maximum****net income Alice can have if she travels towards the optimal leaf node.*

**Example 1:**



**Input:** edges = [[0,1],[1,2],[1,3],[3,4]], bob = 3, amount = [-2,4,2,-4,6]

**Output:** 6

**Explanation:**

The above diagram represents the given tree. The game goes as follows:

- Alice is initially on node 0, Bob on node 3. They open the gates of their respective nodes.

Alice's net income is now -2.

- Both Alice and Bob move to node 1.

  Since they reach here simultaneously, they open the gate together and share the reward.

  Alice's net income becomes -2 + (4 / 2) = 0.

- Alice moves on to node 3. Since Bob already opened its gate, Alice's income remains unchanged.

  Bob moves on to node 0, and stops moving.

- Alice moves on to node 4 and opens the gate there. Her net income becomes 0 + 6 = 6.

Now, neither Alice nor Bob can make any further moves, and the game ends.

It is not possible for Alice to get a higher net income.

**Example 2:**



**Input:** edges = [[0,1]], bob = 1, amount = [-7280,2350]

**Output:** -7280

**Explanation:**

Alice follows the path 0->1 whereas Bob follows the path 1->0.

Thus, Alice opens the gate at node 0 only. Hence, her net income is -7280.

**Constraints:**

* 2 <= n <= 105
* edges.length == n - 1
* edges[i].length == 2
* 0 <= ai, bi < n
* ai != bi
* edges represents a valid tree.
* 1 <= bob < n
* amount.length == n
* amount[i] is an **even** integer in the range [-104, 104].

Solution:

//Approach-1 (DFS for Bob and DFS for Alice)

//T.C : O(n)

//S.C : O(n)

class Solution {

Map<Integer, List<Integer>> adj = new HashMap<>();

Map<Integer, Integer> bobMap = new HashMap<>();

int aliceIncome;

boolean DFSBob(int curr, int t, boolean[] visited) {

visited[curr] = true;

bobMap.put(curr, t);

if (curr == 0) { // reached target

return true;

}

for (int ngbr : adj.getOrDefault(curr, new ArrayList<>())) {

if (!visited[ngbr]) {

if (DFSBob(ngbr, t + 1, visited)) {

return true;

}

}

}

bobMap.remove(curr);

return false;

}

void DFSAlice(int curr, int t, int income, boolean[] visited, int[] amount) {

visited[curr] = true;

if (!bobMap.containsKey(curr) || t < bobMap.get(curr)) {

income += amount[curr];

} else if (t == bobMap.get(curr)) {

income += (amount[curr] / 2);

}

if (adj.getOrDefault(curr, new ArrayList<>()).size() == 1 && curr != 0) { // leaf node

aliceIncome = Math.max(aliceIncome, income);

}

for (int ngbr : adj.getOrDefault(curr, new ArrayList<>())) {

if (!visited[ngbr]) {

DFSAlice(ngbr, t + 1, income, visited, amount);

}

}

}

public int mostProfitablePath(int[][] edges, int bob, int[] amount) {

int n = amount.length; // n nodes (0 to n-1)

aliceIncome = Integer.MIN\_VALUE;

for (int[] edge : edges) {

int u = edge[0];

int v = edge[1];

adj.computeIfAbsent(u, k -> new ArrayList<>()).add(v);

adj.computeIfAbsent(v, k -> new ArrayList<>()).add(u);

}

// DFS on Bob for finding time to reach each node till 0

int time = 0;

boolean[] visited = new boolean[n];

DFSBob(bob, time, visited);

int income = 0;

Arrays.fill(visited, false);

DFSAlice(0, 0, income, visited, amount);

return aliceIncome;

}

}

[**1524. Number of Sub-arrays With Odd Sum**](https://leetcode.com/problems/number-of-sub-arrays-with-odd-sum/)

Medium

Given an array of integers arr, return *the number of subarrays with an****odd****sum*.

Since the answer can be very large, return it modulo 109 + 7.

**Example 1:**

**Input:** arr = [1,3,5]

**Output:** 4

**Explanation:** All subarrays are [[1],[1,3],[1,3,5],[3],[3,5],[5]]

All sub-arrays sum are [1,4,9,3,8,5].

Odd sums are [1,9,3,5] so the answer is 4.

**Example 2:**

**Input:** arr = [2,4,6]

**Output:** 0

**Explanation:** All subarrays are [[2],[2,4],[2,4,6],[4],[4,6],[6]]

All sub-arrays sum are [2,6,12,4,10,6].

All sub-arrays have even sum and the answer is 0.

**Example 3:**

**Input:** arr = [1,2,3,4,5,6,7]

**Output:** 16

**Constraints:**

* 1 <= arr.length <= 105
* 1 <= arr[i] <= 100

Solution:

class Solution {

    public int numOfSubarrays(int[] arr) {

        int Mod=1000000007;

        int evenC=1;

        int oddC=0;

        int prefix=0;

        int res=0;

        for(int num : arr){

            prefix+=num;

            if(prefix%2==0){

                res += oddC;

                evenC++;

            }else{

                res += evenC;

                oddC++;

            }

            res = res % Mod;

        }

        return res;

    }

}

[**1749. Maximum Absolute Sum of Any Subarray**](https://leetcode.com/problems/maximum-absolute-sum-of-any-subarray/)

Medium

You are given an integer array nums. The **absolute sum** of a subarray [numsl, numsl+1, ..., numsr-1, numsr] is abs(numsl + numsl+1 + ... + numsr-1 + numsr).

Return *the****maximum****absolute sum of any****(possibly empty)****subarray of*nums.

Note that abs(x) is defined as follows:

* If x is a negative integer, then abs(x) = -x.
* If x is a non-negative integer, then abs(x) = x.

**Example 1:**

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

**Output:** 5

**Explanation:** The subarray [2,3] has absolute sum = abs(2+3) = abs(5) = 5.

**Example 2:**

**Input:** nums = [2,-5,1,-4,3,-2]

**Output:** 8

**Explanation:** The subarray [-5,1,-4] has absolute sum = abs(-5+1-4) = abs(-8) = 8.

**Constraints:**

* 1 <= nums.length <= 105
* -104 <= nums[i] <= 104

Solution:

class Solution {

public int maxAbsoluteSum(int[] nums) {

int maxSum = Integer.MIN\_VALUE;

int minSum = Integer.MAX\_VALUE;

int curPSum=0;

int curNSum=0;

for(int num : nums){

// pos

curPSum += num;

maxSum = Math.max(maxSum,curPSum);

if(curPSum < 0){

curPSum = 0;

}

// neg

curNSum += num;

minSum = Math.min(minSum,curNSum);

if(curNSum > 0){

curNSum = 0;

}

}

return Math.max(maxSum, Math.abs(minSum));

}

}

[**873. Length of Longest Fibonacci Subsequence**](https://leetcode.com/problems/length-of-longest-fibonacci-subsequence/)

Medium

A sequence x1, x2, ..., xn is *Fibonacci-like* if:

* n >= 3
* xi + xi+1 == xi+2 for all i + 2 <= n

Given a **strictly increasing** array arr of positive integers forming a sequence, return *the****length****of the longest Fibonacci-like subsequence of* arr. If one does not exist, return 0.

A **subsequence** is derived from another sequence arr by deleting any number of elements (including none) from arr, without changing the order of the remaining elements. For example, [3, 5, 8] is a subsequence of [3, 4, 5, 6, 7, 8].

**Example 1:**

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

**Output:** 5

**Explanation:** The longest subsequence that is fibonacci-like: [1,2,3,5,8].

**Example 2:**

**Input:** arr = [1,3,7,11,12,14,18]

**Output:** 3

**Explanation**:The longest subsequence that is fibonacci-like: [1,11,12], [3,11,14] or [7,11,18].

**Constraints:**

* 3 <= arr.length <= 1000
* 1 <= arr[i] < arr[i + 1] <= 109

Solution:

// brute force

class Solution {

    public int lenLongestFibSubseq(int[] arr) {

        int n = arr.length;

        Set<Integer> numSet = new HashSet<>();

        for (int num : arr) {

            numSet.add(num);

        }

        int maxLen = 0;

        for (int i = 0; i < n; i++) {

            for (int j = i + 1; j < n; j++) {

                int prev = arr[j];

                int curr = arr[i] + arr[j];

                int len = 2;

                while (numSet.contains(curr)) {

                    int temp = curr;

                    curr += prev;

                    prev = temp;

                    maxLen = Math.max(maxLen, ++len);

                }

            }

        }

        return maxLen;

    }

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//DP

class Solution {

public int lenLongestFibSubseq(int[] arr) {

int n = arr.length;

int dp[][] = new int[n][n];

int maxLen=0;

for(int cur=2;cur<n;cur++){

int start = 0;

int end = cur-1;

while(start<end){

int sum = arr[start] + arr[end];

if(sum < arr[cur]){

start++;

}else if(sum > arr[cur]){

end--;

}else{

dp[end][cur] = dp[start][end] + 1;

maxLen = Math.max(maxLen, dp[end][cur]);

start++;

end--;

}

}

}

return (maxLen==0)?0:maxLen + 2;

}

}

[**1092. Shortest Common Supersequence**](https://leetcode.com/problems/shortest-common-supersequence/)

Hard

Given two strings str1 and str2, return *the shortest string that has both*str1*and*str2*as****subsequences***. If there are multiple valid strings, return **any** of them.

A string s is a **subsequence** of string t if deleting some number of characters from t (possibly 0) results in the string s.

**Example 1:**

**Input:** str1 = "abac", str2 = "cab"

**Output:** "cabac"

**Explanation:**

str1 = "abac" is a subsequence of "cabac" because we can delete the first "c".

str2 = "cab" is a subsequence of "cabac" because we can delete the last "ac".

The answer provided is the shortest such string that satisfies these properties.

**Example 2:**

**Input:** str1 = "aaaaaaaa", str2 = "aaaaaaaa"

**Output:** "aaaaaaaa"

**Constraints:**

* 1 <= str1.length, str2.length <= 1000
* str1 and str2 consist of lowercase English letters.

Solution:

class Solution {

    public String shortestCommonSupersequence(String str1, String str2) {

        int dp[][] = longestCommonSubsequence(str1, str2);

        int n = dp.length;

        int m = dp[0].length;

        int i=n-1;

        int j=m-1;

        StringBuilder sb = new StringBuilder();

        while(i>0 && j>0){

            if(str1.charAt(i-1) == str2.charAt(j-1)){ //lcs

                sb.append(str1.charAt(i-1));

                i--;

                j--;

            }else if(dp[i-1][j]>dp[i][j-1]){

                sb.append(str1.charAt(i-1));

                i--;

            }else {

                sb.append(str2.charAt(j-1));

                j--;

            }

        }

        while(i>0) {

            sb.append(str1.charAt(i-1));

            i--;

        }

        while(j>0) {

            sb.append(str2.charAt(j-1));

            j--;

        }

        return sb.reverse().toString();

    }

    public int[][] longestCommonSubsequence(String text1, String text2) {

        int n = text1.length();

        int m = text2.length();

        int dp[][] = new int[n+1][m+1];

        dp[0][0] = 0;

        for(int i=1;i<n+1;i++){

            for(int j=1;j<m+1;j++){

                if(text1.charAt(i-1) == text2.charAt(j-1)){

                    dp[i][j] = 1+ dp[i-1][j-1]; //recur(i-1,j-1,text1,text2,dp);

                    // return dp[i][j];

                }else{

                    int case1 = dp[i-1][j]; //recur(i-1,j,text1,text2,dp);

                    int case2 = dp[i][j-1]; //recur(i,j-1,text1,text2,dp);

                    dp[i][j] = Math.max(case1,case2);

                    // return dp[i][j];

                }

            }

        }

        return dp;//recur(n,m,text1,text2,dp);

    }

}

[**2460. Apply Operations to an Array**](https://leetcode.com/problems/apply-operations-to-an-array/)

Easy

You are given a **0-indexed** array nums of size n consisting of **non-negative** integers.

You need to apply n - 1 operations to this array where, in the ith operation (**0-indexed**), you will apply the following on the ith element of nums:

* If nums[i] == nums[i + 1], then multiply nums[i] by 2 and set nums[i + 1] to 0. Otherwise, you skip this operation.

After performing **all** the operations, **shift** all the 0's to the **end** of the array.

* For example, the array [1,0,2,0,0,1] after shifting all its 0's to the end, is [1,2,1,0,0,0].

Return *the resulting array*.

**Note** that the operations are applied **sequentially**, not all at once.

**Example 1:**

**Input:** nums = [1,2,2,1,1,0]

**Output:** [1,4,2,0,0,0]

**Explanation:** We do the following operations:

- i = 0: nums[0] and nums[1] are not equal, so we skip this operation.

- i = 1: nums[1] and nums[2] are equal, we multiply nums[1] by 2 and change nums[2] to 0. The array becomes [1,**4**,**0**,1,1,0].

- i = 2: nums[2] and nums[3] are not equal, so we skip this operation.

- i = 3: nums[3] and nums[4] are equal, we multiply nums[3] by 2 and change nums[4] to 0. The array becomes [1,4,0,**2**,**0**,0].

- i = 4: nums[4] and nums[5] are equal, we multiply nums[4] by 2 and change nums[5] to 0. The array becomes [1,4,0,2,**0**,**0**].

After that, we shift the 0's to the end, which gives the array [1,4,2,0,0,0].

**Example 2:**

**Input:** nums = [0,1]

**Output:** [1,0]

**Explanation:** No operation can be applied, we just shift the 0 to the end.

**Constraints:**

* 2 <= nums.length <= 2000
* 0 <= nums[i] <= 1000

Solution:

class Solution {

public int[] applyOperations(int[] nums) {

int nz=0;

int n = nums.length;

for(int i=0;i<n;i++){

if(i < n-1 && nums[i]!=0 && nums[i]==nums[i+1]){

nums[i]\*=2;

nums[i+1]=0;

}

if(nums[i]!=0){

if(i!=nz){

int temp = nums[i];

nums[i] = nums[nz];

nums[nz] = temp;

}

nz++;

}

}

return nums;

}

}

[**2570. Merge Two 2D Arrays by Summing Values**](https://leetcode.com/problems/merge-two-2d-arrays-by-summing-values/)

Easy

You are given two **2D** integer arrays nums1 and nums2.

* nums1[i] = [idi, vali] indicate that the number with the id idi has a value equal to vali.
* nums2[i] = [idi, vali] indicate that the number with the id idi has a value equal to vali.

Each array contains **unique** ids and is sorted in **ascending** order by id.

Merge the two arrays into one array that is sorted in ascending order by id, respecting the following conditions:

* Only ids that appear in at least one of the two arrays should be included in the resulting array.
* Each id should be included **only once** and its value should be the sum of the values of this id in the two arrays. If the id does not exist in one of the two arrays, then assume its value in that array to be 0.

Return *the resulting array*. The returned array must be sorted in ascending order by id.

**Example 1:**

**Input:** nums1 = [[1,2],[2,3],[4,5]], nums2 = [[1,4],[3,2],[4,1]]

**Output:** [[1,6],[2,3],[3,2],[4,6]]

**Explanation:** The resulting array contains the following:

- id = 1, the value of this id is 2 + 4 = 6.

- id = 2, the value of this id is 3.

- id = 3, the value of this id is 2.

- id = 4, the value of this id is 5 + 1 = 6.

**Example 2:**

**Input:** nums1 = [[2,4],[3,6],[5,5]], nums2 = [[1,3],[4,3]]

**Output:** [[1,3],[2,4],[3,6],[4,3],[5,5]]

**Explanation:** There are no common ids, so we just include each id with its value in the resulting list.

**Constraints:**

* 1 <= nums1.length, nums2.length <= 200
* nums1[i].length == nums2[j].length == 2
* 1 <= idi, vali <= 1000
* Both arrays contain unique ids.
* Both arrays are in strictly ascending order by id.

Solution:

class Solution {

    public int[][] mergeArrays(int[][] nums1, int[][] nums2) {

        int m = nums1.length;

        int n = nums2.length;

        Map<Integer, Integer> map = new HashMap<>();

        List<int[]> result = new ArrayList<>();

        for (int i = 0; i < m; i++) {

            int id = nums1[i][0];

            int val = nums1[i][1];

            map.put(id, map.getOrDefault(id, 0) + val);

        }

        for (int i = 0; i < n; i++) {

            int id = nums2[i][0];

            int val = nums2[i][1];

            map.put(id, map.getOrDefault(id, 0) + val);

        }

        for (Map.Entry<Integer, Integer> entry : map.entrySet()) {

            result.add(new int[]{entry.getKey(), entry.getValue()});

        }

        result.sort(Comparator.comparingInt(a -> a[0]));

        return result.toArray(new int[0][]);

    }

}

------------------------------------------------------------------------------------------------------

Second Approch:

class Solution {

    public int[][] mergeArrays(int[][] nums1, int[][] nums2) {

        int m = nums1.length;

        int n = nums2.length;

        int i = 0, j = 0;

        List<int[]> result = new ArrayList<>();

        while (i < m && j < n) {

            if (nums1[i][0] < nums2[j][0]) {

                result.add(nums1[i]);

              i++;

            } else if (nums2[j][0] < nums1[i][0]) {

                result.add(nums2[j]);

                j++;

            } else {

                result.add(new int[]{nums1[i][0], nums1[i][1] + nums2[j][1]});

                i++;

                j++;

            }

        }

        while (i < m) {

            result.add(nums1[i]);

            i++;

        }

        while (j < n) {

            result.add(nums2[j]);

            j++;

        }

        return result.toArray(new int[0][]);

    }

}

[**2161. Partition Array According to Given Pivot**](https://leetcode.com/problems/partition-array-according-to-given-pivot/)

Medium

You are given a **0-indexed** integer array nums and an integer pivot. Rearrange nums such that the following conditions are satisfied:

* Every element less than pivot appears **before** every element greater than pivot.
* Every element equal to pivot appears **in between** the elements less than and greater than pivot.
* The **relative order** of the elements less than pivot and the elements greater than pivot is maintained.
  + More formally, consider every pi, pj where pi is the new position of the ith element and pj is the new position of the jth element. If i < j and **both** elements are smaller (*or larger*) than pivot, then pi < pj.

Return nums*after the rearrangement.*

**Example 1:**

**Input:** nums = [9,12,5,10,14,3,10], pivot = 10

**Output:** [9,5,3,10,10,12,14]

**Explanation:**

The elements 9, 5, and 3 are less than the pivot so they are on the left side of the array.

The elements 12 and 14 are greater than the pivot so they are on the right side of the array.

The relative ordering of the elements less than and greater than pivot is also maintained. [9, 5, 3] and [12, 14] are the respective orderings.

**Example 2:**

**Input:** nums = [-3,4,3,2], pivot = 2

**Output:** [-3,2,4,3]

**Explanation:**

The element -3 is less than the pivot so it is on the left side of the array.

The elements 4 and 3 are greater than the pivot so they are on the right side of the array.

The relative ordering of the elements less than and greater than pivot is also maintained. [-3] and [4, 3] are the respective orderings.

**Constraints:**

* 1 <= nums.length <= 105
* -106 <= nums[i] <= 106
* pivot equals to an element of nums.

Solution:

class Solution {

    public int[] pivotArray(int[] nums, int pivot) {

        int lCount=0;

        int gCount=0;

        int pCount=0;

        for(int num : nums){

            if(num < pivot){

                lCount++;

            }else if(num > pivot){

                gCount++;

            }else{

                pCount++;

            }

        }

        int i=0;

        int j=lCount;

        int k=lCount + pCount;

        int res[] = new int[nums.length];

        for(int num : nums){

            if(num < pivot){

                res[i] = num;

                i++;

            }else if(num > pivot){

                res[k] = num;

                k++;

            }else{

                res[j] = num;

                j++;

            }

        }

        return res;

    }

}

[**1780. Check if Number is a Sum of Powers of Three**](https://leetcode.com/problems/check-if-number-is-a-sum-of-powers-of-three/)

Medium

Given an integer n, return true *if it is possible to represent*n*as the sum of distinct powers of three.* Otherwise, return false.

An integer y is a power of three if there exists an integer x such that y == 3x.

**Example 1:**

**Input:** n = 12

**Output:** true

**Explanation:** 12 = 31 + 32

**Example 2:**

**Input:** n = 91

**Output:** true

**Explanation:** 91 = 30 + 32 + 34

**Example 3:**

**Input:** n = 21

**Output:** false

**Constraints:**

* 1 <= n <= 107

Solution:

class Solution {

public boolean checkPowersOfThree(int n) {

while(n>0){

if((n%3) == 2){

return false;

}

n/=3;

}

return true;

}

}

[**2965. Find Missing and Repeated Values**](https://leetcode.com/problems/find-missing-and-repeated-values/)

Easy

You are given a **0-indexed** 2D integer matrix grid of size n \* n with values in the range [1, n2]. Each integer appears **exactly once** except a which appears **twice** and b which is **missing**. The task is to find the repeating and missing numbers a and b.

Return *a****0-indexed****integer array*ans*of size*2*where*ans[0]*equals to*a*and*ans[1]*equals to*b*.*

**Example 1:**

**Input:** grid = [[1,3],[2,2]]

**Output:** [2,4]

**Explanation:** Number 2 is repeated and number 4 is missing so the answer is [2,4].

**Example 2:**

**Input:** grid = [[9,1,7],[8,9,2],[3,4,6]]

**Output:** [9,5]

**Explanation:** Number 9 is repeated and number 5 is missing so the answer is [9,5].

**Constraints:**

* 2 <= n == grid.length == grid[i].length <= 50
* 1 <= grid[i][j] <= n \* n
* For all x that 1 <= x <= n \* n there is exactly one x that is not equal to any of the grid members.
* For all x that 1 <= x <= n \* n there is exactly one x that is equal to exactly two of the grid members.
* For all x that 1 <= x <= n \* n except two of them there is exatly one pair of i, j that 0 <= i, j <= n - 1 and grid[i][j] == x.

Solution:

class Solution {

    public int[] findMissingAndRepeatedValues(int[][] grid) {

        int n = grid.length;

        int sq = n \* n;

        int set[] = new int[sq + 1];

        int curSum = 0;

        int ans[] = new int[2];

        for (int i = 0; i < n; i++) {

            for (int j = 0; j < n; j++) {

                if (set[grid[i][j]] != 0) {

                    ans[0] = grid[i][j];

                } else {

                    set[grid[i][j]] = 1;

                    curSum += grid[i][j];

                }

            }

        }

        int totalSum = sq \* (sq + 1) / 2;

        ans[1] = totalSum - curSum;

        return ans;

    }

}

[**2523. Closest Prime Numbers in Range**](https://leetcode.com/problems/closest-prime-numbers-in-range/)

Medium

Given two positive integers left and right, find the two integers num1 and num2 such that:

* left <= num1 < num2 <= right .
* Both num1 and num2 are prime numbers.
* num2 - num1 is the **minimum** amongst all other pairs satisfying the above conditions.

Return the positive integer array ans = [num1, num2]. If there are multiple pairs satisfying these conditions, return the one with the **smallest** num1 value. If no such numbers exist, return [-1, -1]*.*

**Example 1:**

**Input:** left = 10, right = 19

**Output:** [11,13]

**Explanation:** The prime numbers between 10 and 19 are 11, 13, 17, and 19.

The closest gap between any pair is 2, which can be achieved by [11,13] or [17,19].

Since 11 is smaller than 17, we return the first pair.

**Example 2:**

**Input:** left = 4, right = 6

**Output:** [-1,-1]

**Explanation:** There exists only one prime number in the given range, so the conditions cannot be satisfied.

**Constraints:**

* 1 <= left <= right <= 106

Solution:

class Solution {

public int[] closestPrimes(int left, int right) {

// O(R log(logR) + R-L)

// Sieve algorithm to find prime numbers between [1,right]

boolean prime[] = new boolean[right + 1];

Arrays.fill(prime,true);

prime[0] = false;

prime[1] = false;

// Sieve-> O(R log(logR))

for (int p = 2; p \* p <= right; p++) {

if (prime[p]) {

for (int i = p \* p; i <= right; i += p)

prime[i] = false;

}

}

// R-L

// find min diff b/w pair of prime numbers

int res[] = new int[]{-1,-1};

int minDiff = Integer.MAX\_VALUE;

int prev=-1;

for (int i = left; i <= right; i++) {

if (prime[i]){

if(prev == -1){

prev = i;

}else{

if(i - prev < minDiff){

res[0] = prev;

res[1] = i;

minDiff = i-prev;

}

prev=i;

}

}

}

return res;

}

}

[**2379. Minimum Recolors to Get K Consecutive Black Blocks**](https://leetcode.com/problems/minimum-recolors-to-get-k-consecutive-black-blocks/)

Easy

You are given a **0-indexed** string blocks of length n, where blocks[i] is either 'W' or 'B', representing the color of the ith block. The characters 'W' and 'B' denote the colors white and black, respectively.

You are also given an integer k, which is the desired number of **consecutive** black blocks.

In one operation, you can **recolor** a white block such that it becomes a black block.

Return*the****minimum****number of operations needed such that there is at least****one****occurrence of*k*consecutive black blocks.*

**Example 1:**

**Input:** blocks = "WBBWWBBWBW", k = 7

**Output:** 3

**Explanation:**

One way to achieve 7 consecutive black blocks is to recolor the 0th, 3rd, and 4th blocks

so that blocks = "BBBBBBBWBW".

It can be shown that there is no way to achieve 7 consecutive black blocks in less than 3 operations.

Therefore, we return 3.

**Example 2:**

**Input:** blocks = "WBWBBBW", k = 2

**Output:** 0

**Explanation:**

No changes need to be made, since 2 consecutive black blocks already exist.

Therefore, we return 0.

**Constraints:**

* n == blocks.length
* 1 <= n <= 100
* blocks[i] is either 'W' or 'B'.
* 1 <= k <= n

Solution:

class Solution {

public int minimumRecolors(String blocks, int k) {

//first window

int w=0;

for(int i=0;i<k;i++){

if(blocks.charAt(i) == 'W') w++;

}

int res = w;

int n = blocks.length();

//next window

for(int i=1;i<(n - k + 1); i++){

if(blocks.charAt(i-1) == 'W') w--;

if(blocks.charAt(i + k -1) == 'W') w++;

res = Math.min(res,w);

}

return res;

// (N)

}

}

[**3208. Alternating Groups II**](https://leetcode.com/problems/alternating-groups-ii/)

Medium

There is a circle of red and blue tiles. You are given an array of integers colors and an integer k. The color of tile i is represented by colors[i]:

* colors[i] == 0 means that tile i is **red**.
* colors[i] == 1 means that tile i is **blue**.

An **alternating** group is every k contiguous tiles in the circle with **alternating** colors (each tile in the group except the first and last one has a different color from its **left** and **right** tiles).

Return the number of **alternating** groups.

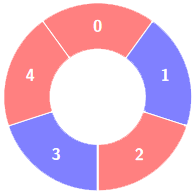
**Note** that since colors represents a **circle**, the **first** and the **last** tiles are considered to be next to each other.

**Example 1:**

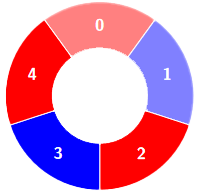
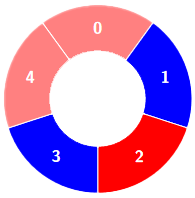
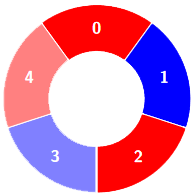
**Input:** colors = [0,1,0,1,0], k = 3

**Output:** 3

**Explanation:**

****

Alternating groups:

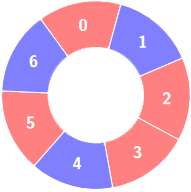


**Example 2:**

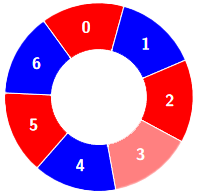
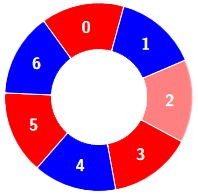
**Input:** colors = [0,1,0,0,1,0,1], k = 6

**Output:** 2

**Explanation:**

****

Alternating groups:

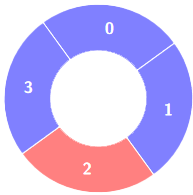


**Example 3:**

**Input:** colors = [1,1,0,1], k = 4

**Output:** 0

**Explanation:**



**Constraints:**

* 3 <= colors.length <= 105
* 0 <= colors[i] <= 1
* 3 <= k <= colors.length

Solution:

class Solution {

    public int numberOfAlternatingGroups(int[] colors, int k) {

        int res = 0;

        int left = 0;

        int n = colors.length;

        for(int right=1; right < (n + k - 1); right++) {

           if(colors[right % n] == colors[(right-1)%n]) {

            left = right;

           }

           if(right - left + 1 == k) {

            res++;

            left++;

           }

        }

        return res;

    }

}

[**3306. Count of Substrings Containing Every Vowel and K Consonants II**](https://leetcode.com/problems/count-of-substrings-containing-every-vowel-and-k-consonants-ii/)

Medium

You are given a string word and a **non-negative** integer k.

Return the total number of substrings of word that contain every vowel ('a', 'e', 'i', 'o', and 'u') **at least** once and **exactly** k consonants.

**Example 1:**

**Input:** word = "aeioqq", k = 1

**Output:** 0

**Explanation:**

There is no substring with every vowel.

**Example 2:**

**Input:** word = "aeiou", k = 0

**Output:** 1

**Explanation:**

The only substring with every vowel and zero consonants is word[0..4], which is "aeiou".

**Example 3:**

**Input:** word = "ieaouqqieaouqq", k = 1

**Output:** 3

**Explanation:**

The substrings with every vowel and one consonant are:

* word[0..5], which is "ieaouq".
* word[6..11], which is "qieaou".
* word[7..12], which is "ieaouq".

**Constraints:**

* 5 <= word.length <= 2 \* 105
* word consists only of lowercase English letters.
* 0 <= k <= word.length - 5

Solution:

class Solution {

    public long countOfSubstrings(String word, int k) {

        return atleast(k,word) - atleast(k+1,word);

    }

    private boolean isConsonant(char ch){

        return (ch!='a' && ch!='e' && ch!='i' && ch!='o' && ch!='u');

    }

    private boolean isAllVowelsPresent(int freq[]){

        return (freq['a'-'a']>0 && freq['e'-'a']>0 && freq['i'-'a']>0

         && freq['o'-'a']>0 && freq['u'-'a']>0);

    }

    private long atleast(int k, String words){

        long count=0;

        int curConsonant=0;

        int freq[] = new int[26];

        int n = words.length();

        int left=0;

        for(int right=0;right<n;right++){

            char ch = words.charAt(right);

            if(isConsonant(ch)){

                curConsonant++;

            }

            freq[ch-'a']++;

            while(curConsonant>=k && isAllVowelsPresent(freq)){

                count += (n - right);

                char c = words.charAt(left);

                if(isConsonant(c)){

                    curConsonant--;

                }

                freq[c-'a']--;

                left++;

            }

        }

        return count;

    }

}

[**1358. Number of Substrings Containing All Three Characters**](https://leetcode.com/problems/number-of-substrings-containing-all-three-characters/)

Medium

Given a string s consisting only of characters *a*, *b* and *c*.

Return the number of substrings containing **at least** one occurrence of all these characters *a*, *b* and *c*.

**Example 1:**

**Input:** s = "abcabc"

**Output:** 10

**Explanation:** The substrings containing at least one occurrence of the characters *a*, *b* and *c are "*abc*", "*abca*", "*abcab*", "*abcabc*", "*bca*", "*bcab*", "*bcabc*", "*cab*", "*cabc*"* and *"*abc*"* (**again**)*.*

**Example 2:**

**Input:** s = "aaacb"

**Output:** 3

**Explanation:** The substrings containing at least one occurrence of the characters *a*, *b* and *c are "*aaacb*", "*aacb*"* and *"*acb*".*

**Example 3:**

**Input:** s = "abc"

**Output:** 1

**Constraints:**

* 3 <= s.length <= 5 x 10^4
* s only consists of *a*, *b* or *c*characters.

Solution:

class Solution {

    public int numberOfSubstrings(String s) {

        int freq[] = new int[3];

        int count=0;

        int curCount=0;

        int left=0;

        int n = s.length();

        for(int right=0;right<n;right++){

            char ch = s.charAt(right);

            freq[ch - 'a']++;

            if(freq[ch-'a'] == 1){

                curCount++;

            }

            while(curCount==3){

                count += (n - right);

                char c = s.charAt(left);

                freq[c - 'a']--;

                if(freq[c-'a'] == 0){

                    curCount--;

                }

                left++;

            }

        }

        return count;

    }

}

[**2529. Maximum Count of Positive Integer and Negative Integer**](https://leetcode.com/problems/maximum-count-of-positive-integer-and-negative-integer/)

Easy

Given an array nums sorted in **non-decreasing** order, return *the maximum between the number of positive integers and the number of negative integers.*

* In other words, if the number of positive integers in nums is pos and the number of negative integers is neg, then return the maximum of pos and neg.

**Note** that 0 is neither positive nor negative.

**Example 1:**

**Input:** nums = [-2,-1,-1,1,2,3]

**Output:** 3

**Explanation:** There are 3 positive integers and 3 negative integers. The maximum count among them is 3.

**Example 2:**

**Input:** nums = [-3,-2,-1,0,0,1,2]

**Output:** 3

**Explanation:** There are 2 positive integers and 3 negative integers. The maximum count among them is 3.

**Example 3:**

**Input:** nums = [5,20,66,1314]

**Output:** 4

**Explanation:** There are 4 positive integers and 0 negative integers. The maximum count among them is 4.

**Constraints:**

* 1 <= nums.length <= 2000
* -2000 <= nums[i] <= 2000
* nums is sorted in a **non-decreasing order**.

Solution:

class Solution {

    public int maximumCount(int[] nums) {

        //first >-1 -> neg

        // first >0 -> pos

        int n = nums.length;

        int end = n-1;

        int negPos = binarySearch(nums, 0, end, -1); //log(n)

        int posPos = binarySearch(nums, negPos, end, 0); //log(n - negPos)

        int posCount = (n - posPos);

        return Math.max(negPos,posCount);

    }

    public int binarySearch(int nums[], int start, int end, int target){

        while(start<=end){

            int mid = start + (end-start)/2;

            if(nums[mid]<=target){

                start = mid+1;

            }else{

                end=mid-1;

            }

        }

        return start;

    }

}

[**3356. Zero Array Transformation II**](https://leetcode.com/problems/zero-array-transformation-ii/)

Medium

You are given an integer array nums of length n and a 2D array queries where queries[i] = [li, ri, vali].

Each queries[i] represents the following action on nums:

* Decrement the value at each index in the range [li, ri] in nums by **at most** vali.
* The amount by which each value is decremented can be chosen **independently** for each index.

A **Zero Array** is an array with all its elements equal to 0.

Return the **minimum** possible **non-negative** value of k, such that after processing the first k queries in **sequence**, nums becomes a **Zero Array**. If no such k exists, return -1.

**Example 1:**

**Input:** nums = [2,0,2], queries = [[0,2,1],[0,2,1],[1,1,3]]

**Output:** 2

**Explanation:**

* **For i = 0 (l = 0, r = 2, val = 1):**
  + Decrement values at indices [0, 1, 2] by [1, 0, 1] respectively.
  + The array will become [1, 0, 1].
* **For i = 1 (l = 0, r = 2, val = 1):**
  + Decrement values at indices [0, 1, 2] by [1, 0, 1] respectively.
  + The array will become [0, 0, 0], which is a Zero Array. Therefore, the minimum value of k is 2.

**Example 2:**

**Input:** nums = [4,3,2,1], queries = [[1,3,2],[0,2,1]]

**Output:** -1

**Explanation:**

* **For i = 0 (l = 1, r = 3, val = 2):**
  + Decrement values at indices [1, 2, 3] by [2, 2, 1] respectively.
  + The array will become [4, 1, 0, 0].
* **For i = 1 (l = 0, r = 2, val = 1):**
  + Decrement values at indices [0, 1, 2] by [1, 1, 0] respectively.
  + The array will become [3, 0, 0, 0], which is not a Zero Array.

**Constraints:**

* 1 <= nums.length <= 105
* 0 <= nums[i] <= 5 \* 105
* 1 <= queries.length <= 105
* queries[i].length == 3
* 0 <= li <= ri < nums.length
* 1 <= vali <= 5

Solution:

class Solution {

  public int minZeroArray(int[] nums, int[][] queries) {

    int[] line = new int[nums.length + 1];

    int decrement = 0;

    int k = 0;

    for (int i = 0; i < nums.length; ++i) {

      while (decrement + line[i] < nums[i]) {

        if (k == queries.length)

          return -1;

        final int l = queries[k][0];

        final int r = queries[k][1];

        final int val = queries[k][2];

        ++k;

        if (r < i)

          continue;

        line[Math.max(l, i)] += val;

        line[r + 1] -= val;

      }

      decrement += line[i];

    }

    return k;

  }

}

Code

Testcase

Test Result

Test Result

[**2226. Maximum Candies Allocated to K Children**](https://leetcode.com/problems/maximum-candies-allocated-to-k-children/)

Medium

You are given a **0-indexed** integer array candies. Each element in the array denotes a pile of candies of size candies[i]. You can divide each pile into any number of **sub piles**, but you **cannot** merge two piles together.

You are also given an integer k. You should allocate piles of candies to k children such that each child gets the **same** number of candies. Each child can be allocated candies from **only one** pile of candies and some piles of candies may go unused.

Return *the****maximum number of candies****each child can get.*

**Example 1:**

**Input:** candies = [5,8,6], k = 3

**Output:** 5

**Explanation:** We can divide candies[1] into 2 piles of size 5 and 3, and candies[2] into 2 piles of size 5 and 1. We now have five piles of candies of sizes 5, 5, 3, 5, and 1. We can allocate the 3 piles of size 5 to 3 children. It can be proven that each child cannot receive more than 5 candies.

**Example 2:**

**Input:** candies = [2,5], k = 11

**Output:** 0

**Explanation:** There are 11 children but only 7 candies in total, so it is impossible to ensure each child receives at least one candy. Thus, each child gets no candy and the answer is 0.

**Constraints:**

* 1 <= candies.length <= 105
* 1 <= candies[i] <= 107
* 1 <= k <= 1012

Solution:

class Solution {

    private boolean canDistr(int[] candies, int mid, long k) {

        int n = candies.length;

        for (int i = 0; i < n; i++) {

            k -= candies[i] / mid;

            if (k <= 0) { // all children got mid candies

                return true; // Early return

            }

        }

        return k <= 0; // all children got the mid candies

    }

    public int maximumCandies(int[] candies, long k) {

        int n = candies.length;

        int maxC = 0;

        long total = 0;

        for (int i = 0; i < n; i++) {

            total += candies[i];

            maxC = Math.max(maxC, candies[i]);

        }

        if (total < k) {

            return 0;

        }

        int l = 1;

        int r = maxC;

        int result = 0;

        while (l <= r) {

            int mid = l + (r - l) / 2;

            if (canDistr(candies, mid, k)) {

                result = mid;

                l = mid + 1;

            } else {

                r = mid - 1;

            }

        }

        return result;

    }

}

[**2560. House Robber IV**](https://leetcode.com/problems/house-robber-iv/)

Medium

There are several consecutive houses along a street, each of which has some money inside. There is also a robber, who wants to steal money from the homes, but he **refuses to steal from adjacent homes**.

The **capability** of the robber is the maximum amount of money he steals from one house of all the houses he robbed.

You are given an integer array nums representing how much money is stashed in each house. More formally, the ith house from the left has nums[i] dollars.

You are also given an integer k, representing the **minimum** number of houses the robber will steal from. It is always possible to steal at least k houses.

Return *the****minimum****capability of the robber out of all the possible ways to steal at least*k*houses*.

**Example 1:**

**Input:** nums = [2,3,5,9], k = 2

**Output:** 5

**Explanation:**

There are three ways to rob at least 2 houses:

- Rob the houses at indices 0 and 2. Capability is max(nums[0], nums[2]) = 5.

- Rob the houses at indices 0 and 3. Capability is max(nums[0], nums[3]) = 9.

- Rob the houses at indices 1 and 3. Capability is max(nums[1], nums[3]) = 9.

Therefore, we return min(5, 9, 9) = 5.

**Example 2:**

**Input:** nums = [2,7,9,3,1], k = 2

**Output:** 2

**Explanation:** There are 7 ways to rob the houses. The way which leads to minimum capability is to rob the house at index 0 and 4. Return max(nums[0], nums[4]) = 2.

**Constraints:**

* 1 <= nums.length <= 105
* 1 <= nums[i] <= 109
* 1 <= k <= (nums.length + 1)/2

Solution:

class Solution {

    public int minCapability(int[] nums, int k) {

        int start=Integer.MAX\_VALUE;

        int end=Integer.MIN\_VALUE;

        for(int num : nums){

            if(num>end){

                end = num;

            }

            if(num < start){

                start = num;

            }

        }

        int ans=0;

        while(start<=end){

            int mid = start + (end-start)/2;

            if(isRobberyPossible(mid, nums, k)){

                ans = mid;

                end = mid-1;

            }else{

                start = mid+1;

            }

        }

        return ans;

    }

    public boolean isRobberyPossible(int capability, int nums[], int minHouses){

        int housesRobbed=0;

        for(int i=0;i<nums.length;i++){

            if(nums[i] <= capability){

                housesRobbed++;

                i++; //skip for adjacent condition

            }

            if(housesRobbed >= minHouses) return true;

        }

        return false;

    }

}