1508. Range Sum of Sorted Subarray Sums

Sorting

Two Pointers Binary Search

Problem Description

Medium Array

continuous subarrays of the given array nums. The problem requires three inputs:

The given problem revolves around finding the sum of a specific subset of a sorted array that contains sums of all non-empty

1. nums - an array of n positive integers. 2. left - the starting index from which to sum (one-indexed).

• Sorting these sums means arranging them in non-decreasing order.

brute force method, being mindful of its limitations.

Use a nested loop to iterate through all possible subarrays:

• Modular arithmetic: to ensure the final sum stays within the specified limits.

To illustrate the solution approach, let's consider a small example:

• The outer loop runs with i from 0 to 2 (the length of nums minus one).

• The inner loop starts with j at i, with s initialized to 0 each time the outer loop starts.

3. right - the ending index up to which to sum (one-indexed). Our objective is to compute the sum of all non-empty continuous subarrays, sort these sums, and then calculate the sum of the

large numbers which could cause integer overflow situations. To clarify: • A non-empty continuous subarray is a sequence of one or more consecutive elements from the array.

elements from the left to right indices in the sorted array. The final answer should be returned modulo 10^9 + 7 to handle very

- Summing from index left to index right implies adding all elements of the sorted sums array starting at index left-1 and ending at index right-1 (since the problem statement indexes from 1, but arrays in most programming languages, including Python, are 0-indexed).

1. The outer loop goes through each element of nums from which a subarray can start.

Intuition The brute force approach to this problem is to generate all possible continuous subarrays of nums, calculate their sums, and then

sort these sums. This can be done by using two nested loops:

This generates an array arr of sums for each subarray. Once we have this array, we sort it. With the sorted array, we now only

need to sum the numbers from left-1 to right-1 since we're using Python's zero-indexing (the original problem indexes from 1).

2. The inner loop extends the subarray from the starting element from the outer loop to the end of nums, calculating the sum for each extension.

- This solution is intuitive but not optimized. It can solve the problem as long as the nums array is not too large since the time
- complexity would otherwise become prohibitive. For larger arrays, an optimized approach should be considered involving more advanced techniques such as prefix sums, heaps, or binary search. However, for the provided solution, we stick to the intuitive

Solution Approach The provided solution iterates through each possible subarray of the given array nums and maintains a running sum. To achieve this, the following steps are implemented:

∘ The outer loop, controlled by variable i, goes from 0 to n-1, where i represents the starting index of a subarray. • For each i, initialize a sum variable s to 0, to calculate the sum of the subarray starting at index i.

arr[left - 1 : right].

In summary, the approach uses:

Return the result.

∘ The inner loop, controlled by variable j, goes from i to n-1. In each iteration of this loop, add the value nums[j] to s. This effectively extends the subarray by one element in each iteration. After adding nums[j] to s, append the new sum to arr.

Once all sums are calculated, sort the list arr with the sort() method. This arranges all the subarray sums in non-decreasing

order. Calculate the sum of the elements from index left-1 to right-1 in the sorted list arr. This is done with the slice notation

Initialize an empty list arr that will hold the sums of all non-empty continuous subarrays.

- Compute the result modulo 10**9 + 7 to get the output within the specified range and handle any potential integer overflow. This is important since the sum of subarray sums could be very large.
- Brute force algorithm: to generate all subarray sums. • **Sorting**: to arrange the sums in non-decreasing order. • Prefix sum technique: by maintaining a running sum s as we iterate through the array.
- The time complexity of this approach is $0(n^2)$ for generating the subarray sums and 0((n*(n+1)/2) * log(n*(n+1)/2)) for sorting, where n is the length of the nums array. The space complexity is 0(n*(n+1)/2) since we are storing the sum of each

possible subarray in arr.

Example Walkthrough

Iteration breakdown:

For i = 1:

remains 5.

Python

from typing import List

for j in range(i, n):

subarray_sums.sort()

mod = 10**9 + 7

return range_sum

class Solution:

Initialize arr: We start with an empty list arr to hold the sums of all non-empty continuous subarrays. **Iterate using nested loops:**

 \circ j = 1:s = 1 + nums[1] (add 2), arr becomes [1, 3].

 \circ j = 2:s = 3 + nums[2] (add 3), arr becomes [1, 3, 6].

 \circ j = 1:s = 0 + nums[1] (add 2), arr becomes [1, 3, 6, 2].

 \circ j = 2:s = 0 + nums[2] (add 3), arr becomes [1, 3, 6, 2, 5, 3].

(index 2) elements in the sorted arr. This gives us the sum of 2 + 3 = 5.

def rangeSum(self, nums: List[int], n: int, left: int, right: int) -> int:

current_sum += nums[j] # Add the current element to the sum

subarray_sums.append(current_sum) # Append the current sum to the list

Note: The '-1' adjustment is required because list indices in Python are 0-based

// This method calculates the sum of values within a given range of subarray sums from nums array

currentSum += nums[j]; // Add the current number to the current subarray sum

subarraySums[index++] = currentSum; // Store the current subarray sum and increment index

Initialize an array to store the sum of contiguous subarrays

Sort the array of subarray sums in non-decreasing order

range_sum = sum(subarray_sums[left - 1 : right]) % mod

public int rangeSum(int[] nums, int n, int left, int right) {

// Initialize an array to store all possible subarray sums

int index = 0; // Index to insert the next sum into subarraySums

int currentSum = 0; // Holds the temporary sum of the current subarray

// Loop over nums array to create subarrays starting at index i

function rangeSum(nums: number[], n: number, left: number, right: number): number {

let currentSum = 0; // Stores the sum of the current subarray

def rangeSum(self, nums: List[int], n: int, left: int, right: int) -> int:

Initialize an array to store the sum of contiguous subarrays

Sort the array of subarray sums in non-decreasing order

range_sum = sum(subarray_sums[left - 1 : right]) % mod

Return the computed sum modulo 10^9 + 7

Define the modulus value to prevent integer overflow issues

Compute the sum of the elements from the 'left' to 'right' indices

let subarraySums: number[] = new Array(n * (n + 1) / 2);

let index = 0; // Index for inserting into subarraySums

let answer = 0; // Variable to store the final answer

// Calculate the sum of all possible subarrays

for (let end = start; end < n; ++end) {</pre>

for (let start = 0; start < n; ++start) {</pre>

// Sort the sums of the subarrays

subarraySums.sort((a, b) => a - b);

const MOD = 1e9 + 7; // The modulo value

for (let i = left - 1; i < right; ++i) {</pre>

// Calculate the number of possible subarrays and initialize an array to store their sums

currentSum += nums[end]; // Add the next element to the current sum

subarraySums[index++] = currentSum; // Store the sum of the subarray

// Calculate the sum of the subarray sums between indices left-1 and right-1 (inclusive)

answer = (answer + subarraySums[i]) % MOD; // Aggregate the sum modulo MOD

Calculate the sum of every contiguous subarray and store it into subarray_sums

subarray_sums.append(current_sum) # Append the current sum to the list

Note: The '-1' adjustment is required because list indices in Python are 0-based

current_sum += nums[j] # Add the current element to the sum

// Loop over nums array to define starting point of subarray

// Calculate the total number of subarray sums

int[] subarraySums = new int[totalSubarrays];

int totalSubarrays = n * (n + 1) / 2;

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

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

Return the computed sum modulo 10^9 + 7

Define the modulus value to prevent integer overflow issues

Compute the sum of the elements from the 'left' to 'right' indices

Sorting arr: We sort arr, resulting in [1, 2, 3, 3, 5, 6].

from the left to right indices, inclusive.

For i = 0: \circ j = 0:s = 0 + nums[0] (add 1), arr becomes [1].

Calculate the result: We sum the elements from left-1 to right-1, which corresponds to the second (index 1) and third

Modular arithmetic: The final step is to take the result modulo 10**9 + 7. Since 5 is already less than 10**9 + 7, the answer

The method returns the result 5, which is the sum of the second and third smallest sums of non-empty continuous subarrays of

Suppose we have an array nums = [1, 2, 3], with left = 2 and right = 3. We need to find the sum of the sorted subarray sums

- \circ j = 2:s = 2 + nums[2] (add 3), arr becomes [1, 3, 6, 2, 5]. For i = 2:
- nums. In this case, the sums of these subarrays are 2 for the subarray [2] and 3 for the subarrays [3] and [1, 2]. Since [3] appears twice, we only count it once in the specified range.
- Solution Implementation

This walkthrough summarizes the steps in the provided solution to compute the sum of subarray sums in a given range after

sorting them, thereby displaying the complete process from initialization through to obtaining the final result.

subarray_sums = [] # Calculate the sum of every contiguous subarray and store it into subarray_sums for i in range(n): current_sum = 0

```
import java.util.Arrays; // Import Arrays class for sorting
class Solution {
```

Java

```
// Sort the array of subarray sums
       Arrays.sort(subarraySums);
        int result = 0; // Initialize the result to 0
        final int mod = (int) 1e9 + 7; // Modulo value to prevent integer overflow
        // Add the values from position "left" to "right" in the sorted subarray sums
        for (int i = left - 1; i < right; ++i) {</pre>
            result = (result + subarraySums[i]) % mod;
        return result; // Return the computed sum
C++
#include <vector>
#include <algorithm>
class Solution {
public:
    int rangeSum(vector<int>& nums, int n, int left, int right) {
        // Create an array to store the sums of all subarrays, with size based on the number of possible subarrays
        vector<int> subarraySums(n * (n + 1) / 2);
        int k = 0; // Index for inserting into subarraySums
       // Calculate the sum of all possible subarrays
        for (int start = 0; start < n; ++start) {</pre>
            int currentSum = 0; // Stores the sum of the current subarray
            for (int end = start; end < n; ++end) {</pre>
                currentSum += nums[end]; // Add the next element to the currentSum
                subarraySums[k++] = currentSum; // Store the sum of the subarray
       // Sort the sums of the subarrays
        sort(subarraySums.begin(), subarraySums.end());
        int answer = 0; // Variable to store the final answer
        const int mod = 1e9 + 7; // The modulo value
       // Calculate the sum of the subarray sums between indices left-1 and right-1 (inclusive)
        for (int i = left - 1; i < right; ++i) {</pre>
            answer = (answer + subarraySums[i]) % mod; // Aggregate the sum modulo mod
        return answer; // Return the final calculated sum
};
```

return answer; // Return the final calculated sum

class Solution:

from typing import List

subarray_sums = []

for i in range(n):

subarray_sums.sort()

mod = 10**9 + 7

Time Complexity:

Space Complexity:

current_sum = 0

for j in range(i, n):

TypeScript

```
return range_sum
Time and Space Complexity
  The given Python code computes the range sum of all possible contiguous subarrays sorted in a non-decreasing order and then
  returns the sum of the subarray values from the left to right indices (1-indexed). Here is the complexity analysis:
```

∘ The inner loop runs n - i times for each iteration of the outer loop, summing elements and appending the sum to the array arr. When i is 0, the inner loop will run n times; when i is n-1, it will run 1 time. On average, it will run n/2 times. \circ The sort operation on the array arr of size n * (n + 1) / 2 (the total number of subarrays) has a time complexity of $0(n^2 * \log(n^2))$

• The outer loop runs n times, where n is the length of the input list nums.

which simplifies to $0(n^2 * log(n))$. ∘ The sum operation over arr[left - 1: right] can be considered O(k), where k is the difference between right and left. However, this is negligible compared to the sorting complexity.

 \circ The space used by the array arr, which stores the n * (n + 1) / 2 sum values, dominates the space complexity. This results in 0(n^2).

 No other significant space-consuming structures or recursive calls that would impact the overall space complexity. Therefore, the space complexity of the code is $O(n^2)$ and the time complexity is $O(n^2 * log(n))$.

Combining these, the overall time complexity is dominated by the sort, resulting in $0(n^2 * log(n))$.