Name: Jungbin Choi — COMP 550-001

Due October 13, 2023 — Homework 3 (13 points)

Notes. Collaboration is allowed, and you may use any resources you'd like. You may call any algorithm given in the notes without providing details. However, you should write and submit your solutions individually, and each submission will be graded individually. In your submission, you should list your collaborators (including course staff) and cite every resource you use. For each problem, I encourage you to attempt it for at least 30 minutes before consulting any people or resources.

Your submission should be a single PDF file. I recommend using Overleaf; a template is on Canvas. If your answers are not clear, you might not get full credit. You should describe your algorithms in pseudocode similar to the examples given in the lectures/notes. I recommend also including a brief description in full sentences.

Problem 1 (5 points). Suppose there are an unlimited number of dimes (10 cents each), nickels (5 cents), and pennies (1 cent). We are a given an integer n, and we want to select the fewest number of coins such that their total value is n cents. The greedy algorithm is to pick as many dimes as possible (i.e., without exceeding n), then as many nickels as possible, then as many pennies as possible.

a. (2p) Implement the greedy algorithm in pseudocode; the level of detail should be similar to the corresponding code in Python. The algorithm should return 3 integers (the number of dimes, nickels, and pennies) and its running time should be $O(n)$.

b. (2p) Prove that the greedy algorithm is correct. Hint: Start by proving that the algorithm chooses the same number of dimes as an optimal solution.

c. (1p) Analyze the running time of your algorithm.

Bonus (0 points): What goes wrong in your proof if the coins had values $(4, 3, 1)$ instead of $(10, 5, 1)$?

Solution.

a.
```
ALG(n):
    int dimes, nickels, pennies = 0, 0, 0
    while n != 0:
        if n > 10:
            dimes += 1
            n -= 10
        else if n > 5:
            nickels += 1
            n -= 5
        else:
            pennies += 1
        n -= 1
    return dimes, nickels, pennies
```

b. Image the value n to be a total point, and that the algorithm keeps taking points off until the total becomes 0. First, you check if the value of dimes (10) can be taken out from the total ($n > 10$), and if it can be do so. This will make it so that if a value of a dime cannot be taken out, then the number of dimes will not increase, but if it can be taken out, then the number of dimes will increase until it no longer can. Then, once the total has dipped below 10, then the next biggest denomination can be taken out from it, the nickel, if and only if the total is greater than 5. Lastly, once the total is below 5, then the rest will be converted into pennies.

c. The while loop will run as long as n is above 0, and it is evident that the sum of the number of dimes, nickels, and pennies equal the number of times the while loop will iterate. Thus, the running time will be $O(\frac{n}{dimes+nickels+pennies})$ which simplifies down to $O(n)$

Problem 2 (4 points). Suppose we are given an array A of n positive integers that represent the values of n different coins. There is an unlimited supply of each coin. We are also given an integer v; our goal is to select the fewest number of coins such that their total value is v cents. For example, if $A = [1, 5, 10]$, then in the previous problem, we showed that the greedy algorithm is optimal. You may assume $A[1] = 1$, but you should not assume that A is sorted.

a. (2p) Give an $O(nv)$-time DP algorithm for this problem. In particular, you should define a subproblem, state the recurrence, and write the corresponding pseudocode. Your algorithm should return the fewest number of coins we need to reach value n.

b. (1p) Prove that your algorithm is correct. In particular, you only need to explain why your recurrence is true.

c. (1p) Analyze the running time of your algorithm.

Solution.

a. **Subproblem:** $OPT[i][j]$ = Optimal (minimum) number of coins possible in the instance of having $\{1, ..., i\}$ coins and a total needed value of j

   Base Case 1: $OPT[1:n][0] = 0$ (When the total value to reach is zero, there are no coins that can do that)
   Base Case 2: $OPT[1:n][1] = 1$ (When the total value to reach is one, only one of coin $A[1]$ can do that)

   Recursion: If $A[i] > j$: $OPT[i][j] = OPT[i-1][j]$
   Else If $j \bmod A[i] = 0$: $OPT[i][j] = min(OPT[i-1][j], OPT[i][j-A[i]] + 1)$
   Else: $OPT[i][j] = OPT[i][j-1] + 1$

Pseudocode:

```
ALG(A, v):
   d[][] = [[inf] * n] * (v + 1)   // 2D Array w/ n rows & v+1 columns
   d[1:n][0] = 0   // Base Case 1
   d[1:n][1] = 1   // Base Case 2

   for j = 2 ... v:
       for i = 1 ... n:
           if A[i] > j:   // If coin's value > capacity
               d[i][j] = d[i-1][j]
           else if j % A[i] == 0:   // If capacity is multiple of coin's value
               d[i][j] = min(d[i-1][j], d[i][j-A[i]] + 1)
           else:
               d[i][j] = d[i][j-1] + 1

   return d[n][v+1]
```

b. I'll explain the recurrence in order (referred to as Case 1, 2, and 3). For Case 1, if the coin's value is bigger than the needed total v, then that coin cannot be counted because it would immediately go over, therefore the previous value $d[i-1][j]$ (which represents the minimum number of coins needed to reach v without using the current coin) is used. For Case 2, if v is a multiple of the current coin, then the minimum number of coins will be smaller than at least one before it because of $A[1]$ always being 1 (thus reaching v with less coins than just all ones), in which case just get the number of coins needed from the last time v was a multiple and add one to it to account for the current place (or just get the value before the current if the previous value is smaller than the number gained from the method described). Lastly for Case 3, if none of the criteria is met, then just add one to the minimum number found at the same coin but at $j-1$ because that represents the minimum number needed to reach the value $j-1$ and then just adding a one to the count to account for adding in a penny.

c. The array initialization will take $O(n*(v+1))$ for n rows and $v+1$ columns (accounts for j = 0). The next two operations will cost $O(1)$ time, so will be left out. The first for loop will take $O(v-1)$ iterations for looping through values 2 through v. Per iteration, the nested for loop will take $O(n)$ time to check through all the coins in array A. Everything inside the nested for loop will take $O(1)$ time, therefore will not be important. Thus, the the loops will take a total of $O((v-1)*n)$ time. Adding all the running times together, you get $O((n*(v+1)) + ((v-1)*n)$, which simplifies down to $O(nv+n+vn-n)$, which further simplifies down to $O(nv)$.

Problem 3 (4 points). Suppose we're given a string A of length n that only contains lowercase letters. We're also given a "scoring" function f whose input can be any string and output is a positive number. Strings with a higher value of f are "better" than strings with a lower value. Our goal is to split A into k substrings $(s_1, s_2, \ldots, s_k)$ such that the total score $\sum_{i=1}^{k} f(s_i)$ is maximized. We get to choose the value of k, and we can assume that calling f always takes $O(1)$ time regardless of the input.

Example. Suppose A = "ilikealgorithms". One feasible solution is to split A into two parts: (ilikea, lgorithms). The score of this split is f(ilikea) + f(lgorithms) which would be rather low (e.g., $2 + 1 = 3$), since "ilikea" and "lgorithms" are not proper words. Another feasible solution is to split A into three parts: (i, like, algorithms). In this case, the score would be f(i) + f(like) + f(algorithms), which would be much higher (e.g., $8 + 12 + 15 = 35$).

a. (2p) Give an $O(n^2)$-time DP algorithm for this problem. In particular, you should define a subproblem, state the recurrence, and write the corresponding pseudocode. Your algorithm should return the maximum score we can achieve by partitioning A into substrings.

b. (1p) Prove that your algorithm is correct. In particular, you only need to explain why your recurrence is true.

c. (1p) Analyze the running time of your algorithm.

Solution.

a. **Subproblem:** $OPT[i]$ = The maximum score possible when given string $A[1:i]$.

   Base Case: $OPT[1]$ = f(A[1]) (The the value for the first character in the string).

   Recursion: $OPT[i] = OPT[i -$ length of subarray from the end of the string] + f(the subarray in question that gives the maximum)

   For example in "ilikealgorithm", if $i = 5$ (the letter e), then the the subarray that can be created from i down and also gives the maximum score would be "like", so the equation for the recursion would be $OPT[i-4] + f(\text{"}like\text{"})$.

   Pseudocode:

```
ALG(A):
    d = [0] * n
    d[1] = f(A[1])

    for i = 2 ... n:
        max_score = f(A[1:i])   // A[1:i] = String from index 1 to i, inclusive

        for j = 0 ... i-2:
            curr_score = f(A[i-j:i]) + d[i-j-1]
            max_score = max(curr_score, max_score)

        d[i] = max_score

    return d[n]
```

b. The recursion works because for every index in the string, the recursion will check the score of the substrings that can be created backwards from the end of the string towards the front. For each substring score found this way, it'll add it to the value found in $OPT[1 :$ wherever the backwards substring stopped - 1] (or in English the maximum score possible found for the substring $A[1 :$ wherever the backwards substring stopped - 1]) and find the new maximum. It'll check every iteration of the backwards substring, and because we have already found the maximum score possible for the substrings created before it, then just by adding the new value found it'll create the newest maximum for string $A[i : 1]$.

c. The array initialization will take $O(n)$. The first for loop will take $O(n)$ because it'll essentially iterate through the entire string. Per iteration, the inner for loop will take $O(n)$ as well because it'll run from 0 to $i - 2$, which at max can be $n - 2$. Thus, the nested loops will take a total of $O(n^2)$ time and putting it all together the total running time is $O(n + n^2)$, which simplifies down to $O(n^2)$.