40. Combination Sum II

Problem Description

In this problem, we are given a set of candidate numbers (an array `candidates`) and a target number (`target`). Our goal is to find all unique combinations where the sum of the numbers in each combination equals the `target`. Importantly, we can only use each number in `candidates` once in any given combination.

Another important requirement is that the solution set must not include any duplicate combinations. Even though the same number can occur multiple times in the `candidates`, it can only appear once in each combination.

Essentially, we need to explore all possible combinations of the numbers that add up to the `target` and return a list of all combinations that meet the criteria without any repetitions.

Intuition

The intuition behind the solution involves a backtracking technique, which is similar to depth-first search (DFS).

Since we need to find all combinations that add up to the target and any number in the candidates can be used only once, we sort the candidates first. Sorting the array helps us to easily skip over duplicate elements and avoid generating duplicate combinations.

Then, we perform DFS with backtracking, starting from the beginning of the `candidates` array and exploring each possibility by either including or excluding a candidate. After including a candidate in the combination, we call the function recursively, reducing the `target` by the value of that candidate and moving to the next index.

While exploring, we keep track of the current sum of the combination, and when the sum equals the target, we add the current combination to the answer. If at any point, the sum exceeds `target` or we reach the end of the candidates array, we backtrack.

To avoid duplicate combinations, we skip subsequent candidates that are the same as the previous one at each stage in the loop. This way, we ensure that if a number has been used in a combination, the same number is not used immediately again to form another similar combination.

The result of the function will be a list of lists, where each inner list is a unique combination of numbers that sum to the target value.

Solution Approach

The solution uses depth-first search (DFS), backtracking, and sorting to effectively find all unique combinations. Let's break down how each part of the code contributes to the solution:

1. **Sorting the Candidates**: The `candidates` array is first sorted to ensure that we can easily skip duplicates.

```
1. candidates.sort()
```

2. **Depth-First Search (DFS) Function**: The `dfs` function is defined to handle the recursion, taking two parameters: `i` (the current index in the `candidates` list) and `s` (the remaining sum needed to reach the `target`).

```
1. def dfs(i: int, s: int):
```

3. **Condition to Add to the Answer**: Inside the `dfs` function, we check if the remaining sum `s` is zero, meaning we found a valid combination that sums to the target. In that case, we add a copy of the current combination to the answer list.

```
1. if s == 0:
2.     ans.append(t[:])
```

4. **Base Cases for Termination**: We return if the index `i` moves beyond the length of the `candidates` or if the remaining sum `s` is less than the current candidate, which means we can't reach the desired total with the current and subsequent candidates since they are all larger.

```
1. if i >= len(candidates) or s < candidates[i]:
2.     return
```

5. **Loop Over the Candidates**: Starting from the current index to the end of `candidates`, we try to include the candidate in the combination.

```
1. for j in range(i, len(candidates)):
```

6. **Skipping Duplicates**: Before including a candidate in the combination, we skip over it if it's the same as its predecessor to avoid duplicates in our answer list (since we've already considered this value in the previous steps).

```
1. if j > i and candidates[j] == candidates[j - 1]:
2.     continue
```

7. **Backtracking**: After including a candidate, the `dfs` function is called recursively with the updated index (`j+1`) and the updated remaining sum (`s - candidates[i]`). After this recursive call, we backtrack by removing the last candidate from the combination and moving on to the next candidate.

```
1. t.append(candidates[i])
2. dfs(j + 1, s - candidates[i])
3. t.pop()
```

The solution utilizes a list `ans` to store all the unique combinations and a temporary list `t` to store the current combination being constructed.

After defining `dfs`, the solution begins the search with:

```
1. ans = []
2. t = []
3. dfs(0, target)
```

The DFS and backtracking continue until all possible combinations that meet the criteria have been explored, after which `ans` is returned, containing all the valid combinations.

This approach efficiently explores all possible combinations and prunes the search space to avoid duplicates and unnecessary searches, thereby finding the solution set in an optimal manner.

Example Walkthrough

Let's assume our `candidates` array is `[2, 5, 2, 1, 2]`, and our `target` is `5`.

Following the solution approach:

1. First, we **sort** the `candidates` to `[1, 2, 2, 2, 5]`.
2. The `dfs` function starts with `i` as 0 (the first index) and the `target` sum `s` as 5.
3. We enter a loop that will iterate through the candidates starting from index `i`.
4. On the first iteration, `j` is `i`, and the candidate is `1`.
   - We **include** `1` in our temporary combination `t` and call `dfs(1, 4)`, which moves to the next index and decrements the target by the included candidate's value (5 - 1 = 4).
5. The function `dfs(1, 4)` starts and enters a loop starting at index `1`; the first candidate it considers is `2`.
   - We **include** `2` to make our combination `[1, 2]` and call `dfs(2, 2)`.
6. The function `dfs(2, 2)` again finds the candidate `2` at index `2`.
   - However, this time `j` is not greater than `i`, which means it's the first occurrence of this candidate in the loop, so we **include** `2` to get `[1, 2, 2]` and call `dfs(3, 0)`.
7. Since the remaining sum `s` is 0 (`dfs(3, 0)`), we've found a valid combination which sums to the target. So we add `[1, 2, 2]` to our answer list `ans`.
8. Now, we backtrack and the temporary combination `t` becomes `[1, 2]` again.
   - The loop in `dfs(2, 2)` continues with the next `j`, as `j` but the candidate `2` at index `j` is a duplicate. Thus, we **skip** it and continue to `j` as 4. Since `candidates[4]` is 5, which is greater than our remaining sum `2`, this does not lead to a valid combination. So, we finish `dfs(2, 2)` without further action.
9. We continue backtracking to `dfs(1, 4)`; it is now `[1]`.
   - The loop resumes with the next `j`, which is `2`. But since `candidates[2]` is a duplicate of the previous candidate, we **skip** it and do the same for `j` as `3`.
   - For `j` as `4`, the candidate is `5`, but adding `5` to `[1]` will exceed the remaining sum (`4`). So no further action is taken.
10. Finally, we backtrack to the very first `dfs(0, 5)`, `t` is now `[]`, and we continue to the next candidate, which is `2` at `j` as `1`.
    - However, this `2` is not considered as it is a duplicate of the previous candidate.
11. The next non-duplicate candidate is the `5` at `j` as `4`.
    - **We include** this `5` and call `dfs(5, 0)`.
    - Since the remaining sum `s` is 0, we've found another valid combination that sums to the target, so `[5]` is added to our answer list `ans`.

Finally, our `dfs` is done exploring all possibilities, and we have `ans` as `[[1, 2, 2], [5]]`, which are the unique combinations that sum to the target `5`.

Python Solution

```
from typing import List

class Solution:
    def combinationSum2(self, candidates: List[int], target: int) -> List[List[int]]:
        # Helper function to perform depth-first search
        def dfs(start_index: int, current_sum: int):
            # If the current sum is 0, we found a valid combination
            if current_sum == 0:
                combination_ans.append(combination[:])
                return
            # If the index is out of bounds or the current element is larger than the current_sum, stop recursion
            if start_index >= len(candidates) or current_sum < candidates[start_index]:
                return

            # Iterate over the candidates starting from the current index
            for i in range(start_index, len(candidates)):
                # Skip duplicates to avoid repeating the same combination
                if i > start_index and candidates[i] == candidates[i - 1]:
                    continue
                # Add the current candidate to the current combination
                combination.append(candidates[i])
                # Recursively call dfs with the next index and the remaining sum
                dfs(i + 1, current_sum - candidates[i])
                # Backtrack by removing the last added candidate
                combination.pop()

        # Sort the candidates to handle duplicates easily
        candidates.sort()
        # This list will hold all unique combinations
        combination_ans = []
        # This temp list will store the current combination
        combination = []
        # Start the depth-first search
        dfs(0, target)
        # Return all unique combinations that add up to the target
        return combination_ans

# Example usage:
# sol = Solution()
# print(sol.combinationSum2([10,1,2,7,6,1,5], 8)
# print(result)  # Output would be a list of lists with all the unique combinations
```

Java Solution

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

public class Solution {
    private List<List<Integer>> combinations = new ArrayList<>(); // List to hold the final combinations.
    private List<Integer> tempCombination = new ArrayList<>(); // Temporary list to hold a single combination.
    private int[] sortedCandidates; // Array to hold the input candidates after sorting.

    // Method to find all possible unique combinations that add up to the target.
    public List<List<Integer>> combinationSum2(int[] candidates, int target) {
        Arrays.sort(candidates); // Sort the array to handle duplicates and to make it easier to prune branches.
        sortedCandidates = candidates; // Storing the sorted array in an instance variable for easy access.
        backtrack(0, target); // Begin the backtrack algorithm.
        return combinations; // Return the final list of combinations.
    }

    // Backtrack method to find combinations.
    private void backtrack(int startIndex, int remainingSum) {
        // If the remainingSum is 0, the current combination is a valid combination.
        if (remainingSum == 0) {
            combinations.add(new ArrayList<>(tempCombination));
            return;
        }

        // If the startIndex is out of bounds or the smallest candidate is greater than the remainingSum, there are no valid combinat
        if (startIndex >= sortedCandidates.length || remainingSum < sortedCandidates[startIndex]) {
            return;
        }

        // Iterate through the candidates starting from startIndex.
        for (int i = startIndex; i < sortedCandidates.length; ++i) {
            // Skip duplicates to avoid redundant combinations.
            if (i > startIndex && sortedCandidates[i] == sortedCandidates[i - 1]) {
                continue;
            }
            // Add the candidate to the current combination.
            tempCombination.add(sortedCandidates[i]);
            // Recursively call the method with the next index and the updated remaining sum.
            backtrack(i + 1, remainingSum - sortedCandidates[i]);
            // Backtrack: remove the last added candidate to try other candidates.
            tempCombination.remove(tempCombination.size() - 1);
        }
    }
}
```

C++ Solution

```
class Solution {
public:
    vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {
        // Sort the candidates numbers to handle duplicates and ease the combination process.
        sort(candidates.begin(), candidates.end());

        // Initialize the answer vector that will hold all unique combinations.
        vector<vector<int>> answer;

        // Temporary vector to store individual combinations.
        vector<int> temp;

        // Define a recursive lambda function to perform depth-first search (DFS) for combinations.
        // 'currentIndex' is the start index for exploring candidates.
        // and 'currentSum' is the remaining sum we need to hit the target.
        function<void(int, int)> dfs = [&](int currentIndex, int currentSum) {
            // Base case: if the remaining sum is zero, we found a valid combination.
            if (currentSum == 0) {
                answer.emplace_back(temp);
                return;
            }

            // If out of bounds or the current number exceeds the remaining sum, backtrack.
            if (currentIndex >= candidates.size() || currentSum < candidates[currentIndex]) {
                return;
            }

            // Iterate through the candidates starting from 'currentIndex'.
            for (int i = currentIndex; i < candidates.size(); ++i) {
                // Skip duplicates to guarantee uniqueness of combinations.
                if (i > currentIndex && candidates[i] == candidates[i - 1]) {
                    continue;
                }
                // Choose candidate 'i' and go deeper into the recursion tree.
                temp.emplace_back(candidates[i]);
                // Recurse with the next index and the new remaining sum.
                dfs(i + 1, currentSum - candidates[i]);
                // Backtrack and remove the previously chosen candidate.
                temp.pop_back();
            }
        };

        // Start the DFS process with index 0 and the target sum.
        dfs(0, target);

        // Return all unique combinations that add up to the target.
        return answer;
    }
};
```

Typescript Solution

```
// This function finds all unique combinations of candidates where the candidate numbers sum to target.
// Each candidate may only be used once in the combination.
// It returns a list of all unique combinations.
function combinationSum2(candidates: number[], target: number): number[][] {
    // Sorts the input candidates to simplify the process of skipping duplicates
    candidates.sort((a, b) => a - b);
    // This will hold all the unique combinations that meet the target sum
    const combinations: number[][] = [];
    // Temporary array to store the current potential combination
    const tempCombination: number[] = [];
    // Depth-first search function to explore all possibilities
    const dfs = (startIndex: number, remainingSum: number) => {
        // If the remaining sum is 0, we found a valid combination
        if (remainingSum === 0) {
            // Add a copy of the current combination to the final results
            combinations.push(tempCombination.slice());
            return;
        }

        // If the startIndex is out of bounds or the smallest candidate exceeds the remainingSum
        // there's no need to continue searching in this branch
        if (startIndex >= candidates.length || remainingSum < candidates[startIndex]) {
            return;
        }

        // Iterate over the candidates starting from the startIndex
        for (let i = startIndex; i < candidates.length; ++i) {
            // Skip duplicates to ensure unique combinations
            if (i > startIndex && candidates[i] === candidates[i - 1]) {
                continue;
            }
            // Include the current candidate in the temporary combination
            tempCombination.push(candidates[i]);
            // Continue exploring further with the next candidates and a reduced remaining sum
            dfs(i + 1, remainingSum - candidates[i]);
            // Backtrack and remove the last candidate from the combination
            tempCombination.pop();
        }
    };

    // Initialize the depth-first search with starting index 0 and the initial target sum
    dfs(0, target);
    // Return all unique combinations found that sum up to the target
    return combinations;
}
```

Time and Space Complexity

The provided Python code solves the combination sum problem where each number in the array `candidates` can only be used once to find all unique combinations that sum up to a given target.

Time Complexity

The time complexity of this code primarily depends on the depth of the recursion and the number of recursive calls made at each level.

1. The recursion depth is at most the target value if we choose candidates with a value of 1 every time. However, since the same candidate cannot be used repeatedly, the recursion depth is constrained by the number of candidates in the worst case.
2. At every level of recursion, we iterate over the remaining candidates, so, in the worst case, the number of recursive calls can be exponential in nature, implied by $O(2^n)$, where n is the number of candidates. However, since we skip duplicates after sorting, the number of branches in the recursion tree can be less than that.

A more accurate bound is not straightforward because it depends on the candidates and the target value. The worst-case time complexity, without considering duplicates, is $O(2^n)$, where n is the number of candidates.

3. The sorting operation at the start of the code takes $O(n \log(n))$ time.

Combining the sorting with the recursion leads to a worst-case time complexity of $O(n \log(n) + 2^n)$. Considering that the exponential part is more dominant, we can approximate it as $O(2^n)$.

Space Complexity

The space complexity of the code consists of:

1. Space used by the recursion stack, which in the worst case is equivalent to the depth of recursion, at most $O(n)$ if all candidates are used.
2. Space for the temporary list `t`, which stores the current combination, will at most contain n values, adding another $O(n)$.
3. Finally, the output list `ans` that could potentially hold all unique combinations of candidates. In the worst case, this could be all possible combinations which can be exponential, represented as $O(2^n)$.

Hence, the overall space complexity, considering the output space and the recursion stack depth, is $O(n + 2^n)$. Typically, the output space can be a separate consideration, and if we exclude it, the space complexity for computation is $O(n)$. However, if we include the space needed for the output, it would be $O(2^n)$ due to the possibility of storing all combinations.