week9

动态规划、状态转移方程串讲

动态规划的题目，递推的方法最为简化。

初学者先从分治、记忆化搜索开始切入动态规划的题目，再转为递推的问题。

动态规划 和 递归或者分治 没有根本上的区别（关键看有无最优的子结构）

共性：找到重复子问题

差异性：最优子结构、中途可以淘汰次优解

DP顺推模板

```
function DP():
  dp = [][] #二维的情况
  for i = 1 ... M {
      for j = 1 ... N {
          dp[i][j] = _Funciton(dp[i',j']...)
      }
  }
  return dp[M][N];
```

重点难点(多练习，以下是动态规划的内功)：

1.dp = [][]

DP状态的定义需要经验，把现实的问题定义成数组，里面保存状态，数组一维二维三维都可能

2.dp[i][j] = _Funcit on(dp[i',j']...)

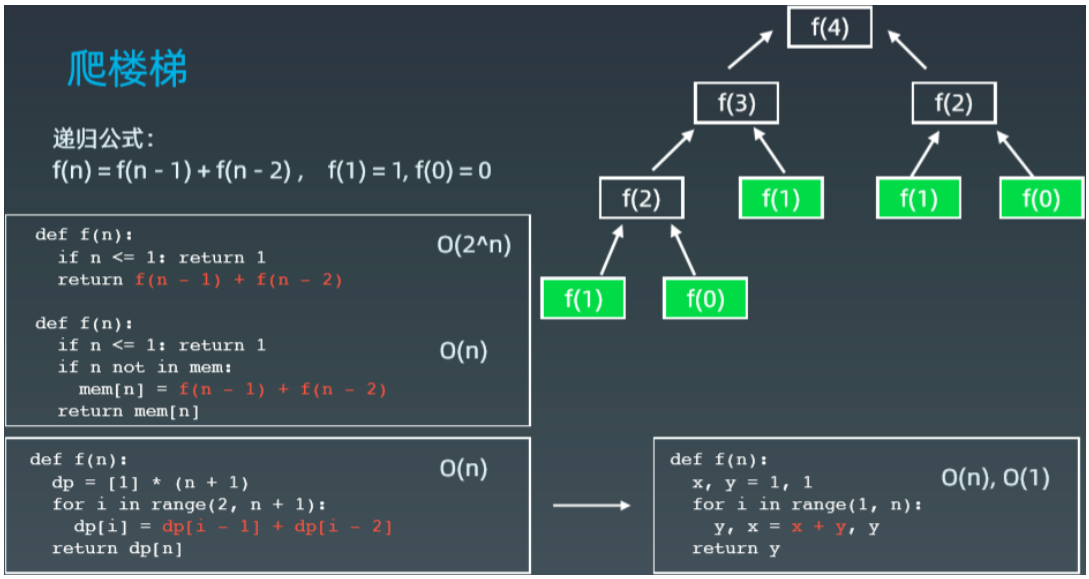
状态转移方程，很多时候要求最值，累加累减；或者在状态转移方程中有小循环，从之前的k个状态中找出最值。

复杂度来源

状态拥有更多维度(二维三维或者更多，甚至需要压缩)一维解决不了马上想到二维

状态方程更加复杂

爬楼梯



拔高题目：

1.可以走数组当中的步伐，x1, x2, ..., xm 步

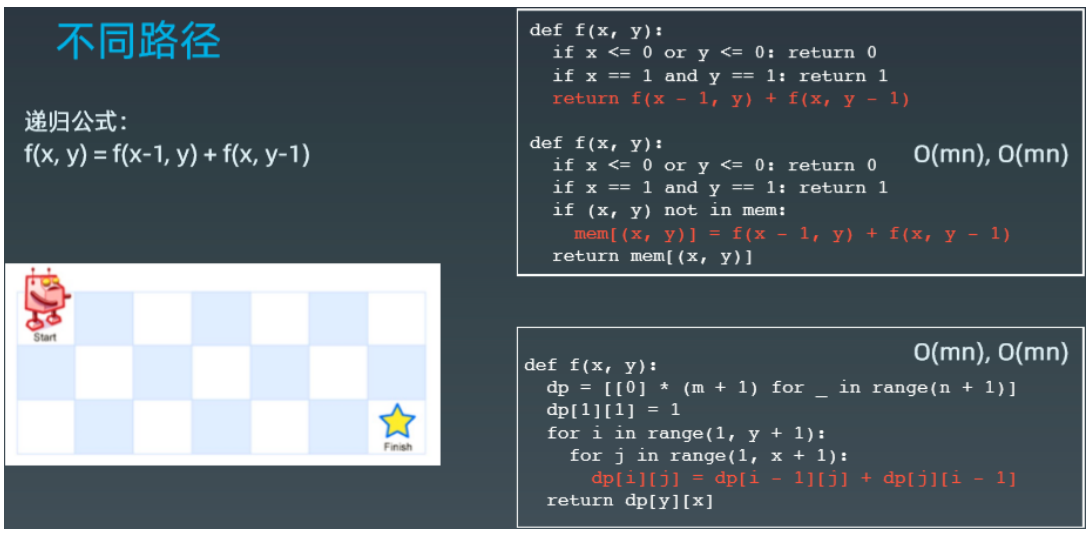
```
for (int i = 2; i < n; ++i)
   for (int j = 0; j < m; ++j)
      a[i] += a[i - x[j]];
```

2.前后不能走相同的步伐

一维不够，需要再加一维

a[i][k] —— i表示上到第几节台阶，k表示当前这一步走的是几步

不同路径



打家劫舍

二维数组的DP在后面较为复杂的动态规划问题中常见，一维数组问题太简单了，面试中肯定会问二维数组的问题。

```
dp[i]状态的定义:  max $ of robbing A[0 -> i]

dp[i] = max(dp[i - 2] + nums[i], dp[i - 1])
```

```
dp[i][0]状态定义: max $ of robbing A[0 -> i] 且没偷 nums[i]
dp[i][1]状态定义: max $ of robbing A[0 -> i] 且偷了 nums[i]


dp[i][0] = max(dp[i - 1][0], dp[i - 1][1]);
dp[i][1] = dp[i - 1][0] + nums[i];
```

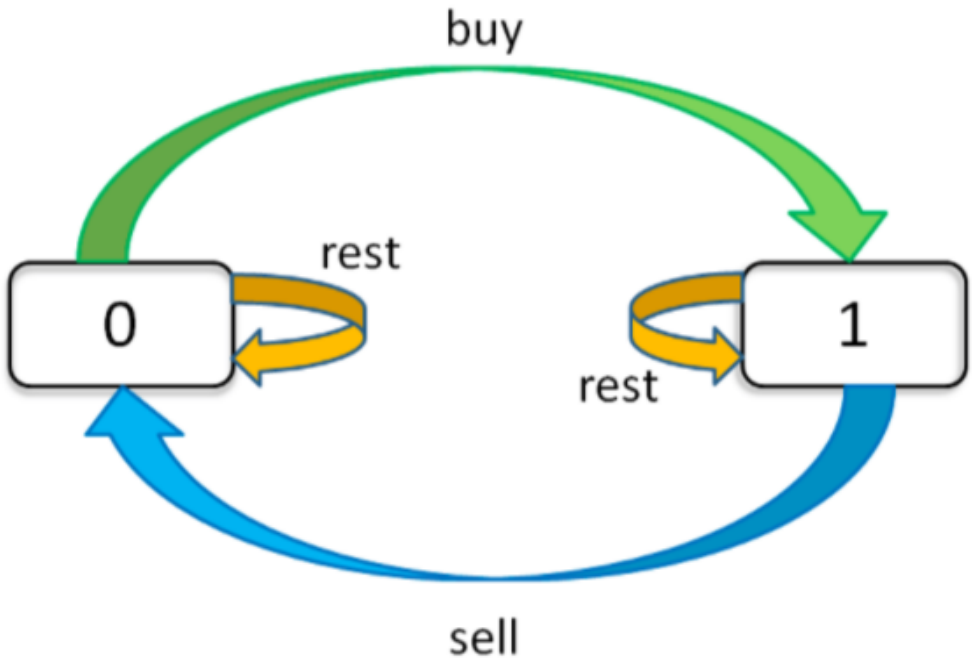
最小路径和

最小路径和问题，和一棵树从上面走下来最短的路径问题一样，都很经典常见。

```
dp[i][j]状态的定义:  minPath(A[1 -> i][1 -> j])


dp[i][j] = min(dp[i - 1][j], dp[i][j - 1]) + A[i][j]
```

买卖股票的最佳时机




```
for 状态1 in 状态1的所有取值:
    for 状态2 in 状态2的所有取值:
        for ...
            dp[状态1][状态2][...] = 择优(选择1, 选择2...)
```

每天都有三种「选择」：买入、卖出、无操作，我们用 buy, sell, rest

问题的「状态」有三个：第一个是天数，第二个是允许交易的最大次数，第三个是当前的持有状态（rest 的状态，1 表示持有，0 表示没有持有）

状态转移方程

dp[i][k][0] = max(dp[i−1][k][0], dp[i−1][k][1] + prices[i])

max(选择 rest , 选择 sell)

解释：今天我没有持有股票，有两种可能：

要么是我昨天就没有持有，然后今天选择 rest，所以我今天还是没有持有；

要么是我昨天持有股票，但是今天我 sell 了，所以我今天没有持有股票了。

dp[i][k][1] = max(dp[i−1][k][1], dp[i−1][k−1][0] − prices[i])

max(选择 rest, 选择 buy)

解释：今天我持有着股票，有两种可能：

要么我昨天就持有着股票，然后今天选择 rest，所以我今天还持有着股票；

要么我昨天本没有持有，但今天我选择 buy，所以今天我就持有股票了。

base case

dp[−1][k][0] = 0

解释：因为 i 是从 0 开始的，所以 i = −1 意味着还没有开始，这时候的利润当然是 0 。

dp[−1][k][1] = −infinity

解释：还没开始的时候，是不可能持有股票的，用负无穷表示这种不可能。

dp[i][0][0] = 0

解释：因为 k 是从 1 开始的，所以 k = 0 意味着根本不允许交易，这时候利润当然是 0 。

dp[i][0][1] = −infinity

解释：不允许交易的情况下，是不可能持有股票的，用负无穷表示这种不可能。

```
base case：
dp[-1][k][0] = dp[i][0][0] = 0
dp[-1][k][1] = dp[i][0][1] = -infinity
状态转移方程：
dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])
dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])
```

编辑距离

1.BFS，最好是双端BFS

2.DP

dp[i][j]：word1[0: i]和word2[0: j]之间的编辑距离

（经验，这一类题目都应该这样做）

case1：

w1 = ...x（长度为 i）

w2 = ...y （长度为 j）

if w1[i] == w2[j]:

    edit_dist(w1, w2) = edit_dist(w1[0: i − 1], w2[0: j − 1])

第一维肯定是w1，第二维肯定是w2，同时肯定是从0开始的，所以可以简写为：

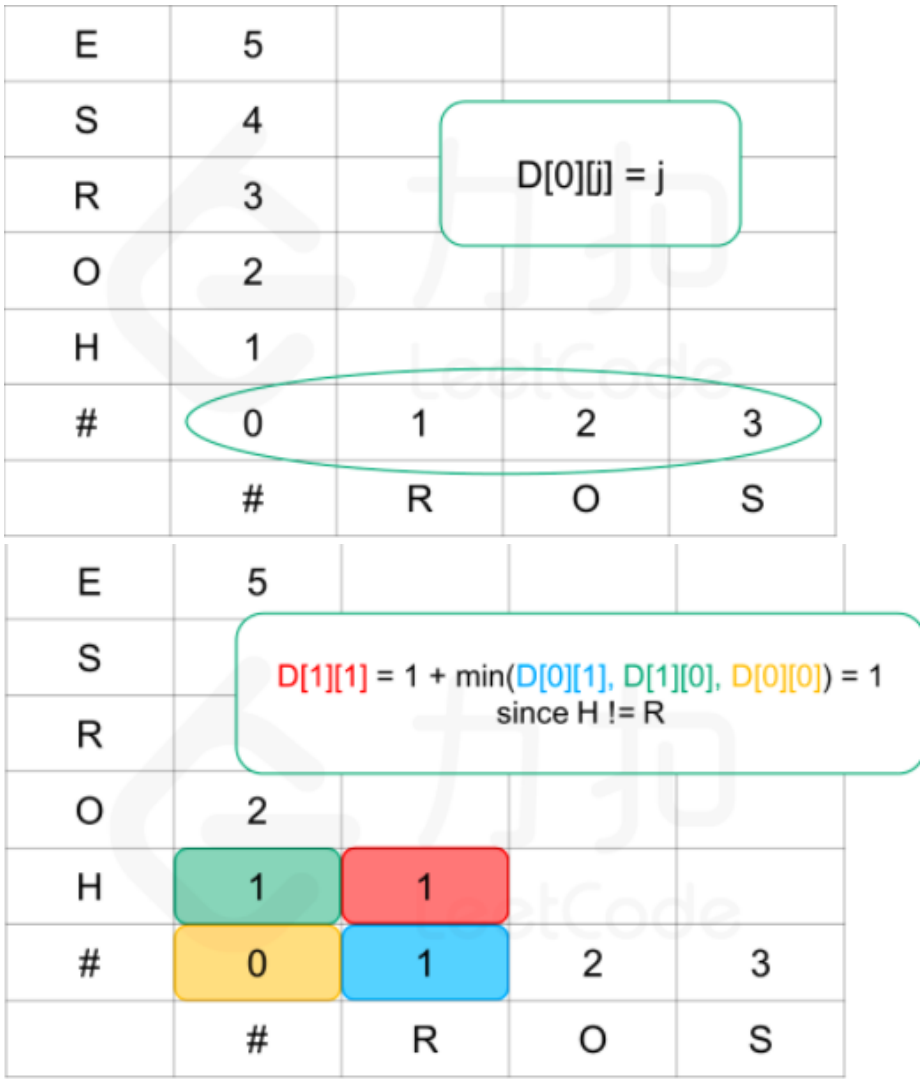
edit_dist(i, j) = edit_dist(i – 1, j – 1) //分治

```
if w1[i] != w2[j]:
    edit_dist(i, j) =
        MIN(edit_dist(i – 1, j – 1) + 1，//把x替换成y，或者把y替换成x，编辑距离+1
            edit_dist(i – 1, j) + 1,      //只删掉x
            edit_dist(i, j – 1) + 1)      //只删掉y
```

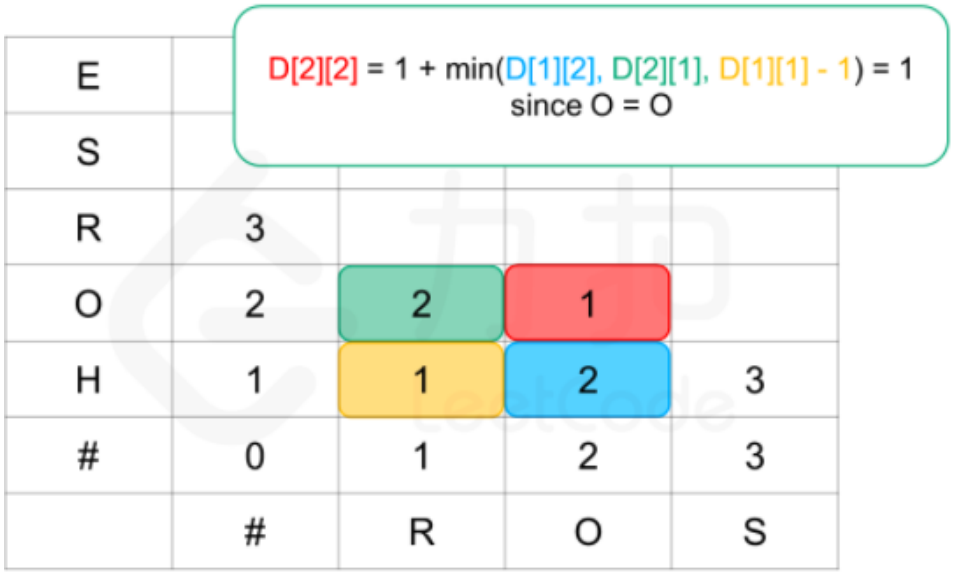
官方题解的动画

https://leetcode-cn.com/problems/edit-distance/solution/bian-ji-ju-chi-by-leetcode-solution/

空字符串和非空字符串的编辑距离等于非空字符串的长度



此处O相同，D[2][2]直接等于D[1][1]



自底向上

```
class Solution:
    def minDistance(self, word1: str, word2: str) -> int:
        n1 = len(word1)
        n2 = len(word2)
        dp = [[0] * (n2 + 1) for _ in range(0, n1 + 1)] #(n1 + 1)行(n2 + 1)列的二维数组
        # 第一行
        for j in range(1, n2 + 1):
            dp[0][j] = dp[0][j - 1] + 1
        # 第一列
        for i in range(1, n1 + 1):
            dp[i][0] = dp[i - 1][0] + 1
        for i in range(1, n1 + 1):
            for j in range(1, n2 + 1):
                if word1[i - 1] == word2[j - 1]:
                    dp[i][j] = dp[i - 1][j - 1]
                    #注意下标，dp给空字符串留了空位，dp和word的下标差一
                else:
                    dp[i][j] = min(dp[i - 1][j - 1] + 1, dp[i - 1][j] + 1, dp[i][j - 1] + 1)
        return dp[n1][n2]
```

自顶向下

```
import functools
class Solution:
    @functools.lru_cache(None)
    def minDistance(self, word1: str, word2:str) -> int:
        if not word1 or not word2:
            return len(word1) + len(word2)
```

```
        if word1[0] == word2[0]:
            return self.minDistance(word1[1:], word2[1:])
        else:
            inserted = 1 + self.minDistance(word1, word2[1:])
            deleted = 1 + self.minDistance(word1[1:], word2)
            replace = 1 + self.minDistance(word1[1:], word2[1:])
            return min(inserted, deleted, replace)
```

Homework

最长上升子序列 官方题解

赛车

最大矩阵 官方题解

字符串算法

字符串及相关算法这部分很重要，面试中高频

字符串基础知识

python、java、Go、JS、C#中字符串是immutable的，string不可变的，加一个字母减一个字母，其实是新生成了一个string，原来的string还是原来的内容。immutable是线程安全的。c++是mutable的。

https://lemire.me/blog/2017/07/07/are-your-strings-immutable/

字符串中第一个唯一的字符

1.brute-force

i – 枚举所有字符

j – 枚举i之后所有字符 //找重复

O(n^2)

2.map(Hashmap 或 Treemap)

Hashmap： O(1)的时间查询是否有重复

Treemap： O(logN)的时间查询是否有重复

O(N) or O(NlogN)

3.用字母对应的下标来统计，ASCII码255位——hash table

```
class Solution {
    public int firstUniqChar(String s) {
        HashMap <Character, Integer> hm = new HashMap();
        //map用来存某个字符出现的频次
        //预处理，把s中的所有字符都统计出来
```

```
        for (int i = 0; i < s.length(); i++) {
            hm.put(s.charAt(i), hm.getOrDefault(s.charAt(i), 0) + 1);
        }
        for (int i = 0; i < s.length(); i++) {
            if (hm.get(s.charAt(i)) == 1) {
                return i;
            }
        }
        return -1;
    }
}
```

字符串转换整数(atoi)

一定一定写的时候要思考清楚每一步在做什么，如果真正出到面试的时候，会发现基本功和练习写小程序的能力非常重要。

1. 去掉前导空格
2. 再是处理正负号
3. 识别数字，注意越界情况。