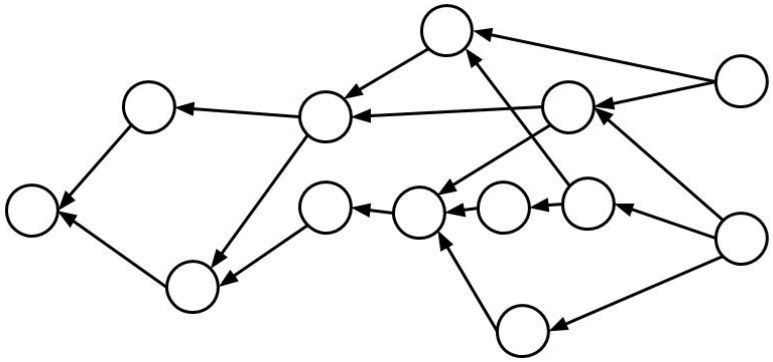
上一章讲到了动态规划问题中的子问题构建方法。在一个动态规划问题中，如果把每个子问题的解看做一个状态，则解题可以被看作在子问题间不断进行状态转移，最终得出原问题答案的过程。构建出子问题之后，找出状态转移的方法就成了关键。这一章将重点讲解找出状态转移公式的方法。

因为子问题结构相似，不妨猜测存在一种状态转移方法适用于一个动态规划问题中所有子问题间的转移，用一个公式来表示这种方法。

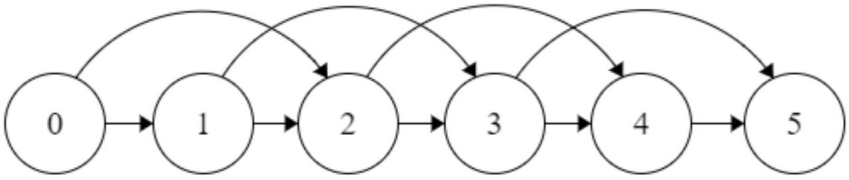
举一个简单的例子：有一个人要上楼梯，楼梯一共有n个台阶，这个人一次可以上1个台阶或2个台阶，问这个人有几种不同的上台阶方法（经百度发现这竟然是Google面试题）。根据上一章提到的方法“把一个问题的解拆分为一系列通向最优解的选择”，可以发现在每一个台阶上我们都可以选择上1个台阶或者上2个台阶，此时我们我们需要记录当前在第几个台阶以及到达当前台阶有几种不同的走法，可以用一维矩阵表示。在这个矩阵中，每个数字本身（value，在这里代表v种不同走法）和该数字所在的位置（index，在这里代表第i个台阶）共同组成一种状态的两个属性。



（from DAGlabs）

接下来就要找出状态转移公式了。首先，通常不是每个子问题都需要向所有其它子问题进行转移（那样的话就是一个全连接图）。只有状态之间存在依赖关系，在图论中相当于两个顶点之间存在边，我们才有机会用上状态转移公式。所以当务之急是找到具有完整依赖关系的一系列状态。为什么说是一系列状态而不是两个状态呢？因为在动态规划中不仅存在一对一的依赖关系（即状态s仅影响状态t，且状态t仅依赖状态s），还存在一对多，多对一的依赖关系。当一个问题中的所有子问题仅存在一对一和多对一的依赖关系时，或仅存在一对一和一对多的依赖关系时，直接用贪心就完事儿，这里不再赘述。剩下的一种贪心解决不了的情况就是一对多和多对一同时交叉存在（即把所有表示依赖关系的有向边替换为无无向边后出现环），这种情况则需要用到动态规划。

如下图所示，不难看出上台阶问题属于一对多和多对一交叉存在的情况。对于某一台阶i，它的状态会影响到台阶i+1和台阶i+2的状态（一对多），同时，台阶i+2的状态被台阶i和台阶i+1所影响（多对一）。



（作者瞎画的）

我们已知在初始状态（上图所示台阶0）有一种走法，需要求出在台阶n有几种走法，其中子问题为在台阶i有几种走法（$0 \le i \le n$）。设在台阶i有f(i)种走法。对于$i \ge 2$，可以发现f(i) = f(i - 1) + f(i - 2)，即所有到台阶i-1的不同路径一次再上1个台阶和所有到台阶i-2的不同路径一次再上2个台阶。这一推导类似递推公式或归纳法证明。有人可能会发现这题的本质和上一章斐波那契数列的例子是一样的，没错，在这里举这个例子的目的是阐述不同的侧重点。有了子问题状态定义和递推公式后可以很容易得出答案，对于上楼梯问题具体答案可以参考上一章。

大部分读者可能会觉得上个例子太简单了，接下来我们用上一章Leetcode的三个例子进行分析。

44. Wildcard Matching

Hard 1448 84 Favorite Share

Given an input string (`s`) and a pattern (`p`), implement wildcard pattern matching with support for `'?'` and `'*'`.

```
'?' Matches any single character.
'*' Matches any sequence of characters (including the empty sequence).
```

The matching should cover the **entire** input string (not partial).

Note:

- `s` could be empty and contains only lowercase letters `a-z`.
- `p` could be empty and contains only lowercase letters `a-z`, and characters like `?` or `*`.

Example 1:

```
Input:
s = "aa"
p = "a"
Output: false
Explanation: "a" does not match the entire string "aa".
```

（from Leetcode）

对于Leetcode 44. Wildcard Matching，我们记录正则表达式p匹配到的位置i，文本s匹配到的位置j，和目前为止匹配成功与否（boolean），可以用二维矩阵dp[i][j]表示。一次字符串之间的比较相当于各出一个当前匹配位置的字符进行匹配。因此，如果排除正则表达式中的'*'，可以推出状态转移公式dp[i+1][j+1] = dp[i][j] && (p[i+1] == s[j+1])。

换句话说，p[0:i+1]（即正则表达式第0到第i+1个字符）和s[0:j+1]（即文本第0到第j+1个字符）是否匹配取决于：

1. p[0:i]和s[0:j]是否匹配
2. p[i+1]与s[j+1]是否匹配

如果以上两个先决条件成立，则p[0:i+1]和s[0:j+1]匹配，此时仅存在一对一的依赖关系。如果p[i+1]为'*'，则dp[i+1][j+1:len(s)]均视作匹配，其中len(s)为文本s的长度。最终dp[len(p)-1][len(s)-1]的值为题目答案。

值得一提的是，'*'的存在导致了一对多的依赖关系出现，但是并没有多对一的依赖关系！所以我们可以发现这是一个披着动态规划皮的贪心问题！此题贪心解法时间复杂度为O(n)！详情请见Leetcode Discussion :)

188. Best Time to Buy and Sell Stock IV

Hard 1027 64 Favorite Share

Say you have an array for which the i-th element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete at most **k** transactions.

Note:
You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

Example 1:

```
Input: [2,4,1], k = 2
Output: 2
Explanation: Buy on day 1 (price = 2) and sell on day 2 (price = 4), profit =
4-2 = 2.
```

Example 2:

```
Input: [3,2,6,5,0,3], k = 2
Output: 7
Explanation: Buy on day 2 (price = 2) and sell on day 3 (price = 6), profit =
6-2 = 4.
             Then buy on day 5 (price = 0) and sell on day 6 (price = 3),
profit = 3-0 = 3.
```

（from Leetcode）

对于Leetcode 188. Best Time to Buy and Sell Stock IV，我们记录天数i，当前交易次数j，和当前最大盈利（int），可以用二维矩阵dp[i][j]表示。假设第i天股票价格为f(i)，一次股票交易相当于在第a天购入股票，在第b天售出股票，获得利润为f(b) - f(a)（可为负数）。因此，可以推出状态转移公式dp[b][j] = dp[a-1][j-1] + f(b) - f(a)。这个公式仅针对一对一的依赖关系，所以接下来的任务就是把一对一的公式扩展到多对一（通常多对一比较符合直觉，不过一对多也不是不行，欢迎看完多对一解法之后自己想一个一对多解法）。怎么扩展呢？这个扩展不是修改公式本身，而是把a的范围扩大（提示：一对多解法则是把b的范围扩大）。阅读题干可知，a的合法取值范围是0到b-1，这就导致出现了b种a的取值，以及相对应的b个dp[a-1][j-1] + f(b) - f(a)的值。那么又怎么根据这b个值给dp[b][j]赋值呢？再次阅读题干可知，我们需要最大盈利，所以在这b个值中取最大值即可。由此，我们的状态转移公式变成了dp[b][j] = MAX{dp[a-1][j-1] + f(b) - f(a) | 0 ≤ a ≤ b-1}，大功告成。最终dp[k][n]的值为题目答案，其中k为允许交易的次数，n为天数。

403. Frog Jump

Hard 667 78 Favorite Share

A frog is crossing a river. The river is divided into x units and at each unit there may or may not exist a stone. The frog can jump on a stone, but it must not jump into the water.

Given a list of stones' positions (in units) in sorted ascending order, determine if the frog is able to cross the river by landing on the last stone. Initially, the frog is on the first stone and assume the first jump must be 1 unit.

If the frog's last jump was *k* units, then its next jump must be either *k* - 1, *k*, or *k* + 1 units. Note that the frog can only jump in the forward direction.

Note:

- The number of stones is ≥ 2 and is < 1,100.
- Each stone's position will be a non-negative integer < 2^{31}.
- The first stone's position is always 0.

Example 1:

```
[0,1,3,5,6,8,12,17]

There are a total of 8 stones.
The first stone at the 0th unit, second stone at the 1st unit,
third stone at the 3rd unit, and so on...
The last stone at the 17th unit.

Return true. The frog can jump to the last stone by jumping
```

（from Leetcode）

对于Leetcode 403. Frog Jump，我们记录青蛙的当前位置i，步长j，和青蛙在当前位置是否能跳该步长（boolean），可以用二维矩阵dp[i][j]表示。一次蛙跳相当于从位置a跳到位置b，在b-a为k+1，k，或k-1的条件下，其中k为上次蛙跳的长度。因此，可以推出状态转移公式dp[b][k+1] = dp[a][k]，dp[c][k] = dp[a][k]，和dp[d][k-1] = dp[a][k]，其中b - a = k + 1，c - a = k，d - a = k - 1。接下来的任务是把一对一的公式扩展到多对一。有人可能会问为什么一对三不算一对多，因为三是常数，而“多”指的是变量。也许一部分人会觉得公式扩展到一对多比较符合直觉，比如Leetcode Discussion某答案，毕竟原公式是一对三，但这只特立独行的作者选择多对一。状态dp[c][k]取决于三个状态，即dp[a][k-1]，dp[a][k]，和dp[a][k+1]，其中a = c - k。因为只需要一条路到达终点，所以三个状态中任意一个或多个可行即表明新的状态可行。假设可行标为True，则新的状态转移公式为dp[c][k] = OR{dp[c-k][k-1]，dp[c-k][k]，dp[c-k][k+1]}。最终OR{dp[k][n] | k = 所有在位置n可能的步长}的值为题目答案。

通过这些例子也许会发现，动态规划中的状态转移公式本身都是一个套路：

1. 先根据题干找出针对一对一依赖关系的状态转移方式；

2. 把这种方式用语言表达出来，再改写成公式；
3. 把这个公式根据子问题互相之间的依赖关系扩展成一对多或多对一的关系，即如何在“多”的一方取一个值赋给“一”的一方，类似的取法比如上面提到的MAX和OR；
4. 用公式将一个或多个子问题解转换成最终答案。

有了上一章找出最优子结构的方法和这一章找出状态转移公式的方法，动态规划问题最主要的难点也就得以解决了。祝大家多加练习之后遇到动态规划问题即可长舒一口气，并告诉自己：“Haha, I’m God of DP”。如果面试官在场也可以选择告诉TA，分享这份喜悦 :)

没想到这章又没写完就到字数了，那么动态规划的优化思路和记忆化搜索将放到下一章讲解，题目难度将有所提升，敬请期待。

相关材料：
https://leetcode.com/problems/wildcard-matching/
https://leetcode.com/problems/best-time-to-buy-and-sell-stock-iv/
https://leetcode.com/problems/frog-jump/