CS427 Assignment #7
Student name: _____
1. Textbook Pg.290 Ex1
2. Textbook Pg.297 Ex5
3.Ones and Zeroes
4. Coin Change

Rules:

- Each pseudocode must specify its input and output.
- Necessary comments should be included.
- When analyzing time/space complexity, specific reasons must be provided.
- For proof questions, a specific derivation process is required. Please refer to the example in the slide.

1. What does dynamic programming have in common with divide-and-conquer? What is a principal difference between them? (20pts)
(You could use some classic algorithm problems to support your point of view.)

Commonalities:

- Both principles aim to reach a solution by breaking a larger problem into smaller subproblems

Differences:

- Dynamic Programming solves subproblems that overlap and so it stores the results in a list or matrix to avoid recalculation
  - Example: 0/1 Knapsack problem builds an optimal value table where many subsets of items repeat in different branches of recursion
- Divide and Conquer solves subproblems independently and combines them into a final solution
  - Example: Merge Sort repeatedly splits an array into two halves, sorts them separately and merges them back together

2. Design a dynamic programming algorithm for the version of the knapsack problem in which there are unlimited quantities of copies for each of the n item kinds given. Indicate the time and space efficiency of the algorithm. (20pts)

Please provide your algorithm with input & output, necessary comments and time/space analysis. (Pseudocode will be fine)

```
// Input: int array containing item weights, corresponding int array containing item values and the weight capacity (W) of the knapsack
// Output: The maximum value of possible items under the capacity
public int unboundedKnapsack(int[] weights, int[] values, int W) {
        int n = weights.length;

        // dp[C] = maximum value achievable with capacity exactly C
        int[] dp = new int[W + 1];

        // Base case: dp[0] = 0 (by default all other entries start at 0 too).
        dp[0] = 0;

        // For each capacity C from 1 to W, compute dp[C]
        for (int C = 1; C <= W; C++) {

            int bestValue = 0;

            // Try taking each item i (possibly multiple times)
            for (int i = 0; i < n; i++) {

                if (weights[i] <= C) {
                    // If we take item i once, remaining capacity is (C - weights[i]).
                    // Because it's unbounded, we can reuse item i again via dp[C - weights[i]].
                    int candidateValue = values[i] + dp[C - weights[i]];

                    if (candidateValue > bestValue) {
                        bestValue = candidateValue;
                    }
                }
            }

            dp[C] = bestValue;
```

```
        }

        return dp[W];
    }
// Time Complexity O(n * W) – we must calculate all the weights by each possible capacity.
// Space Complexity O(W) – dp table grows depending on our capacity.
```

3. You are given an array of binary strings strs and two integers m and n. Return the size of the largest subset of strs such that there are at most m 0's and n 1's in the subset. A set x is a subset of a set y if all elements of x are also elements of y. (30pts)

Please provide your DP algorithm with input & output, necessary comments and time/space analysis. (Pseudocode will be fine). (20pts)

```
// Input: An array of binary strings along with the maximum number of 0s and 1s
// Output: The size of largest subset that that contains at most m 0's and n 1's
class Solution {
    public int findMaxForm(String[] strs, int m, int n) {
    // dp[i][j] = max number of strings we can pick
    // using at most i zeros and j ones
        int[][] dp = new int[m + 1][n + 1];

        // Process each string once (0/1 knapsack style)
        for (String s : strs) {

            // Count zeros and ones in this string
            int zeros = 0;
            int ones = 0;

            for (char c : s.toCharArray()) {
                if (c == '0') {
                    zeros++;
                } else {
                    ones++;
                }
            }
```

```
            // Update dp in reverse so we don't reuse the same string multiple times
            for (int i = m; i >= zeros; i--) {
                for (int j = n; j >= ones; j--) {
                    // Option 1: don't take this string -> dp[i][j]
                    // Option 2: take this string -> 1 + dp[i - zeros][j - ones]
                    dp[i][j] = Math.max(dp[i][j], 1 + dp[i - zeros][j - ones]);
                }
            }
        }

        return dp[m][n];
    }
}
/* Time Complexity O(k * m * n) where k = strs.length
        - The algorithm's complexity pull is determined by the strength of these
          inputs
Space Complexity O(m * n)
        Our dp array stores both ones and zeros
*/
```

Display a table to show your updates at each step. Please use Example 1 as the input case. (10pts)

strs = [“10”,”0001”,”111001”,”1”,”0”] m = 5, n = 3

(0/1)	0	1	2	3
0	0	1	1	1
1	1	2	2	2
2	1	2	3	3
3	1	2	3	3
4	1	2	3	3
5	1	2	3	4

Example 1:

Input: strs = ["10","0001","111001","1","0"], m = 5, n = 3
Output: 4
Explanation: The largest subset with at most 5 0's and 3 1's is {"10", "0001", "1", "0"}, so the answer is 4.
Other valid but smaller subsets include {"0001", "1"} and {"10", "1", "0"}. {"111001"} is an invalid subset because it contains 4 1's, greater than the maximum of 3.

Example 2:
Input: strs = ["10","0","1"], m = 1, n = 1
Output: 2
Explanation: The largest subset is {"0", "1"}, so the answer is 2.

4. You are given an integer array *coins* representing coins of different denominations and an integer *amount* representing a total amount of money.

Return the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1.

You may assume that you have an infinite number of each kind of coin.

Example 1:

Input: coins = [1,2,5], amount = 11
Output: 3
Explanation: 11 = 5 + 5 + 1

Example 2:
Input: coins = [2], amount = 3
Output: -1

Example 3:
Input: coins = [1,2,3,5], amount = 18
Output: 4

Please provide your DP algorithm with input & output, necessary comments and time/space analysis. (Pseudocode will be fine). (20pts)

```java
// Input: an array containing coin denominations and the total amount
// Output: the fewest number of coins required to form the amount
class Solution {
    public int coinChange(int[] coins, int amount) {
        // Edge case: no coins
        if (amount == 0) {
            return 0;
        }

        // dp[a] = minimum number of coins needed to make the amount
        int[] dp = new int[amount + 1];

        // Use a sentinel "infinity" value: amount + 1 is safe because
        // we can never need more than 'amount' coins of denomination 1.
        int INF = amount + 1;

        // Initialize dp array
        for (int a = 0; a <= amount; a++) {
            dp[a] = INF;
        }

        dp[0] = 0; // base case

        // Process each coin
        for (int coin : coins) {
            // For this coin, update all reachable amounts
            for (int a = coin; a <= amount; a++) {
                // If we can make (a - coin), try using this coin once more
                dp[a] = Math.min(dp[a], 1 + dp[a - coin]);
            }
        }

        // If dp[amount] is still INF, it means we couldn't form the 'amount'
        return dp[amount] > amount ? -1 : dp[amount];
    }
}
// Time Complexity O(coins.length * amount): iterates over both the total amount
of coins and the possible amounts from coin to amount
// Space Complexity O(amount): length of the dp array is amount + 1
```

Display a table to show your updates at each step. Please use Example 3 as the input case. (10pts)

Input: coins = [1,2,3,5], amount = 18
Output: 4

Step/amt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Initialization	0	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF
Coin "1" Processed	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Coin "2" Processed	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8	8	9	9
Coin "3" Processed	0	1	1	1	2	2	2	3	3	3	4	4	4	5	5	5	6	6	6
Coin "5" Processed	0	1	1	1	2	1	2	2	2	3	2	3	3	3	4	3	4	4	4