Project 2 – Coin Change Report

Project Group 7: Larissa Hahn, Nathalie Blume, and Nathan Thunem

July 23, 2015

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 C be an array of coin denominations that are available for making change on some TOTAL_VALUE amount.

Coin Array"C"

index	0	1	2
Coin value	2	4	6

The following is a recursive definition for the DP algorithm that solves the change-coin problem. In this definition, OPT is the optimal solution where the number of coins returned on TOTAL_VALUE is the smallest:

```
OPT(TOTAL_VALUE) =
{
        0                                                   if TOTAL_VALUE=0;
        $\min_{i:C[i] <= TOTAL\_VALUE}$ {1 + OPT(TOTAL_VALUE-C[i]}          if TOTAL_VALUE> 0;
}
```

With a DP approach, the optimal solution is sought from the ground up, computing smaller total_values first, storing these in a retrievable form, and moving up toward computing change on larger amounts. Work increasingly shifts from computing change amounts "from scratch" (here: by consulting array C and iteratively trying out different combinations) to looking up optimal solutions to subproblems in a table and then combining them together. This shift is reflected in the way we fill the table below.

TOTAL_ VALUE	0	1	2	3	4	5	6	7	8	9	10	11
OPT = Min. Num of Coins	0	0	1	1	1	1	1	1	2	2	2	2
Optimal Set of Coins	empty-set	empty-set	{2}	{2}	{4}	{4}	{6}	{6}	{4,4}	{4,4}	{6,4}	{6,4}

How the table works, in words:

- *Table structure.* The table displays side-by-side a series of results for increasing TOTAL_VALUEs. This table is different from the max_subarray table. The latter had two dimensions representing the two input arrays and the solution emerged from patterns at the intersection of those two dimensions. The coin_change table has a dimension that represents target (TOTAL_VALUE) and the rest of the table consists of solutions achieved mostly by computations that are not transparent in the table itself. The common idea is that both tables show the carry over flow from sub-problems to their composites.
- *Step 0: Base Case, TOTAL_VALUE = 0*. A "Usable Set" is defined as the set of coin denominations that are less than or equal to TOTAL_VALUE. Here, usable set C(i) = {}; there is no set of coins that will provide change for that value. A 0 is entered on the 2nd row and the empty set on the the 3rd.
- *Step 1: TOTAL_VALUE = 1*. Usable set C(i) = {}. Look for next lesser column with the same C(i) and test whether difference in TOTAL_VALUE is less than the value of any single coin in C(i). This condition is true, so copy over the solution. Here, column 0 is copied into column 1.
- *Step 2: TOTAL_VALUE = 2.* Usable set C(i) = {2}. This is the first time we encounter this C(i). For each value in C(i), we find the solution to "TOTAL_VALUE minus that coin's value", add 1 (for the inclusion of that coin), and choose the solution with the smallest number of coins. Here, since we are at the early stages of filling the table, finding a solution involves looking at how many times each coin in C(i) fits into TOTAL_VALUE ("doing it the hard way"). The best solution is 1 coin of value 2.
- *Step 3: TOTAL_VALUE = 3*. Similar to step 1.
- *Step 4: TOTAL_VALUE = 4.* Usable set C(1) = {2, 4}. This is the first time we encounter this C(i). For each value in C(i), look for the best solution that involves at least one instance of that value. The solution that involves the 2-coin can be resolved with a table look-up to TOTAL_VALUE = 2, iteratively applied until the difference between the set of coins and TOTAL_VALUE is 0 or some positive value smaller than the smallest coin in C. Here, the minimum such solution involves 2 coins and the set {2, 2}. The solution that involves the 4-coin involves a computation "the hard way" and resolves to a solution with 1 coin and the set {4}. This latter solution has the min number of coins and is retained.
- *Step 5: TOTAL_VALUE = 5.* Usable set C(1) = {2, 4}. Similar to step 1.
- *Step 6: TOTAL_VALUE = 6.* Usable set C(1) = {2, 4, 6}. Similar to step 4.
- *Step 7: TOTAL_VALUE = 7.* Usable set C(1) = {2, 4, 6}. Similar to step 1.
- *Step 8: TOTAL_VALUE = 8.* Usable set C(1) = {2, 4, 6}. However the difference between 8 and TOTAL_VALUE the first time this C(i) was encountered (TOTAL_VALUE = 6) is 2, which is at least as much as the value of one of the coins in C(i). So the process involves new iterative searches as in Step 4.

Acknowledgement: https://www.youtube.com/watch?v=18NVyOI_690

2. Algorithm Pseudocode

Algorithm 1: Brute Force or Divide and Conquer

```
int changeSlow(array, value, size)
        numCoins = 0;
        coinCount[size];
        for i in range size →  coinCount[i] = 0
        if value > 0
                //Use recursion to find change combination
                numCoins = helper_changeSlow(array, value, size, coinCount)

int helper_changeSlow(array, value, size, coinCount)
        coin = size-1;
        for i in range size → coinCount[i] = 0 // reset coinCount
        numCoins = 0;
        //Base cases: either 1 coin solution or no more coins to consider
         if value == array[coin]
                 coinCount[coin]++
                 numCoins++
         else if coin <= 0 && value < array[coin]
              numCoins = +inf
        //Special cases: skip if coin too high; Repeat if last coin to try & it fits several times
        else if value < array[coin]
                numCoins = helper_changeSlow(arr, value, (size - 1), coinCount)
        else if coin <= 0
                numCoins = helper_changeSlow(arr, (value - array[coin]), size, coinCount);
                numCoins++;
        //Recursive cases: try both include or don't include current coin in solution
        else
                int c1 = helper_changeSlow(array, value, (size - 1), coinCount);
                int c2 = helper_changeSlow(array, (value - array[coin]), size, coinCount);
                  //Return from recursion: select best solution so far and return that
                if c1 < c2
                        numCoins = c1;
                else
                        coinCount[coin]++;
                        numCoins = c2 + 1;
  return numCoins;
```

Algorithm 2: Greedy

```
changeGreedy(array, value, size)
        total= value
        coin = size-1
```

```
cointCount[size]
//initialize empty array
for i in range(size)
        coinCount[i] = 0
numCoins = 0
while total > 0 and coin >= 0
        if array[coin] <= total
                coinCount[coin]++
                total -= array[coin]
                numCoins++
        else
                coin--
for i in range(size)
        print coinCount[i]
        array[i] = coinCount[i]

print numCoins

return numCoins
```

Algorithm 3: Dynamic Programming

```
int changeDP(int arr[], int value, int size)
        int min_coins[value+1]     //optimal solution for each value
        min_coins[0] = 0           //no change

        //initialize each value to infinity
        for(int i = 1; i < value; ++i)
                min_coins[i] = 9999999

        //find every solution for arr[value]
         for(int i = 1; i <= value; ++i)
                for(int j = 0; j < size; ++j)
                        if(arr[j] <= i)
                if(min_coins[i - arr[j]] + 1 < min_coins[i])
                        min_coins[i] = min_coins[i - arr[j]] + 1

         //makes it possible to print number of each coin  denomination used
         //Take out when doing experimental analysis
         int total = value
         int coin = size-1
         int coinCount[size]

        for(int i = 0; i < size; ++i)
                coinCount[i] = 0
```

```
int numCoins = 0
while(total > 0 && coin >= 0)
          if(arr[coin] <= total)
                    coinCount[coin]++
                    total -= arr[coin]
                   ++numCoins
          else
                   --coin

return min_coins[value]
```

3. Prove that the dynamic programming approach is correct by induction. That is, prove that T[v] = mini:V[i}≤v{T[v − di]+1}, T[0] = 0 is the minimum number of coins possible to make change for value *v*.

Claim. Let T[v] be the minimum number of coins of denominations V[1], V[2], ... , V[i] needed to make change for value v cents.

Proof.
Base Case: Trivially, when making change for 0 cents, the value of the optimal solution is clearly 0 coins.

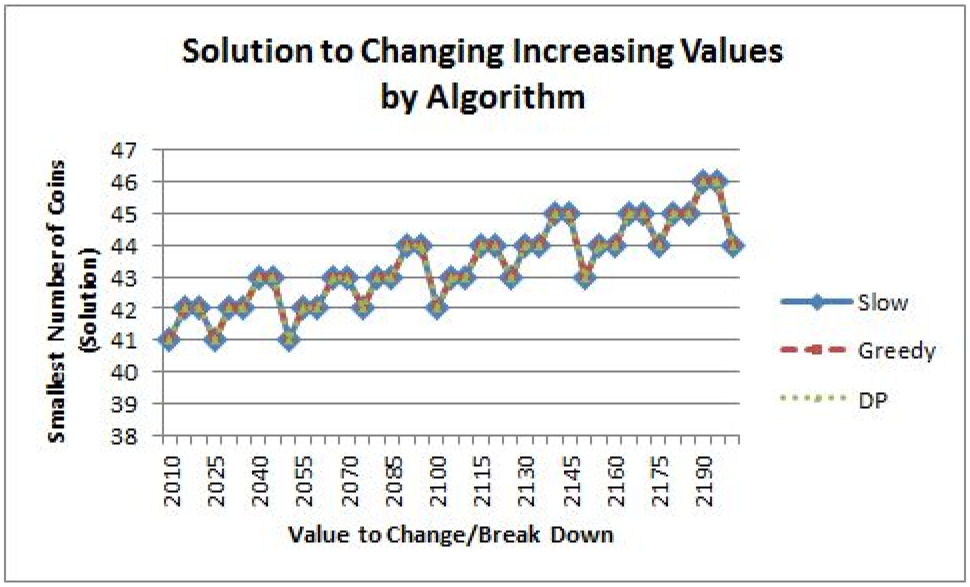
Inductive Case: In the optimal solution to making change for v cents there must exist some first coin V[i] , where V[i] ≤ v. Also, the remaining coins in the optimal solution must themselves be the optimal solution to making change for v − V[i] cents, since coin changing exhibits optimal substructure: where if v>0 and a way to obtain v with T[v] coins uses at least one coin of denomination V[i] (at least one such i exists), then removing this coin we obtain a way to obtain v−V[i] and hence conclude T(v−V[i]) ≤ T(v)−1 for at least one such that 1≤i≤v. On the other hand, if 1≤i≤v and v≥V[i], we obtain a way to obtain v with T(v−V[i])+1 coins by adding a V[i] coin to an optimal way to get v−V[i]. We conclude T(v)≤T(v−V[i])+1 for all 1≤i≤v with v≥V[i].

Thus, if V[i] is the first coin in the optimal solution to making change for v cents, then T[v] = 1 +T[v−V[i]]. For example, one V[i] coin plus T[v−V[i]] coins to optimally make change for v−V[i] cents. Note that we don't know which coin V[i] is the first coin in the optimal solution to making change for v cents; but, we may check all i such possibilities (with the constraint that V[i] ≤ v), and the value of the optimal solution must correspond to the minimum value of 1 + T[v − V[i]], by definition. So finally we reach the following recurrence: T[v] = { 0 if v = 0 mini:V[i]≤v{T[v − V[i]]+1} if v > 0; where it is true that T[v] is the minimum number of coins of denominations V[1], V[2], ... , V[i] needed to make change for value v cents.

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?

The table below show the size of the sets of coins that were discovered by the 3 algorithms in answer to the problem stated above. In theory we expect all three algorithms to find sets whose size increases as the value to which they total up increases. We also expect both DP and changeSlow to find the same (optimal) solutions. Greedy may in some problem statements occasionally finds suboptimal solutions, but here Greedy is expected to find optimal solutions because every element in V is a multiple of the next lower element. This makes V a canonical set (see Question 9). Finally we expect some cyclical fluctuations from the fact that some values (e.g. 2100) are perfectly divisible by large coin denominations while their close neighbors are not (e.g. 2104), and that this progression repeats across the [2010...2200] range.

The figure below shows experimental results for the three results. For each algorithm, the number of coins in the solution is plotted against values in the range [2010-2200]. As Expected, the all algorithms find the same, increasingly large solutions with local cyclical fluctuations.

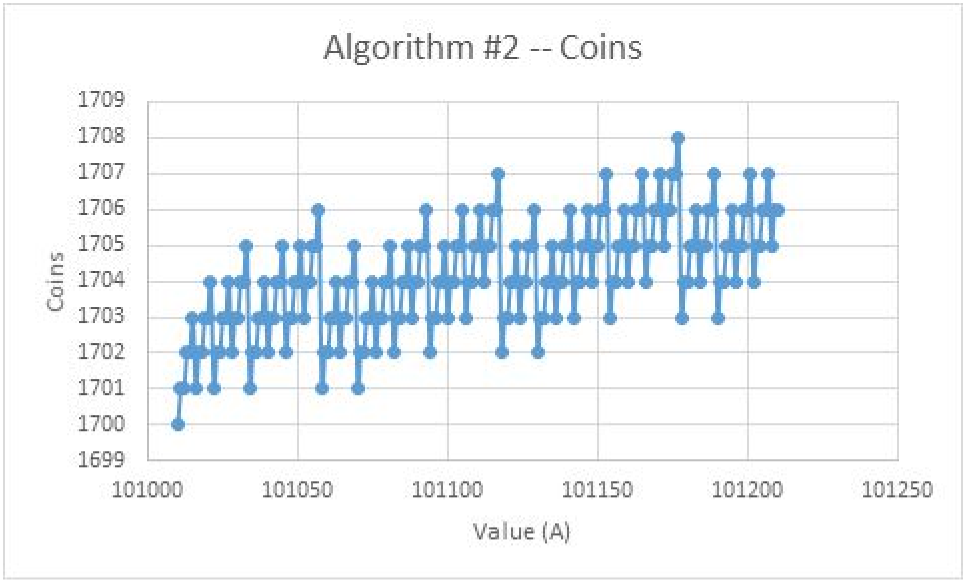


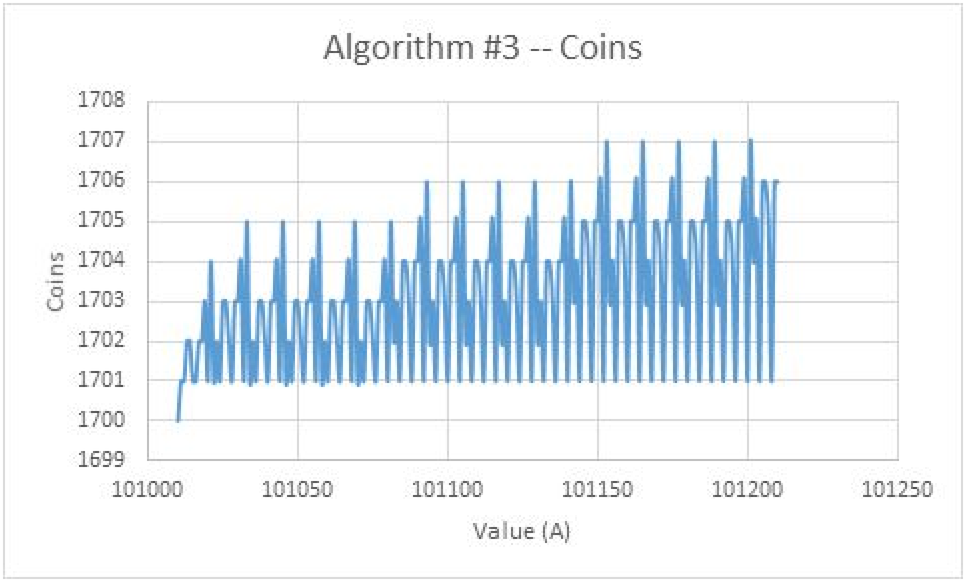
5. Suppose *V*1 = [1, 2, 6, 12, 24, 48, 60] and V2 = [1, 6, 13, 37, 150]. For each integer value of *A* in [2000, 2001, 2002, ..., 2200] determine the number of

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?

Figures 5A below shows the size of the sets of coins that were discovered by the algorithms 2 and 3 using denominations in V1.

Figures 5A

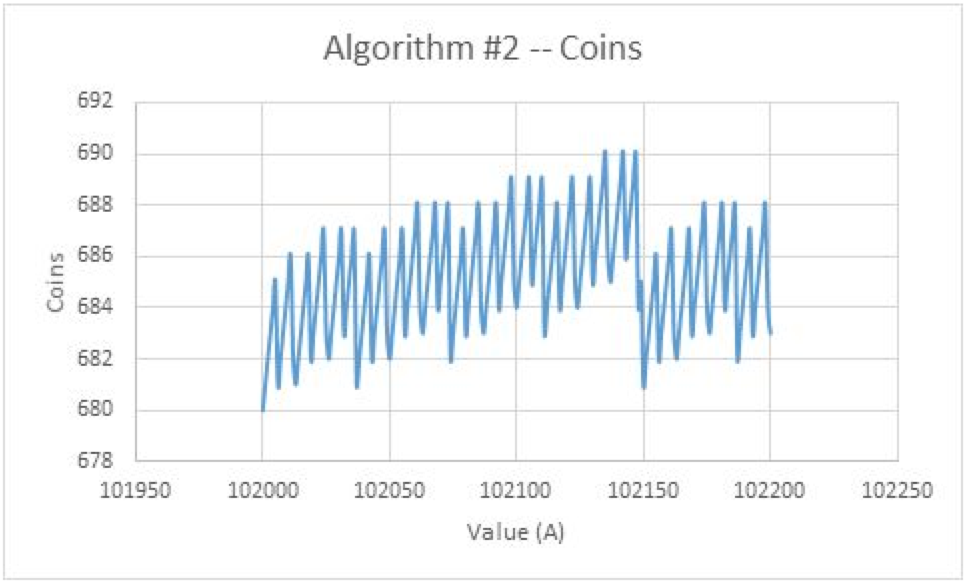


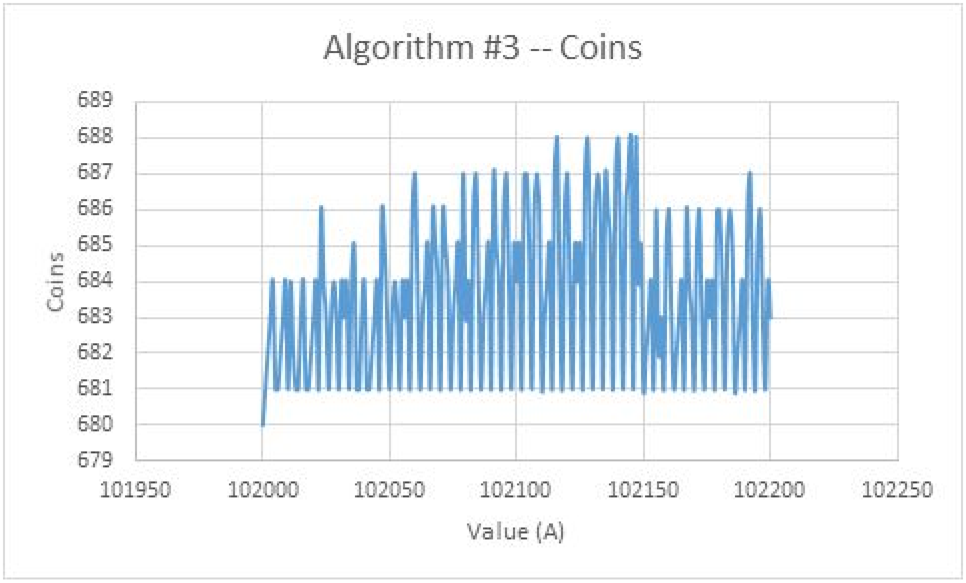


For values V1, the values returned by both algorithms were very similar. There were some values of A that were different by 1-2 coins, which is why the two graphs are no exact mirrors of each other. After analyzing the data, every time the number of coins differed between the two algorithms, Algorithm #3 (DP) used less coins than Algorithm #2 (Greedy).

Figures 5B below shows the size of the sets of coins that were discovered by the algorithms 2 and 3 using denominations in V2.

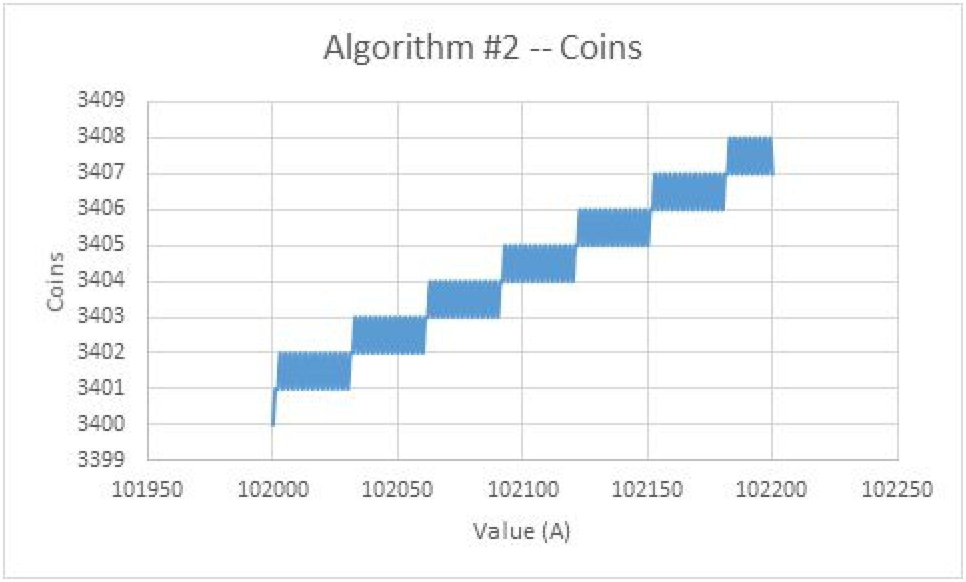
Figures 5B

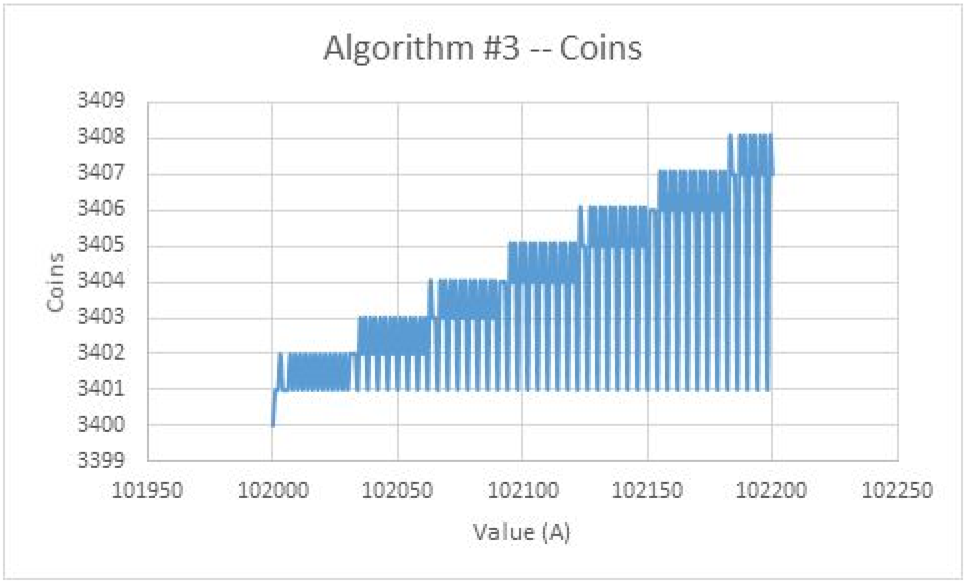




Similar to the experimental analysis of V1, V2 had very similar coin counts between the two algorithms. The numbers differed slightly on some values, usually by 1-2 coins. When they did differ, Algorithm #3 (DP) required fewer coins than algorithm #2 (Greedy).

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.

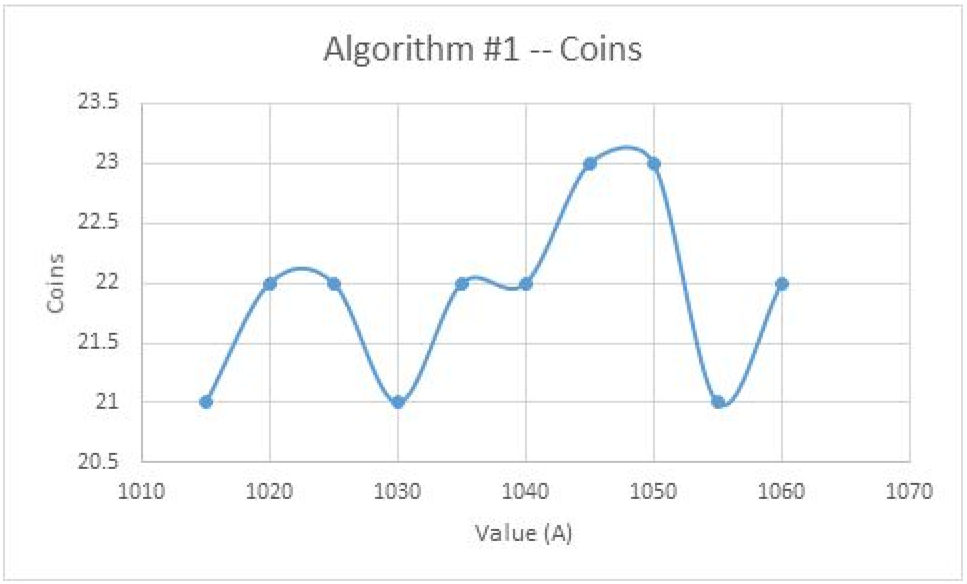


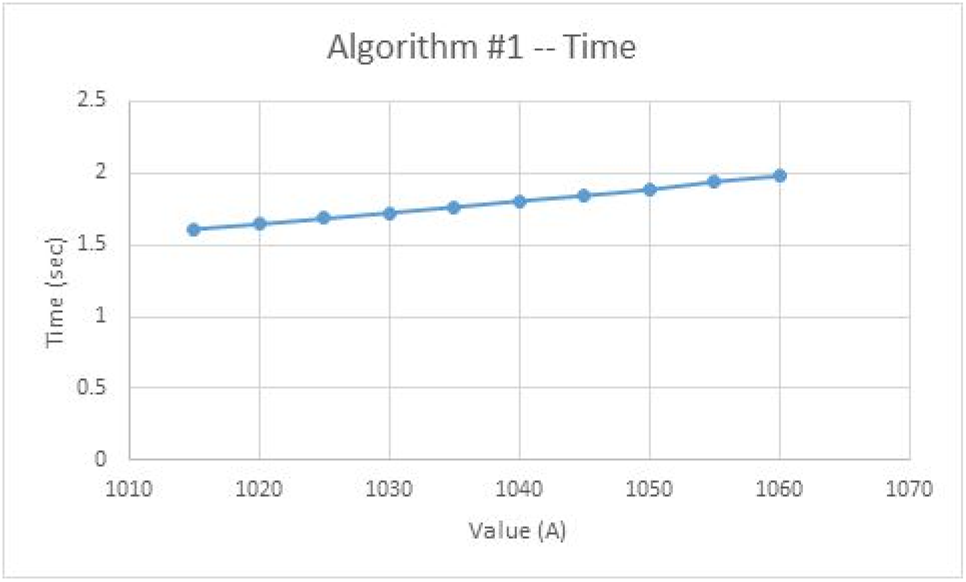


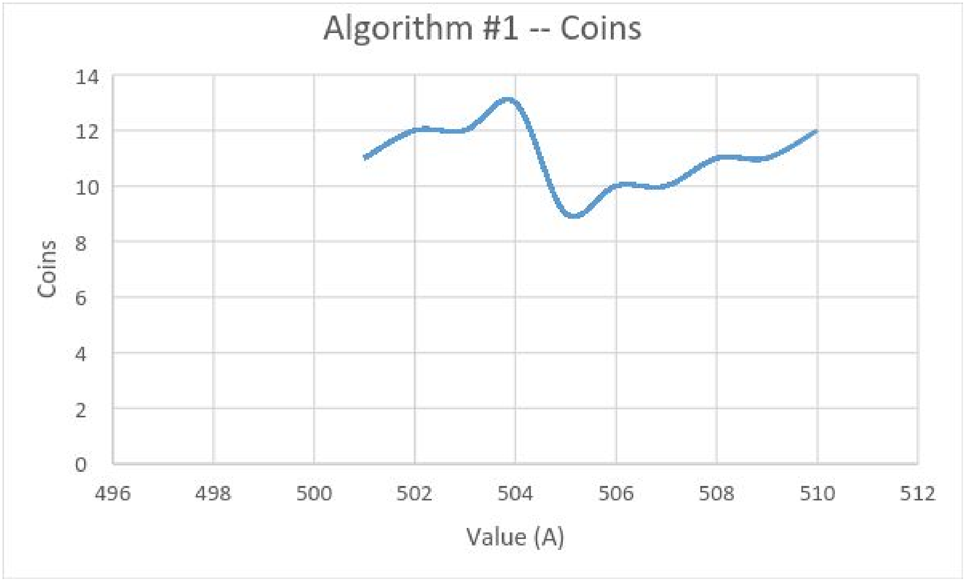
For the most part, the algorithms returned the same amount of coins. For some values of A though, algorithm #3 required less coins than algorithm #2. While algorithm #2 and #3 had a steady increase in the amount of coins needed, algorithm #3 always had sharp drops in the number of coins needed for some values of A. The number of coins needed dropped to 3401, which was 1-2 coins less than algorithm #2 early on, and rose to a maximum of 7 less coins for greater values of A.

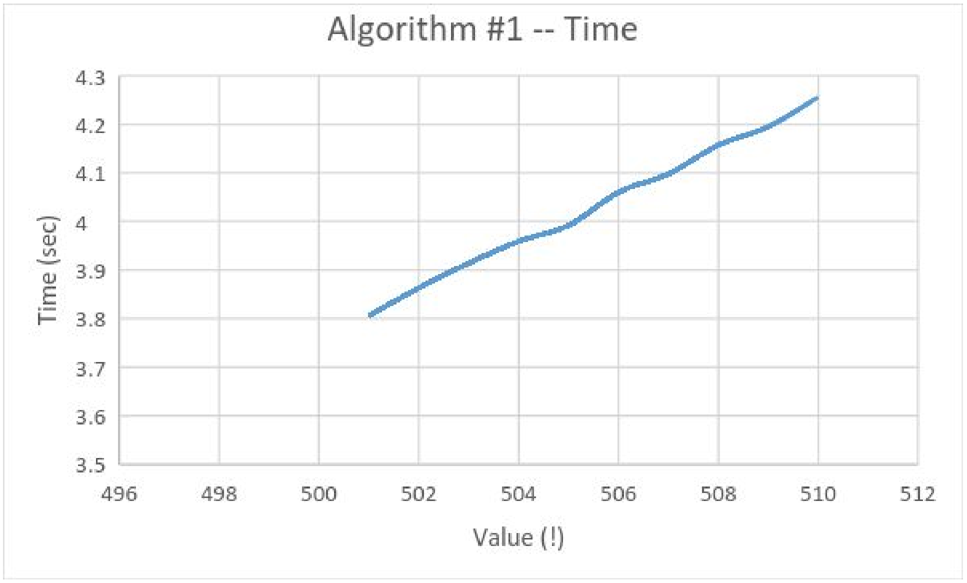
7. For the above situations, determine (experimentally) the running times of the algorithms by fitting trend lines to the data or analyzing the log-log plot. Graph the running time as a function of A. Compare the running times of the different algorithms.

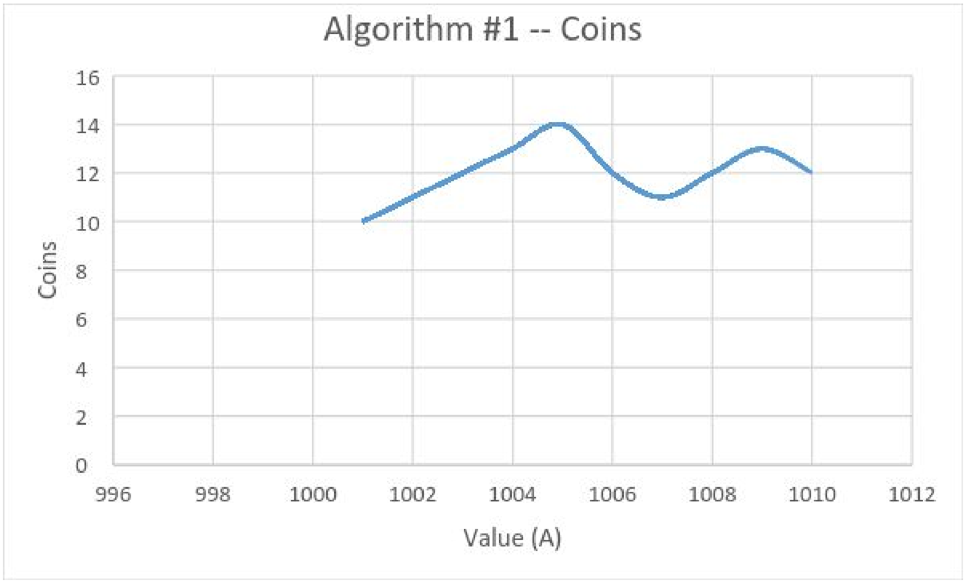
For all scenarios for Algorithm #1, the running times were very slow. To solve this issue, we ran Algorithm 1 with much smaller numbers than for the other Algorithms. Four sample sizes were used to analyze the different scenarios using Algorithm #1. These scenarios are shown below.

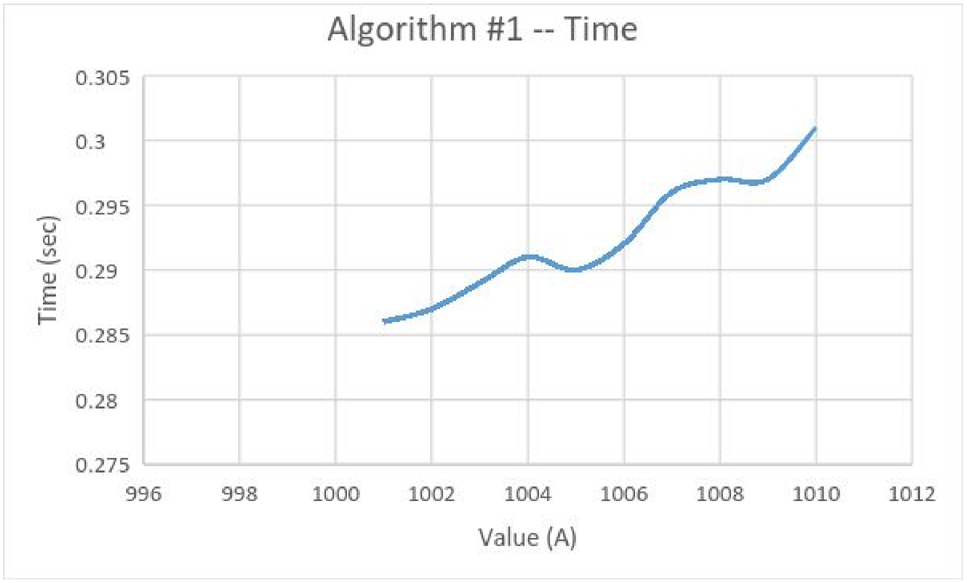


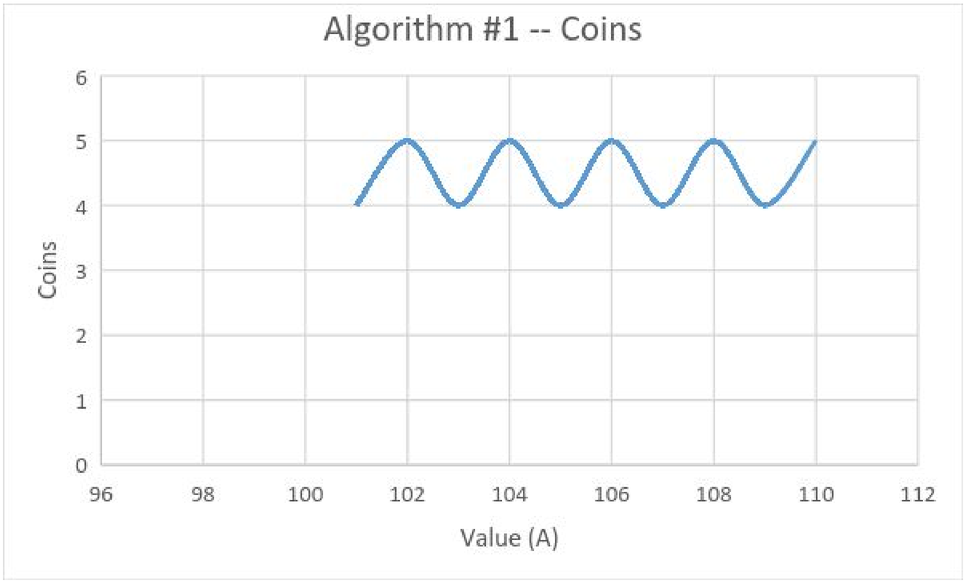

Algorithm #1 -- Time
Time (sec)
2.5
2
1.5
1
0.5
0
1010
1020
1030
1040
1050
1060
1070
Value (A)

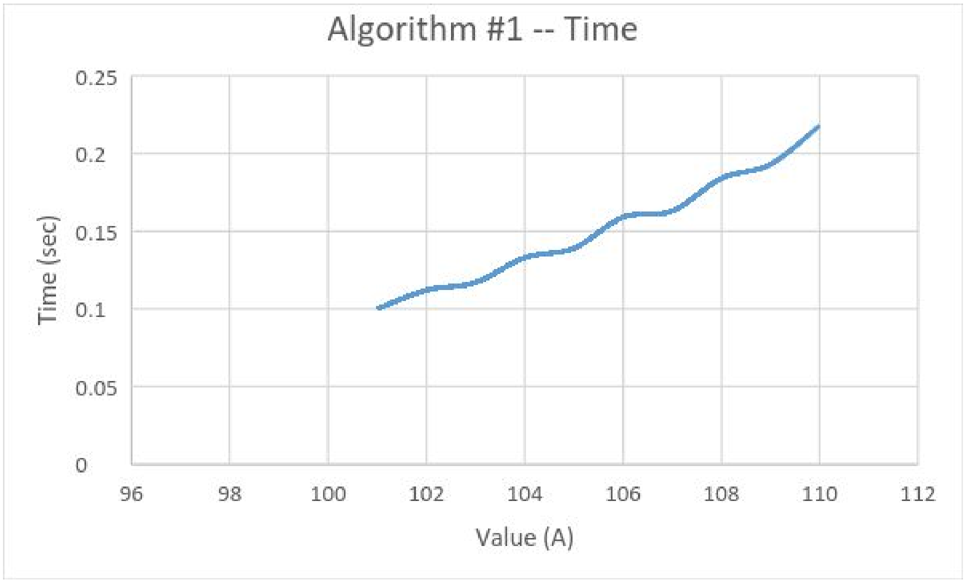

Algorithm #1 -- Coins
Coins
14
12
10
8
6
4
2
0
496
498
500
502
504
506
508
510
512
Value (A)


Algorithm #1 -- Time
Time (sec)
4.3
4.2
4.1
4
3.9
3.8
3.7
3.6
3.5
496
498
500
502
504
506
508
510
512
Value (!)

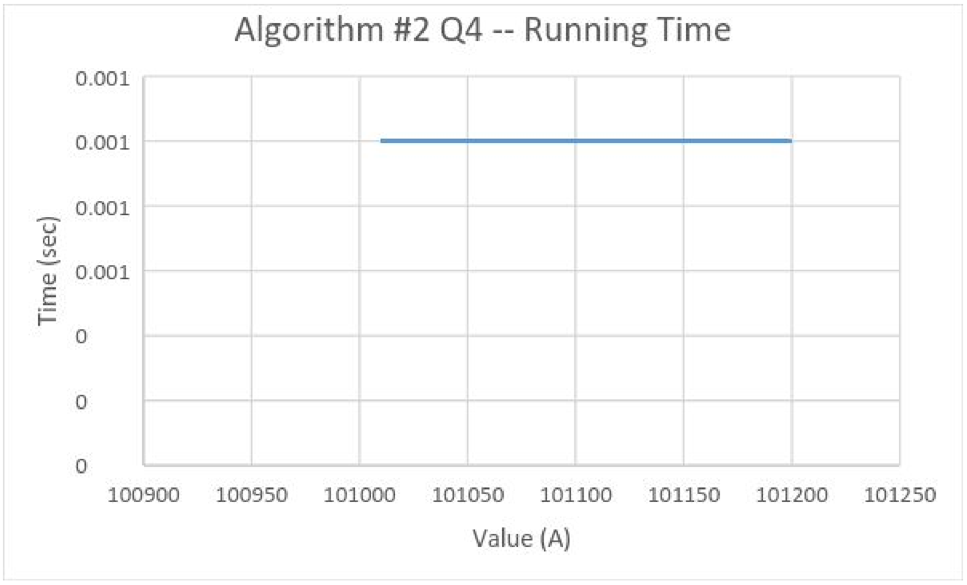

Algorithm #1 -- Coins
Coins
16
14
12
10
8
6
4
2
0
996
998
1000
1002
1004
1006
1008
1010
1012
Value (A)

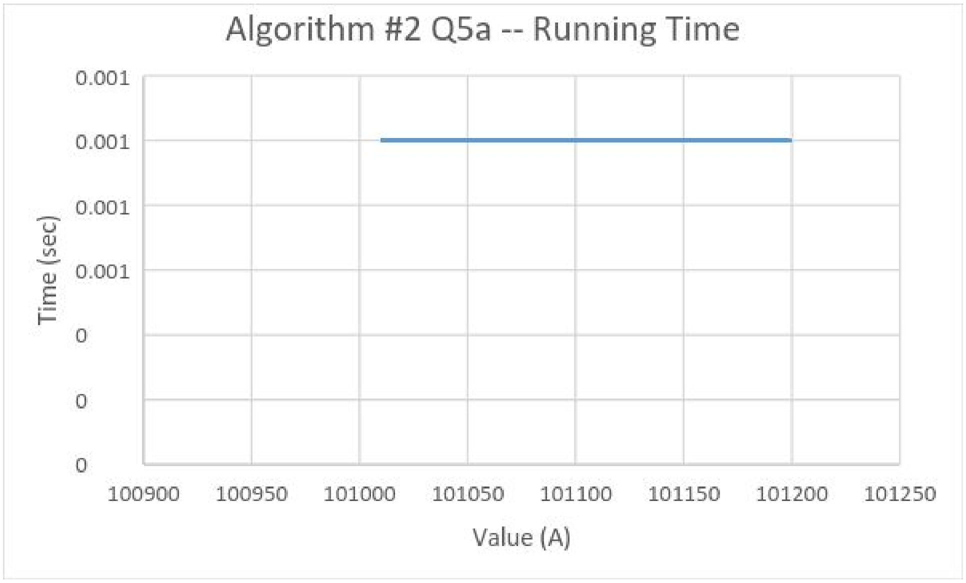

Algorithm #1 -- Time
Time (sec)
0.305
0.3
0.295
0.29
0.285
0.28
0.275
996
998
1000
1002
1004
1006
1008
1010
1012
Value (A)

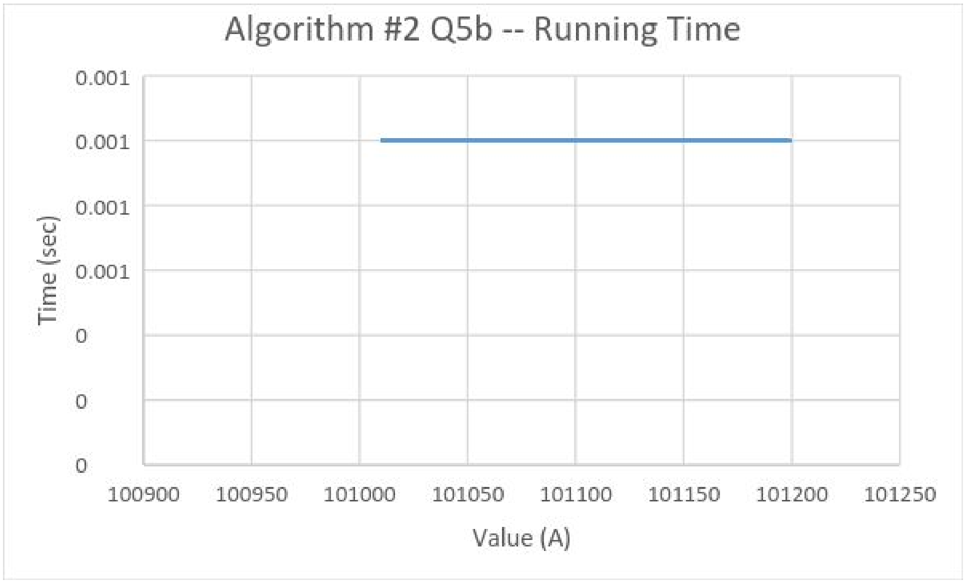

Algorithm #1 -- Coins
Coins
6
5
4
3
2
1
0
96
98
100
102
104
106
108
110
112
Value (A)

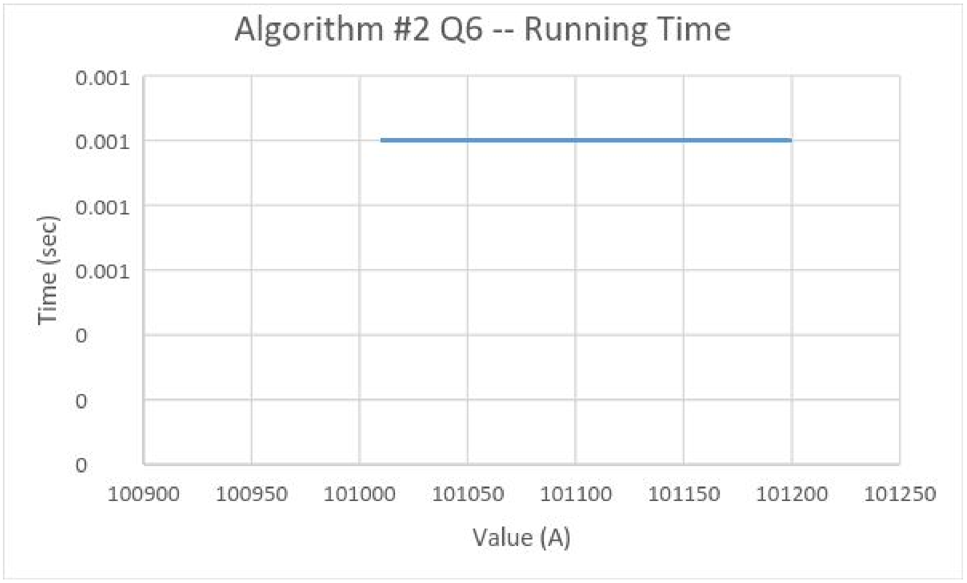


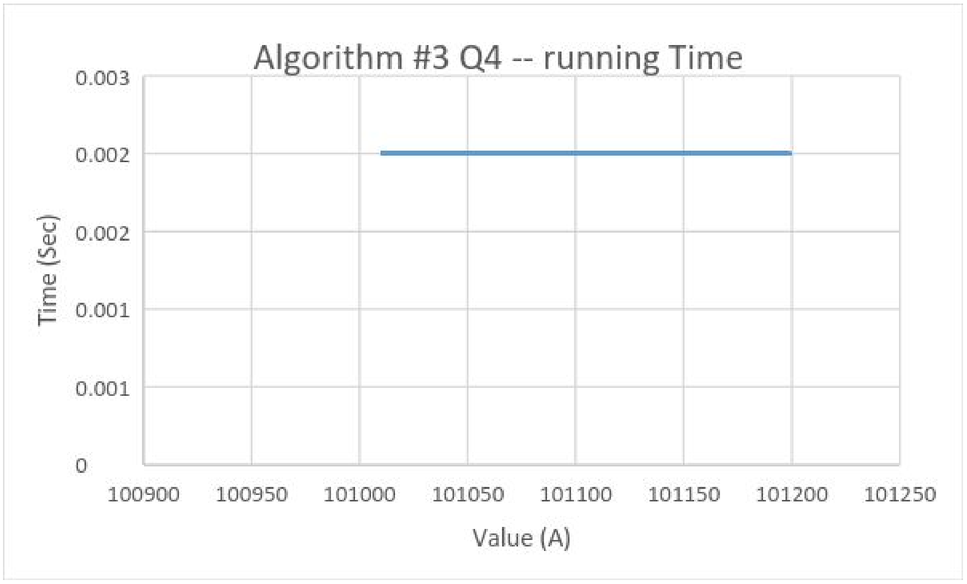
The running times for Algorithms #2, and #3 are shown below, the sample sizes are shown in the X-axis (Value (A)).

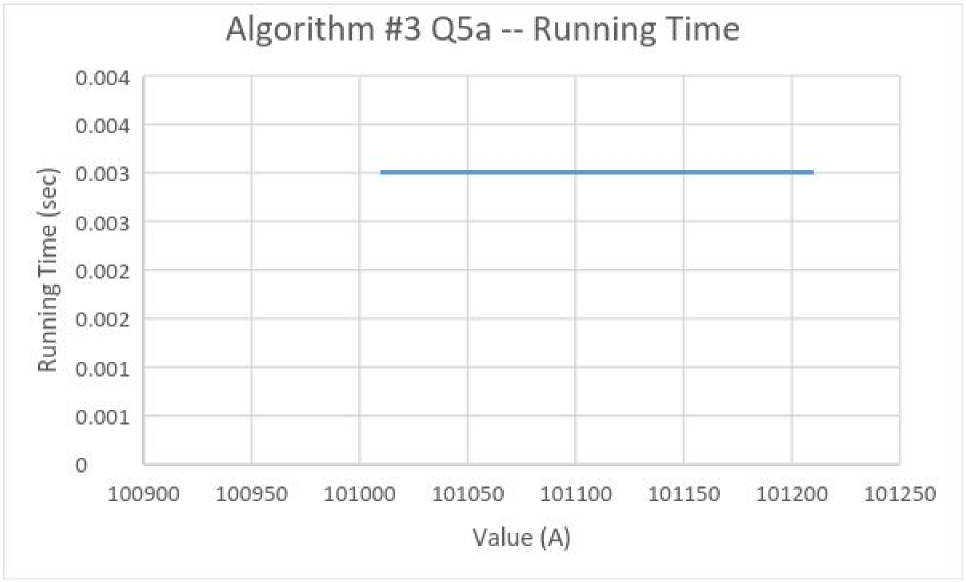


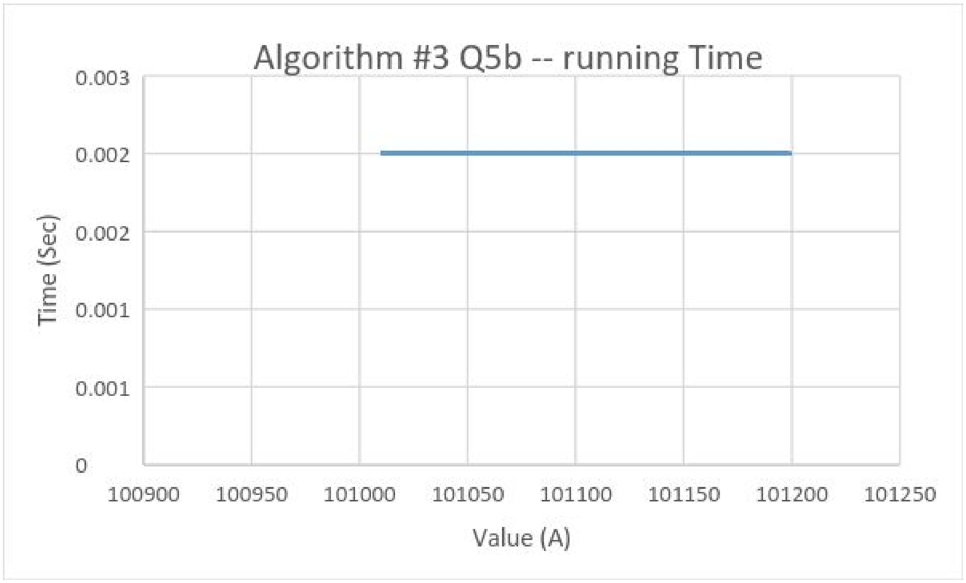

Algorithm #2 Q5a -- Running Time
Time (sec)
0.001
0.001
0.001
0.001
0
0
0
100900
100950
101000
101050
101100
101150
101200
101250
Value (A)

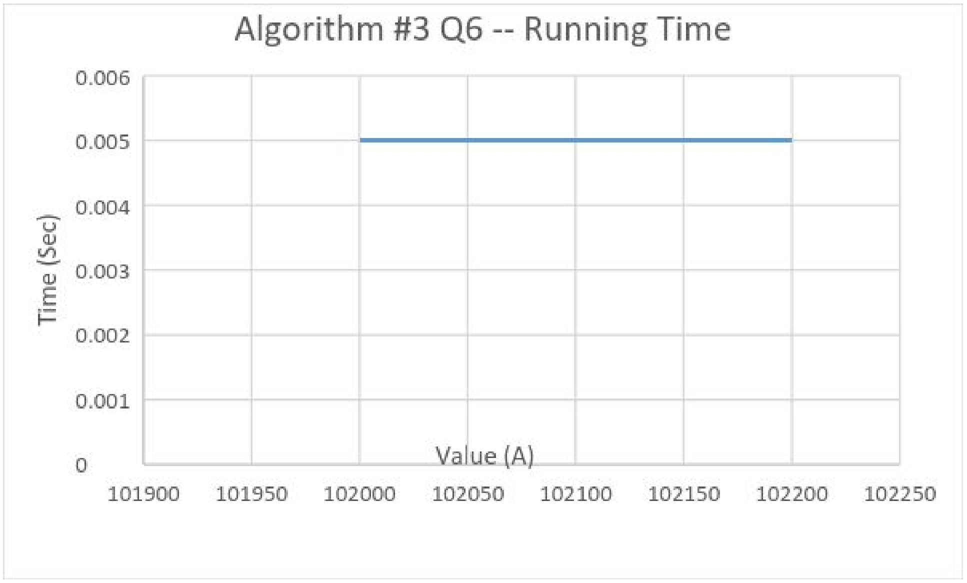

Algorithm #2 Q5b -- Running Time
Time (sec)
0.001
0.001
0.001
0.001
0
0
0
100900
100950
101000
101050
101100
101150
101200
101250
Value (A)


Algorithm #2 Q6 -- Running Time
0.001
0.001
0.001
0.001
0
0
0
Time (sec)
100900
100950
101000
101050
101100
101150
101200
101250
Value (A)


Algorithm #3 Q4 -- running Time
0.003
0.002
0.002
0.001
0.001
0
Time (Sec)
100900
100950
101000
101050
101100
101150
101200
101250
Value (A)


Algorithm #3 Q5a -- Running Time
Running Time (sec)
0.004
0.004
0.003
0.003
0.002
0.002
0.001
0.001
0
100900
100950
101000
101050
101100
101150
101200
101250
Value (A)

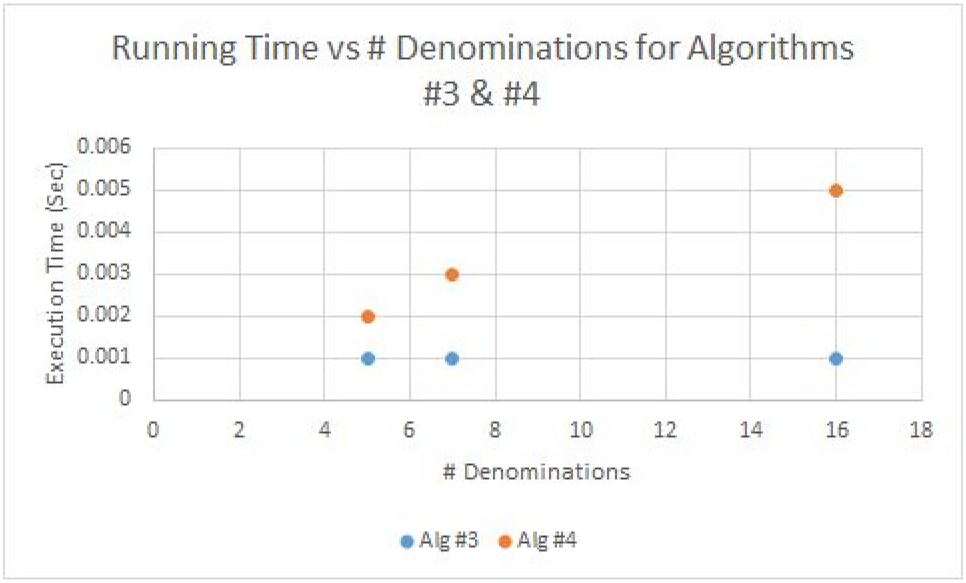

Algorithm #3 Q5b -- running Time
Time (Sec)
0.003
0.002
0.002
0.001
0.001
0
100900
100950
101000
101050
101100
101150
101200
101250
Value (A)

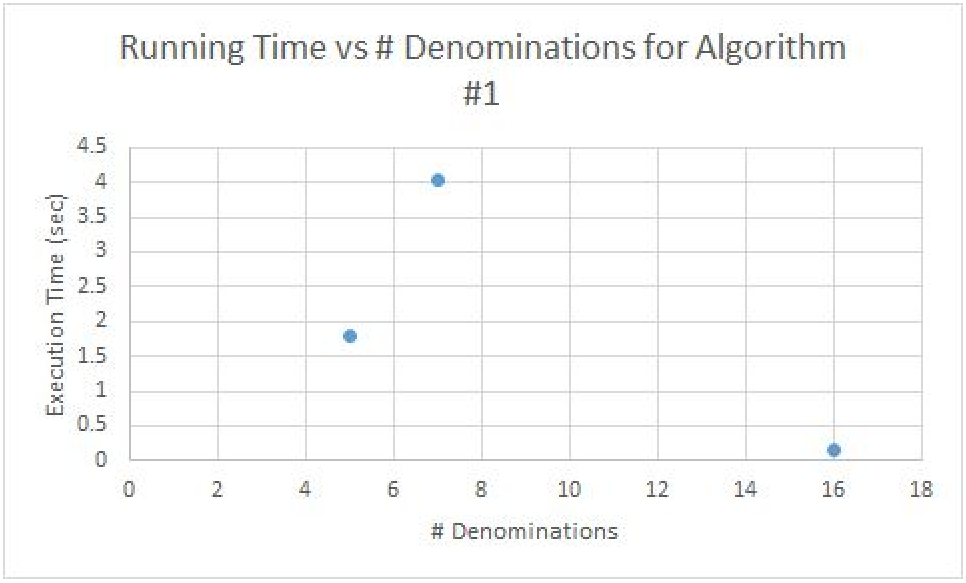


The running times for algorithm #2 and #3 were very fast. Algorithm #2 was by far the fastest, followed by #3. These two algorithms were so fast, that we had to increase the value of A to over 100,000 to get any measureable data. During execution, it was found that no matter what the value of A, algorithm #2 and #3's execution time was constant, as evidenced by the horizontal lines.

Algorithm #1 on the other hand, was considerably slower than the other two algorithms. We had to decrease the values of A to 1000 in order for a realistic test to occur in a reasonable amount of time. Algorithm #1's execution times were fairly linear. The higher the value of A, the slower the execution time.

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?





For algorithm #1, the running times vs the number of denominations was a polynomial function. The highest execution time had 7 denominations. The lowest execution time by far was that with 16 denominations, followed by one with 5 denominations.

For Algorithm #2, the amount of time required for each number of denominations were the same. It was very fast no matter the value of A, or the number of denominations.

For algorithm #3, the running time was proportionate to the number of denominations being calculated for A. The fastest execution time correlated with the least amount of denominations, while the slowest execution time was with by far the highest amount of denominations.

The number of denominations has an effect on algorithms #1 and #2, but not for #3.

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? Explain.

In Q 4-6, we saw that Greedy sometimes arrives at a suboptimal solution. DP by contrast always finds the minimum set of coins to give change on some value (Q3). For certain sets of coin denominations however Greedy is guaranteed to arrive at an optimal solution. One such set is *V* = {p^0 … *p*^n}.

The optimal solution with such a set (regardless of algorithm) follows a regular pattern:

line 1. V = 0 → sol = {};

line 2. V in range 1:(p-1) → sol = {p^0, repeated some number *a* of times until total < p^1);

line 3. V in range p:(p*p-1) → sol = {p^1 repeated some number b of times until total < p^2 + solution to the remainder value as found on line 1);

line 4. For every p-sized step in V → sol = {p^(step iteration number) repeated some number of times until total < p^(step iteration number + 1) + the accumulated solution to remained.}

The closed form of the solution is Set = a.p^n + b.p^(n-1) … z.p^0, with some coefficient set Coef = {a, b...0}.

The reason the two algorithms converge on the same (optimal) solution is that the coins in set *V* = {p^0 … *p*^n} have a multiplicative relationship to each other. That is: p^k = p.p(k-1). As a result the family of sets of coefficients Coef = {a, b … 0} introduced in the closed form solution above has three properties (1) each member set Coef produces the same total value when applied to the coin denomination set, (2) one Coef set associates higher coefficients to higher denominations than the other sets do, and (3) the latter set also has the fewest number of coin/elements.

A DP algorithm finds this optimal set of coefficients by working its way through all possible combinations. A greedy algorithm finds it because it works its way down through the set of coins C from the highest denomination to the lowest, effectively building the largest coefficients possible for the highest denominations.

In terms of execution time, Greedy would be quicker since it works from the largest denominations down (contrary to DP) and the optimal solution favors higher denominations whenever possible.