1.

We are using the “bottom up solution”, where we calculate smaller subproblems first and build them up to reach our final solution. This allows us to cache subproblem results in a table from left to right and check before we call a certain element if we can use an existing ansser, so that we only have to perform calculations for a given element once. This way we are finding the main solution in a efficient way due to fact that this problem has overlapping subproblems and we are only calculating each subproblem once and resusing those answers in our build up phase. This is the steps changedp we will take to fill in the table:

First create a vector to store minimum number of coins needed to sum to each change value i

Initialize vector values to infinity to signify no coins

Initialize first vector element to 0, this is the base case. This corresponds to cell 0,0 in our table

Build solution one by one by first iterating through the values of the change up to the desired change value. In our table, x axis values will correspond to each integer from 0 to change value.

> For each value up to change value, iterate through coin set. In our table, y axis values will correspond to each integer in coin set. Note at the top row values will be same as x values.
>
> > As long as change value is greater or equal to current coin set element value j. In our table, if the change value is less than the coin set value then it does not get a mini-solution calculated and is ignored.
> >
> > > Take the minimum of mini-solution for j and mini-solution for j - current coin set element value + 1, where mini-solution is the minimum number of required to make change for respective change value. In our table, considering that the change value is greater than coin set value we will either add that coin set value
> > >
> > > or replace existing coins in solution while adding that coin (the subtracting coin set value number of elements while adding 1 coin with coin set value demoniation) – depending on which returns the minimum value.

At the end of double for loop processing, main solution will be minimum mini-solution and so return the minimum number of coins needed to sum to each change value i. In our table we will take the bottom right most cell to be our final solution for minimum change required to make some change value.

2.

Brute Force or Divide and Conquer Algorithm

To make change for A cents using divide and conquer, first we find the minimum number of coins needed to make change for every value $x < A$. The solution for finding minimum number of coins to make change for A is found by taking the minimum of these mini-solutions. Translated into a formula $C[A] = 1 + C[A-x]$ where 1 is the solution in terms of A and $C[A-x]$ is the mini-solutions.

Psuedocode for Brute Force or Divide and Conquer Algorithm

```
Main function for returning main solution for minimum number of coins to make V change

Create and initialize coin set objects representing current and minimum coin sets for use in helper function

Initialize solution for minimum number of coins to make V change to infinity

Call recursive helper function to run brute force algorithm

    Once the recursive helper function has returned the minimum solution, record the actual coins in the minimum solution coin set

Return the minimum number of coins to make V change

Recursive helper function for finding mini-solutions for minimum numer of coins to make i change where i < V

Base case: If V is 0, then 0 coins are needed

Constant comparison: If V < i, then i cannot be used

Recursive case:  If V >= i, then i can possibly be used

    Recursively find minimum number of coins needed to make V – i change

        Update the minimum solution with the minimum of current mini-solution and current minimum

Return the minimum solution
```

Dynamic Programming Algorithm

We are using the “bottom up solution”, where we calculate smaller subproblems first and build them up to reach our final solution. This allows us to cache subproblem results in a table and check before we call a certain element if we can use an existing ansser, so that we only have to perform calculations for a given element once. This way we are finding the main solution in a efficient way due to fact that this problem has overlapping subproblems and we are only calculating each subproblem once and resusing those answers in our build up phase.

Psuedocode for Dynamic Programming Algorithm

```
First create a vector to store minimum number of coins needed to sum to each change value i

Initialize vector values to infinity to signify no coins

Inititialize first vector element to 0, this is the case case

Build solution one by one by first iterating through the values of the change up to the desired change
value

    For each value up to change value, iterate through coin set

        As long as change value is greater or equal to current coin set element value j

            Take the minimum of mini-solution for j and mini-solution for j - current coin set
            element value + 1, where mini-solution is the minimum number of required to
            make change for respective change value

At the end of double for loop processing, main solution will be minimum mini-solution and so return the
minimum number of coins needed to sum to each change value i
```

Greedy Algorithm

To make change for A cents using greedy algorithm, from greatest to least value for each element of the vector we find how many times its value can be used to make V change, finding the “optimal” solution in terms of that element. It continues this for each element while V is greater than the current change value. So the result may consist of “optimal” solutions in terms of individual elements but a “suboptimal” solution in terms of the entire set.

Psuedocode for Greedy Algorithm

```
Create and initialize coin set object representing minimum coin count for each coin set element
```

```
    while the current change value is greater than 0 we can keep going

    for each element vector[i] from greatest to smallest (since input data ascending)

        Initalize coin count for that element vector[i]

        For each element vector[i], while current change value greater than or equal to vector[i]

            Increment coin count for vector[i]

            Subtract vector[i] from current change value

        Update i element of coin set object with coin count vector[i]

Loop through coin set object to sum up values of coin counts for vector elements

Return sum
```

3.

Proof: We will prove by induction that $T[v] = \min_{\mathrm{v[i]}} \leq vT[v - V[i]] + 1$ is correct. Here the problem is we need to prove that the above function is the minimum number of coins required to make change value v.

The subproblem is, we are given a coin set CS and for any single coin i in CS then $v - CS[i] \leq v$ where $i > 1$.

The recurrence is, $T[v] = min(CS[i] \leq v)(T[v - CS[i]] + 1)$.

Base case: When $v = 0$, then $T[0] = 0$ and when $v = 1$, since it takes no coins to make zero cents. Likewise, $T[1] = 1$ because $T[1] = min(CS[i] \leq v)(T[0] + 1)$ or 1. This makes sense even without plugging it in because 1 is the lowest possible coin value and it makes 1 cent. The last base case is in the case that $CS[i] = CS[i]$ which because the coin value is equal to the coin set element then 1 is returned. This should happen once for every element value in coin set CS. To prove this for some CS element i, we plug in 0 for $[v - CS[i]$, giving $T[i] = min(CS[i] = CS[i](T[0] + 1)$ which is 1.

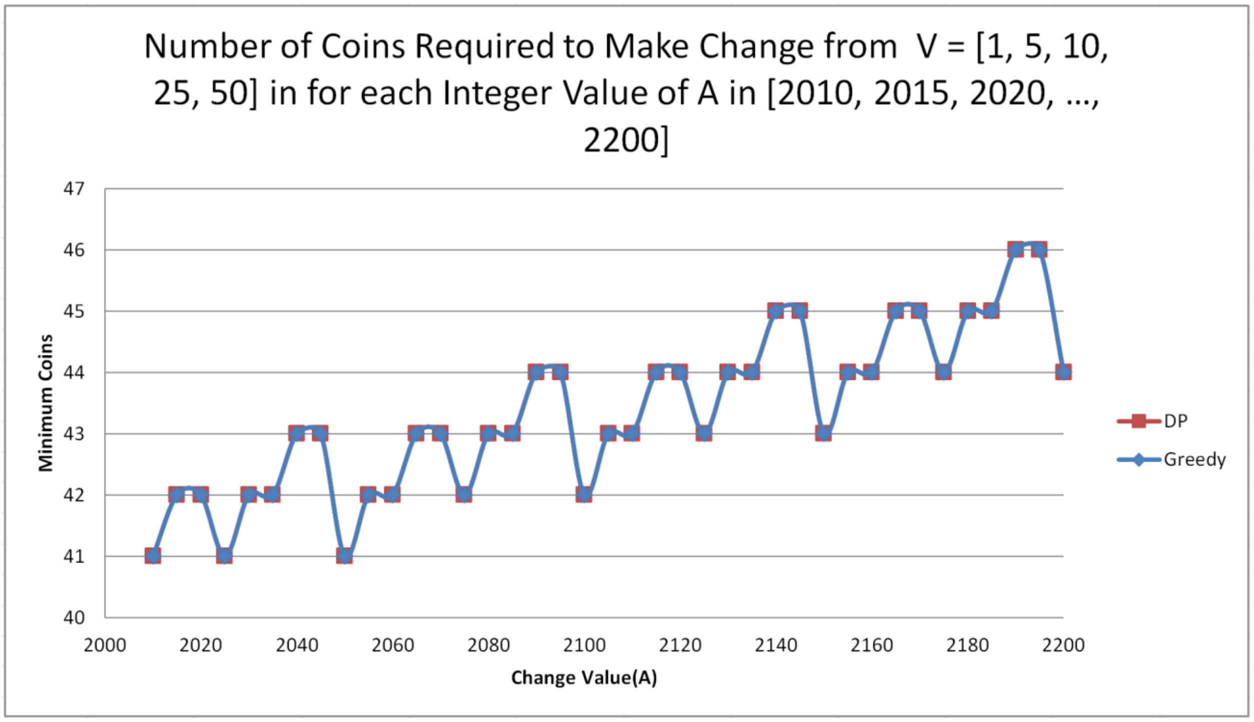
Induction hypothesis: Let $T[k] = \min_{\mathrm{k[i]}} \leq T[k - CS[i] + 1$ is correct for all $k \leq v$.

Induction step: We will show that assuming inductive hypothesis is true, then $T[k + 1] = \min_{\mathrm{k+1[i]}} \leq T[k + 1 - CS[i] + 1$ where $0 \leq k \leq n$ is also true.

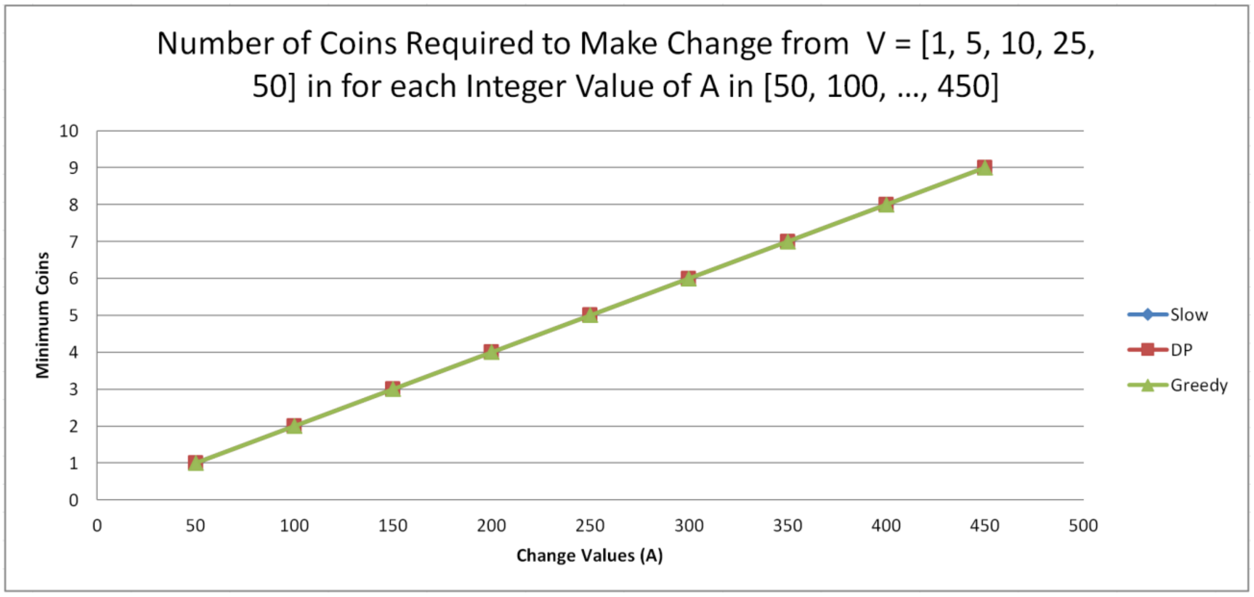
There are two possible cases, when $T[k + 1] = minT(k - CS[i]) + 1 + 1$ and $T[k + 1] < minT(k - CS[i]) + 1$. The case where $T[k + 1]$ is greater is not possible, because CS[i] cannot be greater than change value k so $k - CS[i]$ has to be greater than or equal to 0. The first case corresponds to including other coin to make change of k + 1 (without replacing) and second case is where you take away existing coins in your coin set and replace them with a single larger coin. The second case is where you find a new optimal solution by reducing minimum number of coins needed to make k + 1 change. Since there only these two cases and third case is not possible, then we can conclude the inductive hypothesis must be true.

4

For A of [2010, 2015, 2020, …, 2200] and V equal to [1, 5, 10, 25, 50], changegreedy and changedp returned the same minimum number of coins needed to make change for each value of A.



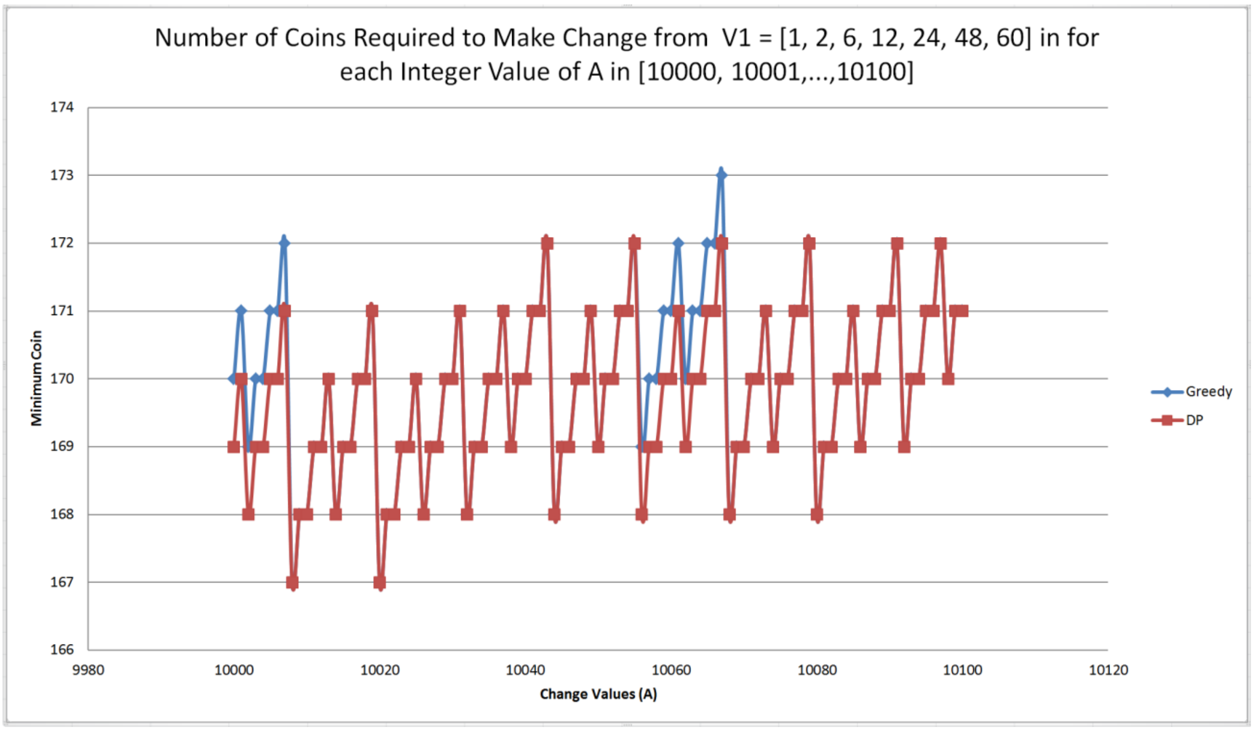
The largest A completable in a reasonable amount of time for changeslow was 450 for V of [1, 5, 10, 25, 50]. For A of [50, 100, …, 450] and V equal to [1, 5, 10, 25, 50], changeslow, changegreedy and changedp returned the same minimum number of coins needed to make change for each value of A.



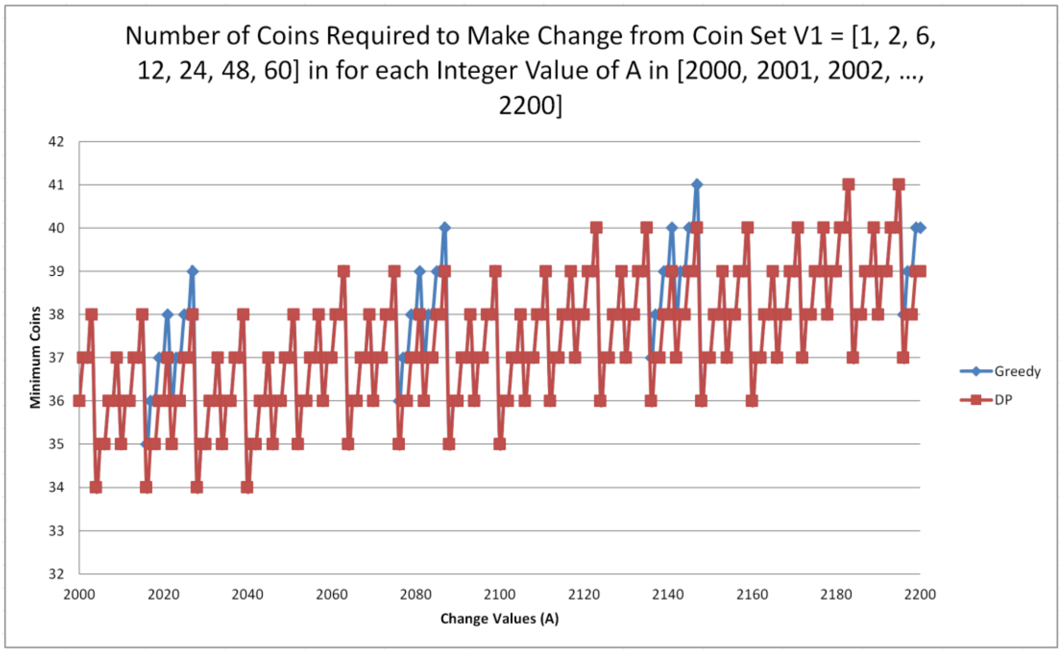
5.

a)

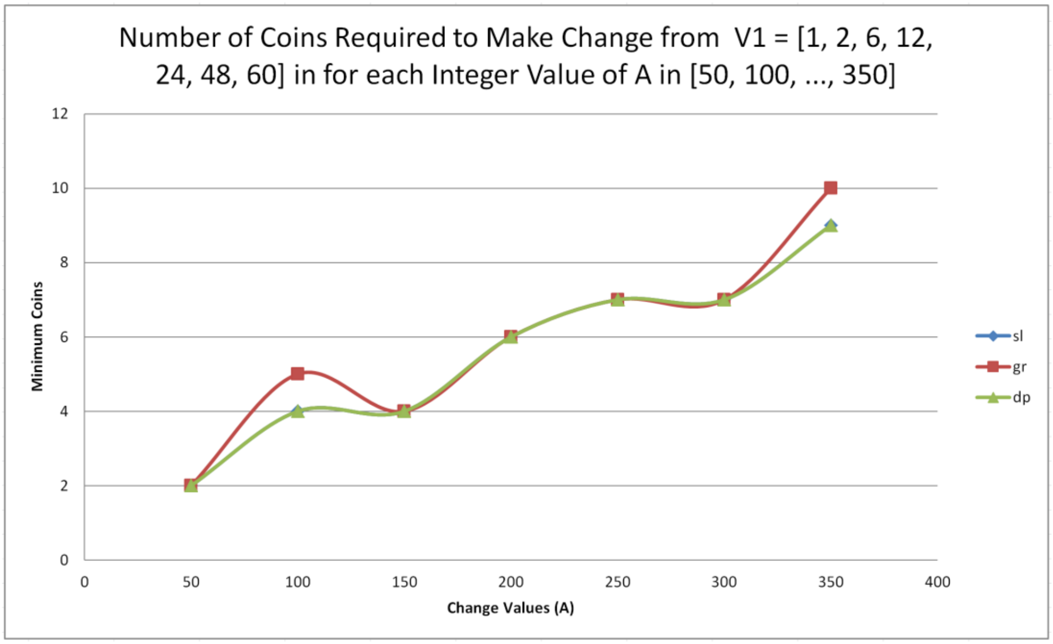
For A of [10000, 10001, …, 10100] and V1 = [1, 2, 6, 12, 24, 48, 60] changegreedy and changedp had different results for minimum number of coins required to make change for each value of A. Change Greedy does not choose the optimal solution for values such as from A in [10000, …., 10007] and A in [10057, …., 10067] in this trial for V1 = [1, 2, 6, 12, 24, 48, 60]. So the greedy algorithm will choose suboptimal solution for about 10 consecutive values of A every interval of around 60. This makes sense because the greedy algorithm can choose too many of the largest coin denominations, which is not present in the optimal solution. The greedy algorithm consistently makes this mistake periodically.



For A of [2000,2001, …, 2200] and V1 = [1, 2, 6, 12, 24, 48, 60] changegreedy does not choose the optimal solution for values such as from A in [2016, …., 2027] and afterwards from A in [2027, …, 2077] it picks the optimal solution in this trial for V1 = [1, 2, 6, 12, 24, 48, 60]. Notice the data sets for A in 2000's and 10000's each increase by 1. <u>So the greedy algorithm will choose suboptimal solution for about 10 consecutive values of A every interval of around 60 for V1 = [1, 2, 6, 12, 24, 48, 60] regardless of data set A when each term in A is increasing by 1. Looking at the data in V1, it is increasing by 1, then 4, then 6 and since the set is not condensed the algorithms do not produce same results.</u>

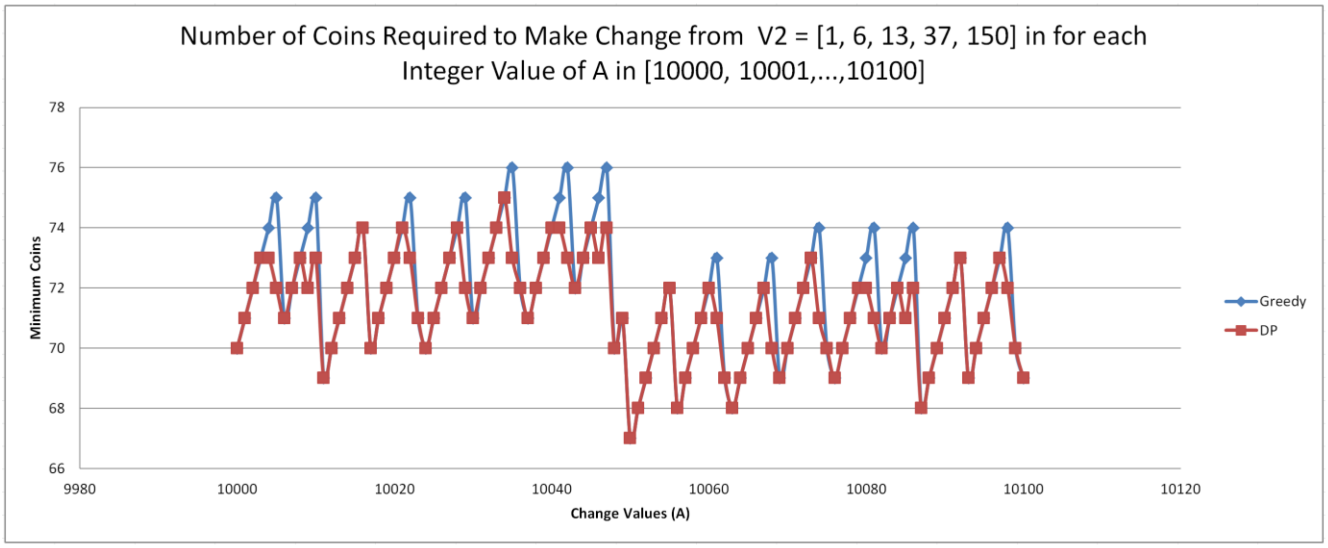


For A of [50,100, …, 350] and V1 = [1, 2, 6, 12, 24, 48, 60] we see that changeslow and changedp are the return the same values but changegreedy does not always choose the optimal solution. Here we see for values such as from A in [50, …, 150] and [300, …, 400] that greedy algorithm will choose suboptimal solution for about 100 consecutive values of A every interval of around 150. So again the greedy algorithm is periodically picking the suboptimal solutions, except with different intervals and different ranges for various sets of A.

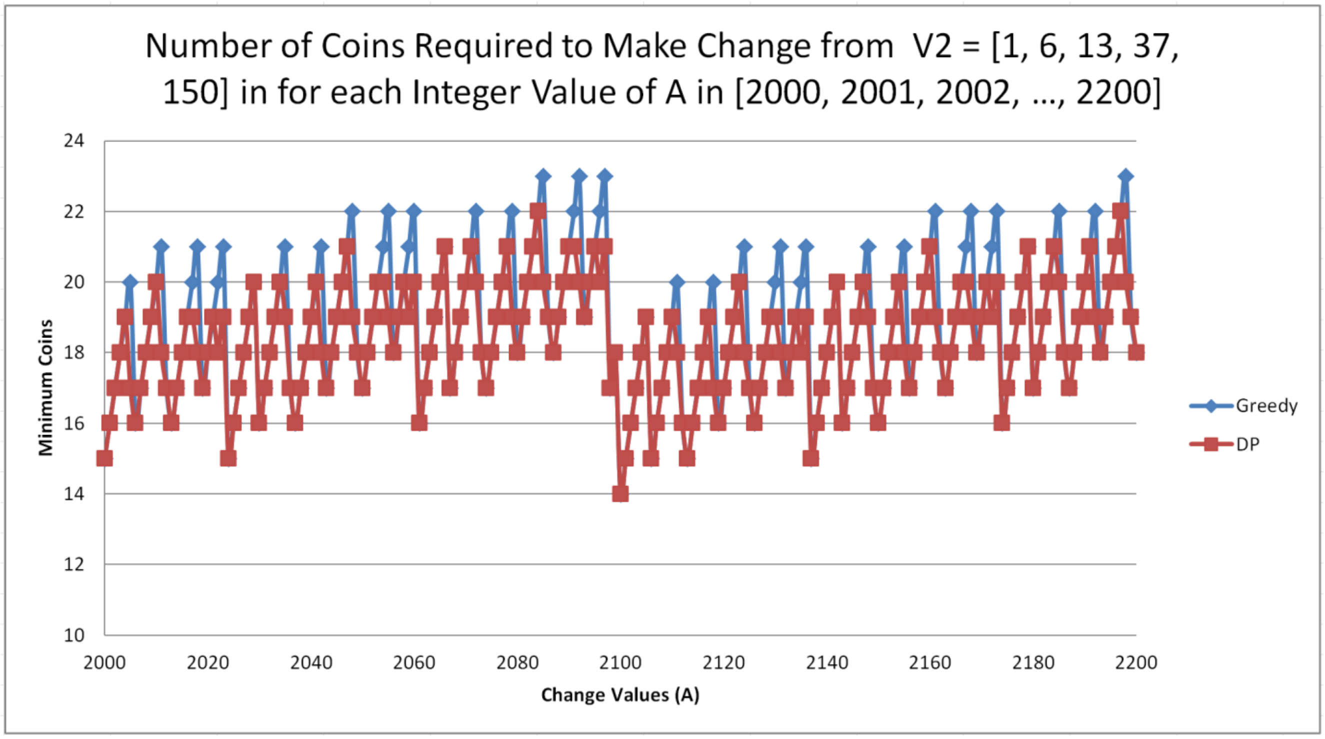


b)

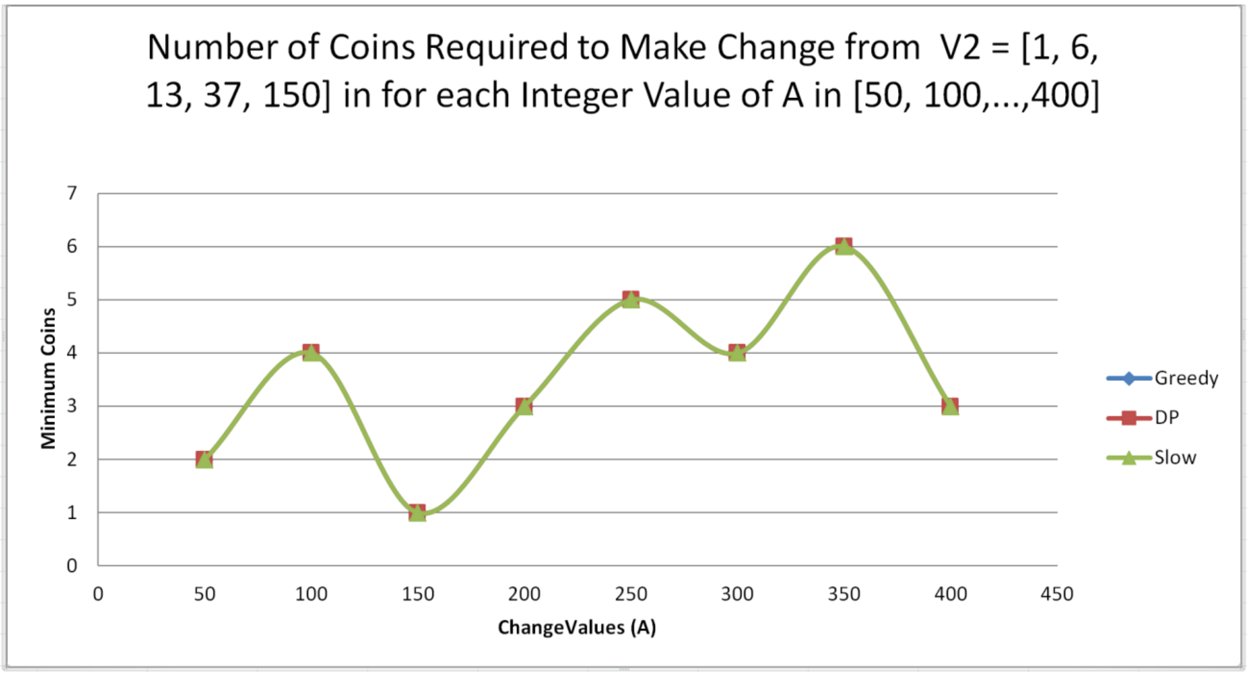
For A of [10000, 10001, …, 10100] and V2 = [1, 6, 13, 37, 150] changegreedy and changedp had different results for minimum number of coins required to make change for each value of A. Change Greedy does not choose the optimal solution for values such as from A in [10003, …., 10005], picks the optimal solution from [10005, …, 10008] then from [10008, …., 10010] goes back to picking the incorrect solution. It continues this up and down pattern five times before embarking on a ~27 A value streak where it picks the optimal solution such as for A in [10021, …., 10023], in [10028, …., 10030] in this trial. So there's a period of around 10 A's where there are five peaks of greedy algorithm picking suboptimal then optimal solution, then around 27 A's where greedy picks the optimal solution each time – this is a pattern. Again the greedy algorithm is periodically picking the suboptimal solutions, except with different intervals and different ranges for various sets of A.



For A of [2000,2001, …, 2200] and V2 = [1, 6, 13, 37, 150] changegreedy does not choose the optimal solution for values such as from A in [2004, …., 2006], picks optimal solutiuon for A in [2006, …, 2010] , and proceeds to pick suboptimal solution over next 2 values of A and then optimal solution for next 4 values of A. It proceeds with five of these peaks until a long stretch of around 10 A's where it picks optimal solution such as from A in [2023, …, 2034]. Notice this is the exact same trend as when A's were in the 10,000s – probably because both data sets were increasing by 1 each time and V2 data is increasing by 5, then 7, then 24 and since the set is not condensed the algorithms do not produce same results.

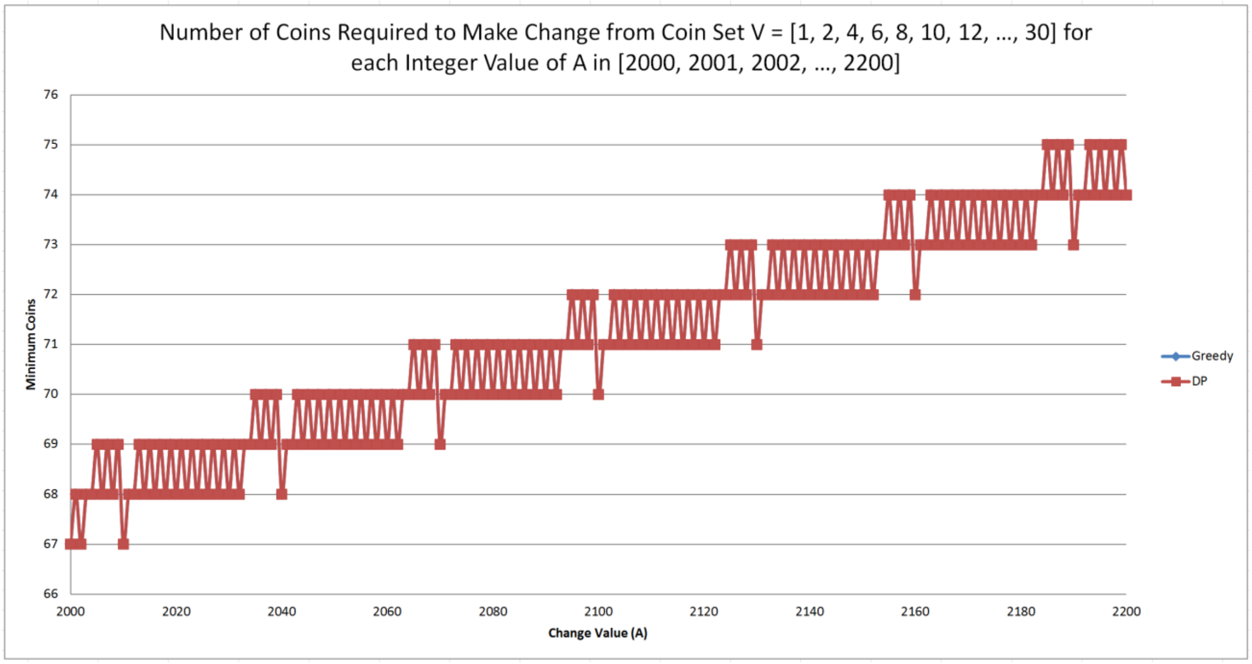


In this chart, the changeslow, changegreedy and changedp returned the same minimum number of coins needed to make change for each value of A for small values of A in [50, 100, …400] showing that for some coin sets and change values the greedy algorithm can provide the optimal solution. It just does not do it consistently but it works for this very small data set.

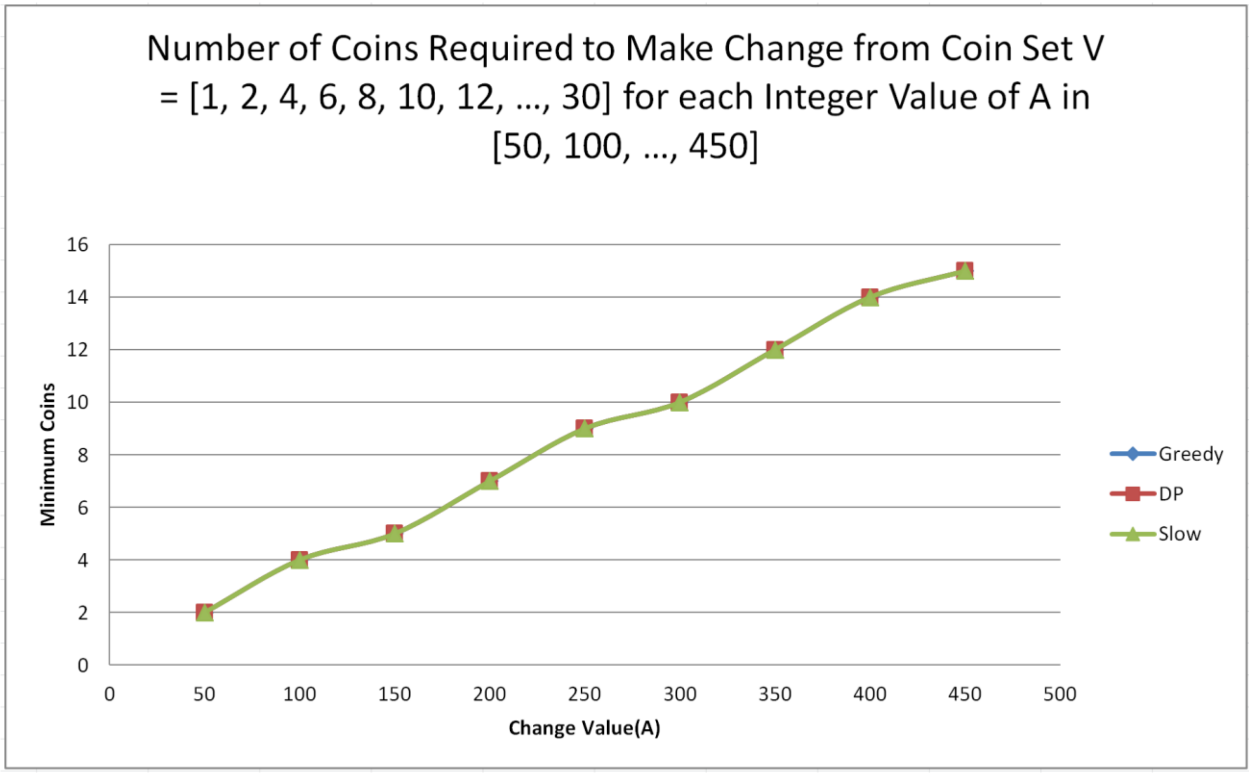


6.

For A of [2000, 2001, 2002, …, 2200] and V = [1, 2, 4, 6, 8, 10, 12, …, 30] changegreedy and changedp had same results for minimum number of coins required to make change for each value of A. This may be due to the fact that numbers in V are much more condensed than in other data sets (only increase from 1-2 consecutively), rather than in other sets where each number varies around the range of 5-20. This denseness of data may cause results for algorithms to be the same.



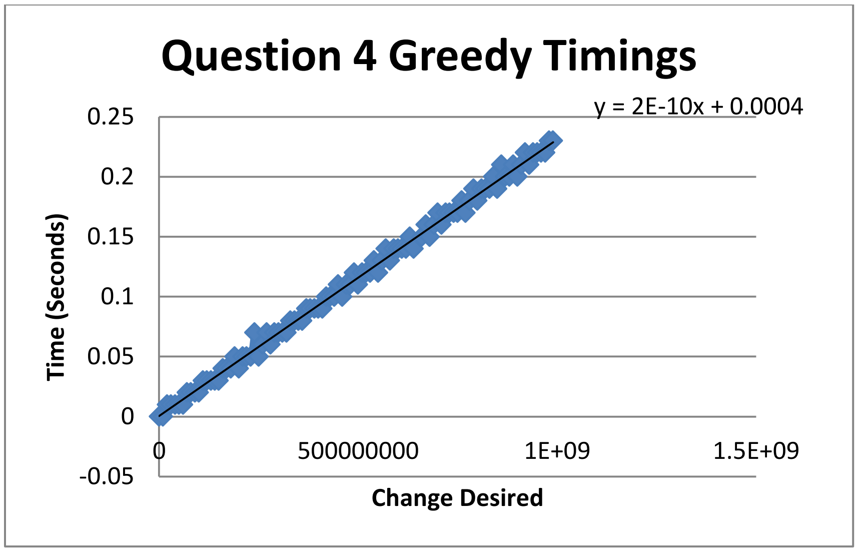
In this chart, the changeslow, changegreedy and changedp returned the same minimum number of coins needed to make change for each value of A for small values of A in [50, 100, …450] showing that for some coin sets (like ones where values are close together) and change values the greedy algorithm can provide the optimal solution. It just does not do it consistently but it works for this very small data set.

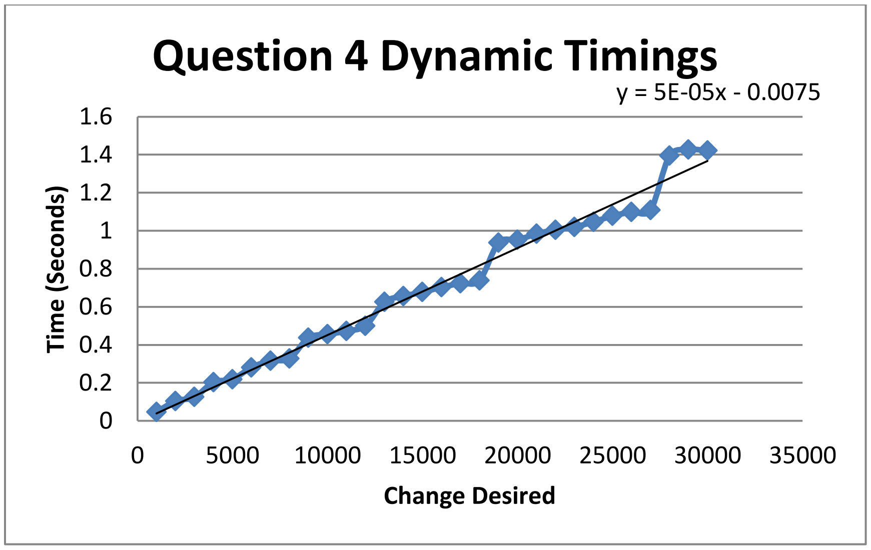


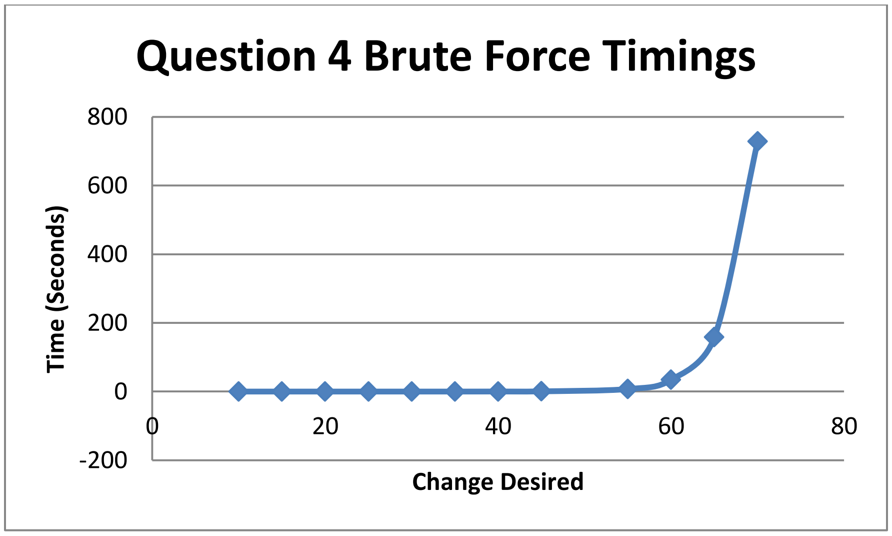
7.

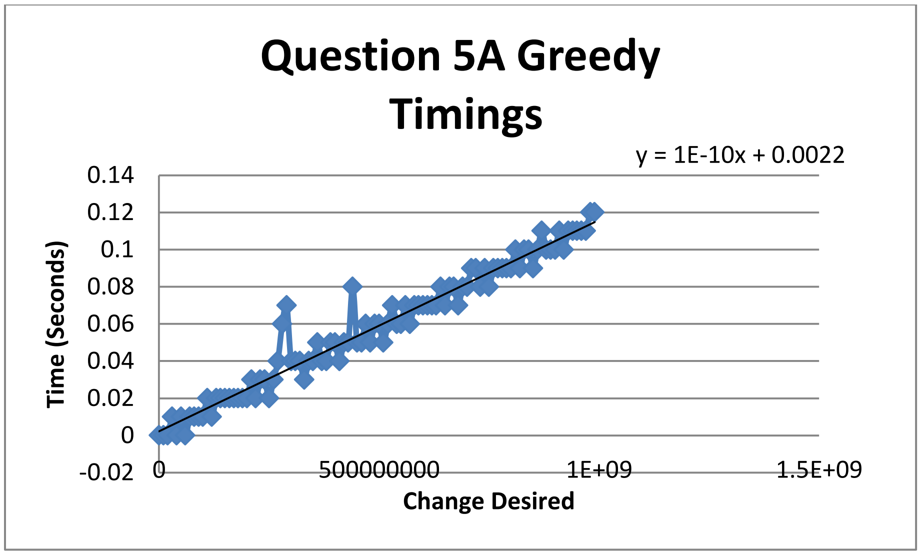
The greedy algorithm is by far the most efficient being 11 orders of magnitude more efficient than the dynamic program, which is Θ(AV). The greedy algorithm is supposed to be Θ(N) where N is some number less than A. Still, the magnitude of difference between the two algorithms should not be so big according to the theoretical assumptions of this test. I think that there is something else going on with the implementations with the algorithms resulting in such a wide difference in performance. Both greedy and dynamic programs seem to follow a linear form in relation to the size of A.

The running times of the brute force algorithm depends highly upon the size of the coinset, V. The point of inflection, where the running times quickly reach infinite, occurs much earlier when there are more than 5 coin values in set V at A = 40. The point of inflection for 5 or less coin values in set V is around 60.

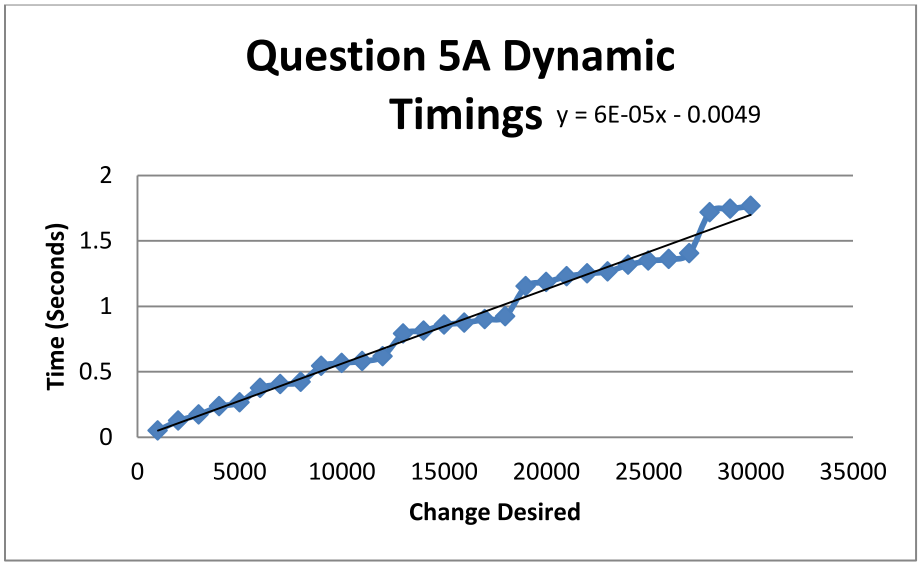

Question 4 Greedy Timings
y = 2E-10x + 0.0004
Time (Seconds)
0.25
0.2
0.15
0.1
0.05
0
-0.05
0
500000000
1E+09
1.5E+09
Change Desired


Question 4 Dynamic Timings
y = 5E-05x - 0.0075
Time (Seconds)
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0
5000
10000
15000
20000
25000
30000
35000
Change Desired

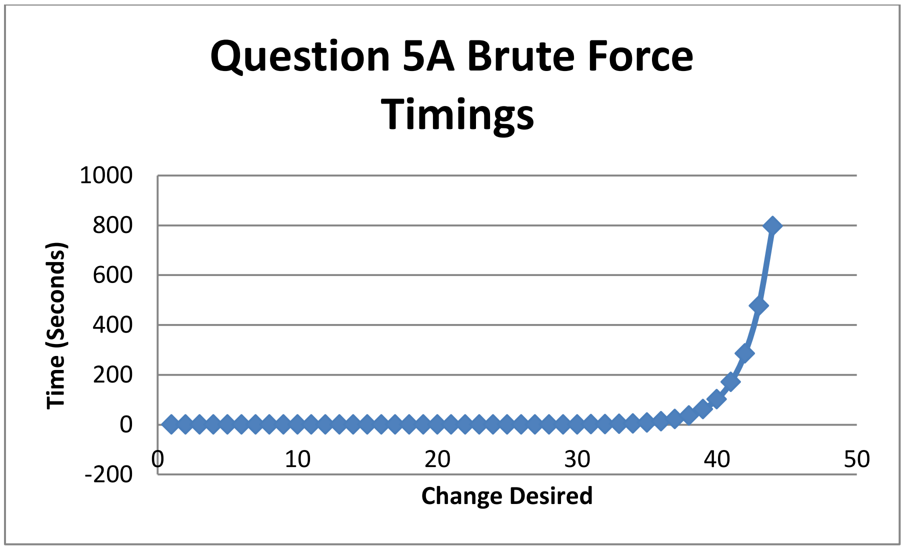

Question 4 Brute Force Timings
Time (Seconds)
800
600
400
200
0
-200
0
20
40
60
80
Change Desired



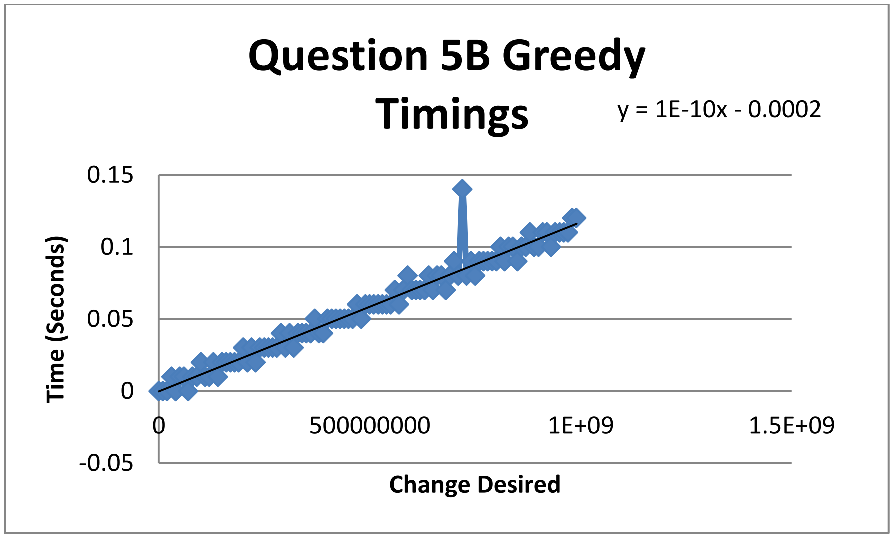
Question 5A Greedy Timings
y = 1E-10x + 0.0022
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
-0.02
Time (Seconds)
0
500000000
1E+09
1.5E+09
Change Desired

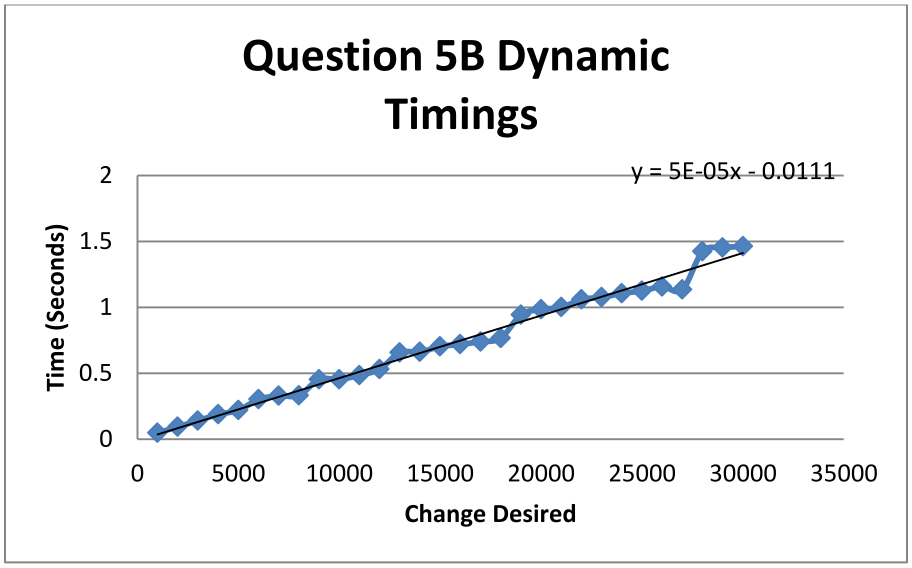


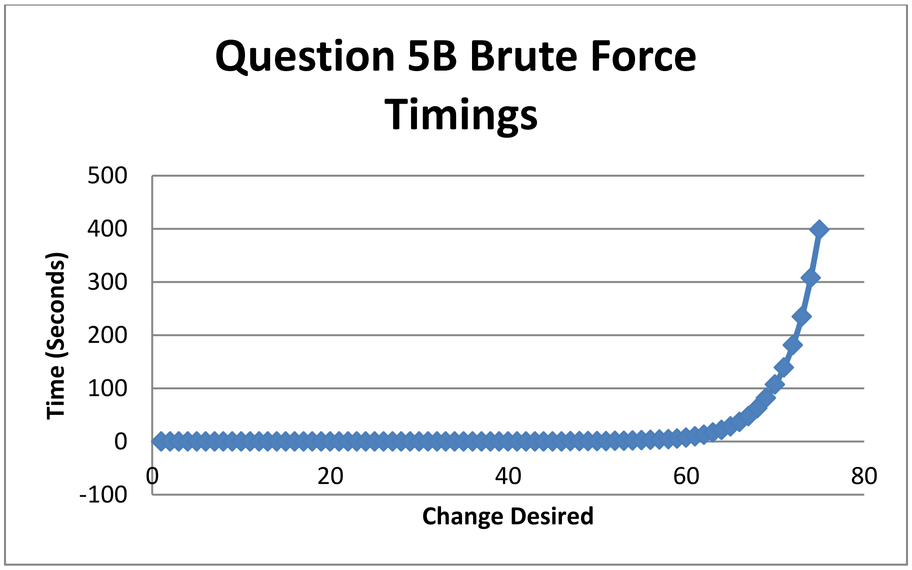
Question 5A Dynamic Timings
y = 6E-05x - 0.0049
2
1.5
1
0.5
0
Time (Seconds)
0
5000
10000
15000
20000
25000
30000
35000
Change Desired

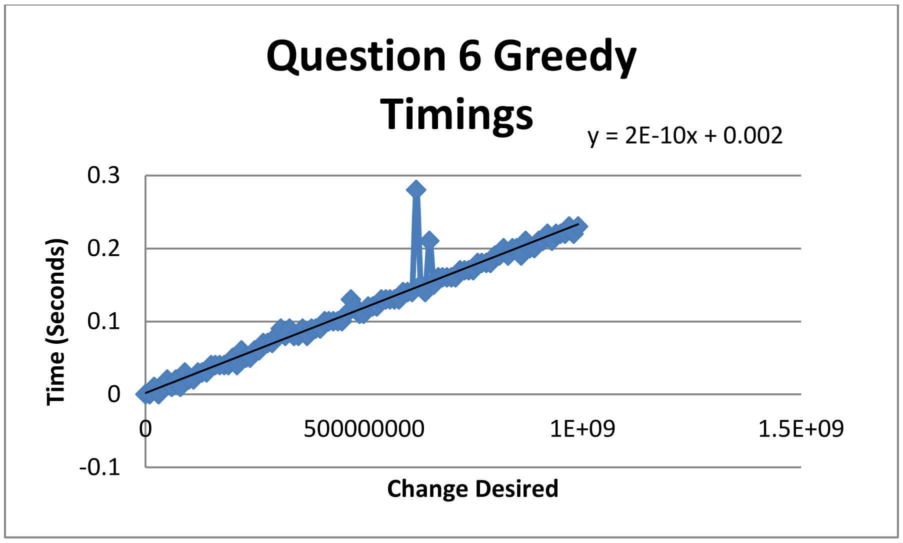


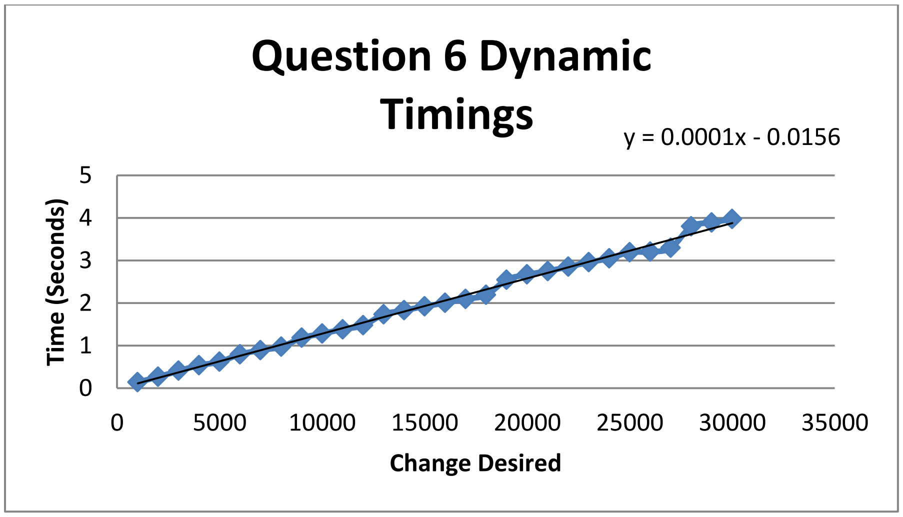
Question 5A Brute Force Timings
1000
800
600
400
200
0
-200
Time (Seconds)
0
10
20
30
40
50
Change Desired

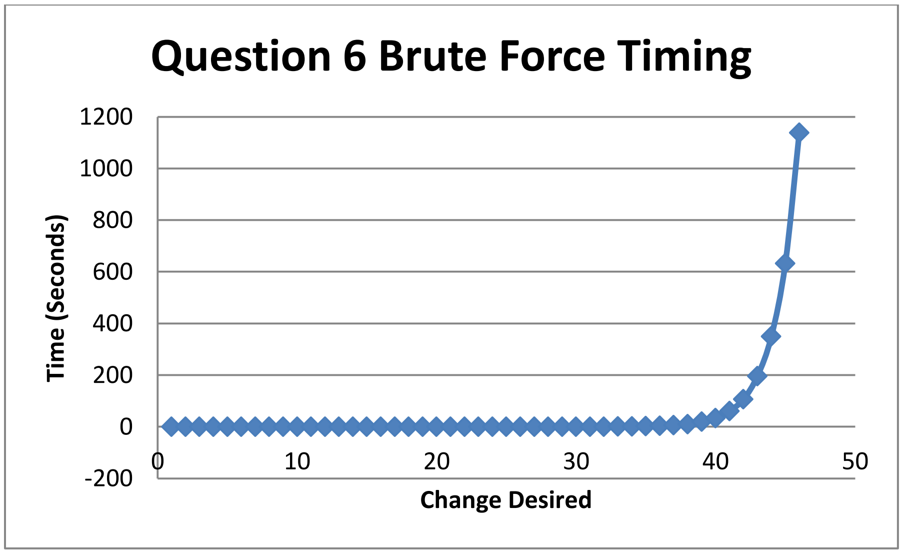

Question 5B Greedy Timings
y = 1E-10x - 0.0002
Time (Seconds)
0.15
0.1
0.05
0
-0.05
0
500000000
1E+09
1.5E+09
Change Desired


Question 5B Dynamic Timings
y = 5E-05x - 0.0111
Time (Seconds)
2
1.5
1
0.5
0
0
5000
10000
15000
20000
25000
30000
35000
Change Desired


Question 5B Brute Force Timings
Time (Seconds)
500
400
300
200
100
0
-100
0
20
40
60
80
Change Desired

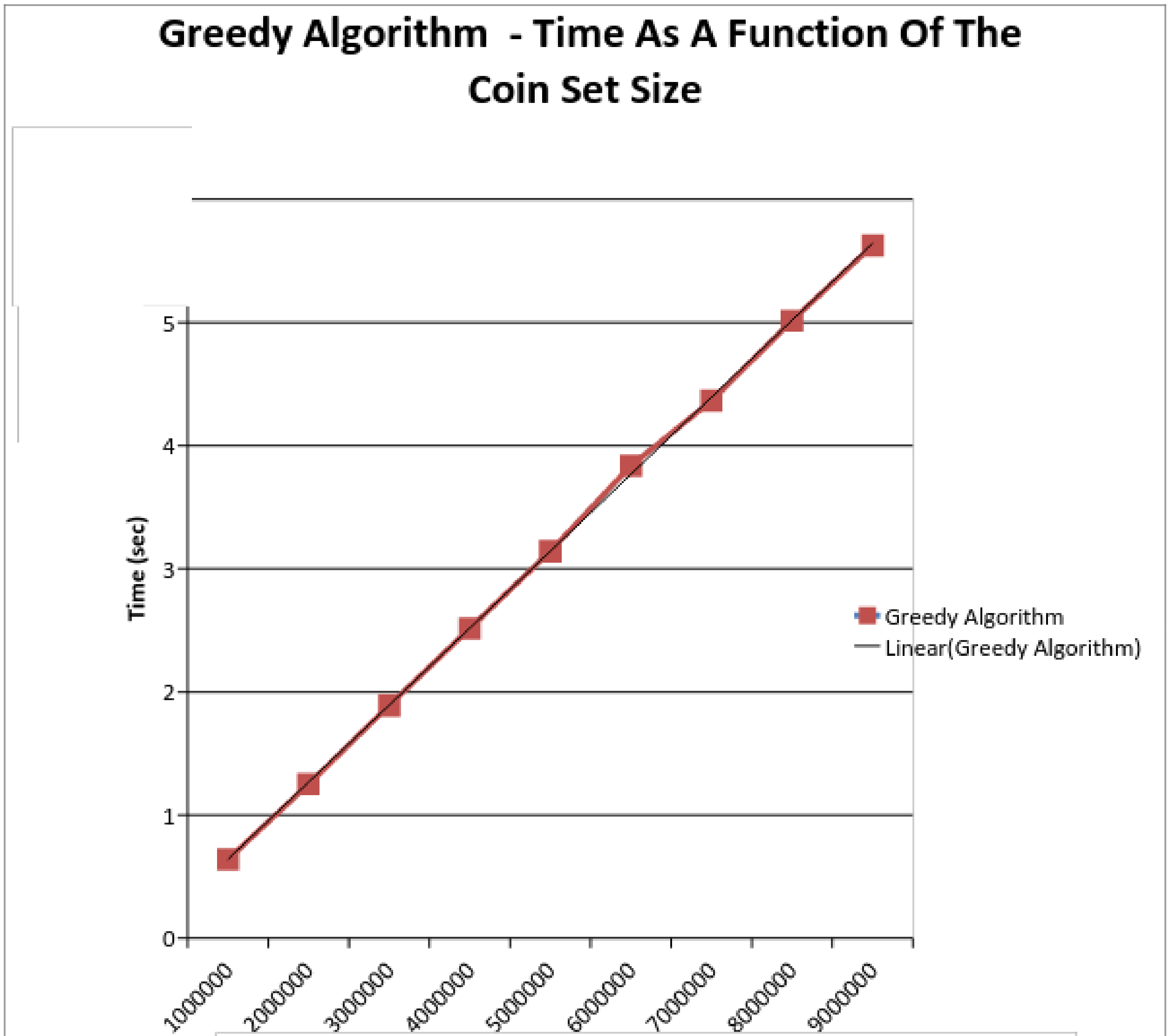

Question 6 Greedy Timings
y = 2E-10x + 0.002
0.3
0.2
0.1
0
-0.1
Time (Seconds)
0
500000000
1E+09
1.5E+09
Change Desired


Question 6 Dynamic Timings
y = 0.0001x - 0.0156
5
4
3
2
1
0
Time (Seconds)
0
5000
10000
15000
20000
25000
30000
35000
Change Desired


Question 6 Brute Force Timing
1200
1000
800
600
400
200
0
-200
Time (Seconds)
0
10
20
30
40
50
Change Desired

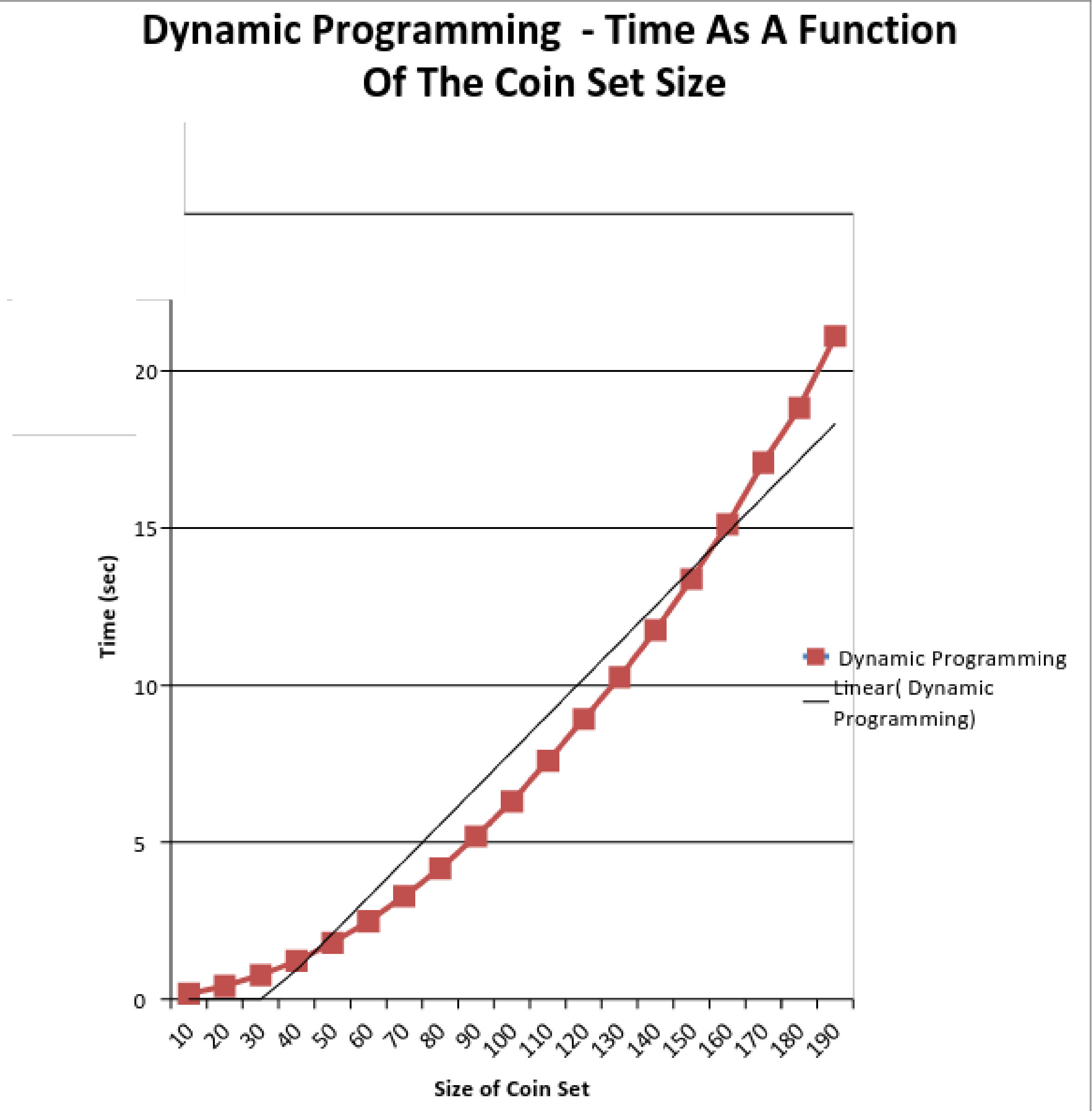
8.

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?



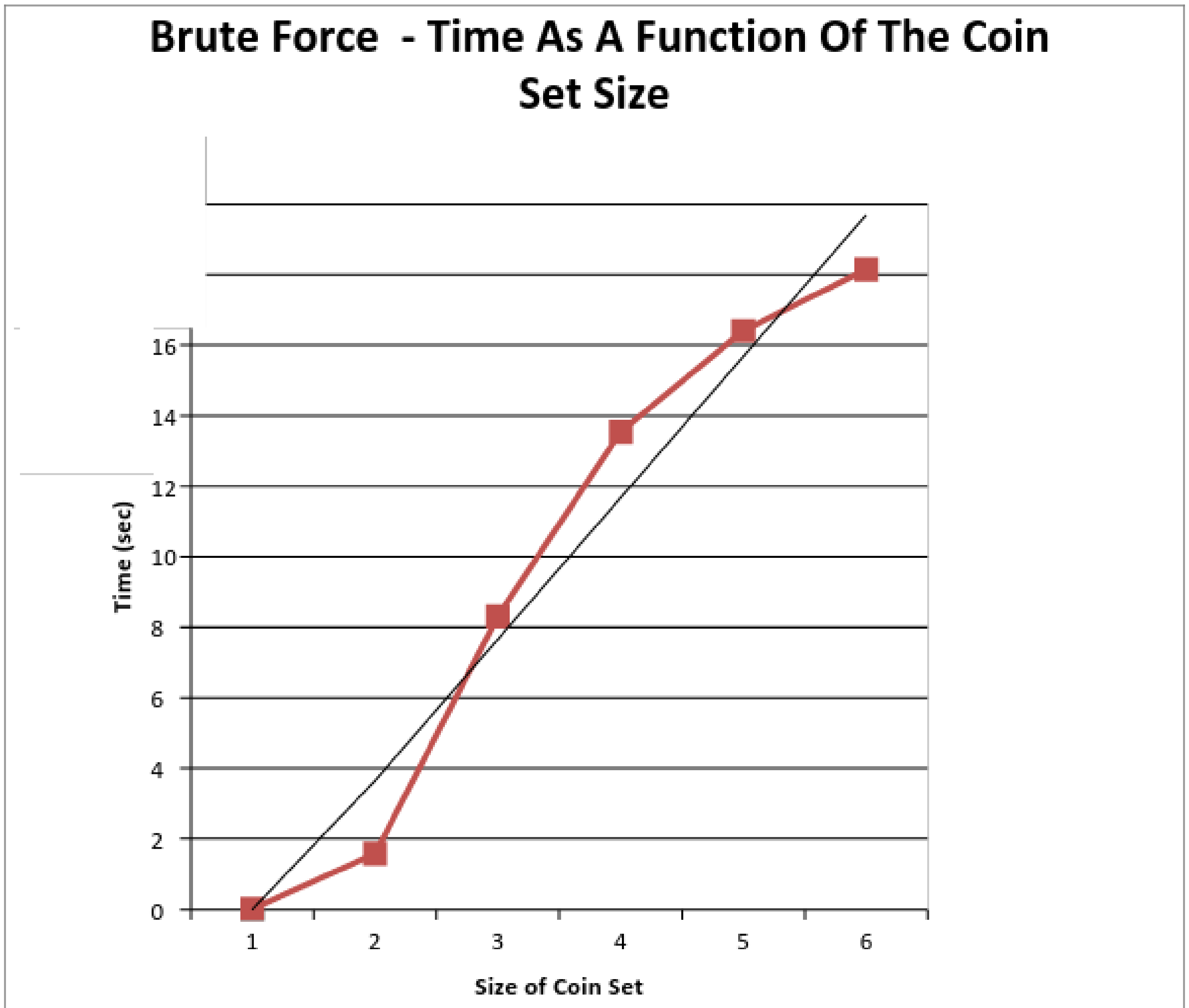
A graph of time as a function of coin set size using the greedydp algorithm. The minimum number of coins to calculate the value 2011 was calculated with an increasing coin set size from 10^6 coins to 10^7 coins. Values in the coinSet start at 1 and increased by a multiple of 2*i + 1 for every i in the coin set (ie: 1, 3, 5, 7…).

For the graph Greedy Algorithm – Time As A Function of Coin Set Size it is shown that the run time increases linearly with the size of the coin set size. This makes sense because the greedy algorithm works by repeatedly iterating through the coin set and subtracting the largest value in the coin set that is <= to the remaining V from V. So as the coin set grows the time to iterate through the set grows linearly. This linear relationship is dependent on the situation when the coinset values get larger than the value V to make change for, the algorithm does not breaking out of the loop when it realizes V is less than the remaining coins in the coin set.



A graph of time as a function of coin set size using the changedp algorithm. The minimum number of coins to calculate the value 2011 was calculated with an increasing coin set size from 10 coins to 190 coins. Values in the coinSet start at 1 and increased by a multiple of 2*i + 1 for every i in the coin set (ie: 1, 3, 5, 7...).

For the graph Dynamic Programming – Time As A Function of Coin Set Size it is shown that the run time increases quadratically as the size of the coin set size increases. This also make sense because the changedp algorithm works be iterating through the coin set once for every value of V (value to make change for). If n is the total time to run the algorithm then n = (coinSet.size) x (V) + C. Since V in the graph above is 2011 then it is always larger than the sizes of the coin sets tested, which produced the quadratic curve. However once the coin set size get much larger than V then the runtime will depend more on the coin set size and produce a linear graph.



A graph of time as a function of coin set size using the changedp algorithm. The minimum number of coins to calculate the value 2011 was calculated with an increasing coin set size from 10 coins to 190 coins.

For the graph Brute Force – Time As A Function of Coin Set Size it is shown that the run time increases as a cubic polynomial function as the size of the coin set size increases. The recursive brute force algorithm works by finding every result that produces a valid solution to the problem and then comparing that solution to the best solution found so far. The time the algorithm can finish the problem in is related to the number of recursive calls needed to solve the problem. The recursion works by iterating through the coin set and recursively calling the function again for V- coin if the coin <= V. Which gives a recursion relation of T(n) = coinSetSize * T(n - coins). So as the coin values approach the value of V then fewer recursive calls are made to solve the problem. As the size of the coins set increases then normally the values of the coin get larger, and the function performs better as the coin values approach V. If V is always much larger than the coin values then the extra coins in the coin set will dramatically slow down the algorithm since the number of recursive calls made is directly related to the coinSetSize, and the coinValues will not reduce the size of n very quickly, which gives exponential runtime.

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.

Since the values in the set share a greatest common factor, which is 3 and each higher value is a product of 3 * a lower value, then the greedy algorithm will give the correct min value to make change for any value of V. The dynamic programming algorithm will also always give the correct solution to the problem but will run much slower than the greedy algorithm. For this set of coins it would be better to use the greedy algorithm.

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

The greedy algorithm is the optima l solution to the coin change problem when the set of coins is such that the algorithm can produce the value V using the minimum number of coins. One set that gives this situation is a coin set where any greater coin in the set can be produced by some number x times a lower value coin . For example the coin set {1, 5, 15, 30} 1 x 5 = 5, 1 x 15 = 15 1 x 30 = 30, 3 x 5 = 15, 6 x 5 = 30, 2 x 15 = 30. There no quicker solution to make change for a value greater the 30 that doesn’t use the 30 cent piece.