Robert Ottolia

Andrew Brown

Luke Brewbaker

Stephen Krueger

CS 325 Winter 2016

**Project 2: Coin Change**

**1. Filling in the table using the Dynamic Programming approach.**

For our dynamic programming algorithm to solve the coin change problem, we compute the solutions to the smaller sub-problems first in a bottom up manner in a table. Based on the results in the table, then, the solution to the ‘top’ (original) problem is then computed. This is a valid way to fill the table because at each step, we are computing the optimal solution to subproblems that in turn can be used to find the optimal solution of the original problem. This technique can be used when the problem exhibits Optimal substructure, which in this, it does.

**2. Pseudocode for each Algorithm**

**2.1 Dynamic Programming Approach**

*coinList* Coin denomination input array

*minCoins*  Table that will be built bottom-up

*coinsUsed* Table that will keep track of coins used

**for** *i* = 0 to length of *total* + 1 **do:**

set table for use case of coin denominations of 1:

*minCoins[0][i]*  *i*

*coinsUsed[0][i]*  *i*

**for** *i = 1* to length of  *coinList* - 1 **do:**

**for** *j = 1* to length of *total + 1* **do:**

**if** *j* < *coinList[i]* **then:**

Current coin to big, get previous best coin total count:

*minCoins[i][j]* *minCoins[i – 1][j]*

**else if** *j >= coinList[i]* **then:**

See if current coin can get us to total with lower count:

*minCoins[i][j]* minimum between *minCoins[i – 1][j]* and *minCoins[i][j – coinList[i]]+1*

Add the coin to coin tracker:

*coinsUsed[i][j]* += 1

**endif**

**endfor**

**endfor**

**return** table position containing minimum # of coins *minCoins[len(coinList)-1][total]]*

**2.2 Greedy Algorithm Approach**

*coinList* Coin denomination input array

*total* Total value desired

*coinCount* Number of coins to reach total

*dictionaryCount* Empty dictionary to hold coin frequencies

**for** *i = 0* to length of *coinList* **do:**

Divide total sum by largest denomination (assuming *coinList* sorted):

*temp = total/coinList[i]*

Get remainder of total after division:

*total = total % coinList[i]*

Update coin count:

*coinCount* *coinCount + temp*

*dictionaryCount[coinList[i]]* = *temp*

**endfor**

**return** *coinCount, dictionaryCount*

**2.3 Brute Force / Recursive Approach**

*coinList* Coin denomination input array

*total* Total value desired

*coinDict* Empty dictionary that will hold coin frequencies

changeSlow(*coinList, total, dict*)

**if** *total* = 0 **then:**

return 0

**endif**

*res* variable to be returned; initially set to INT\_MAX

**for** *i = 0* to length of *coinList:*

**if** *coinList[i]* <= *total* **then:**

Recursively call changeSlow with *coinList[i]* subtracted from total

*temp* changeSlow(*coinList, total – coinList[i], dict)*

Update the coin count

**if** *res* **not equal** INT\_MAX **and** *temp + 1 < res* **then:**

*res* = *temp + 1*

**endif**

**if** *coinList[i]* **not** a key in *dict* **then:**

Set the key in the dictionary

*dict[coinValueList[i]]* = *1*

**else**

Increase the count associated with the key

*dict[coinList[i]]++*

**endif**

**endfor**

**3. Proof By Induction**

Prove: T[*v*] = min[V]<=*v*{T[*v* – V[i]] + 1},

T[0] is min. number of coins possible to make change for value *v.*

Given a set *D* = {*v1,…vm}* of coin denominations, let *f(n)* be the minimum number of coins (with repetition) in *D* needed to obtain sum *n*. Then, *f(n)* >= 0 for all *n* and *f(n)* = 0 when *n* = 0.

In the situation *n* > 0 and a way to obtain sum *n* with *f(n)* coins using at least one coin of denomination *v*k (at least one such *k* exists in this situation), removing this coin we obtain a way to determine *n* - *v*k and can conclude that:

*f(n - v*k*)* <= *f(n)* – 1 for at least 1 <= *k* <= *m*.

However, if 1 <= *k* <= *m* and *n* >= *v*k, we can obtain *n* with *f(n - v*k) + 1 coins by adding a *v*k  coin to an optimal way to get *n - v*k. As such, the below holds:

*f(n) <= f(n - v*k*) + 1* for all 1 <= *k* <= *m* with *n* >= *v*k.

Together, this gives us:

*f(n) =* min{ *f(n - v*k*) +* 1 | 1<= *k* <= *m*, *n* >= *v*k} for all *n* > 0 (1)

And case (1) together with *f(0)* = 0 is exactly what the dynamic programming algorithm uses to compute *f(n)* recursively.