# 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 = [0 : size of array.length]

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

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

change = change – (count \* coin\_values[i])

coin\_array[i] = count

coin\_count = coin\_count + count

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

### Algorithm 3: Dynamic Programming (changedp)

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

min\_coins = 2D array with array.length columns 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** array[j] <= 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 – 1], min\_counts[A – 1]

## Proof of Dynamic Programming Approach

The dynamic programming approach to finding the minimum number of coins that can be used to produce the desired change has the following recursive definition, where V is the set of coins available, v is the value to get the change for, and 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 of T[k].

## Experimental Analysis

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

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

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

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

### Log-Log Plot

### Running Time by Denomination Count

## Dynamic Vs. Greedy for Powers of p

The greedy algorithm fails when there are coin values ck and ck-1 such that . When coins are powers of p, we get , which holds while p ≥ 2. Therefore, the greedy approach always works under these circumstances.