EE3980 Algorithm

HW10. Coin Set Design II

106061225 楊宇羲

1. Problem Description:

In this assignment, we are required to solve a common problem we may encounter in our daily life, which is combinations of each value of coins. There are several methods to accomplish the requirement, and we are going to use the dynamic programming to solve the problem. In the last assignment, we already solved the problem with greedy method, however using greedy method doesn't bring us the optimized solutions. In this assignment we are showing three kinds of dynamic program method to obtain the optimized solution, then analyze their complexities and comparisons among these three kinds of methods.

2. Approach:

To approach this algorithm, I am going to define some global variables *NCoin, p[4]* and *CoinNum[N]*. One of it is the coin set array that represents the total kinds of coin values, in the size of 4, and another one is the number of possible total value of money, and the last CoinNum array stores each value's minimum coin combination. The three methods are respectively DP recursive, DP top-down and DP bottom-up. Although they are all dynamic programming, but there are subtle difference which causes different time complexities. The main function's pseudo code and needed variables are provided below:

```
Global:
p[4] = {1, 5, 10, 50};          // initialize coin set
NCoin = 99;                     // define range from 1 to 99
CoinNum[NCoin];                 // store each value's minimum coin combination
```

```
In main:
Double Sum;                    // is used to store every execution of NCoinGreedy()
Double minCoin;                // is used to store the current minimum average
Double t;                      // for counting time
Algorithm main
{
        // original set
        for I from 1 to NCoin CoinNum[I] := DP_Recursive(I);
        write(CoinSol(NCoin));
        for I from 1 to NCoin CoinNum[I] := DP_TopDown(NCoin);
        write(CoinSol(NCoin));
        DP_BottomUp(NCoin);
        write(CoinSol(NCoin));

        // changing $50
        for p[3] := p[2] + 1 to NCoin do {
                for I from 1 to NCoin CoinNum[I] := DP_Recursive(I);
                if (avg < MinAvg) then {
                        minAvg = avg;
                        C4 := p[3];
                }
        }
        write(CoinSol(NCoin));
        for p[3] := p[2] + 1 to NCoin do {
                for I from 1 to NCoin CoinNum[I] := DP_TopDown(I);
                if (avg < MinAvg) then {
                        minAvg = avg;
                        C4 := p[3];
                }
        }
        write(CoinSol(NCoin));
        for p[3] := p[2] + 1 to NCoin do {
                DP_BottomUp(I);
                if (avg < MinAvg) then {
                        minAvg = avg;
                        C4 := p[3];
                }
        }
        write(CoinSol(NCoin));

        // changing $10
        p[3] := 50;
        for p[2] := p[1] + 1 to p[3] do {
                for I from 1 to NCoin CoinNum[I] := DP_Recursive(I);
                if (avg < MinAvg) then {
                        minAvg = avg;
                        C3 := p[2];
                }
```

```
}
write(CoinSol(NCoin));
for p[2] := p[1] + 1 to p[3] do {
        for I from 1 to NCoin CoinNum[I] := DP_TopDown(I);
        if (avg < MinAvg) then {
                minAvg = avg;
                C3 := p[3];
        }
}
write(CoinSol(NCoin));
for p[2] := p[1] + 1 to p[3] do {
        DP_BottomUp(I);
        if (avg < MinAvg) then {
                minAvg = avg;
                C3 := p[3];
        }
}
write(CoinSol(NCoin));

// changing $10 and $50
for p[2] := (p[1] + 1) to NCoin - 1 do {
        for p[3] := (p[2] + 1) to Ncoin do {
                for I from 1 to NCoin CoinNum[I] := DP_Recursive(i);
                if (avg < MinAvg) then {
                        minAvg := avg;
                        dd := p[2];
                        dd2 := p[3];
                }
        }
}
Write(CoinSol(NCoin));
for p[2] := (p[1] + 1) to NCoin - 1 do {
        for p[3] := (p[2] + 1) to Ncoin do {
                for I from 1 to NCoin CoinNum[I] := DP_TopDown(i);
                if (avg < MinAvg) then {
                        minAvg := avg;
                        dd := p[2];
                        dd2 := p[3];
                }
        }
}
Write(CoinSol(NCoin));
for p[2] := (p[1] + 1) to NCoin - 1 do {
        for p[3] := (p[2] + 1) to Ncoin do {
                DP_BottomUp(i);
                if (avg < MinAvg) then {
                        minAvg := avg;
                        dd := p[2];
```

```
                    dd2 := p[3];
                }
            }
        }
        Write(CoinSol(NCoin));
}
```

We are going to execute dynamic programming for 4 times each for every method, so we execute total for 12 times, respectively on the original set, changing p[3], changing p[2] and changing p[3] & p[2]. The aim of changing set value is to find the value that results in the minimum average of coin number. The original set requires only one time of DP_Recursive, DP_TopDown and DP_BottomUp, while second and third set requires NCoin times of those three methods, the forth set for $NCoin^2/2$ times of those three methods to find the coin value with minimum average, which takes most of the times.

3. Analysis:

In this part I am going to analyze the three different kinds of methods. For the first one DP_Recursive, we can get a general formula and write a function based of this formula:

Coin_R(n) = 1 + Coin_R(n - p[I]), where p[4] = {1, 5, 10, 50}

The NCoinGreedy() pseudo code looks like below:

```
Algorithm DP_Recursive(n)
{
        if n == 1 do return 1;                          // terminal condition
        else {
                min := n;                               // initialize each value to i 1 dollar coins
                for j := 4 to 1 step -1 do {            // check from the biggest value coin
                        if (n - p[j] >= 0) do {
                                check = DP_Recursive(n - p[j]);
                                if (1 + check < min) then min = 1 + check;
                        }
                }
        }
        return min;
```

}

Like in real life as we wish to take as few coins as possible to combine a value, so the recursive function will find the minimum coin number then return it. However, the recursive function can be improved so that we don't have to seek into every possible combinations by using CoinNum[NCoin] to store every value's minimum number.

The next method is the Top-Down method, which is introduced with the CoinNum array to store the minimum coin number of each value. Thus the dynamic program algorithm can be improved like this:

```
Algorithm DP_TopDown(n)
{
        CoinNum[0] := 1;                                    // initialize
        if CoinNum[n] != NCoin then return 1;               // correct answer
        if n >= 2 {
                min := n;                                   // initialize each value to i 1 dollar coins
                for j := 4 to 1 step -1 do {                // check from the biggest value coin
                        if (n - p[j] >= 0) do {
                                if (1 + DP_TopDown(n - p[j]) < min) then {
                                        min = 1 + DP_TopDown(n - p[j]);
                                }
                        }
                }
                CoinNum[n] = min;                           // store to CoinNum
        }
        return CoinNum[n];
}
```

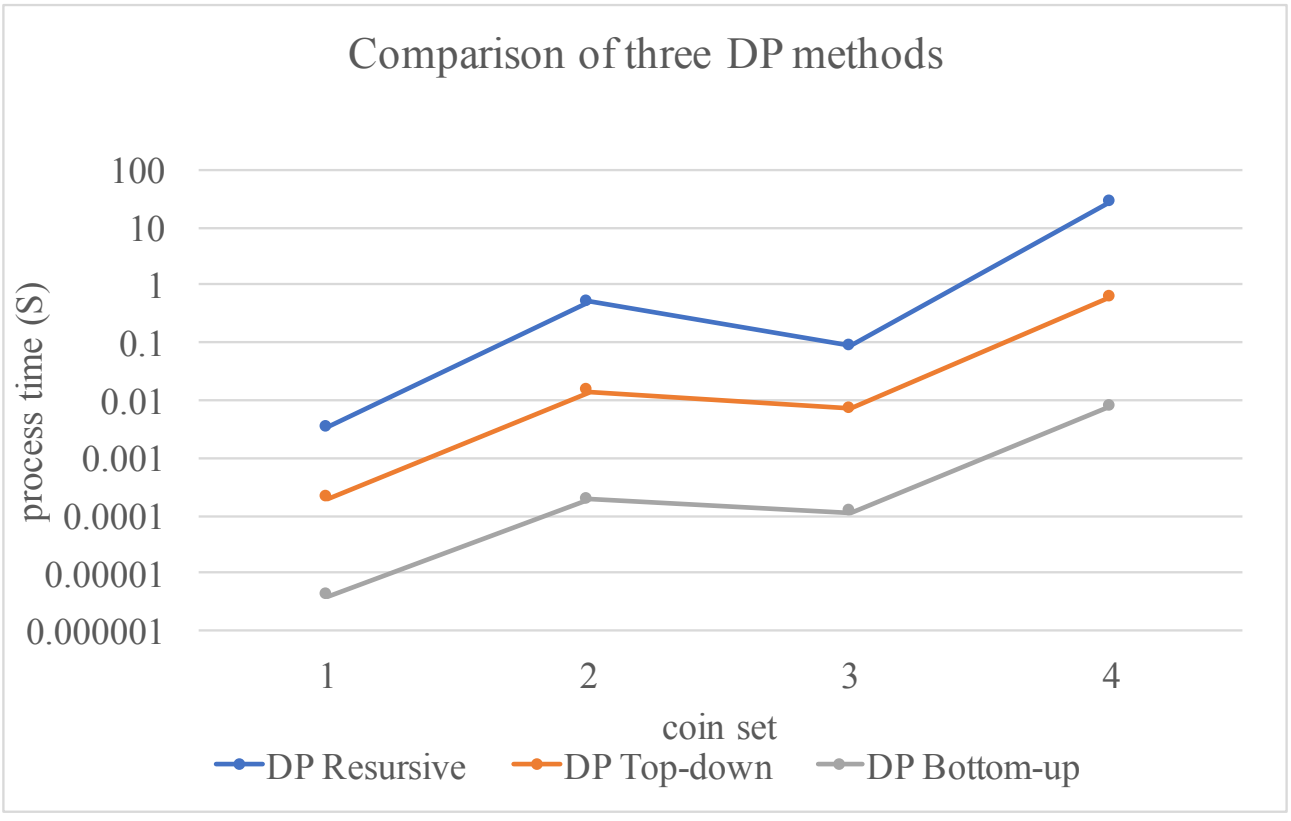
Note that when using this method we have to initialize the CoinNum array every time to set CoinNum[0] to 1 and NCoin, otherwise.

The last method is the Bottom-up method, which requires no recursive function, just a single loop to store all the value in the CoinNum array. Calling it once then it can finish arranging the value of the CoinNum array, and it is also faster than the recursive function.

```
Algorithm DP_TopDown(n)
{
        CoinNum[0] := 1;                                    // initialize
        for I from 1 to NCoin do {
                min := n;                                   // initialize each value to i 1 dollar coins
                for j := 4 to 1 step -1 do {                // check from the biggest value coin
                        if (I - p[j] >= 0) do {
                                if (1 + CoinNum[I - p[j]] < min) then {
                                        min = 1 + CoinNum[I - p[j]];
                                }
                        }
                }
                CoinNum[n] = min;
        }
}
```

4. Results:

Set number	Coin set	Minimum average
1	{1, 5, 10, 50}	$5.\overline{05}$
2	{1, 5, 10, 22}	$4.\overline{30}$
3	{1, 5, 12, 50}	$4.\overline{32}$
4	{1, 5, 18, 25}	$3.\overline{92}$



In this plot, the tendency grows as predicted, and the reason why set 2 spends more time than set 3 is because that set 2 ranges from 10 to 99, set 3 ranges from 50 to 99. Recursive function requires NCoin times of calling, with each time calling at most 10 times (see value 99's combination). Thus, every method has the time complexity of O(N^3) (due to finding forth set's minimum value). As for the time complexity, since recursive method doesn't't require CoinNum array, so it is O(N), while the others are O(N), **where N is size of NCoin**.

5. Observation and conclusion:

	Time Complexity	Space Complexity
DP Resursive	O(NCoin^3)	O(1)
DP Top Down	O(NCoin^3)	O(NCoin)
DP Bottom Up	O(NCoin^3)	O(NCoin)

Since dynamic programming breaks the big problem into lots of subproblems to solve, and what makes it different from greedy method is that it stores the previous steps' outcome, so that we can verify with different choices to make the optimized solution.