CS325 - Project 2: Coin Change
Group 17
Members:

- Michael Lewis
- Patrick Mullaney
- Matt Nutsch

Date: 4-26-2017

1. Give the pseudocode and theoretical asymptotic running time for each algorithm.

1. Implement a Brute Force or Divide and Conquer Algorithm:

Brute Force algorithm:

```
//find the maximum quantity for each coin
//loop through quantities for coin 1 from 0 to the max
 //loop through quantities for coin 2 from 0 to the max
  //loop through quantities for coin 3 from 0 to the max
   //loop through quantities for coin 4 from 0 to the max
    //loop through quantities for coin 5 from 0 to the max
     //loop through quantities for coin 6 from 0 to the max
      //loop through quantities for coin 7 from 0 to the max
       if the total of the coins = the amount sought
        if the number of coins is less than the current minimum
         save the coin combination as the output
```

The theoretical run time should be exponential, O(d^n), where d is the number of denominations and n is the amount. This is because the algorithm has d nested loops through every possible combination for each denomination.

2. Greedy Algorithm:

```
changeGreedy(vector coins, int amount, struct change)
{
        // struct change contains vector to count coin denominations and int for minimum
        number of coins
        // start at end of array (largest denomination), loop downward
        for(int i = coins size; i >=0; i--) {
                //while denomination is less than or equal to amount, subtract and repeat
                while(coin[i] <= amount)
                 {
                        Amount = amount - coin[i]; // subtract coin denomination from amount
                        Change.min++ // increment minimum number of coins
                        Change[i]++ // increment count of coins for that denomination
                }
        }

        Return change;
}
```

Runtime would be Θ(n) as it loops through the array n times with a constant amount of work done each time.

3. Dynamic Programming:

```
changeDP(vector coin, int amount, struct change)
// struct change contains vector to count coin denominations and int for min number of coins
        // declare array as memoization table for minimum coin values
        table[amount + 1]
        // declare array as memoization table to track coins used (frequency of denominations)
        coinsUsed[amount + 1]

        // initialize tables for base case
        table[0] = 0
        coinsUsed[0] = 0

        // initialize remaining elements of tables with sentinel values
        for i = 1 to amount
                table[i] = ∞      // this can be replaced with an arbitrarily large integer
                coinsUsed[i] = -∞         // this can be replaced with a negative integer

        // iterate through values of n less than or equal to the given amount of change
        for n = 1 to amount
                // iterate through all possible coin denominations k
                for k = 0 to size of coin vector
                                if coin[k] <= n
                                // minimum is table i minus that denomination
                                min = table[n – coin[k]
                                if min < currentMin
                                // we have used one more coin, so new min is min + 1
                                        table[n] = min + 1
                                        // assign index j of denomination used to coinUsed[n]
                                        coinsUsed[n] = k

        // set minimum number of coins from struct change to memoization table amount
        change.m = table[amount]

// set denomination amounts to amount from coinsUsed table.
        for i = change.m – 1 to 0
        // increment coin using coinsUsed table as index
change.coin[coinsUsed[amount]]++
// decrement amount according to coinsUsed table
amount -= coin[coinsUsed[amount]];
```

return change

Runtime:
The Dynamic Programming algorithm's runtime will be determined by its nested for loop. The outer for loop runs in linear time *n*, where *n* is equivalent to the amount for which we want to make change. The inner for loop checks each denomination from the coin vector, where *k* is the maximum, against the outer loop's current *n* value in linear. Thus, the inner loop runs at worst in linear time *k*. Therefore, taken together, the nested for loop has a worst case running time of $\Theta(n * k)$. Since this nested for loop dominates the other components of the algorithm, the algorithm itself will have a running time of $\Theta(n * k)$, which is pseudo-quadratic.

---

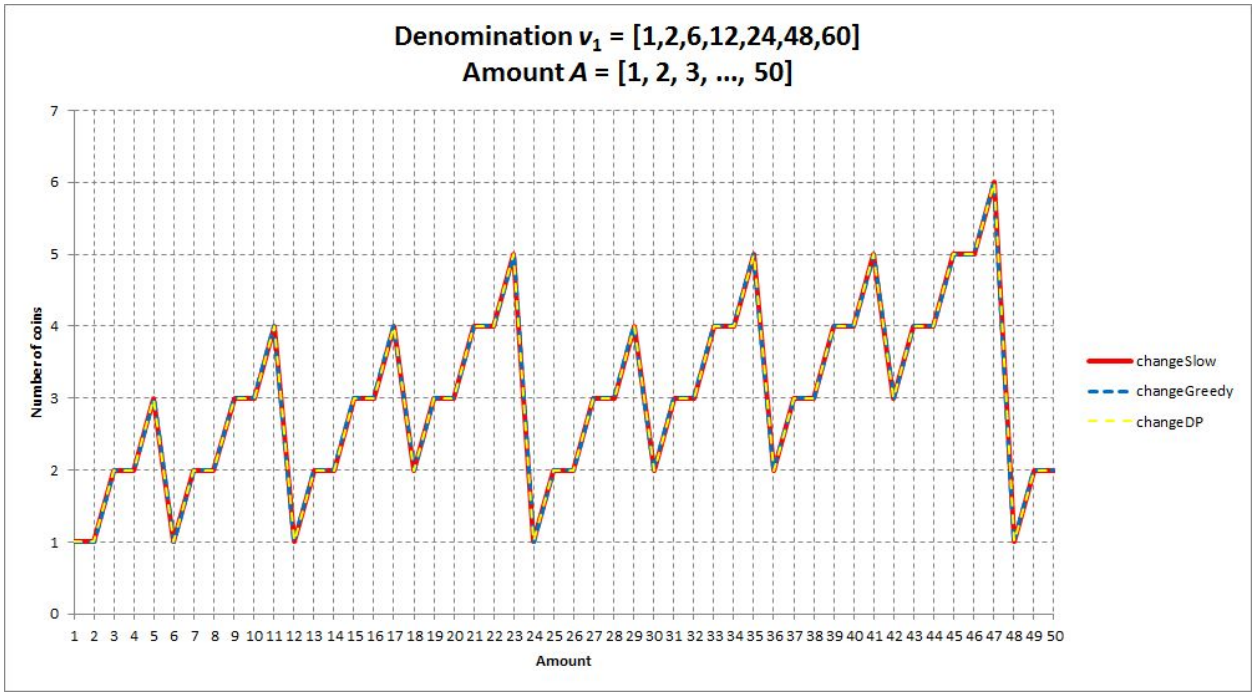
2. Describe, in words, how you fill in the dynamic programming table in changedp. Justify why is this a valid way to fill the table?

Essentially, the table is filled for each value based on previous values, starting at the base, table[0] = 0. The table is then built from the bottom up in this manner, with a table for value i calculate based on the value of table[i less coin[j]]. This is a valid way to fill the table as the value of table[i less coin[j]] will have already been filled as we fill from the bottom up.

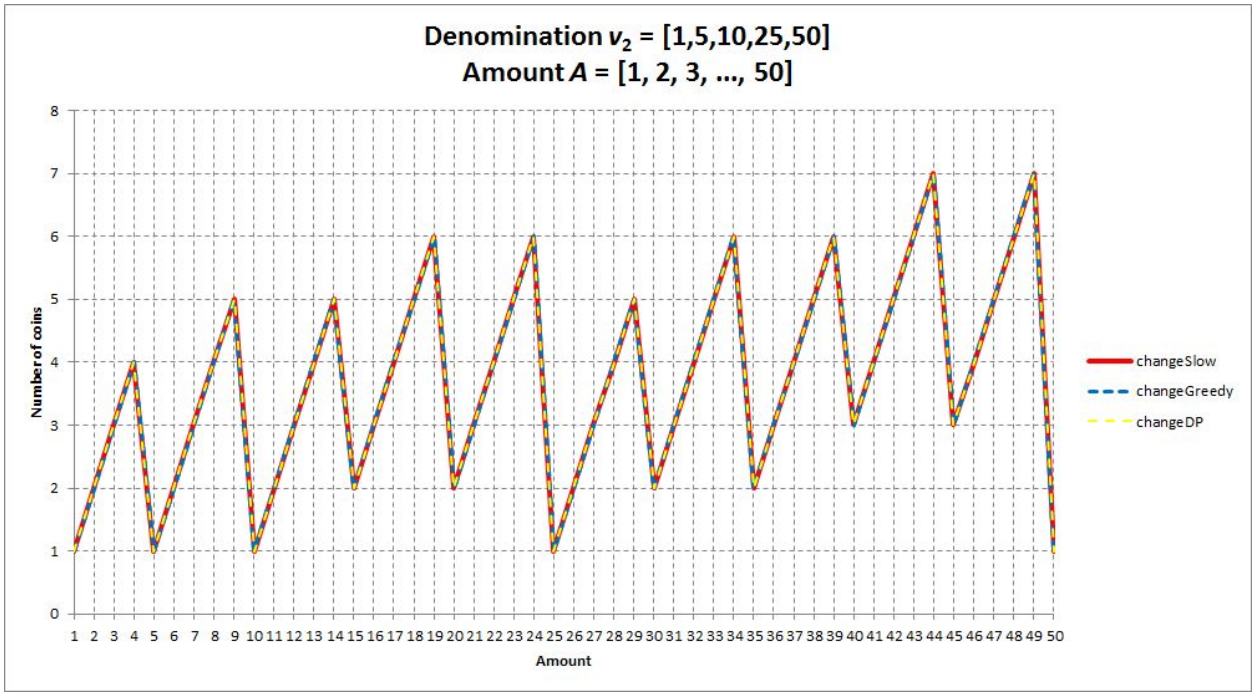
For example, say we have coins 1, 5, and 10. We know that table[1] = 1 and so on up to 5. At table [5], we now have use of a larger denomination and thus the minimum value for table[5] is 1. When we need to calculate table[6], we have already calculated values for 5 and 1, so there is no need to recalculate these values recursively.

---

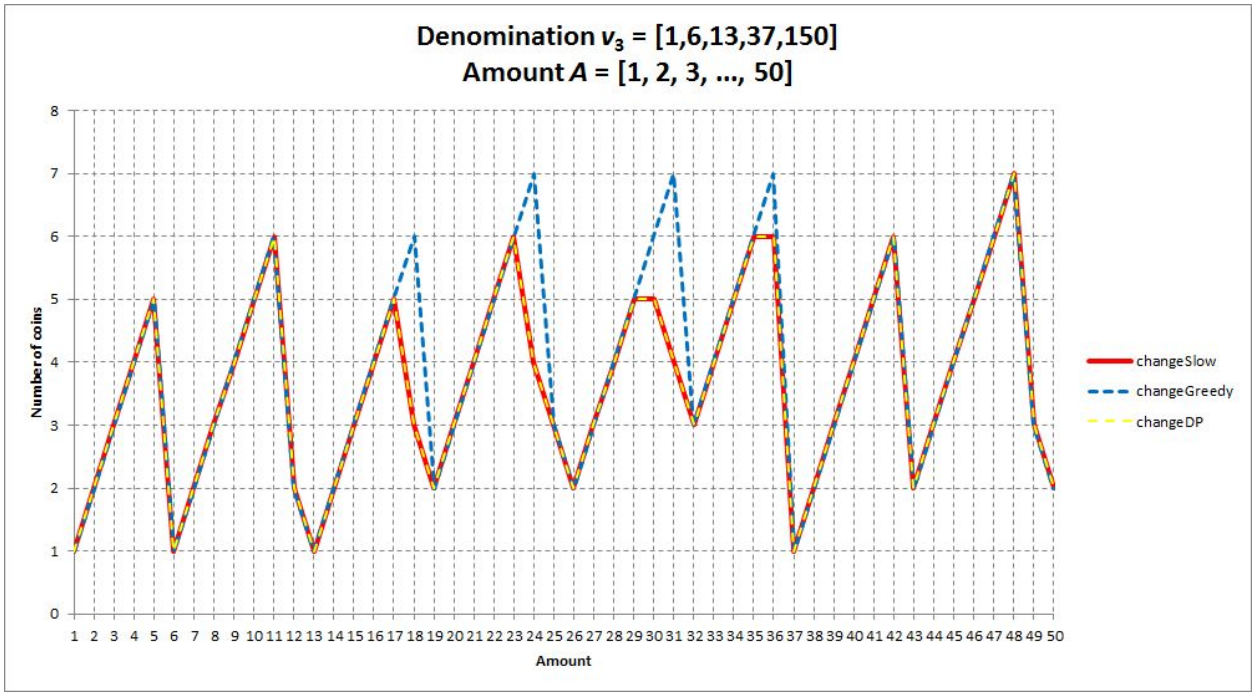
3. Suppose V_1 = [1, 2, 6, 12, 24, 48, 60], V_2 = [1, 5, 10, 25, 50] and V_3 = [1, 6, 13, 37, 150], for each integer value of *A* in [1, 2, 3, ..., 50] determine the number of coins that changeslow. changegreedy and changedp requires for each denomination set. For each integer value of *A* in [2000, 2001, 2002, ..., 2200] determine the number of coins that changegreedy and changedp requires for each denomination set (you can attempt to run changeslow but it will probably be too slow). Plot the **number of coins as a function of *A*** for each algorithm. How do the approaches compare?



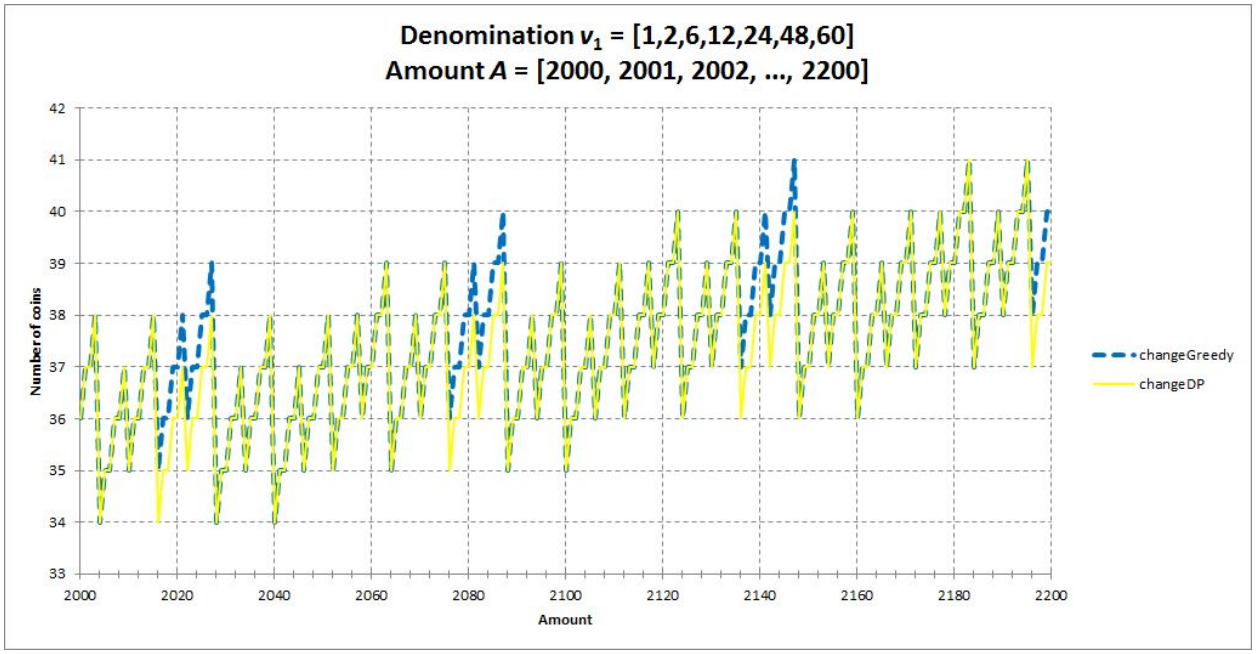
From the above plot, we observe that the changeslow (Brute Force), changegreedy, and changedp approaches give the same results for a coin system of denominations v_1 = [1,2,6,12,24,48,60], when tested with amounts in the interval [1, 2, 3, ..., 50]. It is true that the changeslow and changedp algorithms give the same results. However, while it is not evident from the amount interval in this plot, we will see (in the upcoming plot which expands the results for these denominations to amounts in the interval [2000, 2001, 2002, ..., 2200]) that in actuality, the greedy algorithm does not always give an optimal result for this particular coin system.



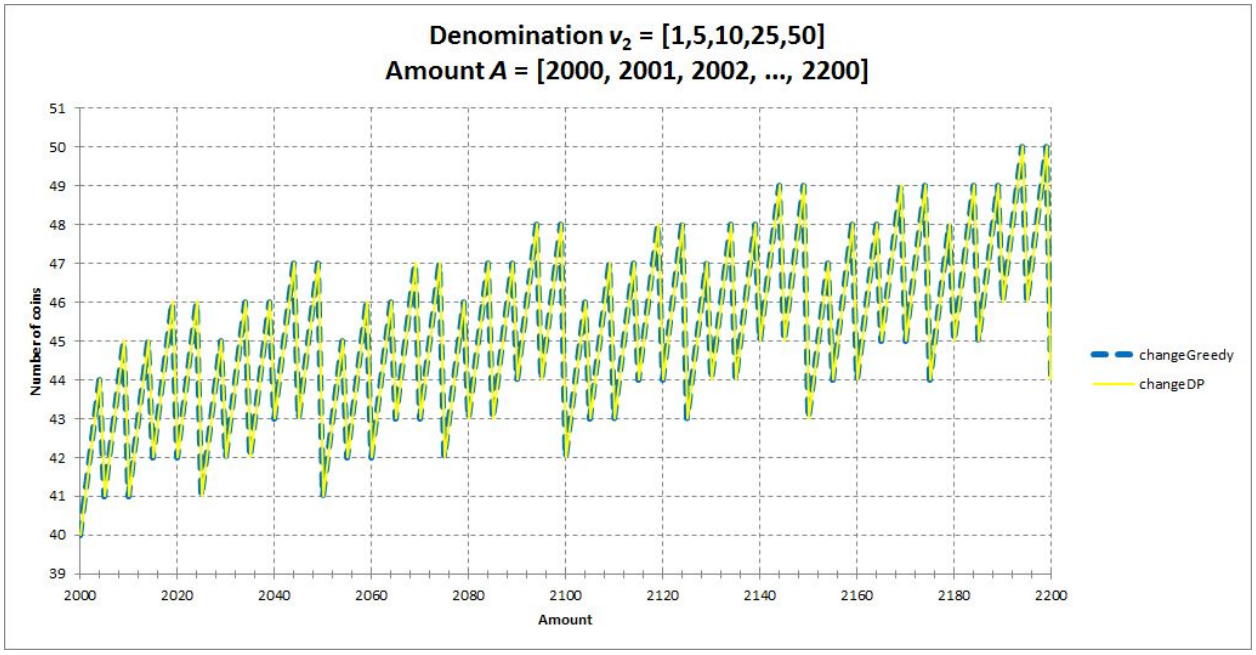
From the above plot, we observe that the changeslow (Brute Force), changegreedy, and changedp approaches give the same results for a coin system of denominations v_2 = [1,5,10,25,50], when tested with amounts in the interval [1, 2, 3, ..., 50]. When expanded to a higher interval (in the upcoming plot which tests the results for these denominations with amounts in the interval [2000, 2001, 2002, ..., 2200]), all three algorithms continue to give the same optimal results for this particular coin system.



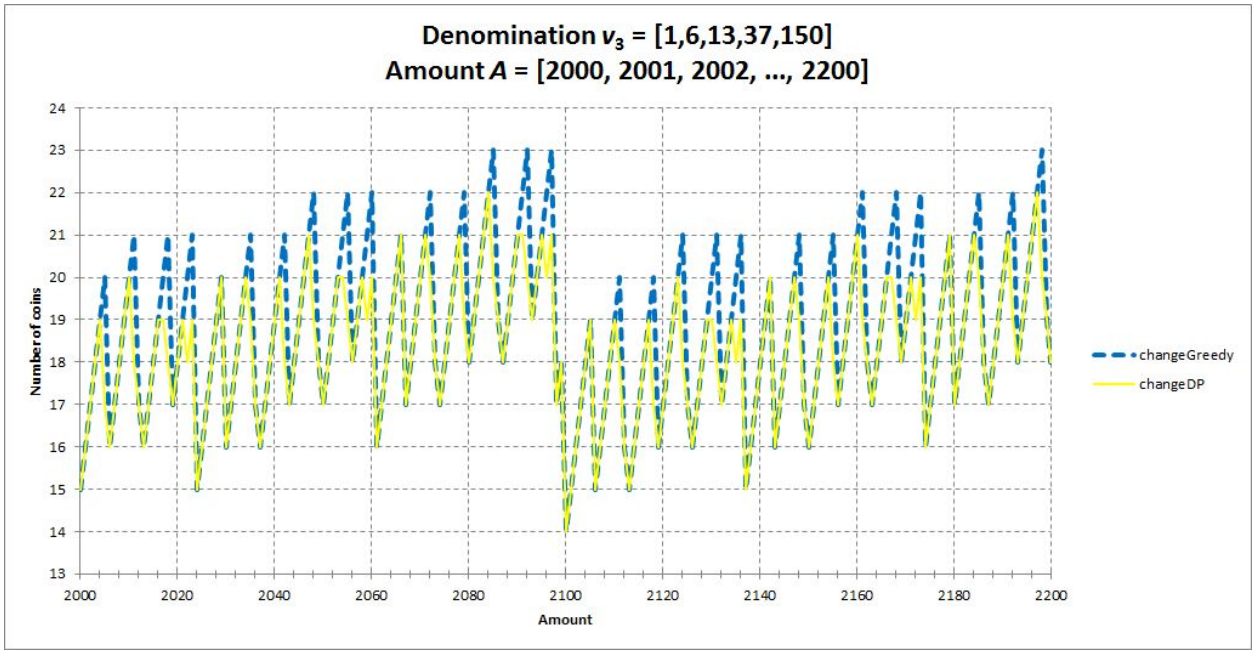
From the above plot, we observe that the changeslow (Brute Force) and changedp approaches give the same results for a coin system of denominations v_3 = [1,6,13,37,150], when tested with amounts in the interval [1, 2, 3, ..., 50]. However, it is evident that the greedy algorithm does not always give an optimal result for this particular coin system, as it yields a greater result for certain amounts.



From the above plot, we observe that the changegreedy and changedp approaches do not always give the same results for a coin system of denominations v_1 = [1,2,6,12,24,48,60]. When tested with amounts in the interval [2000, 2001, 2002, ..., 2200], it is evident that the greedy algorithm may give a suboptimal result for certain amounts with this particular coin system. The fact that this coin system is suboptimal is especially notable here because suboptimal results were not demonstrated by the greedy algorithm in the plot that only tested for amounts in the interval [1, 2, 3, ..., 50].



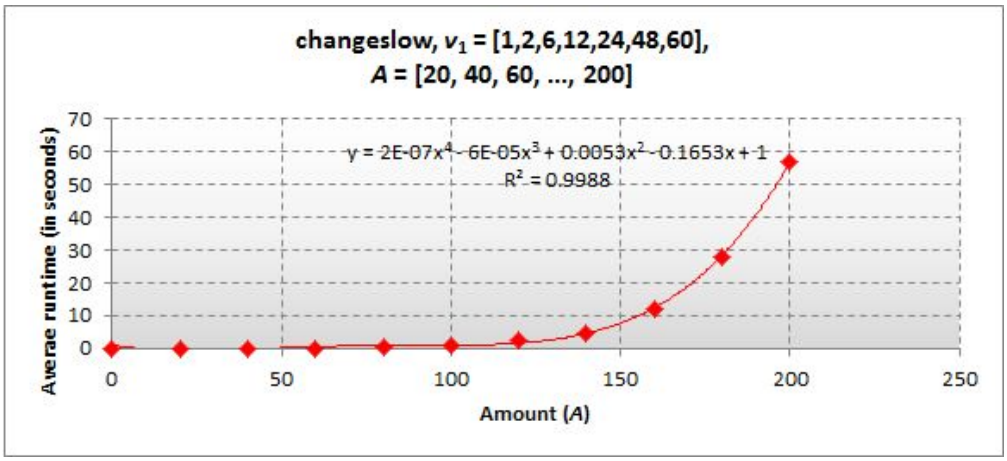
From the above plot, we observe that the changegreedy, and changedp approaches give the same results for a coin system of denominations v_2 = [1,5,10,25,50], when tested with amounts in the interval [2000, 2001, 2002, ..., 2200]. This optimality is consistent with what we observed with the lower interval tested in the earlier plot, which provided the results for these denominations with amounts in the interval [1, 2, 3, ..., 50]).

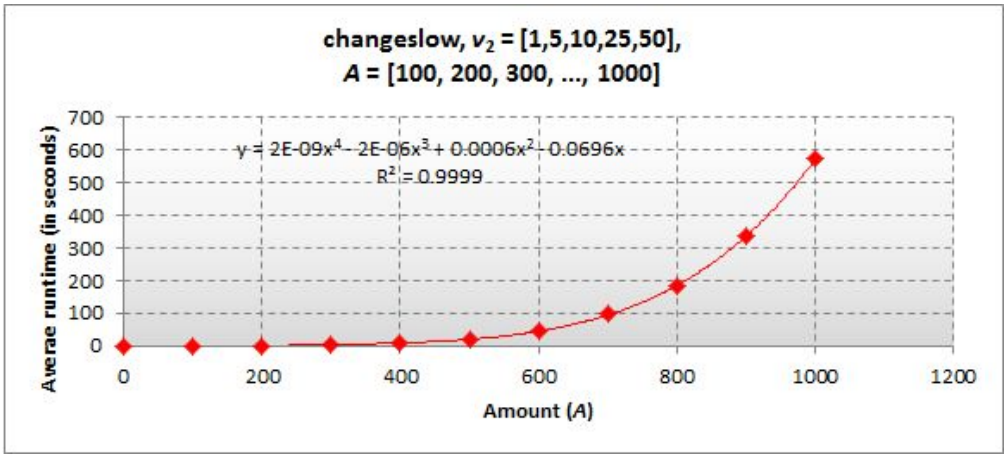


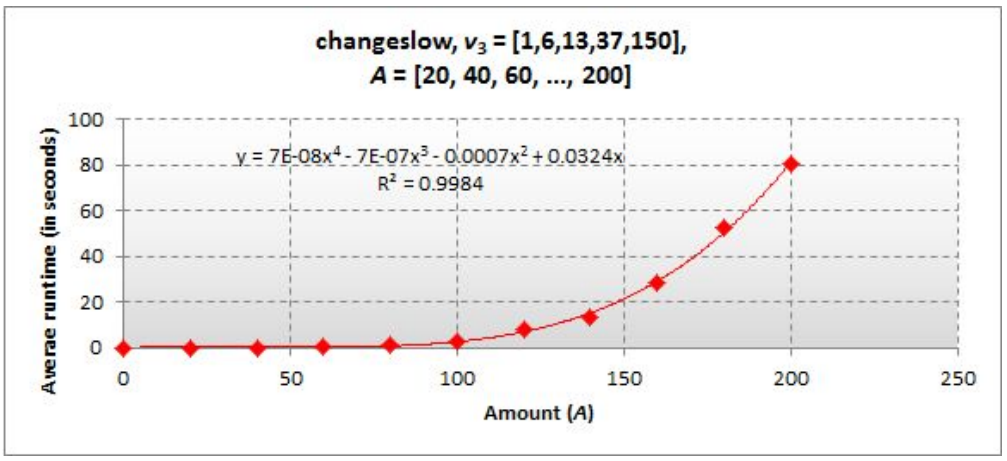
From the above plot, we observe that the changegreedy and changedp approaches do not always give the same results for a coin system of denominations v_1 = [1,2,6,12,24,48,60]. When tested with amounts in the interval [2000, 2001, 2002, ..., 2200], it is evident that the greedy algorithm may give a suboptimal result for certain amounts with this particular coin system.

4. For each of the algorithms collect experimental running time data. Plot the **running time as a function of A** and fit curves to the data. Compare the experimental running times to the theoretical running times.

Runtime Plots for changeslow:



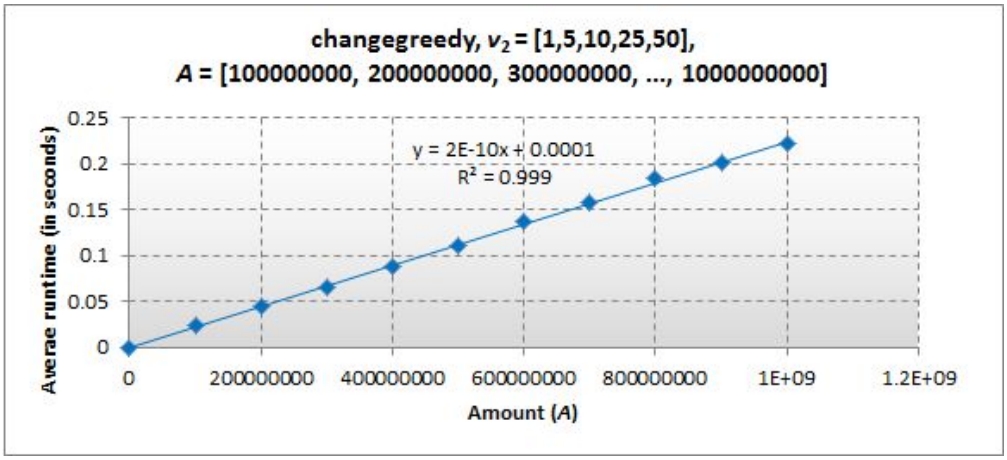


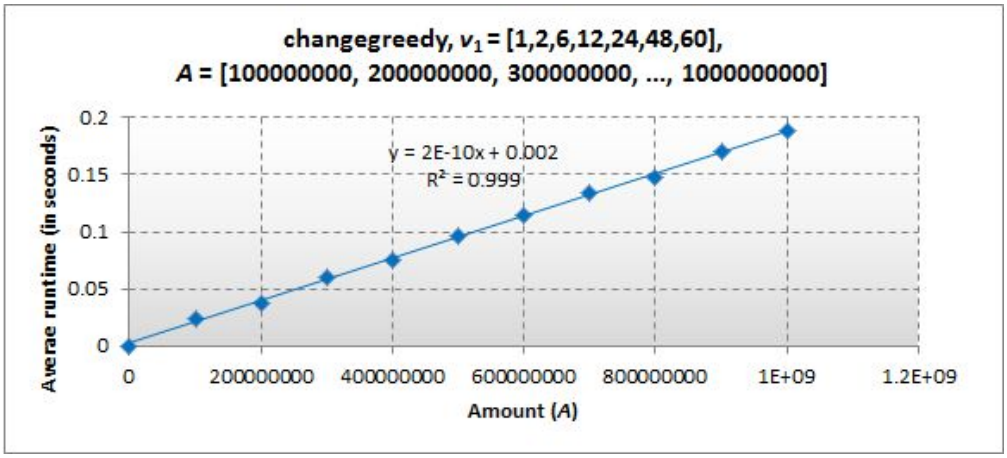


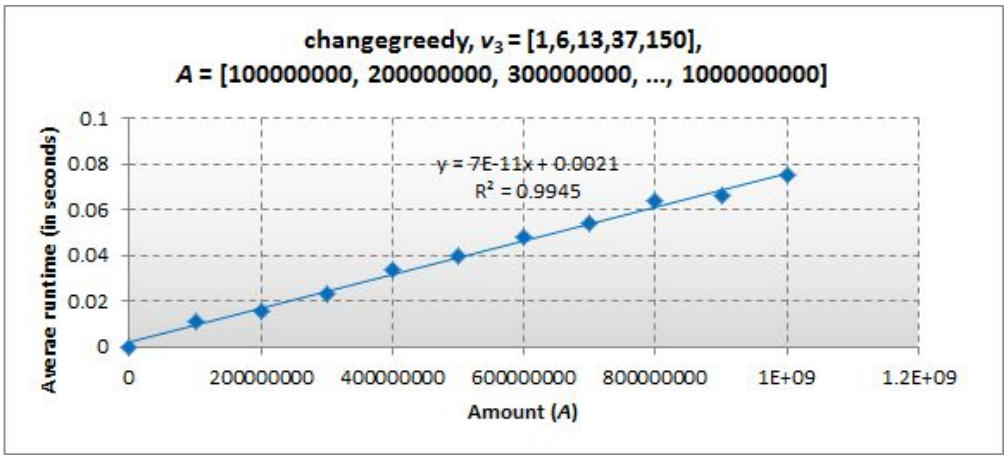
Runtime Analysis for changeslow:

The changeslow algorithm exhibited an exponential growth rate. We found that a higher level polynomial function actually provided a better fit. This correlates with our theoretical expectations for the algorithm, as the performance varies depending on how many coins are in a given system.

Runtime Plots for changegreedy:

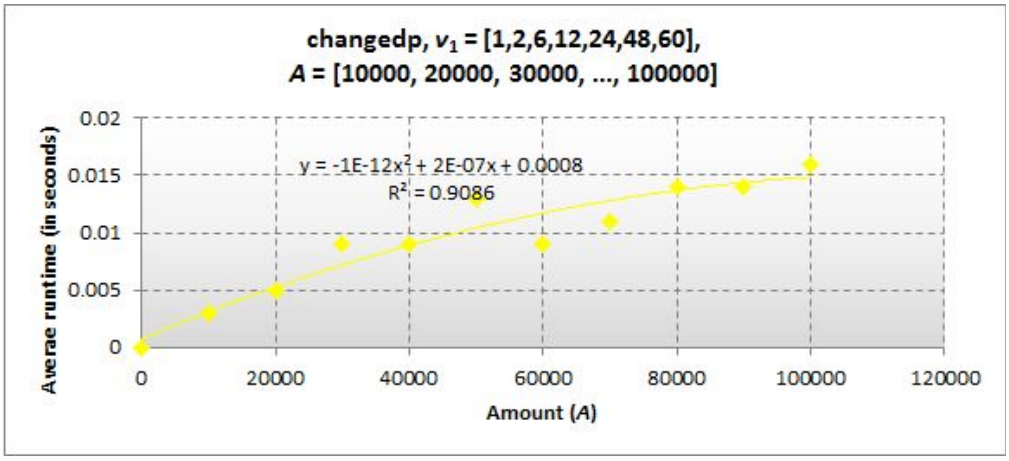


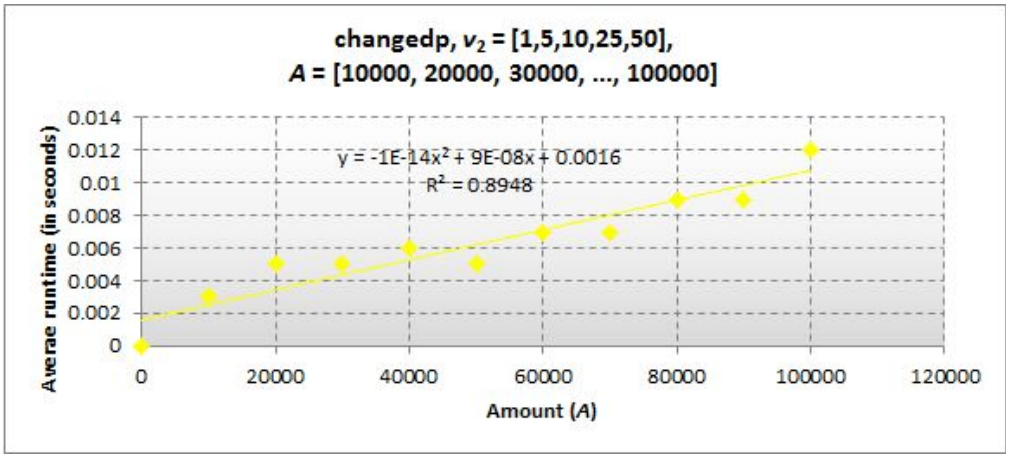


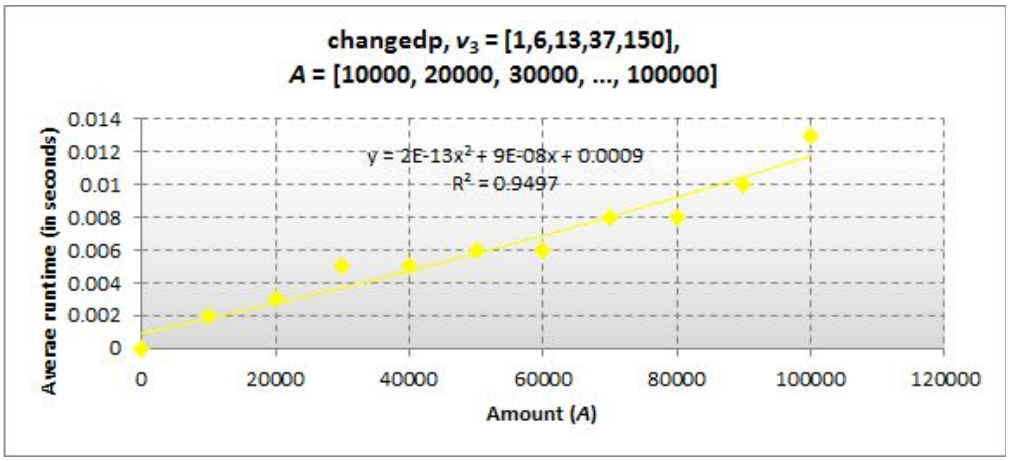


Runtime Analysis for changegreedy:
The changegreedy algorithm exhibited linear time for each of the coin systems tested with. This matches our theoretical of $\Theta(n)$.

Runtime Plots for changedp:

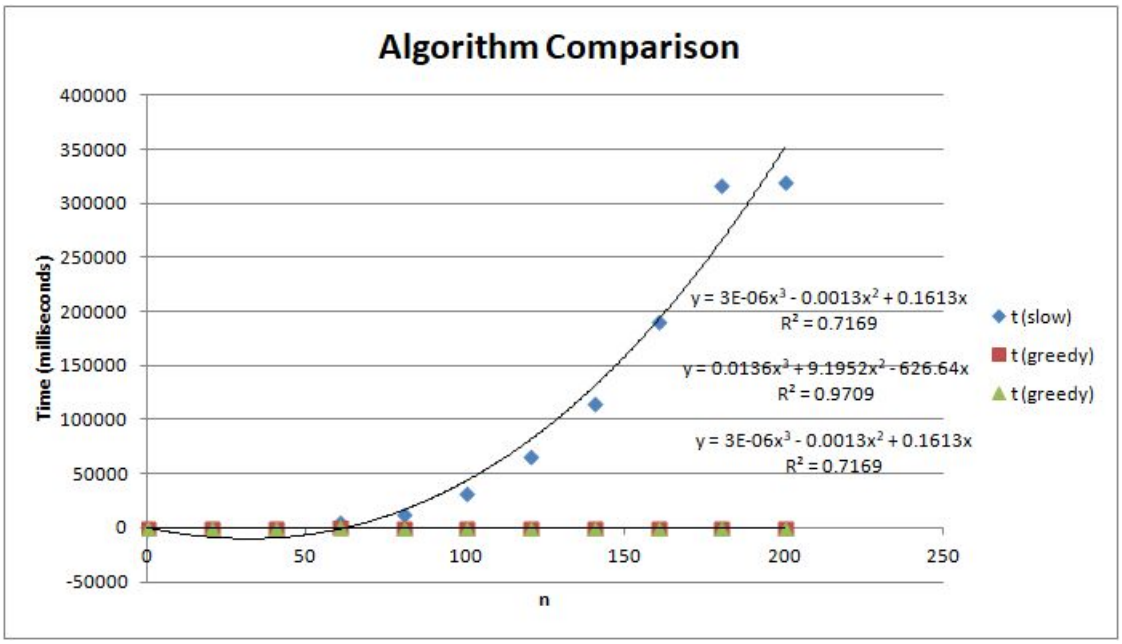


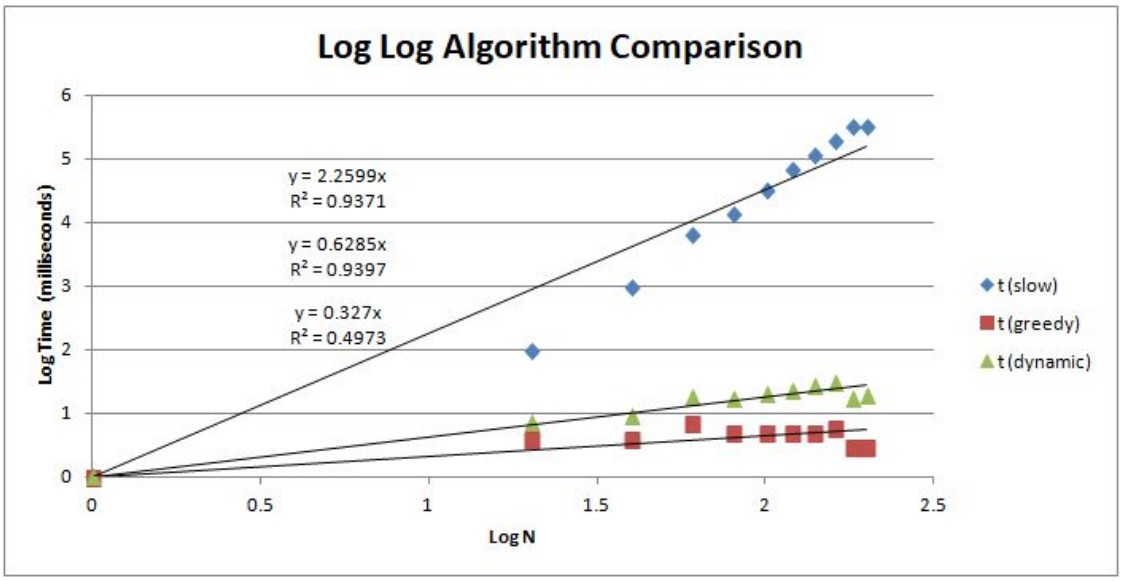




Runtime Analysis for changedp:
The data collected from the tests of the changedp algorithm appears somewhat scattered. While the test of the v_1 coin system appears to fit better with the quadratic equation generated in Excel, the trendline for the v_2 and v_3 coin systems actually appears more linear. Despite this, these plots still fit with our our theoretical runtime of $\Theta(n * k)$, as it is a *pseudo-quadratic* run time, and the correlation of the regression will vary depending upon the test case with the value k.

5. Create a single log-log plot that contains the running time data from all three algorithms. Fit trend lines to each data set. Compare the running times of the different algorithms.





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

The greedy algorithm would be a bit better on performance as it is $\Theta(n)$ as opposed to $\Theta(n*k)$. For correctness, both algorithms would essentially be equal. Given that all elements of V are in the subset of the next element and that each denomination is greater than the sum of its lesser denominations, there wouldn't be an issue with suboptimal solutions. For each calculation, the largest denomination would be selected. As the next smaller denomination is a subset of the denomination selected, there would be no potential errors where the next selected denomination is a suboptimal solution that would not result in the minimum amount of coins. Essentially, no matter what amount we select, as all elements greater than 1 are a multiple of 3, the amount is divisible by the appropriate denomination, with the remainder divisible by either one of the other multiples of 3 in the array or 1.

---

7. Give at least three examples of denominations sets V for which the greedy method is optimal. Why does the greedy method produce optimal values in these cases?

Example 1: One coin system that may be optimal for all denominations would be based on powers of 2 (2^0, 2^1, 2^2, ..., 2^n). For example, coin denominations [1, 2, 4, 8, 16, 32, 64] might give optimal results for amounts tested. This would be due to the fact that any higher denomination is always evenly divisible by any lower denomination. Such divisibility, without the possibility for remainders, results in the greedy algorithm's choice being optimal at any given moment. The immediately optimal solution will be the optimal overall solution, without need to consider the possibility that other denominations could be a better alternative, as with other coin systems where that occurs. While it may be obvious, note that a coin of value 1 (if coins must be of integer value) will always be necessary.

Example 2: The system [1, 13, 52] should also give optimal results for amounts tested with a greedy algorithm, for the same reason expounded upon in Example 1.

Example 3: If it is acceptable for a coin to not be an integer, then the system [½, 3, 27, 54] should also give optimal results for amounts tested with a greedy algorithm, again following the same principle of divisibility previously discussed in Example 1.