Names and Group Number:
Group 13: Nicholas Madan , Avant Mathura , Tatyana Vlaskin
****
Testing [Proof of Correctness]:
****
We tested the validity of our algorithms by running each implementation with an input of coin1.txt file provided by the instructor. The file is submitted as part of this assignment. This file contains an array per line that consists of coin set and change value on the next line. Our main.cpp program reads in this information and calculates minimum number of coins needed to make a change. The output goes to the coin1change.txt file. Please note that slow change appends to the file if it exists, while greedy and dynamic replaces information in the file. Also during the analysis part of the assignment presented below, we were looking at the results and were addressing any discrepancies in values, which happened only with greedy algorithm which is expected.
****
****
Question 1: Describe, in words, how you fill in the dynamic programming table in changedp. Justify why is this a valid way to fill the table?
First, the dynamic programming (dp) table is set to 0 at change amount 0 (index 0).
For each possible amount of change from 1 to the given amount of change (change), we iterate over the array of coin values.
If the value at each point in the coin values array (array_Available_Coins) is less than the selected amount of change, AND if the value in the dp table at the index of the value of the amount of change minus the current coin value is less than the minimum value, then the minimum value is set to the value at that index in the dp table plus 1.
After all coin values are iterated over for each change value, the final minimum value is pushed into the dp table.
Another way to say is that we have a table with available coins as column and required change as rows. Each cell in the table has a minim number of coins for that specific change. The base of the table is filled with zeros, which means that the 1st row is filled with zeros and so on. For example, if our available coins are [1,5,6,8], A=11. Then the table would look like the following:
HORIZONAL- different change amount
VERTICAL – available coins

	0	1	2	3	4	5	6	7	8	9	10	11
1	0	1	2	3	4	5	6	7	8	9	10	11
5	0	1	2	3	4	1	2	3	4	5	2	3
6	0	1	2	3	4	1	1	2	3	4	2	2
8	0	1	2	3	4	1	1	2	1	2	2	2

This is a valid method for filling the table because it account for all possible coins value all the way to the required change value. At each coin value, the smallest possible amount of coins for that value is placed into the table, so each successive value will also have the smaller value.
//https://www.youtube.com/watch?v=Y0ZqKpToTic

****
****
Question 2: Give pseudocode for each algorithm.
****
****
--SLOW CHANGE ALGORITHM--:

```
        Given vector <int> values with the given coin denomination values
        and an integer T_value change representing the amount of change to be made:
        Declare following 3 vectors:
           vector<int> coin;
           vector<int> calculate;
           vector<int> Results;
           vector<int> values;
           for each k from 0 to values.size()
              //initialize all coin.at() with value.at()
              push values.size into coin.size;
           If T_value is 0
                No coin is used
           Else if T_value is 1
                Only 1 coin is used.
           Else
                Call compare_change to get the optimum coin set
Procedure compare_gate
        For each w from 0 to t_value
                Initialize number_of_coins
                Initialize calculate_1 for w
                Initialize calculate_2 for value - w
                Call changeslow for calculate_1
                Call changeslow for calculate_2
        Find the minimum of change

Procedure changeslow        // recursive function
        For each w coins.size()
```

```
          if w is coin.at()
              Increment coin.at()
              Increment sum with coint.at()
              Push coin.size() – w into temp
              Set the equal flag
              Exit
      If equal flag is set
              If sum and value are equal
                      Return the c as change
              Else
                      Call changeslow    //recursive call
              Else
                      Call changeslow            // recursive call
```

****
****

--GREEDY COIN CHANGE ALGORITHM--:

```
              Given vector <int> array_Available_Coins with the given coin
              denomination values and an integer value change representing the
              amount of change to be made:
                // will be used to count each coin denomination
              Initialise vector<int> array_Coin_Count;
              for each j from 0 to array_Available_Coins.size()
                     //initialize all coin denominators to zero
                     push 0 into array_Coin_Count;
              for each i from array_Available_Coins.size()-1 to 0:
                     while the value at array_Available_Coins[i] is less than change:
                            subtract the value at array_Available_Coins[i] from change
                            add 1 to the value array_Coin_Count[i]
```

****
****

-- DYNAMIC COIN CHANGE ALGORITHM--:

```
      Given vector <int> array_Available_Coins with the given coin denomicatio values
      and an integer value change representing the amount of change to be made:
      Decler 3 following vectors:
         vector<int> array_Coin_Count;
         vector<int> array_Minimum_Coins;
         vector<int> optimal_Set_of_Coins;
         vector<int> array_Available_Coins;
         for each j from 0 to array_Available_Coins.size()
            //initialize all coin denominators to zero
            push 0 into array_Coin_Count;
        for each i from 0 to change:
            min = INT_MAX(system maximum integer size):
            for each j from 0 to array_Available_Coins. size():
                    if the value in array_Available_Coins[j] is less than i
                            if the value in array array_Minimum_Coins at index
[i-array_Available_Coins[j]+1 < min
                                    min = value in array array_Minimum_Coins at index [i-
array_Available_Coins[j] +1
                                    coin = j
            push the value of min into array array_Minimum_Coins (the dp table)
            push the value of coin into array optimal_Set_of_Coins {the coin table)
while the value of change is greater than 0:
            //increment coin position for the current change value
            array_Coin_Count[optimal_Set_of_Coins[change]]++
            //decrement change by the coin value
            change -= array_Available_Coins[optimal_Set_of_Coins[change]];
//count total number of coins needed to give a change
int total_Coin_Count =0;
            for (int k = 0; k < (int)array_Coin_Count.size(); k++){
                    total_Coin_Count = total_Coin_Count + array_Coin_Count[k];
}
```

****
****

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

Basic Step:

T[0] =0. This is true because for 0 cents, the optimal number of coins used is 0. It true by definition

Inductive hypothesis:

We make an assumption that for some arbitrary value k:

T[k] is the minimum number of coins used to make change for k cents.

This also assumes that T[m] is also correct, where m is any value less than or equal to k, due to the fact that they have been completed to reach T[k]

Inductive step:

We need to prove that T[k+1] is correct and will produce a minimum set of coins as an answer. T[k+1] will look though all coins with values of V[i] less than or equal to k+1. It will also compare the numbers of coins for solutions T[(k+1) –V[i]] for every value V[i] that is less than or equal to k.

From here we have 2 possibilities:

1. There exists a value pf V[i] that is equal to (k+1) so that (k+1)-V[i] = 0, and T[k+1] = T[0] +1
2. There does not exist a value of V[i] that is equal to (k+1). Hence we check every value V[i] that is less than (k+1) and subtract for that coins value (k+1). Then we compare the remaining amount T[k+1 –V[i]] against the solution table that we already have.

   Due to the fact that we have already produced the solution in the dynamic manner, we have already computed all minimum solutions from T[0,…k] and we know what (k+1-V[i]) is a positive integer that is less than (k+1).

   We can retrieve all solutions T[v+1 –V[i]] and select the one with the smallest coin set, adding V[i] to its coin set to get the answer for T[k+1].

   From our hypothesis all values of T[1,…k] will produce the smallest coin sets and since k+1-V[i] for any V[i] is going to be between 1 and k, that will give us the optimal solution, which we then add 1 to and account for the coin V[i].

   I got this from: https://github.com/awesome-cs325/coin-change/blob/master/report/project1.pdf

   My original inductive step was the following:

   Since T[k+1] = T[(k+1) –i] +1, where i is some value less than or equal to k.

   Then:

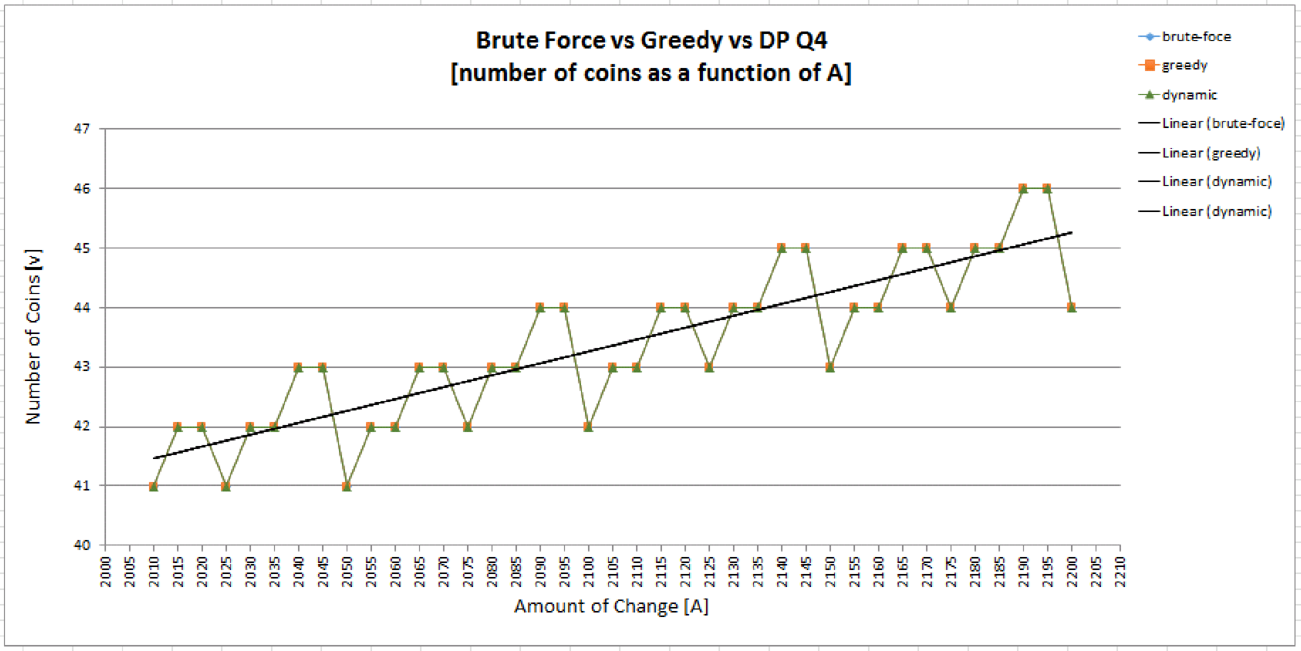
   T[k+1] = T[m] +1, where m is some value less than or equal to k

   Since we know T[m] is the correct number of coins used for m, T[k+1] must also be the correct number of coins used for k+ cents.

****

****

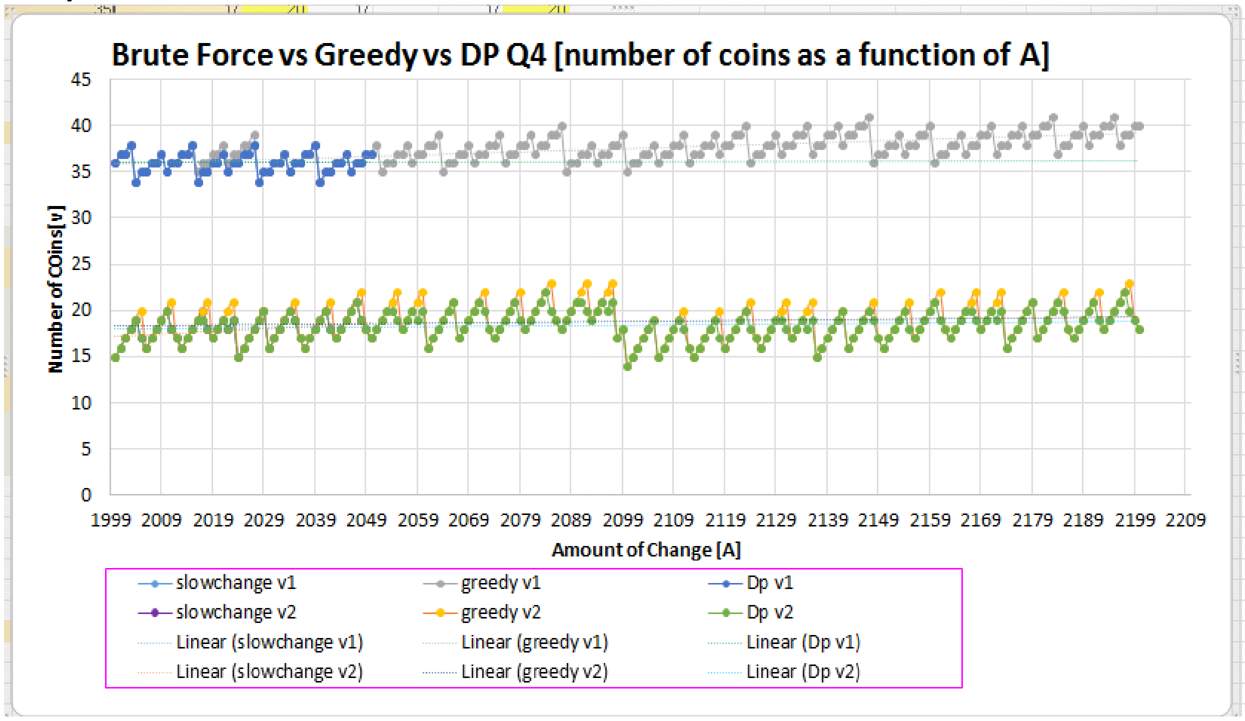
4. Suppose *V* = [1, 5, 10, 25, 50]. For each integer value of *A* in [2010, 2015, 2020, …, 2200] determine the number of coins that changegreedy and changedp requires. You can attempt to run changeslow however if it takes too long you can select smaller values of A and also run the other algorithms on the values. Plot the number of coins as a function of *A* for each algorithm. How do the approaches compare?



The figure about is the graph of the number of coins required to make change for A for the Brute Force, Greedy and Dynamic Programming algorithms where V = [1, 5, 10, 25, 50] and A is values in [2010, 2015, 2020, ..., 2200].   Although Greedy algorithm is not guaranteed to derive the minimum number of coins for any given A, when we ran the program for the all the three algorithm, we observed that all the algorithms produced the same results. The graph above shows that the data is overlapping. Thus, the specific set of coins V = [1, 5, 10, 25, 50] in this case allows it to be correct for all values of A. There are local mins and local max for the number of coins, however, on the global level the amount as the change goes up, the number of coins goes up as well, which is demonstrated by the linear best fit line.

****

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]. You can attempt to run changeslow however if it takes too long you can select smaller values of A and also run all three algorithms on the values. Plot the number of coins as a function of *A* for each algorithm. How do the approaches compare?



When data sets v1 = [1, 2, 6, 12, 24, 48, 60] and v2 = [1, 6, 13, 37, 150] were ran using all three algorithm. Greedy algorithm and dynamic programming algorithm produced different results. Brute Force algorithm and dynamic programming algorithm produced the same results. By looking at the graph above, greedy algorithm had produced higher values of the coin compared to the other algorithms for some values of changes. Yellow and gray color lines are greedy algorithm and dynamic programming algorithm is blue and green color. Brute Force algorithm is purple and light but is overlapped by DP so you can’t see it in the graph. The explanation that the greedy fails is that the choices made by the greety may depend on choices made so far but not on the future choices or all the solutions to the subproblems. Greedy algorithms iteratively makes once choice after another, an in such way reduces each given problem into a smaller ones. Which makes it solve the problem quickly, but it never reconsiders its choice. Greedy work well for problems, which have optimal substructions, while in this case we got few NOT optimal subproblems. This is the main difference from dynamic programming and from the recursive slow change algorithms. After every stage they make decisions based on all the decisions made in the previous stages, and may consider previous path to 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. You can attempt to run changeslow however if it takes too long you can select smaller values of A and also run all three algorithms on the values. Plot the number of coins as a function of *A* for each algorithm.

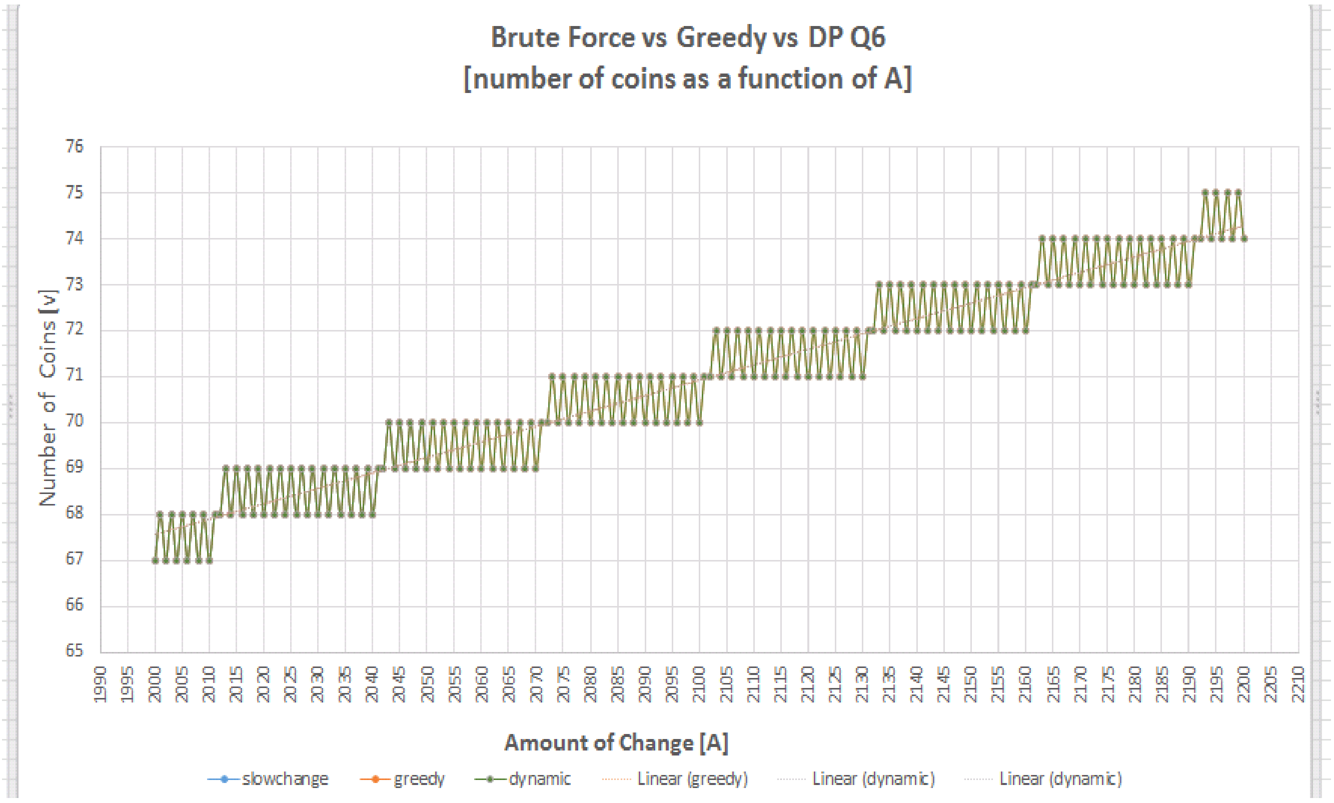
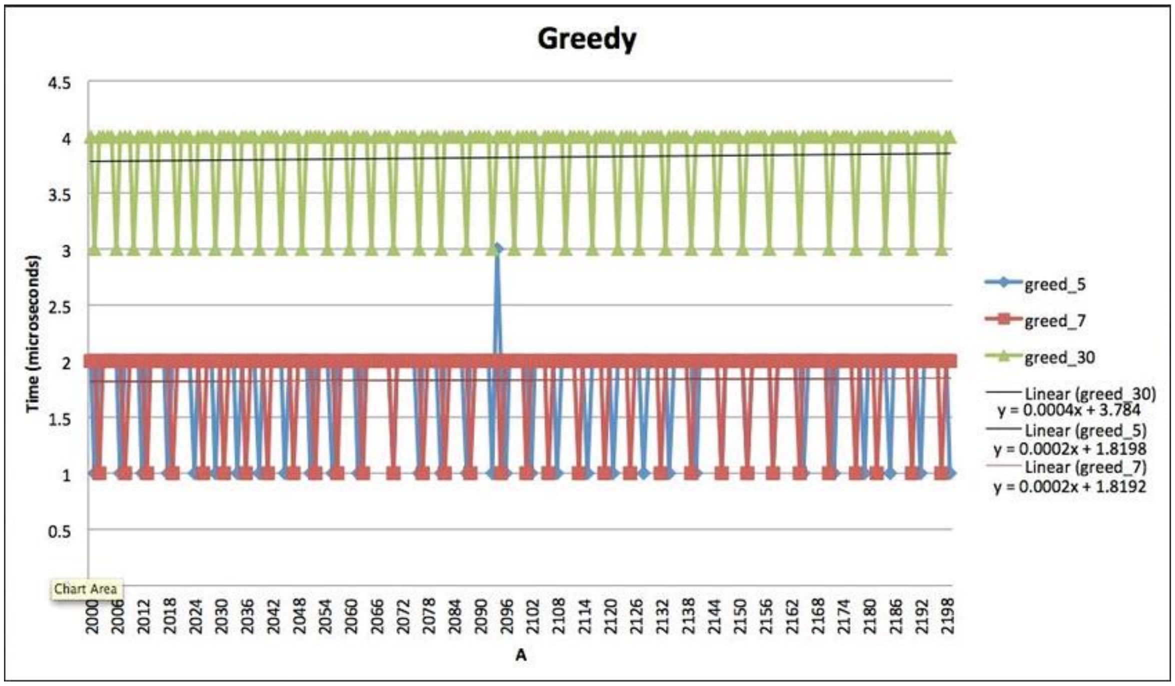
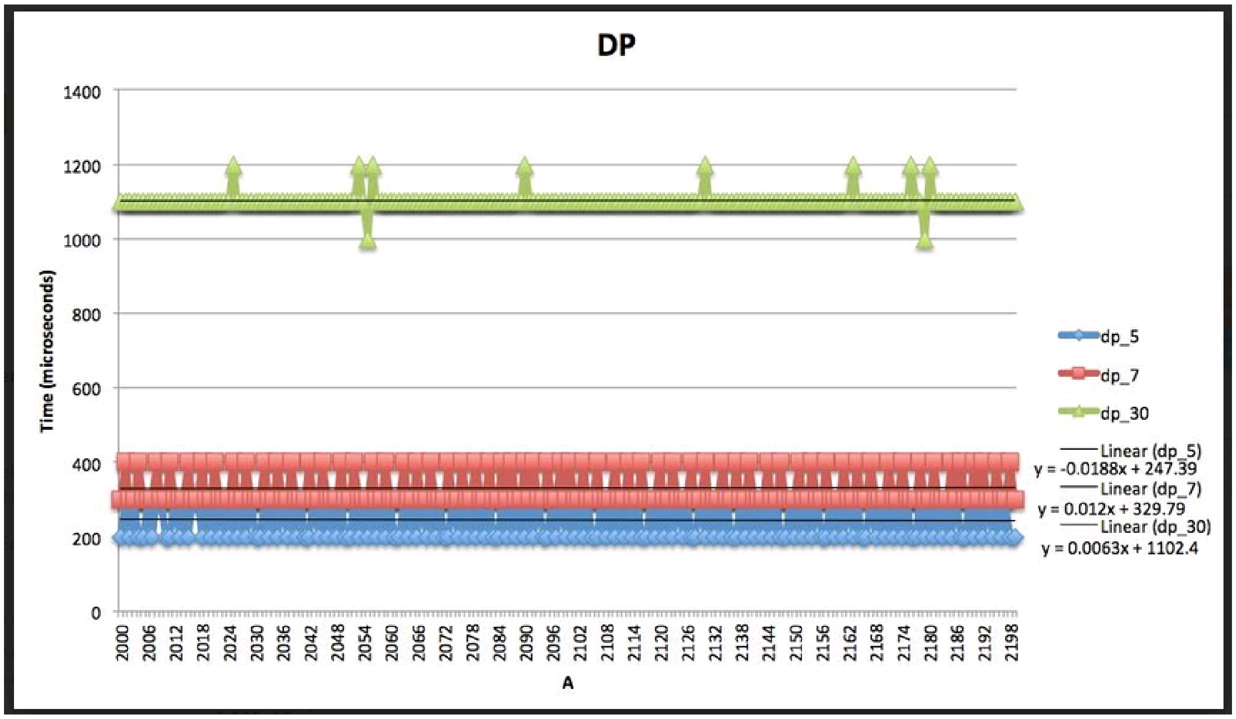


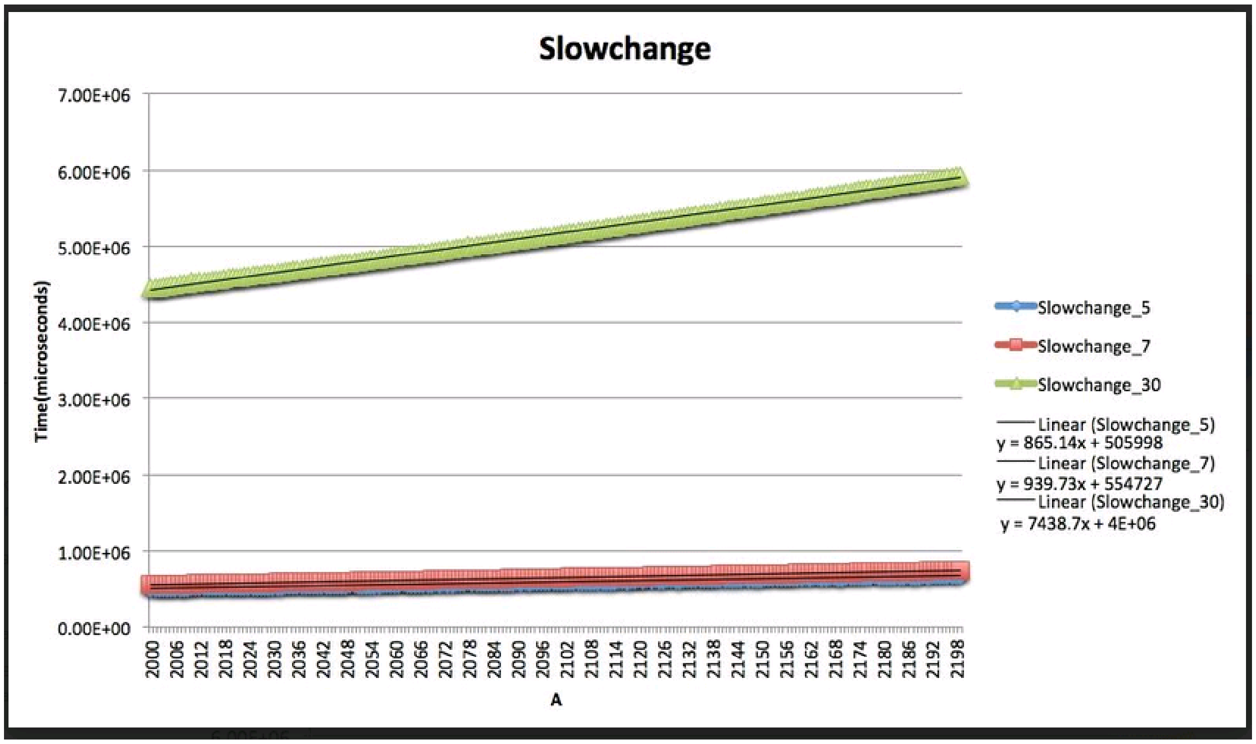
Figure above is the graph of the number of coins required to make change for A for the Greedy and Dynamic Programming algorithms where V = [1, 2, 4, 6, 8, 10, 12, ..., 30] and A is values in [2000, 2001, 2002, ..., 2200] and for the slow change algorithm where A is values in [2000, 2001, 2002, ..., 2050]. We have chosen to stop at A of 2050 for the slow change because it runs extremely slow. From the graph above we observe that all 3 algorithms produce the same results up the change of 2050, please note overlapping curves. Greedy and Dymanic algorithms were tested up to the A of 2020 and both of the algorithms produces the same results. Although the Greedy algorithm is not guaranteed to derive the minimum number of coins for any given A, the specific set of coins V in this case allows it to be correct for all values of A.

****
****
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. Graph the running time as a function of A. Compare the running times of the different algorithms.

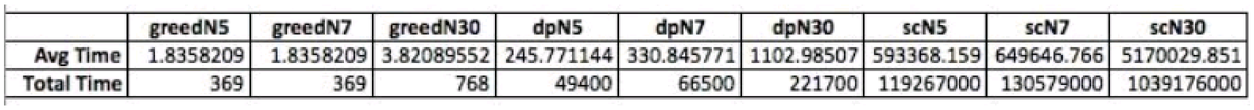
3 figures below present individual timing as a function of A graphs for both the Dynamic Programming, Slow Change and Greedy Algorithms with their respective scales for time (milliseconds). All 3 algorithms were run for the A in the range of 2000 -2020 for 3 different numbers of denominations: 5, 7 and 30. The time between algorithms varies from 1 millisecond to 6E6 milliseconds, so due to the scaling issues we have not made one combined graph for all three algorithms.







According to the run times recorded, the greedy algorithm runs the fastest, with DP coming in second, and finally the slowchange last. All algorithms were timed for each value of (A) from 2000 to 2200. Three different coin array sizes (n) were used to compare the results.   The average time and total time (in microseconds) can be seen in the given table.



	greedN5	greedN7	greedN30	dpN5	dpN7	dpN30	scN5	scN7	scN30
Avg Time	1.8358209	1.8358209	3.82089552	245.771144	330.845771	1102.98507	593368.159	649646.766	5170029.851
Total Time	369	369	768	49400	66500	221700	119267000	130579000	1039176000

From the graphs, we can see that as (A) increases, the time taken to run all algorithms also increases linearly.   Likewise, as the number of coins available increases, the time take to run all algorithms also increases.   If we consider an (n) value of 30 and compare the run times of each algorithm, greedy runs approximately 289 times faster than DP, and about 1.3 million times faster than slowchange.

Looking at the graphs regardless of the number of coin denomination, we observe that greedy algorithm runs faster than dynamic and dynamic runs faster than slow change.

On average it takes:

greedy from 1 – 4 milliseconds to run

dynamic from 200 -1100 milliseconds

slow change from 1E6 -6E6 milliseconds

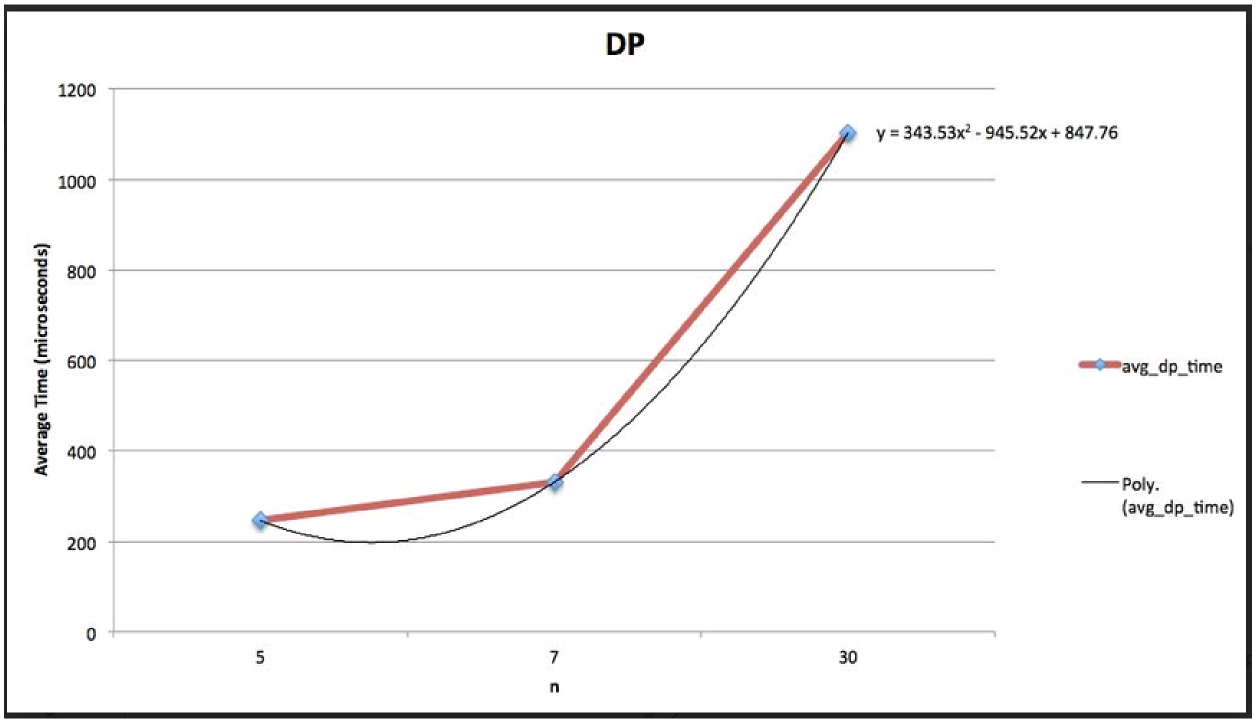
It is clear that Greedy has much better performance than Dynamic and Slow change. While dynamic has much better performance than slow change. However, the faster performance of the greedy algorithm should not be overpriced because in the previous questions we have shown that greedy does not always produce correct results. There is tradeoff between performance and accuracy.

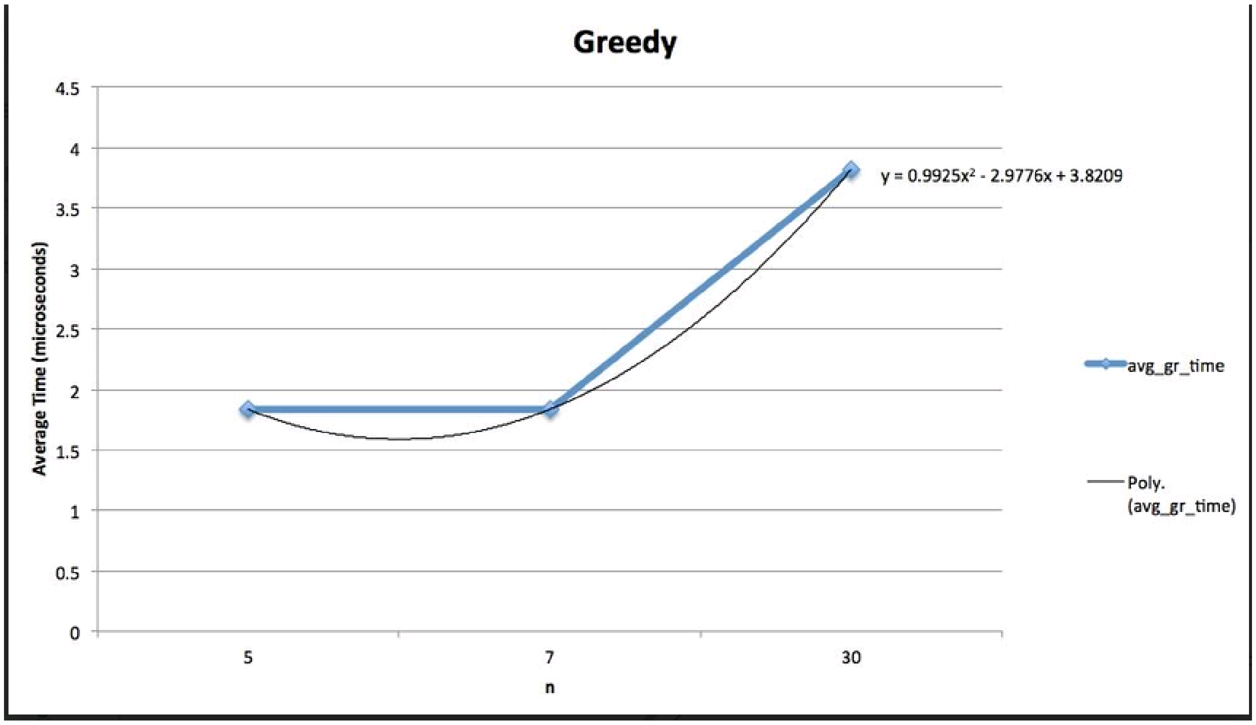
As for the slow change, even though this algorithm guarantees correct result, its extremely slow, the running time is larger than the running times for greedy and dynamic on the magnitude of 10^6 for the test A range. Thus, this algorithm is not preferable when dynamic and slow change algorithms are available. If you look at the best fit line its linear. Some of the possible explanations is that the input to the program is not the worst case input or as slated in the answer key for the practice midterm: Perhaps the algorithm runs in linear time. $O(2^n)$ means at most time $c2^n$. Even a linear algorithm is $O(2^n)$.

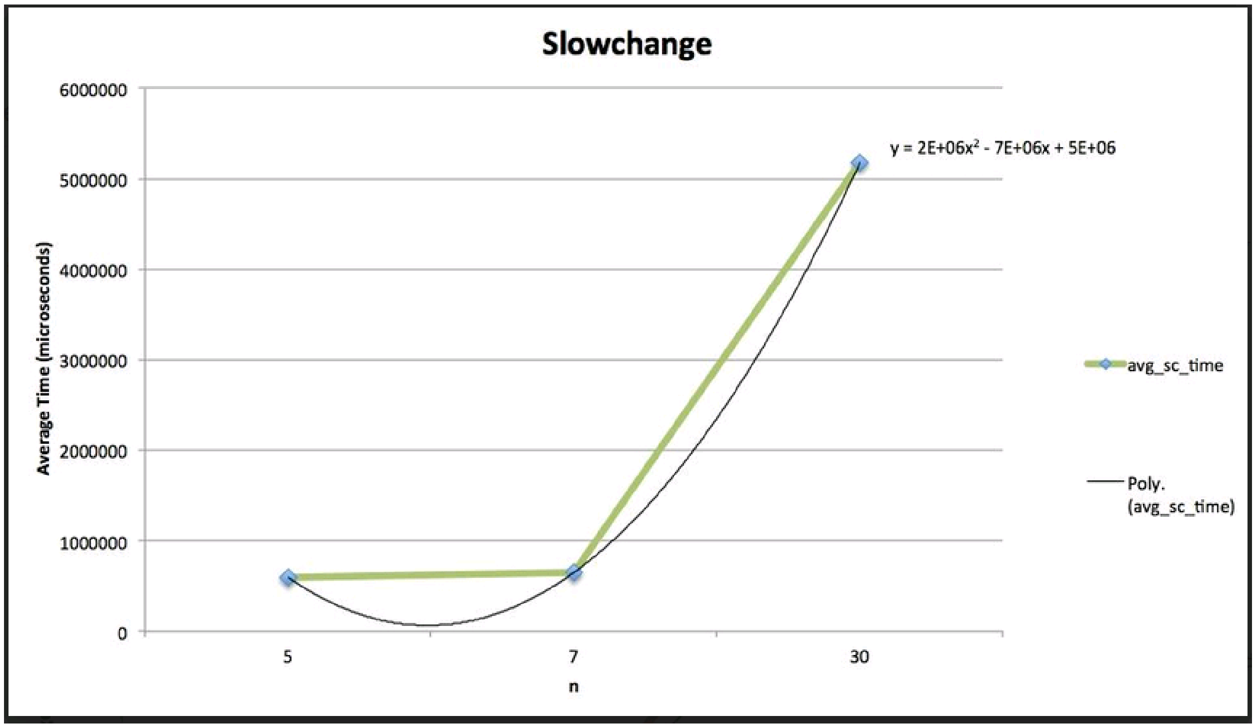
****
****

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?

3 figures below present individual timing as a function of number of denominations graphs for the Dynamic Programming, Slow Change and Greedy Algorithms with their respective scales for time (milliseconds). We only ran 3 different number of domination because its very time consuming.







Based on the experimental data presented above it can be deduced that for all three algorithms, the running time goes up as the number of the denominations goes up. The size of n seems to have an exponential influence on the runtime of all the algorithms approximately equal to n^2. All three algorithms seem to exhibit this same trend according to the graphs.

The running time for the dynamic goes up due to the fact that dynamic algorithm must calculate the minimum number of coins for each subproblem and compare it against all possible coins that were calculated previously. Precisely because number of subproblems is determined by number of coin denominations, as the number of denominations goes up it plays a substantial role on increasing running time.

The greedy algorithm demonstrates increase in running time as well as the number of denominations goes up. The explanation is that greedy algorithm must search though all possible coin denominations, so there is an increase in running time as the number of denominations goes up. However, our expectation was that the effect should be minimal because greedy searches from coins at most once. If the coins work, it takes, if it does not work, it discards it and never goes back. But its seems that that is only demonstated in slightly lower constants in from of n^2, but the running time is similarly to dynamic.

As for the slow change algorithm the running time is also affected as we increase the number of denominations. This is due to the fact that slow change algorithm is recursive. The algorithm tries to use each coin and ask the function again to get min number of coins for a smaller sum. Thus, it has exponential execution time. As the number of denominations goes up there are more coins to recurse through and taking into consideration that the algorithm has exponential running time, increasing number of denominations causes increase in the running time, n^2.

****
****

9. Suppose you are living in a country where coins have values that are powers of *p*, *V* = [*p*0 , *p*^{1} , *p*^{2} , ⋯ , *p*^{n}]. How do you think the dynamic programming and greedy approaches would compare? Explain.

Based on the timing graphs, both greedy and DP both increase in time as the number of coins increases. If p^n become larger, then there will be more coins to process and both algorithms will take longer. If (n) is similar to what we used in our experiments, then this fact will not cause the algorithms to run any slower. Since we are only looking at (A) values from 2000 to 2200, there will exist some limit the value of the coin becomes so large that it is no longer useful for the algorithms. For example, if a coin value is 2500, then it will never be used. It will only cause the algorithms to slow down since they must process this value. In addition to this, as the difference between adjacent coin values increases, the usefulness of such value will decrease. Values of coins that are close to one another will allow the algorithms to divide and conquer subproblems faster. While it is difficult to say which algorithm would benefit most from the coins of powers, it is safe to say that they will only increase in time when higher values of n are implemented.

It seems that performance would depend on the base p. For example, in base 2, the coin values are low enough to impact the number of coins required to make change within the 2000 - 5000 range. When coin value matters, such as the base 2 example, the number of coin values available, say 4 [1,2,4,8,16,32,64] verses 5 [1,2,4,8,16,32] should matter in similar way for both algorithms. In base 100, the power of exponents negates the usefulness of coin variety. For example, we can't use 1005 coin value will not be that useful for small change, which was already mentioned above.

In conclusion, since each coin value is a divisor for the next highest coin value, when p is small enough, there are not much of a gap in the coin value spread and both algorithms should demonstrate similar behavior.

REFERENCES: will be provided upon request