**Solution**

**Overview**

The brute force approach would be to keep an array of all the stock prices, and then iterate backwards on each call to next until you find a price that is greater than the current price.

The problem with this approach is that we are potentially looping over the same elements many times. Let's say that the stock's price history looks like this:

[50, 45, 40, 35, 30]

Now, we call next(46). The problem wants us to find the number of consecutive days where the stock price was less than or equal to the current price. This is equivalent to finding how many days it has been since the stock price was greater than price, which is the 50. With our brute force approach, we would need to iterate back over the [45, 40, 35, 30] before reaching 50.

Now, let's call next(47). The stock price history is now [50, 45, 40, 35, 30, 46]. Once again, we need to iterate over the elements [45, 40, 35, 30] before reaching the 50. However, we already iterated over them on the previous call. Is there a way that we can save information on each call to next so that we don't need to repeat iterating over these elements?

**Approach: Monotonic Stack**

**Intuition**

For each price, we want to find the most recent day that had a price greater than price. In the example used above, when we call next(47), the first element we check is 46. Because 46 is not greater than 47, we need to keep looking.

When we found the answer for 46, we had to look at 30, 35, 40, 45, all of which are not greater than 46.

If 46 is not greater than 47, then all of the numbers that are not greater than 46 are also not greater than 47. Therefore, it is a waste to iterate over those four elements since we already know they are less than 46, which means they must also be less than 47.

When we call next(46), it returns 5 (that's how many days the stock is less than or equal to 46). This tells us that it took 5 days to find a price greater than 46. When we call next(47), we can use this information to our advantage - if it took 5 days to find a price greater than 46, then it will definitely take at least 5 days to find a price greater than 47.

If we save the answer for 46, then we can use this idea to save time on the call to next(47).

**Use a monotonic stack**

A monotonic stack is a stack in which the elements are always sorted. A stack can be monotonically increasing (sorted ascending) or monotonically decreasing (sorted descending). Let's say we have a monotonic decreasing stack. If we want to push x, then all elements that are less than x are popped off first to maintain the sorted property. For example, if we have stack = [623, 532, 125] and we want to push 615, then the 532 and 125 must be popped before the 615 is pushed, otherwise the stack will no longer be sorted.

In this problem, we realized that we can use the answer from earlier days to our advantage for future days. Let's look at the example given in the problem description.

Calls to next are in the order: [100, 80, 60, 70, 60, 75, 85].

The correct output is given as: [1, 1, 1, 2, 1, 4, 6].

At the 100: this is the first day, which means the answer is automatically 1 (only count the current day).

At the 80: look at the previous day. It is 100, which is greater than 80. The answer is 1 (only count the current day).

At the 60: look at the previous day. It is 80, which is greater than 60. The answer is 1.

At the 70: look at the previous day. It is 60, which is not greater than 70. The answer for 60 was 1, which means that it took at least one day to see a price greater than 60, which means it will also take at least one **more** day to see a price greater than 70. Going one more day back, we find 80, which is greater than 70, so the answer is 2.

At the next 60: look at the previous day. It is 70, which is greater than 60. The answer is 1.

At the 75: look at the previous day. It is 60, which is not greater than 75. The answer for this 60 was 1, which means it took at least one day to see a price greater than 60, which means it will also take at least one **more** day to see a price greater than 70. Going one more day back, we find 70, which is also not greater than 75. The answer for 70 was 2, which means it will take two **more** days to find a price greater than 75. Going two days back, we finally find the 80, which gives us the answer 4.

At the 85: look at the previous day. It is 75, which is not greater than 85. The answer for 75 was 4, so we need to look at least 4 days back. Going four days back, we find the 80, which is not greater than 85. The answer for 80 was 1, so we look one more day back, and find 100. The final answer is 6.

**We can implement this logic with a monotonically decreasing stack.**

As you can see, when we walked through the example, we stopped as soon as we found a value greater than price. We iterated backwards as long as the elements we looked at were less than price. This is the exact same logic described earlier (with the example of pushing 615 onto a stack that was [623, 532, 125]).

Declare a monotonic decreasing stack stack, and push stock prices onto stack. For any given call to next(price), look at the top of the stack. While the top of the stack has a value (stock price) less than or equal to price, it should be included in our answer, so pop it off the stack.

How do we calculate the answer? As mentioned in the examples above, we need to "remember" the answer for old calls. Let's push the answer for each day onto the stack along with the price.

The minimum answer for any day is 1, because we can just include the current day as part of the span. Then, if we pop from the stack, we should also add the answer associated with the day popped to our answer. This was also explained in the test case walkthrough - for example, when we got to the 85, we saw the previous element was 75 and took 4 days. That means 4 days were included in the span for 75, and of course all those days will also be included in the span for 85 since 85 > 75.

Monotonic stack problems can be really hard to wrap your head around initially, but it gets a lot easier with practice. If you are having trouble understanding, look at the code and try walking through the example test case ([100, 80, 60, 70, 60, 75, 85]) line by line using paper.

To summarize the algorithm: for a given price, say the previous element was y, where y <= price. However many days are included in the span for y must also be included in the span for price. This must be true because if there was an x > price earlier, that x would have also terminated the y since y <= price.

**Algorithm**

1. Initialize a stack stack. The stack will store elements in the format [price, answer] in a monotonic decreasing manner.
2. On each call to next:
   * First set ans = 1 representing the answer.
   * The top of the stack has a format [priceTop, answerTop]. While priceTop <= price, add answerTop to ans and pop from the stack.
   * Push the current [price, ans] onto the stack.
   * Return ans.

**Implementation**

class StockSpanner {

public:

    stack<pair<int,int>> stack;

    int next(int price) {

        int ans = 1;

        while (!stack.empty() && stack.top().first <= price) {

            ans += stack.top().second;

            stack.pop();

        }

        stack.push({price, ans});

        return ans;

    }

};

/\*\*

 \* Your StockSpanner object will be instantiated and called as such:

 \* StockSpanner\* obj = new StockSpanner();

 \* int param\_1 = obj->next(price);

 \*/

**Complexity Analysis**

Given nn*n* as the number of calls to next,

* Time complexity of each call to next: O(1)O(1)*O*(1)

Even though there is a while loop in next, that while loop can only run nn*n* times total across the entire algorithm. Each element can only be popped off the stack once, and there are up to nn*n* elements.

This is called amortized analysis - if you average out the time it takes for next to run across nn*n* calls, it works out to be O(1)O(1)*O*(1). If one call to next takes a long time because the while loop runs many times, then the other calls to next won't take as long because their while loops can't run as long.

* Space complexity: O(n)O(n)*O*(*n*)

In the worst case scenario for space (when all the stock prices are decreasing), the while loop will never run, which means the stack grows to a size of nn*n*.