Group 28:
Edwin Ramsay Grove
Michael Dambacher
Vijay Kumar

Project 2 : Coin Change Algorithms

1. Give a detailed description of the pseudocode for each algorithm

DP - Coin-Change Algorithm:

We start by creating two array. The first array *OPT[v + 1]* will keep track of the minimal amount of coins needed to make change for a particular value v_i. The second table will keep track of the array position for the last coin added to a particular v_i. We can use the second table to figure out
which coins were used to create change.

We then create array *C[]* that will count how many of each coin was used to make change for V Next we initialize *OPT[0] to 0* and *COINCH[0] to -1*. This is because a solution for a value of 0 is 0 coins and the minimum coins used for that value will never be below -1.

We open a for loop (i = 1 to v), in which we initialize all values greater than 1 in OPT to infinity, as we are seeking the minimum number of coins, and infinity will always be its ceiling. We also initialize all values greater than 1 of COINCH to -1, as the minimum number of coins will always be greater than -1.

Moving on, we being a set of nested for loops —

The ***Outer For Loop*** is *j = 0 to the number of coins* in the input array (*coins.length()*)

The ***Inner For Loop*** is *i = 1 to v* (the value for which we are seeking change): within this loop we:

- check if i is greater than or equal to the value at *coins[i]*. If *i* is ≥, then we check to see if
adding *coin[j]* to a possible solution reduces the total coins to make change.
- if adding the coin reduces the amount of change, we assign *OPT[i] = OPT[i-coins[j] + 1*
- in addition, we record the number of coins of the specific value using *COINCH[i] = j*

To close, we must now record the number of coins used of each denomination into our empty input vector. We do this by running a while loop that runs until *v = = 0.*

The loop operates by adding the specific number of coins of each denomination from the COINCH array into the input array. It decrements *v* by the appropriate amount. Once *v* is equal to zero, the input array will now contain all the correct integers, representing the number of coins for each denomination.

Greedy Coin-Change Algorithm:

Compared to our DP algorithm, the Greedy algorithm is fairly simple.

The function receives as arguments, an integer variable representing the value we are seeking (A), and a vector of ints containing all the denominations we can work with (coins).

To begin, we create an integer variable *numCoins* and assign in the value 0. Next, our algorithm records the size of coins and assigns it to an integer *i.*

Our while loop opens and is conditional upon *A* being greater than 0.

If *A* is greater than the amount of the greatest denomination, we add the denomination and increment the count of the integer *numCoins*. This incrementing will give us the final count of coins used.

During the while loop, if *A* is not ≥ the value at *coins[i]*, we decrement *i* to check the next lower denomination, adding it to our solution if necessary.

The loop continues until *A == 0*.

We are now able to return *numCoins* with the appropriate count of coins.

2. Calculate the asymptotic running time for each algorithm.

The asymptotic running times:

Dynamic Programming algorithm
O(nV) pseudo-polynomial

Greedy algorithm
O(n)

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

For our table we utilize two independent arrays. One array keeps track of the minimum number of coins it takes to make change for every integer up to the value V. This allows us to reference values v_i when we are attempting to find the minimum number of coins to make v_k (where 0 . If we are trying to find the number of coins used to make change for v_k we first check to see if the coin we are attempting to make change with is less than the value. If it is we then check if adding the coin would reduce the total number of coins required to make the change if it is we enter this new value into the array, if it is not then we keep the current value.

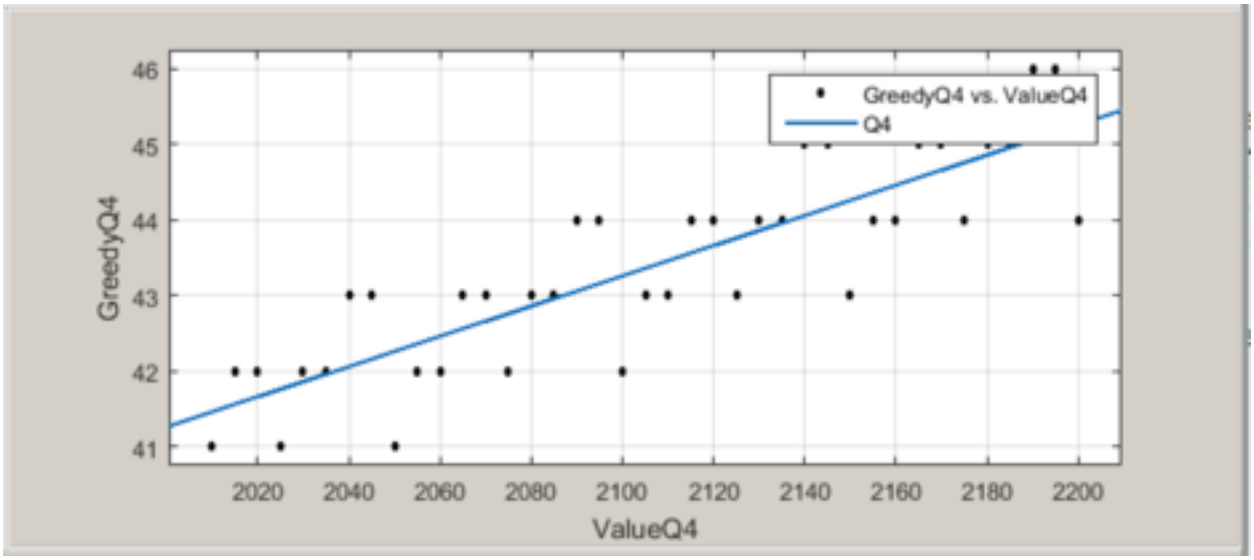
The other array keeps track of the last coin that was used in order to make change for v_k and allows us to keep track of all the coins used for v_k through an algorithm executed later in the function.

Our algorithm begins by initializing V+1 elements in each array. This allows for our base case.

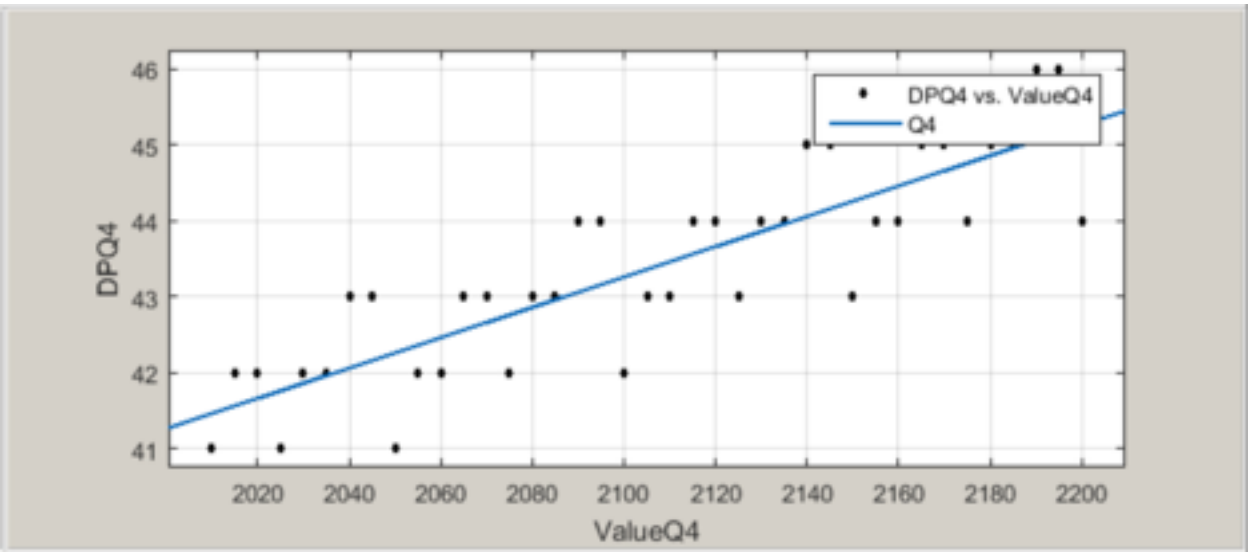
The next pair of nested for-loops is what populates the table data. The outer loop increments through the array holding the coin values while the inner loop increments through both the coinCh array and the OPT array. This part of the algorithm makes sure that every combination is checked when finding the minimum number of coins

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?

Greedy:



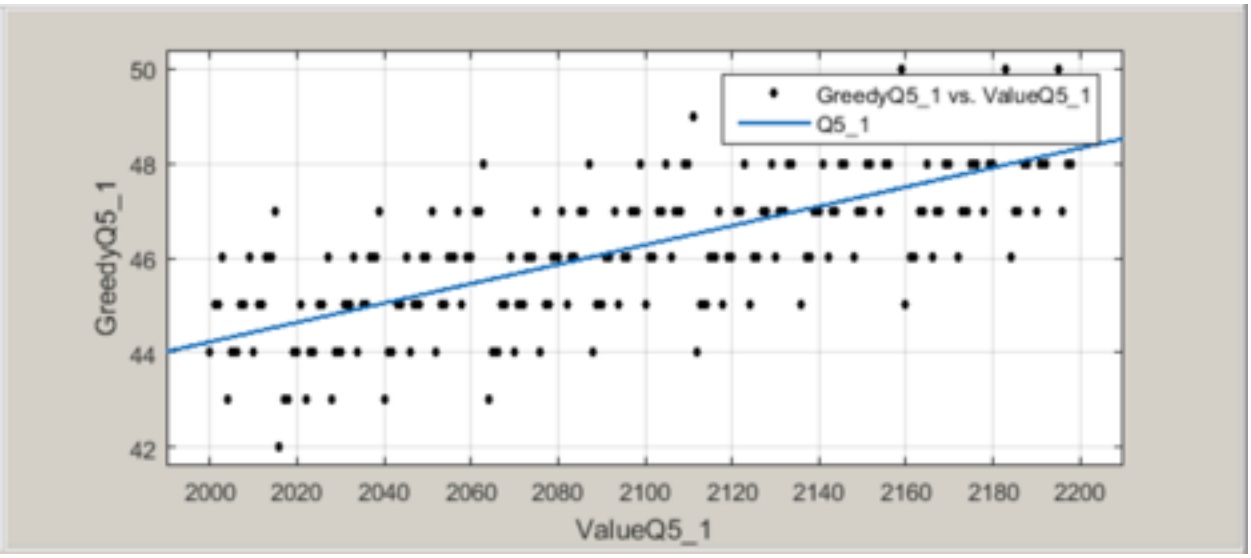
Dynamic Programming:



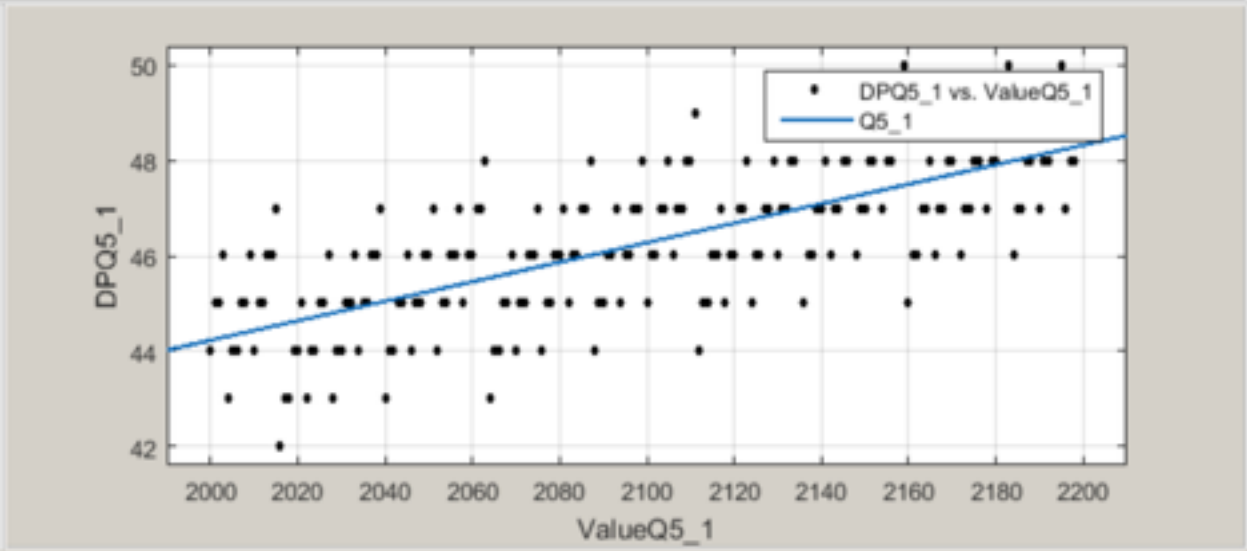
Conclusion: Both the greedy and the dynamic programming algorithms provide the same results for this data set.

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]. Plot the number of coins as a function of A for each algorithm. How do the approaches compare?

Set: V1 = [1, 2, 6, 12, 24, 48, 60]
Greedy:



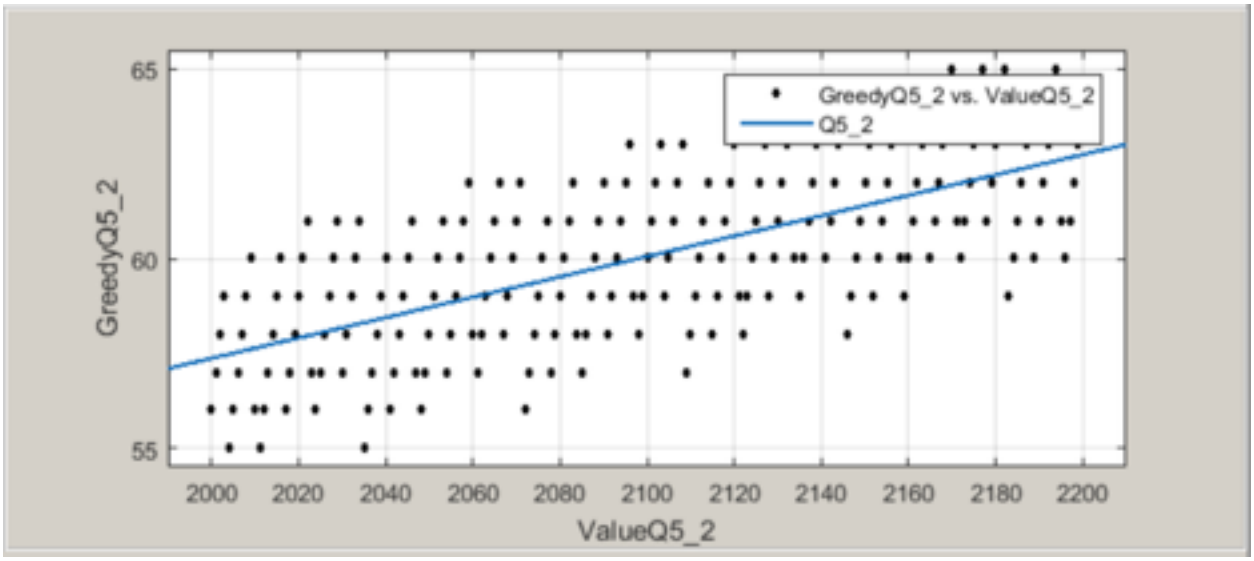
DP:



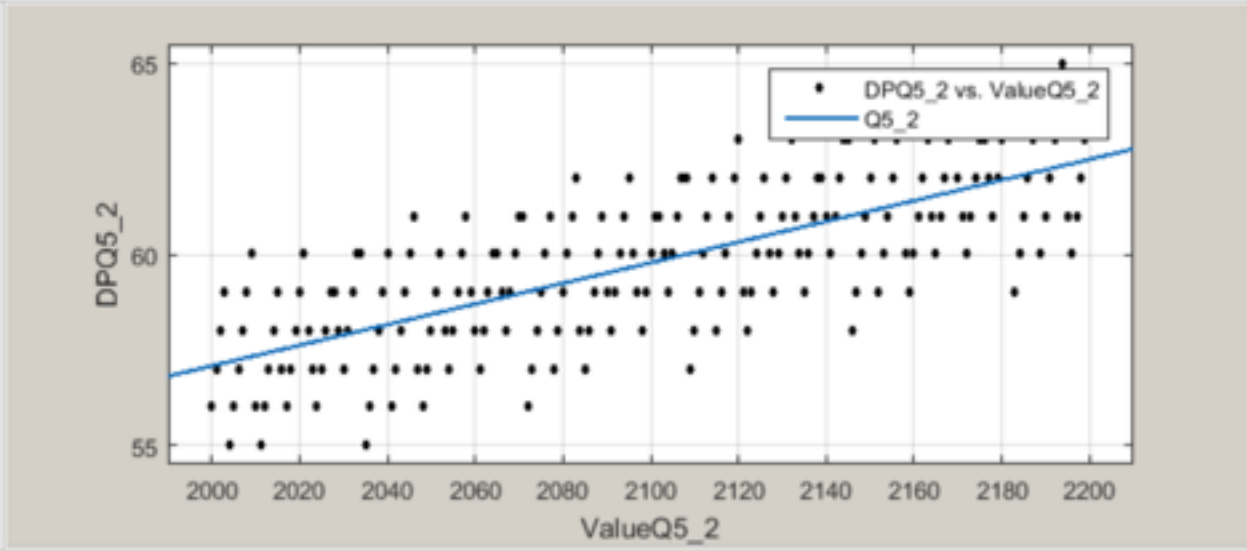
Conclusion: Once again both the greedy and the dynamic programming algorithms provide the same result for this data set.

Set: V2 = [1, 6, 13, 37, 150]

Greedy:



DP:

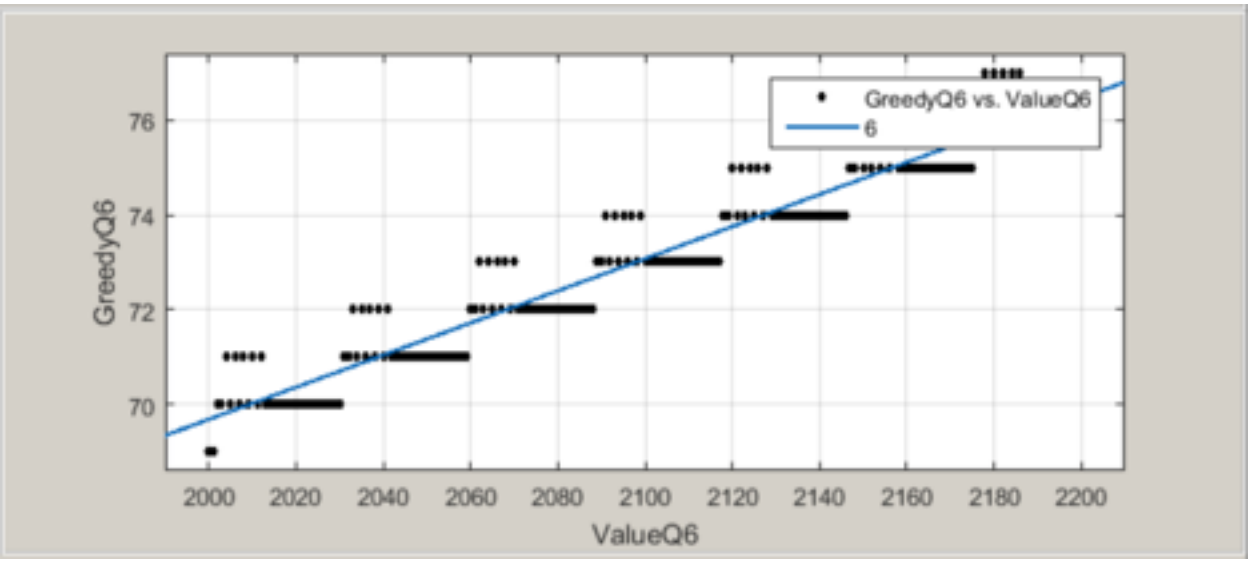


Conclusion: On this set of coins the results differ for some V values. It could be postulated that, given the set of coins for this problem relative to the two previous problems, that if coin c_i is an integer multiple of c_{i-1} , and this pattern holds true for all coins $c_1, c_2, ..., c_n$ that the greedy

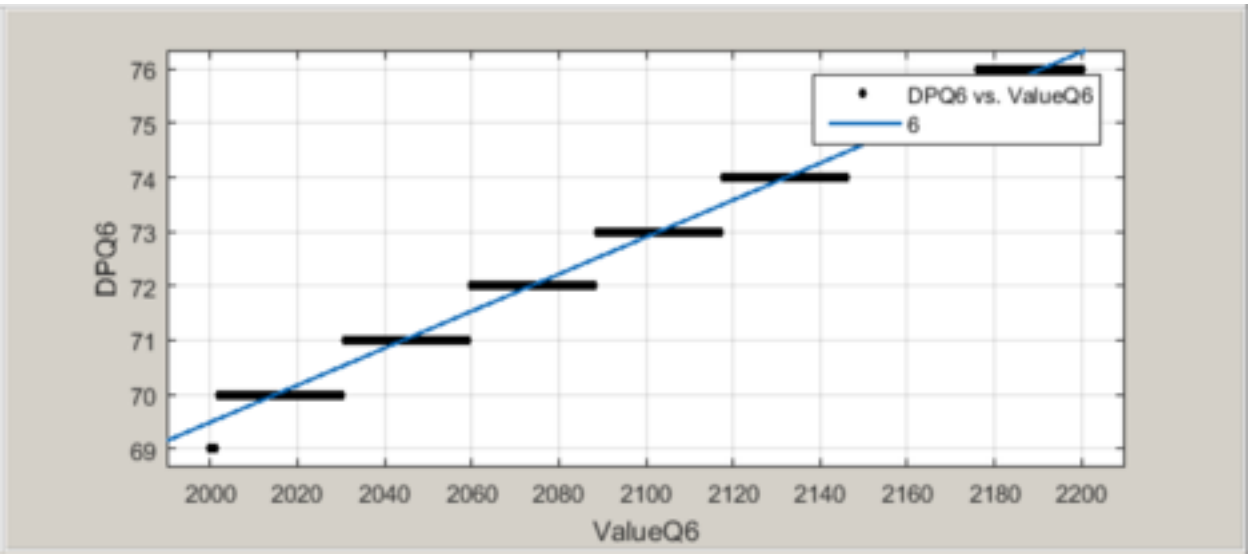
solution will give an optimal result. This pattern does not hold for this particular set of coins and therefore the greedy solution does not always give an optimal solution. The pattern described here is not exhaustive of all coin combinations that may give an optimal solution.

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. Plot the number of coins as a function of A for each algorithm.

Greedy:



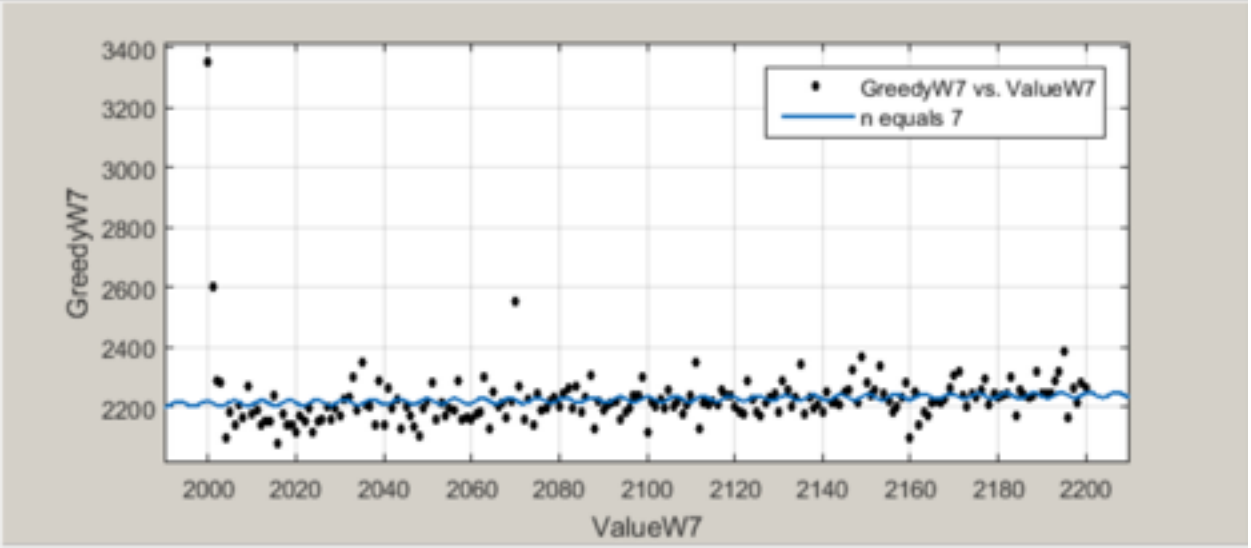
DP:



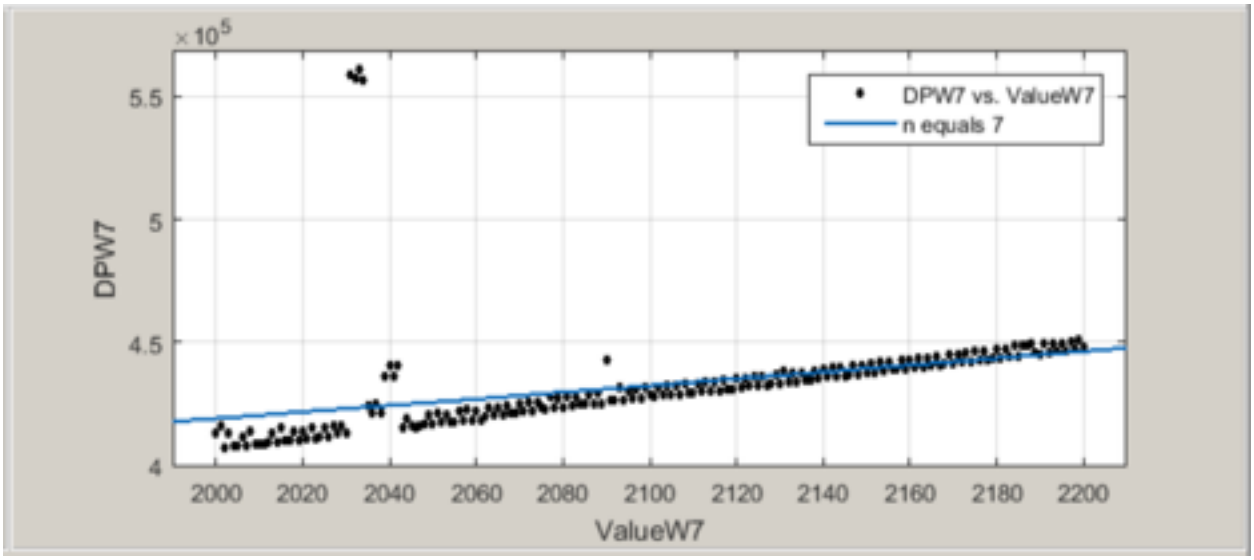
Conclusion: This problem once again has different results for the DP algorithm compared with the greedy. Our postulation that any given coin must be an integer multiple of the previous coin, is further solidified on this test.

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.

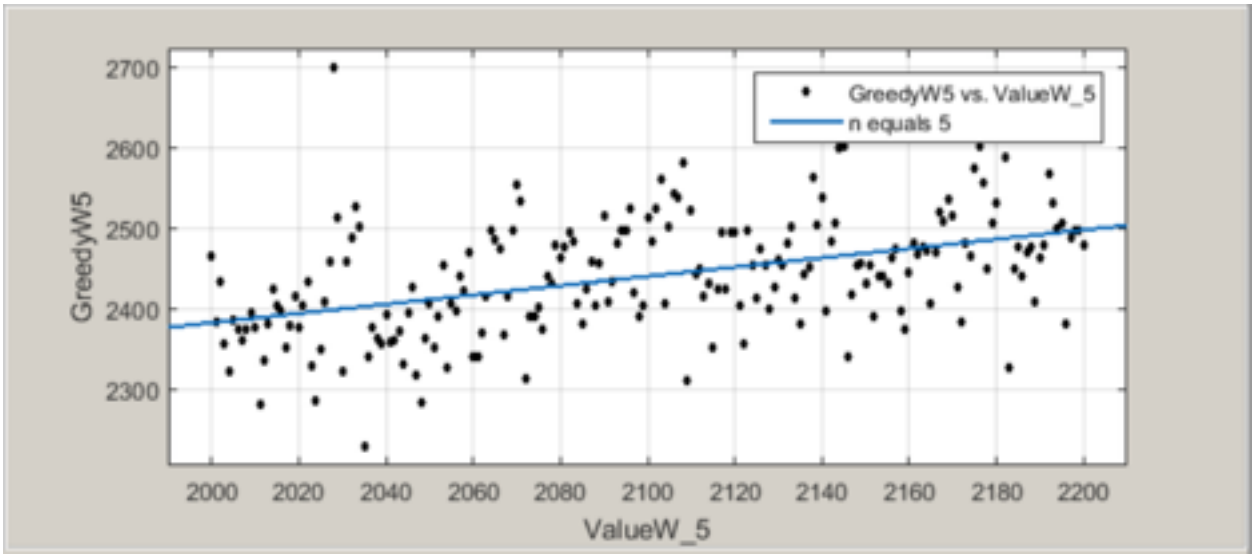
Greedy : Running Time for SET: [1, 2, 6, 12, 24, 48, 60]



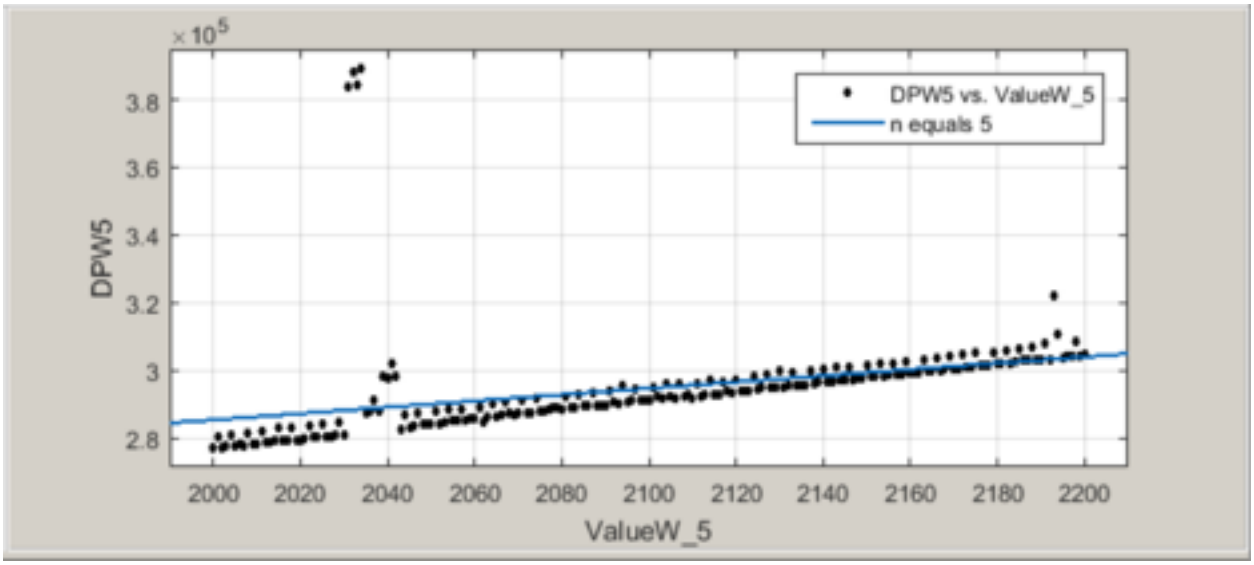
Dynamic Programming: : Running Time for SET: [1, 2, 6, 12, 24, 48, 60]



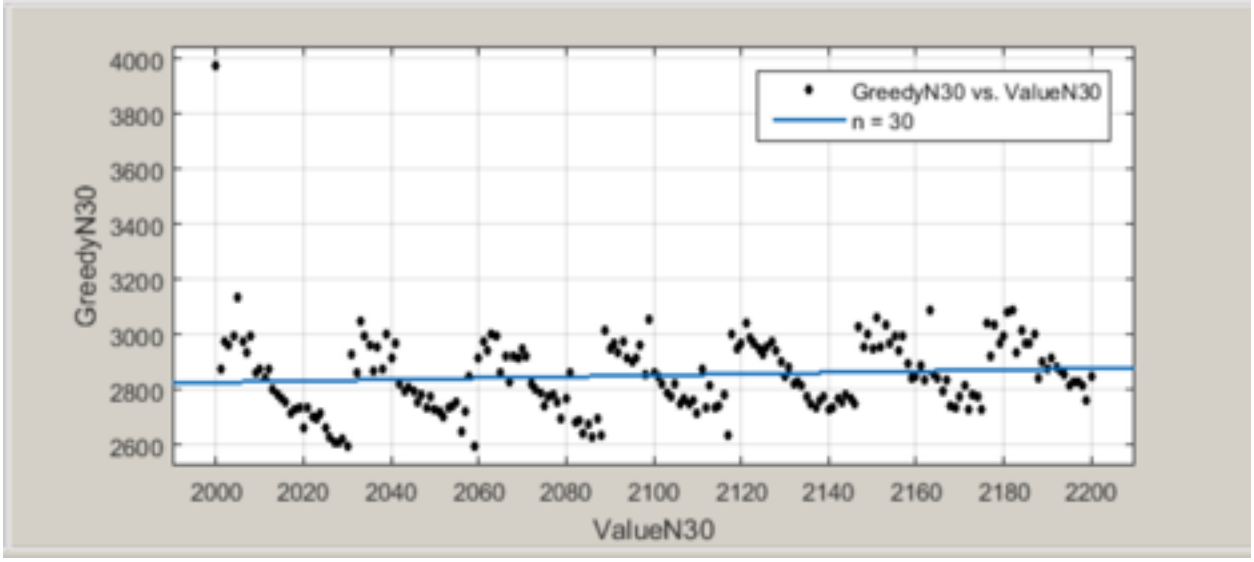
Greedy: Running Time for SET: [1, 6, 13, 37, 150]



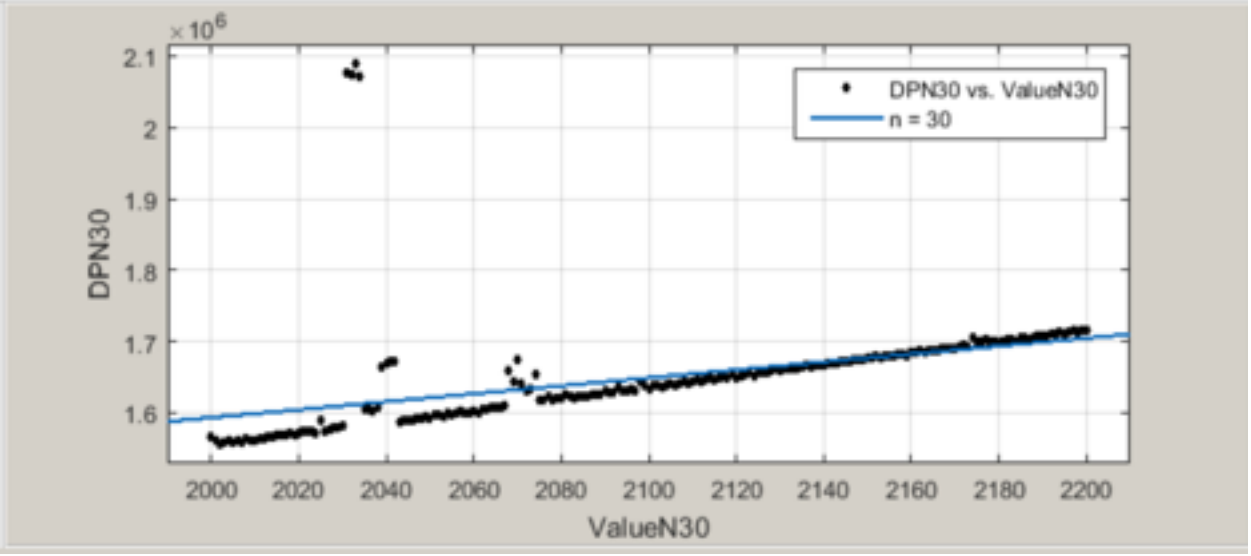
Dynamic Programing: Running Time for SET: [1, 6, 13, 37, 150]



Greedy : Running Time for SET: [1, 2, 4, 6, 8, 10, 12, …, 30]



Dynamic Programming: Running Time for SET: [1, 2, 4, 6, 8, 10, 12, …, 30]

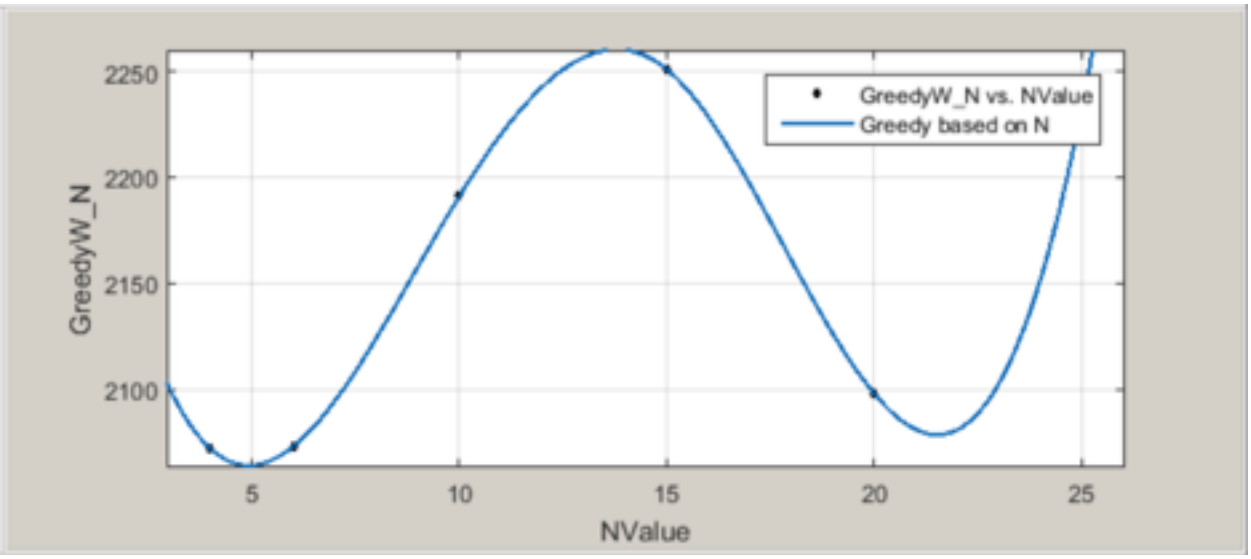


Conclusion: The running times of each set seemed contingent upon the values within the set. For example, when the total V (value) was divisible by the greatest value in the set, the running time decreased significantly, especially for the Greedy Algorithm. The Dynamic Programming

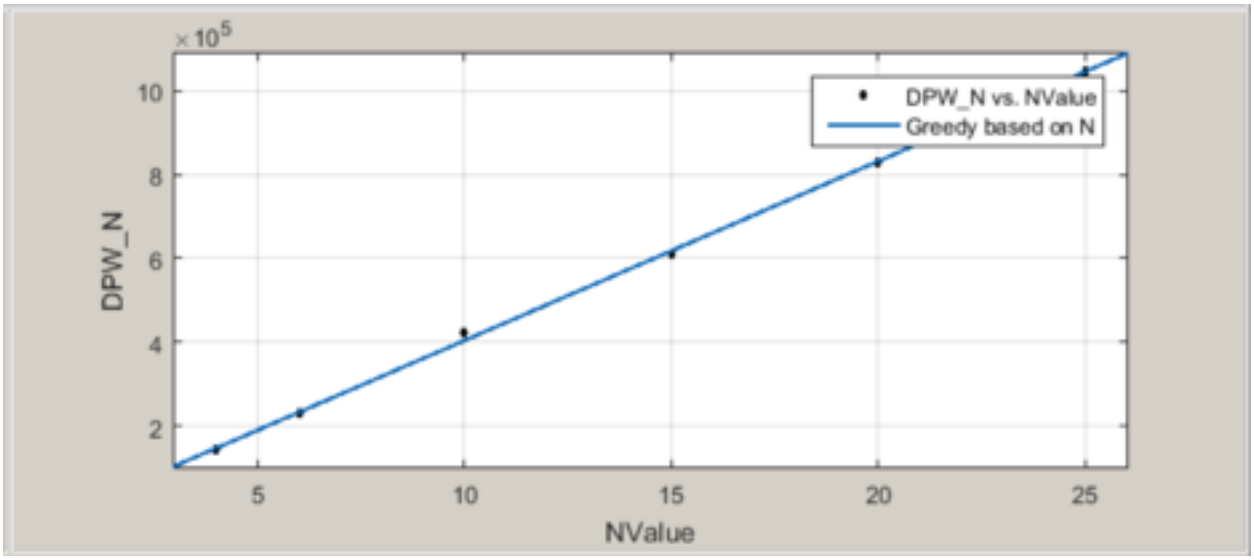
running times always grew in a linear fashion, with a distinct connection to the size of V. Notably, the DP algorithm is anticipated to run in a pseudo-polynomial time. However, the fact that our O(nV) is more similar to O(5V) or O(7v), depending on the number of coins, and since our n << V (n is much much less than V) the graphs shows a linear growth. The Greedy Algorithm running time collected around a slowly ascending slope, but did not tightly bind the line in any way. The plots reflected a scattered spread of running times that seemed to be associated to V's divisibility by the set's highest value.

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?

Greedy : n = 4, 6, 10, 15, 20, 25



Dynamic Programming: n = 4, 6, 10, 15, 20, 25



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.

Throughout our testing we devised a theorem that said if any c_i is an integer multiple c_{i-1} of for all coins $c_1, c_2, ..., c_n$, the greedy solution would give an optimal solution. This theorem does not cover all cases where the greedy algorithm gives an optimal solution but it does cover this particular case since each coin value in this set is an integer multiple of the coin before it.

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

If any c_i is an integer multiple of c_{i-1} for all coins $c_1, c_2, ..., c_n$, the greedy solution will give an optimal solution.