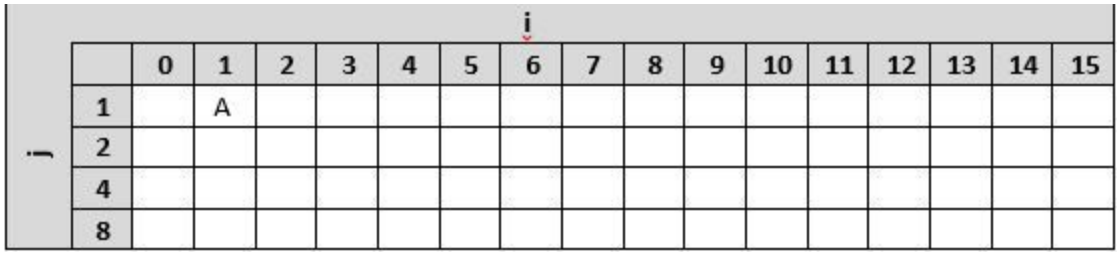
CS 325 Project 2 Report (Group 2)

Project Members:

- Matthew Walz
- Frank Eslami
- Kevin To

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



j \ i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1		A														
2																
4																
8																

1. For visualization, please note the table above that the code eventually creates. The goal is to make change for the amount 15 given coin denominations of {1, 2, 4, 8}.
2. The horizontal axis, i, breaks down the main problem of making change for 15 to its smallest subproblem 0, which is the base case. Then it iterates through each subproblem until the main problem is finally solved.
3. The vertical axis, j, represents that coin denominations available to us.
4. Let T[i][j] represent the table, where i and j are the actual values as seen in the table.
5. Starting at T[1][1] (letter A in table), set this cell to infinity.
6. Check if the current subproblem is >= to the current coin denomination.
7. If so, check if the current subproblem value less the coin denomination + 1 is < the value at T[1][1] (letter A in table above). Since we set A to infinity, this statement is true. Therefore, the value at A is changed to the subproblem value, less the coin denomination, + 1.
8. Then we simply iterate to the next coin denomination and repeat steps 5 – 8 until all coin denominations have been checked. Then we go to the next biggest subproblem, denoted by i++.

This is the optimal solution, because we start at the smallest subproblem, solve for that, store the solution in a table, and proceed to the next biggest subproblem. We solve this next subproblem by referring to the saved solutions of its smaller parts instead of solving for the entire subproblem like one would in a typical recursive call in divide and conquer. This method avoids the downfall of the divide and conquer method by reusing solved subproblems instead of redoing previously executed subproblems.

2. Pseudocode:

Divide and Conquer Implementation:

```
changeslow { //calculate the minimum number of coins
  Let a = amount of change to make
  Let T[] = a table of length = a where each cell tracks the denomination of coin which leads to the minimum
number of coins for the change amount
  Let V[] = coin denominations available for change
    for i = 1 to length.V
      if V[i] == a
        T[a] = i
        minimumNumberOfCoins = 1
      else if V[i] < a
```

```
        minimumNumberOfCoins = min(1 + changeslow(a - V[i])) for all values of i
        T[a] = value of i which produced minimum number of coins
  return minimumNumberOfCoins
}

createCoinArray { //create the array of coins which makes up the minimum number
  Let a, T[], and V[] be the same a, T[], and V[] from changeslow() function
  Let R[] be an array with length = length.V. Each index of R will keep track of the amount of coins of that
index which are used in the minimum change (i.e. if R[2] = 3, then we need 3 coins of denomination = V[2])

  while a > 0
    for i = 1 to length.V
      if i == T[a]
        R[i] = R[i] + 1
        a = a - V[i]
  return R[]
}
```

Greedy Implementation:

```
FindGreedy()
       For each element in the inputArray starting from the last element to the first
              while the current element <= changeAmount
                     subtract the current element value from the changeAmount
                     increment the current denomination in the change denomination array
                     increment the minNumberOfCoins
       return (changeDenominationArray) and (minNumberOfCoins)
```

Dynamic Programming Implementation:

```
Let p = price amount to make change for
Let T[] = be the table cells that count the number of occurrences of each coin denomination
Let C[] = coin denominations available for change
T[0] = 0
for i = 1 to p
       T[i] = infinity
       for j = 0 to count(C[ ])
              if (i >= C[ i ] ) && (1 + T[ i - C[ j ] ] ) < T[ i ] )
                     T[ i ] = 1 + T[ i - C[ j ] ]
return C[ p ]
```

3. Dynamic Programming Proof by Induction:

$$T[v] = \min_{V[i] \le v}\{T[v - V[i]] + 1\},\ T[0] = 0$$

Base case:
Let v = 1
Let V[i] = v = 1, since V[i] <= v
T[1] = $\min_{i:V[i] <= v}${ T[1 - 1] + 1}
= $\min_{i:V[i] <= v}${ T[0] + 1 }
= 1

Inductive case:
Let's assume T[k] = $\min_{i:V[i] <= v}$ {T[k - V[i]] + 1} is true for all k <= n.
We need to prove T[k + 1] is true.

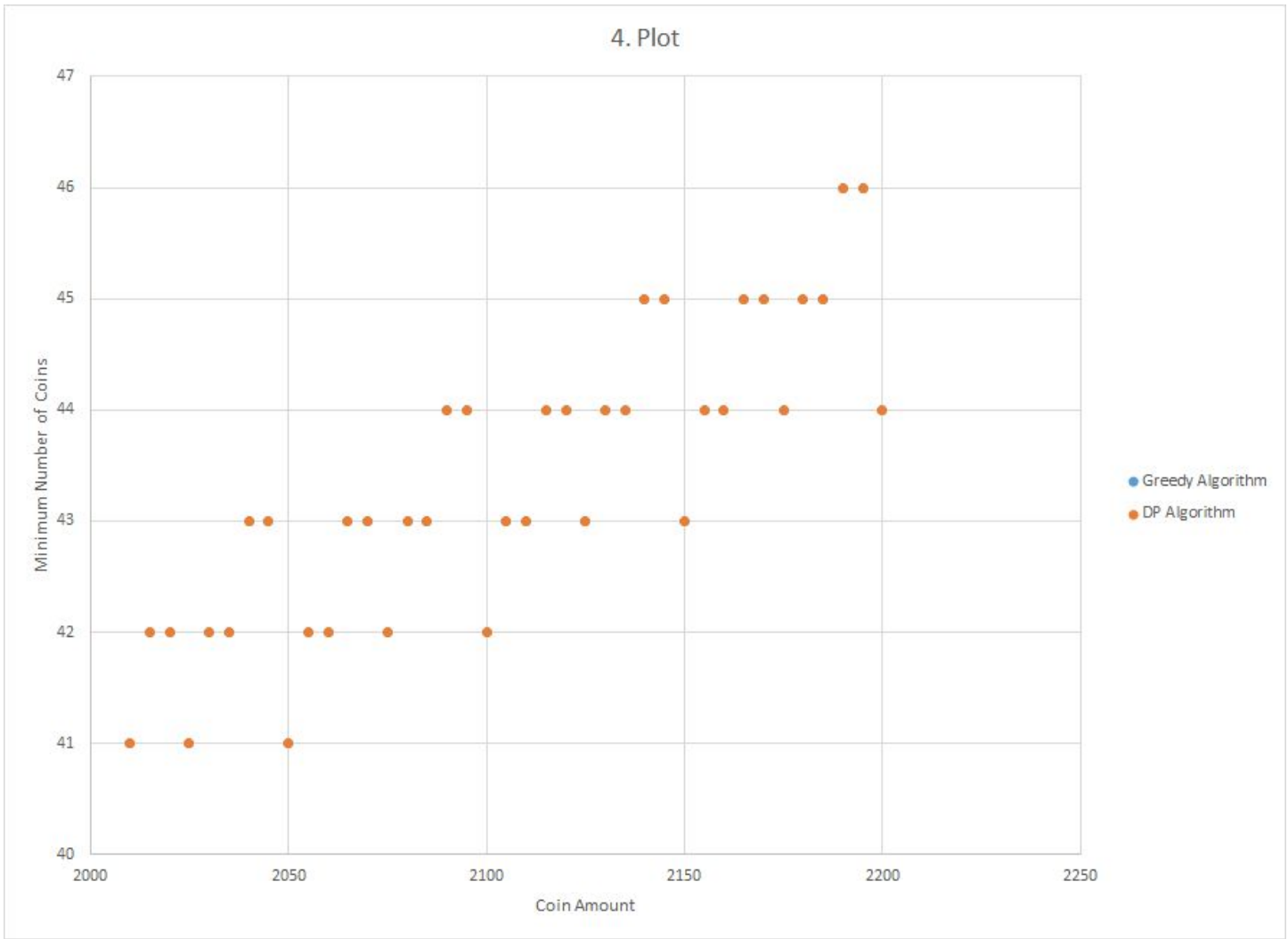
$T[k + 1] = \min_{i:V[i] <= v} \{T[(k + 1) - V[i]] + 1\}$
Here, T[(k + 1) - V[i]] <= T[k], since V[0] must be 1. Since T[k] is true by our inductive hypothesis, our original claim is true. QED.

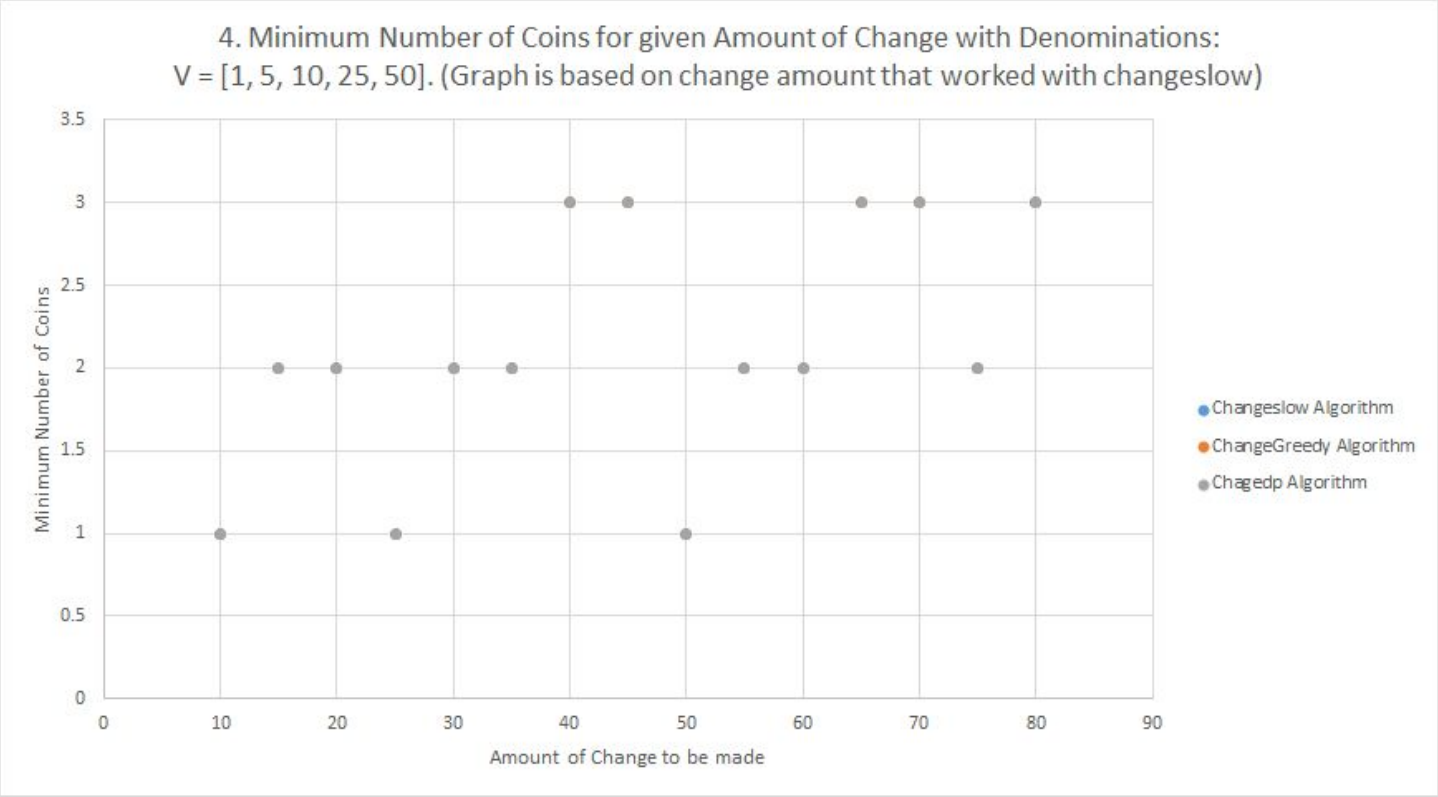
Let v = v - 1
Let V[i] = v - 1, since V[i] <= v
T[v - 1] = min{ T[(v - 1) - (v - 1)] + 1 }
= min{ T[v - 1 - v + 1] + 1 }
= min{ T[0] + 1 }
= 1 $\min_{i:V[i] <= v}$

4. Plot the number of coins as a function of *A* for each algorithm for changedp and changegreedy.

Denominations: *V* = [1, 5, 10, 25, 50].
Change amounts: [2010, 2015, 2020, …, 2200]

There are two graphs below. The first graph uses the coin amounts within the range 2000...2200 for changegreedy and changedp. The second graph uses the coin amounts within the range 10...80 for all three algorithms. There are two graphs because changeslow ran slow for big change amounts.





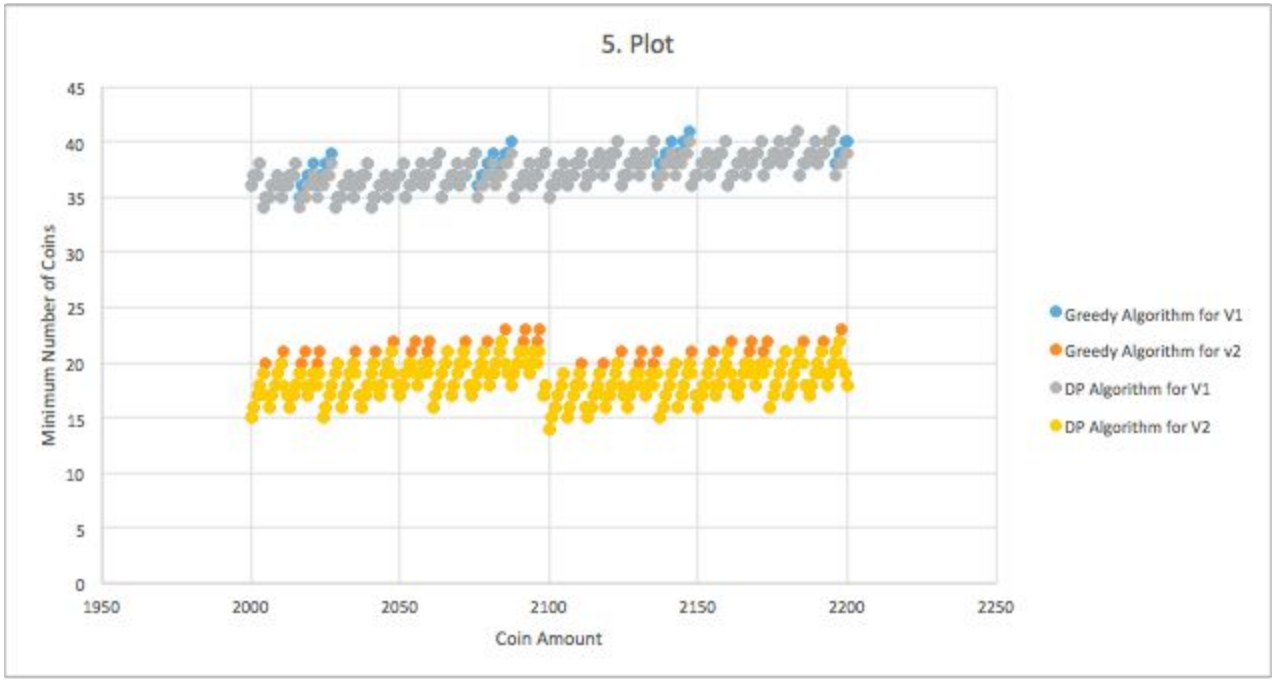
How do the approaches compare?

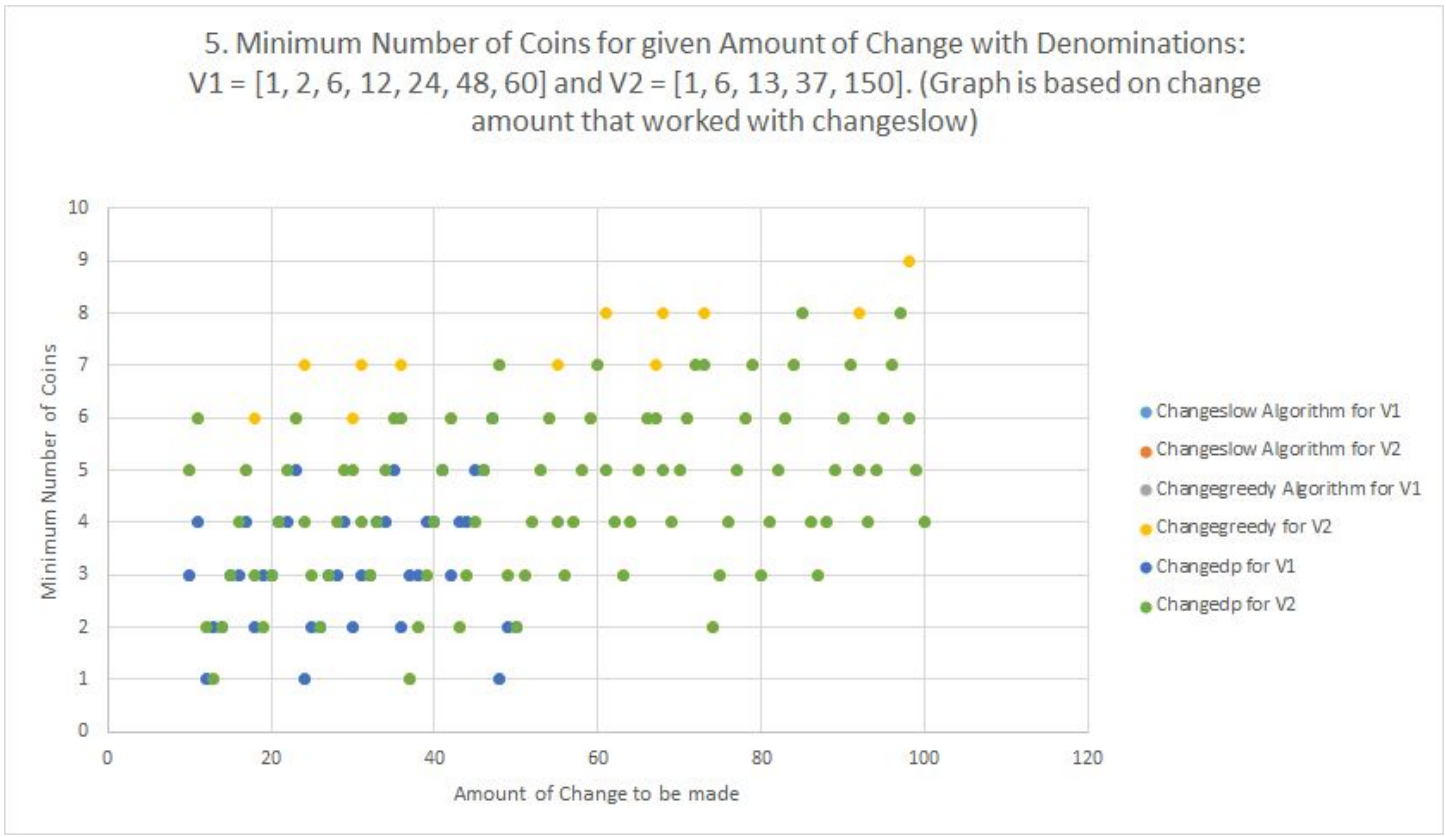
- The approaches are exactly the same. The graph shows that all data points for this specific problem set are overlapping. All three versions of the algorithm provide the same minimum number of coins for this set of denominations.

5. Plot the number of coins as a function of *A* for each algorithm for changedp and changegreedy.

Greedy Algorithm:
Denominations: V_1 = [1, 2, 6, 12, 24, 48, 60] , V_2 = [1, 6, 13, 37, 150]

There are two graphs below. The first graph uses the coin amounts within the range 2000...2200 for changegreedy and changedp. The second graph uses the coin amounts within the range 10...80 for all three algorithms. There are two graphs because changeslow ran slow for big change amounts.





How do the approaches compare?

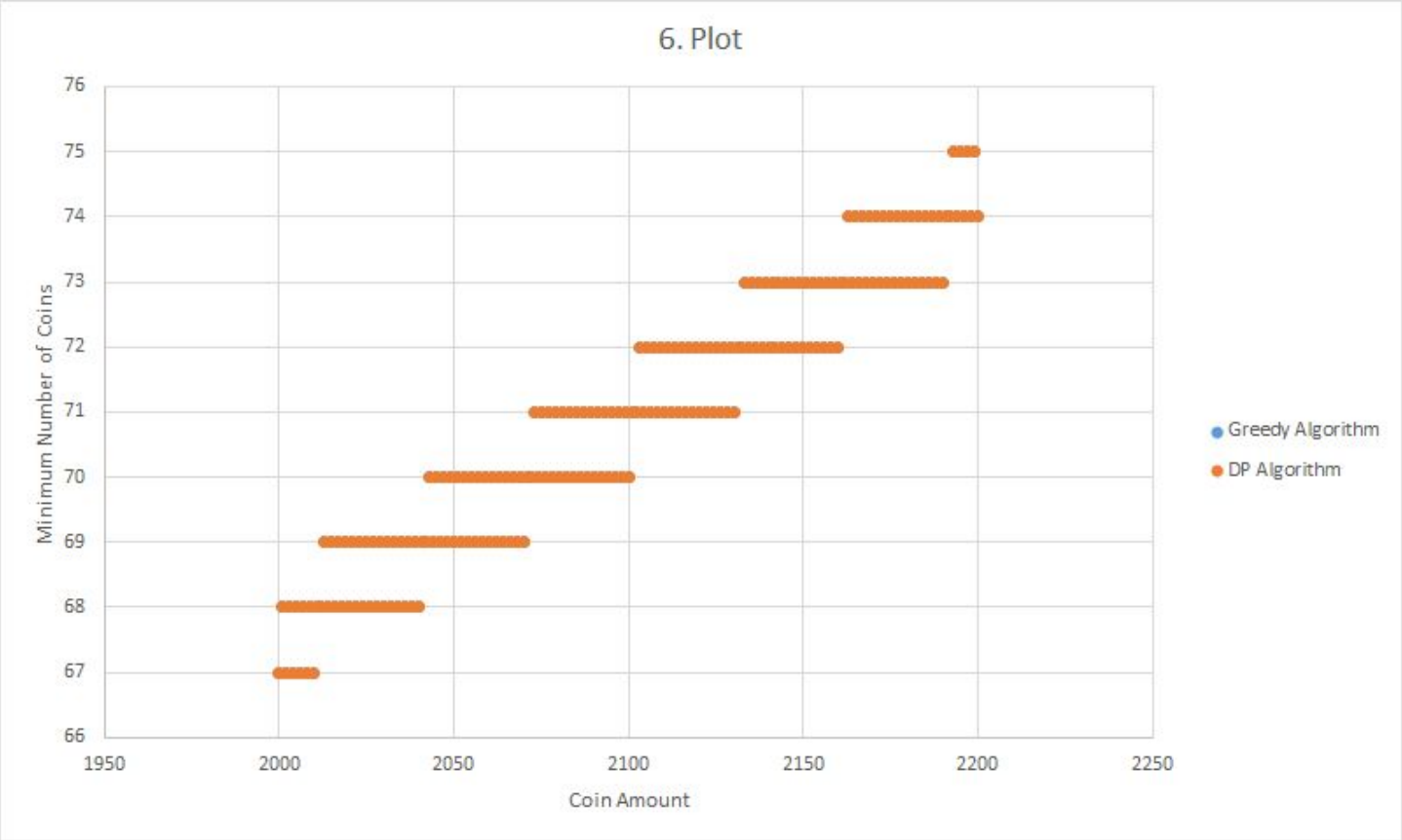
- In all coin amounts above, the greedy algorithm returns a higher minimum number of coins than the dynamic programming and changeslow solutions. This shows that the greedy algorithm does not return the optimal minimal number of coins in certain cases, whereas the dynamic programming and changeslow algorithms always return the minimum number of coins required.

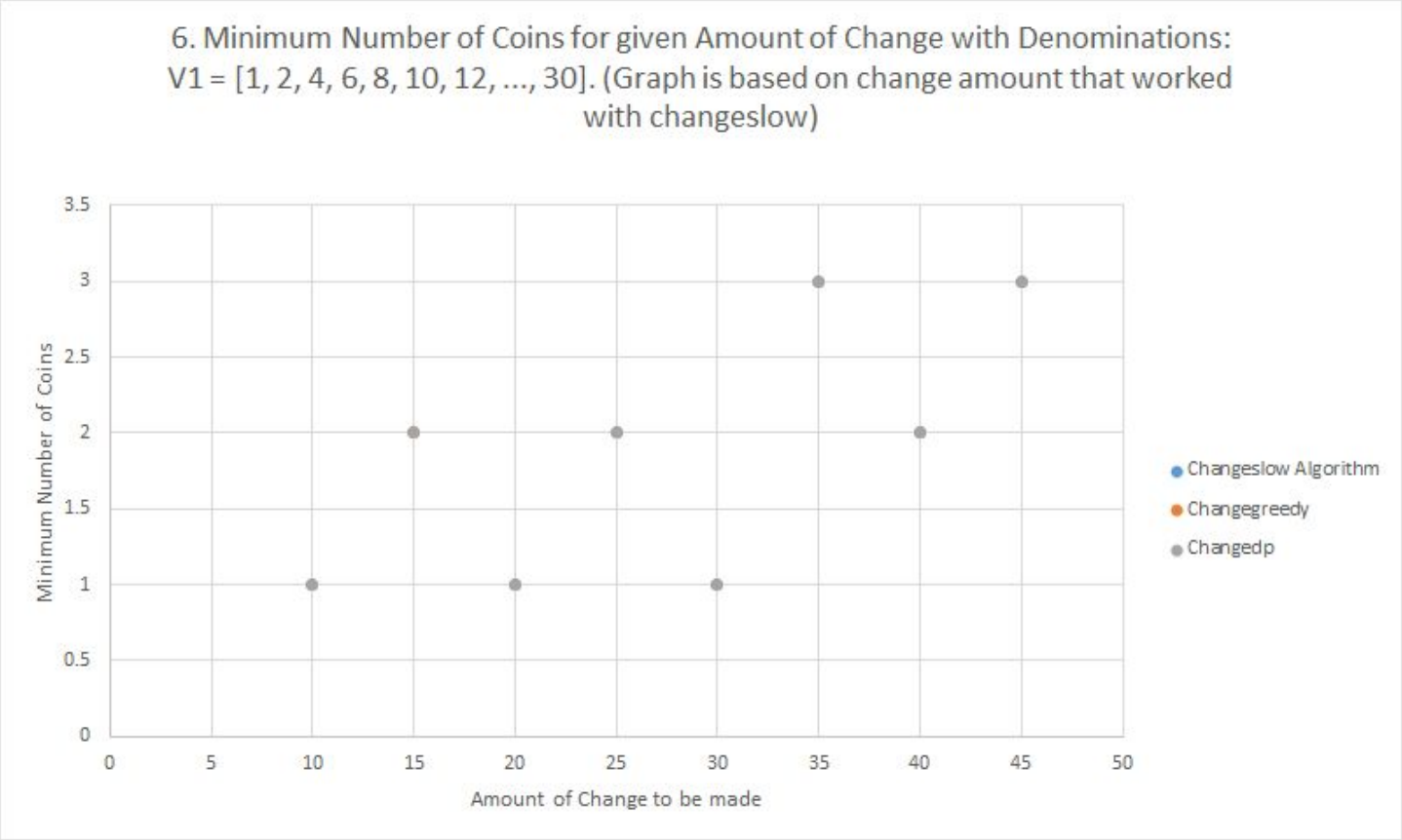
6. Plot the number of coins as a function of *A* for each algorithm for changedp and changegreedy.

Denominations: $V_1 = V = [1, 2, 4, 6, 8, 10, 12, \ldots, 30]$

Change amounts: [2000, 2001, 2002, …, 2200]

There are two graphs below. The first graph uses the coin amounts within the range 2000...2200 for changegreedy and changedp. The second graph uses the coin amounts within the range 10...45 for all three algorithms. There are two graphs because changeslow ran slow for big change amounts.

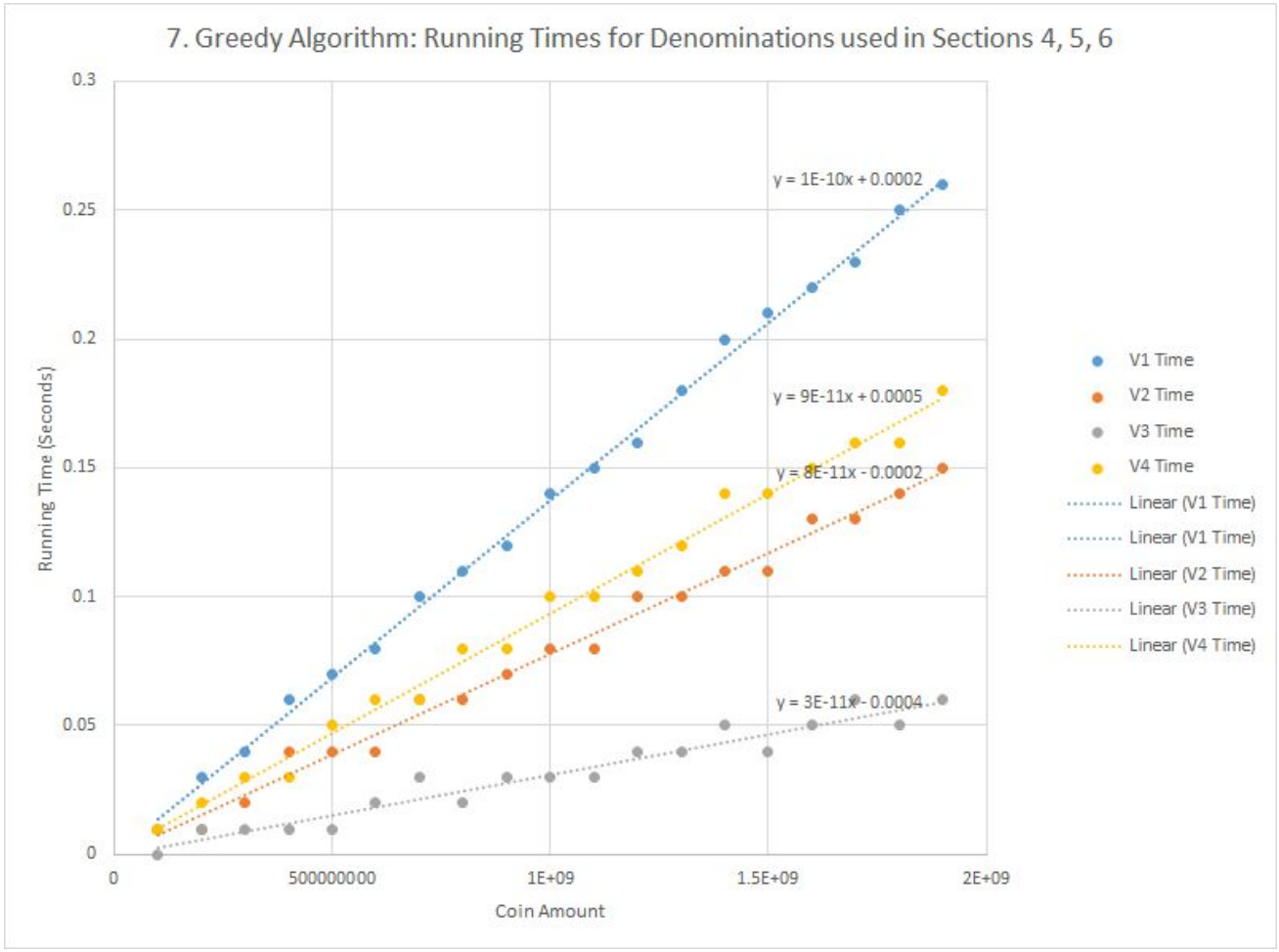


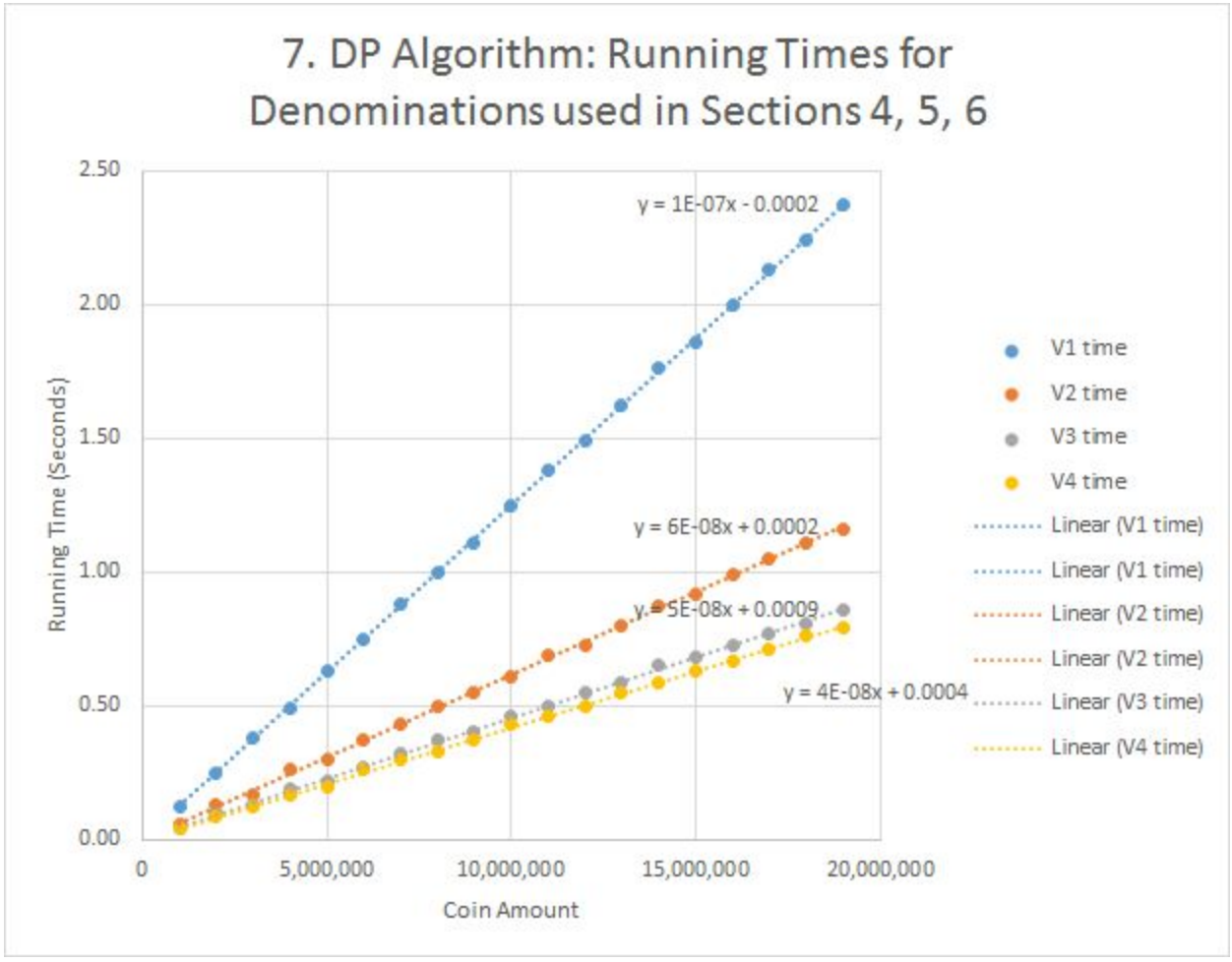


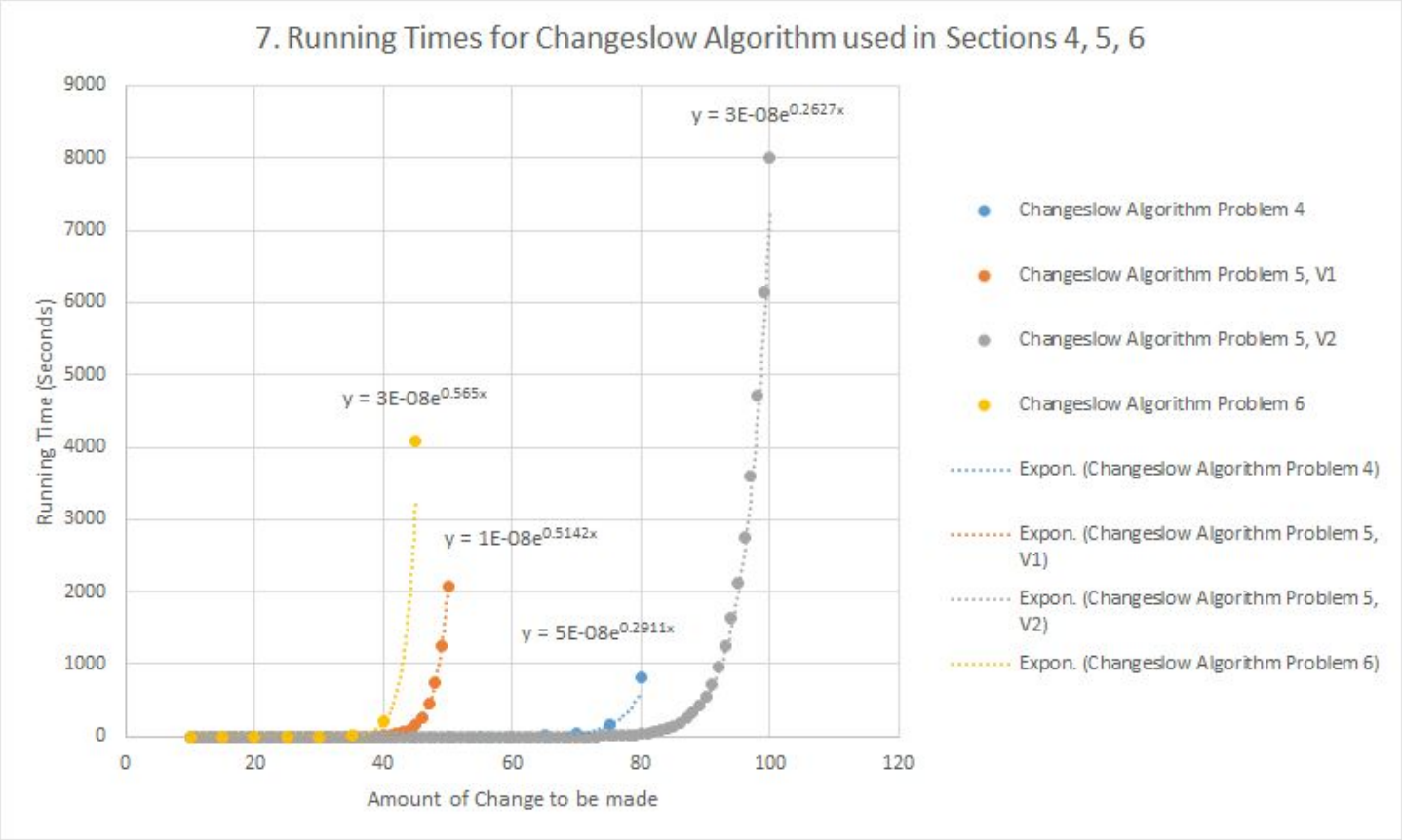
How do the approaches compare?

- With this specific dataset, the greedy and dynamic programming algorithm return the same results. The graph shows all data points overlapping.

7. Plot Running Time as a Function of A


7. Greedy Algorithm: Running Times for Denominations used in Sections 4, 5, 6
Running Time (Seconds)
Coin Amount
0.3
0.25
0.2
0.15
0.1
0.05
0
0
500000000
1E+09
1.5E+09
2E+09
y = 1E-10x + 0.0002
y = 9E-11x + 0.0005
y = 8E-11x - 0.0002
y = 3E-11x - 0.0004
V1 Time
V2 Time
V3 Time
V4 Time
Linear (V1 Time)
Linear (V1 Time)
Linear (V2 Time)
Linear (V3 Time)
Linear (V4 Time)

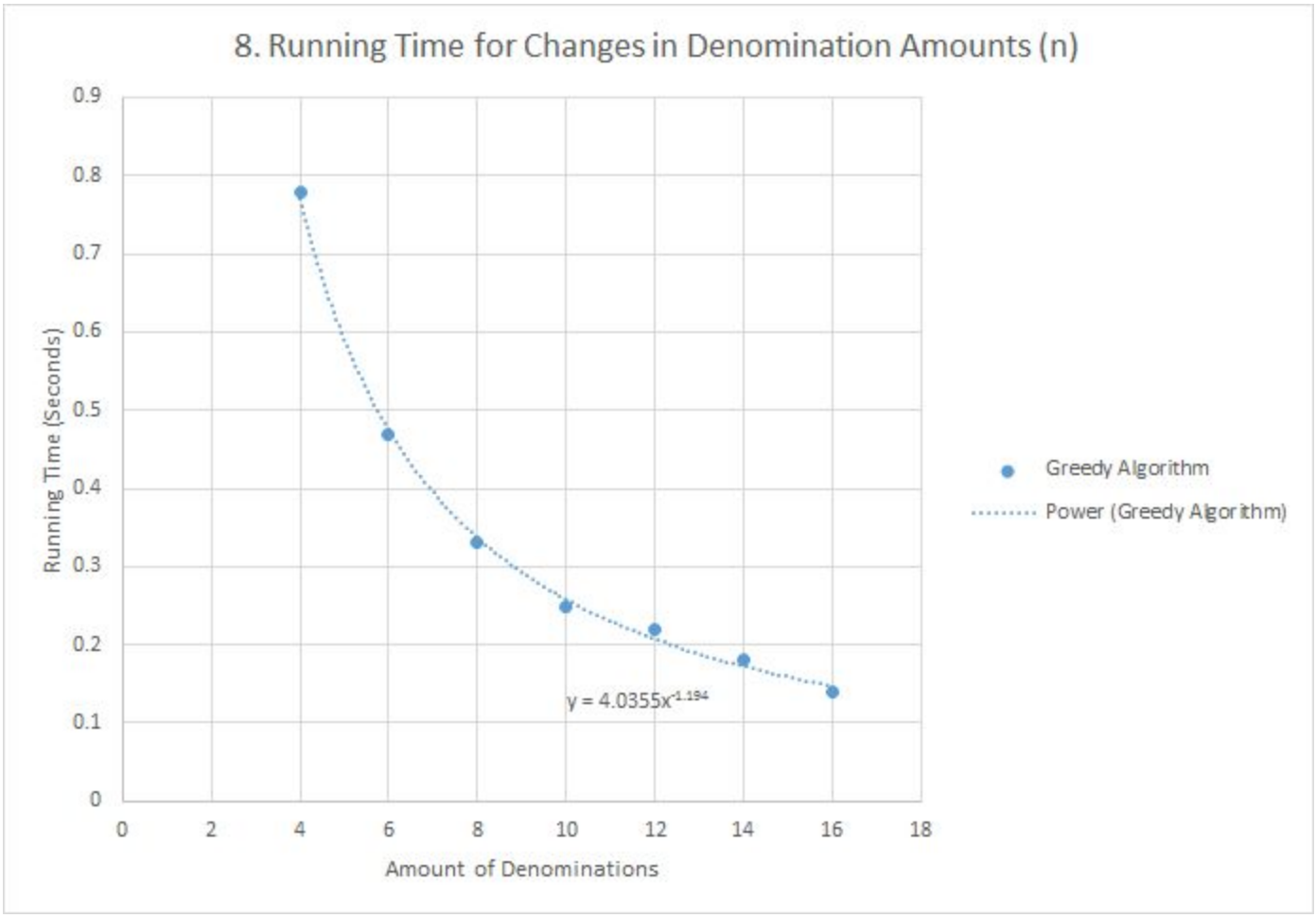

7. DP Algorithm: Running Times for Denominations used in Sections 4, 5, 6
Running Time (Seconds)
2.50
2.00
1.50
1.00
0.50
0.00
0
5,000,000
10,000,000
15,000,000
20,000,000
Coin Amount
y = 1E-07x - 0.0002
y = 6E-08x + 0.0002
y = 5E-08x + 0.0009
y = 4E-08x + 0.0004
V1 time
V2 time
V3 time
V4 time
Linear (V1 time)
Linear (V1 time)
Linear (V2 time)
Linear (V2 time)
Linear (V3 time)
Linear (V4 time)



How do the approaches compare?

- Looking at the regression equation for the greedy and dynamic programming graphs, we can see that the slope for the greedy algorithm is lower than the corresponding slope of the dynamic programming algorithm in all cases. In fact, the greedy algorithm is magnitudes of order lower than the dynamic programming algorithm. This makes sense because the greedy algorithm is a simple loop, while the dynamic programming algorithm is multiple loops trying to fill in a 2D array.
- For the divide and conquer algorithm we see that every plot is exponential for run time. This makes sense as the algorithm is rife with duplicated recursive calls solving the same subproblems repeatedly. This is in stark contrast to the dynamic programming algorithm which solves each subproblem only once and the greedy algorithm which only solves a fraction of the subproblems and solves those just a single time each.

8. Plot Running Time as a Function of Denomination Amounts



8. changedp running times as a function of the number of denominations

V_1 = [1, 2, 4, 6, 8, 10, 12, ..., 30] (problem 6)

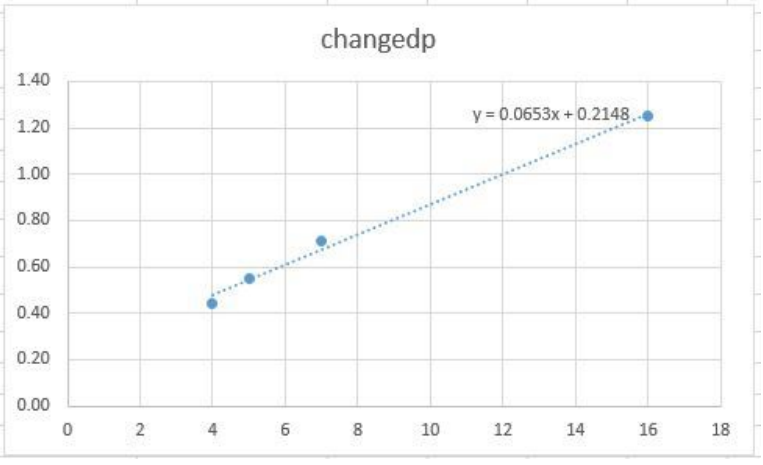
V_2 = [1, 2, 6, 12, 24, 48, 60] (problem 5 v1)

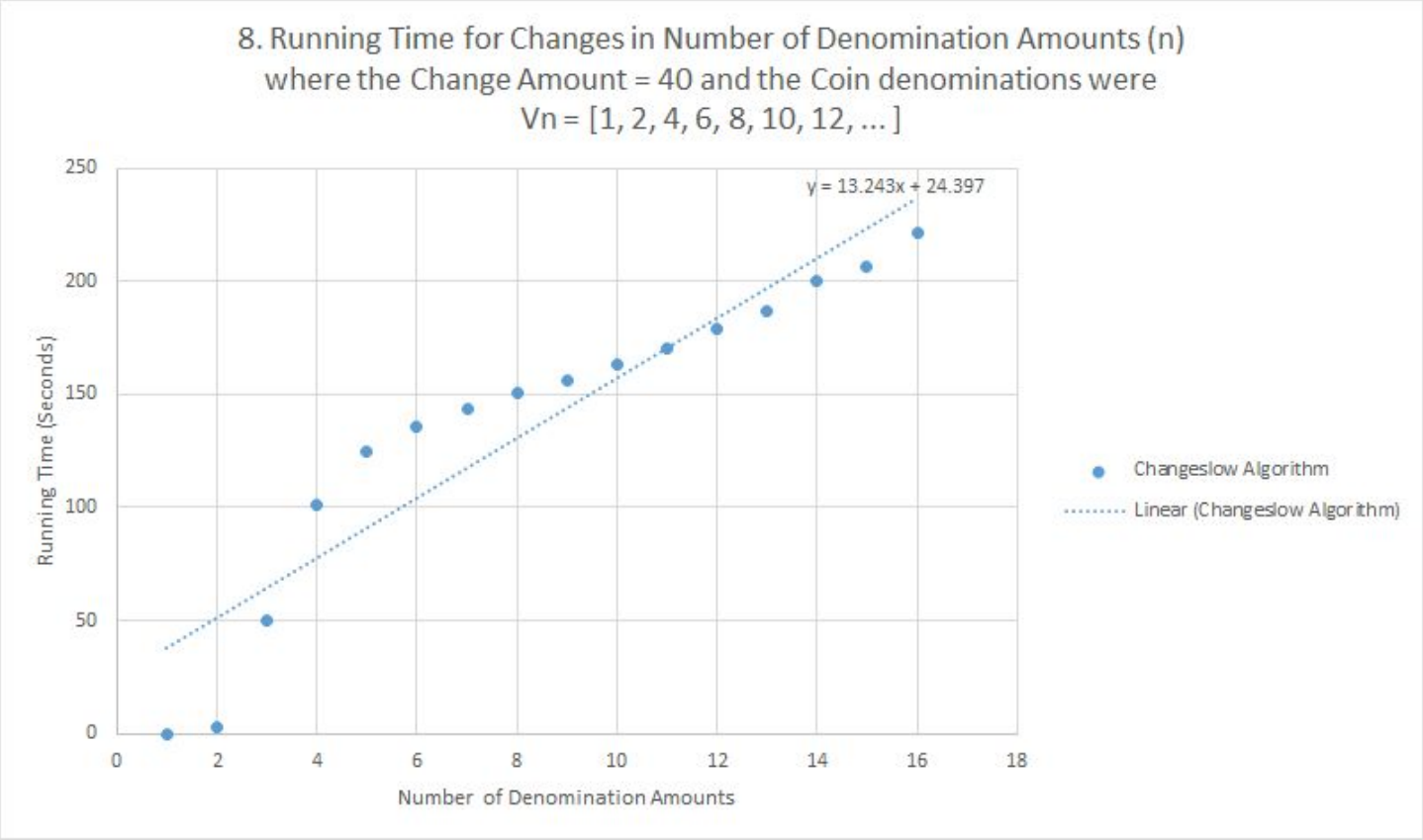
V_3 = [1, 2, 6, 12, 24, 48, 60] (problem 5 v2)

V_4 = [1, 5, 10, 25]

For amount of 10,000,000

n	time (seconds)	Function
16	1.25	V_1
7	0.71	V_2
5	0.55	V_3
4	0.44	V_4





Does the size of n influence the running time?

- It is quite obvious that the size of n has an adverse effect on the running times of both the dynamic programming and brute force algorithms. The reason for this is that as the number of denominations of available coins increases, then the number of subproblems increases. For example if there are only 3 coin denominations, then for every change amount you would only have to compare the minimum coins required for the three cases where each of those coins is chosen whereas if there were 10 coin denominations then you would have 10 subproblems to check (and each of those subproblems would have an additional 10 subproblems compared to three with the three denominations, and so on and so on).
- For the greedy approach, I don't believe the number of denominations is as important as the difference in size between the largest coin denomination and the amount of change to be made. The larger this difference becomes the longer the algorithm takes to run. The results of this plot are a little misleading because what is really happening is that as n increases the difference between the amount of change to be made and the largest single coin denomination is shrinking and this is what is causing the run time to drop. Had the timings been done where the largest denomination value remained and the second lowest was the one removed to lower n, the run times would have been fairly constant.

9. Suppose you are living in a country where coins have values that are powers of p, V = [p^0, p^1, p^2, …, p^n]. How do you think the dynamic programming and greedy approaches would compare?

In this scenario, the minimum number of coins returned by dynamic programming and greedy approaches would be identical which would make the greedy approach the ideal approach due to the reduced run time.

The reason these two approaches would return the same result has to do with the fact that each available coin denomination is a multiple of every other coin denomination (p^1 equals p coins of p^0 value; p^2 equals p coins of p^1 value; … ; p^n equals p coins of p^{n-1} value). Because of this fact there is no amount of change, a, where $a \geq p^n$ where the minimum number of coins of change does not include the coin with value p^n. If we assume that in fact the minimum number of coins to make change of amount , a, where $a \geq p^n$ does not include the coin of denomination p^n, then it must use at least p coins of denomination p^{n-1}. However, it is quite obvious that 1 coin of denomination p^n is less than p coins of denomination p^{n-1} in terms of number of coins. Therefore, since the minimum number of coins results from always taking the largest possible denomination of coin possible, the greedy algorithm will return the same result as the dynamic programming algorithm.