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

You are given an integer array nums. The range of a subarray of nums is the difference between the largest and smallest element in the subarray.

Return the sum of all subarray ranges of nums.

A subarray is a contiguous non-empty sequence of elements within an array.

**Example 1:**

**Input:** nums = [1,2,3]

**Output:** 4

**Explanation:** The 6 subarrays of nums are the following:

[1], range = largest - smallest = 1 - 1 = 0

[2], range = 2 - 2 = 0

[3], range = 3 - 3 = 0

[1,2], range = 2 - 1 = 1

[2,3], range = 3 - 2 = 1

[1,2,3], range = 3 - 1 = 2

So the sum of all ranges is 0 + 0 + 0 + 1 + 1 + 2 = 4.

**Example 2:**

**Input:** nums = [1,3,3]

**Output:** 4

**Explanation:** The 6 subarrays of nums are the following:

[1], range = largest - smallest = 1 - 1 = 0

[3], range = 3 - 3 = 0

[3], range = 3 - 3 = 0

[1,3], range = 3 - 1 = 2

[3,3], range = 3 - 3 = 0

[1,3,3], range = 3 - 1 = 2

So the sum of all ranges is 0 + 0 + 0 + 2 + 0 + 2 = 4.

**Example 3:**

**Input:** nums = [4,-2,-3,4,1]

**Output:** 59

**Explanation:** The sum of all subarray ranges of nums is 59.

**Constraints:**

1 <= nums.length <= 1000

-10^9 <= nums[i] <= 10^9

**Follow-up:** Could you find a solution with O(n) time complexity?

**Attempt 1: 2024-10-07**

**Wrong Solution**

**Error out on below input:**

nums = [-37988,-14446,-34454,-85916,44628,-63469,2405,76071,43291,499,-43933,-10950,

22587,45756,36078,49794,81866,-70327,80649,19025,82130,-53646,99394,63520,20667,41291,

80388,-82451,-17666,52744,-84498,30104,41847,67932,-89959,-42134,-79079,80796,-27089,

9691,-26248,-31934,-20681,33506,16422,-98706,-16321,847,55516,-85834,-3479,-58562,

77791,62111,-15830,33478,79046,-47470,-54997,-56231,11301,3998,73631,47168,66983,98655,

-31405,-11411,50967,-15908,37346,73429,-95644,83331,74868,-23201,70451,73304,38820,

-32124,80413,-23607,65237,88536,29905,-35443,-36683,64419,-25056,73050,17960,16070,

54748,76597,74972,-73098,74704,55261,-38420,-42739,15098,-8078,82487,-34954,-38895,

39994,35077,-36851,87932,7216,-87758,-27817,66742,77803,-16270,41596,-14558,28610,4151,

-2590,-73414,56156,93465,31128,-19581,-44840,-87553,-79674,-2016,3190,62008]

Output = 537100887

Expected = 1537100894

**Wrong Solution with % mod**

class Solution {

public long subArrayRanges(int[] nums) {

int mod = (int)1e9 + 7;

int n = nums.length;

Stack<int[]> ple = new Stack<>(); // Previous less element

Stack<int[]> nle = new Stack<>(); // Next less element

Stack<int[]> pme = new Stack<>(); // Previous more element

Stack<int[]> nme = new Stack<>(); // Next more element

int[] leftLess = new int[n];

int[] rightLess = new int[n];

int[] leftMore = new int[n];

int[] rightMore = new int[n];

for(int i = 0; i < n; i++) {

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

ple.pop();

}

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

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

}

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

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

nle.pop();

}

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

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

}

for(int i = 0; i < n; i++) {

while(!pme.isEmpty() && pme.peek()[0] <= nums[i]) {

pme.pop();

}

leftMore[i] = pme.isEmpty() ? i + 1 : i - pme.peek()[1];

pme.push(new int[]{nums[i], i});

}

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

while(!nme.isEmpty() && nme.peek()[0] < nums[i]) {

nme.pop();

}

rightMore[i] = nme.isEmpty() ? n - i : nme.peek()[1] - i;

nme.push(new int[]{nums[i], i});

}

long result = 0;

for(int i = 0; i < n; i++) {

result = (result + (long) nums[i] \* (leftMore[i] \* rightMore[i] - leftLess[i] \* rightLess[i]) % mod) % mod;

}

return result;

}

}

**The approach seems to be based on calculating the contribution of each element as both the minimum and the maximum in subarrays using stacks. However, the issue likely arises from the use of modulo in places where it may cause unintended results, especially because of negative numbers. In the problem statement, there’s no requirement to use modulo operations.**

**Solution 1: Correct Solution without % mod (30 min, very similar to L907)**

class Solution {

public long subArrayRanges(int[] nums) {

int n = nums.length;

Stack<int[]> ple = new Stack<>(); // Previous less element

Stack<int[]> nle = new Stack<>(); // Next less element

Stack<int[]> pme = new Stack<>(); // Previous more element

Stack<int[]> nme = new Stack<>(); // Next more element

int[] leftLess = new int[n]; // Distance between previous less element and current element

int[] rightLess = new int[n]; // Distance between next less element and current element

int[] leftMore = new int[n]; // Distance between previous more element and current element

int[] rightMore = new int[n]; // Distance between next more element and current element

// Monotonic Increasing Stack

// Previous less element

for (int i = 0; i < n; i++) {

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

ple.pop();

}

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

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

}

// Next less element

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

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

nle.pop();

}

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

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

}

// Monotonic Decreasing Stack

// Previous more element

for (int i = 0; i < n; i++) {

while (!pme.isEmpty() && pme.peek()[0] <= nums[i]) {

pme.pop();

}

leftMore[i] = pme.isEmpty() ? i + 1 : i - pme.peek()[1];

pme.push(new int[]{nums[i], i});

}

// Next more element

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

while (!nme.isEmpty() && nme.peek()[0] < nums[i]) {

nme.pop();

}

rightMore[i] = nme.isEmpty() ? n - i : nme.peek()[1] - i;

nme.push(new int[]{nums[i], i});

}

// No need % mod

long result = 0;

for (int i = 0; i < n; i++) {

result += (long) nums[i] \* (leftMore[i] \* rightMore[i] - leftLess[i] \* rightLess[i]);

}

return result;

}

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to**

<https://leetcode.com/problems/sum-of-subarray-ranges/solutions/1626628/o-n-solution-with-monotonous-stack-full-explaination/>

**Approach 1: Simple O(n^2) solution**

class Solution {

public:

long long subArrayRanges(vector<int>& nums) {

int n=nums.size();

long long res=0;

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

int maxi=nums[i], mini=nums[i];

for(int j=i+1;j<n;j++){

if(nums[j]>maxi)maxi=nums[j];

else if(nums[j]<mini)mini=nums[j];

res+=maxi-mini;

}

}

return res;

}

};

**Approach 2: O(n) solution**

**What is monotonous increase stack?**

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

**What can monotonous increase 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.

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

**(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.

**How can the monotonous increase stack be applied to this problem?**

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

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

(i=0)| (j=4)| (k=7)|

m=4-0=4 n=7-4=3

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)= (j-i) = 4 =m, and the distance between 3 and 1(next less)= (k-j) = 3=n.

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

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

No. of subarrays with 3 being its minimum value= m \* n = (j-i) \* (k-j) =4 \* 3=12

Here, these are-

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

**Proof of multiplication give us the number of sub-arrays**

The max array length with 3 as min element has m + n - 1 elements

[m={9, 7, 8, 3}, n={3, 4, 6}] => {9, 7, 8, 3, 4, 6 } = A (with m + n - 1 elements)

The number of subarray we have for the array A with length m + n - 1 is

=1 + 2 + 3 + 4 + ... + m + n - 1 = Summation(1 +...+ (m + n - 1)),

=(m + n - 1) (m + n) / 2

Since 3 should be the minimum number, we need to subtract number of subarrays which do not contain 3, which are subarrays of [9,7,8] with length (m - 1) and [4,6] with length (n - 1),

number of subarrays for

[9,7,8] is S1 = (m - 1 + 1)(m - 1) / 2 = m(m - 1) / 2

[4,6] is S2 = n(n - 1) / 2

Finally, we have

S3 - S2 - S1 = (m + n - 1)(m + n)/2 - m(m - 1)/2 - n(n - 1)/2

= (n^2 + mn - n + mn + m^2 - m - n^2 + n - m^2 + m) / 2

= (2 \* m \* n) / 2

= m \* n

, which is left distance multiply right distance.

**How much the element 3 contributes to the sum of the minimum element of all subarrays?**

It is 3\*(4\*3)

In general each index i in the array contributes in the sum of the minimum element of all subarrays-

(A[i] \* left[i] \* right[i])

where -

left[i] =the distance between element A[i] and its PLE.

right[i] =the distance between element A[i] and its NLE.

For better understanding of monotonous stak you may refer-

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

**With the same method, we can also find the sum of the maximum element of all subarrays.**

The solution for this problem can be formulated as sum(max(b)) - sum(min(b)), where b ranges over every (contiguous) subarray of n.

Final Solution-

long long subArrayRanges(vector<int>& n) {

return sumSubarrayComp(n, less<int>()) - sumSubarrayComp(n, greater<int>());

}

long long sumSubarrayComp(vector<int>& n, function<bool (int, int)> comp) {

long long res = 0;

vector<int> s;

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

while (!s.empty() && (i == n.size() || comp(n[s.back()], n[i]))) {

int j = s.back(), k = s.size() < 2 ? -1 : s[s.size() - 2];

res += (long long)(i - j) \* (j - k) \* n[j];

s.pop\_back();

}

s.push\_back(i);

}

return res;

}

or

class Solution {

public:

long long subArrayRanges(vector<int>& nums) {

int n=nums.size();

long long sum=0;

stack<int>st;

vector<int> minPrev(n,-1),minNext(n,n),maxPrev(n,-1),maxNext(n,n);

for(int i=0;i<n;i++)

{

while(!st.empty()&&nums[st.top()]>=nums[i]){st.pop();}

if(!st.empty()){minPrev[i]=st.top();}

st.push(i);

}

while(!st.empty()){st.pop();}

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

{

while(!st.empty()&&nums[st.top()]>nums[i]){st.pop();}

if(!st.empty()){minNext[i]=st.top();}

st.push(i);

}

for(int i=0;i<n;i++)

{

while(!st.empty()&&nums[st.top()]<=nums[i]){st.pop();}

if(!st.empty()){maxPrev[i]=st.top();}

st.push(i);

}

while(!st.empty()){st.pop();}

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

{

while(!st.empty()&&nums[st.top()]<nums[i]){st.pop();}

if(!st.empty()){maxNext[i]=st.top();}

st.push(i);

}

for(int i=0;i<n;i++)

{

long long leftMin=i-minPrev[i],rightMin=minNext[i]-i;

long long leftMax=i-maxPrev[i],rightMax=maxNext[i]-i;

sum+=(leftMax\*rightMax-leftMin\*rightMin)\*nums[i];

}

return sum;

}

};

Here's the Java Version

class Solution {

public long subArrayRanges(int[] nums) {

int n = nums.length;

long sum=0;

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

int[] minPrev = new int[n];

int[] minNext = new int[n];

int[] maxPrev = new int[n];

int[] maxNext = new int[n];

Arrays.fill(minPrev , -1);

Arrays.fill(minNext , n);

Arrays.fill(maxPrev , -1 );

Arrays.fill(maxNext , n);

for(int i=0;i<n;i++){

while(!st.isEmpty() && nums[st.peek()] >= nums[i]){

st.pop();

}

if(!st.isEmpty()){

minPrev[i] = st.peek();

}

st.push(i);

}

st = new Stack<>();

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

while(!st.isEmpty() && nums[st.peek()] > nums[i]){

st.pop();

}

if(!st.isEmpty()){

minNext[i] = st.peek();

}

st.push(i);

}

st = new Stack<>();

for(int i=0 ;i<n ;i++){

while(!st.isEmpty() && nums[st.peek()] <= nums[i]){

st.pop();

}

if(!st.isEmpty()){

maxPrev[i] = st.peek();

}

st.push(i);

}

st = new Stack<>();

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

while(!st.isEmpty() && nums[st.peek()] < nums[i]){

st.pop();

}

if(!st.isEmpty()){

maxNext[i] = st.peek();

}

st.push(i);

}

for(int i=0 ;i <n;i++){

long leftMin = i-minPrev[i];

long rightMin = minNext[i]-i;

long leftMax = i-maxPrev[i];

long rightMax = maxNext[i]-i;

sum+= (leftMax\*rightMax - leftMin\*rightMin)\*nums[i];

}

return sum;

}

}

**Refer to**

[L907.Sum of Subarray Minimums (Ref.L2281,L2104)](note://427AAC8865B14CB4BD06E36CBE243A89)