Project Two - Coin Change

CS 325 - Analysis of Algorithms

Group Two

Evan Schallerer
Eli Rademacher
Krisna Irawan

Dynamic Programming Table Explanation

For this problem, we used the same algorithm as the "naïve" implementation which was recursive and incredibly slow, except that we added in a table to keep track of the minimum number of coins needed for each step of the process. The slower naïve implementation works by checking every combination of two values that sum to the amount we are trying to make change for, and finding the least number of coins necessary. It does this recursively and iteratively at the same time, so the same value is calculated multiple times, which is why it's so slow.

If we are trying to make a change amount *x* with coin denominations *y*, then the table is of length *x+1*, where each element in the array contains a list of length of *y*, looking something like this:

(*x*, *y*)	**0**	**1**	**2**	**3**	**4**	**5**
1	0	1	0	1	0	1
2	0	0	1	1	0	0
4	0	0	0	0	1	1
8	0	0	0	0	0	0

Figure 1. The values in the left-most column represent the denominations of change available, while the values in the upper-most column represent the values up to the amount of change we are trying to make. All of the other values represent the number of coins of the corresponding denomination that is needed to make up the value when summed with those above and below it.

As we iterate and recurse through the values, we will calculate the minimum number of coins for 0, 1, 2, to *n* coins, and the table will store the minimum number needed in the table as it goes through. If a value in the table already exists when it is needed, the algorithm uses that value rather than calculating it again. If the value amount is equal to an amount in the denominations of coins allotted, then it only uses one coin to make the appropriate change.

Pseudocode

Algorithm 1 - Brute Force or Divide and Conquer Algorithm

```
changeSlow (v, c, a)
{
     for (i = 0; i < v.length; i = i + 1) {
          if (v[i] equals a)
          {
               c[i] = c[i] + 1
               return c
          }
     }
     i = 0
     minCoins = {}
     while ( i < a ){
          //find min to make i cents
          coins1 = changeSlow(v, c, i)

          //find min to make a - i cents
          coins2 = changeSlow(v, c, a - i)

          if ( coins1.sum < minCoins.sum )
               minCoins = coins1
          if ( coins2.sum < minCoins.sum )
               minCoins = coins2
          i = i + 1
     }
     return minCoins
}
```

Algorithm 2 - Greedy Algorithm

```
changegreedy( v, a )
{
     change = [0] * len(v)
     for ( e in reversed(v) )
     {
          while( (sum(change) + e) < v )
          {
               change[v.index(e)] = change[v.index(e)] + e
          }
     }
     return change
}
```

Algorithm 3 - Dynamic Programming

```
def changedp(value, denominations, memoizedVals):
     coins = [0] * len(denominations)  #initialize to all zeros

     #base case
     if value in denominations:
          coins[coin index] += 1
               memoizedVals[value] = coins
               return coins

          #check all combinations of value
          for i in range(1, value):
               coins1 = 0
               coins2 = 0

               if (memoizedVals[i] == False):
                    coins1 = changedp(i, denominations, memoizedVals)
               else:
                    coins1 = memoizedVals[i]

               if (memoizedVals[value - i] == False):
                    coins2 = changedp(value - i, denominations, memoizedVals)
               else:
                    coins2 = memoizedVals[value - i]


               #store minimum number of coins (force first time)
               if ((sum(coins1) + sum(coins2)) < sum(coins)) or (i == 1):
                    coins = [sum(x) for x in zip(coins1, coins2)]
                    memoizedVals[value] = coins

          return coins


#helper function
def wrapDP(value, denominations):
     #initialize memoized values to false
     memoizedVals = [False] * (value+1)
     return changedp(value, [1, 3, 7, 12], memoizedVals)
```

Proof of Correctness
Making Change
Dynamic Programming

Pre-Conditions:

- Given a value to make change for
- Given a list of denominations of coins available to make change with
- Infinite number of each denomination available

Post-Conditions

- A list of the number of each denomination of coin necessary to make the amount of change given is returned
- The total number of coins required to make the change is returned

Base case: We can always make change for a = 0 by simply returning 0 coins.

Induction Hypothesis: Assume that $f(k)$ is true for some arbitrary positive integer *k*, that is, assume our algorithm returns the minimum number of coins that can be used to make change for some amount *k*.

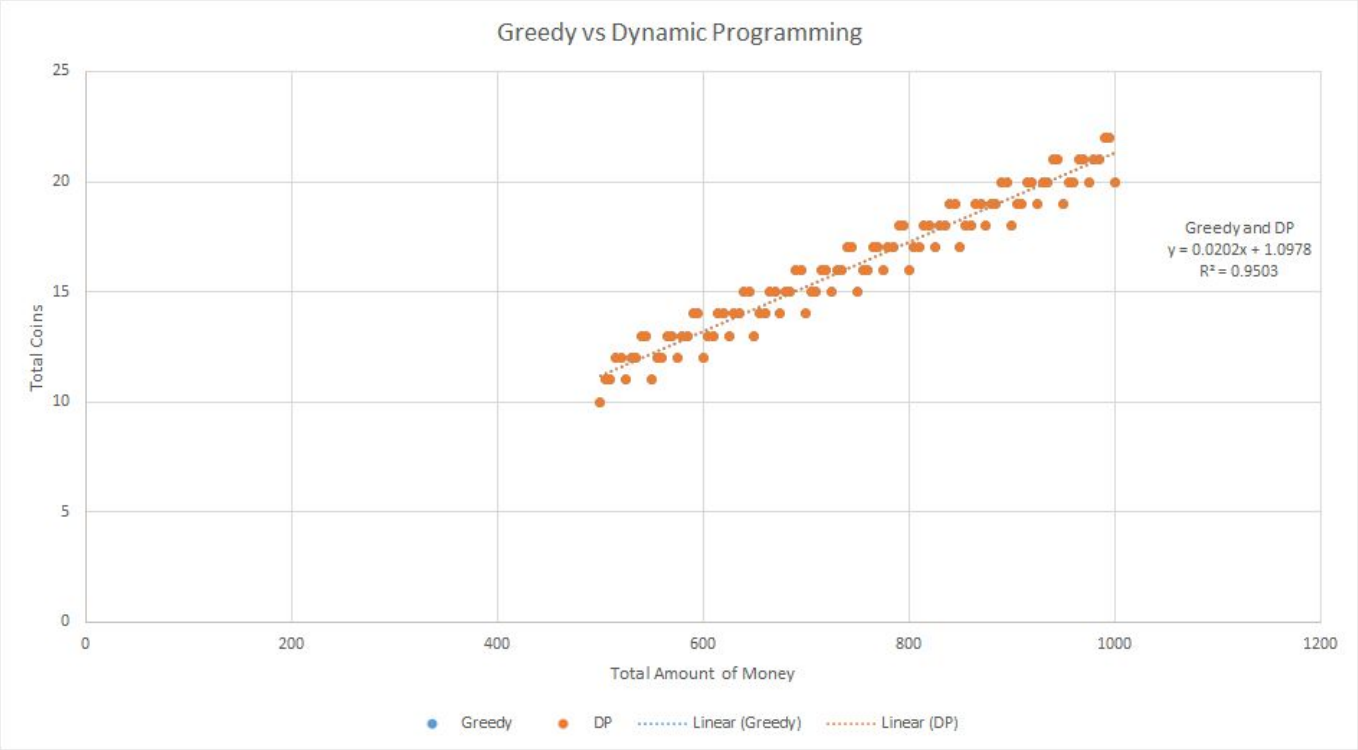
Inductive Step: Given the inductive hypothesis is true, we want to show that $f(k + d_i)$ is also true, where d_i is the value of the i^{th} coin in our monetary system. Normally in an inductive proof, we would try to show that $f(k + 1)$ is true, but in this case, it might not be possible to give exact change for $f(k + 1)$ if we do not have a one-cent piece.

It is apparent that $f(k - d_i) \leq f(k) - 1$ for at least one denomination d_i, because if you can find the minimum number of coins needed to make change for k, then you know that the minimum number of coins needed to make change for $k - d_i$ will be at most one less than the number of coins to make change for k.

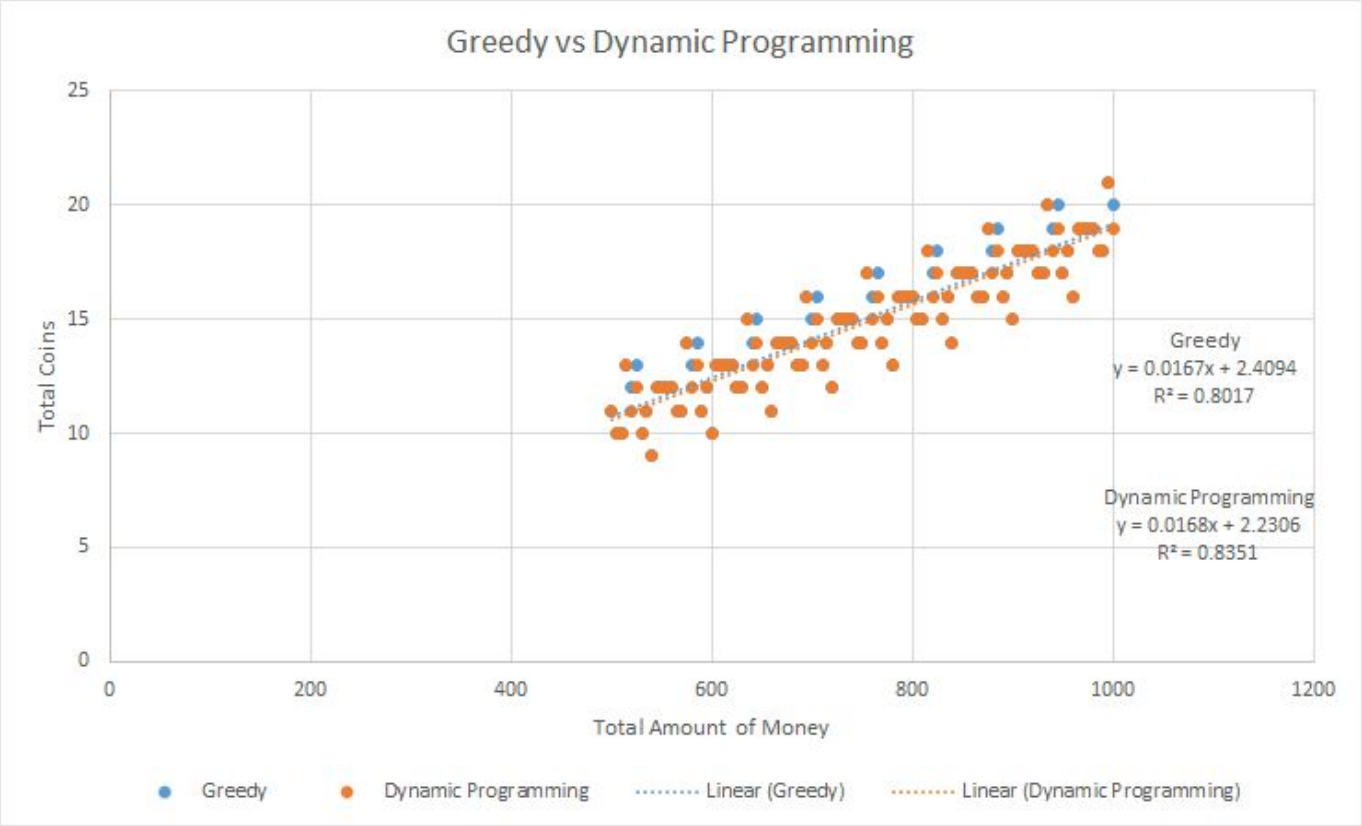
It is also apparent that $f(k) \leq f(k - d_i) + 1$, because the minimum number of coins to make change for $k - d_i$ units of money will be at least one less than the minimum number of coins to make change for k units of money.

Given these two inequalities, by the trichotomy axiom it must be the case that $f(k) = f(k - d_i) + 1$ for some denomination d_i. There is some d_i among the coins we are using that will yield a minimum number of total coins. So we can say $f(k) = min\{f(k - d_i) + 1\}$.
Thus $f(k + d_i) = min\{f(k + d_i - d_i) + 1\} = min\{f(k) + 1\}$. This equation shows that if we know the minimum number of coins that can be used to make change for k units of money, then we can find the minimum number of coins that can be used to make change for $k + 1$ units of money. In other words, if we know that $f(k)$ is true, then we know that $f(k + d_i)$ is also true.

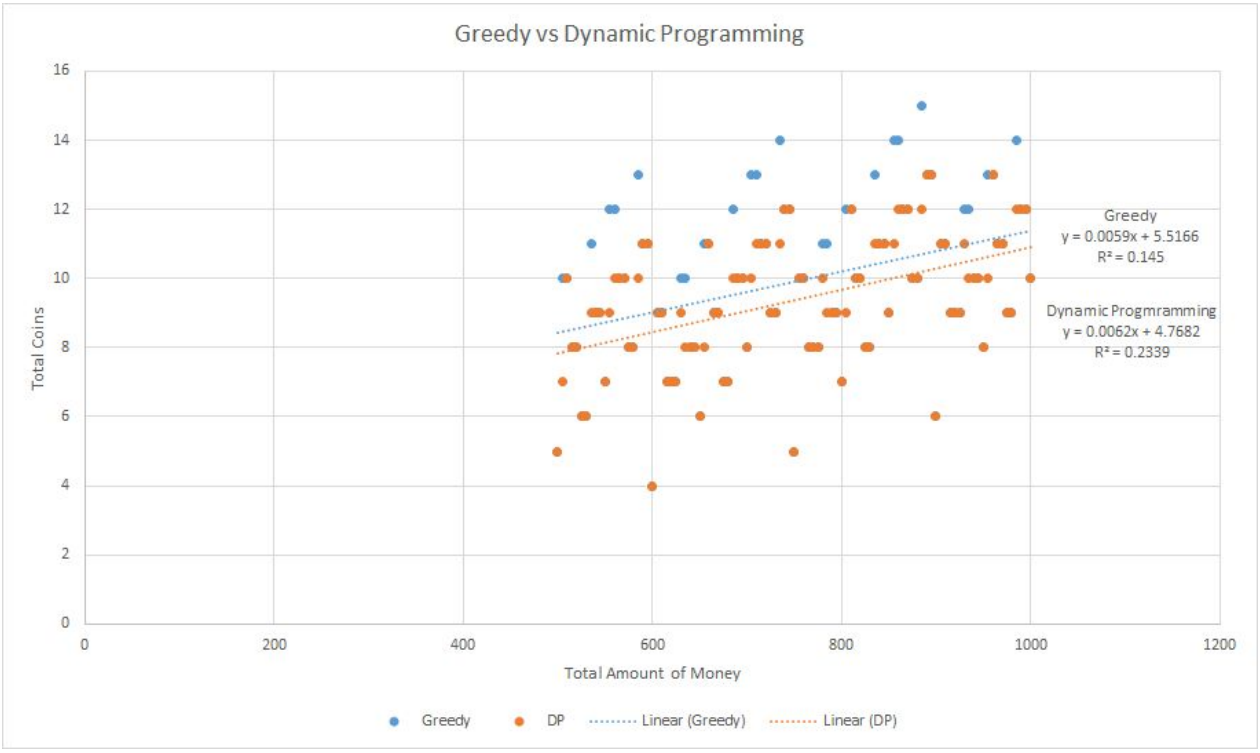
4) Both for Greedy and Dynamic Programming approach have the same total coins for each total amount of money that we are making change for. For both algorithm the number of coins they return is a linear function of the amount of money: c = 0.0202a + 1.0978. Based from the graph, we can see that both of the approaches have the same total coins. Thus, the greedy algorithm is optimal for this V = [1, 5, 10, 25, 50]..



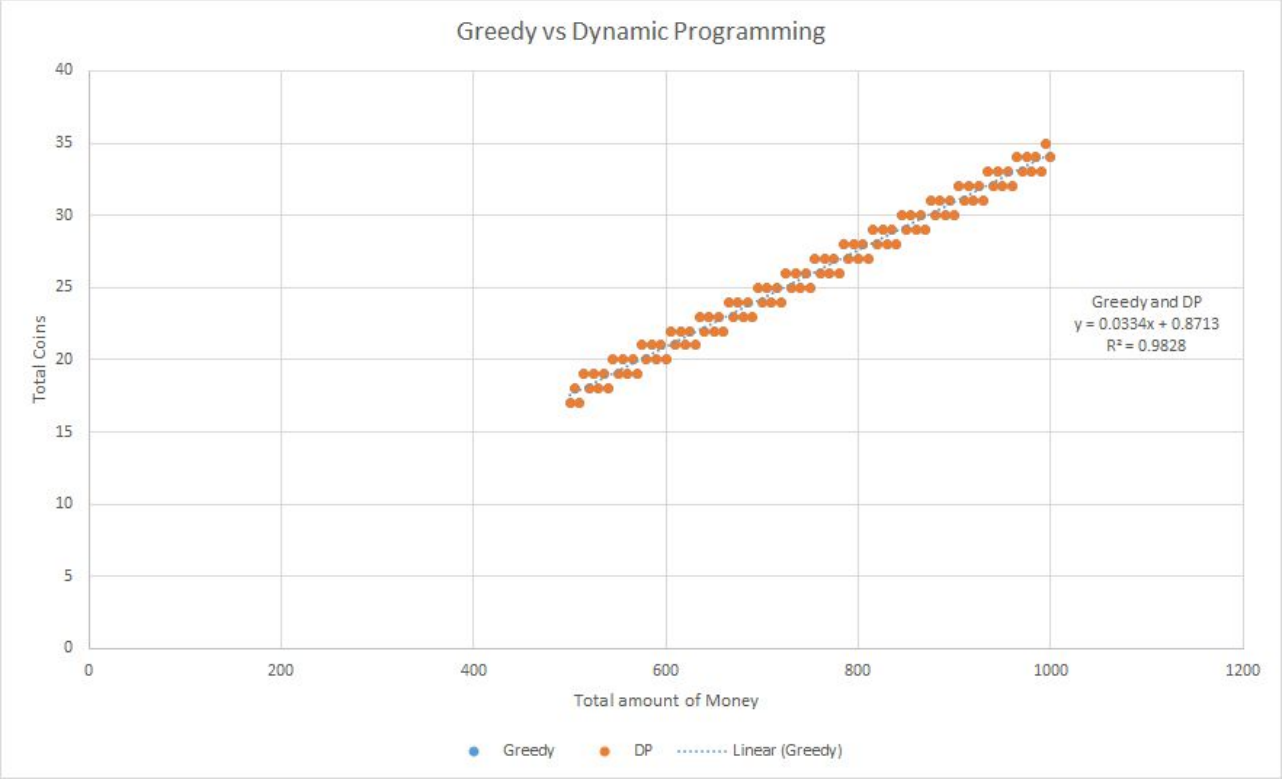
5 a) Because the data points don't overlap and the dynamic programming is always optimal, the greedy algorithm is not optimal for V1 = [1, 2, 6, 12, 24, 48, 60].



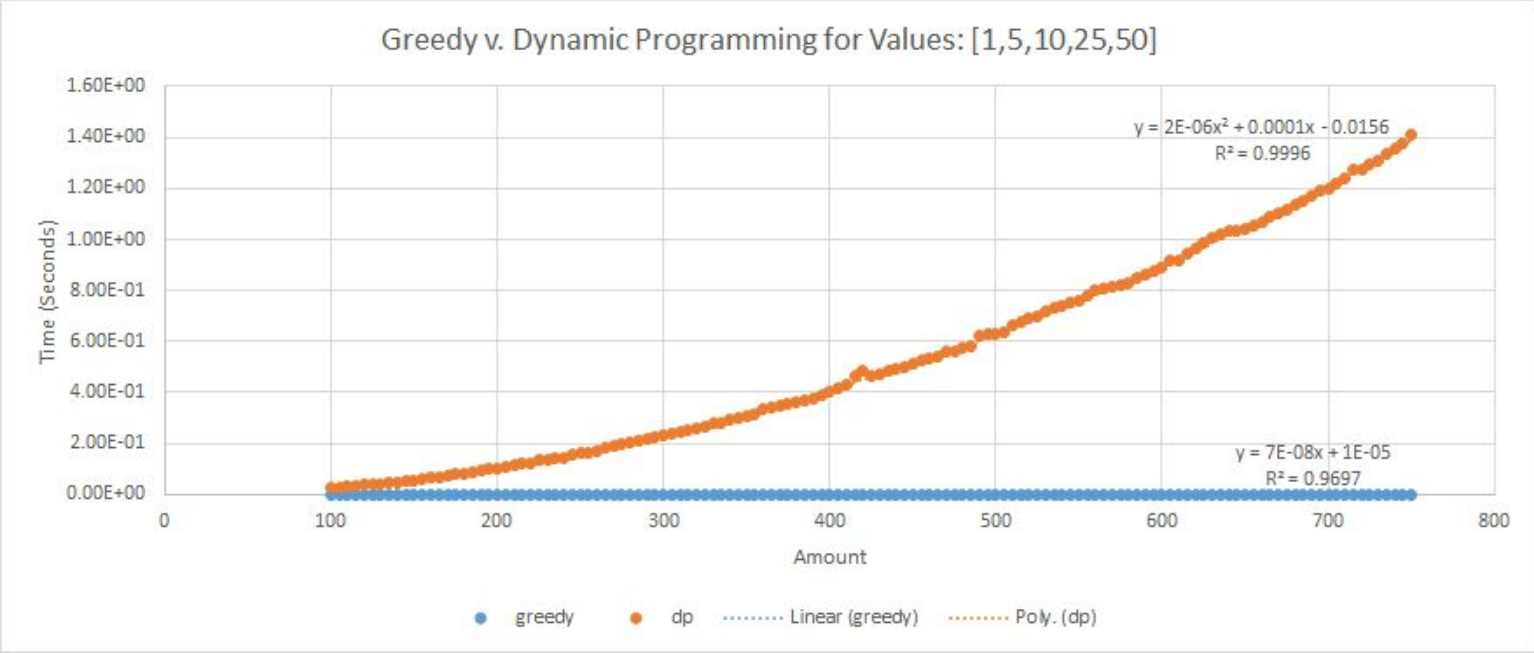
5 b) Because the data points don't overlap and the dynamic programming is always optimal, the greedy algorithm is not optimal for V2 = [1, 6, 13, 37, 150]

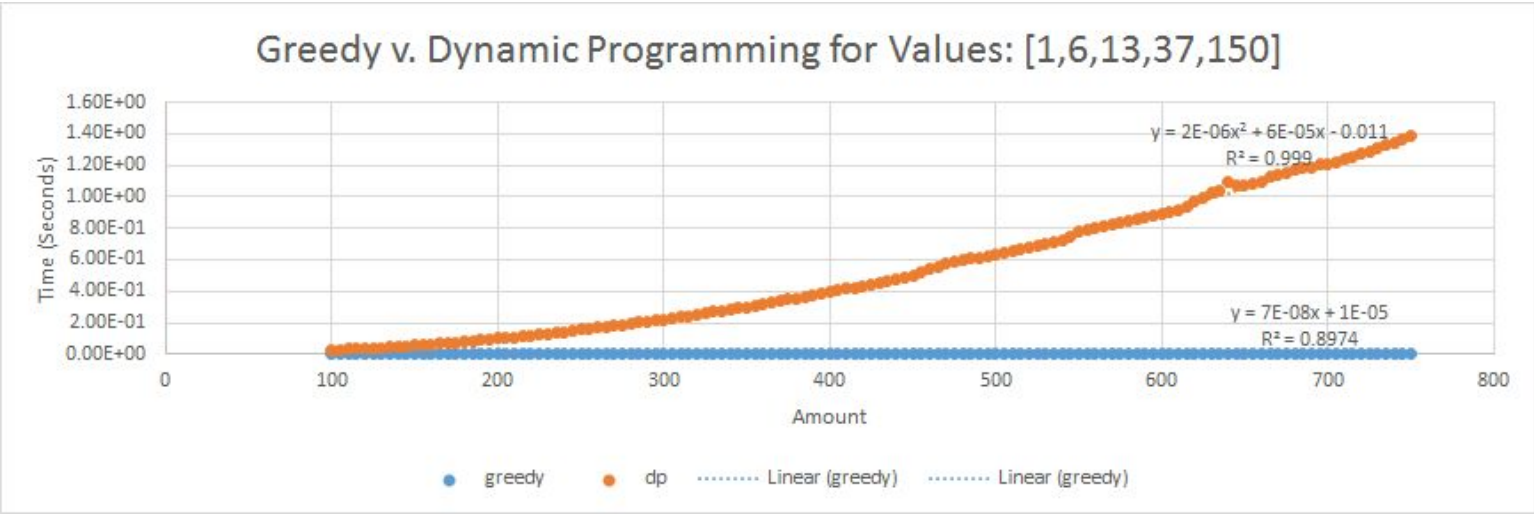


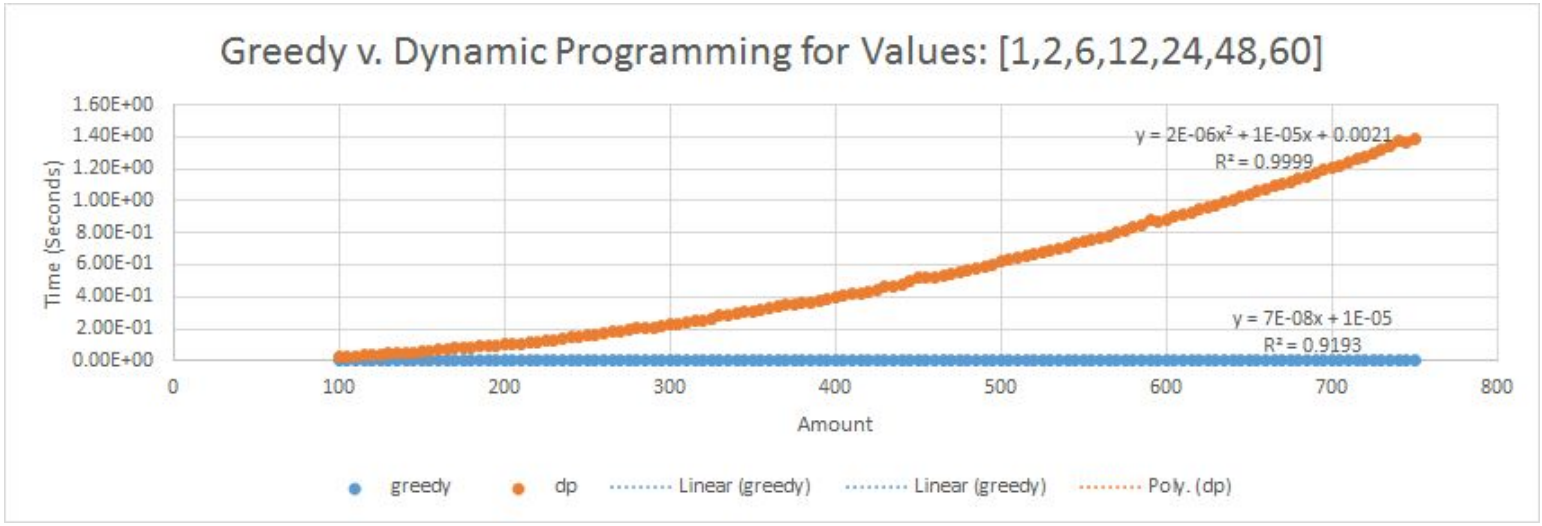
6) Both for Greedy and Dynamic Programming approach have the same total coins for each total amount of money that we are making change for. For both algorithm the number of coins they return is a linear function of the amount of money: c = 0.0334a + 0.8713. Based from the graph, we can see that both of the approaches have the same total coins. Thus, the greedy algorithm is optimal for V = [1, 2, 4, 6, 8, 10, 12, …, 30].

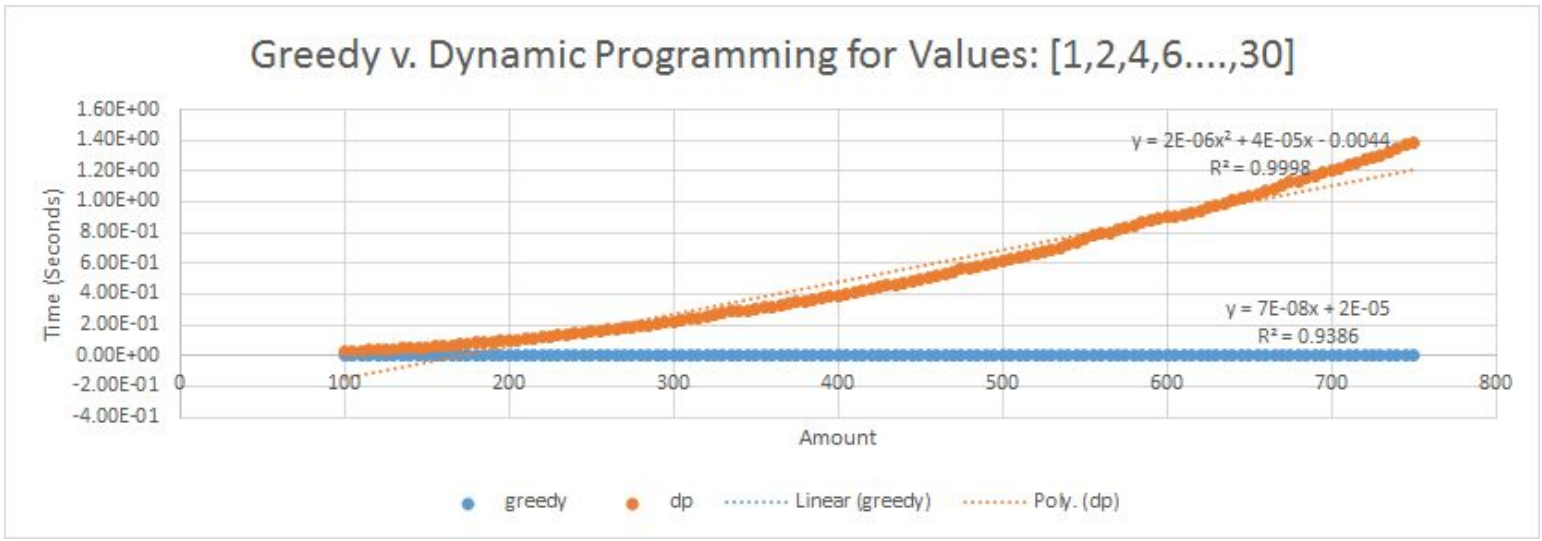


7) Below we show plots of the runtimes that we measured for the greedy and dynamic programming algorithms with various denomination schemes. We plotted the runtimes as a function of the amount for which we wish to make change. The graphs clearly show that the dynamic programming approach is asymptotically faster than the greedy algorithm. We do not show a plot of the runtime of the brute force algorithm, but it can be seen from the data presented in problem 8 that it takes much longer than the other two algorithms.

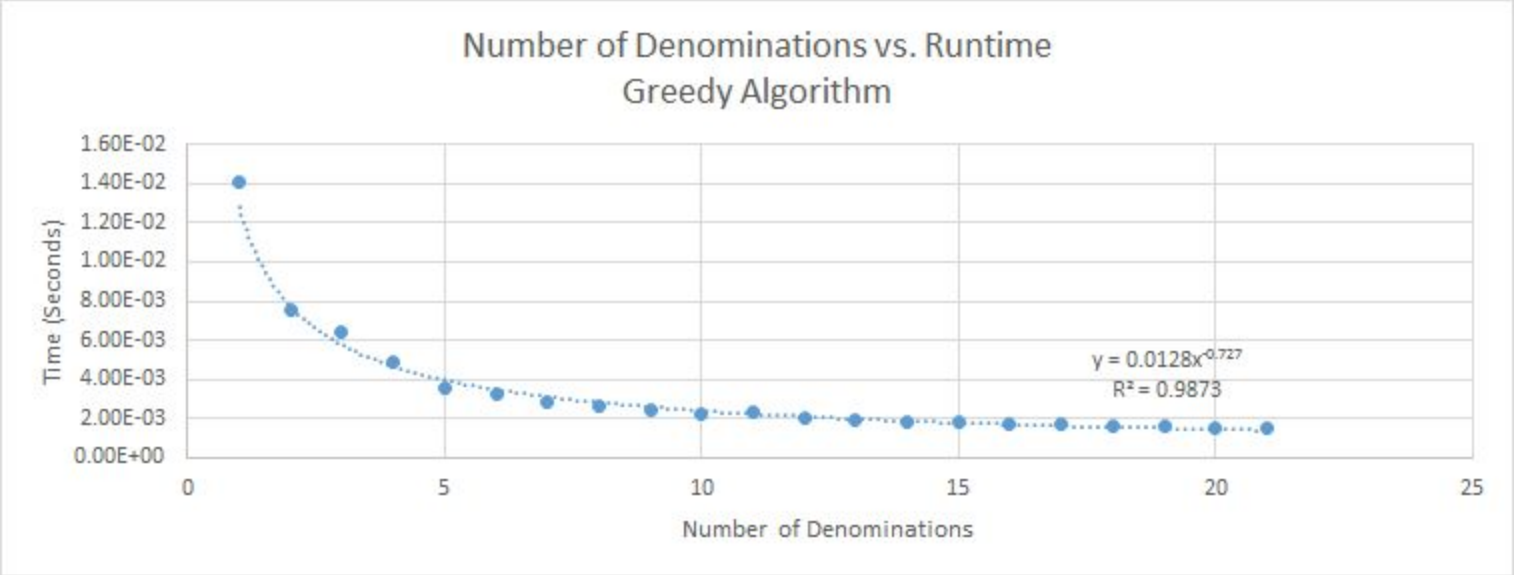




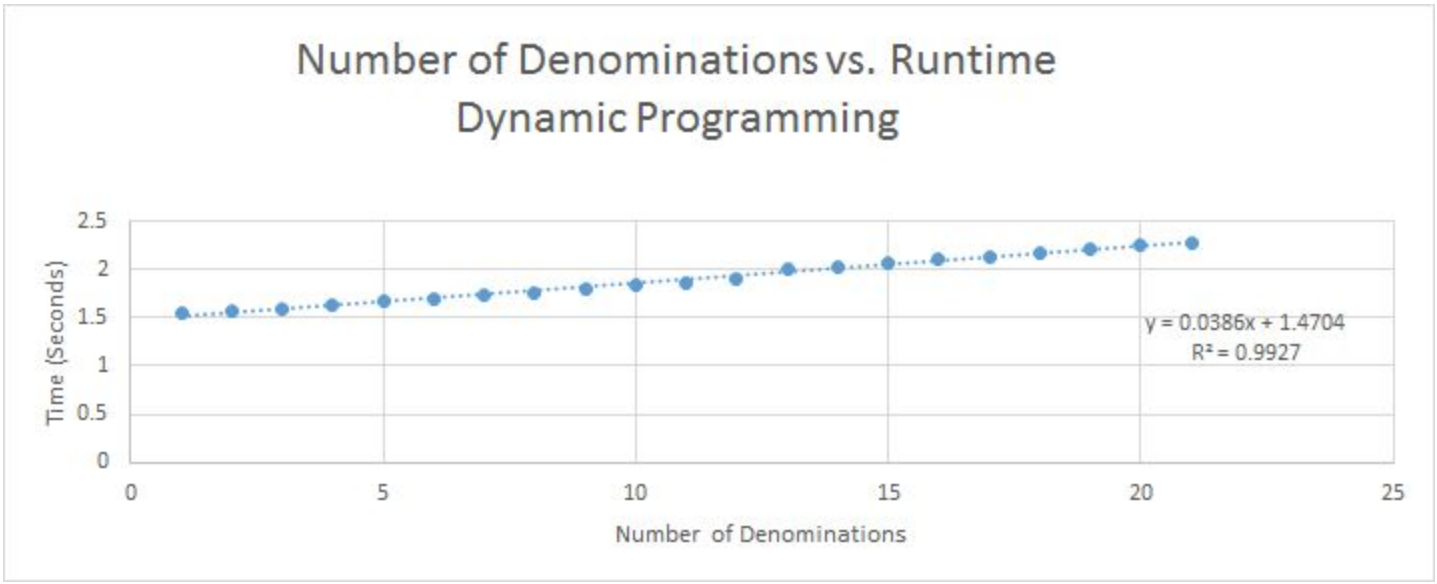

Greedy v. Dynamic Programming for Values: [1,2,6,12,24,48,60]
Time (Seconds)
1.60E+00
1.40E+00
1.20E+00
1.00E+00
8.00E-01
6.00E-01
4.00E-01
2.00E-01
0.00E+00
0
100
200
300
400
500
600
700
800
Amount
y = 2E-06x² + 1E-05x + 0.0021
R² = 0.9999
y = 7E-08x + 1E-05
R² = 0.9193
greedy
dp
Linear (greedy)
Linear (greedy)
Poly. (dp)


Greedy v. Dynamic Programming for Values: [1,2,4,6....,30]
Time (Seconds)
1.60E+00
1.40E+00
1.20E+00
1.00E+00
8.00E-01
6.00E-01
4.00E-01
2.00E-01
0.00E+00
-2.00E-01
-4.00E-01
0
100
200
300
400
500
600
700
800
Amount
y = 2E-06x² + 4E-05x - 0.0044
R² = 0.9998
y = 7E-08x + 2E-05
R² = 0.9386
greedy
dp
Linear (greedy)
Poly. (dp)

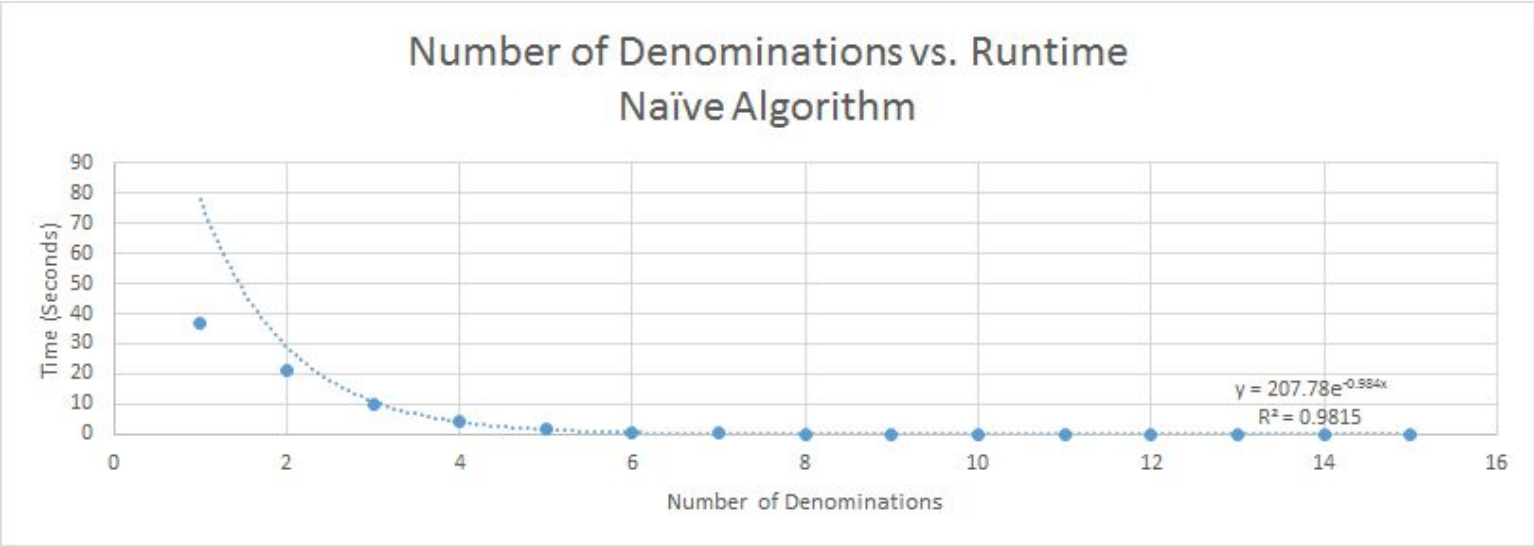
8) It is important to note that experimentally analyzing the runtimes of these algorithms is very difficult, due to the number of variables involved. Beyond the number of denominations and amount of change to be made, the types of denominations available could make a large difference in runtime. Below, we just tested using an increasing denomination size and single value to make change for. This gave us reasonable data, but it is not perfect because of this. Truly evaluating all of the variables in these problems is beyond the scope of this project.



This graph is of the greedy algorithm when run against an increasing number of denominations, ie. [1], [1, 2], [1, 2, 3], [1, 2, 3, 4], etc., and making change for the same amount (3000) each time.



This graph is of the dynamic programming algorithm when run against an increasing number of denominations, ie. [1], [1, 2], [1, 2, 3], [1, 2, 3, 4], etc., and making change for the same amount (1000) each time.



This graph is of the brute force, or divide and conquer (also sometimes called naive), algorithm when run against an increasing number of denominations, ie. [1], [1, 2], [1, 2, 3], [1, 2, 3, 4], etc., and making change for the same amount (15) each time.

9) If the coins are all powers of each other, then the greedy algorithm would likely be very good at coming up with optimal solutions. The cases where the greedy method fails are cases where the denominations of coins do not divide into each other as problems 4, 5, and 6 show. Integer powers, by definition, do divide each other, so the greedy algorithm would produce an optimal solution for such a system of money.

References

http://math.stackexchange.com/questions/1294464/the-change-making-problem-algorithm-proof-at-the-dynamic-programming-method