# 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 |
|  |  |
|  |  |
|  |  |
|  |  |

# 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

# 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