<https://leetcode.com/problems/sum-of-total-strength-of-wizards/description/>

As the ruler of a kingdom, you have an army of wizards at your command.

You are given a **0-indexed** integer array strength, where strength[i] denotes the strength of the ith wizard. For a **contiguous** group of wizards (i.e. the wizards' strengths form a **subarray** of strength), the **total strength** is defined as the product of the following two values:

The strength of the weakest wizard in the group.

The **total** of all the individual strengths of the wizards in the group.

Return the **sum** of the total strengths of **all** contiguous groups of wizards. Since the answer may be very large, return it modulo 10^9 + 7.

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

**Example 1:**

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

**Output:** 44

**Explanation:** The following are all the contiguous groups of wizards:

- [1] from [**1**,3,1,2] has a total strength of min([1]) \* sum([1]) = 1 \* 1 = 1

- [3] from [1,**3**,1,2] has a total strength of min([3]) \* sum([3]) = 3 \* 3 = 9

- [1] from [1,3,**1**,2] has a total strength of min([1]) \* sum([1]) = 1 \* 1 = 1

- [2] from [1,3,1,**2**] has a total strength of min([2]) \* sum([2]) = 2 \* 2 = 4

- [1,3] from [**1,3**,1,2] has a total strength of min([1,3]) \* sum([1,3]) = 1 \* 4 = 4

- [3,1] from [1,**3,1**,2] has a total strength of min([3,1]) \* sum([3,1]) = 1 \* 4 = 4

- [1,2] from [1,3,**1,2**] has a total strength of min([1,2]) \* sum([1,2]) = 1 \* 3 = 3

- [1,3,1] from [**1,3,1**,2] has a total strength of min([1,3,1]) \* sum([1,3,1]) = 1 \* 5 = 5

- [3,1,2] from [1,**3,1,2**] has a total strength of min([3,1,2]) \* sum([3,1,2]) = 1 \* 6 = 6

- [1,3,1,2] from [**1,3,1,2**] has a total strength of min([1,3,1,2]) \* sum([1,3,1,2]) = 1 \* 7 = 7

The sum of all the total strengths is 1 + 9 + 1 + 4 + 4 + 4 + 3 + 5 + 6 + 7 = 44.

**Example 2:**

**Input:** strength = [5,4,6]

**Output:** 213

**Explanation:** The following are all the contiguous groups of wizards:

- [5] from [**5**,4,6] has a total strength of min([5]) \* sum([5]) = 5 \* 5 = 25

- [4] from [5,**4**,6] has a total strength of min([4]) \* sum([4]) = 4 \* 4 = 16

- [6] from [5,4,**6**] has a total strength of min([6]) \* sum([6]) = 6 \* 6 = 36

- [5,4] from [**5,4**,6] has a total strength of min([5,4]) \* sum([5,4]) = 4 \* 9 = 36

- [4,6] from [5,**4,6**] has a total strength of min([4,6]) \* sum([4,6]) = 4 \* 10 = 40

- [5,4,6] from [**5,4,6**] has a total strength of min([5,4,6]) \* sum([5,4,6]) = 4 \* 15 = 60

The sum of all the total strengths is 25 + 16 + 36 + 36 + 40 + 60 = 213.

**Constraints:**

1 <= strength.length <= 10^5

1 <= strength[i] <= 10^9

**Attempt 1: 2024-08-22**

**Solution 1: Prefix Sum + Monotonic Increasing Stack (720 min)**

class Solution {

    public int totalStrength(int[] strength) {

        int n = strength.length;

        int MOD = (int)1e9 + 7;

        // Sum of first k elements

        int[] presum = new int[n + 1];

        for(int i = 0; i < n; i++) {

            presum[i + 1] = (presum[i] + strength[i]) % MOD;

        }

        // Sum of first k prefix

        int[] preSumOfPresum = new int[n + 2];

        for(int i = 0; i <= n; i++) {

            preSumOfPresum[i + 1] = (preSumOfPresum[i] + presum[i]) % MOD;

        }

        // First index on the left strictly less than(<) current strength[i]

        int[] left = new int[n];

        Arrays.fill(left, -1);

        // Monotonic increasing stack to calculate left bounds

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

        for(int i = 0; i < n; i++) {

            while(!stack.isEmpty() && strength[stack.peek()] >= strength[i]) {

                stack.pop();

            }

            if(!stack.isEmpty()) {

                left[i] = stack.peek();

            }

            stack.push(i);

        }

        stack.clear();

        // First index on the right less than or equal to(<=) current strength[i]

        // The reason we use < on left but <= on right is to avoid duplicates

        int[] right = new int[n];

        Arrays.fill(right, n);

        // Monotonic increasing stack to calculate right bounds

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

            while(!stack.isEmpty() && strength[stack.peek()] > strength[i]) {

                stack.pop();

            }

            if(!stack.isEmpty()) {

                right[i] = stack.peek();

            }

            stack.push(i);

        }

        // Calculate the total strength

        long result = 0;

        for(int i = 0; i < n; i++) {

            // positive parts:

            // sumRight \* (i - left[i]) = (presum[i + 1] + presum[i + 2] + ... + presum[right[i]]) \* (i - left[i])

            // negative parts:

            // sumLeft \* (right[i] - i) = (presum[left[i] + 1] + presum[left[i] + 2] + ... + presum[i]) \* (right[i] - i)

            long sumRight = (preSumOfPresum[right[i] + 1] - preSumOfPresum[i + 1] + MOD) % MOD;

            long sumLeft = (preSumOfPresum[i + 1] - preSumOfPresum[left[i] + 1] + MOD) % MOD;

            long positive\_parts = sumRight \* (i - left[i]) % MOD;

            long negative\_parts = sumLeft \* (right[i] - i) % MOD;

            // Why add 'MOD \* 2' ?

            // It is to prevent the subtraction to result into a negative number

            // Assume -> (preSumOfPresum[right[i] + 1] - preSumOfPresum[i + 1]) \* (i - left[i]) turns out to be MOD +1

            // and on taking % MOD the result become "1"

            // So incase the second part (preSumOfPresum[i + 1] - preSumOfPresum[left[i] + 1]) \* (right[i] - i) is a

            // number smaller than MOD then the subtraction of first part and second part would lead to answer

            // being negative, added 2 \* MOD to prevent this

            result += (positive\_parts - negative\_parts + 2 \* MOD) % MOD \* strength[i] % MOD;

            result %= MOD;

        }

        return (int) result;

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to**

<https://leetcode.com/problems/sum-of-total-strength-of-wizards/solutions/2062017/c-prefix-monotonic-stack-o-n-solution-with-thought-process/>

**Most tricky part before involving math:**

For each strength[i], we could find a non-empty index range (left, right) where strength[i] is the min value. So for all subarrays in this range **including** strength[i], the total strength is strength[i] \* the sum of those subarray sums.

left is the first index on the left side i where strength[left] < strength[i]

right is the first index on the right side of i where strength[right] <= strength[i]

These two indices can be pre-calculated using **monotonic stack** (example: LC496. Next Greater Element I).

The reason we use < on left but <= on right is to avoid duplicates.

Here is an example array: 1 **2 3 4 2** 3 4 2 1

For the highlighted subarray 2 3 4 2, we want to calculate the strength using the 2nd 2 but not the first 2.

**How do we get the "sum of all subarrays including strength[i] in range (left, right)" (left and right index inclusive)?**

Let's list the indices:

...left-1, left, **left + 1, left + 2, ... i-1, i, i+1, ... right-1**, right, right+1...

**Let prefix[i] be the prefix sum of first i elements in strength. The prefix array build up 1-index offset based on given input strength array.**

// sum of first k elements

vector<long long> prefix(N + 1, 0L);

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

prefix[i + 1] = (prefix[i] + st[i]) % MOD;

}

The sum of subarrays **including** **i** are:

the subarrays that start with **left+1**:

sum(left+1, ... i) = prefix[i + 1] - prefix[left + 1]

sum(left+1, ... i+1) = prefix[i + 2] - prefix[left + 1]

...

sum(left+1, ... right-1) = prefix[right] - prefix[left + 1]

the subarrays that start with **left+2**:

sum(left+2, ... i) = prefix[i + 1] - prefix[left + 2]

sum(left+2, ... i+1) = prefix[i + 2] - prefix[left + 2]

...

sum(left+2, ... right-1) = prefix[right] - prefix[left + 2]

......

subarrays sets continue .....

......

the subarrays that start with **i**:

sum(i, ... i) = prefix[i + 1] - prefix[i]

sum(i, ... i+1) = prefix[i + 2] - prefix[i]

...

sum(i, ... right-1) = prefix[right] - prefix[i]

Then we combine all above terms, we have:

**positive** parts:

(prefix[i + 1] + prefix[i + 2] + ... + prefix[right]) \* (i - left)

**Note: (i - left) because the subarray sets start from (left + 1) till i, total as (i - (left + 1) + 1) = (i - left) sets**

**negative** parts:

(prefix[left + 1] + prefix[left + 2] + ... + prefix[i]) \* (right - i)

**Note: (i - left) because the subarray sets start from i till (right - 1), total as (right - 1 - i + 1) = (right - i) sets**

The range sum of prefix can be optimized by pre-compute prefix-sum of prefix.

Time complexity: O(N): we have 5 passes of the input array length

Space complexity: O(N): two prefix arrays and a stack (vector) is used

**Note: additional verfication for below:**

The sum of subarrays including i are:

the subarrays that start with **left+1**:

sum(left+1, ... i) = prefix[i + 1] - prefix[left + 1]

sum(left+1, ... i+1) = prefix[i + 2] - prefix[left + 1]

...

sum(left+1, ... right-1) = prefix[right] - prefix[left + 1]

**First need to clarify what "**sum(left+1, ... i)**" means here:**

sum(left+1, ... i) = nums[left+1] + nums[left+2] + ... + nums[i], element at 'left+1' and 'i' are inclusive

sum(left+1, ... i+1) = nums[left+1] + nums[left+2] + ... + nums[i+1], element at 'left+1' and 'i+1' are inclusive

...

sum(left+1, ... right-1) = nums[left+1] + nums[left+2] + ... + nums[right-1], element at 'left+1' and 'right-1' are inclusive

**Step by step to validate the formula accuracy like
"sum(left+1, ... i) = prefix[i + 1] - prefix[left + 1]"**

Let prefix[i] be the prefix sum of first i elements in strength.

The prefix array build up 1-index offset based on given input strength array.

e.g

strength = 3,1,2,4

prefix = 0,3,4,6,10

left(index boundary) = -1,-1,1,2

right(index boundary) = 1,4,4,4

let's take example for i = 2, its corresponding boundary index is left = 1, right = 4

So the statement below:

"sum of all subarrays including strength[i] in range (left, right), left and right index inclusive"

equal to:

"sum of all subarrays including strength[2] in range (1, 4), 1 and 4 index inclusive"

further equal to:

"since inclusive, we just need to calculate sum of all subarrays build up by strength[2] and strength[3]"

The sum of subarrays including i(=2) are:

- the subarrays that start with left+1 (1+1 = 2):

(1)sum(left+1, ... i) = prefix[i + 1] - prefix[left + 1]

=>

sum(1+1, ... 2) = sum(2) means sum of elements in strength array at index 2 = strength[2] = prefix[2 + 1] - prefix[1 + 1] = prefix[3] - prefix[2] = 6 - 4 = 2

the result match expectation as strength[2] = 2

(2)sum(left+1, ... i+1) = prefix[i + 2] - prefix[left + 1]

=>

sum(1+1, ... 3) = sum(2,3) means sum of elements in strength array at index 2 and 3 = strength[2] + strength[3] = prefix[2 + 2] - prefix[1 + 1] = prefix[3] - prefix[2] = 10 - 4 = 6

the result match expectation as strength[2] + strength[3] = 2 + 4 = 6

(3)sum(left+1, ... right-1) = prefix[right] - prefix[left + 1]

=>

sum(1+1, ... 3) same as (2), no need re-calculate

The rest is implementation.

C++ code

int totalStrength(vector<int>& st) {

long long MOD = 1'000'000'007;

const int N = st.size();

// sum of first k elements

vector<long long> prefix(N + 1, 0L);

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

prefix[i + 1] = (prefix[i] + st[i]) % MOD;

}

// sum of first k prefix

vector<long long> prefix\_sum(N + 2, 0L);

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

prefix\_sum[i + 1] = (prefix\_sum[i] + prefix[i]) % MOD;

}

// first index on the left < current st

vector<int> left(N, -1);

// mono increase

vector<int> stack;

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

while (!stack.empty() && st[stack.back()] >= st[i]) {

stack.pop\_back();

}

left[i] = stack.empty() ? -1 : stack.back();

stack.push\_back(i);

}

// first index on the right <= current st

vector<int> right(N, N);

stack.clear();

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

while (!stack.empty() && st[stack.back()] > st[i]) {

stack.pop\_back();

}

right[i] = stack.empty() ? N : stack.back();

stack.push\_back(i);

}

long long res = 0;

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

// Why add 'MOD \* 2' ?

// It is to prevent the subtraction to result into a negative number.

// Assume -> (prefix\_sum[right[i] + 1] - prefix\_sum[i + 1]) \* (i - left[i]) turns out to be MOD +1

// and on taking % MOD the result become "1".

// So incase the second part (prefix\_sum[i + 1] - prefix\_sum[left[i] + 1]) \* (right[i] - i) is a

// number smaller than MOD then the subtraction of first part and second part would lead to answer

// being negative, added 2 \* MOD to prevent this.

res += ((prefix\_sum[right[i] + 1] - prefix\_sum[i + 1]) \* (i - left[i]) % MOD + MOD \* 2 -

(prefix\_sum[i + 1] - prefix\_sum[left[i] + 1]) \* (right[i] - i) % MOD) % MOD \* st[i] % MOD;

res %= MOD;

}

return (int) res;

}

**Java version:**

class Solution {

    public int totalStrength(int[] st) {

        long MOD = 1\_000\_000\_007;

        int N = st.length;

        // sum of first k elements

        long[] prefixSum = new long[N + 1];

        for (int i = 0; i < N; ++i) {

            prefixSum[i + 1] = (prefixSum[i] + st[i]) % MOD;

        }

        // sum of first k prefix

        long[] prefixSumOfPrefixSum = new long[N + 2];

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

            prefixSumOfPrefixSum[i + 1] = (prefixSumOfPrefixSum[i] + prefixSum[i]) % MOD;

        }

        // first index on the left < current st

        int[] left = new int[N];

        Arrays.fill(left, -1);

        Deque<Integer> stack = new ArrayDeque<>();

        // Mono-increasing stack to calculate left bounds

        for (int i = 0; i < N; ++i) {

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

                stack.pop();

            }

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

            stack.push(i);

        }

        // first index on the right <= current st

        int[] right = new int[N];

        Arrays.fill(right, N);

        stack.clear();

        // Mono-increasing stack to calculate right bounds

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

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

                stack.pop();

            }

            right[i] = stack.isEmpty() ? N : stack.peek();

            stack.push(i);

        }

        long res = 0;

        // Calculate the total strength

        for (int i = 0; i < N; ++i) {

            long sumRight = (prefixSumOfPrefixSum[right[i] + 1] - prefixSumOfPrefixSum[i + 1] + MOD) % MOD;

            long sumLeft = (prefixSumOfPrefixSum[i + 1] - prefixSumOfPrefixSum[left[i] + 1] + MOD) % MOD;

// Why add 'MOD \* 2' ?

// It is to prevent the subtraction to result into a negative number.

// Assume -> (prefix\_sum[right[i] + 1] - prefix\_sum[i + 1]) \* (i - left[i]) turns out to be MOD +1

// and on taking % MOD the result become "1".

// So incase the second part (prefix\_sum[i + 1] - prefix\_sum[left[i] + 1]) \* (right[i] - i) is a

// number smaller than MOD then the subtraction of first part and second part would lead to answer

// being negative, added 2 \* MOD to prevent this.

            res += (sumRight \* (i - left[i]) % MOD + MOD \* 2 - sumLeft \* (right[i] - i) % MOD) % MOD \* st[i] % MOD;

            res %= MOD;

        }

        return (int) res;

    }

}

**Refer to**

<https://algo.monster/liteproblems/2281>

**Problem Description**

In this problem, we are given an array strength representing the strength of wizards in a kingdom, indexed starting from 0. We need to compute the sum of the total strengths for all possible contiguous subarrays of wizards. The total strength of a specific group of contiguous wizards is the product of two factors:

The strength of the weakest wizard in the group.

The sum of strengths of all wizards in the group.

The task is to return the sum of total strengths of all such groups. However, since the sum may be huge, we must return the sum modulo (10^9 + 7).

A subarray is defined as a contiguous non-empty sequence from the array.

**Intuition behind the Solution**

The brute force approach to solving this problem would involve finding all possible subarrays and calculating their total strengths, but this would be inefficient for large arrays. To optimize, we can use a stack-based approach to find the bounds within which each wizard is the weakest.

We iterate through the strength array to find, for each wizard, the index of the next weaker wizard to the left and to the right. This helps us determine the range of subarrays where the current wizard would be the weakest and affect the total strength calculation.

Once we have these left and right bounds for each wizard, the next step is to calculate the sum of strengths for all subarrays ending at each position. To do this efficiently, we use prefix sums, which allow us to calculate sums over ranges in constant time.

Finally, we combine the above information to calculate the answer using the formula derived from the definition of total strength and our understanding of the provided bounds. Specifically, we find the sum of products of the strength of the current wizard (the weakest), the sum of strengths for subarrays ending at this wizard's position, and the count of such subarrays.

Let's break down the important steps of our solution:

We use stacks to find the left and right indices (bounds) where the strength[i] is the weakest in a contiguous group.

We calculate prefix sums twice over the array. First, to get the sum of elements up to each index, and then to get the sum of those sums. This helps to find the total sum of strengths in different ranges quickly.

We then calculate the answer using the derived indices and prefix sums, taking care to apply the modulo operation to handle large numbers.

With all these steps, we ensure efficient computation by avoiding repeated sum calculations and by determining bounds for each wizard efficiently.

These optimization techniques allow the solution to run significantly faster than the brute force method and are suitable for arrays of any size.

**Solution Approach**

The solution provided leverages a stack to efficiently determine the bounds of subarrays for which each wizard's strength is the limiting strength. Here is a step-by-step explanation of the implementation details:

**Initialization of Auxiliary Arrays:**

left: An array to store the index of the previous weaker wizard for each wizard.

right: An array to store the index of the next weaker wizard for each wizard.

stk: A stack that will help in finding the indices for the left and right bounds.

**Finding Left Bounds:**

Iterate through the strength array from left to right (increasing index).

For each wizard, pop elements from the stack while the top of the stack represents a wizard with strength greater than or equal to the current wizard, and then store the index of the last popped element as the previous weaker wizard.

If the stack is not empty after popping stronger wizards, the top element will be the index of the previous weaker wizard.

Add the current index to the stack for future comparisons.

**Finding Right Bounds:**

Iterate through the strength array from right to left (decreasing index).

The process is similar to finding the left bounds but in reverse order.

For each wizard, pop elements from the stack while the top of the stack represents a wizard with strength greater than the current wizard.

The index of the last popped element will be used to update the right bound for the current wizard.

Append the current wizard's index to the stack for future comparisons.

**Using Prefix Sums:**

Calculate the prefix sums for strength array using the accumulate function twice. The first accumulation gives us the running total of strengths, and the second accumulation gives us the running total of these running totals. This is stored in the ss array.

Prefix sums allow us to calculate the sum of any subarray in constant time.

**Computing the Answer:**

Iterate through each element in strength again.

For each wizard, calculate two sums using the prefix sums array ss:

a: The sum of strengths from the left bound to the current wizard.

b: The sum of strengths from the current wizard to the right bound.

These sums are adjusted according to the number of wizards included in the subarrays to the left and right of the current wizard (using l and r variables).

The difference (a - b) is weighted by the current wizard's strength (this represents the weakest strength in the subarray).

Take the modulus of the intermediate result to handle the large numbers and update the cumulative ans.

**Final Result:**

After iterating through all wizards, the ans variable holds the sum of the total strengths of all contiguous groups of wizards modulo 10^9 + 7.

To summarize, the algorithm uses a stack-based approach to find the bounds within which each wizard is the weakest, calculates prefix sums to efficiently manage the range sum queries, and combines this information to obtain the final sum of total strengths, while carrying out modular arithmetic operations to comply with the problem's requirement.

**Example Walkthrough**

Let's consider an array strength of wizard strengths given by: [3, 1, 2, 4]. The goal is to find the sum of total strengths for all possible contiguous subarrays.

**Initialization of Auxiliary Arrays:** We initialize left, right, and stk as empty.

**Finding Left Bounds:** We iterate through [3, 1, 2, 4] from left to right:

At strength[0] = 3, stk is empty, so we push 0. left[0] remains -1 (a convention to indicate no previous weaker wizard).

At strength[1] = 1, we pop 3 from stack because 1 is weaker than 3. Now, stk is empty, so we push 1. left[1] is set to -1.

At strength[2] = 2, stk has 1 and 2 is stronger, so we push 2. left[2] is 1.

At strength[3] = 4, stk has 1, 2 and since 2 is weaker, we just push 3. left[3] is 2.

Now, left is [-1, -1, 1, 2].

**Finding Right Bounds:** We do a similar process in reverse:

Start at strength[3] = 4, stk is empty, so we push 3. right[3] remains the length of the array, which is 4, indicating no next weaker wizard.

At strength[2] = 2, we pop 4 from stack because 2 is weaker than 4. We push 2. right[2] is 4.

At strength[1] = 1, nothing is popped because the stack is empty, so we push 1. right[1] is 4.

At strength[0] = 3, we pop 1 and 2 from the stack as 3 is stronger than both. We push 0. right[0] is 1.

Now, right is [1, 4, 4, 4].

**Using Prefix Sums:** We calculate prefix sums ss of strengths: [0, 3, 4, 6, 10] and the sum of sums: [0, 0, 3, 7, 13, 23].

**Computing the Answer:** We iterate again and calculate contributions to the answer:

For wizard at index 0, a = 3 \* (0 + 1), b = 7 \* (1 - 0), and strength is 3. The contribution is 3 \* (a - b) = 3 \* (3 - 7) = -12.

For wizard at index 1, a = 4 \* (1 + 1), b = 23 \* (4 - 1), and strength is 1. The contribution is 1 \* (a - b) = 1 \* (8 - 69) = -61.

For wizard at index 2, a = 6 \* (1 - (-1 + 1)), b = 10 \* (3 - 2), and strength is 2. The contribution is 2 \* (a - b) = 2 \* (6 - 10) = -8.

For wizard at index 3, a = 10 \* (3 - 2), b = 23 \* (4 - 3), and strength is 4. The contribution is 4 \* (a - b) = 4 \* (10 - 23) = -52.

The total strength is the sum of contributions: (-12) + (-61) + (-8) + (-52) = -133.

Note that the negative result occurs due to the way we calculate a - b. The actual contribution is always non-negative since it represents a sum of products of strengths. In practice, we would take each contribution modulo (10^9 + 7) to keep the results properly bounded.

**Final Result:** The algorithm would thus sum these contributions and apply the modulo to ensure they remain within the specific bounds, giving us the sum of the total strengths of all contiguous groups of wizards modulo (10^9 + 7).

By applying this process to larger arrays, we can find the desired sum while keeping the computational cost within manageable bounds, thus solving traditionally expensive problems both effectively and efficiently.

**Solution Implementation**

class Solution {

public int totalStrength(int[] strength) {

// Get the length of the strength array

int n = strength.length;

// Initialize arrays to store the nearest smaller elements' indices

int[] nearestSmallerLeftIndex = new int[n];

int[] nearestSmallerRightIndex = new int[n];

// Initialize these arrays with default values

Arrays.fill(nearestSmallerLeftIndex, -1);

Arrays.fill(nearestSmallerRightIndex, n);

Deque<Integer> stack = new ArrayDeque<>();

// Find the nearest smaller element on the left for each element

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

while (!stack.isEmpty() && strength[stack.peek()] >= strength[i]) {

stack.pop();

}

if (!stack.isEmpty()) {

nearestSmallerLeftIndex[i] = stack.peek();

}

stack.push(i);

}

// Clear the stack for the next iteration

stack.clear();

// Find the nearest smaller element on the right for each element

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

while (!stack.isEmpty() && strength[stack.peek()] > strength[i]) {

stack.pop();

}

if (!stack.isEmpty()) {

nearestSmallerRightIndex[i] = stack.peek();

}

stack.push(i);

}

// Specify the modulus value for the answer

int mod = (int) 1e9 + 7;

// Calculate the prefix sum of the strength array

int[] prefixSum = new int[n + 1];

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

prefixSum[i + 1] = (prefixSum[i] + strength[i]) % mod;

}

// Calculate the prefix sum of the prefix sum array

int[] prefixSumOfPrefixSum = new int[n + 2];

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

prefixSumOfPrefixSum[i + 1] = (prefixSumOfPrefixSum[i] + prefixSum[i]) % mod;

}

// Initialize the answer

long answer = 0;

// Calculate the total strength

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

int value = strength[i];

int left = nearestSmallerLeftIndex[i] + 1;

int right = nearestSmallerRightIndex[i] - 1;

long sumLeft = (long) (i - left + 1) \* (prefixSumOfPrefixSum[right + 2] - prefixSumOfPrefixSum[i + 1]);

long sumRight = (long) (right - i + 1) \* (prefixSumOfPrefixSum[i + 1] - prefixSumOfPrefixSum[left]);

answer = (answer + value \* ((sumLeft - sumRight) % mod)) % mod;

}

// Return the final total strength value, adjusted for mod

return (int) (answer + mod) % mod;

}

}

**Time and Space Complexity**

The given solution aims to find the total strength of non-empty contiguous subarrays of the strength array. To solve this problem efficiently, it uses the monotonic stack technique to precompute the next smaller elements on the left and right for every element.

**Time Complexity**

The first loop iterates over the array to find the next smaller element on the left for each element. This is done in O(n) time because each element is pushed onto and popped from the stack at most once.

The second loop, similar to the first one, iterates over the array to find the next smaller element on the right for each element, which also takes O(n) time.

The computation of the sum of subarray sums, ss, involves two cumulative sum computations, each of which takes O(n) time. Therefore, this step is also O(n).

Finally, the last loop calculates the total strength for each index, within O(n) time since each index operation is constant due to precomputed sums and smaller elements' indexes.

Considering the loops and operations are sequential, the overall time complexity is O(n) where n is the length of the strength array.

**Space Complexity**

The space complexity is primarily impacted by the additional data structures: left, right, stk, and ss. Both left and right arrays and the stk stack have a space complexity of O(n).

Similarly, the ss array, which holds the cumulative sums, has a space complexity of O(n).

The space complexity does not stack because these data structures do not depend on each other for space. Therefore, when taking all the auxiliary data structures into account, the resultant space complexity remains O(n), where n is the length of the strength array.

**Refer to**

[L84.Largest Rectangle in Histogram](note://E2FC22D9715540C5A8F06F60FBBE886F)

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

[L828.Count Unique Characters of All Substrings of a Given String (Ref.L2262)](note://WEBf394faa89bf4c60aa6f2c29c102ef307)