6.8 COMBINATION SUM II

Question:

Given a collection of candidate numbers (candidates) and a target number (target), find all unique combinations in candidates where the candidate numbers sum to target. Each number in candidates may only be used once in the combination. The solution set must not contain duplicate combinations.

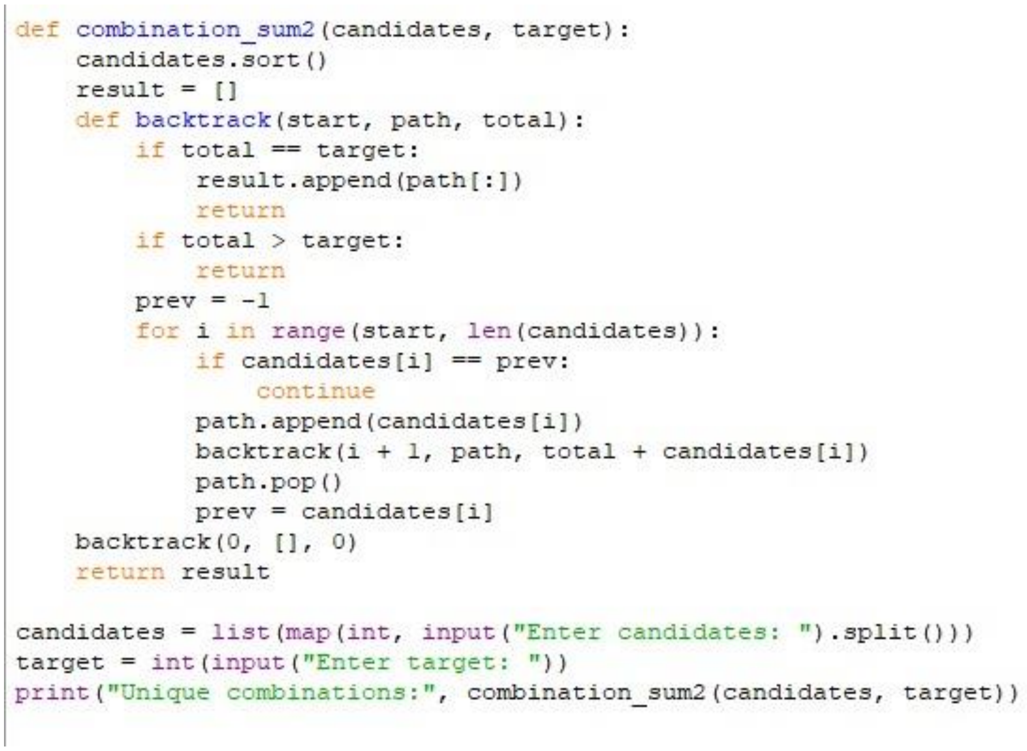
AIM

To implement a Python program that finds all unique combinations of numbers that sum to a target, ensuring each candidate number is used at most once and avoiding duplicate combinations.

ALGORITHM

1. Sort the candidates array to handle duplicates easily.
2. Use backtracking to explore possible combinations.
3. At each recursive step:
   - Skip duplicate numbers to avoid repeating the same combination.
   - Choose the current number if it does not exceed the remaining target.
   - Recurse with updated target and move to the next index (i+1) since each number can only be used once.
4. When the remaining target becomes zero, record the current combination as valid.
5. Return the list of unique valid combinations.

PROGRAM



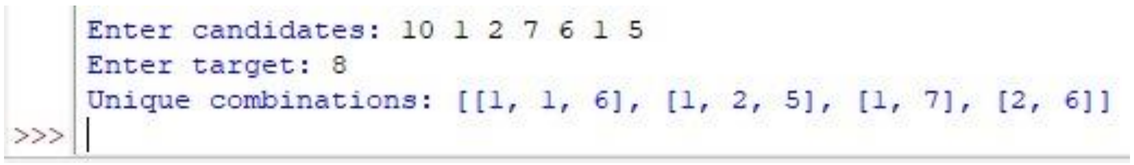
```
def combination_sum2(candidates, target):
    candidates.sort()
    result = []
    def backtrack(start, path, total):
        if total == target:
            result.append(path[:])
            return
        if total > target:
            return
        prev = -1
        for i in range(start, len(candidates)):
            if candidates[i] == prev:
                continue
            path.append(candidates[i])
            backtrack(i + 1, path, total + candidates[i])
            path.pop()
            prev = candidates[i]
    backtrack(0, [], 0)
    return result

candidates = list(map(int, input("Enter candidates: ").split()))
target = int(input("Enter target: "))
print("Unique combinations:", combination_sum2(candidates, target))
```

Input:

candidates = [10,1,2,7,6,1,5] , target = 8

Output:



```
Enter candidates: 10 1 2 7 6 1 5
Enter target: 8
Unique combinations: [[1, 1, 6], [1, 2, 5], [1, 7], [2, 6]]
>>> |
```

RESULT:

Thus, the program is successfully executed and verified to generate all unique combinations that sum to the target with each candidate used at most once.

PERFORMANCE ANALYSIS:

Time Complexity: Exponential in nature due to backtracking, but pruning reduces unnecessary exploration.

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