Project 2 Final Report

By: Colleen Minor, Thomas Prefontaine, and Alexandre Silva

Group #46

Complete project turned into T.E.A.C.H by **ODIN username minorc**.

Question 1:

Greedy change algorithm pseudocode:

```
Input: array and value to be reached
Output: array where C[i] is the number of coins of value V[i], and the amount
of coins used.

Sort array

Going backwards from length of array - 1;
While value > = 0

             If array at index < = value
                                  Value -= array at index
                                  Coin used + 1;
                                  Count of coin used +1;
             Else
                                  Returning array at index = count
                                  Count = 0
                                  If(value == 0)
                                                      Break;
                                  Index -1;
```

Sorting the array takes nlogn time.

Then you have to go through the array of coins n, which means O(n) which I believe would make the asymptotic running time of this algorithm **Theta(n)**, however if the array needs to be sorted, the running time is **Theta(nlogn)**.

Dynamic Programming (changedp) pseudocode:

Notes specific to the dynamic programming (`changedp`) pseudocode:

- V is an array of coin denominations

- A is a value to make change for.

– X is an array where the index(idx) represents a number of cents, and the value at each index is the minimum number of coins it takes to get to idx cents. X[cents] = lowestNumCoins

-Matrix is an array where the index(m_idx) represents a number of cents, and value at each index is a [lowerXIndex, higherXIndex] element that represents the X[idx] indexes of lowest numCoins value for:

X[lowerXIndex] + X[higherXIndex] = numCoins, where

lowerXIndex + higherXIndex = m_idx

```
def changedp(V[1…. N], A):

    X = {0, 1}
    matrix = { [0,0], [0,1] }

    #Build matrix and X
    for i = 2 thru A:
        if i in V:
        #the index(i) is the denomination of one of our coins, so we only
    #need 1 coin to make i cents
            X.append(1)
            matrix.append([0, i])
         else:
            min = 1000
            #loop to find min{ (X[1]+X[len-1]), (X[2]+X[len-2]),
    #(X[3]+X[len-3])... (X[len-1]+X[1]) }
            for j = 1 thru len(X)):
                temp = X[j] + X[len(X) - j]
                if temp < min:
                    min = temp
                    lowerIndex = j
                    upperIndex = len(X) - j

        matrix.append([lowerIndex, upperIndex])
        X.append(min)

  coinPurse = []

  lastVal = last array in matrix

  #fill each element in coinPurse with a 0, in case no coins get added there
  while i < leng(A):
      coinPurse.append(0)
      i++

  loadCoinPurse(matrix, lastVal, X, V, coinPurse)

#return array of coin amounts + last value in X for minimum number coins needed:
  return coinPurse, X[n]

#loadCoinPurse is an algorithm that is cohesive to changedp
#It loads coinPurse with the proper values
def loadCoinPurse(matrix, lastVal, X, V, coinPurse):
```

```
            if lastVal[0] is 0:
                index = V.index(lastVal[1])
                coinPurse[index] += 1

            elif lastVal[0] is 1 and lastVal[1] is not 1:
                index = V.index(1)
                coinPurse[index] += 1
                lastVal = matrix[lastVal[1]]
                loadCoinPurse(matrix, lastVal, X, V, coinPurse)

            elif lastVal[0] is 1 and lastVal[1] is 1:
                index = V.index(1)
                coinPurse[index] += 2

            elif lastVal[0] is not 1:
                index = V.index(lastVal[0])
                coinPurse[index] += 1
                lastVal = matrix[lastVal[1]]
                loadCoinPurse(matrix, lastVal, X, V, coinPurse)
```

changedp runs in **Theta(n*k)** time where k is the amount of subproblems. We have intentionally avoided evaluating loadCoinPurse for this problem, because changedp does not need it to return the minimum number of coins.

Divide & Conquer (slowchange) pseudocode:

```
Divide and Conquer:
//using a nested function: changeslow is nested inside a helper function
def changeslowHelper(coinList, value):

    finalCoins = changeslow(coinList, value)
    coinSum = sum(finalCoins)
    return (finalCoins, coinSum)

     def changeslow(coins, change):
    //minimum number of coins needed initialized as an array
        minCoins[coin] = 0 for each coin that's in coinList
        minCoins[0] = change
    //for all coins, add the coins to the list of minimum coins if the sum is
greater than the sum of temp

        for coin in [c for c in coins if c <= change]:
            temp = ( changeslow(coins, change - coin))
            temp[coins.index(coin)] += 1
            if sum(minCoins) > sum(temp):
                minCoins = temp
                bestSum = temp
        return (minCoins)
```

Divide and conquer runs in **O(n^2)** time.

Question 2:

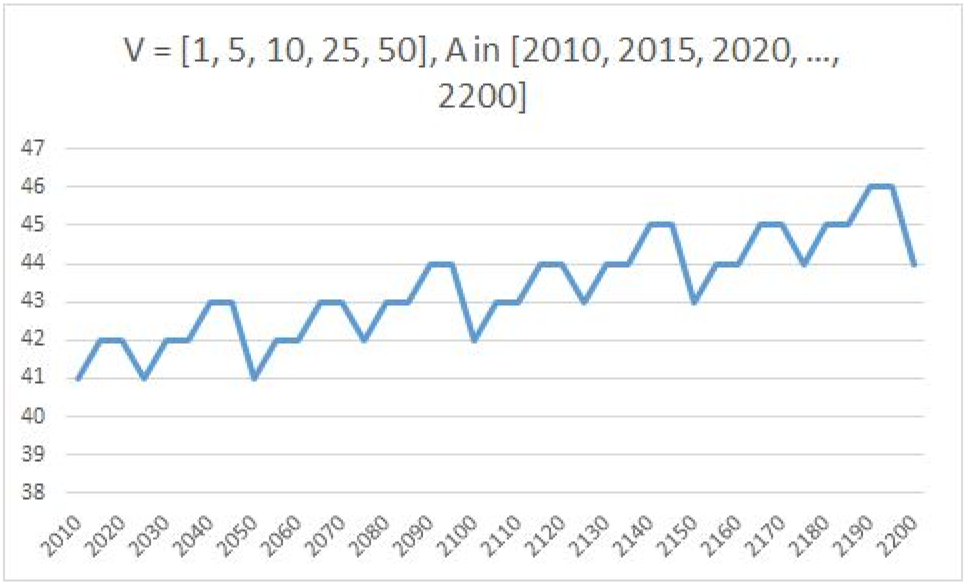
The table in changedp consists of 2 arrays: One to hold the values of the minimum number of coins needed per index (array *X*), and one to hold the values of the two indexes that were added together to find the minimum number of coins (array *matrix*). However, we only need to use matrix (and the function loadCoinPurse) if we need to fill the array that tells us how many of each type of coin we need.

How it works is that the array X starts with (0, 1) for the minimum number of coins needed to get to 0 and 1 cents, then it builds the minimum needed coins for each subsequent value up to A 1 by 1. For each array index to find the minimum number of coins for, it either sets 1 coin needed (if that index is the denomination of one of the coins) or it finds the minimum for { (X[1]+X[n-1]), (X[2]+X[n-2]), (X[3]+X[n-3])... (X[n-1]+X[1]) }. The logic is:

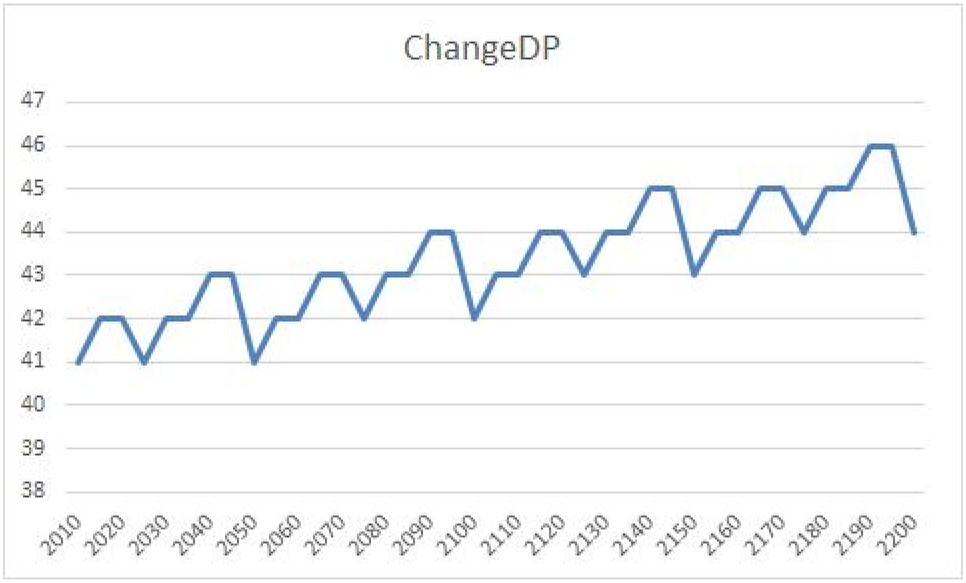
> If X[1] + X[n - 1] = X[n],
> numCoins(X[1]) + numCoins(X[n-1]) = numCoins(X[n])

loadCoinPurse works because the elements of *matrix* will never have both values be number that isn't one of the denominations in A. This pattern allows us to unwind the array with values of of its own indexes until we find all of the denominational coins used.

Question 3:
This is a graph of the greedy algorithm amount of coins it takes from V[i] to get A[i]

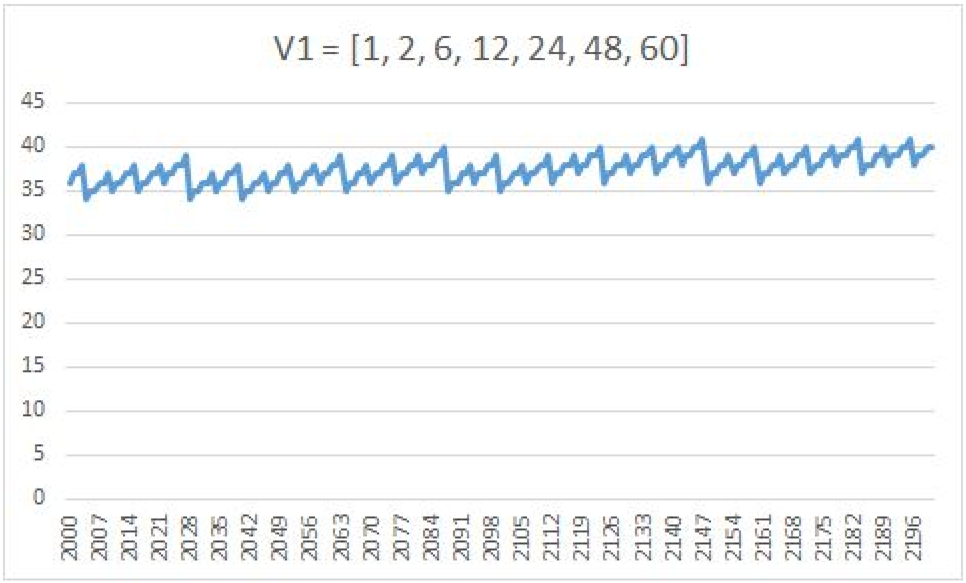


ChangeDp

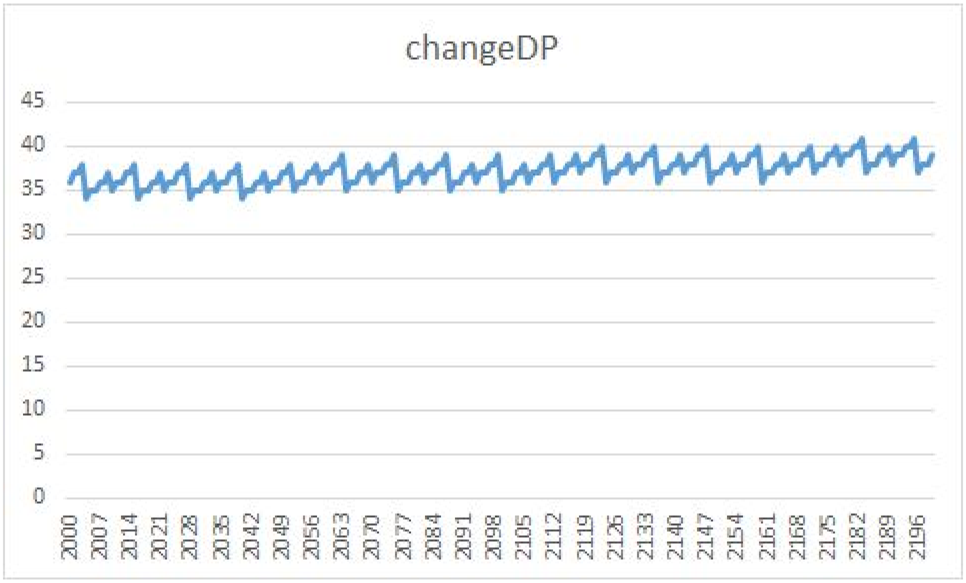


Question 4:
This is a graph of the greedy algorithm amount of coins it takes from V[i] to get A[i]

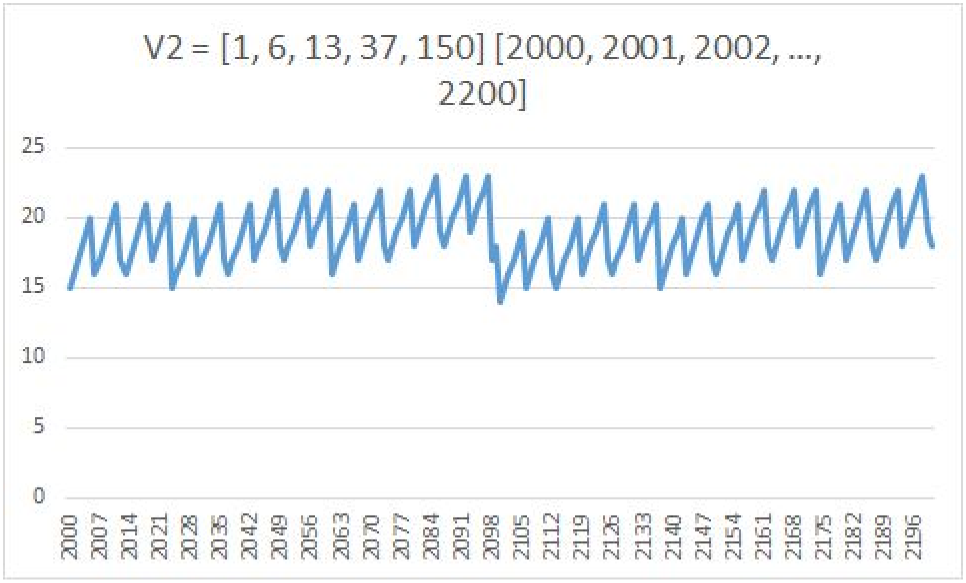
Greedy:



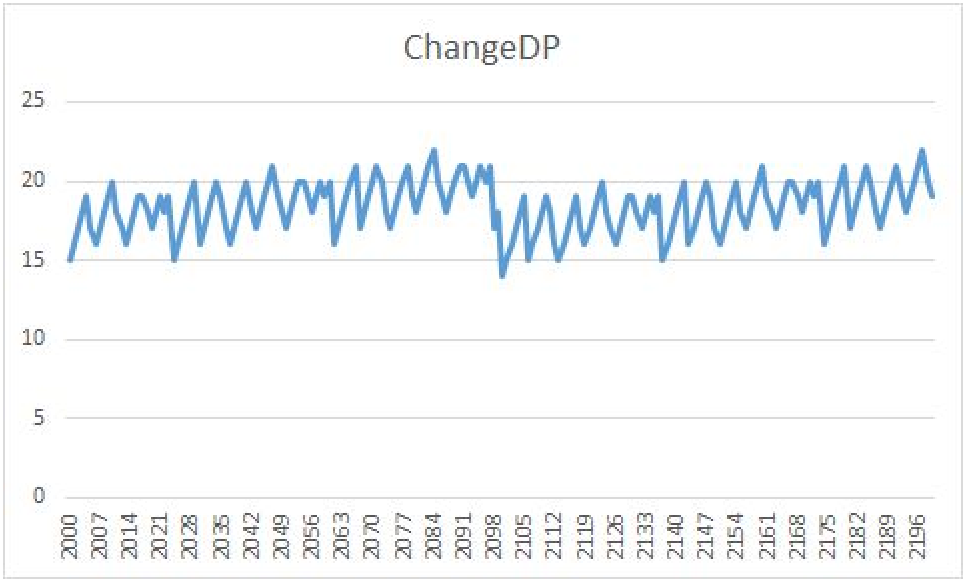
ChangeDP



Greedy:

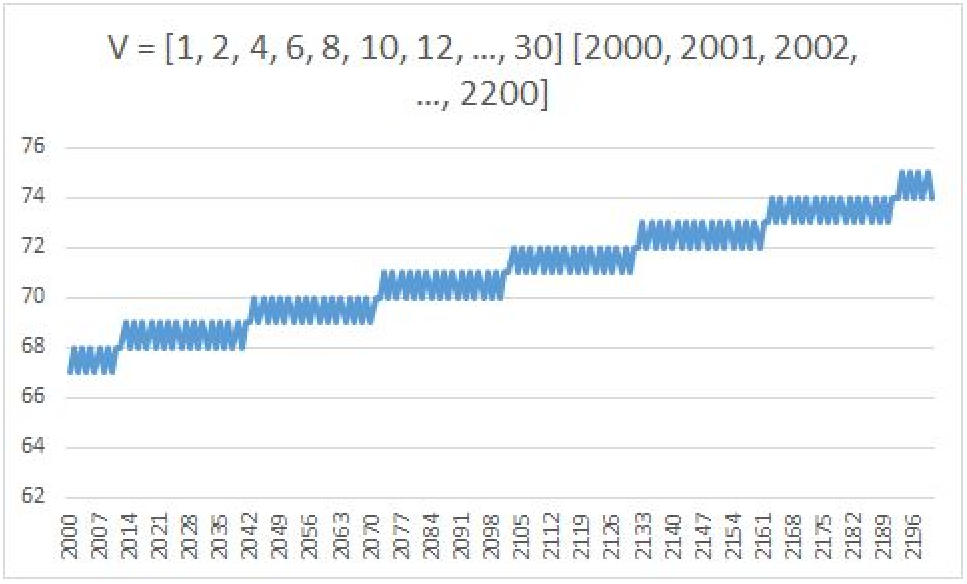


ChangeDP

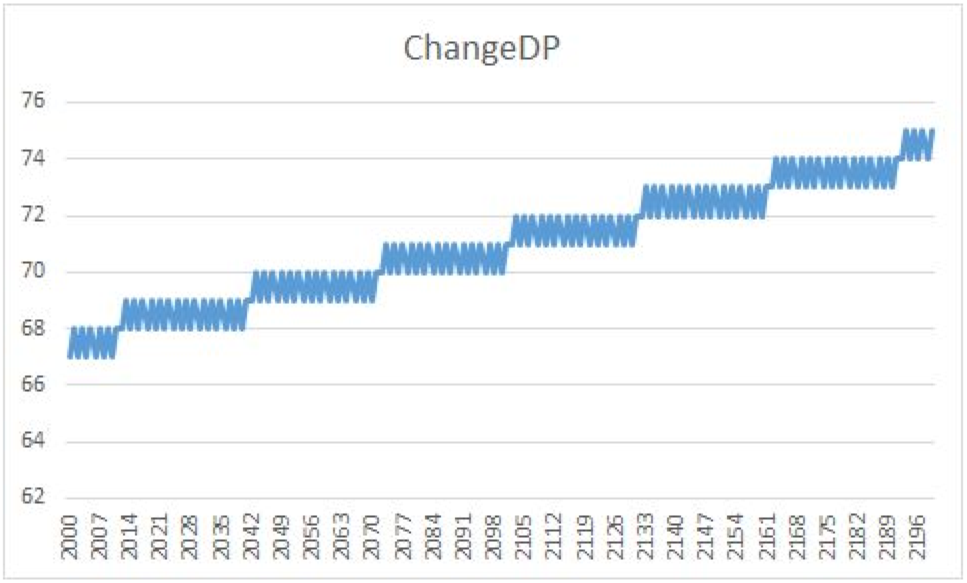


Question 5:
This is a graph of the greedy algorithm amount of coins it takes from V[i] to get A[i]

Greedy



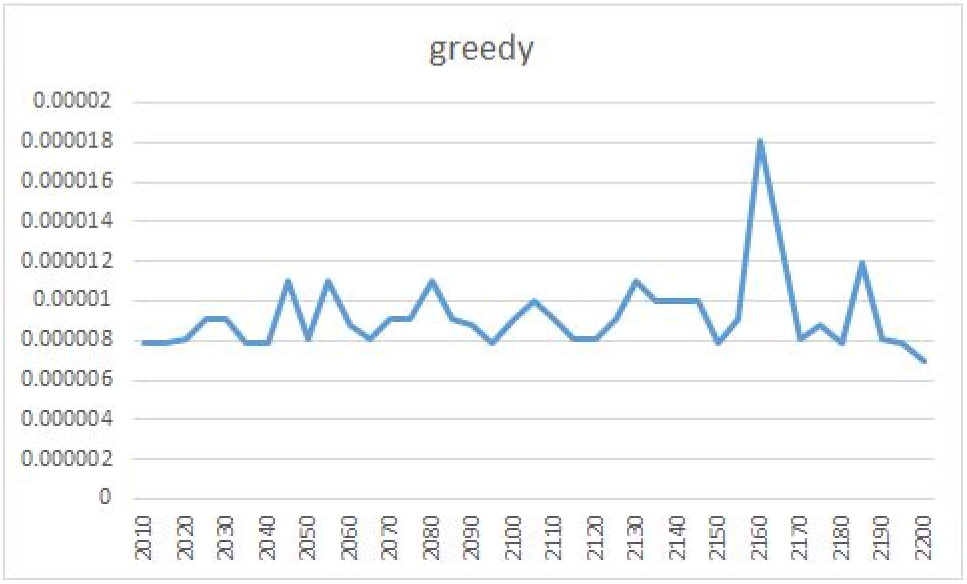
ChangeDP



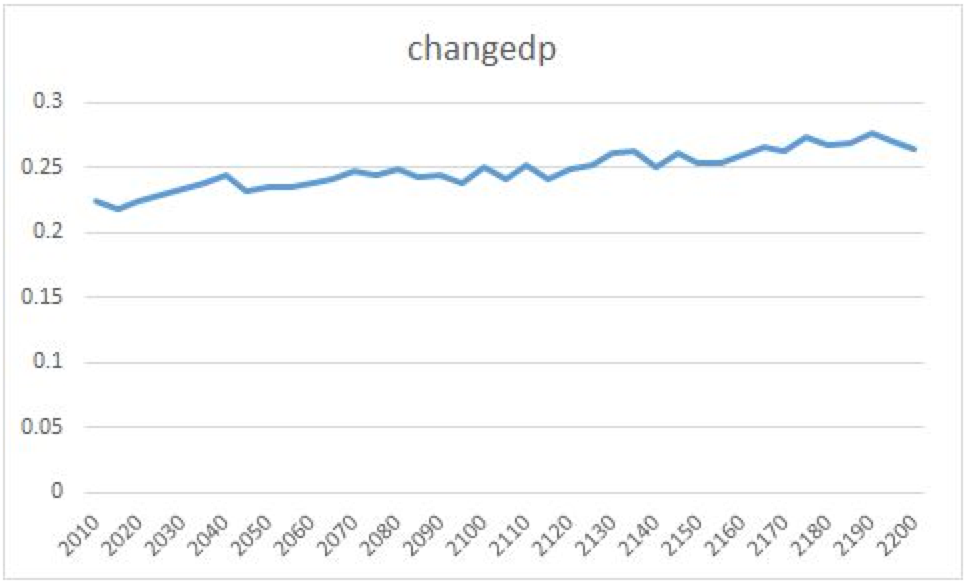
Question 6:

Running times for Question 3 data

Greedy:



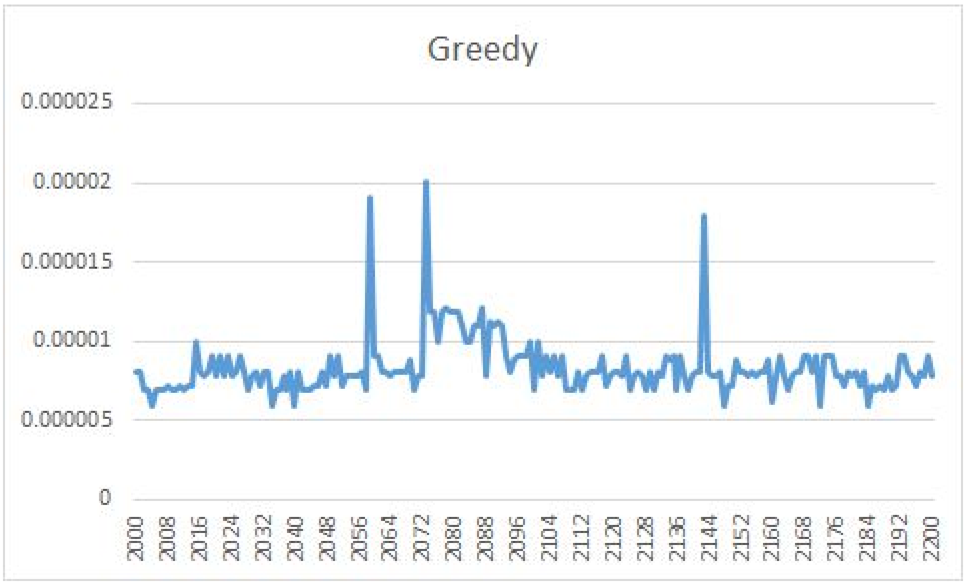
ChangeDP:



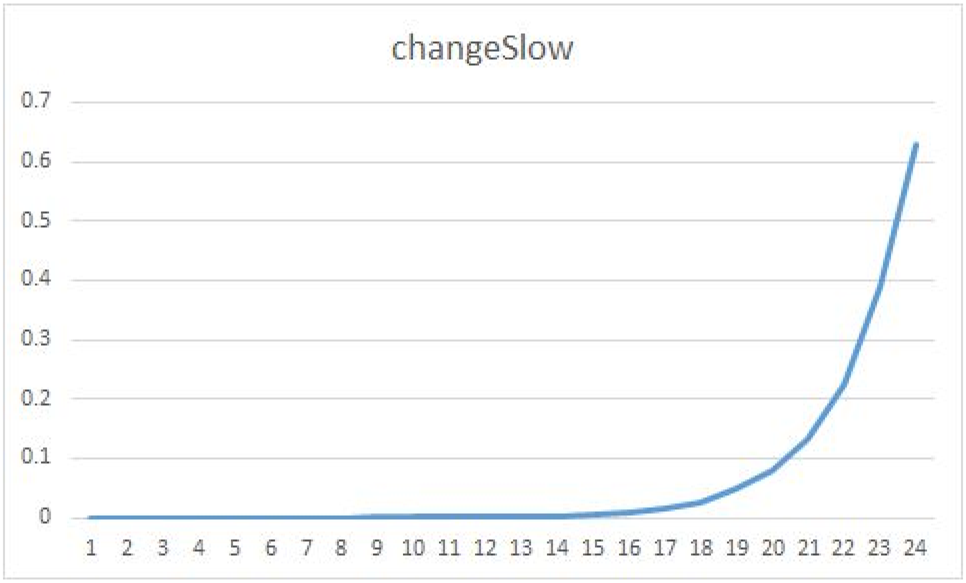
Running times for Question 4 data:

Greedy

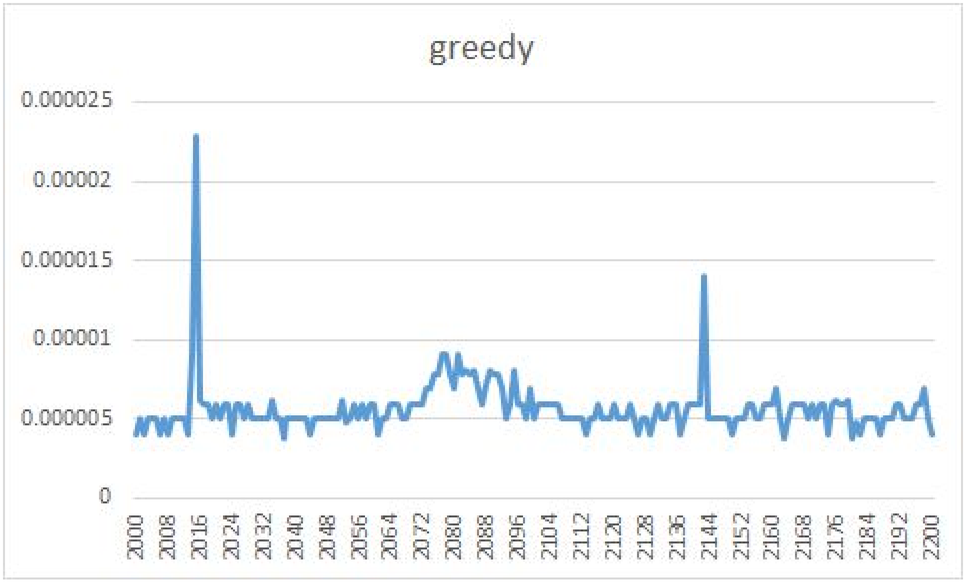
For 1, 2, 6, 12, 24, 48, 60



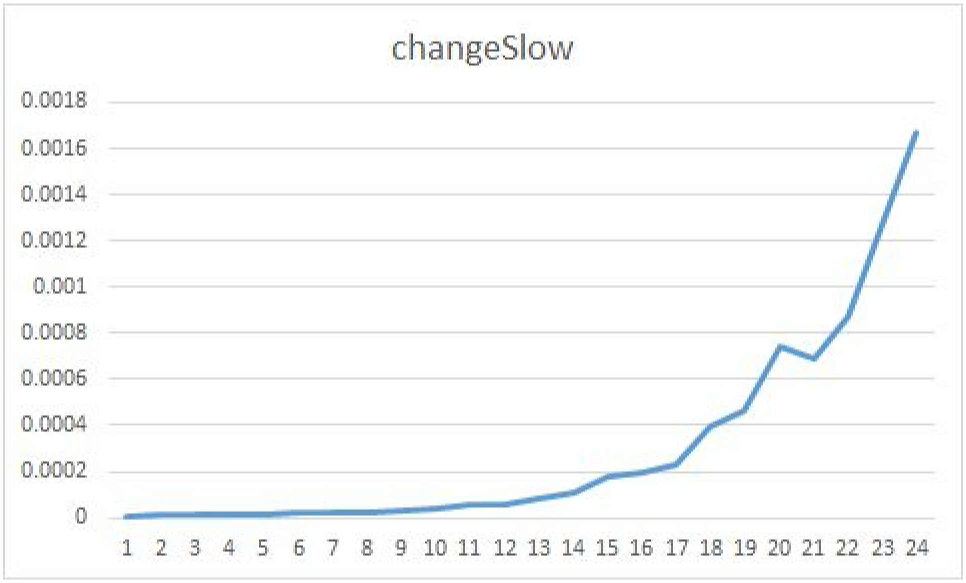
ChangeSlow



For 1, 6, 13, 37, 150

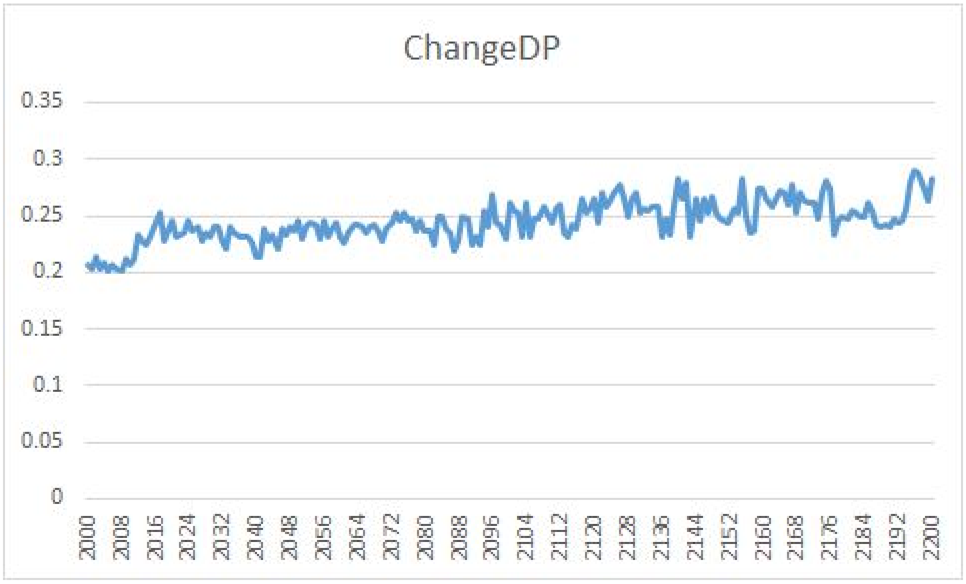


ChangeSlow

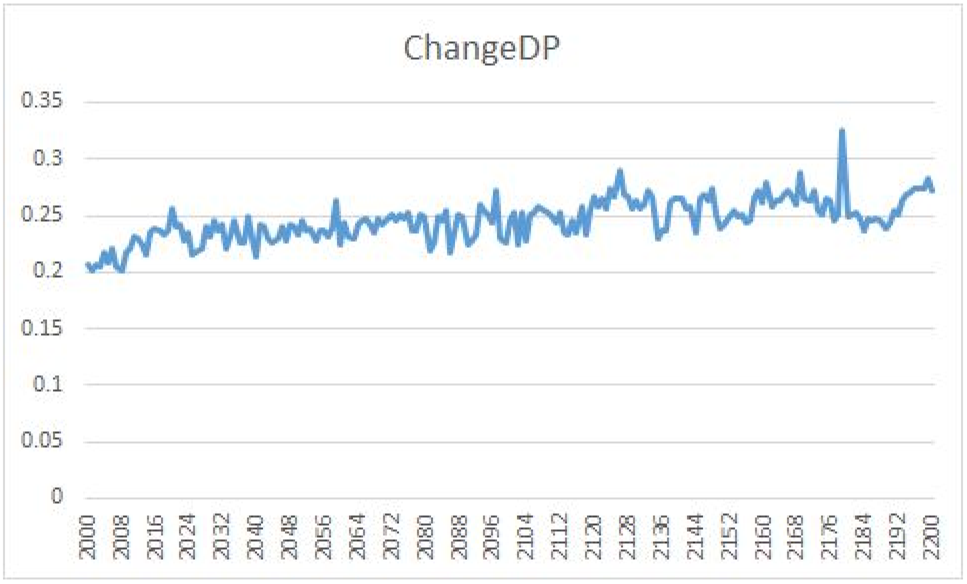


ChangeDP

For 1, 2, 6, 12, 24, 48, 60

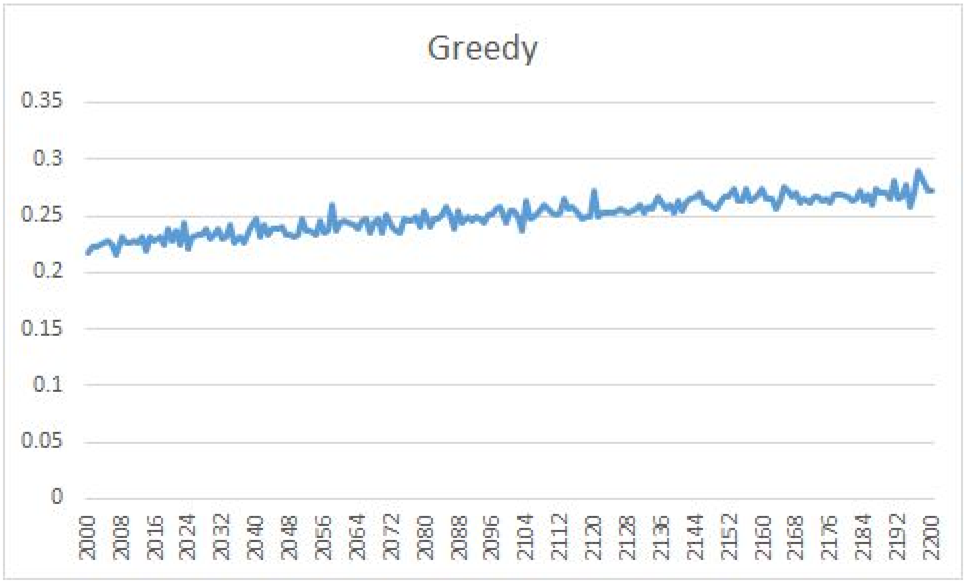


For 1, 6, 13, 37, 150

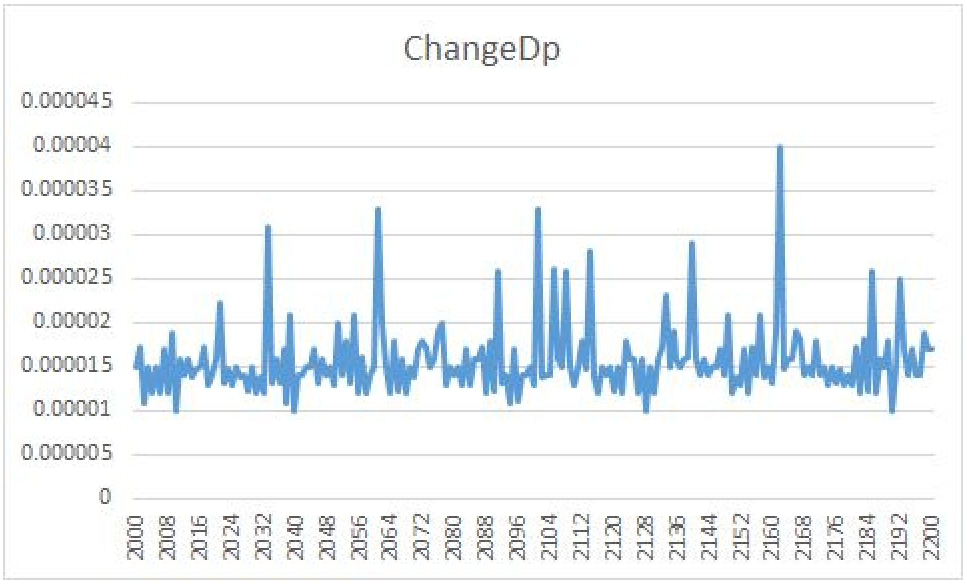


Running times for Question 5 data

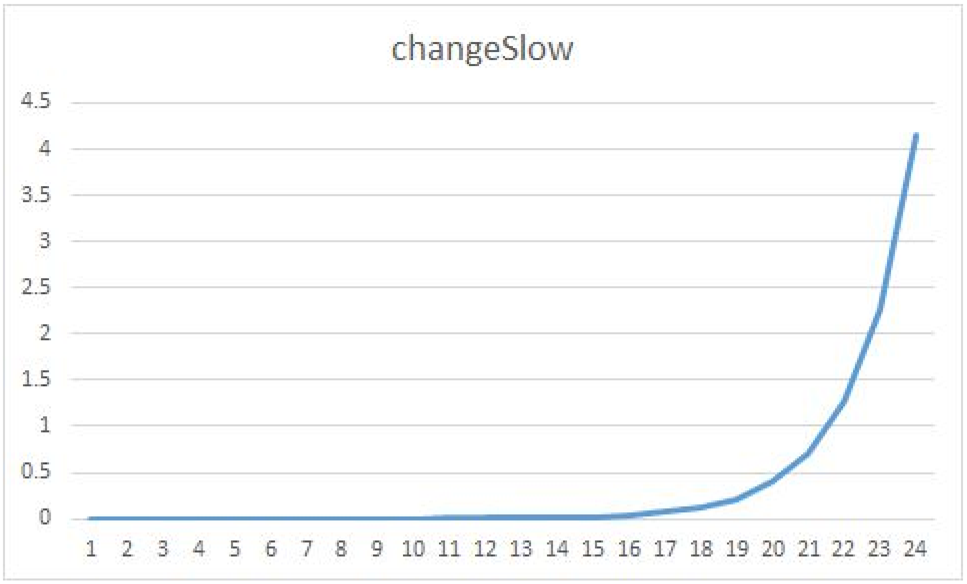
Greedy:



changeDP:



Changeslow where n = 1 - 24

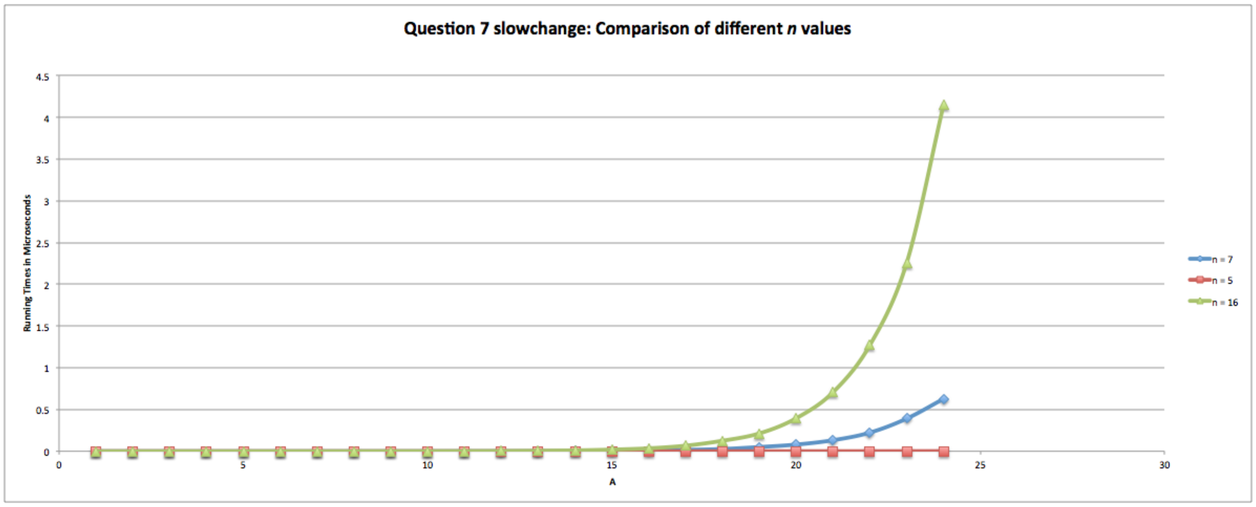


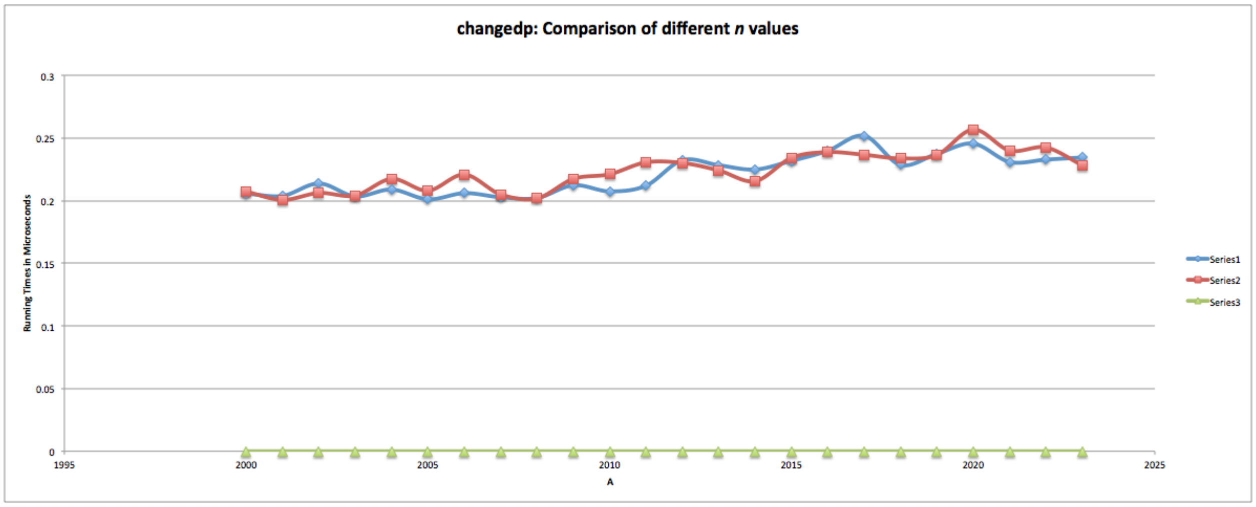
For question 5 data, ChangeDP seems to be quite a bit faster, but for question 3 and 4 data, the greedy algorithm seems to be much faster

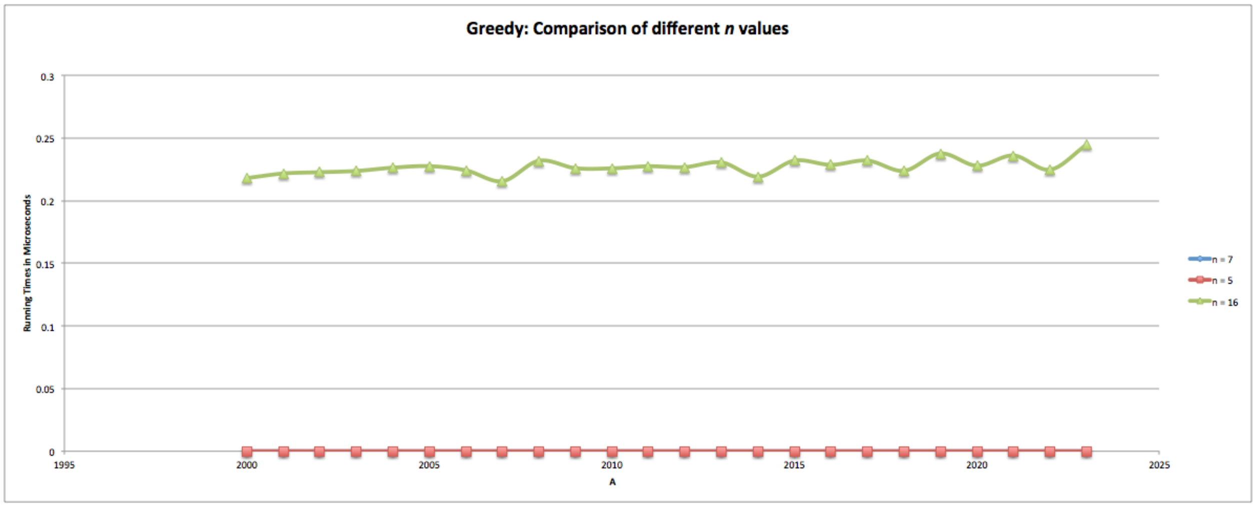
We can also see that with changeSlow it is extremely slow. The reason these runtimes are faster than the runtimes of the other algorithms is because for changeSlow, we had to use a very small value for A (here you see A = 1-24). As you can see, the running time for changeSlow grows exponentially.

Question 7:

For this question we used the data from questions 4-6 to compare how the algorithms ran with the 3 different array sizes (5, 7, and 16).







In that last one, the n=7 line is hiding behind n=5. I'm not sure if these graphs are correct, but I believe that with the increased size of N, the greedy algorithm would increase in running time because it looks at the highest values in the array A. So if you had an amount of 29, but the array of denominations had values like V[1, 5, 8, 23, 56, 78, 345, 200 ...] and even more numbers, it would always go to those high numbers and see if they could be used. This would increase the running time if you added more to V.

Question 8:

8. 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?

I believe greedy would still perform better than dynamic programming, that said, greedy does not always give the optimal solution even though it is faster. I don't think that being powers of p would affect greedy because it is O(n) so it just goes up with the amount of n, not the values in V.