EE 3980 Algorithms

Homework 10. Coin Set Design

105061110 周柏宇

2020/5/22

1. Introduction

In this homework, we want to dynamic programming to obtain the average of the minimum number of coins required to represent any dollar amount less than $100 with a coin set with 4 types of coins. The coin set for problem 1 is given, while for problem 2, 3 and 4, the coin set is partially undetermined. Thus, we need to decide what type of coins achieve the minimum average.

2. Analysis & Implementation

2.1 Problem Formulation

We are given the coin set with 4 types of coins: (C_1, C_2, C_3, C_4), and we want to obtain the minimum of the total number of coins used to represent the dollar amount D. The decision variables are (x_1, x_2, x_3, x_4), where x_i represents the number of C_i used. The optimization problem can be formulated as follows:

$$minimize\ Ncoin = \sum_{i=1}^{4} x_i$$

$$subject\ to \sum_{i=1}^{4} x_i C_i = D$$

$$and\ x_i \geq 0\ and\ x_i \in \mathbb{Z}$$

In this homework, there are 4 problems. All of the problems require us to calculate the average minimum number of coins used for representing dollar amount *D* ranging from 1 to 99, which is denoted as $\overline{Ncoin}$.

Problem 1: The coins set is $(C_1, C_2, C_3, C_4) = (1, 5, 10, 50)$, find $\overline{Ncoin}$.

Problem 2: Assuming C_4 is a variable, find C_4 that minimizes $\overline{Ncoin}$.

Problem 3: Assuming C_3 is a variable, find C_3 that minimizes $\overline{Ncoin}$.

Problem 4: Assuming C_3, C_4 are variables, find C_3, C_4 that minimizes $\overline{Ncoin}$.

For the problems described above, we will use dynamic programming to solve it. First, we formulate the solution. Let $g_n(D)$ be the function that returns the minimum number of coins, using first *n* types of coins, for $1 \leq n \leq 4$. We can derive the solution recursively:

$$g_1(D) = D$$

$$g_n(D) = min\{x_n + g_{n-1}(D - x_n \cdot C_n)\},$$

$$for\ x_n = 0, 1, \ldots, \lfloor D\ /\ C_n \rfloor\ and\ n\ > 1$$

We can save the result of $g_n(D)$ in an array $g[n][D]$, and the array is initialized as follows:

$$g[1][D] = D \; and \; g[n][0] = 0, n = 1,2,3,4$$

$$Otherwise, g[n][D] = -1$$

2.2 Obtain $g[n][D]$ Using Top-down Dynamic Programming

```
1. // Find the minimum number of coins to represent D,
2. // using coin set C with n types of coins.
3. // A top-down dynamic programming approach.
4. // Input: coin set C with n types of coins, D dollars
5. // Output: g[n][D]
6. Algorithm g_TD(C, n, D)
7. {
8.     if (g[n][D] >= 0) return g[n][D];
9.     min := g_TD(C, n - 1, D);
10.      for i := 1 to ⌊D / C[n]⌋ do {
11.          tmp := i + g_TD(C, n - 1, D - i * C[n]);
12.          if (tmp < min) min := tmp;
13.      }
14.      g[n][D] := min;
15.      return g[n][D];
16.  }
```

2.3 Obtain $g[n][D]$ Using Bottom-up Dynamic Programming

```
1. // Find the minimum number of coins to represent D,
2. // using coin set C with n types of coins.
3. // A bottom-up dynamic programming approach.
4. // Input: coin set C with n types of coins, D dollars
5. // Output: g[n][D]
6. Algorithm g_BU(C, n, D)
7. {
8.     for i := 2 to n do {
9.         for j := 1 to D do {
10.             min := g[i - 1][j];
11.             for k := 1 to ⌊j / C[i]⌋ do {
12.                 tmp := k + g[i - 1][j - k * C[i]];
13.                 if (tmp < min) min := tmp;
14.             }
15.             g[i][j] := min;
16.         }
17.     }
18.     return g[n][D];
19. }
```

For the g_BU algorithm, there are 3 for loops. The first two outer loops execute $n-1$ and $D \cdot (n-1)$ times respectively. The innermost loop executes $\mathcal{O}(n \cdot D^2)$. Hence, the time complexity for g_BU is $\mathcal{O}(n \cdot D^2)$. As for the space complexity, there are $\Theta(1)$ local variables and requires $\Theta(n \cdot D)$ space to store the array g. Thus, the overall space complexity is $\Theta(n \cdot D)$.

For the g_TD algorithm, the time complexity and space complexity are asymptotically the same as those of the g_BU algorithm.

2.4 Obtain $g[n][D]$ with Solutions

```
1. // Find the minimum number of coins to represent D,
2. // using coin set C with n types of coins.
3. // A bottom-up dynamic programming approach.
4. // Input: coin set C with n types of coins, D dollars
5. // Output: g[n][D], array sol
6. Algorithm g_sol(C, n, D)
7. {
8.     for i := 2 to n do {
9.         for j := 1 to D do {
10.              min := g[i - 1][j];
11.              x[i][j] := 0;
12.              for k := 1 to ⌊j / C[i]⌋ do {
13.                  tmp := k + g[i - 1][j - k * C[i]];
14.                  if (tmp < min) {
15.                      min := tmp;
16.                      x[i][j] := k; // # coins to take
17.                  }
18.              }
19.              g[i][j] := min;
20.          }
21.      }
22.      // retrieve the solution
23.      tmp := D;
24.      for i := n to 1 step -1 do {
25.          sol[i] := x[i][tmp];
26.          tmp := tmp - sol[i]*C[i];
27.      }
28.      return g[n][D];
29.  }
```

Since the time complexity for `g_sol` is dominate by the dynamic programming part rather than retrieving the solution. Therefore, it also is $\mathcal{O}(n \cdot D^2)$. However, we need an extra $\Theta(n \cdot D)$ to store the array `x` and $\Theta(n)$ to store the array `sol`. It has an overall $\Theta(n \cdot D)$ as well.

2.5 Algorithm Customized for the Homework

In the class, we are advised to use the bottom-up version of the dynamic programming rather than the top-down one. Also, for this specific homework, some modifications can be made to improve efficiency. Recall that the problems are as follows:

Problem 1: The coins set is $(C_1, C_2, C_3, C_4) = (1, 5, 10, 50)$, find $\overline{Ncoin}$.

Problem 2: Assuming C_4 is a variable, find C_4 that minimizes $\overline{Ncoin}$.

Problem 3: Assuming C_3 is a variable, find C_3 that minimizes $\overline{Ncoin}$.

Problem 4: Assuming C_3, C_4 are variables, find C_3, C_4 that minimizes $\overline{Ncoin}$.

We first solve problem 1, and the elements in array g are evaluated. To solve problem 2, we don't need to recalculate most of the value, to be specific, we only need to recalculate the row with n=4. For the same reason, we don't need to recalculate the row with n=1 and 2 for problem 3 and 4. With this argument, we can modify the `g_BU` algorithm.

```
// Find the minimum number of coins to represent D,
// using coin set C with n types of coins.
// A bottom-up dynamic programming approach.
// Input: coin set C with n types of coins, D dollars
//        skip first s rows calculation for g
// Output: g[n][D]
Algorithm g_BU(C, n, D, s)
{
    for i := s + 1 to n do {
        for j := 1 to D do {
            min := g[i - 1][j];
            for k := 1 to ⌊j / C[i]⌋ do {
                tmp := k + g[i - 1][j - k * C[i]];
                if (tmp < min) min := tmp;
            }
            g[i][j] := min;
        }
    }
    return g[n][D];
}
```

It has same time complexity as the original one, but we except it to be more efficient. If we solve the problems sequentially, `g_BU` should be called using:

Problem 1: `g_BU(C, 4, D, 1)`

Problem 2: `g_BU(C, 4, D, 3)`.

Problem 3: `g_BU(C, 4, D, 2)`

Problem 4: `g_BU(C, 4, D, 2)`

3. Result and Observation

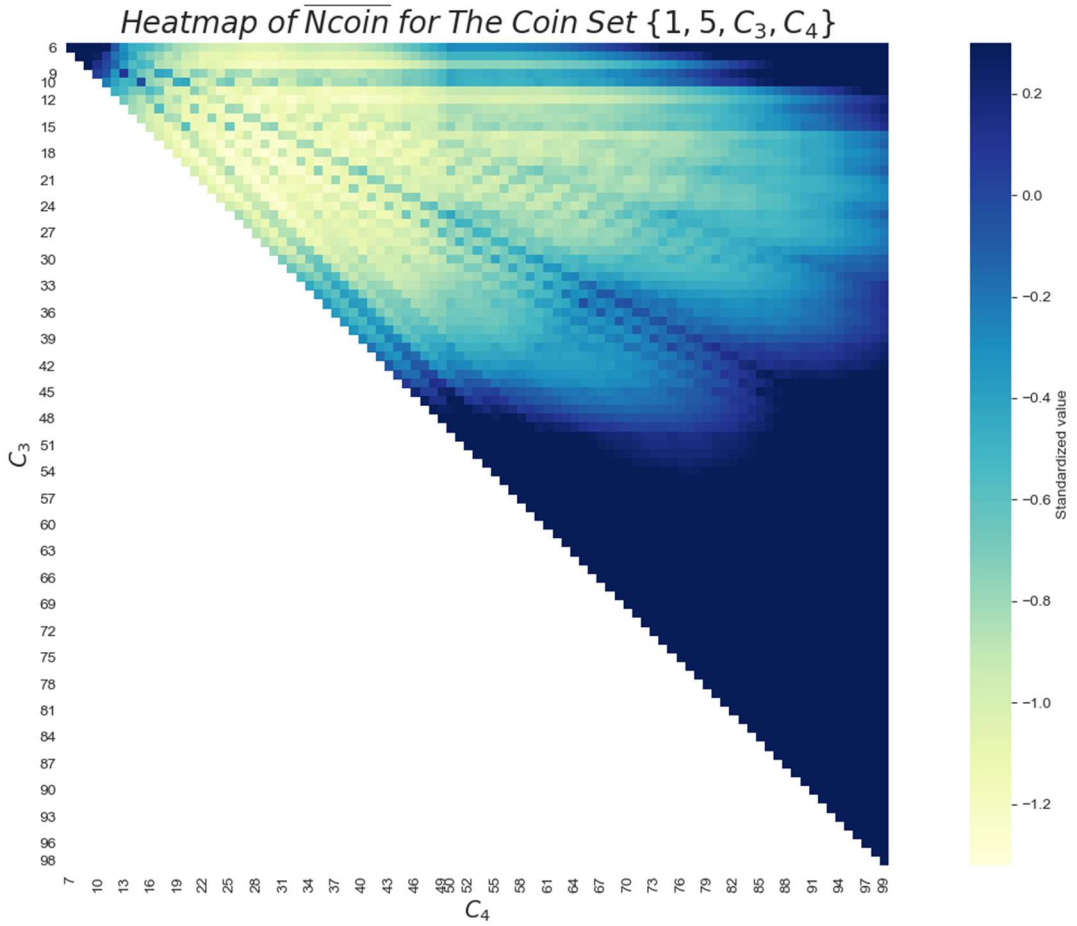
Here is the result we get.

```
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
```

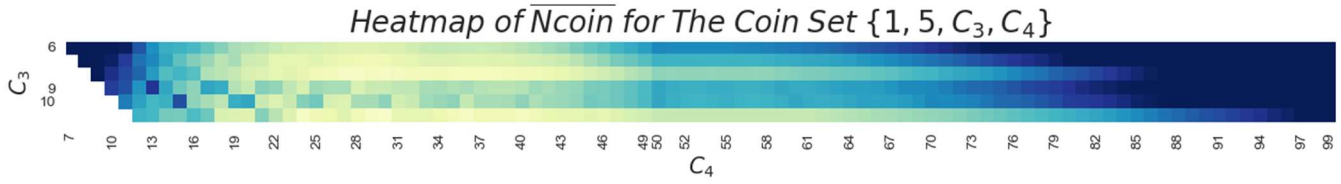
Using `g_sol`, we can print out the solutions.

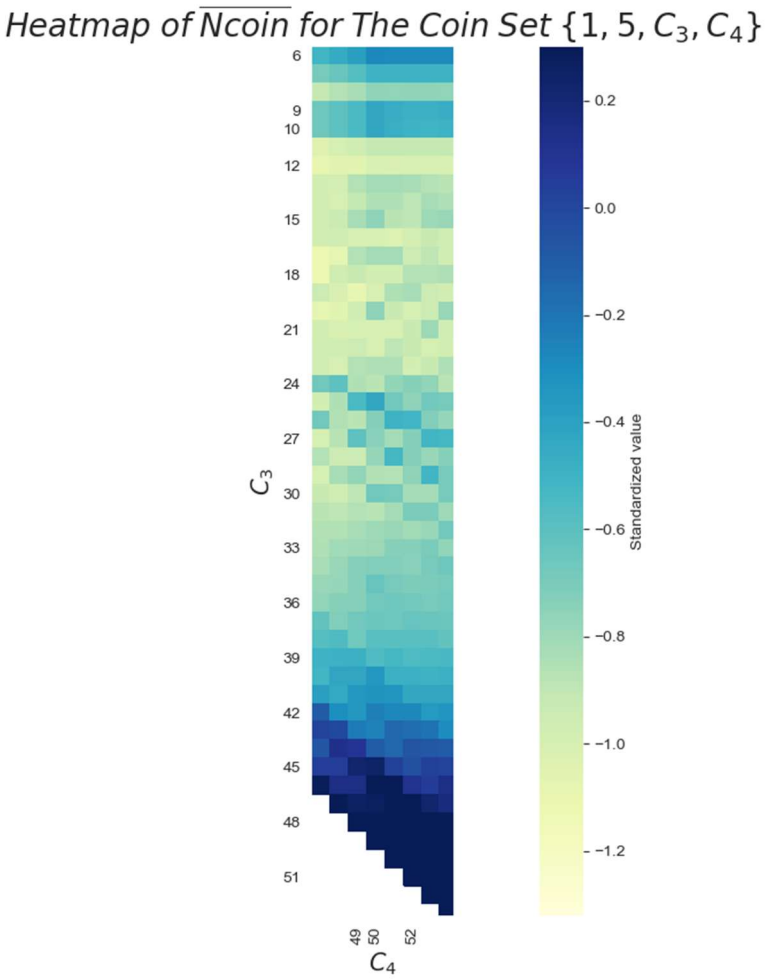
```
 D:  1  5 10 50 Ncoin
---------------------
 1:  1  0  0  0   1
 2:  2  0  0  0   2
 3:  3  0  0  0   3
 4:  4  0  0  0   4
 5:  0  1  0  0   1
          ≈
95:  0  1  4  1   6
96:  1  1  4  1   7
97:  2  1  4  1   8
98:  3  1  4  1   9
99:  4  1  4  1  10
---------------------
Total:            500
Average:      5.05051
```

We also can look at the $\overline{Ncoin}$ that results from all the attempts from problem 4.



We can see that the solution $(C_3, C_4) = (18,\ 25)$ is indeed in the area with smaller $\overline{Ncoin}$. Also, the solutions for problem 2 and 3, $(C_3, C_4) = (10,\ 22)$ and $(C_3, C_4) = (12,\ 50)$, can be somewhat observed as well.





One more observation, although using top-down dynamic programming may have some overhead because of recursive calls, it only evaluates the values that it needs. On the other hand, the bottom-up dynamic programming fills the table from n = 1 to 4 in order to get rid of recursive calls. However, since when filling the column from the lower row (or lower n), we don't know what columns (i.e. amount of dollars) may be used in the evaluation of higher rows (higher n). As a result, we need to evaluate the column from 1 to D, which can be redundant.

Here is the execution time of solving the 4 problems.

Using `g_BU`	Using `g_TD`
2.23175e+00 seconds	8.53939e-02 seconds

The difference is quite noticeable.