Leetcode刷题笔记

目录

[Leetcode刷题笔记 1](#_Toc130057305)

[1. 概要 2](#_Toc130057306)

[2. No0001. 两数之和 2](#_Toc130057307)

[3. No0002. 两数相加 3](#_Toc130057308)

[4. No0112. 路径总和 3](#_Toc130057309)

[5. No0279 4](#_Toc130057310)

[6. No0300. 最长递增子序列长度 5](#_Toc130057311)

[7. No0437. 路径总和 III 6](#_Toc130057312)

[8. No0523. 连续的子数组和 6](#_Toc130057313)

[9. No0525. 连续数组 7](#_Toc130057314)

[10. No0560. 和为k的子数组 7](#_Toc130057315)

[10.1 方法1：枚举 7](#_Toc130057316)

[10.2 方法2：前缀和+哈希表 7](#_Toc130057317)

[11. No0582. 杀掉进程 8](#_Toc130057318)

[12. No0673. 最长递增子序列的个数 8](#_Toc130057319)

[13. No0974. 和可被 K 整除的子数组 9](#_Toc130057320)

[14. No0982. 按位与为0的三元组 9](#_Toc130057321)

[14.1 解法1：双二重循环 9](#_Toc130057322)

[14.2 解法2：枚举+子集优化 10](#_Toc130057323)

[15. No1290. 二进制链表转整数 10](#_Toc130057324)

[16. No1550. 存在三个连续奇数的数组 10](#_Toc130057325)

[17. No1590. 使数组和能被 P 整除 11](#_Toc130057326)

[18. No1605. 给定行和列的和求可行矩阵 11](#_Toc130057327)

[19. No1615. 最大网络秩 12](#_Toc130057328)

[20. No1616.分割两个字符串得到回文串 13](#_Toc130057329)

[21. No1653.使字符串平衡的最少删除次数 13](#_Toc130057330)

[22. No2379.得到 K 个黑块的最少涂色次数 14](#_Toc130057331)

[23. No2383. 赢得比赛需要的最少训练时长 14](#_Toc130057332)

[24. No2389. 和有限的最长子序列 14](#_Toc130057333)

[25. No2488. 统计中位数为 K 的子数组 15](#_Toc130057334)

[26. LCOF047: 礼物的最大价值 15](#_Toc130057335)

[26.1 基本的动态规划问题 15](#_Toc130057336)

[26.2 为什么可以动态规划？ 15](#_Toc130057337)

[26.3 内存优化：滚动数组 16](#_Toc130057338)

[26.4 代码优化：边界处理优化 16](#_Toc130057339)

[26.5 总共有多少种可能的路径？ 16](#_Toc130057340)

[27. LCCI《程序员面试金典(第6版)》 16](#_Toc130057341)

[27.1 17.05：字母与数字 16](#_Toc130057342)

[28. 常用算法、数据结构和技巧 17](#_Toc130057343)

[28.1 位操作 17](#_Toc130057344)

[28.2 先排序后处理 17](#_Toc130057345)

[28.3 二分法 17](#_Toc130057346)

[28.4 树搜索 17](#_Toc130057347)

[28.5 前缀和+哈希表 17](#_Toc130057348)

[28.6 Dynamic Programming 18](#_Toc130057349)

[28.7 滚动数组 18](#_Toc130057350)

[28.8 滑动窗 18](#_Toc130057351)

[28.9 双指针 18](#_Toc130057352)

[28.10 DFS 18](#_Toc130057353)

[28.11 BFS 18](#_Toc130057354)

**[Notations and Abbreviations]**

# 概要

具体问题描述和实现参见同目录中以题号开头的源代码文件。本文着重简要的解题思路说明。

子数组：在题目描述中出现的子数组（如无特别声明）通常是指：数组中一段连续非空区间中的数构成的数组称为子数组

# No0001. 两数之和

【解法1】

暴力方法进行遍历的话需要O(n^2)的时间复杂度。

先排序。然后从两端向中间搜索。

Left初始化为nums[0](排序后)，right初始化为nums[1]。两者之和如果大于target，则right往左移；两者之和如果小于target，则right往右移；如此继续一直到left+right=target成立。

排序的时间复杂度O(n\*log(n))，此后的搜索的时间复杂度为O(n)。因此，总的复杂度为O(n\*log(n))。

需要注意的是，要返回的答案是原数组中的序号。可以保留原数组的copy，在找到两个数以后，再去查询它们各自在原数组中位置（因为题设条件中有“数组中同一个元素在答案里不能重复出现”，所以这个是可行的）。

【解法2】

建立哈希表：h[nums[k]] = k.

然后可以利用哈希表的O(1)查询（查询target-nums[k] for each k）效率

哈希表的建立和查询可以一次性前向遍历完成，因此总的时间复杂度只有O(1)。

# No0002. 两数相加

# No0112. 路径总和



左：示例一的图；右：示例二的图

树的遍历搜索。树搜索可以用深度优先搜索，也可以用广度优先搜索。本题适合于用深度优先搜索，找到即退出。

初始化栈; 将{root, pathsum=root.val}入栈;

While stack is not empty：

Pop one item {node, pathsum} from stack

For each child of node:

Child\_pathsum = pathsum + child.val

If Child\_pathsum == targetSum: Return True

Put {child, child\_pathsum} into stack

Return False

# No0279

【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。换句话说遍历方向要与S[k]代表的含义相匹配。

【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。。。只击败了13%。

# No0437. 路径总和 III

# No0523. 连续的子数组和

前缀和+哈希表。参见No0560，以及16.2

有以下两点不同：

1. 子数组长度不小于2
2. 子数组和为k的倍数

第2点与No0974相同。所以可以在No974的解的基础上考察子数组长度不小于2的情况。此外，本题只要求是否存在而不要求统计个数，所以一旦找到满足条件即可提前退出。

在之前的几道题中，哈希表中没有体现子数组的位置信息。本题需要保留子数组的位置信息，因此哈希表中的value可以改为用一张表来保存子数组的位置信息（终止位置i）。

需要注意的一点是列表的操作问题：



# No0525. 连续数组

前缀和+哈希表。参见No0560，No523, 以及16.2

记pre[i]表示到nums[i]为止的前缀模2和。

创建哈希表，key={1的个数减去0的个数的差值}，value为对应key值的i值列表。一次从左到右的遍历以O(n)的时间复杂度创建哈希表。在此过程中，针对每个i，计算到i为止的满足条件的最长子数组。

初始提交：

执行用时：5268 ms, 在所有 Python3 提交中击败了5.16%的用户

内存消耗：23.7 MB, 在所有 Python3 提交中击败了5.16%的用户

优化：事实上不需要存储对应key值的i值列表，而只需要记忆对应key值第一次出现的i即可，这样既节约存储又节约查询时间。

执行用时：212 ms, 在所有 Python3 提交中击败了44.41%的用户

内存消耗：20.5 MB, 在所有 Python3 提交中击败了34.39%的用户

# No0560. 和为k的子数组

## 方法1：枚举

令S[j,i]表示从nums[j]到nums[i]的子数组和。

最粗暴的方式是二维遍历所有可能的(j,i)组合（时间复杂度是O(n^2)），计算所有各种组合的部分和S[j,i]，考虑到计算S[j,i]的复杂度也是O(n)，所以总的时间复杂度是O(n^3).

进一步，针对特定的j，它的所有部分和S[j,i]（针对不同的i：j<=i<=N-1）的计算不是相互独立的。比如说，S[j,i]= S[j,i-1]+nums[i]。所以可以以一次前向遍历计算出针对某个j所有的S[j,i](i:j<=i<=N-1)。这样的话，时间复杂度可以简化为O(n^2)。而针对j的遍历中，只需要记录一个到当前i位置的累加和，不同j的遍历是串行而且相互独立的，因为空间复杂度为O(1)。

## 方法2：前缀和+哈希表

方法一的瓶颈在于对每个 *i*，我们需要枚举所有的 *j* （（或者，对每一个起始点*j*，要遍历所有的可能的终点*i*））来判断子数组和是否符合条件，这一步是否可以优化呢？答案是可以的。

定义 pre[i] 为 [0..i] 里所有数的和，则 pre[i] 可以由 pre[i−1] 递推而来，即：

pre[i]=pre[i−1]+nums[i]

那么“[j..i] 这个子数组和为 k”这个条件我们可以转化为

pre[i]−pre[j−1]==k

简单移项可得符合条件的下标j 需要满足

pre[j−1]==pre[i]−k

所以我们考虑以 i 结尾的和为 k 的连续子数组个数时只要统计满足以下条件的j的个数：

1. 0<=j<=i
2. pre[j]==pre[i]−k

由于只关心个数，而不关心j的实际值是什么，所以我们可以建立哈希表hmap，以前缀和的值为key，该前缀和值出现的次数为value，这样我们在考察子数组结束位置为i的情况时，只要查询hmap[pre[i]-key]即可用O(1)的复杂度查询出结束位置为i的满足“其和为k”条件的子数组个数。

哈希表hmap的创建可以在从左到右遍历过程中建立，这个需要O(n)的复杂度。而如上所示查询针对每个位置i的满足条件的子数组数只需要O(1)的复杂度，因此总的只需要O(n)的时间复杂度。需要存储一张哈希表，空间复杂度为O(n)。

# No0582. 杀掉进程

搜索（以两张列表形式表示的）树中以指定节点为祖先节点的所有节点。

从某一个节点开始，基于ppid一路向上查询，有两种可能：

1. 最终到达根节点，表明该节点不是指定节点的后代节点。这样的话，沿路的所有节点都不是。
2. 最终到达指定节点，表明该节点是指定节点的后代节点。这样的话，沿路的所有节点都应该删除。

如果指定节点本身是根节点的话，则直接返回原输入数组即可。

从左到右选定上溯的起点，上溯过程中，另外用一个visited数组标记已访问节点1，以避免重复访问。

执行用时：4316 ms, 在所有 Python3 提交中击败了5.97%的用户

内存消耗：26.1 MB, 在所有 Python3 提交中击败了40.30%的用户

如何优化？

# No0673. 最长递增子序列的个数

思路：

# No0974. 和可被 K 整除的子数组

前缀和+哈希表。参见No0560。

本题与No560的差异仅在于从“子数组和等于k”的条件变为“子数组和能被k整除，也即为k的整数倍”。所有，只要两个前缀和对k同余的话，两者前缀和的差就表示一个能被k整除的子数组了。

所以，本题中，用(presume[i]%k)作为哈希表的key即可。

# 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



# No1290. 二进制链表转整数

解题思路：顺着链表进行遍历即可。每往前前进一个节点，当前值乘以2再加上当前节点值。如下所示，采用位操作的方式计算速度能够得到一定程度优化。

执行用时：44 ms, 在所有 Python3 提交中击败了19.17%的用户

内存消耗：14.7 MB, 在所有 Python3 提交中击败了93.80%的用户

优化：用“num = (num << 1) | head.val”替代“num = 2\*num + head.val”

执行用时：36 ms, 在所有 Python3 提交中击败了72.93%的用户

内存消耗：14.9 MB, 在所有 Python3 提交中击败了23.50%的用户

# No1550. 存在三个连续奇数的数组

基本上也是一个滑动窗的问题。要点在于，根据每个滑动窗的判断情况，可以跳跃前进，确保每个数只被判断一次。比如说，如果arr[k]是奇数，arr[k+1]是奇数，arr[k+2]是偶数。那当前窗口判定False，但是接下来可以直接跳到k+3的位置去开始新的滑动窗的判断。

判断是否奇数可以用“if (arr[k] % 2) == 1”，也可以用“if (arr[k] & 1) == 1:”，后者要快一些。

2023-03-09

执行用时：36 ms, 在所有 Python3 提交中击败了65.87%的用户

内存消耗：15 MB, 在所有 Python3 提交中击败了70.66%的用户

# No1590. 使数组和能被 P 整除

解题思路：

要删掉的数的总和必定等于原数组总和S0对p同余，记k= S0%p。所以问题变为找最短的和与k同余（对p）的数组长度。转变后这个问题与No560、No974有一定的相似，但是却又不同。

本题中哈希表的键值用“pre = (pre + nums[i]) % p”，而对比查询时需要满足的条件是满足对p同余于k，即：

if ((pre - k) % p) in hmap:

minlen = min(minlen, i - hmap[((pre - k) % p)])

既然是关注最短子数组，那么哈希表中就应该是存储出现对应键值的最后那个i（参考5）。

执行用时：128 ms, 在所有 Python3 提交中击败了58.39%的用户

内存消耗：35.5 MB, 在所有 Python3 提交中击败了33.13%的用户

# No1605. 给定行和列的和求可行矩阵

本质上就是解一个线性方程组。

根据题目条件可以得到n+m个方程（n为行数，m为列数），但是需要求解n\*m个变量，因此理论上有无穷多个解（当然像{n,m}={2,2}等情况也可能只有唯一解）。

但是题目只要求给出任意一个可行解。经过简单试算，推测可以用贪婪策略求解。但是如何贪婪能确保得到正确的解呢？初步感觉：

A[0,0] = min{rowSum[0],colSum[0]}

If rowSum[0] >= colSum[0]:

置第一列其余数全0;

置A[0,1]= rowSum[0] - colSum[0];

Else:

置第一行其余数全0;

置A[1,0]= colSum[0] - rowSum[0];

由此将A的第一行和第一列都确定了。问题缩减为[n-1,m-1]维的问题，然后按同样方式处理即可。这个称为“减而治之”策略。可以用递归的方式进行实现。但是以上方式能够确保求得正确的解吗？这个需要证明。此外，n和m较大时，以上策略会导致太深的调用堆栈。有没有效率更高的分而治之的方式？

【参考解】



# No1615. 最大网络秩

暴力法就是直接遍历每对城市的网络秩，然后求其最大值。

改进做法：先统计每个城市的道路连接数（记为城市秩）。任何两个城市构成的城市对的网络秩要么等于两个城市秩之和（如果两个城市之间无道路相连）；要么等于两个城市秩之和减一（如果两个城市之间有道路相连）。

首先，遍历roads求出所有城市的城市秩。

其次，不必遍历计算每两个城市之间的网络秩。因为任何两个城市之间最多只有一条道路，最大的网络秩一定出现于城市秩最大的两个城市之间；当多个城市对的城市秩之和相等时，最大网络秩出现在没有相互连接的城市对之间。

特殊情况在于：有3个或3个以上城市秩并列第一的情况；城市秩最大者唯一，但是城市秩并列第二者2个或2个以上。

如果并列第一的有2个或2个以上，查找是否存在相互之间无通路。全部有则2\*max1-1；否则就是2\*max1

如果没有并列第一，但是有两个或者以上的并列第二，查找是否有与第一者无通路的。如果有则max1+max2，否则max1+max2-1

如果既没有并列第一也没有并列第二，则max1+max2[-1]（取决于两者之间是否有连通）

执行用时：64 ms, 在所有 Python3 提交中击败了85.21%的用户

内存消耗：16.1 MB, 在所有 Python3 提交中击败了60.56%的用户

结果还不错，但是有点冗长。有没有什么更精致简介又不是效率的写法呢？

# No1616.分割两个字符串得到回文串

就是找a的前半段和b的后半段能否构成回文串或者a的后半段和b的前半段能否构成回文串。约束条件是a，b的分割点要保持一致。

暴力破解法就是遍历所有len(a)+1种分割点，针对每种分割点确认两种组合是否构成回文串。

【优化方案】

看看分割点在k时的情况与分割点在k+1时的情况存在什么关联。

能否在已经进行了针对分割点k的判断（并且判断结果为False）后，能否以比如说O(1)的复杂度完成针对分割点k+1的判断。

应该从中间向两边搜索，这样如果不存在的话可以今早发现并提前退出。

初始解法超时。。。

【双指针（官解）】



执行用时：96 ms, 在所有 Python3 提交中击败了44.07%的用户

内存消耗：15.9 MB, 在所有 Python3 提交中击败了6.78%的用户

通过测试用例：109 / 109

官解竟然排名如此，高手在民间啊。。。要好好学习一下

# 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。

# No2379.[得到 K 个黑块的最少涂色次数](https://leetcode.cn/problems/minimum-recolors-to-get-k-consecutive-black-blocks/)

因为要求连续黑色块数目，第一感是滑动窗。

取窗宽为k的滑动窗，从左向有移动。基于第一个窗口进行cur\_W2B，min\_W2B的初始化。然后每向右移一格时，根据移出的块和移进的块的颜色进行cur\_W2B，min\_W2B更新。

当min\_W2B变为0时，可以提前退出。

性能有待优化。

执行用时：40 ms, 在所有 Python3 提交中击败了51.63%的用户

内存消耗：15 MB, 在所有 Python3 提交中击败了26.09%的用户

通过测试用例：122 / 122

# No2383. 赢得比赛需要的最少训练时长

每战胜一个对手都要消耗该对手等量energy，所以总能量消耗就是所有对手energy之和。而由于energy需要严格大于对手才能获胜，所以总能量需求是sum(energy)+1，因此需要的训练时间就是：

max(sum(energy)+1-initialEnergy,0)

每战胜一个对手都要消耗该对手等量experience，因此，所需要的增加experience的训练量可以表达为：

max(0, max(experience[i] - initialExperience+sum(experience[0...i-1])) ) for i=0,1,...

# No2389. 和有限的最长子序列

本题求的是子序列长度，只需要考虑个数，不需要考虑nums中元素的相对位置，可以对数组进行排序后再求解。

又，题目要求的是比总和小于等于指定数值的最大序列长度，很显然，这就是排序后的不大于指定数值的最大前缀和的长度。

如果只是指定一个query的话，那就从左到右遍历，一边求前缀和一边进行判断即可。

但是题目要求多个queries的处理，不能针对每个query都进行一次前缀和遍历。而且，queries中的各query值并依序排列的，而且结果中需要与原queries对应的结果，因此不能轻易对queries进行先排序后处理。

可以先求前缀和，然后针对每个query进行二分法查找。

# No2488. 统计中位数为 K 的子数组

# 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。

## 总共有多少种可能的路径？

看到这个题目第一感是路径遍历。。。这里来计算一下总共有多少种可能的路径。这也是一个经典问题。

从左上角到右下角，只允许向右或向下。无论走什么路径都需要n次向下和m次向右。各路径间不同的是向右的行动和向下的行动的顺序不同。换一个方式来考虑，其实就是从总共（n+m）次行动中任选n次来执行向下的操作，其余则执行向右的操作。因此可得可能的路径数为，取题设要求的n和m最大值200,总的路径数是，这是一条难以想象的巨大的数，所以路径遍历完全行不通。

# LCCI《程序员面试金典(第6版)》

## 17.05：字母与数字

# 常用算法、数据结构和技巧

## 位操作

## 先排序后处理

No2389

## 二分法

No2389

Python library: bisect

## 树搜索

No112,

## 前缀和+哈希表

相关题目：推荐按照顺序完成：

[560. 和为 K 的子数组](https://leetcode.cn/problems/subarray-sum-equals-k/)√

[974. 和可被 K 整除的子数组](https://leetcode.cn/problems/subarray-sums-divisible-by-k/)√

[523. 连续的子数组和](https://leetcode.cn/problems/continuous-subarray-sum/)√

[525. 连续数组](https://leetcode.cn/problems/contiguous-array/)√

1590. 使数组和能被 P 整除√

其它：No2389

## Dynamic Programming

## 滚动数组

## 滑动窗

## 双指针

No1616

## 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]