6.7 COMBINATION SUM

Question:

Given an array of distinct integers candidates and a target integer target, return a list of all unique combinations of candidates where the chosen numbers sum to target. You may return the combinations in any order. The same number may be chosen from candidates an unlimited number of times. Two combinations are unique if the frequency of at least one of the chosen numbers is different. The test cases are generated such that the number of unique combinations that sum up to target is less than 150 combinations for the given input.

AIM

To implement a Python program that finds all unique combinations of numbers that sum to a target using backtracking.

ALGORITHM

1. Sort the candidate numbers (optional for optimization).
2. Use backtracking to explore combinations.
3. At each recursive step:
4. - Choose the current number if it does not exceed the remaining target.
5. - Recurse with the updated target (target - current number).
6. - Allow the same number to be chosen again by not moving to the next index.
7. Backtrack when the remaining target becomes zero (valid combination found).
8. Collect all valid combinations.

PROGRAM

```
def combination_sum(candidates, target):
    result = []
    def backtrack(start, path, total):
        if total == target:
            result.append(path[:])
            return
        if total > target:
            return
        for i in range(start, len(candidates)):
            path.append(candidates[i])
            backtrack(i, path, total + candidates[i])
            path.pop()
    backtrack(0, [], 0)
    return result

candidates = list(map(int, input("Enter candidates: ").split()))
target = int(input("Enter target: "))
print("Combinations:", combination_sum(candidates, target))
```

Input:

candidates = [2,3,6,7] , target = 7

Output:

```
Enter candidates: 2 3 6 7
Enter target: 7
Combinations: [[2, 2, 3], [7]]
>>>
```

RESULT:

Thus, the program is successfully executed and verified to generate all unique combinations that sum to the target.

PERFORMANCE ANALYSIS:

Time Complexity: O(2^t) in the worst case, where t is the target value, due to recursive exploration.

Space Complexity: O(t) for recursion depth and storing temporary paths.