**Kristin Schaefer**

**HW5 : Backtracking and Greedy Algorithms**

**Sources:**

**Problem 2**

**1. https://stackoverflow.com/questions/48759175/what-is-the-space-complexity-of-the-python-sort**

**2. https://en.wikipedia.org/wiki/Timsort**

### 1. Backtracking

#### a. description

A target number *n* and a combination length *k* are passed to the function combination(). As a recursive approach is used for this backtracking problem, the combination() function makes use of a helper function combination_helper(). Before calling the helper function, an array, *n_arr*, of integers from *1 to max(n, 9)* is created. We compare the max of *n* and 9 because *k* can be 1-9, but if *n* < 9, than the elements which make up it's sum cannot be greater than 9. Next a function call is made to combination_helper(*n, k, 0, [ ], n, [ ]*) to initialize the backtracking algorithm.

In combination_helper() we iterate through each element in the *n_arr* with a for loop that begins at 0 and goes to *n-k*. If the length of the combination exceeds k elements or if the sum (held in *remainder*) of the combination is greater than the target *n*, then we backtrack to the last selection and continue trying the next possible choice. If the sum (held in *remainder*) of the elements is equal to the target *n*, then there is a comparison if the length of the combination is equal to *k*. If the combination is not equal to *k*, then it is too short, and we backtrack to the previous selection. However, if the combination is equal to *k*, then the combination meets the requirements and is appended to the *result* array.

#### b. pseudocode

```
def combination_helper(n, k, start, result, remainder, combination):

        if len(combination) = k+1:
                return                                  # combination length exceeded k
        if remainder = 0:
                if len(combination) = k:
                        append combination to result
                return                                  # if len != k, then combination will be too short so only return
        else if remainder < 0:
                return                                  # sum exceeded the target

        for i in range(start, len(n)-k):
                append n[i] to combination
                combination_helper(n, k, i+1, result, remainder-n[i], combination)
                combination.pop()                       # backtrack
```

```
def combination(n, k):

        result = [ ]

        # turn n into an array of integers 1, 2, ..., max(n, 9)
                n_arr = [x for x in range(1, max(n, 9))]

        combination_helper(n_arr, k, 0, result, n, [])

        print(result)
```

**Kristin Schaefer**

**HW5 : Backtracking and Greedy Algorithms**

**2. Greedy algorithm**

a. description

The feedDog() function takes two arrays: hunger_level[*1...n*] and biscuit_size[*1...m*]. First we sort both arrays in ascending order with the built-in Python sort method. Next we declare a variable *num_dogs* to hold the number of dogs that have been fed, and this is initialized to 0. We also declare the variables *i* and *j* to represent the current dog and the current biscuit respectively. We initialize both *i* and *j* to 0. Then we use a while loop to iterate through each dog, represented in the *hunger_level* array. If the *hunger_level[i]* of the dog is <= to the *biscuit_size[j]*, then the dog can be fed, so num_dogs, is incremented. *i* and *j* are also incremented because we can move on to the next dog and the next biscuit. However, if the *hunger_level[i]* of the dog is > the *biscuit_size[j]*, then the dog cannot be fed with the current biscuit because it's *hunger_level[i]* will not be satisfied, so we increment *j*, the current biscuit, and see if the next biscuit will satisfy the *hunger_level[i]* of the current dog. After the while loop is complete because there are no more dogs or biscuits, then *num_dogs* is returned.

b. time complexity

The function feedDog(hunger_level[*1...n*], biscuit_size[*1...m*]) sorts two arrays: hunger_level[*1...n*] and biscuit_size[*1...m*].

To sort both arrays I used the Python sort method, which is Timsort, so 2 * O(nlogn) => O(nlogn).