1498. Number of Subsequences That Satisfy the Given Sum Condition

Medium Array Two Pointers Binary Search Sorting Leetcode Link

Problem Description

We are given an array `nums` of integers and another integer `target`. Our goal is to compute the number of non-empty subsequences from the `nums` array such that the sum of the smallest and largest number in each subsequence is less than or equal to the `target`. Since the answer could be very large, we only require the answer modulo `10^9 + 7`. It's noteworthy to mention that a subsequence does not have to consist of consecutive elements and can be formed by deleting some or no elements without changing the order of the remaining elements.

Intuition

To solve this problem, we should think about certain properties of subsequences and constraints. A critical observation is that for a given smallest element, if we can fix the largest element that would satisfy the target constraint, then all combinations of elements between the smallest and largest one will also satisfy the property (because adding elements in between will not affect the smallest and largest values of the subsequence).

So, the steps we might consider are:

1. Sort the array to efficiently manage the smallest and largest elements.
2. Initialize an array `f` to precompute the powers of 2, which represent the number of combinations of elements in between two fixed points, since those can freely be included or excluded.
3. Iterate over the sorted array with a pointer `i` to find a valid smallest element.
4. For each `i`, use binary search (`bisect_right`) to find the largest permissible element `j` where `nums[i] + nums[j]` is still not greater than `target`.
5. The power of 2 at `f[j - i]` now tells us the count of valid subsequences between `i` and `j` because all in-between elements give us that many combinations. We use modular arithmetic for this calculation.
6. We continue until the smallest element alone exceeds half the `target`, since no viable pair (as smallest + largest) will then satisfy the sum constraint.
7. Sum all these counts to get the answer.

This approach minimizes the computation by reducing the problem to a series of binary searches and combinations (powers of 2) within the sorted bounds of the array, leveraging the property of subsequences in a sorted array.

Solution Approach

The solution primarily makes use of sorting, binary search, pre-computed powers of two, and modular arithmetic. Here is the step-by-step explanation:

1. **Sorting**: The first step is to sort the `nums` array. Sorting is essential as it allows us to treat the first element of any subsequence we consider as the minimum and the last as the maximum. This makes it simple to enforce the constraint that the sum of the minimum and maximum elements is less than or equal to `target`.

2. **Precomputing Powers of Two**: Before entering the main loop, we initialize an array `f` where `f[i]` is meant to store `2^i % mod`. This array is filled up to `n` (the length of `nums`) plus one to account for an empty subsequence. The reason for this is that for any fixed pair of minimum and maximum, there are `2^(number of elements between them)` possible subsequences including or excluding these middle elements.

3. **Main Loop**: The main loop iterates over each element `x` in the sorted array `nums`. This element is sampled as a candidate for the minimum value in our subsequence.

4. **Binary Search**: For each candidate minimum element, we use `bisect_right` from the `bisect` module to efficiently find the rightmost index `j` such that the sum of `nums[i]` and `nums[j]` is less than or equal to `target`. The function `bisect_right(nums, target - x, i + 1)` is used to find an index just beyond the largest element that can be paired with `nums[i]` to form a valid subsequence.

5. **Counting Valid Subsequences**: After finding `j`, we calculate the number of valid subsequences that have `nums[i]` as the smallest and `nums[j]` as the largest element. This number is `f[j - i]`, which is the count of all subsequences formed by the elements in between `i` and `j`. This value is added to our running total `ans`, using modular arithmetic to prevent overflow.

6. **Modular Arithmetic**: All arithmetic operations are done modulo `10^9 + 7` (`mod`), as the problem statement requests the final answer to be given modulo this prime number. This ensures that intermediate results and the final answer stay within the bounds of an `int` and do not cause overflow.

7. **Break Condition**: We can break early out of our loop when the minimum element itself is greater than half of the `target`. Since the array is sorted and any pair will at least double the minimum value, no subsequent pairs can have a valid sum, optimizing our solution.

Example Walkthrough

Let's say the `nums` array is `[1, 2, 3, 4, 5]`, and the `target` is `7`. Here's how the solution approach would be applied to this example:

1. **Sorting**: First, we sort the array, but since it is already sorted `[1, 2, 3, 4, 5]`, there's no change.

2. **Precomputing Powers of Two**: Suppose we precompute powers of two modulo `10^9+7` up to `n+1` where `n` is length of `nums`. Our array `f` for powers of 2 will look like this: `[1, 2, 4, 8, 16, ...]` because `f[0] = 2^0 % mod`, `f[1] = 2^1 % mod`, and so on.

3. **Main Loop**: We iterate `i` from the start of the array. Starting with `i=0`, our minimum candidate value is `nums[0] = 1`.

4. **Binary Search**: We perform a binary search to find the border `j`. Using `bisect_right`, we find the largest `j` such that `nums[i] + nums[j] <= target`. For `i=0` and `target=7`, the largest pairable value with `1` is `5`, so `j=4` (index of `5`).

5. **Counting Valid Subsequences**: There are `j - i - 1` elements between `nums[i]` and `nums[j]`, which means `4 - 0 - 1 = 3` elements. Thus, there are `2^3 = 8` possible subsequences. After processing `i=0`, our `ans` is `8 % mod`.

6. **Modular Arithmetic**: We continue the loop and perform the same computations. As we continue, all operations are done modulo `10^9 + 7`.

7. **Break Condition**: We reach a point where the smallest element `nums[i]` is greater than half the target. Here, when `i` reaches `4` (`nums[i]=5`), we can break out of the loop, as no subsequent pairs from the sorted array will satisfy the sum condition. The loop condition saves us from going through unnecessary elements.

In the provided array, we keep picking a smallest element `x` (starting from `1` to `5`), and for each `x`, we find the maximum `y` such that `x+y <= target` using binary search. Then, we calculate all possible subsequences using `f[j - i]`. Summing these counts gives us the total number of valid subsequences whose sum of smallest and largest numbers is less than or equal to the `target`.

By applying this process to the example given, we successfully count all valid subsequences while efficiently managing the constraints through sorting, binary search, and precomputed powers of two, combined with modular arithmetic to keep the numbers manageable.

Python Solution

```python
from bisect import bisect_right

class Solution:
    def numSubseq(self, nums: List[int], target: int) -> int:
        # Constant for modulo operation
        mod = 10**9 + 7
        # Sort the input list to make use of binary search later
        nums.sort()
        n = len(nums)

        # Initialize power_of_two array which stores 2^i values modulo mod
        power_of_two = [1] * (n + 1)
        for i in range(1, n + 1):
            # Efficiently precompute powers of 2 mod mod
            power_of_two[i] = power_of_two[i - 1] * 2 % mod

        # Initialize the answer to 0
        ans = 0
        # Loop through the list
        for i, num in enumerate(nums):
            # Stop if the smallest number in a subsequence is too big
            if num * 2 > target:
                break
            # Find the largest number that can be paired with nums[i]
            # such that their sum does not exceed target.
            j = bisect_right(nums, target - num, i + 1) - 1

            # Add the count of valid subsequences starting with nums[i]
            # The count is the number of different ways to form subsequences
            # from i+1 to j (inclusive), which is simply 2 raised to the power
            # of the number of elements between i and j, modulo mod.
            # This relies on the fact that for every element between i and j,
            # we can choose to either include it or not in our subsequence.
            ans = (ans + power_of_two[j - i]) % mod

        # Return the total count of valid subsequences modulo mod
        return ans
```

Java Solution

```java
class Solution {
    public int numSubseq(int[] nums, int target) {
        // Sort the input array to facilitate the two-pointer approach
        Arrays.sort(nums);

        // Modulus value for avoiding integer overflow
        final int MOD = (int) 1e9 + 7;

        // Get the length of the nums array
        int n = nums.length;

        // Create an array to store powers of 2 mod MOD, up to n
        int[] powersOfTwoMod = new int[n + 1];
        powersOfTwoMod[0] = 1;

        // Precompute the powers of two modulo MOD for later use
        for (int i = 1; i <= n; ++i) {
            powersOfTwoMod[i] = (powersOfTwoMod[i - 1] * 2) % MOD;
        }

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

        // Iterate through the numbers in the sorted array
        for (int i = 0; i < n; ++i) {
            // If the smallest number in the subsequence is greater than half of the target, stop the loop
            if (nums[i] * 2L > target) {
                break;
            }

            // Find the largest index j such that nums[i] + nums[j] <= target
            int j = binarySearch(nums, target - nums[i], i + 1) - 1;

            // Add the count of subsequences using the powers of two precomputed values
            answer = (answer + powersOfTwoMod[j - i]) % MOD;
        }

        // Return the total number of subsequences that satisfy the condition
        return answer;
    }

    // Helper function: binary search to find the rightmost index where nums[index] <= x
    private int binarySearch(int[] nums, int x, int left) {
        int right = nums.length;

        // Continue searching while the search space is valid
        while (left < right) {
            int mid = (left + right) >> 1; // Calculate the middle index

            // Narrow the search to the left half if nums[mid] > x
            if (nums[mid] > x) {
                right = mid;
            } else {
                // Otherwise, narrow the search to the right half
                left = mid + 1;
            }
        }

        // Return the insertion point for x
        return left;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int numSubseq(vector<int>& nums, int target) {
        // Sort the original array to facilitate binary search
        sort(nums.begin(), nums.end());

        // Define mod as the required modulus
        const int mod = 1e9 + 7;
        int n = nums.size();

        // Initialize a vector for fast exponentiation
        vector<int> fastExp(n + 1);
        fastExp[0] = 1;

        // Calculate powers of 2 modulo mod in advance
        for (int i = 1; i <= n; ++i) {
            fastExp[i] = (fastExp[i - 1] * 2) % mod;
        }

        int count = 0; // Initialize count of subsequences

        // Iterate over the nums array to find valid subsequences
        for (int i = 0; i < n; ++i) {
            // If the smallest number is already greater than the target when doubled, break
            if (nums[i] * 2L > target) {
                break;
            }

            // Find the index of the largest number that can be paired with nums[i]
            int j = upper_bound(nums.begin() + i + 1, nums.end(), target - nums[i]) - nums.begin() - 1;

            // Add the number of valid subsequences with nums[i] as the smallest number
            count = (count + fastExp[j - i]) % mod;
        }

        // Return the total count of valid subsequences
        return count;
    }
};
```

Typescript Solution

```typescript
// Define mod as the required modulus
const MOD = 1e9 + 7;

// Function for fast exponentiation
const fastExponentiation = (length: number): number[] => {
    let fastExp: number[] = new Array(length + 1).fill(1);
    for (let i = 1; i <= length; ++i) {
        fastExp[i] = (fastExp[i - 1] * 2) % MOD;
    }
    return fastExp;
};

// Function to count number of subsequences with sum of min + max <= target
const numSubseq = (nums: number[], target: number): number => {
    // Sort the original array to facilitate binary search
    nums.sort((a, b) => a - b);

    const n: number = nums.length;
    // Initialize vector for fast exponentiation
    const fastExp = fastExponentiation(n);

    let count = 0; // Initialize count of subsequences

    // Iterate over the nums array to find valid subsequences
    for (let i = 0; i < n; ++i) {
        // If the smallest number is already greater than the target when doubled, exit loop
        if (nums[i] * 2 > target) {
            break;
        }

        // Find the index of the largest number that can be paired with nums[i]
        let j = upperBound(nums, i, n, target - nums[i]) - 1;

        // Add the number of valid subsequences with nums[i] as the smallest number
        count = (count + fastExp[j - i]) % MOD;
    }

    // Return the total count of valid subsequences
    return count;
};

// Function that works like C++ upper_bound
// It finds the first index in nums where nums[index] is greater than value
const upperBound = (nums: number[], startIndex: number, length: number, value: number): number => {
    let low = startIndex, high = length;
    while (low < high) {
        const mid: number = low + Math.floor((high - low) / 2);
        if (value >= nums[mid]) {
            low = mid + 1;
        } else {
            high = mid;
        }
    }
    return low;
};
```

Time and Space Complexity

The given Python code aims to count the number of subsequnces in an array `nums` that add up to a sum less than or equal to `target`, with a constraint that within each subsequence, the maximum plus minimum value should not exceed `target`. Here's the analysis of its time complexity and space complexity:

Time Complexity

1. `nums.sort()`: This line sorts the array in place which, in the worst case, has a time complexity of `O(n log n)`, where `n` is the length of the array.

2. The loop to calculate powers of 2 (`f[i] = f[i - 1] * 2 % mod`) runs `n` times, hence the time complexity for this loop is `O(n)`.

3. The main loop, which calculates the answer, iterates over the array once (`for i, x in enumerate(nums)`), which gives `O(n)` complexity for the loop's iteration.

4. Inside the loop, `bisect_right` is used, which is an algorithm for binary search in Python, and it has a time complexity of `O(log n)`. However, because it runs once for each element in `nums`, the overall complexity for this part is `O(n log n)`.

The dominating factor in the above analyses is `O(n log n)`. The `sort` operation and the binary search inside the loop both contribute to this complexity, thus the total time complexity of the function is `O(n log n)`.

Space Complexity

1. The space allocated for `f` which stores the increasing powers of 2, up to `n`, is `O(n)` since it holds `n + 1` elements.

2. No additional significant space is used, as the sorting is done in place and the other operations use constant space.

Therefore, the overall space complexity of the function is `O(n)`.