# EE3980 Algorithm

# HW9. Coin Set Design

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 greedy method to solve the problem. Note that the greedy method doesn't bring us the optimal solution. Although it chooses the optimized path in each process, it is unable to record the last situation, implying that once we made a choice then we can't recover it. In this report I am going to implement greedy method and apply it on the problem of coin set design.

**2. Approach:**

To approach this algorithm, I am going to define two global variables *NCoin, p[4]* (although it is not that necessary since there are few functions being called). 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. The only function that is called in this assignment *NCoinGreedy()* executes the greedy method and returns the minimum average of coins being used from $1 to $99. 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
In main:
Double Avg;                         // is used to store every execution of NCoinGreedy()
Double minAvg;                      // is used to store the current minimum average
```

```
Algorithm main
{
    Write (NCoinGreedy());

    for p[3] := p[2] + 1 to NCoin do {
        avg := NCoinGreedy();
        if (avg < MinAvg) then {
            minAvg = avg;
            dd := p[3];
        }
    }
    Write(dd, minAvg);
    p[3] := 50;

    for p[2] := (p[1] + 1) to (p[3] - 1) do {
        avg := NCoinGreedy();
        if (avg < MinAvg) then {
            minAvg := avg;
            dd := p[2];
        }
    }
    Write(dd, minAvg);

    for p[2] := (p[1] + 1) to NCoin - 1 do {
        for p[3] := (p[2] + 1) to Ncoin do {
            avg := NCoinGreedy();
            if (avg < MinAvg) then {
                minAvg := avg;
                dd := p[2];
                dd2 := p[3];
            }
        }
    }
    Write(dd, dd2, minAvg);
}
```

We are going to execute greedy method for 4 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 NCoinGreedy(), while second and third set requires NCoin times of NCoinGreedy(), forth for $NCoin^2/2$ times of NCoinGreedy() to find the coin value with minimum average.

## 3. Analysis:

In this part I am going to discuss how I implement greedy method as c code to solve the problem. In general, we all wish to exchange money to coins with fewer coin number as possible, that is, if we are going to take \$50, then a 50 dollar coin will be more preferable than fifty 1 dollar coin. Thus, to obtain a value by using as few coins as possible, we check the largest coin that is less than the value in every iteration, then subtract the current value with the that coin then check the remaining value to exchange to coin, until the value is smaller than p[2] (which is 5 in this homework). An instance of using greedy method to separate 77 dollars and original coin set with minimum coins looks like this:

1.77 > 50, combination = {50}, 77 - 50 = 27
2. 10 < 27 < 50, combination = (50, 10, 10}, 27 - 10 = 17
3. 10 < 17 < 50, combination = {50, 10, 10}, 17 - 10 = 7
4. 5 < 17 < 10, combination = {50, 10, 10, 5}, 7 - 5 = 2
5. 1 < 2 < 5, combination = {50, 10, 10, 5, 1}, 2 - 1 = 1
6. Combination = {50, 10, 10, 5, 1, 1}

The NCoinGreedy() pseudo code looks like below:

```
Algorithm NcoinGreedy(void)
{
    for I := 1 to NCoin do {                  // total value from 1 to NCoin
        lmin := I;                            // initialize each value to i 1 dollar coins
        value := I;                           // check the next biggest value coin
        for j := 4 to 1 step -1 do {          // check from the biggest value coin
            while (value > p[j]) do {
                lmin := lmin - p[j] + 1;      // update minimum coin number
                value := value - p[j];        // update value
            }
        }
        sum := sum + lmin;                    // calculate sum of all value
    }
    return sum / NCoin;                       // return average
}
```

I used two local variables to store informations I want, *lmin* is for recording the total coin number I used on a value, and *value* is used for checking what is the optimized choice (largest coin value) I can make in this step. In addition, since this assignment only require us to return the average minimum coin number, so there is no need for an extra array with size NCoin to store each value's coin number.

The next problem is that can greedy algorithm always bring us optimized solution, and the answer is not always. For the original set the minimum average coin would be $5.\overline{05}$, and by using the greedy method we can also get the same answer, which brings us the optimal solution. However, the optimal result of greedy method is solved by accident. Since in the original coin set, for every *i* in the range of 1 to 3, we can know that p[i] can be divided by an integer then get p[i - 1]. This property indicates that in every sub problem where we have to make selection, choosing the largest p[i] that is smaller than current value is the optimal solution since multiple coins with value of p[i - 1] can be substituted by a coin of p[i].

Let's look at an example. Once we are looking for the correct p[3] in range of 1 to 99 such that the whole coin set leads to minimum average of coin number, and it's the turn of 22. When we apply greedy method on a total value of 97 with coin set {1, 5, 10, 22}, it might be broken into the combination {22, 22, 22, 22, 5, 1, 1, 1, 1}, which has a total of 9 coins. However, if we adjust the forth step to change 22 to 10, then we can have the combination of {22, 22, 22, 10, 10, 10, 1}, which inclined to a total of 7 coins. If we look closer into the example of 30 dollars, it is easy to observe that the set {10, 10, 10} is better than the set {22, 5, 1, 1, 1}, which confirms that choosing the largest available coin value doesn't always lead to the optimal solutions. If we are going to pursue the absolute optimization, using dynamic programming may be the key to the solution.

## 4. Results:

| Set number | Coin set | Minimum average |
|---|---|---|
| 1 | {1, 5, 10, 50} | $5.\overline{05}$ |
| 2 | {1, 5, 10, 28} | $4.\overline{54}$ |
| 3 | {1, 5, 14, 50} | $4.\overline{52}$ |
| 4 | {1, 5, 13, 42} | $4.\overline{24}$ |

## 5. Observation and conclusion:

For space complexity, since there is no need to store each value's solution, so there are merely constant variables such as sum, lmin, value, minAvg, etc. which is unrelated to the size of NCoin we are going to test, thus the space complexity is O(1). On the other hand, for one time of NCoinGreedy(), we are going to calculate NCoin value's solution, with each of the value at most 6 times (99). So we can then get the total time complexity of O(n^3), which is dominated by the last set.

| | |
|---|---|
| Time complexity | O(n^3) |
| Space somplexity | O(1) |

The greedy method can't reach optimal solution, we have to seek for another method.