CS325 Project 2: Coin Change

Prepared by P2 Group 35

Jason DiMedio(dimedioj)
James Palmer(palmerja) &
Babatunde Ogunsaju(ogunsajb)

Algorithm 1: Brute Force

Pseudo Code

```
CHANGE_SLOW(coin_array, change)
        coin_count = [0,...,coin_array.length - 1] = 0
 Main
      if (change exists in coinArray)
          coin_count[index of change] = 1
          return coin_count, 1
      else if (change = 0)
          return coin_count, 0

          min_coins = infinity

          return coin_count, min_coins
Some function
for (i = 1 to change/2)
                    coin_count_left, min_coins_left = CHANGE_SLOW(coin_array, i)
                    coin_count_right, min_coins_right = CHANGE_SLOW(coin_array, change - i)
                    total_coins = min_coins_left + min_coins_right

                    if total_coins < min_coins:
                              min_coins = total_coins

                              for (j in coin_count_left/right)
                                        coin_count = coin_count_left[j] + coin_count_right[j]
```

Theoretical Run-time Analysis

The algorithm takes two inputs: coinArray and change. The parameter coinArray contains all of the possible denominations of coins that can be used to make change with. The parameter change is the

amount of change that needs to be composed of various combinations of coins that are denoted in the coinArray. From this point forward we will refer to the length of the coinArray as k and the value for change as n.

As can be seen from the pseudo-code above, we first check our k coins to see if any are exactly equal to our change amount. After that, a for-loop runs from 1 to /n , where this value is assumed to be an integer. Our recurrence is of the form: $t(n) = \sum_{i=1}^{n/2} (T(n-i)+T(i)) + k$

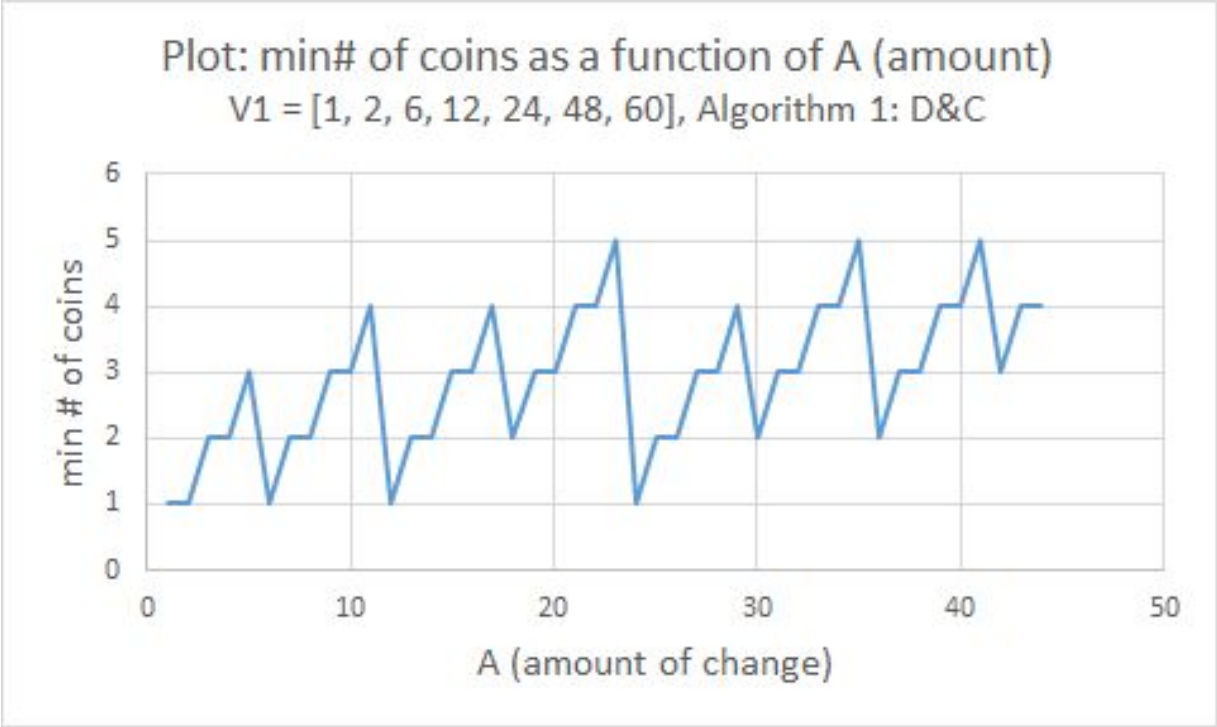
The loop only needs to run to n/2 due to the fact that values above n/2 will simply recompute a sum already computed. As an example, consider the case where n=5 . When i = 2, we compute T(3) + T(2). If we continued on to i = 3 , we would compute T(2)+ T(2) $\equiv$ T(3) = T(2)

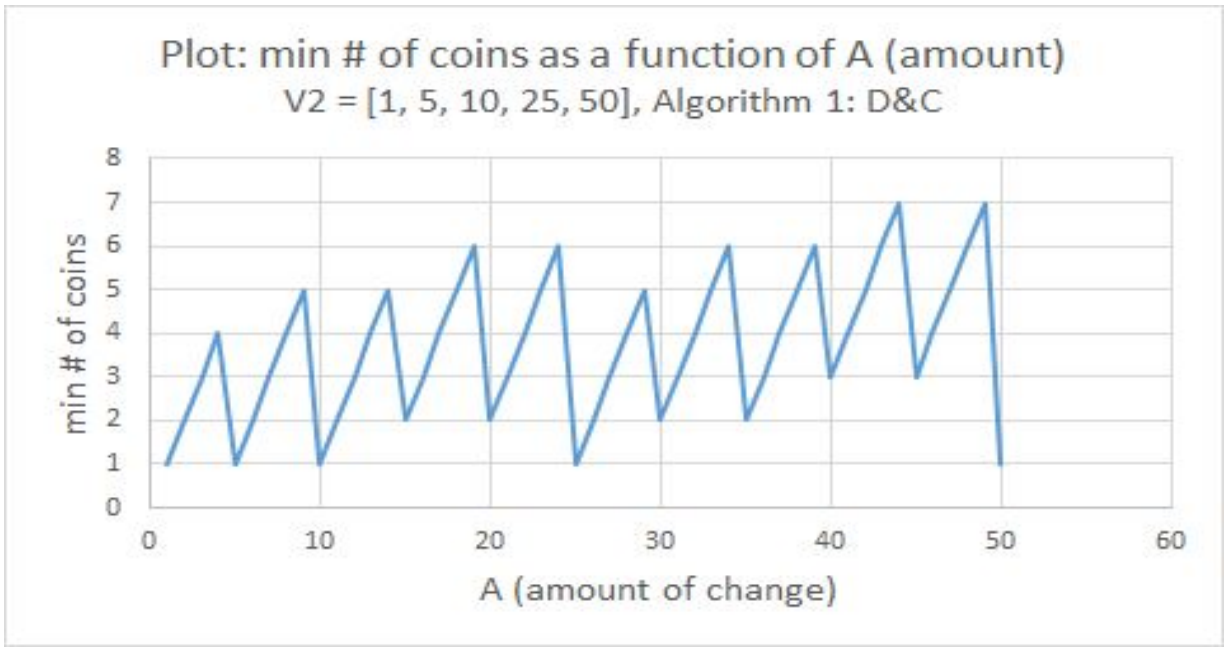
While this recurrence is difficult to solve, we can consider the case without our optimization and consider it an upper bound. Consider the recurrence: $T(n) = \sum_{i=2}^{n-1} (T(n - i) + T(i)) + k$

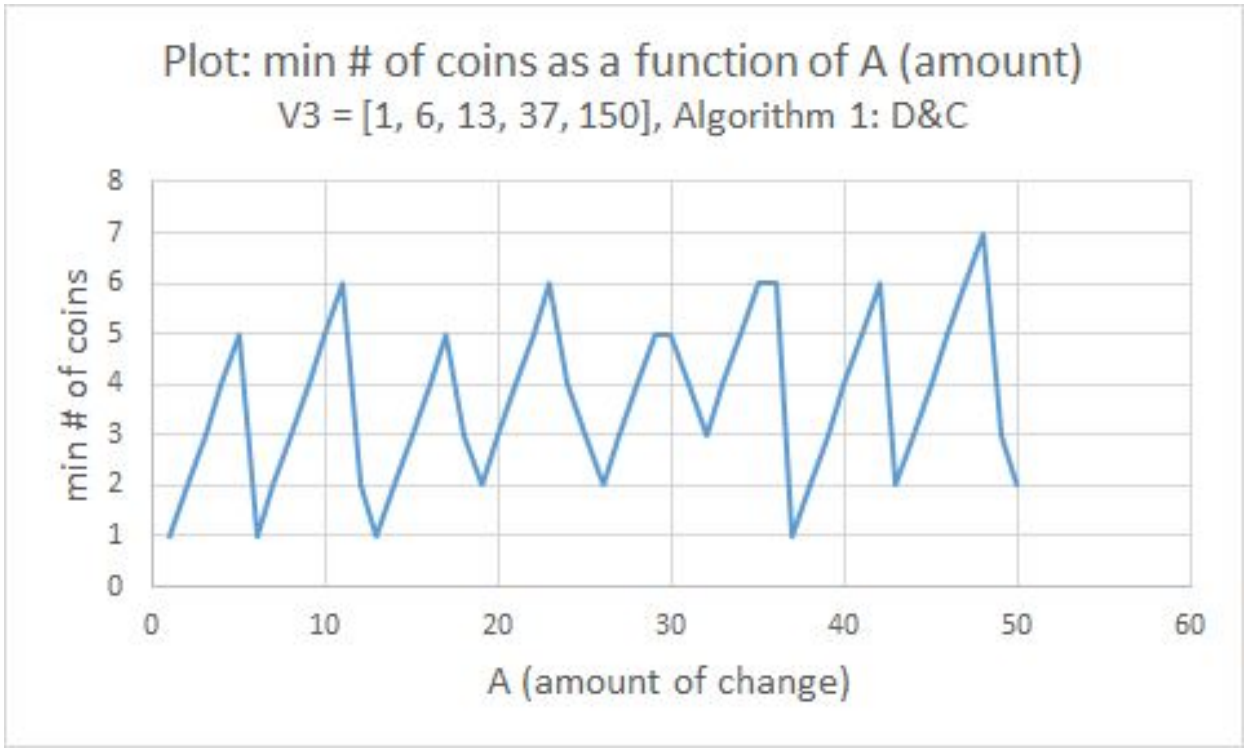
Solving for T(n), we can conclude that: T(n) = 3T(n-1)

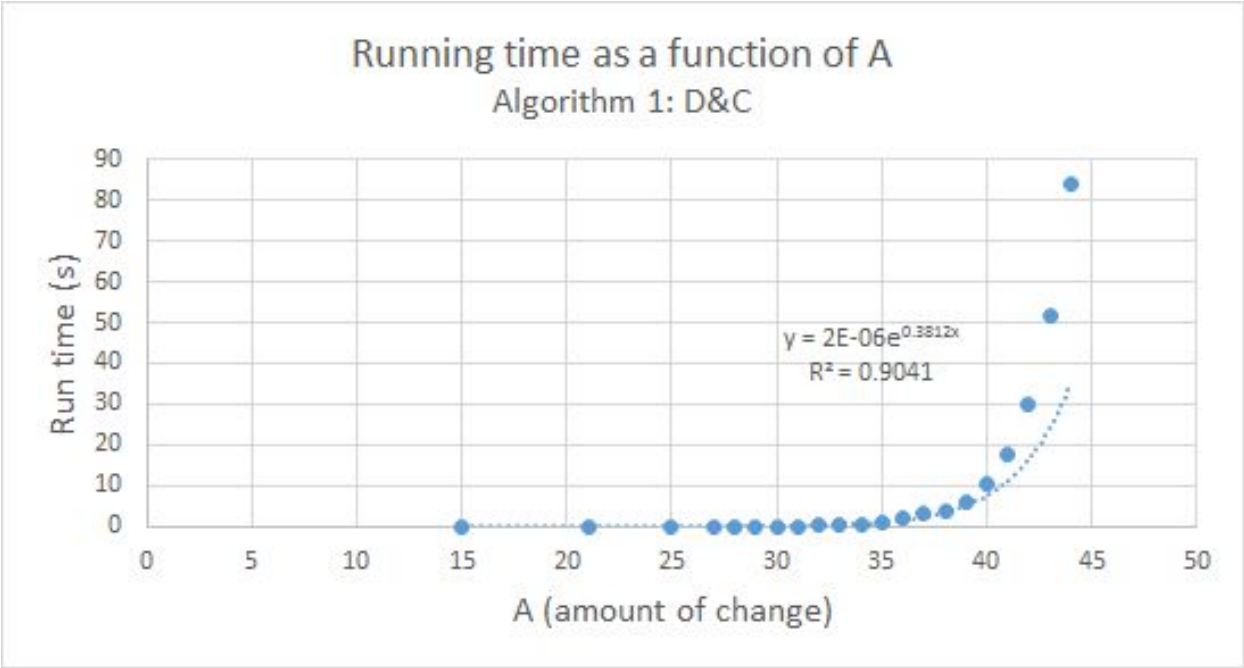
In short, by the Muster Theorem talked about in our teachings and seen with our coding results, with a = 3, b = 1 and d = 0, we can say that T(n) is $O(3^n)$

Experimental Analysis




Plot: min # of coins as a function of A (amount)
V2 = [1, 5, 10, 25, 50], Algorithm 1: D&C
min # of coins
0
1
2
3
4
5
6
7
8
0
10
20
30
40
50
60
A (amount of change)


Plot: min # of coins as a function of A (amount)
V3 = [1, 6, 13, 37, 150], Algorithm 1: D&C
min # of coins
0
1
2
3
4
5
6
7
8
0
10
20
30
40
50
60
A (amount of change)



Algorithm 2: Greedy Algorithm

Pseudo-Code

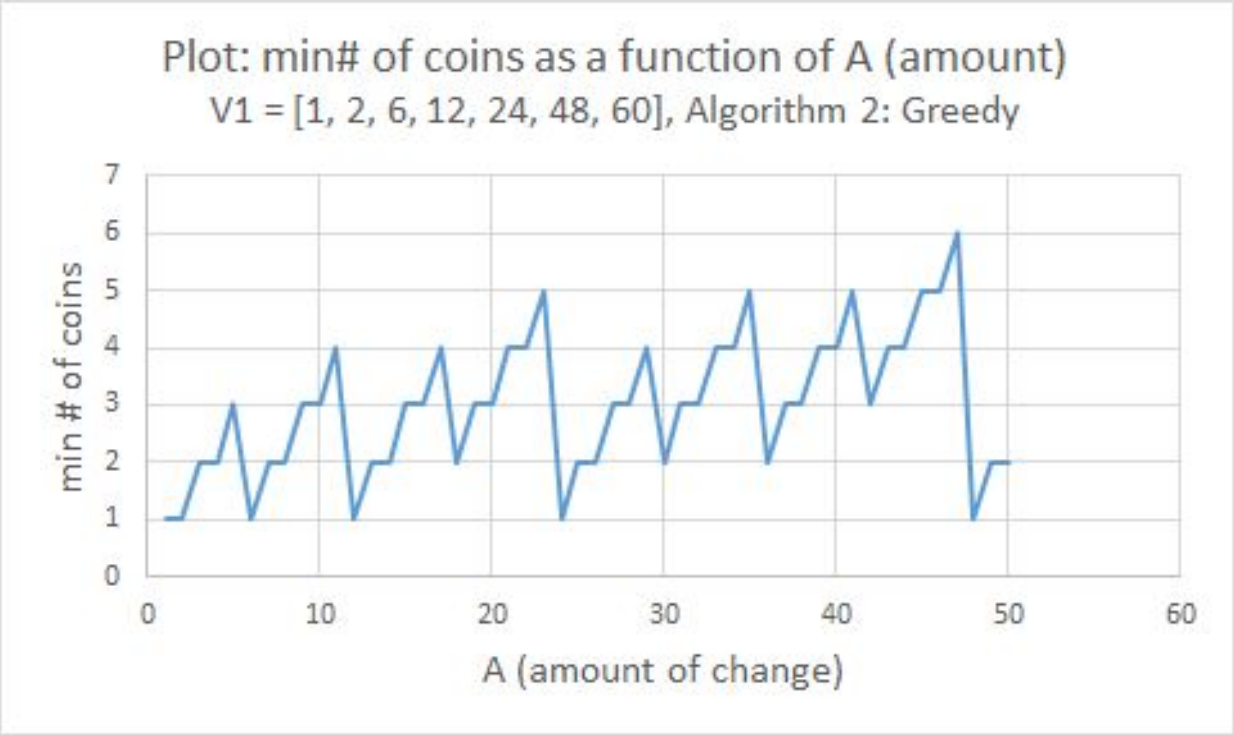
```
Let R be a set of n quantities of each coin denominations in optimal solution
Let min be the minimum number of coins required
for i = n-1 to 0
        while V[i] <= A
                A = A - V[i]
                R[i]++
                min++

return R and min
```

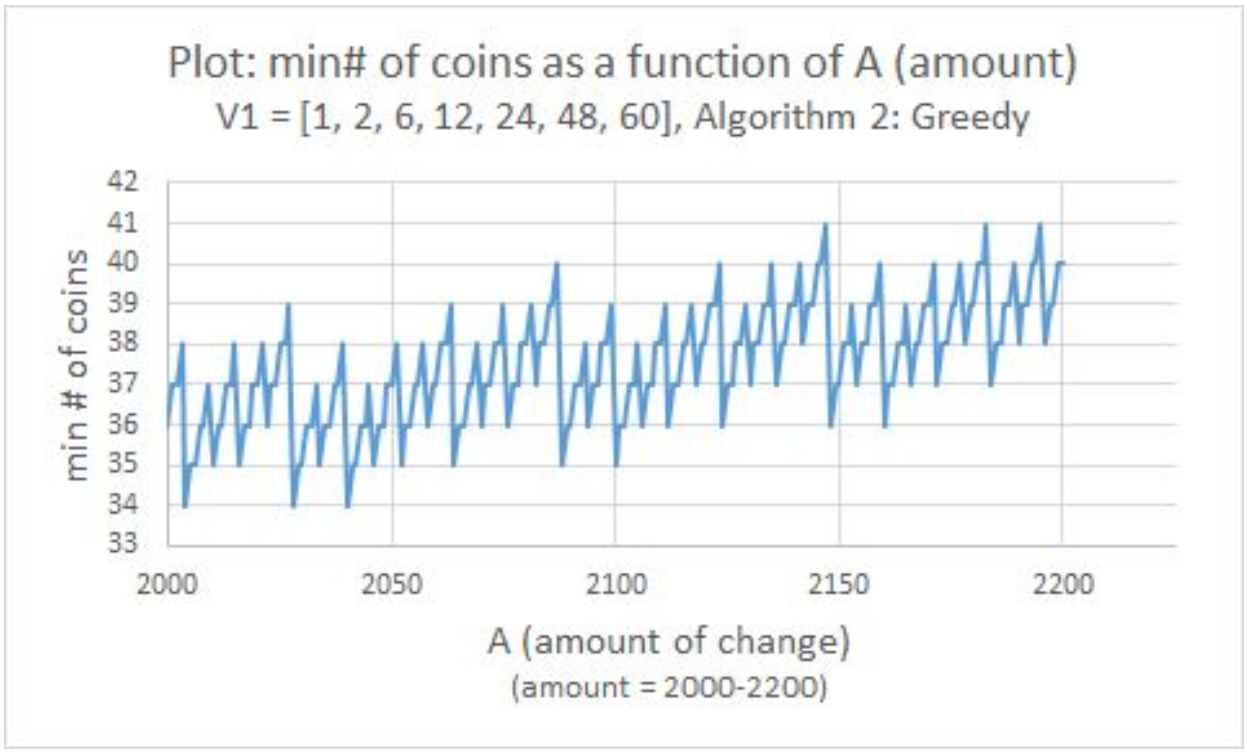
Theoretical Run-time Analysis

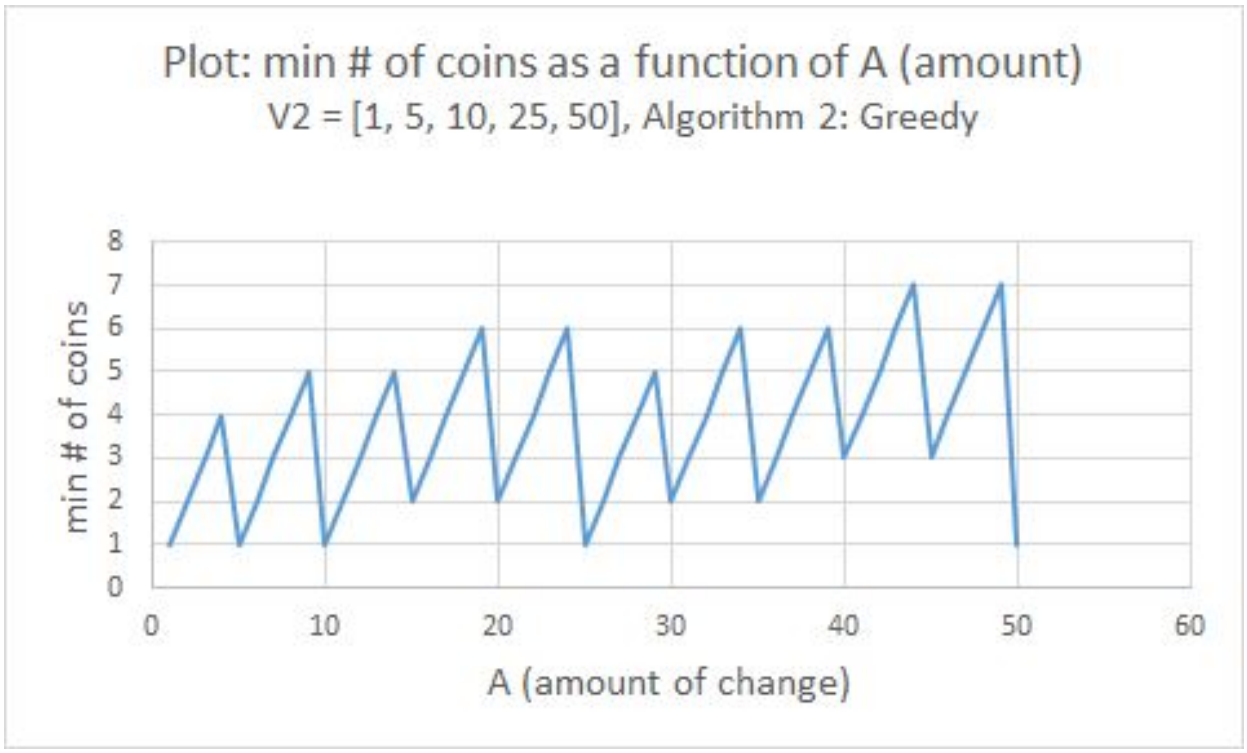
Because V is already sorted, changegreedy() iterates over V exactly once. However, in the process of iterating through V, the algorithm loops through A (amount of change), decrementing A each time by one of the values within V until A reaches 0. If n is the number of elements in V, and k is the number of times A is decremented, then the running time is $\Theta(nk)$.

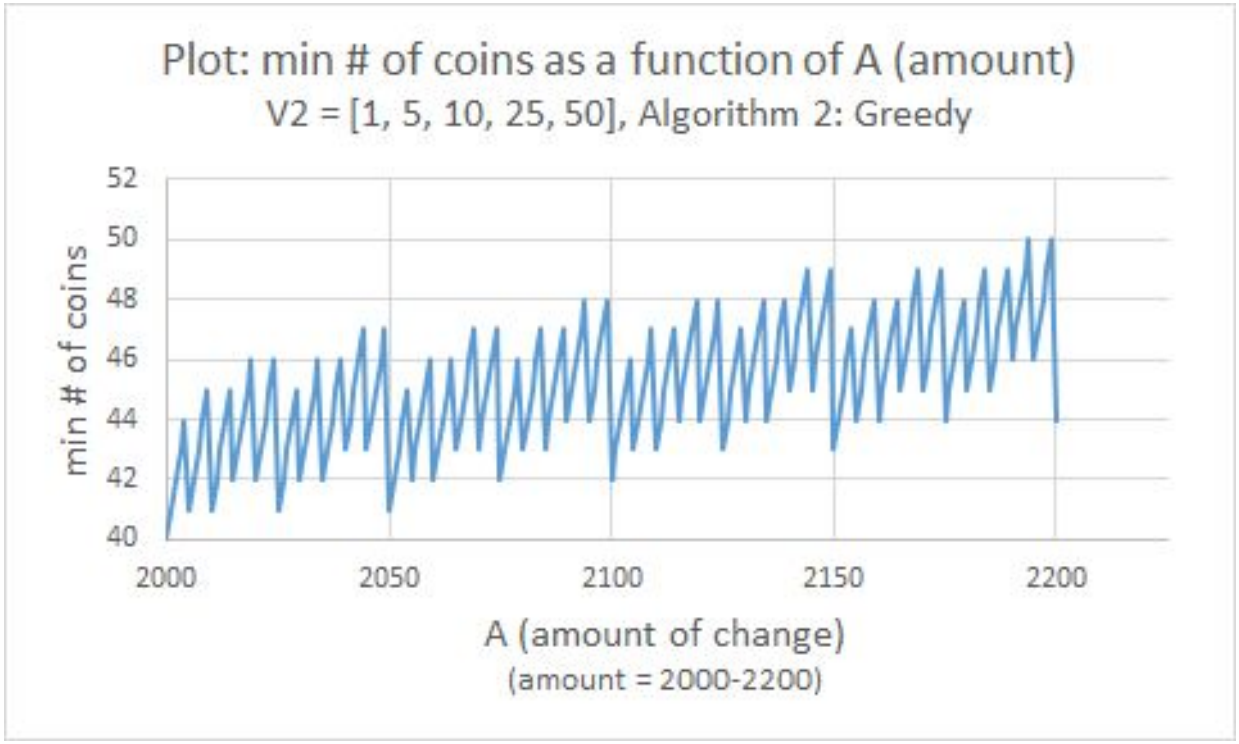
Experimental Analysis

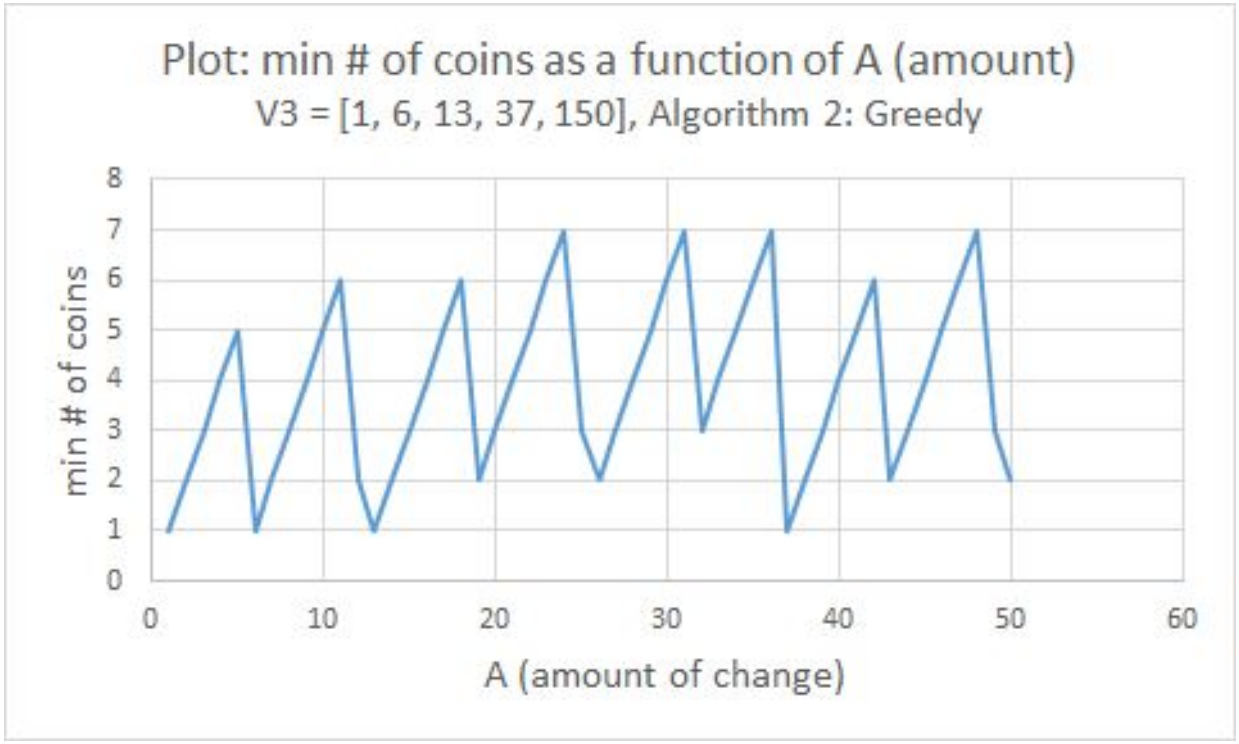
Number of coins as a function of *A*

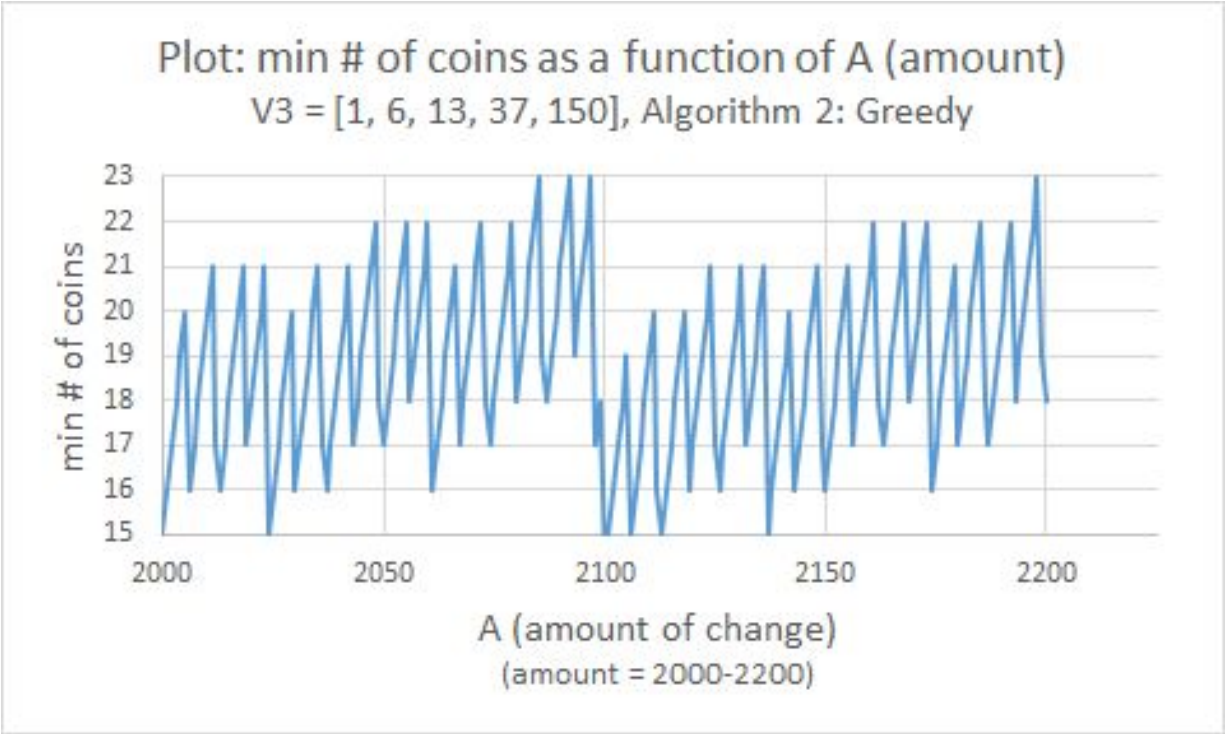


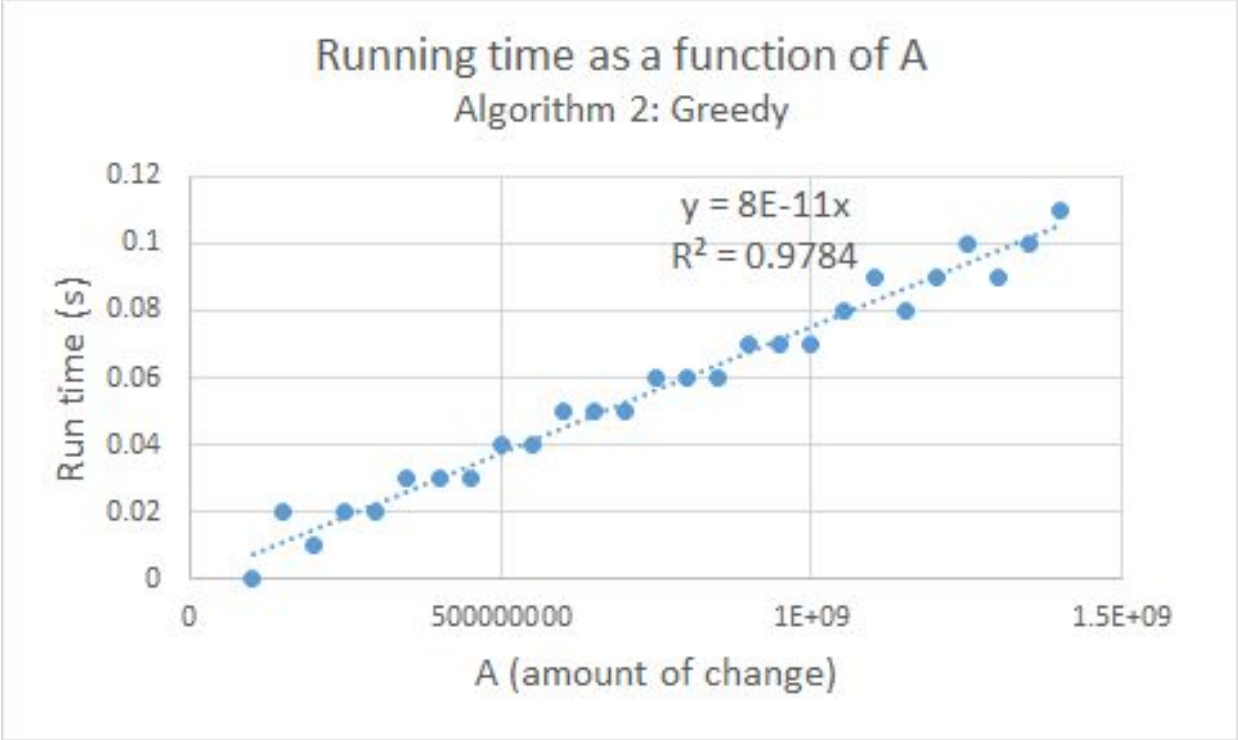

Plot: min# of coins as a function of A (amount)
V1 = [1, 2, 6, 12, 24, 48, 60], Algorithm 2: Greedy
min # of coins
42
41
40
39
38
37
36
35
34
33
2000
2050
2100
2150
2200
A (amount of change)
(amount = 2000-2200)


Plot: min # of coins as a function of A (amount)
V2 = [1, 5, 10, 25, 50], Algorithm 2: Greedy
min # of coins
8
7
6
5
4
3
2
1
0
0
10
20
30
40
50
60
A (amount of change)


Plot: min # of coins as a function of A (amount)
V2 = [1, 5, 10, 25, 50], Algorithm 2: Greedy
min # of coins
52
50
48
46
44
42
40
2000
2050
2100
2150
2200
A (amount of change)
(amount = 2000-2200)


Plot: min # of coins as a function of A (amount)
V3 = [1, 6, 13, 37, 150], Algorithm 2: Greedy
min # of coins
8
7
6
5
4
3
2
1
0
0
10
20
30
40
50
60
A (amount of change)





The experimental results are consistent with the theoretical analysis of $\Theta(nk)$. In the results, the trend line is consistent with a linear equation, with an R^2 value near 1.

Algorithm 3: Dynamic Programming

Pseudo Code

```
1       C[0] = 0
2       for p = 1 to n
3               min = ∞
4               for i = 1 to k
5                       if (p ≥ di) then
6                       if (C[p−di]) + 1 < min) then
7                               min = C[p−di] + 1
8               count[p][i]++
9               C[p] = min
```

Theoretical Run-time Analysis

There are 2 loops in this algorithm, with the first running from 1 to n and the second running from 1 to k. Therefore, the min calculation on line 7 will run n*k times to compute minimum number of coins. This algorithm has an asymptotic runtime of Θ(nk)

Experimental Analysis

How you fill in the dynamic programming table

Our dynamic programming table will be filled in a bottom up manner using the recursive formula

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

This formula can be further stated as follows:

if v = 0, then T[v] = 0 //base case

else

T[v] = min{T[v-V[i]]+1} for V[i]< or = v //recursive case

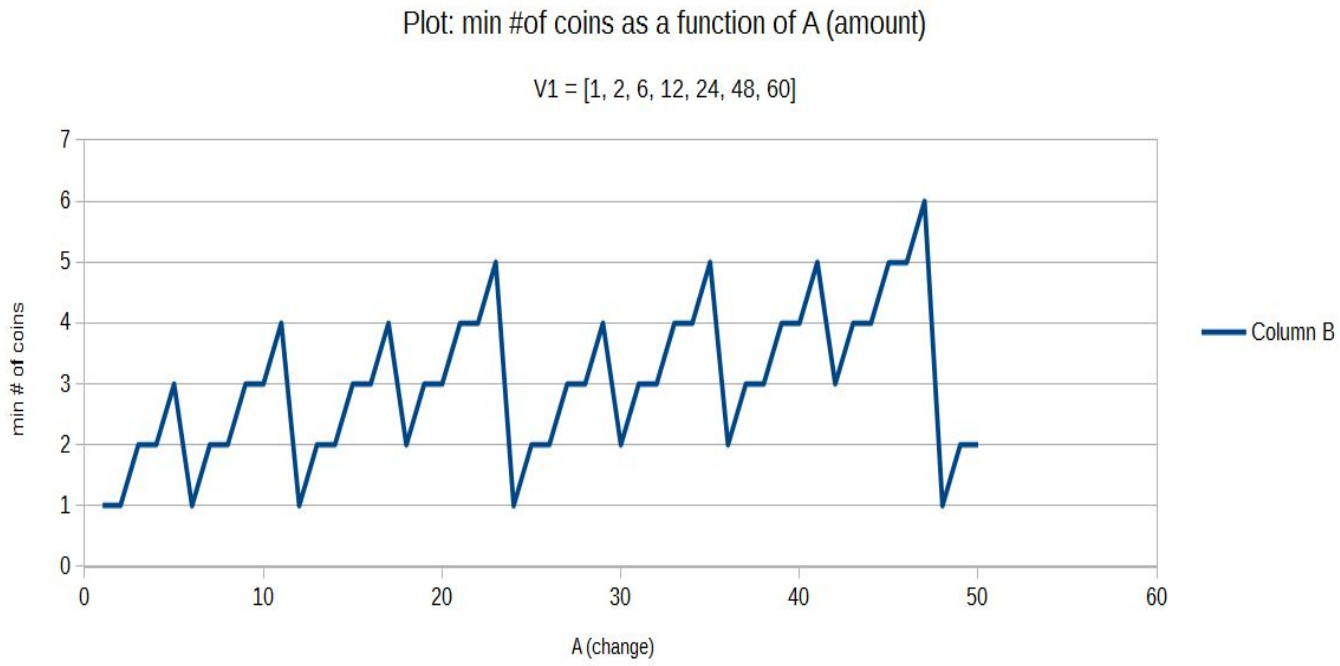
This function will determine the minimum number of coins from V[i] that will add up to the

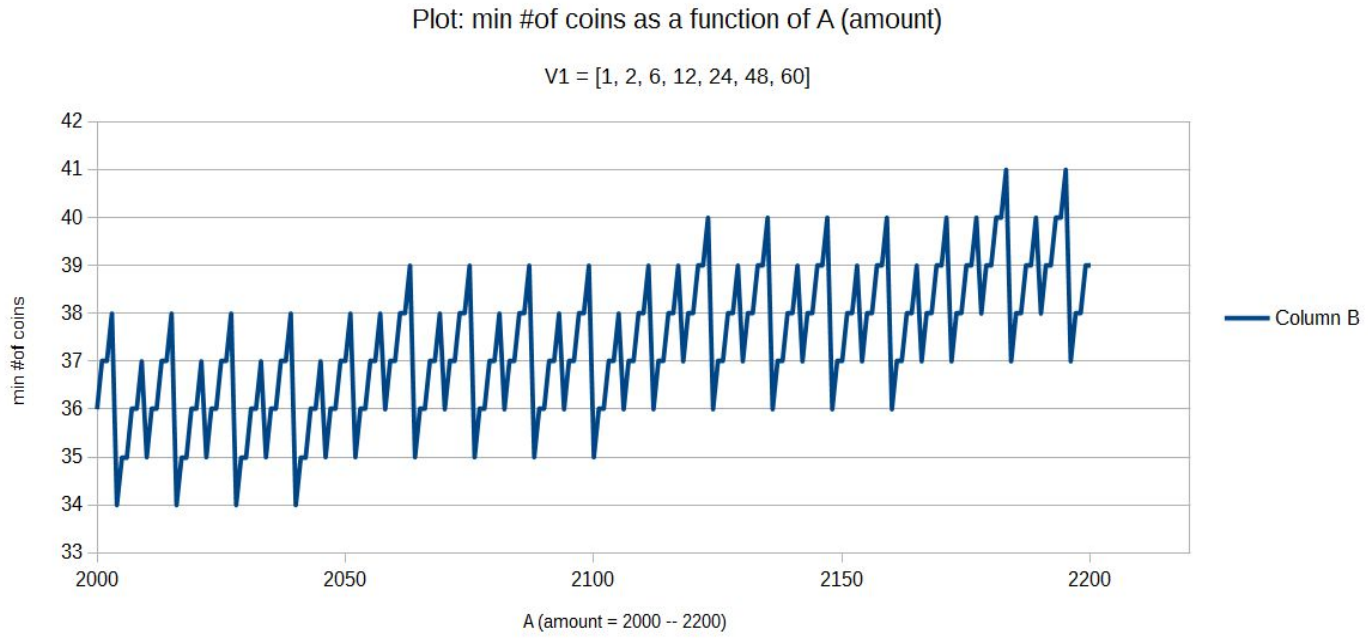
amount v. For every value of v, we determine the minimum number of coins from V[i] to make amount v. Thus we build with a table with index from 0 to v for every T[v]. The first entry, T[0] will be 0 according to our base case. Our approach in changedp() starts checking from v = 1 to determine which denomination coin provides the minimum number of coins for v. For example, a denomination list of V[]={1,5,10,25} would be used as follows to pick a minimum number of coins for any given value v:

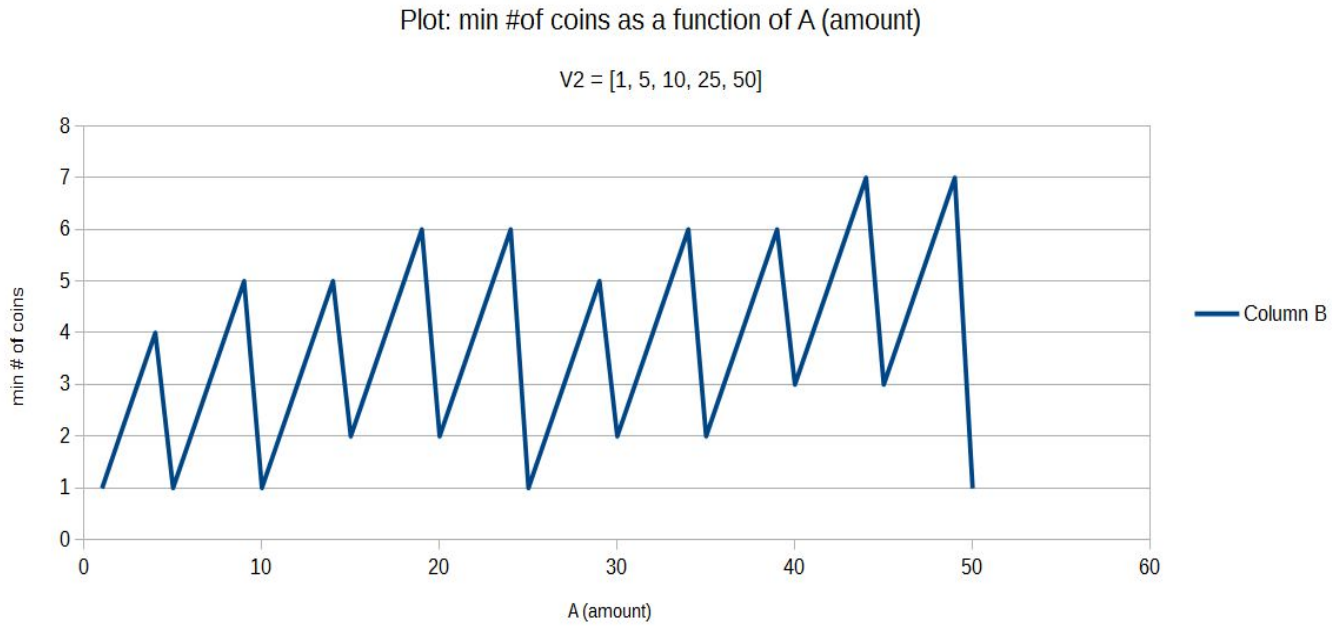
$$T[v] = \min \begin{cases} 1+T(v-1) \\ 1+T(v-5) \\ 1+T(v-10) \\ 1+T(v-25) \end{cases}$$

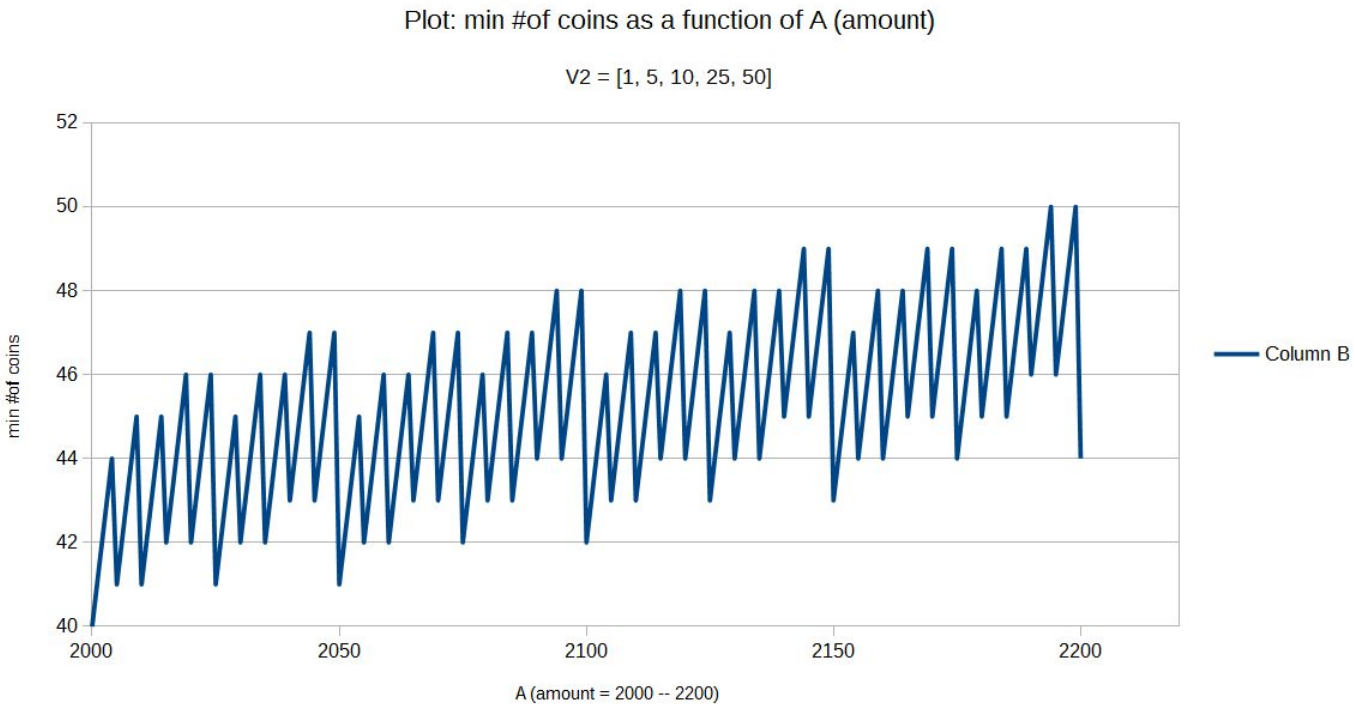
T[v] gets incremented at each stage as an optimal coin value is determined and tallied. Once the table is filled up, the minimum number of coins for any value v would be in the T[v] location. This approach is valid as it incrementally builds from the simplest case of v=1 and finds the optimal number of coins at every stage.

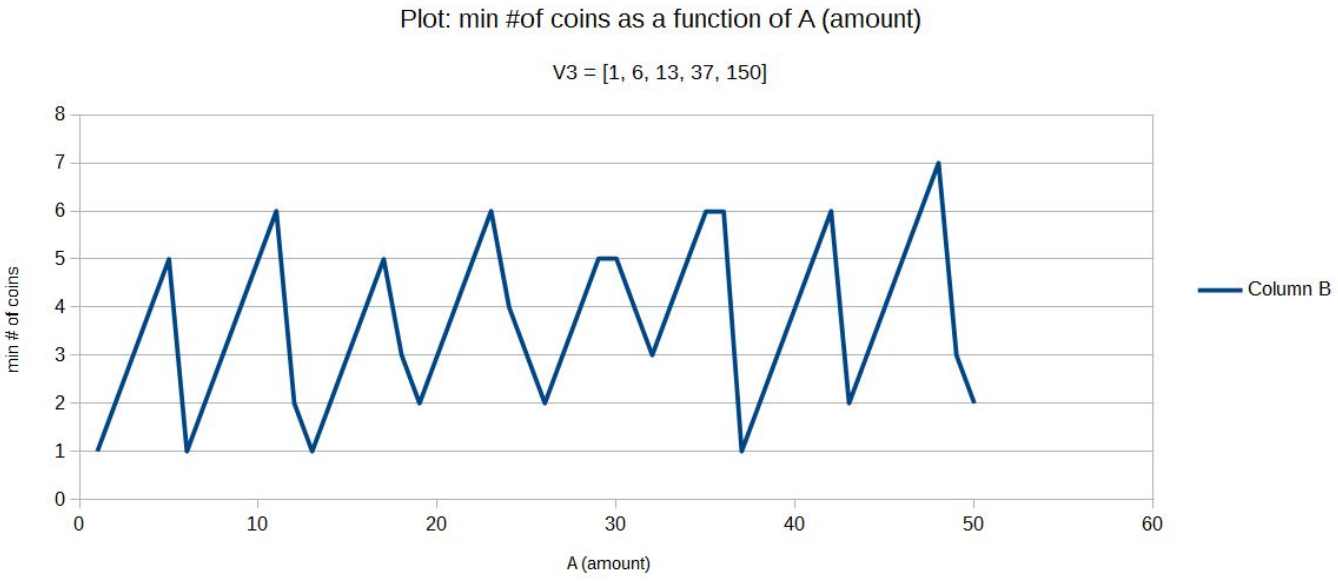
Number of coins as a function of *A*

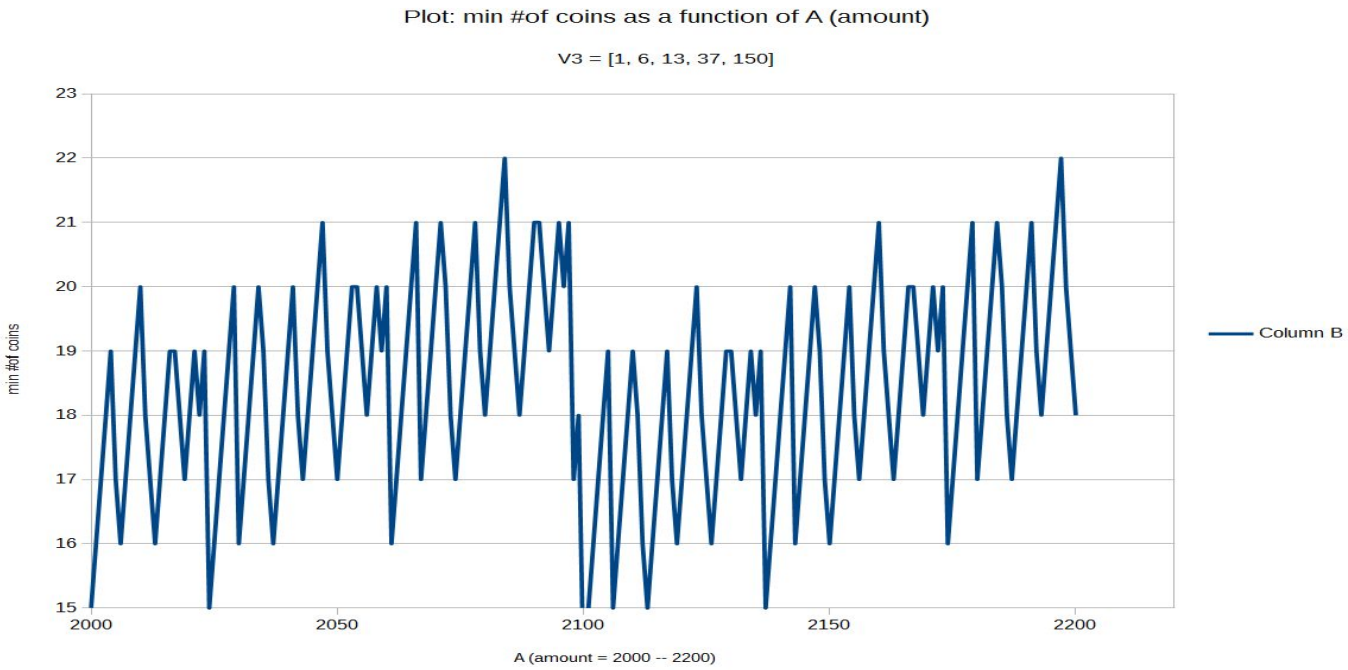



Plot: min #of coins as a function of A (amount)
V1 = [1, 2, 6, 12, 24, 48, 60]
min #of coins
A (amount = 2000 -- 2200)
Column B
42
41
40
39
38
37
36
35
34
33
2000
2050
2100
2150
2200

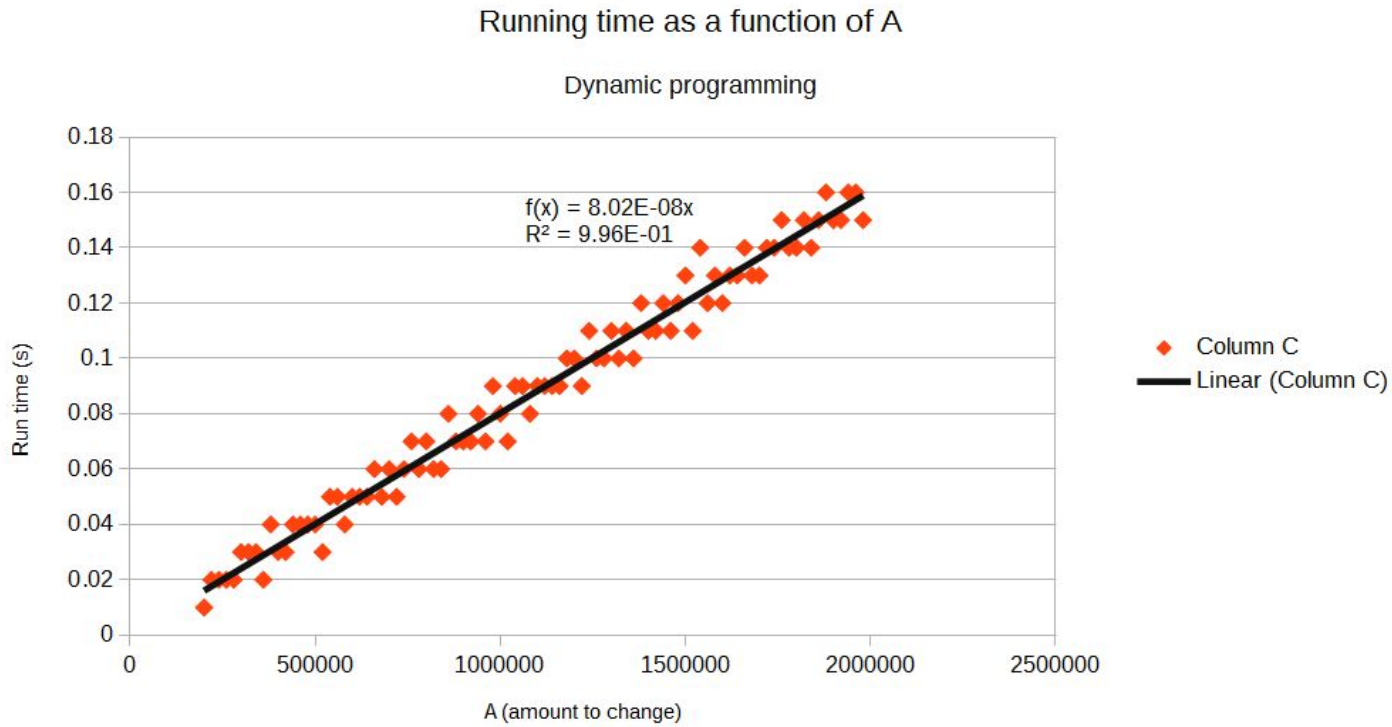

Plot: min #of coins as a function of A (amount)
V2 = [1, 5, 10, 25, 50]
min # of coins
A (amount)
Column B
8
7
6
5
4
3
2
1
0
0
10
20
30
40
50
60


Plot: min #of coins as a function of A (amount)
V2 = [1, 5, 10, 25, 50]
52
50
48
46
44
42
40
min #of coins
2000
2050
2100
2150
2200
A (amount = 2000 -- 2200)
Column B


Plot: min #of coins as a function of A (amount)
V3 = [1, 6, 13, 37, 150]
8
7
6
5
4
3
2
1
0
min # of coins
0
10
20
30
40
50
60
A (amount)
Column B

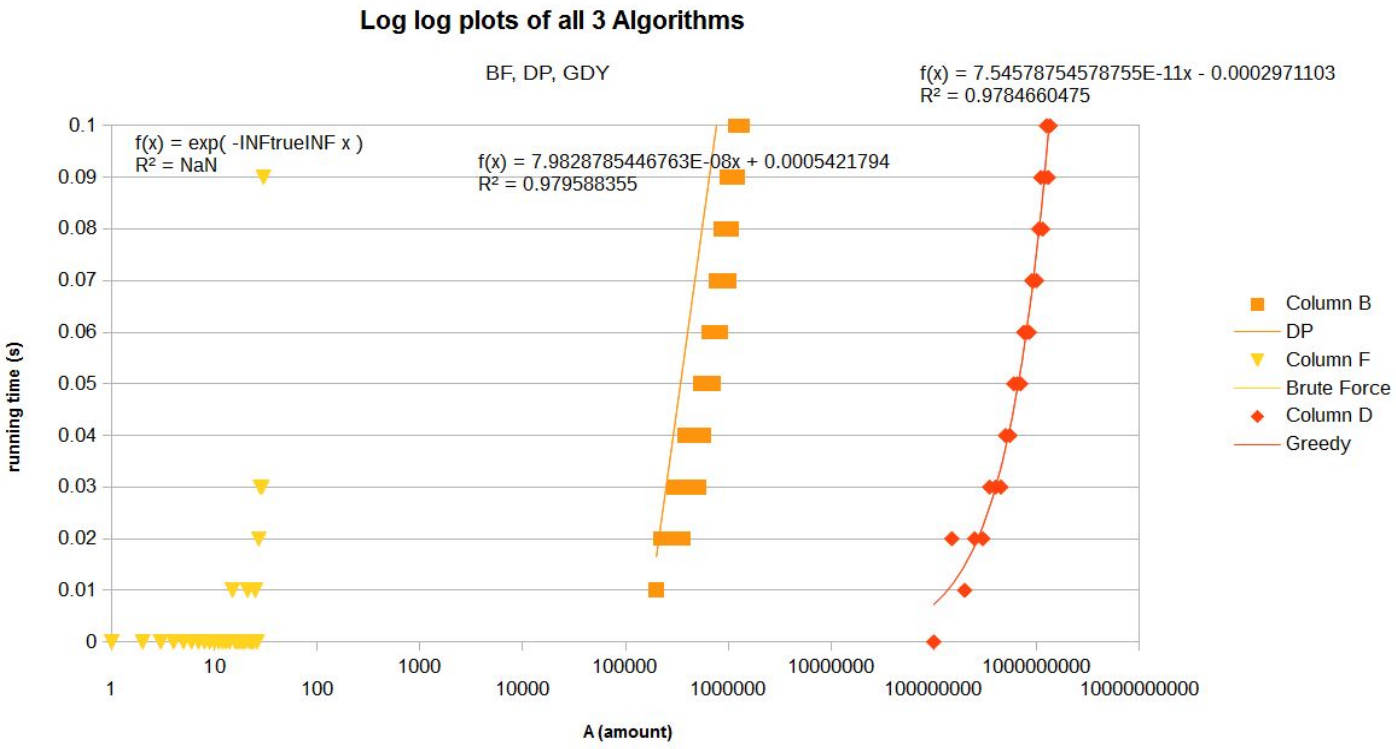


Running time as a function of A



As expected, the theoretical runtime (linear, Θ(nk)) matches the experimental run data collected. *changedp()* execution time data was collected on a large sample size of A (amount to change) with a modest incremental value. The run times for various samples were plotted in the graph and curve fitted to match a linear growth. f(A) = 8.02E-08A, with R^2 = 0.996.

Log-log plot containing runtime data from all 3 algorithms



Dynamic programming vs greedy algorithm on V = [1, 3, 9, 27]

Dynamic programming will always produce an optimal number of coins since it gradually checks for the optimal numbers as amount to change increases. However, I also think greedy algorithm would produce optimal solution every time with this set of numbers as each value divides into the next evenly. Ie Each number is a factor of the next.

Ref: http://people.cs.ksu.edu/~sathish/coinset.pdf

Examples of denominations sets V for which the greedy method is optimal

Based on the proof from the reference material above, where optimal solution with greedy method occur when each value is a factor of the next, the following sets of V would produce optimal values:

V1 = [1, 3, 6, 12]

V2 = [1, 2, 8, 16, 32]

V3 = [1, 5, 10, 20, 40]