Ethan Takla
Kenneth Thompson
Adam Sunderman
Group 38

Project 2 Group 38: Coins

1. Theoretical Run-time Analysis
   a. **Changeslow**: *Changeslow has a worst case theoretical runtime of $O(d^n)$. Where d is the number of denominations and n is the amount we are making change for. Changeslow loops through d calling itself twice at each iteration, once with (d,n) and again with (d, $n-d_i$). In each iteration Changeslow only picks one coin.*

```
changeslow(array, value):
        newArray = []
        coinReturn = value
        #base case
        if array[i] == value:
                newArray[i] = 1
                return newArray, 1
        #recursive case
        for j in array:
                if j <= value:
                        lSideCoinReturn, lSideArray = changeslow(array, j)
                        rSideCoinReturn, rSideArray = changeslow(array, val - j)
                        currentTotal =  lSideCoinReturn + rSideCoinReturn
                        if currentTotal < coinReturn:
                                coinReturn = currentTotal
                                for k in range(len(array)):
                                        newArray[k] = lSideArray[k] + rSideArray[k]
        return newArray, coinReturn
```

   b. **Changegreedy:** *Change greedy has a worst case runtime of N, a best case run time of N, and an average runtime of N. Since it's just one loop, and it does 1 comparison every time, but it has to do that comparison each time, its just a simple, straightforward, N runtime.*

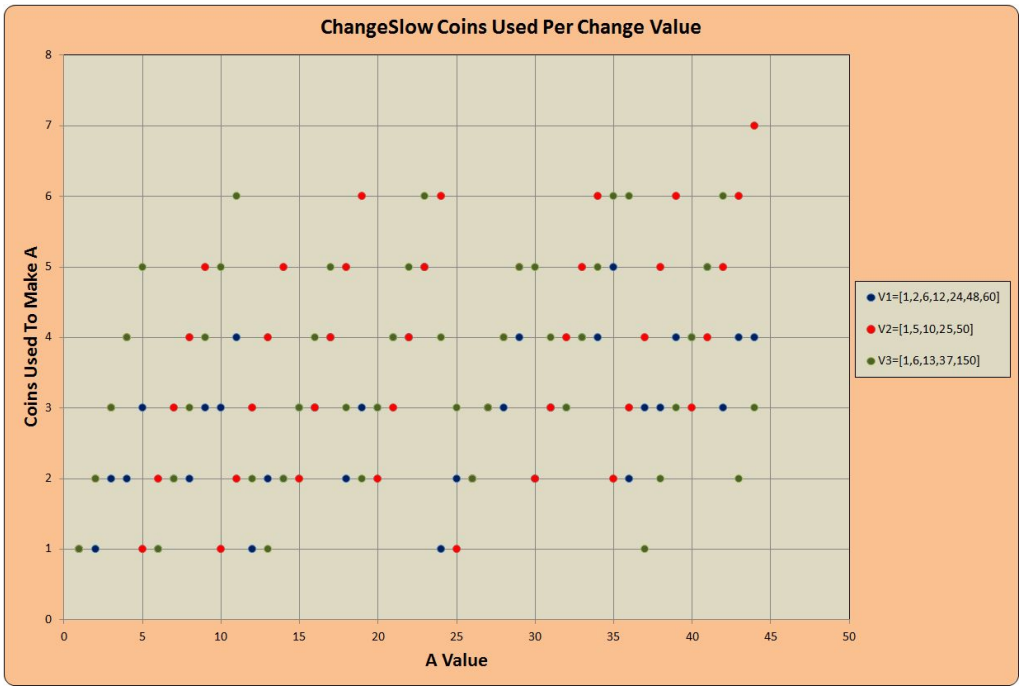
Changegreedy(v, a)
C = length of V, all values initialized to zero
For length to 0
        If V[i] is greater than or equal to A
                A -= V[i]
                C[i] += 1
                Repeat this step until A is greater than V[i]
Return C

c. **Changedp:** *As one can see in the below psuedocode, there is a for loop that executes n times (where n is the amount of change to make), and nested for loop within that executes d times (where d is the total number of coin denominations). The theoretical running time is thus Θ(nd).*

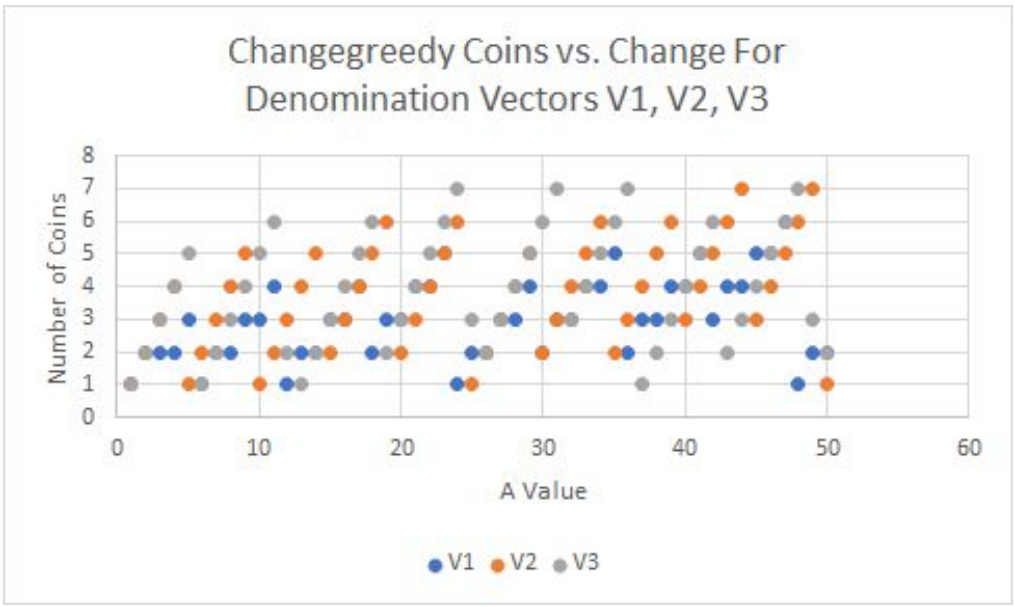
```
/*
V       -> Vector of denominations,
           with size v
T[p]    -> Minimum number of coints
           required for p cents
L[p]    -> Location of the first coin
           in an optimal solution for
           p cents
*/

for i = 1 to amount
    min = INT_MAX
    for j = 0 to v
        if V[j] <= i
            if 1 + T[i-V[j]] < min
                min = 1 + T[i-V[j]]
                coin = j
    T[i] = min
    L[i] = coin

for (i = amount; i > 0; i -= V[L[i]])
    push L[i] to results

minCoins = T[amount]
```

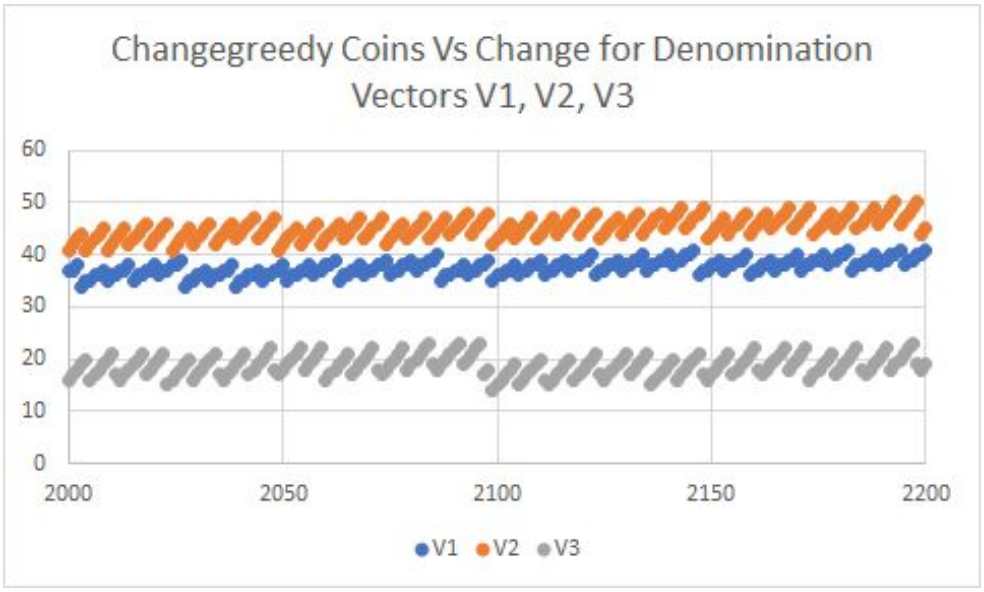
2. *According to the dynamic programming algorithm, if we have a first coin **c** in our best solution, the minimum amount of coins required to make T[p] cents (where **p** is the desired change amount and **d** is the denomination vector), would be T[p] = 1+ T[p-d[c]]. In order to build up the solution, we first start at 1, and iterate up to the desired amount of change. For every change amount **i** in this loop, we try to minimize 1 + T[p-d[c]], assuming d[c] is less than or equal to the current amount we're trying to make change for. Every time a new minimum is found for amount **i**, the min is recorded, and the index of the current denomination is recorded in vector L. This index corresponds to the first coin in the optimal solution for **i** cents. Once the loop and it's nested for loop run, the two tables/vectors T and L remain. In order to generate the result vector with the correct coins, L[p] (which corresponds to the first coin in the optimal solution for p), is first recorded. Subsequently, the coin value of L[p] is subtracted from p, and the new value of L[p] is recorded. This is done in an iterative manner until the amount is less than zero. This method essentially finds the first optimal coin for amount **p**, reduces the amount by this coin value, and then finds the first coin in the optimal solution for this now smaller change amount. Aside from the result vector we can simply find the minimum number of coins by finding T[p].*

3. Approach Comparison
    a. **Changeslow**:

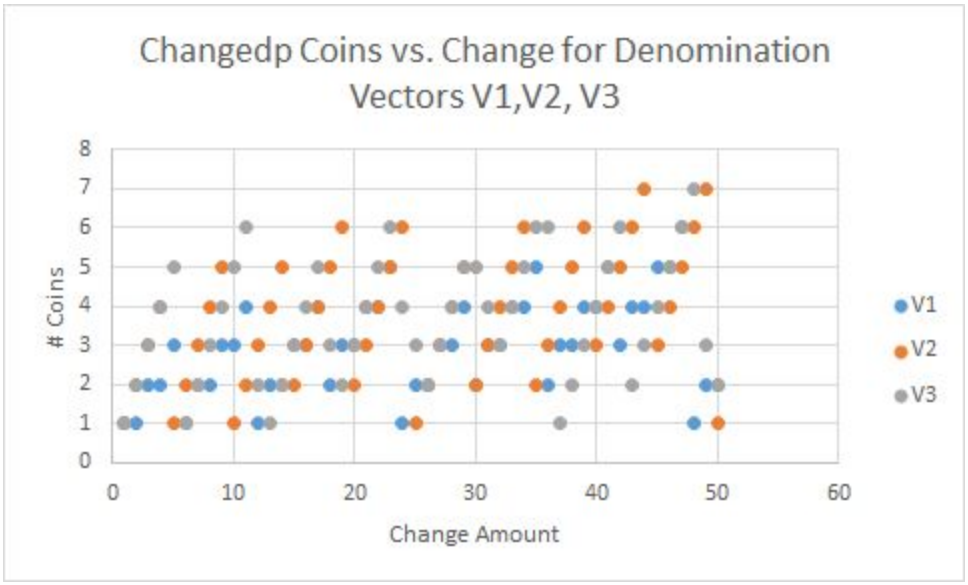


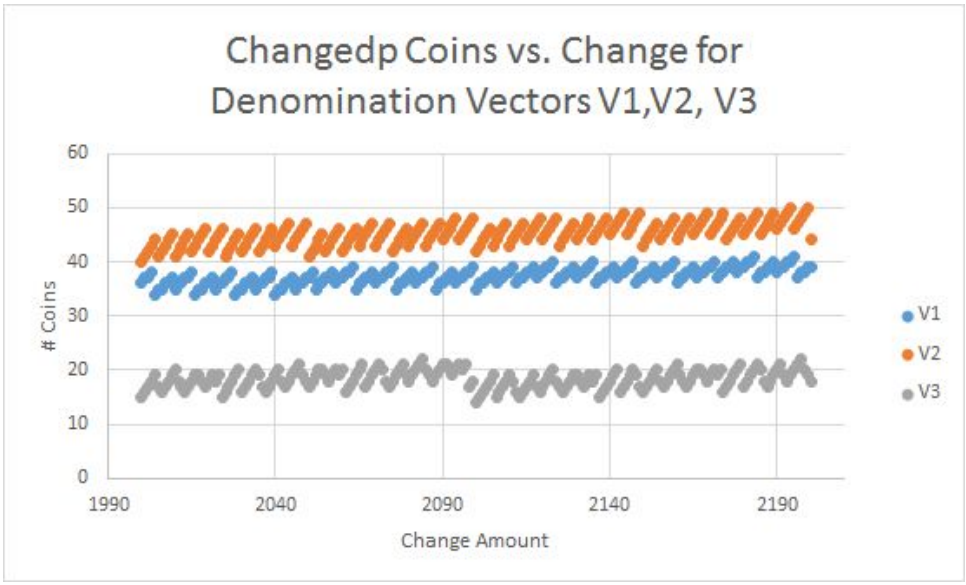
    b. **Changegreedy:**



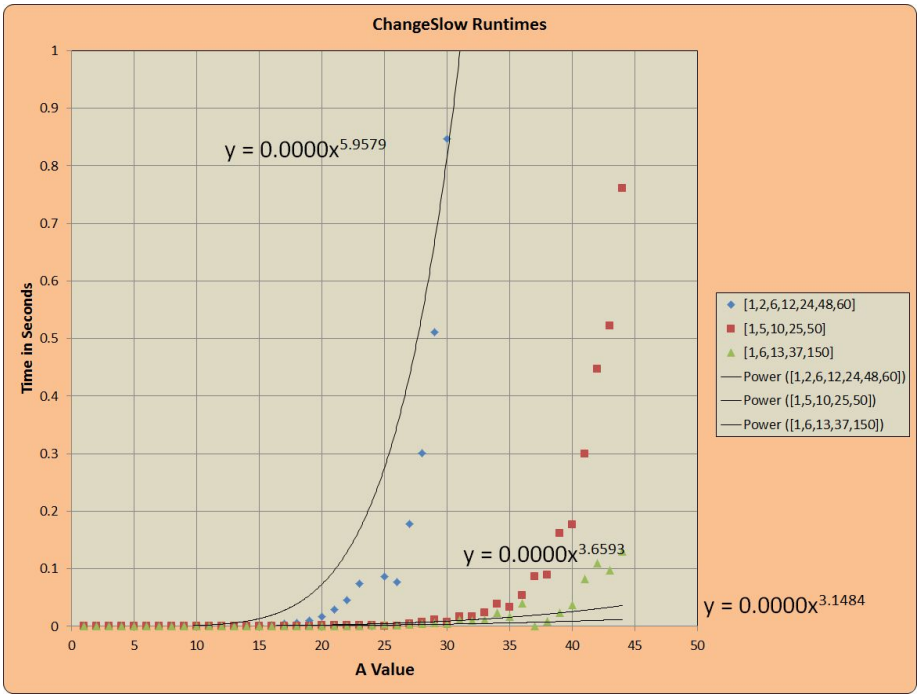


c. **Changedp:** *The two below graphs show the amount of coins needed to create change amounts 1-50 and 2000-2200 respectively for each of the three denomination vectors.*

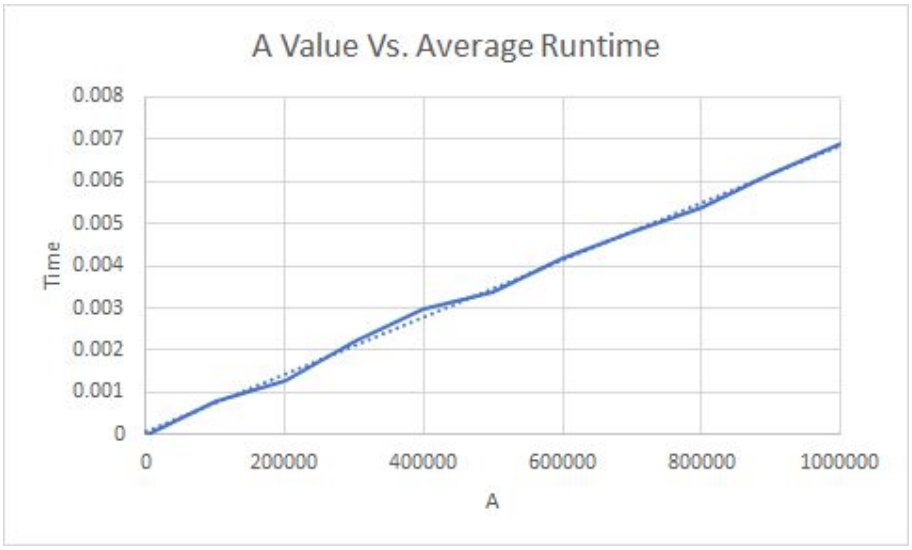



Changedp Coins vs. Change for Denomination Vectors V1,V2, V3
60
50
40
30
20
10
0
Coins
1990
2040
2090
2140
2190
Change Amount
V1
V2
V3

4. Experimental Analysis
    a. **Changeslow**: *Using Excel it was difficult to get an accurate regression model for ChangeSlow. ChangeSlow has two variables that make up an exponential function D, denominations and V, value to make change for. I was unable to figure out how to customize the regression equation to the form of* $f(D,V) = D^V$

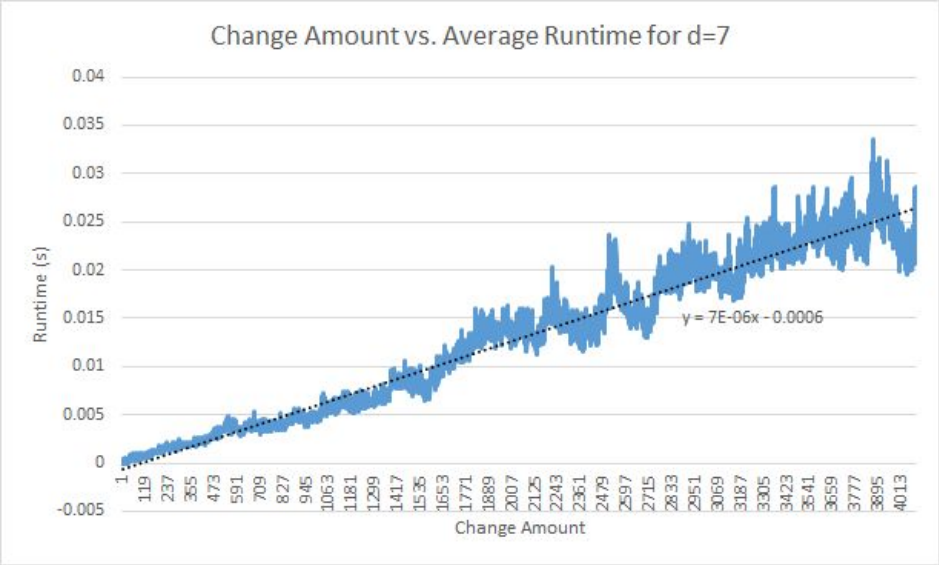


    b. **Changegreedy:**



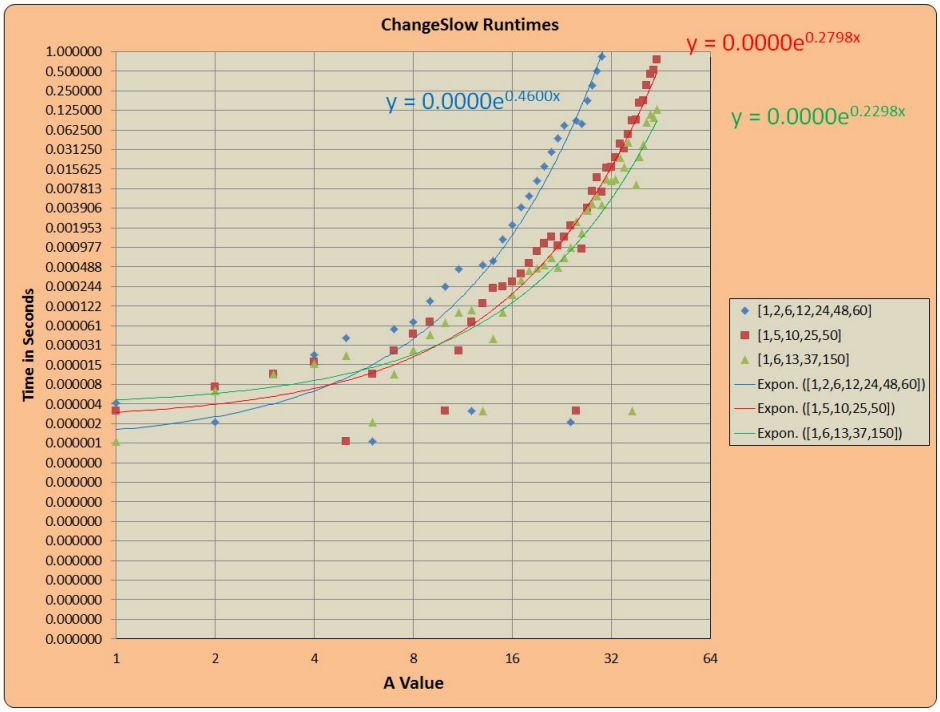
0.0000000068x = y or 6.8*10^-9 * x = y

c. **Changedp:** The theoretical **linear** runtime, *Θ(nd), matches the trend found in the collected data, having an equation of 7E-06n - 0.0006.*

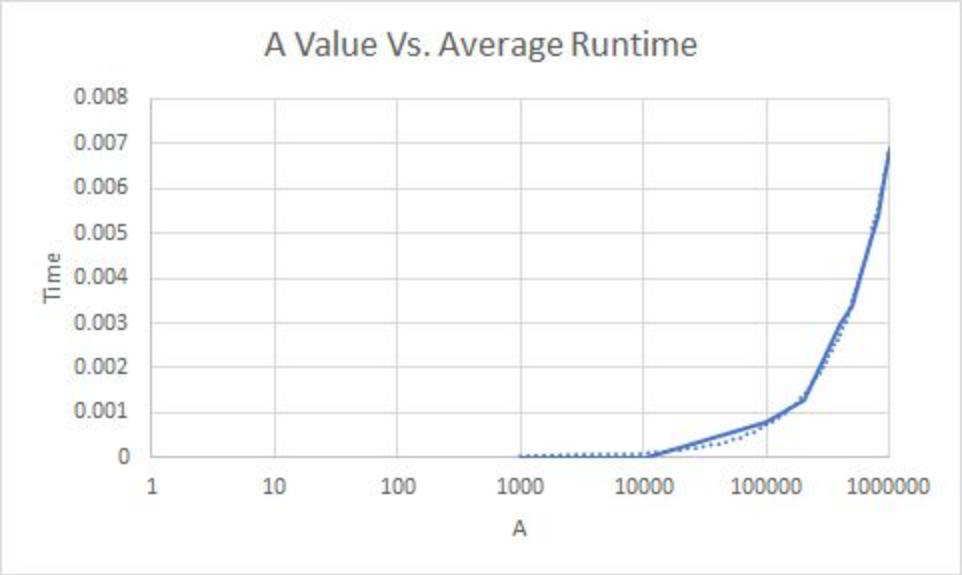


5. Log-Log plots

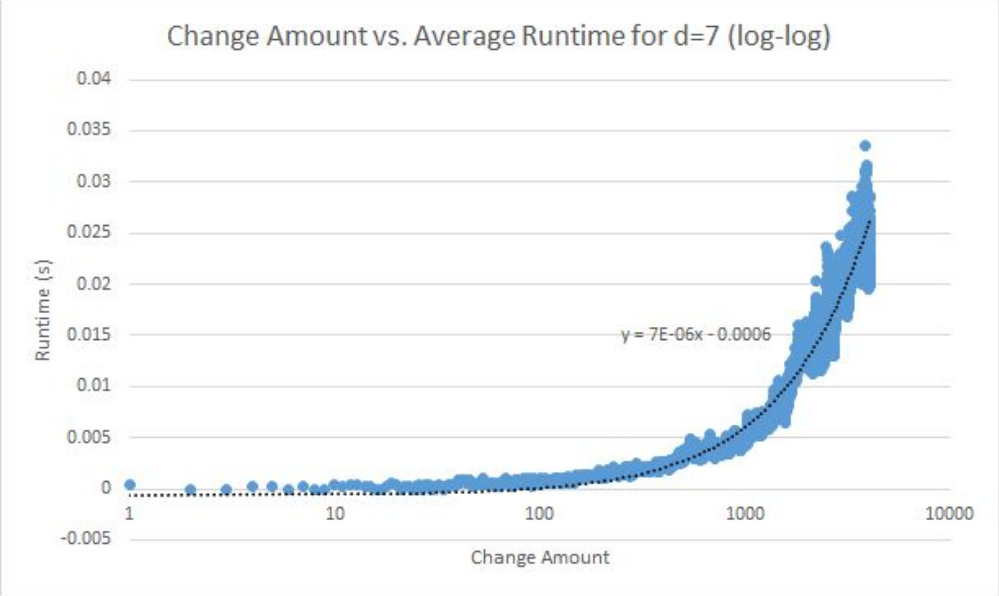
a. **Changeslow**:



b. **Changegreedy:**



c. **Changedp:**



d. **Runtime Comparison:** *Due to the fact that each of the three separate algorithms were run on different computers, plotting them all on the same graph wouldn't really make sense. In addition to this, the n values differed between the three algorithms due to constraints on run times. The greedy and change dp methods have very similar trendlines, while the slow algorithm's runtimes grow much quicker.*

6. *For a coin denomination vector V = [1,3,9,27], both the greedy and dynamic programming algorithms would generate optimal results. Because each element is a factor of it's successor element, the greedy algorithm performs very well. Run time wise, the greedy algorithm would perform faster due to the O(n) vs. Θ(nd) runtimes.*
7. *There are many instances where the Greedy algorithm is optimal. For example, if the denomination is 1, 2, 3, 4, greedy algorithm will always have the best solution. Also for the denomination 1 5 25 125 20, and 1 3 9 27. It works in these cases because they have a optimal substructure, that is numbers are all multiples of each other so any solution a lesser one will work for will also work for a higher one.*