# 刷题日历

|  |  |
| --- | --- |
| 2021年4月25日 | 413等差数列划分(中等) 看题解  64 最小路径和 独立写  542 01矩阵 看题解 有疑惑 |
| 2021年4月26日 | 221 最大正方形  279 完全平方数 完全背包类(滚动数组) |
| 2021年4月27日 | 91 解码方法(中等) 4.22第一次做  139 单词拆分(中等) 不会做 内外循环的优先级  300 最长递增子序列(中等) 熟练 还有新方法:贪心+二分  1143 最长公共子序列(中等) 熟练  416 分割等和子集(中等) 没想到是0-1 并且转移方程有遗漏 与滚动数组混了.  474 一和零(中等) 想到0-1背包 但是不会做 答案有点不确定  322 零钱兑换(中等) 熟练 |
| 2021年4月28日 | 650 只有两个按键的键盘(中等) 不会 |
| 2021年4月29日 | 121 买卖股票的最佳时机(简单) 会  188 买卖股票的最佳时机IV(困难) 不会 大佬教会了  309 买卖股票的最佳时机含冷冻期(中等) 会 |
| 2021年4月30日 | 213 打家劫舍 II 会做 但是官方思路更好  53 最大子序和(简单) 会做  343 整数拆分(中等) 会做  583 两个字符串的删除操作(中等) 会做  646 最长数对链（中等）会做  376 摆动序列(中等) 只想到O(n²) 官方O(n)值得学习  494 目标和(中等) 会做不熟 |
| 2021年5月3日 | 714 买卖股票的最佳时机含手续费 熟练  403 青蛙过河(困难) 会做不熟  44 通配符匹配(困难) 较弱 空间压缩注意初始化的时机 |
|  |  |

# 贪心

# 双指针

# 二分

# 排序

# 搜索

# 六 动态规划

## 题目列表

|  |  |  |  |
| --- | --- | --- | --- |
| 基本动态规划: 一维 | 70 198 413 | 基本动态规划: 二维 |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

## 70 爬楼梯(简单)

dp[i]表示爬到第i阶楼梯的方法数, 最后一步要么跨1步, 要么跨2步, 所以

dp[i] = dp[i–1] + dp[i-2].

显然dp[1]=1, dp[2]=2.

因为每次都只用到前两个结果, 所以可以压缩存储空间.

|  |
| --- |
| class Solution {  public:  int climbStairs(int n) {  if(n < 3) return n;  int a = 1, b = 2, c;  for(int i = 3; i <= n; i++) {  c = a + b;  a = b;  b = c;  }  return c;  }  }; |

## 198 打家劫舍(中等)

题目描述:

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

给定一个代表每个房屋存放金额的非负整数数组，计算你 不触动警报装置的情况下 ，一夜之内能够偷窃到的最高金额。

dp[0][i] 不偷第i家的最大收益

dp[1][i] 偷第i家的最大收益

p[i]第i家的金钱.

对于第i家

(1) 不偷, 上一家可偷可不偷. dp[0][i] = max{dp[0][i - 1], dp[1][i - 1]}.

(2) 偷, 上一家只能不偷. dp[1][i] = dp[0][i - 1] + p[i]

最后返回max{dp[0][size-1], dp[1][size – 1]}

边界:

dp[0][0] = 0;

dp[1][0] = p[0];

|  |
| --- |
| class Solution {  public:  int rob(vector<int>& nums) {  int size = nums.size();  vector<vector<int>> dp(2, vector<int>(size));  dp[0][0] = 0;  dp[1][0] = nums[0];  for(int i = 1; i < size; i++) {  dp[0][i] = max(dp[0][i - 1], dp[1][i - 1]);  dp[1][i] = dp[0][i - 1] + nums[i];  }  return max(dp[0][size - 1], dp[1][size - 1]);  }  }; |

压缩空间

|  |
| --- |
| class Solution {  public:  int rob(vector<int>& nums) {  int a = 0;  int b = nums[0];  for(int i = 1; i < nums.size(); i++) {  int t = a;  a = max(a, b);  b = t + nums[i];  }  return max(a, b);  }  }; |

注意, 在每个循环开始时, 要先用一个临时变量保存a的值.

## 413 等差数列划分(中等)

题目描述:

如果一个数列至少有三个元素，并且任意两个相邻元素之差相同，则称该数列为等差数列。

函数要返回数组 A 中所有为等差数组的子数组个数。

所谓子数组是指A的下标i~j的元素序列(0≤i≤j≤size-1)

A = [1, 2, 3, 4]

返回: 3, A 中有三个子等差数组: [1, 2, 3], [2, 3, 4] 以及自身 [1, 2, 3, 4]。

dp[i] 表示以元素A[i]结尾的等差子数组的数量.

如果A[i]-A[i-1]==A[i-1]-A[i-2]则元素A[i]可以加到前面的序列中形成等差数列, 则dp[i]=dp[i-1]+1, 否则dp[i]=0. 因为等差数列的划分是可以以任意一个元素作为结尾的, 所以最后返回dp[]数组的和.

为什么dp[i]=dp[i-1]+1?

想一下, a b c d ... x是等差数列, 它有n种划法, 等差数列有多少种划分方法与元素的值无关, 只与元素的个数有关.

将y加到原本数列末尾, 形成a b c d ... x y

{b c d ... x y}的个数={a b c d ... x}相等, 所以划分方法也一样, 都是n种.

而{a b c d ... x y}本身也是一种划分, 所以要+1.

|  |
| --- |
| class Solution {  public:  int numberOfArithmeticSlices(vector<int>& nums) {  vector<int> dp(nums.size(), 0);  if(nums.size() < 3)  return 0;  int rslt = 0;  for(int i = 2; i < nums.size(); i++) {  if(nums[i] - nums[i - 1] == nums[i - 1] - nums[i - 2]) {  dp[i] = dp[i - 1] + 1;  rslt += dp[i];  }  }  return rslt;  }  }; |

空间压缩版本

|  |
| --- |
| class Solution {  public:  int numberOfArithmeticSlices(vector<int>& nums) {  int dp = 0;  if(nums.size() < 3)  return 0;  int rslt = 0;  for(int i = 2; i < nums.size(); i++) {  if(nums[i] - nums[i - 1] == nums[i - 1] - nums[i - 2]) {  dp = dp + 1;  rslt += dp;  }  else  dp = 0;  }  return rslt;  }  }; |

## 64 最小路径和

给定一个包含非负整数的 m x n 网格 grid ，请找出一条从左上角到右下角的路径，使得路径上的数字总和为最小。

说明：每次只能向下或者向右移动一步。

dp[i][j]表示到达坐标(i,j)的最小路径长度.

可以从左边或者上边到达坐标(i,j)

dp[i][j]=min(dp[i][j-1], dp[i-1][j])+grid[i][j].

(1)注意要检查坐标的合法性.

(2)要保证dp[i][j-1], dp[i-1][j]都是被计算过

初始时先遍历第一行和第一列,

然后按列遍历完剩下的每一列即可.

这样就不需要考虑坐标的合法性了.

|  |
| --- |
| class Solution {  public:  int minPathSum(vector<vector<int>>& grid) {  int m = grid.size();  int n = grid[0].size();  vector<vector<int>> dp(m, vector<int>(n, 0));  dp[0][0] = grid[0][0];  for(int j = 1; j < n; j++) {  dp[0][j] = dp[0][j - 1] + grid[0][j];  }  for(int i = 1; i < m; i++) {  dp[i][0] = dp[i - 1][0] + grid[i][0];  }  for(int j = 1; j < n; j++) {  for(int i = 1; i < m; i++) {  dp[i][j] = min(dp[i - 1][j], dp[i][j - 1]) + grid[i][j];  }  }  return dp[m - 1][n - 1];  }  }; |

## 542 01矩阵

给定一个由 0 和 1 组成的矩阵，找出每个元素到最近的 0 的距离。

两个相邻元素间的距离为 1 。

一个元素

(1) 如果它是0, 则最近距离为0

(2) 如果它是1, 则最近距离为min(上, 下, 左, 右)+1.

如果用dp[i][j] 如何保证(i,j)的上下左右都是被计算过的?

如果只用一次循环, 是无论如何都做不到某个坐标(i,j)它的上下左右都被计算过.

要找出一个位置从四个方向到达0的最小距离, 可以为四个区域来做

从上方和左方达到的最小距离

从左方和下方到达的最小距离

从下方和右方到达的最小距离

从右方和上方到达的最小距离.

取四个距离的最小者, 就能得到答案. 相当于第64题做4次.

|  |
| --- |
| class Solution {  public:  vector<vector<int>> updateMatrix(vector<vector<int>>& mat) {  int m = mat.size();  int n = mat[0].size();  vector<vector<int>> dp(mat.size(), vector<int>(n, INT\_MAX - 4));  for(int i = 0; i < m; i++) {  for(int j = 0; j < n; j++) {  if(mat[i][j] == 0) {  dp[i][j] = 0;  }  }  }  // 下右  for(int i = 0; i < m; i++) {  for(int j = 0; j < n; j++) {  // dp[i][j] = min(dp[i][j], dp[i - 1][j] + 1, dp[i][j - 1] + 1);  if(i != 0)  dp[i][j] = min(dp[i][j], dp[i - 1][j] + 1);  if(j != 0)  dp[i][j] = min(dp[i][j], dp[i][j - 1] + 1);  }  }  // 上右  for(int i = m - 1; i >= 0; i--) {  for(int j = 0; j < n; j++) {  // dp[i][j] = min(dp[i][j], dp[i + 1][j] + 1, dp[i][j - 1] + 1);  if(i != m - 1)  dp[i][j] = min(dp[i][j], dp[i + 1][j] + 1);  if(j != 0)  dp[i][j] = min(dp[i][j], dp[i][j - 1] + 1);  }  }  // 上左  for(int i = m - 1; i >= 0; i--) {  for(int j = n - 1; j >= 0; j--) {  // dp[i][j] = min(dp[i][j], dp[i + 1][j] + 1, dp[i][j + 1] + 1);  if(i != m - 1)  dp[i][j] = min(dp[i][j], dp[i + 1][j] + 1);  if(j != n - 1)  dp[i][j] = min(dp[i][j], dp[i][j + 1] + 1);  }  }  // 下左  for(int i = 0; i < m; i++) {  for(int j = n - 1; j >=0; j--) {  // dp[i][j] = min(dp[i][j], dp[i - 1][j] + 1, dp[i][j + 1] + 1);  if(i != 0)  dp[i][j] = min(dp[i][j], dp[i - 1][j] + 1);  if(j != n - 1)  dp[i][j] = min(dp[i][j], dp[i][j + 1] + 1);  }  }  return dp;  }  }; |

注意是如何利用两个if语句巧妙的避免了边界行和列的判断.

dp[][]的初始值不要取满INT\_MAX, 只要取一个足够大的数保证+1不会溢出即可.

其实只需要 **下右** 和 **上左** 两个循环即可.

**没想懂**

|  |
| --- |
| class Solution {  public:  vector<vector<int>> updateMatrix(vector<vector<int>>& mat) {  int m = mat.size();  int n = mat[0].size();  vector<vector<int>> dp(mat.size(), vector<int>(n, INT\_MAX - 4));  for(int i = 0; i < m; i++) {  for(int j = 0; j < n; j++) {  if(mat[i][j] == 0) {  dp[i][j] = 0;  }  }  }  // 下右  for(int i = 0; i < m; i++) {  for(int j = 0; j < n; j++) {  // dp[i][j] = min(dp[i][j], dp[i - 1][j] + 1, dp[i][j - 1] + 1);  if(i != 0) {  dp[i][j] = min(dp[i][j], dp[i - 1][j] + 1);  }  if(j != 0) {  dp[i][j] = min(dp[i][j], dp[i][j - 1] + 1);  }  }  }  // 上左  for(int i = m - 1; i >= 0; i--) {  for(int j = n - 1; j >= 0; j--) {  // dp[i][j] = min(dp[i][j], dp[i + 1][j] + 1, dp[i][j + 1] + 1);  if(i != m - 1) {  dp[i][j] = min(dp[i][j], dp[i + 1][j] + 1);  }  if(j != n - 1) {  dp[i][j] = min(dp[i][j], dp[i][j + 1] + 1);  }  }  }  return dp;  }  }; |

至于为什么只需要考虑**下右** 和 **上左,** 下面进行论证.

(1) 一个元素的周围元素, 可以分为左上, 左下, 右下, 右上四个区域. (这些区域是有坐标轴的重叠部分的, 如正左方既属于左上也属于左下)

(2) 某个位置的最终确定值, 一定要**全面考虑**了从四个区域结果来对比出最小值.

但是, 有例外! 比如说顶角的元素, 如右下角, 只需要考虑正左, 正上和左上区域的元素, 因为它其它方向没有元素.

如果按照从左往右, 从上往下(也就是**下右**的方式), 遍历到**最后一个**元素, 这个元素所得到的最小值, 一定是最终的值, 因为它已经全盘考虑了.

(3) 以一个3x3矩阵为例, 我们**看看先下右再上左**的程序执行过程. 矩阵按以下方式标索引:

1 2 3

4 5 6

7 8 9

先下右, 依次遍历1,2,...,9, 到[9]时, 已经得到这个位置的最终值了.

然后上左按9,8,...1的顺序, 位置[9]不用算了, 算位置[8], 在下右遍历时, 元素[8]已经考虑了左上方的元素, 此时, 进行上左遍历, 上左遍历要考虑下方和右方的元素结果, 由于没有下方元素, 所以[8]只要考虑右方[9]的情况, 因为[9]是考虑了除了元素[9]以外其它所有元素得出来的结果, 所以[8]考虑[9]再加上下右遍历已经进行过的工作, 相当于[8]就考虑了全盘的元素. 依次类推, [7], [6],..., 也会考虑了全盘的元素, 所以只需要两轮遍历即可.

## 221 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。

输出4.

dp[i][j]表示以元素[i][j]作为右下角的正方形的面积.

dp[i][j]的值右左边 上边 和 左上的三个元素共同影响.

(1)A[i][j] == 0, dp[i][j]=0;

(2)A[i][j] == 1, dp[i][j] = (sqrt((min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]))+1)²

|  |
| --- |
| class Solution {  public:  int maximalSquare(vector<vector<char>>& matrix) {  int m = matrix.size();  int n = matrix[0].size();  vector<vector<int>> dp(m, vector<int>(n));  int rslt = 0;  for(int i = 0; i < m; i++) {  for(int j = 0; j < n; j++) {  dp[i][j] = 0;  if(matrix[i][j] == '1') {  dp[i][j] = 1;  if(i != 0 && j != 0) {  int left = dp[i][j - 1];  int up = dp[i - 1][j];  int lu = dp[i - 1][j - 1];  int minT = min(min(left, up), lu);  dp[i][j] = sqrt(minT) + 1;  dp[i][j] \*= dp[i][j];  }  }  rslt = max(rslt, dp[i][j]);  }  }  return rslt;  }  }; |

注: 其实可以标记dp为边长.

## 279 完全平方数

题目描述: (相关题目 518零钱兑换II)

给定正整数 n，找到若干个完全平方数（比如 1, 4, 9, 16, ...）使得它们的和等于 n。你需要让组成和的完全平方数的个数最少。

给你一个整数 n ，返回和为 n 的完全平方数的 最少数量 。

完全平方数 是一个整数，其值等于另一个整数的平方；换句话说，其值等于一个整数自乘的积。例如，1、4、9 和 16 都是完全平方数，而 3 和 11 不是。

这道题是一道完全背包的变形.

完全背包: 有n种重量的物品, 物品i的重量为w[i], 价值为c[i], 给定背包的容量V, 求V能容纳的最大物品的价值, 每种物品有无穷多个.

本题相当于, 物品i的重量为w[i]=i², 价值为c[i]=1, 背包容量n, 求在恰恰好装满背包的情况下, 物品的最小价值. w[]的元素从1到不大于n的平方数, 所以w[]的大小为sqrt(n)

dp[i][j]表示在种类1~i中选取物品, 恰好装满容量为j书包的最小价值, 因为c[i]=1, 等价于求最小物品数量.

(1) 如果w[i]>j, 则书包是不可能装下物品i的, 只能从前1~(i-1)种中挑选物品, 故

dp[i][j]=dp[i-1][j];

(2) 如果选择将物品i放入背包里, 因为每样物品有无数种, 则dp[i][j]=dp[i][j-w[i]]+1;

(3) 如果不选择将物品i放入背包里, 则dp[i][j]=dp[i-1][j].

因为题目要求最小的总价值, 所以上面取小者.

**边界: dp[:][0] = 0, dp[0][1:]=+∞**

**边界条件是什么不能与完全背包问题相同, 要针对具体的应用场景具体分析, 只是状态转移可以参考完全背包.**

因为当n为0时, 需要0个数的平方和相加, 当不适用任何一种物品时, 需要无穷多个数转满背包.

|  |
| --- |
| class Solution {  public:  int numSquares(int n){  int m = sqrt(n) + 1;  vector<vector<int>> dp(m, vector<int>(n + 1, INT\_MAX - 4));  // 如物品总类数为0, 背包容量不为0, 则装无穷多件物品  // 背包容量为0, 则0不需要任何平方数相加  for(int i = 0; i < m; i++) {  dp[i][0] = 0;  }  for(int i = 1; i <= m - 1; i++) {  int w = i \* i;  for(int j = 1; j <= n; j++) {  if(j < w) dp[i][j] = dp[i - 1][j];  else {  dp[i][j] = min(dp[i][j - w] + 1, dp[i - 1][j]);  }  }  }  return dp[m - 1][n];  }  }; |

性能: 272ms 8.8MB

滚动数组

每次计算当前dp[i][j]时, 用到正上方和正左方的某个元素, 所以可以压缩存储, 注意子循环按顺序遍历.

|  |
| --- |
| class Solution {  public:  int numSquares(int n){  int m = sqrt(n) + 1;  vector<int> dp(n + 1, INT\_MAX - 4);    dp[0] = 0;  for(int i = 1; i <= m - 1; i++) {  int w = i \* i;  for(int j = w; j <= n; j++) {  dp[j] = min(dp[j - w] + 1, dp[j]);  }  }  return dp[n];  }  }; |

性能: 224ms 8.9MB

滚动数组用法:

计算dp[i][j]时

(1) 用到左上和上方元素, 子循环倒着遍历.

(2) 用到左方和上方元素, 子循环顺着遍历.

(3) 用到左方和上方和左上方元素, 使用pre数组保存上一次的计算结果, 子循环顺着遍历.

## 91 解码方法(中等)

题目描述:

一条包含字母 A-Z 的消息通过以下映射进行了 编码 ：

'A' -> 1

'B' -> 2

...

'Z' -> 26

要 解码 已编码的消息，所有数字必须基于上述映射的方法，反向映射回字母（可能有多种方法）。例如，"11106" 可以映射为：

"AAJF" ，将消息分组为 (1 1 10 6)

"KJF" ，将消息分组为 (11 10 6)

注意，消息不能分组为  (1 11 06) ，因为 "06" 不能映射为 "F" ，这是由于 "6" 和 "06" 在映射中并不等价。

给你一个只含数字的 非空 字符串 s ，请计算并返回 解码 方法的 总数 。

题目数据保证答案肯定是一个 32 位 的整数.

dp[i]表示使用字符串s的前i个字符的解码方法数量.

s[i-2]是倒数第二个字符, 记为x.

s[i-1]是倒数第一个字符, 记为y.

x和y拼接形成的字符串记为xy

(1) x=='0' && y!='0', dp[i]=dp[i-1].

(2) x=='0' && y=='0', return 0.

(2) x!='0' && y=='0' && xy≤"26", dp[i]=dp[i-2].

(3) x!='0' && y=='0' && xy＞"26", return 0.

(5) x!='0' && y!='0' && xy≤"26", dp[i]=dp[i-2]+dp[i-1].

(6) x!='0' && y!='0' && xy>"26", dp[i]=dp[i-1].

边界条件: dp[0]=1, dp[1]=1

特判: 以0开头的字符串直接返回0.

|  |
| --- |
| class Solution {  public:  int numDecodings(string s) {  int len = s.size();  if(s[0] == '0')  return 0;  vector<int> dp(len + 1, 1);  for(int i = 2; i < len + 1; i++) {  char x = s[i-2], y = s[i - 1];  string xy = s.substr(i - 2, 2);  if(x == '0' && y != '0')  dp[i] = dp[i - 1];  else if(x == '0' && y == '0')  return 0;  else if(x != '0' && y == '0') {  if(xy <= "26")  dp[i] = dp[i - 2];  else  return 0;  }  else if(x != '0' && y != '0') {  if(xy <= "26")  dp[i] = dp[i - 2] + dp[i - 1];  else  dp[i] = dp[i - 1];  }  }  return dp[len];  }  }; |

还可以压缩空间.

## 139 单词拆分(中等)

题目描述:

给定一个非空字符串 s 和一个包含非空单词的列表 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。

说明：

拆分时可以重复使用字典中的单词。

你可以假设字典中没有重复的单词。

题目理解:

用空格拆分s以后, 出现的单词必须属于wordDict集合.

先对wordDict按单词长度排序.

dp[i] 表示字符串s的前i个字符是否可以由wordDict集合中的单词组成.

对于word in wordDict, word长度为wLen,

如果最后几个字符与word重合且dp[i-wLen]为true, 则dp[i]=true

边界: dp[0]=true, 其余为false;

|  |
| --- |
| class Solution {  public:  bool wordBreak(string s, vector<string>& wordDict) {  vector<bool> dp(s.size() + 1, false);  dp[0] = true;  for(int i = 1; i <= s.size(); i++) {  for(int j = 0; j < wordDict.size(); j++) { // 遍历每一个单词  int wLen = wordDict[j].size();  if(i < wLen) continue;  // 一定要加这个判断  if(dp[i - wLen] && s.substr(i - wLen, wLen) == wordDict[j]) {  dp[i] = true;  break;  }  }  }  return dp[s.size()];  }  }; |

为什么一定要加判断:

尾巴的几个字符能和某个单词匹配, 不等于去掉尾巴这几个字符以后的字符串拆分单词集合中的单词.

## 300 最长递增子序列(中等)

题目描述:

给你一个整数数组 nums ，找到其中最长严格递增子序列的长度。

子序列是由数组派生而来的序列，删除（或不删除）数组中的元素而不改变其余元素的顺序。例如，[3,6,2,7] 是数组 [0,3,1,6,2,2,7] 的子序列。

dp[j]表示以元素A[j]结尾的最长严格递增子序列的长度.

(1) 对于i<j, 如果存在A[i]<A[j], 则将A[j]加在旧序列后面, 长度增加1.

(2) 对于i<j, 所有的A[i]≥A[j], 则A[j]自己形成新的序列, 长度为1.

因为要求最长, 所以dp[j]取上述结果的最大者.

边界: 任何元素都可以自己形成长度为1的严格递增子序列, 所以dp[]=1

|  |
| --- |
| class Solution {  public:  int lengthOfLIS(vector<int>& nums) {  vector<int> dp(nums.size(), 1);  int rslt = 0;  for(int j = 0; j < nums.size(); j++) {  for(int i = 0; i < j; i++) {  if(nums[i] < nums[j]) {  dp[j] = max(dp[j], dp[i] + 1);  }  }  rslt = max(rslt, dp[j]);  }  return rslt;  }  }; |

## 1143 最长公共子序列(中等)

题目描述:

给定两个字符串 text1 和 text2，返回这两个字符串的最长 公共子序列 的长度。如果不存在 公共子序列 ，返回 0 。

一个字符串的 子序列 是指这样一个新的字符串：它是由原字符串在不改变字符的相对顺序的情况下删除某些字符（也可以不删除任何字符）后组成的新字符串。

例如，"ace" 是 "abcde" 的子序列，但 "aec" 不是 "abcde" 的子序列。

两个字符串的 公共子序列 是这两个字符串所共同拥有的子序列。

dp[i][j]表示字符串text1的前i个字符和字符串text2的前j个字符的最长公共子序列的长度.

最后一个字符是text1[i-1]和text[j-1]

(1) text1[i-1]==text[j-1], dp[i][j]=max(dp[i-1][j-1])+1;

(2) text1[i-1]!=text[j-1], dp[i][j]=max(dp[i-1][j], d[i][j-1]);

动态保存最大值: rslt=max(rslt, dp[i][j]);

边界: dp[][0]=0, dp[0][]=0;

|  |
| --- |
| class Solution {  public:  int longestCommonSubsequence(string text1, string text2) {  vector<vector<int>> dp(text1.size() + 1, vector<int>(text2.size() + 1, 0));  int rslt = 0;  for(int i = 1; i <= text1.size(); i++) {  for(int j = 1; j <= text2.size(); j++) {  if(text1[i - 1] == text2[j - 1])  dp[i][j] = dp[i - 1][j - 1] + 1;  else  dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);  rslt = max(rslt, dp[i][j]);  }  }  return rslt;  }  }; |

空间压缩

使用一个两行的数组, 轮番使用.

因为原本代码中计算dp[i][j]时同时用到了左上 上方和左上方的元素, 所以子循环要顺序遍历.

|  |
| --- |
| class Solution {  public:  int longestCommonSubsequence(string text1, string text2) {  vector<vector<int>> dp(2, vector<int>(text2.size() + 1, 0));  int rslt = 0;  for(int i = 1; i <= text1.size(); i++) {  int pre = (i & 1) ? 0 : 1;  int cur = (i & 1) ? 1 : 0;  for(int j = 1; j <= text2.size(); j++) {  if(text1[i - 1] == text2[j - 1])  dp[cur][j] = dp[pre][j - 1] + 1;  else  dp[cur][j] = max(dp[pre][j], dp[cur][j - 1]);  rslt = max(rslt, dp[cur][j]);  }  }  return rslt;  }  }; |

## 416 分割等和子集(中等)

题目描述:

给你一个 只包含正整数 的 非空 数组 nums 。请你判断是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。

0-1背包的变形

对数组求和sum, 若sum为奇数, 返回false;

dp[i][j]用数组的前i个数, 能否求和为j

前i个数的最后一个数是nums[i-1]

(1) 使用nums[i-1], 要看dp[i-1][j-nums[i-1]]

(2) 不使用nums[[i-1], 要看dp[i][j]=dp[i-1][j]

dp[i][j] = dp[i-1][j-nums[i-1]] || dp[i][j]=dp[i-1][j]

dp[0][0]=true

dp[0][1:]=false

dp[1:][0]=true 总可以构成0和

|  |
| --- |
| class Solution {  public:  bool canPartition(vector<int>& nums) {  int sum = 0;  for(auto x : nums)  sum += x;  if(sum & 1) return false;  sum /= 2;  vector<vector<bool>> dp(nums.size() + 1, vector<bool>(sum + 1, false));  dp[0][0] = true;  for(int i = 1; i <= nums.size(); i++) {  dp[i][0] = true;  for(int j = 1; j <= sum; j++) {  **if(j < nums[i - 1])**  **dp[i][j] = dp[i - 1][j];**  else  dp[i][j] = dp[i - 1][j - nums[i - 1]] || dp[i - 1][j];  }  }  return dp[nums.size()][sum];  }  }; |

红色部分漏写了, 导致出错.

## 474 一和零(中等)

题目描述:

给你一个二进制字符串数组 strs 和两个整数 m 和 n 。

请你找出并返回 strs 的最大子集的大小，该子集中 最多 有 m 个 0 和 n 个 1 。

如果 x 的所有元素也是 y 的元素，集合 x 是集合 y 的 子集 。

对于字符数组中的每个字符串, 记录它的0的个数和1的个数, 存在cnt[2][x]中.

cnt[0][x]表示字符串strs[x]中'0'的个数.

cnt[1][x]表示字符串strs[x]中'1'的个数.

dp[i][j]表示用i个0和j个i能够拼出的字符串的最大数目.

对于strs[k], dp[i-cnt[0][k]][j-cnt[1][k]]+1

从i=n, j=m倒着尝试

对于每个strs[k]

如果选择凑它, 则数目=1+dp[i-cnt[0][k]][j-cnt[1][k]]

如果选择不凑它, 则数目=dp[i][j]

边界: 一开始可以理解为凑空串, dp[][]=0

|  |
| --- |
| class Solution {  public:  int findMaxForm(vector<string>& strs, int m, int n) {  vector<vector<int>> cnt(2, vector<int>(strs.size(), 0));  for(int i = 0; i < strs.size(); i++) {  for(auto c : strs[i]) {  if(c == '0') cnt[0][i]++;  else cnt[1][i]++;  }  }  vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));  for(int k = 0; k < strs.size(); k++) {  int cnt0 = cnt[0][k];  int cnt1 = cnt[1][k];  for(int i = m; i >= cnt0; i--) {  for(int j = n; j >= cnt1; j--) {  dp[i][j] = max(1 + dp[i - cnt0][j - cnt1], dp[i][j]);  }  }  }  return dp[m][n];  }  }; |

## 322 零钱兑换(中等)

题目描述:

给定不同面额的硬币 coins 和一个总金额 amount。编写一个函数来计算可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。

相似题目: 518 零钱兑换II(中等) 求方案数.

这是完全背包问题.

dp[i][j]表示用前i种金币凑成金额j的最少硬币个数.

最后一种硬币是coins[i-1]

(1) 使用coins[i-1], 则硬币个数为1+dp[i][j-coins[i-1]].

(2) 不使用coins[i-1], 则硬币个数为dp[i-1][j].

二者取其小, dp[i][j]=min(dp[i-1][j], 1+dp[i][j-coins[i-1]]);

边界dp[0][1:]=+无穷, dp[][0]=0;

|  |
| --- |
| class Solution {  public:  int coinChange(vector<int>& coins, int amount) {  int m = coins.size();  int n = amount;  vector<vector<int>> dp(m + 1, vector<int>(n + 1, INT\_MAX - 2));  for(int i = 1; i <= m; i++) {  dp[i][0] = 0;  for(int j = 1; j <= n; j++) {  if(j < coins[i - 1])  dp[i][j] = dp[i - 1][j];  else  dp[i][j] = min(1 + dp[i][j - coins[i - 1]], dp[i - 1][j]);  }  }  return dp[m][n] >= INT\_MAX - 4 ? -1 : dp[m][n];  }  }; |

最后要如果没有办法凑成总金额, 要返回-1.

可以尝试以下压缩空间.

## 72 编辑距离(困难)

## 650 只有两个键的键盘(中等)

题目描述:

最初在一个记事本上只有一个字符 'A'。你每次可以对这个记事本进行两种操作：

Copy All (复制全部) : 你可以复制这个记事本中的所有字符(部分的复制是不允许的)。

Paste (粘贴) : 你可以粘贴你上一次复制的字符。

给定一个数字 n 。你需要使用最少的操作次数，在记事本中打印出恰好 n 个 'A'。输出能够打印出 n 个 'A' 的最少操作次数。

输入: 3

输出: 3

解释:

最初, 我们只有一个字符 'A'。

第 1 步, 我们使用 Copy All 操作。

第 2 步, 我们使用 Paste 操作来获得 'AA'。

第 3 步, 我们使用 Paste 操作来获得 'AAA'。

思路:

dp[i]表示得到i个字符的最少操作次数.

最终得到一个字符串, 最后一序列的操作一定是CPP..(中间都是P)..P, C后面至少一次P.

如果C之前有x个字符, 则一序列操作以后有y=x+nx个字符, 其中n是P的次数.

也就是说从状态x到达状态y要经历1+n次操作, 1是C这个操作

所以dp[y]=dp[x]+1+n, 又n=y/x-1, 所以dp[y] = dp[x]+y/x

注意dp要选最小.

对于某个y, 只能遍历y的因子, 即y%x==0

边界dp[1]=0.

|  |
| --- |
| class Solution {  public:  int minSteps(int n) {  vector<int> dp(n + 1, INT\_MAX);  dp[1] = 0;  for(int i = 2; i <= n; i++) {  for(int j = 1; j <= i / 2; j++) {  if(i % j == 0)  dp[i] = min(dp[i], dp[j] + i / j);  }  }  return dp[n];  }  }; |

O(n²)

下面还有一种O(n\*sqrt(n))的代码

|  |
| --- |
| class Solution {  public:  int minSteps(int n) {  vector<int> dp(n + 1, 0);  for (int i = 2; i <= n; i++)  {  dp[i] = i;  /\* 这里的j不需要写成j <= i，因为判断j是不是因数只要检查到开平方的大小就好了，算是个小优化> \*/  for (int j = 2; j \* j <= i; j++)  {  if (i % j == 0)  dp[i] = dp[j] + dp[i / j];  }  }  return dp[n];  }  }; |

因为从

AA 变为AAAAAA和

从

A变为AAA是一样的

得到j个A以后, 将这j个A看作一个整体Z, i个A就是i/j个Z

所以从j个A变为i个A和从1个Z变成i/j个Z是一样的.

所以dp[i] = dp[j] + dp[i / j];

至于为什么不用min(), 从数学上我也没有想懂.

## 10 正则表达式匹配(困难)

## 121 买卖股票的最佳时机(简单)

题目描述:

给定一个数组 prices ，它的第 i 个元素 prices[i] 表示一支给定股票第 i 天的价格。

你只能选择 某一天 买入这只股票，并选择在 未来的某一个不同的日子 卖出该股票。设计一个算法来计算你所能获取的最大利润。

返回你可以从这笔交易中获取的最大利润。如果你不能获取任何利润，返回 0 。

一个不是动态规划的版本, 用这种方法可以找到历史最低点和历史最高点的差值, 并且满足最低点在前, 最高点在后.

|  |
| --- |
| class Solution {  public:  int maxProfit(vector<int>& prices) {  int minp = prices[0];  int rslt = 0;  for(int i = 0; i < prices.size(); i++) {  if(minp > prices[i]) {  minp = prices[i];  }  rslt = max(rslt, prices[i] - minp);  }  return rslt;  }  }; |

好吧, 这就是动态规划.

## 188 买卖股票的最佳时机IV(困难)

题目描述:

给定一个整数数组 prices ，它的第 i 个元素 prices[i] 是一支给定的股票在第 i 天的价格。

设计一个算法来计算你所能获取的最大利润。你最多可以完成 k 笔交易。

注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

如果将连续递增序列按差值从大到小排序, 选择前k个是不是就可以了?

应该不是, 没有证明.

dp[i][j] 前i天(i从0到n)总共进行了j次操作的最大利润(j从0到2k都可)

(1) j是奇数 肯定是买入或不操作 max dp[i-1][j] dp[i-1][j-1]-p[i]

(2) j是偶数 肯定是卖出或不操作 max dp[i-1][j] dp[i-1][j-1]+p[i]

遍历dp[n-1][:] 返回最大

dp[i][0]=0;

|  |
| --- |
| class Solution {  public:  int maxProfit(int k, vector<int>& p) {  int n = p.size();  vector<vector<int>> dp(n + 1, vector<int>(2 \* k + 1, INT\_MIN / 2));  for(int i = 0; i <= n; ++i) {  dp[i][0] = 0;  }  for(int i = 1; i <= n; ++i) {  for(int j = 1; j <= i + 1; ++j) {  if(j > 2 \* k) break;  if(j & 1) { // 奇数  dp[i][j] = max(dp[i-1][j], dp[i-1][j-1]-p[i-1]);  }  else { // 偶数  dp[i][j] = max(dp[i-1][j], dp[i-1][j-1]+p[i-1]);  }  }  }  int rslt = INT\_MIN;  for(int j = 0; j <= 2 \* k; ++j) {  rslt = max(rslt, dp[n][j]);  }  return rslt;  }  }; |

16ms 12.1 MB

滚动数组

|  |
| --- |
| class Solution {  public:  int maxProfit(int k, vector<int>& p) {  int n = p.size();  vector<vector<int>> dp(2, vector<int>(2 \* k + 1, INT\_MIN / 2));  for(int i = 0; i < 2; ++i) {  dp[i][0] = 0;  }  for(int i = 1; i <= n; ++i) {  int pre = i & 1 ? 0 : 1;  int cur = i & 1 ? 1 : 0;  for(int j = 1; j <= i + 1; ++j) {  if(j > 2 \* k) break;  if(j & 1) { // 奇数  dp[cur][j] = max(dp[pre][j], dp[pre][j-1]-p[i-1]);  }  else { // 偶数  dp[cur][j] = max(dp[pre][j], dp[pre][j-1]+p[i-1]);  }  }  }  int rslt = INT\_MIN;  for(int j = 0; j <= 2 \* k; ++j) {  rslt = max(rslt, dp[n & 1][j]);  }  return rslt;  }  }; |

8 ms 10.6 MB

空间压缩

|  |
| --- |
| class Solution {  public:  int maxProfit(int k, vector<int>& p) {  int n = p.size();  // vector<vector<int>> dp(2, vector<int>(2 \* k + 1, INT\_MIN / 2));  vector<int> dp(2 \* k + 1, INT\_MIN / 2);  dp[0] = 0;  for(int i = 1; i <= n; ++i) {  for(int j = min(2 \* k, i + 1); j >= 1; --j) {  if(j & 1) { // 奇数  dp[j] = max(dp[j], dp[j-1]-p[i-1]);  }  else { // 偶数  dp[j] = max(dp[j], dp[j-1]+p[i-1]);  }  }  }  int rslt = INT\_MIN;  for(int j = 0; j <= 2 \* k; ++j) {  rslt = max(rslt, dp[j]);  }  return rslt;  }  }; |

这里子循环是要倒着遍历的, 之前写法是

|  |
| --- |
| for(int i = 1; i <= n; ++i) {  for(int j = i + 1; j >= 1; --j) {  if(j > 2 \* k) break; |

这总写法是不对的, break应该写成continue, 或者这样写

|  |
| --- |
| for(int i = 1; i <= n; ++i) {  for(int j = min(2 \* k, i + 1); j >= 1; --j) { |

## 309 买卖股票的最佳时机含冷冻期(中等)

题目描述:

给定一个整数数组，其中第 i 个元素代表了第 i 天的股票价格 。​

设计一个算法计算出最大利润。在满足以下约束条件下，你可以尽可能地完成更多的交易（多次买卖一支股票）:

你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

卖出股票后，你无法在第二天买入股票 (即冷冻期为 1 天)。

dp[0][i] 第i天末尾持有股票的最大收益

dp[1][i] 第i天末尾不持有股票的最大收益

dp[0][i]

(1) 前一天是冷冻期, 收益为dp[1][i-2]-prices[i]

(2) 前一天不是冷冻期, 收益为max(-prices[i], dp[0][i-1]), 当天买入或前面就有股票.

dp[0][i]三者取最大

dp[1][i]

(1) 可以是今天卖股票 dp[0][i-1]+prices[i]

(2) 也可以是本来就有股票 dp[1][i-1]

二者取最大

边界:

dp[0][0] = -prices[0];

dp[0][1] = max(-prices[1], dp[0][0]);

dp[1][0] = 0;

dp[1][1] = max(dp[0][0] + prices[1], dp[1][0]);

|  |
| --- |
| class Solution {  public:  int maxProfit(vector<int>& prices) {  int n = prices.size();  if(n < 2)  return 0;  vector<vector<int>> dp(2, vector<int>(n));  dp[0][0] = -prices[0];  dp[0][1] = max(-prices[1], dp[0][0]);  dp[1][0] = 0;  dp[1][1] = max(dp[0][0] + prices[1], dp[1][0]);  for(int i = 2; i < n; ++i) {  dp[0][i] = max(dp[1][i-2]-prices[i], max(-prices[i], dp[0][i-1]));  dp[1][i] = max(dp[0][i-1]+prices[i], dp[1][i-1]);  }  return dp[1][n - 1];  }  }; |

0ms 11MB

## 213 打家劫舍 II(中等)

题目描述:

你是一个专业的小偷，计划偷窃沿街的房屋，每间房内都藏有一定的现金。这个地方所有的房屋都 围成一圈 ，这意味着第一个房屋和最后一个房屋是紧挨着的。同时，相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警 。

给定一个代表每个房屋存放金额的非负整数数组，计算你 在不触动警报装置的情况下 ，今晚能够偷窃到的最高金额。

对于所以偷窃方案, 可以分类两类

(1) 偷第0家(意味着一定不能偷第n-1家)

(2) 不偷第0家

dp0[0][i] 偷第0家, 且偷第i家的最大收益(1≤i≤n-2)

dp0[1][i] 偷第0家, 且不偷第i家的最大收益(1≤i≤n-2)

dp0[0][i]

(1) 前一家只能不偷, dp0[1][i-1]+nums[i]

dp0[1][i]

(1) 前一家不偷 dp0[1][i-1]

(2) 前一家偷了 dp0[0][i-1]

dp0[1][i] = max(dp0[1][i-1], dp0[0][i-1]);

边界

dp0[0][1] = -∞

dp0[1][1] = nums[0]

i从2到n-2遍历

dp1[0][i] 不偷第0家, 且偷第i家的最大收益(1≤i≤n-1).

dp1[1][i] 不偷第0家, 且不偷第i家的最大收益(1≤i≤n-1).

dp1[0][i]

(1) 前一家只能不偷 dp1[1][i-1] + nums[i]

dp1[1][i]

(1) 前一家不偷 dp1[1][i-1]

(2) 前一家偷了 dp1[0][i-1]

dp1[1][i] = max(dp1[1][i-1], dp1[0][i-1]);

边界

dp1[0][1] = nums[1]

dp1[1][1] = 0

i从2到n-1遍历

最后返回max(dp0[0][n-2], dp0[1][n-2], dp1[0][n-1], dp1[1][n-1]);

|  |
| --- |
| class Solution {  public:  int rob(vector<int>& nums) {  int n = nums.size();  if(n == 1) return nums[0];  if(n == 2) return max(nums[0], nums[1]);    vector<vector<int>> dp0(2, vector<int>(n)); // 偷第0家的最大收益  dp0[0][1] = INT\_MIN / 2;  dp0[1][1] = nums[0];  vector<vector<int>> dp1(2, vector<int>(n)); // 不偷第0家的最大收益  dp1[0][1] = nums[1];  dp1[1][1] = 0;  // 处理dp0 (偷第0家)  int rslt = 0;  for(int i = 2; i <= n - 2; ++i) {  dp0[0][i] = dp0[1][i - 1] + nums[i];  dp0[1][i] = max(dp0[1][i - 1], dp0[0][i - 1]);  }  rslt = max(dp0[0][n - 2], dp0[1][n - 2]);  // 处理dp1 (不偷第0家)  for(int i = 2; i <= n - 1; ++i) {  // 偷第i家  dp1[0][i] = dp1[1][i - 1] + nums[i];  // 不偷第i家  dp1[1][i] = max(dp1[1][i - 1], dp1[0][i - 1]);  }  rslt = max(rslt, dp1[0][n - 1]);  rslt = max(rslt, dp1[1][n - 1]);  return rslt;  }  }; |

官方题解思路

dp[i] 表示处理到第i家可以得到的最大金额.

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

## 53 最大子序和(简单)

给定一个整数数组 nums ，找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

dp[i]表示以nums[i]结尾的连续子数组的最大和

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

|  |
| --- |
| class Solution {  public:  int maxSubArray(vector<int>& nums) {  int n = nums.size();  vector<int> dp(n);  dp[0] = nums[0];  int rslt = dp[0];  for(int i = 1; i < n; ++i) {  dp[i] = max(nums[i], nums[i] + dp[i - 1]);  rslt = max(rslt, dp[i]);  }  return rslt;  }  }; |

## 343 整数拆分(中等)

题目描述:

给定一个正整数 n，将其拆分为至少两个正整数的和，并使这些整数的乘积最大化。 返回你可以获得的最大乘积。

有点像完全背包

从1~n-1, 背包容量为n

dp[i][j] 第0到第i个数中选择恰好装满容量为j的背包的最大乘积.

(1) 选择数字i, dp[i][j] = dp[i][j-i] \* i

(2) 不选择数字i, dp[i][j] = dp[i-1][j]

dp[i][j] = max(dp[i][j-i] \* i, dp[i-1][j]);

最后是dp[1:n-1][n]中的最大值.

边界: dp[1][:] = 1

|  |
| --- |
| class Solution {  public:  int integerBreak(int n) {  vector<vector<int>> dp(n, vector<int>(n + 1, 1));  for(int i = 2; i <= n - 1; ++i) {  for(int j = 1; j <= n; ++j) {  if(j < i) dp[i][j] = dp[i - 1][j];  else {  dp[i][j] = max(dp[i - 1][j], dp[i][j - i] \* i);  }  }  }  int rslt = 0;  for(int i = 1; i <= n - 1; ++i) {  rslt = max(rslt, dp[i][n]);  }  return rslt;  }  }; |

0ms 6.6MB

空间压缩

|  |
| --- |
| class Solution {  public:  int integerBreak(int n) {  vector<int> dp(n + 1, 1);  int rslt = 1;  for(int i = 2; i <= n - 1; ++i) {  for(int j = i; j <= n; ++j) {  dp[j] = max(dp[j], dp[j - i] \* i);  }  rslt = max(rslt, dp[n]);    }  return rslt;  }  }; |

最大值的寻取没必要用一个独立的循环来找.

## 583 两个字符串的删除操作(中等)

题目描述:

给定两个单词 word1 和 word2，找到使得 word1 和 word2 相同所需的最小步数，每步可以删除任意一个字符串中的一个字符。

找最长公共子序列? 假设长度为x

那么结果为(len1-x)+(len2-x)

dp[i][j] 表示得字符串word1的前i个字符和字符串word2的前j个字符的最长公共子序列.

最后一个字符是word1[i-1]和word2[i-1]

(1) word1[i-1]==word2[i-1], dp[i][j]=dp[i-1][j-1]

(2) word1[i-1]!=word2[i-1], dp[i][j]=max(dp[i-1][j], dp[i][j-1]);

边界:

dp[0][0] = 0;

dp[][0] = 0;

dp[0][] = 0;

|  |
| --- |
| class Solution {  public:  int minDistance(string word1, string word2) {  int len1 = word1.size();  int len2 = word2.size();  vector<vector<int>> dp(len1 +1, vector<int>(len2 + 1, 0));  for(int i = 1; i <= len1; ++i) {  for(int j = 1; j <= len2; ++j) {  if(word1[i - 1] == word2[j - 1])  dp[i][j] = dp[i - 1][j - 1] + 1;  else  dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);  }  }  int x = dp[len1][len2];  return len1 + len2 - 2 \* x;  }  }; |

28 ms 11.9 MB

空间压缩 要用两行的滚动数组

|  |
| --- |
| class Solution {  public:  int minDistance(string word1, string word2) {  int len1 = word1.size();  int len2 = word2.size();  vector<vector<int>> dp(2, vector<int>(len2 + 1, 0));  for(int i = 1; i <= len1; ++i) {  int pre = i & 1 ? 0 : 1;  int cur = i & 1 ? 1 : 0;  for(int j = 1; j <= len2; ++j) {  if(word1[i - 1] == word2[j - 1])  dp[cur][j] = dp[pre][j - 1] + 1;  else  dp[cur][j] = max(dp[pre][j], dp[cur][j - 1]);  }  }  int x = dp[len1 & 1][len2];  return len1 + len2 - 2 \* x;  }  }; |

24 ms 6.9 MB

## 646 最长数对链（中等）

题目描述:

给出 n 个数对。 在每一个数对中，第一个数字总是比第二个数字小。

现在，我们定义一种跟随关系，当且仅当 b < c 时，数对(c, d) 才可以跟在 (a, b) 后面。我们用这种形式来构造一个数对链。

给定一个数对集合，找出能够形成的最长数对链的长度。你不需要用到所有的数对，你可以以任何顺序选择其中的一些数对来构造。

原来的序对是乱序的, 对于一个结果集合set, 如果按set中数对的第一个数字从小到大排序, 那么一次遍历排序完以后的数对中的每个数, 一定是一个严格递增的序列.

所以, 我们可以先将pairs数对数组按第一个数从小到大排序

对于排完序以后的数对数组pairs

dp[j]表示以pairs[j]结尾的数对能形成最大数链的长度

(1) 对于任意i<j, 如果pairs[j][0]≤pairs[i][1], 则pairs[j]自己形成新的数对链, dp[j]=1

(2) 对于i<j, 如果pairs[j][0]>pairs[i][1], 计算dp[i]+1, 选最大的那个

动态保存rslt=max(rslt, dp[i]);

|  |
| --- |
| class Solution {  public:  int findLongestChain(vector<vector<int>>& pairs) {  sort(pairs.begin(), pairs.end(),  [](vector<int> &a, vector<int> &b) -> bool {  if(a[0] != b[0]) return a[0] < b[0];  else return a[1] < b[1];  });  int n = pairs.size();  vector<int> dp(n, 1);  int rslt = 1;  for(int j = 1; j < n; ++j) {  for(int i = 0; i < j; ++i) {  if(pairs[j][0] > pairs[i][1]) {  dp[j] = max(dp[j], dp[i] + 1);  }  rslt = max(rslt, dp[j]);  }  }  return rslt;  }  }; |

424ms 22.3 MB

还可以用区间贪心

对pairs[]按第一个数从大到小排序, 第二个数从小到大排序.

|  |
| --- |
| class Solution {  public:  int findLongestChain(vector<vector<int>>& pairs) {  sort(pairs.begin(), pairs.end(), [](vector<int> &a, vector<int> &b) -> bool {  if(a[0] != b[0]) return a[0] > b[0];  else return a[1] < b[1];  });  int n = pairs.size();  int rslt = 1;  int last0 = pairs[0][0];  for(int j = 1; j < n; ++j) {  if(pairs[j][1] < last0) {  rslt++;  last0 = pairs[j][0];  }  }  return rslt;  }  }; |

68ms 22MB

## 376 摆动序列(中等)

如果连续数字之间的差严格地在正数和负数之间交替，则数字序列称为 摆动序列 。第一个差（如果存在的话）可能是正数或负数。仅有一个元素或者含两个不等元素的序列也视作摆动序列。

例如， [1, 7, 4, 9, 2, 5] 是一个 摆动序列 ，因为差值 (6, -3, 5, -7, 3) 是正负交替出现的。

相反，[1, 4, 7, 2, 5] 和 [1, 7, 4, 5, 5] 不是摆动序列，第一个序列是因为它的前两个差值都是正数，第二个序列是因为它的最后一个差值为零。

子序列 可以通过从原始序列中删除一些（也可以不删除）元素来获得，剩下的元素保持其原始顺序。

给你一个整数数组 nums ，返回 nums 中作为 摆动序列 的 最长子序列的长度 。

dp[0][i] 以元素nums[i]结尾的最长摆动子序列的长度, 最后一个差值是负数

dp[1][i] 以元素nums[i]结尾的最长摆动子序列的长度, 最后一个差值是正数

dp[0][j]

(1) 对于i<j, 不存在nums[j]<nums[i], 则dp[0][j]=1

(2) 对于i<j, 存在nums[j]<nums[i], 计算dp[1][i]+1, 算完以后取最大值

用rslt动态保存最大值

dp[1][j]

(1) 对于i<j, 不存在nums[j]>nums[i], 则dp[1][j]=1

(2) 对于i<j, 存在nums[j]>nums[i], 计算dp[0][i]+1, 算完以后取最大值

用rslt动态保存最大值

边界:

dp[0][0]=1

dp[1][0]=1

dp[0][1] = (nums[1] < nums[0]) ? 2 : 1;

dp[1][1] = (nums[1] > nums[0]) ? 1 : 2;

|  |
| --- |
| class Solution {  public:  int wiggleMaxLength(vector<int>& nums) {  int n = nums.size();  vector<vector<int>> dp(2, vector<int>(n, 1));  if(n == 1) return 1;  dp[0][0] = 1;  dp[1][0] = 1;  dp[0][1] = (nums[1] < nums[0]) ? 2 : 1;  dp[1][1] = (nums[1] > nums[0]) ? 2 : 1;  int rslt = max(dp[0][1], dp[1][1]);  for(int j = 2; j < n; ++j) {  for(int i = 0; i < j; ++i) {  if(nums[j] < nums[i]) {  dp[0][j] = max(dp[0][j], dp[1][i] + 1);  }  if(nums[j] > nums[i]) {  dp[1][j] = max(dp[1][j], dp[0][i] + 1);  }  }  rslt = max(dp[0][j], dp[1][j]);  }  return rslt;  }  }; |

官方题解

上升摆动序列: 当且仅当该序列是摆动序列，且最后一个元素呈上升趋势.

下降摆动序列: 当且仅当该序列是摆动序列，且最后一个元素呈下降趋势.

对于长度为1的序列, 它既是**上升摆动序列**, 也是**下降摆动序列**.

峰: 该元素两侧的相邻元素均小于它.

谷: 该元素两侧的相邻元素均大于它.

位于序列两端的元素既是**峰**也是**谷**.

对于一段相同元素, 我们最多只能选择一个元素, 所以可以忽略相同的元素.

现在假定序列中任意两个元素都不相同.

过度元素: 既不是峰也不是谷, 如{1,2,3,4}中的3.

up[i] 前i个元素的最长上升摆动序列的长度.

dn[i] 前i个元素的最长下降摆动序列的长度.

最后一个元素是nums[i-1]

对于up[i], 从若nums[k]是最长上升摆动序列的最后一个元素, 则nums[k+1]~num[i-1]一定是程稳定的下降趋势的.

对于dn[i], 从若nums[k]是最长下降摆动序列的最后一个元素, 则nums[k+1]~num[i-1]一定是程稳定的上升趋势的.

计算up[i]

(1) nums[i]≤nums[i-1], 无法得到更长的上升摆动序列, up[i]=up[i-1].

(2) nums[i]>nums[i-1]

a) 选择从up转移, nums[i]比nums[k]大或小或等于, 都无法形成更长的**上升摆动序列**.

up[i] = up[i – 1]

b) 选择从dn转移, 可以形成比dn长1的上升摆动序列.

up[i] = dn[i – 1] + 1.

up[i] = max(up[i – 1], dn[i – 1] + 1);

计算dn[i]

(1) nums[i]≥nums[i-1], 无法得到更长的下降摆动序列, dn[i]=dn[i-1]

(2) nums[i]<nums[i-1]

a) 选择从dn转移, nums[i]比nums[k]大或小或等于, 都无法形成更长的**下降摆动序列**.

dn[i] = dn[i – 1]

b) 选择从up转移, 可以形成比up长1的上升摆动序列.

dn[i] = up[i – 1] + 1.

dn[i] = max(dn[i – 1], up[i – 1] + 1);

最终返回max(up[n – 1], dn[n – 1]);

边界: 数组初始化为1

up[1] = nums[1] > nums[0] ? 2 : 1;

dn[1] = nums[1] > nums[0] ? 2 : 1;

|  |
| --- |
| class Solution {  public:  int wiggleMaxLength(vector<int>& nums) {  int n = nums.size();  if(n < 2) return 1;  vector<int> up(n, 1);  vector<int> dn(n, 1);  for(int i = 1; i < n; ++ i) {  if(nums[i] > nums[i - 1]) {  up[i] = max(up[i - 1], dn[i - 1] + 1);  }  else if(nums[i] < nums[i - 1]) {  dn[i] = max(dn[i - 1], up[i - 1] + 1);  }  else {  up[i] = up[i - 1];  dn[i] = dn[i - 1];  }  }  return max(up[n - 1], dn[n - 1]);  }  }; |

空间压缩

|  |
| --- |
| class Solution {  public:  int wiggleMaxLength(vector<int>& nums) {  int n = nums.size();  if(n < 2) return 1;  int up = 1;  int dn = 1;  for(int i = 1; i < n; ++ i) {  if(nums[i] > nums[i - 1]) {  up = max(up, dn + 1);  }  else if(nums[i] < nums[i - 1]) {  dn = max(dn, up + 1);  }  }  return max(up, dn);  }  }; |

## 494 目标和(中等)

给定一个非负整数数组，a1, a2, ..., an, 和一个目标数，S。现在你有两个符号 + 和 -。对于数组中的任意一个整数，你都可以从 + 或 -中选择一个符号添加在前面。

返回可以使最终数组和为目标数 S 的所有添加符号的方法数。

有点像背包问题

将数组的数字求和得sum, sum-target=x, target<sum, x一定为正数,

即从nums[i]中选择数字其和为x得方案数, 0-1背包.

dp[i][j] 表示从前i个数中取数和为j得方案数

(1) 不使用nums[i-1], dp[i][j] = dp[i-1][j]

(2) 使用nums[i-1], dp[i][j] = dp[i-1][j-nums[i-1]]

dp[i][j] = dp[i - 1][j] + dp[i - 1][j - nums[i - 1]];

边界条件

dp[0][0]=1

dp[0][1:] = 0

dp[][0] = 1

|  |
| --- |
| class Solution {  public:  int findTargetSumWays(vector<int>& nums, int target) {  int sum = 0;  int n = nums.size();  int cntZero = 0;  for(int i = 0, j = 0; j < n; ++j) {  if(nums[j] == 0) {  ++cntZero;  continue;  }  sum += nums[j];  nums[i++] = nums[j];  }  if(sum < target) return 0;  int v = sum - target;  if(v & 1) return 0;  v /= 2;  vector<vector<int>> dp(n + 1 - cntZero, vector<int>(v + 1, 0));  dp[0][0] = 1;  for(int i = 1; i < dp.size(); ++i) {  dp[i][0] = 1;  for(int j = 1; j <= v; ++j) {  if(j < nums[i - 1]) dp[i][j] = dp[i - 1][j];  else {  dp[i][j] = dp[i - 1][j] + dp[i - 1][j - nums[i - 1]];  }  }  }  return dp[n - cntZero][v] \* pow(2, cntZero);  }  }; |

4 ms 98.86% 11.9 MB 31.18%

## 714 买卖股票的最佳时机含手续费

题目描述:

给定一个整数数组 prices，其中第 i 个元素代表了第 i 天的股票价格 ；非负整数 fee 代表了交易股票的手续费用。

你可以无限次地完成交易，但是你每笔交易都需要付手续费。如果你已经购买了一个股票，在卖出它之前你就不能再继续购买股票了。

返回获得利润的最大值。

注意：这里的一笔交易指买入持有并卖出股票的整个过程，每笔交易你只需要为支付一次手续费。

买出的时候才算手续费fee

dp[0][i]表示第i天末尾不持有股票得最大收益

dp[1][i]表示第i天末尾持有股票的最大收益

dp[0][i]

(1) 可能是第i天本来就没有股票, dp[0][i] = dp[0][i-1]

(2) 可能是第i天把股票卖出去了, dp[0][i] = dp[1][i - 1] + prices[i]-fee

dp[0][i] = max(dp[0][i - 1], dp[1][i - 1] + prices[i] - fee);

dp[1][i]

(1) 可能是第i天买的股票, dp[1][i] = dp[0][i-1]-prices[i].

(2) 可能是前一天已经有股票了, dp[1][i] = dp[1][i-1]

dp[1][i] = max(dp[0][i - 1] - prices[i], dp[1][i - 1]);

边界

dp[0][0] = 0;

dp[1][0] = -prices[0]

最后返回dp[0][n-1]

标准动态规划

|  |
| --- |
| class Solution {  public:  int maxProfit(vector<int>& prices, int fee) {  int n = prices.size();  vector<int> dp(2);  dp[0] = 0;  dp[1] = -prices[0];  for(int i = 1; i < n; ++i) {  int a = dp[0], b = dp[1];  dp[0] = max(a, b + prices[i] - fee);  dp[1] = max(a - prices[i], b);  }  return dp[0];  }  }; |

144ms 60.6MB

空间压缩版本

|  |
| --- |
| class Solution {  public:  int maxProfit(vector<int>& prices, int fee) {  int n = prices.size();  vector<int> dp(2);  dp[0] = 0;  dp[1] = -prices[0];  for(int i = 1; i < n; ++i) {  int a = dp[0], b = dp[1];  dp[0] = max(a, b + prices[i] - fee);  dp[1] = max(a - prices[i], b);  }  return dp[0];  }  }; |

92ms 53.7MB

## 403 青蛙过河(困难)

题目描述:

一只青蛙想要过河。 假定河流被等分为若干个单元格，并且在每一个单元格内都有可能放有一块石子（也有可能没有）。 青蛙可以跳上石子，但是不可以跳入水中。

给你石子的位置列表 stones（用单元格序号 升序 表示）， 请判定青蛙能否成功过河（即能否在最后一步跳至最后一块石子上）。

开始时， 青蛙默认已站在第一块石子上，并可以假定它第一步只能跳跃一个单位（即只能从单元格 1 跳至单元格 2 ）。

如果青蛙上一步跳跃了 k 个单位，那么它接下来的跳跃距离只能选择为 k - 1、k 或 k + 1 个单位。 另请注意，青蛙只能向前方（终点的方向）跳跃。

dp[i][j]表示青蛙最后一步跳j步是否能跳到石头i上.

边界: dp[1][1] = true;

由于数组的元素范围比较大, 可以用集合里面是否存在元素来表示布尔值.

每一步都要从dp[i][j]推出 dp[k1][j-1] dp[k2][j] dp[k3][j+1]

其中k1表示高度为stones[i]+(j-1)的石头的下标, 当然这样的石头很可能是不存在, 所以通过二分查找来排除不需要计算的项目.

|  |
| --- |
| class Solution {  public:  int binSearch(vector<int> &nums, int left, int right, int val) {  int mid;  while(left < right) {  mid = (right - left) / 2 + left;  if(nums[mid] < val)  left = mid + 1;  else if(nums[mid] > val)  right = mid;  else  return mid;  }  return -1;  }  bool canCross(vector<int>& stones) {  int n = stones.size();  vector<unordered\_set<int>> dp(n);  if(stones[1] != 1) return false;  dp[1].insert(1);  for(int i = 1; i < n - 1; ++i) {  for(auto it = dp[i].begin() ; it != dp[i].end(); it++) {  for(int j = \*it - 1; j <= \*it + 1; j++) {  int k = binSearch(stones, i + 1, n, stones[i] + j);  if(k != -1) {  dp[k].insert(j);  }  }  }  }  return !dp[n - 1].empty();  }  }; |

C++ STL的二分查找函数:

bool binary\_search(arr.begin(), arr.begin() + size, val);

找打分会true, 否则返回false.

## 44 通配符匹配(困难)

题目描述:

给定一个字符串 (s) 和一个字符模式 (p) ，实现一个支持 '?' 和 '\*' 的通配符匹配。

'?' 可以匹配任何单个字符。

'\*' 可以匹配任意字符串（包括空字符串）。

dp[i][j]表示字符串s的前i个字符和字符模式p的前i个字符是否能匹配.

s[i-1]和p[j-1]是最后一个字符.

(1) p[j-1]=='?', 匹配当前字符, dp[i][j]=dp[i-1][j-1]

(2) p[j-1]=='\*', '\*'可以进行匹配(可以重复利用)或不匹配(和空串匹配)

dp[i][j] = d[i-1][j] || dp[i][j-1]

(3) p[j-1]==s[i-1], 匹配当前字符, dp[i][j]=dp[i-1][j-1]

(4) p[j-1]!=s[i-1], dp[i][j] = false

边界:

空串和空串可以匹配dp[0][0] = true

dp[1:][0] = false

dp[0][1:] = p[j - 1] == '\*' ? dp[0][j - 1] : false;

|  |
| --- |
| class Solution {  public:  bool isMatch(string s, string p) {  int m = s.size(), n = p.size();  vector<vector<bool>> dp(m + 1, vector<bool>(n + 1, false));  dp[0][0] = true;  // 边界  for(int i = 1; i <= m; ++i) {  dp[i][0] = false;  }  for(int j = 1; j <= n; ++j) {  dp[0][j] = p[j - 1] == '\*' ? dp[0][j - 1] : false;  }  for(int i = 1; i <= m; ++i) {  for(int j = 1; j <= n; ++j) {  if(p[j - 1] == '\*')  dp[i][j] = dp[i - 1][j] || dp[i][j - 1];  else if(p[j - 1] == '?' || p[j - 1] == s[i - 1])  dp[i][j] = dp[i - 1][j - 1];  else  dp[i][j] = false;  }  }  return dp[m][n];  }  }; |

108ms 11.1MB

黄色高亮语句不能写为return false.

空间压缩

|  |
| --- |
| class Solution {  public:  bool isMatch(string s, string p) {  int m = s.size(), n = p.size();  vector<vector<bool>> dp(2, vector<bool>(n + 1, false));  dp[0][0] = true;  dp[1][0] = false;  for(int j = 1; j <= n; ++j) {  dp[0][j] = p[j - 1] == '\*' ? dp[0][j - 1] : false;  }  for(int i = 1; i <= m; ++i) {  int pre = i & 1 ? 0 : 1;  int cur = i & 1 ? 1 : 0;  dp[cur][0] = false;  for(int j = 1; j <= n; ++j) {  if(p[j - 1] == '\*')  dp[cur][j] = dp[pre][j] || dp[cur][j - 1];  else if(p[j - 1] == '?' || p[j - 1] == s[i - 1])  dp[cur][j] = dp[pre][j - 1];  else  dp[cur][j] = false;  }  }  return dp[m & 1][n];  }  }; |

72ms 7MB

注意黄色高亮语句, 用以代替第0列的初始化.

dp[i-1]对应dp[pre]

dp[i]对应dp[cur]

## 62 不同路径(中等)

## 368 最大整除子集(中等)

# 分治

# 数学

# 位运算

# 简单数据结构

# 字符串

# 链表

# 树

# 图

# 复杂数据结构