# 10802EE 398000 Algorithms

## Homework 10. Coin Set Design

- **Introduction**

In Taiwan, we have four types of coins: \$1, \$5, \$10, and \$50. Using these four types of coins, any dollar amount less than \$100 can be represented. Let $\{C_1, C_2, C_3, C_4\} = \{1, 5, 10, 50\}$, and the numbers of each type of coin be $\{x_1, x_2, x_3, x_4\}$, then the minimum number of coins for D dollars, $D \leq 99$, can be formulated as equation (1) to (3):

$$\text{minimize} \qquad Ncoin = \sum_{i=1}^{4} x_i \qquad (1)$$

$$\text{subject to} \qquad D = \sum_{i=1}^{4} x_i C_i \qquad (2)$$

$$\text{and} \qquad x_i \in Z \;\; and \;\; x_i \geq 0 \qquad (3)$$

Let $g_n(D)$ be the function that returns the minimum number of coins, using n types of coins, $1 \leq n \leq 4$, then one can derive the following recursive equation (4) to (5) of our minimum-coin problem, assuming $C_1 = 1$.

$$\boldsymbol{g_1(D) = D}, \qquad \mathbf{(4)}$$

$$\boldsymbol{g_n(D) = min\{x_n + g_{n-1}(D - x_n C_n)\} \quad n > 1 \;\; and \;\; \left\lfloor \frac{D}{C_n} \right\rfloor \geq x_n \geq 0} \qquad \mathbf{(5)}$$

And our goal is to find $g_4(D)$ since we have 4 types of coins.

In this homework, I will write a function to calculate $g_n(D)$ using **dynamic programming approach**, and using this function to answer following questions:

1. Given $\{C_1, C_2, C_3, C_4\} = \{1, 5, 10, 50\}$, find the average number of coins for D = 1 to 99.

2. Assuming $C_4$ is a variable, find its value that minimizes the average for D = 1 to 99.

3. Assuming $C_3$ is a variable, find its value that minimizes the average for D = 1 to 99.

4. Assuming both $C_3$ and $C_4$ are variables, find their values that minimizes the average for D = 1 to 99.

In **Approach** part, I will illustrate important steps to develop the dynamic programming algorithm of recursive equation (4) and (5). After that, I will explain how to use it to solve questions above. Also, I will show pseudocodes of them and analyze their time complexities and space complexities.

In **Result** part, I will show the answer of above four questions and show some CPU time results to see whether it is consistent to my analysis.

- ## Approach

**1. Develop dynamic programming algorithm of $g_n(D)$ function (equation (4) and (5))**

**The first step** is to derive the mathematical formula. Thankfully, it has already been given in the homework description and shown in **Introduction** part (equation (4) and (5)). Thus, I will just explain how it works.

Please recall that $g_n(D)$ is the function that returns the minimum number of coins for **D** dollars using **n** types of coins. Assume that our coin set is $\{C_1, C_2, C_3, C_4\} = \{1, 5, 10, 50\}$**.** That is, we have four types of coins: \$1, \$5, \$10, and \$50, and the numbers of each type of coin be ($x_1$, $x_2$, $x_3$, $x_4$). For n = 1, it returns D because we set $C_1 = 1$. We need exactly D coins using one dollar to pay D dollars. For n = 2, we have to check $\left\lfloor \frac{D}{C_2} \right\rfloor \geq x_2 \geq \mathbf{0}.$ That is, can we use $C_2$ to pay D dollars? (or whether D is larger or equal to $C_2$?) If it is possible, we try every possible results from $x_2 = 0, 1,$ to less than or equal to $\left\lfloor \frac{D}{C_2} \right\rfloor$ and choose the minimum one. If it is impossible, we just use the answer in n = 1 round. The statement for n = 2 can be generalized to n equal to any positive integers greater than one. Thus, we get the equation (5). Actually, the coin set can also be generalized if $C_1 = 1$.

**The second step** is to use this recursive formula directly to write the program. The pseudocode is shown below:

```
// Recursion version for equation (4) and (5)
// Input: n (n types of coins), D (for D dollars), C array (coin set).
// Output: the minimum number of coins for D dollars using n coins.
Algorithm coinR (n, D, C)
{
    if (n is 1) then
        return D;
    set min to the maximum value;
    for x := 0 to ⌊D/C_n⌋ do {
        if (min > x + coinR(n – 1, D – x * C[n], C)) then {
            min = x + coinR(n – 1, D – x * C[n], C);
        }
    }
    return min;
}
```

**Algorithm 1** coinR is the recursion version for equation (4) and (5).

However, we have to call from n = 4 to n = 1 for any D using coinR. This is quite redundant since we may have the result of $g_{n-1}(D - x_n C_n)$ already. Thus, our **third step** is to use a table to record the results. The pseudocode is shown below:

```
// Top-down version for equation (4) and (5) with the table
// Input: n (n types of coins), D (for D dollars), g array (g[n][D] is gn(D)), C array (coin set).
// Output: the minimum number of coins for D dollars using n coins, the updated g array
Algorithm coinTD(n, D, g, C)
{
    if (n is 1) then
        return D;
    set min to the maximum value;
    for x := 0 to ⌊D/Cn⌋ do {
        if (g[n – 1][D – x * C[n]] is -1) then {    // if g[n – 1][D – x * C[n]] has no value yet
            g[n – 1][D – x * C[n]] = coinTD(n – 1, D – x * C[n], g, C);
        }
        if (min > x + g[n – 1][D – x * C[n]]) then {
            min = x + g[n – 1][D – x * C[n]];
        }
    }
    g[n][D] = min;
    return min;
}
```

**Algorithm 2** coinTD is the top-down version for equation (4) and (5) with the table.

I use a **g array (whose elements are initialized to -1 except g[1 : n][0] are initialized to 0)** to store results of $g_n(D)$. In this way, we can avoid the repeated function call and make the time complexity better. However, there are lots of overlaps using recursion. Therefore, the **fourth step** is to design the bottom up version. The top down version calculates answers from n = 4 to n = 1. The bottom up version calculates answers from n = 1 to n = 4 instead.

The corresponding algorithm is shown in the next page:

```
// Bottom up version for equation (4) and (5) with the table
// Input: n (n types of coins), D (for D dollars), g array (g[n][D] is g_n(D)), C array (coin set).
// Output: the minimum number of coins for D dollars using n coins, the updated g array
Algorithm coinBU(n, D, g, C)
{
    if (n is 1) then
        return D;
    set min to the maximum value;
    for x := 0 to ⌊D / C_n⌋ do {
        if (min > x + g[n – 1][D – x * C[n]]) then {
            min = x + g[n – 1][D – x * C[n]];
        }
    }
    return min;
}
```

**Algorithm 3** coinBU is the bottom-up version for equation (4) and (5).

Before calling coinBU(n, D, g, C), we have to construct the table g[1 : n – 1][1 : D]. Thus, the coinBU_call is needed:

```
// need to call this function before calling coinBU
// Input: n (n types of coins), D (for D dollars), g array (g[n][D] is g_n(D)), C array (coin set).
// Output: the minimum number of coins for D dollars using n coins
Algorithm coinBU_call(n, D, g, C)
{
    for i := 1 to n - 1 do {
        g[i][0] := 0;
        for j := 1 to D do {
            g[i][j] = coinBU(i, j, g, C);
        }
    }
    return coinBU(n, D, g, C);
}
```

**Algorithm 4** coinBU_call needs to be called before calling coinBU.

There is an interesting questions: for calculating single $g_n(D)$, is using bottom-up method really faster than using top-down method?

To answer this question, I print out the g table (take **$g_4(10)$ and $\{C_1, C_2, C_3, C_4\} = \{1, 5, 10, 50\}$** as an example):

| | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | [10] |
|---|---|---|---|---|---|---|---|---|---|---|
| [1] | -1 | -1 | -1 | -1 | 5 | -1 | -1 | -1 | -1 | 10 |
| [2] | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | 2 |
| [3] | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | 1 |
| [4] | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | 1 |

**Figure 1** The result g table using coinTD.

| | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | [10] |
|---|---|---|---|---|---|---|---|---|---|---|
| [1] | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| [2] | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 5 | 2 |
| [3] | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 5 | 1 |
| [4] | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | 1 |

**Figure 2** The result g table using coinBU with calling coinBU_call.

From **Figure 1** and **Figure 2**, if we only want to calculate a **single $g_n(D)$ value**, using coinTD will be faster than using coinBU with calling coinBU_call because coinTD can avoid calculating some unnecessary values in g table. However, for question 1 to 4 in this homework, we have to find $g_4(D)$ from D = 1 to 99. The table will eventually be filled. Therefore, using the bottom up method is still a better way for this homework (I will prove this in **Result 1.** part).

What is the time complexity of coinBU_call? First, we analyze the time complexity of coinBU which is determined by the for-loop from line 6 to 10. We can get $O(\lfloor \frac{D}{C_n} \rfloor)$. It is between $\Omega(1)$ and $O(D)$. Therefore, the time complexity of coinBU_call will be between $\Omega(D)$ and $O(D^2)$. I will do an experiment and show results in **Results 2.** part.

## 2. How to use coinBU to solve question 1 to 4?

Finally, we can start to solve our four questions. I will directly call coinBU instead of calling coinBU_call from main function for the better flexibility. The pseudocode is shown below:

```
// main function to solve four questions (first part)
// Input: none.
// Output: answers to four questions and the CPU time.
Algorithm main()
{
    t = get the current CPU time;
    for n := 1 to 2 do {
        g[n][0] := 0;
        for D := 1 to 99 do {
            g[n][D] := coinBU(n, D, g, C);
        }
    }
    set min to the maximum value;
    g[3][0] := 0;
    g[4][0] := 0;
    for coin := 10 to 99 do {
        for coin2 := coin to 99 do {
            temp := 0;
            C[3] := coin;
            C[4] := coin2;
            for D := 1 to 99 do {
                g[3][D] := coinBU(3, D, g, C);
            }
            for D := 1 to 99 do {
                g[4][D] := coinBU(4, D, g, C);
                temp := temp + g[4][D];
            }
            ans[coin][coin2] := temp;
        }
    }
```

**Algorithm 5.1** main function to solve four questions (first part).

Since question 4 involves question 1 to 3, I use a table to store the total number of coins needed for all combinations of $C_3$ and $C_4$.The first part in main function is to construct this two-dimensional ans table. From line 4 to 9, I construct n = 1 and n = 2 of the g table first since $C_1$ and $C_2$ are fixed. From line 13 to 27, I construct the ans table. Please note that ans[$C_3$][$C_4$] indicates the minimum total number of coins when the coin set is {1, 5, $C_3$, $C_4$}.

```
// main function to solve four questions (second part)
// Input: none.
// Output: answers to four questions and the CPU time.
27    ans1 := ans[10][50] / 99.0;
28    set min, min2, min3 to the maximum value;
29    find the minimum value of ans[10][11 : 99], ans[11:49][50], and ans[i = 11 : 99][i : 99] and
30    record the corresponding coins: dollar2, dollar3, dollar41, and dollar42;
31    ans2 := min2 / 99.0;
32    ans3 := min3 / 99.0;
33    ans4 := min4 / 99.0;
34    t = get the current CPU time - t;
35    write(t, ans1, ans2, ans3, ans4, dollar2, dollar3, dollar41, dollar42);
36 }
```

**Algorithm 5.2** main function to solve four questions (second part).

After constructing the ans table, we can use it to find answers of question 1 to 4. The answer of question 1 is stored at ans[10][50]. The only thing we need to do is ans[10][50] / 99.0 which calculates the average. Line 29 to 30 is to find the minimum value according to the question. For question 2, the coin set is {1, 5, 10, $C_4$}, so we find the minimum value of ans[10][11 : 99]. For question 3, the coin set is {1, 5, $C_3$, 50}, so we find the minimum value of ans[11:49][50]. For question 4, the coin set is {1, 5, $C_3$, $C_4$}, so we find the minimum value of ans[i = 11 : 99][i : 99].

The time complexity of main function is determined by part 1 since part 2 can be done by using double for-loop. There are triple for-loops in part1. Besides, there is a coinBU in the inner-most loop. Thus, the time complexity is between $\Omega(D^3)$ and $O(D^4)$. D is 99 in our case. The space complexity is determined by the ans table. Thus, the space complexity is $O(D^2)$. Results of four questions are shown in **Results 3.** part.

| Algorithm | Overall Space Complexity | Time Complexity |
|---|---|---|
| coinBU | $O(D)$ | Between $\Omega(1)$ and $O(D)$ |
| coinBU_call | $O(D)$ | Between $\Omega(D)$ and $O(D^2)$ |
| main | $O(D^2)$ | Between $\Omega(D^3)$ and $O(D^4)$ |

**Table 1** space and time complexities.

## ● Results

1. The comparison between coinTD and coinBU to find $g_4(D)$ **from D = 1 to 99**.

The pseudocode of testing programs are shown below. Please note that $(C_1, C_2, C_3, C_4) = (1, 5, 10, 50)$ and loop is the static variable.

```
// Input: none
// Output: CPU time, the number of entering the loop in coinTD function
Algorithm testTD()
{
    initialize g array and C array and make loop := 0.
    t := get the current CPU time;
    for i := 1 to 99 do
        g[4][i] := coinTD(4, i, g, C);
    t := get the current CPU time - t;
    write(loop, t);
}
```

**Algorithm 6** testTD to test the CPU time of the top down method.

```
// Input: none
// Output: CPU time, the number of entering the loop in coinBU function
Algorithm testBU()
{
    initialize g array and C array and make loop := 0.
    t := get the current CPU time;
    for i := 1 to 4 do {
        g[i][0] := 0;
        for j := 1 to 99 do
            g[i][j] := coinBU(i, j, g, C);
    }
    t := get the current CPU time - t;
    write(loop, t);
}
```

**Algorithm 7** testBU to test the CPU time of the bottom up method.

The results are shown below:

```
Testing top-down method: CPU Time = 6.10352e-05 seconds
the number of entering the loop = 1747 times
```

**Figure 3** The result g table using testTD.

```
Testing bottom-up method: CPU Time = 4.19617e-05 seconds
the number of entering the loop = 1747 times
```

**Figure 4** The result g table using testBU.

From **Figure3 and Figure4**, the number of entering the loop is the same because both methods have to fill the whole g table. However, **the CPU time of the bottom-up method is less than the top-down method** in finding $g_4(D)$ **from D = 1 to 99**. This result is consistent to my analysis in **Approach** part: use the bottom up method is the better way in this homework.

2. What is the time complexity of coinBU_call? ($\{C_1, C_2, C_3, C_4\} = \{1, 5, 10, 50\}$ as an example)

The pseudocode of testing programs and its results is shown below.

```
// Input: none
// Output: CPU time
Algorithm testBU_2()
{
    initialize g array and C array.
    loop := 0;
    give a D value;
    t := get the current CPU time;
    g[4][D] := coinBU_call(4, D, g, C);
    t := get the current CPU time - t;
    write(t);
}
```

**Algorithm 8** testBU_2 to test the time complexity of coinBU_call.

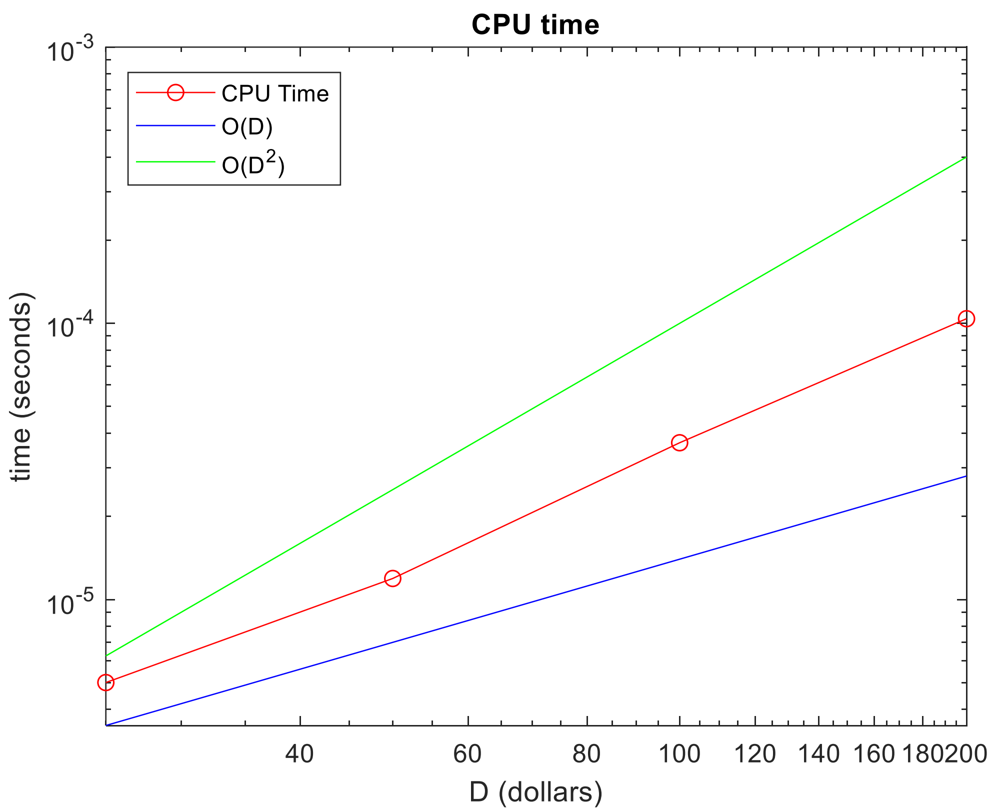


**Figure 5** CPU time of coinBU_call.

From executing testBU_2 with D = 25, 50, 100, 200 respectively, we can get the CPU time graph shown above. It seems that the time complexity is really between $\Omega(D)$ and $O(D^2)$.

3. Answers of four questions:

```
CPU time = 3.874183e-02 seconds
For coin set {1, 5, 10, 50} the average is 5.05051
Coin set {1, 5, 10, 22} has the minimum average of 4.30303
Coin set {1, 5, 12, 50} has the minimum average of 4.32323
Coin set {1, 5, 18, 25} has the minimum average of 3.92929
```

**Figure 6** Answers of four questions.