Keisha Arnold, Jacob Karcz, David Haskell
Group 11
CS 325: Project 2

1.1. Implement a Brute Force or Divide and Conquer Algorithm:
The following divide and conquer algorithm is based upon what is presented at:
http://www.geeksforgeeks.org/find-minimum-number-of-coins-that-make-a-change/

```
int changeslow(V[0...n-1], Vsize,  A ) {
        int k = A;                  //the kth value is the amount we want
        result, solution_1, solution_2
                structs initialized with:  minCoins = INT_MAX &
                                           array of ints, change[Vsize] initialized to 0s

        //base case
        // if we we have a coin in V worth k, return it
        for (each i in V)
                If (V[i] == k)
                        minCoins = 1;
                        change[i] = 1;
                        struct result = (minCoins, change[]);
                        return result;

        //else try all combinations of "i" and "k - i" coins
        for ( i = 1 to i = k - 1)
                solution_1 =  changeslow(V, Vsize,  i )
                solution_2 =  changeslow(V, Vsize,  k - i )
                int newCoinCount  = solution_1->minCoins + solution_2->minCoins
                //pick the best solution so far
                if (newCoinCount < result->minCoins )
                        result->minCoins = newCoinCount
                        for (i in result->change[])
                                change[i] = solution_1->change[i] + solution_2->change[i]
        return result;
}
```

Theoretical asymptotic running time:
In this divide and conquer algorithm, we would have an exponential worst-case running time of $O(A^n)$, where A is the coin amount and n is the number of coins. It uses a recursive technique to generate all combinations of coin frequencies and every coin denomination V_i could have at most A/V_i values. Therefore, the number of possible combinations is:
$A/V_1 * A/V_2 * A/V_3 *...* A/V_n = A^n / V_1 *V_2 * V_3 * V_n$.

1.2. Greedy Algorithm:

// Greedy method selects the largest possible denomination first and keeps adding the next largest denominations until we reach our desired amount of change

```
int changegreedy(V[0...n-1], A]) {
        int ans[n];
        int ansCount = 0;
        for(i = n-1 down to i >=0) {     //start with the largest denomination
                while (A >= V[i]) {      // while amount is greater than the denomination
                        A = A - V[i];    // subtract that denomination from our total amount
                        ans[ansCount] = V[i];
                        ansCount++;
                }
        }
        return (ans[ ], ansCount-1);
}
```

Theoretical asymptotic running time:

This greedy algorithm uses a simple select function that commits to using a the largest coin that is equal to or less than the target value. It will not always produce an optimal solution depending on the coin denominations given because, being “greedy,” it takes advantage of locally optimal choices that allow it to ignore possible ways of getting to the overall optimal solution (see #7 for a detailed explanation).

For the best case run time, A (the target value) would happen to be equal to one of the values in the given set V. In this case, run time would be nearly immediate T(n) = c. The worst case run time would occur if the set V contained the single element V= [1]. This would require the algorithm to subtract 1 from the value A, n times. This gives a run time of T(n) = n. This behavior occurs, to a lesser extent, when A is much greater than the largest coin denomination. Therefore, this greedy algorithm will have an asymptotic linear running time of O(n).

This drastic difference between best case and worst case is shown on the plots of runtime as a function of A(target value). The plots show a number of increasing trends, appearing as separate increasing lines. The base value of each of these lines in the almost immediate solution where A is equal to a certain given denomination. It can also be seen that the number of these separate “increasing lines” is equal to the number of coin denominations that were given in the coin set.

1.3. Dynamic Programming:

// Bottom up method

Adapted from: https://www.youtube.com/watch?v=NJuKJ8sasGk

```
int changedp(V[0...n-1], A]) {
        // make 2 arrays of length A + 1
        int T[A + 1];           // to keep track of minimum number of coins
        int R[A + 1];           // to keep track of which coins were used
        T[0] = 0;               // it takes 0 coins to form a total of 0

        // initialize the arrays
        for(i = 1 to i <= A) {
                T[i] = INT_MAX;
                R[i] = -1;
        }

        for(j = 0 to j < n) {           // iterates through the coins in V
                for(i = 1 to i <= A) {  // iterates from 1 through A
                        if(i >= V[j]) {
                                if(T[i - V[j] + 1 < T[i]) {             // pick the jth coin
                                        T[i] = 1 + T[i - coins[j]];     // # of jth coins
                                        R[i] = j;               // keep track of jth coin in R[i]
                                }
                        }
                }
        }
        return T[A];
}
```

Theoretical asymptotic running time:

The above algorithm uses dynamic programming by using a bottom-up approach to find the minimum number of coins needed to make change for amount A. The running time is pseudo-polynomial O(A*n), since it depends on the size of A respective to n.

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

The changedp algorithm above, uses a bottom up approach to solve our problem. Instead of using a 2-D array as our "table" we used (2) 1-D arrays. The easiest way to justify this is to walk through an example:

V[7, 2, 3, 6] // Set of coins
A = 13 // Total amount of change

(2) 1-D arrays of length A + 1
Array T[ ] will keep track of the minimum number of coins it take to form a certain value.
Array R[ ] will keep track of which coin is used to form its corresponding value.

First we initialize T[0] to 0 since it takes 0 coins to form a total of 0.
Next we initialize the rest of the array T[1...13] to INT_MAX because we don't yet know how many coins it will take to form that total.
Similarly, we initialize all the elements in R[0...13] to -1 because we don't yet know which coins will be used to form that total.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

Array T[ ]

0	1	2	3	4	5	6	7	8	9	10	11	12	13
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Array R[ ]

This is the formula we will follow in our algorithm:

For all j from 0...coins:
V[i] = min(V[i] , 1 + V[i - coins[j]])
In other words...
Minimum # of coins it takes to form total i = the minimum of either (not picking the jth coin , or picking the jth coin)

- We start from the 0th coin, which is 7, so j = 0 and i = 1
  - Total 1 is less than 7, so no matter what we cannot form 1 with just 1 coin that has a denomination of 7. Therefore T[1] remains ∞. The same goes for T[2] through T[6] because we cannot form those totals with a coin of denomination 7.

V[7, 2, 3, 6] // Set of coins

A = 13 // Total amount of change

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

Array T[ ]

0	1	2	3	4	5	6	7	8	9	10	11	12	13
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Array R[ ]

- Now we are at j = 0 and i = 7, where we enter this if-statement -> if(i >= V[j])... since i = 7 and V[j] = 7. Here we can apply our formula:

  min(∞ , 1 + T[7 - 7])

  = min(∞ , 1 + T[0])

  = min(∞ , 1 + 0)

  = 1

  The result of our formula is that we picked the jth coin so T[i] = 1 + T[i - coins[j]] and R[i] = j. Or in other words, we replace ∞ at T[7] with 1 and replace the -1 at R[7] with 0.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	∞	∞	∞	∞	∞	∞	1	∞	∞	∞	∞	∞	∞

Array T[ ]

0	1	2	3	4	5	6	7	8	9	10	11	12	13
-1	-1	-1	-1	-1	-1	-1	0	-1	-1	-1	-1	-1	-1

Array R[ ]

- Now we are at j = 0 and i = 8, where we enter this if-statement -> if(i >= V[j])... since i = 8 and V[j] = 7. Here we can apply our formula:

  min(∞ , 1 + T[8 - 7])

  = min(∞ , 1 + T[1])

  = min(∞ , 1 + ∞)

  = ∞

  The result of our formula is that we did not pick the jth coin, so we do not enter this if-statement -> if(T[i - V[j] + 1 < T[i])... and the values at T[8] and R[8] remain the same.

- If we apply our formula for i = 9 through i = 13, we see that we do not pick the jth coin, so the values at T[i] and R[i] remain the same.

- Now we are at the 1st indexed coin, which is 2, so j = 1 and i = 1.
    - As before we do not enter this if-statement -> if(i >= V[j])... because total 1 is less than 2, so no matter what we cannot form 1 with just 1 coin that has a denomination of 2. Therefore T[1] remains ∞.

V[7, 2, 3, 6] // Set of coins
A = 13 // Total amount of change

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	∞	∞	∞	∞	∞	∞	1	∞	∞	∞	∞	∞	∞

Array T[ ]

0	1	2	3	4	5	6	7	8	9	10	11	12	13
-1	-1	-1	-1	-1	-1	-1	0	-1	-1	-1	-1	-1	-1

Array R[ ]

- Now we are at j = 0 and i = 2, where we enter this if-statement -> if(i >= V[j])... since i = 2 and V[j] = 2. Here we can apply our formula:

  min(∞ , 1 + T[2 - 2])

  = min(∞ , 1 + T[0])

  = min(∞ , 1 + 0)

  = 1

  The result of our formula is that we picked the jth coin so T[i] = 1 + T[i - coins[j]] and R[i] = j. Or in other words, we replace ∞ at T[2] with 1 and replace the -1 at R[2] with 1.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	∞	1	∞	∞	∞	∞	1	∞	∞	∞	∞	∞	∞

Array T[ ]

0	1	2	3	4	5	6	7	8	9	10	11	12	13
-1	-1	1	-1	-1	-1	-1	0	-1	-1	-1	-1	-1	-1

Array R[ ]

- At j = 0 and i = 3, we have:

  min(∞ , 1 + T[3 - 2]) = min(∞ , 1 + T[1]) = min(∞ , 1 + ∞) = ∞

The result of our formula is that we did not pick the jth coin, so we do not enter this if-statement -> if(T[i - V[j] + 1 < T[i])... and the values at T[3] and R[3] remain the same.

- At j = 0 and i = 4, we have:

  min(∞ , 1 + T[4 - 2]) = min(∞ , 1 + T[2]) = min(∞ , 1 + 1) = 2

  The result of our formula is that we picked the jth coin so T[i] = 1 + T[i - coins[j]] and R[i] = j. Or in other words, we replace ∞ at T[4] with 2 and replace the -1 at R[4] with 1.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	∞	1	∞	2	∞	∞	1	∞	∞	∞	∞	∞	∞

Array T[ ]

0	1	2	3	4	5	6	7	8	9	10	11	12	13
-1	-1	1	-1	1	-1	-1	0	-1	-1	-1	-1	-1	-1

Array R[ ]

- We follow this same process for the rest of the 1st (jth) indexed coin until we reach n and our two arrays should look like this (the red indicates the values we changed from the previous iteration):
  - At this point, we can see that it takes 4 coins to make a total value of 13 from a set of 2 coins with denominations of 7 and 2.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	∞	1	∞	2	∞	3	1	4	2	5	3	6	4

Array T[ ]

0	1	2	3	4	5	6	7	8	9	10	11	12	13
-1	-1	1	-1	1	-1	1	0	1	1	1	1	1	1

Array R[ ]

- Now we are at the 2nd indexed coin, which is 3, so j = 2 and i = 1.
  - As before we do not enter this if-statement -> if(i >= V[j])... because total 1 is less than 3, so no matter what we cannot form 1 with just 1 coin that has a denomination of 3. Therefore T[1] remains ∞. Same for i = 2.

V[7, 2, 3, 6] // Set of coins
A = 13 // Total amount of change

0	1	2	3	4	5	6	7	8	9	10	11	12	13

0	∞	1	∞	2	∞	3	1	4	2	5	3	6	4

Array T[ ]

0	1	2	3	4	5	6	7	8	9	10	11	12	13
-1	-1	1	-1	1	-1	1	0	1	1	1	1	1	1

Array R[ ]

- We follow this same process as before for the rest of the 2nd (jth) indexed coin until we reach n and our two arrays should look like this (the red indicates the values we changed from the previous iteration):
- For example, at j = 2 and i = 13, where we enter this if-statement -> if(i >= V[j])... since i = 13 and V[j] = 3. Here we can apply our formula:

  min(4 , 1 + T[13 - 3])

  = min(4 , 1 + T[10])

  = min(4 , 1 + 2)

  = 3

  The result of our formula is that we picked the jth coin so T[i] = 1 + T[i - coins[j]] and R[i] = j. Or in other words, we replace 4 at T[13] with 3 and replace the 1 at R[13] with 2.
  - At this point, we can see that it takes 3 coins to make a total value of 13 from a set of 3 coins with denominations of 7, 2 and 3.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	∞	1	1	2	2	2	1	3	2	2	3	3	3

Array T[ ]

0	1	2	3	4	5	6	7	8	9	10	11	12	13
-1	-1	1	2	1	2	2	0	2	1	2	1	2	2

Array R[ ]

- Now we are at the 3rd indexed coin, which is 6, so j = 3 and i = 1.
  - As before we do not enter this if-statement -> if(i >= V[j])... because total 1 is less than 6, so no matter what we cannot form 1 with just 1 coin that has a denomination of 3. Therefore T[1] remains ∞. Same for i = 2 through i = 5.

V[7, 2, 3, 6] // Set of coins
A = 13 // Total amount of change

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	∞	1	1	2	2	2	1	3	2	2	3	3	3

Array T[ ]

0	1	2	3	4	5	6	7	8	9	10	11	12	13
-1	-1	1	2	1	2	2	0	2	1	2	1	2	2

Array R[ ]

- We follow this same process as before for the rest of the 3rd (jth) indexed coin until we reach n and our two arrays should look like this (the red indicates the values we changed from the previous iteration):

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	∞	1	1	2	2	1	1	2	2	2	3	2	2

Array T[ ]

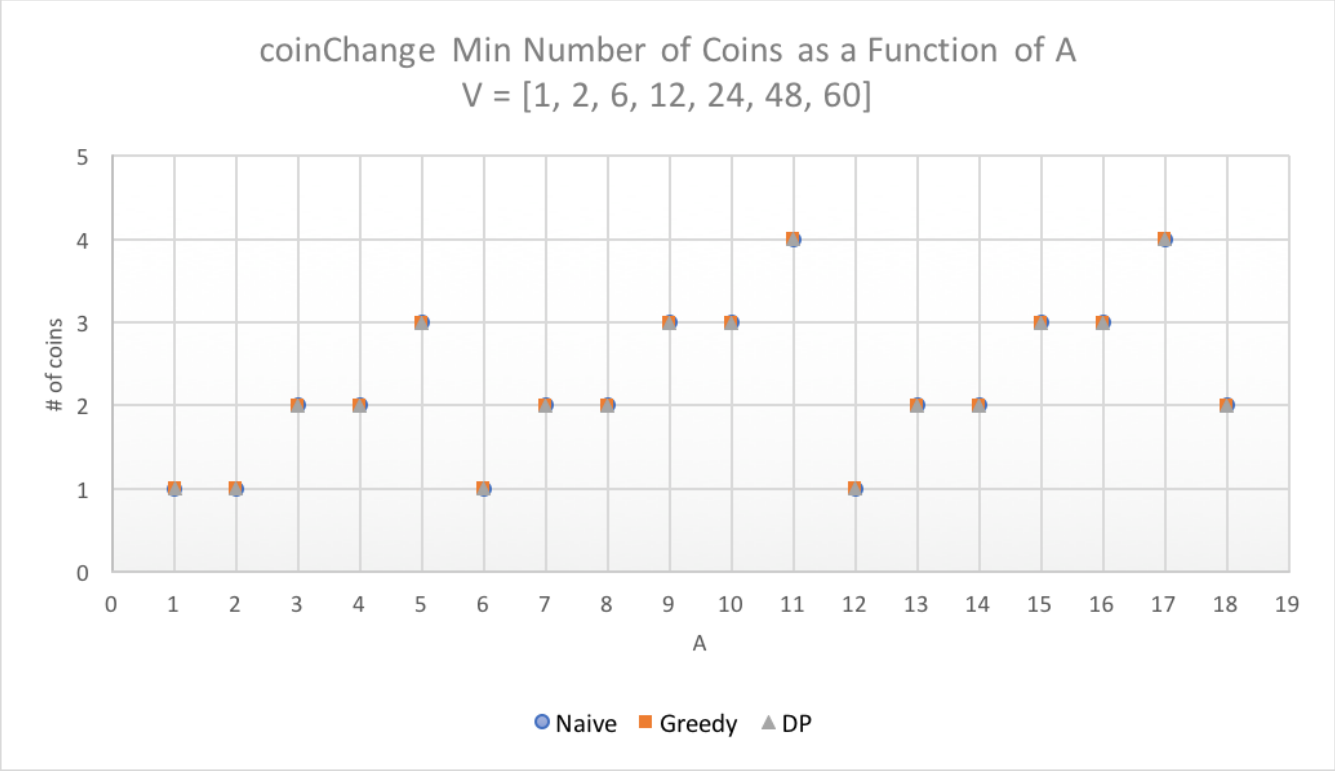
0	1	2	3	4	5	6	7	8	9	10	11	12	13
-1	-1	1	2	1	2	3	0	3	1	2	1	3	3

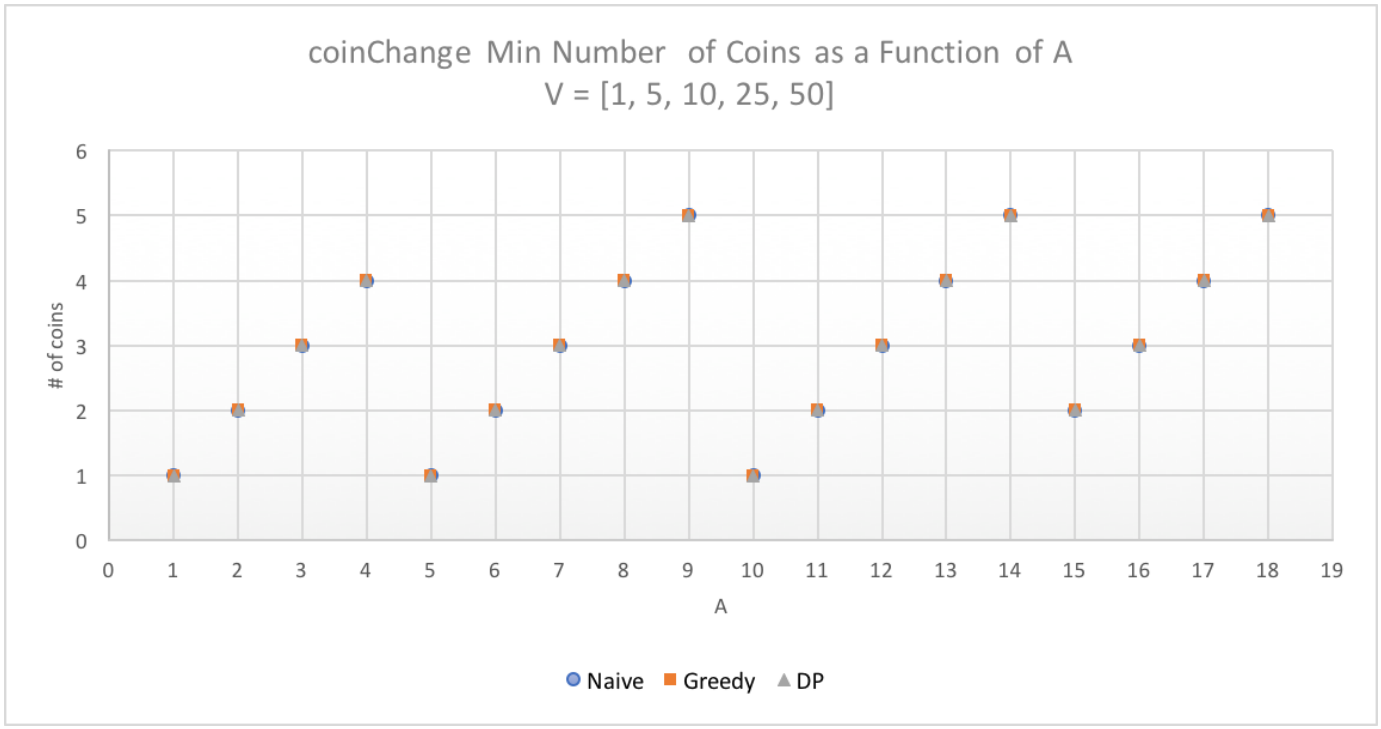
Array R[ ]

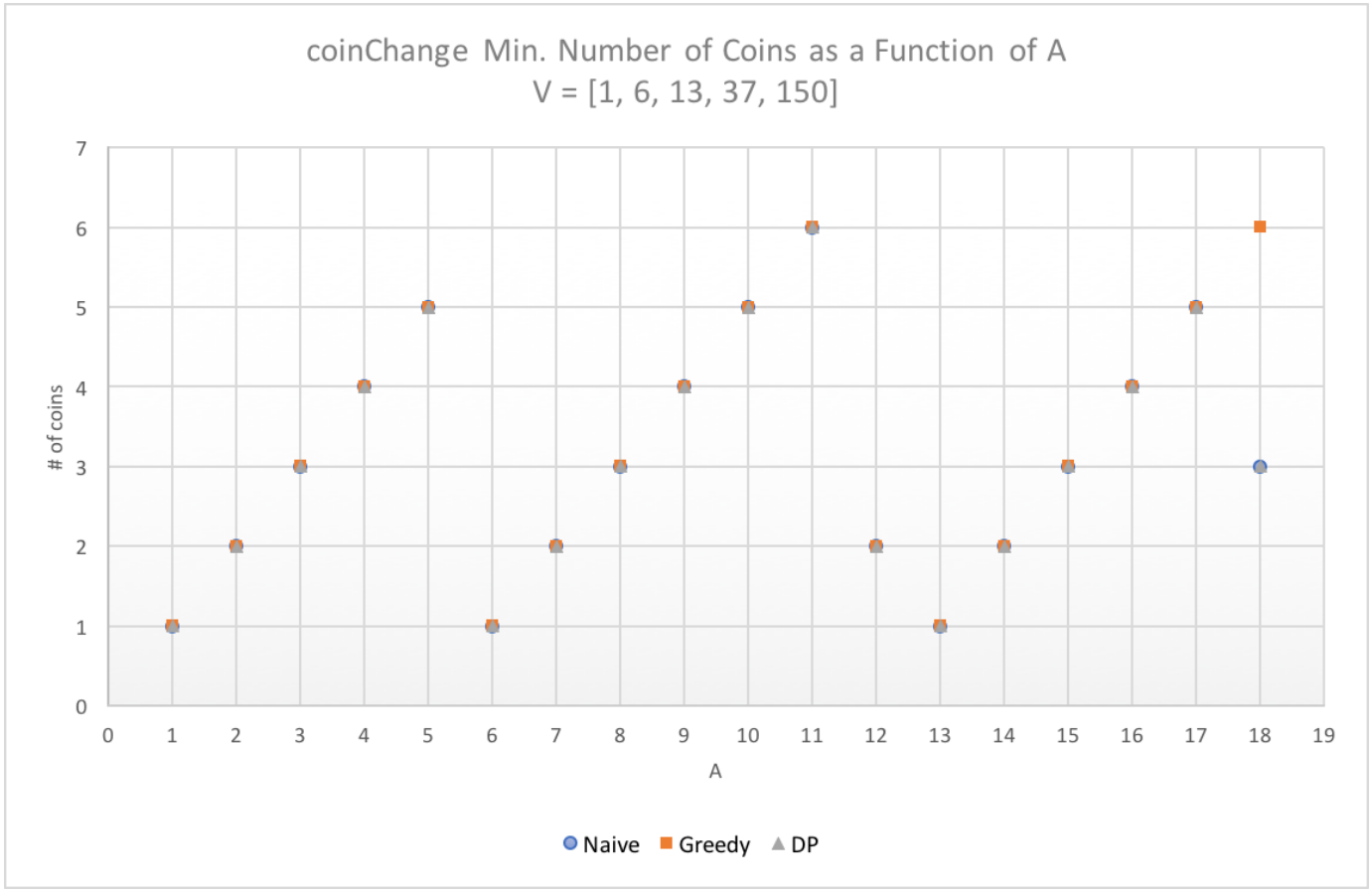
- Finally, we can see that it takes 2 coins to make a total value of 13 from a set of 4 coins with denominations of 7, 2, 3 and 6.
- To see which 2 coins make up our total of 13 we look at our array R[13] = 3, indicating that we used the 3rd coin, which was of denomination 6. 13 - 6 = 7, and R[7] = 0, indicating we used the 0th coin of denomination 7. 7 - 7 = 0, so our final answer is that we used 1 coin of denomination 6, and 1 coin of denomination 7 to get our total of 13.

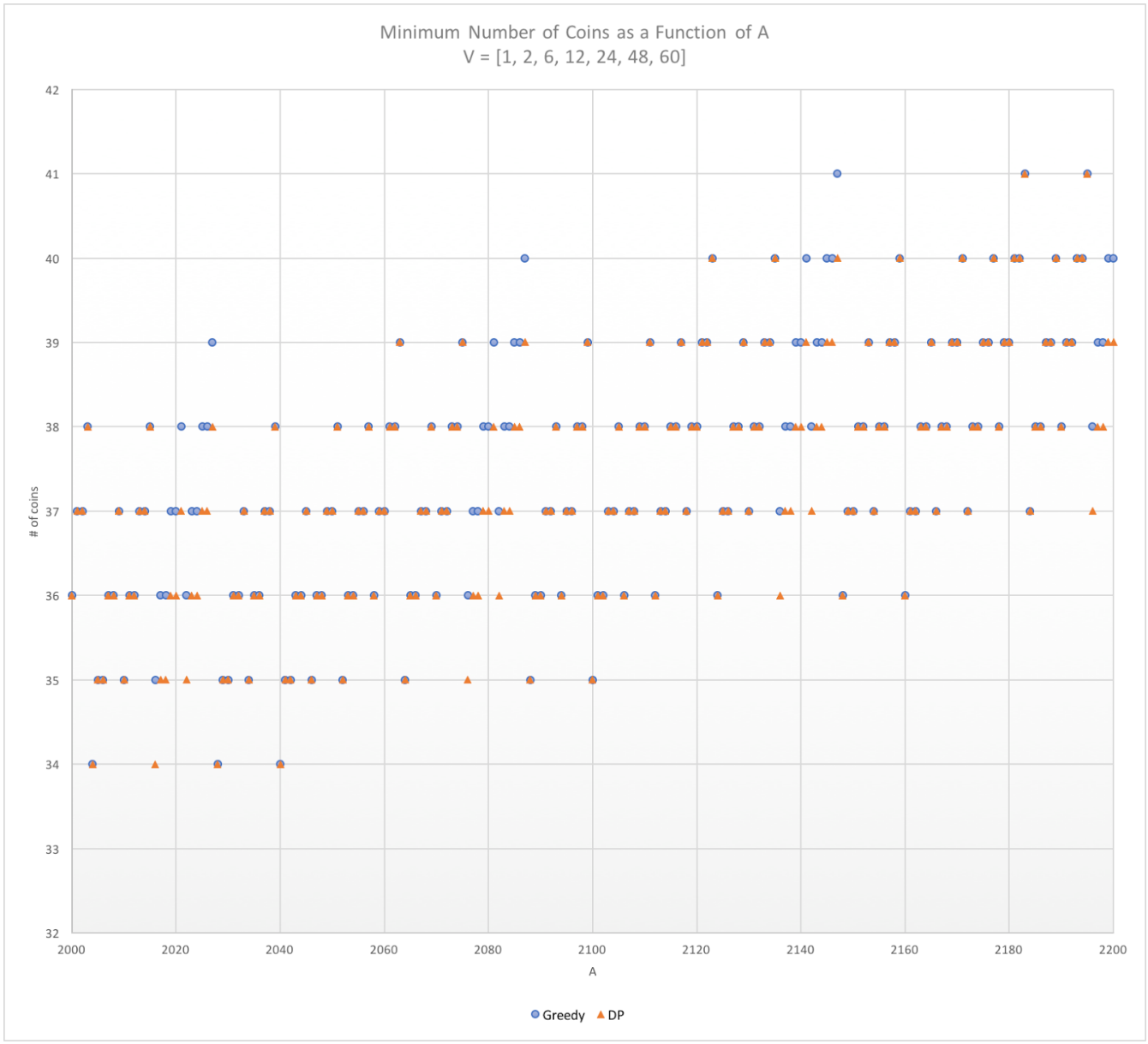
3. Suppose V_1 =[1,2,6,12,24,48,60], V_2 =[1,5,10,25,50] and V_3 =[1,6,13,37,150], for each integer value of *A* in [1, 2, 3, ..., 50] determine the number of coins that changeslow. changegreedy and changedp requires for each denomination set. For each integer value of *A* in [2000, 2001, 2002, ..., 2200] determine the number of coins that changegreedy and changedp requires for each denomination set (you can attempt to run changeslow but it will probably be too slow). Plot the number of coins as a function of *A* for each algorithm. *How do the approaches compare?*

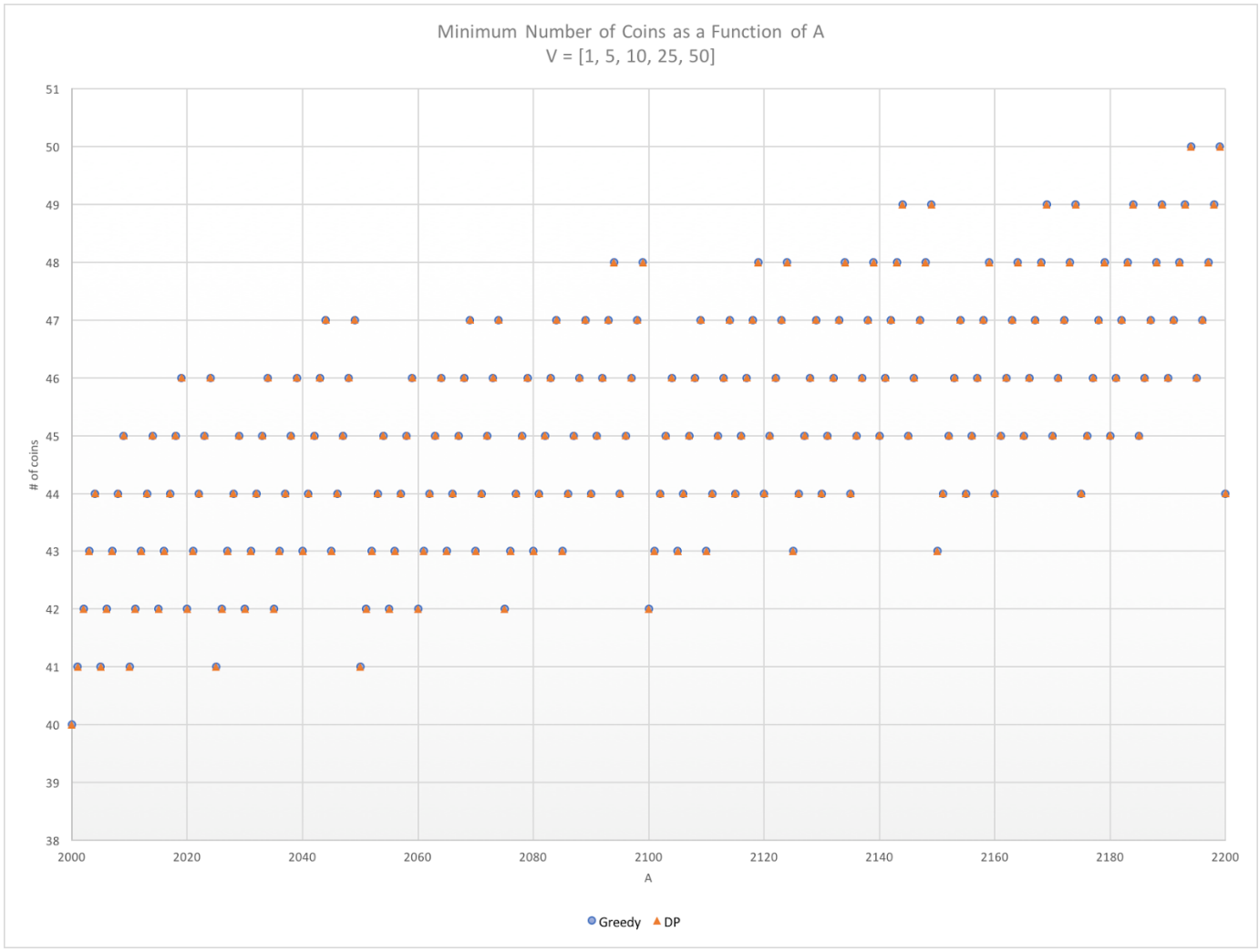
As evidenced in the graphs below, while all three algorithm implementations will usually give the correct result for the minimum number of coins required to make change for amount A from coin denominations V[], the greedy algorithm does not always yield the optimal result. For example, for V = [1, 6, 13, 37, 150] the greedy algorithm commonly fails to return the minimum number of coins required to make change for various amounts (A). It also failed for a few amounts when using the coin denominations C = [1, 2, 6, 12, 24, 48, 60]. Fortunately for the greedy algorithm, it succeeds when using more logical coin denominations (like the accepted norm of V = [1, 5, 10, 25, 50]).

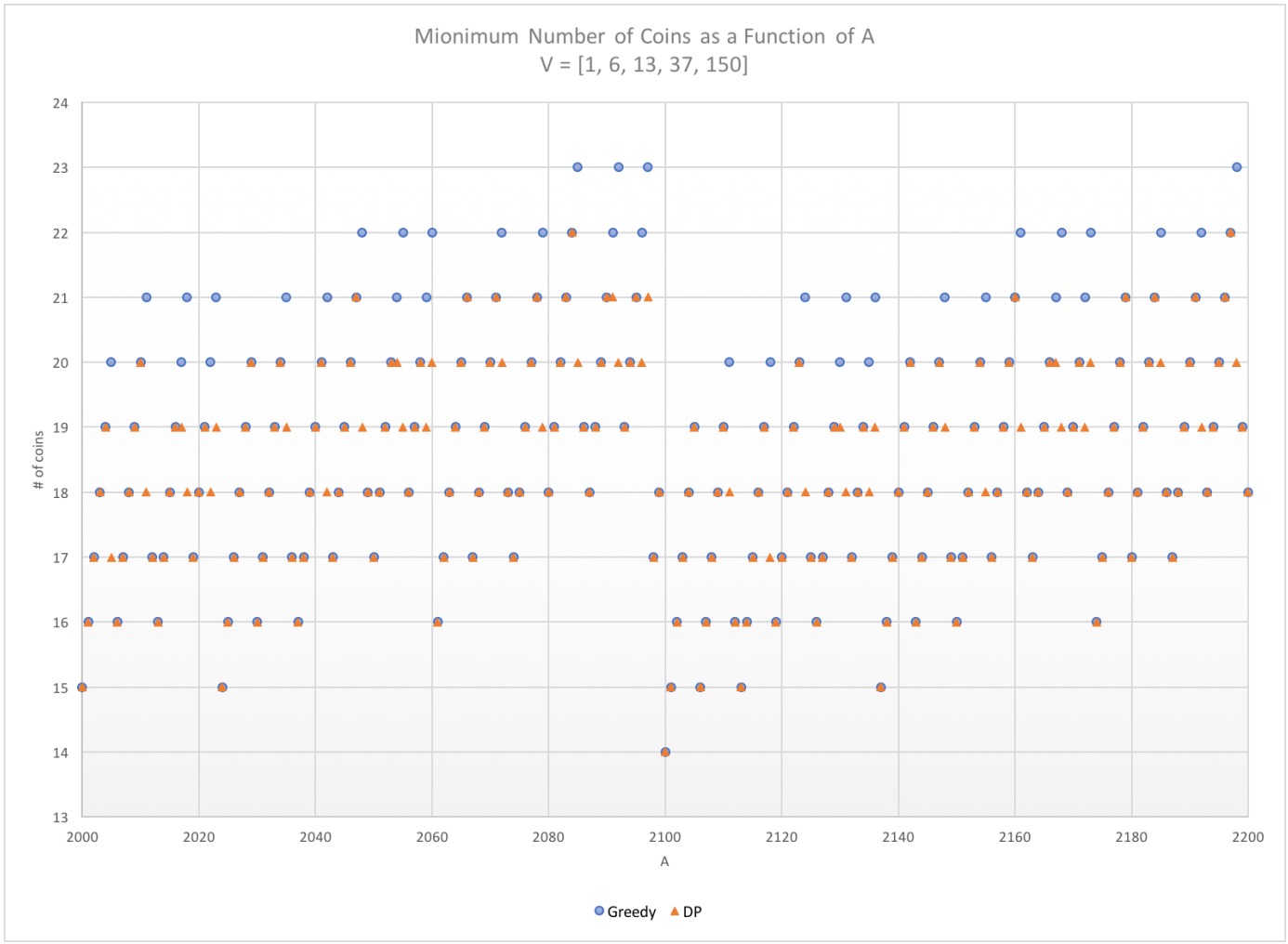



coinChange Min Number of Coins as a Function of A
V = [1, 5, 10, 25, 50]
# of coins
A
Naive
Greedy
DP


coinChange Min. Number of Coins as a Function of A
V = [1, 6, 13, 37, 150]
# of coins
A
Naive
Greedy
DP


Minimum Number of Coins as a Function of A
V = [1, 2, 6, 12, 24, 48, 60]
# of coins
A
Greedy
DP


Minimum Number of Coins as a Function of A
V = [1, 5, 10, 25, 50]
# of coins
A
Greedy
DP

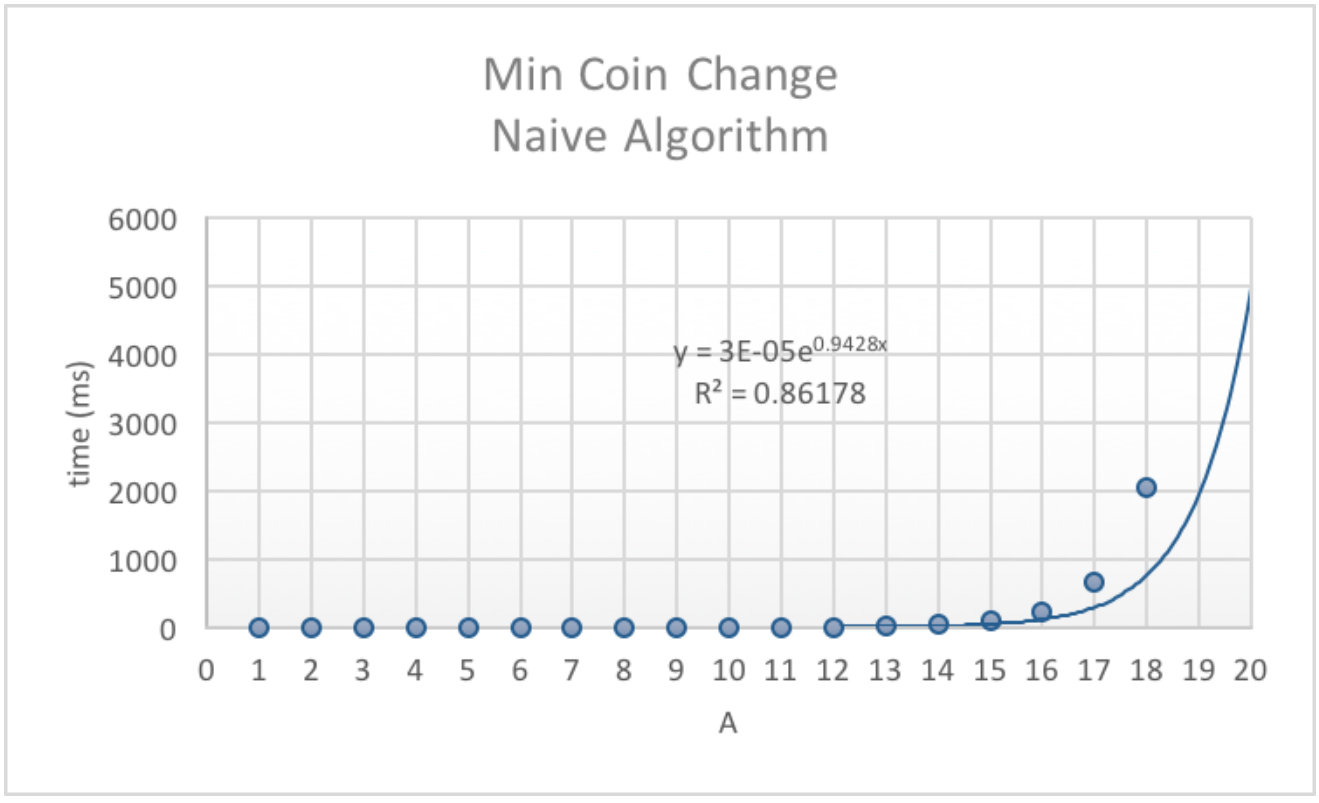

Mionimum Number of Coins as a Function of A
V = [1, 6, 13, 37, 150]
# of coins
A
Greedy
DP

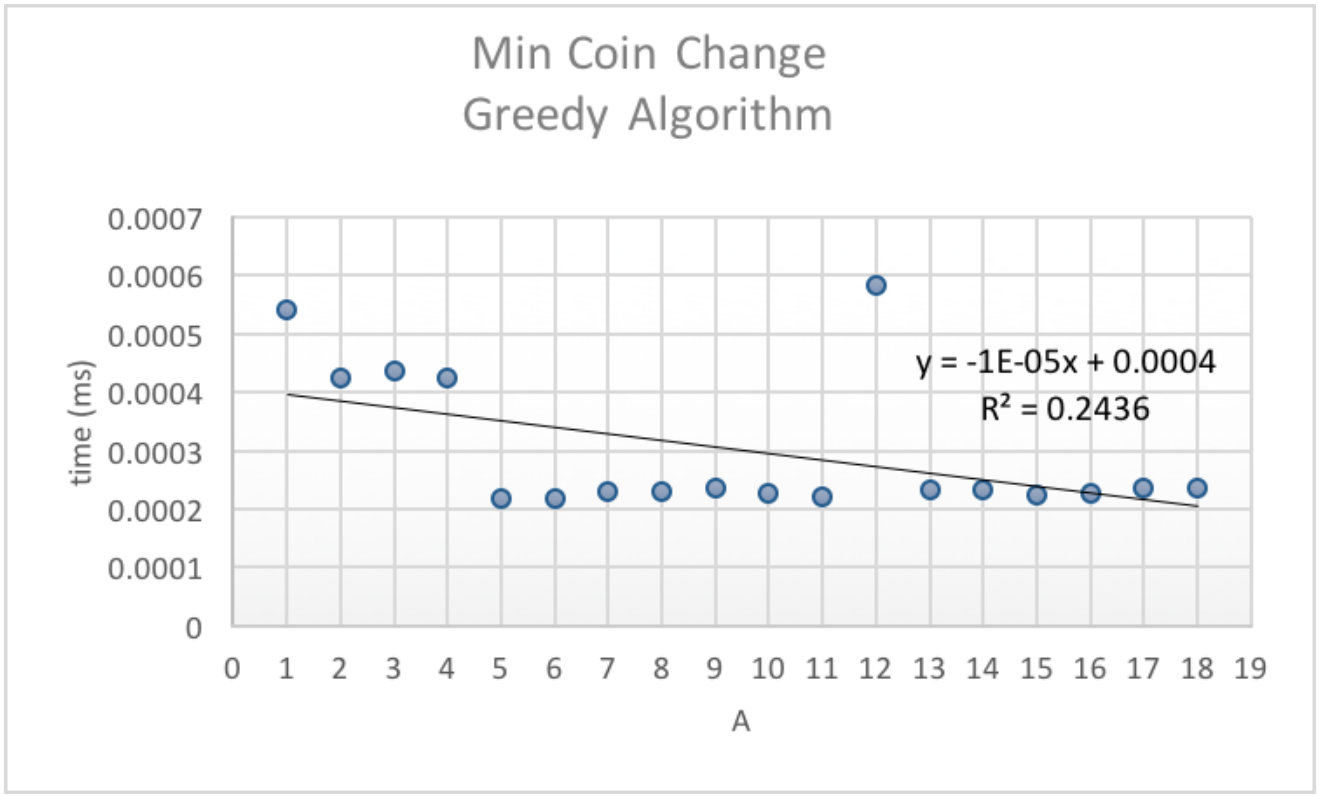
4. For each of the algorithms collect experimental running time data. Plot the running time as a function of A and fit curves to the data. *Compare the experimental running times to the theoretical running times.*

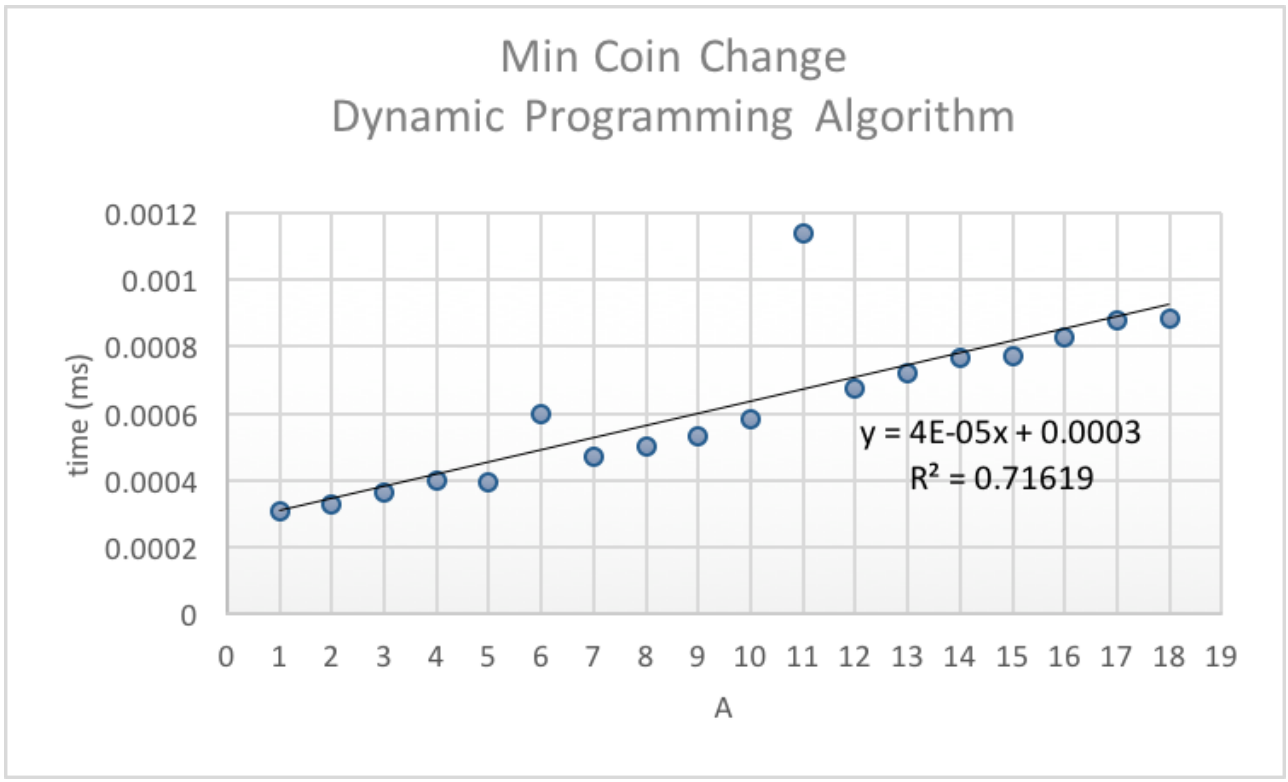
The graphs below were used with V= [1, 5, 10, 25, 50] and A = 1-19, since flip would kill our program beyond that, since our naive algorithm was so slow. We ran the dynamic and greedy algorithms with A = 2000-2200, which showed similar curves, but for brevity only the results of A= 1-19 are shown below.

The theoretical running time of the naive algorithm proved to be much more efficient, as the algorithm's running time seems to surpass the exponential curve. It proved so inefficient, that after modifying function calls and memory management to improve efficiency, it only improved enough to be killed by the flip3 server at A = 19 instead of A = 18 (as a maximal A). Overall, the algorithm exemplifies the inefficiencies that recursive algorithm implementations can introduce to a program.

Alternatively, the greedy and dynamic programming algorithms performed roughly as expected, greedy was the most efficient algorithm in terms of running times, but the result wasn't always correct. The dynamic programming solution was slightly less efficient than the greedy algorithm, but since greedy has a running time of O(n), at O(mn) this was to be expected.

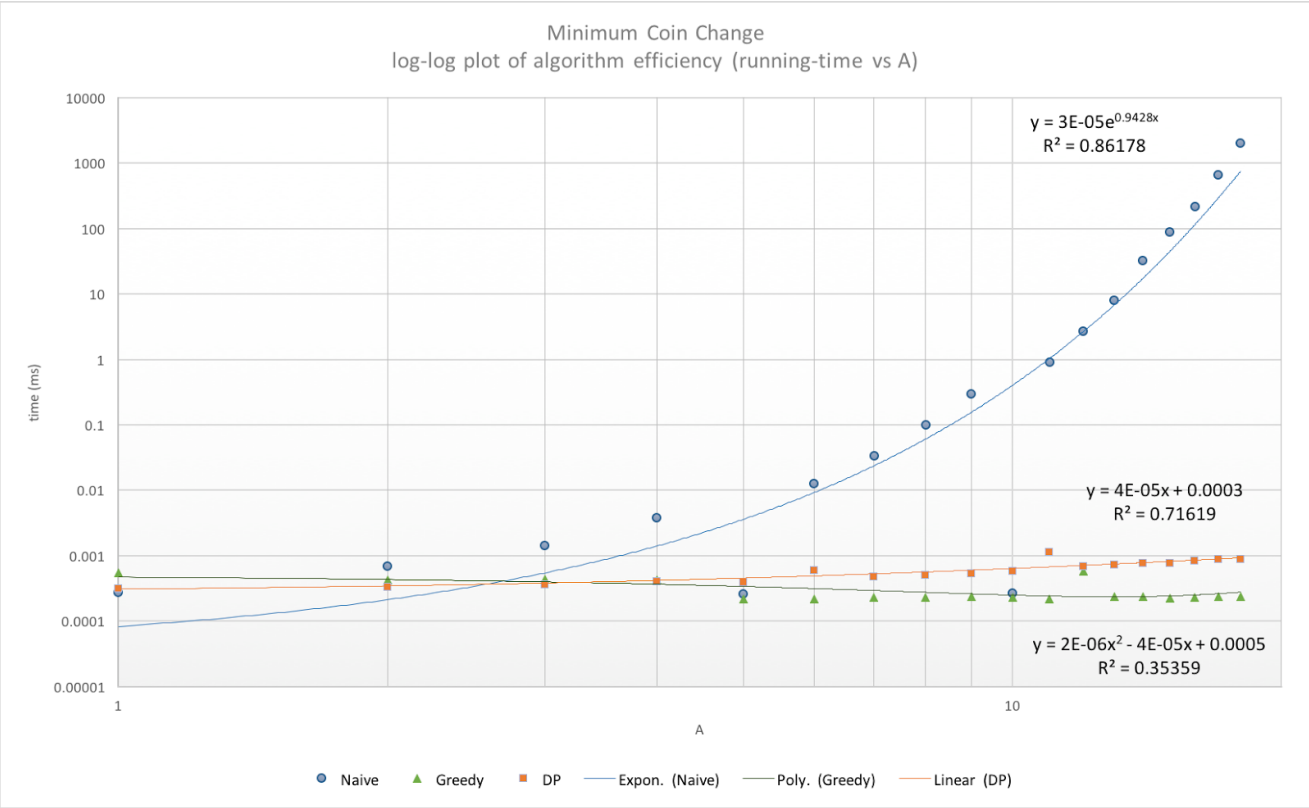



Min Coin Change
Greedy Algorithm
time (ms)
0.0007
0.0006
0.0005
0.0004
0.0003
0.0002
0.0001
0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
A
y = -1E-05x + 0.0004
R² = 0.2436


Min Coin Change
Dynamic Programming Algorithm
time (ms)
0.0012
0.001
0.0008
0.0006
0.0004
0.0002
0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
A
y = 4E-05x + 0.0003
R² = 0.71619

5. Create a single log-log plot that contains the running time data from all three algorithms. Fit trend lines to each data set. *Compare the running times of the different algorithms.*

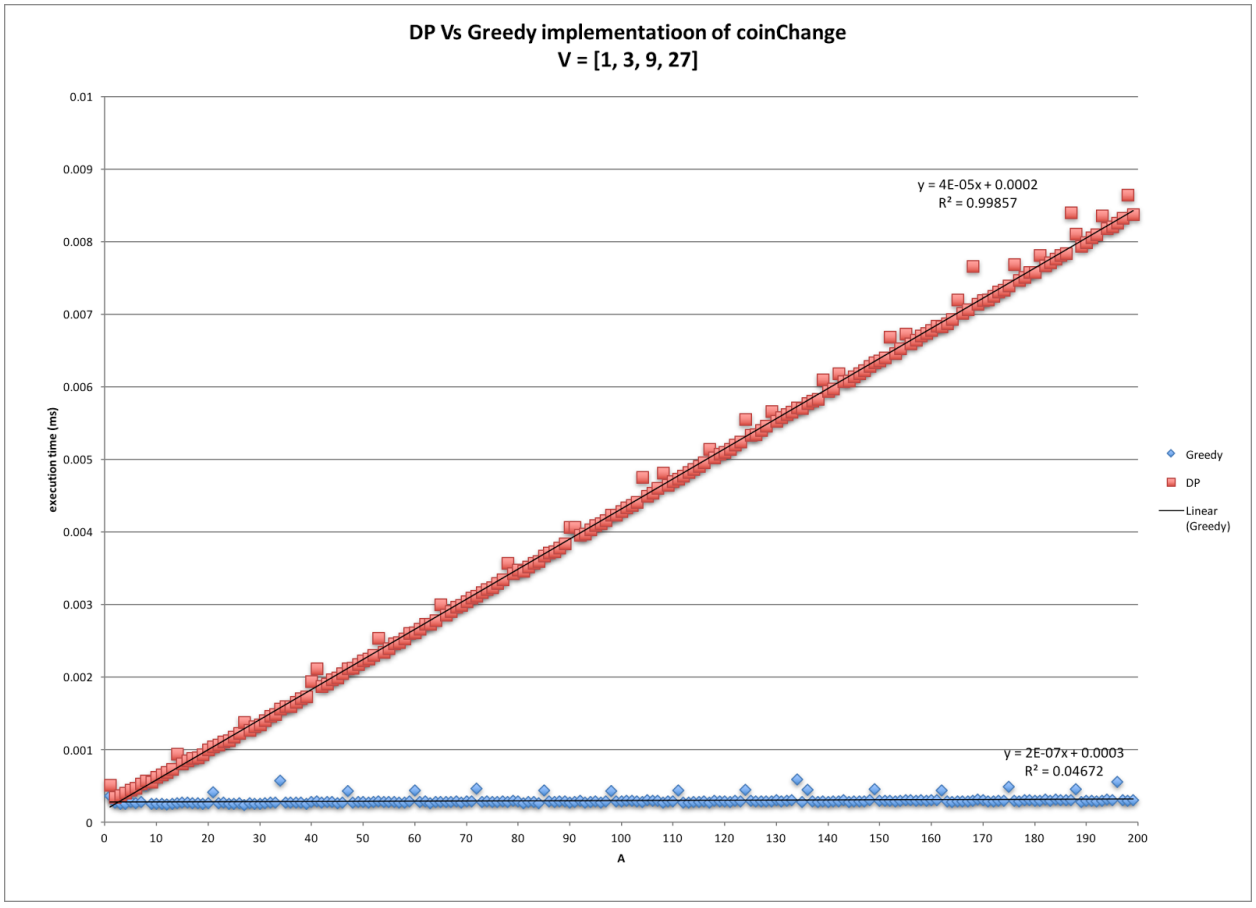
As postulated in the previous question, the log-log graph confirms our suspicions, the greedy implementation of coinChange is the most efficient, while the dynamic implementation trails the running time by fractions of a millisecond. The naive recursive implementation of coinChange has an exponential running time that renders it useless (through correct). Overall, we should accept the timing challenges of the dynamic solution and employ it if A and V will commonly vary, as it always returns the correct solution and has an acceptable running time. Meanwhile the extreme efficiency of the greedy algorithm is offset by the possibility that the result is incorrect.



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

In this case, both the dynamic programming and the greedy algorithms would give optimal results. (See #7 for an explanation as to why a greedy algorithm would produce an optimal result with this coin set.) As can be seen in the above runtime analysis, the greedy algorithm is much faster for calculating the minimum number of coins required to reach a target value. In this situation, this is important not because it is faster, however. It is important to choose to use the greedy algorithm because it is easier for the human mind to follow.

In the US, coins have an increasing value set [1, 5, 10, 25, 50]. This allows for a greedy algorithm to be used (mentally) in calculating change: you simply choose the biggest coins that fit into the total, and work your way down to the penny. This US set is convenient for a population who is used to using a base-10 number system; it is much easier, when giving change, to add and subtract values such as 5, 10, 50, than it is to work with numbers such as 3, 9, and 27. If there were a country that used the set [1, 3, 9, 27], I would be very curious why that set was chosen!



7. Give at least three examples of denominations sets V for which the greedy method is optimal. Why does the greedy method produce optimal values in these cases?

Here are three example denomination sets:
[1, 5, 10, 25, 50, 100] (this is similar to US coin currency)
[1, 2, 4, 8] (this is similar to binary bit values)
[1, 2, 7, 31, 260] (random numbers)

The cases in which the greedy algorithm does NOT produce optimal results are the cases where the denomination of one coin is less than twice the denomination of the coin of previous value. An example of this would be the set [1, 3, 7, 12]. Twice 7 is 14, so the denomination 12 is not twice the previous value. The reason why this creates sub-optimal results for the greedy algorithm can be explained by looking at the algorithm itself.

For our greedy algorithm, we chose a select function to simply take the largest denomination that could fit into the remaining target (or initial) sum. If the target value is 29, for example, the greedy algorithm chooses the denominations: 12,12,3,1,1, which is five coins. The optimal solution, however, is 12,7,7,3, which is only four coins. The greedy algorithm was forced to commit to the locally optimal second choice of 12, which made it impossible to find the optimal solution. Had the 12 coin been a 14-coin, then choosing the second 14-coin would not have overlooked the optimal solution. Indeed, it would have found an even better one: 14,14,1.

To continue with this example, what if the next denomination was greater than twice the previous? This is the case in two of the example sets above (the binary set and the random set). As an example, what if our denominations were: [1, 3, 7, 15] in the case where the target value is 29? Greedy would choose: 15,7,7, which is optimal. How about for [1, 3, 7, 16]? Again, greedy would choose 16,7,3,3, again optimal.

This is not a formal proof, by any means, but the reader may be able to see the logic behind this theory from this example. Essentially, if the denominations are increasing by at least a factor of two, for each step, then this allows for the greedy algorithm to break the problem into increasingly smaller subproblems, while choosing the optimal solution of each subproblem, which is exactly how a bottom-up dynamic programming algorithm is designed. This guarantees an optimal solution.