Leetcode刷题笔记

目录

[Leetcode刷题笔记 1](#_Toc134015837)

[1. 概要 3](#_Toc134015838)

[2. leetcode 3](#_Toc134015839)

[2.1 No0001. 两数之和 3](#_Toc134015840)

[2.2 No0002. 两数相加 4](#_Toc134015841)

[2.3 No0003. 无重复字符的最长子串 4](#_Toc134015842)

[2.4 No0011. Container With Most Water 4](#_Toc134015843)

[2.5 No0012. Integer to Roman 5](#_Toc134015844)

[2.6 No0112. 路径总和 5](#_Toc134015845)

[2.7 No0279. Perfect Squares 5](#_Toc134015846)

[2.8 No0300. 最长递增子序列长度 7](#_Toc134015847)

[2.9 No0436. 寻找右区间 8](#_Toc134015848)

[2.10 No0437. 路径总和 III 8](#_Toc134015849)

[2.11 No0523. 连续的子数组和 8](#_Toc134015850)

[2.12 No0525. 连续数组 8](#_Toc134015851)

[2.13 No0560. 和为k的子数组 9](#_Toc134015852)

[2.13.1 方法1：枚举 9](#_Toc134015853)

[2.13.2 方法2：前缀和+哈希表 9](#_Toc134015854)

[2.14 No0582. 杀掉进程 10](#_Toc134015855)

[2.15 No0673. 最长递增子序列的个数 10](#_Toc134015856)

[2.16 No970. Powerful Integers 10](#_Toc134015857)

[2.17 No0974. 和可被 K 整除的子数组 11](#_Toc134015858)

[2.18 No0982. 按位与为0的三元组 11](#_Toc134015859)

[2.18.1 解法1：双二重循环 11](#_Toc134015860)

[2.18.2 解法2：枚举+子集优化 11](#_Toc134015861)

[2.19 No1003. Check If Word Is Valid After Substitutions 12](#_Toc134015862)

[2.19.1 方法1：有限状态机 12](#_Toc134015863)

[2.19.2 方法2：栈 12](#_Toc134015864)

[2.20 No1012. 至少有 1 位重复的数字 13](#_Toc134015865)

[2.20.1 方法一：数位DP+记忆化搜索 13](#_Toc134015866)

[2.21 No1019. Next Greater Node In Linked List 13](#_Toc134015867)

[2.22 No1023. Camelcase Matching 14](#_Toc134015868)

[2.23 No1026. Maximum Difference Between Node and Ancestor 14](#_Toc134015869)

[2.24 No1027. Longest Arithmetic Subsequence 14](#_Toc134015870)

[2.25 No1033. Moving Stones Until Consecutive 14](#_Toc134015871)

[2.26 No1041. Robot Bounded In Circle 15](#_Toc134015872)

[2.27 No1042. Flower Planting With No Adjacent 15](#_Toc134015873)

[2.27.1 深度优先路径搜索? 15](#_Toc134015874)

[2.27.2 贪心法，颜色标记 16](#_Toc134015875)

[2.28 No1043. Partition Array for Maximum Sum 16](#_Toc134015876)

[2.29 No1105. Filling Bookcase Shelves 17](#_Toc134015877)

[2.30 No1125. 最小的必要团队 18](#_Toc134015878)

[2.31 No1157. Online Majority Element In Subarray 18](#_Toc134015879)

[2.31.1 哈希方法 18](#_Toc134015880)

[2.31.2 找绝对众数 18](#_Toc134015881)

[2.31.3 随机化+二分查找 19](#_Toc134015882)

[2.31.4 摩尔投票+线段树 19](#_Toc134015883)

[2.32 No1161. Maximum Level Sum of a Binary Tree 19](#_Toc134015884)

[2.33 No1163. Last Substring in Lexicographical Order 19](#_Toc134015885)

[2.33.1 动态规划 19](#_Toc134015886)

[2.33.2 双指针 19](#_Toc134015887)

[2.34 No1172. Dinner Plate Stacks 19](#_Toc134015888)

[2.35 No1182. Shortest Distance to Target Color 20](#_Toc134015889)

[2.36 No1187. Make Array Strictly Increasing 20](#_Toc134015890)

[2.37 No1290. 二进制链表转整数 21](#_Toc134015891)

[2.38 No1550. 存在三个连续奇数的数组 22](#_Toc134015892)

[2.39 No1590. 使数组和能被 P 整除 22](#_Toc134015893)

[2.40 No1605. 给定行和列的和求可行矩阵 22](#_Toc134015894)

[2.41 No1615. 最大网络秩 23](#_Toc134015895)

[2.42 No1616.分割两个字符串得到回文串 24](#_Toc134015896)

[2.43 No1625.执行操作后字典序最小的字符串 25](#_Toc134015897)

[2.44 No1626. 无矛盾的最佳球队 26](#_Toc134015898)

[2.45 No1628. Expression Tree With Evaluate Function 27](#_Toc134015899)

[2.46 No1630. 等差子数组 27](#_Toc134015900)

[2.47 No1638. 统计只差一个字符的子串数目 27](#_Toc134015901)

[2.48 No1653.使字符串平衡的最少删除次数 27](#_Toc134015902)

[2.49 No2379.得到 K 个黑块的最少涂色次数 28](#_Toc134015903)

[2.50 No2383. 赢得比赛需要的最少训练时长 28](#_Toc134015904)

[2.51 No2389. 和有限的最长子序列 28](#_Toc134015905)

[2.52 No2395. 和相等的子数组 28](#_Toc134015906)

[2.53 No2399. 检查相同字母间的距离 28](#_Toc134015907)

[2.54 No2404. Most Frequent Even Element 29](#_Toc134015908)

[2.55 No2413. Smallest Even Multiple 29](#_Toc134015909)

[2.56 No2418. Sort the People 29](#_Toc134015910)

[2.57 No2423. Remove Letter To Equalize Frequency 29](#_Toc134015911)

[2.58 No2469. 温度转换--simple 30](#_Toc134015912)

[2.59 No2488. 统计中位数为 K 的子数组 30](#_Toc134015913)

[2.60 No2518. Number of Great Partitions 30](#_Toc134015914)

[3. 剑指offer 30](#_Toc134015915)

[3.1 47: 礼物的最大价值 30](#_Toc134015916)

[3.1.1 基本的动态规划问题 30](#_Toc134015917)

[3.1.2 为什么可以动态规划？ 30](#_Toc134015918)

[3.1.3 内存优化：滚动数组 30](#_Toc134015919)

[3.1.4 代码优化：边界处理优化 31](#_Toc134015920)

[3.1.5 总共有多少种可能的路径？ 31](#_Toc134015921)

[4. LCCI《程序员面试金典(第6版)》 31](#_Toc134015922)

[4.1 17.05：字母与数字 31](#_Toc134015923)

[5. 常用算法、数据结构和技巧 31](#_Toc134015924)

[5.1 位操作 31](#_Toc134015925)

[5.2 先排序后处理 31](#_Toc134015926)

[5.3 二分法: bisect() 32](#_Toc134015927)

[5.4 贪心法 32](#_Toc134015928)

[5.5 颜色标记 32](#_Toc134015929)

[5.6 哈希方法 32](#_Toc134015930)

[5.7 树搜索 32](#_Toc134015931)

[5.8 前缀和+哈希表 32](#_Toc134015932)

[5.9 动态规划Dynamic Programming 32](#_Toc134015933)

[5.9.1 0/1背包问题 32](#_Toc134015934)

[5.9.2 数位DP 32](#_Toc134015935)

[5.10 滚动数组 33](#_Toc134015936)

[5.11 滑动窗 33](#_Toc134015937)

[5.12 双指针 33](#_Toc134015938)

[5.13 DFS 33](#_Toc134015939)

[5.14 BFS，层序遍历 33](#_Toc134015940)

[5.15 摩尔投票 33](#_Toc134015941)

[5.16 线段树 33](#_Toc134015942)

[5.17 有序集合 33](#_Toc134015943)

[5.18 有限状态机 34](#_Toc134015944)

[5.19 栈 34](#_Toc134015945)

[6. 常用数学技巧 34](#_Toc134015946)

[6.1 众数，绝对众数 34](#_Toc134015947)

[6.2 裴蜀定理 34](#_Toc134015948)

[6.3 随机化 34](#_Toc134015949)

**[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（subarray）：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

子串（substring）：与子数组相似，都强调连续的。只不过子串是针对字符串的

子序列：不要求连续。从原序列中删除若干元素，留下的元素保留原来的顺序

# leetcode

## No0001. 两数之和

暴力方法进行遍历的话需要O(n^2)的时间复杂度。

【解法1】

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

## No0003. 无重复字符的最长子串

## No0011. Container With Most Water

求解目标：Area = Max(min(height[i], height[j]) \* (j-i)) {0 <= i < j < height,size()}

双指针：

基本的表达式: area = min(height[i], height[j]) \* (j - i)。使用两个指针分别指示区间左右两端，初始化为整个区间的两端。每次将值小的指针向内移动。

原因是：

面积取决于指针的距离与值小的值乘积，如果值大的指针向内移动，距离一定减小，而区间的高则一定不会大于较小的值，因此面积一定减小。反过来，值小的指针向内移动的话，距离也一定在减小，面积有可能增大也有可能减小。在遍历过程中记录出现过的最大的值即可。因为始终都是值小的指针向内移动的话，最多移动N次，复杂度即为O(N).

执行用时：108 ms, 在所有 Python3 提交中击败了93.91%的用户

内存消耗：25.7 MB, 在所有 Python3 提交中击败了48.00%的用户

但是，为什么这样挪动指针不会错过最大的区间？证明如下：

若暴力枚举，水槽两板围成面积 S(i,j) 的状态总数为 C(n,2) 。假设状态 S(i,j) 下 h[i]<h[j] ，在向内移动短板至 S(i+1,j) ，则相当于消去了 S(i,j−1),S(i,j−2),...,S(i,i+1) 状态集合。而所有消去状态的面积一定都小于当前面积（即 <S(i,j)），因为这些状态：

短板高度：相比 S(i,j) 相同或更短（即 ≤h[i] ）；

底边宽度：相比 S(i,j) 更短；

因此，每轮向内移动短板，所有消去的状态都不会导致面积最大值丢失，证毕。

## No0012. Integer to Roman

## 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

## No0[279. Perfect Squares](https://leetcode.cn/problems/perfect-squares/)

【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%。

## No0436. 寻找右区间

暴力搜索的话是O(n\*\*2)的时间复杂度。

可以对各区间的起始位置进行排序，以便于利用二分法进行搜索。

为了由起始位置恢复所对于的原始位置，可以建立{start: index}的哈希表。

Tips: bisect.bisect\_left()的使用，参考31.3

## 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：

## No970. Powerful Integers

方法1：

扫描x^0,x^1..,，针对每个x^i再扫描y^0, y^1,...搜索满足条件和数。

这种方法中显然y的幂的计算有很多重复。所以提交结果为time-out。

方法2：

先分别求小于等于bound的x和y的幂，得到xpowLst和ypowLst。针对两张表进行求和遍历找满足条件的和数，然后还要进行去重处理。

## 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



## No1003. Check If Word Is Valid After Substitutions

第一感是从s出发基于贪婪策略做减法。找到‘abc’就抠掉，直到最后剩下空字符串即表明s是有效的，否则是无无效的。

贪婪策略能行吗？直感能行。S中有‘abc’的话，必然是作为一个整体插入进来的，因为反方向处理时作为一个整体扣掉是可行的。

暴力方法需要反复扫描。每一轮扫描清除掉当前状态下的‘abc’子串。每一轮扫描处理完的字符串作为下一轮的输入。如果，真正地删除‘abc’子串的话可能会有比较大的处理开销。一种替代方案是设置一个flag数组，标记被删除的字符（而不是真正的删除）。

### 方法1：有限状态机

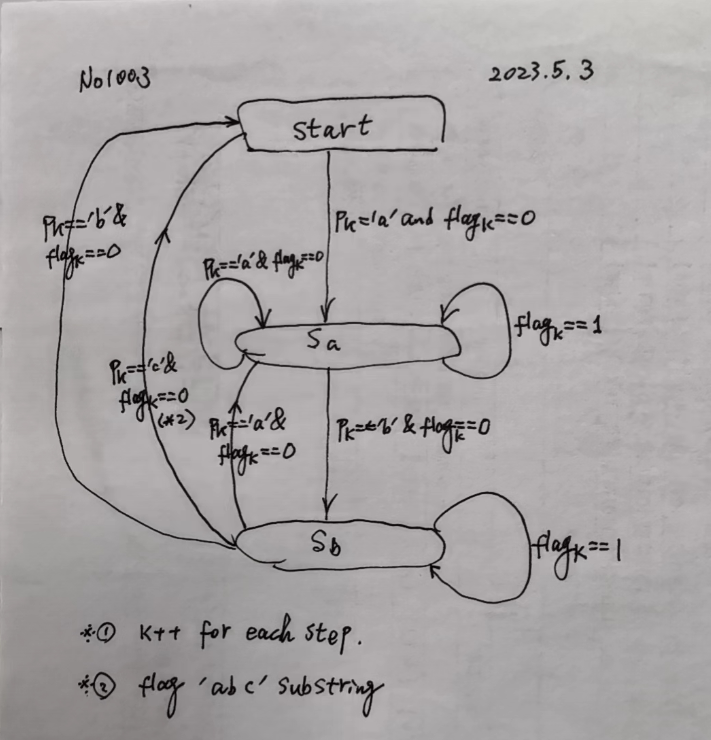


图 1 No1003 有限状态机

该状态机表示某一轮处理。需要反复多轮处理，直到最后所有位置全部标记为1，表示结果为True；或者某一轮处理后flag\_cnt没有变化,表示结果为False。

这一方法虽然可行，但是性能不如人意{5.31%,5.31%}

执行用时：168 ms, 在所有 Python3 提交中击败了5.31%的用户

内存消耗：16.3 MB, 在所有 Python3 提交中击败了5.31%的用户

### 方法2：栈

其实这个问题跟括号匹配问题相类似，可以基于栈来进行处理。

从s中逐个取字符，如果该字符为‘s’则查询栈顶两个字符，如果是’ab’则弹出；否则将当前字符入栈。最终栈变空则表明输入s为有效字符串，否则不是。

代码确实一下子变得非常简洁，但是性能却依然难看{13.27%,5.31%}

## No1012. 至少有 1 位重复的数字

给定正整数 n，返回在[1, n]范围内具有 **至少 1 位** 重复数字的正整数的个数。

暴力方法是遍历1~n针对每个数进行判断。

反过来可以考虑没有重复数字的数的个数。把每个数看作是由9个空位构成，往里填充数字所得（需要排除最左边连续为0的情况）。填充数字确保不重复的情况比较容易统计。🡪互斥原理

比如说9位数的话，相当于没有重复数字的情况就是将0~9无重复填入9个空格，是最左边不能为0。总共有9\*(9!)种。因此当n恰好为10\*\*k时这种统计非常简单。

但是题目要求的是任意的正整数n。需要考虑比如说当n=500时，在3位数这个数段比500大的数就不能纳入统计范围。本题作为difficult级别难就应该难在这里。

### 方法一：数位DP+记忆化搜索

从最高位开始填入各个数字，使用整数掩码 mask 记录前面已经填入过的数字（注意前缀 0 不计入已填入的数字）。假设当前填入第 i 位，如果前面填入的数字与 n 对应位置的数字相同，那么可选的填入数字小于等于 n 在第 i 位的数字，否则可填入全部数字。

记可填入的最大数字为 t，依次尝试填入数字 k∈[0,t]，如果 k 已经出现在 mask 中，那么说明填入数字 k 不合法，否则说明可以填入数字 k，那么尝试填入第 i+1 位的数字。

在填入第 i+1 位的数字时，如果掩码 mask\_i =0 且 k=0 成立，那么说明前面都是前缀 0，掩码 mask\_{i+1}为0，否则 mask\_{i+1}等于 mask\_i 在第 k 位设为一后的值。如果在填入第 i 位时，前面填入的数字与 n 对应位置的数字相同，且在第 i 位填入的数字为 t，那么填入第 i+1 位时，前面填入的数字也与 n 对应位置的数字相同。

注意到，假设当前需要填入第 i 位，且前面填入的数字与 n 对应位置的数字不相同，那么需要求得的不重复数字的正整数个数只与 mask 相关，我们可以使用备忘录 dp 记录该结果，避免重复计算。

## No1019. Next Greater Node In Linked List

暴力方法就是针对每一个节点进行查询。显然，如果像0-1-3-2-4-...中3的后面是2后面再跟上4，对应于3和2的结果都是4，不需要针对3和2重新查询。更复杂一点的情况是0-1-5-2-4-6...2对应的结果是4，而5对应的结果是6。。。有点类似于括号匹配的问题的既视感？

考虑用堆栈的方式实现。栈中存储(节点的原始序号即其值)即可。

从链表中取一个新的节点，如果该节点值比栈顶节点大，则取出栈顶节点并为它赋输出结果，继续取下一个，直到找到比该节点值大的栈顶值或者把栈取空了；然后，将该节点也加入堆栈。第一个节点直接加入栈。

结果列表赋初值全零（代码实现中，因为不知道链表长度，不能先初始化一个等长全零列表。而是每取出一个节点就给结果列表添一个零相当于对应于该节点的零初始化）。

性能不是很理想，如何优化？

执行用时：200 ms, 在所有 Python3 提交中击败了54.05%的用户

内存消耗：20.6 MB, 在所有 Python3 提交中击败了5.40%的用户

## No1023. Camelcase Matching

是不是贪心就可以了呢？

比如说，从左到右遍历pattern中的字符，逐个在query中寻找匹配的字符，但是query中的指针不能走回头路。

执行用时：60 ms, 在所有 Python3 提交中击败了7.06%的用户

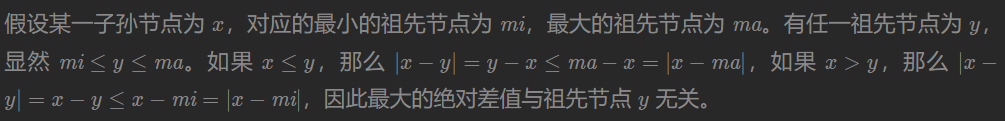
内存消耗：14.8 MB, 在所有 Python3 提交中击败了89.41%的用户

## No1026. Maximum Difference Between Node and Ancestor

题目要求找出所有祖先节点与它的子孙节点的绝对差值的最大值。按照枚举的思路，我们可以枚举子孙节点，然后找出它的所有祖先节点，计算绝对差值。同样地，我们也可以枚举祖先节点，然后找出它的所有子孙节点，计算绝对差值。

以第一种思路为例，并非所有祖先节点都需要被考虑到，我们只需要获取最小的祖先节点以及最大的祖先节点。我们对二叉树执行深度优先搜索，并且记录搜索路径上的节点的最小值 mi与最大值 ma。假设当前搜索的节点值为 val，那么与该子孙节点与它的所有祖先节点的绝对差值最大值为 max(∣val−mi∣,∣val−ma∣)，搜索该节点的左子树与右子树时，对应的 mi=min(mi,val)，ma=max(ma,val)。

为什么只需要获取最小的祖先节点以及最大的祖先节点？



第二种思路是否可行？可行，需要返回当前子树的最小值和最大值，方法类似。

采用基于显式递归的深度优先搜索算法实现。这个也可以看作是动态规划吗？

## No1027. Longest Arithmetic Subsequence

找输入序列的最长等差子序列。

等差数列最关键属性是公差。考虑dp[i][d]表示以nums[i]结尾公差为d的子序列长度。

## No1033. Moving Stones Until Consecutive

贪心法

我们可以假设 x,y,z 分别为从小到大排序后的 a,b,c，来讨论最小和最大移动次数。

当三枚石子连续放置时，即 (y−x)=1 并且 (z−y)=1，不需要额外移动，最小移动次数为 0。

当三枚石子中有两枚是连续放置，或者间隔为 1 时，我们只需对另外一枚石子移动一次，最小移动次数为 1。对于其他情况，我们最小需要移动 2 次。

对于最多移动次数，我们可以考虑将 x 向右（增加 1），或者将 z 向左（减小 1），每次移动都会使得两侧的距离减小 1，所以最多可以移动 z−x−2 次。

## No1041. Robot Bounded In Circle

表达机器人的状态需要两个信息：坐标和方向。

是否存在重复执行N次输入指令组才恢复初始状态的情况呢？存在，例1就是这样的。这种情况如何判定呢？

判断机器人是不是会绕圈就看机器人执行完一组指令（或者若干组指令）后是否恢复了初始状态。

“Circle”是否是严格意义的环路呢？比如说走“8”字形算不算绕圈？算。

**Insight**：

最多4次重复执行给定的指令组。如果最多4次重复还回不到初始状态就肯定回不去了。待证明。

并不一定要运行到回到初始状态才能做最终判断。经过若干次重复执行后能够回到原点，即便方向与初始方向不同也可以判定能够回到初始状态。

## No1042. Flower Planting With No Adjacent

### 深度优先路径搜索?

典型的图搜索问题。关键的是搜索树的构建。

由于只需要找到一种可行的结果（代表搜索树中的一条路径），所以应该使用深度优先路径搜索（使用栈 instead of 队列）。最终得到可行路径的深度就是花园的个数。

节点状态定义：可以用{garden, flower}来表示节点。假定花园总数为N，因为有4种花，所以总共有4N个节点。

如何记录路径信息？深度优先路径搜索过程中需要记忆路径信息，最后根据路径信息恢复出树的列表。用递归调用的方式来（利用函数调用栈）管理路径最方便。但是，对于搜索路径很深（本问题的搜索路径深度就相当于花园个数）的场合不合适。解决方案是，在节点状态中追加父节点信息。最后通过回溯的方式来恢复路径。在本问题中，还需要判断是不是已经给所有的花园都安排了树，所以给节点状态中再追加一个layer信息。当前访问节点的layer等于花园数时，即表示搜索成功了。

如何进行visited管理呢？

初始化：

栈的初始化，visited的初始化（初始化为dict，以方便查询）

根节点（空节点）入栈 # (garden=0, flower=0, parent=0, layer=0)

While 栈不为空：

Node(garden, flower, parent, layer) 🡨 stack.pop()

If layer == 花园个数：

通过回溯的方式得到可行的树列表并返回

遍历Node的邻节点，for each neighbor node:

如果(g, f, p)不在visited中，入栈: stack.append((g, f, p,layer+1))

Assert(0) ## 题目确保一定有解，所以不应到这儿来。到达这儿表示出错

好像行不通，暂时保留以后接着想。

### 贪心法，颜色标记

由于每个花园最多有 3 条路径可以进入或离开，这就说明每个花园最多有 3 个花园与之相邻，而每个花园可选的种植种类有 4 种，这就保证一定存在合法的种植方案满足题目要求。花园中种植不同的花可以视为每个花园只能标记为给定的4种颜色为 1,2,3,4 中的一种，初始化时我们可以为每个花园标记为颜色 0。对于第 i 个花园，统计其周围的花园已经被标记的颜色，然后从未标记的颜色中选一种颜色给其标记即可。整体标记过程如下：

首先建立整个图的邻接列表 adj ；

初始化时，将每个花园节点的颜色全部标记为 0；

遍历每个花园，并统计其相邻的花园的颜色标记，并从未被标记的颜色中找到一种颜色给当前的花园进行标记；

返回所有花园的颜色标记方案即可。

## No1043. Partition Array for Maximum Sum

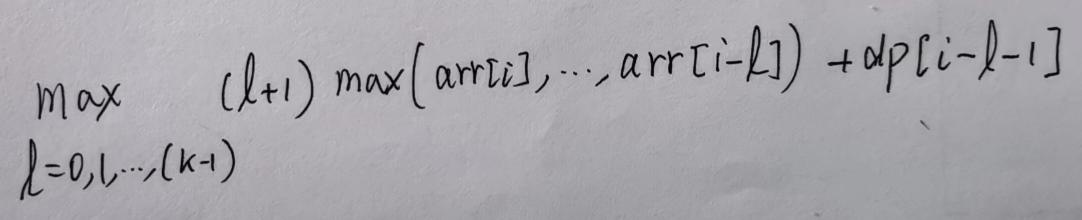
动态规划。

以dp[i]代表arr[0]...arr[i]子数组满足题设条件的最大和。

显然，当i<=k时，dp[i]就等于取前i+1个数中最大值再乘以(i+1)即得。

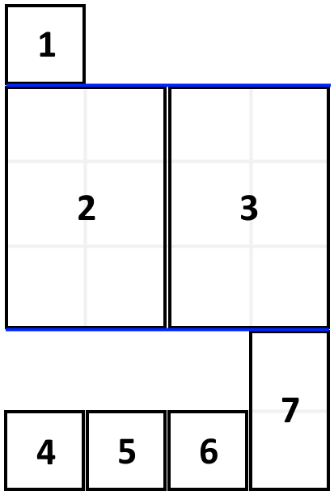
假设已经有了dp[0,...,i](i>=k)，以下推到关于dp[i+1]的状态转移方程。

针对arr[i]，与它相关的分组方案有k种，即{arr[i]}, {arr[i],arr[i-1]},..., {arr[i],arr[i-1]... ,arr[i-k+1]}，由此可以得到：



## No1105. Filling Bookcase Shelves

Example：Input: books = [[1,1],[2,3],[2,3],[1,1],[1,1],[1,1],[1,2]], shelfWidth = 4. Output: 6



第一感：(1) 很像一个俄罗斯方块游戏；(2) 动态规划

然而并不是。。。如果按照类似于俄罗斯方块的规则的话，book2应该放在右上角，book3在book1下面靠左，然后book7可以放在book2的下面。这样总高度只需要5即可。

差异的要点在于，每一层的高度由该层所放的书的最大高度决定，下一层必须完全该层的绝对下方！（当然，其实应该倒过来看才符合自然感觉）。

以dp[i]表示放置前(i)本书时所占据的最小高度。以上例子中，显然有dp[1]=1, dp[2]=3,dp[3]=4... 但是需要注意到，dp[3]和dp[2]所对应的books[1]的放置位置不同（当然会出现不同的情况，否则的话只要按贪心的方式摆放就可以了）。也就是说，dp[i+1]不仅仅依赖于dp[i]的方案，而是要综合考虑dp[0]~dp[i]的方案。

Books[0]的摆放没有选择余地；

Books[1]可以摆放在第1排，也可以摆放在第2排。。。

根据题意，按顺序将这些书摆放到总宽度为 shelfWidth 的书架上。先选几本书放在书架上，然后再建一层书架。重复这个过程，直到把所有的书都放在书架上。

考虑dp[i] 来表示放下前 i 本书所用的最小高度。 因为最多 1000 本书， 每本书高度最大 1000，我们可以把 dp[i] 初始化为 1000000（python中也可以初始化为math.inf或者float(‘inf’)）， 初始化 dp[0] 为零，表示没放书时高度为零。

当我们要放置前 i 本书时候，假定前 j 本书放在上面的书架上，其中 j<i, 前 j 本书放好后剩余的书放在最后一层书架上, 这一层书架的高度是这部分书的高度最大值，由此得到如此递推公式：

dp[i]=min(dp[j]+max(books[k]))

其中，0≤j≤k<i≤n,∑books[k]≤shelfWidth

我们循环遍历 i, 求出 dp[i] 的值（通俗地说就是正向的填表法），最后返回 dp[n] 为最终答案。

## No1125. 最小的必要团队

最小集合覆盖目标的问题。

有没有背包问题的味道？对于每个people有选用和弃用两种选择，所以是0/1-背包问题？

状态压缩：将所需技能映射为数字(1<<k)。各people技能集合可以将各技能值进行按位或操作后用一个数值表示。判断某个peoples集合是否能够覆盖，只需要将他们的技能值进行按位或操作，然后与req\_skills进行按位与后看结果是否等于req\_skills即可。

动态规划

采用自下而上的「动态规划」的思路，用 dp[i] 来表示状态，状态含义是满足技能集合为 i 的最小人数的数组。初始化状态是 dp[0]，为空数组，因为如果不需要任何技能，不用任何人就可以完成。

我们首先依次遍历 peoples，求出当前这个人所有的技能集合 cur\_skill。然后遍历 dp 表中的结果 dp[prev]，其中原来的技能集合用 prev 来表示。设加入当前这个人后新的技能集合是 comb，由原来的技能集合和当前技能集合求并集后，可以得到：comb=prev ∣ cur\_skill。状态转移的规则是：

如果 dp[comb] 不存在，或 dp[prev] 的长度加上 1 小于 dp[comb].size()，那么我们就需要更新 dp[comb] 为 dp[prev]，再将当前人加入到 dp[comb]。如果 dp[comb] 已经存在，不必更新dp。

这里我们更新 dp[comb] 时候，可以采用直接覆盖的方式，因为更新后的结果因为已经包含了当前员工的技能，所以不会再次满足转移规则，而发生重复转移。

最后，所有所需技能的集合用 (1<<n)−1 来表示，其中 n 是 req\_skills 的长度，我们只需要返回最终答案 dp[(1<<n)−1]。

## No1157. Online Majority Element In Subarray

题目要求其实找其中某个出现次数不低于指定门限的元素。

### 哈希方法

针对单个query的话比较简单，对arr进行扫描，对left和right之间的各元素出现次数进行统计，统计结果用哈希表进行维护。直到找到一个满足结果的即返回，否则返回-1.

### 找绝对众数

问题分为两个部分：

找出 可能的 绝对众数（不一定存在绝对众数，但存在绝对众数的话找到的一定是绝对众数）和统计这个数出现的次数。找可能的绝对众数方法很简单，暴力寻找即可（类似于打擂台，具体实现可以看代码，正确性证明略去）。统计次数也使用暴力统计。设数组长度为 n，询问次数为 q，则总时间复杂度为 O(nq)，无法通过。

题设条件“2 \* threshold > right - left + 1”保证了如果存在满足条件的数，一定是绝对众数。

### 随机化+二分查找

### 摩尔投票+线段树

## No1161. Maximum Level Sum of a Binary Tree

广度优先搜索，层序遍历。

由于需要统计每一层的总和，所以在进入队列时需要附上层号信息。

对于从队列中取出的元素进行滚动求和，但是每次切换层时，根据当前层的总和更新最大值，然后将sum复位归零开始新的一行的求和。可以用一个prev\_layer记录上一个元素的层号，用于行切换的判断。

## No1163. Last Substring in Lexicographical Order

Lexicographical Order: 编辑次序

编辑次序排在最后的子串，其首字母一定是原字符串中排序最后的字母，记为c。如果c是唯一的，问题就很简单。但是，比如说“abecdedf”，答案应该是‘edf’而不是‘ecdedf’。

从左往右扫描记录所碰到过的最大的字母记为cmax，并用sstr表示要求结果：

如果下一个字符比cmax还大，则更新cmax

如果下一个字符小于cmax，则将该字符加到sstr尾部

如果下一个字符等于cmax，则复位sstr还是继续添加到sstr尾部，取决于下一个字符。。。

### 动态规划

记dp[j]表示s[j:-1]中按编辑次序排在最后的子串。现在考虑dp[j-1]：

如果s[j-1]比dp[j][0]小，则：dp[j-1] = dp[j]

如果s[j-1]比dp[j][0]大，则：dp[j-1] = s[j-1:-1]

如果s[j-1]等于dp[j][0]，则：对比下一个字母。。。知道找到一个能够决出胜负的

虽然正确，然而超时。。。其实说是动态规划有点牵强了，不需要保留所有dp[j]，只需要存储结果，不管更新这个结果知道最后即可。换句话说dp[j-1]只跟dp[j]有关系，类似于马尔科夫性。

### 双指针

首先，如上所述，不需要存储所有的dp[j]。其次，其实不需要取s的子串，用一个指针即可。所以本题可以用双指针来解决。一个指针指向当前得到的最好结果的其实位置，；另一个指向当前遍历到的地方。两个指针均从右往左移动。

## No1172. Dinner Plate Stacks

算法思路：数组 + 有序集合模拟

用一个数组 stack 来模拟栈，编号为 index 的栈的顶部 stackTop 在数组中的下标 pos 可以通过公式来表示：pos=index×capacity+stackTop。用一个有序集合 poppedPos 来保存被方法 popAtStack 删除的位置。用数组 top 记录每个栈的栈顶元素在栈中的位置，比如 top[1]=2 就表示，编号为 1 的栈，栈顶元素在栈中的下标为 2（从 0 开始计数，capacity>2），即在这个栈中，它上面没有元素，它下面还有两个元素。

执行 push 时，先考虑 poppedPos 中的位置，如果非空，则找出最小的位置，把元素 push 到这个位置。如果为空，则往 stack 后追加，然后更新 top。

执行 popAtStack 时，先找出这个栈现在的栈顶位置，然后把这个位置的元素的下标计算出来并更新栈顶位置，把下标放入 poppedPos，返回元素的值。当然有可能这个栈是空的，此时要返回 −1。

执行 pop 时，情况比较复杂。直观的想法是，返回 stack 元素即可。但 stack 末尾元素可能早已被 popAtStack 删除，因此，应该返回处于 stack 末尾且不位于 popAtStack 中的位置，如果 stack 末尾位置出现在 popAtStack 中，直接把 stack 末尾元素删除，再进行重复判断，直到满足上述条件或者 stack 为空。找到符合条件的位置后，需要更新 top，然后返回元素的值。在执行 pop 时，上述判断可能会执行多次，但是一次 popAtStack 最多带来一次判断，因此不会带来时间复杂度的变大。

复杂度分析

时间复杂度：记 n 为 push 的调用次数。单次 push 的时间复杂度为 O(logn)，单次 popAtStack 的时间复杂度为 O(logn)，pop 单次均摊的时间复杂度为 O(logn)。

空间复杂度：使用到的数组和有序集合空间复杂度均为 O(n)。

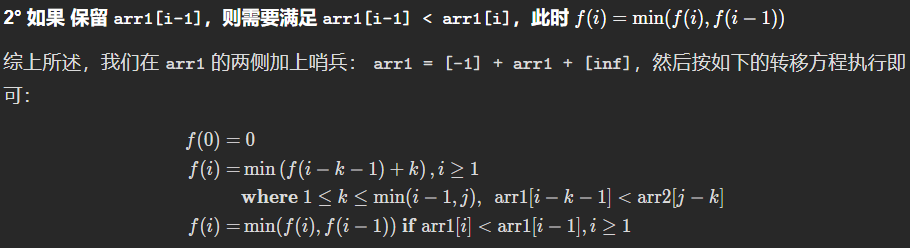
## No1182. Shortest Distance to Target Color

针对每个query，傻傻地向两边搜索对比？

建立一个颜色哈希表，这样每个query只需要哈希表中查询对比就快多了。由于哈希表中的针对每个颜色的序号数组是自然升序的，所以查询时可以用bisect模块来“偷懒”

## No1187. Make Array Strictly Increasing





## 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

## No1625.执行操作后字典序最小的字符串

有没有什么确定性的办法使得每次都按字典序递减的方式进行操作直到最后达到最小？

所给的示例中也不是按照字典序递减的方式进行操作，所以可能不是这种思路。

按照题设的定义，其实字典序的判断可以直接用两数相减而得。

如何判断已经达到字典序最小状态呢？

【解法1：官解】居然都没有看懂。





裴蜀定理，参见30.1

【解法2】

广度优先搜索+模拟+记忆化搜索

## No1626. 无矛盾的最佳球队

假设你是球队的经理。对于即将到来的锦标赛，你想组合一支总体得分最高的球队。球队的得分是球队中所有球员的分数 总和 。然而，球队中的矛盾会限制球员的发挥，所以必须选出一支 没有矛盾 的球队。如果一名年龄较小球员的分数 严格大于 一名年龄较大的球员，则存在矛盾。同龄球员之间不会发生矛盾。给你两个列表 scores 和 ages，其中每组 scores[i] 和 ages[i] 表示第 i 名球员的分数和年龄。请你返回 所有可能的无矛盾球队中得分最高那支的分数 。

目标：总分值最高；

约束：年龄小的球员的分数不能大于比他年龄大的球员的分数

第一感，这是一个动态规划问题，有点背包问题的影子。对于每个球员来说，是采用和不采用两种情况，因此可以看作是0-1背包问题。

先对年龄进行排序。对scores也按相同的方式重排。

首先对数组进行排序，先按年龄升序，如果年龄相同，再按分数升序。设dp[i]为在球员0~i中选出球队（包括选中球员i）能获得的最大分数。状态转移方程如下：

 dp[i]=max{dp[j]}+score[i]（for all dp[j]所代表序列与i不冲突）

考虑不冲突的限制：由于是按年龄升序的，这意味着如果要选出球员i，要求scores[i]不小于前面选出的每一个。

而这个约束并不需要刻意维护前面序列的scores最大值，因为scores[j]同样要不小于前面的每一个，所以只要满足scores[i]>scores[j]即可。上面的i，j只是为了表述方便，代码实现中要改成index[i]和index[j]。

首先我们将所有队员按照分数升序进行排序，分数相同时，则按照年龄升序进行排序，我们用数组 people[n][2] 来表示排序后的n 名球员信息，其中 people[i][0]，people[i][1] 分别为排序后第 i 名球员的分数和年龄。然后我们可以用动态规划来解决该问题，设 dp[i] 为我们最后组建的球队中的最大球员序号为排序后的第 i 名球员时的球队最大分数（此时的球员序号为排序后的新序号），因为我们是按照分数升序排序的，所以最后组建球队的最后一名球员的分数一定不会小于队伍中该球员前面一名球员的分数，所以为了避免矛盾的产生我们只需要让最后组建球队的最后一名球员的年龄不小于该球员前面一名球员的年龄即可，那么转移的方程如下：

dp[i]=max{dp[j]}+people[i][0],j<i&people[j][1]≤people[i][1] (1)

上文讨论的是建立在 i>0 的前提上的，我们还需要考虑动态规划的边界条件，当 i=0 时，只有一名球员 people[0]，此时该球员单独组成一只队伍有： dp[0]=people[0][0]。最后我们返回 max{dp[i],0≤i<n} 即为所有可能的无矛盾的球队的最高分数。

## No1628. Expression Tree With Evaluate Function

## No1630. 等差子数组

单独一对{l[i],r[i]}的确认很简单直观。

这一类问题的关键在于需要进行大量这种查询确认时，利用不同查询之间的相关联性，避免每次查询都从零开始，使得n次查询的时间复杂度不是在单次查询的时间复杂度上单纯地乘以n。

## No1638. 统计只差一个字符的子串数目

给你两个字符串 s 和 t ，请你找出 s 中的非空子串的数目，这些子串满足替换 一个不同字符 以后，是 t 串的子串。换言之，请你找到 s 和 t 串中 恰好 只有一个字符不同的子字符串对的数目。

比方说， "computer" and "computation" 只有一个字符不同： 'e'/'a' ，所以这一对子字符串会给答案加 1 。

请你返回满足上述条件的不同子字符串对数目。一个 子字符串 是一个字符串中连续的字符。

第一感，动态规划。可以看作是edit-distance问题的变形或者延申。

## 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。

## No2106. Maximum Fruits Harvested After at Most K Steps

方法一：二分法

方法二：滑动窗法

## 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。

## No2395. 和相等的子数组

前向遍历，将每两个连续数据的和存入一个哈希表。没得到一个新值在哈希表中查询，找到就返回True，直到最后都每找到的话返回False。

## No2399. 检查相同字母间的距离

用一个哈希表h记录每个字母第一次出现的序号。

从左到右搜索，如果该字母存在于h中，计算两次出现的距离之差是否与distance[i]相符。发现一次不相符即可以提出搜索。

## No2404. Most Frequent Even Element

第一感就是哈希表。

对数组进行遍历，是偶数的存入哈希表{nums: count}。最后对哈希表进行count最大值的搜索。需要注意，在出现次数相同时，要选其中最小的。

## No2413. Smallest Even Multiple

Too simple. 其实就是求lcm(2,n)，如果是偶数就返回n本身，否则返回2\*n.

## No2418. Sort the People

Step1: 创建哈希表{height: names}

Step2: 对heights进行逆序排序

Step3: 按排序后的heights中取从大到小每个height对应的names

## No2423. Remove Letter To Equalize Frequency

思路1：

需要考虑以下几种情况：

1. 各字符个数全1;
2. 只有一种字符；
3. 只有一个1，其余全部相等
4. 有一种字符的个数比其它字符个数（均相等）多一个

思路2：枚举+哈希表

在暴力方法中，我们每次枚举要删除的字符后，都要重新统计所有字符出现的频率，造成了重复的运算。这里我们可以使用哈希表进行优化，在枚举之前，先统计不同字符频率的频率。

假如一个字符出现的频率是 c，那这个字符删除一个后，频率就从 c 变成了 c−1。我们只需要更新哈希表中 c 和 c−1 的频率，再判断哈希表的大小是否为一就可以了。如果哈希表大小为 1，则说明剩余每个字母出现频率相同，我们直接返回 true，反之说明删除当前字符不可行，我们把这个字符的频率就从

c−1 变成 c，更新哈希表进行还原。

要注意更新哈希表后，如果一个键对 ⟨key,value⟩ 中 value=0，我们需要手动删除 key。

最后，当我们尝试过所有不同字符后，还没有找到能删除的字符，使得满足要求，我们返回 false。

## No2469. 温度转换--simple

## No2488. 统计中位数为 K 的子数组

## No2518. Number of Great Partitions

# 剑指offer

## 47: 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

状态压缩

No1125

## 先排序后处理

No2389, No0001

## 二分法: bisect()

No2389, No436

Python library: bisect

## 贪心法

No1042, No1033

## 颜色标记

No1042

## 哈希方法

Python中哈希表可以用字典dict或者集合set实现，取决于是需要存储key-value对还是只需要存储key。

No2395, No0001, No1157

## 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

## 动态规划Dynamic Programming

No1125（自下而上的动态规划思路），No1043, No1027, No1105

### 0/1背包问题

### 数位DP

[数位 DP - OI Wiki (oi-wiki.org)](https://oi-wiki.org/dp/number/)

例1：给定两个正整数a,b ，求在[a,b] 中的所有整数中，每个数码（digit）各出现了多少次。

Leetcode: No233, No600, No1012.

## 滚动数组

## 滑动窗

## 双指针

No1616，No0011

## DFS

No1026,

## BFS，层序遍历

No1161

BFS 算法组成的 3 元素：队列，节点表示，已访问的集合（为了避免队列中插入重复的值）。

插入队列中的有时除了节点以外，还需要层序号等信息，比如No1161。

BFS基本算法流程：

初始化三元素

根节点或者空节点入队列

While (队列不为空):

从队列中取出一个节点node

遍历node的邻节点，如果不在visited集合则加入队列

## 摩尔投票

No1157,

## 线段树

No1157,

## 有序集合

No1172

## 有限状态机

No1003

## 栈

No1003

# 常用数学技巧

## 众数，绝对众数

No1157

## 裴蜀定理

[裴蜀定理 - OI Wiki (oi-wiki.org)](https://oi-wiki.org/math/number-theory/bezouts/)

裴蜀定理，又称贝祖定理（Bézout's lemma）。是一个关于最大公约数的定理。其内容是：

设a,b是不全为零的整数，则存在整数x,y, 使得 ax+by=gcd(a,b).

No1625

## 随机化

No1157

[Reference]

[Revision history]