<https://leetcode.com/problems/sum-of-subarray-minimums/description/>

Given an array of integers arr, find the sum of min(b), where b ranges over every (contiguous) subarray of arr. Since the answer may be large, return the answer **modulo** 10^9 + 7.

**Example 1:**

Input: arr = [3,1,2,4]

Output: 17

Explanation:

Subarrays are [3], [1], [2], [4], [3,1], [1,2], [2,4], [3,1,2], [1,2,4], [3,1,2,4].

Minimums are 3, 1, 2, 4, 1, 1, 2, 1, 1, 1.

Sum is 17.

**Example 2:**

Input: arr = [11,81,94,43,3]

Output: 444

**Constraints:**

1 <= arr.length <= 3 \* 10^4

1 <= arr[i] <= 3 \* 10^4

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

**Solution 1: Monotonic Increasing Stack (360 min)**

class Solution {

    public int sumSubarrayMins(int[] arr) {

        int mod = (int)1e9 + 7;

        int n = arr.length;

        int[] left = new int[n];

        int[] right = new int[n];

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

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

        for(int i = 0; i < n; i++) {

            // Use ">=" to deal with duplicate elements, non-strict less

            while(!ple.isEmpty() && ple.peek()[0] >= arr[i]) {

                ple.pop();

            }

            // Why i + 1 when stack empty ?

            // When ple is empty, which means there is no previous less

            // element for arr[i], in this case, we set left[i]=i + 1 by

            // default. For example [7,8,4,3], there is no PLE for element

            // 4, so left[2]=2 + 1=3. How many subarrays with 4(A[2]) being

            // its minimum value? It's left[2] \* right[2]=3 \* 1

            left[i] = ple.isEmpty() ? i + 1 : i - ple.peek()[1];

            ple.push(new int[] {arr[i], i});

        }

        for(int i = n - 1; i >= 0; i--) {

            // Use strict less than ">" to deal with same value elements

            /\*\*

            arr={71,55,82,55}

            ------------------

            wrong answer 703

            no less than (>=) on both ple (ple.peek()[0] >= arr[i]) and

            nle (nle.peek()[0] >= arr[i])

            left = {1,2,1,4}

            right= {1,3,1,1}

            ------------------

            correct answer 593

            no less than (>=) on ple (ple.peek()[0] >= arr[i]) and strict

            less (>) on nle (nle.peek()[0] > arr[i])

            left = {1,2,1,4}

            right= {1,2,1,1}

            ------------------

            the redundant contribution 703 - 593 = 110 is caused by no less

            than (>=) also in nle, instead should be only strict less (>) in

            nle, the correct nums[1] = 55's contribution should be 2 \* 2 \* 55,

            but in wrong answer is 2 \* 3 \* 55, the difference is 55 \* 2 = 110

            \*/

            while(!nle.isEmpty() && nle.peek()[0] > arr[i]) {

                nle.pop();

            }

            right[i] = nle.isEmpty() ? n - i : nle.peek()[1] - i;

            nle.push(new int[] {arr[i], i});

        }

        long sum = 0;

        for(int i = 0; i < n; i++) {

            sum = (sum + (long)arr[i] \* left[i] \* right[i]) % mod;

        }

        return (int)sum;

    }

}

**Refer to**

<https://leetcode.com/problems/sum-of-subarray-minimums/solutions/178876/stack-solution-with-very-detailed-explanation-step-by-step/>

Before diving into the solution, we first introduce a very important stack type, which is called **monotone stack** .

**What is Monotonic Increasing Stack?**

Roughly speaking, the elements in the an monotonous increase stack keeps an increasing order.

**The typical paradigm for Monotonic Increasing Stack:**

for(int i = 0; i < A.size(); i++){

  while(!in\_stk.empty() && in\_stk.top() > A[i]){

    in\_stk.pop();

  }

  in\_stk.push(A[i]);

}

**What can Monotonic Increasing Stack do?**

**(1) find the previous less element of each element in a vector with O(n) time:**

What is the previous less element of an element?

For example:

[3, 7, 8, 4]

The previous less element of 7 is 3.

The previous less element of 8 is 7.

The previous less element of 4 is 3.

There is no previous less element for 3.

For simplicity of notation, we use abbreviation **PLE** to denote **P**revious **L**ess **E**lement.

C++ code (by slightly modifying the paradigm):

Instead of directly pushing the element itself, here for simplicity, we push the **index**.

We do some record when the index is pushed into the stack.

// previous\_less[i] = j means A[j] is the previous less element of A[i].

// previous\_less[i] = -1 means there is no previous less element of A[i].

vector<int> previous\_less(A.size(), -1);

for(int i = 0; i < A.size(); i++){

  while(!in\_stk.empty() && A[in\_stk.top()] > A[i]){

    in\_stk.pop();

  }

  previous\_less[i] = in\_stk.empty()? -1: in\_stk.top();

  in\_stk.push(i);

}

**(2) find the next less element of each element in a vector with O(n) time:**

What is the next less element of an element?

For example:

[3, 7, 8, 4]

The next less element of 8 is 4.

The next less element of 7 is 4.

There is no next less element for 3 and 4.

For simplicity of notation, we use abbreviation **NLE** to denote **N**ext **L**ess **E**lement.

C++ code (by slightly modifying the paradigm):

We do some record when the index is popped out from the stack.

// next\_less[i] = j means A[j] is the next less element of A[i].

// next\_less[i] = -1 means there is no next less element of A[i].

vector<int> previous\_less(A.size(), -1);

for(int i = 0; i < A.size(); i++){

  while(!in\_stk.empty() && A[in\_stk.top()] > A[i]){

    auto x = in\_stk.top(); in\_stk.pop();

    next\_less[x] = i;

  }

  in\_stk.push(i);

}

**How can the Monotonic Increasing Stack be applied to this problem?**

For example: Consider the element 3 in the following vector:

                            [2, 9, 7, 8, 3, 4, 6, 1]

    |                    |

            the previous less      the next less

                element of 3          element of 3

After finding both **NLE** and **PLE** of 3, we can determine the distance between 3 and 2(previous less) , and the distance between 3 and 1(next less).

In this example, the distance is 4 and 3 respectively.

**How many subarrays with 3 being its minimum value?**

The answer is 4\*3.

9 7 8 3

9 7 8 3 4

9 7 8 3 4 6

7 8 3

7 8 3 4

7 8 3 4 6

8 3

8 3 4

8 3 4 6

3

3 4

3 4 6

**How much the element 3 contributes to the final answer?**

It is 3\*(4\*3).

**What is the final answer?**

Denote by left[i] the distance between element A[i] and its **PLE**.

Denote by right[i] the distance between element A[i] and its **NLE**.

The final answer is, sum(A[i]\*left[i]\*right[i])

**The solution (One pass)**

class Solution {

public:

  int sumSubarrayMins(vector<int>& A) {

    stack<pair<int, int>> in\_stk\_p, in\_stk\_n;

    // left is for the distance to previous less element

    // right is for the distance to next less element

    vector<int> left(A.size()), right(A.size());

    //initialize

    for(int i = 0; i < A.size(); i++) left[i] =  i + 1;

    for(int i = 0; i < A.size(); i++) right[i] = A.size() - i;

    for(int i = 0; i < A.size(); i++){

      // for previous less

      while(!in\_stk\_p.empty() && in\_stk\_p.top().first > A[i]) in\_stk\_p.pop();

      left[i] = in\_stk\_p.empty()? i + 1: i - in\_stk\_p.top().second;

      in\_stk\_p.push({A[i],i});

      // for next less

      while(!in\_stk\_n.empty() && in\_stk\_n.top().first > A[i]){

        auto x = in\_stk\_n.top();in\_stk\_n.pop();

        right[x.second] = i - x.second;

      }

      in\_stk\_n.push({A[i], i});

    }

    int ans = 0, mod = 1e9 +7;

    for(int i = 0; i < A.size(); i++){

      ans = (ans + A[i]\*left[i]\*right[i])%mod;

    }

    return ans;

  }

};

Another version, since both distance arrays will be updated in the for loop, we don't need to initialize it, here's my java version:

class Solution {

    public int sumSubarrayMins(int[] A) {

      // initialize previous less element and next less element of

      // each element in the array

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

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

        int[] leftDistance = new int[A.length];

        int[] rightDistance = new int[A.length];

        for(int i=0; i<A.length; i++)

        {

            // use ">=" to deal with duplicate elements

            while(!previousLess.isEmpty() && previousLess.peek()[0] >= A[i])

            {

                previousLess.pop();

            }

            leftDistance[i] = previousLess.isEmpty() ? i+1 : i - previousLess.peek()[1];

            previousLess.push(new int[]{A[i], i});

        }

        for(int i=A.length-1; i>=0; i--)

        {

            while(!nextLess.isEmpty() && nextLess.peek()[0] > A[i])

            {

                nextLess.pop();

            }

            rightDistance[i] = nextLess.isEmpty() ? A.length - i : nextLess.peek()[1] - i;

            nextLess.push(new int[]{A[i], i});

        }

        int ans = 0;

        int mod = 1000000007;

        for(int i=0; i<A.length; i++)

        {

            ans = (ans + A[i] \* leftDistance[i] \* rightDistance[i]) % mod;

        }

        return ans;

    }

}

**The last thing that needs to be mentioned for handling duplicate elements:**

**Method**: Set **strict less** and **non-strict less**(less than **or equal to**) for finding **NLE** and **PLE** respectively. The order doesn't matter.

For example, the above code for finding **NLE** is **strict less**, while **PLE** is actually **non-strict less**.

**Remark**: Although in both loop conditions the signs are set as >, for NLE, we make records **inside** the loop, while for PLE, records are done **outside** the loop.

Subarrays divide by subarrays length from 1 to 4

len1 subarrays = {71},{55},{82},{55} -> 71,55,82,55

len2 subarrays = {71,55},{55,82},{82,55} -> 55,55,55

len3 subarrays = {71,55,82},{55,82,55} -> 55,55

len4 subarrays = {71,55,82,55} -> 55

55 \* 8 + 71 + 82 = 593

In correct solution we have

arr = {71,55,82,55}

left = {1,2,1,4}

right= {1,2,1,1}

71 \* 1 \* 1 + 55 \* 2 \* 2 + 82 \* 1 \* 1 + 55 \* 4 \* 1 = 593

In wrong solution we have

arr = {71,55,82,55}

left = {1,2,1,4}

right= {1,3,1,1}

71 \* 1 \* 1 + 55 \* 2 \* 3 + 82 \* 1 \* 1 + 55 \* 4 \* 1 = 703

The difference is at 55 \* 2 calculate one more time since

in 'nle' using incorrect '>=' rather than correct '>',

which

cause arr[1] = 55's next less element wrongly catch as

arr[3] = 55 rather than no less element after arr[1] at all.

The duplicate happening because in 'ple' process, since we

use non-strictly relation as '>=' to find previous less

element,

so for arr[3] = 55's previous less element we find

arr[1] = 55 based on '>=', so if in 'nle' process, we still

use

non-strictly relation as '>=' to find next less element,

and for arr[1] = 55's next less element we find arr[3] = 55

based on '>=', the second time build relation between arr[1]

and arr[3] will reflect on right[2] = 3 as wrong count instead

of correct count 2. To remove second time redundant found,

we have to change either one of two non-strictly relation

as '>=' into one strictly relation as '>'.

**More:**

What can monotonous **decrease** stack do?

Some applications of monotone (increase/decrease) stack in Leetcode:

[Next Greater Element II](https://leetcode.com/problems/Next-Greater-Element-II/description/) (a very basic one)

[Largest Rectangle in Histogram](https://leetcode.com/problems/Largest-Rectangle-in-Histogram/description/)(almost the same as this problem)

[Maximal Rectangle](https://leetcode.com/problems/Maximal-Rectangle/description/)(please do this problem after you solve the above one)

[Trapping Rain Water](https://leetcode.com/problems/Trapping-Rain-Water/description/) (challenge)

[Remove Duplicate Letters](https://leetcode.com/problems/remove-duplicate-letters/description/)(challenge)

[Remove K Digits](https://leetcode.com/problems/remove-k-digits/description/)

[Create Maximum Number](https://leetcode.com/problems/create-maximum-number/description/)

[132 Pattern](https://leetcode.com/problems/132-pattern/description/)(challenge, instead of focusing on the elements in the stack, this problem focuses on the elements poped from the monotone stack)

[sliding window maximum](https://leetcode.com/problems/sliding-window-maximum/description/)(challenge, monotone **queue**)

[Max Chunks To Make Sorted II](https://leetcode.com/problems/Max-Chunks-To-Make-Sorted-II/description/)

**Refer to**

<https://leetcode.com/problems/sum-of-subarray-minimums/solutions/2118729/very-detailed-stack-explanation-o-n-images-comments/>

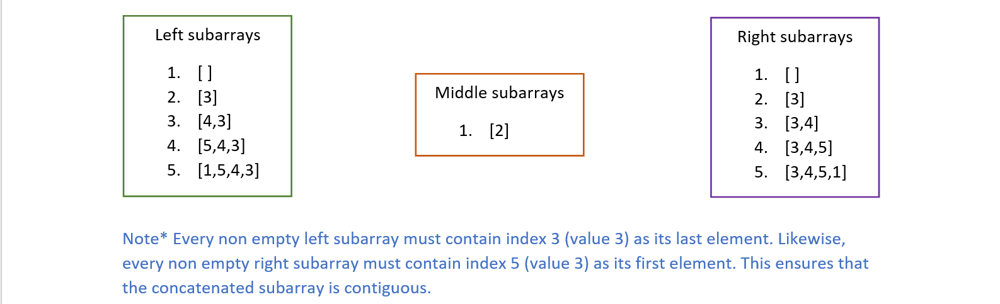
**Monotonic stack** A function is said to be monotonic if it preserves a given order. A monotonically increasing function never decreases. Likewise, a monotonically decreasing function never increases. Similarly, a monotonic stack contains elements that preserve a given order. A monotone increasing stack contains elements that never decrease. Likewise, a monotone decreasing stack contains elements that never increase.

[1,1,1,2,3,4,5,6,7,8,8,9,9,9,10] Monotone Increasing stack

[10,10,9,8,7,6,5,4,4,3,2,1,1,1]  Monotone Decreasing stack

image.png

Consider index 4 with a value of 2. To obtain a contiguous subarray containing element 2, a subarray must be selected from the left, middle, and right subarrays. The concatenated subarrays give a contiguous subarray containing element 2.



Mathematically the total number of subarrays containing 2 can be obtained as follows:

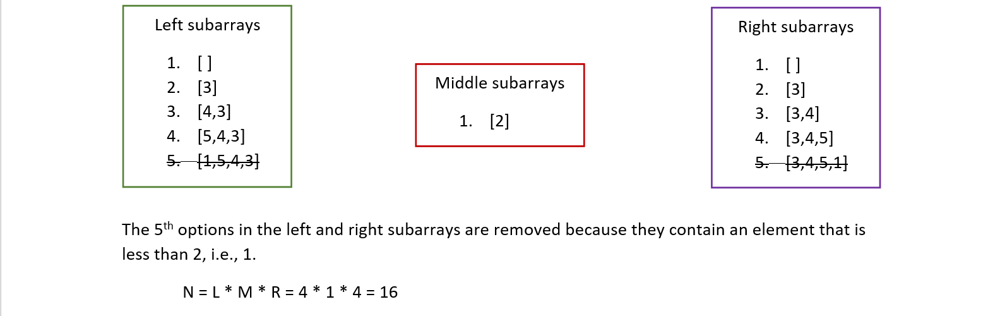
Let L = total number of left subarrays = 5

Let M = total number of middle subarrays = 1

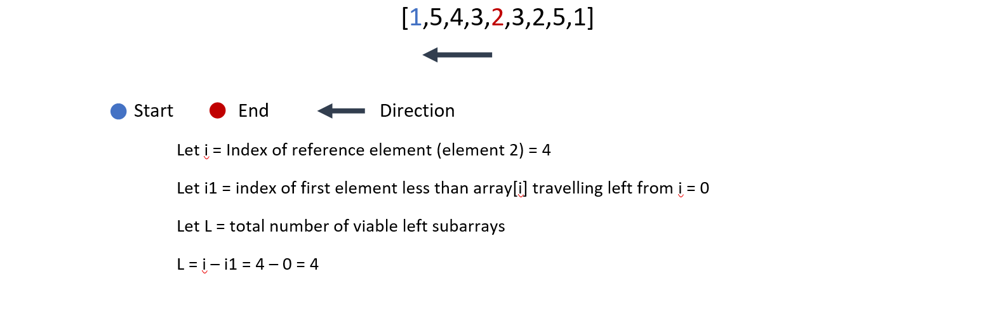
Let R = total number of right subarrays = 5

N = Total number of subarrays containing element in middle subarray N = L \* M \* R = 5 \* 1 \* 5 = 25

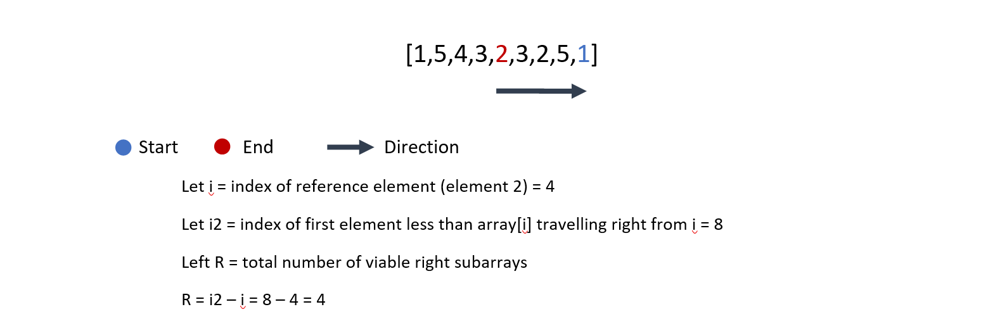
If we want to obtain subarrays with 2 as the minimum element the options to the left and right reduce as follows.



If we start from index 4 (with a value 2) and travel to the left of the array, the first index with a value less than 2 can be used to calculate the total number of viable left subarrays.



Likewise, we can calculate the number of viable right subarrays by travelling to the right of index 4 (element 2)



N = The total number of subarrays with 2 as the minimum = L \* 1 \* R = 4 \* 1 \* 4 = 16

We can obtain the indices i1 and i2 for every given index i in O(n) time using a monotonic stack.

**Maintaining a monotone increasing stack** To maintain a monotone increasing stack, we check if the value of the current index in the array is less than the top of the stack. While true, we continue to pop the top of the stack until we can push the current value.

while len(stack) and array[i] < array[stack[-1]]:

stack.pop()

stack.append(i)

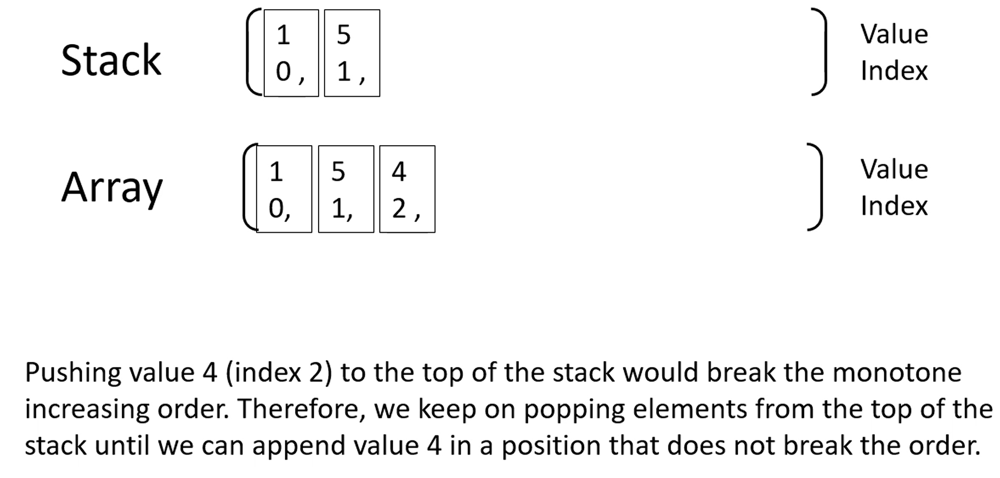
Note\*

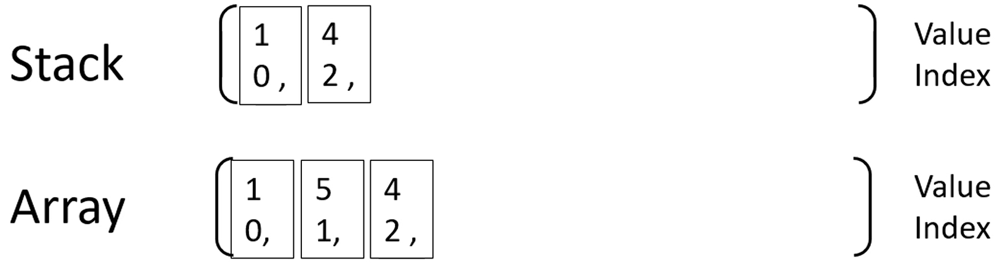
The first lesser value to the right of any index is the value that pops it off the stack. (i2)

The first lesser value to the left of any index is the first value to the left of that index in the stack. (i1)

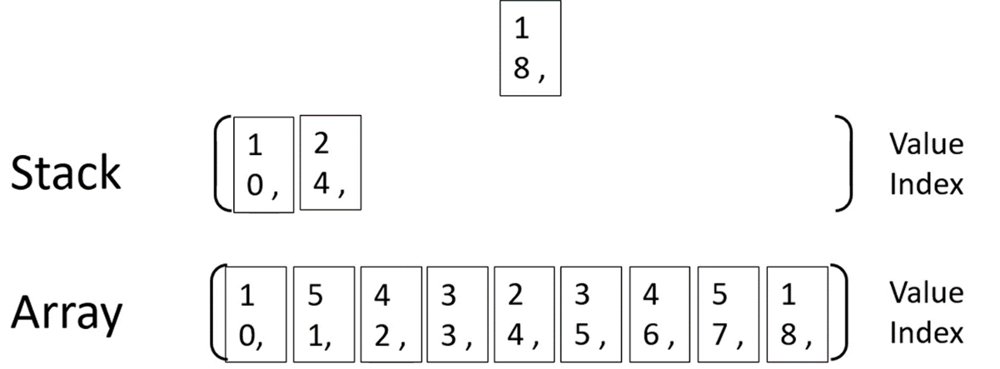
If we have the following array [1,2,3,4,5,6,7] we would never enter the while loop. A sentinel value less than every element in the array is pushed to the top of the array to ensure that we pop every element off the stack. Therefore, obtaining i1 and i2 for every index i in the array.

Suppose we traverse the array up until index 2 (value 4)





Suppose we maintain a monotone increasing stack until we reach the following state



The value 1 at index 8 is about to pop value 2 at index 4 off the stack.

Reference index i = 4 = stack.pop( ), i1 = Stack[-1] = 0, i2 = 8

L = i – i1 = 4 – 0 = 4

R = i2 – i = 8 – 4 = 4

N = 4 \* 4 = 16

Using the above logic, we can calculate to number of times every index in the array was a minimum value in a subarray. Therefore, we can find the sum of all subarray minimums.

class Solution {

    public int sumSubarrayMins(int[] arr) {

        long res = 0;

        Stack<Integer> stack = new Stack<Integer>();

        long M = (long)1e9 + 7;

        stack.push(-1);

        for (int i2 = 0; i2 < arr.length+1; i2++){

            int currVal = (i2<arr.length)? arr[i2] : 0;

            while(stack.peek() !=-1 && currVal < arr[stack.peek()]){

                int index = stack.pop();

                int i1 = stack.peek();

                int left = index - i1;

                int right = i2 - index;

                long add = (long)(left \* right \* (long)arr[index]) % M;

                res += add ;

                res %= M;

            }

            stack.push(i2);

        }

        return (int)res;

    }

}

**Refer to**

[L2281.Sum of Total Strength of Wizards (Ref.L84,L907,L828)](note://WEB4d6d3e6a86ec6eb31f53096b54e62fc5)

[L2104.Sum of Subarray Ranges (Ref.L907)](note://WEB99caea932d57e4abbe84eccbc2674a6c)