491. Non-decreasing Subsequences

Medium Bit Manipulation Array Hash Table **Backtracking**

Problem Description

step, we have two choices:

sequence that can be obtained from another sequence by deleting some or no elements without changing the order of the remaining elements. The subsequences we are looking for should be non-decreasing, meaning each element in the subsequence less than or equal to the subsequent element. Also, each subsequence must contain at least two elements. Unlike combinations or subsets, the order is important here, so sequences with the same elements but in different orders are considered different.

The problem is asking us to find all the different possible subsequences of a given array of integers nums. A subsequence is a

Intuition

The intuition behind the solution is to explore all possible subsequences while maintaining the non-decreasing order constraint.

Include the current element in the subsequence if it's greater than or equal to the last included element. This is to ensure the

We can perform a depth-first search (DFS) to go through all potential subsequences. We'll start with an empty list and at each

non-decreasing order. Skip the current element to consider a subsequence without it.

- However, to avoid duplicates, if the current element is the same as the last element we considered and decided not to include, we skip the current element. This is because including it would result in a subsequence we have already considered.
- Starting from the first element, we will recursively call the DFS function to traverse the array. If we reach the end of the array, and

our temporary subsequence has more than one element, we include it in our answer. The key component of this approach is how we handle duplicates to ensure that we only record unique subsequences while

Solution Approach The implementation of the solution uses a recursive approach known as Depth-First Search (DFS). Let's break down how the given Python code functions:

The function dfs is a recursive function used to perform the depth-first search, starting from the index u in the nums array.

This function has the parameters u, which is the current index in the array; last, which is the last number added to the current subsequence t; and t, which represents the current subsequence being constructed.

to the answer list ans.

not include this element.

least two elements, fulfilling the problem's requirement.

can always be considered for starting a new subsequence.

ensuring that no duplicates are generated.

To summarize, our DFS explores these paths:

performing our DFS.

At the beginning of the dfs function, we check if u equals the length of nums. If it does, we have reached the end of the array. At this point, if the subsequence t has more than one element (making it a valid subsequence), we append a copy of it

If the current element, nums[u], is greater than or equal to the last element (last) included in our temporary subsequence (t), we can choose to include the current element in the subsequence by appending it to t and recursively calling dfs with the next index (u + 1) and the current element as the new last.

After returning from the recursive call, the element added is popped from to backtrack and consider subsequences that do

Additionally, to avoid duplicates, if the current element is different from the last element, we also make a recursive call to dfs

At the end of the call to the dfs from the main function, ans will contain all possible non-decreasing subsequences of at

without including the current element in the subsequence t, regardless of whether it could be included under the nondecreasing criterion. The ans list collects all valid subsequences. The initial DFS call is made with the first index (0), a value that's lower than any

element of the array (-1000 in this case) as the initial last value, and an empty list as the initial subsequence.

- Key elements in this solution are the handling of backtracking by removing the last appended element after the recursive calls, and the checking mechanism to avoid duplicates. The choice of the initial last value is crucial. It must be less than any element we expect in nums, ensuring that the first element
- ans: A list to store all the valid non-decreasing subsequences that have at least two elements. • t: A temporary list used to build each potential subsequence during the depth-first search.

Overall, the solution effectively explores all combinations of non-decreasing subsequences through the depth-first search while

Let's walk through an example to illustrate the solution approach using the given array of integers nums = [1, 2, 2]. 1. Initialize ans as an empty list to store our subsequences and t as an empty list to represent the current subsequence.

2. Start with the first element 1. Since 1 is greater than our initial last value -1000, we can include 1 in t (which is now [1]) and proceed to

5. At the third element (also 2), we check if we just skipped an element with the same value (which we did). If so, we do not include this element

to avoid a duplicate subsequence. If not, since 2 is equal or greater than the last element in t, we could include it in t, and add the resulting

6. Instead, we proceed without including this third element 2. Since we have finished going through the array, and t has less than two elements,

3. At the second element 2, it's greater than the last element in t (which is 1), so we can include 2 in t (now [1, 2]) and proceed to the next element. Now our t is a valid subsequence, so we can add it to ans.

the next index.

Data structures:

Example Walkthrough

we don't add it to ans. 7. Our final ans list contains [1, 2], representing the valid non-decreasing subsequences with at least two elements.

4. Backtrack by popping 2 from t (now [1]) and proceed without including the second element 2.

subsequence [1, 2] to ans again. But since we are skipping duplicates, we do not do this.

the second 2, ensuring our final ans list only included unique non-decreasing subsequences.

Save a copy of the current sequence if it's a valid subsequence

Base case: When we have traversed all elements

subsequences.append(temp_seq[:])

subsequences = [] # To store all the valid subsequences

public List<List<Integer>> findSubsequences(int[] nums) {

return subsequences; // Return the list of subsequences

dfs(0. Integer.MIN VALUE, new ArrayList<>());

1. It defines a method `findSubsequences` which accepts a list of integers.

If the current number can be included in the subsequence

backtrack(0, float('-inf'), []) # Kick-off the backtracking process

private int[] sequence; // Renamed from 'nums' to 'sequence' for better clarity

this sequence = nums; // Assign the given array to the class variable

if (index == nums.size()) { // Base case: reached the end of nums

subsequences.push_back(currentSubsequence);

// This avoids duplicates in the subsequences list.

// Function to find all the increasing subsequences in the given array.

// Call the backtrack function to start processing the subsequences

// lastNumber is the last number added to the current subsequence.

// currentSubsequence holds the current subsequence being explored.

// subsequences is the collection of all valid subsequences found.

subsequences.push([...currentSubsequence]);

if (index === nums.length) { // Base case: reached the end of nums

backtrack(0, Number.MIN_SAFE_INTEGER, nums, currentSubsequence, subsequences);

return;

if (nums[index] >= lastNumber) {

if (nums[index] != lastNumber) {

function findSubsequences(nums: number[]): number[][] {

let subsequences: number[][] = [];

return subsequences;

return;

let currentSubsequence: number[] = [];

// Uses backtracking to find all subsequences.

// index is the current index in nums.

// nums is the input array of numbers.

private List<List<Integer>> subsequences; // List to store the answer subsequences

subsequences = new ArrayList<>(); // Initialize the list to store subsequences

if start index == len(nums):

if len(temp seg) > 1:

if nums[start index] >= prev num:

if nums[start index] != prev num:

return

of array, not enough elements). • The end result for ans is [[1, 2]].

• [1] → [1, 2] (added to ans) → [1] (backtrack) → [1] (skip the second 2) → [1, 2] (skipped because it would be a duplicate) → [1] (end

The key part of this example is that our DFS allowed us to include the first 2, but by using the duplicate check, we did not include

- Solution Implementation **Python**
- class Solution: def findSubsequences(self, nums: List[int]) -> List[List[int]]: def backtrack(start index, prev num, temp seq):

temp seq.append(nums[start index]) # Include the current number backtrack(start index + 1, nums[start index], temp seq) # Recursion with the updated last element temp_seq.pop() # Backtrack and remove the last element

2. It uses backtracking to explore all subsequences, only adding those that are non-decreasing and have a length greater than 1 to

3. The `backtrack` helper function recursively constructs subsequences, avoiding duplicates by not revisiting the same number at ea

// Start the Depth—First Search (DFS) from index 0 with the last picked element as the smallest integer value

To ensure we do not add duplicates, move on if the current number equals the previous number

backtrack(start_index + 1, prev_num, temp_seq) # Recursion without the current number

```
Java
```

import java.util.ArrayList;

import java.util.List;

class Solution {

return subsequences

The provided code does the following:

from typing import List

```
// Helper method to perform DFS
   private void dfs(int index. int lastPicked. List<Integer> currentSubsequence) {
       // Base case: if we've reached the end of the sequence
        if (index == sequence.length) {
            // Check if the current list is a subsequence with more than one element
            if (currentSubsequence.size() > 1) {
                // If it is, add a copy of it to the list of subsequences
                subsequences.add(new ArrayList<>(currentSubsequence));
            return; // End the current DFS path
       // If the current element can be picked (is greater or equal to the last picked element)
        if (sequence[index] >= lastPicked) {
            // Pick the current element by adding it to the currentSubsequence
            currentSubsequence.add(sequence[index]);
            // Continue the DFS with the next index and the new lastPicked element
            dfs(index + 1, sequence[index], currentSubsequence);
            // Backtrack: remove the last element added to the currentSubsequence
            currentSubsequence.remove(currentSubsequence.size() - 1);
       // Perform another DFS to explore the possibility of not picking the current element
       // Only if the current element isn't equal to the last picked one to avoid duplicates
        if (sequence[index] != lastPicked) {
            dfs(index + 1, lastPicked, currentSubsequence);
class Solution {
public:
   // Function to find all the increasing subsequences in the given vector.
   vector<vector<int>> findSubsequences(vector<int>& nums) {
        vector<vector<int>> subsequences;
       vector<int> currentSubsequence;
        backtrack(0, INT MIN, nums, currentSubsequence, subsequences); // Assuming -1000 is a lower bound, we use INT_MIN
       return subsequences;
private:
   // Uses backtracking to find all subsequences.
   // u is the current index in nums.
   // last is the last number added to the current subsequence.
   // nums is the input array of numbers.
   // currentSubsequence holds the current subsequence being explored.
    // subsequences is the collection of all valid subsequences found.
    void backtrack(int index, int lastNumber, vector<int>& nums, vector<int>& currentSubsequence, vector<vector<int>>& subsequences)
```

if (currentSubsequence.size() > 1) $\{ // If the subsequence has more than 1 element, add it to the answer.$

// If the current number can be added to the subsequence according to the problem definition (non-decreasing order)

backtrack(index + 1, nums[index], nums, currentSubsequence, subsequences); // Recursively call with next index.

backtrack(index + 1, lastNumber, nums, currentSubsequence, subsequences); // Recursively call with next index.

function backtrack(index: number, lastNumber: number, nums: number[], currentSubsequence: number[], subsequences: number[][]): void {

if (currentSubsequence.length > 1) { // If the subsequence has more than 1 element, add it to the answer.

currentSubsequence.push back(nums[index]); // Add number to the current subsequence.

currentSubsequence.pop_back(); // Backtrack: remove the number from current subsequence.

// If current number is not equal to the last number added to the subsequence, continue to next index.

TypeScript

```
// If the current number can be added to the subsequence according to the problem definition (non-decreasing order)
    if (nums[index] >= lastNumber) {
        currentSubsequence.push(nums[index]): // Add number to the current subsequence.
        backtrack(index + 1, nums[index], nums, currentSubsequence, subsequences); // Recursively call with the next index.
        currentSubsequence.pop(); // Backtrack: remove the number from current subsequence.
    // If current number is not equal to the last number added to the subsequence, continue to next index.
    // This avoids duplicates in the subsequences list.
    if (nums[index] !== lastNumber) {
        backtrack(index + 1, lastNumber, nums, currentSubsequence, subsequences); // Recursively call with the next index.
// Example usage:
const input: number[] = [4, 6, 7, 7];
const output: number[][] = findSubsequences(input);
console.log(output); // Output the increasing subsequences.
from typing import List
class Solution:
    def findSubsequences(self, nums: List[int]) -> List[List[int]]:
        def backtrack(start index, prev num, temp seq):
            # Base case: When we have traversed all elements
            if start index == len(nums):
                # Save a copy of the current sequence if it's a valid subsequence
                if len(temp sea) > 1:
                    subsequences.append(temp_seq[:])
                return
           # If the current number can be included in the subsequence
            if nums[start index] >= prev num:
                temp seq.append(nums[start index]) # Include the current number
                backtrack(start index + 1, nums[start index], temp seq) # Recursion with the updated last element
                temp_seq.pop() # Backtrack and remove the last element
           # To ensure we do not add duplicates, move on if the current number equals the previous number
            if nums[start index] != prev num:
                backtrack(start_index + 1, prev_num, temp_seq) # Recursion without the current number
        subsequences = [] # To store all the valid subsequences
        backtrack(0, float('-inf'), []) # Kick-off the backtracking process
        return subsequences
# The provided code does the following:
 1. It defines a method `findSubsequences` which accepts a list of integers.
# 2. It uses backtracking to explore all subsequences, only adding those that are non-decreasing and have a length greater than 1 to
# 3. The `backtrack` helper function recursively constructs subsequences, avoiding duplicates by not revisiting the same number at ea
Time and Space Complexity
```

non-decreasing order constraint). Given that we have n elements in nums, in the worst case, each element might participate in the recursion twice—once when it is

Space Complexity

Time Complexity

condition if nums[u] != last prevents some recursive calls when the previous number is the same as the current, which could lead to some pruning, but this pruning does not affect the worst-case complexity, which remains exponential. Therefore, the time complexity of the code is $O(2^n)$.

included and once when it is excluded. This gives us an upper bound of 0(2^n) on the number of recursive calls. However, the

The time complexity of the given code mainly depends on the number of recursive calls it can potentially make. At each step, it

has two choices: either include the current element in the subsequence or exclude it (as long as including it does not violate the

The space complexity consists of two parts: the space used by the recursion call stack and the space used to store the combination t. The space used by the recursion stack in the worst case would be 0(n) because that's the maximum depth the recursive call

stack could reach if you went all the way down including each number one after the other.

The space required for storing the combination t grows as the recursion deepens, but since the elements are only pointers or references to integers and are reused in each recursive call, this does not significantly contribute to the space complexity. However, the temporary arrays formed during the process, which are then copied to ans, could increase the storage requirements. ans itself can grow up to 0(2ⁿ) in size, in the case where every possible subsequence is valid.

Thus, the space complexity is dominated by the size of the answer array ans and the recursive call stack, leading to a total space complexity of 0(n * 2^n), with n being the depth of the recursion (call stack) and 2^n being the size of the answer array in the worst case.