77. Combinations

Medium Backtracking

Leetcode Link

Problem Description

The problem requires us to generate all combinations of `k` numbers from a set of integers ranging from `1` to `n`. A combination is a selection of items from a collection, such that the order of selection does not matter. In mathematical terms, you are asked to find all possible subsets of size `k` from the given range.

For instance, if `n=4` and `k=2`, the function should return combinations like `[1,2]`, `[1,3]`, `[1,4]`, `[2,3]`, `[2,4]` and `[3,4]`. Importantly, combinations do not account for order; thus, `[1,2]` is considered the same as `[2,1]` and should not be listed as a separate combination.

The answer can be returned in any sequence, meaning the combinations do not need to be sorted in any particular order.

Intuition

To solve this problem, we use a method called depth-first search (DFS), which is commonly used to traverse or search tree or graph data structures. The concept can also be applied to recursively generate combinations.

Here's how we can think through the DFS approach:

- Imagine each number `i` in the range `[1, n]` is a potential candidate to be included in a combination. For each candidate, we have two choices: either include it in the current combination `t` or skip it and move on to the next number without adding it.
- Starting from the first number, we dive deeper by making a recursive call with that number included in our current combination. This is the "depth-first" part where we are trying to build a complete combination by adding one number at a time.
- Once we hit a base case where the combination has a length of `k`, we've found a valid combination and add a copy of it to our answer list `ans`. We must remember to add a copy (`t[:]`) because `t` will continue to be modified.
- After exploring the depth with the current number included, we backtrack by removing the last included number (`t.pop()`) and proceed to the next recursive call without the current number, effectively exploring the possibility where the current number is not part of the combination.
- The process continues until all combinations of size `k` have been added to the answer list, involving exploring each number's inclusion and exclusion systematically.

By using the DFS pattern, we ensure all possible combinations are accounted for, by systematically including and excluding each number, and exploring combinations further only until they've reached the desired size `k`.

This is an elegant solution to the problem because it inherently handles the constraint that combinations must be of size `k`, and it requires no special handling to avoid duplicate combinations (since we only ever move forward to higher numbers, never backwards).

Solution Approach

The solution uses DFS, a common algorithm for traversing or searching through a tree or graph structure, to explore all combinations of `k` numbers from range `[1, n]`.

Here's the step-by-step breakdown of how the code implements this approach:

1. Define a helper function `dfs` with a parameter `i`, which represents the current number being considered to be part of a combination.
2. Use a list `t` to store the current state of the combination being built.
3. If the length of `t` matches `k`, a complete combination has been formed. Make a copy of `t` using slicing (`t[:]`) and append it to our answer list `ans`.
4. If `i` goes beyond `n`, it means we've considered all possible numbers and should terminate this path of the DFS.
5. For the current number `i`, add it to the combination list `t` and call `dfs(i + 1)` to consider the next number, diving deeper into the DFS path.
6. After exploring the path with number `i` included, backtrack by removing `i` from `t` using `t.pop()` to undo the previous action.
7. Call `dfs(i + 1)` again to explore the path where `i` is not included in the combination.
8. Initialize the answer list `ans` and the temporary combination list `t` outside the helper function.
9. Start the DFS by calling `dfs(1)`, which will traverse all paths and fill `ans` with all possible combinations.
10. Once the DFS is complete, return the `ans` list, which contains the desired combinations.

The choice of data structures is straightforward. A list is used to keep track of the current combination (`t`) and the final list of combinations (`ans`). The use of list operations like `append()` to add to a combination and `pop()` to remove from it, enable efficient manipulation of the temporary combination while it's being built.

The DFS algorithm elegantly handles both the generation of combinations and the adherence to the constraint of each combination having size `k`, without the need for explicit checks to avoid duplicates or additional data structures.

The code leverages recursion to simplify the logic. It systematically includes each eligible number into a combination, recurses to consider additional numbers, then backtracks to exclude the number and recurse again. This ensures we explore all combinations of numbers from `[1, n]` of size `k`.

Example Walkthrough

Let's take `n = 3` and `k = 2` as a simple example to walk through the solution approach described above. Our aim is to generate all combinations where the size of each combination (`k`) is 2, using integers from 1 to 3.

Here's how we would execute the described algorithm step-by-step:

1. We start by initializing the answer list `ans` to hold our final combinations, and a temporary list `t` to hold the current combination we are building.
2. We begin our DFS traversal by invoking `dfs(1)`. This signals that we will start looking for combinations beginning with the number 1.
3. The first call to `dfs` looks at number 1 (`i = 1`). Since our `t` list is empty, and we haven't reached a combination of length `k` yet, we add 1 to `t`.
4. Next, we make a recursive call to `dfs(2)`, indicating that we should now consider number 2 for inclusion in the combination. Our `t` list currently is `[1]`.
5. In the `dfs(2)` call, we see that `t` still hasn't reached the length `k`, so we add 2 to `t`, which now becomes `[1, 2]`.
6. Upon the next invocation `dfs(3)`, the length of `t` is indeed `k`, so we add a copy of `t` to `ans`. `ans` now contains `[[1, 2]]`.
7. We backtrack by popping number 2 from `t`, then proceed by calling `dfs(3)` to consider combinations starting with `[1]` again but excluding the number 2 this time.
8. Since number 3 is within our range, we add it to `t` which becomes `[1, 3]`. That's another valid combination of length `k`, so we add a copy to `ans`. Now `ans` becomes `[[1, 2], [1, 3]]`.
9. We backtrack again, removing the number 3 from `t` and since there are no more numbers to consider after 3, the function rolls back to the previous stack frame where `i` was 2.
10. Now, we are back to `dfs(2)`, but since we've exhausted possibilities for the number 2, we increment to `dfs(3)` where we try to include the number 3 in the combination. Since 2 was already considered, our `t` list is empty at this point, so we add 3 to `t` to make `[3]`.
11. Calling `dfs(4)` from here we find that there are no more numbers to combine with 3 without consideration of the number 2 (which was already part of a previous combination), and since for this step `i` has exceeded `n`, no further action is taken and we revert back to the moment before including 3.
12. At this point, we have explored and returned from all possible DFS paths for the number 1. Now we increment to considering `dfs(2)`. Following the same procedure, we find the combination `[2, 3]` which is added to `ans`.
13. After exhausting all possible recursive paths, our DFS traversal is complete. Our final `ans` list, which contains all possible combinations, is `[[1, 2], [1, 3], [2, 3]]`.
14. We return `ans`, which is the output of our function.

We have now successfully generated all combinations of `k` numbers from a set of integers ranging from `1` to `n`. The elegance of this approach lies in its methodical inclusion and exclusion of each integer, ensuring all unique combinations are found, while the recursive structure keeps the code simple and concise.

Python Solution

```python
class Solution:
    def combine(self, n: int, k: int) -> List[List[int]]:
        # Helper function for the depth-first search algorithm.
        def dfs(start: int):
            # If the current combination size equals k, save a copy to the answer list.
            if len(current_combination) == k:
                combinations.append(current_combination[:])
                return
            # If the current start exceeds n, stop exploring this path.
            if start > n:
                return
            # Include the current number in the combination and move to the next.
            current_combination.append(start)
            dfs(start + 1)
            # Exclude the current number from the combination and move to the next.
            current_combination.pop()
            dfs(start + 1)

        # Initialize the list to store all possible combinations.
        combinations = []
        # List to store the current combination of numbers.
        current_combination = []
        # Start DFS with the smallest possible number.
        dfs(1)
        # Return all the possible combinations found.
        return combinations

```

Java Solution

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

class Solution {
    private final List<List<Integer>> combinations = new ArrayList<>();
    private final List<Integer> currentCombination = new ArrayList<>();
    private int totalNumbers;
    private int combinationSize;

    // Generates all combinations of numbers of size k from a set of numbers from 1 to n.
    public List<List<Integer>> combine(int n, int k) {
        totalNumbers = n;
        combinationSize = k;
        backtrack(1); // Start the backtracking algorithm from the first element
        return combinations;
    }

    // Uses backtracking to find all combinations.
    private void backtrack(int startIndex) {
        // If the current combination's size is equal to the desired size,
        // add it to the list of combinations.
        if (currentCombination.size() == combinationSize) {
            combinations.add(new ArrayList<>(currentCombination));
            return;
        }

        // If the index has gone beyond the last number, end the current path.
        if (startIndex > totalNumbers) {
            return;
        }

        // Include the current number in the combination and move to the next number.
        currentCombination.add(startIndex);
        backtrack(startIndex + 1);

        // Exclude the current number from the combination and move to the next number.
        currentCombination.remove(currentCombination.size() - 1);
        backtrack(startIndex + 1);
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <functional> // For std::function

class Solution {
public:
    // Returns all possible combinations of k numbers out of 1...n
    vector<vector<int>> combine(int n, int k) {
        vector<vector<int>> combinations; // Stores all the combinations
        vector<int> current; // Stores the current combination

        // Depth-first search (DFS) to find all combinations
        std::function<void(int)> dfs = [&](int start) {
            // If the current combination is of size k, add it to the answers
            if (current.size() == k) {
                combinations.emplace_back(current);
                return;
            }

            // If we've passed the last number, stop the recursion
            if (start > n) {
                return;
            }

            // Include the current number and go deeper
            current.emplace_back(start);
            dfs(start + 1); // Recurse with the next number

            // Exclude the current number and go deeper
            current.pop_back();
            dfs(start + 1); // Recurse with the next number
        };

        // Start the recursion with the first number
        dfs(1);

        // Return all the found combinations
        return combinations;
    }
};

```

Typescript Solution

```typescript
// Function to generate all possible combinations of k numbers out of the range [1, n].
function combine(n: number, k: number): number[][] {
    // Initialize the array to hold the resulting combinations.
    const combinations: number[][] = [];
    // Temporary array to hold the current combination.
    const currentCombination: number[] = [];
    // Depth-first search function to explore all possible combinations.
    const depthFirstSearch = (currentIndex: number) => {
        // If the current combination's length is k, a complete combination has been found.
        if (currentCombination.length === k) {
            // Add a copy of the current combination to the results.
            combinations.push(currentCombination.slice());
            return;
        }
        // If the currentIndex exceeds n, we've explored all numbers, so return.
        if (currentIndex > n) {
            return;
        }
        // Include the current index in the current combination and move to the next number.
        currentCombination.push(currentIndex);
        depthFirstSearch(currentIndex + 1);
        // Exclude the current index from the current combination and move to the next number.
        currentCombination.pop();
        depthFirstSearch(currentIndex + 1);
    };
    // Start the depth-first search from number 1.
    depthFirstSearch(1);
    // Return all the generated combinations.
    return combinations;
}

```

Time and Space Complexity

The provided Python code performs combinations of `n` numbers taken `k` at a time. It uses a depth-first search (DFS) recursive strategy to generate all possible combinations.

Time Complexity

The time complexity of this algorithm can be determined by considering the number of recursive calls. At each point, the function has the choice to include a number in the combination or to move past it without including it. This results in the algorithm having a binary choice for each of the `n` numbers, which hints at a `O(2^n)` time complexity.

However, due to the nature of combinations, the recursive calls early terminate when the length of the temporary list `t` equals `k`. Therefore, the time complexity is better approximated by the number of `k`-combinations of `n`, which is `O(n choose k)`. Using binomial coefficient, the time complexity can be expressed as `O(n! / (k! * (n - k)!))`.

Space Complexity

The space complexity includes the space for the output list and the space used by the call stack due to recursion.

1. **Output List**: The output list will hold `C(n, k)` combinations, and each combination is a list of `k` elements. Therefore, the space needed for the output list is `O(n choose k * k)`.
2. **Recursion Stack**: The maximum depth of the recursion is `n` because in the worst case, the algorithm would go as deep as trying to decide whether to include the last number. Therefore, the space used by the call stack is `O(n)`.

When considering space complexity, it is important to recognize that the complexity depends on the maximum space usage at any time. So, the space complexity of the DFS recursive solution is dominated by the space needed for the output list. Hence, the space complexity is `O(n choose k * k)`.

In summary:

- The time complexity is `O(n! / (k! * (n - k)!))`.
- The space complexity is `O(n choose k * k)`.