901. Online Stock Span

Medium Stack Design Data Stream Monotonic Stack Leetcode Link

Problem Description

The objective is to design an algorithm that can process a series of daily stock prices and determine the stock's price span for any given day. The price span is defined as the maximum number of consecutive days (counting backwards from the current day) during which the stock's price was less than or equal to that day's price. For example, if the sequence of stock prices over the last few days was `[6, 5, 4, 5]` and the price today is `5`, the span would be `2`, as the price today and the day before were `5` or less.

The main challenge is to efficiently calculate the span while handling a potentially large stream of daily prices. A naive approach that checks all previous stock prices daily would not be efficient enough.

Intuition

To find a span efficiently, we use a *monotonic stack*. This is a fundamental data structure that is useful when we want to find the next element greater or smaller in an array. In this case, we are looking for the first price greater than the current price as we move backward in time.

The monotonic stack maintains a decreasing sequence of prices along with their corresponding spans. When a new price arrives, we compare it with the price at the top of the stack. If the new price is greater, we pop elements from the stack, accumulating their spans as we go, until we find a higher price in the stack or the stack is empty.

We then push the current price and the accumulated span onto the stack. The accumulated span is the answer we are looking for - it tells us how far back we can go with prices less than or equal to the current price.

This approach is efficient because each price is processed and pushed to the stack exactly once, and popped at most once. This gives the solution a time complexity that is average-case O(1) for each call to `next`, which is much more efficient than the naive approach.

Solution Approach

The `StockSpanner` class is implemented using a monotonic stack pattern. This pattern is particularly useful for problems where we need to know the nearest element that is greater or smaller than the current element. In this case, our stack stores pairs of the form `(price, cnt)` where `price` is the stock price and `cnt` is the span of the stock for that price.

The stack is maintained in a way that the prices are monotonically decreasing from the bottom to the top. This means that when we consider a new stock price, we can repeatedly compare it with the elements from the top of the stack. If the new stock price is higher than the price at the top of the stack, we pop the stack and add the span (`cnt`) of that popped price to a running count. This process continues until we find a price in the stack that is greater than the current price or the stack becomes empty.

Here is how the implementation is performed step-by-step:

- When `next(price)` is called, we set a variable `cnt` to 1. This `cnt` will accumulate the span of days for the current price.
- We then enter a loop that keeps running as long as there is an element in the stack and the top element's price is less than or equal to the current price.
- Inside the loop, we pop the top element from the stack and add its span (second element of the pair) to `cnt`.
- After breaking out of the loop (either by finding a larger price or emptying the stack), we push the current price and accumulated span (`cnt`) as a pair onto the stack.
- The `next` function then returns the accumulated `cnt`, which represents the span of the stock price for the given day.

Each `next` operation runs in O(1) average time complexity because even though we have a while loop inside the function, each element is pushed and popped from the stack at most once due to the property of a monotonic stack.

This clever use of a monotonic stack provides a time-efficient solution to calculating the spans, avoiding the need to traverse the entire list of previous stock prices for every call. As a result, the algorithm scales well with a large number of `next` operations.

Example Walkthrough

Let's illustrate the solution approach with a small example. Assume the stock prices over the past few days were `[100, 80, 60, 70, 60, 75, 85]`. We will walk through the sequence of `next()` operations with these prices and discuss how the `StockSpanner` class calculates the span for each day.

1. **next(100)**: The stack is initially empty, so the span for the first price (100) is 1 (day). We push `(100, 1)` onto the stack.

   Stack: `[(100, 1)]`

2. **next(80)**: 80 is less than 100, so the span is also 1. We push `(80, 1)` onto the stack.

   Stack: `[(100, 1), (80, 1)]`

3. **next(60)**: Similarly, 60 is less than 80, so the span is again 1. Push `(60, 1)` onto the stack.

   Stack: `[(100, 1), (80, 1), (60, 1)]`

4. **next(70)**: Now 70 is greater than 60, so we start popping from the stack. We pop `(60, 1)` since 60 is less than 70, and we accumulate the span, so `cnt` becomes 2. The top is now 80 which is greater than 70, so we stop and push `(70, 2)` onto the stack.

   Stack: `[(100, 1), (80, 1), (70, 2)]`

5. **next(60)**: The incoming price is 60 which is less than 70, so span is 1. Push `(60, 1)` onto the stack.

   Stack: `[(100, 1), (80, 1), (70, 2), (60, 1)]`

6. **next(75)**: For incoming price 75, we pop `(60, 1)` and `(70, 2)` because those are smaller than 75 and accumulate their spans to `cnt` which now becomes 4. Now the top is 80 which is greater, so we stop and push `(75, 4)` onto the stack.

   Stack: `[(100, 1), (80, 1), (75, 4)]`

7. **next(85)**: For incoming price 85, we first pop `(75, 4)` as it is smaller and add its span to `cnt` making `cnt` 5. Next, we pop `(80, 1)` and add its span, so `cnt` is now 6. Now the top is 100 which is greater, so we push `(85, 6)` onto the stack.

   Stack: `[(100, 1), (85, 6)]`

Through the sequence of `next()` operations, we determined the spans without having to loop through all previous stock prices every time. The output sequence of spans would be `[1, 1, 1, 2, 1, 4, 6]` for the respective stock prices. This demonstrates the efficiency of the monotonic stack in calculating the price spans.

Python Solution

```python
class StockSpanner:
    def __init__(self):
        self.stack = []  # This stack will keep track of stock prices and their spans

    def next(self, price: int) -> int:
        # The count starts at 1, since the span of at least the current day is included
        span_count = 1
        # We pop prices from the stack if they are less than or equal to the current price.
        # When a lower price is encountered, its span is added to the count.
        while self.stack and self.stack[-1][0] <= price:
            span_count += self.stack.pop()[1]
        # Once all smaller (or equal) prices are popped, we append the current price and its calculated span.
        self.stack.append((price, span_count))
        return span_count

# Example Usage:
# obj = StockSpanner()
# span = obj.next(price)
```

Java Solution

```java
import java.util.Deque;
import java.util.ArrayDeque;

class StockSpanner {
    // Stack to keep track of stock prices and span counts
    // Each stack element is an array with the first element being the stock price
    // and the second being the span count for that price
    private Deque<int[]> stack = new ArrayDeque<>();

    // Constructor for StockSpanner
    public StockSpanner() {
    }

    // This method is called for every new stock price and returns the span count
    // The span count is the number of consecutive days the stock price has been
    // less than or equal to the current day's price, including the current day
    public int next(int price) {
        // Start the count as 1 for the current day
        int count = 1;

        // Continue popping elements from the stack when the current price is greater
        // than the price at the stack's top and accumulate the span counts
        while (!stack.isEmpty() && stack.peek()[0] <= price) {
            count += stack.pop()[1];
        }

        // Push the current price along with its span count onto the stack
        stack.push(new int[] {price, count});

        // Return the computed span count for the current price
        return count;
    }
}

/**
 * The following "how to use" example is not part of the class itself and serves only as an illustration.
 *
 * How to use:
 * StockSpanner stockSpanner = new StockSpanner();
 * int spanCount = stockSpanner.next(price);
 */
```

C++ Solution

```cpp
#include <stack>
using namespace std;

class StockSpanner {
public:
    StockSpanner() {
        // Constructor, no initialization needed since the stack is empty initially
    }

    // Processes a new price, returning the span of that price
    int next(int price) {
        int span = 1; // Span always starts at least at 1 (for the current day)

        // Continue popping from the stack while the stack is not empty and
        // the price at the top of the stack is less than or equal to the current price
        while (!pricesAndSpans.empty() && pricesAndSpans.top().first <= price) {
            span += pricesAndSpans.top().second; // Add the span of the price at the top
            pricesAndSpans.pop(); // Remove the price and its span now that it's processed
        }

        // Push current price and its calculated span onto the stack
        pricesAndSpans.emplace(price, span);

        // Return the span of the current price
        return span;
    }

private:
    // A stack that stores pairs of price and corresponding span
    stack<pair<int, int>> pricesAndSpans;
};

/**
 * Your StockSpanner object will be instantiated and called as such:
 * StockSpanner* obj = new StockSpanner();
 * int param_1 = obj->next(price);
 */
```

Typescript Solution

```typescript
// Initialize an empty stack to keep track of stock prices and spans.
let stack: number[][] = [];

/**
 * Processes the next price.
 *
 * @param price - The current day's stock price.
 * @returns The span (number of consecutive days) of the stock price
 * ending with the current day for which the price is less than or equal
 * to the current day's price.
 */
function next(price: number): number {
    // Start the current span at 1 (itself)
    let span = 1;

    // If the stack is not empty and the top stack price is less than or equal
    // to the current price, keep popping from the stack and increment the span
    // by the span of the popped prices as their span becomes part of the current
    // span since our price is greater than those prices.
    while (stack.length && stack[stack.length - 1][0] <= price) {
        span += stack.pop()[1];
    }

    // Push the current price and its calculated span into the stack.
    stack.push([price, span]);

    // Return the span for the current price.
    return span;
}

// Now functions next can be called globally with the current day's price
// and will process the stock price span as per the described logic.
// For example:
// var spanToday = next(100);
// This will calculate the span for the stock price 100.
```

Time and Space Complexity

Time Complexity

The time complexity of the `StockSpanner` class operations are as follows:

- `__init__`: The constructor has a time complexity of `O(1)` as it only initializes an empty list.

- `next`: The time complexity of this function is amortized `O(1)`. Although at first glance it seems to be `O(n)` per call (because of the `while` loop), it's important to note that each element is added once and removed at most once. Therefore, over `n` calls, there will be at most `n` additions and `n` removals, leading to an average complexity of `O(1)` per call. The reference answer states the time complexity as `O(n)`, which could be referring to the worst-case scenario for a single call, but amortized over multiple calls, it's constant.

Space Complexity

The space complexity of the `StockSpanner` class is `O(n)`, where `n` is the number of prices processed. In the worst case, if the prices are strictly decreasing, all prices will be stored in the stack `self.stk`.