EE3980 Algorithms

Hw10 Coin Set Design

106061146 陳兆廷

**Introduction:**

In this homework, I will be analyzing, implementing, and observing 1 algorithm. The goal of the algorithm is to get optimal coin sets using the pre-defined coin values, and the output of them will be the average coins needed to get N dollars.

During the analysis process, I will first introduce why and how dynamic programming can be applied to this task. Then, I will be using counting method to calculate the time complexities of the algorithm. Finally, I will calculate their space complexity for the total spaces used by the algorithm.

The implementation of the algorithm on C code will find the average amount of coins needed for 1 ~ 99 dollars. Furthermore, I will find the optimal values of the 3rd and 4th coin value to get the minimum average coins needed.

**Analysis:**

1. **Dynamic programming:**

Dynamic programming isto simplifying a complicated problem by **breaking it down into simpler sub-problems** in a [recursive](https://en.wikipedia.org/wiki/Recursion) manner. In this case, since there are several ways to add up the same amount of money, that is to say, we can break any amount of money into simpler sub-set of coins in a recursive manner. For example, for the number 99, we can break it down to 50 and 49. And for 49, we can break it down to 10 and 39 … etc. Therefore, dynamic programming is a perfect way to implement this task.

1. **Variables set:**

|  |  |
| --- | --- |
| N | target dollars to get minimum |
| p[3] | Coin values that can be used. In this case, {1, 5, 10, 50}. |
| Coin[n] | Minimum amount of coins needed |
| s[n] | Solution for the combination of coins |

1. **Recursive function:**

Using recursive function, we can formulate the solution as:

Same as getting the change in real life, we tend to find the maximum change we can get, deduct it and find the next maximum … etc.

Therefore, we can develop a recursive algorithm:

1. // Find min # of coins to get n dollars. p[1 : 4] is the price table.
2. // Input: int n, coin table p[1 : n]
3. // Output: min # of coins.
4. Algorithm coin\_R(p, n)
5. {
6. **if** (n = 1) **return** 1 ;
7. min := INF ; // no cut.
8. **for** i := 1 to 4 **do** { // check all combination using recursion.
9. **if** (1 + rod\_R(p, n − p[i] ) < min) then {
10. min := 1 + rod\_R(p, n − p[i]);
11. }
12. }
13. **return** min ;
14. }

Example of coin\_R(p, n) unrolling:

|  |  |  |  |
| --- | --- | --- | --- |
| coin\_R(p, 7) | 1 + coin\_R(p, 2) | 2 + coin\_R(p, 1) | 3 |
| coin\_R(p, 6) | 1 + coin\_R(p, 1) | 2 |  |
| coin\_R(p, 5) | 1 |  |  |
| coin\_R(p, 4) | 1 + coin\_R(p, 3) | 2 + coin\_R(p, 2) | … |

The efficiency of the recursive coin set algorithm can be improved significantly using an array to store Coin counts before. Coin[0 : n].

1. **Top-down dynamic programming:**

Instead of calculate the coin-counts every time, we store them in Coin-counts array, Coin[0: n]. And modified the recursive function:

Before calling the coin\_TD(p, n, coin) function, the coin array should be initialized as:

1. // Find min # of coins to get n dollars. p[1 : 4] is the price table.
2. // Input: int n, coin table p[1 : n]
3. // Output: min # of coins and Coin[1: n]
4. Algorithm rod\_TD(p, n, Coin)
5. {
6. Coin[1] := 1 ;
7. **for** i := 2 to n **do** {
8. min := INF;
9. **for** j := 1 to 4 **do** {
10. **if** (1 + rod\_TD(p, i − p[j], r) < min) then {
11. min := 1 + rod\_TD(p, i − p[j], r);
12. }
13. }
14. Coin[i] := min ;
15. }
16. **return** Coin[n] ;
17. }
18. **Buttom-up dynamic programming with solution, getCoin(p, n, Coin, s):**

A corresponding bottom-up dynamic programming algorithm is as the following:

1. // Find min # of coins to get n dollars. p[1 : 4] is the price table.
2. // Input: int n, coin table p[1 : n], solution table s[1: n]
3. // Output: min # of coins and Coin[1: n]
4. Algorithm getCoin(p, n, Coin, s)
5. {
6. Coin[1] := 1 ;
7. **for** i := 2 to n **do** {
8. min := INF;
9. **for** j := 1 to 4 **do** {
10. **if** (1 + Coin[i − p[j]] < min) then {
11. min := 1 + Coin[i − p[j]];
12. s[i] := p[j];
13. }
14. }
15. Coin[i] := min ;
16. }
17. **return** Coin[n] ;
18. }

For getCoin(p, n, coin), for loop on 7 ~ 15 executes n times. The inner for loop on 9 ~ 12 executes several times according to N. Thus, the total **time complexity is O(N^2)**. **The space complexity is O(n)** for coin[0 : n] array.

1. **getCoin() printing solution:**

The following algorithm can be used to print the solutions

1. // print the coin sets. p[1 : 4] is the price table.
2. // Input: int n, coin table p[1 : n], Coin[1 : n], s[1 : n]
3. // Output: each combination of the coin set
4. Algorithm printCoin(p, n, Coin, s)
5. {
6. **for** i := 1 to n **do** {
7. j := i;
8. **while** j >= 0 **do** {
9. print s[j]
10. j = j – s[i];
11. }
12. }
13. }
14. **Time & Space:**

|  |  |
| --- | --- |
|  | *getCoin()* |
| Time complexity | **O(N^2)** |
| Space complexity | **O(N)** |

**Implementation:**

1. **Workflow:**

For the first task, the workflow is as follow:

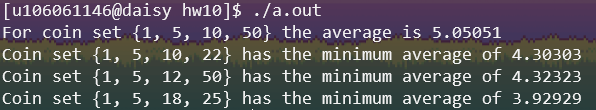
1. Algorithm first(p, n, Coin, s)
2. {
3. getCoin(99);
4. **for** i := 1 to 99 **do** {
5. sum := sum + s[i];
6. }
7. sum := sum / 100
8. }

For the second and third task, I tested the possible value of coin from 1 to 99. the workflow is as follow:

1. Algorithm first(p, n, Coin, s)
2. {
3. min := INF;
4. **for** i := p[2 or 3] to 99 or p[3] **do** {
5. p[2 or 3] := i;
6. getCoin(99);
7. **for** j := 1 to 99 **do** {
8. sum := sum + s[j];
9. }
10. sum := sum / 99
11. **if** sum < min then min := sum;
12. }
13. }

For the forth task, the workflow is as follow:

1. Algorithm first(p, n, Coin, s)
2. {
3. min := INF;
4. **for** i := p[1] to 99 **do** {
5. p[2] := i;
6. **for** j := p[2] to 99 **do** {
7. p[3] := i;
8. getCoin(100);
9. **for** k := 1 to 99 **do** {
10. sum := sum + s[k];
11. }
12. sum := sum / 99
13. **if** sum < min then min := sum;
14. }
15. }
16. }
17. **Result:**

****

* 1. When p = {1, 5, 10, 50}:

The average number of coins from 1 to 99 is 5.05.

* 1. When p = {1, 5, 10, dd}:

When dd = 22, the average number of coins from 1 to 99 is 4.3.

* 1. When p = {1, 5, dd, 50}:

When dd = 12, the average number of coins from 1 to 99 is 4.32.

* 1. When p = {1, 5, dd, dd2}:

When dd = 18, dd2 = 25, the average number of coins from 1 to 99 is 3.93.

**Conclusions:**

1. Coin set design problem can be implemented by dynamic programming.
2. Time and space complexities of *getCoin()*:

|  |  |
| --- | --- |
|  | *getCoin()* |
| Time complexity | **O(N^2)** |
| Space complexity | **O(N)** |

1. Result

