Ben Ford
John Lewis Kilgore
Gunnar Martin
CS325 - Spring 2016

Project #2 Report - Group 6

Item 1: Pseudocode and Running Time

Brute Force (changeslow)

Pseudocode

```
def changeslow(denomArray, targetAmt):

      bestArray = [targetAmt]
      bestCoinCount = targetAmt

      totalCombos = 1; valArray = []
      for i in range(1, len(denomArray)):
            valArray.append(totalCombos)
            bestArray.append(0)
            totalCombos = totalCombos * (targetAmt / denomArray[i] + 1)

      for j in range(0, totalCombos):
            coinArray = idToArray(j, valArray)
            v0 = 0;      coinCount = 0
            data = getTotal(coinArray, denomArray)
            total = data[0]

            if total <= targetAmt:
                  v0 = (targetAmt - total)
                  total = total + v0
                  coinCount = data[1] + v0
                  coinArray.insert(0,v0)

                  if coinCount < bestCoinCount:
                        bestArray = coinArray
                        bestCoinCount = coinCount

      return [bestArray, bestCoinCount]

def idToArray(dividend, valArray):
      currArray = []

      for i in range(len(valArray)-1,-1,-1):
            currArray.insert(0,(dividend / valArray[i]))
            dividend = (dividend % valArray[i])

      return currArray
```

```
def getTotal(coinArray, denomArray):
      total = 0; coinCount = 0

      for i in range(1, len(denomArray)):
            total += coinArray[i-1] * denomArray[i]
            coinCount += coinArray[i-1]

      return [total, coinCount]
```

Asymptotic Running Time

For changeslow, we chose to do a brute force implementation to solve the problem. The goal of this algorithm is to iterate through every possible combination of coins that could add up to the target amount of coins. First, the algorithm loops through the coin denominations array (denomArray) to determine the total combinations of coins that could be used to get to the target coin amount (targetAmt). Next, it loops through each combination to check the total amount of the coins. As it checks the total value of the coins, it stores the array that has be lowest count of coins. The asymptotic runtime of this algorithm is heavily dependent on 2 things. The first is the amount of change we are asked to make. The second is the number of coins in the array V, the denominations we have to work with.

The driving factor of this asymptotic runtime of this algorithm will be how many denomination of coins will be passed into the denominations array that are less that the target value of A. This will determine how many combinations of different coins we have to try and drive the runtime of the main “loop” of our algorithm that iterates through every combination. If we suppose that A = 5, the worst-case scenario of the array that gets passed in is V = [1, 2, 3, 4, 5]. For each denomination (excluding v[0]), the algorithm will be limited to loop through each count of that denomination, limited to the amount of that denomination that makes the total greater than A. In the case above, there would be 3 possibilities for v[1] (5/2=2+1=3). There would be 2 possibilities for v[2] (5/3=1+1=2). And there would also be 2 possibilities for v[3] and v[4]. So the total valid combinations would be 3*2*2*2=24. To try and see a pattern, if we increased A to be 6 ,the worst-case array would be V = [1, 2, 3, 4, 5, 6] and the total valid combinations would be 4*3*2*2*2=672. If we increase again to A=7, the total combinations is 4*3*2*2*2*2 = 1,344. As we can see, when we increase A by one, the runtime gets worse by at least a factor of 2. So we know that the runtime is bounded below by 2^A. We also know that it will not be worse that a factor of A, so we know that the runtime is bounded above by A^A. So...

$T(A) = O(A^A)$ and
$T(A) = \Omega(2^A)$

Assuming a worse-case scenario array V, where V[0]=1 and V[1]>V[2]>....V[A]=A, such that there is a V[i] for each positive integer between 1 and A.

Greedy (changegreedy)

Pseudocode

```
def changegreedy(coinDenominations, valueOfChange):
      #set number of coins that were used to 0
      numberOfCoinsUsed = 0
      #set an array of 0s that is the same length of the coinDenomicaitons
      arrayOfCoinisUsed[0] = []
      i = length of coinDenominations
      #loop through the coinDenominations array until the valueOfChange = 0
      while(valueOfChange > 0):
            #if the largest available coin is <= remaining change value use
            that coin
            if(coinDenominations[i] <= valueOfChange):
                  #subtract the largest coin available from the remaining
                  value
                  valueOfChange-= coinDenominations[i]
                  #increase the spot of the coin used
                  arrayOfCoinisUsed[i] += 1
                  #increase the number of coin used amount
                  numberOfCoinsUsed  += 1
            Else:
                  #decrease in the coinDenominations to a lower value coin
                  i -= 1
      return [arrayOfCoinisUsed, numberOfCoinsUsed]
```

Asymptotic Running Time

For changegreedy, we are using a naive greedy approach to solve the coin problem. This approach uses the largest coin that is possible until only the coin with the value of 1 can be used. In the worst case run time would be O(n) where the minimum coin is used every time. The best case would be that the largest coin is the value we are trying to find so Ω(1).

Dynamic Programming (changedp)

Pseudocode

```
def changedp(coinDenominations, valueOfChange):
   #set up array to hold minimum coins for each value
   minimumNumberOfCoins = [None for x in range(valueOfChange + 1)]
   #base: value of zero is no coins
   minimumNumberOfCoins[0] = []

   #look at each value from 1 up to the valueOfChange argument
   for i in range(1, valueOfChange + 1):
      for coin in coinDenominations:
         # if coin is bigger than current i, then skip it
         if coin > i: continue
         #else if, minimum is zero OR (current minimum - coin value + 1) is less than current minimum,
         #new minimum value has been found!
         elif not minimumNumberOfCoins[i] or len(minimumNumberOfCoins[i - coin]) + 1 <  len(minimumNumberOfCoins[i]):
            if minimumNumberOfCoins[i - coin] != None:
               #new minimum is found
               minimumNumberOfCoins[i] = minimumNumberOfCoins[i - coin][:]
               minimumNumberOfCoins[i].append(coin)

   minimumCoinSolution = [0]*len(coinDenominations)
   for c in minimumNumberOfCoins[-1]:
      minimumCoinSolution[coinDenominations.index(c)] += 1

   #return the number of coins in the minimum value (size of last in minimumNumberOfCoins)
   return [minimumCoinSolution, len(minimumNumberOfCoins[-1])]
```

Asymptotic Running Time

This algorithm uses a dynamic programming implementation to return the minimum number of coins for a given value and a given array of coin denominations. It looks to fill up the minimumNumberOfCoins table with the min value at each integer from 1 up to the valueOfChange. Next it will look at each coin in the coinDenominations arrays(it only considers coins less than the current value it is looking at). If the current minimum has zero coins or it meets the condition that the number of coins in the current minimum minus the value of the current coin plus 1 is greater than the length of the current minimum, then a new minimum is found. It then assigns the new minimum value and keeps track of the new coins that are added with that minimum. It then goes and count the number of each denomination used in the minimum solution (which is the last element in minimumNumberOfCoins). The function returns the number of coins used in the minimum solution and the count of each denomination used in the minimum solution. This algorithm is expected to run with a linear time multiplied by the number of subproblems in has too look at before the value.

Item 2: Filling the Dynamic Programming Table in changedp

A “bottoms up” approach is needed to fill a possible table of solution when using the Dynamic Programming algorithm. Once the table is filled you will look to choose an optimal solution at that given value.

To create the table you would have a row for each coin denomination(sorted from minimum to maximum) and a column for each value 0 through N. In each table cell we will be assigning a number for the “minimum number of coins” needed to get the value in that column.

When filling each entry in the table, start with the smallest denomination and calculate how many of that coin it takes to get the value specified by that column’s header. Then move down to the next denomination row. In this second row, calculate how many coins of this denomination it would take to get the value specified in the column header. Then compare this minimum number to the one record in the first row’s cell in the same column. Whichever value is lower is recorded into that cell on the second row.

This process is repeated for each larger denomination row. Each time using the cell directly above the one you are looking to record a new number for the minimum comparison, since this cell will always hold the current minimum amount of coins for that value column.

Justification for filling the table this way: By the time you are done filling out the table, the last row will have the value for the overall minimum amount of coins for each column value. You would also be able to calculate which coins make up this minimum.

Item 3: Proof of Correctness for the Dynamic Programming Approach

Prove that:
$T(v) = \min_{V[i] <= v}\{T[v - V[i]] + 1\}, T[0] = 0$
is the minimum number of coins possible to make change for value v

Base Case: T[0] = 0. This correctly finds the minimum coins to make “0” which is 0 coins.

By Induction:
assume the equation is true when v = k,
in other words that using the given equation we know that T(k) = “minimum coins needed to make k” is true .

Inductive Step:
Must now show that T(k+1) is also true.
At this point, the minimum amount of coins for T(k+1) will be one of the following:

Case 1:
It will be the minimum number of coins for T(k) plus one more coin, such as:
$T(k+1) = T(k) + 1 = \min\{T[k - V[i]] + 1\} + 1$, which is also true based on assumption that the T(k) was true

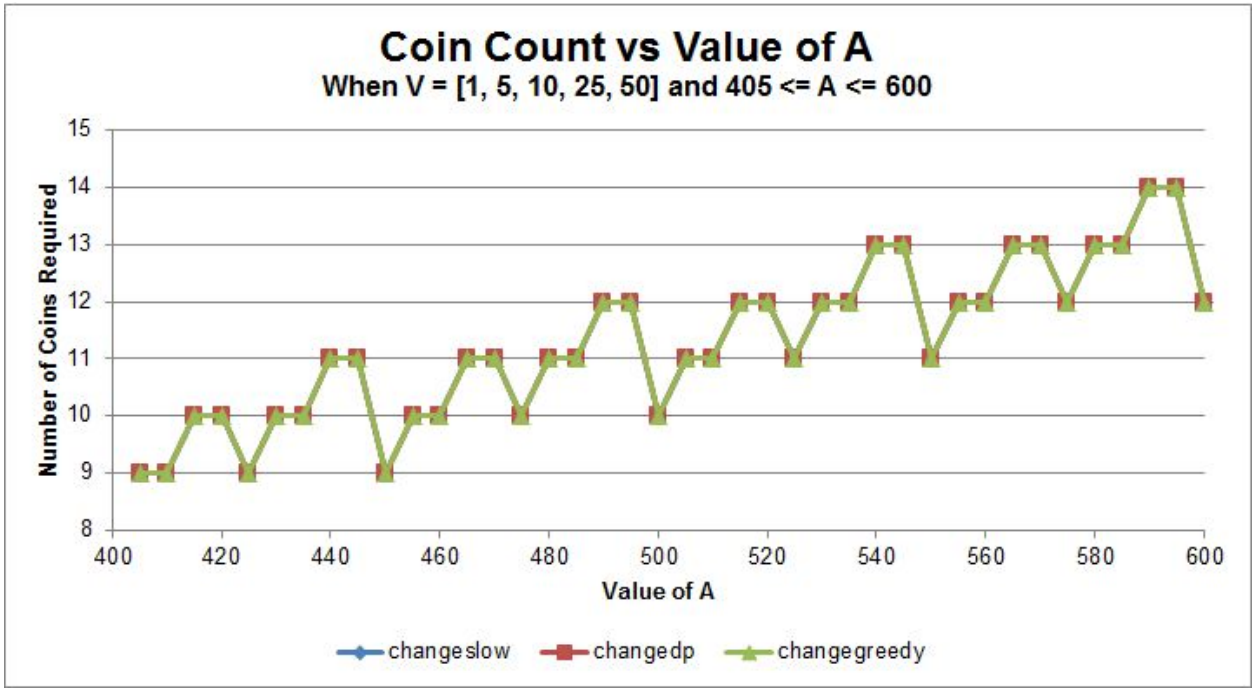
Case 2:
T(k+1) will have a solution that is smaller than the solution for T(k). Which will establish a new minimum for this value. This case is true since it is less coins than T(k), which was assumed to be a true minimum.

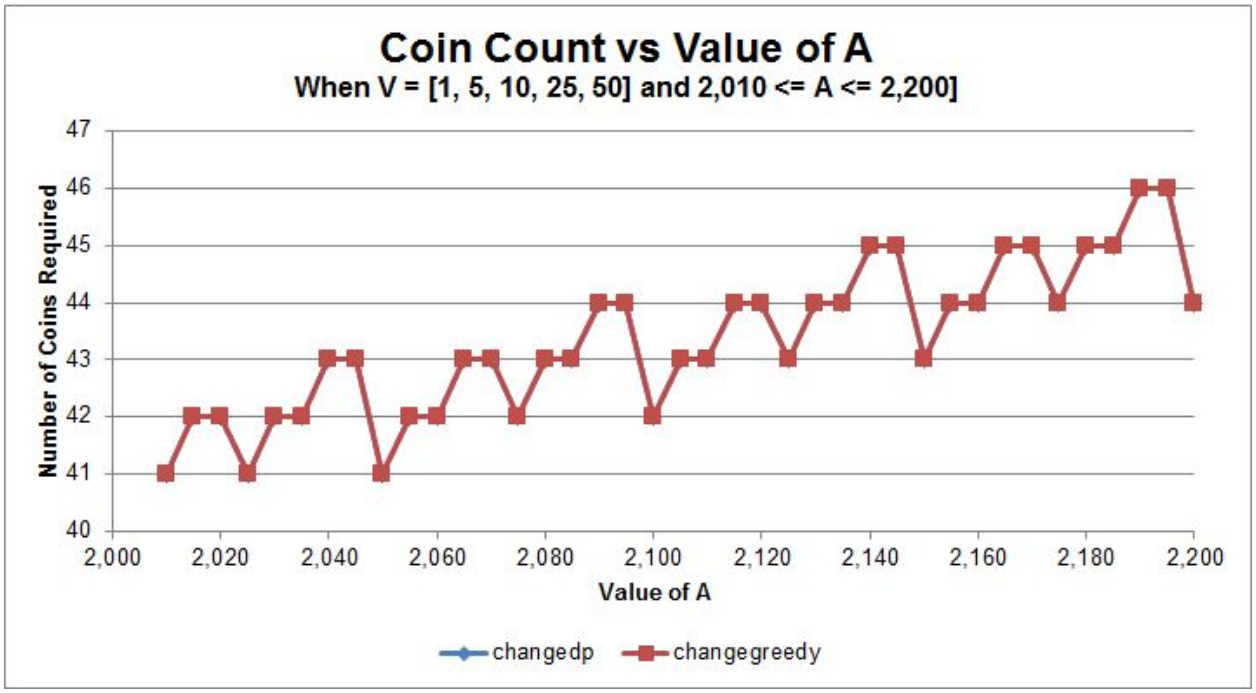
(**For example, when you reach the value threshold that can use a single larger value denomination rather than multiple smaller coins)

Both cases hold for T(k+1) therefore,

This shows that:
$T(v) = \min_{V[i] <= v}\{T[v - V[i]] + 1\}, T[0] = 0$
is the minimum number of coins possible to make change for value v

Item 4: Test Case #1

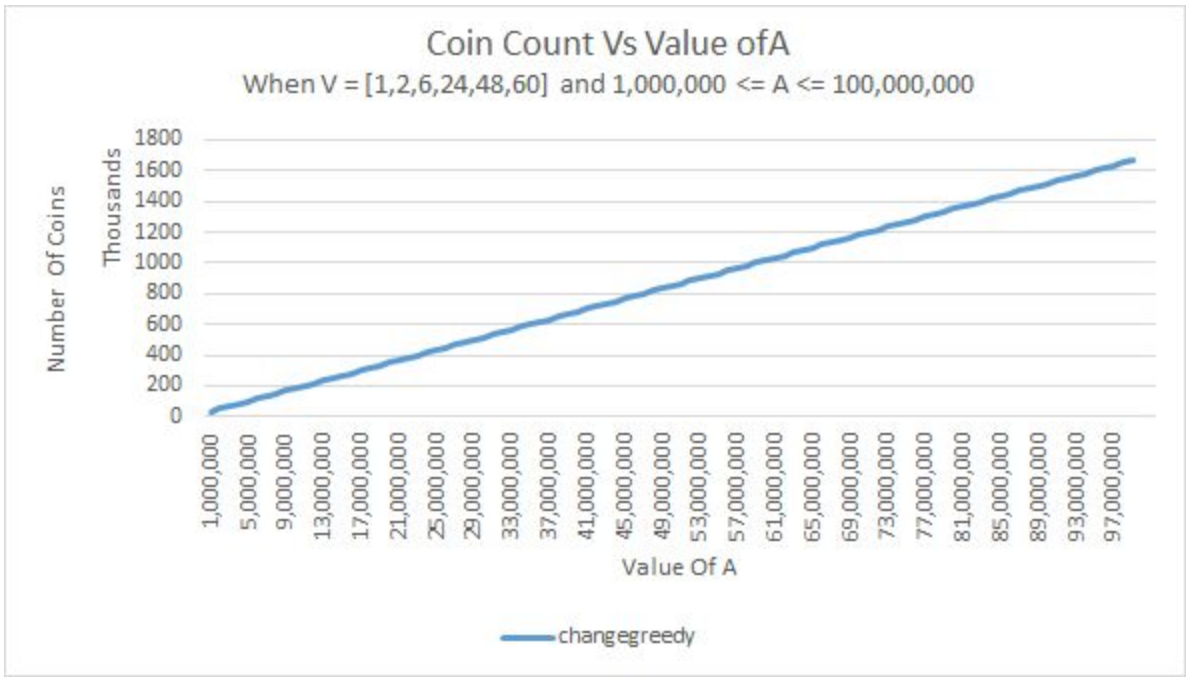


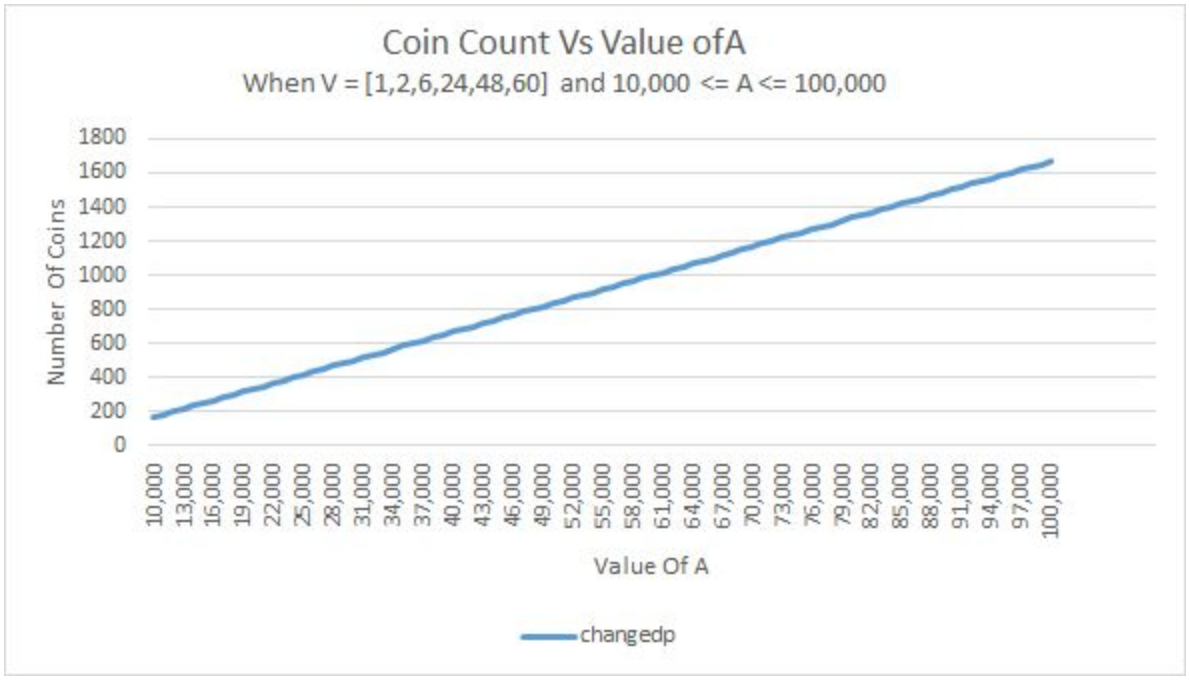


For our implementations of changeslow, changedp, and changegreedy, all 3 algorithms produced the same count of coins for each test value when V = [1, 5, 10, 25, 50] and A was in 5 integer intervals between 405 and 600. For the same V array, using A values in 5 integer interval between 2,010 and 2,200, the alogrithms changedp and changegreedy produced the exact same coin counts. The algorithm changeslow was not tested in the 2nd set of intervals, because it would take too long to run.

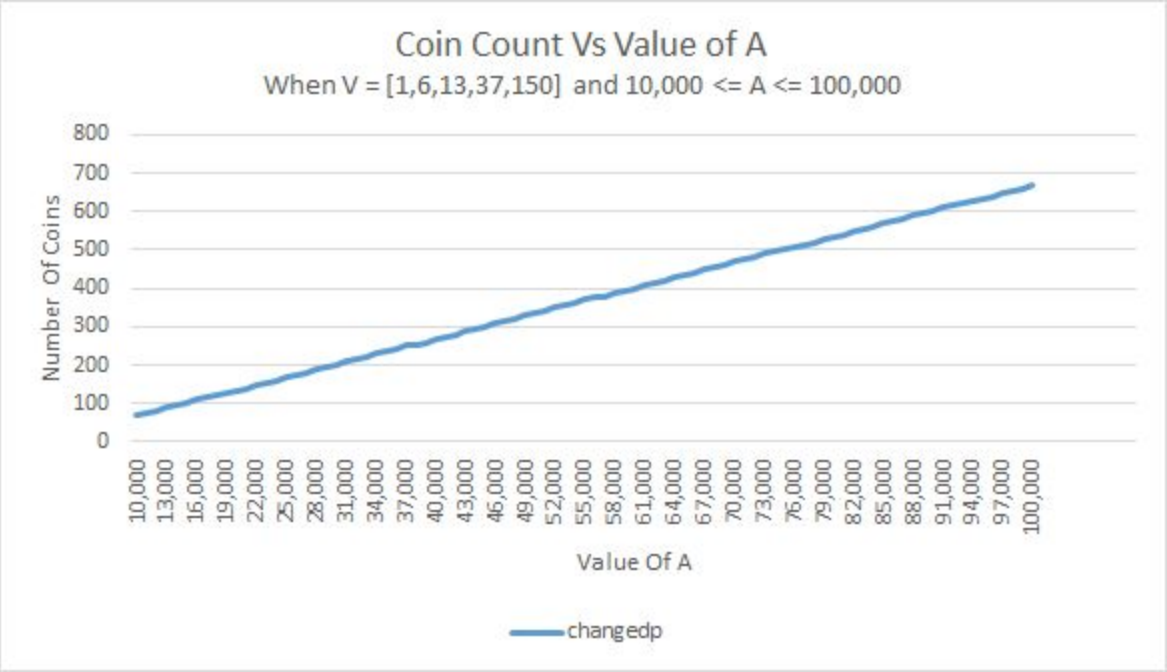
Item 5: Test Case #2

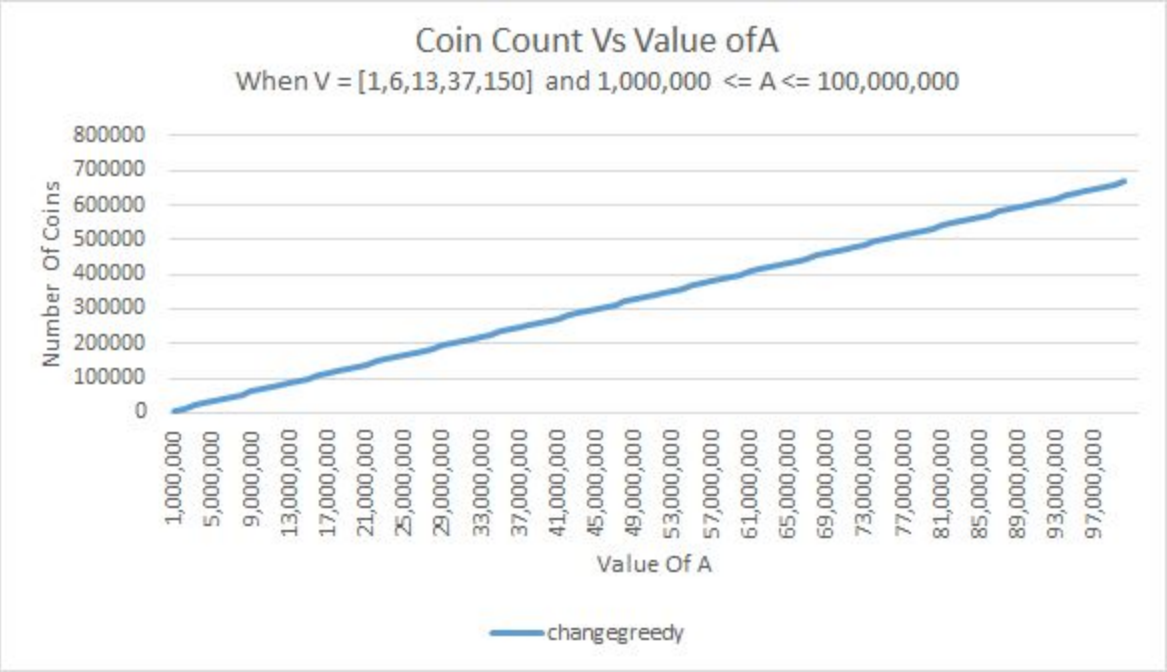
V_1 = [1,2,6,12,24,48,60]





V_2 = [1,6,13,37,150]





The results for Change Greedy vs Change DP were very similar. Overall Change DP was consistently more efficient in making change. There was a quite a bit of overlap though between the greedy and dynamic programming algorithms.

Item 6: Test Case #3

V = [1,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30]
A = [2000, 2001, 2002, 2003, … , 2199, 2200]

Where V is the change value and A is the coin denominations available

Here is a plot for number of coins as a function of A:

Note: That the values for the change values (in list “A”) were too large for for the “changeslow” algorithm to handle, so this plot only includes the results for changegreedy and changedp. Each of these two algorithm had them same exact solution for the number of coins for each of the change values in “A”. (orange line is directly on top of the blue line)

There was an oscillating pattern that developed for the number of coins for each A. For every 30 values, the number of coins for up by 1 for each odd number then back down by 1 for each even number. This pattern makes sense because every 30 values the next value will have the ability to choose another “30” valued coin which was the largest denomination in our given V array. Also, the odd numbered values had to add a “1” valued coin to the previous solution to get the number of coins, but the even numbered values could get rid of the “1” coins from the previous solution and replace it with the next even valued coin in the V array.

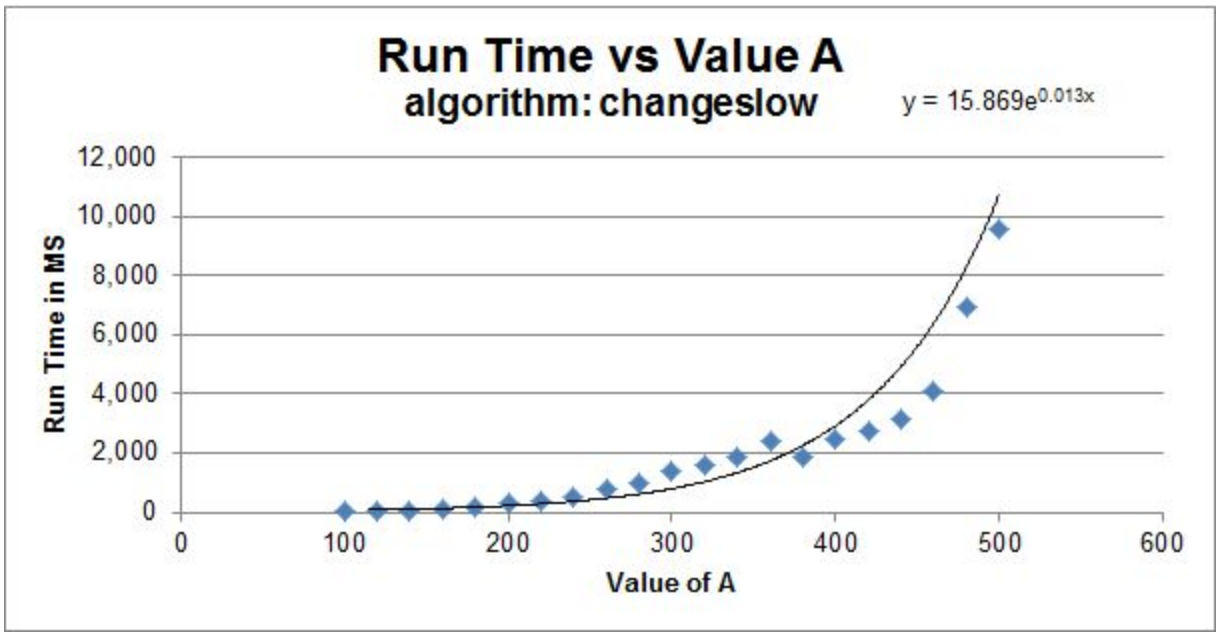
Item 7: Comparison of run times and value A (for Items #4-6)

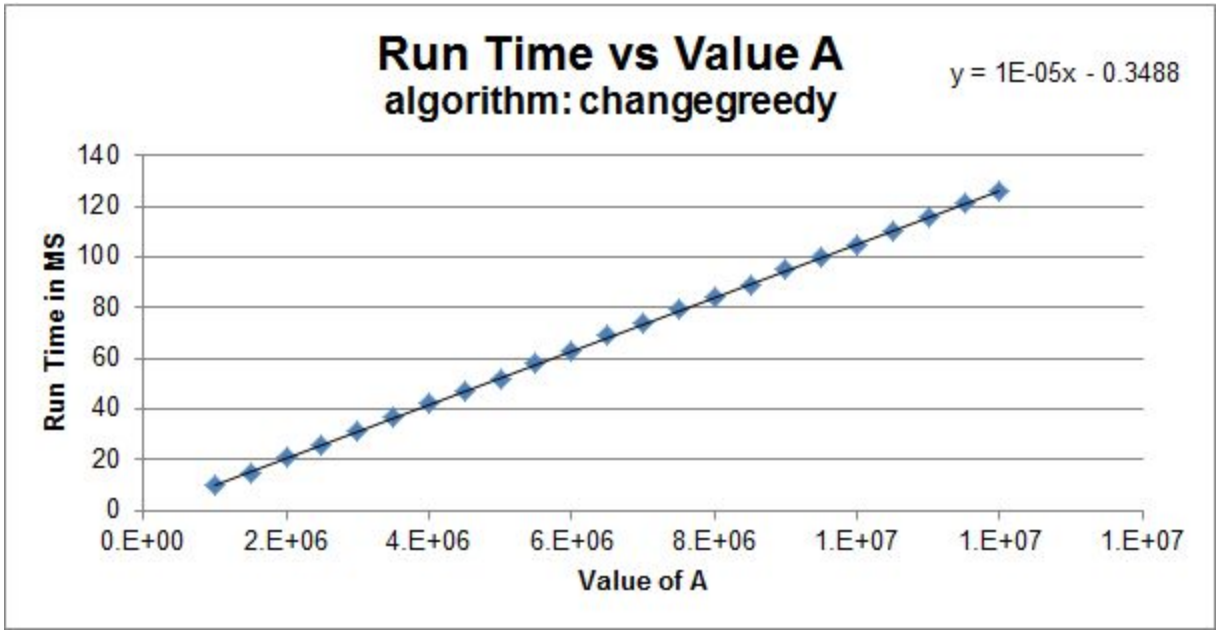
Scenario from Item #4:

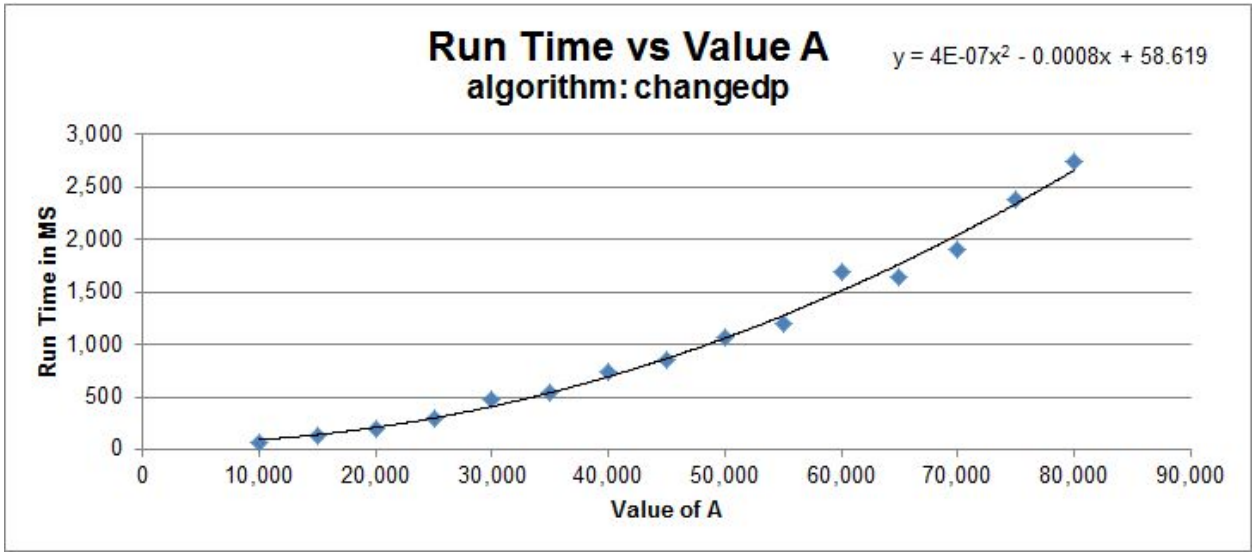
For the comparison of run times from the scenario in item #4, the following ranges of A were used for each algorithm:

Changeslow: A = [100, 120, 140 … , 460, 480, 500]
Changegreedy: A = [100000, 150000, 200000, … , 1100000, 11500000, 12000000]
Changedp: A = [2000, 2100, 2200, 2300, …, 3700, 3800, 3900]

Note: Different ranges were needed for each algorithm since at certain values of A changeslow would cause an overflow error and changeslow/changedp would become too slow(taking multiple minutes or more per calculation)







Comparison:

As we can tell by the completely different ranges of A that we had to use for the 3 different algorithms, changegreedy ran the fastest, followed by changedp, with changeslow bringing up the rear. The trend line for change greedy was linear, while changedp was quadratic, and changeslow was exponential.

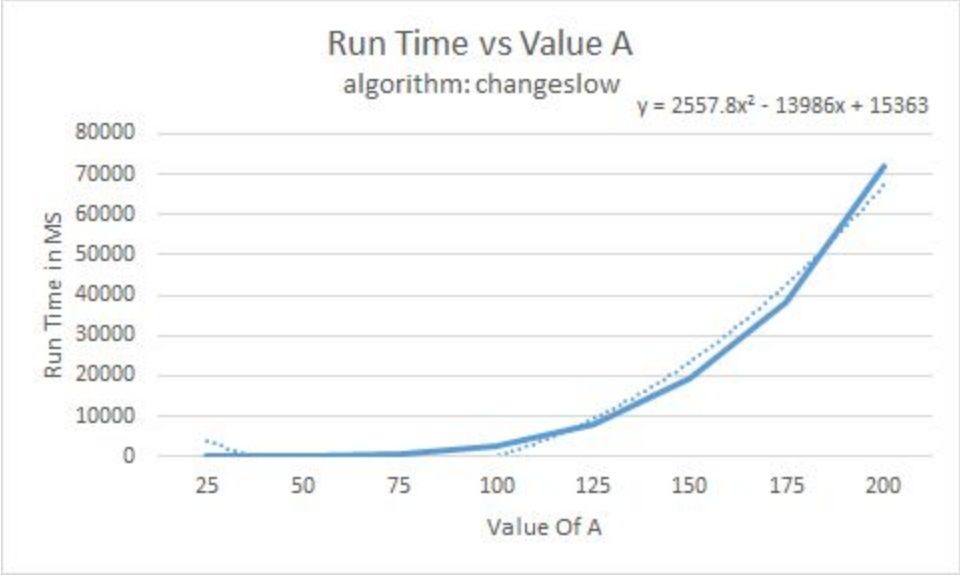
Scenario from Item #5:

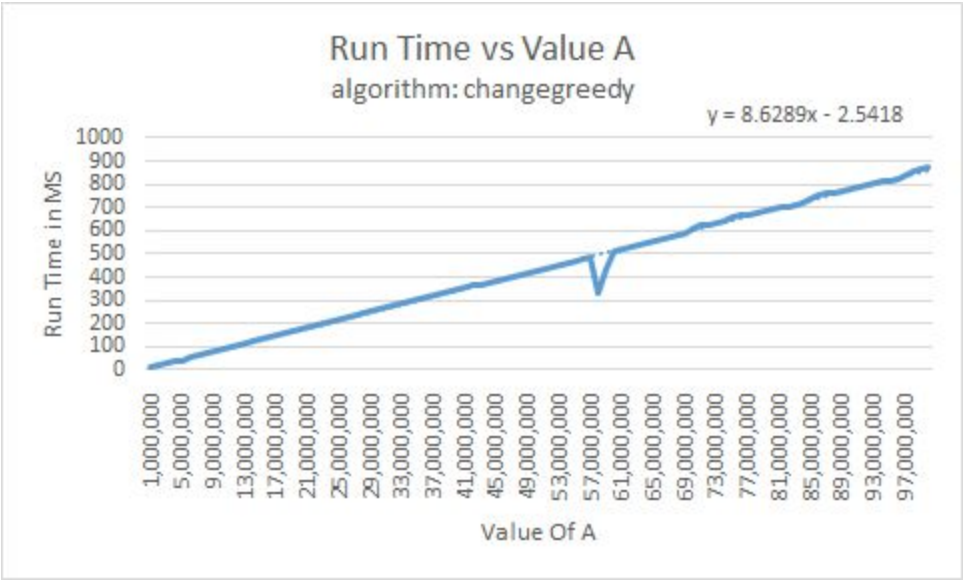
For the comparison of run times from the scenario in item #5, the following ranges of A were used for each algorithm:

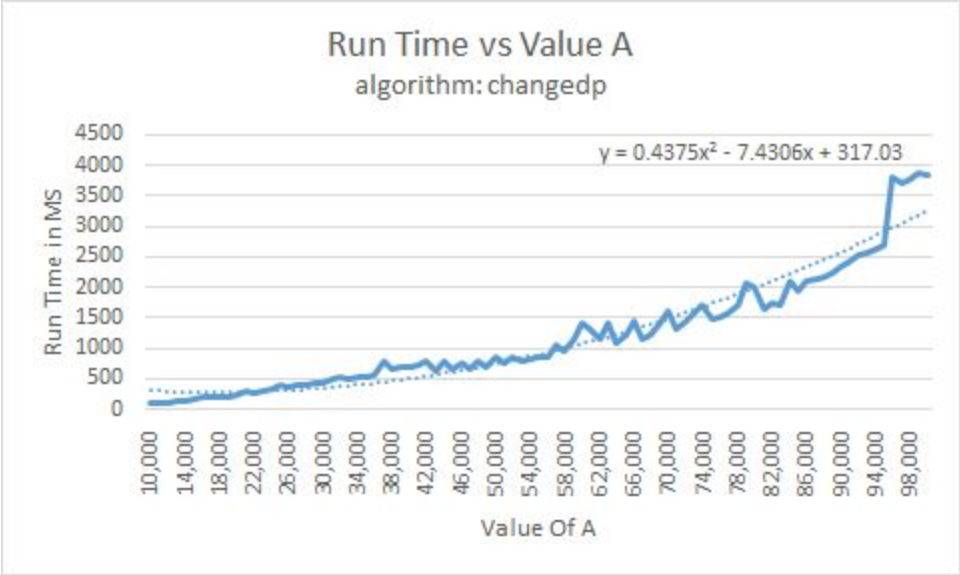
V_1 = [1,2,6,12,24,48,60]

Note: Different ranges were needed for each algorithm since at certain values of A changeslow would cause an overflow error and changeslow/changedp would become too slow(taking multiple minutes or more per calculation)

Changeslow: A = [100,125,150,...,150,175,200]
Changegreedy: A = [1000000, 2000000, 3000000, … , 98000000, 99000000, 100000000]
Changedp: A = [10000, 11000, 12000, … , 98000, 99000, 100000]



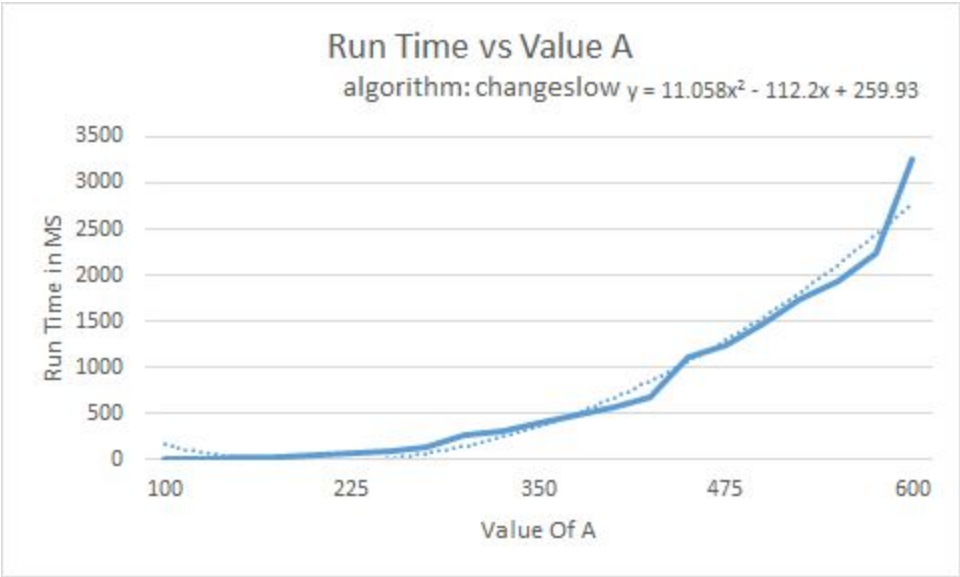


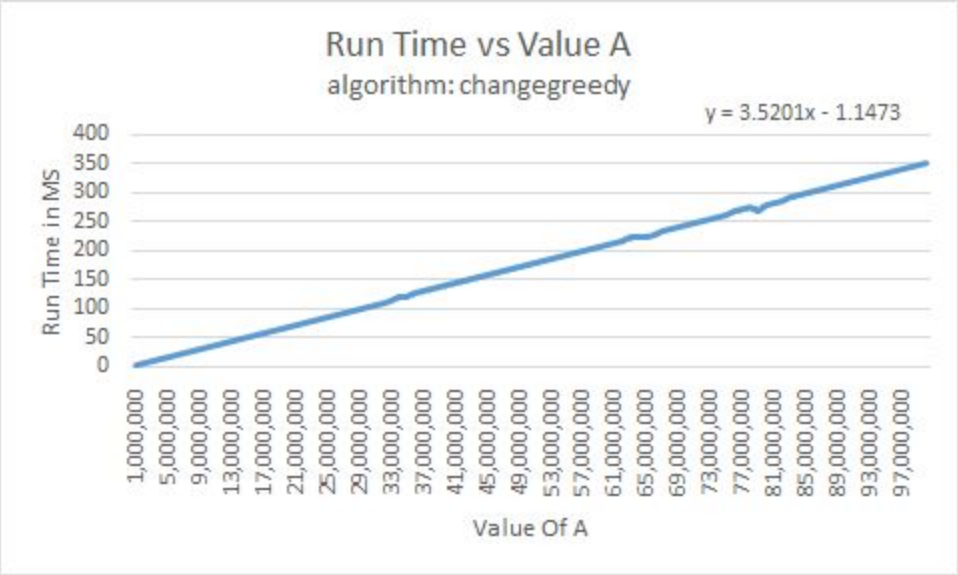


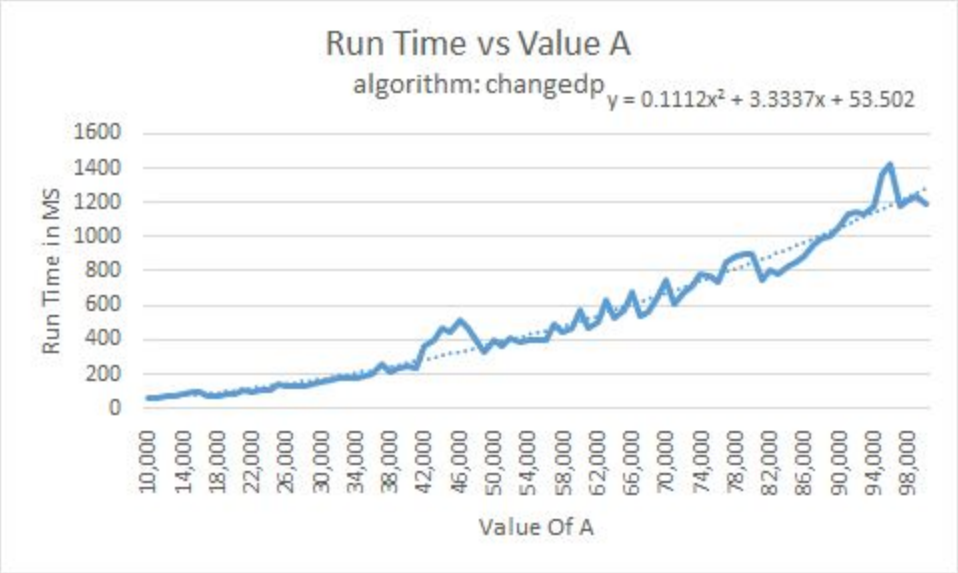
V_2 = [1,6,13,37,150]

Note: Different ranges were needed for each algorithm since at certain values of A changeslow would cause an overflow error and changeslow/changedp would become too slow(taking multiple minutes or more per calculation)

Changeslow: A = [100,125,150,...,550,575,600]
Changegreedy: A = [1000000, 2000000, 3000000, … , 98000000, 99000000, 100000000]
Changedp: A = [10000, 11000, 12000, … , 98000, 99000, 100000]







Comparison:

When comparing the algorithms changegreedy consistently ran the fastest. When looking at the regression lines changegreedy ran in linear time whereas changedp ran in quadratic time when the value of A got very large. Changeslow was alway quadratic and took a long time to run.

Scenario from Item #6:

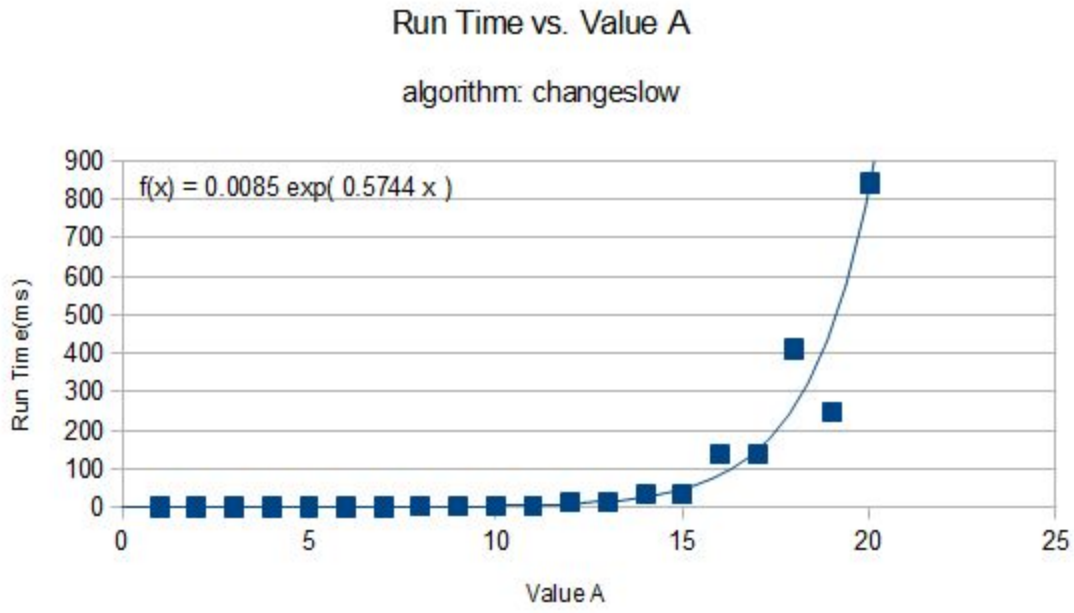
For the comparison of run times from the scenario in item #6, the following ranges of A were used for each algorithm:

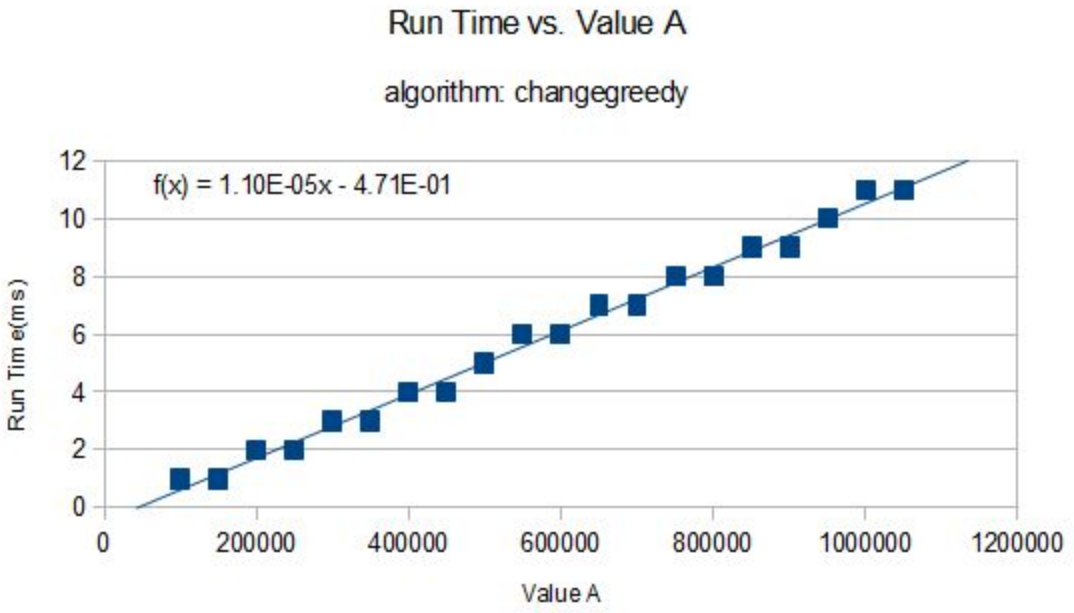
Changeslow: A = [1, 2, 3, 4, … , 18, 19, 20]

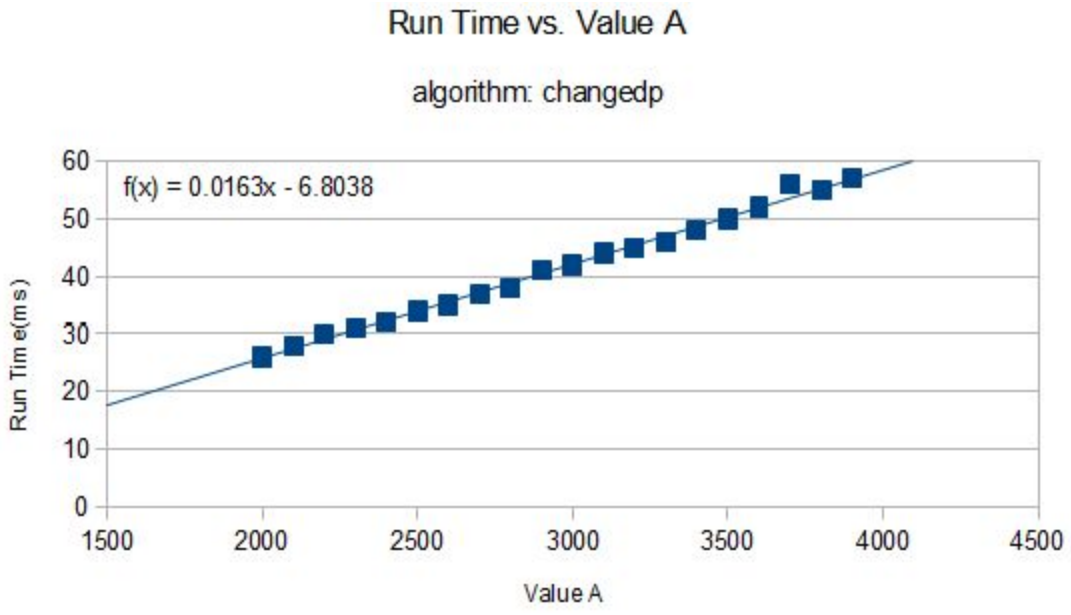
Changegreedy: A = [100000, 150000, 200000, 250000, … , 950000, 1000000, 1050000]

Changedp: A = [2000, 2100, 2200, 2300, …, 3700, 3800, 3900]

Note: Different ranges were needed for each algorigthm since at certain values of A changeslow would cause an overflow error and changeslow/changedp would become too slow(taking multiple minutes or more per calculation)







Comparison:

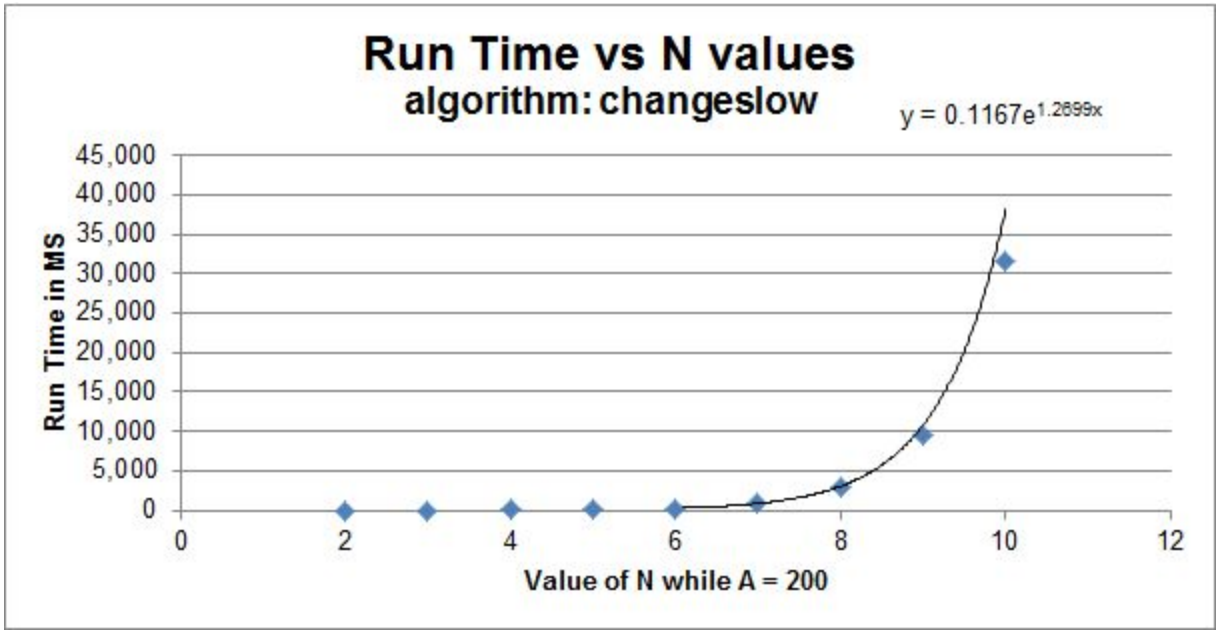
Changeslow ran in exponential time as expected. Once changeslow got past around A=15 it blew up very fast. So it would only be an algorithm to consider in the case that you are making change for 15 cents or less.

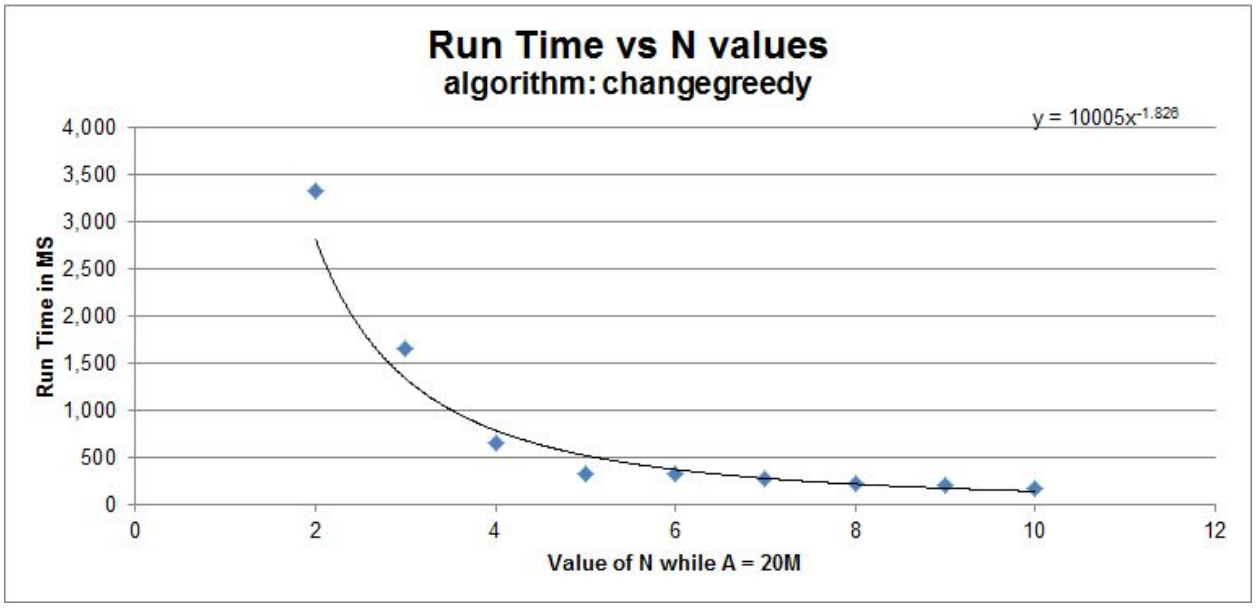
Both changegreedy and changedp ran in linear time as expected. However changegreedy was a much faster algorithm and didn't not run over 0ms in this situation until you started to make change over 100000 cents. However either would be a much better choise than changeslow but it looks like changegreedy is the optimal algorithm for the situation in item #6.

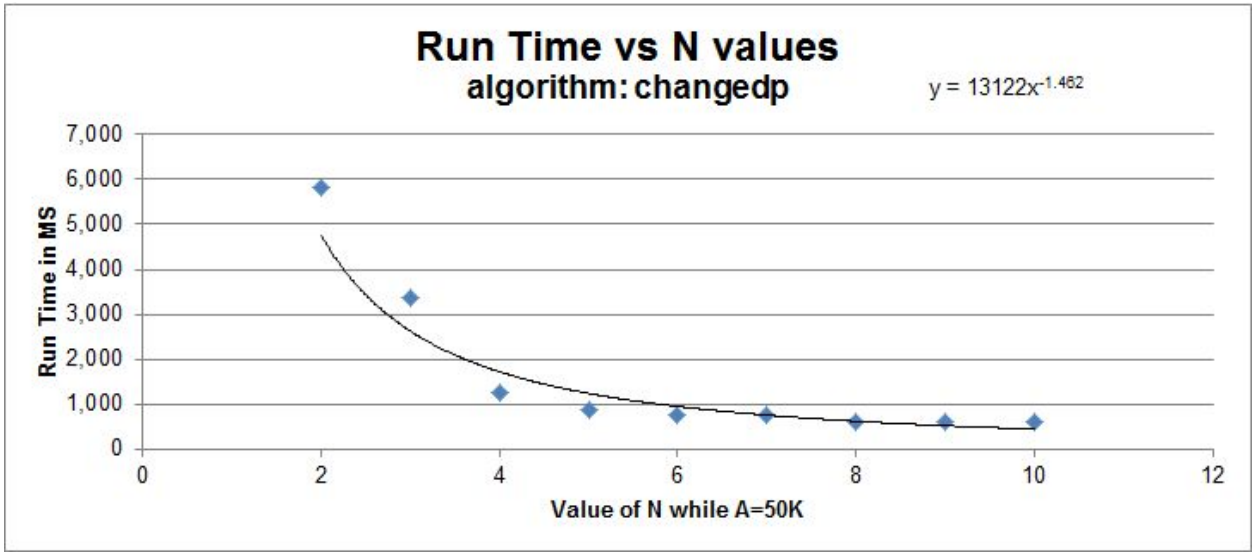
Item 8: Comparison of run times and number of denominations(for Items #4-6)

Scenario from Item #4:

For the comparison, we held A at the below constants, while changing N (the number of coins in the denomination array).
For changeslow A=200; for changegreedy A=20,000,000; for changedp A=50,000.
Every N beyond the original array, we increased the next N by 10.
So when N=5, A = [1, 5, 10, 25, 50] and when N=6, A =[1, 5, 10, 25, 50, 60].


Run Time vs N values
algorithm: changeslow
y = 0.1167e^1.2899x
45,000
40,000
35,000
30,000
25,000
20,000
15,000
10,000
5,000
0
Run Time in MS
0
2
4
6
8
10
12
Value of N while A = 200


Run Time vs N values
algorithm: changegreedy
y = 10005x^-1.826
4,000
3,500
3,000
2,500
2,000
1,500
1,000
500
0
Run Time in MS
0
2
4
6
8
10
12
Value of N while A = 20M

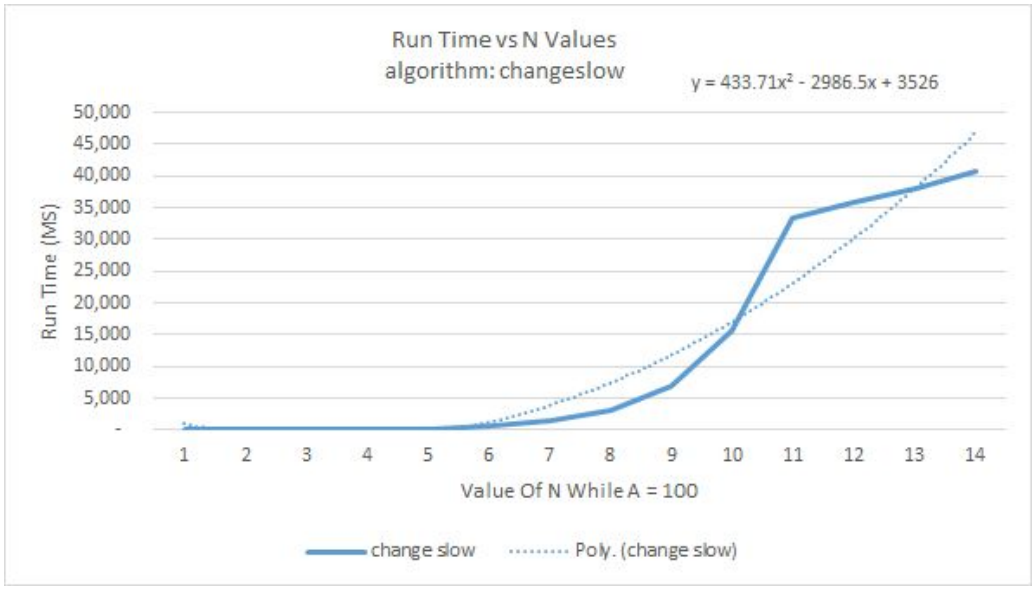


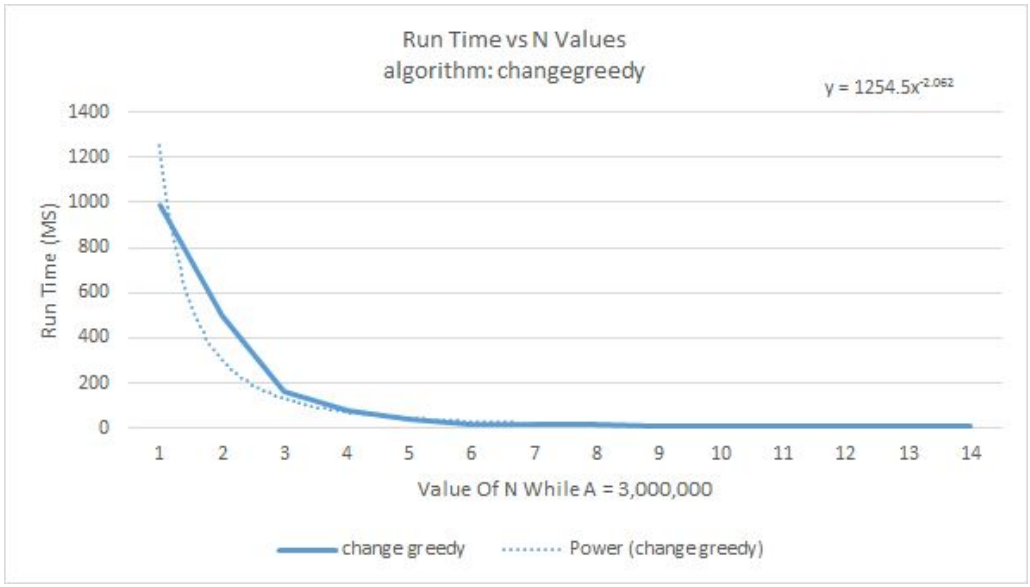
As we can see, changing the number of N values has a significant effect on all 3 algorithms. For changeslow, inceasing N increased the runtime exponentially. For both changedp and changegreedy, increasing N decreased the runtime polynomially.

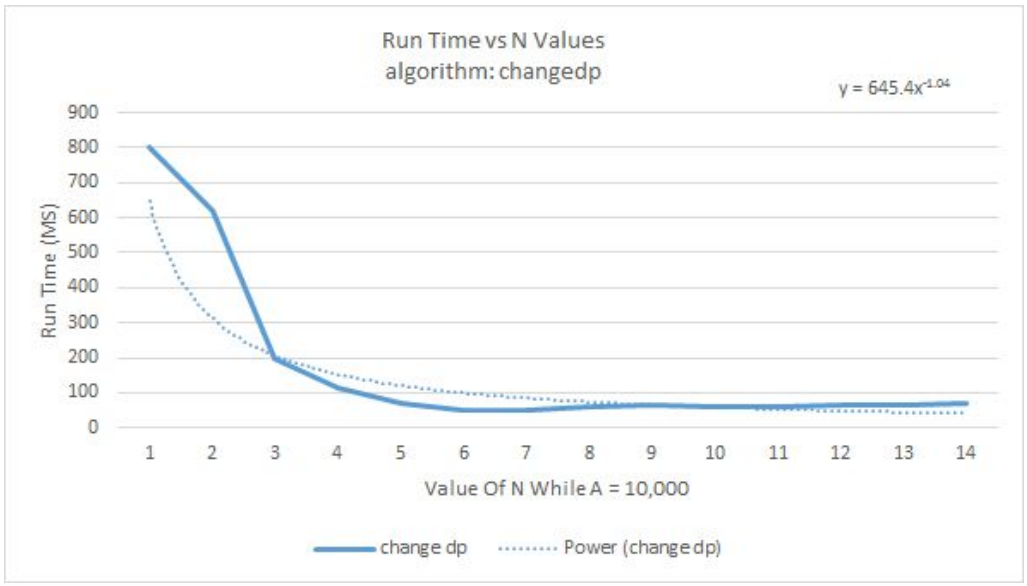
Scenario from Item #5:

For the comparison, we held A at the below constants, while changing N (the number of coins in the denomination array).
For changeslow A=100; for changegreedy A=3,000,000; for changedp A=10,000.
Every N beyond the original array, we increased the next N by 10.

V_1
So when N=7, A = [1, 2, 6, 12, 24, 48, 60] and when N=8, A =[1, 2, 6, 12, 24, 48, 60, 70,...].

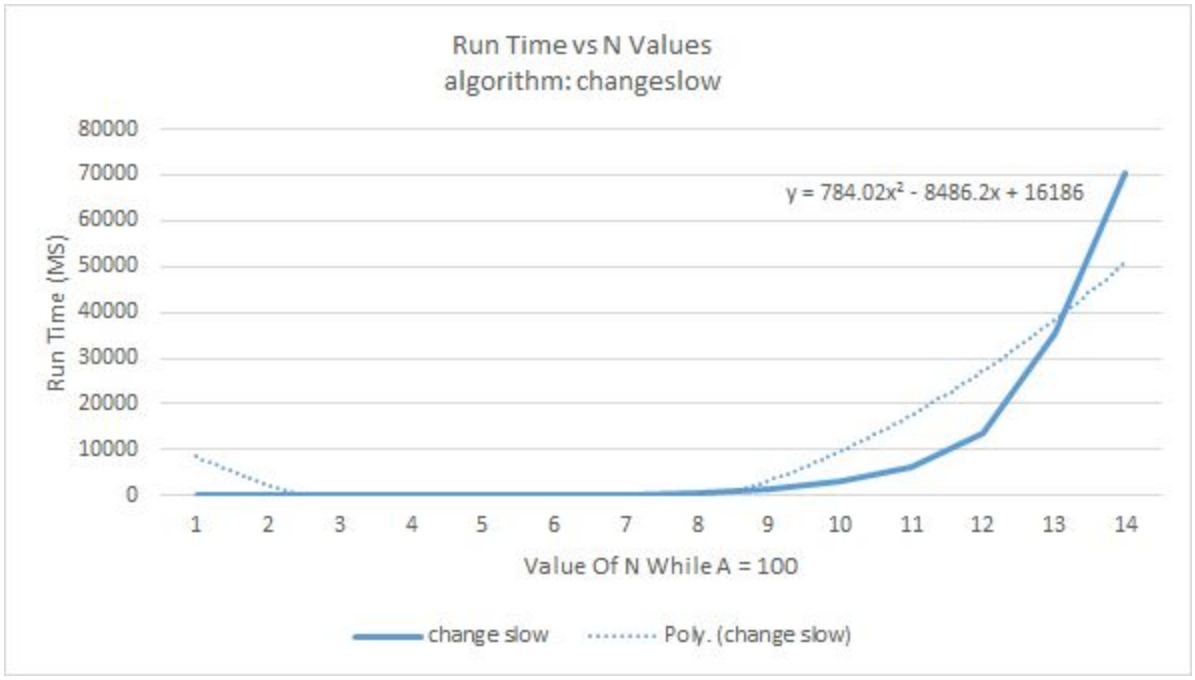


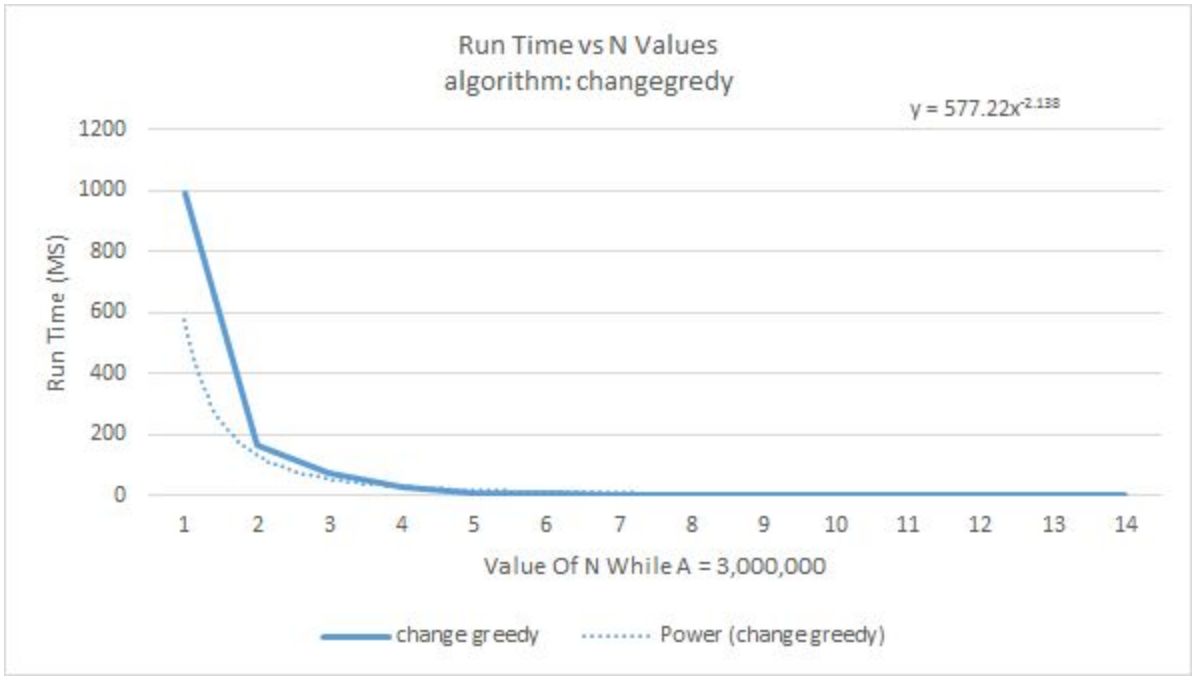


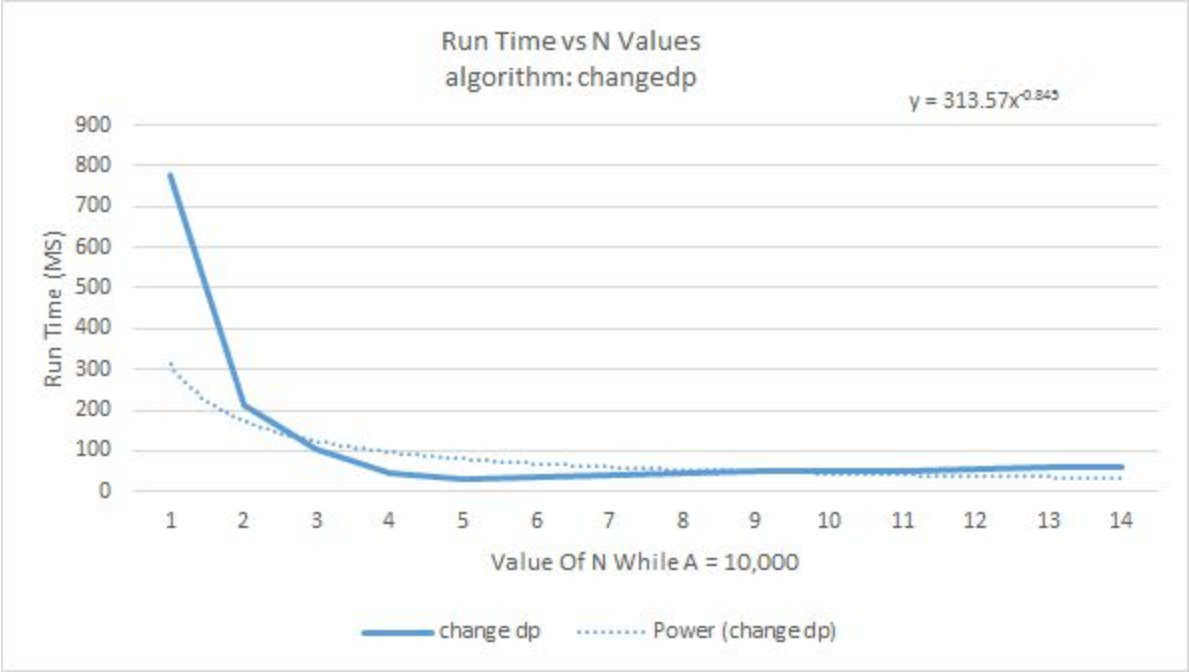


V_2

So when N=5, A = [1, 6, 13, 37, 150] and when N=6, A =[1, 6, 13, 37, 150,160,...].


Run Time vs N Values
algorithm: changeslow
y = 784.02x² - 8486.2x + 16186
Run Time (MS)
80000
70000
60000
50000
40000
30000
20000
10000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14
Value Of N While A = 100
change slow
Poly. (change slow)


Run Time vs N Values
algorithm: changegredy
y = 577.22x^-2.138
Run Time (MS)
1200
1000
800
600
400
200
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14
Value Of N While A = 3,000,000
change greedy
Power (change greedy)



Changing the size of N affected all of the algorithms, for the changegreedy and changedp it had the largest effect at the smaller N's. This was due to the fact that there had to be more smaller coins taken. Changeslow still had a quadratic run time but it was very efficient at the smaller Ns.

Scenario from Item #6:

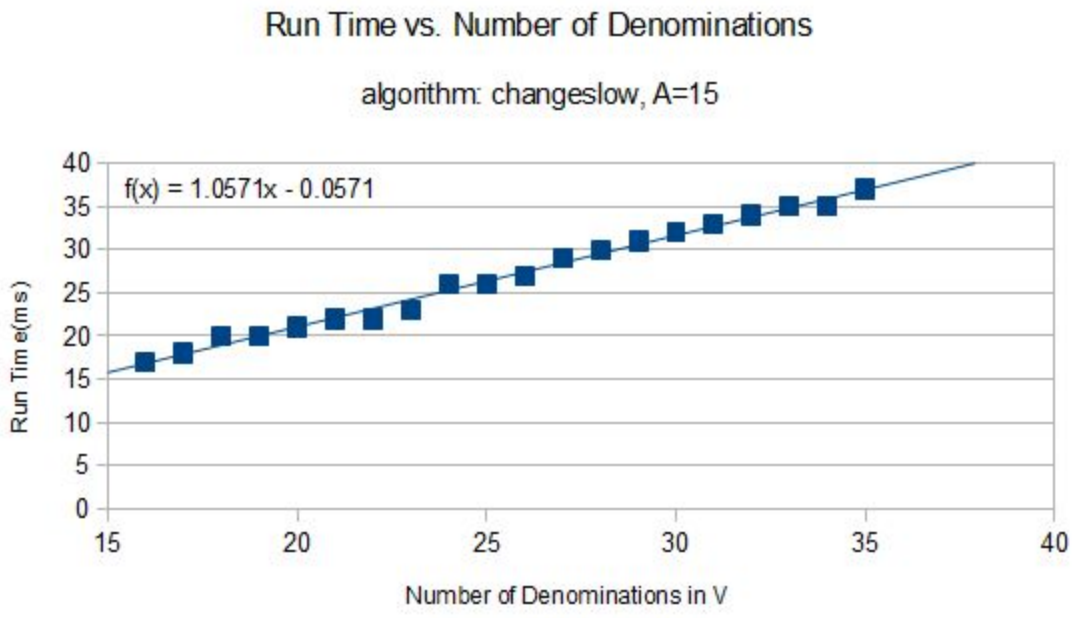
For the comparison, we held A at the below constants, while changing N (the number of coins in the denomination array).

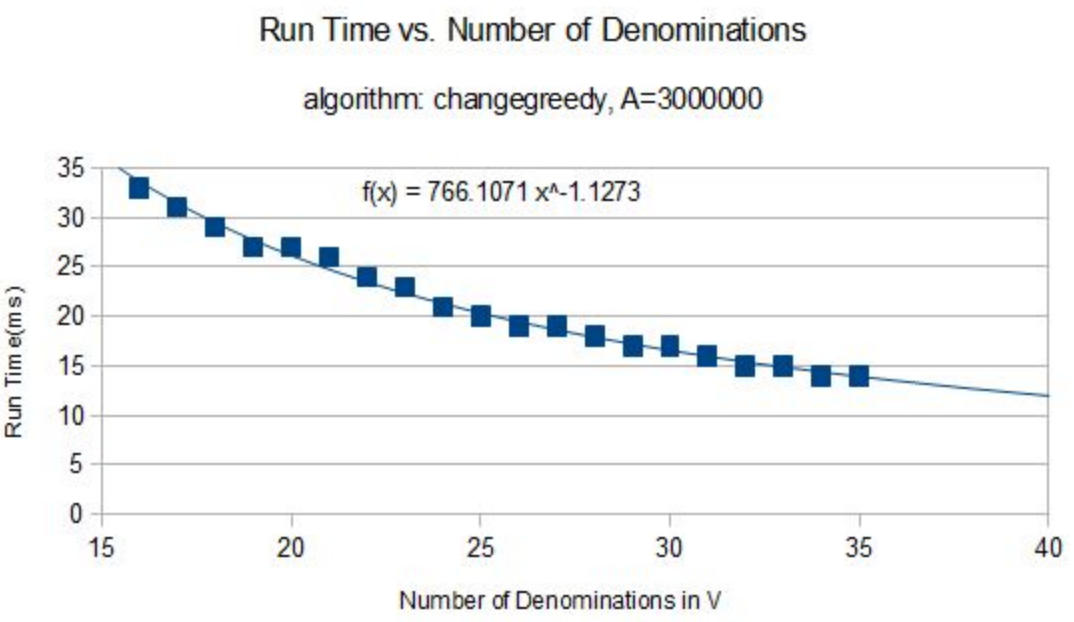
For changeslow A=15; for changegreedy A=3,000,000; for changedp A=6,000.

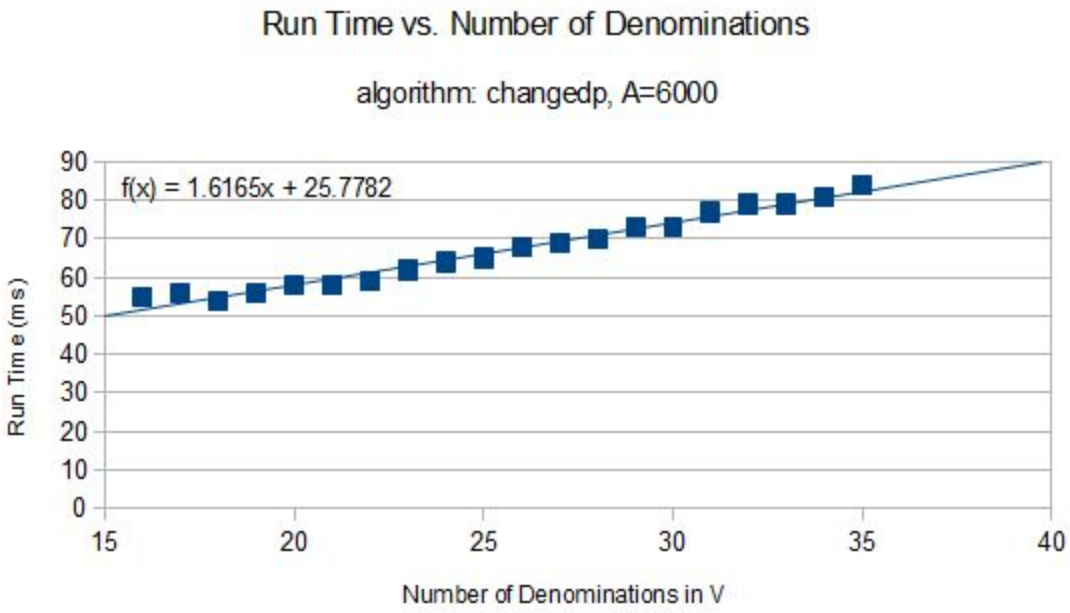
Every N beyond the original array, we increased the next N by 2 (which follows the pattern of the array V given in situation #6).

So when N=16, A =[1,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30] and when N=17, [1,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32], etc.

Here are the run time vs. number of denomination plots for all three algorithms.


Run Time vs. Number of Denominations
algorithm: changeslow, A=15
f(x) = 1.0571x - 0.0571
Run Time(ms)
Number of Denominations in V
0
5
10
15
20
25
30
35
40
15
20
25
30
35
40


Run Time vs. Number of Denominations
algorithm: changegreedy, A=3000000
f(x) = 766.1071 x^-1.1273
Run Time(ms)
Number of Denominations in V
0
5
10
15
20
25
30
35
15
20
25
30
35
40



From these three plots we can see that both changeslow and changedp algorithms will increase in running time as the number of denominations increases as the value A is held constants. This can be due to the fact that these two algorithms both look at all the values in the denominations array when they are running.

On the other hand the changegreedy algorithm decreases in running time as the number of denominations increases as the the value A is held constant. This happens because the greedy algorithm always takes as many of the highest valued coin it can. It this scenario the value of A is much higher than any coins in the denomination array, therefore any new higher value coin will require the algorithm to take less coins than the previous solution(thus given a quicker running time).

Item 9: Additional Comparison of changedp and changegreedy (A in powers of p)

To further look at the difference between the changedep and the changegreedy implementations, let consider the case where:

The coin denominations are powers of p, for example:

V = [1, 3, 9, 27]

Testing for small values of A=1 to 30, both had the same solutions but with 0 ms running time.

Testing for larger values of A=10001 to 10030, both once again had the same solutions but the running times were different. “Changegreedy” performed the calculations once again with 0 ms running time reported. On the other hand, “changedp” performed each calculation with a running time of around 57ms to 80ms. This shows that “changedp” would be less efficient in this particular situation than “changegreedy”.

The greedy implementation is the optimal solution in this case.

Item 10: Conditions for greedy algorithm to be optimal solution
Let's looks at a situation where the greedy algorithm is not optimal:

Situation #1:
Let the denominations be: V = [1, 4, 5]
And say we want to find the make change for A = 8

Greedy: Would first choose one "5" coin, then, three "1" coins for **4 coins total**
However, the optimal solution is: two "4" coins for **2 coins total**

Situation #2:
For this situation, let's take the same V from the first situation but increase the biggest coin by 1:
So that the denominations are: V = [1, 4, 6]
And say we want to find the make change for A = 8 again

Greedy: Would first choose one "6" coin, then, two "1" coins for **3 coins total**
However, the optimal solution is: two "4" coins for **2 coins total**
Our solution is closer this time, but is still not optimal

Situation #3:
For this situation, let's take the same V from the second situation but increase the biggest coin by 1:
So that the denominations are: V = [1, 4, 7]
And say we want to find the make change for A = 8 again

Greedy: Would first choose one "7" coin, then, one "1" coin for **2 coins total**
This matches the optimal solution is: two "4" coins for **2 coins total (either 7+1 or 4+4)**

Analysis:
Now we will look closer at the coin denominations 2 and 3 in each of the cases.

For the failed cases:
Situation 1 has a difference of 6 - 4 = 2 between the 2nd and 3rd denomination
Situation 2 has a difference of 7 - 4 = 3 between the 2nd and 3rd denomination

For the successful case:
Situation 3 has a difference of 8 - 4 = 4 between the 2nd and 3rd denomination
This threshold occurs when the 3rd coin is equal to twice the second coin, or:

Coin[i+1] = 2 * Coin[i]

Additionally we can also see that this will hold for the case when the coin is greater than twice the coin before it too.
So more generally we can say that:

Coin[i+1] >= 2 * Coin[i]

Concluding, that the greedy algorithm will be the optimal solution when each coin in the denomination list is equal to or greater than twice the coin that comes immediately before it.