Students: Jesse Thoren, Jason Goldfine-Middleton, Shawn Hillyer (Group 11)
Assignment: Project 2: Coin Change

Question 1

Changeslow Pseudocode:

```
//inputs: array of denominations available (D), amount to make change for (a)
//outputs: array of coins used for each denomination (minarr), minimum amount of coins
(minamt).
Changeslow:
    Initialize minarr to zero for each denomination.
    Initialize minamt to a //Assumes that nothing but "1" coins are used.

    If the amount to make change for is zero:
        return minarr, minamt

    Loop through each denomination (D[i]) doing:
        If the denomination is greater than the total remaining change, skip.
        If the denomination is less than or equal to the total remaining change, then:
            Get results from recursion for a-D[i] //res.minarr, res.minamt
            If minamt > res.minamt+1 we have a better solution, then
                //Account for the coin we took away for recursion
                Set minamt to res.minamt+1
                Set minarr to res.minarr, but increment res.minarr[i] by 1

    return minarr, minamt
```

Changeslow Asymptotic Running Time:

It's a bit difficult to find a solid asymptotic running time for changeslow because it appears to depend on both the length of the denomination array D, and the values contained in the denomination array.

That is: $T(n) = T(n-D[0]) + T(n-D[1]) + T(n-D[2]) + \ldots + T(n-D[len(D)-1]) + O(1)$.
However, we know that this is faster for higher denomination values, because more of the amount is cut off each time. We can calculate the worst case scenario by assuming that the values for
$1 = d_1 < d_2 < \ldots < d_{len(D)}$ are consecutive positive integers: $1 < 2 < \ldots < len(D)$

Then the recursion becomes $T(n) = T(n-1) + T(n-2) + T(n-3) + \dots + T(n-\text{len}(D)) + O(1)$.

We can further simplify by running more of the leading terms through the recursion and observing the leading coefficients and the coefficients of the final term.

Calculating recursion for $T(n-1)$:
$T(n) = (T(n-2) + T(n-3) + \dots + T(n-\text{len}(D)-1)) + (T(n-2) + \dots + T(n - \text{len}(D))) + O(1)$.
$T(n) = 2T(n-2) + 2T(n-3) + \dots + 2T(n-\text{len}(D)) + T(n-\text{len}(D)-1) + O(1)$.

Calculating recursion for $T(n-2)$:
$T(n) = 2(T(n-3) + T(n-4) + \dots + T(n-\text{len}(D)-2)) + 2(T(n-3) + T(n-4) + \dots + T(n-\text{len}(D))) + T(n-\text{len}(D) - 1) + O(1)$.
$T(n) = 4T(n-3) + 4T(n-4) + \dots + 4T(n-\text{len}(D)) + 3T(n-\text{len}(D)-1) + 2T(n-\text{len}(D)-2) + O(1)$.

Calculating recursion for $T(n-3)$:
$T(n) = 4(T(n-4) + T(n-5) + \dots + T(\text{len}(D)-3)) + 4(T(n-4) + T(n-5) + \dots + T(n-\text{len}(D))) + 3T(n-\text{len}(D)-1) + 2T(n-\text{len}(D)-2)$.
$T(n) + 8T(n-4) + 8T(n-5) + \dots + 8T(n-\text{len}(D)) + 7T(n-\text{len}(D)-1) + 6T(n-\text{len}(D)-2) + 4T(n-\text{len}(D)-3) + O(1)$.

Note that each time we process a recursion, the leading coefficient is doubled, but it seems like this behaviour will stop as soon as we get to $T(n-\text{len}(D))$. The coefficients that follow this are the between 2^n and a progressively larger power of 2. Hence eventually we will have a recursion of the form:

$T(n) = 2^{\text{len}(D)-1}T(n-\text{len}(D)) + (2^{\text{len}(D)-1}-2^1)T(n-\text{len}(D)-1) + (2^{\text{len}(D)-1}-2^2)T(n-\text{len}(D)-2) + \dots + (2^{\text{len}(D)-1}-2^{\text{len}(D)-2})T(n-2\text{len}(D)+1)$.

Then:
$T(n) = (2^{\text{len}(D)}-2^1)T(n-\text{len}(D)-1) + (2^{\text{len}(D)}-2^2)T(n-\text{len}(D)-2) + \dots + (2^{\text{len}(D)}-2^{\text{len}(D)-2})T(n-2\text{len}(D)+1) + (2^{\text{len}(D)}-2^{\text{len}(D)-1})T(n-2\text{len}(D))$.

Then:
$T(n) = (2^{\text{len}(D)+1}-2^2-2^1)T(n-\text{len}(D)-2) + \dots + (2^{\text{len}(D)+1} - 2^{\text{len}(D)-1}-2^1)T(n-2\text{len}(D)) + (2^{\text{len}(D)+1}-2^{\text{len}(D)}-2^1)T(n-2\text{len}(D)-1)$.

Note that even after the initial len(D) terms, the leading coefficient is still $2^{\text{number of recursions calculated}}$ - some constant. Hence, we can expect that the coefficient when we are calculating $T(1)$ will be 2^n - some constant, which implies that $T(n) = O(2^n)$.

Changegreedy Pseudocode:

```
//inputs: array of denominations available (D), amount to make change for (a)
//outputs: array of coins used for each denomination (minarr), minimum amount of coins
(minamt)
Changegreedy:
    Initialize minarr to 0 for each denomination in D.
    Initialize minamt to 0.

    Loop through each denomination, starting with the largest coin first.
        While a is greater than or equal to the current denomination:
            Add one to the current denomination in minarr
            Add one to the minamt
            Subtract the value of the current denomination from a

    Return minarr, minamt
```

Changegreedy Asymptotic Running Time:

In the worst case scenario, we have to loop through each element in the denomination array once, and then use only pennies to make change for a, because all the denominations in the denomination array happen to be too large. Hence the asymptotic running time of the greedy algorithm is O(a) + O(len(D) = O(a + len(D)) = O(n), so the greedy algorithm runs in linear time.

Changedp Pseudocode:

```
//inputs: array of denominations available (D), amount to make change for (a)
//Outputs:array of coins used for each denomination (minarr), minimum amount of coins
(minamt)
Changedp:
    Initialize a global array to memorize results.
    Call the helper function to compute using dynamic programming.

Changedphelper:
    Initialize minarr to zero for each denomination.
    Initialize minamt to a //Assumes that nothing but “1” coins are used.

    If the amount to make change for is zero:
        return minarr, minamt
    If the amount has been memorized previously:
        return the memorized minarray and minamt

    Loop through each denomination (D[i]) doing:
        If the denomination is greater than the total remaining change, skip.
        If the denomination is less than or equal to the total remaining change, then:
            Get results from recursion for a-D[i] //res.minarr, res.minamt
            If minamt > res.minamt+1 we have a better solution, then
                //Account for the coin we took away for recursion
                Set minamt to res.minamt+1
                Set minarr to res.minarr, but increment res.minarr[i] by 1
        Store the best result in the memorized array.

    return minarr, minamt
```

Changedp Asymptotic Running Time:

We use the fact that this problem has an optimal substructure property to split the problem into len(D) subproblems, one for each of the denominations in D. Note that the maximum recursion depth of the “longest” is equal to the amount a, and occurs in the case when we choose only pennies.

Because the algorithm memorizes previously calculated solutions in an array, we can solve each of the len(D) subproblems in at most n iterations, so the asymptotic running time of the algorithm is O(len(D)*a), where D is the array of denominations available to use and a is the amount we are making change for.

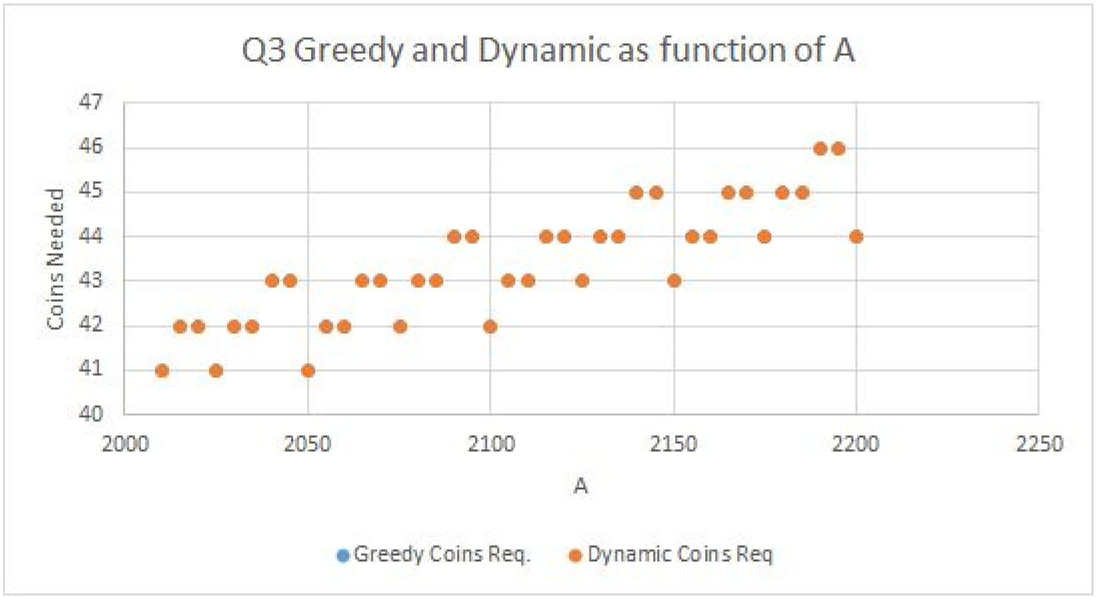
Question 2

Our approach to the dynamic programming algorithm is a top down approach for calculating individual subproblems, but the dynamic programming array is actually filled from smaller amounts upwards.

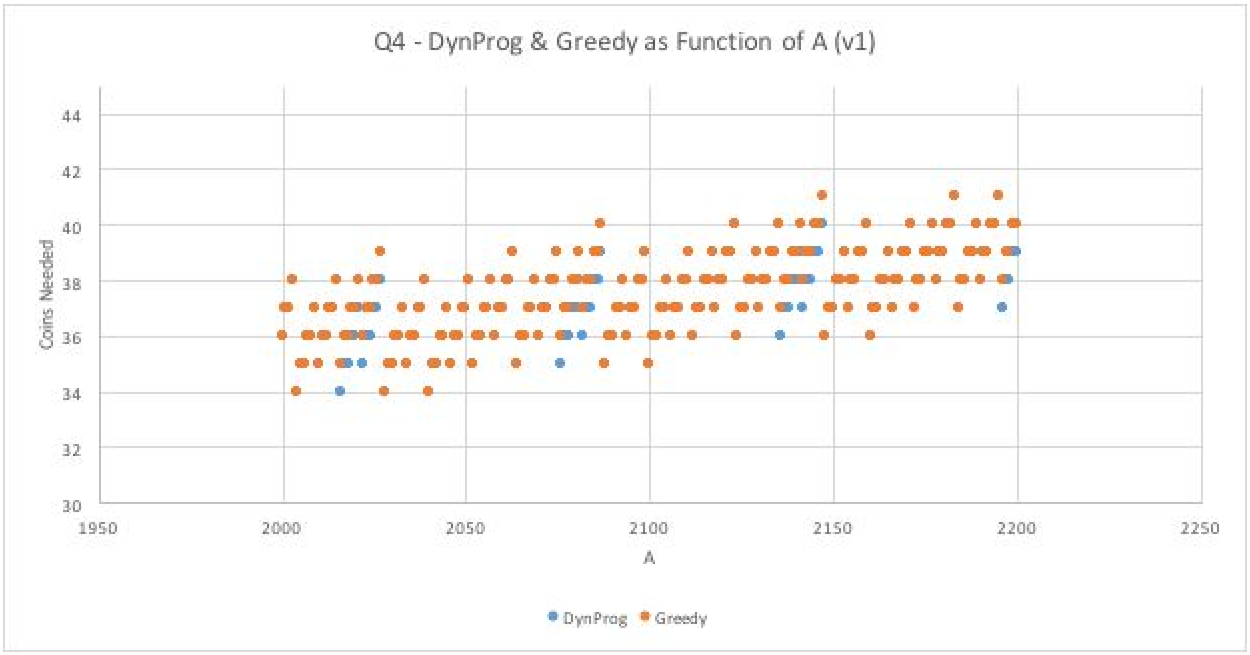
The recursion called inside changedphelper creates a large call stack by calling successively smaller values, before finally being able to compute a value for an amount of 1, which gets stored in the memo table, and is returned up to the next recursion in the call stack. Because a penny is always an included denomination in the problem, we are assured by the algorithm that in the calculation of any amount = a (where a>=2), the recursion for an amount = a-1 has to be calculated, stored in the memo table, and returned up the call stack first.

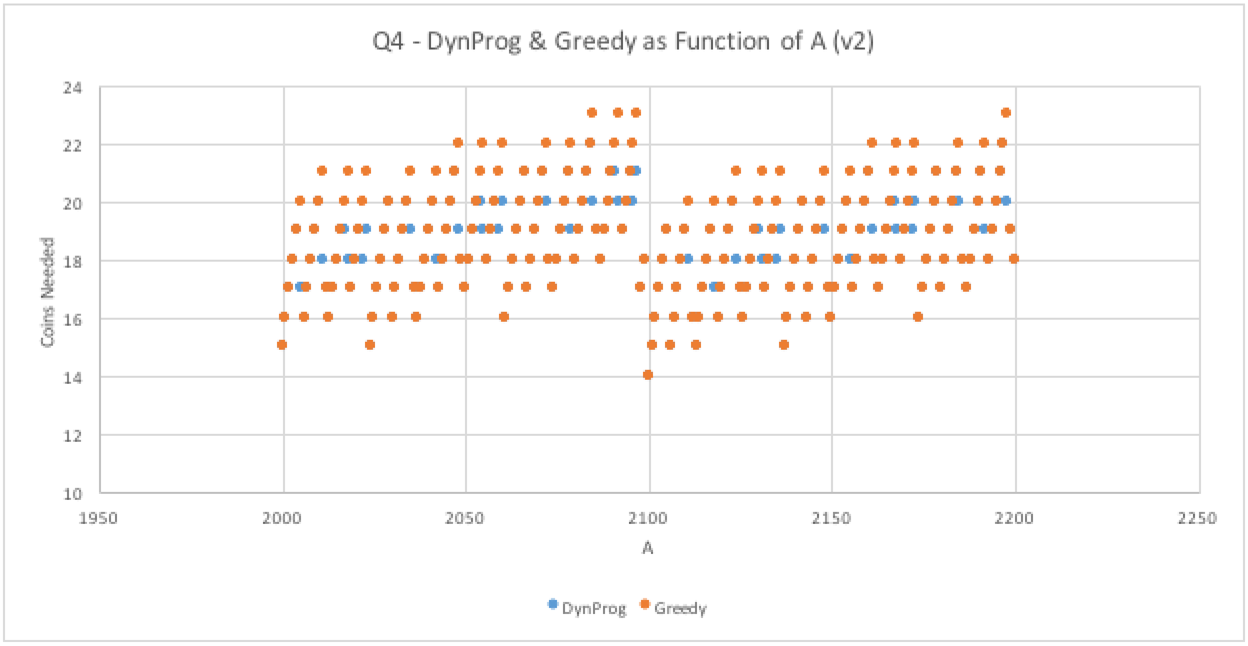
Question 3

The algorithms return the same number of coins for the denominations given. In this case, the greedy algorithm is optimally calculating the number of coins required. Note that the data points for Greedy Coins Req. are not visible in the plot below because they overlap exactly with Dynamic Coins Req. For this particular set of denominations and total change amounts, the greedy function did produce optimal results in every case.



Question 4





What we see here is that the greedy algorithm implementation differs from dynamic programming algorithm at times, suggesting that it won't always produce an optimal solution for these sets of coin denominations. However, for v1, the maximum error in the solution produced by the greedy algorithm was 1, meaning it thought one more coin was necessary in the solution. So in this case, it was fairly optimal. In the case of v2 on the other hand, the greedy algorithm was off by as much as three coins, which is quite unacceptable when the optimal solution is 20 coins or fewer.

Question 5

Figure 1: As A grows, number of coins needed grows. Dynamic Programming and Greedy algorithms have the same return values. For verification, one could more easily inspect the results of running the algorithms as per script ‘p2q6timing.py’, results saved to ‘Q5-changedp-v-standard.csv’ and ‘Q5-changegreedy-v-standard.csv’. Our approach was to simply compare the return value from each algorithm using subtraction and verify that the max value in that column was 0, although this work was down manually after loading the *.csv data into Excel worksheets.

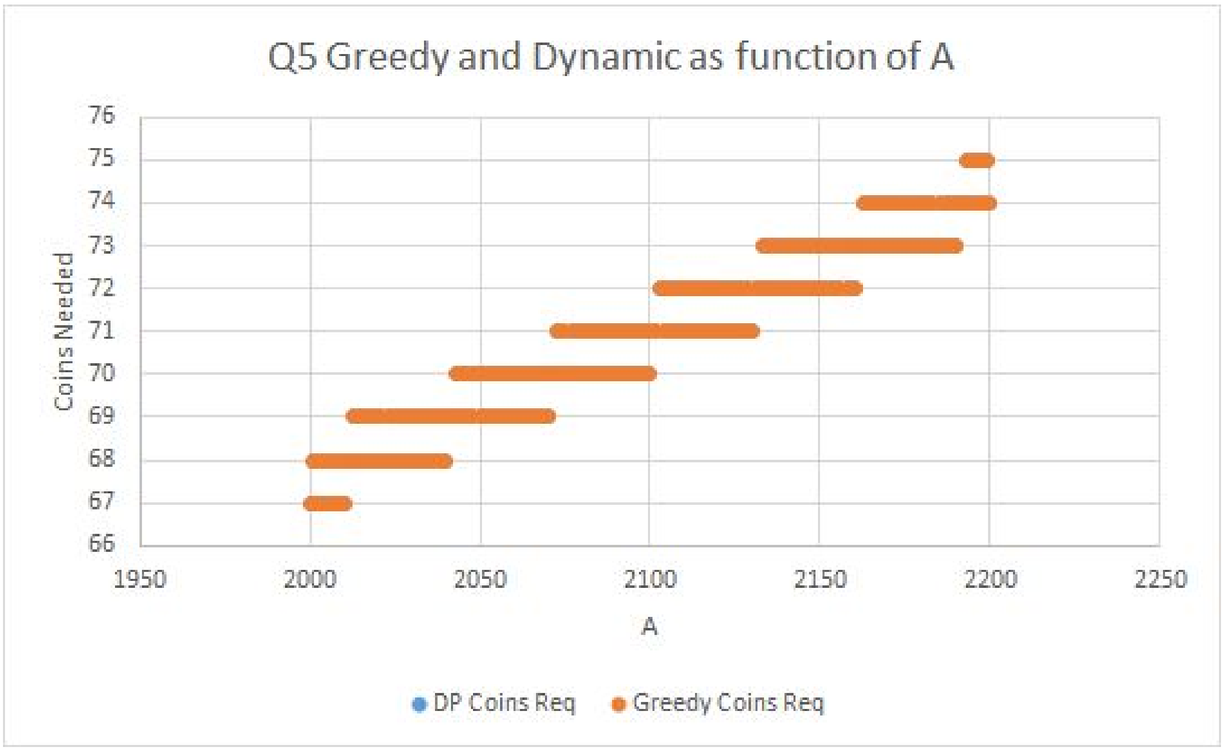
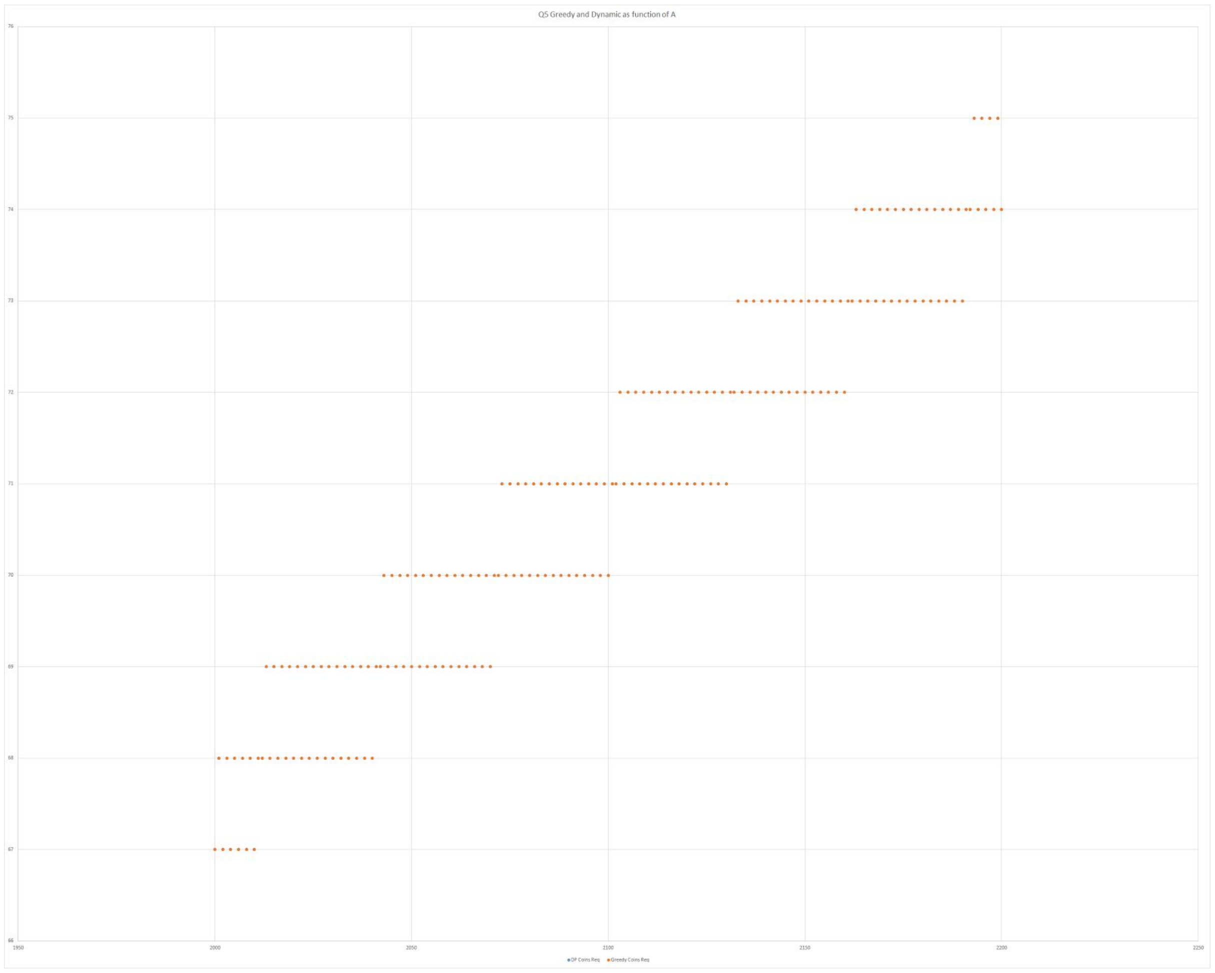


Figure 2: Zoomed out to show the values are actually “ping-ponging” up and down as A grows.



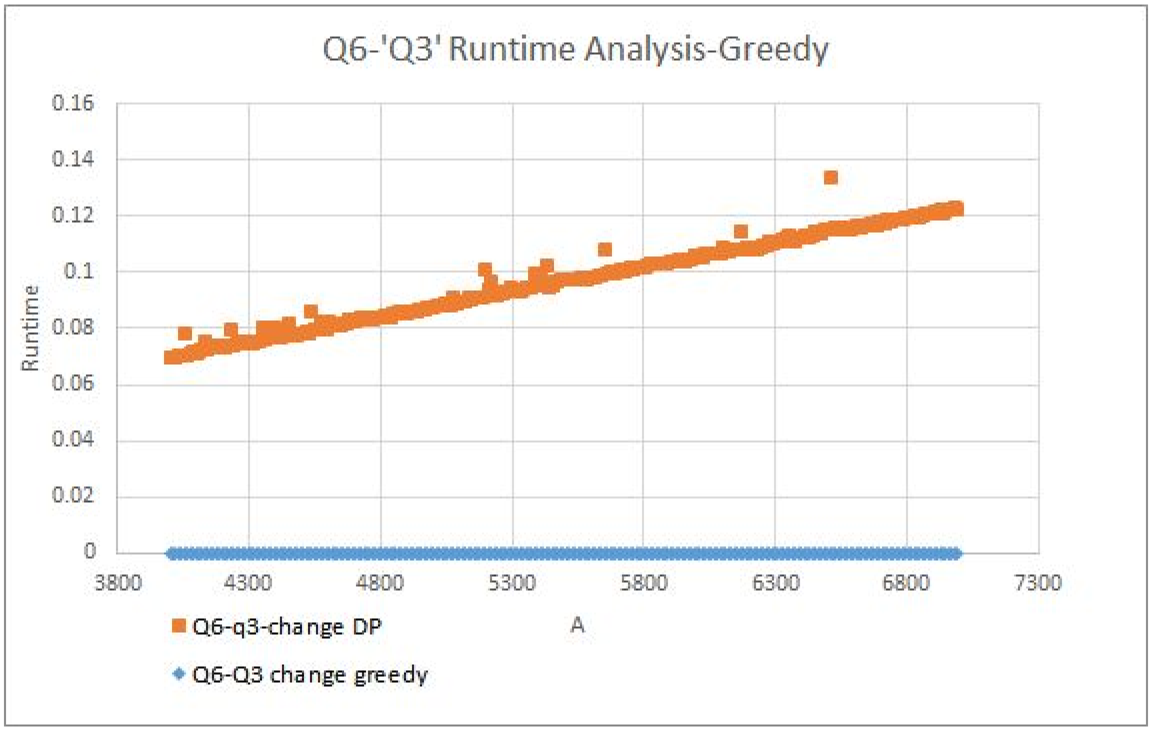
Question 6

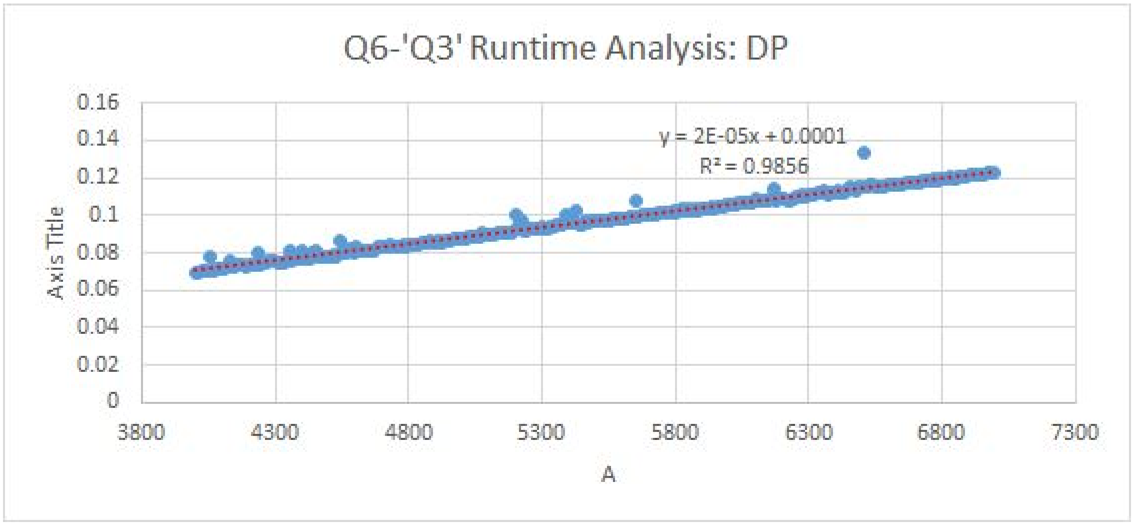
Both functions run in linear time, but the Greedy algorithm is definitely faster than the Dynamic Programming algorithm. This is to be expected as there is no recursion or table to fill out and use for lookups of prior values. The DP algorithm has a tighter r^2 value and generally had less variance and outliers.

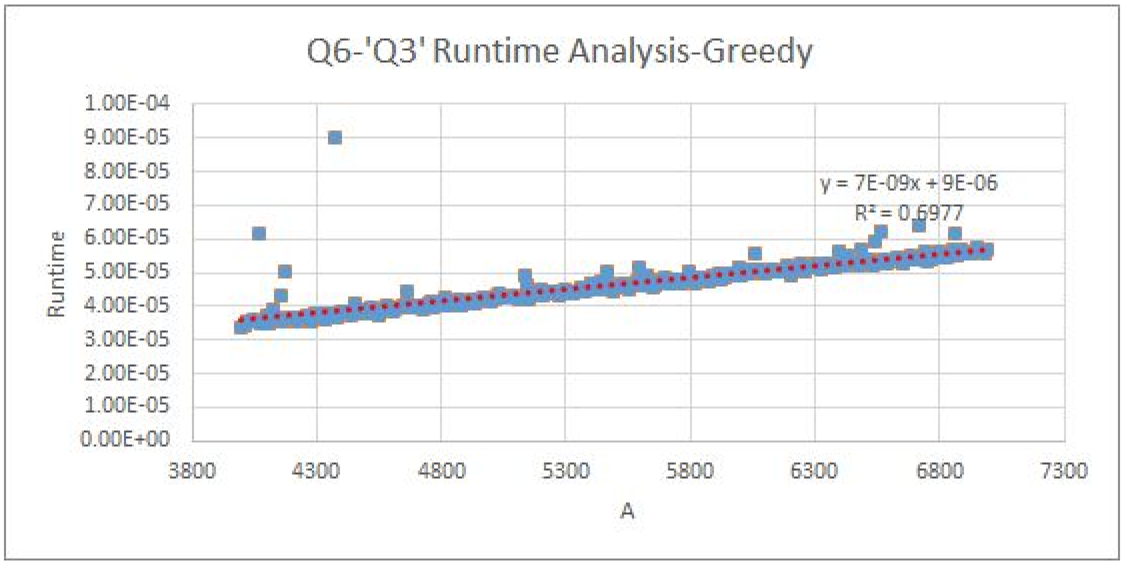
Runtime Analysis of data set from Problem 3

Both algorithms run in linear time, although the greedy algorithm has a much smaller coefficient factor. As expected, the naive greedy approach tends to shortcut a lot of the work that the dynamic approach must do to find a solution. Since we know that the greedy algorithm produced optimal solutions for the particular input set we were given for problem 3, it would serve us well to use that algorithm here, even though the asymptotic analysis for both functions is O(n).

In the first of the three plots below, you can see the dynamic programming algorithm in orange and the greedy algorithm in blue. While both are linear, the DP algorithm grows much more quickly (although not asymptotically more quickly).

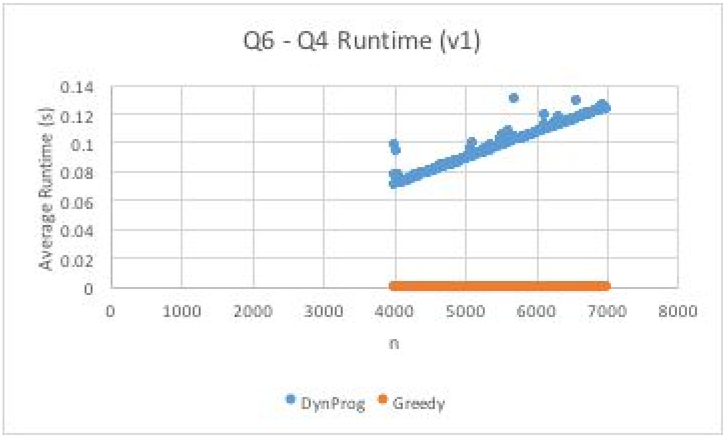


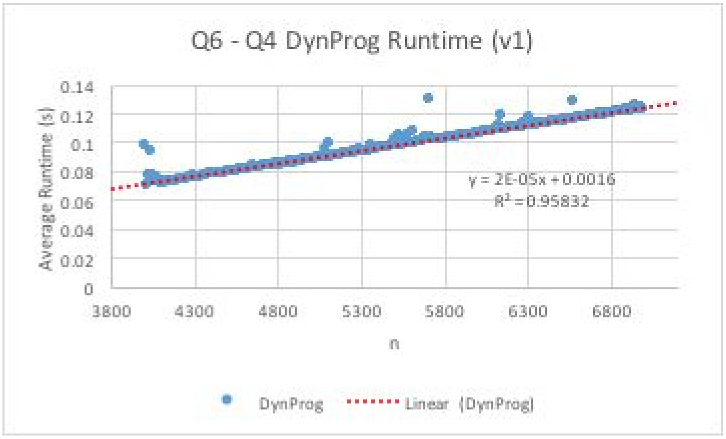


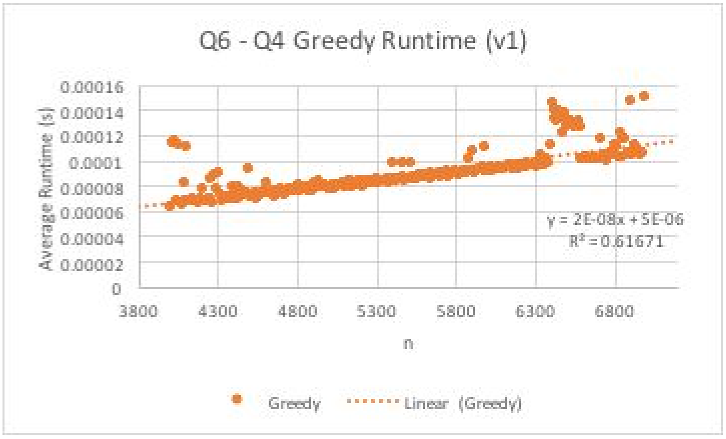


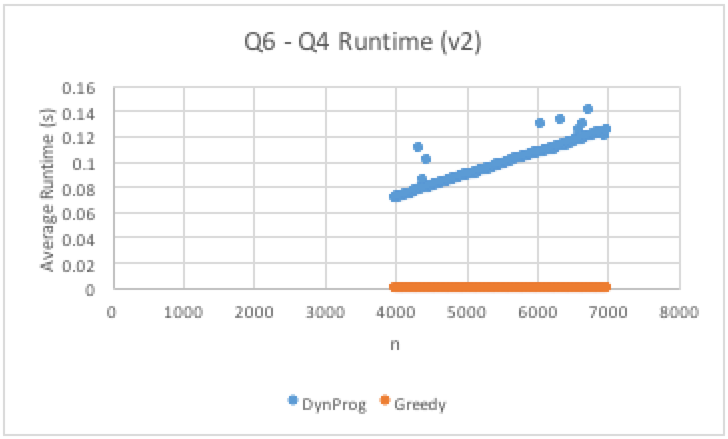
Runtime Analysis of data set from Problem 4

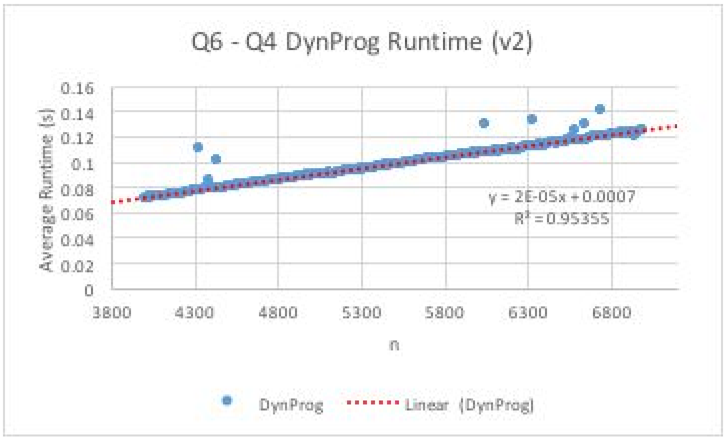
The timing results for both the v1 and v2 vectors from problem 4 present a similar picture. Again we see that both algorithms experimentally ran in time linearly proportional to the amount of change, but the greedy algorithm ran much faster. In this case however, we know that the greedy algorithm did not produce optimal results over the data set in problem 4, suggesting that we would need to forgo speed if we wanted optimal results in every case. The inaccuracy resulting from the use of the greedy algorithm in the case of v1 would be more acceptable perhaps, but with v2 it would be hard to justify.

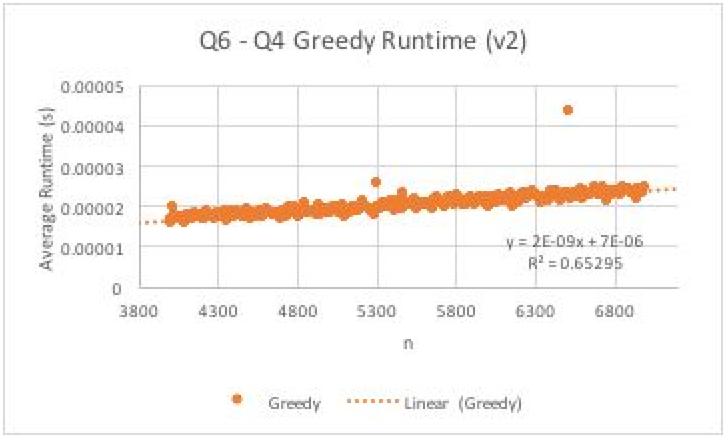

Q6 - Q4 Runtime (v1)
Average Runtime (s)
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
0
1000
2000
3000
4000
5000
6000
7000
8000
n
DynProg
Greedy


Q6 - Q4 DynProg Runtime (v1)
Average Runtime (s)
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
3800
4300
4800
5300
5800
6300
6800
n
y = 2E-05x + 0.0016
R² = 0.95832
DynProg
Linear (DynProg)


Q6 - Q4 Greedy Runtime (v1)
Average Runtime (s)
0.00016
0.00014
0.00012
0.0001
0.00008
0.00006
0.00004
0.00002
0
3800
4300
4800
5300
5800
6300
6800
n
y = 2E-08x + 5E-06
R² = 0.61671
Greedy
Linear (Greedy)

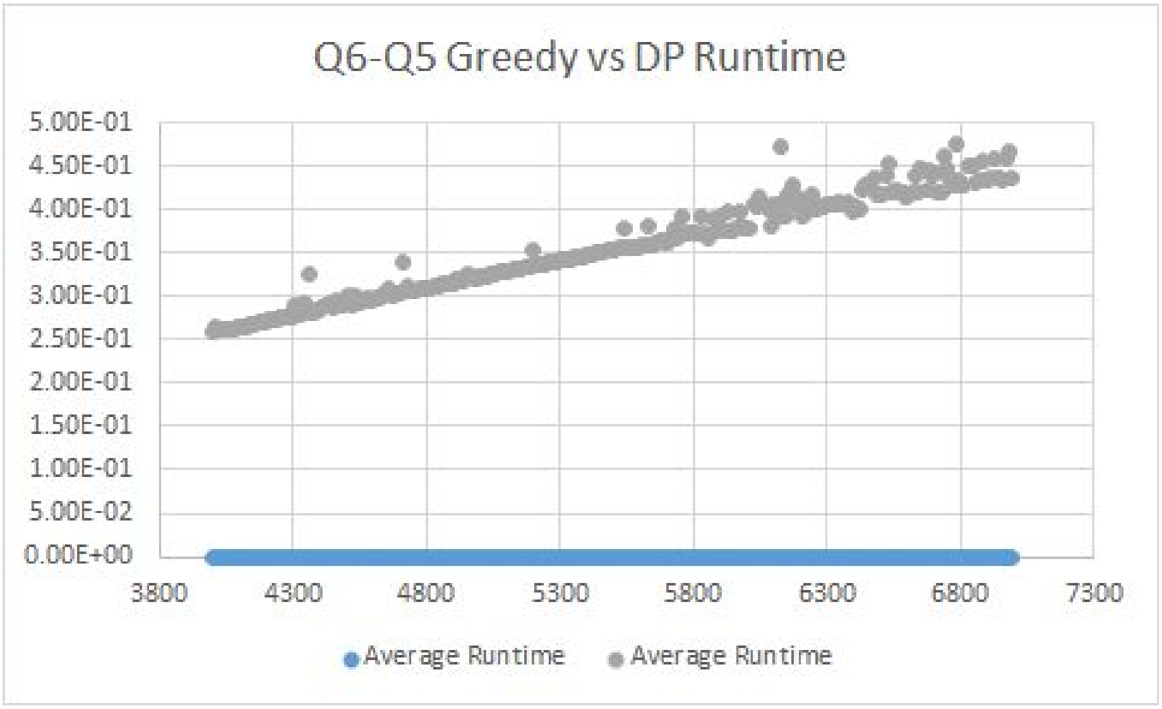


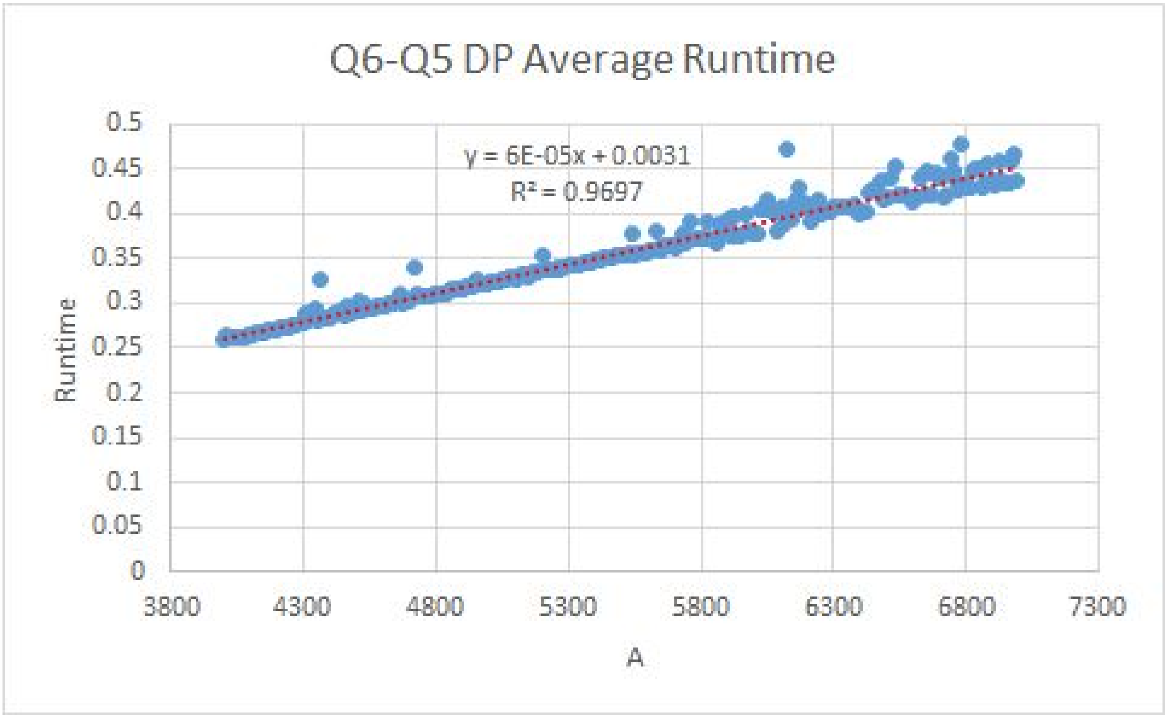


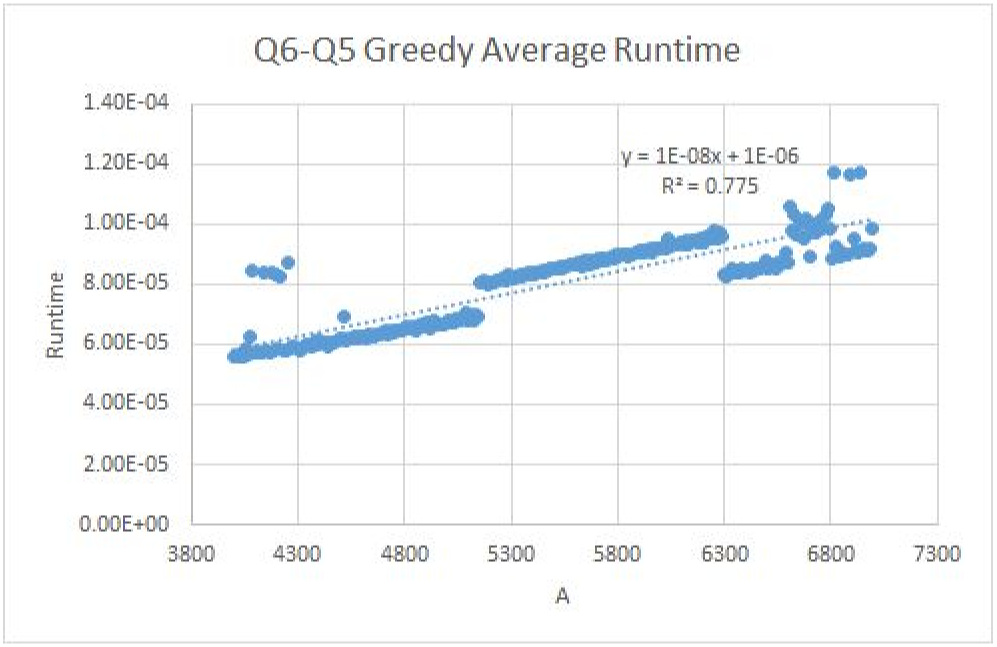


Runtime Analysis of data set from Problem 5

As the plots below demonstrate, the same relationship between the running times of the greedy and dynamic algorithms occurred with the coin denominations used in problem 5. Since the greedy algorithm produced an optimal solution for every change amount we tested in problem 5, it would be preferable to use the greedy algorithm here just to shave off that coefficient factor. If we can ensure correct results with a faster linear algorithm (especially one that is simpler to code), we ought to do so.

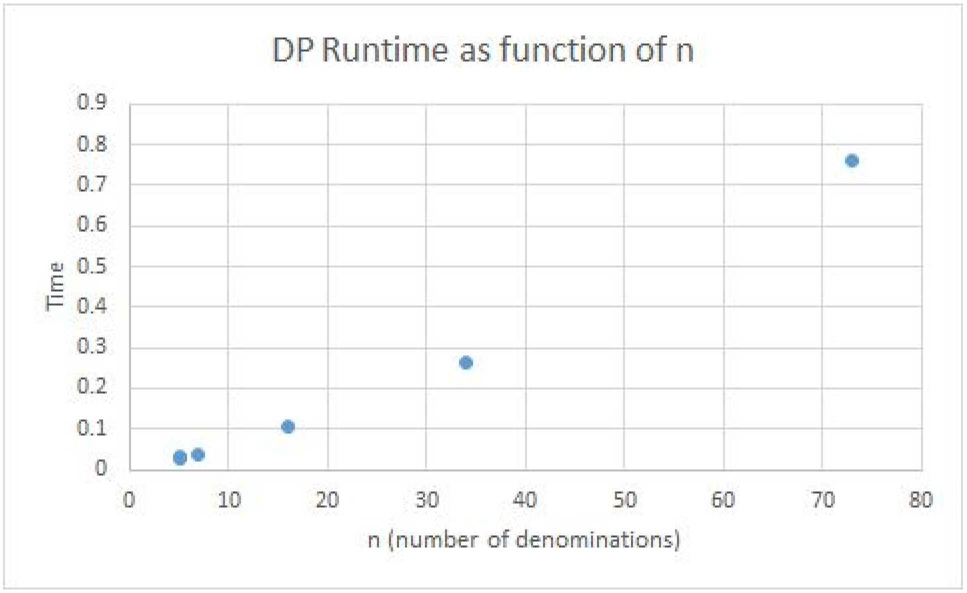




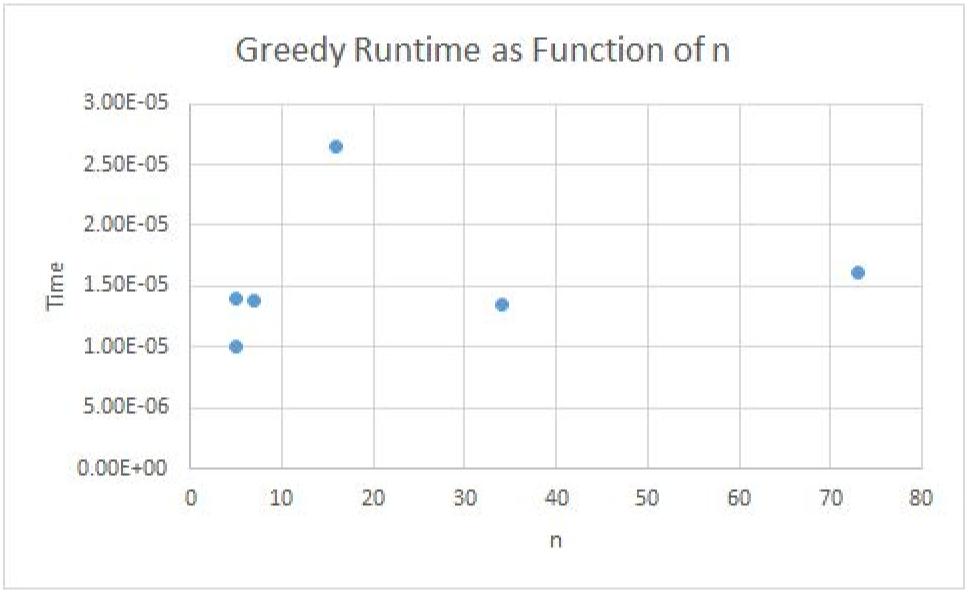


Question 7

We set A to “2000” and used various configurations of the denomination arrays, V, to generate the runtimes depicted. The variable n represents the number of coin denominations available per the question’s nomenclature. We did figure that certain relationships between the different coin denominations contained within a set might affect the results of our testing, but we were not sure how to account for this properly. Nevertheless, the results are plotted below for the dynamic programming and greedy algorithms. In the lectures, we learned that a DP algorithm’s runtime is influenced by the time needed to memoize the results in a table. Therefore, if A is the change amount and n is the number of denominations available, we might use a two-dimensional array of size A*n to memoize our subproblems’ optimal solutions. In the graph below we can see that the runtime changed linearly as n increased. This makes sense because we fixed A to a set value.



In the greedy algorithm, the relationship appears to be more constant. Note that as the data points approach n = 0, a best fit line would not intersect the time axis near 0. This suggests that the greedy algorithm has an initial overhead. What could account for this overhead? In lecture, we learned that a greedy algorithm runs in time proportional to A when n is relatively small. In this case, A was much greater than n, which meant that its influence on the runtime was significant while n's influence was negligible. It therefore makes sense that the graph suggests an almost O(1) runtime.



Question 8

In any country where the coins have values that are consecutive distinct powers of p, such as V=[1, 3, 9, 27], but in general $[d^0, d^1, d^2, \ldots, d^n]$ where d is some integer greater than 1 and n is nonnegative, the dynamic programming approach and greedy approach will always yield exactly the same result.

Proof (by contrapositive):
Suppose there exists an amount, a, and a denomination array $D = [d^0, d^1, d^2, \ldots, d^n]$ with $d > 1$ and non-negative n, such that the dynamic programming approach yields a different minimum amount of coins than the greedy approach.

Suppose that the greedy algorithm generates a minarray of $[g_0, g_1, g_2, \ldots, g_n]$. Note that the minimum amount of coins used here is then $G = g_0 + g_1 + g_2 + \ldots + g_n$. Suppose also that the dp algorithm generates a minarray of $[p_0, p_1, \ldots, p_n]$, where each g_i is not necessarily equal to p_i. Note that the minimum amount of coins used here is $P = p_0 + p_1 + p_2 + \ldots + p_n$. Because we assumed the greedy approach and dp approach yield different results, we know that $P \neq G$.

Critically of note is that for each value g_i for $0 \leq i \leq n-1$, $g_i < d$. Because D consists of a sequence of powers of d, if $g_i = d$ then the greedy algorithm would have chosen $g_i = 0$ and g_{i+1} to be one unit larger; likewise if $g_i > d$, the greedy algorithm would have chosen g_i to be d units less and g_{i+1} to be one unit larger.

Suppose that the dynamic programming and greedy results differ at g_n and p_n. We know that the greedy algorithm takes as many units from the amount as is possible, so $g_n = \text{floor}(a/d^n)$. We know that p_n cannot be greater than g_n, because we would be left with a negative amount of coins to make change for. Likewise, we know that p_n cannot be less than g_n because if $p_n = g_n - 1$, then to compensate $p_{n-1} = g_{n-1} + d$, to make an equal amount of change using the next smaller amount of coin… which would cause the dp algo to equally trade 1 coin for d coins, which is nonoptimal. Hence, we can conclude that $p_n = g_n$.

We consider next that g_{n-1} may be unequal to p_{n-1}. As above, we can assert that the greedy algorithm takes the most coins it can, p_{n-1} cannot be greater than g_{n-1} because that would leave a negative amount of coins, and p_{n-1} cannot be less than g_{n-1} because that would require the dp algo to use at least d-1 more coins than the greedy algorithm. Hence, $g_{n-1} = p_{n-1}$.
In essence, this is a reduction of the original proof, for values amount = a - floor(a/d^n), $D = [d^0, d^1, d^2, \ldots, d^{n-1}]$, $G = g_0 + g_1 + g_2 + \ldots + g_{n-1}$, and $P = p_0 + p_1 + p_2 + \ldots + p_{n-1}$.

The proof continues to reduce in such a way, until we've asserted that $p_i = g_i$ for $0 \leq i \leq n$, which implies that $P = G$, a contradiction to our assumption. Hence, by contrapositive, we have that in a case where the values of a denomination are all consecutive distinct powers of some positive integer p, the dynamic programming approach and the greedy approach yield the same result.