Project 1 Coins Report

Group 19
John Fitzpatrick, Conner Pacala
July 19, 2016

Psudocode:

```
changegreedy(coins, value):
   while value > 0 and index >= 0:
      If coins[index] <= value:
         coinArray[index] += 1
         value -= coins[index]
         numCoins += 1
      else:
         Index -= 1

   return coinList, numCoins


changedp(coins, value):
   minCoins = [LARGEVAL] * (value + 1)
   minCoins[0] = 0
   route =

   for i = 1, i < value + 1:
      for j = 0, j < len(coins):
         if coins[j] <= i:
            prev = minCoins[i - coinVal[j]]
            If prev + 1 < minCoins[i]:
               minCoins[i] = prevNum + 1

   Return minCoins[value]
```

Asymptotic Running Time:

changegreedy has an asymptomatic running time of O(n). There is a single loop whose runtime depends on the total value you are trying to find coins for.

changedp has an asymptomatic running time of $O(c^2v)$ where c is the total number of coins of different denominations and v is the total value you are trying to find coins for. There are three nested for loops, the first one running ‘value’ number of times so it is O(v) and the second and

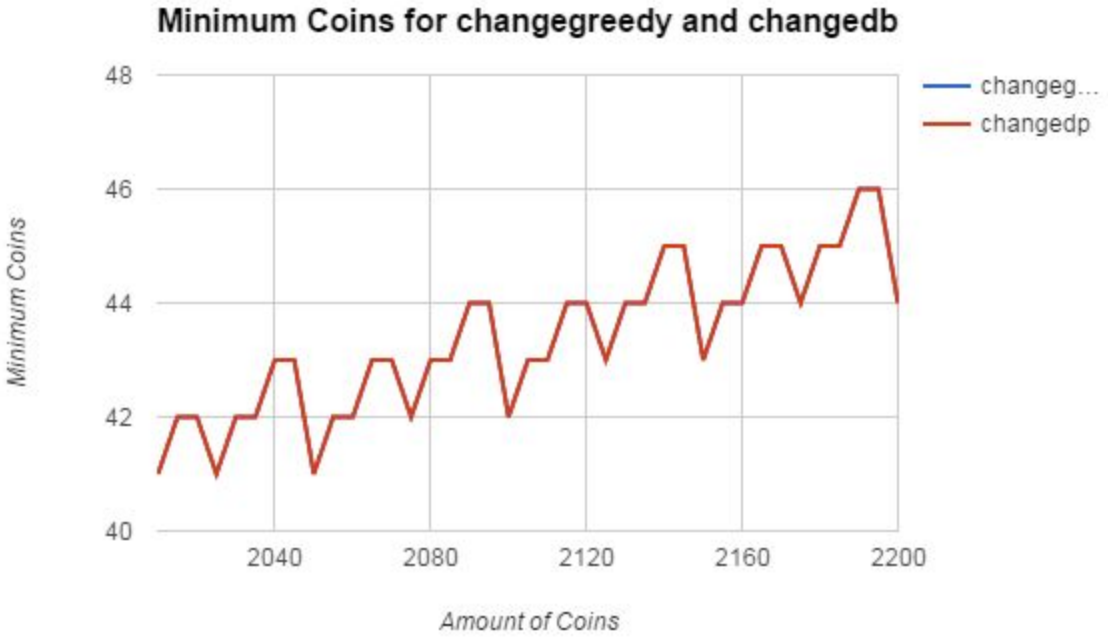
third run number of coins times for every value so it is O(c). Since they are nested it's multiplicative yielding a total time of $O(c^2v)$. Luckily, the number of coins in the array in practical applications are likely to be much smaller than the total value, so the c^2 will have less of an effect than the v in the run time.

Filling the changedp table:

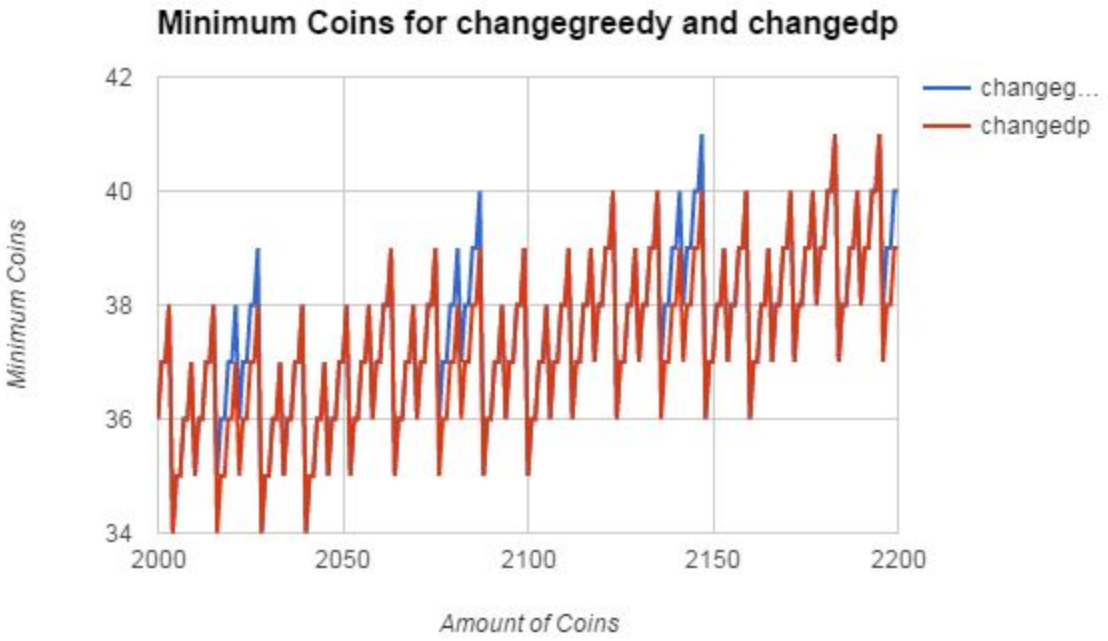
Each spot in the table represents the number of coins needed to get 'index' amount of currency (e.g. index 1 is the number of coins needed to get 1 coin). The algorithm builds the table from 0 to 'value' number of coins, finding the minimum number of coins needed for each value and storing it in the table. It does this by setting the base case (0 total value) to 0 for each coin and all other values in the table to a large value. It then checks each possible value from 1 to the passed amount. The value at table[current] is compared to the minimum number of coins needed to complete table[current - current coin value] + 1. If this result is less than the current value stored in table, that value is stored in the array.

This works because it finds the minimum number of coins at each step. It will find the closest value (e.g. with a 1 cent coin it will check the previous table entry and with a 5 cent coin it will check the entry 5 indices previous if applicable) and only ever add a single coin to that entry. So it will check every single coin and find the minimum possible of all coins.

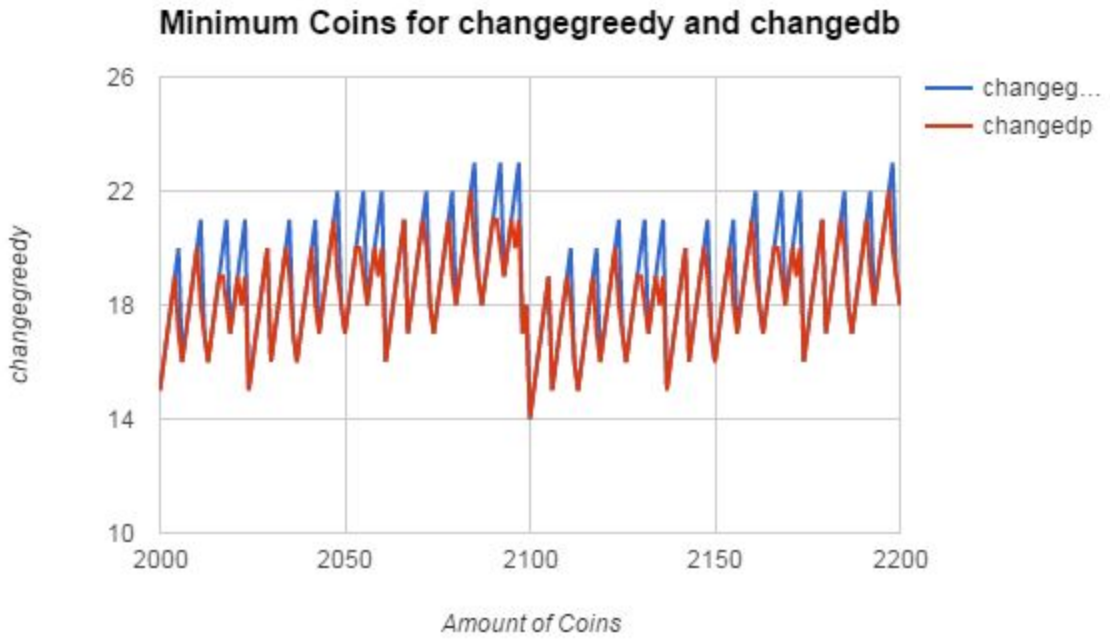
There is also a second table that stores the number of coins of each type that are needed to complete the current value. This table is updated every time the minimum coin table is updated and uses the same principle of



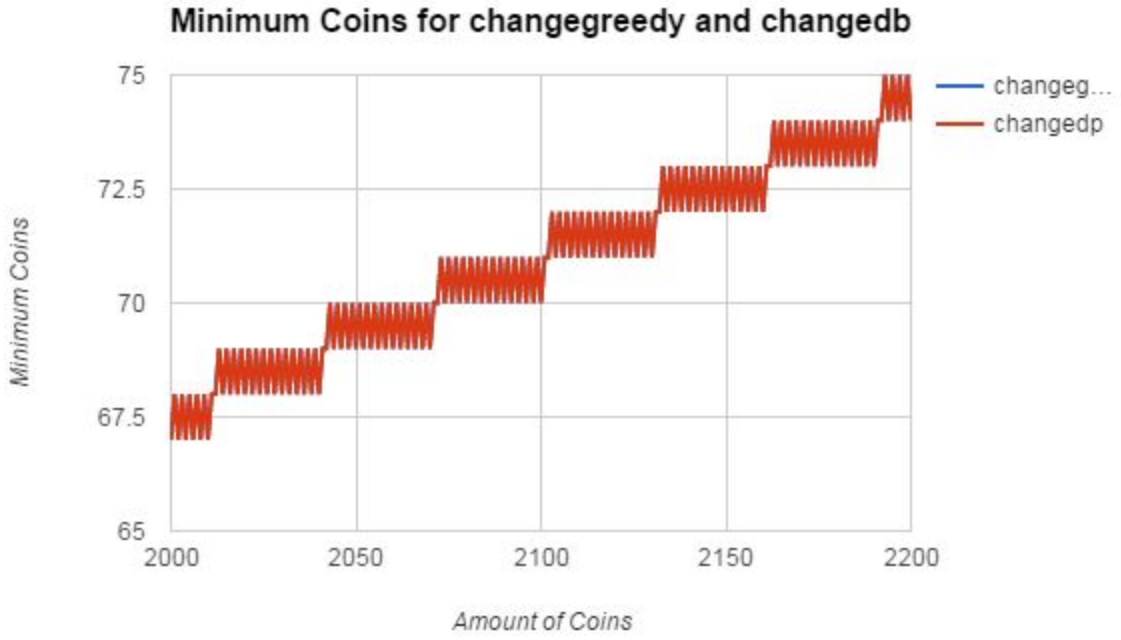
Changegreedy and changedp function's minimum coins as a function of amount of coins for set V = [1, 5, 10, 25, 50] and A [2010, 2015, … 2200]. As you can see, the two algorithms produce the same results for this amount range and these coin denominations.



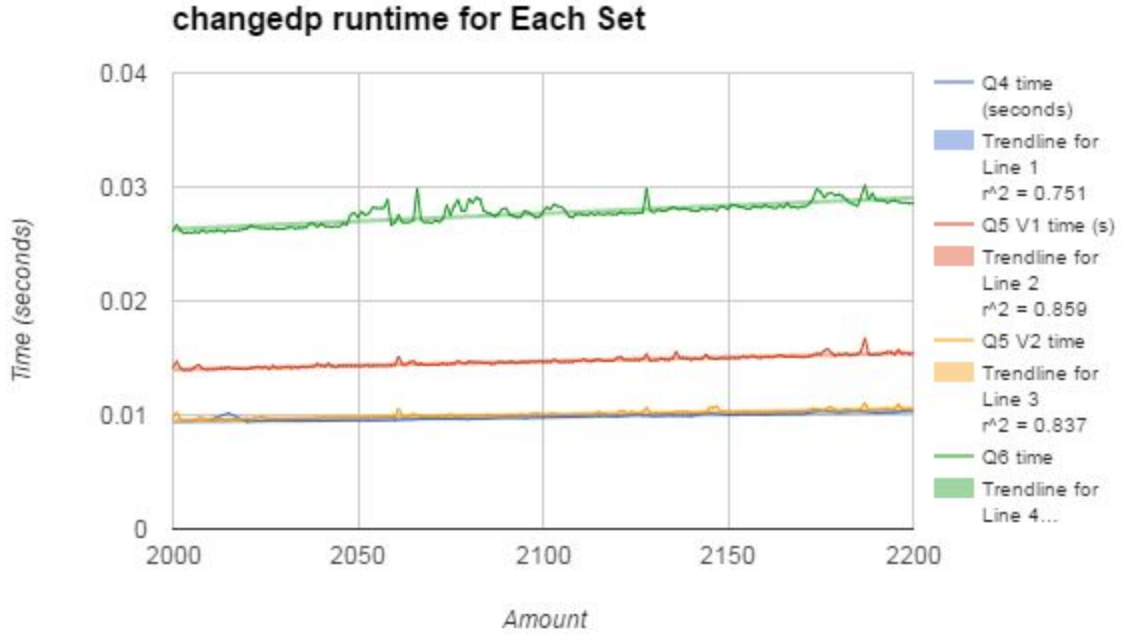
Changegreedy and changedp functions minimum coins as a function of amount of coins for set V = [1, 2, 6, 12, 24, 48, 60] and A from 2000 to 2200. AS you can see, the changegreedy function differs from the changedp function in a few areas (e.g. when A = 2141, changegreedy returns 40 coins and changedp returns 39 coins).

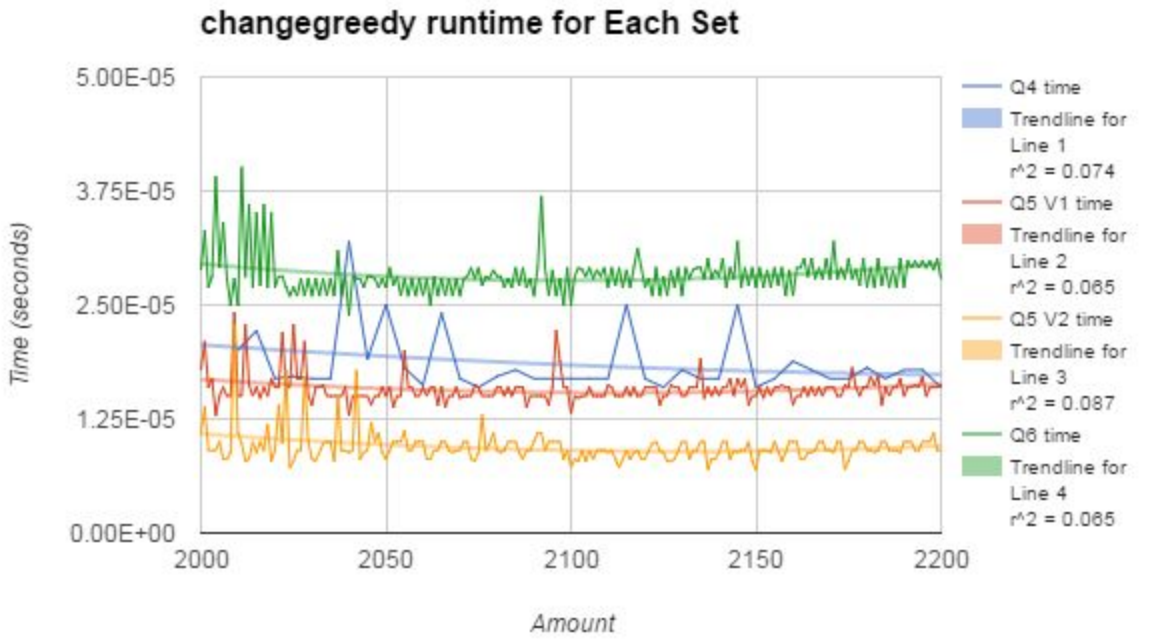


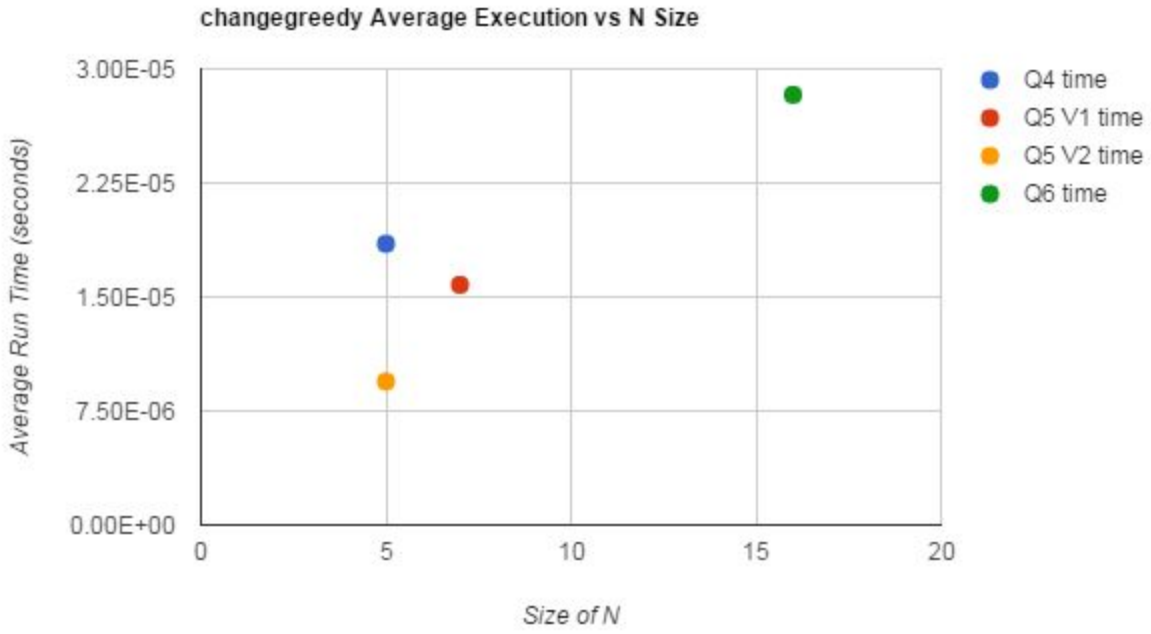
Changegreedy and changedp functions minimum coins as a function of amount of coins for set V = [1, 6, 13, 37, 150] and A from 2000 to 2200. As you can see, the changegreedy function differs from the changedp function in most areas and the difference is greater than 1 in many cases.

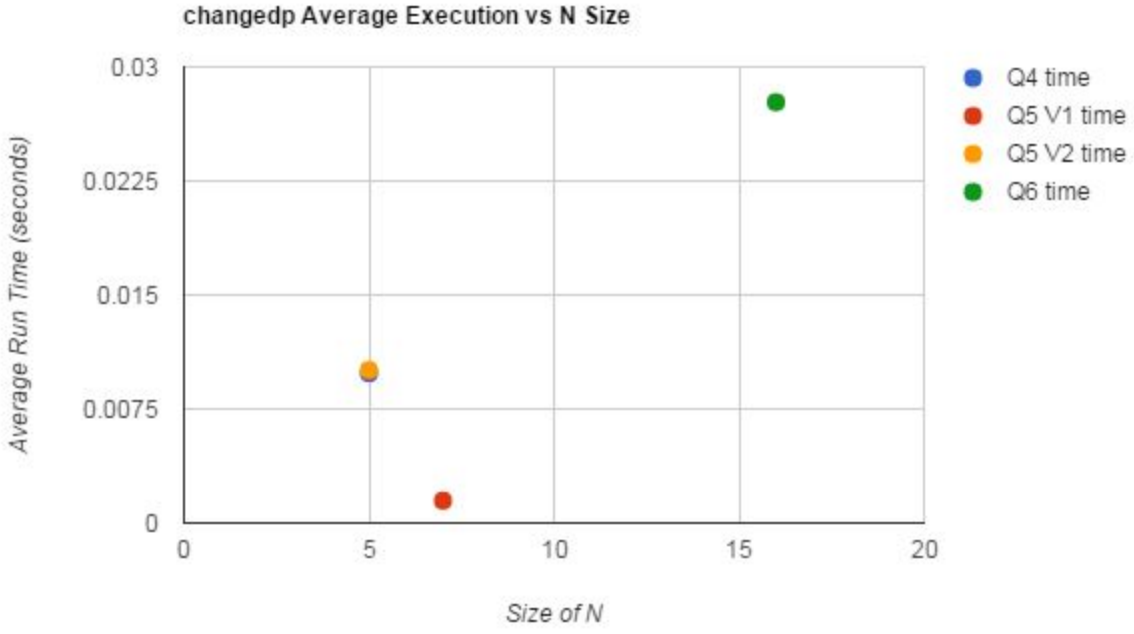


Changegreedy and changedp functions minimum coins as a function of amount of coins for set V = [1, 2, 4, 6, 8, 10, …, 30] and A from 2000 to 2200. As you can see, the changegreedy function and changedp functions follow the same trend.


changedp runtime for Each Set
0.04
0.03
0.02
0.01
0
Time (seconds)
2000
2050
2100
2150
2200
Amount
Q4 time (seconds)
Trendline for Line 1 r^2 = 0.751
Q5 V1 time (s)
Trendline for Line 2 r^2 = 0.859
Q5 V2 time
Trendline for Line 3 r^2 = 0.837
Q6 time
Trendline for Line 4...


changegreedy runtime for Each Set
5.00E-05
3.75E-05
2.50E-05
1.25E-05
0.00E+00
Time (seconds)
2000
2050
2100
2150
2200
Amount
Q4 time
Trendline for Line 1 r^2 = 0.074
Q5 V1 time
Trendline for Line 2 r^2 = 0.065
Q5 V2 time
Trendline for Line 3 r^2 = 0.087
Q6 time
Trendline for Line 4 r^2 = 0.065


changegreedy Average Execution vs N Size
3.00E-05
2.25E-05
1.50E-05
7.50E-06
0.00E+00
Average Run Time (seconds)
0
5
10
15
20
Size of N
Q4 time
Q5 V1 time
Q5 V2 time
Q6 time


changedp Average Execution vs N Size
0.03
0.0225
0.015
0.0075
0
Average Run Time (seconds)
0
5
10
15
20
Size of N
Q4 time
Q5 V1 time
Q5 V2 time
Q6 time

Suppose you are living in a country where coins have values that are powers of p, V = [1, 3, 9, 27]. How do you think the dynamic programming and greedy approaches would compare? Explain.

In such a situation the greedy algorithm would likely always outperform the dynamic programming approach. This is because the greedy algorithm is simple and uses the maximum amount of each largest denomination coin that is still less than the remaining total. It therefore will almost always be faster than O(n), but in this particular instance it will also always be correct. There is more detail in the answer for question 10, but essentially anything that uses multiples of a number like that will always be most efficient and accurate with a greedy algorithm. Additionally, the dynamic programming approach is bounded by A, the amount of change to make, where the greedy approach is bounded by the number of coins, so as long as you need change for more than 4 cents it will be faster.

Under what conditions does the greedy algorithm produce an optimal solution? Explain.

The greedy algorithm will produce optimal solutions when all values V[2] to V[i] are multiples of a common base number. For example, the US currency uses multiples of 0.05 (0.05·1 = 0.05, 0.05·2 = 0.10, 0.05·5 = 0.25, 0.05·10 = 0.50, 0.05·20 = 1.00), so the best solution is always going to be the highest multiple of five that you can choose, plus the pennies to handle any remaining value less than 5. So any value can be represented as (0.05 ·x + y ·pennies). Since you will always have to choose y pennies regardless of the value of x , the solution depends entirely on the values of x that are chosen, and the total adds up the fastest when you use the largest values of x possible.