Leetcode刷题笔记

目录

[Leetcode刷题笔记 1](#_Toc129171783)

[1. No0279 1](#_Toc129171784)

[2. No0300: 最长递增子序列长度 3](#_Toc129171785)

[3. No0673: 最长递增子序列的个数 4](#_Toc129171786)

[4. No0982：按位与为0的三元组 4](#_Toc129171787)

[4.1 解法1：双二重循环 4](#_Toc129171788)

[4.2 解法2：枚举+子集优化 5](#_Toc129171789)

[5. No1653：使字符串平衡的最少删除次数 5](#_Toc129171790)

[6. LCOF047: 礼物的最大价值 5](#_Toc129171791)

[6.1 基本的动态规划问题 6](#_Toc129171792)

[6.2 内存优化：滚动数组 6](#_Toc129171793)

[6.3 代码优化：边界处理优化 6](#_Toc129171794)

[7. 常用算法和数据结构 7](#_Toc129171795)

[7.1 Dynamic Programming 7](#_Toc129171796)

[7.2 滚动数组 7](#_Toc129171797)

[7.3 DFS 7](#_Toc129171798)

[7.4 BFS 7](#_Toc129171799)

**[Notations and Abbreviations]**

# No0279

【问题描述】

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（可以重复）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。

【解题分析】

本题可以转化为图搜索中的最短路径问题，因此可以用广度优先搜索算法来解决。

举个例子，令n = 30, m=floor(sqrt(n)) = 5. 则构造n的完全平方数的和，可以使用模块（即数字）为1~5。从中任选一个k使用得到(n=30)的邻节点(n-k\*k)。针对每个节点都执行同样操作，直到最后到达值为0的节点。如下图所示：



Edge上的数字表示要用于构成完全平方数和的数（其实是其平方），对应的子节点的值等于父节点的值减去该数的平方。

广度优先搜索的三个基本要素：

1. 队列管理
2. 节点的表示以及邻节点的遍历
3. 已访问节点的管理

算法流程如下(python-style, general flow for BFS)：

初始化：

创建Queue, visited对象

Add start node (n) to Queue, together with its layer (for start node, it should be zero)

Add start node to visited

While (Queue is not empty):

value, layer = Queue.pop()

if value is the target (it is 0 in this problem):

return layer+1

Traverse all the neighbour nodes, for each of them:

If neighbourNode is not in visited:

Add neighbourNode to visited:

Queue.push(neighbourNode)

在本问题中， 在本问题中，邻节点即从当前节点出发，将每个可用的数（的平方）用一次（即减去）后能到达的下一个值。可以以递推关系式表示如下：



【优化】

四平方定理： 任何一个正整数都可以表示成不超过四个整数的平方之和。 推论：满足四数平方和定理的数n（四个整数的情况），必定满足 n=4^a(8b+7)

这道题如果知道数学定理之后，相当于告诉你：

任何正整数都可以拆分成不超过4个数的平方和 ---> 答案只可能是1,2,3,4

如果一个数最少可以拆成4个数的平方和，则这个数还满足 n = (4^a)\*(8b+7) ---> 因此可以先看这个数是否满足上述公式，如果不满足，答案就是1,2,3了

如果这个数本来就是某个数的平方，那么答案就是1，否则答案就只剩2,3了

如果答案是2，即n=a^2+b^2，那么我们可以枚举a，来验证，如果验证通过则答案是2

只能是3

# No0300: 最长递增子序列长度

初始思路（以下两种思路是等价的，只不过一个是从前往后看，一个是从后往前看）：

思路1：以L(k) 代表数组nums[k:]中最长递增子列长度。有以下递推关系：

如果nums[k:]中最长递增子列的第一个数小于nums[k-1]，L(k-1)=L(k)+1

否则，L(k-1)=L(k)

思路2：以L(k) 代表数组nums[:k]中最长递增子列长度。有以下递推关系：

如果nums[:k]中最长递增子列的最后一个数小于nums[k]，L(k+1)=L(k)+1

否则，L(k+1)=L(k)

但是，这样基于长度的递推关系构建不够。必需基于最长子序列构建递推关系。如下所示：

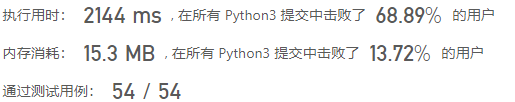
记S[k]代表nums[:k]的最长子序列。

比较nums[k]与S[k][-1], S[k-1][-1],...~~直到找到第一个满足nums[k] > S[j][-1]~~，搜索最长的满足nums[k] > S[j][-1]的S[j\_max]，则S[k+1]= S[j j\_max]+ nums[k](这里+表示列表串联)

注：以S[k]代表nums[k:]的最长子序列求递推关系的做法理论上也可以，但是要从尾部开始反向遍历。如果仍然正向遍历的话，需要显式的递归调用，运行效率极低。

【优化方案】

事实上并不需要保存各S[k]，只需要保存对应最长子序列的最后一个数及其长度即可。这样优化后时间和空间性能都应该得到改善。不过，还不够好。。。如下所示：



# No0673: 最长递增子序列的个数

思路：

# No0982：按位与为0的三元组

暴力破解的话，复杂度为O(n^3)。

## 解法1：双二重循环

分而治之。

先进行二重循环，并统计出现的每种结果的个数，存储于一个数组（或哈希表）A中。数组大小取决于输入数的范围。本题中数据范围定义为[0,2\*\*16)，因此该数组大小为最大2\*\*16。

然后再进行数组A与nums之间相与的二重循环。

这样，复杂度降为O(n^2+2^16\*n)~ O(n^2)。

## 解法2：枚举+子集优化



# No1653：使字符串平衡的最少删除次数

删除操作的目的是使得最终字符串中前半部分全部是“a”，后半部分全部是“b”。要么删除前半部分的“b”，要么删除后半部分的“a”。从前往后遍历，看到字母“b”时需要决定是保留它还是删除它。如果是保留的话，则其后所有的“a”都要删除。

第一感是动态规划。记dp(s)表示针对字符串s为满足条件的最少删除次数。则

dp(s) = min( *1 + dp(s[1:])*, *num of ‘a’ in s[1:]* )

但是，如果直接对这种形式的递归关系式进行编程的话，会出现递归深度太大而失败。

简化的方式参见代码。

# LCOF047: 礼物的最大价值

剑指offer 47

看到题目第一反应迪杰斯特拉算法……一开始琢磨了半天路径遍历。。。其实是一个经典的动态规划问题。

## 基本的动态规划问题

递推关系如下所示：

S[I,J] = grid[I,J] + max(S[I+1,J], S[I,J+1])

DP建表时从右下角向左上角反向进行。

内存性能非常不堪。。。



当然，反过来从左上角出发到右下角进行递推也可以（S[I,J]的含义与上面的不同）：

S[I,J] = grid[I,J] + max(S[I-1,J], S[I,J-1])

## 为什么可以动态规划？

这道题可以使用动态规划的原因在于，只允许向右或者向下。这样，在任何一条路径中，如果(i-1,j)和(i,j)都在路径中，则一定是先经过(i-1,j)，后经过(i,j)，这样确保动态规划适用的前提条件成立。对于 (i,j-1)和(i,j)也是同样道理。

如果我们可以向四个方向进行移动，那么上面的限制就不再满足，我们也就不能使用动态规划来解决了。

## 内存优化：滚动数组

很显然，在上述从下到上从右到左的刷表过程中，刷倒数第2行时只需要倒数第1行的信息，刷倒数第3行时只需要倒数第2行的信息。。。刷倒数第k+1行时只需要倒数第k行的信息。。。所以，并不需要维护一张n\*m的表，而只需要维护一行的数据，就在一的行数组上进行in-place的更新即可。这个数组复用的技巧通常被称之为滚动数组。这样的话，空间复杂度就由O(nm)变成了O(max(n,m))

当然，只维护一列的数据亦可，具体哪个更好取决于行数大还是列数大。



## 代码优化：边界处理优化

这一类棋盘类或者网格类问题通常有一个边界约束。边界上的处理与中央格点的处理会有所不同，这个就需要在循环处理中针对边界元素进行特殊处理，会使得代码处理比较冗长。

一个常见的技巧是为“棋盘”或“网格”外面加一圈边。这样原有棋盘上的点全部变成内点，就可以统一处理，前提是对外加的这一圈边进行适当的初始化。

# 常用算法和数据结构

## Dynamic Programming

## 滚动数组

## DFS

## BFS

BFS 其实是很简单的基础算法，抓住如下几点即可轻松写出不易错的 baseline:

BFS 算法组成的 3 元素：队列，入队出队的节点，已访问的集合。

队列：先入先出的容器；

节点：最好写成单独的类，比如本例写成 (value,step) 元组。也可写成 (value,visited)，看自己喜好和题目；

已访问集合：为了避免队列中插入重复的值

BFS算法组成的套路：

初始化三元素：

Node = node(n) queue = [Node] visited = set([Node.value])

操作队列 —— 弹出队首节点：

vertex = queue.pop(0)

操作弹出的节点 —— 根据业务生成子节点（一个或多个）：

[node(vertex.value - n\*n, Node.step+1) for n in range(1,int(vertex.value\*\*.5)+1)]

判断这些节点 —— 符合业务条件，则return，不符合业务条件，且不在已访问集合，则追加到队尾，并加入已访问集合：

if i==0:

return new\_vertex.step

elif i not in visited:

queue.append(new\_vertex)

visited.add(i)```

若以上遍历完成仍未return，下面操作返回未找到代码：

return -1

[Reference]

[Revision history]