Scott Williams
Travis Robinson
Ben Henning
CS 325 - 401
Group 6
Project 2

How to fill in the dynamic programming table for changedp:

The way that we used to fill in the changedp table was to start at the bottom, initializing the first row to all zero's. On subsequent rows, we would go through each element of that row, going back a number of rows equal to the coin value of that element, and see which one was the smallest. Then we'd copy that row into the current row, and increment the appropriate element.

As an example, if we're on row 15, and the first element has a coin value of 1, we go back to row 14 and count the number of coins there. Then we go to element 2, and if it has a coin value of 5, we go back to row 10 (15-5) and count the number of coins on that row; we repeat this for all coin values, until we know which row gives us the smallest total number of coins; we then set the current row equal to that row, and increment the index that gave the smallest row.

Pseudocode for each algorithm:

Algorithm 1 (changeslow):

changeslow(A, V, length)

```
let C be a vector of length elements
if A = 0
        for i = 0 to length
                C[i] = 0
        return C
for i = 0 to length
        if V[i] <= A
                let C2 = changeslow(A - V[i], V, length)
                C2[i] = C2[i] + 1;
                let returnedSum = sum(C2[0] ... C2[length]
                let currentSum = sum(C[0] ... C[length]
                if returnedSum < currentSum
                        C = C2
return C
```

Algorithm 2 (changegreedy):

```
runningTotal = A
denomination = array_of_coin_values.length-1
arrayC[length] = 0;
while(denomination>=0){
        if (inputArray[denomination]<= runningTotal{
                runningTotal = runningTotal-inputArray[denomination]
                arrayC[denomination]++;
        }
        else{
                denomination--;
        }
}
```

Algorithm 3 (changedp):

```
arrayC[A+1][#_of_coin_denominations]
initialize row 1 to 0
for(rows=1<=A){
        for(columns =0<#_of_coin_denominations){
                int denomination;
                int m = +infinity
                for(i=length;i>=0;i--){
                        if(array_of_coins[i]<= rows){
                                subM = 0
                                for(j<#_of_coin_denominations){
                                        subM = subM+arrayC[rows-array_of_coins[i]][j]
                                }
                                if(subM<m){
                                        m=subM
                                        denomination = i
                                }
                        }
                }
                for(h<#_of_coin_denominations){
                        arrayC[rows][h]=arrayC[rows-array_of_coins[i]][h]
                }
                arrayC[rows][denomination]++
        }
}
```

Proof for the dynamic programming approach:

Given a set V = {v_1, …, v_k} of coin denominations, let *T(v)* denote the minimum number of coins (with repetition) in V needed to get sum *v*. Then *T(v)* must be greater than 0 for all *v* and *T(v)* = 0 implies that v = 0.

If *v* > 0 and a way to find *v* with *T(v)* coins uses at least one coin of denomination *V[i]* (at least one such *i* exists), then by removing this coin we have a way to find *v - V[i]* and can conclude that:

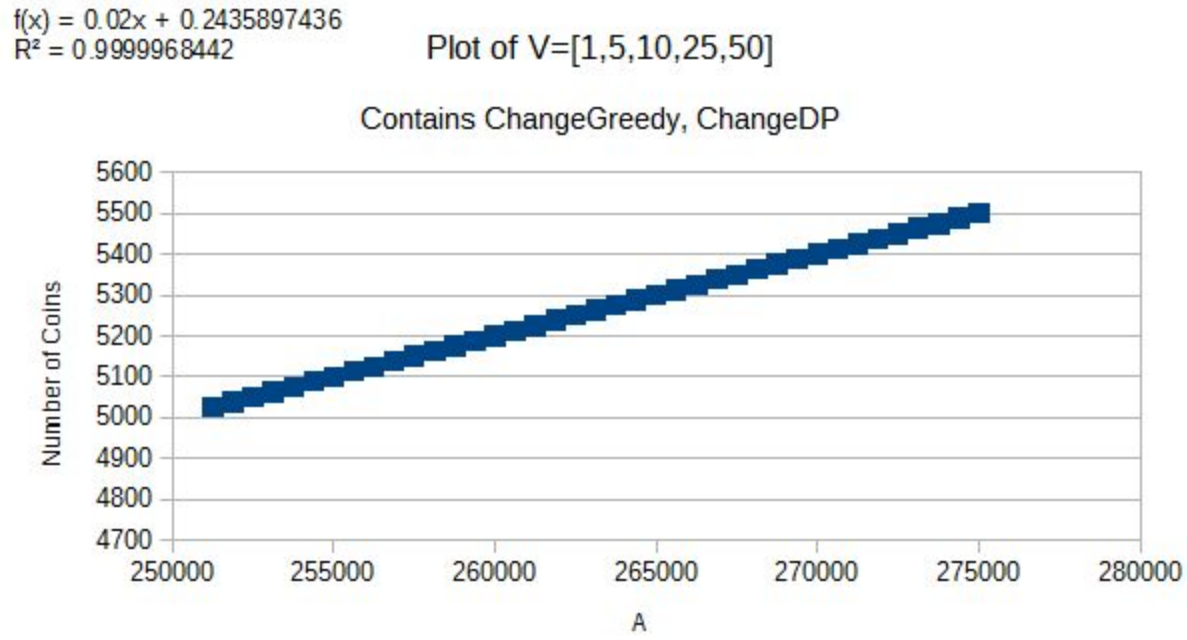
T(v - V[i]) is less than or equal to *f(v) - 1* for at least one value of *i* between *1* and *k*

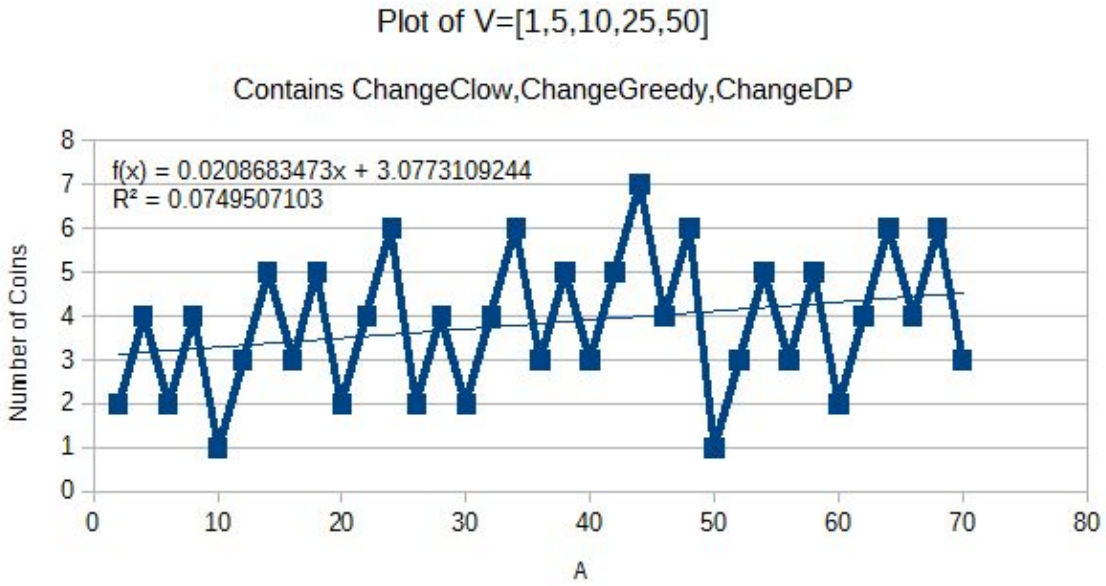
Similarly, we can conclude that if *i* is between *1* and *k* and n is greater than or equal to V[i], we can find *T(v - V[i]) + 1* coins by adding a *V[i]* coin. We conclude that:

T(v) is less than or equal to *T(v - V[i]) + 1* for all values of *V[i]* between *1* and *k* with n being greater than or equal to *V[i]*

This means that $T(v) = min_{v[i]v}\{ T[v - V[i]] + 1 \}, T[0] = 0$

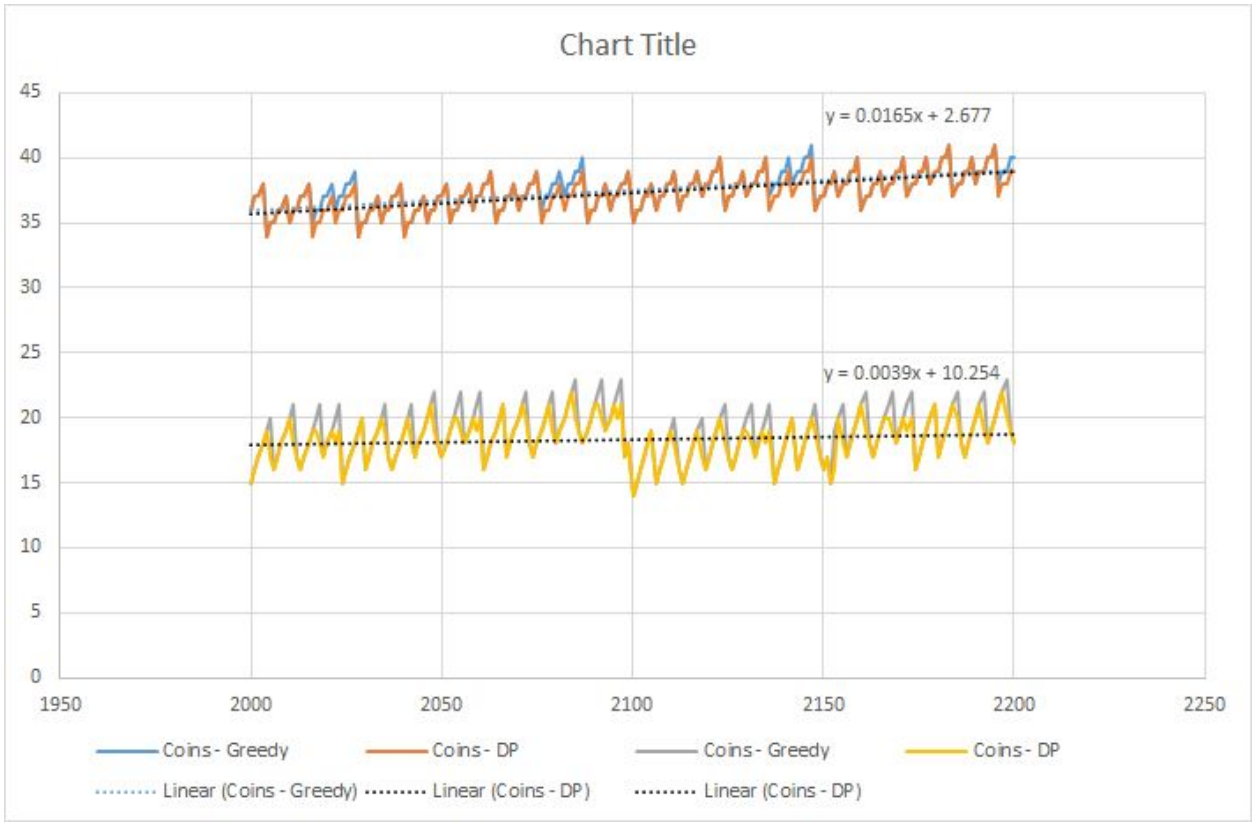
4. Comparison of V = [1, 5, 10, 25, 50]

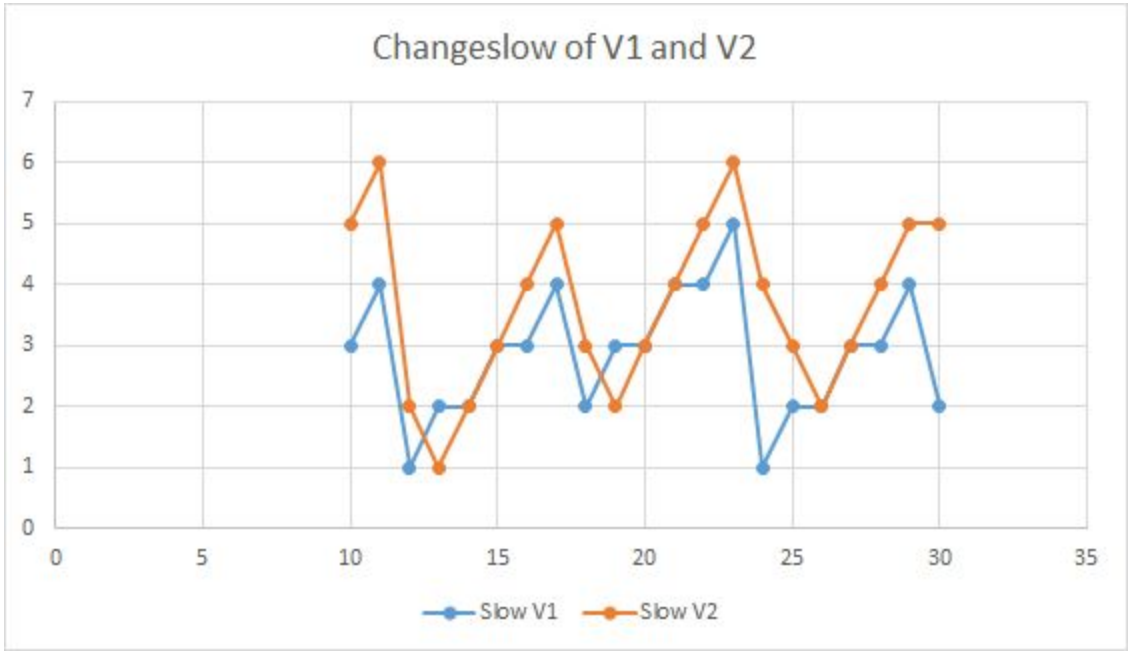




For the values of A that were tested, all approaches produced the same number of coins needed for each value of A. This is due to the value of coin denominations used; if one coin is at least twice as large as the next smaller coin denomination, then the greedy method will produce correct results. For this set of denominations, then, the greedy method is superior because it operates faster and is easier to code. The slow method is inferior because it is slower than the others.

5. Comparison of V_1 = [1,2,6,12,24,48,60] and V_2 = [1,6,13,37,150]

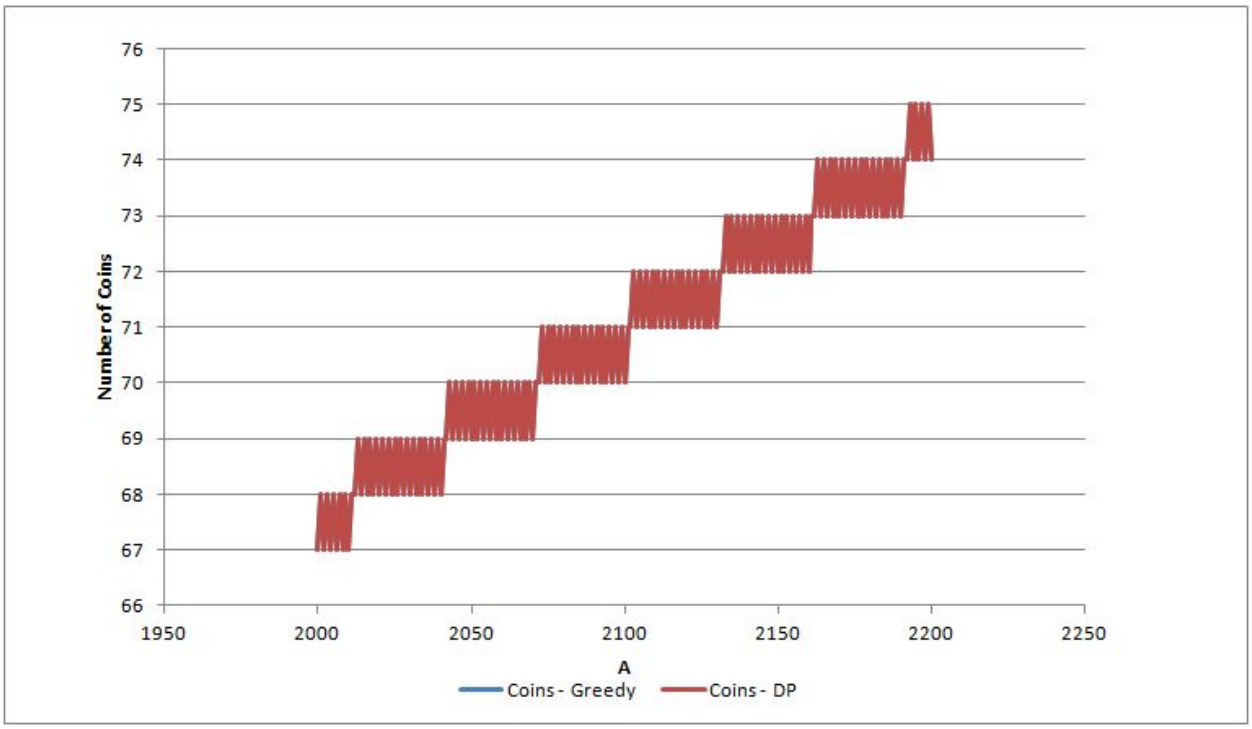




For V_1 we see that there are several instances where the greedy method produces a result that is not optimal, in that it produces numbers of coins that are higher than what the dynamic approach gives. For V_2, the greedy method still produces some instances where it requires too many coins, but these are are much more rare than under V_1. This difference is because of the

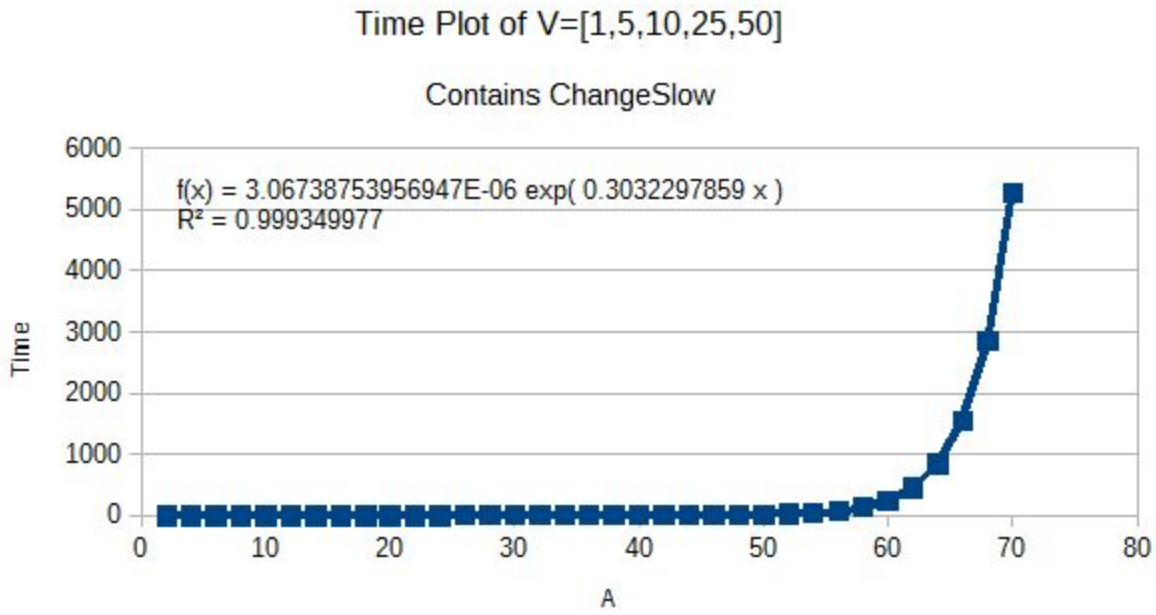
value of the coin denominations; when the coins are multiples of their predecessor, the greedy method is much better able to produce correct results than than it otherwise would.

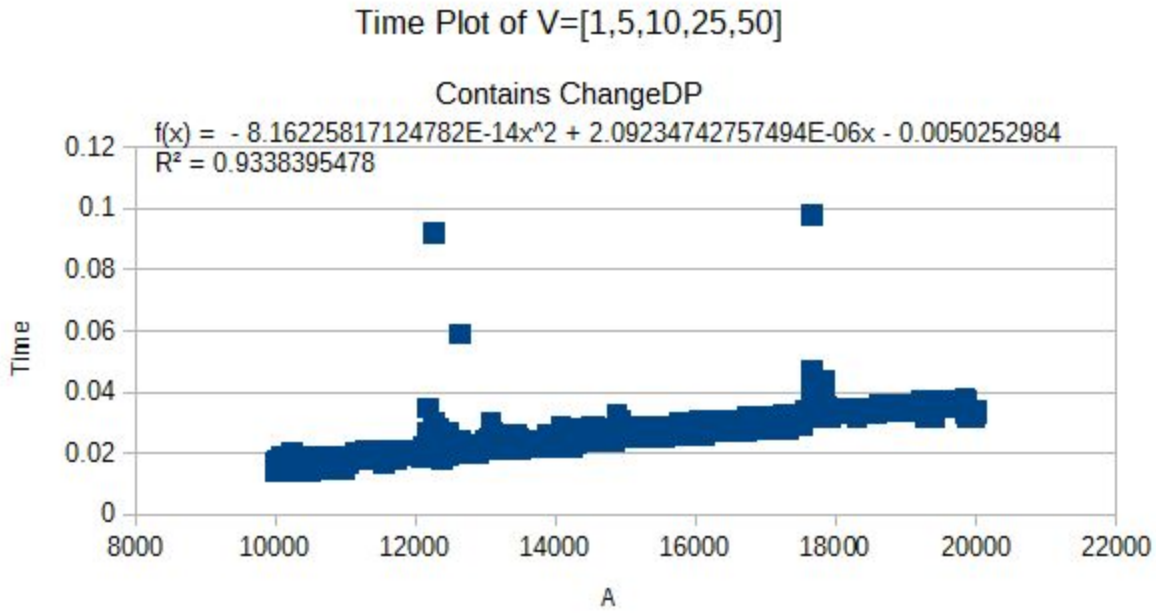
6. Comparison of V = [1, 2, 4, 8,10, …, 30]

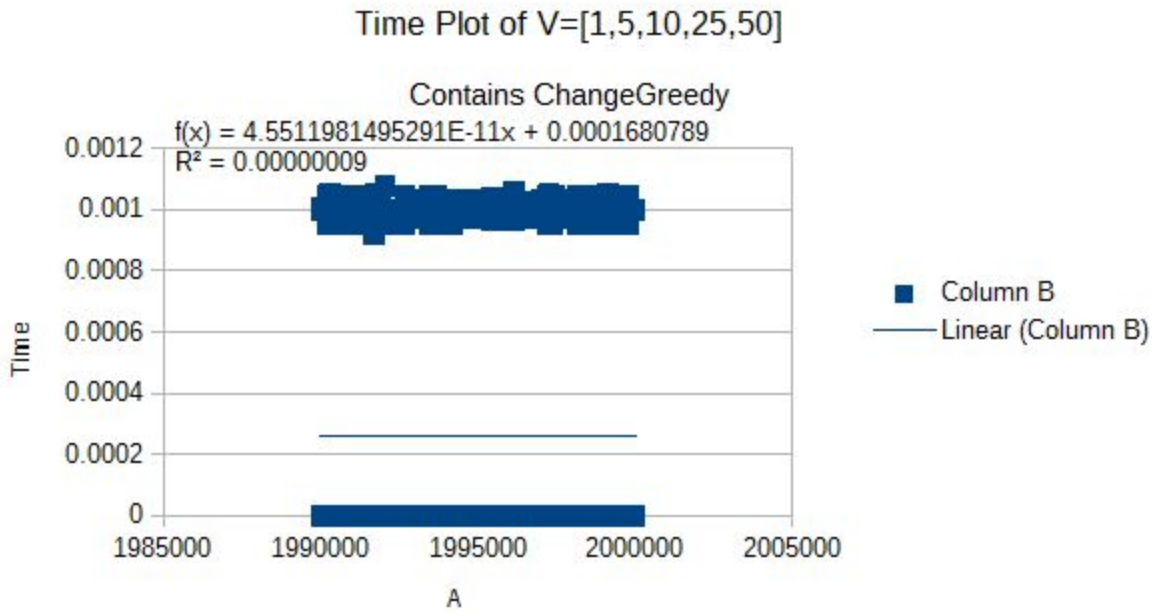


7. Time Plots for 4-6

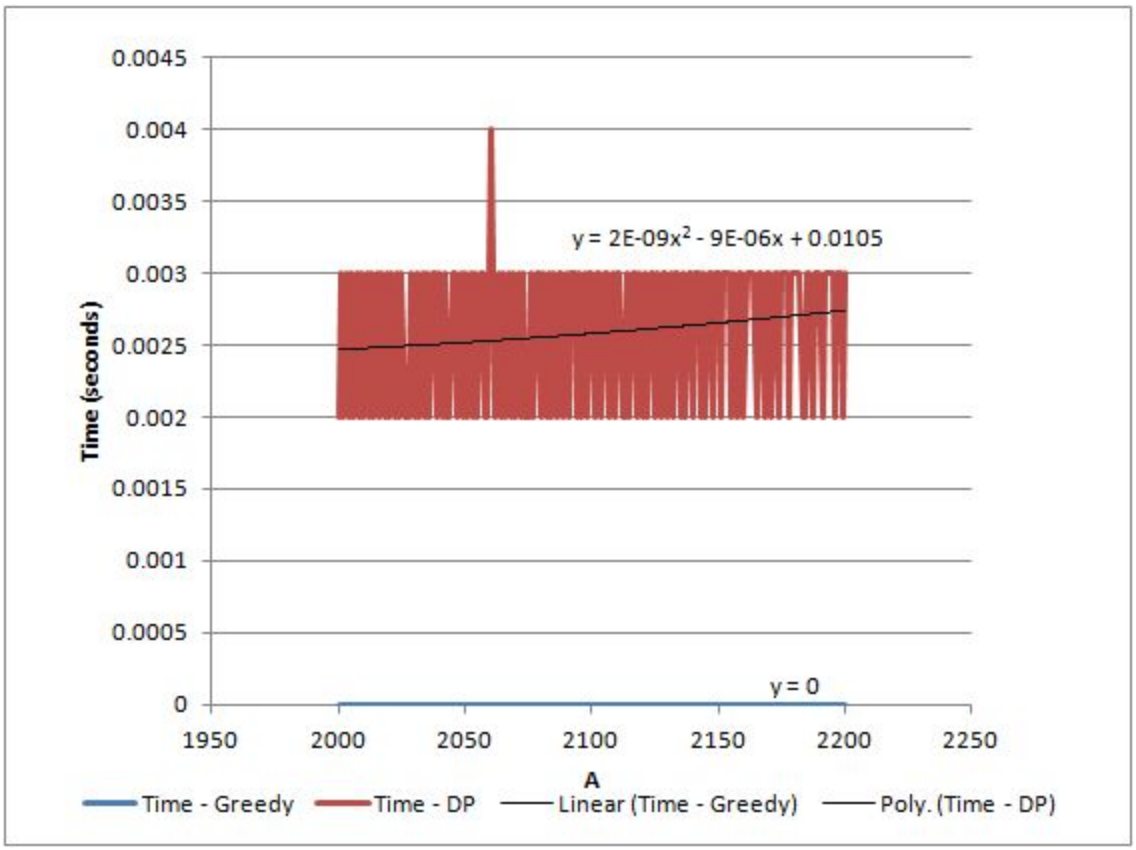
Time Plots for V=[1,5,10,25,50]



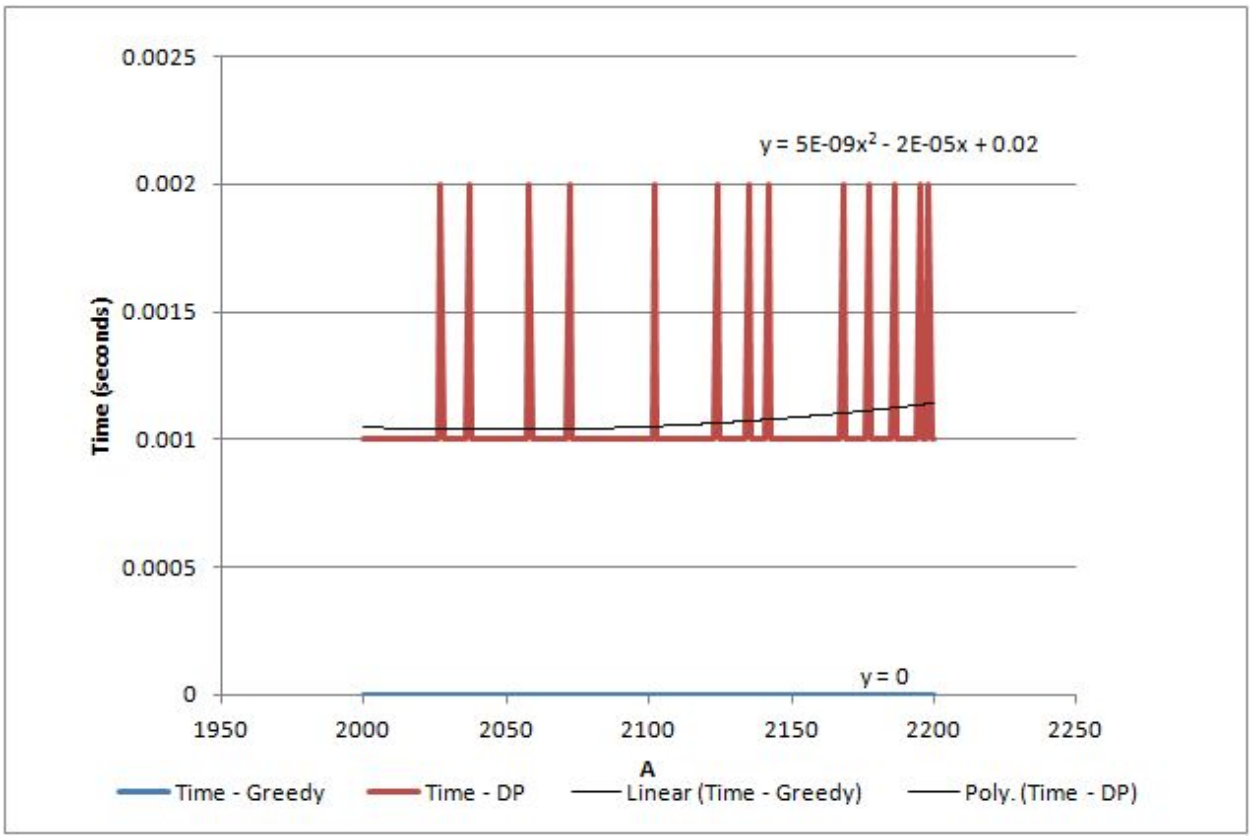




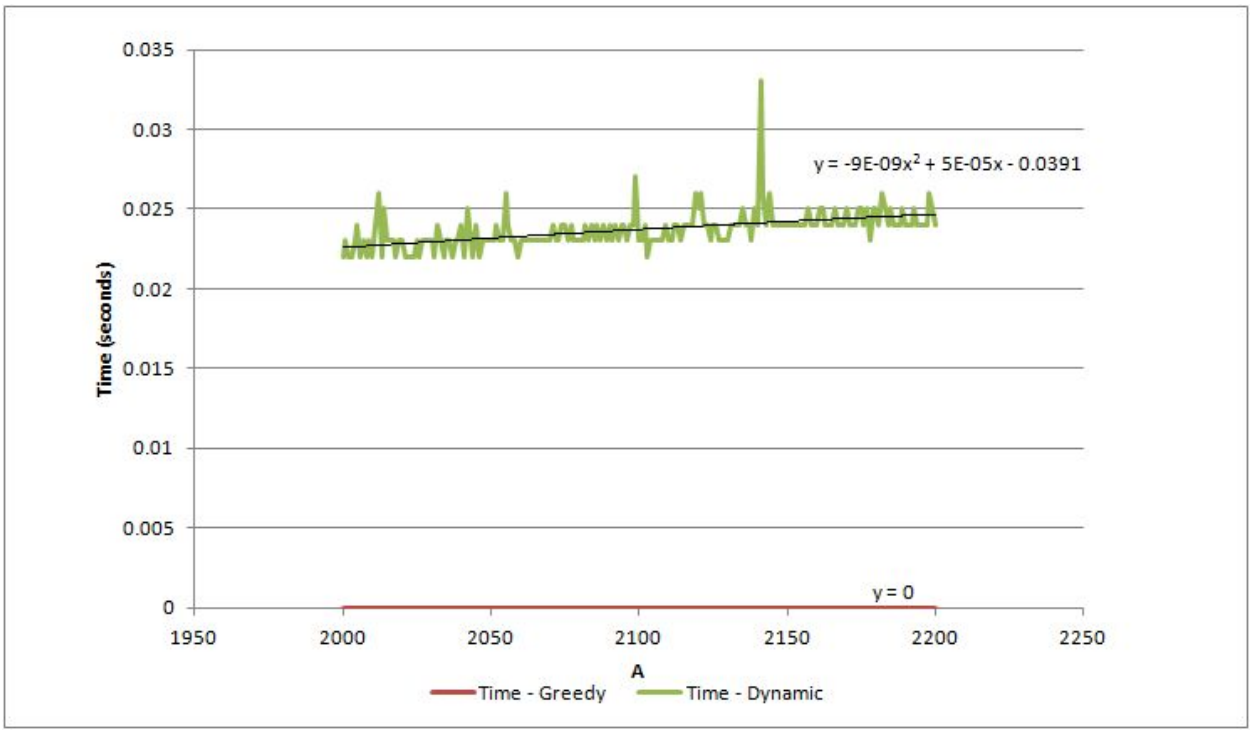
Time plot for V = [1, 2, 6, 12 ,24, 48, 60]



Time plot for V = [1, 6, 13, 37, 150]



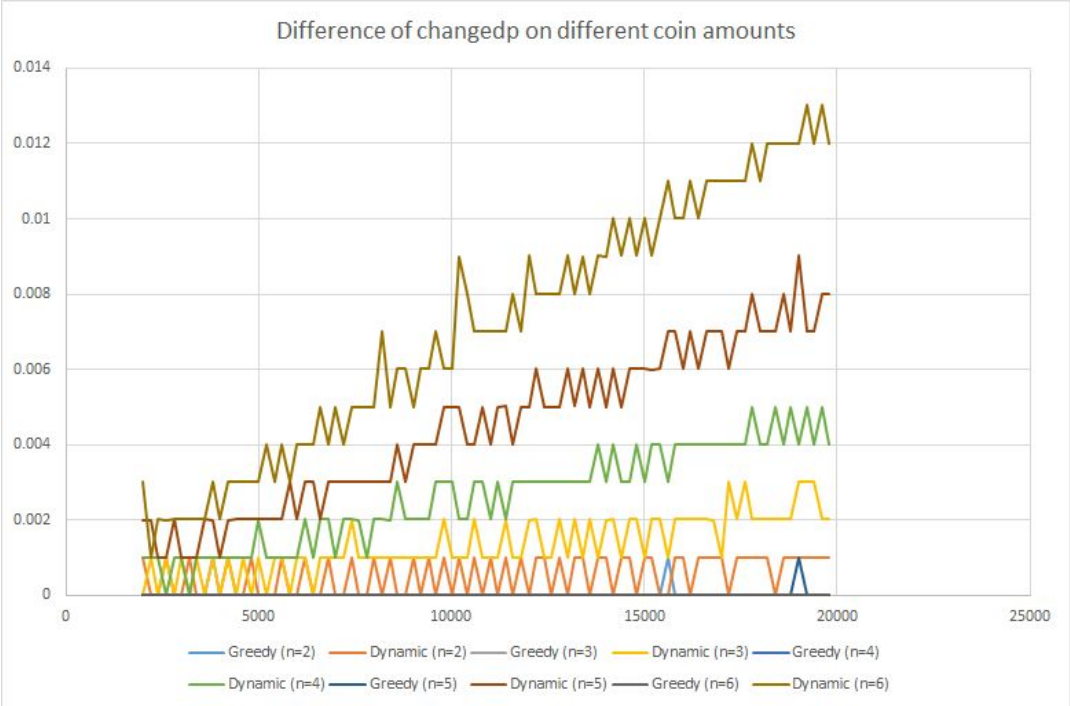
Time plot for V = [1, 2, 4, 8,10, …, 30]



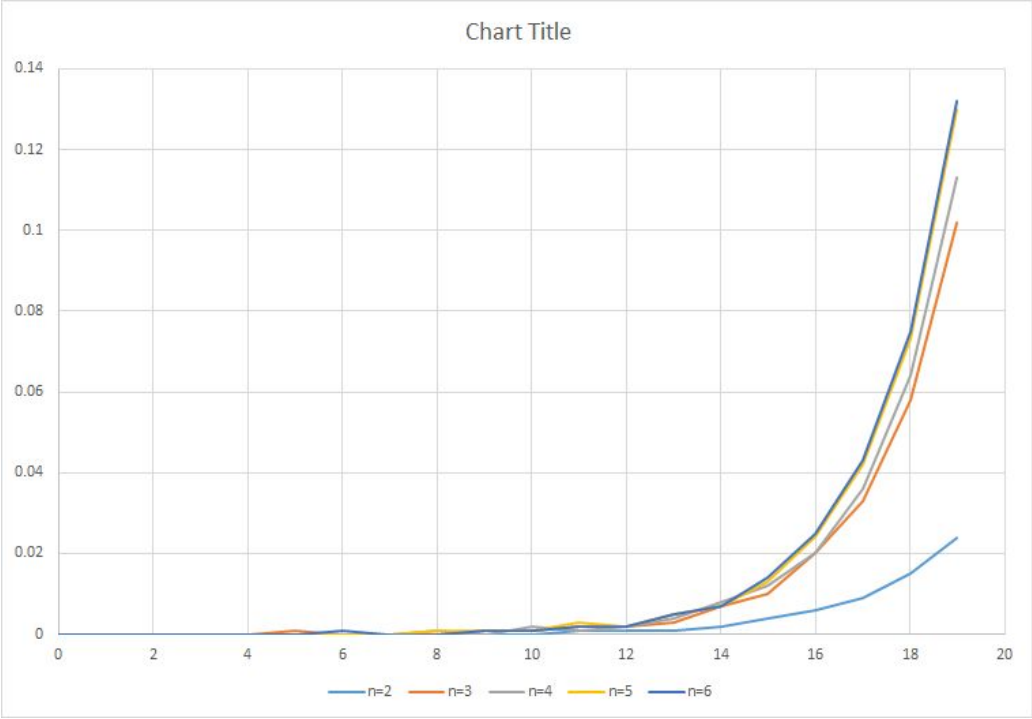
From the plots given using the different denominations with the different values of A, we can see that the changeslow algorithm is very slow; we see that as an exponential algorithm, it doesn't take a very large A before it runs into processing times measured in minutes or hours. The changedp is very quick, and, like changeslow, produces accurate results. The quickest method, changegreedy, gives very fast results; even when processing A values measured in the millions it would still oftentimes produce times that were too small for us to measure; unfortunately, the greedy approach, while very fast, does not produce correct results for all denomination sets.

8. Running times as a number of denominations

Graph of Changedp and changegreedy



Graph of changeslow over small values of A and n



Increasing the number of coins for changeslow drastically increases the amount of time it takes, even with small values of A. Changegreedy is a constant 0s for all values tested; up to 100,000. The plot of all changedp run-times are linear, but the constant associated with each plot increases exponentially with the number of coins.

9. Coin values with powers of p.

The difference in accuracy will depend on the value of p; for values less than two, changedp will not always produce accurate results, because changegreedy will only work if each denomination is at least twice as large as the previous; if p is greater than two, greedy will always produce correct results; otherwise, it doesn't seem like there would be much of a difference between any other denomination of coin. Changegreedy would be incredibly fast but sometimes inaccurate (if p is less than two) and Changedp would be slower, but always using the least amount of coins. The biggest factor in running time seems to be the number of coins, rather than the value of each coin.