Project Two: Coin Change

Group 12

Group Members

- Kyle Guthrie
- Michael C. Stramel
- Alex Miranda

Divide and Conquer

Pseudo-code

```
CHANGE_SLOW(coin_array, change)
    coin_count = [0,...,coin_array.length - 1] = 0

        if (change exists in coinArray)
            coin_count[index of change] = 1
            return coin_count, 1
        else if (change = 0)
            return coin_count, 0

           min_coins = infinity

           for (i = 1 to change/2)
               coin_count_left, min_coins_left =
                CHANGE_SLOW(coin_array, i)
               coin_count_right, min_coins_right =
                CHANGE_SLOW(coin_array, change - i)
               total_coins = min_coins_left + min_coins_right

               if total_coins < min_coins:
                   min_coins = total_coins

                   for (j in coin_count_left/right)
                       coin_count =
                        coin_count_left[j] + coin_count_right[j]

    return coin_count, min_coins
```

Runtime Analysis

The algorithm takes two inputs: coinArray and change. The parameter coinArray contains all of the possible denominations of coins that can be used to make change with. The parameter change is the amount of change that needs to be composed of various combinations of coins that are denoted in the coinArray. From this point forward we will refer to the length of the coinArray as k and the value for change as n.

As can be seen from the pseudo-code above, we first check our k coins to see if any are exactly equal to our change amount. After that, a for-loop runs from 1 to $\frac{n}{2}$, where this value is assumed to be an integer. Our recurrence is of the form:

$$T(n) = \sum_{i=1}^{\frac{n}{2}} (T(n-i) + T(i)) + k$$

The loop only needs to run to $\frac{n}{2}$ due to the fact that values above $\frac{n}{2}$ will simply recompute a sum already computed. As an exmpale, consider the case where $n = 5$. When $i = 2$, we compute $T(3) + T(2)$. If we continued on to $i = 3$, we would compute $T(2) + T(3) \equiv T(3) + T(2)$.

While this recurrence is difficult to solve, we can consider the case without our optimization and consider it an upper bound. Consider the recurrence:

$$T(n) = \sum_{i=1}^{n-1} (T(n-i) + T(i)) + k$$

The resulting sum will look like:

$$T(n-1) + T(n-2) + \ldots + T(2) + T(1) + T(1) + T(2) + \ldots + T(n-2) + T(n-1)$$

Which we can see is equivalent to:

$$2 \cdot (T(n-1) + T(n-2) + \ldots + T(2) + T(1))$$

So, our recurrence can be expressed as:

$$T(n) = 2 \sum_{i=1}^{n-1} T(i) + k$$

As an additional step, we note that $T(n-1)$ is:

$$T(n-1) = 2 \sum_{i=1}^{n-2} T(i) + k$$

We can compute the difference between $T(n)$ and $T(n-1)$ as:

$$\begin{aligned} T(n) - T(n-1) &= \left(2 \sum_{i=1}^{n-1} T(i) + k\right) - \left(2 \sum_{i=1}^{n-2} T(i) + k\right) \\ &= 2T(n-1) \end{aligned}$$

Solving for T(n), we find that:

$$T(n) = 3T(n-1)$$

By the Muster Theorem, with $a = 3$, $b = 1$, and $d = 0$, we can say that $T(n)$ is $O(3^n)$.

Greedy

Pseudo-code

```
GREEDY(coin_array, change)
    coin_count = [0,...,coin_array.length - 1] = 0

    i = coin_count.length - 1

    while change > 0
        if change >= coin_array[i]
            change -= coin_array[i]
            coin_count[i]++
        else
            i--

    return coin_count, sum(coin_count[0,...,coin_count.length])
```

Runtime Analysis

The algorithm takes two inputs: coinArray and change. The parameter coinArray contains all of the possible denominations of coins that can be used to make change with. The parameter change is the amount of change that needs to be composed of various combinations of coins that are denoted in the coinArray. From this point forward we will refer to the length of the coinArray as k and the value for change as n.

As can be seen from the pseudo-code above we first start with a while-loop that starts with $n = change$ and continues until $n = 0$ (until change has been made). Within that loop each element of coinArray (from largest to smallest, $k - 1$ to 0 is checked against the remaining value of change n to be made and subtracted if possible. If a given coin is too large to be subtracted from the remaining change, then the next lower coin is attempted. Because the lowest coin value is guaranteed by the problem statement to have $value = 1$, a solution is to the problem is guaranteed.

The while-loop is the only iterating construct. But how long does it run? In the very best case - the highest coin value in coinArray wil be equal to change n. An example might be [1 25 50 100] with $n = 100$. In this case, the loop will run just once for $O(1)$ complexity. In the worst case - all coin values in coinArray (other than 1) are greater than change n. An example might be [1 25 50 100] with $n = 24$.

In this worst case the while-loop will run k times - once for each value stored in coinArray as it searches for a coin value less than or equal to n. In this case the only coin value that works is 1 and therefore the while loop will run n times - subtracting off a value of 1 for each execution of the loop. This leaves us with a runtime of:

$$O(k + n - 1) + O(1)$$

The -1 above is removing double booking of the first subtraction of $value = 1$ from n (both at the end of the k "loop" and beginning of the n "loop"). The constant term -1 drops off as well as the $O(1)$ term because the $O(k + n)$ will dominate as n or k become increasingly large. Therefore the finalized runtime of the greedy change making algorithm is:

$$O(k + n)$$

Dynamic Programming

Pseudo-code

```
CHANGEDP(coin_array, change) {
    min_coins_used = [0,...,(change + 1)]
    min_coins_count = [0,...,(change + 1)]

    for coin in range(0,...,(change + 1)) {
        coin_count = coin
        latest_coin = 1

        for i in [coin_val in coin_array where coin_val <= coin]{
            if min_coins_count[coin - i] + 1 < coin_count {
                coin_count = min_coins_count[coin - i] + 1
                latest_coin = i
            }
        }
        min_coins_count[coin] = coin_count
        min_coins_used[coin] = latest_coin
    }

    min_count = min_coins_count[change]
    min_used = []

    for k in coin_array {
        min_used.append(0)
    }

    change_iter = change

    while change_iter > 0 {
        coin_val = min_coins_used[change_iter]
        p = 0
        for j in coin_array {
            if coin_val == j {
                min_used[p] += 1
            }
            p += 1
        }
        change_iter = change_iter - coin_val
    }

    return min_used, min_count
}
```

Runtime Analysis

The algorithm takes two inputs: coinArray and change. The parameter coinArray contains all of the possible denominations of coins that can be used to make change with. The parameter change is the amount of change that needs to be composed of various combinations of coins that are denoted in the coinArray. From this point forward we will refer to the length of the coinArray as k and the value for change as n.

As can be seen from the pseudo-code above we first start with a for-loop that iterates from 0 to n. Each change value between 0 and n are then checked against each denomination in the coinArray. This check occurs in the inner for loop where the values are from 0 to k in the worst case. We say worst case because if the change amount is less than any coin denominations within the coinArray then those coin denomination(s) are skipped within the loop. Therefore this portion of the algorithm will have a runtime of $O(n * k)$. The next for loop creates the min_used array and sets it to a length k which will add a factor of $O(k)$ to the runtime. After that there is a while loop that runs similarly to the first nested for loop so it also will have a runtime of $O(n * k)$. The various variable assignments and index look-ups for the arrays within the method all will have $O(1)$. This leaves us with a runtime of:

$$2 * O(n * k) + O(k) + O(1)$$

Where the constant term 2 drops off as well as the $O(k)$ and $O(1)$ terms because the $O(n * k)$ will dominate as n becomes increasingly large. Therefore the finalized runtime of the dynamic programming change making algorithm is:

$$O(n * k)$$

Dynamic Programming Table (Explained)

As described in the assignment, our dynamic programming approach uses table T indexed by values of change 0, 1, 2, ..., A where $T[v]$ is the minimum number of coins needed to make change for v and $T[0] = 0$.

$$T(v) = \min_{V[i] \leq v} \{T[v - V[i]] + 1\}$$

So how does our algorithm fill out this table T? Starting with $T[1]$ and increasing to $T[A]$ - using information already determined for smaller values of v, and considering the impact of adding one (and only one) additional coin from those available. This is best described by example.

In our example our coins are $[1, 5, 10, 25, 50]$ and we are trying to find $T[27]$. By definition we already know $T[0]$ thru $T[26]$. The available coins are 1, 5, 10, and 25. 50 isn't available because it's greater than 27.

So first we consider the coin with value of 1. We are adding this single coin to an existing pile of coins. Simple math tells us that because we are adding a single coin of value 1 to make 27 total, we must already have a total value of 26 prior to adding the coin. We can look up $T[26]$ because it's already been solved in a prior step. $T[26] = 2$ (a quarter and a penny), so a potential value of $T[27]$ is 3 (2 + 1). Before we declare $T[27] = 3$ we need to check the other available coins for a smaller possibility.

Next we consider the coin with value of 5. By the same logic described above, because we are adding a single coin of value 5 to make 27 total, we must already have a total value of 22 prior to adding the coin. We can look up $T[22]$ because it's already been solved in a prior step. $T[22] = 4$ (two dimes and two pennies), so another potential value of $T[27]$ is 5 (4 + 1).

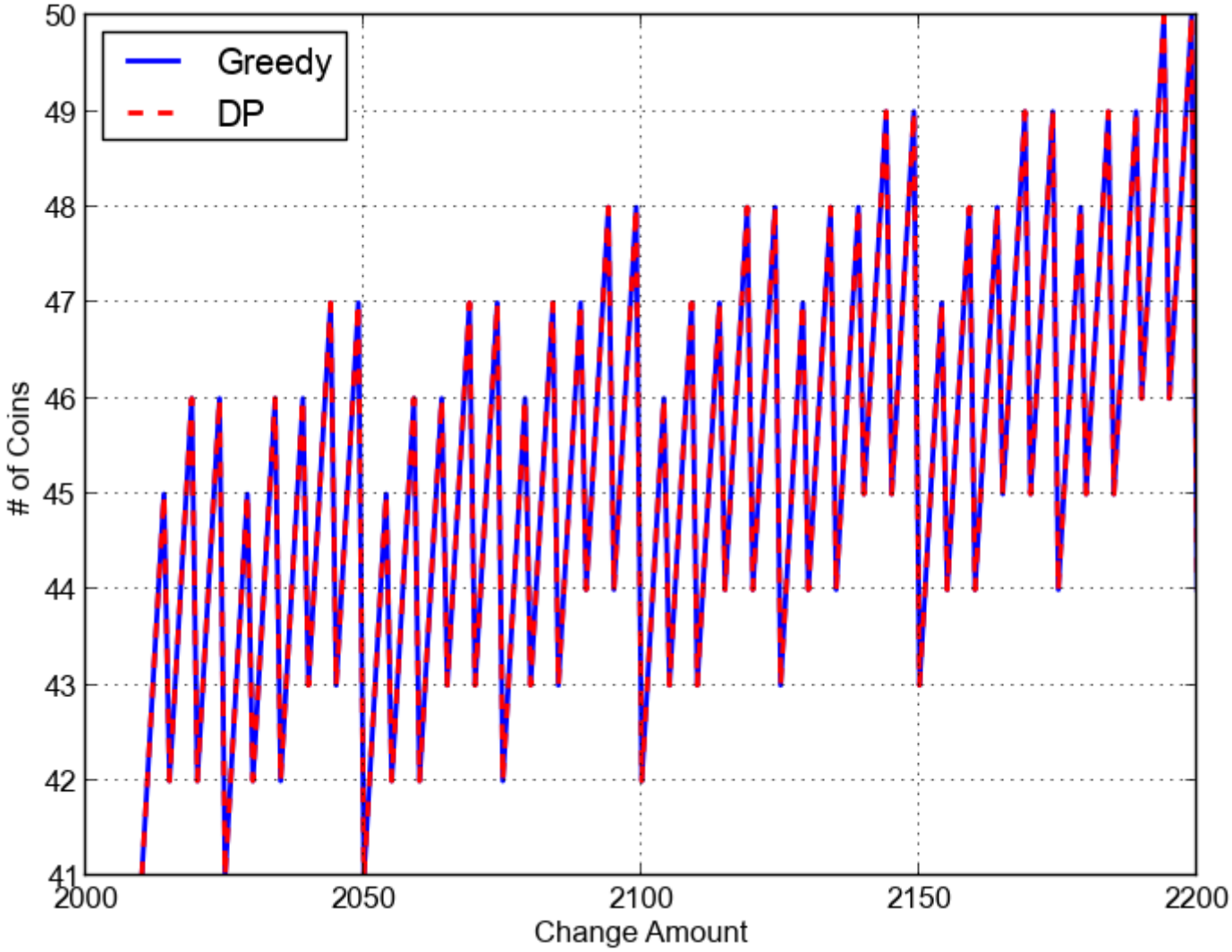
Next we consider the coin with value of 10. By the same logic described above, because we are adding a single coin of value 10 to make 27 total, we must already have a total value of 17 prior to adding the coin. We can look up $T[17]$ because it's already been solved in a prior step. $T[17] = 4$ (a dime, a nickle and two pennies), so another potential value of $T[27]$ is 5 (4 + 1).

Finally we consider the coin with value of 25. By the same logic described above, because we are adding a single coin of value 25 to make 27 total, we must already have a total value of 2 prior to adding the coin. We can look up $T[2]$ because it's already been solved in a prior step. $T[2] = 2$ (two pennies), so another potential value of $T[27]$ is 3 (2 + 1).

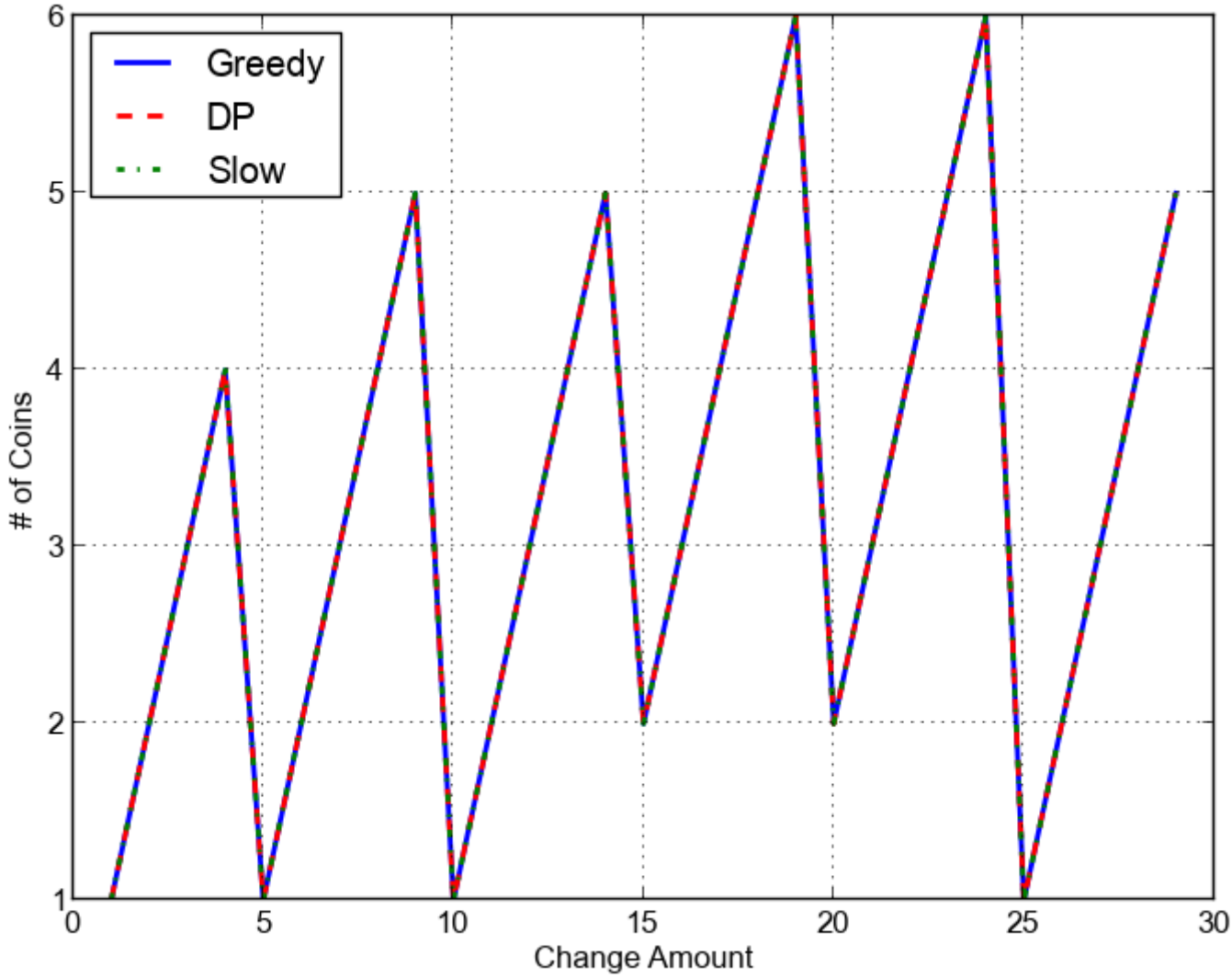
Therefore, looking at the minimum possible values we declare $T[27] = 3$. This method of filling out $T(v)$ is valid because it considers and compares the impact of adding 1 coin of each available value at all steps while taking into account the prior known minimum coins for lesser values of v.

Question 3

V=[1, 5, 10, 25, 50] with A=[2010, 2015, 2020, ..., 2200]



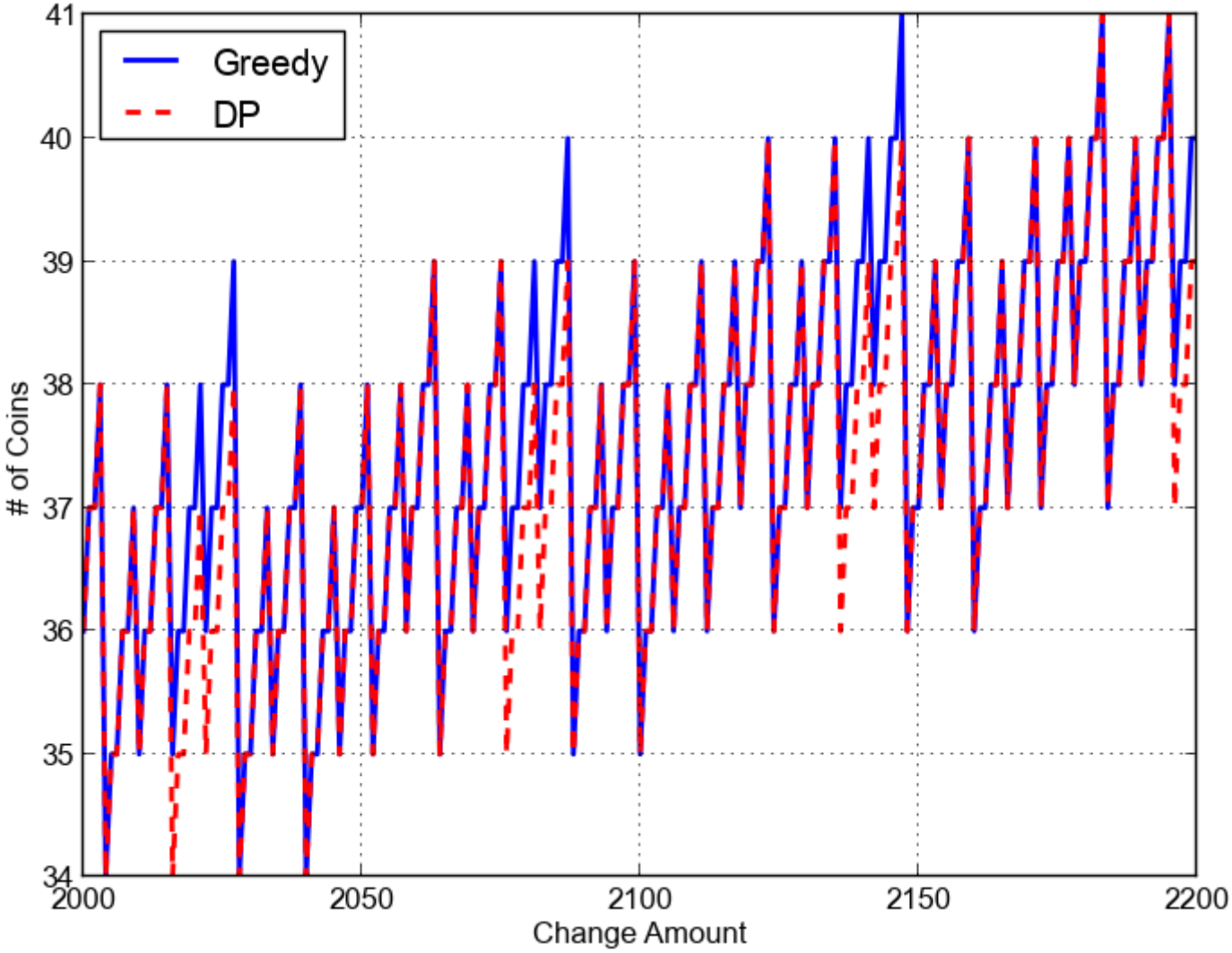
V=[1, 5, 10, 25, 50] with A=[1, 2, 3, ..., 30]



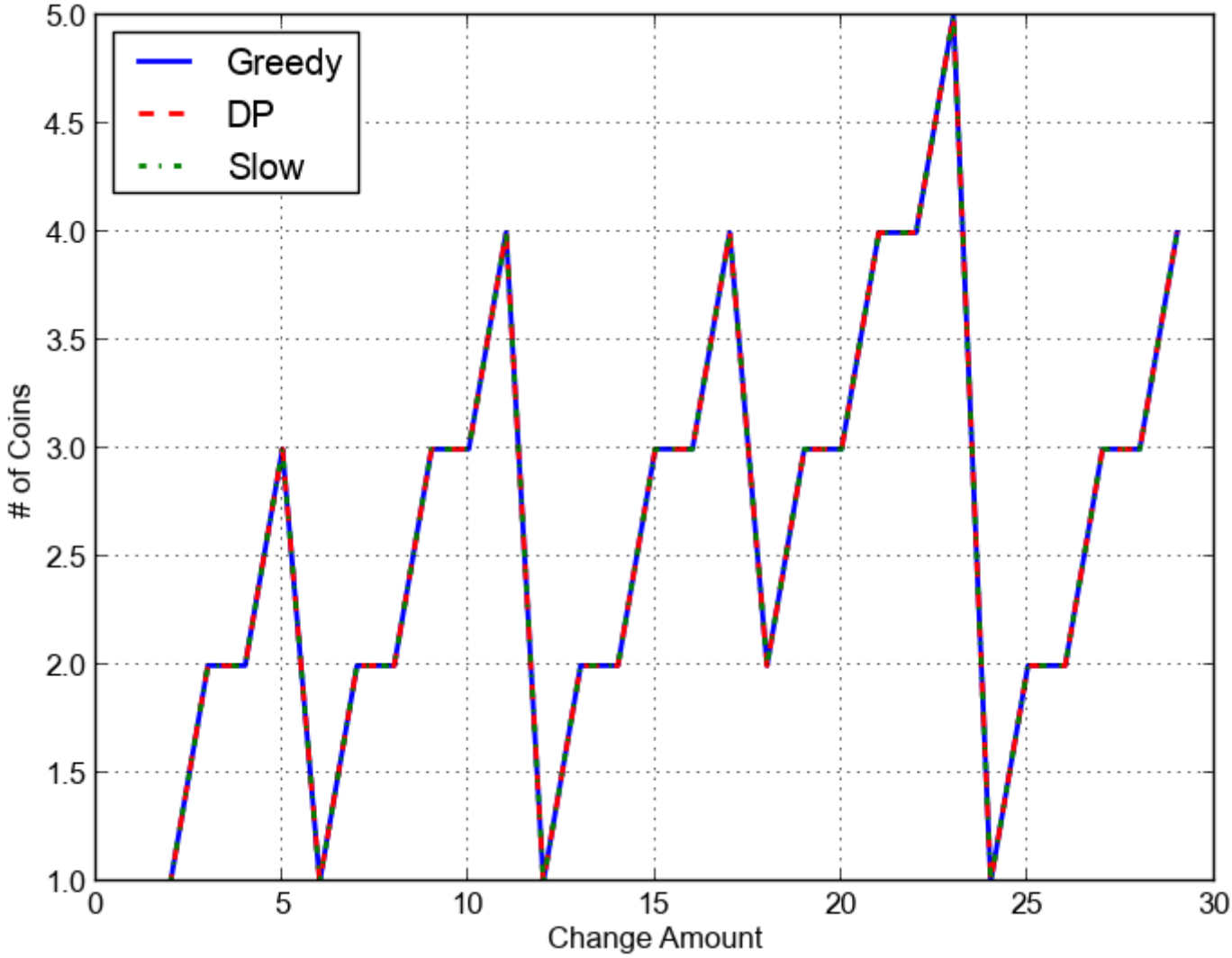
Note that the "changeslow" algorithm had to be run for $A < 30$ due to extremely long running times. The plots above suggest that the coin system [1,5,10,25,50] may be canonical[1]. A coin system is canonical if the number of coins given in change by the greedy algorithm is optimal for all amounts. **Our observations show that greedy is equal to DP (and Slow) *for all values of A that were plotted* for this coin system.**

Question 4

V=[1, 2, 6, 12, 24, 48, 60] with A=[2000, 2001, 2002, ..., 2200]

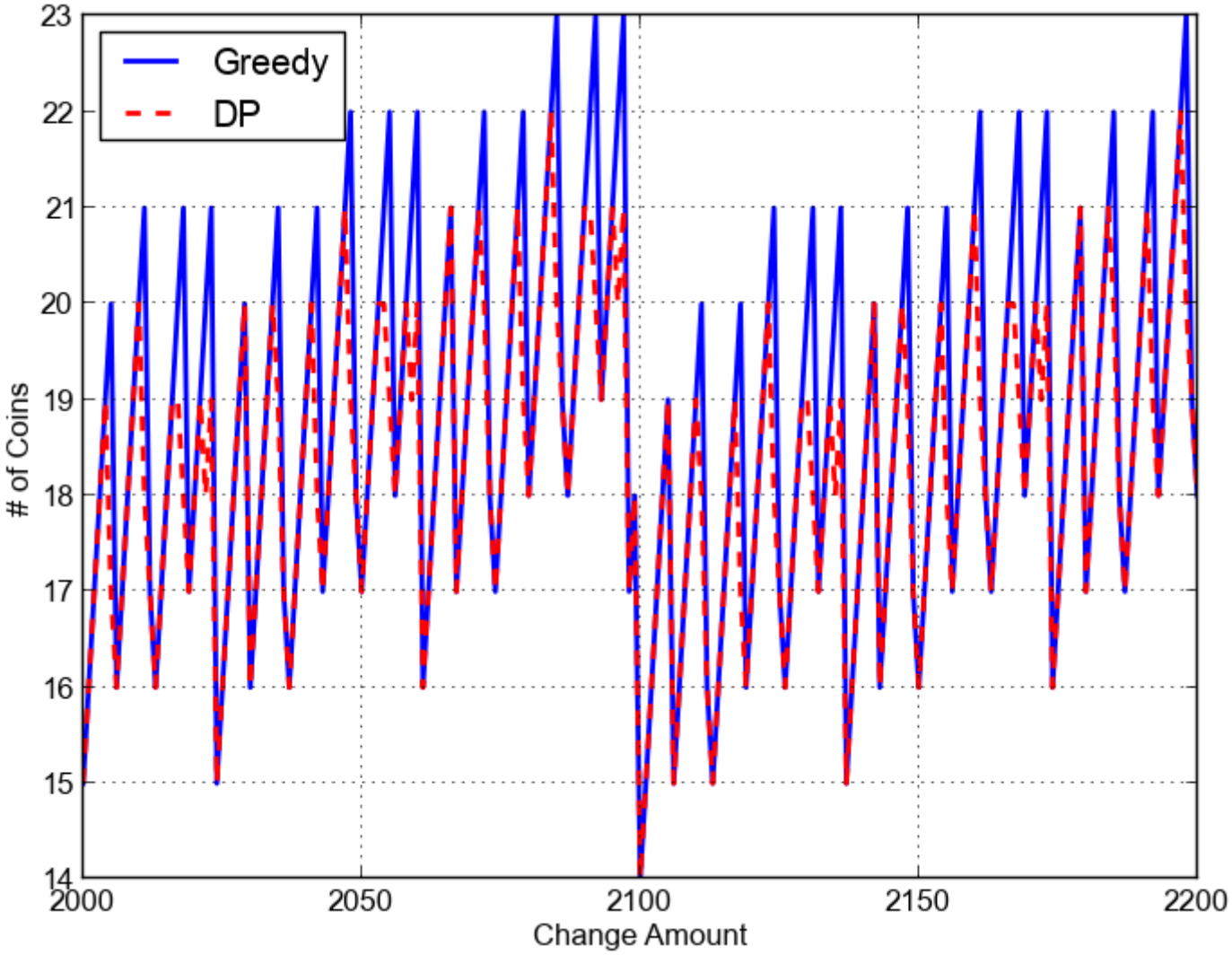


V=[1, 2, 6, 12, 24, 48, 60] with A=[1, 2, 3, ..., 30]

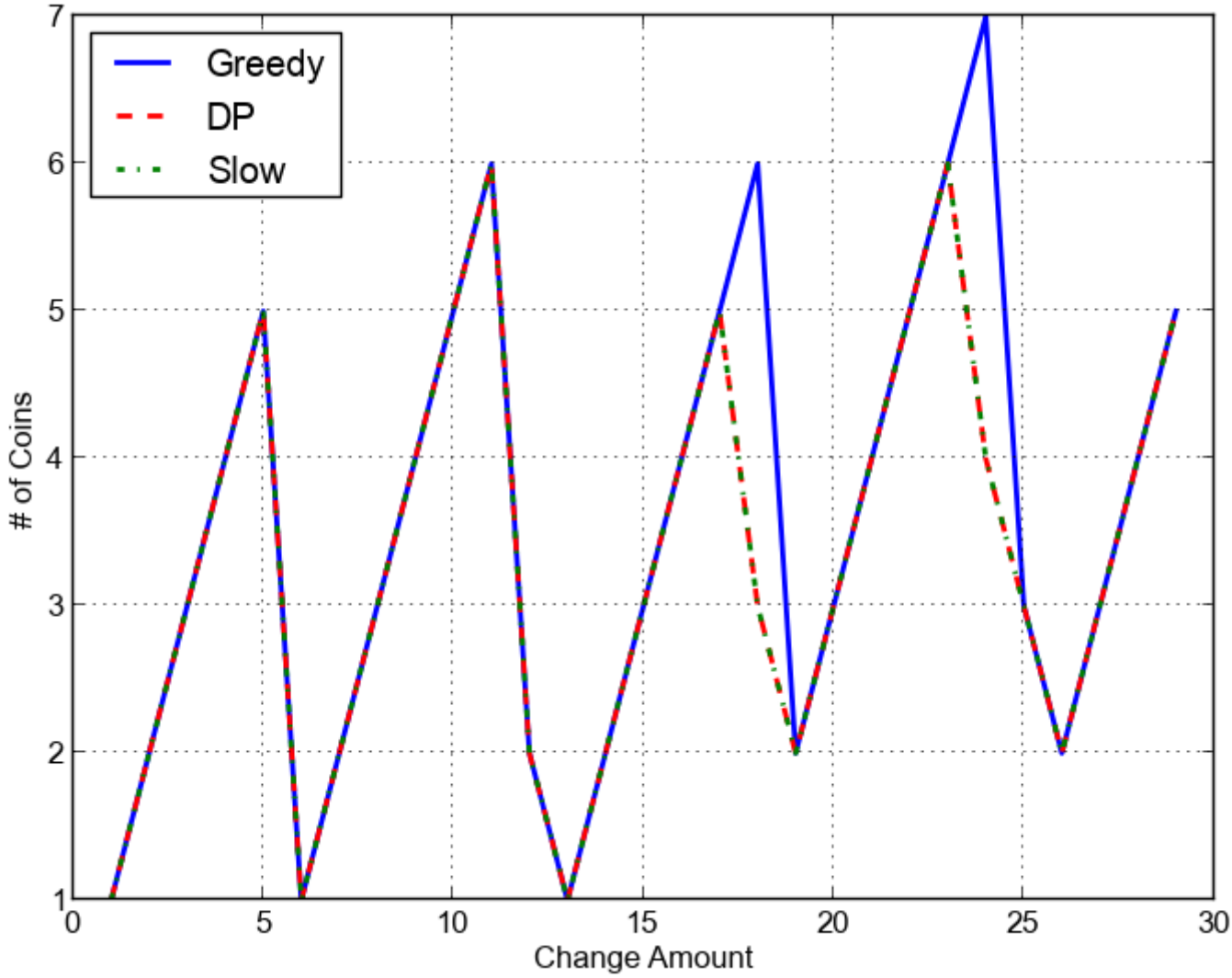


Note that the "changeslow" algorithm had to be run for $A < 30$ due to extremely long running times. The plots above suggest that the coin system [1,2,6,12,24,48,60] **IS NOT** canonical[1]. A coin system is canonical if the number of coins given in change by the greedy algorithm is optimal for all amounts. **Our observations show that greedy is NOT equal to DP (or Slow) *for all values of A that were plotted* for this coin system.** There are multiple values of A between 2000 and 2200 for which greedy is sub-optimal.

V=[1, 6, 13, 37, 150] with A=[2000, 2001, 2002, ..., 2200]



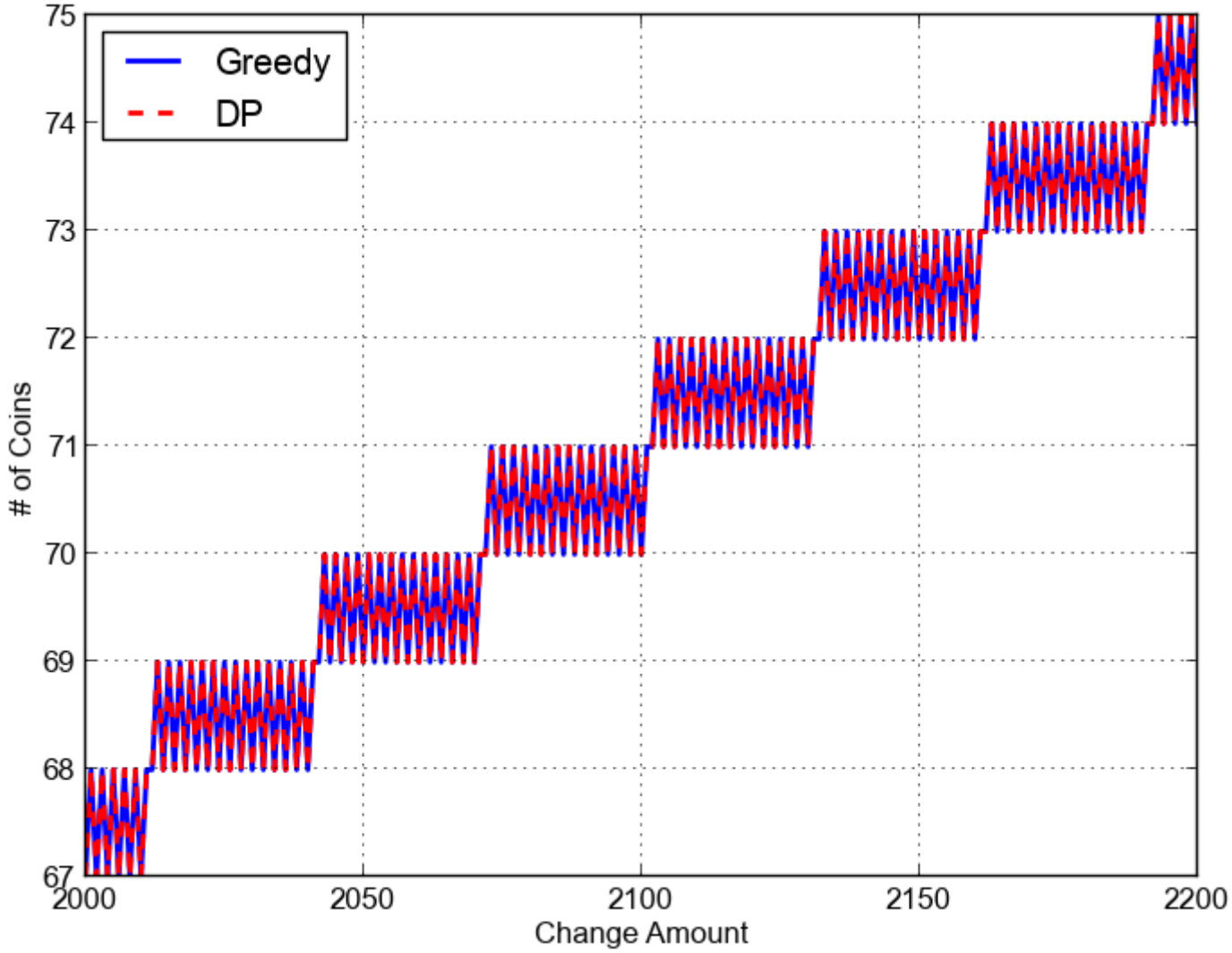
V=[1, 6, 13, 37, 150] with A=[1, 2, 3, ..., 30]



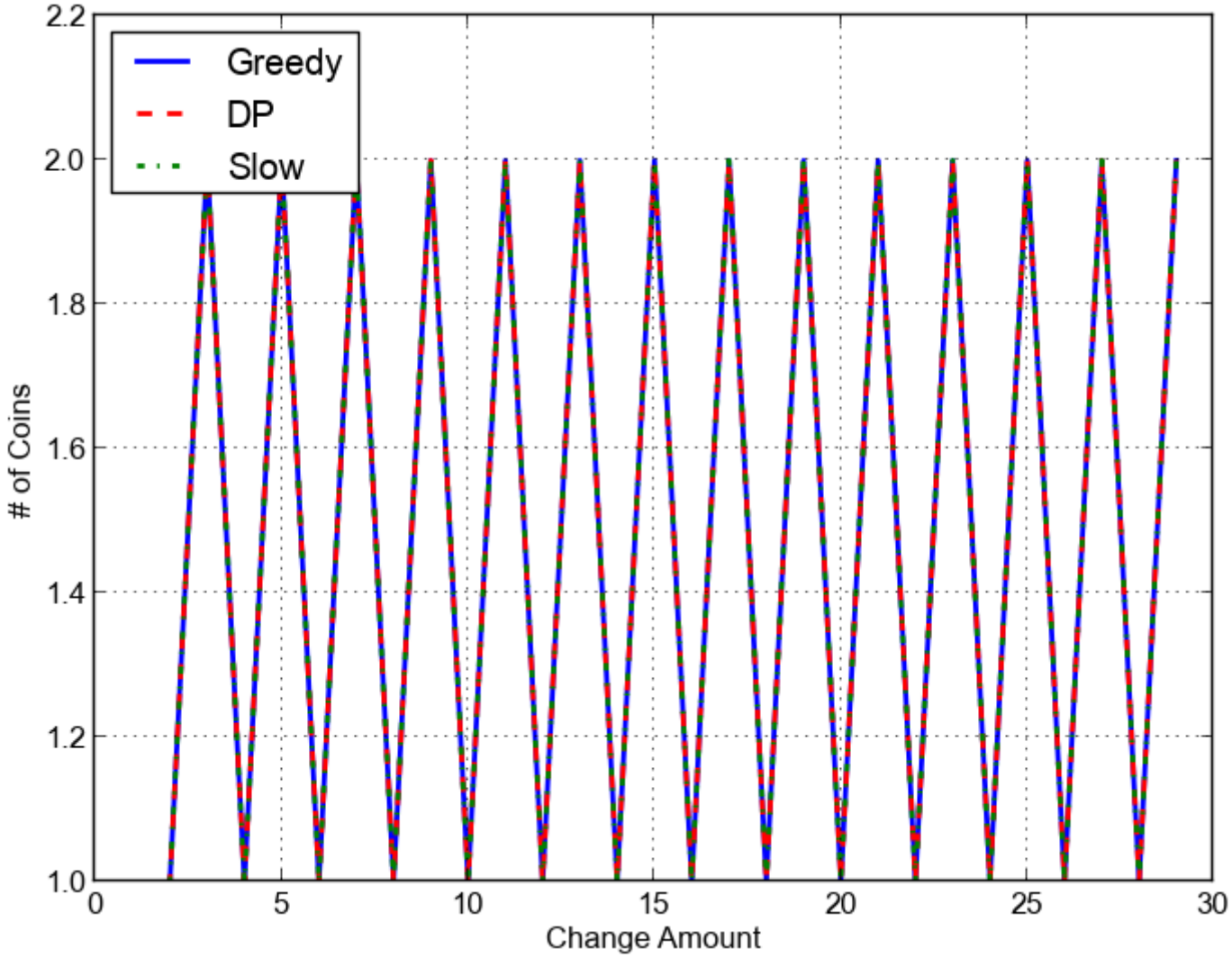
Note that the "changeslow" algorithm had to be run for $A < 30$ due to extremely long running times. The plots above suggest that the coin system [1,6,13,37,150] **IS NOT** canonical[1]. A coin system is canonical if the number of coins given in change by the greedy algorithm is optimal for all amounts. **Our observations show that greedy is NOT equal to DP (or Slow) *for all values of A that were plotted* for this coin system.** There are multiple values of A between 1 and 30 and between 2000 and 2200 for which greedy is sub-optimal.

Question 5

V=[1, 2, 4, 6, 8, ..., 30] with A=[2000, 2001, 2002, ..., 2200]



V=[1, 2, 4, 6, 8, ..., 30] with A=[1, 2, 3, ..., 30]

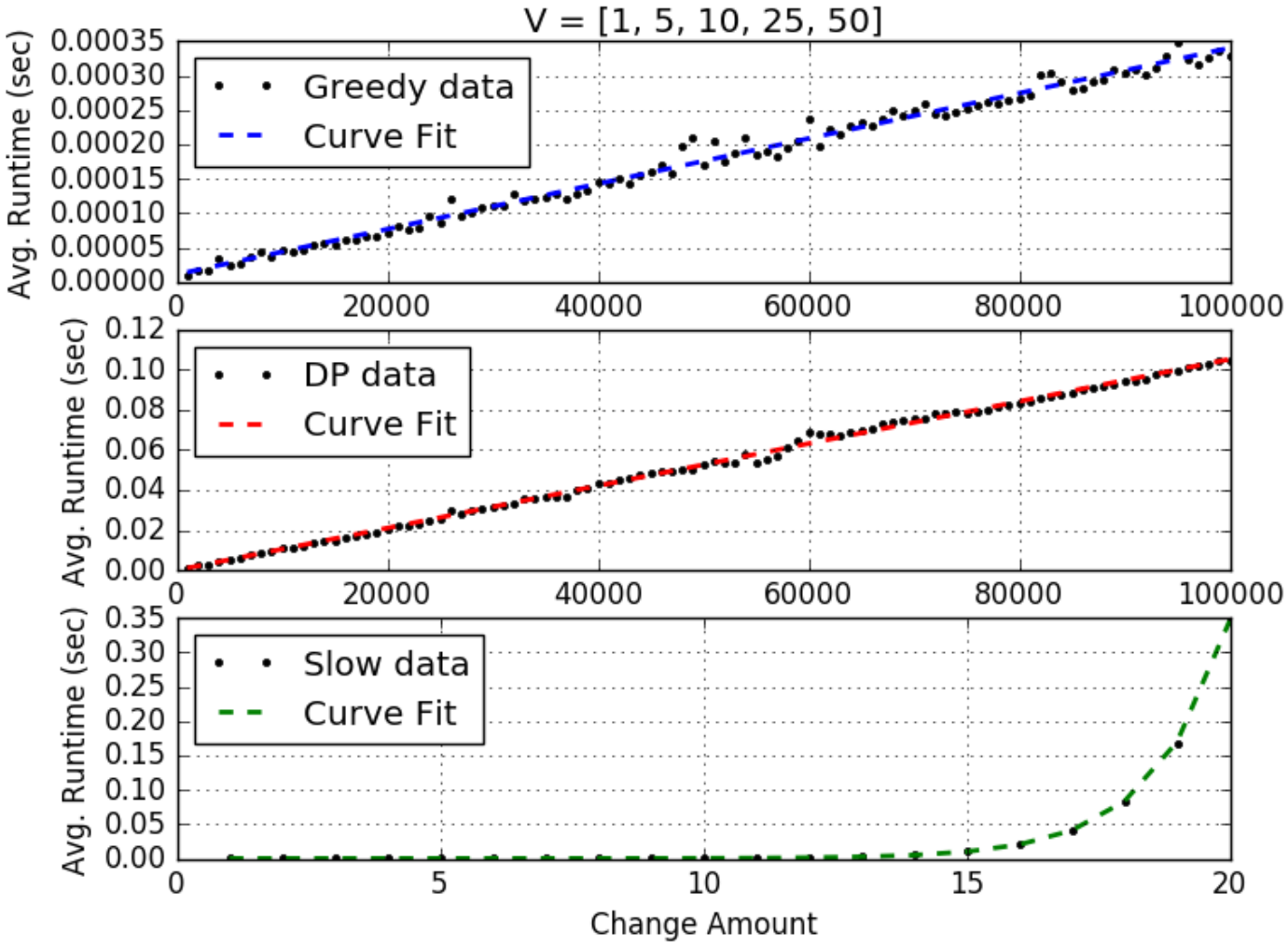


Note that the "changeslow" algorithm had to be run for $A < 30$ due to extremely long running times. The plots above suggest that the coin system [1,2,4,6,8,...,30] may be canonical[1]. A coin system is canonical if the number of coins given in change by the greedy algorithm is optimal for all amounts. **Our observations show that greedy is equal to DP (and Slow) *for all values of A that were plotted* for this coin system.**

Question 6

Question 3 Runtime plots

Runtime Plots for the Slow, DP, and Greedy Algorithms



Note that the "changeslow" algorithm had to be run for a smaller range of A due to extremely long running times.

Greedy Fit Curve Equation:

$$y = 3.2995 * 10^{-9} * x + 1.0700 * 10^{-5}$$

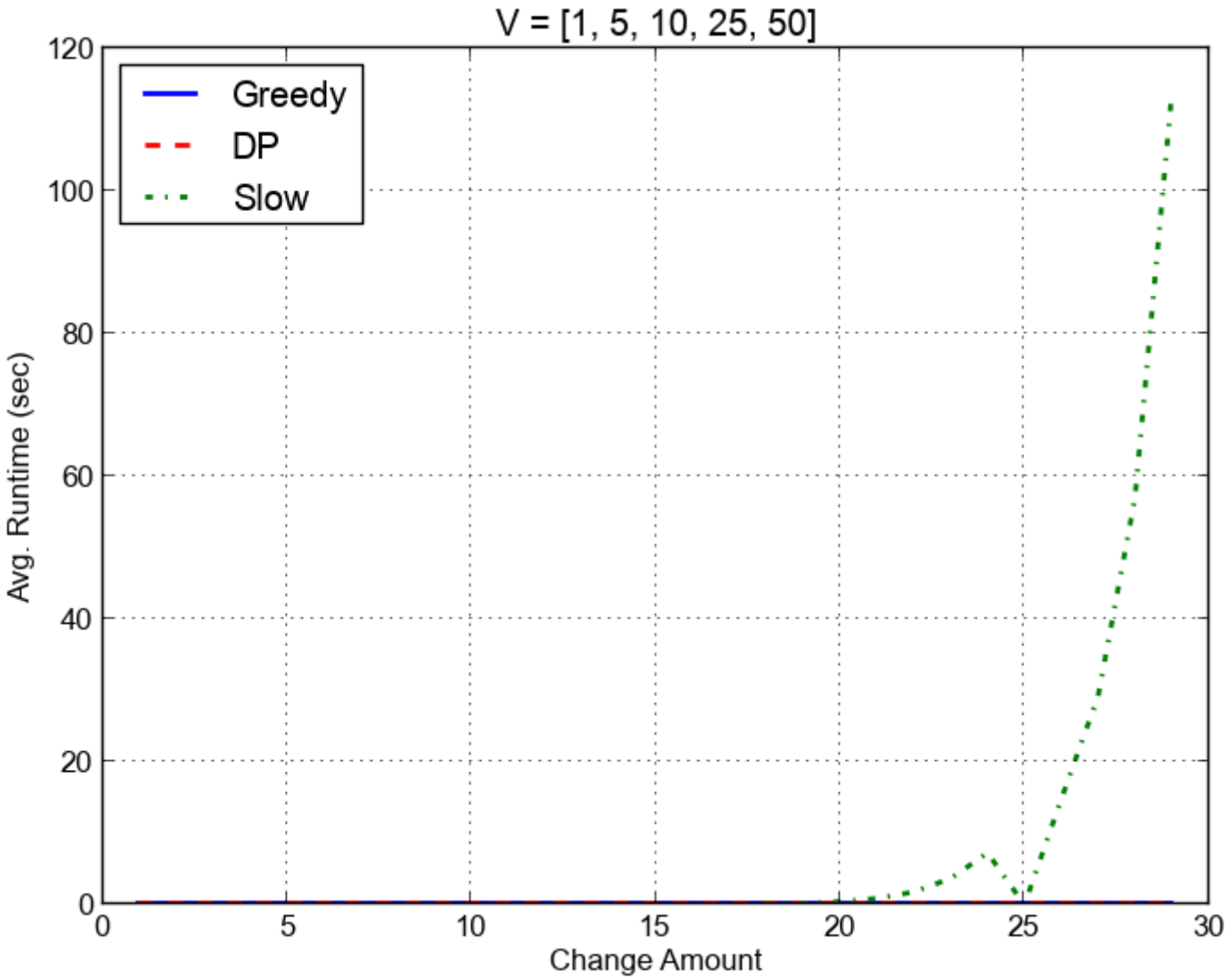
Dynamic Programming Fit Curve Equation:

$$y = 1.05375 * 10^{-6} * x - 3.46787 * 10^{-5}$$

Slow Fit Curve Equation:

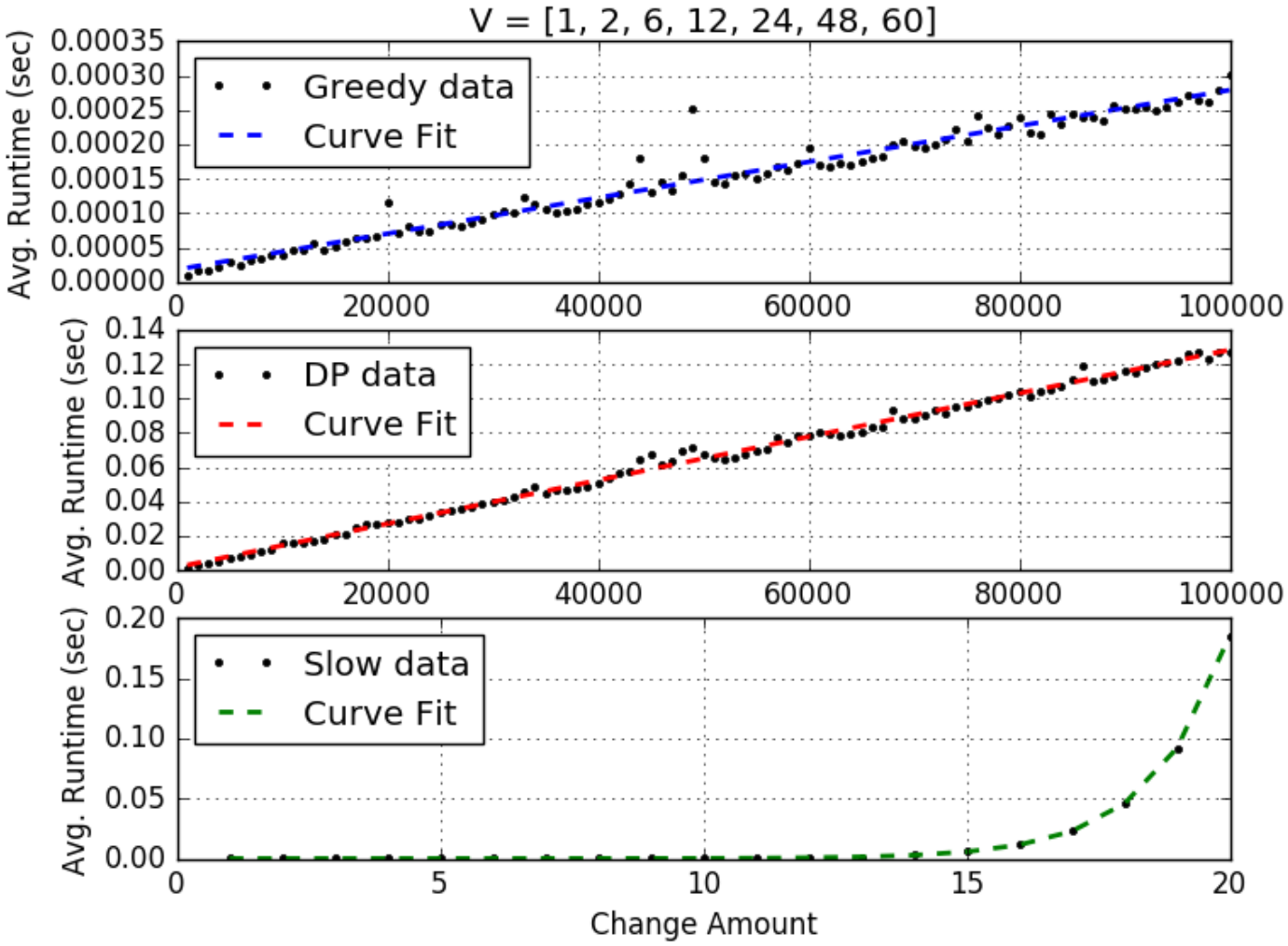
$$y = 2.18862 * 10^{-7} * e^{0.7136x} + 0.0002267$$

Runtime Plot for the Slow, DP, and Greedy Algorithms



Question 4 Runtime Plots

Runtime Plots for the Slow, DP, and Greedy Algorithms Part A



Note that the "changeslow" algorithm had to be run for a smaller range of A due to extremely long running times.

Greedy Fit Curve Equation:

$$y = 2.6146 * 10^{-9} * x + 1.77676 * 10^{-5}$$

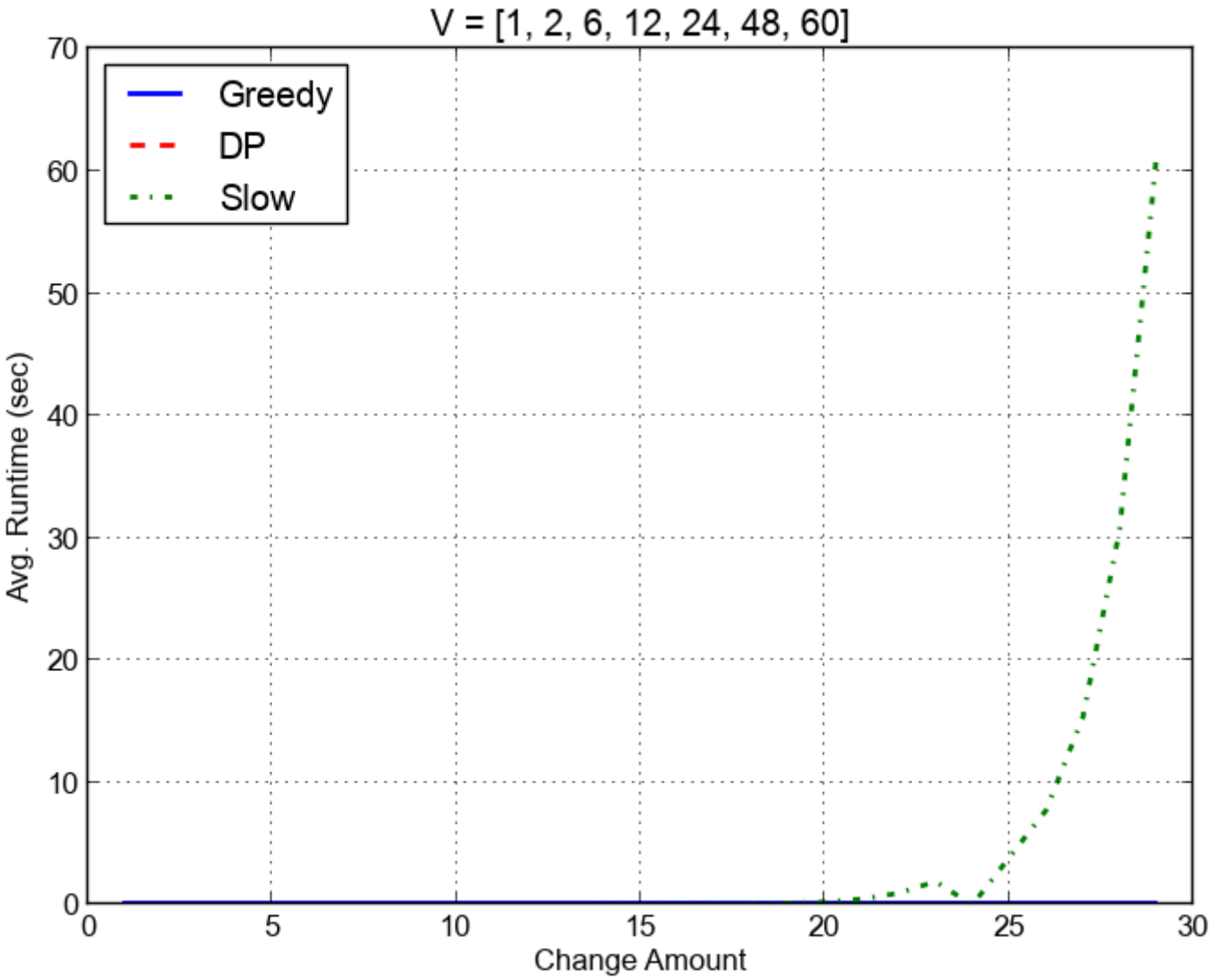
Dynamic Programming Fit Curve Equation:

$$y = 1.2664 * 10^{-6} * x + 0.001705$$

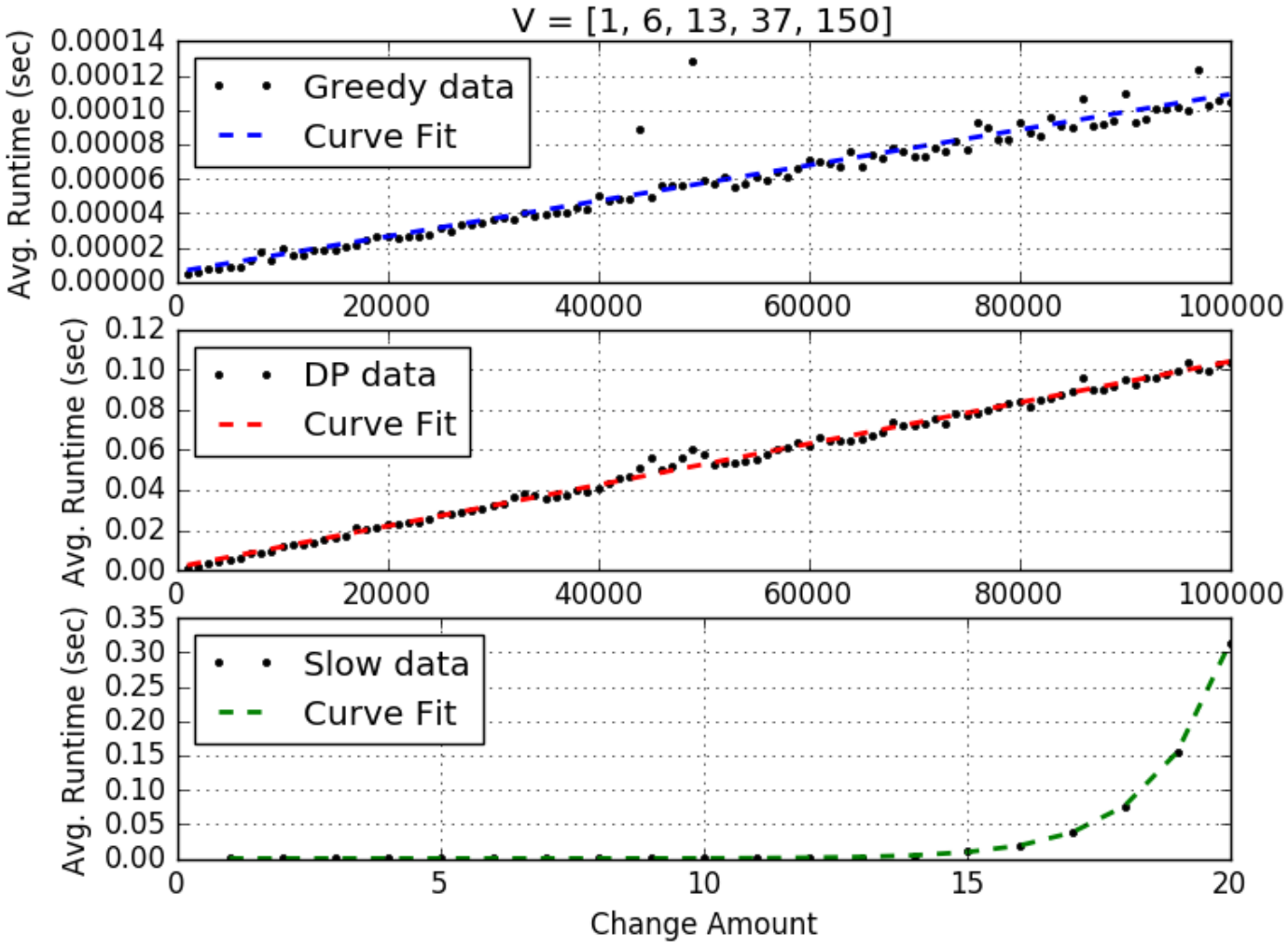
Slow Fit Curve Equation:

$$y = 1.51844 * 10^{-7} * e^{0.701x} + 5.2627 * 10^{-5}$$

Runtime Plot for the Slow, DP, and Greedy Algorithms Part A



Runtime Plots for the Slow, DP, and Greedy Algorithms Part B



Note that the "changeslow" algorithm had to be run for a smaller range of A due to extremely long running times.

Greedy Fit Curve Equation:

$$y = 1.0347 * 10^{-9} * x + 5.740 * 10^{-6}$$

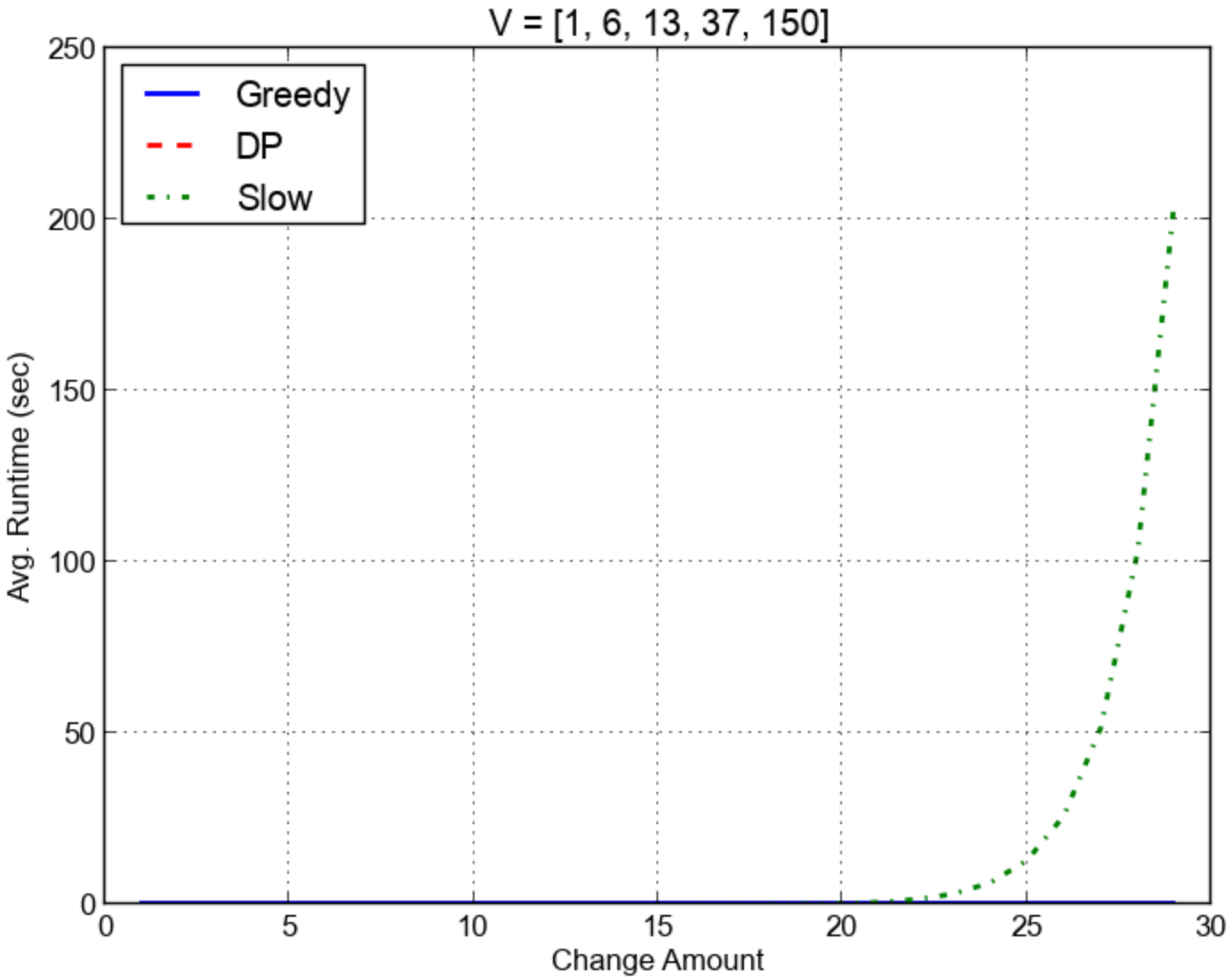
Dynamic Programming Fit Curve Equation:

$$y = 1.02725 * 10^{-6} * x + 0.00146$$

Slow Fit Curve Equation:

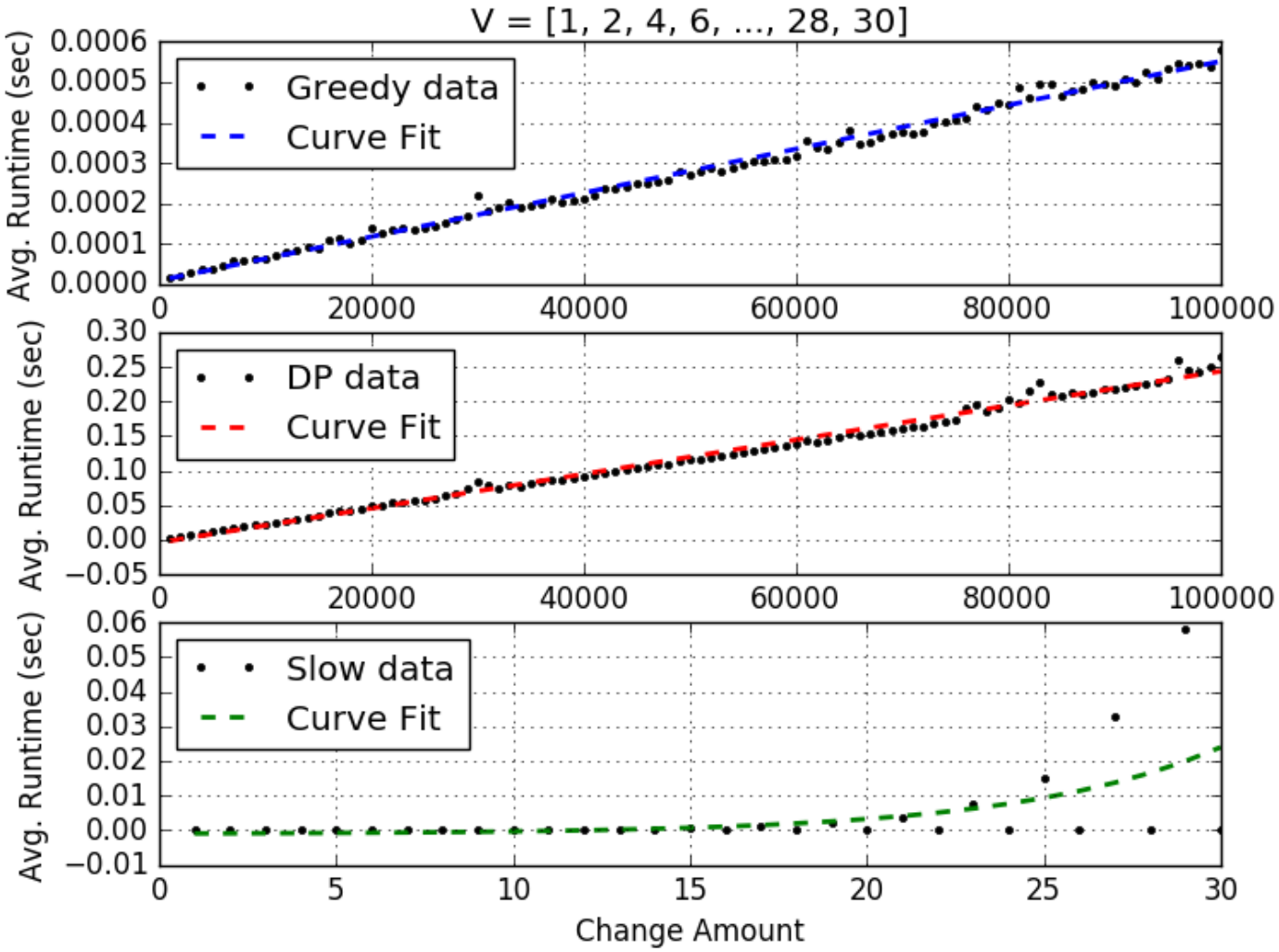
$$y = 2.252 * 10^{-7} * e^{0.7072x} + 0.000162$$

Runtime Plot for the Slow, DP, and Greedy Algorithms Part B



Question 5 Runtime Plots

Runtime Plots for the Slow, DP, and Greedy Algorithms



Greedy Fit Curve Equation:

$$y = 5.43669 * 10^{-9} * x + 8.425 * 10^{-6}$$

Dynamic Programming Fit Curve Equation:

$$y = 2.4784 * 10^{-6} * x - 0.0041$$

Slow Fit Curve Equation:

$$y = 0.00013 * e^{0.175x} - 0.001122$$

As we can see in the plots above the best fitting curves are linear for the greedy and dynamic programming cases and exponential for the slow case. The trends observed by the fit lines match what we would expect from our runtime analysis whereby the greedy has a theoretical runtime of $O(n + k)$, the dp has a theoretical runtime of $O(n * k)$ and the slow has a theoretical runtime of $O(3^n)$. The greedy and dynamic programming runtimes are observed to be linear because k (length of the coin denominations array) is held constant while n (the change amount) is varying, therefore the experimental curves are still consistent the theoretical runtimes. In Question 5, the curve fit of the slow algorithm is slightly degraded by the fact that all even values of A bypass the exponential behavior due to the change being made in a single coin. This is evident by the run time for even values of A being approximately 0. If a curve fit was drawn through the odd values of A, it would match the behavior from Q3/Q4a/Q4b.

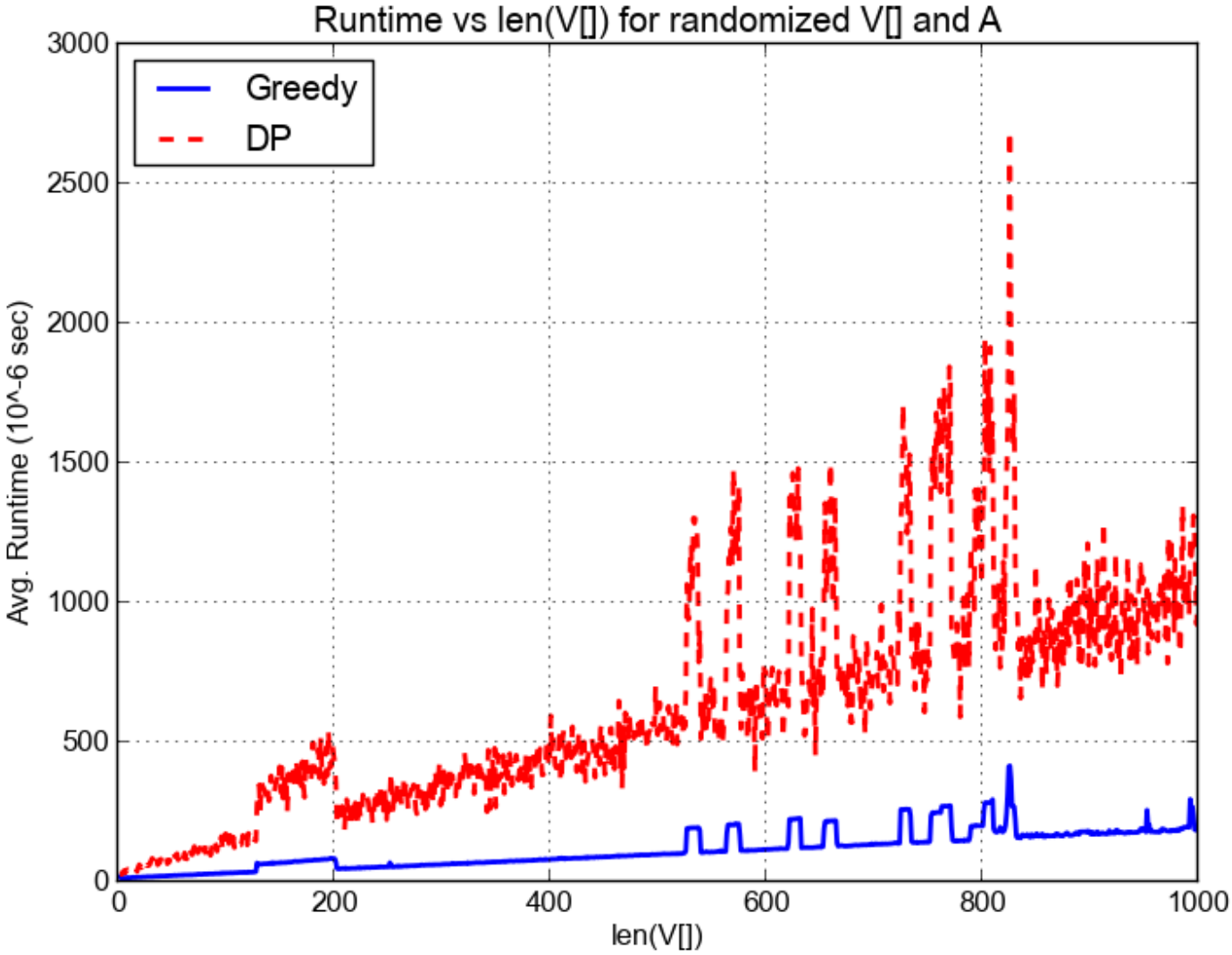
Question 7

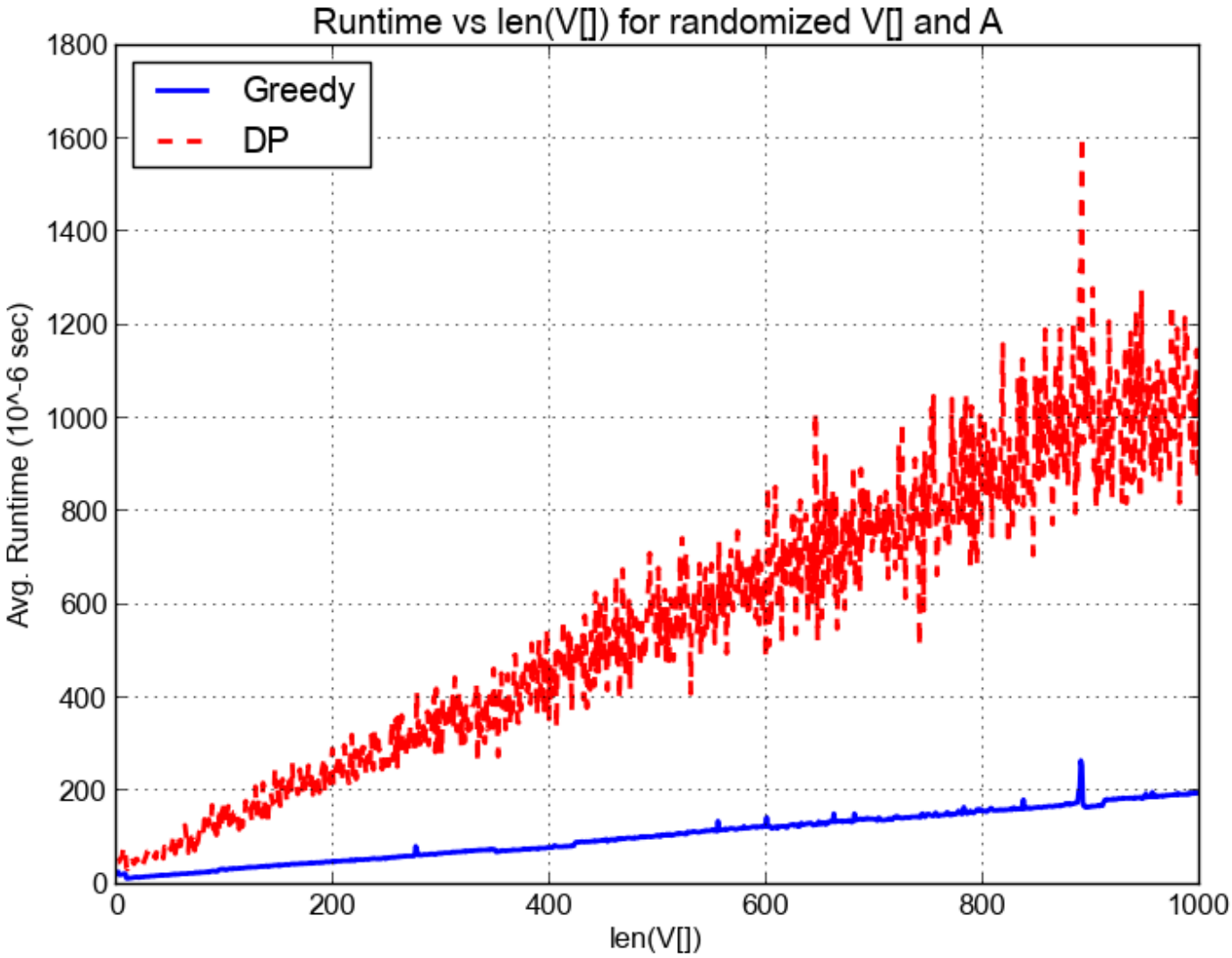
This question requires us to think about running times as a function of the size of the coin array V. This question is more complex than it first appears because there are a number of different corner cases. Many of these are dependent upon the relationship between the change value, A, and V.

Take the case where A = V[len(V)-1]. An example would be A=50 with V=[1,5,10,25,50]. In this case greedy will run in O(1) time because it will take the very first coin it looks at and the problem will be solved. We could modify the coin array such that V=[1,2,3,...,48,49,50], increasing the size of V by a factor of 10, and have absolutely no effect on the runtime of the greedy algorithm.

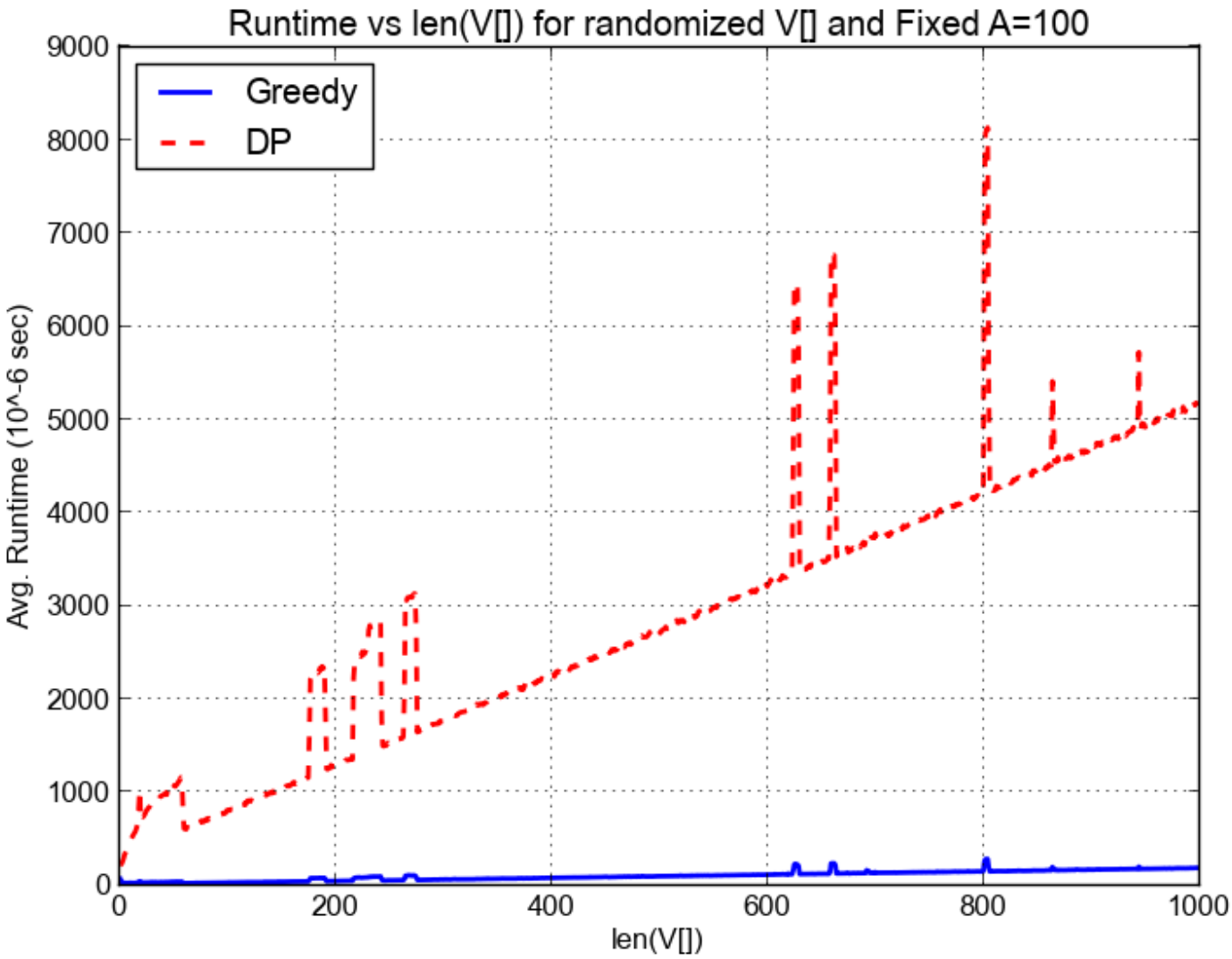
On the opposite end of the spectrum would A=53 with V=[1,5,10,25,50]. In this case, because 53 is a prime number, greedy will run in O(V + A) time because the problem can only be solved with a 50 cent coin and 3 one cent coins (value of 1). In this case modification of the coin array such that V=[1,2,3,...,48,49,50], increasing the size of V by a factor of 10, will have an effect on the runtime of the greedy algorithm.

The complex relationship between change A and coin array V is on display in the following two plots which were generated with identical code. Coin arrays of sizes 1 to 1000 (with coin values 1 to <= 5000) and change values from 1 to 30 were randomnly generated and run through both the greedy and DP algorithms. The entire exercise was repeated 10 times and the average runtimes are plotted below. Both plots display the mostly linear relationship between length of V and runtime, the first plot exhibits quite a bit of scatter from about 100 < len(V) < 200 and again from about 550 < len(V) < 850. The exact arrays and change values were not stored so we can only speculate on the nature of the relationships between A and V in these noisy regions.

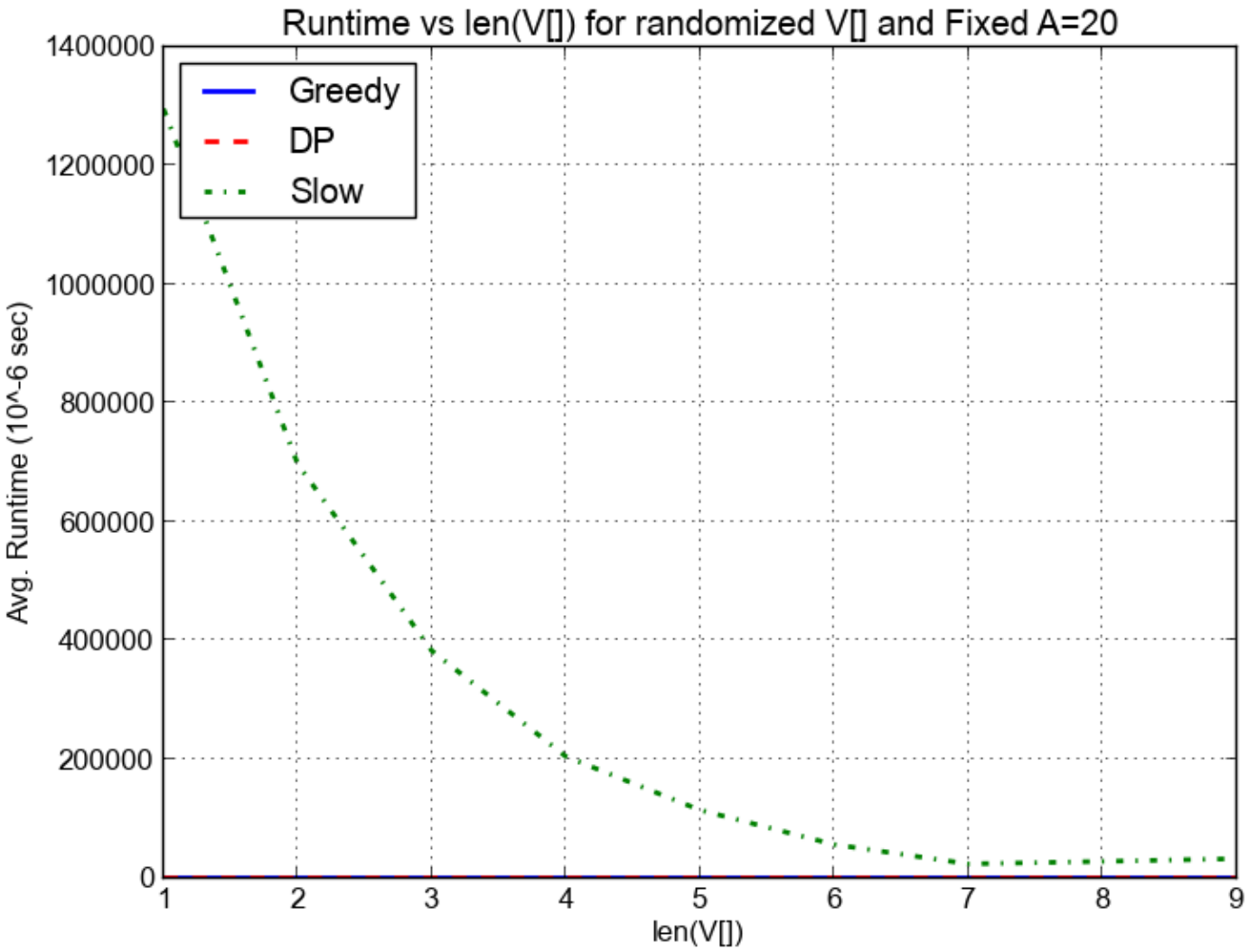

Runtime vs len(V[]) for randomized V[] and A
Greedy
DP
Avg. Runtime (10^-6 sec)
len(V[])
0
500
1000
1500
2000
2500
3000
0
200
400
600
800
1000



We also spent some time investigating whether any noise was being introduced by our letting change A randomnly vary between 1 and 30. So we ran the experiment again. We tried various fixed values of A and observed two things. The first observation is that fixing the value of A dramatatically cuts down on the noise in the plot. Secondly, a larger value of A only changes the plot in that it changes the slope of the DP curve. The effect on the greedy curve is minimal, especially for larger values of A, because it starts to dominate over len(V). One such plot with fixed value A=100 is shown below.



Finally, we wanted to create at least one plot looking at the slow change algorithm. We used the same code as above but fixed change A at 20 for runtime purposes. Increasing A to a higher value was uneccessary to understanding how the slowchange algorithm responds to increasing size of coin array V. The plot below shows that the slowchange algorithm is actually worst for the smallest size of V and dramatically and quickly improves as size of V increases. This behavior is due to the fact that as the size of V increases, the number of base cases increases. The base cases allow the algorithm to terminate without entering further recursive calls. In the case where V = [1], the recursion must run all the way to the bottom. In essence, adding elements to V decreases the "full-ness" of the recursion tree.



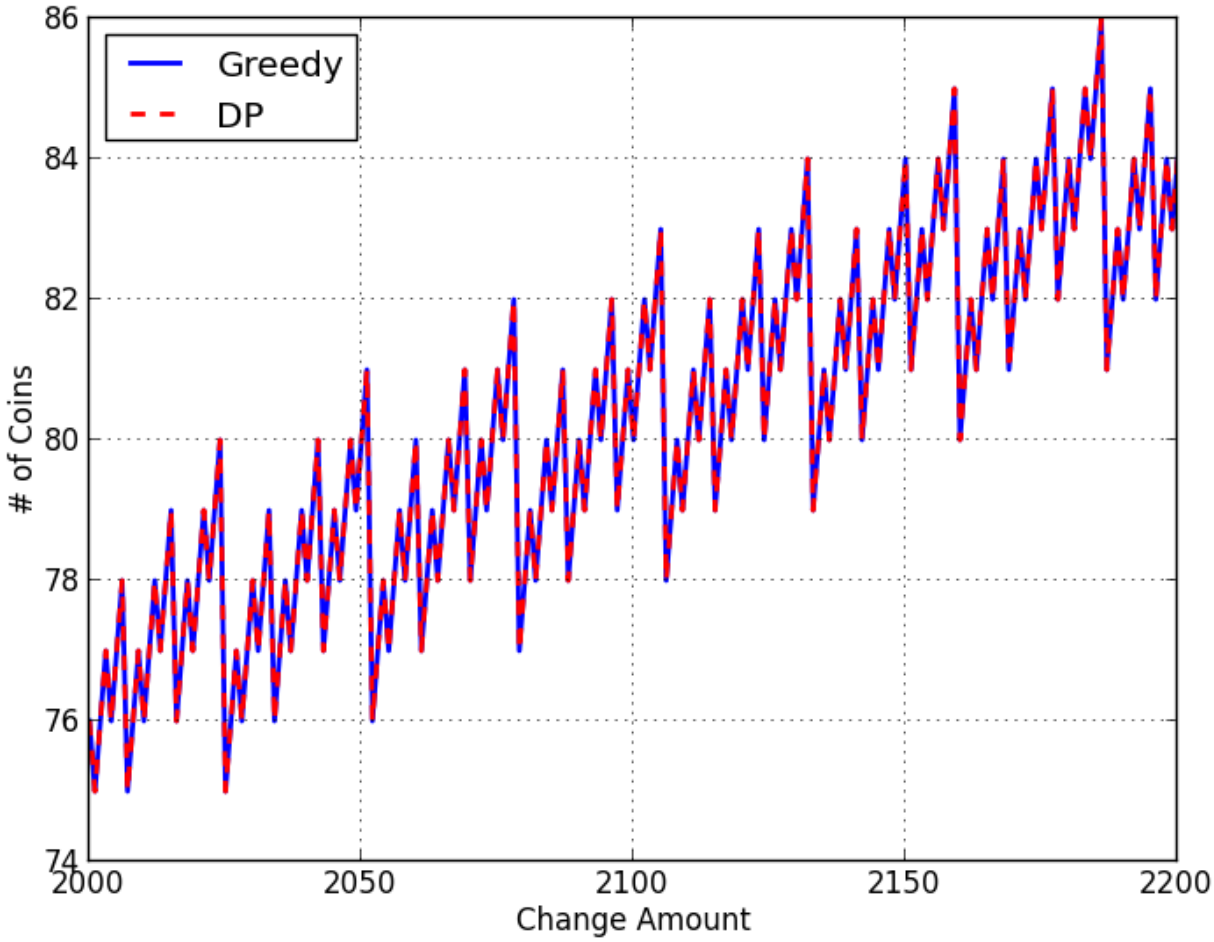
Question 8

Suppose you are living in a country where coins have values that are powers of p, i.e. $V = [1, 3, 9, 27]$. How do you think the dynamic programming and greedy approaches would compare? Explain.

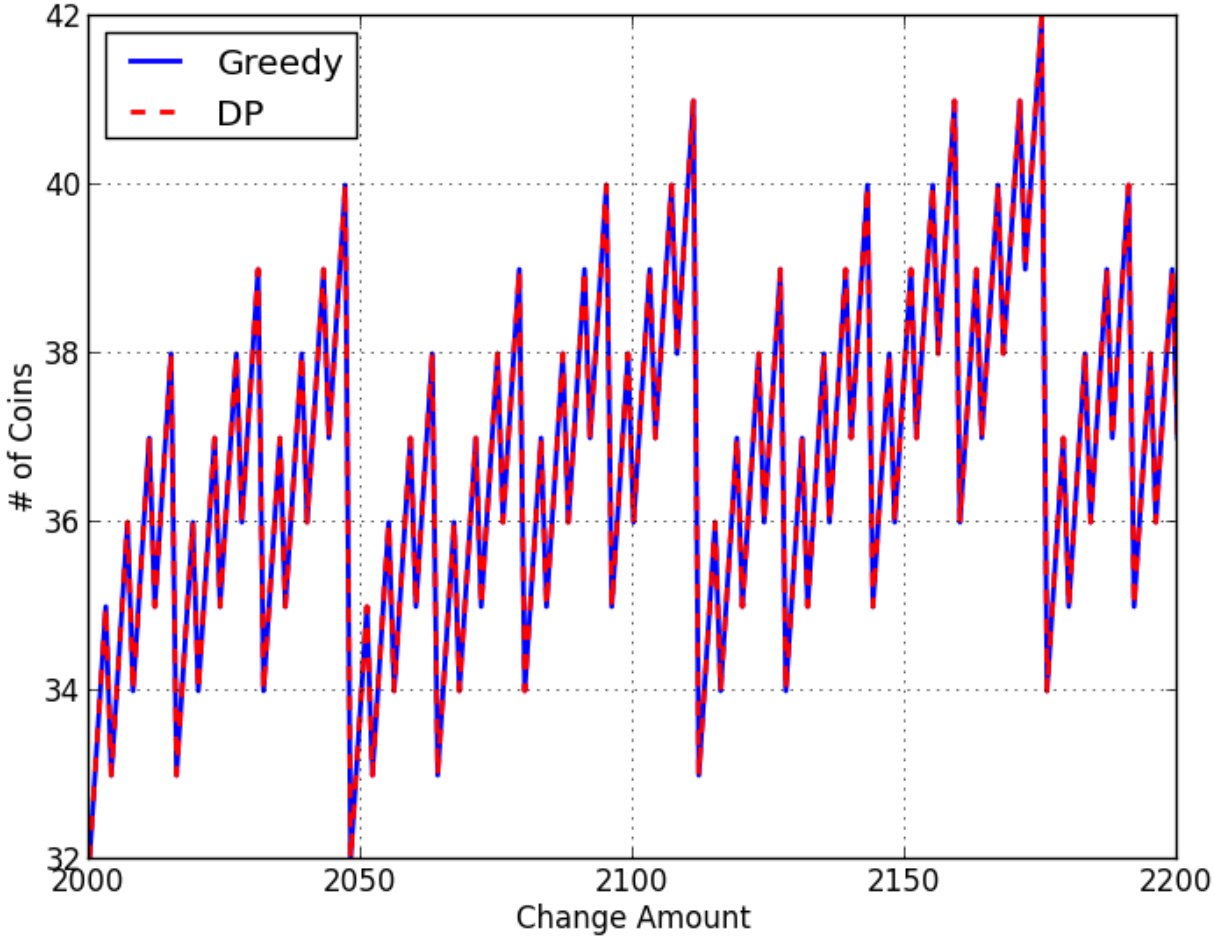
The dynamic programming (dp) implementation will always yield the optimal solution (minimum number of coins to make change) while the greedy implementation will only produce an optimal solution under certain conditions.

Below are plots comparing the greedy and dp solutions for making change using various values of p.

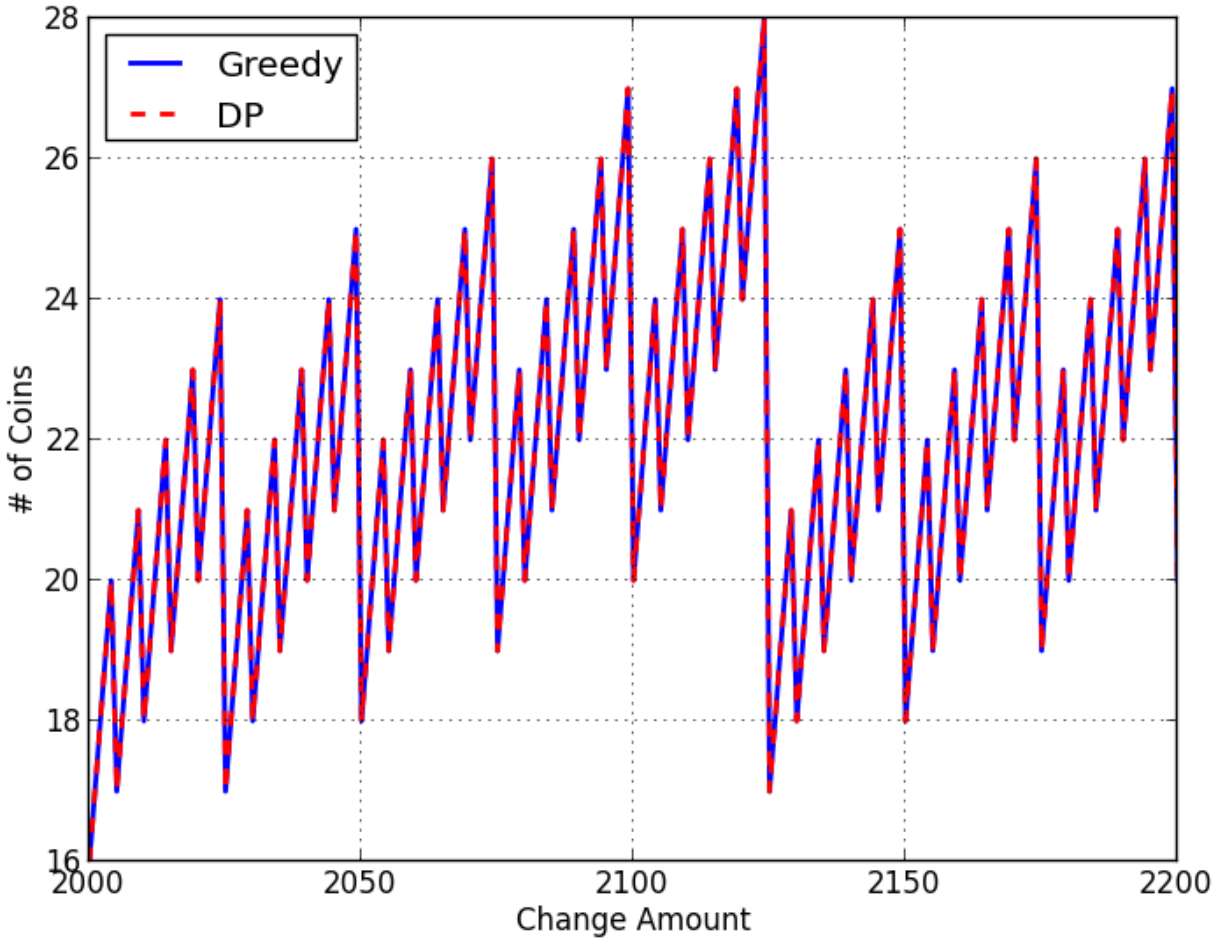
When $p = 3$:


Greedy
DP
# of Coins
Change Amount
86
84
82
80
78
76
74
2000
2050
2100
2150
2200

When $p = 4$:


Greedy
DP
# of Coins
Change Amount
42
40
38
36
34
32
2000
2050
2100
2150
2200

When $p = 5$:



Based on the experimental results above, it seems likely that the greedy solution is equivalent to the optimal solution when the coin denominations are different powers of some integer p.

Let's attempt to prove that fact. Consider the case when $V = [p^0] = [1]$. In this case, it's obvious that the greedy solution is equivalent to the optimal solution, since there is only one coin to choose from.

Now consider the case when $V = [p^0, p^1, \ldots, p^k]$ for some $k \geq 1$. If we wish to make change for amount A, there are 3 cases to consider:

1. A is equal to some power of p
2. $A < p^k$
3. $A > p^k$

In the first case, the greedy algorithm will correctly choose the optimal choice of using a single coin. In the second case, we can use the optimal solution from the case with coins $\hat{V} = [p^0, p^1, \ldots, p^{k-1}]$. In the third case, the best we can do is use p^k coins until the point at which we reduce the problem to the second case. The alternative in the third case would be to use p^n coins of denomination p^{k-n} for some $n \leq k$. This is inefficient compared to using a single p^k coin.

These cases will mimic the greedy algorithms logic -- using the highest valued usable coin until the change is made.

References

[1] Xuan Cai (2009). "Canonical Coin Systems for CHANGE-MAKING Problems". Proceedings of the Ninth International Conference on Hybrid Intelligent Systems.