Problem 1: Backtracking Approach

From an array of positive values A, find all unique combinations summing to target integer S. Don't reuse values past their frequencies. Return an empty array if no solution exists. Implement a backtracking algorithm.

1a) Pseudocode

```
Start a procedure taking an input array and target sum.
Initialize an empty results array.
Create a dictionary with the frequencies of each value in the input array.
Remove all repeats from the input array. Sort it, as the memo now tracks frequencies.
Call a recursive helper function that takes the input array, an indice integer, the results array, the target sum, an empty combination array, and the memo dictionary:
... Loop through the input array.
... If the input value's frequency is above zero and the input value is lower or equal to the current target sum:
... ... Append the input value to the combination array.
... ... Decrement the memo frequency for the input value by one.
... ... Call the helper function again.
... ... Increment the memo frequency for the input value by one, so new combinations can use it.
... ... Remove the last element of the combination array, so new input values can be considered.
... If the target sum is zero, add the combination array and stop.
... Else if the target sum is below zero, stop.
Return the final result with all unique combinations.
```

1b) Solution

See external file Amount.py

1c) Time Complexity: Polynomial — $O(n^k)$

The time complexity is polynomial $O(n^k)$, where n is the input size, and k is the target sum. The code loops for n, considering the remaining values up to k in a worst-case scenario, as each value is decremented from k.

Problem 2: Greedy Algorithm

A group of dogs n have hunger levels $[1, ...h]$, where h is that dog's appetite. There are m biscuits with sizes $[1, ...s]$, where s is the biscuit's size.

Find how many dogs one can feed, whether zero or more. Don't reuse biscuits or feed a dog twice.

2a) Pseudocode

```
Start a procedure taking input arrays for dogs' hunger levels and biscuit sizes.

Create an empty results array.

Sort the two input arrays by descending order for the greedy algorithm aspect — i.e., to choose each local
solution optimally, dogs must have the biscuit size closest to their hunger levels.

Loop the hunger levels input array:

... Initialize an indice at zero.

... While this indice is less than the total biscuits:

... ... If the biscuit is larger or equal to the dog's hunger level:

... ... ... Add the hunger level to the results array.

... ... ... Remove the biscuit from the second input array, just to avoid repeat feedings.

... ... Either leave the loop, or increment the indice by one.

The length of the results array is how many dogs can eat.
```

2b) Solution

See external file FeedDog.py

2c) Time Complexity: Polynomial — $O(n \cdot m)$

The time complexity is polynomial $O(n \cdot m)$, where n is the hunger_level array, and m is the biscuit_sizes array. Two sort built-ins in the code are $O(n \cdot log_n)$, but the complexity is still dictated by the input array sizes.

One array can be much larger or smaller than the other, and the code iterates through both with a nested *while* in a *for* loop. The procedure iterates linearly, making the optimal choice at each step for the greedy algorithm in choosing the smallest possible biscuit for each dog while still feeding them.