Group 11
Charles Bennett
Conrad Lewin
Vlad Predovic

CS 325 Project 2: Coin Change

1. Theoretical Runtime Analysis

Brute-Force Algorithm

Pseudocode:

```
Function change_slow(list_of_coins, current_amt)
        coin_tally = an array of size equal to list_of_coins initialized to 0
        coin_tally[0] = current_amt
        For coin_value in list_of_coins,
                if the coin_value is less than current_amt
                        compare_tally = change_slow(list_of_coins, current_amt - coin_value)
                        compare_tally[index of coin_value in list_of_coins] += 1

                        If the sum of all values in array compare_tally< sum coin_tally
                                coin_tally = compare_tally
        return coin_tally
```

Asymptotic Analysis

This algorithm solves the coin-change problem by trying every possible combination of coins. The function has two inputs, a list of coin values and the amount to make from the coins. An array representing the coin count, where each index corresponds to the coin value in the array list_of_coins, is initialized to 0. The first value of this list is set equal to the current_amt because this function assumes that the first coin denomination will always be 1 cent. For each coin in list_of_coins if the coin_value is less than change, the function calls itself to search for the same amount minus the value of that coin. Once the array is returned (with 1 added to the index where the coin_value was removed), it is compared to the current coin_tally, if less a form of making change with less coins has been found!

The time complexity can be analyzed from the fact that which each loop spawns up to n more recursive calls depending on the size of the new current_amt. The same subproblems are

solved multiple times. However, in the case where a coin_value is greater than the current_amt, there are no additional recursive calls. This allows the setting of an upper limit. It is safe to say that:

$T(n) = O(2^n)$

Greedy Algorithm

Pseudocode:

```
Function changegreedy(coins, amount)
    maxIndex  ← coins.length - 1
    coinCount ← zero filled array of size coins.length
    numCoins ← 0

    while amount > 0
        if coins[maxIndex] <= amount
            coinCount[maxIndex] ← amount / coins[maxIndex]
            amount ← amount - (coins[maxIndex] * coinCount[maxIndex])
            numCoins ← numCoins + coinCount[maxIndex]

        maxIndex ← maxIndex - 1

    return (coinCount, numCoins)
```

Asymptotic Analysis

This algorithm tracks the index of the largest coin value in the coins array that can be subtracted from the given amount, those coins which were used to achieve the amount and the total number of coins used. Before jumping into the while loop, the largest coin able to be subtracted from the amount is assumed to be the last element in the coins array, insofar as the coins array is sorted in ascending order, and the number of coins needed to achieve the amount is considered to be zero. The coinCount array acts as a parallel array in relation to the coins array, that is, the indices of coinCount correlate to the denominations represented in the coins array at each index.

The while loop depends on the current amount of cents still able to be represented by a particular denomination, assuming that the smallest coin value is 1, so that a solution will always be available. Once inside the loop, we check to see if the currently considered largest value can be subtracted from the current amount. If so, we find out how many times this value can be removed from the amount by performing integer division, recording the quotient in the coinCount array at the relevant index and then subtracting the product of the coin value and the aforementioned

quotient from the current amount. The numCoins variable is then updated to reflect the number of coins represented in the coinCount array. Finally, the maxIndex is reduced so that the next largest denomination can be tested against the modified amount during the next iteration of the loop. The algorithm will then return an array detailing which coins were used to achieve the original amount as well as the total number of coins used.

In the best case scenario, the value at coins[maxIndex] will either equal the amount or equal some value evenly divisible by coins[maxIndex] during the first iteration of the loop, and so the algorithm will complete its task in constant time. However, if we consider the worst case scenario, then, after completing the constant time operations that must be completed prior to entering the loop, the iterative step of the algorithm will have to examine each denomination stored in the coins array before the amount reaches zero. In this way, the greedy algorithm will operate with a complexity of O(n) in which n is the number of denominations. Thus,

$$T(n) = c + cn$$
$$T(n) = c(n+1)$$
$$T(n) = O(n)$$

Dynamic Programming Algorithm

Pseudocode:

```
Function changedp(coins,amount)
   minCoins = [0] * (amount + 1)
   coinsUsed = [0] * (amount + 1)
   numCoins = [0] * length(coins)

   for each changeSubproblem in (amount + 1)
         coinsNeeded = changeSubproblem
         if changeSubproblem == 0
               lastCoinUsed = 0
         else:
               lastCoinUsed = 1

         For each coin in coins
               if coin <= changeSubproblem
                     if (1 + minCoins[changeSubproblem - coin] < coinsNeeded)
                           coinsNeeded = 1 + minCoins[changeSubproblem - coin]
                           lastCoinUsed = coin
```

```
            minCoins[changeSubproblem] = coinsNeeded
            coinsUsed[changeSubproblem] = lastCoinUsed

    coin = amount

    while coin > 0
            coinUsed = coinsUsed[coin]
            numCoins[coins.index(coinUsed)] += 1
            coin -= coinUsed

    return (numCoins, minCoins[amount])
```

Asymptotic Analysis

The dynamic programming change function takes an array of coins (denominations) and an amount and returns an array (numCoins) which is the number of coins used for each coin value and minCoins[amount] which is the minimum number of coins needed to make the amount given.

First we create 3 arrays. The first array, minCoins, is the number of coins needed to make an amount of change. The second array is coinsUsed, is the table used to calculate the quantity of each coin used to make an amount of change. Both of these arrays are filled with zeroes up to the amount of change. The third array, numCoins, holds the final results. It contains the quantity used for each coin value.

The for loop is a bottom-up approach that solves small sub-problems first and runs up to the given amount. The base case is established, such that either zero or one coin is needed for every cent in the sub-problem. The inner for loop does a majority of the work and runs through each coin in the coins array and uses the optimal sub-structure of the sub-problems to determine the minimum number of coins required for the current sub-problem. Once the for loop is complete, the result is stored in the minCoins table, and the last coin used is stored in the coinsUsed table. The while loop goes through each coin used and calculates how many coins of each value were used. Lastly, the function returns the numCoins array and the minimum number of coins needed to make the given amount.

The outer for loop runs up to the given amount, and the inner for loop runs through each denomination in the coins array. Due to the nested for loop, the runtime for the dynamic programming change function is **O(kn)** or **O(A*C)**, where A is the amount needed to make change for and C is the number of denominations. The runtime graph will be linear.

2. Describe, in words, how you fill in the dynamic programming table in changedp. Justify why is this a valid way to fill the table?

An array labeled minCoins where each index represents a cash amount is used to keep track of the minimum cost to make certain change. The array is the size of the change that needs to be made initially. This approach makes sense because this is the maximum value that needs to be made so finding the sum of any change A - V[i] where V[i] is a coin denomination will always be less than the max amount. In this way only specific values in the table are filled, whose index will be a certain coin subtracted from the current change.

Ultimately this solves the issue from the brute force method where subproblems are recalculated every time to deduce the cost of a larger amount of change.

3. Prove that the dynamic programming approach is correct by induction. That is, prove that $T[v]= \min_{V[i]\leq v}\{T[v-V[i]]+1\}$, T [0] = 0 is the minimum number of coins possible to make change for value v.

Given a set of coin values V=[V[1],V[2]....V[i]] *in* N and a value of v *in* N that needs to be made by the smallest combination of these coins. Let T[v] be the minimum number of coins to make the value v. We assume V[1] will always be 1.

Take the following two statements to be the base cases:

To prove $T[v]= \min_{V[i]\leq v}\{T[v-V[i]]+1\}$ we can first show that T[0] = 0 since it would take 0 coins to make 0 cents. It can also be said that for any value of change $v = V[i]$, $T[V[i]] = 1$ because since it is given that the coin V[i] is in the set V, only 1 coin is needed to make this amount of change..
So for a given array V=[1,2,5], T[V[1]] = T[1] = 1. T[V[2]] = T[2] = 1. T[V[5]] = T[5] = 1.

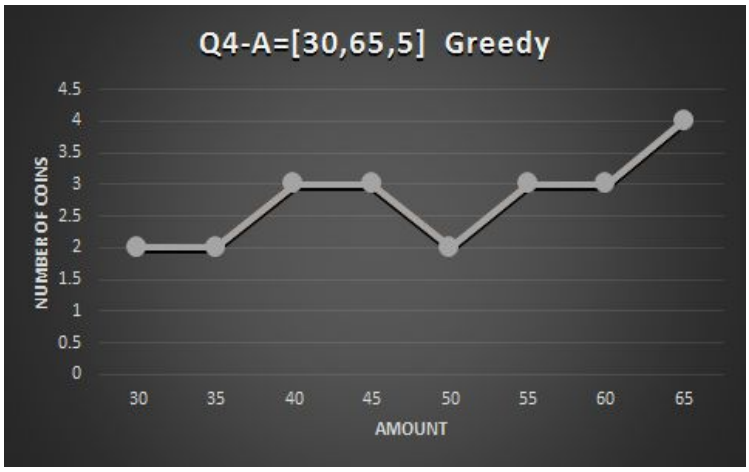
Inductive hypothesis:

Assume that this relation is true for all T[k] where $k \leq v$, and v is the amount of change to ultimately make as stated above and T[k] is the optimal amount of coins for change k:
$T[k+1]=\min_{V[i]\leq k+1}\{T[(k+1)-V[i]]+1\}$, $k \leq v$

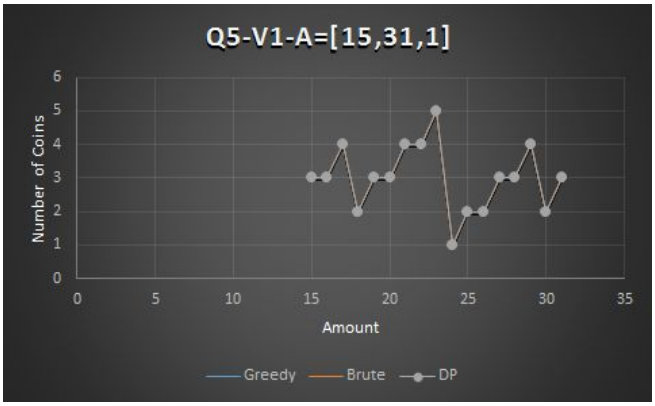
Since $k + 1$ - V[i] ≤k, $T[(k+1)-V[i]]$ is already the optimal value stored in the table according t the inductive hypothesis. Additionally, If v > 0 then it is assumed that a subset of V = v for a given combination of the specific coins V[i]. Then it can be stated that if an arbitrary coin is subtracted from the total v then we must be able to reach that value by adding just 1 coin so: v - V[i] $\leq$ T[v] -1. Rearranging the solution gives T[v] $\geq v - V[i] + 1$ while $v \geq V[i]$ which satisfies the statement to prove.

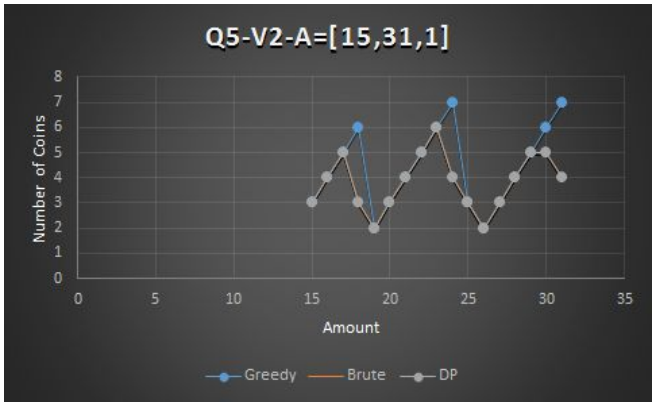
4. Suppose V = [1, 5, 10, 25, 50]. For each integer value of A in [2010, 2015, 2020, …, 2200] determine the number of coins that changegreedy and changedp requires.Plot the number of coins as a function of A for each algorithm. How do the approaches compare? *For the brute force method A consisted of the set [30,35….65]*

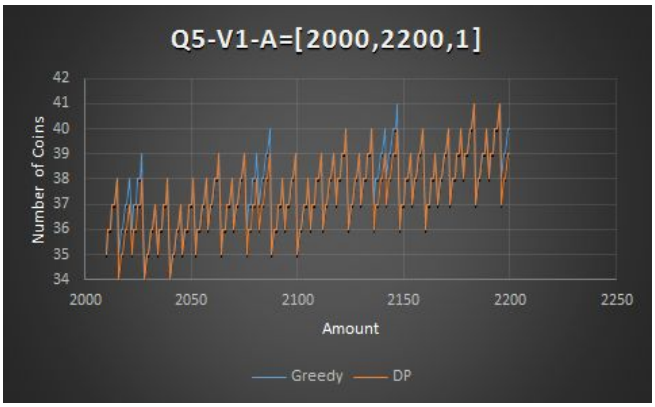
The number of coins used is equal for each method for each amount, meaning that for these particular denominations, the greedy algorithm will produce correct answers in the shortest amount of time.

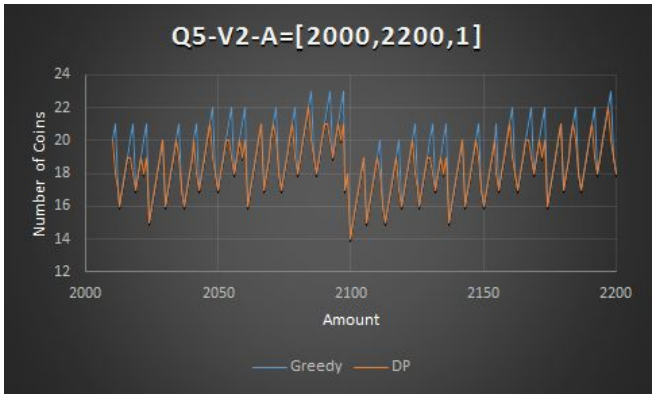


5. Suppose V1 = [1, 2, 6, 12, 24, 48, 60] and V2 = [1, 6, 13, 37, 150]. For each integer value of A in [2000, 2001, 2002, …, 2200] determine the number of coins that changegreedy and changedp requires. If your algorithms run too fast try [10,000, 10,001, 10,003, …, 10,100]. You can attempt to run changeslow however if it takes too long you can select smaller values of A and also run all three algorithms on the values. Plot the number of coins as a function of A for each algorithm. How do the approaches compare? *For the brute force method A consisted of the set [15,16,17…..31]*



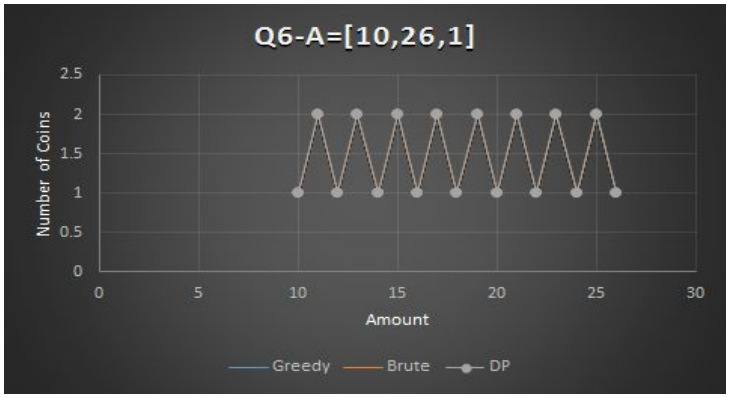


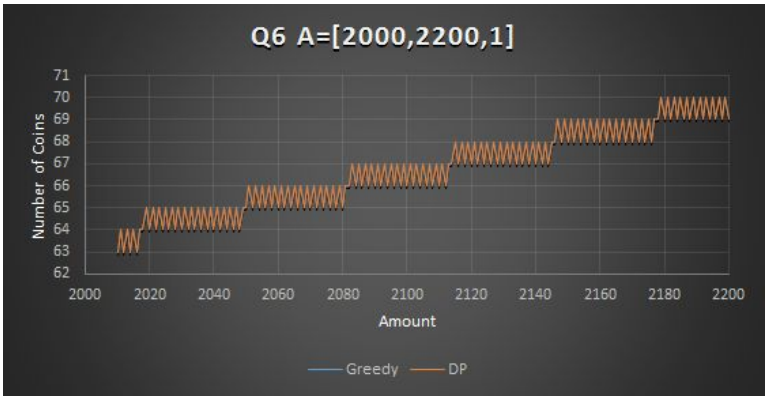




For the denominations used in Q5, the greedy algorithm sometimes fails and uses too many coins for the given amount. This is evidenced by the spikes in the graphs from [2000,2200] for the greedy algorithm. The errors seem to occur at regular intervals, so there is a pattern to them. The DP method consistently gets the correct answers.

6. Suppose V = [1, 2, 4, 6, 8, 10, 12, …, 30]. For each integer value of A in [2000, 2001, 2002, …, 2200] determine the number of coins that changegreedy and changedp requires. You can attempt to run changeslow however if it takes too long you can select smaller values of A and also run all three algorithms on the values. Plot the number of coins as a function of A for each algorithm. *For the brute force method A consisted of the set [10,11,12….26]*



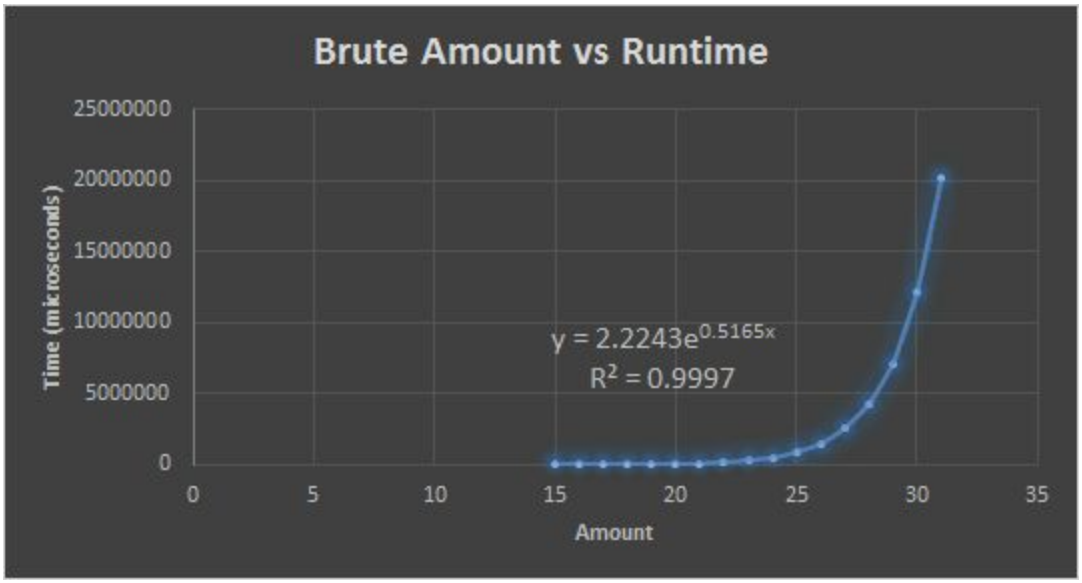


The greedy and DP methods both have the same number of coins for each amount. The graph shows that for the denominations used, the number of coins used will fluctuate back and forth for

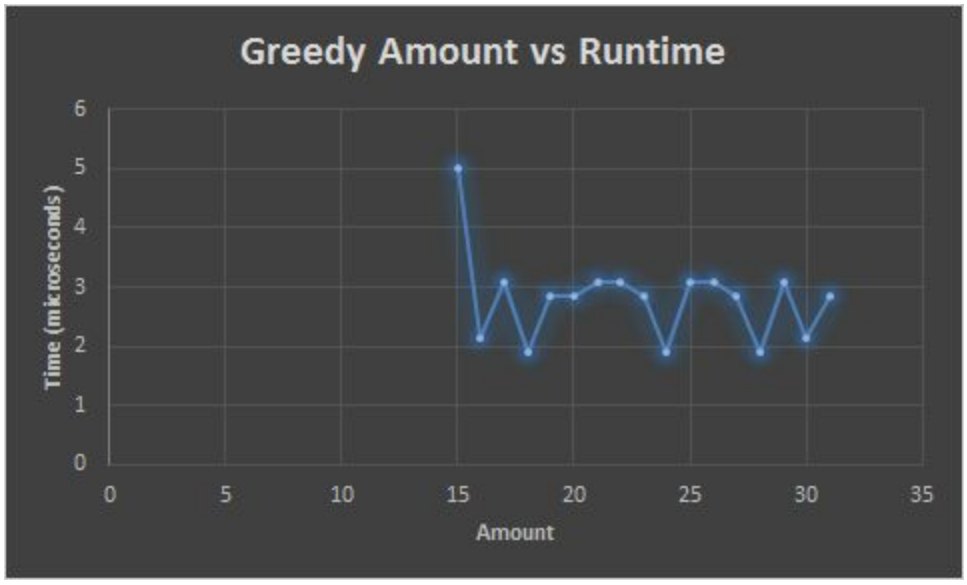
each successive value For every 30 amount or so, another coin is used by each algorithm. The greedy method produces the correct results for this problem.

7. For the above situations, determine (experimentally) the running times of the algorithms by fitting trend lines to the data or analyzing the log-log plot. Plot the running time as a function of A. Compare the running times of the different algorithms.

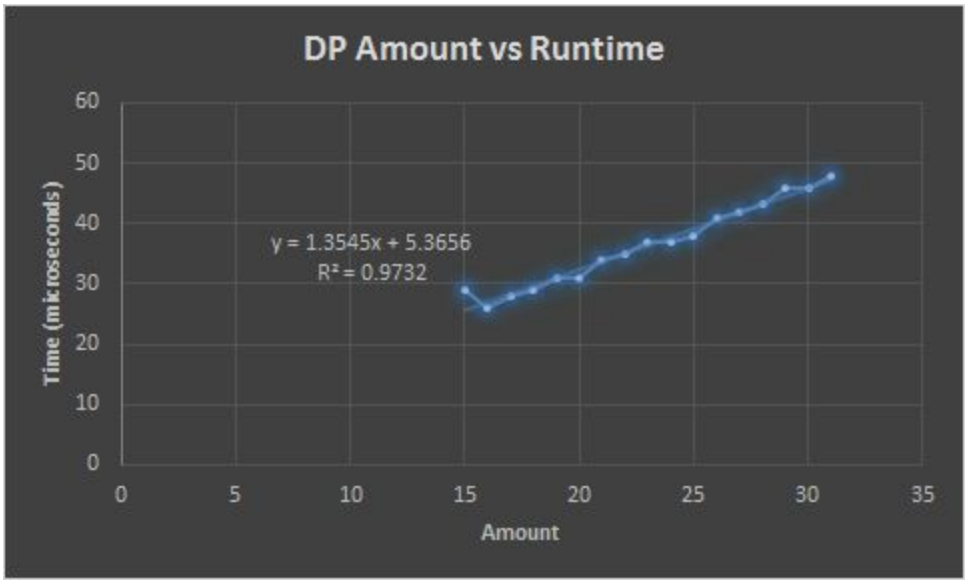
The following graphs are based on the data from Q5 where V = [1,2,6,12,24,48,60]
Brute force is exponential $O(2^n)$.



Greedy runtime cannot be determined experimentally, because runtimes fluctuate up and down.

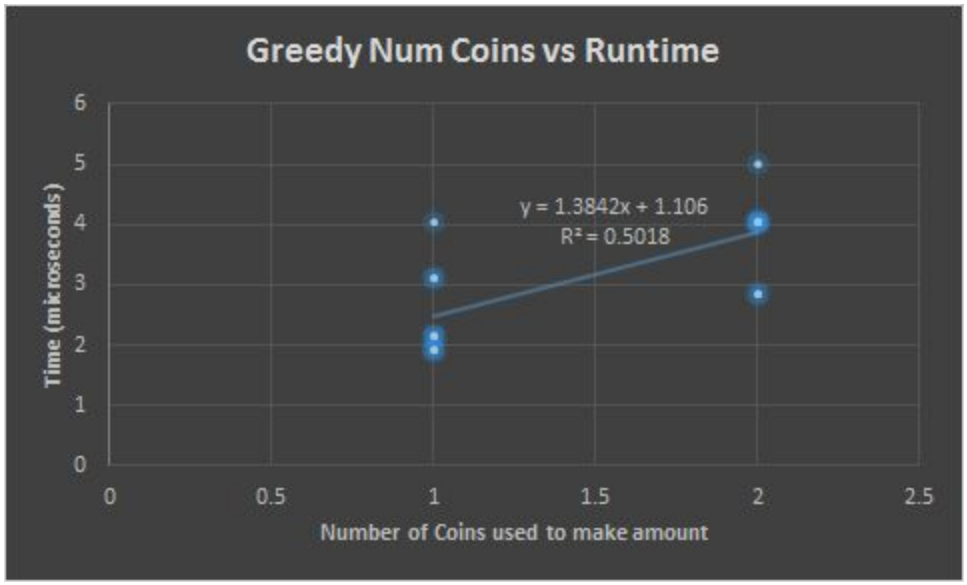


The dynamic programming algorithm is linear O(kn).

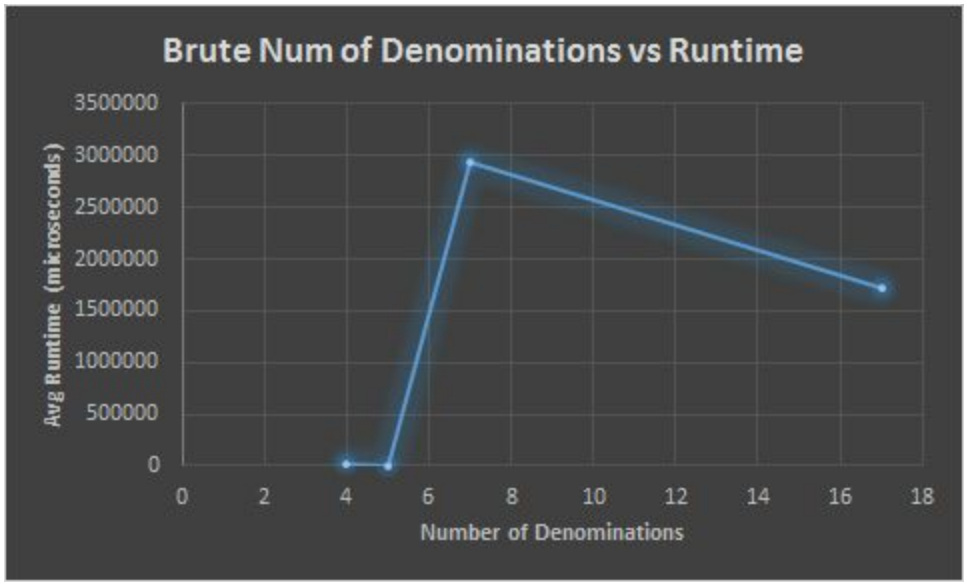


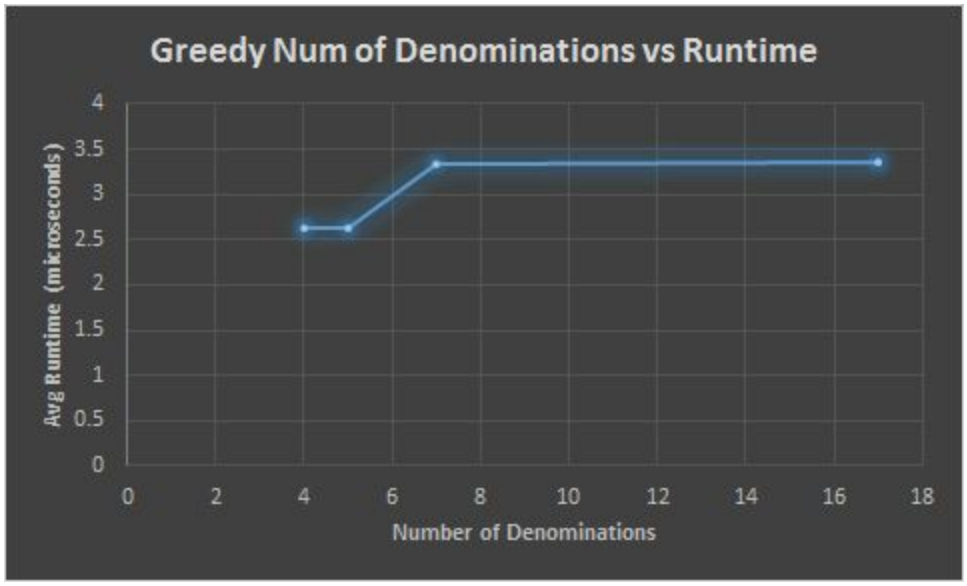
The brute force method is the slowest by a longshot because of its recursion, with an exponential runtime of $O(2^n)$. The dynamic programming method is much faster, with a linear runtime of O(kn). The greedy method is the fastest, but has a fluctuating runtime. The theoretical runtime for the greedy method is linear O(n), but this couldn't be found by plotting amount vs runtime.
It seems that the runtime for the greedy algorithm depends on how many coins are needed to make the given amount. This makes sense because the algorithm stops once it acquires enough coins to make the given amount. When there is only one coin used, the algorithm stops quicker than if multiple coins are used.

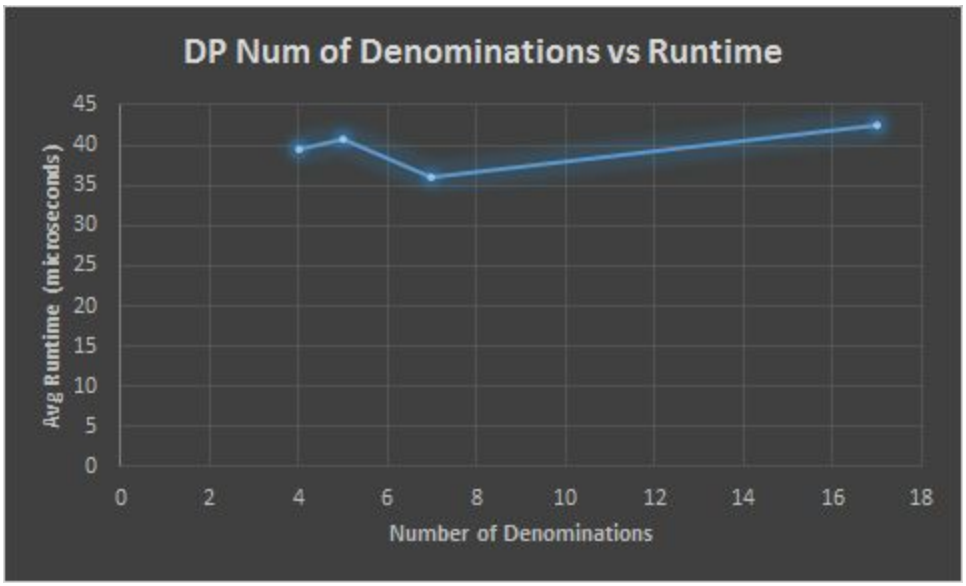
Data is from Q6:



8. Use the data from questions 4-6 and any new data you have generated. Plot running times as a function of number of denominations (i.e. V=[1, 10, 25, 50] has four different denominations so n=4). Does the size of n influence the running times of any of the algorithms?



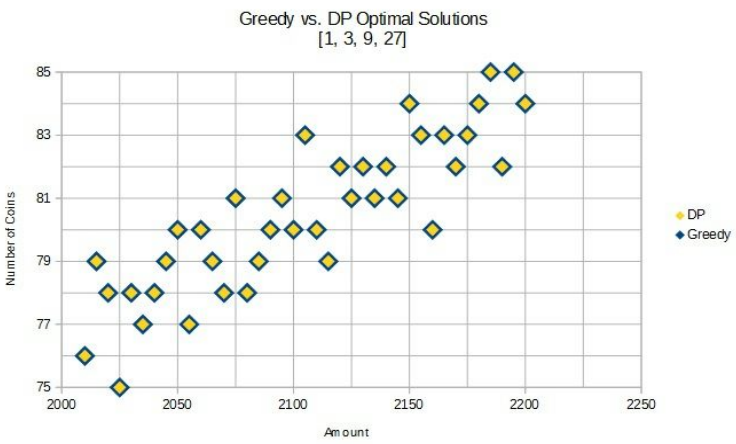


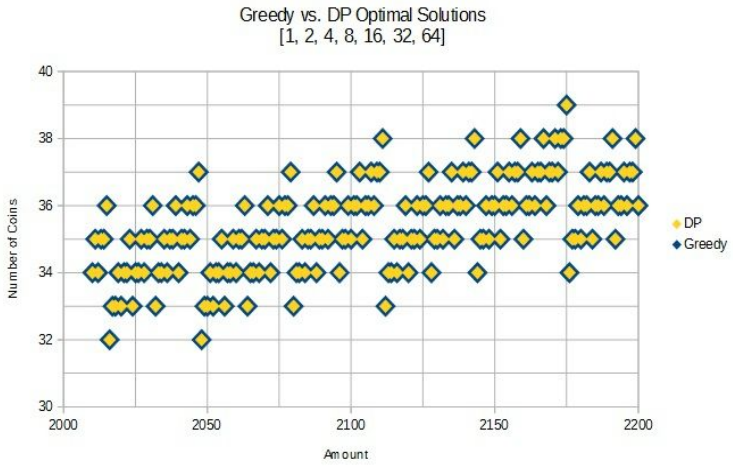


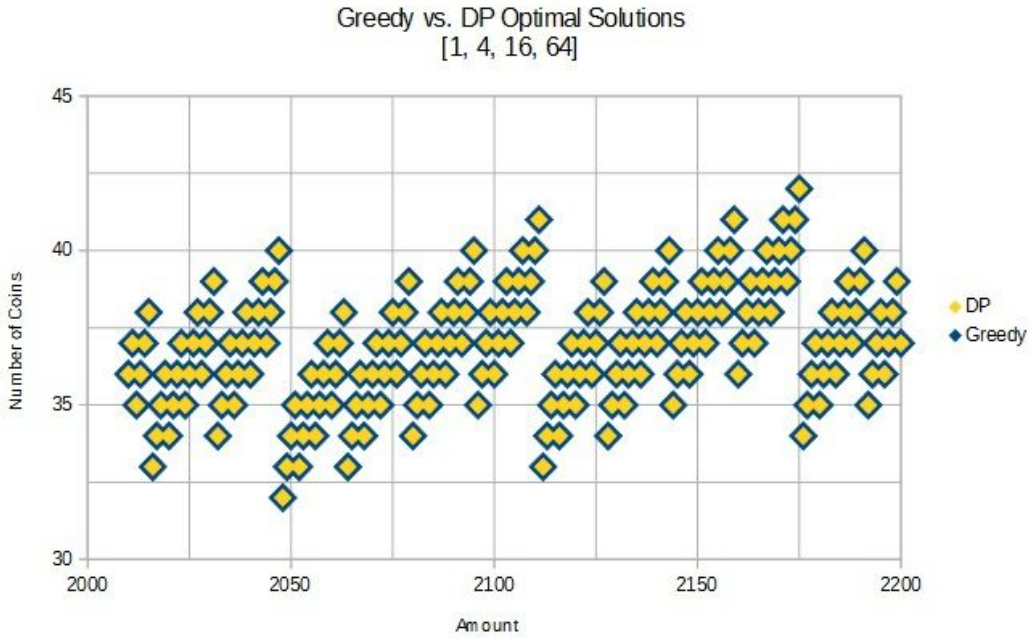
The number of denominations (n) had little effect on the greedy method and the dynamic programming method for the values of n that we used, but the brute force method was affected dramatically once the number of denominations reached a certain point. Theoretically, at far greater values of n, the DP method should be affected by the number of denominations, because the inner for loop runs through each denomination, giving it a runtime of O(kn).

9. Suppose you are living in a country where coins have values that are powers of p, V = [1, 3, 9, 27]. How do you think the dynamic programming and greedy approaches would compare? Explain.

Performing experimental tests using both greedy and dynamic programming algorithms on three different coin systems $V_1 = [1,3,9,27]$, $V_2 = [1,2,4,8,16,32]$ and $V_3 = [1,4,16,64]$ all yielded optimal results and can be seen in the graphs below.







From this perspective, we can see that the approach that will prove most suitable will not depend on accuracy, but on speed.

More formally, we can refer to the conclusions made in part 10 below, namely that there are two requirements on a coin system that allow for a greedy algorithm to consistently produce optimal solutions. These being:

1) All denominations, excluding a coin whose value is 1, must have a GCD equal to the second smallest value in the system.

2) The sum of the second largest coin value with itself, whether greater than, less than or equal to the largest denomination in a particular coin system must allow change to be made for this sum using a greedy algorithm that produces an optimal solution with no more than two coins.

Following these requirements we can generalize to all coin systems S whose denominations obey the following structure: $S = [C^0, C^1, ..., C^{n-1}, C^n]$. Satisfying the first requirement is trivial, since all denominations, being a power of some integer C will share a $GCD = C^1$. We can demonstrate the second requirement by proving that $C^{n-1} + C^{n-1} \leq C^n$. Using basic algebra and substitution,

$$C^{n-1} + C^{n-1} \leq C^n$$
$$\frac{2C^n}{C} \leq C^n$$

This inequality holds true for all $n \geq 0$ and $C \geq 2$. Since we can exclude consideration of a denomination equal to one, these conditions do not detract from the conclusion. Furthermore, as is demonstrated in part 10 below, the second requirement is easily satisfied by a sum that is less than or equal to the largest coin value. For example, using V_1, let $A = 9 + 9 = 18$. Since $18 \leq 27$, a greedy algorithm will produce an optimal solution involving two coins both of value 9. Considering V_2, let $A = 16 + 16 = 32$. Since $32 \leq 32$, a greedy algorithm will produce an optimal solution involving one coin of value 32.

Thus, both experimentally and theoretically, we can see that there should be no difference between a greedy and dynamic approach, as relates to the question of accuracy, when using a coin system whose values are a power of a given integer. In such cases, the speed of each algorithm should be considered, to which the greedy solution will most likely prevail.

10. Under what conditions does the greedy algorithm produce an optimal solution? Explain.

Whether or not the greedy algorithm produces a consistently optimal solution seems to depend, at least in part, on the available coin denominations it has to work with when making change. We can see from the data represented in this report that the greedy algorithm works when V is either [1, 5, 10, 25, 50] or [1, 2, 4, 6, 8, 10, 12, …, 30]. The question is then, what do these coin systems have in common? What is immediately obvious is that the values in each coin system, excluding 1, all have a GCD equal to the second smallest denomination in the system set. This means that, each set, when given in ascending order and excluding 1, is a sequence of the form $a_n = GCD(r)$, where r is the result of some calculation involving n. For example, if $r = n^2 - 2n + 2$, and GCD = 5, then $a_n = 5(n^2 - 2n + 2) = \{5,\ 10,\ 25,\ 50\}$. However, this assessment falls apart when looking at the V1 graphs in part 5. The GCD is 2 for the coin system [1, 2, 6, 12, 24, 48, 60], and yet the greedy algorithm is not guaranteed to produce an optimal solution. The difference here is that this set is considered merely a tight coin system, meaning that the greedy algorithm will not fail using this coin system on amounts smaller than its highest denomination.[1] The following chart demonstrates this for V = [1, 2, 6, 12, 24, 48, 60]:

[1] Section 1.1 definition 3 of Canonical Coin Systems for Change-Making Problems by Xuan Cai and Yiyuan Zheng of Shanghai Jiao Tong University (http://arxiv.org/PS_cache/arxiv/pdf/0809/0809.0400v1.pdf)

Amount	DP Solution	Greedy Solution		Amount	DP Solution	Greedy Solution
1	1	1		31	3	3
2	1	1		32	3	3
3	2	2		33	4	4
4	2	2		34	4	4
5	3	3		35	5	5
6	1	1		36	2	2
7	2	2		37	3	3
8	2	2		38	3	3
9	3	3		39	4	4
10	3	3		40	4	4
11	4	4		41	5	5
12	1	1		42	3	3
13	2	2		43	4	4
14	2	2		44	4	4
15	3	3		45	5	5
16	3	3		46	5	5
17	4	4		47	6	6
18	2	2		48	1	1
19	3	3		49	2	2
20	3	3		50	2	2
21	4	4		51	3	3
22	4	4		52	3	3
23	5	5		53	4	4
24	1	1		54	2	2
25	2	2		55	3	3
26	2	2		56	3	3
27	3	3		57	4	4
28	3	3		58	4	4
29	4	4		59	5	5
30	2	2		60	1	1

Running similar tests on [1, 5, 10, 25, 50] and [1, 2, 4, 6, 8, 10, 12, …, 30] demonstrates that these coins systems can also be considered tight. However, we can see from the graph V5-V2-A=[15,31,1] in part 5 that when V = [1, 6, 13, 37, 150], greedy fails well before 150 and so is not a tight coin system. Since all three tight coins systems produce optimal or near optimal solutions we can begin to consider which properties a tight coin system must possess in order for the greedy algorithm to work without counterexample.

If we look at the second to last denominations in [1, 5, 10, 25, 50] and [1, 2, 4, 6, 8, 10, 12, …, 30], and multiply each of these value by 2, we should get an amount greater than or equal to the largest denomination in its particular system. For [1, 5, 10, 25, 50], we get $A_1 = 50$ and for [1, 2, 4, 6, 8, 10, 12, …, 30], we get $A_2 = 56$. If A is equal to the largest denomination, then we know the optimal solution is 1 coin, otherwise, if A is greater than the largest denomination, then we know that at least one optimal solution for A is 2 coins (namely the sum of the second largest denomination with itself). In both of the aforementioned coin systems, this property holds true, that is A_1 can be achieved with a single coin (50) and A_2 can be solved using the greedy algorithm with a

minimum of 2 coins (30 and 26). Considering the tight coin system [1, 2, 6, 12, 24, 48, 60] that fails to consistently produce an optimal solution, we can see that it also fails this newly established requirement. For example, if $A_1 = 96$, then an optimal solution should be two 48 cent coins, and yet the greedy algorithm, taking the path $96 - 60 = 36$; $36 - 24 = 12$; $12 - 12 = 0$, asserts a three coin solution.[2] Therefore, the necessary conditions in which the greedy algorithm will result in an optimal solution depends on whether or not the coin system is considered tight, and if the sum of the second to last denomination with itself can be represented using no more than two coins, while a sufficient condition seems to be a coin system in which all denominations share a GCD equal to the the second smallest denomination in the system. With these assumptions in mind, let us perform a more formal analysis to either confirm or reject these suppositions.

Formal Analysis:

Given a coin system $S = [C_1, C_2, ..., C_m, C_{m+1}]$ where $C_1 < C_2 < ... < C_m < C_{m+1}$ and the GCD of C_i for all $1 < i \leq m+1$ is some integer $q = C_2$, we must show that if $C_m + C_m \geq C_{m+1}$, then the optimal solutions for $A = C_m + C_m$ is no more than two coins.

Let $C_m = qr$ and $C_{m+1} = qr + qs$ where r and s are nonnegative integers. Now, let $A = C_m + C_m$, where $A \geq C_{m+1}$. If $A = C_{m+1}$, then the greedy solution will simply subtract C_{m+1} from A, reducing A to zero and producing an optimal solution involving a single coin. If, however, $A > C_{m+1}$, then the greedy algorithm will subtract C_{m+1} from A, reducing A to a value greater than 0. Using basic algebra and substitution,

$$A = A - C_{m+1}$$
$$A = [q(r) + q(r)] - [q(r) + q(s)]$$
$$A = 2qr - qr - qs$$
$$A = qr - qs$$
$$A = q(r - s)$$

Thus, if S contains some C_i where $C_i = q(r - s)$, or, in other words, if a coin system supports a denomination whose value is equal to $C_m - (C_{m+1} - C_m)$, then the greedy solution will produce an optimal solution. Note that this also applies to ordered subsets of S in which the max index of S is reduced by some value greater than or equal to 1 and less than $m - 2$. In some cases, however, forming various subsets of S yields an $A < C_{m+1}$. Can we still consider S as well as those coin systems that comprise S to be optimal? For example, if $S = [1, 5, 10, 25, 50]$, this coin system

[2] Section 4 Theorem 11 and section 5 of Canonical Coin Systems for Change-Making Problems by Xuan Cai and Yiyuan Zheng of Shanghai Jiao Tong University (http://arxiv.org/PS_cache/arxiv/pdf/0809/0809.0400v1.pdf)

would be considered optimal under the current definition, however, if we examine a subset of S such that $S_1 = [1, 5, 10, 25]$, we see that $C_m + C_m < C_{m+1}$. And yet, in much the same way that an optimal solution is produced when $C_m + C_m = C_{m+1}$, if $C_m + C_m < C_{m+1}$, then the optimal solution will inevitably involve two coins whose value is C_m.

Therefore, the greedy algorithm will produce an optimal solution using a coin system that meets these two requirements:

1) All denominations, excluding a coin whose value is 1, must have a GCD equal to the second smallest value in the system.

2) The sum of the second largest coin value with itself, whether greater than, less than or equal to the largest denomination in a particular coin system must allow change to be made for this sum using a greedy algorithm that produces an optimal solution with no more than two coins.