CS 325 Project 2: Coin Change Algorithms
Group 26: Aaron Boutin, Edward Francis, Jessica Spokoyny
Project Report

1. Pseudocode and asymptotic running time for each algorithm

The following sections provide pseudocode and theoretical asymptotic running time analysis for three algorithms used to find the minimum coins required to make a given amount of change, given an array of coin values V and an amount of change to make A. The algorithms return both the minimum number of coins needed to make A, and the values of the coins that gives that amount.

Algorithm 1: Divide & Conquer

Pseudocode

```
changeslow(V, K, V.length, CoinValueIndex)

   struct Results result                    // Holds two values, m and C[]

   if (K < 0)                               // Base case that negative values for K can't be
represented
         result.m = INFINITY
         return result

   if (K == 0)                              // Base case when an exact match is found for what is left
         result.C[CoinValueIndex]++    // This adds the coin to its C[]
         result.m = 0
         return result;

   recursiveCallArray[];
   For i to V.length                 // populates array with returned structs that hold min values

         recursiveCallArray[i] = changeslow(V, k - V[i], V.length, i)

   minMval = INFINITY;
   index = 0;
   for For i to V.length                    // This performs the function of finding the min of calls
         if (recursiveCallArray[i].m <= minMval)
               minMval = recursiveCallArray[i].m
               index = i

   recursiveCallArray[index].m++    // the min of the calls coin count is incremented before return
```

```
if (CoinValueIndex >= 0)                    // prevents initial call from adding
      recursiveCallArray[index].C[CoinValueIndex]++

return recursiveCallArray[index]
```

Asymptotic Run-Time

In this Divide & Conquer, each recursive call happens a number of times equal to V.length, the number of denominations. So, the recurrence relation can be described as T(n) = T(n-v1) + T(n-v2) + … + T(n-vj) + C, where C represents a constant amount of work done at each step, to compare the results of coin combinations for sub-problems and choose the best one.

Using the Muster Theorem, and summing up these into aT(n-b) where ‘a’ represents the number of denominations in V and is generally greater than 1 coin, and b varies between V[0] and V[max], based on the value of the denominations. Because a will generally be greater than 1, and a constant amount of work is performed at each level of the recursion, we have a situation like the “case 3” Muster Theorem examples, indicating that the run time complexity will vary in **O(a^n/b)**, that is, running time will be exponential as the input varies.

The only time it is not exponential is when there is one denomination of coin, which will give us T(n) = T(n-V[0]) + C, which is O(n).

Algorithm 2: Greedy

Pseudocode

```
int changegreedy(int *V, int *C, int n, int A) {
        int m = 0; // initialize a total coin counter to zero;
        int tmp = A; // let tmp hold the value of the amount for which change is sought
        int i = V.size - 1; //let i represent the index of the largest value coin)
        while (tmp > 0 and i >= 0) { //perform the follow steps, so long as tmp has not
been reduced to zero and there is still a coin to process:
                if (tmp >= V[i]) { //for the current coin, starting with the largest, if the coin
value is greater than or equal to the remaining amount, subtract it from amount, add one
of that coin to an array tracking coin values, and add one to the min coin count.
                        tmp = tmp - V[i];
                        C[i]++;
                        m++;
                }
                else { //otherwise, move to the next smaller coin
                        i--;
```

```
            }
        }

            return m;} //return the total amount of coins needed, and the coin combo
    (possibly by updating an array parameter)
```

Asymptotic run time

In this function, all operations besides the while loop run in constant time O(1). The number of times the while loop must be executed depends on the variable n (the amount of change that must be made) and 'V.size()' the length of the array of coin denominations.

For example, if we have V = [1, 5, 10, 25, 50] and amount = 50, the loop will run only once and have a complexity of O(1). This is the best case. In the worst case, however, we could have V = [1, 50, 100, 250, 500] and amount = 49. The while loop would need to execute 5 times to find a coin value less than 49, and then 49 times to make change with the 1 coin.

This gives us a run time complexity of O(n + V.size()) + O(1), which is equivalent to O(n + V.size()), which supposing a fixed number of coin denominations would typically be O(n).

Algorithm 3: Dynamic Programming

Pseudocode

Function will take an array of coin values and a target amount as parameters, and will return (or print out) an optimal combination of values that minimizes the number of coins required to make the target amount. It will take a dynamic programming approach, by building up a table of solutions for sub problems from the bottom up, and then determining the result based on this table.

Define a struct "solution" with the following member variables:

```
    Int coins;
    String values;
    Bool found;

changedp(vector<int> values, int amount){
    Create a table of solution structs, of size "amount +1"
        Initialize each solution's "found" status to 0
    Establish the base case by setting the solution for sTable[0]
        Set coins = 0
        Set found = 1
        Set the values string to "" (empty)
```

For amount k, from 1…. N (user input amount)
Set up variables to track minCoins and minCoinsValues
For each value of coin in the values array
Find the amount index = k - value of coin
If index >= 0 and sTable[index].found == 1
If sTable[index].coins < minCoins
Update minCoins and minCoinsValues variables for the amount
Using the minCoins and minCoinsValues, update the sTable entry for amount k, and set found status to 1.
Return minCoins and minCoinsValues for amount = N, based on sTable[n].coins and sTable[n].values.
}

Asymptotic run time

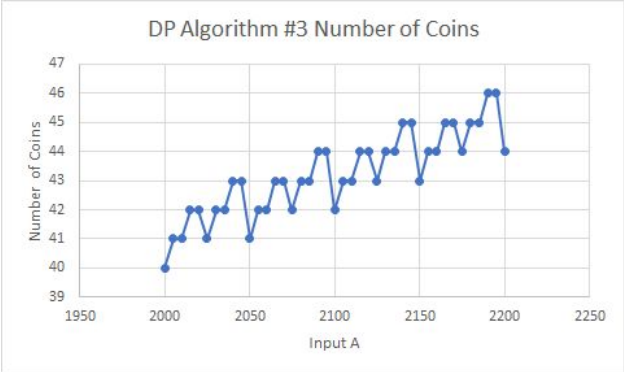
Running time is dominated by the process of filling in each value in the solutions table for subproblems of amounts from 0 up to A. The amount of work required to fill in each value is proportional to the number of coin denominations, because the solution for each amount A - V[i] must be checked to find the best subproblem solution to use to build the current solutions Table entry. Run time is therefore O(V.size() * n), where V.size() represents the number of denominations and n is the amount A of change to make, leading in most cases with a relatively small number of values in V to O(n) performance. In building up the table of optimum coin totals and values for each amount 0….n, a maximum of #coin values comparisons are made for each n, to check entries already listed earlier the lookup table, before selecting the best entry with optimally minimum value and adding 1 new coin to the total. After the table has been constructed from the bottom up, the algorithm simply returns the optimum value for amount A by indexing into the table, which does not affect the asymptotic running time.

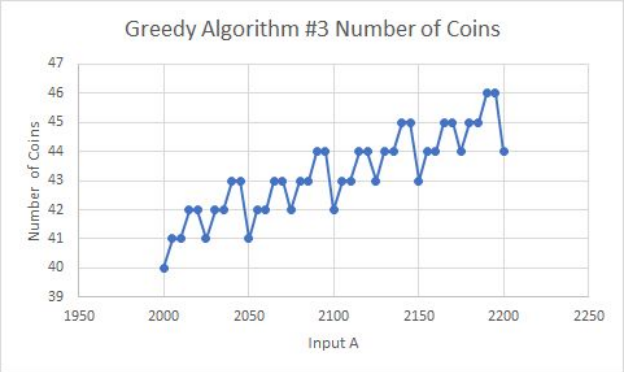
2. Describe how we filled in the dynamic programming table in changedp

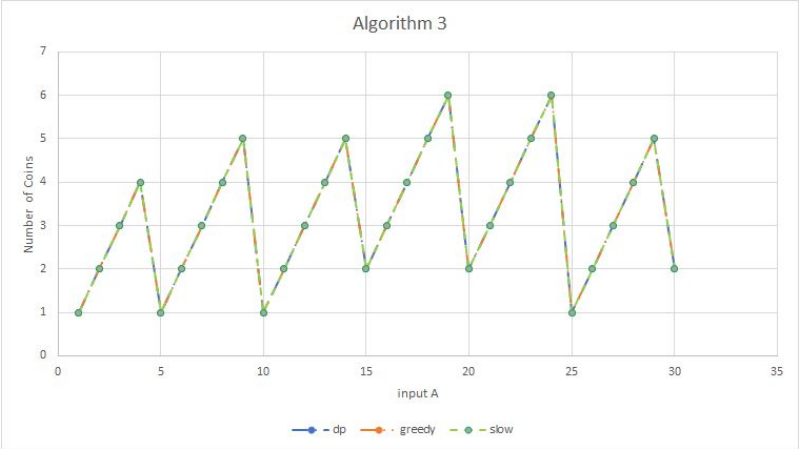
To fill in the dynamic programming table in changedp, we took a bottom-up approach. We started by filling in the optimum number of coins and coin combination for A = 0, which was simply 0 coins and an empty set. We then took an approach that worked upwards from i = 1 through i = A to fill in the optimum number of coins for each i, by first finding the optimum number of coins to make A - V[j] amount, for each coin j in the coin values array, then selecting the best (least number of coins) solution among these candidates, and adding one coin V[j] to the solution for i. If no solution to make A - V[j] coins could be found for any j, then we marked that amount i as an impossible sum and excluded it from consideration when finding optimum coin values for larger amounts.

This approach is effective because it is based in the fact that an optimum solution for the overall problem amount A is built up from optimal solutions to smaller subproblems of size less than A.

3. Number of coins as a function of A: V = [1, 5, 10, 25, 50], A in [2010, 2015… 2200]



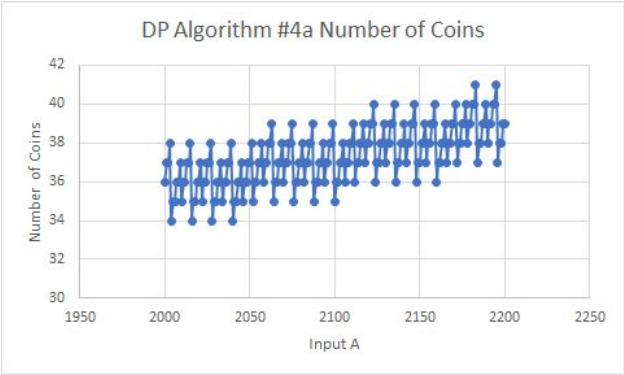


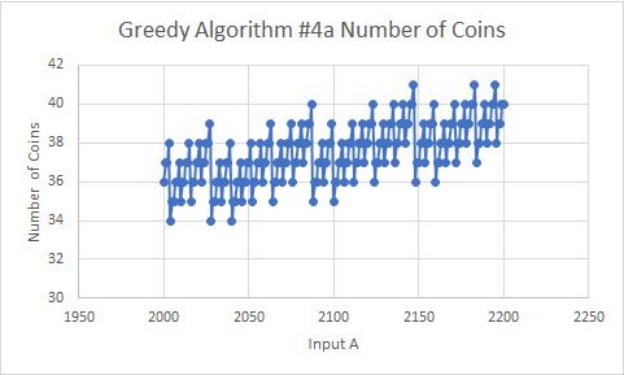


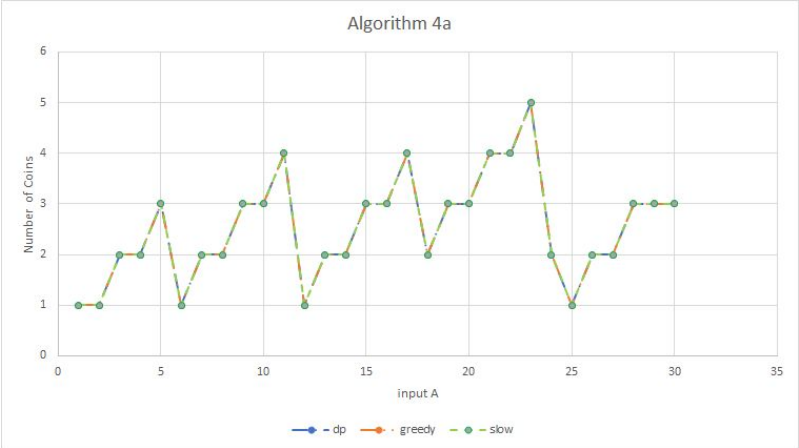
The graphs above indicate that all three algorithms return the same results for number of coins required to make an amount A, for both large and small values of A.

In the first two graphs, the dynamic programming algorithm and the greedy algorithm appear to return identical results for A = 2000, 2001, …. 2200, where V = [1, 5, 10, 25, 50]. In the final graph data for all three algorithms is overlaid on the same set of axis, indicating that the results for all three algorithms are the same for amounts A = 1… 30.

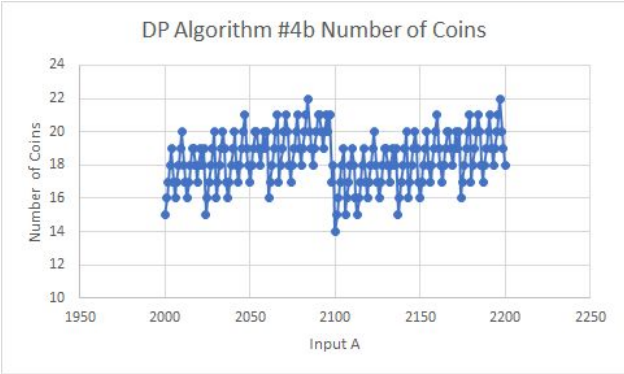
4. Number of coins as a function of A: V1 = [1, 2, 6, 12, 24, 48, 60], V2 = [1, 6, 13, 37, 150], A in [2000, 2001… 2200]

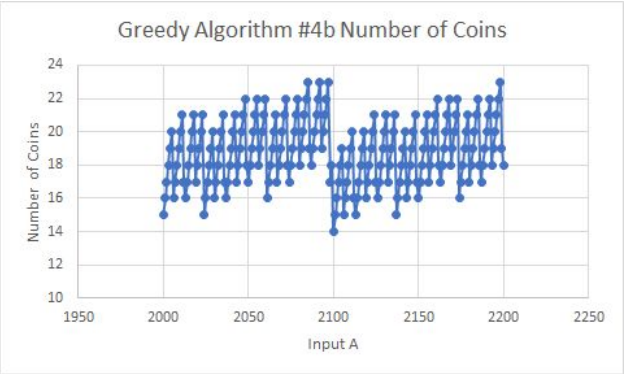


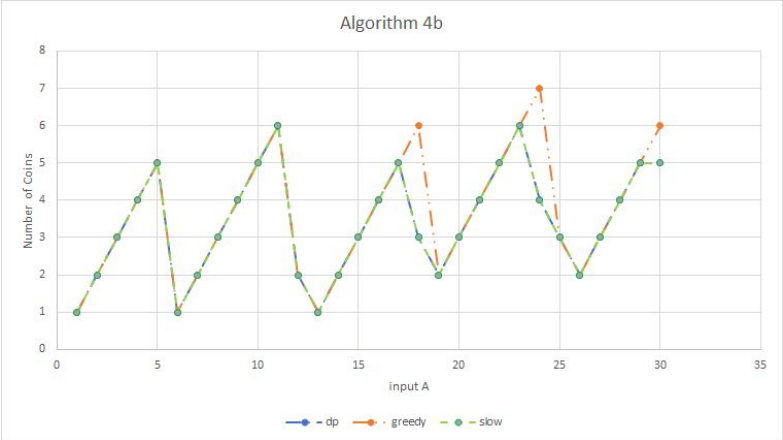




The above graphs are based on V1 = [1, 2, 6, 12, 24, 48, 60]





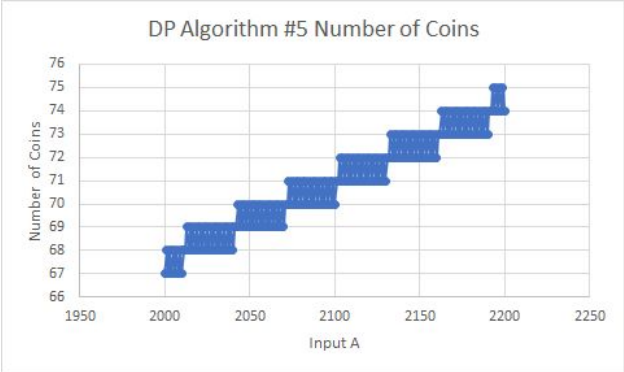


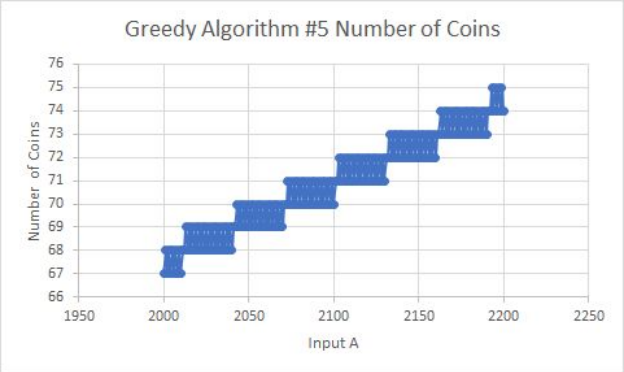
The above graphs are based on V2 = [1, 6, 13, 37, 150]

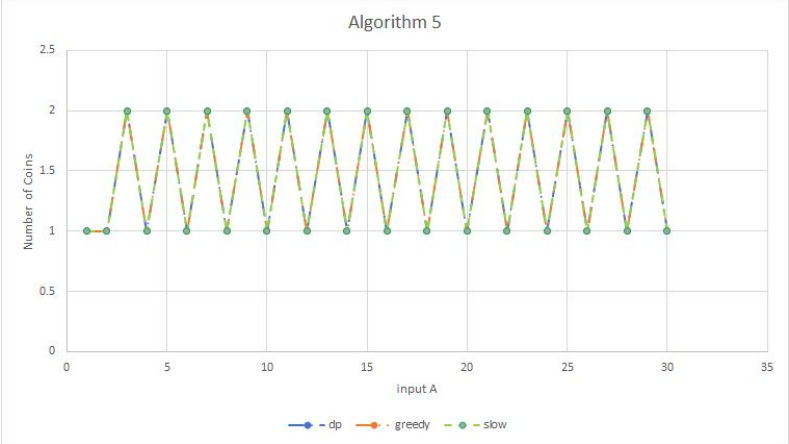
In terms of number of coins, the greedy algorithm and the dynamic programming algorithm appear to follow a similar pattern of growth in number of coins required, as the input size A increases. Both algorithms follow a stair-stepping growth pattern, where each algorithm's coin requirements increase fairly quickly and then drop down periodically, while maintaining an overall positive growth trend, due to the effect of increasing values of coins replacing several smaller coins in the optimum combination.

The does appear to be some difference between the DP algorithm's values for max number of coins and the greed algorithm's values for maximum number of coins, for the second data set, where V2 = [1, 6, 13, 37, 150]. The greedy algorithm appears to result in solutions with greater number of coins in several instances, while the dynamic programming algorithm generally has a lower coin count. This is particularly clear from the combined graph outlined in red, comparing the number of coins identified by each algorithm, for lower amounts A.

5. Coins as a function of A: V = [1,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30], A in [2000, 2001… 2200]



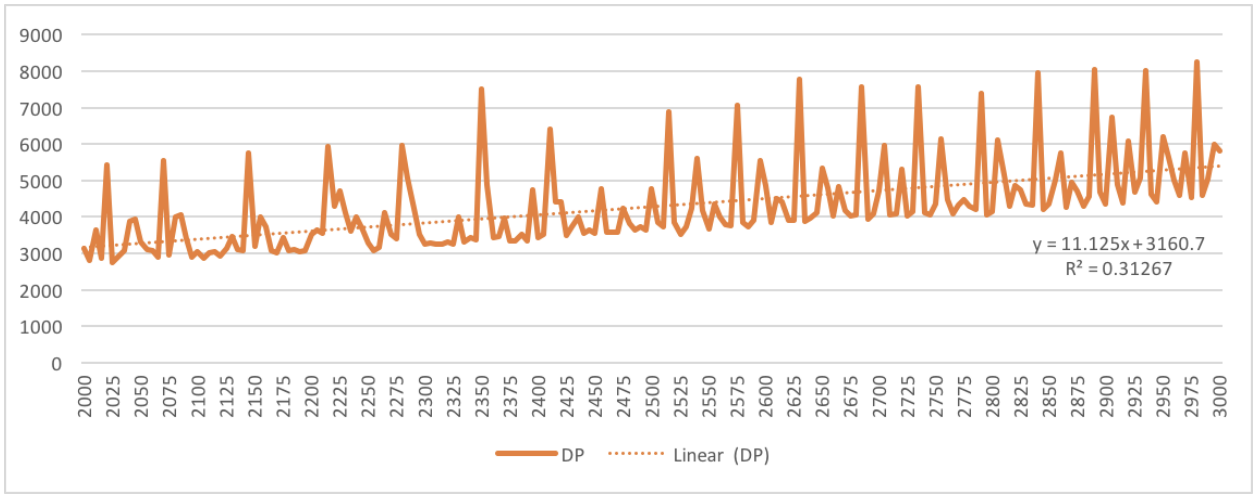


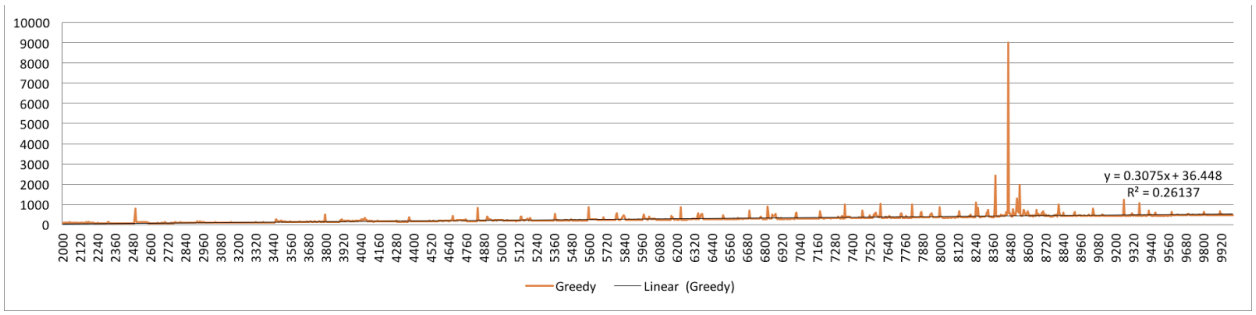


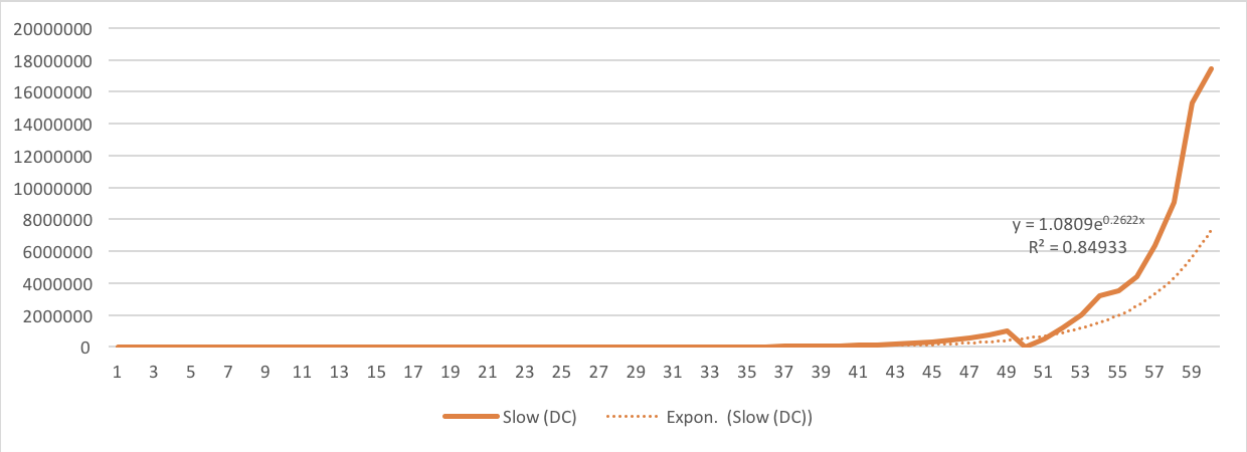
Letting V = [1,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30] and calculating the minimum number of coins for A = [2000, 2001,... 2200], the dynamic programming algorithm and the greedy algorithm appear to produce identical results in terms of their optimum number of coins as a function of A. For smaller A between 1 and 30, all three algorithms appear to return the same number of coins.

6. For the above situations, determine the running times by fitting trends lines to the data or analyzing the log-log plot, and plot running time as a function of A

a. V = [1, 5, 10, 25, 50], A in [2010, 2015… 2200]





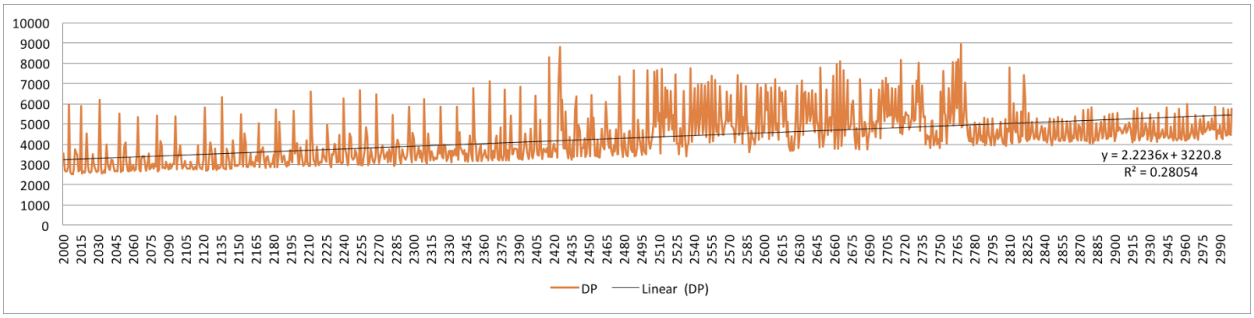


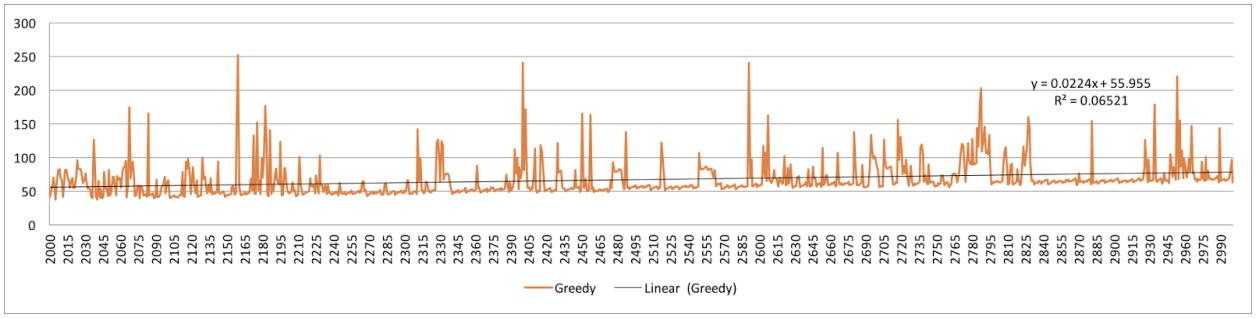
The above charts show running time data for the dynamic programming, greedy, and slow (divide and conquer) algorithms, with V = [1, 5, 10, 25, 50]. The range of values for A was modified to provide more basis for curve fitting and asymptotic running time analysis. The range A[2000… 3000] was used for the DP algorithm, the range A[2000… 10,000] was used for the greedy algorithm, and the range A[0… 60] was used for the DC algorithm. Times are shown in milliseconds.

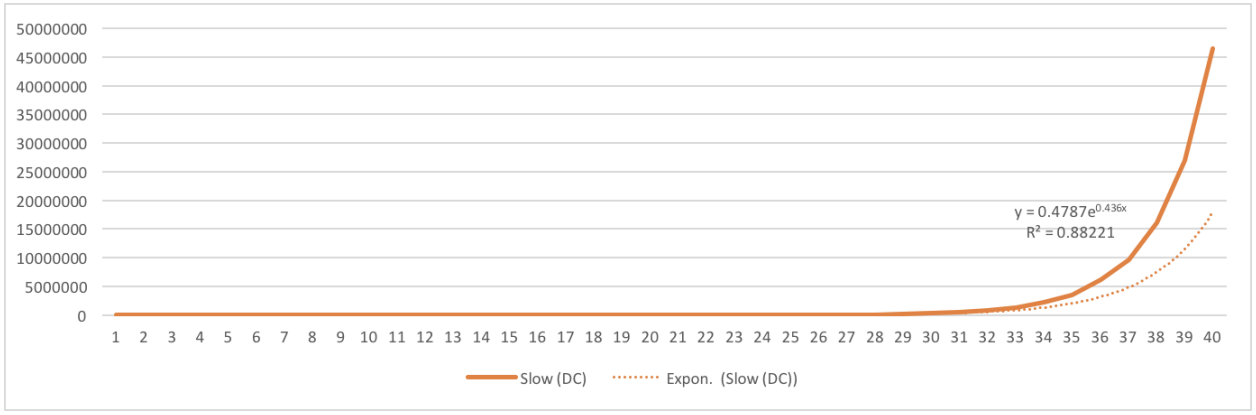
Linear curves were the best fit for the DP and greedy algorithm run time data, indicating that asymptotic running time for both algorithms is O(n). An exponential curve was the best fit for the slow (divide and conquer) algorithm's running time data, indicating that this algorithm is O(2^n).

b. V1 = [1, 2, 6, 12, 24, 48, 60], V2 = [1, 6, 13, 37, 150]

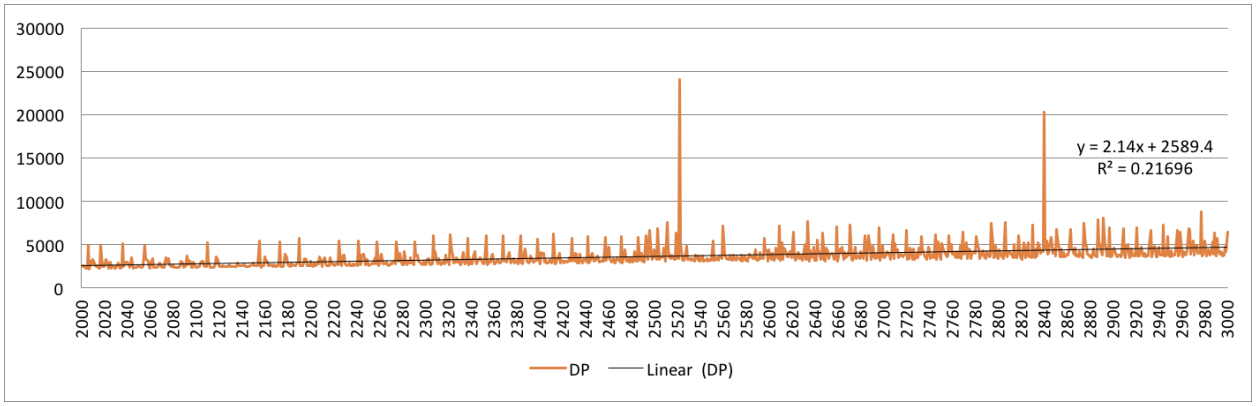
With coin values V1 = [1, 2, 6, 12, 24, 48, 60], running times and trend lines were as shown in the following graphs. Times are shown in milliseconds.

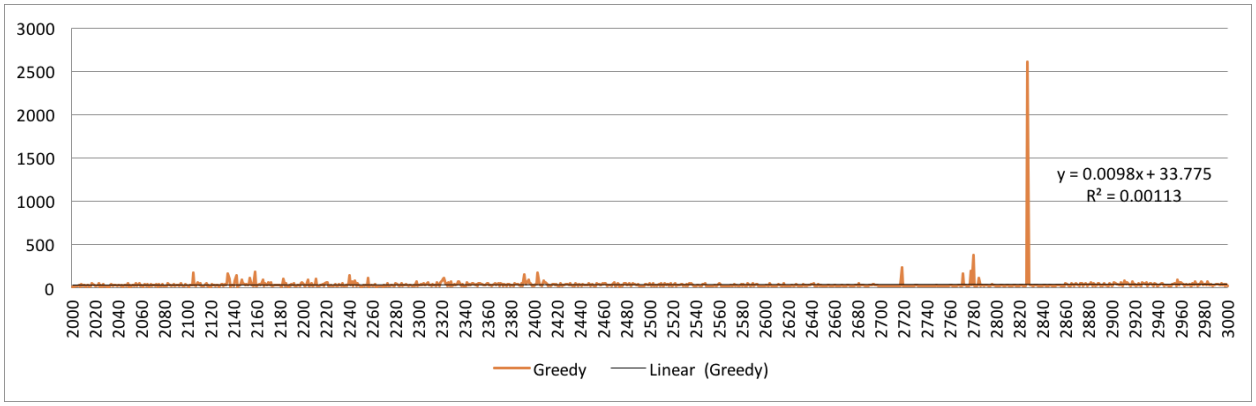


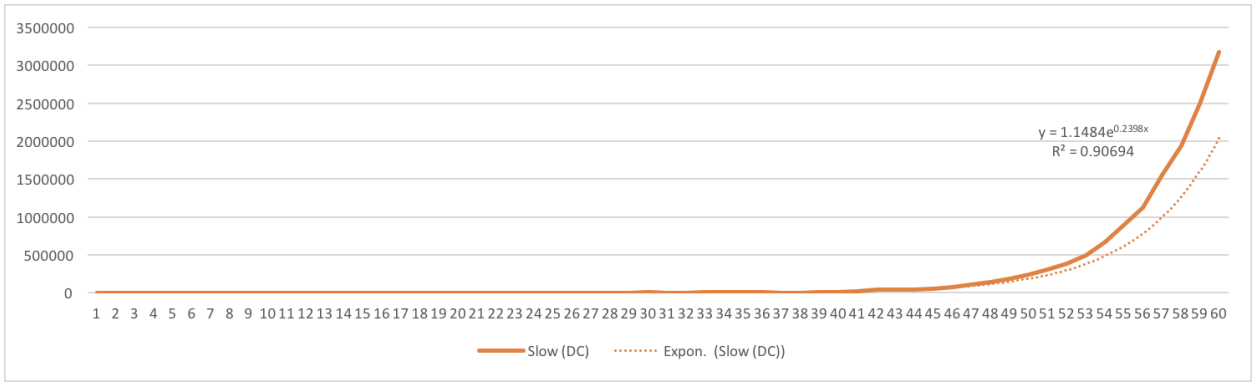




With coin values V2 = [1, 6, 13, 37, 150], running times and trend lines were as shown in the following graphs. Times are shown in milliseconds.



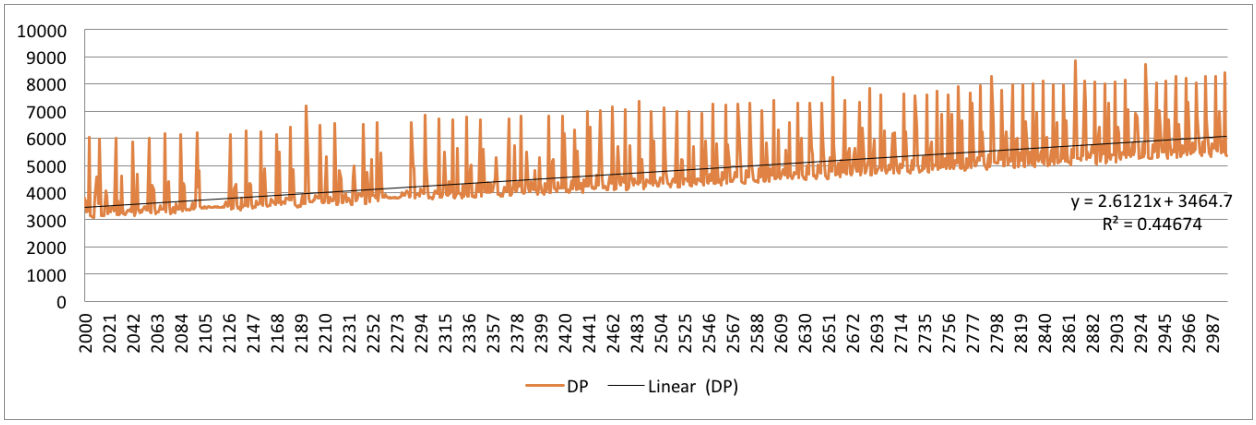


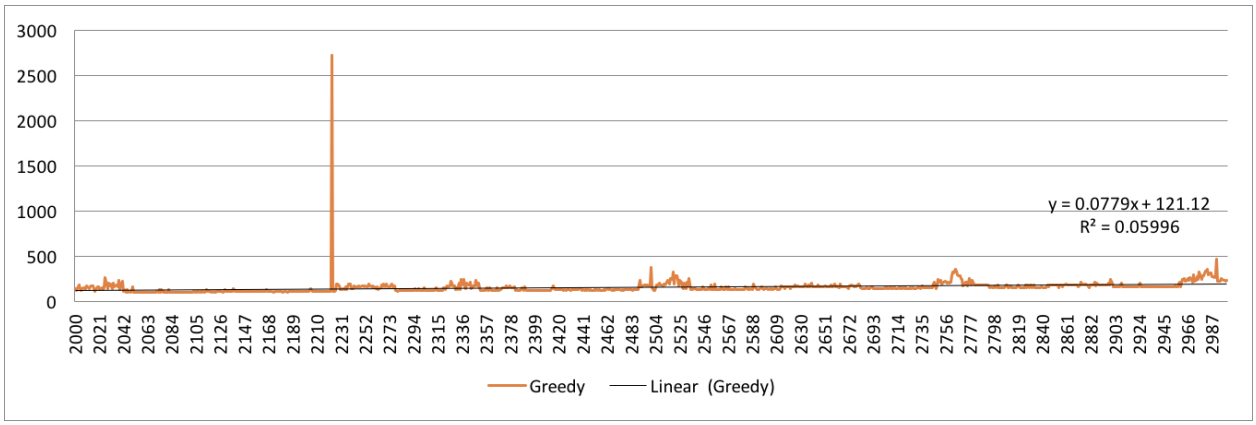


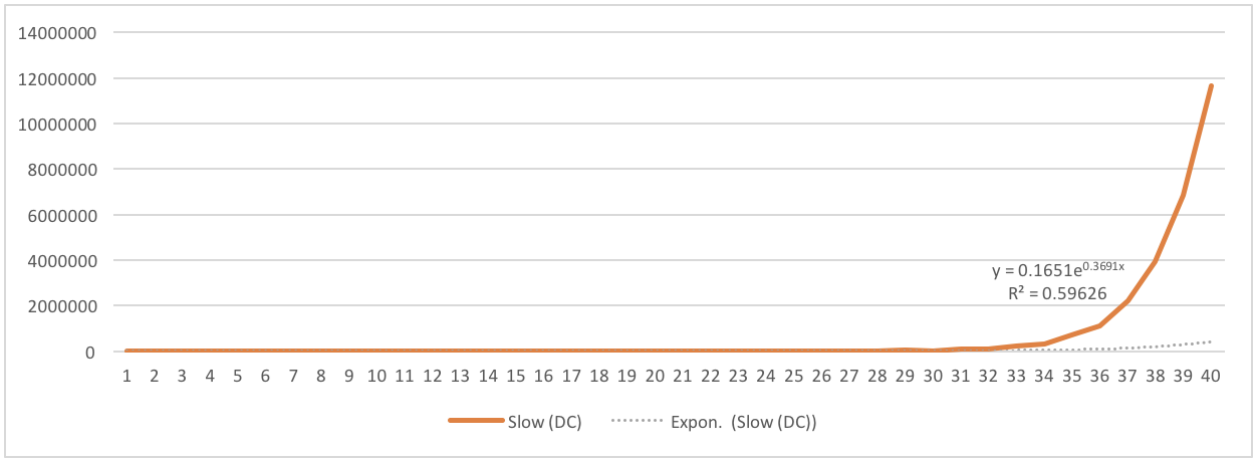
For both sets of input values V1 = [1, 2, 6, 12, 24, 48, 60] and V2 = [1, 6, 13, 37, 150], the best-fit curves were again linear for the DP and greedy algorithms, and again exponential for the DC algorithm, supporting the conclusion that DP and greedy are O(n) while DC is O(2^n).

c. V = [1,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30]

The following graphs show running time and trends as a function of input amount size A, with coin values V = [1,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30]. Running time is measured in microseconds.







From the above graphs, running time for the DP and greedy algorithms appears to grow linearly with increasing A for this coin set, while the DC algorithm grows exponentially, indicating big-O complexity of O(n), O(n), and O(2^n) for these three algorithms.

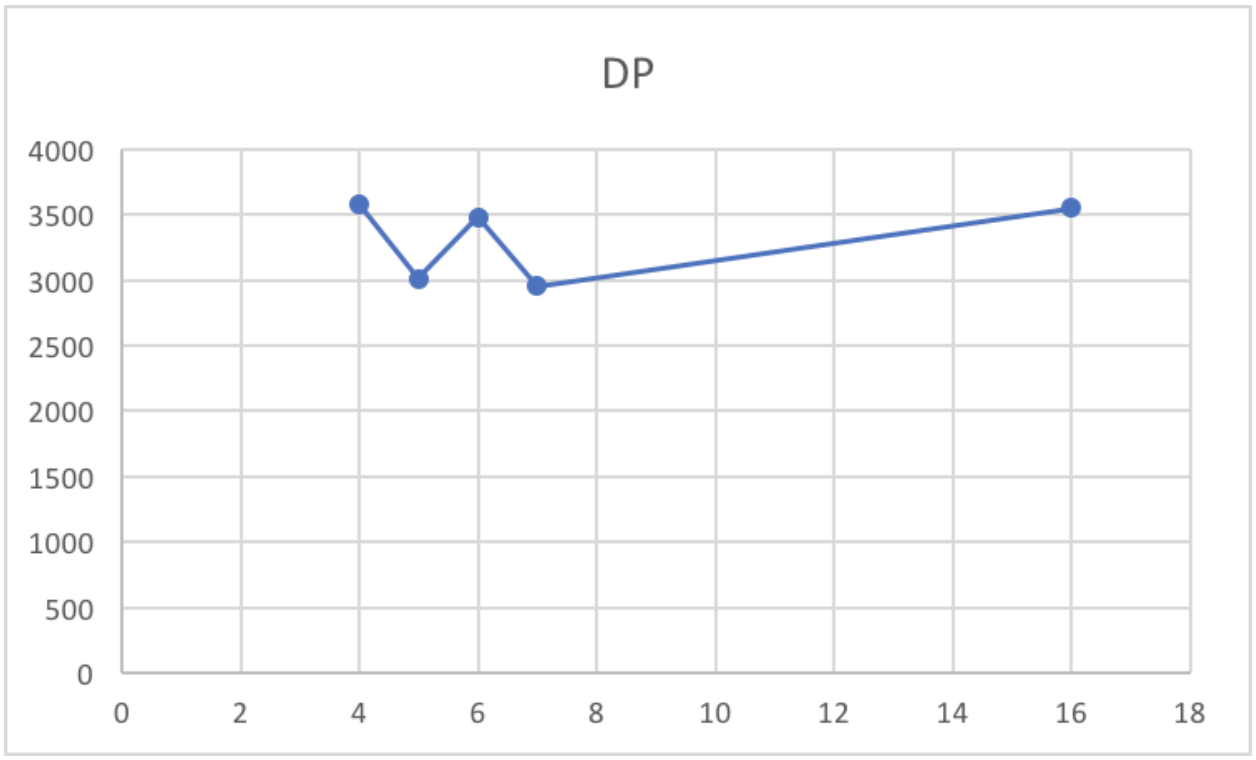
7. Plot running time as a function of number of denominations

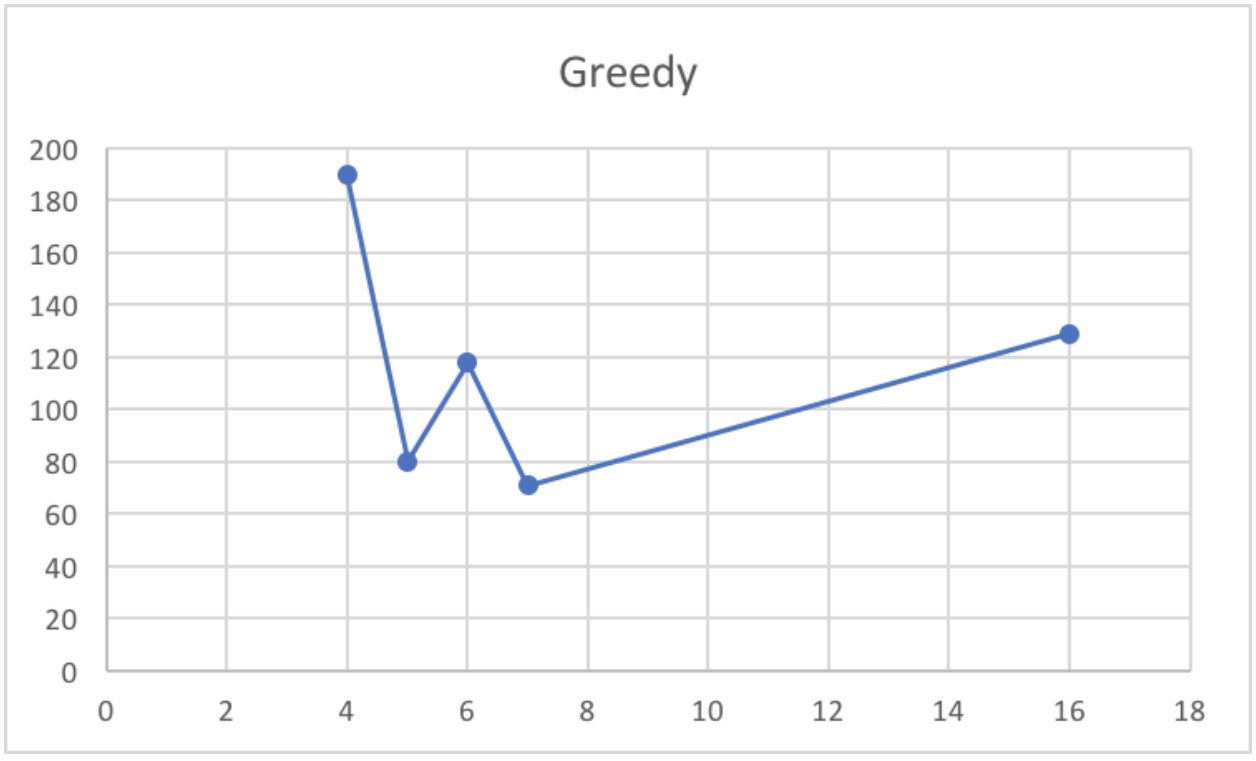
To plot running time as a function of number of denominations, we ran each algorithm with several sets of coins V and a constant amount size A, and collected running time data. For the dynamic programming and greedy algorithms we used A = 2000, and for the divide and conquer algorithm we used A = 30, due to its slower running time. All times were recorded in microseconds.

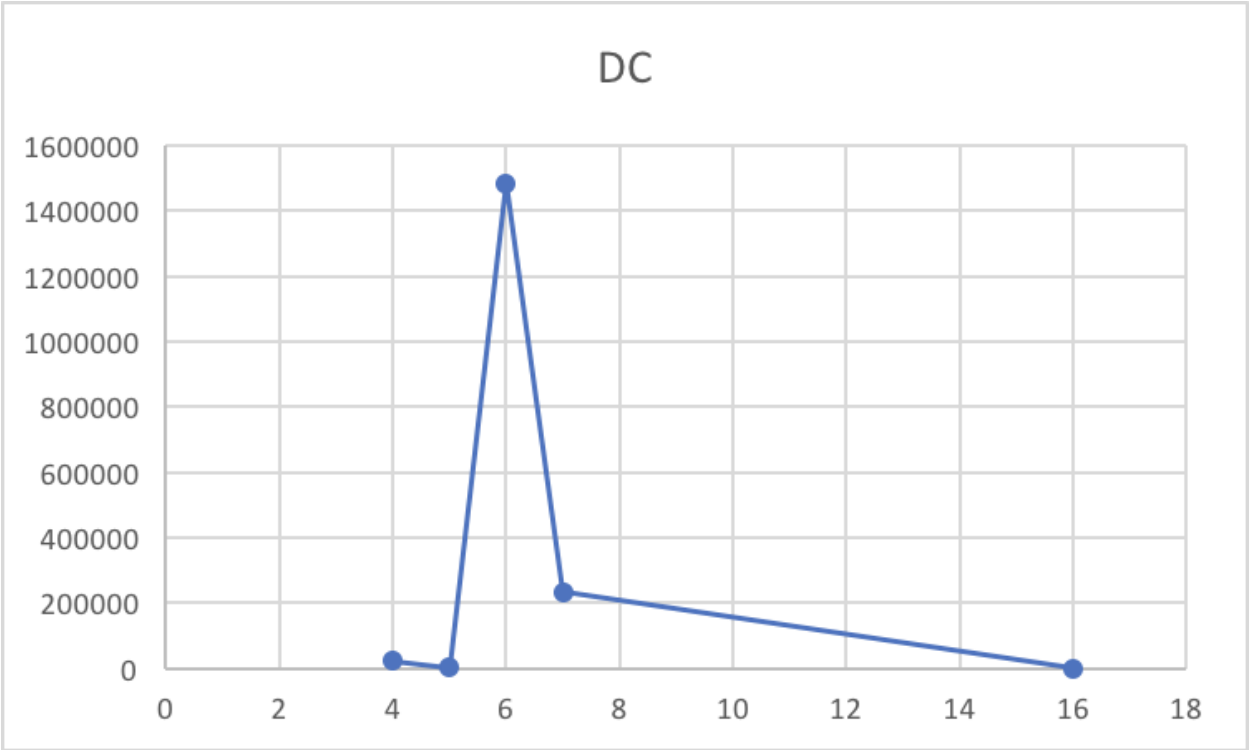
We used input sets V as shown in the following table.

Set V	Number of denominations N
[1, 3, 9, 27]	4
[1, 5, 10, 25, 50]	5
[1, 2, 4, 8, 16, 32]	6
[1, 2, 6, 12, 24, 48, 60]	7
[1,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30]	16

Graphs showing experimental results are shown below.


DP
4000
3500
3000
2500
2000
1500
1000
500
0
0
2
4
6
8
10
12
14
16
18


Greedy
200
180
160
140
120
100
80
60
40
20
0
0
2
4
6
8
10
12
14
16
18

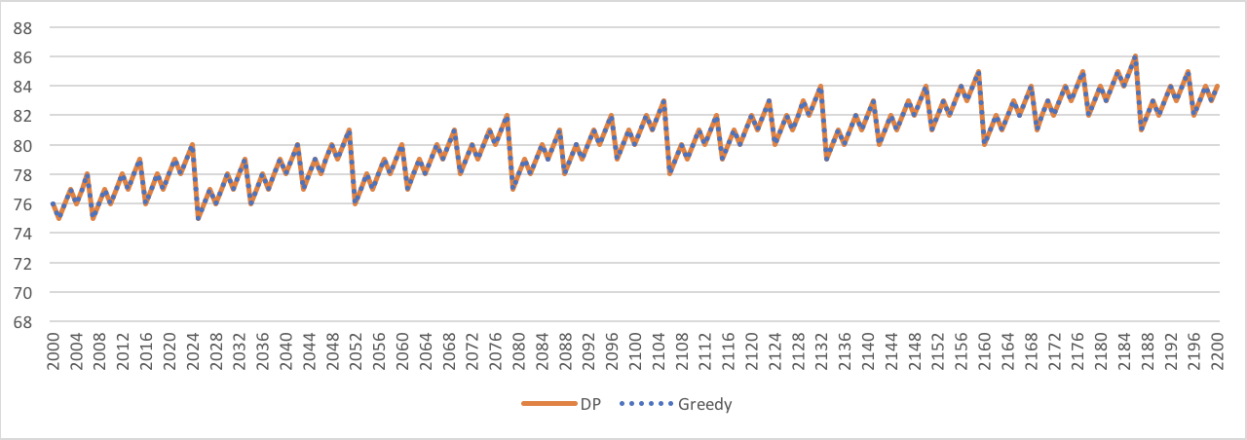


The experimental data do not clearly demonstrate a correlation between number of coin denominations in V and algorithmic running time. This may be attributable in part to weaknesses in our experimental design. For example, in the case of the divide and conquer algorithm, in order to gather running time data in a time-efficient manner, we were forced to use a relatively small value A for our experimental analysis. This necessarily excluded certain larger denomination coins from consideration, which limited the growth of running time as V.size() was increased.

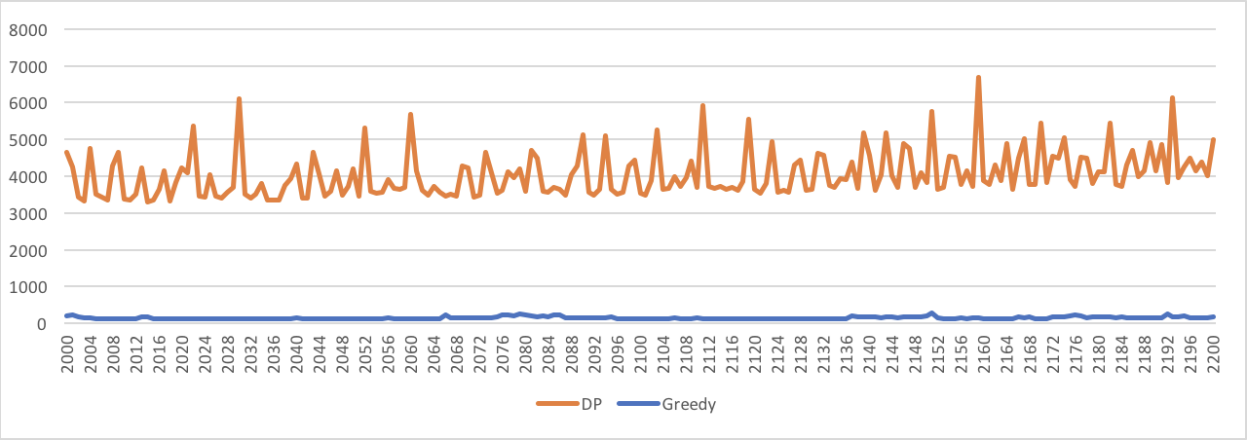
For the dynamic programming and greedy algorithms, based on our theoretical analysis we would expect the running time for these algorithms to gradually increase as the number of denominations increases. However, the range of denomination counts tested was too small to make this growth clear, compared to cyclical oscillations in running time that are also apparent in the time-as-a-function-of-A graphs. Using a larger range of denomination counts would have been preferable, but our approach to generating valid sets of denominations compatible with both the dynamic programming and greedy algorithms (discussed in response to question 9 below) would have imposed a low limit on the number of sets of denominations that we could produce, without quickly exceeding even very large A with the size of our largest coin.

8. Suppose coins have values V = [1, 3, 9, 27]. How do dynamic programming and greedy approaches compare?

The greedy approach should be faster for these values of V, while still finding the same optimum coin solutions produced by the dynamic programming algorithm. This expectation is confirmed by the following charts, which show that the number of coins found by both algorithms is the same for amounts A = [2000, 2001… 2200], and that running time is consistently greater for the dynamic programming approach over the same interval.



Coins as a function of A, dynamic programming and greedy algorithms



Running time as a function of A, dynamic programming and greedy algorithms

9. Give additional examples of denominations sets V for which the greedy method is optimal. Why does the greedy method work in these cases?

Other denominations sets V1 = [1, 5, 25, 125], V2 = [1, 2, 4, 8, 16, 32], and V3 = [1, 4, 16, 64, 256] would also be well-suited to a greedy algorithm approach.

It appears to be sufficient, although not necessary, that the denominations be successive powers 0…. k of some base integer n, so that the coin values begin with 1 (equal to n^0) and range through n^k. While there are other coin combinations that are compatible with a greedy algorithm for making change (such as the US coin system with values 1, 5, 10, 25, 50), the "successive powers of a base" approach is effective, which is a fact that allows us to quickly find the base-n representation of a value and make efficient use of a positional notation system.

To see why the "successive powers of a base" approach finds an optimal minimum-coin solution using the greedy approach, it could be helpful to consider a counterexample. That is, suppose that there was an optimal solution to a case where the greedy approach would not work. This would mean that there was some index i in the array of coin values, where the number of coins chosen for i was not at a maximum, and where the number of coins chosen for some index j < i, was as a result greater than it would be using the greedy approach. Given this situation, it would be possible to combine values for the "excess coins" of value less than i, and to replace these coins with a coin of value indexed i. But, this would result in a solution with a smaller number of coins, because multiple smaller coins will be needed to create the sum at index i. This contradiction shows the impossibility of a "successive powers of a base" set of coin denominations for which the greedy algorithm approach does not work.

We also considered the possibility that the greedy method works in these cases because they meet the criteria: $c_i > \sum_{j=1}^{i-1} c_j$

For example, when V1 = [1, 5, 25, 125],
When i = 2, $c_i = 5 > (c_1) = 1$
When i = 3, $c_i = 25 > (c_1 + c_2) = (1 + 5) = 6$
When i = 4, $c_i = 125 > (c_1 + c_2 + c_3) = (1 + 5 + 25) = 31$

another example if when V2 = [1, 2, 4, 8, 16, 32],
When i = 2, $c_i = 2 > (c_1) = 1$
When i = 3, $c_i = 4 > (c_1 + c_2) = (1 + 2) = 3$
When i = 4, $c_i = 8 > (c_1 + c_2 + c_3) = (1 + 2 + 4) = 7$

When i = 5, $c_i = 16 > (c_1 + c_2 + c_3 + c_4) = (1 + 2 + 4 + 8) = 15$
When i = 6, $c_i = 32 > (c_1 + c_2 + c_3 + c_4 + c_5) = (1 + 2 + 4 + 8 + 16) = 31$

However, we found that this is not a completely valid test for the applicability of the greedy method, because there are cases that satisfy this principle, for which the greedy algorithm is not effective. For example, we saw previously that the set of coin denominations V = [1, 6, 13, 37, 150] does not work with the greedy algorithm, although it does meet the criteria $c_i > \sum_{j=1}^{i-1} c_j$.

For example, for this set V, when A is 18, the greedy algorithm will indicate that 6 coins are required to make change (13 and 5 ones), while in fact only 3 coins are necessary (3 sixes).