2389. Longest Subsequence With Limited Sum

Easy Greedy Array Binary Search Prefix Sum Sorting Leetcode Link

Problem Description

The problem gives us two integer arrays: `nums` which is of length `n`, and `queries` which is of length `m`. Our task is to find, for each query, the maximum size of a subsequence that can be extracted from `nums` such that the sum of its elements does not exceed the value given by the corresponding element in `queries`.

It's important to recognize what a subsequence is. A subsequence is a new sequence generated from the original array where some (or no) elements are deleted, but the order of the remaining elements is retained. It's also worth noticing that there might be more than one subsequence that meets the condition for a query, and among those, we are looking to report the length of the largest one.

For example, suppose `nums = [4, 5, 2]`, and we have a single-element `queries = [6]`. The subsequences of `nums` that have sums less than or equal to `6` are `[4]`, `[5]`, `[2]`, and `[4, 2]`. The longest of these is `[4, 2]`, which has a length of `2`.

Intuition

To solve this problem, we first need to sort the `nums` array. Sorting will help us easily find subsequences that maximize size while minimizing their sum—since we'll have guaranteed that we're always adding the smallest possible numbers (from the sorted array) to reach the sum required in the queries.

Next, we calculate the prefix sum of the `nums` array after sorting it. This prefix sum array (`s`) represents the sum of elements of `nums` up to and including the `i`th element. So the first element of the prefix sum will be equal to the first element of `nums`, the second will be the sum of the first two elements of `nums`, and so on.

Once we have the prefix sum array, we can easily determine, for each query, how many elements can be taken from the `nums` array without exceeding the query sum. We do this by finding the rightmost index in the prefix sum array that is less than or equal to the query value. This index tells us the number of elements that can be included in the subsequence for a particular query.

The `bisect_right` function from Python's `bisect` module is a binary search function which returns the index where an element should be inserted to maintain the order. In our case, for each query, it finds the first element in the sorted prefix sum array `s` that is greater than the query value—thus, the index we get is actually the length of the subsequence, because array indices are 0-based in Python.

Our final answer is an array of lengths, each corresponding to the maximum subsequence lengths that can be picked for each query without exceeding their sums.

The beauty of this approach is that by utilizing the sorted array and prefix sums, the time-consuming task of checking each possible subsequence is avoided, leading to a much more efficient solution.

Solution Approach

The implementation of the solution can be broken down into the following steps:

1. **Sort the Input Array**: We start by sorting `nums` in ascending order. This arrangement guarantees that when we choose subsequences, we are choosing the smallest available elements first, which helps us maximize the number of elements in the subsequence for any given sum.

   ```
   nums.sort()
   ```

2. **Calculate Prefix Sums**: We then compute a prefix sum array `s` from the sorted array. The prefix sum is a common technique used in algorithm design for quickly finding the sum of elements in a subarray. In Python, the `accumulate` function from the `itertools` module can compute the prefix sums efficiently.

   ```
   s = list(accumulate(nums))
   ```

3. **Search with Binary Search**: For each query, we use the `bisect_right` function to perform a binary search over the prefix sum array. This binary search will find the point where the query value (the sum we are not supposed to exceed) would be inserted to keep the array `s` sorted. Since arrays in Python are 0-indexed, the insertion index found by `bisect_right` directly corresponds to the number of elements we can sum up from the sorted `nums` array to not exceed the query value. This is equivalent to the length of the required subsequence.

   ```
   [bisect_right(s, q) for q in queries]
   ```

The implementation uses the following concepts:

- **Sorting**: An essential part of this approach is to sort the `nums` array. Sorting helps us with a greedy-like approach to keep adding the smallest elements to reach the subsequence sum.
- **Prefix Sums**: The prefix sum array is constructed to have quick access to the sum of numbers from the start of the `nums` array up to any given point.
- **Binary Search**: The `bisect_right` function helps us to quickly find the rightmost location where a given query value would fit in the prefix sum array, which translates directly to the answer of how many elements can be used.
- **Greedy Approach**: The sorted array in combination with the prefix sums represents a greedy choice to choose the smallest elements to form subsequences.

The overall complexity of the algorithm is dominated by the sort operation, which is (O(n \log n)), where `n` is the length of the `nums` array. The prefix sum and binary searches for each query are (O(n)) and (O(\log n)) respectively, which are eclipsed by the sort operation when analyzing overall algorithm performance.

Example Walkthrough

Let's take small example arrays to walk through the solution approach described.

Suppose we have `nums = [3, 7, 5, 6]` and `queries = [7, 12, 5]`.

1. **Sort the Input Array**: We sort `nums` in ascending order.

   ```
   nums.sort()  # nums becomes [3, 5, 6, 7]
   ```

2. **Calculate Prefix Sums**: Next, we calculate the prefix sums of the sorted `nums` array.

   ```
   s = list(accumulate(nums))  # s becomes [3, 8, 14, 21]
   ```

3. **Search with Binary Search**: We then perform binary searches on `s` by using `bisect_right` for each query.

   For the query value `7`:

   ```
   bisect_right(s, 7)  # returns 2 because '7' could be inserted at index 2 to keep 's' sorted
   ```

   The subsequence `[3, 5]` sums up to `8`, which is greater than `7`. Thus, we only take the first element `[3]`, resulting in a max subsequence length of `1`.

   For the query value `12`:

   ```
   bisect_right(s, 12)  # returns 3 because '12' could be inserted at index 3 to keep 's' sorted
   ```

   The subsequence `[3, 5, 6]` sums up to `14`, exceeding `12`. We exclude the last element and get `[3, 5]`, resulting in a max subsequence length of `2`.

   For the query value `5`:

   ```
   bisect_right(s, 5)  # returns 1 because '5' could be inserted at index 1 to keep 's' sorted
   ```

   Here, only the first element `[3]` is less than or equal to `5`, giving us a max subsequence length of `1`.

The final answer for this example would be an array `[1, 2, 1]` corresponding to the max subsequence lengths for each query.

Python Solution

```python
from bisect import bisect_right
from itertools import accumulate

class Solution:
    def answerQueries(self, nums: List[int], queries: List[int]) -> List[int]:
        # Sort the array 'nums' to facilitate prefix sum calculation and binary search.
        nums.sort()

        # Calculate the prefix sums of the sorted array.
        # 'prefix_sums' will hold the sum of numbers from start to the current index.
        prefix_sums = list(accumulate(nums))

        # Process each query and find out how many numbers in the sorted array
        # have a sum less than or equal to the query number using binary search.
        # This is done by finding the rightmost index to insert the query number
        # in the prefix sum array such that it remains sorted.
        # The result is stored in a list.
        return [bisect_right(prefix_sums, query) for query in queries]

```

Java Solution

```java
class Solution {
    public int[] answerQueries(int[] nums, int[] queries) {
        // First, sort the array of nums.
        Arrays.sort(nums);

        // Create a prefix sum array using the sorted nums array.
        for (int i = 1; i < nums.length; ++i) {
            nums[i] += nums[i - 1];
        }

        // The length of the queries array.
        int numQueries = queries.length;

        // Initialize the answer array to store the results of each query.
        int[] answers = new int[numQueries];

        // Process each query using the search method.
        for (int i = 0; i < numQueries; ++i) {
            answers[i] = search(nums, queries[i]);
        }
        return answers;
    }

    // Perform a binary search to find the maximum length of a non-empty
    // subsequence that is less than or equal to the query value.
    private int search(int[] prefixSums, int targetValue) {
        int left = 0, right = prefixSums.length;

        // While the search space is valid
        while (left < right) {
            // Find the middle index of the current search space
            int mid = (left + right) >> 1;

            // If the subsequence sum at mid is greater than the target value,
            // we need to look to the left half of the search space.
            if (prefixSums[mid] > targetValue) {
                right = mid;
            } else {
                // Otherwise, we look to the right half of the search space.
                left = mid + 1;
            }
        }

        // The binary search returns the maximum length of a non-empty
        // subsequence whose sum is less than or equal to the target value.
        return left;
    }
}

```

C++ Solution

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

class Solution {
public:
    // Method to answer the queries based on the prefix sums of the sorted nums array
    vector<int> answerQueries(vector<int>& nums, vector<int>& queries) {
        // First, sort the input 'nums' array in non-decreasing order
        sort(nums.begin(), nums.end());

        // Create prefix sums in place
        // After this loop, each element at index 'i' in 'nums' will represent the sum of all
        // elements from index 0 to 'i'
        for (int i = 1; i < nums.size(); i++) {
            nums[i] += nums[i - 1];
        }

        // Initialize the vector to store the answers to the queries
        vector<int> answers;

        // Iterate through each query
        for (const auto& query : queries) {
            // 'upper_bound' returns an iterator pointing to the first element that is greater than 'query'
            // Subtracting 'nums.begin()' from the iterator gives the number of elements that can be summed without exceeding 'query'
            answers.push_back(upper_bound(nums.begin(), nums.end(), query) - nums.begin());
        }

        // Return the final answers for the queries
        return answers;
    }
};

```

Typescript Solution

```typescript
/**
 * Process and answer a set of queries based on prefix sums of a sorted array.
 *
 * @param nums The initial array of numbers.
 * @param queries An array of query values.
 * @returns An array of results, each representing the maximum length of
 *          a non-empty contiguous subarray that has a sum less than or
 *          equal to the query value.
 */
function answerQueries(nums: number[], queries: number[]): number[] {
    // Sort the nums array in ascending order
    nums.sort((a, b) => a - b);

    // Calculate prefix sums in place, each element becomes the sum of all
    // previous elements in the sorted array
    for (let i = 1; i < nums.length; i++) {
        nums[i] += nums[i - 1];
    }

    // Answer array to hold the maximum lengths per query
    const answers: number[] = [];

    // Binary search function to find out the maximum length of subarray
    // for a single query
    const binarySearch = (prefixSums: number[], target: number): number => {
        let left = 0;
        let right = prefixSums.length;

        // Perform a binary search to find the right position where the sum
        // exceeds the query value
        while (left < right) {
            const mid = Math.floor((left + right) / 2);
            if (prefixSums[mid] > target) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }

        // Return the index, which represents the maximum length of subarray
        return left;
    };

    // Iterate through each query and use the binary search function
    // to find the maximum length of subarray that fits the query condition
    for (const query of queries) {
        answers.push(binarySearch(nums, query));
    }

    // Return the final answers array with maximum lengths per query
    return answers;
}

```

Time and Space Complexity

The given code snippet defines a function `answerQueries` which takes a list of numbers `nums` and a list of queries `queries`, then returns a list of integers. Let's analyze the time and space complexity of this function:

Time Complexity

1. `nums.sort()`: Sorting the `nums` list has a time complexity of `O(n log n)`, where `n` is the length of `nums`.
2. `list(accumulate(nums))`: The `accumulate` function computes the prefix sums of the sorted `nums` list. Since it goes through each element once, it has a time complexity of `O(n)`.
3. `[bisect_right(s, q) for q in queries]`: For each query `q` in `queries`, the `bisect_right` function performs a binary search on the list `s` of prefix sums. Each binary search operation has a time complexity of `O(log n)`, and there are `m` queries, so the overall time complexity for this step is `O(m log n)`, where `m` is the length of `queries`.

Combining these, the overall time complexity is `O(n log n) + O(n) + O(m log n)`. Since `O(n log n)` is the dominant term, the time complexity simplifies to `O(n log n + m log n)`.

Space Complexity

1. Sorting the list `nums` in-place doesn't require additional space, so its space complexity is `O(1)`.
2. The list `s` of prefix sums would require additional space proportional to the length of `nums`. Hence, the space complexity for this step is `O(n)`.
3. The list comprehension for bisect_right does not use additional space apart from the output list, which contains `m` elements. So, the space complexity for the output list is `O(m)`.

Since `O(n)` and `O(m)` are not part of the same operation, we consider both for the overall space complexity. Thus, the overall space complexity of the function is `O(n + m)`.