Assignment: Backtracking & Greedy Algorithms

1. **Implement a backtracking algorithm**

   Given a collection of amount values (A) and a target sum (S), find all unique combinations in A where the amount values sum up to S. Return these combinations in the form of a list. Each amount value may be used only the number of times it occurs in list A. The solution set should not contain duplicate combinations. Amounts will be positive numbers.
   Return an empty list if no possible solution exists.

   Example: A = [11,1,3,2,6,1,5]; Target Sum = 8
   Result = [[3, 5], [2, 6], [1, 2, 5], [1, 1, 6]]

a. Describe a backtracking algorithm to solve this problem

To solve this problem using backtracking, we first sort the input array and extract unique numbers along with their counts. This ensures each number is used only as many times as it appears. We then use a recursive function to explore combinations that sum up to the target. At each step, we add a number to the current combination, reduce its count, and recurse with the updated sum. If the sum reaches zero, we add the combination to the result. If it exceeds the target, we stop that path. After exploring, we backtrack by removing the number and restoring its count. This process ensures we find all unique combinations without duplicates.

Pseudocode:

```
Function amount(A, S):
    Sort A
    unique_nums = []
    counts = []

    # Create unique numbers and their counts
    For num in A:
        If unique_nums is empty or last element of unique_nums ≠ num:
            Append num to unique_nums
            Append 1 to counts
        Else:
            Increment last element of counts
```

```
    result = []
    Call backtrack(0, S, unique_nums, counts, [], result)
    Return result

Function backtrack(start, remainder, unique_nums, counts, combination, result):
    If remainder == 0:
        Append copy of combination to result
        Return
    If remainder < 0:
    Return

    For i from start to length(unique_nums):
        If counts[i] > 0:
            Append unique_nums[i] to combination
            Decrement counts[i]

            # Recursive call with updated remainder
            Call backtrack(i, remainder - unique_nums[i], unique_nums, counts, combination, result)

            # Undo choice (backtrack)
            Remove last element from combination
            Increment counts[i]
```

c. What is the time complexity of your implementation, you may find time complexity in detailed or state whether it is linear/polynomial/exponential. etc.?

$O(2^n)$; exponential – we are exploring all possible subsets

2. **Implement a Greedy algorithm**

You are a pet store owner and you own few dogs. Each dog has a specific hunger level given by array hunger_level [1..n] (i^{th} dog has hunger level of hunger_level [i]). You have couple of dog biscuits of size given by biscuit_size [1...m]. Your goal to satisfy maximum number of hungry dogs. You need to find the number of dogs we can satisfy.
If a dog has hunger hunger_level[i], it can be satisfied only by taking a biscuit of size biscuit_size [j] >= hunger_level [i] (i.e biscuit size should be greater than or equal to hunger level to satisfy a dog.)
If no dog can be satisfied return 0.
Conditions:
You cannot give same biscuit to two dogs.
Each dog can get only one biscuit.
Example 1:

Input: hunger_level[1,2,3], biscuit_size[1,1]
Output: 1
Explanation: Only one dog with hunger level of 1 can be satisfied with one cookie of size 1.

Example 2:

Input: hunger_level[2, 1], biscuit_size[1,3,2]
Output: 2
Explanation: Two dogs can be satisfied. The biscuit sizes are big enough to satisfy the hunger level of both the dogs.

a. Describe a greedy algorithm to solve this problem

To satisfy the most dogs, we use a greedy approach by giving each dog the smallest biscuit that meets its hunger level. First, we sort both lists in ascending order so we can start with the smallest values. Then, using a two-pointer technique, we check if a biscuit is big enough for the current dog. If it is, we assign it and move to the next dog. If not, we try the next biscuit. This continues until we run out of biscuits, or all dogs are fed. This method ensures we use biscuits efficiently and satisfy the maximum number of dogs.

Pseudocode:

```
Function feedDog(hunger_level, biscuit_size)

    Sort hunger_level in ascending order

    Sort biscuit_size in ascending order


    Set dog_index ← 0  # Pointer for hunger_level
```

```
Set biscuit_index ← 0  # Pointer for biscuit_size
Set satisfied_dogs ← 0  # Counter for satisfied dogs

While dog_index < length(hunger_level) and biscuit_index < length(biscuit_size) do
    If biscuit_size[biscuit_index] ≥ hunger_level[dog_index] then
        Increment satisfied_dogs
        Increment dog_index  # Move to the next dog
    Increment biscuit_index  # Move to the next biscuit

    RETURN satisfied_dogs
```

c. Analyze the time complexity of the approach.

The time complexity of this approach is determined by two main steps of sorting the arrays and iterating through them. First we sort hunger_level array, which takes $O(n\ logn)$, where n is the number of dogs. Similarly, sorting biscuit_size array takes $O(m\ logm)$, where m is the number of biscuits. After sorting, we use a two-pointer technique to iterate through both arrays, which runs in $O(n + m)$ time. The final time complexity of the algorithm is $O(n\ logn + m\ logm)$