<https://leetcode.com/problems/find-good-days-to-rob-the-bank/description/>

You and a gang of thieves are planning on robbing a bank. You are given a **0-indexed** integer array security, where security[i] is the number of guards on duty on the ith day. The days are numbered starting from 0. You are also given an integer time.

The ith day is a good day to rob the bank if:

There are at least time days before and after the ith day,

The number of guards at the bank for the time days **before** i are **non-increasing**, and

The number of guards at the bank for the time days **after** i are **non-decreasing**.

More formally, this means day i is a good day to rob the bank if and only if security[i - time] >= security[i - time + 1] >= ... >= security[i] <= ... <= security[i + time - 1] <= security[i + time].

Return a list of **all** days (**0-indexed**) that are good days to rob the bank. The order that the days are returned in does not matter.

**Example 1:**

**Input:** security = [5,3,3,3,5,6,2], time = 2

**Output:** [2,3]

**Explanation:**

On day 2, we have security[0] >= security[1] >= security[2] <= security[3] <= security[4].

On day 3, we have security[1] >= security[2] >= security[3] <= security[4] <= security[5].

No other days satisfy this condition, so days 2 and 3 are the only good days to rob the bank.

**Example 2:**

**Input:** security = [1,1,1,1,1], time = 0

**Output:** [0,1,2,3,4]

**Explanation:**

Since time equals 0, every day is a good day to rob the bank, so return every day.

**Example 3:**

**Input:** security = [1,2,3,4,5,6], time = 2

**Output:** []

**Explanation:**

No day has 2 days before it that have a non-increasing number of guards.

Thus, no day is a good day to rob the bank, so return an empty list.

**Constraints:**

1 <= security.length <= 10^5

0 <= security[i], time <= 10^5

**Attempt 1: 2024-08-27**

**Solution 1: Prefix Sum + LIS (30 min)**

class Solution {

    public List<Integer> goodDaysToRobBank(int[] security, int time) {

        int n = security.length;

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

        if(time == 0) {

            // If time is 0, every day is a good day to rob the bank

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

                result.add(i);

            }

            return result;

        }

        int[] nonIncreasingDays = new int[n];

        int[] nonDecreasingDays = new int[n];

        // For non-increasing

        for(int i = 1; i < n; i++) {

            if(security[i] <= security[i - 1]) {

                nonIncreasingDays[i] = nonIncreasingDays[i - 1] + 1;

            }

            // No need reset nonIncreasingDays[i] to 0 since

            // default as 0 when security[i] < security[i - 1]

        }

        // For non-decreasing

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

            if(security[i] <= security[i + 1]) {

                nonDecreasingDays[i] = nonDecreasingDays[i + 1] + 1;

            }

            // No need reset nonDecreasingDays[i] to 0 since

            // default as 0 when security[i] > security[i + 1]

        }

        // Find good days to rob the bank

        for(int i = 0; i < n; i++) {

            if(nonIncreasingDays[i] >= time && nonDecreasingDays[i] >= time) {

                result.add(i);

            }

        }

        return result;

    }

}

Time Complexity: O(N)

Space Complexity: O(N)

**Refer to chatGPT**

LeetCode problem 2100 is titled "Find Good Days to Rob the Bank." In this problem, you are given an array security where security[i] represents the security level at day i. You also have an integer time. A day is considered a good day to rob the bank if:

For the past time days, the security level has been non-increasing.

For the next time days, the security level has been non-decreasing.

You need to find all good days to rob the bank.

**Java Solution**

To solve this problem, we can use a two-pass approach:

Left Pass: Count the number of consecutive non-increasing days up to each day.

Right Pass: Count the number of consecutive non-decreasing days starting from each day.

Identify the days that meet both conditions.

Here's how you can implement this solution in Java:

class Solution {

public List<Integer> goodDaysToRobBank(int[] security, int time) {

int n = security.length;

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

if (time == 0) {

// If time is 0, every day is a good day to rob the bank

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

result.add(i);

}

return result;

}

// Arrays to store non-increasing and non-decreasing counts

int[] nonIncreasing = new int[n];

int[] nonDecreasing = new int[n];

// Fill non-increasing count

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

if (security[i] <= security[i - 1]) {

nonIncreasing[i] = nonIncreasing[i - 1] + 1;

}

}

// Fill non-decreasing count

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

if (security[i] <= security[i + 1]) {

nonDecreasing[i] = nonDecreasing[i + 1] + 1;

}

}

// Find good days to rob the bank

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

if (nonIncreasing[i] >= time && nonDecreasing[i] >= time) {

result.add(i);

}

}

return result;

}

}

**Explanation**

Initialization:

We check if time is 0. If it is, every day is a good day to rob the bank, so we return all days.

We create two arrays, nonIncreasing and nonDecreasing, to store the count of consecutive non-increasing and non-decreasing days, respectively.

Left Pass:

We fill the nonIncreasing array by checking if security[i] <= security[i-1]. If true, we increment the count for nonIncreasing[i].

Right Pass:

We fill the nonDecreasing array by checking if security[i] <= security[i+1]. If true, we increment the count for nonDecreasing[i].

Finding Good Days:

We iterate through the days starting from time to n-time-1. For each day, we check if both nonIncreasing[i] and nonDecreasing[i] are greater than or equal to time. If both conditions are met, we add the day to the result.

**Example**

For the given input:

int[] security = {5, 3, 3, 3, 5, 6, 2};

int time = 2;

The output will be [2, 3] because:

On day 2, the security levels for the past 2 days are non-increasing [5, 3, 3] and for the next 2 days are non-decreasing [3, 3, 5].

On day 3, the security levels for the past 2 days are non-increasing [3, 3, 3] and for the next 2 days are non-decreasing [3, 5, 6].

**Time Complexity**

O(n), where n is the number of days. We make two passes over the array to fill the nonIncreasing and nonDecreasing arrays and one more pass to find the good days, resulting in a linear runtime.

This solution efficiently finds the good days to rob the bank while adhering to the problem constraints.

**Question 1: Why the for loop direction on calculating 'nonIncreasingDays' must from right to left and NOT from left to right as normal ?**

**The wrong solution below if we change to left to right as below:**

**security = [5,3,3,3,5,6,2], time = 2**

**Output = [3]**

**Expected = [2,3]**

class Solution {

    public List<Integer> goodDaysToRobBank(int[] security, int time) {

        int n = security.length;

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

        if(time == 0) {

            // If time is 0, every day is a good day to rob the bank

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

                result.add(i);

            }

            return result;

        }

        int[] nonIncreasingDays = new int[n];

        int[] nonDecreasingDays = new int[n];

        // For non-increasing

        for(int i = 1; i < n; i++) {

            if(security[i] <= security[i - 1]) {

                nonIncreasingDays[i] = nonIncreasingDays[i - 1] + 1;

            }

            // No need reset nonIncreasingDays[i] to 0 since

            // default as 0 when security[i] < security[i - 1]

        }

        // For non-decreasing

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

        //    if(security[i] <= security[i + 1]) {

        //        nonDecreasingDays[i] = nonDecreasingDays[i + 1] + 1;

        //    }

        //    // No need reset nonDecreasingDays[i] to 0 since

        //    // default as 0 when security[i] > security[i + 1]

        //}

// Test: change loop direction as left to right and error out

        for(int i = 1; i < n; i++) {

            if(security[i] >= security[i - 1]) {

                nonDecreasingDays[i] = nonDecreasingDays[i - 1] + 1;

            }

        }

        // Find good days to rob the bank

        for(int i = 0; i < n; i++) {

            if(nonIncreasingDays[i] >= time && nonDecreasingDays[i] >= time) {

                result.add(i);

            }

        }

        return result;

    }

}

**The Issue**

In the original approach, the loop for the nonDecreasingDays array went from right to left, and it worked as intended because **it correctly counted the number of consecutive non-decreasing days starting from each day (calculate non-decreasing future days). However, when you changed the loop to run from left to right, it started counting the number of non-decreasing days ending at each day instead of starting from each day.** This change affects how the conditions for a "good day" are evaluated.

**Problem with the Modified Solution**

In your modified solution, the nonDecreasingDays[i] value is calculated based on the previous day's value (nonDecreasingDays[i - 1]), which makes it accumulate the number of non-decreasing days up to day i. When you use this logic to determine if a day is a good day to rob the bank, it doesn't correctly capture the requirement of having a certain number of future non-decreasing days.

**Why the Output is Wrong**

For the input security = [5,3,3,3,5,6,2] and time = 2, the expected output is [2, 3]. These are the days where:

There are at least 2 consecutive non-increasing days before (i.e., days 0 and 1 are non-increasing before day 2).

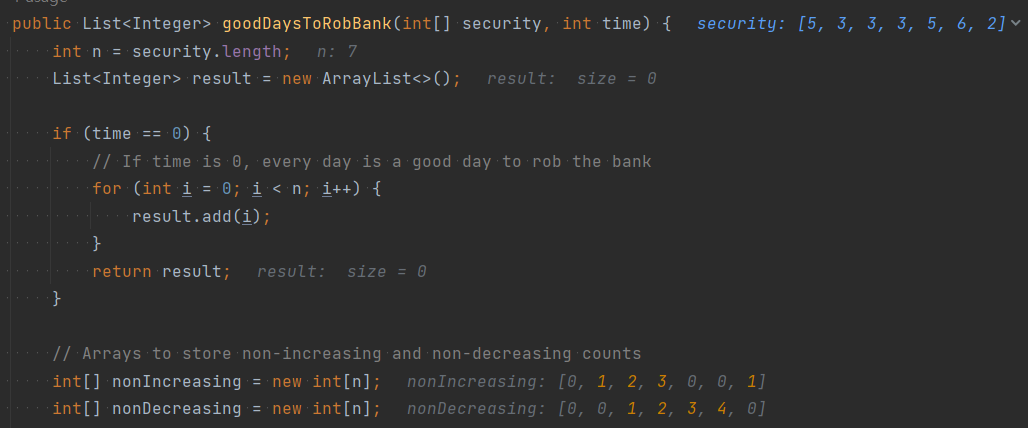
There are at least 2 consecutive non-decreasing days after (i.e., days 4 and 5 are non-decreasing after day 2).

In the modified solution:

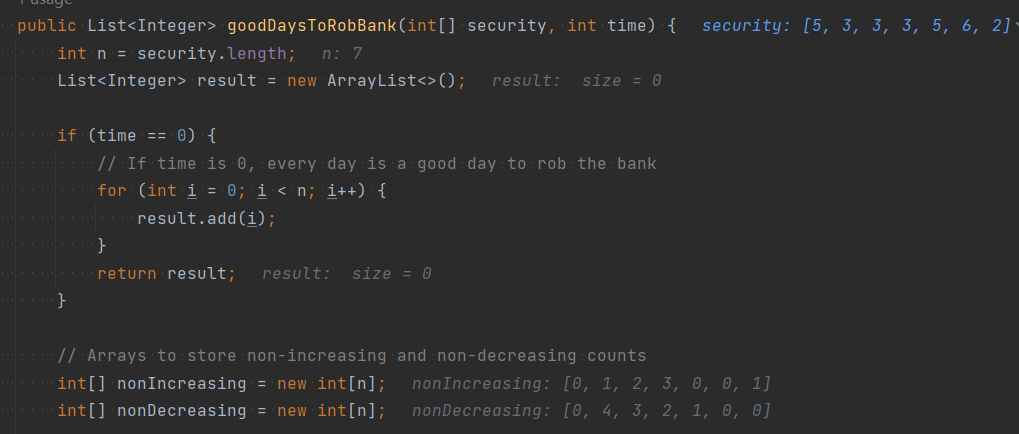
Day 2 (i = 2): **nonDecreasingDays[i] is incorrectly evaluated. The count considers only past non-decreasing days, not future days.**

This causes the nonDecreasingDays value at day 2 to be insufficient to meet the criteria, so day 2 is not marked as a "good day" in your result.

**For example, in the wrong solution if given input security = [5,3,3,3,5,6,2] and time = 2, the nonDecreasingDays wrongly calculated as [0, 0, 1, 2, 3, 4, 0], which represents the non-decreasing days for each day start from left to right, which physically means past days for that day, instead of future days for that day.**



**The correct result for nonDecreasing array should be [0, 4, 3, 2, 1, 0, 0]**



**Question 2: When calculating LIS, nested for loop NOT required for Leetcode 2100 but rqeuired in Leetcode 1671, 300, ... etc ?**

**There is a question if compare this solution against Leetcode 1671 solution below:**

// Sample from Leetcode 1671 nested for loop when calculating LIS

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

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

if(nums[j] < nums[i]) {

left\_to\_right[i] = Math.max(left\_to\_right[i], left\_to\_right[j] + 1);

}

}

}

for(int i = len - 1; i >= 0; i--) {

    for(int j = len - 1; j > i; j--) {

        if(nums[j] < nums[i]) {

            right\_to\_left[i] = Math.max(right\_to\_left[i], right\_to\_left[j] + 1);

        }

    }

}

**The inner for loop scan j from 0 to i - 1 based on outer for loop i, which requires two for loop to fill in DP array "left\_to\_right", but in Leetcode 2100 here, to fill in DP array "nonIncreasing" or "nonDecreasing" we only need one for loop ?**

// Sample from Leetcode 2100 NO nested for loop required when calculating LIS

// Fill non-increasing count

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

if (security[i] <= security[i - 1]) {

nonIncreasing[i] = nonIncreasing[i - 1] + 1;

}

}

// Fill non-decreasing count

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

if (security[i] <= security[i + 1]) {

nonDecreasing[i] = nonDecreasing[i + 1] + 1;

}

}

**Refer to chatGPT**

The difference in the implementation of the two solutions (LeetCode 1671 and LeetCode 2100) is primarily due to the nature of the problem they are solving. Both problems deal with increasing or decreasing sequences, but they do so in different contexts and have different requirements for how they track these sequences. Let's break down why LeetCode 2100 only needs one loop per direction, while LeetCode 1671 requires nested loops:

**LeetCode 2100: "Find Good Days to Rob the Bank"**

**Problem Context:**

In this problem, you are checking whether there are time consecutive non-increasing days before a given day and time consecutive non-decreasing days after that day.

**Approach:**

You can use two simple arrays (nonIncreasing and nonDecreasing) to keep track of these counts.

A single loop from left to right fills the nonIncreasing array by checking if the current day's security level is less than or equal to the previous day's. If so, you increase the count.

Similarly, another single loop from right to left fills the nonDecreasing array.

**Why Only One Loop?**

**最大的不同是 Leetcode 2100 处理的是连续临近元素 consecutive array element，而不是 subsequent array element**

**The task is to track consecutive trends rather than searching for the longest increasing subsequence. Each day's status depends only on the previous day's status in one pass and doesn't require looking back over multiple days.**

**This means a straightforward linear scan with a rolling update is sufficient for this problem.**

**LeetCode 1671: "Minimum Number of Removals to Make Mountain Array"**

**Problem Context:**

Here, you need to create a mountain array (an increasing sequence followed by a decreasing sequence) with the minimum number of removals.

The solution involves finding the longest increasing subsequence (LIS) from both left to right and right to left and then combining these results.

**Approach:**

You maintain two DP arrays (left\_to\_right and right\_to\_left).

To find the LIS, you need to compare each element with all previous elements to ensure it forms a longer increasing sequence.

**Why Nested Loops?**

**To fill the left\_to\_right DP array for LIS:**

**Each element i must be compared to all previous elements j (j < i) to check if it can extend the subsequence ending at j.**

**Similarly, for the right\_to\_left DP array, you compare each element with all subsequent elements.**

**The nested loops are necessary because finding LIS is inherently a comparison-based task where each element needs to know how it relates to all previous elements to build the longest subsequence.**

**Key Differences**

Nature of the Problem:

LeetCode 2100: Checking consecutive trends over a fixed window (time). This only needs a single pass to keep a count.

LeetCode 1671: Searching for the longest subsequence, which requires comparisons against all previous elements, leading to nested loops.

DP Array Updates:

LeetCode 2100: Simple counting of consecutive non-increasing or non-decreasing days, updated in a linear pass.

LeetCode 1671: Dynamic Programming update based on potential subsequence lengths, requiring information from all previous steps to determine the longest valid sequence.

Computational Complexity:

LeetCode 2100: O(n) time complexity because each array is filled with a single pass.

LeetCode 1671: O(n^2) time complexity due to the nested loops required to compare each element with every other previous element.

**Summary**

The single loop in LeetCode 2100 is possible because it is sufficient to track a simple trend of consecutive increases or decreases. Each day's status only depends on the immediately preceding day's status.

In contrast, LeetCode 1671 needs nested loops to handle the longest increasing subsequence, as each element may potentially extend multiple different subsequences, which requires comparing it to all previous elements.

By understanding the specific requirements of each problem, we can see why different implementations are necessary to achieve the desired outcomes efficiently.

**Refer to**

[L1800.Maximum Ascending Subarray Sum (Ref.L300,L2100)](note://WEB449a32091b93466663ead85373aa7142)

[L300.Longest Increasing Subsequence](note://20C954956F6E4AA6A3575EA3146A1FCA)

[L1671.Minimum Number of Removals to Make Mountain Array (Ref.L300)](note://A717168457D646DF8CE8A9054DD61C12)