Oregon State University

CS325 Analysis of Algorithms

Fall 2015

Project 2

Authors:
Gerald Gale, Cierra Shawe, Grant Smith

Group 11

October 28, 2015

Table of Contents

1) Description: Filling Dynamic Programming Table

For creating a table for dynamic programming, we would just need to store the minimum number of coins for a specific amount of change. If we were to have the requirement of V = [1, 5, 10, 25, 50] with change of 75 then we would just end up with [0, 0, 0, 1, 1]. From there, we would just list the least required number of change for each increased amount of change to give back. If we did this for each subproblem, then the aggregate problem can reference any of these amounts for a given number of change to give the correct answer. Thus one does not need to store every possible combination of change that can be returned, just the minimum for each amount. An example of what the table could look like:

Red text is specifically just a comment, not in the actual table.

[0, 0, 0, 1, 1]	Input change: 75
2	
[1, 0, 0, 1, 1]	Input change: 76
3	
[2, 0, 0, 1, 1]	Input change: 77
4	

2) Pseudo Code of Algorithms

Brute Force or Divide and Conquer (Algo1)

```
def changeSlow(array, change):
    solution = []
    result  = []
    for i in range(0,len(array)+1):
        solution.append(0)

    minChange = change
    for i in range(0, change/array[-1]+1): #1
    for j in range(0, change/array[-2]+1):#2
    for k in range(0, change/array[-3]+1):#3
    for l in range(0, change/array[-4]+1):#4 ….
        if len(array) >= 5:
        for m in range(0, change/array[-5]+1):#5
            if (i*array[4] + j*array[3]+ k*array[2]+l*array[1]+m*array[0]) == change:
                solution[0] = m
                solution[1] = l
                solution[2] = k
                solution[3] = j
                solution[4] = i
                solution[5] = i+j+k+l+m
```

```
                        if solution[5] < minChange:
                                minChange = solution[5]
                else:
                        if (i*array[3] + j*array[2] + k*array[1]+ l*array[0]) == change:
                                solution[0] = l
                                solution[1] = k
                                solution[2] = j
                                solution[3] = i
                                solution[4] = i+j+k+l
                                if solution[4] < minChange:
                                        minChange = solution[4]
                                        result = solution

        return result
```

Note
This algorithm isn't very versatile, it's only designed to run up to 5 different denominations, however it could be modified to do more. It's just brute forcing the numbers, because the recursive version was going too deep and crashing the program.

Greedy Algorithm (Algo2):

```
changeGreedy(array, desiredChange)
        numberOfCoins to hold eventual result
        Set total to zero
        for i while 0 <= i <= length of array
                if(array[i] <= desiredChange)
                        numberOfCoins = desiredChange / array[i]
                        total = total + numberOfCoins
                        subtract accumulated total from desiredChange

        Add numberOfCoins to end of array.
        return array
```

Dynamic Programming Algorithm (Algo 3):

```
changeDP(currencyValuesArray, amountToReturn):
        minArray = [0]
        firstCoinArray = [0]
        coin = 0

        for j in range(1, amountToReturn+1):
                min = -1
                for i in range(0, len(currencyValuesArray)):
                        if currencyValuesArray[i] <= j:
                                if min == -1:
```

```
                    min = 1 + minArray[j - currencyValuesArray[i]]
                    coin = i
                elif 1 + minArray[j - currencyValuesArray[i]] < min:
                    min = 1 + minArray[j - currencyValuesArray[i]]
                    coin = i
        minArray.append(min)
        firstCoinArray.append(coin)

    coins = []
    while amountToReturn > 0:
        coins.append(currencyValuesArray[firstCoinArray[amountToReturn]])
        amountToReturn = amountToReturn -
                        currencyValuesArray[firstCoinArray[amountToReturn]]

    numberOfCoins = len(coins)

    result = []
    for coin in currencyValuesArray:
        result.append(coins.count(coin))

    result.append(numberOfCoins)

    return result
```

3) Prove Dynamic Programming: Induction

Base Case:

T[0] = 0.

Inductive Case:

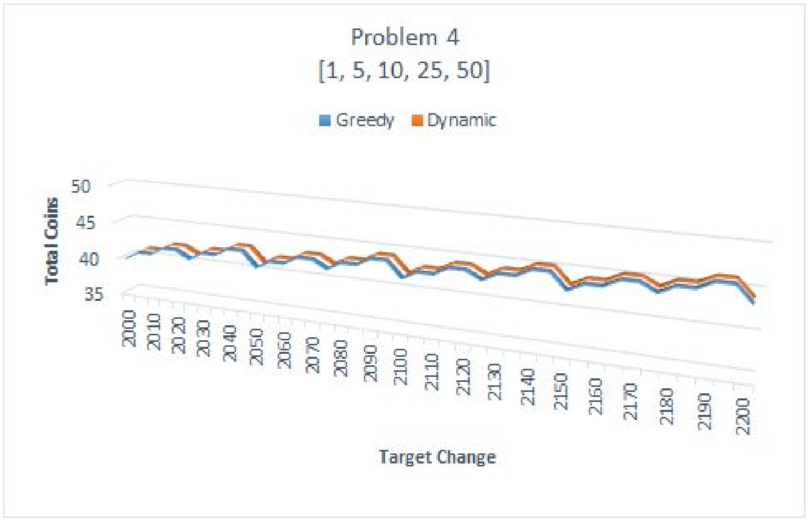
Assume the dynamic programming algorithm approach always gives the minimum number of coins for given amount of change and possible denominations.

In the best solution in making change of v cents, there exists some first coin V[i] where V[i] is less than v. The remaining coins in the optimal solution must be making change for v - V[i] cents. For example, if V[i] is the first coin in an optimal solution to v cents then T[v - V[i]] + 1 will optimally make change for v cents. It will check for all possibilities such that V[i] < v. The value of optimal solution must be the minimum value of T[v - V[i]] + 1.

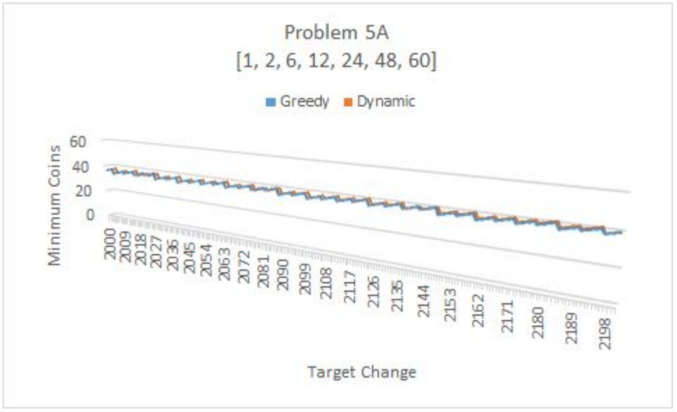
4) Data & Review of Varied Denominations and Inputs

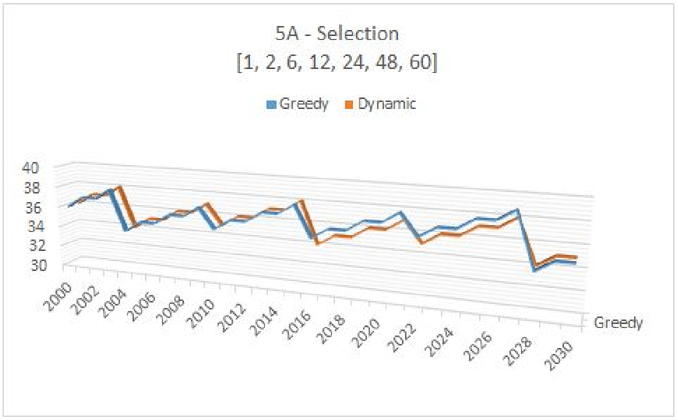
Problems 4-6:
In review the algorithms for Greedy and DP, when looking at V = [p0, p1, p2, … , pn]
Greedy will generally run faster for the statistical generic input. This is because, Greedy is geared to always look for the largest possible denomination first in array A, while DP will search for all values. If Greedy stumbles upon a large value early on, it will always do better in run time. However, faster does not always equal better, as DP has shown to be more accurate in finding the minimum possible amount of coins to make change.

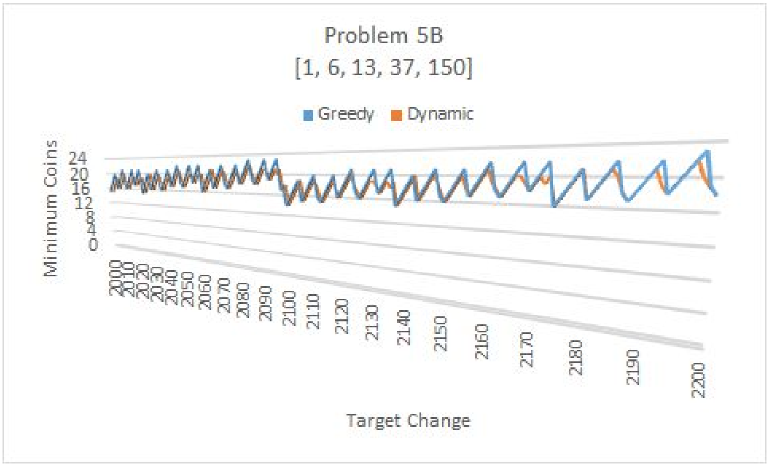
Coin values: [1, 5, 10, 25, 50], Desired change: [2000, 2005, … , 2200]
For the greedy and dynamic programming approach, the number of coins was exactly the same. This is partially due to the incremental values jumping by 5. The charts below illustrate the resulting minimum number of coins for Greedy and DP.

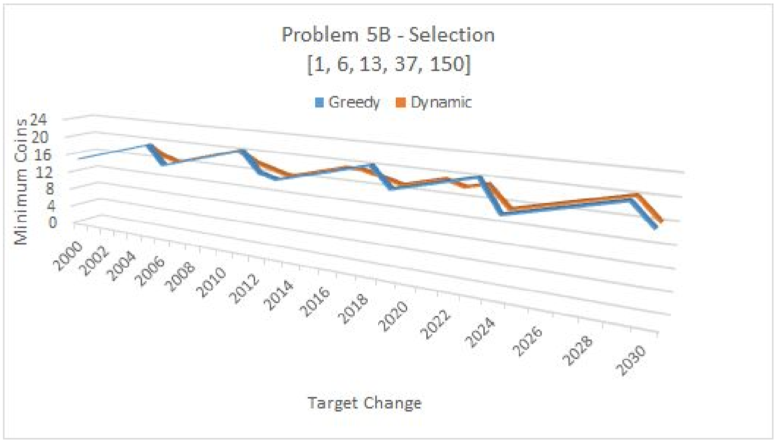


Part A - Coin values: [1, 2, 6, 12, 24, 48, 60], Desired change: [2000, 2001, … , 2200]
Part B - Coin values: [1, 6, 13, 37, 150], Desired change: [2000, 2001, … , 2200]
For even numbered denominations and singularly incremented targets, we start to see a change in the data. Due to the size of the data, we have presented a “targeted” chart for values 2000-2030 that allows scrutiny in greater detail. From this chart we can see that the greedy algorithm will choose the biggest option immediately, but end up with more coins used. For part B, you can see that the use of prime denominations has hindered the greedy algorithm considerably (again a “targeted” chart is presented for improved viewing). The dynamic programming approach may be outperformed by the greedy approach, but will always return the lowest minimum number of coins

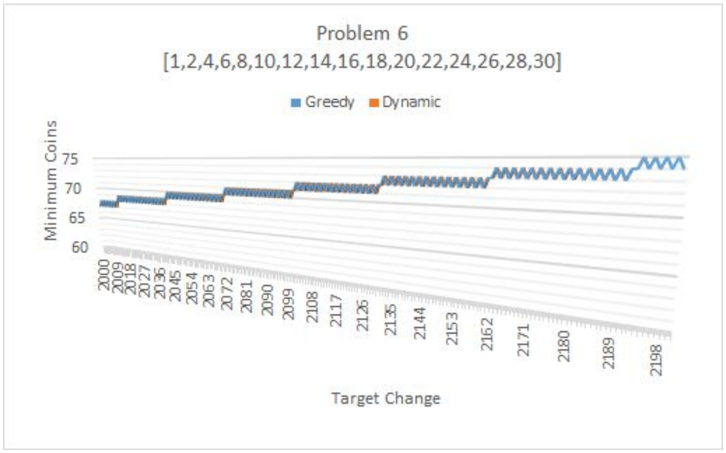

Problem 5A
[1, 2, 6, 12, 24, 48, 60]
Greedy
Dynamic
Minimum Coins
60
40
20
0
2000
2009
2018
2027
2036
2045
2054
2063
2072
2081
2090
2099
2108
2117
2126
2135
2144
2153
2162
2171
2180
2189
2198
Target Change

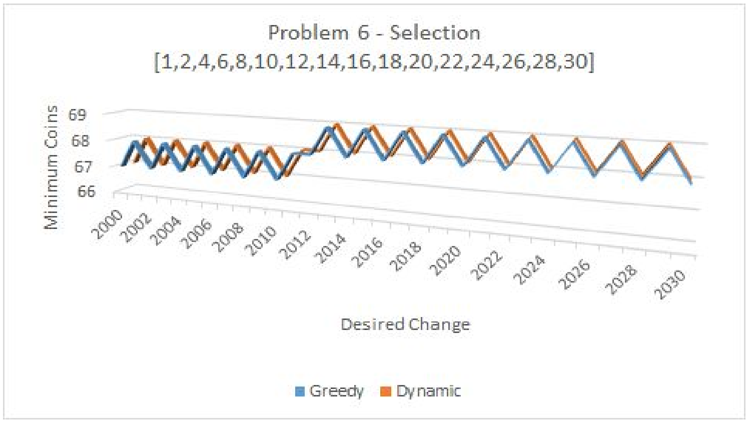

5A - Selection
[1, 2, 6, 12, 24, 48, 60]
Greedy
Dynamic
40
38
36
34
32
30
2000
2002
2004
2006
2008
2010
2012
2014
2016
2018
2020
2022
2024
2026
2028
2030
Greedy


Problem 5B
[1, 6, 13, 37, 150]
Greedy
Dynamic
Minimum Coins
24
20
16
12
8
4
0
2000
2010
2020
2030
2040
2050
2060
2070
2080
2090
2100
2110
2120
2130
2140
2150
2160
2170
2180
2190
2200
Target Change



Coin values: [1,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30], Desired change: [2000, 2001, … , 2200]
Due to the full set of even numbers from 2-30, the greedy algorithm matches the DP approach this time, as there is no efficiency penalty for selecting the largest coins first, with the 2 and 1 coin values available.

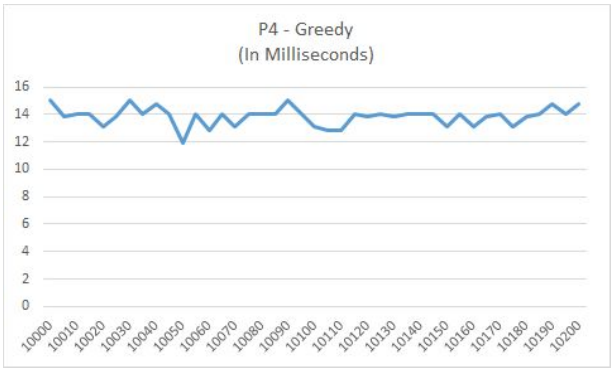


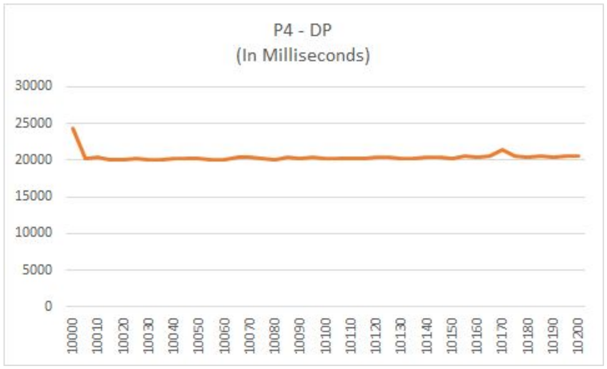


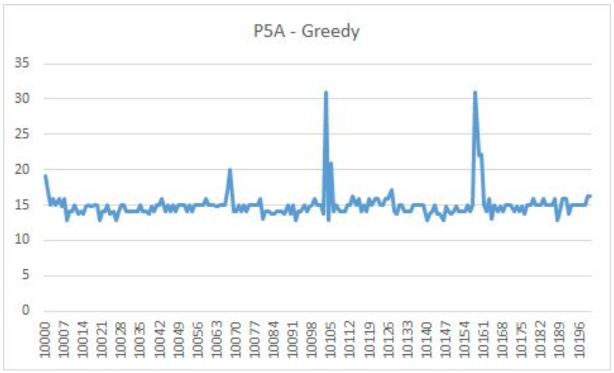
5) Experimental Review of Data

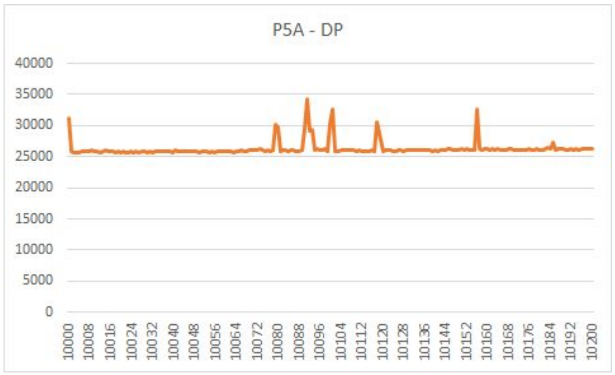
Problem 7:

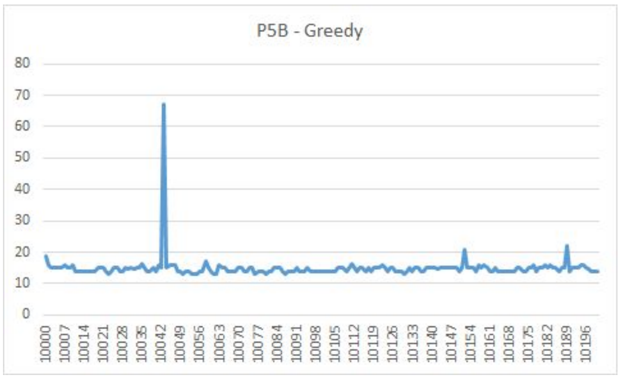
The run times of the Greedy algorithm, are significantly quicker than the DP approach, as shown by the charts below. With little variation (aside from anomalous spikes), the Greedy approach was not affected by the size of the denomination array, or the increment of them. This evidence is a strong indicator that the Greedy method should be considered viable on even extraordinarily large datasets, if the absolute minimum in 100% of those cases is not required. In order to record large enough run times, ranges were adjusted to start at a target value of 10000, and increase incrementally as dictated by the referenced problem numbers.

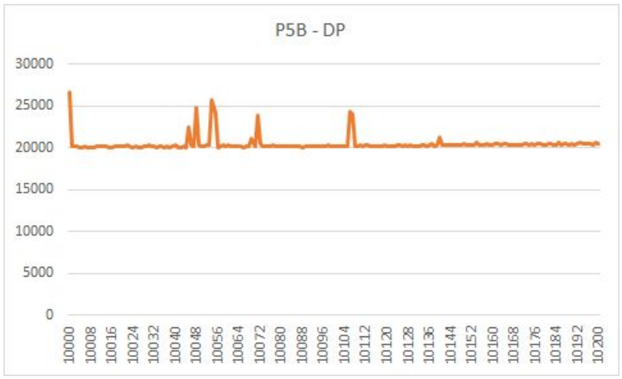


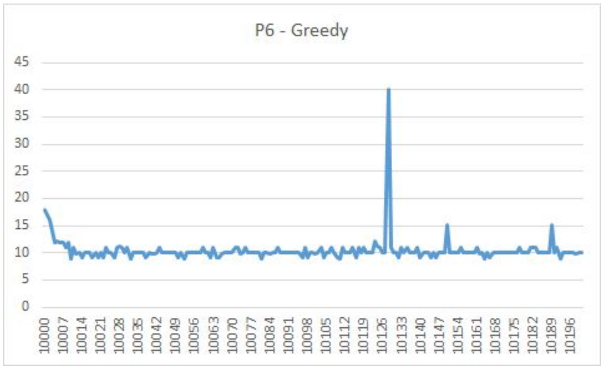


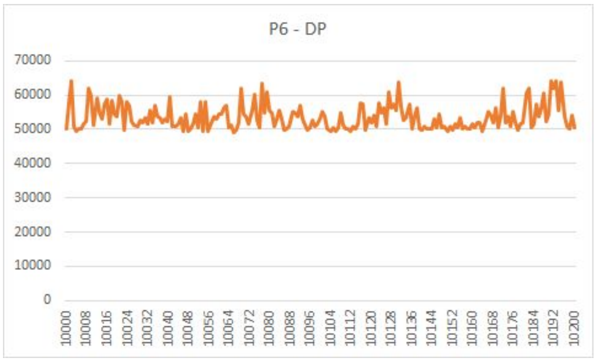








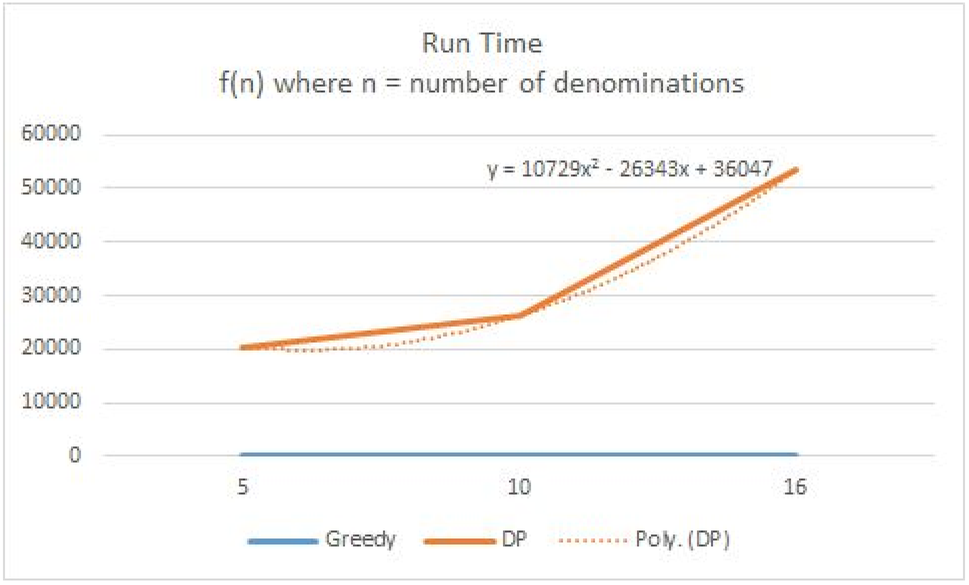




6) Additional Experimental Review of Data

Problem 8:

For analysis of the effect of the number of denominations (n), we can show below that the Greedy method runs at an almost constant level, as additionally shown in the section above. The Dynamic Programming approach on the other hand is clearly affected by n. A polynomial trendline for the Dynamic Programming run time with the slope of the equation is displayed below as well.



7) Experimental Review of Powered Coin Values Data

Problem 9:

Due to the large increases in value, would result in higher minimums. Based on earlier experimentation, we can see that a greater number of options will sometimes allow the greedy algorithm to find the same results as a dynamic approach. This is most effective in problem 6 above, where the greedy algorithm can “save itself” with a relatively small addition of coins. With exponential denominations, there is less chance to find a low minimum, as the growth rate is significantly higher.