1. **LCS(最长公共子序列问题)**

（1） LCS(x – 1,  y – 1) + 1如果Ax ＝ By

这对应L(x,y) = L(x,- 1 y- 1)末尾接上Ax

（2.1） LCS(x – 1, y)  如果Ax ≠ By且LCS(x – 1, y) ≥LCS(x, y – 1)

这对应L(x,y)= L(x – 1, y)

（2.2） LCS(x, y – 1)  如果Ax ≠ By且LCS(x – 1, y) <LCS(x, y – 1)

这对应L(x,y) = L(x, y – 1)

（3） 0 如果 x =0或者y = 0

这对应L(x,y)=空序列

1. **最长单增子序列**

Dp[i]表示以a[i]结尾的单调增子序列长度

Dp[i] = max(Dp[i],Dp[j]+1),其中j<I,显然，如果j是一个单调增子序列，且位于i之前，那么Dp[i]至少等于以j为结尾的单调增序列长+1；

1. **最短编辑距离**

这里res[i][j]表示子串word1[0....i-1]和字串word2[0...j-1]的最短编辑距离。当word1[i]和word2[j]相等，那么res[i][j]自然等于res[i-1][j-1]; 当word1[i]和word2[j]不相等时，分三类情况，方法1是删去word1[i],试图让word1[0....i-1]转变为word2[0...j]，那么其最小编辑距离等于res[i-1][j]+1; 方法2是将word2[j]插入到word1[i]后，如果这样能匹配的话，显然word1[0...i]要与word2[0....j-1]匹配，故其最小编辑距离等于res[i][j-1]+1， 方法3是将word1[i]替换为word2[j],这样如果匹配的话，显然word1[0...i-1]要与word2[0...j-1]匹配，故其最小编辑距离等于res[i-1][j-1]+1

1. **字符串匹配(KMP算法)**
2. **最长回文子串**

Dp[i][j]表示区间[i,j]的字串是否回文

Dp[i][i] = true;（单个字符一定是回文串）

先遍历一遍长度为2的情况，即s[j]和s[j+1]如果相等,则dp[j][j+1] = true;

遍历每个可能的回文长度i=3---->size

J表示区间起始，end表示区间结束；

如果s[j] == s[end] 并且dp[j+1][end-1]为true，则dp[j][end]为true，并更新最长回文长度为i，并保存该回文串的起始地址j

**6.单词拆分**

给定一个**非空**字符串 s 和一个包含**非空**单词列表的字典 wordDict，判定 s 是否可以被空格拆分为一个或多个在字典中出现的单词。

Dp[i]表示到字符s的第i个字符为止的字串是否可以被拆分。

bool wordBreak(string s, vector<string>& wordDict) {

vector<bool> dp(s.size()+1,false);

dp[0] = true;

for(int i = 1; i < s.size()+1; ++i){

for(auto str : wordDict){ //每次循环都遍历字典中的所有单词

//如果单前字串长度大于等于单词长度，且该单词又是字串的后缀，则该字串可能被拆分；如果已经拆分好，就永远为true了，否则取决于其前缀是否可以被拆分

if(i >= str.size() && str == s.substr(i-str.size(),str.size())){

dp[i] = dp[i] || dp[i-str.size()];

}

}

}

return dp[s.size()];

}

**7.最大正方形**

在一个由 0 和 1 组成的二维矩阵内，找到只包含 1 的最大正方形，并返回其面积。

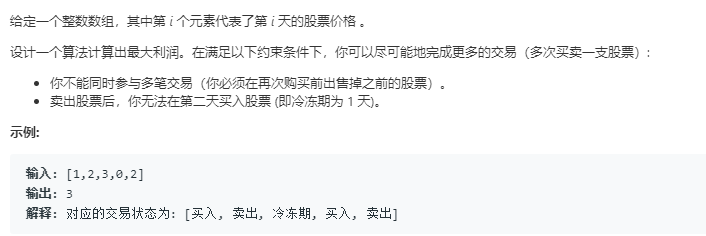


设dp[i][j]表示以二维矩阵的第i行和第j列的点为左下角的最大正方形变长。

那么可以发现，如果该点为1，则最大正方形边长取决于该点左侧、右侧和左上对应的正方形中边长最小的那个+1.

所以dp[i][j] = min(dp[i-1][j],dp[i][j-1],dp[i-1][j-1]);

**8.最佳买卖股票时机含冷冻期**



解:

设定两个数组hold和profits分别表示到第i天为止，持有成本(成本是负数)的最大值(也可以理解为用正数表示的成本的最小值)和获得利润的最大值。目标就是让利润最大

转移方程为

当天数小于等于1时，hold[i] = max(hold[i-1],-prices[1]);

当天数大于1时:

Hold[i] = max(hold[i-1],profits[i-2] – prices[i])—即第i天不买入或者买入，如果不买入，那么第i天的成本就是第i-1天的成本，如果买入，那么由于冷冻期的存在，第i天的成本为到第i-2天为止的最大利润(上一次卖出必然是在i-2天之前，如果是i-1天卖出，第i天就冷冻了)减去当日价格。

Profits[i] = max(profits[i-1],hold[i-1] + prices[i])---即第i天不卖出或者卖出。如果不卖出，那么迄今为止的利润就是前i-1天的最大利润；如果卖出，则迄今为止的利润就是当日卖出价格减去i-1天前的成本(因为成本是负的，所以用加法)；

**普通解法：**

用dp[i]表示第i天前的最大获利情况，则可以得到转移方程

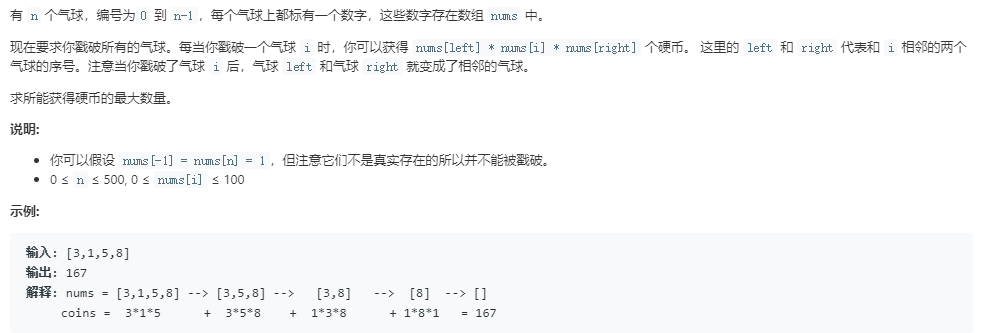
If( j >= 2) dp[i] = max(difference + dp[j-2],dp[i])

Else Dp[i] = max(difference,dp[i]);

其中difference表示第i天的价格与第j天的价格之差---即第j天买入，第i天卖出；



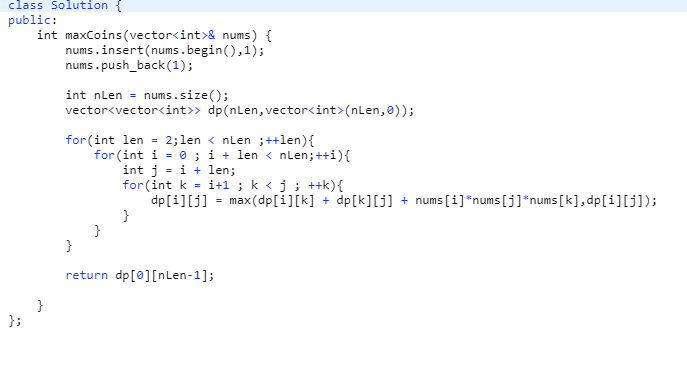
**9.戳气球**



解：

设dp[i][j]表示从i+1----j-1范围内戳气球可以得到的最大回报值。设每次最后一个被戳破的气球标号为k(k的范围是i+1---j-1)那么，转移方程为：

Dp[i][j] = max(Dp[i][j],nums[i]\*nums[j]\*nums[k] + dp[i][k] + dp[k][j])



**10.分割动态子集**



解:

这个问题可以看作,在数组内，是否有任意个元素的和可以恰好等于数组之和的一半

那么，这其实就转化为了一个01背包问题：

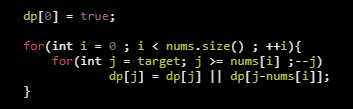
背包中各个物品的价值等于元素的值，重量均为“1“，现在是要从中选取物品使得刚好填满容量为V的背包；

这里用dp[V]表示是否能恰好填满大小为V的背包—也就是能否恰好求得和为target。

转移方程为:

Dp[V] = dp[V] || dp[V – nums[i] ]

且根据01背包的优化方法，对于每一个i，V的遍历顺序应当从V到nums[i],这样，当前需要的状态才由上一时刻决定。

关键代码部分如下：  


**11、不同的子序列**

**题目描述:**



**解:**

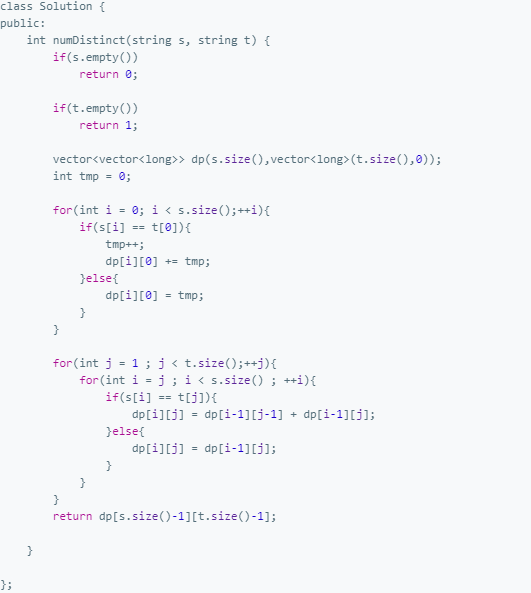
利用动态规划的思想

Dp[i][j]表示到s[i]为止,包含到t[j]为止得字串个数. 则有转移方程:

Dp[i][j] = Dp[i-1][j-1] + Dp[i-1][j] (s[i] == t[j])

Dp[i][j] = Dp[i-1][j] (s[i] != t[j])

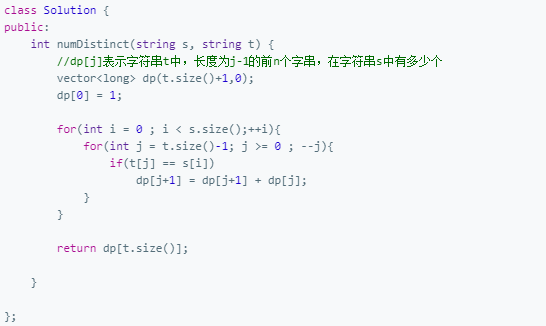
代码如下:



**优化:可以将二维的dp数组降为一维，和背包问题的优化方法类似:**

这里想到这么优化的原因和01背包问题相似. 优化前的转移方程式dp[i][j] = dp[i-1][j-1] + dp[i-1][j]，可以类似这么认为:"i时刻的状态仅仅与i-1时刻有关"。如果我们让i递增，实际上并不需要单独为i开辟空间了。比如，当前状态是dp[j]，而当前时刻为i，即在dp[j]在i时刻还没有更新的时候，dp[j]保存的自然就是上一时刻(i-1)的状态，也就是对应于优化前的dp[i-1][j]啦！ 那么为什么需要倒序呢？仔细想想转移方程dp[j ] = dp[j] + dp[j-1]，其中dp[j]和dp[j-1]都应当是“上一时刻的状态”,如果是顺序的话，这时候的dp[j-1]显然已经更新过了，也就不是“上一时刻”而是“这一时刻”的状态了。如果是倒序的话,这时候dp[j-1]还没有更新，所以也保存的是“上一时刻的状态”！

代码如下



**12、股票问题通解**





