Introduction to Dynamic Programming

1 Fibonacci Again!

Let us consider the problem to find the nth fibonacci number. Here's the infamous recursive algorithm to solve the problem:

```
int fibonacci(int n) {
        if (n == 1) return 1;
        if (n == 2) return 1;
        return fibonacci(n-1) + fibonacci(n-2);
}
```

While this algorithm is correct, it is **terribly slow**! Suppose I want to find fibonacci(6), then the recursion will call fibonacci(3) three times! Furthermore, each time I invoke fibonacci(3), I'll call fibonacci(2) and fibonacci(1). Do I really need to recalculate fibonacci(3) every time I call it? Of course not! Instead, we can "remember" the solution:

```
// fib[i] will store the result of fibonacci(i). Initially all 0.
int[] fib = new fib[32];

int fibonacci(int n) {
        if (n == 1) return 1;
        if (n == 2) return 1;
        if (fib[n] != 0) return fib[n]; // already calculated fibonacci(n)

        return fib[n] = fibonacci(n-1) + fibonacci(n-2);
}
```

This algorithm is in fact dynamic programming! Let us analyze the algorithm more carefully:

1. We used the **recurrence relation** fibonacci(n) = fibonacci(n-1) + fibonacci(n-2). Essentially, we broke the problem of finding nth Fibonacci number into smaller subproblems - finding the (n-1)th and (n-2)th fibonacci number.

2. We characterized the problem of finding the nth fibonacci number by a single integer n. This is the **state** of the problem.

3. We realized that we might be solving the same subproblem multiple times, so we can optimize the performance by remembering our solution for the subproblem.

2 Dynamic Programming - Defined Formally

Dynamic programming is essentially a divide and conquer algorithm. We use a recursive algorithm to solve the problem by solving its subproblems, and we remember to remember the solution to the subproblem so we don't do extra work. There are 2 key components to a dynamic programming algorithm: the states and the recurrence relation.

2.1 States

The state is how we use to describe the problem/subproblem. When we define the state, we must adhere to the following properties:

- The state must *uniquely* describe the problem/subproblem.
- There is exactly one state to describe a particular problem/subproblem. In other words, how we reached this subproblem should not affect how the state is defined.

2.2 Recurrence Relation

The recurrence relation is the ”formula” that allow us to construct the solution of the current problem based on the solutions of the subproblems. The recurrence relation is defined based on the state. Thus, the formula can only depend on the information encoded by the state. The important thing to note is that the recurrence can not be ”circular”. In other word, the recurrence relation cannot depend on the result of what we are trying to find. This will lead to infinite recursion.

2.3 Implementation

Once we have define the state and the recurrence relation, the next step is to implement it. Suppose we have defined the state of the problem by described the class State and the solution we want is described the by class Solution. The pseudocode to implement the DP algorithm is as follow:

```
// used to store the solutions for subproblems
Map<State, Solution> memo;

Solution DP(State s) {
        // base case of recurrence
        if (s == base_case) return base_solution;

        // if the solution has already been found
        if (memo.containsKey(s)) return memo.get(s);

        // find the solution using the recurrence relationship
        Solution ans = recurrence_relation(s);

        // store and return the solution
        memo.put(s, ans);
        return ans;
}
```

Note that in the pseudocode, I used a Map to remember the solution for the state. In general, it is better to encode the state using integers, and then use an (multi-dimensional) array to store the results.

3 Putting It Together

From the previous section, it's not hard to see that the hardest part to come up with DP algorithm is to define the state and find the recurrence relation. While this may seem easy, there are a few points to keep in mind:

- We are storing the solution for each state and we have limited memory. So the total number of states, or called the **state space** cannot be too large (eg. 2^n when $n = 100$).
- The recurrence relation **MUST NOT BE CIRCULAR**.
- We also have a limited amount of time to run our algorithm. In general, the run-time complexity of the DP algorithm is O(state space $*$ recurrence relation). So even if the state space is small enough, we must take care that our recurrence relation is not overly complex.

4 Example

A classical problem in DP is the coin changing problem. Suppose there are several denominations of coins in a currency and you are trying to make a particular amount N. Assuming that you have an infinite number of coins of each denomination, what is the minimum number of coins you need? The state of the problem is fairly easy to see (there isn't much variables involved). The state is simply the amount that we are trying to make - we'll represent the state of trying to make the change for n by the integer n. The recurrence relation then follows:

$$f(n) = 1 + \min_{denominations\ d} f(n-d)$$

```
int memo[128]; // initialized to -1

int min_coin(int n) {
        // base cases
        if (n < 0) return INF;
        if (n == 0) return 0;
        if (memo[n] != -1)

        // recurrence relation
        int ans = INF;
        for (int i = 0; i < num_denomination; ++i)
                ans = min(ans, min_coin(n - denominations[i]));

        return memo[n] = ans + 1;
}
```