# 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 ~ 100 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 is to simplifying a complicated problem by **breaking it down into simpler sub-problems** in a recursive 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.

## 2. 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 |

## 3. Recursive function:

Using recursive function, we can formulate the solution as:

$$Coin\_R(n) = 1 + Coin\_R(n - p[i]), p[i] \in \{1, 5, 10, 50\}$$

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:

```
// Find min # of coins to get n dollars. p[1 : 4] is the price table.
// Input: int n, coin table p[1 : n]
// Output: min # of coins.
Algorithm coin_R(p, n)
{
    if (n = 1) return 1 ;
    min := INF ; // no cut.
    for i := 1 to 4 do { // check all combination using recursion.
```

```
        if (1 + rod_R(p, n - p[i] ) < min) then {
            min := 1 + rod_R(p, n - p[i]);
        }
    }
    return min ;
}
```

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].

**4. 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:

$$Coin[n] = 1 + Coin\big[n - p[i]\big], p[i] \ \in \{1, 5, 10, 50\}$$

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

$$Coin[i] = \begin{cases} 1, & if\ i = 1, \\ \infty, & otherwise. \end{cases}$$

```
// Find min # of coins to get n dollars. p[1 : 4] is the price table.
// Input: int n, coin table p[1 : n]
// Output: min # of coins and Coin[1: n]
```

```
Algorithm rod_TD(p, n, Coin)
{
    Coin[1] := 1 ;
    for i := 2 to n do {
        min := INF;
        for j := 1 to 4 do {
            if (1 + rod_TD(p, i - p[j], r) < min) then {
                min := 1 + rod_TD(p, i - p[j], r);
            }
        }
        Coin[i] := min ;
    }
    return Coin[n] ;
}
```

## 5. Buttom-up dynamic programming with solution, getCoin(p, n, Coin, s):

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

```
// Find min # of coins to get n dollars. p[1 : 4] is the price table.
// Input: int n, coin table p[1 : n], solution table s[1: n]
// Output: min # of coins and Coin[1: n]
Algorithm getCoin(p, n, Coin, s)
{
    Coin[1] := 1 ;
    for i := 2 to n do {
        min := INF;
        for j := 1 to 4 do {
            if (1 + Coin[i - p[j]] < min) then {
                min := 1 + Coin[i - p[j]];
               s[i] := p[j];
            }
        }
        Coin[i] := min ;
    }
    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.

## 6. getCoin() printing solution:

The following algorithm can be used to print the solutions

```
// print the coin sets. p[1 : 4] is the price table.
// Input: int n, coin table p[1 : n], Coin[1 : n], s[1 : n]
// Output: each combination of the coin set
Algorithm printCoin(p, n, Coin, s)
{
    for i := 1 to n do {
        j := i;
        while j >= 0 do {
            print s[j]
            j = j - s[i];
        }
    }
}
```

## 7. Time & Space:

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

## Implementation:

## 1. Workflow:

For the first task, the workflow is as follow:

```
Algorithm first(p, n, Coin, s)
{
    getCoin(100);
    for i := 1 to 100 do {
        sum := sum + s[i];
    }
    sum := sum / 100
}
```

For the second and third task, I tested the possible value of coin from 1 to 100.

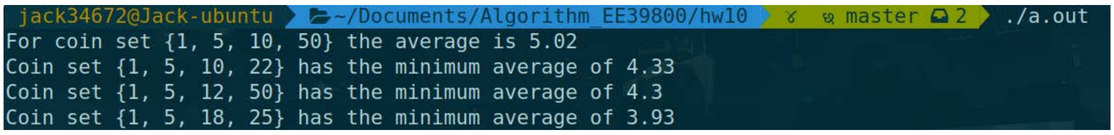
the workflow is as follow:

```
Algorithm first(p, n, Coin, s)
{
    min := INF;
    for i := p[2 or 3] to 100 or p[3] do {
        p[2 or 3] := i;
        getCoin(100);
        for j := 1 to 100 do {
            sum := sum + s[j];
        }
        sum := sum / 100
        if sum < min then min := sum;
    }
}
```

For the forth task, the workflow is as follow:

```
Algorithm first(p, n, Coin, s)
{
    min := INF;
    for i := p[1] to 100 do {
        p[2] := i;
        for j := p[2] to 100 do {
            p[3] := i;
            getCoin(100);
            for k := 1 to 100 do {
                sum := sum + s[k];
            }
            sum := sum / 100
            if sum < min then min := sum;
        }
    }
}
```

## 2. Result:



```
jack34672@Jack-ubuntu  ~/Documents/Algorithm_EE39800/hw10  master 2  ./a.out
For coin set {1, 5, 10, 50} the average is 5.02
Coin set {1, 5, 10, 22} has the minimum average of 4.33
Coin set {1, 5, 12, 50} has the minimum average of 4.3
Coin set {1, 5, 18, 25} has the minimum average of 3.93
```

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

The average number of coins from 1 to 100 is 5.02.

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

When dd = 22, the average number of coins from 1 to 100 is 4.33.

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

When dd = 12, the average number of coins from 1 to 100 is 4.3.

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

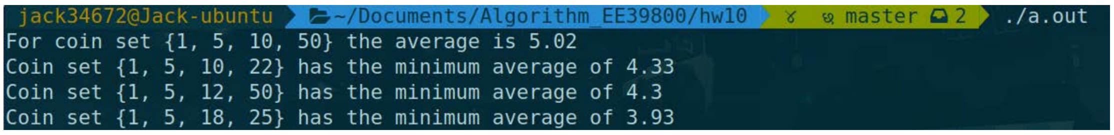
When dd = 18, dd2 = 25, the average number of coins from 1 to 100 is 3.93.

**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)** |

3. Result



```
 jack34672@Jack-ubuntu  ~/Documents/Algorithm_EE39800/hw10   master 2  ./a.out
For coin set {1, 5, 10, 50} the average is 5.02
Coin set {1, 5, 10, 22} has the minimum average of 4.33
Coin set {1, 5, 12, 50} has the minimum average of 4.3
Coin set {1, 5, 18, 25} has the minimum average of 3.93
```