Project 2
Coin Change

Samuel Jones
Ryan Mackey
Thomas Mathew

Part 1: Give a detailed description of the pseudocode for each algorithm.

1.1 Algorithm 1: Greedy

Our greedy algorithm takes two parameters: a vector “v”, which contains the values of the coins to be used, and an int “amount” which is the amount of change to be made. The program then declares a couple of variables as follows:

- a struct, “myChange” to hold the result
- two integers used for temporary purposes: tmpsize, set equal to (vectorsize - 1), and tmpamt, set equal to the input parameter “amount”
- a temporary vector of integers “tmpVect”, and an integer “min”, set equal to 0.

The main work of the algorithm is contained within two nested while loops. The outer loop simply checks if we have reached tmpamt == 0. If we have, then we have finished and the loop exits. Otherwise, we execute the inner while loop, push the results of that loop into our tmpVect, and decrement tmpsize, which keeps track of the coin value we are currently evaluating.

The inner loop executes as long as “tmpamt” is greater than or equal to the element (aka coin) at v[tmpsize]. If that holds, we subtract one “coin” from tmpamt and increment “i”, which is used to count the number of coins we want at the current value. Once tmpamy is less than the current vector element (coin), we decrement tmpsize in the greater loop to check the next (smaller) coin size in the vector.

Once the outer while loop finishes, we use a for loop to push the results into our struct, myChange, set the min variable for myChange to 0, and return myChange.

Pseudo Code

```
changeGreedy(vector<int> coinVector, int amount...){

//Define variables for use here

While (tempAmount > 0){
        While (tempAmount >= v[tempSize]){                    //check coin size. If
                tempAmount = tempAmount  - currentCoinSize;   //small enough, “add”
                coinCountAtThisValue++;                       //coin & incr. counter
        }
tempVector.push_back(numberOfCoinsAtThisValue);
tempSize = tempSize -1;
}

For (j = vectorSize; j >=0; j--){
//Push the values onto the myChange vector
}
Return myChange;
```

1.2 Algorithm 2: Dynamic Programming

For our dynamic programming solution, we started off by creating a table and filled it up looking for any patterns. We used denominations across the top(smallest on left and increasing to the right) and desired change on the left side(smallest on top and increasing doing down). After filling up a decent portion, we noticed that you could use the previous solution in a cell located the number of cells above the current cell equal to the denomination for that column. Then using that solution, you just added one coin of that denomination. If when you went up the columns denomination's worth of cells and you were less than 0, you would use the solution from the column to the left.

Pseudo Code

```
/ m is size of coins array (number of different coins)
int minCoins(int coins[], int m, int V)
{
   // table[i] will be storing the minimum number of coins
   // required for i value.  So table[V] will have result
   int table[V+1];

   // Base case (If given value V is 0)
   table[0] = 0;

   // Initialize all table values as Infinite
   for (int i=1; i<=V; i++)
      table[i] = INT_MAX;

   // Compute minimum coins required for all
   // values from 1 to V
   for (int i=1; i<=V; i++)
   {
      // Go through all coins smaller than i
      for (int j=0; j<m; j++)
        if (coins[j] <= i)
        {
            int sub_res = table[i-coins[j]];
            if (sub_res != INT_MAX && sub_res + 1 < table[i])
                table[i] = sub_res + 1;
        }
   }
   return table[V];
}
```

Part 2: **Calculate the Asymptotic running time for each Algorithm.**

For the greedy Algorithm, the nested while loops (with "n" representing Amount) will have a runtime of n^2, with them both are checking for "tempamt" to fall below a threshold. The following for loop is unrelated to the while loops, so it's runtime is just n. So, the total runtime is: n + n^2.

For the DP Algorithm the running time is the (number of coins * the change), or of nC where C is the change. Because we are filling a 2d table where the runtime for each cell is constant we end up with Θ(nC)

Part 3: **Describe, in words, how you fill in the dynamic programming table in changedp. Justify why is this a valid way to fill the table?**

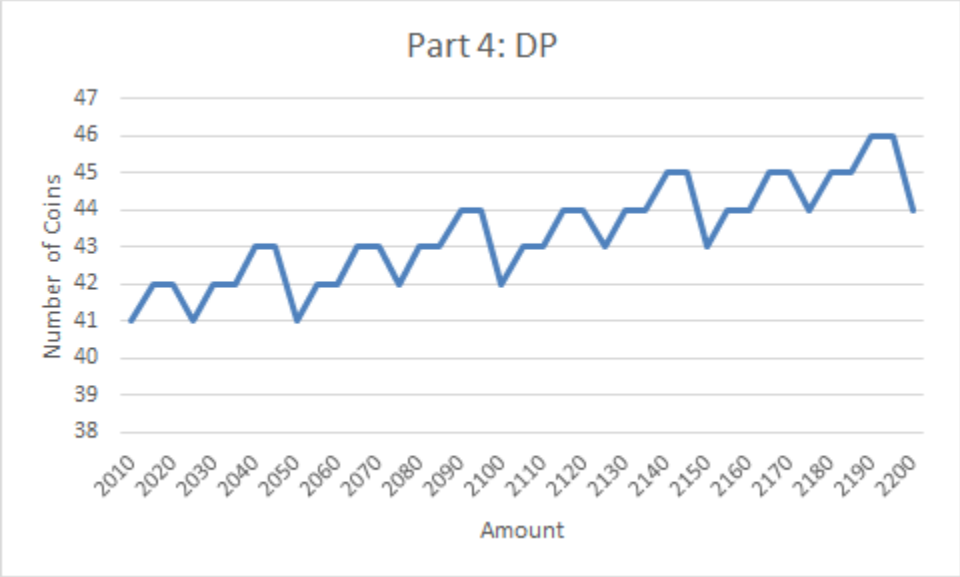
For each coin j, $V_j \leq i$, look at the minimum number of coins found for the $i-V_j$ sum. We don't need to compute because we have already stored the solution to previous subproblems in our table. Let this number be m. If m+1 is less than the minimum number of coins already found for current sum i, then we record the new result.

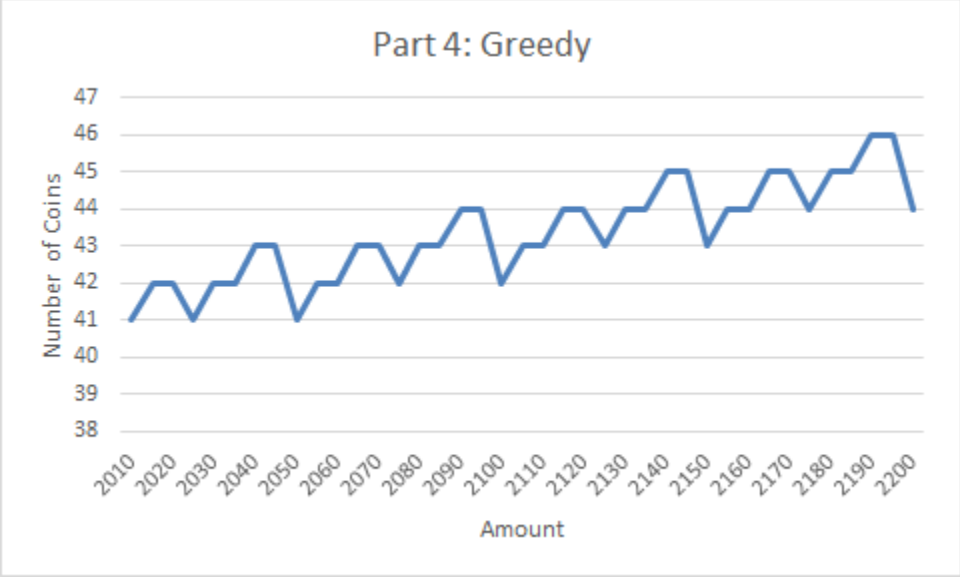
For example, given coins with values 1, 3, and 5 with a desired sum of 11:

We record for state 0 (sum 0) we have find a solution with a minimum number of 0 coins. We then go to sum 1. First, we mark that we haven't yet found a solution for this problem by setting the min to int_max. Then we see that only coin 1 is less than or equal to the current sum. Analyzing it, we see that for sum $1-V_1= 0$ we have a solution with 0 coins. Because we add one coin to this solution, we'll have a solution with 1 coin for sum 1. It's the only solution yet found for this sum. We write (save) it. Then we proceed to the next state – sum 2.

We again see that the only coin which is less or equal to this sum is the first coin, having a value of 1. The optimal solution found for sum (2-1) = 1 is coin 1. This coin 1 plus the first coin will sum up to 2, and thus make a sum of 2 with the help of only 2 coins. This is the best and only solution for sum 2. Now we proceed to sum 3. Because the best solution for sum 2 that we found has 2 coins, the new solution for sum 3 will have 3 coins. Now let's take the second coin with value equal to 3. The sum for which this coin needs to be added to make 3 , is 0. We do the same for sum 4, and get a solution of 2 coins – 1+3. And we complete this pattern until we make it to the sum of 11.

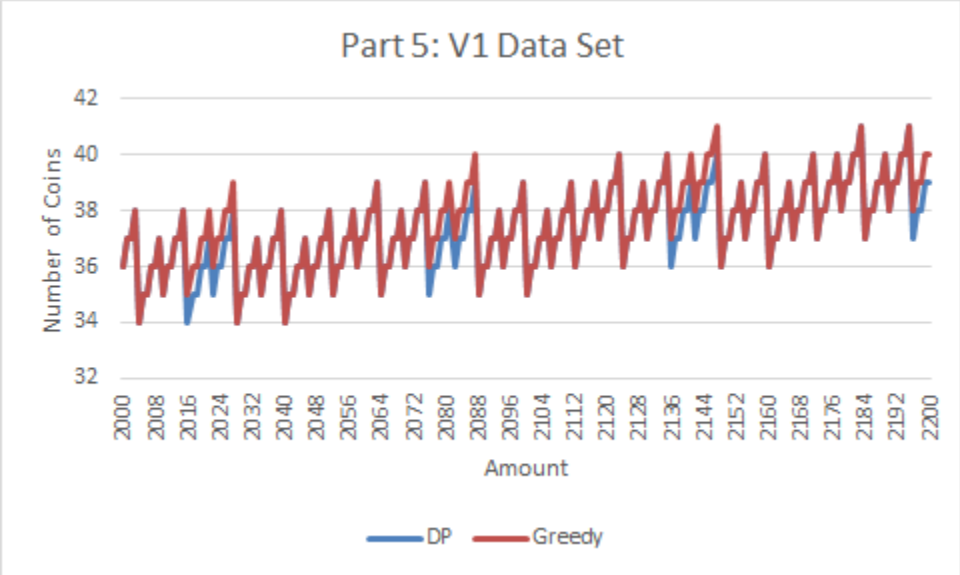
<u>Part 4:</u> Suppose V = [1, 5, 10, 25, 50]. For each integer value of A in [2010, 2015, 2020, …, 2200] determine the number of coins that changegreedy and changedp requires. <u>*Plot the number of coins as a function of A for each algorithm*</u>. How do the approaches compare?

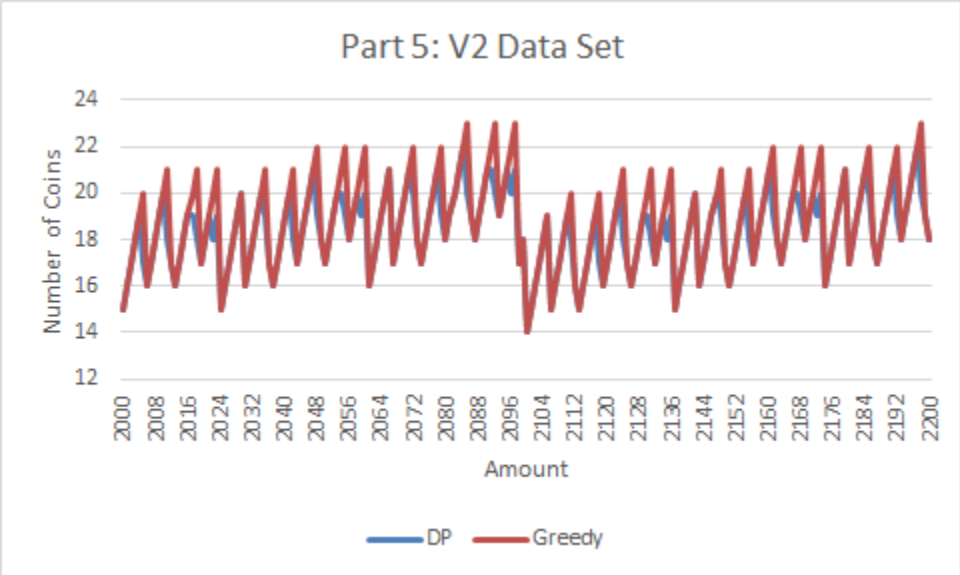




In this case, the performance of the two Algorithms was identical. The coins selected by each algorithm were sometimes different, but in this case the number of coins was always the same.

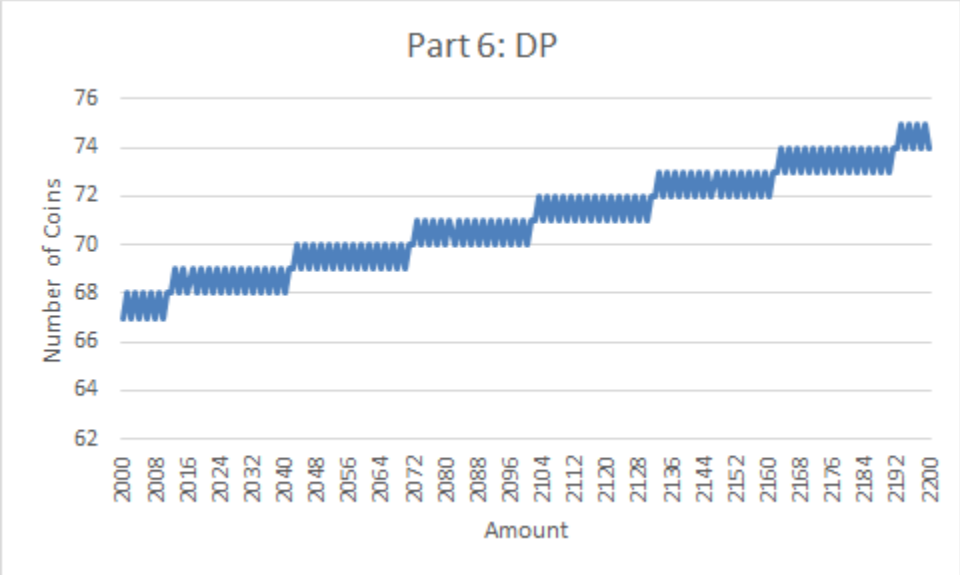
<u>Part 5:</u> Suppose V1 = [1, 2, 6, 12, 24, 48, 60] and V2 = [1, 6, 13, 37, 150]. For each integer value of A in [2000, 2001, 2002, …, 2200] determine the number of coins that changegreedy and changedp requires. If your algorithms run too fast try [10,000, 10,001, 10,003, …, 10,100]. *<u>Plot the number of coins as a function of A for each algorithm</u>*. How do the approaches compare?

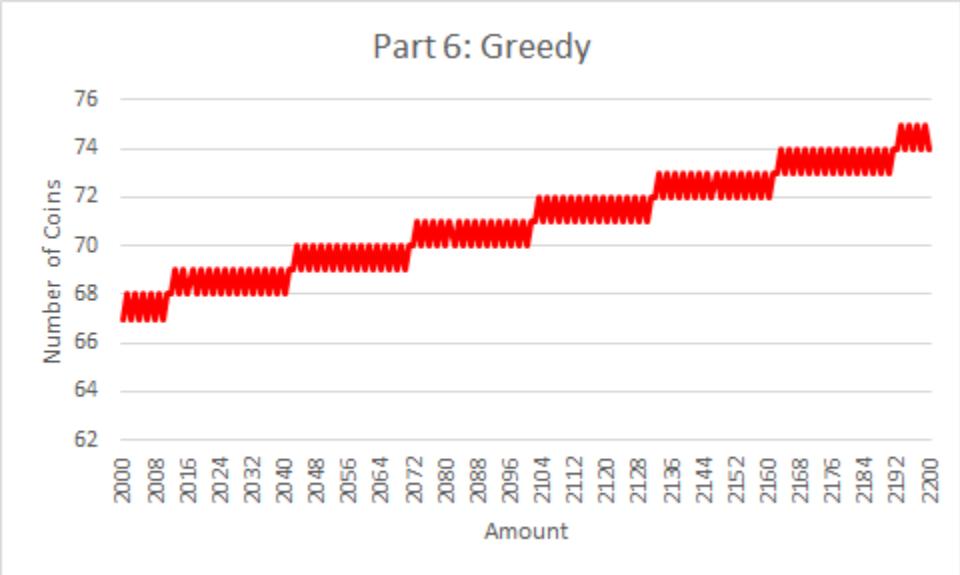




In both cases, DP never takes more coins than Greedy. Sometimes it is superior, and sometimes they are equal in performance.

<u>Part 6:</u> Suppose V = [1, 2, 4, 6, 8, 10, 12, …, 30]. For each integer value of A in [2000, 2001, 2002, …, 2200] determine the number of coins that changegreedy and changedp requires. <u>*Plot the number of coins as a function of A for each algorithm.*</u>

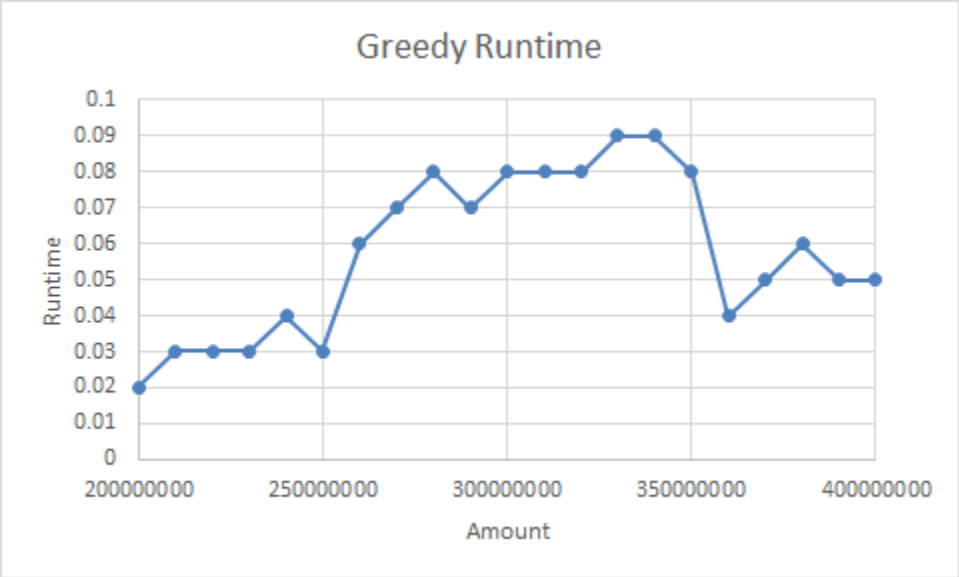


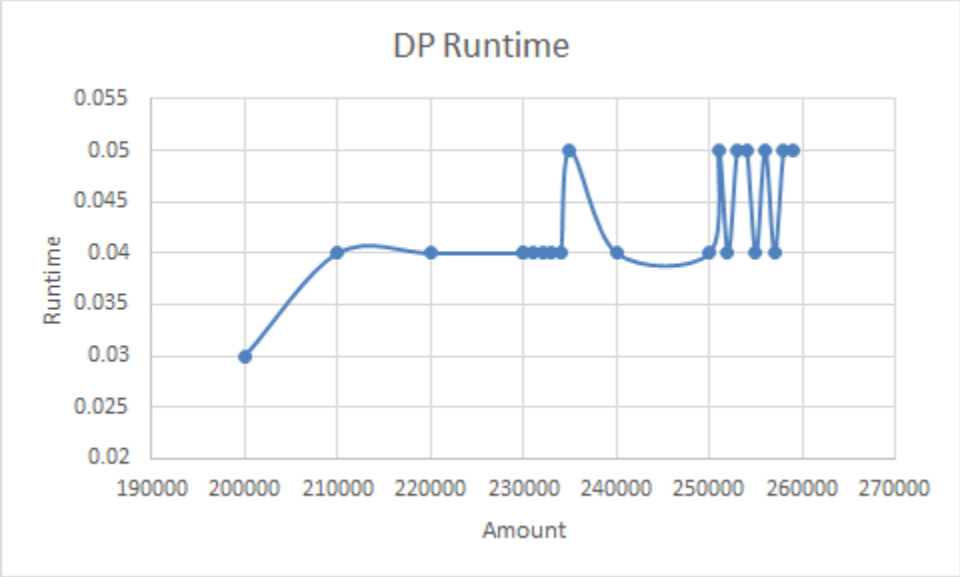


The performance is once again identical here. This is due to the V array that is being used. It includes all multiples of 2 (until 30), and 1. So the biggest “mistake” that the Greedy algorithm can make is being off by 1, which is then fixed by adding a coin of value 1. DP should not and does not have an effect on performance in this case.

<u>Part 7:</u> For the above situations, determine (experimentally) the running times of the algorithms by fitting trend lines to the data or analyzing the log-log plot. <u>*Plot the running time as a function of A.*</u> Compare the running times of the different algorithms.

Here are the results of our runtime experiments:

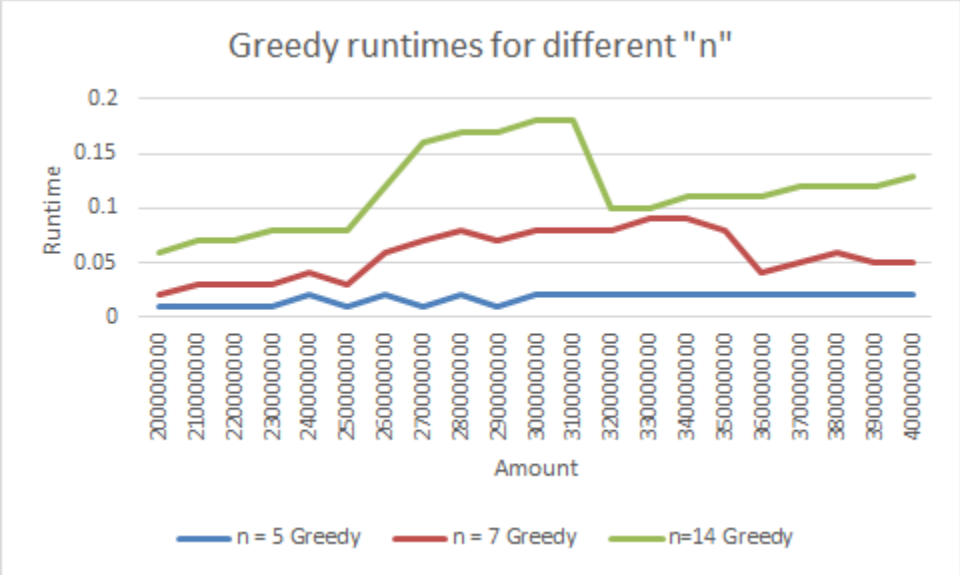


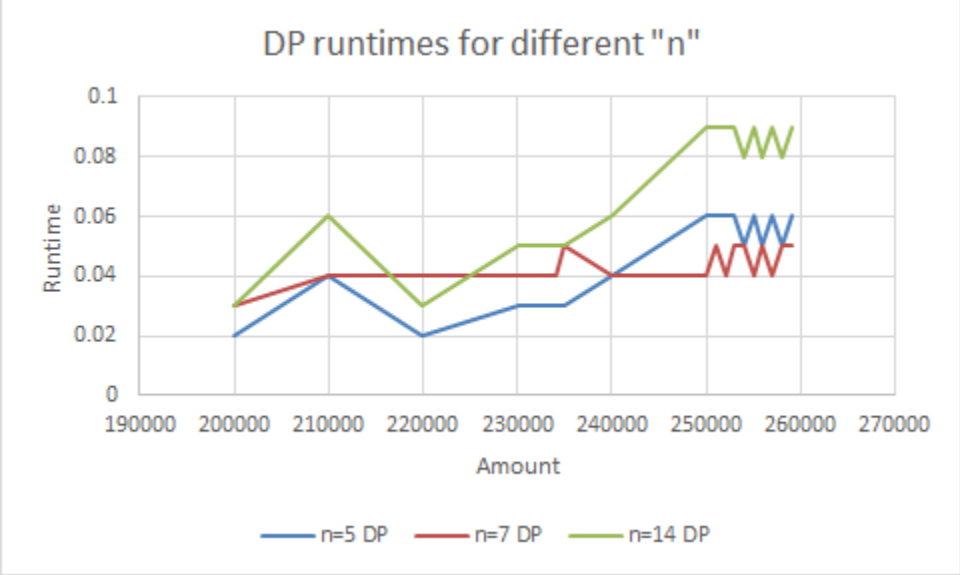


Note the large difference in Amount between the two charts. For Greedy, the scale is 300 million < Amount < 400 million. For DP we have 190,000 < Amount < 270,000.

<u>Part 8:</u> Use the data from questions 4-6 and any new data you have generated. Plot running times as a function of number of denominations (i.e. V=[1, 10, 25, 50] has four different denominations so n=4). Does the size of n influence the running times of any of the algorithms?

Here are our results:





In both cases, the n=14 clearly tends to have a longer runtime for the same Amount when compared to n=5 and n=7. For Greedy, n=7 also clearly has longer runtime than n=5 but for DP this relationship is inconclusive.

Part 9: **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.**

Take the example for V given. If Amount = 35, the greedy algorithm will take 27, leaving 8. Then it will take 3 twice and 1 twice, for 5 coins total. If Amount = 36, it will take 27, leaving 9, then take 9, for 9 coins total. So clearly Amount = 35 was a “bad” case for the greedy algorithm. What could have been done differently with DP? Actually, nothing! If DP took 27 first, the remainder would follow as with the Greedy. If DP tried to take 3 9’s, then two threes, and 2 1’s, the solution is suboptimal. So we can see that the greedy approach is efficient, and perhaps optimal for this type of V[ ].

Part 10: **Under what conditions does the greedy algorithm produce an optimal solution? Explain.**

The optimal solution is one where the lower coins are multiples of the higher coins. This allows for the larger coins to effectively replace the smaller coins. Because the greedy algorithm starts from the largest and works toward the smaller coins, if the smaller coins are unique numbers compared to the large number you will not be able to effectively step down through all the coins. The main issue arise when you have coins close in size that to not continue to decrease.

We will look at the example for part 4. That is based off of [1,5,10,25,50]. We can see how they are all related and continue to decrease in size. From part 4 we can see that the DP and greedy yield the same results, but let's add another coin to the mix a 20 value coin so now we have [1,5,10,20,25,50] and look for change of 40. Without the 20 coin we would get use: 25, 10 and 5. This 3 coin solution is correct for both greedy and DP. Now we include the 20 value and with greedy we get the same 3 coins of 25, 10, and 5. With DP our optimal result is 2 coins of 20 and 20.

The conclusion is that coin sizes need to share similar factors and not be close in value. The US system uses multiples of 5 and a decrease/increase factor of 2. In problem 9, 3 is the multiple and increase decrease factor.