DP Making Change
Tom Bohbot
April 2021

1) Explain the dynamic programming approach taken to solve the "make change" problem: My program iterates (let this variable be i) from the beginning of the list of denominations until the end, and has an inner loop (let this variable be j) which iterates from 1 until the integer N+1 inputted in the method. As each denomination is being iterated over I check if den_i equals j. If it does then I have found an exact match and store 1 in c[i][j] (there cannot exist a more optimal answer as 1 is the lowest number possible). The dynamic aspect of the algorithm takes place in the following step. My algorithm will now check if a more optimal solution exists through using a previous solution. If I have a solution that exists for (N+1)-j then I will check if its solution plus one extra coin is more efficient then what I previously had. To be more efficient means that the solution is no longer zero, and if it is not zero then it is less then what it was previously. If N still has no solution found then it is set to pseudo_infinity. When i completes its final iteration c[amount_of_coins_inputted][N+1] will either be set to its optimal solution or pseudo_infinity if no solution exists.

   The runtime of this code is O(N*denominations.length) as that is what the nested loops represent and everything else in the code is made up of constant time operations. The space complexity of this code is O(N*denominations.length) since I must create and return a double dimension array that is of size [denominations.length][N+1].

2) Supply a fully-specified recurrence that solves the problem:

$$
c[i][j] = \begin{cases}
1 & \text{if } den_i = j \\
\min \begin{cases} c[i-1][j] \\ c[i][j-den_i]+1 \end{cases} & \text{if } den_i < j \text{ and } c[i][j] \text{ != } 0 \text{ and } c[i][j-den_i]+1 \text{ != neither } \infty \text{ or } 0 \\
c[i][j-den_i]+1 & \text{if } den_i < j \text{ and } c[i][j] = 0 \text{ and } c[i][j-den_i]+1 \text{ != neither } \infty \text{ or } 0 \\
\infty & \text{if } den_i > j \text{ and } c[i-1][j] = \text{ either } \infty \text{ or } 0
\end{cases}
$$

3) A brief explanation of your “payout” algorithm and the “Big- O” space and computation requirements:

My payout method initially verifies that an optimal solution exists and if the array c inputted has pseudo infinity set to N then I throw an exception. The way I find how many coins of each denomination must be returned is through a while loop. Before my while loop begins I use two variables being N and i. I also use an array of size denominations.length to track how many of each denomination is used. Initially I set i to the last denomination’s index in the list. My loop checks when the last time is that c[i][N] changes as this represents an element that must be used. Since I am looping backwards, I am really looking for the first time my loop sees a change. Once I see a change I decrement N by by denominations[i]. I increment my return array’s ith element by 1 as one coin is confirmed to be used, and I reset i to the last denomination in the list’s index. If I don’t find a change then I simple decrement i by 1. I repeat this process until either i or N is less than or equal to 0. Once my program breaks I would have checked every index that my program uses.

In essence I used dynamic programming to find an element that must have been used in N, and then once I find that one element I look for the element that must have been used, I look for the one element that must have been used for N-element_just_found. I repeat this process until I have either exhausted all denominations or N is <= 0 as that means that I found the answer. Once I have filled my array up of how many times each denomination is used, and the while loop breaks I am ready to simply return the array.

This method run in O(denominations.length*N) time as i loops from denominations.length to 1. However, i can be reset to denominations.length only N times as only N changes can occur, since the last that N can be decremented by is 1. The space complexity of this algorithm is O(denominations.length) since the only variable used that contains more than one element is my array that tracks how many of each denomination is used.