2100. Find Good Days to Rob the Bank

Medium Array Dynamic Programming Prefix Sum

Problem Description

In this problem, you are working with a group of thieves planning a bank heist. You have access to information about the bank's security, specifically the number of guards on duty on each day. The objective is to identify the days on which robbing the bank would be most feasible, based on the security guard patterns.

You are given two inputs:

1. A list of integers, `security`, where `security[i]` represents the number of guards on duty on day `i`.
2. An integer, `time`, which determines the number of days before and after day `i` that you must consider when evaluating if it's a good day to commit the robbery.

A day `i` is considered a good day to rob the bank if the following conditions are met:

- There's a span of at least `time` days before and after day `i`.
- The number of guards decreases or stays the same on each of the `time` days before day `i`.
- The number of guards increases or stays the same on each of the `time` days after day `i`.

Formally, this means day `i` can be chosen if `security[i - time] >= security[i - time + 1] >= ... >= security[i] <= ... <= security[i + time - 1] <= security[i + time]`.

The goal is to return a list of all the days that meet these criteria. It's important to note that the days in the returned list do not need to be in any particular order.

Intuition

To arrive at the solution for this problem, we need to analyze the pattern of guards before and after each potential day `i`. Specifically, we want to find days where there's a non-increasing number of guards for `time` days before `i`, and a non-decreasing number of guards for `time` days after `i`.

Here's the intuition behind the solution:

- First, we need to handle an edge case. If the total number of days available (`n`) is not sufficient to have `time` days before and after any given day, (i.e., if `n <= time * 2`), it's not possible to find any suitable day for the heist.
- We use two lists, `left` and `right`, to keep track of how many consecutive days before and after day `i` (including day `i` itself) satisfy the non-increasing and non-decreasing conditions, respectively.
- We iterate through the `security` list to fill out the `left` list. Starting from the second day (index 1), we check if the number of guards is less than or equal to the number of guards the day before. If it is, we increment the count from the previous day in the `left` list. This gives us the number of days counting backward from `i` with non-increasing guards.
- We do a similar iteration for the `right` list, but in reverse order, starting at the second-to-last day and moving backwards. If the number of guards is less than or equal to the number of guards the following day, we increment the count from the following day in the `right` list to get the number of days counting forward from `i` with non-decreasing guards.
- Once we have the `left` and `right` lists populated, we can iterate through them and select days `i` where both the `left` and `right` values are at least `time`.
- These days are our candidates for when the bank can be robbed, and we return these days as our final result.

The solution effectively uses dynamic programming concepts, where we build up our knowledge of guard patterns before and after each day, saving time by not needing to re-calculate the patterns for each day from scratch.

Solution Approach

To implement the solution for finding the best days to rob a bank, the following approach is used:

1. **Initialize Variables:** Firstly, we establish the length of the `security` list and create two lists, `left` and `right`, of the same length initialized to zeros. These lists will be used to track the count of consecutive days meeting the conditions leading up to (for `left`) and following (for `right`) each index in `security`.
2. **Populate the `left` List:** We iterate through the `security` list from the second element to the end (`for i in range(1, n)`). For each day `i`, if the number of guards is the same or fewer (`security[i] <= security[i - 1]`), it means the conditions for a non-increasing sequence up to day `i` are satisfied. Therefore, we increment the count from the day before in the `left` list (`left[i] = left[i - 1] + 1`).
3. **Populate the `right` List:** Similarly, we iterate through the `security` list in the reverse order starting from the second-to-last element (`for i in range(n - 2, -1, -1)`). If the number of guards is the same or fewer compared to the next day (`security[i] <= security[i + 1]`), the non-decreasing condition is satisfied for the time frame after day `i`. Consequently, we increment the count from the next day in the `right` list (`right[i] = right[i + 1] + 1`).
4. **Identify Good Days:** After building the `left` and `right` lists, the next step is to go through each day and check if it can be considered a good day for the robbery (`if for i in range(n) if time <= min(left[i], right[i])`). A day `i` is suitable if there's a non-increasing sequence of day counts in `left` and a non-decreasing sequence in `right`, both of at least length `time`. This is checked by confirming the value at index `i` in both `left` and `right` lists are greater than or equal to `time`. Days that meet these criteria are added to our result list.
5. **Return Result**: The final operation returns the list of days that are good for a bank robbery.

This implementation is efficient with a time complexity of O(n), where n is the number of days, because it makes a single pass through the `security` list for populating `left` and `right` lists and then another pass to collect the good days.

The data structures and algorithms used in this solution are characteristic of dynamic programming, where the problem is broken down into simpler subproblems (identifying non-increasing and non-decreasing sequences) that are solved once and used in the final computation.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose the input is as follows:

- `security`: [3, 4, 3, 2, 4, 1, 5]
- `time`: 2

Following the solution approach:

1. **Initialize Variables**: We have `security` with length `n = 7`. We initialize two lists, `left` and `right`, with zeros: `left = [0, 0, 0, 0, 0, 0, 0]` and `right = [0, 0, 0, 0, 0, 0, 0]`.
2. **Populate the `left` List**: We iterate from the second element:
   - For `i = 1`, `security[1]` is 4 and `security[i-1]` is 3, so `left[1]` remains 0 (guards increased).
   - For `i = 2`, `security[2]` is 3 and `security[i-1]` is 4, so `left[2]` becomes `left[1] + 1 = 1`.
   - For `i = 3`, `security[3]` is 2 and `security[i-1]` is 3, so `left[3]` becomes `left[2] + 1 = 2`.
   - Continuing this process, we get `left = [0, 0, 1, 2, 0, 1, 0]`.
3. **Populate the `right` List**: Iterating in reverse order starting from the second-to-last element:
   - For `i = 5`, `security[5]` is 1 and `security[i+1]` is 5, so `right[5]` becomes `right[6] + 1 = 1`.
   - For `i = 4`, `security[5]` is 4 and `security[i+1]` is 1, so `right[4]` remains 0 (guards decreased).
   - For `i = 3`, `security[3]` is 2 and `security[i+1]` is 4, so `right[3]` becomes `right[4] + 1 = 1`.
   - Continuing this process, we get `right = [0, 0, 0, 1, 0, 1, 0]`.
4. **Identify Good Days**: Now we check each day:
   - Day 2 (index 1) is not good because `min(left[1], right[1])` is 0, which is less than `time`.
   - Day 3 (index 2) is not good because `min(left[2], right[2])` is 0, less than `time`.
   - Day 4 (index 3) is a good day because `min(left[3], right[3])` is 1, which equals `time`.
   - No other day meets the criteria, so we get the result list as `[3]` (using indices starting from 0).
5. **Return Result**: The final returned result is `[3]`, indicating that the only good day for the robbery is day 4, if we consider the days indexed starting from 1.

In this example, only day 4 fulfills the condition of having a non-increasing sequence for `time` days before it and a non-decreasing sequence for `time` days after it. Thus, according to our algorithm, this would be the suitable day to plan the heist.

Solution Implementation

Python

```python
from typing import List  # Import the List type from the typing module

class Solution:
    def goodDaysToRobBank(self, security: List[int], time: int) -> List[int]:
        # Get the total number of days to analyze
        num_days = len(security)

        # If the total number of days is less than twice the required time, return empty list
        if num_days <= time * 2:
            return []

        # Initialize two lists to keep track of the non-increasing and non-decreasing sequences
        non_increasing = [0] * num_days
        non_decreasing = [0] * num_days

        # Count the number of days before the current day where the security value is non-increasing
        for i in range(1, num_days):
            if security[i] <= security[i - 1]:
                non_increasing[i] = non_increasing[i - 1] + 1

        # Count the number of days after the current day where the security value is non-decreasing
        for i in range(num_days - 2, -1, -1):
            if security[i] <= security[i + 1]:
                non_decreasing[i] = non_decreasing[i + 1] + 1

        # Find the days where the non-increasing and non-decreasing counts satisfy the time constraint
        good_days = [i for i in range(num_days) if time <= min(non_increasing[i], non_decreasing[i])]

        return good_days  # Return the list of good days to rob the bank
```

Java

```java
class Solution {
    public List<Integer> goodDaysToRobBank(int[] security, int time) {
        // Get the length of the `security` array.
        int n = security.length;
        // If the array is not sufficient to have days before and after the time period, return an empty list.
        if (n <= time * 2) {
            return Collections.emptyList();
        }

        // Arrays to keep track of the non-increasing trend to the left and non-decreasing trend to the right.
        int[] nonIncreasingLeft = new int[n];
        int[] nonDecreasingRight = new int[n];

        // Populate the nonIncreasingLeft array by checking if each day is non-increasing compared to the previous day.
        for (int i = 1; i < n; ++i) {
            if (security[i] <= security[i - 1]) {
                nonIncreasingLeft[i] = nonIncreasingLeft[i - 1] + 1;
            }
        }

        // Populate the nonDecreasingRight array by checking if each day is non-decreasing compared to the next day.
        for (int i = n - 2; i >= 0; --i) {
            if (security[i] <= security[i + 1]) {
                nonDecreasingRight[i] = nonDecreasingRight[i + 1] + 1;
            }
        }

        // To store the good days to rob the bank.
        List<Integer> goodDays = new ArrayList<>();

        // Check each day to see if it can be a good day to rob the bank.
        for (int i = time; i + time < n; ++i) {
            // A day is good if there are at least `time` days before and after it forming non-increasing and non-decreasing trends.
            if (time <= Math.min(nonIncreasingLeft[i], nonDecreasingRight[i])) {
                goodDays.add(i);
            }
        }

        // Return the list of good days.
        return goodDays;
    }
}
```

C++

```cpp
#include <vector>

class Solution {
public:
    // Function to find the days when it is good to rob the bank.
    std::vector<int> goodDaysToRobBank(std::vector<int>& security, int time) {
        // Get the size of the security vector
        int n = security.size();

        // If the total days are not enough to have 'time' days before and after, return empty list
        if (n <= time * 2) return {};

        // Dynamic programming arrays to store the non-increasing and non-decreasing streaks
        std::vector<int> leftIs, rightIs;  // From the left (non-increasing streak lengths)
        std::vector<int> rightDs, rightIs; // From the right (non-decreasing streak lengths)

        // Calculate the non-increasing streak from the left
        for (int i = 1; i < n; ++i) {
            if (security[i] <= security[i - 1]) {
                leftIs[i] = leftIs[i - 1] + 1;
            }
        }

        // Calculate the non-decreasing streak from the right
        for (int i = n - 2; i >= 0; --i) {
            if (security[i] <= security[i + 1]) {
                rightIs[i] = rightIs[i + 1] + 1;
            }
        }

        // Vector to store the good days to rob the bank
        std::vector<int> ans;

        // Corrected the two 'time' to 'time'
        for (int i = time; i + time < n; ++i) {
            // Check if the non-increasing streak on the left and the non-decreasing streak on the right
            // are both at least 'time' days long
            if (time <= std::min(leftIs[i], rightIs[i])) {
                ans.push_back(i); // If the condition is satisfied, it's a good day to rob
            }
        }

        // Return the resultant vector of good days to rob the bank
        return ans;
    }
};
```

TypeScript

```typescript
function goodDaysToRobBank(security: number[], time: number): number[] {
    const days = security.length;

    // If there are not enough days to support the 'time' constraint on both sides, return an empty list.
    if (days <= time * 2) {
        return [];
    }

    // Left and right arrays to track the number of non-increasing and non-decreasing days, respectively.
    const nonIncreasingCounts = new Array(days).fill(0);
    const nonDecreasingCounts = new Array(days).fill(0);

    // Fill in the nonIncreasingCounts and nonDecreasingCounts arrays.
    for (let i = 1; i < days; ++i) {
        if (security[i] <= security[i - 1]) {
            nonDecreasingCounts[i] = nonIncreasingCounts[i - 1] + 1;
        }
        if (security[days - i - 1] <= security[days - i]) {
            nonDecreasingCounts[days - i - 1] = nonDecreasingCounts[days - i] + 1;
        }
    }

    // Initialize the response array to store good days to rob the bank.
    const goodDays = [];

    // Iterate to find good days according to the 'time' constraint.
    for (let i = time; i < days - time; ++i) {
        // Check if the current day is good day by comparing non-increasing and non-decreasing counts to 'time'.
        if (time <= Math.min(nonIncreasingCounts[i], nonDecreasingCounts[i])) {
            goodDays.push(i);
        }
    }

    // Return the list of good days to rob the bank.
    return goodDays;
}
```

```python
from typing import List  # Import the List type from the typing module

class Solution:
    def goodDaysToRobBank(self, security: List[int], time: int) -> List[int]:
        # Get the total number of days to analyze
        num_days = len(security)

        # If the total number of days is less than twice the required time, return empty list
        if num_days <= time * 2:
            return []

        # Initialize two lists to keep track of the non-increasing and non-decreasing sequences
        non_increasing = [0] * num_days
        non_decreasing = [0] * num_days

        # Count the number of days before the current day where the security value is non-increasing
        for i in range(1, num_days):
            if security[i] <= security[i - 1]:
                non_increasing[i] = non_increasing[i - 1] + 1

        # Count the number of days after the current day where the security value is non-decreasing
        for i in range(num_days - 2, -1, -1):
            if security[i] <= security[i + 1]:
                non_decreasing[i] = non_decreasing[i + 1] + 1

        # Find the days where the non-increasing and non-decreasing counts satisfy the time constraint
        good_days = [i for i in range(num_days) if time <= min(non_increasing[i], non_decreasing[i])]

        return good_days  # Return the list of good days to rob the bank
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code can be broken down into the following parts:

1. Initializing the `left` and `right` arrays, each with a length of `n`. This operation takes `O(n)` time.
2. Filling in the `left` array with the lengths of non-increasing subsequences from the left: This requires iterating over the entire `security` list of length `n`, resulting in `O(n)` time.
3. Filling in the `right` array with the lengths of non-decreasing subsequences from the right: Similar to the `left` array, we iterate once in reverse over the length of the `security` list, which takes `O(n)` time.
4. Constructing the list of good days to rob the bank: We again iterate over the `security` list and compare the `left` and `right` values, which is an `O(n)` operation.

Adding all these operations together, the time complexity is `O(n) + O(n) + O(n) + O(n)`, which simplifies to `O(n)`.

Space Complexity

The space complexity of the code can also be analyzed based on the data structures being used:

1. The `left` and `right` arrays, both of which are of size `n`. These contribute `2n` space complexity.
2. The result list that is output has at most `n` elements. However, this space is accounted for in the result and not typically counted towards auxiliary space being used by the algorithm.

Considering these two factors, the space complexity is `O(n)` for the auxiliary space used, not including the output data structure.