Lab 07. Dynamic Programming 2: Counting Problems

> Don't repeat yourself. It's not only repetitive, it's redundant, and people have heard it before.
>
> Lemony Snicket, Who Could That Be at This Hour

In theoretical computer science, a computational problem asks for a solution in terms of an algorithm. A computational problem will give you some instances and a solution/goal.

We can classify computational problems into several types.

- Decision problem: to answer **TRUE** or **FALSE**.

  > Instance: A graph $G = (V, E)$, $s, t \in V$ represents start and target vertex.
  >
  > Is there exists a path from s to t on graph G?

- Counting problem: to answer the number of solutions.

  > Instance: A graph $G = (V, E)$, $s, t \in V$ represents start and target vertex.
  >
  > How many different paths we can find from s to t on graph G?

- Optimization problem: Find the **best** solution.

  > Instance: A weighted graph $G = (V, E)$, $s, t \in V$ represents start and target vertex.
  >
  > Find a path from s to t which minimizes the summation of weights.

"Dynamic programming" (hereinafter referred to as "DP") is an algorithm design paradigm suitable for solving optimization problems and counting problems.

The most naive way for solving counting problems and optimization problem is by a brute-force algorithm. Brute-Force is a general problem-solving approach that enumerates all the possible solution candidates and check/verify each candidate satisfied the constraints or not.

Brute-force algorithms are easy to understand, and good to proof the correctness. But in most of the cases, they are not efficient enough. To improve performance and maintenance, we have 2 principles in software engineering, "YAGNI" principle and "DRY" principle.

- YAGNI: You aren't gonna need it.
- DRY: Don't repeat yourself.

DP and Branch-and-bound are two algorithm design paradigms that improve the brute-force approach by "YAGNI" and "DRY" problems.

Let's first discuss "YAGNI" first. DP is suited for the problems that have recurrence structure, in other words, the solution of the problem could be combined by the solution of subproblems.

Assume that we're solving the above optimization problem, the shortest path problem. In typical brute-force algorithm, we may list all permutations of V, then test the legalities (all adjacent vertices pair $(v_i, v_{i+1}) \in E$) and measure the cost. But consider about the recurrence structure, we only need to check:

1. Direct link, $(s, t) \in E$
2. Path through other vertex $m \in V$, than the answer should be the combination of 2 subproblems (G, s, m) and (G, m, t).

Even though we try all the possible midpoints, it still much faster than brute-force approach.

For "DRY" principle, we can memorize the solution of each subproblems in a table. Thus, we only compute the solution first time. If a subproblem is queried several times, we only return the value that we store in the table since for second time and after.

Question 1. We solve the question by dividing the problem to subproblems both in DP paradigm and divide-and-conquer algorithm. What's the difference between them?

Implement a DP approach: Fibonacci number

The Fibonacci sequence is a sequence in which each element is the sum of the previous two elements. A native recursive implementation is as follows:

```
int F(int n) {
    if (n < 1) return 0;
    else if (n == 1) return 1;
    else return Fib(n-2) + Fib(n-1);
}
```

Assume $T(n)$ is the time complexity approximation of the above program. We have $T(0) = c_0, T(1) = c_1, T(n) = T(n-2) + T(n-1) + c_2$ when $n > 1$. Consider the call graph of the above program; the number of leaves is equal to the count of $c_0 + c_1$, and the number of internal nodes is equal to the count of c_2. Because every internal node has exactly two children, we have the count of $c_2 = \frac{|E|}{2}$. For every tree, we have $|E| = |V| - 1 \rightarrow 2c_2 = c_0 + c_1 + c_2 - 1 \rightarrow c_2 = c_0 + c_1 - 1$.

We can calculate the count of $c_0 + c_1$ by let $c_0 = c_1 = 1$ and $c_2 = 0$. In this situation, we have $T(0) = 1, T(1) = 1, T(n) = T(n-2) + (n-1)$. We can find that $T(0) = F(1)$ and $T(1) = F(2)$, so we have $T(n) = F(n+1)$.

The time complexity of $T(n)$ is equal to the count of $c_0 + c_1 + c_2 = c_0 + c_1 + c_0 + c_1 - 1 = 2F(n+1) - 1$. By the close form of Fibonacci numbers, we know the above program runs in exponential time.

The above program is slow because we need to recalculate the value every single time. If improve above program by DP approach, store known answer to the table, the program will much faster than the original one.

Top-down or bottom-up

When we runs a DP approach algorithm, we have 2 different solving strategy of subproblems, namely, top-down and bottom-up.

- Top-down: Check the current problem is solved or not. If yes, return the answer. Otherwise, solve the problem recursively.

```
#include <vector>

int F(int n) {
    static std::vector<int> Fib = {0, 1};
    if (Fib.size() <= n)
        Fib.resize(n+1, -1);
    if (Fib[n] == -1)
        Fib[n] = F(n-2) + F(n-1);
    return Fib[n];
}
```

- Bottom-up: Solve all subproblems that be depended on any other bigger problems iteratively.

```
#include <vector>

int F(int n) {
    static std::vector<int> Fib = {0, 1};
    if (Fib.size() <= n) {
        int i = Fib.size();
        Fib.resize(n + 1);
        for (; i <= n; ++i)
            Fib[i] = Fib[i-2] + Fib[i-1];
    }
    return Fib[n];
}
```

There is no absolute superiority between the top-down and bottom-up approaches. The main advantage of the top-down approach is that it solves only the necessary subproblems. In contrast, the bottom-up approach solves all subproblems iteratively, which often results in a better constant factor by avoiding the overhead of recursive calls.

Climbing stairs

> You are climbing a staircase. It takes n steps to reach the top. We can climb either 1 or 2 steps every round. How many ways can we climb to the top?

Climbing stairs is a typical counting problem. A counting problem asks the number of solutions, in other words, the distinct ways that we can reach the given state. In these kinds of problems, we usually consider the last step, and solve the left part recursively.

We can solve the problems by these steps:

1. Assume we have a function that can solve the problem that the parameters are all the given instances.
   $f(n)$
2. Consider all the possible last steps, and find the answer recursively.
   Last step may climb either 1 or 2 steps $\rightarrow f(n) = f(n-1) + f(n-2)$
3. List up all the base states:
   3.1 At the beginning, we have only one state -- climbing no steps.
   3.2 We do not climb down, so we cannot reach negative steps.

$$f(n) = \begin{cases} 1, & n = 0, \text{ by } 3.1 \\ 0, & n < 0, \text{ by } 3.2 \\ f(n-1) + f(n-2) & \text{otherwise, by } 2 \end{cases} \tag{1}$$

Using the above equation set, we can solve the problem by the way we compute Fibonacci numbers.

Question 2. In basketball, each successful shot gains either 2 or 3 points. Please write a program to calculate how many distinct ways that a team can get exactly n points.

Question 3. Some of the stairs were broken. We can not step on the broken stairs. Given the broken stairs by $B = \{b_1, b_2, \ldots, b_m\}, 1 \leq b_1 < b_2 < \cdots < b_m \leq n$, how many ways can we climb to total n stairs?

Coin change - count ways to make sum

> Given coins denominations and an amount of money. You may assume that you have enough numbers of each kind of coin. Please find the number of combinations of coins that make up that amount.

Formally,

> Instance: $C = \{c_1, c_2, \ldots, c_n\}, m$ where $c_i, n \in \mathbb{N}$
> Result: number of $A = \{a_1, a_2, \ldots, a_n\}$ that satisfied $\sum_{i=1}^{n} a_i c_i = m$

The coin change problem asks for combinations, not permutations. Thus, we decide the quantity of one kind of coin we will use in each step.

1. Let $C_k = \{c_1, c_2, \ldots, c_k\}, k \leq n$. The transition function should be $f(C_k, m) \rightarrow \mathbb{N}$
2. $a_k c_k < m \rightarrow a_k \in \{\mathbb{N} | 0 \leq a_c \leq \lfloor \frac{m}{c_k} \rfloor\}$. $f(C_k, m) = \sum_{i=0}^{\lfloor \frac{m}{c_k} \rfloor} f(C_{k-1}, m - i \cdot c_k)$
3. $m = 0$ shows a correct total amount, and $m < 0$ is not a legal solution.

$$f(C_k, m) = \begin{cases} 0 & m < 0 \\ 1 & m = 0 \\ f(C_k, m) = \sum_{i=0}^{\lfloor \frac{m}{c_k} \rfloor} f(C_{k-1}, m - i \cdot c_k) & \text{otherwise} \end{cases} \tag{2}$$

Because C is known, we can represent C_k by k. Thus, we can use a 2-dimension table, `dp[n][m]`, to cache the result of subproblems. Followed is a bottom-up approach implementation.

```
#include <vector>

int coin_change(const std::vector<int> &C, int m) {
    int dp[C.size()][m+1];
    for (int ik = 0; ik < C.size(); ++ik)
        dp[ik][0] = 1;
    for (int im = 1; im <= m; ++im)
        dp[0][im] = (im % C[0] == 0)? 1: 0;
    for (int ik = 1; ik < C.size(); ++ik)
        for (int im = 1; im <= m; ++im) {
            dp[ik][im] = 0;
            for (int a = 0; a*C[ik] <= im; ++a)
                dp[ik][im] += dp[ik - 1][im - a*C[ik]];
        }
    return dp[C.size()-1][m];
}
```

Consider $f(C_k, m)$ and $f(C_k, m + c_k)$.

$$f(C_k, m + c_k) = \sum_{i=0}^{\lfloor \frac{m}{c_k} \rfloor + 1} f(C_{k-1}, m - i \cdot c_k) = \sum_{i=0}^{\lfloor \frac{m}{c_k} \rfloor} f(C_{k-1}, m - i \cdot c_k) + f(C_{k-1}, m + c_k) = f(C_k, m) + f(C_{k-1}, m + c_k) \quad (3)$$

Thus, we can rewrite the function

$$f(C_k, m) = \begin{cases} 0 & m < 0 \\ 1 & m = 0 \\ f(C_k, m) = f(C_k, m - c_k) + f(C_{k-1}, m) & \text{otherwise} \end{cases} \quad (4)$$

then to implement like:

```
#include <vector>

int coin_change(const std::vector<int> &C, int m) {
    int dp[C.size()][m+1];
    for (int ik = 0; ik < C.size(); ++ik)
        dp[ik][0] = 1;
    for (int im = 1; im <= m; ++im)
        dp[0][im] = (im % C[0] == 0)? 1: 0;
    for (int ik = 1; ik < C.size(); ++ik)
        for (int im = 1; im <= m; ++im) {
            dp[ik][im] = dp[ik-1][im];
            if (im >= C[ik])
                dp[ik][im] += dp[ik][im - C[ik]];
        }
    return dp[C.size()-1][m];
}
```

Question 4. Will sorting C by increasing order improve the performance or not? Why?

Question 5. Since $f(C_k, *)$ only depends on $f(C_k, *)$ and $f(C_{k-1}, *)$, we can cache the data in $2 \times m$. The skill named rolling. Please rewrite above algorithm to a rolling table version.

> Hint: store $f(C_k, m)$ to `dp[k%2][m]`