Susan Kuretski
Ava Petley
Robert Sparks
Project Group 2
CS325-401, Fall 2015

Project 2 Report

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

Let n = the coin amount and k = the coin denomination values. Begin by making change for n = 1.

If the n = 0, then this would hit the base case, and the coins required would be zero. So for each value of n = 0, k would also be zero. The table can be initialized representing as such. If k > n, then k = 0. For example, if you have amount of 1 and coin denominations of 2, 5, 10, it is not possible to have change.

Progress with each value of n i.e. n = 2, n = 3...until n = amount. At each increment of n, evaluate the minimum of (n - k) +1 and n. Whichever is the minimum, that is the minimum number of coins for that n and k. Record that number in the table. Continue this method through n and k. While progressing through the table, previous calculations can be referenced. This is a valid way to fill in the table since:

For a n > 0, let v = to the coin denomination values. Suppose $v_1 = 1$, $v_2 = 3$, and $v_3 = 5$ and n = 10.

$C(n) = \min \{C(n - v_i)\} + 1$

$C(1) = \min\{C(1 - 1)\} + 1 = 1$

$C(2) = \min\{C(2-1)\} + 1 = \min\{C(1)\} + 1 = 2$

$C(3) = \min\{C(3-1), C(3-3)\} + 1 = \min\{C(2), C(0)\} + 1 = 1$

$C(4) = \min\{C(4-1), C(4-3)\} + 1 = \min\{C(3), C(1)\} + 1 = 2$

$C(5) = \min\{C(5-1), C(5-3), C(5-5)\} + 1 = \min\{C(4), C(2), C(0)\} + 1 = 1$

… and so on.

For each calculation, we can refer back to the value of C(n). For example, to find $C(4) = \min\{C(3),C(1)\} + 1$, we can look at the value from C(3) and C(1) instead of recalculating it.

Filling in the table would be considered a bottom-up approach since we start by calculating the smallest values for C(n) up to n.

2. Give pseudocode for each algorithm.

Algorithm 1: Recursive/Brute Force

```
changeslow(A, n)
      if n = 0
            return 1
      minCoins = ∞
      if n < 0
            return 0
      for i = 0 to A.length
            if(n >= A[i])
                  minCoins = min(changeslow(N - A[i] + 1, minCoins))
      return minCoins
```

There are 3 base cases:

1. n = 0. This would occur is there was zero amount to get change. Naturally the change would be zero as well.
2. n < 0. There is a negative amount of money. In this case, no change could be made.
3. n >= 1 and A[i] <= 0. There is an amount of money, but there is no change available.

Time complexity is $\Theta(2^n)$. Given input n is a positive integer, this algorithm will go through all possible coin combinations (inclusive and exclusive) without regard to set calculation duplication.

Algorithm 2: Greedy Algorithm

Pre-condition: The array is sorted in ascending order and has at least one element, and the first element is not zero.

```
changegreedy(A, n)
      m = 0
      for i = A.size -1 downto 0
            numOf = ⌊n/A[i]⌋
            n = n mod A[i]
            m += numOf
      return m
```

Time complexity is O(n) since this algorithm iterates through the length of the array.

Algorithm 3: Dynamic Programming

Pre-condition: The array is sorted in ascending order and has at least one element.

```
changedp(A, k, n)
     C[0] ← 0
     for p ← 1 to n
          min ← ∞
          for i ← 1 to k
               if A[i] ≤ p
                    if 1 + C[p − A[i]] < min
                         min ← 1 + C[p − A[i]]
                         coin ← i
          C[p] ← min
          S[p] ← coin
     return C, S
```

The time complexity is Θ(nk) since we are iterating through n and then also iterating through k.

3. Prove that the dynamic programming approach is correct by induction. That is, prove that

$T[v] = \min_{v[i] <= v}\{T[v - V[i]] + 1\}$, T[0] = 0 is the minimum number of coins possible to make change for value v.

Define subproblem:

Let T[v] be the minimum number of coins (optimal solution) from the denominations of T_1, T_2,...,T_i to make change for v amount.

Show Optimal Substructure (Overlapping Subproblems)

Suppose for v amount there are coin denominations of T_1, T_2, T_3,...,T_i and suppose we break this solution into two halves. The right half's solution is the optimal amount to make change for v-b amount using coin denominations of T_1, T_2, T_3,..., T_i. Subsequently, the left half's solution is the optimal amount to make change for b amount using coin denominations of T_1, T_2, T_3,..., T_i.

To prove this by contradiction, suppose there is a better solution to make change from v-b amount from coins of T_1, T_2, T_3,..., T_i than the right half's solution. Substitute this solution for the previous right half's solution. However, this cannot be true since this would cause the entire solution to be false.

For example, suppose v = 5 and $T_1 = 1$, $T_2 = 3$, $T_3 = 5$.

T[0] = 0

T[1] = min{T[1-1]} + 1 = min{T[0]} + 1 = 1

T[2] = min{T[2-1]} + 1 = min{T[1]} + 1 = 2

T[3] = min{T[3-1],T[3-3]} + 1 = min{T[2],T[0]} + 1 = 1

T[4] = min{T[4-1], T[4-3]} + 1 = min{T[3],T[1]} + 1 = 2

T[5] = min{T[5-1], T[5-3], T[5-5]} + 1 = min{T[4],T[2], T[0]} + 1 = 1

If a subproblem of T[v] had a better solution, then it could cause subsequent subproblems to be false since each subsequent subproblem overlaps with previous subproblems.

Write a recurrence:

$T[v] = T[0] = 0$ if $v = 0$

$T[v] = \min_{V[i] <= v}\{1 + T[v - V_i]\}$ if $v > 0$

State your base cases:

The base case occurs if v = 0, in which making change for 0 amount, the optimal solution would be 0 coins or the empty set. Also, it should be noted if V is an empty set, then there are zero ways to make change.

Prove the recurrence:

Base case: T[0] = 0. For zero amount, there are zero coins in the optimal solution. True. V (coin denominations set) is empty. In this case, there are no coins available to make change. This is also true.

Induction: For the optimal solution, there must exist some initial coin V_i where $V_i <= v$. In addition, from showing the optimal substructure, the subsequent coins in the optimal solution must also be in the optimal solution for $v - V_i$ amount. If V_i is the first coin in the optimal solution for making change for v amount, then one V_i coin + $T[v - V_i]$ coin(s) optimally make change for $v- V_i$ amount. Since the optimal solution in this problem requires the smallest amount of coins needed, the minimum is taken.

Present the algorithm/Prove the algorithm evaluates the recurrence:

```
changedp(A, k, n)
      C[0] ← 0
      for p ← 1 to n
            min ← ∞
            for i ← 1 to k
                  if A[i] ≤ p
                        if 1 + C[p − A[i]] < min
                              min ← 1 + C[p − A[i]]
                              coin ← i
            C[p] ← min
            S[p] ← coin
      return C, S
```

C[0] ← 0 is the base case.

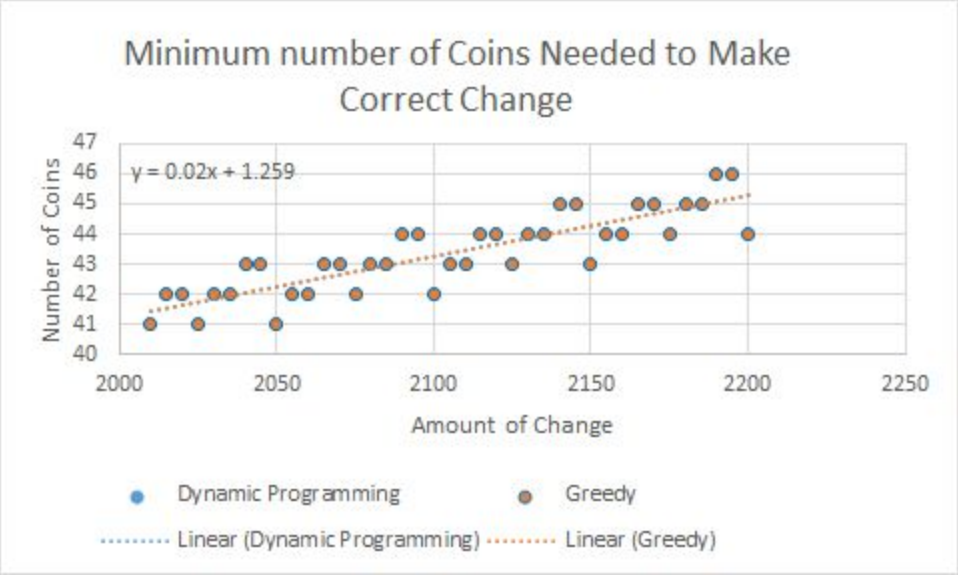
`for p ← 1 to n` shows the algorithm is evaluating for each subproblem, starting at 1 and incrementing to n amount. p does not start at zero since we have covered our base in the previous statement.
`min ← ∞` is our sentinel value.
`for i ← 1 to k` shows the algorithm is evaluating for each index of the coin denominations. This for-loop begins at 1 since our base case also occurs if the coin denominations set is an empty set i.e. there must be at least one index in A. The following if-statements will then execute the $\{\min_{V[i] <= v}\{1 + T[v - V_i]\}$ of our recurrence. From the bottom-up, this algorithm considers the base cases and executes $\{\min_{V[i] <= v}\{1 + T[v - V_i]\}$ from 1 to n (the initial coin amount) and from 1 to k (the number of coin denominations).

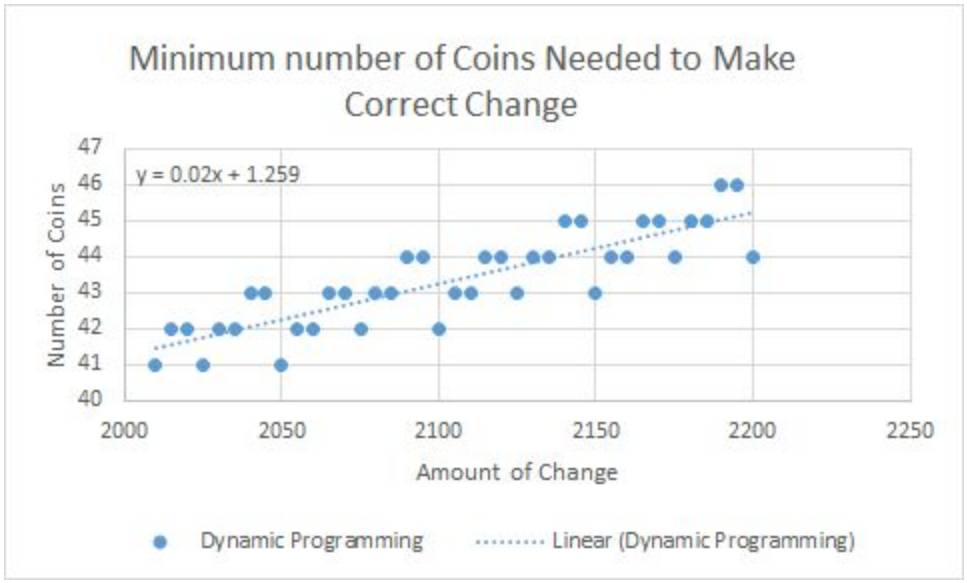
Prove the algorithm is correct:

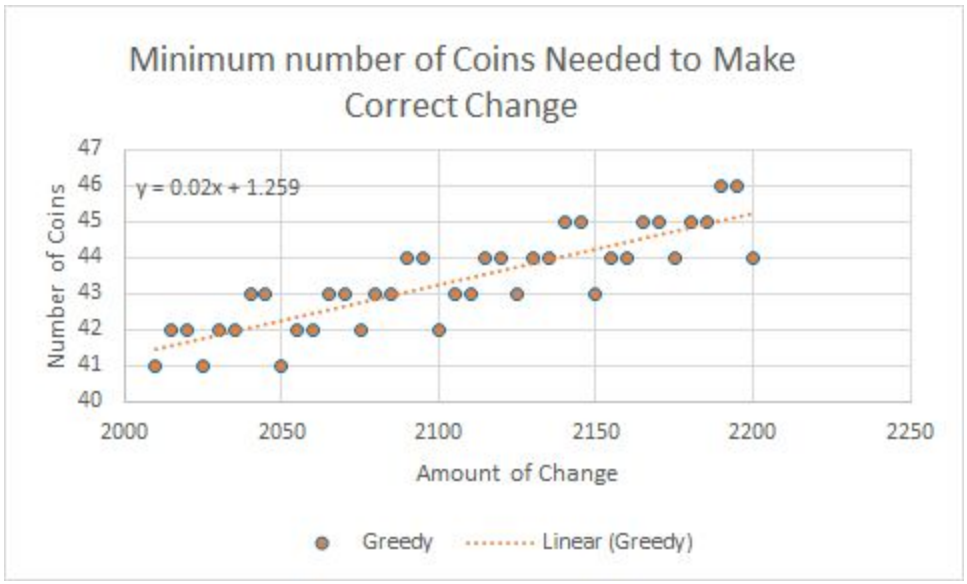
This algorithm is correct if it returns the correct C and S, where C is the minimum number of coins required to make change for n, and S is the index of the first coin in the optimal solution. From the previous section, we can see that the algorithm correctly represents the recurrence if 0 <= p <= n, and will terminate at p = n and i = k.

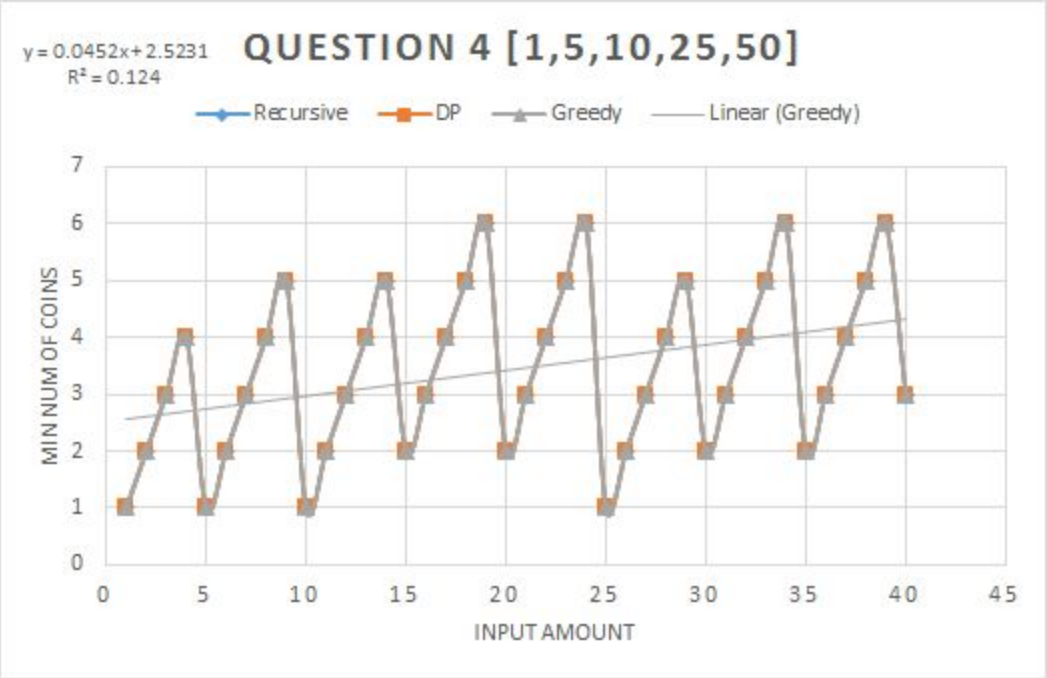
4. Suppose V = [1, 5, 10, 25, 50]. For each integer value of A in [2010, 2015, 2020, …, 2200] determine the number of coins that changegreedy and changedp requires. You can attempt to run changeslow however if it takes too long you can select smaller values of A and also run the other algorithms on the values. Plot the number of coins as a function of A for each algorithm. How do the approaches compare?

All of the data points are the same so greedy covers up dynamic.



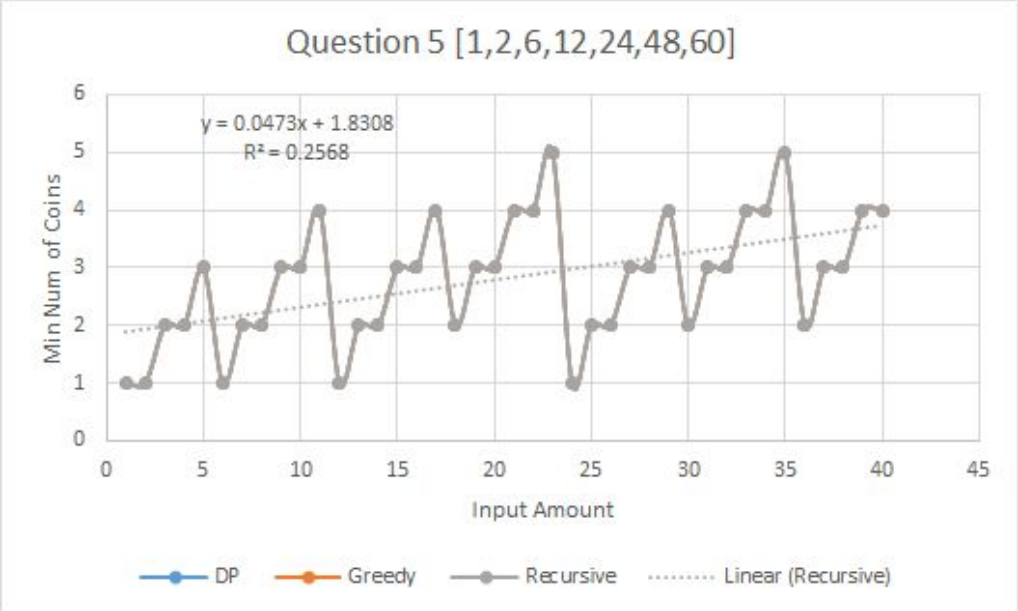

Minimum number of Coins Needed to Make Correct Change
y = 0.02x + 1.259
Number of Coins
47
46
45
44
43
42
41
40
2000
2050
2100
2150
2200
2250
Amount of Change
Dynamic Programming
Linear (Dynamic Programming)

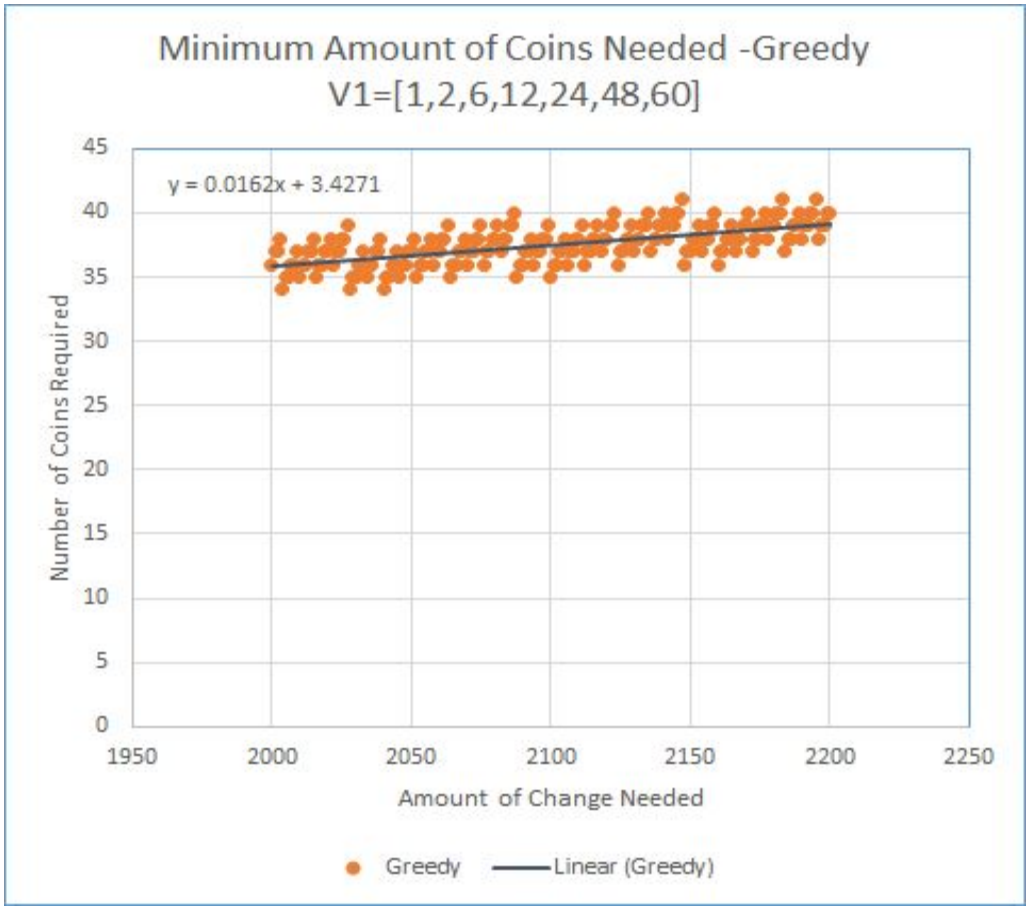

Minimum number of Coins Needed to Make Correct Change
y = 0.02x + 1.259
Number of Coins
47
46
45
44
43
42
41
40
2000
2050
2100
2150
2200
2250
Amount of Change
Greedy
Linear (Greedy)

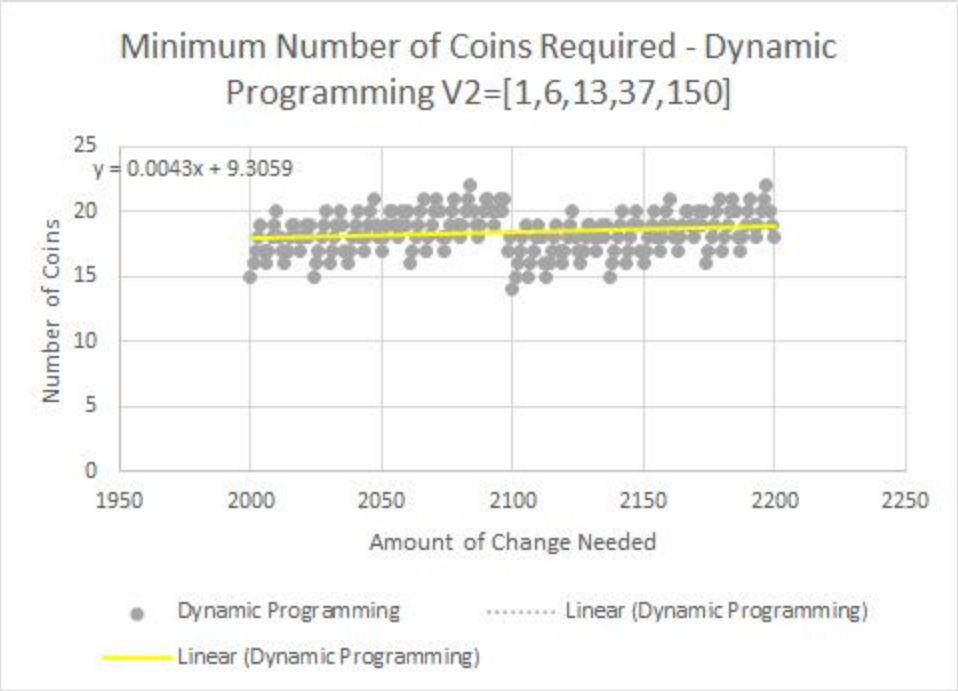


It appears that each algorithm for the coin denominations of [1,5,10,25,50] result in the same minimum number of coins. The graphs indicate for each input amount the minimum numbers of coins was the same since their linear equations were exactly the same.

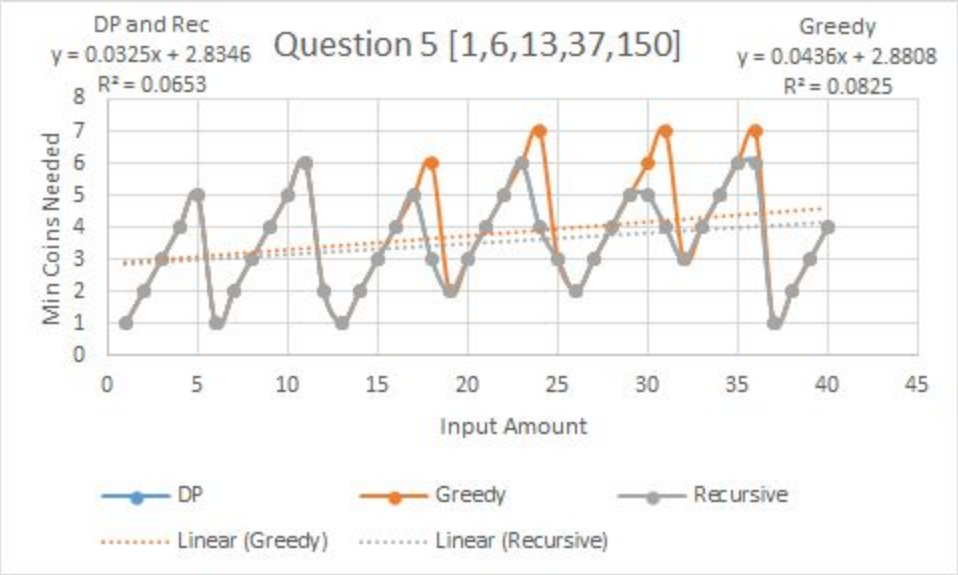
5. Suppose V1 = [1, 2, 6, 12, 24, 48, 60] and V2 = [1, 6, 13, 37, 150]. For each integer value of A in [2000, 2001, 2002, …, 2200] determine the number of coins that changegreedy and changedp requires. If your algorithms run too fast try [10,000, 10,001, 10,003, …, 10,100]. You can attempt to run changeslow however if it takes too long you can select smaller values of A and also run all three algorithms on the values. Plot the number of coins as a function of A for each algorithm. How do the approaches compare?


Question 5 [1,2,6,12,24,48,60]
y = 0.0473x + 1.8308
R² = 0.2568
Min Num of Coins
Input Amount
DP
Greedy
Recursive
Linear (Recursive)


Minimum Amount of Coins Needed -Greedy
V1=[1,2,6,12,24,48,60]
y = 0.0162x + 3.4271
Number of Coins Required
Amount of Change Needed
Greedy
Linear (Greedy)

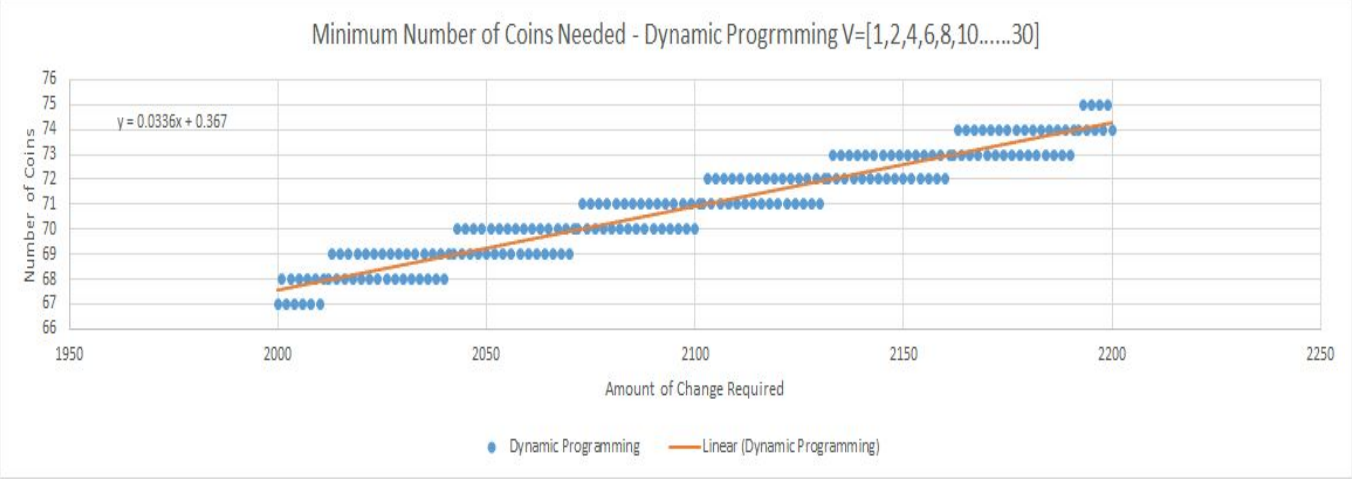

Minimum Number of Coins Required - Dynamic Programming V2=[1,6,13,37,150]
y = 0.0043x + 9.3059
Number of Coins
Amount of Change Needed
Dynamic Programming
Linear (Dynamic Programming)
Linear (Dynamic Programming)

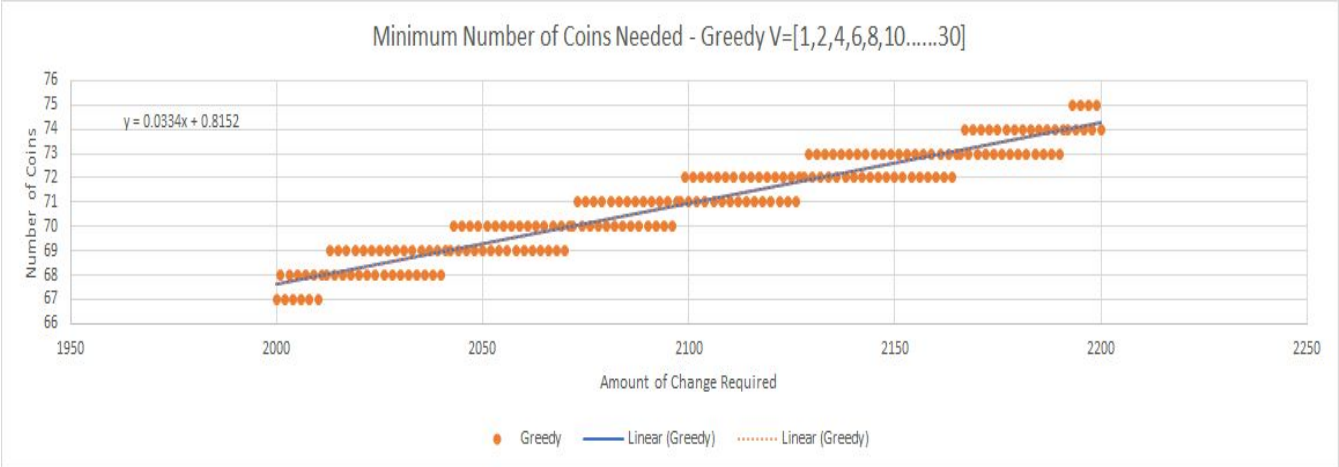

Minimum Number of Coins Required - Greedy V2=[1,6,13,37,150]
y = 0.0043x + 9.8797
NUmber of Coins
Amound of Change Needed
Greedy
Linear (Greedy)
Linear (Greedy)

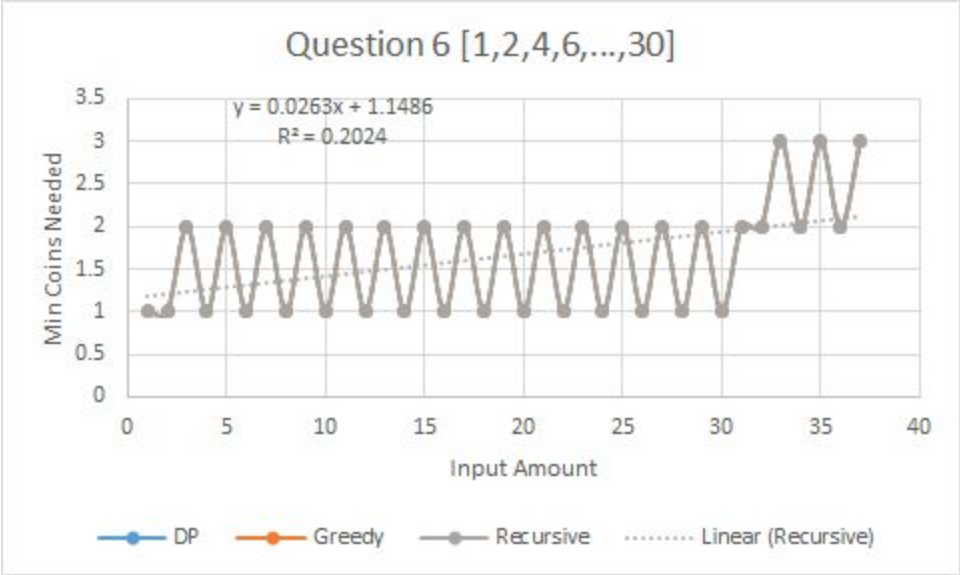


It appears the greedy algorithm does not always result in the same minimum number of coins. In fact, the greedy algorithm returns a higher number of minimum coins in some cases in regards to the dynamic programming and recursive algorithms. This set of coins illustrates the non-optimal results this greedy algorithm can produce.

6. 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. You can attempt to run changeslow however if it takes too long you can select smaller values of A and also run all three algorithms on the values. Plot the number of coins as a function of A for each algorithm.

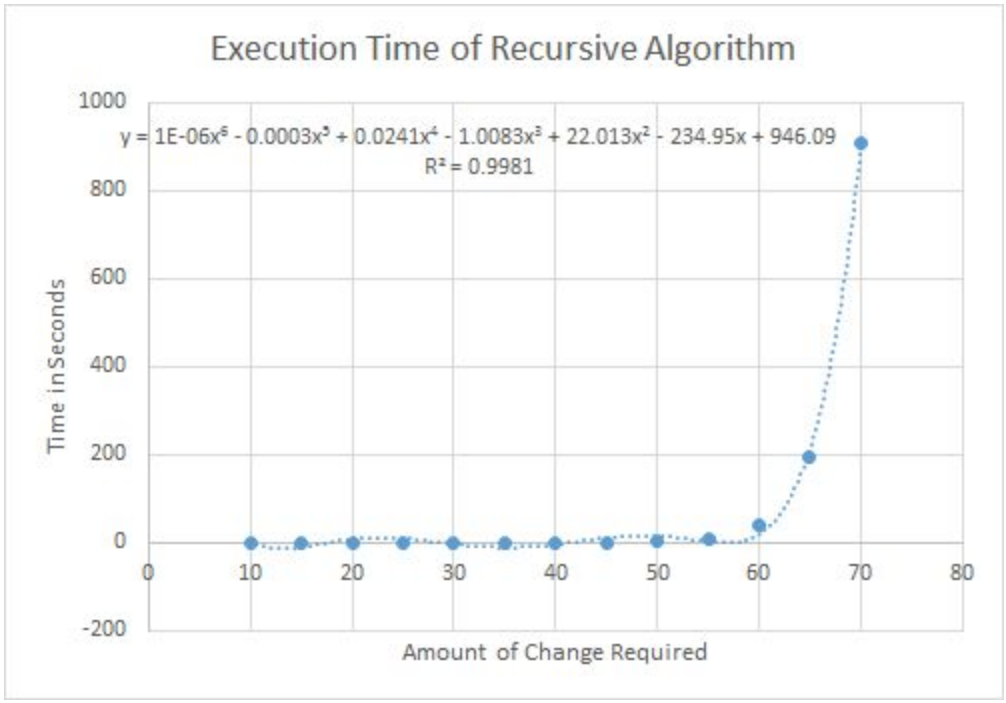


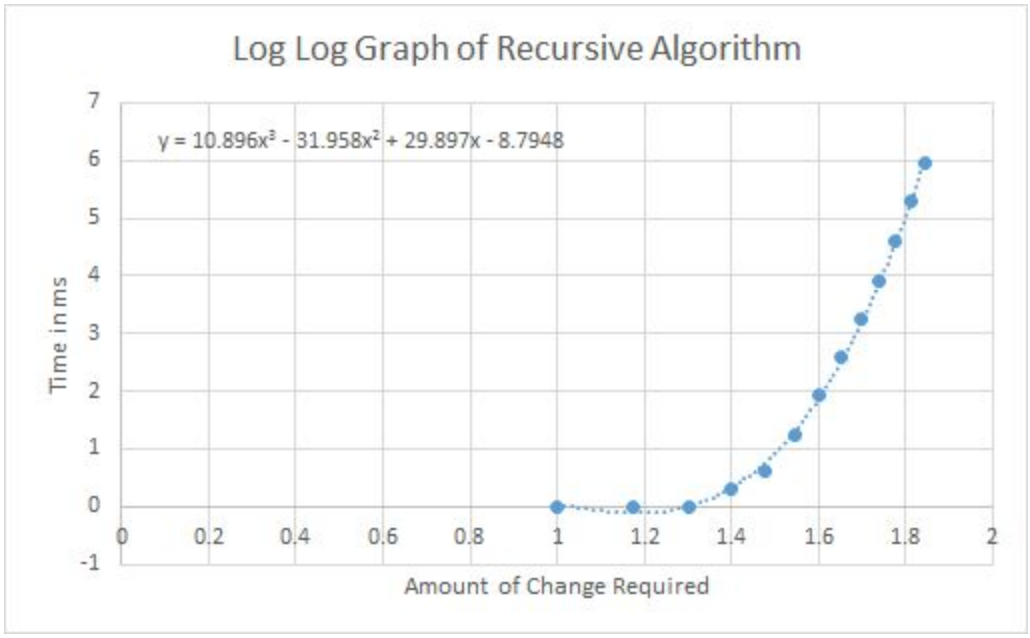


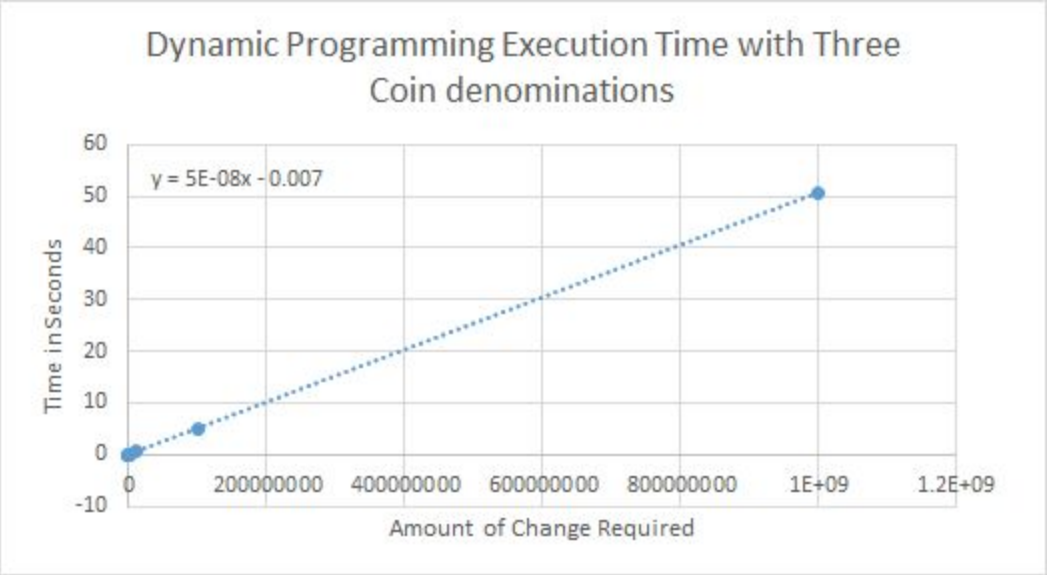


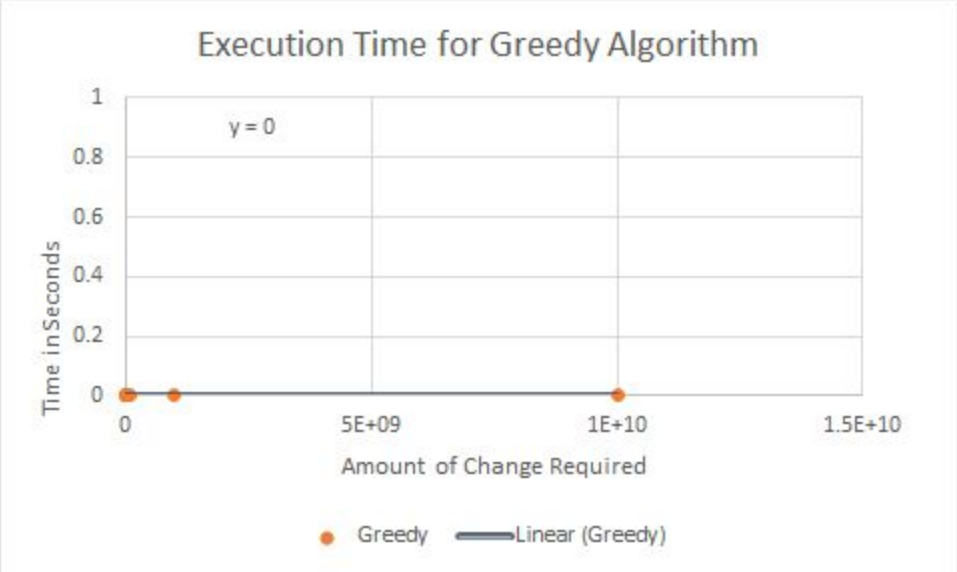
Here, the difference in the linear equations between the greedy algorithm and the dynamic programming indicates the difference between the two. With a steeper slope line, the greedy algorithm shows us that it does not show the minimum number of coins at all times. Also within the testing limits, it appears the dynamic and recursive algorithms produce the same results.

7. 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. Graph the running time as a function of A. Compare the running times of the different algorithms.


Execution Time of Recursive Algorithm
y = 1E-06x⁶ - 0.0003x⁵ + 0.0241x⁴ - 1.0083x³ + 22.013x² - 234.95x + 946.09
R² = 0.9981
Time in Seconds
Amount of Change Required
1000
800
600
400
200
0
-200
0
10
20
30
40
50
60
70
80


Log Log Graph of Recursive Algorithm
y = 10.896x³ - 31.958x² + 29.897x - 8.7948
Time in ms
Amount of Change Required
7
6
5
4
3
2
1
0
-1
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2



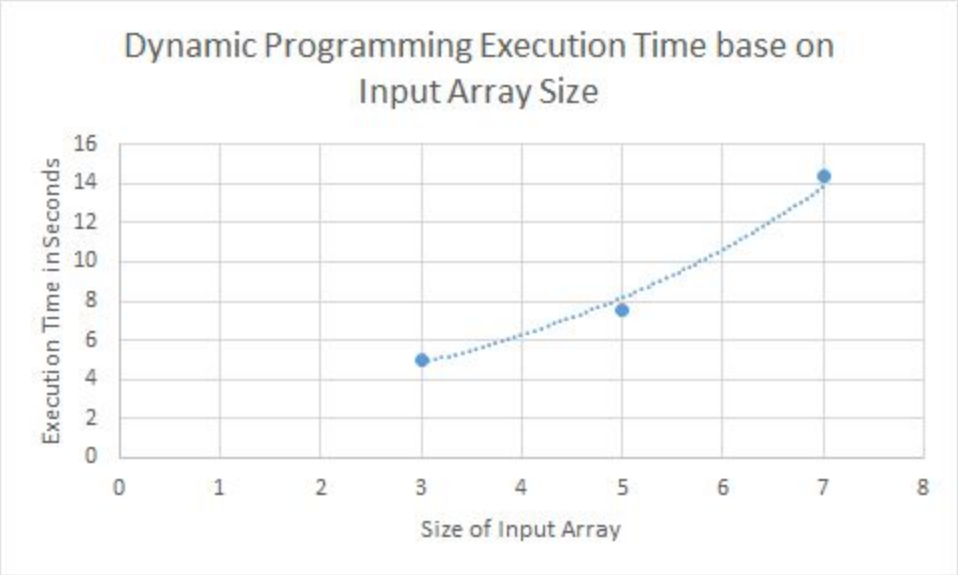


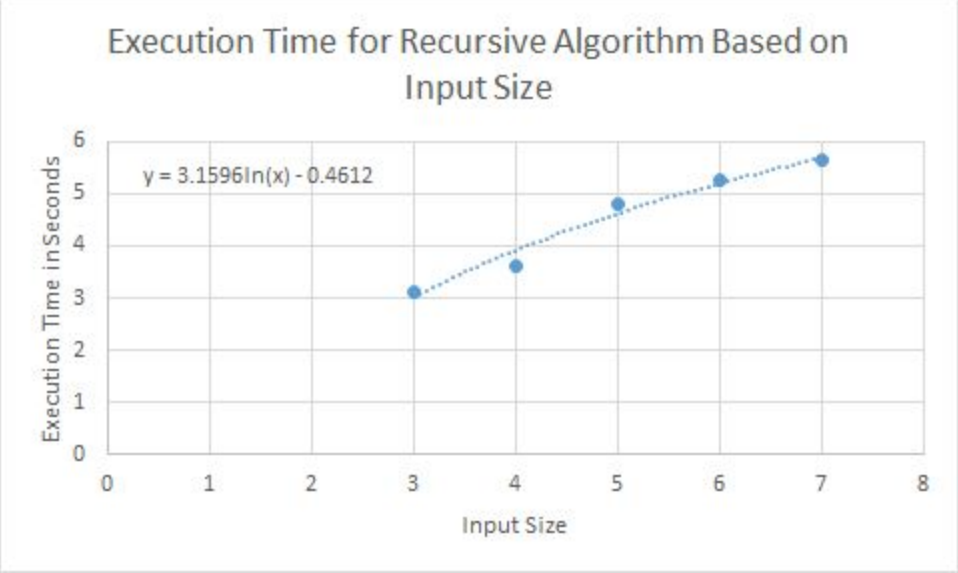
For the greedy algorithm: Since we used an implementation that involved dividing the initial amount repeatedly, execution time could not be measured since the time was constant (too fast for the timing clock). For each starting amount of coins, it would be divided by the same number of coins in the coin array. Hence, no matter the size of the input, the number of executions will be the same.

So in terms of running times, the slowest is recursive at exponential time. Then, the dynamic programming at linear time. Finally, the fastest is the greedy algorithm due to its constant time as discussed above.

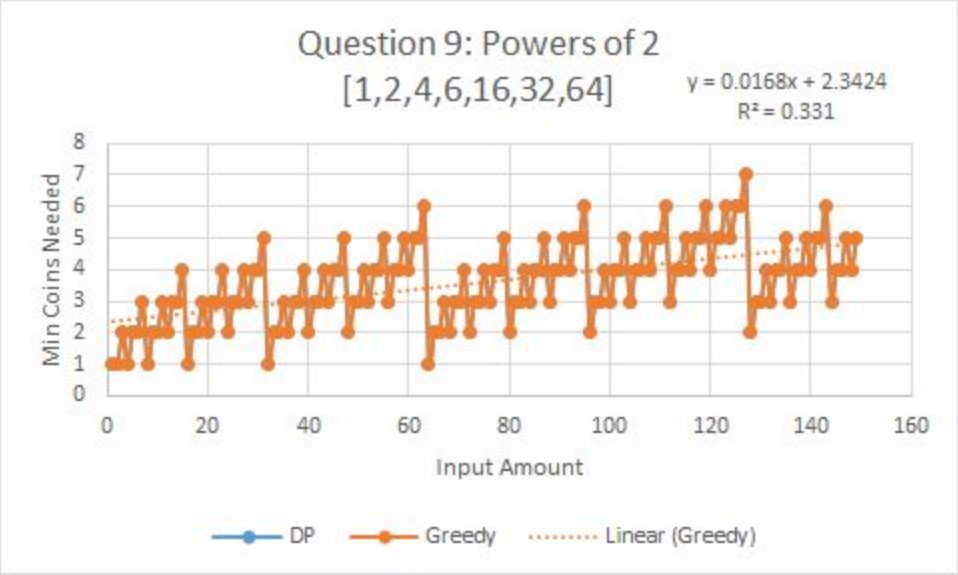
8. Use the data from questions 4-6 and any new data you have generated. Plot running times as a function of number of denominations (i.e. V=[1, 10, 25, 50] has four different denominations so n=4). Does the size of n influence the running times of any of the algorithms?

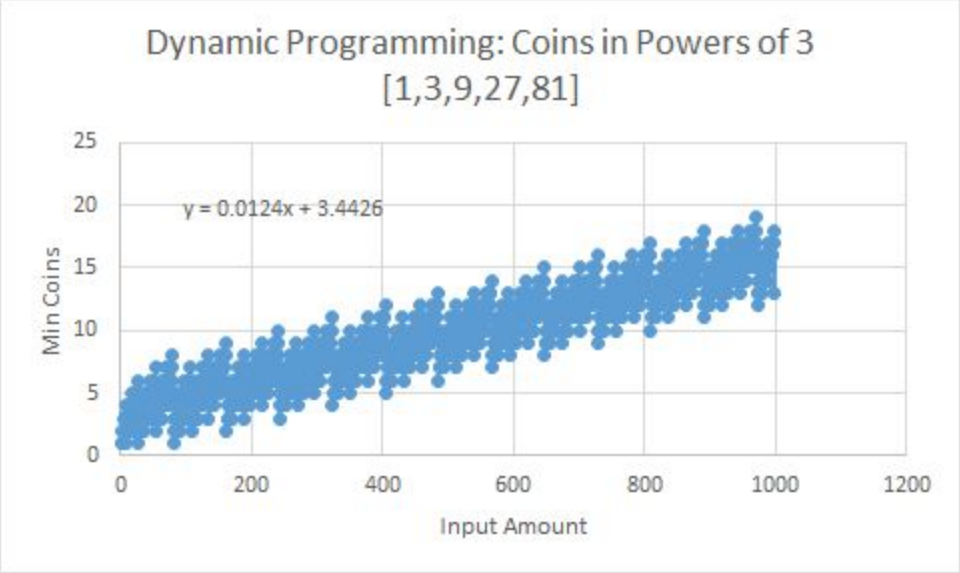
Yes, size influences the running times for the dynamic programming algorithm as well as the recursive. As you can see from the graphs, as the input size increases so does the execution time. The dynamic programming algorithm has a sort of exponential curve whereas the recursive seems to be logarithmic. We were unable to measure differences with the greedy algorithm since it was too fast for us to measure.

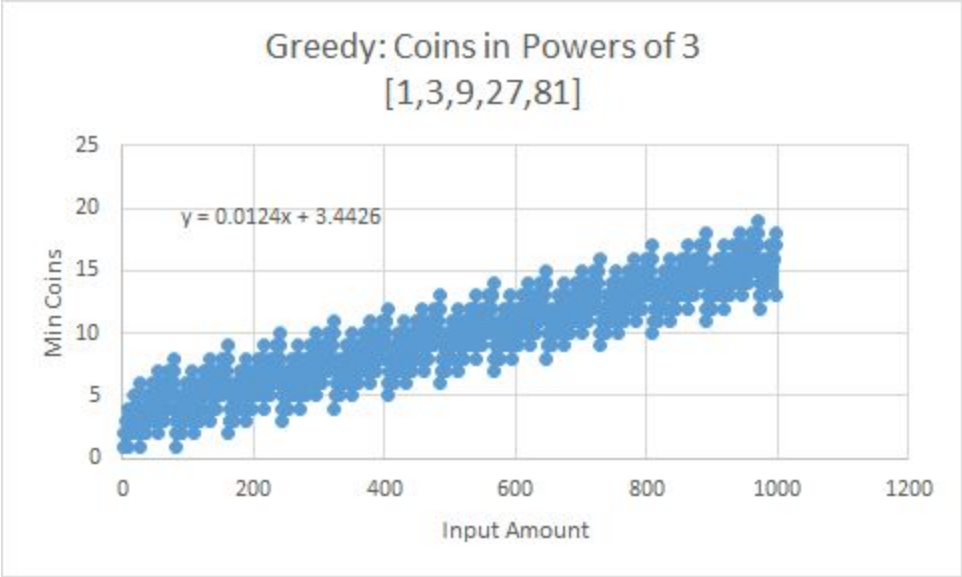




9. Suppose you are living in a country where coins have values that are powers of p, V = [p0 , p1 , p2 , ⋯ , pn]. How do you think the dynamic programming and greedy approaches would compare? Explain.







The greedy and dynamic programming approach appear to have the same results. This was also run with values of 243 and 729 in the powers of 3 set. The two had the exact same results and trendline. The reason for this is because these sets of numbers are canonical. In a sense, each number is a different representation of the same object. In our greedy implementation, the initial amount is divided by the first coin denomination value, and then repeatedly gets divided, so we're essentially dividing by the same type of object with different representations.