CS 325 Project2: Coin Change

Group 34 members: Changxu Yan, Trung-Duong Nguyen, Shengpei Yuan

1. Pseudocode

Greedy Algorithm

```
-   changeGreedy(int v[], int A, int length):
-   coin_count_array[length]    //The array used to store count of each coin
-   while(A>0)
-       for i = length-1 to 0, i decrease
-           if(v[i]<=A)
-               A = A – v[i]
-               minimum_count_value++;
-               coin_count_array[i]++;
-   break and go back to search the v[] array from the largest element again
-   return (minimum_count_value, coin_count_array[])
```

Dynamic Programming Algorithm

```
-   changeDP(int v[], int A, int length):
-   minimum_number[A+1]      // set all of elements in this array equal to infinity
-   coin_number_matrix [A+1][lenght] // the matrix used to store number of each
    coin
-
-   for i = 1 to A
-       for j = 0 to length
-           if(v[j] <= i and minimum_number[i-v[j]] + 1 < minimum_number[i])
-               minimum_number[i] = minimum_number[i-v[j]] + 1;
-               for k =0 to length
-                   coin_number_matrix [i][k] = 0    // clean this row in this matrix
-                   // add previous results to current row
-                   coin_count_matrix[i][k] = coin_count_matrix[i-v[j]][k];
-               // increase the number of current coin
-               coin_count_matrix[i][j] += 1
-   return minimum_count[A],    coin_count_matrix[A][1 to length]        // last
    row in matrix
```

n = change we need, m = number of kinds of coins

- Greedy Algorithm:
  In this algorithm, each denomination we calculate the number of coins we can take until we would make more than the amount of change asked for, so the

running time is linear and T(n) = O(n)

- DP Algorithm:
  In this algorithm, there is a nested loop. n is the amount of money, and m is the number of coins, so T(n) = O(n*m).

2. In the DP algorithm, I used one array and one table to solve this problem. The array is used to store all of minimum numbers of coins from 0 to A as memorization. On the other hand, that table is also used to store all of number of each coin from 0 to A as memorization.

First of all, we focus on this array. At the beginning, we will find the minimum when the A is 0 and 1. As value of change is increasing, we can use (current A – current coin) to find previous minimum number of coins in this array. For example, now the current A is 10 and current coin is 7. So we can check the minimum number of coins when A is 3 (from 10-7) from this array memorization. On the other hand, probably other kinds of coins can make the number minimum, so we also need to compare them. Take an specific example, v=[1, 3, 5] and A=3

0	1	2	

When the A is 3:

Step 1: check the coin array, the minimum number is 3 when current coin is 1. Because 3-1 = 2, we use third result m_n(2)=2 from the minimum array, and it pluses 1.

0	1	2	2 + 1 = 3

Step 2: Next, when we use coin whose value is 3, we will find 3-3 = 0, we use the first result m_n(0)=0 from the minimum array, and it pluses 1, so it equal to 1

0	1	2	0 + 1 = 1

Step 3: Compare this result with previous result, we find step 2 is the optimal solution.

So the formal should be this:

$$m\_n(i) = \min\{m\_n(i - v_j) + 1\}$$

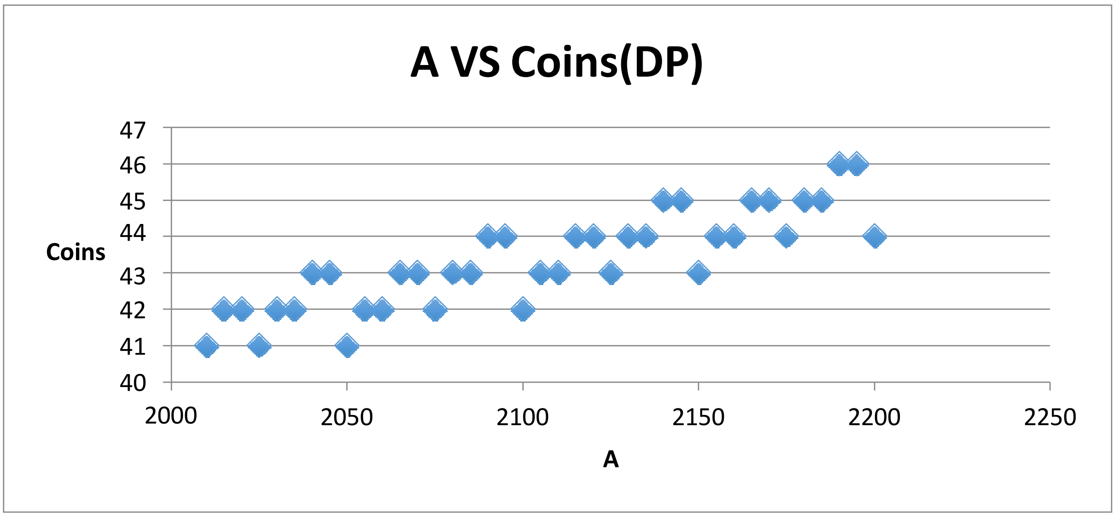
As for the matrix, it is also implemented by dynamic programming. in matrix, we can also use current row (all elements should be zero in current row) to plus (current A – current coin) row firstly, and then current coin increases one. For example, v=[1, 3, 5] and A=2

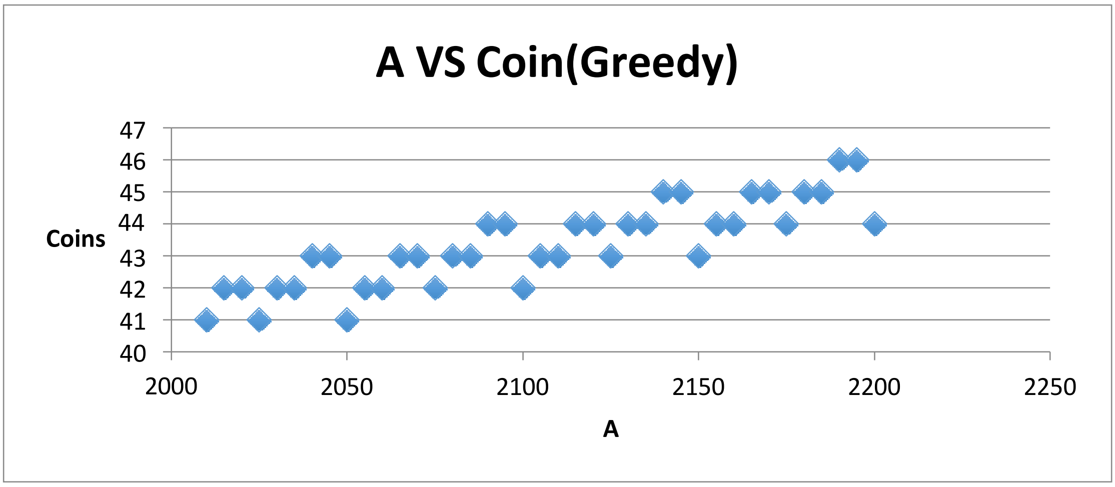
A	1	3	5
0	0	0	0
1	1	0	0
2	2	0	0

Step 1: we add current row 2 to (current change – current coin = 1) row, and then we get [1, 0, 0]

Step 2: to increase number of current coin, and then we get [1+1, 0, 0]. So these are results of each coin.

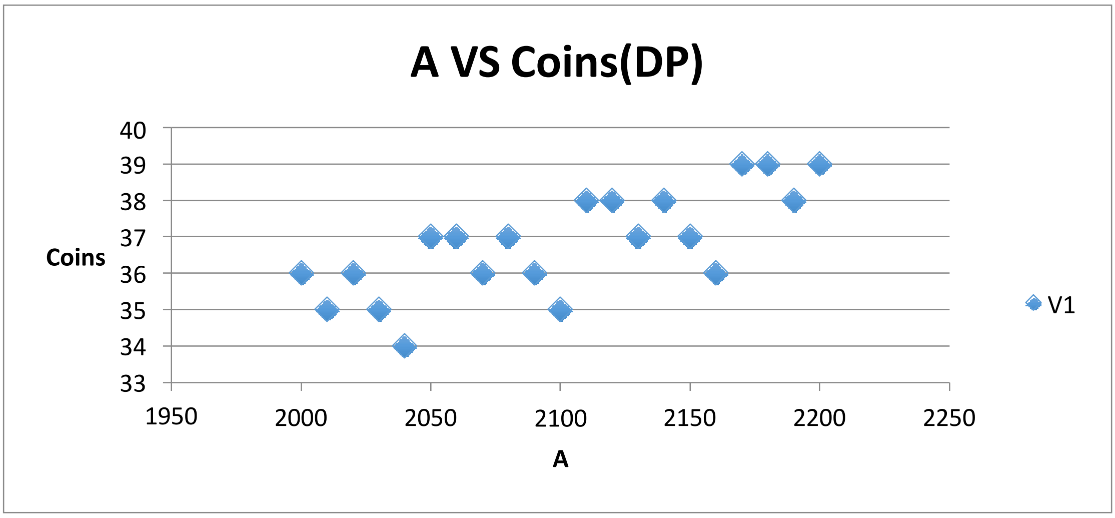
3. V = [1, 5, 10, 25, 50] and A is in [2010, 2015, …, 2200]

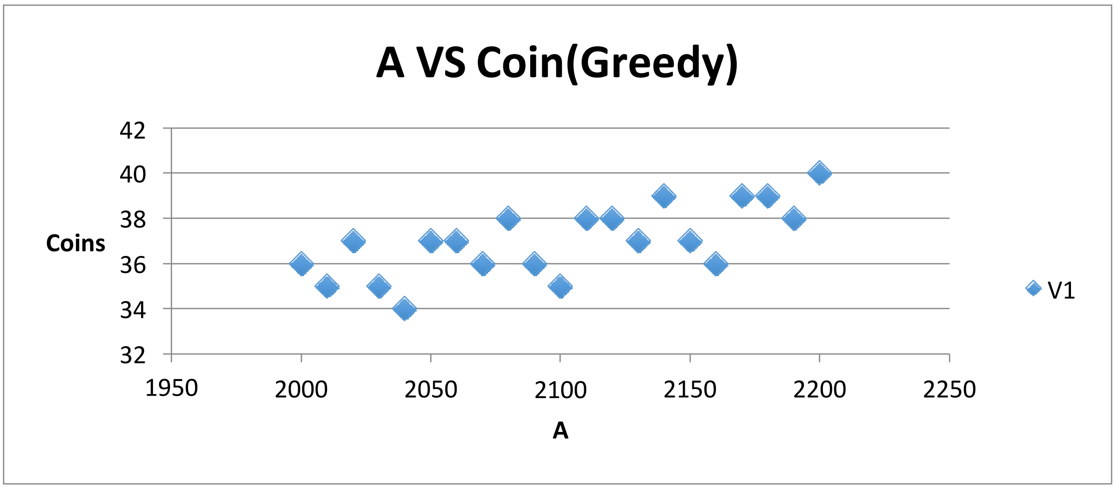




According to these two plots, we can find these two kinds of algorithm have identical results for A from 2010 to 20200.

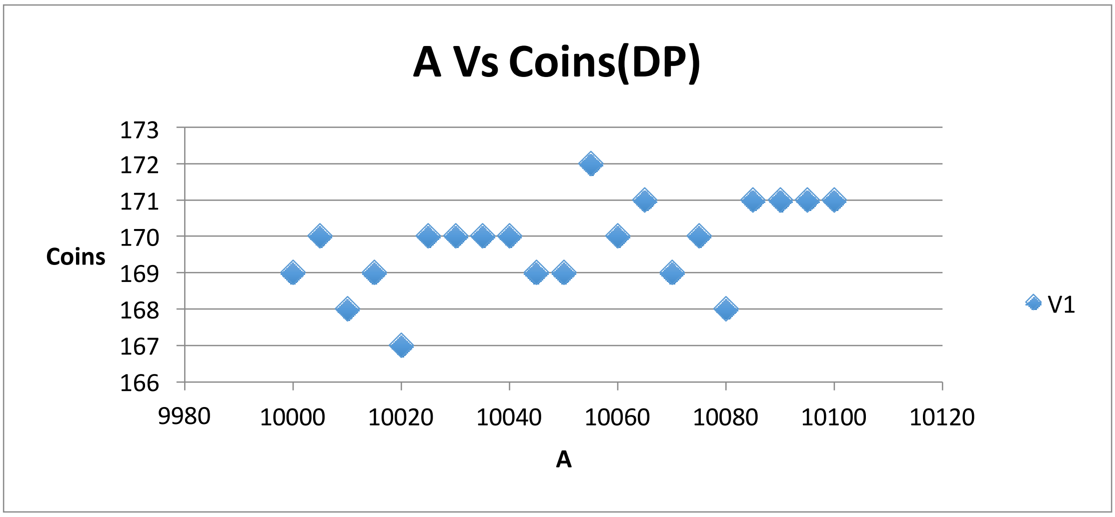
4. V_1=[1, 2, 6, 12, 24, 48, 60] and A is in [2000, 2001, …, 2200]

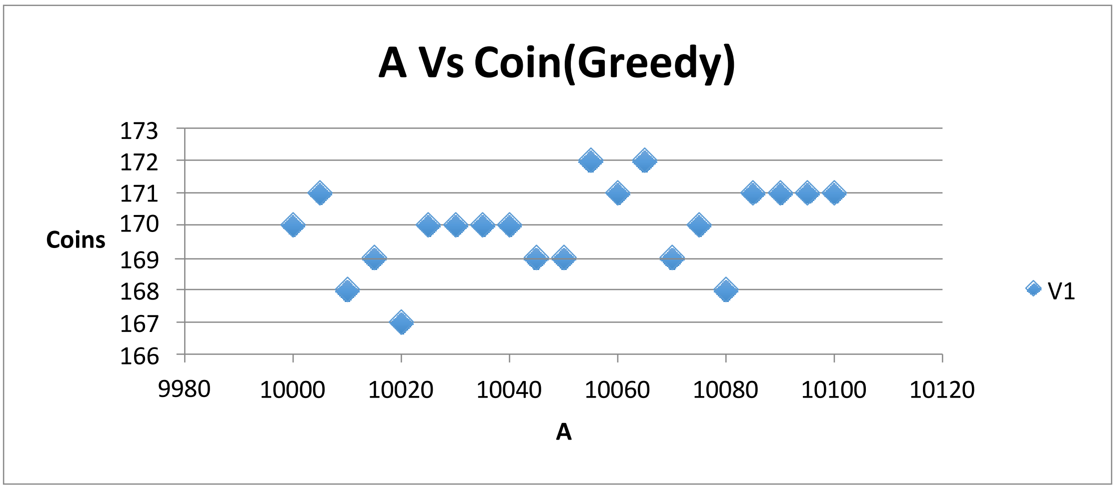




According to these two plots, we can find that DP algorithm has lesser minimum number of coins than Greedy algorithm on some A. In the other word, x is minimum number of DP and y is minimum number of Greedy, so $x \leq y$, when x, $y \epsilon \{2000, 2001, \ldots, 2200\}$.

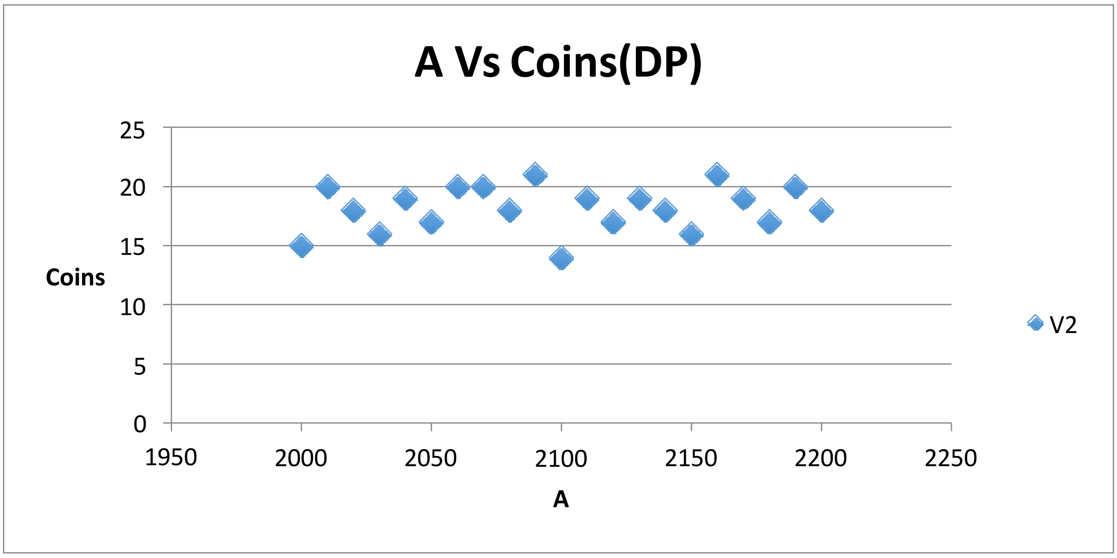
V_1=[1, 2, 6, 12, 24, 48, 60] and A is in [10000, 10001, …, 10100]

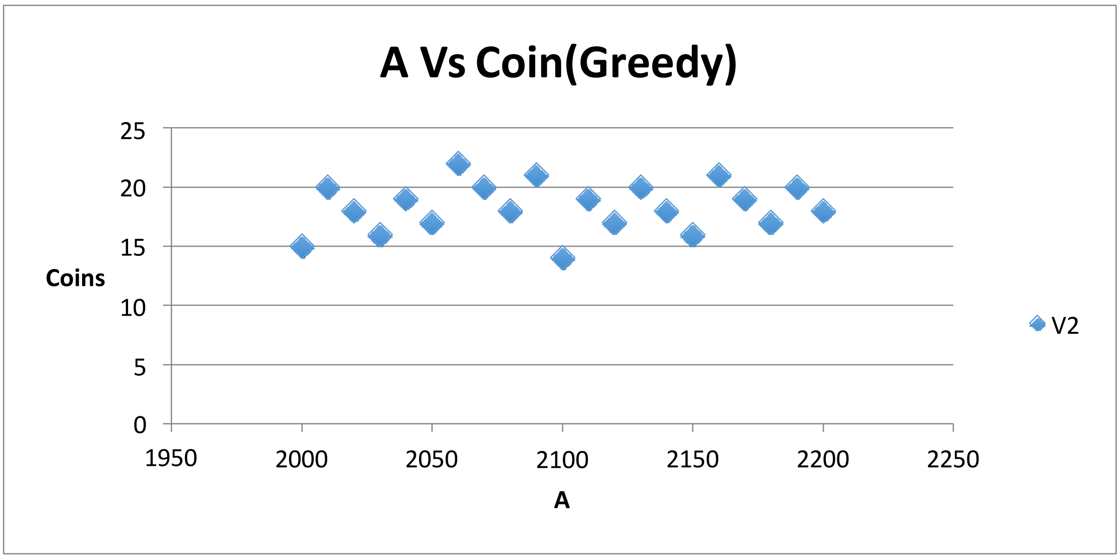




According to these two plots, we can find that DP algorithm has lesser minimum number of coins than Greedy algorithm on some A. In the other word, x is minimum number of DP and y is minimum number of Greedy, so $x \leq y$, when x, $y\epsilon\{10000, 10001, \dots, 10100\}$.

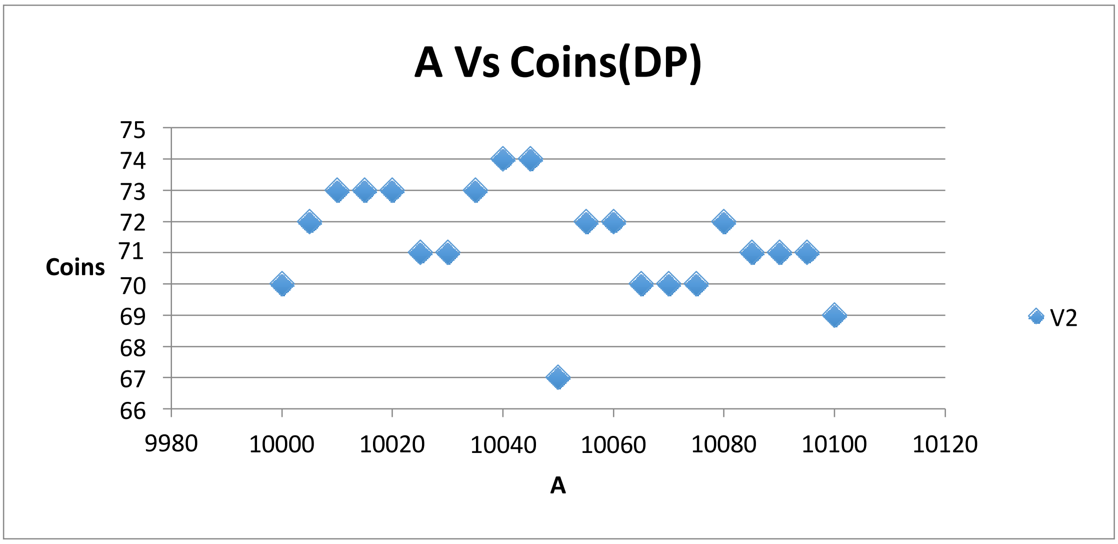
V_2=[1, 6, 13, 37, 150] and A is in [2000, 2001, …, 2200]

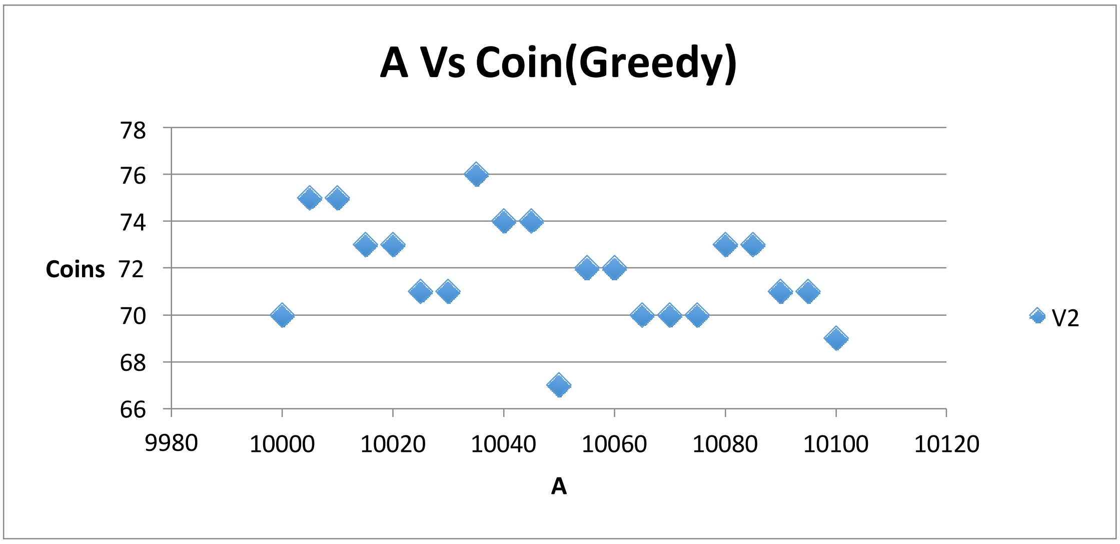




According to these two plots, we can find that DP algorithm has lesser minimum number of coins than Greedy algorithm on some A. In the other word, x is minimum number of DP and y is minimum number of Greedy, so $x \leq y$, when x, $y\epsilon\{2000, 2001, \dots, 2200\}$.

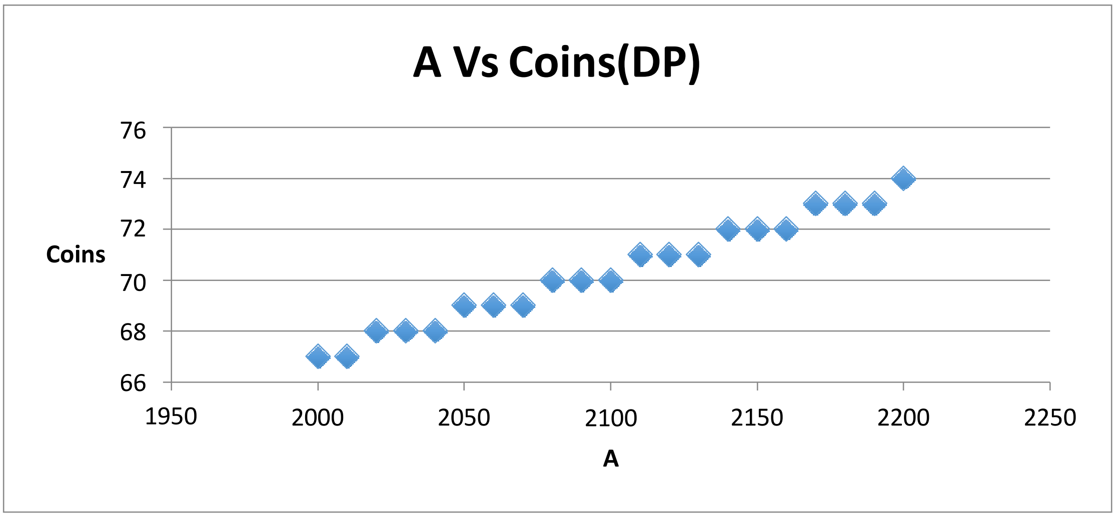
V_2=[1, 6, 13, 37, 150] and A is in [10000, 10001, …, 10100]

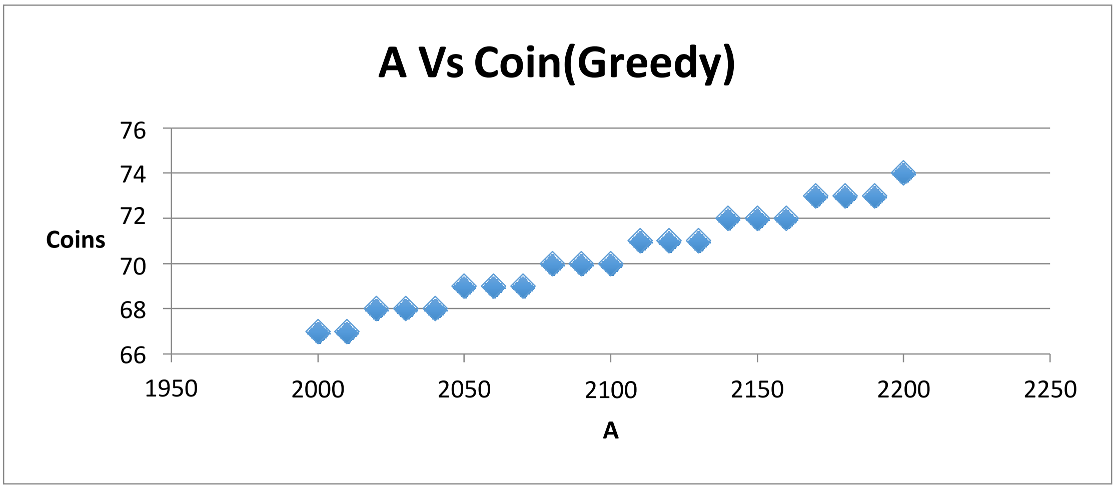




According to these two plots, we can find that DP algorithm has lesser minimum number of coins than Greedy algorithm on some A. In the other word, x is minimum number of DP and y is minimum number of Greedy, so $x \leq y$, when x, $y \epsilon \{10000, 10001, \dots, 10100\}$.

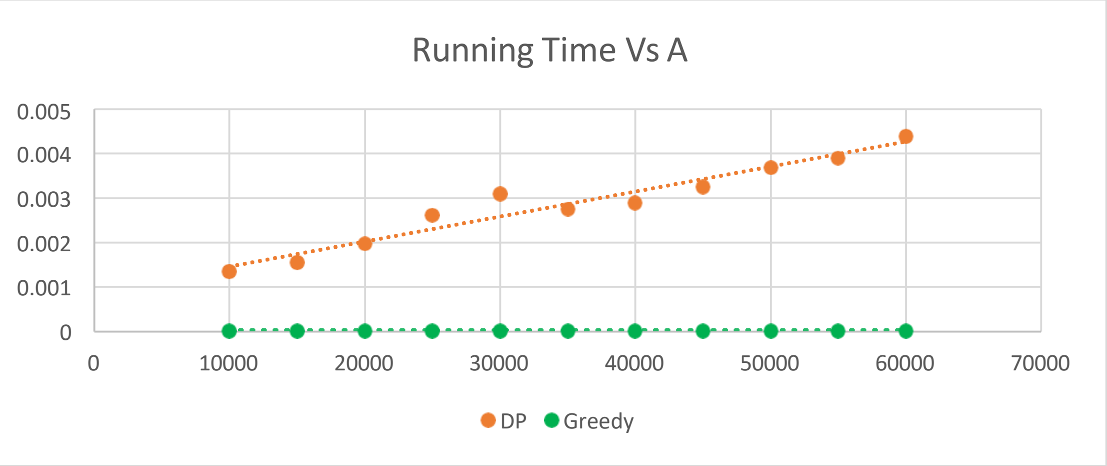
5. V = [1, 2, 4, 6, 8, …, 30] and A is in [2000, 2001, 2002, …, 2200]

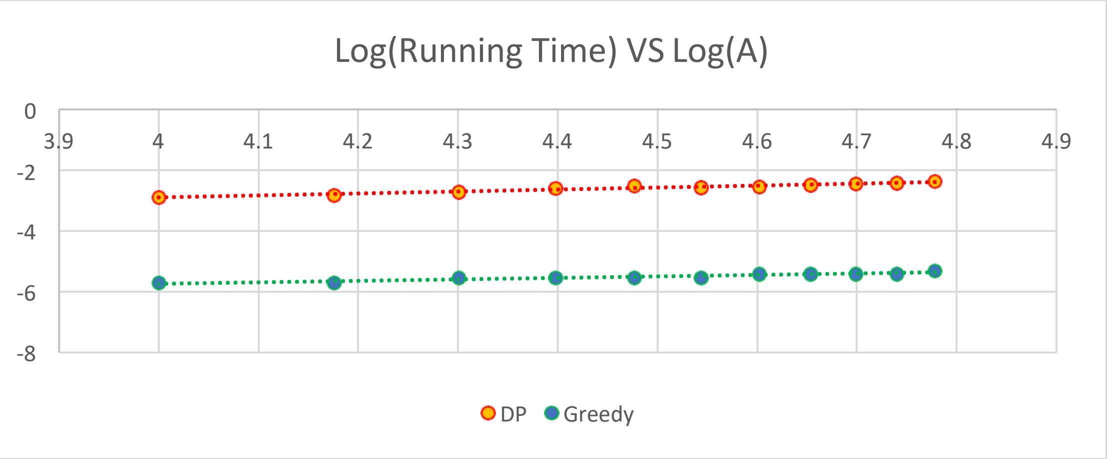




According to these two plots, we can find these two kinds of algorithm have identical results for A from 2000 to 20200.

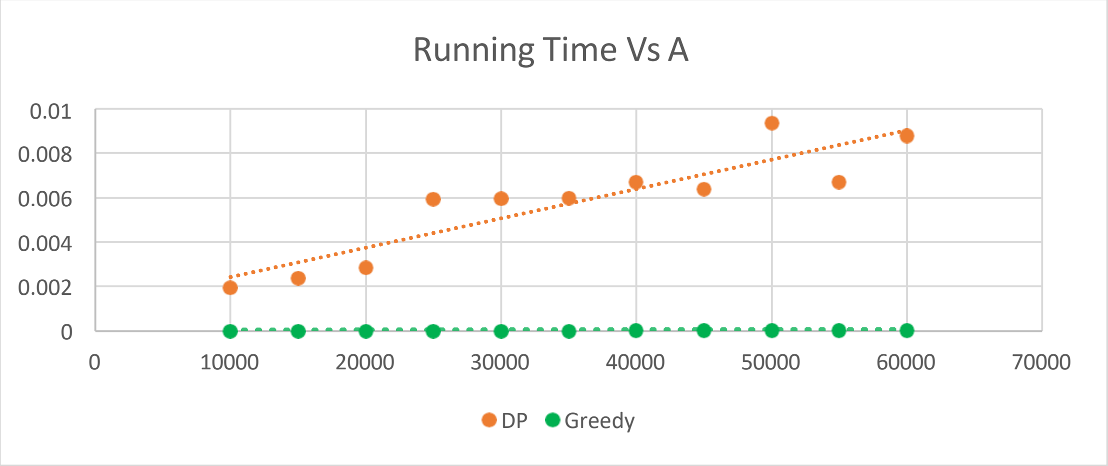
6. In this question, we used value of A in [10000, 15000, 20000, …, 60000] to test all coin arrayies from question 3-5.
V = [1, 5, 10, 25, 50]

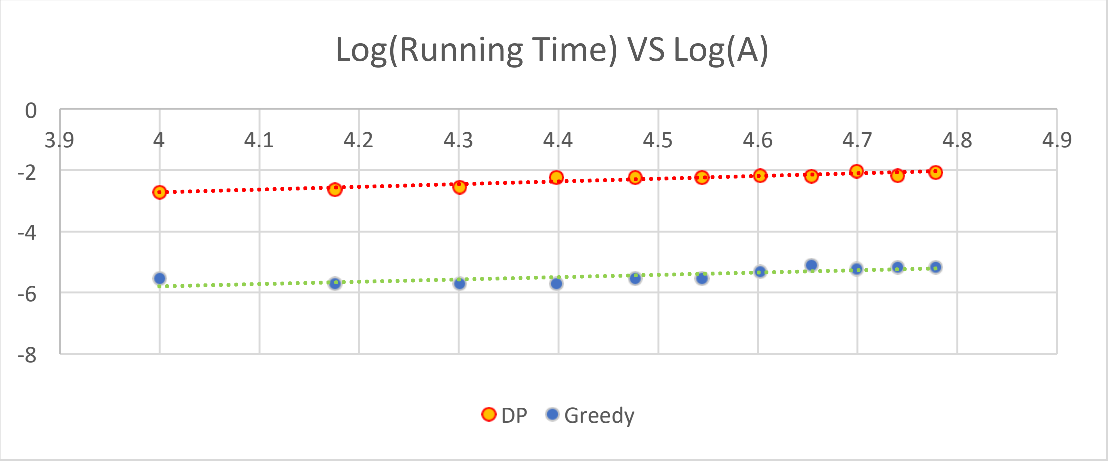




According to these results, we can find the running time of greedy algorithm is faster than dynamic programming algorithm.

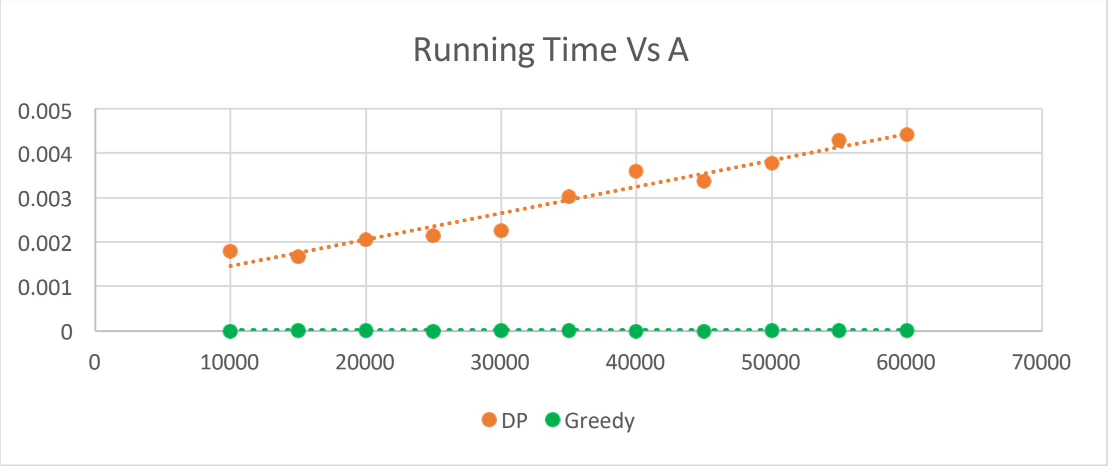
V=[1, 2, 6, 12, 24, 48, 60]

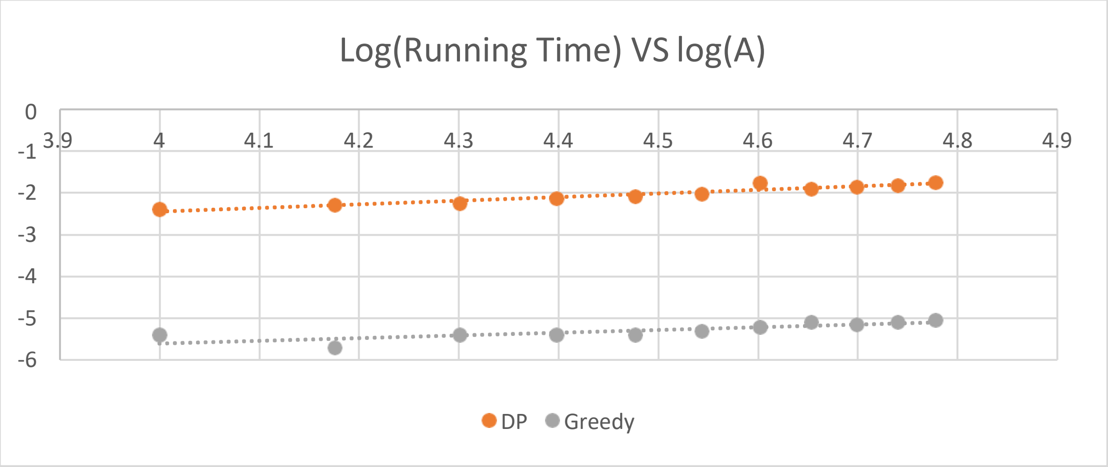




According to these results, we can find the running time of greedy algorithm is faster than dynamic programming algorithm.

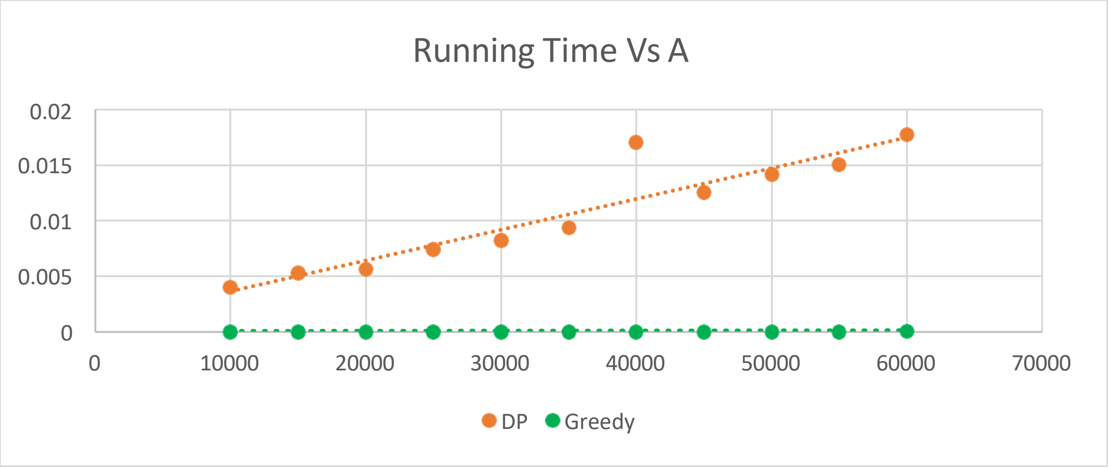
V=[1, 6, 13, 37, 150]

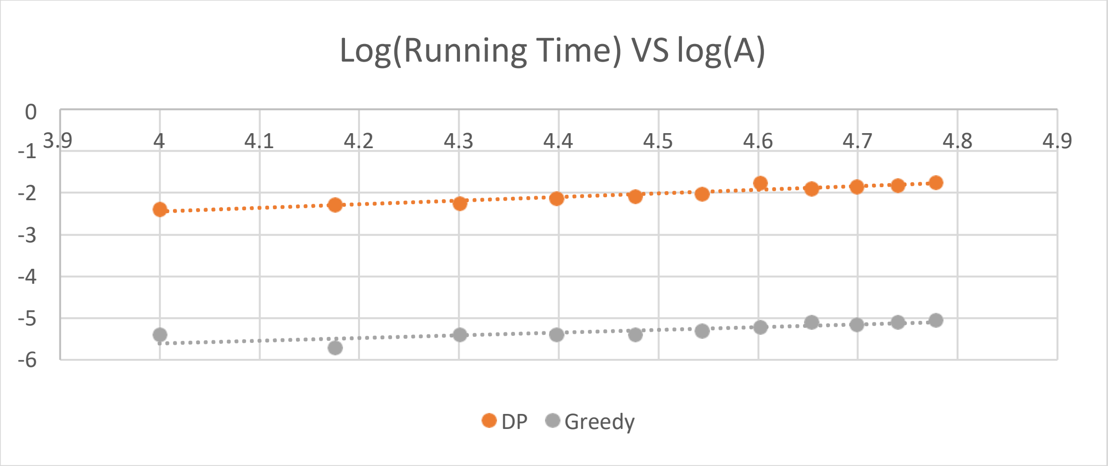




According to these results, we can find the running time of greedy algorithm is faster than dynamic programming algorithm.

V = [1, 2, 3, 4, 6, …, 30]





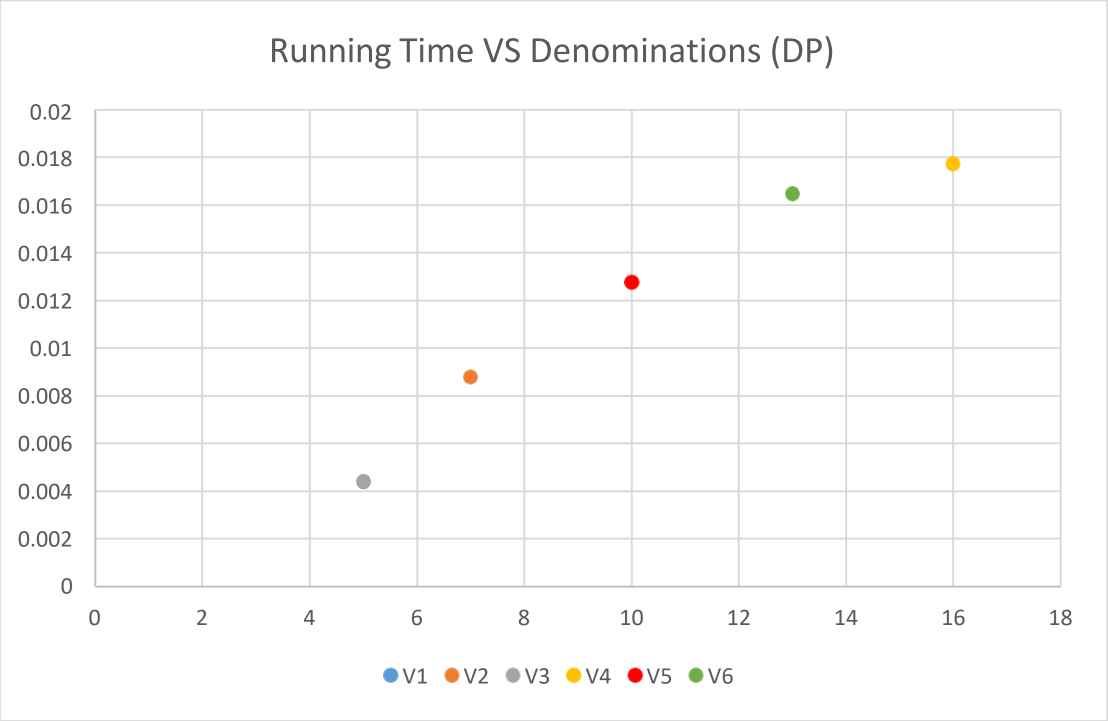
According to these results, we can find the running time of greedy algorithm is faster than dynamic programming algorithm.

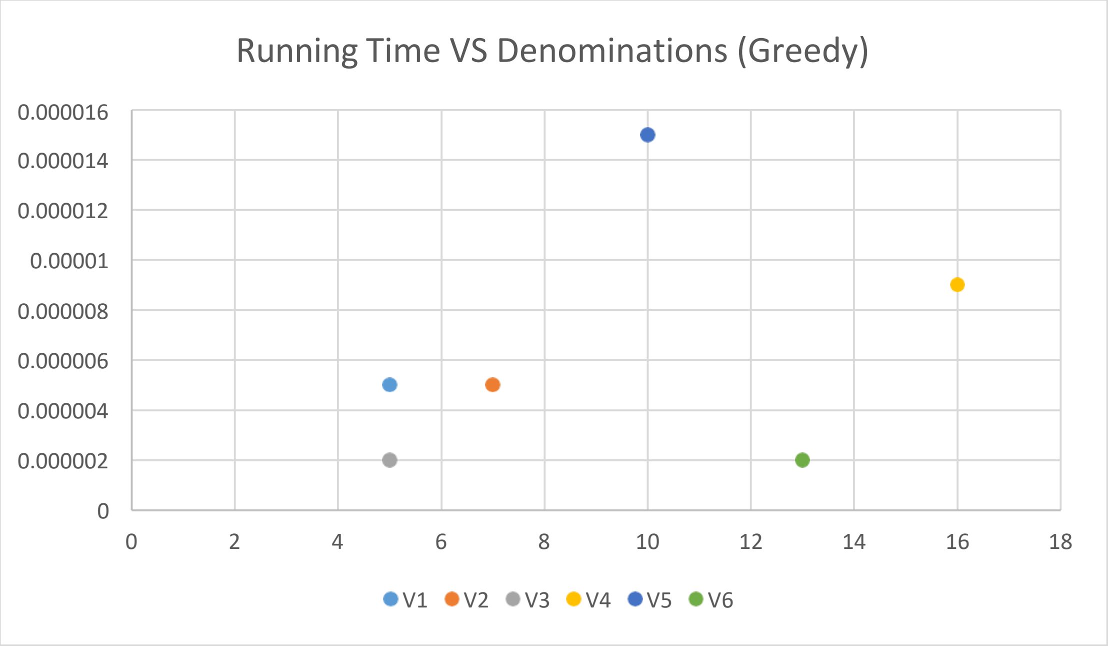
7. In this question, we used six kinds of coin arrays, four of them are from question

4-6, and other two of them are created by ourselves. In addition, the running time of each group coin array is time they take when A is 60000.

V1 = [1, 5, 10, 25, 50]	size = 5
V2 = [1, 2, 6, 12, 24, 48, 60]	size = 7
V3 = [1, 6, 13, 37, 150]	size = 5
V4 = [1, 2, 4, 6, 8, 10, …, 30]	size = 16
V5 = [1, 2, 4, 6, 8, 10, 12, 14, 16, 18]	size = 10
V6= [1, 2, 5, 8, 10, 20, 30, 50, 70, 90, 100, 120, 150]	size = 13

Size sorting V1 = V3 < V2 < V5 < V6 < V4





The size of number of denominations did influence the running times of dynamic programming. In the theory of dynamic programming, we need to build table as a memorization and then the running time of this algorithm is (n*m), so n and m are both factors of influence. If the number of denominations going bigger, in other words, the n will be increased, which means the dynamic programming will do more calculations to give more possible solutions.

The plots we generated are just proved our intuition. The plot for the dynamic programming shows when the number of denominations growing bigger then the running time has increased as the number of denominations were grew. On the other hand, we can see running time of V1 and V3 overlap, because these two groups of coin have the same denominations, so running time of them are very close.

However, the running time of greedy algorithm will not be influenced by number of denominations. Instead of that, greedy algorithm will always check the biggest element first to see if that is a possible optimal solution in a set of denominations. Because of that, the greedy algorithm will not be going to check all denominations, so the number of denominations is does not matter for greedy.

For the plot of greedy, the V1 and V2, V3 and v6, these two pairs have similar running time even the number of denomination has been changed. In V1 and V2, the largest numbers are 50 and 60, these two numbers are very close. On the other hand, in V3 and V6, the largest number is 150 for both of them. For those four sets of denominations that they all have a same characteristic in common which is they all contains a denomination that is big enough to make the greedy can running fast. For the V4 and B5 they do have a common characteristic also which is their biggest denomination is not big enough and have less difference as other denominations in V4 and V5. More than that, there has another evidence that the V5 has 10 denominations in a set and the V4 has 16 denominations, but the V5 is consumed more time than V4; so the number of denominations is does not matter for the greedy algorithm.

8. In this case, we could assume the interval of A is from 1 to 100, and the reason is amount of coins we generally use is on this interval in our real life. Results we got are they have the same solutions when we did test for A in [1, 2, …, 100]. So I think greedy algorithm should be used in this case. Because both of them have the same optimal solutions, so we should choose algorithm whose running time is short under this condition. The greedy is more efficient than DP algorithm, so we choose it. On the other hand, in the real world, if a man wants to calculate the minimum number of coins depend on brain, it should use greedy algorithm, because the DP algorithm is not good method for human's brain.

9. As we know, the greedy algorithm cannot always give us the optimal solution,

because only optimal under certain conditions. The problem has the optimal substructure property and the algorithm satisfies the greedy-choice property. But a test solution from the previously problem that both greedy and dynamic programming can give us the optimal solution. So we assume in some specific set of denominations that will give the greedy algorithm a better chance to have optimal or most optimal solution as good as dynamic programming. From a formula view is if the denomination of coins can satisfy $C^2,\ C^3,\ \dots,\ C^4\ with\ c > 1$, which is same as the set of denominations in Question 8 that each denomination should be a factor of its next denomination then the greedy will have the best chance to give the optimal solution.