1714. Sum Of Special Evenly-Spaced Elements In Array

Hard Array Dynamic Programming Leetcode Link

Problem Description

The problem requires us to answer a set of queries on an array of non-negative integers, `nums`. Each query is represented by a pair `[x_i, y_i]`, where `x_i` is the starting index to consider in the array and `y_i` is the stride for selecting elements. Specifically, for each query, we want to find the sum of every `nums[j]` such that `j` is greater than or equal to `x_i` and the difference `(j - x_i)` is divisible by `y_i`. Importantly, the answers to the queries should be returned modulo `10^9 + 7`.

Intuition

The intuition behind the solution is based on the observation that direct computation of every query could be inefficient, especially when there are many queries or the array is very large. Thus, we need to optimize the process. For each `y_i` that is small (specifically, not larger than the square root of the size of `nums`), we can precompute a suffix sum array `suf` that helps answer the queries quickly. This exploits the fact that with smaller strides, there's more repetition and structure to use to our advantage.

For larger strides (`y_i` greater than the square root of the size of `nums`), it might be less efficient to use precomputation due to the sparsity of the elements we'd be summing. Therefore, for such strides, it's better to compute the sum directly per query.

This approach balances between precomputation for frequent, structured queries, and on-the-fly computation for less structured and less frequently occurring query patterns.

Solution Approach

The solution approach uses a combination of precomputation and direct calculation to handle the two scenarios efficiently: smaller strides (small `y_i`) and larger strides (large `y_i`).

Precomputation for smaller strides

First, we identify the threshold for small strides, which is the square root of `n`, the length of `nums`. We prepare a 2D array `suf` where each row corresponds to a different stride length upto the threshold. The idea is to calculate suffix sums starting from each index `j`. For stride `i`, `suf[i][j]` represents the sum of elements `nums[j]`, `nums[j+i]`, `nums[j+2i]`, and so on till the end of the array.

The precomputation occurs like this:

1. Iterate over each stride length `i` from `1` to `m`, where `m` is the square root of `n`.
2. For each stride length `i`, iterate backwards through the `nums` array starting from the last index.
3. Calculate the sum incrementally, adding `nums[j]` to the next value in the prefix already computed which is `suf[i][min(n, j + i)]`. This is because the next element in the sequence we're summing would be `j+i` elements ahead considering the stride.

This process essentially fills up the suffix sum array with all the necessary sums for smaller strides.

Direct calculation for larger strides

For each query with stride `y > m`, the direct calculation takes place as follows:

1. Starting at index `x`, generate a range of indices by slicing the `nums` array from `x` to the end of the array with step `y`. This selects every `y`th element starting at `x`.
2. Sum the selected elements.
3. Apply modulo `10^9 + 7` on the sum to avoid integer overflow and to conform with the problem requirements.

By combining these two approaches, we obtain an efficient method to answer all types of queries. The algorithm only uses precomputation for scenarios where it significantly reduces complexity, and falls back to direct summation when precomputation would not offer a speedup.

The main algorithm consists of:

- Precomputing the `suf` array for smaller strides.
- Iterating over each query.
- Checking if the stride `y` of the query is smaller or equal to `m`. If it is, use the precomputed `suf` value to answer the query. If it's larger, directly calculate the sum using slicing on the `nums` array.
- Each answer is then appended to the `ans` list, after applying the modulo operation.
- Finally, return the `ans` list as the result.

This hybrid approach is particularly efficient for handling a mix of query types on potentially large datasets, as it minimizes unnecessary computation while making use of precomputation wherever beneficial.

Example Walkthrough

Let's walk through an example to illustrate the solution approach.

Suppose we have the following `nums` array and queries:

```
nums = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
queries = [[1, 2], [0, 3], [2, 1]]
```

Let's consider `m` to be the threshold for small strides, which is the square root of the length of `nums`. Here, `n = 11`, so `m = sqrt(11) ≈ 3`. Thus, any stride `y_i <= 3` will involve precomputation.

Initial Setup

Precompute the suffix sum array `suf` for the strides less than or equal to `m`.

For stride 1: suf[1][...] = [35, 32, 31, 30, 29, 24, 22, 20, 14, 8, 5]

For stride 2: suf[2][...] = [36, 1, 25, 4, 21, 9, 11, 6, 8, 3, 5]

For stride 3: suf[3][...] = [22, 1, 6, 1, 14, 9, 2, 6, 5, 3, 5]

We now have precomputed sums for all smaller strides.

Answering Queries

- Query [1, 2]: Since the stride is 2 (which is less than or equal to `m`), we use the precomputed `suf` array.
  - Answer: `suf[2][1] = 1` (which is sum of `nums[1]`, `nums[1+2]`, `nums[1+4]`, etc., up to the end of array).
- Query [0, 3]: Since the stride is 3 (which is less than or equal to `m`), we also use the precomputed `suf` array.
  - Answer: `suf[3][0] = 22` (which is sum of `nums[0]`, `nums[0+3]`, `nums[0+6]`, etc., up to the end of array).
- Query [2, 1]: As the stride is 1, we refer again to the precomputed `suf` array.
  - Answer: `suf[1][2] = 31` (which is sum of `nums[2]`, `nums[3]`, `nums[4]`, ... , until the end of the array).

Finally, for each answer, we apply the modulo `10^9 + 7` operation.

- Final Answer List: `[1 mod (10^9 + 7), 22 mod (10^9 + 7), 31 mod (10^9 + 7)]` or `[1, 22, 31]` since all values are already less than `10^9 + 7`.

This completes the example, which demonstrates how the algorithm efficiently answers queries by using a combination of precomputation for smaller strides and direct computation for larger ones.

Python Solution

```python
from typing import List
from math import sqrt

class Solution:
    def solve(self, nums: List[int], queries: List[List[int]]) -> List[int]:
        MOD = 10**9 + 7  # Define the modulus for result as per problem statement to avoid large integers
        n = len(nums)  # The total number of elements in nums
        sqrt_n = int(sqrt(n))  # The square root of the length of nums, which determines the threshold
        prefix_sums = [[0] * (n + 1) for _ in range(sqrt_n + 1)]  # Initialize the prefix sums matrix

        # Fill the prefix sums matrix for all blocks with size up to sqrt(n)
        for block_size in range(1, sqrt_n + 1):
            for i in range(n - 1, -1, -1):  # Start from the end to compute prefix sums
                next_index = min(n, i + block_size)
                prefix_sums[block_size][i] = prefix_sums[block_size][next_index] + nums[i]

        results = []  # This will store the results of each query
        for start, interval in queries:
            # If the interval is under the square root threshold, use precomputed sums
            if interval <= sqrt_n:
                result = prefix_sums[interval][start] % MOD
            else:
                # For intervals larger than square root of n, calculate the sum on the fly
                result = sum(nums[start::interval]) % MOD
            results.append(result)

        return results  # Return the results list for all queries

```

Java Solution

```java
class Solution {
    public int[] solve(int[] nums, int[][] queries) {
        // Calculate the length of the nums array and the square root of that length
        int numLength = nums.length;
        int squareRootOfNumLength = (int) Math.sqrt(numLength);

        // Define the modulo value to avoid overflow
        final int mod = (int) 1e9 + 7;

        // Create a 2D array for storing suffix sums
        int[][] suffixSums = new int[squareRootOfNumLength + 1][numLength + 1];

        // Calculate suffix sums for blocks with size up to the square root of numLength
        for (int i = 1; i <= squareRootOfNumLength; ++i) {
            for (int j = numLength - 1; j >= 0; --j) {
                suffixSums[i][j] = (suffixSums[i][Math.min(numLength, j + i)] + nums[j]) % mod;
            }
        }

        // Get the number of queries and initialize an array to store the answers
        int queryCount = queries.length;
        int[] answers = new int[queryCount];

        // Process each query
        for (int i = 0; i < queryCount; ++i) {
            int startIndex = queries[i][0];
            int stepSize = queries[i][1];

            // If the step size is within the computed suffix sums, use the precomputed value
            if (stepSize <= squareRootOfNumLength) {
                answers[i] = suffixSums[stepSize][startIndex];
            } else {
                // If the step size is larger, calculate the sum on the fly
                int sum = 0;
                for (int j = startIndex; j < numLength; j += stepSize) {
                    sum = (sum + nums[j]) % mod;
                }
                answers[i] = sum;
            }
        }

        // Return the array of answers
        return answers;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <cstring>
#include <cmath>

class Solution {
public:
    std::vector<int> solve(std::vector<int>& nums, std::vector<std::vector<int>>& queries) {
        int numsSize = nums.size();
        int blockSize = static_cast<int>(sqrt(numsSize)); // The size of each block for the sqrt decomposition.
        const int mod = 1e9 + 7; // The modulo value to prevent integer overflow.
        int suffix[blockSize + 1][numsSize + 1]; // Suffix sums matrix.

        // Initialize the suffix sums matrix with zeros.
        memset(suffix, 0, sizeof(suffix));

        // Pre-compute the suffix sums for all possible blocks.
        for (int i = 1; i <= blockSize; ++i) {
            for (int j = numsSize - 1; j >= 0; --j) {
                suffix[i][j] = (suffix[i][std::min(numsSize, j + i)] + nums[j]) % mod;
            }
        }

        std::vector<int> ans; // Vector to store the answers to the queries.

        // Iterate over each query and calculate the sum accordingly.
        for (auto& query : queries) {
            int start = query[0], step = query[1]; // Start index and step for the current query.

            // If the step is less than or equal to the block size, use the precomputed suffix sums.
            if (step <= blockSize) {
                ans.push_back(suffix[step][start]);
            } else {
                // Otherwise, perform a brute-force sum calculation.
                int sum = 0;
                for (int i = start; i < numsSize; i += step) {
                    sum = (sum + nums[i]) % mod;
                }
                ans.push_back(sum); // Add the computed sum to the answers vector.
            }
        }

        return ans; // Return the vector of answers.
    }
};

```

Typescript Solution

```typescript
// Defining a function to solve the query based on the array of numbers and list of queries
function solve(nums: number[], queries: number[][]): number[] {
    const arrayLength: number = nums.length;                        // Length of the nums array
    const sqrtLength: number = Math.floor(Math.sqrt(arrayLength)); // Sqrt decomposition length
    const modulus: number = 1e9 + 7;                               // Define modulus for large numbers

    // Suffix arrays to store pre-computed sums. These are used to answer queries efficiently for small y values
    const suffixSums: number[][] = Array(sqrtLength + 1)
        .fill(0)
        .map(() => Array(arrayLength + 1).fill(0));

    // Pre-compute the suffix sums for each possible block size up to the square root of the length of the array
    for (let blockSize = 1; blockSize <= sqrtLength; ++blockSize) {
        for (let startIndex = arrayLength - 1; startIndex >= 0; --startIndex) {
            let nextIndex = Math.min(arrayLength, startIndex + blockSize);
            suffixSums[blockSize][startIndex] = (suffixSums[blockSize][nextIndex] + nums[startIndex]) % modulus;
        }
    }

    // Array to hold answer for each query
    const answers: number[] = [];

    // Process each query
    for (const [startIndex, stepSize] of queries) {
        if (stepSize <= sqrtLength) {
            // If stepSize is within the pre-computed range, use pre-computed sum for efficiency
            answers.push(suffixSums[stepSize][startIndex]);
        } else {
            // For larger step sizes, calculate the sum manually
            let sum = 0;
            for (let i = startIndex; i < arrayLength; i += stepSize) {
                sum = (sum + nums[i]) % modulus;
            }
            answers.push(sum); // Append the sum to the answers array
        }
    }

    return answers; // Return the array containing sums for each query
}

```

Time and Space Complexity

Time Complexity

The time complexity of the precomputation step (building the `suf` array) is `O(m * n)`, where `m` is `int(sqrt(n))` and `n` is the length of the input array `nums`. This is because the outer loop runs `m` times and the inner loop runs `n` times.

For each query in `queries`, there are two cases to consider:

1. When `y <= m`: The complexity for this case is `O(1)` because the result is directly accessed from the precomputed `suf` array.
2. When `y > m`: The complexity is `O(n / y)` because the sum is computed using slicing with a step `y`, so it touches every `y`th element.

Given that there are `q` queries, if we denote the number of queries where `y > m` as `q1` and where `y <= m` as `q2`, then the total time for all queries is `O(q1 * (n / y) + q2)`. However, in the worst case, all queries could be such that `y > m`, which makes it `O(q * (n / y))`.

The total time complexity is therefore `O(m * n + q * (n / y))`. In the worst case scenario where `y` is just above `m`, this could be approximated as `O(m * n + q * n / m)`, which simplifies to `O(m * n + q * sqrt(n))`.

Space Complexity

The space complexity is primarily due to the additional 2D list `suf`. Since `suf` has a size of `(m+1) * (n+1)`, its space complexity is `O(m * n)`. Additionally, the space used by `ans` to store the results grows linearly with the number of queries `q`, hence `O(q)`. Therefore, the total space complexity is `O(m * n + q)`.