39. Combination Sum

Medium Array Backtracking

Problem Description

This problem presents us with a challenge: given a list of distinct integers called `candidates` and a target integer called `target`, we need to find all unique combinations of these candidates that add up to the target. The crucial part to note is that each number from `candidates` can be used repeatedly—an unlimited number of times.

A combination is considered unique if the count of at least one of the numbers in it differs from that in any other combination. The result can be in any order and doesn't have to be sorted.

To narrow down the possibilities, the problem assures us that the total number of unique combinations won't exceed 150 for the input provided.

Intuition

The intuition behind solving this problem involves employing a classic backtracking approach called Depth-First Search (DFS). The idea is to explore all possible combinations that can add up to the target by building them incrementally and backtracking whenever we either achieve the target sum or exceed it.

1. **Sorting Candidates**: We start by sorting the list of candidates. While not necessary for the solution's correctness, this step can optimize the process by allowing us to stop early when the current candidate exceeds the remaining target sum.
2. **DFS Function**: We define a recursive function, `dfs`, to explore different combinations. The function takes two parameters: `i` representing the index of the current candidate and `s` for the remaining sum we need to reach the target. Within this function, we have two base cases:
   - If `s` becomes `0`, we've found a valid combination and add a copy of the current combination to the list of answers.
   - If `i` is out of bounds or the current candidate exceeds `s`, we cannot make a combination with the present choices and need to backtrack.
3. **Exploring Candidates**: At each step, we have two choices:
   - Skip the current candidate and proceed to the next one using `dfs(i + 1, s)`.
   - Include the current candidate in the current combination, adjust the remaining sum `s` by subtracting the candidate's value, and stay at the current index assuming we can pick it again with `dfs(i, s - candidates[i])`. We also need to add the included candidate to the temporary list `t`.
4. Backtracking: After exploring the option with the current candidate included, we backtrack by removing it from the combination using `t.pop()` to restore the state for other explorations.
5. **Avoiding Duplicates**: Sorting candidates and ensuring that we always start with the smallest possible candidate for each slot in a combination avoid duplication naturally because each combination will be built up in a sorted manner.
6. **Execution**: We initialize the list to store our temporary and final answers, `t` and `ans`, respectively, and start our DFS with `dfs(0, target)`.

This approach ensures that we thoroughly explore all viable combinations of candidates that can add up to `target`, respecting the constraint of unique frequency for each number within the combinations.

Solution Approach

The implementation of the solution utilizes a `dfs` function that orchestrates a recursive depth-first search to explore different combinations. Let's dissect how the algorithm, data structures, and patterns work in tandem:

1. **Algorithm**: The key algorithmic technique here is backtracking, which is a type of DFS. Backtracking is a systematic way to iterate through all the possible configurations of a search space. It is designed to decompose a complex problem into simpler subproblems. If the current solution is not workable or optimal, it's abandoned (hence the term 'backtrack').
2. **Data Structures**: We use two lists in this solution - `ans` to store the final list of combinations and `t` as a temporary list to keep an ongoing combination. These lists are essential for the search to build the result and to maintain the state during recursion.
3. **Patterns**: The implementation follows a common pattern found in DFS/backtracking solutions called "decide-explore-un-decide". At every step, we make a decision (include the current candidate or not), explore further down the path (recursive call), and then undo the decision (pop the candidate).

Let's review the implementation step-by-step with a reference to the code:

```python
def dfs(i: int, s: int):
    if s == 0:
        ans.append(t[:])  # Found valid combination
        return
    if i >= len(candidates) or s < candidates[i]:
        return  # Cannot go further, backtrack
    dfs(i + 1, s)  # Skip current candidate and explore further
    t.append(candidates[i])  # Include current candidate
    dfs(i, s - candidates[i])  # Continue with the included candidate
    t.pop()  # Remove the last candidate and explore other possibilities
```

This block of code is the heart of the solution. The `dfs` function is called recursively to explore each candidate:

- **Checking for Completion**: If `s` is `0`, we've matched our target, so we copy the current list `t` to our answer list `ans`.
- **Termination Conditions**: If `i` is out of range or the current candidate is larger than `s`, we cannot find a valid combination on this path, so we return to explore a different path.
- **Exploring Without the Current Candidate**: If we call `dfs(i + 1, s)`, we're exploring possibilities without the current candidate.
- **Including the Current Candidate**: By adding `candidates[i]` to `t` before making the recursive call, we explore the option where we include the current candidate.
- Backtracking: After the recursive call that includes the candidate returns, we pop that candidate from `t`. This step is essential, as it brings us back to the decision point where we did not include the candidate - all set for the next iteration.

The initial call to `dfs(0, target)` kickstarts the exploration. Given that we sorted `candidates` at the beginning, the DFS will proceed from the smallest to the largest candidate, which again helps in avoiding duplicates and unnecessary explorations. As we accumulate combinations, they are stored in the `ans` list, which we eventually return as the result.

In summary, the solution implements a backtracking DFS strategy to generate all possible unique combinations of numbers that add up to a target sum. This method is both efficient and a classic example of how problems involving combinations can be solved.

Example Walkthrough

Let's use a small example to illustrate the solution approach with the `candidates` list as `[2, 3, 6]` and a `target` of `7`.

1. **Sorting Candidates**: Firstly, the candidates list `[2, 3, 6]` is sorted, yielding the same list as it is already sorted.
2. **Initial Call**: We begin the search by calling `dfs(0, 7)` implying that we start with the first candidate and the target sum of `7`.
3. **First Exploration**:
   - We did not reach a base case yet, so we try `dfs(0 + 1, 7)` which is `dfs(1, 7)`. We're skipping the first candidate (2) and exploring the next candidate (3).
   - Simultaneously, we try the other path where we include the first candidate (2) by calling `dfs(0, 5)`, since `7 - 2 = 5`. Our temporary combination list `t` now has `[2]`.
4. **Exploration with Second Candidate**:
   - In the call for `dfs(1, 7)`, we repeat the process. Again, it splits into two recursive calls: `dfs(2, 7)` (skipping the current candidate, 3) and `dfs(1, 4)` (including the current candidate, 3). Now `t = [3]`.
   - Meanwhile, exploring the other branch with `dfs(0, 5)`, we again include the first candidate, making the recursive call `dfs(0, 3)`. The list `t` is now `[2, 2]`.
5. **Finding a Combination**:
   - Continuing with `dfs(0, 3)`, we yet again choose to include the first candidate. Now, `dfs(0, 1)` is called and `t = [2, 2, 2]`.
   - With `dfs(0, 1)`, we can no longer include 2 as it's greater than the target, so we explore with the next candidate by calling `dfs(1, 1)`. Since there isn't a candidate with value `1`, this path doesn't yield a valid combination, and we backtrack.
   - Backtracking to `dfs(0, 3)`, we pop out the last candidate and try `dfs(1, 3)`. Since 3 is a candidate, we find a combination `[2, 2, 3]` which adds up to `7`, so that is added to our `ans` list.
6. **Continued Exploration and Backtracking**:
   - We continue this process, exploring different permutations and excluding those that exceed the target sum. All the while, we backtrack correctly by removing the last added candidate from `t` upon going up a level in the search tree.
7. **Result**: Eventually, we end up with the `ans` list containing all unique combinations that sum up to `7`, which could be `[[2, 2, 3]]` in this small example.

The complete process involved recursively selecting candidates until we reached the target or exceeded it. This methodically covered all possible unique combinations, ensuring that we met the problem's conditions effectively.

Python Solution

```python
from typing import List  # Import necessary List type from the typing module for type annotation

class Solution:
    def combinationSum(self, candidates: List[int], target: int) -> List[List[int]]:
        # Helper function to perform depth-first search for combinations
        def dfs(index: int, current_sum: int):
            # Check if the current combination equals target sum
            if current_sum == 0:
                # If current sum is zero, we found a valid combination, add it to the answer list
                combinations.append(combination_so_far[:])
                return
            if index >= len(candidates) or current_sum < candidates[index]:
                # If we've reached the end of candidates array or current_sum
                # is less than the candidate at the index, stop exploring this path
                return
            # Recurse without including the current candidate
            dfs(index + 1, current_sum)
            # Choose the current candidate
            combination_so_far.append(candidates[index])
            # Recurse including the current candidate
            dfs(index, current_sum - candidates[index])
            # Backtrack by removing the current candidate
            combination_so_far.pop()

        # Sort the candidates to help with early stopping in dfs
        candidates.sort()
        # This is a temporary list to hold the current combination
        combination_so_far = []
        # This is the list to hold all unique combinations that sum up to target
        combinations = []
        # Start the dfs from the first candidate with the summation equal to target
        dfs(0, target)
        # Return all unique combinations found
        return combinations

```

Java Solution

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

class Solution {
    private List<List<Integer>> combinations = new ArrayList<>(); // Store the list of all combinations
    private List<Integer> currentCombination = new ArrayList<>(); // Current combination being explored
    private int[] candidateNumbers; // Array of candidate numbers

    // Method to find all unique combinations where the candidate numbers sum up to target
    public List<List<Integer>> combinationSum(int[] candidates, int target) {
        Arrays.sort(candidates); // Sort the array of candidates to optimize the process
        this.candidateNumbers = candidates; // Store the global reference for candidate numbers
        backtrack(0, target);
        return combinations;
    }

    // Helper method to perform the depth-first search
    private void backtrack(int startIndex, int remainingSum) {
        if (remainingSum == 0) {
            // If the remaining sum is 0, we found a valid combination
            combinations.add(new ArrayList<>(currentCombination));
            return;
        }
        if (startIndex >= candidateNumbers.length || remainingSum < candidateNumbers[startIndex]) {
            // If startIndex is out of bounds or the smallest candidate exceeds remainingSum
            return;
        }

        // Skip the current candidate and move to the next one
        backtrack(startIndex + 1, remainingSum);

        // Include the current candidate in the current combination
        currentCombination.add(candidateNumbers[startIndex]);
        // Continue exploring with the current candidate (since we can use the same number multiple times)
        backtrack(startIndex, remainingSum - candidateNumbers[startIndex]);
        // Backtrack and remove the last element before trying the next candidate
        currentCombination.remove(currentCombination.size() - 1);
    }
}

```

C++ Solution

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

class Solution {
public:
    // Finds all unique combinations of candidates that sum up to the given target
    std::vector<std::vector<int>> combinationSum(std::vector<int>& candidates, int target) {
        // Sort the candidates to improve efficiency and ensure combinations are found in order
        std::sort(candidates.begin(), candidates.end());

        std::vector<std::vector<int>> combinations; // Final result; a list of all combinations
        std::vector<int> currentCombo;              // Current combination being explored

        // The recursive function to perform depth-first search to find all combinations
        std::function<void(int, int)> depthFirstSearch = [&](int index, int remaining) {
            // If remaining sum is zero, we have found a valid combination
            if (remaining == 0) {
                combinations.emplace_back(currentCombo);
                return;
            }
            // If we've exhausted all candidates or the remaining sum is too small, backtrack
            if (index >= candidates.size() || remaining < candidates[index]) {
                return;
            }

            // Skip the current candidate and move to the next one
            depthFirstSearch(index + 1, remaining);

            // Include the current candidate and continue searching with reduced remaining sum
            currentCombo.push_back(candidates[index]);
            depthFirstSearch(index, remaining - candidates[index]);

            // Backtrack: remove the last candidate from the current combo
            currentCombo.pop_back();
        };

        // Start the depth-first search, starting from the first candidate and target sum
        depthFirstSearch(0, target);

        return combinations;
    }
};

```

Typescript Solution

```typescript
function combinationSum(candidates: number[], target: number): number[][] {
    // Sort the array to handle combinations in ascending order.
    candidates.sort((a, b) => a - b);

    // This will hold all unique combinations that sum up to the target.
    const combinations: number[][] = [];

    // Temporary array to store the current combination.
    const currentCombination: number[] = [];

    // Helper function to find all combinations.
    const findCombinations = (startIndex: number, remainingSum: number) => {
        // If remaining sum is zero, we found a valid combination.
        if (remainingSum === 0) {
            combinations.push(currentCombination.slice());
            return;
        }

        // If the startIndex is outside the bounds or the smallest candidate
        // is larger than the remaining sum, there's no point in exploring further.
        if (startIndex >= candidates.length || remainingSum < candidates[startIndex]) {
            return;
        }

        // Skip the current candidate and move to the next.
        findCombinations(startIndex + 1, remainingSum);

        // Include the current candidate and explore.
        currentCombination.push(candidates[startIndex]);
        findCombinations(startIndex, remainingSum - candidates[startIndex]);

        // Backtrack and remove the current candidate from the current combination.
        currentCombination.pop();
    };

    // Initialize the recursive function with the starting index and initial target sum.
    findCombinations(0, target);
    return combinations;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code primarily depends on the number of potential combinations that can be formed with the given `candidates` array that sum up to the `target`. Considering the array has a length `n` and the recursion involves iterating over candidates and including/excluding them, we get a recursion tree with a depth that could potentially go up to `target/min(candidates)`, if we keep using the smallest element. This leads to an exponential number of possibilities. Thus, the time complexity of the algorithm is `O(2^n)` in the worst case, when the recursion tree is fully developed. However, since we often return early when `s < candidates[i]`, this is an upper bound.

Space Complexity

The space complexity of the algorithm is also important to consider. It is mainly used by recursion stack space and the space to store combinations. The maximum depth of the recursion could be `target/min(candidates)` which would at most be `O(target)` if `1` is in the candidates. However, the space required for the list `t`, which is used to store the current combination, is also dependent on the target and could at most have `target` elements when 1 is in the candidates. The space for `ans` depends on the number of combinations found. Since it's hard to give an exact number without knowing the specifics of `candidates` and `target`, we consider it for the upper bound space complexity. Thus, as the list `ans` grows with each combination found, in the worst case, it could store a combination for every possible subset of `candidates`, leading to a space complexity of `O(2^n * target)`, where `2^n` is the number of combinations and `target` is the maximum size of any combination.

However, if we look at the auxiliary space excluding the space taken by the output (which is the space `ans` takes), then the space complexity is `O(target)` due to the depth of the recursive call stack and the temporary list `t`.