891. Sum of Subsequence Widths

Hard Array Math Sorting

Problem Description

The problem requires us to calculate the sum of widths for all non-empty subsequences of an input array `nums`, where the width of a subsequence is defined as the difference between the maximum and minimum elements in that subsequence. For instance, in the subsequence `[3, 6, 2, 7]`, the width would be `7 - 2 = 5`. Since a sequence can have a large number of subsequences, and thus the answer can be quite large, we are asked to return the result modulo `10**9 + 7`.

A subsequence is different from a subset, as a subsequence maintains the relative order of elements as they appear in the original array, whereas a subset does not. For example, `[3, 6, 2, 7]` is a subsequence of `[0, 3, 1, 6, 2, 2, 7]` but `[3, 7, 6, 2]` is not since the order of elements is changed.

The challenge here is to find an efficient way to calculate the sum of widths of all possible non-empty subsequences without having to enumerate each one, as that would be computationally infeasible for large arrays.

Intuition

To understand the solution, let's start with a simple observation about subsequences. When we choose a subsequence from the sorted array, each element can either be the maximum, the minimum, or neither in that subsequence. If we focus on one element, we can observe that it will be the maximum in some of the subsequences and the minimum in others.

Considering an element `nums[i]`, it will be the maximum element in any subsequence including elements from `nums[i]` to the end of the array `nums`. The number of such subsequences can be found by counting the number of combinations of the remaining elements, which is `2**(len(nums) - i - 1)`, since each of the remaining elements can be either included or not included in the subsequence.

Similarly, `nums[i]` can be the minimum element in any subsequence starting from the beginning of the array and including up to `nums[i]`. The number of such subsequences will be `2**i`, following the same logic.

By sorting the array, we ensure that for each pair of elements `(nums[i], nums[j])` where `i < j`, `nums[i]` will always be the minimum and `nums[j]` will always be the maximum if both are used in a subsequence.

The solution calculates the sum of `nums[j] * 2**j - nums[i] * 2**i` over all pairs `(i, j)` where `i < j`. This product is added to the answer if `nums[j]` is the maximum and subtracted if `nums[i]` is the minimum. We iterate through the array, keeping track of the power of 2, and for each element, we add and subtract its contribution to the final sum. To avoid large numbers, we take modulo `10**9 + 7` at each step.

In code, `p` is the power of 2, which is doubled (left-shifted) at each step to represent the increasing powers of 2. The variable `ans` accumulates the sum of widths modulo `10**9 + 7`.

This approach allows us to find the sum of widths efficiently without calculating each subsequence explicitly.

Solution Approach

The solution makes use of a single-pass algorithm along with some simple mathematical insights into the properties of subsequences in a sorted array. Here's how it's implemented, referring to the provided solution code:

1. Sort the input list `nums`. Sorting is the essential first step as it allows us to consider the impact of each element in the array as a potential minimum or maximum in the subsequence. Sorting provides easy access to sequences' bounds for width calculations.
2. Initialize two variables, `ans` to collect the answer, and `p` to keep track of powers of 2. The power of 2 is important in this solution because for any element `nums[i]`, there are `2**i` subsequences for which it can be the minimum and `2**(n-i-1)` (where `n` is the length of the list) subsequences for which it can be the maximum.
3. Iterate through the sorted list with a loop using `enumerate(nums)` so that both the index `i` and value `v` are available. Perform the following calculation for each element:

   a. Calculate and update `ans` with the width contributed by the current element as the maximum and minimum. This is done by adding `(v + p)` for the subsequences where it acts as maximum and subtracting `(nums[-i - 1] + p)` (which is the element at the same distance from the end of the list, therefore `nums[i]`'s counterpart) for the subsequences where it acts as minimum.

   b. Every time we consider a new element, we are looking at subsequences that have one more potential member. This is why we double `(+= 1)` the value of `p` (to represent `p * 2`) for each step, as each step considers subsequences with one more element than the previous step.
4. Take modulus `10**9 + 7` for the updated `ans` and `p` at each step to ensure that numbers stay within integer bounds and prevent overflow.

The key insight for the algorithm is that each element's total contribution to the width of all subsequences can be calculated in isolation by multiplying its value by the number of subsequences in which it serves as a maximum, and subtracting the result of multiplying it by the number of subsequences where it serves as a minimum.

Data structures and patterns:

- **Sorting** - A fundamental step that helps in calculating the contribution of each element to subsequences' width accurately.
- **Bitwise operations** - Specifically left shift operation (`<<`), which is used as an efficient way to calculate powers of 2.
- **Modular arithmetic** - To handle large numbers and avoid overflow, as well as to produce the output as per problem statement requirements.

The algorithm's time complexity is `O(n log n)` due to the initial sorting, and its space complexity is `O(1)`, not counting the input and output, as only constant extra space is needed.

Example Walkthrough

Let's illustrate the solution approach with a small example. Consider the array `nums = [2, 1, 3]`.

1. First, we sort `nums` to get `[1, 2, 3]`. This sorted array will help us easily identify each number's contribution as a minimum or maximum in subsequences.
2. We initialize `ans` as 0 (the eventual answer) and `p` as 1 (representing 2^0, the power of 2 for the first element since there are no elements before it).
3. We iterate through the sorted `nums` with their index `i`.
   - a. For the first element `1` at index `0`:
     - It can be the minimum in `2**0 = 1` subsequence (itself only).
     - As a minimum, it contributes `1 * 2**0 = 1 * 1 = 1` to `ans`, but since it is a minimum, we actually subtract it, so `ans = ans - 1 = 0 - 1`.
     - It can be the maximum in `2**(3-0-1) = 2**2 = 4` subsequences (the subsequences `[1, 1, 2]`, `[1, 1, 3]`, and `[1, 2, 3]`).
     - We double `p` for the next element, so now `p = 2`.
   - b. For the second element `2` at index `1`:
     - It can be the minimum in `2**1 = 2` subsequences (`[2]` and `[2, 3]`).
     - It can be the maximum in `2**(3-1-1) = 2**1 = 2` subsequences (`[1, 2]` and `[2]`).
     - We add its contribution as `ans = ans + 2 * 2 - 2 * 1 = -1 + 4 - 2 = 1`.
     - Double `p`, so `p = 4`.
   - c. For the third and final element `3` at index `2`:
     - It can be the minimum in `2**2 = 4` subsequences (`[3]`, `[2, 3]`, `[1, 3]`, and `[1, 2, 3]`).
     - It contributes `3 * 4 - 3 * 1 = 12 - 3 = 9` to `ans`.
     - So, we update `ans = ans + 9 = 1 + 9 = 10`.
4. We did not need to use modular arithmetic here since the numbers are small, but in the solution, at each step where numbers could overflow, we would take `ans %= (10**9 + 7)`.

The final `ans` is `10`, which represents the sum of the widths of all possible non-empty subsequences of the original `nums` array. Therefore, for the input `[2, 1, 3]`, the sum of widths is `10`.

This method demonstrates how each element contributes to the width calculation without explicitly enumerating all subsequences, thereby optimizing the computation.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def sumSubseqWidths(self, A: List[int]) -> int:
        # Define the modulus as per the problem statement to avoid large integers.
        mod = 10**9 + 7

        # Sort the array in non-decreasing order.
        A.sort()

        # Initialize the result variable to store the sum of widths.
        result = 0

        # Initialize power to be used for computing number of subsequences.
        power = 1

        # Loop through the array while computing the contribution of
        # each element to the sum of widths of all subsequences.
        for i, value in enumerate(A):
            # The width contribution for each element is the difference between
            # the current value and the value at the mirrored index from the end,
            # multiplied by the current power of 2, representing the count of
            # subsequences it will be a part of as a min or max.
            # This is because there are power number of subsequences where A[i]
            # is the maximum and power number where A[-i-1] is the minimum.
            # We take the modulus to handle large numbers.
            result = (result + (value - A[-i - 1]) * power) % mod

            # Bitwise left shift of power (equivalent to multiplying by 2)
            # to reflect the increase in the number of subsequences
            # that include the next element.
            power = (power << 1) % mod

        # Return the final result after considering all elements.
        return result
```

Java

```java
class Solution {
    // Define the module value to be used for the modulo operation.
    private static final int MOD = (int) 1e9 + 7;

    public int sumSubseqWidths(int[] nums) {
        // Sort the input array in non-decreasing order.
        Arrays.sort(nums);
        // Initialize accumulator for the answer.
        int answer = 0;
        // Initialize power term (2^i) which will be used in the sum calculation.
        long powerOfTwo = 1;
        // Get the length of nums array.
        int n = nums.length;

        // Loop over each element to calculate contribution to the answer.
        for (int i = 0; i < n; ++i) {
            // Update the answer by adding the difference between the current element and
            // the mirror element (from the end) multiplied by the current power of 2.
            // The MOD is added before taking modulo to ensure the subtraction does not result in negative values.
            answer = (answer + (nums[i] - nums[n - i - 1]) * powerOfTwo + MOD) % MOD;
            // Update powerOfTwo for the next iteration. Shift left is equivalent to multiplying by 2.
            // Take modulo to ensure that the value never overflows.
            powerOfTwo = (powerOfTwo << 1) % MOD;
        }

        // Cast the final answer to int, as required by the problem statement, and return it.
        return (int) answer;
    }
}
```

C++

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    // Define the modulus to handle the large numbers as the problem might require
    // operating under a large prime (10^9 + 7 is often used in problems to avoid integer overflow)
    static constexpr int MOD = 1e9 + 7;

    int sumSubseqWidths(vector<int>& nums) {
        // First, sort the numbers in non-decreasing order
        std::sort(nums.begin(), nums.end());

        long long answer = 0; // Use long long for intermediate results to prevent overflow
        long long powerOfTwo = 1; // We'll use powers of 2, starting with 2^0 = 1
        int n = nums.size(); // Store the size of the nums array for repeated use

        // Iterate over each element in the sorted array
        for (int i = 0; i < n; ++i) {
            // The width of a subsequence is the difference between max and min
            // For each element, calculate the number of subsequences where it's the max
            // and the number where it's the min, and subtract the latter from the former.
            // Multiply this count by the current element's value, adjusted for the modulo
            answer = (answer + (nums[i] - nums[n - i - 1]) * powerOfTwo % MOD + MOD) % MOD;

            // Update the power of 2 for the next iteration, adjusting for the modulo
            powerOfTwo = (powerOfTwo << 1) % MOD;
        }

        // Return the final answer, after considering all elements
        return static_cast<int>(answer); // Cast to int before returning as per function signature
    }
};
```

TypeScript

```typescript
// Define the modulus to handle potential large numbers and avoid integer overflow
const MOD: number = 1e9 + 7;

function sumSubseqWidths(nums: number[]): number {
    // Sort the numbers in non-decreasing order
    nums.sort((a, b) => a - b);

    let answer: number = 0; // Use a number for result as TypeScript doesn't have integer overflow concerns in the same way as C++
    let powerOfTwo: number = 1; // Start with 2^0 which is 1
    const n: number = nums.length; // Store the length of the nums array

    // Iterate over each element in the sorted array
    for (let i = 0; i < n; ++i) {
        // The width of a subsequence is the difference between max and min
        // For each element, calculate the number of subsequences where it's the max
        // and the number where it's the min, and subtract the latter from the former.
        // Multiply this count by the current element's value, adjusted for the modulo
        answer = (answer + (nums[i] - nums[n - i - 1]) * powerOfTwo % MOD + MOD) % MOD;

        // Update the power of 2 for the next iteration, adjusting for the modulo
        powerOfTwo = (powerOfTwo + 2) % MOD;
    }

    // Return the answer as per function signature
    return answer;
}
```

```python
from typing import List

class Solution:
    def sumSubseqWidths(self, A: List[int]) -> int:
        # Define the modulus as per the problem statement to avoid large integers.
        mod = 10**9 + 7

        # Sort the array in non-decreasing order.
        A.sort()

        # Initialize the result variable to store the sum of widths.
        result = 0

        # Initialize power to be used for computing number of subsequences.
        power = 1

        # Loop through the array while computing the contribution of
        # each element to the sum of widths of all subsequences.
        for i, value in enumerate(A):
            # The width contribution for each element is the difference between
            # the current value and the value at the mirrored index from the end,
            # multiplied by the current power of 2, representing the count of
            # subsequences it will be a part of as a min or max.
            # This is because there are power number of subsequences where A[i]
            # is the maximum and power number where A[-i-1] is the minimum.
            # We take the modulus to handle large numbers.
            result = (result + (value - A[-i - 1]) * power) % mod

            # Bitwise left shift of power (equivalent to multiplying by 2)
            # to reflect the increase in the number of subsequences
            # that include the next element.
            power = (power << 1) % mod

        # Return the final result after considering all elements.
        return result
```

Time and Space Complexity

The time complexity of the provided code is `O(n log n)` due to the `nums.sort()` operation, which is the most time-consuming part. Sorting the array is necessary before we can calculate the summation of subsequence widths. The following loop runs in linear time `O(n)`, but it does not dominate the sorting's time complexity.

The space complexity of the code is `O(1)` because it uses a fixed amount of extra space. Variables such as `ans`, `p`, `mod`, and the iteration variables `i` and `v` do not depend on the size of the input array `nums`. No additional space that scales with input size is used, as the sorting operation is performed in-place (Python's `sort()` method for lists).