216. Combination Sum III

Medium Array Backtracking

Problem Description

The problem requires us to find all unique combinations of `k` different numbers, which sum up to a target number `n`, using only numbers from `1` to `9`. Each number can be used at most once in any combination. The resulting combinations are to be returned in a list, where the order of combinations does not matter, and no combination is repeated.

For example, if `k=3` and `n=7`, we have to find all unique combinations of three distinct numbers between `1` and `9` that add up to `7`. One such valid combination could be `[1, 2, 4]` because `1+2+4=7`.

Intuition

To find the solution, we can utilize a **Depth First Search (DFS)** algorithm. DFS will help us explore all possible combinations of numbers recursively while adhering to the constraints. Here's the intuition behind using DFS for this problem:

- We begin by considering numbers from `1` to `9` and use each of them as a starting point of a combination.
- To build a combination, we add a number to the current combination (`t` in the given code) and recursively call the DFS function to add the next number.
  - While adding the new number, we have three conditions to check:
    1. We must not exceed the target sum `n`.
    2. We should not use more than `k` numbers.
    3. We cannot use numbers greater than `9`.
- If the sum of numbers in the current combination equals `n` and we have used exactly `k` numbers, then we found a valid combination which we add to the answer list (`ans`).
- After exploring a number's inclusion, we backtrack by removing the number from the current combination and exploring the possibility of not including that number.
- Through this process of including and excluding each number, and backtracking after exploring each possibility, we ensure that all valid combinations are found.
- Each combination is built up incrementally from the smaller numbers towards the larger ones to avoid repeated combinations and maintain uniqueness.

The beauty of DFS in this situation is that it inherently avoids duplicates and handles the "each number used at most once" constraint by the recursive nature of its implementation. DFS explores each branch fully (one specific number added vs. not added) before backtracking, which helps cover all potential unique combinations without repetition.

Solution Approach

The provided solution uses a **Depth First Search (DFS)** algorithm to explore all possible unique combinations of numbers that add up to `n` using at most `k` numbers. Here is a step-by-step breakdown of the approach, referring to specific parts of the implementation:

1. **DFS Function**: The function `dfs(i: int, s: int)` is a recursive function where `i` represents the current number that we are considering adding to the combination, and `s` is the remaining sum that we need to achieve. The solution initializes the function by calling `dfs(1, n)`.
2. **Base Conditions**:
   - The first base condition checks whether the remaining sum `s` is `0`. If `s` is `0` and we have exactly `k` numbers in our temporary combination `t`, then we have found a valid combination. We then append a a copy of `t` (using `t[:]` to create a copy) to our answers list `ans`.
   - The second base condition checks whether the current number `i` is greater than `9` or `i` is greater than the remaining sum `s` or we have already selected `k` numbers. In any of these cases, the function returns without making further recursive calls since no valid combination can be completed under these conditions.
3. **Choosing the Current Number**:
   - We add the current number `i` to our temporary combination `t`. This is the "exploring the possibility of including the number" part of the DFS.
4. **Recursive Call for Next Number**:
   - After including the current number, we make a recursive call to `dfs(i + 1, s - i)`. This call will explore combinations where the current number `i` is part of the solution. By passing `i + 1`, we are ensuring that the same number is not used more than once. By passing `s - i`, we are reducing the sum by the value of the number we've included.
5. Backtracking:
   - After the recursive call, we backtrack by removing the last number that we added — `t.pop()`. This is where we explore the possibility of excluding the current number.
6. **Recursive Call without the Current Number**:
   - Another recursive call `dfs(i + 1, s)` is then made. This call will explore combinations that exclude the current number `i`.
7. **Initialization**:
   - The function maintains a list `ans` to collect all valid combinations found by the DFS.
   - A list `t` is used to build a temporary combination that is being explored.
8. **Return Result**:
   - Once the initial call to `dfs(1, n)` has completed, all possible combinations have been explored, and the `ans` list contains all valid combinations as required by the problem. This list is returned as the final result.

Through the combination of recursive DFS, building combinations incrementally, making sure that each number is used at most once, and backtracking after exploring each possibility, the solution efficiently finds all the valid combinations that sum up to `n`. The use of a temporary list `t` for tracking the current combination and the answer list `ans` are examples of data structures used in this problem to store intermediate and final results, respectively.

Example Walkthrough

Let's consider a small example to illustrate the solution approach with `k=2` and `n=5`. We need to find all unique pairs of numbers from `1` to `9` that sum up to `5`.

1. **Initialization**: The `ans` list is initialized, which will store all valid combinations. The temporary combination list `t` is also initialized and the function `dfs(1, 5)` is called to begin the search for combinations that sum up to `5`.
2. **First Call to DFS**: `dfs(1, 5)` represents the state where we are considering whether to include the number `1` in our combination and the remaining sum to reach our target `n` is `5`. At this state, two branches of recursive calls will occur: one where we include the number `1` and another where we don't.
3. **Including the Number '1'**: We include `1` to our temporary list `t` which is now `[1]`, and the remaining sum `s` becomes `4`. We call `dfs(2, 4)` to explore next numbers.
4. **Recursive Call `dfs(2, 4)`**: Now we are evaluating whether to include the number `2`. We choose to include `2`, so `t` updates to `[1, 2]` and call `dfs(3, 2)` because now the remaining sum `s` is `2`.
5. **Base Condition Met**: The recursive call `dfs(3, 2)` will recursively lead to a point where the remaining sum `s` is `0`, and we have exactly `2` numbers in `t`. When `t = [1, 4]`, `s` becomes `0`. We add `[1, 4]` to `ans`, which is one valid combination.
6. **Backtracking**: After adding `[1, 4]` to `ans`, we backtrack by removing `4` from `t`. Now we have `t = [1]` again, and we call `dfs(3, 5)` since we still need to explore combinations starting with `1` without including `4`.
7. **Not Including the Number '1'**: We also need to consider combinations that do not include `1`. After the first branch is fully explored with `1` in `t`, we remove `1` to explore the other branch. We call `dfs(2, 5)` to move on to the next number without including `1`.
8. **Exploring Further Combinations**: The process of including and excluding numbers continues, now starting from `2` and exploring combinations `[2, 3]`, `[3, 2]`, ..., etc., until all valid pairs are found. This results in finding another valid pair `[2, 3]`.
9. **Return Result**: Once all numbers from `1` to `9` have been considered through recursive calls and backtracking, the search is complete. The `ans` list which stored all valid combinations (`[[1, 4], [2, 3]]`) is returned.

At the end of execution, the returned value is `[[1, 4], [2, 3]]`, which includes all unique combinations of 2 different numbers that sum up to 5.

This example walkthrough demonstrates the step-by-step approach taken by the DFS algorithm to explore all possibilities of including or excluding each number, checking for base conditions, backtracking as necessary, and ultimately collecting all the valid combinations into the answer list.

Python Solution

```python
from typing import List

class Solution:
    def combinationSum3(self, k: int, n: int) -> List[List[int]]:
        # Helper function to perform depth-first search
        def dfs(start: int, remaining_sum: int):
            # If the remaining sum is 0 and we have 'k' numbers, we found a valid combination
            if remaining_sum == 0 and len(combination) == k:
                results.append(combination[:])  # Add a copy of the current combination
                return
            # If we have gone past 9, if the current number is greater than the remaining sum,
            # or if we already have 'k' numbers, back out of the recursion
            if start > 9 or start > remaining_sum or len(combination) >= k:
                return
            # Include the current number and continue the search
            combination.append(start)
            dfs(start + 1, remaining_sum - start)
            # Exclude the current number (backtrack) and continue the search
            combination.pop()
            dfs(start + 1, remaining_sum)

        # This list holds all combinations found
        results = []
        # The current combination being explored
        combination = []
        # Start the search with 1 as the smallest number and 'n' as the target sum
        dfs(1, n)
        return results

# Example usage:
# sol = Solution()
# print(sol.combinationSum3(3, 7))  # Output: [[1, 2, 4]]
# print(sol.combinationSum3(3, 9))  # Output: [[1, 2, 6], [1, 3, 5], [2, 3, 4]]

```

Java Solution

```java
class Solution {
    // List to store the final combinations
    private List<List<Integer>> combinations = new ArrayList<>();
    // Temporary list for the current combination
    private List<Integer> currentCombination = new ArrayList<>();
    // The number of numbers to use in each combination
    private int combinationLength;

    // The public method that initiates the combination search
    public List<List<Integer>> combinationSum3(int k, int n) {
        this.combinationLength = k;
        searchCombinations(1, n);
        return combinations;
    }

    // Helper method to perform depth-first search for combinations
    private void searchCombinations(int start, int remainingSum) {
        // If remaining sum is zero and the current combination's size is k
        if (remainingSum == 0) {
            if (currentCombination.size() == combinationLength) {
                // Found a valid combination, add a copy to the result list
                combinations.add(new ArrayList<>(currentCombination));
            }
            return; // Backtrack
        }
        // If the current number exceeds 9, the remaining sum, or if we have enough numbers in the current combination
        if (start > 9 || start > remainingSum || currentCombination.size() >= combinationLength) {
            return; // Cannot find a valid combination from here, backtrack
        }
        // Include 'start' in the current combination
        currentCombination.add(start);
        // Continue to the next number with the updated remaining sum
        searchCombinations(start + 1, remainingSum - start);
        // Exclude 'start' from the current combination (backtrack)
        currentCombination.remove(currentCombination.size() - 1);
        // Continue to the next number without including 'start'
        searchCombinations(start + 1, remainingSum);
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    vector<vector<int>> combinationSum3(int k, int n) {
        // ans stores all the unique combinations
        vector<vector<int>> ans;

        // temp stores the current combination
        vector<int> temp;

        // Define DFS function to find combinations
        function<void(int, int)> dfs = [&](int start, int sum) {
            // If the remaining sum is zero and the current combination has k numbers, add to answer
            if (sum == 0 && temp.size() == k) {
                ans.emplace_back(temp);
                return;
            }
            // Prune the search space if the current number exceeds the needed sum, or the size of the
            // combination exceeds or if the current number is greater than 9
            if (start > 9 || start > sum || temp.size() >= k) {
                return;
            }
            // Include the current number and move to the next number
            temp.emplace_back(start);
            dfs(start + 1, sum - start);

            // Exclude the current number and move to the next number
            temp.pop_back();
            dfs(start + 1, sum);
        };

        // Begin DFS with number 1 and target sum n
        dfs(1, n);
        return ans;
    }
};

```

Typescript Solution

```typescript
function combinationSum3(setSize: number, targetSum: number): number[][] {
    // Initialize a list to hold all the unique combinations
    const combinations: number[][] = [];
    // Temporary list to hold the current combination
    const tempCombination: number[] = [];

    // Define a depth-first search function to explore possible combinations
    const depthFirstSearch = (start: number, remainingSum: number) => {
        // If remaining sum is 0 and combination size equals the target set size, we found a valid combination
        if (remainingSum === 0 && tempCombination.length === setSize) {
            // Add a copy of the valid combination to the list of combinations
            combinations.push([...tempCombination]);
            return;
        }

        // Termination condition: if the start number is greater than 9,
        // greater than the remaining sum needed, or the combination size exceeds the set size
        if (start > 9 || start > remainingSum || tempCombination.length > setSize) {
            return;
        }

        // Include the current number in the combination and move to the next number
        tempCombination.push(start);
        depthFirstSearch(start + 1, remainingSum - start);

        // Exclude the current number from the combination and move to the next number
        tempCombination.pop();
        depthFirstSearch(start + 1, remainingSum);
    };

    // Start the depth-first search with 1 as the starting number and the target sum
    depthFirstSearch(1, targetSum);

    // Return all found combinations
    return combinations;
}

```

Time and Space Complexity

The given Python code defines a method `combinationSum3` that finds all possible combinations of `k` different numbers that add up to a number `n`, where each number is from 1 to 9 (inclusive) and each combination is unique.

Time Complexity

The time complexity of this function can be analyzed by looking at the recursive `dfs` calls. Each call to `dfs` potentially makes two further calls, corresponding to including the current number `i` or excluding it. We can view this as a binary tree of decisions, where at each step, we decide whether to include a number in our current combination (`t.append(i)`) or not (`dfs(i + 1, s)`).

Each number between 1 and 9 is considered exactly once in the context of a particular combination, and there are `9` choices at the first call, `8` at the second, and so forth. However, the depth of our recursion is limited by `k` (the size of each combination) and the fact that we do not consider subsequent numbers once we reach a sum greater than `n` or a combination length equal to `k`.

Therefore, the time complexity is **`O(9! / (9 - k)!)`** in the worst case (when `n` is large enough to allow all combinations of `k` numbers). Note that since `k <= 9`, this time complexity remains within a polynomial bound with respect to `9` (the number of different numbers we can pick).

Space Complexity

The space complexity is determined by the space needed for the recursive call stack and the space used to store the combinations. In the worst case, the recursion can go as deep as `k`, as we stop further recursion when the length of the temporary list `t` reaches `k`. Therefore the recursion call stack will contribute **`O(k)`**.

The space for storing all the combinations also needs to be considered. We have a list of lists to store the valid combinations, and, at most, each combination contains `k` elements. In the worst case, the number of combinations stored will also be bounded by the total number of combinations of `k` numbers out of `9`, which is **`O(9! / (k!(9 - k)!))`** (this is the binomial coefficient representing the number of ways to choose `k` distinct integers from a set of `9`).

However, because these combinations are part of the output, we often do not count this as extra space in terms of space complexity analysis (since the space is required to represent the output). Thus the space complexity is usually considered to be **`O(k)`**, which accounts only for the recursion depth and not the output storage.

Combining both the recursion call stack and the output storage, and if we were to include the output as part of the space complexity, our total space complexity would be **`O(9! / (k!(9 - k)!) + k)`**.

The complexities provided assume that `k` and `n` are fixed and do not grow with the input size since they are parameters and not part of the input. Therefore, we should clarify that this complexity analysis is specific to this problem and not generally applicable to variations where `k` or `n` might not be bounded by small constants.