Garrett Bauer
Brendan Byers
Eddie Fox

Group 42

CS 325 Project 2: Coins

Pseudocode and Theoretical Run-time Analysis:

Algorithm 1: Divide and Conquer

Pseudocode:

Values is the array of denominations.
Number is the amount of currency we are trying to make change for.

```
Function changeslow(values[], number):
        if(values is empty):
                Return [INT_MAX]

        Var goal := number
        Var totals[values.size()]
        Var it := values.size() - 1

        while(goal > 0):
                if(values[it] <= goal):
                        Goal -= values[it]
                        Totals[it]++
                Else:
                        It--

        Var temp[] = values
        temp.pop_back()        //Remove last element
        Var compare[] = changeslow(temp[], number)

        Var sum1 = 0, sum2 = 0

        for(n in totals[]):
                Sum1 += n
        for(n in compare[]):
                Sum2 += n
```

Return totals[] if sum1 > sum2, otherwise return compare[]

Theoretical Asymptotic Running Time:

Exponential running time. Recursive solutions are continually re-calculated instead of saving values of previously calculated solutions. As the amount increases, the changeslow function must be called an increasing amount of times. Similar to why the Fibonacci numbers recursive solution is exponential.

Algorithm 2: Greedy

Pseudocode:

Values is the array of denominations.
Number is the amount of currency we are trying to make change for.

Function changeGreedy(values[], number):

```
Var goal = number
Var totals[values.size]
Var it = values.size() - 1

While (goal > 0):
        If values[it] <= goal:
                goal -= values[it]
                totals[it]++
        Else:
                It--

End while loop

Return totals
```

Theoretical Asymptotic Running Time:

Linear runtime. Most of the operations in this algorithm are constant time. In general, the running time of the algorithm will scale linearly with the amount to be calculated as the amount, because the amount of times the loop is executed is dependent on the amount. The larger the amount, the more times the loop will need to run to subtract denominations until goal is reduced to 0. It will scale by a linear factor because each loop is just decreasing the amount by the largest denomination not yet processed each loop and decrementing the denomination index once the denomination amount can no longer be subtracted. Both of these are constant time

O(1) operations. It isn't quite O(n) because the runtime is dependent on the exact denominations, but as factors such as amounts increase, it should be close enough to O(n). Running time = O(n)

Algorithm 3: Dynamic Programming

Pseudocode:

```
Function changedp(values[], number):
    if(not number):
        Var none = [0]
        Return none

    Var T[] = [INT_MAX-1, …] //Size of T[] is number+1
    Var R[] = [-1, -1 …] //Size of R[] is number+1

    T[0] = 0

    for(j=0; j<values.size(); j++):
        for(i = 1; i <= number; i++):
            if(i >= values[j]):
                if(T[i-values[j]]+1 < T[i]):
                    T[i] = 1 + T[i-values[j]]
                    R[i] = j

    Var Used[] = [0, 0, …] //Size of used[] is values.size()
    Var start = R.size()-1
    while(start != 0):
        Var j = R[start]
        Used[j]++
        Start = start-values[j]

    Return used[]
```

Theoretical Asymptotic Running Time:

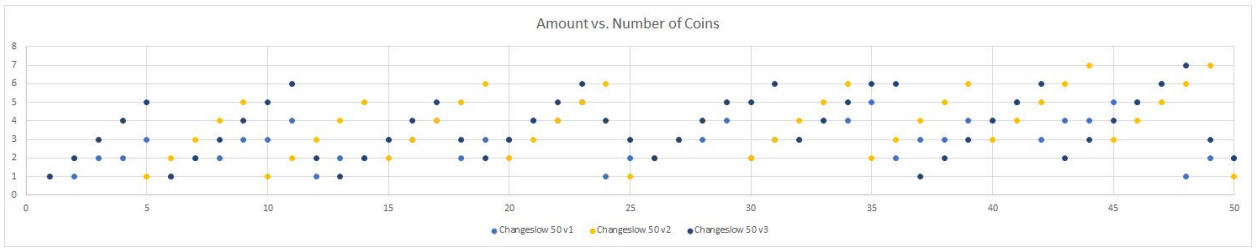
The runtime for changedp is $O(n * m)$ where n is the length of the array values[] (the number of denominations) and m is the amount of change we are trying to calculate. This is because the main nested loop does m calculations n times. Every other calculation in the function is either constant time or irrelevant as n and m approach infinity.

How the changedp dynamic programming table was filled:

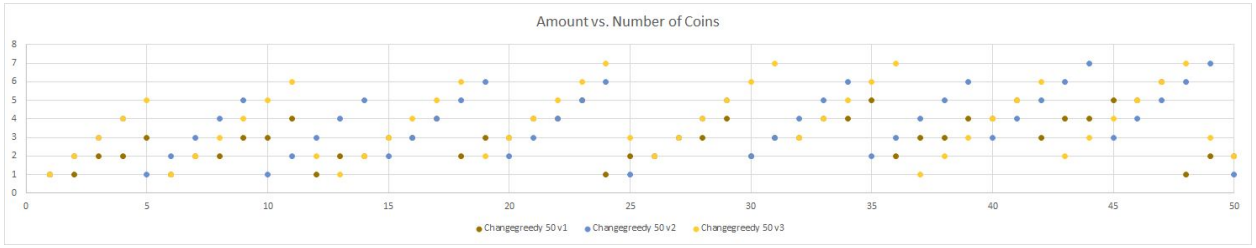
Each solution in the set of all solutions must be between 1 and n number of coins. The table that holds the values is the same size as the number to make change of. Single coin solutions obviously use only one coin and are easy to figure out. If a solution takes one coin, there is no way to get any fewer coins. If the value takes more than one coin, the it'll be the summation of coins previously figured out. As this algorithm goes through possible numbers, it saves values in the table, allowing them to be referenced later to quickly figure out the optimal solution.

Plot of Number of coins as a function of Amount:

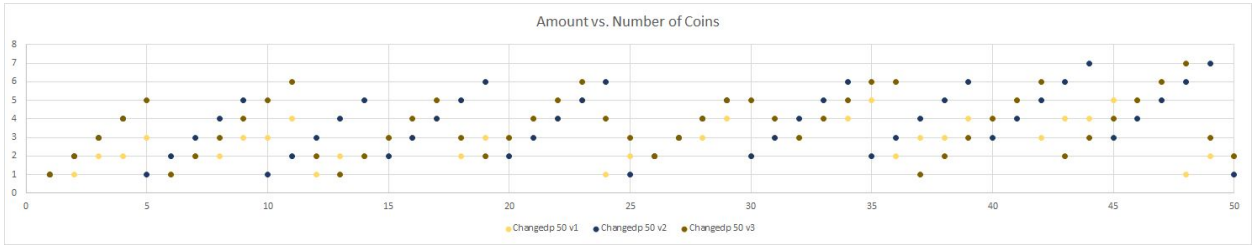
Recursive:



Greedy:



Dynamic Programming:

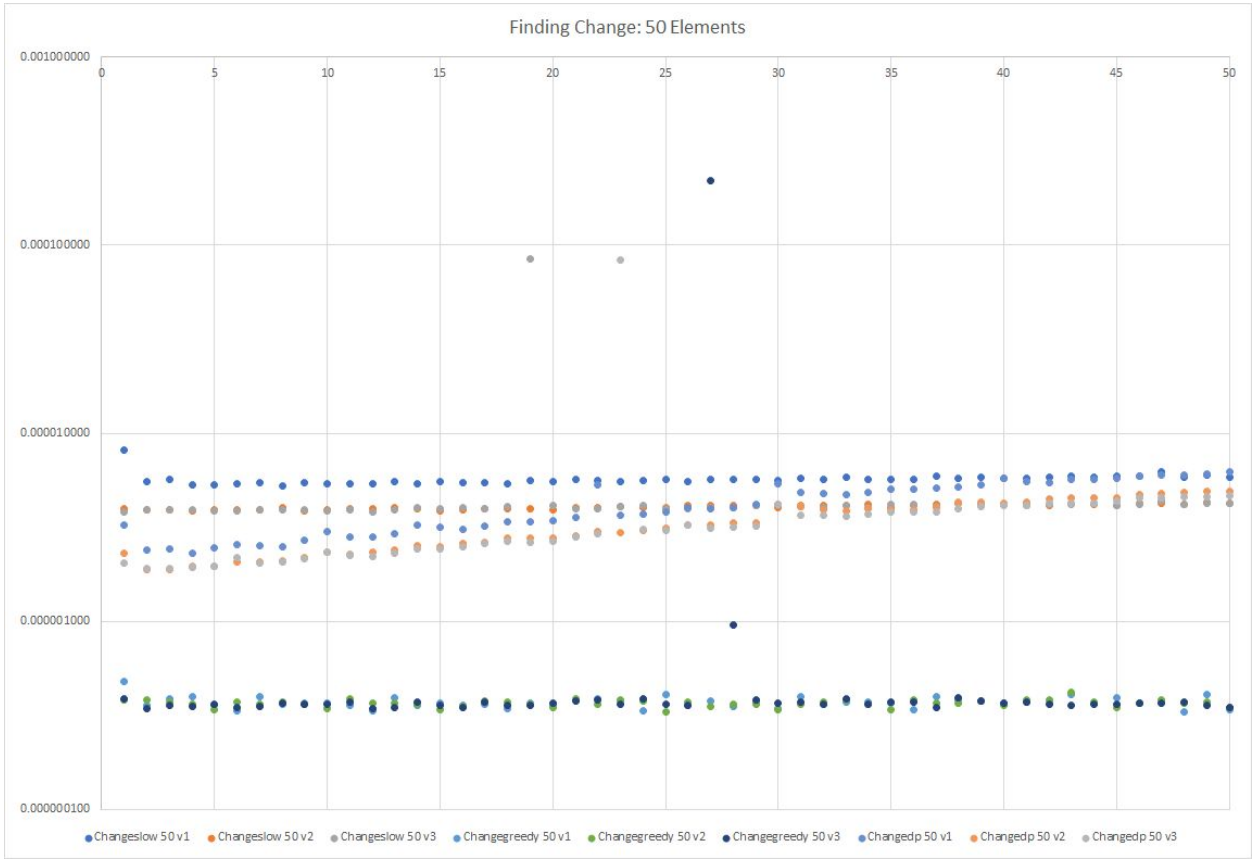


Comparison of Approaches:

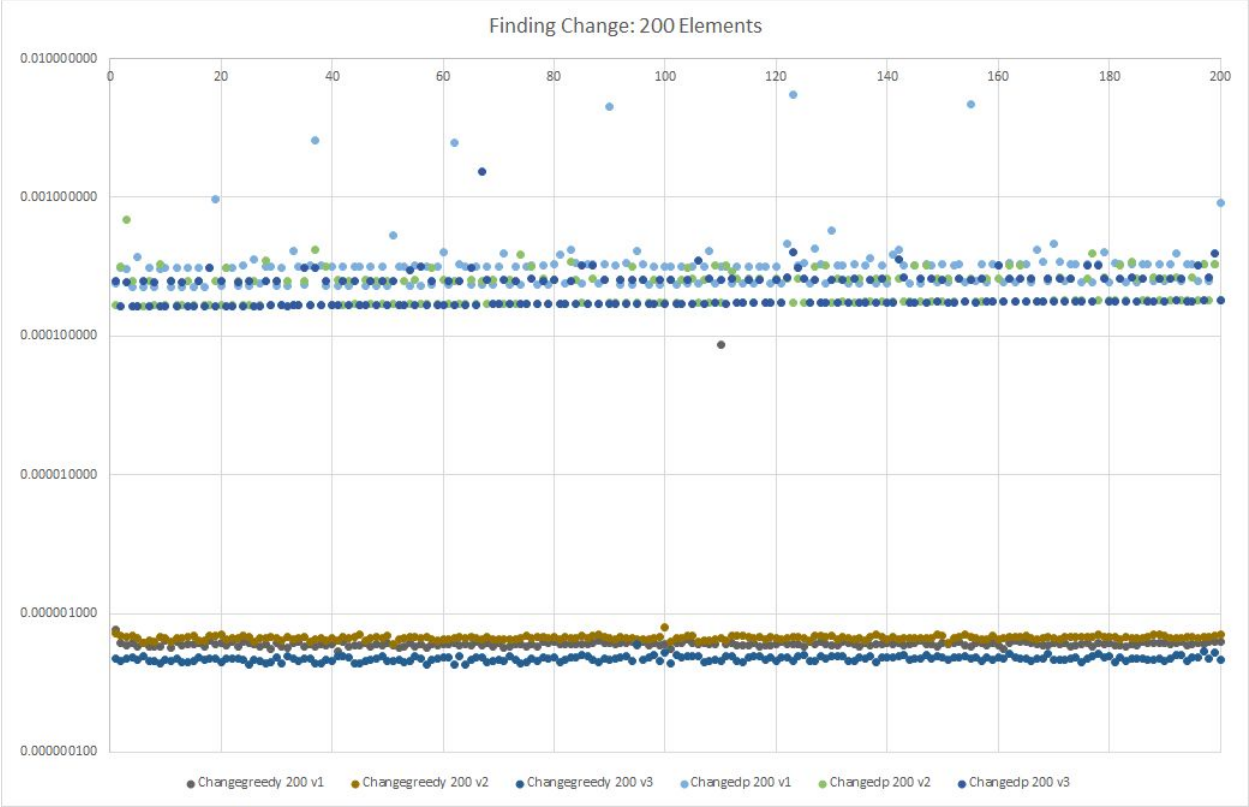
The recursive and dynamic algorithms agreed on the number of coins regardless of the denominations used or the amount, while the greedy algorithm occasionally differed.

Plot of running time as a function of A:

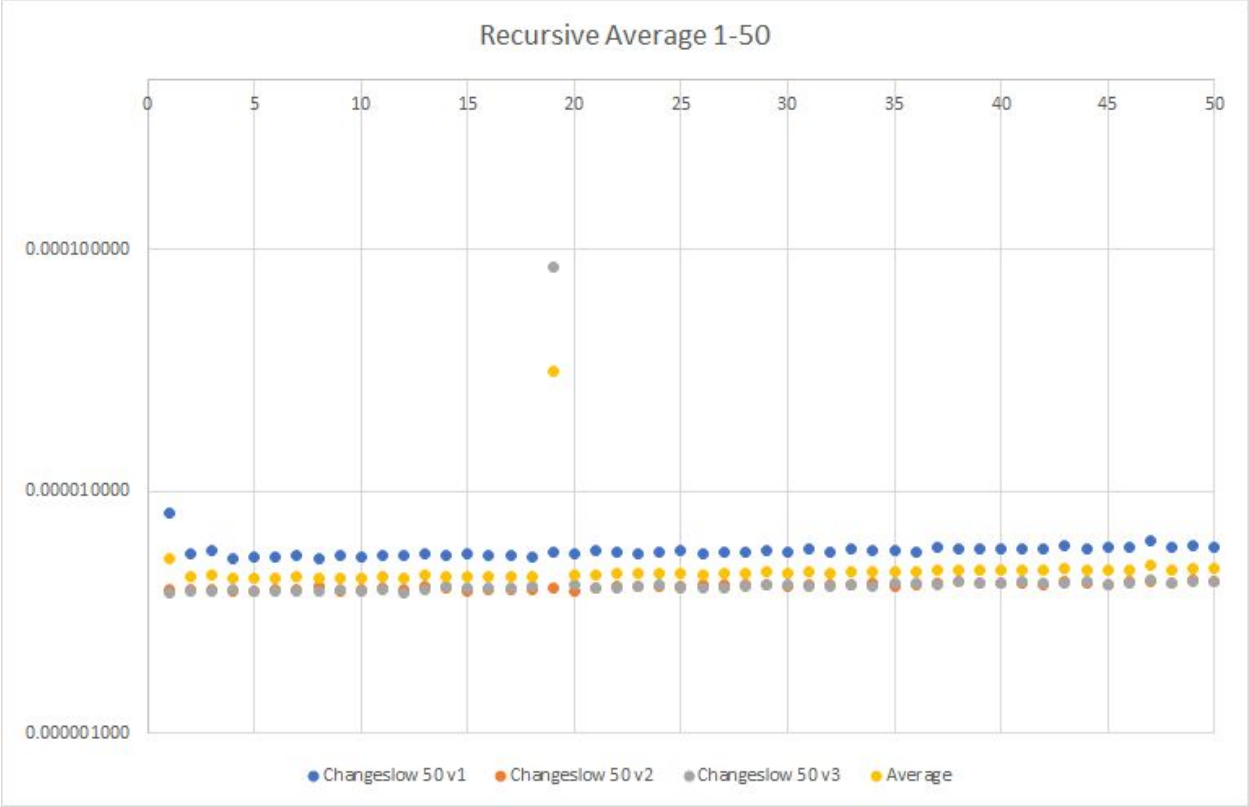
All algorithms, all denominations sets on one graph. Amounts 1-50:



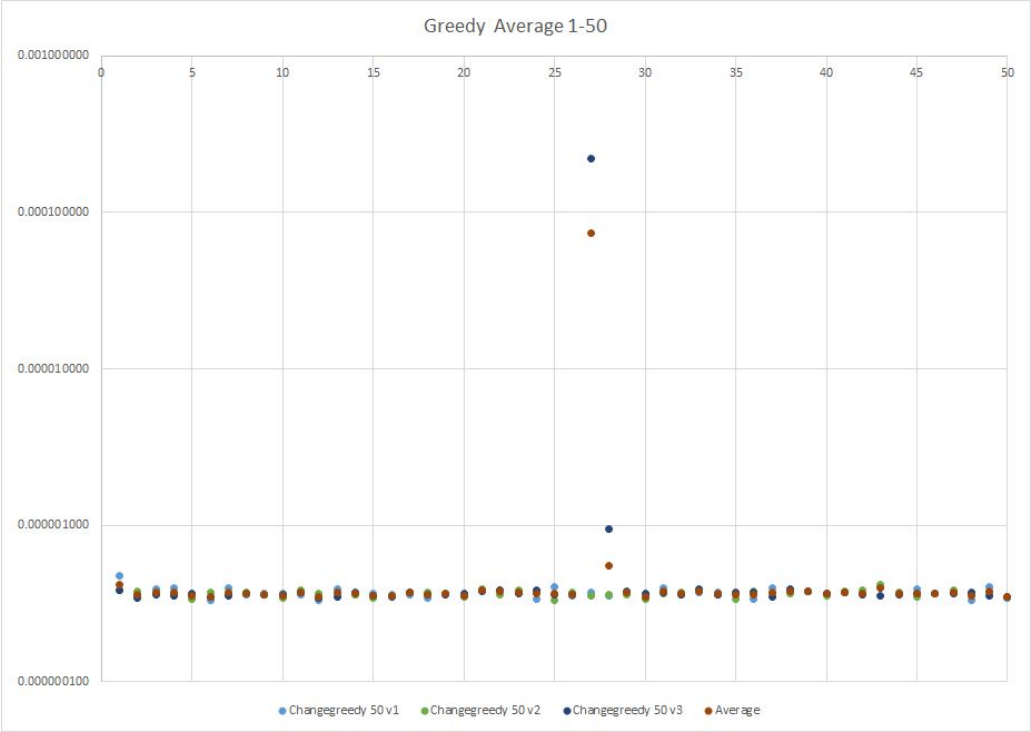
All algorithms, all denomination sets on one graph. Amounts 2000-2200:



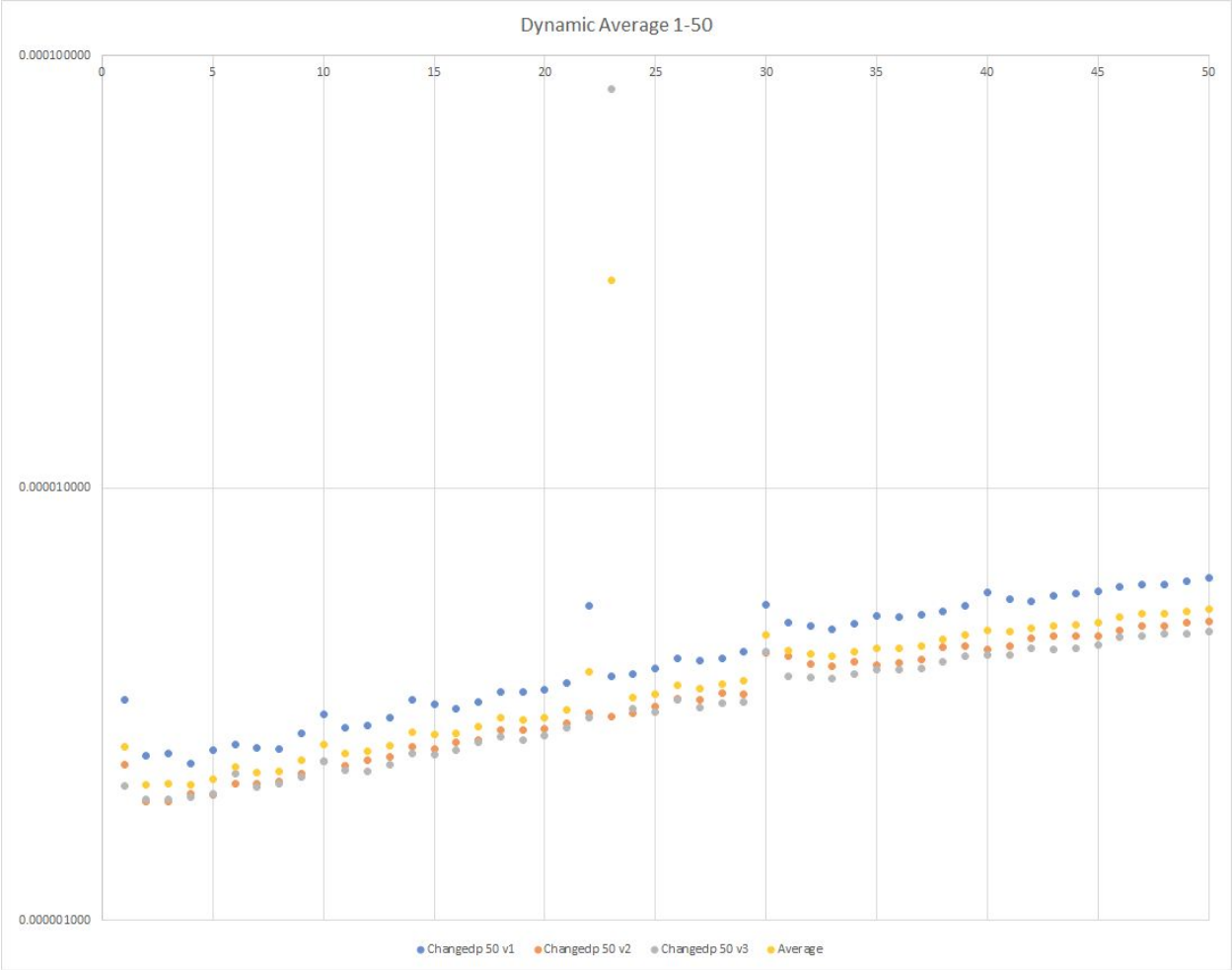
Recursive:


Recursive Average 1-50
0
5
10
15
20
25
30
35
40
45
50
0.000100000
0.000010000
0.000001000
Changeslow 50 v1
Changeslow 50 v2
Changeslow 50 v3
Average

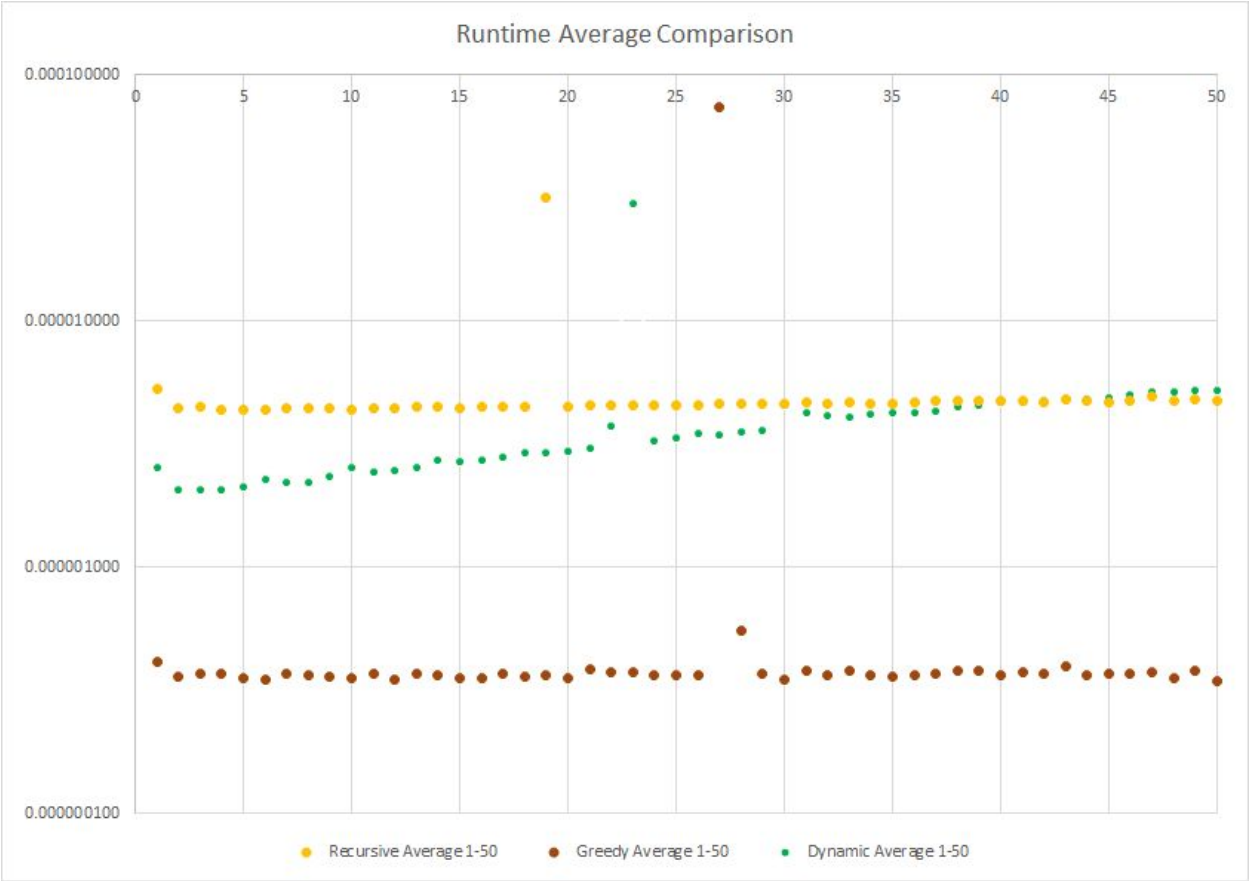
Greedy:


Greedy Average 1-50
0.001000000
0.000100000
0.000010000
0.000001000
0.000000100
0
5
10
15
20
25
30
35
40
45
50
Changegreedy 50 v1
Changegreedy 50 v2
Changegreedy 50 v3
Average

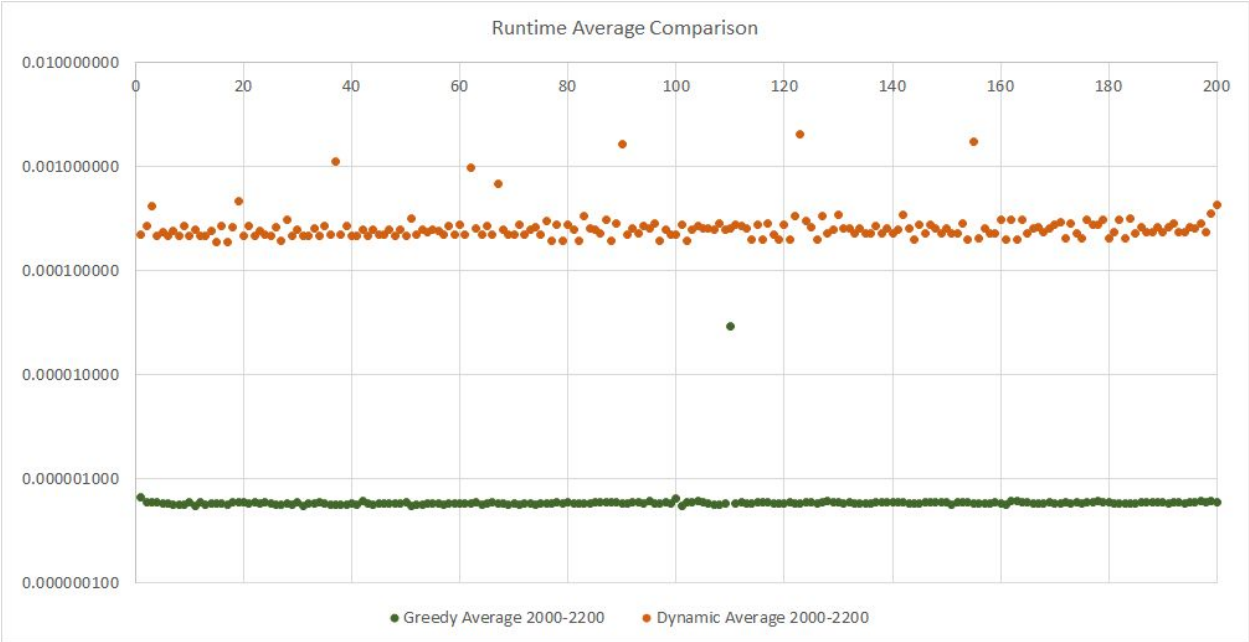
Dynamic Programming:



Averages of all algorithms compared, Amounts 1-50:



Averages of all algorithms compared, Amounts 2000-2200:



Curve Fit Equation for each Algorithm:

Recursive:

$7*10^{-9} * x + 4*10^{-6}$

Greedy:

$1*10^{-10} * x + 4*10^{-7}$

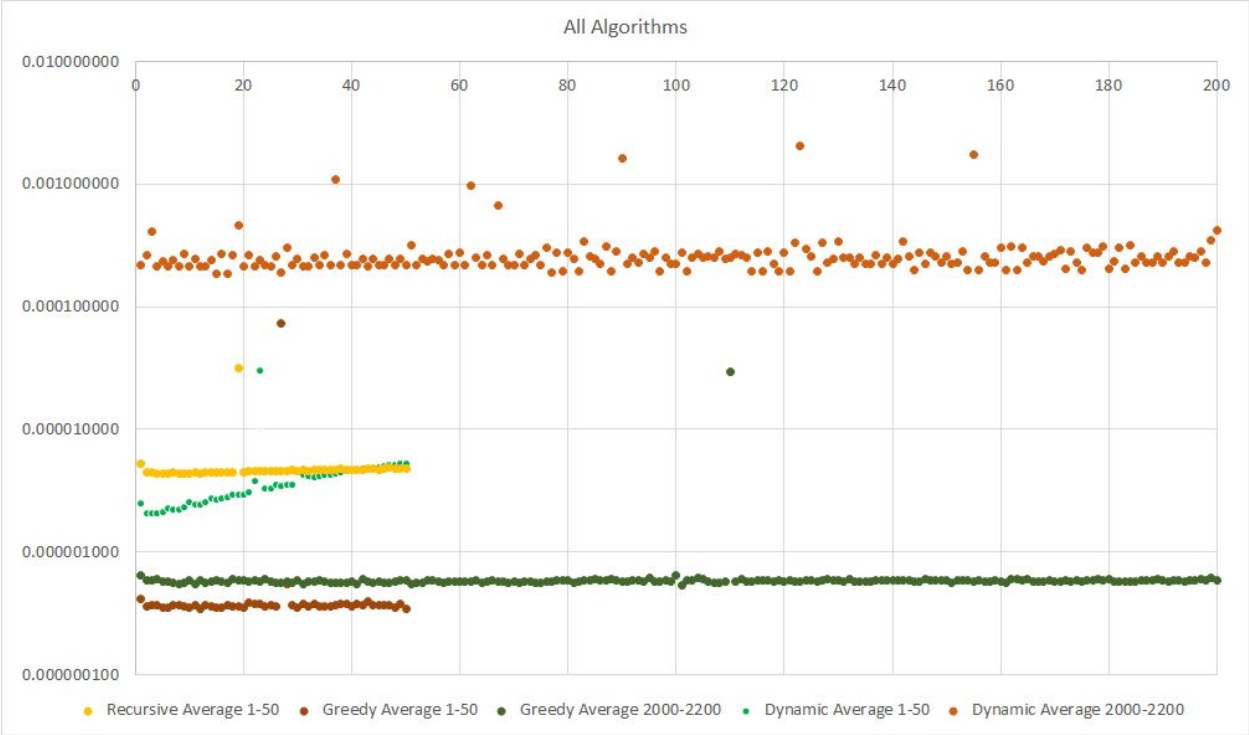
Dynamic Programming:

$6*10^{-8} * x + 2*10^{-6}$

Comparison of practical and theoretical running times:

As expected, the algorithms ran roughly in order of speed (fastest to slowest) of Greedy, Dynamic, and recursive. We expected dramatic differences in runtime, but it is likely that at such low constants of amount of change, the sheer speed of a computer overwhelmed the theoretical runtime differences at 1-50. It is likely that somewhere between 50 and 2000-2200, the theoretical difference does manifest itself and explains why we couldn't record the runtime for Recursive (probably too high), for A = 2000-2200.

Log-log plot of all three algorithms:



Comparison of algorithmic running time:
As expected, greedy is the fastest algorithm, followed by dynamic and then recursive.

Powers of Three Denomination Approach Comparison:

Generally, the greedy algorithm will operate faster than even dynamic programming, but does not guarantee an optimal solution. A dynamic programming solution guarantees an optimal solution, so it is generally preferable, even if it takes somewhat longer. For the given denominations V = [1, 3, 9, 27], a greedy solution will always produce an optimal solution. Because with these denominations, it is equally optimal to a dynamic programming algorithms, but as fast or faster, you would always want to use a greedy algorithm in this country.

Situations Where the Greedy Algorithm is Optimal:

Several examples:

V = [1, 3, 9, 27, 81]
V = [1, 2, 4, 8, 32]

V = [1, 10, 20, 40]
V = [1, 4, 16, 64]

The greedy algorithm is guaranteed to be optimal in cases where each successive denomination is divisible by all the denominations before it. This is because such an arrangement makes each larger denomination a multiple of the ones before it, in other words, a multiple of all other possible denominations. In cases where it is possible to use a largest or larger denomination coin, it is always optimal because forming the exact same amount with smaller denominations is guaranteed, due to the aforementioned divisibility, but it would take a greater number of coins and be sub-optimal. Because using the largest denominations when possible (given the denomination stipulation) always contributes to an optimal global solution, we do not need to worry about having local maximums, and can proceed to always subtract the largest denomination, switching to the next largest when the remaining amount is sufficiently depleted.