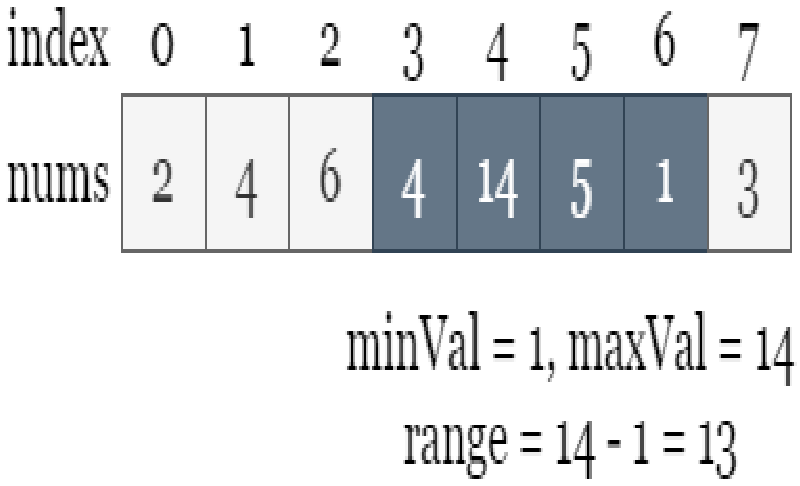
Sum of Subarray Ranges
LeetCode
68739
Oct 19, 2022
Editorial

Solution

Overview

Define the **range** of a subarray as the difference between the largest and smallest element in the subarray:



The task is to find the sum of all subarray ranges of the given array `nums`.

Approach 1: Two Loops

Intuition

Let's start with a brute force solution, that is, to find and iterate over all subarrays of `nums`, and get the sum of their ranges.

1. Set `answer = 0`.
2. Iterate over every left index of subarrays `left`.
3. With every fixed `left`, iterate over every right index `right` of subarrays.

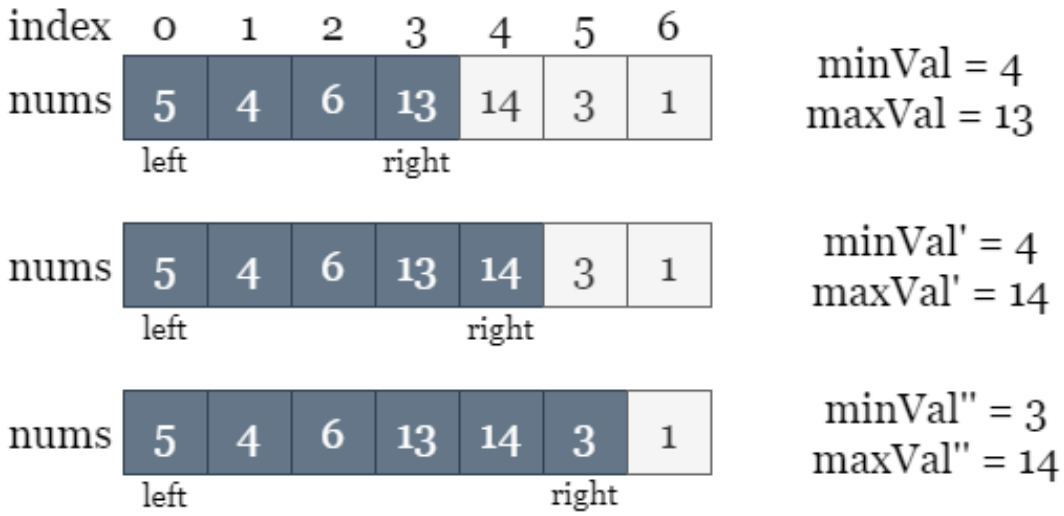
4. For each subarray `[left, right]`, iterate over it to find its minimum value `minVal` and maximum value `maxVal`.
5. Increment `answer` by `maxVal - minVal`.

This approach contains three nested loops which make the time complexity quite high, so it may not pass all test cases. But we can consider this as a prompt for better approaches!

Note that for a fixed `left` index, two adjacent arrays only differ by one element. Suppose the previous array is `[left, right]` and the new array is `[left, right + 1]`, we can get the `minVal, maxVal` for the new subarray, by updating `minVal, maxVal` of the previous array using `nums[right + 1]`.

- `minVal = min(minVal, nums[right + 1])`
- `maxVal = max(maxVal, nums[right + 1])`

Therefore, the average time for finding the range of one subarray is reduced to $O(1)$. Please refer to the following picture.



Algorithm

1. Set `answer = 0`.
2. Iterate over every left index of subarrays `left`.
3. With every fixed `left`, initialize `minVal = maxVal = nums[left]`, iterate over every right index `right` of subarrays.
4. For each right index `right`, update `minVal` and `maxVal` by `nums[right]`. Then update `answer += maxVal - minVal`.

Implementation

```
class Solution {


    public long subArrayRanges(int[] nums) {
        int n = nums.length;
        long answer = 0;


        for (int left = 0; left < n; ++left) {
            int minVal = nums[left], maxVal = nums[left];
            for (int right = left; right < n; ++right) {
                minVal = Math.min(minVal, nums[right]);
                maxVal = Math.max(maxVal, nums[right]);
                answer += maxVal - minVal;
            }
        }
        return answer;
    }
}
```

Complexity Analysis

Let n be the size of the input array `nums`.

- Time complexity: $O(n2)$
    - We have two nested iterations over `nums`.
    - In each step, we update `minVal, maxVal` and `answer`, it takes constant time.
    - To sum up, the overall time complexity is $O(n2)$.
- Space complexity: $O(1)$
    - We only need to update three variables `minVal`, `maxVal` and `answer`.

Approach 2: Monotonic Stack

Intuition

From the definition of the sum of all subarray ranges:

$$\sum_k \text{range}_k = \sum_k (\text{maxVal}_k - \text{minVal}_k) = \sum_k \text{maxVal}_k - \sum_k \text{minVal}_k$$

It implies that we can calculate these two partial sums separately.

Let's think of this problem differently, instead of finding each subarray and getting its `minVal` and `maxVal`, we focus on each number. If we can find that, for each number `nums[i]`, the number of subarrays having `nums[i]` as its **minimum value** is `minTime[i]`. Then the sum of `minVal` can be rewritten as:

$$\sum_k \text{minVal}_k = \sum_{i=1}^{n} \text{minTime}[i] \cdot \text{nums}[i]$$

For example, we have found `minTime = [1, 4, 1]` for the array `[X, Y, Z]` by some means (which will be explained in detail soon), then the sum of `minVal` is `1 * X + 4 * Y + 1 * Z`. We don't need to know exactly which array holds which value as the minimum, but only the number of times each number is taken as the minimum!

Now the task becomes finding `minTime[i]` for each index `i`.

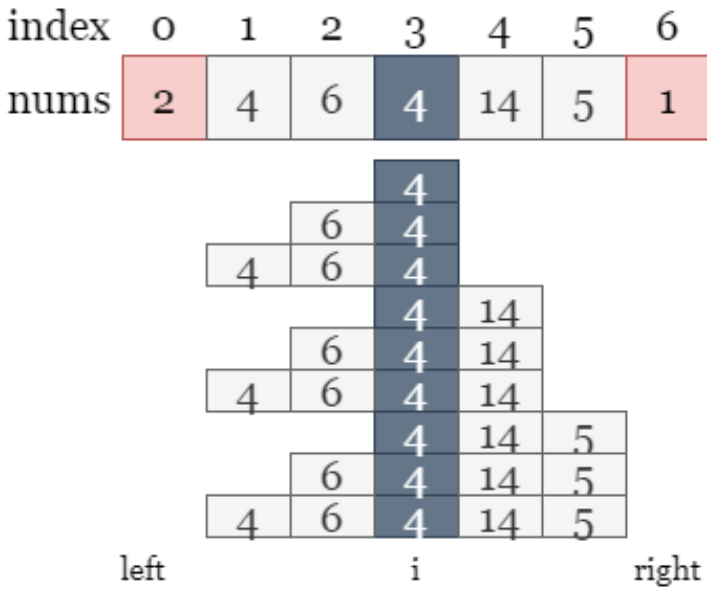
Notice that `minTime[i]` depends on:

- The number of consecutive elements **larger than or equal to** `nums[i]` on its left side. In other words, to find the index `left` where the value is **less than** `nums[i]`.
- The number of consecutive elements **larger than or equal to** `nums[i]` on its right side. In other words, to find the index `right` where the value is **strictly less than** `nums[i]`.

Now we have (i - left) positions to put the starting position of the subarray, and (right - i) positions to put the ending position of the subarray. Therefore, we have (i - left) * (right - i) valid subarrays in total, so we can calculate `minTime[i]` as follows:

$$\text{minTime}[i] = (\text{right} - i) \cdot (i - \text{left})$$
$$\text{range}_i = \text{minTime}[i] \cdot \text{nums}[i]$$

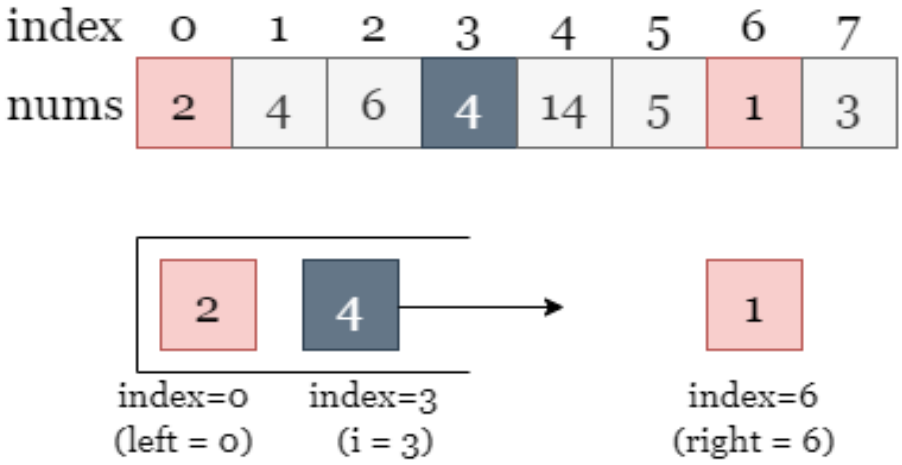
In the array shown below, `nums[3] = 4` has `left = 0` and `right = 6`, thus the number of subarrays having `nums[3]` as the minimum is `minTime[3] = (6 - 3) * (3 - 0) = 9`, meaning that there are 9 subarays having `nums[3]` as the minimum.



To calculate `minTime[i]` for every index, we can use a stack to maintain a monotonically increasing sequence during the iteration over `nums`:

- What is the left index `left`? The element on `nums[i]`'s left in the stack.
- What is the right index `right`? The element we are using to pop `nums[i]` from the stack.

In other words, `minTime[i]` is not calculated when we add `nums[i]` to the stack, but when we **pop** `nums[i]` from the stack, because only then are the left and right indexes clear to us. Then we can calculate `minTime[i]` using: $\text{minTime}[i]=(\text{right}-i)\cdot(i-\text{left})$. As shown in the picture below, when we encounter `nums[6] = 1`, we should pop `nums[3] = 4` from the stack, which is the time to calculate `minTime[3]`.

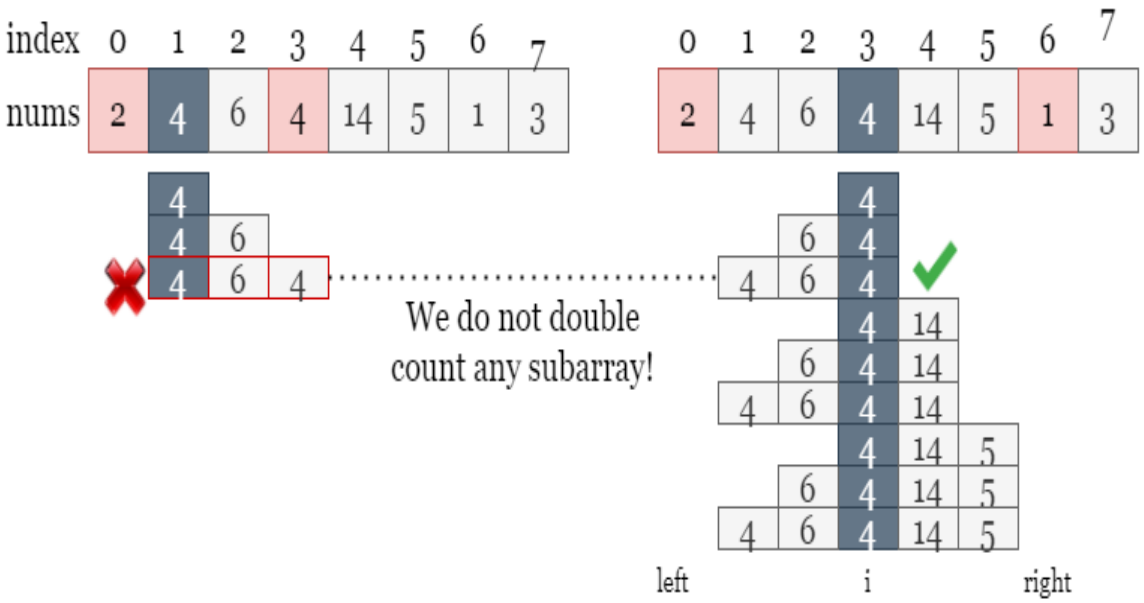


How to handle the edge cases?

- If the stack is empty after we pop `nums[i]` from it, we can't find the any index as the left boundary, so we set the left index as `-1`, which means that all the numbers on `nums[i]`'s left are within the range `[left, i]`.
- In order to pop the remaining elements from the stack after the iteration over `nums` stops, we set the right boundaries of all the remaining elements as `n`, which means that all the numbers on `nums[i]`'s right are within the range `[i, right]`. That's why we iterate from `i = 0` to `i = n`: to use `i = n` as the right boundary index to pop all the remaining elements from the stack.

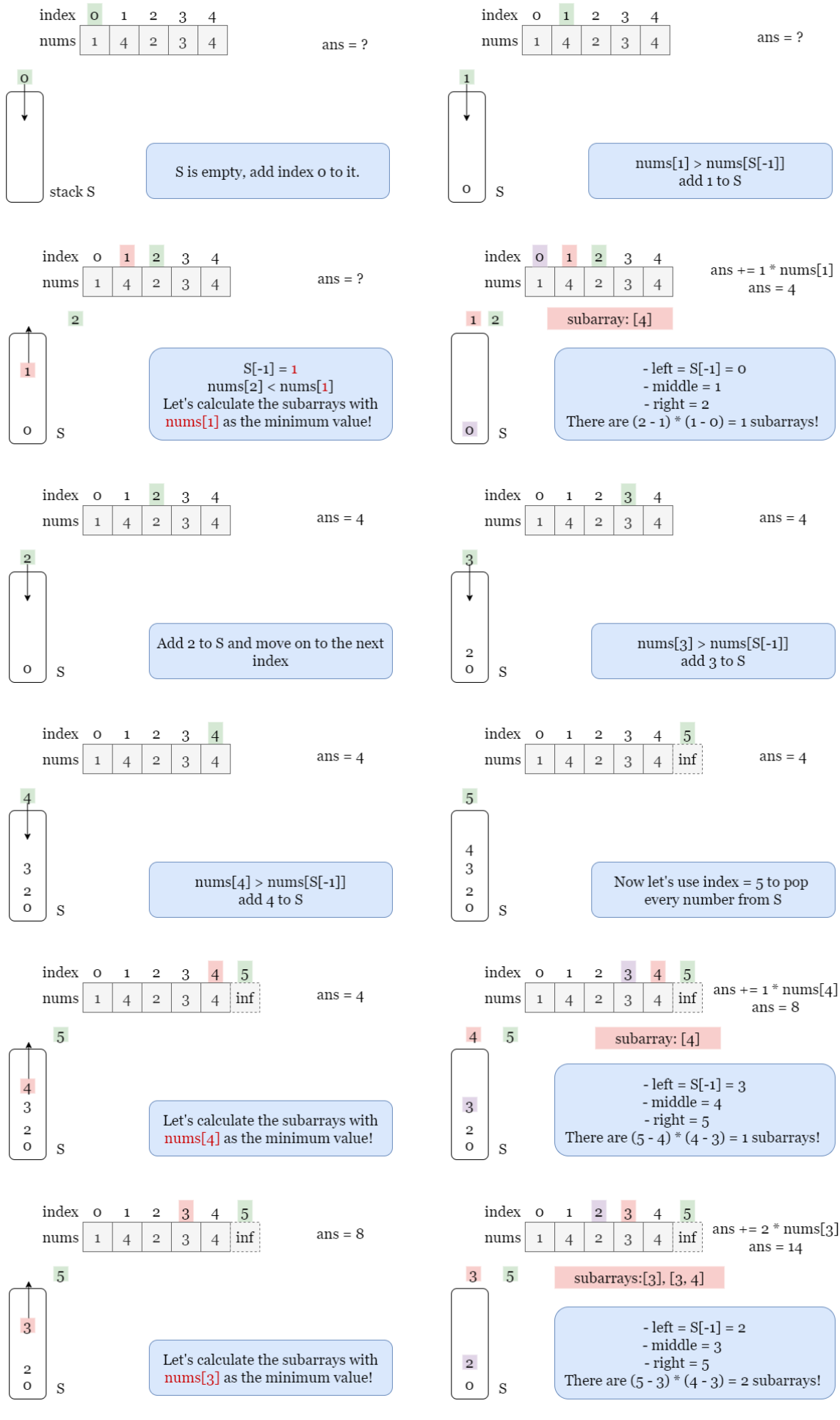
Will there by any duplicated calculation?

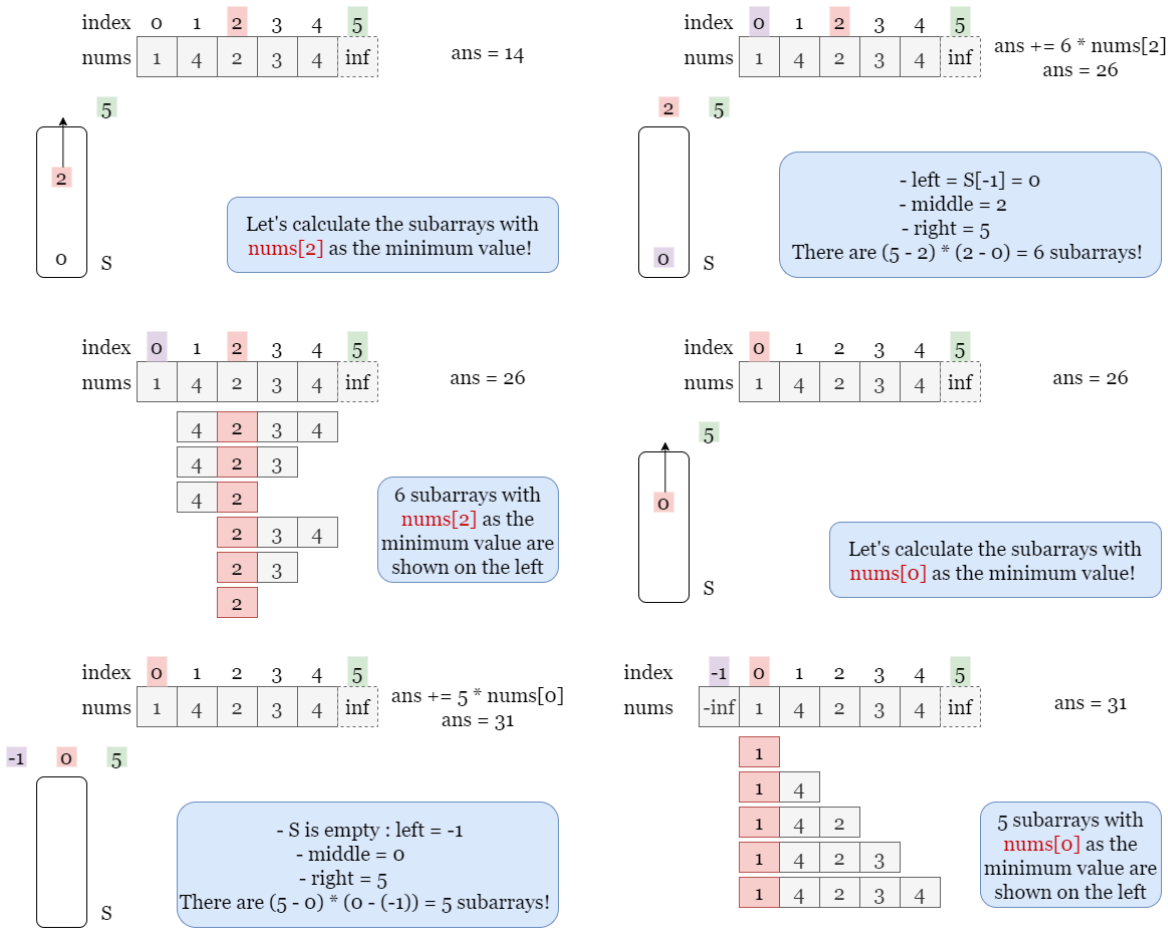
One might think, what if there are identical values that are close or adjacent, do we double count any subarray? The answer is NO! Although several identical values `A` may be adjacent to each other, the subarrays of the previous `A` will never take the following `A` as their minimum. As shown in the picture below, subarrays using the first `4` as the minimum don't cross the second `4`, thus we won't double count any subarray!



With each subproblem solved, we can move on to the results!

Please take the following slides as an example of getting the total sum of `minVal`.


index 0 1 2 3 4
nums 1 4 2 3 4
ans = ?
stack S
S is empty, add index 0 to it.
nums[1] > nums[S[-1]]
add 1 to S
S[-1] = 1
nums[2] < nums[1]
Let's calculate the subarrays with nums[1] as the minimum value!
ans += 1 * nums[1]
ans = 4
subarray: [4]
- left = S[-1] = 0
- middle = 1
- right = 2
There are (2 - 1) * (1 - 0) = 1 subarrays!
Add 2 to S and move on to the next index
nums[3] > nums[S[-1]]
add 3 to S
nums[4] > nums[S[-1]]
add 4 to S
Now let's use index = 5 to pop every number from S
Let's calculate the subarrays with nums[4] as the minimum value!
ans += 1 * nums[4]
ans = 8
- left = S[-1] = 3
- middle = 4
- right = 5
There are (5 - 4) * (4 - 3) = 1 subarrays!
Let's calculate the subarrays with nums[3] as the minimum value!
ans += 2 * nums[3]
ans = 14
subarrays:[3], [3, 4]
- left = S[-1] = 2
- middle = 3
- right = 5
There are (5 - 3) * (4 - 3) = 2 subarrays!



Note that this iteration is to get the sum of `minVal`. We also need to find the sum of `maxVal` in a similar way, by reversing the comparison condition, then get the sum of ranges using the first equation in this chapter. The job is done!

> If you are not much familiar with stack, we suggest you read our Leetcode Explore Card and have some knowledge of it beforehand.

Algorithm

1. Initialize an empty stack `stack`, get the size of `nums` as `n`.
2. Iterate over every index from `0` to `n` (inclusive). For each index `right`, if either of the following two condition is met:
   - `index = n`
   - `stack` is not empty and `nums[mid] >= nums[right]`, where `mid` is its top value:

   go to step 3.
   Otherwise, repeat step 2.

3. Calculate the number of subarrays with `nums[mid]` as its minimum value:
    - Pop `mid` from stack.
    - If `stack` is empty, set `left = -1`, otherwise, `left` equals the top element from `stack`.
    - Increment `answer` by `(right - mid) * (mid - left)`.
    - Repeat step 2.

Implementation

```
class Solution {
    public long subArrayRanges(int[] nums) {
        int n = nums.length;
        long answer = 0;
        Stack<Integer> stack = new Stack<>();

        // Find the sum of all the minimum.
        for (int right = 0; right <= n; ++right) {
            while (
                !stack.isEmpty() &&
                (right == n || nums[stack.peek()] >= nums[right])
            ) {
                int mid = stack.peek();
                stack.pop();
                int left = stack.isEmpty() ? -1 : stack.peek();
                answer -= (long) nums[mid] * (right - mid) * (mid - left);
            }
            stack.add(right);
        }

        // Find the sum of all the maximum.
        stack.clear();
```

```
        for (int right = 0; right <= n; ++right) {
            while (
                !stack.isEmpty() &&
                (right == n || nums[stack.peek()] <= nums[right])
            ) {
                int mid = stack.peek();
                stack.pop();
                int left = stack.isEmpty() ? -1 : stack.peek();
                answer += (long) nums[mid] * (right - mid) * (mid - left);
            }
            stack.add(right);
        }
        return answer;
    }
}
```

Complexity Analysis

Let n be the size of the input array `nums`.

- Time complexity: $O(n)$
    - To find the total sum of `minVal`, we only need one iteration over `nums`, and each number will be added to and popped from `stack` once, these also apply for finding `maxVal`.
    - Therefore the overall time complexity is $O(n)$.
- Space complexity: $O(n)$
    - We use a (monotonic) stack to keep the increasing (decreasing) sequence, in the worst-case scenario, there may be $O(n)$ numbers in the stack, which takes $O(n)$ space.