Solution: Backtracking & Greedy Algorithms

1. **Implement a backtracking algorithm**

   You are given a numbers k and n. Find all unique combinations of numbers of length k that add to n. The condition on choosing a number of length k is that:

   You can use only numbers from 1 to 9

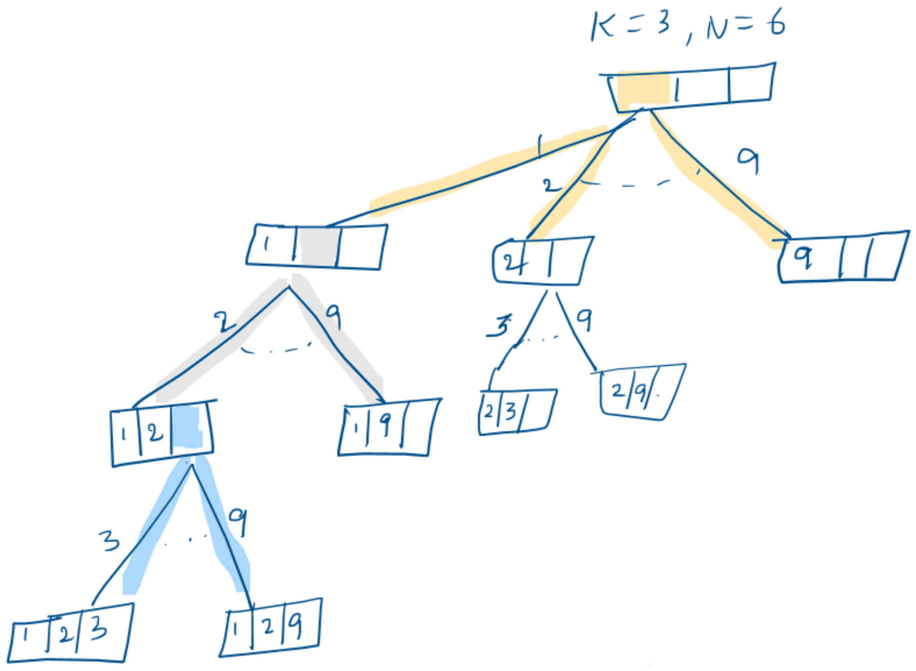
   You cannot repeat a digit a number

   Example 1:
   Input: k = 3, n = 6
   Output: [[1,2,3]]

   Example 2:
   Input k = 3, n = 9
   Output: [[1,2,6], [1,3,5], [2,3,4]]

   a. Describe a backtracking algorithm to solve the problem.

      This figure shows the tires for first position (highlighted), it will be repeated for remaining positions expanding the tree further. Checking fewer options as we go forward.



   b.

      Write the pseudocode for the algorithm that you described in a. Your function signature should be **combination(n, k)** and it should return an array of all the combinations as shown in the example.

```
helper(result, n, k, current_choices, index):
    if(n == 0 and len(current_choices)==k):
        result.append(make a deepcopy of current_choices)
        return

    elif(n < 0 or len(current_choices)==k):
        return #not valid

    for i in range(index,10):
        current_choices.append(i)
        helper(result, n - i,k, current_choices, i+1)

        backtrack by removing last added number in current_choices


combination(n,k):
    result = []
    helper(result, n, k, [], 1)
    return result
```

c. Implement the pseudocode to solve the problem. Name your file **Sum.py**

```python
def helper(result, n, k, current_choices, index):
    if(n == 0 and len(current_choices)==k):
        result.append(list(current_choices)) #making deepcopy
        return

    elif (n < 0 or len(current_choices)==k):
        return #not valid

    for i in range(index,10):
        current_choices.append(i)
        helper(result, n - i,k, current_choices, i+1)

        #backtrack
        current_choices.pop()

def combination(n,k):
    result = []
    helper(result, n, k, [], 1)
    return result
```

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.

a. 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.

b. 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
```

c. 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)