Coin Change Problem (Dynamic Programing)

Problem URL: https://www.hackerrank.com/challenges/coin-change/problem

Primitive Algorithm: Try to search all possible combinations of coins:

Example:

```
C = {1, 2, 3};
m = 3;
n = 7;
```

pseudocode:

```
an array c[] with size of m records how many coins each type used;
an integer time;
loops on all combinations of coins:
    if sum of values in c[] equals n, times++;
loops end;
return time;
```

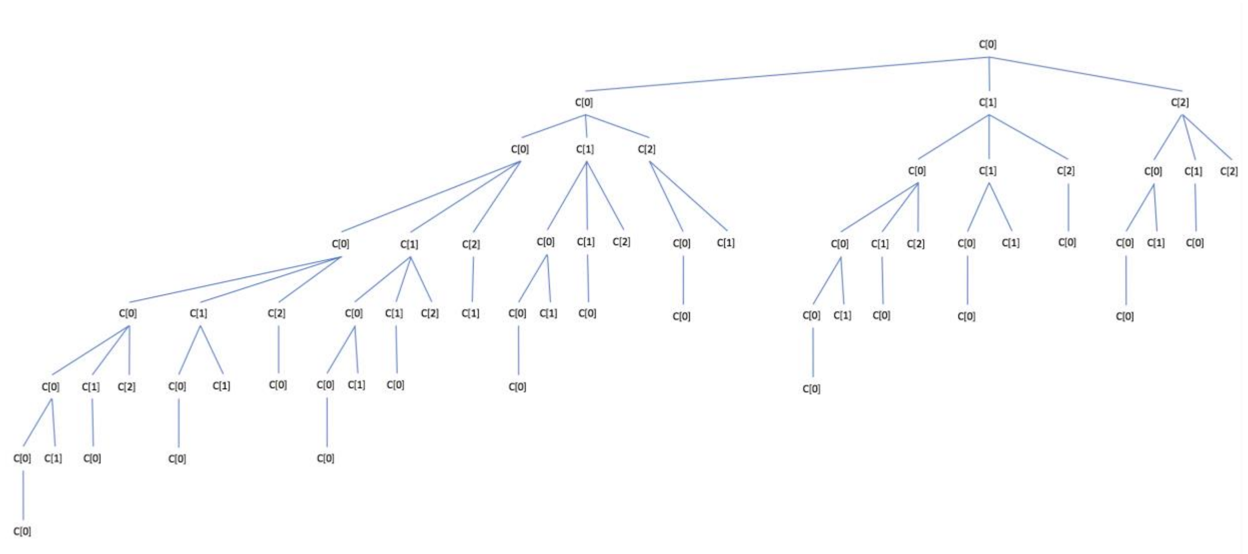
runtime analysis:

there are loops over all types and the worst case is looping m times on each type: O(m^n);

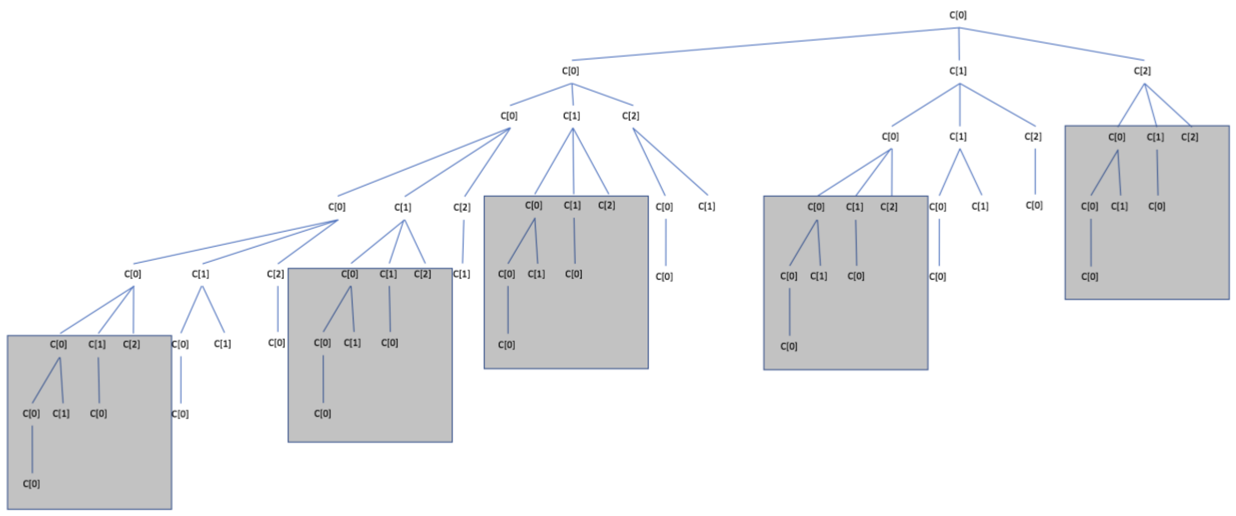
Memory usage:

We created an one-dimension array: O(m);

Recursive path:



we can clearly see that there are many duplicated subtrees, such as:



so, if we can find a method to record that status, we can cut all of them to increase efficiency. Then, we find out that duplication happens when reaching the same statuses. In this question, same amount of remaining money, so we could use a collection to save information after encountering the status at very first.

Improved Algorithm: While searching, we also record how much we have already changed (which also helps us to calculate how much left).

pseudocode:

```
an array dp[][] which first dimension records combinations and second
dimension records status(how much we have already changed).
For each combination:
    For each status:
        combinations now in dp[][] = new combinations (new status) +
        recorded status;
loops end;
return dp last status;
```

runtime analysis:

there are two loops: one is O(n), another is O(m). Therefore, the runtime is O(mn).

Memory usage:

We created a two-dimension array: one dimension is O(n), another is O(m). Therefore, the usage is O(mn);

C++ Code:

```
long getWays(long n, vector < long > c){
    int m = c.size();
    long dp[m + 1][n + 1];

    for(int i = 0; i <= m; i++) {
        dp[i][0] = 1;
    }

    for(int i = 0; i <= n; i++) {
        dp[0][i] = 0;
    }

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (j - c[i - 1] >= 0) {
                dp[i][j] = dp[i - 1][j] + dp[i][j - c[i - 1]];
            } else {
                dp[i][j] = dp[i - 1][j];
            }
        }
    }

    return dp[m][n];
}
```

this algorithm is what we call Dynamic Programming, which takes the advantage of memorization.