2389. Longest Subsequence With Limited Sum

Sorting

Binary Search Prefix Sum

# **Problem Description**

array) to reach the sum required in the queries.

**Greedy** Array

Easy

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

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

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

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 ith 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 <u>prefix sum</u> array, we can easily determine, for each query, how many elements can be taken from the **nums** 

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

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

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

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

Calculate Prefix Sums: We then compute a prefix sum array s from the sorted array. The prefix sum is a common technique

**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

**Sorting:** An essential part of this approach is to sort the nums array. Sorting helps us with a greedy-like approach to keep

Binary Search: The bisect\_right function helps us to quickly find the rightmost location where a given query value would fit

**Greedy Approach:** The sorted array in combination with the prefix sums represents a greedy choice to choose the smallest

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

### 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.

nums.sort()

**Solution Approach** 

subsequence for any given sum.

s = list(accumulate(nums))

equivalent to the length of the required subsequence.

sort operation when analyzing overall algorithm performance.

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

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

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

[bisect\_right(s, q) for q in queries]

The implementation uses the following concepts:

elements to form subsequences.

possible subsequence is avoided, leading to a much more efficient solution.

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

adding the smallest elements to reach the subsequence sum. **Prefix Sums**: The <u>prefix sum</u> array is constructed to have quick access to the sum of numbers from the start of the nums array up to any given point.

in the prefix sum array, which translates directly to the answer of how many elements can be used.

- **Example Walkthrough** Let's take small example arrays to walk through the solution approach described.
  - Calculate Prefix Sums: Next, we calculate the prefix sums of the sorted nums array. s = list(accumulate(nums)) # s becomes [3, 8, 14, 21]

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

Search with Binary Search: We then perform binary searches on s by using bisect\_right for each query.

bisect\_right(s, 7) # returns 2 because '7' could be inserted at index 2 to keep 's' sorted

bisect\_right(s, 12) # returns 3 because '12' could be inserted at index 3 to keep 's' sorted

# For the query value 12:

For the query value 7:

subsequence length of 1.

For the query value 5:

Solution Implementation

nums.sort()

**Python** 

Java

class Solution {

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.

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

bisect\_right(s, 5) # returns 1 because '5' could be inserted at index 1 to keep 's' sorted

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

from bisect import bisect right from itertools import accumulate class Solution:

def answerQueries(self, nums: List[int], queries: List[int]) -> List[int]:

# Calculate the prefix sums of the sorted array.

public int[] answerQueries(int[] nums, int[] queries) {

# in the prefix sum array such that it remains sorted.

// Create a prefix sum array using the sorted nums array.

// Perform a binary search to find the maximum length of a non-empty

// Find the middle index of the current search space

// we need to look to the left half of the search space.

// The binary search returns the maximum length of a non-empty

// subsequence whose sum is less than or equal to the target value.

// If the subsequence sum at mid is greater than the target value,

// Otherwise, we look to the right half of the search space.

return [bisect\_right(prefix\_sums, query) for query in queries]

prefix\_sums = list(accumulate(nums))

# The result is stored in a list.

// First, sort the array of nums.

nums[i] += nums[i - 1];

// The length of the gueries array.

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

int left = 0, right = prefixSums.length;

int mid = (left + right) >> 1;

if (prefixSums[mid] > targetValue) {

// Answer array to hold the maximum lengths per query

const mid = Math.floor((left + right) / 2);

if (prefixSums[mid] > target) {

answers.push(binarySearch(nums, query));

// Return the final answers array with maximum lengths per query

# Calculate the prefix sums of the sorted array.

# in the prefix sum array such that it remains sorted.

return [bisect\_right(prefix\_sums, query) for query in queries]

prefix\_sums = list(accumulate(nums))

# The result is stored in a list.

def answerQueries(self, nums: List[int], queries: List[int]) -> List[int]:

# Sort the array 'nums' to facilitate prefix sum calculation and binary search.

# 'prefix sums' will hold the sum of numbers from start to the current index.

# 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 guery number

// Binary search function to find out the maximum length of subarray

const binarySearch = (prefixSums: number[], target: number): number => {

// Perform a binary search to find the right position where the sum

// Return the index, which represents the maximum length of subarray

const answers: number[] = [];

let right = prefixSums.length;

// exceeds the query value

right = mid;

left = mid + 1;

while (left < right) {</pre>

for (const query of queries) {

} else {

return left;

return answers;

class Solution:

**Time Complexity** 

it has a time complexity of O(n).

complexity for the output list is O(m).

from bisect import bisect right

from itertools import accumulate

nums.sort()

// for a single query

let left = 0;

answers[i] = search(nums, queries[i]);

int numQueries = queries.length;

for (int i = 1; i < nums.length; ++i) {</pre>

Arrays.sort(nums);

return answers;

while (left < right) {</pre>

} else {

return left;

right = mid;

left = mid + 1;

# Sort the array 'nums' to facilitate prefix sum calculation and binary search.

# 'prefix sums' will hold the sum of numbers from start to the current index.

# 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

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

## // subsequence that is less than or equal to the query value. private int search(int[] prefixSums, int targetValue) { // While the search space is valid

```
C++
#include <vector>
#include <algorithm>
class Solution {
public:
    // Method to answer the gueries based on the prefix sums of the sorted nums array
    vector<int> answerOueries(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++) {</pre>
            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& guery : gueries) {
            // '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
/**
 * 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) \Rightarrow 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++) {</pre>
        nums[i] += nums[i - 1];
```

```
};
// Iterate through each guery and use the binary search function
// to find the maximum length of subarray that fits the query condition
```

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:

2. list(accumulate(nums)): The accumulate function computes the prefix sums of the sorted nums list. Since it goes through each element once,

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.

1. nums.sort(): Sorting the nums list has a time complexity of O(n log n), where n is the length of nums.

- Combining these, the overall time complexity is  $0(n \log n) + 0(n) + 0(m \log n)$ . Since  $0(n \log n)$  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 0(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 0(n).

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).

3. The list comprehension for bisect\_right does not use additional space apart from the output list, which contains m elements. So, the space