<https://leetcode.com/problems/online-stock-span/description/>

Design an algorithm that collects daily price quotes for some stock and returns **the span** of that stock's price for the current day.

The **span** of the stock's price in one day is the maximum number of consecutive days (starting from that day and going backward) for which the stock price was less than or equal to the price of that day.

* For example, if the prices of the stock in the last four days is [7,2,1,2] and the price of the stock today is 2, then the span of today is 4 because starting from today, the price of the stock was less than or equal 2 for 4 consecutive days.
* Also, if the prices of the stock in the last four days is [7,34,1,2] and the price of the stock today is 8, then the span of today is 3 because starting from today, the price of the stock was less than or equal 8 for 3 consecutive days.

Implement the StockSpanner class:

* StockSpanner() Initializes the object of the class.
* int next(int price) Returns the **span** of the stock's price given that today's price is price.

**Example 1:**

Input

["StockSpanner", "next", "next", "next", "next", "next", "next", "next"]

[[], [100], [80], [60], [70], [60], [75], [85]]

Output

[null, 1, 1, 1, 2, 1, 4, 6]

Explanation

StockSpanner stockSpanner = new StockSpanner();

stockSpanner.next(100); // return 1

stockSpanner.next(80); // return 1

stockSpanner.next(60); // return 1

stockSpanner.next(70); // return 2

stockSpanner.next(60); // return 1

stockSpanner.next(75); // return 4, because the last 4 prices (including today's price of 75) were less than or equal to today's price.

stockSpanner.next(85); // return 6

**Constraints:**

* 1 <= price <= 105
* At most 104 calls will be made to next

**Attempt 1: 2023-03-19**

**Solution 1: Decreasing Monotonic Stack (30 min)**

class StockSpanner {

Stack<int[]> stack;

public StockSpanner() {

stack = new Stack<int[]>();

}

public int next(int price) {

int count = 1;

while(!stack.isEmpty() && stack.peek()[0] <= price) {

count += stack.pop()[1];

}

stack.push(new int[] {price, count});

return count;

}

}

/\*\*

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

\* StockSpanner obj = new StockSpanner();

\* int param\_1 = obj.next(price);

\*/

**Refer to**

<https://leetcode.com/problems/online-stock-span/solutions/640358/java-solution-with-visualization-and-easy-explained/>

**Let's start with code, then we move to visualization!**

**Algorithm**

Initialize a stack stack. The stack will store elements in the format [price, answer] in a monotonic decreasing manner.

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.

class StockSpanner {

/\*

We should have a stack of a pair of (current price, maximum number of consecutive days)

Since we don't have an access to the indicies.

\*/

Stack<int[]> s;

public StockSpanner() {

s = new Stack<>();

}

/\*

Let's trace the algorithm together on [100, 80, 60, 70, 60, 75, 85]

1. calling StockSpanner.next(100) should result in first element in our stack will be (100, 1) (s.size() == 1)

2. calling StockSpanner.next(80) should result in second element in our stack will be (80, 1) (s.size() == 2)

3. calling StockSpanner.next(60) should result in third element in our stack will be (60, 1) (s.size() == 3)

4. Now on calling StockSpanner.next(70) we should add span of (60) + 1 {the current day}

and remove it from stack (70, 2) (s.size() == 3)

5. Now on calling StockSpanner.next(60) result in fourth element in our stack will be (60, 1) (s.size() == 4)

6. Now on calling StockSpanner.next(75) we should add span of (60) and (70) + 1 {the current day}

and remove it from stack : (75, 4) (s.size() == 3)

7. Now on calling StockSpanner.next(85) we should add span of (75) and (80) + 1 {the current day}

and remove it from stack : (85, 6) (s.size() == 2)

\*/

public int next(int price) {

int span = 1;

while (!s.isEmpty() && price >= s.peek()[0]) { // If the current price is greater than stack peek.

span += s.peek()[1];

s.pop();

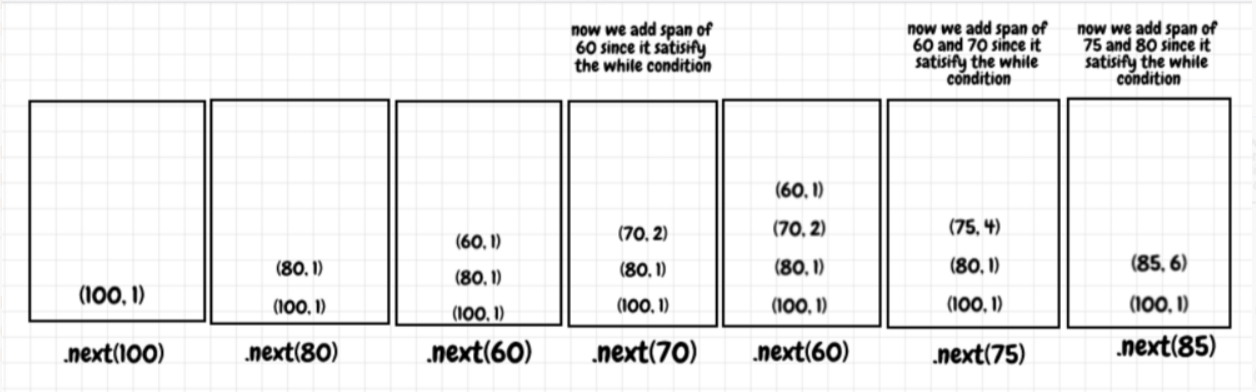
}

s.push(new int[]{price, span});

return span;

}

}



**Complexity Analysis**

Given n as the number of calls to next,

* Time complexity of each call to next: O(1)
* Even though there is a while loop in next, that while loop can only run n times total across the entire algorithm. Each element can only be popped off the stack once, and there are up to n elements.
* This is called amortized analysis - if you average out the time it takes for next to run across n calls, it works out to be 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)
* 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 n.