Solution: 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.
   Each amount value may be used only once in the combination.
   The solution set should not contain duplicate combinations.
   Amounts will be positive numbers.

   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.
   b. Implement the solution in a function **amount(A, S)**. Name your file **Amount.py**

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

   This is similar to the combination of 'Permutation problem' and 'combination sum problem' covered in the exploration.

   a. Implement the solution in a function **amount(A, S)**. Name your file **Amount.py**

```python
def Amount(A, S):
    result = []
    A = sorted(A)

    for i in range(len(A)):
        num = [A[i]]
        if S - num[0] > 0:

            for i in Amount(A[i + 1:], S - num[0]):
                if num + i not in result:
                    result.append(num + i)

        elif S - num[0] == 0:
            result.append(num)
            return result

    return result


print(Amount([11,1,3,2,6,1,5], 8))
```

   b. What is the time complexity of your implementation?
   Backtracking method has exponential time complexity.

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] (ith 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.)

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[1,2], biscuit_size[1,2,3]

Output: 2

Explanation: Two dogs can be satisfied. The biscuit sizes are big enough to satisfy the hunger level of both the dogs.

c. Describe a greedy algorithm to solve this problem

Each dog can be given a closest higher biscuit. To do this we will first sort both hunger level and biscuit sizes in ascending order and then greedily pick one biscuit that is closest highest size to each hunger value.

d. Write an algorithm implementing the approach. Your function signature should be **feedDog(hunger_level[], biscuit_size[])**. Name your file **FeedDog.py**

```
def feedDog(hunger_level, biscuit_size):
    hunger_level = sorted(hunger_level)
    biscuit_size = sorted(biscuit_size)

    dogsCount = len(hunger_level)
    biscuitCount = len(biscuit_size)

    i = 0 # pointer for dogs
    j = 0 # pointer for biscuit

    while( i < dogsCount and j <biscuitCount):
        if(hunger_level[i] <= biscuit_size[j]):
            i += 1
            j += 1
        else:
            j += 1

    return i
```

e. Analyse the time complexity of the approach.

The time complexity is dominated by sort operations. Assume the count of dogs is m and count of biscuts is n. If we use merge sort to sort these: mlogm + nlogn will be the time required to sort. The while loop will run max(m,n) times.

O(mlogm + nlogn + max(m,n)) ➔ O(mlogm + nlogn)