Spring 2017

Project 2: Coin Change

Group #14

Team: Elizabeth, Phillip, Aleita
5-2-2017

Coin Change Problem

Given coins of denominations (value) $1 = v_1 < v_2 < \ldots < v_n$, we wish to make change for an amount A using as few coins as possible. Assume that v_i's and A are integers. All values of A will have a solution since $v_1 = 1$.

Formally, an algorithm for this problem should take as input:

- An array V where V[i] is the value of the coin of the i^{th} denomination.
- A value A which is the amount of change we are asked to make

The algorithms return an array C where C[i] is the number of coins of value V[i] to return as change and m the minimum number of coins it took. You must return exact change so

$$\sum_{i=1}^{n} V[i] \cdot C[i] = A$$

The objective is to minimize the number of coins returned or:

$$\min \sum_{i=1}^{n} C[i]$$

We begin our report by analyzing the individual algorithms used to solve the problem: Slow, Greedy, and Dynamic Programming. Additional analysis is provided in section Comparative Analysis (page x), which addresses project questions 5 – 7, ending with question 3.

Slow Recursive Algorithm

The changeSlow() algorithm is a recursive algorithm that calculates the minimum amount of coins needed to make change of any A amount by comparing almost all possible combinations of the coins. Extremely inefficient, the slow algorithm considers the minimum of two possibilities, 1) using coin V, or 2) not selecting coin V. The function recursively calls itself to iterate through the coins from the top down to find the best solution. Since this problem is setup assuming that the smallest coin is 1, you will note there 3 base cases in the following code.

Slow Implementation:

The slow function is split up between 2 functions: 1) the calling function changeSlow() and 2) the calculating algorithm changeCalc(), with the help of class Result.

```
33  class Result
34  {
35      public:
36          int count;
37          vector<int> array;
38          Result() {
39              count = 0;
40          }
41          Result(int a) {
42              count = 0;
43              array.resize(a);
44          }
45  };
```

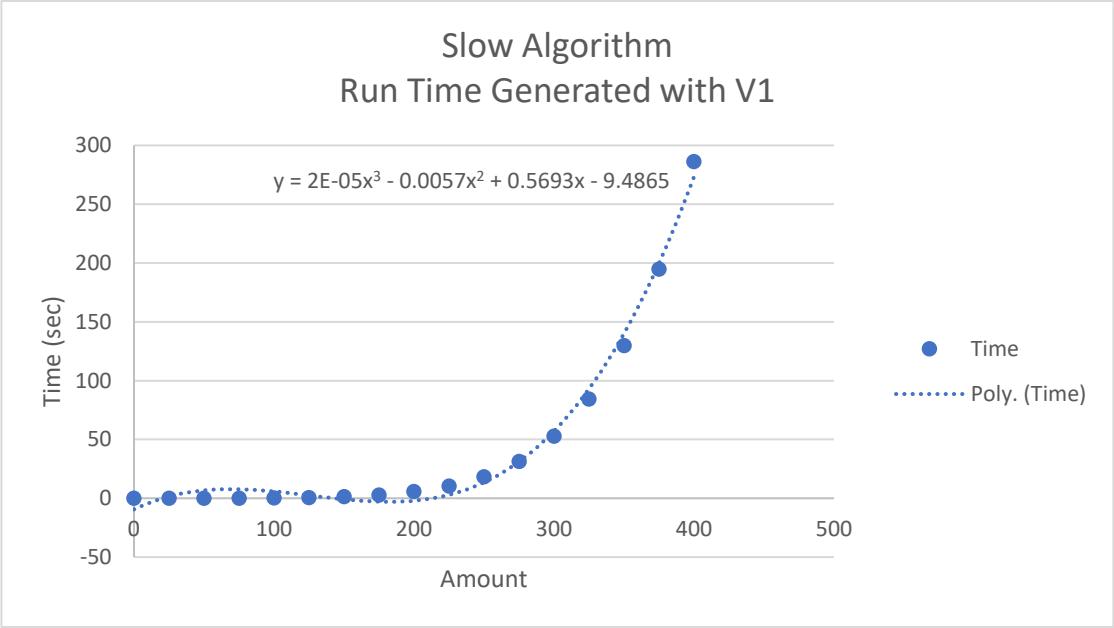
```
110  vector <int> changeSlow(vector<int> array, int amount) {
111      int vSize = array.size();
112      vector <int> returnArray;       //Create array to return  results
113      returnArray.resize(vSize);
114      int coinIndex = array.size()-1;
115      Result C = changeCalc(array, amount, coinIndex);  //Call to recursive algorithm
116      returnArray = C.array;
117      return returnArray;
118  }

 61  Result changeCalc(vector <int> V, int A, int j) {
 62      int Vsize = V.size();
 63      Result X;
 64      X = Result(Vsize);
 65
 66      if( A == 0) {       //Base case
 67          X.count = 0;
 68          return X;
 69      }
 70      if (j < 0) {        //if j < 0 there are no more coins to consider
 71          X.count = INT_MAX;
 72          return X;
 73      }
 74      if( A == V[j]) {
 75          X.count += 1;
 76          X.array[j] += 1;
 77          return X;
 78      }
 79      if(A < V[j]) {      //Iterate to coins that can be used
 80          X = changeCalc(V, A, j-1);
 81          return X;
 82      }
 83      else {  //Split problem into 2 subproblems
 84          X = changeCalc(V, A-V[j], j);   //Use coin V[j]
 85          X.array[j] += 1;
 86          X.count += 1;
 87
 88          Result Y;
 89          Y = Result(Vsize);              //Without coin V[j]
 90          Y = changeCalc(V, A, j-1);
 91
 92          if(X.count < Y.count) { //Return whichever result uses less coins
 93              return X;
 94          }
 95          else {
 96              return Y;
 97          }
 98      }
 99  }
```

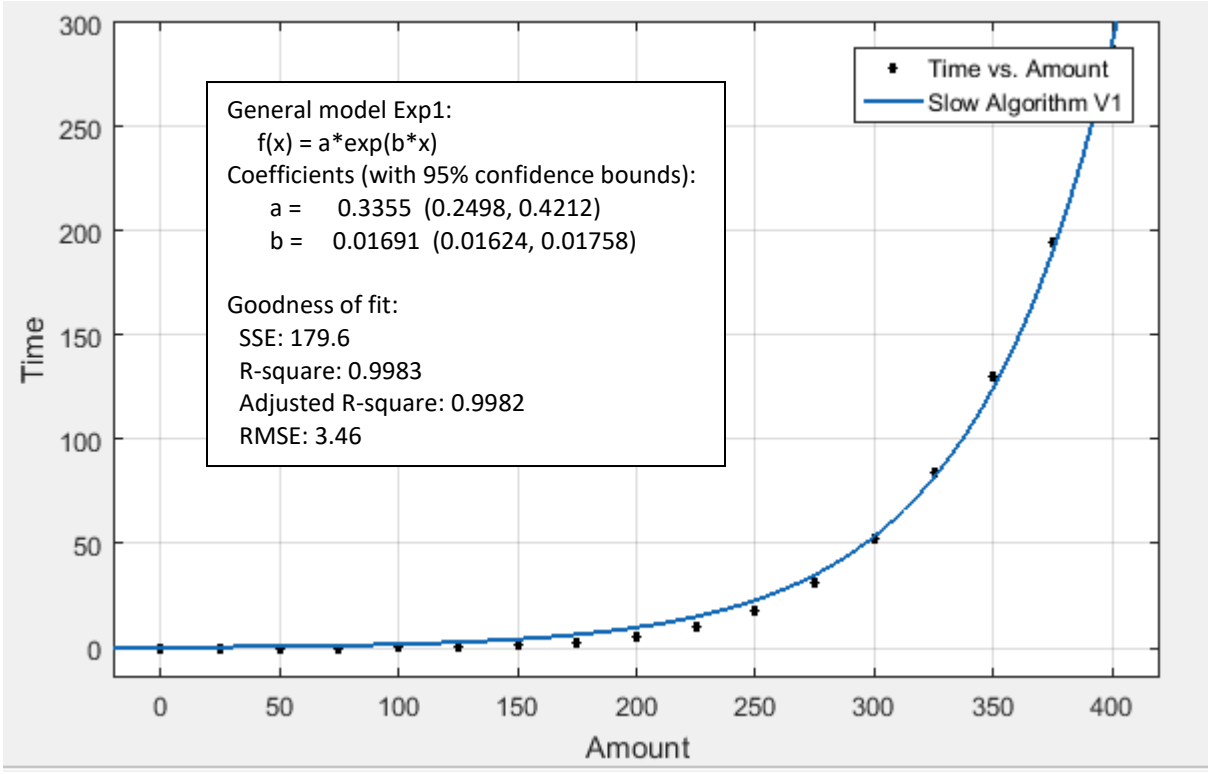
Slow Asymptotic Analysis:

Given that our algorithm depends upon 2 main inputs, A being the amount and n being the number of coins, our asymptotic runtime would consider that. In general, the asymptotic runtime for a slow algorithm to calculate minimum change is $O(A^n)$. The theoretical asymptotic runtime of our implementation is $O(3^n)$.

Slow Experimental Run-Time:



MATLAB CURVE FIT



Since there are technically 2 factors (n coins, A amount) we expected, the experimental results based on A amount are as expected. The trendline was exponential with a R value of .998 per MATLAB and third degree polynomial in Excel with R value of .992. Based on our research the brute force algorithm should be exponential.

Greedy Algorithm

The changegreedy() algorithm is a greedy algorithm that calculates the minimum amount of coins needed to make change by taking an input array V[] containing coin denominations, and a amount of change A. The algorithm returns an array that contains the same number of elements as the input array V[] but with values corresponding to the number of coins per denomination. The algorithm then uses a simple summing loop (outside pseudocode) to return a single value of the number of coins needed to make change.

The algorithm is greedy because it works off a concept of completing a single comparison that chooses a locally optimal solution that does not always create a globally optimal solution. The below pseudo code shows the changegreedy() algorithm. The local comparison simply takes the largest value coin possible, subtract the value of this coin from the amount of change to be made, and then repeats this process with the remaining amount of change for each smaller denomination coin. As can be seen from the experimental run time data later in this report, this produces a relatively superior algorithm from a timing standpoint but does not always produce the optimal results.

Greedy Implementation:

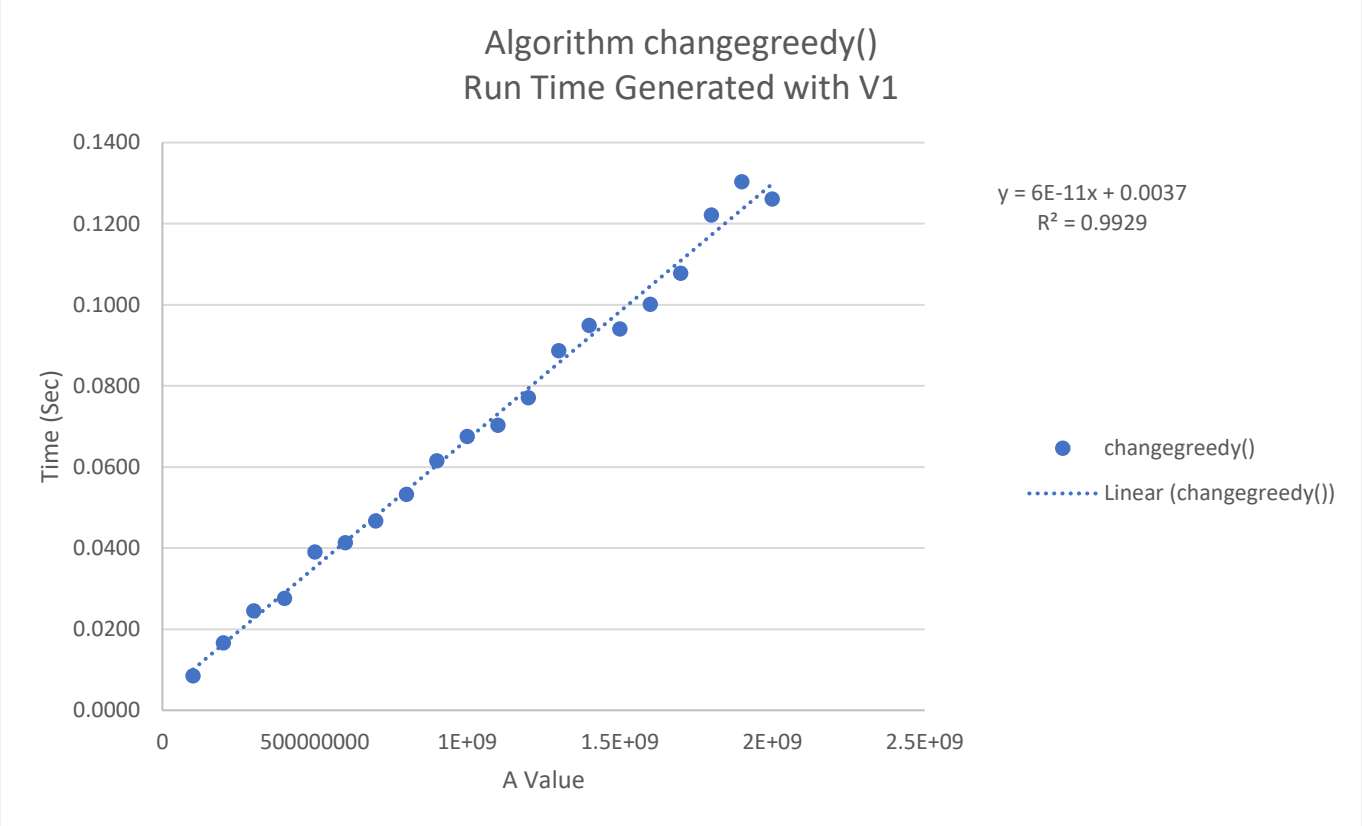
```
150  vector <int> changegreedy(vector <int> array, int total)
151  {
152      vector <int> returnArray;
153
154      int runTotal = total;
155      int coinCount = 0;
156      int coinIndex = array.size() - 1;
157      int arraySize = array.size();
158
159      while(runTotal != 0 || arraySize != 0) {
160          coinCount = 0;
161
162          while(array[coinIndex] <= runTotal) {
163              runTotal = runTotal - array[coinIndex];
164              coinCount++;
165          }
166
167          if(returnArray.size() == 0) {
168              returnArray.push_back(coinCount);
169          }
170
171          else {
172             returnArray.insert(returnArray.begin(), coinCount);
173          }
174
175          coinIndex--;
176          arraySize--;
177      }
178      return returnArray;
179  }
```

Greedy Asymptotic Analysis:

The asymptotic running time for the changegreedy() algorithm would be O(n). This was calculated based off the best and worst case scenarios of the two while loops in the above algorithm. The conditional statements at the bottom of the outer while loop will run in constant time and therefore can be ignored. This means that the limitation of this algorithm comes from the total amount of potential iterations that both the outer and inner while loops would cycle. The best-case scenario is having an input array V, with only one element having a coin denomination equal to the total amount of A. This would have exactly one iteration of the outer while loop and one iteration of the inner while loop. This essentially would run in constant time. The worst-case scenario would be having a large value for A and having only one element in V that is equal to 1. This would cause the algorithm to cycle through in inner loop n times if n is equal to A. This would run O(n). Because of this worst case run time, the asymptotic running time for the changegreedy() algorithm would O(n).

Greedy Experimental Run-Time:

The experimental run time data matches the theoretical analysis of this algorithm. The asymptotic run time should be O(N) depending on the relation of the input coins in the array V and the value of A that the algorithm is being run against. This means, that over a very large range of A values, the equation should fit within a certain degree of certainty a linear trend line. As can be seen above, this is the case with a linear trend line being fit to the experimental data with a R squared value of 0.99285.



Dynamic Programming Algorithm

The dynamic algorithm works by memorizing incremental steps of the solution. It is exhaustive in nature but reduces the overall calculations in higher value sets because the subsets or sub problems are stored in a table, the higher numbers can pull from smaller sets or by adding one. In the case of coins, as we progress through the problem for target value computation of X we are required to first calculate an x-a and then adding coin y we can store a value of y at a locating x-a and then use this set to find the most appropriate method to make change X.

The algorithm works by taking two inputs, a vector of possible coins (integers) and a target integer. We iterate through a set of possible i in range of target then loop through possible coin values to see which ones are in the appropriate range. We then find the minimum number of coins for each subset and store these in a table

DP Implementation:

```
168  vector <int> changedp(vector <int> array, int total)
169  {
170      int target  = total; // renamed total to target
171      vector<int> coinArray = array; // renamed input array of coin values
172      vector<int> coinCount(coinArray.size(), 0); // empty array to hold output
173
174      vector<int> table ( target +1, INT_MAX); // table of subs
175      vector<int> tempTable (target + 1, 0); // table of subs
176      int count = 0;  // coins count
177      int subCount; // coin count in subs
178      int tempTarget = target;
179      int temp = 0;
180
181      //for zero values
182      table[0] = 0;
183      tempTable[0] = 0;
184
185      // result table iterate through possible i in range target
186      for(int i = 1; i <= target; i++){
187          int subVal = target;
188          //loop through coin values
189          for(int coin = coinArray.size()-1; coin >= 0; coin--){
190              // is this coin in the apropriate range, less than target value?
191              if(coinArray[coin] <= i){
192                  // increase count for i
193                  subCount = table[i-coinArray[coin]];
194                  // find mininum
195                  if(subCount != INT_MAX && subCount+1 < table[i]){
196                      table[i] = subCount+1;
197                      tempTable[i] = coin;    // store coin
198                      if(subVal > coinArray[coin]){
199                          subVal -= (subVal/coinArray[coin])*coinArray[coin];
200                      }
201                  }
202              }
203          }
204      }
205
206      while(tempTarget > 0){
207          temp = coinArray[tempTable[tempTarget]];
208          for(int i = 0; i < coinArray.size(); i++){
209              if(temp == coinArray[i]){
210                  coinCount[i]+=1;
211              }
212          }
213          tempTarget = tempTarget - coinArray[tempTable[tempTarget]];
214      }
215      return coinCount;
216  }
```

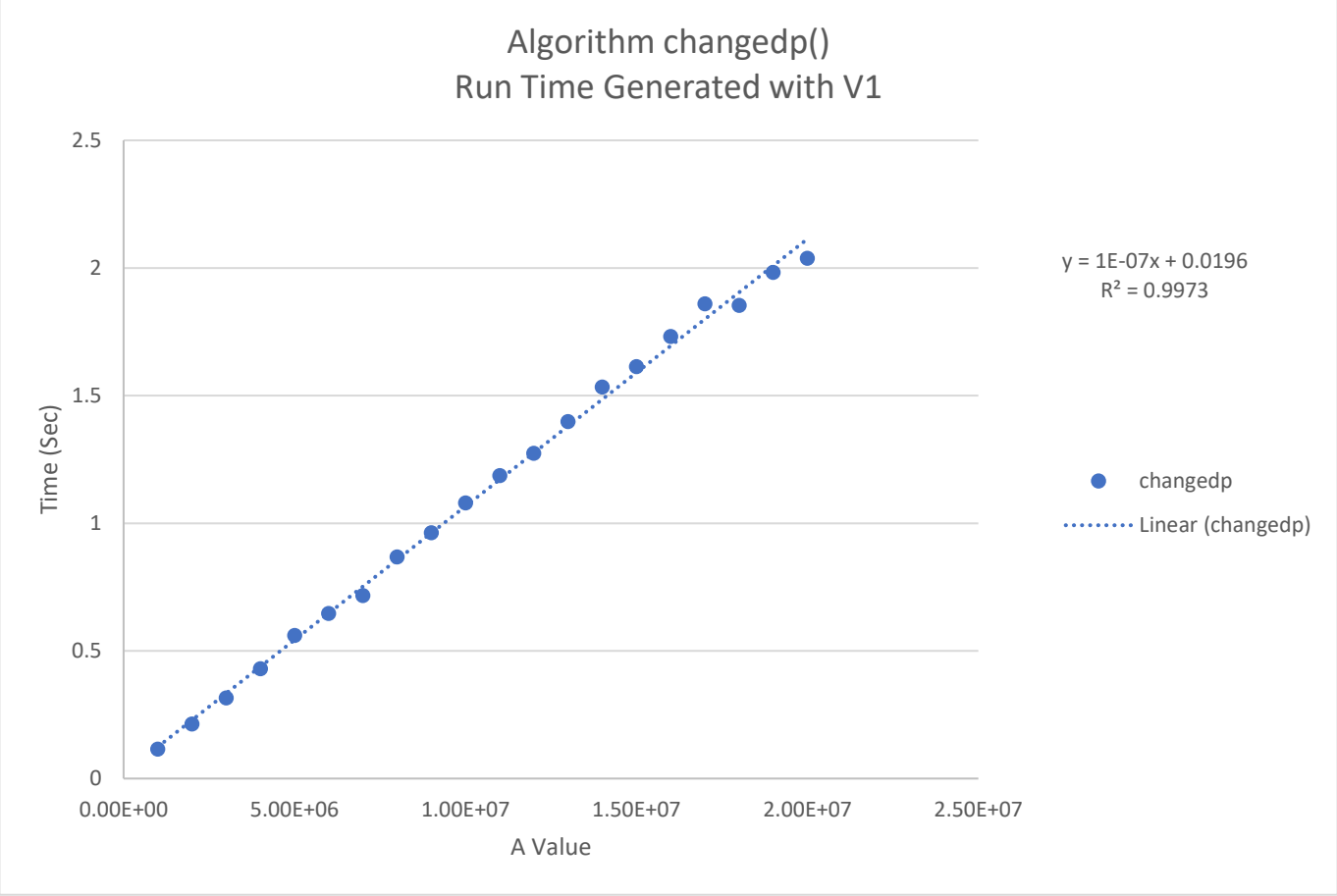
DP Asymptotic Analysis:

Based on the above implementation we are able to see one loop starting at 1 and running until target (n for reference) value (1 to n). There are three nested 'ifs' in this step where we compare the input target value to the possible coin values, and then run the table comparisons. Worst case for this loop is the value 'coin' or 'a' for easier reference. For this part of the function we have a run time of O(n*a). We run a second while loop which is also O(n*a) time because it runs through all values from target (a) to zero.

2*O(n*a) + O(a) + O(1)

Which we can reduce to O(n*a) because the other parts of the algorithm become insignificant as the algorithm continues.

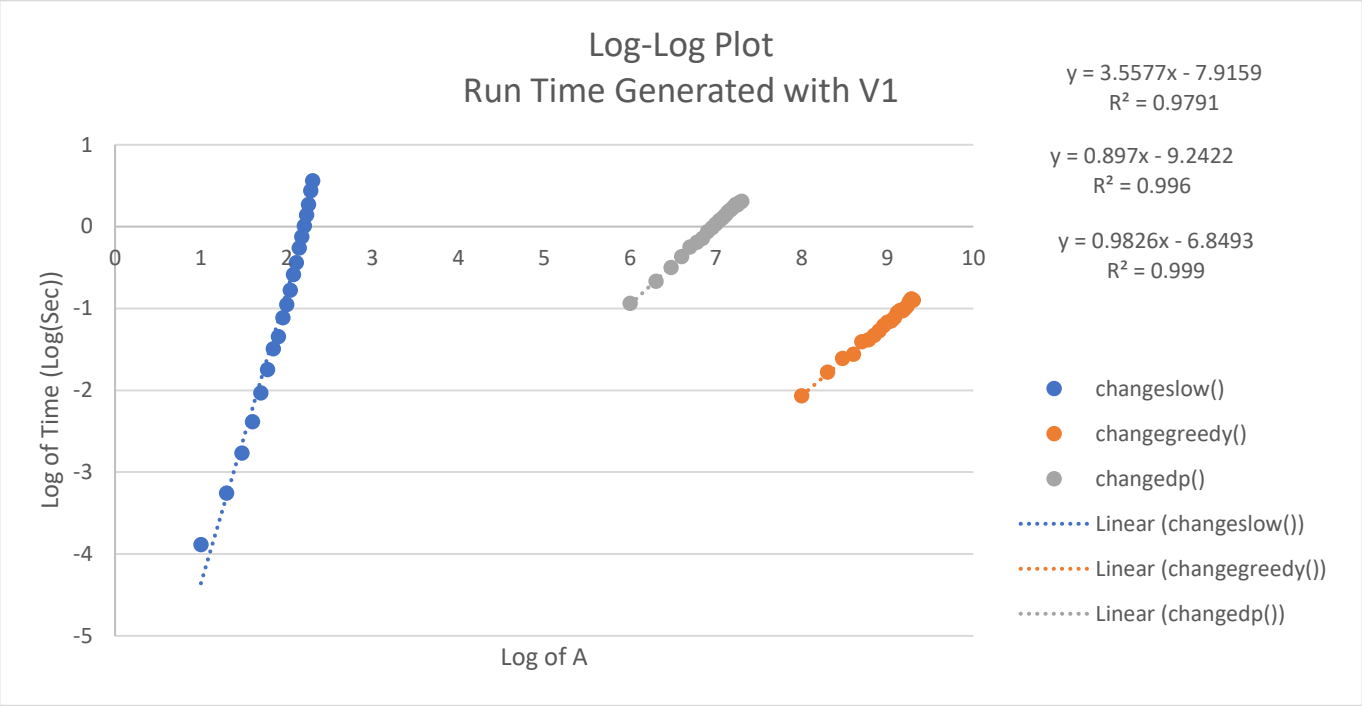
DP Experimental Run-Time:



It takes significant datasets to produce output which demonstrate the efficiency of the dynamic programming set with this problem. Greedy appears more efficient with lower values, it is not until very large value sets which we are able to see the significant changes in the dynamics timing output.

Comparative Analysis

Question #5: Log-Log Plot



The log-log plot was generated by taking the run time data of the three algorithms and plotting both axises over their equivalent log base 10 ranges. The resulting trend lines can be used to infer the asymptotic running time by directly comparing the resulting slopes to the theoretical power of N. The log-log slop for our changeSlow() algorithm makes us think that possibly the run time should be a polynomial of O(n^3), but after researching we are confident that it should be an exponential trend. The changegreedy() log-log plot resulted in a trend line with a slope of 0.897. This matches the anticipated asymptotic runtime of O(N) with a worst case slope of being 1. Because of the variation between the coin denomination array V and the input size of A, it makes sense that the run time would not match the exact slope of 1 but would be close. The changedp() log-log plot resulted in a trend line with a slope of 0.9826. This also matches the theoretical asymptotic runtime of O(N*a). For this runtime analysis, the value of "a" is the length of the coin input array V and the value of N would be the change amount A. For the runtime testing, N is relatively much larger than "a" with a size maxing out at 20 million. Because the size of "a" is only 7, the theoretical run time would behave much more like O(N). This matches the log-log trend line slope of 0.9826.

Question #6: Comparison of Greedy vs. DP

Suppose you are living in a country where coins have values V = [1, 3, 9, 27]. How do you think the dynamic programming and greedy approaches would compare? Explain.

For this array V, the greedy algorithm would outperform the dynamic programming algorithm. As discussed above in the timing analysis, for all values of A, the greedy algorithm will always outperform the dynamic programming algorithm. The only area that the dynamic programming algorithm could beat the greedy algorithm is accuracy of results. This example array, V = [1, 3, 9, 27], is unique in that each subsequent array element after the first

element is divisible by the same value of 3. In other words, there would be a perfectly linear line of slope 1 if each array element was plotted on a chart with the x-axis as the array element index and the y-axis being the log base 3 equivalent of the array value. A greedy algorithm runs optimally when the solution to the greedy local decision matches the global optimal solution. Another way of describing this is that the greedy algorithm has an optimal substructure. When in input array V is in the form where each subsequent index value has the same factor as the previous two array elements then we know that the resulting solution for the changegreedy() algorithm will have optimal substructure. The changegreedy() algorithm works by taking the largest denomination coin and counting the total number of those coins and subtracting that off the total. This continues down each denomination of coin until the base value of 1. A non-optimal substructure for the greedy algorithm occurs when the concentration of all coins returned is not on the high value coins. This makes sense because if you have the greatest combination of higher value coins possible this will overall return the fewest total coins for any amount of change A. A non-optimal local decision occurs when the preceding value coin do not share the same factor. This could lead to a situation where the the concentration of return coins is split between the high value coins and low value coins returning overall more coins than the dynamic programming algorithm, which is designed to alway return the minimum value of coins regardless of V or A. Because this above array, V = [1, 3, 9, 27], follows this common factor format, the greedy algorithm will always return the optimal result coins and will always run faster than the dynamic programming algorithm meaning it is the optimal algorithm overall.

Question #7: Denomination Sets

Give at least three examples of denominations sets V for which the greedy method is optimal. Why does the greedy method produce optimal values in these cases?

As discussed above, the greedy algorithm produces optimal values when the array V has every subsequent array index value share the same factor as the preceding two array elements. The example in the last question was V = [1, 3, 9, 27], where each array element shared a factor of 3. Other examples include:

V = [1, 2, 4, 8]	All share factor of 2.
V = [1, 3, 9, 27]	All share factor of 3.
V = [1, 4, 16, 64]	All share factor of 4.
V = [1, 5, 25, 125]	All share factor of 5.
V = [1, 6, 36, 216]	All share factor of 6.

These all share the same form of V = [x^0, x^1, x^2, x^3] where the value of x would be any arbitrary factor. Continuing off of the example above, all of these arrays, follow the form of producing linear lines on a log plot where the x-axis is the array index value and the y-axis is the log base x of the value stored in that array index. Once again, continuing off of the explanation produced in the question above, all these arrays produce optimal results for the greedy algorithm because they all produce optimal substructure for the changegreedy() algorithm. Having each subsequent index value share the same factor, it eliminates the chance of skipping a coin value index that could then replace that coin with a higher combination of lower value coins. An example given in the project description documentation shows a non-optimal input array of V = [1, 3, 7, 12] that does not follow the form V = [x^0, x^1, x^2, x^3]. This produces a return of C = [0, 1, 2, 1] for the dynamic programming algorithm and a C = [2, 1, 0, 2] for the greedy algorithm. This clearly shows how the greedy algorithm can produce sub-optimal results if the input array does not produce optimal substructure. Having an input array follow the form V = [x^0, x^1, x^2,

x^3] produces optimal substructure because each array index will always produce the highest possible amount of coins for that array index.

Question #3 : Number of Coins as a function of Amount

See plots following analysis.

The three approaches all appear to produce the anticipated results from this experiment. The idea behind the three algorithms is that the changeslow() and changedp() algorithms should always produce the fewest number of coin combinations with changeslow() asymptotically running the slowest of the three algorithms and changedp() running slightly slower than the changegreedy() algorithm. The changegreedy() algorithm should run the fastest of the three algorithm but will not always produce the smallest value of coins for certain ranges of A and for certain input arrays V. The experimental timing data confirms the theoretical timing differences of these three algorithms and the number of coin analysis above confirms that no matter which input array between V1, V2, and V3 the changeslow() and changedp() always produce the same amount of minimum number of coins for change. The changegreedy() algorithm results also confirm the anticipated larger number of minimum coins depending on the input array V and the input value of A.

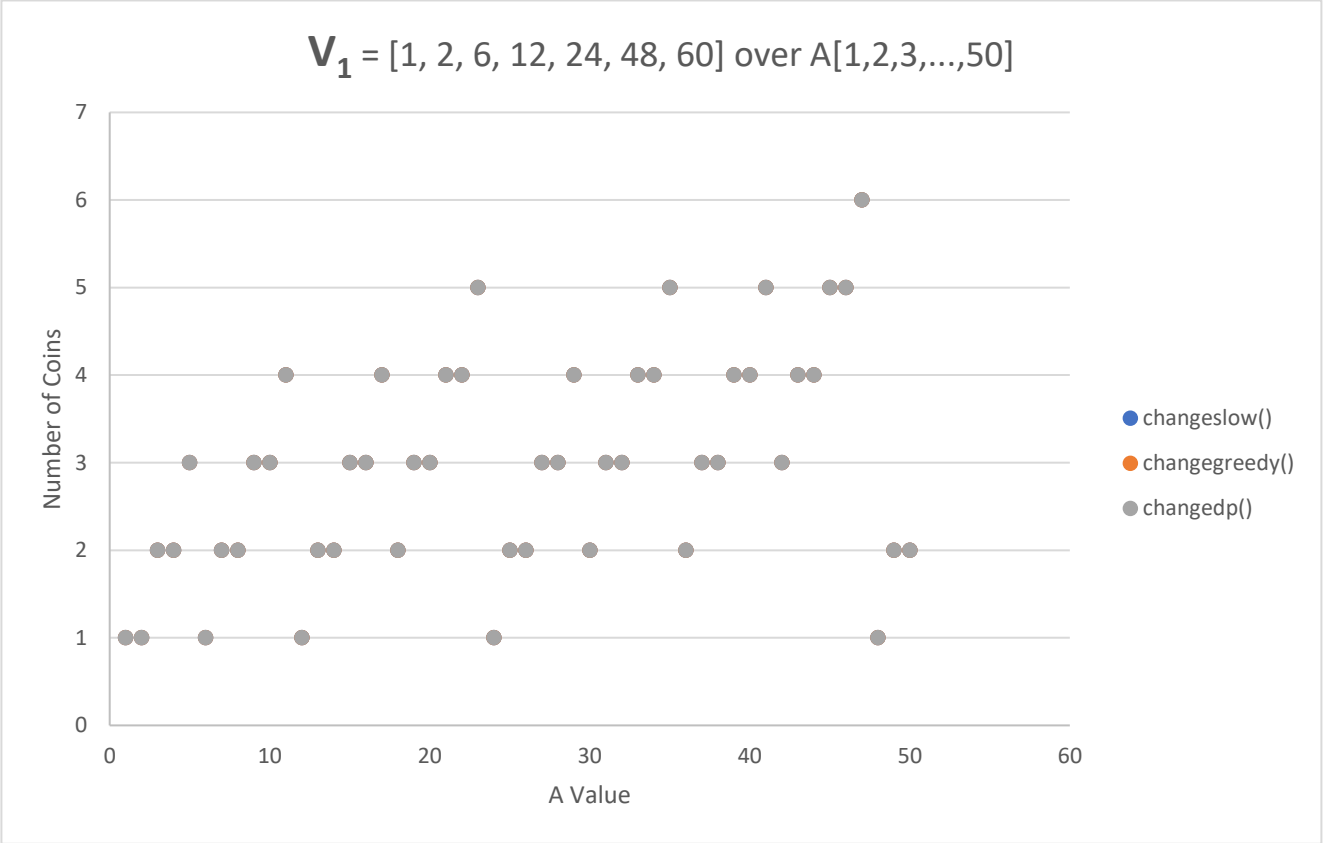
For V sets:

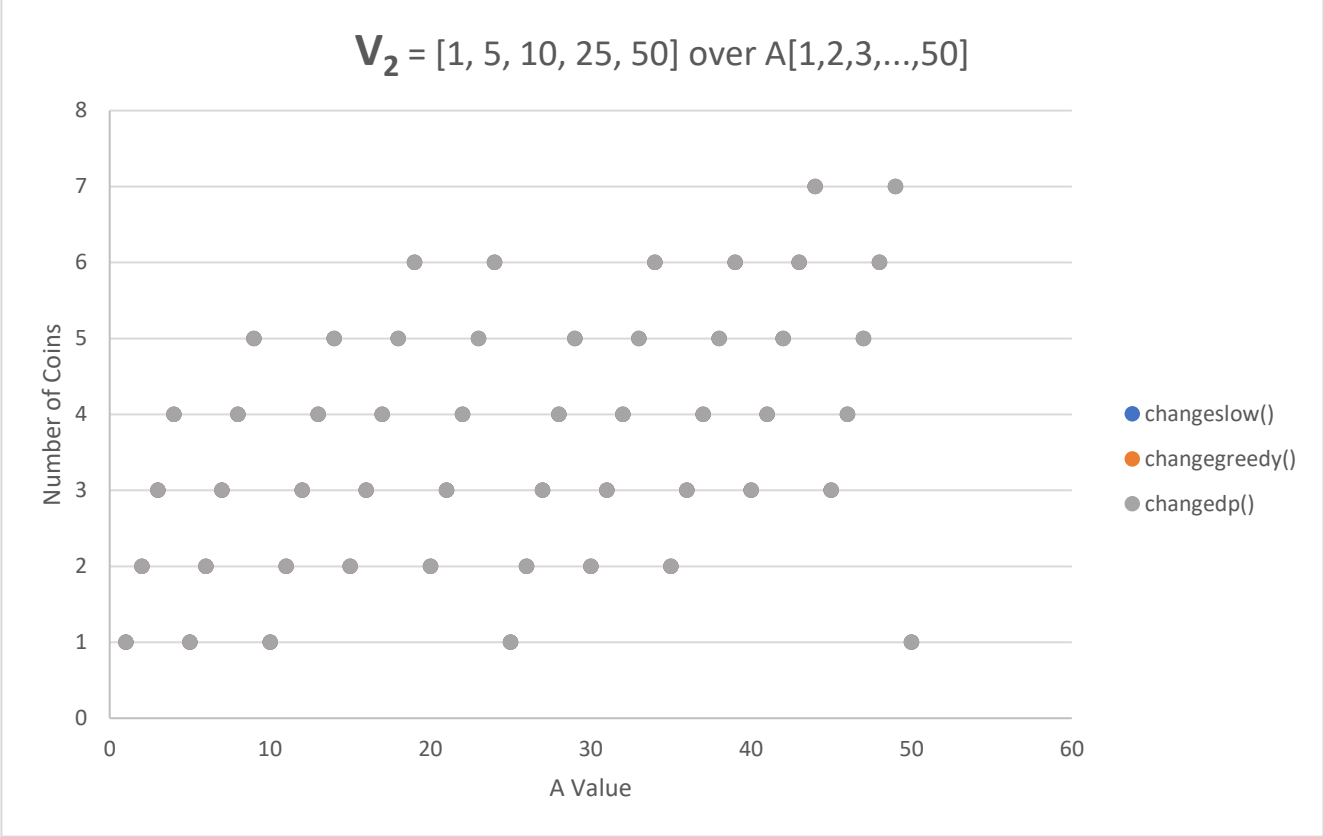
V_1 = [1, 2, 6, 12, 24, 48, 60]

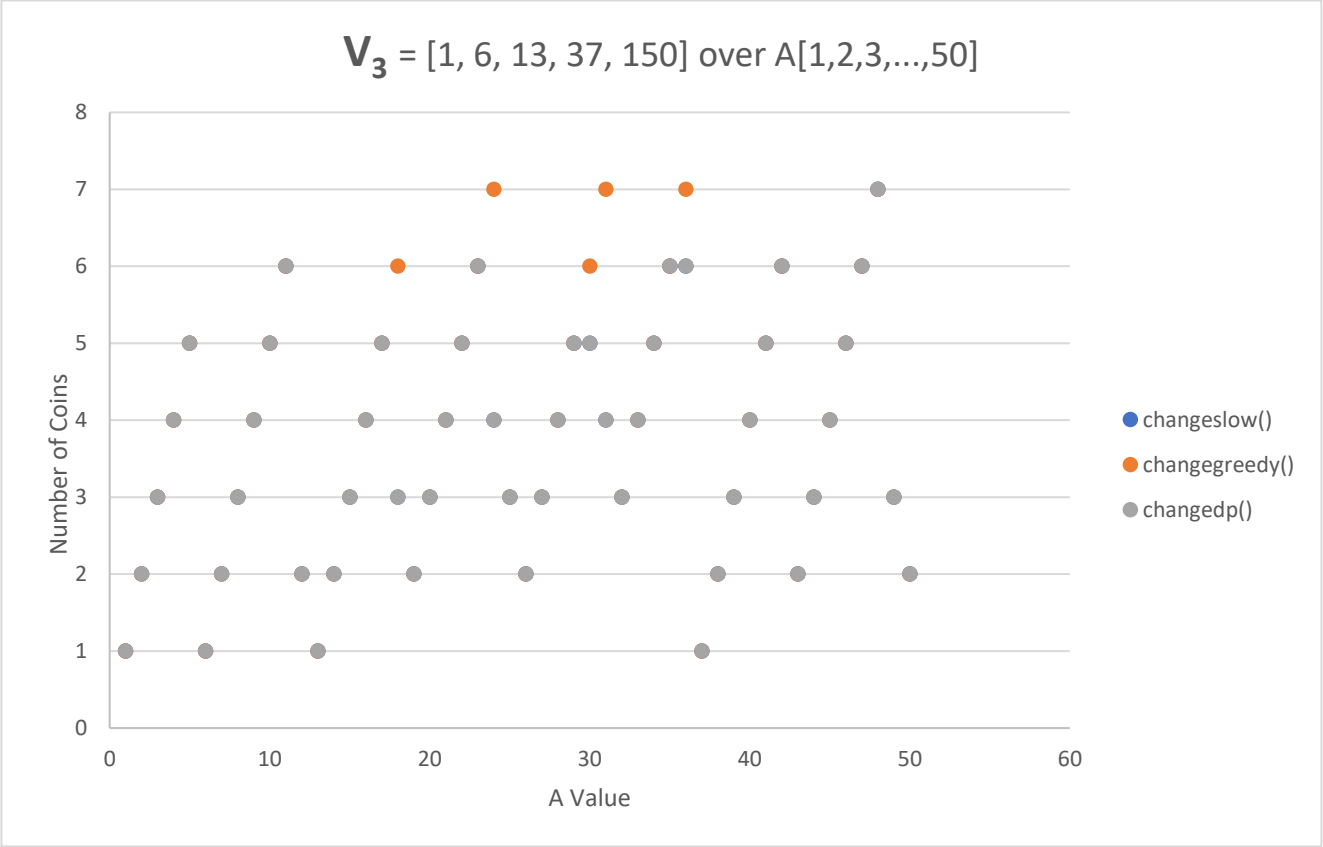
V_2 = [1, 5, 10, 25, 50]

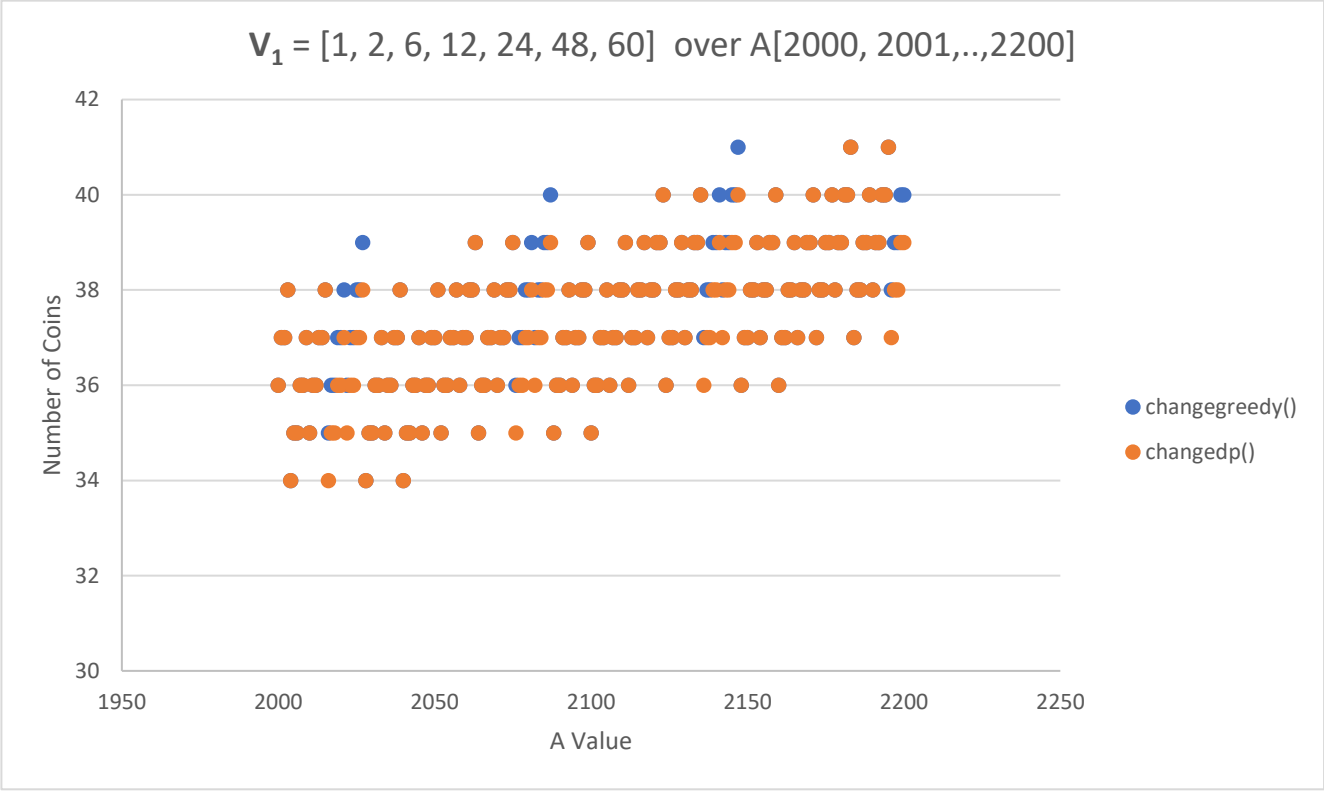
V_3 = [1, 6, 13, 37, 150]

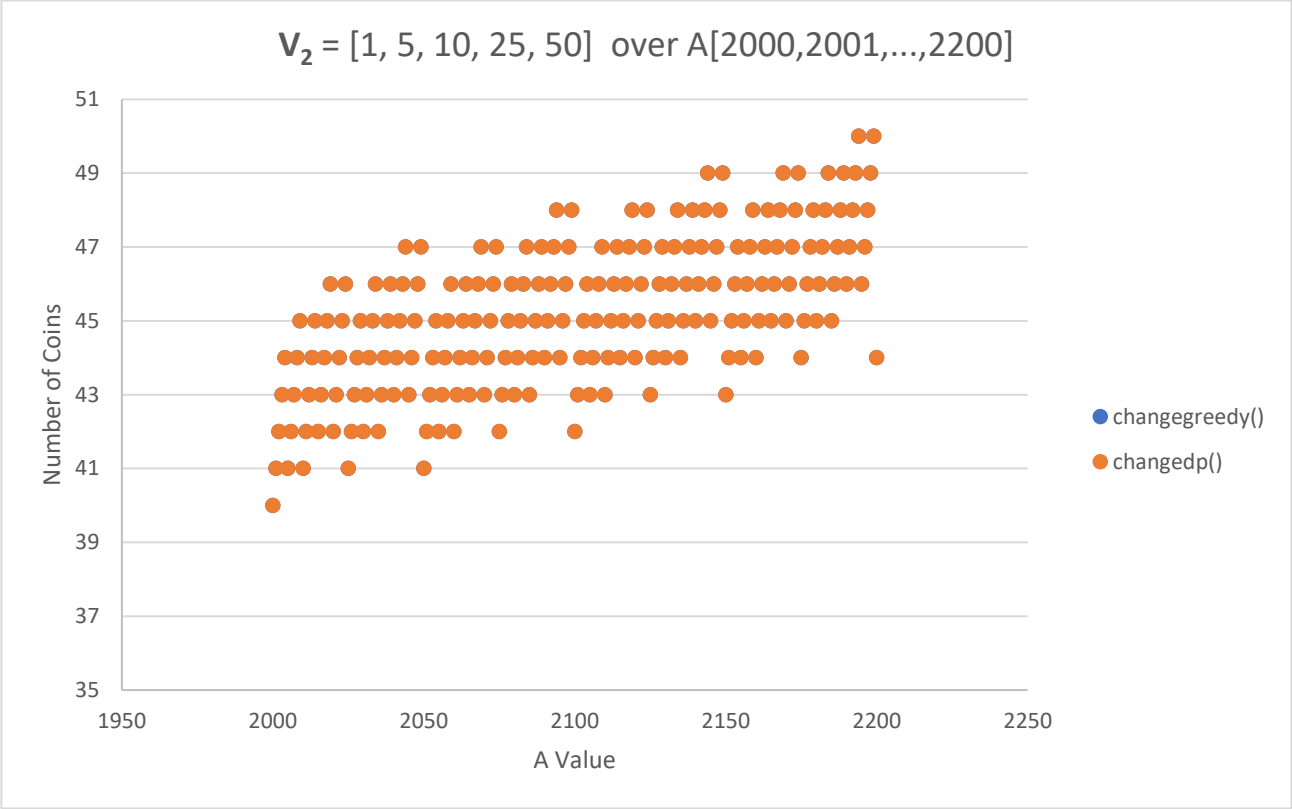
A in [1, 2, 3, …, 50]

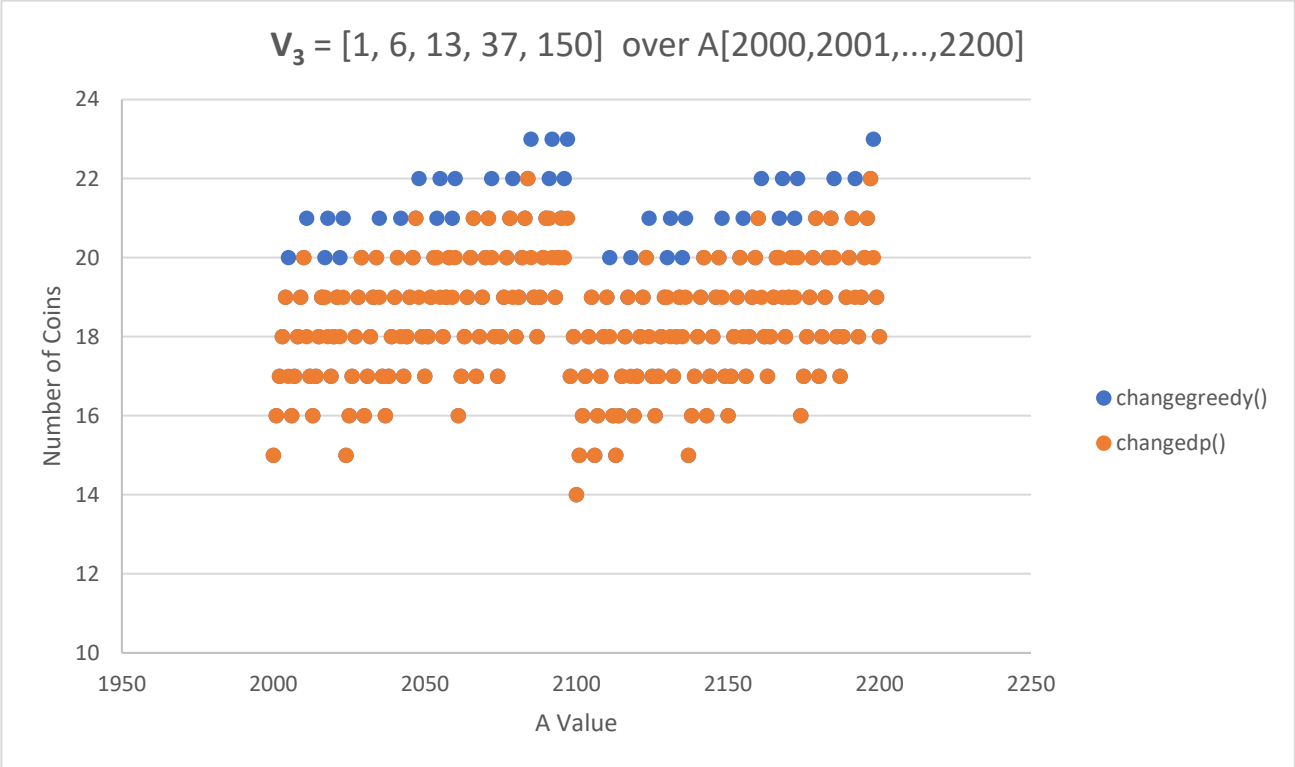

V_1 = [1, 2, 6, 12, 24, 48, 60] over A[1,2,3,...,50]
Number of Coins
A Value
changeslow()
changegreedy()
changedp()


V_2 = [1, 5, 10, 25, 50] over A[1,2,3,...,50]
Number of Coins
A Value
changeslow()
changegreedy()
changedp()


V_3 = [1, 6, 13, 37, 150] over A[1,2,3,...,50]
Number of Coins
A Value
changeslow()
changegreedy()
changedp()


V_1 = [1, 2, 6, 12, 24, 48, 60] over A[2000, 2001,..,2200]
Number of Coins
A Value
changegreedy()
changedp()





Resources used for this assignment:

- MATLAB
- EXCEL

- LECTURE