EE3980 Algorithms

Homework 9. Coin Set Design

By 105061212 王家駿

1. Introduction

There are four type of coins with different values used in Taiwan: 1, 5, 10, 50 dollars. Any price under 100 could be represented by the summation of multiple coins. In the beginning of this homework, we calculate the total number of coins used to represent the price from 1 dollar to 99 dollars by using the greedy method, and also calculate the average coins used for a specific price. Later, we redesign the coin value by three methods, replacing the largest value, replacing the second largest value, replacing both the largest two values, to get the minimum average coins used of each condition.

2. Implementation

2.1. Greedy method

For implementation, we use the greedy method, choosing the optimal solution so far at each step, as the main algorithm for solving our problem.

For a specific price P, when we try to get the minimum number of coins, we would like to pick the largest coin at first. Due to the fact that we want to reduce the price as soon as possible, picking the largest coin at first stands for reduce the price

for the most. Thus, every time when we want to take coin, we must choose the largest one whose value is not bigger than the price remained.

In addition, since the smallest value of coins is always fixed to 1 dollar, we could ensure that every positive integer price P could find its solution, with the worst-case of picking all 1-dollar coins.

```
1.  int NCoinGreedy(D, NCoin, Coins[])              // greedy algorithm
2.  {
3.      cnt := 0;                                   // initialize
4.      for i := 1 to D do {                        // for any values in 1:D
5.          dollar = i;
6.          for j := NCoin - 1 to 1 step -1 do {    // take from the biggest
7.              cnt := cnt + dollar / Coins[j];     // counting
8.              dollar := dollar % Coins[j];        // dollars remained
9.          }
10.         cnt := cnt + dollar;                    // counting the rest
11.     }
12.
13.     return cnt;
14. }
```

The input D means the maximum price we could get, and NCoin means how many types of coins with different values, and the Coins array stores the information of each coin value with its length NCoin. The loop at line 4 tries any price value from 1 to D; the loop at line 6 determines the number of coins picked for different values, and it would always pick from the coin with largest value since the Coins array is

sorted to an increasing order. At the end of the loop iterations, we could get the total number of coins used with regard to the coin values we had known.

For the time complexity, the loop at line 4 would run for D times, which indicates the input price range, and the loop at line 6 would run for NCoins times. In this problem, NCoins is 4, so we could seem the loop as constant time. We could even unroll this loop when NCoins is small. Since the other statement in the function are constant-time, the time complexity is $O(D) = O(n)$.

For the space complexity, we need two variables D and NCoins, and a fixed length array Coins as the input part, which is constant. And for the extra space needed, we need extra space for four variables, which is also constant. Thus, the space complexity is $O(1)$.

2.2. Solving for the current dollars

In this function, we use the greedy method mentioned above to solve the current situation in Taiwan. That is, find the number of coins used with 1-dollar, 5-dollar, 10-dollar, and 50-dollar coins.

Since the coin values are known, we could directly use the NCoinGreedy function to solve the problem.

```
1.  void current(D, NCoin, Coins[])            // function for current dollars
```

```
{
    assign coin values with {1, 5, 10, 50}
    total_coin = NCoinGreedy(D, NCoin, Coins);  // calculate coins needed
    avg = total_coin / D;                       // calculate average
    print out result avg;
}
```

The time and space complexity is same as the NCoinGreedy function, since the function calls it one time, and the rest statements take constant time.

Time complexity: $O(n)$

Space complexity: $O(1)$

2.3. Solving for redesign the value of the largest one

In this function, we replace the value of the largest coin, and find the value V such that it could make the average coins used least.

Since we don't know the exact value we would like to replace, we couldn't directly use the NCoinGreedy function. Instead, we try all possibilities that V might have, and find the minimum of the result. So, at each step of attempt, we could use the NCoinGreedy function to get the value.

```
void redesign1(D, NCoin, Coins[])               // redesign the biggest
{
    assign coin values with {1, 5, 10, V}

    // get value when coin3 is the successor of coin2
    Coins[3] := Coins[2] + 1;
    min_total_coin := NCoinGreedy(D, NCoin, Coins);
```

```
8.
9.      // for any possibility of coin3
10.     for i := Coins[2] + 2 to D do {
11.         // calculate coins needed
12.         Coins[3] := i;
13.         total_coin := NCoinGreedy(D, NCoin, Coins);
14.         if (total_coin < min_total_coin) {       // find minimum
15.             min_total_coin := total_coin;
16.         }
17.     }
18.
19.     avg = min_total_coin / D;                    // calculate average value
20.     print out result avg and V;
21. }
```

In the function, we set V to be Coins[2]+1 at the beginning, since V is always the largest value, and then iterate until V=D. During the iteration, we keep tracing for the minimum coins used and the target coin value. Thus, when the iteration goes to the end, we could get the minimum value and print out the result.

For the time complexity, the loop at line 11 would run for D-Coins[2]-1 times, and it would call the NCoinGreedy function in each iteration step. In addition, the other statements would not affect the complexity since they all take constant time. Thus, the time complexity is $O\big((\mathrm{D} - \mathrm{Coins}[2] - 1) * \mathrm{D}\big) = O(\mathrm{D} * \mathrm{D}) = O(n^2)$.

For the space complexity, the extra space we need is same as the current function we mentioned above, because there are just two new variables added. So, the space complexity is still O(1).

Time complexity: $O(n^2)$

Space complexity: $O(1)$

2.4. Solving for redesign the value of the second largest one

In this function, we replace the value of the second largest coin, and find the value V such that it could make the average coins used least.

The implementation is similar to the previous function. We just need to take the variable V to be the second largest one instead of the largest one.

```
1.  void redesign2(D, NCoin, Coins[])              // redesign the biggest
2.  {
3.      assign coin values with {1, 5, V, 50}
4.
5.      // get value when coin3 is the successor of coin1
6.      Coins[2] := Coins[1] + 1;
7.      min_total_coin := NCoinGreedy(D, NCoin, Coins);
8.
9.      // for any posibility of coin2
10.     for i := Coins[1] + 2 to Coins[3] - 1 do {
11.         // calculate coins needed
12.         Coins[2] := i;
13.         total_coin := NCoinGreedy(D, NCoin, Coins);
14.         if (total_coin < min_total_coin) {       // find minimum
15.             min_total_coin := total_coin;
16.         }
17.     }
18.
19.     avg = min_total_coin / D;                    // calculate average value
20.     print out result avg and V;
21. }
```

Another different part is the range of V. Since the V has to be the second large value, the possible value of V must be tried from Coins[1]+1 to Coins[3]-1 to ensure that it is not bigger than Coins[3] and smaller than Coins[1].

Because the implementation method is almost the same as the previous function, the time and space complexity are same, too.

Time complexity: $O(n^2)$

Space complexity: $O(1)$

2.5. Solving for redesign the value of the largest two

In this function, we replace the value of the largest two coins, and find the value V1 and V2 such that they could make the average coins used least.

The previous two functions solve for the situation that only one value is unknown, but the problem becomes two values are unknown. However, we could still use the similar method to solve the task.

In the previous two functions, we iterate though the array for V to try every possible solution. Yet, we could iterate through the array for finding V1, and iterate through the array for V2 simultaneously. So, the structure would become nested loops, with the outer loop trying the possibility of V1 and the inner trying for V2.

```
1.  void redesign3(D, NCoin, Coins[])              // redesign the two biggest
2.  {
3.      assign coin values {1, 5, V1, V2}
4.
5.      // get value when coin2 is the successor of coin1
6.      // and coin3 is the successor of coin2
7.      Coins[2] := Coins[1] + 1;
8.      Coins[3] := Coins[2] + 1;
9.      min_total_coin = NCoinGreedy(D, NCoin, Coins);
10.
11.     // for any possibility of coin2
12.     for i := Coins[1] + 2 to  D - 1 do {
13.         Coins[2] := i;
14.         // for any possibility of coin3
15.         for j := i + 1 to D do {
16.             // calculate coins needed
17.             Coins[3] := j;
18.             total_coin = NCoinGreedy(D, NCoin, Coins);
19.             if (total_coin < min_total_coin) {   // find minimum
20.                 min_total_coin = total_coin;
21.             }
22.         }
23.     }
24.
25.     avg = min_total_coin / D;                   // calculate average
26.
27.     print out result avg, V1, V2;
28. }
```

To satisfy the condition that V1 and V2 are the largest two, we could use the same method we mentioned above. V1 iterates from Coins[1]+1 to D-1 (D for V2), and V2 iterates from V1+1 to D.

For the time complexity, there are two loops at line 12 and 15 separately. V1 would iterate for D-Coins[1]-1 times, and V2 would iterate D-V1-2 times. The inner function would call the NCoinGreedy function for one time. Thus, the time complexity must become $O(D^2 * D) = O(n^3)$.

For the space complexity, it would take some extra space for variables, but it doesn't affect the complexity. Thus, the space complexity must be same as the previous functions, which is $O(1)$.

Time complexity: $O(n^3)$

Space complexity: $O(1)$

3. Executing results

We run the program and record the results of minimum values found.

Method	Value of each coin	Average
Reality	{1, 5, 10, 50}	5.05051
Replace the largest	{1, 5, 10, 28}	4.54545
Replace the second largest	{1, 5, 14, 50}	4.52525
Replace the largest two	{1, 5, 13, 42}	4.24242

4. Result analysis and conclusion

From the results, we could observe that the average number of coins used in the currency system of Taiwan is about 5.05. When we modify one of the coin value, the average number could reduce to about 4.5, and if we modify two of them, the average number could reduce to about 4.24.

However, although we could lower the average number of coins, it is still hard to use in reality, since the coin values are not multiples of 5, which would increase the difficulty in calculation. Thus, the current coin values still have their advantage in the real world.

In the greedy algorithm function, we want to make the best choice in every step, so we always choose the largest element remained. However, we haven't show that the best choice of every step could lead to an optimal solution of the problem. In the case Coins = {1, 5, 13, 14} for a specific target price 18, the greedy algorithm would take the largest (14-dollar) coin first and 4 dollars remained. Then it would take four 1-dollar coins and the total coins used is 5. While in the reality, we could just pick a 5-dollar coin and a 13-dollar coin and the number of total coins becomes 2.

In the case of reality system where Coins = {1, 5, 10, 50}, the greedy algorithm works well. Because the latter ones are multiples of former ones for all element in Coins, if a specific price P>50 could be represented as the composition of lower value

10’s, it must be revised by getting a 50-dollar coin instead of five 10-dollar coins. The relation could be extended to the lower prices, 10, 5, and 1, so the greedy algorithm is suitable for this given input.

Thus, in some cases, the best choice in each step doesn’t lead to the optimal solution of this problem. The solution to a specific price depends not only on the largest element but on the compositions of other values. So, the greedy algorithm is not suitable for this problem, we need to add other methods like brute-force or dynamic programming to find out the optimal solution.