CS 325 Benjamin Olson
Group 14 David Profio
October 28, 2015 Timothy Robinson

1. For the dynamic programming table in changedp we fill it bottoms up by first providing the solution for T[0]. We then fill the remainder of the table by iteratively increasing our amount in the table and solving for it up to the amount A. We find the minimum amount of coins for T[i] by taking each coin available to us that is less than the amount i and subtracting it from the amount. We then take that difference and retrieve the coins it takes to get that amount. Then we add one to the coin column of whichever coin we just subtracted to account for that coin. Once we have found the possible coin solutions in that previous method we take the minimum of those solutions. This is a valid method as we always know that we have a coin with the value of 1. This means we are able to build T[1], and thus we are able to build the rest of our table. If we were not guaranteed a coin with a value of 1 then this method of building the table would be flawed.

2. **Algorithm 1 - changeslow**

```
changeslow(Array V, int A, Array change) // where v is the array of denominations,  A is the
amount, and change is the Array to store the change amounts
{
    changeCount = A
    tempChangeCount = 0

    // base case
    for (i = 0 to size of V)
    {
        if (V[i] == A)
        {
            change[i]++
            return
        }
    }

    for (i = 1 to <= A/2)
    {
        Array tempchange(change.size(), 0)
        Array tempchange2(change.size(), 0)

        changeslow(V, A - i, tempchange)
        changeslow(V, i, tempchange2)

        for ( j = 0 to tempchange.size())
        {
            tempchange[j] += tempchange2[j]
        }

        tempChangeCount = sum of tempchange contents

        if (tempChangeCount <= changeCount)
```

```
                {
                        changeCount = tempChangeCount
                        change = tempchange
                }

        }

}
```

Algorithm 2 - changegreedy

```
changegreedy(Array V, int A)   // Where V is the array of denominations and A is the Amount
{
        Array coins = Array size of V filled with 0;

        index = the maximum coin denomination position

        while (A > 0)
        {
                while (V[index] > A)
                {
                        index--
                }

                A - V[index]

                coins[index]++
        }

        return coins
}
```

Algorithms 3 - changedp

```
changedp(vector<int> V, int A, vector<int> &change)
{
        Array of Arrays solutions
        Array retValue

        // set solutions[0]
        Array zeroSolution(V.size(), 0)
        solutions[0] = zeroSolution

        // bottom up fill the solutions table
        for (i = 1 to <= A)
```

```
    {

        Array tempchange(V.size(), 0)
        int count = A
        int j = 0

        // while the coin at V[j] is less than or equal to value A subtract V[j] from A and
get the solution out of the solution table. We know that is is there since we are building from
V[0] up
        while (j < V.size() and V[j] <= i)
        {
            Array tempChange(V.size(), 0)
            int tempCount = A

            tempChange = solutions[i - V[j]]
            tempChange[j]++; // add one to tempchange to cover the addition of one
V[i] coin
            tempCount = sum of tempChange contents

            if (tempCount <= count)
            {
                retValue = tempChange;
                count = tempCount
            }

            j++
        }

        // solutions[i] = min from above while loop
        solutions.push_back(retValue);
    }

    change = solutions[A]
}
```

3.

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

Step 1: Define Subproblem:

The problem is that we are to find the minimum number of coins needed to make v change.

Subproblem: v - coin of any denomination in set V <= v.
Therefore, all subproblems are enumerated as v - V[i] for V[i] <= v and 1 <= i, where V is the set of coin denominations {1 = v1 < v2 < ... < vn}.

Step 2: Write a Recurrence:

Let T[v] = minimum number of coins needed to make v change using set V of coin denominations.

Then the recurrence is:

T[v] = min(V[i]<=v) {T[v - V[i]] + 1}

Step 3: Base Cases:

Case 1: T[0] = 0 coins
Case 2: T[1] = 1 coin
Case 3: T[V[i]] = 1 coin

Step 4: Prove Recurrence of Step 2

Prove Base Cases:

Case 1: T[0] = 0 coins is given to us by definition of the problem, and it is also evident that it takes no coins to make change of v=0.

Case 2: We know T[1] = 1 coin because v1 = 1, the coin denomination of lowest value in set V.

Therefore, T[1] = min(V[i]<=v) {T[1-1] + 1}
= T[0] + 1 (only v1 can be taken from v=1, leaving only one subproblem T[0])
= 1

Case 3: T[V[i]] = min(V[i]<=V[i]) {T[V[i] - V[i]] + 1]}

We can observe that for each coin value in set V, there is only one subproblem where min(V[i]<=V[i]) condition is satisfied as min(V[i]=V[i]). Furthermore, since V[i] - V[i] = 0 for each coin denomination V[i], this gives us the following:

T[V[i]] = min(V[i]=V[i]) {T[0] + 1]} = T[0] + 1 = 1.

Therefore, we've proven that T[V[i]] = 1.

Other cases where v != 0, 1, V[i]:

Part of this proof will involve constructing the formula by observing relevant mathematical properties.
We know by V[i]<= v that v - V[i] >= 0.

We also know that 1 coin accounts for V[i], so that by taking V[i] amount away from v, we are left with amount v - V[i] plus that one coin. Namely, v = v - V[i] + V[i] is taking a coin away from v and then adding it back on.

This gives us T[v - V[i]] + 1 coins, the minimum number of coins needed that must include a coin of value V[i].

This is based on something given to us that we must assume, that T[n] is the minimum number of coins needed to make change n.

Substituting (v - V[i]) for n, it follows that (v - V[i]) is the minimum number of coins needed to make T[v - V[i]].

Now since the size of set V >= 1, there may be more than one coin value that can be deducted from change v. But it still holds that for every value of V[i], we find T[v - V[i]] + 1, the minimum number of coins. For every V[i] at a current level of recursion, there is a recursive call, leading T[n] closer to a base case, because n = v - V[i] and so the subproblem amount of change at each subsequent sublevel of recursion has been decremented by a coin value V[i].

Furthermore, if base case 3 is reached, one more recursive call will lead to case 1. Otherwise, base case 2 will be reached.

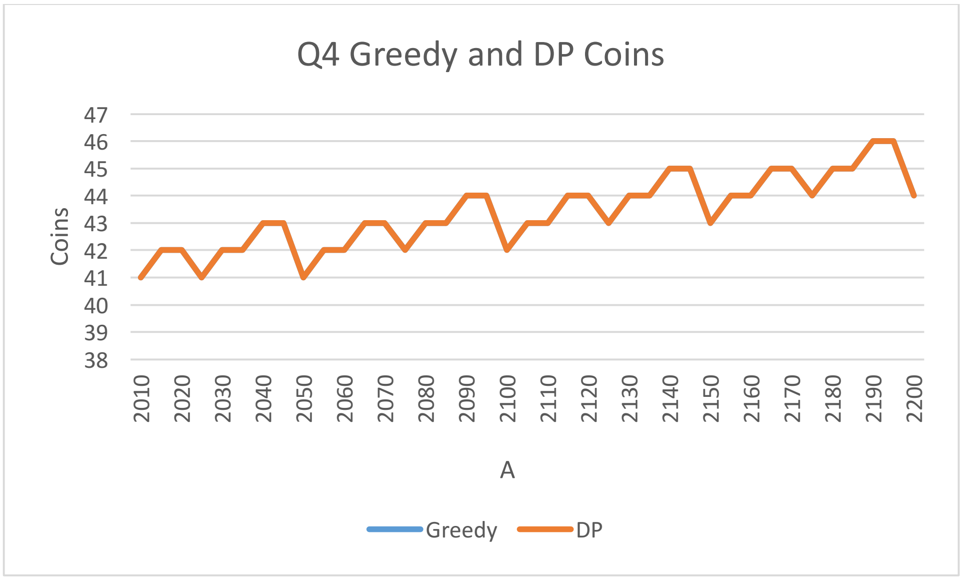
Since we are taking the minimum over all the coin values in V for T[v - V[i]] + 1, we know that for whichever value of i gives this minimum T[v - V[i]] coins, v - V[i] < v where V[i] is positive, so that we must add at least 1 coin to arrive at v. Because 1 is the least number of coins we must add to arrive at amount v, by chosing the coin of value V[i] we now have proven that
T[v - V[i]] + 1 is the least number of coins needed to make change for amount v.

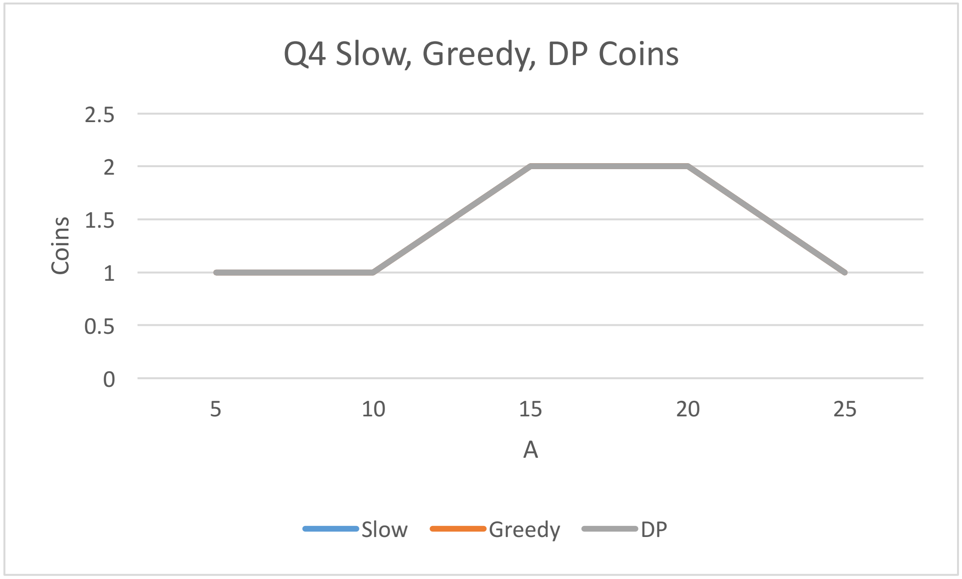
Restated: T[v] = min(V[i]<=V) {T[v - V[i]] + 1}.
This completes our proof.

Note: The iterative, bottom-up DP algorithm provided builds a table of solutions to T[v] for change values 0, 1, ... , v by following this recursive definition starting from the base case T[0] back up the recursion with T[v] as the last problem solved.

Benjamin Olson
David Profio
Timothy Robinson

4.



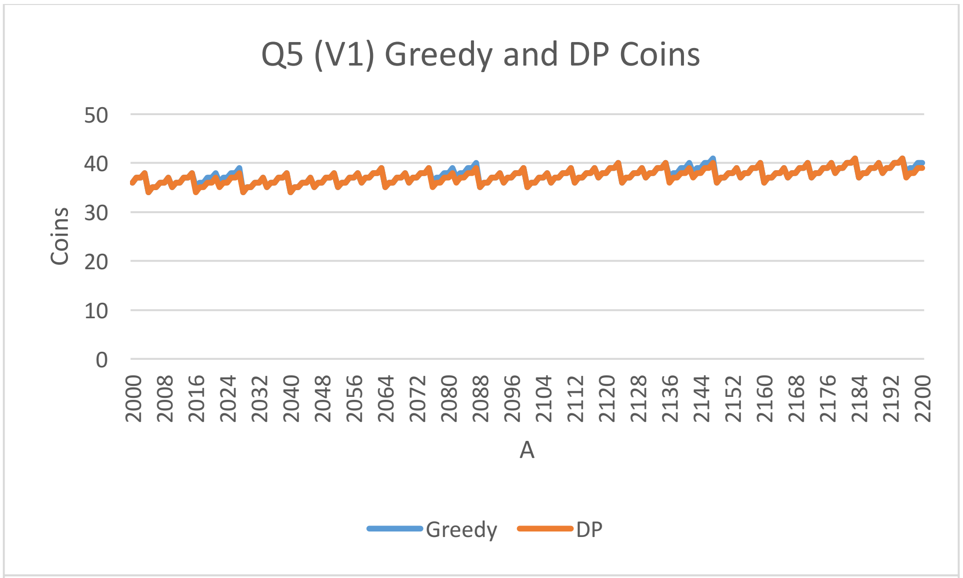


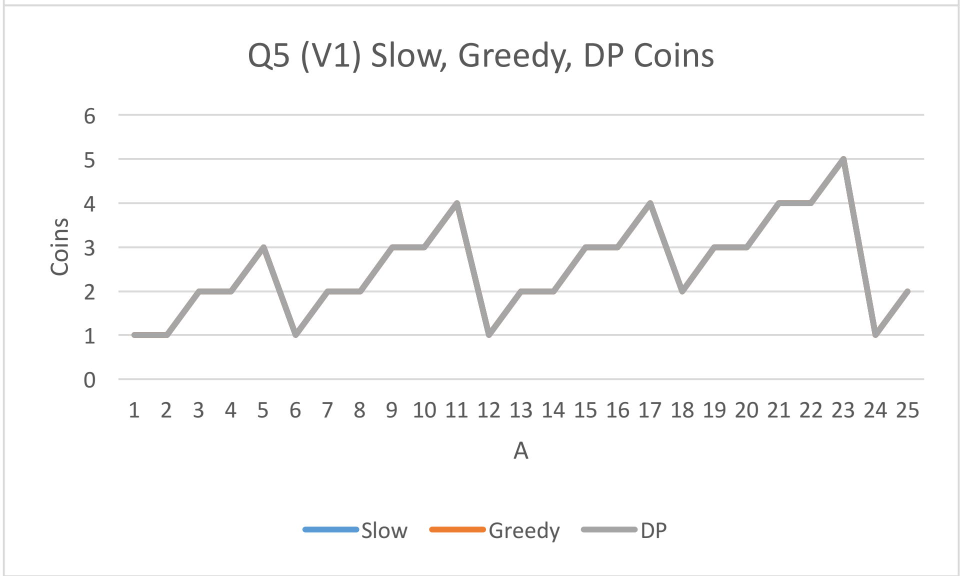
In the first graph above, changegreedy and changedp ran over values of *A* equal to [2010, 2015, 2020, …, 2200] for coin denominations *V* equal to [1, 5, 10, 25, 50]. For this data set, the greedy and dynamic programming approaches produced the same number of coins needed to make appropriate change for each value.

The largest data set completable in a reasonable amount of time for changeslow was 25 for these coin denomination. Each of the three algorithms ran over the smaller collection of values of *A* equal to [5, 10, 15, 20, 25]. Each algorithm produced the same resultant number of coins necessary in the change calculation.

Benjamin Olson
David Profio
Timothy Robinson

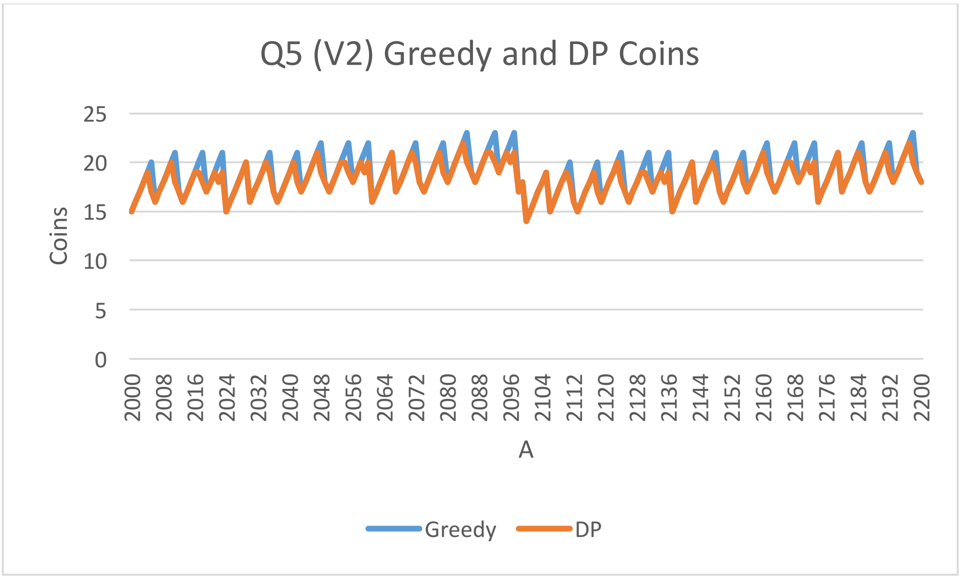
5.

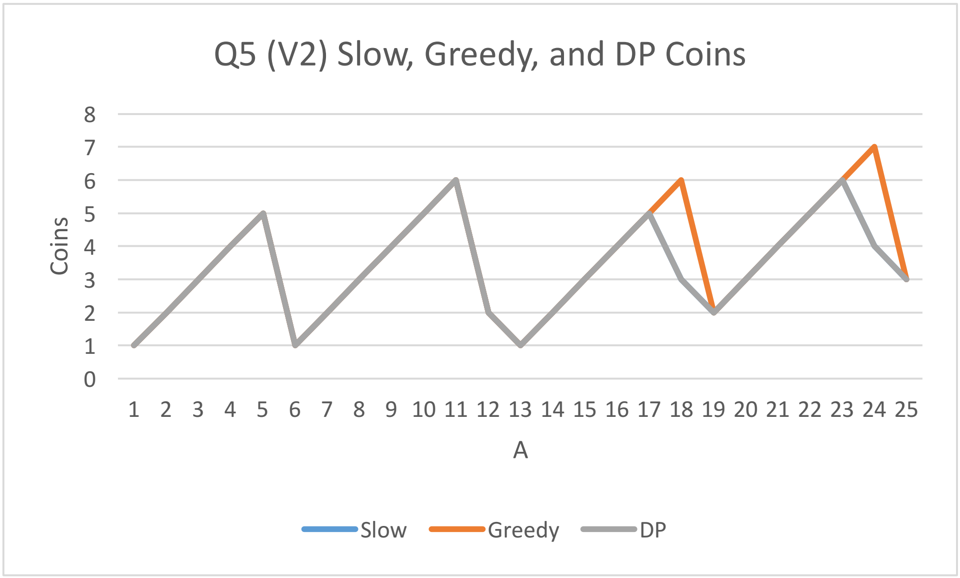




In the first graph above, changegreedy and changedp ran over values of *A* equal to [2000, 2001, 2002, …, 2200] for coin denominations *V1* equal to [1, 2, 6, 12, 24, 48, 60]. For this data set, the greedy and dynamic programming approaches differed depending on the value of A in the number of coins calculated to produce the value of *A*. The greedy algorithm chooses non-optimal solutions from the coin denominations in a periodic fashion, where this period lasts for 60 values of *A*, i.e. the largest coin denomination in this data set. The greedy algorithm's non-optimal solutions last for about 12 consecutive values of *A* in each of these periods. This is an expected experimental result, as the greedy algorithm is periodically choosing "too many" of the largest coin denominations than are found in the optimal solution, as the greedy algorithm chooses as many of the largest coins as possible. Only periodically does this effect hinder the greedy algorithm's ability to choose optimal solutions. The number of consecutive non-optimal solutions are loosely tied to the size difference between the largest and second largest coin denominations. This is due to the dynamic programming approach calculating a better use of coins smaller than the largest denomination.

The largest data set completable in a reasonable amount of time for changeslow was 25 for these coin denominations. Each of the three algorithms ran over the smaller collection of values of *A* equal to [1, 2, 3, …, 25]. Each algorithm produced the same resultant number of coins necessary in the change calculation.

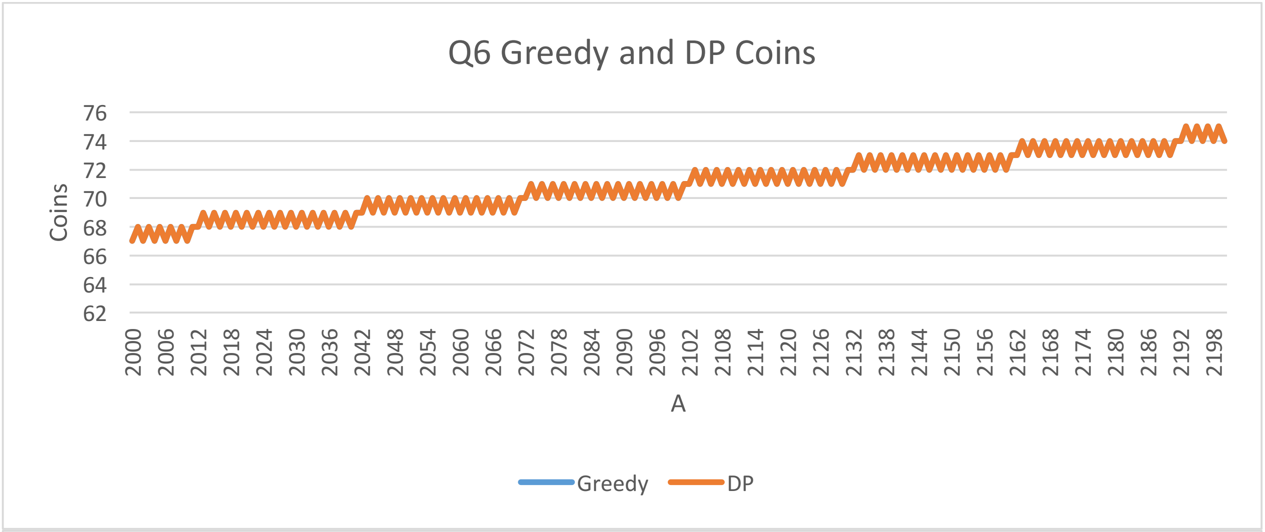


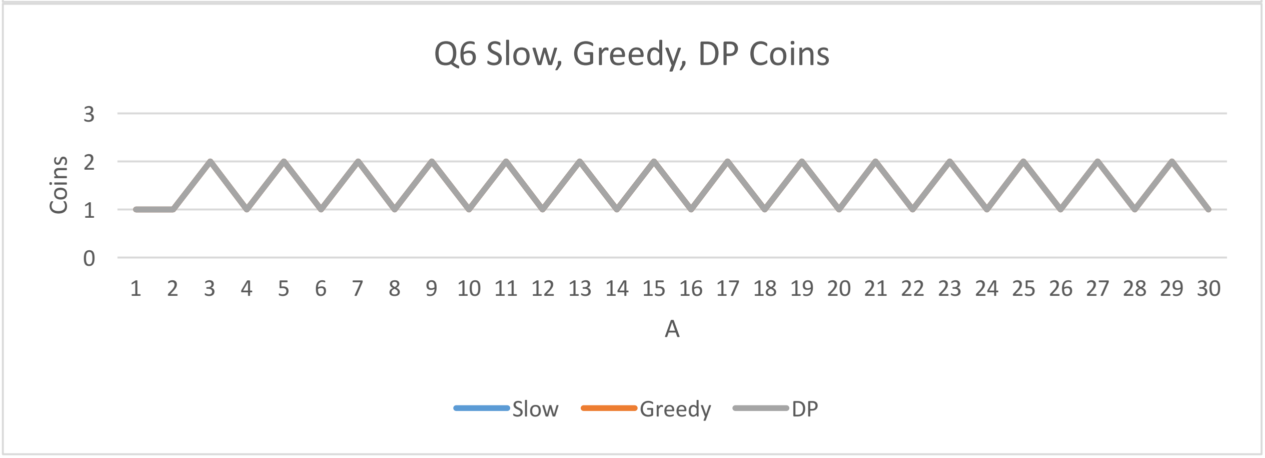


In the first graph of *V2* above, changegreedy and changedp ran over values of *A* equal to [2000, 2001, 2002, …, 2200] for coin denominations *V2* equal to [1, 6, 13, 37, 150]. For this data set, the greedy and dynamic programming approaches produced results similar to those seen with denominations *V1*, though the disparity in results is more pronounced. For fewer values of *A*, the greedy approach produced optimal solutions for fewer values of *A*. There is still a periodicity in the non-optimal solutions produced by the greedy algorithm, for which the period is longer both in bound and in the consecutive values of *A* for which the greedy approach produces non-optimal results. This is expected to be more pronounced for *V2* than *V1* due to the greater disparity in coin denomination sizes in *V2*. This causes the greedy algorithm to be less effective in producing optimal results for the majority of values of *A* as compared to the dynamic programming approach.

The largest data set completable in a reasonable amount of time for changeslow was 25 for these coin denominations. Each of the three algorithms ran over the smaller collection of values of *A* equal to [1, 2, 3, …, 25]. Optimal solutions in this data set are found by both the slow approach and the dynamic programming approach, while the greedy algorithm produced non-optimal solutions with regularity even on this smaller data set.

6.



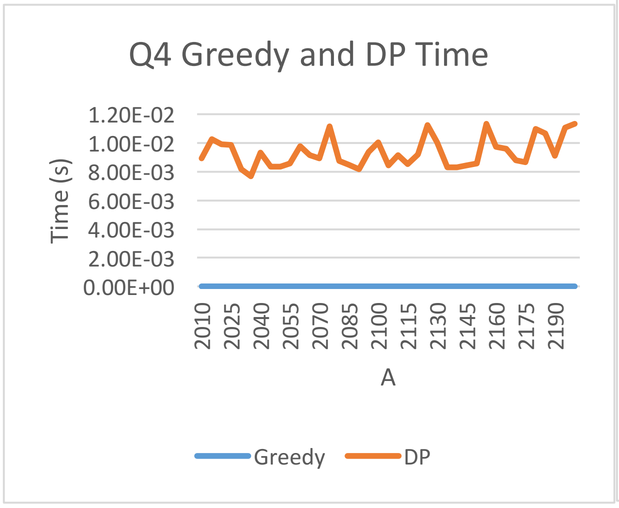


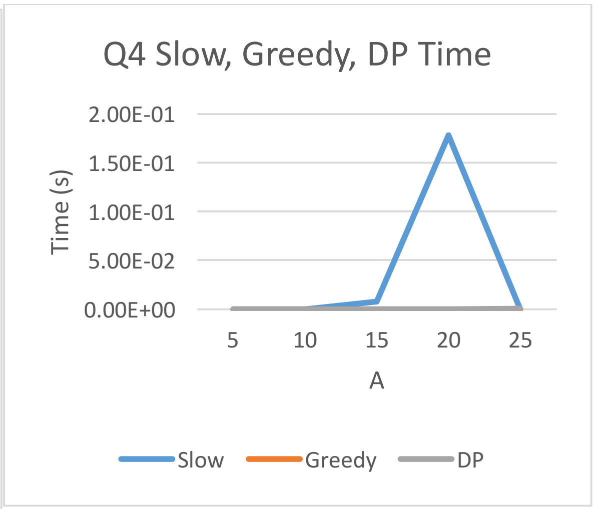
Benjamin Olson
David Profio
Timothy Robinson

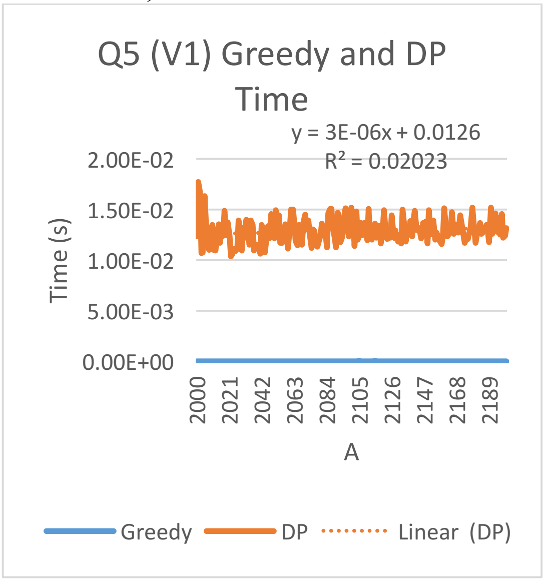
In the first graph above, changegreedy and changedp ran over values of *A* equal to [2000, 2001, 2002, …, 2200] for coin denominations *V* equal to [1, 2, 4, 6, 8, …, 30]. For this data set, the greedy and dynamic programming approaches produced the same number of coins needed to make appropriate change for each value. Due to the dense coin denominations, the behavior seen in the graph is expected, as the approaches are less likely to differ in their solutions. The overall periodicity of the number of coins required while generally steadily rising is expected, as it is clear in the data when a value of *A* requires ever increasing numbers of coins while having a diverse set of denominations available to create a solution.

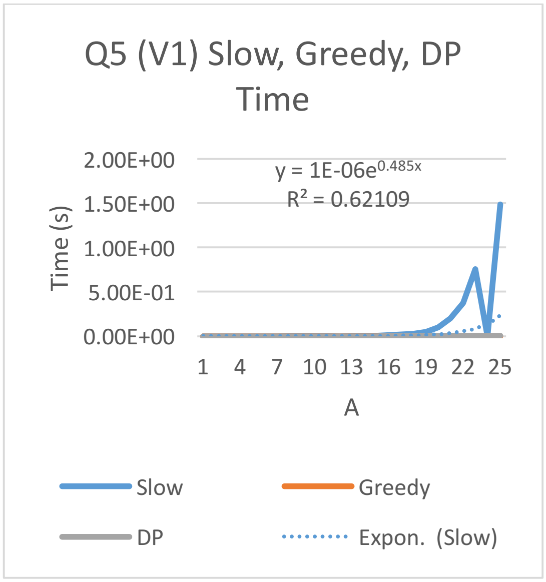
The largest data set completable in a reasonable amount of time for changeslow was 30 for these coin denominations. Each of the three algorithms ran over the smaller collection of values of *A* equal to [1, 2, 3, …, 30]. Optimal solutions in this data set are found by all three algorithms, the straightforward periodicity expected due to the diverse number of coin denominations available. The non-increasing periodicity is due to our *A* values only reaching our maximum coin denomination.

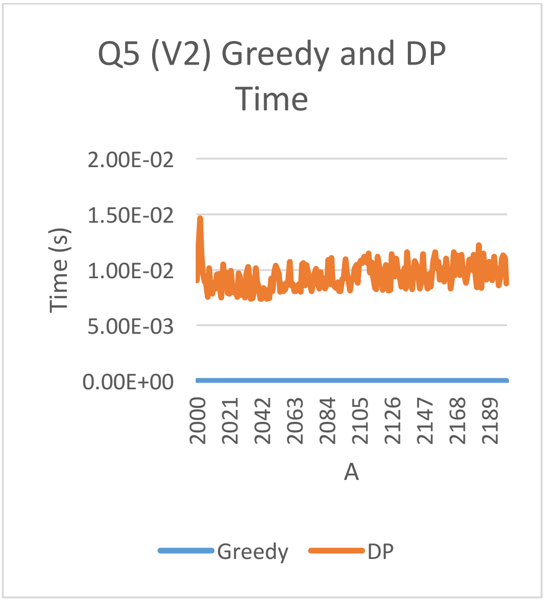
7. The time of each of the above algorithm runs are represented in the below graphs. Additional experimental runs were required to determine better accuracy for the algorithms. Each graph is labelled according to the corresponding to the question from which values of *A* and *V* came.

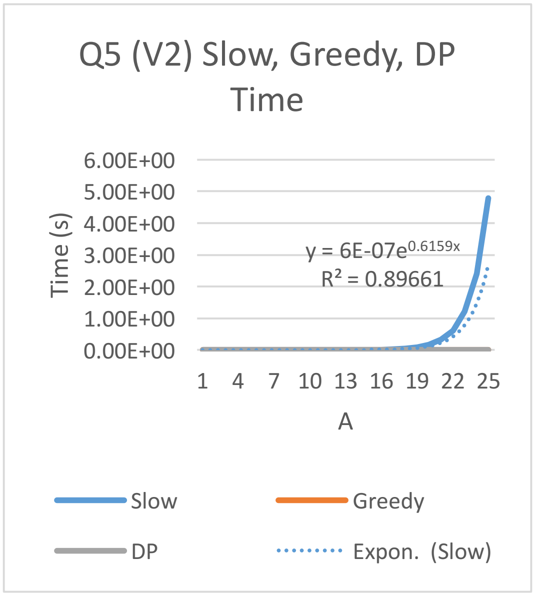


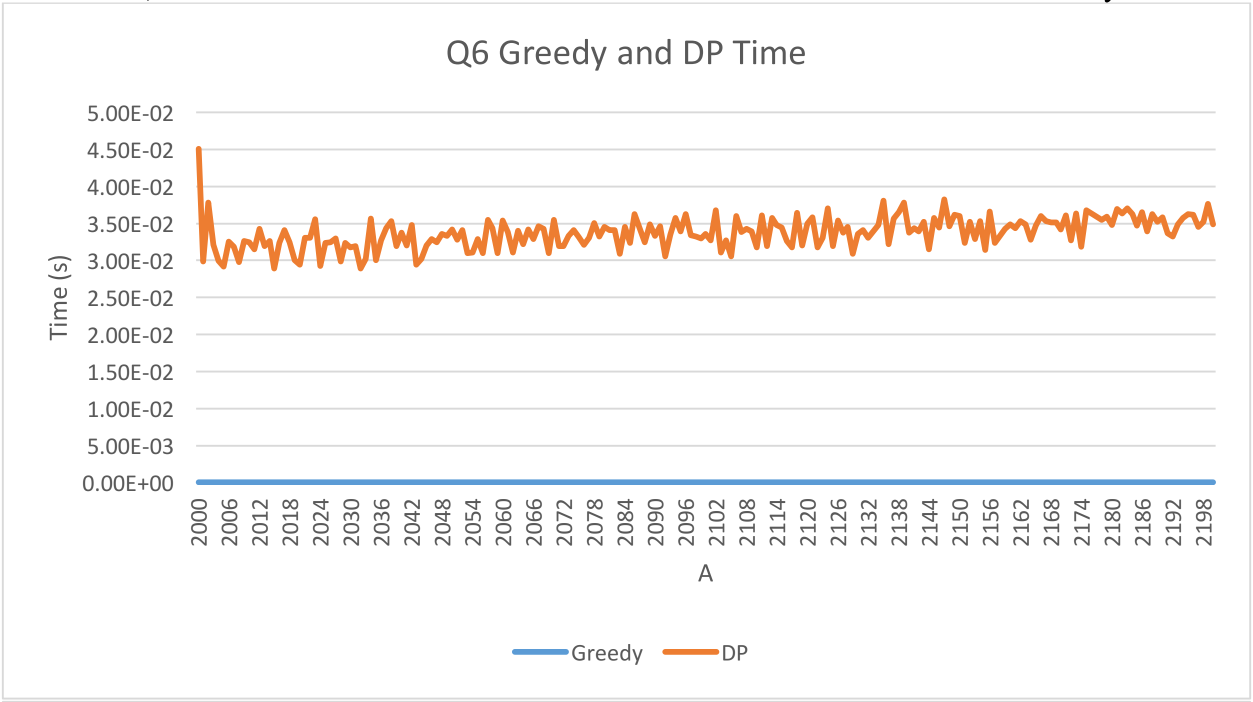


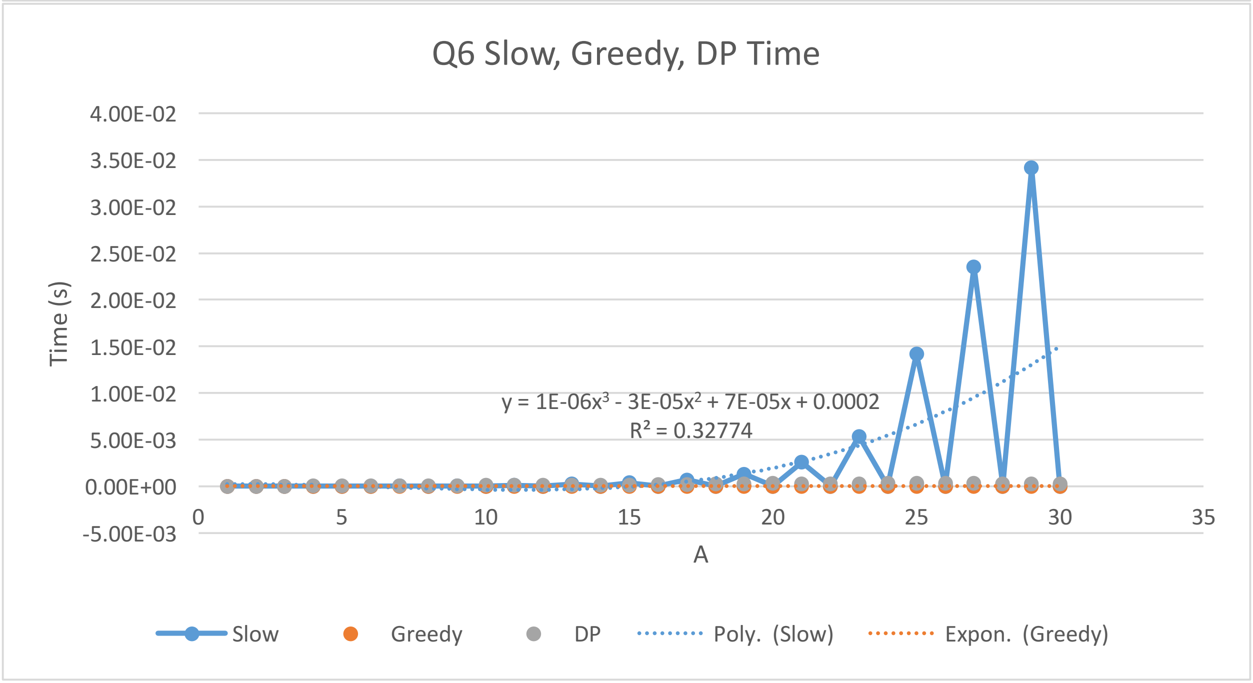

Q5 (V1) Greedy and DP Time
y = 3E-06x + 0.0126
R² = 0.02023
Time (s)
2.00E-02
1.50E-02
1.00E-02
5.00E-03
0.00E+00
2000
2021
2042
2063
2084
2105
2126
2147
2168
2189
A
Greedy
DP
Linear (DP)


Q5 (V1) Slow, Greedy, DP Time
y = 1E-06e^0.485x
R² = 0.62109
Time (s)
2.00E+00
1.50E+00
1.00E+00
5.00E-01
0.00E+00
1
4
7
10
13
16
19
22
25
A
Slow
Greedy
DP
Expon. (Slow)


Q5 (V2) Greedy and DP Time
Time (s)
2.00E-02
1.50E-02
1.00E-02
5.00E-03
0.00E+00
2000
2021
2042
2063
2084
2105
2126
2147
2168
2189
A
Greedy
DP


Q5 (V2) Slow, Greedy, DP Time
y = 6E-07e^0.6159x
R² = 0.89661
Time (s)
6.00E+00
5.00E+00
4.00E+00
3.00E+00
2.00E+00
1.00E+00
0.00E+00
1
4
7
10
13
16
19
22
25
A
Slow
Greedy
DP
Expon. (Slow)



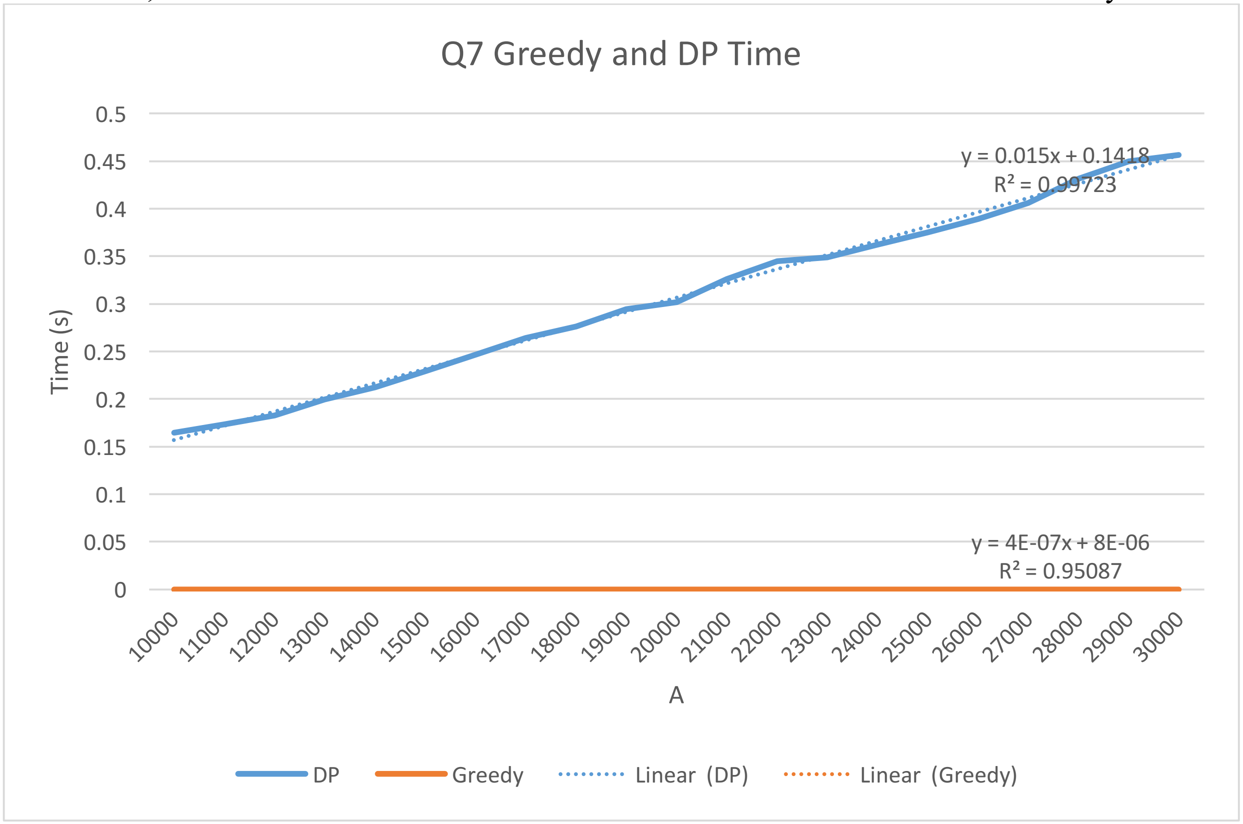


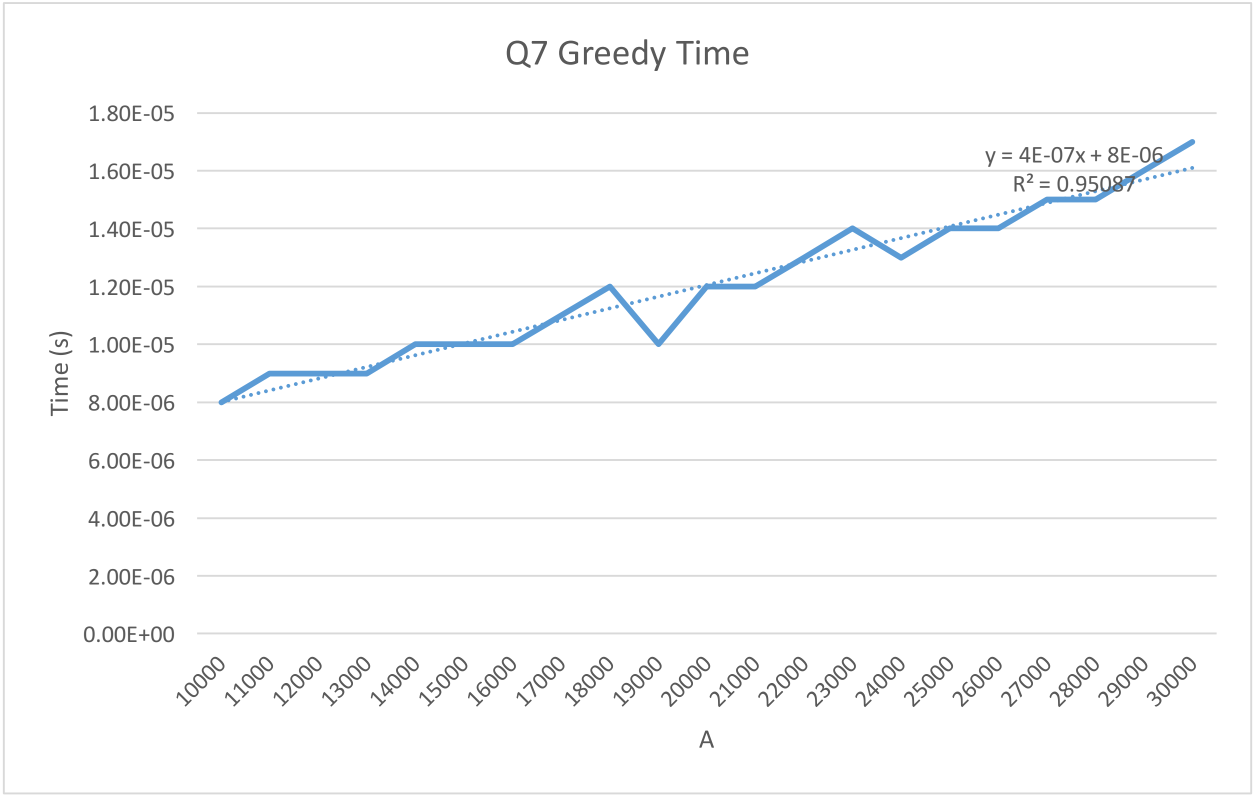
It is clear from the time graphs from the algorithm runs from questions 4 – 6 that a data set better suited for measuring time is required for the greedy and dynamic programming algorithms. The only algorithm for which we can gain some insight in the relationship between *A* and the runtime from the previous questions' data sets is changeslow. Though the data set for question 6 directly above produced a relatively poorly fit line of a polynomial for changeslow's runtime in *A*, the best approximation of the relationship is found in the data set from question 5, using the coin denominations *V2*.

The challenge here in determining the runtime of changeslow lies in both the inevitable exponential increase in runtime by A, causing a limited set of results to be available, and the possibility of "getting lucky" with an A value and the coin denominations available. The latter of those challenges can be particularly seen in question 5, using coin denominations $V1$. Though the best fit trendline is indeed exponential, there is an outlier found at A value 24, for which only one coin is needed. This solution is found immediately by each approach, causing data analysis difficulties for the slow algorithm as a whole.

That being said, the general trend of changeslow's runtime is best represented in question 5, $V2$, for which the best approximation of its runtime is of the order $runtime = e^{0.616\,A}$, and is thus exponential with respect to A.

Benjamin Olson
David Profio
Timothy Robinson



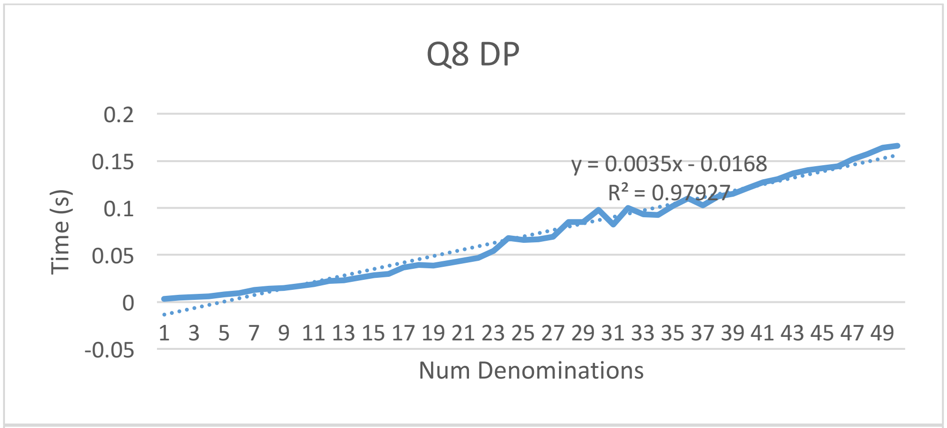


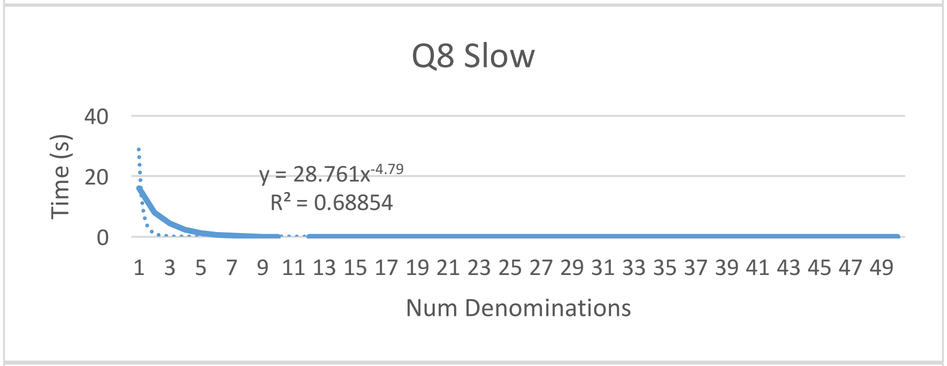
Benjamin Olson
David Profio
Timothy Robinson

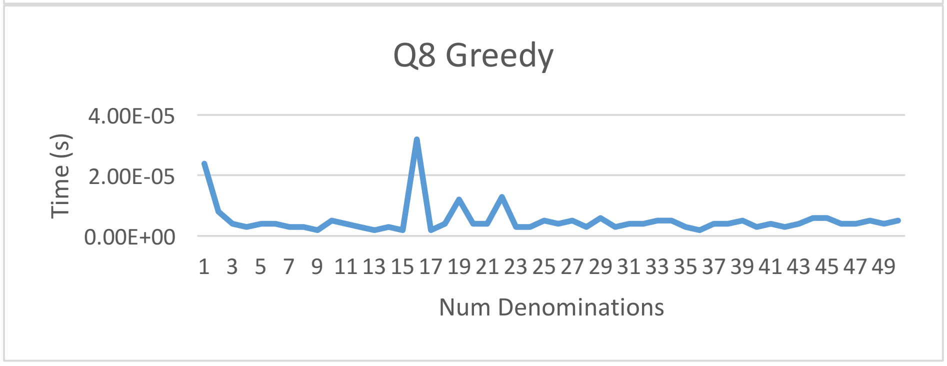
For the above graphs, changegreedy and changedp ran over greater values of A so we could see a greater picture of the experimental time complexity. Here, we used values of A equal to [10000, 11000, 12000, …, 30000], using coin denominations from question 6, with V equal to [1, 2, 4, 6, 8, …, 30]. The first graph of the above two contains both the greedy and dynamic programming algorithm runs, while the second is a zoomed graph highlighting the data set for the greedy algorithm run.

It is quite apparent in the first graph that the runtime of the dynamic programming approach is linear in A, represented by the well fitting line of the order $runtime = 0.015x$. The greedy algorithm, though much, much faster than the dynamic programming algorithm, is also linear in A, though with a modestly increasing slope. The greedy runtime is of the order $runtime = 4 * 10^{-7}x$.

8.







The data set used to create the above graphs involved using a constant value of *A* (2003 for changegreedy and changedp; 25 for changeslow) and a varying set of coin denominations. The varying coin denominations followed the pattern described below:

$$V_0 = [1]$$
$$V_1 = [1, 4]$$
$$V_2 = [1, 4, 7]$$
$$V_3 = [1, 4, 7, 10]$$
$$\vdots$$
$$V_{49} = [1, 4, 7, \ldots, 148]$$

The values of *A* were chosen to be 25 for changeslow due to the maximum reasonable size for which changeslow could be run for any coin denomination (rendering some of the denominations useless in the dataset for changeslow), while 2003 was chosen due to its status as a prime number, requiring a diverse range of denominations to create for each V_n.

Analyzing the above three graphs, it is clear that the greedy algorithm has very little correlation in its runtime in *n*, where *n* in V_n is the size of *V*. For the greedy approach, the size of *n* does not influence its runtime.

The dynamic programming algorithm, on the other hand, is influenced by *n*. For the dynamic programming approach, the runtime is linear in *n*, i.e. the dynamic programming approach takes twice as long to calculate a solution for twice the number of available coin denominations.

An interesting graph here is the changeslow graph. The changeslow graph actually took quite a long time to calculate a solution recursively for the small number of denominations, while operating more quickly when larger denominations were available to the algorithm. Though the runtime appears to exponentially *decrease* in *n* after multiple algorithm runs, we hesitate to declare this as an absolute relationship, as it is easy to imagine a situation in which even moderately large values of *n* would still produce exorbitant runtimes for changeslow.

9. We will demonstrate that by-in-large, the greedy algorithm is much better than the dp algorithm for coin sets of type $V = [p^0, p^1, p^2, \ldots, p^n]$. Also, we'll state that the greedy algorithm gives accurate results only for coin sets of type $V = [p^0, p^1, p^2, \ldots, p^n]$ but is not always accurate for other coin sets V, such as with A = 98 and $V = [1, 49, 50]$.

For all possible values of *n*, we know from question 8 above that as *n* gets larger, the dynamic programming approach will slow linearly in *n*, while the greedy approach will not experience a slow down and will continue to perform much more quickly than the dynamic programming approach.

We know that changegreedy will always take the highest possible coin, start there and then work its way down coin denominations in *V* and thus will have more issues for larger values of both *p* and n. We know from question 5 above that changegreedy produces fewer optimal solutions when there is a greater disparity in the size of the coins in the denomination. The effect of the worse performance of changegreedy is from a smaller coin fitting more squarely in A than a larger coin. Since some smaller coins fit more squarely in A, we may be able to use fewer coins while using smaller coins than taking the largest coin and making change of what's left over. Let's look at an example we can construct from question 5, *V2* coin denominations.

So *V2*: [1, 6, 13, 37, 150]
Take A to be 185, a multiple of 37. This can be made with [0, 0, 0, 5, 0], 5 coins.
Using the greedy approach, this is made with [3, 1 , 2, 0, 1], 7 coins.

Now say that we construct $V = [p^0, p^1, p^2, \ldots, p^n]$, where every coin denomination is divisible by the coin smaller than it. An example like the one above cannot be replicated, as each coin denomination will fit just as squarely using a larger coin as any coin below it. So if a country uses coins that are all powers of p, then the greedy algorithm actually gets optimal solutions every time, while performing much faster than the dynamic programming algorithm.

To take further example comparisons between greedy and dp algorithms, we will start with relatively small values of p:

With a relatively smaller difference between V[i] and V[i+1], in other words, small p value, taking an example p = 3, and V = [1, 3, 9, 27], A = 100, the greedy algorithm chooses in order coins of the following values:

27, 27, 27, 9, 9, 1 = 6 coins.

In this example, there is a call for each coin chosen, for a total of 6 calls. We can observe here that the greedy algorithm always makes (number of coins chosen) calls.

Now, running this same example on the dp algorithm, we find there are $\Theta(n)$ calls = 100 calls, because whether it be the iterative dp or recursive dp, what we end up doing is filling the solution table from values of n = 0 to 100, calculating every possible subproblem. So we see that the greedy algorithm avoids calculating every possible subproblem by working top-down (largest to smallest subproblems) instead of bottom-up (smallest to largest subproblems).

Therefore, compared to the dynamic programming algorithm, we can observe that for relatively small values of p, the performance gain in choosing the greedy algorithm over dp is more significant.

So what if p were relatively large? For example, what if p = 100, A = 199, V = [1, 100, 10000, 1000000]?

Then (1) the greedy algorithm makes the following iterations, adding to "coins" from one iteration to the next:
It 1: T[199], chosen: v2 = 100, coins = 1, remaining change = 99

It 2: T[99], chosen: v1 = 1, coins = 2, remaining change = 98
It 3: T[98], chosen: v1 = 1, coins = 3, remaining change = 97
...(Each successive iteration follows the pattern of choosing v1=1.)
It 99+1=100: T[1], chosen: v1 = 1, coins = 100, remaining change = 0 --> END.

Greedy total iterations = 100, solution: 100 coins

And (2) the iterative dp algorithm makes the following iterations:
After setting T[0] = 0,
It 1: T[1] = 1
It 2: T[2] = 2
...(This pattern continues.)
It 99: T[99] = 99
It 100: T[100] = 1
It 101: T[101] = 2
...
It 199: T[199] = 100

DP total iterations = 199, solution: 100 coins.
(Note: the memoized-recursive dp algorithm must also make 199 calls.)

We can observe from this that (1) the greedy algorithm is still better than (2) the dp algorithm, but by a smaller margin for relatively large values of p where A is close to p, in this case, a time difference of 99 calls * constant time.

However, if we were to add the following logic to both dp and greedy algorithms and compare greedy to recursive dp:

```
if (1 = v1 <= n remaining change < v2)
   T[n] = n
```

in this example of A = 199, V = [1, 100, 10000, 1000000]:
Greedy total iterations = 2
- and -
Recursive DP splits into multiple subproblems ending with T[99], at which point we start memoizing with T[99] and walking back up the recursion tree, also saving us 99 calls, so...

Original recursive DP: 199 calls
New recursive DP: 100 calls (to get from T[199] to T[99]).

Such time reductions can be made with additional logic to compensate for relatively large differences between V[i] and V[i+1], but the greedy algorithm will still be faster.

*****

On an additional note, taking the example input A = 54 and V=[1, 15, 50], we can observe that both the greedy and dp algorithms will give us correct results.

In other words, both algorithms give us minimum coins = 5.

However, by proof-by-construction, we know that the greedy algorithm does not give us correct results for all types of input.

Take, for example, A = 98, V = [1, 49, 50].

The greedy algorithm chooses coins of value 50, 1, 1, 1, and 1, in that order, for a total of 5 coins.

Yet the correct answer is the set of coins valued 49, 49, for a total of 2 coins.
In contrast, the DP algorithm gives us this correct solution.

*****

Conclusion:

The greedy algorithm is faster than the dynamic programming algorithm for sets of V of type
$V = [p^0, p^1, p^2, \ldots, p^n]$.
Also, the greedy algorithm will always be accurate for sets of V of type
$V = [p^0, p^1, p^2, \ldots, p^n]$, but not accurate for all other types of set V.

And although slower overall, the DP algorithm will always be accurate for all sets of type V.