**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;

}

}