CS325 Project 2 - Coins

Group 3: Ian Henry, Dennis Tat, Jaden Yuros

Q1: Pseudocode

Brute Force Pseudocode

```
changeslow(coinList[n...m], change)

    changeHelper(parameters)
        If (runningTotal == change)          // this is a valid solution
            let formattedSolution = storeResult(runningTotal, coinList)
            add formattedSolution to listOfSolutions
        If (runningTotal > change)           // this is not a valid solution
            break
        Else, For all coin denominations in coinList
            If the coin denomination is >= to the last coin added to runningTotal
                Copy the runningTotal to a new list, newTotal
                Add the current coin denomination to newTotal
                Recursively call changeHelper with newTotal

    storeResult(validSolution, coinList)
        let coinTally[ x ] = list of length x, where x is the # of different coin denominations
        For all coin denominations in coinList
            let count =  the quantity of times that coin is in the valid solution
            Store count in coinTally[ i ], , where i is the current denomination

    changeHelper(parameters)            // first call, will recursively find all valid solutions
                                        // storing the results in listOfSolutions
    let listOfSolutions [ ] = list where all valid solutions are stored
    let fewestCoins =  -infinity
    let index = -1
    For all solutions in listOfSolutions
        If ( quantity of coins < fewestCoins)
            fewestCoins = quantity
            Index = index of current solution
    Return (fewestCoins, listofSolutions[index])
```

Brute Force Asymptotic Analysis

The brute force algorithm is started by a call to the *changeHelper* function. This function will recursively find all valid combinations of coins that add up to the desired amount. Let $A = \textit{the amount of change desired}$, and $c_1, c_2, c_3, \ldots, c_n = \textit{n unique coin denominations}$. This function will try all combinations of $x, y, \ldots, z, \textit{where } x, y, \ldots z \textit{ are some constant} \geq 0$, such that $xc_1 + yc_2 + \cdots + zc_n \leq A < xc_1 + yc_2 + \cdots + zc_n + c_n$.
Let $m = x * y * \cdots * z, \textit{ and is some constant } \geq 0$. This results in $x^n y^n z^n \textit{ or } m^{n+d}$ total combinations, $\textit{where d is some constant } > 0$. Each combination step is performed in $\Theta(1)$ time. The *changeHelper* function will run in $\Theta(m^{n+d}) + \Theta(1) = \Theta(2^n)$ time

The *storeResult* function will run in $\Theta(n)$ since it needs to check each coin denomination, and will run each time there is a valid combination, or *V times.* The total runtime for this function is $\Theta(Vn) = \Theta(n)$.

Once all solutions have been found, it will choose the best solution by iterating over all solutions, *V*, and choose the optimal solution. This is accomplished in linear $\Theta(n)$ time.
The entire result of the brute force approach is $\Theta(2^n) + \Theta(n) + \Theta(n) = \Theta(2^n)$.

Greedy Pseudocode

```
changegreedy(list[n…m], change)
    let C[ ] = size of list
    let runningTotal = 0
    let m = 0

    while runningTotal < change
        start at the end of list
            if the denomination <= change
                add the denomination to running total
                increment the denomination in C
                increment m
            else move to the next highest denomination
    return C[ ], m
```

Greedy Asymptotic Analysis

This greedy algorithm is naive, which means it may not produce the optimal solution every time. In the greedy algorithm, a while loop keeps track of the running total and compares it with the amount of change to be made.

In the while loop, the count of the denominations and total number of coins are incremented if a certain denomination can be added to the running total. These increments are both done in constant time.

The while loop run time is dependent on how large the change amount is. It also depends on the specific denominations available, but in general, this algorithm will loop more times if the change amount is larger. So we have a complexity of the greedy algorithm is Θ(n) * Θ(1) = Θ(n), where n is the amount of change to be made.

Dynamic Pseudocode

```
changedp(list[n…m], change)
    let C[ ] = size of list
    let m = 0
    let values[ ] = size of change, initialize with infinity
    let denominations[ ] = size of change, initialize with -1

    for each denomination in list
        for each value in values
            if the value <= the denomination
                values[value] = min(values[value], values[value - denomination] + 1)
                if values[value - denomination] + 1 is min
                    denominations[value] = denomination

    m = values[length]
    for each m
        fill C with the quantity of each denomination by referencing denominations[change]
            decrement by each denomination
    return C[ ], m
```

Dynamic Asymptotic Analysis

The dynamic function finds the minimum quantity of coins required to make n amount of change by building two arrays of length n + 1, to account for a 0 index representing value 0. In order to fill the first array (representing [0, 1, 2 … n] values of change), the algorithm iterates k amount of times of the array of size n, where k represents the number of denominations, and so executes in k* n, where each iteration requires a constant time time operation of comparing two values to find the minimum, and depending on the outcome, a constant time operation of replacing the value of the first array at the index with the new minimum and the value of the second array at the index with the index value of the current denomination (k at the current iteration). . To build the return array it requires a loop running k amount of times and performing a constant time operation. As n becomes larger, this loop would become insignificant and therefore the algorithm’s running time is Θ(kn).

Q2: changedp

The changedp array indexed at [0, 1, 2 … n] represents the quantity of coins required to make n_i change. This array is filled by iterating k amount of times over the array, where k represents the number of denominations. Each iteration consists of comparing the current contents of the array at n_i to determine if it can be replaced by the value of the array at n_i - (the index - the denomination) + 1. This is a valid way to fill the table because it ensures that the table at n_k for each value of k is filled, and thus there is access to the smallest elements to combine for any value of change possible given the denominations. Since the denominations are ordered from smallest to largest, it ensures that the most basic (often the most coins required) solutions are complete as you move up the denominations, and so if a higher denomination can reduce the number of coins required for a given solution, the coins can easily be calculated by referencing previous elements in the array less the denomination, otherwise, no changes are made to the array. Since an optimal solution requires computation of each combination of coins to find the minimum, this does so in an efficient manner by storing old calculations which can be easily accessed as an array element - as opposed to access through recursive calculations.

Q3: changedp Proof by Induction

Prove $T[v] = min_{V[i] \le v}\{T[v - V[i]] + 1\}$, $T[0] = 0$

Base Step:
Assume $T[v]$ *is the optimal number of coins needed to make change for value,* v*, and where* $v > 0$*. Let* V *be a set of coin denominations that are needed to make change for* v*. There must be some first coin,* $V[i]$ $where$ $V[i] \le v$ *which creates the optimal solution,* $T[v]$*.*

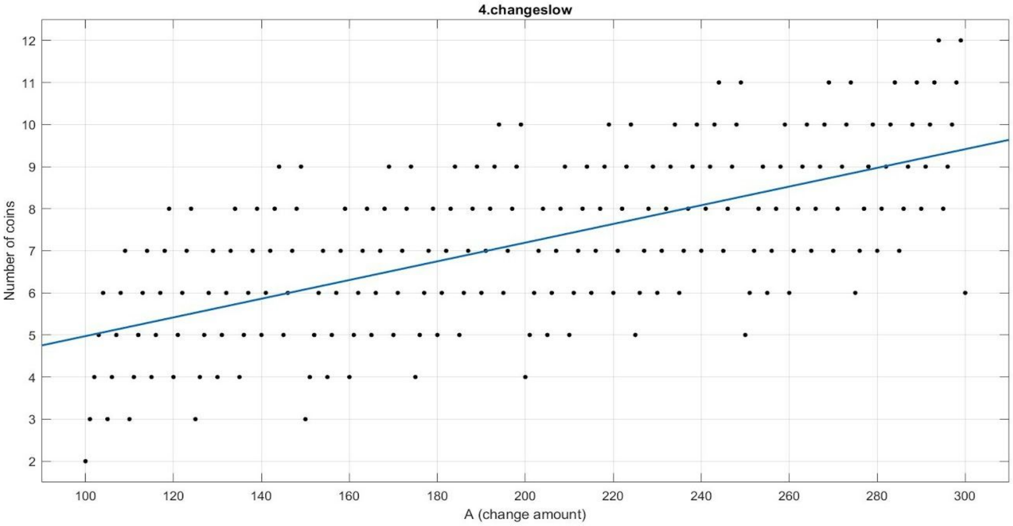
Inductive Step:
Since $V[i]$ *is part of the optimal solution, then* $T[v - V[i]]$ *must also be made up of coins composing the optimal solution. Since* $V[i]$ *is a single coin, then* $T[v - V[i]]$ *contains one fewer coin than* $T[v]$*, that is to say* $T[v] = T[v - V[i]] + 1$

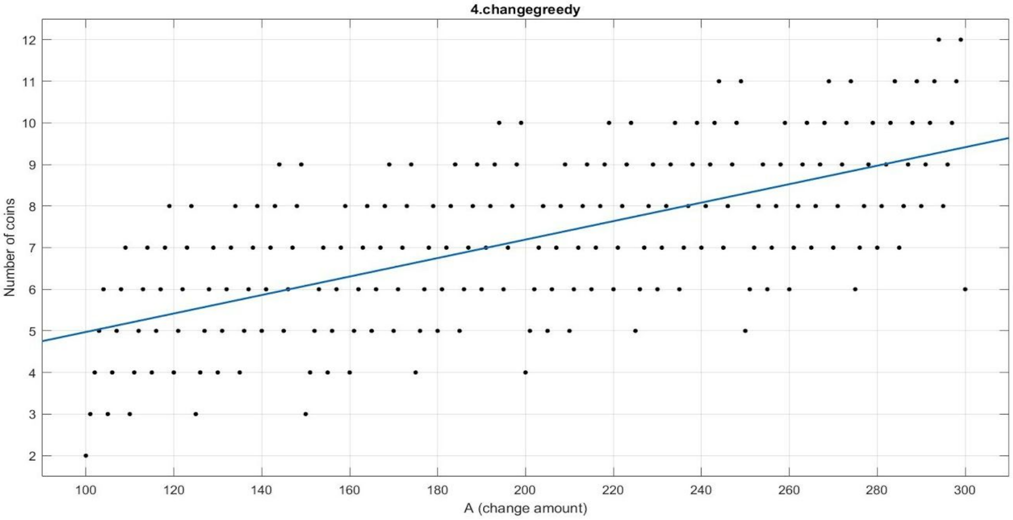
In the case we need to make change for $v = 0$, we clearly will need zero coins. If we use one coin with a denomination > 0, that would result in $v > 0$ and $T[v] > 0$, both of which are contradictions. Therefore $T[0] = 0$

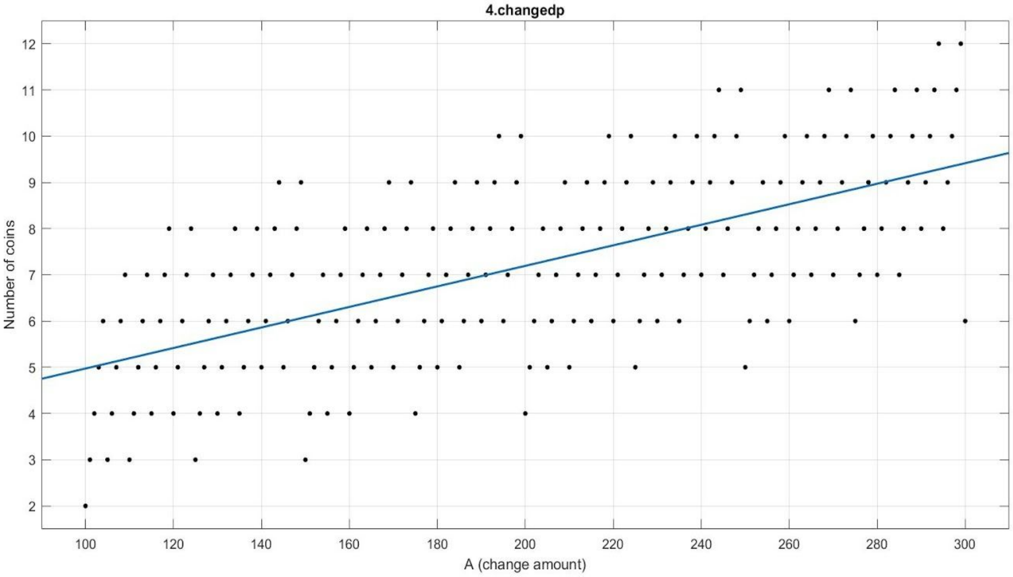
Conclusion:
From the base and inductive steps, we can conclude that $T[v] = min_{V[i] \le v}\{T[v - V[i]] + 1\}$ *and* $T[0] = 0$

For the experimental analysis, we decided to plot the algorithms on the same scale (except for the cases noted) in each scenario to easily compare them. We tested a lower range of values of A so that the changeslow algorithm could run a in a reasonable amount of time. All testing was completed on Dennis' computer.

Q4: V = [1, 5, 10, 25, 50]

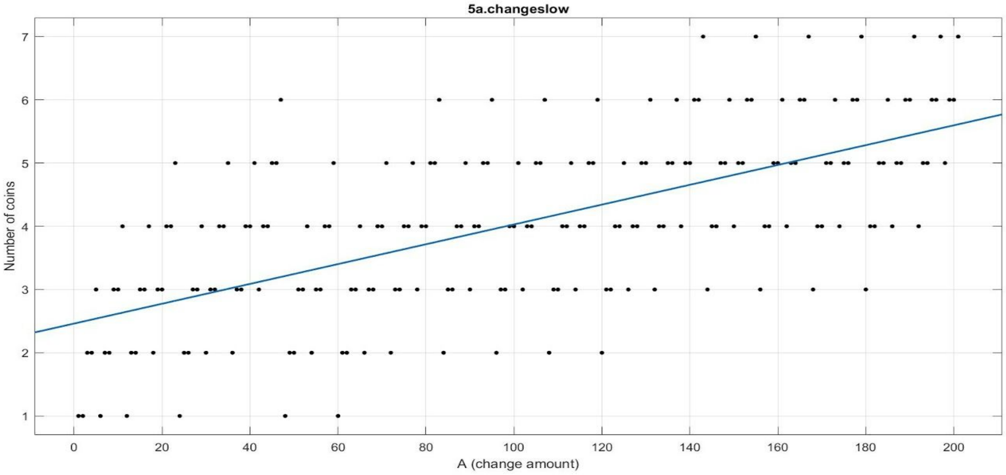


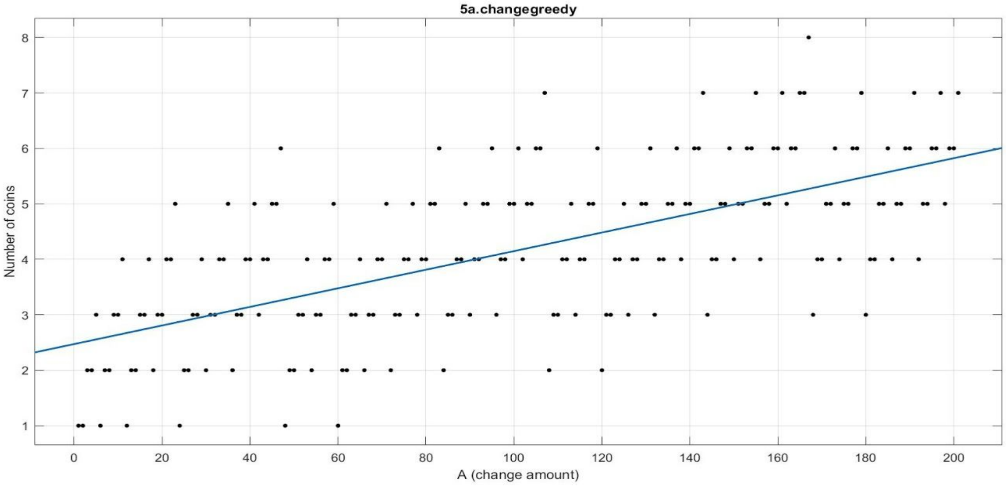


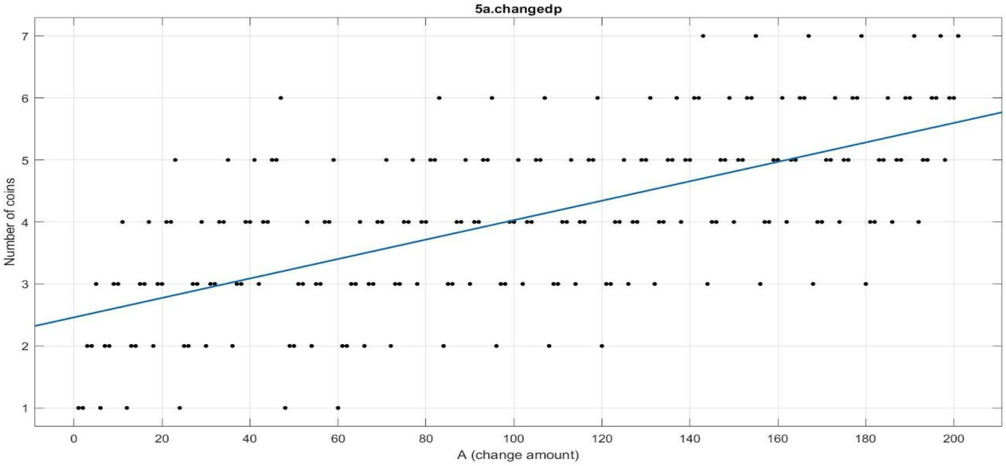


With this set of coin denominations, all three algorithms came up with the same results when comparing the number of coins.

Q5: $V_1 = [1, 2, 6, 12, 24, 48, 60]$

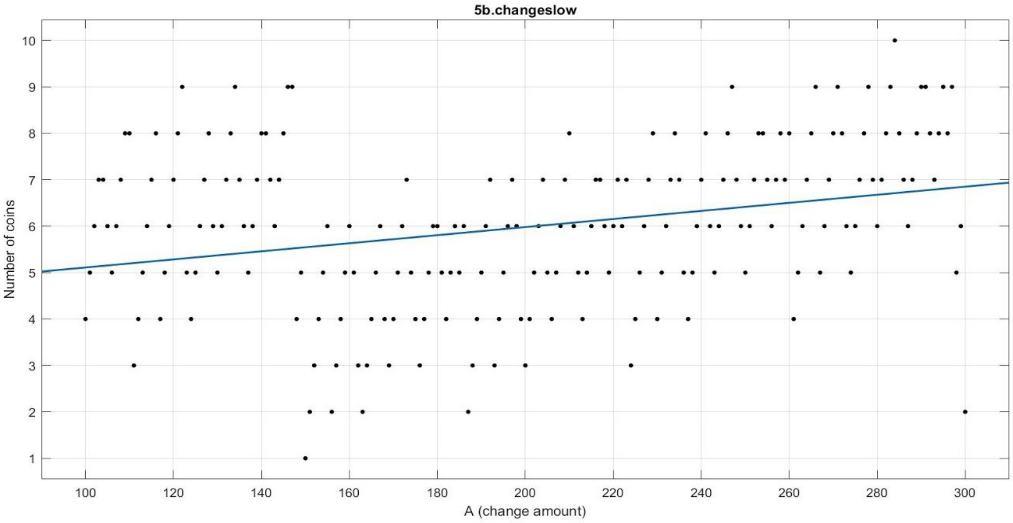


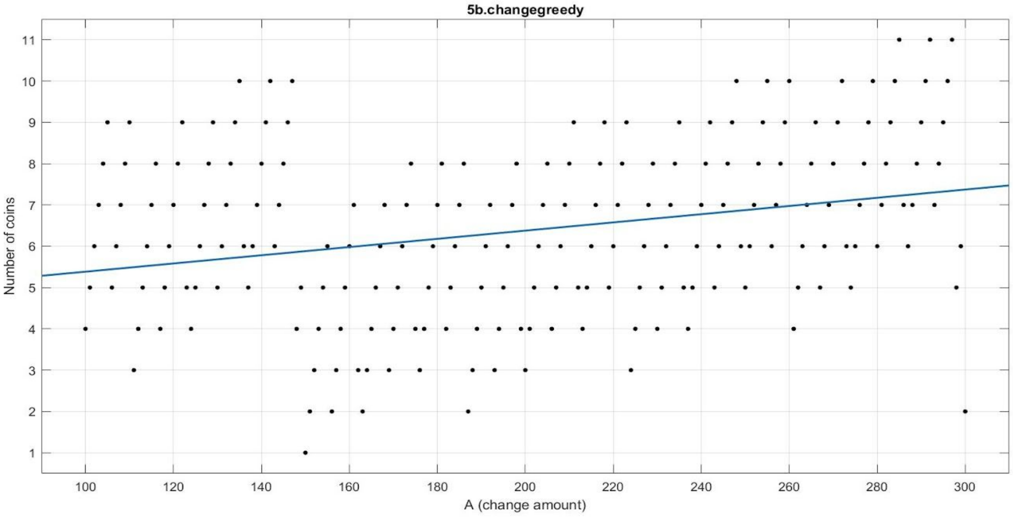


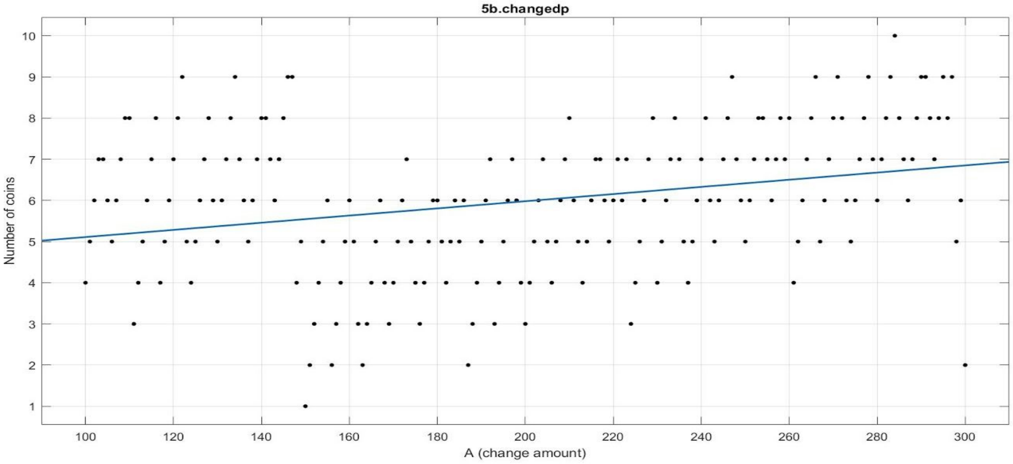


With this set of coin denominations, we lowered the range of values for A so that changeslow could run in a reasonable amount of time. Here, we see that changeslow and changedp came up with the same results. The changegreedy algorithm had a few instances of being less optimal by having an additional coin, but this was expected because it is a naive algorithm.

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

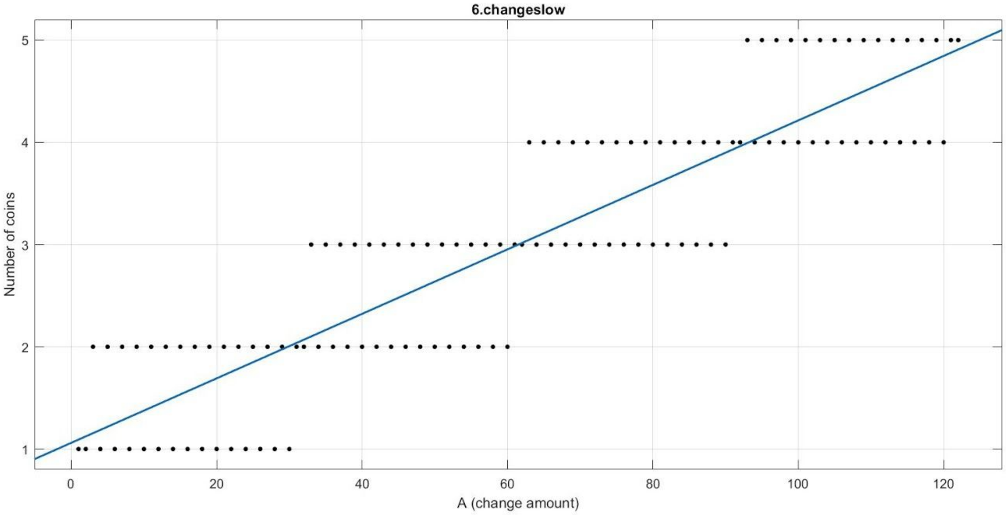


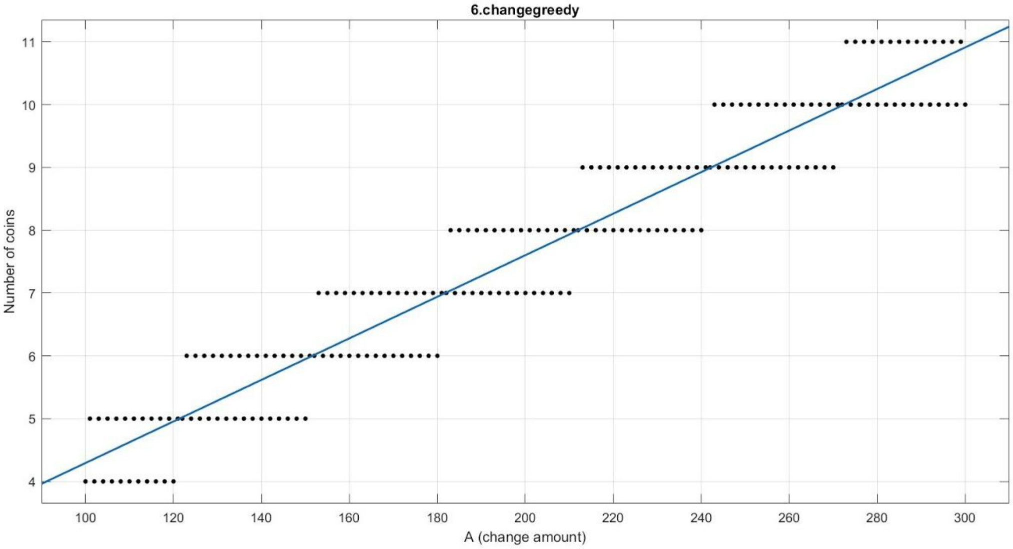


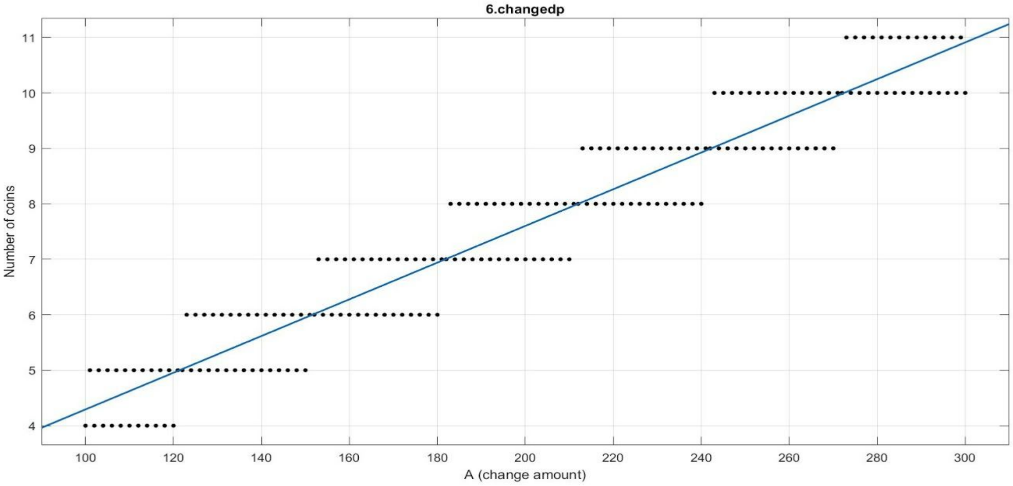


Again, we see here that changeslow and changedp had the same results, while changegreedy had more instances of needing an additional coin to make the amount of change.

Q6: V = [1, 2, 4, 6, 8, 10, 12, …, 30]



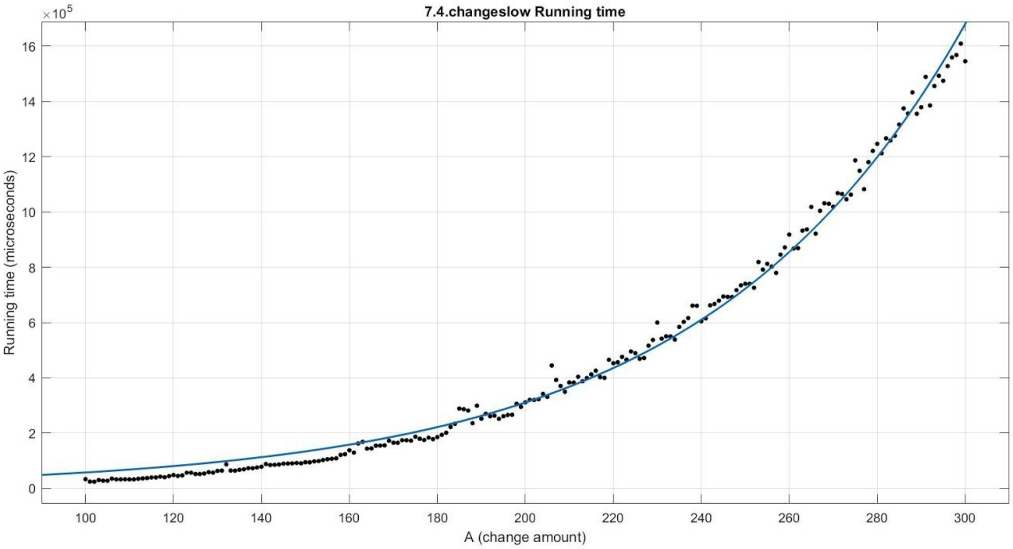


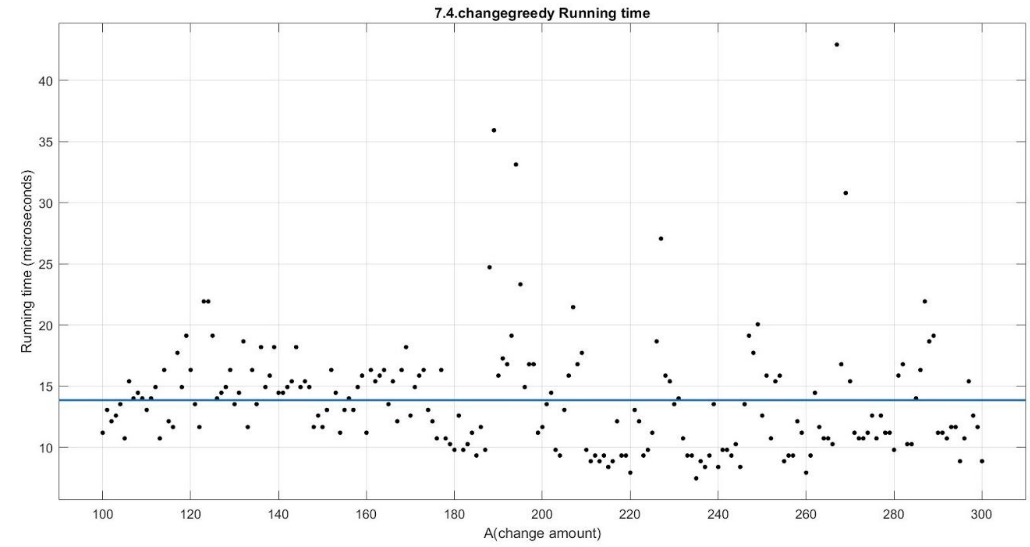


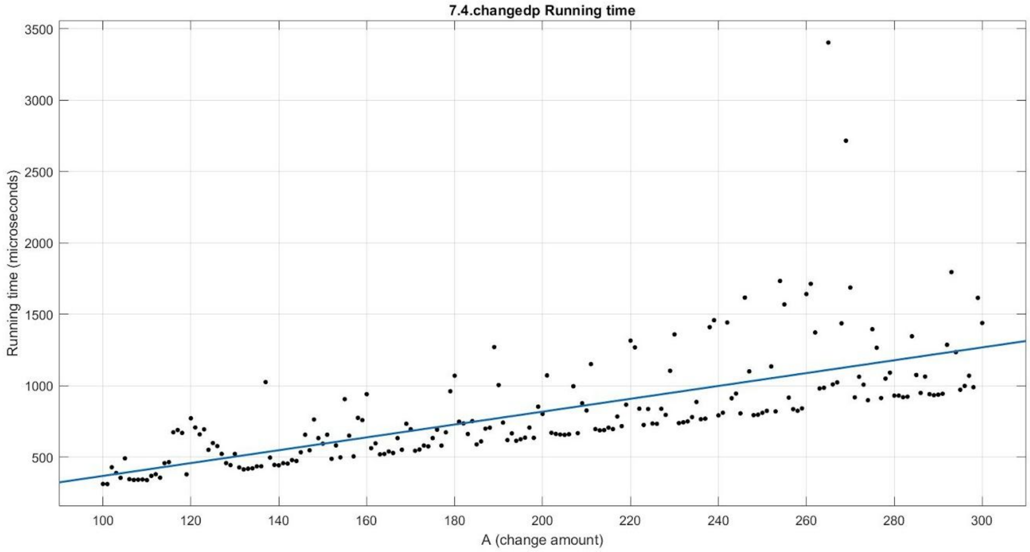
For this set of denominations, we tested a different range for changeslow so that it could run in a reasonable amount of time. We see that changegreedy and changedp came up with similar results. Theoretically, if changeslow was tested on the same range, we would expect the same results as the other algorithms, since it produces the optimal solution.

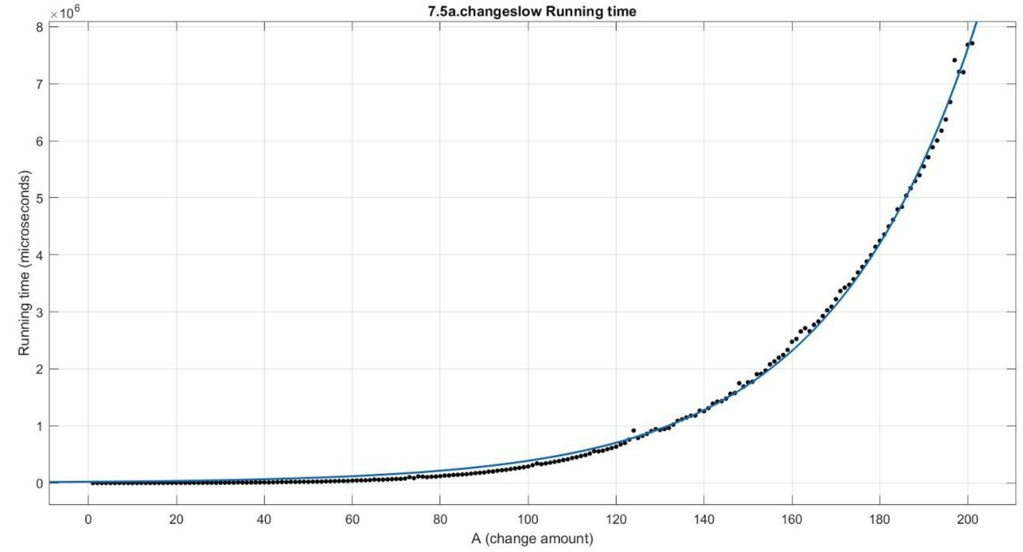
Q7: Running Time as a Function of A

For the following plot titles, 7.x corresponds to a previous questions/scenario in this report.

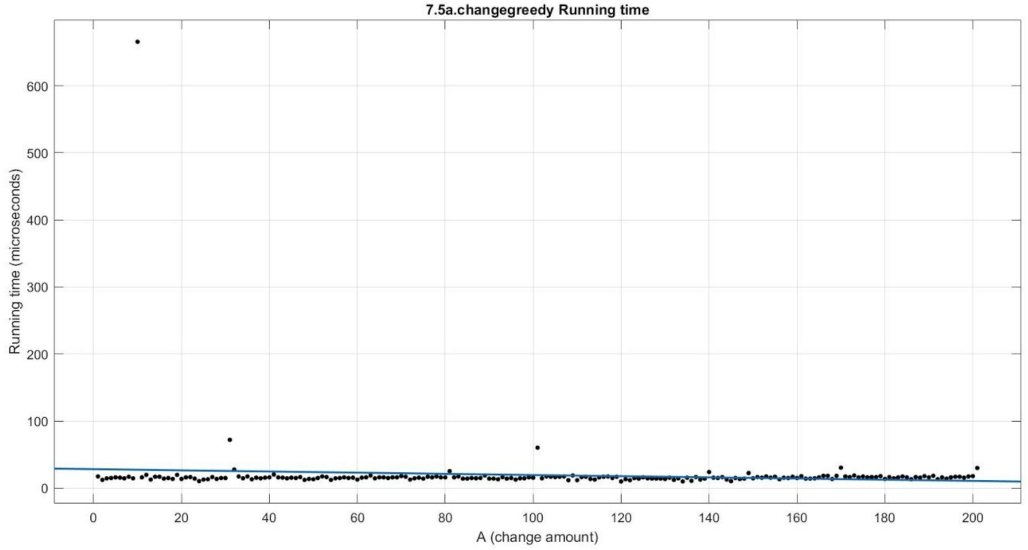



7.4.changegreedy Running time
Running time (microseconds)
40
35
30
25
20
15
10
100
120
140
160
180
200
220
240
260
280
300
A(change amount)

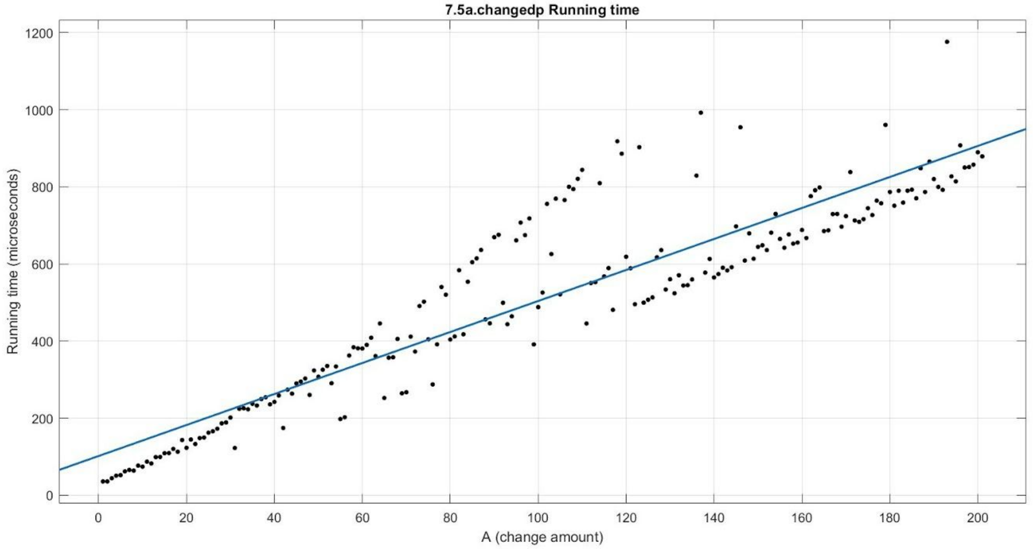

7.4.changedp Running time
Running time (microseconds)
3500
3000
2500
2000
1500
1000
500
100
120
140
160
180
200
220
240
260
280
300
A (change amount)



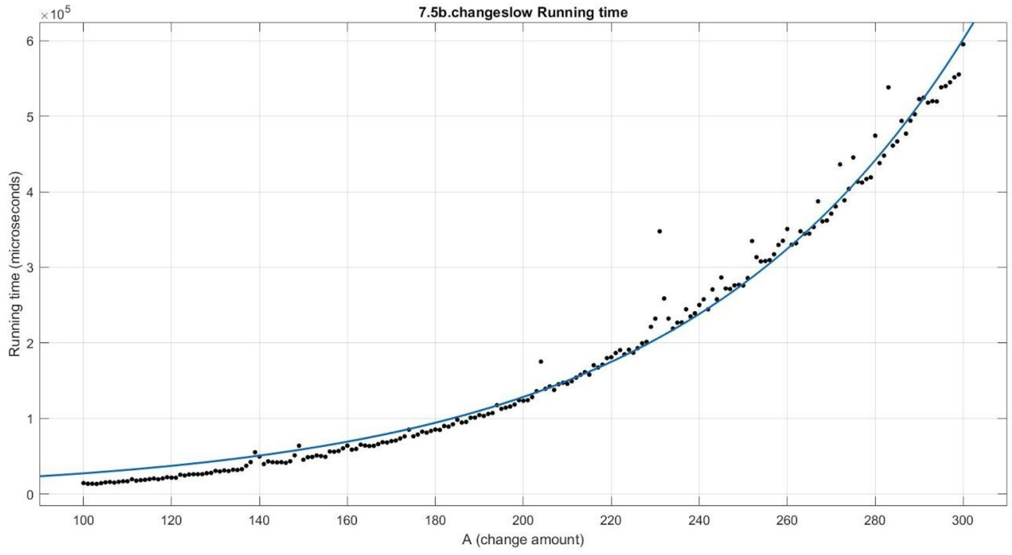
7.5a.changeslow Running time

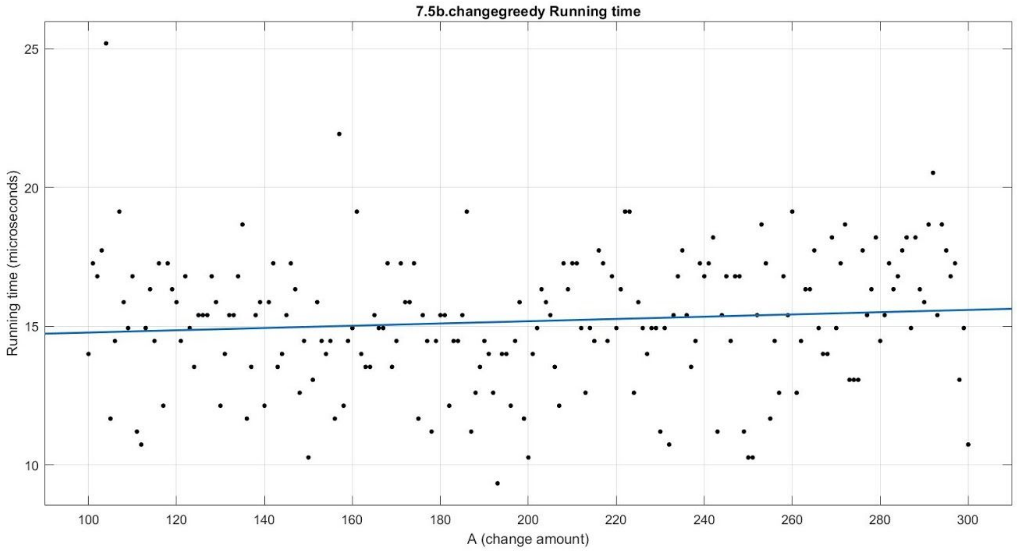


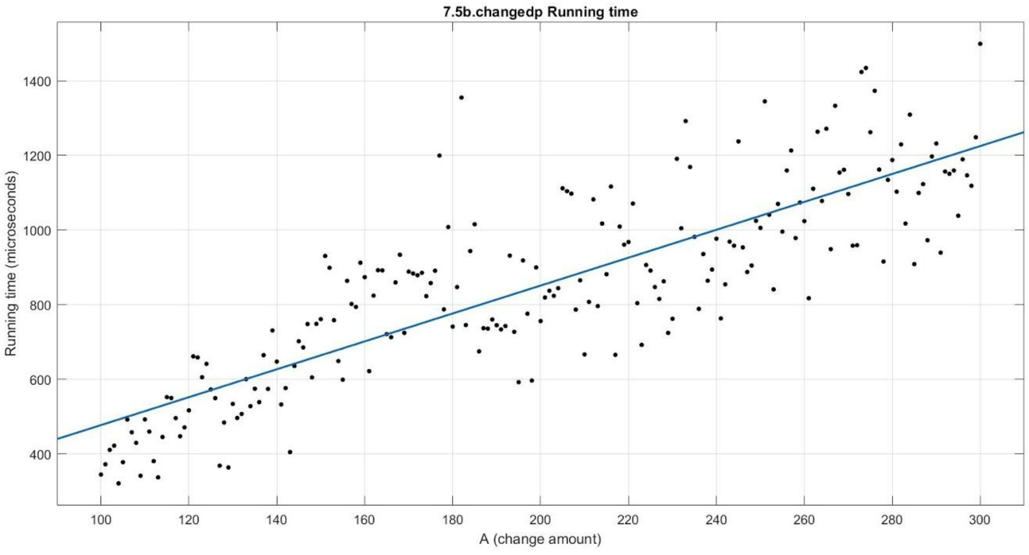
7.5a.changegreedy Running time

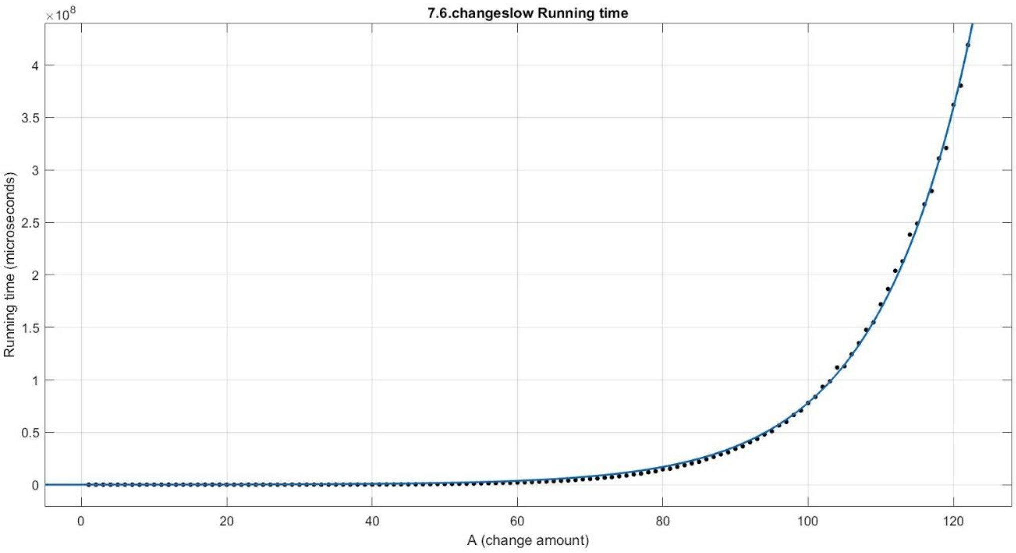


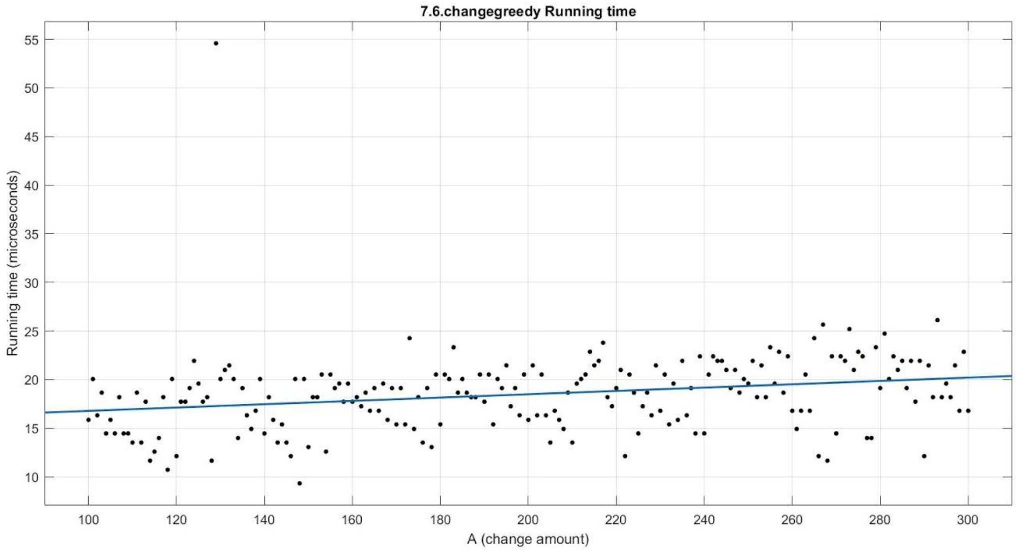
7.5a.changedp Running time

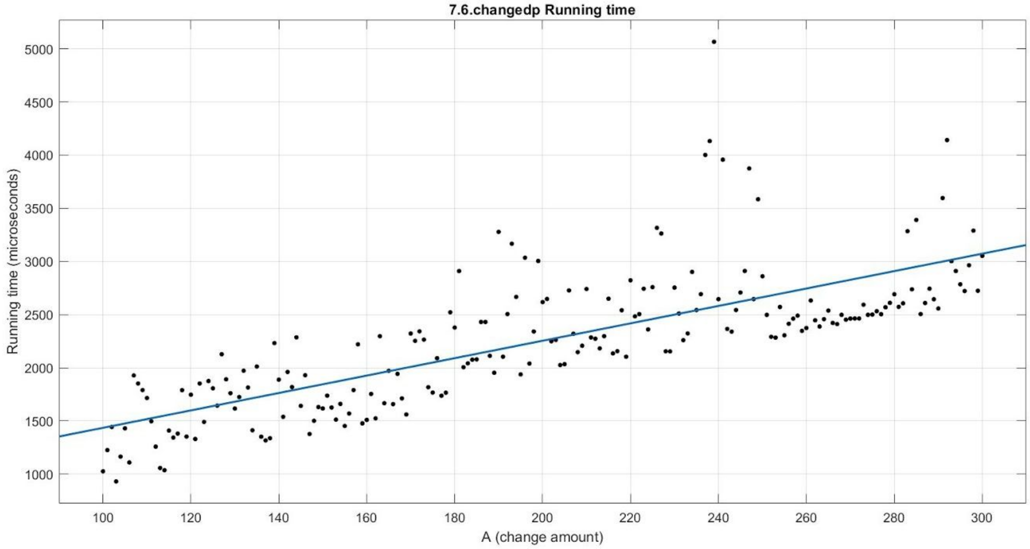

7.5b.changeslow Running time
×10^5
Running time (microseconds)
0
1
2
3
4
5
6
100
120
140
160
180
200
220
240
260
280
300
A (change amount)


7.5b.changegreedy Running time
Running time (microseconds)
10
15
20
25
100
120
140
160
180
200
220
240
260
280
300
A (change amount)


7.5b.changedp Running time
Running time (microseconds)
400
600
800
1000
1200
1400
100
120
140
160
180
200
220
240
260
280
300
A (change amount)


7.6.changeslow Running time
×10^8
0
0.5
1
1.5
2
2.5
3
3.5
4
Running time (microseconds)
0
20
40
60
80
100
120
A (change amount)


7.6.changegreedy Running time
10
15
20
25
30
35
40
45
50
55
Running time (microseconds)
100
120
140
160
180
200
220
240
260
280
300
A (change amount)

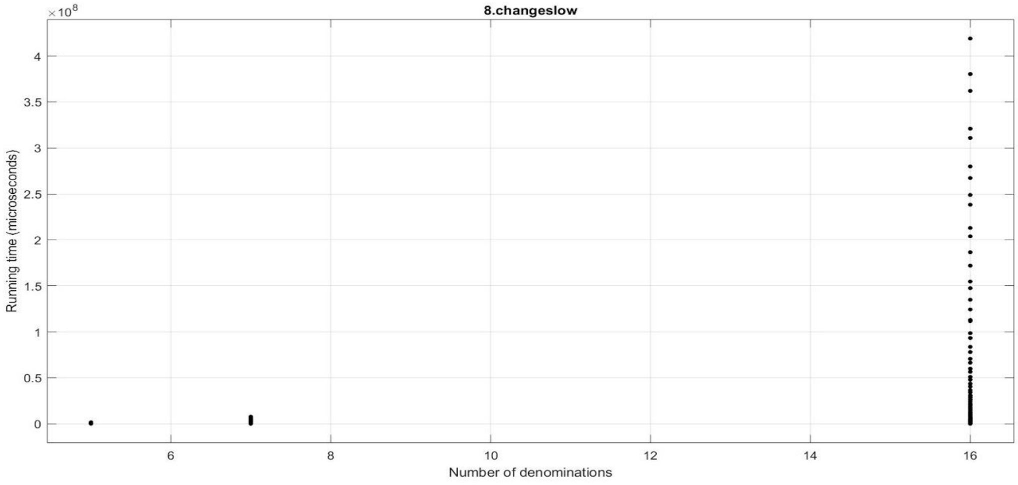

7.6.changedp Running time
1000
1500
2000
2500
3000
3500
4000
4500
5000
Running time (microseconds)
100
120
140
160
180
200
220
240
260
280
300
A (change amount)

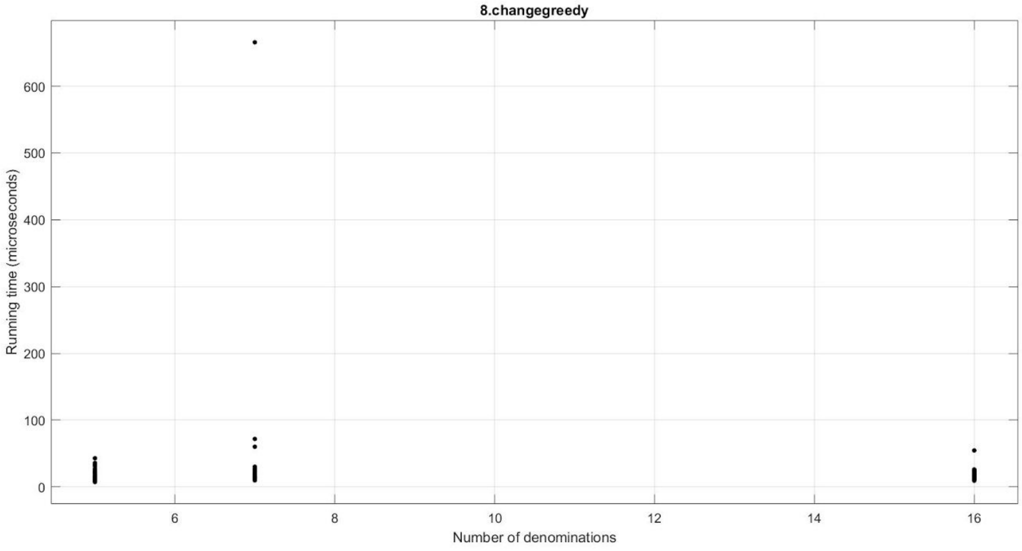
For each set of running times, changeslow ran the slowest, followed by changedp, and changegreedy ran the fastest. In all scenarios, the changeslow algorithm very clearly had an exponential running time as A got larger.

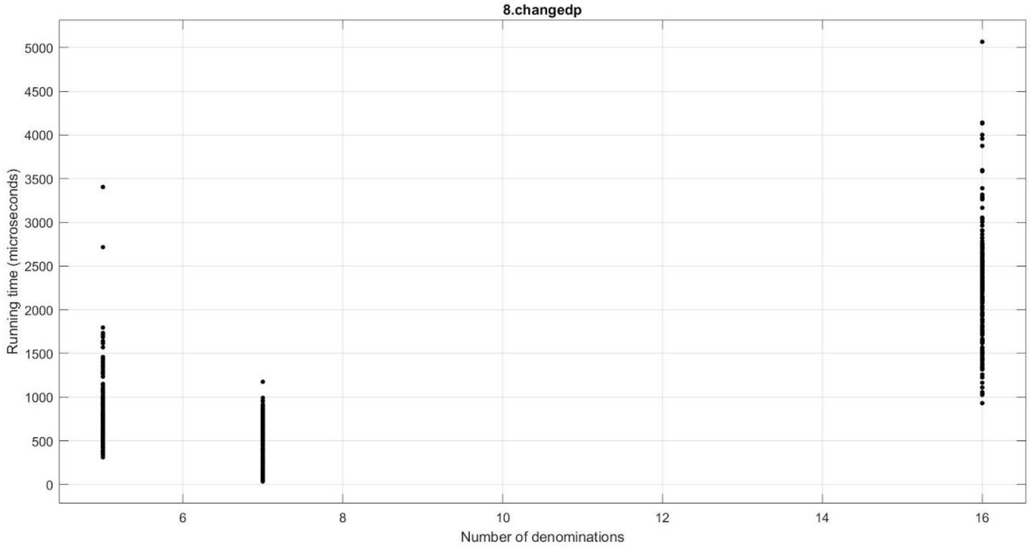
The changedp algorithm appeared to have a linear running time as A got larger. While some plots had varying outliers, there was a linear trend in all scenarios.

The changegreedy algorithm also appeared to have a linear running time, but with a lower slope than changedp (so the running times grew at a slower rate than changedp). In scenarios 4, 5a, and 5b, the slopes of the running times appeared to be almost constant or negative. This is probably due to the fact that changegreedy runs extremely fast compared to the other algorithms. While we were able to measure variations of hundreds of microseconds for changegreedy, we must keep in mind that hundreds of microseconds still isn't very long. Theoretically, we would expect a longer running time for extremely large values of A. In scenario 6, the changegreedy algorithm appeared to have a clearer, positive linear trend than the other scenarios. This could be due to the fact that scenario 6 had many more coin denominations than the other scenarios, and as a result, we were able to see more variation in running time as the values of A grew larger.

Q8: Running Time as a Function of Denominations







The number of denominations was a factor in the changeslow and changedp algorithms, while it was not a factor for changegreedy. This is expected, because changeslow calculates every solution to decide on the optimal one. With more denominations, there will be more possible solutions to choose from.

Similarly, changedp calculates and stores solutions (and uses the stored solutions to calulate the next solution), so the number of denominations will factor into the running time. In changegreedy, the algorithm only calculates a single solution, and when it switches coin denominations within that solution, it does so in constant time. The changegreedy algorithm does not necessarily need to check all the coin denominations to arrive at a solution.

Q9: Greedy vs. Dynamic for V = [1, 3, 9, 27]

We would expect the greedy and dynamic approaches would produce the same optimal results for the number of coins used to make change. Each denomination is a factor of the next value, so there is no negative effect if the greedy algorithm chooses a larger denomination over a smaller one. For example, a set of three 9 coins can be replaced by one 27 coin. This is similar to the partial knapsack problem, because each denomination is like a fraction of the next, and a greedy approach is able to provide the optimal solution to the partial knapsack problem.

Q10: Greedy Algorithm- Optimal Solutions

Like in question 9, the greedy algorithm will produce an optimal solution when the denominations are each factors of the next value (the larger coins are multiples of the smaller coins). In this case, choosing the largest coin value will always be more efficient. For example, if the algorithm chooses multiple smaller coins instead of the largest coin, then it is already not choosing the optimal solution, because the smaller coins will add up to the same value as the largest single coin, and having multiple smaller coins versus a single large coin is not optimal.