2110. Number of Smooth Descent Periods of a Stock

Medium | Array | Math | Dynamic Programming | Leetcode Link

Problem Description

You are tasked with analyzing the price history of a stock, given as an array of integers named `prices`. Each element in this array corresponds to the price of the stock on a specific day. A *smooth descent period* is identified when there is a sequence of one or more consecutive days where each day's stock price is exactly `1` lower than the previous day's price (except for the first day of this sequence, which does not have this restriction).

The goal is to calculate the total number of smooth descent periods in the stock price history provided by `prices`.

Intuition

The intuition behind solving this problem is to traverse the `prices` array and identify segments where the prices are decreasing by exactly `1` each day. These segments are the smooth descent periods we are interested in counting.

To achieve this, we can iterate over the `prices` array using two pointers or indices. The first pointer `i` marks the start of a potential descent period, while the second pointer `j` explores the array to find the end of this descent. As long as the difference between the prices of two consecutive days (`prices[j - 1] - prices[j]`) equals `1`, we continue moving `j` forward, extending the current descent period.

Once we reach the end of a descent period (when the difference is not equal to `1`), we calculate the total number of descent periods within the segment marked by `i` and `j`. This calculation can be done by using the formula for the sum of the first `n` natural numbers, as the number of descent periods formed by a contiguous descending sequence is equivalent to the sum of an arithmetic series starting from `1`. The formula `ans += (1 + cnt) * cnt // 2` is used, where `cnt` is the length of the current descent period.

After adding to the total count `ans`, we set `i` to the current position of `j` and proceed to find the next potential descent period in the array. This process continues until we have traversed the entire array and evaluated all potential smooth descent periods.

By using this approach, we ensure a linear time complexity of O(n), where `n` is the number of days in the `prices` array, since each element is visited only once.

Solution Approach

The solution uses a straightforward linear scan algorithm with two pointers to traverse the `prices` array without the need for additional data structures. Here's a step-by-step breakdown of how it's implemented:

1. Initialize a variable `ans` to `0`. This will hold the cumulative number of smooth descent periods.
2. Set two pointers (or indices) `i` and `j`. `i` starts at `0`, marking the beginning of a potential smooth descent sequence.
3. Initiate a `while` loop that will run as long as `i` is less than the length of the `prices` array (`n`).
4. Inside the loop, increment `j` starting from `i+1` as long as `j` is less than `n` and the price difference between two consecutive days is exactly `1` (`prices[j - 1] - prices[j] == 1`). This locates the end of the current descending period.
5. Once the end of a descent period is found, calculate the length of this descent period (`cnt = j - i`). This count represents the number of contiguous days in the current smooth descent period.
6. Use the arithmetic series sum formula to find the total number of smooth descent periods in the current sequence. The formula to use is `(1 + cnt) * cnt // 2`, which is then added to `ans`.
7. After computing the number of descent periods for the current segment, move `i` to `j`, the position where the current descent segment ended. This is to start checking for a new descent period from this point forward.
8. Repeat steps 4 to 7 until the entire `prices` array has been scanned.
9. After completing the traversal, return the total count `ans` as the final answer.

The implementation uses the concept that the sum of an arithmetic series can calculate the number of distinct smooth descent periods within a given range. Since in each contiguous descent sequence, the difference is '1', it forms a series like `1, 2, ..., cnt`. The sum of these numbers, representing different lengths of smooth descent periods that can be formed, is calculated using the equation `(1 + cnt) * cnt // 2`. By summing up such counts for all descending sequences found in `prices`, we obtain the overall number of smooth descent periods in the entire array.

Example Walkthrough

To illustrate the solution approach, let's consider a small example where the given `prices` array is `[5, 4, 3, 2, 8, 7, 6, 5, 4, 10]`.

Let's walk through the steps outlined in the solution approach:

1. Initialize `ans` to `0`. This is where we will accumulate the number of smooth descent periods.
2. Set pointers `i` and `j` to `0` and `1`, respectively, to start tracking a potential smooth descent period from the beginning of the array.
3. We now enter the `while` loop since `i` (0) is less than the length of the `prices` array (10).
4. The first potential descent starts at index `0` with a price of `5`. We start moving `j` forward and find that `prices[0] - prices[1]` is `1`, and the same goes for `prices[1] - prices[2]` and `prices[2] - prices[3]`, until we reach `prices[3] - prices[4]` which is not `1` (since `2 - 8` is `-6`). Therefore, we have identified the first descent period from index `0` to `3`.
5. We calculate the length of this descent period: `cnt = j - i` which in this case is `4 - 0 = 4`.
6. Using the arithmetic series sum formula, we add the number of smooth descents in this sequence to `ans`: `(1 + cnt) * cnt // 2` which is `(1 + 4) * 4 // 2 = 10`. So now, `ans = 10`.
7. We move `i` to the position where `j` is now (`i = j`), making `i` `4`, and look for the next descent period starting from index `4`.
8. Now `j` moves forward again, and we see that we have a descent from `8` at index `4` to `4` at index `8`. We perform the same calculations as before and find that `cnt = j - i = 9 - 4 = 5`. The number of descents is `(1 + cnt) * cnt // 2` which equals `(1 + 5) * 5 // 2 = 15`, and we add this to `ans` making it now `10 + 15 = 25`.
9. Finally, we have finished the `while` loop since `j` has reached the end of the `prices` array.
10. Return the total count `ans` which is `25`. This is the total number of smooth descent periods in the example price history.

This walkthrough demonstrates the process and calculations in the solution approach using the given problem description and solution strategy.

Python Solution

```python
class Solution:
    def getDescentPeriods(self, prices: List[int]) -> int:
        # Initialize the total number of descent periods
        total_periods = 0

        # Initialize the starting index and find the length of the prices list
        start_index = 0
        length = len(prices)

        # Iterate through the prices list
        while start_index < length:
            # Find the consecutive descending sequence by checking the price difference
            end_index = start_index + 1
            while end_index < length and prices[end_index - 1] - prices[end_index] == 1:
                end_index += 1

            # Calculate the length of the descent sequence
            descent_length = end_index - start_index

            # Using the arithmetic series sum formula, (n/2)*(first term + last term),
            # here it simplifies to (1 + descent_length) * descent_length / 2
            # since the difference between consecutive terms is 1.
            total_periods += (1 + descent_length) * descent_length // 2

            # Move the start index to the end of the current descent sequence
            start_index = end_index

        # Return the total number of descent periods found
        return total_periods

```

Java Solution

```java
class Solution {
    public long getDescentPeriods(int[] prices) {
        long totalDescentPeriods = 0; // Initialize the result variable to keep track of the total descent periods.
        int n = prices.length; // Get the total number of elements in the prices array.

        // Iterate over the prices array.
        for (int start = 0, end = 0; start < n; start = end) {
            // Initialize end to the next element after start for the next potential descent.
            end = start + 1;

            // Find a contiguous subarray where each pair of consecutive elements
            // have difference equals to 1. This forms a descent period.
            // The while loop will continue until the condition fails,
            // indicating we've reached the end of the current descent period.
            while (end < n && prices[end - 1] - prices[end] == 1) {
                ++end;
            }

            // Calculate the length of the current descent period.
            int periodLength = end - start;

            // Using the arithmetic progression sum formula to count all individual
            // and overlapping periods: n*(n+1)/2, where n is the length of the current descent.
            totalDescentPeriods += (1L + periodLength) * periodLength / 2;
        }

        // Return the total number of descent periods found in the prices array.
        return totalDescentPeriods;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    long long getDescentPeriods(vector<int>& prices) {
        long long totalDescentPeriods = 0;  // Holds the sum of all descent periods
        int sequenceLength = prices.size(); // Total number of elements in prices

        // Loop through each price in the vector
        for (int start = 0, end = 0; start < sequenceLength; start = end) {
            end = start + 1;

            // Continue to find descending contiguous subsequences where each
            // element is one less than the previous one
            while (end < sequenceLength && prices[end - 1] - prices[end] == 1) {
                ++end;
            }

            // Calculate the length of the contiguous subsequence
            int count = end - start;

            // Add the total number of descent periods that can be
            // formed with the subsequence of length 'count'
            totalDescentPeriods += (1LL + count) * count / 2;
        }

        return totalDescentPeriods; // Return the final total of descent periods
    }
};

```

Typescript Solution

```typescript
// Counts the total number of "descent periods" in a given array of prices.
// A "descent period" is defined as one or more consecutive days where the
// price of a stock decreases by exactly 1 each day.
function getDescentPeriods(prices: number[]): number {
    let totalDescentPeriods = 0; // Store the total count of descent periods.
    const pricesLength = prices.length; // The length of the 'prices' array.

    // Iterate through the 'prices' array to identify descent periods.
    for (let startIndex = 0, endIndex = 0; startIndex < pricesLength; startIndex = endIndex) {
        endIndex = startIndex + 1;

        // Check if the next price is one less than the current; if so, extend the descent period.
        while (endIndex < pricesLength && prices[endIndex - 1] - prices[endIndex] === 1) {
            endIndex++;
        }

        // Calculate the number of prices in the current descent period.
        const descentPeriodLength = endIndex - startIndex;

        // Calculate the count of descent periods using the formula for the sum of the first n integers.
        totalDescentPeriods += Math.floor(((1 + descentPeriodLength) * descentPeriodLength) / 2);
    }

    // Return the total count of descent periods found in the 'prices' array.
    return totalDescentPeriods;
}

```

Time and Space Complexity

The provided Python code defines a method `getDescentPeriods` which calculates the total number of "descent periods" within a list of prices. A descent period is defined as a sequence of consecutive days where the price of each day is one less than the price of the day before.

Time Complexity:

The time complexity of the code is `O(n)`, where `n` is the number of elements in the `prices` list. This is because the function uses a while loop (and a nested while loop) that traverses the list exactly once. Each element is visited once by the outer loop, and the inner loop advances the index `j` without revisiting any previously checked elements. The computations within the loops are simple arithmetic operations, which take constant time. Therefore, the number of operations increases linearly with the number of elements in the input list, resulting in a linear time complexity.

Space Complexity:

The space complexity of the code is `O(1)`. The code only uses a constant amount of space (variables `ans`, `i`, `n`, `j`, and `cnt`) that does not depend on the input list's size. It does not use any additional data structures that grow with the input size, so the amount of space used remains constant even as the size of the input list increases.