# Project 2: Coin Change

## Dynamic Programming Table

Our changedp algorithm loops starting from 1 up to A. For each iteration of the loop, the algorithm loops through each coin value, starting with the lowest (1), and subtracts the coin value from the table row number to find the minimum count for the current A minus the current coin value, plus one for the current coin. If this count is less than the minimum so far, it sets the minimum to the new low and stores the index of the coin value. After the coin value loop terminates, the minimum count is set to the lowest possible count for the current A. It then adds a new row to the table by creating a copy of the row for the current A minus the coin value that produced the minimum count, and increments the column of the table corresponding to that coin value. When the outer loop terminates, the final row of the table contains the correct counts for each value of coin.

## Pseudocode

### Algorithm 1: Brute Force (changeslow)

**for** i = array.length downto 1

**if** x == A

coin\_array = [0 : length of array.length]

coin\_arr[i] = coin\_arr[i] + 1

**return** coin\_arr, 1

temp\_array, coin\_array = [], []

temp\_count, coin\_count = 0, 0

**for** i = array.length downto 1

**if** array[i] < A

temp\_array, temp\_count = changeslow(array, A – array[i])

temp\_count = temp\_count + 1

temp\_array[i] = temp\_array[i] + 1

**if** coin\_count = 0 or temp\_count < coin\_count

coin\_count = temp\_count

coin\_array = temp\_array

**return** coin\_array, coin\_count

### Algorithm 2: Greedy Algorithm (changegreedy)

change = A

coin\_count = 0

coin\_array = [array of size array.length initialized to 0]

**for** i = coin\_values.length downto 1

**if** coin\_values[i] <= change

coin\_array[i] = floor(change/coin\_values[i])

coin\_count = coin\_count + coin\_array[i]

change = change MOD coin\_values[i]

**return** coin\_array, coin\_count

### Algorithm 3: Dynamic Programming (changedp)

min\_counts = 1D array with A elements initialized to 0

min\_coins = [0..A, 0..array.length - 1] 2D array initialized to 0

**for** i = 1 to A

min\_count = NULL

coin\_index = NULL

**for** j = 0 to array.length – 1

coin\_val = array[j]

current\_count = min\_counts[i – coin\_val] + 1

**if** coin\_val <= i and (min\_count == NULL or current\_count < min\_count)

min\_count = current\_count

coin\_index = j

min\_coins[i] = copy of min\_coins[i – 1]

min\_coins[i][coin\_index] = current\_coins[i][coin\_index] + 1

**return** min\_coins[A], min\_counts[A]

## Proof of Dynamic Programming Approach

Let T[v] be the minimum number of coins that can be used to produce the value v using only coins in the set V. The dynamic programming approach has the following recursive definition, where i is the index of the coin value in V. We also have that V[1] is always 1.

First, we can show that T[1] = minV[i]≤1{T[1 – V[i]] + 1} = T[0] + 1 = 1 because the only valid value in V that is less than or equal to 1 is V[1] = 1. Now we will assume that T[k] = minV[i]≤k{T[k – V[i]] + 1} for all 1 ≤ k ≤ v. We know that k – V[i] will always be some value j, where j < k because min{V[i]} is defined as 1. Furthermore, because we find the minimum only when V[i] ≤ k, we know that V[i] is at least 1 and at most k, therefore 0 ≤ j < k. By the optimal substructure property of the problem, it follows that T[j] is an optimal solution for the minimum number of coins to make j. Now we attempt to find T[j+1] = minV[i]≤j+1{T[j+1 – V[i]] + 1}. Because 1 ≤ V[i] ≤ j, we know that 0 ≤ j+1 – V[i] ≤ j. We also know that j < k, therefore j+1 – V[i] < k. Because j, V[i], and k are all integer values, it follows that j+1 ≤ k, which proves that T[j+1] is also an optimal substructure.

## Experimental Analysis

### Comparison of Number of Coins for Greedy and Dynamic

#### V = [1, 5, 10, 25, 50] and A = [2010, 2015, …, 2200]

Greedy and Dynamic Programming both returned the same values for each value of A. Brute Force did not complete for these values of A after 1 hour.

On a much smaller A [10, 15, 20, …, 50], all approaches provided identical answers.

#### V1 = [1, 2, 6, 12, 24, 48, 60], V2 = [1, 6, 13, 37, 150], and A = [2000, 2001, …, 2200]

For the first set of A, the Greedy algorithm occasionally required a higher number of coins. For most values, the algorithms returned the same result.

For the second set of A, the Greedy algorithm often required a higher number of coins.

On a smaller comparison set of A [15, 16, 17, …, 40] for V1, all algorithms returned the same values for each A.

On the same set of A [15, 16, 17, …, 40] for V2, Brute Force and Dynamic Programming both returned the same values, while Greedy occasionally returned a higher number of coins required.

#### V = [1, 2, 4, 6, 8, 10, 12, …, 30] and A = [2000, 2001, …, 2200]

For the full set of A, Greedy and Dynamic Programming returned the same values.

On the smaller set of A [15, 16, 17, …, 40], all algorithms returned the same values.

### Log-Log Plot

### Running Time by Denomination Count

For each of the graphs below, on the X axis, 1 is n=5, 2 is n=7, 3 is n=5, and 4 is n=30.

From plotting these graphs, the number of denominations seems to affect the Brute Force unpredictably (with 7 being far outside the range of the other running times). Dynamic Programming seems to run a little slower as you increase the number of denominations, with 1 and 3 (both n = 5) running faster than 2 (n = 7), and all three of those running faster than 4 (n = 30). Greedy ran incredibly quickly on all algorithms, and did not seem to be impacted by the number of denominations.

## Dynamic Vs. Greedy for Powers of p

If our coin values are V = [p0, p1, p2, …, pn], we can see that p2 = p(p1). In other words, we multiply a value by the root to get the next value, or divide by the root to get the previous value. For any value A, we would divide by the coin value with the largest power i such that pi ≤ A. There is no better combination of smaller values because all of the values are multiples of each other. Therefore, the greedy approach will always be more efficient.