Combination Sum Documentation

Problem Description:

Given an array of distinct integers `candidates` and a target integer `target`, this problem requires finding all unique combinations of candidates where the chosen numbers sum to the target. Each number in the array can be used an unlimited number of times. Two combinations are considered unique if the frequency of at least one of the chosen numbers is different.

Examples:

Example 1:

Input:

candidates = [2,3,6,7]

target = 7`

Output: [[2,2,3],[7]]

Explanation:

- 2 and 3 are candidates, and 2 + 2 + 3 = 7. Note that 2 can be used multiple times.
- 7 is a candidate, and 7 = 7.
- These are the only two combinations.

Example 2:

Input:

candidates = [2,3,5]

target = 8

Output: [[2,2,2,2],[2,3,3],[3,5]]

Example 3:

Input:

candidates = [2]

target = 1

Output: `[]`

Constraints:

- `1 <= candidates.length <= 30`
- `2 <= candidates[i] <= 40`
- All elements of `candidates` are distinct.
- `1 <= target <= 40`

Solution Approach:

The problem can be solved using backtracking. Here's the basic approach:

1. Define a recursive function `backtrack` to explore all possible combinations.
2. Within the `backtrack` function:
    a. Base case:
    b. If the `target` becomes 0, append the current `path` to the `result`.
    c. Recursive case:
    d. Iterate through the candidates from the `start` index.
    e. Add the current candidate to the `path`.
    f. Recursively call `backtrack` with the updated `target` and `path`.
    g. Pop the last element from `path` to backtrack.
3. Initialize an empty `result` list.
4. Start the backtracking process with `backtrack(0, target, [], result)`.
5. Return the `result` list containing all unique combinations.

Code Explanation:

- The `combinationSum` method initializes the backtracking process and returns the result.
- Inside the `backtrack` function, the algorithm explores all possible combinations to find the target sum.
- Each valid combination is appended to the `result` list.
- Finally, the `result` list containing all unique combinations is returned.

Complexity Analysis:

Time Complexity: The time complexity of this solution is exponential, as it explores all possible combinations.

Space Complexity: The space complexity is also exponential, considering the recursive stack space required for backtracking.

Test Cases:

The provided solution has been tested against the provided test cases and yields the expected outputs.