Sum of Subarray Ranges

Problem Statement: Given an integer array nums, determine the range of a subarray, defined as the difference between the largest and smallest elements within the subarray. Calculate and return the sum of all subarray ranges of nums.

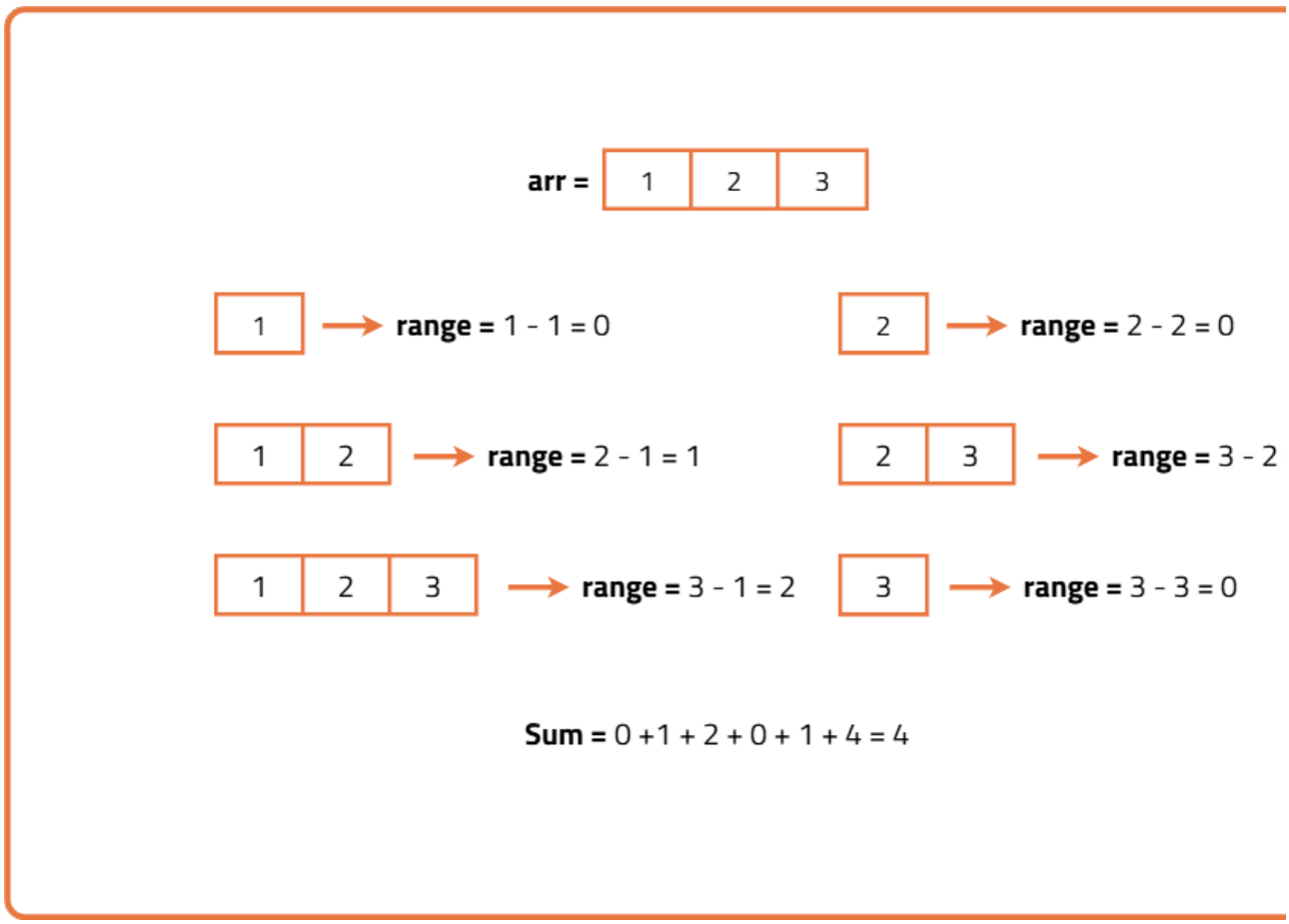
A subarray is defined as a contiguous, non-empty sequence of elements within the array.

Examples

```
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.
```

Brute Force
Algorithm

- Initialize a variable to store the total sum as 0
- Loop through the array to fix the starting index of the subarray
- Initialize two variables to track the minimum and maximum for the current subarray
- Use an inner loop to extend the subarray to the right
- Update the minimum and maximum values as the subarray grows
- Calculate the range as maximum minus minimum and add it to the total sum
- Repeat this for all possible subarrays
- Return the total sum after processing all subarrays



Code

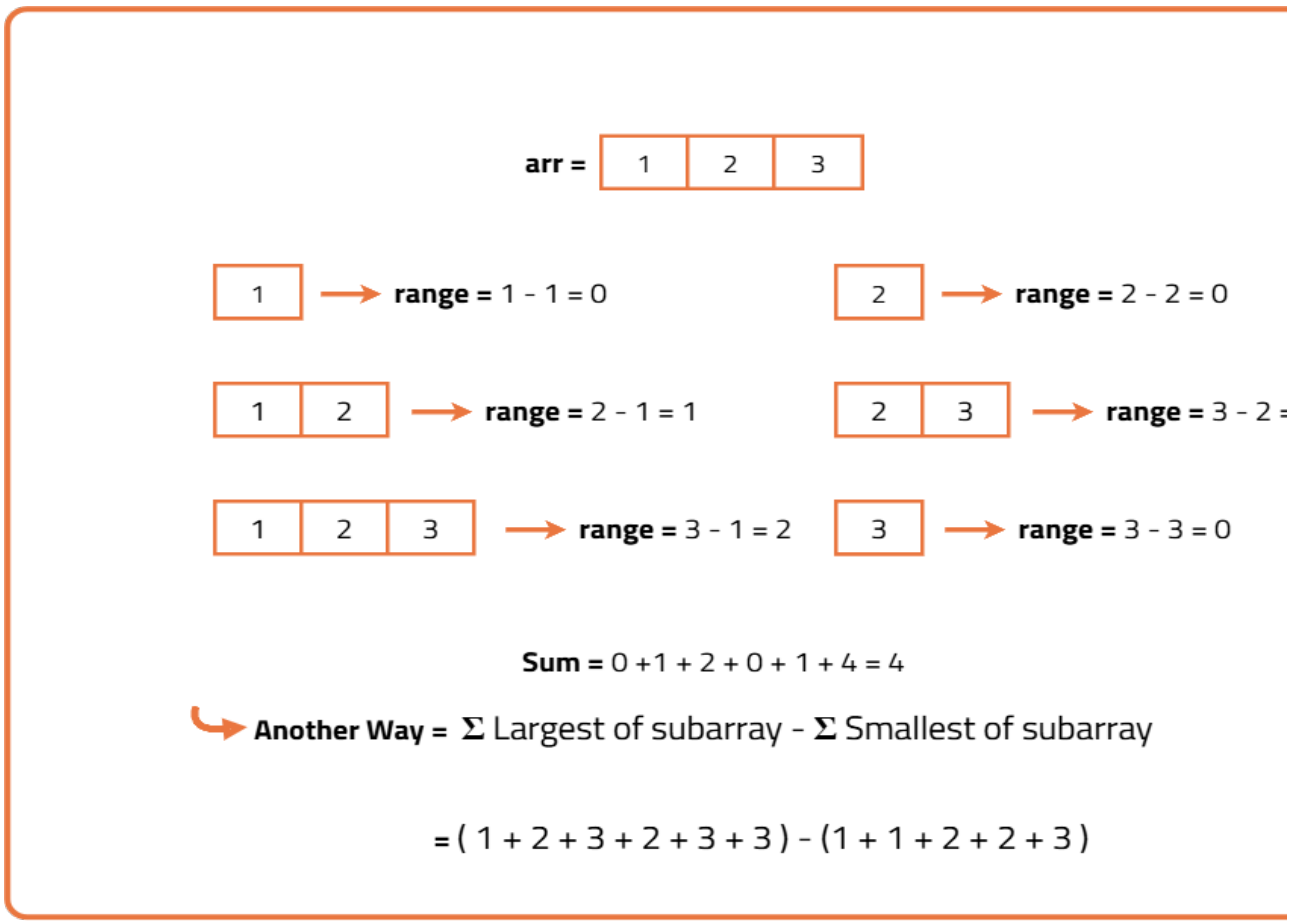
```
import java.util.*;

// Solution class with the core logic
class Solution {

    // Function to find the sum of subarray ranges in all subarrays
    public long subArrayRanges(int[] arr) {
        // Size of array
        int n = arr.length;

        // Variable to store the final sum
        long sum = 0;

        // Traverse each starting index of subarrays
        for (int i = 0; i < n; i++) {

            // Initialize smallest and largest for current subarray
            int smallest = arr[i];
            int largest = arr[i];

            // Traverse subarrays starting from i
            for (int j = i; j < n; j++) {
                // Update smallest element seen so far
                smallest = Math.min(smallest, arr[j]);

                // Update largest element seen so far
                largest = Math.max(largest, arr[j]);

                // Add the current range (max - min) to the total sum
                sum += (largest - smallest);
```

```
            }
        }

        // Return the computed total sum
        return sum;
    }
}

// Main class with the main() method
public class Main {
    public static void main(String[] args) {
        // Input array
        int[] arr = {1, 2, 3};

        // Create instance of Solution class
        Solution sol = new Solution();

        // Call the function to calculate sum of subarray ranges
        long ans = sol.subArrayRanges(arr);

        // Print the result
        System.out.println("The sum of subarray ranges is: " + ans);
    }
}
```

Complexity Analysis

Optimal Approach
Algorithm

- Find the index of the next smaller element to the right for each element using NSE logic
- Find the index of the previous smaller or equal element to the left using PSEE logic
- Find the index of the next greater element to the right for each element using NGE logic
- Find the index of the previous greater or equal element to the left using PGEE logic
- Use NSE and PSEE to calculate the total contribution of each element as the minimum in subarrays
- Use NGE and PGEE to calculate the total contribution of each element as the maximum in subarrays
- Subtract the total of subarray minimums from the total of subarray maximums
- Return the result as the sum of ranges of all subarrays



Code

```
import java.util.*;

// Solution class with all helper methods and final logic
class Solution {

    // Function to find indices of Next Smaller Elements
    private int[] findNSE(int[] arr) {
        int n = arr.length;
        int[] ans = new int[n];
        Stack<Integer> st = new Stack<>();
        for (int i = n - 1; i >= 0; i--) {
            while (!st.isEmpty() && arr[st.peek()] >= arr[i]) {
                st.pop();
            }
            ans[i] = !st.isEmpty() ? st.peek() : n;
            st.push(i);
        }
        return ans;
    }

    // Function to find indices of Next Greater Elements
    private int[] findNGE(int[] arr) {
        int n = arr.length;
        int[] ans = new int[n];
        Stack<Integer> st = new Stack<>();
        for (int i = n - 1; i >= 0; i--) {
            while (!st.isEmpty() && arr[st.peek()] <= arr[i]) {
                st.pop();
            }
            ans[i] = !st.isEmpty() ? st.peek() : n;
            st.push(i);
```

```
    }
    return ans;
}

// Function to find indices of Previous Smaller or Equal Elements
private int[] findPSEE(int[] arr) {
    int n = arr.length;
    int[] ans = new int[n];
    Stack<Integer> st = new Stack<>();
    for (int i = 0; i < n; i++) {
        while (!st.isEmpty() && arr[st.peek()] > arr[i]) {
            st.pop();
        }
        ans[i] = !st.isEmpty() ? st.peek() : -1;
        st.push(i);
    }
    return ans;
}

// Function to find indices of Previous Greater or Equal Elements
private int[] findPGEE(int[] arr) {
    int n = arr.length;
    int[] ans = new int[n];
    Stack<Integer> st = new Stack<>();
    for (int i = 0; i < n; i++) {
        while (!st.isEmpty() && arr[st.peek()] < arr[i]) {
            st.pop();
        }
        ans[i] = !st.isEmpty() ? st.peek() : -1;
        st.push(i);
    }
    return ans;
}

// Function to compute sum of subarray minimums
private long sumSubarrayMins(int[] arr) {
    int n = arr.length;
    int[] nse = findNSE(arr);
    int[] psee = findPSEE(arr);
    long sum = 0;
    for (int i = 0; i < n; i++) {
        int left = i - psee[i];
        int right = nse[i] - i;
        long freq = 1L * left * right;
        sum += freq * arr[i];
    }
    return sum;
}

// Function to compute sum of subarray maximums
private long sumSubarrayMaxs(int[] arr) {
    int n = arr.length;
    int[] nge = findNGE(arr);
    int[] pgee = findPGEE(arr);
    long sum = 0;
    for (int i = 0; i < n; i++) {
        int left = i - pgee[i];
        int right = nge[i] - i;
        long freq = 1L * left * right;
        sum += freq * arr[i];
    }
    return sum;
}
```

```
    // Function to compute total sum of subarray ranges (max - min)
    public long subArrayRanges(int[] arr) {
        return sumSubarrayMaxs(arr) - sumSubarrayMins(arr);
    }
}

// Separate class containing the main method
public class Main {
    public static void main(String[] args) {
        // Sample input
        int[] arr = {1, 2, 3};

        // Create an instance of Solution class
        Solution sol = new Solution();

        // Compute the sum of subarray ranges
        long ans = sol.subArrayRanges(arr);

        // Print the result
        System.out.println("The sum of subarray ranges is: " + ans);
    }
}
```

Complexity Analysis

Time Complexity: O(N), since calculating the sum of subarray maximums takes O(N) time and calculating the sum of subarray minimums takes O(N) time.

Space Complexity: O(N), since calculating the sum of subarray maximums requires O(N) space and calculating the sum of subarray minimums requires O(N) space.