Design and Analysis of Algorithms

CSCI 3650 Spring 2018 Extra Credit Work

You have to solve the following problems and submit them. Make sure all your answer sheets are stapled or otherwise bound together. Your solutions are due at the beginning of the class on Tuesday, the 17th April.

You are expected to work individually on this. You can consult the textbook, class notes and the slides. However, you are not allowed to "Google up" the answers. You are expected to write **clear pseudocode** for all questions that are asking you to develop algorithms. It is not enough to describe your idea in English. However, you don't have to write actual programs.

The purpose of this assignment is to give you experience in designing algorithms based on the dynamic programming technique (and to pull up the scores). There is only one problem in this assignment, although it has several parts. Your task is to tackle this problem using the dynamic programming technique.

The problem that we will consider is the so called **coin changing** problem that we discussed in class. Consider the problem of developing an algorithm that will determine how many of each type of coin,such as quarters, dimes and so forth, it takes to equal the given amount *such that the number of coins used is minimal.*

To begin with let us assume that the coins available are dollars, quarters, dimes, nickels and pennies only. For example, 77 cents can be changed to 3 quarters and two pennies. The number of coins used is 5 and you should convince yourself that we can not total 77 cents with less than 5 coins.

To tender change for a given amount with the minimum number of coins we would first like to give out as many quarters as possible. For the remaining amount we would like to give out as many dimes as possible, and so on. Finally we hand out pennies for whats left. This method comes naturally and instinctively to us. We may have done it before without realizing that we were handing out the minimum possible number of coins for the amount in question. Computer scientists call this method the greedy method. It is no surprise that such a method comes naturally to us humans!. The greedy method is often employed to solve many an apparently intractable problems in computer science and you would be surprised how often it works. The basic method is to grab as much as possible at every step in the solution.

As we proved in class, with the given values for the coins, this greedy algorithm always produces an optimal solution to our problem. However, with a different series of values for the coins, the greedy algorithm may not work.

Suppose the country of Weirdonia has three types of coins worth 1 cent, 9 cents and 10 cents. To make up an amount of 36 cents, the greedy algorithm would use

9 coins (3 coins of 10 cents each and 6 coins of 1 cent each). One could actually do better, by using 4 coins of 9 cents each.

We can develop an algorithm using dynamic programming that always works, no matter what the values of the coins are. You are going to do that for this problem. But, I will guide you through the process so that you don't get overwhelmed.

Suppose the currency we are using has available coins of n different denominations. Let a coin of denomination i, $1 \leq i \leq n$, have value d_i. We assume that each $d_i > 0$. Further, we assume that the denominations are presented to us in increasing order i.e., $d_1 < d_2 < d_3 < \ldots < d_n$. Finally, we assume that d_1 is 1 so that it is feasible to make up any amount. Let m be the amount for which we want to determine the minimal number of coins needed.

Define $c[i, j]$ to be the minimum number of coins required to pay an amount of j units, $0 \leq j \leq m$, using only coins of denominations 1 to i, $1 \leq i \leq n$. Now, carry out the following steps.

As a first task, one should develop a recurrence relation for $c[i, j]$ by analyzing the structure of the optimal solution. Observe that, if $j < d_i$, then it is simply not possible to pick a coin of denomination i as we would already be exceeding the value. Hence, in this case $c[i, j] = c[i-1, j]$. On the other hand, if $j \geq d_i$, we could potentially pick a coin of denomination i. We now have a choice. If we choose not to pick a coin of denomination i, the best we can do is to use $c[i-1, j]$ coins to pay off. If we choose to pick a coin of denomination i, we still need to distribute $j - d_i$ amount and the minimum number of coins needed for that would be $c[i, j - d_i]$. But, then the total number of coins we used is $1 + c[i, j - d_i]$, as we picked a coin of denomination i first. So, the minimum number of coins needed to pay off j units is

$$c[i, j] = \begin{cases} c[i-1, j] & \text{if } j < d_i \\ \min(c[i-1, j], 1 + c[i, j - d_i]) & \text{if } j \geq d_i \end{cases}$$

The base conditions are $c[i, 0] = 0$ for all $1 \leq i \leq n$ and $c[1, j] = j$ for all $j \geq 0$. I hope, the base conditions need no explanation.

1. Using the recurrence relation developed, write a simple recursive algorithm to compute $c[i, j]$. Note that, by simply calling this recursive algorithm with parameters n and m, we can determine the minimum number of coins needed to pay an amount of m units using the available coins of n different denominations. [10 Pts.]

2. The English coinage before decimalization included half-crowns (30 pence), florins (24 pence), shillings (12 pence), sixpences (6 pence) and pennies. The coinage also included several other coins such as farthing and three pence, but we will ignore them to limit the number of denominations to 5. Suppose, I want to dispense an amount of 48 pence using the coinage of 5 different denominations, we will call your recursive program with parameters $(5, 48)$. Draw the **complete** recursion tree generated by this call. For each node in the tree, simply write the parameters of that recursive call. $c(5, 48)$ will be the root of the recursion tree. Observe that it involves lots of repeated subproblems. Clearly,

mark the repeated subproblems in your recursion tree. If a bigger subproblem is repeating, do not worry about marking the repetitions of the sub-subproblems of that bigger subproblem. [10 Pts.]

3. As you observed in your answer to the previous part, typically the simple implementation would have terrible computational complexity as it involves solving repeated subproblems. Dynamic programming method is to avoid doing solving repeated subproblems and in this part you will build an implementation for the coin changing problem using the dynamic programming technique. Design an improvised recursive algorithm to solve the same problem that ensures that no subproblem is solved more than once. Basically maintain an appropriate table and use the *memoization* technique. This is also sometimes called as the *top down* approach. (10 Pts.)

4. In practice, even when there are no repeated subproblems recursive implementations involves considerable overhead as the system has to set up stack space etc. Hence, non-recursive implementations are often preferred even if the worst case complexity is the same. Design an algorithm using the *bottom up* approach for solving the same problem using the dynamic programming technique. (10 Pts.)

5. Illustrate the operation of your bottom up algorithm by means of the following example. The different denominations available are 1 cent, 4 cents and 6 cents. The amount that we are concerned with is 15 cents. Show, the entire table and do not just give me the final answer. (10 Pts.)

6. Assuming that the table $C[1..n, 0..m]$ is already built, devise an algorithm to determine an optimal solution to the coin changing problem. Note that the algorithms that you developed in the previous part only computed the value of an optimal solution and not an optimal solution itself. Specifically, your algorithm should output an integer array $X[1..n]$ with the property that $X[i]$ is the number of coins of i^{th} denomination that could be used while disbursing an amount of m cents optimally. (10 Pts.)