Robert Ottolia

Andrew Brown

Luke Brewbaker

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.

**4.**

**DP and Greedy algorithms both return same results. Changeslow would not complete this scenario, so we ran it with smaller A sizes (and also compared DP and Greedy at those smaller sizes):**

Note: data for all three algorithms is here, but stacked on top of each other.

All algorithms returned the same results using these coin denominations and A sizes. It is easy to understand why—because we are incrementing the value needed by 5, and the coin denominations are US coins, which are also increments of 5 aside from the penny.

**5.**

For both of these, the red series is the greedy algorithm, and the blue is the dynamic algorithm. In the first chart, red and blue are overlapping, meaning they produced the same results. The reason for this is that the larger denominations are all divisible by 12, and since the values needed are so large, we are rarely, if ever, going to need to use the smaller denominations.

For the second chard, you do see a blue dot at 17, meaning the dynamic algorithm returned a 17, and the greedy algorithm returned a 20. This is where value needed is 2005. So the greedy algorithm would have gone 13\*15, then 37 \* 1, 1 \* 13, then 5 \* 1 = 2005. Alltogether 20 coins, because you are starting with the higher denominations. Whereas the dynamic algorithm will do 150 \* 13, + 37 \* 1, + 6 \* 3. 17 coins altogether. In this case, it worked out better to try every possible scenario, and the greedy algorithm was suboptimal.

Although changeslow did not work on this larger sizes, we ran them on smaller inputs, along with dynamic and greedy algorithms to compare:

For these smaller sizes of A, all algorithms returned the same responses. However we have already seen with the larger sizes of A, and the same demoninations [1, 6, 13, 37, 150], there are instances where the greedy algorithm is suboptimal.

**6.**

Blue represents dynamic programming results, and red represents greedy algorithm. In this case, we can see that for the most part they overlap, but there were two instances where dynamic programming returned a better results (smaller # of coins) than greedy algorithm. For A = 2004, the greedy algorithm chooses 66 \* 30, and then + 22, + 4. So 68 total coins. However, if you use 30 one less time, and go 65\* 30, + 28, +26, then it is 27 coins to reach 2004. Another example of the greedy algorithm being suboptimal.

Although slowchange could not handle such a large input, we did test the same denominations but with smaller sizes of A. We also ran the dynamic and greedy algorithms at these smaller input sizes to test side by side:

All three algorithms returned the same results. However we have already seen that with larger values of A, you run the risk of getting suboptimal results with the greedy algorithm.

**7. Graphs of Running Times as a Function of A**

Despite some aberrations, which could be explained by other processes taking up memory, the plots are very linear, and so a linear trendline was used. Therefore the running time would be a function of value needed \* size of coin denom array, or O(nA).

**7.2 Greedy Algorithm Results**

The greedy algorithm is best fitted to a linear trendline. The algorithm will inspect at most N coin denominations, so the run time is O(N).

**7.3 ChangeSlow Results**

The brute force algorithm is best fitted to an exponential trendline. It is clear to see that as input sizes grow, the algorithm very quickly takes exponentially longer to run.

**8.**

Brute force algorithm does go up drastically with increased coin array sizes. The more coins, the more work it is doing. Dynamic Programming algorithm showed increases as well, but they were linear as you would expect. The Greedy algorithm did show increases, but they were very small. This is not a surprise since often you will not iterate through the whole array.

**9. Situation when coin denomination sets are powers of P**

In situation where coin sets are powers of *p* (i.e. {*p0, p1, … pm*}; c > 1 , m >= 1), using a greedy algorithm will always result in an optimal solution. To see why this is the case, let solution *Sm-1* be the optimal solution when there are *p0, p1, … pm-1* denominations. Making a greedy choice for *p*m will then yield an optimal solution for *Sm*. Because all *p0, p1, … pm* are commonly divisible by *p,* any non-optimal solution can be migrated to an optimal solution by merging changes from *pi* to *pj* where *j > i.* The dynamic programming approach also gives the optimal solution.

**10. Determining whether greedy approach will result in optimal solution**

In order to determine the conditions in which using a greedy algorithm will result in an optimal solution with regard to the coin change problem, it is necessary to research the characteristics of the sets in which the greedy algorithm produces an optimal result and compare it to those sets in which it does not.

According to Kozen and Zaks[1], coin denomination sets in which a greedy algorithm produces an optimal solution for all amounts are called *canonical* systems. Then, for a non-canonical coin system, there exists an amount *c* in which using the greedy algorithm will produce a suboptimal amount of coins; and *c* is called a counterexample in this case.

Further, Kozen and Zaks determined that if a counterexample *x* exists, (for a coin set {*c*1, *c*2, … *c*m}), the smallest such *x* lies in the range:

*c3* + 1 < *x* < *cm + cm-1*

Consider the coin set {1, 3, 4}. According to the theorem above, the smallest example would lie in range 5 < *x* < 7. Here we see that *x* can be 6, and if the total is 6, then the greedy algorithm uses 3 coins to reach that total (4 + 1 + 1), while the optimal is 2 coins (3 + 3).

[1] *Optimal Bounds for the Change-Making Problem* Dexter Kozen and Shumel Zaks http://www.cs.cornell.edu/~kozen/papers/change.pdf