CS 325 Project 2: Coin Change

Nathan Stauffer, Daniel Olivas, Joanna Lew
Project Group 7

Spring 2017

Theoretical Run-time Analysis

Give the pseudocode and theoretical asymptotic running time for each algorithm.

(a) **Divide and Conquer Algorithm**

The idea for this algorithm was to approach the problem recursively. The base case is, if the amount is 0, then the number of coins required must also be 0. For all other amounts, subtract the value of a coin. If the coin value was larger than the amount, disregard the result and try a smaller coin value. Otherwise, if the coin value was smaller than the amount, recursively call the function on the new amount and add 1 to the result. Perform the subtraction and recursive call for all possible coin values, and choose the minimum number of coins required for the amount.

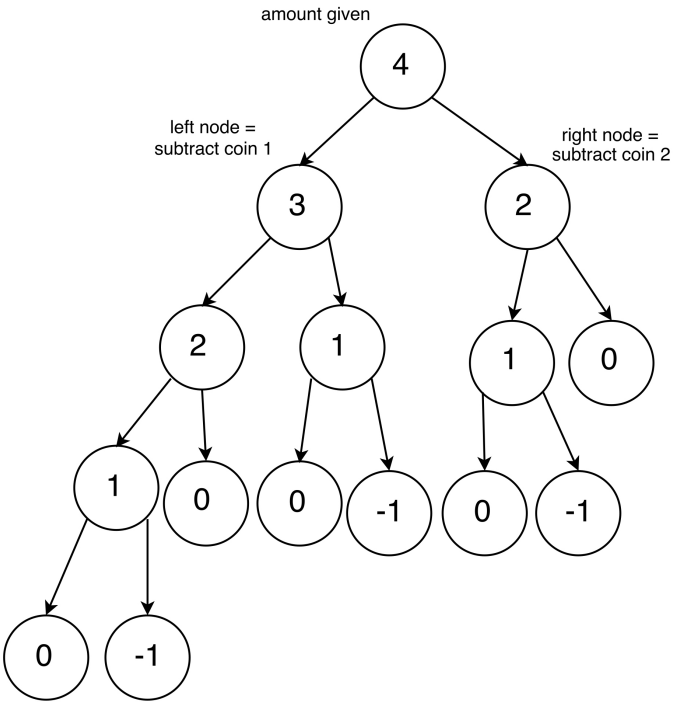
The psuedocode is provided as follows:

```
COIN_RECURSIVE(coin values[0 ... n - 1], amount A){
   coins vector holds sum of all coins for each amount
    subcoins vector holds table of subvectors of size values.size
    initialize all elements of subcoins to 0

    base case: if amount is 0...
      return subcoins

    helperFunction(values, amount, &coins, &subcoins)
    result vector filled with last values.size elements of subcoins
    return result
}

HELPERFUNCTION(coin values[0 ... n - 1], amount A, vector &coins, vector
    &subcoins){
   size = number of different coin values

    base case: if amount is 0...
      return 0
```

```
    min = amount + 1
    for each different coin value...
      get result of helperFunction(values, amount - (value of that coin),
          &both vectors)
        if result is less than min...
         result becomes new min
         for each different coin value...
         save coin count to corresponding subvector of subcoins

    save min to coins[amount - 1]
    return coins[amount - 1]
}
```

For the algorithm's base case (amount $= 0$), it simply returns a value. Therefore, $T(1) = \Theta(1)$.

For all other cases, the algorithm has exponential run time, similar to a recursive Fibonacci algorithm. This can best be illustrated with a tree diagram. For example, given two coins of value 1 and 2, and an amount 4, the recursive calls can be graphically shown as follows.



We can estimate the runtime of the algorithm by noting that the height of the tree is always $A + 1$, where A is the amount to be turned into change. The number of children nodes is dependent on the number of coin values. The larger the coin value, the faster the algorithm will reach 0 or a negative number, so the number of full levels is dependent on the coin value. However, since there is always a coin value of 1, the run time will always be at least $O(A)$.

In the worst case, the amount A will be very large, the number of coins values will be large, and the values will be sequential and small. The run time cannot be accurately approximated due to the dependency between amount, coin values, and number of coin values (as the number of full levels cannot be determined). However, we can see that it will be an exponential function since each new calculated amount will have n children nodes. Mathematically, the runtime must look something like $\Omega(n^{\frac{A}{v[max]}-1})$, where n is the number of coin values and $v[max]$ is the largest coin value.

(b) **Greedy Algorithm**

The idea for this algorithm was to use the greedy approach to the problem. Subtract the largest coin value possible from the amount. If the coin value is greater than the amount, go down one coin size and repeat. The psuedocode is as follows:

```
COIN_GREEDY(coinValues[0 ... n-1], amount){
   i = n - 1;            // largest coin value

   while (amount > 0){
      if (coinValues[i] < amount){
         amount = amount - coinValues[i]
            increment count of coinValues[i]
        }
        else{
         i--;          // go down a coin value
        }
    }
   return count of coinValues;
}
```

In the best case scenario, where the largest coin value is equal to the amount, the greedy algorithm takes constant time $O(1)$. The algorithm will subtract the coin value from the amount and return a count of one.

In the worst case scenario, where all coin values except the smallest are larger than the amount, and the smallest coin value is 1, the greedy algorithm runs at $O(n-1+A)+O(1)$, where n is the number of possible coin values and A is the amount. The algorithm will attempt to subtract all coin values from the amount, before subtracting 1 from the amount A times.

Assuming large A and n, the runtime would be $O(n + A)$.

(c) **Dynamic Programming Algorithm**

The dynamic programming approach to the coin problem involves keeping track of previously calculated solutions using two extra data structures - a list that keeps track of the maximum possible sum of coins that make up each change amount from 0 to the desired amount, and a table that essentially holds subvectors of correct coin counts for each change amount from 0 to the desired amount. For example, if our coin values were the standard U.S. $[1, 5, 10, 25]$ cents, and we were trying to find the minimum possible

coins to make 100 cents, we would use a list of size 101 and a table of 101 rows and 4 columns (one for each coin size). Each row would be a subvector keeping track of the coin combinations for each amount from 0 to the desired amount. Here is the pseudocode, using *sumlist* as the list of maximum sums and *table* as the table of subvectors:

```
COIN_DP(coin values[0....n-1], final amount A){
   max = A + 1
    size = size of coin values vector (number of coins)
    table = vector(max * size) (initialized all zeros)
    sum_list = vector(max) (all initialized to max except index 0 initialized
       to 0)

    for each amount "a" up to A...
      for each coin value "c" in values...
         if a coin of that size fits in "a"
               update highest sum for that "a" in sum_list - if applicable
                if highest sum needed updated, coin count needs updated, so for
                    as many coins as are in values...
                  from k = 0 to size - 1...
                     table[a * size + k] = table[(a - (coin size)) * size + k]
                    add one to the coin count for coin "c"

    final subvector of "size" elements is returned containing correct coin
       counts for amount A
}
```

There is certainly a run-time/memory trade-off for this algorithm. For large A, the two extra data structures required will be quite large. However, the run-time is significantly reduced by the fact that each previous calculation is remembered. In terms of specifically analyzing just the run-time aspect, the changedp algorithm requires at least two nested loops - one for each amount up to A, and nested inside that loop, another for each coin value. Therefore, the best case run-time is $O(An)$, where n is the number of different coin values.

In the worst case, the algorithm would not only need to execute the previously mentioned nested loops, but would also need to execute a third nested loop through each coin size, performing constant work in that loop. Therefore, the worst case run-time would be $O(An^2)$.

Explanation of changeDP table

As mentioned above, the table created to hold the minimum coin combinations for every change amount from 0 to the desired amount (A) can be thought of as a list of subvectors. For instance, if there are four possible coin values, each *row* or *subvector* would contain four elements. The first four would be the answer to the problem if A was 0, the second four if A was 1, and so on. Each time a larger A needs solved, the algorithm can reference the table for the subvector that

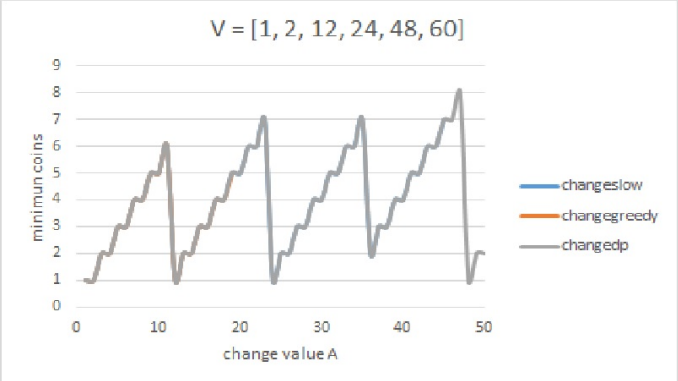
holds the previously solved solution for A - 1. This is how we go about updating the next row of the table for each lesser amount a from 0 to A. Essentially the next row of the table needs to know three things (in this order):

1. the solution found in the previous row of the table (for $a - 1$)
2. whether or not a multiple of one coin can be replaced by one of a larger coin
3. if applicable, what is the largest denomination that can fill the remaining shortage (and how many)

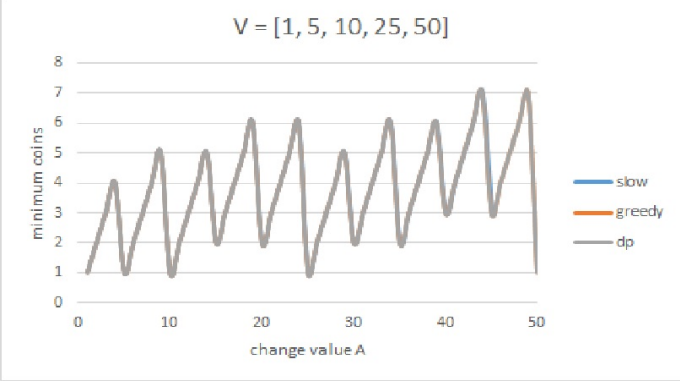
It is a complex way of saying that we continue to add smaller values of coins until we can replace two or more of them with a larger one. The algorithm set up so that when a certain current amount of smaller coins (sm) are replaced by a larger coin (lg), the amount of smaller coins can be reset to $sm - (valueof(lg))$. The reason this algorithm can be relied on to correctly fill the table is that it performs the necessary steps listed above in that order.

Minimum Coin Operations

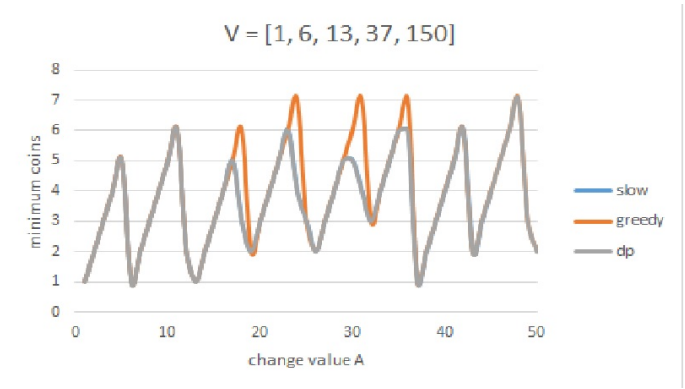
V1 = [1, 2, 6, 12, 24, 48, 60] for each integer value of A in [1, 2, 3, . . . , 50]



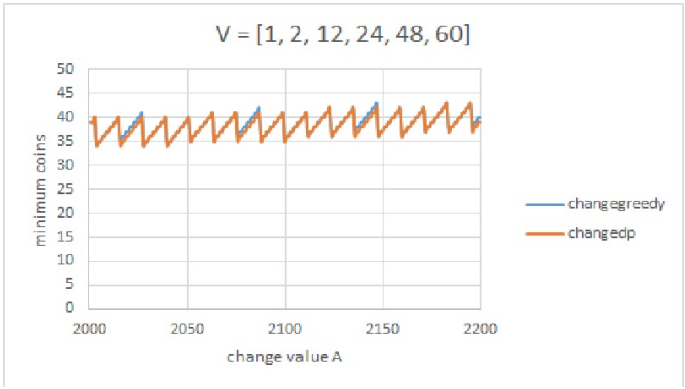
V2 = [1, 5, 10, 25, 50] for each integer value of A in [1, 2, 3, . . . , 50]



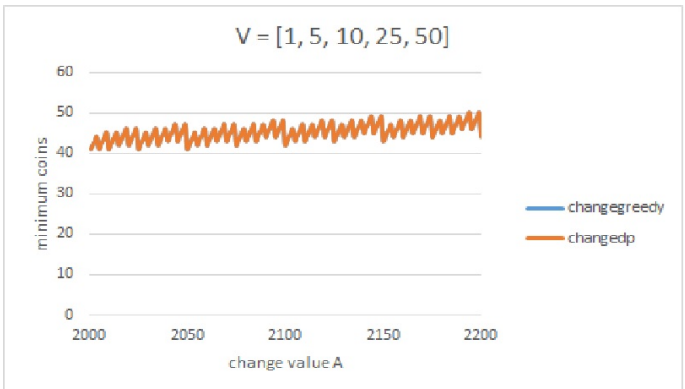
V3 = [1, 6, 13, 37, 150] for each integer value of A in [1, 2, 3, . . . , 50]



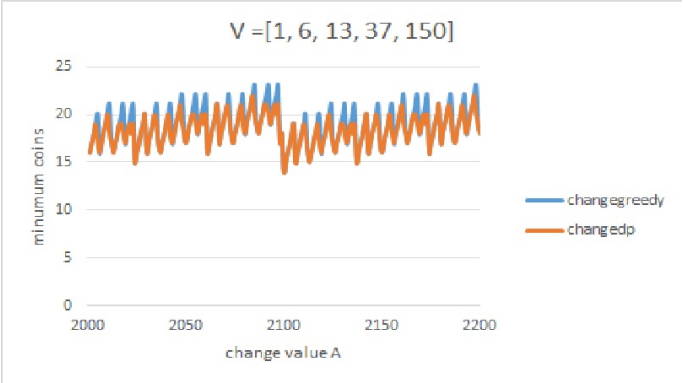
V1 = [1, 2, 6, 12, 24, 48, 60] for each integer value of A in [2000, 2001, 2002, . . . , 2200]



V2 = [1, 5, 10, 25, 50] for each integer value of A in [2000, 2001, 2002, . . . , 2200]



V3 = [1, 6, 13, 37, 150] for each integer value of A in [2000, 2001, 2002, . . . , 2200]



How do the approaches compare?

For each denomination set, the divide and conquer algorithm accurately select the minimum coins needed in for each integer values of A in [1, 2, 3, . . . , 45]. While the divide and conquer algorithm delivers to the optimal solution, it is not able to handle larger integer values like greedy algorithm and dynamic programming.

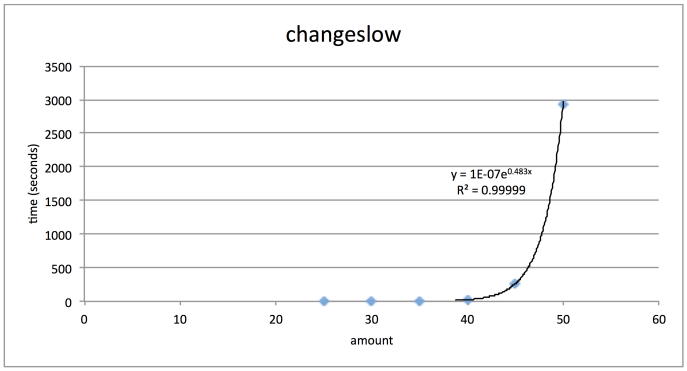
For each denomination set, the dynamic programming algorithm accurately returns the optimal solution for all the denominations for each integer value of A in [1, 2, 3, . . . , 50] and integer value of A in [2000, 2001, 2002, . . . , 2200].

Greedy algorithm returns the minimum coins for denominations V1 = [1, 2, 6, 12, 24, 48, 60] for each integer value of A in [1, 2, 3, . . . , 50] but not integer value of A in [2000, 2001, 2002, . . . , 2200]. Greedy algorithm correctly returns the minimum coins for denominations V1 = [1, 2, 6, 12, 24, 48, 60] for each integer value of A in [1, 2, 3, . . . , 50] and integer value of A in [2000, 2001, 2002, . . . , 2200]. Greedy algorithm does not return optimized solutions of V3 = [1, 6, 13, 37, 150] for either integer value of A in [1, 2, 3, . . . , 50] or integer value of A in [2000, 2001, 2002, . . . 2200].

Experimental Running Times

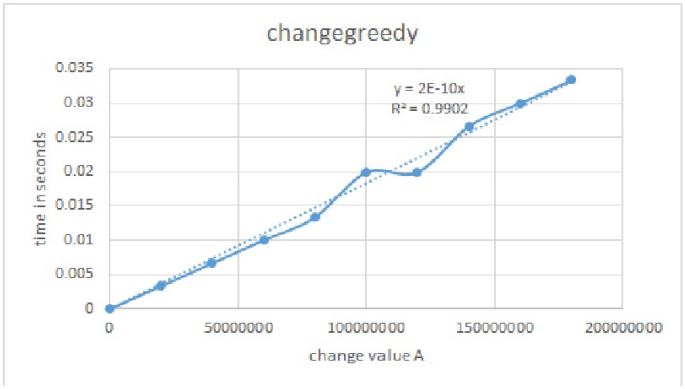
changeslow
We calculated the average running time for changeslow for 10 sizes between [5, 50] increasing 5 each time. Below is a plot of the time vs change values.
Using Excel's regression analysis, we determined the best-fit curve to be $T(A) = (1{\times}10^{-7})\, e^{0.483A}$.
The run time is therefore a exponential function.

This matches our theoretical run time and our expected result. As noted in our theoretical run time analysis, the algorithm runs much, much slower for problems with large amount and small coin values. This is due to the number of children nodes (representing the number of recursive calls) increasing exponentially, as every calculated amount must also be at least $O(A)$, where A is the new amount, due to the coin value 1.



changegreedy

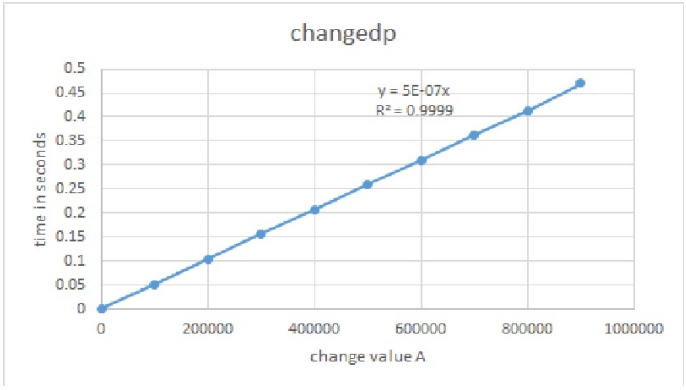
We calculated the average running time for changegreedy for 10 sizes between [0, 180000000] increasing 20000000 each time. Below is a plot of the time vs amount.



Using Excel's regression analysis, we determined the best-fit curve to be $T(A) = (2 \times 10^{-10})\, A$. The run time is therefore a linear equation.
The exponential asymptotic run-time is $O(A)$, which matches the theoretical runtime in the case when $n << A$.

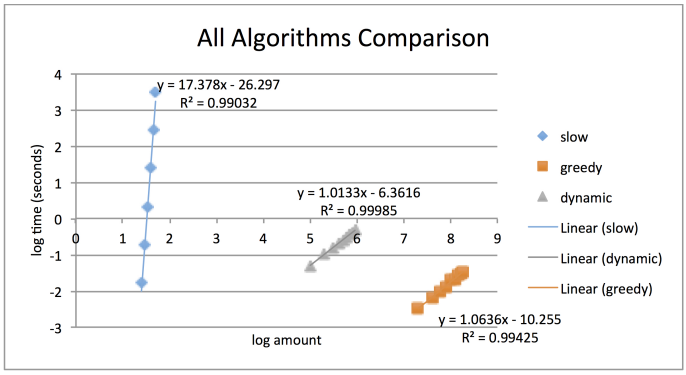
changedp

We calculated the average running time for changedp for 10 sizes between [0, 900000] increasing 100000 each time. Below is a plot of the time vs amount.



Using Excels regression analysis, we determined the best-fit curve to be $T(A) = (2 \times 10^{-7})\, A$. The run time is therefore a linear function, which matches our theoretical run time in the case when $n << A$.

Comparison of all algorithms We can compare the runtimes of all three algorithms on a log-log plot.



From the log-log plot, one can see that the dynamic algorithm and greedy algorithm both have linear runtime, as both slopes are approximately 1. As a result, the greedy approach should be used in cases where memory is preferable to accuracy, while the dynamic approach should be used when accuracy is required.

The run time of the recursive algorithm is not accurately shown, as log-log plots are used for polynomial functions and not exponentials. However, one can deduce that the run time of the recursive algorithm is extremely large, compared with the dynamic and greedy algorithms, due to its almost vertical slope.

Analysis for different coin values

Question to consider: 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.

Answer: For coin values of [1, 3, 9, 27] and [1, 5, 10, 25] the greedy approach would provide a similar performance. Since the approach involves simply adding as many coins as possible of the highest value, then as many of the next highest value, etc, in these two cases it does not make much of a difference because those coin values are very similar and are similarly spaced apart. If the *number* of coins increased, it would affect the algorithm much more. The dynamic programming approach would not be affected significantly either by this change in coin values. Since the algorithm requires an extra n iterations (n being the number of coins) each time a new coin's value can be added to the overall amount, those extra iterations will be necessary every time in these two examples because in both, there is a coin with a value of one. This means it will *always* be possible to add another coin to the amount, and therefore always necessary to run the extra loop. If the smallest value coin becomes a larger denomination, the extra loop

will not have to run for every amount and the dp algorithm could possibly become more efficient

Question to consider: 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?

Answer:

1. V = [1, 2, 3, 4]
   The greedy method will search first for the highest coin value and include as many as possible in the final amount. It will then move to the next highest coin value, and so on. In this example, the coin values are consecutive, which means there will be no "gaps" in the search for next best coin value. In other words, as soon as the maximum number of 4 cent coins is used, the only possible remaining amounts to be covered are 3, 2, and 1 - all of which are already available as coin values.

2. V = [1, 2, 10, 40, 80]
   The greedy method can be proven to be optimal for this value set by thinking of the scenario that would make the greedy method *not* optimal. The greedy method would not work if it can be shown that two or more of a smaller value gives a more optimal amount than using the highest value. For example, in the set [1, 4, 5], the optimal solution for eight cents is 2 four cent coins - a solution that does not involve using the highest value coin. The original set in question, however, would never produce this scenario since there is always a multiple of each smaller value coin that is the *exact same* as one coin of the next highest value.

3. V = [1, 3, 13, 50]
   This set can be proven optimal for use with a greedy algorithm by a similar reasoning used in example 2. Even though these coin values do not evenly go into one another as in that previous example, they still follow the principle that no value can be more ideally replaced by two of a smaller value. For example, there is no optimal way to 51 cents or greater that does not involve first using at least one 50 cent coin. The same can be said for each smaller coin value.