

Fighting for dream !

xjtu-zhongyingLi
2018-11-07

目录

1. 两数之和
2. 两数相加
3. 无重复字符的最长子串
4. 两个排序数组的中位数
5. 最长回文子串
6. Z字形变换
7. 反转整数
8. 字符串转整数 (atoi)
9. 回文数
10. 正则表达式匹配
11. 盛最多水的容器
12. 整数转罗马数字
13. 罗马数字转整数
14. 最长公共前缀
15. 三数之和
16. 最接近的三数之和
17. 电话号码的字母组合
18. 四数之和
19. 删除链表的倒数第N个节点
20. 有效的括号
21. 合并两个有序链表
22. 括号生成
23. 合并K个排序链表
24. 两两交换链表中的节点
25. k个一组翻转链表
26. 删除排序数组中的重复项
27. 移除元素
28. 实现strStr()
29. 两数相除
30. 与所有单词相关联的字串
31. 下一个排列
32. 最长有效括号
33. 搜索旋转排序数组
34. 在排序数组中查找元素的第一个和最后一个位置
35. 搜索插入位置
36. 有效的数独
37. 解数独
38. 报数
39. 组合总和
40. 组合总和 II
41. 缺失的第一个正数
42. 接雨水
43. 字符串相乘
44. 通配符匹配
45. 跳跃游戏 II
46. 全排列

- 47. 全排列 II
- 48. 旋转图像
- 49. 字母异位词分组
- 50. Pow(x, n)
- 51. N皇后
- 52. N皇后 II
- 53. 最大子序和
- 54. 螺旋矩阵
- 55. 跳跃游戏
- 56. 合并区间
- 57. 插入区间
- 58. 最后一个单词的长度
- 59. 螺旋矩阵 II
- 60. 第k个排列
- 61. 旋转链表
- 62. 不同的路径
- 63. 不同的路径之二
- 64. 最小路径和
- 65. 验证数字
- 66. 加一运算
- 67. 二进制数相加
- 68. 文本左右对齐
- 69. 求平方根
- 70. 爬楼梯
- 71. 简化路径
- 72. 编辑距离
- 73. 矩阵赋零
- 74. 搜索一个二维矩阵
- 75. 颜色排序
- 76. 最小窗口子串
- 77. 组合项
- 78. 子集合
- 79. 词语搜索
- 80. 有序数组中去除重复项之二
- 81. 在旋转有序数组中搜索之二
- 82. 移除有序链表中的重复项之二
- 83. 移除有序链表中的重复项
- 84. 直方图中最大的矩形
- 85. 最大矩形
- 86. 划分链表
- 87. 爬行字符串
- 88. 混合插入有序数组
- 89. 格雷码
- 90. 子集合之二
- 91. 解码方法
- 92. 倒置链表之二
- 93. 复原IP地址
- 94. 二叉树的中序遍历

95. 独一无二的二叉搜索树之二

96. 独一无二的二叉搜索树

97. 交织相错的字符串

98. 验证二叉搜索树

99. 复原二叉搜索树

100. 判断相同树

101. 判断对称树

102. 二叉树层序遍历

103. 二叉树的之字形层序遍历

104. 二叉树的最大深度

105. 由先序和中序遍历建立二叉树

106. 由中序和后序遍历建立二叉树

107. 二叉树层序遍历之二

108. 将有序数组转为二叉搜索树

109. 将有序链表转为二叉搜索树

110. 平衡二叉树

111. 二叉树的最小深度

112. 二叉树的路径和

113. 二叉树路径之和之二

114. 将二叉树展开成链表

115. 不同的子序列

116. 每个节点的右向指针

117. 每个节点的右向指针之二

118. 杨辉三角

119. 杨辉三角之二

120. 三角形

121. 买卖股票的最佳时间

122. 买股票的最佳时间之二

123. 买股票的最佳时间之三

124. 求二叉树的最大路径和

125. 验证回文字符串

126. 词语阶梯之二

127. 词语阶梯

128. 求最长连续序列

129. 求根到叶节点数字之和

130. 包围区域

131. 拆分回文串

132. 拆分回文串之二

133. 无向图的复制

134. 加油站问题

135. 分糖果问题

136. 单独的数字

137. 单独的数字之二

138. 拷贝带有随机指针的链表

139. 拆分词句

140. 拆分词句之二

141. 单链表中的环

142. 单链表中的环之二

- 143. 链表重排序
- 144. 二叉树的先序遍历
- 145. 二叉树的后序遍历
- 146. 最近最少使用页面置换缓存器
- 147. 链表插入排序
- 148. 链表排序
- 149. 共线点个数
- 150. 计算逆波兰表达式
- 151. 翻转字符串中的单词
- 152. 求最大子数组乘积
- 153. 寻找旋转有序数组的最小值
- 154. 寻找旋转有序数组的最小值之二
- 155. 最小栈
- 156. 二叉树的上下颠倒
- 157. 用Read4来读取N个字符
- 158. 用Read4来读取N个字符之二 - 多次调用
- 159. 最多有两个不同字符的最长子串
- 160. 求两个链表的交点
- 161. 一个编辑距离
- 162. 求数组的局部峰值
- 163. 缺失区间
- 164. 求最大间距
- 165. 版本比较
- 166. 分数转循环小数
- 167. 两数之和之二 - 输入数组有序
- 168. 求Excel表列名称
- 169. 求众数
- 170. 两数之和之三 - 数据结构设计
- 171. 求Excel表列序号
- 172. 求阶乘末尾零的个数
- 173. 二叉搜索树迭代器
- 174. 地牢游戏
- 175. 联合两表
- 176. 第二高薪水
- 177. 第N高薪水
- 178. 分数排行
- 179. 最大组合数
- 180. 连续的数字
- 181. 员工挣得比经理多
- 182. 重复的邮箱
- 183. 从未下单订购的顾客
- 184. 系里最高薪水
- 185. 系里前三高薪水
- 186. 翻转字符串中的单词之二
- 187. 求重复的DNA序列
- 188. 买卖股票的最佳时间之四
- 189. 旋转数组
- 190. 翻转位

- 191. 位1的个数
- 192. 单词频率
- 193. 验证电话号码
- 194. 转置文件
- 195. 第十行
- 196. 删除重复邮箱
- 197. 上升温度
- 198. 打家劫舍
- 199. 二叉树的右侧视图
- 200. 岛屿的数量
- 201. 数字范围位相与
- 202. 快乐数
- 203. 移除链表元素
- 204. 质数的个数
- 205. 同构字符串
- 206. 倒置链表
- 207. 课程清单
- 208. 实现字典树(前缀树)
- 209. 最短子数组之和
- 210. 课程清单之二
- 211. 添加和查找单词-数据结构设计
- 212. 词语搜索之二
- 213. 打家劫舍之二
- 214. 最短回文串
- 215. 数组中第k大的数字
- 216. 组合之和之三
- 217. 包含重复值
- 218. 天际线问题
- 219. 包含重复值之二
- 220. 包含重复值之三
- 221. 最大正方形
- 222. 求完全二叉树的节点个数
- 223. 矩形面积
- 224. 基本计算器
- 225. 用队列来实现栈
- 226. 翻转二叉树
- 227. 基本计算器之二
- 228. 总结区间
- 229. 求众数之二
- 230. 二叉搜索树中的第K小的元素
- 231. 判断2的次方数
- 232. 用栈来实现队列
- 233. 数字1的个数
- 234. 回文链表
- 235. 二叉搜索树的最小共同父节点
- 236. 二叉树的最小共同父节点
- 237. 删除链表的节点
- 238. 除本身之外的数组之积

- 239. 滑动窗口最大值
- 240. 搜索一个二维矩阵之二
- 241. 添加括号的不同方式
- 242. 验证变位词
- 243. 最短单词距离
- 244. 最短单词距离之二
- 245. 最短单词距离之三
- 246. 对称数
- 247. 对称数之二
- 248. 对称数之三
- 249. 群组偏移字符串
- 250. 计数相同值子树的个数
- 251. 压平二维向量
- 252. 会议室
- 253. 会议室之二
- 254. 因子组合
- 255. 验证二叉搜索树的先序序列
- 256. 粉刷房子
- 257. 二叉树路径
- 258. 加数字
- 259. 三数之和较小值
- 260. 单独的数字之三
- 261. 图验证树
- 262. 旅行和用户
- 263. 丑陋数
- 264. 丑陋数之二
- 265. 粉刷房子之二
- 266. 回文全排列
- 267. 回文全排列之二
- 268. 丢失的数字
- 269. 另类字典
- 270. 最近的二分搜索树的值
- 271. 加码解码字符串
- 272. 最近的二分搜索树的值之二
- 273. 整数转为英文单词
- 274. 求H指数
- 275. 求H指数之二
- 276. 粉刷篱笆
- 277. 寻找名人
- 278. 第一个坏版本
- 279. 完全平方数
- 280. 摆动排序
- 281. 之字形迭代器
- 282. 表达式增加操作符
- 283. 移动零
- 284. 顶端迭代器
- 285. 二叉搜索树中的中序后继节点
- 286. 墙和门

- 287. 寻找重复数
- 288. 独特的单词缩写
- 289. 生命游戏
- 290. 词语模式
- 291. 词语模式之二
- 292. 尼姆游戏
- 293. 翻转游戏
- 294. 翻转游戏之二
- 295. 找出数据流的中位数
- 296. 最佳开会地点
- 297. 二叉树的序列化和去序列化
- 298. 二叉树最长连续序列
- 299. 公母牛游戏
- 300. 最长递增子序列
- 301. 移除非法括号
- 302. 包含黑像素的最小矩阵
- 303. 区域和检索 - 不可变
- 304. 二维区域和检索 - 不可变
- 305. 岛屿的数量之二
- 306. 加法数
- 307. 区域和检索 - 可变
- 308. 二维区域和检索 - 可变
- 309. 买股票的最佳时间含冷冻期
- 310. 最小高度树
- 311. 稀疏矩阵相乘
- 312. 打气球游戏
- 313. 超级丑陋数
- 314. 二叉树的竖直遍历
- 315. 计算后面较小数字的个数
- 316. 移除重复字母
- 317. 建筑物的最短距离
- 318. 单词长度的最大积
- 319. 灯泡开关
- 320. 通用简写
- 321. 创建最大数
- 322. 硬币找零
- 323. 无向图中的连通区域的个数
- 324. 摆动排序之二
- 325. 最大子数组之和为k
- 326. 判断3的次方数
- 327. 区间和计数
- 328. 奇偶链表
- 329. 矩阵中的最长递增路径
- 330. 补丁数组
- 331. 验证二叉树的先序序列化
- 332. 重建行程单
- 333. 最大的二分搜索子树
- 334. 递增的三元子序列

- 335. 自交
- 336. 回文对
- 337. 打家劫舍之三
- 338. 计数位
- 339. 嵌套链表权重和
- 340. 最多有K个不同字符的最长子串
- 341. 压平嵌套链表迭代器
- 342. 判断4的次方数
- 343. 整数拆分
- 344. 翻转字符串
- 345. 翻转字符串中的元音字母
- 346. 从数据流中移动平均值
- 347. 前K个高频元素
- 348. 设计井字棋游戏
- 349. 两个数组相交
- 350. 两个数组相交之二
- 351. 安卓解锁模式
- 352. 分离区间的数据流
- 353. 设计贪吃蛇游戏
- 354. 俄罗斯娃娃信封
- 355. 设计推特
- 356. 直线对称
- 357. 计算各位不相同的数字个数
- 358. 按距离为k隔离重排字符串
- 359. 记录速率限制器
- 360. 变换数组排序
- 361. 炸弹人
- 362. 设计点击计数器
- 363. 最大矩阵和不超过K
- 364. 嵌套链表权重和之二
- 365. 水罐问题
- 366. 找二叉树的叶节点
- 367. 检验完全平方数
- 368. 最大可整除的子集合
- 369. 链表加一运算
- 370. 范围相加
- 371. 两数之和
- 372. 超级次方
- 373. 找和最小的K对数字
- 374. 猜数字大小
- 375. 猜数字大小之二
- 376. 摆动子序列
- 377. 组合之和之四
- 378. 有序矩阵中第K小的元素
- 379. 设计电话目录
- 380. 常数时间内插入删除和获得随机数
- 381. 常数时间内插入删除和获得随机数 - 允许重复
- 382. 链表随机节点

- 383. 赎金条
- 384. 数组洗牌
- 385. 迷你解析器
- 386. 字典顺序的数字
- 387. 字符串第一个不同字符
- 388. 最长的绝对文件路径
- 389. 寻找不同
- 390. 淘汰游戏
- 391. 完美矩形
- 392. 是子序列
- 393. 编码验证
- 394. 解码字符串
- 395. 至少有K个重复字符的最长子字符串
- 396. 旋转函数
- 397. 整数替换
- 398. 随机拾取序列
- 399. 求除法表达式的值
- 400. 第N位
- 401. 二进制表
- 402. 去掉K位数字
- 403. 青蛙过河
- 404. 左子叶之和
- 405. 数字转为十六进制
- 406. 根据高度重建队列
- 407. 收集雨水之二
- 408. 验证单词缩写
- 409. 最长回文串
- 410. 分割数组的最大值
- 411. 最短的独一无二的单词缩写
- 412. 嘶嘶嗡嗡
- 413. 算数切片
- 414. 第三大的数
- 415. 字符串相加
- 416. 相同子集和分割
- 417. 太平洋大西洋水流
- 418. 调整屏幕上的句子
- 419. 平板上的战船
- 420. 密码强度检查器
- 421. 数组中异或值最大的两个数字
- 422. 验证单词平方
- 423. 从英文中重建数字
- 424. 最长重复字符置换
- 425. 单词平方
- 426. Convert Binary Search Tree to Sorted Doubly Linked List
- 427. 建立四叉树
- 428. Serialize and Deserialize N-ary Tree
- 429. N叉树的层序遍历
- 430. 扁平化多级双向链表

431. Encode N-ary Tree to Binary Tree

432. 全O(1)的数据结构

433. 最小基因变化

434. 字符串中的分段数量

435. 非重叠区间

436. 找右区间

437. 二叉树的路径和之三

438. 找出字符串中所有的变位词

439. 三元表达式解析器

440. 字典顺序的第K小数字

441. 排列硬币

442. 找出数组中所有重复项

443. 字符串压缩

444. 序列重建

445. 两个数字相加之二

446. 算数切片之二 - 子序列

447. 回旋镖的数量

448. 找出数组中所有消失的数字

449. 二叉搜索树的序列化和去序列化

450. 删除二叉搜索树中的节点

451. 根据字符出现频率排序

452. 最少数量的箭引爆气球

453. 最少移动次数使数组元素相等

454. 四数之和之二

455. 分点心

456. 132模式

457. 环形数组循环

458. 可怜的猪

459. 重复子字符串模式

460. 最近最不常用页面置换缓存器

461. 汉明距离

462. 最少移动次数使数组元素相等之二

463. 岛屿周长

464. 我能赢吗

465. 最优账户平衡

466. 计数重复个数

467. 封装字符串中的独特子字符串

468. 验证IP地址

469. 凸多边形

470. 用 Rand7() 实现 Rand10()

471. 最短长度编码字符串

472. 连接的单词

473. 火柴棍组成正方形

474. 一和零

475. 加热器

476. 补数

477. 全部汉明距离

478. 在圆内随机生成点

- 479. 最大回文串乘积
- 480. 滑动窗口中位数
- 481. 神奇字符串
- 482. 注册码格式化
- 483. 最小的好基数
- 484. 找全排列
- 485. 最大连续1的个数
- 486. 预测赢家
- 487. 最大连续1的个数之二
- 488. 祖玛游戏
- 489. Robot Room Cleaner
- 490. 迷宫
- 491. 递增子序列
- 492. 构建矩形
- 493. 翻转对
- 494. 目标和
- 495. 提莫攻击
- 496. 下一个较大的元素之一
- 497. 非重叠矩形中的随机点
- 498. 对角线遍历
- 499. 迷宫之三
- 500. 键盘行
- 501. 找二分搜索数的众数
- 502. IPO 上市
- 503. 下一个较大的元素之二
- 504. 基数七
- 505. 迷宫之二
- 506. 相对排名
- 507. 完美数字
- 508. 出现频率最高的子树和
- 509. 寻找最左下树结点的值
- 510. 自由之路
- 511. 找树每行最大的结点值
- 512. 最长回文子序列
- 513. 超级洗衣机
- 514. 硬币找零之二
- 515. 检测大写格式
- 516. 最长非共同子序列之一
- 517. 最长非共同子序列之二
- 518. 连续的子数组之和
- 519. 优美排列
- 520. 单词缩写
- 521. 扫雷游戏
- 522. 二叉搜索树的最小绝对差
- 523. 孤独的像素之一
- 524. 数组中差为K的数对
- 525. 孤独的像素之二
- 526. 设计精简URL地址

- 527. 编码和解码精简URL地址
- 528. 从字符串创建二叉树
- 529. 复数相乘
- 530. 将二叉搜索树BST转为较大树
- 531. 最短时间差
- 532. 有序数组中的单独元素
- 533. 翻转字符串之二
- 534. 零一矩阵
- 535. 二叉树的直径
- 536. 输出比赛匹配对
- 537. 二叉树的边界
- 538. 移除盒子
- 539. 朋友圈
- 540. 分割数组成和相同的子数组
- 541. 二叉树最长连续序列之二
- 542. 学生出勤记录之一
- 543. 最优分隔
- 544. 砖头墙壁
- 545. 分割串联字符串
- 546. 下一个较大的元素之三
- 547. 翻转字符串中的单词之三
- 548. 四叉树交集
- 549. N叉树的最大深度
- 550. K子数组和为K
- 551. 数组分割之一
- 552. 矩阵中最长的连续1
- 553. 二叉树的坡度
- 554. 寻找最近的回文串
- 555. 数组嵌套
- 556. 重塑矩阵
- 557. 字符串中的全排列
- 558. 最大化休假日
- 559. Median Employee Salary
- 560. Managers with at Least 5 Direct Reports
- 561. Find Median Given Frequency of Numbers
- 562. 另一个树的子树
- 563. 松鼠模拟
- 564. Winning Candidate
- 565. 分糖果
- 566. 出界的路径
- 567. Employee Bonus
- 568. Get Highest Answer Rate Question
- 569. Find Cumulative Salary of an Employee
- 570. Count Student Number in Departments
- 571. 最短无序连续子数组
- 572. 结束进程
- 573. 两个字符串的删除操作
- 574. Find Customer Referee

- 575. Investments in 2016
- 576. Customer Placing the Largest Number of Orders
- 577. 竖立栅栏
- 578. 设计内存文件系统
- 579. N-ary Tree Preorder Traversal
- 580. N-ary Tree Postorder Transversal
- 581. 标签验证器
- 582. 分数加减法
- 583. 验证正方形
- 584. 最长和谐子序列
- 585. Big Countries
- 586. Classes More Than 5 Students
- 587. Friend Requests I: Overall Acceptance Rate
- 588. 范围相加之二
- 589. 两个表单的最小坐标和
- 590. 非负整数不包括连续的1
- 591. Human Traffic of Stadium
- 592. Friend Requests II: Who Has Most Friend?
- 593. Consecutive Available Seats
- 594. 设计压缩字符串的迭代器
- 595. 可以放置花
- 596. 根据二叉树创建字符串
- 597. Sales Person
- 598. Tree Node
- 599. 在系统中寻找重复文件
- 600. Triangle Judgement
- 601. 合法的三角形个数
- 602. Shortest Distance in a Plane
- 603. Shortest Distance in a Line
- 604. Second Degree Follower
- 605. Average Salary: Departments VS Company
- 606. 字符串中增添加粗标签
- 607. 合并二叉树
- 608. Students Report By Geography
- 609. Biggest Single Number
- 610. Not Boring Movies
- 611. 任务行程表
- 612. Design circular queue
- 613. 二叉树中增加一行
- 614. 数组中的最大距离
- 615. 最小因数分解
- 616. 三个数字的最大乘积
- 617. K个翻转对数组
- 618. 课程清单之三
- 619. 设计Excel表格求和公式
- 620. 最小的范围
- 621. 平方数之和
- 622. 找数组的错排

- 623. 设计日志存储系统
- 624. 函数的独家时间
- 625. 二叉树的层平均值
- 626. 购物优惠
- 627. 解码方法之二
- 628. 解方程
- 629. 设计循环双端队列
- 630. 设计搜索自动补全系统
- 631. 子数组的最大平均值
- 632. 子数组的最大平均值之二
- 633. 设置不匹配
- 634. 链对的最大长度
- 635. 回文子字符串
- 636. 替换单词
- 637. 刀塔二参议院
- 638. 两键的键盘
- 639. 四键的键盘
- 640. 寻找重复树
- 641. 两数之和之四 - 输入是二叉搜索树
- 642. 最大二叉树
- 643. 打印二叉树
- 644. 硬币路径
- 645. 判断路线绕圈
- 646. 寻找K个最近元素
- 647. 将数组分割成连续子序列
- 648. 移除9
- 649. 图片平滑器
- 650. 二叉树的最大宽度
- 651. 划分等价树
- 652. 奇怪的打印机
- 653. 非递减数列
- 654. 二叉树的路径和之四
- 655. 优美排列之二
- 656. 乘法表中的第K小的数字
- 657. 修剪一棵二叉搜索树
- 658. 大置换
- 659. 二叉树中第二小的结点
- 660. 灯泡开关之二
- 661. 最长递增序列的个数
- 662. 最长连续递增序列
- 663. 为高尔夫赛事砍树
- 664. 实现神奇字典
- 665. 映射配对之和
- 666. 验证括号字符串
- 667. 二十四点游戏
- 668. 验证回文字符串之二
- 669. 下一个最近时间点
- 670. 棒球游戏

- 671. K个空槽
- 672. 冗余的连接
- 673. 重复字符串匹配
- 674. 最长相同值路径
- 675. 棋盘上骑士的可能性
- 676. 三个非重叠子数组的最大和
- 677. 员工重要度
- 678. 贴片拼单词
- 679. 前K个高频词
- 680. 有交替位的二进制数
- 681. 不同岛屿的个数
- 682. 岛的最大面积
- 683. 统计二进制子字符串
- 684. 数组的度
- 685. 分割K个等和的子集
- 686. 下落的方块
- 687. 二叉搜索树中的搜索
- 688. 二叉搜索树中的插入操作
- 689. Search in a Sorted Array of Unknown Size
- 690. 数据流中的第K大元素
- 691. 二分查找
- 692. 设计哈希集合
- 693. 设计哈希映射
- 694. 设计链表
- 695. Insert into cyclic sorted list
- 696. 转换成小写字母
- 697. 黑名单中的随机数
- 698. 不同岛屿的个数之二
- 699. 两个字符串的最小ASCII删除和
- 700. 子数组乘积小于K
- 701. 买股票的最佳时间含交易费
- 702. 范围模块
- 703. 最大栈
- 704. 一位和两位字符
- 705. 最长的重复子数组
- 706. 找第K小的数对儿距离
- 707. 字典中的最长单词
- 708. 账户合并
- 709. 移除注释
- 710. 糖果消消乐
- 711. 寻找中枢点
- 712. 拆分链表成部分
- 713. 原子的个数
- 714. 最小窗口序列
- 715. 自整除数字
- 716. 我的日历之一
- 717. 计数不同的回文子序列的个数
- 718. 我的日历之二

- 719. 我的日历之三
- 720. 洪水填充
- 721. 句子相似度
- 722. 行星碰撞
- 723. 句子相似度之二
- 724. 单调递增数字
- 725. 日常温度
- 726. 删除与赚取
- 727. 捡樱桃
- 728. 二叉树中最近的叶结点
- 729. 网络延迟时间
- 730. 找比目标值大的最小字母
- 731. 前后缀搜索
- 732. 爬楼梯的最小损失
- 733. 至少是其他数字两倍的最大数
- 734. 最短完整的单词
- 735. 边角矩形的数量
- 736. 数组中的第k个最大元素

“ 数学是人类智慧的结晶，统计学习是数学领域最璀璨的明珠，算法可以让这个明珠照亮整个世界! ”

— zhongyingLi

1. 两数之和

给定一个整数数组和一个目标值，找出数组中和为目标值的两个数。

你可以假设每个输入只对应一种答案，且同样的元素不能被重复利用。

示例:

给定 `nums = [2, 7, 11, 15]`, `target = 9`

因为 `nums[0] + nums[1] = 2 + 7 = 9`

所以返回 `[0, 1]`

解法1:

```
1 class Solution {
2 public:
3     vector<int> twoSum(vector<int>& nums, int target) {
4         unordered_map<int, int> m;
5         vector<int> res;
6         for (int i = 0; i < nums.size(); ++i) {
7             m[nums[i]] = i;
8         }
9         for (int i = 0; i < nums.size(); ++i) {
10            int t = target - nums[i];
11            if (m.count(t) && m[t] != i) {
12                res.push_back(i);
13                res.push_back(m[t]);
14                break;
15            }
16        }
17        return res;
18    }
19};
```

CPP

解法2:

```
1 class Solution {
2 public:
3     vector<int> twoSum(vector<int>& nums, int target) {
4         unordered_map<int, int> m;
5         for (int i = 0; i < nums.size(); ++i) {
6             if (m.count(target - nums[i])) {
7                 return {i, m[target - nums[i]]};
8             }
9             m[nums[i]] = i;
10        }
11        return {};
12    }
13};
```

CPP

2. 两数相加

给定两个非空链表来表示两个非负整数。位数按照逆序方式存储，它们的每个节点只存储单个数字。将两数相加返回一个新的链表。

你可以假设除了数字 0 之外，这两个数字都不会以零开头。

示例：

输入: (2 -> 4 -> 3) + (5 -> 6 -> 4)

输出: 7 -> 0 -> 8

原因: 342 + 465 = 807

```

1 class Solution {
2 public:
3     ListNode *addTwoNumbers(ListNode *l1, ListNode *l2) {
4         ListNode *res = new ListNode(-1);
5         ListNode *cur = res;
6         int carry = 0;
7         while (l1 || l2) {
8             int n1 = l1 ? l1->val : 0;
9             int n2 = l2 ? l2->val : 0;
10            int sum = n1 + n2 + carry;
11            carry = sum / 10;
12            cur->next = new ListNode(sum % 10);
13            cur = cur->next;
14            if (l1) l1 = l1->next;
15            if (l2) l2 = l2->next;
16        }
17        if (carry) cur->next = new ListNode(1);
18        return res->next;
19    }
20}

```

CPP

3. 无重复字符的最长子串

给定一个字符串，找出不含有重复字符的最长子串的长度。

示例：

给定 "abcabcbb"，没有重复字符的最长子串是 "abc"，那么长度就是3。

给定 "bbbbbb"，最长的子串就是 "b"，长度是1。

给定 "pwwkew"，最长子串是 "wke"，长度是3。请注意答案必须是一个子串，"pwke" 是子序列而不是子串。

解法1：

```

1 class Solution {
2 public:
3     int lengthOfLongestSubstring(string s) {
4         int m[256] = {0}, res = 0, left = 0;
5         for (int i = 0; i < s.size(); ++i) {
6             if (m[s[i]] == 0 || m[s[i]] < left) {
7                 res = max(res, i - left + 1);
8             } else {
9                 left = m[s[i]];
10            }
11            m[s[i]] = i + 1;
12        }
13        return res;
14    }
15 };

```

解法2:

```

1 class Solution {
2 public:
3     int lengthOfLongestSubstring(string s) {
4         vector<int> m(256, -1);
5         int res = 0, left = -1;
6         for (int i = 0; i < s.size(); ++i) {
7             left = max(left, m[s[i]]);
8             m[s[i]] = i;
9             res = max(res, i - left);
10        }
11        return res;
12    }
13 };

```

解法3

```

1 class Solution {
2 public:
3     int lengthOfLongestSubstring(string s) {
4         int res = 0, left = 0, i = 0, n = s.size();
5         unordered_set<char> t;
6         while (i < n) {
7             if (!t.count(s[i])) {
8                 t.insert(s[i++]);
9                 res = max(res, (int)t.size());
10            } else {
11                t.erase(s[left++]);
12            }
13        }
14        return res;
15    }
16 };

```

解法4:

```

1 class Solution {
2 public:
3     int lengthOfLongestSubstring(string s) {
4         int res = 0, left = 0, i = 0, n = s.size();
5         unordered_map<char, int> m;
6         for (int i = 0; i < n; ++i) {
7             left = max(left, m[s[i]]);
8             m[s[i]] = i + 1;
9             res = max(res, i - left + 1);
10        }
11        return res;
12    }
13 };

```

4. 两个排序数组的中位数

给定两个大小为 m 和 n 的有序数组 nums1 和 nums2 。

请找出这两个有序数组的中位数。要求算法的时间复杂度为 $O(\log(m+n))$ 。

你可以假设 nums1 和 nums2 均不为空。

示例 1:

```

nums1 = [1, 3]
nums2 = [2]

```

中位数是 2.0

示例 2:

```

nums1 = [1, 2]
nums2 = [3, 4]

```

中位数是 $(2 + 3)/2 = 2.5$

解法1:

```

1 class Solution {
2 public:
3     double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
4         int total = nums1.size() + nums2.size();
5         if (total % 2 == 1) {
6             return findKth(nums1, 0, nums2, 0, total / 2 + 1);
7         } else {
8             return (findKth(nums1, 0, nums2, 0, total / 2) + findKth(nums1, 0, nums2, 0,
9 total / 2 + 1)) / 2;
10        }
11    }
12    double findKth(vector<int> &nums1, int i, vector<int> &nums2, int j, int k) {
13        if (nums1.size() - i > nums2.size() - j) return findKth(nums2, j, nums1, i, k);
14        if (nums1.size() == i) return nums2[j + k - 1];
15        if (k == 1) return min(nums1[i], nums2[j]);
16        int pa = min(i + k / 2, int(nums1.size())), pb = j + k - pa + i;
17        if (nums1[pa - 1] < nums2[pb - 1])
18            return findKth(nums1, pa, nums2, j, k - pa + i);
19        else if (nums1[pa - 1] > nums2[pb - 1])
20            return findKth(nums1, i, nums2, pb, k - pb + j);
21        else
22            return nums1[pa - 1];
23    }
24};

```

解法2:

```

1 class Solution {
2 public:
3     double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
4         int m = nums1.size(), n = nums2.size();
5         return (findKth(nums1, nums2, (m + n + 1) / 2) + findKth(nums1, nums2, (m + n + 2)
6 / 2)) / 2.0;
7     }
8     int findKth(vector<int> nums1, vector<int> nums2, int k) {
9         int m = nums1.size(), n = nums2.size();
10        if (m > n) return findKth(nums2, nums1, k);
11        if (m == 0) return nums2[k - 1];
12        if (k == 1) return min(nums1[0], nums2[0]);
13        int i = min(m, k / 2), j = min(n, k / 2);
14        if (nums1[i - 1] > nums2[j - 1]) {
15            return findKth(nums1, vector<int>(nums2.begin() + j, nums2.end()), k - j);
16        } else {
17            return findKth(vector<int>(nums1.begin() + i, nums1.end()), nums2, k - i);
18        }
19        return 0;
20    }
21};

```

解法3:

```

1 class Solution {
2 public:
3     double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
4         int m = nums1.size(), n = nums2.size();
5         if (m < n) return findMedianSortedArrays(nums2, nums1);
6         if (n == 0) return ((double)nums1[(m - 1) / 2] + (double)nums1[m / 2]) / 2.0;
7         int left = 0, right = n * 2;
8         while (left <= right) {
9             int mid2 = (left + right) / 2;
10            int mid1 = m + n - mid2;
11            double L1 = mid1 == 0 ? INT_MIN : nums1[(mid1 - 1) / 2];
12            double L2 = mid2 == 0 ? INT_MIN : nums2[(mid2 - 1) / 2];
13            double R1 = mid1 == m * 2 ? INT_MAX : nums1[mid1 / 2];
14            double R2 = mid2 == n * 2 ? INT_MAX : nums2[mid2 / 2];
15            if (L1 > R2) left = mid2 + 1;
16            else if (L2 > R1) right = mid2 - 1;
17            else return (max(L1, L2) + min(R1, R2)) / 2;
18        }
19        return -1;
20    }
21 };

```

5. 最长回文子串

给定一个字符串 s，找到 s 中最长的回文子串。你可以假设 s 的最大长度为1000。

示例 1:

输入: "babad"
 输出: "bab"
 注意: "aba"也是一个有效答案。

示例 2:

输入: "cbbd"
 输出: "bb"

解法1:

```

1 class Solution {
2 public:
3     string longestPalindrome(string s) {
4         int startIdx = 0, left = 0, right = 0, len = 0;
5         for (int i = 0; i < s.size() - 1; ++i) {
6             if (s[i] == s[i + 1]) {
7                 left = i;
8                 right = i + 1;
9                 searchPalindrome(s, left, right, startIdx, len);
10            }
11            left = right = i;
12            searchPalindrome(s, left, right, startIdx, len);
13        }
14        if (len == 0) len = s.size();
15        return s.substr(startIdx, len);
16    }
17    void searchPalindrome(string s, int left, int right, int &startIdx, int &len) {
18        int step = 1;
19        while ((left - step) >= 0 && (right + step) < s.size()) {
20            if (s[left - step] != s[right + step]) break;
21            ++step;
22        }
23        int wide = right - left + 2 * step - 1;
24        if (len < wide) {
25            len = wide;
26            startIdx = left - step + 1;
27        }
28    }
29}

```

解法2:

```

1 class Solution {
2 public:
3     string longestPalindrome(string s) {
4         int dp[s.size()][s.size()] = {0}, left = 0, right = 0, len = 0;
5         for (int i = 0; i < s.size(); ++i) {
6             for (int j = 0; j < i; ++j) {
7                 dp[j][i] = (s[i] == s[j] && (i - j < 2 || dp[j + 1][i - 1]));
8                 if (dp[j][i] && len < i - j + 1) {
9                     len = i - j + 1;
10                    left = j;
11                    right = i;
12                }
13            }
14            dp[i][i] = 1;
15        }
16        return s.substr(left, right - left + 1);
17    }
18}

```

解法3:

```

1 class Solution {
2 public:
3     string longestPalindrome(string s) {
4         string t ="$#";
5         for (int i = 0; i < s.size(); ++i) {
6             t += s[i];
7             t += '#';
8         }
9         int p[t.size()] = {0}, id = 0, mx = 0, resId = 0, resMx = 0;
10        for (int i = 0; i < t.size(); ++i) {
11            p[i] = mx > i ? min(p[2 * id - i], mx - i) : 1;
12            while (t[i + p[i]] == t[i - p[i]]) ++p[i];
13            if (mx < i + p[i]) {
14                mx = i + p[i];
15                id = i;
16            }
17            if (resMx < p[i]) {
18                resMx = p[i];
19                resId = i;
20            }
21        }
22        return s.substr((resId - resMx) / 2, resMx - 1);
23    }
24 };

```

6. Z字形变换

将字符串 "PAYPALISHIRING" 以Z字形排列成给定的行数:

P	A	H	N			
A	P	L	S	I	I	G
Y	I	R				

之后从左往右，逐行读取字符: "PAHNAPLSIIGYIR"

实现一个将字符串进行指定行数变换的函数:

```
string convert(string s, int numRows);
```

示例 1:

输入: s = "PAYPALISHIRING", numRows = 3
输出: "PAHNAPLSIIGYIR"

示例 2:

输入: s = "PAYPALISHIRING", numRows = 4

输出: "PINALSIGYAHRPI"

解释:

```
P     I     N
A   L S   I G
Y A   H R
P     I
```

```
1 class Solution {
2 public:
3     string convert(string s, int numRows) {
4         if (numRows <= 1) return s;
5         string res = "";
6         int size = 2 * numRows - 2;
7         for (int i = 0; i < numRows; ++i) {
8             for (int j = i; j < s.size(); j += size) {
9                 res += s[j];
10                int tmp = j + size - 2 * i;
11                if (i != 0 && i != numRows - 1 && tmp < s.size()) res += s[tmp];
12            }
13        }
14        return res;
15    }
16};
```

CPP

7. 反转整数

给定一个 32 位有符号整数，将整数中的数字进行反转。

示例 1:

输入: 123

输出: 321

示例 2: +

输入: -123

输出: -321

示例 3:

输入: 120

输出: 21

注意:

假设我们的环境只能存储 32 位有符号整数，其数值范围是 [-231, 231 - 1]。根据这个假设，如果反转后的整数溢出，则返回 0。

解法1:

```

1 class Solution {
2 public:
3     int reverse(int x) {
4         long long res = 0;
5         bool isPositive = true;
6         if (x < 0) {
7             isPositive = false;
8             x *= -1;
9         }
10        while (x > 0) {
11            res = res * 10 + x % 10;
12            x /= 10;
13        }
14        if (res > INT_MAX) return 0;
15        if (isPositive) return res;
16        else return -res;
17    }
18 };

```

解法2:

```

1 class Solution {
2 public:
3     int reverse(int x) {
4         int res = 0;
5         while (x != 0) {
6             if (abs(res) > INT_MAX / 10) return 0;
7             res = res * 10 + x % 10;
8             x /= 10;
9         }
10        return res;
11    }
12 };

```

解法3:

```

1 class Solution {
2 public:
3     int reverse(int x) {
4         long long res = 0;
5         while (x != 0) {
6             res = 10 * res + x % 10;
7             x /= 10;
8         }
9         return (res > INT_MAX || res < INT_MIN) ? 0 : res;
10    }
11 };

```

解法4:

```

1 class Solution {
2 public:
3     int reverse(int x) {
4         int res = 0;
5         while (x != 0) {
6             int t = res * 10 + x % 10;
7             if (t / 10 != res) return 0;
8             res = t;
9             x /= 10;
10        }
11        return res;
12    }
13 };

```

8. 字符串转整数 (atoi)

实现 atoi，将字符串转为整数。

在找到第一个非空字符之前，需要移除掉字符串中的空格字符。如果第一个非空字符是正号或负号，选取该符号，并将其与后面尽可能多的连续的数字组合起来，这部分字符即为整数的值。如果第一个非空字符是数字，则直接将其与之后连续的数字字符组合起来，形成整数。

字符串可以在形成整数的字符后面包括多余的字符，这些字符可以被忽略，它们对于函数没有影响。

当字符串中的第一个非空字符序列不是个有效的整数；或字符串为空；或字符串仅包含空白字符时，则不进行转换。

若函数不能执行有效的转换，返回 0。

说明：

假设我们的环境只能存储 32 位有符号整数，其数值范围是 [-231, 231 - 1]。如果数值超过可表示的范围，则返回 INT_MAX (231 - 1) 或 INT_MIN (-231)。

示例 1：

输入： "42"
输出： 42

示例 2：

输入： " -42"
输出： -42
解释： 第一个非空白字符为 '-'，它是一个负号。
我们尽可能将负号与后面所有连续出现的数字组合起来，最后得到 -42 。

示例 3：

输入： "4193 with words"
输出： 4193
解释： 转换截止于数字 '3'，因为它的下一个字符不为数字。

示例 4：

输入: "words and 987"
 输出: 0
 解释: 第一个非空字符是 'w' , 但它不是数字或正、负号。
 因此无法执行有效的转换。

示例 5:

输入: "-91283472332"
 输出: -2147483648
 解释: 数字 "-91283472332" 超过 32 位有符号整数范围。
 因此返回 INT_MIN (-231) 。

```
1 class Solution {
2 public:
3     int myAtoi(string str) {
4         if (str.empty()) return 0;
5         int sign = 1, base = 0, i = 0, n = str.size();
6         while (i < n && str[i] == ' ') ++i;
7         if (str[i] == '+' || str[i] == '-') {
8             sign = (str[i++] == '+') ? 1 : -1;
9         }
10        while (i < n && str[i] >= '0' && str[i] <= '9') {
11            if (base > INT_MAX / 10 || (base == INT_MAX / 10 && str[i] - '0' > 7)) {
12                return (sign == 1) ? INT_MAX : INT_MIN;
13            }
14            base = 10 * base + (str[i++] - '0');
15        }
16        return base * sign;
17    }
18};
```

CPP

9. 回文数

判断一个整数是否是回文数。回文数是指正序（从左向右）和倒序（从右向左）读都是一样的整数。

示例 1:

输入: 121
 输出: true

示例 2:

输入: -121
 输出: false
 解释: 从左向右读, 为 -121 。 从右向左读, 为 121- 。因此它不是一个回文数。

示例 3:

输入: 10
 输出: false
 解释: 从右向左读, 为 01 。因此它不是一个回文数。

进阶：

你能不将整数转为字符串来解决这个问题吗？

解法1：

```
1 class Solution {
2 public:
3     bool isPalindrome(int x) {
4         if (x < 0) return false;
5         int div = 1;
6         while (x / div >= 10) div *= 10;
7         while (x > 0) {
8             int left = x / div;
9             int right = x % 10;
10            if (left != right) return false;
11            x = (x % div) / 10;
12            div /= 100;
13        }
14        return true;
15    }
16};
```

CPP

解法2：

```
1 class Solution {
2 public:
3     bool isPalindrome(int x) {
4         if (x < 0 || (x % 10 == 0 && x != 0)) return false;
5         int revertNum = 0;
6         while (x > revertNum) {
7             revertNum = revertNum * 10 + x % 10;
8             x /= 10;
9         }
10        return x == revertNum || x == revertNum / 10;
11    }
12};
```

CPP

解法3：

```

1 class Solution {
2 public:
3     bool isPalindrome(int x) {
4         if (x < 0 || (x % 10 == 0 && x != 0)) return false;
5         return reverse(x) == x;
6     }
7     int reverse(int x) {
8         int res = 0;
9         while (x != 0) {
10             if (res > INT_MAX / 10) return -1;
11             res = res * 10 + x % 10;
12             x /= 10;
13         }
14         return res;
15     }
16 };

```

10. 正则表达式匹配

给定一个字符串 (s) 和一个字符模式 (p)。实现支持 '.' 和 '*' 的正则表达式匹配。

'.' 匹配任意单个字符。

'*' 匹配零个或多个前面的元素。

匹配应该覆盖整个字符串 (s)，而不是部分字符串。

说明:

- s 可能为空，且只包含从 a-z 的小写字母。
- p 可能为空，且只包含从 a-z 的小写字母，以及字符 . 和 *。

示例 1:

输入:

s = "aa"

p = "a"

输出: false

解释: "a" 无法匹配 "aa" 整个字符串。

示例 2:

输入:

s = "aa"

p = "a*"

输出: true

解释: '*' 代表可匹配零个或多个前面的元素，即可以匹配 'a'。因此，重复 'a' 一次，字符串可变为 "aa"。

示例 3:

输入：
 s = "ab"
 p = ".*"
 输出: true
 解释: ".*" 表示可匹配零个或多个('*')任意字符('.')。

示例 4:

输入：
 s = "aab"
 p = "c*a*b"
 输出: true
 解释: 'c' 可以不被重复, 'a' 可以被重复一次。因此可以匹配字符串 "aab"。

示例 5:

输入：
 s = "mississippi"
 p = "mis*is*p*."
 输出: false

解法1:

```
1 class Solution {
2 public:
3     bool isMatch(string s, string p) {
4         if (p.empty()) return s.empty();
5         if (p.size() == 1) {
6             return (s.size() == 1 && (s[0] == p[0] || p[0] == '.'));
7         }
8         if (p[1] != '*') {
9             if (s.empty()) return false;
10            return (s[0] == p[0] || p[0] == '.') && isMatch(s.substr(1), p.substr(1));
11        }
12        while (!s.empty() && (s[0] == p[0] || p[0] == '.')) {
13            if (isMatch(s, p.substr(2))) return true;
14            s = s.substr(1);
15        }
16        return isMatch(s, p.substr(2));
17    }
18};
```

CPP

解法2:

```

1 class Solution {
2 public:
3     bool isMatch(string s, string p) {
4         if (p.empty()) return s.empty();
5         if (p.size() > 1 && p[1] == '*') {
6             return isMatch(s, p.substr(2)) || (!s.empty() && (s[0] == p[0] || p[0] == '.')) 
7             && isMatch(s.substr(1), p));
8         } else {
9             return !s.empty() && (s[0] == p[0] || p[0] == '.') && isMatch(s.substr(1),
10 p.substr(1));
11         }
12     }
13 };

```

解法3：

```

1 class Solution {
2 public:
3     bool isMatch(string s, string p) {
4         int m = s.size(), n = p.size();
5         vector<vector<bool>> dp(m + 1, vector<bool>(n + 1, false));
6         dp[0][0] = true;
7         for (int i = 0; i <= m; ++i) {
8             for (int j = 1; j <= n; ++j) {
9                 if (j > 1 && p[j - 1] == '*') {
10                     dp[i][j] = dp[i][j - 2] || (i > 0 && (s[i - 1] == p[j - 2] || p[j - 2]
11 == '.')) && dp[i - 1][j];
12                 } else {
13                     dp[i][j] = i > 0 && dp[i - 1][j - 1] && (s[i - 1] == p[j - 1] || p[j - 1] == '.');
14                 }
15             }
16         }
17         return dp[m][n];
18     }
19 };

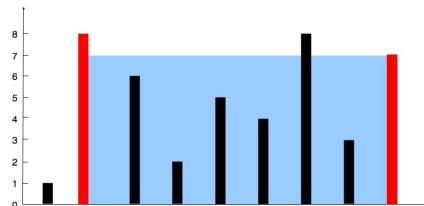
```

11. 盛最多水的容器

给定 n 个非负整数 a_1, a_2, \dots, a_n ，每个数代表坐标中的一个点 (i, a_i) 。在坐标内画 n 条垂直线，垂直线 i 的两个端点分别为 (i, a_i) 和 $(i, 0)$ 。

找出其中的两条线，使得它们与 x 轴共同构成的容器可以容纳最多的水。

说明：你不能倾斜容器，且 n 的值至少为 2。



图中垂直线代表输入数组 $[1,8,6,2,5,4,8,3,7]$ 。在此情况下，容器能够容纳水（表示为蓝色部分）的最大值为 49。

示例：

输入： [1,8,6,2,5,4,8,3,7]
 输出： 49

这道求装最多水的容器的题和那道 Trapping Rain Water 收集雨水 很类似，但又有些不同，那道题让求整个能收集雨水的量，这道只是让求最大的一个的装水量，而且还有点不同的是，那道题容器边缘不能算在里面，而这道题却可以算，相比较来说还是这道题容易一些，我们需要定义i和j两个指针分别指向数组的左右两端，然后两个指针向中间搜索，每移动一次算一个值和结果比较取较大的，容器装水量的算法是找出左右两个边缘中较小的那个乘以两边缘的距离，代码如下：

解法1：

```
1 class Solution {
2 public:
3     int maxArea(vector<int>& height) {
4         int res = 0, i = 0, j = height.size() - 1;
5         while (i < j) {
6             res = max(res, min(height[i], height[j]) * (j - i));
7             height[i] < height[j] ? ++i : --j;
8         }
9         return res;
10    }
11};
```

CPP

下面这种方法是对上面的方法进行了小幅度的优化，对于相同的高度们直接移动i和j就行了，不再进行计算容量了，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     int maxArea(vector<int>& height) {
4         int res = 0, i = 0, j = height.size() - 1;
5         while (i < j) {
6             int h = min(height[i], height[j]);
7             res = max(res, h * (j - i));
8             while (i < j && h == height[i]) ++i;
9             while (i < j && h == height[j]) --j;
10        }
11        return res;
12    }
13};
```

CPP

12. 整数转罗马数字

罗马数字包含以下七种字符： I, V, X, L, C, D 和 M。

字符	数值
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

例如，罗马数字 2 写做 II，即为两个并列的 1。12 写做 XII，即为 X + II。27 写做 XXVII，即为 XX + V + II。

通常情况下，罗马数字中小的数字在大的数字的右边。但也存在特例，例如 4 不写做 IIII，而是 IV。数字 1 在数字 5 的左边，所表示的数等于大数 5 减小数 1 得到的数值 4。同样地，数字 9 表示为 IX。这个特殊的规则只适用于以下六种情况：

- I 可以放在 V (5) 和 X (10) 的左边，来表示 4 和 9.
- X 可以放在 L (50) 和 C (100) 的左边，来表示 40 和 90.
- C 可以放在 D (500) 和 M (1000) 的左边，来表示 400 和 900.

给定一个整数，将其转为罗马数字。输入确保在 1 到 3999 的范围内。

示例 1:

输入： 3
输出： "III"

示例 2:

输入： 4
输出： "IV"

示例 3:

输入： 9
输出： "IX"

示例 4:

输入： 58
输出： "LVIII"
解释： L = 50, V = 5, III = 3.

示例 5:

输入： 1994
输出： "MCMXCIV"
解释： M = 1000, CM = 900, XC = 90, IV = 4.

之前那篇文章写的是罗马数字转化成整数(<http://www.cnblogs.com/grandyang/p/4120857.html>)，这次变成了整数转化成罗马数字，基本算法还是一样。由于题目中限定了输入数字的范围(1 - 3999)，使得题目变得简单了不少。

例如整数 1437 的罗马数字为 MCDXXXVII，我们不难发现，千位，百位，十位和个位上的数分别用罗马数字表示了。1000 - M, 400 - CD, 30 - XXX, 7 - VII。所以我们要做的就是用取商法分别提取各个位上的数字，然后分别表示出来：

100 - C

200 - CC

300 - CCC

400 - CD

500 - D

600 - DC

700 - DCC

800 - DCCC

900 - CM

我们可以分为四类，100到300一类，400一类，500到800一类，900最后一类。每一位上的情况都是类似的，代码如下：

解法1：

```

1 class Solution {
2 public:
3     string intToRoman(int num) {
4         string res = "";
5         char roman[] = {'M', 'D', 'C', 'L', 'X', 'V', 'I'};
6         int value[] = {1000, 500, 100, 50, 10, 5, 1};
7
8         for (int n = 0; n < 7; n += 2) {
9             int x = num / value[n];
10            if (x < 4) {
11                for (int i = 1; i <= x; ++i) res += roman[n];
12            } else if (x == 4) res = res + roman[n] + roman[n - 1];
13            else if (x > 4 && x < 9) {
14                res += roman[n - 1];
15                for (int i = 6; i <= x; ++i) res += roman[n];
16            }
17            else if (x == 9) res = res + roman[n] + roman[n - 2];
18            num %= value[n];
19        }
20        return res;
21    }
22 };

```

CPP

本题由于限制了输入数字范围这一特殊性，故而还有一种利用贪婪算法的解法，建立一个数表，每次通过查表找出当前最大的数，减去再继续查表。参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     string intToRoman(int num) {
4         string res = "";
5         vector<int> val{1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1};
6         vector<string> str{"M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV",
7         "I"};
8         for (int i = 0; i < val.size(); ++i) {
9             while (num >= val[i]) {
10                 num -= val[i];
11                 res += str[i];
12             }
13         }
14     return res;
15 }
16 };

```

下面这种方法个人感觉属于比较投机取巧的方法，把所有的情况都列了出来，然后直接按位查表，O(1)的时间复杂度啊，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     string intToRoman(int num) {
4         string res = "";
5         vector<string> v1{"", "M", "MM", "MMM"};
6         vector<string> v2{"", "C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCC", "CM"};
7         vector<string> v3{"", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX", "XC"};
8         vector<string> v4{"", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX"};
9         return v1[num / 1000] + v2[(num % 1000) / 100] + v3[(num % 100) / 10] + v4[num %
10 10];
11 }
12 };

```

13. 罗马数字转整数

罗马数字包含以下七种字符：I、V、X、L、C、D 和 M。

字符	数值
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

例如，罗马数字 2 写做 II，即为两个并列的 1。12 写做 XII，即为 X + II。27 写做 XXVII，即为 XX + V + II。

通常情况下，罗马数字中小的数字在大的数字的右边。但也存在特例，例如 4 不写做 IIII，而是 IV。数字 1 在数字 5 的左边，所表示的数等于大数 5 减小数 1 得到的数值 4。同样地，数字 9 表示为 IX。这个特殊的规则只适用于以下六种情况：

- I 可以放在 V(5) 和 X(10) 的左边，来表示 4 和 9。
- X 可以放在 L(50) 和 C(100) 的左边，来表示 40 和 90。

- C 可以放在 D (500) 和 M (1000) 的左边，来表示 400 和 900。

给定一个罗马数字，将其转换成整数。输入确保在 1 到 3999 的范围内。

示例 1:

输入: "III"

输出: 3

示例 2:

输入: "IV"

输出: 4

示例 3:

输入: "IX"

输出: 9

示例 4:

输入: "LVIII"

输出: 58

解释: L = 50, V= 5, III = 3.

示例 5:

输入: "MCMXCIV"

输出: 1994

解释: M = 1000, CM = 900, XC = 90, IV = 4.

罗马数转化成数字问题，我们需要对于罗马数字很熟悉才能完成转换。以下截自百度百科：

罗马数字是最早的数字表示方式，比阿拉伯数字早2000多年，起源于罗马。

如今我们最常见的罗马数字就是钟表的表盘符号： I , II , III , IV (IIII) , V , VI , VII , VIII , IX , X , XI , XII.....

对应阿拉伯数字（就是现在国际通用的数字），就是1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12。（注：阿拉伯数字其实是古代印度人发明的，后来由阿拉伯人传入欧洲，被欧洲人误称为阿拉伯数字。）

- 1、相同的数字连写，所表示的数等于这些数字相加得到的数，如： III = 3；
- 2、小的数字在大的数字的右边，所表示的数等于这些数字相加得到的数，如： VIII = 8; XII = 12;
- 3、小的数字，（限于 I 、 X 和C）在大的数字的左边，所表示的数等于大数减小数得到的数，如： IV= 4; IX= 9;
- 4、正常使用时，连写的数字重复不得超过三次。（表盘上的四点钟“IIII”例外）
- 5、在一个数的上面画一条横线，表示这个数扩大1000倍。

有几条须注意掌握：

- 1、基本数字 I 、 X 、 C 中的任何一个，自身连用构成数目，或者放在大数的右边连用构成数目，都不能超过三个；放在大数的左边只能用一个。
- 2、不能把基本数字V、L、D 中的任何一个作为小数放在大数的左边采用相减的方法构成数目；放在大数的右边采用相加的方式构成数目，只能使用一个。

- 3、V 和 X 左边的小数字只能用 I。
- 4、L 和 C 左边的小数字只能用 X。
- 5、D 和 M 左边的小数字只能用 C。

而这道题好就好在没有让我们来验证输入字符串是不是罗马数字，这样省掉不少功夫。我们需要用到map数据结构，来将罗马数字的字母转化为对应的整数值，因为输入的一定是罗马数字，那么我们只要考虑两种情况即可：

第一，如果当前数字是最后一个数字，或者之后的数字比它小的话，则加上当前数字

第二，其他情况则减去这个数字

解法1：

```
1 class Solution {
2 public:
3     int romanToInt(string s) {
4         int res = 0;
5         unordered_map<char, int> m{{'I', 1}, {'V', 5}, {'X', 10}, {'L', 50}, {'C', 100},
6         {'D', 500}, {'M', 1000}};
7         for (int i = 0; i < s.size(); ++i) {
8             int val = m[s[i]];
9             if (i == s.size() - 1 || m[s[i+1]] <= m[s[i]]) res += val;
10            else res -= val;
11        }
12        return res;
13    }
};
```

CPP

我们也可以每次跟前面的数字比较，如果小于等于前面的数字，我们先加上当前的数字，如果大于的前面的数字，我们加上当前的数字减去二倍前面的数字，这样可以把在上一个循环多加数减掉，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     int romanToInt(string s) {
4         int res = 0;
5         unordered_map<char, int> m{{'I', 1}, {'V', 5}, {'X', 10}, {'L', 50}, {'C', 100},
6         {'D', 500}, {'M', 1000}};
7         for (int i = 0; i < s.size(); ++i) {
8             if (i == 0 || m[s[i]] <= m[s[i - 1]]) res += m[s[i]];
9             else res += m[s[i]] - 2 * m[s[i - 1]];
10        }
11        return res;
12    }
};
```

CPP

14. 最长公共前缀

编写一个函数来查找字符串数组中的最长公共前缀。

如果不存在公共前缀，返回空字符串 ""。

示例1：

输入: ["flower", "flow", "flight"]
 输出: "fl"

示例2:

输入: ["dog", "racecar", "car"]
 输出: ""
 解释: 输入不存在公共前缀。

说明:

所有输入只包含小写字母 a-z

这道题让我们求一系列字符串的共同前缀，没有什么特别的技巧，无脑查找即可，我们定义两个变量i和j，其中i是遍历搜索字符串中的字符，j是遍历字符串集中的每个字符串。这里将单词上下排好，则相当于一个各行长度有可能不相等的二维数组，我们遍历顺序和一般的横向逐行遍历不同，而是采用纵向逐列遍历，在遍历的过程中，如果某一行没有了，说明其为最短的单词，因为共同前缀的长度不能长于最短单词，所以此时返回已经找出的共同前缀。我们每次取出第一个字符串的某一个位置的单词，然后遍历其他所有字符串的对应位置看是否相等，如果有不满足的直接返回res，如果都相同，则将当前字符存入结果，继续检查下一个位置的字符，参见代码如下：

解法1:

CPP

```

1 class Solution {
2 public:
3     string longestCommonPrefix(vector<string>& strs) {
4         if (strs.empty()) return "";
5         string res = "";
6         for (int j = 0; j < strs[0].size(); ++j) {
7             char c = strs[0][j];
8             for (int i = 1; i < strs.size(); ++i) {
9                 if (j >= strs[i].size() || strs[i][j] != c) {
10                     return res;
11                 }
12             }
13             res.push_back(c);
14         }
15         return res;
16     }
17 };

```

我们可以对上面的方法进行适当精简，如果我们发现当前某个字符和下一行对应位置的字符不相等，说明不会再有更长的共同前缀了，我们直接通过用substr的方法直接取出共同前缀的子字符串。如果遍历结束前没有返回结果的话，说明第一个单词就是公共前缀，返回为结果即可。代码如下：

解法2:

```

1 class Solution {
2 public:
3     string longestCommonPrefix(vector<string>& strs) {
4         if (strs.empty()) return "";
5         for (int j = 0; j < strs[0].size(); ++j) {
6             for (int i = 0; i < strs.size() - 1; ++i) {
7                 if (j >= strs[i].size() || j >= strs[i + 1].size() || strs[i][j] != strs[i
8 + 1][j]) {
9                     return strs[i].substr(0, j);
10                }
11            }
12        }
13        return strs[0];
14    }
15 };

```

我们再来看一种解法，这种方法给输入字符串数组排了个序，想想这样做有什么好处？既然是按字母顺序排序的话，那么有共同字母多的两个字符串会被排到一起，而跟大家相同的字母越少的字符串会被挤到首尾两端，那么如果有共同前缀的话，一定出现在首尾两端的字符串中，所以我们只需要找首尾字母串的共同前缀即可。比如例子1排序后为 ["flight", "flow", "flower"]，例子2排序后为 ["cat", "dog", "racecar"]，虽然例子2没有共同前缀，但也可以认为共同前缀是空串，且出现在首尾两端的字符串中。由于是按字母顺序排的，而不是按长度，所以首尾字母的长度关系不知道，为了防止溢出错误，我们只遍历这种较短的那个的长度，找出共同前缀返回即可，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     string longestCommonPrefix(vector<string>& strs) {
4         if (strs.empty()) return "";
5         sort(strs.begin(), strs.end());
6         int i = 0, len = min(strs[0].size(), strs.back().size());
7         while (i < len && strs[0][i] == strs.back()[i]) ++i;
8         return strs[0].substr(0, i);
9     }
10 };

```

15. 三数之和

给定一个包含 n 个整数的数组 nums ，判断 nums 中是否存在三个元素 a, b, c ，使得 $a + b + c = 0$ ？找出所有满足条件且不重复的三元组。

注意：答案中不可以包含重复的三元组。

例如，给定数组 $\text{nums} = [-1, 0, 1, 2, -1, -4]$ ，

满足要求的三元组集合为：

```
[
  [-1, 0, 1],
  [-1, -1, 2]
]
```

这道题让我们求三数之和，比之前那道Two Sum要复杂一些，博主考虑过先fix一个数，然后另外两个数使用Two Sum那种HashMap的解法，但是会有重复结果出现，就算使用set来去除重复也不行，会TLE，看来此题并不是考我们Two Sum的解法。那么我们来分析一下这道题的特点，要我们找出三个数且和为0，那么除了三个数全是0的情况之外，肯定会有负数和正数，我们

还是要先fix一个数，然后去找另外两个数，我们只要找到两个数且和为第一个fix数的相反数就行了，既然另外两个数不能使用Two Sum的那种解法来找，如果能更有效的定位呢？我们肯定不希望遍历所有两个数的组合吧，所以如果数组是有序的，那么我们就可以用双指针以线性时间复杂度来遍历所有满足题意的两个数组合。

我们对原数组进行排序，然后开始遍历排序后的数组，这里注意不是遍历到最后一个停止，而是到倒数第三个就可以了。这里我们可以先做个剪枝优化，就是当遍历到正数的时候就break，为啥呢，因为我们的数组现在是有序的了，如果第一个要fix的数就是正数了，那么后面的数字就都是正数，就永远不会出现和为0的情况了。然后我们还要加上重复就跳过的处理，处理方法是从第二个数开始，如果和前面的数字相等，就跳过，因为我们不想把相同的数字fix两次。对于遍历到的数，用0减去这个fix的数得到一个target，然后只需要再之后找到两个数之和等于target即可。我们用两个指针分别指向fix数字之后开始的数组首尾两个数，如果两个数和正好为target，则将这两个数和fix的数一起存入结果中。然后就是跳过重复数字的步骤了，两个指针都需要检测重复数字。如果两数之和小于target，则我们将左边那个指针i右移一位，使得指向的数字增大一些。同理，如果两数之和大于target，则我们将右边那个指针j左移一位，使得指向的数字减小一些。代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<vector<int>> threeSum(vector<int>& nums) {
4         vector<vector<int>> res;
5         sort(nums.begin(), nums.end());
6         if (nums.empty() || nums.back() < 0 || nums.front() > 0) return {};
7         for (int k = 0; k < nums.size(); ++k) {
8             if (nums[k] > 0) break;
9             if (k > 0 && nums[k] == nums[k - 1]) continue;
10            int target = 0 - nums[k];
11            int i = k + 1, j = nums.size() - 1;
12            while (i < j) {
13                if (nums[i] + nums[j] == target) {
14                    res.push_back({nums[k], nums[i], nums[j]});
15                    while (i < j && nums[i] == nums[i + 1]) ++i;
16                    while (i < j && nums[j] == nums[j - 1]) --j;
17                    ++i; --j;
18                } else if (nums[i] + nums[j] < target) ++i;
19                else --j;
20            }
21        }
22        return res;
23    }
24 };

```

CPP

或者我们也可以利用set的不能包含重复项的特点来防止重复项的产生，那么我们就不需要检测数字是否被fix过两次，不过个人觉得还是前面那种解法更好一些，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<vector<int>> threeSum(vector<int>& nums) {
4         set<vector<int>> res;
5         sort(nums.begin(), nums.end());
6         if (nums.empty() || nums.back() < 0 || nums.front() > 0) return {};
7         for (int k = 0; k < nums.size(); ++k) {
8             if (nums[k] > 0) break;
9             int target = 0 - nums[k];
10            int i = k + 1, j = nums.size() - 1;
11            while (i < j) {
12                if (nums[i] + nums[j] == target) {
13                    res.insert({nums[k], nums[i], nums[j]});
14                    while (i < j && nums[i] == nums[i + 1]) ++i;
15                    while (i < j && nums[j] == nums[j - 1]) --j;
16                    ++i; --j;
17                } else if (nums[i] + nums[j] < target) ++i;
18                else --j;
19            }
20        }
21        return vector<vector<int>>(res.begin(), res.end());
22    }
23 };

```

16. 最接近的三数之和

给定一个包括 n 个整数的数组 nums 和一个目标值 target。找出 nums 中的三个整数，使得它们的和与 target 最接近。返回这三个数的和。假定每组输入只存在唯一答案。

例如，给定数组 `nums = [-1, 2, 1, -4]`，和 `target = 1`.

与 `target` 最接近的三个数的和为 `2`. ($-1 + 2 + 1 = 2$).

这道题让我们求最接近给定值的三数之和，是在之前那道 3Sum 三数之和的基础上又增加了些许难度，那么这道题让我们返回这个最接近于给定值的值，即我们要保证当前三数和跟给定值之间的差的绝对值最小，所以我们需要定义一个变量`diff`用来记录差的绝对值，然后我们还是要先将数组排个序，然后开始遍历数组，思路跟那道三数之和很相似，都是先确定一个数，然后用两个指针`left`和`right`来滑动寻找另外两个数，每确定两个数，我们求出此三数之和，然后算和给定值的差的绝对值存在`newDiff`中，然后和`diff`比较并更新`diff`和结果`closest`即可，代码如下：

```

1 class Solution {
2 public:
3     int threeSumClosest(vector<int>& nums, int target) {
4         int closest = nums[0] + nums[1] + nums[2];
5         int diff = abs(closest - target);
6         sort(nums.begin(), nums.end());
7         for (int i = 0; i < nums.size() - 2; ++i) {
8             int left = i + 1, right = nums.size() - 1;
9             while (left < right) {
10                 int sum = nums[i] + nums[left] + nums[right];
11                 int newDiff = abs(sum - target);
12                 if (diff > newDiff) {
13                     diff = newDiff;
14                     closest = sum;
15                 }
16                 if (sum < target) ++left;
17                 else --right;
18             }
19         }
20         return closest;
21     }
22 };

```

17. 电话号码的字母组合

给定一个仅包含数字 2-9 的字符串，返回所有它能表示的字母组合。

给出数字到字母的映射如下（与电话按键相同）。注意 1 不对应任何字母。

示例:

```

输入: "23"
输出: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"]

```

这道题让我们求电话号码的字母组合，即数字2到9中每个数字可以代表若干个字母，然后给一串数字，求出所有可能的组合，相类似的题目有Path Sum II, Subsets II, Permutations, Permutations II, Combinations, Combination Sum 和 Combination Sum II 等等。我们用递归Recursion来解，我们需要建立一个字典，用来保存每个数字所代表的字符串，然后我们还需要一个变量level，记录当前生成的字符串的字符个数，实现套路和上述那些题十分类似。在递归函数中我们首先判断level，如果跟digits中数字的个数相等了，我们将当前的组合加入结果res中，然后返回。否则我们通过digits中的数字到dict中取出字符串，然后遍历这个取出的字符串，将每个字符都加到当前的组合后面，并调用递归函数即可，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     vector<string> letterCombinations(string digits) {
4         if (digits.empty()) return {};
5         vector<string> res;
6         string dict[] = {"", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"};
7         letterCombinationsDFS(digits, dict, 0, "", res);
8         return res;
9     }
10    void letterCombinationsDFS(string digits, string dict[], int level, string out,
11    vector<string> &res) {
12        if (level == digits.size()) {res.push_back(out); return;}
13        string str = dict[digits[level] - '0'];
14        for (int i = 0; i < str.size(); ++i) {
15            letterCombinationsDFS(digits, dict, level + 1, out + string(1, str[i]), res);
16        }
17    }
18 };

```

这道题我们也可以用迭代Iterative来解，在遍历digits中所有的数字时，我们先建立一个临时的字符串数组t，然后跟上面解法的操作一样，通过数字到dict中取出字符串str，然后遍历取出字符串中的所有字符，再遍历当前结果res中的每一个字符串，将字符加到后面，并加入到临时字符串数组t中。取出的字符串str遍历完成后，将临时字符串数组赋值给结果res，具体实现参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<string> letterCombinations(string digits) {
4         if (digits.empty()) return {};
5         vector<string> res{""};
6         string dict[] = {"", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"};
7         for (int i = 0; i < digits.size(); ++i) {
8             vector<string> t;
9             string str = dict[digits[i] - '0'];
10            for (int j = 0; j < str.size(); ++j) {
11                for (string s : res) t.push_back(s + str[j]);
12            }
13            res = t;
14        }
15        return res;
16    }
17 };

```

18. 四数之和

给定一个包含 n 个整数的数组 nums 和一个目标值 target，判断 nums 中是否存在四个元素 a, b, c 和 d，使得 a + b + c + d 的值与 target 相等？找出所有满足条件且不重复的四元组。

注意：

答案中不可以包含重复的四元组。

给定数组 `nums = [1, 0, -1, 0, -2, 2]`, 和 `target = 0`。

满足要求的四元组集合为：

```
[  
    [-1, 0, 0, 1],  
    [-2, -1, 1, 2],  
    [-2, 0, 0, 2]  
]
```

LeetCode中关于数字之和还有其他几道，分别是Two Sum 两数之和，3Sum 三数之和，3Sum Closest 最近三数之和，虽然难度在递增，但是整体的套路都是一样的，在这里为了避免重复项，我们使用了STL中的set，其特点是不能有重复，如果新加入的数在set中原本就存在的话，插入操作就会失败，这样能很好的避免的重复项的存在。此题的O(n^3)解法的思路跟3Sum 三数之和基本没啥区别，就是多加了一层for循环，其他的都一样，代码如下：

```
1 class Solution {  
2     public:  
3         vector<vector<int>> fourSum(vector<int> &nums, int target) {  
4             set<vector<int>> res;  
5             sort(nums.begin(), nums.end());  
6             for (int i = 0; i < int(nums.size() - 3); ++i) {  
7                 for (int j = i + 1; j < int(nums.size() - 2); ++j) {  
8                     if (j > i + 1 && nums[j] == nums[j - 1]) continue;  
9                     int left = j + 1, right = nums.size() - 1;  
10                    while (left < right) {  
11                        int sum = nums[i] + nums[j] + nums[left] + nums[right];  
12                        if (sum == target) {  
13                            vector<int> out{nums[i], nums[j], nums[left], nums[right]};  
14                            res.insert(out);  
15                            ++left; --right;  
16                        } else if (sum < target) ++left;  
17                        else --right;  
18                    }  
19                }  
20            }  
21            return vector<vector<int>>(res.begin(), res.end());  
22        }  
23    };
```

CPP

19. 删除链表的倒数第N个节点

给定一个链表，删除链表的倒数第 n 个节点，并且返回链表的头结点。

示例：

给定一个链表：1->2->3->4->5，和 `n = 2`.

当删除了倒数第二个节点后，链表变为 1->2->3->5.

说明：

给定的 n 保证是有效的。

进阶：

你能尝试使用一趟扫描实现吗？

这道题让我们移除链表倒数第N个节点，限定n一定是有效的，即n不会大于链表中的元素总数。还有题目要求我们一次遍历解决问题，那么就得想些比较巧妙的方法了。比如我们首先要考虑的时，如何找到倒数第N个节点，由于只允许一次遍历，所以我们不能用一次完整的遍历来统计链表中元素的个数，而是遍历到对应位置就应该移除了。那么我们需要用两个指针来帮助我们解题，pre和cur指针。首先cur指针先向前走N步，如果此时cur指向空，说明N为链表的长度，则需要移除的为首元素，那么此时我们返回head→next即可，如果cur存在，我们再继续往下走，此时pre指针也跟着走，直到cur为最后一个元素时停止，此时pre指向要移除元素的前一个元素，我们再修改指针跳过需要移除的元素即可。代码如下：

```

1 class Solution {
2 public:
3     ListNode* removeNthFromEnd(ListNode* head, int n) {
4         if (!head->next) return NULL;
5         ListNode *pre = head, *cur = head;
6         for (int i = 0; i < n; ++i) cur = cur->next;
7         if (!cur) return head->next;
8         while (cur->next) {
9             cur = cur->next;
10            pre = pre->next;
11        }
12        pre->next = pre->next->next;
13        return head;
14    }
15 };

```

CPP

20. 有效的括号

给定一个只包括'(', ')', '{', '}', '[', ']' 的字符串，判断字符串是否有效。

有效字符串需满足：

- 左括号必须用相同类型的右括号闭合。
- 左括号必须以正确的顺序闭合。

注意空字符串可被认为是有效字符串。

示例1:

输入: "()"
输出: true

示例 2:

输入: "()"[]{}"
输出: true

示例 3:

输入: "[]"
输出: false

示例 4:

输入: "()"
输出: false

示例 5:

输入: "{}[]"
输出: true

这道题让我们验证输入的字符串是否为括号字符串，包括大括号，中括号和小括号。这里我们需要用一个栈，我们开始遍历输入字符串，如果当前字符为左半边括号时，则将其压入栈中，如果遇到右半边括号时，若此时栈为空，则直接返回false，如不为空，则取出栈顶元素，若为对应的左半边括号，则继续循环，反之返回false，代码如下：

```
1 class Solution {
2 public:
3     bool isValid(string s) {
4         stack<char> parentheses;
5         for (int i = 0; i < s.size(); ++i) {
6             if (s[i] == '(' || s[i] == '[' || s[i] == '{') parentheses.push(s[i]);
7             else {
8                 if (parentheses.empty()) return false;
9                 if (s[i] == ')' && parentheses.top() != '(') return false;
10                if (s[i] == ']' && parentheses.top() != '[') return false;
11                if (s[i] == '}' && parentheses.top() != '{') return false;
12                 parentheses.pop();
13             }
14         }
15         return parentheses.empty();
16     }
17 }
```

21. 合并两个有序链表

将两个有序链表合并为一个新的有序链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

示例：

输入: 1->2->4, 1->3->4
输出: 1->1->2->3->4->4

这道混合插入有序链表和我之前那篇混合插入有序数组非常的相似Merge Sorted Array，仅仅是数据结构由数组换成了链表而已，代码写起来反而更简洁。具体思想就是新建一个链表，然后比较两个链表中的元素值，把较小的那个链到新链表中，由于两个输入链表的长度可能不同，所以最终会有一个链表先完成插入所有元素，则直接另一个未完成的链表直接链入新链表的末尾。代码如下：

解法1：

```

1 class Solution {
2 public:
3     ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
4         ListNode *dummy = new ListNode(-1), *cur = dummy;
5         while (l1 && l2) {
6             if (l1->val < l2->val) {
7                 cur->next = l1;
8                 l1 = l1->next;
9             } else {
10                 cur->next = l2;
11                 l2 = l2->next;
12             }
13             cur = cur->next;
14         }
15         cur->next = l1 ? l1 : l2;
16         return dummy->next;
17     }
18 };

```

下面我们来看递归的写法，当某个链表为空了，就返回另一个。然后核心还是比较当前两个节点值大小，如果l1的小，那么对于l1的下一个节点和l2调用递归函数，将返回值赋值给l1.next，然后返回l1；否则就对于l2的下一个节点和l1调用递归函数，将返回值赋值给l2.next，然后返回l2，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
4         if (!l1) return l2;
5         if (!l2) return l1;
6         if (l1->val < l2->val) {
7             l1->next = mergeTwoLists(l1->next, l2);
8             return l1;
9         } else {
10            l2->next = mergeTwoLists(l1, l2->next);
11            return l2;
12        }
13    }
14 };

```

下面这种递归的写法去掉了if从句，看起来更加简洁一些，但是思路并没有什么不同：

```

1 class Solution {
2 public:
3     ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
4         if (!l1) return l2;
5         if (!l2) return l1;
6         ListNode *head = l1->val < l2->val ? l1 : l2;
7         ListNode *nonhead = l1->val < l2->val ? l2 : l1;
8         head->next = mergeTwoLists(head->next, nonhead);
9         return head;
10    }
11 };

```

我们还可以三行搞定，简直丧心病狂有木有！

```

1 class Solution {
2 public:
3     ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
4         if (!l1 || (l2 && l1->val > l2->val)) swap(l1, l2);
5         if (l1) l1->next = mergeTwoLists(l1->next, l2);
6         return l1;
7     }
8 };

```

22. 括号生成

给出 n 代表生成括号的对数，请你写出一个函数，使其能够生成所有可能的并且有效的括号组合。

例如，给出 n = 3，生成结果为：

```

[
    "((()))",
    "(()())",
    "(())()",
    "()(())",
    "()()()"
]

```

在LeetCode中有关括号的题共有三道，除了这一道的另外两道是 Valid Parentheses 验证括号和 Longest Valid Parentheses 最长有效括号，这道题给定一个数字n，让生成共有n个括号的所有正确的形式，对于这种列出所有结果的题首先还是考虑用递归 Recursion 来解，由于字符串只有左括号和右括号两种字符，而且最终结果必定是左括号3个，右括号3个，所以我们定义两个变量left和right分别表示剩余左右括号的个数，如果在某次递归时，左括号的个数大于右括号的个数，说明此时生成的字符串中右括号的个数大于左括号的个数，即会出现')('这样的非法串，所以这种情况直接返回，不继续处理。如果left和right都为0，则说明此时生成的字符串已有3个左括号和3个右括号，且字符串合法，则存入结果中后返回。如果以上两种情况都不满足，若此时 left 大于 0，则调用递归函数，注意参数的更新，若right大于0，则调用递归函数，同样要更新参数。代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<string> generateParenthesis(int n) {
4         vector<string> res;
5         generateParenthesisDFS(n, n, "", res);
6         return res;
7     }
8     void generateParenthesisDFS(int left, int right, string out, vector<string> &res) {
9         if (left > right) return;
10        if (left == 0 && right == 0) res.push_back(out);
11        else {
12            if (left > 0) generateParenthesisDFS(left - 1, right, out + '(', res);
13            if (right > 0) generateParenthesisDFS(left, right - 1, out + ')', res);
14        }
15    }
16 };

```

再来看那一种方法，这种方法是CareerCup书上给的方法，感觉也是满巧妙的一种方法，这种方法的思想是找左括号，每找到一个左括号，就在其后面加一个完整的括号，最后再在开头加一个0，就形成了所有的情况，需要注意的是，有时候会出现重复的情况，所以我们用set数据结构，好处是如果遇到重复项，不会加入到结果中，最后我们再把set转为vector即可，参见代码如下：

n=1: 0

n=2: (0) 00

n=3:)(() ((0 000

解法2:

```

1 | class Solution {
2 | public:
3 |     vector<string> generateParenthesis(int n) {
4 |         set<string> t;
5 |         if (n == 0) t.insert("");
6 |         else {
7 |             vector<string> pre = generateParenthesis(n - 1);
8 |             for (auto a : pre) {
9 |                 for (int i = 0; i < a.size(); ++i) {
10 |                     if (a[i] == '(') {
11 |                         a.insert(a.begin() + i + 1, '(');
12 |                         a.insert(a.begin() + i + 2, ')');
13 |                         t.insert(a);
14 |                         a.erase(a.begin() + i + 1, a.begin() + i + 3);
15 |                     }
16 |                 }
17 |                 t.insert("()" + a);
18 |             }
19 |         }
20 |         return vector<string>(t.begin(), t.end());
21 |     }
22 | };

```

CPP

23. 合并K个排序链表

合并 k 个排序链表，返回合并后的排序链表。请分析和描述算法的复杂度。

示例:

输入:

```

[
  1->4->5,
  1->3->4,
  2->6
]
输出: 1->1->2->3->4->5->6

```

这道题让我们合并k个有序链表，之前我们做过一道Merge Two Sorted Lists 混合插入有序链表，是混合插入两个有序链表。这道题增加了难度，变成合并k个有序链表了，但是不管合并几个，基本还是要两两合并。那么我们首先考虑的方法是能不能利用之前那道题的解法来解答此题。答案是肯定的，但是需要修改，怎么修改呢，最先想到的就是两两合并，就是前两个先合并，合并好了再跟第三个，然后第四个直到第k个。这样的思路是对的，但是效率不高，没法通过OJ，所以我们只能换一种思路，这里就需要用到分治法 Divide and Conquer Approach。简单来说就是不停的对半划分，比如k个链表先划分为合并两个k/2个链表的任务，再不停的往下划分，直到划分成只有一个或两个链表的任务，开始合并。举个例子来说比如合并6个链表，那么按照分治法，我们首先分别合并1和4,2和5,3和6。这样下一次只需合并3个链表，我们再合并1和3，最后和2合并就可以了。参见代码如下：

解法1：

CPP

```

1 class Solution {
2 public:
3     ListNode *mergeKLists(vector<ListNode *> &lists) {
4         if (lists.size() == 0) return NULL;
5         int n = lists.size();
6         while (n > 1) {
7             int k = (n + 1) / 2;
8             for (int i = 0; i < n / 2; ++i) {
9                 lists[i] = mergeTwoLists(lists[i], lists[i + k]);
10            }
11            n = k;
12        }
13        return lists[0];
14    }
15
16    ListNode *mergeTwoLists(ListNode *l1, ListNode *l2) {
17        ListNode *head = new ListNode(-1);
18        ListNode *cur = head;
19        while (l1 && l2) {
20            if (l1->val < l2->val) {
21                cur->next = l1;
22                l1 = l1->next;
23            } else {
24                cur->next = l2;
25                l2 = l2->next;
26            }
27            cur = cur->next;
28        }
29        if (l1) cur->next = l1;
30        if (l2) cur->next = l2;
31        return head->next;
32    }
33}

```

我们再来看另一种解法，这种解法利用了最小堆这种数据结构，我们首先把k个链表的首元素都加入最小堆中，它们会自动排好序。然后我们每次取出最小的那个元素加入我们最终结果的链表中，然后把取出元素的下一个元素再加入堆中，下次仍从堆中取出最小的元素做相同的操作，以此类推，直到堆中没有元素了，此时k个链表也合并为了一个链表，返回首节点即可，代码如下：

```

1 struct cmp {
2     bool operator () (ListNode *a, ListNode *b) {
3         return a->val > b->val;
4     }
5 };
6
7 class Solution {
8 public:
9     ListNode *mergeKLists(vector<ListNode *> &lists) {
10        priority_queue<ListNode*, vector<ListNode*>, cmp> q;
11        for (int i = 0; i < lists.size(); ++i) {
12            if (lists[i]) q.push(lists[i]);
13        }
14        ListNode *head = NULL, *pre = NULL, *tmp = NULL;
15        while (!q.empty()) {
16            tmp = q.top();
17            q.pop();
18            if (!pre) head = tmp;
19            else pre->next = tmp;
20            pre = tmp;
21            if (tmp->next) q.push(tmp->next);
22        }
23        return head;
24    }
25 }

```

24. 两两交换链表中的节点

给定一个链表，两两交换其中相邻的节点，并返回交换后的链表。

示例：

给定 1->2->3->4，你应该返回 2->1->4->3.

说明：

- 你的算法只能使用常数的额外空间。
- 你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

这道题不算难，是基本的链表操作题，我们可以分别用递归和迭代来实现。对于迭代实现，还是需要建立dummy节点，注意在连接节点的时候，最好画个图，以免把自己搞晕了，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     ListNode* swapPairs(ListNode* head) {
4         ListNode *dummy = new ListNode(-1), *pre = dummy;
5         dummy->next = head;
6         while (pre->next && pre->next->next) {
7             ListNode *t = pre->next->next;
8             pre->next->next = t->next;
9             t->next = pre->next;
10            pre->next = t;
11            pre = t->next;
12        }
13        return dummy->next;
14    }
15 };

```

递归的写法就更简洁了，实际上利用了回溯的思想，递归遍历到链表末尾，然后先交换末尾两个，然后依次往前交换：

```

1 class Solution {
2 public:
3     ListNode* swapPairs(ListNode* head) {
4         if (!head || !head->next) return head;
5         ListNode *t = head->next;
6         head->next = swapPairs(head->next->next);
7         t->next = head;
8         return t;
9     }
10 };

```

25. k个一组翻转链表

给出一个链表，每 k 个节点一组进行翻转，并返回翻转后的链表。

k 是一个正整数，它的值小于或等于链表的长度。如果节点总数不是 k 的整数倍，那么将最后剩余节点保持原有顺序。

示例：

给定这个链表：1→2→3→4→5

当 k = 2 时，应当返回：2→1→4→3→5

当 k = 3 时，应当返回：3→2→1→4→5

说明：

- 你的算法只能使用常数的额外空间。
- 你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

这道题让我们以每k个为一组来翻转链表，实际上是把原链表分成若干小段，然后分别对其进行翻转，那么肯定总共需要两个函数，一个是用来分段的，一个是用来翻转的，我们就以题目中给的例子来看，对于给定链表1→2→3→4→5，一般在处理链表问题时，我们大多时候都会在开头再加一个dummy node，因为翻转链表时头结点可能会变化，为了记录当前最新的头结点的位置而引入的dummy node，那么我们加入dummy node后的链表变为-1→1→2→3→4→5，如果k为3的话，我们的目标是将1,2,3翻转一下，那么我们需要一些指针，pre和next分别指向要翻转的链表的前后的位置，然后翻转后pre的位置更新到如下新的位置：

```
-1->1->2->3->4->5
|           |
pre         next

-1->3->2->1->4->5
|   |
pre next
```

以此类推，只要next走过k个节点，就可以调用翻转函数来进行局部翻转了，代码如下所示：

```
1 class Solution {
2 public:
3     ListNode *reverseKGroup(ListNode *head, int k) {
4         if (!head || k == 1) return head;
5         ListNode *dummy = new ListNode(-1);
6         ListNode *pre = dummy, *cur = head;
7         dummy->next = head;
8         int i = 0;
9         while (cur) {
10             ++i;
11             if (i % k == 0) {
12                 pre = reverseOneGroup(pre, cur->next);
13                 cur = pre->next;
14             } else {
15                 cur = cur->next;
16             }
17         }
18         return dummy->next;
19     }
20     ListNode *reverseOneGroup(ListNode *pre, ListNode *next) {
21         ListNode *last = pre->next;
22         ListNode *cur = last->next;
23         while (cur != next) {
24             last->next = cur->next;
25             cur->next = pre->next;
26             pre->next = cur;
27             cur = last->next;
28         }
29         return last;
30     }
31 }
```

我们也可以在一个函数中完成，我们首先遍历整个链表，统计出链表的长度，然后如果长度大于等于k，我们开始交换节点，当k=2时，每段我们只需要交换一次，当k=3时，每段需要交换2此，所以i从1开始循环，注意交换一段后更新pre指针，然后num自减k，直到num<k时循环结束，参见代码如下：

```

1 class Solution {
2 public:
3     ListNode* reverseKGroup(ListNode* head, int k) {
4         ListNode *dummy = new ListNode(-1), *pre = dummy, *cur = pre;
5         dummy->next = head;
6         int num = 0;
7         while (cur = cur->next) ++num;
8         while (num >= k) {
9             cur = pre->next;
10            for (int i = 1; i < k; ++i) {
11                ListNode *t = cur->next;
12                cur->next = t->next;
13                t->next = pre->next;
14                pre->next = t;
15            }
16            pre = cur;
17            num -= k;
18        }
19        return dummy->next;
20    }
21 };

```

我们也可以使用递归来做，我们用head记录每段的开始位置，cur记录结束位置的下一个节点，然后我们调用reverse函数来将这段翻转，然后得到一个new_head，原来的head就变成了末尾，这时候后面接上递归调用下一段得到的新节点，返回new_head即可，参见代码如下：

```

1 class Solution {
2 public:
3     ListNode* reverseKGroup(ListNode* head, int k) {
4         ListNode *cur = head;
5         for (int i = 0; i < k; ++i) {
6             if (!cur) return head;
7             cur = cur->next;
8         }
9         ListNode *new_head = reverse(head, cur);
10        head->next = reverseKGroup(cur, k);
11        return new_head;
12    }
13    ListNode* reverse(ListNode* head, ListNode* tail) {
14        ListNode *pre = tail;
15        while (head != tail) {
16            ListNode *t = head->next;
17            head->next = pre;
18            pre = head;
19            head = t;
20        }
21        return pre;
22    }
23 };

```

26. 删除排序数组中的重复项

给定一个排序数组，你需要在原地删除重复出现的元素，使得每个元素只出现一次，返回移除后数组的新长度。

不要使用额外的数组空间，你必须在原地修改输入数组并在使用 O(1) 额外空间的条件下完成。

示例 1:

给定数组 `nums = [1,1,2]`,

函数应该返回新的长度 2，并且原数组 `nums` 的前两个元素被修改为 1, 2。

你不需要考虑数组中超出新长度后面的元素。

示例 2:

给定 `nums = [0,0,1,1,1,2,2,3,3,4]`,

函数应该返回新的长度 5，并且原数组 `nums` 的前五个元素被修改为 0, 1, 2, 3, 4。

你不需要考虑数组中超出新长度后面的元素。

说明

为什么返回数值是整数，但输出的答案是数组呢？

请注意，输入数组是以“引用”方式传递的，这意味着在函数里修改输入数组对于调用者是可见的。

你可以想象内部操作如下：

```
// nums 是以“引用”方式传递的。也就是说，不对实参做任何拷贝
int len = removeDuplicates(nums);

// 在函数里修改输入数组对于调用者是可见的。
// 根据你的函数返回的长度，它会打印出数组中该长度范围内的所有元素。
for (int i = 0; i < len; i++) {
    print(nums[i]);
}
```

这道题要我们从有序数组中去除重复项，和之前那道 Remove Duplicates from Sorted List 移除有序链表中的重复项的题很类似，但是要简单一些，因为毕竟数组的值可以通过下标直接访问，而链表不行。那么这道题的解题思路是，我们使用快慢指针来记录遍历的坐标，最开始时两个指针都指向第一个数字，如果两个指针指的数字相同，则快指针向前走一步，如果不同，则两个指针都向前走一步，这样当快指针走完整个数组后，慢指针当前的坐标加1就是数组中不同数字的个数，代码如下：

解法1:

```
1 class Solution {
2 public:
3     int removeDuplicates(vector<int>& nums) {
4         if (nums.empty()) return 0;
5         int pre = 0, cur = 0, n = nums.size();
6         while (cur < n) {
7             if (nums[pre] == nums[cur]) ++cur;
8             else nums[++pre] = nums[cur++];
9         }
10        return pre + 1;
11    }
12};
```

CPP

我们也可以用for循环来写，这里的j就是上面解法中的pre，i就是cur，所以本质上都是一样的，参见代码如下：

解法2:

```

1 class Solution {
2 public:
3     int removeDuplicates(vector<int>& nums) {
4         if (nums.empty()) return 0;
5         int j = 0, n = nums.size();
6         for (int i = 0; i < n; ++i) {
7             if (nums[i] != nums[j]) nums[++j] = nums[i];
8         }
9         return j + 1;
10    }
11 };

```

27. 移除元素

给定一个数组 `nums` 和一个值 `val`, 你需要原地移除所有数值等于 `val` 的元素, 返回移除后数组的新长度。

不要使用额外的数组空间, 你必须在原地修改输入数组并在使用 O(1) 额外空间的条件下完成。

元素的顺序可以改变。你不需要考虑数组中超出新长度后面的元素。

示例 1:

给定 `nums = [3,2,2,3]`, `val = 3`,

函数应该返回新的长度 2, 并且 `nums` 中的前两个元素均为 2。

你不需要考虑数组中超出新长度后面的元素。

示例 2:

给定 `nums = [0,1,2,2,3,0,4,2]`, `val = 2`,

函数应该返回新的长度 5, 并且 `nums` 中的前五个元素为 0, 1, 3, 0, 4。

注意这五个元素可为任意顺序。

你不需要考虑数组中超出新长度后面的元素。

说明:

为什么返回数值是整数, 但输出的答案是数组呢?

请注意, 输入数组是以“引用”方式传递的, 这意味着在函数里修改输入数组对于调用者是可见的。

你可以想象内部操作如下:

```

// nums 是以“引用”方式传递的。也就是说, 不对实参作任何拷贝
int len = removeElement(nums, val);

// 在函数里修改输入数组对于调用者是可见的。
// 根据你的函数返回的长度, 它会打印出数组中该长度范围内的所有元素。
for (int i = 0; i < len; i++) {
    print(nums[i]);
}

```

这道题让我们移除一个数组中和给定值相同的数字，并返回新的数组的长度。是一道比较容易的题，我们只需要一个变量用来计数，然后遍历原数组，如果当前的值和给定值不同，我们就把当前值覆盖计数变量的位置，并将计数变量加1。代码如下：

```
1 class Solution {
2 public:
3     int removeElement(vector<int>& nums, int val) {
4         int res = 0;
5         for (int i = 0; i < nums.size(); ++i) {
6             if (nums[i] != val) nums[res++] = nums[i];
7         }
8         return res;
9     }
10 };
```

CPP

28. 实现strStr()

实现 strStr() 函数。

给定一个 haystack 字符串和一个 needle 字符串，在 haystack 字符串中找出 needle 字符串出现的第一个位置 (从0开始)。如果不存在，则返回 -1。

示例 1:

输入: haystack = "hello", needle = "ll"
输出: 2

示例 2:

输入: haystack = "aaaaa", needle = "bba"
输出: -1

说明:

当 needle 是空字符串时，我们应当返回什么值呢？这是一个在面试中很好的问题。

对于本题而言，当 needle 是空字符串时我们应当返回 0。这与C语言的 strstr() 以及 Java的 indexOf() 定义相符。

这道题让我们在一个字符串中找另一个字符串第一次出现的位置，那我们首先要做一些判断，如果子字符串为空，则返回0，如果子字符串长度大于母字符串长度，则返回-1。然后我们开始遍历母字符串，我们并不需要遍历整个母字符串，而是遍历到剩下的长度和子字符串相等的位置即可，这样可以提高运算效率。然后对于每一个字符，我们都遍历一遍子字符串，一个一个字符的对应比较，如果对应位置有不等的，则跳出循环，如果一直都没有跳出循环，则说明子字符串出现了，则返回起始位置即可，代码如下：

```

1 class Solution {
2     public:
3         int strStr(string haystack, string needle) {
4             if (needle.empty()) return 0;
5             int m = haystack.size(), n = needle.size();
6             if (m < n) return -1;
7             for (int i = 0; i <= m - n; ++i) {
8                 int j = 0;
9                 for (j = 0; j < n; ++j) {
10                     if (haystack[i + j] != needle[j]) break;
11                 }
12                 if (j == n) return i;
13             }
14         }
15     };
16 };

```

29. 两数相除

给定两个整数，被除数 dividend 和除数 divisor。将两数相除，要求不使用乘法、除法和 mod 运算符。

返回被除数 dividend 除以除数 divisor 得到的商。

示例 1:

输入: dividend = 10, divisor = 3
输出: 3

示例 2:

输入: dividend = 7, divisor = -3
输出: -2

说明:

- 被除数和除数均为 32 位有符号整数
- 除数不为 0
- 假设我们的环境只能存储 32 位有符号整数，其数值范围是 $[-2^{31}, 2^{31} - 1]$ 。本题中，如果除法结果溢出，则返回 $2^{31} - 1$

这道题让我们求两数相除，而且规定我们不能用乘法、除法和取余操作，那么我们还可以用另一神器位操作Bit Operation，思路是，如果被除数大于或等于除数，则进行如下循环，定义变量t等于除数，定义计数p，当t的两倍小于等于被除数时，进行如下循环，t扩大一倍，p扩大一倍，然后更新res和m。这道题的OJ给的一些test case非常的讨厌，因为输入的都是int型，比如被除数是-2147483648，在int范围内，当除数是-1时，结果就超出了int范围，需要返回INT_MAX，所以对于这种情况我们就开始用if判定，将其和除数为0的情况放一起判定，返回INT_MAX。然后我们还要根据被除数和除数的正负来确定返回值的正负，这里我们采用长整型long来完成所有的计算，最后返回值乘以符号即可，代码如下：

解法1:

```

1 class Solution {
2 public:
3     int divide(int dividend, int divisor) {
4         if (divisor == 0 || (dividend == INT_MIN && divisor == -1)) return INT_MAX;
5         long long m = abs((long long)dividend), n = abs((long long)divisor), res = 0;
6         int sign = ((dividend < 0) ^ (divisor < 0)) ? -1 : 1;
7         if (n == 1) return sign == 1 ? m : -m;
8         while (m >= n) {
9             long long t = n, p = 1;
10            while (m >= (t << 1)) {
11                t <<= 1;
12                p <<= 1;
13            }
14            res += p;
15            m -= t;
16        }
17        return sign == 1 ? res : -res;
18    }
19 };

```

我们可以使上面的解法变得更加简洁：

```

1 class Solution {
2 public:
3     int divide(int dividend, int divisor) {
4         long long m = abs((long long)dividend), n = abs((long long)divisor), res = 0;
5         if (m < n) return 0;
6         while (m >= n) {
7             long long t = n, p = 1;
8             while (m > (t << 1)) {
9                 t <<= 1;
10                p <<= 1;
11            }
12            res += p;
13            m -= t;
14        }
15        if ((dividend < 0) ^ (divisor < 0)) res = -res;
16        return res > INT_MAX ? INT_MAX : res;
17    }
18 };

```

我们也可以通过递归的方法来解，思路都一样：

```

1 class Solution {
2 public:
3     int divide(int dividend, int divisor) {
4         long long res = 0;
5         long long m = abs((long long)dividend), n = abs((long long)divisor);
6         if (m < n) return 0;
7         long long t = n, p = 1;
8         while (m > (t << 1)) {
9             t <<= 1;
10            p <<= 1;
11        }
12        res += p + divide(m - t, n);
13        if ((dividend < 0) ^ (divisor < 0)) res = -res;
14        return res > INT_MAX ? INT_MAX : res;
15    }
16 };

```

30. 与所有单词相关联的字串

给定一个字符串 s 和一些长度相同的单词 words。在 s 中找出可以恰好串联 words 中所有单词的子串的起始位置。

注意子串要与 words 中的单词完全匹配，中间不能有其他字符，但不需要考虑 words 中单词串联的顺序。

示例 1:

输入：

```
s = "barfoothefoobarman",
words = ["foo", "bar"]
```

输出：[0,9]

解释：从索引 0 和 9 开始的子串分别是 "barfoor" 和 "foobar" 。

输出的顺序不重要，[9,0] 也是有效答案。

示例 2:

输入：

```
s = "wordgoodstudentgoodword",
words = ["word", "student"]
```

输出：[]

这道题让我们求串联所有单词的子串，就是说给定一个长字符串，再给定几个长度相同的单词，让我们找出串联给定所有单词的子串的起始位置，还是蛮有难度的一道题。这道题我们需要用到两个哈希表，第一个哈希表先把所有的单词存进去，然后从开头开始一个个遍历，停止条件为当剩余字符个数小于单词集里所有字符的长度。这时候我们需要定义第二个哈希表，然后每次找出给定单词长度的子串，看其是否在第一个哈希表里，如果没有，则break，如果有，则加入第二个哈希表，但相同的词只能出现一次，如果多了，也break。如果正好匹配完给定单词集里所有的单词，则把i存入结果中，具体参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     vector<int> findSubstring(string s, vector<string>& words) {
4         vector<int> res;
5         if (s.empty() || words.empty()) return res;
6         int n = words.size(), m = words[0].size();
7         unordered_map<string, int> m1;
8         for (auto &a : words) ++m1[a];
9         for (int i = 0; i <= (int)s.size() - n * m; ++i) {
10             unordered_map<string, int> m2;
11             int j = 0;
12             for (j = 0; j < n; ++j) {
13                 string t = s.substr(i + j * m, m);
14                 if (m1.find(t) == m1.end()) break;
15                 ++m2[t];
16                 if (m2[t] > m1[t]) break;
17             }
18             if (j == n) res.push_back(i);
19         }
20     return res;
21 }
22 };

```

这道题还有一种O(n)时间复杂度的解法，设计思路非常巧妙，但是感觉很难想出来，博主目测还未到达这种水平。这种方法不再是一个字符一个字符的遍历，而是一个词一个词的遍历，比如根据题目中的例子，字符串s的长度n为18，words数组中有两个单词(cnt=2)，每个单词的长度len均为3，那么遍历的顺序为0, 3, 6, 8, 12, 15，然后偏移一个字符1, 4, 7, 9, 13, 16，然后再偏移一个字符2, 5, 8, 10, 14, 17，这样就可以把所有情况都遍历到，我们还是先用一个哈希表m1来记录words里的所有词，然后我们从0开始遍历，用left来记录左边界的位置，count表示当前已经匹配的单词的个数。然后我们一个单词一个单词的遍历，如果当前遍历的到的单词t在m1中存在，那么我们将其加入另一个哈希表m2中，如果在m2中个数小于等于m1中的个数，那么我们count自增1，如果大于了，那么需要做一些处理，比如下面这种情况，s = barfoofoo, words = {bar, foo, abc}，我们给words中新加了一个abc，目的是为了遍历到barfoo不会停止，那么当遍历到第二foo的时候，m2[foo]=2，而此时m1[foo]=1，这是后已经不连续了，所以我们要移动左边界left的位置，我们先把第一个词t1=bar拿出来，然后将m2[t1]自减1，如果此时m2[t1]<m1[t1]了，说明一个匹配没了，那么对应的count也要自减1，然后左边界加上个len，这样就可以了。如果某个时刻count和cnt相等了，说明我们成功匹配了一个位置，那么将当前左边界left存入结果res中，此时去掉最左边的一个词，同时count自减1，左边界右移len，继续匹配。如果我们匹配到一个不在m1中的词，那么说明跟前面已经断开了，我们重置m2，count为0，左边界left移到j+len，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<int> findSubstring(string s, vector<string>& words) {
4         if (s.empty() || words.empty()) return {};
5         vector<int> res;
6         int n = s.size(), cnt = words.size(), len = words[0].size();
7         unordered_map<string, int> m1;
8         for (string w : words) ++m1[w];
9         for (int i = 0; i < len; ++i) {
10             int left = i, count = 0;
11             unordered_map<string, int> m2;
12             for (int j = i; j <= n - len; j += len) {
13                 string t = s.substr(j, len);
14                 if (m1.count(t)) {
15                     ++m2[t];
16                     if (m2[t] <= m1[t]) {
17                         ++count;
18                     } else {
19                         while (m2[t] > m1[t]) {
20                             string t1 = s.substr(left, len);
21                             --m2[t1];
22                             if (m2[t1] < m1[t1]) --count;
23                             left += len;
24                         }
25                     }
26                     if (count == cnt) {
27                         res.push_back(left);
28                         --m2[s.substr(left, len)];
29                         --count;
30                         left += len;
31                     }
32                 } else {
33                     m2.clear();
34                     count = 0;
35                     left = j + len;
36                 }
37             }
38         }
39         return res;
40     }
41 };

```

31. 下一个排列

实现获取下一个排列的函数，算法需要将给定数字序列重新排列成字典序中下一个更大的排列。

如果不存在下一个更大的排列，则将数字重新排列成最小的排列（即升序排列）。

必须原地修改，只允许使用额外常数空间。

以下是一些例子，输入位于左侧列，其相应输出位于右侧列。

1,2,3 → 1,3,2
 3,2,1 → 1,2,3
 1,1,5 → 1,5,1

这道题让我们求下一个排列顺序，有题目中给的例子可以看出来，如果给定数组是降序，则说明是全排列的最后一一种情况，则下一个排列就是最初初始情况，可以参见之前的博客 Permutations 全排列。我们再来看下面一个例子，有如下的一个数组

1 2 7 4 3 1

下一个排列为：

1 3 1 2 4 7

那么是如何得到的呢，我们通过观察原数组可以发现，如果从末尾往前看，数字逐渐变大，到了2时才减小的，然后我们再从后往前找第一个比2大的数字，是3，那么我们交换2和3，再把此时3后面的所有数字转置一下即可，步骤如下：

1 2 7 4 3 1

1 2 7 4 3 1

1 3 7 4 2 1

1 3 1 2 4 7

解法1：

```
1 class Solution {
2 public:
3     void nextPermutation(vector<int> &num) {
4         int i, j, n = num.size();
5         for (i = n - 2; i >= 0; --i) {
6             if (num[i + 1] > num[i]) {
7                 for (j = n - 1; j > i; --j) {
8                     if (num[j] > num[i]) break;
9                 }
10                swap(num[i], num[j]);
11                reverse(num.begin() + i + 1, num.end());
12                return;
13            }
14        }
15        reverse(num.begin(), num.end());
16    }
17}
```

CPP

下面这种写法更简洁一些，但是整体思路和上面的解法没有什么区别，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     void nextPermutation(vector<int>& nums) {int n = nums.size(), i = n - 2, j = n - 1;
4         while (i >= 0 && nums[i] >= nums[i + 1]) --i;
5         if (i >= 0) {
6             while (nums[j] <= nums[i]) --j;
7             swap(nums[i], nums[j]);
8         }
9         reverse(nums.begin() + i + 1, nums.end());
10    }
11}
```

CPP

32. 最长有效括号

给定一个只包含 '(' 和 ')' 的字符串，找出最长的包含有效括号的子串的长度。

示例 1:

输入: "()"
 输出: 2
 解释: 最长有效括号子串为 "()"

示例 2:

输入: ")()()"
 输出: 4
 解释: 最长有效括号子串为 "()()"

这道求最长有效括号比之前那道 Valid Parentheses 验证括号难度要大一些，这里我们还是借助栈来求解，需要定义个start变量来记录合法括号串的起始位置，我们遍历字符串，如果遇到左括号，则将当前下标压入栈，如果遇到右括号，如果当前栈为空，则将下一个坐标位置记录到start，如果栈不为空，则将栈顶元素取出，此时若栈为空，则更新结果和*i - start + 1*中的较大值，否则更新结果和*i - 栈顶元素*中的较大值，代码如下：

```
1 class Solution {
2 public:
3     int longestValidParentheses(string s) {
4         int res = 0, start = 0;
5         stack<int> m;
6         for (int i = 0; i < s.size(); ++i) {
7             if (s[i] == '(') m.push(i);
8             else if (s[i] == ')') {
9                 if (m.empty()) start = i + 1;
10                else {
11                    m.pop();
12                    res = m.empty() ? max(res, i - start + 1) : max(res, i - m.top());
13                }
14            }
15        }
16        return res;
17    }
18 }
```

CPP

33. 搜索旋转排序数组

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

(例如，数组 [0,1,2,4,5,6,7] 可能变为 [4,5,6,7,0,1,2])。

搜索一个给定的目标值，如果数组中存在这个目标值，则返回它的索引，否则返回 -1 。

你可以假设数组中不存在重复的元素。

你的算法时间复杂度必须是 $O(\log n)$ 级别。

示例 1:

输入: nums = [4,5,6,7,0,1,2], target = 0
 输出: 4

示例 2:

输入: `nums = [4,5,6,7,0,1,2], target = 3`
 输出: -1

这道题让在旋转数组中搜索一个给定值，若存在返回坐标，若不存在返回-1。我们还是考虑二分搜索法，但是这道题的难点在于我们不知道原数组在哪旋转了，我们还是用题目中给的例子来分析，对于数组[0 1 2 4 5 6 7] 共有下列七种旋转方法：

0	1	2	4	5	6	7
7	0	1	2	4	5	6
6	7	0	1	2	4	5
5	6	7	0	1	2	4
4	5	6	7	0	1	2
2	4	5	6	7	0	1
1	2	4	5	6	7	0

二分搜索法的关键在于获得了中间数后，判断下面要搜索左半段还是右半段，我们观察上面红色的数字都是升序的，由此我们可以观察出规律，如果中间的数小于最右边的数，则右半段是有序的，若中间数大于最右边数，则左半段是有序的，我们只要在有序的半段里用首尾两个数组来判断目标值是否在这一区域内，这样就可以确定保留哪半边了，代码如下：

```
1 class Solution {
2 public:
3     int search(vector<int>& nums, int target) {
4         int left = 0, right = nums.size() - 1;
5         while (left <= right) {
6             int mid = left + (right - left) / 2;
7             if (nums[mid] == target) return mid;
8             else if (nums[mid] < nums[right]) {
9                 if (nums[mid] < target && nums[right] >= target) left = mid + 1;
10                else right = mid - 1;
11            } else {
12                if (nums[left] <= target && nums[mid] > target) right = mid - 1;
13                else left = mid + 1;
14            }
15        }
16        return -1;
17    }
18};
```

CPP

34. 在排序数组中查找元素的第一个和最后一个位置

给定一个按照升序排列的整数数组 `nums`，和一个目标值 `target`。找出给定目标值在数组中的开始位置和结束位置。

你的算法时间复杂度必须是 $O(\log n)$ 级别。

如果数组中不存在目标值，返回 `[-1, -1]`。

示例 1:

输入: nums = [5,7,7,8,8,10], target = 8
 输出: [3,4]

示例 2:

输入: nums = [5,7,7,8,8,10], target = 6
 输出: [-1,-1]

这道题让我们在一个有序整数数组中寻找相同目标值的起始和结束位置，而且限定了时间复杂度为 $O(\log n)$ ，这是典型的二分查找法的时间复杂度，所以这道题我们也要用此方法，我们的思路是首先对原数组使用二分查找法，找出其中一个目标值的位置，然后向两边搜索找出起始和结束的位置，代码如下：

解法1:

```
1 class Solution {
2 public:
3     vector<int> searchRange(vector<int>& nums, int target) {
4         int idx = search(nums, 0, nums.size() - 1, target);
5         if (idx == -1) return {-1, -1};
6         int left = idx, right = idx;
7         while (left > 0 && nums[left - 1] == nums[idx]) --left;
8         while (right < nums.size() - 1 && nums[right + 1] == nums[idx]) ++right;
9         return {left, right};
10    }
11    int search(vector<int>& nums, int left, int right, int target) {
12        if (left > right) return -1;
13        int mid = left + (right - left) / 2;
14        if (nums[mid] == target) return mid;
15        else if (nums[mid] < target) return search(nums, mid + 1, right, target);
16        else return search(nums, left, mid - 1, target);
17    }
18};
```

CPP

可能有些人会觉得上面的算法不是严格意义上的 $O(\log n)$ 的算法，因为在最坏的情况下会变成 $O(n)$ ，比如当数组里的数全是目标值的话，从中间向两边找边界就会一直遍历完整个数组，那么我们下面来看一种真正意义上的 $O(\log n)$ 的算法，使用两次二分查找法，第一次找到左边界，第二次调用找到右边界即可，具体代码如下：

解法2:

```

1 class Solution {
2 public:
3     vector<int> searchRange(vector<int>& nums, int target) {
4         vector<int> res(2, -1);
5         int left = 0, right = nums.size() - 1;
6         while (left < right) {
7             int mid = left + (right - left) / 2;
8             if (nums[mid] < target) left = mid + 1;
9             else right = mid;
10        }
11        if (nums[right] != target) return res;
12        res[0] = right;
13        right = nums.size();
14        while (left < right) {
15            int mid = left + (right - left) / 2;
16            if (nums[mid] <= target) left = mid + 1;
17            else right = mid;
18        }
19        res[1] = left - 1;
20        return res;
21    }
22 }

```

35. 搜索插入位置

给定一个排序数组和一个目标值，在数组中找到目标值，并返回其索引。如果目标值不存在于数组中，返回它将会被按顺序插入的位置。

你可以假设数组中无重复元素。

示例 1:

输入: [1,3,5,6], 5
输出: 2

示例 2:

输入: [1,3,5,6], 2
输出: 1

示例 3:

输入: [1,3,5,6], 7
输出: 4

示例 4:

输入: [1,3,5,6], 0
输出: 0

这道题基本没有什么难度，实在不理解为啥还是Medium难度的，完完全全的应该是Easy啊，三行代码搞定的题，只需要遍历一遍原数组，若当前数字大于或等于目标值，则返回当前坐标，如果遍历结束了，说明目标值比数组中任何一个数都要大，则返回数组长度n即可，代码如下：

解法1：

```
1 class Solution {
2 public:
3     int searchInsert(vector<int>& nums, int target) {
4         for (int i = 0; i < nums.size(); ++i) {
5             if (nums[i] >= target) return i;
6         }
7         return nums.size();
8     }
9 };
```

CPP

当然，我们还可以用二分搜索法来优化我们的时间复杂度，而且个人认为这种方法应该是面试官们想要考察的算法吧，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     int searchInsert(vector<int>& nums, int target) {
4         if (nums.back() < target) return nums.size();
5         int left = 0, right = nums.size() - 1;
6         while (left < right) {
7             int mid = left + (right - left) / 2;
8             if (nums[mid] == target) return mid;
9             else if (nums[mid] < target) left = mid + 1;
10            else right = mid;
11        }
12        return right;
13    }
14 };
```

CPP

36. 有效的数独

判断一个 9x9 的数独是否有效。只需要根据以下规则，验证已经填入的数字是否有效即可。

- 数字 1-9 在每一行只能出现一次。
- 数字 1-9 在每一列只能出现一次。
- 数字 1-9 在每一个以粗实线分隔的 3x3 宫内只能出现一次。

5	3			7				
6			1	9	5			
	9	8				6		
8				6				3
4			8		3			1
7				2				6
	6				2	8		
		4	1	9				5
			8			7	9	

上图是一个部分填充的有效的数独。

数独部分空格内已填入了数字，空白格用'.'表示。

示例 1:

输入：

```
[  
  ["5","3",".",".","7",".",".",".","."],  
  ["6",".",".","1","9","5",".",".","."],  
  [".","9","8",".",".",".","6","."],  
  ["8",".",".",".","6",".",".",".","3"],  
  ["4",".",".","8",".","3",".",".","1"],  
  ["7",".",".",".","2",".",".",".","6"],  
  [".","6",".",".",".","2","8","."],  
  [".",".",".","4","1","9",".",".","5"],  
  [".",".",".","8",".",".","7","9"]  
]
```

输出： true

示例 2:

输入：

```
[  
  ["8","3",".",".","7",".",".",".","."],  
  ["6",".",".","1","9","5",".",".","."],  
  [".","9","8",".",".",".","6","."],  
  ["8",".",".","6",".",".",".","3"],  
  ["4",".",".","8",".","3",".",".","1"],  
  ["7",".",".",".","2",".",".",".","6"],  
  [".","6",".",".",".","2","8","."],  
  [".",".",".","4","1","9",".",".","5"],  
  [".",".",".","8",".",".","7","9"]  
]
```

输出： false

解释：除了第一行的第一个数字从 5 改为 8 以外，空格内其他数字均与 示例1 相同。

但由于位于左上角的 3x3 宫内有两个 8 存在，因此这个数独是无效的。

说明：

- 一个有效的数独（部分已被填充）不一定是可解的。
- 只需要根据以上规则，验证已经填入的数字是否有效即可。
- 给定数独序列只包含数字 1-9 和字符'.'

- 给定数独永远是 9x9 形式的。

这道题让我们验证一个方阵是否为数独矩阵，判断标准是看各行各列是否有重复数字，以及每个小的3x3的小方阵里面是否有重复数字，如果都无重复，则当前矩阵是数独矩阵，但不代表待数独矩阵有解，只是单纯的判断当前未填完的矩阵是否是数独矩阵。那么根据数独矩阵的定义，我们在遍历每个数字的时候，就看看包含当前位置的行和列以及3x3小方阵中是否已经出现该数字，那么我们需要三个标志矩阵，分别记录各行，各列，各小方阵是否出现某个数字，其中行和列标志下标很好对应，就是小方阵的下标需要稍稍转换一下，具体代码如下：

```

1 class Solution {
2 public:
3     bool isValidSudoku(vector<vector<char> > &board) {
4         if (board.empty() || board[0].empty()) return false;
5         int m = board.size(), n = board[0].size();
6         vector<vector<bool> > rowFlag(m, vector<bool>(n, false));
7         vector<vector<bool> > colFlag(m, vector<bool>(n, false));
8         vector<vector<bool> > cellFlag(3, vector<vector<bool>>(3, vector<bool>(n, false)));
9         for (int i = 0; i < m; ++i) {
10             for (int j = 0; j < n; ++j) {
11                 if (board[i][j] >= '1' && board[i][j] <= '9') {
12                     int c = board[i][j] - '1';
13                     if (rowFlag[i][c] || colFlag[c][j] || cellFlag[3 * (i / 3) + j / 3][c])
14                         return false;
15                     rowFlag[i][c] = true;
16                     colFlag[c][j] = true;
17                     cellFlag[3 * (i / 3) + j / 3][c] = true;
18                 }
19             }
20         }
21         return true;
22     }
23 };

```

37. 解数独

编写一个程序，通过已填充的空格来解决数独问题。

一个数独的解法需遵循如下规则：

- 数字 1-9 在每一行只能出现一次。
- 数字 1-9 在每一列只能出现一次。
- 数字 1-9 在每一个以粗实线分隔的 3x3 宫内只能出现一次。

空白格用 ' ' 表示。

image::images/question_37_1.png[width="30%", height="35%"]

一个数独。

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

这道求解数独的题是在之前那道 Valid Sudoku 验证数独的基础上的延伸，之前那道题让我们验证给定的数组是否为数独数组，这道让我们求解数独数组，跟此题类似的有 Permutations 全排列，Combinations 组合项，N-Queens N皇后问题等等，其中尤其是跟 N-Queens N皇后问题的解题思路及其相似，对于每个需要填数字的格子带入1到9，每代入一个数字都判定其是否合法，如果合法就继续下一次递归，结束时把数字设回'.'，判断新加入的数字是否合法时，只需要判定当前数字是否合法，不需要判定这个数组是否为数独数组，因为之前加进的数字都是合法的，这样可以使程序更加高效一些，具体实现如代码所示：

```

1 class Solution {
2 public:
3     void solveSudoku(vector<vector<char>> &board) {
4         if (board.empty() || board.size() != 9 || board[0].size() != 9) return;
5         solveSudokuDFS(board, 0, 0);
6     }
7     bool solveSudokuDFS(vector<vector<char>> &board, int i, int j) {
8         if (i == 9) return true;
9         if (j >= 9) return solveSudokuDFS(board, i + 1, 0);
10        if (board[i][j] == '.') {
11            for (int k = 1; k <= 9; ++k) {
12                board[i][j] = (char)(k + '0');
13                if (isValid(board, i, j)) {
14                    if (solveSudokuDFS(board, i, j + 1)) return true;
15                }
16                board[i][j] = '.';
17            }
18        } else {
19            return solveSudokuDFS(board, i, j + 1);
20        }
21        return false;
22    }
23    bool isValid(vector<vector<char>> &board, int i, int j) {
24        for (int col = 0; col < 9; ++col) {
25            if (col != j && board[i][j] == board[i][col]) return false;
26        }
27        for (int row = 0; row < 9; ++row) {
28            if (row != i && board[i][j] == board[row][j]) return false;
29        }
30        for (int row = i / 3 * 3; row < i / 3 * 3 + 3; ++row) {
31            for (int col = j / 3 * 3; col < j / 3 * 3 + 3; ++col) {
32                if ((row != i || col != j) && board[i][j] == board[row][col]) return false;
33            }
34        }
35        return true;
36    }
37}

```

38. 报数

报数序列是一个整数序列，按照其中的整数的顺序进行报数，得到下一个数。其前五项如下：

```
1.      1
2.      11
3.      21
4.      1211
5.      111221
```

1 被读作 "one 1" ("一个一")，即 11。

11 被读作 "two 1s" ("两个一")，即 21。

21 被读作 "one 2", "one 1" ("一个二", "一个一")，即 1211。

给定一个正整数 n ($1 \leq n \leq 30$)，输出报数序列的第 n 项。

注意：整数顺序将表示为一个字符串。

示例 1：

```
输入: 1
输出: "1"
```

示例 2：

```
输入: 4
输出: "1211"
```

这道计数和读法问题还是第一次遇到，看似挺复杂，其实仔细一看，算法很简单，就是对于前一个数，找出相同元素的个数，把这个数和该元素存到新的string里。代码如下：

```
1 class Solution {
2 public:
3     string countAndSay(int n) {
4         if (n <= 0) return "";
5         string res = "1";
6         while (--n) {
7             string cur = "";
8             for (int i = 0; i < res.size(); ++i) {
9                 int cnt = 1;
10                while (i + 1 < res.size() && res[i] == res[i + 1]) {
11                    ++cnt;
12                    ++i;
13                }
14                cur += to_string(cnt) + res[i];
15            }
16            res = cur;
17        }
18        return res;
19    }
20};
```

CPP

博主出于好奇打印出了前12个数字，发现一个很有意思的现象，不管打印到后面多少位，出现的数字只是由1,2和3组成，网上也有人发现了并分析了原因 (<http://www.cnblogs.com/TenosDoIt/p/3776356.html>)，前十二个数字如下：

```
1 | 1
2 | 1 1
3 | 2 1
4 | 1 2 1 1
5 | 1 1 1 2 2 1
6 | 3 1 2 2 1 1
7 | 1 3 1 1 2 2 2 1
8 | 1 1 1 3 2 1 3 2 1 1
9 | 3 1 1 3 1 2 1 1 1 3 1 2 2 1
10 | 1 3 2 1 1 3 1 1 1 2 3 1 1 3 1 1 2 2 1 1
11 | 1 1 1 3 1 2 2 1 1 3 3 1 1 2 1 3 2 1 1 3 2 1 2 2 2 1
12 | 3 1 1 3 1 1 2 2 2 1 2 3 2 1 1 2 1 1 1 3 1 2 2 1 1 3 1 2 1 1 3 2 1 1
```

CPP

39. 组合总和

给定一个无重复元素的数组 candidates 和一个目标数 target ，找出 candidates 中所有可以使数字和为 target 的组合。

candidates 中的数字可以无限制重复被选取。

说明：

- 所有数字（包括 target）都是正整数。
- 解集不能包含重复的组合。

示例 1：

输入： candidates = [2,3,6,7]， target = 7，
所求解集为：

```
[  
    [7],  
    [2,2,3]  
]
```

示例 2：

输入： candidates = [2,3,5]， target = 8，
所求解集为：

```
[  
    [2,2,2,2],  
    [2,3,3],  
    [3,5]  
]
```

像这种结果要求返回所有符合要求解的题十有八九都是要利用到递归，而且解题的思路都大同小异，相类似的题目有 Path Sum II 二叉树路径之和之二，Subsets II 子集合之二，Permutations 全排列，Permutations II 全排列之二，Combinations 组合项等等，如果仔细研究这些题目发现都是一个套路，都是需要另写一个递归函数，这里我们新加入三个变量，start记录当前的递归到的下标，out为一个解，res保存所有已经得到的解，每次调用新的递归函数时，此时的target要减去当前数组的的数，具体看代码如下：

```

1 class Solution {
2 public:
3     vector<vector<int>> combinationSum(vector<int> &candidates, int target) {
4         vector<vector<int>> res;
5         vector<int> out;
6         sort(candidates.begin(), candidates.end());
7         combinationSumDFS(candidates, target, 0, out, res);
8         return res;
9     }
10    void combinationSumDFS(vector<int> &candidates, int target, int start, vector<int>
11 &out, vector<vector<int>> &res) {
12        if (target < 0) return;
13        else if (target == 0) res.push_back(out);
14        else {
15            for (int i = start; i < candidates.size(); ++i) {
16                out.push_back(candidates[i]);
17                combinationSumDFS(candidates, target - candidates[i], i, out, res);
18                out.pop_back();
19            }
20        }
21    }
22 };

```

40. 组合总和 II

给定一个数组 candidates 和一个目标数 target，找出 candidates 中所有可以使数字和为 target 的组合。

candidates 中的每个数字在每个组合中只能使用一次。

说明：

- 所有数字（包括目标数）都是正整数。
- 解集不能包含重复的组合。

示例 1：

输入： candidates = [10,1,2,7,6,1,5]， target = 8，

所求解集为：

```
[
    [1, 7],
    [1, 2, 5],
    [2, 6],
    [1, 1, 6]
]
```

示例 2：

输入： candidates = [2,5,2,1,2]， target = 5，

所求解集为：

```
[
    [1,2,2],
    [5]
]
```

这道题跟之前那道 Combination Sum 组合之和 本质没有区别，只需要改动一点点即可，之前那道题给定数组中的数字可以重复使用，而这道题不能重复使用，只需要在之前的基础上修改两个地方即可，首先在递归的for循环里加上if (*i* > start && num[i] == num[i - 1]) continue; 这样可以防止res中出现重复项，然后就在递归调用combinationSum2DFS里面的参数换成*i*+1，这样就不会重复使用数组中的数字了，代码如下：

```

1 class Solution {
2 public:
3     vector<vector<int>> combinationSum2(vector<int> &num, int target) {
4         vector<vector<int>> res;
5         vector<int> out;
6         sort(num.begin(), num.end());
7         combinationSum2DFS(num, target, 0, out, res);
8         return res;
9     }
10    void combinationSum2DFS(vector<int> &num, int target, int start, vector<int> &out,
11    vector<vector<int>> &res) {
12        if (target < 0) return;
13        else if (target == 0) res.push_back(out);
14        else {
15            for (int i = start; i < num.size(); ++i) {
16                if (i > start && num[i] == num[i - 1]) continue;
17                out.push_back(num[i]);
18                combinationSum2DFS(num, target - num[i], i + 1, out, res);
19                out.pop_back();
20            }
21        }
22    }
};

```

41. 缺失的第一个正数

给定一个未排序的整数数组，找出其中没有出现的最小的正整数。

示例 1:

输入： [1,2,0]
输出： 3

示例 2:

输入： [3,4,-1,1]
输出： 2

示例 3:

输入： [7,8,9,11,12]
输出： 1

这道题让我们找缺失的首个正数，由于限定了O(n)的时间，所以一般的排序方法都不能用，最开始我没有看到还限制了空间复杂度，所以想到了用HashSet来解，这个思路很简单，第一遍遍历数组把所有的数都存入HashSet中，并且找出数组的最大值，下次循环从1开始递增找数字，哪个数字找不到就返回哪个数字，如果一直找到了最大的数字，则返回最大值+1，代码如下：

解法1:

```

1 class Solution {
2 public:
3     int firstMissingPositive(vector<int>& nums) {
4         int mx = 0;
5         unordered_set<int> s;
6         for (int num : nums) {
7             if (num <= 0) continue;
8             s.insert(num);
9             mx = max(mx, num);
10        }
11        for (int i = 1; i <= mx; ++i) {
12            if (!s.count(i)) return i;
13        }
14        return mx + 1;
15    }
16 };

```

但是上面的解法不是O(1)的空间复杂度，所以我们需要另想一种解法，既然不能建立新的数组，那么我们只能覆盖原有数组，我们的思路是把1放在数组第一个位置nums[0]，2放在第二个位置nums[1]，即需要把nums[i]放在nums[nums[i] - 1]上，那么我们遍历整个数组，如果nums[i] != i + 1，而nums[i]为整数且不大于n，另外nums[i]不等于nums[nums[i] - 1]的话，我们将两者位置调换，如果不满足上述条件直接跳过，最后我们再遍历一遍数组，如果对应位置上的数不正确则返回正确的数，代码如下：

解法2：

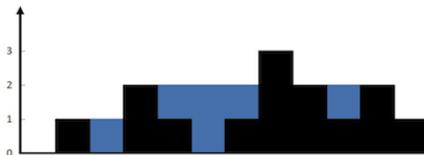
```

1 class Solution {
2 public:
3     int firstMissingPositive(vector<int>& nums) {
4         int n = nums.size();
5         for (int i = 0; i < n; ++i) {
6             while (nums[i] > 0 && nums[i] <= n && nums[nums[i] - 1] != nums[i]) {
7                 swap(nums[i], nums[nums[i] - 1]);
8             }
9         }
10        for (int i = 0; i < n; ++i) {
11            if (nums[i] != i + 1) return i + 1;
12        }
13        return n + 1;
14    }
15 };

```

42. 接雨水

给定n个非负整数表示每个宽度为1的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。



上面是由数组 [0,1,0,2,1,0,1,3,2,1,2,1] 表示的高度图，在这种情况下，可以接 6 个单位的雨水（蓝色部分表示雨水）。感谢 Marcos 贡献此图

示例：

输入： [0,1,0,2,1,0,1,3,2,1,2,1]
 输出： 6

这道收集雨水的题跟之前的那道 Largest Rectangle in Histogram 直方图中最大的矩形 有些类似，但是又不太一样，我们先来看一种方法，这种方法是基于动态规划Dynamic Programming的，我们维护一个一维的dp数组，这个DP算法需要遍历两遍数组，第一遍遍历dp[i]中存入i位置左边的最大值，然后开始第二遍遍历数组，第二次遍历时找右边最大值，然后和左边最大值比较取其中的较小值，然后跟当前值A[i]相比，如果大于当前值，则将差值存入结果，代码如下：

解法1：

```

1 class Solution {
2 public:
3     int trap(vector<int>& height) {
4         int res = 0, mx = 0, n = height.size();
5         vector<int> dp(n, 0);
6         for (int i = 0; i < n; ++i) {
7             dp[i] = mx;
8             mx = max(mx, height[i]);
9         }
10        mx = 0;
11        for (int i = n - 1; i >= 0; --i) {
12            dp[i] = min(dp[i], mx);
13            mx = max(mx, height[i]);
14            if (dp[i] > height[i]) res += dp[i] - height[i];
15        }
16        return res;
17    }
18 };

```

CPP

最后我们来看一种只需要遍历一次即可的解法，这个算法需要left和right两个指针分别指向数组的首尾位置，从两边向中间扫描，在当前两指针确定的范围内，先比较两头找出较小值，如果较小值是left指向的值，则从左向右扫描，如果较小值是right指向的值，则从右向左扫描，若遇到的值比当较小值小，则将差值存入结果，如遇到的值大，则重新确定新的窗口范围，以此类推直至left和right指针重合，具体参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int trap(vector<int>& height) {
4         int res = 0, l = 0, r = height.size() - 1;
5         while (l < r) {
6             int mn = min(height[l], height[r]);
7             if (mn == height[l]) {
8                 ++l;
9                 while (l < r && height[l] < mn) {
10                     res += mn - height[l++];
11                 }
12             } else {
13                 --r;
14                 while (l < r && height[r] < mn) {
15                     res += mn - height[r--];
16                 }
17             }
18         }
19         return res;
20     }
21 };

```

43. 字符串相乘

给定两个以字符串形式表示的非负整数 num1 和 num2，返回 num1 和 num2 的乘积，它们的乘积也表示为字符串形式。

示例 1:

输入: num1 = "2", num2 = "3"
输出: "6"

示例 2:

输入: num1 = "123", num2 = "456"
输出: "56088"

说明:

- num1 和 num2 的长度小于110。
- num1 和 num2 只包含数字 0-9。
- num1 和 num2 均不以零开头，除非是数字 0 本身。
- 不能使用任何标准库的大数类型（比如 BigInteger）或直接将输入转换为整数来处理。

这道题让我们求两个字符串数字的相乘，输入的两个数和返回的数都是以字符串格式储存的，这样做的原因可能是这样可以计算超大数相乘，可以不受int或long的数值范围的约束，那么我们该如何来计算乘法呢，我们小时候都学过多位数的乘法过程，都是每位相乘然后错位相加，那么这里就是用到这种方法，参见网友JustDoIt的博客，把错位相加后的结果保存到一个一维数组中，然后分别每位上算进位，最后每个数字都变成一位，然后要做的是去除掉首位0，最后把每位上的数字按顺序保存到结果中即可，代码如下：

```

1 class Solution {
2 public:
3     string multiply(string num1, string num2) {
4         string res;
5         int n1 = num1.size(), n2 = num2.size();
6         int k = n1 + n2 - 2, carry = 0;
7         vector<int> v(n1 + n2, 0);
8         for (int i = 0; i < n1; ++i) {
9             for (int j = 0; j < n2; ++j) {
10                 v[k - i - j] += (num1[i] - '0') * (num2[j] - '0');
11             }
12         }
13         for (int i = 0; i < n1 + n2; ++i) {
14             v[i] += carry;
15             carry = v[i] / 10;
16             v[i] %= 10;
17         }
18         int i = n1 + n2 - 1;
19         while (v[i] == 0) --i;
20         if (i < 0) return "0";
21         while (i >= 0) res.push_back(v[i--] + '0');
22         return res;
23     }
24 };

```

44. 通配符匹配

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）。

两个字符串完全匹配才算匹配成功。

说明：

- s 可能为空，且只包含从 a-z 的小写字母。
- p 可能为空，且只包含从 a-z 的小写字母，以及字符 ? 和 *。

示例 1：

输入：

s = "aa"

p = "a"

输出：false

解释："a" 无法匹配 "aa" 整个字符串。

示例 2：

输入：

s = "aa"

p = "*"

输出：true

解释：'*' 可以匹配任意字符串。

示例 3:

输入：

s = "cb"

p = "?a"

输出：false

解释：'?' 可以匹配 'c'，但第二个 'a' 无法匹配 'b'。

示例 4:

输入：

s = "adceb"

p = "*a*b"

输出：true

解释：第一个 '*' 可以匹配空字符串，第二个 '*' 可以匹配字符串 "dce"。

示例 5:

输入：

s = "acdcb"

p = "a*c?b"

输入：false

这道题通配符匹配问题还是小有难度的，这道里用了贪婪算法Greedy Algorithm来解，由于有特殊字符*和?，其中?能代替任何字符，*能代替任何字符串，那么我们需要定义几个额外的指针，其中scur和pcur分别指向当前遍历到的字符，再定义pstar指向p中最后一个*的位置，sstar指向此时对应的s的位置，具体算法如下：

- 定义scur, pcur, sstar, pstar
- 如果*scur存在
- 如果*scur等于*pcur或者*pcur为 '?', 则scur和pcur都自增1
- 如果*pcur为'*', 则pstar指向pcur位置, pcur自增1, 且sstar指向scur
- 如果pstar存在, 则pcur指向pstar的下一个位置, scur指向sstar自增1后的位置
- 如果pcur为'*', 则pcur自增1
- 若*pcur存在, 返回False, 若不存在, 返回True

解法1:

```

1 bool isMatch(char *s, char *p) {
2     char *scur = s, *pcur = p, *sstar = NULL, *pstar = NULL;
3     while (*scur) {
4         if (*scur == *pcur || *pcur == '?') {
5             ++scur;
6             ++pcur;
7         } else if (*pcur == '*') {
8             pstar = pcur++;
9             sstar = scur;
10        } else if (pstar) {
11            pcur = pstar + 1;
12            scur = ++sstar;
13        } else return false;
14    }
15    while (*pcur == '*') ++pcur;
16    return !*pcur;
17 }
```

这道题也能用动态规划Dynamic Programming来解，写法跟之前那道题Regular Expression Matching很像，但是还是不一样。外卡匹配和正则匹配最大的区别就是在星号的使用规则上，对于正则匹配来说，星号不能单独存在，前面必须要有一个字符，而星号存在的意义就是表明前面这个字符的个数可以是任意个，包括0个，那么就是说即使前面这个字符并没有在s中出现过也无所谓，只要后面的能匹配上就可以了。而外卡匹配就不是这样的，外卡匹配中的星号跟前面的字符没有半毛钱关系，如果前面的字符没有匹配上，那么直接返回false了，根本不用管星号。而星号存在的作用是可以表示任意的字符串，当然只是当匹配字符串缺少一些字符的时候起作用，当匹配字符串包含目标字符串没有的字符时，将无法成功匹配。

解法2：

```

1 class Solution {
2 public:
3     bool isMatch(string s, string p) {
4         int m = s.size(), n = p.size();
5         vector<vector<bool>> dp(m + 1, vector<bool>(n + 1, false));
6         dp[0][0] = true;
7         for (int i = 1; i <= n; ++i) {
8             if (p[i - 1] == '*') dp[0][i] = dp[0][i - 1];
9         }
10        for (int i = 1; i <= m; ++i) {
11            for (int j = 1; j <= n; ++j) {
12                if (p[j - 1] == '*') {
13                    dp[i][j] = dp[i - 1][j] || dp[i][j - 1];
14                } else {
15                    dp[i][j] = (s[i - 1] == p[j - 1] || p[j - 1] == '?') && dp[i - 1][j - 1];
16                }
17            }
18        }
19    }
20    return dp[m][n];
21 }
```

45. 跳跃游戏 II

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

你的目标是使用最少的跳跃次数到达数组的最后一个位置。

示例：

输入： [2,3,1,1,4]

输出： 2

解释： 跳到最后一个位置的最小跳跃数是 2。

从下标为 0 跳到下标为 1 的位置，跳 1 步，然后跳 3 步到达数组的最后一个位置。

说明：

假设你总是可以到达数组的最后一个位置。

这题是之前那道Jump Game 跳跃游戏 的延伸，那题是问能不能到达最后一个数字，而此题只让我们求到达最后一个位置的最少跳跃数，貌似是默认一定能到达最后位置的？此题的核心方法是利用贪婪算法Greedy的思想来解，想想为什么呢？为了较快的跳到末尾，我们想知道每一步能跳的范围，这里贪婪并不是要在能跳的范围中选跳力最远的那个位置，因为这样选下来不一定是最优解，这么一说感觉又有点不像贪婪算法了。我们这里贪的是一个能到达的最远范围，我们遍历当前跳跃能到的所有位置，然后根据该位置上的跳力来预测下一步能跳到的最远距离，跳出一个最远的范围，一旦当这个范围到达末尾时，当前所用的步数一定是最小步数。我们需要两个变量cur和pre分别来保存当前的能到达的最远位置和之前能到达的最远位置，只要cur未达到最后一个位置则循环继续，pre先赋值为cur的值，表示上一次循环后能到达的最远位置，如果当前位置i小于等于pre，说明还是在上一跳能到达的范围内，我们根据当前位置加跳力来更新cur，更新cur的方法是比较当前的cur和*i + A[i]*之中的较大值，如果题目中未说明是否能到达末尾，我们还可以判断此时pre和cur是否相等，如果相等说明cur没有更新，即无法到达末尾位置，返回-1，代码如下：

解法1：

```

1 class Solution {
2 public:
3     int jump(vector<int>& nums) {
4         int res = 0, n = nums.size(), i = 0, cur = 0;
5         while (cur < n - 1) {
6             ++res;
7             int pre = cur;
8             for (; i <= pre; ++i) {
9                 cur = max(cur, i + nums[i]);
10            }
11            if (pre == cur) return -1; // May not need this
12        }
13        return res;
14    }
15 };

```

CPP

还有一种写法，跟上面那解法略有不同，但是本质的思想还是一样的，关于此解法的详细分析可参见网友实验室小纸贴校外版的博客，这里cur是当前能到达的最远位置，last是上一步能到达的最远位置，我们遍历数组，首先用*i + nums[i]*更新cur，这个在上面解法中讲过了，然后判断如果当前位置到达了last，即上一步能到达的最远位置，说明需要再跳一次了，我们将last赋值为cur，并且步数res自增1，这里我们小优化一下，判断如果cur到达末尾了，直接break掉即可，代码如下：

解法2：

```

1 class Solution {
2 public:
3     int jump(vector<int>& nums) {
4         int res = 0, n = nums.size(), last = 0, cur = 0;
5         for (int i = 0; i < n - 1; ++i) {
6             cur = max(cur, i + nums[i]);
7             if (i == last) {
8                 last = cur;
9                 ++res;
10                if (cur >= n - 1) break;
11            }
12        }
13        return res;
14    }
15 };

```

46. 全排列

给定一个没有重复数字的序列，返回其所有可能的全排列。

示例：

输入： [1,2,3]

输出：

```
[
[1,2,3],
[1,3,2],
[2,1,3],
[2,3,1],
[3,1,2],
[3,2,1]
]
```

这道题是求全排列问题，给的输入数组没有重复项，这跟之前的那道 Combinations 组合项和类似，解法基本相同，但是不同点在于那道不同的数字顺序只算一种，是一道典型的组合题，而此题是求全排列问题，还是用递归DFS来求解。这里我们需要用到一个visited数组来标记某个数字是否访问过，然后在DFS递归函数从的循环应从头开始，而不是从level开始，这是和 Combinations 组合项不同的地方，其余思路大体相同，代码如下：

解法1

```

1 class Solution {
2 public:
3     vector<vector<int>> permute(vector<int> &num) {
4         vector<vector<int>> res;
5         vector<int> out;
6         vector<int> visited(num.size(), 0);
7         permuteDFS(num, 0, visited, out, res);
8         return res;
9     }
10    void permuteDFS(vector<int> &num, int level, vector<int> &visited, vector<int> &out,
11    vector<vector<int>> &res) {
12        if (level == num.size()) res.push_back(out);
13        else {
14            for (int i = 0; i < num.size(); ++i) {
15                if (visited[i] == 0) {
16                    visited[i] = 1;
17                    out.push_back(num[i]);
18                    permuteDFS(num, level + 1, visited, out, res);
19                    out.pop_back();
20                    visited[i] = 0;
21                }
22            }
23        }
24    }
25 };

```

还有一种递归的写法，更简单一些，这里是每次交换num里面的两个数字，经过递归可以生成所有的排列情况，代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<vector<int>> permute(vector<int> &num) {
4         vector<vector<int>> res;
5         permuteDFS(num, 0, res);
6         return res;
7     }
8     void permuteDFS(vector<int> &num, int start, vector<vector<int>> &res) {
9         if (start >= num.size()) res.push_back(num);
10        for (int i = start; i < num.size(); ++i) {
11            swap(num[start], num[i]);
12            permuteDFS(num, start + 1, res);
13            swap(num[start], num[i]);
14        }
15    }
16 };

```

最后再来看一种方法，这种方法是CareerCup书上的方法，也挺不错的，这道题的思想是这样的：

当n=1时，数组中只有一个数a1，其全排列只有一种，即为a1

当n=2时，数组中此时有a1a2，其全排列有两种，a1a2和a2a1，那么此时我们考虑和上面那种情况的关系，我们发现，其实就是在a1的前后两个位置分别加入了a2

当n=3时，数组中有a1a2a3，此时全排列有六种，分别为a1a2a3, a1a3a2, a2a1a3, a2a3a1, a3a1a2, 和 a3a2a1。那么根据上面的结论，实际上是在a1a2和a2a1的基础上在不同的位置上加入a3而得到的。

_a1_a2_: a3a1a2, a1a3a2, a1a2a3

_a2_a1_: a3a2a1, a2a3a1, a2a1a3

解法3

```
1 class Solution {
2 public:
3     vector<vector<int>> permute(vector<int> &num) {
4         if (num.empty()) return vector<vector<int>>(1, vector<int>());
5         vector<vector<int>> res;
6         int first = num[0];
7         num.erase(num.begin());
8         vector<vector<int>> words = permute(num);
9         for (auto &a : words) {
10             for (int i = 0; i <= a.size(); ++i) {
11                 a.insert(a.begin() + i, first);
12                 res.push_back(a);
13                 a.erase(a.begin() + i);
14             }
15         }
16         return res;
17     }
18 }
```

CPP

47. 全排列 II

给定一个可包含重复数字的序列，返回所有不重复的全排列。

示例：

输入： [1,1,2]

输出：

```
[  
    [1,1,2],  
    [1,2,1],  
    [2,1,1]  
]
```

这道题是之前那道 Permutations 全排列的延伸，由于输入数组有可能出现重复数字，如果按照之前的算法运算，会有重复排列产生，我们要避免重复的产生，在递归函数中要判断前面一个数和当前的数是否相等，如果相等，前面的数必须已经使用了，即对应的visited中的值为1，当前的数字才能使用，否则需要跳过，这样就不会产生重复排列了，代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<vector<int>> permuteUnique(vector<int> &num) {
4         vector<vector<int>> res;
5         vector<int> out;
6         vector<int> visited(num.size(), 0);
7         sort(num.begin(), num.end());
8         permuteUniqueDFS(num, 0, visited, out, res);
9         return res;
10    }
11    void permuteUniqueDFS(vector<int> &num, int level, vector<int> &visited, vector<int>
12 &out, vector<vector<int>> &res) {
13        if (level >= num.size()) res.push_back(out);
14        else {
15            for (int i = 0; i < num.size(); ++i) {
16                if (visited[i] == 0) {
17                    if (i > 0 && num[i] == num[i - 1] && visited[i - 1] == 0) continue;
18                    visited[i] = 1;
19                    out.push_back(num[i]);
20                    permuteUniqueDFS(num, level + 1, visited, out, res);
21                    out.pop_back();
22                    visited[i] = 0;
23                }
24            }
25        }
26    }
27 };

```

还有一种比较简便的方法，在Permutations的基础上稍加修改，我们用set来保存结果，利用其不会有重复项的特点，然后我们再递归函数中的swap的地方，判断如果i和start不相同，但是nums[i]和nums[start]相同的情况下跳过，继续下一个循环，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<vector<int>> permuteUnique(vector<int>& nums) {
4         set<vector<int>> res;
5         permute(nums, 0, res);
6         return vector<vector<int>> (res.begin(), res.end());
7     }
8     void permute(vector<int> &nums, int start, set<vector<int>> &res) {
9         if (start >= nums.size()) res.insert(nums);
10        for (int i = start; i < nums.size(); ++i) {
11            if (i != start && nums[i] == nums[start]) continue;
12            swap(nums[i], nums[start]);
13            permute(nums, start + 1, res);
14            swap(nums[i], nums[start]);
15        }
16    }
17 };

```

48. 旋转图像

给定一个 $n \times n$ 的二维矩阵表示一个图像。

将图像顺时针旋转 90 度。

说明：

你必须在原地旋转图像，这意味着你需要直接修改输入的二维矩阵。请不要使用另一个矩阵来旋转图像。

示例 1：

```
给定 matrix =
[
    [1,2,3],
    [4,5,6],
    [7,8,9]
],
```

原地旋转输入矩阵，使其变为：

```
[

    [7,4,1],
    [8,5,2],
    [9,6,3]
]
```

示例 2：

```
给定 matrix =
[
    [ 5, 1, 9,11],
    [ 2, 4, 8,10],
    [13, 3, 6, 7],
    [15,14,12,16]
],
```

原地旋转输入矩阵，使其变为：

```
[

    [15,13, 2, 5],
    [14, 3, 4, 1],
    [12, 6, 8, 9],
    [16, 7,10,11]
]
```

在计算机图像处理里，旋转图片是很常见的，由于图片的本质是二维数组，所以也就变成了对数组的操作处理，翻转的本质就是某个位置上数移动到另一个位置上，比如用一个简单的例子来分析：

1 2 3 7 4 1

4 5 6 → 8 5 2

7 8 9 9 6 3

对于90度的翻转有很多方法，一步或多步都可以解，我们先来看一种直接的方法，对于当前位置，计算旋转后的新位置，然后再计算下一个新位置，第四个位置又变成当前位置了，所以这个方法每次循环换四个数字，如下所示：

1 2 3 7 2 1 7 4 1

4 5 6 → 4 5 6 → 8 5 2

7 8 9 9 8 3 9 6 3

解法1：

```

1 class Solution {
2 public:
3     void rotate(vector<vector<int> > &matrix) {
4         int n = matrix.size();
5         for (int i = 0; i < n / 2; ++i) {
6             for (int j = i; j < n - 1 - i; ++j) {
7                 int tmp = matrix[i][j];
8                 matrix[i][j] = matrix[n - 1 - j][i];
9                 matrix[n - 1 - j][i] = matrix[n - 1 - i][n - 1 - j];
10                matrix[n - 1 - i][n - 1 - j] = matrix[j][n - 1 - i];
11                matrix[j][n - 1 - i] = tmp;
12            }
13        }
14    }
15 };

```

CPP

还有一种解法，首先以从对角线为轴翻转，然后再以x轴中线上下翻转即可得到结果，如下图所示(其中蓝色数字表示翻转轴)：

1 2 3	9 6 3	7 4 1
4 5 6	→ 8 5 2	→ 8 5 2 +
7 8 9	7 4 1	9 6 3

解法2：

```

1 class Solution {
2 public:
3     void rotate(vector<vector<int> > &matrix) {
4         int n = matrix.size();
5         for (int i = 0; i < n - 1; ++i) {
6             for (int j = 0; j < n - i; ++j) {
7                 swap(matrix[i][j], matrix[n - 1 - j][n - 1 - i]);
8             }
9         }
10        for (int i = 0; i < n / 2; ++i) {
11            for (int j = 0; j < n; ++j) {
12                swap(matrix[i][j], matrix[n - 1 - i][j]);
13            }
14        }
15    }
16 };

```

CPP

最后再来看一种方法，这种方法首先对原数组取其转置矩阵，然后把每行的数字翻转可得到结果，如下所示(其中蓝色数字表示翻转轴)：

1 2 3	1 4 7	7 4 1
4 5 6	→ 2 5 8	→ 8 5 2 +
7 8 9	3 6 9	9 6 3

解法3：

```

1 class Solution {
2 public:
3     void rotate(vector<vector<int> > &matrix) {
4         int n = matrix.size();
5         for (int i = 0; i < n; ++i) {
6             for (int j = i + 1; j < n; ++j) {
7                 swap(matrix[i][j], matrix[j][i]);
8             }
9             reverse(matrix[i].begin(), matrix[i].end());
10        }
11    }
12 };

```

49. 字母异位词分组

给定一个字符串数组，将字母异位词组合在一起。字母异位词指字母相同，但排列不同的字符串。

示例：

输入： ["eat", "tea", "tan", "ate", "nat", "bat"],

输出：

```
[
  ["ate","eat","tea"],
  ["nat","tan"],
  ["bat"]
]
```

说明：

- 所有输入均为小写字母。
- 不考虑答案输出的顺序。

这道题让我们群组给定字符串集中所有的错位词，所谓的错位词就是两个字符串中字母出现的次数都一样，只是位置不同，比如abc, bac, cba等它们就互为错位词，那么我们如何判断两者是否是错位词呢，我们发现如果把错位词的字符顺序重新排列，那么会得到相同的结果，所以重新排序是判断是否互为错位词的方法，由于错位词重新排序后都会得到相同的字符串，我们以此作为key，将所有错位词都保存到字符串数组中，建立key和字符串数组之间的映射，最后再存入结果res中即可，擦巾代码如下：

解法1

```

1 class Solution {
2 public:
3     vector<vector<string>> groupAnagrams(vector<string>& strs) {
4         vector<vector<string>> res;
5         unordered_map<string, vector<string>> m;
6         for (string str : strs) {
7             string t = str;
8             sort(t.begin(), t.end());
9             m[t].push_back(str);
10        }
11        for (auto a : m) {
12            res.push_back(a.second);
13        }
14        return res;
15    }
16 };

```

下面这种解法没有用到排序，提高了运算效率，我们用一个大小为26的int数组来统计每个单词中字符出现的次数，然后将int数组转为一个唯一的字符串，跟字符串数组进行映射，这样我们就不用给字符串排序了，代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<vector<string>> groupAnagrams(vector<string>& strs) {
4         vector<vector<string>> res;
5         unordered_map<string, vector<string>> m;
6         for (string str : strs) {
7             vector<int> cnt(26, 0);
8             string t = "";
9             for (char c : str) ++cnt[c - 'a'];
10            for (int d : cnt) t += to_string(d) + "/";
11            m[t].push_back(str);
12        }
13        for (auto a : m) {
14            res.push_back(a.second);
15        }
16        return res;
17    }
18 };

```

50. Pow(x, n)

实现 pow(x, n) ，即计算 x 的 n 次幂函数。

示例 1：

输入： 2.00000, 10
输出： 1024.00000

示例 2：

输入： 2.10000, 3
输出： 9.26100

示例 3:

输入: 2.00000, -2
输出: 0.25000

解释: $2^{-2} = 1/2^2 = 1/4 = 0.25$

说明

$-100.0 < x < 100.0$

这道题让我们求 x 的 n 次方，如果我们只是简单的用个for循环让 x 乘以自己 n 次的话，未免也把LeetCode上的题想的太简单了，一句话形容图样图森破啊。OJ因超时无法通过，所以我们需要优化我们的算法，使其在更有效的算出结果来。我们可以用递归来折半计算，每次把 n 缩小一半，这样 n 最终会缩小到0，任何数的0次方都为1，这时候我们再往回乘，如果此时 n 是偶数，直接把上次递归得到的值算个平方返回即可，如果是奇数，则还需要乘上个 x 的值。还有一点需要引起我们的注意的是 n 有可能为负数，对于 n 是负数的情况，我们可以先用其绝对值计算出一个结果再取其倒数即可，代码如下：

解法1:

```
1 class Solution {
2 public:
3     double myPow(double x, int n) {
4         if (n < 0) return 1 / power(x, -n);
5         return power(x, n);
6     }
7     double power(double x, int n) {
8         if (n == 0) return 1;
9         double half = power(x, n / 2);
10        if (n % 2 == 0) return half * half;
11        return x * half * half;
12    }
13};
```

CPP

还有一种写法可以只用一个函数即可，在每次递归中处理 n 的正负，然后做相应的变换即可，代码如下：

解法2:

```
1 class Solution {
2 public:
3     double myPow(double x, int n) {
4         if (n == 0) return 1;
5         double half = myPow(x, n / 2);
6         if (n % 2 == 0) return half * half;
7         else if (n > 0) return half * half * x;
8         else return half * half / x;
9     }
10};
```

CPP

这道题还有迭代的解法，我们让 i 初始化为 n ，然后看 i 是否是2的倍数，是的话 x 乘以自己，否则 res 乘以 x ， i 每次循环缩小一半，直到为0停止循环。最后看 n 的正负，如果为负，返回其倒数，参见代码如下：

解法3:

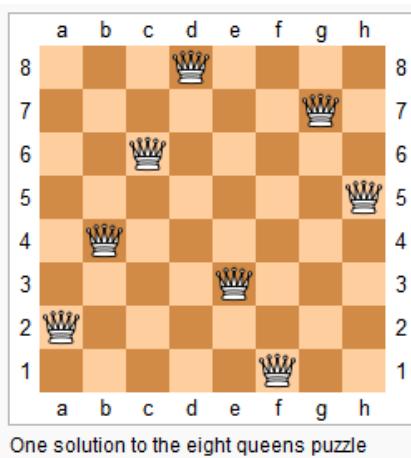
```

1 class Solution {
2 public:
3     double myPow(double x, int n) {
4         double res = 1.0;
5         for (int i = n; i != 0; i /= 2) {
6             if (i % 2 != 0) res *= x;
7             x *= x;
8         }
9         return n < 0 ? 1 / res : res;
10    }
11 };

```

51. N皇后

n 皇后问题研究的是如何将 n 个皇后放置在 n×n 的棋盘上，并且使皇后彼此之间不能相互攻击



One solution to the eight queens puzzle

上图为 8 皇后问题的一种解法。

给定一个整数 n，返回所有不同的 n 皇后问题的解决方案。

每一种解法包含一个明确的 n 皇后问题的棋子放置方案，该方案中 'Q' 和 '!' 分别代表了皇后和空位。

经典的N皇后问题，基本所有的算法书中都会包含的问题，经典解法为回溯递归，一层一层的向下扫描，需要用到一个pos数组，其中pos[i]表示第i行皇后的位置，初始化为-1，然后从第0开始递归，每一行都一次遍历各列，判断如果在该位置放置皇后会不会有冲突，以此类推，当到最后一行的皇后放好后，一种解法就生成了，将其存入结果res中，然后再还会继续完成搜索所有的情况，代码如下：

```

1 class Solution {
2 public:
3     vector<vector<string>> solveNQueens(int n) {
4         vector<vector<string>> res;
5         vector<int> pos(n, -1);
6         solveNQueensDFS(pos, 0, res);
7         return res;
8     }
9     void solveNQueensDFS(vector<int> &pos, int row, vector<vector<string>> &res) {
10        int n = pos.size();
11        if (row == n) {
12            vector<string> out(n, string(n, '.'));
13            for (int i = 0; i < n; ++i) {
14                out[i][pos[i]] = 'Q';
15            }
16            res.push_back(out);
17        } else {
18            for (int col = 0; col < n; ++col) {
19                if (isValid(pos, row, col)) {
20                    pos[row] = col;
21                    solveNQueensDFS(pos, row + 1, res);
22                    pos[row] = -1;
23                }
24            }
25        }
26    }
27    bool isValid(vector<int> &pos, int row, int col) {
28        for (int i = 0; i < row; ++i) {
29            if (col == pos[i] || abs(row - i) == abs(col - pos[i])) {
30                return false;
31            }
32        }
33        return true;
34    }
35}

```

52. N皇后 II

n 皇后问题研究的是如何将 n 个皇后放置在 $n \times n$ 的棋盘上，并且使皇后彼此之间不能相互攻击。



上图为 8 皇后问题的一种解法。

给定一个整数 n，返回 n 皇后不同的解决方案的数量。

这道题是之前那道 N-Queens N皇后问题的延伸，说是延伸其实我觉得两者顺序应该颠倒一样，上一道题比这道题还要稍稍复杂一些，两者本质上没有啥区别，都是要用回溯法Backtracking来解，如果理解了之前那道题的思路，此题只要做很小的改动即可，不再需要求出具体的皇后的摆法，只需要每次生成一种解法时，计数器加一即可，代码如下：

```

1 class Solution {
2 public:
3     int totalNQueens(int n) {
4         int res = 0;
5         vector<int> pos(n, -1);
6         totalNQueensDFS(pos, 0, res);
7         return res;
8     }
9     void totalNQueensDFS(vector<int> &pos, int row, int &res) {
10        int n = pos.size();
11        if (row == n) ++res;
12        else {
13            for (int col = 0; col < n; ++col) {
14                if (isValid(pos, row, col)) {
15                    pos[row] = col;
16                    totalNQueensDFS(pos, row + 1, res);
17                    pos[row] = -1;
18                }
19            }
20        }
21    }
22    bool isValid(vector<int> &pos, int row, int col) {
23        for (int i = 0; i < row; ++i) {
24            if (col == pos[i] || abs(row - i) == abs(col - pos[i])) {
25                return false;
26            }
27        }
28        return true;
29    }
30};

```

53. 最大子序和

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。

示例:

输入: [-2,1,-3,4,-1,2,1,-5,4],
输出: 6
解释: 连续子数组 [4,-1,2,1] 的和最大，为 6。

进阶:

如果你已经实现复杂度为 O(n) 的解法，尝试使用更为精妙的分治法求解。

这道题让我们求最大子数组之和，并且要我们用两种方法来解，分别是O(n)的解法，还有用分治法Divide and Conquer Approach，这个解法的时间复杂度是O(nlgn)，那我们就先来看O(n)的解法，定义两个变量res和curSum，其中res保存最终要返回的结果，即最大的子数组之和，curSum初始值为0，每遍历一个数字num，比较curSum + num和num中的较大值存入curSum，然后再把res和curSum中的较大值存入res，以此类推直到遍历完整个数组，可得到最大子数组的值存在res中，代码如下：

解法1

```

1 class Solution {
2 public:
3     int maxSubArray(vector<int>& nums) {
4         int res = INT_MIN, curSum = 0;
5         for (int num : nums) {
6             curSum = max(curSum + num, num);
7             res = max(res, curSum);
8         }
9         return res;
10    }
11 };

```

题目还要求我们用分治法Divide and Conquer Approach来解，这个分治法的思想就类似于二分搜索法，我们需要把数组一分为二，分别找出左边和右边的最大子数组之和，然后还要从中间开始向左右分别扫描，求出的最大值分别和左右两边得出的最大值相比较取最大的那个，代码如下：

解法2：

```

1 class Solution {
2 public:
3     int maxSubArray(vector<int>& nums) {
4         if (nums.empty()) return 0;
5         return helper(nums, 0, (int)nums.size() - 1);
6     }
7     int helper(vector<int>& nums, int left, int right) {
8         if (left >= right) return nums[left];
9         int mid = left + (right - left) / 2;
10        int lmax = helper(nums, left, mid - 1);
11        int rmax = helper(nums, mid + 1, right);
12        int mmax = nums[mid], t = mmax;
13        for (int i = mid - 1; i >= left; --i) {
14            t += nums[i];
15            mmax = max(mmax, t);
16        }
17        t = mmax;
18        for (int i = mid + 1; i <= right; ++i) {
19            t += nums[i];
20            mmax = max(mmax, t);
21        }
22        return max(mmax, max(lmax, rmax));
23    }
24 };

```

54. 螺旋矩阵

给定一个包含 $m \times n$ 个元素的矩阵（ m 行, n 列），请按照顺时针螺旋顺序，返回矩阵中的所有元素。

示例 1：

输入：

```
[
  [ 1, 2, 3 ],
  [ 4, 5, 6 ],
  [ 7, 8, 9 ]
]
```

输出： [1,2,3,6,9,8,7,4,5]

这道题让我们将一个矩阵按照螺旋顺序打印出来，我们只能一条边一条边的打印，首先我们要从给定的 $m \times n$ 的矩阵中算出按螺旋顺序有几个环，注意最终间的环可以是一个数字，也可以是一行或者一列。环数的计算公式是 $\min(m, n) / 2$ ，知道了环数，我们可以对每个环的边按顺序打印，比如对于题目中给的那个例子，个边生成的顺序是(用颜色标记了数字) Red → Green → Blue → Yellow → Black

1 2 3

4 5 6

7 8 9

我们定义p, q为当前环的高度和宽度，当p或者q为1时，表示最后一个环只有一行或者一列，可以跳出循环。此题的难点在于下标的转换，如何正确的转换下标是解此题的关键，我们可以对照着上面的 3×3 的例子来完成下标的填写，代码如下：

```

1 class Solution {
2 public:
3     vector<int> spiralOrder(vector<vector<int> > &matrix) {
4         vector<int> res;
5         if (matrix.empty() || matrix[0].empty()) return res;
6         int m = matrix.size(), n = matrix[0].size();
7         int c = m > n ? (n + 1) / 2 : (m + 1) / 2;
8         int p = m, q = n;
9         for (int i = 0; i < c; ++i, p -= 2, q -= 2) {
10            for (int col = i; col < i + q; ++col)
11                res.push_back(matrix[i][col]);
12            for (int row = i + 1; row < i + p; ++row)
13                res.push_back(matrix[row][i + q - 1]);
14            if (p == 1 || q == 1) break;
15            for (int col = i + q - 2; col >= i; --col)
16                res.push_back(matrix[i + p - 1][col]);
17            for (int row = i + p - 2; row > i; --row)
18                res.push_back(matrix[row][i]);
19        }
20        return res;
21    }
22};

```

CPP

55. 跳跃游戏

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个位置。

示例 1:

输入: [2,3,1,1,4]

输出: true

解释: 从位置 0 到 1 跳 1 步，然后跳 3 步到达最后一个位置。

这道题说的是有一个非负整数的数组，每个数字表示在当前位置的基础上最多可以走的步数，求判断能不能到达最后一个位置，开始我以为是必须刚好到达最后一个位置，超过了不算，其实是理解题意有误，因为每个位置上的数字表示的是最多可以走的步数而不是像玩大富翁一样摇骰子摇出几一定要走几步。那么我们可以用动态规划Dynamic Programming来解，我们维护一个一位数组dp，其中dp[i]表示达到i位置时剩余的步数，那么难点就是推导状态转移方程啦。我们想啊，到达当前位置的剩余步数跟

什么有关呢，其实是跟上一个位置的剩余步数和上一个位置的跳力有关，这里的跳力就是原数组中每个位置的数字，因为其代表了以当前位置为起点能到达的最远位置。所以当前位置的剩余步数（dp值）和当前位置的跳力中的较大那个数决定了当前能到的最远距离，而下一个位置的剩余步数（dp值）就等于当前的这个较大值减去1，因为需要花一个跳力到达下一个位置，所以我们就有状态转移方程了： $dp[i] = \max(dp[i - 1], nums[i - 1]) - 1$ ，如果当某一个时刻dp数组的值为负了，说明无法抵达当前位置，则直接返回false，最后我们判断dp数组最后一位是否为非负数即可知道是否能抵达该位置，代码如下：

解法1：

```
1 class Solution {
2 public:
3     bool canJump(vector<int>& nums) {
4         vector<int> dp(nums.size(), 0);
5         for (int i = 1; i < nums.size(); ++i) {
6             dp[i] = max(dp[i - 1], nums[i - 1]) - 1;
7             if (dp[i] < 0) return false;
8         }
9         return dp.back() >= 0;
10    }
11};
```

CPP

其实这题最好的解法不是DP，而是贪婪算法Greedy Algorithm，因为我们并不是很关心每一个位置上的剩余步数，我们只希望知道能否到达末尾，也就是说我们只对最远能到达的位置感兴趣，所以我们维护一个变量reach，表示最远能到达的位置，初始化为0。遍历数组中每一个数字，如果当前坐标大于reach或者reach已经抵达最后一个位置则跳出循环，否则就更新reach的值为其和*i + nums[i]*中的较大值，其中*i + nums[i]*表示当前位置能到达的最大位置，代码如下：

解法2：

```
1 class Solution {
2 public:
3     bool canJump(vector<int>& nums) {
4         int n = nums.size(), reach = 0;
5         for (int i = 0; i < n; ++i) {
6             if (i > reach || reach >= n - 1) break;
7             reach = max(reach, i + nums[i]);
8         }
9         return reach >= n - 1;
10    }
11};
```

CPP

56. 合并区间

给出一个区间的集合，请合并所有重叠的区间。

示例 1：

输入： [[1,3],[2,6],[8,10],[15,18]]
 输出： [[1,6],[8,10],[15,18]]
 解释： 区间 [1,3] 和 [2,6] 重叠，将它们合并为 [1,6].

这道和之前那道 Insert Interval 插入区间很类似，这次题目要求我们合并区间，之前那题明确了输入区间集是有序的，而这题没有，所以我们首先要做的就是给区间集排序，由于我们要排序的是个结构体，所以我们要定义自己的comparator，才能用sort来排序，我们以start的值从小到大来排序，排完序我们就可以开始合并了，首先把第一个区间存入结果中，然后从第二个开

始遍历区间集，如果结果中最后一个区间和遍历的当前区间无重叠，直接将当前区间存入结果中，如果有重叠，将结果中最后一个区间的end值更新为结果中最后一个区间的end和当前end值之中的较大值，然后继续遍历区间集，以此类推可以得到最终结果，代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<Interval> merge(vector<Interval>& intervals) {
4         if (intervals.empty()) return {};
5         sort(intervals.begin(), intervals.end(), [] (Interval &a, Interval &b) {return
6             a.start < b.start;});
7         vector<Interval> res{intervals[0]};
8         for (int i = 1; i < intervals.size(); ++i) {
9             if (res.back().end < intervals[i].start) {
10                 res.push_back(intervals[i]);
11             } else {
12                 res.back().end = max(res.back().end, intervals[i].end);
13             }
14         }
15         return res;
16     }
17 };

```

CPP

下面这种解法将起始位置和结束位置分别存到了两个不同的数组starts和ends中，然后分别进行排序，之后用两个指针i和j，初始化时分别指向starts和ends数组的首位置，然后如果i指向starts数组中的最后一个位置，或者当starts数组上i+1位置上的数字大于ends数组的i位置上的数时，此时说明区间已经不连续了，我们来看题目中的例子，排序后的starts和ends为：

starts: 1 2 8 15

ends: 3 6 10 18

红色为i的位置，蓝色为j的位置，那么此时starts[i+1]为8，ends[i]为6，8大于6，所以此时不连续了，将区间[starts[j], ends[i]]，即 [1, 6] 加入结果res中，然后j赋值为i+1继续循环，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<Interval> merge(vector<Interval>& intervals) {
4         int n = intervals.size();
5         vector<Interval> res;
6         vector<int> starts, ends;
7         for (int i = 0; i < n; ++i) {
8             starts.push_back(intervals[i].start);
9             ends.push_back(intervals[i].end);
10        }
11        sort(starts.begin(), starts.end());
12        sort(ends.begin(), ends.end());
13        for (int i = 0, j = 0; i < n; ++i) {
14            if (i == n - 1 || starts[i + 1] > ends[i]) {
15                res.push_back(Interval(starts[j], ends[i]));
16                j = i + 1;
17            }
18        }
19        return res;
20    }
21 };

```

这道题还有另一种解法，这个解法直接调用了之前那道题 Insert Interval 插入区间的函数，由于插入的过程中也有合并的操作，所以我们可以建立一个空的集合，然后把区间集的每一个区间当做一个新的区间插入结果中，也可以得到合并后的结果，那道题中的四种解法都可以在这里使用，但是没必要都列出来，这里只选了那道题中的解法二放到这里，代码如下：

解法3：

```

1 class Solution {
2 public:
3     vector<Interval> merge(vector<Interval>& intervals) {
4         vector<Interval> res;
5         for (int i = 0; i < intervals.size(); ++i) {
6             res = insert(res, intervals[i]);
7         }
8         return res;
9     }
10    vector<Interval> insert(vector<Interval>& intervals, Interval newInterval) {
11        vector<Interval> res;
12        int n = intervals.size(), cur = 0;
13        for (int i = 0; i < n; ++i) {
14            if (intervals[i].end < newInterval.start) {
15                res.push_back(intervals[i]);
16                ++cur;
17            } else if (intervals[i].start > newInterval.end) {
18                res.push_back(intervals[i]);
19            } else {
20                newInterval.start = min(newInterval.start, intervals[i].start);
21                newInterval.end = max(newInterval.end, intervals[i].end);
22            }
23        }
24        res.insert(res.begin() + cur, newInterval);
25        return res;
26    }
27 };

```

57. 插入区间

给出一个无重叠的，按照区间起始端点排序的区间列表。

在列表中插入一个新的区间，你需要确保列表中的区间仍然有序且不重叠（如果有必要的话，可以合并区间）。

示例 1：

输入： intervals = [[1,3],[6,9]], newInterval = [2,5]

输出： [[1,5],[6,9]]

这道题让我们在一系列非重叠的区间中插入一个新的区间，可能还需要和原有的区间合并，那么我们需要对给区间集一个一个的遍历比较，那么会有两种情况，重叠或是不重叠，不重叠的情况最好，直接将新区间插入到对应的位置即可，重叠的情况比较复杂，有时候会有多个重叠，我们需要更新新区域的范围以便包含所有重叠，之后将新区间加入结果res，最后将后面的区间再加入结果res即可。具体思路是，我们用一个变量cur来遍历区间，如果当前cur区间的结束位置小于要插入的区间的起始位置的话，说明没有重叠，则将cur区间加入结果res中，然后cur自增1。直到有cur越界或有重叠while循环退出，然后再用一个while循环处理所有重叠的区间，每次用取两个区间起始位置的较小值，和结束位置的较大值来更新要插入的区间，然后cur自增1。直到cur越界或者没有重叠时while循环退出。之后将更新好的新区间加入结果res，然后将cur之后的区间再加入结果res中即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<Interval> insert(vector<Interval>& intervals, Interval newInterval) {
4         vector<Interval> res;
5         int n = intervals.size(), cur = 0;
6         while (cur < n && intervals[cur].end < newInterval.start) {
7             res.push_back(intervals[cur++]);
8         }
9         while (cur < n && intervals[cur].start <= newInterval.end) {
10            newInterval.start = min(newInterval.start, intervals[cur].start);
11            newInterval.end = max(newInterval.end, intervals[cur].end);
12            ++cur;
13        }
14        res.push_back(newInterval);
15        while (cur < n) {
16            res.push_back(intervals[cur++]);
17        }
18        return res;
19    }
20 };

```

下面这种方法的思路跟上面的解法很像，只不过没有用while循环，而是使用的是for循环，但是思路上没有太大的区别，变量cur还是用来记录新区间该插入的位置，稍有不同的地方在于在for循环中已经将新区间后面不重叠的区间也加进去了，for循环结束后就只需要插入新区间即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<Interval> insert(vector<Interval>& intervals, Interval newInterval) {
4         vector<Interval> res;
5         int n = intervals.size(), cur = 0;
6         for (int i = 0; i < n; ++i) {
7             if (intervals[i].end < newInterval.start) {
8                 res.push_back(intervals[i]);
9                 ++cur;
10            } else if (intervals[i].start > newInterval.end) {
11                res.push_back(intervals[i]);
12            } else {
13                newInterval.start = min(newInterval.start, intervals[i].start);
14                newInterval.end = max(newInterval.end, intervals[i].end);
15            }
16        }
17        res.insert(res.begin() + cur, newInterval);
18        return res;
19    }
20 };

```

下面这种解法就是把上面解法的for循环改为了while循环，其他的都没有变，代码如下：

解法3：

```

1 class Solution {
2 public:
3     vector<Interval> insert(vector<Interval>& intervals, Interval newInterval) {
4         vector<Interval> res;
5         int n = intervals.size(), cur = 0, i = 0;
6         while (i < n) {
7             if (intervals[i].end < newInterval.start) {
8                 res.push_back(intervals[i]);
9                 ++cur;
10            } else if (intervals[i].start > newInterval.end) {
11                res.push_back(intervals[i]);
12            } else {
13                newInterval.start = min(newInterval.start, intervals[i].start);
14                newInterval.end = max(newInterval.end, intervals[i].end);
15            }
16            ++i;
17        }
18        res.insert(res.begin() + cur, newInterval);
19        return res;
20    }
21 };

```

如果学过Design Pattern的，对Iterator Pattern比较熟悉的也可应用Iterator来求解，本质还是一样的，只是写法略有不同，代码如下：

解法4：

```

1 class Solution {
2 public:
3     vector<Interval> insert(vector<Interval>& intervals, Interval newInterval) {
4         vector<Interval> res;
5         vector<Interval>::iterator it = intervals.begin();
6         int cur = 0;
7         while (it != intervals.end()) {
8             if (it->end < newInterval.start) {
9                 res.push_back(*it);
10                ++cur;
11            } else if (it->start > newInterval.end) {
12                res.push_back(*it);
13            } else {
14                newInterval.start = min(newInterval.start, it->start);
15                newInterval.end = max(newInterval.end, it->end);
16            }
17            ++it;
18        }
19        res.insert(res.begin() + cur, newInterval);
20        return res;
21    }
22 }

```

58. 最后一个单词的长度

给定一个仅包含大小写字母和空格' '的字符串，返回其最后一个单词的长度。

如果不存在最后一个单词，请返回0。

说明：一个单词是指由字母组成，但不包含任何空格的字符串。

示例：

输入："Hello World"

输出：5

这道题难度不是很大。先对输入字符串做预处理，去掉开头和结尾的空格，然后用一个计数器来累计非空格的字符串的长度，遇到空格则将计数器清零。代码如下：

解法1：

```

1 class Solution {
2 public:
3     int lengthOfLastWord(const char *s) {
4         int len = strlen(s);
5         int left = 0;
6         int right = len - 1;
7         int count = 0;
8         while (s[left] == ' ') ++left;
9         while (s[right] == ' ') --right;
10        for (int i = left; i <= right; ++i) {
11            if (s[i] == ' ') count = 0;
12            else ++count;
13        }
14        return count;
15    }
16 };

```

昨晚睡觉前又想到了一种解法，其实不用上面那么复杂的，我们关心的主要是非空格的字符，那么我们实际上在遍历字符串的时候，如果遇到非空格的字符，我们只需要判断其前面一个位置的字符是否为空格，如果是的话，那么当前肯定是一个新词的开始，将计数器重置为1，如果不是的话，说明正在统计一个词的长度，计数器自增1即可。但是需要注意的是，当i=0的时候，无法访问前一个字符，所以这种情况要特别判断一下，归为计数器自增1那类。参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int lengthOfLastWord(string s) {
4         int res = 0;
5         for (int i = 0; i < s.size(); ++i) {
6             if (s[i] != ' ') {
7                 if (i != 0 && s[i - 1] == ' ') res = 1;
8                 else ++res;
9             }
10        }
11        return res;
12    }
13 };

```

下面这种方法是第一种解法的优化版本，由于我们只关于最后一个单词的长度，所以开头有多少个空格起始我们并不在意，我们从字符串末尾开始，先将末尾的空格都去掉，然后开始找非空格的字符的长度即可，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int lengthOfLastWord(string s) {
4         int right = s.size() - 1, res = 0;
5         while (right >= 0 && s[right] == ' ') --right;
6         while (right >= 0 && s[right] != ' ') {
7             --right;
8             ++res;
9         }
10        return res;
11    }
12 };

```

59. 螺旋矩阵 II

给定一个正整数 n，生成一个包含 1 到 n² 所有元素，且元素按顺时针顺序螺旋排列的正方形矩阵。

示例：

输入：3

输出：

```
[  
 [ 1, 2, 3 ],  
 [ 8, 9, 4 ],  
 [ 7, 6, 5 ]  
]
```

此题跟之前那道 Spiral Matrix 螺旋矩阵 本质上没什么区别，就相当于个类似逆运算的过程，这道题是要按螺旋的顺序来填数，由于给定矩形是个正方形，我们计算环数时用 n / 2 来计算，若 n 为奇数时，此时最中间的那个点没有被算在环数里，所以最后需要单独赋值，还是下标转换问题是难点，参考之前 Spiral Matrix 螺旋矩阵 的讲解来转换下标吧，代码如下：

```
1 class Solution {  
2 public:  
3     vector<vector<int>> generateMatrix(int n) {  
4         vector<vector<int>> res(n, vector<int>(n, 1));  
5         int val = 1, p = n;  
6         for (int i = 0; i < n / 2; ++i, p -= 2) {  
7             for (int col = i; col < i + p; ++col)  
8                 res[i][col] = val++;  
9             for (int row = i + 1; row < i + p; ++row)  
10                res[row][i + p - 1] = val++;  
11                for (int col = i + p - 2; col >= i; --col)  
12                    res[i + p - 1][col] = val++;  
13                    for (int row = i + p - 2; row > i; --row)  
14                        res[row][i] = val++;  
15                }  
16                if (n % 2 != 0) res[n / 2][n / 2] = val;  
17                return res;  
18            }  
19        };
```

CPP

60. 第 k 个排列

给出集合 [1,2,3,...,n]，其所有元素共有 n! 种排列。

按大小顺序列出所有排列情况，并一一标记，当 n = 3 时，所有排列如下：

- "123"
- "132"
- "213"
- "231"
- "312"
- "321"

给定 n 和 k，返回第 k 个排列。

说明：

- 给定 n 的范围是 [1, 9]。
- 给定 k 的范围是[1, n!]。

这道题是让求出n个数字的第k个排列组合，由于其特殊性，我们不用将所有的排列组合的情况都求出来，然后返回其第k个，我们可以只求出第k个排列组合即可，那么难点就在于如何知道数字的排列顺序，可参见网友喜刷刷的博客，首先我们要知道当n = 3时，其排列组合共有 $3! = 6$ 种，当n = 4时，其排列组合共有 $4! = 24$ 种，我们就以n = 4, k = 17的情况来分析，所有排列组合情况如下：

```

1234
1243
1324
1342
1423
1432
2134
2143
2314
2341
2413
2431
3124
3142
3214
3241
3412      <--- k = 17
3421
4123
4132
4213
4231
4312
4321

```

我们可以发现，每一位上1,2,3,4分别都出现了6次，当最高位上的数字确定了，第二高位每个数字都出现了2次，当第二高位也确定了，第三高位上的数字都只出现了1次，当第三高位确定了，那么第四高位上的数字也只能出现一次，下面我们来看k = 17这种情况的每位数字如何确定，由于k = 17是转化为数组下标为16：

最高位可取1,2,3,4中的一个，每个数字出现 $3! = 6$ 次，所以k = 16的第一位数字的下标为 $16 / 6 = 2$ ，在 "1234" 中即3被取出。这里我们的k是要求的坐标为k的全排列序列，我们定义 k' 为当最高位确定后，要求的全排列序列在新范围中的位置，同理， k'' 为当第二高位确定后，所要求的全排列序列在新范围中的位置，以此类推，下面来具体看看：

第二位此时从1,2,4中取一个， $k = 16$ ，则此时的 $k' = 16 \% (3!) = 4$ ，如下所示，而剩下的每个数字出现 $2! = 2$ 次，所以第二数字的下标为 $4 / 2 = 2$ ，在 "124" 中即4被取出。

```

3124
3142
3214
3241
3412      <--- k' = 4
3421

```

第三位此时从1,2中去一个， $k' = 4$ ，则此时的 $k'' = 4 \% (2!) = 0$ ，如下所示，而剩下的每个数字出现 $1! = 1$ 次，所以第三个数字的下标为 $0 / 1 = 0$ ，在 "12" 中即1被取出。

```
3412 <--- k'' = 0
3421
```

第四位是从2中取一个， $k'' = 0$ ，则此时的 $k''' = 0 \% (1!) = 0$ ，如下所示，而剩下的每个数字出现 $0! = 1$ 次，所以第四个数字的下标为 $0 / 1 = 0$ ，在 "2" 中即2被取出。

```
3412 <--- k''' = 0
```

那么我们就可以找出规律了

```
a1 = k / (n - 1)!
k1 = k

a2 = k1 / (n - 2)!
k2 = k1 % (n - 2)!

...
an-1 = kn-2 / 1!
kn-1 = kn-2 % 1!

an = kn-1 / 0!
kn = kn-1 % 0!
```

```
1 class Solution {
2 public:
3     string getPermutation(int n, int k) {
4         string res;
5         string num = "123456789";
6         vector<int> f(n, 1);
7         for (int i = 1; i < n; ++i) f[i] = f[i - 1] * i;
8         --k;
9         for (int i = n; i >= 1; --i) {
10             int j = k / f[i - 1];
11             k %= f[i - 1];
12             res.push_back(num[j]);
13             num.erase(j, 1);
14         }
15         return res;
16     }
17 }
```

CPP

61. 旋转链表

Given a list, rotate the list to the right by k places, where k is non-negative.

For example:

Given 1->2->3->4->5->NULL and k = 2,
return 4->5->1->2->3->NULL.

这道旋转链表的题和之前那道 Rotate Array 旋转数组 很类似，但是比那道要难一些，因为链表的值不能通过下表来访问，只能一个一个的走，我刚开始拿到这题首先想到的就是用快慢指针来解，快指针先走 k 步，然后两个指针一起走，当快指针走到末尾时，慢指针的下一个位置是新的顺序的头结点，这样就可以旋转链表了，自信满满的写完程序，放到OJ上跑，以为能一次通

过，结果跪在了各种特殊情况，首先一个就是当原链表为空时，直接返回NULL，还有就是当k大于链表长度和k远远大于链表长度时该如何处理，我们需要首先遍历一遍原链表得到链表长度n，然后k对n取余，这样k肯定小于n，就可以用上面的算法了，代码如下：

解法1：

```

1 class Solution {
2 public:
3     ListNode *rotateRight(ListNode *head, int k) {
4         if (!head) return NULL;
5         int n = 0;
6         ListNode *cur = head;
7         while (cur) {
8             ++n;
9             cur = cur->next;
10        }
11        k %= n;
12        ListNode *fast = head, *slow = head;
13        for (int i = 0; i < k; ++i) {
14            if (fast) fast = fast->next;
15        }
16        if (!fast) return head;
17        while (fast->next) {
18            fast = fast->next;
19            slow = slow->next;
20        }
21        fast->next = head;
22        fast = slow->next;
23        slow->next = NULL;
24        return fast;
25    }
26 };

```

这道题还有一种解法，跟上面的方法类似，但是不用快慢指针，一个指针就够了，原理是先遍历整个链表获得链表长度n，然后此时把链表头和尾链接起来，在往后走 $n - k \% n$ 个节点就到达新链表的头结点前一个点，这时断开链表即可，代码如下：

解法2：

```

1 class Solution {
2 public:
3     ListNode *rotateRight(ListNode *head, int k) {
4         if (!head) return NULL;
5         int n = 1;
6         ListNode *cur = head;
7         while (cur->next) {
8             ++n;
9             cur = cur->next;
10        }
11        cur->next = head;
12        int m = n - k % n;
13        for (int i = 0; i < m; ++i) {
14            cur = cur->next;
15        }
16        ListNode *newhead = cur->next;
17        cur->next = NULL;
18        return newhead;
19    }
20};

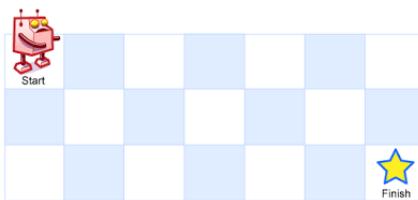
```

62. 不同的路径

A robot is located at the top-left corner of a $m \times n$ grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

How many possible unique paths are there?



Above is a 3×7 grid. How many possible unique paths are there?

Note: m and n will be at most 100.

这道题让求所有不同的路径的个数，一开始还真把我难住了，因为之前好像没有遇到过这类的问题，所以感觉好像有种无从下手的感觉。在网上找攻略之后才恍然大悟，原来这跟之前那道 Climbing Stairs 爬梯子问题很类似，那道题是说可以每次能爬一格或两格，问到达顶部的所有不同爬法的个数。而这道题是每次可以向下走或者向右走，求到达最右下角的所有不同走法的个数。那么跟爬梯子问题一样，我们需要用动态规划Dynamic Programming来解，我们可以维护一个二维数组dp，其中 $dp[i][j]$ 表示到当前位置不同的走法的个数，然后可以得到递推式为: $dp[i][j] = dp[i-1][j] + dp[i][j-1]$ ，这里为了节省空间，我们使用一维数组dp，一行一行的刷新也可以，代码如下：

解法1:

```

1 class Solution {
2 public:
3     int uniquePaths(int m, int n) {
4         vector<int> dp(n, 1);
5         for (int i = 1; i < m; ++i) {
6             for (int j = 1; j < n; ++j) {
7                 dp[j] += dp[j - 1];
8             }
9         }
10        return dp[n - 1];
11    }
12 };

```

这道题其实还有另一种很数学的解法，参见网友Code Ganker的博客，实际相当于机器人总共走了 $m + n - 2$ 步，其中 $m - 1$ 步向下走， $n - 1$ 步向右走，那么总共不同的方法个数就相当于在步数里面 $m - 1$ 和 $n - 1$ 中较小的那个数的取法，实际上是一道组合数的问题，写出代码如下：

解法2：

```

1 class Solution {
2 public:
3     int uniquePaths(int m, int n) {
4         double num = 1, denom = 1;
5         int small = m > n ? n : m;
6         for (int i = 1; i <= small - 1; ++i) {
7             num *= m + n - 1 - i;
8             denom *= i;
9         }
10        return (int)(num / denom);
11    }
12 };

```

63. 不同的路径之二

Follow up for "Unique Paths":

Now consider if some obstacles are added to the grids. How many unique paths would there be?

An obstacle and empty space is marked as 1 and 0 respectively in the grid.

For example,

There is one obstacle in the middle of a 3x3 grid as illustrated below.

```
[  
[0,0,0],  
[0,1,0],  
[0,0,0]  
]
```

The total number of unique paths is 2.

Note: m and n will be at most 100.

这道题是之前那道 Unique Paths 不同的路径的延伸，在路径中加了一些障碍物，还是用动态规划Dynamic Programming来解，不同的是当遇到为1的点，将该位置的dp数组中的值清零，其余和之前那道题并没有什么区别，代码如下：

解法1:

```

1 class Solution {
2 public:
3     int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
4         if (obstacleGrid.empty() || obstacleGrid[0].empty() || obstacleGrid[0][0] == 1)
5             return 0;
6         vector<vector<int>> dp(obstacleGrid.size(), vector<int>(obstacleGrid[0].size(),
7 0));
8         for (int i = 0; i < obstacleGrid.size(); ++i) {
9             for (int j = 0; j < obstacleGrid[i].size(); ++j) {
10                 if (obstacleGrid[i][j] == 1) dp[i][j] = 0;
11                 else if (i == 0 && j == 0) dp[i][j] = 1;
12                 else if (i == 0 && j > 0) dp[i][j] = dp[i][j - 1];
13                 else if (i > 0 && j == 0) dp[i][j] = dp[i - 1][j];
14                 else dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
15             }
16         }
17         return dp.back().back();
18     }
19 };

```

或者我们也可以使用一维dp数组来解，省一些空间，参见代码如下：

解法2:

```

1 class Solution {
2 public:
3     int uniquePathsWithObstacles(vector<vector<int>> &obstacleGrid) {
4         if (obstacleGrid.empty() || obstacleGrid[0].empty()) return 0;
5         int m = obstacleGrid.size(), n = obstacleGrid[0].size();
6         if (obstacleGrid[0][0] == 1) return 0;
7         vector<int> dp(n, 0);
8         dp[0] = 1;
9         for (int i = 0; i < m; ++i) {
10             for (int j = 0; j < n; ++j) {
11                 if (obstacleGrid[i][j] == 1) dp[j] = 0;
12                 else if (j > 0) dp[j] += dp[j - 1];
13             }
14         }
15         return dp[n - 1];
16     }
17 };

```

64. 最小路径和

Given a $m \times n$ grid filled with non-negative numbers, find a path from top left to bottom right which minimizes the sum of all numbers along its path.

Note: You can only move either down or right at any point in time.

这道题跟之前那道 Dungeon Game 地牢游戏没有什么太大的区别，都需要用动态规划Dynamic Programming来做，这应该算是DP问题中比较简单的一类，我们维护一个二维的dp数组，其中 $dp[i][j]$ 表示当前位置的最小路径和，递推式也容易写出来 $dp[i][j] = grid[i][j] + \min(dp[i - 1][j], dp[i][j - 1])$ ，反正难度不算大，代码如下：

```

1 class Solution {
2 public:
3     int minPathSum(vector<vector<int>> &grid) {
4         int m = grid.size(), n = grid[0].size();
5         int dp[m][n];
6         dp[0][0] = grid[0][0];
7         for (int i = 1; i < m; ++i) dp[i][0] = grid[i][0] + dp[i - 1][0];
8         for (int i = 1; i < n; ++i) dp[0][i] = grid[0][i] + dp[0][i - 1];
9         for (int i = 1; i < m; ++i) {
10             for (int j = 1; j < n; ++j) {
11                 dp[i][j] = grid[i][j] + min(dp[i - 1][j], dp[i][j - 1]);
12             }
13         }
14     return dp[m - 1][n - 1];
15 }
16 };

```

65. 验证数字

Validate if a given string is numeric.

Some examples:

"0" => true
 " 0.1 " => true
 "abc" => false
 "1 a" => false
 "2e10" => true

Note: It is intended for the problem statement to be ambiguous. You should gather all requirements up front before implementing one.

这道验证数字的题比想象中的要复杂的多，有很多情况需要考虑，而OJ上给这道题的分类居然是Easy，Why？而10.9% 的全场最低的Accept Rate正说明这道题的难度，网上有很多解法，有利用有限自动机Finite Automata Machine的程序写的简洁优雅(<http://blog.csdn.net/kenden23/article/details/18696083>)，还有利用正则表达式，更是写的丧心病狂的简洁(<http://blog.csdn.net/fightforyourdream/article/details/12900751>)。而我主要还是用最一般的写法，参考了网上另一篇博文(<http://yucoding.blogspot.com/2013/05/leetcode-question-118-valid-number.html>)，处理各种情况。

首先，从题目中给的一些例子可以分析出来，我们所需要关注的除了数字以外的特殊字符有空格 ' '，小数点 '.'，自然数'e/E'，还要加上正负号 '+/-'，除了这些字符需要考虑意外，出现了任何其他的字符，可以马上判定不是数字。下面我们来一一分析这些出现了也可能是数字的特殊字符： +

1. 空格 '': 空格分为两种情况需要考虑，一种是出现在开头和末尾的空格，一种是出现在中间的字符。出现在开头和末尾的空格不影响数字，而一旦中间出现了空格，则立马不是数字。解决方法：预处理时去掉字符的首位空格，中间再检测到空格，则判定不是数字。
2. 小数点 '!': 小数点需要分的情况较多，首先的是小数点只能出现一次，但是小数点可以出现在任何位置，开头(".3")，中间("1.e2")，以及结尾("1.")，而且需要注意的是，小数点不能出现在自然数'e/E'之后，如 "1e.1" false, "1e1.1" false。还有，当小数点位于末尾时，前面必须是数字，如 "1." true, "-." false。解决方法：开头中间结尾三个位置分开讨论情况。
3. 自然数'e/E': 自然数的前后必须有数字，即自然数不能出现在开头和结尾，如 "e" false, ".e1" false, "3.e" false, "3.e1" true。而且小数点只能出现在自然数之前，还有就是自然数前面不能是符号，如 "+e1" false, "1+e" false. 解决方法：开头中间结尾三个位置分开讨论情况。
4. 正负号 '+/-': 正负号可以再开头出现，可以再自然数e之后出现，但不能是最后一个字符，后面得有数字，如 "+1.e+5" true。解决方法：开头中间结尾三个位置分开讨论情况。

下面我们开始正式分开头中间结尾三个位置来讨论情况：

1. 在讨论三个位置之前做预处理，去掉字符串首尾的空格，可以采用两个指针分别指向开头和结尾，遇到空格则跳过，分别指向开头结尾非空格的字符。
2. 对首字符处理，首字符只能为数字或者正负号 '+/-'，我们需要定义三个flag在标示我们是否之前检测到过小数点，自然数和正负号。首字符如为数字或正负号，则标记对应的flag，若不是，直接返回false。
3. 对中间字符的处理，中间字符会出现五种情况，数字，小数点，自然数，正负号和其他字符。

若是数字，标记flag并通过。

若是自然数，则必须是第一次出现自然数，并且前一个字符不能是正负号，而且之前一定要出现过数字，才能标记flag通过。

若是正负号，则之前的字符必须是自然数e，才能标记flag通过。

若是小数点，则必须是第一次出现小数点并且自然数没有出现过，才能标记flag通过。

若是其他，返回false。

1. 对尾字符处理，最后一个字符只能是数字或小数点，其他字符都返回false。

若是数字，返回true。

若是小数点，则必须是第一次出现小数点并且自然数没有出现过，还有前面必须是数字，才能返回true。

解法1：

```

1 class Solution {
2 public:
3     bool isNumber(string s) {
4         int len = s.size();
5         int left = 0, right = len - 1;
6         bool eExisted = false;
7         bool dotExisted = false;
8         bool digitExisted = false;
9         // Delete spaces in the front and end of string
10        while (s[left] == ' ') ++left;
11        while (s[right] == ' ') --right;
12        // If only have one char and not digit, return false
13        if (left >= right && (s[left] < '0' || s[left] > '9')) return false;
14        // Process the first char
15        if (s[left] == '.') dotExisted = true;
16        else if (s[left] >= '0' && s[left] <= '9') digitExisted = true;
17        else if (s[left] != '+' && s[left] != '-') return false;
18        // Process the middle chars
19        for (int i = left + 1; i <= right - 1; ++i) {
20            if (s[i] >= '0' && s[i] <= '9') digitExisted = true;
21            else if (s[i] == 'e' || s[i] == 'E') { // e/E cannot follow +/-, must follow a
22                digit
23                if (!eExisted && s[i - 1] != '+' && s[i - 1] != '-' && digitExisted)
24                    eExisted = true;
25                else return false;
26            } else if (s[i] == '+' || s[i] == '-') { // +/- can only follow e/E
27                if (s[i - 1] != 'e' && s[i - 1] != 'E') return false;
28            } else if (s[i] == '.') { // dot can only occur once and cannot occur after e/E
29                if (!dotExisted && !eExisted) dotExisted = true;
30                else return false;
31            } else return false;
32        }
33        // Process the last char, it can only be digit or dot, when is dot, there should be
34        no dot and e/E before and must follow a digit
35        if (s[right] >= '0' && s[right] <= '9') return true;
36        else if (s[right] == '.' && !dotExisted && !eExisted && digitExisted) return true;
37        else return false;
38    }
39};

```

上面的写法略为复杂，我们尝试着来优化一下，根据上面的分析，所有的字符可以分为六大类，空格，符号，数字，小数点，自然底数和其他字符，我们需要五个标志变量，num, dot, exp, sign分别表示数字，小数点，自然底数和符号是否出现，numAfterE表示自然底数后面是否有数字，那么我们分别来看各种情况：

- 空格：我们需要排除的情况是，当前位置是空格而后面一位不为空格，但是之前有数字，小数点，自然底数或者符号出现时返回false。
- 符号：符号前面如果有字符的话必须是空格或者是自然底数，标记sign为true。
- 数字：标记num和numAfterE为true。
- 小数点：如果之前出现过小数点或者自然底数，返回false，否则标记dot为true。
- 自然底数：如果之前出现过自然底数或者之前从未出现过数字，返回false，否则标记exp为true， numAfterE为false。
- 其他字符：返回false。

最后返回num && numAfterE即可。

解法2：

```

1 class Solution {
2 public:
3     bool isNumber(string s) {
4         bool num = false, numAfterE = true, dot = false, exp = false, sign = false;
5         int n = s.size();
6         for (int i = 0; i < n; ++i) {
7             if (s[i] == ' ') {
8                 if (i < n - 1 && s[i + 1] != ' ' && (num || dot || exp || sign)) return
9                     false;
10            } else if (s[i] == '+' || s[i] == '-') {
11                if (i > 0 && s[i - 1] != 'e' && s[i - 1] != ' ') return false;
12                sign = true;
13            } else if (s[i] >= '0' && s[i] <= '9') {
14                num = true;
15                numAfterE = true;
16            } else if (s[i] == '.') {
17                if (dot || exp) return false;
18                dot = true;
19            } else if (s[i] == 'e') {
20                if (exp || !num) return false;
21                exp = true;
22                numAfterE = false;
23            } else return false;
24        }
25        return num && numAfterE;
26    }
27 };

```

66. 加一运算

Given a non-negative number represented as an array of digits, plus one to the number.

The digits are stored such that the most significant digit is at the head of the list.

将一个数字的每个位上的数字分别存到一个一维向量中，最高位在最开头，我们需要给这个数字加一，即在末尾数字加一，如果末尾数字是9，那么则会有进位问题，而如果前面位上的数字仍为9，则需要继续向前进位。具体算法如下：首先判断最后一位是否为9，若不是，直接加一返回，若是，则该位赋0，再继续查前一位，同样的方法，知道查完第一位。如果第一位原本为9，加一后会产生新的一位，那么最后要做的是，查运算完的第一位是否为0，如果是，则在最前头加一个1。代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<int> plusOne(vector<int> &digits) {
4         int n = digits.size();
5         for (int i = n - 1; i >= 0; --i) {
6             if (digits[i] == 9) digits[i] = 0;
7             else {
8                 digits[i] += 1;
9                 return digits;
10            }
11        }
12        if (digits.front() == 0) digits.insert(digits.begin(), 1);
13        return digits;
14    }
15 };

```

我们也可以使用跟之前那道Add Binary类似的做法，我们将carry初始化为1，然后相当于digits加了一个0，处理方法跟之前那道题一样，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<int> plusOne(vector<int>& digits) {
4         if (digits.empty()) return digits;
5         int carry = 1, n = digits.size();
6         for (int i = n - 1; i >= 0; --i) {
7             if (carry == 0) return digits;
8             int sum = digits[i] + carry;
9             digits[i] = sum % 10;
10            carry = sum / 10;
11        }
12        if (carry == 1) digits.insert(digits.begin(), 1);
13        return digits;
14    }
15 };

```

67. 二进制数相加

Given two binary strings, return their sum (also a binary string).

For example,
a = "11"
b = "1"
Return "100".

二进制数相加，并且保存在string中，要注意的是如何将string和int之间互相转换，并且每位相加时，会有进位的可能，会影响之后相加的结果。而且两个输入string的长度也可能不同。这时我们需要新建一个string，它的长度是两条输入string中的较大的那个，并且把较短的那个输入string通过在开头加字符‘0’来补的较大的那个长度。这时候我们逐个从两个string的末尾开始取出字符，然后转为数字，想加，如果大于等于2，则标记进位标志carry，并且给新string加入一个字符‘0’。代码如下：

解法1：

```

1 class Solution {
2 public:
3     string addBinary(string a, string b) {
4         string res;
5         int na = a.size();
6         int nb = b.size();
7         int n = max(na, nb);
8         bool carry = false;
9         if (na > nb) {
10             for (int i = 0; i < na - nb; ++i) b.insert(b.begin(), '0');
11         }
12         else if (na < nb) {
13             for (int i = 0; i < nb - na; ++i) a.insert(a.begin(), '0');
14         }
15         for (int i = n - 1; i >= 0; --i) {
16             int tmp = 0;
17             if (carry) tmp = (a[i] - '0') + (b[i] - '0') + 1;
18             else tmp = (a[i] - '0') + (b[i] - '0');
19             if (tmp == 0) {
20                 res.insert(res.begin(), '0');
21                 carry = false;
22             }
23             else if (tmp == 1) {
24                 res.insert(res.begin(), '1');
25                 carry = false;
26             }
27             else if (tmp == 2) {
28                 res.insert(res.begin(), '0');
29                 carry = true;
30             }
31             else if (tmp == 3) {
32                 res.insert(res.begin(), '1');
33                 carry = true;
34             }
35         }
36         if (carry) res.insert(res.begin(), '1');
37         return res;
38     }
39 };

```

下面这种写法又巧妙又简洁，用了两个指针分别指向a和b的末尾，然后每次取出一个字符，转为数字，若无法取出字符则按0处理，然后定义进位carry，初始化为0，将三者加起来，对2取余即为当前位的数字，对2取商即为当前进位的值，记得最后还要判断下carry，如果为1的话，要在结果最前面加上一个1，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     string addBinary(string a, string b) {
4         string res = "";
5         int m = a.size() - 1, n = b.size() - 1, carry = 0;
6         while (m >= 0 || n >= 0) {
7             int p = m >= 0 ? a[m--] - '0' : 0;
8             int q = n >= 0 ? b[n--] - '0' : 0;
9             int sum = p + q + carry;
10            res = to_string(sum % 2) + res;
11            carry = sum / 2;
12        }
13        return carry == 1 ? "1" + res : res;
14    }
15 };

```

68. 文本左右对齐

Given an array of words and a length L, format the text such that each line has exactly L characters and is fully (left and right) justified.

You should pack your words in a greedy approach; that is, pack as many words as you can in each line. Pad extra spaces ' ' when necessary so that each line has exactly Lcharacters.

Extra spaces between words should be distributed as evenly as possible. If the number of spaces on a line do not divide evenly between words, the empty slots on the left will be assigned more spaces than the slots on the right.

For the last line of text, it should be left justified and no extra space is inserted between words.

For example,
words: ["This", "is", "an", "example", "of", "text", "justification."]
L: 16.

Return the formatted lines as:

```
[
    "This    is    an",
    "example  of text",
    "justification.  "
]
```

Note: Each word is guaranteed not to exceed L in length.

click to show corner cases.

Corner Cases:

A line other than the last line might contain only one word. What should you do in this case?
In this case, that line should be left-justified.

我将这道题翻译为文本的左右对齐是因为这道题像极了word软件里面的文本左右对齐功能，这道题我前前后后折腾了快四个小时终于通过了OJ，完成了之后想着去网上搜搜看有没有更简单的方法，搜了一圈发现都差不多，都挺复杂的，于是乎就按自己的思路来说吧，由于返回的结果是多行的，所以我们在处理的时候也要一行一行的来处理，首先要做的就是确定每一行能放下的

单词数，这个不难，就是比较n个单词的长度和加上n - 1个空格的长度跟给定的长度L来比较即可，找到了一行能放下的单词个数，然后计算出这一行存在的空格的个数，是用给定的长度L减去这一行所有单词的长度和。得到了空格的个数之后，就要在每个单词后面插入这些空格，这里有两种情况，比如某一行有两个单词"to" 和 "a"，给定长度L为6，如果这行不是最后一行，那么应该输出"to a"，如果是最后一行，则应该输出 "to a "，所以这里需要分情况讨论，最后一行的处理方法和其他行之间略有不同。最后一个难点就是，如果一行有三个单词，这时候中间有两个空，如果空格数不是2的倍数，那么左边的空间里要比右边的空间里多加入一个空格，那么我们只需要用总的空格数除以空间个数，能除尽最好，说明能平均分配，除不尽的话就多加个空格放在左边的空间里，以此类推，具体实现过程还是看代码吧：

```

1 class Solution {
2 public:
3     vector<string> fullJustify(vector<string> &words, int L) {
4         vector<string> res;
5         int i = 0;
6         while (i < words.size()) {
7             int j = i, len = 0;
8             while (j < words.size() && len + words[j].size() + j - i <= L) {
9                 len += words[j++].size();
10            }
11            string out;
12            int space = L - len;
13            for (int k = i; k < j; ++k) {
14                out += words[k];
15                if (space > 0) {
16                    int tmp;
17                    if (j == words.size()) {
18                        if (j - k == 1) tmp = space;
19                        else tmp = 1;
20                    } else {
21                        if (j - k - 1 > 0) {
22                            if (space % (j - k - 1) == 0) tmp = space / (j - k - 1);
23                            else tmp = space / (j - k - 1) + 1;
24                        } else tmp = space;
25                    }
26                    out.append(tmp, ' ');
27                    space -= tmp;
28                }
29            }
30            res.push_back(out);
31            i = j;
32        }
33        return res;
34    }
35}

```

69. 求平方根

Implement int sqrt(int x).

Compute and return the square root of x.

这道题要求我们求平方根，我们能想到的方法就是算一个候选值的平方，然后和x比较大小，为了缩短查找时间，我们采用二分搜索法来找平方根，这里属于博主之前总结的LeetCode Binary Search Summary 二分搜索法小结中的第三类的变形，找最后一个不大于目标值的数，代码如下：

解法1：

```

1 class Solution {
2 public:
3     int mySqrt(int x) {
4         if (x <= 1) return x;
5         int left = 0, right = x;
6         while (left < right) {
7             int mid = left + (right - left) / 2;
8             if (x / mid >= mid) left = mid + 1;
9             else right = mid;
10        }
11        return right - 1;
12    }
13 };

```

这道题还有另一种解法，是利用牛顿迭代法，记得高数中好像讲到过这个方法，是用逼近法求方程根的神器，在这里也可以借用一下，可参见网友Annie Kim's Blog的博客，因为要求 $x^2 = n$ 的解，令 $f(x) = x^2 - n$ ，相当于求解 $f(x)=0$ 的解，可以求出递推式如下：

$$x_{i+1} = x_i - (x_i^2 - n) / (2x_i) = x_i - x_i / 2 + n / (2x_i) = x_i / 2 + n / 2x_i = (x_i + n/x_i) / 2$$

解法2：

```

1 class Solution {
2 public:
3     int mySqrt(int x) {
4         if (x == 0) return 0;
5         double res = 1, pre = 0;
6         while (abs(res - pre) > 1e-6) {
7             pre = res;
8             res = (res + x / res) / 2;
9         }
10        return int(res);
11    }
12 };

```

也是牛顿迭代法，写法更加简洁一些，注意为了防止越界，声明为长整型，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int mySqrt(int x) {
4         long res = x;
5         while (res * res > x) {
6             res = (res + x / res) / 2;
7         }
8         return res;
9     }
10 };

```

70. 爬楼梯

```

1 class Solution {
2     public:
3         int climbStairs(int n) {
4             if(n == 1) return 1;
5             if(n == 2) return 2;
6             int cur = 2, pre = 1;
7             int result = 0;
8             for(int i=3; i<n; ++i){
9                 result = cur + pre;
10                pre = cur;
11                cur = result;
12            }
13
14            return cur + pre;
15        }
16    };

```

71. 简化路径

Given an absolute path for a file (Unix-style), simplify it.

For example,
path = "/home/", => "/home"
path = "/a/./b/../../c/", => "/c"

[click to show corner cases.](#)

Corner Cases:

Did you consider the case where path = "/../"?
In this case, you should return "/".
Another corner case is the path might contain multiple slashes '/' together, such as
"/home//foo/".
In this case, you should ignore redundant slashes and return "/home/foo".

这道题让简化给定的路径，光根据题目中给的那个例子还真不太好总结出规律，应该再加上两个例子 path = "/a./b../c/", => "/a/c" 和 path = "/a./b/c/", => "/a/b/c"，这样我们就可以知道中间是"."的情况直接去掉，是".."时删掉它上面挨着的一个路径，而下面的边界条件给的一些情况中可以得知，如果是空的话返回"/"，如果有多个"/"只保留一个。那么我们可以把路径看做是由一个或多个"/"分割开的众多子字符串，把它们分别提取出来一一处理即可，代码如下：

解法1：

```

1 class Solution {
2 public:
3     string simplifyPath(string path) {
4         vector<string> v;
5         int i = 0;
6         while (i < path.size()) {
7             while (path[i] == '/' && i < path.size()) ++i;
8             if (i == path.size()) break;
9             int start = i;
10            while (path[i] != '/' && i < path.size()) ++i;
11            int end = i - 1;
12            string s = path.substr(start, end - start + 1);
13            if (s == "..") {
14                if (!v.empty()) v.pop_back();
15            } else if (s != ".") {
16                v.push_back(s);
17            }
18        }
19        if (v.empty()) return "/";
20        string res;
21        for (int i = 0; i < v.size(); ++i) {
22            res += '/' + v[i];
23        }
24        return res;
25    }
26};

```

还有一种解法是利用了C语言中的函数strtok来分隔字符串，但是需要把string和char*类型相互转换，转换方法请猛戳[这里](#)。除了这块不同，其余的思想和上面那种解法相同，代码如下：

解法2：

```

1 class Solution {
2 public:
3     string simplifyPath(string path) {
4         vector<string> v;
5         char *cstr = new char[path.length() + 1];
6         strcpy(cstr, path.c_str());
7         char *pch = strtok(cstr, "/");
8         while (pch != NULL) {
9             string p = string(pch);
10            if (p == "..") {
11                if (!v.empty()) v.pop_back();
12            } else if (p != ".") {
13                v.push_back(p);
14            }
15            pch = strtok(NULL, "/");
16        }
17        if (v.empty()) return "/";
18        string res;
19        for (int i = 0; i < v.size(); ++i) {
20            res += '/' + v[i];
21        }
22        return res;
23    }
24};

```

C++中也有专门处理字符串的机制，我们可以使用stringstream来分隔字符串，然后对每一段分别处理，思路和上面的方法相似，参见代码如下：

解法3：

```
1 class Solution {
2 public:
3     string simplifyPath(string path) {
4         string res, t;
5         stringstream ss(path);
6         vector<string> v;
7         while (getline(ss, t, '/')) {
8             if (t == "" || t == ".") continue;
9             if (t == ".." && !v.empty()) v.pop_back();
10            else if (t != "..") v.push_back(t);
11        }
12        for (string s : v) res += "/" + s;
13        return res.empty() ? "/" : res;
14    }
15 }
```

72. 编辑距离

Given two words word1 and word2, find the minimum number of steps required to convert word1 to word2. (each operation is counted as 1 step.)

You have the following 3 operations permitted on a word:

- a) Insert a character
- b) Delete a character
- c) Replace a character

这道题让求从一个字符串转变到另一个字符串需要的变换步骤，共有三种变换方式，插入一个字符，删除一个字符，和替换一个字符。根据以往的经验，对于字符串相关的题目十有八九都是用动态规划Dynamic Programming来解，这道题也不例外。这道题我们需要维护一个二维的数组dp，其中 $dp[i][j]$ 表示从word1的前*i*个字符转换到word2的前*j*个字符所需要的步骤。那我们可以先给这个二维数组dp的第一行第一列赋值，这个很简单，因为第一行和第一列对应的总有一个字符串是空串，于是转换步骤完全是另一个字符串的长度。跟以往的DP题目类似，难点还是在于找出递推式，我们可以举个例子来看，比如word1是“bbc”，word2是“abcd”，那么我们可以得到dp数组如下：

\emptyset	a	b	c	d
\emptyset	0	1	2	3
b	1	1	1	2
c	2	2	1	2
c	3	3	2	1

我们通过观察可以发现，当 $word1[i] == word2[j]$ 时， $dp[i][j] = dp[i - 1][j - 1]$ ，其他情况时， $dp[i][j]$ 是其左，左上，上的三个值中的最小值加1，那么可以得到递推式为：

```
dp[i][j] =      /   dp[i - 1][j - 1],  if word1[i - 1] == word2[j - 1]
           \   min(dp[i - 1][j - 1], min(dp[i - 1][j], dp[i][j - 1])) + 1  else
```

```

1 class Solution {
2 public:
3     int minDistance(string word1, string word2) {
4         int n1 = word1.size(), n2 = word2.size();
5         int dp[n1 + 1][n2 + 1];
6         for (int i = 0; i <= n1; ++i) dp[i][0] = i;
7         for (int i = 0; i <= n2; ++i) dp[0][i] = i;
8         for (int i = 1; i <= n1; ++i) {
9             for (int j = 1; j <= n2; ++j) {
10                 if (word1[i - 1] == word2[j - 1]) {
11                     dp[i][j] = dp[i - 1][j - 1];
12                 } else {
13                     dp[i][j] = min(dp[i - 1][j - 1], min(dp[i - 1][j], dp[i][j - 1])) + 1;
14                 }
15             }
16         }
17         return dp[n1][n2];
18     }
19 };
20

```

73. 矩阵赋零

Given a $m \times n$ matrix, if an element is 0, set its entire row and column to 0. Do it in place.

[click to show follow up.](#)

Follow up:

Did you use extra space?

A straight forward solution using $O(mn)$ space is probably a bad idea.

A simple improvement uses $O(m + n)$ space, but still not the best solution.

Could you devise a constant space solution?

据说这题是CareerCup上的原题，我还没有刷CareerCup，所以不知道啦，不过这题也不算难，虽然我也是看了网上的解法照着写的，但是下次遇到绝对想的起来。这道题中说的空间复杂度为 $O(mn)$ 的解法自不用多说，直接新建一个和matrix等大小的矩阵，然后一行一行的扫，只要有0，就将新建的矩阵的对应行全赋0，行扫完再扫列，然后把更新完的矩阵赋给matrix即可，这个算法的空间复杂度太高。将其优化到 $O(m+n)$ 的方法是，用一个长度为m的一维数组记录各行中是否有0，用一个长度为n的一维数组记录各列中是否有0，最后直接更新matrix数组即可。这道题的要求是用 $O(1)$ 的空间，那么我们就不能新建数组，我们考虑就用原数组的第一行第一列来记录各行各列是否有0。

- 先扫描第一行第一列，如果有0，则将各自的flag设置为true
- 然后扫描除去第一行第一列的整个数组，如果有0，则将对应的第一行和第一列的数字赋0
- 再次遍历除去第一行第一列的整个数组，如果对应的第一行和第一列的数字有一个为0，则将当前值赋0
- 最后根据第一行第一列的flag来更新第一行第一列

代码如下：

```

1 class Solution {
2 public:
3     void setZeroes(vector<vector<int> > &matrix) {
4         if (matrix.empty() || matrix[0].empty()) return;
5         int m = matrix.size(), n = matrix[0].size();
6         bool rowZero = false, colZero = false;
7         for (int i = 0; i < m; ++i) {
8             if (matrix[i][0] == 0) colZero = true;
9         }
10        for (int i = 0; i < n; ++i) {
11            if (matrix[0][i] == 0) rowZero = true;
12        }
13        for (int i = 1; i < m; ++i) {
14            for (int j = 1; j < n; ++j) {
15                if (matrix[i][j] == 0) {
16                    matrix[0][j] = 0;
17                    matrix[i][0] = 0;
18                }
19            }
20        }
21        for (int i = 1; i < m; ++i) {
22            for (int j = 1; j < n; ++j) {
23                if (matrix[0][j] == 0 || matrix[i][0] == 0) {
24                    matrix[i][j] = 0;
25                }
26            }
27        }
28        if (rowZero) {
29            for (int i = 0; i < n; ++i) matrix[0][i] = 0;
30        }
31        if (colZero) {
32            for (int i = 0; i < m; ++i) matrix[i][0] = 0;
33        }
34    }
35 };

```

74. 搜索一个二维矩阵

Write an efficient algorithm that searches for a value in an $m \times n$ matrix. This matrix has the following properties:

Integers in each row are sorted from left to right.

The first integer of each row is greater than the last integer of the previous row.

For example,

Consider the following matrix:

```
[
  [1, 3, 5, 7],
  [10, 11, 16, 20],
  [23, 30, 34, 50]
]
```

Given target = 3, return true.

这道题要求搜索一个二维矩阵，由于给的矩阵是有序的，所以很自然的想到要用二分查找法，我们可以在第一列上先用一次二分查找法找到目标值所在的行的位置，然后在该行上再用一次二分查找法来找是否存在目标值，代码如下：

解法1：

```
1 class Solution {
2 public:
3     bool searchMatrix(vector<vector<int>> &matrix, int target) {
4         if (matrix.empty() || matrix[0].empty()) return false;
5         if (target < matrix[0][0] || target > matrix.back().back()) return false;
6         int left = 0, right = matrix.size() - 1;
7         while (left <= right) {
8             int mid = (left + right) / 2;
9             if (matrix[mid][0] == target) return true;
10            else if (matrix[mid][0] < target) left = mid + 1;
11            else right = mid - 1;
12        }
13        int tmp = right;
14        left = 0;
15        right = matrix[tmp].size() - 1;
16        while (left <= right) {
17            int mid = (left + right) / 2;
18            if (matrix[tmp][mid] == target) return true;
19            else if (matrix[tmp][mid] < target) left = mid + 1;
20            else right = mid - 1;
21        }
22        return false;
23    }
24};
```

当然这道题也可以使用一次二分查找法，如果我们按S型遍历该二维数组，可以得到一个有序的一维数组，那么我们只需要用一次二分查找法，而关键就在于坐标的转换，如何把二维坐标和一维坐标转换是关键点，把一个长度为n的一维数组转化为m*n的二维数组(m*n = n)后，那么原一维数组中下标为i的元素将出现在二维数组中的[i/n][i%n]的位置，有了这一点，代码很好写出来了：

解法2：

```
1 class Solution {
2 public:
3     bool searchMatrix(vector<vector<int>> &matrix, int target) {
4         if (matrix.empty() || matrix[0].empty()) return false;
5         if (target < matrix[0][0] || target > matrix.back().back()) return false;
6         int m = matrix.size(), n = matrix[0].size();
7         int left = 0, right = m * n - 1;
8         while (left <= right) {
9             int mid = (left + right) / 2;
10            if (matrix[mid / n][mid % n] == target) return true;
11            else if (matrix[mid / n][mid % n] < target) left = mid + 1;
12            else right = mid - 1;
13        }
14        return false;
15    }
16};
```

75. 颜色排序

Given an array with n objects colored red, white or blue, sort them so that objects of the same color are adjacent, with the colors in the order red, white and blue.

Here, we will use the integers 0, 1, and 2 to represent the color red, white, and blue respectively.

Note:

You are not suppose to use the library's sort function for this problem.

[click to show follow up.](#)

Follow up:

A rather straight forward solution is a two-pass algorithm using counting sort.

First, iterate the array counting number of 0's, 1's, and 2's, then overwrite array with total number of 0's, then 1's and followed by 2's.

Could you come up with an one-pass algorithm using only constant space?

这道题的本质还是一道排序的题，题目中给出提示说可以用计数排序，需要遍历数组两遍，那么先来看这种方法，因为数组中只有三个不同的元素，所以实现起来很容易。

- 首先遍历一遍原数组，分别记录0,1,2的个数
- 然后更新原数组，按个数分别赋上0, 1, 2

解法1：

```

1 class Solution {
2 public:
3     void sortColors(int A[], int n) {
4         int count[3] = {0}, idx = 0;
5         for (int i = 0; i < n; ++i) ++count[A[i]];
6         for (int i = 0; i < 3; ++i) {
7             for (int j = 0; j < count[i]; ++j) {
8                 A[idx++] = i;
9             }
10        }
11    }
12 }
```

CPP

题目中还要让只遍历一次数组来求解，那么我需要用双指针来做，分别从原数组的首尾往中心移动。

- 定义red指针指向开头位置，blue指针指向末尾位置
- 从头开始遍历原数组，如果遇到0，则交换该值和red指针指向的值，并将red指针后移一位。若遇到2，则交换该值和blue指针指向的值，并将blue指针前移一位。若遇到1，则继续遍历。

解法2：

```

1 class Solution {
2 public:
3     void sortColors(int A[], int n) {int red = 0, blue = n - 1;
4         for (int i = 0; i <= blue; ++i) {
5             if (A[i] == 0) {
6                 swap(A[i], A[red++]);
7             } else if (A[i] == 2) {
8                 swap(A[i--], A[blue--]);
9             }
10        }
11    }
12 };

```

76. 最小窗口子串

Given a string S and a string T, find the minimum window in S which will contain all the characters in T in complexity O(n).

Example:

Input: S = "ADOBECODEBANC", T = "ABC"

Output: "BANC"

Note:

If there is no such window in S that covers all characters in T, return the empty string "".
If there is such window, you are guaranteed that there will always be only one unique minimum window in S.

这道题给了我们一个原字符串S，还有一个目标字符串T，让我们在S中找到一个最短的子串，使得其包含了T中的所有的字母，并且限制了时间复杂度为O(n)。这道题的要求是要在O(n)的时间里实现找到这个最小窗口字串，那么暴力搜索Brute Force肯定是不能用的，因为遍历所有的子串的时间复杂度是平方级的。那么我们想，时间复杂度卡的这么严，说明我们必须在一次遍历中完成任务，当然遍历若干次也是O(n)，但不一定有这个必要，尝试就一次遍历拿下！那么再来想，既然既然要包含T中所有的字母，那么肯定对于T中的每个字母，肯定要快速查找是否在子串中，既然总时间都卡在了O(n)，我们肯定不想在这里还浪费时间，那么就空间换时间（也就算法题中可以这么干了，七老八十岁的富翁就算用大别野也换不来时间啊。依依东望，望的就是时间呐T.T），使用HashMap，建立T中每个字母与其出现次数之间的映射，那么你可能会有疑问，为啥不用HashSet呢，别急，讲到后面你就知道用HashMap有多妙，简直妙不可言～

目前在脑子一片浆糊的情况下，我们还是从简单的例子来分析吧，题目例子中的S有点长，我们换个短的 S = "ADBANC", T = "ABC"，那么我们肉眼遍历一遍S呗，首先第一个是A，嗯很好，T中有，第二个是D，T中没有，不理它，第三个是B，嗯很好，T中有，第四个又是A，多了一个，礼多人不怪嘛，收下啦，第五个是N，一边凉快去，第六个终于是C了，那么貌似好像需要整个S串，其实不然，我们注意之前有多一个A，但么我们就算去掉第一个A，也没事，因为第四个A可以代替之，第二个D也可以去掉，因为不在T串中，第三个B就不能再去掉了，不然就没有B了。所以最终的答案就"BANC"了。通过上面的描述，你有没有发现一个有趣的现象，我们是先扩展，再收缩，就好像一个窗口一样，先扩大右边界，然后再收缩左边界，上面的例子中我们是右边界无法扩大了后才开始收缩左边界，实际上对于复杂的例子，有可能是扩大右边界，然后缩小一下左边界，然后再扩大右边界等等。这就很像一个不停滑动的窗口了，这就是大名鼎鼎的滑动窗口Sliding Window了，简直是神器啊，能解很多子串，子数组，子序列等等的问题，是必须要熟练掌握的啊！

下面我们来考虑用代码来实现，先来回答一下前面埋下的伏笔，为啥要用HashMap，而不是HashSet，现在应该很显而易见了吧，因为要统计T串中字母的个数，而不是仅仅看某个字母是否在T串中出现。统计好T串中字母的个数之后，我们开始遍历S串，对于S中的每个遍历到的字母，都在HashMap中的映射值减1，如果减1后的映射值仍大于等于0，说明当前遍历到的字母是T串中的字母，我们使用一个计数器cnt，使其自增1。当cnt和T串字母个数相等时，说明此时的窗口已经包含了T串中的所有字母，此时更新一个minLen和结果res，这里的minLen是我们维护的一个全局变量，用来记录出现过的包含T串所有字母的最短的子串的长度，结果res就是这个最短的子串。然后我们要开始收缩左边界，由于我们遍历的时候，对映射值减了1，所以此时去除字母的时候，就要把减去的1加回来，此时如果加1后的值大于0了，说明此时我们少了一个T中的字母，那么cnt值就要减1

了，然后移动左边界left。那么你可能会疑问，对于不在T串中的字母的映射值也这么加呀减呀的，真的大丈夫（带脚布）吗？其实没啥事，因为对于不在T串中的字母，我们减1后，变-1，cnt不会增加，之后收缩左边界的时候，映射值加1后为0，cnt也不会减少，所以并没有什么影响啦，下面是具体的步骤啦：

- 我们最开始先扫描一遍T，把对应的字符及其出现的次数存到HashMap中。
- 然后开始遍历S，就把遍历到的字母对应的HashMap中的value减一，如果减1后仍大于等于0，cnt自增1。
- 如果cnt等于T串长度时，开始循环，纪录一个字串并更新最小字串值。然后将子窗口的左边界向右移，如果某个移除掉的字母是T串中不可缺少的字母，那么cnt自减1，表示此时T串并没有完全匹配。

解法1：

```

1 class Solution {
2 public:
3     string minWindow(string s, string t) {
4         string res = "";
5         unordered_map<char, int> letterCnt;
6         int left = 0, cnt = 0, minLen = INT_MAX;
7         for (char c : t) ++letterCnt[c];
8         for (int i = 0; i < s.size(); ++i) {
9             if (--letterCnt[s[i]] >= 0) ++cnt;
10            while (cnt == t.size()) {
11                if (minLen > i - left + 1) {
12                    minLen = i - left + 1;
13                    res = s.substr(left, minLen);
14                }
15                if (++letterCnt[s[left]] > 0) --cnt;
16                ++left;
17            }
18        }
19        return res;
20    }
21 };

```

CPP

这道题也可以不用HashMap，直接用个int的数组来代替，因为ASCII只有256个字符，所以用个大小为256的int数组即可代替HashMap，但由于一般输入字母串的字符只有128个，所以也可以只用128，其余部分的思路完全相同，虽然只改了一个数据结构，但是运行速度提高了一倍，说明数组还是比HashMap快啊，代码如下：

解法2：

```

1 class Solution {
2 public:
3     string minWindow(string s, string t) {
4         string res = "";
5         vector<int> letterCnt(128, 0);
6         int left = 0, cnt = 0, minLen = INT_MAX;
7         for (char c : t) ++letterCnt[c];
8         for (int i = 0; i < s.size(); ++i) {
9             if (--letterCnt[s[i]] >= 0) ++cnt;
10            while (cnt == t.size()) {
11                if (minLen > i - left + 1) {
12                    minLen = i - left + 1;
13                    res = s.substr(left, minLen);
14                }
15                if (++letterCnt[s[left]] > 0) --cnt;
16                ++left;
17            }
18        }
19        return res;
20    }
21 };

```

77. 组合项

Given two integers n and k, return all possible combinations of k numbers out of 1 ... n.

For example,
If n = 4 and k = 2, a solution is:

```
[
[2,4],
[3,4],
[2,3],
[1,2],
[1,3],
[1,4],
]
```

这道题让求1到n共n个数字里k个数的组合数的所有情况，还是要用深度优先搜索DFS来解，根据以往的经验，像这种要求出所有结果的集合，一般都是用DFS调用递归来解。那么我们建立一个保存最终结果的大集合res，还要定义一个保存每一个组合的小集合out，每次放一个数到out里，如果out里数个数到了k个，则把out保存到最终结果中，否则在下一层中继续调用递归。网友u010500263的博客里有一张图很好的说明了递归调用的顺序，请点击[这里](#)。根据上面分析，可写出代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<vector<int>> combine(int n, int k) {
4         vector<vector<int>> res;
5         vector<int> out;
6         helper(n, k, 1, out, res);
7         return res;
8     }
9     void helper(int n, int k, int level, vector<int>& out, vector<vector<int>>& res) {
10        if (out.size() == k) {res.push_back(out); return;}
11        for (int i = level; i <= n; ++i) {
12            out.push_back(i);
13            helper(n, k, i + 1, out, res);
14            out.pop_back();
15        }
16    }
17 };

```

对于 $n = 5, k = 3$, 处理的结果如下:

```

1 2 3
1   2   4
1   2   5
1   3   4
1   3   5
1   4   5
2   3   4
2   3   5
2   4   5
3   4   5

```

我们再来看一种递归的写法, 此解法没用helper当递归函数, 而是把本身就当作了递归函数, 写起来十分的简洁, 也是非常有趣的一种解法。这个解法用到了一个重要的性质 $C(n, k) = C(n-1, k-1) + C(n-1, k)$, 这应该在我们高中时候学排列组合的时候学过吧, 博主也记不清了。总之, 翻译一下就是, 在 n 个数中取 k 个数的组合项个数, 等于在 $n-1$ 个数中取 $k-1$ 个数的组合项个数再加上在 $n-1$ 个数中取 k 个数的组合项个数之和。这里博主就不证明了, 因为我也不太会, 就直接举题目中的例子来说明吧:

$$C(4, 2) = C(3, 1) + C(3, 2)$$

我们不难写出 $C(3, 1)$ 的所有情况: [1], [2], [3], 还有 $C(3, 2)$ 的所有情况: [1, 2], [1, 3], [2, 3]。我们发现二者加起来为6, 正好是 $C(4, 2)$ 的个数之和。但是我们仔细看会发现, $C(3, 2)$ 的所有情况包含在 $C(4, 2)$ 之中, 但是 $C(3, 1)$ 的每种情况只有一个数字, 而我们需要的结果 $k=2$, 其实很好办, 每种情况后面都加上4, 于是变成了: [1, 4], [2, 4], [3, 4], 加上 $C(3, 2)$ 的所有情况: [1, 2], [1, 3], [2, 3], 正好就得到了 $n=4, k=2$ 的所有情况了。参见代码如下:

解法2:

```

1 class Solution {
2 public:
3     vector<vector<int>> combine(int n, int k) {
4         if (k > n || k < 0) return {};
5         if (k == 0) return {{}};
6         vector<vector<int>> res = combine(n - 1, k - 1);
7         for (auto &a : res) a.push_back(n);
8         for (auto &a : combine(n - 1, k)) res.push_back(a);
9         return res;
10    }
11 };

```

我们再来看一种迭代的写法，也是一种比较巧妙的方法。这里每次先递增最右边的数字，存入结果res中，当右边的数字超过了n，则增加其左边的数字，然后将当前数组赋值为左边的数字，再逐个递增，直到最左边的数字也超过了n，停止循环。对于n=4, k=2时，遍历的顺序如下所示：

```

0 0 #initialization
1 0
1 1
1 2 #push_back
1 3 #push_back
1 4 #push_back
1 5
2 5
2 2
2 3 #push_back
2 4 #push_back
...
3 4 #push_back
3 5
4 5
4 4
4 5
5 5 #stop

```

解法3：

```

1 class Solution {
2 public:
3     vector<vector<int>> combine(int n, int k) {
4         vector<vector<int>> res;
5         vector<int> out(k, 0);
6         int i = 0;
7         while (i >= 0) {
8             ++out[i];
9             if (out[i] > n) --i;
10            else if (i == k - 1) res.push_back(out);
11            else {
12                ++i;
13                out[i] = out[i - 1];
14            }
15        }
16        return res;
17    }
18 };

```

78. 子集合

Given a set of distinct integers, S, return all possible subsets.

Note:

Elements in a subset must be in non-descending order.
The solution set must not contain duplicate subsets.

For example,
If S = [1,2,3], a solution is:

```
[  
    [3],  
    [1],  
    [2],  
    [1,2,3],  
    [1,3],  
    [2,3],  
    [1,2],  
    []  
]
```

这道求子集的问题，由于其要列出所有结果，按照以往的经验，肯定是要用递归来做。这道题其实它的非递归解法相对来说更简单一点，下面我们先来看非递归的解法，由于题目要求子集合中数字的顺序是非降序排列的，所有我们需要预处理，先给输入数组排序，然后再进一步处理，最开始我在想的时候，是想按照子集的长度由少到多全部写出来，比如子集长度为0的就是空集，空集是任何集合的子集，满足条件，直接加入。下面长度为1的子集，直接一个循环加入所有数字，子集长度为2的话可以用两个循环，但是这种想法到后面就行不通了，因为循环的个数不能无限的增长，所以我们必须换一种思路。我们可以一位一位的网上叠加，比如对于题目中给的例子[1,2,3]来说，最开始是空集，那么我们现在要处理1，就在空集上加1，为[1]，现在我们有两个自己[]和[1]，下面我们来处理2，我们在之前的子集基础上，每个都加个2，可以分别得到[2]，[1, 2]，那么现在所有的子集合为[], [1], [2], [1, 2]，同理处理3的情况可得[3], [1, 3], [2, 3], [1, 2, 3]，再加上之前的子集就是所有的子集合了，代码如下：

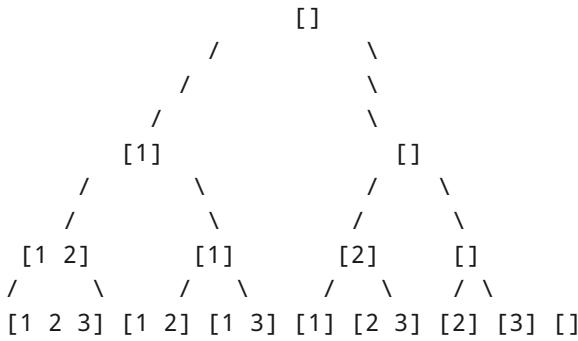
解法1：

```
1 class Solution {  
2     CPP  
3 public:  
4     vector<vector<int>> subsets(vector<int> &S) {  
5         vector<vector<int>> res(1);  
6         sort(S.begin(), S.end());  
7         for (int i = 0; i < S.size(); ++i) {  
8             int size = res.size();  
9             for (int j = 0; j < size; ++j) {  
10                 res.push_back(res[j]);  
11                 res.back().push_back(S[i]);  
12             }  
13         }  
14         return res;  
15     }  
16 }
```

整个添加的顺序为：

```
[]
[1]
[2]
[1 2]
[3]
[1 3]
[2 3]
[1 2 3]
```

下面来看递归的解法，相当于一种深度优先搜索，参见网友JustDoIt的博客，由于原集合每一个数字只有两种状态，要么存在，要么不存在，那么在构造子集时就有选择和不选择两种情况，所以可以构造一棵二叉树，左子树表示选择该层处理的节点，右子树表示不选择，最终的叶节点就是所有子集合，树的结构如下：



解法2：

```

1 class Solution {
2 public:
3     vector<vector<int>> subsets(vector<int> &S) {
4         vector<vector<int>> res;
5         vector<int> out;
6         sort(S.begin(), S.end());
7         getSubsets(S, 0, out, res);
8         return res;
9     }
10    void getSubsets(vector<int> &S, int pos, vector<int> &out, vector<vector<int>> &res) {
11        res.push_back(out);
12        for (int i = pos; i < S.size(); ++i) {
13            out.push_back(S[i]);
14            getSubsets(S, i + 1, out, res);
15            out.pop_back();
16        }
17    }
18}
```

整个添加的顺序为：

```
[]
[1]
[1 2]
[1 2 3]
[1 3]
[2]
[2 3]
[3]
```

最后我们再来看一种解法，这种解法是CareerCup书上给的一种解法，想法也比较巧妙，把数组中所有的数分配一个状态，true表示这个数在子集中出现，false表示在子集中不出现，那么对于一个长度为n的数组，每个数字都有出现与不出现两种情况，所以共有 2^n 种情况，那么我们把每种情况都转换出来就是子集了，我们还是用题目中的例子，[1 2 3]这个数组共有8个子集，每个子集的序号的二进制表示，把是1的位对应原数组中的数字取出来就是一个子集，八种情况都取出来就是所有的子集了，参见代码如下：

	1	2	3	Subset
0	F	F	F	[]
1	F	F	T	3
2	F	T	F	2
3	F	T	T	23
4	T	F	F	1
5	T	F	T	13
6	T	T	F	12
7	T	T	T	123

解法3：

```
1 class Solution {
2 public:
3     vector<vector<int>> subsets(vector<int> &S) {
4         vector<vector<int>> res;
5         sort(S.begin(), S.end());
6         int max = 1 << S.size();
7         for (int k = 0; k < max; ++k) {
8             vector<int> out = convertIntToSet(S, k);
9             res.push_back(out);
10        }
11        return res;
12    }
13    vector<int> convertIntToSet(vector<int> &S, int k) {
14        vector<int> sub;
15        int idx = 0;
16        for (int i = k; i > 0; i >>= 1) {
17            if ((i & 1) == 1) {
18                sub.push_back(S[idx]);
19            }
20            ++idx;
21        }
22        return sub;
23    }
24}
```

CPP

79. 词语搜索

Given a 2D board and a word, find if the word exists in the grid.

The word can be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once.

For example,

Given board =

```
[  
    ["ABCE"],  
    ["SFCS"],  
    ["ADEE"]  
]  
word = "ABCCED", -> returns true,  
word = "SEE", -> returns true,  
word = "ABCB", -> returns false.
```

这道题是典型的深度优先遍历DFS的应用，原二维数组就像是一个迷宫，可以上下左右四个方向行走，我们以二维数组中每一个数都作为起点和给定字符串做匹配，我们还需要一个和原数组等大小的visited数组，是bool型的，用来记录当前位置是否已经被访问过，因为题目要求一个cell只能被访问一次。如果二维数组board的当前字符和目标字符串word对应的字符相等，则对其上下左右四个邻字符分别调用DFS的递归函数，只要有一个返回true，那么就表示可以找到对应的字符串，否则就不能找到，具体看代码实现如下：

解法1：

```
1 | class Solution {  
2 | public:  
3 |     bool exist(vector<vector<char>>& board, string word) {  
4 |         if (board.empty() || board[0].empty()) return false;  
5 |         int m = board.size(), n = board[0].size();  
6 |         vector<vector<bool>> visited(m, vector<bool>(n, false));  
7 |         for (int i = 0; i < m; ++i) {  
8 |             for (int j = 0; j < n; ++j) {  
9 |                 if (search(board, word, 0, i, j, visited)) return true;  
10 |             }  
11 |         }  
12 |         return false;  
13 |     }  
14 |     bool search(vector<vector<char>>& board, string word, int idx, int i, int j,  
15 |     vector<vector<bool>>& visited) {  
16 |         if (idx == word.size()) return true;  
17 |         int m = board.size(), n = board[0].size();  
18 |         if (i < 0 || j < 0 || i >= m || j >= n || visited[i][j] || board[i][j] !=  
19 |         word[idx]) return false;  
20 |         visited[i][j] = true;  
21 |         bool res = search(board, word, idx + 1, i - 1, j, visited)  
22 |             || search(board, word, idx + 1, i + 1, j, visited)  
23 |             || search(board, word, idx + 1, i, j - 1, visited)  
24 |             || search(board, word, idx + 1, i, j + 1, visited);  
25 |         visited[i][j] = false;  
26 |         return res;  
    }  
};
```

CPP

我们还可以不用visited数组，直接对board数组进行修改，将其遍历过的位置改为井号，记得递归调用完后需要恢复之前的状态，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     bool exist(vector<vector<char>>& board, string word) {
4         if (board.empty() || board[0].empty()) return false;
5         int m = board.size(), n = board[0].size();
6         for (int i = 0; i < m; ++i) {
7             for (int j = 0; j < n; ++j) {
8                 if (search(board, word, 0, i, j)) return true;
9             }
10        }
11    return false;
12 }
13 bool search(vector<vector<char>>& board, string word, int idx, int i, int j) {
14     if (idx == word.size()) return true;
15     int m = board.size(), n = board[i].size();
16     if (i < 0 || j < 0 || i >= m || j >= n || board[i][j] != word[idx]) return false;
17     char c = board[i][j];
18     board[i][j] = '#';
19     bool res = search(board, word, idx + 1, i - 1, j)
20             || search(board, word, idx + 1, i + 1, j)
21             || search(board, word, idx + 1, i, j - 1)
22             || search(board, word, idx + 1, i, j + 1);
23     board[i][j] = c;
24     return res;
25 }
26 };
```

80. 有序数组中去除重复项之二

Follow up for "Remove Duplicates":
What if duplicates are allowed at most twice?

For example,
Given sorted array A = [1,1,1,2,2,3],

Your function should return length = 5, and A is now [1,1,2,2,3].

这道题是之前那道 Remove Duplicates from Sorted Array 有序数组中去除重复项 的延续，这里允许最多重复的次数是两次，那么我们就需要用一个变量count来记录还允许有几次重复，count初始化为1，如果出现过一次重复，则count递减1，那么下次再出现重复，快指针直接前进一步，如果这时候不是重复的，则count恢复1，由于整个数组是有序的，所以一旦出现不重复的数，则一定比这个数大，此数之后不会再有重复项。理清了上面的思路，则代码很好写了：

```

1 class Solution {
2 public:
3     int removeDuplicates(int A[], int n) {
4         if (n <= 2) return n;
5         int pre = 0, cur = 1, count = 1;
6         while (cur < n) {
7             if (A[pre] == A[cur] && count == 0) ++cur;
8             else {
9                 if (A[pre] == A[cur]) --count;
10                else count = 1;
11                A[++pre] = A[cur++];
12            }
13        }
14        return pre + 1;
15    }
16 };

```

81. 在旋转有序数组中搜索之二

Follow up for "Search in Rotated Sorted Array":

What if duplicates are allowed?

Would this affect the run-time complexity? How and why?

Write a function to determine if a given target is in the array.

这道是之前那道 Search in Rotated Sorted Array 在旋转有序数组中搜索 的延伸，现在数组中允许出现重复数字，这个也会影响我们选择哪半边继续搜索，由于之前那道题不存在相同值，我们在比较中间值和最右值时就完全符合之前所说的规律：如果中间的数小于最右边的数，则右半段是有序的，若中间数大于最右边数，则左半段是有序的。而如果可以有重复值，就会出现下面两种情况，[3 1 1] 和 [1 1 3 1]，对于这两种情况中间值等于最右值时，目标值3既可以在左边又可以在右边，那怎么办么，对于这种情况其实处理非常简单，只要把最右值向左一位即可继续循环，如果还相同则继续移，直到移到不同值为止，然后其他部分还采用 Search in Rotated Sorted Array 在旋转有序数组中搜索 中的方法，可以得到代码如下：

```

1 class Solution {
2 public:
3     bool search(int A[], int n, int target) {
4         if (n == 0) return false;
5         int left = 0, right = n - 1;
6         while (left <= right) {
7             int mid = (left + right) / 2;
8             if (A[mid] == target) return true;
9             else if (A[mid] < A[right]) {
10                 if (A[mid] < target && A[right] >= target) left = mid + 1;
11                 else right = mid - 1;
12             } else if (A[mid] > A[right]){
13                 if (A[left] <= target && A[mid] > target) right = mid - 1;
14                 else left = mid + 1;
15             } else --right;
16         }
17         return false;
18     }
19 };

```

82. 移除有序链表中的重复项之二

Given a sorted linked list, delete all nodes that have duplicate numbers, leaving only distinct numbers from the original list.

For example,
 Given 1->2->3->3->4->4->5, return 1->2->5.
 Given 1->1->1->2->3, return 2->3.

和之前那道 (<http://www.cnblogs.com/grandyang/p/4066453.html>) 不同的地方是这里要删掉所有的重复项，由于链表开头可能会有重复项，被删掉的话头指针会改变，而最终却还需要返回链表的头指针。所以需要定义一个新的节点，然后链上原链表，然后定义一个前驱指针和一个现指针，每当前驱指针指向新建的节点，现指针从下一个位置开始往下遍历，遇到相同的则继续往下，直到遇到不同项时，把前驱指针的next指向下面那个不同的元素。如果现指针遍历的第一个元素就不相同，则把前驱指针向下移一位。代码如下：

```
1 class Solution {
2 public:
3     ListNode *deleteDuplicates(ListNode *head) {
4         if (!head || !head->next) return head;
5
6         ListNode *start = new ListNode(0);
7         start->next = head;
8         ListNode *pre = start;
9         while (pre->next) {
10             ListNode *cur = pre->next;
11             while (cur->next && cur->next->val == cur->val) cur = cur->next;
12             if (cur != pre->next) pre->next = cur->next;
13             else pre = pre->next;
14         }
15         return start->next;
16     }
17 };
```

CPP

83. 移除有序链表中的重复项

Given a sorted linked list, delete all duplicates such that each element appear only once.

For example,
 Given 1->1->2, return 1->2.
 Given 1->1->2->3->3, return 1->2->3.

移除有序链表中的重复项需要定义个指针指向该链表的第一个元素，然后第一个元素和第二个元素比较，如果重复了，则删掉第二个元素，如果不重复，指针指向第二个元素。这样遍历完整个链表，则剩下的元素没有重复项。代码如下：

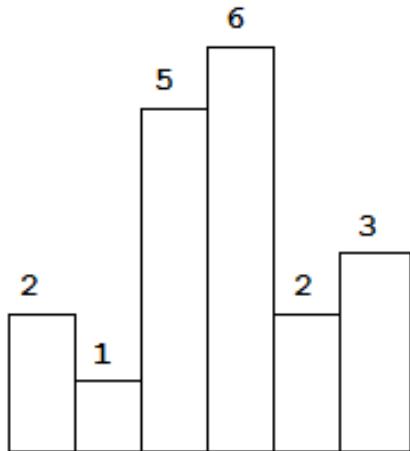
```

1 class Solution {
2 public:
3     ListNode *deleteDuplicates(ListNode *head) {
4         if (!head || !head->next) return head;
5
6         ListNode *start = head;
7         while (start && start->next) {
8             if (start->val == start->next->val) {
9                 ListNode *tmp = start->next;
10                start->next = start->next->next;
11                delete tmp;
12            } else start = start->next;
13        }
14        return head;
15    }
16 };

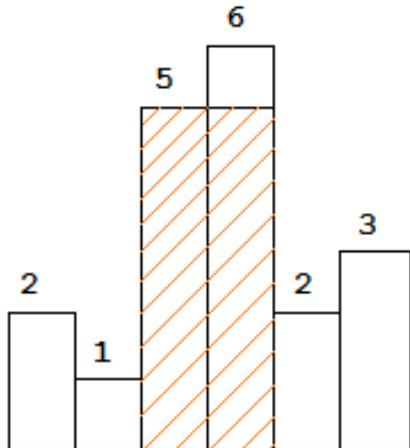
```

84. 直方图中最大的矩形

Given n non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest rectangle in the histogram.



Above is a histogram where width of each bar is 1, given height = [2,1,5,6,2,3].



The largest rectangle is shown in the shaded area, which has area = 10 unit.

```
For example,
Given height = [2,1,5,6,2,3],
return 10.
```

这道题让求直方图中最大的矩形，刚开始看到求极值问题以为要用DP来做，可是想不出递推式，只得作罢。这道题如果用暴力搜索法估计肯定没法通过OJ，但是我也没想出好的优化方法，在网上搜到了网友水中的鱼的博客，发现他想出了一种很好的优化方法，就是遍历数组，每找到一个局部峰值，然后向前遍历所有的值，算出共同的矩形面积，每次对比保留最大值，代码如下：

解法1：

```
1 class Solution {
2 public:
3     int largestRectangleArea(vector<int> &height) {
4         int res = 0;
5         for (int i = 0; i < height.size(); ++i) {
6             if (i + 1 < height.size() && height[i] <= height[i + 1]) {
7                 continue;
8             }
9             int minH = height[i];
10            for (int j = i; j >= 0; --j) {
11                minH = min(minH, height[j]);
12                int area = minH * (i - j + 1);
13                res = max(res, area);
14            }
15        }
16        return res;
17    }
18};
```

后来又在网上发现一种比较流行的解法，是利用栈来解，可参见网友实验室小纸贴校外版的博客，但是经过仔细研究，其核心思想跟上面那种剪枝的方法有异曲同工之妙，这里维护一个栈，用来保存递增序列，相当于上面那种方法的找局部峰值。我们可以看到，直方图矩形面积要最大的话，需要尽可能的使得连续的矩形多，并且最低一块的高度要高。有点像木桶原理一样，总是最低的那块板子决定桶的装水量。那么既然需要用单调栈来做，首先要考虑到底用递增栈，还是用递减栈来做。我们想啊，递增栈是维护递增的顺序，当遇到小于栈顶元素的数就开始处理，而递减栈正好相反，维护递减的顺序，当遇到大于栈顶元素的数开始处理。那么根据这道题的特点，我们需要按从高板子到低板子的顺序处理，先处理最高的板子，宽度为1，然后再处理旁边矮一些的板子，此时长度为2，因为之前的高板子可组成矮板子的矩形，因此我们需要一个递增栈，当遇到大的数字直接进栈，而当遇到小于栈顶元素的数字时，就要取出栈顶元素进行处理了，那取出的顺序就是从高板子到矮板子了，于是乎遇到的较小的数字只是一个触发，表示现在需要开始计算矩形面积了，为了使得最后一块板子也被处理，这里用了个小trick，在高度数组最后面加上一个0，这样原先的最后一个板子也可以被处理了。由于栈顶元素是矩形的高度，那么关键就是求出来宽度，那么跟之前那道 Trapping Rain Water一样，单调栈中不能放高度，而是需要放坐标。由于我们先取出栈中最高的板子，那么就可以先算出长度为1的矩形面积了，然后再取下一个板子，此时根据矮板子的高度算长度为2的矩形面积，以此类推，知道数字大于栈顶元素为止，再次进栈，巧妙的一比！关于单调栈问题可以参见博主的一篇总结帖LeetCode Monotonous Stack Summary 单调栈小结，代码如下：

解法2：

```

1 class Solution {
2 public:
3     int largestRectangleArea(vector<int> &height) {
4         int res = 0;
5         stack<int> st;
6         height.push_back(0);
7         for (int i = 0; i < height.size(); ++i) {
8             if (st.empty() || height[st.top()] < height[i]) {
9                 st.push(i);
10            } else {
11                int cur = st.top(); st.pop();
12                res = max(res, height[cur] * (st.empty() ? i : (i - st.top() - 1)));
13                --i;
14            }
15        }
16        return res;
17    }
18 };

```

我们可以将上面的方法稍作修改，使其更加简洁一些：

解法3：

```

1 class Solution {
2 public:
3     int largestRectangleArea(vector<int>& heights) {
4         int res = 0;
5         stack<int> st;
6         heights.push_back(0);
7         for (int i = 0; i < heights.size(); ++i) {
8             while (!st.empty() && heights[st.top()] >= heights[i]) {
9                 int cur = st.top(); st.pop();
10                res = max(res, heights[cur] * (st.empty() ? i : (i - st.top() - 1)));
11            }
12            st.push(i);
13        }
14        return res;
15    }
16 };

```

85. 最大矩形

Given a 2D binary matrix filled with 0's and 1's, find the largest rectangle containing all ones and return its area.

此题是之前那道的 Largest Rectangle in Histogram 直方图中最大的矩形 的扩展，这道题的二维矩阵每一层向上都可以看做一个直方图，输入矩阵有多少行，就可以形成多少个直方图，对每个直方图都调用 Largest Rectangle in Histogram 直方图中最大的矩形中的方法，就可以得到最大的矩形面积。那么这道题唯一要做的就是将每一层构成直方图，由于题目限定了输入矩阵的字符只有 '0' 和 '1' 两种，所以处理起来也相对简单。方法是，对于每一个点，如果是 '0'，则赋0，如果是 '1'，就赋之前的height值加上1。具体参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int maximalRectangle(vector<vector<char>> &matrix) {
4         int res = 0;
5         vector<int> height;
6         for (int i = 0; i < matrix.size(); ++i) {
7             height.resize(matrix[i].size());
8             for (int j = 0; j < matrix[i].size(); ++j) {
9                 height[j] = matrix[i][j] == '0' ? 0 : (1 + height[j]);
10            }
11            res = max(res, largestRectangleArea(height));
12        }
13        return res;
14    }
15    int largestRectangleArea(vector<int> &height) {
16        int res = 0;
17        stack<int> s;
18        height.push_back(0);
19        for (int i = 0; i < height.size(); ++i) {
20            if (s.empty() || height[s.top()] <= height[i]) s.push(i);
21            else {
22                int tmp = s.top();
23                s.pop();
24                res = max(res, height[tmp] * (s.empty() ? i : (i - s.top() - 1)));
25                --i;
26            }
27        }
28        return res;
29    }
30 };

```

我们也可以在一个函数内完成，这样代码看起来更加简洁一些：

解法2：

```

1 class Solution {
2 public:
3     int maximalRectangle(vector<vector<char>>& matrix) {
4         if (matrix.empty() || matrix[0].empty()) return 0;
5         int res = 0, m = matrix.size(), n = matrix[0].size();
6         vector<int> height(n + 1, 0);
7         for (int i = 0; i < m; ++i) {
8             stack<int> s;
9             for (int j = 0; j < n + 1; ++j) {
10                 if (j < n) {
11                     height[j] = matrix[i][j] == '1' ? height[j] + 1 : 0;
12                 }
13                 while (!s.empty() && height[s.top()] >= height[j]) {
14                     int cur = s.top(); s.pop();
15                     res = max(res, height[cur] * (s.empty() ? j : (j - s.top() - 1)));
16                 }
17                 s.push(j);
18             }
19         }
20         return res;
21     }
22 };

```

下面这种方法的思路很巧妙，height数组和上面一样，这里的left数组表示左边界是1的位置，right数组表示右边界是1的位置，那么对于任意一行的第j个位置，矩形为 $(right[j] - left[j]) * height[j]$ ，我们举个例子来说明，比如给定矩阵为：

```
[  
 [1, 1, 0, 0, 1],  
 [0, 1, 0, 0, 1],  
 [0, 0, 1, 1, 1],  
 [0, 0, 1, 1, 1],  
 [0, 0, 0, 0, 1]  
 ]
```

第0行：

```
h: 1 1 0 0 1  
l: 0 0 0 0 4  
r: 2 2 5 5 5
```

第1行：

```
h: 1 1 0 0 1  
l: 0 0 0 0 4  
r: 2 2 5 5 5
```

第2行：

```
h: 0 0 1 1 3  
l: 0 0 2 2 4  
r: 5 5 5 5 5
```

第3行：

```
h: 0 0 2 2 4  
l: 0 0 2 2 4  
r: 5 5 5 5 5
```

第4行：

```
h: 0 0 0 0 5  
l: 0 0 0 0 4  
r: 5 5 5 5 5
```

解法3：

```
1 class Solution {
2 public:
3     int maximalRectangle(vector<vector<char>>& matrix) {
4         if (matrix.empty() || matrix[0].empty()) return 0;
5         int res = 0, m = matrix.size(), n = matrix[0].size();
6         vector<int> height(n, 0), left(n, 0), right(n, n);
7         for (int i = 0; i < m; ++i) {
8             int cur_left = 0, cur_right = n;
9             for (int j = 0; j < n; ++j) {
10                 if (matrix[i][j] == '1') ++height[j];
11                 else height[j] = 0;
12             }
13             for (int j = 0; j < n; ++j) {
14                 if (matrix[i][j] == '1') left[j] = max(left[j], cur_left);
15                 else {left[j] = 0; cur_left = j + 1;}
16             }
17             for (int j = n - 1; j >= 0; --j) {
18                 if (matrix[i][j] == '1') right[j] = min(right[j], cur_right);
19                 else {right[j] = n; cur_right = j;}
20             }
21             for (int j = 0; j < n; ++j) {
22                 res = max(res, (right[j] - left[j]) * height[j]);
23             }
24         }
25     }
26     return res;
27 }
```

我们也可以通过合并一些for循环，使得运算速度更快一些：

解法4：

```

1 class Solution {
2 public:
3     int maximalRectangle(vector<vector<char>>& matrix) {
4         if (matrix.empty() || matrix[0].empty()) return 0;
5         int res = 0, m = matrix.size(), n = matrix[0].size();
6         vector<int> height(n, 0), left(n, 0), right(n, n);
7         for (int i = 0; i < m; ++i) {
8             int cur_left = 0, cur_right = n;
9             for (int j = 0; j < n; ++j) {
10                 if (matrix[i][j] == '1') {
11                     ++height[j];
12                     left[j] = max(left[j], cur_left);
13                 } else {
14                     height[j] = 0;
15                     left[j] = 0;
16                     cur_left = j + 1;
17                 }
18             }
19             for (int j = n - 1; j >= 0; --j) {
20                 if (matrix[i][j] == '1') {
21                     right[j] = min(right[j], cur_right);
22                 } else {
23                     right[j] = n;
24                     cur_right = j;
25                 }
26                 res = max(res, (right[j] - left[j]) * height[j]);
27             }
28         }
29         return res;
30     }
31 };

```

86. 划分链表

Given a linked list and a value x , partition it such that all nodes less than x come before nodes greater than or equal to x .

You should preserve the original relative order of the nodes in each of the two partitions.

For example,

Given $1 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 5 \rightarrow 2$ and $x = 3$,
return $1 \rightarrow 2 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 5$.

这道题要求我们划分链表，把所有小于给定值的节点都移到前面，大于该值的节点顺序不变，相当于一个局部排序的问题。那么可以想到的一种解法是首先找到第一个大于或等于给定值的节点，用题目中给的例子来说就是先找到4，然后再找小于3的值，每找到一个就将其取出置于4之前即可，代码如下：

解法1:

```

1 class Solution {
2 public:
3     ListNode *partition(ListNode *head, int x) {
4         ListNode *dummy = new ListNode(-1);
5         dummy->next = head;
6         ListNode *pre = dummy, *cur = head;;
7         while (pre->next && pre->next->val < x) pre = pre->next;
8         cur = pre;
9         while (cur->next) {
10             if (cur->next->val < x) {
11                 ListNode *tmp = cur->next;
12                 cur->next = tmp->next;
13                 tmp->next = pre->next;
14                 pre->next = tmp;
15                 pre = pre->next;
16             } else {
17                 cur = cur->next;
18             }
19         }
20         return dummy->next;
21     }
22 };

```

这种解法的链表变化顺序为：

1 -> 4 -> 3 -> 2 -> 5 -> 2

1 -> 2 -> 4 -> 3 -> 5 -> 2

1 -> 2 -> 2 -> 4 -> 3 -> 5

此题还有一种解法，就是将所有小于给定值的节点取出组成一个新的链表，此时原链表中剩余的节点的值都大于或等于给定值，只要将原链表直接接在新链表后即可，代码如下：

解法2：

```

1 class Solution {
2 public:
3     ListNode *partition(ListNode *head, int x) {
4         if (!head) return head;
5         ListNode *dummy = new ListNode(-1);
6         ListNode *newDummy = new ListNode(-1);
7         dummy->next = head;
8         ListNode *cur = dummy, *p = newDummy;
9         while (cur->next) {
10             if (cur->next->val < x) {
11                 p->next = cur->next;
12                 p = p->next;
13                 cur->next = cur->next->next;
14                 p->next = NULL;
15             } else {
16                 cur = cur->next;
17             }
18         }
19         p->next = dummy->next;
20         return newDummy->next;
21     }
22 };

```

此种解法链表变化顺序为：

Original: 1 -> 4 -> 3 -> 2 -> 5 -> 2

New:

Original: 4 -> 3 -> 2 -> 5 -> 2

New: 1

Original: 4 -> 3 -> 5 -> 2

New: 1 -> 2

Original: 4 -> 3 -> 5

New: 1 -> 2 -> 2

Original:

New: 1 -> 2 -> 2 -> 4 -> 3 -> 5

87. 爬行字符串

Given a string s_1 , we may represent it as a binary tree by partitioning it to two non-empty substrings recursively.

Below is one possible representation of $s_1 = "great"$:

```
      great
      /   \
    gr     eat
    / \   / \
  g   r   e   at
               / \
             a   t
```

To scramble the string, we may choose any non-leaf node and swap its two children.

For example, if we choose the node "gr" and swap its two children, it produces a scrambled string "rgeat".

```
      rgeat
      /   \
    rg     eat
    / \   / \
  r   g   e   at
               / \
             a   t
```

We say that "rgeat" is a scrambled string of "great".

Similarly, if we continue to swap the children of nodes "eat" and "at", it produces a scrambled string "rgtae".

```
      rgtae
      /   \
    rg     tae
    / \   / \
  r   g   ta   e
               / \
             t   a
```

We say that "rgtae" is a scrambled string of "great".

Given two strings s_1 and s_2 of the same length, determine if s_2 is a scrambled string of s_1 .

这道题定义了一种爬行字符串，就是说假如把一个字符串当做一个二叉树的根，然后它的非空子字符串是它的子节点，然后交换某个子字符串的两个子节点，重新爬行回去形成一个新的字符串，这个新字符串和原来的字符串互为爬行字符串。这道题可以用递归Recursion或是动态规划Dynamic Programming来做，我们先来看递归的解法，参见网友uniEagle的博客，简单的说，就是 s_1 和 s_2 是scramble的话，那么必然存在一个在 s_1 上的长度 l_1 ，将 s_1 分成 s_{11} 和 s_{12} 两段，同样有 s_{21} 和 s_{22} 。要么 s_{11} 和 s_{21} 是scramble的并且 s_{12} 和 s_{22} 是scramble的；要么 s_{11} 和 s_{22} 是scramble的并且 s_{12} 和 s_{21} 是scramble的。就拿题目中的例子 rgeat 和 great 来说，rgeat 可分成 rg 和 eat 两段，great 可分成 gr 和 eat 两段，rg 和 gr 是scrambled的，eat 和 eat 当然是scrambled。根据这点，我们可以写出代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool isScramble(string s1, string s2) {
4         if (s1.size() != s2.size()) return false;
5         if (s1 == s2) return true;
6         string str1 = s1, str2 = s2;
7         sort(str1.begin(), str1.end());
8         sort(str2.begin(), str2.end());
9         if (str1 != str2) return false;
10        for (int i = 1; i < s1.size(); ++i) {
11            string s11 = s1.substr(0, i);
12            string s12 = s1.substr(i);
13            string s21 = s2.substr(0, i);
14            string s22 = s2.substr(i);
15            if (isScramble(s11, s21) && isScramble(s12, s22)) return true;
16            s21 = s2.substr(s1.size() - i);
17            s22 = s2.substr(0, s1.size() - i);
18            if (isScramble(s11, s21) && isScramble(s12, s22)) return true;
19        }
20        return false;
21    }
22 };

```

当然，这道题也可以用动态规划Dynamic Programming，根据以往的经验来说，跟字符串有关的题十有八九可以用DP来做，那么难点就在于如何找出递推公式。参见网友Code Ganker的博客，这其实是一道三维动态规划的题目，我们提出维护量res[i][j][n]，其中i是s1的起始字符，j是s2的起始字符，而n是当前的字符串长度，res[i][j][len]表示的是以i和j分别为s1和s2起点的长度为len的字符串是不是互为scramble。

有了维护量我们接下来看看递推式，也就是怎么根据历史信息来得到res[i][j][len]。判断这个是不是满足，其实我们首先是把当前s1[i...i+len-1]字符串劈一刀分成两部分，然后分两种情况：第一种是左边和s2[j...j+len-1]左边部分是不是scramble，以及右边和s2[j...j+len-1]右边部分是不是scramble；第二种情况是左边和s2[j...j+len-1]右边部分是不是scramble，以及右边和s2[j...j+len-1]左边部分是不是scramble。如果以上两种情况有一种成立，说明s1[i...i+len-1]和s2[j...j+len-1]是scramble的。而对于判断这些左右部分是不是scramble我们是有历史信息的，因为长度小于n的所有情况我们都在前面求解过了（也就是长度是最外层循环）。

上面说的是劈一刀的情况，对于s1[i...i+len-1]我们有len-1种劈法，在这些劈法中只要有一种成立，那么两个串就是scramble的。

总结起来递推式是res[i][j][len] = || (res[i][j][k]&&res[i+k][j+k][len-k] || res[i][j+len-k][k]&&res[i+k][j][len-k]) 对于所有 $1 \leq k < len$ ，也就是对于所有len-1种劈法的结果求或运算。因为信息都是计算过的，对于每种劈法只需要常量操作即可完成，因此求解递推式是需要O(len)（因为len-1种劈法）。

如此总时间复杂度因为是三维动态规划，需要三层循环，加上每一步需要线行时间求解递推式，所以是O(n^4)。虽然已经比较高了，但是至少不是指数量级的，动态规划还是有很大优势的，空间复杂度是O(n^3)。代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool isScramble(string s1, string s2) {
4         if (s1.size() != s2.size()) return false;
5         if (s1 == s2) return true;
6         int n = s1.size();
7         vector<vector<vector<bool>> dp(n, vector<vector<bool>>(n, vector<bool>(n + 1,
8 false)));
9         for (int i = 0; i < n; ++i) {
10             for (int j = 0; j < n; ++j) {
11                 dp[i][j][1] = s1[i] == s2[j];
12             }
13         }
14         for (int len = 2; len <= n; ++len) {
15             for (int i = 0; i <= n - len; ++i) {
16                 for (int j = 0; j <= n - len; ++j) {
17                     for (int k = 1; k < len; ++k) {
18                         if ((dp[i][j][k] && dp[i + k][j + k][len - k]) || (dp[i + k][j][len -
19 - k] && dp[i + len - k][k])) {
20                             dp[i][j][len] = true;
21                         }
22                     }
23                 }
24             }
25         }
26         return dp[0][0][n];
27     }
28 };

```

上面的代码的实现过程如下，首先按单个字符比较，判断它们之间是否是scrambled的。在更新第二个表中第一个值(gr和rg是否为scrambled的)时，比较了第一个表中的两种构成，一种是g与r, r与g，另一种是g与g, r与r，其中后者是真，只要其中一个为真，则将该值赋真。其实这个原理和之前递归的原理很像，在判断某两个字符串是否为scrambled时，比较它们所有可能的拆分方法的子字符串是否是scrambled的，只要有一个种拆分方法为真，则比较的两个字符串一定是scrambled的。比较 rge 和 gre 的实现过程如下所示：

	r	g	e
g	x	✓	x
r	✓	x	x
e	x	x	✓

	rg	ge
gr	✓	x
re	x	x

	rge
gre	✓

DP的另一种写法，参考网友加载中..的博客，思路都一样，代码如下：

解法3：

```

1 class Solution {
2 public:
3     bool isScramble(string s1, string s2) {
4         if (s1.size() != s2.size()) return false;
5         if (s1 == s2) return true;
6         int n = s1.size();
7         vector<vector<vector<bool>> dp (n, vector<vector<bool>>(n, vector<bool>(n + 1,
8 false)));
9         for (int i = n - 1; i >= 0; --i) {
10             for (int j = n - 1; j >= 0; --j) {
11                 for (int k = 1; k <= n - max(i, j); ++k) {
12                     if (s1.substr(i, k) == s2.substr(j, k)) {
13                         dp[i][j][k] = true;
14                     } else {
15                         for (int t = 1; t < k; ++t) {
16                             if ((dp[i][j][t] && dp[i + t][j + t][k - t]) || (dp[i][j + k - t][t] && dp[i + t][j][k - t])) {
17                                 dp[i][j][k] = true;
18                                 break;
19                             }
20                         }
21                     }
22                 }
23             }
24         }
25     }
26     return dp[0][0][n];
27 }
28 };

```

下面这种解法和第一个解法思路相同，只不过没有用排序算法，而是采用了类似于求异构词的方法，用一个数组来保存每个字母出现的次数，后面判断Scramble字符串的方法和之前的没有区别：

解法4：

```

1 class Solution {
2 public:
3     bool isScramble(string s1, string s2) {
4         if (s1 == s2) return true;
5         if (s1.size() != s2.size()) return false;
6         int n = s1.size(), m[26] = {0};
7         for (int i = 0; i < n; ++i) {
8             ++m[s1[i] - 'a'];
9             --m[s2[i] - 'a'];
10        }
11        for (int i = 0; i < 26; ++i) {
12            if (m[i] != 0) return false;
13        }
14        for (int i = 1; i < n; ++i) {
15            if ((isScramble(s1.substr(0, i), s2.substr(0, i)) && isScramble(s1.substr(i),
16 s2.substr(i))) || (isScramble(s1.substr(0, i), s2.substr(n - i)) &&
17 isScramble(s1.substr(i), s2.substr(0, n - i)))) {
18                return true;
19            }
20        }
21    return false;
22 }
23 };

```

下面这种解法实际上是解法二的递归形式，我们用了memo数组来减少了大量的运算，注意这里的memo数组一定要有三种状态，初始化为-1，区域内为scramble是1，不是scramble是0，这样就避免了已经算过了某个区间，但由于不是scramble，从而又进行一次计算，从而会TLE，感谢网友bamboo提供的思路，参见代码如下：

解法5：

```

1 class Solution {
2 public:
3     bool isScramble(string s1, string s2) {
4         if (s1 == s2) return true;
5         if (s1.size() != s2.size()) return false;
6         int n = s1.size();
7         vector<vector<vector<int>>> memo(n, vector<vector<int>>(n, vector<int>(n + 1,
8 -1)));
9         return helper(s1, s2, 0, 0, n, memo);
10    }
11    bool helper(string& s1, string& s2, int idx1, int idx2, int len,
12    vector<vector<vector<int>>& memo) {
13        if (len == 0) return true;
14        if (len == 1) memo[idx1][idx2][len] = s1[idx1] == s2[idx2];
15        if (memo[idx1][idx2][len] != -1) return memo[idx1][idx2][len];
16        for (int k = 1; k < len; ++k) {
17            if ((helper(s1, s2, idx1, idx2, k, memo) && helper(s1, s2, idx1 + k, idx2 + k,
18 len - k, memo)) || (helper(s1, s2, idx1, idx2 + len - k, k, memo) && helper(s1, s2, idx1 +
19 k, idx2, len - k, memo))) {
20                return memo[idx1][idx2][len] = 1;
21            }
22        }
23        return memo[idx1][idx2][len] = 0;
24    }
25 };

```

88. 混合插入有序数组

Given two sorted integer arrays A and B, merge B into A as one sorted array.

Note:

You may assume that A has enough space (size that is greater or equal to m + n) to hold additional elements from B. The number of elements initialized in A and B are m and n respectively.

混合插入有序数组，由于两个数组都是有序的，所有只要按顺序比较大小即可。最先想到的方法是建立一个m+n大小的新数组，然后逐个从A和B数组中取出元素比较，把较小的加入新数组，然后在考虑A数组有剩余和B数组有剩余的两种情况，最后在把新数组的元素重新赋值到A数组中即可。代码如下：

解法1：

```

1 class Solution {
2 public:
3     void merge(int A[], int m, int B[], int n) {
4         if (m <= 0 && n <= 0) return;
5         int a = 0, b = 0;
6         int C[m + n];
7         for (int i = 0; i < m + n; ++i) {
8             if (a < m && b < n) {
9                 if (A[a] < B[b]) {
10                     C[i] = A[a];
11                     ++a;
12                 }
13             else {
14                 C[i] = B[b];
15                 ++b;
16             }
17         }
18         else if (a < m && b >= n) {
19             C[i] = A[a];
20             ++a;
21         }
22         else if (a >= m && b < n) {
23             C[i] = B[b];
24             ++b;
25         }
26         else return;
27     }
28     for (int i = 0; i < m + n; ++i) A[i] = C[i];
29 }
30 };

```

这样固然没错，但是还有更简洁的方法，而且不用申请新变量。算法思想是：由于合并后A数组的大小必定是 $m+n$ ，所以从最后面开始往前赋值，先比较A和B中最后一个元素的大小，把较大的那个插入到 $m+n-1$ 的位置上，再依次向前推。如果A中所有的元素都比B小，那么前 m 个还是A原来的内容，没有改变。如果A中的数组比B大的，当A循环完了，B中还有元素没加入A，直接用一个循环把B中所有的元素覆盖到A剩下的位置。代码如下：

解法2：

```

1 class Solution {
2 public:
3     void merge(int A[], int m, int B[], int n) {
4         int count = m + n - 1;
5         --m; --n;
6         while (m >= 0 && n >= 0) A[count--] = A[m] > B[n] ? A[m--] : B[n--];
7         while (n >= 0) A[count--] = B[n--];
8     }
9 };

```

89. 格雷码

The gray code is a binary numeral system where two successive values differ in only one bit.

Given a non-negative integer n representing the total number of bits in the code, print the sequence of gray code. A gray code sequence must begin with 0.

For example, given $n = 2$, return [0,1,3,2]. Its gray code sequence is:

```
00 - 0
01 - 1
11 - 3
10 - 2
```

Note:

For a given n , a gray code sequence is not uniquely defined.

For example, [0,2,3,1] is also a valid gray code sequence according to the above definition.

For now, the judge is able to judge based on one instance of gray code sequence. Sorry about that.

这道题是关于格雷码的，猛地一看感觉完全没接触过格雷码，但是看了维基百科后，隐约的感觉原来好像哪门课提到过，哎全还给老师了。这道题如果不了解格雷码，还真不太好做，幸亏脑补了维基百科，上面说格雷码是一种循环二进制单位距离码，主要特点是两个相邻数的代码只有一位二进制数不同的编码，格雷码的处理主要是位操作 Bit Operation，LeetCode 中关于位操作的题也挺常见，比如 Repeated DNA Sequences 求重复的DNA序列，Single Number 单独的数字，和 Single Number II 单独的数字之二 等等。三位的格雷码和二进制数如下：

Int	Grey Code	Binary
0	000	000
1	001	001
2	011	010
3	010	011
4	110	100
5	111	101
6	101	110
7	100	111

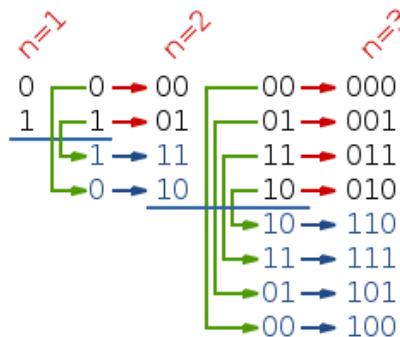
其实这道题还有多种解法。首先来看一种最简单的，是用到格雷码和二进制数之间的相互转化，可参见我之前的博客 Conversion of grey code and binary 格雷码和二进制数之间的转换，明白了转换方法后，这道题完全没有难度，代码如下：

解法1：

```
1 class Solution {
2 public:
3     vector<int> grayCode(int n) {
4         vector<int> res;
5         for (int i = 0; i < pow(2,n); ++i) {
6             res.push_back((i >> 1) ^ i);
7         }
8         return res;
9     }
10 };
```

CPP

然后我们来看看其他的解法，参考维基百科上关于格雷码的性质，有一条是说镜面排列的， n 位元的格雷码可以从 $n-1$ 位元的格雷码以上下镜射后加上新位元的方式快速的得到，如下图所示一般。



有了这条性质，我们很容易的写出代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<int> grayCode(int n) {
4         vector<int> res{0};
5         for (int i = 0; i < n; ++i) {
6             int size = res.size();
7             for (int j = size - 1; j >= 0; --j) {
8                 res.push_back(res[j] | (1 << i));
9             }
10        }
11        return res;
12    }
13 };

```

CPP

维基百科上还有一条格雷码的性质是直接排列，以二进制为0值的格雷码为第零项，第一项改变最右边的位元，第二项改变右起第一个为1的位元的左边位元，第三、四项方法同第一、二项，如此反复，即可排列出n个位元的格雷码。根据这条性质也可以写出代码，不过相比前面的略微复杂，代码如下：

```

0 0 0
0 0 1
0 1 1
0 1 0
1 1 0
1 1 1
1 0 1
1 0 0

```

解法3：

```

1 class Solution {
2 public:
3     vector<int> grayCode(int n) {
4         vector<int> res{0};
5         int len = pow(2, n);
6         for (int i = 1; i < len; ++i) {
7             int pre = res.back();
8             if (i % 2 == 1) {
9                 pre = (pre & (len - 2)) | ((~pre) & 1);
10            } else {
11                int cnt = 1, t = pre;
12                while ((t & 1) != 1) {
13                    ++cnt;
14                    t >>= 1;
15                }
16                if ((pre & (1 << cnt)) == 0) pre |= (1 << cnt);
17                else pre &= ~(1 << cnt);
18            }
19            res.push_back(pre);
20        }
21        return res;
22    }
23 };

```

上面三种解法都需要事先了解格雷码及其性质，假如我们之前并没有接触过格雷码，那么我们其实也可以用比较笨的方法来找出结果，比如下面这种方法用到了一个set来保存已经产生的结果，我们从0开始，遍历其二进制每一位，对其取反，然后看其是否在set中出现过，如果没有，我们将其加入set和结果res中，然后再对这个数的每一位进行遍历，以此类推就可以找出所有的格雷码了，参见代码如下：

解法4：

```

1 class Solution {
2 public:
3     vector<int> grayCode(int n) {
4         vector<int> res;
5         unordered_set<int> s;
6         helper(n, s, 0, res);
7         return res;
8     }
9     void helper(int n, unordered_set<int>& s, int out, vector<int>& res) {
10        if (!s.count(out)) {
11            s.insert(out);
12            res.push_back(out);
13        }
14        for (int i = 0; i < n; ++i) {
15            int t = out;
16            if ((t & (1 << i)) == 0) t |= (1 << i);
17            else t &= ~(1 << i);
18            if (s.count(t)) continue;
19            helper(n, s, t, res);
20            break;
21        }
22    }
23 };

```

既然递归方法可以实现，那么就有对应的迭代的写法，当然需要用stack来辅助，参见代码如下：

解法5：

```

1 class Solution {
2 public:
3     vector<int> grayCode(int n) {
4         vector<int> res{0};
5         unordered_set<int> s;
6         stack<int> st;
7         st.push(0);
8         s.insert(0);
9         while (!st.empty()) {
10             int t = st.top(); st.pop();
11             for (int i = 0; i < n; ++i) {
12                 int k = t;
13                 if ((k & (1 << i)) == 0) k |= (1 << i);
14                 else k &= ~(1 << i);
15                 if (s.count(k)) continue;
16                 s.insert(k);
17                 st.push(k);
18                 res.push_back(k);
19                 break;
20             }
21         }
22         return res;
23     }
24 }

```

90. 子集合之二

Given a collection of integers that might contain duplicates, S, return all possible subsets.

Note:

Elements in a subset must be in non-descending order.
The solution set must not contain duplicate subsets.

For example,
If S = [1,2,2], a solution is:

```
[
[2],
[1],
[1,2,2],
[2,2],
[1,2],
[]]
```

这道子集合之二是之前那道 Subsets 子集合的延伸，这次输入数组允许有重复项，其他条件都不变，只需要在之前那道题解法的基础上稍加改动便可以做出来，我们先来看非递归解法，拿题目中的例子[1 2 2]来分析，根据之前 Subsets 子集合里的分析可知，当处理到第一个2时，此时的子集合为[], [1], [2], [1, 2]，而这时再处理第二个2时，如果在[]和[1]后直接加2会产生重复，所以只能在上一个循环生成的后两个子集合后面加2，发现了这一点，题目就可以做了，我们用last来记录上一个处理的数字，然后判定当前的数字和上面的是否相同，若不同，则循环还是从0到当前子集的个数，若相同，则新子集个数减去之前循环时子集的个数当做起点来循环，这样就不会产生重复了，代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<vector<int>> subsetsWithDup(vector<int> &S) {
4         if (S.empty()) return {};
5         vector<vector<int>> res(1);
6         sort(S.begin(), S.end());
7         int size = 1, last = S[0];
8         for (int i = 0; i < S.size(); ++i) {
9             if (last != S[i]) {
10                 last = S[i];
11                 size = res.size();
12             }
13             int newSize = res.size();
14             for (int j = newSize - size; j < newSize; ++j) {
15                 res.push_back(res[j]);
16                 res.back().push_back(S[i]);
17             }
18         }
19         return res;
20     }
21 };

```

CPP

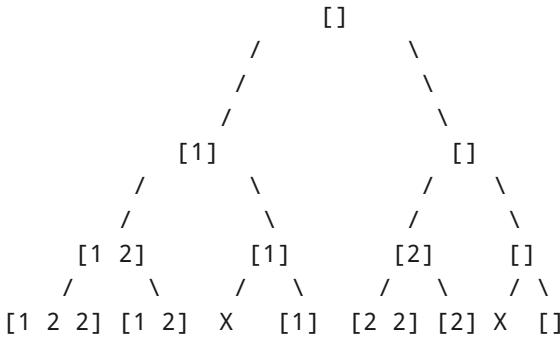
整个添加的顺序为：

```

[]
[1]
[2]
[1 2]
[2 2]
[1 2 2]

```

对于递归的解法，根据之前 Subsets 子集合 里的构建树的方法，在处理到第二个2时，由于前面已经处理了一次2，这次我们只在添加过2的[2] 和 [1 2]后面添加2，其他的都不添加，那么这样构成的二叉树如下图所示：



代码只需在原有的基础上增加一句话，`while (S[i] == S[i + 1]) ++i;` 这句话的作用是跳过树中为X的叶节点，因为它们是重复的子集，应被抛弃。代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<vector<int>> subsetsWithDup(vector<int> &S) {
4         if (S.empty()) return {};
5         vector<vector<int>> res;
6         vector<int> out;
7         sort(S.begin(), S.end());
8         getSubsets(S, 0, out, res);
9         return res;
10    }
11    void getSubsets(vector<int> &S, int pos, vector<int> &out, vector<vector<int>> &res) {
12        res.push_back(out);
13        for (int i = pos; i < S.size(); ++i) {
14            out.push_back(S[i]);
15            getSubsets(S, i + 1, out, res);
16            out.pop_back();
17            while (i + 1 < S.size() && S[i] == S[i + 1]) ++i;
18        }
19    }
20 };

```

整个添加的顺序为：

```

[]
[1]
[1 2]
[1 2 2]
[2]
[2 2]

```

91. 解码方法

A message containing letters from A-Z is being encoded to numbers using the following mapping:

```

'A' -> 1
'B' -> 2
...
'Z' -> 26

```

Given an encoded message containing digits, determine the total number of ways to decode it.

For example,

Given encoded message "12", it could be decoded as "AB" (1 2) or "L" (12).

The number of ways decoding "12" is 2.

这道题要求解码方法，跟之前那道 Climbing Stairs 爬梯子问题非常的相似，但是还有一些其他的限制条件，比如说一位数时不能为0，两位数不能大于26，其十位上的数也不能为0，出去这些限制条件，跟爬梯子基本没啥区别，也勉强算特殊的斐波那契数列，当然需要用动态规划Dynamci Programming来解。建立一位dp数组，长度比输入数组长多多2，全部初始化为1，因为斐波那契数列的前两项也为1，然后从第三个数开始更新，对应数组的第一个数。对每个数组首先判断其是否为0，若是将改为dp赋0，若不是，赋上一个dp值，此时相当如加上了dp[i - 1]，然后看数组前一位是否存在，如果存在且满足前一位不是0，且和当前为一起组成的两位数不大于26，则当前dp值加上dp[i - 2]，至此可以看出来跟斐波那契数组的递推式一样，代码如下：

解法1：

```

1 class Solution {
2 public:
3     int numDecodings(string s) {
4         if (s.empty() || (s.size() > 1 && s[0] == '0')) return 0;
5         vector<int> dp(s.size() + 1, 0);
6         dp[0] = 1;
7         for (int i = 1; i < dp.size(); ++i) {
8             dp[i] = (s[i - 1] == '0') ? 0 : dp[i - 1];
9             if (i > 1 && (s[i - 2] == '1' || (s[i - 2] == '2' && s[i - 1] <= '6'))) {
10                 dp[i] += dp[i - 2];
11             }
12         }
13     return dp.back();
14 }
15 };

```

下面这种方法跟上面的方法的思路一样，只是写法略有不同：

解法2：

```

1 class Solution {
2 public:
3     int numDecodings(string s) {
4         if (s.empty()) return 0;
5         vector<int> dp(s.size() + 1, 0);
6         dp[0] = 1;
7         for (int i = 1; i < dp.size(); ++i) {
8             if (s[i - 1] != '0') dp[i] += dp[i - 1];
9             if (i >= 2 && s.substr(i - 2, 2) <= "26" && s.substr(i - 2, 2) >= "10") {
10                 dp[i] += dp[i - 2];
11             }
12         }
13     return dp.back();
14 }
15 };

```

我们再来看一种空间复杂度为O(1)的解法，我们用两个变量c1, c2来分别表示s[i-1]和s[i-2]的解码方法，然后我们从i=1开始遍历，也就是字符串的第二个字符，我们判断如果当前字符为'0'，说明当前字符不能单独拆分出来，只能和前一个字符一起，我们先将c1赋为0，然后我们看前面的字符，如果前面的字符是1或者2时，我们就可以更新c1 = c1 + c2，然后c2 = c1 - c2，其实c2赋值为之前的c1，如果不满足这些条件的话，那么c2 = c1，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int numDecodings(string s) {
4         if (s.empty() || s.front() == '0') return 0;
5         int c1 = 1, c2 = 1;
6         for (int i = 1; i < s.size(); ++i) {
7             if (s[i] == '0') c1 = 0;
8             if (s[i - 1] == '1' || (s[i - 1] == '2' && s[i] <= '6')) {
9                 c1 = c1 + c2;
10                c2 = c1 - c2;
11            } else {
12                c2 = c1;
13            }
14        }
15        return c1;
16    }
17 };

```

92. 倒置链表之二

Reverse a linked list from position m to n. Do it in-place and in one-pass.

For example:

Given 1->2->3->4->5->NULL, m = 2 and n = 4,

return 1->4->3->2->5->NULL.

Note:

Given m, n satisfy the following condition:

$1 \leq m \leq n \leq \text{length of list}$.

很奇怪为何没有倒置链表之一，就来了这个倒置链表之二，不过猜也能猜得到之一就是单纯的倒置整个链表，而这道作为延伸的地方就是倒置其中的某一小段。对于链表的问题，根据以往的经验一般都是要建一个dummy node，连上原链表的头结点，这样的话就算头结点变动了，我们还可以通过dummy→next来获得新链表的头结点。这道题的要求是只通过一次遍历完成，就拿题目中的例子来说，变换的是2,3,4这三个点，那么我们可以先取出2，用front指针指向2，然后当取出3的时候，我们把3加到2的前面，把front指针前移到3，依次类推，到4后停止，这样我们得到一个新链表4→3→2, front指针指向4。对于原链表连说，有两个点的位置很重要，需要用指针记录下来，分别是1和5，因为当2,3,4被取走时，原链表就变成了1→5→NULL，要把新链表插入的时候需要这两个点的位置。1的位置很好找，因为知道m的值，我们用pre指针记录1的位置，5的位置最后才能记录，当4结点被取走后，5的位置需要记下来，这样我们就可以把倒置后的那一小段链表加入到原链表中。代码如下：

```

1 class Solution {
2 public:
3     ListNode *reverseBetween(ListNode *head, int m, int n) {
4         ListNode *dummy = new ListNode(-1);
5         dummy->next = head;
6         ListNode *cur = dummy;
7         ListNode *pre, *front, *last;
8         for (int i = 1; i <= m - 1; ++i) cur = cur->next;
9         pre = cur;
10        last = cur->next;
11        for (int i = m; i <= n; ++i) {
12            cur = pre->next;
13            pre->next = cur->next;
14            cur->next = front;
15            front = cur;
16        }
17        cur = pre->next;
18        pre->next = front;
19        last->next = cur;
20        return dummy->next;
21    }
22 };

```

93. 复原IP地址

Given a string containing only digits, restore it by returning all possible valid IP address combinations.

For example:

Given "25525511135",

return ["255.255.11.135", "255.255.111.35"]. (Order does not matter)

这道题要求是复原IP地址，IP地址对我们并不陌生，就算我们不是学CS的，只要我们是广大网友之一，就应该对其并不陌生。IP地址由32位二进制数组成，为便于使用，常以XXX.XXX.XXX.XXX形式表现，每组XXX代表小于或等于255的10进制数。所以说IP地址总共有四段，每一段可能有一位，两位或者三位，范围是[0, 255]，题目明确指出输入字符串只含有数字，所以当某段是三位时，我们要判断其是否越界 (>255)，还有一点很重要的是，当只有一位时，0可以成某一段，如果有两位或三位时，像00, 01, 001, 011, 000等都是不合法的，所以我们还是需要有一个判定函数来判断某个字符串是否合法。这道题其实也可以看做是字符串的分段问题，在输入字符串中加入三个点，将字符串分为四段，每一段必须合法，求所有可能的情况。根据目前刷了这么多题，得出了两个经验，一是只要遇到字符串的子序列或配准问题首先考虑动态规划DP，二是只要遇到需要求出所有可能情况首先考虑用递归。这道题并非是求字符串的子序列或配准问题，更符合第二种情况，所以我们要用递归来解。我们用k来表示当前还需要分的段数，如果k = 0，则表示三个点已经加入完成，四段已经形成，若这时字符串刚好为空，则将当前分好的结果保存。若k != 0，则对于每一段，我们分别用一位，两位，三位来尝试，分别判断其合不合法，如果合法，则调用递归继续分剩下的字符串，最终和求出所有合法组合，代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<string> restoreIpAddresses(string s) {
4         vector<string> res;
5         restore(s, 4, "", res);
6         return res;
7     }
8     void restore(string s, int k, string out, vector<string> &res) {
9         if (k == 0) {
10             if (s.empty()) res.push_back(out);
11         }
12         else {
13             for (int i = 1; i <= 3; ++i) {
14                 if (s.size() >= i && isValid(s.substr(0, i))) {
15                     if (k == 1) restore(s.substr(i), k - 1, out + s.substr(0, i), res);
16                     else restore(s.substr(i), k - 1, out + s.substr(0, i) + ".", res);
17                 }
18             }
19         }
20     }
21     bool isValid(string s) {
22         if (s.empty() || s.size() > 3 || (s.size() > 1 && s[0] == '0')) return false;
23         int res = atoi(s.c_str());
24         return res <= 255 && res >= 0;
25     }
26 };

```

我们也可以省掉isValid函数，直接在调用递归之前用if语句来滤掉不符合题意的情况，这里面用了`k != std::to_string(val).size()`，其实并不难理解，比如当`k=3`时，说明应该是个三位数，但如果字符串是"010"，那么转为整型`val=10`，再转回字符串就是"10"，此时的长度和`k`值不同了，这样就可以找出不合要求的情况了，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<string> restoreIpAddresses(string s) {
4         vector<string> res;
5         helper(s, 0, "", res);
6         return res;
7     }
8     void helper(string s, int n, string out, vector<string>& res) {
9         if (n == 4) {
10             if (s.empty()) res.push_back(out);
11         } else {
12             for (int k = 1; k < 4; ++k) {
13                 if (s.size() < k) break;
14                 int val = atoi(s.substr(0, k).c_str());
15                 if (val > 255 || k != std::to_string(val).size()) continue;
16                 helper(s.substr(k), n + 1, out + s.substr(0, k) + (n == 3 ? "" : "."), res);
17             }
18         }
19     }
20 };

```

由于每段数字最多只能有三位，而且只能分为四段，所以情况并不是很多，我们可以使用暴力搜索的方法，每一段都循环1到3，然后当4段位数之和等于原字符串长度时，我们进一步判断每段数字是否不大于255，然后滤去不合要求的数字，加入结果中即可，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     vector<string> restoreIpAddresses(string s) {
4         vector<string> res;
5         for (int a = 1; a < 4; ++a)
6             for (int b = 1; b < 4; ++b)
7                 for (int c = 1; c < 4; ++c)
8                     for (int d = 1; d < 4; ++d)
9                         if (a + b + c + d == s.size()) {
10                             int A = stoi(s.substr(0, a));
11                             int B = stoi(s.substr(a, b));
12                             int C = stoi(s.substr(a + b, c));
13                             int D = stoi(s.substr(a + b + c, d));
14                             if (A <= 255 && B <= 255 && C <= 255 && D <= 255) {
15                                 string t = to_string(A) + "." + to_string(B) + "." + to_string(C) + "."
16                                 + to_string(D);
17                                 if (t.size() == s.size() + 3) res.push_back(t);
18                             }
19                         }
20         return res;
21     }
22 };

```

94. 二叉树的中序遍历

Given a binary tree, return the inorder traversal of its nodes' values.

For example:

Given binary tree {1,#,2,3},

```

1
 \
 2
 /
3

```

return [1,3,2].

Note: Recursive solution is trivial, could you do it iteratively?

confused what "{1,#,2,3}" means? > read more on how binary tree is serialized on OJ.

二叉树的中序遍历顺序为左-根-右，可以有递归和非递归来解，其中非递归解法又分为两种，一种是使用栈来接，另一种不需要使用栈。我们先来看递归方法，十分直接，对左子结点调用递归函数，根节点访问值，右子节点再调用递归函数，代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<int> inorderTraversal(TreeNode *root) {
4         vector<int> res;
5         inorder(root, res);
6         return res;
7     }
8     void inorder(TreeNode *root, vector<int> &res) {
9         if (!root) return;
10        if (root->left) inorder(root->left, res);
11        res.push_back(root->val);
12        if (root->right) inorder(root->right, res);
13    }
14 };

```

下面我们再来看非递归使用栈的解法，也是符合本题要求使用的解法之一，需要用栈来做，思路是从根节点开始，先将根节点压入栈，然后再将其所有左子结点压入栈，然后取出栈顶节点，保存节点值，再将当前指针移到其右子节点上，若存在右子节点，则在下次循环时又可将其所有左子结点压入栈中。这样就保证了访问顺序为左-根-右，代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<int> inorderTraversal(TreeNode *root) {
4         vector<int> res;
5         stack<TreeNode*> s;
6         TreeNode *p = root;
7         while (p || !s.empty()) {
8             while (p) {
9                 s.push(p);
10                p = p->left;
11            }
12            p = s.top();
13            s.pop();
14            res.push_back(p->val);
15            p = p->right;
16        }
17        return res;
18    }
19 };

```

下面这种解法跟Binary Tree Preorder Traversal中的解法二几乎一样，就是把结点值加入结果res的步骤从if中移动到了else中，因为中序遍历的顺序是左-根-右，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     vector<int> inorderTraversal(TreeNode* root) {
4         vector<int> res;
5         stack<TreeNode*> s;
6         TreeNode *p = root;
7         while (!s.empty() || p) {
8             if (p) {
9                 s.push(p);
10                p = p->left;
11            } else {
12                TreeNode *t = s.top(); s.pop();
13                res.push_back(t->val);
14                p = t->right;
15            }
16        }
17        return res;
18    }
19 };

```

下面我们来看另一种很巧妙的解法，这种方法不需要使用栈，所以空间复杂度为常量，这种非递归不用栈的遍历方法有个专门的名字，叫Morris Traversal，在介绍这种方法之前，我们先来引入一种新型树，叫 Threaded binary tree，这个还不好翻译，我第一眼看上去以为是叫线程二叉树，但是感觉好像又跟线程没啥关系，后来看到网上有人翻译为螺纹二叉树，但本人认为这翻译也不太敢直视，很容易让人联想到为计划生育做出突出贡献的某世界著名品牌，但是苦于找不到更合理的翻译方法，就暂且叫螺纹二叉树吧。我们先来看看维基百科上关于它的英文定义：

A binary tree is threaded by making all right child pointers that would normally be null point to the inorder successor of the node (if it exists), and all left child pointers that would normally be null point to the inorder predecessor of the node.

就是说螺纹二叉树实际上是把所有原本为空的右子节点指向了中序遍历顺序之后的那个节点，把所有原本为空的左子节点都指向了中序遍历之前的那个节点，具体例子可以点击[这里](#)。那么这道题跟这个螺纹二叉树又有啥关系呢？由于我们既不能用递归，又不能用栈，那我们如何保证访问顺序是中序遍历的左-根-右呢。原来我们需要构建一个螺纹二叉树，我们需要将所有为空的右子节点指向中序遍历的下一个节点，这样我们中序遍历完左子结点后，就能顺利的回到其根节点继续遍历了。具体算法如下：

1. 初始化指针cur指向root

2. 当cur不为空时

- 如果cur没有左子结点

 - a) 打印出cur的值

 - b) 将cur指针指向其右子节点

 - 反之

 - 将pre指针指向cur的左子树中的最右子节点

 - * 若pre不存在右子节点

 - a) 将其右子节点指回cur

 - b) cur指向其左子节点

 - * 反之

 - a) 将pre的右子节点置空

 - b) 打印cur的值

 - c) 将cur指针指向其右子节点

解法4:

```

1 class Solution {
2 public:
3     vector<int> inorderTraversal(TreeNode *root) {
4         vector<int> res;
5         if (!root) return res;
6         TreeNode *cur, *pre;
7         cur = root;
8         while (cur) {
9             if (!cur->left) {
10                 res.push_back(cur->val);
11                 cur = cur->right;
12             } else {
13                 pre = cur->left;
14                 while (pre->right && pre->right != cur) pre = pre->right;
15                 if (!pre->right) {
16                     pre->right = cur;
17                     cur = cur->left;
18                 } else {
19                     pre->right = NULL;
20                     res.push_back(cur->val);
21                     cur = cur->right;
22                 }
23             }
24         }
25         return res;
26     }
27 };

```

CPP

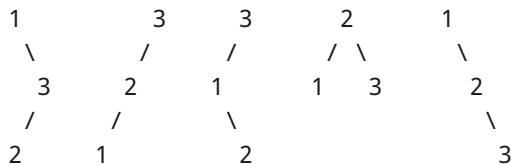
其实Morris遍历不仅仅对中序遍历有用，对先序和后序同样有用，具体可参见网友NOALGO博客，和 Annie Kim's Blog的博客。所以对二叉树的三种常见遍历顺序(先序，中序，后序)就有三种解法(递归，非递归，Morris遍历)，总共有九段代码呀，熟练掌握这九种写法才算初步掌握了树的遍历挖~~至于二叉树的层序遍历也有递归和非递归解法，至于有没有Morris遍历的解法还有待大神们的解答，若真有也请劳烦告知博主一声~~

95. 独一无二的二叉搜索树之二

Given n, generate all structurally unique BST's (binary search trees) that store values 1...n.

For example,

Given n = 3, your program should return all 5 unique BST's shown below.

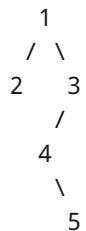


confused what "{1,#,2,3}" means? > read more on how binary tree is serialized on OJ.

OJ's Binary Tree Serialization:

The serialization of a binary tree follows a level order traversal, where '#' signifies a path terminator where no node exists below.

Here's an example:



The above binary tree is serialized as "{1,2,3,#,#,4,#,#,5}".

这道题是之前的 Unique Binary Search Trees 独一无二的二叉搜索树的延伸，之前那个只要求算出所有不同的二叉搜索树的个数，这道题让把那些二叉树都建立出来。这种建树问题一般来说都是用递归来解，这道题也不例外，划分左右子树，递归构造。至于递归函数中为啥都用的是指针，是参考了网友水中的鱼的博客，若不用指针，全部实例化的话会存在大量的对象拷贝，要调用拷贝构造函数，具体我也不太懂，反正感觉挺有道理的，不明觉厉啊--!!!

```

1 class Solution {
2 public:
3     vector<TreeNode *> generateTrees(int n) {
4         if (n == 0) return {};
5         return *generateTreesDFS(1, n);
6     }
7     vector<TreeNode*> *generateTreesDFS(int start, int end) {
8         vector<TreeNode*> *subTree = new vector<TreeNode*>();
9         if (start > end) subTree->push_back(NULL);
10        else {
11            for (int i = start; i <= end; ++i) {
12                vector<TreeNode*> *leftSubTree = generateTreesDFS(start, i - 1);
13                vector<TreeNode*> *rightSubTree = generateTreesDFS(i + 1, end);
14                for (int j = 0; j < leftSubTree->size(); ++j) {
15                    for (int k = 0; k < rightSubTree->size(); ++k) {
16                        TreeNode *node = new TreeNode(i);
17                        node->left = (*leftSubTree)[j];
18                        node->right = (*rightSubTree)[k];
19                        subTree->push_back(node);
20                    }
21                }
22            }
23        }
24        return subTree;
25    }
26 }
27

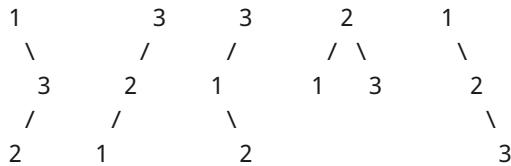
```

96. 独一无二的二叉搜索树

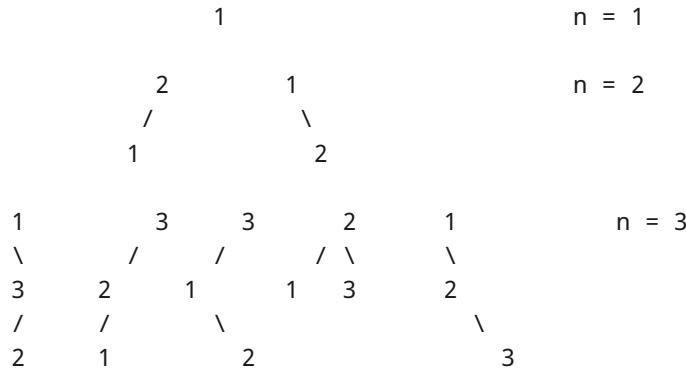
Given n , how many structurally unique BST's (binary search trees) that store values $1 \dots n$?

For example,

Given $n = 3$, there are a total of 5 unique BST's.



这道题实际上是 Catalan Number卡塔兰数的一个例子，如果对卡塔兰数不熟悉的童鞋可能真不太好做。话说其实我也是今天才知道的好嘛-.-|||，为啥我以前都不知道捏？！为啥卡塔兰数不像斐波那契数那样人尽皆知呢，是我太孤陋寡闻么？！不过今天知道也不晚，不断的学习新的东西，这才是刷题的意义所在嘛！好了，废话不多说了，赶紧回到题目上来吧。我们先来看当 $n = 1$ 的情况，只能形成唯一的一棵二叉搜索树， n 分别为 1, 2, 3 的情况如下所示：



就跟斐波那契数列一样，我们把 $n = 0$ 时赋为1，因为空树也算一种二叉搜索树，那么 $n = 1$ 时的情况可以看做是其左子树个数乘以右子树的个数，左右字数都是空树，所以1乘1还是1。那么 $n = 2$ 时，由于1和2都可以为根，分别算出来，再把它们加起来即可。 $n = 2$ 的情况可由下面式子算出：

$$\begin{aligned} dp[2] &= dp[0] * dp[1] \quad (1 \text{ 为根的情况}) \\ &\quad + dp[1] * dp[0] \quad (2 \text{ 为根的情况}) \end{aligned}$$

同理可写出 $n = 3$ 的计算方法：

$$\begin{aligned} dp[3] &= dp[0] * dp[2] \quad (1 \text{ 为根的情况}) \\ &\quad + dp[1] * dp[1] \quad (2 \text{ 为根的情况}) \\ &\quad + dp[2] * dp[0] \quad (3 \text{ 为根的情况}) \end{aligned}$$

由此可以得出卡塔兰数列的递推式为：

$$C_0 = 1 \quad \text{and} \quad C_{n+1} = \sum_{i=0}^n C_i C_{n-i} \quad \text{for } n \geq 0.$$

我们根据以上的分析，可以写出代码如下：

```

1 class Solution {
2 public:
3     int numTrees(int n) {
4         vector<int> dp(n + 1, 0);
5         dp[0] = 1;
6         dp[1] = 1;
7         for (int i = 2; i <= n; ++i) {
8             for (int j = 0; j < i; ++j) {
9                 dp[i] += dp[j] * dp[i - j - 1];
10            }
11        }
12        return dp[n];
13    }
14 }
```

CPP

97. 交织相错的字符串

Given s1, s2, s3, find whether s3 is formed by the interleaving of s1 and s2.

For example,

Given:

```
s1 = "aabcc",
s2 = "dbbca",
```

When s3 = "adbcbcbcac", return true.

When s3 = "aadbbaaccc", return false.

这道求交织相错的字符串和之前那道 Word Break 拆分词句 的题很类似，就想我之前说的只要是遇到字符串的子序列或是匹配问题直接就上动态规划Dynamic Programming，其他的都不要考虑，什么递归呀的都是浮云，千辛万苦的写了递归结果拿到OJ上妥妥Time Limit Exceeded，能把人气昏了，所以还是直接就考虑DP解法省事些。一般来说字符串匹配问题都是更新一个二维dp数组，核心就在于找出递推公式。那么我们还是从题目中给的例子出发吧，手动写出二维数组dp如下：

Ø	d	b	b	c	a
Ø	T	F	F	F	F
a	T	F	F	F	F
a	T	T	T	T	F
b	F	T	T	F	F
c	F	F	T	T	T
c	F	F	F	T	T

首先，这道题的大前提是字符串s1和s2的长度和必须等于s3的长度，如果不等于，肯定返回false。那么当s1和s2是空串的时候，s3必然是空串，则返回true。所以直接给dp[0][0]赋值true，然后若s1和s2其中的一个为空串的话，那么另一个肯定和s3的长度相等，则按位比较，若相同且上一个位置为True，赋True，其余情况都赋False，这样的二维数组dp的边缘就初始化好了。下面只需要找出递推公式来更新整个数组即可，我们发现，在任意非边缘位置dp[i][j]时，它的左边或上边有可能为True或是False，两边都可以更新过来，只要有一条路通着，那么这个点就可以为True。那么我们得分别来看，如果左边的为True，那么我们去除当前对应的s2中的字符串s2[j - 1] 和 s3中对应的位置的字符相比（计算对应位置时还要考虑已匹配的s1中的字符），为s3[j - 1 + i]，如果相等，则赋True，反之赋False。而上边为True的情况也类似，所以可以求出递推公式为：

```
dp[i][j] = (dp[i - 1][j] && s1[i - 1] == s3[i - 1 + j]) || (dp[i][j - 1] && s2[j - 1] == s3[j - 1 + i]);
```

其中dp[i][j] 表示的是 s2 的前 i 个字符和 s1 的前 j 个字符是否匹配 s3 的前 i+j 个字符，根据以上分析，可写出代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool isInterleave(string s1, string s2, string s3) {
4         if (s1.size() + s2.size() != s3.size()) return false;
5         int n1 = s1.size();
6         int n2 = s2.size();
7         vector<vector<bool>> dp(n1 + 1, vector<bool>(n2 + 1, false));
8         dp[0][0] = true;
9         for (int i = 1; i <= n1; ++i) {
10             dp[i][0] = dp[i - 1][0] && (s1[i - 1] == s3[i - 1]);
11         }
12         for (int i = 1; i <= n2; ++i) {
13             dp[0][i] = dp[0][i - 1] && (s2[i - 1] == s3[i - 1]);
14         }
15         for (int i = 1; i <= n1; ++i) {
16             for (int j = 1; j <= n2; ++j) {
17                 dp[i][j] = (dp[i - 1][j] && s1[i - 1] == s3[i - 1 + j]) || (dp[i][j - 1] &&
18 s2[j - 1] == s3[j - 1 + i]);
19             }
20         }
21     return dp[n1][n2];
22     }
23 };

```

我们也可以把for循环合并到一起，用if条件来处理边界情况，整体思路和上面的解法没有太大的区别，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool isInterleave(string s1, string s2, string s3) {
4         if (s1.size() + s2.size() != s3.size()) return false;
5         int n1 = s1.size(), n2 = s2.size();
6         vector<vector<bool>> dp(n1 + 1, vector<bool>(n2 + 1, false));
7         for (int i = 0; i <= n1; ++i) {
8             for (int j = 0; j <= n2; ++j) {
9                 if (i == 0 && j == 0) {
10                     dp[i][j] = true;
11                 } else if (i == 0) {
12                     dp[i][j] = dp[i][j - 1] && s2[j - 1] == s3[i + j - 1];
13                 } else if (j == 0) {
14                     dp[i][j] = dp[i - 1][j] && s1[i - 1] == s3[i + j - 1];
15                 } else {
16                     dp[i][j] = (dp[i - 1][j] && s1[i - 1] == s3[i + j - 1]) || (dp[i][j - 1] && s2[j - 1] == s3[i + j - 1]);
17                 }
18             }
19         }
20     return dp[n1][n2];
21     }
22 };

```

这道题也可以使用带优化的DFS来做，我们使用一个哈希集合，用来保存匹配失败的情况，我们分别用变量i, j, 和k来记录字符串s1, s2, 和s3匹配到的位置，初始化的时候都传入0。在递归函数中，首先根据i和j，算出key值，由于我们的哈希集合中只能放一个数字，而我们要encode两个数字i和j，所以通过用i乘以s3的长度再加上j来得到key，此时我们看，如果key已经在集合中，直接返回false，因为集合中存的是无法匹配的情况。然后先来处理corner case的情况，如果i等于s1的长度了，说明s1的字

符都匹配完了，此时s2剩下的字符和s3剩下的字符可以直接进行匹配了，所以我们直接返回两者是否能匹配的bool值。同理，如果j等于s2的长度了，说明s2的字符都匹配完了，此时s1剩下的字符和s3剩下的字符可以直接进行匹配了，所以我们直接返回两者是否能匹配的bool值。如果s1和s2都有剩余字符，那么当s1的当前字符等于s3的当前字符，那么调用递归函数，注意i和k都加上1，如果递归函数返回true，则当前函数也返回true；还有一种情况是，当s2的当前字符等于s3的当前字符，那么调用递归函数，注意j和k都加上1，如果递归函数返回true，那么当前函数也返回true。如果匹配失败了，则将key加入集合中，并返回false即可，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     bool isInterleave(string s1, string s2, string s3) {
4         if (s1.size() + s2.size() != s3.size()) return false;
5         unordered_set<int> s;
6         return helper(s1, 0, s2, 0, s3, 0, s);
7     }
8     bool helper(string& s1, int i, string& s2, int j, string& s3, int k,
9     unordered_set<int>& s) {
10        int key = i * s3.size() + j;
11        if (s.count(key)) return false;
12        if (i == s1.size()) return s2.substr(j) == s3.substr(k);
13        if (j == s2.size()) return s1.substr(i) == s3.substr(k);
14        if ((s1[i] == s3[k] && helper(s1, i + 1, s2, j, s3, k + 1, s)) ||
15            (s2[j] == s3[k] && helper(s1, i, s2, j + 1, s3, k + 1, s))) return true;
16        s.insert(key);
17        return false;
18    }
};
```

CPP

既然DFS可以，那么BFS也就坐不住了，也要出来浪一波。这里我们需要用队列queue来辅助运算，如果将解法一讲解中的那个二维dp数组列出来的TF图当作一个迷宫的话，那么BFS的目的就是要从(0, 0)位置找一条都是T的路径通到(n1, n2)位置，这里我们还要使用哈希集合，不过此时保存到是已经遍历过的位置，队列中还是存key值，key值的encode方法跟上面DFS解法的相同，初识时放个0进去。然后我们进行while循环，循环条件除了q不为空，还有一个是k小于n3，因为匹配完s3中所有的字符就结束了。然后由于是一层层的遍历，所以要直接循环queue中元素个数的次数，在for循环中，对队首元素进行解码，得到i和j值，如果i小于n1，说明s1还有剩余字符，如果s1当前字符等于s3当前字符，那么把s1的下一个位置i+1跟j一起加码算出key值，如果该key值不在于集合中，则加入集合，同时加入队列queue中；同理，如果j小于n2，说明s2还有剩余字符，如果s2当前字符等于s3当前字符，那么把s2的下一个位置j+1跟i一起加码算出key值，如果该key值不在于集合中，则加入集合，同时加入队列queue中。for循环结束后，k自增1。最后如果匹配成功的话，那么queue中应该只有一个(n1, n2)的key值，且k此时等于n3，所以当queue为空或者k不等于n3的时候都要返回false，参见代码如下：

解法4：

```

1 class Solution {
2 public:
3     bool isInterleave(string s1, string s2, string s3) {
4         if (s1.size() + s2.size() != s3.size()) return false;
5         int n1 = s1.size(), n2 = s2.size(), n3 = s3.size(), k = 0;
6         unordered_set<int> s;
7         queue<int> q{{0}};
8         while (!q.empty() && k < n3) {
9             int len = q.size();
10            for (int t = 0; t < len; ++t) {
11                int i = q.front() / n3, j = q.front() % n3; q.pop();
12                if (i < n1 && s1[i] == s3[k]) {
13                    int key = (i + 1) * n3 + j;
14                    if (!s.count(key)) {
15                        s.insert(key);
16                        q.push(key);
17                    }
18                }
19                if (j < n2 && s2[j] == s3[k]) {
20                    int key = i * n3 + j + 1;
21                    if (!s.count(key)) {
22                        s.insert(key);
23                        q.push(key);
24                    }
25                }
26            }
27            ++k;
28        }
29        return !q.empty() && k == n3;
30    }
31 };

```

98. 验证二叉搜索树

Given a binary tree, determine if it is a valid binary search tree (BST).

Assume a BST is defined as follows:

The left subtree of a node contains only nodes with keys less than the node's key.
The right subtree of a node contains only nodes with keys greater than the node's key.
Both the left and right subtrees must also be binary search trees.
confused what "{1,#,2,3}" means? > read more on how binary tree is serialized on OJ.

这道验证二叉搜索树有很多种解法，可以利用它本身的性质来做，即左<根<右，也可以通过利用中序遍历结果为有序数列来做，下面我们先来看最简单的一种，就是利用其本身性质来做，初始化时带入系统最大值和最小值，在递归过程中换成它们自己的节点值，用long代替int就是为了包括int的边界条件，代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool isValidBST(TreeNode *root) {
4         return isValidBST(root, LONG_MIN, LONG_MAX);
5     }
6     bool isValidBST(TreeNode *root, long mn, long mx) {
7         if (!root) return true;
8         if (root->val <= mn || root->val >= mx) return false;
9         return isValidBST(root->left, mn, root->val) && isValidBST(root->right, root->val,
10 mx);
11     }
12 };

```

这题实际上简化了难度，因为一般的二叉搜索树是左<=根<右，而这道题设定为左<根<右，那么就可以用中序遍历来做。因为如果不去掉左=根这个条件的话，那么下边两个数用中序遍历无法区分：

```

20      20
 /       \
20      20

```

它们的中序遍历结果都一样，但是左边的是BST，右边的不是BST。去掉等号的条件则相当于去掉了这种限制条件。下面我们来看使用中序遍历来做，这种方法思路很直接，通过中序遍历将所有的节点值存到一个数组里，然后再来判断这个数组是不是有序的，代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool isValidBST(TreeNode *root) {
4         if (!root) return true;
5         vector<int> vals;
6         inorder(root, vals);
7         for (int i = 0; i < vals.size() - 1; ++i) {
8             if (vals[i] >= vals[i + 1]) return false;
9         }
10        return true;
11    }
12    void inorder(TreeNode *root, vector<int> &vals) {
13        if (!root) return;
14        inorder(root->left, vals);
15        vals.push_back(root->val);
16        inorder(root->right, vals);
17    }
18 };

```

下面这种解法跟上面那个很类似，都是用递归的中序遍历，但不同之处是不将遍历结果存入一个数组遍历完成再比较，而是每当遍历到一个新节点时和其上一个节点比较，如果不大于上一个节点那么则返回false，全部遍历完成后返回true。代码如下：

解法3：

```

1 class Solution {
2 public:
3     TreeNode *pre;
4     bool isValidBST(TreeNode *root) {
5         int res = 1;
6         pre = NULL;
7         inorder(root, res);
8         if (res == 1) return true;
9         else false;
10    }
11    void inorder(TreeNode *root, int &res) {
12        if (!root) return;
13        inorder(root->left, res);
14        if (!pre) pre = root;
15        else {
16            if (root->val <= pre->val) res = 0;
17            pre = root;
18        }
19        inorder(root->right, res);
20    }
21 };

```

当然这道题也可以用非递归来做，需要用到栈，因为中序遍历可以非递归来实现，所以只要在其上面稍加改动便可，代码如下：

解法4：

```

1 class Solution {
2 public:
3     bool isValidBST(TreeNode* root) {
4         stack<TreeNode*> s;
5         TreeNode *p = root, *pre = NULL;
6         while (p || !s.empty()) {
7             while (p) {
8                 s.push(p);
9                 p = p->left;
10            }
11            TreeNode *t = s.top(); s.pop();
12            if (pre && t->val <= pre->val) return false;
13            pre = t;
14            p = t->right;
15        }
16        return true;
17    }
18 };

```

最后还有一种方法，由于中序遍历还有非递归且无栈的实现方法，称之为Morris遍历，可以参考我之前的博客 Binary Tree Inorder Traversal，这种实现方法虽然写起来比递归版本要复杂的多，但是好处在于是O(1)空间复杂度，参见代码如下：

解法5：

```

1 class Solution {
2 public:
3     bool isValidBST(TreeNode *root) {
4         if (!root) return true;
5         TreeNode *cur = root, *pre, *parent = NULL;
6         bool res = true;
7         while (cur) {
8             if (!cur->left) {
9                 if (parent && parent->val >= cur->val) res = false;
10                parent = cur;
11                cur = cur->right;
12            } else {
13                pre = cur->left;
14                while (pre->right && pre->right != cur) pre = pre->right;
15                if (!pre->right) {
16                    pre->right = cur;
17                    cur = cur->left;
18                } else {
19                    pre->right = NULL;
20                    if (parent->val >= cur->val) res = false;
21                    parent = cur;
22                    cur = cur->right;
23                }
24            }
25        }
26        return res;
27    }
28};

```

99. 复原二叉搜索树

Two elements of a binary search tree (BST) are swapped by mistake.

Recover the tree without changing its structure.

Note:

A solution using O(n) space is pretty straight forward. Could you devise a constant space solution?

confused what "{1,#,2,3}" means? > read more on how binary tree is serialized on OJ.

这道题要求我们复原一个二叉搜索树，说是其中有两个的顺序被调换了，题目要求上说O(n)的解法很直观，这种解法需要用到递归，用中序遍历树，并将所有节点存到一个一维向量中，把所有节点值存到另一个一维向量中，然后对存节点值的一维向量排序，在将排好的数组按顺序赋给节点。这种最一般的解法可针对任意个数目的节点错乱的情况，这里先贴上此种解法：

解法1：

```

1 class Solution {
2 public:
3     void recoverTree(TreeNode *root) {
4         vector<TreeNode*> list;
5         vector<int> vals;
6         inorder(root, list, vals);
7         sort(vals.begin(), vals.end());
8         for (int i = 0; i < list.size(); ++i) {
9             list[i]->val = vals[i];
10        }
11    }
12    void inorder(TreeNode *root, vector<TreeNode*> &list, vector<int> &vals) {
13        if (!root) return;
14        inorder(root->left, list, vals);
15        list.push_back(root);
16        vals.push_back(root->val);
17        inorder(root->right, list, vals);
18    }
19 };

```

然后我上网搜了许多其他解法，看到另一种是用双指针来代替一维向量的，但是这种方法用到了递归，也不是O(1)空间复杂度的解法，这里需要三个指针，first, second分别表示第一个和第二个错乱位置的节点，pre指向当前节点的中序遍历的前一个节点。这里用传统的中序遍历递归来做，不过再应该输出节点值的地方，换成了判断pre和当前节点值的大小，如果pre的大，若first为空，则将first指向pre指的节点，把second指向当前节点。这样中序遍历完整个树，若first和second都存在，则交换它们的节点值即可。这个算法的空间复杂度仍为O(n)，n为树的高度，代码如下：

解法2：

```

1 class Solution {
2 public:
3     TreeNode *pre;
4     TreeNode *first;
5     TreeNode *second;
6     void recoverTree(TreeNode *root) {
7         pre = NULL;
8         first = NULL;
9         second = NULL;
10        inorder(root);
11        if (first && second) swap(first->val, second->val);
12    }
13    void inorder(TreeNode *root) {
14        if (!root) return;
15        inorder(root->left);
16        if (!pre) pre = root;
17        else {
18            if (pre->val > root->val) {
19                if (!first) first = pre;
20                second = root;
21            }
22            pre = root;
23        }
24        inorder(root->right);
25    }
26 };

```

这道题的真正符合要求的解法应该用的Morris遍历，这是一种非递归且不使用栈，空间复杂度为O(1)的遍历方法，可参见我之前的博客Binary Tree Inorder Traversal 二叉树的中序遍历，在其基础上做些修改，加入first, second和parent指针，来比较当前节点值和中序遍历的前一节点值的大小，跟上面递归算法的思路相似，代码如下：

解法3：

```

1 class Solution {
2 public:
3     void recoverTree(TreeNode *root) {
4         TreeNode *first = NULL, *second = NULL, *parent = NULL;
5         TreeNode *cur, *pre;
6         cur = root;
7         while (cur) {
8             if (!cur->left) {
9                 if (parent && parent->val > cur->val) {
10                     if (!first) first = parent;
11                     second = cur;
12                 }
13                 parent = cur;
14                 cur = cur->right;
15             } else {
16                 pre = cur->left;
17                 while (pre->right && pre->right != cur) pre = pre->right;
18                 if (!pre->right) {
19                     pre->right = cur;
20                     cur = cur->left;
21                 } else {
22                     pre->right = NULL;
23                     if (parent->val > cur->val) {
24                         if (!first) first = parent;
25                         second = cur;
26                     }
27                     parent = cur;
28                     cur = cur->right;
29                 }
30             }
31         }
32         if (first && second) swap(first->val, second->val);
33     }
34 };

```

100. 判断相同树

Given two binary trees, write a function to check if they are equal or not.

Two binary trees are considered equal if they are structurally identical and the nodes have the same value.

判断两棵树是否相同和之前的判断两棵树是否对称都是一样的原理，利用深度优先搜索DFS来递归。代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool isSameTree(TreeNode *p, TreeNode *q) {
4         if (!p && !q) return true;
5         if ((p && !q) || (!p && q) || (p->val != q->val)) return false;
6         return isSameTree(p->left, q->left) && isSameTree(p->right, q->right);
7     }
8 };

```

这道题还有非递归的解法，因为二叉树的四种遍历(层序，先序，中序，后序)均有各自的迭代和递归的写法，这里我们先来看先序的迭代写法，相当于同时遍历两个数，然后每个节点都进行比较，参见代码如下：

解法2：

```

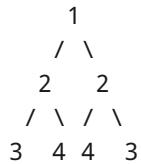
1 class Solution {
2 public:
3     bool isSameTree(TreeNode* p, TreeNode* q) {
4         stack<TreeNode*> s1, s2;
5         if (p) s1.push(p);
6         if (q) s2.push(q);
7         while (!s1.empty() && !s2.empty()) {
8             TreeNode *t1 = s1.top(); s1.pop();
9             TreeNode *t2 = s2.top(); s2.pop();
10            if (t1->val != t2->val) return false;
11            if (t1->left) s1.push(t1->left);
12            if (t2->left) s2.push(t2->left);
13            if (s1.size() != s2.size()) return false;
14            if (t1->right) s1.push(t1->right);
15            if (t2->right) s2.push(t2->right);
16            if (s1.size() != s2.size()) return false;
17        }
18        return s1.size() == s2.size();
19    }
20};

```

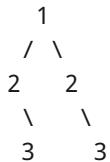
101. 判断对称树

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).

For example, this binary tree is symmetric:



But the following is not:



Note:

Bonus points if you could solve it both recursively and iteratively.

判断二叉树是否是平衡树，比如有两个节点n1, n2，我们需要比较n1的左子节点的值和n2的右子节点的值是否相等，同时还要比较n1的右子节点的值和n2的左子结点的值是否相等，以此类推比较完所有的左右两个节点。我们可以用递归和迭代两种方法来实现，写法不同，但是算法核心都一样。

解法1(递归法):

```

1 | class Solution {
2 | public:
3 |     bool isSymmetric(TreeNode *root) {
4 |         if (!root) return true;
5 |         return isSymmetric(root->left, root->right);
6 |     }
7 |     bool isSymmetric(TreeNode *left, TreeNode *right) {
8 |         if (!left && !right) return true;
9 |         if (left && !right || !left && right || left->val != right->val) return false;
10 |         return isSymmetric(left->left, right->right) && isSymmetric(left->right, right-
11 | >left);
12 |     }
13 | };
  
```

CPP

解法2(迭代法):

```

1 class Solution {
2 public:
3     bool isSymmetric(TreeNode *root) {
4         if (!root) return true;
5         queue<TreeNode*> q1, q2;
6         q1.push(root->left);
7         q2.push(root->right);
8
9         while (!q1.empty() && !q2.empty()) {
10            TreeNode *node1 = q1.front();
11            TreeNode *node2 = q2.front();
12            q1.pop();
13            q2.pop();
14            if((node1 && !node2) || (!node1 && node2)) return false;
15            if (node1) {
16                if (node1->val != node2->val) return false;
17                q1.push(node1->left);
18                q1.push(node1->right);
19                q2.push(node2->right);
20                q2.push(node2->left);
21            }
22        }
23        return true;
24    }
25 };

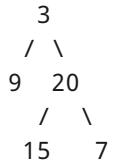
```

102. 二叉树层序遍历

Given a binary tree, return the level order traversal of its nodes' values. (ie, from left to right, level by level).

For example:

Given binary tree {3,9,20,#,#,15,7},



return its level order traversal as:

```

[
  [3],
  [9,20],
  [15,7]
]
  
```

层序遍历二叉树是典型的广度优先搜索BFS的应用，但是这里稍微复杂一点的是，我们要把各个层的数分开，存到一个二维向量里面，大体思路还是基本相同的，建立一个queue，然后先把根节点放进去，这时候找根节点的左右两个子节点，这时候去掉根节点，此时queue里的元素就是下一层的所有节点，用一个for循环遍历它们，然后存到一个一维向量里，遍历完之后再把这个一维向量存到二维向量里，以此类推，可以完成层序遍历。代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<vector<int>> levelOrder(TreeNode *root) {
4         vector<vector<int>> res;
5         if (root == NULL) return res;
6
7         queue<TreeNode*> q;
8         q.push(root);
9         while (!q.empty()) {
10             vector<int> oneLevel;
11             int size = q.size();
12             for (int i = 0; i < size; ++i) {
13                 TreeNode *node = q.front();
14                 q.pop();
15                 oneLevel.push_back(node->val);
16                 if (node->left) q.push(node->left);
17                 if (node->right) q.push(node->right);
18             }
19             res.push_back(oneLevel);
20         }
21         return res;
22     }
23 };

```

下面我们来看递归的写法，核心就在于我们需要一个二维数组，和一个变量level，当level递归到上一层的个数，我们新建一个空层，继续往里面加数字，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<vector<int>> levelOrder(TreeNode* root) {
4         vector<vector<int>> res;
5         levelorder(root, 0, res);
6         return res;
7     }
8     void levelorder(TreeNode *root, int level, vector<vector<int>> &res) {
9         if (!root) return;
10        if (res.size() == level) res.push_back({});
11        res[level].push_back(root->val);
12        if (root->left) levelorder(root->left, level + 1, res);
13        if (root->right) levelorder(root->right, level + 1, res);
14    }
15 };

```

103. 二叉树的之字形层序遍历

Given a binary tree, return the zigzag level order traversal of its nodes' values. (ie, from left to right, then right to left for the next level and alternate between).

For example:

Given binary tree {3,9,20,#,#,15,7},

```
    3
   / \
  9  20
 /   \
15    7
```

return its zigzag level order traversal as:

```
[
  [3],
  [20,9],
  [15,7]
]
```

confused what "{1,#,2,3}" means? > read more on how binary tree is serialized on OJ.

这道二叉树的之字形层序遍历是之前那道[LeetCode] Binary Tree Level Order Traversal 二叉树层序遍历的变形，不同之处在于一行是从左到右遍历，下一行是从右往左遍历，交叉往返的之字形的层序遍历。根据其特点我们用到栈的后进先出的特点，这道题我们维护两个栈，相邻两行分别存到两个栈中，进栈的顺序也不相同，一个栈是先进左子结点然后右子节点，另一个栈是先进右子节点然后左子结点，这样出栈的顺序就是我们想要的之字形了，代码如下：

```

1 class Solution {
2 public:
3     vector<vector<int>> zigzagLevelOrder(TreeNode *root) {
4         vector<vector<int>> res;
5         if (!root) return res;
6         stack<TreeNode*> s1;
7         stack<TreeNode*> s2;
8         s1.push(root);
9         vector<int> out;
10        while (!s1.empty() || !s2.empty()) {
11            while (!s1.empty()) {
12                TreeNode *cur = s1.top();
13                s1.pop();
14                out.push_back(cur->val);
15                if (cur->left) s2.push(cur->left);
16                if (cur->right) s2.push(cur->right);
17            }
18            if (!out.empty()) res.push_back(out);
19            out.clear();
20            while (!s2.empty()) {
21                TreeNode *cur = s2.top();
22                s2.pop();
23                out.push_back(cur->val);
24                if (cur->right) s1.push(cur->right);
25                if (cur->left) s1.push(cur->left);
26            }
27            if (!out.empty()) res.push_back(out);
28            out.clear();
29        }
30        return res;
31    }
32 };

```

104. 二叉树的最大深度

Given a binary tree, find its maximum depth.

The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

求二叉树的最大深度问题用到深度优先搜索DFS，递归的完美应用，跟求二叉树的最小深度问题原理相同。代码如下：

解法1:

```

1 class Solution {
2 public:
3     int maxDepth(TreeNode* root) {
4         if (!root) return 0;
5         return 1 + max(maxDepth(root->left), maxDepth(root->right));
6     }
7 };

```

我们也可以使用层序遍历二叉树，然后计数总层数，即为二叉树的最大深度，参见代码如下：

解法2:

```

1 class Solution {
2 public:
3     int maxDepth(TreeNode* root) {
4         if (!root) return 0;
5         int res = 0;
6         queue<TreeNode*> q;
7         q.push(root);
8         while (!q.empty()) {
9             ++res;
10            int n = q.size();
11            for (int i = 0; i < n; ++i) {
12                TreeNode *t = q.front(); q.pop();
13                if (t->left) q.push(t->left);
14                if (t->right) q.push(t->right);
15            }
16        }
17        return res;
18    }
19 };

```

105. 由先序和中序遍历建立二叉树

Given preorder and inorder traversal of a tree, construct the binary tree.

Note:

You may assume that duplicates do not exist in the tree.

这道题要求用先序和中序遍历来建立二叉树，跟之前那道Construct Binary Tree from Inorder and Postorder Traversal 由中序和后序遍历建立二叉树原理基本相同，针对这道题，由于先序的顺序的第一个肯定是根，所以原二叉树的根节点可以知道，题目中给了一个很关键的条件就是树中没有相同元素，有了这个条件我们就可以在中序遍历中也定位出根节点的位置，并以根节点的位置将中序遍历拆分为左右两个部分，分别对其递归调用原函数。代码如下：

```

1 class Solution {
2 public:
3     TreeNode *buildTree(vector<int> &preorder, vector<int> &inorder) {
4         return buildTree(preorder, 0, preorder.size() - 1, inorder, 0, inorder.size() - 1);
5     }
6     TreeNode *buildTree(vector<int> &preorder, int pLeft, int pRight, vector<int> &inorder,
7     int iLeft, int iRight) {
8         if (pLeft > pRight || iLeft > iRight) return NULL;
9         int i;
10        for (i = iLeft; i <= iRight; ++i) {
11            if (preorder[pLeft] == inorder[i]) break;
12        }
13        TreeNode *cur = new TreeNode(preorder[pLeft]);
14        cur->left = buildTree(preorder, pLeft + 1, pLeft + i - iLeft, inorder, iLeft, i -
15        1);
16        cur->right = buildTree(preorder, pLeft + i - iLeft + 1, pRight, inorder, i + 1,
17        iRight);
18        return cur;
19    }
20};

```

106. 由中序和后序遍历建立二叉树

Given inorder and postorder traversal of a tree, construct the binary tree.

Note:

You may assume that duplicates do not exist in the tree.

这道题要求从中序和后序遍历的结果来重建原二叉树，我们知道中序的遍历顺序是左-根-右，后序的顺序是左-右-根，对于这种树的重建一般都是采用递归来做，可参见我之前的一篇博客Convert Sorted Array to Binary Search Tree 将有序数组转为二叉搜索树。针对这道题，由于后序的顺序的最后一个肯定是根，所以原二叉树的根节点可以知道，题目中给了一个很关键的条件就是树中没有相同元素，有了这个条件我们就可以在中序遍历中也定位出根节点的位置，并以根节点的位置将中序遍历拆分为左右两个部分，分别对其递归调用原函数。代码如下：

```

1 class Solution {
2 public:
3     TreeNode *buildTree(vector<int> &inorder, vector<int> &postorder) {
4         return buildTree(inorder, 0, inorder.size() - 1, postorder, 0, postorder.size() -
5 );
6     }
7     TreeNode *buildTree(vector<int> &inorder, int iLeft, int iRight, vector<int>
8 &postorder, int pLeft, int pRight) {
9         if (iLeft > iRight || pLeft > pRight) return NULL;
10        TreeNode *cur = new TreeNode(postorder[pRight]);
11        int i = 0;
12        for (i = iLeft; i < inorder.size(); ++i) {
13            if (inorder[i] == cur->val) break;
14        }
15        cur->left = buildTree(inorder, iLeft, i - 1, postorder, pLeft, pLeft + i - iLeft -
16        1);
17        cur->right = buildTree(inorder, i + 1, iRight, postorder, pLeft + i - iLeft, pRight -
- 1);
18        return cur;
19    }
20};

```

上述代码中需要小心的地方就是递归是postorder的左右index很容易写错，比如 pLeft + i - iLeft - 1, 这个又长又不好记，首先我们要记住 i - iLeft 是计算inorder中根节点位置和左边起始点的距离，然后再加上postorder左边起始点然后再减1。我们可以这样分析，如果根节点就是左边起始点的话，那么拆分的话左边序列应该为空集，此时i - iLeft 为0， pLeft + 0 - 1 < pLeft, 那么再递归调用时就会返回NULL, 成立。如果根节点是左边起始点紧跟的一个，那么i - iLeft 为1， pLeft + 1 - 1 = pLeft，再递归调用时还会生成一个节点，就是pLeft位置上的节点，为原二叉树的一个叶节点。

107. 二叉树层序遍历之二

Given a binary tree, return the bottom-up level order traversal of its nodes' values. (ie, from left to right, level by level from leaf to root).

For example:

Given binary tree {3,9,20,#,#,15,7},

```

3
/ \
9  20
 / \
15  7

```

return its bottom-up level order traversal as:

```
[
[15,7],
[9,20],
[3]
]
```

从底部层序遍历其实还是从顶部开始遍历，只不过最后存储的方式有所改变，可以参见我之前的博文
<http://www.cnblogs.com/grandyang/p/4051321.html>。代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<vector<int>> levelOrderBottom(TreeNode *root) {
4         vector<vector<int>> res;
5         if (root == NULL) return res;
6
7         queue<TreeNode*> q;
8         q.push(root);
9         while (!q.empty()) {
10             vector<int> oneLevel;
11             int size = q.size();
12             for (int i = 0; i < size; ++i) {
13                 TreeNode *node = q.front();
14                 q.pop();
15                 oneLevel.push_back(node->val);
16                 if (node->left) q.push(node->left);
17                 if (node->right) q.push(node->right);
18             }
19             res.insert(res.begin(), oneLevel);
20         }
21         return res;
22     }
23 }
```

CPP

下面我们来看递归的解法，核心就在于我们需要一个二维数组，和一个变量level，当level递归到上一层的个数，我们新建一个空层，继续往里面加数字，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<vector<int>> levelOrderBottom(TreeNode* root) {
4         vector<vector<int>> res;
5         levelorder(root, 0, res);
6         return vector<vector<int>> (res.rbegin(), res.rend());
7     }
8     void levelorder(TreeNode *root, int level, vector<vector<int>> &res) {
9         if (!root) return;
10        if (res.size() == level) res.push_back({});
11        res[level].push_back(root->val);
12        if (root->left) levelorder(root->left, level + 1, res);
13        if (root->right) levelorder(root->right, level + 1, res);
14    }
15 };

```

108. 将有序数组转为二叉搜索树

Given an array where elements are sorted in ascending order, convert it to a height balanced BST.

这道题是要将有序数组转为二叉搜索树，所谓二叉搜索树，是一种始终满足左<根<右的特性，如果将二叉搜索树按中序遍历的话，得到的就是一个有序数组了。那么反过来，我们可以得知，根节点应该是有序数组的中间点，从中间点分为左右两个有序数组，在分别找出其中间点作为原中间点的左右两个子节点，这不就是是二分查找法的核心思想么。所以这道题考的就是二分查找法，代码如下：

```

1 class Solution {
2 public:
3     TreeNode *sortedArrayToBST(vector<int> &num) {
4         return sortedArrayToBST(num, 0, num.size() - 1);
5     }
6     TreeNode *sortedArrayToBST(vector<int> &num, int left, int right) {
7         if (left > right) return NULL;
8         int mid = (left + right) / 2;
9         TreeNode *cur = new TreeNode(num[mid]);
10        cur->left = sortedArrayToBST(num, left, mid - 1);
11        cur->right = sortedArrayToBST(num, mid + 1, right);
12        return cur;
13    }
14 };

```

109. 将有序链表转为二叉搜索树

Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST.

这道题是要求把有序链表转为二叉搜索树，和之前那道Convert Sorted Array to Binary Search Tree 将有序数组转为二叉搜索树思路完全一样，只不过是操作的数据类型有所差别，一个是数组，一个是链表。数组方便就方便在可以通过index直接访问任意一个元素，而链表不行。由于二分查找法每次需要找到中点，而链表的查找中间点可以通过快慢指针来操作，可参见之前的两篇博客Reorder List 链表重排序和Linked List Cycle II 单链表中的环之二有关快慢指针的应用。找到中点后，要以中点的值建立一个数的根节点，然后需要把原链表断开，分为前后两个链表，都不能包含原中节点，然后再分别对这两个链表递归调用原函数，分别连上左右子节点即可。代码如下：

```

1  /**
2   * Definition for singly-linked list.
3   * struct ListNode {
4   *     int val;
5   *     ListNode *next;
6   *     ListNode(int x) : val(x), next(NULL) {}
7   * };
8   */
9  /**
10  * Definition for binary tree
11  * struct TreeNode {
12  *     int val;
13  *     TreeNode *left;
14  *     TreeNode *right;
15  *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
16  * };
17  */
18 class Solution {
19 public:
20     TreeNode *sortedListToBST(ListNode *head) {
21         if (!head) return NULL;
22         if (!head->next) return new TreeNode(head->val);
23         ListNode *slow = head;
24         ListNode *fast = head;
25         ListNode *last = slow;
26         while (fast->next && fast->next->next) {
27             last = slow;
28             slow = slow->next;
29             fast = fast->next->next;
30         }
31         fast = slow->next;
32         last->next = NULL;
33         TreeNode *cur = new TreeNode(slow->val);
34         if (head != slow) cur->left = sortedListToBST(head);
35         cur->right = sortedListToBST(fast);
36         return cur;
37     }
38 };

```

110. 平衡二叉树

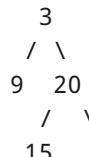
Given a binary tree, determine if it is height-balanced.

For this problem, a height-balanced binary tree is defined as:

a binary tree in which the depth of the two subtrees of every node never differ by more than 1.

Example 1:

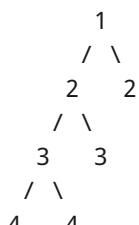
Given the following tree [3,9,20,null,null,15,7]:



Return true.

Example 2:

Given the following tree [1,2,2,3,3,null,4,4]:



Return false.

求二叉树是否平衡，根据题目中的定义，高度平衡二叉树是每一个结点的两个子树的深度差不能超过1，那么我们肯定需要一个求各个点深度的函数，然后对每个节点的两个子树来比较深度差，时间复杂度为 $O(N \lg N)$ ，代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool isBalanced(TreeNode *root) {
4         if (!root) return true;
5         if (abs(getDepth(root->left) - getDepth(root->right)) > 1) return false;
6         return isBalanced(root->left) && isBalanced(root->right);
7     }
8     int getDepth(TreeNode *root) {
9         if (!root) return 0;
10        return 1 + max(getDepth(root->left), getDepth(root->right));
11    }
12 };

```

CPP

上面那个方法正确但不是很高效率，因为每一个点都会被上面的点计算深度时访问一次，我们可以进行优化。方法是如果我们发现子树不平衡，则不计算具体的深度，而是直接返回-1。那么优化后的方法为：对于每一个节点，我们通过checkDepth方法递归获得左右子树的深度，如果子树是平衡的，则返回真实的深度，若不平衡，直接返回-1，此方法时间复杂度 $O(N)$ ，空间复杂度 $O(H)$ ，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool isBalanced(TreeNode *root) {
4         if (checkDepth(root) == -1) return false;
5         else return true;
6     }
7     int checkDepth(TreeNode *root) {
8         if (!root) return 0;
9         int left = checkDepth(root->left);
10        if (left == -1) return -1;
11        int right = checkDepth(root->right);
12        if (right == -1) return -1;
13        int diff = abs(left - right);
14        if (diff > 1) return -1;
15        else return 1 + max(left, right);
16    }
17 };

```

111. 二叉树的最小深度

Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

二叉树的经典问题之最小深度问题就是就最短路径的节点个数，还是用深度优先搜索DFS来完成，万能的递归啊。。。请看代码：

```

1 /**
2  * Definition for binary tree
3  * struct TreeNode {
4  *     int val;
5  *     TreeNode *left;
6  *     TreeNode *right;
7  *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
8  * };
9 */
10 class Solution {
11 public:
12     int minDepth(TreeNode *root) {
13         if (root == NULL) return 0;
14         if (root->left == NULL && root->right == NULL) return 1;
15
16         if (root->left == NULL) return minDepth(root->right) + 1;
17         else if (root->right == NULL) return minDepth(root->left) + 1;
18         else return 1 + min(minDepth(root->left), minDepth(root->right));
19     }
20 }
21 };

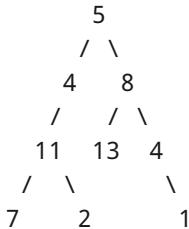
```

112. 二叉树的路径和

Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.

For example:

Given the below binary tree and sum = 22,



return true, as there exist a root-to-leaf path 5->4->11->2 which sum is 22.

这道求二叉树的路径需要用深度优先算法DFS的思想来遍历每一条完整的路径，也就是利用递归不停找子节点的左右子节点，而调用递归函数的参数只有当前节点和sum值。首先，如果输入的是一个空节点，则直接返回false，如果如果输入的只有一个根节点，则比较当前根节点的值和参数sum值是否相同，若相同，返回true，否则false。这个条件也是递归的终止条件。下面我们要开始递归了，由于函数的返回值是True/False，我们可以同时两个方向一起递归，中间用或||连接，只要有一个是True，整个结果就是True。递归左右节点时，这时候的sum值应该是原sum值减去当前节点的值。代码如下：

```

1 class Solution {
2 public:
3     bool hasPathSum(TreeNode *root, int sum) {
4         if (root == NULL) return false;
5         if (root->left == NULL && root->right == NULL && root->val == sum) return true;
6         return hasPathSum(root->left, sum - root->val) || hasPathSum(root->right, sum -
7             root->val);
8     }
9 };

```

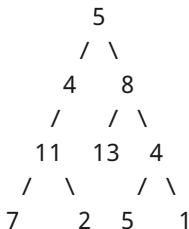
CPP

113. 二叉树路径之和之二

Given a binary tree and a sum, find all root-to-leaf paths where each path's sum equals the given sum.

For example:

Given the below binary tree and sum = 22,



return

```
[
  [5,4,11,2],
  [5,8,4,5]
]
```

这道二叉树路径之和在之前的基础上又需要找出路径(可以参见我之前的博客

<http://www.cnblogs.com/grandyang/p/4036961.html>),但是基本思想都一样,还是需要用深度优先搜索DFS,只不过数据结构相对复杂一点,需要用到二维的vector,而且每当DFS搜索到新节点时,都要保存该节点。而且每当找出一条路径之后,都将这个保存为一维vector的路径保存到最终结果二维vector中。并且,每当DFS搜索到子节点,发现不是路径和时,返回上一个结点时,需要把该节点从一维vector中移除。代码如下:

解法1:

```

1 class Solution {
2 public:
3     vector<vector<int>> pathSum(TreeNode *root, int sum) {
4         vector<vector<int>> res;
5         vector<int> out;
6         helper(root, sum, out, res);
7         return res;
8     }
9     void helper(TreeNode* node, int sum, vector<int>& out, vector<vector<int>>& res) {
10        if (!node) return;
11        out.push_back(node->val);
12        if (sum == node->val && !node->left && !node->right) {
13            res.push_back(out);
14        }
15        helper(node->left, sum - node->val, out, res);
16        helper(node->right, sum - node->val, out, res);
17        out.pop_back();
18    }
19 };

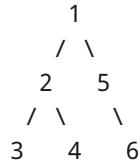
```

114. 将二叉树展开成链表

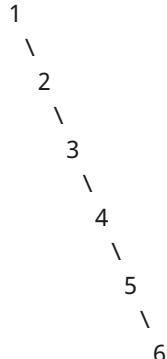
Given a binary tree, flatten it to a linked list in-place.

For example,

Given



The flattened tree should look like:



[click to show hints.](#)

Hints:

If you notice carefully in the flattened tree, each node's right child points to the next node of a pre-order traverse

这道题要求把二叉树展开成链表，根据展开后形成的链表的顺序分析出是使用先序遍历，那么只要是数的遍历就有递归和非递归的两种方法来求解，这里我们也用两种方法来求解。首先来看递归版本的，思路是先利用DFS的思路找到最左子节点，然后回到其父节点，把其父节点和右子节点断开，将原左子结点连上父节点的右子节点上，然后再把原右子节点连到新右子节点的右子节点上，然后再回到上一父节点做相同操作。代码如下：

解法1：

```

1 class Solution {
2 public:
3     void flatten(TreeNode *root) {
4         if (!root) return;
5         if (root->left) flatten(root->left);
6         if (root->right) flatten(root->right);
7         TreeNode *tmp = root->right;
8         root->right = root->left;
9         root->left = NULL;
10        while (root->right) root = root->right;
11        root->right = tmp;
12    }
13 };
  
```

CPP

例如，对于下面的二叉树，上述算法的变换的过程如下：

```

1
 / \
2   5
 / \   \
3   4   6

```

```

1
 / \
2   5
 \   \
3   6
 \
  4

```

```

1
 \
2
 \
  3
  \
   4
   \
    5
    \
     6

```

下面我们再来看非迭代版本的实现，这个方法是从根节点开始出发，先检测其左子结点是否存在，如存在则将根节点和其右子节点断开，将左子结点及其后面所有结构一起连到原右子节点的位置，把原右子节点连到元左子结点最后面的右子节点之后。代码如下：

解法2：

```

1 class Solution {
2 public:
3     void flatten(TreeNode *root) {
4         TreeNode *cur = root;
5         while (cur) {
6             if (cur->left) {
7                 TreeNode *p = cur->left;
8                 while (p->right) p = p->right;
9                 p->right = cur->right;
10                cur->right = cur->left;
11                cur->left = NULL;
12            }
13            cur = cur->right;
14        }
15    }
16 }

```

CPP

例如，对于下面的二叉树，上述算法的变换的过程如下：

```

1
/
2   5
/ \
3   4   6

```

```

1
\
2
/ \
3   4
  \
    5
  \
    6

```

```

1
\
2
\ 
3
  \
    4
  \
    5
  \
    6

```

前序迭代解法如下：

解法3：

```

1 class Solution {
2 public:
3     void flatten(TreeNode* root) {
4         if (!root) return;
5         stack<TreeNode*> s;
6         s.push(root);
7         while (!s.empty()) {
8             TreeNode *t = s.top(); s.pop();
9             if (t->left) {
10                 TreeNode *r = t->left;
11                 while (r->right) r = r->right;
12                 r->right = t->right;
13                 t->right = t->left;
14                 t->left = NULL;
15             }
16             if (t->right) s.push(t->right);
17         }
18     }
19 }

```

CPP

此题还可以延伸到用中序，后序，层序的遍历顺序来展开原二叉树，分别又有其对应的递归和非递归的方法，有兴趣的童鞋可以自行实现。

115. 不同的子序列

Given a string S and a string T, count the number of distinct subsequences of T in S.

A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, "ACE" is a subsequence of "ABCDE" while "AEC" is not).

Here is an example:

S = "rabbbit", T = "rabbit"

Return 3.

看到有关字符串的子序列或者配准类的问题，首先应该考虑的就是用动态规划Dynamic Programming来求解，这个应成为条件反射。而所有DP问题的核心就是找出递推公式，想这道题就是递推一个二维的dp数组，下面我们从题目中给的例子来分析，这个二维dp数组应为：

	Ø	r	a	b	b	b	i	t
Ø	1	1	1	1	1	1	1	1
r	0	1	1	1	1	1	1	1
a	0	0	1	1	1	1	1	1
b	0	0	0	1	2	3	3	3
b	0	0	0	0	1	3	3	3
i	0	0	0	0	0	0	3	3
t	0	0	0	0	0	0	0	3

首先，若原字符串和子序列都为空时，返回1，因为空串也是空串的一个子序列。若原字符串不为空，而子序列为为空，也返回1，因为空串也是任意字符串的一个子序列。而当原字符串为空，子序列不为空时，返回0，因为非空字符串不能当空字符串的子序列。理清这些，二维数组dp的边缘便可以初始化了，下面只要找出递推式，就可以更新整个dp数组了。我们通过观察上面的二维数组可以发现，当更新到 $dp[i][j]$ 时， $dp[i][j] \geq dp[i][j - 1]$ 总是成立，再进一步观察发现，当 $T[i - 1] == S[j - 1]$ 时， $dp[i][j] = dp[i][j - 1] + dp[i - 1][j - 1]$ ，若不等， $dp[i][j] = dp[i][j - 1]$ ，所以，综合以上，递推式为：

$$dp[i][j] = dp[i][j - 1] + (T[i - 1] == S[j - 1] ? dp[i - 1][j - 1] : 0)$$

根据以上分析，可以写出代码如下：

```

1 class Solution {
2 public:
3     int numDistinct(string S, string T) {
4         int dp[T.size() + 1][S.size() + 1];
5         for (int i = 0; i <= S.size(); ++i) dp[0][i] = 1;
6         for (int i = 1; i <= T.size(); ++i) dp[i][0] = 0;
7         for (int i = 1; i <= T.size(); ++i) {
8             for (int j = 1; j <= S.size(); ++j) {
9                 dp[i][j] = dp[i][j - 1] + (T[i - 1] == S[j - 1] ? dp[i - 1][j - 1] : 0);
10            }
11        }
12        return dp[T.size()][S.size()];
13    }
14 }
```

CPP

116. 每个节点的右向指针

Given a binary tree

```
struct TreeLinkNode {
    TreeLinkNode *left;
    TreeLinkNode *right;
    TreeLinkNode *next;
}
```

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.

Initially, all next pointers are set to NULL.

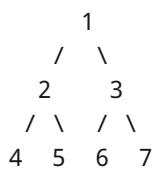
Note:

You may only use constant extra space.

You may assume that it is a perfect binary tree (ie, all leaves are at the same level, and every parent has two children).

For example,

Given the following perfect binary tree,



After calling your function, the tree should look like:

```

 1 -> NULL
 /
 2 -> 3 -> NULL
 / \ / \
4->5->6->7 -> NULL

```

这道题实际上是树的层序遍历的应用，可以参考之前的博客Binary Tree Level Order Traversal 二叉树层序遍历，既然是遍历，就有递归和非递归两种方法，最好两种方法都要掌握，都要会写。下面先来看递归的解法，由于是完全二叉树，所以若节点的左子结点存在的话，其右子节点必定存在，所以左子结点的next指针可以直接指向其右子节点，对于其右子节点的处理方法是，判断其父节点的next是否为空，若不为空，则指向其next指针指向的节点的左子结点，若为空则指向NULL，代码如下：

解法1：

```

1 class Solution {
2 public:
3     void connect(TreeLinkNode *root) {
4         if (!root) return;
5         if (root->left) root->left->next = root->right;
6         if (root->right) root->right->next = root->next? root->next->left : NULL;
7         connect(root->left);
8         connect(root->right);
9     }
10 };

```

CPP

对于非递归的解法要稍微复杂一点，但也不算特别复杂，需要用到queue来辅助，由于是层序遍历，每层的节点都按顺序加入queue中，而每当从queue中取出一个元素时，将其next指针指向queue中下一个节点即可。代码如下：

解法2：

```
1 class Solution {
2 public:
3     void connect(TreeLinkNode *root) {
4         if (!root) return;
5         queue<TreeLinkNode*> q;
6         q.push(root);
7         q.push(NULL);
8         while (true) {
9             TreeLinkNode *cur = q.front();
10            q.pop();
11            if (cur) {
12                cur->next = q.front();
13                if (cur->left) q.push(cur->left);
14                if (cur->right) q.push(cur->right);
15            } else {
16                if (q.size() == 0 || q.front() == NULL) return;
17                q.push(NULL);
18            }
19        }
20    }
21};
```

CPP

上面的方法巧妙的通过给queue中添加空指针NULL来达到分层的目的，使每层的最后一个节点的next可以指向NULL，那么我们可以换一种方法来实现分层，我们对于每层的开头元素开始遍历之前，先统计一下该层的总个数，用个for循环，这样for循环结束的时候，我们就知道该层已经被遍历完了，这也是一种好办法：

解法3：

```
1 class Solution {
2 public:
3     void connect(TreeLinkNode *root) {
4         if (!root) return;
5         queue<TreeLinkNode*> q;
6         q.push(root);
7         while (!q.empty()) {
8             int size = q.size();
9             for (int i = 0; i < size; ++i) {
10                 TreeLinkNode *t = q.front(); q.pop();
11                 if (i < size - 1) {
12                     t->next = q.front();
13                 }
14                 if (t->left) q.push(t->left);
15                 if (t->right) q.push(t->right);
16             }
17         }
18     }
19};
```

CPP

上面三种方法虽然呀，但是都不符合题意，题目中要求用O(1)的空间复杂度，所以我们来看下面这种碉堡了的方法。用两个指针start和cur，其中start标记每一层的起始节点，cur用来遍历该层的节点，设计思路之巧妙，不得不服啊：

解法4：

```

1 class Solution {
2 public:
3     void connect(TreeLinkNode *root) {
4         if (!root) return;
5         TreeLinkNode *start = root, *cur = NULL;
6         while (start->left) {
7             cur = start;
8             while (cur) {
9                 cur->left->next = cur->right;
10                if (cur->next) cur->right->next = cur->next->left;
11                cur = cur->next;
12            }
13            start = start->left;
14        }
15    }
16 };

```

CPP

117. 每个节点的右向指针之二

Follow up for problem "Populating Next Right Pointers in Each Node".

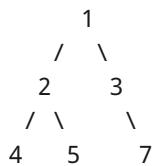
What if the given tree could be any binary tree? Would your previous solution still work?

Note:

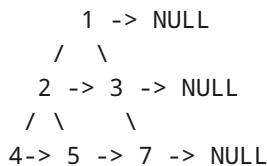
You may only use constant extra space.

For example,

Given the following binary tree,



After calling your function, the tree should look like:



这道是之前那道Populating Next Right Pointers in Each Node 每个节点的右向指针的延续，原本的完全二叉树的条件不再满足，但是整体的思路还是很相似，仍然有递归和非递归的解法。我们先来看递归的解法，这里由于子树有可能残缺，故需要平行扫描父节点同层的节点，找到他们的左右子节点。代码如下：

解法1：

```

1 class Solution {
2 public:
3     void connect(TreeLinkNode *root) {
4         if (!root) return;
5         TreeLinkNode *p = root->next;
6         while (p) {
7             if (p->left) {
8                 p = p->left;
9                 break;
10            }
11            if (p->right) {
12                p = p->right;
13                break;
14            }
15            p = p->next;
16        }
17        if (root->right) root->right->next = p;
18        if (root->left) root->left->next = root->right ? root->right : p;
19        connect(root->right);
20        connect(root->left);
21    }
22 };

```

对于非递归的方法，我惊喜的发现之前的方法直接就能用，完全不需要做任何修改，算法思路可参见之前的博客Populating Next Right Pointers in Each Node 每个节点的右向指针，代码如下：

解法2：

```

1 class Solution {
2 public:
3     void connect(TreeLinkNode *root) {
4         if (!root) return;
5         queue<TreeLinkNode*> q;
6         q.push(root);
7         while (!q.empty()) {
8             int len = q.size();
9             for (int i = 0; i < len; ++i) {
10                 TreeLinkNode *t = q.front(); q.pop();
11                 if (i < len - 1) t->next = q.front();
12                 if (t->left) q.push(t->left);
13                 if (t->right) q.push(t->right);
14             }
15         }
16     }
17 };

```

虽然以上的两种方法都能通过OJ，但其实它们都不符合题目的要求，题目说只能使用constant space，可是OJ却没有写专门检测space使用情况的test，那么下面贴上constant space的解法，这个解法也是用的层序遍历，只不过没有使用queue了，我们建立一个dummy结点来指向每层的首结点的前一个结点，然后指针t用来遍历这一层，我们实际上是遍历一层，然后连下一层的next，首先从根结点开始，如果左子结点存在，那么t的next连上左子结点，然后t指向其next指针；如果root的右子结点存在，那么t的next连上右子结点，然后t指向其next指针。此时root的左右子结点都连上了，此时root向右平移一位，指向其next指针，如果此时root不存在了，说明当前层已经遍历完了，我们重置t为dummy结点，root此时为dummy→next，即下一层的首结点，然后dummy的next指针清空，或者也可以将t的next指针清空，因为前面已经将t赋值为dummy了。那么现在想一想，为什么要清空？因为我们用dummy的目的就是要直到下一行的首结点的位置即dummy→next，而一旦将root赋值为

dummy→next了之后，这个dummy的使命就已经完成了，必须要断开，如果不斷开的话，那么假设现在root是叶结点了，那么while循环还会执行，不会进入前两个if，然后root右移赋空之后，会进入最后一个if，之前没有断开dummy→next的话，那么root又指向之前的叶结点了，死循环诞生了，跪了。所以一定要记得清空哦，呵呵哒~代码如下：

解法3：

```

1 class Solution {
2 public:
3     void connect(TreeLinkNode *root) {
4         TreeLinkNode *dummy = new TreeLinkNode(0), *t = dummy;
5         while (root) {
6             if (root->left) {
7                 t->next = root->left;
8                 t = t->next;
9             }
10            if (root->right) {
11                t->next = root->right;
12                t = t->next;
13            }
14            root = root->next;
15            if (!root) {
16                t = dummy;
17                root = dummy->next;
18                dummy->next = NULL;
19            }
20        }
21    }
22 }

```

CPP

118. 杨辉三角

Given numRows, generate the first numRows of Pascal's triangle.

For example, given numRows = 5,
Return

```
[
    [1],
    [1,1],
    [1,2,1],
    [1,3,3,1],
    [1,4,6,4,1]
]
```

杨辉三角是二项式系数的一种写法，如果熟悉杨辉三角的五个性质，那么很好生成，可参见我的上一篇博文：

<http://www.cnblogs.com/grandyang/p/4031536.html>

具体生成算是：每一行的首个和结尾一个数字都是1，从第三行开始，中间的每个数字都是上一行的左右两个数字之和。代码如下：

```

1 class Solution {
2 public:
3     vector<vector<int>> generate(int numRows) {
4         vector<vector<int>> res;
5         if (numRows <= 0) return res;
6         res.assign(numRows, vector<int>(1));
7         for (int i = 0; i < numRows; ++i) {
8             res[i][0] = 1;
9             if (i == 0) continue;
10            for (int j = 1; j < i; ++j) {
11                res[i].push_back(res[i-1][j] + res[i-1][j-1]);
12            }
13            res[i].push_back(1);
14        }
15        return res;
16    }
17 };

```

119. 杨辉三角之二

Given an index k, return the kth row of the Pascal's triangle.

For example, given k = 3,
Return [1,3,3,1].

Note:

Could you optimize your algorithm to use only O(k) extra space?

杨辉三角想必大家并不陌生，应该最早出现在初高中的数学中，其实就是二项式系数的一种写法。

```

      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
 1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1

```

杨辉三角形第n层（顶层称第0层，第1行，第n层即第n+1行，此处n为包含0在内的自然数）正好对应于二项式 $(a+b)^n$ 展开的系数。例如第二层1 2 1是幂指数为2的二项式 $(a+b)^2$ 展开形式 $a^2 + 2ab + b^2$ 的系数。 =

杨辉三角主要有下列五条性质：

杨辉三角以正整数构成，数字左右对称，每行由1开始逐渐变大，然后变小，回到1。

第n行的数字个数为n个。

第n行的第k个数字为组合数 C_{n-1}^{k-1} 。

第n行数字和为 2^{n-1} 。

除每行最左侧与最右侧的数字以外，每个数字等于它的左上方与右上方两个数字之和（也就是说，第n行第k个数字等于第n-1行的第k-1个数字与第k个数字的和）。这是因为有组合恒等式： $C_n^i = C_{n-1}^{i-1} + C_{n-1}^i$ 。可用此性质写出整个杨辉三角形。 +

由于题目有额外限制条件，程序只能使用O(k)的额外空间，那么这样就不能把每行都算出来，而是要用其他的方法，我最先考虑用的是第三条性质，算出每个组合数来生成第n行系数，代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<int> getRow(int rowIndex) {
4         vector<int> out;
5         if (rowIndex < 0) return out;
6
7         for (int i = 0; i <= rowIndex; ++i) {
8             if (i == 0 || i == rowIndex)
9                 out.push_back(1);
10            else
11                out.push_back (computeCnk(rowIndex, i));
12        }
13        return out;
14    }
15
16    int computeCnk(int n, int k) {
17        if (k > n) return 0;
18        else if (k > n/2) k = n - k;
19        int numerator = 1, denominator = 1;
20        for (int i = 0; i < k; ++i) {
21            numerator *= n - i;
22            denominator *= k - i;
23        }
24        if (denominator != 0) return numerator/denominator;
25        else return 0;
26    }
27};

```

本地调试输出前十行，没啥问题，拿到OJ上测试，程序在第18行跪了，中间有个系数不正确。那么问题出在哪了呢，仔细找找，原来出在计算组合数那里，由于算组合数时需要算连乘，而整形数int的数值范围只有-32768到32768之间，那么一旦n值过大，连乘肯定无法计算。而丧心病狂的OJ肯定会测试到成百上千行，所以这个方法不行。那么我们再来考虑利用第五条性质，除了第一个和最后一个数字之外，其他的数字都是上一行左右两个值之和。那么我们只需要两个for循环，除了第一个数为1之外，后面的数都是上一次循环的数值加上它前面位置的数值之和，不停地更新每一个位置的值，便可以得到第n行的数字，具体实现代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<int> getRow(int rowIndex) {
4         vector<int> out;
5         if (rowIndex < 0) return out;
6
7         out.assign(rowIndex + 1, 0);
8         for (int i = 0; i <= rowIndex; ++i) {
9             if (i == 0) {
10                 out[0] = 1;
11                 continue;
12             }
13             for (int j = rowIndex; j >= 1; --j) {
14                 out[j] = out[j] + out[j-1];
15             }
16         }
17         return out;
18     }
19 };

```

120. 三角形

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.

For example, given the following triangle

```
[
    [2],
    [3,4],
    [6,5,7],
    [4,1,8,3]
]
```

The minimum path sum from top to bottom is 11 (i.e., $2 + 3 + 5 + 1 = 11$).

Note:

Bonus point if you are able to do this using only $O(n)$ extra space, where n is the total number of rows in the triangle.

这道题和Dungeon Game 地牢游戏非常的类似，都是用动态规划Dynamic Programming来求解的问题。而且递推式也比较容易看出来，我最先想到的方法是：

从第二行开始， $\text{triangle}[i][j] = \min(\text{triangle}[i - 1][j - 1], \text{triangle}[i - 1][j])$ ，然后两边的数字直接赋值上一行的边界值，由于限制了空间复杂度，所以我干脆直接就更新triangle数组，代码如下：

解法1：

```

1 class Solution {
2 public:
3     int minimumTotal(vector<vector<int> > &triangle) {
4         int n = triangle.size();
5         for (int i = 1; i < n; ++i) {
6             for (int j = 0; j < triangle[i].size(); ++j) {
7                 if (j == 0) triangle[i][j] += triangle[i - 1][j];
8                 else if (j == triangle[i].size() - 1) triangle[i][j] += triangle[i - 1][j - 1];
9             }
10            else {
11                triangle[i][j] += min(triangle[i - 1][j - 1], triangle[i - 1][j]);
12            }
13        }
14    }
15    int res = triangle[n - 1][0];
16    for (int i = 0; i < triangle[n - 1].size(); ++i) {
17        res = min(res, triangle[n - 1][i]);
18    }
19    return res;
20 }
21 };

```

这种方法可以通过OJ，但是毕竟修改了原始数组triangle，并不是很理想的方法。在网上搜到一种更好的DP方法，这种方法复制了三角形最后一行，作为用来更新的一位数组。然后逐个遍历这个DP数组，对于每个数字，和它之后的元素比较选择较小的再加上上面一行相邻位置的元素做为新的元素，然后一层一层的向上扫描，整个过程和冒泡排序的原理差不多，最后最小的元素都冒到前面，第一个元素即为所求。代码如下：

解法2：

```

1 class Solution {
2 public:
3     int minimumTotal(vector<vector<int> > &triangle) {
4         int n = triangle.size();
5         vector<int> dp(triangle.back());
6         for (int i = n - 2; i >= 0; --i) {
7             for (int j = 0; j <= i; ++j) {
8                 dp[j] = min(dp[j], dp[j + 1]) + triangle[i][j];
9             }
10        }
11        return dp[0];
12    }
13 };

```

下面我们来看一个例子，对于输入数组：

```
-1
2 3
1 -1 -3
5 3 -1 2
```

下面我们来看DP数组的变换过程。

```
DP: 5 3 -1 2
DP: 4 3 -1 2
DP: 4 -2 -1 2
DP: 4 -2 -4 2
DP: 0 -2 -4 2
DP: 0 -1 -4 2
DP: -2 -1 -4 2
```

121. 买卖股票的最佳时间

Say you have an array for which the i th element is the price of a given stock on day i .

If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit.

这道题相当简单，感觉达不到Medium的难度，只需要遍历一次数组，用一个变量记录遍历过数中的最小值，然后每次计算当前值和这个最小值之间的差值最为利润，然后每次选较大的利润来更新。当遍历完成后当前利润即为所求，代码如下：

```
1 class Solution {
2 public:
3     int maxProfit(vector<int>& prices) {
4         int res = 0, buy = INT_MAX;
5         for (int price : prices) {
6             buy = min(buy, price);
7             res = max(res, price - buy);
8         }
9         return res;
10    }
11};
```

CPP

122. 买股票的最佳时间之二

Say you have an array for which the i th element is the price of a given stock on day i .

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times). However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

这道跟之前那道Best Time to Buy and Sell Stock 买卖股票的最佳时间很类似，但都比较容易解答。这道题由于可以无限次买入和卖出。我们都知道炒股想挣钱当然是低价买入高价抛出，那么这里我们只需要从第二天开始，如果当前价格比之前价格高，则把差值加入利润中，因为我们可以昨天买入，今日卖出，若明日价更高的话，还可以今日买入，明日再抛出。以此类推，遍历完整个数组后即可求得最大利润。代码如下：

```

1 class Solution {
2 public:
3     int maxProfit(vector<int>& prices) {
4         int res = 0, n = prices.size();
5         for (int i = 0; i < n - 1; ++i) {
6             if (prices[i] < prices[i + 1]) {
7                 res += prices[i + 1] - prices[i];
8             }
9         }
10        return res;
11    }
12 };

```

CPP

123. 买股票的最佳时间之三

Say you have an array for which the i th element is the price of a given stock on day i .

Design an algorithm to find the maximum profit. You may complete at most two transactions.

Note:

You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

这道是买股票的最佳时间系列问题中最难最复杂的一道，前面两道Best Time to Buy and Sell Stock 买卖股票的最佳时间和Best Time to Buy and Sell Stock II 买股票的最佳时间之二的思路都非常的简洁明了，算法也很简单。而这道是要求最多交易两次，找到最大利润，还是需要用动态规划Dynamic Programming来解，而这里我们需要两个递推公式来分别更新两个变量local和global，参见网友Code Ganker的博客，我们其实可以求至少 k 次交易的最大利润，找到通解后可以设定 $k = 2$ ，即为本题的解答。我们定义 $local[i][j]$ 为在到达第 i 天时最多可进行 j 次交易并且最后一次交易在最后一天卖出的最大利润，此为局部最优。然后我们定义 $global[i][j]$ 为在到达第 i 天时最多可进行 j 次交易的最大利润，此为全局最优。它们的递推式为：

```

local[i][j] = max(global[i - 1][j - 1] + max(diff, 0), local[i - 1][j] + diff)

global[i][j] = max(local[i][j], global[i - 1][j])

```

其中局部最优值是比较前一天并少交易一次的全局最优加上大于0的差值，和前一天的局部最优加上差值中取较大值，而全局最优比较局部最优和前一天的全局最优。代码如下：

解法1：

```

1 class Solution {
2 public:
3     int maxProfit(vector<int> &prices) {
4         if (prices.empty()) return 0;
5         int n = prices.size(), g[n][3] = {0}, l[n][3] = {0};
6         for (int i = 1; i < prices.size(); ++i) {
7             int diff = prices[i] - prices[i - 1];
8             for (int j = 1; j <= 2; ++j) {
9                 l[i][j] = max(g[i - 1][j - 1] + max(diff, 0), l[i - 1][j] + diff);
10                g[i][j] = max(l[i][j], g[i - 1][j]);
11            }
12        }
13        return g[n - 1][2];
14    }
15 };

```

下面这种解法用一维数组来代替二维数组，可以极大的节省了空间，由于覆盖的顺序关系，我们需要j从2到1，这样可以取到正确的g[j-1]值，而非已经被覆盖过的值，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int maxProfit(vector<int> &prices) {
4         if (prices.empty()) return 0;
5         int g[3] = {0};
6         int l[3] = {0};
7         for (int i = 0; i < prices.size() - 1; ++i) {
8             int diff = prices[i + 1] - prices[i];
9             for (int j = 2; j >= 1; --j) {
10                 l[j] = max(g[j - 1] + max(diff, 0), l[j] + diff);
11                 g[j] = max(l[j], g[j]);
12             }
13         }
14         return g[2];
15     }
16 };

```

我们如果假设prices数组为1, 3, 2, 9, 那么我们来看每次更新时local 和 global 的值:

第一天两次交易: 第一天一次交易:

local: 0 0 0

local: 0 0 0

global: 0 0 0

global: 0 0 0

第二天两次交易:

第二天一次交易:

local: 0 0 2

local: 0 2 2

global: 0 0 2

global: 0 2 2

第三天两次交易:

第三天一次交易:

local: 0 2 2

local: 0 1 2

global: 0 2 2

global: 0 2 2

第四天两次交易:

第四天一次交易:

local: 0 1 9

local: 0 8 9

global: 0 2 9

global: 0 8 9

在网友@loveahnee的提醒下, 发现了其实上述的递推公式关于 $local[i][j]$ 的可以稍稍化简一下, 我们之前定义的 $local[i][j]$ 为在到达第*i*天时最多可进行*j*次交易并且最后一次交易在最后一天卖出的最大利润, 然后网友@fgvlt解释了一下第 *i* 天卖第 *j* 支股票的话, 一定是下面的一种:

1. 今天刚买的

那么 $Local(i, j) = Global(i-1, j-1)$

相当于啥都没干

2. 昨天买的

那么 $Local(i, j) = Global(i-1, j-1) + diff$

等于 $Global(i-1, j-1)$ 中的交易, 加上今天干的那一票

3. 更早之前买的

那么 $Local(i, j) = Local(i-1, j) + diff$

昨天别卖了, 留到今天卖

但其实第一种情况是不需要考虑的, 因为当天买当天卖不会增加利润, 完全是重复操作, 这种情况可以归纳在 $global[i-1][j-1]$ 中, 所以我们就不需要 $\max(0, diff)$ 了, 那么由于两项都加上了 $diff$, 所以我们可以把 $diff$ 抽到 \max 的外面, 所以更新后的递推公式为:

```
local[i][j] = max(global[i - 1][j - 1], local[i - 1][j]) + diff
```

```
global[i][j] = max(local[i][j], global[i - 1][j])
```

124. 求二叉树的最大路径和

Given a binary tree, find the maximum path sum.

The path may start and end at any node in the tree.

For example:

Given the below binary tree,



Return 6.

这道求二叉树的最大路径和是一道蛮有难度的题，难就难在起始位置和结束位置可以为任意位置，我当然是又不会了，于是上网看看大神们的解法，像这种类似数的遍历的题，一般来说都需要用DFS来求解，我们先来看一个简单的例子：



由于这是一个很简单的例子，我们很容易就能找到最长路径为7-11-4-13，那么怎么用递归来找出正确的路径和呢？根据以往的经验，树的递归解法一般都是递归到叶节点，然后开始边处理边回溯到根节点。那么我们就假设此时已经递归到结点7了，那么其没有左右子节点，所以如果以结点7为根结点的子树最大路径和就是7。然后回溯到结点11，如果以结点11为根结点的子树，我们知道最大路径和为 $7+11+2=20$ 。但是当回溯到结点4的时候，对于结点11来说，就不能同时取两条路径了，只能取左路径，或者是右路径，所以当根结点是4的时候，那么结点11只能取其左子结点7，因为7大于2。所以，对于每个结点来说，我们要知道经过其左子结点的path之和大还是经过右子结点的path之和大。那么我们的递归函数返回值就可以定义为以当前结点为根结点，到叶节点的最大路径之和，然后全局路径最大值放在参数中，用结果res来表示。

在递归函数中，如果当前结点不存在，那么直接返回0。否则就分别对其左右子节点调用递归函数，由于路径和有可能为负数，而我们当然不希望加上负的路径和，所以我们和0相比，取较大的那个，就是要不加，加就要加正数。然后我们来更新全局最大值结果res，就是以左子结点为终点的最大path之和加上以右子结点为终点的最大path之和，还要加上当前结点值，这样就组成了一个完整的路径。而我们返回值是取left和right中的较大值加上当前结点值，因为我们返回值的定义是以当前结点为终点的path之和，所以只能取left和right中较大的那个值，而不是两个都要，参见代码如下：

```

1 class Solution {
2 public:
3     int maxPathSum(TreeNode* root) {
4         int res = INT_MIN;
5         helper(root, res);
6         return res;
7     }
8     int helper(TreeNode* node, int& res) {
9         if (!node) return 0;
10        int left = max(helper(node->left, res), 0);
11        int right = max(helper(node->right, res), 0);
12        res = max(res, left + right + node->val);
13        return max(left, right) + node->val;
14    }
15 }

```

CPP

讨论：这道题有一个很好的Follow up，就是返回这个最大路径，那么就复杂很多，因为我们的递归函数就不能返回路径和了，而是返回该路径上所有的结点组成的数组，递归的参数还要保留最大路径之和，同时还需要最大路径结点的数组，然后对左右子节点调用递归函数后得到的是数组，我们要统计出数组之和，并且跟0比较，如果小于0，和清零，数组清空。然后就是更新最大路径之和跟数组啦，还要拼出来返回值数组，代码长了很多，有兴趣的童鞋可以在评论区贴上你的代码~

125. 验证回文字符串

Given a string, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.

For example,
 "A man, a plan, a canal: Panama" is a palindrome.
 "race a car" is not a palindrome.

Note:

Have you consider that the string might be empty? This is a good question to ask during an interview.

For the purpose of this problem, we define empty string as valid palindrome.

验证回文字符串是比较常见的问题，所谓回文，就是一个正读和反读都一样的字符串，比如“level”或者“noon”等等就是回文串。但是这里，加入了空格和非字母数字的字符，增加了些难度，但其实原理还是很简单：只需要建立两个指针，left和right，分别从字符的开头和结尾处开始遍历整个字符串，如果遇到非字母数字的字符就跳过，继续往下找，直到找到下一个字母数字或者结束遍历，如果遇到大写字母，就将其转为小写。等左右指针都找到字母数字时，比较这两个字符，若相等，则继续比较下面两个分别找到的字母数字，若不相等，直接返回false。

时间复杂度为O(n)，代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool isPalindrome(string s) {
4         int left = 0, right = s.size() - 1;
5         while (left < right) {
6             if (!isAlphaNum(s[left])) ++left;
7             else if (!isAlphaNum(s[right])) --right;
8             else if ((s[left] + 32 - 'a') % 32 != (s[right] + 32 - 'a') % 32) return false;
9             else {
10                 ++left; --right;
11             }
12         }
13         return true;
14     }
15     bool isAlphaNum(char &ch) {
16         if (ch >= 'a' && ch <= 'z') return true;
17         if (ch >= 'A' && ch <= 'Z') return true;
18         if (ch >= '0' && ch <= '9') return true;
19         return false;
20     }
21 };

```

CPP

我们也可以用系统自带的判断是否是数母字符的判断函数isalnum，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool isPalindrome(string s) {
4         int left = 0, right = s.size() - 1;
5         while (left < right) {
6             if (!isalnum(s[left])) ++left;
7             else if (!isalnum(s[right])) --right;
8             else if ((s[left] + 32 - 'a') % 32 != (s[right] + 32 - 'a') % 32) return false;
9             else {
10                 ++left; --right;
11             }
12         }
13         return true;
14     }
15 };

```

对于该问题的扩展，还有利用Manacher算法来求解最长回文字符串问题，参见我的另一篇博文Manacher's Algorithm 马拉车算法。

126. 词语阶梯之二

Given two words (start and end), and a dictionary, find all shortest transformation sequence(s) from start to end, such that:

Only one letter can be changed at a time
 Each intermediate word must exist in the dictionary
 For example,

Given:

```

start = "hit"
end = "cog"
dict = ["hot", "dot", "dog", "lot", "log"]

```

Return

```

[
    ["hit", "hot", "dot", "dog", "cog"],
    ["hit", "hot", "lot", "log", "cog"]
]

```

Note:

All words have the same length.

All words contain only lowercase alphabetic characters.

个人感觉这道题是相当有难度的一道题，它比之前那道Word Ladder 词语阶梯要复杂很多，全场第四低的通过率12.9%正说明了这道题的难度，我也是研究了网上别人的解法很久才看懂，然后照葫芦画瓢的写了出来，下面这种解法的核心思想是BFS，大概思路如下：我们的目的是找出所有的路径，我们建立一个路径集paths，用以保存所有路径，然后是起始路径p，在p中先把起始单词放进去。然后定义两个整型变量level，和minLevel，其中level是记录循环中当前路径的长度，minLevel是记录最短路径的长度，这样的好处是，如果某条路径的长度超过了已有的最短路径的长度，那么舍弃，这样会提高运行速度，相当于一种剪枝。还要定义一个set变量words，用来记录已经循环过的路径中的词，然后就是BFS的核心了，循环路径集paths里的内容，取出队首路径，如果该路径长度大于level，说明字典中的有些词已经存入路径了，如果在路径中重复出现，则肯定不是最短路径，所以我们需要在字典中将这些词删去，然后将words清空，对循环对剪枝处理。然后我们取出当前路径的最后一个词，对每个字母进行替换并在字典中查找是否存在替换后的新词，这个过程在之前那道Word Ladder 词语阶梯里面也有。如果替换后的新词在字典中存在，将其加入words中，并在原有路径的基础上加上这个新词生成一条新路径，如果这个新词就是结束词，则此新路径为一条完整的路径，加入结果中，并更新minLevel，若不是结束词，解将新路径加入路径集中继续循环。写了这么多，不知道你看晕了没有，还是看代码吧，这个最有效：

```
1 class Solution {
2 public:
3     vector<vector<string>> findLadders(string beginWord, string endWord, vector<string>&
4 wordList) {
5         vector<vector<string>> res;
6         unordered_set<string> dict(wordList.begin(), wordList.end());
7         vector<string> p{beginWord};
8         queue<vector<string>> paths;
9         paths.push(p);
10        int level = 1, minLevel = INT_MAX;
11        unordered_set<string> words;
12        while (!paths.empty()) {
13            auto t = paths.front(); paths.pop();
14            if (t.size() > level) {
15                for (string w : words) dict.erase(w);
16                words.clear();
17                level = t.size();
18                if (level > minLevel) break;
19            }
20            string last = t.back();
21            for (int i = 0; i < last.size(); ++i) {
22                string newLast = last;
23                for (char ch = 'a'; ch <= 'z'; ++ch) {
24                    newLast[i] = ch;
25                    if (!dict.count(newLast)) continue;
26                    words.insert(newLast);
27                    vector<string> nextPath = t;
28                    nextPath.push_back(newLast);
29                    if (newLast == endWord) {
30                        res.push_back(nextPath);
31                        minLevel = level;
32                    } else paths.push(nextPath);
33                }
34            }
35        }
36        return res;
37    }
38};
```

127. 词语阶梯

Given two words (beginWord and endWord), and a dictionary's word list, find the length of shortest transformation sequence from beginWord to endWord, such that:

Only one letter can be changed at a time.

Each transformed word must exist in the word list. Note that beginWord is not a transformed word.

Note:

Return 0 if there is no such transformation sequence.

All words have the same length.

All words contain only lowercase alphabetic characters.

You may assume no duplicates in the word list.

You may assume beginWord and endWord are non-empty and are not the same.

Example 1:

Input:

```
beginWord = "hit",
endWord = "cog",
wordList = ["hot", "dot", "dog", "lot", "log", "cog"]
```

Output: 5

Explanation: As one shortest transformation is "hit" -> "hot" -> "dot" -> "dog" -> "cog", return its length 5.

Example 2:

Input:

```
beginWord = "hit"
endWord = "cog"
wordList = ["hot", "dot", "dog", "lot", "log"]
```

Output: 0

Explanation: The endWord "cog" is not in wordList, therefore no possible transformation.

这道词句阶梯的问题给了我们一个单词字典，里面有一系列很相似的单词，然后给了一个起始单词和一个结束单词，每次变换只能改变一个单词，并且中间过程的单词都必须是单词字典中的单词，让我们求出最短的变化序列的长度。这道题还是挺有难度的，我当然是看了别人的解法才写出来的，这没啥的，从不会到完全掌握才是成长嘛~

当拿到题就懵逼的我们如何才能找到一个科学的探索解题的路径呢，那就是先别去管代码实现，如果让我们肉身解题该怎么做呢？让你将 'hit' 变为 'cog'，那么我们发现这两个单词没有一个相同的字母，所以我们就尝试呗，博主会先将第一个 'h' 换成 'c'，看看 'cit' 在不在字典中，发现不在，那么把第二个 'i' 换成 'o'，看看 'hot' 在不在，发现在，完美！然后尝试 'cot' 或者 'hog'，发现都不在，那么就比较麻烦了，我们没法快速的达到目标单词，需要一些中间状态，但我们怎么知道中间状态是什么。简单粗暴的方法就是brute force，遍历所有的情况，我们将起始单词的每一个字母都用26个字母来替换，比如起始单词 'hit' 就要替换为 'ait', 'bit', 'cit', 'yit', 'zit'，将每个替换成的单词都在字典中查找一下，如果说有的话，那么说明可能是潜在的路径，要保存下来。那么现在就有个问题，比如我们换到了 'hot' 的时候，此时发现在字典中存在，那么下一步我们是继续试接下来的 'hpt', 'hqt', 'hrt'... 还是直接从 'hot' 的首字母开始换 'aot', 'bot', 'cot' ... 这实际上就是BFS和DFS的区别，到底是广度优先，还是深度优先。讲到这里，不知道你有没有觉得这个跟什么很像？对了，跟迷宫遍历很像啊，你想啊，迷宫中每个点有上下左右四个方向可以走，而这里有26个字母，就是二十六个方向可以走，本质上没有啥区别啊！如果熟悉迷宫遍历的童鞋们应该知道，应该用BFS来求最短路径的长度，这也不难理解啊，DFS相当于一条路走到黑啊，你走的那条道不一定是最短的啊。而BFS相当于一个小圈慢慢的一层一层扩大，相当于往湖里扔个石头，一圈一圈扩大的水波纹那种感觉，当水波纹碰到湖上的树叶时，那么此时水圈的半径就是圆心到树叶的最短距离。脑海中有没有浮现出这个生动的场景呢？

明确了要用BFS，我们可以开始解题了，为了提到字典的查找效率，我们使用HashSet保存所有的单词。然后我们需要一个HashMap，来建立某条路径结尾单词和该路径长度之间的映射，并把起始单词映射为1。既然是BFS，我们需要一个队列queue，把起始单词排入队列中，开始队列的循环，取出队首词，然后对其每个位置上的字符，用26个字母进行替换，如果此时和结尾单词相同了，就可以返回取出词在哈希表中的值加一。如果替换词在字典中存在但在哈希表中不存在，则将替换词排入队列中，并在哈希表中的值映射为之前取出词加一。如果循环完成则返回0，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int ladderLength(string beginWord, string endWord, vector<string>& wordList) {
4         unordered_set<string> wordSet(wordList.begin(), wordList.end());
5         unordered_map<string, int> pathCnt{{beginWord, 1}};
6         queue<string> q{{beginWord}};
7         while (!q.empty()) {
8             string word = q.front(); q.pop();
9             for (int i = 0; i < word.size(); ++i) {
10                 string newWord = word;
11                 for (char ch = 'a'; ch <= 'z'; ++ch) {
12                     newWord[i] = ch;
13                     if (wordSet.count(newWord) && newWord == endWord) return pathCnt[word]
14 + 1;
15                     if (wordSet.count(newWord) && !pathCnt.count(newWord)) {
16                         q.push(newWord);
17                         pathCnt[newWord] = pathCnt[word] + 1;
18                     }
19                 }
20             }
21         }
22     }
23     return 0;
24 };

```

CPP

其实我们并不需要上面解法中的HashMap，由于BFS的遍历机制就是一层一层的扩大的，那么我们只要记住层数就行，然后在while循环中使用一个小trick，加一个for循环，表示遍历完当前队列中的个数后，层数就自增1，这样的话我们就省去了HashMap，而仅仅用一个变量res来记录层数即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int ladderLength(string beginWord, string endWord, vector<string>& wordList) {
4         unordered_set<string> wordSet(wordList.begin(), wordList.end());
5         queue<string> q{{beginWord}};
6         int res = 0;
7         while (!q.empty()) {
8             for (int k = q.size(); k > 0; --k) {
9                 string word = q.front(); q.pop();
10                if (word == endWord) return res + 1;
11                for (int i = 0; i < word.size(); ++i) {
12                    string newWord = word;
13                    for (char ch = 'a'; ch <= 'z'; ++ch) {
14                        newWord[i] = ch;
15                        if (wordSet.count(newWord) && newWord != word) {
16                            q.push(newWord);
17                            wordSet.erase(newWord);
18                        }
19                    }
20                }
21            }
22            ++res;
23        }
24        return 0;
25    }
26};

```

128. 求最长连续序列

Given an unsorted array of integers, find the length of the longest consecutive elements sequence.

For example,

Given [100, 4, 200, 1, 3, 2],

The longest consecutive elements sequence is [1, 2, 3, 4]. Return its length: 4.

Your algorithm should run in O(n) complexity.

这道题要求求最长连续序列，并给定了O(n)复杂度限制，我们的思路是，使用一个集合set存入所有的数字，然后遍历数组中的每个数字，如果其在集合中存在，那么将其移除，然后分别用两个变量pre和next算出其前一个数跟后一个数，然后在集合中循环查找，如果pre在集合中，那么将pre移除集合，然后pre再自减1，直至pre不在集合之中，对next采用同样的方法，那么next-pre-1就是当前数字的最长连续序列，更新res即可。代码如下：

解法1：

```

1 class Solution {
2 public:
3     int longestConsecutive(vector<int>& nums) {
4         int res = 0;
5         unordered_set<int> s(nums.begin(), nums.end());
6         for (int val : nums) {
7             if (!s.count(val)) continue;
8             s.erase(val);
9             int pre = val - 1, next = val + 1;
10            while (s.count(pre)) s.erase(pre--);
11            while (s.count(next)) s.erase(next++);
12            res = max(res, next - pre - 1);
13        }
14        return res;
15    }
16 };

```

我们也可以采用哈希表来做，刚开始哈希表为空，然后遍历所有数字，如果该数字不在哈希表中，那么我们分别看其左右两个数字是否在哈希表中，如果在，则返回其哈希表中映射值，若不在，则返回0，然后我们将left+right+1作为当前数字的映射，并更新res结果，然后更新d-left和d-right的映射值，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int longestConsecutive(vector<int>& nums) {
4         int res = 0;
5         unordered_map<int, int> m;
6         for (int d : nums) {
7             if (!m.count(d)) {
8                 int left = m.count(d - 1) ? m[d - 1] : 0;
9                 int right = m.count(d + 1) ? m[d + 1] : 0;
10                int sum = left + right + 1;
11                m[d] = sum;
12                res = max(res, sum);
13                m[d - left] = sum;
14                m[d + right] = sum;
15            }
16        }
17        return res;
18    }
19 };

```

129. 求根到叶节点数字之和

Given a binary tree containing digits from 0-9 only, each root-to-leaf path could represent a number.

An example is the root-to-leaf path 1->2->3 which represents the number 123.

Find the total sum of all root-to-leaf numbers.

For example,



The root-to-leaf path 1->2 represents the number 12.

The root-to-leaf path 1->3 represents the number 13.

Return the sum = 12 + 13 = 25.

这道求根到叶节点数字之和的题跟之前的求Path Sum 二叉树的路径和很类似，都是利用DFS递归来解，这道题由于不是单纯的把各个节点的数字相加，而是每到一个新的数字，要把原来的数字扩大10倍之后再相加。代码如下：

```

1 class Solution {
2 public:
3     int sumNumbers(TreeNode *root) {
4         return sumNumbersDFS(root, 0);
5     }
6     int sumNumbersDFS(TreeNode *root, int sum) {
7         if (!root) return 0;
8         sum = sum * 10 + root->val;
9         if (!root->left && !root->right) return sum;
10        return sumNumbersDFS(root->left, sum) + sumNumbersDFS(root->right, sum);
11    }
12 };

```

130. 包围区域

Given a 2D board containing 'X' and 'O', capture all regions surrounded by 'X'.

A region is captured by flipping all 'O's into 'X's in that surrounded region.

For example,

```

X X X X
X O O X
X X O X
X O X X

```

After running your function, the board should be:

```

X X X X
X X X X
X X X X
X O X X

```

这道题有点像围棋，将包住的O都变成X，但不同的是边缘的O不算被包围，跟之前那道Number of Islands 岛屿的数量很类似，都可以用DFS来解。刚开始我的思路是DFS遍历中间的O，如果没有到达边缘，都变成X，如果到达了边缘，将之前变成X的再变回来。但是这样做非常的不方便，在网上看到大家普遍的做法是扫面矩阵的四条边，如果有O，则用DFS遍历，将所有连着的O都变成另一个字符，比如\$，这样剩下的O都是被包围的，然后将这些O变成X，把\$变回O就行了。代码如下：

解法1：

```

1 class Solution {
2 public:
3     void solve(vector<vector<char>>& board) {
4         for (int i = 0; i < board.size(); ++i) {
5             for (int j = 0; j < board[i].size(); ++j) {
6                 if ((i == 0 || i == board.size() - 1 || j == 0 || j == board[i].size() - 1)
7 && board[i][j] == '0')
8                     solveDFS(board, i, j);
9             }
10        }
11        for (int i = 0; i < board.size(); ++i) {
12            for (int j = 0; j < board[i].size(); ++j) {
13                if (board[i][j] == '0') board[i][j] = 'X';
14                if (board[i][j] == '$') board[i][j] = 'O';
15            }
16        }
17    }
18    void solveDFS(vector<vector<char>> &board, int i, int j) {
19        if (board[i][j] == '0') {
20            board[i][j] = '$';
21            if (i > 0 && board[i - 1][j] == '0')
22                solveDFS(board, i - 1, j);
23            if (j < board[i].size() - 1 && board[i][j + 1] == '0')
24                solveDFS(board, i, j + 1);
25            if (i < board.size() - 1 && board[i + 1][j] == '0')
26                solveDFS(board, i + 1, j);
27            if (j > 1 && board[i][j - 1] == '0')
28                solveDFS(board, i, j - 1);
29        }
30    }
31 };

```

有网友提问上面的代码中红色部分为啥是 $j > 1$ 而不是 $j > 0$ ，为啥 $j > 0$ 无法通过OJ的最后一个大数据集合，我开始也不知道其中奥秘，直到被另一个网友提醒在本地机子上可以通过最后一个大数据集合，于是我也写了一个程序来验证，请参见验证LeetCode Surrounded Regions 包围区域的DFS方法。发现 $j > 0$ 是正确的，可以得到相同的结果。

下面这种解法还是DFS解法，只是递归函数的写法稍有不同，但是本质上并没有太大的区别，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     void solve(vector<vector<char>>& board) {
4         if (board.empty() || board[0].empty()) return;
5         int m = board.size(), n = board[0].size();
6         for (int i = 0; i < m; ++i) {
7             for (int j = 0; j < n; ++j) {
8                 if (i == 0 || i == m - 1 || j == 0 || j == n - 1) {
9                     if (board[i][j] == 'O') dfs(board, i, j);
10                }
11            }
12        }
13        for (int i = 0; i < m; ++i) {
14            for (int j = 0; j < n; ++j) {
15                if (board[i][j] == 'O') board[i][j] = 'X';
16                if (board[i][j] == '$') board[i][j] = 'O';
17            }
18        }
19    }
20    void dfs(vector<vector<char>> &board, int x, int y) {
21        int m = board.size(), n = board[0].size();
22        vector<vector<int>> dir{{0,-1},{-1,0},{0,1},{1,0}};
23        board[x][y] = '$';
24        for (int i = 0; i < dir.size(); ++i) {
25            int dx = x + dir[i][0], dy = y + dir[i][1];
26            if (dx >= 0 && dx < m && dy > 0 && dy < n && board[dx][dy] == 'O') {
27                dfs(board, dx, dy);
28            }
29        }
30    }
31 };

```

我们也可以使用迭代的解法，但是整体的思路还是一样的，我们在找到边界上的O后，然后利用队列queue进行BFS查找和其相连的所有O，然后都标记上美元号。最后的处理还是先把所有的O变成X，然后再把美元号变回O即可，参见代码如下：

解法3：

```
1 class Solution {
2 public:
3     void solve(vector<vector<char>>& board) {
4         if (board.empty() || board[0].empty()) return;
5         int m = board.size(), n = board[0].size();
6         for (int i = 0; i < m; ++i) {
7             for (int j = 0; j < n; ++j) {
8                 if (i != 0 && i != m - 1 && j != 0 && j != n - 1) continue;
9                 if (board[i][j] != '0') continue;
10                board[i][j] = '$';
11                queue<int> q{{i * n + j}};
12                while (!q.empty()) {
13                    int t = q.front(), x = t / n, y = t % n; q.pop();
14                    if (x >= 1 && board[x - 1][y] == '0') {board[x - 1][y] = '$'; q.push(t - n);}
15                    if (x < m - 1 && board[x + 1][y] == '0') {board[x + 1][y] = '$'; q.push(t + n);}
16                    if (y >= 1 && board[x][y - 1] == '0') {board[x][y - 1] = '$'; q.push(t - 1);}
17                    if (y < n - 1 && board[x][y + 1] == '0') {board[x][y + 1] = '$';}
18                q.push(t + 1);}
19            }
20        }
21    }
22 }
23 }
24 }
25 }
26 }
27 }
28 }
```

131. 拆分回文串

Given a string s , partition s such that every substring of the partition is a palindrome.

Return all possible palindrome partitioning of s.

For example, given s = "aab",

Return

```
[  
  ["aa", "b"],  
  ["a", "a", "b"]  
]
```

这又是一道需要用DFS来解的题目，既然题目要求找到所有可能拆分成回文数的情况，那么肯定是所有的情况都要遍历到，对于每一个子字符串都要分别判断一次是不是回文数，那么肯定有一个判断回文数的子函数，还需要一个DFS函数用来递归，再加上原本的这个函数，总共需要三个函数来求解。代码如下：

```

1 class Solution {
2 public:
3     vector<vector<string>> partition(string s) {
4         vector<vector<string>> res;
5         vector<string> out;
6         partitionDFS(s, 0, out, res);
7         return res;
8     }
9     void partitionDFS(string s, int start, vector<string> &out, vector<vector<string>>
10 &res) {
11         if (start == s.size()) {
12             res.push_back(out);
13             return;
14         }
15         for (int i = start; i < s.size(); ++i) {
16             if (isPalindrome(s, start, i)) {
17                 out.push_back(s.substr(start, i - start + 1));
18                 partitionDFS(s, i + 1, out, res);
19                 out.pop_back();
20             }
21         }
22     }
23     bool isPalindrome(string s, int start, int end) {
24         while (start < end) {
25             if (s[start] != s[end]) return false;
26             ++start;
27             --end;
28         }
29         return true;
30     }
31 };

```

那么，对原字符串的所有子字符串的访问顺序是什么呢，如果原字符串是 abcd，那么访问顺序为: a → b → c → d → cd → bc → bcd → ab → abc → abcd，这是对于没有两个或两个以上子回文串的情况。那么假如原字符串是 aabc，那么访问顺序为: a → a → b → c → bc → ab → abc → aa → b → c → bc → aab → aabc，中间当检测到aa时候，发现是回文串，那么对于剩下的bc当做做一个新串来检测，于是有 b → c → bc，这样扫描了所有情况，即可得出最终答案。

132. 拆分回文串之二

Given a string s, partition s such that every substring of the partition is a palindrome.

Return the minimum cuts needed for a palindrome partitioning of s.

For example, given s = "aab",
Return 1 since the palindrome partitioning ["aa", "b"] could be produced using 1 cut.

这道题是让找到把原字符串拆分成回文串的最小切割数，需要用动态规划Dynamic Programming来做，使用DP的核心是在于找出递推公式，之前有道地牢游戏Dungeon Game的题也是需要用DP来做，而那道题是二维DP来解，这道题由于只是拆分一个字符串，需要一个一维的递推公式，我们还是从后往前推，递推公式为: $dp[i] = \min(dp[i], 1+dp[j+1]) \quad i \leq j < n$ ，那么还有个问题，是否对于i到j之间的子字符串s[i][j]每次都判断一下是否是回文串，其实这个也可以用DP来简化，其DP递推公式为P[i][j] = s[i] == s[j] && P[i+1][j-1]，其中P[i][j] = true if [i,j]为回文。代码如下：

```

1 class Solution {
2 public:
3     int minCut(string s) {
4         int len = s.size();
5         bool P[len][len];
6         int dp[len + 1];
7         for (int i = 0; i <= len; ++i) {
8             dp[i] = len - i - 1;
9         }
10        for (int i = 0; i < len; ++i) {
11            for (int j = 0; j < len; ++j) {
12                P[i][j] = false;
13            }
14        }
15        for (int i = len - 1; i >= 0; --i) {
16            for (int j = i; j < len; ++j) {
17                if (s[i] == s[j] && (j - i <= 1 || P[i + 1][j - 1])) {
18                    P[i][j] = true;
19                    dp[i] = min(dp[i], dp[j + 1] + 1);
20                }
21            }
22        }
23        return dp[0];
24    }
25 };

```

133. 无向图的复制

Clone an undirected graph. Each node in the graph contains a label and a list of its neighbors.

OJ's undirected graph serialization:
Nodes are labeled uniquely.

We use # as a separator for each node, and , as a separator for node label and each neighbor of the node.

As an example, consider the serialized graph {0,1,2#1,2#2,2}.

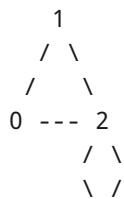
The graph has a total of three nodes, and therefore contains three parts as separated by #.

First node is labeled as 0. Connect node 0 to both nodes 1 and 2.

Second node is labeled as 1. Connect node 1 to node 2.

Third node is labeled as 2. Connect node 2 to node 2 (itself), thus forming a self-cycle.

Visually, the graph looks like the following:



这道无向图的复制问题和之前的拷贝带有随机指针的链表有些类似，那道题的难点是如何处理每个节点的随机指针，这道题目的难点在于如何处理每个节点的neighbors，由于在深度拷贝每一个节点后，还要将其所有neighbors放到一个vector中，而如何避免重复拷贝呢？这道题好就好在所有节点值不同，所以我们可以使用哈希表来对应节点值和新生成的节点。对于图的遍历的两大基本方法是深度优先搜索DFS和广度优先搜索BFS，此题的两种解法可参见网友爱做饭的小莹子的博客，这里我们使用深度优先搜索DFS来解答此题，代码如下：

```

1 class Solution {
2 public:
3     UndirectedGraphNode *cloneGraph(UndirectedGraphNode *node) {
4         unordered_map<int, UndirectedGraphNode*> umap;
5         return clone(node, umap);
6     }
7     UndirectedGraphNode *clone(UndirectedGraphNode *node, unordered_map<int,
8 UndirectedGraphNode*> &umap) {
9         if (!node) return node;
10        if (umap.count(node->label)) return umap[node->label];
11        UndirectedGraphNode *newNode = new UndirectedGraphNode(node->label);
12        umap[node->label] = newNode;
13        for (int i = 0; i < node->neighbors.size(); ++i) {
14            (newNode->neighbors).push_back(clone(node->neighbors[i], umap));
15        }
16        return newNode;
17    }
18 };

```

134. 加油站问题

There are N gas stations along a circular route, where the amount of gas at station i is gas[i].

You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from station i to its next station (i+1). You begin the journey with an empty tank at one of the gas stations.

Return the starting gas station's index if you can travel around the circuit once, otherwise return -1.

Note:

The solution is guaranteed to be unique.

这道转圈加油问题不算很难，只要想通其中的原理就很简单。我们首先要知道能走完整个环的前提是gas的总量要大于cost的总量，这样才会有起点的存在。假设开始设置起点start = 0，并从这里出发，如果当前的gas值大于cost值，就可以继续前进，此时到下一个站点，剩余的gas加上当前的gas再减去cost，看是否大于0，若大于0，则继续前进。当到达某一站点时，若这个值小于0了，则说明从起点到这个点中间的任何一个点都不能作为起点，则把起点设为下一个点，继续遍历。当遍历完整个环时，当前保存的起点即为所求。代码如下：

解法1：

```

1 class Solution {
2 public:
3     int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
4         int total = 0, sum = 0, start = 0;
5         for (int i = 0; i < gas.size(); ++i) {
6             total += gas[i] - cost[i];
7             sum += gas[i] - cost[i];
8             if (sum < 0) {
9                 start = i + 1;
10                sum = 0;
11            }
12        }
13        return (total < 0) ? -1 : start;
14    }
15 };

```

我们也可以从后往前遍历，用一个变量mx来记录出现过的剩余油量的最大值，total记录当前剩余油量的值，start还是记录起点的位置。当total大于mx的时候，说明当前位置可以作为起点，更新start，并且更新mx。为啥呢？因为我们每次total加上的都是当前位置的油量减去消耗，如果这个差值大于0的话，说明当前位置可以当作起点，因为从当前位置到末尾都不会出现油量不够的情况，而一旦差值小于0的话，说明当前位置如果是起点的话，油量就不够，无法走完全程，所以我们不更新起点位置start。最后结束后我们还是看totoa是否大于等于0，如果其小于0的话，说明没有任何一个起点能走完全程，因为总油量都不够，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
4         int total = 0, mx = -1, start = 0;
5         for (int i = gas.size() - 1; i >= 0; --i) {
6             total += gas[i] - cost[i];
7             if (total > mx) {
8                 start = i;
9                 mx = total;
10            }
11        }
12        return (total < 0) ? -1 : start;
13    }
14 };

```

135. 分糖果问题

There are N children standing in a line. Each child is assigned a rating value.

You are giving candies to these children subjected to the following requirements:

Each child must have at least one candy.

Children with a higher rating get more candies than their neighbors.

What is the minimum candies you must give?

这道题看起来很难，其实解法并没有那么复杂，当然我也是看了别人的解法才做出来的，先来看看两遍遍历的解法，首先初始化每个人一个糖果，然后这个算法需要遍历两遍，第一遍从左向右遍历，如果右边的小盆友的等级高，等加一个糖果，这样保证了一个方向上高等级的糖果多。然后再从右向左遍历一遍，如果相邻两个左边的等级高，而左边的糖果又少的话，则左边糖果数为右边糖果数加一。最后再把所有小盆友的糖果数都加起来返回即可。代码如下：

解法1：

```

1 class Solution {
2 public:
3     int candy(vector<int>& ratings) {
4         int res = 0, n = ratings.size();
5         vector<int> nums(n, 1);
6         for (int i = 0; i < n - 1; ++i) {
7             if (ratings[i + 1] > ratings[i]) nums[i + 1] = nums[i] + 1;
8         }
9         for (int i = n - 1; i > 0; --i) {
10             if (ratings[i - 1] > ratings[i]) nums[i - 1] = max(nums[i - 1], nums[i] + 1);
11         }
12         for (int num : nums) res += num;
13         return res;
14     }
15 };

```

CPP

下面来看一次遍历的方法，相比于遍历两次的思路简单明了，这种只遍历一次的解法就稍有些复杂了。首先我们给第一个同学一个糖果，那么对于接下来的一个同学就有三种情况：

1. 接下来的同学的rating等于前一个同学，那么给接下来的同学一个糖果就行。
2. 接下来的同学的rating大于前一个同学，那么给接下来的同学的糖果数要比前一个同学糖果数加1。
3. 接下来的同学的rating小于前一个同学，那么我们此时不知道应该给这个同学多少个糖果，需要看后面的情况。

对于第三种情况，我们不确定要给几个，因为要是只给1个的话，那么有可能接下来还有rating更小的同学，总不能一个都不给吧。也不能直接给前一个同学的糖果数减1，有可能给多了，因为如果后面再没人了的话，其实只要给一个就行了。还有就是，如果后面好几个rating越来越小的同学，那么前一个同学的糖果数可能还得追加，以保证最后面的同学至少能有1个糖果。来一个例子吧，四个同学，他们的rating如下：

1 3 2 1

先给第一个rating为1的同学一个糖果，然后从第二个同学开始遍历，第二个同学rating为3，比1大，所以多给一个糖果，第二个同学得到两个糖果。下面第三个同学，他的rating为2，比前一个同学的rating小，如果我们此时给1个糖果的话，那么rating更小的第四个同学就得不到糖果了，所以我们要给第四个同学1个糖果，而给第三个同学2个糖果，此时要给第二个同学追加1个糖果，使其能够比第三个同学的糖果数多至少一个。那么我们就需要统计出多有个连着的同学的rating变小，用变量cnt来记录，找出了最后一个减小的同学，那么就可以往前推，每往前一个加一个糖果，这就是个等差数列，我们可以直接利用求和公式算出这些rating减小的同学的糖果之和。然后我们还要看第一个开始减小的同学的前一个同学需不需要追加糖果，只要比较cnt和pre的大小，pre是之前同学得到的最大糖果数，二者做差加1就是需要追加的糖果数，加到结果res中即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int candy(vector<int>& ratings) {
4         if (ratings.empty()) return 0;
5         int res = 1, pre = 1, cnt = 0;
6         for (int i = 1; i < ratings.size(); ++i) {
7             if (ratings[i] >= ratings[i - 1]) {
8                 if (cnt > 0) {
9                     res += cnt * (cnt + 1) / 2;
10                if (cnt >= pre) res += cnt - pre + 1;
11                cnt = 0;
12                pre = 1;
13            }
14            pre = (ratings[i] == ratings[i - 1]) ? 1 : pre + 1;
15            res += pre;
16        } else {
17            ++cnt;
18        }
19    }
20    if (cnt > 0) {
21        res += cnt * (cnt + 1) / 2;
22        if (cnt >= pre) res += cnt - pre + 1;
23    }
24    return res;
25 }
26 };

```

136. 单独的数字

Given an array of integers, every element appears twice except for one. Find that single one.

Note:

Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

本来是一道非常简单的题，但是由于加上了时间复杂度必须是O(n)，并且空间复杂度为O(1)，使得不能用排序方法，也不能使用map数据结构。那么只能另辟蹊径，需要用位操作Bit Operation来解此题，这个解法如果让我想，肯定想不出来，因为谁会想到用逻辑异或来解题呢。逻辑异或的真值表为：

异或运算A xor B的真值表如下：

A	B	\oplus
F	F	F
F	T	T
T	F	T
T	T	F

由于数字在计算机是以二进制存储的，每位上都是0或1，如果我们把两个相同的数字异或，0与0异或是0,1与1异或也是0，那么我们会得到0。根据这个特点，我们把数组中所有的数字都异或起来，则每对相同的数字都会得0，然后最后剩下的数字就是那个只有1次的数字。这个方法确实很赞，但是感觉一般人不会忘异或上想，绝对是为CS专业的同学设计的好题呀，赞一个~~

```

1 class Solution {
2 public:
3     int singleNumber(vector<int>& nums) {
4         int res = 0;
5         for (auto num : nums) res ^= num;
6         return res;
7     }
8 };

```

137. 单独的数字之二

Given an array of integers, every element appears three times except for one. Find that single one.

Note:

Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

这道题是之前那道 Single Number 单独的数字 的延伸，那道题的解法就比较独特，是利用计算机按位储存数字的特性来做的，这道题就是除了一个单独的数字之外，数组中其他的数字都出现了三次，那么还是要利用位操作 Bit Operation 来解此题。我们可以建立一个32位的数字，来统计每一位上1出现的个数，我们知道如果某一位上为1的话，那么如果该整数出现了三次，对3去余为0，我们把每个数的对应位都加起来对3取余，最终剩下来的那个数就是单独的数字。代码如下：

解法1：

```

1 class Solution {
2 public:
3     int singleNumber(vector<int>& nums) {
4         int res = 0;
5         for (int i = 0; i < 32; ++i) {
6             int sum = 0;
7             for (int j = 0; j < nums.size(); ++j) {
8                 sum += (nums[j] >> i) & 1;
9             }
10            res |= (sum % 3) << i;
11        }
12        return res;
13    }
14 };

```

还有一种解法，思路很相似，用3个整数来表示INT的各位的出现次数情况，one表示出现了1次，two表示出现了2次。当出现3次的时候该位清零。最后答案就是one的值。

ones 代表第ith位只出现一次的掩码变量
 twos 代表第ith位只出现两次的掩码变量
 threes 代表第ith位只出现三次的掩码变量

假设现在有一个数字1，那么我们更新one的方法就是‘或’这个1，则one就变成了1，而two的更新方法是用上一个状态下的one去‘与’上数字1，然后‘或’上这个结果，这样假如之前one是1，那么此时two也会变成1，这make sense，因为说明是当前位遇到两个1了；反之如果之前one是0，那么现在two也就是0。注意更新的顺序是先更新two，再更新one，不理解的话只要带个只有一个数字1的输入数组看一下就不难理解了。然后我们更新three，如果此时one和two都是1了，那么由于我们先更新的two，再更新的one，two为1，说明此时至少有两个数字1了，而此时one为1，说明了此时已经有了三个数字1，这块要仔细想清楚，因为one是要‘或’一个1的，值能为1，说明之前one为0，实际情况是，当第二个1来的时候，two先更新为1，此时one再更新为0，下面three就是0了，那么‘与’上three的相反数1不会改变one和two的值；那么当第三个1来的时候，two还是1，此时one就更新为1了，那么three就更新为1了，此时就要清空one和two了，让它们‘与’上three的相反数0即可，最终结果将会保存在one中，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int singleNumber(vector<int>& nums) {
4         int one = 0, two = 0, three = 0;
5         for (int i = 0; i < nums.size(); ++i) {
6             two |= one & nums[i];
7             one ^= nums[i];
8             three = one & two;
9             one &= ~three;
10            two &= ~three;
11        }
12        return one;
13    }
14 }
```

CPP

下面这种解法思路也十分巧妙，根据上面解法的思路，我们把数组中数字的每一位累加起来对3取余，剩下的结果就是那个单独数组该位上的数字，由于我们累加的过程都要对3取余，那么每一位上累加的过程就是 $0 \rightarrow 1 \rightarrow 2 \rightarrow 0$ ，换成二进制的表示为 $00 \rightarrow 01 \rightarrow 10 \rightarrow 00$ ，那么我们可以写出对应关系：

$00 (+) 1 = 01$

$01 (+) 1 = 10$

$10 (+) 1 = 00 (\bmod 3)$

那么我们用ab来表示开始的状态，对于加1操作后，得到的新状态的ab的算法如下：

```
b = b xor r & ~a;
a = a xor r & ~b;
```

我们这里的ab就是上面的三种状态00, 01, 10的十位和各位，刚开始的时候，a和b都是0，当此时遇到数字1的时候，b更新为1，a更新为0，就是01的状态；再次遇到1的时候，b更新为0，a更新为1，就是10的状态；再次遇到1的时候，b更新为0，a更新为0，就是00的状态，相当于重置了；最后的结果保存在b中。明白了上面的分析过程，就能写出代码如下：

解法3：

```

1 class Solution {
2 public:
3     int singleNumber(vector<int>& nums) {
4         int a = 0, b = 0;
5         for (int i = 0; i < nums.size(); ++i) {
6             b = (b ^ nums[i]) & ~a;
7             a = (a ^ nums[i]) & ~b;
8         }
9         return b;
10    }
11 };

```

138. 拷贝带有随机指针的链表

A linked list is given such that each node contains an additional random pointer which could point to any node in the list or null.

Return a deep copy of the list.

这道链表的深度拷贝题的难点就在于如何处理随机指针的问题，由于每一个节点都有一个随机指针，这个指针可以为空，也可以指向链表的任意一个节点，如果我们在每生成一个新节点给其随机指针赋值时，都要去遍历原链表的话，OJ上肯定会超时，所以我们可以考虑用Hash map来缩短查找时间，第一遍遍历生成所有新节点时同时建立一个原节点和新节点的哈希表，第二遍给随机指针赋值时，查找时间是常数级。代码如下：

解法1：

```

1 class Solution {
2 public:
3     RandomListNode *copyRandomList(RandomListNode *head) {
4         if (!head) return NULL;
5         RandomListNode *res = new RandomListNode(head->label);
6         RandomListNode *node = res;
7         RandomListNode *cur = head->next;
8         map<RandomListNode*, RandomListNode*> m;
9         m[head] = res;
10        while (cur) {
11            RandomListNode *tmp = new RandomListNode(cur->label);
12            node->next = tmp;
13            m[cur] = tmp;
14            node = node->next;
15            cur = cur->next;
16        }
17        node = res;
18        cur = head;
19        while (node) {
20            node->random = m[cur->random];
21            node = node->next;
22            cur = cur->next;
23        }
24        return res;
25    }
26 };

```

当然，如果使用哈希表占用额外的空间，如果这道题限制了空间的话，就要考虑别的方法。下面这个方法很巧妙，具体细节可参见神网友水中的鱼的博客，该方法可以分为以下三个步骤：

1. 在原链表的每个节点后面拷贝出一个新的节点
2. 依次给新的节点的随机指针赋值，而且这个赋值非常容易 $\text{cur} \rightarrow \text{next} \rightarrow \text{random} = \text{cur} \rightarrow \text{random} \rightarrow \text{next}$
3. 断开链表可得到深度拷贝后的新链表

解法2：

```

1 class Solution {
2 public:
3     RandomListNode *copyRandomList(RandomListNode *head) {
4         if (!head) return NULL;
5         RandomListNode *cur = head;
6         while (cur) {
7             RandomListNode *node = new RandomListNode(cur->label);
8             node->next = cur->next;
9             cur->next = node;
10            cur = node->next;
11        }
12        cur = head;
13        while (cur) {
14            if (cur->random) {
15                cur->next->random = cur->random->next;
16            }
17            cur = cur->next->next;
18        }
19        cur = head;
20        RandomListNode *res = head->next;
21        while (cur) {
22            RandomListNode *tmp = cur->next;
23            cur->next = tmp->next;
24            if (tmp->next) tmp->next = tmp->next->next;
25            cur = cur->next;
26        }
27        return res;
28    }
29 };

```

139. 拆分词句

Given a non-empty string s and a dictionary wordDict containing a list of non-empty words, determine if s can be segmented into a space-separated sequence of one or more dictionary words.

Note:

The same word in the dictionary may be reused multiple times in the segmentation.
You may assume the dictionary does not contain duplicate words.

Example 1:

Input: s = "leetcode", wordDict = ["leet", "code"]

Output: true

Explanation: Return true because "leetcode" can be segmented as "leet code".

Example 2:

Input: s = "applepenapple", wordDict = ["apple", "pen"]

Output: true

Explanation: Return true because "applepenapple" can be segmented as "apple pen apple".
Note that you are allowed to reuse a dictionary word.

Example 3:

Input: s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]

Output: false

这道拆分句问题是看给定的词句能分被拆分成字典里面的内容，这是一道很经典的题目，解法不止一种，考察的范围很广，属于我们必须要熟练掌握的题目。那么先来想brute force的解法，就拿例子1来分析，如果字典中只有两个单词，我们怎么去判断，是不是可以将原字符串s分成任意两段，然后再看分成的单词是否在字典中。注意这道题说是单词可以重复使用，所以可以分成任意段，而且字典中的单词可以有很多个，这就增加了题目的难度，很多童鞋就在这里迷失了，毫无头绪。那么，就由博主来给各位指点迷津吧（此处应有掌声）。

既然要分段，看子字符串是否在字典中，由于给定的字典是数组（之前还是HashSet呢），那么我们肯定不希望每次查找都需要遍历一遍数组，费劲！还是把字典中的所有单词都存入HashSet中吧，这样我们就有了常数时间级的查找速度，perfect！好，我们得开始给字符串分段了，怎么分，只能一个一个分了，先看第一个字母是否在字典中，如果不在的话，好办，说明这种分法肯定是错的。问题是在的话，后面的那部分怎么处理，难道还用for循环？你也不知道还要分多少段，怎么用for循环。对于这种不知道怎么处理的情况，一个万能的做法是丢给递归函数，让其去递归求解，这里我们suppose递归函数会返回我们一个正确的值，如果返回的是true的话，表明我们现在分成的两段都在字典中，我们直接返回true即可，因为只要找出一种情况就行了。这种调用递归函数的方法就是brute force的解法，我们遍历了所有的情况，优点是写法简洁，思路清晰，缺点是存在大量的重复计算，被OJ啪啪打脸。所以我们需要进行优化，使用记忆数组memo来保存所有已经计算过的结果，再下次遇到的时候，直接从cache中取，而不是再次计算一遍。这种使用记忆数组memo的递归写法，和使用dp数组的迭代写法，乃解题的两大神器，凡凡事能用dp解的题，一般也有用记忆数组的递归解法，好似一对形影不离的好基友~关于dp解法，博主要在下文中讲解。这里我们的记忆数组memo[i]定义为范围为[0, i)的子字符串是否可以拆分，初始化为-1，表示没有计算过，如果可以拆分，则赋值为1，反之为0。在之前讲brute force解法时，博主要提到的是讲分成两段的后半段的调用递归函数，我们也可以不取出子字符串，而是用一个start变量，来标记分段的位置，这样递归函数中只需要从start的位置往后遍历即可，在递归函数更新记忆数组memo即可，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     bool wordBreak(string s, vector<string>& wordDict) {
4         unordered_set<string> wordSet(wordDict.begin(), wordDict.end());
5         vector<int> memo(s.size(), -1);
6         return check(s, wordSet, 0, memo);
7     }
8     bool check(string s, unordered_set<string>& wordSet, int start, vector<int>& memo) {
9         if (start >= s.size()) return true;
10        if (memo[start] != -1) return memo[start];
11        for (int i = start + 1; i <= s.size(); ++i) {
12            if (wordSet.count(s.substr(start, i - start)) && check(s, wordSet, i, memo)) {
13                memo[start] = 1;
14            }
15        }
16        return memo[start] = 0;
17    }
18 };

```

这道题其实还是一道经典的DP题目，也就是动态规划Dynamic Programming。博主曾经说玩子数组或者子字符串且求极值的题，基本就是DP没差了，虽然这道题没有求极值，但是玩子字符串也符合DP的状态转移的特点。把一个人的温暖转移到另一个人的胸膛... 咳咳，跑错片场了，那是爱情转移~ 强行拉回，DP解法的两大难点，定义dp数组跟找出状态转移方程，先来看dp数组的定义，这里我们就用一个一维的dp数组，其中dp[i]表示范围[0, i)内的子串是否可以拆分，注意这里dp数组的长度比s串的长度大1，是因为我们要handle空串的情况，我们初始化dp[0]为true，然后开始遍历。注意这里我们需要两个for循环来遍历，因为此时已经没有递归函数了，所以我们必须要遍历所有的子串，我们用j把[0, i)范围内的子串分为了两部分，[0, j)和[j, i)，其中范围[0, j)就是dp[j]，范围[j, i)就是s.substr(j, i-j)，其中dp[j]是之前的状态，我们已经算出来了，可以直接取，只需要在字典中查找s.substr(j, i-j)是否存在了，如果二者均为true，将dp[i]赋为true，并且break掉，此时就不需要再用j去分[0, i)范围了，因为[0, i)范围已经可以拆分了。最终我们返回dp数组的最后一个值，就是整个数组是否可以拆分的布尔值了，代码如下：

```

1 class Solution {
2 public:
3     bool wordBreak(string s, vector<string>& wordDict) {
4         unordered_set<string> wordSet(wordDict.begin(), wordDict.end());
5         vector<bool> dp(s.size() + 1);
6         dp[0] = true;
7         for (int i = 0; i < dp.size(); ++i) {
8             for (int j = 0; j < i; ++j) {
9                 if (dp[j] && wordSet.count(s.substr(j, i - j))) {
10                     dp[i] = true;
11                     break;
12                 }
13             }
14         }
15         return dp.back();
16     }
17 };

```

下面我们从题目中给的例子来分析：

```

1
le e
lee ee e
leet
leetc eetc etc tc c
leetco eetco etco tco co o
leetcod eetcod etcod tcod cod od d
leetcode eetcode etcode tcode code
T F F F T F F F T

```

我们知道算法的核心思想是逐行扫描，每一行再逐个字符扫描，每次都在组合出一个新的字符串都要到字典里去找，如果有的话，则跳过此行，继续扫描下一行。

既然DFS都可以解题，那么BFS也就坐不住了，也要出来蹦跶一下。其实本质跟递归的解法没有太大的区别，递归解法在调用递归的时候，原先的状态被存入了栈中，这里BFS是存入了队列中，使用visited数组来标记已经算过的位置，作用跟memo数组一样，从队列中取出一个位置进行遍历，把可以拆分的新位置存入队列中，遍历完成后标记当前位置，然后再回到队列中去取即可，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     bool wordBreak(string s, vector<string>& wordDict) {
4         unordered_set<string> wordSet(wordDict.begin(), wordDict.end());
5         vector<bool> visited(s.size());
6         queue<int> q{{0}};
7         while (!q.empty()) {
8             int start = q.front(); q.pop();
9             if (!visited[start]) {
10                 for (int i = start + 1; i <= s.size(); ++i) {
11                     if (wordSet.count(s.substr(start, i - start))) {
12                         q.push(i);
13                         if (i == s.size()) return true;
14                     }
15                 }
16                 visited[start] = true;
17             }
18         }
19         return false;
20     }
21 };

```

CPP

140. 拆分词句之二

Given a non-empty string s and a dictionary wordDict containing a list of non-empty words, add spaces in s to construct a sentence where each word is a valid dictionary word. Return all such possible sentences.

Note:

The same word in the dictionary may be reused multiple times in the segmentation.
You may assume the dictionary does not contain duplicate words.

Example 1:

Input:

```
s = "catsanddog"
wordDict = ["cat", "cats", "and", "sand", "dog"]
```

Output:

```
[  
    "cats and dog",  
    "cat sand dog"  
]
```

Example 2:

Input:

```
s = "pineapplepenapple"
wordDict = ["apple", "pen", "applepen", "pine", "pineapple"]
```

Output:

```
[  
    "pine apple pen apple",  
    "pineapple pen apple",  
    "pine applepen apple"  
]
```

Explanation: Note that you are allowed to reuse a dictionary word.

Example 3:

Input:

```
s = "catsandog"
wordDict = ["cats", "dog", "sand", "and", "cat"]
```

Output:

```
[]
```

这道题是之前那道Word Break 拆分词句的拓展，那道题只让我们判断给定的字符串能否被拆分成字典中的词，而这道题加大了难度，让我们求出所有可以拆分成的情况，就像题目中给的例子所示。之前的版本中字典wordDict的数据类型是HashSet，现在的不知为何改成了数组vector，而且博主看到第二个例子就笑了，PPAP么，哈哈。

根据博主行走江湖多年的经验，像这种返回结果要列举所有情况的题，十有八九都是要用递归来的。当我们一时半会没有啥思路的时候，先不要考虑代码如何实现，如果就给你一个s和wordDict，不看Output的内容，你会怎么找出结果。比如对于例子1，博主可能会先扫一遍wordDict数组，看有没有单词可以当s的开头，那么我们可以发现cat和cats都可以，比如我们先选了cat，那么此时s就变成了"sanddog"，我们再在数组里找单词，发现了sand可以，最后剩一个dog，也在数组中，于是一个结果就出来了。然后回到开头选cats的话，那么此时s就变成了"anddog"，我们再在数组里找单词，发现了and可以，最后剩一个dog，也在数组中，于是另一个结果也就出来了。那么这个查询的方法很适合用递归来实现，因为s改变后，查询的机制并不变，很适合调用递归函数。再者，我们要明确的是，如果不使用记忆数组做减少重复计算的优化，那么递归方法跟brute force没什么区别，大概率无法通过OJ。所以我们要避免重复计算，如何避免呢，还是看上面的分析，如果当s变成"sanddog"的时候，那么此时我们知道其可以拆分成sand和dog，当某个时候如果我们又遇到了这个"sanddog"的时候，我们难道还需要再调用递归算一遍吗，当然不希望啦，所以我们要将这个中间结果保存起来，由于我们必须同时保存s和其所有的拆分的字符串，那么可以使用一个HashMap，来建立二者之间的映射，那么在递归函数中，我们首先检测当前s是否已经有映射，有的话直接返回即可，如果s为空了，我们如何处理呢，题目中说了给定的s不会为空，但是我们递归函数处理时s是会变空的，这时候我们是直接返回空集吗，这里有个小trick，我们其实放一个空字符串返回，为啥要这么做呢？我们观察题目中的Output，发现单词之间是有空格，而最后一个单词后面没有空格，所以这个空字符串就起到了标记当前单词是最后一个，那么我们就不要再加空格了。接着往下

看，我们遍历wordDict数组，如果某个单词是s字符串中的开头单词的话，我们对后面部分调用递归函数，将结果保存到rem中，然后遍历里面的所有字符串，和当前的单词拼接起来，这里就用到了我们前面说的trick。for循环结束后，记得返回结果res之前建立其和s之间的映射，方便下次使用，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     vector<string> wordBreak(string s, vector<string>& wordDict) {
4         unordered_map<string, vector<string>> m;
5         return helper(s, wordDict, m);
6     }
7     vector<string> helper(string s, vector<string>& wordDict, unordered_map<string,
8     vector<string>>& m) {
9         if (m.count(s)) return m[s];
10        if (s.empty()) return {""};
11        vector<string> res;
12        for (string word : wordDict) {
13            if (s.substr(0, word.size()) != word) continue;
14            vector<string> rem = helper(s.substr(word.size()), wordDict, m);
15            for (string str : rem) {
16                res.push_back(word + (str.empty() ? "" : " ") + str);
17            }
18        }
19        m[s] = res;
20    }
};
```

我们也可以将将主函数本身当作递归函数，这样就不用单独的使用一个递归函数了，不过我们的HashMap必须是全局了，写在外部就好了，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     unordered_map<string, vector<string>> m;
4     vector<string> wordBreak(string s, vector<string>& wordDict) {
5         if (m.count(s)) return m[s];
6         if (s.empty()) return {""};
7         vector<string> res;
8         for (string word : wordDict) {
9             if (s.substr(0, word.size()) != word) continue;
10            vector<string> rem = wordBreak(s.substr(word.size()), wordDict);
11            for (string str : rem) {
12                res.push_back(word + (str.empty() ? "" : " ") + str);
13            }
14        }
15        m[s] = res;
16    }
};
```

141. 单链表中的环

Given a linked list, determine if it has a cycle in it.

Follow up:

Can you solve it without using extra space?

这道题是快慢指针的经典应用。只需要设两个指针，一个每次走一步的慢指针和一个每次走两步的快指针，如果链表里有环的话，两个指针最终肯定会相遇。实在是太巧妙了，要是我肯定想不出来。代码如下：

```
1 class Solution {
2 public:
3     bool hasCycle(ListNode *head) {
4         ListNode *slow = head, *fast = head;
5         while (fast && fast->next) {
6             slow = slow->next;
7             fast = fast->next->next;
8             if (slow == fast) return true;
9         }
10        return false;
11    }
12};
```

CPP

142. 单链表中的环之二

Given a linked list, return the node where the cycle begins. If there is no cycle, return null.

Follow up:

Can you solve it without using extra space?

这个求单链表中的环的起始点是之前那个判断单链表中是否有环的延伸，可参见我之前的一篇文章

(<http://www.cnblogs.com/grandyang/p/4137187.html>). 还是要设快慢指针，不过这次要记录两个指针相遇的位置，当两个指针相遇后，让其一指针从链表头开始，一步两步，一步一步似爪牙，似魔鬼的步伐。。。哈哈，打住打住。。。此时再相遇的位置就是链表中环的起始位置。代码如下：

```
1 class Solution {
2 public:
3     ListNode *detectCycle(ListNode *head) {
4         ListNode *slow = head, *fast = head;
5         while (fast && fast->next) {
6             slow = slow->next;
7             fast = fast->next->next;
8             if (slow == fast) break;
9         }
10        if (!fast || !fast->next) return NULL;
11        slow = head;
12        while (slow != fast) {
13            slow = slow->next;
14            fast = fast->next;
15        }
16        return fast;
17    }
18};
```

CPP

143. 链表重排序

Given a singly linked list L: L₀→L₁→...→L_{n-1}→L_n,
reorder it to: L₀→L_n→L₁→L_{n-1}→L₂→L_{n-2}→...

You must do this in-place without altering the nodes' values.

For example,
Given {1,2,3,4}, reorder it to {1,4,2,3}.

这道链表重排序问题可以拆分为以下三个小问题:

1. 使用快慢指针来找到链表的中点，并将链表从中点处断开，形成两个独立的链表。
2. 将第二个链翻转。
3. 将第二个链表的元素间隔地插入第一个链表中。

```

1 class Solution {
2 public:
3     void reorderList(ListNode *head) {
4         if (!head || !head->next || !head->next->next) return;
5         ListNode *fast = head;
6         ListNode *slow = head;
7         while (fast->next && fast->next->next) {
8             slow = slow->next;
9             fast = fast->next->next;
10        }
11        ListNode *mid = slow->next;
12        slow->next = NULL;
13        ListNode *last = mid;
14        ListNode *pre = NULL;
15        while (last) {
16            ListNode *next = last->next;
17            last->next = pre;
18            pre = last;
19            last = next;
20        }
21        while (head && pre) {
22            ListNode *next = head->next;
23            head->next = pre;
24            pre = pre->next;
25            head->next->next = next;
26            head = next;
27        }
28    }
29 };

```

CPP

144. 二叉树的先序遍历

Given a binary tree, return the preorder traversal of its nodes' values.

For example:

Given binary tree {1,#,2,3},

```
1
 \
 2
 /
3
```

return [1,2,3].

Note: Recursive solution is trivial, could you do it iteratively?

一般我们提到树的遍历，最常见的有先序遍历，中序遍历，后序遍历和层序遍历，它们用递归实现起来都非常的简单。而题目的要求是不能使用递归求解，于是只能考虑到用非递归的方法，这就要用到stack来辅助运算。由于先序遍历的顺序是“根-左-右”，算法为：

1. 把根节点push到栈中
2. 循环检测栈是否为空，若不空，则取出栈顶元素，保存其值，然后看其右子节点是否存在，若存在则push到栈中。再看其左子节点，若存在，则push到栈中。

代码如下：

解法1：

```
1 class Solution {
2 public:
3     vector<int> preorderTraversal(TreeNode* root) {
4         if (!root) return {};
5         vector<int> res;
6         stack<TreeNode*> s{{root}};
7         while (!s.empty()) {
8             TreeNode *t = s.top(); s.pop();
9             res.push_back(t->val);
10            if (t->right) s.push(t->right);
11            if (t->left) s.push(t->left);
12        }
13        return res;
14    }
15};
```

CPP

下面这种写法使用了一个辅助结点p，这种写法其实可以看作是一个模版，对应的还有中序和后序的模版写法，形式很统一，方便于记忆。辅助结点p初始化为根结点，while循环的条件是栈不为空或者辅助结点p不为空，在循环中首先判断如果辅助结点p存在，那么先将p加入栈中，然后将p的结点值加入结果res中，此时p指向其左子结点。否则如果p不存在的话，表明没有左子结点，我们取出栈顶结点，将p指向栈顶结点的右子结点，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<int> preorderTraversal(TreeNode* root) {
4         vector<int> res;
5         stack<TreeNode*> s;
6         TreeNode *p = root;
7         while (!s.empty() || p) {
8             if (p) {
9                 s.push(p);
10                res.push_back(p->val);
11                p = p->left;
12            } else {
13                TreeNode *t = s.top(); s.pop();
14                p = t->right;
15            }
16        }
17        return res;
18    }
19 };

```

145. 二叉树的后序遍历

Given a binary tree, return the postorder traversal of its nodes' values.

For example:

Given binary tree {1,#,2,3},

```

1
 \
 2
 /
3
return [3,2,1].

```

Note: Recursive solution is trivial, could you do it iteratively?

经典题目，求二叉树的后序遍历的非递归方法，跟前序，中序，层序一样都需要用到栈，后续的顺序是左-右-根，所以当一个节点值被取出来时，它的左右子节点要么不存在，要么已经被访问过了。我们先将根结点压入栈，然后定义一个辅助结点head，while循环的条件是栈不为空，在循环中，首先将栈顶结点t拿出来，如果栈顶结点没有左右子结点，或者其左子结点是head，或者其右子结点是head的情况下。我们将栈顶结点值加入结果res中，并将栈顶元素移出栈，然后将head指向栈顶元素；否则的话就看如果右子结点不为空，将其加入栈，再看左子结点不为空的话，就加入栈，注意这里先右后左的顺序是因为栈的后入先出的特点，可以使得左子结点先被处理。下面来看为什么是这三个条件呢，首先如果栈顶元素如果没有左右子结点的话，说明其是叶结点，而且我们的入栈顺序保证了左子结点先被处理，所以此时的结点值就可以直接加入结果res了，然后移出栈，将head指向这个叶结点，这样的话head每次就是指向前一个处理过并且加入结果res的结点，那么如果栈顶结点的左子结点或者右子结点是head的话，说明其子结点已经加入结果res了，那么就可以处理当前结点了，代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<int> postorderTraversal(TreeNode* root) {
4         if (!root) return {};
5         vector<int> res;
6         stack<TreeNode*> s{{root}};
7         TreeNode *head = root;
8         while (!s.empty()) {
9             TreeNode *t = s.top();
10            if ((!t->left && !t->right) || t->left == head || t->right == head) {
11                res.push_back(t->val);
12                s.pop();
13                head = t;
14            } else {
15                if (t->right) s.push(t->right);
16                if (t->left) s.push(t->left);
17            }
18        }
19        return res;
20    }
21 };

```

由于后序遍历的顺序是左-右-根，而先序遍历的顺序是根-左-右，二者其实还是很相近的，我们可以先在先序遍历的方法上做些小改动，使其遍历顺序变为根-右-左，然后翻转一下，就是左-右-根啦，翻转的方法我们使用反向Q，哦不，是反向加入结果res，每次都在结果res的开头加入结点值，而改变先序遍历的顺序就只要该遍历一下入栈顺序，先左后右，这样出栈处理的时候就是先右后左啦，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<int> postorderTraversal(TreeNode* root) {
4         if (!root) return {};
5         vector<int> res;
6         stack<TreeNode*> s{{root}};
7         while (!s.empty()) {
8             TreeNode *t = s.top(); s.pop();
9             res.insert(res.begin(), t->val);
10            if (t->left) s.push(t->left);
11            if (t->right) s.push(t->right);
12        }
13        return res;
14    }
15 };

```

那么在Binary Tree Preorder Traversal中的解法二也可以改动一下变成后序遍历，改动的思路跟上面的解法一样，都是先将先序遍历的根-左-右顺序变为根-右-左，再翻转变为后序遍历的左-右-根，翻转还是改变结果res的加入顺序，然后把更新辅助结点p的左右顺序换一下即可，代码如下：

解法3：

```

1 class Solution {
2 public:
3     vector<int> postorderTraversal(TreeNode* root) {
4         vector<int> res;
5         stack<TreeNode*> s;
6         TreeNode *p = root;
7         while (!s.empty() || p) {
8             if (p) {
9                 s.push(p);
10                res.insert(res.begin(), p->val);
11                p = p->right;
12            } else {
13                TreeNode *t = s.top(); s.pop();
14                p = t->left;
15            }
16        }
17        return res;
18    }
19 };

```

论坛上还有一种双栈的解法，其实本质上跟解法二没什么区别，都是利用了改变先序遍历的顺序来实现后序遍历的，参见代码如下：

解法4：

```

1 class Solution {
2 public:
3     vector<int> postorderTraversal(TreeNode* root) {
4         if (!root) return {};
5         vector<int> res;
6         stack<TreeNode*> s1, s2;
7         s1.push(root);
8         while (!s1.empty()) {
9             TreeNode *t = s1.top(); s1.pop();
10            s2.push(t);
11            if (t->left) s1.push(t->left);
12            if (t->right) s1.push(t->right);
13        }
14        while (!s2.empty()) {
15            res.push_back(s2.top()->val); s2.pop();
16        }
17        return res;
18    }
19 };

```

146. 最近最少使用页面置换缓存器

Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: get and set.

get(key) - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.

set(key, value) - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item.

这道题让我们实现一个LRU缓存器，LRU是Least Recently Used的简写，就是最近最少使用的意思。那么这个缓存器主要有两个成员函数，get和put，其中get函数是通过输入key来获得value，如果成功获得后，这对(key, value)升至缓存器中最常用的位置（顶部），如果key不存在，则返回-1。而put函数是插入一对新的(key, value)，如果原缓存器中有该key，则需要先删除掉原有的，将新的插入到缓存器的顶部。如果不存在，则直接插入到顶部。若加入新的值后缓存器超过了容量，则需要删掉一个最不常用的值，也就是底部的值。具体实现时我们需要三个私有变量，cap, l和m，其中cap是缓存器的容量大小，l是保存缓存器内容的列表，m是哈希表，保存关键值key和缓存器各项的迭代器之间映射，方便我们以O(1)的时间内找到目标项。

然后我们再来看get和put如何实现，get相对简单些，我们在m中查找给定的key，如果存在则将此项移到顶部，并返回value，若不存在返回-1。对于put，我们也是现在m中查找给定的key，如果存在就删掉原有项，并在顶部插入新来项，然后判断是否溢出，若溢出则删掉底部项(最不常用项)。代码如下：

```

1 class LRUCache{
2 public:
3     LRUCache(int capacity) {
4         cap = capacity;
5     }
6
7     int get(int key) {
8         auto it = m.find(key);
9         if (it == m.end()) return -1;
10        l.splice(l.begin(), l, it->second);
11        return it->second->second;
12    }
13
14    void put(int key, int value) {
15        auto it = m.find(key);
16        if (it != m.end()) l.erase(it->second);
17        l.push_front(make_pair(key, value));
18        m[key] = l.begin();
19        if (m.size() > cap) {
20            int k = l.rbegin()->first;
21            l.pop_back();
22            m.erase(k);
23        }
24    }
25
26 private:
27     int cap;
28     list<pair<int, int>> l;
29     unordered_map<int, list<pair<int, int>>::iterator> m;
30 };

```

147. 链表插入排序

Sort a linked list using insertion sort.

链表的插入排序实现原理很简单，就是一个元素一个元素的从原链表中取出来，然后按顺序插入到新链表中，时间复杂度为O(n²)，是一种效率并不是很高的算法，但是空间复杂度为O(1)，以高时间复杂度换取了低空间复杂度。代码如下：

```
1 class Solution {
2 public:
3     ListNode* insertionSortList(ListNode* head) {
4         ListNode *dummy = new ListNode(-1), *cur = dummy;
5         while (head) {
6             ListNode *t = head->next;
7             cur = dummy;
8             while (cur->next && cur->next->val <= head->val) {
9                 cur = cur->next;
10            }
11            head->next = cur->next;
12            cur->next = head;
13            head = t;
14        }
15        return dummy->next;
16    }
17};
```

148. 链表排序

Sort a linked list in $O(n \log n)$ time using constant space complexity.

常见排序方法有很多，插入排序，选择排序，堆排序，快速排序，冒泡排序，归并排序，桶排序等等。。它们的时间复杂度不尽相同，而这里题目限定了时间必须为 $O(nlgn)$ ，符合要求只有快速排序，归并排序，堆排序，而根据单链表的特点，最适于用归并排序。代码如下：

解法1：

```
1 class Solution {
2 public:
3     ListNode* sortList(ListNode* head) {
4         if (!head || !head->next) return head;
5         ListNode *slow = head, *fast = head, *pre = head;
6         while (fast && fast->next) {
7             pre = slow;
8             slow = slow->next;
9             fast = fast->next->next;
10        }
11        pre->next = NULL;
12        return merge(sortList(head), sortList(slow));
13    }
14    ListNode* merge(ListNode* l1, ListNode* l2) {
15        ListNode *dummy = new ListNode(-1);
16        ListNode *cur = dummy;
17        while (l1 && l2) {
18            if (l1->val < l2->val) {
19                cur->next = l1;
20                l1 = l1->next;
21            } else {
22                cur->next = l2;
23                l2 = l2->next;
24            }
25            cur = cur->next;
26        }
27        if (l1) cur->next = l1;
28        if (l2) cur->next = l2;
29        return dummy->next;
30    }
31};
```

下面这种方法也是归并排序，而且在merge函数中也使用了递归，这样使代码更加简洁啦~

解法2：

```

1 class Solution {
2 public:
3     ListNode* sortList(ListNode* head) {
4         if (!head || !head->next) return head;
5         ListNode *slow = head, *fast = head, *pre = head;
6         while (fast && fast->next) {
7             pre = slow;
8             slow = slow->next;
9             fast = fast->next->next;
10        }
11        pre->next = NULL;
12        return merge(sortList(head), sortList(slow));
13    }
14    ListNode* merge(ListNode* l1, ListNode* l2) {
15        if (!l1) return l2;
16        if (!l2) return l1;
17        if (l1->val < l2->val) {
18            l1->next = merge(l1->next, l2);
19            return l1;
20        } else {
21            l2->next = merge(l1, l2->next);
22            return l2;
23        }
24    }
25};

```

149. 共线点个数

Given n points on a 2D plane, find the maximum number of points that lie on the same straight line.

这道题给了我们一堆二维点，然后让我们求最大的共线点的个数，根据初中数学我们知道，两点确定一条直线，而且可以写成 $y = ax + b$ 的形式，所有共线的点都满足这个公式。所以这些给定点两两之间都可以算一个斜率，每个斜率代表一条直线，对每一条直线，带入所有的点看是否共线并计算个数，这是整体的思路。但是还有两点特殊情况需要考虑，一是当两个点重合时，无法确定一条直线，但这也是共线的情况，需要特殊处理。二是斜率不存在的情况，由于两个点 (x_1, y_1) 和 (x_2, y_2) 的斜率 k 表示为 $(y_2 - y_1) / (x_2 - x_1)$ ，那么当 $x_1 = x_2$ 时斜率不存在，这种共线情况需要特殊处理。我们需要用到哈希表来记录斜率和共线点个数之间的映射，其中第一种重合点的情况我们假定其斜率为`INT_MIN`，第二种情况我们假定其斜率为`INT_MAX`，这样都可以用map映射了。我们还需要顶一个变量`duplicate`来记录重合点的个数，最后只需和哈希表中的数字相加即为共线点的总数，这种方法现在已经无法通过OJ了，贴出来权当纪念。代码如下：

解法1：

```

1 class Solution {
2 public:
3     int maxPoints(vector<Point>& points) {
4         int res = 0;
5         for (int i = 0; i < points.size(); ++i) {
6             unordered_map<float, int> m;
7             int duplicate = 1;
8             for (int j = i + 1; j < points.size(); ++j) {
9                 if (points[i].x == points[j].x && points[i].y == points[j].y) {
10                     ++duplicate;
11                 } else if (points[i].x == points[j].x) {
12                     ++m[INT_MAX];
13                 } else {
14                     float slope = (float)(points[j].y - points[i].y) / (points[j].x -
15                     points[i].x);
16                     ++m[slope];
17                 }
18             }
19             res = max(res, duplicate);
20             for (auto it = m.begin(); it != m.end(); ++it) {
21                 res = max(res, it->second + duplicate);
22             }
23         }
24         return res;
25     }
};

```

由于通过斜率来判断共线需要用到除法，而用double表示的双精度小数在有的系统里不一定准确，为了更加精确无误的计算共线，我们应当避免除法，从而避免无线不循环小数的出现，那么怎么办呢，我们把除数和被除数都保存下来，不做除法，但是我们要让这两数分别除以它们的最大公约数，这样例如8和4，4和2，2和1，这三组商相同的数就都会存到一个映射里面，同样也能实现我们的目标，而求GCD的函数如果用递归来写那么一行就搞定了，呵不呵，这个方法能很好的避免除法的出现，算是牺牲了空间来保证精度吧，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int maxPoints(vector<Point>& points) {
4         int res = 0;
5         for (int i = 0; i < points.size(); ++i) {
6             map<pair<int, int>, int> m;
7             int duplicate = 1;
8             for (int j = i + 1; j < points.size(); ++j) {
9                 if (points[i].x == points[j].x && points[i].y == points[j].y) {
10                     ++duplicate; continue;
11                 }
12                 int dx = points[j].x - points[i].x;
13                 int dy = points[j].y - points[i].y;
14                 int d = gcd(dx, dy);
15                 ++m[{dx / d, dy / d}];
16             }
17             res = max(res, duplicate);
18             for (auto it = m.begin(); it != m.end(); ++it) {
19                 res = max(res, it->second + duplicate);
20             }
21         }
22         return res;
23     }
24     int gcd(int a, int b) {
25         return (b == 0) ? a : gcd(b, a % b);
26     }
27 };

```

令我惊奇的是，这道题的OJ居然容忍brute force的方法通过，那么我感觉下面这种O(n³)的解法之所以能通过OJ，可能还有一个原因就是用了比较高效的判断三点共线的方法。一般来说判断三点共线有三种方法，斜率法，周长法，面积法(请参见这个帖子)。而其中通过判断叉积为零的面积法是挺好的。比如说有三个点A(x₁, y₁)、B(x₂, y₂)、C(x₃, y₃)，那么判断三点共线就是判断下面这个等式是否成立：

question 149

行列式的求法不用多说吧，不会的话回去翻线性代数，当初少打点刀塔不就好啦~

解法3：

```

1 class Solution {
2 public:
3     int maxPoints(vector<Point>& points) {
4         int res = 0;
5         for (int i = 0; i < points.size(); ++i) {
6             int duplicate = 1;
7             for (int j = i + 1; j < points.size(); ++j) {
8                 int cnt = 0;
9                 long long x1 = points[i].x, y1 = points[i].y;
10                long long x2 = points[j].x, y2 = points[j].y;
11                if (x1 == x2 && y1 == y2) {++duplicate; continue;}
12                for (int k = 0; k < points.size(); ++k) {
13                    int x3 = points[k].x, y3 = points[k].y;
14                    if (x1 * y2 + x2 * y3 + x3 * y1 - x3 * y2 - x2 * y1 - x1 * y3 == 0) {
15                        ++cnt;
16                    }
17                }
18                res = max(res, cnt);
19            }
20            res = max(res, duplicate);
21        }
22        return res;
23    }
24};

```

150. 计算逆波兰表达式

Evaluate the value of an arithmetic expression in Reverse Polish Notation.

Valid operators are +, -, *, /. Each operand may be an integer or another expression.

Some examples:

```

["2", "1", "+", "3", "*"] -> ((2 + 1) * 3) -> 9
["4", "13", "5", "/", "+"] -> (4 + (13 / 5)) -> 6

```

逆波兰表达式就是把操作数放前面，把操作符后置的一种写法，我们通过观察可以发现，第一个出现的运算符，其前面必有两个数字，当这个运算符和之前两个数字完成运算后从原数组中删去，把得到一个新的数字插入到原来的位置，继续做相同运算，直至整个数组变为一个数字。于是按这种思路写了代码如下，但是拿到OJ上测试，发现会有Time Limit Exceeded的错误，无奈只好上网搜答案，发现大家都是用栈做的。仔细想想，这道题果然应该是栈的完美应用啊，从前往后遍历数组，遇到数字则压入栈中，遇到符号，则把栈顶的两个数字拿出来运算，把结果再压入栈中，直到遍历完整个数组，栈顶数字即为最终答案。代码如下：

解法1：

```

1 class Solution {
2 public:
3     int evalRPN(vector<string> &tokens) {
4         if (tokens.size() == 1) return atoi(tokens[0].c_str());
5         stack<int> s;
6         for (int i = 0; i < tokens.size(); ++i) {
7             if (tokens[i] != "+" && tokens[i] != "-" && tokens[i] != "*" && tokens[i] != "/")
8                 {
9                     s.push(atoi(tokens[i].c_str()));
10                } else {
11                    int m = s.top();
12                    s.pop();
13                    int n = s.top();
14                    s.pop();
15                    if (tokens[i] == "+") s.push(n + m);
16                    if (tokens[i] == "-") s.push(n - m);
17                    if (tokens[i] == "*") s.push(n * m);
18                    if (tokens[i] == "/") s.push(n / m);
19                }
20            }
21        }
22        return s.top();
23    }
24};

```

我们也可以用递归来做，由于一个有效的逆波兰表达式的末尾必定是操作符，所以我们可以从末尾开始处理，如果遇到操作符，向前两个位置调用递归函数，找出前面两个数字，然后进行操作将结果返回，如果遇到的是数字直接返回即可，参见代码如下

解法2：

```

1 class Solution {
2 public:
3     int evalRPN(vector<string>& tokens) {
4         int op = tokens.size() - 1;
5         return helper(tokens, op);
6     }
7     int helper(vector<string>& tokens, int& op) {
8         string s = tokens[op];
9         if (s == "+" || s == "-" || s == "*" || s == "/") {
10             int v2 = helper(tokens, --op);
11             int v1 = helper(tokens, --op);
12             if (s == "+") return v1 + v2;
13             else if (s == "-") return v1 - v2;
14             else if (s == "*") return v1 * v2;
15             else return v1 / v2;
16         } else {
17             return stoi(s);
18         }
19     }
20 };

```

151. 翻转字符串中的单词

Given an input string, reverse the string word by word.

For example,

Given s = "the sky is blue",
return "blue is sky the".

这道题让我们翻转字符串中的单词，题目中给了我们写特别说明，如果单词之间遇到多个空格，只能返回一个，而且首尾不能有单词，并且对C语言程序员要求空间复杂度为O(1)，所以我们只能对原字符串s之间做修改，而不能声明新的字符串。那么我们如何翻转字符串中的单词呢，我们的做法是，先整个字符串整体翻转一次，然后再分别翻转每一个单词（或者先分别翻转每一个单词，然后再整个字符串整体翻转一次），此时就能得到我们需要的结果了。那么这里我们需要定义一些变量来辅助我们解题，storeIndex表示当前存储到的位置，n为字符串的长度。我们先给整个字符串反转一下，然后我们开始循环，遇到空格直接跳过，如果是非空格字符，我们此时看storeIndex是否为0，为0的话表示第一个单词，不用增加空格；如果不为0，说明不是第一个单词，需要在单词中间加一个空格，然后我们要找到下一个单词的结束位置我们用一个while循环来找下一个为空格的位置，在此过程中继续覆盖原字符串，找到结束位置了，下面就来翻转这个单词，然后更新i为结尾位置，最后遍历结束，我们剪裁原字符串到storeIndex位置，就可以得到我们需要的结果，代码如下：

解法1：

```
1 class Solution {
2 public:
3     void reverseWords(string &s) {
4         int storeIndex = 0, n = s.size();
5         reverse(s.begin(), s.end());
6         for (int i = 0; i < n; ++i) {
7             if (s[i] != ' ') {
8                 if (storeIndex != 0) s[storeIndex++] = ' ';
9                 int j = i;
10                while (j < n && s[j] != ' ') s[storeIndex++] = s[j++];
11                reverse(s.begin() + storeIndex - (j - i), s.begin() + storeIndex);
12                i = j;
13            }
14        }
15        s.resize(storeIndex);
16    }
17};
```

CPP

下面我们来看使用字符串流类stringstream的解法，我们先把字符串装载入字符串流中，然后定义一个临时变量tmp，然后把第一个单词赋给s，这里需要注意的是，如果含有非空格字符，那么每次>>操作就会提取连在一起的非空格字符，那么我们每次将其加在s前面即可；如果原字符串为空，那么就不会进入while循环；如果原字符串为许多空格字符连在一起，那么第一个>>操作就会提取出这些空格字符放入s中，然后不进入while循环，这时候我们只要判断一下s的首字符是否为空格字符，是的话就将s清空即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     void reverseWords(string &s) {
4         istringstream is(s);
5         string tmp;
6         is >> s;
7         while(is >> tmp) s = tmp + " " + s;
8         if(!s.empty() && s[0] == ' ') s = "";
9     }
10 };

```

下面这种方法也是使用stringstream来做，但是我们使用了getline来做，第三个参数是设定分隔字符，我们用空格字符来分隔，这个跟上面的>>操作是有不同的，每次只能过一个空格字符，如果有多个空格字符连在一起，那么t会赋值为空字符串，所以我们在处理t的时候首先要判断其是否为空，是的话直接跳过，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     void reverseWords(string &s) {
4         istringstream is(s);
5         s = "";
6         string t = "";
7         while (getline(is, t, ' ')) {
8             if (t.empty()) continue;
9             s = (s.empty() ? t : (t + " " + s));
10        }
11    }
12 };
13

```

而如果我们使用Java的String的split函数来做的话就非常简单了，没有那么多的幺蛾子，简单明了，我们首先将原字符串调用trim()来去除冗余空格，然后调用split()来分隔，分隔符设为"\s+"，这其实是一个正则表达式，\s表示空格字符，+表示可以有一个或多个空格字符，那么我们就可以把单词分隔开装入一个字符串数组中，然后我们从末尾开始，一个个把单词取出来加入结果res中，并且单词之间加上空格字符，注意我们把第一个单词留着不取，然后返回的时候再加上即可，参见代码如下：

解法2：

```

1 public class Solution {
2     public String reverseWords(String s) {
3         String res = "";
4         String[] words = s.trim().split("\s+");
5         for (int i = words.length - 1; i > 0; --i) {
6             res += words[i] + " ";
7         }
8         return res + words[0];
9     }
10 }

```

下面这种方法就更加的简单了，疯狂的利用到了Java的内置函数，这也是Java的强大之处，注意这里的分隔符没有用正则表达式，而是直接放了个空格符进去，后面还是有+号，跟上面的写法得到的效果是一样的，然后我们对字符串数组进行翻转，然后调用join()函数来把字符串数组拼接成一个字符串，中间夹上空格符即可，参见代码如下：

解法3：

```

1 public class Solution {
2     public String reverseWords(String s) {
3         String[] words = s.trim().split(" +");
4         Collections.reverse(Arrays.asList(words));
5         return String.join(" ", words);
6     }
7 }
```

152. 求最大子数组乘积

Find the contiguous subarray within an array (containing at least one number) which has the largest product.

For example, given the array [2,3,-2,4],
the contiguous subarray [2,3] has the largest product = 6.

这个求最大子数组乘积问题是由最大子数组之和问题演变而来，但是却比求最大子数组之和要复杂，因为在求和的时候，遇到0，不会改变最大值，遇到负数，也只是会减小最大值而已。而在求最大子数组乘积的问题中，遇到0会使整个乘积为0，而遇到负数，则会使最大乘积变成最小乘积，正因为有负数和0的存在，使问题变得复杂了不少。

比如，我们现在有一个数组[2, 3, -2, 4]，我们可以很容易的找出所有的连续子数组，[2], [3], [-2], [4], [2, 3], [3, -2], [-2, 4], [2, 3, -2], [3, -2, 4], [2, 3, -2, 4]，然后可以很轻松的算出最大的子数组乘积为6，来自子数组[2, 3]。

那么我们如何写代码来实现自动找出最大子数组乘积呢，我最先想到的方法比较简单粗暴，就是找出所有的子数组，然后算出每一个子数组的乘积，然后比较找出最大的一个，需要两个for循环，第一个for遍历整个数组，第二个for遍历含有当前数字的子数组，就是按以下顺序找出子数组：[2], [2, 3], [2, 3, -2], [2, 3, -2, 4], [3], [3, -2], [3, -2, 4], [-2], [-2, 4], [4]，我在本地测试的一些数组全部通过，于是兴高采烈的拿到OJ上测试，结果丧心病狂的OJ用一个有15000个数字的数组来测试，然后说我程序的运行时间超过了要求值，我看我的代码，果然如此，时间复杂度O(n^2)，得想办法只用一次循环搞定。我想来想去想不出好方法，于是到网上搜各位大神的解决方法。其实这道题最直接的方法就是用DP来做，而且要用两个dp数组，其中f[i]表示子数组[0, i]范围内的最大子数组乘积，g[i]表示子数组[0, i]范围内的最小子数组乘积，初始化时f[0]和g[0]都初始化为nums[0]，其余都初始化为0。那么从数组的第二个数字开始遍历，那么此时的最大值和最小值只会在这三个数字之间产生，即f[i-1]*nums[i], g[i-1]*nums[i]，和nums[i]。所以我们用三者中的最大值来更新f[i]，用最小值来更新g[i]，然后用f[i]来更新结果res即可，参见代码如下：

解法1：

```

1 class Solution {
2     public:
3         int maxProduct(vector<int>& nums) {
4             int res = nums[0], n = nums.size();
5             vector<int> f(n, 0), g(n, 0);
6             f[0] = nums[0];
7             g[0] = nums[0];
8             for (int i = 1; i < n; ++i) {
9                 f[i] = max(max(f[i - 1] * nums[i], g[i - 1] * nums[i]), nums[i]);
10                g[i] = min(min(f[i - 1] * nums[i], g[i - 1] * nums[i]), nums[i]);
11                res = max(res, f[i]);
12            }
13            return res;
14        }
15    };

```

我们可以对上面的解法进行空间上的优化，以下摘自OJ官方解答，大体思路相同，写法更加简洁：

Besides keeping track of the largest product, we also need to keep track of the smallest product. Why? The smallest product, which is the largest in the negative sense could become the maximum when being multiplied by a negative number.

解法2：

```
1 class Solution {
2 public:
3     int maxProduct(vector<int>& nums) {
4         if (nums.empty()) return 0;
5         int res = nums[0], mn = nums[0], mx = nums[0];
6         for (int i = 1; i < nums.size(); ++i) {
7             int tmax = mx, tmin = mn;
8             mx = max(max(nums[i], tmax * nums[i]), tmin * nums[i]);
9             mn = min(min(nums[i], tmax * nums[i]), tmin * nums[i]);
10            res = max(res, mx);
11        }
12        return res;
13    }
14};
```

CPP

下面这种方法也是用两个变量来表示当前最大值和最小值的，但是没有无脑比较三个数，而是对于当前的`nums[i]`值进行了正负情况的讨论：

1. 当遍历到一个正数时，此时的最大值等于之前的最大值乘以这个正数和当前正数中的较大值，此时的最小值等于之前的最小值乘以这个正数和当前正数中的较小值。
2. 当遍历到一个负数时，我们先用一个变量`t`保存之前的最大值`mx`，然后此时的最大值等于之前最小值乘以这个负数和当前负数中的较大值，此时的最小值等于之前保存的最大值`t`乘以这个负数和当前负数中的较小值。
3. 在每遍历完一个数时，都要更新最终的最大值。

P.S. 如果这里改成求最小值的话，就是求最小子数组乘积，并且时间复杂度是醉人的 $O(n)$ ，是不是很强大呢，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int maxProduct(vector<int>& nums) {
4         int res = nums[0], mx = res, mn = res;
5         for (int i = 1; i < nums.size(); ++i) {
6             if (nums[i] > 0) {
7                 mx = max(mx * nums[i], nums[i]);
8                 mn = min(mn * nums[i], nums[i]);
9             } else {
10                 int t = mx;
11                 mx = max(mn * nums[i], nums[i]);
12                 mn = min(t * nums[i], nums[i]);
13             }
14             res = max(res, mx);
15         }
16         return res;
17     }
18 };

```

下面这道题使用了一个trick来将上面解法的分情况讨论合成了一种，在上面的解法中我们分析了当`nums[i]`为正数时，最大值和最小值的更新情况，为负数时，稍有不同的就是最小值更新时要用到之前的最大值，而不是更新后的最大值，所以我们才要用变量`t`来保存之前的结果。而下面这种方法的巧妙处在于先判断一个当前数字是否是负数，是的话就交换最大值和最小值。那么此时的`mx`就是之前的`mn`，所以`mx`的更新还是跟上面的方法是统一的，而在在更新`mn`的时候，之前的`mx`已经保存到`mn`中了，而且并没有改变，所以可以直接拿来用，不得不说，确实叼啊，参见代码如下：

解法4：

```

1 class Solution {
2 public:
3     int maxProduct(vector<int>& nums) {
4         int res = nums[0], mx = res, mn = res;
5         for (int i = 1; i < nums.size(); ++i) {
6             if (nums[i] < 0) swap(mx, mn);
7             mx = max(nums[i], mx * nums[i]);
8             mn = min(nums[i], mn * nums[i]);
9             res = max(res, mx);
10        }
11        return res;
12    }
13 };

```

再来看一种画风不太一样的解法，这种解法遍历了两次，一次是正向遍历，一次是反向遍历，相当于正向建立一个累加积数组，每次用出现的最大值更新结果`res`，然后再反向建立一个累加积数组，再用出现的最大值更新结果`res`，注意当遇到0的时候，`prod`要重置为1，参见代码如下：

解法5：

```

1 class Solution {
2 public:
3     int maxProduct(vector<int>& nums) {
4         int res = nums[0], prod = 1, n = nums.size();
5         for (int i = 0; i < n; ++i) {
6             res = max(res, prod *= nums[i]);
7             if (nums[i] == 0) prod = 1;
8         }
9         prod = 1;
10        for (int i = n - 1; i >= 0; --i) {
11            res = max(res, prod *= nums[i]);
12            if (nums[i] == 0) prod = 1;
13        }
14        return res;
15    }
16 };

```

153. 寻找旋转有序数组的最小值

Suppose a sorted array is rotated at some pivot unknown to you beforehand.

(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

Find the minimum element.

You may assume no duplicate exists in the array.

这道寻找旋转有序数组的最小值肯定不能通过直接遍历整个数组来寻找，这个方法过于简单粗暴，这样的话，旋不旋转就没有意义。应该考虑将时间复杂度从简单粗暴的O(n)缩小到O(lgn)，这时候二分查找法就浮现在脑海。

首先要判断这个有序数组是否旋转了，通过比较第一个和最后一个数的大小，如果第一个数小，则没有旋转，直接返回这个数。如果第一个数大，就要进一步搜索。我们定义left和right两个指针分别指向开头和结尾，还要找到中间那个数，然后和left指的数比较，如果中间的数大，则继续二分查找右半段数组，反之查找左半段。终止条件是当左右两个指针相邻，返回小的那个。代码如下：

```

1 class Solution {
2 public:
3     int findMin(vector<int> &num) {
4         int left = 0, right = num.size() - 1;
5         if (num[left] > num[right]) {
6             while (left != (right - 1)) {
7                 int mid = (left + right) / 2;
8                 if (num[left] < num[mid]) left = mid;
9                 else right = mid;
10            }
11            return min(num[left], num[right]);
12        }
13        return num[0];
14    }
15 };

```

154. 寻找旋转有序数组的最小值之二

Follow up for "Find Minimum in Rotated Sorted Array":

What if duplicates are allowed?

Would this affect the run-time complexity? How and why?

Suppose a sorted array is rotated at some pivot unknown to you beforehand.

(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

Find the minimum element.

The array may contain duplicates.

寻找旋转有序重复数组的最小值是对之前问题的延伸(<http://www.cnblogs.com/grandyang/p/4032934.html>)，当数组中存在大量的重复数字时，就会破坏二分查找法的机制，我们无法取得O(lgn)的时间复杂度，又将会回到简单粗暴的O(n)，比如如下两种情况：

{2, 2, 2, 2, 2, 2, 2, 2, 0, 1, 1, 2} 和 {2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 2}，我们发现，当第一个数字和最后一个数字，还有中间那个数字全部相等的时候，二分查找法就崩溃了，因为它无法判断到底该去左半边还是右半边。这种情况下，我们将左指针右移一位，略过一个相同数字，这对结果不会产生影响，因为我们只是去掉了相同的，然后对剩余的部分继续用二分查找法，在最坏的情况下，比如数组所有元素都相同，时间复杂度会升到O(n)，参见代码如下：

```

1 class Solution {
2 public:
3     int findMin(vector<int> &nums) {
4         if (nums.empty()) return 0;
5         int left = 0, right = nums.size() - 1, res = nums[0];
6         while (left < right - 1) {
7             int mid = left + (right - left) / 2;
8             if (nums[left] < nums[mid]) {
9                 res = min(res, nums[left]);
10                left = mid + 1;
11            } else if (nums[left] > nums[mid]) {
12                res = min(res, nums[right]);
13                right = mid;
14            } else ++left;
15        }
16        res = min(res, nums[left]);
17        res = min(res, nums[right]);
18        return res;
19    }
20 };

```

CPP

155. 最小栈

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

```

push(x) -- Push element x onto stack.
pop() -- Removes the element on top of the stack.
top() -- Get the top element.
getMin() -- Retrieve the minimum element in the stack.

```

这道最小栈跟原来的栈相比就是多了一个功能，可以返回该栈的最小值。使用两个栈来实现，一个栈来按顺序存储push进来的数据，另一个用来存出现过的最小值。代码如下：

解法1：

CPP

```

1 class MinStack {
2 public:
3     /** initialize your data structure here. */
4     MinStack() {}
5
6     void push(int x) {
7         s1.push(x);
8         if (s2.empty() || x <= s2.top()) s2.push(x);
9     }
10
11    void pop() {
12        if (s1.top() == s2.top()) s2.pop();
13        s1.pop();
14    }
15
16    int top() {
17        return s1.top();
18    }
19
20    int getMin() {
21        return s2.top();
22    }
23
24 private:
25     stack<int> s1, s2;
26 };

```

需要注意的是上面的Java解法中的pop0中，为什么不能用注释掉那两行的写法，我之前也不太明白为啥不能对两个stack同时调用peek0函数来比较，如果是这种写法，那么不管s1和s2对栈顶元素是否相等，永远返回false。这是为什么呢，这我们就要到Java的对于peek的定义了，对于peek0函数的返回值并不是int类型，而是一个Object类型，这是一个基本的对象类型，如果我们直接用==来比较的话，那么肯定不会返回true，因为是两个不同的对象，所以我们一定要先将一个转为int型，然后再和另一个进行比较，这样才能得到我们想要的答案，这也是Java和C++的一个重要的不同点吧。

那么下面我们再来看另一种解法，这种解法只用到了一个栈，还需要一个整型变量min_val来记录当前最小值，初始化为整型最小值，然后如果需要进栈的数字小于等于当前最小值min_val，那么将min_val压入栈，并且将min_val更新为当前数字。在出栈操作时，先将栈顶元素移出栈，再判断该元素是否和min_val相等，相等的话我们将min_val更新为新栈顶元素，再将新栈顶元素移出栈即可，参见代码如下：

解法2：

```
1 class MinStack {
2 public:
3     /** initialize your data structure here. */
4     MinStack() {
5         min_val = INT_MAX;
6     }
7
8     void push(int x) {
9         if (x <= min_val) {
10             st.push(min_val);
11             min_val = x;
12         }
13         st.push(x);
14     }
15
16     void pop() {
17         int t = st.top(); st.pop();
18         if (t == min_val) {
19             min_val = st.top(); st.pop();
20         }
21     }
22
23     int top() {
24         return st.top();
25     }
26
27     int getMin() {
28         return min_val;
29     }
30 private:
31     int min_val;
32     stack<int> st;
33 };
```

156. 二叉树的上下颠倒

Given a binary tree where all the right nodes are either leaf nodes with a sibling (a left node that shares the same parent node) or empty, flip it upside down and turn it into a tree where the original right nodes turned into left leaf nodes. Return the new root.

For example:

Given a binary tree {1,2,3,4,5},

```

1
/
2   3
/
4   5
return the root of the binary tree [4,5,2,#,#,3,1].

```

```

4
/
5   2
/
3   1

```

这道题让我们把一棵二叉树上下颠倒一下，而且限制了右节点要么为空要么一定会有对应的左节点。上下颠倒后原来二叉树的最左子节点变成了根节点，其对应的右节点变成了其左子节点，其父节点变成了其右子节点，相当于顺时针旋转了一下。对于一般树的题都会有迭代和递归两种解法，这道题也不例外，那么我们先来看看递归的解法。对于一个根节点来说，我们的目标是将其左子节点变为根节点，右子节点变为左子节点，原根节点变为右子节点，那么我们首先判断这个根节点是否存在，且其有没有左子节点，如果不满足这两个条件的话，直接返回即可，不需要翻转操作。那么我们不停的对左子节点调用递归函数，直到到达最左子节点开始翻转，翻转好最左子节点后，开始回到上一个左子节点继续翻转即可，直至翻转完整棵树，参见代码如下：

解法1:

```

1 | class Solution {
2 | public:
3 |     TreeNode *upsideDownBinaryTree(TreeNode *root) {
4 |         if (!root || !root->left) return root;
5 |         TreeNode *l = root->left, *r = root->right;
6 |         TreeNode *res = upsideDownBinaryTree(l);
7 |         l->left = r;
8 |         l->right = root;
9 |         root->left = NULL;
10 |        root->right = NULL;
11 |        return res;
12 |    }
13 | };

```

CPP

下面我们来看迭代的方法，和递归方法相反的时，这个是从上往下开始翻转，直至翻转到最左子节点，参见代码如下：

解法2:

```

1 class Solution {
2 public:
3     TreeNode *upsideDownBinaryTree(TreeNode *root) {
4         TreeNode *cur = root, *pre = NULL, *next = NULL, *tmp = NULL;
5         while (cur) {
6             next = cur->left;
7             cur->left = tmp;
8             tmp = cur->right;
9             cur->right = pre;
10            pre = cur;
11            cur = next;
12        }
13        return pre;
14    }
15 };

```

157. 用Read4来读取N个字符

The API: int read4(char *buf) reads 4 characters at a time from a file.

The return value is the actual number of characters read. For example, it returns 3 if there is only 3 characters left in the file.

By using the read4 API, implement the function int read(char *buf, int n) that reads n characters from the file.

Note:

The read function will only be called once for each test case.

这道题给了我们一个Read4函数，每次可以从一个文件中最多读出4个字符，如果文件中的字符不足4个字符时，返回准确的当前剩余的字符数。现在让我们实现一个最多能读取n个字符的函数。这题有迭代和递归的两种解法，我们先来看迭代的方法，思路是我们每4个读一次，然后把读出的结果判断一下，如果为0的话，说明此时的buf已经被读完，跳出循环，直接返回res和n之中的较小值。否则一直读入，直到读完n个字符，循环结束，最后再返回res和n之中的较小值，参见代码如下：

解法1:

```

1 int read4(char *buf);
2
3 class Solution {
4 public:
5     int read(char *buf, int n) {
6         int res = 0;
7         for (int i = 0; i <= n / 4; ++i) {
8             int cur = read4(buf + res);
9             if (cur == 0) break;
10            res += cur;
11        }
12        return min(res, n);
13    }
14 };

```

下面来看递归的解法，这个也不难，我们对buf调用read4函数，然后判断返回值t，如果返回值t大于等于n，说明此时n不大于4，直接返回n即可，如果此返回值t小于4，直接返回t即可，如果都不是，则直接返回调用递归函数加上4，其中递归函数的buf应往后推4个字符，此时n变成n-4即可，参见代码如下：

解法2:

```

1 int read4(char *buf);
2
3 class Solution {
4 public:
5     int read(char *buf, int n) {
6         int t = read4(buf);
7         if (t >= n) return n;
8         if (t < 4) return t;
9         return 4 + read(&buf[4], n - 4);
10    }
11 };

```

158. 用Read4来读取N个字符之二 - 多次调用

The API: `int read4(char *buf)` reads 4 characters at a time from a file.

The return value is the actual number of characters read. For example, it returns 3 if there is only 3 characters left in the file.

By using the `read4` API, implement the function `int read(char *buf, int n)` that reads `n` characters from the file.

Note:

The `read` function may be called multiple times.

这道题是之前那道Read N Characters Given Read4的拓展，那道题说`read`函数只能调用一次，而这道题说`read`函数可以调用多次，那么难度就增加了，为了更简单直观的说明问题，我们举个简单的例子吧，比如：

`buf = "ab", [read(1), read(2)]`, 返回 `["a", "b"]`

那么第一次调用`read(1)`后，从`buf`中读出一个字符，那么就是第一个字符`a`，然后又调用了一个`read(2)`，想取出两个字符，但是`buf`中只剩一个`b`了，所以就把取出的结果就是`b`。再来看一个例子：

`buf = "a", [read(0), read(1), read(2)]`, 返回 `["", "a", ""]`

第一次调用`read(0)`，不取任何字符，返回空，第二次调用`read(1)`，取一个字符，`buf`中只有一个字符，取出为`a`，然后再调用`read(2)`，想取出两个字符，但是`buf`中没有字符了，所以取出为空。

但是这道题我不太懂的地方是明明函数返回的是`int`类型啊，为啥OJ的output都是`vector<char>`类的，然后我就在网上找了下面两种能通过OJ的解法，大概看了看，也是看的个一知半解，貌似是用两个变量`readPos`和`writePos`来记录读取和写的位置，`i`从0到`n`开始循环，如果此时读和写的位置相同，那么我们调用`read4`函数，将结果赋给`writePos`，把`readPos`置零，如果`writePos`为零的话，说明`buf`中没有东西了，返回当前的坐标`i`。然后我们用内置的`buff`变量的`readPos`位置覆盖输入字符串`buf`的`i`位置，如果完成遍历，返回`n`，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int read(char *buf, int n) {
4         for (int i = 0; i < n; ++i) {
5             if (readPos == writePos) {
6                 writePos = read4(buff);
7                 readPos = 0;
8                 if (writePos == 0) return i;
9             }
10            buf[i] = buff[readPos++];
11        }
12        return n;
13    }
14 private:
15     int readPos = 0, writePos = 0;
16     char buff[4];
17 };

```

下面这种方法和上面的方法基本相同，稍稍改变了些解法，使得看起来更加简洁一些：

解法2：

```

1 class Solution {
2 public:
3     int read(char *buf, int n) {
4         int i = 0;
5         while (i < n && (readPos < writePos || (readPos = 0) < (writePos = read4(buff))))
6             buf[i++] = buff[readPos++];
7         return i;
8     }
9     char buff[4];
10    int readPos = 0, writePos = 0;
11 };

```

159. 最多有两个不同字符的最长子串

Given a string S, find the length of the longest substring T that contains at most two distinct characters.

For example,

Given S = "eceba",

T is "ece" which its length is 3.

这道题给我们一个字符串，让我们求最多有两个不同字符的最长子串。那么我们首先想到的是用哈希表来做，哈希表记录每个字符的出现次数，然后如果哈希表中的映射数量超过两个的时候，我们需要删掉一个映射，比如此时哈希表中e有2个，c有1个，此时把b也存入了哈希表，那么就有三对映射了，这时我们的left是0，先从e开始，映射值减1，此时e还有1个，不删除，left自增1。这是哈希表里还有三对映射，此时left是1，那么到c了，映射值减1，此时e映射为0，将e从哈希表中删除，left自增1，然后我们更新结果为i - left + 1，以此类推直至遍历完整个字符串，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int lengthOfLongestSubstringTwoDistinct(string s) {
4         int res = 0, left = 0;
5         unordered_map<char, int> m;
6         for (int i = 0; i < s.size(); ++i) {
7             ++m[s[i]];
8             while (m.size() > 2) {
9                 if (--m[s[left]] == 0) m.erase(s[left]);
10                ++left;
11            }
12            res = max(res, i - left + 1);
13        }
14        return res;
15    }
16 };

```

我们除了用哈希表来映射字符出现的个数，我们还可以映射每个字符最新的坐标，比如题目中的例子"eceba"，遇到第一个e，映射其坐标0，遇到c，映射其坐标1，遇到第二个e时，映射其坐标2，当遇到b时，映射其坐标3，每次我们都判断当前哈希表中的映射数，如果大于2的时候，那么我们需要删掉一个映射，我们还是从left=0时开始向右找，我们看每个字符在哈希表中的映射值是否等于当前坐标left，比如第一个e，哈希表此时映射值为2，不等于left的0，那么left自增1，遇到c的时候，哈希表中c的映射值是1，和此时的left相同，那么我们把c删掉，left自增1，再更新结果，以此类推直至遍历完整个字符串，参见代码如下：

```

1 class Solution {
2 public:
3     int lengthOfLongestSubstringTwoDistinct(string s) {
4         int res = 0, left = 0;
5         unordered_map<char, int> m;
6         for (int i = 0; i < s.size(); ++i) {
7             m[s[i]] = i;
8             while (m.size() > 2) {
9                 if (m[s[left]] == left) m.erase(s[left]);
10                ++left;
11            }
12            res = max(res, i - left + 1);
13        }
14        return res;
15    }
16 };

```

后来又在网上看到了一种解法，这种解法是维护一个sliding window，指针left指向起始位置，right指向window的最后一个位置，用于定位left的下一个跳转位置，思路如下：

1. 若当前字符和前一个字符相同，继续循环。
2. 若不同，看当前字符和right指的字符是否相同
 - (1) 若相同，left不变，右边跳到i - 1
 - (2) 若不同，更新结果，left变为right+1，right变为i - 1

最后需要注意在循环结束后，我们还要比较res和s.size() - left的大小，返回大的，这是由于如果字符串是"ecebaaa"，那么当left=3时，i=5,6的时候，都是继续循环，当i加到7时，跳出了循环，而此时正确答案应为"baaa"这4个字符，而我们的res只更新到了"ece"这3个字符，所以我们最后要判断s.size() - left和res的大小。

另外需要说明的是这种解法仅适用于于不同字符数为2个的情况，如果为k个的话，还是需要用上面两种解法。

解法3：

```

1 class Solution {
2 public:
3     int lengthOfLongestSubstringTwoDistinct(string s) {
4         int left = 0, right = -1, res = 0;
5         for (int i = 1; i < s.size(); ++i) {
6             if (s[i] == s[i - 1]) continue;
7             if (right >= 0 && s[right] != s[i]) {
8                 res = max(res, i - left);
9                 left = right + 1;
10            }
11            right = i - 1;
12        }
13        return max(s.size() - left, res);
14    }
15 };

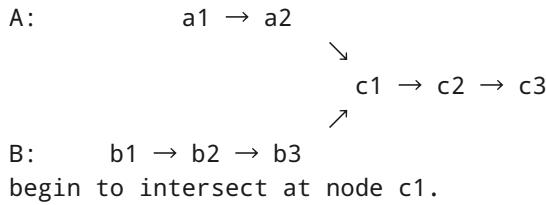
```

CPP

160. 求两个链表的交点

Write a program to find the node at which the intersection of two singly linked lists begins.

For example, the following two linked lists:



Notes:

If the two linked lists have no intersection at all, return null.
 The linked lists must retain their original structure after the function returns.
 You may assume there are no cycles anywhere in the entire linked structure.
 Your code should preferably run in $O(n)$ time and use only $O(1)$ memory.

Credits:

Special thanks to @stellari for adding this problem and creating all test cases.

我还以为以后在不能免费做OJ的题了呢，想不到OJ又放出了不需要买书就能做的题，业界良心啊，哈哈。这道求两个链表的交点题要求执行时间为 $O(n)$ ，则不能利用类似冒泡法原理去暴力查找相同点，事实证明如果链表很长的话，那样的方法效率很低。我也想到会不会是像之前删除重复元素的题一样需要用两个指针来遍历，可是想了好久也没想出来怎么弄。无奈上网搜大神们的解法，发觉其实解法很简单，因为如果两个链长度相同的话，那么对应的一个个比下去就能找到，所以只需要把长链表变短即可。具体算法为：分别遍历两个链表，得到分别对应的长度。然后求长度的差值，把较长的那个链表向后移动这个差值的个数，然后一一比较即可。代码如下：

解法1：

```

1 class Solution {
2 public:
3     ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
4         if (!headA || !headB) return NULL;
5         int lenA = getLength(headA), lenB = getLength(headB);
6         if (lenA < lenB) {
7             for (int i = 0; i < lenB - lenA; ++i) headB = headB->next;
8         } else {
9             for (int i = 0; i < lenA - lenB; ++i) headA = headA->next;
10        }
11        while (headA && headB && headA != headB) {
12            headA = headA->next;
13            headB = headB->next;
14        }
15        return (headA && headB) ? headA : NULL;
16    }
17    int getLength(ListNode* head) {
18        int cnt = 0;
19        while (head) {
20            ++cnt;
21            head = head->next;
22        }
23        return cnt;
24    }
25};

```

这道题还有一种特别巧妙的方法，虽然题目中强调了链表中不存在环，但是我们可以用环的思想来做，我们让两条链表分别从各自的开头开始往后遍历，当其中一条遍历到末尾时，我们跳到另一个条链表的开头继续遍历。两个指针最终会相等，而且只有两种情况，一种情况是在交点处相遇，另一种情况是在各自的末尾的空节点处相等。为什么一定会相等呢，因为两个指针走过的路程相同，是两个链表的长度之和，所以一定会相等。这个思路真的很巧妙，而且更重要的是代码写起来特别的简洁，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
4         if (!headA || !headB) return NULL;
5         ListNode *a = headA, *b = headB;
6         while (a != b) {
7             a = a ? a->next : headB;
8             b = b ? b->next : headA;
9         }
10        return a;
11    }
12};

```

161. 一个编辑距离

Given two strings S and T, determine if they are both one edit distance apart.

这道题是之前那道Edit Distance的拓展，然而这道题并没有那道题难，这道题只让我们判断两个字符串的编辑距离是否为1，那么我们只需分下列三种情况来考虑就行了：

1. 两个字符串的长度之差大于1，那么直接返回False
2. 两个字符串的长度之差等于1，那么长的那个字符串去掉一个字符，剩下的应该和短的字符串相同
3. 两个字符串的长度之差等于0，那么两个字符串对应位置的字符只能有一处不同。

分析清楚了所有的情况，代码就很好写了，参见如下：

解法1：

```

1 class Solution {
2 public:
3     bool isOneEditDistance(string s, string t) {
4         if (s.size() < t.size()) swap(s, t);
5         int m = s.size(), n = t.size(), diff = m - n;
6         if (diff >= 2) return false;
7         else if (diff == 1) {
8             for (int i = 0; i < n; ++i) {
9                 if (s[i] != t[i]) {
10                     return s.substr(i + 1) == t.substr(i);
11                 }
12             }
13             return true;
14         } else {
15             int cnt = 0;
16             for (int i = 0; i < m; ++i) {
17                 if (s[i] != t[i]) ++cnt;
18             }
19             return cnt == 1;
20         }
21     }
22 };

```

CPP

我们实际上可以让代码写的更加简洁，只需要对比两个字符串对应位置上的字符，如果遇到不同的时候，这时我们看两个字符串的长度关系，如果相等，那么我们比较当前位置后的字串是否相同，如果s的长度大，那么我们比较s的下一个位置开始的子串，和t的当前位置开始的子串是否相同，反之如果t的长度大，那么我们比较t的下一个位置开始的子串，和s的当前位置开始的子串是否相同。如果循环结束，都没有找到不同的字符，那么此时我们看两个字符串的长度是否相差1，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool isOneEditDistance(string s, string t) {
4         for (int i = 0; i < min(s.size(), t.size()); ++i) {
5             if (s[i] != t[i]) {
6                 if (s.size() == t.size()) return s.substr(i + 1) == t.substr(i + 1);
7                 else if (s.size() < t.size()) return s.substr(i) == t.substr(i + 1);
8                 else return s.substr(i + 1) == t.substr(i);
9             }
10        }
11        return abs((int)s.size() - (int)t.size()) == 1;
12    }
13 };

```

162. 求数组的局部峰值

A peak element is an element that is greater than its neighbors.

Given an input array where $\text{num}[i] \neq \text{num}[i+1]$, find a peak element and return its index.

The array may contain multiple peaks, in that case return the index to any one of the peaks is fine.

You may imagine that $\text{num}[-1] = \text{num}[n] = -\infty$.

For example, in array [1, 2, 3, 1], 3 is a peak element and your function should return the index number 2.

[click to show spoilers.](#)

Credits:

Special thanks to @ts for adding this problem and creating all test cases.

这道题是求数组的一个峰值，这个峰值可以是局部的最大值，这里用遍历整个数组找最大值肯定会出现Time Limit Exceeded，我们要考虑使用类似于二分查找法来缩短时间，由于只是需要找到任意一个峰值，那么我们在确定二分查找折半后中间那个元素后，和紧跟的那个元素比较下大小，如果大于，则说明峰值在前面，如果小于则在后面。这样就可以找到一个峰值了，代码如下：

解法1：

```

1 class Solution {
2 public:
3     int findPeakElement(vector<int>& nums) {
4         int left = 0, right = nums.size() - 1;
5         while (left < right) {
6             int mid = left + (right - left) / 2;
7             if (nums[mid] < nums[mid + 1]) left = mid + 1;
8             else right = mid;
9         }
10        return right;
11    }
12 };

```

下面这种解法就更加的巧妙了，由于题目中说明了局部峰值一定存在，那么实际上可以从第二个数字开始往后遍历，如果第二个数字比第一个数字小，说明此时第一个数字就是一个局部峰值；否则就往后继续遍历，现在是个递增趋势，如果此时某个数字小于前面那个数字，说明前面数字就是一个局部峰值，返回位置即可。如果循环结束了，说明原数组是个递增数组，返回最后一个位置即可，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     int findPeakElement(vector<int>& nums) {
4         for (int i = 1; i < nums.size(); ++i) {
5             if (nums[i] < nums[i - 1]) return i - 1;
6         }
7         return nums.size() - 1;
8     }
9 };
```

CPP

163. 缺失区间

Given a sorted integer array where the range of elements are [0, 99] inclusive, return its missing ranges.

For example, given [0, 1, 3, 50, 75], return ["2", "4->49", "51->74", "76->99"]

这道题让我们求缺失区间，跟之前那道Summary Ranges很类似，这道题让我们求缺失的空间，给了一个空间的范围[lower upper]，缺失的空间的范围需要在给定的空间范围内。我们首先将lower赋给l，然后开始遍历nums数组，如果i小于nums长度且当前数字小于等于upper，我们让r等于当前数字，否则如果当i等于nums的长度时或者当前数字大于upper时，将r赋为upper+1。然后判断l和r的值，若相同，l自增1，否则当r大于l时，说明缺失空间存在，我们看l和r是否差1，如果是，说明只缺失了一个数字，若不是，则说明缺失了一个区间，我们分别加上数字或者区间即可，参见代码如下：

```
1 class Solution {
2 public:
3     vector<string> findMissingRanges(vector<int>& nums, int lower, int upper) {
4         vector<string> res;
5         int l = lower;
6         for (int i = 0; i <= nums.size(); ++i) {
7             int r = (i < nums.size() && nums[i] <= upper) ? nums[i] : upper + 1;
8             if (l == r) ++l;
9             else if (r > l) {
10                 res.push_back(r - l == 1 ? to_string(l) : to_string(l) + "->" + to_string(r
11 - 1));
12                 l = r + 1;
13             }
14         }
15         return res;
16     }
17 };
```

CPP

164. 求最大间距

Given an unsorted array, find the maximum difference between the successive elements in its sorted form.

Try to solve it in linear time/space.

Return 0 if the array contains less than 2 elements.

You may assume all elements in the array are non-negative integers and fit in the 32-bit signed integer range.

Credits:

Special thanks to @porker2008 for adding this problem and creating all test cases.

遇到这类问题肯定先想到的是要给数组排序，但是题目要求是要线性的时间和空间，那么只能用桶排序或者基排序。这里我用桶排序Bucket Sort来做，首先找出数组的最大值和最小值，然后要确定每个桶的容量，即为 $(\text{最大值} - \text{最小值}) / \text{个数} + 1$ ，在确定桶的个数，即为 $(\text{最大值} - \text{最小值}) / \text{桶的容量} + 1$ ，然后需要在每个桶中找出局部最大值和最小值，而最大间距的两个数不在同一个桶中，而是一个桶的最小值和另一个桶的最大值之间的间距。代码如下：

```

1 class Solution {
2 public:
3     int maximumGap(vector<int> &numss) {
4         if (numss.empty()) return 0;
5         int mx = INT_MIN, mn = INT_MAX, n = numss.size();
6         for (int d : numss) {
7             mx = max(mx, d);
8             mn = min(mn, d);
9         }
10        int size = (mx - mn) / n + 1;
11        int bucket_nums = (mx - mn) / size + 1;
12        vector<int> bucket_min(bucket_nums, INT_MAX);
13        vector<int> bucket_max(bucket_nums, INT_MIN);
14        set<int> s;
15        for (int d : numss) {
16            int idx = (d - mn) / size;
17            bucket_min[idx] = min(bucket_min[idx], d);
18            bucket_max[idx] = max(bucket_max[idx], d);
19            s.insert(idx);
20        }
21        int pre = 0, res = 0;
22        for (int i = 1; i < n; ++i) {
23            if (!s.count(i)) continue;
24            res = max(res, bucket_min[i] - bucket_max[pre]);
25            pre = i;
26        }
27        return res;
28    }
29 };

```

CPP

165. 版本比较

Compare two version numbers version1 and version2.

If version1 > version2 return 1, if version1 < version2 return -1, otherwise return 0.

You may assume that the version strings are non-empty and contain only digits and the . character.

The . character does not represent a decimal point and is used to separate number sequences.

For instance, 2.5 is not "two and a half" or "half way to version three", it is the fifth second-level revision of the second first-level revision.

Here is an example of version numbers ordering:

0.1 < 1.1 < 1.2 < 13.37

Credits:

Special thanks to @ts for adding this problem and creating all test cases.

这道题调试了好久，一直不想上网搜别人的解法，因为感觉自己可以做出来，改来改去最后终于通过了，再上网一搜，发现果然和别人的方法不同，小有成就感。我的思路是：由于两个版本号所含的小数点个数不同，有可能是1和1.1.1比较，还有可能开头有无效0，比如01和1就是相同版本，还有可能末尾无效0，比如1.0和1也是同一版本。对于没有小数点的数字，可以默认为最后一位是小数点，而版本号比较的核心思想是相同位置的数字比较，比如题目给的例子，1.2和13.37比较，我们都知道应该显示1和13比较，13比1大，所以后面的不用再比了，再比如1.1和1.2比较，前面都是1，则比较小数点后面的数字。那么算法就是每次对应取出相同位置的小数点之前所有的字符，把他们转为数字比较，若不同则可直接得到答案，若相同，再对应往下取。如果一个数字已经没有小数点了，则默认取出为0，和另一个比较，这样也解决了末尾无效0的情况。代码如下：

解法1：

```

1 class Solution {
2 public:
3     int compareVersion(string version1, string version2) {
4         int n1 = version1.size(), n2 = version2.size();
5         int i = 0, j = 0, d1 = 0, d2 = 0;
6         string v1, v2;
7         while (i < n1 || j < n2) {
8             while (i < n1 && version1[i] != '.') {
9                 v1.push_back(version1[i++]);
10            }
11            d1 = atoi(v1.c_str());
12            while (j < n2 && version2[j] != '.') {
13                v2.push_back(version2[j++]);
14            }
15            d2 = atoi(v2.c_str());
16            if (d1 > d2) return 1;
17            else if (d1 < d2) return -1;
18            v1.clear(); v2.clear();
19            ++i; ++j;
20        }
21        return 0;
22    }
23 };

```

CPP

当然我们也可以不使用将字符串转为整型的atoi函数，我们可以一位一位的累加，参加如下代码：

解法2：

```

1 class Solution {
2 public:
3     int compareVersion(string version1, string version2) {
4         int n1 = version1.size(), n2 = version2.size();
5         int i = 0, j = 0, d1 = 0, d2 = 0;
6         while (i < n1 || j < n2) {
7             while (i < n1 && version1[i] != '.') {
8                 d1 = d1 * 10 + version1[i++]- '0';
9             }
10            while (j < n2 && version2[j] != '.') {
11                d2 = d2 * 10 + version2[j++]- '0';
12            }
13            if (d1 > d2) return 1;
14            else if (d1 < d2) return -1;
15            d1 = d2 = 0;
16            ++i; ++j;
17        }
18        return 0;
19    }
20 };

```

由于这道题我们需要将版本号以'.'分开，那么我们可以借用强大的字符串流stringstream的功能来实现分段和转为整数，使用这种方法写的代码很简洁，如下所示：

解法3：

```

1 class Solution {
2 public:
3     int compareVersion(string version1, string version2) {
4         istringstream v1(version1 + "."), v2(version2 + ".");
5         int d1 = 0, d2 = 0;
6         char dot = '.';
7         while (v1.good() || v2.good()) {
8             if (v1.good()) v1 >> d1 >> dot;
9             if (v2.good()) v2 >> d2 >> dot;
10            if (d1 > d2) return 1;
11            else if (d1 < d2) return -1;
12            d1 = d2 = 0;
13        }
14        return 0;
15    }
16 };

```

最后我们来看一种用C语言的字符串指针来实现的方法，这个方法的关键是用到将字符串转为长整型的strtol函数，关于此函数的用法可以参见我的另一篇博客strtol 函数用法。参见代码如下：

解法4：

```

1 class Solution {
2 public:
3     int compareVersion(string version1, string version2) {
4         int res = 0;
5         char *v1 = (char*)version1.c_str(), *v2 = (char*)version2.c_str();
6         while (res == 0 && (*v1 != '\0' || *v2 != '\0')) {
7             long d1 = *v1 == '\0' ? 0 : strtol(v1, &v1, 10);
8             long d2 = *v2 == '\0' ? 0 : strtol(v2, &v2, 10);
9             if (d1 > d2) return 1;
10            else if (d1 < d2) return -1;
11            else {
12                if (*v1 != '\0') ++v1;
13                if (*v2 != '\0') ++v2;
14            }
15        }
16        return res;
17    }
18 };

```

166. 分数转循环小数

Given two integers representing the numerator and denominator of a fraction, return the fraction in string format.

If the fractional part is repeating, enclose the repeating part in parentheses.

For example,

Given numerator = 1, denominator = 2, return "0.5".
 Given numerator = 2, denominator = 1, return "2".
 Given numerator = 2, denominator = 3, return "0.(6)".

Credits:

Special thanks to @Shangrila for adding this problem and creating all test cases.

这道题还是比较有意思的，开始还担心万一结果是无限不循环小数怎么办，百度之后才发现原来可以写成分数的都是有理数，而有理数要么是有限的，要么是无限循环小数，无限不循环的叫无理数，例如圆周率pi或自然数e等，小学数学没学好，汗！由于还存在正负情况，处理方式是按正数处理，符号最后在判断，那么我们需要把除数和被除数取绝对值，那么问题就来了：由于整型数INT的取值范围是-2147483648 ~ 2147483647，而对-2147483648取绝对值就会超出范围，所以我们需要先转为long long型再取绝对值。那么怎么样找循环呢，肯定是再得到一个数字后要看看之前有没有出现这个数。为了节省搜索时间，我们采用哈希表来存数每个小数位上的数字。还有一个小技巧，由于我们要算出小数每一位，采取的方法是每次把余数乘10，再除以除数，得到的商即为小数的下一位数字。等到新算出来的数字在之前出现过，则在循环开始处加左括号，结束处加右括号。代码如下：

```

1 class Solution {
2 public:
3     string fractionToDecimal(int numerator, int denominator) {
4         int s1 = numerator >= 0 ? 1 : -1;
5         int s2 = denominator >= 0 ? 1 : -1;
6         long long num = abs( (long long)numerator );
7         long long den = abs( (long long)denominator );
8         long long out = num / den;
9         long long rem = num % den;
10        unordered_map<long long, int> m;
11        string res = to_string(out);
12        if (s1 * s2 == -1 && (out > 0 || rem > 0)) res = "-" + res;
13        if (rem == 0) return res;
14        res += ".";
15        string s = "";
16        int pos = 0;
17        while (rem != 0) {
18            if (m.find(rem) != m.end()) {
19                s.insert(m[rem], "(");
20                s += ")";
21                return res + s;
22            }
23            m[rem] = pos;
24            s += to_string((rem * 10) / den);
25            rem = (rem * 10) % den;
26            ++pos;
27        }
28        return res + s;
29    }
30};

```

167. 两数之和之二 - 输入数组有序

Given an array of integers that is already sorted in ascending order, find two numbers such that they add up to a specific target number.

The function twoSum should return indices of the two numbers such that they add up to the target, where index1 must be less than index2. Please note that your returned answers (both index1 and index2) are not zero-based.

You may assume that each input would have exactly one solution.

Input: numbers={2, 7, 11, 15}, target=9

Output: index1=1, index2=2

这又是一道Two Sum的衍生题，作为LeetCode开山之题，我们务必要把Two Sum及其所有的衍生题都拿下，这道题其实应该更容易一些，因为给定的数组是有序的，而且题目中限定了一定会有解，我最开始想到的方法是二分法来搜索，因为一定有解，而且数组是有序的，那么第一个数字肯定要小于目标值target，那么我们每次用二分法来搜索target - numbers[i]即可，代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<int> twoSum(vector<int>& numbers, int target) {
4         for (int i = 0; i < numbers.size(); ++i) {
5             int t = target - numbers[i], left = i + 1, right = numbers.size();
6             while (left < right) {
7                 int mid = left + (right - left) / 2;
8                 if (numbers[mid] == t) return {i + 1, mid + 1};
9                 else if (numbers[mid] < t) left = mid + 1;
10                else right = mid;
11            }
12        }
13        return {};
14    }
15 };

```

但是上面那种方法并不efficient，时间复杂度是O(nlgn)，我们再来看一种O(n)的解法，我们只需要两个指针，一个指向开头，一个指向末尾，然后向中间遍历，如果指向的两个数相加正好等于target的话，直接返回两个指针的位置即可，若小于target，左指针右移一位，若大于target，右指针左移一位，以此类推直至两个指针相遇停止，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<int> twoSum(vector<int>& numbers, int target) {
4         int l = 0, r = numbers.size() - 1;
5         while (l < r) {
6             int sum = numbers[l] + numbers[r];
7             if (sum == target) return {l + 1, r + 1};
8             else if (sum < target) ++l;
9             else --r;
10        }
11        return {};
12    }
13 };

```

168. 求Excel表列名称

Given a positive integer, return its corresponding column title as appear in an Excel sheet.

For example:

```

1 -> A
2 -> B
3 -> C
...
26 -> Z
27 -> AA
28 -> AB

```

Credits:

Special thanks to @ifanchu for adding this problem and creating all test cases.

此题和Excel Sheet Column Number 求Excel表列序号是一起的，但是我在这题上花的时间远比上面一道多，起始原理都一样，就是一位一位的求，此题从低位往高位求，每进一位，则把原数缩小26倍，再对26取余，之后减去余数，再缩小26倍，以此类推，可以求出各个位置上的字母。最后只需将整个字符串翻转一下即可。代码如下：

解法1:

```

1 class Solution {
2 public:
3     string convertToTitle(int n) {
4         string res = "";
5         while (n) {
6             if (n % 26 == 0) {
7                 res += 'Z';
8                 n -= 26;
9             }
10            else {
11                res += n%26 - 1 + 'A';
12                n -= n%26;
13            }
14            n /= 26;
15        }
16        reverse(res.begin(), res.end());
17        return res;
18    }
19 };

```

CPP

然后我们可以写的更简洁一些：

解法2:

```

1 class Solution {
2 public:
3     string convertToTitle(int n) {
4         string res;
5         while (n) {
6             res += --n % 26 + 'A';
7             n /= 26;
8         }
9         return string(res.rbegin(), res.rend());
10    }
11 };

```

CPP

这道题还可以用递归来解，而且可以丧心病狂的压缩到一行代码来解：

解法3:

```

1 class Solution {
2 public:
3     string convertToTitle(int n) {
4         return n == 0 ? "" : convertToTitle(n / 26) + (char)(--n % 26 + 'A');
5     }
6 };

```

CPP

169. 求众数

Given an array of size n , find the majority element. The majority element is the element that appears more than $\lfloor n/2 \rfloor$ times.

You may assume that the array is non-empty and the majority element always exist in the array.

Example 1:

Input: [3,2,3]

Output: 3

Example 2:

Input: [2,2,1,1,1,2,2]

Output: 2

这是求众数的问题，有很多种解法，其中我感觉比较好的有两种，一种是用哈希表，这种方法需要 $O(n)$ 的时间和空间，另一种是用一种叫摩尔投票法 Moore Voting，需要 $O(n)$ 的时间和 $O(1)$ 的空间，比前一种方法更好。这种投票法先将第一个数字假设为众数，然后把计数器设为1，比较下一个数和此数是否相等，若相等则计数器加一，反之减一。然后看此时计数器的值，若为零，则将下一个值设为候选众数。以此类推直到遍历完整个数组，当前候选众数即为该数组的众数。不仔细弄懂摩尔投票法的精髓的话，过一阵子还是会忘记的，首先要明确的是这个办法是有前提的，就是数组中一定要有众数的存在才能使用，下面我们来看本算法的思路，这是一种先假设候选者，然后再进行验证的算法。我们现将数组中的第一个数假设为众数，然后进行统计其出现的次数，如果遇到同样的数，则计数器自增1，否则计数器自减1，如果计数器减到了0，则更换下一个数字为候选者。这是一个很巧妙的设定，也是本算法的精髓所在，为啥遇到不同的要计数器减1呢，为啥减到0了又要更换候选者呢？首先是有那个强大的前提存在，一定会有一个出现超过半数的数字存在，那么如果计数器减到0了话，说明目前不是候选者数字的个数已经跟候选者的出现个数相同了，那么这个候选者已经很weak，不一定能出现超过半数，我们选择更换当前的候选者。那有可能你会有疑问，那万一后面又大量的出现了之前的候选者怎么办，不需要担心，如果之前的候选者在后面大量出现的话，其又会重新变为候选者，直到最终验证成为正确的众数，佩服算法的提出者啊，代码如下：

解法1:

```
1 class Solution {
2 public:
3     int majorityElement(vector<int>& nums) {
4         int res = 0, cnt = 0;
5         for (int num : nums) {
6             if (cnt == 0) {res = num; ++cnt;}
7             else (num == res) ? ++cnt : --cnt;
8         }
9         return res;
10    }
11};
```

CPP

下面这种解法利用到了位操作Bit Manipulation来解，将中位数按位来建立，从0到31位，每次统计下数组中该位上0和1的个数，如果1多，那么我们将结果res中该位变为1，最后累加出来的res就是中位数了，相当赞的方法，这种思路尤其在这道题的延伸Majority Element II中有重要的应用，参见代码如下：

解法2:

```

1 class Solution {
2 public:
3     int majorityElement(vector<int>& nums) {
4         int res = 0, n = nums.size();
5         for (int i = 0; i < 32; ++i) {
6             int ones = 0, zeros = 0;
7             for (int num : nums) {
8                 if (ones > n / 2 || zeros > n / 2) break;
9                 if ((num & (1 << i)) != 0) ++ones;
10                else ++zeros;
11            }
12            if (ones > zeros) res |= (1 << i);
13        }
14        return res;
15    }
16 };

```

170. 两数之和之三 - 数据结构设计

Design and implement a TwoSum class. It should support the following operations: add and find.

add - Add the number to an internal data structure.

find - Find if there exists any pair of numbers which sum is equal to the value.

For example,

add(1); add(3); add(5);

find(4) -> true

find(7) -> false

这道题让我们设计一个Two Sum的数据结构，跟LeetCode的第一道题Two Sum没有什么太大的不一样，作为LeetCode的首题，Two Sum的名气不小啊，正所谓平生不会TwoSum，刷尽LeetCode也枉然。记得原来在背单词的时候，总是记得第一个单词是abandon，结果有些人背来背去还在abandon，有时候想想刷题其实跟背GRE红宝书没啥太大的区别，都是一个熟练功夫，并不需要有多高的天赋，只要下足功夫，都能达到一个很不错的水平，套用一句鸡汤词来激励下吧，“有些时候我们的努力程度根本达不到需要拼天赋的地步”，好了，不闲扯了，来看题吧。不过这题也没啥可讲的，会做Two Sum的这题就很简单了，我们先来看用哈希表的解法，我们把每个数字和其出现的次数建立映射，然后我们遍历哈希表，对于每个值，我们先求出此值和目标值之间的差值t，然后我们需要分两种情况来看，如果当前值不等于差值t，那么只要哈希表中有差值t就返回True，或者是当差值t等于当前值时，如果此时哈希表的映射次数大于1，则表示哈希表中还有另一个和当前值相等的数字，二者相加就是目标值，参见代码如下：

解法1:

```

1 class TwoSum {
2 public:
3     void add(int number) {
4         ++m[number];
5     }
6     bool find(int value) {
7         for (auto a : m) {
8             int t = value - a.first;
9             if ((t != a.first && m.count(t)) || (t == a.first && a.second > 1)) {
10                 return true;
11             }
12         }
13         return false;
14     }
15 private:
16     unordered_map<int, int> m;
17 };

```

另一种解法不用哈希表，而是unordered_multiset来做，但是原理和上面一样，参见代码如下：

解法2：

```

1 class TwoSum {
2 public:
3     void add(int number) {
4         s.insert(number);
5     }
6     bool find(int value) {
7         for (auto a : s) {
8             int cnt = a == value - a ? 1 : 0;
9             if (s.count(value - a) > cnt) {
10                 return true;
11             }
12         }
13         return false;
14     }
15 private:
16     unordered_multiset<int> s;
17 };

```

171. 求Excel表列序号

Related to question Excel Sheet Column Title

Given a column title as appear in an Excel sheet, return its corresponding column number.

For example:

```
A -> 1
B -> 2
C -> 3
...
Z -> 26
AA -> 27
AB -> 28
```

Credits:

Special thanks to @ts for adding this problem and creating all test cases.

这题实际上相当于一种二十六进制转十进制的问题，并不难，只要一位一位的转换即可。代码如下：

```
1 class Solution {
2 public:
3     int titleToNumber(string s) {
4         int n = s.size();
5         int res = 0;
6         int tmp = 1;
7         for (int i = n; i >= 1; --i) {
8             res += (s[i - 1] - 'A' + 1) * tmp;
9             tmp *= 26;
10        }
11        return res;
12    }
13};
```

CPP

172. 求阶乘末尾零的个数

Given an integer n, return the number of trailing zeroes in n!.

Note: Your solution should be in logarithmic time complexity.

Credits:

Special thanks to @ts for adding this problem and creating all test cases.

这道题并没有什么难度，是让求一个数的阶乘末尾0的个数，也就是要找乘数中10的个数，而10可分解为2和5，而我们可知2的数量又远大于5的数量，那么此题即便为找出5的个数。仍需注意的一点就是，像25,125，这样的不只含有一个5的数字需要考虑进去。代码如下：

解法1：

```

1 class Solution {
2 public:
3     int trailingZeroes(int n) {
4         int res = 0;
5         while (n) {
6             res += n / 5;
7             n /= 5;
8         }
9         return res;
10    }
11 };

```

这题还有递归的解法，思路和上面完全一样，写法更简洁了，一行搞定碉堡了。

解法2：

```

1 class Solution {
2 public:
3     int trailingZeroes(int n) {
4         return n == 0 ? 0 : n / 5 + trailingZeroes(n / 5);
5     }
6 };

```

173. 二叉搜索树迭代器

Implement an iterator over a binary search tree (BST). Your iterator will be initialized with the root node of a BST.

Calling `next()` will return the next smallest number in the BST.

Note: `next()` and `hasNext()` should run in average $O(1)$ time and uses $O(h)$ memory, where h is the height of the tree.

Credits:

Special thanks to @ts for adding this problem and creating all test cases.

这道题主要就是考二叉树的中序遍历的非递归形式，需要额外定义一个栈来辅助，二叉搜索树的建树规则就是左<根<右，用中序遍历即可从小到大取出所有节点。代码如下：

```
1 class BSTIterator {
2     public:
3         BSTIterator(TreeNode *root) {
4             while (root) {
5                 s.push(root);
6                 root = root->left;
7             }
8         }
9
10        /** @return whether we have a next smallest number */
11        bool hasNext() {
12            return !s.empty();
13        }
14
15        /** @return the next smallest number */
16        int next() {
17            TreeNode *n = s.top();
18            s.pop();
19            int res = n->val;
20            if (n->right) {
21                n = n->right;
22                while (n) {
23                    s.push(n);
24                    n = n->left;
25                }
26            }
27            return res;
28        }
29    private:
30        stack<TreeNode*> s;
31 }
```

174. 地牢游戏

The demons had captured the princess (P) and imprisoned her in the bottom-right corner of a dungeon. The dungeon consists of M x N rooms laid out in a 2D grid. Our valiant knight (K) was initially positioned in the top-left room and must fight his way through the dungeon to rescue the princess.

The knight has an initial health point represented by a positive integer. If at any point his health point drops to 0 or below, he dies immediately.

Some of the rooms are guarded by demons, so the knight loses health (negative integers) upon entering these rooms; other rooms are either empty (0's) or contain magic orbs that increase the knight's health (positive integers).

In order to reach the princess as quickly as possible, the knight decides to move only rightward or downward in each step.

Write a function to determine the knight's minimum initial health so that he is able to rescue the princess.

For example, given the dungeon below, the initial health of the knight must be at least 7 if he follows the optimal path RIGHT-> RIGHT -> DOWN -> DOWN.

```
-2(K)      -3      3
-5          -10     1
10         30     -5 (P)
```

Notes:

The knight's health has no upper bound.

Any room can contain threats or power-ups, even the first room the knight enters and the bottom-right room where the princess is imprisoned.

Credits:

Special thanks to @stellari for adding this problem and creating all test cases.

这道王子救公主的题还是蛮新颖的，我最初的想法是比较右边和下边的数字的大小，去大的那个，但是这个算法对某些情况不成立，比如下面的情况：

```
1(K)      -3      3
0          -2      0
-3        -3     -3(P)
```

如果按我的那种算法走的路径为 1 -> 0 -> -2 -> 0 -> -3，这样的话骑士的起始血量要为5，而正确的路径应为 1 -> -3 -> 3 -> 0 -> -3，这样骑士的骑士血量只需为3。无奈只好上网看大神的解法，发现统一都是用动态规划Dynamic Programming来做，建立一个二维数组dp，其中 $dp[i][j]$ 用来表示当前位置 (i, j) 出发的起始血量，最先处理的是公主所在的房间的起始生命值，然后慢慢向第一个房间扩散，不断的得到各个位置的最优的生命值。逆向推正是本题的精髓所在啊，仔细想想也是，如果从起始位置开始遍历，我们并不知道初始时应该初始化的血量，但是到达公主房间后，我们知道血量至少不能小于1，如果公主房间还需要掉血的话，那么掉血后剩1才能保证起始位置的血量最小。那么下面来推导状态转移方程，首先考虑每个位置的血量是由什么决定的，骑士会挂主要是因为去了下一个房间时，掉血量大于本身的血值，而能去的房间只有右边和下边，所以当前位置的血量是由右边和下边房间的可生存血量决定的，进一步来说，应该是由较小的可生存血量决定的，因为较我们需要起始血量尽可能的少，所以用较小的可生存血量减去当前房间的数字，如果是非正数的话，说明当前房间的数字是整数，那么当前房间的生存血量可以是1，所以我们的状态转移方程是 $dp[i][j] = \max(1, \min(dp[i+1][j], dp[i][j+1]) - dungeon[i][j])$ 。为了更好的处理边界情况，我们的二维dp数组比原数组的行数列数均多1个，先都初始化为整型数最大值INT_MAX，由于我们知道到达公主房间后，骑士火拼完的血量至少为1，那么此时公主房间的右边和下边房间里的数字我们就都设置为1，这样到达公主房间的生存血量就是1减去公主房间的数字和1相比较，取较大值，就没有问题了，代码如下：

解法1:

```

1 class Solution {
2 public:
3     int calculateMinimumHP(vector<vector<int>>& dungeon) {
4         int m = dungeon.size(), n = dungeon[0].size();
5         vector<vector<int>> dp(m + 1, vector<int>(n + 1, INT_MAX));
6         dp[m][n - 1] = 1; dp[m - 1][n] = 1;
7         for (int i = m - 1; i >= 0; --i) {
8             for (int j = n - 1; j >= 0; --j) {
9                 dp[i][j] = max(1, min(dp[i + 1][j], dp[i][j + 1])) - dungeon[i][j];
10            }
11        }
12        return dp[0][0];
13    }
14 };

```

CPP

我们可以对空间进行优化，使用一个一维的dp数组，并且不停的覆盖原有的值，参见代码如下：

解法2:

```

1 class Solution {
2 public:
3     int calculateMinimumHP(vector<vector<int>>& dungeon) {
4         int m = dungeon.size(), n = dungeon[0].size();
5         vector<int> dp(n + 1, INT_MAX);
6         dp[n - 1] = 1;
7         for (int i = m - 1; i >= 0; --i) {
8             for (int j = n - 1; j >= 0; --j) {
9                 dp[j] = max(1, min(dp[j], dp[j + 1])) - dungeon[i][j];
10            }
11        }
12        return dp[0];
13    }
14 };
15

```

CPP

175. 联合两表

Table: Person

Column Name	Type
PersonId	int
FirstName	varchar
LastName	varchar

PersonId is the primary key column for this table.

Table: Address

Column Name	Type
AddressId	int
PersonId	int
City	varchar
State	varchar

AddressId is the primary key column for this table.

Write a SQL query for a report that provides the following information for each person in the Person table, regardless if there is an address for each of those people:

FirstName, LastName, City, State

LeetCode还出了是来到数据库的题，来那么也来做做吧，这道题是第一道，相对来说比较简单，是一道两表联合查找的问题，我们需要用到Join操作，关于一些Join操作可以看我之前的博客SQL Left Join, Right Join, Inner Join, and Natural Join 各种Join小结，最直接的方法就是用Left Join来做，根据PersonId这项来把两个表联合起来：

解法1:

```
1 | SELECT Person.FirstName, Person.LastName, Address.City, Address.State FROM Person LEFT JOIN
   Address ON Person.PersonId = Address.PersonId;
```

在使用Left Join时，我们也可以使用关键Using来声明我们相用哪个列名来进行联合：

解法2:

```
1 | SELECT Person.FirstName, Person.LastName, Address.City, Address.State FROM Person LEFT JOIN
   Address USING(PersonId);
```

或者我们可以加上Natural关键字，这样我们就不用声明具体的列，MySQL可以自行搜索相同的列：

解法3:

```
1 | SELECT Person.FirstName, Person.LastName, Address.City, Address.State FROM Person NATURAL
   LEFT JOIN Address;
```

176. 第二高薪水

Write a SQL query to get the second highest salary from the Employee table.

Id	Salary
1	100
2	200
3	300

For example, given the above Employee table, the second highest salary is 200. If there is no second highest salary, then the query should return null.

这道题让我们找表中某列第二大的数，这道题有很多种解法，先来看一种使用Limit和Offset两个关键字的解法，MySQL中Limit后面的数字限制了我们返回数据的个数，Offset是偏移量，那么如果我们想找第二高薪水，我们首先可以先对薪水进行降序排列，然后我们将Offset设为1，那么就是从第二个开始，也就是第二高薪水，然后我们将Limit设为1，就是只取出第二高薪水，如果将Limit设为2，那么就将第二高和第三高薪水都取出来：

解法1:

```
1 | SELECT Salary FROM Employee GROUP BY Salary
2 | UNION ALL (SELECT NULL AS Salary)
3 | ORDER BY Salary DESC LIMIT 1 OFFSET 1;
```

SQL

我们也可以使用Max函数来做，这个返回最大值，逻辑是我们取出的不包含最大值的数字中的最大值，即为第二大值：

解法2:

```
1 | SELECT MAX(Salary) FROM Employee
2 | WHERE Salary NOT IN
3 | (SELECT MAX(Salary) FROM Employee);
```

SQL

下面这种方法和上面基本一样，就是用小于号<代替了Not in关键字，效果相同：

解法3:

```
1 | SELECT MAX(Salary) FROM Employee
2 | Where Salary <
3 | (SELECT MAX(Salary) FROM Employee);
```

SQL

最后来看一种可以扩展到找到第N高的薪水的方法，只要将下面语句中的1改为N-1即可，第二高的薪水带入N-1就是1，下面语句的逻辑是，假如我们要找第二高的薪水，那么我们允许其中一个最大值存在，然后在其余的数字中找出最大的，即为整个的第二大的值；

解法4:

```
1 | SELECT MAX(Salary) FROM Employee E1
2 | WHERE 1 =
3 | (SELECT COUNT(DISTINCT(E2.Salary)) FROM Employee E2
4 | WHERE E2.Salary > E1.Salary);
```

SQL

177. 第N高薪水

Write a SQL query to get the nth highest salary from the Employee table.

Id	Salary
1	100
2	200
3	300

For example, given the above Employee table, the nth highest salary where n = 2 is 200. If there is no nth highest salary, then the query should return null.

这道题是之前那道Second Highest Salary的拓展，根据之前那道题的做法，我们可以很容易的将其推展为N，根据对Second Highest Salary中解法一的分析，我们只需要将OFFSET后面的1改为N-1就行了，但是这样MySQL会报错，估计不支持运算，那么我们可以在前面加一个SET N = N - 1，将N先变成N-1再做也是一样的：

解法1：

```
1 CREATE FUNCTION getNthHighestSalary(N INT) RETURNS INT          SQL
2 BEGIN
3     SET N = N - 1;
4     RETURN (
5         SELECT DISTINCT Salary FROM Employee GROUP BY Salary
6         ORDER BY Salary DESC LIMIT 1 OFFSET N
7     );
8 END
```

根据对Second Highest Salary中解法四的分析，我们只需要将其1改为N-1即可，这里却支持N-1的计算，参见代码如下：

解法2：

```
1 CREATE FUNCTION getNthHighestSalary(N INT) RETURNS INT          SQL
2 BEGIN
3     RETURN (
4         SELECT MAX(Salary) FROM Employee E1
5         WHERE N - 1 =
6             (SELECT COUNT(DISTINCT(E2.Salary)) FROM Employee E2
7             WHERE E2.Salary > E1.Salary)
8     );
9 END
```

当然我们也可以通过将最后的>改为 \geq ，这样我们就可以将N-1换成N了：

解法3：

```
1 CREATE FUNCTION getNthHighestSalary(N INT) RETURNS INT          SQL
2 BEGIN
3     RETURN (
4         SELECT MAX(Salary) FROM Employee E1
5         WHERE N =
6             (SELECT COUNT(DISTINCT(E2.Salary)) FROM Employee E2
7             WHERE E2.Salary >= E1.Salary)
8     );
9 END
```

178. 分数排行

Write a SQL query to rank scores. If there is a tie between two scores, both should have the same ranking. Note that after a tie, the next ranking number should be the next consecutive integer value. In other words, there should be no "holes" between ranks.

Id	Score
1	3.50
2	3.65
3	4.00
4	3.85
5	4.00
6	3.65

For example, given the above Scores table, your query should generate the following report (order by highest score):

Score	Rank
4.00	1
4.00	1
3.85	2
3.65	3
3.65	3
3.50	4

这道题给了我们一个分数表，让我们给分数排序，要求是相同的分数在相同的名次，下一个分数在相连的下一个名次，中间不能有空缺数字，这道题我是完全照着史蒂芬大神的帖子来写的，膜拜大神中...大神总结了四种方法，那么我们一个一个的来膜拜学习，首先看第一种解法，解题的思路是对于每一个分数，找出表中有多少个大于或等于该分数的不同的分数，然后按降序排列即可，参见代码如下：

解法1：

```
1 | SELECT Score,
2 | (SELECT COUNT(DISTINCT Score) FROM Scores WHERE Score >= s.Score) Rank
3 | FROM Scores s ORDER BY Score DESC;
```

SQL

跟上面的解法思想相同，就是写法上略有不同：

解法2：

```
1 | SELECT Score,
2 | (SELECT COUNT(*) FROM (SELECT DISTINCT Score s FROM Scores) t WHERE s >= Score) Rank
3 | FROM Scores ORDER BY Score DESC;
```

SQL

下面这种解法使用了内交，Join是Inner Join的简写形式，自己和自己内交，条件是右表的分数大于等于左表，然后群组起来根据分数的降序排列，十分巧妙的解法：

解法3：

```

1 | SELECT s.Score, COUNT(DISTINCT t.Score) Rank
2 | FROM Scores s JOIN Scores t ON s.Score <= t.Score
3 | GROUP BY s.Id ORDER BY s.Score DESC;

```

下面这种解法跟上面三种的画风就不太一样了，这里用了两个变量，变量使用时其前面需要加@，这里的:=是赋值的意思，如果前面有Set关键字，则可以直接用=号来赋值，如果没有，则必须要使用:=来赋值，两个变量rank和pre，其中rank表示当前的排名，pre表示之前的分数，下面代码中的<>表示不等于，如果左右两边不相等，则返回true或1，若相等，则返回false或0。初始化rank为0，pre为-1，然后按降序排列分数，对于分数4来说，pre赋为4，和之前的pre值-1不同，所以rank要加1，那么分数4的rank就为1，下一个分数还是4，那么pre赋值为4和之前的4相同，所以rank要加0，所以这个分数4的rank也是1，以此类推就可以计算出所有分数的rank了。

解法4：

```

1 | SELECT Score,
2 | @rank := @rank + (@pre <> (@pre := Score)) Rank
3 | FROM Scores, (SELECT @rank := 0, @pre := -1) INIT
4 | ORDER BY Score DESC;

```

179. 最大组合数

Given a list of non negative integers, arrange them such that they form the largest number.

For example, given [3, 30, 34, 5, 9], the largest formed number is 9534330.

Note: The result may be very large, so you need to return a string instead of an integer.

Credits:

Special thanks to @ts for adding this problem and creating all test cases.

这道题给了我们一个数组，让我们将其拼接成最大的数，那么根据题目中给的例子来看，主要就是要给给定数组进行排序，但是排序方法不是普通的升序或者降序，因为9要排在最前面，而9既不是数组中最大的也不是最小的，所以我们要自定义排序方法。如果不参考网友的解法，我估计是无法想出来的。这种解法对于两个数字a和b来说，如果将其都转为字符串，如果ab>ba，则a排在前面，比如9和34，由于934>349，所以9排在前面，再比如说30和3，由于303<330，所以3排在30的前面。按照这种规则对原数组进行排序后，将每个数字转化为字符串再连接起来就是最终结果。代码如下：

```

1 | class Solution {
2 | public:
3 |     string largestNumber(vector<int>& nums) {
4 |         string res;
5 |         sort(nums.begin(), nums.end(), [] (int a, int b) {
6 |             return to_string(a) + to_string(b) > to_string(b) + to_string(a);
7 |         });
8 |         for (int i = 0; i < nums.size(); ++i) {
9 |             res += to_string(nums[i]);
10 |         }
11 |         return res[0] == '0' ? "0" : res;
12 |     }
13 | };

```

180. 连续的数字

Write a SQL query to find all numbers that appear at least three times consecutively.

Id	Num
1	1
2	1
3	1
4	2
5	1
6	2
7	2

For example, given the above Logs table, 1 is the only number that appears consecutively for at least three times.

这道题给了我们一个Logs表，让我们找Num列中连续出现相同数字三次的数字，那么由于需要找三次相同数字，所以我们需要建立三个表的实例，我们可以用l1分别和l2, l3内交，l1和l2的Id下一个位置比，l1和l3的下两个位置比，然后将Num都相同的数字返回即可：

解法1：

```
1 | SELECT DISTINCT l1.Num FROM Logs l1
2 | JOIN Logs l2 ON l1.Id = l2.Id - 1
3 | JOIN Logs l3 ON l1.Id = l3.Id - 2
4 | WHERE l1.Num = l2.Num AND l2.Num = l3.Num;
```

SQL

下面这种方法没用用到Join，而是直接在三个表的实例中查找，然后把四个条件限定上，就可以返回正确结果了：

解法2：

```
1 | SELECT DISTINCT l1.Num FROM Logs l1, Logs l2, Logs l3
2 | WHERE l1.Id = l2.Id - 1 AND l2.Id = l3.Id - 1
3 | AND l1.Num = l2.Num AND l2.Num = l3.Num;
```

SQL

再来看一种画风截然不同的方法，用到了变量count和pre，分别初始化为0和-1，然后需要注意的是用到了IF语句，MySQL里的IF语句和我们所熟知的其他语言的if不太一样，相当于我们所熟悉的三元操作符a?b:c，若a真返回b，否则返回c，具体可看这个帖子。那么我们先来看对于Num列的第一个数字1，pre由于初始化是-1，和当前Num不同，所以此时count赋1，此时给pre赋为1，然后Num列的第二个1进来，此时的pre和Num相同了，count自增1，到Num列的第三个1进来，count增加到了3，此时满足了where条件，t.n >= 3，所以1就被select出来了，以此类推遍历完整个Num就可以得到最终结果：

解法3：

```
1 | SELECT DISTINCT Num FROM (
2 |   SELECT Num, @count := IF(@pre = Num, @count + 1, 1) AS n, @pre := Num
3 |   FROM Logs, (SELECT @count := 0, @pre := -1) AS init
4 | ) AS t WHERE t.n >= 3;
```

SQL

181. 员工挣得比经理多

The Employee table holds all employees including their managers. Every employee has an Id, and there is also a column for the manager Id.

Id	Name	Salary	ManagerId
1	Joe	70000	3
2	Henry	80000	4
3	Sam	60000	NULL
4	Max	90000	NULL

Given the Employee table, write a SQL query that finds out employees who earn more than their managers. For the above table, Joe is the only employee who earns more than his manager.

Employee
Joe

这道题给我们了一个Employee表，里面有员工的薪水信息和其经理的信息，经理也属于员工，其经理Id为空，让我们找出薪水比其经理高的员工，那么就是一个很简单的比较问题了，我们可以生成两个实例对象进行内交通过ManagerId和Id，然后限制条件是一个Salary大于另一个即可：

解法1:

```
1 | SELECT e1.Name FROM Employee e1
2 | JOIN Employee e2 ON e1.ManagerId = e2.Id
3 | WHERE e1.Salary > e2.Salary;
```

SQL

我们也可以不用Join，直接把条件都写到where里也行：

解法2:

```
1 | SELECT e1.Name FROM Employee e1, Employee e2
2 | WHERE e1.ManagerId = e2.Id AND e1.Salary > e2.Salary;
```

SQL

182. 重复的邮箱

Write a SQL query to find all duplicate emails in a table named Person.

Id	Email
1	a@b.com
2	c@d.com
3	a@b.com

For example, your query should return the following for the above table:

Email
a@b.com

Note: All emails are in lowercase.

这道题让我们求重复的邮箱，那么最直接的方法就是用Group by...Having Count(*)...的固定搭配来做，代码如下：

解法1：

```
1 | SELECT Email FROM Person GROUP BY Email
2 | HAVING COUNT(*) > 1;
```

SQL

我们还可以用内交来做，用Email来内交两个表，然后返回Id不同的行，则说明两个不同的人使用了相同的邮箱：

解法2：

```
1 | SELECT DISTINCT p1.Email FROM Person p1
2 | JOIN Person p2 ON p1.Email = p2.Email
3 | WHERE p1.Id <> p2.Id;
```

SQL

183. 从未下单订购的顾客

Suppose that a website contains two tables, the Customers table and the Orders table. Write a SQL query to find all customers who never order anything.

Table: Customers.

Id	Name
1	Joe
2	Henry
3	Sam
4	Max

Table: Orders.

Id	CustomerId
1	3
2	1

Using the above tables as example, return the following:

Customers
Henry
Max

这道题让我们给了我们一个Customers表和一个Orders表，让我们找到从来没有下单的顾客，那么我们最直接的方法就是找没有在Orders表中出现的顾客Id就行了，用Not in关键字，如下所示：

解法1：

```
1 | SELECT Name AS Customers FROM Customers
2 | WHERE Id NOT IN (SELECT CustomerId FROM Orders);
```

SQL

或者我们也可以用左交来联合两个表，只要找出右边的CustomerId为Null的顾客就是木有下单的：

解法2：

```
1 | SELECT Name AS Customers FROM Customers
2 | LEFT JOIN Orders ON Customers.Id = Orders.CustomerId
3 | WHERE Orders.CustomerId IS NULL;
```

SQL

我们还可以用Not exists关键字来做，原理和Not in很像，参见代码如下：

解法3：

```
1 | SELECT Name AS Customers FROM Customers c
2 | WHERE NOT EXISTS (SELECT * FROM Orders o WHERE o.CustomerId = c.Id);
```

SQL

184. 系里最高薪水

The Employee table holds all employees. Every employee has an Id, a salary, and there is also a column for the department Id.

Id	Name	Salary	DepartmentId
1	Joe	70000	1
2	Henry	80000	2
3	Sam	60000	2
4	Max	90000	1

The Department table holds all departments of the company.

Id	Name
1	IT
2	Sales

Write a SQL query to find employees who have the highest salary in each of the departments. For the above tables, Max has the highest salary in the IT department and Henry has the highest salary in the Sales department.

Department	Employee	Salary
IT	Max	90000
Sales	Henry	80000

这道题让给了我们两张表，Employee表和Department表，让我们找系里面薪水最高的人的，实际上这题是Second Highest Salary和Combine Two Tables的结合题，我们既需要联合两表，又要找到最高薪水，那么我们首先让两个表内交起来，然后将结果表需要的列都标明，然后就是要找最高的薪水，我们用Max关键字来实现，参见代码如下：

解法1：

```
1 | SELECT d.Name AS Department, e1.Name AS Employee, e1.Salary FROM Employee e1
2 | JOIN Department d ON e1.DepartmentId = d.Id WHERE Salary IN
3 | (SELECT MAX(Salary) FROM Employee e2 WHERE e1.DepartmentId = e2.DepartmentId);
```

SQL

我们也可以不用Join关键字，直接用Where将两表连起来，然后找最高薪水的方法和上面相同：

解法2：

```
1 | SELECT d.Name AS Department, e.Name AS Employee, e.Salary FROM Employee e, Department d
2 | WHERE e.DepartmentId = d.Id AND e.Salary = (SELECT MAX(Salary) FROM Employee e2 WHERE
e2.DepartmentId = d.Id);
```

SQL

下面这种方法没用用到Max关键字，而是用了 \geq 符号，实现的效果跟Max关键字相同，参见代码如下：

解法3：

```

1 | SELECT d.Name AS Department, e.Name AS Employee, e.Salary FROM Employee e, Department d
2 | WHERE e.DepartmentId = d.Id AND e.Salary >= ALL (SELECT Salary FROM Employee e2 WHERE
   | e2.DepartmentId = d.Id);

```

SQL

185. 系里前三高薪水

The Employee table holds all employees. Every employee has an Id, and there is also a column for the department Id.

Id	Name	Salary	DepartmentId
1	Joe	70000	1
2	Henry	80000	2
3	Sam	60000	2
4	Max	90000	1
5	Janet	69000	1
6	Randy	85000	1

The Department table holds all departments of the company.

Id	Name
1	IT
2	Sales

Write a SQL query to find employees who earn the top three salaries in each of the department. For the above tables, your SQL query should return the following rows.

Department	Employee	Salary
IT	Max	90000
IT	Randy	85000
IT	Joe	70000
Sales	Henry	80000
Sales	Sam	60000

这道题是之前那道Department Highest Salary的拓展，难度标记为Hard，还是蛮有难度的一道题，综合了前面很多题的知识点，首先看使用Select Count(Distinct)的方法，我们内交Employee和Department两张表，然后我们找出比当前薪水高的最多只能有两个，那么前三高的都能被取出来了，参见代码如下：

解法1：

```

1 | SELECT d.Name AS Department, e.Name AS Employee, e.Salary FROM Employee e
2 | JOIN Department d on e.DepartmentId = d.Id
3 | WHERE (SELECT COUNT(DISTINCT Salary) FROM Employee WHERE Salary > e.Salary
4 | AND DepartmentId = d.Id) < 3 ORDER BY d.Name, e.Salary DESC;

```

SQL

下面这种方法将上面方法中的<3换成了IN (0, 1, 2)，是一样的效果：

解法2：

```

1 | SELECT d.Name AS Department, e.Name AS Employee, e.Salary FROM Employee e, Department d
2 | WHERE (SELECT COUNT(DISTINCT Salary) FROM Employee WHERE Salary > e.Salary
3 | AND DepartmentId = d.Id) IN (0, 1, 2) AND e.DepartmentId = d.Id ORDER BY d.Name, e.Salary
| DESC;

```

SQL

或者我们也可以使用Group by Having Count(Distinct ..) 关键字来做:

解法3:

```

1 | SELECT d.Name AS Department, e.Name AS Employee, e.Salary FROM
2 | (SELECT e1.Name, e1.Salary, e1.DepartmentId FROM Employee e1 JOIN Employee e2
3 | ON e1.DepartmentId = e2.DepartmentId AND e1.Salary <= e2.Salary GROUP BY e1.Id
4 | HAVING COUNT(DISTINCT e2.Salary) <= 3) e JOIN Department d ON e.DepartmentId = d.Id
5 | ORDER BY d.Name, e.Salary DESC;

```

SQL

下面这种方法略微复杂一些，用到了变量，跟Consecutive Numbers中的解法三使用的方法一样，目的是为了给每个人都按照薪水的高低增加一个rank，最后返回rank值小于等于3的项即可，参见代码如下：

解法4:

```

1 | SELECT d.Name AS Department, e.Name AS Employee, e.Salary FROM
2 | (SELECT Name, Salary, DepartmentId,
3 | @rank := IF(@pre_d = DepartmentId, @rank + (@pre_s <> Salary), 1) AS rank,
4 | @pre_d := DepartmentId, @pre_s := Salary
5 | FROM Employee, (SELECT @pre_d := -1, @pre_s := -1, @rank := 1) AS init
6 | ORDER BY DepartmentId, Salary DESC) e JOIN Department d ON e.DepartmentId = d.Id
7 | WHERE e.rank <= 3 ORDER BY d.Name, e.Salary DESC;
8 |

```

SQL

186. 翻转字符串中的单词之二

Given an input string, reverse the string word by word. A word is defined as a sequence of non-space characters.

The input string does not contain leading or trailing spaces and the words are always separated by a single space.

For example,

Given s = "the sky is blue",

return "blue is sky the".

Could you do it in-place without allocating extra space?

这道题让我们翻转一个字符串中的单词，跟之前那题Reverse Words in a String没有区别，由于之前那道题我们就是用in-place的方法做的，而这道题反而更简化了题目，因为不考虑首尾空格了和单词之间的多空格了，方法还是很简单，先把每个单词翻转一遍，再把整个字符串翻转一遍，或者也可以调换个顺序，先翻转整个字符串，再翻转每个单词，参见代码如下：

```

1 class Solution {
2 public:
3     void reverseWords(string &s) {
4         int left = 0;
5         for (int i = 0; i <= s.size(); ++i) {
6             if (i == s.size() || s[i] == ' ') {
7                 reverse(s, left, i - 1);
8                 left = i + 1;
9             }
10        }
11        reverse(s, 0, s.size() - 1);
12    }
13    void reverse(string &s, int left, int right) {
14        while (left < right) {
15            char t = s[left];
16            s[left] = s[right];
17            s[right] = t;
18            ++left; --right;
19        }
20    }
21 };

```

187. 求重复的DNA序列

All DNA is composed of a series of nucleotides abbreviated as A, C, G, and T, for example: "ACGAATTCCG". When studying DNA, it is sometimes useful to identify repeated sequences within the DNA.

Write a function to find all the 10-letter-long sequences (substrings) that occur more than once in a DNA molecule.

For example,

Given s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT",

Return:

["AAAAACCCCC", "CCCCAAAAAA"].

看到这道题想到这应该属于CS的一个重要分支生物信息Bioinformatics研究的内容，研究DNA序列特征的重要意义自然不用多说，但是对于我们广大码农来说，还是专注于算法吧，此题还是用位操作Bit Manipulation来求解，计算机由于其二进制存储的特点可以很巧妙的解决一些问题，像之前的Single Number 单独的数字和Single Number II 单独的数字之二都是很巧妙利用位操作来求解。此题由于构成输入字符串的字符只有四种，分别是A, C, G, T，下面我们来看下它们的ASCII码用二进制来表示：

A: 0100 0001 C: 0100 0011 G: 0100 0111 T: 0101 0100

由于我们的目的是利用位来区分字符，当然是越少位越好，通过观察发现，每个字符的后三位都不相同，故而我们可以用末尾三位来区分这四个字符。而题目要求是10个字符长度的串，每个字符用三位来区分，10个字符需要30位，在32位机上也OK。为了提取出后30位，我们还需要用个mask，取值为0x7fffffff，用此mask可取出后27位，再向左平移三位即可。算法的思想是，当取出第十个字符时，将其存在哈希表里，和该字符串出现频率映射，之后每向左移三位替换一个字符，查找新字符串在哈希表里出现次数，如果之前刚好出现过一次，则将当前字符串存入返回值的数组并将其出现次数加一，如果从未出现过，则将其映射到1。为了能更清楚的阐述整个过程，我们用题目中给的例子来分析整个过程：

首先我们取出前九个字符AAAAACCCCC，根据上面的分析，我们用三位来表示一个字符，所以这九个字符可以用二进制表示为001001001001011011011011，然后我们继续遍历字符串，下一个进来的是C，则当前字符为AAAAACCCCC，二进制表示为001001001001011011011011011011，然后我们将其存入哈希表中，用二进制的好处是可以用一个int变量来表示任意十个字符序列，比起直接存入字符串大大的节省了内存空间，然后再读入下一个字符C，则此时字符串为AAAACCCCCA，我们还是存入其二进制的表示形式，以此类推，当某个序列之前已经出现过了，我们将其存入结果res中即可，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     vector<string> findRepeatedDnaSequences(string s) {
4         vector<string> res;
5         if (s.size() <= 10) return res;
6         int mask = 0x7fffffff;
7         unordered_map<int, int> m;
8         int cur = 0, i = 0;
9         while (i < 9) {
10             cur = (cur << 3) | (s[i++] & 7);
11         }
12         while (i < s.size()) {
13             cur = ((cur & mask) << 3) | (s[i++] & 7);
14             if (m.find(cur) != m.end()) {
15                 if (m[cur] == 1) res.push_back(s.substr(i - 10, 10));
16                 ++m[cur];
17             } else {
18                 m[cur] = 1;
19             }
20         }
21         return res;
22     }
23 }
```

CPP

上面的方法可以写的更简洁一些，这里我们可以用set来代替哈希表，只要当前的数已经在哈希表中存在了，我们就将其加入res中，这里我们res也定义成set，这样就可以利用set的不能有重复项的特点，从而得到正确的答案，最后我们将set转为vector即可，参见代码如下：

解法2:

```

1 class Solution {
2 public:
3     vector<string> findRepeatedDnaSequences(string s) {
4         unordered_set<string> res;
5         unordered_set<int> st;
6         int cur = 0, i = 0;
7         while (i < 9) cur = cur << 3 | (s[i++] & 7);
8         while (i < s.size()) {
9             cur = ((cur & 0x7fffffff) << 3) | (s[i++] & 7);
10            if (st.count(cur)) res.insert(s.substr(i - 10, 10));
11            else st.insert(cur);
12        }
13        return vector<string>(res.begin(), res.end());
14    }
15 };

```

上面的方法都是用三位来表示一个字符，这里我们可以用两位来表示一个字符，00表示A，01表示C，10表示G，11表示T，那么我们总共需要20位就可以表示十个字符流，其余的思路跟上面的方法完全相同，注意这里的mask只需要表示18位，所以变成了0x3ffff，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     vector<string> findRepeatedDnaSequences(string s) {
4         unordered_set<string> res;
5         unordered_set<int> st;
6         unordered_map<int, int> m{{'A', 0}, {'C', 1}, {'G', 2}, {'T', 3}};
7         int cur = 0, i = 0;
8         while (i < 9) cur = cur << 2 | m[s[i++]];
9         while (i < s.size()) {
10            cur = ((cur & 0x3ffff) << 2) | (m[s[i++]]);
11            if (st.count(cur)) res.insert(s.substr(i - 10, 10));
12            else st.insert(cur);
13        }
14        return vector<string>(res.begin(), res.end());
15    }
16 };

```

如果我们不需要考虑节省内存空间，那我们可以直接将10个字符组成字符串存入set中，那么也就不需要mask啥的了，但是思路还是跟上面的方法相同：

解法4：

```

1 class Solution {
2 public:
3     vector<string> findRepeatedDnaSequences(string s) {
4         set<string> res, st;
5         for (int i = 0; i + 9 < s.size(); ++i) {
6             string t = s.substr(i, 10);
7             if (st.count(t)) res.insert(t);
8             else st.insert(t);
9         }
10    return vector<string>{res.begin(), res.end()};
11 }
12 };

```

188. 买卖股票的最佳时间之四

Say you have an array for which the i th element is the price of a given stock on day i .

Design an algorithm to find the maximum profit. You may complete at most k transactions.

Note:

You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

Credits:

Special thanks to @Freezen for adding this problem and creating all test cases.

这道题实际上是之前那道 Best Time to Buy and Sell Stock III 买股票的最佳时间之三的一般情况的推广，还是需要用动态规划Dynamic programming来解决，具体思路如下：

这里我们需要两个递推公式来分别更新两个变量local和global，参见网友Code Ganker的博客，我们其实可以求至少 k 次交易的最大利润。我们定义local[i][j]为在到达第*i*天时最多可进行j次交易并且最后一次交易在最后一天卖出的最大利润，此为局部最优。然后我们定义global[i][j]为在到达第*i*天时最多可进行j次交易的最大利润，此为全局最优。它们的递推式为：

```

local[i][j] = max(global[i - 1][j - 1] + max(diff, 0), local[i - 1][j] + diff)

global[i][j] = max(local[i][j], global[i - 1][j]),

```

其中局部最优值是比较前一天并少交易一次的全局最优加上大于0的差值，和前一天的局部最优加上差值后相比，两者之中取较大值，而全局最优比较局部最优和前一天的全局最优。

但这道题还有个坑，就是如果 k 的值远大于prices的天数，比如 k 是好几百万，而prices的天数就为若干天的话，上面的DP解法就非常的没有效率，应该直接用Best Time to Buy and Sell Stock II 买股票的最佳时间之二的方法来求解，所以实际上这道题是之前的二和三的综合体，代码如下：

```

1 class Solution {
2 public:
3     int maxProfit(int k, vector<int> &prices) {
4         if (prices.empty()) return 0;
5         if (k >= prices.size()) return solveMaxProfit(prices);
6         int g[k + 1] = {0};
7         int l[k + 1] = {0};
8         for (int i = 0; i < prices.size() - 1; ++i) {
9             int diff = prices[i + 1] - prices[i];
10            for (int j = k; j >= 1; --j) {
11                l[j] = max(g[j - 1] + max(diff, 0), l[j] + diff);
12                g[j] = max(g[j], l[j]);
13            }
14        }
15        return g[k];
16    }
17    int solveMaxProfit(vector<int> &prices) {
18        int res = 0;
19        for (int i = 1; i < prices.size(); ++i) {
20            if (prices[i] - prices[i - 1] > 0) {
21                res += prices[i] - prices[i - 1];
22            }
23        }
24        return res;
25    }
26 };

```

189. 旋转数组

Rotate an array of n elements to the right by k steps.

For example, with n = 7 and k = 3, the array [1,2,3,4,5,6,7] is rotated to [5,6,7,1,2,3,4].

Note:

Try to come up as many solutions as you can, there are at least 3 different ways to solve this problem.

[show hint]

Hint:

Could you do it in-place with O(1) extra space?

Credits:

Special thanks to @Freezen for adding this problem and creating all test cases.

新题抢先刷，这道题标为Easy，应该不是很难，我们先来看一种O(n)的空间复杂度的方法，我们复制一个和nums一样的数组，然后利用映射关系 $i \rightarrow (i+k)\%n$ 来交换数字。代码如下：

解法1：

```

1 class Solution {
2 public:
3     void rotate(vector<int>& nums, int k) {
4         vector<int> t = nums;
5         for (int i = 0; i < nums.size(); ++i) {
6             nums[(i + k) % nums.size()] = t[i];
7         }
8     }
9 };

```

由于提示中要求我们空间复杂度为 $O(1)$ ，所以我们不能用辅助数组，上面的思想还是可以使用的，但是写法就复杂的多，而且需要用到很多的辅助变量，我们用题目中的例子来展示下面这种算法的`nums`的变化过程：

```

1 2 3 4 5 6 7
1 2 3 1 5 6 7
1 2 3 1 5 6 4
1 2 7 1 5 6 4
1 2 7 1 5 3 4
1 6 7 1 5 3 4
1 6 7 1 2 3 4
5 6 7 1 2 3 4

```

解法2：

```

1 class Solution {
2 public:
3     void rotate(vector<int>& nums, int k) {
4         if (nums.empty() || (k %= nums.size()) == 0) return;
5         int n = nums.size(), start = 0, i = 0, cur = nums[i], cnt = 0;
6         while (cnt++ < n) {
7             i = (i + k) % n;
8             int t = nums[i];
9             nums[i] = cur;
10            if (i == start) {
11                ++start; ++i;
12                cur = nums[i];
13            } else {
14                cur = t;
15            }
16        }
17    }
18 };

```

根据热心网友waruzhi的留言，这道题其实还有种类似翻转字符的方法，思路是先把前 $n-k$ 个数字翻转一下，再把后 k 个数字翻转一下，最后再把整个数组翻转一下：

```

1 2 3 4 5 6 7
4 3 2 1 5 6 7
4 3 2 1 7 6 5
5 6 7 1 2 3 4

```

解法3：

```

1 class Solution {
2 public:
3     void rotate(vector<int>& nums, int k) {
4         if (nums.empty() || (k %= nums.size()) == 0) return;
5         int n = nums.size();
6         reverse(nums.begin(), nums.begin() + n - k);
7         reverse(nums.begin() + n - k, nums.end());
8         reverse(nums.begin(), nums.end());
9     }
10 };

```

由于旋转数组的操作也可以看做从数组的末尾取k个数组放入数组的开头，所以我们用STL的push_back和erase可以很容易的实现这些操作。

解法4：

```

1 class Solution {
2 public:
3     void rotate(vector<int>& nums, int k) {
4         if (nums.empty() || (k %= nums.size()) == 0) return;
5         int n = nums.size();
6         for (int i = 0; i < n - k; ++i) {
7             nums.push_back(nums[0]);
8             nums.erase(nums.begin());
9         }
10    }
11 };

```

下面这种方法其实还蛮独特的，通过不停的交换某两个数字的位置来实现旋转，根据网上这个帖子的思路改写而来，数组改变过程如下：

```

1 2 3 4 5 6 7
5 2 3 4 1 6 7
5 6 3 4 1 2 7
5 6 7 4 1 2 3
5 6 7 1 4 2 3
5 6 7 1 2 4 3
5 6 7 1 2 3 4

```

解法5：

```

1 class Solution {
2 public:
3     void rotate(vector<int>& nums, int k) {
4         if (nums.empty()) return;
5         int n = nums.size(), start = 0;
6         while (n && (k %= n)) {
7             for (int i = 0; i < k; ++i) {
8                 swap(nums[i + start], nums[n - k + i + start]);
9             }
10            n -= k;
11            start += k;
12        }
13    }
14 };

```

190. 翻转位

Reverse bits of a given 32 bits unsigned integer.

For example, given input 43261596 (represented in binary as 0000001010010100000111010011100), return 964176192 (represented in binary as 00111001011110000010100101000000).

Follow up:

If this function is called many times, how would you optimize it?

Related problem: Reverse Integer

Credits:

Special thanks to @ts for adding this problem and creating all test cases.

这道题又是在考察位操作Bit Operation，LeetCode中有关位操作的题也有不少，比如 Repeated DNA Sequences, Single Number, Single Number II，和 Grey Code 等等。跟上面那些题比起来，这道题简直不能再简单了。那么对于这道题，我们只需要把要翻转的数从右向左一位位的取出来，如果取出来的是1，我们将结果res左移一位并且加上1；如果取出来的是0，我们将结果res左移一位，然后将n右移一位即可，代码如下：

解法1：

```

1 class Solution {
2 public:
3     uint32_t reverseBits(uint32_t n) {
4         uint32_t res = 0;
5         for (int i = 0; i < 32; ++i) {
6             if (n & 1 == 1) {
7                 res = (res << 1) + 1;
8             } else {
9                 res = res << 1;
10            }
11            n = n >> 1;
12        }
13        return res;
14    }
15 };

```

我们可以简化上面的代码，去掉if...else...结构，可以结果res左移一位，然后再判断n的最低位是否为1，是的话那么结果res加上1，然后将n右移一位即可，代码如下：

解法2：

```
1 class Solution {
2 public:
3     uint32_t reverseBits(uint32_t n) {
4         uint32_t res = 0;
5         for (int i = 0; i < 32; ++i) {
6             res <<= 1;
7             if ((n & 1) == 1) ++res;
8             n >>= 1;
9         }
10        return res;
11    }
12};
```

CPP

我们继续简化上面的解法，将if判断句直接揉进去，通过‘或’上一个n的最低位即可，用n‘与’1提取最低位，然后将n右移一位即可，代码如下：

解法3：

```
1 class Solution {
2 public:
3     uint32_t reverseBits(uint32_t n) {
4         uint32_t res = 0;
5         for (int i = 0; i < 32; ++i) {
6             res = (res << 1) | (n & 1);
7             n >>= 1;
8         }
9         return res;
10    }
11};
```

CPP

博主还能进一步简化，这里我们不更新n的值，而是直接将n右移i位，然后通过‘与’1来提取出该位，加到左移一位后的结果res中即可，参加代码如下：

解法4：

```
1 class Solution {
2 public:
3     uint32_t reverseBits(uint32_t n) {
4         uint32_t res = 0;
5         for (int i = 0; i < 32; ++i) {
6             res = (res << 1) + (n >> i & 1);
7         }
8         return res;
9     }
10};
```

CPP

我们也可以换一种角度来做，首先将n右移i位，然后通过‘与’1来提取出该位，然后将其左移(32 - i)位，然后‘或’上结果res，就是其翻转后应该在的位置，参见代码如下：

解法5：

```

1 class Solution {
2 public:
3     uint32_t reverseBits(uint32_t n) {
4         uint32_t res = 0;
5         for (int i = 0; i < 32; ++i) {
6             res |= ((n >> i) & 1) << (31 - i);
7         }
8         return res;
9     }
10 };

```

得票最高的大神的解题方案参考：

for 8 bit binary number abcdefgh, the process is as follow:
 abcdefgh -> efghabcd -> ghefcda -> hgfedcba

解法6：

```

1 class Solution {
2 public:
3     uint32_t reverseBits(uint32_t n) {
4         n = (n >> 16) | (n << 16);
5         n = ((n & 0xff00ff00) >> 8) | ((n & 0x00ff00ff) << 8);
6         n = ((n & 0xf0f0f0f0) >> 4) | ((n & 0x0f0f0f0f) << 4);
7         n = ((n & 0xcccccccc) >> 2) | ((n & 0x33333333) << 2);
8         n = ((n & 0aaaaaaaa) >> 1) | ((n & 0x55555555) << 1);
9         return n;
10    }
11 };

```

191. 位1的个数

Write a function that takes an unsigned integer and returns the number of '1' bits it has (also known as the Hamming weight).

For example, the 32-bit integer '11' has binary representation 00000000000000000000000000001011, so the function should return 3.

很简单的一道位操作Bit Manipulation的题，最近新出的三道题都没有啥难度啊，这样会误导新人的，做了这三道得出个LeetCode没啥难度的结论，其实里面好题真的不少，难题也很多，经典题也多，反正就是赞赞赞，32个赞。

```

1 class Solution {
2 public:
3     int hammingWeight(uint32_t n) {
4         int res = 0;
5         for (int i = 0; i < 32; ++i) {
6             res += (n & 1);
7             n = n >> 1;
8         }
9         return res;
10    }
11 };

```

192. 单词频率

Write a bash script to calculate the frequency of each word in a text file words.txt.

For simplicity sake, you may assume:

words.txt contains only lowercase characters and space ' ' characters.

Each word must consist of lowercase characters only.

Words are separated by one or more whitespace characters.

For example, assume that words.txt has the following content:

the day is sunny the the

the sunny is is

Your script should output the following, sorted by descending frequency:

the 4

is 3

sunny 2

day 1

Note:

Don't worry about handling ties, it is guaranteed that each word's frequency count is unique.

[show hint]

Hint:

Could you write it in one-line using Unix pipes?

这道题给了我们一个文本文件，让我们统计里面单词出现的个数，提示中让我们用管道Pipes来做，在之前那道Tenth Line中，我们使用过管道命令，管道命令的讲解请参见这个帖子。提示中让我们用管道连接各种命令，然后一行搞定，那么我们先来看第一种解法，乍一看啥都不明白，咋办？没关系，容我慢慢来讲解。首先用的关键字是grep命令，该命令一种强大的文本搜索工具，它能使用正则表达式搜索文本，并把匹配的行打印出来，详解请参见这个帖子。后面紧跟的-oE '[a-z]+'参数表示原文本内容变成一个单词一行的存储方式，于是此时文本的内容就变成了：

```
the
day
is
sunny
the
the
the
sunny
is
```

下面的sort命令就是用来排序的，参见这个帖子。排完序的结果为：

```
day
is
is
is
sunny
sunny
the
the
the
the
```

后面的uniq命令是表示去除重复行命令(参见这个帖子)，后面的参数-c表示在每行前加上表示相应行目出现次数的前缀编号，得到结果如下：

```
1 day
3 is
2 sunny
4 the
```

然后我们再sort一下，后面的参数-nr表示按数值进行降序排列，得到结果：

```
4 the
3 is
2 sunny
1 day
```

而最后的awk命令就是将结果输出，两列颠倒位置即可：

解法1：

```
1 | grep -oE '[a-z]+' words.txt | sort | uniq -c | sort -nr | awk '{print $2" "$1}' SQL
```

下面这种方法用的关键字是tr命令，该命令主要用来进行字符替换或者大小写替换，详解请参见这个帖子。后面紧跟的-s参数表示如果发现连续的字符，就把它们缩减为1个，而后面的' '和'\n'就是空格和回车，意思是把所有的空格都换成回车，那么第一段命令tr -s ' ' '\n' < words.txt 就好理解了，将单词之间的空格都换成回车，跟上面的第一段实现的作用相同，后面就完全一样了，参见上面的讲解：

解法2：

```
1 | tr -s ' ' '\n' < words.txt | sort | uniq -c | sort -nr | awk '{print $2, $1}'
```

SQL

下面这种方法就没有用管道命令进行一行写法了，而是使用了强大的文本分析工具awk进行写类C的代码来实现，这种写法在之前的那道Transpose File已经讲解过了，这里就不多说了，最后要注意的是sort命令的参数-nr -k 2表示按第二列的降序数值排列：

解法3：

```
1 | awk '{
2     for (i = 1; i <= NF; ++i) ++s[$i];
3 } END {
4     for (i in s) print i, s[i];
5 }' words.txt | sort -nr -k 2
```

SQL

193. 验证电话号码

Given a text file file.txt that contains list of phone numbers (one per line), write a one liner bash script to print all valid phone numbers.

You may assume that a valid phone number must appear in one of the following two formats: (xxx) xxx-xxxx or xxx-xxx-xxxx. (x means a digit)

You may also assume each line in the text file must not contain leading or trailing white spaces.

For example, assume that file.txt has the following content:

```
987-123-4567
123 456 7890
(123) 456-7890
```

Your script should output the following valid phone numbers:

```
987-123-4567
(123) 456-7890
```

这道题让我们验证数字串是否为正确的电话号码的格式，而且规定了正确的格式只有两种(xxx) xxx-xxxx or xxx-xxx-xxxx，那么我们可以看出来区别就是在前几个字符，而后八个字符都相同。这题有多种解法，我们首先来看使用awk命令的解法，关于awk的介绍可以参见这个帖子。这道题是难点是如何写匹配的正则表达式，关于Bash脚本的正则表达式讲解请参见这个帖子。那么首先来看'.../'表示中间的是要匹配的正则表达式，然后脱字符^匹配一行的开头，美元符\$在正则表达式中匹配行尾，然后再看中间的部分，[0-9]{3}表示匹配三个数字，圆括号括起一组正则表达式. 它和"|"操作符或在用expr进行子字符串提取(substring extraction)一起使用很有用。那么([0-9]{3}-|[0-9]{3})就可以理解了，它匹配了xxx-和(xxx)这两种形式的字符串，然后后面的就好理解了，匹配xxx-xxxx这样的字符串，参见代码如下：

解法1：

```
1 | awk '/^(([0-9]{3}-|([0-9]{3})) [0-9]{3}-[0-9]{4}$/' file.txt
```

SQL

下面来看使用sed命令的解法，关于sed的讲解可以参见这个帖子。那么我们先来看后面的两个参数，-n表示关闭默认输出，默认将自动打印所有行，这样就不会打印出不符合要求的数字串了。-r表示支持扩展正则+?0 {} |。后面的正则表达式和上面都相同，就是后面多了一个p，在用sed时，p和-n合用，表示打印某一行，这样才能把符合要求的行打印出来：

解法2：

```
1 | sed -n -r '/^([0-9]{3}-|([0-9]{3}\ ))[0-9]{3}-[0-9]{4}$/' file.txt
```

SQL

再来看使用grep命令的做法，关于grep的讲解可以参见这个帖子。我没有查到那个-P参数的用法，有没有大神来点拨一下，后面的正则表达式思路跟上面的相同，只不过用d{3}来表示[0-9]{3}，道理都一样，参见代码如下：

解法3：

```
1 | grep -P '^(\d{3}-|\d{3}\ )\d{3}-\d{4}$' file.txt
```

SQL

194. 转置文件

Given a text file file.txt, transpose its content.

You may assume that each row has the same number of columns and each field is separated by the ' ' character.

For example, if file.txt has the following content:

```
name age
alice 21
ryan 30
```

Output the following:

```
name alice ryan
age 21 30
```

这道题让我们转置一个文件，其实感觉就是把文本内容当做一个矩阵，每个单词空格隔开看做是矩阵中的一个元素，然后将转置后的内容打印出来。那么我们先来看使用awk关键字的做法，关于awk的介绍可以参见这个帖子。其中NF表示当前记录中的字段个数，就是有多少列，NR表示已经读出的记录数，就是行号，从1开始。那么在这里NF是2，因为文本只有两列，这里面这个for循环还跟我们通常所熟悉for循环不太一样，通常我们认为i只能是1和2，然后循环就结束了，而这里的i实际上遍历的数字为1,2,1,2,1,2，我们可能看到实际上循环了3遍1和2，而行数正好是3，可能人家就是这个机制吧。知道了上面这些，那么下面的代码就不难理解了，遍历过程如下：

```
i = 1, s = [name]
i = 2, s = [name; age]
i = 1, s = [name alice; age]
i = 2, s = [name alice; age 21]
i = 1, s = [name alice ryan; age 21]
i = 2, s = [name alice ryan; age 21 30]
```

然后我们再将s中的各行打印出来即可，参见代码如下：

解法1：

```

1 awk '{
2     for (i = 1; i <= NF; ++i) {
3         if (NR == 1) s[i] = $i;
4         else s[i] = s[i] " " $i;
5     }
6 } END {
7     for (i = 1; s[i] != ""; ++i) {
8         print s[i];
9     }
10}' file.txt

```

下面这种方法和上面的思路完全一样，但是代码风格不一样，上面是C语言风格，而这个完全就是Bash脚本的风格了，我们用read关键字，我们可以查看read的用法read: usage: read [-ers] [-u fd] [-t timeout] [-p prompt] [-a array] [-n nchars] [-d delim] [name ...]。那么我们知道-a表示数组，将读出的每行内容存入数组line中，那么下一行for中的一堆特殊字符肯定让你头晕眼花，关于shell中的特殊变量可以参见这个帖子，其实我也不能算特别理解下面的代码，大概觉得跟上面的思路一样，求大神来具体给讲解下哈：

解法2:

```

1 while read -a line; do
2     for ((i = 0; i < "${#line[@]}"; ++i)); do
3         a[$i]="${a[$i]} ${line[$i]}"
4     done
5 done < file.txt
6 for ((i = 0; i < ${#a[@]}; ++i)); do
7     echo ${a[i]}
8 done

```

195. 第十行

How would you print just the 10th line of a file?

For example, assume that file.txt has the following content:

```

Line 1
Line 2
Line 3
Line 4
Line 5
Line 6
Line 7
Line 8
Line 9
Line 10

```

Your script should output the tenth line, which is:

```

Line 10
[show hint]

```

Hint:

1. If the file contains less than 10 lines, what should you output?
2. There's at least three different solutions. Try to explore all possibilities.

这道题让我们用Bash脚本来打印一个txt文件的第十行，可以用很多种方法来实现，我们先来看一种是用awk来实现的方法，awk是强大的文本分析工具，具有流控制、数学运算、进程控制、内置的变量和函数、循环和判断的功能，具体可以参见这个帖子。其中NR表示行数，\$0表示当前记录，所以我们可以用if来判断行数为第十行时，将内容打印出来即可：

解法1：

```
1 | awk '{if(NR == 10) print $0}' file.txt
```

SQL

我们也可以用更简洁的写法来打印出第十行如下：

解法2：

```
1 | awk 'NR == 10' file.txt
```

SQL

我们也可以使用流编辑工具sed来做，关于sed的讲解可以参见这个帖子。-n默认表示打印所有行，p限定了具体打印的行数：

解法3：

```
1 | sed -n 10p file.txt
```

SQL

我们也可以使用tail和head关键字来打印，关于tail和head的用法详解请参见这个帖子。其中head表示从头开始打印，tail表示从结尾开始打印，-你表示根据文件行数进行打印，一些区别与联系请见下列例子：

tail -n 3 file.txt：打印file文件的最后三行内容

tail -n +3 file.txt：从file文件第三行开始打印所有内容

head -n 3 file.txt：打印file文件的前三行

head -n -3 file.txt：打印file文件除了最后三行的所有内容

至于竖杠|为管道命令，讲解参见这个帖子，用法：command 1 | command 2 他的功能是把第一个命令command1执行的结果作为command 2的输入传给command 2。了解了这些知识，那么下面一行命令就很好理解了，首先输入file文件从第十行开始的所有内容，然后将输出内容的第一行打印出来即为第十行：

解法4：

```
1 | tail -n +10 file.txt | head -n 1
```

SQL

下面这种方法跟上面刚好相反，先输出file文件的前十行，然后从输出的第十行开始打印，那么也能正好打印第十行的内容：

解法5：

```
1 | head -n 10 file.txt | tail -n +10
```

SQL

196. 删除重复邮箱

Write a SQL query to delete all duplicate email entries in a table named Person, keeping only unique emails based on its smallest Id.

```
+----+-----+
| Id | Email      |
+----+-----+
| 1  | john@example.com |
| 2  | bob@example.com |
| 3  | john@example.com |
+----+-----+
```

Id is the primary key column for this table.

For example, after running your query, the above Person table should have the following rows:

```
+----+-----+
| Id | Email      |
+----+-----+
| 1  | john@example.com |
| 2  | bob@example.com |
+----+-----+
```

这道题让我们删除重复邮箱，那我们可以首先找出所有不重复的邮箱，然后取个反就是重复的邮箱，都删掉即可，那么我们如何找出所有不重复的邮箱呢，我们可以按照邮箱群组起来，然后用Min关键字挑出较小的，然后取补集删除即可：

解法1:

```
1 | DELETE FROM Person WHERE Id NOT IN
2 | (SELECT Id FROM (SELECT MIN(Id) Id FROM Person GROUP BY Email) p);
```

SQL

我们也可以使用内交让两个表以邮箱关联起来，然后把相同邮箱且Id大的删除掉，参见代码如下：

解法2:

```
1 | DELETE p2 FROM Person p1 JOIN Person p2
2 | ON p2.Email = p1.Email WHERE p2.Id > p1.Id;
```

SQL

我们也可以不用Join，而直接用where将两表关联起来也行：

解法3:

```
1 | DELETE p2 FROM Person p1, Person p2
2 | WHERE p1.Email = p2.Email AND p2.Id > p1.Id;
```

SQL

197. 上升温度

Given a Weather table, write a SQL query to find all dates' Ids with higher temperature compared to its previous (yesterday's) dates.

Id(INT)	Date(DATE)	Temperature(INT)
1	2015-01-01	10
2	2015-01-02	25
3	2015-01-03	20
4	2015-01-04	30

For example, return the following Ids for the above Weather table:

Id
2
4

这道题给了我们一个Weather表，让我们找出比前一天温度高的Id，由于Id的排列未必是按顺序的，所以我们要找前一天就得根据日期来找，我们可以使用MySQL的函数DATEDIFF来计算两个日期的差值，我们的限制条件是温度高且日期差1，参见代码如下：

解法1：

```
1 | SELECT w1.Id FROM Weather w1, Weather w2
2 | WHERE w1.Temperature > w2.Temperature AND DATEDIFF(w1.Date, w2.Date) = 1;
```

SQL

下面这种解法我们使用了MySQL的TO_DAYS函数，用来将日期换算成天数，其余跟上面相同：

解法2：

```
1 | SELECT w1.Id FROM Weather w1, Weather w2
2 | WHERE w1.Temperature > w2.Temperature AND TO_DAYS(w1.Date) = TO_DAYS(w2.Date) + 1;
```

SQL

我们也可以使用Subdate函数，来实现日期减1，参见代码如下：

解法3：

```
1 | SELECT w1.Id FROM Weather w1, Weather w2
2 | WHERE w1.Temperature > w2.Temperature AND SUBDATE(w1.Date, 1) = w2.Date;
```

SQL

最后来一种完全不一样的解法，使用了两个变量pre_t和pre_d分别表示上一个温度和上一个日期，然后当前温度要大于上一温度，且日期差为1，满足上述两条件的话选出来为Id，否则为NULL，然后更新pre_t和pre_d为当前的值，最后选出的Id不为空即可：

解法4：

```

1 | SELECT Id FROM (
2 |   SELECT CASE WHEN Temperature > @pre_t AND DATEDIFF(Date, @pre_d) = 1 THEN Id ELSE NULL END
3 |   AS Id,
4 |   @pre_t := Temperature, @pre_d := Date
5 |   FROM Weather, (SELECT @pre_t := NULL, @pre_d := NULL) AS init ORDER BY Date ASC
) id WHERE Id IS NOT NULL;

```

198. 打家劫舍

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security system connected and it will automatically contact the police if two adjacent houses were broken into on the same night.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight without alerting the police.

Credits:

Special thanks to @ifanchu for adding this problem and creating all test cases. Also thanks to @ts for adding additional test cases.

这道题的本质相当于在一列数组中取出一个或多个不相邻数，使其和最大。那么我们对于这类求极值的问题首先考虑动态规划 Dynamic Programming 来解，我们维护一个一位数组 dp，其中 dp[i] 表示到 i 位置时不相邻数能形成的最大和，那么状态转移方程怎么写呢，我们先拿一个简单的例子来分析一下，比如说 nums 为 {3, 2, 1, 5}，那么我们来看我们的 dp 数组应该是什么样的，首先 dp[0]=3 没啥疑问，再看 dp[1] 是多少呢，由于 3 比 2 大，所以我们抢第一个房子的 3，当前房子的 2 不抢，所以 dp[1]=3，那么再来看 dp[2]，由于不能抢相邻的，所以我们可以用再前面的一个的 dp 值加上当前的房间值，和当前房间的前面一个 dp 值比较，取较大值当做当前 dp 值，所以我们可以得到状态转移方程 $dp[i] = \max(num[i] + dp[i - 2], dp[i - 1])$ ，由此看出我们需要初始化 dp[0] 和 dp[1]，其中 dp[0] 即为 num[0]，dp[1] 此时应该为 $\max(num[0], num[1])$ ，代码如下：

解法1：

```

1 | class Solution {
2 | public:
3 |     int rob(vector<int> &num) {
4 |         if (num.size() <= 1) return num.empty() ? 0 : num[0];
5 |         vector<int> dp = {num[0], max(num[0], num[1])};
6 |         for (int i = 2; i < num.size(); ++i) {
7 |             dp.push_back(max(num[i] + dp[i - 2], dp[i - 1]));
8 |         }
9 |         return dp.back();
10 |     }
11 | };

```

还有一种解法，核心思想还是用 DP，分别维护两个变量 a 和 b，然后按奇偶分别来更新 a 和 b，这样就可以保证组成最大和的数字不相邻，代码如下：

解法2：

```

1 class Solution {
2 public:
3     int rob(vector<int> &nums) {
4         int a = 0, b = 0;
5         for (int i = 0; i < nums.size(); ++i) {
6             if (i % 2 == 0) {
7                 a = max(a + nums[i], b);
8             } else {
9                 b = max(a, b + nums[i]);
10            }
11        }
12        return max(a, b);
13    }
14 };

```

上述方法还可以进一步简洁如下：

解法3：

```

1 class Solution {
2 public:
3     int rob(vector<int> &nums) {
4         int a = 0, b = 0;
5         for (int i = 0; i < nums.size(); ++i) {
6             int m = a, n = b;
7             a = n + nums[i];
8             b = max(m, n);
9         }
10        return max(a, b);
11    }
12 };

```

199. 二叉树的右侧视图

Given a binary tree, imagine yourself standing on the right side of it, return the values of the nodes you can see ordered from top to bottom.

For example:

Given the following binary tree,

```

1           <---
 /   \
2     3       <---
 \   \
 5   4       <---

```

You should return [1, 3, 4].

Credits:

Special thanks to @amrsaqr for adding this problem and creating all test cases.

这道题要求我们打印出二叉树每一行最右边的一个数字，实际上是求二叉树层序遍历的一种变形，我们只需要保存每一层最右边的数字即可，可以参考我之前的博客 Binary Tree Level Order Traversal 二叉树层序遍历，这道题只要在之前那道题上稍加修改即可得到结果，还是需要用到数据结构队列queue，遍历每层的节点时，把下一层的节点都存入到queue中，每当开始新一层

节点的遍历之前，先把新一层最后一个节点值存到结果中，代码如下：

```

1 class Solution {
2 public:
3     vector<int> rightSideView(TreeNode *root) {
4         vector<int> res;
5         if (!root) return res;
6         queue<TreeNode*> q;
7         q.push(root);
8         while (!q.empty()) {
9             res.push_back(q.back()->val);
10            int size = q.size();
11            for (int i = 0; i < size; ++i) {
12                TreeNode *node = q.front();
13                q.pop();
14                if (node->left) q.push(node->left);
15                if (node->right) q.push(node->right);
16            }
17        }
18        return res;
19    }
20 };

```

CPP

200. 岛屿的数量

Given a 2d grid map of '1's (land) and '0's (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example 1:

```

11110
11010
11000
00000
Answer: 1

```

Example 2:

```

11000
11000
00100
00011
Answer: 3

```

Credits:

Special thanks to @mithmatt for adding this problem and creating all test cases.

这道求岛屿数量的题的本质是求矩阵中连续区域的个数，很容易想到需要用深度优先搜索DFS来解，我们需要建立一个visited数组用来记录某个位置是否被访问过，对于一个为'1'且未被访问过的位置，我们递归进入其上下左右位置上为'1'的数，将其visited对应值赋为true，继续进入其所有相连的邻位置，这样可以将这个连通区域所有的数找出来，并将其对应的visited中的值赋true，找完次区域后，我们将结果res自增1，然后我们在继续找下一个为'1'且未被访问过的位置，以此类推直至遍历完整个原数组即可得到最终结果，代码如下：

```

1 class Solution {
2 public:
3     int numIslands(vector<vector<char>> &grid) {
4         if (grid.empty() || grid[0].empty()) return 0;
5         int m = grid.size(), n = grid[0].size(), res = 0;
6         vector<vector<bool>> visited(m, vector<bool>(n, false));
7         for (int i = 0; i < m; ++i) {
8             for (int j = 0; j < n; ++j) {
9                 if (grid[i][j] == '1' && !visited[i][j]) {
10                     numIslandsDFS(grid, visited, i, j);
11                     ++res;
12                 }
13             }
14         }
15         return res;
16     }
17     void numIslandsDFS(vector<vector<char>> &grid, vector<vector<bool>> &visited, int x,
18 int y) {
19         if (x < 0 || x >= grid.size()) return;
20         if (y < 0 || y >= grid[0].size()) return;
21         if (grid[x][y] != '1' || visited[x][y]) return;
22         visited[x][y] = true;
23         numIslandsDFS(grid, visited, x - 1, y);
24         numIslandsDFS(grid, visited, x + 1, y);
25         numIslandsDFS(grid, visited, x, y - 1);
26         numIslandsDFS(grid, visited, x, y + 1);
27     }
};

```

201. 数字范围位相与

Given a range [m, n] where $0 \leq m \leq n \leq 2147483647$, return the bitwise AND of all numbers in this range, inclusive.

For example, given the range [5, 7], you should return 4.

Credits:

Special thanks to @amrsaqr for adding this problem and creating all test cases.

又是一道考察位操作Bit Operation的题，相似的题目在LeetCode中还真不少，比如Repeated DNA Sequences 求重复的DNA序列，Single Number 单独的数字，Single Number II 单独的数字之二，Grey Code 格雷码，和Reverse Bits 翻转位 等等，那么这道题其实并不难，我们先从题目中给的例子来分析，[5, 7]里共有三个数字，分别写出它们的二进制为：

101 110 111

相与后的结果为100，仔细观察我们可以得出，最后的数是该数字范围内所有的数的左边共同的部分，如果上面那个例子不太明显，我们再来看一个范围[26, 30]，它们的二进制如下：

11010 11011 11100 11101 11110

发现了规律后，我们只要写代码找到左边公共的部分即可，我们可以从建立一个32位都是1的mask，然后每次向左移一位，比较m和n是否相同，不同再继续左移一位，直至相同，然后把m和mask相与就是最终结果，代码如下：

解法1：

```

1 class Solution {
2 public:
3     int rangeBitwiseAnd(int m, int n) {
4         int d = INT_MAX;
5         while ((m & d) != (n & d)) {
6             d <= 1;
7         }
8         return m & d;
9     }
10 };

```

此题还有另一种解法，不需要用mask，直接平移m和n，每次向右移一位，直到m和n相等，记录下所有平移的次数i，然后再把m左移i位即为最终结果，代码如下：

解法2：

```

1 class Solution {
2 public:
3     int rangeBitwiseAnd(int m, int n) {
4         int i = 0;
5         while (m != n) {
6             m >>= 1;
7             n >>= 1;
8             ++i;
9         }
10        return (m << i);
11    }
12 };

```

下面这种方法有点叼，就一行搞定了，通过递归来做的，如果n大于m，那么就对m和n分别除以2，并且调用递归函数，对结果再乘以2，一定要乘回来，不然就不对了，就举一个最简单的例子， $m = 10, n = 11$ ，注意这里是二进制表示的，然后各自除以2，都变成了1，调用递归返回1，这时候要乘以2，才能变回10，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int rangeBitwiseAnd(int m, int n) {
4         return (n > m) ? (rangeBitwiseAnd(m / 2, n / 2) << 1) : m;
5     }
6 };

```

下面这种方法也不错，也很简单，如果m小于n就进行循环，n与上n-1，那么为什么要这样呢，举个简单的例子呗，110与上(110-1)，得到100，这不就相当于去掉最低位的1么，n就这样每次去掉最低位的1，如果小于等于m了，返回此时的n即可，参见代码如下：

解法4：

```

1 class Solution {
2 public:
3     int rangeBitwiseAnd(int m, int n) {
4         while (m < n) n &= (n - 1);
5         return n;
6     }
7 };

```

202. 快乐数

Write an algorithm to determine if a number is "happy".

A happy number is a number defined by the following process: Starting with any positive integer, replace the number by the sum of the squares of its digits, and repeat the process until the number equals 1 (where it will stay), or it loops endlessly in a cycle which does not include 1. Those numbers for which this process ends in 1 are happy numbers.

Example: 19 is a happy number

```

1^2 + 9^2 = 82
8^2 + 2^2 = 68
6^2 + 8^2 = 100
1^2 + 0^2 + 0^2 = 1

```

Credits:

Special thanks to @mithmatt and @ts for adding this problem and creating all test cases.

这道题定义了一种快乐数，就是说对于某一个正整数，如果对其各个位上的数字分别平方，然后再加起来得到一个新的数字，再进行同样的操作，如果最终结果变成了1，则说明是快乐数，如果一直循环但不是1的话，就不是快乐数，那么现在任意给我们一个正整数，让我们判断这个数是不是快乐数，题目中给的例子19是快乐数，那么我们来看一个不是快乐数的情况，比如数字11有如下的计算过程：

```

1^2 + 1^2 = 2
2^2 = 4
4^2 = 16
1^2 + 6^2 = 37
3^2 + 7^2 = 58
5^2 + 8^2 = 89
8^2 + 9^2 = 145
1^2 + 4^2 + 5^2 = 42
4^2 + 2^2 = 20
2^2 + 0^2 = 4

```

我们发现在算到最后时数字4又出现了，那么之后的数字又都会重复之前的顺序，这个循环中不包含1，那么数字11不是一个快乐数，发现了规律后就要考虑怎么用代码来实现，我们可以用set来记录所有出现过的数字，然后每出现一个新数字，在set中查看是否存在，若不存在则加入表中，若存在则跳出循环，并且判断此数是否为1，若为1返回true，不为1返回false，代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool isHappy(int n) {
4         set<int> s;
5         while (n != 1) {
6             int t = 0;
7             while (n) {
8                 t += (n % 10) * (n % 10);
9                 n /= 10;
10            }
11            n = t;
12            if (s.count(n)) break;
13            else s.insert(n);
14        }
15        return n == 1;
16    }
17 };

```

其实这道题也可以不用set来做，我们并不需要太多的额外空间，关于非快乐数有个特点，循环的数字中必定会有4，这里就不做证明了，我也没有证明，就是利用这个性质，就可以不用set了，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool isHappy(int n) {
4         while (n != 1 && n != 4) {
5             int t = 0;
6             while (n) {
7                 t += (n % 10) * (n % 10);
8                 n /= 10;
9            }
10            n = t;
11        }
12        return n == 1;
13    }
14 };

```

203. 移除链表元素

Remove all elements from a linked list of integers that have value val.

Example

Given: 1 --> 2 --> 6 --> 3 --> 4 --> 5 --> 6, val = 6
Return: 1 --> 2 --> 3 --> 4 --> 5

Credits:

Special thanks to @mithmatt for adding this problem and creating all test cases.

这道移除链表元素是链表的基本操作之一，没有太大的难度，就是考察了基本的链表遍历和设置指针的知识点，我们只需定义几个辅助指针，然后遍历原链表，遇到与给定值相同的元素，将该元素的前后连个节点连接起来，然后删除该元素即可，要注意的是还是需要在链表开头加上一个dummy node，具体实现参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     ListNode* removeElements(ListNode* head, int val) {
4         ListNode *dummy = new ListNode(-1), *pre = dummy;
5         dummy->next = head;
6         while (pre->next) {
7             if (pre->next->val == val) {
8                 ListNode *t = pre->next;
9                 pre->next = t->next;
10                t->next = NULL;
11                delete t;
12            } else {
13                pre = pre->next;
14            }
15        }
16        return dummy->next;
17    }
18 };

```

我们也可以用递归来解，写法很简洁，通过递归调用到链表末尾，然后回来，需要要删的元素，将链表next指针指向下一个元素即可：

解法2：

```

1 class Solution {
2 public:
3     ListNode* removeElements(ListNode* head, int val) {
4         if (!head) return NULL;
5         head->next = removeElements(head->next, val);
6         return head->val == val ? head->next : head;
7     }
8 };

```

204. 质数的个数

Description:

Count the number of prime numbers less than a non-negative number, n

[click to show more hints.](#)

References:

[How Many Primes Are There?](#)

[Sieve of Eratosthenes](#)

Credits:

Special thanks to @mithmatt for adding this problem and creating all test cases.

这道题给定一个非负数n，让我们求小于n的质数的个数，题目中给了充足的提示，解题方法就在第二个提示埃拉托斯特尼筛法 Sieve of Eratosthenes中，这个算法的过程如下图所示，我们从2开始遍历到根号n，先找到第一个质数2，然后将其所有的倍数全部标记出来，然后到下一个质数3，标记其所有倍数，一次类推，直到根号n，此时数组中未被标记的数字就是质数。我们需要一个n-1长度的bool型数组来记录每个数字是否被标记，长度为n-1的原因是题目说是小于n的质数个数，并不包括n。然后我们用两个for循环来实现埃拉托斯特尼筛法，难度并不是很大，代码如下所示：

Prime numbers									
2	3	4	5	6	7	8	9	10	
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120

CPP

```

1 class Solution {
2 public:
3     int countPrimes(int n) {
4         vector<bool> num(n - 1, true);
5         num[0] = false;
6         int res = 0, limit = sqrt(n);
7         for (int i = 2; i <= limit; ++i) {
8             if (num[i - 1]) {
9                 for (int j = i * i; j < n; j += i) {
10                     num[j - 1] = false;
11                 }
12             }
13         }
14         for (int j = 0; j < n - 1; ++j) {
15             if (num[j]) ++res;
16         }
17         return res;
18     }
19 };

```

205. 同构字符串

Given two strings s and t, determine if they are isomorphic.

Two strings are isomorphic if the characters in s can be replaced to get t.

All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character but a character may map to itself.

For example,

Given "egg", "add", return true.

Given "foo", "bar", return false.

Given "paper", "title", return true.

Note:

You may assume both s and t have the same length.

这道题让我们求同构字符串，就是说原字符串中的每个字符可由另外一个字符替代，可以被其本身替代，相同的字符一定要被同一个字符替代，且一个字符不能被多个字符替代，即不能出现一对多的映射。根据一对一映射的特点，我们需要用两个哈希表分别来记录原字符串和目标字符串中字符出现情况，由于ASCII码只有256个字符，所以我们可以用一个256大小的数组来代替哈希表，并初始化为0，我们遍历原字符串，分别从源字符串和目标字符串取出一个字符，然后分别在两个哈希表中查找其值，若不相等，则返回false，若相等，将其值更新为i + 1，因为默认的值是0，所以我们更新值为i + 1，这样当 i=0 时，则映射为1，如果不加1的话，那么就无法区分是否更新了，代码如下：

```

1 class Solution {
2 public:
3     bool isIsomorphic(string s, string t) {
4         int m1[256] = {0}, m2[256] = {0}, n = s.size();
5         for (int i = 0; i < n; ++i) {
6             if (m1[s[i]] != m2[t[i]]) return false;
7             m1[s[i]] = i + 1;
8             m2[t[i]] = i + 1;
9         }
10        return true;
11    }
12 };

```

CPP

206. 倒置链表

Reverse a singly linked list.

[click to show more hints.](#)

Hint:

A linked list can be reversed either iteratively or recursively. Could you implement both?

之前做到 Reverse Linked List II 倒置链表之二的时候我还纳闷怎么只有二没有一呢，原来真是忘了啊，现在才加上，这道题跟之前那道题比起来简单不少，题目为了增加些许难度，让我们分别用迭代和递归来实现，但难度还是不大。我们先来看迭代的解法，思路是在原链表之前建立一个dummy node，因为首节点会变，然后从head开始，将之后的一个节点移到dummy node之后，重复此操作知道head成为末节点为止，代码如下：

解法1：

```

1 class Solution {
2 public:
3     ListNode* reverseList(ListNode* head) {
4         if (!head) return head;
5         ListNode *dummy = new ListNode(-1);
6         dummy->next = head;
7         ListNode *cur = head;
8         while (cur->next) {
9             ListNode *tmp = cur->next;
10            cur->next = tmp->next;
11            tmp->next = dummy->next;
12            dummy->next = tmp;
13        }
14        return dummy->next;
15    }
16 };

```

CPP

下面我们来看递归解法，代码量更少，递归解法的思路是，不断的进入递归函数，直到head指向最后一个节点，p指向之前一个节点，然后调换head和p的位置，再返回上一层递归函数，再交换p和head的位置，每次交换后，head节点后面都是交换好的顺序，直到p为首个节点，然后再交换，首节点就成了为节点，此时整个链表也完成了翻转，代码如下：

解法2：

```
1 class Solution {
2     public:
3         ListNode* reverseList(ListNode* head) {
4             if (!head || !head->next) return head;
5             ListNode *p = head;
6             head = reverseList(p->next);
7             p->next->next = p;
8             p->next = NULL;
9             return head;
10        }
11    };
CPP
```

207. 课程清单

There are a total of n courses you have to take, labeled from 0 to n - 1.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: [0,1]

Given the total number of courses and a list of prerequisite pairs, is it possible for you to finish all courses?

For example:

2, [[1,0]]

There are a total of 2 courses to take. To take course 1 you should have finished course 0. So it is possible.

2, [[1,0],[0,1]]

There are a total of 2 courses to take. To take course 1 you should have finished course 0, and to take course 0 you should also have finished course 1. So it is impossible.

[click to show more hints.](#)

Hints:

This problem is equivalent to finding if a cycle exists in a directed graph. If a cycle exists, no topological ordering exists and therefore it will be impossible to take all courses.

There are several ways to represent a graph. For example, the input prerequisites is a graph represented by a list of edges. Is this graph representation appropriate?

Topological Sort via DFS - A great video tutorial (21 minutes) on Coursera explaining the basic concepts of Topological Sort.

Topological sort could also be done via BFS.

这道课程清单的问题对于我们学生来说应该不陌生，因为我们在选课的时候经常会遇到想选某一门课程，发现选它之前必须先上了哪些课程，这道题给了很多提示，第一条就告诉我们了这道题的本质就是在有向图中检测环。LeetCode中关于图的题很少，有向图的仅此一道，还有一道关于无向图的题是Clone Graph 无向图的复制。个人认为图这种数据结构相比于树啊，链表啊什么的要更为复杂一些，尤其是有向图，很麻烦。第二条提示是在讲如何来表示一个有向图，可以用边来表示，边是由两个端点组成的，用两个点来表示边。第三第四条提示揭示了此题有两种解法，DFS和BFS都可以解此题。我们先来看BFS的解法，我们定义二维数组graph来表示这个有向图，一位数组in来表示每个顶点的入度。我们开始先根据输入来建立这个有向图，并将入度数

组也初始化好。然后我们定义一个queue变量，将所有入度为0的点放入队列中，然后开始遍历队列，从graph里遍历其连接的点，每到达一个新节点，将其入度减一，如果此时该点入度为0，则放入队列末尾。直到遍历完队列中所有的值，若此时还有节点的入度不为0，则说明环存在，返回false，反之则返回true。代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool canFinish(int numCourses, vector<vector<int>>& prerequisites) {
4         vector<vector<int> > graph(numCourses, vector<int>(0));
5         vector<int> in(numCourses, 0);
6         for (auto a : prerequisites) {
7             graph[a[1]].push_back(a[0]);
8             ++in[a[0]];
9         }
10        queue<int> q;
11        for (int i = 0; i < numCourses; ++i) {
12            if (in[i] == 0) q.push(i);
13        }
14        while (!q.empty()) {
15            int t = q.front();
16            q.pop();
17            for (auto a : graph[t]) {
18                --in[a];
19                if (in[a] == 0) q.push(a);
20            }
21        }
22        for (int i = 0; i < numCourses; ++i) {
23            if (in[i] != 0) return false;
24        }
25        return true;
26    }
27 };

```

下面我们来看DFS的解法，也需要建立有向图，还是用二维数组来建立，和BFS不同的是，我们像现在需要一个一维数组visit来记录访问状态，这里有三种状态，0表示还未访问过，1表示已经访问了，-1表示有冲突。大体思路是，先建立好有向图，然后从第一个门课开始，找其可构成哪门课，暂时将当前课程标记为已访问，然后对新得到的课程调用DFS递归，直到出现新的课程已经访问过了，则返回false，没有冲突的话返回true，然后把标记为已访问的课程改为未访问。代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool canFinish(int numCourses, vector<vector<int> >& prerequisites) {
4         vector<vector<int> > graph(numCourses, vector<int>(0));
5         vector<int> visit(numCourses, 0);
6         for (auto a : prerequisites) {
7             graph[a[1]].push_back(a[0]);
8         }
9         for (int i = 0; i < numCourses; ++i) {
10             if (!canFinishDFS(graph, visit, i)) return false;
11         }
12         return true;
13     }
14     bool canFinishDFS(vector<vector<int> > &graph, vector<int> &visit, int i) {
15         if (visit[i] == -1) return false;
16         if (visit[i] == 1) return true;
17         visit[i] = -1;
18         for (auto a : graph[i]) {
19             if (!canFinishDFS(graph, visit, a)) return false;
20         }
21         visit[i] = 1;
22         return true;
23     }
24 };

```

208. 实现字典树(前缀树)

Implement a trie with insert, search, and startsWith methods.

Note:

You may assume that all inputs are consist of lowercase letters a-z.

字母树的插入（Insert）、删除（Delete）和查找（Find）都非常简单，用一个一重循环即可，即第*i* 次循环找到前*i* 个字母所对应的子树，然后进行相应的操作。实现这棵字母树，我们用最常见的数组保存（静态开辟内存）即可，当然也可以开动态的指针类型（动态开辟内存）。至于结点对儿子的指向，一般有三种方法：

- 1、对每个结点开一个字母集大小的数组，对应的下标是儿子所表示的字母，内容则是这个儿子对应在大数组上的位置，即标号；
- 2、对每个结点挂一个链表，按一定顺序记录每个儿子是谁；
- 3、使用左儿子右兄弟表示法记录这棵树。

三种方法，各有特点。第一种易实现，但实际的空间要求较大；第二种，较易实现，空间要求相对较小，但比较费时；第三种，空间要求最小，但相对费时且不易写。

我们先来看第一种实现方法，这种方法实现起来简单直观，字母的字典树每个节点要定义一个大小为26的子节点指针数组，然后用一个标志符用来记录到当前位置为止是否为一个词，初始化的时候讲26个子节点都赋为空。那么insert操作只需要对于要插入的字符串的每一个字符算出其的位置，然后找是否存在这个子节点，若不存在则新建一个，然后再查找下一个。查找词和找前缀操作跟insert操作都很类似，不同点在于若不存在子节点，则返回false。查找次最后还要看标识位，而找前缀直接返回true即可。代码如下：

```

1 class TrieNode {
2     public:
3         // Initialize your data structure here.
4         TrieNode *child[26];
5         bool isWord;
6         TrieNode() : isWord(false){
7             for (auto &a : child) a = NULL;
8         }
9     };
10
11 class Trie {
12     public:
13         Trie() {
14             root = new TrieNode();
15         }
16
17         // Inserts a word into the trie.
18         void insert(string s) {
19             TrieNode *p = root;
20             for (auto &a : s) {
21                 int i = a - 'a';
22                 if (!p->child[i]) p->child[i] = new TrieNode();
23                 p = p->child[i];
24             }
25             p->isWord = true;
26         }
27
28         // Returns if the word is in the trie.
29         bool search(string key) {
30             TrieNode *p = root;
31             for (auto &a : key) {
32                 int i = a - 'a';
33                 if (!p->child[i]) return false;
34                 p = p->child[i];
35             }
36             return p->isWord;
37         }
38
39         // Returns if there is any word in the trie
40         // that starts with the given prefix.
41         bool startsWith(string prefix) {
42             TrieNode *p = root;
43             for (auto &a : prefix) {
44                 int i = a - 'a';
45                 if (!p->child[i]) return false;
46                 p = p->child[i];
47             }
48             return true;
49         }
50
51     private:
52         TrieNode* root;
53     };

```

209. 最短子数组之和

Given an array of n positive integers and a positive integer s, find the minimal length of a subarray of which the sum \geq s. If there isn't one, return 0 instead.

For example, given the array [2,3,1,2,4,3] and s = 7, the subarray [4,3] has the minimal length under the problem constraint.

[click to show more practice.](#)

More practice:

If you have figured out the O(n) solution, try coding another solution of which the time complexity is O(n log n).

Credits:

Special thanks to @Freezen for adding this problem and creating all test cases.

这道题给定了我们一个数字，让我们求子数组之和大于等于给定值的最小长度，跟之前那道 Maximum Subarray 最大子数组有些类似，并且题目中要求我们实现O(n)和O(nlgn)两种解法，那么我们先来看O(n)的解法，我们需要定义两个指针left和right，分别记录子数组的左右的边界位置，然后我们让right向右移，直到子数组和大于等于给定值或者right达到数组末尾，此时我们更新最短距离，并且将left像右移一位，然后再sum中减去移去的值，然后重复上面的步骤，直到right到达末尾，且left到达临界位置，即要么到达边界，要么再往右移动，和就会小于给定值。代码如下：

解法1：

```

1 class Solution {
2 public:
3     int minSubArrayLen(int s, vector<int>& nums) {
4         if (nums.empty()) return 0;
5         int left = 0, right = 0, sum = 0, len = nums.size(), res = len + 1;
6         while (right < len) {
7             while (sum < s && right < len) {
8                 sum += nums[right++];
9             }
10            while (sum >= s) {
11                res = min(res, right - left);
12                sum -= nums[left++];
13            }
14        }
15        return res == len + 1 ? 0 : res;
16    }
17 };

```

CPP

同样的思路，我们也可以换一种写法，参考代码如下：

解法2：

```

1 class Solution {
2 public:
3     int minSubArrayLen(int s, vector<int>& nums) {
4         int res = INT_MAX, left = 0, sum = 0;
5         for (int i = 0; i < nums.size(); ++i) {
6             sum += nums[i];
7             while (left <= i && sum >= s) {
8                 res = min(res, i - left + 1);
9                 sum -= nums[left++];
10            }
11        }
12        return res == INT_MAX ? 0 : res;
13    }
14 };

```

下面我们再来看看O(nlg n)的解法，这个解法要用到二分查找法，思路是，我们建立一个比原数组长一位的sums数组，其中sums[i]表示nums数组中[0, i - 1]的和，然后我们对于sums中每一个值sums[i]，用二分查找法找到子数组的右边界位置，使该子数组之和大于sums[i] + s，然后我们更新最短长度的距离即可。代码如下：

解法3：

```

1 class Solution {
2 public:
3     int minSubArrayLen(int s, vector<int>& nums) {
4         int len = nums.size(), sums[len + 1] = {0}, res = len + 1;
5         for (int i = 1; i < len + 1; ++i) sums[i] = sums[i - 1] + nums[i - 1];
6         for (int i = 0; i < len + 1; ++i) {
7             int right = searchRight(i + 1, len, sums[i] + s, sums);
8             if (right == len + 1) break;
9             if (res > right - i) res = right - i;
10        }
11        return res == len + 1 ? 0 : res;
12    }
13    int searchRight(int left, int right, int key, int sums[]) {
14        while (left <= right) {
15            int mid = (left + right) / 2;
16            if (sums[mid] >= key) right = mid - 1;
17            else left = mid + 1;
18        }
19        return left;
20    }
21 };

```

我们也可以不用为二分查找法专门写一个函数，直接嵌套在for循环中即可，参加代码如下：

解法4：

```

1 class Solution {
2 public:
3     int minSubArrayLen(int s, vector<int>& nums) {
4         int res = INT_MAX, n = nums.size();
5         vector<int> sums(n + 1, 0);
6         for (int i = 1; i < n + 1; ++i) sums[i] = sums[i - 1] + nums[i - 1];
7         for (int i = 0; i < n; ++i) {
8             int left = i + 1, right = n, t = sums[i] + s;
9             while (left <= right) {
10                 int mid = left + (right - left) / 2;
11                 if (sums[mid] < t) left = mid + 1;
12                 else right = mid - 1;
13             }
14             if (left == n + 1) break;
15             res = min(res, left - i);
16         }
17         return res == INT_MAX ? 0 : res;
18     }
19 };

```

讨论：本题有一个很好的Follow up，就是去掉所有数字是正数的限制条件，而去掉这个条件会使得累加数组不一定会是递增的了，那么就不能使用二分法，同时双指针的方法也会失效，只能另辟蹊径了。其实博主觉得同时应该去掉大于s的条件，只保留 $\text{sum}=s$ 这个要求，因为这样我们可以再建立累加数组后用2sum的思路，快速查找 $s-\text{sum}$ 是否存在，如果有了大于的条件，还得继续遍历所有大于 $s-\text{sum}$ 的值，效率提高不了多少。

210. 课程清单之二

There are a total of n courses you have to take, labeled from 0 to n - 1.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: [0,1]

Given the total number of courses and a list of prerequisite pairs, return the ordering of courses you should take to finish all courses.

There may be multiple correct orders, you just need to return one of them. If it is impossible to finish all courses, return an empty array.

For example:

2, [[1,0]]

There are a total of 2 courses to take. To take course 1 you should have finished course 0. So the correct course order is [0,1]

4, [[1,0],[2,0],[3,1],[3,2]]

There are a total of 4 courses to take. To take course 3 you should have finished both courses 1 and 2. Both courses 1 and 2 should be taken after you finished course 0. So one correct course order is [0,1,2,3]. Another correct ordering is[0,2,1,3].

Note:

The input prerequisites is a graph represented by a list of edges, not adjacency matrices. Read more about how a graph is represented.

[click to show more hints.](#)

Hints:

This problem is equivalent to finding the topological order in a directed graph. If a cycle exists, no topological ordering exists and therefore it will be impossible to take all courses.
Topological Sort via DFS - A great video tutorial (21 minutes) on Coursera explaining the basic concepts of Topological Sort.

Topological sort could also be done via BFS.

这题是之前那道 Course Schedule 课程清单的扩展，那道题只让我们判断是否能完成所有课程，即检测有向图中是否有环，而这道题我们得找出要上的课程的顺序，即有向图的拓扑排序，这样一来，难度就增加了，但是由于我们有之前那道的基础，而此题正是基于之前解法的基础上稍加修改，我们从queue中每取出一个数组就将其存在结果中，最终若有向图中有环，则结果中元素的个数不等于总课程数，那我们将结果清空即可。代码如下：

```

1 class Solution {
2 public:
3     vector<int> findOrder(int numCourses, vector<pair<int, int>>& prerequisites) {
4         vector<int> res;
5         vector<vector<int>> graph(numCourses, vector<int>(0));
6         vector<int> in(numCourses, 0);
7         for (auto &a : prerequisites) {
8             graph[a.second].push_back(a.first);
9             ++in[a.first];
10        }
11        queue<int> q;
12        for (int i = 0; i < numCourses; ++i) {
13            if (in[i] == 0) q.push(i);
14        }
15        while (!q.empty()) {
16            int t = q.front();
17            res.push_back(t);
18            q.pop();
19            for (auto &a : graph[t]) {
20                --in[a];
21                if (in[a] == 0) q.push(a);
22            }
23        }
24        if (res.size() != numCourses) res.clear();
25        return res;
26    }
27 };

```

211. 添加和查找单词-数据结构设计

Design a data structure that supports the following two operations:

```

void addWord(word)
bool search(word)
search(word) can search a literal word or a regular expression string containing only letters
a-z or .. A . means it can represent any one letter.

```

For example:

```

addWord("bad")
addWord("dad")
addWord("mad")
search("pad") -> false
search("bad") -> true
search(".ad") -> true
search("b..") -> true

```

Note:

You may assume that all words are consist of lowercase letters a-z.

[click to show hint.](#)

You should be familiar with how a Trie works. If not, please work on this problem: Implement Trie (Prefix Tree) first.

LeetCode出新题的速度越来越快了，有点跟不上节奏的感觉了。这道题如果做过之前的那道 Implement Trie (Prefix Tree) 实现字典树(前缀树)的话就没有太大的难度了，还是要用到字典树的结构，唯一不同的地方就是search的函数需要重新写一下，因为这道题里面'.'可以代替任意字符，所以一旦有了'.'，就需要查找所有的子树，只要有一个返回true，整个search函数就返回

true，典型的DFS的问题，其他部分跟上一道实现字典树没有太大区别，代码如下：

CPP

```

1 class WordDictionary {
2 public:
3     struct TrieNode {
4         public:
5             TrieNode *child[26];
6             bool isWord;
7             TrieNode() : isWord(false) {
8                 for (auto &a : child) a = NULL;
9             }
10            };
11
12        WordDictionary() {
13            root = new TrieNode();
14        }
15
16        // Adds a word into the data structure.
17        void addWord(string word) {
18            TrieNode *p = root;
19            for (auto &a : word) {
20                int i = a - 'a';
21                if (!p->child[i]) p->child[i] = new TrieNode();
22                p = p->child[i];
23            }
24            p->isWord = true;
25        }
26
27        // Returns if the word is in the data structure. A word could
28        // contain the dot character '.' to represent any one letter.
29        bool search(string word) {
30            return searchWord(word, root, 0);
31        }
32
33        bool searchWord(string &word, TrieNode *p, int i) {
34            if (i == word.size()) return p->isWord;
35            if (word[i] == '.') {
36                for (auto &a : p->child) {
37                    if (a && searchWord(word, a, i + 1)) return true;
38                }
39                return false;
40            } else {
41                return p->child[word[i] - 'a'] && searchWord(word, p->child[word[i] - 'a'], i +
42 1);
43            }
44        }
45
46    private:
47        TrieNode *root;
48    };

```

212. 词语搜索之二

Given a 2D board and a list of words from the dictionary, find all words in the board.

Each word must be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once in a word.

For example,

Given words = ["oath", "pea", "eat", "rain"] and board =

```
[  
  ['o','a','a','n'],  
  ['e','t','a','e'],  
  ['i','h','k','r'],  
  ['i','f','l','v']  
]
```

Return ["eat", "oath"].

Note:

You may assume that all inputs are consist of lowercase letters a-z.

[click to show hint.](#)

You would need to optimize your backtracking to pass the larger test. Could you stop backtracking earlier?

If the current candidate does not exist in all words' prefix, you could stop backtracking immediately. What kind of data structure could answer such query efficiently? Does a hash table work? Why or why not? How about a Trie? If you would like to learn how to implement a basic trie, please work on this problem: [Implement Trie \(Prefix Tree\)](#) first.

这道题是在之前那道Word Search 词语搜索的基础上做了些拓展，之前是给一个单词让判断是否存在，现在是给了一堆单词，让返回所有存在的单词，在这道题最开始更新的几个小时内，用brute force是可以通过OJ的，就是在之前那题的基础上多加一个for循环而已，但是后来出题者其实是想考察字典树的应用，所以加了一个超大的test case，以至于brute force无法通过，强制我们必须用字典树来求解。LeetCode中有关字典树的题还有 [Implement Trie \(Prefix Tree\)](#) 实现字典树(前缀树)和 [Add and Search Word - Data structure design](#) 添加和查找单词-数据结构设计，那么我们在这题中只要实现字典树中的insert功能就行了，查找单词和前缀就没有必要了，然后DFS的思路跟之前那道Word Search 词语搜索基本相同，请参见代码如下：

```

1 class Solution {
2 public:
3     struct TrieNode {
4         TrieNode *child[26];
5         string str;
6         TrieNode() : str("") {
7             for (auto &a : child) a = NULL;
8         }
9     };
10    struct Trie {
11        TrieNode *root;
12        Trie() : root(new TrieNode()) {}
13        void insert(string s) {
14            TrieNode *p = root;
15            for (auto &a : s) {
16                int i = a - 'a';
17                if (!p->child[i]) p->child[i] = new TrieNode();
18                p = p->child[i];
19            }
20            p->str = s;
21        }
22    };
23    vector<string> findWords(vector<vector<char>>& board, vector<string>& words) {
24        vector<string> res;
25        if (words.empty() || board.empty() || board[0].empty()) return res;
26        vector<vector<bool>> visit(board.size(), vector<bool>(board[0].size(), false));
27        Trie T;
28        for (auto &a : words) T.insert(a);
29        for (int i = 0; i < board.size(); ++i) {
30            for (int j = 0; j < board[i].size(); ++j) {
31                if (T.root->child[board[i][j] - 'a']) {
32                    search(board, T.root->child[board[i][j] - 'a'], i, j, visit, res);
33                }
34            }
35        }
36        return res;
37    }
38    void search(vector<vector<char>> &board, TrieNode *p, int i, int j,
39    vector<vector<bool>> &visit, vector<string> &res) {
40        if (!p->str.empty()) {
41            res.push_back(p->str);
42            p->str.clear();
43        }
44        int d[][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
45        visit[i][j] = true;
46        for (auto &a : d) {
47            int nx = a[0] + i, ny = a[1] + j;
48            if (nx >= 0 && nx < board.size() && ny >= 0 && ny < board[0].size() &&
49            !visit[nx][ny] && p->child[board[nx][ny] - 'a']) {
50                search(board, p->child[board[nx][ny] - 'a'], nx, ny, visit, res);
51            }
52        }
53        visit[i][j] = false;
54    }
55};

```

213. 打家劫舍之二

Note: This is an extension of House Robber.

After robbing those houses on that street, the thief has found himself a new place for his thievery so that he will not get too much attention. This time, all houses at this place are arranged in a circle. That means the first house is the neighbor of the last one. Meanwhile, the security system for these houses remain the same as for those in the previous street.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight without alerting the police.

Credits:

Special thanks to @Freezen for adding this problem and creating all test cases.

这道题是之前那道House Robber 打家劫舍的拓展，现在房子排成了一个圆圈，则如果抢了第一家，就不能抢最后一家，因为首尾相连了，所以第一家和最后一家只能抢其中的一家，或者都不抢，那我们这里变通一下，如果我们把第一家和最后一家分别去掉，各算一遍能抢的最大值，然后比较两个值取其中较大的一个即为所求。那我们只需参考之前的House Robber 打家劫舍中的解题方法，然后调用两边取较大值，代码如下：

解法1：

```

1 class Solution {
2 public:
3     int rob(vector<int>& nums) {
4         if (nums.size() <= 1) return nums.empty() ? 0 : nums[0];
5         return max(rob(nums, 0, nums.size() - 1), rob(nums, 1, nums.size()));
6     }
7     int rob(vector<int> &nums, int left, int right) {
8         if (right - left <= 1) return nums[left];
9         vector<int> dp(right, 0);
10        dp[left] = nums[left];
11        dp[left + 1] = max(nums[left], nums[left + 1]);
12        for (int i = left + 2; i < right; ++i) {
13            dp[i] = max(nums[i] + dp[i - 2], dp[i - 1]);
14        }
15        return dp.back();
16    }
17 };

```

解法2：

```

1 class Solution {
2 public:
3     int rob(vector<int>& nums) {
4         if (nums.size() <= 1) return nums.empty() ? 0 : nums[0];
5         return max(rob(nums, 0, nums.size() - 1), rob(nums, 1, nums.size()));
6     }
7     int rob(vector<int> &nums, int left, int right) {
8         int a = 0, b = 0;
9         for (int i = left; i < right; ++i) {
10             int m = a, n = b;
11             a = n + nums[i];
12             b = max(m, n);
13         }
14         return max(a, b);
15     }
16 };

```

解法3：

```

1 class Solution {
2 public:
3     int rob(vector<int>& nums) {
4         if (nums.size() <= 1) return nums.empty() ? 0 : nums[0];
5         vector<int> v1 = nums, v2 = nums;
6         v1.erase(v1.begin()); v2.pop_back();
7         return max(rob_house(v1), rob_house(v2));
8     }
9     int rob_house(vector<int> &nums) {
10         int a = 0, b = 0;
11         for (int i = 0; i < nums.size(); ++i) {
12             if (i % 2 == 0) {
13                 a += nums[i];
14                 a = max(a, b);
15             } else {
16                 b += nums[i];
17                 b = max(a, b);
18             }
19         }
20         return max(a, b);
21     }
22 };

```

214. 最短回文串

Given a string S, you are allowed to convert it to a palindrome by adding characters in front of it. Find and return the shortest palindrome you can find by performing this transformation.

For example:

Given "aacecaaa", return "aaacecaaa".

Given "abcd", return "dcbabcd".

Credits:

Special thanks to @ifanchu for adding this problem and creating all test cases. Thanks to @Freezen for additional test cases.

这道题让我们求最短的回文串，LeetCode中关于回文串的其他的题目有 Palindrome Number 验证回文数字，Validate Palindrome 验证回文字符串，Palindrome Partitioning 拆分回文串，Palindrome Partitioning II 拆分回文串之二和 Longest Palindromic Substring 最长回文串。题目让我们在给定字符串s的前面加上最少个字符，使之变成回文串，那么我们来看题目中给的两个例子，最坏的情况下是s中没有相同的字符，那么最小需要添加字符的个数为s.size() - 1个，第一个例子的字符串包含一个回文串，只需再在前面添加一个字符即可，还有一点需要注意的是，前面添加的字符串都是从s的末尾开始，一位一位往前添加的，那么我们只需要知道从s末尾开始需要添加到前面的个数。这道题如果用brute force无法通过OJ，所以我们需要用一些比较巧妙的方法来解。这里我们用到了KMP算法，KMP算法是一种专门用来匹配字符串的高效的算法，具体方法可以参见这篇博文从头到尾彻底理解KMP。我们把s和其转置r连接起来，中间加上一个其他字符，形成一个新的字符串t，我们还需要一个和t长度相同的一位数组next，其中next[i]表示从t[i]到开头的子串的相同前缀后缀的个数，具体可参考KMP算法中解释。最后我们把不相同的个数对应的字符串添加到s之前即可，代码如下：

解法1：

```

1 class Solution {
2 public:
3     string shortestPalindrome(string s) {
4         string r = s;
5         reverse(r.begin(), r.end());
6         string t = s + "#" + r;
7         vector<int> next(t.size(), 0);
8         for (int i = 1; i < t.size(); ++i) {
9             int j = next[i - 1];
10            while (j > 0 && t[i] != t[j]) j = next[j - 1];
11            next[i] = (j += t[i] == t[j]);
12        }
13        return r.substr(0, s.size() - next.back()) + s;
14    }
15 }
```

CPP

从上面的Java和C的代码中，我们可以看出来C和Java在使用双等号上的明显的不同，感觉Java对于双等号对使用更加苛刻一些，比如Java中的双等号只对primitive类数据结构(比如int, char等)有效，但是即便有效，也不能将结果直接当1或者0来用。而对于一些从Object派生出来的类，比如Integer或者String等，不能直接用双等号来比较，而是要用其自带的equals()函数来比较，因为双等号判断的是不是同一个对象，而不是他们所表示的值是否相同。同样需要注意的是，Stack的peek()函数取出的也是对象，不能直接和另一个Stack的peek()取出的对象直接双等，而是使用equals或者先将其中一个强行转换成primitive类，再和另一个强行比较。

下面这种方法的写法比较简洁，虽然不是明显的KMP算法，但是也有其的影子在里面，首先我们还是先将其的翻转字符串搞出来，然后比较原字符串s的前缀后翻转字符串t的对应位置的后缀是否相等，起始位置是比较s和t是否相等，如果相等，说明s本身就是回文串，不用添加任何字符，直接返回即可；如果不相等，s去掉最后一位，t去掉第一位，继续比较，以此类推直至有相等，或者循环结束，这样我们就能将两个字符串在正确的位置拼接起来了。很有意思的是，这种方法对应Java写法却会TLE，无法通过OJ。

解法2：

```

1 class Solution {
2 public:
3     string shortestPalindrome(string s) {
4         string t = s;
5         reverse(t.begin(), t.end());
6         int n = s.size(), i = 0;
7         for (i = n; i >= 0; --i) {
8             if (s.substr(0, i) == t.substr(n - i)) {
9                 break;
10            }
11        }
12        return t.substr(0, n - i) + s;
13    }
14 };

```

215. 数组中第k大的数字

Find the kth largest element in an unsorted array. Note that it is the kth largest element in the sorted order, not the kth distinct element.

Example 1:

Input: [3,2,1,5,6,4] and k = 2

Output: 5

Example 2:

Input: [3,2,3,1,2,4,5,5,6] and k = 4

Output: 4

Note:

You may assume k is always valid, $1 \leq k \leq$ array's length.

这道题让我们求数组中第k大的数字，怎么求呢，当然首先想到的是给数组排序，然后求可以得到第k大的数字。先看一种利用c++的STL中的集成的排序方法，不用我们自己实现，这样的话这道题只要两行就完事了，代码如下：

解法1:

```

1 class Solution {
2 public:
3     int findKthLargest(vector<int>& nums, int k) {
4         sort(nums.begin(), nums.end());
5         return nums[nums.size() - k];
6     }
7 };

```

下面这种解法是利用了priority_queue的自动排序的特性，跟上面的解法思路上没有什么区别，当然我们也可以换成multiset来做，一个道理，参见代码如下：

解法2:

```

1 class Solution {
2 public:
3     int findKthLargest(vector<int>& nums, int k) {
4         priority_queue<int> q(nums.begin(), nums.end());
5         for (int i = 0; i < k - 1; ++i) {
6             q.pop();
7         }
8         return q.top();
9     }
10 };

```

上面两种方法虽然简洁，但是确不是本题真正想考察的东西，可以说有一定的偷懒嫌疑。这道题最好的解法应该是下面这种做法，用到了快速排序Quick Sort的思想，这里排序的方向是从小往大排。对快排不熟悉的童鞋们随意上网搜些帖子看下吧，多如牛毛啊，总有一款适合你。核心思想是每次都要先找一个中枢点Pivot，然后遍历其他所有的数字，像这道题从小往大排的话，就把小于中枢点的数字放到左半边，把大于中枢点的放在右半边，这样中枢点是整个数组中第几大的数字就确定了，虽然左右两部分不一定是完全有序的，但是并不影响本题要求的结果，所以我们求出中枢点的位置，如果正好是k-1，那么直接返回该位置上的数字；如果大于k-1，说明要求的数字在左半部分，更新右边界，再求新的中枢点位置；反之则更新右半部分，求中枢点的位置；不得不说，这个思路真的是巧妙啊～

解法3：

```

1 class Solution {
2 public:
3     int findKthLargest(vector<int>& nums, int k) {
4         int left = 0, right = nums.size() - 1;
5         while (true) {
6             int pos = partition(nums, left, right);
7             if (pos == k - 1) return nums[pos];
8             else if (pos > k - 1) right = pos - 1;
9             else left = pos + 1;
10        }
11    }
12    int partition(vector<int>& nums, int left, int right) {
13        int pivot = nums[left], l = left + 1, r = right;
14        while (l <= r) {
15            if (nums[l] < pivot && nums[r] > pivot) {
16                swap(nums[l++], nums[r--]);
17            }
18            if (nums[l] >= pivot) ++l;
19            if (nums[r] <= pivot) --r;
20        }
21        swap(nums[left], nums[r]);
22        return r;
23    }
24 };

```

216. 组合之和三

Find all possible combinations of k numbers that add up to a number n, given that only numbers from 1 to 9 can be used and each combination should be a unique set of numbers.

Ensure that numbers within the set are sorted in ascending order.

Example 1:

Input: k = 3, n = 7

Output:

[[1,2,4]]

这道题是组合之和系列的第三道题，跟之前两道Combination Sum 组合之和，Combination Sum II 组合之和之二都不太一样，那两道的联系比较紧密，变化不大，而这道跟它们最显著的不同就是这道题的个数是固定的，为k。个人认为这道题跟那道 Combinations 组合项更相似一些，但是那道题只是排序，对k个数字之和又没有要求。所以实际上这道题是它们的综合体，两者杂糅到一起就是这道题的解法了，n是k个数字之和，如果n小于0，则直接返回，如果n正好等于0，而且此时out中数字的个数正好为k，说明此时是一个正确解，将其存入结果res中，具体实现参见代码如下：

```

1 class Solution {
2 public:
3     vector<vector<int>> combinationSum3(int k, int n) {
4         vector<vector<int>> res;
5         vector<int> out;
6         combinationSum3DFS(k, n, 1, out, res);
7         return res;
8     }
9     void combinationSum3DFS(int k, int n, int level, vector<int> &out, vector<vector<int>>
10 &res) {
11         if (n < 0) return;
12         if (n == 0 && out.size() == k) res.push_back(out);
13         for (int i = level; i <= 9; ++i) {
14             out.push_back(i);
15             combinationSum3DFS(k, n - i, i + 1, out, res);
16             out.pop_back();
17         }
18     }
};
```

CPP

217. 包含重复值

Given an array of integers, find if the array contains any duplicates. Your function should return true if any value appears at least twice in the array, and it should return false if every element is distinct.

这道题不算难题，就是使用一个哈希表，遍历整个数组，如果哈希表里存在，返回false，如果不存在，则将其放入哈希表中，代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool containsDuplicate(vector<int>& nums) {
4         unordered_map<int, int> m;
5         for (int i = 0; i < nums.size(); ++i) {
6             if (m.find(nums[i]) != m.end()) return true;
7             ++m[nums[i]];
8         }
9         return false;
10    }
11 };

```

这题还有另一种解法，就是先将数组排个序，然后再比较相邻两个数字是否相等，时间复杂度取决于排序方法，代码如下：

解法2：

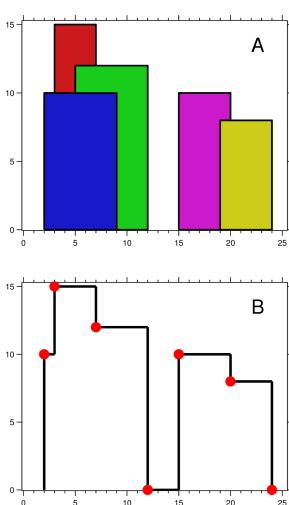
```

1 class Solution {
2 public:
3     bool containsDuplicate(vector<int>& nums) {
4         sort(nums.begin(), nums.end());
5         for (int i = 1; i < nums.size(); ++i) {
6             if (nums[i] == nums[i - 1]) return true;
7         }
8         return false;
9     }
10 };

```

218. 天际线问题

A city's skyline is the outer contour of the silhouette formed by all the buildings in that city when viewed from a distance. Now suppose you are given the locations and height of all the buildings as shown on a cityscape photo (Figure A), write a program to output the skyline formed by these buildings collectively (Figure B).



这道题一打开又是图又是这么长的题目的，看起来感觉应该是一道相当复杂的题，但是做完之后发现也就那么回事，虽然我不会做，是学习的别人的解法。这道求天际线的题目应该算是比较新颖的题，要是非要在之前的题目中找一道类似的题，也就只有 Merge Intervals 合并区间了吧，但是与那题不同的是，这道题不是求被合并成的空间，而是求轮廓线的一些关键的转折点，这就比较复杂了，我们通过仔细观察题目中给的那个例子可以发现，要求的红点都跟每个小区间的左右区间点有密切的关系，而且进一步发现除了每一个封闭区间的最右边的结束点是楼的右边界点，其余的都是左边界点，而且每个红点的纵坐标都是当前重合处的最高楼的高度，但是在右边界的那个楼的就不算了。在网上搜了很多帖子，发现网友Brian Gordon的帖子图文并茂，什么

动画渐变啊，横向扫描啊，简直叼到没朋友啊，但是叼到极致后就懒的一句一句的去读了，这里博主还是讲解另一位网友百草园的博客吧。这里用到了multiset数据结构，其好处在于其中的元素是按堆排好序的，插入新元素进去还是有序的，而且执行删除元素也可方便的将元素删掉。这里为了区分左右边界，将左边界的高度存为负数，建立左边界和负高度的pair，再建立右边界和高度的pair，存入数组中，都存进去了以后，给数组按照左边界排序，这样我们就可以按顺序来处理那些关键的节点了。我们要在multiset中放入一个0，这样在某个没有和其他建筑重叠的右边界上，我们就可以将封闭点存入结果res中。下面我们按顺序遍历这些关键节点，如果遇到高度为负值的pair，说明是左边界，那么将正高度加入multiset中，然后取出此时集合中最高的高度，即最后一个数字，然后看是否跟pre相同，这里的pre是上一个状态的高度，初始化为0，所以第一个左边界的高度绝对不为0，所以肯定会存入结果res中。接下来如果碰到了一个更高的楼的左边界的话，新高度存入multiset的话会排在最后面，那么此时cur取来也跟pre不同，可以将新的左边界点加入结果res。第三个点遇到绿色建筑的左边界点时，由于其高度低于红色的楼，所以cur取出来还是红色楼的高度，跟pre相同，直接跳过。下面遇到红色楼的右边界，此时我们首先将红色楼的高度从multiset中删除，那么此时cur取出的绿色楼的高度就是最高啦，跟pre不同，则可以将红楼的右边界横坐标和绿楼的高度组成pair加到结果res中，这样就成功的找到我们需要的拐点啦，后面都是这样类似的情况。当某个右边界点没有跟任何楼重叠的话，删掉当前的高度，那么multiset中就只剩0了，所以跟当前的右边界横坐标组成pair就是封闭点啦，具体实现参看代码如下：

CPP

```

1 class Solution {
2 public:
3     vector<pair<int, int>> getSkyline(vector<vector<int>>& buildings) {
4         vector<pair<int, int>> h, res;
5         multiset<int> m;
6         int pre = 0, cur = 0;
7         for (auto &a : buildings) {
8             h.push_back({a[0], -a[2]});
9             h.push_back({a[1], a[2]});
10        }
11        sort(h.begin(), h.end());
12        m.insert(0);
13        for (auto &a : h) {
14            if (a.second < 0) m.insert(-a.second);
15            else m.erase(m.find(a.second));
16            cur = *m.rbegin();
17            if (cur != pre) {
18                res.push_back({a.first, cur});
19                pre = cur;
20            }
21        }
22        return res;
23    }
24 };

```

219. 包含重复值之二

Given an array of integers and an integer k, find out whether there are two distinct indices i and j in the array such that nums[i] = nums[j] and the difference between i and j is at most k. (New Version)

这道题是之前那道Contains Duplicate 包含重复值的延伸，不同之处在于那道题只要我们判断下数组中是否有重复值，而这道题限制了数组中只许有一组重复的数字，而且他们坐标差不能超过k。那么我们首先需要一个哈希表，来记录每个数字和其坐标的映射，然后我们需要一个变量d来记录第一次出现重复数字的坐标差。由于题目要求只能有一组重复的数字，所以我们在遇到重复数字时，首先判断d是否已经存了值，如果d已经有值了，说明之前有过了重复数字，则直接返回false即可。如果没有，则此时给d附上值。在网上看到有些解法在这里就直接判断d和k的关系然后返回结果了，其实这样是不对的。因为题目要求只能有一组重复数，就是说如果后面又出现了重复数，就没法继续判断了。所以正确的做法应该是扫描完整个数组后在判断，先看d有没有存入结果，如果没有，则说明没出现过重复数，返回false即可。如果d有值，再跟k比较，返回对应的结果。OJ的test case没有包含所有的情况，比如当nums = [1, 2, 3, 1, 3], k = 3时，实际上应该返回false，但是有些返回true的算法也能通过OJ。个人认为正确的解法应该如下：

```

1 class Solution {
2 public:
3     bool containsNearbyDuplicate(vector<int>& nums, int k) {
4         unordered_map<int, int> m;
5         for (int i = 0; i < nums.size(); ++i) {
6             if (m.find(nums[i]) != m.end() && i - m[nums[i]] <= k) return true;
7             else m[nums[i]] = i;
8         }
9         return false;
10    }
11 };

```

220. 包含重复值之三

Given an array of integers, find out whether there are two distinct indices i and j in the array such that the difference between $\text{nums}[i]$ and $\text{nums}[j]$ is at most t and the difference between i and j is at most k .

这道题跟之前两道Contains Duplicate 包含重复值和Contains Duplicate II 包含重复值之二的关联并不是很大，前两道起码跟重复值有关，这道题的焦点不是在重复值上面，反而是关注与不同的值之间的关系，这里有两个限制条件，两个数字的坐标差不能大于 k ，值差不能大于 t 。这道题如果用brute force会超时，所以我们只能另辟蹊径。这里我们使用map数据结构来解，用来记录数字和其下标之间的映射。这里需要两个指针 i 和 j ，刚开始 i 和 j 都指向0，然后 i 开始向右走遍历数组，如果 i 和 j 之差大于 k ，且 m 中有 $\text{nums}[j]$ ，则删除并 j 加一。这样保证了 m 中所有的数的下标之差都不大于 k ，然后我们用map数据结构的lower_bound()函数来找一个特定范围，就是大于或等于 $\text{nums}[i] - t$ 的地方，所有小于这个阈值的数和 $\text{nums}[i]$ 的差的绝对值会大于 t （可自行带数检验）。然后检测后面的所有的数字，如果数的差的绝对值小于等于 t ，则返回true。最后遍历完整个数组返回false。代码如下：

```

1 class Solution {
2 public:
3     bool containsNearbyAlmostDuplicate(vector<int>& nums, int k, int t) {
4         map<long long, int> m;
5         int j = 0;
6         for (int i = 0; i < nums.size(); ++i) {
7             if (i - j > k) m.erase(nums[j++]);
8             auto a = m.lower_bound((long long)nums[i] - t);
9             if (a != m.end() && abs(a->first - nums[i]) <= t) return true;
10            m[nums[i]] = i;
11        }
12        return false;
13    }
14 };

```

221. 最大正方形

Given a 2D binary matrix filled with 0's and 1's, find the largest square containing all 1's and return its area.

For example, given the following matrix:

```
1 0 1 0 0
1 0 1 1 1
1 1 1 1 1
1 0 0 1 0
Return 4.
```

Credits:

Special thanks to @Freezen for adding this problem and creating all test cases.

这道题我刚看到的时候，马上联想到了之前的一道Number of Islands 岛屿的数量，但是仔细一对比，发现又不太一样，那道题1的形状不确定，很适合DFS的特点，而这道题要找的是正方形，是非常有特点的形状，所以并不需要用到DFS，要论相似，我倒认为这道Maximal Rectangle 最大矩形更相似一些。这道题的解法不止一种，我们先来看一种brute force的方法，这种方法的机理就是就是把数组中每一个点都当成正方形的左顶点来向右下方扫描，来寻找最大正方形。具体的扫描方法是，确定了左顶点后，再往下扫的时候，正方形的竖边长度就确定了，只需要找到横边即可，这时候我们使用直方图的原理，从其累加值能反映出上面的值是否全为1，之前也有一道关于直方图的题Largest Rectangle in Histogram 直方图中最大的矩形。通过这种方法我们就可以找出最大的正方形，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     int maximalSquare(vector<vector<char> >& matrix) {
4         int res = 0;
5         for (int i = 0; i < matrix.size(); ++i) {
6             vector<int> v(matrix[i].size(), 0);
7             for (int j = i; j < matrix.size(); ++j) {
8                 for (int k = 0; k < matrix[j].size(); ++k) {
9                     if (matrix[j][k] == '1') ++v[k];
10                }
11                res = max(res, getSquareArea(v, j - i + 1));
12            }
13        }
14        return res;
15    }
16    int getSquareArea(vector<int> &v, int k) {
17        if (v.size() < k) return 0;
18        int count = 0;
19        for (int i = 0; i < v.size(); ++i) {
20            if (v[i] != k) count = 0;
21            else ++count;
22            if (count == k) return k * k;
23        }
24        return 0;
25    }
26};
```

下面这个方法用到了建立累计和数组的方法，可以参见之前那篇博客Range Sum Query 2D - Immutable。原理是建立好了累加和数组后，我们开始遍历二维数组的每一个位置，对于任意一个位置(i, j)，我们从该位置往(0,0)点遍历所有的正方形，正方形的个数为 $\min(i,j)+1$ ，由于我们有了累加和矩阵，能快速的求出任意一个区域之和，所以我们能快速得到所有子正方形之和，比较正方形之和跟边长的平方是否相等，相等说明正方形中的数字均为1，更新res结果即可，参见代码如下：

解法2：

CPP

```

1 class Solution {
2 public:
3     int maximalSquare(vector<vector<char>>& matrix) {
4         if (matrix.empty() || matrix[0].empty()) return 0;
5         int m = matrix.size(), n = matrix[0].size(), res = 0;
6         vector<vector<int>> sum(m, vector<int>(n, 0));
7         for (int i = 0; i < matrix.size(); ++i) {
8             for (int j = 0; j < matrix[i].size(); ++j) {
9                 int t = matrix[i][j] - '0';
10                if (i > 0) t += sum[i - 1][j];
11                if (j > 0) t += sum[i][j - 1];
12                if (i > 0 && j > 0) t -= sum[i - 1][j - 1];
13                sum[i][j] = t;
14                int cnt = 1;
15                for (int k = min(i, j); k >= 0; --k) {
16                    int d = sum[i][j];
17                    if (i - cnt >= 0) d -= sum[i - cnt][j];
18                    if (j - cnt >= 0) d -= sum[i][j - cnt];
19                    if (i - cnt >= 0 && j - cnt >= 0) d += sum[i - cnt][j - cnt];
20                    if (d == cnt * cnt) res = max(res, d);
21                    ++cnt;
22                }
23            }
24        }
25        return res;
26    }
27 };

```

我们还可以进一步的优化时间复杂度到O(n²)，做法是使用DP，简历一个二维dp数组，其中dp[i][j]表示到达(i, j)位置所能组成最大正方形的边长。我们首先来考虑边界情况，也就是当i或j为0的情况，那么在首行或者首列中，必定有一个方向长度为1，那么就无法组成长度超过1的正方形，最多能组成长度为1的正方形，条件是当前位置为1。边界条件处理完了，再来看一般情况的递推公式怎么办，对于任意一点dp[i][j]，由于该点是正方形的右下角，所以该点的右边，下边，右下边都不用考虑，关心的就是左边，上边，和左上边。这三个位置的dp值suppose都应该算好的，还有就是要知道一点，只有当前(i, j)位置为1，dp[i][j]才有可能大于0，否则dp[i][j]一定为0。当(i, j)位置为1，此时要看dp[i-1][j-1], dp[i][j-1], 和dp[i-1][j]这三个位置，我们找其中最小的值，并加上1，就是dp[i][j]的当前值了，这个并不难想，毕竟不能有0存在，所以只能取交集，最后再用dp[i][j]的值来更新结果res的值即可，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int maximalSquare(vector<vector<char>>& matrix) {
4         if (matrix.empty() || matrix[0].empty()) return 0;
5         int m = matrix.size(), n = matrix[0].size(), res = 0;
6         vector<vector<int>> dp(m, vector<int>(n, 0));
7         for (int i = 0; i < m; ++i) {
8             for (int j = 0; j < n; ++j) {
9                 if (i == 0 || j == 0) dp[i][j] = matrix[i][j] - '0';
10                else if (matrix[i][j] == '1') {
11                    dp[i][j] = min(dp[i - 1][j - 1], min(dp[i][j - 1], dp[i - 1][j])) + 1;
12                }
13                res = max(res, dp[i][j]);
14            }
15        }
16        return res * res;
17    }
18 };

```

下面这种解法进一步的优化了空间复杂度，此时只需要用一个一维数组就可以解决，为了处理边界情况，padding了一位，所以dp的长度是m+1，然后还需要一个变量pre来记录上一个层的dp值，我们更新的顺序是行优先，就是先往下遍历，用一个临时变量t保存当前dp值，然后看如果当前位置为1，则更新dp[i]为dp[i], dp[i-1], 和pre三者之间的最小值，再加上1，来更新结果res，如果当前位置为0，则重置当前dp值为0，因为只有一维数组，每个位置会被重复使用，参见代码如下：

解法4：

```

1 class Solution {
2 public:
3     int maximalSquare(vector<vector<char>>& matrix) {
4         if (matrix.empty() || matrix[0].empty()) return 0;
5         int m = matrix.size(), n = matrix[0].size(), res = 0, pre = 0;
6         vector<int> dp(m + 1, 0);
7         for (int j = 0; j < n; ++j) {
8             for (int i = 1; i <= m; ++i) {
9                 int t = dp[i];
10                if (matrix[i - 1][j] == '1') {
11                    dp[i] = min(dp[i], min(dp[i - 1], pre)) + 1;
12                    res = max(res, dp[i]);
13                } else {
14                    dp[i] = 0;
15                }
16                pre = t;
17            }
18        }
19        return res * res;
20    }
21 };

```

222. 求完全二叉树的节点个数

Given a complete binary tree, count the number of nodes.

Definition of a complete binary tree from Wikipedia:

In a complete binary tree every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible. It can have between 1 and 2^h nodes inclusive at the last level h .

完美二叉树一定是完全二叉树，而完全二叉树不一定是完美二叉树。那么这道题给的完全二叉树就有可能是完美二叉树，若是完美二叉树，节点个数很好求，为 $2^h - 1$ ， h 为该完美二叉树的高度。这道题可以用递归和非递归两种方法来解。我们先来看递归的方法，思路是分别找出以当前节点为根节点的左子树和右子树的高度并对比，如果相等，则说明是满二叉树，直接返回节点个数，如果不相等，则节点个数为左子树的节点个数加上右子树的节点个数再加1(根节点)，其中左右子树节点个数的计算可以使用递归来计算，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     int countNodes(TreeNode* root) {
4         int hLeft = 0, hRight = 0;
5         TreeNode *pLeft = root, *pRight = root;
6         while (pLeft) {
7             ++hLeft;
8             pLeft = pLeft->left;
9         }
10        while (pRight) {
11            ++hRight;
12            pRight = pRight->right;
13        }
14        if (hLeft == hRight) return pow(2, hLeft) - 1;
15        return countNodes(root->left) + countNodes(root->right) + 1;
16    }
17};
```

CPP

递归的解法还有一种解法如下所示：

解法2：

```
1 class Solution {
2 public:
3     int countNodes(TreeNode* root) {
4         int hLeft = leftHeight(root);
5         int hRight = rightHeight(root);
6         if (hLeft == hRight) return pow(2, hLeft) - 1;
7         return countNodes(root->left) + countNodes(root->right) + 1;
8     }
9     int leftHeight(TreeNode* root) {
10        if (!root) return 0;
11        return 1 + leftHeight(root->left);
12    }
13    int rightHeight(TreeNode* root) {
14        if (!root) return 0;
15        return 1 + rightHeight(root->right);
16    }
17};
```

CPP

223. 矩形面积

Find the total area covered by two rectilinear rectangles in a 2D plane.

Each rectangle is defined by its bottom left corner and top right corner as shown in the figure.



Assume that the total area is never beyond the maximum possible value of int.

Credits:

Special thanks to @mithmatt for adding this problem, creating the above image and all test cases.

这道题不算一道很难的题，但是我还是花了很久才做出来，刚开始我尝试找出所以有重叠的情况，发现有很多种情况，很麻烦。后来换了一种思路，尝试先找出所有的不相交的情况，只有四种，一个矩形在另一个的上下左右四个位置不重叠，这四种情况下返回两个矩形面积之和。其他所有情况下两个矩形是有交集的，这时候我们只要算出长和宽，即可求出交集区域的大小，然后从两个巨型面积之和中减去交集面积就是最终答案。求交集区域的长和宽也不难，由于交集都是在中间，所以横边的左端点是两个矩形左顶点横坐标的较大值，右端点是两个矩形右顶点的较小值，同理，竖边的下端点是两个矩形下顶点纵坐标的较大值，上端点是两个矩形上顶点纵坐标的较小值。代码如下：

解法1:

```
1 class Solution {
2 public:
3     int computeArea(int A, int B, int C, int D, int E, int F, int G, int H) {
4         int sum = (C - A) * (D - B) + (H - F) * (G - E);
5         if (E >= C || F >= D || B >= H || A >= G) return sum;
6         return sum - ((min(G, C) - max(A, E)) * (min(D, H) - max(B, F)));
7     }
8 };
9 }
```

CPP

当然，这三行还可以丧心病狂地合成一行，那么LeetCode中我遇见的第一次一行解题的方法如下所示：

解法2:

```
1 class Solution {
2 public:
3     int computeArea(int A, int B, int C, int D, int E, int F, int G, int H) {
4         return (C - A) * (D - B) + (H - F) * (G - E) - (max((min(G, C) - max(A, E)), 0) *
5 max((min(D, H) - max(B, F)), 0));
6     }
7 };
8 }
```

CPP

224. 基本计算器

Implement a basic calculator to evaluate a simple expression string.

The expression string may contain open (and closing parentheses), the plus + or minus sign - , non-negative integers and empty spaces .

You may assume that the given expression is always valid.

Some examples:

"1 + 1" = 2
 " 2-1 + 2 " = 3
 "(1+(4+5+2)-3)+(6+8)" = 23

Note: Do not use the eval built-in library function.

这道题让我们实现一个基本的计算器来计算简单的算数表达式，而且题目限制了表达式中只有加减号，数字，括号和空格，没有乘除，那么就没啥计算的优先级之分了。于是这道题就变的没有那么复杂了。我们需要一个栈来辅助计算，用个变量sign来表示当前的符号，我们遍历给定的字符串s，如果遇到了数字，由于可能是个多位数，所以我们要用while循环把之后的数字都读进来，然后用sign*num来更新结果res；如果遇到了加号，则sign赋为1，如果遇到了减号，则赋为-1；如果遇到了左括号，则把当前结果res和符号sign压入栈，res重置为0，sign重置为1；如果遇到了右括号，结果res乘以栈顶的符号，栈顶元素出栈，结果res加上栈顶的数字，栈顶元素出栈。代码如下：

解法1：

```

1 class Solution {
2 public:
3     int calculate(string s) {
4         int res = 0, sign = 1, n = s.size();
5         stack<int> st;
6         for (int i = 0; i < n; ++i) {
7             char c = s[i];
8             if (c >= '0') {
9                 int num = 0;
10                while (i < n && s[i] >= '0') {
11                    num = 10 * num + s[i++]- '0';
12                }
13                res += sign * num;
14                --i;
15            } else if (c == '+') {
16                sign = 1;
17            } else if (c == '-') {
18                sign = -1;
19            } else if (c == '(') {
20                st.push(res);
21                st.push(sign);
22                res = 0;
23                sign = 1;
24            } else if (c == ')') {
25                res *= st.top(); st.pop();
26                res += st.top(); st.pop();
27            }
28        }
29        return res;
30    }
31 };

```

CPP

下面这种方法和上面的基本一样，只不过对于数字的处理略微不同，上面的方法是连续读入数字，而这种方法是使用了一个变量来保存读入的num，所以在遇到其他字符的时候，都要用sign*num来更新结果res，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int calculate(string s) {
4         int res = 0, num = 0, sign = 1, n = s.size();
5         stack<int> st;
6         for (int i = 0; i < n; ++i) {
7             char c = s[i];
8             if (c >= '0') {
9                 num = 10 * num + (c - '0');
10            } else if (c == '+' || c == '-') {
11                res += sign * num;
12                num = 0;
13                sign = (c == '+') ? 1 : -1;
14            } else if (c == '(') {
15                st.push(res);
16                st.push(sign);
17                res = 0;
18                sign = 1;
19            } else if (c == ')') {
20                res += sign * num;
21                num = 0;
22                res *= st.top(); st.pop();
23                res += st.top(); st.pop();
24            }
25        }
26        res += sign * num;
27        return res;
28    }
29};

```

在做了Basic Calculator III之后，再反过来这道题，发现递归处理括号的方法在这道题也同样适用，我们用一个变量cnt，遇到左括号自增1，遇到右括号自减1，当cnt为0的时候，说明括号正好完全匹配，这个trick在验证括号是否valid的时候经常使用到。然后我们就是根据左右括号的位置提取出中间的子字符串调用递归函数，返回值赋给num，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int calculate(string s) {
4         int res = 0, num = 0, sign = 1, n = s.size();
5         for (int i = 0; i < n; ++i) {
6             char c = s[i];
7             if (c >= '0' && c <= '9') {
8                 num = 10 * num + (c - '0');
9             } else if (c == '(') {
10                 int j = i, cnt = 0;
11                 for (; i < n; ++i) {
12                     if (s[i] == '(') ++cnt;
13                     if (s[i] == ')') --cnt;
14                     if (cnt == 0) break;
15                 }
16                 num = calculate(s.substr(j + 1, i - j - 1));
17             }
18             if (c == '+' || c == '-' || i == n - 1) {
19                 res += sign * num;
20                 num = 0;
21                 sign = (c == '+') ? 1 : -1;
22             }
23         }
24         return res;
25     }
26 };

```

225. 用队列来实现栈

Implement the following operations of a stack using queues.

`push(x)` -- Push element `x` onto stack.
`pop()` -- Removes the element on top of the stack.
`top()` -- Get the top element.
`empty()` -- Return whether the stack is empty.

Notes:

You may assume that all operations are valid (for example, no pop or top operations will be called on an empty stack).

Depending on your language, queue may not be supported natively. You may simulate a queue by using a list or deque (double-ended queue), as long as you use only standard operations of a queue -- which means only push to back, pop from front, size, and is empty operations are valid.

这道题让我们用队列来实现栈，队列和栈作为两种很重要的数据结构，它们最显著的区别就是，队列是先进先出，而栈是先进后出。题目要求中又给定了限制条件只能用queue的基本操作，像`back()`这样的操作是禁止使用的。那么怎么样才能让先进先出的特性模拟出先进后出呢，这里就需要另外一个队列来辅助操作，我们总共需要两个队列，其中一个队列用来放最后加进来的数，模拟栈顶元素。剩下所有的数都按顺序放入另一个队列中。当`push`操作时，将新数字先加入模拟栈顶元素的队列中，如果此时队列中有数字，则将原本有的数字放入另一个队列中，让新数字在这队中，用来模拟栈顶元素。当`top`操作时，如果模拟栈顶的队列中有数字则直接返回，如果没有则到另一个队列中通过平移数字取出最后一个数字加入模拟栈顶的队列中。当`pop`操作时，先执行`top()`操作，保证模拟栈顶的队列中有数字，然后再将该数字移除即可。当`empty`操作时，当两个队列都为空时，栈为空。代码如下：

解法1：

```

1 class Stack {
2     public:
3         // Push element x onto stack.
4         void push(int x) {
5             q2.push(x);
6             while (q2.size() > 1) {
7                 q1.push(q2.front());
8                 q2.pop();
9             }
10        }
11
12        // Removes the element on top of the stack.
13        void pop(void) {
14            top();
15            q2.pop();
16        }
17
18        // Get the top element.
19        int top(void) {
20            if (q2.empty()) {
21                for (int i = 0; i < (int)q1.size() - 1; ++i) {
22                    q1.push(q1.front());
23                    q1.pop();
24                }
25                q2.push(q1.front());
26                q1.pop();
27            }
28            return q2.front();
29        }
30
31        // Return whether the stack is empty.
32        bool empty(void) {
33            return q1.empty() && q2.empty();
34        }
35
36    private:
37        queue<int> q1, q2;
38    };

```

这道题还有另一种解法，可以参见另一道类似的题Implement Queue using Stacks 用栈来实现队列，我个人来讲比较偏爱下面这种方法，比较好记，只要实现对了push函数，后面三个直接调用队列的函数即可。这种方法的原理就是每次把新加入的数据插入到前头，这样队列保存的顺序和栈的顺序是相反的，它们的取出方式也是反的，那么反过来，就是我们需要的顺序了。我们需要一个辅助队列tmp，把s的元素也逆着顺序存入tmp中，此时加入新元素x，再把tmp中的元素存回来，这样就是我们要的顺序了，其他三个操作也就直接调用队列的操作即可，参见代码如下：

解法2：

```

1 class Stack {
2 public:
3     // Push element x onto stack.
4     void push(int x) {
5         queue<int> tmp;
6         while (!q.empty()) {
7             tmp.push(q.front());
8             q.pop();
9         }
10        q.push(x);
11        while (!tmp.empty()) {
12            q.push(tmp.front());
13            tmp.pop();
14        }
15    }
16
17    // Removes the element on top of the stack.
18    void pop() {
19        q.pop();
20    }
21
22    // Get the top element.
23    int top() {
24        return q.front();
25    }
26
27    // Return whether the stack is empty.
28    bool empty() {
29        return q.empty();
30    }
31
32 private:
33     queue<int> q;
34 };

```

226. 翻转二叉树

这道题让我们翻转二叉树，是树的基本操作之一，不算难题。最下面那句话实在有些木有节操啊，不知道是Google说给谁的。反正这道题确实难度不大，可以用递归和非递归两种方法来解。先来看递归的方法，写法非常简洁，五行代码搞定，交换当前左右节点，并直接调用递归即可，代码如下：

解法1：

```

1 class Solution {
2 public:
3     TreeNode* invertTree(TreeNode* root) {
4         if (!root) return NULL;
5         TreeNode *tmp = root->left;
6         root->left = invertTree(root->right);
7         root->right = invertTree(tmp);
8         return root;
9     }
10 };

```

非递归的方法也不复杂，跟二叉树的层序遍历一样，需要用queue来辅助，先把根节点排入队列中，然后从队中取出来，交换其左右节点，如果存在则分别将左右节点在排入队列中，以此类推直到队列中木有节点了停止循环，返回root即可。代码如下：

解法2：

```

1 class Solution {
2 public:
3     TreeNode* invertTree(TreeNode* root) {
4         if (!root) return NULL;
5         queue<TreeNode*> q;
6         q.push(root);
7         while (!q.empty()) {
8             TreeNode *node = q.front(); q.pop();
9             TreeNode *tmp = node->left;
10            node->left = node->right;
11            node->right = tmp;
12            if (node->left) q.push(node->left);
13            if (node->right) q.push(node->right);
14        }
15        return root;
16    }
17 };

```

CPP

227. 基本计算器之二

Implement a basic calculator to evaluate a simple expression string.

The expression string contains only non-negative integers, +, -, *, / operators and empty spaces . The integer division should truncate toward zero.

You may assume that the given expression is always valid.

Some examples:

"3+2*2" = 7
" 3/2 " = 1
" 3+5 / 2 " = 5

Note: Do not use the eval built-in library function.

Credits:

Special thanks to @ts for adding this problem and creating all test cases.

这道题是之前那道Basic Calculator 基本计算器的拓展，不同之处在于那道题的计算符号只有加和减，而这题加上了乘除，那么就牵扯到了运算优先级的问题，好在这道题去掉了括号，还适当的降低了难度，估计再出一道的话就该加上括号了。不管那么多，这道题先按木有有括号来处理，由于存在运算优先级，我们采取的措施是使用一个栈保存数字，如果该数字之前的符号是加或减，那么把当前数字压入栈中，注意如果是减号，则加入当前数字的相反数，因为减法相当于加上一个相反数。如果之前的符号是乘或除，那么从栈顶取出一个数字和当前数字进行乘或除的运算，再把结果压入栈中，那么完成一遍遍历后，所有的乘或除都运算完了，再把栈中所有的数字都加起来就是最终结果了。代码如下：

解法1：

```
1 class Solution {
2 public:
3     int calculate(string s) {
4         int res = 0, num = 0, n = s.size();
5         char op = '+';
6         stack<int> st;
7         for (int i = 0; i < n; ++i) {
8             if (s[i] >= '0') {
9                 num = num * 10 + s[i] - '0';
10            }
11            if ((s[i] < '0' && s[i] != ' ') || i == n - 1) {
12                if (op == '+') st.push(num);
13                if (op == '-') st.push(-num);
14                if (op == '*' || op == '/') {
15                    int tmp = (op == '*') ? st.top() * num : st.top() / num;
16                    st.pop();
17                    st.push(tmp);
18                }
19                op = s[i];
20                num = 0;
21            }
22        }
23        while (!st.empty()) {
24            res += st.top();
25            st.pop();
26        }
27        return res;
28    }
29};
```

在做了Basic Calculator III之后，再反过来头来看这道题，发现只要将处理括号的部分去掉直接就可以在这道题上使用，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int calculate(string s) {
4         int res = 0, curRes = 0, num = 0, n = s.size();
5         char op = '+';
6         for (int i = 0; i < n; ++i) {
7             char c = s[i];
8             if (c >= '0' && c <= '9') {
9                 num = num * 10 + c - '0';
10            }
11            if (c == '+' || c == '-' || c == '*' || c == '/' || i == n - 1) {
12                switch (op) {
13                    case '+': curRes += num; break;
14                    case '-': curRes -= num; break;
15                    case '*': curRes *= num; break;
16                    case '/': curRes /= num; break;
17                }
18                if (c == '+' || c == '-' || i == n - 1) {
19                    res += curRes;
20                    curRes = 0;
21                }
22                op = c;
23                num = 0;
24            }
25        }
26        return res;
27    }
28};

```

228. 总结区间

Given a sorted integer array without duplicates, return the summary of its ranges.

For example, given [0,1,2,4,5,7], return ["0->2","4->5","7"].

这道题给定我们一个有序数组，让我们总结区间，具体来说就是让我们找出连续的序列，然后首尾两个数字之间用个“→”来连接，那么我只需遍历一遍数组即可，每次检查下一个数是不是递增的，如果是，则继续往下遍历，如果不是了，我们还要判断此时是一个数还是一个序列，一个数直接存入结果，序列的话要存入首尾数字和箭头“→”。我们需要两个变量i和j，其中i是连续序列起始数字的位置，j是连续数列的长度，当j为1时，说明只有一个数字，若大于1，则是一个连续序列，代码如下：

```

1 class Solution {
2 public:
3     vector<string> summaryRanges(vector<int>& nums) {
4         vector<string> res;
5         int i = 0, n = nums.size();
6         while (i < n) {
7             int j = 1;
8             while (i + j < n && nums[i + j] - nums[i] == j) ++j;
9             res.push_back(j <= 1 ? to_string(nums[i]) : to_string(nums[i]) + "->" +
10               to_string(nums[i + j - 1]));
11             i += j;
12         }
13         return res;
14     }
15 };

```

229. 求众数之二

Given an integer array of size n , find all elements that appear more than $\lfloor n/3 \rfloor$ times. The algorithm should run in linear time and in $O(1)$ space.

Hint:

How many majority elements could it possibly have?

Do you have a better hint? Suggest it!

这道题让我们求出现次数大于 $n/3$ 的众数，而且限定了时间和空间复杂度，那么就不能排序，也不能使用哈希表，这么苛刻的限制条件只有一种方法能解了，那就是摩尔投票法 Moore Voting，这种方法在之前那道题Majority Element 求众数中也使用了。题目中给了一条很重要的提示，让我们先考虑可能会有多少个众数，经过举了很多例子分析得出，任意一个数组出现次数大于 $n/3$ 的众数最多有两个，具体的证明我就不会了，我也不是数学专业的。那么有了这个信息，我们使用投票法的核心是找出两个候选众数进行投票，需要两遍遍历，第一遍历找出两个候选众数，第二遍遍历重新投票验证这两个候选众数是否为众数即可，选候选众数方法和前面那篇Majority Element 求众数一样，由于之前那题题目中限定了一定会有众数存在，故而省略了验证候选众数的步骤，这道题却没有这种限定，即满足要求的众数可能存在，所以要有验证。代码如下：

```

1 class Solution {
2 public:
3     vector<int> majorityElement(vector<int>& nums) {
4         vector<int> res;
5         int m = 0, n = 0, cm = 0, cn = 0;
6         for (auto &a : nums) {
7             if (a == m) ++cm;
8             else if (a == n) ++cn;
9             else if (cm == 0) m = a, cm = 1;
10            else if (cn == 0) n = a, cn = 1;
11            else --cm, --cn;
12        }
13        cm = cn = 0;
14        for (auto &a : nums) {
15            if (a == m) ++cm;
16            else if (a == n) ++cn;
17        }
18        if (cm > nums.size() / 3) res.push_back(m);
19        if (cn > nums.size() / 3) res.push_back(n);
20        return res;
21    }
22 };

```

CPP

230. 二叉搜索树中的第K小的元素

Given a binary search tree, write a function `kthSmallest` to find the kth smallest element in it.

Note:

You may assume k is always valid, $1 \leq k \leq \text{BST's total elements}$.

Follow up:

What if the BST is modified (insert/delete operations) often and you need to find the kth smallest frequently? How would you optimize the `kthSmallest` routine?

Hint:

Try to utilize the property of a BST.

What if you could modify the BST node's structure?

The optimal runtime complexity is $O(\text{height of BST})$.

这又是一道关于二叉搜索树 Binary Search Tree 的题，LeetCode 中关于 BST 的题有 Validate Binary Search Tree 验证二叉搜索树，Recover Binary Search Tree 复原二叉搜索树，Binary Search Tree Iterator 二叉搜索树迭代器，Unique Binary Search Trees 独一无二的二叉搜索树，Unique Binary Search Trees II 独一无二的二叉搜索树之二，Convert Sorted Array to Binary Search Tree 将有序数组转为二叉搜索树 和 Convert Sorted List to Binary Search Tree 将有序链表转为二叉搜索树。

那么这道题给的提示是让我们用 BST 的性质来解题，最重要的性质是就是左 < 根 < 右，那么如果用中序遍历所有的节点就会得到一个有序数组。所以解题的关键还是中序遍历啊。关于二叉树的中序遍历可以参见我之前的博客 Binary Tree Inorder Traversal 二叉树的中序遍历，里面有很多种方法可以用，我们先来看一种非递归的方法，中序遍历最先遍历到的是最小的结点，那么我们只要用一个计数器，每遍历一个结点，计数器自增 1，当计数器到达 k 时，返回当前结点值即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int kthSmallest(TreeNode* root, int k) {
4         int cnt = 0;
5         stack<TreeNode*> s;
6         TreeNode *p = root;
7         while (p || !s.empty()) {
8             while (p) {
9                 s.push(p);
10                p = p->left;
11            }
12            p = s.top(); s.pop();
13            ++cnt;
14            if (cnt == k) return p->val;
15            p = p->right;
16        }
17        return 0;
18    }
19 };

```

CPP

当然，此题我们也可以用递归来解，还是利用中序遍历来解，代码如下：

解法2：

```

1 class Solution {
2 public:
3     int kthSmallest(TreeNode* root, int k) {
4         return kthSmallestDFS(root, k);
5     }
6     int kthSmallestDFS(TreeNode* root, int &k) {
7         if (!root) return -1;
8         int val = kthSmallestDFS(root->left, k);
9         if (k == 0) return val;
10        if (--k == 0) return root->val;
11        return kthSmallestDFS(root->right, k);
12    }
13 };

```

再来看一种分治法的思路，由于BST的性质，我们可以快速定位出第k小的元素是在左子树还是右子树，我们首先计算出左子树的结点个数总和cnt，如果k小于等于左子树结点总和cnt，说明第k小的元素在左子树中，直接对左子结点调用递归即可。如果k大于cnt+1，说明目标值在右子树中，对右子结点调用递归函数，注意此时的k应为k-cnt-1，应为已经减少了cnt+1个结点。如果k正好等于cnt+1，说明当前结点即为所求，返回当前结点值即可，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int kthSmallest(TreeNode* root, int k) {
4         int cnt = count(root->left);
5         if (k <= cnt) {
6             return kthSmallest(root->left, k);
7         } else if (k > cnt + 1) {
8             return kthSmallest(root->right, k - cnt - 1);
9         }
10        return root->val;
11    }
12    int count(TreeNode* node) {
13        if (!node) return 0;
14        return 1 + count(node->left) + count(node->right);
15    }
16 };

```

这道题的Follow up中说假设该BST被修改的很频繁，而且查找第k小元素的操作也很频繁，问我们如何优化。其实最好的方法还是像上面的解法那样利用分治法来快速定位目标所在的位置，但是每个递归都遍历左子树所有结点来计算个数的操作并不高效，所以我们应该修改原树结点的结构，使其保存包括当前结点和其左右子树所有结点的个数，这样我们使用的时候就可以快速得到任何左子树结点总数来帮我们快速定位目标值了。定义了新结点结构体，然后就要生成新树，还是用递归的方法生成新树，注意生成的结点的count值要累加其左右子结点的count值。然后在求第k小元素的函数中，我们先生成新的树，然后调用递归函数。在递归函数中，不能直接访问左子结点的count值，因为左子节结点不一定存在，所以我们先判断，如果左子结点存在的话，那么跟上面解法的操作相同。如果不存在的话，当此时k为1的时候，直接返回当前结点值，否则就对右子结点调用递归函数，k自减1，参见代码如下：

解法4：

```

1 class Solution {
2 public:
3     struct MyTreeNode {
4         int val;
5         int count;
6         MyTreeNode *left;
7         MyTreeNode *right;
8         MyTreeNode(int x) : val(x), count(1), left(NULL), right(NULL) {}
9     };
10
11    MyTreeNode* build(TreeNode* root) {
12        if (!root) return NULL;
13        MyTreeNode *node = new MyTreeNode(root->val);
14        node->left = build(root->left);
15        node->right = build(root->right);
16        if (node->left) node->count += node->left->count;
17        if (node->right) node->count += node->right->count;
18        return node;
19    }
20
21    int kthSmallest(TreeNode* root, int k) {
22        MyTreeNode *node = build(root);
23        return helper(node, k);
24    }
25
26    int helper(MyTreeNode* node, int k) {
27        if (node->left) {
28            int cnt = node->left->count;
29            if (k <= cnt) {
30                return helper(node->left, k);
31            } else if (k > cnt + 1) {
32                return helper(node->right, k - 1 - cnt);
33            }
34            return node->val;
35        } else {
36            if (k == 1) return node->val;
37            return helper(node->right, k - 1);
38        }
39    }
40};

```

231. 判断2的次方数

Given an integer, write a function to determine if it is a power of two.

Example 1:

Input: 1
Output: true
Example 2:

Input: 16
Output: true

这道题让我们判断一个数是否为2的次方数，而且要求时间和空间复杂度都为常数，那么对于这种玩数字的题，我们应该首先考虑位操作 Bit Operation。在LeetCode中，位操作的题有很多，比如比如 Repeated DNA Sequences, Single Number, Single Number II, Grey Code, Reverse Bits, Bitwise AND of Numbers Range, Number of 1 Bits 和 Divide Two Integers 等等。那么我们来观察下2的次方数的二进制写法的特点：

1	2	4	8	16
1	10	100	1000	10000

那么我们很容易看出来2的次方数都只有一个1，剩下的都是0，所以我们的解题思路就有了，我们只要每次判断最低位是否为1，然后向右移位，最后统计1的个数即可判断是否是2的次方数，代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool isPowerOfTwo(int n) {
4         int cnt = 0;
5         while (n > 0) {
6             cnt += (n & 1);
7             n >>= 1;
8         }
9         return cnt == 1;
10    }
11 };

```

CPP

这道题还有一个技巧，如果一个数是2的次方数的话，根据上面分析，那么它的二进数必然是最高位为1，其它都为0，那么如果此时我们减1的话，则最高位会降一位，其余为0的位现在都变为1，那么我们把两数相与，就会得到0，用这个性质也能来解题，而且只需一行代码就可以搞定，如下所示：

解法2：

```

1 class Solution {
2 public:
3     bool isPowerOfTwo(int n) {
4         return (n > 0) && (!(n & (n - 1)));
5     }
6 };

```

CPP

232. 用栈来实现队列

Implement the following operations of a queue using stacks.

push(x) -- Push element x to the back of queue.
pop() -- Removes the element from in front of queue.
peek() -- Get the front element.
empty() -- Return whether the queue is empty.

Notes:

You must use only standard operations of a stack -- which means only push to top, peek/pop from top, size, and is empty operations are valid.

Depending on your language, stack may not be supported natively. You may simulate a stack by using a list or deque (double-ended queue), as long as you use only standard operations of a stack.

You may assume that all operations are valid (for example, no pop or peek operations will be called on an empty queue).

这道题让我们用栈来实现队列，之前我们做过一道相反的题目Implement Stack using Queues 用队列来实现栈，是用队列来实现栈。这道题颠倒了个顺序，起始并没有太大的区别，栈和队列的核心不同点就是栈是先进后出，而队列是先进先出，那么我们要用栈的先进后出的特性来模拟出队列的先进先出。那么怎么做呢，其实很简单，只要我们在插入元素的时候每次都从前面插入即可，比如如果一个队列是1,2,3,4，那么我们在栈中保存为4,3,2,1，那么返回栈顶元素1，也就是队列的首元素，则问题迎刃而解。所以此题的难度是push函数，我们需要一个辅助栈tmp，把s的元素也逆着顺序存入tmp中，此时加入新元素x，再把tmp中的元素存回来，这样就是我们要的顺序了，其他三个操作也就直接调用栈的操作即可，参见代码如下：

解法1：

```

1  class Queue {
2  public:
3      // Push element x to the back of queue.
4      void push(int x) {
5          stack<int> tmp;
6          while (!s.empty()) {
7              tmp.push(s.top());
8              s.pop();
9          }
10         s.push(x);
11         while (!tmp.empty()) {
12             s.push(tmp.top());
13             tmp.pop();
14         }
15     }
16
17     // Removes the element from in front of queue.
18     void pop(void) {
19         s.pop();
20     }
21
22     // Get the front element.
23     int peek(void) {
24         return s.top();
25     }
26
27     // Return whether the queue is empty.
28     bool empty(void) {
29         return s.empty();
30     }
31
32 private:
33     stack<int> s;
34 };

```

上面那个解法虽然简单，但是效率不高，因为每次在push的时候，都要翻转两边栈，下面这个方法使用了两个栈_new和_old，其中新进栈的都先缓存在_new中，入队要pop和peek的时候，才将_new中所有元素移到_old中操作，提高了效率，代码如下：

解法2：

```

1  class Queue {
2  public:
3      // Push element x to the back of queue.
4      void push(int x) {
5          _new.push(x);
6      }
7
8      void shiftStack() {
9          if (_old.empty()) {
10             while (!_new.empty()) {
11                 _old.push(_new.top());
12                 _new.pop();
13             }
14         }
15     }
16
17     // Removes the element from in front of queue.
18     void pop(void) {
19         shiftStack();
20         if (!_old.empty()) _old.pop();
21     }
22
23     // Get the front element.
24     int peek(void) {
25         shiftStack();
26         if (!_old.empty()) return _old.top();
27         return 0;
28     }
29
30     // Return whether the queue is empty.
31     bool empty(void) {
32         return _old.empty() && _new.empty();
33     }
34
35 private:
36     stack<int> _old, _new;
37 };

```

233. 数字1的个数

Given an integer n, count the total number of digit 1 appearing in all non-negative integers less than or equal to n.

For example:

Given n = 13,

Return 6, because digit 1 occurred in the following numbers: 1, 10, 11, 12, 13.

Hint:

Beware of overflow.

这道题让我们比给定数小的所有数中1出现的个数，之前有道类似的题Number of 1 Bits 位1的个数，那道题是求转为二进数后1的个数，我开始以为这道题也是要用那题的方法，其实不是的，这题实际上相当于一道找规律的题。那么为了找出规律，我们就先来列举下所有含1的数字，并每10个统计下个数，如下所示：

1的个数 数字范围	含1的数字
1 [1, 9]	1
11 [10, 19]	10 11 12 13 14 15 16 17 18 19
1 [20, 29]	21
1 [30, 39]	31
1 [40, 49]	41
1 [50, 59]	51
1 [60, 69]	61
1 [70, 79]	71
1 [80, 89]	81
1 [90, 99]	91
11 [100, 109]	100 101 102 103 104 105 106 107 108 109
21 [110, 119]	110 111 112 113 114 115 116 117 118 119
11 [120, 129]	120 121 122 123 124 125 126 127 128 129
...	...
...	

通过上面的列举我们可以发现，100以内的数字，除了10-19之间有11个‘1’之外，其余都只有1个。如果我们不考虑[10, 19]区间上那多出来的10个‘1’的话，那么我们在对任意一个两位数，十位数上的数字(加1)就代表1出现的个数，这时候我们再把多出的10个加上即可。比如56就有 $(5+1)+10=16$ 个。如何知道是否要加上多出的10个呢，我们就要看十位上的数字是否大于等于2，是的话就要加上多余的10个‘1’。那么我们就可以用 $(x+8)/10$ 来判断一个数是否大于等于2。对于三位数区间 [100, 199] 内的数也是一样，除了[110, 119]之间多出的10个数之外，共21个‘1’，其余的每10个数的区间都只有11个‘1’，所以 [100, 199] 内共有 $21 + 11 * 9 = 120$ 个‘1’。那么现在想想[0, 999]区间内‘1’的个数怎么求？根据前面的结果，[0, 99] 内共有20个，[100, 199] 内共有120个，而其他每100个数内‘1’的个数也应该符合之前的规律，即也是20个，那么总共就有 $120 + 20 * 9 = 300$ 个‘1’。那么还是可以用相同的方法来判断并累加1的个数，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     int countDigitOne(int n) {
4         int res = 0, a = 1, b = 1;
5         while (n > 0) {
6             res += (n + 8) / 10 * a + (n % 10 == 1) * b;
7             b += n % 10 * a;
8             a *= 10;
9             n /= 10;
10        }
11        return res;
12    }
13 };

```

CPP

解法2:

```

1 class Solution {
2 public:
3     int countDigitOne(int n) {
4         int res = 0;
5         for (long k = 1; k <= n; k *= 10) {
6             long r = n / k, m = n % k;
7             res += (r + 8) / 10 * k + (r % 10 == 1 ? m + 1 : 0);
8         }
9         return res;
10    }
11 };

```

CPP

234. 回文链表

Given a singly linked list, determine if it is a palindrome. Follow up: Could you do it in O(n) time and O(1) space?

这道题让我们判断一个链表是否为回文链表，LeetCode中关于回文串的题共有六道，除了这道，其他的五道为 Palindrome Number 验证回文数字， Validate Palindrome 验证回文字符串， Palindrome Partitioning 拆分回文串， Palindrome Partitioning II 拆分回文串之二 和 Longest Palindromic Substring 最长回文串。链表比字符串难的地方就在于不能通过坐标来直接访问，而只能从头开始遍历到某个位置。那么根据回文串的特点，我们需要比较对应位置的值是否相等，那么我们首先需要找到链表的中点，这个可以用快慢指针来实现，使用方法可以参见之前的两篇Convert Sorted List to Binary Search Tree 将有序链表转为二叉搜索树 和 Reorder List 链表重排序，我们使用快慢指针找中点的原理是fast和slow两个指针，每次快指针走两步，慢指针走一步，等快指针走完时，慢指针的位置就是中点。我们还需要用栈，每次慢指针走一步，都把值存入栈中，等到达中点时，链表的前半段都存入栈中了，由于栈的后进先出的性质，就可以和后半段链表按照回文对应的顺序比较了。代码如下：

解法1:

```

1 class Solution {
2 public:
3     bool isPalindrome(ListNode* head) {
4         if (!head || !head->next) return true;
5         ListNode *slow = head, *fast = head;
6         stack<int> s;
7         s.push(head->val);
8         while (fast->next && fast->next->next) {
9             slow = slow->next;
10            fast = fast->next->next;
11            s.push(slow->val);
12        }
13        if (!fast->next) s.pop();
14        while (slow->next) {
15            slow = slow->next;
16            int tmp = s.top(); s.pop();
17            if (tmp != slow->val) return false;
18        }
19        return true;
20    }
21 };

```

这道题的Follow Up让我们用O(1)的空间，那就是说我们不能使用stack了，那么如果代替stack的作用呢，用stack的目的是为了利用其后进先出的特点，好倒着取出前半段的元素。那么现在我们不用stack了，如何倒着取元素呢。我们可以在找到中点后，将后半段的链表翻转一下，这样我们就可以按照回文的顺序比较了，参见代码如下：

解法2：

```

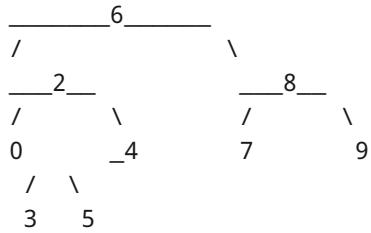
1 class Solution {
2 public:
3     bool isPalindrome(ListNode* head) {
4         if (!head || !head->next) return true;
5         ListNode *slow = head, *fast = head;
6         while (fast->next && fast->next->next) {
7             slow = slow->next;
8             fast = fast->next->next;
9         }
10        ListNode *last = slow->next, *pre = head;
11        while (last->next) {
12            ListNode *tmp = last->next;
13            last->next = tmp->next;
14            tmp->next = slow->next;
15            slow->next = tmp;
16        }
17        while (slow->next) {
18            slow = slow->next;
19            if (pre->val != slow->val) return false;
20            pre = pre->next;
21        }
22        return true;
23    }
24 };

```

235. 二叉搜索树的最小共同父节点

Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes in the BST.

According to the definition of LCA on Wikipedia: "The lowest common ancestor is defined between two nodes v and w as the lowest node in T that has both v and w as descendants (where we allow a node to be a descendant of itself)."



For example, the lowest common ancestor (LCA) of nodes 2 and 8 is 6. Another example is LCA of nodes 2 and 4 is 2, since a node can be a descendant of itself according to the LCA definition.

这道题让我们求二叉搜索树的最小共同父节点, LeetCode中关于BST的题有Validate Binary Search Tree 验证二叉搜索树, Recover Binary Search Tree 复原二叉搜索树, Binary Search Tree Iterator 二叉搜索树迭代器, Unique Binary Search Trees 独一无二的二叉搜索树, Unique Binary Search Trees II 独一无二的二叉搜索树之二, Convert Sorted Array to Binary Search Tree 将有序数组转为二叉搜索树, Convert Sorted List to Binary Search Tree 将有序链表转为二叉搜索树 和 Kth Smallest Element in a BST 二叉搜索树中的第K小的元素。这道题我们可以用递归来求解, 我们首先来看题目中给的例子, 由于二叉搜索树的特点是左<根<右, 所以根节点的值一直都是中间值, 大于左子树的所有节点值, 小于右子树的所有节点值, 那么我们可以做如下的判断, 如果根节点的值大于p和q之间的较大值, 说明p和q都在左子树中, 那么此时我们就进入根节点的左子节点继续递归, 如果根节点小于p和q之间的较小值, 说明p和q都在右子树中, 那么此时我们就进入根节点的右子节点继续递归, 如果都不是, 则说明当前根节点就是最小共同父节点, 直接返回即可, 参见代码如下:

解法1:

```

1 class Solution {
2 public:
3     TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
4         if (!root) return NULL;
5         if (root->val > max(p->val, q->val))
6             return lowestCommonAncestor(root->left, p, q);
7         else if (root->val < min(p->val, q->val))
8             return lowestCommonAncestor(root->right, p, q);
9         else return root;
10    }
11 }
  
```

CPP

当然, 此题也有非递归的写法, 用个while循环来代替递归调用即可, 然后不停的更新当前的根节点, 也能实现同样的效果, 代码如下:

解法2:

```

1 class Solution {
2 public:
3     TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
4         while (true) {
5             if (root->val > max(p->val, q->val)) root = root->left;
6             else if (root->val < min(p->val, q->val)) root = root->right;
7             else break;
8         }
9         return root;
10    }
11 };

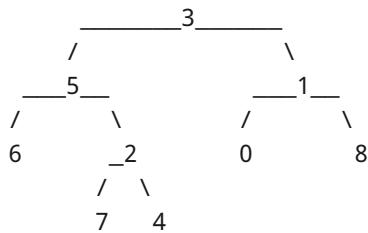
```

236. 二叉树的最小共同父节点

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

According to the definition of LCA on Wikipedia: "The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant of itself)."

Given the following binary tree: `root = [3,5,1,6,2,0,8,null,null,7,4]`



Example 1:

Input: `root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1`

Output: 3

Explanation: The LCA of of nodes 5 and 1 is 3.

Note:

All of the nodes' values will be unique.

p and q are different and both values will exist in the binary tree.

这道求二叉树的最小共同父节点的题是之前那道 Lowest Common Ancestor of a Binary Search Tree 的Follow Up。跟之前那题不同的地方是，这道题是普通是二叉树，不是二叉搜索树，所以就不能利用其特有的性质，所以我们只能在二叉树中来搜索 p 和 q ，然后从路径中找到最后一个相同的节点即为父节点，我们可以用递归来实现，在递归函数中，我们首先看当前结点是否为空，若为空则直接返回空，若为 p 或 q 中的任意一个，也直接返回当前结点。否则的话就对其左右子结点分别调用递归函数，由于这道题限制了 p 和 q 一定都在二叉树中存在，那么如果当前结点不等于 p 或 q ， p 和 q 要么分别位于左右子树中，要么同时位于左子树，或者同时位于右子树，那么我们分别来讨论：

若 p 和 q 要么分别位于左右子树中，那么对左右子结点调用递归函数，会分别返回 p 和 q 结点的位置，而当前结点正好就是 p 和 q 的最小共同父结点，直接返回当前结点即可，这就是题目中的例子1的情况。

若 p 和 q 同时位于左子树，这里有两种情况，一种情况是left会返回 p 和 q 中较高的那个位置，而right会返回空，所以我们最终返回非空的left即可，这就是题目中的例子2的情况。还有一种情况是会返回 p 和 q 的最小父结点，就是说当前结点的左子树中的某个结点才是 p 和 q 的最小父结点，会被返回。

若p和q同时位于右子树，同样这里有两种情况，一种情况是right会返回p和q中较高的那个位置，而left会返回空，所以我们最终返回非空的right即可，还有一种情况是会返回p和q的最小父结点，就是说当前结点的右子树中的某个结点才是p和q的最小父结点，会被返回，写法很简洁，代码如下：

解法1：

```
1 class Solution {
2 public:
3     TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
4         if (!root || p == root || q == root) return root;
5         TreeNode *left = lowestCommonAncestor(root->left, p, q);
6         TreeNode *right = lowestCommonAncestor(root->right, p, q);
7         if (left && right) return root;
8         return left ? left : right;
9     }
10};
```

上述代码可以进行优化一下，如果当前结点不为空，且既不是p也不是q，那么根据上面的分析，p和q的位置就有三种情况，p和q要么分别位于左右子树中，要么同时位于左子树，或者同时位于右子树。我们需要优化的情况就是当p和q同时位于左子树或右子树中，而且返回的结点并不是p或q，那么就是p和q的最小父结点了，已经求出来了，就不用再对右结点调用递归函数了，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
4         if (!root || p == root || q == root) return root;
5         TreeNode *left = lowestCommonAncestor(root->left, p, q);
6         if (left && left != p && left != q) return left;
7         TreeNode *right = lowestCommonAncestor(root->right, p, q);
8         if (left && right) return root;
9         return left ? left : right;
10    }
11};
```

此题还有一种情况，题目中没有明确说明p和q是否是树中的节点，如果不是，应该返回NULL，而上面的方法就不正确了，对于这种情况请参见 Cracking the Coding Interview 5th Edition 的第233-234页。

237. 删除链表的节点

Write a function to delete a node (except the tail) in a singly linked list, given only access to that node.

Supposed the linked list is 1 -> 2 -> 3 -> 4 and you are given the third node with value 3, the linked list should become 1 -> 2 -> 4 after calling your function.

这道题让我们删除链表的一个节点，更通常不同的是，没有给我们链表的起点，只给我们了一个要删的节点，跟我们以前遇到的情况不太一样，我们之前要删除一个节点的方法是要有其前一个节点的位置，然后将其前一个节点的next连向要删节点的下一个，然后delete掉要删的节点即可。这道题的处理方法是先把当前节点的值用下一个节点的值覆盖了，然后我们删除下一个节点即可，代码如下：

```

1 class Solution {
2 public:
3     void deleteNode(ListNode* node) {
4         node->val = node->next->val;
5         ListNode *tmp = node->next;
6         node->next = tmp->next;
7         delete tmp;
8     }
9 };

```

238. 除本身之外的数组之积

Given an array of n integers where $n > 1$, `nums`, return an array `output` such that `output[i]` is equal to the product of all the elements of `nums` except `nums[i]`.

Solve it without division and in $O(n)$.

For example, given [1,2,3,4], return [24,12,8,6].

Follow up:

Could you solve it with constant space complexity? (Note: The output array does not count as extra space for the purpose of space complexity analysis.)

这道题给定我们一个数组，让我们返回一个新数组，对于每一个位置上的数是其他位置上数的乘积，并且限定了时间复杂度 $O(n)$ ，并且不让我们用除法。如果让用除法的话，那这道题就应该属于Easy，因为可以先遍历一遍数组求出所有数字之积，然后除以对应位置的上的数字。但是这道题禁止我们使用除法，那么我们只能另辟蹊径。我们想，对于某一个数字，如果我们知道其前面所有数字的乘积，同时也知道后面所有的数乘积，那么二者相乘就是我们要的结果，所以我们只要分别创建出这两个数组即可，分别从数组的两个方向遍历就可以分别创建出乘积累积数组。参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     vector<int> productExceptSelf(vector<int>& nums) {
4         int n = nums.size();
5         vector<int> fwd(n, 1), bwd(n, 1), res(n);
6         for (int i = 0; i < n - 1; ++i) {
7             fwd[i + 1] = fwd[i] * nums[i];
8         }
9         for (int i = n - 1; i > 0; --i) {
10             bwd[i - 1] = bwd[i] * nums[i];
11         }
12         for (int i = 0; i < n; ++i) {
13             res[i] = fwd[i] * bwd[i];
14         }
15         return res;
16     }
17 };

```

我们可以对上面的方法进行空间上的优化，由于最终的结果都是要乘到结果`res`中，所以我们可以不用单独的数组来保存乘积，而是直接累积到`res`中，我们先从前面遍历一遍，将乘积的累积存入`res`中，然后从后面开始遍历，用到一个临时变量`right`，初始化为1，然后每次不断累积，最终得到正确结果，参见代码如下：

解法2:

```

1 class Solution {
2 public:
3     vector<int> productExceptSelf(vector<int>& nums) {
4         vector<int> res(nums.size(), 1);
5         for (int i = 1; i < nums.size(); ++i) {
6             res[i] = res[i - 1] * nums[i - 1];
7         }
8         int right = 1;
9         for (int i = nums.size() - 1; i >= 0; --i) {
10            res[i] *= right;
11            right *= nums[i];
12        }
13     return res;
14 }
15 };

```

239. 滑动窗口最大值

Given an array `nums`, there is a sliding window of size `k` which is moving from the very left of the array to the very right. You can only see the `k` numbers in the window. Each time the sliding window moves right by one position.

For example,

Given `nums` = [1,3,-1,-3,5,3,6,7], and `k` = 3.

Window position	Max
[1 3 -1]	3
-3 5 3 6 7	
1 [3 -1 -3]	3
5 3 6 7	
1 3 [-1 -3 5]	5
3 6 7	
1 3 -1 [-3 5 3]	5
-3 5 3 6 7	
1 3 -1 -3 [5 3 6]	6
5 3 6 7	
1 3 -1 -3 5 [3 6 7]	7

Therefore, return the max sliding window as [3,3,5,5,6,7].

Note:

You may assume `k` is always valid, $1 \leq k \leq$ input array's size.

Follow up:

Could you solve it in linear time?

Hint:

How about using a data structure such as deque (double-ended queue)?

The queue size need not be the same as the window's size.

Remove redundant elements and the queue should store only elements that need to be considered.

这道题给定了一个数组，还给了一个窗口大小k，让我们每次向右滑动一个数字，每次返回窗口内的数字的最大值，而且要求我们代码的时间复杂度为O(n)。提示我们要用双向队列deque来解题，并提示我们窗口中只留下有用的值，没用的全移除掉。果然Hard的题目我就是不会做，网上看到了别人的解法才明白，解法又巧妙有简洁，膜拜啊。大概思路是用双向队列保存数字的下标，遍历整个数组，如果此时队列的首元素是i-k的话，表示此时窗口向右移了一步，则移除队首元素。然后比较队尾元素和将要进来的值，如果小的话就都移除，然后此时我们把队首元素加入结果中即可，参见代码如下：

```

1 class Solution {
2 public:
3     vector<int> maxSlidingWindow(vector<int>& nums, int k) {
4         vector<int> res;
5         deque<int> q;
6         for (int i = 0; i < nums.size(); ++i) {
7             if (!q.empty() && q.front() == i - k) q.pop_front();
8             while (!q.empty() && nums[q.back()] < nums[i]) q.pop_back();
9             q.push_back(i);
10            if (i >= k - 1) res.push_back(nums[q.front()]);
11        }
12        return res;
13    }
14 };

```

240. 搜索一个二维矩阵之二

Write an efficient algorithm that searches for a value in an $m \times n$ matrix. This matrix has the following properties:

Integers in each row are sorted in ascending from left to right.
 Integers in each column are sorted in ascending from top to bottom.
 For example,

Consider the following matrix:

```
[  
 [1, 4, 7, 11, 15],  
 [2, 5, 8, 12, 19],  
 [3, 6, 9, 16, 22],  
 [10, 13, 14, 17, 24],  
 [18, 21, 23, 26, 30]
```

Given target = 5, return true.

Given target = 20, return false.

突然发现LeetCode很喜欢从LintCode上盗题，这是逼我去刷LintCode的节奏么？！这道题让我们在一个二维数组中快速的搜索的一个数字，这个二维数组各行各列都是按递增顺序排列的，是之前那道Search a 2D Matrix 搜索一个二维矩阵的延伸，那道题的不同在于每行的第一个数字比上一行的最后一个数字大，是一个整体蛇形递增的数组。所以那道题可以将二维数组展开成一个一位数组用一次二查搜索。而这道题没法那么做，这道题有它自己的特点。如果我们观察题目中给的那个例子，我们可以发现有两个位置的数字很有特点，左下角和右上角的数。左下角的18，往上所有的数变小，往右所有数增加，那么我们就可以和目标数相比较，如果目标数大，就往右搜，如果目标数小，就往上搜。这样就可以判断目标数是否存在。当然我们也可以把起始数放在右上角，往左和下搜，停止条件设置正确就行。代码如下：

```

1 class Solution {
2 public:
3     bool searchMatrix(vector<vector<int>> &matrix, int target) {
4         if (matrix.empty() || matrix[0].empty()) return false;
5         if (target < matrix[0][0] || target > matrix.back().back()) return false;
6         int x = matrix.size() - 1, y = 0;
7         while (true) {
8             if (matrix[x][y] > target) --x;
9             else if (matrix[x][y] < target) ++y;
10            else return true;
11            if (x < 0 || y >= matrix[0].size()) break;
12        }
13        return false;
14    }
15 };

```

241. 添加括号的不同方式

Given a string of numbers and operators, return all possible results from computing all the different possible ways to group numbers and operators. The valid operators are +, - and *.

Example 1

Input: "2-1-1".

```

((2-1)-1) = 0
(2-(1-1)) = 2
Output: [0, 2]

```

这道题给我们一个可能含有加减乘的表达式，让我们在任意位置添加括号，求出所有可能表达式的不同值。这道题跟之前的那道Unique Binary Search Trees II 独一无二的二叉搜索树之二用的方法一样，用递归来解，划分左右子树，递归构造。

```

1 class Solution {
2 public:
3     vector<int> diffWaysToCompute(string input) {
4         vector<int> res;
5         for (int i = 0; i < input.size(); ++i) {
6             if (input[i] == '+' || input[i] == '-' || input[i] == '*') {
7                 vector<int> left = diffWaysToCompute(input.substr(0, i));
8                 vector<int> right = diffWaysToCompute(input.substr(i + 1));
9                 for (int j = 0; j < left.size(); ++j) {
10                     for (int k = 0; k < right.size(); ++k) {
11                         if (input[i] == '+') res.push_back(left[j] + right[k]);
12                         else if (input[i] == '-') res.push_back(left[j] - right[k]);
13                         else res.push_back(left[j] * right[k]);
14                     }
15                 }
16             }
17         }
18         if (res.empty()) res.push_back(atoi(input.c_str()));
19         return res;
20     }
21 };

```

242. 验证变位词

Given two strings s and t, write a function to determine if t is an anagram of s.

For example,

```
s = "anagram", t = "nagaram", return true.  
s = "rat", t = "car", return false.
```

Note:

You may assume the string contains only lowercase alphabets.

这不算一道难题，核心点就在于使用哈希表映射，我们还是用一个数组来代替哈希表，使用类似方法的题目有Minimum Window Substring 最小窗口子串，Isomorphic Strings 同构字符串，Longest Substring Without Repeating Characters 最长无重复子串 和 1.1 Unique Characters of a String 字符串中不同的字符。我们先判断两个字符串长度是否相同，不相同直接返回false。然后把s中所有的字符出现个数统计起来，存入一个大小为26的数组中，因为题目中限定了输入字符串为小写字母组成。然后我们再来统计t字符串，如果发现不匹配则返回false。参见代码如下：

```
1 class Solution {  
2     public:  
3         bool isAnagram(string s, string t) {  
4             if (s.size() != t.size()) return false;  
5             int m[26] = {0};  
6             for (int i = 0; i < s.size(); ++i) ++m[s[i] - 'a'];  
7             for (int i = 0; i < t.size(); ++i) {  
8                 if (--m[t[i] - 'a'] < 0) return false;  
9             }  
10            return true;  
11        }  
12    };
```

CPP

243. 最短单词距离

Given a list of words and two words word1 and word2, return the shortest distance between these two words in the list.

For example,

Assume that words = ["practice", "makes", "perfect", "coding", "makes"].

Given word1 = "coding", word2 = "practice", return 3.

Given word1 = "makes", word2 = "coding", return 1.

Note:

You may assume that word1 does not equal to word2, and word1 and word2 are both in the list.

这道题让我们给了我们一个单词数组，又给定了两个单词，让我们求这两个单词之间的最小距离，限定了两个单词不同，而且都在数组中。我最先想到的方法比较笨，我首先想的是要用哈希表来做，建立每个单词和其所有出现位置数组的映射，但是后来想想，反正建立映射也要遍历一遍数组，我们还不如直接遍历一遍数组，直接把两个给定单词所有出现的位置分别存到两个数组里，然后我们在对两个数组进行两两比较更新结果，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int shortestDistance(vector<string>& words, string word1, string word2) {
4         vector<int> idx1, idx2;
5         int res = INT_MAX;
6         for (int i = 0; i < words.size(); ++i) {
7             if (words[i] == word1) idx1.push_back(i);
8             else if (words[i] == word2) idx2.push_back(i);
9         }
10        for (int i = 0; i < idx1.size(); ++i) {
11            for (int j = 0; j < idx2.size(); ++j) {
12                res = min(res, abs(idx1[i] - idx2[j]));
13            }
14        }
15        return res;
16    }
17 };

```

上面的那种方法并不高效，我们其实需要遍历一次数组就可以了，我们用两个变量p1,p2初始化为-1，然后我们遍历数组，遇到单词1，就将其位置存在p1里，若遇到单词2，就将其位置存在p2里，如果此时p1, p2都不为-1了，那么我们更新结果，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int shortestDistance(vector<string>& words, string word1, string word2) {
4         int p1 = -1, p2 = -1, res = INT_MAX;
5         for (int i = 0; i < words.size(); ++i) {
6             if (words[i] == word1) p1 = i;
7             else if (words[i] == word2) p2 = i;
8             if (p1 != -1 && p2 != -1) res = min(res, abs(p1 - p2));
9         }
10        return res;
11    }
12 };

```

下面这种方法只用一个辅助变量idx，初始化为-1，然后遍历数组，如果遇到等于两个单词中的任意一个的单词，我们在看idx是否为-1，若不为-1，且指向的单词和当前遍历到的单词不同，我们更新结果，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int shortestDistance(vector<string>& words, string word1, string word2) {
4         int idx = -1, res = INT_MAX;
5         for (int i = 0; i < words.size(); ++i) {
6             if (words[i] == word1 || words[i] == word2) {
7                 if (idx != -1 && words[idx] != words[i]) {
8                     res = min(res, i - idx);
9                 }
10                idx = i;
11            }
12        }
13        return res;
14    }
15 };

```

244. 最短单词距离之二

This is a follow up of Shortest Word Distance. The only difference is now you are given the list of words and your method will be called repeatedly many times with different parameters. How would you optimize it?

Design a class which receives a list of words in the constructor, and implements a method that takes two words word1 and word2 and return the shortest distance between these two words in the list.

For example,

Assume that words = ["practice", "makes", "perfect", "coding", "makes"].

Given word1 = "coding", word2 = "practice", return 3.

Given word1 = "makes", word2 = "coding", return 1.

Note:

You may assume that word1 does not equal to word2, and word1 and word2 are both in the list.

这道题是之前那道Shortest Word Distance的拓展，不同的是这次我们需要多次调用求最短单词距离的函数，那么用之前那道题的解法二和三就非常不高效，而当时我们摒弃的解法一的思路却可以用到这里，我们用哈希表来建立每个单词和其所有出现的位置的映射，然后在找最短单词距离时，我们只需要取出该单词在哈希表中映射的位置数组进行两两比较即可，参见代码如下：

解法1：

```

1 class WordDistance {
2 public:
3     WordDistance(vector<string>& words) {
4         for (int i = 0; i < words.size(); ++i) {
5             m[words[i]].push_back(i);
6         }
7     }
8
9     int shortest(string word1, string word2) {
10        int res = INT_MAX;
11        for (int i = 0; i < m[word1].size(); ++i) {
12            for (int j = 0; j < m[word2].size(); ++j) {
13                res = min(res, abs(m[word1][i] - m[word2][j]));
14            }
15        }
16        return res;
17    }
18
19 private:
20     unordered_map<string, vector<int> > m;
21 };

```

我们可以优化上述的代码，使查询的复杂度由上面的O(MN)变为O(M+N)，其中M和N为两个单词的长度，我们需要两个指针i和j来指向位置数组中的某个位置，开始初始化都为0，然后比较位置数组中的数字，将较小的一个的指针向后移动一位，直至其中一个数组遍历完成即可，参见代码如下：

解法2：

```

1 class WordDistance {
2 public:
3     WordDistance(vector<string>& words) {
4         for (int i = 0; i < words.size(); ++i) {
5             m[words[i]].push_back(i);
6         }
7     }
8
9     int shortest(string word1, string word2) {
10        int i = 0, j = 0, res = INT_MAX;
11        while (i < m[word1].size() && j < m[word2].size()) {
12            res = min(res, abs(m[word1][i] - m[word2][j]));
13            m[word1][i] < m[word2][j] ? ++i : ++j;
14        }
15        return res;
16    }
17
18 private:
19     unordered_map<string, vector<int> > m;
20 };

```

245. 最短单词距离之三

This is a follow up of Shortest Word Distance. The only difference is now word1 could be the same as word2.

Given a list of words and two words word1 and word2, return the shortest distance between these two words in the list.

word1 and word2 may be the same and they represent two individual words in the list.

For example,

Assume that words = ["practice", "makes", "perfect", "coding", "makes"].

Given word1 = "makes", word2 = "coding", return 1.

Given word1 = "makes", word2 = "makes", return 3.

Note:

You may assume word1 and word2 are both in the list.

这道题还是让我们求最短单词距离，有了之前两道题Shortest Word Distance II和Shortest Word Distance的基础，就大大降低了题目本身的难度。这道题增加了一个条件，就是说两个单词可能会相同，所以在第一题中的解法的基础上做一些修改，我最先想的解法是基于第一题中的解法二，由于会有相同的单词的情况，那么p1和p2就会相同，这样结果就会变成0，显然不对，所以我们要对word1和word2是否相等的情况分开处理，如果相等了，由于p1和p2会相同，所以我们需要一个变量t来记录上一个位置，这样如果t不为-1，且和当前的p1不同，我们可以更新结果，如果word1和word2不等，那么还是按原来的方法做，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int shortestWordDistance(vector<string>& words, string word1, string word2) {
4         int p1 = -1, p2 = -1, res = INT_MAX;
5         for (int i = 0; i < words.size(); ++i) {
6             int t = p1;
7             if (words[i] == word1) p1 = i;
8             if (words[i] == word2) p2 = i;
9             if (p1 != -1 && p2 != -1) {
10                 if (word1 == word2 && t != -1 && t != p1) {
11                     res = min(res, abs(t - p1));
12                 } else if (p1 != p2) {
13                     res = min(res, abs(p1 - p2));
14                 }
15             }
16         }
17         return res;
18     }
19 }
```

CPP

上述代码其实可以优化一下，我们并不需要变量t来记录上一个位置，我们将p1初始化为数组长度，p2初始化为数组长度的相反数，然后当word1和word2相等的情况下，我们用p1来保存p2的结果，p2赋为当前位置i，这样我们就可以更新结果了，如果word1和word2不相等，则还跟原来的做法一样，这种思路真是挺巧妙的，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int shortestWordDistance(vector<string>& words, string word1, string word2) {
4         int p1 = words.size(), p2 = -words.size(), res = INT_MAX;
5         for (int i = 0; i < words.size(); ++i) {
6             if (words[i] == word1) p1 = word1 == word2 ? p2 : i;
7             if (words[i] == word2) p2 = i;
8             res = min(res, abs(p1 - p2));
9         }
10        return res;
11    }
12 };

```

我们再来看一种更进一步优化的方法，只用一个变量idx，这个idx的作用就相当于记录上一次的位置，当前idx不等-1时，说明当前i和idx不同，然后我们在word1和word2相同或者words[i]和words[idx]相同的情况下更新结果，最后别忘了将idx赋为i，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int shortestWordDistance(vector<string>& words, string word1, string word2) {
4         int idx = -1, res = INT_MAX;
5         for (int i = 0; i < words.size(); ++i) {
6             if (words[i] == word1 || words[i] == word2) {
7                 if (idx != -1 && (word1 == word2 || words[i] != words[idx])) {
8                     res = min(res, i - idx);
9                 }
10                idx = i;
11            }
12        }
13        return res;
14    }
15 };
16

```

246. 对称数

A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down).

Write a function to determine if a number is strobogrammatic. The number is represented as a string.

Example 1:

Input: "69"
Output: true
Example 2:

Input: "88"
Output: true

这道题定义了一种对称数，就是说一个数字旋转180度和原来一样，也就是倒过来看一样，比如609，倒过来还是609等等，满足这种条件的数字其实没有几个，只有0,1,8,6,9。这道题其实可以看做求回文数的一种特殊情况，我们还是用双指针来检测，那么首尾两个数字如果相等的话，那么只有它们是0,1,8中间的一个才行，如果它们不相等的话，必须一个是6一个是9，或者一个是9一个是6，其他所有情况均返回false，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     bool isStrobogrammatic(string num) {
4         int l = 0, r = num.size() - 1;
5         while (l <= r) {
6             if (num[l] == num[r]) {
7                 if (num[l] != '1' && num[l] != '0' && num[l] != '8') {
8                     return false;
9                 }
10            } else {
11                if ((num[l] != '6' || num[r] != '9') && (num[l] != '9' || num[r] != '6')) {
12                    return false;
13                }
14            }
15            ++l; --r;
16        }
17        return true;
18    }
19};
```

由于满足题意的数字不多，所以我们可以用哈希表来做，把所有符合题意的映射都存入哈希表中，然后双指针扫描，看对应位置的两个数字是否在哈希表里存在映射，若不存在，返回false，遍历完成返回true，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     bool isStrobogrammatic(string num) {
4         unordered_map<char, char> m {{'0', '0'}, {'1', '1'}, {'8', '8'}, {'6', '9'}, {'9', '6'}};
5         for (int i = 0; i <= num.size() / 2; ++i) {
6             if (m[num[i]] != num[num.size() - i - 1]) return false;
7         }
8         return true;
9     }
10};
```

247. 对称数之二

A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down).

Find all strobogrammatic numbers that are of length = n.

For example,

Given n = 2, return ["11", "69", "88", "96"].

Hint:

Try to use recursion and notice that it should recurse with n - 2 instead of n - 1.

这道题是之前那道Strobogrammatic Number的拓展，那道题让我们判断一个数是否是对称数，而这道题让我们找出长度为n的所有对称数，我们肯定不能一个数一个数的来判断，那样太不高效了，而且OJ肯定也不会答应。题目中给了提示说可以用递归来做，而且提示了递归调用n-2，而不是n-1。我们先来列举一下n为0,1,2,3,4的情况：

n = 0: none

n = 1: 0, 1, 8

n = 2: 11, 69, 88, 96

n = 3: 101, 609, 808, 906, 111, 619, 818, 916, 181, 689, 888, 986

n = 4: 1001, 6009, 8008, 9006, 1111, 6119, 8118, 9116, 1691, 6699, 8698, 9696, 1881, 6889, 8888, 9886, 1961, 6969, 8968, 9966

我们注意观察n=0和n=2，可以发现后者是在前者的基础上，每个数字的左右增加了[1 1], [6 9], [8 8], [9 6]，看n=1和n=3更加明显，在0的左右增加[1 1]，变成了101，在0的左右增加[6 9]，变成了609，在0的左右增加[8 8]，变成了808，在0的左右增加[9 6]，变成了906，然后在分别在1和8的左右两边加那四组数，我们实际上是从m=0层开始，一层一层往上加的，需要注意的是当加到了n层的时候，左右两边不能加[0 0]，因为0不能出现在两位数及多位数的开头，在中间递归的过程中，需要有在数字左右两边各加上0的那种情况，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<string> findStrobogrammatic(int n) {
4         return find(n, n);
5     }
6     vector<string> find(int m, int n) {
7         if (m == 0) return {""};
8         if (m == 1) return {"0", "1", "8"};
9         vector<string> t = find(m - 2, n), res;
10        for (auto a : t) {
11            if (m != n) res.push_back("0" + a + "0");
12            res.push_back("1" + a + "1");
13            res.push_back("6" + a + "9");
14            res.push_back("8" + a + "8");
15            res.push_back("9" + a + "6");
16        }
17        return res;
18    }
19}

```

CPP

这道题还有迭代的解法，感觉也相当的巧妙，需要从奇偶来考虑，奇数赋为0,1,8，偶数赋为空，如果是奇数，就从i=3开始搭建，因为计算i=3需要i=1，而我们已经初始化了i=1的情况，如果是偶数，我们从i=2开始搭建，我们也已经初始化了i=0的情况，所以我们可以用for循环来搭建到i=n的情况，思路和递归一样，写法不同而已，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<string> findStrobogrammatic(int n) {
4         vector<string> one{"0", "1", "8"}, two{}, res = two;
5         if (n % 2 == 1) res = one;
6         for (int i = (n % 2) + 2; i <= n; i += 2) {
7             vector<string> t;
8             for (auto a : res) {
9                 if (i != n) t.push_back("0" + a + "0");
10                t.push_back("1" + a + "1");
11                t.push_back("6" + a + "9");
12                t.push_back("8" + a + "8");
13                t.push_back("9" + a + "6");
14            }
15            res = t;
16        }
17        return res;
18    }
19 };

```

248. 对称数之三

A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down).

Write a function to count the total strobogrammatic numbers that exist in the range of low <= num <= high.

For example,

Given low = "50", high = "100", return 3. Because 69, 88, and 96 are three strobogrammatic numbers.

Note:

Because the range might be a large number, the low and high numbers are represented as string.

这道题是之前那两道Strobogrammatic Number II和Strobogrammatic Number的拓展，又增加了难度，让我们找到给定范围内的对称数的个数，我们当然不能一个一个的判断是不是对称数，我们也不能直接每个长度调用第二道中的方法，保存所有的对称数，然后再统计个数，这样OJ会提示内存超过允许的范围，所以我们的解法是基于第二道的基础上，不保存所有的结果，而是在递归中直接计数，根据之前的分析，需要初始化n=0和n=1的情况，然后在其基础上进行递归，递归的长度len从low到high之间遍历，然后我们看当前单词长度有没有达到len，如果达到了，我们首先要去掉开头是0的多位数，然后去掉长度和low相同但小于low的数，和长度和high相同但大于high的数，然后结果自增1，然后分别给当前单词左右加上那五对对称数，继续递归调用，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     int strobogrammaticInRange(string low, string high) {
4         int res = 0;
5         for (int i = low.size(); i <= high.size(); ++i) {
6             find(low, high, "", i, res);
7             find(low, high, "0", i, res);
8             find(low, high, "1", i, res);
9             find(low, high, "8", i, res);
10        }
11        return res;
12    }
13    void find(string low, string high, string path, int len, int &res) {
14        if (path.size() >= len) {
15            if (path.size() != len || (len != 1 && path[0] == '0')) return;
16            if ((len == low.size() && path.compare(low) < 0) || (len == high.size() &&
17 path.compare(high) > 0)) {
18                return;
19            }
20            ++res;
21        }
22        find(low, high, "0" + path + "0", len, res);
23        find(low, high, "1" + path + "1", len, res);
24        find(low, high, "6" + path + "9", len, res);
25        find(low, high, "8" + path + "8", len, res);
26        find(low, high, "9" + path + "6", len, res);
27    }
};
```

上述代码可以稍微优化一下，得到如下的代码：

解法2：

```

1 class Solution {
2 public:
3     int strobogrammaticInRange(string low, string high) {
4         int res = 0;
5         find(low, high, "", res);
6         find(low, high, "0", res);
7         find(low, high, "1", res);
8         find(low, high, "8", res);
9         return res;
10    }
11    void find(string low, string high, string w, int &res) {
12        if (w.size() >= low.size() && w.size() <= high.size()) {
13            if ((w.size() == low.size() && w.compare(low) < 0) || (w.size() == high.size())
14                && w.compare(high) > 0)) {
15                return;
16            }
17            if (!(w.size() > 1 && w[0] == '0')) ++res;
18        }
19        if (w.size() + 2 > high.size()) return;
20        find(low, high, "0" + w + "0", res);
21        find(low, high, "1" + w + "1", res);
22        find(low, high, "6" + w + "9", res);
23        find(low, high, "8" + w + "8", res);
24        find(low, high, "9" + w + "6", res);
25    }
};

```

249. 群组偏移字符串

Given a string, we can "shift" each of its letter to its successive letter, for example: "abc" -> "bcd". We can keep "shifting" which forms the sequence:

"abc" -> "bcd" -> ... -> "xyz"

Given a list of strings which contains only lowercase alphabets, group all strings that belong to the same shifting sequence.

For example, given: ["abc", "bcd", "acef", "xyz", "az", "ba", "a", "z"],
Return:

```
[
  ["abc", "bcd", "xyz"],
  ["az", "ba"],
  ["acef"],
  ["a", "z"]
]
```

Note: For the return value, each inner list's elements must follow the lexicographic order.

这道题让我们重组偏移字符串，所谓偏移字符串，就是一个字符串的每个字符按照字母顺序表偏移相同量得到的另一个字符串，两者互为偏移字符串，注意相同字符串是偏移字符串的一种特殊情况，因为偏移量为0。现在给了我们一堆长度不同的字符串，让我们把互为偏移字符串的归并到一起，我最开始想的是建立字符串和该长度的所有偏移字符串的映射，但是很明显的错误是相同长度的不一定都是偏移字符串，比如'ab'和'ba'，所以只能用哈希表来建立一个字符串和所有和此字符串是偏移字符串的集合之间的映射，由于题目要求结果是按字母顺序的，所以用multiset来保存结果，一来可以保存重复字符串，二来可以自动排序。然后我还写了一个判断两个字符串是否互为偏移字符串的函数，注意在比较两个字母距离时采用了加26，再对26取余的trick。

我们遍历给定字符串集，对于遍历到的字符串，我们再遍历哈希表，和每个关键字调用isShifted函数来比较，如果互为偏移字符串，则加入其对应的字符串集，并标记flag，最后遍历完哈希表，没有跟任何关键字互为偏移，那么就新建一个映射，最后要做的就是把multiset转换为vector即可，参见代码如下：

解法1：

CPP

```

1 class Solution {
2 public:
3     vector<vector<string>> groupStrings(vector<string>& strings) {
4         vector<vector<string>> res;
5         unordered_map<string, multiset<string>> m;
6         for (auto a : strings) {
7             bool b = false;
8             for (auto it = m.begin(); it != m.end(); ++it) {
9                 if (isShifted(it->first, a)) {
10                     it->second.insert(a);
11                     b = true;
12                 }
13             }
14             if (!b) m[a] = {a};
15         }
16         for (auto it = m.begin(); it != m.end(); ++it)
17             res.push_back(vector<string>(it->second.begin(), it->second.end()));
18         return res;
19     }
20     bool isShifted(string s1, string s2) {
21         if (s1.size() != s2.size()) return false;
22         int diff = (s1[0] + 26 - s2[0]) % 26;
23         for (int i = 1; i < s1.size(); ++i) {
24             if ((s1[i] + 26 - s2[i]) % 26 != diff) return false;
25         }
26         return true;
27     }
28 }
29 };

```

上面那个方法挺复杂的，其实有更好的方法，网友的智慧无穷啊，上面那个方法的不高效之处在于对于每个遍历到的字符串，都要和哈希表中所有的关键字都比较一次，而其实我们可以更加巧妙的利用偏移字符串的特点，那就是字符串的每个字母和首字符的相对距离都是相等的，比如abc和efg互为偏移，对于abc来说，b和a的距离是1，c和a的距离是2，对于efg来说，f和e的距离是1，g和e的距离是2。再来看一个例子，az和yx，z和a的距离是25，x和y的距离也是25(直接相减是-1，这就是要加26然后取余的原因)，那么这样的话，所有互为偏移的字符串都有个unique的距离差，我们根据这个来建立映射就可以很好的进行单词分组了，这个思路真实太赞了，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<vector<string>> groupStrings(vector<string>& strings) {
4         vector<vector<string>> res;
5         unordered_map<string, multiset<string>> m;
6         for (auto a : strings) {
7             string t = "";
8             for (char c : a) {
9                 t += to_string((c + 26 - a[0]) % 26) + ",";
10            }
11            m[t].insert(a);
12        }
13        for (auto it = m.begin(); it != m.end(); ++it) {
14            res.push_back(vector<string>(it->second.begin(), it->second.end()));
15        }
16        return res;
17    }
18 };

```

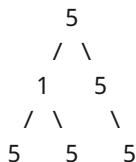
250. 计数相同值子树的个数

Given a binary tree, count the number of uni-value subtrees.

A Uni-value subtree means all nodes of the subtree have the same value.

For example:

Given binary tree,



return 4.

这道题让我们求相同值子树的个数，就是所有节点值都相同的子树的个数，之前有道求最大BST子树的题Largest BST Subtree，感觉挺像的，都是关于子树的问题，解题思路也可以参考一下，我们可以用递归来做，第一种解法的思路是先序遍历树的所有节点，然后对每一个节点调用判断以当前节点为根的字数的所有节点是否相同，判断方法可以参考之前那题Same Tree，用的是分治法的思想，分别对左右字数分别调用递归，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int res = 0;
4     int countUnivalSubtrees(TreeNode* root) {
5         if (!root) return res;
6         if (isUnival(root, root->val)) ++res;
7         countUnivalSubtrees(root->left);
8         countUnivalSubtrees(root->right);
9         return res;
10    }
11    bool isUnival(TreeNode *root, int val) {
12        if (!root) return true;
13        return root->val == val && isUnival(root->left, val) && isUnival(root->right, val);
14    }
15 };

```

但是上面的那种解法不是很高效，含有大量的重复check，我们想想能不能一次遍历就都搞定，我们这样想，符合条件的相同值的字数肯定是有叶节点的，而且叶节点也都相同(注意单独的一个叶节点也被看做是一个相同值子树)，那么我们可以从下往上check，采用后序遍历的顺序，左右根，我们还是递归调用函数，对于当前遍历到的节点，如果对其左右子节点分别递归调用函数，返回均为true的话，那么说明当前节点的值和左右子树的值都相同，那么又多了一棵树，所以结果自增1，然后返回当前节点值和给定值(其父节点值)是否相同，从而回归上一层递归调用。这里特别说明一下在子函数中要使用那个单竖杠或，为什么不用双竖杠的或，因为单竖杠的或是位或，就是说左右两部分都需要被计算，然后再或，C++这里将true当作1，false当作0，然后进行Bit OR 运算。不能使用双竖杠或的原因是，如果是双竖杠或，一旦左半边为true了，整个就直接是true了，右半边就不会再计算了，这样的话，一旦右子树中有值相同的子树也不会被计算到结果res中了，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int countUnivalSubtrees(TreeNode* root) {
4         int res = 0;
5         isUnival(root, -1, res);
6         return res;
7     }
8     bool isUnival(TreeNode* root, int val, int& res) {
9         if (!root) return true;
10        if (!isUnival(root->left, root->val, res) | !isUnival(root->right, root->val, res))
11    {
12        return false;
13    }
14    ++res;
15    return root->val == val;
16 }
17 };

```

我们还可以变一种写法，让递归函数直接返回以当前节点为根的相同值子树的个数，然后参数里维护一个引用类型的布尔变量，表示以当前节点为根的子树是否为相同值子树，我们首先对当前节点的左右子树分别调用递归函数，然后把结果加起来，我们现在要来看当前节点是不是和其左右子树节点值相同，当前我们首先要确认左右子节点的布尔型变量均为true，这样保证左右子节点分别都是相同值子树的根，然后我们看如果左子节点存在，那么左子节点值需要和当前节点值相同，如果右子节点存在，那么右子节点值要和当前节点值相同，若上述条件均满足的话，说明当前节点也是相同值子树的根节点，返回值再加1，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int countUnivalSubtrees(TreeNode* root) {
4         bool b = true;
5         return isUnival(root, b);
6     }
7     int isUnival(TreeNode *root, bool &b) {
8         if (!root) return 0;
9         bool l = true, r = true;
10        int res = isUnival(root->left, l) + isUnival(root->right, r);
11        b = l && r && (root->left ? root->val == root->left->val : true) && (root->right ?
12        root->val == root->right->val : true);
13        return res + b;
14    }
15};

```

上面三种都是令人看得头晕的递归写法，那么我们也来看一种迭代的写法，迭代写法是在后序遍历Binary Tree Postorder Traversal的基础上修改而来，我们需要用set来保存所有相同值子树的根节点，对于我们遍历到的节点，如果其左右子节点均不存在，那么此节点为叶节点，符合题意，加入结果set中，如果左子节点不存在，那么右子节点必须已经在结果set中，而且当前节点值需要和右子节点值相同才能将当前节点加入结果set中，同样的，如果右子节点不存在，那么左子节点必须已经存在set中，而且当前节点值要和左子节点值相同才能将当前节点加入结果set中。最后，如果左右子节点均存在，那么必须都已经在set中，并且左右子节点值都要和根节点值相同才能将当前节点加入结果set中，其余部分跟后序遍历的迭代写法一样，参见代码如下：

解法4:

```
1 class Solution {
2 public:
3     int countUnivalSubtrees(TreeNode* root) {
4         set<TreeNode*> res;
5         if (!root) return 0;
6         stack<TreeNode*> s;
7         s.push(root);
8         TreeNode *head = root;
9         while (!s.empty()) {
10             TreeNode *t = s.top();
11             if ((!t->left && !t->right) || t->left == head || t->right == head) {
12                 if (!t->left && !t->right) {
13                     res.insert(t);
14                 } else if (!t->left && res.find(t->right) != res.end() && t->right->val ==
15 t->val) {
16                     res.insert(t);
17                 } else if (!t->right && res.find(t->left) != res.end() && t->left->val ==
18 t->val) {
19                     res.insert(t);
20                 } else if (t->left && t->right && res.find(t->left) != res.end() &&
21 res.find(t->right) != res.end() && t->left->val == t->val && t->right->val == t->val) {
22                     res.insert(t);
23                 }
24                 s.pop();
25                 head = t;
26             } else {
27                 if (t->right) s.push(t->right);
28                 if (t->left) s.push(t->left);
29             }
30         }
31         return res.size();
32     }
33 };
```

251. 压平二维向量

Implement an iterator to flatten a 2d vector.

For example,

Given 2d vector =

```
[  
    [1,2],  
    [3],  
    [4,5,6]  
]
```

By calling next repeatedly until hasNext returns false, the order of elements returned by next should be: [1,2,3,4,5,6].

Hint:

How many variables do you need to keep track?

Two variables is all you need. Try with x and y.

Beware of empty rows. It could be the first few rows.

To write correct code, think about the invariant to maintain. What is it?

The invariant is x and y must always point to a valid point in the 2d vector. Should you maintain your invariant ahead of time or right when you need it?

Not sure? Think about how you would implement hasNext(). Which is more complex?

Common logic in two different places should be refactored into a common method.

Follow up:

As an added challenge, try to code it using only iterators in C++ or iterators in Java.

这道题让我们压平一个二维向量数组，并且实现一个iterator的功能，包括next和hasNext函数，那么最简单的方法就是将二维数组按顺序先存入到一个一维数组里，然后此时只要维护一个变量i来记录当前遍历到的位置，hasNext函数看当前坐标是否小于元素总数，next函数即为取出当前位置元素，坐标后移一位，参见代码如下：

解法1：

```
1 class Vector2D {  
2     public:  
3         Vector2D(vector<vector<int>>& vec2d) {  
4             for (auto a : vec2d) {  
5                 v.insert(v.end(), a.begin(), a.end());  
6             }  
7         }  
8         int next() {  
9             return v[i++];  
10        }  
11        bool hasNext() {  
12            return i < v.size();  
13        }  
14    private:  
15        vector<int> v;  
16        int i = 0;  
17    };
```

CPP

下面我们来看另一种解法，不直接转换为一维数组，而是维护两个变量x和y，我们将x和y初始化为0，对于hasNext函数，我们检查当前x是否小于总行数，y是否和当前行的列数相同，如果相同，说明要转到下一行，则x自增1，y初始化为0，若此时x还是小于总行数，说明下一个值可以被取出来，那么在next函数就可以直接取出行为x，列为y的数字，并将y自增1，参见代码如下：

解法2:

```

1 class Vector2D {
2 public:
3     Vector2D(vector<vector<int>>& vec2d) {
4         v = vec2d;
5         x = y = 0;
6     }
7     int next() {
8         return v[x][y++];
9     }
10    bool hasNext() {
11        while (x < v.size() && y == v[x].size()) {
12            ++x;
13            y = 0;
14        }
15        return x < v.size();
16    }
17 private:
18     vector<vector<int>> v;
19     int x, y;
20 };

```

CPP

题目中的Follow up让我们用interator来做，C++中iterator不像Java中的那么强大，自己本身并没有包含next和hasNext函数，所以我们得自己来实现，我们将x定义为行的iterator，再用个end指向二维数组的末尾，定义一个整型变量y来指向列位置，实现思路和上一种解法完全相同，只是写法略有不同，参见代码如下：

解法3:

```

1 class Vector2D {
2 public:
3     Vector2D(vector<vector<int>>& vec2d) {
4         x = vec2d.begin();
5         end = vec2d.end();
6     }
7     int next() {
8         return (*x)[y++];
9     }
10    bool hasNext() {
11        while (x != end && y == (*x).size()) {
12            ++x;
13            y = 0;
14        }
15        return x != end;
16    }
17 private:
18     vector<vector<int>>::iterator x, end;
19     int y = 0;
20 };

```

CPP

252. 会议室

Given an array of meeting time intervals consisting of start and end times $[[s_1, e_1], [s_2, e_2], \dots]$ ($s_i < e_i$), determine if a person could attend all meetings.

For example,
Given $[[0, 30], [5, 10], [15, 20]]$,
return false.

这道题给了我们一堆会议的时间，问我们能不能同时参见所有的会议，这实际上就是求区间是否有交集的问题，那么最简单暴力的方法就是每两个区间比较一下，看是否有overlap，有的话直接返回false就行了。比较两个区间a和b是否有overlap，我们可以检测两种情况，如果a的起始位置大于等于b的起始位置，且此时a的起始位置小于b的结束位置，那么一定有overlap，另一种情况是a和b互换位置，如果b的起始位置大于等于a的起始位置，且此时b的起始位置小于a的结束位置，那么一定有overlap，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     bool canAttendMeetings(vector<Interval>& intervals) {
4         for (int i = 0; i < intervals.size(); ++i) {
5             for (int j = i + 1; j < intervals.size(); ++j) {
6                 if ((intervals[i].start >= intervals[j].start && intervals[i].start <
7 intervals[j].end) || (intervals[j].start >= intervals[i].start && intervals[j].start <
8 intervals[i].end)) return false;
9             }
10         }
11         return true;
12     }
};
```

CPP

我们可以先给所有区间排个序，用起始时间的先后来排，然后我们从第二个区间开始，如果开始时间早于前一个区间的结束时间，则说明会议时间有冲突，返回false，遍历完成后没有冲突，则返回true，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     bool canAttendMeetings(vector<Interval>& intervals) {
4         sort(intervals.begin(), intervals.end(), [](<const Interval &a, <const Interval &b)
5 {return a.start < b.start;});
6         for (int i = 1; i < intervals.size(); ++i) {
7             if (intervals[i].start < intervals[i - 1].end) {
8                 return false;
9             }
10        }
11        return true;
12    }
};
```

CPP

253. 会议室之二

Given an array of meeting time intervals consisting of start and end times $[[s_1, e_1], [s_2, e_2], \dots]$ ($s_i < e_i$), find the minimum number of conference rooms required.

For example,
Given $[[0, 30], [5, 10], [15, 20]]$,
return 2.

这道题是之前那道Meeting Rooms的拓展，那道题只让我们是否能参加所有的会，也就是看会议之间有没有时间冲突，而這道题让我们求最少需要安排几个会议室，有时间冲突的肯定需要安排在不同的会议室。这道题有好几种解法，我们先来看使用TreeMap来做的，我们遍历时间区间，对于起始时间，映射值自增1，对于结束时间，映射值自减1，然后我们定义结果变量res，和房间数rooms，我们遍历TreeMap，时间从小到大，房间数每次加上映射值，然后更新结果res，遇到起始时间，映射是正数，则房间数会增加，如果一个时间是一个会议的结束时间，也是另一个会议的开始时间，则映射值先减后加仍为0，并不用分配新的房间，而结束时间的映射值为负数更不会增加房间数，利用这种思路我们可以写出代码如下：

解法1：

```

1 class Solution {
2 public:
3     int minMeetingRooms(vector<Interval>& intervals) {
4         map<int, int> m;
5         for (auto a : intervals) {
6             ++m[a.start];
7             --m[a.end];
8         }
9         int rooms = 0, res = 0;
10        for (auto it : m) {
11            res = max(res, rooms += it.second);
12        }
13        return res;
14    }
15 };

```

CPP

第二种方法是用两个一维数组来做，分别保存起始时间和结束时间，然后各自排个序，我们定义结果变量res和结束时间指针endpos，然后我们开始遍历，如果当前起始时间小于结束时间指针的时间，则结果自增1，反之结束时间指针自增1，这样我们可以找出重叠的时间段，从而安排新的会议室，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int minMeetingRooms(vector<Interval>& intervals) {
4         vector<int> starts, ends;
5         int res = 0, endpos = 0;
6         for (auto a : intervals) {
7             starts.push_back(a.start);
8             ends.push_back(a.end);
9         }
10        sort(starts.begin(), starts.end());
11        sort(ends.begin(), ends.end());
12        for (int i = 0; i < intervals.size(); ++i) {
13            if (starts[i] < ends[endpos]) ++res;
14            else ++endpos;
15        }
16        return res;
17    }
18 };

```

再来一看一种使用最小堆来解题的方法，这种方法先把所有的时间区间按照起始时间排序，然后新建一个最小堆，开始遍历时间区间，如果堆不为空，且首元素小于等于当前区间的起始时间，我们去掉堆中的首元素，把当前区间的结束时间压入堆，由于最小堆是小的在前面，那么假如首元素小于等于起始时间，说明上一个会议已经结束，可以用该会议室开始下一个会议了，所以不用分配新的会议室，遍历完成后堆中元素的个数即为需要的会议室的个数，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int minMeetingRooms(vector<Interval>& intervals) {
4         sort(intervals.begin(), intervals.end(), [] (const Interval &a, const Interval &b)
5 {return a.start < b.start;});
6         priority_queue<int, vector<int>, greater<int>> q;
7         for (auto a : intervals) {
8             if (!q.empty() && q.top() <= a.start) q.pop();
9             q.push(a.end);
10        }
11        return q.size();
12    }
13 };

```

254. 因子组合

Numbers can be regarded as product of its factors. For example,

$$\begin{aligned} 8 &= 2 \times 2 \times 2; \\ &= 2 \times 4. \end{aligned}$$

Write a function that takes an integer n and return all possible combinations of its factors.

Note:

Each combination's factors must be sorted ascending, for example: The factors of 2 and 6 is [2, 6], not [6, 2].

You may assume that n is always positive.

Factors should be greater than 1 and less than n.

这道题给了我们一个正整数n，让我们写出所有的因子相乘的形式，而且规定了因子从小到大的顺序排列，那么对于这种需要列出所有情况的题目，通常都是用回溯法来求解的，由于题目中说明了1和n本身不能算其因子，那么我们可以从2开始遍历到n，如果当前的数i可以被n整除，说明i是n的一个因子，我们将其存入一位数组out中，然后递归调用n/i，此时不从2开始遍历，而是从i遍历到n/i，停止的条件是当n等于1时，如果此时out中有因子，我们将这个组合存入结果res中，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<vector<int>> getFactors(int n) {
4         vector<vector<int>> res;
5         helper(n, 2, {}, res);
6         return res;
7     }
8     void helper(int n, int start, vector<int> out, vector<vector<int>> &res) {
9         if (n == 1) {
10             if (out.size() > 1) res.push_back(out);
11         } else {
12             for (int i = start; i <= n; ++i) {
13                 if (n % i == 0) {
14                     out.push_back(i);
15                     helper(n / i, i, out, res);
16                     out.pop_back();
17                 }
18             }
19         }
20     }
21 };

```

CPP

下面这种方法用了个小trick，我们仔细观察题目中给的两个例子的结果，可以发现每个组合的第一个数字都没有超过n的平方根，这个也很好理解，由于要求序列是从小到大排列的，那么如果第一个数字大于了n的平方根，而且n本身又不算因子，那么后面那个因子也必然要与n的平方根，这样乘起来就必然会超过n，所以不会出现这种情况。那么我们刚开始在2到n的平方根之间进行遍历，如果遇到因子，先复制原来的一位数组out为一个新的位数组new_out，然后把此因子i加入new_out，然后再递归调用n/i，并且从i遍历到n/i的平方根，之后再把n/i放入new_out，并且存入结果res，由于层层迭代的调用，凡是本身能继续拆分成更小因数的都能在之后的迭代中拆分出来，并且加上之前结果，最终都会存res中，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<vector<int>> getFactors(int n) {
4         vector<vector<int>> res;
5         helper(n, 2, {}, res);
6         return res;
7     }
8     void helper(int n, int start, vector<int> out, vector<vector<int>> &res) {
9         for (int i = start; i <= sqrt(n); ++i) {
10             if (n % i == 0) {
11                 vector<int> new_out = out;
12                 new_out.push_back(i);
13                 helper(n / i, i, new_out, res);
14                 new_out.push_back(n / i);
15                 res.push_back(new_out);
16             }
17         }
18     }
19 };

```

上面两种解法虽有些小不同，但是构成结果的顺序都是相同，对于题目中给的两个例子 $n = 12$ 和 $n = 32$ ，结果如下：

```

n = 12
2 2 3
2 6
3 4

```

```

n = 32
2 2 2 2 2
2 2 2 4
2 2 8
2 4 4
2 16
4 8

```

上面两种方法得到的结果跟题目中给的答案的顺序不同，虽然顺序不同，但是并不影响其通过OJ。我们下面就给出生成题目中的顺序的解法，这种方法也不难理解，还是从2遍历到 n 的平方根，如果 i 是因子，那么我们递归调用 n/i ，结果用 v 来保存，然后我们新建一个包含 i 和 n/i 两个因子的序列 out ，然后将其存入结果 res ，然后我们再遍历之前递归 n/i 的所得到的序列，如果 i 小于等于某个序列的第一个数字，那么我们将其插入该序列的首位置，然后将序列存入结果 res 中，我们举个例子，比 $n = 12$ ，那么刚开始 $i = 2$ ，是因子，然后对6调用递归，得到{2, 3}，然后此时将{2, 6}先存入结果中，然后发现 i (此时为2)小于等于{2, 3}中的第一个数字2，那么将2插入首位置得到{2, 2, 3}加入结果，然后此时 i 变成3，还是因子，对4调用递归，得到{2, 2}，此时先把{3, 4}存入结果，然后发现 i (此时为3)大于{2, 2}中的第一个数字2，不做任何处理直接返回，这样我们就得到正确的结果了：

解法3：

```

1 class Solution {
2 public:
3     vector<vector<int>> getFactors(int n) {
4         vector<vector<int>> res;
5         for (int i = 2; i * i <= n; ++i) {
6             if (n % i == 0) {
7                 vector<vector<int>> v = getFactors(n / i);
8                 vector<int> out{i, n / i};
9                 res.push_back(out);
10            for (auto a : v) {
11                if (i <= a[0]) {
12                    a.insert(a.begin(), i);
13                    res.push_back(a);
14                }
15            }
16        }
17    }
18    return res;
19 }
20 };

```

这种方法对于对于题目中给的两个例子 $n = 12$ 和 $n = 32$, 结果和题目中给的相同.

255. 验证二叉搜索树的先序序列

Given an array of numbers, verify whether it is the correct preorder traversal sequence of a binary search tree.

You may assume each number in the sequence is unique.

Follow up:

Could you do it using only constant space complexity?

这道题让给了我们一个一维数组, 让我们验证其是否为一个二叉搜索树的先序遍历出的顺序, 我们都知道二叉搜索树的性质是左<根<右, 如果用中序遍历得到的结果就是有序数组, 而先序遍历的结果就不是有序数组了, 但是难道一点规律都没有了吗, 其实规律还是有的, 根据二叉搜索树的性质, 当前节点的值一定大于其左子树中任何一个节点值, 而且其右子树中的任何一个节点值都不能小于当前节点值, 那么我们可以用这个性质来验证, 举个例子, 比如下面这棵二叉搜索树:

```

5
/ \
2   6
/ \
1   3

```

其先序遍历的结果是{5, 2, 1, 3, 6}, 我们先设一个最小值low, 然后遍历数组, 如果当前值小于这个最小值low, 返回false, 对于根节点, 我们将其压入栈中, 然后往后遍历, 如果遇到的数字比栈顶元素小, 说明是其左子树的点, 继续压入栈中, 直到遇到的数字比栈顶元素大, 那么就是右边的值了, 我们需要找到是哪个节点的右子树, 所以我们更新low值并删掉栈顶元素, 然后继续和下一个栈顶元素比较, 如果还是大于, 则继续更新low值和删掉栈顶, 直到栈为空或者当前栈顶元素大于当前值停止, 压入当前值, 这样如果遍历完整个数组之前都没有返回false的话, 最后返回true即可, 参见代码如下:

解法1:

```

1 class Solution {
2 public:
3     bool verifyPreorder(vector<int>& preorder) {
4         int low = INT_MIN;
5         stack<int> s;
6         for (auto a : preorder) {
7             if (a < low) return false;
8             while (!s.empty() && a > s.top()) {
9                 low = s.top(); s.pop();
10            }
11            s.push(a);
12        }
13        return true;
14    }
15 };

```

下面这种方法和上面的思路相同，为了使空间复杂度为常量，我们不能使用stack，所以我们直接修改preorder，将low值存在preorder的特定位置即可，前提是不能影响当前的遍历，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool verifyPreorder(vector<int>& preorder) {
4         int low = INT_MIN, i = -1;
5         for (auto a : preorder) {
6             if (a < low) return false;
7             while (i >= 0 && a > preorder[i]) {
8                 low = preorder[i--];
9             }
10            preorder[++i] = a;
11        }
12        return true;
13    }
14 };

```

下面这种方法使用了分治法，跟之前那道验证二叉搜索树的题Validate Binary Search Tree的思路很类似，我们在递归函数中维护一个下界lower和上届upper，那么当前遍历到的节点值必须在(lower, upper)区间之内，然后我们在给定的区间内搜第一个大于当前节点值的点，然后以此为分界，左右两部分分别调用递归函数，注意左半部分的upper更新为当前节点值val，表明左子树的节点值都必须小于当前节点值，而右半部分的递归的lower更新为当前节点值val，表明右子树的节点值都必须大于当前节点值，如果左右两部分的返回结果均为真，则整体返回真，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     bool verifyPreorder(vector<int>& preorder) {
4         return helper(preorder, 0, preorder.size() - 1, INT_MIN, INT_MAX);
5     }
6     bool helper(vector<int> &preorder, int start, int end, int lower, int upper) {
7         if (start > end) return true;
8         int val = preorder[start], i = 0;
9         if (val <= lower || val >= upper) return false;
10        for (i = start + 1; i <= end; ++i) {
11            if (preorder[i] >= val) break;
12        }
13        return helper(preorder, start + 1, i - 1, lower, val) && helper(preorder, i, end,
14 val, upper);
15    }
16 };

```

256. 粉刷房子

There are a row of n houses, each house can be painted with one of the three colors: red, blue or green. The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color.

The cost of painting each house with a certain color is represented by a $n \times 3$ cost matrix. For example, $\text{costs}[0][0]$ is the cost of painting house 0 with color red; $\text{costs}[1][2]$ is the cost of painting house 1 with color green, and so on... Find the minimum cost to paint all houses.

Note:

All costs are positive integers.

这道题说有n个房子，每个房子可以用红绿蓝三种颜色刷，每个房子的用每种颜色刷的花费都不同，限制条件是相邻的房子不能用相同颜色来刷，现在让我们求刷完所有的房子的最低花费是多少。这题跟House Robber II和House Robber很类似，不过那题不是每个房子都抢，相邻的房子不抢，而这道题是每个房子都刷，相邻的房子不能刷同一种颜色。而Paint Fence那道题主要考察我们有多少种刷法，这几道题很类似，但都不一样，需要我们分别区分。但是它们的解题思想都一样，需要用动态规划Dynamic Programming来做，这道题我们需要维护一个二维的动态数组dp，其中 $dp[i][j]$ 表示刷到第*i+1*房子用颜色*j*的最小花费，递推式为：

$$dp[i][j] = dp[i][j] + \min(dp[i - 1][(j + 1) \% 3], dp[i - 1][(j + 2) \% 3]);$$

这个也比较好理解，如果当前的房子要用红色刷，那么上一个房子只能用绿色或蓝色来刷，那么我们要求刷到当前房子，且当前房子用红色刷的最小花费就等于当前房子用红色刷的钱加上刷到上一个房子用绿色和刷到上一个房子用蓝色的较小值，这样当我们算到最后一个房子时，我们只要取出三个累计花费的最小值即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int minCost(vector<vector<int>>& costs) {
4         if (costs.empty() || costs[0].empty()) return 0;
5         vector<vector<int>> dp = costs;
6         for (int i = 1; i < dp.size(); ++i) {
7             for (int j = 0; j < 3; ++j) {
8                 dp[i][j] += min(dp[i - 1][(j + 1) % 3], dp[i - 1][(j + 2) % 3]);
9             }
10        }
11        return min(min(dp.back()[0], dp.back()[1]), dp.back()[2]);
12    }
13 };

```

由于只有红绿蓝三张颜色，所以我们就可以分别写出各种情况，这样写可能比上面的写法更加一目了然一些，更容易理解一点吧：

解法2：

```

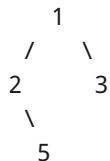
1 class Solution {
2 public:
3     int minCost(vector<vector<int>>& costs) {
4         if (costs.empty() || costs[0].empty()) return 0;
5         vector<vector<int>> dp = costs;
6         for (int i = 1; i < dp.size(); ++i) {
7             dp[i][0] += min(dp[i - 1][1], dp[i - 1][2]);
8             dp[i][1] += min(dp[i - 1][0], dp[i - 1][2]);
9             dp[i][2] += min(dp[i - 1][0], dp[i - 1][1]);
10        }
11        return min(min(dp.back()[0], dp.back()[1]), dp.back()[2]);
12    }
13 };

```

257. 二叉树路径

Given a binary tree, return all root-to-leaf paths.

For example, given the following binary tree:



All root-to-leaf paths are:

["1->2->5", "1->3"]

这道题给我们一个二叉树，让我们返回所有根到叶节点的路径，跟之前那道Path Sum II很类似，比那道稍微简单一些，不需要计算路径和，只需要无脑返回所有的路径即可，那么思路还是用递归来解，博主之前就强调过，玩树的题目，十有八九都是递归，而递归的核心就是不停的DFS到叶结点，然后在回溯回去。在递归函数中，当我们遇到叶结点的时候，即没有左右子结点，

那么此时一条完整的路径已经形成了，我们加上当前的叶结点后存入结果res中，然后回溯。注意这里结果res需要reference，而out是不需要引用的，不然回溯回去还要删除新添加的结点，很麻烦。为了减少判断空结点的步骤，我们在调用递归函数之前都检验一下非空即可，代码也很简洁，参见如下：

解法1：

```
1 class Solution {
2 public:
3     vector<string> binaryTreePaths(TreeNode* root) {
4         vector<string> res;
5         if (root) helper(root, "", res);
6         return res;
7     }
8     void helper(TreeNode* node, string out, vector<string>& res) {
9         if (!node->left && !node->right) res.push_back(out + to_string(node->val));
10        if (node->left) helper(node->left, out + to_string(node->val) + "->", res);
11        if (node->right) helper(node->right, out + to_string(node->val) + "->", res);
12    }
13};
```

下面再来看一种递归的方法，这个方法直接在一个函数中完成递归调用，不需要另写一个helper函数，核心思想和上面没有区别，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     vector<string> binaryTreePaths(TreeNode* root) {
4         if (!root) return {};
5         if (!root->left && !root->right) return {to_string(root->val)};
6         vector<string> left = binaryTreePaths(root->left);
7         vector<string> right = binaryTreePaths(root->right);
8         left.insert(left.end(), right.begin(), right.end());
9         for (auto &a : left) {
10             a = to_string(root->val) + "->" + a;
11         }
12         return left;
13     }
14};
```

还是递归写法，从论坛中扒下来的解法，核心思路都一样啦，写法各有不同而已，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     vector<string> binaryTreePaths(TreeNode* root) {
4         if (!root) return {};
5         if (!root->left && !root->right) return {to_string(root->val)};
6         vector<string> res;
7         for (string str : binaryTreePaths(root->left)) {
8             res.push_back(to_string(root->val) + "->" + str);
9         }
10        for (string str : binaryTreePaths(root->right)) {
11            res.push_back(to_string(root->val) + "->" + str);
12        }
13        return res;
14    }
15 };

```

258. 加数字

Given a non-negative integer num, repeatedly add all its digits until the result has only one digit.

For example:

Given num = 38, the process is like: 3 + 8 = 11, 1 + 1 = 2. Since 2 has only one digit, return it.

Follow up:

Could you do it without any loop/recursion in O(1) runtime?

Hint:

A naive implementation of the above process is trivial. Could you come up with other methods?

What are all the possible results?

How do they occur, periodically or randomly?

You may find this Wikipedia article useful.

这道题让我们求数根，所谓树根，就是将大于10的数的各个位上的数字相加，若结果还大于0的话，则继续相加，直到数字小于10为止。那么根据这个性质，我们可以写出一个解法如下：

解法1:

```

1 class Solution {
2 public:
3     int addDigits(int num) {
4         while (num / 10 > 0) {
5             int sum = 0;
6             while (num > 0) {
7                 sum += num % 10;
8                 num /= 10;
9             }
10            num = sum;
11        }
12        return num;
13    }
14 };

```

但是这个解法在出题人看来又trivial又naive，需要想点高逼格的解法，一行搞定碉堡了，那么我们先来观察1到20的所有树根：

```

1   1
2   2
3   3
4   4
5   5
6   6
7   7
8   8
9   9
10  1
11  2
12  3
13  4
14  5
15  6
16  7
17  8
18  9
19  1
20  2

```

根据上面的列举，我们可以得出规律，每9个一循环，所有大于9的数的树根都是对9取余，那么对于等于9的数对9取余就是0了，为了得到其本身，而且同样也要对大于9的数适用，我们就用 $(n-1)\%9+1$ 这个表达式来包括所有的情况，所以解法如下：

解法2：

```

1 class Solution {
2 public:
3     int addDigits(int num) {
4         return (num - 1) % 9 + 1;
5     }
6 };

```

CPP

259. 三数之和较小值

Given an array of n integers nums and a target, find the number of index triplets i, j, k with $0 \leq i < j < k < n$ that satisfy the condition $\text{nums}[i] + \text{nums}[j] + \text{nums}[k] < \text{target}$.

For example, given $\text{nums} = [-2, 0, 1, 3]$, and $\text{target} = 2$.

Return 2. Because there are two triplets which sums are less than 2:

```

[-2, 0, 1]
[-2, 0, 3]
Follow up:
Could you solve it in O(n^2) runtime?

```

这道题是3Sum问题的一个变形，让我们求三数之和小于一个目标值，那么最简单的方法就是穷举法，将所有的可能的三个数字的组合都遍历一遍，比较三数之和跟目标值之间的大小，小于的话则结果自增1，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int threeSumSmaller(vector<int>& nums, int target) {
4         int res = 0;
5         sort(nums.begin(), nums.end());
6         for (int i = 0; i < int(nums.size() - 2); ++i) {
7             int left = i + 1, right = nums.size() - 1, sum = target - nums[i];
8             for (int j = left; j <= right; ++j) {
9                 for (int k = j + 1; k <= right; ++k) {
10                     if (nums[j] + nums[k] < sum) ++res;
11                 }
12             }
13         }
14         return res;
15     }
16 };

```

题目中的Follow up让我们在O(n^2)的时间复杂度内实现，那么我们借鉴之前那两道题3Sum Closest和3Sum中的方法，采用双指针来做，这里面有个trick就是当判断三个数之和小于目标值时，此时结果应该加上right-left，以为数组排序了以后，如果加上num[right]小于目标值的话，那么加上一个更小的数必定也会小于目标值，然后我们将左指针右移一位，否则我们将右指针左移一位，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int threeSumSmaller(vector<int>& nums, int target) {
4         if (nums.size() < 3) return 0;
5         int res = 0, n = nums.size();
6         sort(nums.begin(), nums.end());
7         for (int i = 0; i < n - 2; ++i) {
8             int left = i + 1, right = n - 1;
9             while (left < right) {
10                 if (nums[i] + nums[left] + nums[right] < target) {
11                     res += right - left;
12                     ++left;
13                 } else {
14                     --right;
15                 }
16             }
17         }
18         return res;
19     }
20 };

```

260. 单独的数字之三

Given an array of numbers `nums`, in which exactly two elements appear only once and all the other elements appear exactly twice. Find the two elements that appear only once.

For example:

Given `nums = [1, 2, 1, 3, 2, 5]`, return `[3, 5]`.

Note:

The order of the result is not important. So in the above example, `[5, 3]` is also correct. Your algorithm should run in linear runtime complexity. Could you implement it using only constant space complexity?

这道题是之前那两道Single Number 单独的数字和 Single Number II 单独的数字之二的再次延伸，说实话，这类位操作Bit Manipulation的题，如果之前没有遇到过类似的题目，愣想是很难想出来的，于是我只能上网搜大神们的解法，发现还真是巧妙啊。这道题其实是很巧妙的利用了Single Number 单独的数字的解法，因为那道解法是可以准确的找出只出现了一次的数字，但前提是其他数字必须出现两次才行。而这题有两个数字都只出现了一次，那么我们如果能想办法把原数组分为两个小数组，不相同的两个数字分别在两个小数组中，这样分别调用Single Number 单独的数字的解法就可以得到答案。那么如何实现呢，首先我们先把原数组全部异或起来，那么我们会得到一个数字，这个数字是两个不相同的数字异或的结果，我们取出其中任意一位为‘1’的位，为了方便起见，我们用 `a &= -a` 来取出最右端为‘1’的位，然后和原数组中的数字挨个相与，那么我们要求的两个不同的数字就被分到了两个小组中，分别将两个小组中的数字都异或起来，就可以得到最终结果了，参见代码如下：

```
1 class Solution {
2 public:
3     vector<int> singleNumber(vector<int>& nums) {
4         int diff = accumulate(nums.begin(), nums.end(), 0, bit_xor<int>());
5         diff &= -diff;
6         vector<int> res(2, 0);
7         for (auto &a : nums) {
8             if (a & diff) res[0] ^= a;
9             else res[1] ^= a;
10        }
11        return res;
12    }
13};
```

CPP

261. 图验证树

Given `n` nodes labeled from `0` to `n - 1` and a list of undirected edges (each edge is a pair of nodes), write a function to check whether these edges make up a valid tree.

For example:

Given `n = 5` and `edges = [[0, 1], [0, 2], [0, 3], [1, 4]]`, return `true`.

Given `n = 5` and `edges = [[0, 1], [1, 2], [2, 3], [1, 3], [1, 4]]`, return `false`.

Hint:

Given `n = 5` and `edges = [[0, 1], [1, 2], [3, 4]]`, what should your return? Is this case a valid tree?

According to the definition of tree on Wikipedia: “a tree is an undirected graph in which any two vertices are connected by exactly one path. In other words, any connected graph without simple cycles is a tree.”

Note: you can assume that no duplicate edges will appear in edges. Since all edges are undirected, `[0, 1]` is the same as `[1, 0]` and thus will not appear together in edges.

这道题给了我们一个无向图，让我们来判断其是否为一棵树，我们知道如果是树的话，所有的节点必须是连接的，也就是说必须是连通图，而且不能有环，所以我们的焦点就变成了验证是否是连通图和是否含有环。我们首先用DFS来做，根据pair来建立一个图的结构，用邻接链表来表示，还需要一个一位数组v来记录某个节点是否被访问过，然后我们用DFS来搜索节点0，遍历的思想是，当DFS到某个节点，先看当前节点是否被访问过，如果已经被访问过，说明环存在，直接返回false，如果未被访问过，我们现在将其状态标记为已访问过，然后我们到邻接链表里去找跟其相邻的节点继续递归遍历，注意我们还需要一个变量pre来记录上一个节点，以免回到上一个节点，这样遍历结束后，我们就把和节点0相邻的节点都标记为true，然后我们在看v里面是否还有没被访问过的节点，如果有，则说明图不是完全连通的，返回false，反之返回true，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool validTree(int n, vector<pair<int, int>>& edges) {
4         vector<vector<int>> g(n, vector<int>());
5         vector<bool> v(n, false);
6         for (auto a : edges) {
7             g[a.first].push_back(a.second);
8             g[a.second].push_back(a.first);
9         }
10        if (!dfs(g, v, 0, -1)) return false;
11        for (auto a : v) {
12            if (!a) return false;
13        }
14        return true;
15    }
16    bool dfs(vector<vector<int>> &g, vector<bool> &v, int cur, int pre) {
17        if (v[cur]) return false;
18        v[cur] = true;
19        for (auto a : g[cur]) {
20            if (a != pre) {
21                if (!dfs(g, v, a, cur)) return false;
22            }
23        }
24        return true;
25    }
26 };

```

CPP

下面我们来看BFS的解法，思路很相近，需要用queue来辅助遍历，这里我们没有用一维向量来标记节点是否访问过，而是用了一个set，如果遍历到一个节点，在set中没有，则加入set，如果已经存在，则返回false，还有就是在遍历邻接链表的时候，遍历完成后需要将节点删掉，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool validTree(int n, vector<pair<int, int>>& edges) {
4         vector<unordered_set<int>> g(n, unordered_set<int>());
5         unordered_set<int> s{{0}};
6         queue<int> q{{0}};
7         for (auto a : edges) {
8             g[a.first].insert(a.second);
9             g[a.second].insert(a.first);
10        }
11        while (!q.empty()) {
12            int t = q.front(); q.pop();
13            for (auto a : g[t]) {
14                if (s.count(a)) return false;
15                s.insert(a);
16                q.push(a);
17                g[a].erase(t);
18            }
19        }
20        return s.size() == n;
21    }
22 }

```

我们再来看Union Find的方法，这种方法对于解决连通图的问题很有效，思想是我们遍历节点，如果两个节点相连，我们将其roots值连上，这样可以帮助我们找到环，我们初始化roots数组为-1，然后对于一个pair的两个节点分别调用find函数，得到的值如果相同的话，则说明环存在，返回false，不同的话，我们将其roots值union上，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     bool validTree(int n, vector<pair<int, int>>& edges) {
4         vector<int> roots(n, -1);
5         for (auto a : edges) {
6             int x = find(roots, a.first), y = find(roots, a.second);
7             if (x == y) return false;
8             roots[x] = y;
9         }
10        return edges.size() == n - 1;
11    }
12    int find(vector<int> &roots, int i) {
13        while (roots[i] != -1) i = roots[i];
14        return i;
15    }
16 }

```

262. 旅行和用户

The Trips table holds all taxi trips. Each trip has a unique Id, while Client_Id and Driver_Id are both foreign keys to the Users_Id at the Users table. Status is an ENUM type of ('completed', 'cancelled_by_driver', 'cancelled_by_client').

Id Client_Id Driver_Id City_Id Status Request_at
1 1 10 1 completed 2013-10-01
2 2 11 1 cancelled_by_driver 2013-10-01
3 3 12 6 completed 2013-10-01
4 4 13 6 cancelled_by_client 2013-10-01
5 1 10 1 completed 2013-10-02
6 2 11 6 completed 2013-10-02
7 3 12 6 completed 2013-10-02
8 2 12 12 completed 2013-10-03
9 3 10 12 completed 2013-10-03
10 4 13 12 cancelled_by_driver 2013-10-03

The Users table holds all users. Each user has an unique Users_Id, and Role is an ENUM type of ('client', 'driver', 'partner').

Users_Id Banned Role
1 No client
2 Yes client
3 No client
4 No client
10 No driver
11 No driver
12 No driver
13 No driver

Write a SQL query to find the cancellation rate of requests made by unbanned clients between Oct 1, 2013 and Oct 3, 2013. For the above tables, your SQL query should return the following rows with the cancellation rate being rounded to two decimal places.

Day Cancellation Rate
2013-10-01 0.33
2013-10-02 0.00
2013-10-03 0.50

这道题给了我们一个Trips表里面有一些Id和状态，还有请求时间，然后还有一个Users表，里面有顾客和司机的信息，然后有该顾客和司机有没有被Ban的信息，让我们返回一个结果看某个时间段内由没有被ban的顾客提出的取消率是多少，其实题目没有说清楚顾客到底包不包括司机，其实是包括的，由司机提出的取消请求也应计算进去，我们用Case When ... Then ... Else ... End关键字来做，我们用cancelled%来表示开头是cancelled的所有项，这样就包括了driver和client，然后分母是所有项，限制条件里限定了时间段，然后是没有被ban的，由于结果需要保留两位小数，所以我们用Round关键字且给定参数2即可，参见代码如下：

解法1：

```

1 | SELECT t.Request_at Day, ROUND(SUM(CASE WHEN t.Status LIKE 'cancelled%' THEN 1 ELSE 0
2 | END)/COUNT(*), 2) 'Cancellation Rate'
3 | FROM Trips t JOIN Users u ON t.Client_Id = u.Users_Id AND u.Banned = 'No'
| WHERE t.Request_at BETWEEN '2013-10-01' AND '2013-10-03' GROUP BY t.Request_at;

```

SQL

上面的Case When ... Then ... Else ... End关键字也可以用If关键字来替换，实现的效果一样：

解法2：

```

1 | SELECT Request_at Day, ROUND(COUNT(IF(Status != 'completed', TRUE, NULL)) / COUNT(*), 2)
2 | 'Cancellation Rate'
3 | FROM Trips WHERE (Request_at BETWEEN '2013-10-01' AND '2013-10-03') AND Client_Id IN
| (SELECT Users_Id FROM Users WHERE Banned = 'No') GROUP BY Request_at;

```

CPP

263. 丑陋数

Write a program to check whether a given number is an ugly number.

Ugly numbers are positive numbers whose prime factors only include 2, 3, 5. For example, 6, 8 are ugly while 14 is not ugly since it includes another prime factor 7.

Note that 1 is typically treated as an ugly number.

这道题让我们检测一个数是否为丑陋数，所谓丑陋数就是其质数因子只能是2,3,5。那么最直接的办法就是不停的除以这些质数，如果剩余的数字是1的话就是丑陋数了，有两种写法，如下所示：

解法1：

```

1 | class Solution {
2 | public:
3 |     bool isUgly(int num) {
4 |         while (num >= 2) {
5 |             if (num % 2 == 0) num /= 2;
6 |             else if (num % 3 == 0) num /= 3;
7 |             else if (num % 5 == 0) num /= 5;
8 |             else return false;
9 |         }
10 |         return num == 1;
11 |     }
12 | };

```

CPP

解法2：

```

1 | class Solution {
2 | public:
3 |     bool isUgly(int num) {
4 |         if (num <= 0) return false;
5 |         while (num % 2 == 0) num /= 2;
6 |         while (num % 3 == 0) num /= 3;
7 |         while (num % 5 == 0) num /= 5;
8 |         return num == 1;
9 |     }
10 | };

```

CPP

264. 丑陋数之二

Write a program to find the n-th ugly number.

Ugly numbers are positive numbers whose prime factors only include 2, 3, 5. For example, 1, 2, 3, 4, 5, 6, 8, 9, 10, 12 is the sequence of the first 10 ugly numbers.

Note that 1 is typically treated as an ugly number.

Hint:

The naive approach is to call isUgly for every number until you reach the nth one. Most numbers are not ugly. Try to focus your effort on generating only the ugly ones.

An ugly number must be multiplied by either 2, 3, or 5 from a smaller ugly number.

The key is how to maintain the order of the ugly numbers. Try a similar approach of merging from three sorted lists: L1, L2, and L3.

Assume you have U_k , the kth ugly number. Then U_{k+1} must be $\min(L_1 * 2, L_2 * 3, L_3 * 5)$.

这道题是之前那道Ugly Number 丑陋数的延伸，这里让我们找到第n个丑陋数，还好题目中给了很多提示，基本上相当于告诉我们解法了，根据提示中的信息，我们知道丑陋数序列可以拆分为下面3个子列表：

- (1) $1 \times 2, 2 \times 2, 2 \times 2, 3 \times 2, 3 \times 2, 4 \times 2, 5 \times 2 \dots$
- (2) $1 \times 3, 1 \times 3, 2 \times 3, 2 \times 3, 2 \times 3, 3 \times 3, 3 \times 3 \dots$
- (3) $1 \times 5, 1 \times 5, 1 \times 5, 1 \times 5, 2 \times 5, 2 \times 5, 2 \times 5 \dots$

仔细观察上述三个列表，我们可以发现每个子列表都是一个丑陋数分别乘以2,3,5，而要求的丑陋数就是从已经生成的序列中取出的，我们每次都从三个列表中取出当前最小的那个加入序列，请参见代码如下：

```

1 class Solution {
2 public:
3     int nthUglyNumber(int n) {
4         vector<int> res(1, 1);
5         int i2 = 0, i3 = 0, i5 = 0;
6         while (res.size() < n) {
7             int m2 = res[i2] * 2, m3 = res[i3] * 3, m5 = res[i5] * 5;
8             int mn = min(m2, min(m3, m5));
9             if (mn == m2) ++i2;
10            if (mn == m3) ++i3;
11            if (mn == m5) ++i5;
12            res.push_back(mn);
13        }
14        return res.back();
15    }
16 }
```

CPP

265. 粉刷房子之二

There are a row of n houses, each house can be painted with one of the k colors. The cost of painting each house with a certain color is represented by a $n \times k$ cost matrix. For example, $\text{costs}[0][0]$ is the cost of painting house 0 with color 0; $\text{costs}[1][2]$ is the cost of painting house 1 with color 2, and so on... Find the minimum cost to paint all houses.

The cost of painting each house with a certain color is represented by a $n \times k$ cost matrix. For example, $\text{costs}[0][0]$ is the cost of painting house 0 with color 0; $\text{costs}[1][2]$ is the cost of painting house 1 with color 2, and so on... Find the minimum cost to paint all houses.

Note:

All costs are positive integers.

Follow up:

Could you solve it in $O(nk)$ runtime?

这道题是之前那道Paint House的拓展，那道题只让用红绿蓝三种颜色来粉刷房子，而这道题让我们用 k 种颜色，这道题不能用之前那题的解法，会TLE。这题的解法的思路还是用DP，但是在找不同颜色的最小值不是遍历所有不同颜色，而是用min1和min2来记录之前房子的最小和第二小的花费的颜色，如果当前房子颜色和min1相同，那么我们用min2对应的值计算，反之我们用min1对应的值，这种解法实际上也包含了求次小值的方法，感觉也是一种很棒的解题思路，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     int minCostII(vector<vector<int>>& costs) {
4         if (costs.empty() || costs[0].empty()) return 0;
5         vector<vector<int>> dp = costs;
6         int min1 = -1, min2 = -1;
7         for (int i = 0; i < dp.size(); ++i) {
8             int last1 = min1, last2 = min2;
9             min1 = -1; min2 = -1;
10            for (int j = 0; j < dp[i].size(); ++j) {
11                if (j != last1) {
12                    dp[i][j] += last1 < 0 ? 0 : dp[i - 1][last1];
13                } else {
14                    dp[i][j] += last2 < 0 ? 0 : dp[i - 1][last2];
15                }
16                if (min1 < 0 || dp[i][j] < dp[i][min1]) {
17                    min2 = min1; min1 = j;
18                } else if (min2 < 0 || dp[i][j] < dp[i][min2]) {
19                    min2 = j;
20                }
21            }
22        }
23        return dp.back()[min1];
24    }
25 }
```

CPP

下面这种解法不需要建立二维dp数组，直接用三个变量就可以保存需要的信息即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int minCostII(vector<vector<int>>& costs) {
4         if (costs.empty() || costs[0].empty()) return 0;
5         int min1 = 0, min2 = 0, idx1 = -1;
6         for (int i = 0; i < costs.size(); ++i) {
7             int m1 = INT_MAX, m2 = m1, id1 = -1;
8             for (int j = 0; j < costs[i].size(); ++j) {
9                 int cost = costs[i][j] + (j == idx1 ? min2 : min1);
10                if (cost < m1) {
11                    m2 = m1; m1 = cost; id1 = j;
12                } else if (cost < m2) {
13                    m2 = cost;
14                }
15            }
16            min1 = m1; min2 = m2; idx1 = id1;
17        }
18        return min1;
19    }
20 };

```

266. 回文全排列

Given a string, determine if a permutation of the string could form a palindrome.

For example,
 "code" -> False, "aab" -> True, "carerac" -> True.

Hint:

Consider the palindromes of odd vs even length. What difference do you notice?
 Count the frequency of each character.
 If each character occurs even number of times, then it must be a palindrome. How about character which occurs odd number of times?

这道题让我们判断一个字符串的全排列有没有是回文字符串的，那么根据题目中的提示，我们分字符串的个数是奇偶的情况来讨论，如果是偶数的话，由于回文字符串的特性，每个字母出现的次数一定是偶数次，当字符串是奇数长度时，只有一个字母出现的次数是奇数，其余均为偶数，那么利用这个特性我们就可以解题，我们建立每个字母和其出现次数的映射，然后我们遍历哈希表，统计出现次数为奇数的字母的个数，那么只有两种情况是回文数，第一种是没有出现次数为奇数的字母，再一个就是字符串长度为奇数，且只有一个出现次数为奇数的字母，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool canPermutePalindrome(string s) {
4         unordered_map<char, int> m;
5         int cnt = 0;
6         for (auto a : s) ++m[a];
7         for (auto it = m.begin(); it != m.end(); ++it) {
8             if (it->second % 2) ++cnt;
9         }
10        return cnt == 0 || (s.size() % 2 == 1 && cnt == 1);
11    }
12 };

```

那么我们再来看一种解法，这种方法用到了一个set，我们遍历字符串，如果某个字母不在set中，我们加入这个字母，如果字母已经存在，我们删除该字母，那么最终如果set中没有字母或是只有一个字母时，说明是回文串，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     bool canPermutePalindrome(string s) {
4         set<char> t;
5         for (auto a : s) {
6             if (t.find(a) == t.end()) t.insert(a);
7             else t.erase(a);
8         }
9         return t.empty() || t.size() == 1;
10    }
11 };
12 }
```

CPP

再来看一种bitset的解法，这种方法也很巧妙，我们建立一个256大小的bitset，每个字母根据其ASCII码值的不同都有其对应的位置，然后我们遍历整个字符串，遇到一个字符，就将其对应的位置的二进制数flip一下，就是0变1，1变0，那么遍历完成后，所有出现次数为偶数的对应位置还应该为0，而出现次数为奇数的时候，对应位置就为1了，那么我们最后只要统计1的个数，就知道出现次数为奇数的字母的个数了，只要个数小于2就是回文数，参见代码如下：

解法3：

```
1 class Solution {
2 public:
3     bool canPermutePalindrome(string s) {
4         bitset<256> b;
5         for (auto a : s) {
6             b.flip(a);
7         }
8         return b.count() < 2;
9     }
10 };
```

CPP

267. 回文全排列之二

Given a string s, return all the palindromic permutations (without duplicates) of it. Return an empty list if no palindromic permutation could be formed.

For example:

Given s = "aabb", return ["abba", "baab"].

Given s = "abc", return [].

Hint:

If a palindromic permutation exists, we just need to generate the first half of the string. To generate all distinct permutations of a (half of) string, use a similar approach from: Permutations II or Next Permutation.

这道题是之前那道Palindrome Permutation的拓展，那道题只是让判断存不存在回文全排列，而这题让我们返回所有的回文全排列，此题给了我们充分的提示：如果回文全排列存在，我们只需要生成前半段字符串即可，后面的直接根据前半段得到。那么我们再进一步思考，由于回文字符串有奇偶两种情况，偶数回文串例如abba，可以平均分成前后半段，而奇数回文串例如abcba，需要分成前中后三段，需要注意的是中间部分只能是一个字符，那么我们可以分析得出，如果一个字符串的回文字符串要存在，那么奇数个的字符只能有0个或1个，其余的必须是偶数个，所以我们可以用哈希表来记录所有字符的出现个数，然后我们找出出现奇数次数的字符加入mid中，如果有两个或两个以上的奇数个数的字符，那么返回空集，我们对于每个字符，不管其奇偶，都将其个数除以2的个数的字符加入t中，这样做的原因是如果是偶数个，那么将其一般加入t中，如果是奇数，如果有1个，那么除以2是0，不会有字符加入t，如果是3个，那么除以2是1，取一个加入t。等我们获得了t之后，t是就是前半段字符，我们对其做全排列，每得到一个全排列，我们加上mid和该全排列的逆序列就是一种所求的回文字符串，这样我们就可以得到所有的回文全排列了。在全排序的子函数中有一点需要注意的是，如果我们直接用数组来保存结果时，并且t中如果有重复字符的话可能会出现重复项，比如t = "baa" 的话，那么最终生成的结果会有重复项，不信可以自己尝试一下。这里简单的说明一下，当start=0, i=1时，我们交换后得到aba，在之后当start=1, i=2时，交换后可以得到aab。但是在之后回到第一层当baa后，当start=0, i=2时，交换后又得到了aab，重复就产生了。那么其实最简单当去重复的方法就是将结果res定义成HashSet，利用其去重复的特性，可以保证我们得到的是没有重复的，参见代码如下：

解法1：

CPP

```

1 class Solution {
2 public:
3     vector<string> generatePalindromes(string s) {
4         unordered_set<string> res;
5         unordered_map<char, int> m;
6         string t = "", mid = "";
7         for (auto a : s) ++m[a];
8         for (auto it : m) {
9             if (it.second % 2 == 1) mid += it.first;
10            t += string(it.second / 2, it.first);
11            if (mid.size() > 1) return {};
12        }
13        permute(t, 0, mid, res);
14        return vector<string>(res.begin(), res.end());
15    }
16    void permute(string &t, int start, string mid, unordered_set<string> &res) {
17        if (start >= t.size()) {
18            res.insert(t + mid + string(t.rbegin(), t.rend()));
19        }
20        for (int i = start; i < t.size(); ++i) {
21            if (i != start && t[i] == t[start]) continue;
22            swap(t[i], t[start]);
23            permute(t, start + 1, mid, res);
24            swap(t[i], t[start]);
25        }
26    }
27};

```

下面这种方法和上面的方法很相似，不同之处在于求全排列的方法略有不同，上面那种方法是通过交换字符的位置来生成不同的字符串，而下面这种方法是通过加不同的字符来生成全排列字符串，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<string> generatePalindromes(string s) {
4         vector<string> res;
5         unordered_map<char, int> m;
6         string t = "", mid = "";
7         for (auto a : s) ++m[a];
8         for (auto &it : m) {
9             if (it.second % 2 == 1) mid += it.first;
10            it.second /= 2;
11            t += string(it.second, it.first);
12            if (mid.size() > 1) return {};
13        }
14        permute(t, m, mid, "", res);
15        return res;
16    }
17    void permute(string &t, unordered_map<char, int> &m, string mid, string out,
18    vector<string> &res) {
19        if (out.size() >= t.size()) {
20            res.push_back(out + mid + string(out.rbegin(), out.rend()));
21            return;
22        }
23        for (auto &it : m) {
24            if (it.second > 0) {
25                --it.second;
26                permute(t, m, mid, out + it.first, res);
27                ++it.second;
28            }
29        }
30    }
31 };

```

在来看一种利用了std提供的next_permutation函数来实现的方法，这样就大大减轻了我们的工作量，但是这种方法个人感觉算是有些投机取巧了，不知道面试的时候面试官允不允许这样做，贴上来拓宽一下思路也是好的：

解法3：

```

1 class Solution {
2 public:
3     vector<string> generatePalindromes(string s) {
4         vector<string> res;
5         unordered_map<char, int> m;
6         string t = "", mid = "";
7         for (auto a : s) ++m[a];
8         for (auto it : m) {
9             if (it.second % 2 == 1) mid += it.first;
10            t += string(it.second / 2, it.first);
11            if (mid.size() > 1) return {};
12        }
13        sort(t.begin(), t.end());
14        do {
15            res.push_back(t + mid + string(t.rbegin(), t.rend()));
16        } while (next_permutation(t.begin(), t.end()));
17        return res;
18    }
19 };

```

268. 丢失的数字

Given an array containing n distinct numbers taken from 0, 1, 2, ..., n, find the one that is missing from the array.

For example,

Given `nums = [0, 1, 3]` return 2.

Note:

Your algorithm should run in linear runtime complexity. Could you implement it using only constant extra space complexity?

这道题给我们n个数字，是0到n之间的数但是有一个数字去掉了，让我们寻找这个数字，要求线性的时间复杂度和常数级的空间复杂度。那么最直观的一个方法是用等差数列的求和公式求出0到n之间所有的数字之和，然后再遍历数组算出给定数字的累积和，然后做减法，差值就是丢失的那个数字，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     int missingNumber(vector<int>& nums) {
4         int sum = 0, n = nums.size();
5         for (auto &a : nums) {
6             sum += a;
7         }
8         return 0.5 * n * (n + 1) - sum;
9     }
10};
```

CPP

这题还有一种解法，使用位操作Bit Manipulation来解的，用到了异或操作的特性，相似的题目有Single Number 单独的数字，Single Number II 单独的数字之二和Single Number III 单独的数字之三。那么思路是既然0到n之间少了一个数，我们将这个少了一个数的数组和0到n之间完整的数组异或一下，那么相同的数字都变为0了，剩下的就是少了的那个数字了，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     int missingNumber(vector<int>& nums) {
4         int res = 0;
5         for (int i = 0; i < nums.size(); ++i) {
6             res ^= (i + 1) ^ nums[i];
7         }
8         return res;
9     }
10};
```

CPP

这道题还可以用二分查找法来做，我们首先要对数组排序，然后我们用二分查找法算出中间元素的下标，然后用元素值和下标值之间做对比，如果元素值大于下标值，则说明缺失的数字在左边，此时将right赋为mid，反之则将left赋为mid+1。那么看到这里，作为读者的你可能会提出，排序的时间复杂度都不止O(n)，何必要多此一举用二分查找，还不如用上面两种方法呢。对，你说的没错，但是在面试的时候，有可能人家给你的数组就是排好序的，那么此时用二分查找法肯定要优于上面两种方法，所以这种方法最好也要掌握以下~

解法3：

```

1 class Solution {
2 public:
3     int missingNumber(vector<int>& nums) {
4         sort(nums.begin(), nums.end());
5         int left = 0, right = nums.size();
6         while (left < right) {
7             int mid = left + (right - left) / 2;
8             if (nums[mid] > mid) right = mid;
9             else left = mid + 1;
10        }
11        return right;
12    }
13 };

```

在CareerCup中有一道类似的题，5.7 Find Missing Integer 查找丢失的数，但是那道题不让我们直接访问整个int数字，而是只能访问其二进制表示数中的某一位，强行让我们使用位操作Bit Manipulation来解题，也是蛮有意思的一道题。

269. 另类字典

There is a new alien language which uses the latin alphabet. However, the order among letters are unknown to you. You receive a list of words from the dictionary, where words are sorted lexicographically by the rules of this new language. Derive the order of letters in this language.

For example,
Given the following words in dictionary,

```
[
    "wrt",
    "wrf",
    "er",
    "ett",
    "rftt"
]
```

The correct order is: "wertf".

Note:

You may assume all letters are in lowercase.
If the order is invalid, return an empty string.
There may be multiple valid order of letters, return any one of them is fine.

这道题让给了我们一些按“字母顺序”排列的单词，但是这个字母顺序不是我们熟知的顺序，而是另类的顺序，让我们根据这些“有序”的单词来找出新的字母顺序，这实际上是一道有向图的问题，跟之前的那两道Course Schedule II和Course Schedule的解法很类似，我们先来看BFS的解法，我们需要一个set来保存我们可以推测出来的顺序关系，比如题目中给的例子，我们可以推出的顺序关系有：

```
t->f  
w->e  
r->t  
e->r
```

那么set就用来保存这些pair，我们还需要另一个set来保存所有出现过的字母，需要一个一维数组in来保存每个字母的入度，另外还要一个queue来辅助拓扑遍历，我们先遍历单词集，把所有字母先存入ch，然后我们每两个相邻的单词比较，找出顺序pair，然后我们根据这些pair来赋度，我们把ch中入度为0的字母都排入queue中，然后开始遍历，如果字母在set中存在，则将其pair中对应的字母的入度减1，若此时入度减为0了，则将对应的字母排入queue中并且加入结果res中，直到遍历完成，我们看结果res和ch中的元素个数是否相同，若不相同则说明可能有环存在，返回空字符串，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     string alienOrder(vector<string>& words) {
4         set<pair<char, char> > s;
5         unordered_set<char> ch;
6         vector<int> in(256, 0);
7         queue<char> q;
8         string res = "";
9         for (auto a : words) ch.insert(a.begin(), a.end());
10        for (int i = 0; i < words.size() - 1; ++i) {
11            int mn = min(words[i].size(), words[i + 1].size()), j = 0;
12            for (; j < min(words[i].size(), words[i + 1].size()); ++j) {
13                if (words[i][j] != words[i + 1][j]) {
14                    s.insert(make_pair(words[i][j], words[i + 1][j]));
15                    break;
16                }
17            }
18            if (j == mn && words[i].size() > words[i + 1].size()) return "";
19        }
20        for (auto a : s) ++in[a.second];
21        for (auto a : ch) {
22            if (in[a] == 0) {
23                q.push(a);
24                res += a;
25            }
26        }
27        while (!q.empty()) {
28            char c = q.front(); q.pop();
29            for (auto a : s) {
30                if (a.first == c) {
31                    --in[a.second];
32                    if (in[a.second] == 0) {
33                        q.push(a.second);
34                        res += a.second;
35                    }
36                }
37            }
38        }
39        return res.size() == ch.size() ? res : "";
40    }
41 };

```

下面来看一种DFS的解法，思路和BFS的很类似，我们需要建立一个二维的bool数组g，然后把出现过的字母的对应的位置都赋为true，然后我们把可以推出的顺序的对应位置也赋为true，然后我们就开始递归调用，如果递归函数有返回false的，说明有冲突，则返回false，递归调用结束后结果res中存了入度不为0的字母，最后把入度为0的字母加到最后面，最后把结果res翻转一下即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     string alienOrder(vector<string>& words) {
4         vector<vector<bool>> g(26, vector<bool>(26, false));
5         vector<bool> path(26, false);
6         string res = "";
7         for (string word : words) {
8             for (char c : word) {
9                 g[c - 'a'][c - 'a'] = true;
10            }
11        }
12        for (int i = 0; i < words.size() - 1; ++i) {
13            int mn = min(words[i].size(), words[i + 1].size()), j = 0;
14            for (; j < mn; ++j) {
15                if (words[i][j] != words[i + 1][j]) {
16                    g[words[i][j] - 'a'][words[i + 1][j] - 'a'] = true;
17                    break;
18                }
19            }
20            if (j == mn && words[i].size() > words[i + 1].size()) return "";
21        }
22        for (int i = 0; i < 26; ++i) {
23            if (!dfs(g, i, path, res)) return "";
24        }
25        for (int i = 0; i < 26; ++i) {
26            if (g[i][i]) res += (char)(i + 'a');
27        }
28        return string(res.rbegin(), res.rend());
29    }
30    bool dfs(vector<vector<bool>> &g, int idx, vector<bool> &path, string &res) {
31        if (!g[idx][idx]) return true;
32        path[idx] = true;
33        for (int i = 0; i < 26; ++i) {
34            if (i == idx || !g[idx][i]) continue;
35            if (path[i]) return false;
36            if (!dfs(g, i, path, res)) return false;
37        }
38        path[idx] = false;
39        g[idx][idx] = false;
40        res += (char)(idx + 'a');
41        return true;
42    }
43 };

```

270. 最近的二分搜索树的值

Given a non-empty binary search tree and a target value, find the value in the BST that is closest to the target.

Note:

Given target value is a floating point.

You are guaranteed to have only one unique value in the BST that is closest to the target.

这道题让我们找一个二分搜索数的跟给定值最接近的一个节点值，由于是二分搜索树，所以我最先想到用中序遍历来做一个一个的比较，维护一个最小值，不停的更新，实际上这种方法并没有提高效率，用其他的遍历方法也可以，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     int closestValue(TreeNode* root, double target) {
4         double d = numeric_limits<double>::max();
5         int res = 0;
6         stack<TreeNode*> s;
7         TreeNode *p = root;
8         while (p || !s.empty()) {
9             while (p) {
10                 s.push(p);
11                 p = p->left;
12             }
13             p = s.top(); s.pop();
14             if (d >= abs(target - p->val)) {
15                 d = abs(target - p->val);
16                 res = p->val;
17             }
18             p = p->right;
19         }
20         return res;
21     }
22 };

```

CPP

实际我们可以利用二分搜索树的特点(左<根<右)来快速定位，由于根节点是中间值，我们在往下遍历时，我们根据目标值和根节点的值大小关系来比较，如果目标值小于节点值，则我们应该找更小的值，于是我们到左子树去找，反之我们去右子树找，参见代码如下：

解法2:

```

1 class Solution {
2 public:
3     int closestValue(TreeNode* root, double target) {
4         int res = root->val;
5         while (root) {
6             if (abs(res - target) >= abs(root->val - target)) {
7                 res = root->val;
8             }
9             root = target < root->val ? root->left : root->right;
10        }
11        return res;
12    }
13 };

```

CPP

以上两种方法都是迭代的方法，下面我们来看递归的写法，下面这种递归的写法和上面迭代的方法思路相同，都是根据二分搜索树的性质来优化查找，但是递归的写法用的是回溯法，先遍历到叶节点，然后一层一层的往回走，把最小值一层一层的运回来，参见代码如下：

解法3:

```

1 class Solution {
2 public:
3     int closestValue(TreeNode* root, double target) {
4         int a = root->val;
5         TreeNode *t = target < a ? root->left : root->right;
6         if (!t) return a;
7         int b = closestValue(t, target);
8         return abs(a - target) < abs(b - target) ? a : b;
9     }
10 };

```

再来看另一种递归的写法，思路和上面的都相同，写法上略有不同，用if来分情况，参见代码如下：

解法4：

```

1 class Solution {
2 public:
3     int closestValue(TreeNode* root, double target) {
4         int res = root->val;
5         if (target < root->val && root->left) {
6             int l = closestValue(root->left, target);
7             if (abs(res - target) >= abs(l - target)) res = l;
8         } else if (target > root->val && root->right) {
9             int r = closestValue(root->right, target);
10            if (abs(res - target) >= abs(r - target)) res = r;
11        }
12        return res;
13    }
14 };

```

最后来看一种分治法的写法，这种方法相当于解法一的递归写法，并没有利用到二分搜索树的性质来优化搜索，参见代码如下：

解法5：

```

1 class Solution {
2 public:
3     int closestValue(TreeNode* root, double target) {
4         double diff = numeric_limits<double>::max();
5         int res = 0;
6         helper(root, target, diff, res);
7         return res;
8     }
9     void helper(TreeNode *root, double target, double &diff, int &res) {
10        if (!root) return;
11        if (diff >= abs(root->val - target)) {
12            diff = abs(root->val - target);
13            res = root->val;
14        }
15        helper(root->left, target, diff, res);
16        helper(root->right, target, diff, res);
17    }
18 };

```

271. 加码解码字符串

Design an algorithm to encode a list of strings to a string. The encoded string is then sent over the network and is decoded back to the original list of strings.

Machine 1 (sender) has the function:

```
string encode(vector<string> strs) {  
    // ... your code  
    return encoded_string;  
}
```

Machine 2 (receiver) has the function:

```
vector<string> decode(string s) {  
    //... your code  
    return strs;  
}
```

Note:

The string may contain any possible characters out of 256 valid ascii characters. Your algorithm should be generalized enough to work on any possible characters.

Do not use class member/global/static variables to store states. Your encode and decode algorithms should be stateless.

Do not rely on any library method such as eval or serialize methods. You should implement your own encode/decode algorithm.

这道题让我们给字符串加码再解码，先有码再无码，然后题目中并没有限制我们加码的方法，那么我们的方法只要能成功的把有码变成无码就行了，具体变换方法我们自己设计。由于我们需要把一个字符串集变成一个字符串，然后把这个字符串再还原成原来的字符串集，最开始我想能不能在每一个字符串中间加个空格把它们连起来，然后再按空格来隔开，但是这种方法的问题是原来的一个字符串中如果含有空格，那么还原的时候就会被分隔成两个字符串，所以我们必须还要加上长度的信息，我们的加码方法是长度+"/"+字符串，比如对于"a","ab","abc"，我们就变成"1/a2/ab3/abc"，那么我们解码的时候就有规律可寻，先寻找"/"，然后之前的就是要取出的字符个数，从"/"后取出相应个数即可，以此类推直至没有"/"了，这样我们就得到高清无码的字符串集了，参见代码如下：

解法1：

```

1 class Codec {
2 public:
3     // Encodes a list of strings to a single string.
4     string encode(vector<string>& strs) {
5         string res = "";
6         for (auto a : strs) {
7             res.append(to_string(a.size())).append("/").append(a);
8         }
9         return res;
10    }
11    // Decodes a single string to a list of strings.
12    vector<string> decode(string s) {
13        vector<string> res;
14        int i = 0;
15        while (i < s.size()) {
16            auto found = s.find("/", i);
17            int len = atoi(s.substr(i, found).c_str());
18            res.push_back(s.substr(found + 1, len));
19            i = found + len + 1;
20        }
21        return res;
22    }
23 };

```

上面的方法是用一个变量i来记录当前遍历到的位置，我们也可以通过修改修改s，将已经解码的字符串删掉，最终s变为空的时候停止循环，参见代码如下：

解法2：

```

1 class Codec {
2 public:
3     // Encodes a list of strings to a single string.
4     string encode(vector<string>& strs) {
5         string res = "";
6         for (auto a : strs) {
7             res.append(to_string(a.size())).append("/").append(a);
8         }
9         return res;
10    }
11    // Decodes a single string to a list of strings.
12    vector<string> decode(string s) {
13        vector<string> res;
14        while (!s.empty()) {
15            int found = s.find("/");
16            int len = atoi(s.substr(0, found).c_str());
17            s = s.substr(found + 1);
18            res.push_back(s.substr(0, len));
19            s = s.substr(len);
20        }
21        return res;
22    }
23 };

```

272. 最近的二分搜索树的值之二

Given a non-empty binary search tree and a target value, find k values in the BST that are closest to the target.

Note:

Given target value is a floating point.

You may assume k is always valid, that is: $k \leq \text{total nodes}$.

You are guaranteed to have only one unique set of k values in the BST that are closest to the target.

Follow up:

Assume that the BST is balanced, could you solve it in less than $O(n)$ runtime (where $n = \text{total nodes}$)?

Hint:

1. Consider implement these two helper functions:
 - i. `getPredecessor(N)`, which returns the next smaller node to N.
 - ii. `getSuccessor(N)`, which returns the next larger node to N.
2. Try to assume that each node has a parent pointer, it makes the problem much easier.
3. Without parent pointer we just need to keep track of the path from the root to the current node using a stack.
4. You would need two stacks to track the path in finding predecessor and successor node separately.

这道题是之前那道Closest Binary Search Tree Value的拓展，那道题只让我们找出离目标值最近的一个节点值，而这道题让我们找出离目标值最近的k个节点值，难度瞬间增加了不少，我最先想到的方法是用中序遍历将所有节点值存入到一个一维数组中，由于二分搜索树的性质，这个一维数组是有序的，然后我们再在有序数组中需要和目标值最近的k个值就简单的多，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<int> closestKValues(TreeNode* root, double target, int k) {
4         vector<int> res, v;
5         inorder(root, v);
6         int idx = 0;
7         double diff = numeric_limits<double>::max();
8         for (int i = 0; i < v.size(); ++i) {
9             if (diff >= abs(target - v[i])) {
10                 diff = abs(target - v[i]);
11                 idx = i;
12             }
13         }
14         int left = idx - 1, right = idx + 1;
15         for (int i = 0; i < k; ++i) {
16             res.push_back(v[idx]);
17             if (left >= 0 && right < v.size()) {
18                 if (abs(v[left] - target) > abs(v[right] - target)) {
19                     idx = right;
20                     ++right;
21                 } else {
22                     idx = left;
23                     --left;
24                 }
25             } else if (left >= 0) {
26                 idx = left;
27                 --left;
28             } else if (right < v.size()) {
29                 idx = right;
30                 ++right;
31             }
32         }
33         return res;
34     }
35     void inorder(TreeNode *root, vector<int> &v) {
36         if (!root) return;
37         inorder(root->left, v);
38         v.push_back(root->val);
39         inorder(root->right, v);
40     }
41 };

```

还有一种解法是直接在中序遍历的过程中完成比较，当遍历到一个节点时，如果此时结果数组不到k个，我们直接将此节点值加入res中，如果该节点值和目标值的差值的绝对值小于res的首元素和目标值差值的绝对值，说明当前值更靠近目标值，则将首元素删除，末尾加上当前节点值，反之的话说明当前值比res中所有的值都更偏离目标值，由于中序遍历的特性，之后的值会更加的遍历，所以此时直接返回最终结果即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<int> closestKValues(TreeNode* root, double target, int k) {
4         vector<int> res;
5         inorder(root, target, k, res);
6         return res;
7     }
8     void inorder(TreeNode *root, double target, int k, vector<int> &res) {
9         if (!root) return;
10        inorder(root->left, target, k, res);
11        if (res.size() < k) res.push_back(root->val);
12        else if (abs(root->val - target) < abs(res[0] - target)) {
13            res.erase(res.begin());
14            res.push_back(root->val);
15        } else return;
16        inorder(root->right, target, k, res);
17    }
18 };

```

下面这种方法是上面那种方法的迭代写法，原理一模一样，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     vector<int> closestKValues(TreeNode* root, double target, int k) {
4         vector<int> res;
5         stack<TreeNode*> s;
6         TreeNode *p = root;
7         while (p || !s.empty()) {
8             while (p) {
9                 s.push(p);
10                p = p->left;
11            }
12            p = s.top(); s.pop();
13            if (res.size() < k) res.push_back(p->val);
14            else if (abs(p->val - target) < abs(res[0] - target)) {
15                res.erase(res.begin());
16                res.push_back(p->val);
17            } else break;
18            p = p->right;
19        }
20        return res;
21    }
22 };

```

在来看一种利用最大堆来解题的方法，堆里保存的一个差值diff和节点值的pair，我们中序遍历二叉树(也可以用其他遍历方法)，然后对于每个节点值都计算一下和目标值之差的绝对值，由于最大堆的性质，diff大的自动拍到最前面，我们维护k个pair，如果超过了k个，就把堆前面大的pair删掉，最后留下的k个pair，我们将pair中的节点值取出存入res中返回即可，参见代码如下：

解法4：

```

1 class Solution {
2 public:
3     vector<int> closestKValues(TreeNode* root, double target, int k) {
4         vector<int> res;
5         priority_queue<pair<double, int>> q;
6         inorder(root, target, k, q);
7         while (!q.empty()) {
8             res.push_back(q.top().second);
9             q.pop();
10        }
11        return res;
12    }
13    void inorder(TreeNode *root, double target, int k, priority_queue<pair<double, int>>
14 &q) {
15        if (!root) return;
16        inorder(root->left, target, k, q);
17        q.push({abs(root->val - target), root->val});
18        if (q.size() > k) q.pop();
19        inorder(root->right, target, k, q);
20    }
21};

```

下面的这种方法用了两个栈，pre和suc，其中pre存小于目标值的数，suc存大于目标值的数，开始初始化pre和suc的时候，要分别将最接近目标值的稍小值和稍大值压入pre和suc，然后我们循环k次，每次比较pre和suc的栈顶元素，看谁更接近目标值，将其存入结果res中，然后更新取出元素的栈，依次类推直至取完k个数返回即可，参见代码如下：

解法5：

```

1 class Solution {
2 public:
3     vector<int> closestKValues(TreeNode* root, double target, int k) {
4         vector<int> res;
5         stack<TreeNode*> pre, suc;
6         while (root) {
7             if (root->val <= target) {
8                 pre.push(root);
9                 root = root->right;
10            } else {
11                suc.push(root);
12                root = root->left;
13            }
14        }
15        while (k-- > 0) {
16            if (suc.empty() || !pre.empty() && target - pre.top()->val < suc.top()->val -
17 target) {
18                res.push_back(pre.top()->val);
19                getPredecessor(pre);
20            } else {
21                res.push_back(suc.top()->val);
22                getSuccessor(suc);
23            }
24        }
25        return res;
26    }
27    void getPredecessor(stack<TreeNode*> &pre) {
28        TreeNode *t = pre.top(); pre.pop();
29        if (t->left) {
30            pre.push(t->left);
31            while (pre.top()->right) pre.push(pre.top()->right);
32        }
33    }
34    void getSuccessor(stack<TreeNode*> &suc) {
35        TreeNode *t = suc.top(); suc.pop();
36        if (t->right) {
37            suc.push(t->right);
38            while (suc.top()->left) suc.push(suc.top()->left);
39        }
40    }
41 };

```

273. 整数转为英文单词

Convert a non-negative integer to its english words representation. Given input is guaranteed to be less than 231 - 1.

For example,

123 -> "One Hundred Twenty Three"

12345 -> "Twelve Thousand Three Hundred Forty Five"

1234567 -> "One Million Two Hundred Thirty Four Thousand Five Hundred Sixty Seven"

Hint:

Did you see a pattern in dividing the number into chunk of words? For example, 123 and 123000. Group the number by thousands (3 digits). You can write a helper function that takes a number less than 1000 and convert just that chunk to words.

There are many edge cases. What are some good test cases? Does your code work with input such as 0? Or 1000010? (middle chunk is zero and should not be printed out)

这道题让我们把一个整型数转为用英文单词描述，就像在check上写钱数的方法，我最开始的方法特别复杂，因为我用了几个switch语句来列出所有的单词，但是我看网上大神们的解法都是用数组来枚举的，特别的巧妙而且省地方，膜拜学习中。题目中给足了提示，首先告诉我们要3个一组的进行处理，而且题目中限定了输入数字范围为0到231 - 1之间，最高只能到billion位，3个一组也只需处理四组即可，那么我们需要些一个处理三个一组数字的函数，我们需要把1到19的英文单词都列出来，放到一个数组里，还要把20,30, ... 到90的英文单词列出来放到另一个数组里，然后我们需要用写技巧，比如一个三位数n，百位数表示为n/100，后两位数一起表示为n%100，十位数表示为n%100/10，个位数表示为n%10，然后我们看后两位数是否小于20，小于的话直接从数组中取出单词，如果大于等于20的话，则分别将十位和个位数字的单词从两个数组中取出来。然后再来处理百位上的数字，还要记得加上Hundred。主函数中调用四次这个帮助函数，然后中间要插入"Thousand", "Million", "Billion"到对应的位置，最后check一下末尾是否有空格，把空格都删掉，返回的时候检查下输入是否为0，是的话要返回'Zero'。参见代码如下：

```

1 class Solution {
2 public:
3     string numberToWords(int num) {
4         string res = convertHundred(num % 1000);
5         vector<string> v = {"Thousand", "Million", "Billion"};
6         for (int i = 0; i < 3; ++i) {
7             num /= 1000;
8             res = num % 1000 ? convertHundred(num % 1000) + " " + v[i] + " " + res : res;
9         }
10        while (res.back() == ' ') res.pop_back();
11        return res.empty() ? "Zero" : res;
12    }
13    string convertHundred(int num) {
14        vector<string> v1 = {"", "One", "Two", "Three", "Four", "Five", "Six", "Seven",
15 "Eight", "Nine", "Ten", "Eleven", "Twelve", "Thirteen", "Fourteen", "Fifteen", "Sixteen",
16 "Seventeen", "Eighteen", "Nineteen"};
17        vector<string> v2 = {"", "", "Twenty", "Thirty", "Forty", "Fifty", "Sixty",
18 "Seventy", "Eighty", "Ninety"};
19        string res;
20        int a = num / 100, b = num % 100, c = num % 10;
21        res = b < 20 ? v1[b] : v2[b / 10] + (c ? " " + v1[c] : "");
22        if (a > 0) res = v1[a] + " Hundred" + (b ? " " + res : "");
        return res;
    }
};

CPP

```

274. 求H指数

Given an array of citations (each citation is a non-negative integer) of a researcher, write a function to compute the researcher's h-index.

According to the definition of h-index on Wikipedia: "A scientist has index h if h of his/her N papers have at least h citations each, and the other N - h papers have no more than h citations each."

For example, given citations = [3, 0, 6, 1, 5], which means the researcher has 5 papers in total and each of them had received 3, 0, 6, 1, 5 citations respectively. Since the researcher has 3 papers with at least 3 citations each and the remaining two with no more than 3 citations each, his h-index is 3.

Note: If there are several possible values for h, the maximum one is taken as the h-index.

这道题让我们求H指数，这个质数是用来衡量研究人员的学术水平的质数，定义为一个人的学术文章有n篇分别被引用了n次，那么H指数就是n。而且wiki上直接给出了算法，可以按照如下方法确定某人的H指数：1、将其发表的所有SCI论文按被引次数从高到低排序；2、从前往后查找排序后的列表，直到某篇论文的序号大于该论文被引次数。所得序号减一即为H指数。我也就没多想，直接按照上面的方法写出了代码：

```

1 class Solution {
2 public:
3     int hIndex(vector<int>& citations) {
4         sort(citations.begin(), citations.end(), greater<int>());
5         for (int i = 0; i < citations.size(); ++i) {
6             if (i >= citations[i]) return i;
7         }
8         return citations.size();
9     }
10 };
11 
```

275. 求H指数之二

Follow up for H-Index: What if the citations array is sorted in ascending order? Could you optimize your algorithm?

Hint:

Expected runtime complexity is in $O(\log n)$ and the input is sorted.

这题是之前那道H-Index 求H指数的拓展，输入数组是有序的，让我们在 $O(\log n)$ 的时间内完成计算，看到这个时间复杂度，应该有很敏锐的意识应该用二分查找法，我们最先初始化left和right为0和数组长度len-1，然后取中间值mid，比较citations[mid]和len-mid做比较，如果前者大，则right移到mid之前，反之right移到mid之后，终止条件是left>right，最后返回len-left即可，参见代码如下：

```

1 class Solution {
2 public:
3     int hIndex(vector<int>& citations) {
4         int len = citations.size(), left = 0, right = len - 1;
5         while (left <= right) {
6             int mid = 0.5 * (left + right);
7             if (citations[mid] == len - mid) return len - mid;
8             else if (citations[mid] > len - mid) right = mid - 1;
9             else left = mid + 1;
10        }
11        return len - left;
12    }
13 };
```

276. 粉刷篱笆

There is a fence with n posts, each post can be painted with one of the k colors.

You have to paint all the posts such that no more than two adjacent fence posts have the same color.

Return the total number of ways you can paint the fence.

Note:

n and k are non-negative integers.

这道题让我们粉刷篱笆，有n个部分需要刷，有k种颜色的油漆，规定了不能有超过两个的相同颜色涂的部分，问我们总共有多少种刷法。那么我们首先来分析一下，如果n=0的话，说明没有需要刷的部分，直接返回0即可，如果n为1的话，那么有几种颜色，就有几种刷法，所以应该返回k，当n=2时，k=2时，我们可以分两种情况来统计，一种是相邻部分没有相同的，一种相同部分有相同的颜色，那么对于没有相同的，对于第一个格子，我们有k种填法，对于下一个相邻的格子，由于不能相同，所以我们只有k-1种填法。而有相同部分颜色的刷法和上一个格子的不同颜色刷法相同，因为我们下一格的颜色和之前那个格子颜色刷成一样的即可，最后总共的刷法就是把不同和相同两个刷法加起来，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int numWays(int n, int k) {
4         if (n == 0) return 0;
5         int same = 0, diff = k;
6         for (int i = 2; i <= n; ++i) {
7             int t = diff;
8             diff = (same + diff) * (k - 1);
9             same = t;
10        }
11        return same + diff;
12    }
13 };

```

CPP

下面这种解法和上面那方法几乎一样，只不过多了一个变量，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int numWays(int n, int k) {
4         if (n == 0) return 0;
5         int same = 0, diff = k, res = same + diff;
6         for (int i = 2; i <= n; ++i) {
7             same = diff;
8             diff = res * (k - 1);
9             res = same + diff;
10        }
11        return res;
12    }
13 };
14

```

CPP

277. 寻找名人

Suppose you are at a party with n people (labeled from 0 to $n - 1$) and among them, there may exist one celebrity. The definition of a celebrity is that all the other $n - 1$ people know him/her but he/she does not know any of them.

Now you want to find out who the celebrity is or verify that there is not one. The only thing you are allowed to do is to ask questions like: "Hi, A. Do you know B?" to get information of whether A knows B. You need to find out the celebrity (or verify there is not one) by asking as few questions as possible (in the asymptotic sense).

You are given a helper function `bool knows(a, b)` which tells you whether A knows B. Implement a function `int findCelebrity(n)`, your function should minimize the number of calls to `knows`.

Note: There will be exactly one celebrity if he/she is in the party. Return the celebrity's label if there is a celebrity in the party. If there is no celebrity, return -1.

这道题让我们在一群人群中寻找名人，所谓名人就是每个人都认识他，他却不认识任何人，限定了只有1个或0个名人，给定了一个API函数，输入a和b，用来判断a是否认识b，让我们尽可能少的调用这个函数，来找出人群中的名人。我最先想的方法是建立一个一维数组用来标记每个人的名人候选状态，开始均初始化为true，表示每个人都是名人候选人，然后我们一个人一个人的验证其是否为名人，对于候选者i，我们遍历所有其他人j，如果i认识j，或者j不认识i，说明i不可能是名人，那么我们标记其为false，然后验证下一个候选者，反之如果i不认识j，或者j认识i，说明j不可能是名人，标记之。对于每个候选者i，如果遍历了一圈而其候选者状态仍为true，说明i就是名人，返回即可，如果遍历完所有人没有找到名人，返回-1，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     int findCelebrity(int n) {
4         vector<bool> candidate(n, true);
5         for (int i = 0; i < n; ++i) {
6             for (int j = 0; j < n; ++j) {
7                 if (candidate[i] && i != j) {
8                     if (knows(i, j) || !knows(j, i)) {
9                         candidate[i] = false;
10                        break;
11                    } else {
12                        candidate[j] = false;
13                    }
14                }
15            }
16            if (candidate[i]) return i;
17        }
18        return -1;
19    }
20 };

```

CPP

我们其实可以不用一维数组来标记每个人的状态，我们对于不是名人的i，直接break，继续检查下一个，但是由于我们没有标记后面候选人的状态，所以有可能会重复调用一些`knows`函数，所以下面这种方法虽然省了空间，但是调用`knows`函数的次数可能会比上面的方法次数要多，参见代码如下：

解法2:

```

1 class Solution {
2 public:
3     int findCelebrity(int n) {
4         for (int i = 0, j = 0; i < n; ++i) {
5             for (j = 0; j < n; ++j) {
6                 if (i != j && (knows(i, j) || !knows(j, i))) break;
7             }
8             if (j == n) return i;
9         }
10    return -1;
11 }
12 };

```

下面这种方法是网上比较流行的一种方法，设定候选人res为0，原理是先遍历一遍，对于遍历到的人i，若候选人res认识i，则将候选人res设为i，完成一遍遍历后，我们来检测候选人res是否真正是名人，我们如果判断不是名人，则返回-1，如果并没有冲突，返回res，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int findCelebrity(int n) {
4         int res = 0;
5         for (int i = 0; i < n; ++i) {
6             if (knows(res, i)) res = i;
7         }
8         for (int i = 0; i < n; ++i) {
9             if (res != i && (knows(res, i) || !knows(i, res))) return -1;
10        }
11    return res;
12 }
13 };

```

278. 第一个坏版本

You are a product manager and currently leading a team to develop a new product. Unfortunately, the latest version of your product fails the quality check. Since each version is developed based on the previous version, all the versions after a bad version are also bad.

Suppose you have n versions [1, 2, ..., n] and you want to find out the first bad one, which causes all the following ones to be bad.

You are given an API bool isBadVersion(version) which will return whether version is bad. Implement a function to find the first bad version. You should minimize the number of calls to the API.

Credits:

Special thanks to @jianchao.li.fighter for adding this problem and creating all test cases.

话说这个叫李建超的哥们太赞了，自从LeetCode开始收费后，大多数的免费题都是这哥们出的，还写了所有的test cases，32个赞。这道题说是有一系列版本，其中有一个版本是坏的，而且后面跟着的全是坏的，给了一个API函数可以用来判定当前版本是否是坏的，让我们尽可能少的调用这个API，找出第一个坏版本。那么这种搜索题最先开始考虑用二分查找法把，效率高嘛。由于这题很有规律，好版本和坏版本一定有个边界，那么我们用二分法来找这个边界，对mid值调用API函数，如果是坏版本，说

明边界在左边，则把mid赋值给right，如果是好版本，则说明边界在右边，则把mid+1赋给left，最后返回left即可。需要注意的是，OJ里有个坑，那就是如果left和right都特别大的话，那么left+right可能会溢出，我们的处理方法就是变成left + (right - left) / 2，很好的避免的溢出问题，参见代码如下：

解法1：

```
1 bool isBadVersion(int version); CPP
2
3 class Solution {
4 public:
5     int firstBadVersion(int n) {
6         int left = 1, right = n;
7         while (left < right) {
8             int mid = left + (right - left) / 2;
9             if (isBadVersion(mid)) right = mid;
10            else left = mid + 1;
11        }
12        return left;
13    }
14};
```

如果习惯了二分搜索法从0开始找，可以用下面的方法：

解法2：

```
1 bool isBadVersion(int version); CPP
2
3 class Solution {
4 public:
5     int firstBadVersion(int n) {
6         int left = 0, right = n - 1;
7         while (left < right) {
8             int mid = left + (right - left) / 2;
9             if (isBadVersion(mid + 1)) right = mid;
10            else left = mid + 1;
11        }
12        return right + 1;
13    }
14};
```

279. 完全平方数

Given a positive integer n, find the least number of perfect square numbers (for example, 1, 4, 9, 16, ...) which sum to n.

For example, given n = 12, return 3 because $12 = 4 + 4 + 4$; given n = 13, return 2 because $13 = 4 + 9$.

Credits:

Special thanks to @jianchao.li.fighter for adding this problem and creating all test cases.

又是超哥一个人辛苦的更新题目，一个人托起LeetCode免费题的一片天空啊，赞一个~ 这道题说是给我们一个正整数，求它最少能由几个完全平方数组成。这道题是考察四平方和定理，to be honest，这是我第一次听说这个定理，天啦撸，我的数学是语文老师教的么？！闲话不多扯，回来做题。先来看第一种很高效的方法，根据四平方和定理，任意一个正整数均可表示为4个整数的平方和，其实是可以表示为4个以内的平方数之和，那么就是说返回结果只有1,2,3或4其中的一个，首先我们将数字化简一下，由于

一个数如果含有因子4，那么我们可以把4都去掉，并不影响结果，比如2和8,3和12等等，返回的结果都相同，读者可自行举更多的栗子。还有一个可以化简的地方就是，如果一个数除以8余7的话，那么肯定是由4个完全平方数组成，这里就不证明了，因为我也不会证明，读者可自行举例验证。那么做完两步后，一个很大的数有可能就会变得很小了，大大减少了运算时间，下面我们就来尝试的将其拆为两个平方数之和，如果拆成功了那么就会返回1或2，因为其中一个平方数可能为0. (注：由于输入的n是正整数，所以不存在两个平方数均为0的情况)。注意下面的!!a + !!b这个表达式，可能很多人不太理解这个的意思，其实很简单，感叹号!表示逻辑取反，那么一个正整数逻辑取反为0，再取反为1，所以用两个感叹号!!的作用就是看a和b是否为正整数，都为正整数的话返回2，只有一个正整数的话返回1，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int numSquares(int n) {
4         while (n % 4 == 0) n /= 4;
5         if (n % 8 == 7) return 4;
6         for (int a = 0; a * a <= n; ++a) {
7             int b = sqrt(n - a * a);
8             if (a * a + b * b == n) {
9                 return !!a + !!b;
10            }
11        }
12        return 3;
13    }
14 }
```

CPP

这道题远不止这一种解法，我们还可以用动态规划Dynamic Programming来做，我们建立一个长度为n+1的一维dp数组，将第一个值初始化为0，其余值都初始化为INT_MAX，i从0循环到n，j从1循环到 $i+j^2 \leq n$ 的位置，然后每次更新 $dp[i+j^2]$ 的值，动态更新dp数组，其中 $dp[i]$ 表示正整数i能少能由多个完全平方数组成，那么我们求n，就是返回 $dp[n]$ 即可，也就是dp数组的最后一个数字，参见代码如下：

解法2：

```

1 // DP
2 class Solution {
3 public:
4     int numSquares(int n) {
5         vector<int> dp(n + 1, INT_MAX);
6         dp[0] = 0;
7         for (int i = 0; i <= n; ++i) {
8             for (int j = 1; i + j * j <= n; ++j) {
9                 dp[i + j * j] = min(dp[i + j * j], dp[i] + 1);
10            }
11        }
12        return dp.back();
13    }
14 }
```

CPP

下面再来看一种DP解法，这种解法跟上面有些不同，上面那种解法是初始化了整个长度为n+1的dp数字，但是初始化的顺序不定的，而这个种方法只初始化了第一个值为0，那么在循环里计算，每次增加一个dp数组的长度，里面那个for循环一次循环结束就算好下一个数由几个完全平方数组成，直到增加到第n+1个，返回即可，想更直观的看这两种DP方法的区别，建议每次循环后都打印出dp数字的值来观察其更新的顺序，参见代码如下：

解法3：

```

1 // DP
2 class Solution {
3 public:
4     int numSquares(int n) {
5         vector<int> dp(1, 0);
6         while (dp.size() <= n) {
7             int m = dp.size(), val = INT_MAX;
8             for (int i = 1; i * i <= m; ++i) {
9                 val = min(val, dp[m - i * i] + 1);
10            }
11            dp.push_back(val);
12        }
13        return dp.back();
14    }
15 };

```

最后我们来介绍一种递归Recursion的解法，这种方法的好处是写法简洁，但是运算效率不敢恭维。我们的目的是遍历所有比n小的完全平方数，然后对n与完全平方数的差值递归调用函数，目的是不断更新最终结果，知道找到最小的那个，参见代码如下：

解法4：

```

1 // Recursion
2 class Solution {
3 public:
4     int numSquares(int n) {
5         int res = n, num = 2;
6         while (num * num <= n) {
7             int a = n / (num * num), b = n % (num * num);
8             res = min(res, a + numSquares(b));
9             ++num;
10        }
11        return res;
12    }
13 };

```

PS：解法二三四的运算效率真的不高，强推解法一，高效又易懂，如果想强行优化后三个算法，可以将解法一的前两个if判断加到后三个的算法的开头，能很大的提高运算效率。

280. 摆动排序

Given an unsorted array nums, reorder it in-place such that $\text{nums}[0] \leq \text{nums}[1] \geq \text{nums}[2] \leq \text{nums}[3] \dots$

For example, given $\text{nums} = [3, 5, 2, 1, 6, 4]$, one possible answer is $[1, 6, 2, 5, 3, 4]$.

这道题让我们求摆动排序，跟Wiggle Sort II相比起来，这道题的条件宽松很多，只因为多了一个等号。由于等号的存在，当数组中有重复数字存在的情况时，也很容易满足题目的要求。这道题我们先来看一种时间复杂度为 $O(nlgn)$ 的方法，思路是先给数组排个序，然后我们只要每次把第三个数和第二个数调换位置，第五个数和第四个数调换位置，以此类推直至数组末尾，这样我们就能完成摆动排序了，参见代码如下：

解法1：

```

1 // Time Complexity O(nlgn)
2 class Solution {
3 public:
4     void wiggleSort(vector<int> &nums) {
5         sort(nums.begin(), nums.end());
6         if (nums.size() <= 2) return;
7         for (int i = 2; i < nums.size(); i += 2) {
8             swap(nums[i], nums[i - 1]);
9         }
10    }
11 };

```

这道题还有一种 $O(n)$ 的解法，根据题目要求的 $\text{nums}[0] \leq \text{nums}[1] \geq \text{nums}[2] \leq \text{nums}[3] \dots$ ，我们可以总结出如下规律：

当*i*为奇数时， $\text{nums}[i] \geq \text{nums}[i - 1]$

当*i*为偶数时， $\text{nums}[i] \leq \text{nums}[i - 1]$

那么我们只要对每个数字，根据其奇偶性，跟其对应的条件比较，如果不符合就和前面的数交换位置即可，参见代码如下：

解法2：

```

1 // Time Complexity O(n)
2 class Solution {
3 public:
4     void wiggleSort(vector<int> &nums) {
5         if (nums.size() <= 1) return;
6         for (int i = 1; i < nums.size(); ++i) {
7             if ((i % 2 == 1 && nums[i] < nums[i - 1]) || (i % 2 == 0 && nums[i] > nums[i - 1])) {
8                 swap(nums[i], nums[i - 1]);
9             }
10        }
11    }
12 };

```

281. 之字形迭代器

Given two 1d vectors, implement an iterator to return their elements alternately.

For example, given two 1d vectors:

```
v1 = [1, 2]
v2 = [3, 4, 5, 6]
```

By calling next repeatedly until hasNext returns false, the order of elements returned by next should be: [1, 3, 2, 4, 5, 6].

Follow up: What if you are given k 1d vectors? How well can your code be extended to such cases?

Clarification for the follow up question - Update (2015-09-18):

The "Zigzag" order is not clearly defined and is ambiguous for $k > 2$ cases. If "Zigzag" does not look right to you, replace "Zigzag" with "Cyclic". For example, given the following input:

```
[1,2,3]
[4,5,6,7]
[8,9]
It should return [1,4,8,2,5,9,3,6,7].
```

这道题让我们写一个之字形迭代器，跟之前那道Flatten 2D Vector有些类似，那道题是横向打印，这道题是纵向打印，虽然方向不同，但是实现思路都是大同小异。我最先想到的方法是用两个变量i和j分别记录两个向量的当前元素位置，初始化为0，然后当 $i \leq j$ 时，则说明需要打印v1数组的元素，反之则打印v2数组中的元素。在hasNext函数中，当i或j打印等于对应数组的长度时，我们将其赋为一个特大值，这样不影响我们打印另一个数组的值，只有当i和j都超过格子数组的长度时，返回false，参见代码如下：

解法1：

```
1 class ZigzagIterator {
2 public:
3     ZigzagIterator(vector<int>& v1, vector<int>& v2) {
4         v.push_back(v1);
5         v.push_back(v2);
6         i = j = 0;
7     }
8     int next() {
9         return i <= j ? v[0][i++] : v[1][j++];
10    }
11    bool hasNext() {
12        if (i >= v[0].size()) i = INT_MAX;
13        if (j >= v[1].size()) j = INT_MAX;
14        return i < v[0].size() || j < v[1].size();
15    }
16 private:
17     vector<vector<int>> v;
18     int i, j;
19 }
```

CPP

下面我们来看另一种解法，这种解法直接在初始化的时候就两个数组按照之字形的顺序存入另一个一位数组中了，那么我们就按顺序打印新数组中的值即可，参见代码如下：

解法2：

```

1 class ZigzagIterator {
2 public:
3     ZigzagIterator(vector<int>& v1, vector<int>& v2) {
4         int n1 = v1.size(), n2 = v2.size(), n = max(n1, n2);
5         for (int i = 0; i < n; ++i) {
6             if (i < n1) v.push_back(v1[i]);
7             if (i < n2) v.push_back(v2[i]);
8         }
9     }
10    int next() {
11        return v[i++];
12    }
13    bool hasNext() {
14        return i < v.size();
15    }
16 private:
17     vector<int> v;
18     int i = 0;
19 };

```

由于题目中的Follow up让我们考虑将输入换成k个数组的情况，那么上面两种解法显然就不适用了，所以我们需要一种通解。我们可以采用queue加iterator的方法，用一个queue里面保存iterator的pair，在初始化的时候，有几个数组就生成几个pair放到queue中，每个pair保存该数组的首位置和尾位置的iterator，在next()函数中，我们取出queue队首的一个pair，如果当前的iterator不等于end()，我们将其下一个位置的iterator和end存入队尾，然后返回当前位置的值。在hasNext()函数中，我们只需要看queue是否为空即可，参见代码如下：

解法3：

```

1 class ZigzagIterator {
2 public:
3     ZigzagIterator(vector<int>& v1, vector<int>& v2) {
4         if (!v1.empty()) q.push(make_pair(v1.begin(), v1.end()));
5         if (!v2.empty()) q.push(make_pair(v2.begin(), v2.end()));
6     }
7     int next() {
8         auto it = q.front().first, end = q.front().second;
9         q.pop();
10        if (it + 1 != end) q.push(make_pair(it + 1, end));
11        return *it;
12    }
13    bool hasNext() {
14        return !q.empty();
15    }
16 private:
17     queue<pair<vector<int>::iterator, vector<int>::iterator>> q;
18 };

```

282. 表达式增加操作符

Given a string that contains only digits 0-9 and a target value, return all possibilities to add operators +, -, or * between the digits so they evaluate to the target value.

Examples:

```
"123", 6 -> ["1+2+3", "1*2*3"]
"232", 8 -> ["2*3+2", "2+3*2"]
"00", 0 -> ["0+0", "0-0", "0*0"]
"3456237490", 9191 -> []
```

这道题给了我们一个只由数字组成的字符串，让我们再其中添加+,-或*号来形成一个表达式，该表达式的计算结果为给定了target值，让我们找出所有符合要求的表达式来。题目中给的几个例子其实并不好，很容易让人误以为是必须拆成个位数字，其实不是的，比如"123"，15能返回"12+3"，说明连着的数字也可以。如果要在过往的题中找一道相似的题，我觉得跟Combination Sum II 组合之和之二很类似。不过这道题要更复杂麻烦一些。还是用递归来解题，我们需要两个变量diff和curNum，一个用来记录将要变化的值，另一个是当前运算后的值，而且它们都需要用long long型的，因为字符串转为int型很容易溢出，所以我们用长整型。对于加和减，diff就是即将要加上的数和即将要减去的数的负值，而对于乘来说稍有些复杂，此时的diff应该是上一次的变化的diff乘以即将要乘上的数，有点不好理解，那我们来举个例子，比如 $2+3*2$ ，即将要运算到乘以2的时候，上次循环的 $curNum = 5$, $diff = 3$ ，而如果我们要算这个乘2的时候，新的变化值diff应为 $3*2=6$ ，而我们要把之前+3操作的结果去掉，再加上新的diff，即 $(5-3)+6=8$ ，即为新表达式 $2+3*2$ 的值，有点难理解，大家自己一步一步推算吧。

还有一点需要注意的是，如果输入为"000",0的话，容易出现以下的错误：

```
Wrong: ["0+0+0", "0+0-0", "0+0*0", "0-0+0", "0-0-0", "0-0*0", "0*0+0", "0*0-0", "0*0*0", "0+00", "0-00",
"0*00", "00+0", "00-0",
"00*0", "000"]

Correct: ["0*0*0", "0*0+0", "0*0-0", "0+0*0", "0+0+0", "0+0-0", "0-0*0", "0-0+0", "0-0-0"]
```

我们可以看到错误的结果中有0开头的字符串出现，明显这不是数字，所以我们要去掉这些情况，过滤方法也很简单，我们只要判断长度大于1且首字符是'0'的字符串，将其滤去即可，参见代码如下：

```

1 class Solution {
2 public:
3     vector<string> addOperators(string num, int target) {
4         vector<string> res;
5         addOperatorsDFS(num, target, 0, 0, "", res);
6         return res;
7     }
8     void addOperatorsDFS(string num, int target, long long diff, long long curNum, string
9 out, vector<string> &res) {
10     if (num.size() == 0 && curNum == target) {
11         res.push_back(out);
12     }
13     for (int i = 1; i <= num.size(); ++i) {
14         string cur = num.substr(0, i);
15         if (cur.size() > 1 && cur[0] == '0') return;
16         string next = num.substr(i);
17         if (out.size() > 0) {
18             addOperatorsDFS(next, target, stoll(cur), curNum + stoll(cur), out + "+" +
19 cur, res);
20             addOperatorsDFS(next, target, -stoll(cur), curNum - stoll(cur), out + "—" +
21 cur, res);
22             addOperatorsDFS(next, target, diff * stoll(cur), (curNum - diff) + diff *
23 stoll(cur), out + "*" + cur, res);
24         } else {
25             addOperatorsDFS(next, target, stoll(cur), stoll(cur), cur, res);
26         }
27     }
28 }

```

283. 移动零

Given an array `nums`, write a function to move all 0's to the end of it while maintaining the relative order of the non-zero elements.

For example, given `nums = [0, 1, 0, 3, 12]`, after calling your function, `nums` should be `[1, 3, 12, 0, 0]`.

Note:

You must do this in-place without making a copy of the array.

Minimize the total number of operations.

这道题让我们将一个给定数组中所有的0都移到后面，把非零数前移，要求不能改变非零数的相对应的位置关系，而且不能拷贝额外的数组，那么只能用替换法in-place来做，需要用两个指针，一个不停的向后扫，找到非零位置，然后和前面那个指针交换位置即可，参见下面的代码：

解法1：

```

1 class Solution {
2 public:
3     void moveZeroes(vector<int>& nums) {
4         for (int i = 0, j = 0; i < nums.size(); ++i) {
5             if (nums[i]) {
6                 swap(nums[i], nums[j++]);
7             }
8         }
9     }
10 };

```

下面这种解法的思路跟上面的没啥区别，写法稍稍复杂了一点。

解法2：

```

1 class Solution {
2 public:
3     void moveZeroes(vector<int>& nums) {
4         int left = 0, right = 0;
5         while (right < nums.size()) {
6             if (nums[right]) {
7                 swap(nums[left++], nums[right]);
8             }
9             ++right;
10        }
11    }
12 };

```

284. 顶端迭代器

Given an Iterator class interface with methods: next() and hasNext(), design and implement a PeekingIterator that support the peek() operation -- it essentially peek() at the element that will be returned by the next call to next().

Here is an example. Assume that the iterator is initialized to the beginning of the list: [1, 2, 3].

Call next() gets you 1, the first element in the list.

Now you call peek() and it returns 2, the next element. Calling next() after that still return 2.

You call next() the final time and it returns 3, the last element. Calling hasNext() after that should return false.

Hint:

Think of "looking ahead". You want to cache the next element.

Is one variable sufficient? Why or why not?

Test your design with call order of peek() before next() vs next() before peek().

For a clean implementation, check out Google's guava library source code.

Follow up: How would you extend your design to be generic and work with all types, not just integer?

这道题让我们实现一个顶端迭代器，在普通的迭代器类Iterator的基础上增加了peek的功能，就是返回查看下一个值的功能，但是不移动指针，next()函数才会移动指针，那我们可以定义一个变量专门来保存下一个值，再用一个bool型变量标记是否保存了下一个值，再调用原来的一些成员函数，就可以实现这个顶端迭代器了，参见代码如下：

CPP

```

1  class Iterator {
2      struct Data;
3      Data* data;
4  public:
5      Iterator(const vector<int>& nums);
6      Iterator(const Iterator& iter);
7      virtual ~Iterator();
8      // Returns the next element in the iteration.
9      int next();
10     // Returns true if the iteration has more elements.
11     bool hasNext() const;
12 };
13
14
15 class PeekingIterator : public Iterator {
16 public:
17     PeekingIterator(const vector<int>& nums) : Iterator(nums) {
18         // Initialize any member here.
19         // **DO NOT** save a copy of nums and manipulate it directly.
20         // You should only use the Iterator interface methods.
21         _flag = false;
22     }
23
24     // Returns the next element in the iteration without advancing the iterator.
25     int peek() {
26         if (!_flag) {
27             _value = Iterator::next();
28             _flag = true;
29         }
30         return _value;
31     }
32
33     // hasNext() and next() should behave the same as in the Iterator interface.
34     // Override them if needed.
35     int next() {
36         if (!_flag) return Iterator::next();
37         _flag = false;
38         return _value;
39     }
40
41     bool hasNext() const {
42         if (_flag) return true;
43         if (Iterator::hasNext()) return true;
44         return false;
45     }
46 private:
47     int _value;
48     bool _flag;
49 };

```

285. 二叉搜索树中的中序后继节点

Given a binary search tree and a node in it, find the in-order successor of that node in the BST.

Note: If the given node has no in-order successor in the tree, return null.

这道题让我们求二叉搜索树的某个节点的中序后继节点，那么我们根据BST的性质知道其中序遍历的结果是有序的，是我最先用的方法是用迭代的中序遍历方法，然后用一个bool型的变量b，初始化为false，我们进行中序遍历，对于遍历到的节点，我们首先看如果此时b已经为true，说明之前遍历到了p，那么此时我们返回当前节点，如果b仍为false，我们看遍历到的节点和p是否相同，如果相同，我们此时将b赋为true，那么下一个遍历到的节点就能返回了，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     TreeNode* inorderSuccessor(TreeNode* root, TreeNode* p) {
4         stack<TreeNode*> s;
5         bool b = false;
6         TreeNode *t = root;
7         while (t || !s.empty()) {
8             while (t) {
9                 s.push(t);
10                t = t->left;
11            }
12            t = s.top(); s.pop();
13            if (b) return t;
14            if (t == p) b = true;
15            t = t->right;
16        }
17        return NULL;
18    }
19 };

```

CPP

下面这种方法是用的中序遍历的递归写法，我们需要两个全局变量pre和suc，分别用来记录祖先节点和后继节点，我们初始化将他们都赋为NULL，然后在进行递归中序遍历时，对于遍历到的节点，我们首先看pre和p是否相同，如果相同，则suc赋为当前节点，然后将pre赋为root，那么在遍历下一个节点时，pre就起到记录上一个节点的作用，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     TreeNode* inorderSuccessor(TreeNode* root, TreeNode* p) {
4         if (!p) return NULL;
5         inorder(root, p);
6         return suc;
7     }
8     void inorder(TreeNode *root, TreeNode *p) {
9         if (!root) return;
10        inorder(root->left, p);
11        if (pre == p) suc = root;
12        pre = root;
13        inorder(root->right, p);
14    }
15 private:
16     TreeNode *pre = NULL, *suc = NULL;
17 };

```

再来看一种更简单的方法，这种方法充分地利用到了BST的性质，我们首先看根节点值和p节点值的大小，如果根节点值大，说明p节点肯定在左子树中，那么此时我们先将res赋为root，然后root移到其左子节点，循环的条件是root存在，我们再比较此时root值和p节点值的大小，如果还是root值大，我们重复上面的操作，如果p节点值，那么我们将root移到其右子节点，这样当root为空时，res指向的就是p的后继节点，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     TreeNode* inorderSuccessor(TreeNode* root, TreeNode* p) {
4         TreeNode *res = NULL;
5         while (root) {
6             if (root->val > p->val) {
7                 res = root;
8                 root = root->left;
9             } else root = root->right;
10        }
11        return res;
12    }
13 };

```

上面那种方法也可以写成递归形式，写法也比较简洁，但是需要把思路理清，当根节点值小于等于p节点值，说明p的后继节点一定在右子树中，所以对右子节点递归调用此函数，如果根节点值大于p节点值，那么有可能根节点就是p的后继节点，或者左子树中的某个节点是p的后继节点，所以先对左子节点递归调用此函数，如果返回空，说明根节点是后继节点，返回即可，如果不为空，则将那个节点返回，参见代码如下：

解法4：

```

1 class Solution {
2 public:
3     TreeNode* inorderSuccessor(TreeNode* root, TreeNode* p) {
4         if (!root) return NULL;
5         if (root->val <= p->val) {
6             return inorderSuccessor(root->right, p);
7         } else {
8             TreeNode *left = inorderSuccessor(root->left, p);
9             return left ? left : root;
10        }
11    }
12 };

```

286. 墙和门

You are given a $m \times n$ 2D grid initialized with these three possible values.

-1 - A wall or an obstacle.

0 - A gate.

INF - Infinity means an empty room. We use the value $2^{31} - 1 = 2147483647$ to represent INF as you may assume that the distance to a gate is less than 2147483647.

Fill each empty room with the distance to its nearest gate. If it is impossible to reach a gate, it should be filled with INF.

For example, given the 2D grid:

```

INF -1 0 INF
INF INF INF -1
INF -1 INF -1
0 -1 INF INF

```

After running your function, the 2D grid should be:

```

3 -1 0 1
2 2 1 -1
1 -1 2 -1
0 -1 3 4

```

这道题类似一种迷宫问题，规定了-1表示墙，0表示门，让求每个点到门的最近的曼哈顿距离，这其实类似于求距离场Distance Map的问题，那么我们先考虑用DFS来解，思路是，我们搜索0的位置，每找到一个0，以其周围四个相邻点为起点，开始DFS遍历，并带入深度值1，如果遇到的值大于当前深度值，我们将位置值赋为当前深度值，并对当前点的四个相邻点开始DFS遍历，注意此时深度值需要加1，这样遍历完成后，所有的位置就被正确地更新了，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     void wallsAndGates(vector<vector<int>>& rooms) {
4         for (int i = 0; i < rooms.size(); ++i) {
5             for (int j = 0; j < rooms[i].size(); ++j) {
6                 if (rooms[i][j] == 0) dfs(rooms, i, j, 0);
7             }
8         }
9     }
10    void dfs(vector<vector<int>>& rooms, int i, int j, int val) {
11        if (i < 0 || i >= rooms.size() || j < 0 || j >= rooms[i].size() || rooms[i][j] <
12 val) return;
13        rooms[i][j] = val;
14        dfs(rooms, i + 1, j, val + 1);
15        dfs(rooms, i - 1, j, val + 1);
16        dfs(rooms, i, j + 1, val + 1);
17        dfs(rooms, i, j - 1, val + 1);
18    }
19 };

```

那么下面我们再来看BFS的解法，需要借助queue，我们首先把门的位置都排入queue中，然后开始循环，对于门位置的四个相邻点，我们判断其是否在矩阵范围内，并且位置值是否大于上一位置的值加1，如果满足这些条件，我们将当前位置赋为上一位置加1，并将次位置排入queue中，这样等queue中的元素遍历完了，所有位置的值就被正确地更新了，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     void wallsAndGates(vector<vector<int>>& rooms) {
4         queue<pair<int, int>> q;
5         vector<vector<int>> dirs{{0, -1}, {-1, 0}, {0, 1}, {1, 0}};
6         for (int i = 0; i < rooms.size(); ++i) {
7             for (int j = 0; j < rooms[i].size(); ++j) {
8                 if (rooms[i][j] == 0) q.push({i, j});
9             }
10        }
11        while (!q.empty()) {
12            int i = q.front().first, j = q.front().second; q.pop();
13            for (int k = 0; k < dirs.size(); ++k) {
14                int x = i + dirs[k][0], y = j + dirs[k][1];
15                if (x < 0 || x >= rooms.size() || y < 0 || y >= rooms[0].size() || rooms[x]
16 [y] < rooms[i][j] + 1) continue;
17                rooms[x][y] = rooms[i][j] + 1;
18                q.push({x, y});
19            }
20        }
21    }
22 };

```

287. 寻找重复数

Given an array `nums` containing $n + 1$ integers where each integer is between 1 and n (inclusive), prove that at least one duplicate element must exist. Assume that there is only one duplicate number, find the duplicate one.

Note:

You must not modify the array (assume the array is read only).

You must use only constant extra space.

Your runtime complexity should be less than $O(n^2)$.

这道题给了我们 $n+1$ 个数，所有的数都在 $[1, n]$ 区域内，首先让我们证明必定会有一个重复数，这不禁让我想起了小学华罗庚奥数中的抽屉原理(又叫鸽巢原理)，即如果有十个苹果放到九个抽屉里，如果苹果全在抽屉里，则至少有一个抽屉里有两个苹果，这里就不证明了，直接来做题吧。题目要求我们不能改变原数组，即不能给原数组排序，又不能用多余空间，那么哈希表神马的也就不用考虑了，又说时间小于 $O(n^2)$ ，也就不能用brute force的方法，那我们也就只能考虑用二分搜索法了，我们在区别 $[1, n]$ 中搜索，首先求出中点mid，然后遍历整个数组，统计所有小于等于mid的数的个数，如果个数大于mid，则说明重复值在 $[mid+1, n]$ 之间，反之，重复值应在 $[1, mid-1]$ 之间，然后依次类推，直到搜索完成，此时的low就是我们要求的重复值，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int findDuplicate(vector<int>& nums) {
4         int low = 1, high = nums.size() - 1;
5         while (low < high) {
6             int mid = low + (high - low) * 0.5;
7             int cnt = 0;
8             for (auto a : nums) {
9                 if (a <= mid) ++cnt;
10            }
11            if (cnt <= mid) low = mid + 1;
12            else high = mid;
13        }
14        return low;
15    }
16 };

```

CPP

经过热心网友waruzhi的留言提醒还有一种 $O(n)$ 的解法，并给了参考帖子，发现真是一种不错的解法，其核心思想快慢指针在之前的题目Linked List Cycle II中就有应用，这里应用的更加巧妙一些，由于题目限定了区间 $[1, n]$ ，所以可以巧妙的利用坐标和数值之间相互转换，而由于重复数字的存在，那么一定会形成环，我们用快慢指针可以找到环并确定环的起始位置，确实是太巧妙了！

解法2：

```

1 class Solution {
2 public:
3     int findDuplicate(vector<int>& nums) {
4         int slow = 0, fast = 0, t = 0;
5         while (true) {
6             slow = nums[slow];
7             fast = nums[nums[fast]];
8             if (slow == fast) break;
9         }
10        while (true) {
11            slow = nums[slow];
12            t = nums[t];
13            if (slow == t) break;
14        }
15        return slow;
16    }
17 };

```

288. 独特的单词缩写

An abbreviation of a word follows the form <first letter><number><last letter>. Below are some examples of word abbreviations:

a) it --> it (no abbreviation)

1

b) d|o|g --> d1g

1 1 1

1---5----0----5--8

c) i|nternationalizatio|n --> i18n

1

1---5----0

d) l|ocalizatio|n --> l10n

Assume you have a dictionary and given a word, find whether its abbreviation is unique in the dictionary. A word's abbreviation is unique if no other word from the dictionary has the same abbreviation.

Example:

Given dictionary = ["deer", "door", "cake", "card"]

```

isUnique("dear") -> false
isUnique("cart") -> true
isUnique("cane") -> false
isUnique("make") -> true

```

这道题让我们求独特的单词缩写，但是题目中给的例子不是很清晰，我们来看下面三种情况：

1. dictionary = {"dear"}, isUnique("door") -> false
2. dictionary = {"door", "door"}, isUnique("door") -> true
3. dictionary = {"dear", "door"}, isUnique("door") -> false

从上面三个例子我们可以看出，当缩写一致的时候，字典中的单词均和给定单词相同时，那么返回true。我们需要用哈希表来建立缩写形式和其对应的单词的映射，把所有缩写形式的相同单词放到一个set中，然后我们在判断是否unique的时候只需要看给定单词的缩写形式的set里面该单词的个数是否和set中的元素总数相同，相同的话就是上面的第二种情况，返回true。需要注意的是由于set中不能有重复值，所有上面第二种情况只会有一个door存在set里，但是并不影响判断结果，参见代码如下：

解法1：

```
1 class ValidWordAbbr {
2 public:
3     ValidWordAbbr(vector<string> &dictionary) {
4         for (auto a : dictionary) {
5             string k = a[0] + to_string(a.size() - 2) + a.back();
6             m[k].insert(a);
7         }
8     }
9     bool isUnique(string word) {
10        string k = word[0] + to_string(word.size() - 2) + word.back();
11        return m[k].count(word) == m[k].size();
12    }
13 private:
14     unordered_map<string, set<string>> m;
15 }
```

CPP

如果我们想省一些空间，也可以不用set，那么我们如何区分上面的第二和第三种情况呢，我们在遇到哈希表中没有当前缩写形式的时候，将该缩写形式和当前单词建立映射，如果该缩写形式已经存在，那么我们看如果映射的单词不是当前单词，我们将映射单词改为空字符串，这样做的原因是，在对于第三种情况dictionary = {"dear", "door"}时，遍历dear时，建立d2r和dear的映射，当遍历到door的时候，由于door和dear不同，我们将映射改为d2r和“”映射，而对于第二种情况 dictionary = {"door", "door"}，保留d2r和door的映射，那么这样在判断door是否unique时，就可以区别第二种和第三种情况了，参见代码如下：

解法2：

```
1 class ValidWordAbbr {
2 public:
3     ValidWordAbbr(vector<string> &dictionary) {
4         for (auto a : dictionary) {
5             string k = a[0] + to_string(a.size() - 2) + a.back();
6             if (m.find(k) != m.end() && m[k] != a) m[k] = "";
7             else m[k] = a;
8         }
9     }
10    bool isUnique(string word) {
11        string k = word[0] + to_string(word.size() - 2) + word.back();
12        return m.find(k) == m.end() || m[k] == word;
13    }
14 private:
15     unordered_map<string, string> m;
16 }
```

CPP

289. 生命游戏

According to the Wikipedia's article: "The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970."

Given a board with m by n cells, each cell has an initial state live (1) or dead (0). Each cell interacts with its eight neighbors (horizontal, vertical, diagonal) using the following four rules (taken from the above Wikipedia article):

Any live cell with fewer than two live neighbors dies, as if caused by under-population.

Any live cell with two or three live neighbors lives on to the next generation.

Any live cell with more than three live neighbors dies, as if by over-population..

Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

Write a function to compute the next state (after one update) of the board given its current state.

Follow up:

Could you solve it in-place? Remember that the board needs to be updated at the same time: You cannot update some cells first and then use their updated values to update other cells.

In this question, we represent the board using a 2D array. In principle, the board is infinite, which would cause problems when the active area encroaches the border of the array. How would you address these problems?

这道题是有名的康威生命游戏，而我又是第一次听说这个东东，这是一种细胞自动机，每一个位置有两种状态，1为活细胞，0为死细胞，对于每个位置都满足如下的条件：

1. 如果活细胞周围八个位置的活细胞数少于两个，则该位置活细胞死亡
2. 如果活细胞周围八个位置有两个或三个活细胞，则该位置活细胞仍然存活
3. 如果活细胞周围八个位置有超过三个活细胞，则该位置活细胞死亡
4. 如果死细胞周围正好有三个活细胞，则该位置死细胞复活

由于题目中要求我们用置换方法in-place来解题，所以我们就不能新建一个相同大小的数组，那么我们只能更新原有数组，但是题目中要求所有的位置必须被同时更新，但是在循环程序中我们还是一个位置一个位置更新的，那么当一个位置更新了，这个位置成为其他位置的neighbor时，我们怎么知道其未更新的状态呢，我们可以使用状态机转换：

状态0：死细胞转为死细胞

状态1：活细胞转为活细胞

状态2：活细胞转为死细胞

状态3：死细胞转为活细胞

最后我们对所有状态对2取余，那么状态0和2就变成死细胞，状态1和3就是活细胞，达成目的。我们先对原数组进行逐个扫描，对于每一个位置，扫描其周围八个位置，如果遇到状态1或2，就计数器累加1，扫完8个邻居，如果少于两个活细胞或者大于三个活细胞，而且当前位置是活细胞的话，标记状态2，如果正好有三个活细胞且当前是死细胞的话，标记状态3。完成一遍扫描后再对数据扫描一遍，对2取余变成我们想要的结果。参见代码如下：

```

1 class Solution {
2 public:
3     void gameOfLife(vector<vector<int>>& board) {
4         int m = board.size(), n = m ? board[0].size() : 0;
5         int dx[] = {-1, -1, -1, 0, 1, 1, 1, 0};
6         int dy[] = {-1, 0, 1, 1, 1, 0, -1, -1};
7         for (int i = 0; i < m; ++i) {
8             for (int j = 0; j < n; ++j) {
9                 int cnt = 0;
10                for (int k = 0; k < 8; ++k) {
11                    int x = i + dx[k], y = j + dy[k];
12                    if (x >= 0 && x < m && y >= 0 && y < n && (board[x][y] == 1 || board[x]
13 [y] == 2)) {
14                        ++cnt;
15                    }
16                }
17                if (board[i][j] && (cnt < 2 || cnt > 3)) board[i][j] = 2;
18                else if (!board[i][j] && cnt == 3) board[i][j] = 3;
19            }
20        }
21        for (int i = 0; i < m; ++i) {
22            for (int j = 0; j < n; ++j) {
23                board[i][j] %= 2;
24            }
25        }
26    }
27 };

```

290. 词语模式

Given a pattern and a string str, find if str follows the same pattern.

Examples:

pattern = "abba", str = "dog cat cat dog" should return true.
 pattern = "abba", str = "dog cat cat fish" should return false.
 pattern = "aaaa", str = "dog cat cat dog" should return false.
 pattern = "abba", str = "dog dog dog dog" should return false.

Notes:

Both pattern and str contains only lowercase alphabetical letters.

Both pattern and str do not have leading or trailing spaces.

Each word in str is separated by a single space.

Each letter in pattern must map to a word with length that is at least 1.

这道题给我们一个模式字符串，又给我们一个单词字符串，让我们求单词字符串中单词出现的规律是否符合模式字符串中的规律。那么首先想到就是用哈希表来做，建立模式字符串中每个字符和单词字符串每个单词之间的映射，而且这种映射必须是一对一关系的，不能'a'和'b'同时对应'dog'，所以我们在碰到一个新字符时，首先检查其是否在哈希表中出现，若出现，其映射的单词若不是此时对应的单词，则返回false。如果没有在哈希表中出现，我们还要遍历一遍哈希表，看新遇到的单词是否已经是哈希表中的映射，如果没有，再跟新遇到的字符建立映射，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     bool wordPattern(string pattern, string str) {
4         unordered_map<char, string> m;
5         istringstream in(str);
6         int i = 0;
7         for (string word; in >> word; ++i) {
8             if (m.find(pattern[i]) != m.end()) {
9                 if (m[pattern[i]] != word) return false;
10            } else {
11                for (unordered_map<char, string>::iterator it = m.begin(); it != m.end();
12                     ++it) {
13                    if (it->second == word) return false;
14                }
15                m[pattern[i]] = word;
16            }
17        }
18        return i == pattern.size();
19    }
20};

```

当然这道题也可以用两个哈希表来完成，分别将字符和单词都映射到当前的位置，那么我们在遇到新字符和单词时，首先看两个哈希表是否至少有一个映射存在，如果有一个存在，则比较两个哈希表映射值是否相同，不同则返回false。如果两个表都不存在映射，则同时添加两个映射，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool wordPattern(string pattern, string str) {
4         unordered_map<char, int> m1;
5         unordered_map<string, int> m2;
6         istringstream in(str);
7         int i = 0;
8         for (string word; in >> word; ++i) {
9             if (m1.find(pattern[i]) != m1.end() || m2.find(word) != m2.end()) {
10                 if (m1[pattern[i]] != m2[word]) return false;
11             } else {
12                 m1[pattern[i]] = m2[word] = i + 1;
13             }
14         }
15         return i == pattern.size();
16     }
17 };

```

291. 词语模式之二

Given a pattern and a string str, find if str follows the same pattern.

Here follow means a full match, such that there is a bijection between a letter in pattern and a non-empty substring in str.

Examples:

```
pattern = "abab", str = "redblueredblue" should return true.
pattern = "aaaa", str = "asdadasdasd" should return true.
pattern = "aabb", str = "xyzabcxzyabc" should return false.
```

这道题是之前那道Word Pattern的拓展，之前那道题词语之间都有空格隔开，这样我们可以一个单词一个单词的读入，然后来判断是否符合给定的特征，而这道题没有空格了，那么难度就大大的增加了，因为我们不知道对应的单词是什么，所以得自行分开，那么我们可以用回溯法来生成每一种情况来判断，我们还是需要用哈希表来建立模式字符和单词之间的映射，我们还需要用变量p和r来记录当前递归到的模式字符和单词串的位置，在递归函数中，如果p和r分别等于模式字符串和单词字符串的长度，说明此时匹配成功结束了，返回ture，反之如果一个达到了而另一个没有，说明匹配失败了，返回false。如果不满足上述条件的话，我们取出当前位置的模式字符，然后从单词串的r位置开始往后遍历，每次取出一个单词，如果模式字符已经存在哈希表中，而且对应的单词和取出的单词也相等，那么我们再次调用递归函数在下一个位置，如果返回true，那么我们就返回true。反之如果该模式字符不在哈希表中，我们要看有没有别的模式字符已经映射了当前取出的单词，如果没有的话，我们建立新的映射，并且调用递归函数，注意如果递归函数返回false了，我们要在哈希表中删去这个映射，参见代码如下：

解法1：

CPP

```
1 class Solution {
2 public:
3     bool wordPatternMatch(string pattern, string str) {
4         unordered_map<char, string> m;
5         return helper(pattern, 0, str, 0, m);
6     }
7     bool helper(string pattern, int p, string str, int r, unordered_map<char, string> &m) {
8         if (p == pattern.size() && r == str.size()) return true;
9         if (p == pattern.size() || r == str.size()) return false;
10        char c = pattern[p];
11        for (int i = r; i < str.size(); ++i) {
12            string t = str.substr(r, i - r + 1);
13            if (m.count(c) && m[c] == t) {
14                if (helper(pattern, p + 1, str, i + 1, m)) return true;
15            } else if (!m.count(c)) {
16                bool b = false;
17                for (auto it : m) {
18                    if (it.second == t) b = true;
19                }
20                if (!b) {
21                    m[c] = t;
22                    if (helper(pattern, p + 1, str, i + 1, m)) return true;
23                    m.erase(c);
24                }
25            }
26        }
27        return false;
28    }
29};
```

下面这种方法和上面那种方法很类似，不同点在于使用了set，而使用其的原因也是为了记录所有和模式字符建立过映射的单词，这样我们就不用每次遍历哈希表了，只要在set中查找取出的单词是否存在，如果存在了则跳过后面的处理，反之则进行和上面相同的处理，注意还要在set中插入新的单词，最后也要同时删除掉，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool wordPatternMatch(string pattern, string str) {
4         unordered_map<char, string> m;
5         set<string> s;
6         return helper(pattern, 0, str, 0, m, s);
7     }
8     bool helper(string pattern, int p, string str, int r, unordered_map<char, string> &m,
9     set<string> &s) {
10     if (p == pattern.size() && r == str.size()) return true;
11     if (p == pattern.size() || r == str.size()) return false;
12     char c = pattern[p];
13     for (int i = r; i < str.size(); ++i) {
14         string t = str.substr(r, i - r + 1);
15         if (m.count(c) && m[c] == t) {
16             if (helper(pattern, p + 1, str, i + 1, m, s)) return true;
17         } else if (!m.count(c)) {
18             if (s.count(t)) continue;
19             m[c] = t;
20             s.insert(t);
21             if (helper(pattern, p + 1, str, i + 1, m, s)) return true;
22             m.erase(c);
23             s.erase(t);
24         }
25     }
26     return false;
27 }
};
```

再来看一种不写helper函数的解法，可以调用自身，思路和上面的方法完全相同，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     bool wordPatternMatch(string pattern, string str) {
4         if (pattern.empty()) return str.empty();
5         if (m.count(pattern[0])) {
6             string t = m[pattern[0]];
7             if (t.size() > str.size() || str.substr(0, t.size()) != t) return false;
8             if (wordPatternMatch(pattern.substr(1), str.substr(t.size()))) return true;
9         } else {
10            for (int i = 1; i <= str.size(); ++i) {
11                if (s.count(str.substr(0, i))) continue;
12                m[pattern[0]] = str.substr(0, i);
13                s.insert(str.substr(0, i));
14                if (wordPatternMatch(pattern.substr(1), str.substr(i))) return true;
15                m.erase(pattern[0]);
16                s.erase(str.substr(0, i));
17            }
18        }
19        return false;
20    }
21    unordered_map<char, string> m;
22    unordered_set<string> s;
23 };
24

```

292. 尼姆游戏

You are playing the following Nim Game with your friend: There is a heap of stones on the table, each time one of you take turns to remove 1 to 3 stones. The one who removes the last stone will be the winner. You will take the first turn to remove the stones.

Both of you are very clever and have optimal strategies for the game. Write a function to determine whether you can win the game given the number of stones in the heap.

For example, if there are 4 stones in the heap, then you will never win the game: no matter 1, 2, or 3 stones you remove, the last stone will always be removed by your friend.

Hint:

If there are 5 stones in the heap, could you figure out a way to remove the stones such that you will always be the winner?

有史以来最少代码量的解法，虽然解法很简单，但是题目还是蛮有意思的，题目说给我们一堆石子，每次可以拿一个两个或三个，两个人轮流拿，拿到最后一个石子的人获胜，现在给我们一堆石子的个数，问我们能不能赢。那么我们就从最开始分析，由于是我们先拿，那么3个以内(包括3个)的石子，我们直接赢，如果共4个，那么我们一定输，因为不管我们取几个，下一个人一次都能取完。如果共5个，我们赢，因为我们可以取一个，然后变成4个让别人取，根据上面的分析我们赢，所以我们列出1到10个的情况如下：

```

1   Win
2   Win
3   Win
4   Lost
5   Win
6   Win
7   Win
8   Lost
9   Win
10  Win

```

由此我们可以发现规律，只要是4的倍数个，我们一定会输，所以对4取余即可，参见代码如下：

```

1 class Solution {
2 public:
3     bool canWinNim(int n) {
4         return n % 4;
5     }
6 };

```

CPP

讨论：我们来generalize一下这道题，当可以拿 $1 \sim n$ 个石子时，那么个数为 $(n+1)$ 的整数倍时一定会输，我们试着证明一下这个结论，若当前共有 $m^*(n+1)$ 个石子，那么：

当 $m=1$ 时，即剩 $n+1$ 个的时候，肯定会输，因为不管你取 $1 \sim n$ 中的任何一个数字，另一个人都可以取完。

当 $m > 1$ 时，即有 $m^*(n+1)$ 的时候，不管你先取 $1 \sim n$ 中的任何一个数字 x ，另外一个人一定会取 $n+1-x$ 个，这样总数就变成了 $(m-1)^*(n+1)$ ，第二个人就一直按这个策略取，那么直到剩 $n+1$ 个的时候，就便变成 $m=1$ 的情况，一定会输。

[293. 翻转游戏](#)

You are playing the following Flip Game with your friend: Given a string that contains only these two characters: + and -, you and your friend take turns to flip two consecutive "++" into "--". The game ends when a person can no longer make a move and therefore the other person will be the winner.

Write a function to compute all possible states of the string after one valid move.

For example, given $s = "++++"$, after one move, it may become one of the following states:

```
[  
    "--++",  
    "+--+",  
    "++--"  
]
```

If there is no valid move, return an empty list [].

这道题让我们把相邻的两个+变成-, 真不是一道难题, 我们就从第二个字母开始遍历, 每次判断当前字母是否为+, 和之前那个字母是否为+, 如果都为加, 则将翻转后的字符串存入结果中即可, 参见代码如下:

```
1 | class Solution {  
2 | public:  
3 |     vector<string> generatePossibleNextMoves(string s) {  
4 |         vector<string> res;  
5 |         for (int i = 1; i < s.size(); ++i) {  
6 |             if (s[i] == '+' && s[i - 1] == '+') {  
7 |                 res.push_back(s.substr(0, i - 1) + "--" + s.substr(i + 1));  
8 |             }  
9 |         }  
10 |         return res;  
11 |     }  
12 | };
```

CPP

294. 翻转游戏之二

You are playing the following Flip Game with your friend: Given a string that contains only these two characters: + and -, you and your friend take turns to flip two consecutive "++" into "--". The game ends when a person can no longer make a move and therefore the other person will be the winner.

Write a function to determine if the starting player can guarantee a win.

Example:

Input: $s = "++++"$

Output: true

Explanation: The starting player can guarantee a win by flipping the middle "++" to become "+++".

这道题是之前那道Flip Game的拓展, 让我们判断先手的玩家是否能赢, 那么我们可以穷举所有的情况, 用回溯法来解题, 我们的思路跟上面那题类似, 也是从第二个字母开始遍历整个字符串, 如果当前字母和之前那个字母都是+, 那么我们递归调用将这两个位置变为-的字符串, 如果返回false, 说明当前玩家可以赢, 结束循环返回false, 参见代码如下:

解法1:

```

1 class Solution {
2 public:
3     bool canWin(string s) {
4         for (int i = 1; i < s.size(); ++i) {
5             if (s[i] == '+' && s[i - 1] == '+' && !canWin(s.substr(0, i - 1) + "--" +
6 s.substr(i + 1))) {
7                 return true;
8             }
9         }
10    return false;
11 }

```

第二种解法和第一种解法一样，只是用find函数来查找++的位置，然后把位置赋值给i，然后还是递归调用canWin函数，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool canWin(string s) {
4         for (int i = -1; (i = s.find("++", i + 1)) >= 0;) {
5             if (!canWin(s.substr(0, i) + "--" + s.substr(i + 2))) {
6                 return true;
7             }
8         }
9     return false;
10 }
11 }

```

295. 找出数据流的中位数

Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.

Examples:

[2,3,4] , the median is 3

[2,3] , the median is $(2 + 3) / 2 = 2.5$

Design a data structure that supports the following two operations:

`void addNum(int num)` - Add a integer number from the data stream to the data structure.
`double findMedian()` - Return the median of all elements so far.

For example:

```

add(1)
add(2)
findMedian() -> 1.5
add(3)
findMedian() -> 2

```

这道题给我们一个数据流，让我们找出中位数，由于数据流中的数据并不是有序的，所以我们首先应该想个方法让其有序。如果我们用vector来保存数据流的话，每进来一个新数据都要给数组排序，很不高效。所以之后想到用multiset这个数据结构，是有序保存数据的，但是它不能用下标直接访问元素，找中位数也不高效。这里用到的解法十分巧妙，我们使用大小堆来解决问题，其中大堆保存右半段较大的数字，小堆保存左半段较小的数组。这样整个数组就被中间分为两段了，由于堆的保存方式是由大到小，我们希望大堆里面的数据是从小到大，这样取第一个来计算中位数方便。我们用到一个小技巧，就是存到大堆里的数先取反再存，这样由大到小存下来的顺序就是实际上我们想要的从小到大的顺序。当大堆和小堆中的数字一样多时，我们取出大堆小堆的首元素求平均值，当小堆元素多时，取小堆首元素为中位数，参见代码如下：

解法1：

```

1  class MedianFinder {
2  public:
3
4      // Adds a number into the data structure.
5      void addNum(int num) {
6          small.push(num);
7          large.push(-small.top());
8          small.pop();
9          if (small.size() < large.size()) {
10              small.push(-large.top());
11              large.pop();
12          }
13      }
14
15     // Returns the median of current data stream
16     double findMedian() {
17         return small.size() > large.size() ? small.top() : 0.5 * (small.top() -
18         large.top());
19     }
20
21 private:
22     priority_queue<long> small, large;
23 };

```

CPP

上述方法是用priority_queue来实现堆功能的，下面我们还可用multiset来实现堆，参见代码如下：

解法2：

```

1 class MedianFinder {
2 public:
3
4     // Adds a number into the data structure.
5     void addNum(int num) {
6         small.insert(num);
7         large.insert(-*small.begin());
8         small.erase(small.begin());
9         if (small.size() < large.size()) {
10            small.insert(-*large.begin());
11            large.erase(large.begin());
12        }
13    }
14
15    // Returns the median of current data stream
16    double findMedian() {
17        return small.size() > large.size() ? *small.begin() : 0.5 * (*small.begin() -
18 *large.begin());
19    }
20
21 private:
22     multiset<long> small, large;
23 };

```

296. 最佳开会地点

A group of two or more people wants to meet and minimize the total travel distance. You are given a 2D grid of values 0 or 1, where each 1 marks the home of someone in the group. The distance is calculated using Manhattan Distance, where $\text{distance}(p_1, p_2) = |p_2.x - p_1.x| + |p_2.y - p_1.y|$.

For example, given three people living at (0,0), (0,4), and (2,2):

```

1 - 0 - 0 - 0 - 1
|   |   |   |
0 - 0 - 0 - 0 - 0
|   |   |   |
0 - 0 - 1 - 0 - 0

```

The point (0,2) is an ideal meeting point, as the total travel distance of $2+2+2=6$ is minimal. So return 6.

这道题让我们求最佳的开会地点，该地点需要到每个为1的点的曼哈顿距离之和最小，题目中给了我们提示，让我们先从一维的情况来分析，那么我们先看一维时有两个点A和B的情况，

_____ A _____ P _____ B _____

那么我们可以发现，只要开会为位置P在[A, B]区间内，不管在哪，距离之和都是A和B之间的距离，如果P不在[A, B]之间，那么距离之和就会大于A和B之间的距离，那么我们现在再加两个点C和D：

_____ C _____ A _____ P _____ B _____ D _____

我们通过分析可以得出，P点的最佳位置就是在[A, B]区间内，这样和四个点的距离之和为AB距离加上CD距离，在其他任意一点的距离都会大于这个距离，那么分析出来了上述规律，这题就变得很容易了，我们只要给位置排好序，然后用最后一个坐标减去第一个坐标，即CD距离，倒数第二个坐标减去第二个坐标，即AB距离，以此类推，直到最中间停止，那么一维的情况分析出来了，二维的情况就是两个一维相加即可，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     int minTotalDistance(vector<vector<int>>& grid) {
4         vector<int> rows, cols;
5         for (int i = 0; i < grid.size(); ++i) {
6             for (int j = 0; j < grid[i].size(); ++j) {
7                 if (grid[i][j] == 1) {
8                     rows.push_back(i);
9                     cols.push_back(j);
10                }
11            }
12        }
13        return minTotalDistance(rows) + minTotalDistance(cols);
14    }
15    int minTotalDistance(vector<int> v) {
16        int res = 0;
17        sort(v.begin(), v.end());
18        int i = 0, j = v.size() - 1;
19        while (i < j) res += v[j--] - v[i++];
20        return res;
21    }
22 };

```

CPP

我们也可以不用多写一个函数，直接对rows和cols同时处理，稍稍能简化些代码：

解法2:

```

1 class Solution {
2 public:
3     int minTotalDistance(vector<vector<int>>& grid) {
4         vector<int> rows, cols;
5         for (int i = 0; i < grid.size(); ++i) {
6             for (int j = 0; j < grid[i].size(); ++j) {
7                 if (grid[i][j] == 1) {
8                     rows.push_back(i);
9                     cols.push_back(j);
10                }
11            }
12        }
13        sort(cols.begin(), cols.end());
14        int res = 0, i = 0, j = rows.size() - 1;
15        while (i < j) res += rows[j] - rows[i] + cols[j--] - cols[i++];
16        return res;
17    }
18 };

```

CPP

297. 二叉树的序列化和去序列化

Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be reconstructed later in the same or another computer environment.

Design an algorithm to serialize and deserialize a binary tree. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary tree can be serialized to a string and this string can be deserialized to the original tree structure.

For example, you may serialize the following tree

```
1
 / \
2   3
 / \
4   5
```

as "[1,2,3,null,null,4,5]", just the same as how LeetCode OJ serializes a binary tree. You do not necessarily need to follow this format, so please be creative and come up with different approaches yourself.

Note: Do not use class member/global/static variables to store states. Your serialize and deserialize algorithms should be stateless.

这道题让我们对二叉树进行序列化和去序列化的操作。序列化就是将一个数据结构或物体转化为一个位序列，可以存进一个文件或者内存缓冲器中，然后通过网络连接在相同的或者另一个电脑环境中被还原，还原的过程叫做去序列化。现在让我们来序列化和去序列化一个二叉树，并给了我们例子。这题有两种解法，分别为先序遍历的递归解法和层序遍历的非递归解法。先来看先序遍历的递归解法，非常的简单易懂，我们需要接入输入和输出字符串流`istringstream`和`ostringstream`，对于序列化，我们从根节点开始，如果节点存在，则将值存入输出字符串流，然后分别对其左右子节点递归调用序列化函数即可。对于去序列化，我们先读入第一个字符，以此生成一个根节点，然后再对根节点的左右子节点递归调用去序列化函数即可，参见代码如下：

解法1：

```

1 class Codec {
2 public:
3     // Encodes a tree to a single string.
4     string serialize(TreeNode* root) {
5         ostringstream out;
6         serialize(root, out);
7         return out.str();
8     }
9     // Decodes your encoded data to tree.
10    TreeNode* deserialize(string data) {
11        istringstream in(data);
12        return deserialize(in);
13    }
14 private:
15    void serialize(TreeNode *root, ostringstream &out) {
16        if (root) {
17            out << root->val << ' ';
18            serialize(root->left, out);
19            serialize(root->right, out);
20        } else {
21            out << "# ";
22        }
23    }
24    TreeNode* deserialize(istringstream &in) {
25        string val;
26        in >> val;
27        if (val == "#") return nullptr;
28        TreeNode *root = new TreeNode(stoi(val));
29        root->left = deserialize(in);
30        root->right = deserialize(in);
31        return root;
32    }
33 };

```

另一种方法是层序遍历的非递归解法，这种方法略微复杂一些，我们需要借助queue来做，本质是BFS算法，也不是很难理解，就是BFS算法的常规套路稍作修改即可，参见代码如下：

解法2：

```

1  class Codec {
2  public:
3      // Encodes a tree to a single string.
4      string serialize(TreeNode* root) {
5          ostringstream out;
6          queue<TreeNode*> q;
7          if (root) q.push(root);
8          while (!q.empty()) {
9              TreeNode *t = q.front(); q.pop();
10             if (t) {
11                 out << t->val << ' ';
12                 q.push(t->left);
13                 q.push(t->right);
14             } else {
15                 out << "# ";
16             }
17         }
18         return out.str();
19     }
20     // Decodes your encoded data to tree.
21     TreeNode* deserialize(string data) {
22         if (data.empty()) return nullptr;
23         istringstream in(data);
24         queue<TreeNode*> q;
25         string val;
26         in >> val;
27         TreeNode *res = new TreeNode(stoi(val)), *cur = res;
28         q.push(cur);
29         while (!q.empty()) {
30             TreeNode *t = q.front(); q.pop();
31             if (!(in >> val)) break;
32             if (val != "#") {
33                 cur = new TreeNode(stoi(val));
34                 q.push(cur);
35                 t->left = cur;
36             }
37             if (!(in >> val)) break;
38             if (val != "#") {
39                 cur = new TreeNode(stoi(val));
40                 q.push(cur);
41                 t->right = cur;
42             }
43         }
44         return res;
45     }
46 };

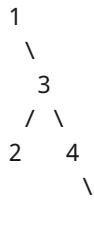
```

298. 二叉树最长连续序列

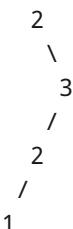
Given a binary tree, find the length of the longest consecutive sequence path.

The path refers to any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The longest consecutive path need to be from parent to child (cannot be the reverse).

For example,



Longest consecutive sequence path is 3-4-5, so return 3.



Longest consecutive sequence path is 2-3,not3-2-1, so return 2.

这道题让我们求二叉树的最长连续序列，关于二叉树的题基本都需要遍历树，而递归遍历写起来特别简单，下面这种解法是用到了递归版的先序遍历，我们对于每个遍历到的节点，我们看节点值是否比参数值(父节点值)大1，如果是则长度加1，否则长度重置为1，然后更新结果res，再递归调用左右子节点即可，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     int longestConsecutive(TreeNode* root) {
4         if (!root) return 0;
5         int res = 0;
6         dfs(root, root->val, 0, res);
7         return res;
8     }
9     void dfs(TreeNode *root, int v, int out, int &res) {
10        if (!root) return;
11        if (root->val == v + 1) ++out;
12        else out = 1;
13        res = max(res, out);
14        dfs(root->left, root->val, out, res);
15        dfs(root->right, root->val, out, res);
16    }
17 };

```

CPP

下面这种写法是利用分治法的思想，对左右子节点分别处理，如果左子节点存在且节点值比其父节点值大1，则递归调用函数，如果节点值不是刚好大1，则递归调用重置了长度的函数，对于右子节点的处理情况和左子节点相同，参见代码如下：

解法2:

```

1 class Solution {
2 public:
3     int longestConsecutive(TreeNode* root) {
4         if (!root) return 0;
5         int res = 0;
6         dfs(root, 1, res);
7         return res;
8     }
9     void dfs(TreeNode *root, int len, int &res) {
10        res = max(res, len);
11        if (root->left) {
12            if (root->left->val == root->val + 1) dfs(root->left, len + 1, res);
13            else dfs(root->left, 1, res);
14        }
15        if (root->right) {
16            if (root->right->val == root->val + 1) dfs(root->right, len + 1, res);
17            else dfs(root->right, 1, res);
18        }
19    }
20 };

```

下面这种递归写法相当简洁，但是核心思想和上面两种方法并没有太大的区别，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int longestConsecutive(TreeNode* root) {
4         return helper(root, NULL, 0);
5     }
6     int helper(TreeNode *root, TreeNode *p, int res) {
7         if (!root) return res;
8         res = (p && root->val == p->val + 1) ? res + 1 : 1;
9         return max(res, max(helper(root->left, root, res), helper(root->right, root,
10 res)));
11     }
12 };

```

上面三种都是递归的写法，下面我们来看看迭代的方法，写法稍稍复杂一些，用的还是DFS的思想，以层序来遍历树，对于遍历到的节点，我们看其左右子节点有没有满足题意的，如果左子节点比其父节点大1，若右子节点存在，则排入queue，指针移到左子节点，反之若右子节点比其父节点大1，若左子节点存在，则排入queue，指针移到右子节点，依次类推直到queue为空，参见代码如下：

解法4：

```

1 class Solution {
2     public:
3         int longestConsecutive(TreeNode* root) {
4             if (!root) return 0;
5             int res = 0;
6             queue<TreeNode*> q;
7             q.push(root);
8             while (!q.empty()) {
9                 int len = 1;
10                TreeNode *t = q.front(); q.pop();
11                while ((t->left && t->left->val == t->val + 1) || (t->right && t->right->val ==
12 t->val + 1)) {
13                    if (t->left && t->left->val == t->val + 1) {
14                        if (t->right) q.push(t->right);
15                        t = t->left;
16                    } else if (t->right && t->right->val == t->val + 1) {
17                        if (t->left) q.push(t->left);
18                        t = t->right;
19                    }
20                    ++len;
21                }
22                if (t->left) q.push(t->left);
23                if (t->right) q.push(t->right);
24                res = max(res, len);
25            }
26            return res;
27        }
28    };
29}

```

299. 公母牛游戏

You are playing the following Bulls and Cows game with your friend: You write a 4-digit secret number and ask your friend to guess it, each time your friend guesses a number, you give a hint, the hint tells your friend how many digits are in the correct positions (called "bulls") and how many digits are in the wrong positions (called "cows"), your friend will use those hints to find out the secret number.

For example:

```

Secret number: 1807
Friend's guess: 7810
Hint: 1 bull and 3 cows. (The bull is 8, the cows are 0, 1 and 7.)

```

这道题提出了一个叫公牛母牛的游戏，其实就是之前文曲星上有的猜数字的游戏，有一个四位数字，你猜一个结果，然后根据你猜的结果和真实结果做对比，提示有多少个数字和位置都正确的叫做bulls，还提示有多少数字正确但位置不对的叫做cows，根据这些信息来引导我们继续猜测正确的数字。这道题并没有让我们实现整个游戏，而只用实现一次比较即可。给出两个字符串，让我们找出分别几个bulls和cows。这题需要用哈希表，来建立数字和其出现次数的映射。我最开始想的方法是用两次遍历，第一次遍历找出所有位置相同且值相同的数字，即bulls，并且记录secret中不是bulls的数字出现的次数。然后第二次遍历我们针对guess中不是bulls的位置，如果在哈希表中存在，cows自增1，然后映射值减1，参见如下代码：

解法1：

```

1 class Solution {
2 public:
3     string getHint(string secret, string guess) {
4         int m[256] = {0}, bulls = 0, cows = 0;
5         for (int i = 0; i < secret.size(); ++i) {
6             if (secret[i] == guess[i]) ++bulls;
7             else ++m[secret[i]];
8         }
9         for (int i = 0; i < secret.size(); ++i) {
10             if (secret[i] != guess[i] && m[guess[i]]) {
11                 ++cows;
12                 --m[guess[i]];
13             }
14         }
15         return to_string(bulls) + "A" + to_string(cows) + "B";
16     }
17 };

```

我们其实可以用一次循环就搞定的，在处理不是bulls的位置时，我们看如果secret当前位置数字的映射值小于0，则表示其在guess中出现过，cows自增1，然后映射值加1，如果guess当前位置的数字的映射值大于0，则表示其在secret中出现过，cows自增1，然后映射值减1，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     string getHint(string secret, string guess) {
4         int m[256] = {0}, bulls = 0, cows = 0;
5         for (int i = 0; i < secret.size(); ++i) {
6             if (secret[i] == guess[i]) ++bulls;
7             else {
8                 if (m[secret[i]]++ < 0) ++cows;
9                 if (m[guess[i]]-- > 0) ++cows;
10            }
11        }
12        return to_string(bulls) + "A" + to_string(cows) + "B";
13    }
14 };

```

最后我们还可以稍作修改写的更简洁一些，a是bulls的值，b是bulls和cows之和，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     string getHint(string secret, string guess) {
4         int m[256] = {0}, a = 0, b = 0, i = 0;
5         for (char s : secret) {
6             char g = guess[i++];
7             a += s == g;
8             b += (m[s]++ < 0) + (m[g]-- > 0);
9         }
10        return to_string(a) + "A" + to_string(b - a) + "B";
11    }
12 };

```

300. 最长递增子序列

Given an unsorted array of integers, find the length of longest increasing subsequence.

Example:

Input: [10, 9, 2, 5, 3, 7, 101, 18]

Output: 4

Explanation: The longest increasing subsequence is [2, 3, 7, 101], therefore the length is 4.

Note:

There may be more than one LIS combination, it is only necessary for you to return the length.

Your algorithm should run in O(n²) complexity.

Follow up: Could you improve it to O(n log n) time complexity?

这道题让我们求最长递增子串Longest Increasing Subsequence的长度，简称LIS的长度。我最早接触到这道题是在LintCode上，可参见我之前的博客Longest Increasing Subsequence 最长递增子序列，那道题写的解法略微复杂，下面我们来看其他的一些解法。首先来看一种动态规划Dynamic Programming的解法，这种解法的时间复杂度为O(n²)，类似brute force的解法，我们维护一个一维dp数组，其中dp[i]表示以nums[i]为结尾的最长递增子串的长度，对于每一个nums[i]，我们从第一个数再搜索到i，如果发现某个数小于nums[i]，我们更新dp[i]，更新方法为dp[i] = max(dp[i], dp[j] + 1)，即比较当前dp[i]的值和那个小于num[i]的数的dp值加1的大小，我们就这样不断的更新dp数组，到最后dp数组中最大的值就是我们要返回的LIS的长度，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int lengthOfLIS(vector<int>& nums) {
4         vector<int> dp(nums.size(), 1);
5         int res = 0;
6         for (int i = 0; i < nums.size(); ++i) {
7             for (int j = 0; j < i; ++j) {
8                 if (nums[i] > nums[j]) {
9                     dp[i] = max(dp[i], dp[j] + 1);
10                }
11            }
12            res = max(res, dp[i]);
13        }
14        return res;
15    }
16 };

```

CPP

下面我们来看一种优化时间复杂度到O(nlg n)的解法，这里用到了二分查找法，所以才能加快运行时间哇。思路是，我们先建立一个数组ends，把首元素放进去，然后比较之后的元素，如果遍历到的新元素比ends数组中的首元素小的话，替换首元素为此新元素，如果遍历到的新元素比ends数组中的末尾元素还大的话，将此新元素添加到ends数组末尾(注意不覆盖原末尾元素)。如果遍历到的新元素比ends数组首元素大，比尾元素小时，此时用二分查找法找到第一个不小于此新元素的位置，覆盖掉位置的原来的数字，以此类推直至遍历完整个nums数组，此时ends数组的长度就是我们要求的LIS的长度，特别注意的是ends数组的值可能不是一个真实的LIS，比如若输入数组nums为{4, 2, 4, 5, 3, 7}，那么算完后的ends数组为{2, 3, 5, 7}，可以发现它不是一个原数组的LIS，只是长度相等而已，千万要注意这点。参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int lengthOfLIS(vector<int>& nums) {
4         if (nums.empty()) return 0;
5         vector<int> ends{nums[0]};
6         for (auto a : nums) {
7             if (a < ends[0]) ends[0] = a;
8             else if (a > ends.back()) ends.push_back(a);
9             else {
10                 int left = 0, right = ends.size();
11                 while (left < right) {
12                     int mid = left + (right - left) / 2;
13                     if (ends[mid] < a) left = mid + 1;
14                     else right = mid;
15                 }
16                 ends[right] = a;
17             }
18         }
19         return ends.size();
20     }
21 };

```

我们来看一种思路更清晰的二分查找法，跟上面那种方法很类似，思路是先建立一个空的dp数组，然后开始遍历原数组，对于每一个遍历到的数字，我们用二分查找法在dp数组找第一个不小于它的数字，如果这个数字不存在，那么直接在dp数组后面加上遍历到的数字，如果存在，则将这个数字更新为当前遍历到的数字，最后返回dp数组的长度即可，注意的是，跟上面的方法一样，特别注意的是dp数组的值可能不是一个真实的LIS。参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int lengthOfLIS(vector<int>& nums) {
4         vector<int> dp;
5         for (int i = 0; i < nums.size(); ++i) {
6             int left = 0, right = dp.size();
7             while (left < right) {
8                 int mid = left + (right - left) / 2;
9                 if (dp[mid] < nums[i]) left = mid + 1;
10                else right = mid;
11            }
12            if (right >= dp.size()) dp.push_back(nums[i]);
13            else dp[right] = nums[i];
14        }
15        return dp.size();
16    }
17 };

```

下面我们来看两种比较tricky的解法，利用到了C++中STL的lower_bound函数，lower_bound返回数组中第一个不小于指定值的元素，跟上面的算法类似，我们还需要一个一维数组v，然后对于遍历到的nums中每一个元素，找其lower_bound，如果没有lower_bound，说明新元素比一维数组的尾元素还要大，直接添加到数组v中，跟解法二的思路相同了。如果有lower_bound，说明新元素不是最大的，将其lower_bound替换为新元素，这个过程跟算法二的二分查找法的部分实现相同功能，最后也是返回数组v的长度，注意数组v也不一定是真实的LIS，参见代码如下：

解法4：

```

1 class Solution {
2 public:
3     int lengthOfLIS(vector<int>& nums) {
4         vector<int> v;
5         for (auto a : nums) {
6             auto it = lower_bound(v.begin(), v.end(), a);
7             if (it == v.end()) v.push_back(a);
8             else *it = a;
9         }
10        return v.size();
11    }
12 };

```

既然能用lower_bound，那么upper_bound就耐不住寂寞了，也要出来解个题。upper_bound是返回数组中第一个大于指定值的元素，和lower_bound的区别时，它不能返回和指定值相等的元素，那么当新进来的数和数组中尾元素一样大时，upper_bound无法返回这个元素，那么按算法三的处理方法是加到数组中，这样就不是严格的递增子串了，所以我们要做个处理，在处理每个新进来的元素时，先判断数组v中有无此元素，有的话直接跳过，这样就避免了相同数字的情况，参见代码如下：

解法5：

```

1 class Solution {
2 public:
3     int lengthOfLIS(vector<int>& nums) {
4         vector<int> v;
5         for (auto a : nums) {
6             if (find(v.begin(), v.end(), a) != v.end()) continue;
7             auto it = upper_bound(v.begin(), v.end(), a);
8             if (it == v.end()) v.push_back(a);
9             else *it = a;
10        }
11        return v.size();
12    }
13 };

```

301. 移除非法括号

Remove the minimum number of invalid parentheses in order to make the input string valid.
Return all possible results.

Note: The input string may contain letters other than the parentheses (and).

Examples:

```

"()())()" -> ["()()", "(())()"]
"(a)())()" -> ["(a)()", "(a())()"]
")()" -> []

```

这道题让我们移除最少的括号使得给定字符串为一个合法的含有括号的字符串，我们从小数学里就有括号，所以应该对合法的含有括号的字符串并不陌生，字符串中的左右括号数应该相同，而且每个右括号左边一定有其对应的左括号，而且题目中给的例子也说明了去除方法不唯一，我们需要找出所有合法的取法。参考了网上大神的解法，这道题首先可以用BFS来解，我们先把给定字符串排入队中，然后取出检测其是否合法，若合法直接返回，不合法的话，我们对其进行遍历，对于遇到的左右括号的字符，我们去掉括号字符生成一个新的字符串，如果这个字符串之前没有遇到过，将其排入队中，我们用哈希集合记录一个字符串是否出现过。我们对队列中的每个元素都进行相同的操作，直到队列为空还没找到合法的字符串的话，那就返回空集，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<string> removeInvalidParentheses(string s) {
4         vector<string> res;
5         unordered_set<string> visited{{s}};
6         queue<string> q{{s}};
7         bool found = false;
8         while (!q.empty()) {
9             string t = q.front(); q.pop();
10            if (isValid(t)) {
11                res.push_back(t);
12                found = true;
13            }
14            if (found) continue;
15            for (int i = 0; i < t.size(); ++i) {
16                if (t[i] != '(' && t[i] != ')') continue;
17                string str = t.substr(0, i) + t.substr(i + 1);
18                if (!visited.count(str)) {
19                    q.push(str);
20                    visited.insert(str);
21                }
22            }
23        }
24        return res;
25    }
26    bool isValid(string t) {
27        int cnt = 0;
28        for (int i = 0; i < t.size(); ++i) {
29            if (t[i] == '(') ++cnt;
30            else if (t[i] == ')' && --cnt < 0) return false;
31        }
32        return cnt == 0;
33    }
34 };

```

CPP

下面来看一种递归解法，这种解法首先统计了多余的半括号的数量，用cnt1表示多余的左括号，cnt2表示多余的右括号，因为给定字符串左右括号要么一样多，要么左括号多，要么右括号多，也可能左右括号都多，比如")("。所以cnt1和cnt2要么都为0，要么都大于0，要么一个为0，另一个大于0。好，下面进入我们的递归函数，首先判断，如果当cnt1和cnt2都为0时，说明此时左右括号个数相等了，我们调用isValid子函数来判断是否正确，正确的话加入结果res中并返回即可。否则从start开始遍历，这里的变量start表示当前递归开始的位置，我们不需要每次都从头开始，会有大量重复计算。而且对于多个相同的半括号在一起，我们只删除第一个，比如"0)",这里有两个右括号，我们不管删第一个还是删第二个右括号都会得到"0"，没有区别，所以只用算一次就行了，我们通过和上一个字符比较，如果不相同，说明是第一个右括号，如果相同则直接跳过。此时来看如果cnt1大于0，说明此时左括号多，而如果当前字符正好是左括号的时候，我们可以删掉当前左括号，继续调用递归，此时cnt1的值就应该减1，因为已经删掉了一个左括号。同理，如果cnt2大于0，说明此时右括号多，而如果当前字符正好是右括号的时候，我们可以删掉当前右括号，继续调用递归，此时cnt2的值就应该减1，因为已经删掉了一个右括号。参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<string> removeInvalidParentheses(string s) {
4         vector<string> res;
5         int cnt1 = 0, cnt2 = 0;
6         for (char c : s) {
7             cnt1 += (c == '(');
8             if (cnt1 == 0) cnt2 += (c == ')');
9             else cnt1 -= (c == ')');
10        }
11        helper(s, 0, cnt1, cnt2, res);
12        return res;
13    }
14    void helper(string s, int start, int cnt1, int cnt2, vector<string>& res) {
15        if (cnt1 == 0 && cnt2 == 0) {
16            if (isValid(s)) res.push_back(s);
17            return;
18        }
19        for (int i = start; i < s.size(); ++i) {
20            if (i != start && s[i] == s[i - 1]) continue;
21            if (cnt1 > 0 && s[i] == '(') {
22                helper(s.substr(0, i) + s.substr(i + 1), i, cnt1 - 1, cnt2, res);
23            }
24            if (cnt2 > 0 && s[i] == ')') {
25                helper(s.substr(0, i) + s.substr(i + 1), i, cnt1, cnt2 - 1, res);
26            }
27        }
28    }
29    bool isValid(string t) {
30        int cnt = 0;
31        for (int i = 0; i < t.size(); ++i) {
32            if (t[i] == '(') ++cnt;
33            else if (t[i] == ')' && --cnt < 0) return false;
34        }
35        return cnt == 0;
36    }
37};

```

下面这种解法是论坛上的高票解法，思路确实很巧妙。递归函数的参数中，last_i表示当前遍历到的位置，相当上面解法中的start，last_j表示上一个删除的位置，这样可以避免重复计算。然后有个括号字符数组，初始化时放入左括号和右括号，博主认为这个字符数组是此解法最精髓的地方，因为其顺序可以改变，可以变成反向括号，这个就比较叼了，后面再讲它到底有多叼吧。我们在递归函数中，从last_i开始遍历，在找正向括号的时候，用变量cnt表示括号数组中的左括号出现的次数，遇到左括号自增1，遇到右括号自减1。当左括号大于等于右括号的时候，我们直接跳过。这个循环的目的是要删除多余的右括号，所以当cnt小于0的时候，我们从上一个删除位置last_j开始遍历，如果当前是右括号，且是第一个右括号（关于这块可以参见上面解法中的分析），我们删除当前右括号，并调用递归函数。注意这个for循环结束后要直接返回，因为进这个for循环的都是右括号多的，删到最后最多是删成和左括号一样多，不需要再去翻转删左括号。好，最后来说这个最叼的翻转，当字符串的左括号个数大于等于右括号的时候，不会进入第二个for循环，自然也不会return。那么由于左括号的个数可能会要大于右括号，所以我们还要删除多余的左括号，所以我们将字符串反转一下，比如"(0"，反转变成")("，此时虽然我们还是要删除多余的左括号，但是反转后就没有合法的括号了，所以变成了找反向括号")("，那么还是可以删除多余的左括号，然后我们判断此时括号数组的状态，如果是正向括号，说明此时正要删除左括号，那么就调用递归函数，last_i和last_j均重置为0，括号数组初始化为反向括号。如果此时已经是反向括号了，说明之前的左括号已经删掉了变成了")("，然后又反转了一下，变回来了"0"，那么就可以直接加入结果res了，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     vector<string> removeInvalidParentheses(string s) {
4         vector<string> res;
5         helper(s, 0, 0, {'(', ')'}, res);
6         return res;
7     }
8     void helper(string s, int last_i, int last_j, vector<char> p, vector<string>& res) {
9         int cnt = 0;
10        for (int i = last_i; i < s.size(); ++i) {
11            if (s[i] == p[0]) ++cnt;
12            else if (s[i] == p[1]) --cnt;
13            if (cnt >= 0) continue;
14            for (int j = last_j; j <= i; ++j) {
15                if (s[j] == p[1] && (j == last_j || s[j] != s[j - 1])) {
16                    helper(s.substr(0, j) + s.substr(j + 1), i, j, p, res);
17                }
18            }
19        }
20    }
21    string rev = string(s.rbegin(), s.rend());
22    if (p[0] == '(') helper(rev, 0, 0, ')', '(', res);
23    else res.push_back(rev);
24 }
25 };

```

302. 包含黑像素的最小矩阵

An image is represented by a binary matrix with 0 as a white pixel and 1 as a black pixel. The black pixels are connected, i.e., there is only one black region. Pixels are connected horizontally and vertically. Given the location (x, y) of one of the black pixels, return the area of the smallest (axis-aligned) rectangle that encloses all black pixels.

For example, given the following image:

```
[  
"0010",  
"0110",  
"0100"  
]
```

and $x = 0, y = 2$,

Return 6.

这道题给我们一个二维矩阵，表示一个图片的数据，其中1代表黑像素，0代表白像素，现在让我们找出一个最小的矩阵可以包括所有的黑像素，还给了我们一个黑像素的坐标，我们先来看Brute Force的方法，这种方法的效率不高，遍历了整个数组，如果遇到了1，就更新矩形的返回，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int minArea(vector<vector<char>>& image, int x, int y) {
4         int left = y, right = y, up = x, down = x;
5         for (int i = 0; i < image.size(); ++i) {
6             for (int j = 0; j < image[i].size(); ++j) {
7                 if (image[i][j] == '1') {
8                     left = min(left, j);
9                     right = max(right, j);
10                    up = min(up, i);
11                    down = max(down, i);
12                }
13            }
14        }
15        return (right - left + 1) * (down - up + 1);
16    }
17 };

```

下面这种解法是解法一的递归写法，本质上来说跟上面的解法没有啥区别，也没有任何的优化，所以仍然可以认为是暴力搜索法，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int minArea(vector<vector<char>>& image, int x, int y) {
4         int left = y, right = y, up = x, down = x;
5         dfs(image, x, y, left, right, up, down);
6         return (right - left + 1) * (down - up + 1);
7     }
8     void dfs(vector<vector<char>> &image, int x, int y, int &left, int &right, int &up, int
9 &down) {
10     if (x < 0 || x >= image.size() || y < 0 || y >= image[0].size() || image[x][y] !=
11     '1') return;
12     left = min(left, y);
13     right = max(right, y);
14     up = min(up, x);
15     down = max(down, x);
16     image[x][y] = '2';
17     dfs(image, x + 1, y, left, right, up, down);
18     dfs(image, x - 1, y, left, right, up, down);
19     dfs(image, x, y + 1, left, right, up, down);
20     dfs(image, x, y - 1, left, right, up, down);
21 }
22 };

```

我们再来看一种优化了时间复杂度的解法，这是一种二分搜索法，以给定的一个黑像素(x, y)为中心，分别用二分法快速找到整个黑色区域的上下左右的临界点，然后直接算出面积。首先我们来看上边界怎么找，既然是以(x, y)为中心，而且上边界又是某个行数，那么其范围肯定在[0, x]之间，能成为上边界的条件是该行中至少有一个点是1，那么其列数的范围就在[0, n]之间，我们在进行二分搜索的时候，先根据i, j算出中间行mid，然后列数从0开始遍历，直到找到为1的点，或者越界位置，然后我们判断列数是否越界，越界的话，说明当前行没有1，此时更新i为mid+1，如果找到了1，那么更新j为mid。找下边界也是同样的道理，但是跟上边界稍微又些不同的地方是，如果当前行找到了1，我们应该再往下找，那么i应该更新为mid+1；如果没找到，就应该往上找，靠近(x, y)点；所以两种情况只是在二分法更新范围的地方正好想法，所以我们可以用一个bool型的变量opt来决定还如何更新行数。

下面我们来看如何确定左边界和右边界，其实跟确定上下边界大同小异。左边界是列数，若以(x, y)点为中心，那么其范围便是[0, y]，因为我们之前已经确定了上下边界up和down了，所以左边界点的行数范围就是[up, down]，那么同理，当我们通过i, j求出了中间列mid时，我们就要遍历该列，找到为1的点，所以此时我们是用image[k][mid]，而在找上下边界时，我们用的是image[mid][k]，还是顺序不一样，我们可以用另外一个bool型变量h来控制，h表示horizontal，就是水平遍历的意思。这样我们通过两个bool型变量就可以用一个函数来涵盖四种情况的二分搜索，是不是很叼？下面更新i或j的时候参考上下边界的分析，应该不难理解，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int minArea(vector<vector<char>>& image, int x, int y) {
4         int m = image.size(), n = image[0].size();
5         int up = binary_search(image, true, 0, x, 0, n, true);
6         int down = binary_search(image, true, x + 1, m, 0, n, false);
7         int left = binary_search(image, false, 0, y, up, down, true);
8         int right = binary_search(image, false, y + 1, n, up, down, false);
9         return (right - left) * (down - up);
10    }
11    int binary_search(vector<vector<char>> &image, bool h, int i, int j, int low, int high,
12    bool opt) {
13        while (i < j) {
14            int k = low, mid = (i + j) / 2;
15            while (k < high && (h ? image[mid][k] : image[k][mid]) == '0') ++k;
16            if (k < high == opt) j = mid;
17            else i = mid + 1;
18        }
19        return i;
20    }
21 };

```

303. 区域和检索 - 不可变

Given an integer array nums, find the sum of the elements between indices i and j ($i \leq j$), inclusive.

Example:

Given nums = [-2, 0, 3, -5, 2, -1]

```

sumRange(0, 2) -> 1
sumRange(2, 5) -> -1
sumRange(0, 5) -> -3

```

Note:

You may assume that the array does not change.

There are many calls to sumRange function.

这道题让我们检索一个数组的某个区间的所有数字之和，题目中给了两条条件，首先数组内容不会变化，其次有很多的区间和检索。那么我们用传统的遍历相加来求每次区间和检索，十分的不高效，而且无法通过OJ。所以这道题的难点就在于是否能想到来用建立累计直方图的思想来建立一个累计和的数组dp，其中dp[i]表示[0, i]区间的数字之和，那么[i, j]就可以表示为dp[j]-dp[i-1]，这里要注意一下当i=0时，直接返回dp[j]即可，参见代码如下：

解法1：

```

1 class NumArray {
2 public:
3     NumArray(vector<int> &nums) {
4         dp = nums;
5         for (int i = 1; i < nums.size(); ++i) {
6             dp[i] += dp[i - 1];
7         }
8     }
9     int sumRange(int i, int j) {
10        return i == 0? dp[j] : dp[j] - dp[i - 1];
11    }
12 private:
13     vector<int> dp;
14 };

```

当然，我们也可以通过增加一位dp的长度，来避免在sumRange中检测i是否为0，参见代码如下：

解法2：

```

1 class NumArray {
2 public:
3     NumArray(vector<int> &nums) {
4         dp.resize(nums.size() + 1, 0);
5         for (int i = 1; i <= nums.size(); ++i) {
6             dp[i] = dp[i - 1] + nums[i - 1];
7         }
8     }
9     int sumRange(int i, int j) {
10        return dp[j + 1] - dp[i];
11    }
12 private:
13     vector<int> dp;
14 };

```

304. 二维区域和检索 - 不可变

Given a 2D matrix matrix, find the sum of the elements inside the rectangle defined by its upper left corner (row1, col1) and lower right corner (row2, col2).

3	0	1	4	2
5	6	3	2	1
1	2	0	1	5
4	1	0	1	7
1	0	3	0	5

The above rectangle (with the red border) is defined by (row1, col1) = (2, 1) and (row2, col2) = (4, 3), which contains sum = 8.

Example:

```
Given matrix = [
    [3, 0, 1, 4, 2],
    [5, 6, 3, 2, 1],
    [1, 2, 0, 1, 5],
    [4, 1, 0, 1, 7],
    [1, 0, 3, 0, 5]
]
```

```
sumRegion(2, 1, 4, 3) -> 8
sumRegion(1, 1, 2, 2) -> 11
sumRegion(1, 2, 2, 4) -> 12
```

Note:

You may assume that the matrix does not change.

There are many calls to sumRegion function.

You may assume that row1 ≤ row2 and col1 ≤ col2.

这道题让我们求一个二维区域和的检索，是之前那道题Range Sum Query - Immutable 区域和检索的延伸。有了之前那道题的基础，我们知道这道题其实也是换汤不换药，还是要建立一个累计区域和的数组，然后根据边界值的加减法来快速求出给定区域之和。这里我们维护一个二维数组dp，其中dp[i][j]表示累计区间(0, 0)到(i, j)这个矩形区间所有的数字之和，那么此时如果我们想要快速求出(r1, c1)到(r2, c2)的矩形区间时，只需 $dp[r2][c2] - dp[r2][c1 - 1] - dp[r1 - 1][c2] + dp[r1 - 1][c1 - 1]$ 即可，下面的代码中我们由于用了辅助列和辅助行，所以下标会有些变化，参见代码如下：

```
1 class NumMatrix {
2 public:
3     NumMatrix(vector<vector<int> > &matrix) {
4         if (matrix.empty() || matrix[0].empty()) return;
5         dp.resize(matrix.size() + 1, vector<int>(matrix[0].size() + 1, 0));
6         for (int i = 1; i <= matrix.size(); ++i) {
7             for (int j = 1; j <= matrix[0].size(); ++j) {
8                 dp[i][j] = dp[i][j - 1] + dp[i - 1][j] - dp[i - 1][j - 1] + matrix[i - 1][j - 1];
9             }
10        }
11    }
12    int sumRegion(int row1, int col1, int row2, int col2) {
13        return dp[row2 + 1][col2 + 1] - dp[row1][col2 + 1] - dp[row2 + 1][col1] + dp[row1][col1];
14    }
15
16 private:
17     vector<vector<int> > dp;
18 }
```

CPP

305. 岛屿的数量之二

A 2d grid map of m rows and n columns is initially filled with water. We may perform an addLand operation which turns the water at position (row, col) into a land. Given a list of positions to operate, count the number of islands after each addLand operation. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example:

Given m = 3, n = 3, positions = [[0,0], [0,1], [1,2], [2,1]].

Initially, the 2d grid grid is filled with water. (Assume 0 represents water and 1 represents land).

```
0 0 0  
0 0 0  
0 0 0
```

Operation #1: addLand(0, 0) turns the water at grid[0][0] into a land.

```
1 0 0  
0 0 0  Number of islands = 1  
0 0 0
```

Operation #2: addLand(0, 1) turns the water at grid[0][1] into a land.

```
1 1 0  
0 0 0  Number of islands = 1  
0 0 0
```

Operation #3: addLand(1, 2) turns the water at grid[1][2] into a land.

```
1 1 0  
0 0 1  Number of islands = 2  
0 0 0
```

Operation #4: addLand(2, 1) turns the water at grid[2][1] into a land.

```
1 1 0  
0 0 1  Number of islands = 3  
0 1 0
```

We return the result as an array: [1, 1, 2, 3]

Challenge:

Can you do it in time complexity O(k log mn), where k is the length of the positions?

这道题是之前那道Number of Islands的拓展，难度增加了不少，因为这次是一个点一个点的增加，每增加一个点，都要统计一下现在总共的岛屿个数，最开始初始化时没有陆地，如下：

```
0 0 0
0 0 0
0 0 0
```

假如我们在(0, 0)的位置增加一个陆地，那么此时岛屿数量为1：

```
1 0 0
0 0 0
0 0 0
```

假如我们再在(0, 2)的位置增加一个陆地，那么此时岛屿数量为2：

```
1 0 1
0 0 0
0 0 0
```

假如我们再在(0, 1)的位置增加一个陆地，那么此时岛屿数量却又变为1：

```
1 1 1
0 0 0
0 0 0
```

假如我们再在(1, 1)的位置增加一个陆地，那么此时岛屿数量仍为1：

```
1 1 1
0 1 0
0 0 0
```

那么我们为了解决这种陆地之间会合并的情况，最好能够将每个陆地都标记出其属于哪个岛屿，这样就会方便我们统计岛屿个数。这种群组类问题，很适合使用联合查找 Union Find 来做，又叫并查集 Disjoint Set，LeetCode中使用这种解法的题目还不少呢，比如Friend Circles, Graph Valid Tree, Redundant Connection II 等等。一般来说，UF算法的思路是每个个体先初始化为不同的群组，然后遍历有关联的两个个体，如果发现其getRoot函数的返回值不同，则手动将二者加入一个群组，然后总群组数自减1。这里就要分别说一下root数组，和getRoot函数。两个同群组的个体，通过getRoot函数一定会返回相同的值，但是其在root 数组中的值不一定相同，我们可以类比成getRoot函数返回的是祖先，如果两个人的祖先相同，那么其是属于一个家族的（这里不是指人类共同的祖先哈）。root可以用数组或者HashMap来表示，如果个体是数字的话，那么数组就OK，如果个体是字符串的话，可能就需要用HashMap了。root数组的初始化可以有两种，可以均初始化为-1，或者都初始化为不同的数字，博主一般喜欢初始化为不同的数字。getRoot函数的写法也可用递归或者迭代的方式，可参见博主之前的帖子Redundant Connection II中的讨论部分。这么一说感觉UF算法的东西还蛮多的，啥时候博主写个UF总结贴吧。

那么具体来看这道题吧，此题跟经典的UF使用场景有一点点的区别，因为一般的场景中两个个体之间只有两种关系，属于一个群组或者不属于同一个群组，而这道题里面由于water的存在，就多了一种情况，我们只需要事先检测一下当前位置是不是岛屿就行了，总之问题不大。一般来说我们的root数组都是使用一维数组，方便一些，那么这里就可以将二维数组encode为一维的，于是我们需要一个长度为m*n的一维数组来标记各个位置属于哪个岛屿，我们假设每个位置都是一个单独岛屿，岛屿编号可以用其坐标位置表示，但是我们初始化时将其都赋为-1，这样方便我们知道哪些位置尚未变成岛屿。然后我们开始遍历陆地数组，将其岛屿编号设置为其坐标位置，然后岛屿计数加1，我们此时开始遍历其上下左右的位置，遇到越界或者岛屿标号为-1的情况直接跳过，现在知道我们初始化为-1的好处了吧，遇到是water的地方直接跳过。否则我们用getRoot来查找邻居位置的岛屿编号，同时也用getRoot来查找当前点的编号，这一步就是经典的UF算法的操作了，因为当前这两个land是相邻的，它们是属于一个岛屿，所以其getRoot函数的返回值suppose应该是相等的，但是如果返回值不同，说明我们需要合并岛屿，将两个返回值建立关联，并将岛屿计数cnt减1。当我们遍历完当前点的所有邻居时，该合并的都合并完了，将此时的岛屿计数cnt存入结果中，参见代码如下：

```

1 class Solution {
2 public:
3     vector<int> numIslands2(int m, int n, vector<pair<int, int>>& positions) {
4         vector<int> res;
5         int cnt = 0;
6         vector<int> roots(m * n, -1);
7         vector<vector<int>> dirs{{0, -1}, {-1, 0}, {0, 1}, {1, 0}};
8         for (auto a : positions) {
9             int id = n * a.first + a.second;
10            if (roots[id] == -1) {
11                roots[id] = id;
12                ++cnt;
13            }
14            for (auto dir : dirs) {
15                int x = a.first + dir[0], y = a.second + dir[1], cur_id = n * x + y;
16                if (x < 0 || x >= m || y < 0 || y >= n || roots[cur_id] == -1) continue;
17                int p = findRoot(roots, cur_id), q = findRoot(roots, id);
18                if (p != q) {
19                    roots[p] = q;
20                    --cnt;
21                }
22            }
23            res.push_back(cnt);
24        }
25        return res;
26    }
27    int findRoot(vector<int>& roots, int id) {
28        return (id == roots[id]) ? id : findRoot(roots, roots[id]);
29    }
30 };

```

306. 加法数

Additive number is a positive integer whose digits can form additive sequence.

A valid additive sequence should contain at least three numbers. Except for the first two numbers, each subsequent number in the sequence must be the sum of the preceding two.

For example:

"112358" is an additive number because the digits can form an additive sequence: 1, 1, 2, 3, 5, 8.

$1 + 1 = 2, 1 + 2 = 3, 2 + 3 = 5, 3 + 5 = 8$

"199100199" is also an additive number, the additive sequence is: 1, 99, 100, 199.

$1 + 99 = 100, 99 + 100 = 199$

Note: Numbers in the additive sequence cannot have leading zeros, so sequence 1, 2, 03 or 1, 02, 3 is invalid.

Given a string represents an integer, write a function to determine if it's an additive number.

Follow up:

How would you handle overflow for very large input integers?

这道题定义了一种加法数，就是至少含有三个数字，除去前两个数外，每个数字都是前面两个数字的和，题目中给了许多例子，也限定了一些不合法的情况，比如两位数以上不能以0开头等等，让我们来判断一个数是否是加法数。开始我还想是否能用动态规划来解，可是发现不会写递推式，只得作罢。其实这题可用Brute Force的思想来解，我们让第一个数字先从一位开始，第二

个数字从一位，两位，往高位开始搜索，前两个数字确定了，相加得到第三位数字，三个数组排列起来形成一个字符串，和原字符串长度相比，如果小于原长度，那么取出上一次计算的第二个和第三个数，当做新一次计算的前两个数，用相同的方法得到第三个数，再加入当前字符串，再和原字符串长度相比，以此类推，直到当前字符串长度不小于原字符串长度，比较两者是否相同，相同返回true，不相同则继续循环。如果所有情况都遍历完了还是没有返回true，则说明不是Additive Number，返回false，参见代码如下：

```

1 class Solution {
2 public:
3     bool isAdditiveNumber(string num) {
4         for (int i = 1; i < num.size(); ++i) {
5             for (int j = i + 1; j < num.size(); ++j) {
6                 string s1 = num.substr(0, i);
7                 string s2 = num.substr(i, j - i);
8                 long long d1 = atoll(s1.c_str()), d2 = atoll(s2.c_str());
9                 if ((s1.size() > 1 && s1[0] == '0') || (s2.size() > 1 && s2[0] == '0'))
10                continue;
11                long long next = d1 + d2;
12                string nexts = to_string(next);
13                string now = s1 + s2 + nexts;
14                while (now.size() < num.size()) {
15                    d1 = d2;
16                    d2 = next;
17                    next = d1 + d2;
18                    nexts = to_string(next);
19                    now += nexts;
20                }
21                if (now == num) return true;
22            }
23        }
24        return false;
25    }
};

```

307. 区域和检索 - 可变

Given an integer array nums, find the sum of the elements between indices i and j ($i \leq j$), inclusive.

The update(i , val) function modifies $nums$ by updating the element at index i to val .

Example:

Given $nums = [1, 3, 5]$

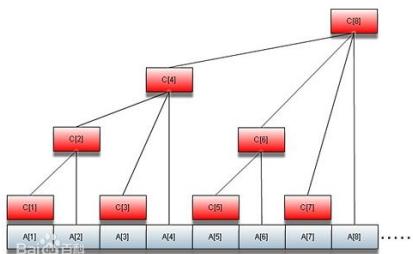
```
sumRange(0, 2) -> 9
update(1, 2)
sumRange(0, 2) -> 8
```

Note:

The array is only modifiable by the update function.

You may assume the number of calls to update and sumRange function is distributed evenly.

这道题是之前那道Range Sum Query - Immutable 区域和检索 - 不可变的延伸，之前那道题由于数组的内容不会改变，所以我们只需要建立一个累计数组就可以支持快速的计算区间值了，而这道题说数组的内容会改变，如果我们还是用之前的方法建立累计和数组，那么每改变一个数字，之后所有位置的数字都要改变，这样如果有很多更新操作的话，就会十分不高效。这道题我们要使用一种新的数据结构，叫做树状数组Binary Indexed Tree，又称Fenwick Tree，这是一种查询和修改复杂度均为 $O(\log n)$ 的数据结构。这个树状数组比较有意思，所有的奇数位置的数字和原数组对应位置的相同，偶数位置是原数组若干位置之和，假如原数组A($a_1, a_2, a_3, a_4 \dots$)，和其对应的树状数组C($c_1, c_2, c_3, c_4 \dots$)有如下关系：



```

C1 = A1
C2 = A1 + A2
C3 = A3
C4 = A1 + A2 + A3 + A4
C5 = A5
C6 = A5 + A6
C7 = A7
C8 = A1 + A2 + A3 + A4 + A5 + A6 + A7 + A8
...

```

那么是如何确定某个位置到底是有几个数组成的呢，原来是根据坐标的最低位Low Bit来决定的，所谓的最低位，就是二进制数的最右边的一个1开始，加上后面的0(如果有的话)组成的数字，例如1到8的最低位如下所示：

坐标	二进制	最低位
1	0001	1
2	0010	2
3	0011	1
4	0100	4
5	0101	1
6	0110	2
7	0111	1
8	1000	8

...

最低位的计算方法有两种，一种是 $x \& (x \wedge (x-1))$ ，另一种是利用补码特性 $x \& -x$ 。

这道题我们先根据给定输入数组建立一个树状数组bit，然后更新某一位数字时，根据最低位的值来更新后面含有这一位数字的地方，一般只需要更新部分偶数位置的值即可，在计算某一位置的前缀和时，利用树状数组的性质也能高效的算出来，参见代码如下：

```

1 class NumArray {
2 public:
3     NumArray(vector<int> &nums) {
4         num.resize(nums.size() + 1);
5         bit.resize(nums.size() + 1);
6         for (int i = 0; i < nums.size(); ++i) {
7             update(i, nums[i]);
8         }
9     }
10    void update(int i, int val) {
11        int diff = val - num[i + 1];
12        for (int j = i + 1; j < num.size(); j += (j&-j)) {
13            bit[j] += diff;
14        }
15        num[i + 1] = val;
16    }
17    int sumRange(int i, int j) {
18        return getSum(j + 1) - getSum(i);
19    }
20    int getSum(int i) {
21        int res = 0;
22        for (int j = i; j > 0; j -= (j&-j)) {
23            res += bit[j];
24        }
25        return res;
26    }
27
28 private:
29     vector<int> num;
30     vector<int> bit;
31 };

```

308. 二维区域和检索 - 可变

Given a 2D matrix matrix, find the sum of the elements inside the rectangle defined by its upper left corner (row1, col1) and lower right corner (row2, col2).

3	0	1	4	2
5	6	3	2	1
1	2	0	1	5
4	1	0	1	7
1	0	3	0	5

The above rectangle (with the red border) is defined by (row1, col1) = (2, 1) and (row2, col2) = (4, 3), which contains sum = 8.

Example:

```
Given matrix = [
    [3, 0, 1, 4, 2],
    [5, 6, 3, 2, 1],
    [1, 2, 0, 1, 5],
    [4, 1, 0, 1, 7],
    [1, 0, 3, 0, 5]
]
```

```
sumRegion(2, 1, 4, 3) -> 8
```

```
update(3, 2, 2)
```

```
sumRegion(2, 1, 4, 3) -> 10
```

Note:

The matrix is only modifiable by the update function.

You may assume the number of calls to update and sumRegion function is distributed evenly.

You may assume that row1 ≤ row2 and col1 ≤ col2.

这道题让我们求二维区域和检索，而且告诉我们数组中的值可能变化，这是之前那道Range Sum Query 2D - Immutable的拓展，由于我们之前做过一维数组的可变和不可变的情况Range Sum Query - Mutable和Range Sum Query - Immutable，那么为了能够通过OJ，我们还是需要用到树状数组Binary Indexed Tree(参见Range Sum Query - Mutable)，其查询和修改的复杂度均为 $O(\log n)$ ，那么我们还是要建立树状数组，我们根据数组中的每一个位置，建立一个二维的树状数组，然后还需要一个getSum函数，以便求得从(0, 0)到(i, j)的区间的数字和，然后在求某一个区间和时，就利用其四个顶点的区间和关系可以快速求出，参见代码如下：

解法1：

```

1 class NumMatrix {
2 public:
3     NumMatrix(vector<vector<int>> &matrix) {
4         if (matrix.empty() || matrix[0].empty()) return;
5         mat.resize(matrix.size() + 1, vector<int>(matrix[0].size() + 1, 0));
6         bit.resize(matrix.size() + 1, vector<int>(matrix[0].size() + 1, 0));
7         for (int i = 0; i < matrix.size(); ++i) {
8             for (int j = 0; j < matrix[i].size(); ++j) {
9                 update(i, j, matrix[i][j]);
10            }
11        }
12    }
13
14    void update(int row, int col, int val) {
15        int diff = val - mat[row + 1][col + 1];
16        for (int i = row + 1; i < mat.size(); i += i&-i) {
17            for (int j = col + 1; j < mat[i].size(); j += j&-j) {
18                bit[i][j] += diff;
19            }
20        }
21        mat[row + 1][col + 1] = val;
22    }
23
24    int sumRegion(int row1, int col1, int row2, int col2) {
25        return getSum(row2 + 1, col2 + 1) - getSum(row1, col2 + 1) - getSum(row2 + 1, col1)
26        + getSum(row1, col1);
27    }
28
29    int getSum(int row, int col) {
30        int res = 0;
31        for (int i = row; i > 0; i -= i&-i) {
32            for (int j = col; j > 0; j -= j&-j) {
33                res += bit[i][j];
34            }
35        }
36        return res;
37    }
38
39 private:
40     vector<vector<int>> mat;
41     vector<vector<int>> bit;
42 };

```

我在网上还看到了另一种解法，这种解法并没有用到树状数组，而是利用了列之和，所谓列之和，就是 (i, j) 就是 $(0, j) + (1, j) + \dots + (i, j)$ 之和，相当于把很多个一维的区间之和拼到了一起，那么我们在构造函数中需要建立起这样一个列之和矩阵，然后再更新某一个位置时，我们只需要将该列中改变的位置下面的所有数字更新一下即可，而在求某个区间和时，只要将相差的各列中对应的起始和结束的行上的值的差值累加起来即可，参见代码如下：

解法2：

```

1 class NumMatrix {
2 public:
3     NumMatrix(vector<vector<int>> &matrix) {
4         if (matrix.empty() || matrix[0].empty()) return;
5         mat = matrix;
6         colSum.resize(matrix.size() + 1, vector<int>(matrix[0].size(), 0));
7         for (int i = 1; i < colSum.size(); ++i) {
8             for (int j = 0; j < colSum[0].size(); ++j) {
9                 colSum[i][j] = colSum[i - 1][j] + matrix[i - 1][j];
10            }
11        }
12    }
13
14    void update(int row, int col, int val) {
15        for (int i = row + 1; i < colSum.size(); ++i) {
16            colSum[i][col] += val - mat[row][col];
17        }
18        mat[row][col] = val;
19    }
20
21    int sumRegion(int row1, int col1, int row2, int col2) {
22        int res = 0;
23        for (int j = col1; j <= col2; ++j) {
24            res += colSum[row2 + 1][j] - colSum[row1][j];
25        }
26        return res;
27    }
28
29 private:
30     vector<vector<int>> mat;
31     vector<vector<int>> colSum;
32 };

```

309. 买股票的最佳时间含冷冻期

Say you have an array for which the i th element is the price of a given stock on day i .

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times) with the following restrictions:

You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

After you sell your stock, you cannot buy stock on next day. (ie, cooldown 1 day)

Example:

```

prices = [1, 2, 3, 0, 2]
maxProfit = 3
transactions = [buy, sell, cooldown, buy, sell]

```

这道题又是关于买卖股票的问题，之前有四道类似的题目Best Time to Buy and Sell Stock 买卖股票的最佳时间，Best Time to Buy and Sell Stock II 买股票的最佳时间之二， Best Time to Buy and Sell Stock III 买股票的最佳时间之三和Best Time to Buy and Sell Stock IV 买卖股票的最佳时间之四。而这道题与上面这些不同之处在于加入了一个冷冻期Cooldown之说，就是如果某天卖了股票，那么第二天不能买股票，有一天的冷冻期。这道题我不太会，于是看到了网上大神的解法，点这里。根据他的解法，此题需要维护三个一维数组buy, sell, 和rest。其中：

buy[i]表示在第i天之前最后一个操作是买，此时的最大收益。

sell[i]表示在第i天之前最后一个操作是卖，此时的最大收益。

rest[i]表示在第i天之前最后一个操作是冷冻期，此时的最大收益。

我们写出递推式为：

```
buy[i] = max(rest[i-1] - price, buy[i-1])
sell[i] = max(buy[i-1] + price, sell[i-1])
rest[i] = max(sell[i-1], buy[i-1], rest[i-1])
```

上述递推式很好的表示了在买之前有冷冻期，买之前要卖掉之前的股票。一个小技巧是如何保证[buy, rest, buy]的情况不会出现，这是由于buy[i] <= rest[i]，即rest[i] = max(sell[i-1], rest[i-1]), 这保证了[buy, rest, buy]不会出现。

另外，由于冷冻期的存在，我们可以得出rest[i] = sell[i-1]，这样，我们可以将上面三个递推式精简到两个：

```
buy[i] = max(sell[i-2] - price, buy[i-1])
sell[i] = max(buy[i-1] + price, sell[i-1])
```

我们还可以做进一步优化，由于i只依赖于i-1和i-2，所以我们可以完成O(1)的空间复杂度完成算法，参见代码如下：

```
1 class Solution {
2 public:
3     int maxProfit(vector<int>& prices) {
4         int buy = INT_MIN, pre_buy = 0, sell = 0, pre_sell = 0;
5         for (int price : prices) {
6             pre_buy = buy;
7             buy = max(pre_sell - price, pre_buy);
8             pre_sell = sell;
9             sell = max(pre_buy + price, pre_sell);
10        }
11        return sell;
12    }
13 }
```

CPP

310. 最小高度树

For a undirected graph with tree characteristics, we can choose any node as the root. The result graph is then a rooted tree. Among all possible rooted trees, those with minimum height are called minimum height trees (MHTs). Given such a graph, write a function to find all the MHTs and return a list of their root labels.

Format

The graph contains n nodes which are labeled from 0 to n - 1. You will be given the number n and a list of undirected edges (each edge is a pair of labels).

You can assume that no duplicate edges will appear in edges. Since all edges are undirected, [0, 1] is the same as [1, 0] and thus will not appear together in edges.

这道题虽然是树的题目，但是跟其最接近的题目是Course Schedule 课程清单和Course Schedule II 课程清单之二。由于LeetCode中的树的题目主要都是针对于二叉树的，而这道题虽说是树但其实本质是想考察图的知识，这道题刚开始在拿到的时候，我最先想到的解法是遍历的点，以每个点都当做根节点，算出高度，然后找出最小的，但是一时半会又写不出程序来，于是上网看看大家的解法，发现大家推崇的方法是一个类似剥洋葱的方法，就是一层一层的褪去叶节点，最后剩下的一一个或两个节点就是我们要求的最小高度树的根节点，这种思路非常的巧妙，而且实现起来也不难，跟之前那到课程清单的题一样，我们需要建立一个图g，是一个二维数组，其中g[i]是一个一维数组，保存了i节点可以到达的所有节点。我们开始将所有只有一个连接边的节点(叶节点)都存入到一个队列queue中，然后我们遍历每一个叶节点，通过图来找到和其相连的节点，并且在其相连节点的集合中将该叶节点删去，如果删完后此节点也变成一个叶节点了，加入队列中，再下一轮删除。那么我们删到什么时候呢，当节点数小于等于2时候停止，此时剩下的一个或两个节点就是我们要求的最小高度树的根节点啦，参见代码如下：

```

1 class Solution {
2 public:
3     vector<int> findMinHeightTrees(int n, vector<pair<int, int> & edges) {
4         if (n == 1) return {0};
5         vector<int> res;
6         vector<unordered_set<int>> adj(n);
7         queue<int> q;
8         for (auto edge : edges) {
9             adj[edge.first].insert(edge.second);
10            adj[edge.second].insert(edge.first);
11        }
12        for (int i = 0; i < n; ++i) {
13            if (adj[i].size() == 1) q.push(i);
14        }
15        while (n > 2) {
16            int size = q.size();
17            n -= size;
18            for (int i = 0; i < size; ++i) {
19                int t = q.front(); q.pop();
20                for (auto a : adj[t]) {
21                    adj[a].erase(t);
22                    if (adj[a].size() == 1) q.push(a);
23                }
24            }
25        }
26        while (!q.empty()) {
27            res.push_back(q.front()); q.pop();
28        }
29        return res;
30    }
31 };

```

311. 稀疏矩阵相乘

Given two sparse matrices A and B, return the result of AB.

You may assume that A's column number is equal to B's row number.

Example:

```
A = [
  [ 1, 0, 0 ],
  [-1, 0, 3]
]
```

```
B = [
  [ 7, 0, 0 ],
  [ 0, 0, 0 ],
  [ 0, 0, 1 ]
]
```

$$AB = \begin{vmatrix} 1 & 0 & 0 \\ -1 & 0 & 3 \end{vmatrix} \times \begin{vmatrix} 7 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{vmatrix} = \begin{vmatrix} 7 & 0 & 0 \\ -7 & 0 & 3 \\ 0 & 0 & 1 \end{vmatrix}$$

这道题让我们实现稀疏矩阵相乘，稀疏矩阵的特点是矩阵中绝大多数的元素为0，而相乘的结果是还应该是稀疏矩阵，即还是大多数元素为0，那么我们使用传统的矩阵相乘的算法肯定会处理大量的0乘0的无用功，所以我们需要适当的优化算法，使其可以顺利通过OJ，我们知道一个 $i \times k$ 的矩阵A乘以一个 $k \times j$ 的矩阵B会得到一个 $i \times j$ 大小的矩阵C，那么我们来看结果矩阵中的某个元素 $C[i][j]$ 是怎么来的，起始是 $A[i][0]*B[0][j] + A[i][1]*B[1][j] + \dots + A[i][k]*B[k][j]$ ，那么为了不重复计算0乘0，我们首先遍历A数组，要确保 $A[i][k]$ 不为0，才继续计算，然后我们遍历B矩阵的第k行，如果 $B[K][j]$ 不为0，我们累加结果矩阵 $res[i][j] += A[i][k] * B[k][j]$ ；这样我们就能高效的算出稀疏矩阵的乘法，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     vector<vector<int>> multiply(vector<vector<int>>& A, vector<vector<int>>& B) {
4         vector<vector<int>> res(A.size(), vector<int>(B[0].size()));
5         for (int i = 0; i < A.size(); ++i) {
6             for (int k = 0; k < A[0].size(); ++k) {
7                 if (A[i][k] != 0) {
8                     for (int j = 0; j < B[0].size(); ++j) {
9                         if (B[k][j] != 0) res[i][j] += A[i][k] * B[k][j];
10                    }
11                }
12            }
13        }
14        return res;
15    }
16};
```

CPP

再来看另一种方法，这种方法其实核心思想跟上面那种方法相同，稍有不同的是我们用一个二维矩阵矩阵来记录每一行中，各个位置中不为0的列数和其对应的值，然后我们遍历这个二维矩阵，取出每行中不为零的列数和值，然后遍历B中对应行进行累加相乘，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<vector<int>> multiply(vector<vector<int>>& A, vector<vector<int>>& B) {
4         vector<vector<int>> res(A.size(), vector<int>(B[0].size()));
5         vector<vector<pair<int, int>>> v(A.size(), vector<pair<int, int>>());
6         for (int i = 0; i < A.size(); ++i) {
7             for (int k = 0; k < A[i].size(); ++k) {
8                 if (A[i][k] != 0) v[i].push_back({k, A[i][k]});
9             }
10        }
11        for (int i = 0; i < A.size(); ++i) {
12            for (int k = 0; k < v[i].size(); ++k) {
13                int col = v[i][k].first;
14                int val = v[i][k].second;
15                for (int j = 0; j < B[0].size(); ++j) {
16                    res[i][j] += val * B[col][j];
17                }
18            }
19        }
20        return res;
21    }
22 };

```

312. 打气球游戏

Given n balloons, indexed from 0 to $n-1$. Each balloon is painted with a number on it represented by array nums . You are asked to burst all the balloons. If you burst balloon i you will get $\text{nums}[\text{left}] * \text{nums}[i] * \text{nums}[\text{right}]$ coins. Here left and right are adjacent indices of i . After the burst, the left and right then becomes adjacent.

Find the maximum coins you can collect by bursting the balloons wisely.

Note:

- (1) You may imagine $\text{nums}[-1] = \text{nums}[n] = 1$. They are not real therefore you can not burst them.
- (2) $0 \leq n \leq 500$, $0 \leq \text{nums}[i] \leq 100$

Example:

Given [3, 1, 5, 8]

Return 167

```

nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> []
coins = 3*1*5      + 3*5*8      + 1*3*8      + 1*8*1      = 167

```

这道题提出了一种打气球的游戏，每个气球都对应着一个数字，我们每次打爆一个气球，得到的金币数是被打爆的气球的数字和其两边的气球上的数字相乘，如果旁边没有气球了，则按1算，以此类推，求能得到的最多金币数。像这种求极值问题，我们一般都要考虑用动态规划Dynamic Programming来做，我们维护一个二维动态数组 dp ，其中 $\text{dp}[i][j]$ 表示打爆区间 $[i, j]$ 中的所有气球能得到的最多金币。题目中说明了边界情况，当气球周围没有气球的时候，旁边的数字按1算，这样我们可以在原数组两边各填充一个1，这样方便于计算。这道题的最难点就是找递归式，如下所示：

```

dp[i][j] = max(dp[i][j], nums[i - 1]*nums[k]*nums[j + 1] + dp[i][k - 1] + dp[k + 1][j])
( i <= k <= j )

```

有了递推式，我们可以写代码，我们其实只是更新了 dp 数组的右上三角区域，我们最终要返回的值存在 $\text{dp}[1][n]$ 中，其中 n 是两端添加1之前数组 nums 的个数。参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     int maxCoins(vector<int>& nums) {
4         int n = nums.size();
5         nums.insert(nums.begin(), 1);
6         nums.push_back(1);
7         vector<vector<int> > dp(nums.size(), vector<int>(nums.size() , 0));
8         for (int len = 1; len <= n; ++len) {
9             for (int left = 1; left <= n - len + 1; ++left) {
10                 int right = left + len - 1;
11                 for (int k = left; k <= right; ++k) {
12                     dp[left][right] = max(dp[left][right], nums[left - 1] * nums[k] *
13                         nums[right + 1] + dp[left][k - 1] + dp[k + 1][right]);
14                 }
15             }
16         }
17         return dp[1][n];
18     }
19 };

```

CPP

对于题目中的例子[3, 1, 5, 8]，得到的dp数组如下：

0	0	0	0	0	0
0	3	30	159	167	0
0	0	15	135	159	0
0	0	0	40	48	0
0	0	0	0	40	0
0	0	0	0	0	0

这题还有递归解法，思路都一样，就是写法略有不同，参见代码如下：

解法2:

```

1 class Solution {
2 public:
3     int maxCoins(vector<int>& nums) {
4         int n = nums.size();
5         nums.insert(nums.begin(), 1);
6         nums.push_back(1);
7         vector<vector<int> > dp(nums.size(), vector<int>(nums.size() , 0));
8         return burst(nums, dp, 1 , n);
9     }
10    int burst(vector<int> &nums, vector<vector<int> > &dp, int left, int right) {
11        if (left > right) return 0;
12        if (dp[left][right] > 0) return dp[left][right];
13        int res = 0;
14        for (int k = left; k <= right; ++k) {
15            res = max(res, nums[left - 1] * nums[k] * nums[right + 1] + burst(nums, dp,
16 left, k - 1) + burst(nums, dp, k + 1, right));
17        }
18        dp[left][right] = res;
19        return res;
20    }
21 };

```

CPP

313. 超级丑陋数

Write a program to find the nth super ugly number.

Super ugly numbers are positive numbers whose all prime factors are in the given prime list primes of sizek. For example, [1, 2, 4, 7, 8, 13, 14, 16, 19, 26, 28, 32] is the sequence of the first 12 super ugly numbers given primes = [2, 7, 13, 19] of size 4.

Note:

- (1) 1 is a super ugly number for any given primes.
- (2) The given numbers in primes are in ascending order.
- (3) $0 < k \leq 100$, $0 < n \leq 106$, $0 < \text{primes}[i] < 1000$.

这道题让我们求超级丑陋数，是之前那两道Ugly Number 丑陋数和Ugly Number II 丑陋数之二的延伸，质数集合可以任意给定，这就增加了难度。但是本质上和Ugly Number II 丑陋数之二没有什么区别，由于我们不知道质数的个数，我们可以用一个idx数组来保存当前的位置，然后我们从每个子链中取出一个数，找出其中最小值，然后更新idx数组对应位置，注意有可能最小值不止一个，要更新所有最小值的位置，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int nthSuperUglyNumber(int n, vector<int>& primes) {
4         vector<int> res(1, 1), idx(primes.size(), 0);
5         while (res.size() < n) {
6             vector<int> tmp;
7             int mn = INT_MAX;
8             for (int i = 0; i < primes.size(); ++i) {
9                 tmp.push_back(res[idx[i]] * primes[i]);
10            }
11            for (int i = 0; i < primes.size(); ++i) {
12                mn = min(mn, tmp[i]);
13            }
14            for (int i = 0; i < primes.size(); ++i) {
15                if (mn == tmp[i]) ++idx[i];
16            }
17            res.push_back(mn);
18        }
19        return res.back();
20    }
21 };

```

CPP

上述代码可以稍稍改写一下，变得更简洁一些，原理完全相同，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int nthSuperUglyNumber(int n, vector<int>& primes) {
4         vector<int> dp(n, 1), idx(primes.size(), 0);
5         for (int i = 1; i < n; ++i) {
6             dp[i] = INT_MAX;
7             for (int j = 0; j < primes.size(); ++j) {
8                 dp[i] = min(dp[i], dp[idx[j]] * primes[j]);
9             }
10            for (int j = 0; j < primes.size(); ++j) {
11                if (dp[i] == dp[idx[j]] * primes[j]) {
12                    ++idx[j];
13                }
14            }
15        }
16        return dp.back();
17    }
18 };

```

314. 二叉树的竖直遍历

Given a binary tree, return the vertical order traversal of its nodes' values. (ie, from top to bottom, column by column).

If two nodes are in the same row and column, the order should be from left to right.

Examples:

Given binary tree [3,9,20,null,null,15,7],

```

3
/
9  20
/
15  7

```

return its vertical order traversal as:

```

[
[9],
[3,15],
[20],
[7]
]
```

这道题让我们竖直遍历二叉树，并把每一列存入一个二维数组，我们看题目中给的第一个例子，3和15属于同一列，3在前，第二个例子中，3,5,2在同一列，3在前，5和2紧随其后，那么我们隐约的可以感觉到好像是一种层序遍历的前后顺序，那么我们如何来确定列的顺序呢，我们可以把根节点给个序号0，然后开始层序遍历，凡是左子节点则序号减1，右子节点序号加1，这样我们可以通过序号来把相同列的节点值放到一起，我们用一个TreeMap来建立序号和其对应的节点值的映射，用TreeMap的另一个好处是其自动排序功能可以让我们的列从左到右，由于层序遍历需要用到queue，我们此时queue里不能只存节点，而是要存序号和节点组成的pair，这样我们每次取出就可以操作序号，而且排入队中的节点也赋上其正确的序号，代码如下：

```

1 class Solution {
2 public:
3     vector<vector<int>> verticalOrder(TreeNode* root) {
4         vector<vector<int>> res;
5         if (!root) return res;
6         map<int, vector<int>> m;
7         queue<pair<int, TreeNode*>> q;
8         q.push({0, root});
9         while (!q.empty()) {
10             auto a = q.front(); q.pop();
11             m[a.first].push_back(a.second->val);
12             if (a.second->left) q.push({a.first - 1, a.second->left});
13             if (a.second->right) q.push({a.first + 1, a.second->right});
14         }
15         for (auto a : m) {
16             res.push_back(a.second);
17         }
18         return res;
19     }
20 };

```

315. 计算后面较小数字的个数

You are given an integer array nums and you have to return a new counts array. The counts array has the property where counts[i] is the number of smaller elements to the right of nums[i].

Example:

Given nums = [5, 2, 6, 1]

To the right of 5 there are 2 smaller elements (2 and 1).
To the right of 2 there is only 1 smaller element (1).
To the right of 6 there is 1 smaller element (1).
To the right of 1 there is 0 smaller element.
Return the array [2, 1, 1, 0].

这道题给定我们一个数组，让我们计算每个数字右边所有小于这个数字的个数，目测我们不能用brute force，OJ肯定不答应，那么我们为了提高运算效率，首先可以使用用二分搜索法，思路是将给定数组从最后一个开始，用二分法插入到一个新的数组，这样新数组就是有序的，那么此时该数字在新数组中的坐标就是原数组中其右边所有较小数字的个数，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<int> countSmaller(vector<int>& nums) {
4         vector<int> t, res(nums.size());
5         for (int i = nums.size() - 1; i >= 0; --i) {
6             int left = 0, right = t.size();
7             while (left < right) {
8                 int mid = left + (right - left) / 2;
9                 if (t[mid] >= nums[i]) right = mid;
10                else left = mid + 1;
11            }
12            res[i] = right;
13            t.insert(t.begin() + right, nums[i]);
14        }
15        return res;
16    }
17 };

```

上面使用二分搜索法是一种插入排序的做法，我们还可以用C++中的STL的一些自带的函数来帮助我们，比如求距离distance，或是求第一个不小于当前数字的函数lower_bound，这里利用这两个函数代替了上一种方法中的二分搜索的部分，两种方法的核心思想都是相同的，构造有序数组，找出新加进来的数组在有序数组中对应的位置存入结果中即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<int> countSmaller(vector<int>& nums) {
4         vector<int> t, res(nums.size());
5         for (int i = nums.size() - 1; i >= 0; --i) {
6             int d = distance(t.begin(), lower_bound(t.begin(), t.end(), nums[i]));
7             res[i] = d;
8             t.insert(t.begin() + d, nums[i]);
9         }
10        return res;
11    }
12 };

```

再来看一种利用二分搜索树来解的方法，我们来构造一棵二分搜索树，稍有不同的地方是我们需要加一个变量smaller来记录比当前节点值小的所有节点的个数，我们每插入一个节点，会判断其和根节点的大小，如果新的节点值小于根节点值，则其会插入到左子树中，我们此时要增加根节点的smaller，并继续递归调用左子节点的insert。如果节点值大于根节点值，则需要递归调用右子节点的insert并加上根节点的smaller，并加1，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     struct Node {
4         int val, smaller;
5         Node *left, *right;
6         Node(int v, int s) : val(v), smaller(s), left(NULL), right(NULL) {}
7     };
8     int insert(Node *&root, int v) {
9         if (!root) return (root = new Node(v, 0)), 0;
10        if (root->val > v) return root->smaller++, insert(root->left, v);
11        else return insert(root->right, v) + root->smaller + (root->val < v ? 1 : 0);
12    }
13    vector<int> countSmaller(vector<int>& nums) {
14        vector<int> res(nums.size());
15        Node *root = NULL;
16        for (int i = nums.size() - 1; i >= 0; --i) {
17            res[i] = insert(root, nums[i]);
18        }
19        return res;
20    }
21 };

```

316. 移除重复字母

Given a string which contains only lowercase letters, remove duplicate letters so that every letter appear once and only once. You must make sure your result is the smallest in lexicographical order among all possible results.

Example:

Given "bcabc"
Return "abc"

Given "cbacdcbc"
Return "acdb"

这道题让我们移除重复字母，使得每个字符只能出现一次，而且结果要按字母顺序排，前提是不能打乱其原本的相对位置。我们的解题思路是：先建立一个哈希表来统计每个字母出现的次数，还需要一个visited数字来纪录每个字母是否被访问过，我们遍历整个字符串，对于遍历到的字符，先在哈希表中将其值减一，然后看visited中是否被访问过，若访问过则继续循环，说明该字母已经出现在结果中并且位置已经安排妥当。如果没访问过，我们和结果中最后一个字母比较，如果该字母的ASCII码小并且结果中的最后一个字母在哈希表中的值不为0(说明后面还会出现这个字母)，那么我们此时就要在结果中删去最后一个字母且将其标记为未访问，然后加上当前遍历到的字母，并且将其标记为已访问，以此类推直至遍历完整个字符串s，此时结果里的字符串即为所求。这里有个小技巧，我们一开始给结果字符串res中放个"0"，就是为了在第一次比较时方便，如果为空就没法和res中的最后一个字符比较了，而'0'的ASCII码要小于任意一个字母的，所以不会有错误。最后我们返回结果时再去掉开头那个'0'即可，参见代码如下：

```

1 class Solution {
2 public:
3     string removeDuplicateLetters(string s) {
4         int m[256] = {0}, visited[256] = {0};
5         string res = "0";
6         for (auto a : s) ++m[a];
7         for (auto a : s) {
8             --m[a];
9             if (visited[a]) continue;
10            while (a < res.back() && m[res.back()]) {
11                visited[res.back()] = 0;
12                res.pop_back();
13            }
14            res += a;
15            visited[a] = 1;
16        }
17        return res.substr(1);
18    }
19 };

```

317. 建筑物的最短距离

You want to build a house on an empty land which reaches all buildings in the shortest amount of distance. You can only move up, down, left and right. You are given a 2D grid of values 0, 1 or 2, where:

Each 0 marks an empty land which you can pass by freely.

Each 1 marks a building which you cannot pass through.

Each 2 marks an obstacle which you cannot pass through.

For example, given three buildings at (0,0), (0,4), (2,2), and an obstacle at (0,2):

```

1 - 0 - 2 - 0 - 1
|   |   |   |
0 - 0 - 0 - 0 - 0
|   |   |   |
0 - 0 - 1 - 0 - 0

```

The point (1,2) is an ideal empty land to build a house, as the total travel distance of $3+3+1=7$ is minimal. So return 7.

Note:

There will be at least one building. If it is not possible to build such house according to the above rules, return -1.

这道题给我们了一些建筑物的坐标和一些障碍物的坐标，让我们找一个位置，使其到所有建筑物的曼哈顿距离之和最小，起初我觉得这题应该算Best Meeting Point那道题的拓展，不同之处在于这道题有了障碍物的存在，这样就使得直接使用曼哈顿距离的计算公式变得不可行，因为在有些情况下，障碍物完全封死了某个建筑物，那么这时候应该返回-1。所以这道题只能使用遍历迷宫的思想来解，那么这题就和之前那道Walls and Gates很类似，但是这道题用DFS就会很麻烦，因为我们的目标是要建立Distance Map，所以BFS的特性使得其非常适合建立距离场，而DFS由于是沿着一个方向一股脑的搜索，然后会面临着更新距离的问题，只有当递归函数都调用结束后，距离场才建立好，那么我们累加距离场时又得整个遍历一遍，非常不高效。主要原因还是由于DFS的搜索方式不适合距离场，因为BFS遍历完一个点后，不会再更改这个点的值，而DFS会反复的更改同一个点的值，我强行用DFS写出的方法无法通过OJ最后一个大集合，所以这道题还是老老实实地用BFS来解题吧，还是需要借助queue来遍历，我们对于每一个建筑的位置都进行一次全图的BFS遍历，每次都建立一个dist的距离场，由于我们BFS遍历需要标记应经访问过的位置，而我们并不想建立一个visit的二维矩阵，那么怎么办呢，这里用一个小trick，我们第一遍历的时候，都是找0的位置，遍历完后，我们将其赋为-1，这样下一轮遍历我们就找-1的位置，然后将其都赋为-2，以此类推直至遍历完所有的建筑物，

然后在遍历的过程中更新dist和sum的值，注意我们的dist算是个局部变量，每次都初始化为grid，真正的距离场累加在sum中，由于建筑的位置在grid中是1，所以dist中初始化也是1，累加到sum中就需要减1，我们用sum中的值来更新结果res的值，最后根据res的值看是否要返回-1，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int shortestDistance(vector<vector<int>>& grid) {
4         int res = INT_MAX, val = 0, m = grid.size(), n = grid[0].size();
5         vector<vector<int>> sum = grid;
6         vector<vector<int>> dirs{{0,-1},{-1,0},{0,1},{1,0}};
7         for (int i = 0; i < grid.size(); ++i) {
8             for (int j = 0; j < grid[i].size(); ++j) {
9                 if (grid[i][j] == 1) {
10                     res = INT_MAX;
11                     vector<vector<int>> dist = grid;
12                     queue<pair<int, int>> q;
13                     q.push({i, j});
14                     while (!q.empty()) {
15                         int a = q.front().first, b = q.front().second; q.pop();
16                         for (int k = 0; k < dirs.size(); ++k) {
17                             int x = a + dirs[k][0], y = b + dirs[k][1];
18                             if (x >= 0 && x < m && y >= 0 && y < n && grid[x][y] == val) {
19                                 --grid[x][y];
20                                 dist[x][y] = dist[a][b] + 1;
21                                 sum[x][y] += dist[x][y] - 1;
22                                 q.push({x, y});
23                                 res = min(res, sum[x][y]);
24                             }
25                         }
26                     }
27                     --val;
28                 }
29             }
30         }
31         return res == INT_MAX ? -1 : res;
32     }
33 };

```

下面这种方法也是网上比较流行的解法，我们还是用BFS来做，其中dist是累加距离场，cnt表示某个位置已经计算过的建筑数，变量buildingCnt为建筑的总数，我们还是用queue来辅助计算，注意这里的dist的更新方式跟上面那种方法的不同，这里的dist由于是累积距离场，所以不能用dist其他位置的值来更新，而是需要直接加上和建筑物之间的距离，这里用level来表示，每遍历一层，level自增1，这样我们就需要所加个for循环，来控制每一层中的level值是相等的，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int shortestDistance(vector<vector<int>>& grid) {
4         int res = INT_MAX, buildingCnt = 0, m = grid.size(), n = grid[0].size();
5         vector<vector<int>> dist(m, vector<int>(n, 0)), cnt = dist;
6         vector<vector<int>> dirs{{0,-1},{-1,0},{0,1},{1,0}};
7         for (int i = 0; i < m; ++i) {
8             for (int j = 0; j < n; ++j) {
9                 if (grid[i][j] == 1) {
10                     ++buildingCnt;
11                     queue<pair<int, int>> q;
12                     q.push({i, j});
13                     vector<vector<bool>> visited(m, vector<bool>(n, false));
14                     int level = 1;
15                     while (!q.empty()) {
16                         int size = q.size();
17                         for (int s = 0; s < size; ++s) {
18                             int a = q.front().first, b = q.front().second; q.pop();
19                             for (int k = 0; k < dirs.size(); ++k) {
20                                 int x = a + dirs[k][0], y = b + dirs[k][1];
21                                 if (x >= 0 && x < m && y >= 0 && y < n && grid[x][y] == 0
22 && !visited[x][y]) {
23                                     dist[x][y] += level;
24                                     ++cnt[x][y];
25                                     visited[x][y] = true;
26                                     q.push({x, y});
27                                 }
28                             }
29                         }
30                         ++level;
31                     }
32                 }
33             }
34         }
35         for (int i = 0; i < m; ++i) {
36             for (int j = 0; j < n; ++j) {
37                 if (grid[i][j] == 0 && cnt[i][j] == buildingCnt) {
38                     res = min(res, dist[i][j]);
39                 }
40             }
41         }
42         return res == INT_MAX ? -1 : res;
43     }
44 };

```

318. 单词长度的最大积

Given a string array words, find the maximum value of length(word[i]) * length(word[j]) where the two words do not share common letters. You may assume that each word will contain only lower case letters. If no such two words exist, return 0.

Example 1:

Given ["abcw", "baz", "foo", "bar", "xtfn", "abcdef"]
 Return 16
 The two words can be "abcw", "xtfn".

这道题给我们了一个单词数组，让我们求两个没有相同字母的单词的长度之积的最大值。我开始想的方法是每两个单词先比较，如果没有相同字母，则计算其长度之积，然后每次更新结果就能找到最大值。但是我开始想的两个单词比较的方法是利用哈希表先将一个单词的所有出现的字母存入哈希表，然后检查另一个单词的各个字母是否在哈希表出现过，若都没出现过，则说明两个单词没有相同字母，则计算两个单词长度之积并更新结果。但是这种判断方法无法通过OJ的大数据集，上网搜大神们的解法，都是用了mask，因为题目中说都是小写字母，那么只有26位，一个整型数int有32位，我们可以用后26位来对应26个字母，若为1，说明该对应位置的字母出现过，那么每个单词的都可由一个int数字表示，两个单词没有共同字母的条件是这两个int数想与为0，用这个判断方法可以通过OJ，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     int maxProduct(vector<string>& words) {
4         int res = 0;
5         vector<int> mask(words.size(), 0);
6         for (int i = 0; i < words.size(); ++i) {
7             for (char c : words[i]) {
8                 mask[i] |= 1 << (c - 'a');
9             }
10            for (int j = 0; j < i; ++j) {
11                if (!(mask[i] & mask[j])) {
12                    res = max(res, int(words[i].size() * words[j].size()));
13                }
14            }
15        }
16        return res;
17    }
18};
```

CPP

还有一种写法，借助哈希表，映射每个mask的值和其单词的长度，每算出一个单词的mask，遍历哈希表里的值，如果和其中的mask值相与为0，则将当前单词的长度和哈希表中存的单词长度相乘并更新结果，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     int maxProduct(vector<string>& words) {
4         int res = 0;
5         unordered_map<int, int> m;
6         for (string word : words) {
7             int mask = 0;
8             for (char c : word) {
9                 mask |= 1 << (c - 'a');
10            }
11            m[mask] = max(m[mask], int(word.size()));
12            for (auto a : m) {
13                if (!(mask & a.first)) {
14                    res = max(res, (int)word.size() * a.second);
15                }
16            }
17        }
18        return res;
19    }
20};
```

CPP

319. 灯泡开关

There are n bulbs that are initially off. You first turn on all the bulbs. Then, you turn off every second bulb. On the third round, you toggle every third bulb (turning on if it's off or turning off if it's on). For the nth round, you only toggle the last bulb. Find how many bulbs are on after n rounds.

Example:

Given n = 3.

At first, the three bulbs are [off, off, off].

After first round, the three bulbs are [on, on, on].

After second round, the three bulbs are [on, off, on].

After third round, the three bulbs are [on, off, off].

So you should return 1, because there is only one bulb is on.

这道题给了我们n个灯泡，第一次打开所有的灯泡，第二次每两个更改灯泡的状态，第三次每三个更改灯泡的状态，以此类推，第n次每n个更改灯泡的状态。让我们求n次后，所有亮的灯泡的个数。此题是CareerCup 6.6 Toggle Lockers 切换锁的状态。

那么我们来看这道题吧，还是先枚举个小例子来分析下，比如只有5个灯泡的情况，'X'表示灭，'√'表示亮，如下所示：

初始状态:	X	X	X	X	X
第一次:	√	√	√	√	√
第二次:	√	X	√	X	√
第三次:	√	X	X	X	√
第四次:	√	X	X	√	√
第五次:	√	X	X	√	X

那么最后我们发现五次遍历后，只有1号和4号灯泡是亮的，而且很巧的是它们都是平方数，是巧合吗，还是其中有什么玄机。我们仔细想想，对于第n个灯泡，只有当次数是n的因子的之后，才能改变灯泡的状态，即n能被当前次数整除，比如当n为36时，它的因数有(1,36), (2,18), (3,12), (4,9), (6,6)，可以看到前四个括号里成对出现的因数各不相同，括号中前面的数改变了灯泡状态，后面的数又变回去了，等于灯泡的状态没有发生变化，只有最后那个(6,6)，在次数6的时候改变了一次状态，没有对应其它的状态能将其变回去了，所以灯泡就一直是点亮状态的。所以所有平方数都有这么一个相等的因数对，即所有平方数的灯泡都将会是点亮的状态。

那么问题就简化为了求1到n之间完全平方数的个数，我们可以用force brute来比较从1开始的完全平方数和n的大小，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     int bulbSwitch(int n) {
4         int res = 1;
5         while (res * res <= n) ++res;
6         return res - 1;
7     }
8 };

```

CPP

还有一种方法更简单，我们直接对n开方，在C++里的sqrt函数返回的是一个整型数，这个整型数的平方最接近于n，即为n包含的所有完全平方数的个数，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int bulbSwitch(int n) {
4         return sqrt(n);
5     }
6 };

```

CPP

讨论：这道题有个follow up就是，如果我们toggle的顺序不是1, 2, 3, 4..., 而是1, 3, 5, 7..., 或者是2, 4, 6, 8... 的话，还怎么做？博主没有想出像解法二那样简便的方法，只是大概想了想，如果各位大神有更好的方法，请一定要在下方留言啊。博主想的是，比如对于1, 3, 5, 7..., 那么就是先把所有的灯点亮，然后关掉3, 6, 9, 12, 15...等的灯，然后toggle的是5, 10, 15...等等，然后再toggle的是7, 14, 21...，我们发现，纯2的倍数的灯永远不会被改变，比如2, 4, 8, 16... 这些灯状态不会变，有些灯只会变一次，比如3, 6, 9等，而有些灯会变两次，比如15 (3x5), 21 (3x7), 35 (5x7) 等，有些灯会变三次，比如105 (3x5x7)，那么我们可以观察出规律了，toggle的次数跟奇数因子的数字有关（注意这里的奇数因子不包括1），只要有奇数个奇因子，那么灯就是灭的，只要有偶数个奇因子，那么灯就是亮的。

320. 通用简写

Write a function to generate the generalized abbreviations of a word.

Example:

Given word = "word", return the following list (order does not matter):

```
["word", "1ord", "w1rd", "wo1d", "wor1", "2rd", "w2d", "wo2", "1o1d", "1or1", "w1r1", "1o2",
 "2r1", "3d", "w3", "4"]
```

这道题让我们对一个单词进行部分简写，简写的规则是若干个字母可以用数字来表示，但是不能有两个相邻的数字，具体可以参考题目中给的例子，根据我以往的经验，这种列举所有情况的必定是要用DFS来写的，但是我一时半会又没想到该咋递归，后来我数了一下题目中给的例子的所有情况的个数，是16个，而word有4个字母，刚好是2的4次方，这是巧合吗，当然不是，后来我又发现如果把0到15的二进制写出来，每一个可以对应一种情况，如下所示：

```

0000 word
0001 wor1
0010 wo1d
0011 wo2
0100 w1rd
0101 w1r1
0110 w2d
0111 w3
1000 1ord
1001 1or1
1010 1o1d
1011 1o2
1100 2rd
1101 2r1
1110 3d
1111 4

```

那么我们就可以观察出规律，凡是0的地方都是原来的字母，单独的1还是1，如果是若干个1连在一起的话，就要求出1的个数，用这个数字来替换对应的字母，既然规律找出来了，那么代码就很好写了，如下所示：

解法1：

```

1 class Solution {
2 public:
3     vector<string> generateAbbreviations(string word) {
4         vector<string> res;
5         for (int i = 0; i < pow(2, word.size()); ++i) {
6             string out = "";
7             int cnt = 0, t = i;
8             for (int j = 0; j < word.size(); ++j) {
9                 if (t & 1 == 1) {
10                     ++cnt;
11                     if (j == word.size() - 1) out += to_string(cnt);
12                 } else {
13                     if (cnt != 0) {
14                         out += to_string(cnt);
15                         cnt = 0;
16                     }
17                     out += word[j];
18                 }
19                 t >>= 1;
20             }
21             res.push_back(out);
22         }
23         return res;
24     }
25 };

```

CPP

上述方法返回结果的顺序为：

["word","1ord","w1rd","2rd","wo1d","1o1d","w2d","3d","wor1","1or1","w1r1","2r1","wo2","1o2","w3","4"]

我们可以对上面代码稍稍改写一下，变的稍微简洁一点：

解法2：

```

1 class Solution {
2 public:
3     vector<string> generateAbbreviations(string word) {
4         vector<string> res;
5         for (int i = 0; i < pow(2, word.size()); ++i) {
6             string out = "";
7             int cnt = 0;
8             for (int j = 0; j < word.size(); ++j) {
9                 if ((i >> j) & 1) ++cnt;
10                else {
11                    if (cnt != 0) {
12                        out += to_string(cnt);
13                        cnt = 0;
14                    }
15                    out += word[j];
16                }
17            }
18            if (cnt > 0) out += to_string(cnt);
19            res.push_back(out);
20        }
21        return res;
22    }
23 };

```

那么迭代的写法看完了，来考虑一些递归的写法吧，上网搜了一下，发现下面三种写法比较容易理解.

解法3:

```

1 class Solution {
2 public:
3     vector<string> generateAbbreviations(string word) {
4         vector<string> res{word};
5         helper(word, 0, res);
6         return res;
7     }
8     void helper(string word, int pos, vector<string> &res) {
9         for (int i = pos; i < word.size(); ++i) {
10            for (int j = 1; i + j <= word.size(); ++j) {
11                string t = word.substr(0, i);
12                t += to_string(j) + word.substr(i + j);
13                res.push_back(t);
14                helper(t, i + 1 + to_string(j).size(), res);
15            }
16        }
17    }
18 };

```

上述方法返回结果的顺序为：

["word","1ord","1o1d","1o2","1or1","2rd","2r1","3d","4","w1rd","w1r1","w2d","w3","wo1d","wo2","wor1"]

解法4:

```

1 class Solution {
2 public:
3     vector<string> generateAbbreviations(string word) {
4         vector<string> res;
5         helper(word, 0, 0, "", res);
6         return res;
7     }
8     void helper(string word, int pos, int cnt, string out, vector<string> &res) {
9         if (pos == word.size()) {
10             if (cnt > 0) out += to_string(cnt);
11             res.push_back(out);
12         } else {
13             helper(word, pos + 1, cnt + 1, out, res);
14             helper(word, pos + 1, 0, out + (cnt > 0 ? to_string(cnt) : "") + word[pos], res);
15         }
16     }
17 };

```

上述方法返回结果的顺序为：

["4","3d","2r1","2rd","1o2","1o1d","1or1","1ord","w3","w2d","w1r1","w1rd","wo2","wo1d","wor1","word"]

解法5：

```

1 class Solution {
2 public:
3     vector<string> generateAbbreviations(string word) {
4         vector<string> res;
5         res.push_back(word.size() == 0 ? "" : to_string(word.size()));
6         for (int i = 0; i < word.size(); ++i) {
7             for (auto a : generateAbbreviations(word.substr(i + 1))) {
8                 string left = i > 0 ? to_string(i) : "";
9                 res.push_back(left + word.substr(i, 1) + a);
10            }
11        }
12        return res;
13    }
14 };

```

上述方法返回结果的顺序为：

["4","w3","wo2","wor1","word","wo1d","w1r1","w1rd","w2d","1o2","1or1","1ord","1o1d","2r1","2rd","3d"]

321. 创建最大数

Given two arrays of length m and n with digits 0-9 representing two numbers. Create the maximum number of length k $\leq m + n$ from digits of the two. The relative order of the digits from the same array must be preserved. Return an array of the k digits. You should try to optimize your time and space complexity.

Example 1:

```
nums1 = [3, 4, 6, 5]
nums2 = [9, 1, 2, 5, 8, 3]
k = 5
return [9, 8, 6, 5, 3]
```

这道题给了我们两个数组，里面数字是无序的，又给我们一个k值为 $k \leftarrow m + n$ ，然后我们从两个数组中共挑出k个数，数字之间的相对顺序不变，求能组成的最大的数。这道题的难度是Hard，博主木有想出解法，参考网上大神们的解法来做的。由于k的大小不定，所以有三种可能，第一种是当k为0时，两个数组中都不取数；第二种是当k不大于其中一个数组的长度时，有可能只从一个数组中取数；第三种情况是k大于其中一个数组的长度，则需要从两个数组中分别取数，至于每个数组中取几个，每种情况都要考虑到，然后每次更新结果即可。对于分别从两个数组中取数字的情况，我们需要将两个取出的小数组混合排序成一个数组，小数组中各自的数字之间的相对顺序不变。我们还需要一个函数来从数组中取若干个数字的函数，而且取出的数要最大。比如当前数组长度为n，需要取出k个数字，我们定义一个变量 $drop = n - k$ ，表示需要丢弃的数字的个数，我们遍历数组中的数字，进行下列循环，如果此时 $drop$ 为整数，且结果数组长度不为0，结果数组的尾元素小于当前遍历的元素，则去掉结果数组的尾元素，此时 $drop$ 自减1，重复循环直至上述任意条件不满足为止，然后把当前元素加入结果数组中，最后我们返回结果数组中的前k个元素。对于两个数组的混合，我们只要从两个数组开头每次取两个，把大的加入结果数组，然后删掉这个大的，然后继续取一对比较，直到两个数组都为空停止。参见代码如下：

```

1 class Solution {
2 public:
3     vector<int> maxNumber(vector<int>& nums1, vector<int>& nums2, int k) {
4         int m = nums1.size(), n = nums2.size();
5         vector<int> res;
6         for (int i = max(0, k - n); i <= min(k, m); ++i) {
7             res = max(res, mergeVector(maxVector(nums1, i), maxVector(nums2, k - i)));
8         }
9         return res;
10    }
11    vector<int> maxVector(vector<int> nums, int k) {
12        int drop = nums.size() - k;
13        vector<int> res;
14        for (int num : nums) {
15            while (drop && res.size() && res.back() < num) {
16                res.pop_back();
17                --drop;
18            }
19            res.push_back(num);
20        }
21        res.resize(k);
22        return res;
23    }
24    vector<int> mergeVector(vector<int> nums1, vector<int> nums2) {
25        vector<int> res;
26        while (nums1.size() + nums2.size()) {
27            vector<int> &tmp = nums1 > nums2 ? nums1 : nums2;
28            res.push_back(tmp[0]);
29            tmp.erase(tmp.begin());
30        }
31        return res;
32    }
33 };

```

322. 硬币找零

You are given coins of different denominations and a total amount of money amount. Write a function to compute the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1.

Example 1:

```

coins = [1, 2, 5], amount = 11
return 3 (11 = 5 + 5 + 1)

```

这道题给我们了一些可用的硬币值，又给了一个钱数，问我们最小能用几个硬币来找零。根据题目中的例子可知，不是每次都会给全1,2,5的硬币，有时候没有1分硬币，那么有的钱数就没法找零，需要返回-1。这道题跟CareerCup上的那道9.8 Represent N Cents 美分的组成有些类似，那道题给全了所有的美分，25,10,5,1，然后给我们一个钱数，问我们所有能够找零的方法，而这道题只让我们求出最小的那种。没啥特别好的思路就首先来考虑brute force吧，暴力搜索如果也没思路肿么办-_-|||。那我们怎么办，还是来看例子1，如果不考虑代码实现，你怎么手动找出答案。博主会先取出一个最大的数字5，比目标值11要小，由于这里的硬币是可以重复使用的，所以博主会再取个5出来，现在是10，还是比11要小，这是再取5会超，那就往前取，取2，也会超出，于是就取1，刚好是11。那么我们的暴力搜索法也是这种思路，首先要给数组排个序，因为我们想要从最大的开始取，我们的递归函数需要一个变量start，初识化为数组的最后一个位置，当前目标值target，还有当前使用的硬币个数cur，以及最终结果res。在递归函数，我们首先判断如果target小于0了，直接返回。若target为0了，说明我们当前使用的硬币已经组成了目标值，用cur来更新结果res。否则就从start开始往前遍历硬币，对每个硬币都调用递归函数，此时target应该减去当前的硬币值，cur应该自增1，代码参见评论区七楼。但是暴力搜索Brute Force的方法会超时TLE，所以我们考虑一下其他的方法吧。

如果大家刷题有一阵子了的，那么应该会知道，对于求极值问题，我们还是主要考虑动态规划Dynamic Programming来做，好处是保留了一些中间状态的计算值，这样可以避免大量的重复计算。我们维护一个一维动态数组dp，其中dp[i]表示钱数为i时的最小硬币数的找零，注意由于数组是从0开始的，所以我们要多申请一位，数组大小为amount+1，这样最终结果就可以保存在dp[amount]中了。初始化dp[0] = 0，因为目标值若为0时，就不需要硬币了。其他值可以初始化为整型最大值，或者是amount+1，为啥呢，因为最小的硬币是1，所以amount最多需要amount个硬币，amount+1也就相当于整型最大值的作用了。好，接下来就是要找状态转移方程了，没思路？不要紧！回归例子1，假设我取了一个值为5的硬币，那么由于目标值是11，所以是不是假如我们知道dp[6]，那么就知道了组成11的dp值了？所以我们更新dp[i]的方法就是遍历每个硬币，如果遍历到的硬币值小于i值（比如我们不能用值为5的硬币去更新dp[3]）时，我们用 $dp[i - coins[j]] + 1$ 来更新dp[i]，所以状态转移方程为：

$$dp[i] = \min(dp[i], dp[i - coins[j]] + 1);$$

其中coins[j]为第j个硬币，而 $i - coins[j]$ 为钱数i减去其中一个硬币的值，剩余的钱数在dp数组中找到值，然后加1和当前dp数组中的值做比较，取较小的那个更新dp数组。先来看迭代的写法如下所示：

解法1：

```
1 class Solution {
2 public:
3     int coinChange(vector<int>& coins, int amount) {
4         vector<int> dp(amount + 1, amount + 1);
5         dp[0] = 0;
6         for (int i = 1; i <= amount; ++i) {
7             for (int j = 0; j < coins.size(); ++j) {
8                 if (coins[j] <= i) {
9                     dp[i] = min(dp[i], dp[i - coins[j]] + 1);
10                }
11            }
12        }
13        return (dp[amount] > amount) ? -1 : dp[amount];
14    }
15};
```

CPP

迭代的DP解法有一个好基友，就是递归+memo数组的解法，说其是递归形式的DP解法也没错，但博主比较喜欢说成是递归加记忆数组。其目的都是为了保存中间计算结果，避免大量的重复计算，从而提高运算效率，思路都一样，仅仅是写法有些区别：

解法2：

```
1 class Solution {
2 public:
3     int coinChange(vector<int>& coins, int amount) {
4         vector<int> memo(amount + 1, INT_MAX);
5         memo[0] = 0;
6         return coinChangeDFS(coins, amount, memo);
7     }
8     int coinChangeDFS(vector<int>& coins, int target, vector<int>& memo) {
9         if (target < 0) return -1;
10        if (memo[target] != INT_MAX) return memo[target];
11        for (int i = 0; i < coins.size(); ++i) {
12            int tmp = coinChangeDFS(coins, target - coins[i], memo);
13            if (tmp >= 0) memo[target] = min(memo[target], tmp + 1);
14        }
15        return memo[target] = (memo[target] == INT_MAX) ? -1 : memo[target];
16    }
17};
```

CPP

再来看一种使用HashMap来当记忆数组的递归解法：

解法3：

```

1 class Solution {
2 public:
3     int coinChange(vector<int>& coins, int amount) {
4         unordered_map<int, int> memo;
5         memo[0] = 0;
6         return coinChangeDFS(coins, amount, memo);
7     }
8     int coinChangeDFS(vector<int>& coins, int target, unordered_map<int, int>& memo) {
9         if (target < 0) return -1;
10        if (memo.count(target)) return memo[target];
11        int cur = INT_MAX;
12        for (int i = 0; i < coins.size(); ++i) {
13            int tmp = coinChangeDFS(coins, target - coins[i], memo);
14            if (tmp >= 0) cur = min(cur, tmp + 1);
15        }
16        return memo[target] = (cur == INT_MAX) ? -1 : cur;
17    }
18 };

```

CPP

难道这题一定要DP来做吗，我们来看网友hello_world00提供的一种解法，这其实是对暴力搜索的解法做了很好的优化，不仅不会TLE，而且击败率相当的高！对比Brute Force的方法，这里在递归函数中做了很好的优化。首先是判断start是否小于0，因为我们需要从coin中取硬币，不能越界。下面就是优化的核心了，看target是否能整除coins[start]，这是相当叼的一步，比如假如我们的目标值是15，如果我们当前取出了大小为5的硬币，我们做除法，可以立马知道只用大小为5的硬币就可以组成目标值target，那么我们用cur + target/coins[start] 来更新结果res。之后的for循环也相当叼，不像暴力搜索中的那样从start位置开始往前遍历coins中的硬币，而是遍历 target/coins[start] 的次数，由于不能整除，我们只需要对余数调用递归函数，而且我们要把次数每次减1，并且再次求余数。举个例子，比如coins=[1,2,3]，amount=11，那么11除以3，得3余2，那么我们的i从3开始遍历，这里有一步非常有用的剪枝操作，没有这一步，还是会TLE，而加上了这一步，直接击败百分之九十九以上，可以说是天壤之别。那就是判断若 $cur + i \geq res - 1$ 成立，直接break，不调用递归。这里解释一下， $cur + i$ 自不必说，是当前硬币个数 cur 加上新加的 i 个硬币，我们都是知道 $cur + i$ 如果大于等于 res 的话，那么 res 是不会被更新的，那么为啥这里是大于等于 $res - 1$ 呢？因为能运行到这一步，说明之前是无法整除的，那么余数一定存在，所以再次调用递归函数的target不为0，那么如果整除的话， cur 至少会加上1，所以又跟 res 相等了，还是不会使得 res 变得更小。解释到这里应该比较明白了吧，有疑问的请在下方留言哈，参见代码如下：

解法4：

```

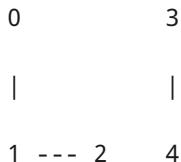
1 class Solution {
2 public:
3     int coinChange(vector<int>& coins, int amount) {
4         int res = INT_MAX, n = coins.size();
5         sort(coins.begin(), coins.end());
6         helper(coins, n - 1, amount, 0, res);
7         return (res == INT_MAX) ? -1 : res;
8     }
9     void helper(vector<int>& coins, int start, int target, int cur, int& res) {
10        if (start < 0) return;
11        if (target % coins[start] == 0) {
12            res = min(res, cur + target / coins[start]);
13            return;
14        }
15        for (int i = target / coins[start]; i >= 0; --i) {
16            if (cur + i >= res - 1) break;
17            helper(coins, start - 1, target - i * coins[start], cur + i, res);
18        }
19    }
20};

```

323. 无向图中的连通区域的个数

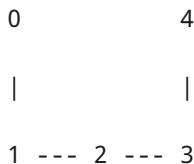
Given n nodes labeled from 0 to $n - 1$ and a list of undirected edges (each edge is a pair of nodes), write a function to find the number of connected components in an undirected graph.

Example 1:



Given $n = 5$ and $\text{edges} = [[0, 1], [1, 2], [3, 4]]$, return 2.

Example 2:



Given $n = 5$ and $\text{edges} = [[0, 1], [1, 2], [2, 3], [3, 4]]$, return 1.

Note:

You can assume that no duplicate edges will appear in edges. Since all edges are undirected, $[0, 1]$ is the same as $[1, 0]$ and thus will not appear together in edges.

这道题让我们求无向图中连通区域的个数，LeetCode中关于图Graph的题屈指可数，解法都有类似的特点，都是要先构建邻接链表Adjacency List来做。这道题的一种解法是利用DFS来做，思路是给每个节点都有个flag标记其是否被访问过，对于一个未访问过的节点，我们将结果自增1，因为这肯定是一个新的连通区域，然后我们通过邻接链表来遍历与其相邻的节点，并将他们都标记成已访问过，遍历完所有的连通节点后我们继续寻找下一个未访问过的节点，以此类推直至所有的节点都被访问过了，那么此时我们也就求出来了连通区域的个数。

解法1：

```

1 class Solution {
2 public:
3     int countComponents(int n, vector<pair<int, int> &> edges) {
4         int res = 0;
5         vector<vector<int> > g(n);
6         vector<bool> v(n, false);
7         for (auto a : edges) {
8             g[a.first].push_back(a.second);
9             g[a.second].push_back(a.first);
10        }
11        for (int i = 0; i < n; ++i) {
12            if (!v[i]) {
13                ++res;
14                dfs(g, v, i);
15            }
16        }
17        return res;
18    }
19    void dfs(vector<vector<int> > &g, vector<bool> &v, int i) {
20        if (v[i]) return;
21        v[i] = true;
22        for (int j = 0; j < g[i].size(); ++j) {
23            dfs(g, v, g[i][j]);
24        }
25    }
26};

```

CPP

这道题还有一种比较巧妙的方法，不用建立邻接链表，也不用DFS，思路是建立一个root数组，下标和节点值相同，此时root[i]表示节点i属于group i，我们初始化了n个部分(res = n)，假设开始的时候每个节点都属于一个单独的区间，然后我们开始遍历所有的edge，对于一条边的两个点，他们起始时在root中的值不相同，这时候我们将结果减1，表示少了一个区间，然后更新其中一个节点的root值，使两个节点的root值相同，那么这样我们就能把连通区间的所有节点的root值都标记成相同的值，不同连通区间的root值不相同，这样也能找出连通区间的个数。

解法2：

```

1 class Solution {
2 public:
3     int countComponents(int n, vector<pair<int, int> &> edges) {
4         int res = n;
5         vector<int> root(n);
6         for (int i = 0; i < n; ++i) root[i] = i;
7         for (auto a : edges) {
8             int x = find(root, a.first), y = find(root, a.second);
9             if (x != y) {
10                 --res;
11                 root[y] = x;
12             }
13         }
14     return res;
15 }
16 int find(vector<int> &root, int i) {
17     while (root[i] != i) i = root[i];
18     return i;
19 }
20 };

```

324. 摆动排序之二

Given an unsorted array nums, reorder it such that nums[0] < nums[1] > nums[2] < nums[3]....

Example 1:

Input: nums = [1, 5, 1, 1, 6, 4]
Output: One possible answer is [1, 4, 1, 5, 1, 6].

这道题给了我们一个无序数组，让我们排序成摆动数组，满足 $\text{nums}[0] < \text{nums}[1] > \text{nums}[2] < \text{nums}[3] \dots$ ，并给了我们例子。我们可以先给数组排序，然后在做调整。调整的方法是找到数组的中间的数，相当于把有序数组从中间分成两部分，然后从前半段的末尾取一个，在从后半的末尾去一个，这样保证了第一个数小于第二个数，然后从前半段取倒数第二个，从后半段取倒数第二个，这保证了第二个数大于第三个数，且第三个数小于第四个数，以此类推直至都取完，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     void wiggleSort(vector<int>& nums) {
4         vector<int> tmp = nums;
5         int n = nums.size(), k = (n + 1) / 2, j = n;
6         sort(tmp.begin(), tmp.end());
7         for (int i = 0; i < n; ++i) {
8             nums[i] = i & 1 ? tmp[--j] : tmp[--k];
9         }
10    }
11 };

```

这道题的Follow up让我们用O(n)的时间复杂度和O(1)的空间复杂度，这个真的比较难，参见网友的解答，(未完待续。。)

解法2:

```

1 class Solution {
2 public:
3     void wiggleSort(vector<int>& nums) {
4         #define A(i) nums[(1 + 2 * i) % (n | 1)]
5         int n = nums.size(), i = 0, j = 0, k = n - 1;
6         auto midptr = nums.begin() + n / 2;
7         nth_element(nums.begin(), midptr, nums.end());
8         int mid = *midptr;
9         while (j <= k) {
10             if (A(j) > mid) swap(A(i++), A(j++));
11             else if (A(j) < mid) swap(A(j), A(k--));
12             else ++j;
13         }
14     }
15 };

```

325. 最大子数组之和为k

Given an array `nums` and a target value `k`, find the maximum length of a subarray that sums to `k`. If there isn't one, return 0 instead.

Example 1:

Given `nums` = [1, -1, 5, -2, 3], `k` = 3,
return 4. (because the subarray [1, -1, 5, -2] sums to 3 and is the longest)

这道题给我们一个一维数组`nums`, 让我们求和为`k`最大子数组, 默认子数组必须连续, 题目中提醒我们必须要在O(n)的时间复杂度完成, 我试了下brute force无法通过OJ, 那么根据题目中的提示标签, 我们需要用哈希表和累积和来做, 关于累积和的用法可以参看我之前的博客Range Sum Query - Immutable, 那么建立累积和的好处显而易见, 如果当前累积和正好等于`k`, 那么从开头到此位置的子数组就是一个符合要求的解, 但不一定是最长的子数组, 而使用哈希表来建立累积和和其坐标之间的映射, 我们就从题目中给的例子进行分析:

`nums`: [1, -1, 5, -2, 3], `k` = 3

`sums`: [1, 0, 5, 3, 6]

我们可以看到累积和的第四个数字为3, 和`k`相同, 则说明前四个数字就是符合题意的一个子数组, 再来看第二个例子:

`nums`: [-2, -1, 2, 1], `k` = 1

`sums`: [-2, -3, -1, 0]

我们发现累积和中没有数字等于`k`, 但是我们知道这个例子的答案是[-1, 2], 那么我们看累积和数组的第一和第三个数字, 我们是否能看出一些规律呢, 没错, 第三个数字-1减去`k`, 得到第一个数字, 这就是规律, 这也是累积和求区间和的方法, 但是由于累计和数组中可能会有重复数字, 而哈希表的关键字不能相同, 比如下面这个例子:

`nums`: [1, 0, -1], `k` = -1

`sums`: [1, 1, 0]

我们发现累积和数组的第一个和第二个数字都为1, 那么如何建立映射呢, 我想的是用一个一维数组将其都存起来, 然后比较的话就比较数组中的第一个数字, 当我们建立完哈希表后, 开始遍历这个哈希表, 当累积和跟`k`相同时, 我们更新`res`, 不相同的话我们检测当前值减去`k`得到的值在哈希表中存不存在, 如果存在就更新结果, 参见代码如下:

解法1:

```

1 class Solution {
2 public:
3     int maxSubArrayLen(vector<int>& nums, int k) {
4         if (nums.empty()) return 0;
5         int res = 0;
6         unordered_map<int, vector<int>> m;
7         m[nums[0]].push_back(0);
8         vector<int> sum = nums;
9         for (int i = 1; i < nums.size(); ++i) {
10            sum[i] += sum[i - 1];
11            m[sum[i]].push_back(i);
12        }
13        for (auto it : m) {
14            if (it.first == k) res = max(res, it.second.back() + 1);
15            else if (m.find(it.first - k) != m.end()) {
16                res = max(res, it.second.back() - m[it.first - k][0]);
17            }
18        }
19        return res;
20    }
21 };

```

然而当我上网看大神们的解法时，才发现我图样图森破，根本不需要我写的那么复杂，我们不需要另外创建一个累积和的数组，而是直接用一个变量sum边累加边处理，而且我们哈希表也完全不用建立和一维数组的映射，只要保存第一个出现该累积和的位置，后面再出现直接跳过，这样算下来就是最长的子数组，对于想出这解法的人，博主只想说，阁下何不随风起，扶摇直上九万里~参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int maxSubArrayLen(vector<int>& nums, int k) {
4         int sum = 0, res = 0;
5         unordered_map<int, int> m;
6         for (int i = 0; i < nums.size(); ++i) {
7             sum += nums[i];
8             if (sum == k) res = i + 1;
9             else if (m.count(sum - k)) res = max(res, i - m[sum - k]);
10            if (!m.count(sum)) m[sum] = i;
11        }
12        return res;
13    }
14 };

```

326. 判断3的次方数

Given an integer, write a function to determine if it is a power of three.

Follow up:

Could you do it without using any loop / recursion?

这道题让我们判断一个数是不是3的次方数，在LeetCode中，有一道类似的题目Power of Two，那道题有个非常简单的方法，由于2的次方数实在太有特点，最高位为1，其他位均为0，所以特别容易，而3的次方数没有显著的特点，最直接的方法就是不停地除以3，看最后的余数是否为1，要注意考虑输入是负数和0的情况，参见代码如下：

解法1:

```
1 class Solution {
2 public:
3     bool isPowerOfThree(int n) {
4         while (n && n % 3 == 0) {
5             n /= 3;
6         }
7         return n == 1;
8     }
9 };
```

CPP

题目中的Follow up让我们不用循环，那么有一个投机取巧的方法，由于输入是int，正数范围是0-2³¹，在此范围内允许的最大的3的次方数为3¹⁹=1162261467，那么我们只要看这个数能否被n整除即可，参见代码如下：

解法2:

```
1 class Solution {
2 public:
3     bool isPowerOfThree(int n) {
4         return (n > 0 && 1162261467 % n == 0);
5     }
6 };
```

CPP

最后还有一种巧妙的方法，利用对数的换底公式来做，高中学过的换底公式为 $\log_a b = \log_c b / \log_c a$ ，那么如果n是3的倍数，则 $\log_3 n$ 一定是整数，我们利用换底公式可以写为 $\log_3 n = \log_{10} n / \log_{10} 3$ ，注意这里一定要用10为底数，不能用自然数或者2为底数，否则当n=243时会出错，原因请看这个帖子。现在问题就变成了判断 $\log_{10} n / \log_{10} 3$ 是否为整数，在C++中判断数字a是否为整数，我们可以用 $a - \text{int}(a) == 0$ 来判断，参见代码如下：

解法3:

```
1 class Solution {
2 public:
3     bool isPowerOfThree(int n) {
4         return (n > 0 && int(log10(n) / log10(3)) - log10(n) / log10(3) == 0);
5     }
6 };
```

CPP

327. 区间和计数

Given an integer array `nums`, return the number of range sums that lie in `[lower, upper]` inclusive.

Range sum $S(i, j)$ is defined as the sum of the elements in `nums` between indices i and j ($i \leq j$), inclusive.

Note:

A naive algorithm of $O(n^2)$ is trivial. You MUST do better than that.

Example:

Given `nums` = [-2, 5, -1], `lower` = -2, `upper` = 2,

Return 3.

The three ranges are : [0, 0], [2, 2], [0, 2] and their respective sums are: -2, -1, 2.

这道题给了我们一个数组，又给了我们一个下限和一个上限，让我们求有多少个不同的区间使得每个区间的和在给定的上下限之间。这道题的难度系数给的是Hard，的确是一道难度不小的题，题目中也说了Brute Force的方法太Naive了，那么我们只能另想方法了。To be honest，这题完全超出了我的能力范围，所以我也没挣扎了，直接上网搜大神们的解法啦。首先根据前面的那几道类似题Range Sum Query - Mutable 区域和检索 - 可变，Range Sum Query 2D - Immutable 二维区域和检索和Range Sum Query - Immutable 区域和检索 - 不可变的解法可知类似的区间和的问题一定是要计算累积和sum的，其中 $sum[i] = nums[0] + nums[1] + \dots + nums[i]$ ，对于某个i来说，只有那些满足 $lower \leq sum[i] - sum[j] \leq upper$ 的j能形成一个区间[j, i] 满足题意，那么我们的目标就是来找到有多少个这样的j ($0 \leq j < i$) 满足 $sum[i] - upper \leq sum[j] \leq sum[i] - lower$ ，我们可以用C++中由红黑树实现的multiset数据结构可以对其中数据排序，然后用upperbound和lowerbound来找临界值。
lower_bound是找数组中第一个不小于给定值的数(包括等于情况)，而upper_bound是找数组中第一个大于给定值的数，那么两者相减，就是j的个数，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int countRangeSum(vector<int>& nums, int lower, int upper) {
4         int res = 0;
5         long long sum = 0;
6         multiset<long long> sums;
7         sums.insert(0);
8         for (int i = 0; i < nums.size(); ++i) {
9             sum += nums[i];
10            res += distance(sums.lower_bound(sum - upper), sums.upper_bound(sum - lower));
11            sums.insert(sum);
12        }
13        return res;
14    }
15 };

```

CPP

我们再来看一种方法，这种方法的思路和前一种一样，只是没有STL的multiset和lower_bound和upper_bound函数，而是使用了Merge Sort来解，在混合的过程中，我们已经给左半边[start, mid) 和右半边 [mid, end) 排序了。当我们遍历左半边，对于每个i，我们需要在右半边找出k和j，使其满足：

j是第一个满足 $sums[j] - sums[i] > upper$ 的下标

k是第一个满足 $sums[k] - sums[i] \geq lower$ 的下标

那么在[lower, upper]之间的区间的个数是j - k，同时我们也需要另一个下标t，用来拷贝所有满足 $sums[t] < sums[i]$ 到一个寄存器Cache中以完成混合排序的过程。(注意这里sums可能会整型溢出，我们使用长整型long long代替)。

解法2：

```

1 class Solution {
2 public:
3     int countRangeSum(vector<int>& nums, int lower, int upper) {
4         vector<long> sums(nums.size() + 1, 0);
5         for (int i = 0; i < nums.size(); ++i) {
6             sums[i + 1] = sums[i] + nums[i];
7         }
8         return countAndMergeSort(sums, 0, sums.size(), lower, upper);
9     }
10    int countAndMergeSort(vector<long> &sums, int start, int end, int lower, int upper) {
11        if (end - start <= 1) return 0;
12        int mid = start + (end - start) / 2;
13        int cnt = countAndMergeSort(sums, start, mid, lower, upper) +
14        countAndMergeSort(sums, mid, end, lower, upper);
15        int j = mid, k = mid, t = mid;
16        vector<int> cache(end - start, 0);
17        for (int i = start, r = 0; i < mid; ++i, ++r) {
18            while (k < end && sums[k] - sums[i] < lower) ++k;
19            while (j < end && sums[j] - sums[i] <= upper) ++j;
20            while (t < end && sums[t] < sums[i]) cache[r++] = sums[t++];
21            cache[r] = sums[i];
22            cnt += j - k;
23        }
24        copy(cache.begin(), cache.begin() + t - start, sums.begin() + start);
25        return cnt;
26    }
};

```

328. 奇偶链表

Given a singly linked list, group all odd nodes together followed by the even nodes. Please note here we are talking about the node number and not the value in the nodes.

You should try to do it in place. The program should run in O(1) space complexity and O(nodes) time complexity.

Example:

Given 1->2->3->4->5->NULL,
return 1->3->5->2->4->NULL.

这道题给了我们一个链表，让我们分开奇偶节点，所有奇节点在前，偶节点在后。我们可以使用两个指针来做，pre指向奇节点，cur指向偶节点，然后把偶节点cur后面的那个奇节点提前到pre的后面，然后pre和cur各自前进一步，此时cur又指向偶节点，pre指向当前奇节点的末尾，以此类推直至把所有的偶节点都提前了即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     ListNode* oddEvenList(ListNode* head) {
4         if (!head || !head->next) return head;
5         ListNode *pre = head, *cur = head->next;
6         while (cur && cur->next) {
7             ListNode *tmp = pre->next;
8             pre->next = cur->next;
9             cur->next = cur->next->next;
10            pre->next->next = tmp;
11            cur = cur->next;
12            pre = pre->next;
13        }
14        return head;
15    }
16 };

```

还有一种解法，用两个奇偶指针分别指向奇偶节点的起始位置，另外需要一个单独的指针even_head来保存偶节点的起点位置，然后把奇节点的指向偶节点的下一个(一定是奇节点)，此奇节点后移一步，再把偶节点指向下一个奇节点的下一个(一定是偶节点)，此偶节点后移一步，以此类推直至末尾，此时把分开的偶节点的链表连在奇节点的链表后即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     ListNode* oddEvenList(ListNode* head) {
4         if (!head || !head->next) return head;
5         ListNode *odd = head, *even = head->next, *even_head = even;
6         while (even && even->next) {
7             odd = odd->next = even->next;
8             even = even->next = odd->next;
9         }
10        odd->next = even_head;
11        return head;
12    }
13 };

```

329. 矩阵中的最长递增路径

Given an integer matrix, find the length of the longest increasing path.

From each cell, you can either move to four directions: left, right, up or down. You may NOT move diagonally or move outside of the boundary (i.e. wrap-around is not allowed).

Example 1:

```

nums = [
[9,9,4],
[6,6,8],
[2,1,1]
]
Return 4
The longest increasing path is [1, 2, 6, 9].

```

这道题给我们一个二维数组，让我们求矩阵中最长的递增路径，规定我们只能上下左右行走，不能走斜线或者是超过了边界。那么这道题的解法要用递归和DP来解，用DP的原因是为了提高效率，避免重复运算。我们需要维护一个二维动态数组dp，其中dp[i][j]表示数组中以(i,j)为起点的最长递增路径的长度，初始将dp数组都赋为0，当我们用递归调用时，遇到某个位置(x,y)，如果dp[x][y]不为0的话，我们直接返回dp[x][y]即可，不需要重复计算。我们需要以数组中每个位置都为起点调用递归来做，比较找出最大值。在以一个位置为起点用DFS搜索时，对其四个相邻位置进行判断，如果相邻位置的值大于上一个位置，则对相邻位置继续调用递归，并更新一个最大值，搜索完成后返回即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<vector<int>> dirs = {{0, -1}, {-1, 0}, {0, 1}, {1, 0}};
4     int longestIncreasingPath(vector<vector<int>>& matrix) {
5         if (matrix.empty() || matrix[0].empty()) return 0;
6         int res = 1, m = matrix.size(), n = matrix[0].size();
7         vector<vector<int>> dp(m, vector<int>(n, 0));
8         for (int i = 0; i < m; ++i) {
9             for (int j = 0; j < n; ++j) {
10                 res = max(res, dfs(matrix, dp, i, j));
11             }
12         }
13         return res;
14     }
15     int dfs(vector<vector<int>> &matrix, vector<vector<int>> &dp, int i, int j) {
16         if (dp[i][j]) return dp[i][j];
17         int mx = 1, m = matrix.size(), n = matrix[0].size();
18         for (auto a : dirs) {
19             int x = i + a[0], y = j + a[1];
20             if (x < 0 || x >= m || y < 0 || a[0] == a[1] || matrix[x][y] <= matrix[i][j])
21             continue;
22             int len = 1 + dfs(matrix, dp, x, y);
23             mx = max(mx, len);
24         }
25         dp[i][j] = mx;
26         return mx;
27     }
28 };

```

下面再来看一种BFS的解法，需要用queue来辅助遍历，我们还是需要dp数组来减少重复运算。遍历数组中的每个数字，跟上面的解法一样，把每个遍历到的点都当作BFS遍历的起始点，需要优化的是，如果当前点的dp值大于0了，说明当前点已经计算过了，我们直接跳过。否则就新建一个queue，然后把当前点的坐标加进去，再用一个变量cnt，初始化为1，表示当前点为起点的递增长度，然后进入while循环，然后cnt自增1，这里先自增1没有关系，因为只有当周围有合法的点时候才会用cnt来更新。由于当前结点周围四个相邻点距当前点距离都一样，所以采用类似二叉树层序遍历的方式，先出当前queue的长度，然后遍历跟长度相同的次数，取出queue中的首元素，对周围四个点进行遍历，计算出相邻点的坐标后，要进行合法性检查，横纵坐标不能越界，且相邻点的值要大于当前点的值，并且相邻点点dp值要小于cnt，才有更新的必要。用cnt来更新dp[x][y]，并用cnt来更新结果res，然后把相邻点排入queue中继续循环即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int longestIncreasingPath(vector<vector<int>>& matrix) {
4         if (matrix.empty() || matrix[0].empty()) return 0;
5         int m = matrix.size(), n = matrix[0].size(), res = 1;
6         vector<vector<int>> dirs{{0,-1},{-1,0},{0,1},{1,0}};
7         vector<vector<int>> dp(m, vector<int>(n, 0));
8         for (int i = 0; i < m; ++i) {
9             for (int j = 0; j < n; ++j) {
10                 if (dp[i][j] > 0) continue;
11                 queue<pair<int, int>> q{{i, j}};
12                 int cnt = 1;
13                 while (!q.empty()) {
14                     ++cnt;
15                     int len = q.size();
16                     for (int k = 0; k < len; ++k) {
17                         auto t = q.front(); q.pop();
18                         for (auto dir : dirs) {
19                             int x = t.first + dir[0], y = t.second + dir[1];
20                             if (x < 0 || x >= m || y < 0 || y >= n || matrix[x][y] <=
21 matrix[t.first][t.second] || cnt <= dp[x][y]) continue;
22                             dp[x][y] = cnt;
23                             res = max(res, cnt);
24                             q.push({x, y});
25                         }
26                     }
27                 }
28             }
29         }
30         return res;
31     }
32 };

```

330. 补丁数组

Given a sorted positive integer array nums and an integer n, add/patch elements to the array such that any number in range [1, n] inclusive can be formed by the sum of some elements in the array. Return the minimum number of patches required.

Example 1:

nums = [1, 3], n = 6

Return 1.

Combinations of nums are [1], [3], [1,3], which form possible sums of: 1, 3, 4.

Now if we add/patch 2 to nums, the combinations are: [1], [2], [3], [1,3], [2,3], [1,2,3].

Possible sums are 1, 2, 3, 4, 5, 6, which now covers the range [1, 6].

So we only need 1 patch.

这道题给我们一个有序的正数数组`nums`, 又给了我们一个正整数`n`, 问我们最少需要给`nums`加几个数字, 使其能组成 $[1, n]$ 之间的所有数字, 注意数组中的元素不能重复使用, 否则的话只有要有1, 就能组成所有的数字了。这道题我又不会了, 上网看到了史蒂芬大神的解法, 膜拜啊, 这里就全部按他的解法来讲吧。我们定义一个变量`miss`, 用来表示 $[0, n]$ 之间最小的不能表示的值, 那么初始化为1, 为啥不为0呢, 因为 $n=0$ 没啥意义, 直接返回0了。那么此时我们能表示的范围是 $[0, miss)$, 表示此时我们能表示0到`miss-1`的数, 如果此时的`num <= miss`, 那么我们可以把我们能表示数的范围扩大到 $[0, miss+num)$, 如果`num>miss`, 那么此时我们需要添加一个数, 为了能最大限度的增加表示数范围, 我们加上`miss`它本身, 以此类推直至遍历完整个数组, 我们可以得到结果。下面我们就来举个例子说明:

给定`nums = [1, 2, 4, 11, 30]`, `n = 50`, 我们需要让 $[0, 50]$ 之间所有的数字都能被`nums`中的数字之和表示出来。

首先使用1, 2, 4可能表示出0到7之间的所有数, 表示范围为 $[0, 8)$, 但我们不能表示8, 因为下一个数字11太大了, 所以我们要在数组里加上一个8, 此时能表示的范围是 $[0, 16)$, 那么我们需要插入16吗, 答案是不需要, 因为我们数组有1和4, 可以组成5, 而下一个数字11, 加一起能组成16, 所以有了数组中的11, 我们此时能表示的范围扩大到 $[0, 27)$, 但我们没法表示27, 因为30太大了, 所以此时我们给数组中加入一个27, 那么现在能表示的范围是 $[0, 54)$, 已经满足要求了, 我们总共添加了两个数8和27, 所以返回2即可。

解法1:

```
1 class Solution {
2 public:
3     int minPatches(vector<int>& nums, int n) {
4         long miss = 1, res = 0, i = 0;
5         while (miss <= n) {
6             if (i < nums.size() && nums[i] <= miss) {
7                 miss += nums[i++];
8             } else {
9                 miss += miss;
10                ++res;
11            }
12        }
13        return res;
14    }
15};
```

CPP

下面这种方法跟上面那种方法原理都一样, 稍有不同之处在于真正的patch了`nums`数组, 把需要插入的数字真正的加入了数组中, 那么最后用新数组的长度减去原始长度就知道我们加入了几个数字了。

解法2:

```
1 class Solution {
2 public:
3     int minPatches(vector<int>& nums, int n) {
4         long miss = 1, k = nums.size(), i = 0;
5         while (miss <= n) {
6             if (i >= nums.size() || nums[i] > miss) {
7                 nums.insert(nums.begin() + i, miss);
8             }
9             miss += nums[i++];
10        }
11        return nums.size() - k;
12    }
13};
```

CPP

331. 验证二叉树的先序序列化

One way to serialize a binary tree is to use pre-order traversal. When we encounter a non-null node, we record the node's value. If it is a null node, we record using a sentinel value such as #.

```

 _9_
 / \
3   2
/ \  / \
4  1  #  6
/ \ / \  / \
# # # #  # #

```

For example, the above binary tree can be serialized to the string "9,3,4,#,#,1,#,#,2,#,6,#,#", where # represents a null node.

Given a string of comma separated values, verify whether it is a correct preorder traversal serialization of a binary tree. Find an algorithm without reconstructing the tree.

Each comma separated value in the string must be either an integer or a character '#' representing null pointer.

You may assume that the input format is always valid, for example it could never contain two consecutive commas such as "1,,3".

Example 1:

"9,3,4,#,#,1,#,#,2,#,6,#,#"

Return true

这道题给了我们一个类似序列化二叉树后的字符串，关于二叉树的序列化和去序列化可以参见我之前的博客Serialize and Deserialize Binary Tree，这道题让我们判断给定的字符串是不是一个正确的序列化的二叉树的字符串。那么根据之前那边博客的解法，我们还是要用istringstream来操作字符串，C++里面没有像Java那样有字符串的split函数，可以直接分隔任意字符串，我们只能使用getline这个函数，来将字符串流的内容都存到一个vector数组中。我们通过举一些正确的例子，比如"9,3,4,#,#,1,#,#,2,#,6,#,#" 或者"9,3,4,#,#,1,#,#,2,#,6,#,#"等等，可以观察出如下两个规律：

1. 数字的个数总是比#号少一个

2. 最后一个一定是#号

那么我们加入先不考虑最后一个#号，那么此时数字和#号的个数应该相同，如果我们初始化一个为0的计数器，遇到数字，计数器加1，遇到#号，计数器减1，那么到最后计数器应该还是0。下面我们再来看两个返回False的例子， "#,7,6,9,#,#,#" 和 "7,2,#,2,#,#,6,#"，那么通过这两个反例我们可以看出，如果根节点为空的话，后面不能再有节点，而且不能有三个连续的#号出现。所以我们再加减计数器的时候，如果遇到#号，且此时计数器已经为0了，再减就成负数了，就直接返回False了，因为正确的序列里，任何一个位置i，在[0, i]范围内的#号数都不大于数字的个数的。当循环完成后，我们检测计数器是否为0的同时还要看看最后一个字符是不是#号。参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool isValidSerialization(string preorder) {
4         istringstream in(preorder);
5         vector<string> v;
6         string t = "";
7         int cnt = 0;
8         while (getline(in, t, ',')) v.push_back(t);
9         for (int i = 0; i < v.size() - 1; ++i) {
10             if (v[i] == "#") {
11                 if (cnt == 0) return false;
12                 --cnt;
13             } else ++cnt;
14         }
15         return cnt == 0 && v.back() == "#";
16     }
17 };

```

下面这种解法由网友edyyy提供，不需要建立解法一中的额外数组，而是边解析边判断，遇到不合题意的情况直接返回false，而不用全部解析完再来验证是否合法，提高了运算的效率。我们用一个变量degree表示能容忍""的个数，degree初始化为1。再用一个布尔型变量degree_is_zero来记录degree此时是否为0的状态，这样的设计很巧妙，可以cover到""开头，但后面还跟有数字的情况，比如"1,2"这种情况，当检测到""时，degree自减1，此时若degree为0了，degree_is_zero赋值为true，那么如果后面还跟有其他东西的话，在下次循环开始前，先判断degree_is_zero，如果为true的话，直接返回false。而当首字符为数字的话，degree自增1，那么此时degree就成了2，表示后面可以再容忍两个""。当循环退出的时候，此时判断degree是否为0，因为我们要补齐""的个数，少了也是不对的，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool isValidSerialization(string preorder) {
4         istringstream in(preorder);
5         string t = "";
6         int degree = 1;
7         bool degree_is_zero = false;
8         while (getline(in, t, ',')) {
9             if (degree_is_zero) return false;
10            if (t == "#") {
11                if (--degree == 0) degree_is_zero = true;
12            } else ++degree;
13        }
14        return degree == 0;
15    }
16 };

```

下面这种解法就更加巧妙了，连字符串解析都不需要了，用一个变量capacity来记录能容忍""的个数，跟上面解法中的degree一个作用，然后我们给preorder末尾加一个逗号，这样可以处理末尾的""。我们遍历preorder字符串，如果遇到了非逗号的字符，直接跳过，否则的话capacity自减1，如果此时capacity小于0了，直接返回true。此时再判断逗号前面的字符是否为""，如果不是的话，capacity自增2。这种设计非常巧妙，如果逗号前面是""，我们capacity自减1没问题，因为容忍了一个""；如果前面是数字，那么先自减的1，可以看作是初始化的1被减了，然后再自增2，因为每多一个数字，可以多容忍两个""，最后还是要判断capacity是否为0，跟上面的解法一样，我们要补齐#"的个数，少了也是不对的，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool isValidSerialization(string preorder) {
4         int capacity = 1;
5         preorder += ",";
6         for (int i = 0; i < preorder.size(); ++i) {
7             if (preorder[i] != ',') continue;
8             if (--capacity < 0) return false;
9             if (preorder[i - 1] != '#') capacity += 2;
10        }
11        return capacity == 0;
12    }
13 };

```

332. 重建行程单

Given a list of airline tickets represented by pairs of departure and arrival airports [from, to], reconstruct the itinerary in order. All of the tickets belong to a man who departs from JFK. Thus, the itinerary must begin with JFK.

Note:

If there are multiple valid itineraries, you should return the itinerary that has the smallest lexical order when read as a single string. For example, the itinerary ["JFK", "LGA"] has a smaller lexical order than ["JFK", "LGB"].

All airports are represented by three capital letters (IATA code).

You may assume all tickets may form at least one valid itinerary.

Example 1:

```

tickets = [["MUC", "LHR"], ["JFK", "MUC"], ["SFO", "SJC"], ["LHR", "SFO"]]
Return ["JFK", "MUC", "LHR", "SFO", "SJC"].

```

这道题给我们一堆飞机票，让我们建立一个行程单，如果有多种方法，取其中字母顺序小的那种方法。这道题的本质是有向图的遍历问题，那么LeetCode关于有向图的题只有两道Course Schedule和Course Schedule II，而那两道是关于有向图的顶点的遍历的，而本题是关于有向图的边的遍历。每张机票都是有向图的一条边，我们需要找出一条经过所有边的路径，那么DFS不是我们的不二选择。先来看递归的结果，我们首先把图建立起来，通过邻接链表来建立。由于题目要求解法按字母顺序小的，那么我们考虑用multiset，可以自动排序。等我们图建立好了以后，从节点JFK开始遍历，只要当前节点映射的multiset里有节点，我们取出这个节点，将其在multiset里删掉，然后继续递归遍历这个节点，由于题目中限定了一定会有解，那么等图中所有的multiset中都没有节点的时候，我们把当前节点存入结果中，然后再一层层回溯回去，将当前节点都存入结果，那么最后我们结果中存的顺序和我们需要的相反的，我们最后再翻转一下即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<string> findItinerary(vector<pair<string, string>> tickets) {
4         vector<string> res;
5         unordered_map<string, multiset<string>> m;
6         for (auto a : tickets) {
7             m[a.first].insert(a.second);
8         }
9         dfs(m, "JFK", res);
10        return vector<string> (res.rbegin(), res.rend());
11    }
12    void dfs(unordered_map<string, multiset<string>>& m, string s, vector<string>& res) {
13        while (m[s].size()) {
14            string t = *m[s].begin();
15            m[s].erase(m[s].begin());
16            dfs(m, t, res);
17        }
18        res.push_back(s);
19    }
20 };

```

下面我们来看迭代的解法，需要借助栈来实现，来实现回溯功能。比如对下面这个例子：

```
tickets = [["JFK", "KUL"], ["JFK", "NRT"], ["MRT", "JFK"]]
```

那么建立的图如下：

```
JFK -> KUL, NRT
```

```
NRT -> JFK
```

由于`multiset`是按顺序存的，所有KUL会在NRT之前，那么我们起始从JFK开始遍历，先到KUL，但是KUL没有下家了，这时候图中的边并没有遍历完，此时我们需要将KUL存入栈中，然后继续往下遍历，最后再把栈里的节点存回结果即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<string> findItinerary(vector<pair<string, string>> tickets) {
4         vector<string> res;
5         stack<string> st{{"JFK"}};
6         unordered_map<string, multiset<string>> m;
7         for (auto t : tickets) {
8             m[t.first].insert(t.second);
9         }
10        while (!st.empty()) {
11            string t = st.top();
12            if (m[t].empty()) {
13                res.insert(res.begin(), t);
14                st.pop();
15            } else {
16                st.push(*m[t].begin());
17                m[t].erase(m[t].begin());
18            }
19        }
20        return res;
21    }
22 };

```

333. 最大的二分搜索子树

Given a binary tree, find the largest subtree which is a Binary Search Tree (BST), where largest means subtree with largest number of nodes in it.

Note:

A subtree must include all of its descendants.

Here's an example:

```

10
 / \
5  15
 / \ \
1  8  7

```

The Largest BST Subtree in this case is the highlighted one.

The return value is the subtree's size, which is 3.

Hint:

You can recursively use algorithm similar to 98. Validate Binary Search Tree at each node of the tree, which will result in $O(n\log n)$ time complexity.

Follow up:

Can you figure out ways to solve it with $O(n)$ time complexity?

这道题让我们求一棵二分树的最大二分搜索子树，所谓二分搜索树就是满足左<根<右的二分树，我们需要返回这个二分搜索子树的节点个数。题目中给的提示说我们可以用之前那道Validate Binary Search Tree的方法来做，时间复杂度为 $O(n^2)$ ，这种方法是把每个节点都当做根节点，来验证其是否是二叉搜索数，并记录节点的个数，若是二叉搜索树，就更新最终结果，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int largestBSTSubtree(TreeNode* root) {
4         int res = 0;
5         dfs(root, res);
6         return res;
7     }
8     void dfs(TreeNode *root, int &res) {
9         if (!root) return;
10        int d = countBFS(root, INT_MIN, INT_MAX);
11        if (d != -1) {
12            res = max(res, d);
13            return;
14        }
15        dfs(root->left, res);
16        dfs(root->right, res);
17    }
18    int countBFS(TreeNode *root, int mn, int mx) {
19        if (!root) return 0;
20        if (root->val <= mn || root->val >= mx) return -1;
21        int left = countBFS(root->left, mn, root->val);
22        if (left == -1) return -1;
23        int right = countBFS(root->right, root->val, mx);
24        if (right == -1) return -1;
25        return left + right + 1;
26    }
27 };

```

下面我们来看一种更简洁的写法，对于每一个节点，都来验证其是否是BST，如果是的话，我们就统计节点的个数即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int largestBSTSubtree(TreeNode* root) {
4         if (!root) return 0;
5         if (isValid(root, INT_MIN, INT_MAX)) return count(root);
6         return max(largestBSTSubtree(root->left), largestBSTSubtree(root->right));
7     }
8     bool isValid(TreeNode* root, int mn, int mx) {
9         if (!root) return true;
10        if (root->val <= mn || root->val >= mx) return false;
11        return isValid(root->left, mn, root->val) && isValid(root->right, root->val, mx);
12    }
13    int count(TreeNode* root) {
14        if (!root) return 0;
15        return count(root->left) + count(root->right) + 1;
16    }
17 };

```

题目中的Follow up让我们用O(n)的时间复杂度来解决问题，我们还是采用DFS的思想来解题，由于时间复杂度的限制，只允许我们遍历一次整个二叉树，由于满足题目要求的二叉搜索子树必定是有叶节点的，所以我们的思路就是先递归到最左子节点，然后逐层往上递归，对于每一个节点，我们都记录当前最大的BST的节点数，当做为左子树的最大值，和做为右子树的最小值，当

每次遇到左子节点不存在或者当前节点值大于左子树的最大值，且右子树不存在或者当前节点值小于右子树的最小值时，说明BST的节点数又增加了一个，我们更新结果及其参数，如果当前节点不是BST的节点，那么我们更新BST的节点数res为左右子节点的各自的BST的节点数的较大值，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int largestBSTSubtree(TreeNode* root) {
4         int res = 0, mn = INT_MIN, mx = INT_MAX;
5         bool d = isValidBST(root, mn, mx, res);
6         return res;
7     }
8     bool isValidBST(TreeNode *root, int &mn, int &mx, int &res) {
9         if (!root) return true;
10        int left_n = 0, right_n = 0, left_mn = INT_MIN;
11        int right_mn = INT_MIN, left_mx = INT_MAX, right_mx = INT_MAX;
12        bool left = isValidBST(root->left, left_mn, left_mx, left_n);
13        bool right = isValidBST(root->right, right_mn, right_mx, right_n);
14        if (left && right) {
15            if ((!root->left || root->val > left_mx) && (!root->right || root->val <
16 right_mn)) {
17                res = left_n + right_n + 1;
18                mn = root->left ? left_mn : root->val;
19                mx = root->right ? right_mx : root->val;
20                return true;
21            }
22        }
23        res = max(left_n, right_n);
24        return false;
25    }
};

```

334. 递增的三元子序列

Given an unsorted array return whether an increasing subsequence of length 3 exists or not in the array.

Formally the function should:

Return true if there exists i, j, k such that arr[i] < arr[j] < arr[k] given $0 \leq i < j < k \leq n-1$ else return false.

Your algorithm should run in $O(n)$ time complexity and $O(1)$ space complexity.

Examples:

Given [1, 2, 3, 4, 5],
return true.

这道题让我们求一个无序数组中是否有任意三个数字是递增关系的，我最先想的方法是用一个dp数组， $dp[i]$ 表示在*i*位置之前小于等于*nums[i]*的数字的个数(包括其本身)，我们初始化dp数组都为1，然后我们开始遍历原数组，对当前数字*nums[i]*，我们遍历其之前的所有数字，如果之前某个数字*nums[j]*小于*nums[i]*，那么我们更新 $dp[i] = \max(dp[i], dp[j] + 1)$ ，如果此时 $dp[i]$ 到3了，则返回true，若遍历完成，则返回false，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool increasingTriplet(vector<int>& nums) {
4         vector<int> dp(nums.size(), 1);
5         for (int i = 0; i < nums.size(); ++i) {
6             for (int j = 0; j < i; ++j) {
7                 if (nums[j] < nums[i]) {
8                     dp[i] = max(dp[i], dp[j] + 1);
9                     if (dp[i] >= 3) return true;
10                }
11            }
12        }
13        return false;
14    }
15 };

```

但是题目中要求我们O(n)的时间复杂度和O(1)的空间复杂度，上面的那种方法一条都没满足，所以白写了。我们下面来看满足题意的方法，这个思路是使用两个指针m1和m2，初始化为整型最大值，我们遍历数组，如果m1大于等于当前数字，则将当前数字赋给m1；如果m1小于当前数字且m2大于等于当前数字，那么将当前数字赋给m2，一旦m2被更新了，说明一定会有个数小于m2，那么我们就成功的组成了一个长度为2的递增子序列，所以我们一旦遍历到比m2还大的数，我们直接返回ture。如果我们遇到比m1小的数，还是要更新m1，有可能的话也要更新m2为更小的值，毕竟m2的值越小，能组成长度为3的递增序列的可能性越大，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool increasingTriplet(vector<int>& nums) {
4         int m1 = INT_MAX, m2 = INT_MAX;
5         for (auto a : nums) {
6             if (m1 >= a) m1 = a;
7             else if (m2 >= a) m2 = a;
8             else return true;
9         }
10        return false;
11    }
12 };

```

如果觉得上面的解法不容易想出来，那么如果能想出下面这种解法，估计面试官也会为你点赞。这种方法的虽然不满足常数空间的要求，但是作为对暴力搜索的优化，也是一种非常好的解题思路。这个解法的思路是建立两个数组，forward数组和backward数组，其中forward[i]表示[0, i]之间最小的数，backward[i]表示[i, n-1]之间最大的数，那么对于任意一个位置i，如果满足 forward[i] < nums[i] < backward[i]，则表示这个递增三元子序列存在，举个例子来看吧，比如：

nums: 8 3 5 1 6

foward: 8 3 3 1 1

backward: 8 6 6 6 6

我们发现数字5满足forward[i] < nums[i] < backward[i]，所以三元子序列存在。

解法3：

```

1 class Solution {
2 public:
3     bool increasingTriplet(vector<int>& nums) {
4         if (nums.size() < 3) return false;
5         int n = nums.size();
6         vector<int> f(n, nums[0]), b(n, nums.back());
7         for (int i = 1; i < n; ++i) {
8             f[i] = min(f[i - 1], nums[i]);
9         }
10        for (int i = n - 2; i >= 0; --i) {
11            b[i] = max(b[i + 1], nums[i]);
12        }
13        for (int i = 0; i < n; ++i) {
14            if (nums[i] > f[i] && nums[i] < b[i]) return true;
15        }
16        return false;
17    }
18 };

```

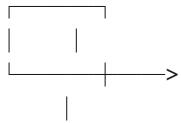
335. 自交

You are given an array x of n positive numbers. You start at point $(0,0)$ and moves $x[0]$ metres to the north, then $x[1]$ metres to the west, $x[2]$ metres to the south, $x[3]$ metres to the east and so on. In other words, after each move your direction changes counter-clockwise.

Write a one-pass algorithm with $O(1)$ extra space to determine, if your path crosses itself, or not.

Example 1:

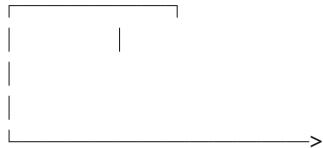
Given $x = [2, 1, 1, 2]$,



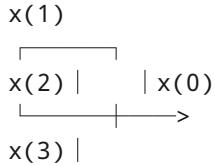
Return true (self crossing)

Example 2:

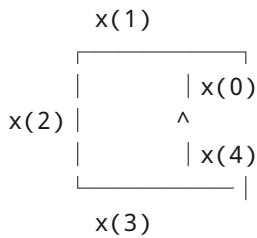
Given $x = [1, 2, 3, 4]$,



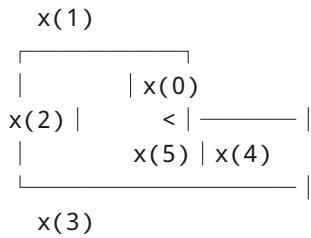
这道题给了我们一个一位数组，每个数字是个移动量，按照上左下右的顺序来前进每一个位移量，问我们会不会和之前的轨迹相交，而且限定了常量的空间复杂度，我立马想到了贪吃蛇游戏，但是这条蛇不会自动向前移动哈。言归正传，这题我不会，参考的网上大神们的解法，实际上相交的情况只有以下三种情况：



第一类是第四条边和第一条边相交的情况，需要满足的条件是第一条边大于等于第三条边，第四条边大于等于第二条边。同样适用于第五条边和第二条边相交，第六条边和第三条边相交等等，依次向后类推的情况...



第二类是第五条边和第一条边重合相交的情况，需要满足的条件是第二条边和第四条边相等，第五条边大于等于第三条边和第一条边的差值，同样适用于第六条边和第二条边重合相交的情况等等依次向后类推...



第三类是第六条边和第一条边相交的情况，需要满足的条件是第四条边大于等于第二条边，第三条边大于等于第五条边，第五条边大于等于第三条边和第一条边的差值，第六条边大于等于第四条边和第二条边的差值，同样适用于第七条边和第二条边相交的情况等等依次向后类推...

那么根据上面的分析，我们不难写出代码如下：

```

1 class Solution {
2 public:
3     bool isSelfCrossing(vector<int>& x) {
4         for (int i = 3; i < x.size(); ++i) {
5             if (x[i] >= x[i - 2] && x[i - 3] >= x[i - 1]) {
6                 return true;
7             }
8             if (i >= 4 && x[i-1] == x[i-3] && x[i] >= x[i-2] - x[i-4]) {
9                 return true;
10            }
11            if (i >= 5 && x[i-2] >= x[i-4] && x[i-3] >= x[i-1] && x[i-1] >= x[i-3] - x[i-5]
12 && x[i] >= x[i-2] - x[i-4]) {
13                 return true;
14            }
15        }
16        return false;
17    }
};
```

CPP

336. 回文对

Given a list of unique words. Find all pairs of distinct indices (i, j) in the given list, so that the concatenation of the two words, i.e. $\text{words}[i] + \text{words}[j]$ is a palindrome.

Example 1:

Given words = ["bat", "tab", "cat"]

Return [[0, 1], [1, 0]]

The palindromes are ["battab", "tabbat"]

这道题给我们了许多单词，让我们找出回文对，就是两个单词拼起来是个回文字符串，我最开始尝试的是brute force的方法，每两个单词都拼接起来然后判断是否是回文字符串，但是通过不了OJ，会超时，可能这也是这道题标为Hard的原因之一吧，那么我们只能找别的方法来做，通过学习大神们的解法，发现如下两种方法比较好，其实两种方法的核心思想都一样，写法略有不同而已，那么我们先来看第一种方法吧，要用到哈希表来建立每个单词和其位置的映射，然后需要一个set来保存出现过的单词的长度，算法的思想是，遍历单词集，对于遍历到的单词，我们对其翻转一下，然后在哈希表查找翻转后的字符串是否存在，注意不能和原字符串的坐标位置相同，因为有可能一个单词翻转后和原单词相等，现在我们只是处理了bat和tab的情况，还存在abcd和cba, dcb和abcd这些情况需要考虑，这就是我们为啥需要用set，由于set是自动排序的，我们可以找到当前单词长度在set中的iterator，然后从开头开始遍历set，遍历比当前单词小的长度，比如abcd翻转后为dcba，我们发现set中有长度为3的单词，然后我们dd是否为回文串，若是，再看cba是否存在于哈希表，若存在，则说明dcba和dcba是回文对，存入结果中，对于dcb和aabcd这类的情况也是同样处理，我们要在set里找的字符串要在遍历到的字符串的左边和右边分别尝试，看是否是回文对，这样遍历完单词集，就能得到所有的回文对，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<vector<int>> palindromePairs(vector<string>& words) {
4         vector<vector<int>> res;
5         unordered_map<string, int> m;
6         set<int> s;
7         for (int i = 0; i < words.size(); ++i) {
8             m[words[i]] = i;
9             s.insert(words[i].size());
10        }
11        for (int i = 0; i < words.size(); ++i) {
12            string t = words[i];
13            int len = t.size();
14            reverse(t.begin(), t.end());
15            if (m.count(t) && m[t] != i) {
16                res.push_back({i, m[t]});
17            }
18            auto a = s.find(len);
19            for (auto it = s.begin(); it != a; ++it) {
20                int d = *it;
21                if (isValid(t, 0, len - d - 1) && m.count(t.substr(len - d))) {
22                    res.push_back({i, m[t.substr(len - d)]});
23                }
24                if (isValid(t, d, len - 1) && m.count(t.substr(0, d))) {
25                    res.push_back({m[t.substr(0, d)], i});
26                }
27            }
28        }
29        return res;
30    }
31    bool isValid(string t, int left, int right) {
32        while (left < right) {
33            if (t[left++] != t[right--]) return false;
34        }
35        return true;
36    }
37 };

```

下面这种方法没有用到set，但实际上循环的次数要比上面多，因为这种方法对于遍历到的字符串，要验证其所有可能的子串，看其是否在哈希表里存在，并且能否组成回文对，anyway，既然能通过OJ，说明还是比brute force要快的，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<vector<int>> palindromePairs(vector<string>& words) {
4         vector<vector<int>> res;
5         unordered_map<string, int> m;
6         for (int i = 0; i < words.size(); ++i) m[words[i]] = i;
7         for (int i = 0; i < words.size(); ++i) {
8             int l = 0, r = 0;
9             while (l <= r) {
10                 string t = words[i].substr(l, r - l);
11                 reverse(t.begin(), t.end());
12                 if (m.count(t) && i != m[t] && isValid(words[i].substr(l == 0 ? r : 0, l ==
13 0 ? words[i].size() - r: 1))) {
14                     if (l == 0) res.push_back({i, m[t]});
15                     else res.push_back({m[t], i});
16                 }
17                 if (r < words[i].size()) ++r;
18                 else ++l;
19             }
20         }
21         return res;
22     }
23     bool isValid(string t) {
24         for (int i = 0; i < t.size() / 2; ++i) {
25             if (t[i] != t[t.size() - 1 - i]) return false;
26         }
27         return true;
28     }
29 };

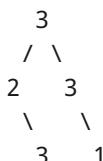
```

337. 打家劫舍之三

The thief has found himself a new place for his thievery again. There is only one entrance to this area, called the "root." Besides the root, each house has one and only one parent house. After a tour, the smart thief realized that "all houses in this place forms a binary tree". It will automatically contact the police if two directly-linked houses were broken into on the same night.

Determine the maximum amount of money the thief can rob tonight without alerting the police.

Example 1:



Maximum amount of money the thief can rob = 3 + 3 + 1 = 7.

这道题是之前那两道House Robber II和House Robber的拓展，这个小偷又偷出新花样了，沿着二叉树开始偷，碉堡了，题目中给的例子看似好像是要每隔一个偷一次，但实际上不一定只隔一个，比如如下这个例子：

```

4
/
1
/
2
/
3

```

如果隔一个偷，那么是 $4+2=6$ ，其实最优解应为 $4+3=7$ ，隔了两个，所以说纯粹是怎么多怎么来，那么这种问题是很典型的递归问题，我们可以利用回溯法来做，因为当前的计算需要依赖之前的结果，那么我们对于某一个节点，如果其左子节点存在，我们通过递归调用函数，算出不包含左子节点返回的值，同理，如果右子节点存在，算出不包含右子节点返回的值，那么此节点的最大值可能有两种情况，一种是该节点值加上不包含左子节点和右子节点的返回值之和，另一种是左右子节点返回值之和不包含当期节点值，取两者的较大值返回即可，但是这种方法无法通过OJ，超时了，所以我们必须优化这种方法，这种方法重复计算了很多地方，比如要完成一个节点的计算，就得一直找左右子节点计算，我们可以把已经算过的节点用哈希表保存起来，以后递归调用的时候，现在哈希表里找，如果存在直接返回，如果不存在，等计算出来后，保存到哈希表中再返回，这样方便以后再调用，参见代码如下：

解法1：

```

1 | class Solution {
2 | public:
3 |     int rob(TreeNode* root) {
4 |         unordered_map<TreeNode*, int> m;
5 |         return dfs(root, m);
6 |     }
7 |     int dfs(TreeNode *root, unordered_map<TreeNode*, int> &m) {
8 |         if (!root) return 0;
9 |         if (m.count(root)) return m[root];
10 |         int val = 0;
11 |         if (root->left) {
12 |             val += dfs(root->left->left, m) + dfs(root->left->right, m);
13 |         }
14 |         if (root->right) {
15 |             val += dfs(root->right->left, m) + dfs(root->right->right, m);
16 |         }
17 |         val = max(val + root->val, dfs(root->left, m) + dfs(root->right, m));
18 |         m[root] = val;
19 |         return val;
20 |     }
21 | };

```

CPP

下面再来看一种方法，这种方法的递归函数返回一个大小为2的一维数组res，其中res[0]表示不包含当前节点值的最大值，res[1]表示包含当前值的最大值，那么我们在遍历某个节点时，首先对其左右子节点调用递归函数，分别得到包含与不包含左子节点值的最大值，和包含于不包含右子节点值的最大值，那么当前节点的res[0]就是左子节点两种情况的较大值加上右子节点两种情况的较大值，res[1]就是不包含左子节点值的最大值加上不包含右子节点值的最大值，和当前节点值之和，返回即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int rob(TreeNode* root) {
4         vector<int> res = dfs(root);
5         return max(res[0], res[1]);
6     }
7     vector<int> dfs(TreeNode *root) {
8         if (!root) return vector<int>(2, 0);
9         vector<int> left = dfs(root->left);
10        vector<int> right = dfs(root->right);
11        vector<int> res(2, 0);
12        res[0] = max(left[0], left[1]) + max(right[0], right[1]);
13        res[1] = left[0] + right[0] + root->val;
14        return res;
15    }
16 };
17

```

下面这种解法由网友edyyy提供，仔细看了一下，也非常的巧妙，思路和解法二有些类似。这里的helper函数返回当前结点为根结点的最大rob的钱数，里面的两个参数l和r表示分别从左子结点和右子结点开始rob，分别能获得的最大钱数。在递归函数里面，如果当前结点不存在，直接返回0。否则我们对左右子结点分别调用递归函数，得到l和r。另外还得到四个变量，ll和lr表示左子结点的左右子结点的最大rob钱数，rl和rr表示右子结点的最大rob钱数。那么我们最后返回的值其实是两部分的值比较，其中一部分的值是当前的结点值加上ll, lr, rl, 和rr这四个值，这不难理解，因为抢了当前的房屋，那么左右两个子结点就不能再抢了，但是再下一层的四个子结点都是可以抢的；另一部分是不抢当前房屋，而是抢其左右两个子结点，即l+r的值，返回两个部分的值中的较大值即可，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int rob(TreeNode* root) {
4         int l = 0, r = 0;
5         return helper(root, l, r);
6     }
7     int helper(TreeNode* node, int& l, int& r) {
8         if (!node) return 0;
9         int ll = 0, lr = 0, rl = 0, rr = 0;
10        l = helper(node->left, ll, lr);
11        r = helper(node->right, rl, rr);
12        return max(node->val + ll + lr + rl + rr, l + r);
13    }
14 };

```

338. 计数位

Given a non negative integer number num. For every numbers i in the range $0 \leq i \leq \text{num}$ calculate the number of 1's in their binary representation and return them as an array.

Example:

For num = 5 you should return [0,1,1,2,1,2].

Follow up:

It is very easy to come up with a solution with run time $O(n * \text{sizeof(integer)})$. But can you do it in linear time $O(n)$ /possibly in a single pass?

Space complexity should be $O(n)$.

Can you do it like a boss? Do it without using any builtin function like `__builtin_popcount` in c++ or in any other language.

Hint:

You should make use of what you have produced already.

Divide the numbers in ranges like [2-3], [4-7], [8-15] and so on. And try to generate new range from previous.

Or does the odd/even status of the number help you in calculating the number of 1s?

这道题给我们一个整数n，然我们统计从0到n每个数的二进制写法的1的个数，存入一个一维数组中返回，题目中明确表示不希望我们一个数字一个数字，一位一位的傻算，而是希望我们找出规律，而且题目中也提示了我们注意[2-3], [4-7], [8-15]这些区间的规律，那么我们写出0到15的数的二进制和1的个数如下：

0	0000	0
1	0001	1
2	0010	1
3	0011	2
4	0100	1
5	0101	2
6	0110	2
7	0111	3
8	1000	1
9	1001	2
10	1010	2
11	1011	3
12	1100	2
13	1101	3
14	1110	3
15	1111	4

我最先看出的规律是这样的，除去前两个数字0个1，从2开始，2和3，是[21, 22)区间的，值为1和2。而4到7属于[22, 23)区间的，值为1,2,2,3，前半部分1和2和上一区间相同，2和3是上面的基础上每个数字加1。再看8到15，属于[23, 24)区间的，同样满足上述规律，所以可以写出代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<int> countBits(int num) {
4         if (num == 0) return {0};
5         vector<int> res{0, 1};
6         int k = 2, i = 2;
7         while (i <= num) {
8             for (i = pow(2, k - 1); i < pow(2, k); ++i) {
9                 if (i > num) break;
10                int t = (pow(2, k) - pow(2, k - 1)) / 2;
11                if (i < pow(2, k - 1) + t) res.push_back(res[i - t]);
12                else res.push_back(res[i - t] + 1);
13            }
14            ++k;
15        }
16        return res;
17    }
18 };

```

下面来看一种投机取巧的方法，直接利用了built-in的函数bitset的count函数可以直接返回1的个数，题目中说了不提倡用这种方法，写出来只是多一种思路而已：

解法2：

```

1 class Solution {
2 public:
3     vector<int> countBits(int num) {
4         vector<int> res;
5         for (int i = 0; i <= num; ++i) {
6             res.push_back(bitset<32>(i).count());
7         }
8         return res;
9     }
10 };

```

下面这种方法相比第一种方法就要简洁很多了，这个规律找的更好，规律是，从1开始，遇到偶数时，其1的个数和该偶数除以2得到的数字的1的个数相同，遇到奇数时，其1的个数等于该奇数除以2得到的数字的1的个数再加1，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     vector<int> countBits(int num) {
4         vector<int> res{0};
5         for (int i = 1; i <= num; ++i) {
6             if (i % 2 == 0) res.push_back(res[i / 2]);
7             else res.push_back(res[i / 2] + 1);
8         }
9         return res;
10    }
11 };

```

下面这种方法就更加巧妙了，巧妙的利用了*i&(i - 1)*，这个本来是用来判断一个数是否是2的指数的快捷方法，比如8，二进制位1000，那么8&(8-1)为0，只要为0就是2的指数，那么我们现在来看一下0到15的数字和其对应的*i&(i - 1)*值：

```

i    bin      '1'    i&(i-1)
0    0000    0
-----
1    0001    1    0000
-----
2    0010    1    0000
3    0011    2    0010
-----
4    0100    1    0000
5    0101    2    0100
6    0110    2    0100
7    0111    3    0110
-----
8    1000    1    0000
9    1001    2    1000
10   1010    2    1000
11   1011    3    1010
12   1100    2    1000
13   1101    3    1100
14   1110    3    1100
15   1111    4    1110

```

我们可以发现每个i值都是i&(i-1)对应的值加1，这样我们就可以写出代码如下：

解法4：

```

1 class Solution {
2 public:
3     vector<int> countBits(int num) {
4         vector<int> res(num + 1, 0);
5         for (int i = 1; i <= num; ++i) {
6             res[i] = res[i & (i - 1)] + 1;
7         }
8         return res;
9     }
10 };

```

CPP

339. 嵌套链表权重和

Given a nested list of integers, return the sum of all integers in the list weighted by their depth.

Each element is either an integer, or a list -- whose elements may also be integers or other lists.

Example 1:

Given the list [[1,1],2,[1,1]], return 10. (four 1's at depth 2, one 2 at depth 1)

这道题定义了一种嵌套链表的结构，链表可以无限往里嵌套，规定每嵌套一层，深度加1，让我们求权重之和，就是每个数字乘以其权重，再求总和。那么我们考虑，由于嵌套层数可以很大，所以我们用深度优先搜索DFS会很简单，每次遇到嵌套的，递归调用函数，一层一层往里算就可以了，我最先想的方法是遍历给的嵌套链表的数组，对于每个嵌套链表的对象，调用getSum函数，并赋深度值1，累加起来返回。在getSum函数中，首先判断其是否为整数，如果是，则返回当前深度乘以整数，如果不是，那么我们再遍历嵌套数组，对每个嵌套链表再调用递归函数，将返回值累加起来返回即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int depthSum(vector<NestedInteger>& nestedList) {
4         int res = 0;
5         for (auto a : nestedList) {
6             res += getSum(a, 1);
7         }
8         return res;
9     }
10    int getSum(NestedInteger ni, int level) {
11        int res = 0;
12        if (ni.isInteger()) return level * ni.getInteger();
13        for (auto a : ni.getList()) {
14            res += getSum(a, level + 1);
15        }
16        return res;
17    }
18 };

```

但其实上面的方法可以优化，我们可以把给的那个嵌套链表的一维数组直接当做一个嵌套链表的对象，然后调用递归函数，递归函数的处理方法跟上面一样，只不过用了个三元处理使其看起来更加简洁了一些：

解法2：

```

1 class Solution {
2 public:
3     int depthSum(vector<NestedInteger>& nestedList) {
4         return helper(nestedList, 1);
5     }
6     int helper(vector<NestedInteger>& nl, int depth) {
7         int res = 0;
8         for (auto a : nl) {
9             res += a.isInteger() ? a.getInteger() * depth : helper(a.getList(), depth + 1);
10        }
11        return res;
12    }
13 };

```

340. 最多有K个不同字符的最长子串

Given a string, find the length of the longest substring T that contains at most k distinct characters.

For example, Given s = "eceba" and k = 2,

T is "ece" which its length is 3.

这道题是之前那道Longest Substring with At Most Two Distinct Characters的拓展，而且那道题中的解法一和解法二直接将2换成k就行了，具体讲解请参考之前那篇博客：

解法1：

```

1 class Solution {
2 public:
3     int lengthOfLongestSubstringKDistinct(string s, int k) {
4         int res = 0, left = 0;
5         unordered_map<char, int> m;
6         for (int i = 0; i < s.size(); ++i) {
7             ++m[s[i]];
8             while (m.size() > k) {
9                 if (--m[s[left]] == 0) m.erase(s[left]);
10                ++left;
11            }
12            res = max(res, i - left + 1);
13        }
14        return res;
15    }
16 };

```

具体讲解请参考之前那篇博客Longest Substring with At Most Two Distinct Characters，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int lengthOfLongestSubstringKDistinct(string s, int k) {
4         int res = 0, left = 0;
5         unordered_map<char, int> m;
6         for (int i = 0; i < s.size(); ++i) {
7             m[s[i]] = i;
8             while (m.size() > k) {
9                 if (m[s[left]] == left) m.erase(s[left]);
10                ++left;
11            }
12            res = max(res, i - left + 1);
13        }
14        return res;
15    }
16 };

```

341. 压平嵌套链表迭代器

Given a nested list of integers, implement an iterator to flatten it.

Each element is either an integer, or a list -- whose elements may also be integers or other lists.

Example 1:

Given the list [[1,1],2,[1,1]],

By calling next repeatedly until hasNext returns false, the order of elements returned by next should be: [1,1,2,1,1].

这道题让我们建立压平嵌套链表的迭代器，关于嵌套链表的数据结构最早出现在Nested List Weight Sum中，而那道题是用的递归的方法来解的，而迭代器一般都是用迭代的方法来解的，而递归一般都需用栈来辅助遍历，由于栈的后进先出的特性，我们在对向量遍历的时候，从后往前把对象压入栈中，那么第一个对象最后压入栈就会第一个取出来处理，我们的hasNext()函数需要

遍历栈，并进行处理，如果栈顶元素是整数，直接返回true，如果不是，那么移除栈顶元素，并开始遍历这个取出的list，还是从后往前压入栈，循环停止条件是栈为空，返回false，参见代码如下：

解法1：

```

1 class NestedIterator {
2 public:
3     NestedIterator(vector<NestedInteger> &nestedList) {
4         for (int i = nestedList.size() - 1; i >= 0; --i) {
5             s.push(nestedList[i]);
6         }
7     }
8
9     int next() {
10        NestedInteger t = s.top(); s.pop();
11        return t.getInteger();
12    }
13
14    bool hasNext() {
15        while (!s.empty()) {
16            NestedInteger t = s.top();
17            if (t.isInteger()) return true;
18            s.pop();
19            for (int i = t.getList().size() - 1; i >= 0; --i) {
20                s.push(t.getList()[i]);
21            }
22        }
23        return false;
24    }
25
26 private:
27     stack<NestedInteger> s;
28 };

```

我们也可以使用deque来代替stack，实现思路和上面完全一样，参见代码如下：

解法2：

```
1 class NestedIterator {
2 public:
3     NestedIterator(vector<NestedInteger> &nestedList) {
4         for (auto a : nestedList) {
5             d.push_back(a);
6         }
7     }
8
9     int next() {
10        NestedInteger t = d.front(); d.pop_front();
11        return t.getInteger();
12    }
13
14    bool hasNext() {
15        while (!d.empty()) {
16            NestedInteger t = d.front();
17            if (t.isInteger()) return true;
18            d.pop_front();
19            for (int i = 0; i < t.getList().size(); ++i) {
20                d.insert(d.begin() + i, t.getList()[i]);
21            }
22        }
23        return false;
24    }
25
26 private:
27     deque<NestedInteger> d;
28 };
```

虽说迭代器是要用迭代的方法，但是我们可以强行使用递归来解，怎么个强行法呢，就是我们使用一个队列queue，在构造函数的时候就利用迭代的方法把这个嵌套链表全部压平展开，然后在调用hasNext()和next()就很简单了：

解法3：

```

1 class NestedIterator {
2 public:
3     NestedIterator(vector<NestedInteger> &nestedList) {
4         make_queue(nestedList);
5     }
6
7     int next() {
8         int t = q.front(); q.pop();
9         return t;
10    }
11
12    bool hasNext() {
13        return !q.empty();
14    }
15
16 private:
17    queue<int> q;
18    void make_queue(vector<NestedInteger> &nestedList) {
19        for (auto a : nestedList) {
20            if (a.isInteger()) q.push(a.getInteger());
21            else make_queue(a.getList());
22        }
23    }
24 };

```

342. 判断4的次方数

Given an integer (signed 32 bits), write a function to check whether it is a power of 4.

Example:

Given num = 16, return true. Given num = 5, return false.

Follow up: Could you solve it without loops/recursion?

这道题让我们判断一个数是否为4的次方数，那么最直接的方法就是不停的除以4，看最终结果是否为1，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool isPowerOfFour(int num) {
4         while (num && (num % 4 == 0)) {
5             num /= 4;
6         }
7         return num == 1;
8     }
9 };

```

还有一种方法是跟Power of Three中的解法三一样，使用换底公式来做，讲解请参见之前那篇博客：

解法2：

```

1 class Solution {
2 public:
3     bool isPowerOfFour(int num) {
4         return num > 0 && int(log10(num) / log10(4)) - log10(num) / log10(4) == 0;
5     }
6 };

```

下面这种方法是网上比较流行的一种解法，思路很巧妙，首先根据Power of Two中的解法二，我们知道 $num \& (num - 1)$ 可以用来判断一个数是否为2的次方数，更进一步说，就是二进制表示下，只有最高位是1，那么由于是2的次方数，不一定是4的次方数，比如8，所以我们还要其他的限定条件，我们仔细观察可以发现，4的次方数的最高位的1都是计数位，那么我们只需与上一个数 $0x55555555 \iff 101010101010101010101010101$ ，如果得到的数还是其本身，则可以肯定其为4的次方数：

解法3：

```

1 class Solution {
2 public:
3     bool isPowerOfFour(int num) {
4         return num > 0 && !(num & (num - 1)) && (num & 0x55555555) == num;
5     }
6 };

```

或者我们在确定其是2的次方数了之后，发现只要是4的次方数，减1之后可以被3整除，所以可以写出代码如下：

解法4：

```

1 class Solution {
2 public:
3     bool isPowerOfFour(int num) {
4         return num > 0 && !(num & (num - 1)) && (num - 1) % 3 == 0;
5     }
6 };

```

343. 整数拆分

Given a positive integer n , break it into the sum of at least two positive integers and maximize the product of those integers. Return the maximum product you can get.

For example, given $n = 2$, return 1 ($2 = 1 + 1$); given $n = 10$, return 36 ($10 = 3 + 3 + 4$).

Note: you may assume that n is not less than 2 .

Hint:

There is a simple $O(n)$ solution to this problem.

You may check the breaking results of n ranging from 7 to 10 to discover the regularities.

这道题给了我们一个正整数n，让我们拆分成至少两个正整数之和，使其乘积最大，题目提示中让我们用O(n)来解题，而且告诉我们找7到10之间的规律，那么我们一点一点的来分析：

正整数从1开始，但是1不能拆分成两个正整数之和，所以不能当输出。

那么2只能拆成1+1，所以乘积也为1。

数字3可以拆分成2+1或1+1+1，显然第一种拆分方法乘积大为2。

数字4拆成2+2，乘积最大，为4。

数字5拆成3+2，乘积最大，为6。

数字6拆成3+3，乘积最大，为9。

数字7拆为3+4，乘积最大，为12。

数字8拆为3+3+2，乘积最大，为18。

数字9拆为3+3+3，乘积最大，为27。

数字10拆为3+3+4，乘积最大，为36。

....

那么通过观察上面的规律，我们可以看出从5开始，数字都需要先拆出所有的3，一直拆到剩下一个数为2或者4，因为剩4就不用再拆了，拆成两个2和不拆没有意义，而且4不能拆出一个3剩一个1，这样会比拆成2+2的乘积小。那么这样我们就可以写代码了，先预处理n为2和3的情况，然后先将结果res初始化为1，然后当n大于4开始循环，我们结果自乘3，n自减3，根据之前的分析，当跳出循环时，n只能是2或者4，再乘以res返回即可：

解法1：

```

1 class Solution {
2 public:
3     int integerBreak(int n) {
4         if (n == 2 || n == 3) return n - 1;
5         int res = 1;
6         while (n > 4) {
7             res *= 3;
8             n -= 3;
9         }
10        return res * n;
11    }
12 };

```

CPP

我们再来观察上面列出的10之前数字的规律，我们还可以发现数字7拆分结果是数字4的三倍，而7比4正好大3，数字8拆分结果是数字5的三倍，而8比5大3，后面都是这样的规律，那么我们可以把数字6之前的拆分结果都列举出来，然后之后的数通过查表都能计算出来，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int integerBreak(int n) {
4         vector<int> dp{0, 0, 1, 2, 4, 6, 9};
5         for (int i = 7; i <= n; ++i) {
6             dp.push_back(3 * dp[i - 3]);
7         }
8         return dp[n];
9     }
10 };

```

下面这种解法是热心网友留言告诉博主的，感觉很叼，故而补充上来。是解法一的一种变形写法，不再使用while循环了，而是直接分别算出能拆出3的个数和最后剩下的余数2或者4，然后直接相乘得到结果，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int integerBreak(int n) {
4         if (n == 2 || n == 3) return n - 1;
5         if (n == 4) return 4;
6         n -= 5;
7         return (int)pow(3, (n / 3 + 1)) * (n % 3 + 2);
8     }
9 };

```

344. 翻转字符串

Write a function that takes a string as input and returns the string reversed.

Example:

Given s = "hello", return "olleh".

这道题没什么难度，直接从两头往中间走，同时交换两边的字符即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     string reverseString(string s) {
4         int left = 0, right = s.size() - 1;
5         while (left < right) {
6             char t = s[left];
7             s[left++] = s[right];
8             s[right--) = t;
9         }
10     }
11 }
12 };

```

我们也可以用swap函数来帮助我们翻转：

解法2：

```

1 class Solution {
2 public:
3     string reverseString(string s) {
4         int left = 0, right = s.size() - 1;
5         while (left < right) {
6             swap(s[left++], s[right--]);
7         }
8         return s;
9     }
10 };

```

345. 翻转字符串中的元音字母

Write a function that takes a string as input and reverse only the vowels of a string.

Example 1:

Given s = "hello", return "holle".

Example 2:

Given s = "leetcode", return "leotcede".

这道题让我们翻转字符串中的元音字母，元音字母有五个a,e,i,o,u，需要注意的是大写的也算，所以总共有十个字母。我们写一个isVowel的函数来判断当前字符是否为元音字母，如果两边都是元音字母，那么我们交换，如果左边的不是，向右移动一位，如果右边的不是，则向左移动一位，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     string reverseVowels(string s) {
4         int left = 0, right= s.size() - 1;
5         while (left < right) {
6             if (isVowel(s[left]) && isVowel(s[right])) {
7                 swap(s[left++], s[right--]);
8             } else if (isVowel(s[left])) {
9                 --right;
10            } else {
11                ++left;
12            }
13        }
14        return s;
15    }
16    bool isVowel(char c) {
17        return c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u' || c == 'A' || c ==
18        'E' || c == 'I' || c == 'O' || c == 'U';
19    }
};

```

或者我们也可以用自带函数find_first_of和find_last_of来找出包含给定字符串中任意一个字符的下一个位置进行交换即可：

解法2:

```

1 class Solution {
2 public:
3     string reverseVowels(string s) {
4         int left = 0, right = s.size() - 1;
5         while (left < right) {
6             left = s.find_first_of("aeiouAEIOU", left);
7             right = s.find_last_of("aeiouAEIOU", right);
8             if (left < right) {
9                 swap(s[left++], s[right--]);
10            }
11        }
12     return s;
13 }
14 };

```

我们也可以把元音字母都存在一个字符串里，然后每遇到一个字符，就到元音字符串里去找，如果存在就说明当前字符是元音字符，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     string reverseVowels(string s) {
4         int left = 0, right = s.size() - 1;
5         string t = "aeiouAEIOU";
6         while (left < right) {
7             if (t.find(s[left]) == string::npos) ++left;
8             else if (t.find(s[right]) == string::npos) --right;
9             else swap(s[left++], s[right--]);
10        }
11     return s;
12 }
13 };

```

346. 从数据流中移动平均值

Given a stream of integers and a window size, calculate the moving average of all integers in the sliding window.

For example,

```

MovingAverage m = new MovingAverage(3);
m.next(1) = 1
m.next(10) = (1 + 10) / 2
m.next(3) = (1 + 10 + 3) / 3
m.next(5) = (10 + 3 + 5) / 3

```

这道题定义了一个MovingAverage类，里面可以存固定个数字，然后我们每次读入一个数字，如果加上这个数字后总个数大于限制的个数，那么我们移除最早进入的数字，然后返回更新后的平均数，这种先进先出的特性最适合使用队列queue来做，而且我们还需要一个double型的变量sum来记录当前所有数字之和，这样有新数字进入后，如果没有超出限制个数，则sum加上这个数字，如果超出了，那么sum先减去最早的数字，再加上这个数字，然后返回sum除以queue的个数即可：

```

1 class MovingAverage {
2 public:
3     MovingAverage(int size) {
4         this->size = size;
5         sum = 0;
6     }
7
8     double next(int val) {
9         if (q.size() >= size) {
10             sum -= q.front(); q.pop();
11         }
12         q.push(val);
13         sum += val;
14         return sum / q.size();
15     }
16
17 private:
18     queue<int> q;
19     int size;
20     double sum;
21 };

```

347. 前K个高频元素

Given a non-empty array of integers, return the k most frequent elements.

Example 1:

Input: nums = [1,1,1,2,2,3], k = 2
Output: [1,2]

这道题给了我们一个数组，让我们统计前k个高频的数字，那么对于这类的统计数字的问题，首先应该考虑用HashMap来做，建立数字和其出现次数的映射，然后再按照出现次数进行排序。我们可以用堆排序来做，使用一个最大堆来按照映射次数从大到小排列，在C++中使用priority_queue来实现，默认是最大堆，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<int> topKFrequent(vector<int>& nums, int k) {
4         unordered_map<int, int> m;
5         priority_queue<pair<int, int>> q;
6         vector<int> res;
7         for (auto a : nums) ++m[a];
8         for (auto it : m) q.push({it.second, it.first});
9         for (int i = 0; i < k; ++i) {
10             res.push_back(q.top().second); q.pop();
11         }
12         return res;
13     }
14 };

```

当然，既然可以使用最大堆，还有一种可以自动排序的数据结构TreeMap，也是可以的，这里就不写了，因为跟上面的写法基本没啥区别，就是换了一个数据结构。

我们还可以使用桶排序，在建立好数字和其出现次数的映射后，我们按照其出现次数将数字放到对应的位置中去，这样我们从桶的后面向前面遍历，最先得到的就是出现次数最多的数字，我们找到k个后返回即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<int> topKFrequent(vector<int>& nums, int k) {
4         unordered_map<int, int> m;
5         vector<vector<int>> bucket(nums.size() + 1);
6         vector<int> res;
7         for (auto a : nums) ++m[a];
8         for (auto it : m) {
9             bucket[it.second].push_back(it.first);
10        }
11        for (int i = nums.size(); i >= 0; --i) {
12            for (int j = 0; j < bucket[i].size(); ++j) {
13                res.push_back(bucket[i][j]);
14                if (res.size() == k) return res;
15            }
16        }
17        return res;
18    }
19 };

```

CPP

348. 设计井字棋游戏

Design a Tic-tac-toe game that is played between two players on a $n \times n$ grid.

You may assume the following rules:

A move is guaranteed to be valid and is placed on an empty block.

Once a winning condition is reached, no more moves is allowed.

A player who succeeds in placing n of their marks in a horizontal, vertical, or diagonal row wins the game.

Example:

Given $n = 3$, assume that player 1 is "X" and player 2 is "O" in the board.

```
TicTacToe toe = new TicTacToe(3);
```

```
toe.move(0, 0, 1); -> Returns 0 (no one wins)
```

```
|X| | |
| | | // Player 1 makes a move at (0, 0).
| | |
```

```
toe.move(0, 2, 2); -> Returns 0 (no one wins)
```

```
|X| |O|
| | | // Player 2 makes a move at (0, 2).
| | |
```

CareerCup上的原题，请参见我之前的博客17.2 Tic Tac Toe。我们首先来 $O(n^2)$ 的解法，这种方法的思路很straightforward，就是建立一个 $n \times n$ 大小的board，其中0表示该位置没有棋子，1表示玩家1放的子，2表示玩家2。那么棋盘上每增加一个子，我们都扫描当前行列，对角线，和逆对角线，看看是否有三子相连的情况，有的话则返回对应的玩家，没有则返回0，参见代码如下：

解法1：

```

1 class TicTacToe {
2     public:
3         /** Initialize your data structure here. */
4         TicTacToe(int n) {
5             board.resize(n, vector<int>(n, 0));
6         }
7
8         int move(int row, int col, int player) {
9             board[row][col] = player;
10            int i = 0, j = 0, n = board.size();
11            for (j = 1; j < n; ++j) {
12                if (board[row][j] != board[row][j - 1]) break;
13            }
14            if (j == n) return player;
15            for (i = 1; i < n; ++i) {
16                if (board[i][col] != board[i - 1][col]) break;
17            }
18            if (i == n) return player;
19            if (row == col) {
20                for (i = 1; i < n; ++i) {
21                    if (board[i][i] != board[i - 1][i - 1]) break;
22                }
23                if (i == n) return player;
24            }
25            if (row + col == n - 1) {
26                for (i = 1; i < n; ++i) {
27                    if (board[n - i - 1][i] != board[n - i][i - 1]) break;
28                }
29                if (i == n) return player;
30            }
31            return 0;
32        }
33
34     private:
35         vector<vector<int>> board;
36     };

```

Follow up中让我们用更高效的方法，那么根据提示中的，我们建立一个大小为n的一维数组rows和cols，还有变量对角线diag和逆对角线rev_diag，这种方法的思路是，如果玩家1在第一行某一列放了一个子，那么rows[0]自增1，如果玩家2在第一行某一列放了一个子，则rows[0]自减1，那么只有当rows[0]等于n或者-n的时候，表示第一行的子都是一个玩家放的，则游戏结束返回该玩家即可，其他各行各列，对角线和逆对角线都是这种思路，参见代码如下：

解法2：

```

1 class TicTacToe {
2 public:
3     /** Initialize your data structure here. */
4     TicTacToe(int n): rows(n), cols(n), N(n), diag(0), rev_diag(0) {}
5
6     int move(int row, int col, int player) {
7         int add = player == 1 ? 1 : -1;
8         rows[row] += add;
9         cols[col] += add;
10        diag += (row == col ? add : 0);
11        rev_diag += (row == N - col - 1 ? add : 0);
12        return (abs(rows[row]) == N || abs(cols[col]) == N || abs(diag) == N ||
13 abs(rev_diag) == N) ? player : 0;
14    }
15
16 private:
17     vector<int> rows, cols;
18     int diag, rev_diag, N;
19 };

```

349. 两个数组相交

Given two arrays, write a function to compute their intersection.

Example:

Given `nums1 = [1, 2, 2, 1]`, `nums2 = [2, 2]`, return `[2]`.

Note:

Each element in the result must be unique.

The result can be in any order.

这道题让我们找两个数组交集的部分（不包含重复数字），难度不算大，我们可以用个set把`nums1`都放进去，然后遍历`nums2`的元素，如果在set中存在，说明是交集的部分，加入结果的set中，最后再把结果转为vector的形式即可：

解法1:

```

1 class Solution {
2 public:
3     vector<int> intersection(vector<int>& nums1, vector<int>& nums2) {
4         set<int> s(nums1.begin(), nums1.end()), res;
5         for (auto a : nums2) {
6             if (s.count(a)) res.insert(a);
7         }
8         return vector<int>(res.begin(), res.end());
9     }
10 };

```

我们还可以使用两个指针来做，先给两个数组排序，然后用两个指针分别指向两个数组的开头，然后比较两个数组的大小，把小的数字的指针向后移，如果两个指针指的数字相等，那么看结果`res`是否为空，如果为空或者是最后一个数字和当前数字不等的话，将该数字加入结果`res`中，参见代码如下：

解法2:

```

1 class Solution {
2 public:
3     vector<int> intersection(vector<int>& nums1, vector<int>& nums2) {
4         vector<int> res;
5         int i = 0, j = 0;
6         sort(nums1.begin(), nums1.end());
7         sort(nums2.begin(), nums2.end());
8         while (i < nums1.size() && j < nums2.size()) {
9             if (nums1[i] < nums2[j]) ++i;
10            else if (nums1[i] > nums2[j]) ++j;
11            else {
12                if (res.empty() || res.back() != nums1[i]) {
13                    res.push_back(nums1[i]);
14                }
15                ++i; ++j;
16            }
17        }
18        return res;
19    }
20 };

```

我们还可以使用二分查找法来做，思路是将一个数组排序，然后遍历另一个数组，把遍历到的每个数字在排序号的数组中用二分查找法搜索，如果能找到则放入结果set中，这里我们用到了set的去重复的特性，最后我们将set转为vector即可：

解法3：

```

1 class Solution {
2 public:
3     vector<int> intersection(vector<int>& nums1, vector<int>& nums2) {
4         set<int> res;
5         sort(nums2.begin(), nums2.end());
6         for (auto a : nums1) {
7             if (binarySearch(nums2, a)) {
8                 res.insert(a);
9             }
10        }
11        return vector<int>(res.begin(), res.end());
12    }
13    bool binarySearch(vector<int> &nums, int target) {
14        int left = 0, right = nums.size();
15        while (left < right) {
16            int mid = left + (right - left) / 2;
17            if (nums[mid] == target) return true;
18            else if (nums[mid] < target) left = mid + 1;
19            else right = mid;
20        }
21        return false;
22    }
23 };

```

或者我们也可以使用STL的set_intersection函数来找出共同元素，很方便：

解法4：

```

1 class Solution {
2 public:
3     vector<int> intersection(vector<int>& nums1, vector<int>& nums2) {
4         set<int> s1(nums1.begin(), nums1.end()), s2(nums2.begin(), nums2.end()), res;
5         set_intersection(s1.begin(), s1.end(), s2.begin(), s2.end(), inserter(res,
6             res.begin())));
7         return vector<int>(res.begin(), res.end());
8     }
9 };

```

350. 两个数组相交之二

Given two arrays, write a function to compute their intersection.

Example 1:

Input: nums1 = [1,2,2,1], nums2 = [2,2]
Output: [2,2]

Note:

Each element in the result should appear as many times as it shows in both arrays.

The result can be in any order.

Follow up:

What if the given array is already sorted? How would you optimize your algorithm?

What if nums1's size is small compared to nums2's size? Which algorithm is better?

What if elements of nums2 are stored on disk, and the memory is limited such that you cannot load all elements into the memory at once?

这道题是之前那道Intersection of Two Arrays的拓展，不同之处在于这道题允许我们返回重复的数字，而且是尽可能多的返回，之前那道题是说有重复的数字只返回一个就行。那么这道题我们用哈希表来建立nums1中字符和其出现个数之间的映射，然后遍历nums2数组，如果当前字符在哈希表中的个数大于0，则将此字符加入结果res中，然后哈希表的对应值自减1，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<int> intersect(vector<int>& nums1, vector<int>& nums2) {
4         unordered_map<int, int> m;
5         vector<int> res;
6         for (auto a : nums1) ++m[a];
7         for (auto a : nums2) {
8             if (m[a]-- > 0) res.push_back(a);
9         }
10    }
11 }
12 
```

再来看一种方法，这种方法先给两个数组排序，然后用两个指针分别指向两个数组的起始位置，如果两个指针指的数字相等，则存入结果中，两个指针均自增1，如果第一个指针指的数字大，则第二个指针自增1，反之亦然，参见代码如下：

解法2：

```

1 class Solution {
2     public:
3         vector<int> intersect(vector<int>& nums1, vector<int>& nums2) {
4             vector<int> res;
5             int i = 0, j = 0;
6             sort(nums1.begin(), nums1.end());
7             sort(nums2.begin(), nums2.end());
8             while (i < nums1.size() && j < nums2.size()) {
9                 if (nums1[i] == nums2[j]) {
10                     res.push_back(nums1[i]);
11                     ++i; ++j;
12                 } else if (nums1[i] > nums2[j]) {
13                     ++j;
14                 } else {
15                     ++i;
16                 }
17             }
18             return res;
19         }
20     };

```

351. 安卓解锁模式

Given an Android 3x3 key lock screen and two integers m and n , where $1 \leq m \leq n \leq 9$, count the total number of unlock patterns of the Android lock screen, which consist of minimum of m keys and maximum n keys.

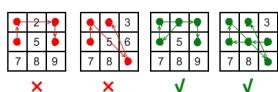
Rules for a valid pattern:

Each pattern must connect at least m keys and at most n keys.

All the keys must be distinct.

If the line connecting two consecutive keys in the pattern passes through any other keys, the other keys must have previously selected in the pattern. No jumps through non selected key is allowed.

The order of keys used matters.



Explanation:

	1		2		3	
	4		5		6	
	7		8		9	

Invalid move: 4 - 1 - 3 - 6

Line 1 - 3 passes through key 2 which had not been selected in the pattern.

Invalid move: 4 - 1 - 9 - 2

Line 1 - 9 passes through key 5 which had not been selected in the pattern.

Valid move: 2 - 4 - 1 - 3 - 6

Line 1 - 3 is valid because it passes through key 2, which had been selected in the pattern

Valid move: 6 - 5 - 4 - 1 - 9 - 2

Line 1 - 9 is valid because it passes through key 5, which had been selected in the pattern.

Example:

Given m = 1, n = 1, return 9.

这道题乍一看题目这么长以为是一个设计题，其实不是，这道题还是比较有意思的，起码跟实际结合的比较紧密。这道题说的是安卓机子的解锁方法，有9个数字键，如果密码的长度范围在[m, n]之间，问所有的解锁模式共有多少种，注意题目中给出的一些非法的滑动模式。那么我们先来看一下哪些是非法的，首先1不能直接到3，必须经过2，同理的有4到6, 7到9, 1到7, 2到8, 3到9，还有就是对角线必须经过5，例如1到9, 3到7等。我们建立一个二维数组jumps，用来记录两个数字键之间是否有中间键，然后再用一个一位数组visited来记录某个键是否被访问过，然后我们用递归来解，我们先对1调用递归函数，在递归函数中，我们遍历1到9每个数字next，然后找他们之间是否有jump数字，如果next没被访问过，并且jump为0，或者jump被访问过，我们对next调用递归函数。数字1的模式个数算出来后，由于1,3,7,9是对称的，所以我们乘4即可，然后再对数字2调用递归函数，2,4,6,9也是对称的，再乘4，最后单独对5调用一次，然后把所有的加起来就是最终结果了，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     int numberOfPatterns(int m, int n) {
4         int res = 0;
5         vector<bool> visited(10, false);
6         vector<vector<int>> jumps(10, vector<int>(10, 0));
7         jumps[1][3] = jumps[3][1] = 2;
8         jumps[4][6] = jumps[6][4] = 5;
9         jumps[7][9] = jumps[9][7] = 8;
10        jumps[1][7] = jumps[7][1] = 4;
11        jumps[2][8] = jumps[8][2] = 5;
12        jumps[3][9] = jumps[9][3] = 6;
13        jumps[1][9] = jumps[9][1] = jumps[3][7] = jumps[7][3] = 5;
14        res += helper(1, 1, 0, m, n, jumps, visited) * 4;
15        res += helper(2, 1, 0, m, n, jumps, visited) * 4;
16        res += helper(5, 1, 0, m, n, jumps, visited);
17        return res;
18    }
19    int helper(int num, int len, int res, int m, int n, vector<vector<int>> &jumps,
20    vector<bool> &visited) {
21        if (len >= m) ++res;
22        ++len;
23        if (len > n) return res;
24        visited[num] = true;
25        for (int next = 1; next <= 9; ++next) {
26            int jump = jumps[num][next];
27            if (!visited[next] && (jump == 0 || visited[jump])) {
28                res = helper(next, len, res, m, n, jumps, visited);
29            }
30        }
31        visited[num] = false;
32        return res;
33    }
34};

```

下面这种方法很简洁，但是不容易理解，讲解请看这个帖子。其中used是一个9位的mask，每位对应一个数字，如果为1表示存在，0表示不存在， (i_1, j_1) 是之前的位置， (i, j) 是当前的位置，所以滑动是从 (i_1, j_1) 到 (i, j) ，中间点为 $((i_1+i)/2, (j_1+j)/2)$ ，这里的I和J分别为 i_1+i 和 j_1+j ，还没有除以2，所以I和J都是整数。如果I%2或者J%2不为0，说明中间点的坐标不是整数，即中间点不存在，如果中间点存在，如果中间点被使用了，则这条线也是成立的，可以调用递归，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int numberOfPatterns(int m, int n) {
4         return count(m, n, 0, 1);
5     }
6     int count(int m, int n, int used, int i1, int j1) {
7         int res = m <= 0;
8         if (!n) return 1;
9         for (int i = 0; i < 3; ++i) {
10            for (int j = 0; j < 3; ++j) {
11                int I = i1 + i, J = j1 + j, used2 = used | (1 << (i * 3 + j));
12                if (used2 > used && (I % 2 || J % 2 || used2 & (1 << (I / 2 * 3 + J / 2))))
13                {
14                    res += count(m - 1, n - 1, used2, i, j);
15                }
16            }
17        }
18        return res;
19    }
};
```

352. 分离区间的数据流

Given a data stream input of non-negative integers $a_1, a_2, \dots, a_n, \dots$, summarize the numbers seen so far as a list of disjoint intervals.

For example, suppose the integers from the data stream are 1, 3, 7, 2, 6, ..., then the summary will be:

```
[1, 1]
[1, 1], [3, 3]
[1, 1], [3, 3], [7, 7]
[1, 3], [7, 7]
[1, 3], [6, 7]
```

Follow up:

What if there are lots of merges and the number of disjoint intervals are small compared to the data stream's size?

这道题说有个数据流每次提供一个数字，然后让我们组成一系列分离的区间，这道题跟之前那道Insert Interval很像，思路也很像，每进来一个新的数字val，我们都生成一个新的区间 $[val, val]$ ，然后将其插入到当前的区间里，注意分情况讨论，无重叠，相邻，和有重叠分开讨论处理，参见代码如下：

```
1 class SummaryRanges {
2 public:
3     /** Initialize your data structure here. */
4     SummaryRanges() {}
5
6     void addNum(int val) {
7         Interval cur(val, val);
8         vector<Interval> res;
9         int pos = 0;
10        for (auto a : v) {
11            if (cur.end + 1 < a.start) {
12                res.push_back(a);
13            } else if (cur.start > a.end + 1) {
14                res.push_back(a);
15                ++pos;
16            } else {
17                cur.start = min(cur.start, a.start);
18                cur.end = max(cur.end, a.end);
19            }
20        }
21        res.insert(res.begin() + pos, cur);
22        v = res;
23    }
24
25    vector<Interval> getIntervals() {
26        return v;
27    }
28
29 private:
30     vector<Interval> v;
31 }
```

353. 设计贪吃蛇游戏

Design a Snake game that is played on a device with screen size = width x height. Play the game online if you are not familiar with the game.

The snake is initially positioned at the top left corner (0,0) with length = 1 unit.

You are given a list of food's positions in row-column order. When a snake eats the food, its length and the game's score both increase by 1.

Each food appears one by one on the screen. For example, the second food will not appear until the first food was eaten by the snake.

When a food does appear on the screen, it is guaranteed that it will not appear on a block occupied by the snake.

Example:

Given width = 3, height = 2, and food = [[1,2],[0,1]].

```
Snake snake = new Snake(width, height, food);
```

Initially the snake appears at position (0,0) and the food at (1,2).

```
|S| | |  
| | |F|
```

```
snake.move("R"); -> Returns 0
```

感觉最近LeetCode经常出一些design类的题目啊，难道算法类的题目都出完了吗，这道题让我们设计一个贪吃蛇的游戏，这是个简化版的，但是游戏规则还是保持不变，蛇可以往上下左右四个方向走，吃到食物就会变长1个，如果碰到墙壁或者自己的躯体，游戏就会结束。我们需要一个一维数组来保存蛇身的位置，由于蛇移动的过程的蛇头向前走一步，蛇尾也跟着往前，中间的躯体还在原来的位置，所以移动的结果就是，蛇头变到新位置，去掉蛇尾的位置即可。需要注意的是去掉蛇尾的位置是在检测和蛇身的碰撞之前还是之后，如果是之后则无法通过这个test case: [[3,3,[[2,0],[0,0]],["D"],["D"],["U"]]]，如果是之前就没有问题了，检测蛇头和蛇身是否碰撞使用的是count(snake.begin0, snake.end0, head)，总体来说不算一道难题，参见代码如下：

```

1 class SnakeGame {
2     public:
3         /** Initialize your data structure here.
4             @param width - screen width
5             @param height - screen height
6             @param food - A list of food positions
7             E.g food = [[1,1], [1,0]] means the first food is positioned at [1,1], the second
8             is at [1,0]. */
9         SnakeGame(int width, int height, vector<pair<int, int>> food) {
10             this->width = width;
11             this->height = height;
12             this->food = food;
13             score = 0;
14             snake.push_back({0, 0});
15         }
16
17         /** Moves the snake.
18             @param direction - 'U' = Up, 'L' = Left, 'R' = Right, 'D' = Down
19             @return The game's score after the move. Return -1 if game over.
20             Game over when snake crosses the screen boundary or bites its body. */
21         int move(string direction) {
22             auto head = snake.front(), tail = snake.back();
23             snake.pop_back();
24             if (direction == "U") --head.first;
25             else if (direction == "L") --head.second;
26             else if (direction == "R") ++head.second;
27             else if (direction == "D") ++head.first;
28             if (count(snake.begin(), snake.end(), head) || head.first < 0 || head.first >=
29                 height || head.second < 0 || head.second >= width) {
30                 return -1;
31             }
32             snake.insert(snake.begin(), head);
33             if (!food.empty() && head == food.front()) {
34                 food.erase(food.begin());
35                 snake.push_back(tail);
36                 ++score;
37             }
38             return score;
39         }
40
41     private:
42         int width, height, score;
43         vector<pair<int, int>> food, snake;
44     };

```

354. 俄罗斯娃娃信封

You have a number of envelopes with widths and heights given as a pair of integers (w, h). One envelope can fit into another if and only if both the width and height of one envelope is greater than the width and height of the other envelope.

What is the maximum number of envelopes can you Russian doll? (put one inside other)

Example:

Given envelopes = [[5,4],[6,4],[6,7],[2,3]], the maximum number of envelopes you can Russian doll is 3 ([2,3] => [5,4] => [6,7])

这道题给了我们一堆大小不一的信封，让我们像套俄罗斯娃娃那样把这些信封都给套起来，这道题实际上是之前那道Longest Increasing Subsequence的具体应用，而且难度增加了，从一维变成了两维，但是万变不离其宗，解法还是一样的，首先来看DP的解法，这是一种brute force的解法，首先要给所有的信封按从小到大排序，首先根据宽度从小到大排，如果宽度相同，那么高度小的在前面，这是STL里面sort的默认排法，所以我们不用写其他的comparator，直接排就可以了，然后我们开始遍历，对于每一个信封，我们都遍历其前面所有的信封，如果当前信封的长和宽都比前面那个信封的大，那么我们更新dp数组，通过 $dp[i] = \max(dp[i], dp[j] + 1)$ 。然后我们每遍历完一个信封，都更新一下结果res，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int maxEnvelopes(vector<pair<int, int>>& envelopes) {
4         int res = 0, n = envelopes.size();
5         vector<int> dp(n, 1);
6         sort(envelopes.begin(), envelopes.end());
7         for (int i = 0; i < n; ++i) {
8             for (int j = 0; j < i; ++j) {
9                 if (envelopes[i].first > envelopes[j].first && envelopes[i].second >
10 envelopes[j].second) {
11                     dp[i] = max(dp[i], dp[j] + 1);
12                 }
13             }
14             res = max(res, dp[i]);
15         }
16         return res;
17     }
18 };

```

CPP

我们还可以使用二分查找法来优化速度，我们首先要做的还是给信封排序，但是这次排序和上面有些不同，信封的宽度还是从小到大排，但是宽度相等时，我们让高度大的在前面。那么现在问题就简化了成了找高度数字中的LIS，完全就和之前那道Longest Increasing Subsequence一样了，所以我们还是使用之前那题解法来做，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int maxEnvelopes(vector<pair<int, int>>& envelopes) {
4         vector<int> dp;
5         sort(envelopes.begin(), envelopes.end(), [](const pair<int, int> &a, const
6             pair<int, int> &b){
7             if (a.first == b.first) return a.second > b.second;
8             return a.first < b.first;
9         });
10        for (int i = 0; i < envelopes.size(); ++i) {
11            int left = 0, right = dp.size(), t = envelopes[i].second;
12            while (left < right) {
13                int mid = left + (right - left) / 2;
14                if (dp[mid] < t) left = mid + 1;
15                else right = mid;
16            }
17            if (right >= dp.size()) dp.push_back(t);
18            else dp[right] = t;
19        }
20        return dp.size();
21    }
22};

```

既然可以用二分查找法，那么使用STL的自带函数lower_bound也没啥问题了，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int maxEnvelopes(vector<pair<int, int>>& envelopes) {
4         vector<int> dp;
5         sort(envelopes.begin(), envelopes.end(), [](const pair<int, int> &a, const
6             pair<int, int> &b){
7             if (a.first == b.first) return a.second > b.second;
8             return a.first < b.first;
9         });
10        for (int i = 0; i < envelopes.size(); ++i) {
11            auto it = lower_bound(dp.begin(), dp.end(), envelopes[i].second);
12            if (it == dp.end()) dp.push_back(envelopes[i].second);
13            else *it = envelopes[i].second;
14        }
15        return dp.size();
16    }
17};

```

讨论：这道题的一个follow up是信封可以旋转，怎么的最长序列？答案是<3,4>加入，然后<4,3>也加入，再找最长序列。

355. 设计推特

Design a simplified version of Twitter where users can post tweets, follow/unfollow another user and is able to see the 10 most recent tweets in the user's news feed. Your design should support the following methods:

postTweet(userId, tweetId): Compose a new tweet.

getNewsFeed(userId): Retrieve the 10 most recent tweet ids in the user's news feed. Each item in the news feed must be posted by users who the user followed or by the user herself. Tweets must be ordered from most recent to least recent.

follow(followerId, followeeId): Follower follows a followee.

unfollow(followerId, followeeId): Follower unfollows a followee.

Example:

```
Twitter twitter = new Twitter();

// User 1 posts a new tweet (id = 5).
twitter.postTweet(1, 5);

// User 1's news feed should return a list with 1 tweet id -> [5].
twitter.getNewsFeed(1);
```

这道题让我们设计个简单的推特，具有发布消息，获得新鲜事，添加关注和取消关注等功能。我们需要用两个哈希表来做，第一个是建立用户和其所有好友之间的映射，另一个是建立用户和其所有消息之间的映射。由于获得新鲜事是需要按时间顺序排列的，那么我们可以用一个整型变量cnt来模拟时间点，每发一个消息，cnt自增1，那么我们就知道cnt大的是最近发的。那么我们在建立用户和其所有消息之间的映射时，还需要建立每个消息和其时间点cnt之间的映射。这道题的主要难点在于实现getNewsFeed()函数，这个函数获取自己和好友的最近10条消息，我们的做法是用户也添加到自己的好友列表中，然后遍历该用户的所有好友，遍历每个好友的所有消息，维护一个大小为10的哈希表，如果新遍历到的消息比哈希表中最早的消息要晚，那么将这个消息加入，然后删除掉最早的那个消息，这样我们就可以找出最近10条消息了，参见代码如下：

解法1：

```

1 class Twitter {
2 public:
3     /** Initialize your data structure here. */
4     Twitter() {
5         cnt = 0;
6     }
7
8     /** Compose a new tweet. */
9     void postTweet(int userId, int tweetId) {
10        follow(userId, userId);
11        tweets[userId].insert({cnt++, tweetId});
12    }
13
14     /** Retrieve the 10 most recent tweet ids in the user's news feed. Each item in the
15 news feed must be posted by users who the user followed or by the user herself. Tweets must
16 be ordered from most recent to least recent. */
17     vector<int> getNewsFeed(int userId) {
18         vector<int> res;
19         map<int, int> top10;
20         for (auto it = friends[userId].begin(); it != friends[userId].end(); ++it) {
21             int t = *it;
22             for (auto a = tweets[t].begin(); a != tweets[t].end(); ++a) {
23                 top10.insert({a->first, a->second});
24                 if (top10.size() > 10) top10.erase(top10.begin());
25             }
26         }
27         for (auto it = top10.rbegin(); it != top10.rend(); ++it) {
28             res.push_back(it->second);
29         }
30         return res;
31     }
32
33     /** Follower follows a followee. If the operation is invalid, it should be a no-op. */
34     void follow(int followerId, int followeeId) {
35         friends[followerId].insert(followeeId);
36     }
37
38     /** Follower unfollows a followee. If the operation is invalid, it should be a no-op.
39 */
40     void unfollow(int followerId, int followeeId) {
41         if (followerId != followeeId) {
42             friends[followerId].erase(followeeId);
43         }
44     }
45
46 private:
47     int cnt;
48     unordered_map<int, set<int>> friends;
49     unordered_map<int, map<int, int>> tweets;
50 };

```

下面这种方法和上面的基本一样，就是在保存用户所有消息的时候，用的是vector<pair<int, int>>，这样我们可以用priority_queue来帮助我们找出最新10条消息，参见代码如下：

解法2：

```

1  class Twitter {
2      public:
3          /** Initialize your data structure here. */
4          Twitter() {
5              cnt = 0;
6          }
7
8          /** Compose a new tweet. */
9          void postTweet(int userId, int tweetId) {
10             follow(userId, userId);
11             tweets[userId].push_back({cnt++, tweetId});
12         }
13
14         /** Retrieve the 10 most recent tweet ids in the user's news feed. Each item in the
15         news feed must be posted by users who the user followed or by the user herself. Tweets must
16         be ordered from most recent to least recent. */
17         vector<int> getNewsFeed(int userId) {
18             vector<int> res;
19             priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> q;
20             for (auto it = friends[userId].begin(); it != friends[userId].end(); ++it) {
21                 for (auto a = tweets[*it].begin(); a != tweets[*it].end(); ++a) {
22                     if (q.size() > 0 && q.top().first > a->first && q.size() > 10) break;
23                     q.push(*a);
24                     if (q.size() > 10) q.pop();
25                 }
26             }
27             while (!q.empty()) {
28                 res.push_back(q.top().second);
29                 q.pop();
30             }
31             reverse(res.begin(), res.end());
32             return res;
33         }
34
35         /** Follower follows a followee. If the operation is invalid, it should be a no-op. */
36         void follow(int followerId, int followeeId) {
37             friends[followerId].insert(followeeId);
38         }
39
40         /** Follower unfollows a followee. If the operation is invalid, it should be a no-op. */
41     */
42         void unfollow(int followerId, int followeeId) {
43             if (followerId != followeeId) {
44                 friends[followerId].erase(followeeId);
45             }
46         }
47
48     private:
49         int cnt;
50         unordered_map<int, set<int>> friends;
51         unordered_map<int, vector<pair<int, int>>> tweets;
52     };

```

356. 直线对称

Given n points on a 2D plane, find if there is such a line parallel to y-axis that reflect the given set of points.

Example 1:

Given points = [[1,1],[-1,1]], return true.

Example 2:

Given points = [[1,1],[-1,-1]], return false.

Follow up:

Could you do better than O(n²)?

Hint:

Find the smallest and largest x-value for all points.

If there is a line then it should be at $y = (\min X + \max X) / 2$.

For each point, make sure that it has a reflected point in the opposite side.

这道题给了我们一堆点，问我们存不存在一条平行于y轴的直线，使得所有的点关于该直线对称。题目中的提示给的相当充分，我们只要按照提示的步骤来做就可以解题了。首先我们找到所有点的横坐标的最大值和最小值，那么二者的平均值就是中间直线的横坐标，然后我们遍历每个点，如果都能找到直线对称的另一个点，则返回true，反之返回false，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool isReflected(vector<pair<int, int>>& points) {
4         unordered_map<int, set<int>> m;
5         int mx = INT_MIN, mn = INT_MAX;
6         for (auto a : points) {
7             mx = max(mx, a.first);
8             mn = min(mn, a.first);
9             m[a.first].insert(a.second);
10        }
11        double y = (double)(mx + mn) / 2;
12        for (auto a : points) {
13            int t = 2 * y - a.first;
14            if (!m.count(t) || !m[t].count(a.second)) {
15                return false;
16            }
17        }
18        return true;
19    }
20 };

```

CPP

下面这种解法没有求最大值和最小值，而是把所有的横坐标累加起来，然后求平均数，基本思路都相同，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool isReflected(vector<pair<int, int>>& points) {
4         if (points.empty()) return true;
5         set<pair<int, int>> pts;
6         double y = 0;
7         for (auto a : points) {
8             pts.insert(a);
9             y += a.first;
10        }
11        y /= points.size();
12        for (auto a : pts) {
13            if (!pts.count({y * 2 - a.first, a.second})) {
14                return false;
15            }
16        }
17        return true;
18    }
19 };

```

357. 计算各位不相同的数字个数

Given a non-negative integer n , count all numbers with unique digits, x , where $0 \leq x < 10^n$.

Example:

Given $n = 2$, return 91. (The answer should be the total numbers in the range of $0 \leq x < 100$, excluding [11,22,33,44,55,66,77,88,99])

Hint:

A direct way is to use the backtracking approach.

Backtracking should contains three states which are (the current number, number of steps to get that number and a bitmask which represent which number is marked as visited so far in the current number). Start with state $(0,0,0)$ and count all valid number till we reach number of steps equals to $10n$.

This problem can also be solved using a dynamic programming approach and some knowledge of combinatorics.

Let $f(k) =$ count of numbers with unique digits with length equals k .

$f(1) = 10, \dots, f(k) = 9 * 9 * 8 * \dots * (9 - k + 2)$ [The first factor is 9 because a number cannot start with 0].

这道题让我们找一个范围内的各位上不相同的数字，比如123就是各位不相同的数字，而11,121,222就不是这样的数字。那么我们根据提示中的最后一条可以知道，一位数的满足要求的数字是10个(0到9)，二位数的满足题意的是81个，[10 - 99]这90个数字中去掉[11,22,33,44,55,66,77,88,99]这9个数字，还剩81个。通项公式为 $f(k) = 9 * 9 * 8 * \dots * (9 - k + 2)$ ，那么我们就可以根据 n 的大小，把[1, n]区间位数通过通项公式算出来累加起来即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int countNumbersWithUniqueDigits(int n) {
4         if (n == 0) return 1;
5         int res = 0;
6         for (int i = 1; i <= n; ++i) {
7             res += count(i);
8         }
9         return res;
10    }
11    int count(int k) {
12        if (k < 1) return 0;
13        if (k == 1) return 10;
14        int res = 1;
15        for (int i = 9; i >= (11 - k); --i) {
16            res *= i;
17        }
18        return res * 9;
19    }
20 };

```

下面这种方法是上面方法的精简版，思路完全一样：

解法2：

```

1 class Solution {
2 public:
3     int countNumbersWithUniqueDigits(int n) {
4         if (n == 0) return 1;
5         int res = 10, cnt = 9;
6         for (int i = 2; i <= n; ++i) {
7             cnt *= (11 - i);
8             res += cnt;
9         }
10        return res;
11    }
12 };

```

最后我们来看题目提示中所说的回溯的方法，我们需要一个变量used，其二进制第*i*位为1表示数字*i*出现过，刚开始我们遍历1到9，对于每个遍历到的数字，现在used中标记已经出现过，然后在调用递归函数。在递归函数中，如果这个数字小于最大值，则结果res自增1，否则返回res。然后遍历0到9，如果当前数字没有在used中出现过，此时在used中标记，然后给当前数字乘以10加上*i*，再继续调用递归函数，这样我们可以遍历到所有的情况，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int countNumbersWithUniqueDigits(int n) {
4         int res = 1, max = pow(10, n), used = 0;
5         for (int i = 1; i < 10; ++i) {
6             used |= (1 << i);
7             res += search(i, max, used);
8             used &= ~(1 << i);
9         }
10        return res;
11    }
12    int search(int pre, int max, int used) {
13        int res = 0;
14        if (pre < max) ++res;
15        else return res;
16        for (int i = 0; i < 10; ++i) {
17            if (!(used & (1 << i))) {
18                used |= (1 << i);
19                int cur = 10 * pre + i;
20                res += search(cur, max, used);
21                used &= ~(1 << i);
22            }
23        }
24        return res;
25    }
26 };

```

358. 按距离为k隔离重排字符串

Given a non-empty string str and an integer k, rearrange the string such that the same characters are at least distance k from each other.

All input strings are given in lowercase letters. If it is not possible to rearrange the string, return an empty string "".

Example 1:

str = "aabbc", k = 3

Result: "abcabc"

The same letters are at least distance 3 from each other.

这道题给了我们一个字符串str，和一个整数k，让我们对字符串str重新排序，使得其中相同的字符之间的距离不小于k，这道题的难度标为Hard，看来不是省油的灯。的确，这道题的解法用到了哈希表，堆，和贪婪算法。这道题我最开始想的算法没有通过OJ的大集合超时了，下面的方法是参考网上大神的解法，发现十分的巧妙。我们需要一个哈希表来建立字符和其出现次数之间的映射，然后需要一个堆来保存这一堆映射，按照出现次数来排序。然后如果堆不为空我们就开始循环，我们找出k和str长度之间的较小值，然后从0遍历到这个较小值，对于每个遍历到的值，如果此时堆为空了，说明此位置没法填入字符了，返回空字符串，否则我们从堆顶取出一对映射，然后把字母加入结果res中，此时映射的个数减1，如果减1后的个数仍大于0，则我们将此映射加入临时集合v中，同时str的个数len减1，遍历完一次，我们把临时集合中的映射对由加入堆中，参见代码如下：

```

1 class Solution {
2 public:
3     string rearrangeString(string str, int k) {
4         if (k == 0) return str;
5         string res;
6         int len = (int)str.size();
7         unordered_map<char, int> m;
8         priority_queue<pair<int, char>> q;
9         for (auto a : str) ++m[a];
10        for (auto it = m.begin(); it != m.end(); ++it) {
11            q.push({it->second, it->first});
12        }
13        while (!q.empty()) {
14            vector<pair<int, int>> v;
15            int cnt = min(k, len);
16            for (int i = 0; i < cnt; ++i) {
17                if (q.empty()) return "";
18                auto t = q.top(); q.pop();
19                res.push_back(t.second);
20                if (--t.first > 0) v.push_back(t);
21                --len;
22            }
23            for (auto a : v) q.push(a);
24        }
25        return res;
26    }
27 };

```

359. 记录速率限制器

Design a logger system that receive stream of messages along with its timestamps, each message should be printed if and only if it is not printed in the last 10 seconds.

Given a message and a timestamp (in seconds granularity), return true if the message should be printed in the given timestamp, otherwise returns false.

It is possible that several messages arrive roughly at the same time.

Example:

```

Logger logger = new Logger();

// logging string "foo" at timestamp 1
logger.shouldPrintMessage(1, "foo"); returns true;

```

这道题让我们设计一个记录系统每次接受信息并保存时间戳，然后让我们打印出该消息，前提是最近10秒内没有打印出这个消息。这不是一道难题，我们可以用哈希表来做，建立消息和时间戳之间的映射，如果某个消息不再哈希表中，我们建立其和时间戳的映射，并返回true。如果已经在哈希表里了，我们看当前时间戳是否比哈希表中保存的时间戳大10，如果是，更新哈希表，并返回true，反之返回false，参见代码如下：

解法1：

```

1 class Logger {
2 public:
3     Logger() {}
4
5     bool shouldPrintMessage(int timestamp, string message) {
6         if (!m.count(message)) {
7             m[message] = timestamp;
8             return true;
9         }
10        if (timestamp - m[message] >= 10) {
11            m[message] = timestamp;
12            return true;
13        }
14        return false;
15    }
16
17 private:
18     unordered_map<string, int> m;
19 };

```

我们还可以写的更精简一些，如下所示：

解法2：

```

1 class Logger {
2 public:
3     Logger() {}
4
5     bool shouldPrintMessage(int timestamp, string message) {
6         if (timestamp < m[message]) return false;
7         m[message] = timestamp + 10;
8         return true;
9     }
10
11 private:
12     unordered_map<string, int> m;
13 };

```

360. 变换数组排序

Given a sorted array of integers nums and integer values a, b and c. Apply a function of the form $f(x) = ax^2 + bx + c$ to each element x in the array.

The returned array must be in sorted order.

Expected time complexity: $O(n)$

Example:

$\text{nums} = [-4, -2, 2, 4]$, $a = 1$, $b = 3$, $c = 5$,

Result: $[3, 9, 15, 33]$

$\text{nums} = [-4, -2, 2, 4]$, $a = -1$, $b = 3$, $c = 5$

Result: $[-23, -5, 1, 7]$

这道题给了我们一个数组，又给了我们一个抛物线的三个系数，让我们求带入抛物线方程后求出的数组成的有序数组。那么我们首先来看O(nlgn)的解法，这个解法没啥可说的，就是每个算出来再排序，这里我们用了最小堆来帮助我们排序，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<int> sortTransformedArray(vector<int>& nums, int a, int b, int c) {
4         vector<int> res;
5         priority_queue<int, vector<int>, greater<int>> q;
6         for (auto d : nums) {
7             q.push(a * d * d + b * d + c);
8         }
9         while (!q.empty()) {
10            res.push_back(q.top()); q.pop();
11        }
12        return res;
13    }
14 };

```

CPP

但是题目中的要求让我们在O(n)中实现，那么我们只能另辟蹊径。其实这道题用到了大量的高中所学的关于抛物线的数学知识，我们知道，对于一个方程 $f(x) = ax^2 + bx + c$ 来说，如果 $a > 0$ ，则抛物线开口朝上，那么两端的值比中间的大，而如果 $a < 0$ ，则抛物线开口朝下，则两端的值比中间的小。而当 $a=0$ 时，则为直线方法，是单调递增或递减的。那么我们可以利用这个性质来解题，题目中说明了给定数组nums是有序的，如果不是有序的，我想很难有O(n)的解法。正因为输入数组是有序的，我们可以根据a来分情况讨论：

当 $a > 0$ ，说明两端的值比中间的值大，那么此时我们从结果res后往前填数，用两个指针分别指向nums数组的开头和结尾，指向的两个数就是抛物线两端的数，将它们之中较大的数先存入res的末尾，然后指针向中间移，重复比较过程，直到把res都填满。

当 $a < 0$ ，说明两端的值比中间的小，那么我们从res的前面往后填，用两个指针分别指向nums数组的开头和结尾，指向的两个数就是抛物线两端的数，将它们之中较小的数先存入res的开头，然后指针向中间移，重复比较过程，直到把res都填满。

当 $a=0$ ，函数是单调递增或递减的，那么从前往后填和从后往前填都可以，我们可以将这种情况和 $a > 0$ 合并。

解法2：

```

1 class Solution {
2 public:
3     vector<int> sortTransformedArray(vector<int>& nums, int a, int b, int c) {
4         int n = nums.size(), i = 0, j = n - 1;
5         vector<int> res(n);
6         int idx = a >= 0 ? n - 1 : 0;
7         while (i <= j) {
8             if (a >= 0) {
9                 res[idx--] = cal(nums[i], a, b, c) >= cal(nums[j], a, b, c) ?
10                    cal(nums[i++], a, b, c) : cal(nums[j--], a, b, c);
11             } else {
12                 res[idx++] = cal(nums[i], a, b, c) >= cal(nums[j], a, b, c) ? cal(nums[j-
13                    ], a, b, c) : cal(nums[i++], a, b, c);
14             }
15         }
16         return res;
17     }
18     int cal(int x, int a, int b, int c) {
19         return a * x * x + b * x + c;
20     }
21 };

```

361. 炸弹人

Given a 2D grid, each cell is either a wall 'W', an enemy 'E' or empty '0' (the number zero), return the maximum enemies you can kill using one bomb.

The bomb kills all the enemies in the same row and column from the planted point until it hits the wall since the wall is too strong to be destroyed.

Note that you can only put the bomb at an empty cell.

Example:

For the given grid

```

0 E 0 0
E 0 W E
0 E 0 0

```

return 3. (Placing a bomb at (1,1) kills 3 enemies)

这道题相当于一个简单的炸弹人游戏，让我想起了小时候玩的红白机的炸弹人游戏，放一个炸弹，然后爆炸后会炸出个‘十’字，上下左右的东西都炸掉了。这道题是个简化版，字母E代表敌人，W代表墙壁，这里说明了炸弹无法炸穿墙壁。数字0表示可以放炸弹的位置，让我们找出一个放炸弹的位置可以炸死最多的敌人。那么我最开始想出的方法是建立四个累加数组v1, v2, v3, v4，其中v1是水平方向从左到右的累加数组，v2是水平方向从右到左的累加数组，v3是竖直方向从上到下的累加数组，v4是竖直方向从下到上的累加数组，我们建立好这个累加数组后，对于任意位置(i, j)，其可以炸死的最多敌人数就是v1[i][j] + v2[i][j] + v3[i][j] + v4[i][j]，最后我们通过比较每个位置的累加和，就可以得到结果，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int maxKilledEnemies(vector<vector<char>>& grid) {
4         if (grid.empty() || grid[0].empty()) return 0;
5         int m = grid.size(), n = grid[0].size(), res = 0;
6         vector<vector<int>> v1(m, vector<int>(n, 0)), v2 = v1, v3 = v1, v4 = v1;
7         for (int i = 0; i < m; ++i) {
8             for (int j = 0; j < n; ++j) {
9                 int t = (j == 0 || grid[i][j] == 'W') ? 0 : v1[i][j - 1];
10                v1[i][j] = grid[i][j] == 'E' ? t + 1 : t;
11            }
12            for (int j = n - 1; j >= 0; --j) {
13                int t = (j == n - 1 || grid[i][j] == 'W') ? 0 : v2[i][j + 1];
14                v2[i][j] = grid[i][j] == 'E' ? t + 1 : t;
15            }
16        }
17        for (int j = 0; j < n; ++j) {
18            for (int i = 0; i < m; ++i) {
19                int t = (i == 0 || grid[i][j] == 'W') ? 0 : v3[i - 1][j];
20                v3[i][j] = grid[i][j] == 'E' ? t + 1 : t;
21            }
22            for (int i = m - 1; i >= 0; --i) {
23                int t = (i == m - 1 || grid[i][j] == 'W') ? 0 : v4[i + 1][j];
24                v4[i][j] = grid[i][j] == 'E' ? t + 1 : t;
25            }
26        }
27        for (int i = 0; i < m; ++i) {
28            for (int j = 0; j < n; ++j) {
29                if (grid[i][j] == '0') {
30                    res = max(res, v1[i][j] + v2[i][j] + v3[i][j] + v4[i][j]);
31                }
32            }
33        }
34    }
35    return res;
36 }

```

我在论坛里看到了史蒂芬大神提出的另一种解法，感觉挺巧妙，就搬了过来。这种解法比较省空间，写法也比较简洁，需要一个rowCnt变量，用来记录到下一个墙之前的敌人个数。还需要一个数组colCnt，其中colCnt[j]表示第j列到下一个墙之前的敌人个数。算法思路是遍历整个数组grid，对于一个位置grid[i][j]，对于水平方向，如果当前位置是开头一个或者前面一个是墙壁，我们开始从当前位置往后遍历，遍历到末尾或者墙的位置停止，计算敌人个数。对于竖直方向也是同样，如果当前位置是开头一个或者上面一个是墙壁，我们开始从当前位置向下遍历，遍历到末尾或者墙的位置停止，计算敌人个数。可能会有人有疑问，为啥rowCnt就可以用一个变量，而colCnt就需要用一个数组呢，为啥colCnt不能也用一个变量呢？原因是由于我们的遍历顺序决定的，我们是逐行遍历的，在每行的开头就统计了该行的敌人总数，所以再该行遍历没必要用数组，但是每次移动时就会换到不同的列，我们总不能没换个列就重新统计一遍吧，所以就在第一行时一起统计了存到数组中供后来使用。有了水平方向和竖直方向敌人的个数，那么如果当前位置是0，表示可以放炸弹，我们更新结果res即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int maxKilledEnemies(vector<vector<char>>& grid) {
4         if (grid.empty() || grid[0].empty()) return 0;
5         int m = grid.size(), n = grid[0].size(), res = 0, rowCnt, colCnt[n];
6         for (int i = 0; i < m; ++i) {
7             for (int j = 0; j < n; ++j) {
8                 if (j == 0 || grid[i][j - 1] == 'W') {
9                     rowCnt = 0;
10                for (int k = j; k < n && grid[i][k] != 'W'; ++k) {
11                    rowCnt += grid[i][k] == 'E';
12                }
13            }
14            if (i == 0 || grid[i - 1][j] == 'W') {
15                colCnt[j] = 0;
16                for (int k = i; k < m && grid[k][j] != 'W'; ++k) {
17                    colCnt[j] += grid[k][j] == 'E';
18                }
19            }
20            if (grid[i][j] == '0') {
21                res = max(res, rowCnt + colCnt[j]);
22            }
23        }
24    }
25    return res;
26 }
27 };

```

362. 设计点击计数器

Design a hit counter which counts the number of hits received in the past 5 minutes.

Each function accepts a timestamp parameter (in seconds granularity) and you may assume that calls are being made to the system in chronological order (ie, the timestamp is monotonically increasing). You may assume that the earliest timestamp starts at 1.

It is possible that several hits arrive roughly at the same time.

Example:

```

HitCounter counter = new HitCounter();

// hit at timestamp 1.
counter.hit(1);

// hit at timestamp 2.
counter.hit(2);

```

这道题让我们设计一个点击计数器，能够返回五分钟内的点击数，提示了有可能同一时间内有多个点击。由于操作都是按时间顺序的，下一次的时间戳都会大于等于本次的时间戳，那么最直接的方法就是用一个队列queue，每次点击时都将当前时间戳加入queue中，然后在需要获取点击数时，我们从队列开头开始看，如果开头的时间戳在5分钟以外了，就删掉，直到开头的时间戳在5分钟以内为止，然后返回queue的元素个数即为所求的点击数，参见代码如下：

解法1：

```

1 class HitCounter {
2 public:
3     /** Initialize your data structure here. */
4     HitCounter() {}
5
6     /** Record a hit.
7         @param timestamp - The current timestamp (in seconds granularity). */
8     void hit(int timestamp) {
9         q.push(timestamp);
10    }
11
12    /** Return the number of hits in the past 5 minutes.
13        @param timestamp - The current timestamp (in seconds granularity). */
14    int getHits(int timestamp) {
15        while (!q.empty() && timestamp - q.front() >= 300) {
16            q.pop();
17        }
18        return q.size();
19    }
20
21 private:
22     queue<int> q;
23 };

```

下面这种方法和上面的方法很像，用了一个数组保存所有的时间戳，然后要返回点击数时，只需要从开头找到第一个在5分钟的时间戳的坐标，然后用数组总长度减去这个坐标即可。和上面的方法不同的是，这个方法不删掉之前的时间戳，缺点是会很占空间，而且越到后面效率越低，参见代码如下：

解法2：

```

1 class HitCounter {
2 public:
3     /** Initialize your data structure here. */
4     HitCounter() {}
5
6     /** Record a hit.
7         @param timestamp - The current timestamp (in seconds granularity). */
8     void hit(int timestamp) {
9         v.push_back(timestamp);
10    }
11
12    /** Return the number of hits in the past 5 minutes.
13        @param timestamp - The current timestamp (in seconds granularity). */
14    int getHits(int timestamp) {
15        int i, j;
16        for (i = 0; i < v.size(); ++i) {
17            if (v[i] > timestamp - 300) {
18                break;
19            }
20        }
21        return v.size() - i;
22    }
23
24 private:
25     vector<int> v;
26 };

```

由于Follow up中说每秒中会有很多点击，下面这种方法就比较巧妙了，定义了两个大小为300的一维数组times和hits，分别用来保存时间戳和点击数，在点击函数中，将时间戳对300取余，然后看此位置中之前保存的时间戳和当前的时间戳是否一样，一样说明是同一个时间戳，那么对应的点击数自增1，如果不样，说明已经过了五分钟了，那么将对应的点击数重置为1。那么在返回点击数时，我们需要遍历times数组，找出所有在5分中内的位置，然后把hits中对应位置的点击数都加起来即可，参见代码如下：

解法3：

CPP

```

1 class HitCounter {
2 public:
3     /** Initialize your data structure here. */
4     HitCounter() {
5         times.resize(300);
6         hits.resize(300);
7     }
8
9     /** Record a hit.
10      @param timestamp - The current timestamp (in seconds granularity). */
11     void hit(int timestamp) {
12         int idx = timestamp % 300;
13         if (times[idx] != timestamp) {
14             times[idx] = timestamp;
15             hits[idx] = 1;
16         } else {
17             ++hits[idx];
18         }
19     }
20
21     /** Return the number of hits in the past 5 minutes.
22      @param timestamp - The current timestamp (in seconds granularity). */
23     int getHits(int timestamp) {
24         int res = 0;
25         for (int i = 0; i < 300; ++i) {
26             if (timestamp - times[i] < 300) {
27                 res += hits[i];
28             }
29         }
30         return res;
31     }
32
33 private:
34     vector<int> times, hits;
35 };

```

363. 最大矩阵和不超过K

Given a non-empty 2D matrix `matrix` and an integer `k`, find the max sum of a rectangle in the matrix such that its sum is no larger than `k`.

Example:

```
Given matrix = [
    [1, 0, 1],
    [0, -2, 3]
]
k = 2
```

The answer is 2. Because the sum of rectangle $[[0, 1], [-2, 3]]$ is 2 and 2 is the max number no larger than `k` (`k = 2`).

Note:

The rectangle inside the matrix must have an area > 0 .

What if the number of rows is much larger than the number of columns?

这道题给了我们一个二维数组，让我们求和不超过的K的最大子矩形，那么我们首先可以考虑使用brute force来解，就是遍历所有的子矩形，然后计算其和跟K比较，找出不超过K的最大值即可。就算是暴力搜索，我们也可以使用优化的算法，比如建立累加和，参见之前那道题Range Sum Query 2D - Immutable，我们可以快速求出任何一个区间和，那么下面的方法就是这样的，当遍历到(i, j)时，我们计算sum(i, j)，表示矩形(0, 0)到(i, j)的和，然后我们遍历这个矩形中所有的子矩形，计算其和跟K相比，这样既可遍历到原矩形的所有子矩形，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     int maxSumSubmatrix(vector<vector<int>>& matrix, int k) {
4         if (matrix.empty() || matrix[0].empty()) return 0;
5         int m = matrix.size(), n = matrix[0].size(), res = INT_MIN;
6         int sum[m][n];
7         for (int i = 0; i < m; ++i) {
8             for (int j = 0; j < n; ++j) {
9                 int t = matrix[i][j];
10                if (i > 0) t += sum[i - 1][j];
11                if (j > 0) t += sum[i][j - 1];
12                if (i > 0 && j > 0) t -= sum[i - 1][j - 1];
13                sum[i][j] = t;
14                for (int r = 0; r <= i; ++r) {
15                    for (int c = 0; c <= j; ++c) {
16                        int d = sum[i][j];
17                        if (r > 0) d -= sum[r - 1][j];
18                        if (c > 0) d -= sum[i][c - 1];
19                        if (r > 0 && c > 0) d += sum[r - 1][c - 1];
20                        if (d <= k) res = max(res, d);
21                    }
22                }
23            }
24        }
25        return res;
26    }
27};
```

CPP

下面这个算法进一步的优化了运行时间，这个算法是基于计算二维数组中最大子矩阵和的算法，可以参见youtube上的这个视频 Maximum Sum Rectangular Submatrix in Matrix dynamic programming/2D kadane。这个算法巧妙在把二维数组按行或列拆成多个一维数组，然后利用一维数组的累加和来找符合要求的数字，这里用了lower_bound来加快我们的搜索速度，也可以使用二分搜索法来替代。我们建立一个集合set，然后开始先放个0进去，为啥要放0呢，因为我们要找lower_bound(curSum - k)，当curSum和k相等时，0就可以被返回了，这样我们就能更新结果了。由于我们对于一维数组建立了累积和，那么sum[i,j] = sum[i] - sum[j]，其中sums[i,j]就是目标子数组需要其和小于等于k，然后sums[j]是curSum，而sum[i]就是我们要找值，当我们使用二分搜索法找sum[i]时，sum[i]的和需要 \geq sum[j] - k，所以也可以使用lower_bound来找，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int maxSumSubmatrix(vector<vector<int>>& matrix, int k) {
4         if (matrix.empty() || matrix[0].empty()) return 0;
5         int m = matrix.size(), n = matrix[0].size(), res = INT_MIN;
6         for (int i = 0; i < n; ++i) {
7             vector<int> sum(m, 0);
8             for (int j = i; j < n; ++j) {
9                 for (int k = 0; k < m; ++k) {
10                     sum[k] += matrix[k][j];
11                 }
12                 int curSum = 0, curMax = INT_MIN;
13                 set<int> s;
14                 s.insert(0);
15                 for (auto a : sum) {
16                     curSum += a;
17                     auto it = s.lower_bound(curSum - k);
18                     if (it != s.end()) curMax = max(curMax, curSum - *it);
19                     s.insert(curSum);
20                 }
21                 res = max(res, curMax);
22             }
23         }
24         return res;
25     }
26 };

```

364. 嵌套链表权重之二

Given a nested list of integers, return the sum of all integers in the list weighted by their depth.

Each element is either an integer, or a list -- whose elements may also be integers or other lists.

Different from the previous question where weight is increasing from root to leaf, now the weight is defined from bottom up. i.e., the leaf level integers have weight 1, and the root level integers have the largest weight.

Example 1:

Given the list [[1,1],2,[1,1]], return 8. (four 1's at depth 1, one 2 at depth 2)

这道题是之前那道Nested List Weight Sum的拓展，与其不同的是，这道题的深度越深，权重越小，和之前刚好相反。但是解题思路没有变，还可以用DFS来做，那么由于遍历的时候不知道最终的depth有多深，则不能遍历的时候就直接累加结果，我最开始的想法是在遍历的过程中建立一个二维数组，把每层的数字都保存起来，然后最后知道了depth后，再来计算权重和，比如题目中给的两个例子，建立的二维数组分别为：

[[1,1],2,[1,1]]:

```
1 1 1 1
2
```

[1,[4,[6]]]:

```
1
4
6
```

这样我们就能算出权重和了，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     int depthSumInverse(vector<NestedInteger>& nestedList) {
4         int res = 0;
5         vector<vector<int>> v;
6         for (auto a : nestedList) {
7             helper(a, 0, v);
8         }
9         for (int i = v.size() - 1; i >= 0; --i) {
10            for (int j = 0; j < v[i].size(); ++j) {
11                res += v[i][j] * (v.size() - i);
12            }
13        }
14        return res;
15    }
16    void helper(NestedInteger &ni, int depth, vector<vector<int>> &v) {
17        vector<int> t;
18        if (depth < v.size()) t = v[depth];
19        else v.push_back(t);
20        if (ni.isInteger()) {
21            t.push_back(ni.getInteger());
22            if (depth < v.size()) v[depth] = t;
23            else v.push_back(t);
24        } else {
25            for (auto a : ni.getList()) {
26                helper(a, depth + 1, v);
27            }
28        }
29    }
30};
```

CPP

其实上面的方法可以简化，由于每一层的数字不用分别保存，每个数字分别乘以深度再相加，跟每层数字先相加起来再乘以深度是一样的，这样我们只需要一个一维数组就可以了，只要把各层的数字和保存起来，最后再计算权重和即可：

解法2：

```

1 class Solution {
2 public:
3     int depthSumInverse(vector<NestedInteger>& nestedList) {
4         int res = 0;
5         vector<int> v;
6         for (auto a : nestedList) {
7             helper(a, 0, v);
8         }
9         for (int i = v.size() - 1; i >= 0; --i) {
10            res += v[i] * (v.size() - i);
11        }
12        return res;
13    }
14    void helper(NestedInteger ni, int depth, vector<int> &v) {
15        if (depth >= v.size()) v.resize(depth + 1);
16        if (ni.isInteger()) {
17            v[depth] += ni.getInteger();
18        } else {
19            for (auto a : ni.getList()) {
20                helper(a, depth + 1, v);
21            }
22        }
23    }
24};

```

下面这个方法就比较巧妙了，由史蒂芬大神提出来的，这个方法用了两个变量unweighted和weighted，非权重和跟权重和，初始化均为0，然后如果nestedList不为空开始循环，先声明一个空数组nextLevel，遍历nestedList中的元素，如果是数字，则非权重和加上这个数字，如果是数组，就加入nextLevel，这样遍历完成后，第一层的数字和保存在非权重和unweighted中了，其余元素都存入了nextLevel中，此时我们将unweighted加到weighted中，将nextLevel赋给nestedList，这样再进入下一层计算，由于上一层的值还在unweighted中，所以第二层计算完将unweighted加入weighted中时，相当于第一层的数字和被加了两次，这样就完美的符合要求了，这个思路又巧妙又牛B，大神就是大神啊，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int depthSumInverse(vector<NestedInteger>& nestedList) {
4         int unweighted = 0, weighted = 0;
5         while (!nestedList.empty()) {
6             vector<NestedInteger> nextLevel;
7             for (auto a : nestedList) {
8                 if (a.isInteger()) {
9                     unweighted += a.getInteger();
10                } else {
11                    nextLevel.insert(nextLevel.end(), a.getList().begin(),
12 a.getList().end());
13                }
14            }
15            weighted += unweighted;
16            nestedList = nextLevel;
17        }
18        return weighted;
19    }
20};

```

下面这种算法是常规的BFS解法，利用上面的建立两个变量unweighted和weighted的思路，大体上没什么区别：

解法4：

CPP

```

1 class Solution {
2 public:
3     int depthSumInverse(vector<NestedInteger>& nestedList) {
4         int unweighted = 0, weighted = 0;
5         queue<vector<NestedInteger>> q;
6         q.push(nestedList);
7         while (!q.empty()) {
8             int size = q.size();
9             for (int i = 0; i < size; ++i) {
10                 vector<NestedInteger> t = q.front(); q.pop();
11                 for (auto a : t) {
12                     if (a.isInteger()) unweighted += a.getInteger();
13                     else if (!a.getList().empty()) q.push(a.getList());
14                 }
15             }
16             weighted += unweighted;
17         }
18         return weighted;
19     }
20 };

```

365. 水罐問題

You are given two jugs with capacities x and y litres. There is an infinite amount of water supply available. You need to determine whether it is possible to measure exactly z litres using these two jugs.

If z liters of water is measurable, you must have z liters of water contained within one or both buckets by the end.

Operations allowed:

Fill any of the jugs completely with water.

Empty any of the jugs.

Pour water from one jug into another till the other jug is completely full or the first jug itself is empty.

Example 1: (From the famous "Die Hard" example)

Input: $x = 3$, $y = 5$, $z = 4$

Output: True

这是一道脑筋急转弯题，我想很多人以前应该听过这道题目，有一个容量为3升和一个容量为5升的水罐，问我们如何准确的称出4升的水。我想很多人都知道怎么做，先把5升水罐装满水，倒到3升水罐里，这时5升水罐里还有2升水，然后把3升水罐里的水都倒掉，把5升水罐中的2升水倒入3升水罐中，这时候把5升水罐解满，然后往此时有2升水的3升水罐里倒水，这样5升水罐倒出1升后还剩4升即为所求。这个很多人都知道，但是这道题随意给我们了三个参数，问有没有解法，这就比较难了。这里我就照搬网上大神的讲解吧：

这道问题其实可以转换为有一个很大的容器，我们有两个杯子，容量分别为x和y，问我们通过用两个杯子往里倒水，和往外舀水，问能不能使容器中的水刚好为z升。那么我们可以用一个公式来表达：

$$z = m * x + n * y$$

其中m, n为舀水和倒水的次数，正数表示往里舀水，负数表示往外倒水，那么题目中的例子可以写成： $4 = (-2) * 3 + 2 * 5$ ，即3升的水罐往外倒了两次水，5升水罐往里舀了两次水。那么问题就变成了对于任意给定的x,y,z，是否存在m和n使得上面的等式成立。根据裴蜀定理， $ax + by = d$ 的解为 $d = \text{gcd}(x, y)$ ，那么我们只要只要 $z \% d == 0$ ，上面的等式就有解，所以问题就迎刃而解了，我们只要看z是不是x和y的最大公约数的倍数就行了，别忘了还有个限制条件 $x + y >= z$ ，因为x和y不可能称出比它们之和还多的水，参见代码如下：

```
1 class Solution {
2 public:
3     bool canMeasureWater(int x, int y, int z) {
4         return z == 0 || (x + y >= z && z % gcd(x, y) == 0);
5     }
6     int gcd(int x, int y) {
7         return y == 0 ? x : gcd(y, x % y);
8     }
9 }
```

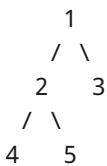
CPP

366. 找二叉树的叶节点

Given a binary tree, find all leaves and then remove those leaves. Then repeat the previous steps until the tree is empty.

Example:

Given binary tree



Returns [4, 5, 3], [2], [1].

这道题给了我们一个二叉树，让我们返回其每层的叶节点，就像剥洋葱一样，将这个二叉树一层一层剥掉，最后一个剥掉根节点。那么题目中提示说要用DFS来做，思路是这样的，每一个节点从左子节点和右子节点分开走可以得到两个深度，由于成为叶节点的条件是左右子节点都为空，所以我们取左右子节点中较大值加1为当前节点的深度值，知道了深度值就可以将节点值加入到结果res中的正确位置了，求深度的方法我们可以参见Maximum Depth of Binary Tree中求最大深度的方法，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<vector<int>> findLeaves(TreeNode* root) {
4         vector<vector<int>> res;
5         helper(root, res);
6         return res;
7     }
8     int helper(TreeNode *root, vector<vector<int>> &res) {
9         if (!root) return -1;
10        int depth = 1 + max(helper(root->left, res), helper(root->right, res));
11        if (depth >= res.size()) res.resize(depth + 1);
12        res[depth].push_back(root->val);
13        return depth;
14    }
15 };

```

下面这种DFS方法没有用计算深度的方法，而是使用了一层层剥离的方法，思路是遍历二叉树，找到叶节点，将其赋值为NULL，然后加入leaves数组中，这样一层层剥洋葱般的就可以得到最终结果了：

解法2：

```

1 class Solution {
2 public:
3     vector<vector<int>> findLeaves(TreeNode* root) {
4         vector<vector<int>> res;
5         while (root) {
6             vector<int> leaves;
7             root = remove(root, leaves);
8             res.push_back(leaves);
9         }
10        return res;
11    }
12    TreeNode* remove(TreeNode *node, vector<int> &leaves) {
13        if (!node) return NULL;
14        if (!node->left && !node->right) {
15            leaves.push_back(node->val);
16            return NULL;
17        }
18        node->left = remove(node->left, leaves);
19        node->right = remove(node->right, leaves);
20        return node;
21    }
22 };

```

367. 检验完全平方数

Given a positive integer num, write a function which returns True if num is a perfect square else False.

Note: Do not use any built-in library function such as sqrt.

Example 1:

Input: 16

Returns: True

这道题给了我们一个数，让我们判断其是否为完全平方数，那么显而易见的是，肯定不能使用brute force，这样太不高效了，那么最小是能以指数的速度来缩小范围，那么我最先想出的方法是这样的，比如一个数字49，我们先对其除以2，得到24，发现24的平方大于49，那么再对24除以2，得到12，发现12的平方还是大于49，再对12除以2，得到6，发现6的平方小于49，于是遍历6到12中的所有数，看有没有平方等于49的，有就返回true，没有就返回false，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     bool isPerfectSquare(int num) {
4         if (num == 1) return true;
5         long x = num / 2, t = x * x;
6         while (t > num) {
7             x /= 2;
8             t = x * x;
9         }
10        for (int i = x; i <= 2 * x; ++i) {
11            if (i * i == num) return true;
12        }
13        return false;
14    }
15};
```

CPP

下面这种方法也比较高效，从1搜索到sqrt(num)，看有没有平方正好等于num的数：

解法2：

```
1 class Solution {
2 public:
3     bool isPerfectSquare(int num) {
4         for (int i = 1; i <= num / i; ++i) {
5             if (i * i == num) return true;
6         }
7         return false;
8     }
9};
```

CPP

我们也可以使用二分查找法来做，要查找的数为mid*mid，参见代码如下：

解法3：

```
1 class Solution {
2 public:
3     bool isPerfectSquare(int num) {
4         long left = 0, right = num;
5         while (left <= right) {
6             long mid = left + (right - left) / 2, t = mid * mid;
7             if (t == num) return true;
8             else if (t < num) left = mid + 1;
9             else right = mid - 1;
10        }
11        return false;
12    }
13};
```

CPP

下面这种方法就是纯数学解法了，利用到了这样一条性质，完全平方数是一系列奇数之和，例如：

```

1 = 1
4 = 1 + 3
9 = 1 + 3 + 5
16 = 1 + 3 + 5 + 7
25 = 1 + 3 + 5 + 7 + 9
36 = 1 + 3 + 5 + 7 + 9 + 11
...
1+3+...+(2n-1) = (2n-1 + 1)n/2 = n*n

```

这里就不做证明了，我也不再证明，知道了这条性质，就可以利用其来解题了，时间复杂度为O(sqrt(n))。

解法4：

```

1 class Solution {
2 public:
3     bool isPerfectSquare(int num) {
4         int i = 1;
5         while (num > 0) {
6             num -= i;
7             i += 2;
8         }
9         return num == 0;
10    }
11 };

```

CPP

下面这种方法是第一种方法的类似方法，更加精简了，时间复杂度为O(lgn)：

解法5：

```

1 class Solution {
2 public:
3     bool isPerfectSquare(int num) {
4         long x = num;
5         while (x * x > num) {
6             x = (x + num / x) / 2;
7         }
8         return x * x == num;
9     }
10 };

```

CPP

这道题其实还有O(1)的解法，这你敢信？简直太丧心病狂了，详情请参见论坛上的这个帖子。

[368. 最大可整除的子集合](#)

Given a set of distinct positive integers, find the largest subset such that every pair (S_i , S_j) of elements in this subset satisfies: $S_i \% S_j = 0$ or $S_j \% S_i = 0$.

If there are multiple solutions, return any subset is fine.

Example 1:

nums: [1,2,3]

Result: [1,2] (of course, [1,3] will also be ok)

这道题给了我们一个数组，让我们求这样一个子集合，集合中的任意两个数相互取余均为0，而且提示中说明了要使用DP来解。那么我们考虑，较小数对较大数取余一定不为0，那么问题就变成了看较大数能不能整除这个较小数。那么如果数组是无序的，处理起来就比较麻烦，所以我们首先可以先给数组排序，这样我们每次就只要看后面的数字能否整除前面的数字。定义一个动态数组dp，其中dp[i]表示到数字nums[i]位置最大可整除的子集合的长度，还需要一个一维数组parent，来保存上一个能整除的数字的位置，两个整型变量mx和mx_idx分别表示最大子集合的长度和起始数字的位置，我们可以从后往前遍历数组，对于某个数字再遍历到末尾，在这个过程中，如果nums[j]能整除nums[i]，且 $dp[i] < dp[j] + 1$ 的话，更新 $dp[i]$ 和 $parent[i]$ ，如果 $dp[i]$ 大于mx了，还要更新mx和mx_idx，最后循环结束后，我们来填res数字，根据parent数组来找到每一个数字，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<int> largestDivisibleSubset(vector<int>& nums) {
4         sort(nums.begin(), nums.end());
5         vector<int> dp(nums.size(), 0), parent(nums.size(), 0), res;
6         int mx = 0, mx_idx = 0;
7         for (int i = nums.size() - 1; i >= 0; --i) {
8             for (int j = i; j < nums.size(); ++j) {
9                 if (nums[j] % nums[i] == 0 && dp[i] < dp[j] + 1) {
10                     dp[i] = dp[j] + 1;
11                     parent[i] = j;
12                     if (mx < dp[i]) {
13                         mx = dp[i];
14                         mx_idx = i;
15                     }
16                 }
17             }
18         }
19         for (int i = 0; i < mx; ++i) {
20             res.push_back(nums[mx_idx]);
21             mx_idx = parent[mx_idx];
22         }
23         return res;
24     }
25 };

```

CPP

下面这种方法和上面解法的思路基本一样，只不过dp数组现在每一项保存一个pair，相当于上面解法中的dp和parent数组揉到一起表示了，然后的不同就是下面的方法是从前往后遍历的，每个数字又要遍历到开头，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<int> largestDivisibleSubset(vector<int>& nums) {
4         sort(nums.begin(), nums.end());
5         vector<int> res;
6         vector<pair<int, int>> dp(nums.size());
7         int mx = 0, mx_idx = 0;
8         for (int i = 0; i < nums.size(); ++i) {
9             for (int j = i; j >= 0; --j) {
10                 if (nums[i] % nums[j] == 0 && dp[i].first < dp[j].first + 1) {
11                     dp[i].first = dp[j].first + 1;
12                     dp[i].second = j;
13                     if (mx < dp[i].first) {
14                         mx = dp[i].first;
15                         mx_idx = i;
16                     }
17                 }
18             }
19         }
20         for (int i = 0; i < mx; ++i) {
21             res.push_back(nums[mx_idx]);
22             mx_idx = dp[mx_idx].second;
23         }
24         return res;
25     }
26 };

```

369. 链表加一运算

Given a non-negative number represented as a singly linked list of digits, plus one to the number.

The digits are stored such that the most significant digit is at the head of the list.

Example:

Input:

1->2->3

Output:

1->2->4

这道题给了我们一个链表，用来模拟一个三位数，表头是高位，现在让我们进行加1运算，这道题的难点在于链表无法通过坐标来访问元素，只能通过遍历的方式进行，而这题刚好让我们从链尾开始操作，从后往前，遇到进位也要正确的处理，最后还有可能要在开头补上一位。那么我们反过来想，如果链尾是高位，那么进行加1运算就方便多了，直接就可以边遍历边进行运算处理，那么我们可以做的就是先把链表翻转一下，然后现在就是链尾是高位了，我们进行加1处理运算结束后，再把链表翻转回来即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     ListNode* plusOne(ListNode* head) {
4         if (!head) return head;
5         ListNode *rev_head = reverse(head), *cur = rev_head, *pre = cur;
6         int carry = 1;
7         while (cur) {
8             pre = cur;
9             int t = cur->val + carry;
10            cur->val = t % 10;
11            carry = t / 10;
12            if (carry == 0) break;
13            cur = cur->next;
14        }
15        if (carry) pre->next = new ListNode(1);
16        return reverse(rev_head);
17    }
18    ListNode* reverse(ListNode *head) {
19        if (!head) return head;
20        ListNode *dummy = new ListNode(-1), *cur = head;
21        dummy->next = head;
22        while (cur->next) {
23            ListNode *t = cur->next;
24            cur->next = t->next;
25            t->next = dummy->next;
26            dummy->next = t;
27        }
28        return dummy->next;
29    }
30};

```

我们也可以通过递归来实现，这样我们就不用翻转链表了，通过递归一层一层的调用，最先处理的是链尾元素，我们将其加1，然后看是否有进位，返回进位，然后回溯到表头，加完进位，如果发现又产生了新的进位，那么我们在最开头加上一个新节点即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     ListNode* plusOne(ListNode* head) {
4         if (!head) return head;
5         int carry = helper(head);
6         if (carry == 1) {
7             ListNode *res = new ListNode(1);
8             res->next = head;
9             return res;
10        }
11        return head;
12    }
13    int helper(ListNode *node) {
14        if (!node) return 1;
15        int carry = helper(node->next);
16        int sum = node->val + carry;
17        node->val = sum % 10;
18        return sum / 10;
19    }
20};

```

下面这种方法比较巧妙了，思路是遍历链表，找到右起第一个不为9的数字，如果找不到这样的数字，说明所有数字均为9，那么在表头新建一个值为0的新节点，进行加1处理，然后把右边所有的数字都置为0即可。举例来说：

比如1->2->3，那么第一个不为9的数字为3，对3进行加1，变成4，右边没有节点了，所以不做处理，返回1->2->4。

再比如说8->9->9，找第一个不为9的数字为8，进行加1处理变成了9，然后把后面的数字都置0，得到结果9->0->0。

再来看9->9->9的情况，找不到不为9的数字，那么再前面新建一个值为0的节点，进行加1处理变成了1，把后面的数字都置0，得到1->0->0->0。

解法3：

```

1 class Solution {
2 public:
3     ListNode* plusOne(ListNode* head) {
4         ListNode *cur = head, *right = NULL;
5         while (cur) {
6             if (cur->val != 9) right = cur;
7             cur = cur->next;
8         }
9         if (!right) {
10            right = new ListNode(0);
11            right->next = head;
12            head = right;
13        }
14        ++right->val;
15        cur = right->next;
16        while (cur) {
17            cur->val = 0;
18            cur = cur->next;
19        }
20        return head;
21    }
22 };

```

CPP

最后这种解法是解法二的迭代写法，我们用到栈，利用栈的先进后出机制，就可以实现从后往前的处理节点，参见代码如下：

解法4：

```
1 class Solution {
2 public:
3     ListNode* plusOne(ListNode* head) {
4         stack<ListNode*> s;
5         ListNode *cur = head;
6         while (cur) {
7             s.push(cur);
8             cur = cur->next;
9         }
10        int carry = 1;
11        while (!s.empty() && carry) {
12            ListNode *t = s.top(); s.pop();
13            int sum = t->val + carry;
14            t->val = sum % 10;
15            carry = sum / 10;
16        }
17        if (carry) {
18            ListNode *new_head = new ListNode(1);
19            new_head->next = head;
20            head = new_head;
21        }
22        return head;
23    }
24};
```

370. 范围相加

Assume you have an array of length n initialized with all 0's and are given k update operations.

Each operation is represented as a triplet: [startIndex, endIndex, inc] which increments each element of subarray A[startIndex ... endIndex] (startIndex and endIndex inclusive) with inc.

Return the modified array after all k operations were executed.

Example:

Given:

```
length = 5,
updates = [
    [1, 3, 2],
    [2, 4, 3],
    [0, 2, -2]
]
```

Output:

`[-2, 0, 3, 5, 3]`

Explanation:

Initial state:

`[0, 0, 0, 0, 0]`

After applying operation [1, 3, 2]:

`[0, 2, 2, 2, 0]`

After applying operation [2, 4, 3]:

`[0, 2, 5, 5, 3]`

After applying operation [0, 2, -2]:

`[-2, 0, 3, 5, 3]`

Hint:

Thinking of using advanced data structures? You are thinking it too complicated.

For each update operation, do you really need to update all elements between i and j?

Update only the first and end element is sufficient.

The optimal time complexity is O(k + n) and uses O(1) extra space.

这道题刚添加的时候我就看到了，当时只有1个提交，0个接受，于是我赶紧做，提交成功后发现我是第一个提交成功的，哈哈，头一次做沙发啊，有点小激动~这道题的提示说了我们肯定不能把范围内的所有数字都更新，而是只更新开头结尾两个数字就行了，那么我们的做法就是在开头坐标startIndex位置加上inc，而在结束位置加1的地方加上-inc，那么根据题目中的例子，我们可以得到一个数组，`nums = {-2, 2, 3, 2, -2, -3}`，然后我们发现对其做累加和就是我们要求的结果`result = {-2, 0, 3, 5, 3}`，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<int> getModifiedArray(int length, vector<vector<int>>& updates) {
4         vector<int> res, nums(length + 1, 0);
5         for (int i = 0; i < updates.size(); ++i) {
6             nums[updates[i][0]] += updates[i][2];
7             nums[updates[i][1] + 1] -= updates[i][2];
8         }
9         int sum = 0;
10        for (int i = 0; i < length; ++i) {
11            sum += nums[i];
12            res.push_back(sum);
13        }
14        return res;
15    }
16 };

```

我们可以在空间上稍稍优化下上面的代码，用res来代替nums，最后把res中最后一个数字去掉即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<int> getModifiedArray(int length, vector<vector<int>>& updates) {
4         vector<int> res(length + 1);
5         for (auto a : updates) {
6             res[a[0]] += a[2];
7             res[a[1] + 1] -= a[2];
8         }
9         for (int i = 1; i < res.size(); ++i) {
10            res[i] += res[i - 1];
11        }
12        res.pop_back();
13        return res;
14    }
15 }

```

371. 两数之和

Calculate the sum of two integers a and b, but you are not allowed to use the operator + and -.

Example:

Given a = 1 and b = 2, return 3.

这道题是CareerCup上的一道原题，难道现在LeetCode的新题都是到处抄来的么，讲解可以参见我之前的博客18.1 Add Two Numbers。简而言之就是用异或算不带进位的和，用与并左移1位来算进位，然后把两者加起来即可，先来看递归的写法如下：

解法1：

```

1 class Solution {
2 public:
3     int getSum(int a, int b) {
4         if (b == 0) return a;
5         int sum = a ^ b;
6         int carry = (a & b) << 1;
7         return getSum(sum, carry);
8     }
9 };

```

上面的解法可以精简到一行，哈哈，叼不叼？

解法2：

```

1 class Solution {
2 public:
3     int getSum(int a, int b) {
4         return b == 0 ? a : getSum(a ^ b, (a & b) << 1);
5     }
6 };

```

也可以写成迭代的样子，思路都是一样的~

解法3：

```

1 class Solution {
2 public:
3     int getSum(int a, int b) {
4         while (b) {
5             int carry = (a & b) << 1;
6             a = a ^ b;
7             b = carry;
8         }
9         return a;
10    }
11 };

```

372. 超级次方

Your task is to calculate $a^b \bmod 1337$ where a is a positive integer and b is an extremely large positive integer given in the form of an array.

Example1:

```

a = 2
b = [3]

```

Result: 8

这道题让我们求一个数的很大的次方对1337取余的值，开始一直在想这个1337有什么玄机，为啥突然给这么一个数，感觉很突兀，后来想来想去也没想出来为啥，估计就是怕结果太大无法表示，随便找个数取余吧。那么这道题和之前那道Pow(x, n)的解法很类似，我们都得对半缩小，不同的是后面都要加上对1337取余。由于给定的指数b是一个一维数组的表示方法，我们要折半缩小处理起来肯定十分不方便，所以我们采用按位来处理，比如 $2^{23} = (2^2)^{10} * 2^3$ ，所以我们可以从b的最高位开始，算出个结果存入res，然后到下一位是，res的十次方再乘以a的该位次方再对1337取余，参见代码如下：

```

1 class Solution {
2 public:
3     int superPow(int a, vector<int>& b) {
4         long long res = 1;
5         for (int i = 0; i < b.size(); ++i) {
6             res = pow(res, 10) * pow(a, b[i]) % 1337;
7         }
8         return res;
9     }
10    int pow(int x, int n) {
11        if (n == 0) return 1;
12        if (n == 1) return x % 1337;
13        return pow(x % 1337, n / 2) * pow(x % 1337, n - n / 2) % 1337;
14    }
15 };

```

373. 找和最小的K对数字

You are given two integer arrays `nums1` and `nums2` sorted in ascending order and an integer `k`.

Define a pair (u,v) which consists of one element from the first array and one element from the second array.

Find the k pairs $(u_1,v_1), (u_2,v_2) \dots (u_k,v_k)$ with the smallest sums.

Example 1:

Given `nums1 = [1,7,11]`, `nums2 = [2,4,6]`, `k = 3`

Return: `[1,2],[1,4],[1,6]`

The first 3 pairs are returned from the sequence:

`[1,2],[1,4],[1,6],[7,2],[7,4],[11,2],[7,6],[11,4],[11,6]`

这道题给了我们两个数组，让我们从每个数组中任意取出一个数字来组成不同的数字对，返回前 K 个和最小的数字对。那么这道题有多种解法，我们首先来看brute force的解法，这种方法我们从0循环到数组的个数和 k 之间的较小值，这样做的好处是如果 k 远小于数组个数时，我们不需要计算所有的数字对，而是最多计算 $k*k$ 个数字对，然后将其都保存在`res`里，这时候我们给`res`排序，用我们自定义的比较器，就是和的比较，然后把比 k 多出的数字对删掉即可，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     vector<pair<int, int>> kSmallestPairs(vector<int>& nums1, vector<int>& nums2, int k) {
4         vector<pair<int, int>> res;
5         for (int i = 0; i < min((int)nums1.size(), k); ++i) {
6             for (int j = 0; j < min((int)nums2.size(), k); ++j) {
7                 res.push_back({nums1[i], nums2[j]});
8             }
9         }
10        sort(res.begin(), res.end(), [](pair<int, int> &a, pair<int, int> &b){return
11            a.first + a.second < b.first + b.second;});
12        if (res.size() > k) res.erase(res.begin() + k, res.end());
13        return res;
14    }
15 };

```

我们也可以使用multimap来做，思路是我们将数组对之和作为key存入multimap中，利用其自动排序的机制，这样我们就可以省去sort的步骤，最后把前k个存入res中即可：

解法2：

```
1 class Solution {
2 public:
3     vector<pair<int, int>> kSmallestPairs(vector<int>& nums1, vector<int>& nums2, int k) {
4         vector<pair<int, int>> res;
5         multimap<int, pair<int, int>> m;
6         for (int i = 0; i < min((int)nums1.size(), k); ++i) {
7             for (int j = 0; j < min((int)nums2.size(), k); ++j) {
8                 m.insert({nums1[i] + nums2[j], {nums1[i], nums2[j]}});
9             }
10        }
11        for (auto it = m.begin(); it != m.end(); ++it) {
12            res.push_back(it->second);
13            if (--k == 0) return res;
14        }
15        return res;
16    }
17};
```

下面这种方式用了priority_queue，也需要我们自定义比较器，整体思路和上面的没有什么区别：

解法3：

```
1 class Solution {
2 public:
3     vector<pair<int, int>> kSmallestPairs(vector<int>& nums1, vector<int>& nums2, int k) {
4         vector<pair<int, int>> res;
5         priority_queue<pair<int, int>, vector<pair<int, int>>, cmp> q;
6         for (int i = 0; i < min((int)nums1.size(), k); ++i) {
7             for (int j = 0; j < min((int)nums2.size(), k); ++j) {
8                 if (q.size() < k) {
9                     q.push({nums1[i], nums2[j]});
10                } else if (nums1[i] + nums2[j] < q.top().first + q.top().second) {
11                    q.push({nums1[i], nums2[j]}); q.pop();
12                }
13            }
14        }
15        while (!q.empty()) {
16            res.push_back(q.top()); q.pop();
17        }
18        return res;
19    }
20    struct cmp {
21        bool operator()(pair<int, int> &a, pair<int, int> &b) {
22            return a.first + a.second < b.first + b.second;
23        }
24    };
25};
```

下面这种方法比较另类，我们遍历nums1数组，对于nums1数组中的每一个数字，我们并不需要遍历nums2中所有的数字，实际上，对于nums1中的数字，我们只需要记录nums2中下一个可能组成数字对的坐标，这里我们使用一个idx数组，其中idx[i]表示的数字是nums1[i]将从nums2数组上开始寻找的位置，因为 {nums1[i], nums2[i - 1]} 已经被加入到了结果res中，这种方法其实也是一种地毯式搜索，但是并不需要遍历完所有的组合，因为我们有idx数组来进行剪枝。我们suppose需要进行k次循

环，但是题目中没有说我们一定能取出k对数字，而我们能取出的对儿数跟数组nums1和nums2的长度有关，最多能取出二者的长度之积的对儿数，所以我们取其跟k之间的较小值为循环次数。我们定义idx数组，长度为nums1的长度，初始化均为0。下面开始循环，在每次循环中，我们新建变量cur，记录从nums1中取数的位置，初始化为0，使用变量sum来记录一个当前最小的两数之和，初始化为正无穷。然后开始遍历数组nums1，更新sum的条件有两个，第一个是idx[i]上的数要小于nums2的长度，因为其是在nums2开始寻找的位置，当然不能越界，第二个条件的候选的两个数组 nums1[i] 和 nums2[idx[i]] 之和小于等于 sum。同时满足这两个条件就可以更新sum了，同时更新cur为i，表示当前从nums1取出数字的位置。当遍历nums1的for循环结束后，此时cur的位置就是要从nums1取出的数字的位置，根据idx[cur]从nums2中取出对应的数组，形成数对儿存入结果res中，然后idx[cur]自增1，因为当前位置的数字已经用过了，下次遍历直接从后面一个数字开始吧，这是本解法的设计精髓所在，一定要弄清楚idx数组的意义，参见代码如下：

解法4：

```

1 class Solution {
2 public:
3     vector<pair<int, int>> kSmallestPairs(vector<int>& nums1, vector<int>& nums2, int k) {
4         vector<pair<int, int>> res;
5         int size = min(k, int(nums1.size() * nums2.size()));
6         vector<int> idx(nums1.size(), 0);
7         for (int t = 0; t < size; ++t) {
8             int cur = 0, sum = INT_MAX;
9             for (int i = 0; i < nums1.size(); ++i) {
10                 if (idx[i] < nums2.size() && sum >= nums1[i] + nums2[idx[i]]) {
11                     cur = i;
12                     sum = nums1[i] + nums2[idx[i]];
13                 }
14             }
15             res.push_back({nums1[cur], nums2[idx[cur]]});
16             ++idx[cur];
17         }
18         return res;
19     }
20 };

```

CPP

374. 猜数字大小

We are playing the Guess Game. The game is as follows:

I pick a number from 1 to n. You have to guess which number I picked.

Every time you guess wrong, I'll tell you whether the number is higher or lower.

You call a pre-defined API guess(int num) which returns 3 possible results (-1, 1, or 0):

- 1 : My number is lower
- 1 : My number is higher
- 0 : Congrats! You got it!

这道题是一道典型的猜价格的问题，根据对方说高了还是低了来缩小范围，最简单快速的方法就是折半搜索法，原理很简单，参见代码如下：

```

1 // Forward declaration of guess API.
2 // @param num, your guess
3 // @return -1 if my number is lower, 1 if my number is higher, otherwise return 0
4 int guess(int num);
5
6 class Solution {
7 public:
8     int guessNumber(int n) {
9         if (guess(n) == 0) return n;
10        int left = 1, right = n;
11        while (left < right) {
12            int mid = left + (right - left) / 2, t = guess(mid);
13            if (t == 0) return mid;
14            else if (t == 1) left = mid;
15            else right = mid;
16        }
17        return left;
18    }
19 };
20

```

375. 猜数字大小之二

We are playing the Guess Game. The game is as follows:

I pick a number from 1 to n. You have to guess which number I picked.

Every time you guess wrong, I'll tell you whether the number I picked is higher or lower.

However, when you guess a particular number x, and you guess wrong, you pay \$x. You win the game when you guess the number I picked.

Given a particular $n \geq 1$, find out how much money you need to have to guarantee a win.

Hint:

The best strategy to play the game is to minimize the maximum loss you could possibly face. Another strategy is to minimize the expected loss. Here, we are interested in the first scenario.

Take a small example ($n = 3$). What do you end up paying in the worst case?

Check out this article if you're still stuck.

The purely recursive implementation of minimax would be worthless for even a small n . You MUST use dynamic programming.

As a follow-up, how would you modify your code to solve the problem of minimizing the expected loss, instead of the worst-case loss?

此题是之前那道Guess Number Higher or Lower的拓展，难度增加了不少，根据题目中的提示，这道题需要用到Minimax极小化极大算法，关于这个算法可以参见这篇讲解，并且题目中还说明了要用DP来做，那么我们需要建立一个二维的dp数组，其中 $dp[i][j]$ 表示从数字*i*到*j*之间猜中任意一个数字最少需要花费的钱数，那么我们需要遍历每一段区间 $[j, i]$ ，维护一个全局最小值 $global_min$ 变量，然后遍历该区间中的每一个数字，计算局部最大值 $local_max = k + \max(dp[j][k - 1], dp[k + 1][i])$ ，这个正好是将该区间在每一个位置都分为两段，然后取当前位置的花费加上左右两段中较大的花费之和为局部最大值，为啥要取两者之间的较大值呢，因为我们要cover所有的情况，就得取最坏的情况。然后更新全局最小值，最后在更新 $dp[j][i]$ 的时候看*j*和*i*是否是相邻的，相邻的话赋为*i*，否则赋为 $global_min$ 。这里为啥又要取较小值呢，因为dp数组是求的 $[j, i]$ 范围中的最低cost，比如只有两个数字1和2，那么肯定是猜1的cost低，是不有点晕，没关系，博主继续来绕你。我们想，如果只有一个数字，那么我们不用猜，cost为0。如果有两个数字，比如1和2，我们猜1，即使不对，我们cost也比猜2要低。如果有三个数字1, 2, 3，那么我们就先猜2，根据对方的反馈，就可以确定正确的数字，所以我们的cost最低为2。如果有四个数字1, 2, 3, 4，那么情况就有点复杂了，那么我们的策略是用*k*来遍历所有的数字，然后再根据*k*分成的左右两个区间，取其中的较大cost加上*k*。

当*k*为1时，左区间为空，所以cost为0，而右区间2, 3, 4，根据之前的分析应该取3，所以整个cost就是 $1+3=4$ 。

当*k*为2时，左区间为1，cost为0，右区间为3, 4，cost为3，整个cost就是 $2+3=5$ 。

当*k*为3时，左区间为1, 2，cost为1，右区间为4，cost为0，整个cost就是 $3+1=4$ 。

当*k*为4时，左区间1, 2, 3，cost为2，右区间为空，cost为0，整个cost就是 $4+2=6$ 。

综上*k*的所有情况，此时我们应该取整体cost最小的，即4，为最后的答案，这就是极小化极大算法，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int getMoneyAmount(int n) {
4         vector<vector<int>> dp(n + 1, vector<int>(n + 1, 0));
5         for (int i = 2; i <= n; ++i) {
6             for (int j = i - 1; j > 0; --j) {
7                 int global_min = INT_MAX;
8                 for (int k = j + 1; k < i; ++k) {
9                     int local_max = k + max(dp[j][k - 1], dp[k + 1][i]);
10                    global_min = min(global_min, local_max);
11                }
12                dp[j][i] = j + 1 == i ? j : global_min;
13            }
14        }
15        return dp[1][n];
16    }
17 }
```

CPP

下面这种是递归解法，建立了记忆数组memo，减少了重复计算，提高了运行效率，核心思想跟上面的解法相同，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int getMoneyAmount(int n) {
4         vector<vector<int>> memo(n + 1, vector<int>(n + 1, 0));
5         return helper(1, n, memo);
6     }
7     int helper(int start, int end, vector<vector<int>>& memo) {
8         if (start >= end) return 0;
9         if (memo[start][end] > 0) return memo[start][end];
10        int res = INT_MAX;
11        for (int k = start; k <= end; ++k) {
12            int t = k + max(helper(start, k - 1, memo), helper(k + 1, end, memo));
13            res = min(res, t);
14        }
15        memo[start][end] = res;
16    }
17 };

```

376. 摆动子序列

A sequence of numbers is called a wiggle sequence if the differences between successive numbers strictly alternate between positive and negative. The first difference (if one exists) may be either positive or negative. A sequence with fewer than two elements is trivially a wiggle sequence.

For example, [1,7,4,9,2,5] is a wiggle sequence because the differences (6,-3,5,-7,3) are alternately positive and negative. In contrast, [1,4,7,2,5] and [1,7,4,5,5] are not wiggle sequences, the first because its first two differences are positive and the second because its last difference is zero.

Given a sequence of integers, return the length of the longest subsequence that is a wiggle sequence. A subsequence is obtained by deleting some number of elements (eventually, also zero) from the original sequence, leaving the remaining elements in their original order.

Examples:

Input: [1,7,4,9,2,5]

Output: 6

The entire sequence is a wiggle sequence.

这道题给了我们一个数组，让我们求最长摆动子序列，关于摆动Wiggle数组，可以参见LC上之前的两道题Wiggle Sort和Wiggle Sort II。题目中给的tag说明了这道题可以用DP和Greedy两种方法来做，那么我们先来看DP的做法，我们维护两个dp数组p和q，其中p[i]表示到i位置时首差值为正的摆动子序列的最大长度，q[i]表示到i位置时首差值为负的摆动子序列的最大长度。我们从i=1开始遍历数组，然后对于每个遍历到的数字，再从开头位置遍历到这个数字，然后比较nums[i]和nums[j]，分别更新对应的位置，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int wiggleMaxLength(vector<int>& nums) {
4         if (nums.empty()) return 0;
5         vector<int> p(nums.size(), 1);
6         vector<int> q(nums.size(), 1);
7         for (int i = 1; i < nums.size(); ++i) {
8             for (int j = 0; j < i; ++j) {
9                 if (nums[i] > nums[j]) p[i] = max(p[i], q[j] + 1);
10                else if (nums[i] < nums[j]) q[i] = max(q[i], p[j] + 1);
11            }
12        }
13        return max(p.back(), q.back());
14    }
15 };

```

题目中有个Follow up说要在O(n)的时间内完成，而Greedy算法正好可以达到这个要求，这里我们不在维护两个dp数组，而是维护两个变量p和q，然后遍历数组，如果当前数字比前一个数字大，则 $p=q+1$ ，如果比前一个数字小，则 $q=p+1$ ，最后取p和q中的较大值跟n比较，取较小的那个，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int wiggleMaxLength(vector<int>& nums) {
4         int p = 1, q = 1, n = nums.size();
5         for (int i = 1; i < n; ++i) {
6             if (nums[i] > nums[i - 1]) p = q + 1;
7             else if (nums[i] < nums[i - 1]) q = p + 1;
8         }
9         return min(n, max(p, q));
10    }
11 };

```

377. 组合之和之四

Given an integer array with all positive numbers and no duplicates, find the number of possible combinations that add up to a positive integer target.

Example:

```
nums = [1, 2, 3]
target = 4
```

The possible combination ways are:

```
(1, 1, 1, 1)
(1, 1, 2)
(1, 2, 1)
(1, 3)
(2, 1, 1)
(2, 2)
(3, 1)
```

Note that different sequences are counted as different combinations.

Therefore the output is 7.

Follow up:

What if negative numbers are allowed in the given array?

How does it change the problem?

What limitation we need to add to the question to allow negative numbers?

这道题是组合之和系列的第四道，我开始想当然的一位还是用递归来解，结果写出来发现TLE了，的确OJ给了一个test case为[4,1,2] 32，这个结果是39882198，用递归需要好几秒的运算时间，实在是不高效，估计这也是为啥只让返回一个总和，而不是返回所有情况，不然机子就爆了。而这道题的真正解法应该是用DP来做，解题思想有点像之前爬梯子的那道题Climbing Stairs，我们需要一个一维数组dp，其中dp[i]表示目标数为i的解的个数，然后我们从1遍历到target，对于每一个数i，遍历nums数组，如果*i*>=x，dp[i] += dp[i - x]。这个也很好理解，比如说对于[1,2,3] 4，这个例子，当我们在计算dp[3]的时候，3可以拆分为1+x，而x即为dp[2]，3也可以拆分为2+x，此时x为dp[1]，3同样可以拆为3+x，此时x为dp[0]，我们把所有的情况加起来就是组成3的所有情况了，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     int combinationSum4(vector<int>& nums, int target) {
4         vector<int> dp(target + 1);
5         dp[0] = 1;
6         for (int i = 1; i <= target; ++i) {
7             for (auto a : nums) {
8                 if (i >= a) dp[i] += dp[i - a];
9             }
10        }
11        return dp.back();
12    }
13};
```

CPP

如果target远大于nums数组的个数的话，上面的算法可以做适当的优化，先给nums数组排个序，然后从1遍历到target，对于i小于数组中的数字x时，我们直接break掉，因为后面的数更大，其余地方不变，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int combinationSum4(vector<int>& nums, int target) {
4         vector<int> dp(target + 1);
5         dp[0] = 1;
6         sort(nums.begin(), nums.end());
7         for (int i = 1; i <= target; ++i) {
8             for (auto a : nums) {
9                 if (i < a) break;
10                dp[i] += dp[i - a];
11            }
12        }
13        return dp.back();
14    }
15 };

```

378. 有序矩阵中第K小的元素

Given a $n \times n$ matrix where each of the rows and columns are sorted in ascending order, find the k th smallest element in the matrix.

Note that it is the k th smallest element in the sorted order, not the k th distinct element.

Example:

```
matrix = [
```

```
    [ 1,  5,  9],
    [10, 11, 13],
    [12, 13, 15]
```

```
],
```

```
k = 8,
```

```
return 13.
```

这道题让我们求有序矩阵中第 K 小的元素，这道题的难点在于数组并不是蛇形有序的，意思是当前行的最后一个元素并不一定会小于下一行的首元素，所以我们并不能直接定位第 K 小的元素，所以只能另辟蹊径。先来看一种利用堆的方法，我们使用一个最大堆，然后遍历数组每一个元素，将其加入堆，根据最大堆的性质，大的元素会排到最前面，然后我们看当前堆中的元素个数是否大于 k ，大于的话就将首元素去掉，循环结束后我们返回堆中的首元素即为所求：

解法1：

```

1 class Solution {
2 public:
3     int kthSmallest(vector<vector<int>>& matrix, int k) {
4         priority_queue<int> q;
5         for (int i = 0; i < matrix.size(); ++i) {
6             for (int j = 0; j < matrix[i].size(); ++j) {
7                 q.emplace(matrix[i][j]);
8                 if (q.size() > k) q.pop();
9             }
10        }
11        return q.top();
12    }
13 };

```

这题我们也可以用二分查找法来做，我们由于是有序矩阵，那么左上角的数字一定是最小的，而右下角的数字一定是最大的，所以这个是我们搜索的范围，然后我们算出中间数字mid，由于矩阵中不同行之间的元素并不是严格有序的，所以我们要在每一行都查找一下mid，我们使用upper_bound，这个函数是查找第一个大于目标数的元素，如果目标数在比该行的尾元素大，则upper_bound返回该行元素的个数，如果目标数比该行首元素小，则upper_bound返回0，我们遍历完所有的行可以找出中间数是第几小的数，然后k比较，进行二分查找，left和right最终会相等，并且会变成数组中第k小的数字。举个例子来说吧，比如数组为：

```
[1 2
12 100]
k = 3
那么刚开始left = 1, right = 100, mid = 50, 遍历完 cnt = 3, 此时right更新为50
此时left = 1, right = 50, mid = 25, 遍历完之后 cnt = 3, 此时right更新为25
此时left = 1, right = 25, mid = 13, 遍历完之后 cnt = 3, 此时right更新为13
此时left = 1, right = 13, mid = 7, 遍历完之后 cnt = 2, 此时left更新为8
此时left = 8, right = 13, mid = 10, 遍历完之后 cnt = 2, 此时left更新为11
此时left = 11, right = 12, mid = 11, 遍历完之后 cnt = 2, 此时left更新为12
循环结束，left和right均为12，任意返回一个即可。
```

本解法的整体时间复杂度为 $O(n \lg n * \lg X)$ ，其中X为最大值和最小值的差值，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     int kthSmallest(vector<vector<int>>& matrix, int k) {
4         int left = matrix[0][0], right = matrix.back().back();
5         while (left < right) {
6             int mid = left + (right - left) / 2, cnt = 0;
7             for (int i = 0; i < matrix.size(); ++i) {
8                 cnt += upper_bound(matrix[i].begin(), matrix[i].end(), mid) -
9                     matrix[i].begin();
10            }
11            if (cnt < k) left = mid + 1;
12            else right = mid;
13        }
14        return left;
15    }
};
```

CPP

上面的解法还可以进一步优化到 $O(n \lg X)$ ，其中X为最大值和最小值的差值，我们并不用对每一行都做二分搜索法，我们注意到每列也是有序的，我们可以利用这个性质，从数组的左下角开始查找，如果比目标值小，我们就向右移一位，而且我们知道当前列的当前位置的上面所有的数字都小于目标值，那么 $cnt += i+1$ ，反之则向上移一位，这样我们也能算出cnt的值。其余部分跟上面的方法相同，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int kthSmallest(vector<vector<int>>& matrix, int k) {
4         int left = matrix[0][0], right = matrix.back().back();
5         while (left < right) {
6             int mid = left + (right - left) / 2;
7             int cnt = search_less_equal(matrix, mid);
8             if (cnt < k) left = mid + 1;
9             else right = mid;
10        }
11        return left;
12    }
13    int search_less_equal(vector<vector<int>>& matrix, int target) {
14        int n = matrix.size(), i = n - 1, j = 0, res = 0;
15        while (i >= 0 && j < n) {
16            if (matrix[i][j] <= target) {
17                res += i + 1;
18                ++j;
19            } else {
20                --i;
21            }
22        }
23        return res;
24    }
25 };

```

379. 设计电话目录

Design a Phone Directory which supports the following operations:

get: Provide a number which is not assigned to anyone.

check: Check if a number is available or not.

release: Recycle or release a number.

Example:

```
// Init a phone directory containing a total of 3 numbers: 0, 1, and 2.
PhoneDirectory directory = new PhoneDirectory(3);
```

又是一道设计题，让我们设计一个电话目录管理系统，可以分配电话号码，查询某一个号码是否已经被使用，释放一个号码，需要注意的是，之前释放的号码下一次应该被优先分配。这题对C++解法的时间要求非常苛刻，尝试了好几几种用set，或者stack/queue，或者使用vector的push_back等等，都TLE了，终于找到了一种可以通过OJ的解法。这里用两个一维数组recycle和flag，分别来保存被回收的号码和某个号码的使用状态，还有变量max_num表示最大数字，next表示下一个可以分配的数字，idx表示recycle数组中可以被重新分配的数字的位置，然后在get函数中，没法分配的情况是，当next等于max_num并且index小于等于0，此时返回-1。否则我们先看recycle里有没有数字，有的话先分配recycle里的数字，没有的话再分配next。记得更新相对应的flag中的使用状态，参见代码如下：

```

1 class PhoneDirectory {
2 public:
3     /** Initialize your data structure here
4         @param maxNumbers - The maximum numbers that can be stored in the phone directory.
5 */
6     PhoneDirectory(int maxNumbers) {
7         max_num = maxNumbers;
8         next = idx = 0;
9         recycle.resize(max_num);
10        flag.resize(max_num, 1);
11    }
12
13    /** Provide a number which is not assigned to anyone.
14        @return - Return an available number. Return -1 if none is available. */
15    int get() {
16        if (next == max_num && idx <= 0) return -1;
17        if (idx > 0) {
18            int t = recycle[--idx];
19            flag[t] = 0;
20            return t;
21        }
22        flag[next] = false;
23        return next++;
24    }
25
26    /** Check if a number is available or not. */
27    bool check(int number) {
28        return number >= 0 && number < max_num && flag[number];
29    }
30
31    /** Recycle or release a number. */
32    void release(int number) {
33        if (number >= 0 && number < max_num && !flag[number]) {
34            recycle[idx++] = number;
35            flag[number] = 1;
36        }
37    }
38 private:
39     int max_num, next, idx;
40     vector<int> recycle, flag;
41 };

```

380. 常数时间内插入删除和获得随机数

Design a data structure that supports all following operations in average O(1) time.

`insert(val)`: Inserts an item val to the set if not already present.

`remove(val)`: Removes an item val from the set if present.

`getRandom`: Returns a random element from current set of elements. Each element must have the same probability of being returned.

Example:

```

// Init an empty set.
RandomizedSet randomizedSet = new RandomizedSet();

```

这道题让我们在常数时间范围内实现插入删除和获得随机数操作，如果这道题没有常数时间的限制，那么将会是一道非常简单的题，我们直接用一个set就可以搞定所有的操作。但是由于时间的限制，我们无法在常数时间内实现获取随机数，所以只能另辟蹊径。此题的正确解法是利用到了一个一维数组和一个哈希表，其中数组用来保存数字，哈希表用来建立每个数字和其在数组中的位置之间的映射，对于插入操作，我们先看这个数字是否已经在哈希表中存在，如果存在的话直接返回false，不存在的话，我们将其插入到数组的末尾，然后建立数字和其位置的映射。删除操作是比较tricky的，我们还是要先判断其是否在哈希表里，如果没有，直接返回false。由于哈希表的删除是常数时间的，而数组并不是，为了使数组删除也能常数级，我们实际上将要删除的数字和数组的最后一个数字调换个位置，然后修改对应的哈希表中的值，这样我们只需要删除数组的最后一个元素即可，保证了常数时间内的删除。而返回随机数对于数组来说就很简单了，我们只要随机生成一个位置，返回该位置上的数字即可，参见代码如下：

```

1 class RandomizedSet {
2 public:
3     /** Initialize your data structure here. */
4     RandomizedSet() {}
5
6     /** Inserts a value to the set. Returns true if the set did not already contain the
7      * specified element. */
8     bool insert(int val) {
9         if (m.count(val)) return false;
10        nums.push_back(val);
11        m[val] = nums.size() - 1;
12        return true;
13    }
14
15    /** Removes a value from the set. Returns true if the set contained the specified
16     * element. */
17    bool remove(int val) {
18        if (!m.count(val)) return false;
19        int last = nums.back();
20        m[last] = m[val];
21        nums[m[val]] = last;
22        nums.pop_back();
23        m.erase(val);
24        return true;
25    }
26
27    /** Get a random element from the set. */
28    int getRandom() {
29        return nums[rand() % nums.size()];
30    }
31 private:
32     vector<int> nums;
33     unordered_map<int, int> m;
34 };

```

381. 常数时间内插入删除和获得随机数 - 允许重复

Design a data structure that supports all following operations in average O(1) time.

Note: Duplicate elements are allowed.

`insert(val)`: Inserts an item `val` to the collection.

`remove(val)`: Removes an item `val` from the collection if present.

`getRandom`: Returns a random element from current collection of elements. The probability of each element being returned is linearly related to the number of same value the collection contains.

这题是之前那道Insert Delete GetRandom O(1)的拓展，与其不同的是，之前那道题不能有重复数字，而这道题可以有，那么就不能像之前那道题那样建立每个数字和其坐标的映射了，但是我们可以建立数字和其所有出现位置的集合之间的映射，虽然写法略有不同，但是思路和之前那题完全一样，都是将数组最后一个位置的元素和要删除的元素交换位置，然后删掉最后一个位置上的元素。对于insert函数，我们将要插入的数字在nums中的位置加入m[val]数组的末尾，然后在数组nums末尾加入val，我们判断是否有重复只要看m[val]数组只有刚加的val一个值还是有多个值。remove函数是这题的难点，我们首先看哈希表中有没有val，没有的话直接返回false。然后我们取出nums的尾元素，把尾元素哈希表中的位置数组中的最后一个位置更新为m[val]的尾元素，这样我们就可以删掉m[val]的尾元素了，如果m[val]只有一个元素，那么我们把这个映射直接删除。然后我们将nums数组中的尾元素删除，并把尾元素赋给val所在的位置，注意我们在建立哈希表的映射的时候需要用堆而不是普通的vector数组，因为我们每次remove操作后都会移除nums数组的尾元素，如果我们用vector来保存数字的坐标，而且只移出末尾数字的话，有可能出现前面的坐标大小超过了此时nums的大小的情况，就会出错，所以我们用优先队列对所有的相同数字的坐标进行自动排序，每次把最大位置的坐标移出即可，参见代码如下：

解法1：

CPP

```

1 class RandomizedCollection {
2 public:
3     /** Initialize your data structure here. */
4     RandomizedCollection() {}
5
6     /** Inserts a value to the collection. Returns true if the collection did not already
7     contain the specified element. */
8     bool insert(int val) {
9         m[val].push(nums.size());
10        nums.push_back(val);
11        return m[val].size() == 1;
12    }
13
14    /** Removes a value from the collection. Returns true if the collection contained the
15     specified element. */
16    bool remove(int val) {
17        if (m[val].empty()) return false;
18        int idx = m[val].top();
19        m[val].pop();
20        if (nums.size() - 1 != idx) {
21            int t = nums.back();
22            nums[idx] = t;
23            m[t].pop();
24            m[t].push(idx);
25        }
26        nums.pop_back();
27        return true;
28    }
29
30    /** Get a random element from the collection. */
31    int getRandom() {
32        return nums[rand() % nums.size()];
33    }
34 private:
35     vector<int> nums;
36     unordered_map<int, priority_queue<int>> m;
37 };

```

有网友指出上面的方法其实不是真正的O(1)时间复杂度，因为优先队列的push不是常数级的，博主一看果然是这样的，为了严格的遵守O(1)的时间复杂度，我们将优先队列换成unordered_set，其插入删除的操作都是常数量级的，其他部分基本不用变，参见代码如下：

解法2：

```

1 class RandomizedCollection {
2 public:
3     /** Initialize your data structure here. */
4     RandomizedCollection() {}
5
6     /** Inserts a value to the collection. Returns true if the collection did not already
7 contain the specified element. */
8     bool insert(int val) {
9         m[val].insert(nums.size());
10        nums.push_back(val);
11        return m[val].size() == 1;
12    }
13
14    /** Removes a value from the collection. Returns true if the collection contained the
15 specified element. */
16    bool remove(int val) {
17        if (m[val].empty()) return false;
18        int idx = *m[val].begin();
19        m[val].erase(idx);
20        if (nums.size() - 1 != idx) {
21            int t = nums.back();
22            nums[idx] = t;
23            m[t].erase(nums.size() - 1);
24            m[t].insert(idx);
25        }
26        nums.pop_back();
27        return true;
28    }
29
30    /** Get a random element from the collection. */
31    int getRandom() {
32        return nums[rand() % nums.size()];
33    }
34
35 private:
36     vector<int> nums;
37     unordered_map<int, unordered_set<int>> m;
38 };

```

382. 链表随机节点

Given a singly linked list, return a random node's value from the linked list. Each node must have the same probability of being chosen.

Follow up:

What if the linked list is extremely large and its length is unknown to you? Could you solve this efficiently without using extra space?

这道题给了我们一个链表，让我们随机返回一个节点，那么最直接的方法就是先统计出链表的长度，然后根据长度随机生成一个位置，然后从开头遍历到这个位置即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     /** @param head The linked list's head. Note that the head is guaranteed to be not
4      * null, so it contains at least one node. */
5     Solution(ListNode* head) {
6         len = 0;
7         ListNode *cur = head;
8         this->head = head;
9         while (cur) {
10             ++len;
11             cur = cur->next;
12         }
13     }
14
15     /** Returns a random node's value. */
16     int getRandom() {
17         int t = rand() % len;
18         ListNode *cur = head;
19         while (t) {
20             --t;
21             cur = cur->next;
22         }
23         return cur->val;
24     }
25 private:
26     int len;
27     ListNode *head;
28 };

```

Follow up中说链表可能很长，我们没法提前知道长度，这里用到了著名的水塘抽样Reservoir Sampling的思路，由于限定了head一定存在，所以我们先让返回值res等于head的节点值，然后让cur指向head的下一个节点，定义一个变量i，初始化为2，若cur不为空我们开始循环，我们在 $[0, i - 1]$ 中取一个随机数，如果取出来0，那么我们更新res为当前的cur的节点值，然后此时自增一，cur指向其下一个位置，这里其实相当于我们维护了一个大小为1的水塘，然后我们随机数生成为0的话，我们交换水塘中的值和当前遍历到的值，这样可以保证每个数字的概率相等，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     /** @param head The linked list's head. Note that the head is guaranteed to be not
4      * null, so it contains at least one node. */
5     Solution(ListNode* head) {
6         this->head = head;
7     }
8
9     /** Returns a random node's value. */
10    int getRandom() {
11        int res = head->val, i = 2;
12        ListNode *cur = head->next;
13        while (cur) {
14            int j = rand() % i;
15            if (j == 0) res = cur->val;
16            ++i;
17            cur = cur->next;
18        }
19        return res;
20    }
21 private:
22     ListNode *head;
23 };

```

383. 贼金条

Given an arbitrary ransom note string and another string containing letters from all the magazines, write a function that will return true if the note can be constructed from the magazines; otherwise, it will return false.

Each letter in the magazine string can only be used once in your ransom note.

Note:

You may assume that both strings contain only lowercase letters.

```

canConstruct("a", "b") -> false
canConstruct("aa", "ab") -> false
canConstruct("aa", "aab") -> true

```

非常简单的一道题，就是用哈希Map统计字符的个数，参见代码如下：

```

1 class Solution {
2 public:
3     bool canConstruct(string ransomNote, string magazine) {
4         unordered_map<char, int> m;
5         for (char c : magazine) ++m[c];
6         for (char c : ransomNote) {
7             if (--m[c] < 0) return false;
8         }
9         return true;
10    }
11 };

```

384. 数组洗牌

Shuffle a set of numbers without duplicates.

这道题让我们给数组洗牌，也就是随机打乱顺序，那么由于之前那道题Linked List Random Node我们接触到了水塘抽样Reservoir Sampling的思想，这道题实际上这道题也是用类似的思路，我们遍历数组每个位置，每次都随机生成一个坐标位置，然后交换当前遍历位置和随机生成的坐标位置的数字，这样如果数组有n个数字，那么我们也随机交换了n组位置，从而达到了洗牌的目的，这里需要注意的是 $i + \text{rand}() \% (\text{res.size()} - i)$ 不能写成 $\text{rand}() \% \text{res.size}()$ ，虽然也能通过OJ，但是根据这个帖子的最后部分的概率图表，前面那种写法不是真正的随机分布，应该使用Knuth shuffle算法，感谢热心网友们的留言，参见代码如下：

```

1 class Solution {
2 public:
3     Solution(vector<int> nums): v(nums) {}
4
5     /** Resets the array to its original configuration and return it. */
6     vector<int> reset() {
7         return v;
8     }
9
10    /** Returns a random shuffling of the array. */
11    vector<int> shuffle() {
12        vector<int> res = v;
13        for (int i = 0; i < res.size(); ++i) {
14            int t = i + rand() % (res.size() - i);
15            swap(res[i], res[t]);
16        }
17        return res;
18    }
19
20 private:
21     vector<int> v;
22 };

```

CPP

385. 迷你解析器

Given a nested list of integers represented as a string, implement a parser to deserialize it.

Each element is either an integer, or a list -- whose elements may also be integers or other lists.

Note: You may assume that the string is well-formed:

String is non-empty.
String does not contain white spaces.
String contains only digits 0-9, [, - , ,].

这道题让我们实现一个迷你解析器用来把一个字符串解析成NestInteger类，关于这个嵌套链表类的题我们之前做过三道，Nested List Weight Sum II, Flatten Nested List Iterator, 和Nested List Weight Sum。应该对这个类并不陌生了，我们可以先用递归来做，思路是，首先判断s是否为空，为空直接返回，不为空的话看首字符是否为'[', 不是的话说明s为一个整数，我们直接返回结果。如果首字符是'[', 且s长度小于等于2，说明没有内容，直接返回结果。反之如果s长度大于2，我们从i=1开始遍历，我们需要一个变量start来记录某一层的其实位置，用cnt来记录跟其实位置是否为同一深度，cnt=0表示同一深度，由于中间每段都是由逗号隔开，所以当我们判断当cnt为0，且当前字符是逗号或者已经到字符串末尾了，我们把start到当前位置之间的字符串取出来递归调用函数，把返回结果加入res中，然后start更新为i+1。如果遇到'[', 计数器cnt自增1，若遇到']', 计数器cnt自减1。参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     NestedInteger deserialize(string s) {
4         if (s.empty()) return NestedInteger();
5         if (s[0] != '[') return NestedInteger(stoi(s));
6         if (s.size() <= 2) return NestedInteger();
7         NestedInteger res;
8         int start = 1, cnt = 0;
9         for (int i = 1; i < s.size(); ++i) {
10             if (cnt == 0 && (s[i] == ',' || i == s.size() - 1)) {
11                 res.add(deserialize(s.substr(start, i - start)));
12                 start = i + 1;
13             } else if (s[i] == '[') ++cnt;
14             else if (s[i] == ']') --cnt;
15         }
16         return res;
17     }
18 };

```

我们也可以使用迭代的方法来做，这样就需要使用栈来辅助，变量start记录起始位置，我们遍历字符串，如果遇到'[', 我们给栈中加上一个空的NestedInteger，如果遇到的字符数逗号或者']', 如果i>start，那么我们给栈顶元素调用add来新加一个NestedInteger，初始化参数传入start到i之间的子字符串转为的整数，然后更新start=i+1，当遇到的']'时，如果此时栈中元素多于1个，那么我们将栈顶元素取出，加入新的栈顶元素中通过调用add函数，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     NestedInteger deserialize(string s) {
4         if (s.empty()) return NestedInteger();
5         if (s[0] != '[') return NestedInteger(stoi(s));
6         stack<NestedInteger> st;
7         int start = 1;
8         for (int i = 0; i < s.size(); ++i) {
9             if (s[i] == '[') {
10                 st.push(NestedInteger());
11                 start = i + 1;
12             } else if (s[i] == ',' || s[i] == ']') {
13                 if (i > start) {
14                     st.top().add(NestedInteger(stoi(s.substr(start, i - start))));
15                 }
16                 start = i + 1;
17                 if (s[i] == ']') {
18                     if (st.size() > 1) {
19                         NestedInteger t = st.top(); st.pop();
20                         st.top().add(t);
21                     }
22                 }
23             }
24         }
25         return st.top();
26     }
27 };

```

还有一种方法是利用C++ STL中的字符串流处理类`istringstream`，我们需要对几个函数有些了解，比如`clear()`是重置字符串流中的字符串，`get()`是获得下一个字符，`peek()`是返回首字符，`>>num`是读取出合法的整数，如果无法读取出整数，需要调用`clear()`来重置字符串，否则调用`get()`会出错。思路跟上面的递归解法相同，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     NestedInteger deserialize(string s) {
4         istringstream in(s);
5         return deserialize(in);
6     }
7     NestedInteger deserialize(istringstream& in) {
8         int num;
9         if (in >> num) return NestedInteger(num);
10        in.clear();
11        in.get();
12        NestedInteger list;
13        while (in.peek() != ']') {
14            list.add(deserialize(in));
15            if (in.peek() == ',') {
16                in.get();
17            }
18        }
19        in.get();
20        return list;
21    }
22 };

```

386. 字典顺序的数字

Given an integer n, return 1 - n in lexicographical order.

For example, given 13, return: [1,10,11,12,13,2,3,4,5,6,7,8,9].

Please optimize your algorithm to use less time and space. The input size may be as large as 5,000,000.

这道题给了我们一个整数n，让我们把区间[1,n]的所有数字按照字典顺序来排列，题目中也给了我们字典顺序的例子。那么我们需要重新排序，我最开始想到的方法是重写`sort`方法的`comparator`，思路是把所有数字都转为字符串，然后两个字符串按位相比，然后排好序后再转回数字，这种方法通过不了OJ的大集合，说明本题不是想考我们这种方法。我在论坛里看到大家普遍使用的是下面这种方法，学习了一下，感觉思路十分巧妙，估计我自己肯定想不出来。这种思路是按个位数遍历，在遍历下一个个位数之前，先遍历十位数，十位数的高位为之前的个位数，只要这个多位数并没有超过n，就可以一直往后遍历，如果超过了，我们除以10，然后再加1，如果加1后末尾形成了很多0，那么我们要用个`while`循环把0都去掉，然后继续运算，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<int> lexicalOrder(int n) {
4         vector<int> res(n);
5         int cur = 1;
6         for (int i = 0; i < n; ++i) {
7             res[i] = cur;
8             if (cur * 10 <= n) {
9                 cur *= 10;
10            } else {
11                if (cur >= n) cur /= 10;
12                cur += 1;
13                while (cur % 10 == 0) cur /= 10;
14            }
15        }
16        return res;
17    }
18 };

```

下面这种方法是上面解法的递归形式，思路并没有什么不同，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<int> lexicalOrder(int n) {
4         vector<int> res;
5         for (int i = 1; i <= 9; ++i) {
6             helper(i, n, res);
7         }
8         return res;
9     }
10    void helper(int cur, int n, vector<int>& res) {
11        if (cur > n) return;
12        res.push_back(cur);
13        for (int i = 0; i <= 9; ++i) {
14            if (cur * 10 + i <= n) {
15                helper(cur * 10 + i, n, res);
16            } else break;
17        }
18    }
19 };

```

387. 字符串第一个不同字符

Given a string, find the first non-repeating character in it and return it's index. If it doesn't exist, return -1.

Examples:

```
s = "leetcode"
return 0.
```

这道题确实没有什么难度，我们只要用哈希表建立每个字符和其出现次数的映射，然后按顺序遍历字符串，找到第一个出现次数为1的字符，返回其位置即可，参见代码如下：

```

1 class Solution {
2 public:
3     int firstUniqChar(string s) {
4         unordered_map<char, int> m;
5         for (char c : s) ++m[c];
6         for (int i = 0; i < s.size(); ++i) {
7             if (m[s[i]] == 1) return i;
8         }
9         return -1;
10    }
11 };

```

388. 最长的绝对文件路径

Suppose we abstract our file system by a string in the following manner:

The string "dir\n\tsubdir1\n\tsubdir2\n\t\tfile.ext" represents:

```

dir
  subdir1
  subdir2
    file.ext

```

The directory dir contains an empty sub-directory subdir1 and a sub-directory subdir2 containing a file file.ext.

The string
"dir\n\tsubdir1\n\t\tfile1.ext\n\t\tsubsubdir1\n\t\tsubdir2\n\t\t\tsubsubdir2\n\t\t\t\tfile2.ext"
represents:

```

dir
  subdir1
    file1.ext
    subsubdir1
  subdir2
    subsubdir2
      file2.ext

```

The directory dir contains two sub-directories subdir1 and subdir2. subdir1 contains a file file1.ext and an empty second-level sub-directory subsubdir1. subdir2 contains a second-level sub-directory subsubdir2 containing a file file2.ext.

We are interested in finding the longest (number of characters) absolute path to a file within our file system. For example, in the second example above, the longest absolute path is "dir/subdir2/subsubdir2/file2.ext", and its length is 32 (not including the double quotes).

Given a string representing the file system in the above format, return the length of the longest absolute path to file in the abstracted file system. If there is no file in the system, return 0.

这道题给了我们一个字符串，里面包含\n和\t这种表示回车和空格的特殊字符，让我们找到某一个最长的绝对文件路径，要注意的是，最长绝对文件路径不一定是最深的路径，我们可以用哈希表来建立深度和当前深度的绝对路径长度之间的映射，那么当当前深度下的文件的绝对路径就是文件名长度加上哈希表中当前深度对应的长度。我们的思路是遍历整个字符串，遇到\n或者\t就停下来，然后我们判断，如果遇到的是回车，我们把这段文件名提取出来，如果里面包含'.'，说明是文件，我们更新res长度，如果不包含点，说明是文件夹，我们深度level自增1，然后建立当前深度和总长度之间的映射，然后我们将深度level重置为0。之前如果遇到的是空格\t，那么我们深度加一，通过累加\t的个数，我们可以得知当前文件或文件夹的深度，然后做对应的处理，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int lengthLongestPath(string input) {
4         int res = 0, n = input.size(), level = 0;
5         unordered_map<int, int> m {{0, 0}};
6         for (int i = 0; i < n; ++i) {
7             int start = i;
8             while (i < n && input[i] != '\n' && input[i] != '\t') ++i;
9             if (i >= n || input[i] == '\n') {
10                 string t = input.substr(start, i - start);
11                 if (t.find('.') != string::npos) {
12                     res = max(res, m[level] + (int)t.size());
13                 } else {
14                     ++level;
15                     m[level] = m[level - 1] + (int)t.size() + 1;
16                 }
17                 level = 0;
18             } else {
19                 ++level;
20             }
21         }
22         return res;
23     }
24 };

```

CPP

下面这种方法用到了字符串流机制，通过getline函数可以一行一行的获取数据，实际上相当于根据回车符\n把每段分割开了，然后对于每一行，我们找最后一个空格符\t的位置，然后可以得到文件或文件夹的名字，然后我们判断其是文件还是文件夹，如果是文件就更新res，如果是文件夹就更新哈希表的映射，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int lengthLongestPath(string input) {
4         int res = 0;
5         istringstream ss(input);
6         unordered_map<int, int> m {{0, 0}};
7         string line = "";
8         while (getline(ss, line)) {
9             int level = line.find_last_of('\t') + 1;
10            int len = line.substr(level).size();
11            if (line.find('.') != string::npos) {
12                res = max(res, m[level] + len);
13            } else {
14                m[level + 1] = m[level] + len + 1;
15            }
16        }
17        return res;
18    }
19 };

```

CPP

389. 寻找不同

Given two strings s and t which consist of only lowercase letters.

String t is generated by random shuffling string s and then add one more letter at a random position.

Find the letter that was added in t.

Example:

Input:

```
s = "abcd"
t = "abcde"
```

Output:

```
e
```

这道题给了我们两个字符串s和t, t是在s的任意一个地方加上了一个字符，让我们找出新加上的那个字符。这道题确实不是一道难题，首先第一反应的方法就是用哈希表来建立字符和个数之间的映射，如果在遍历t的时候某个映射值小于0了，那么返回该字符即可，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     char findTheDifference(string s, string t) {
4         unordered_map<char, int> m;
5         for (char c : s) ++m[c];
6         for (char c : t) {
7             if (--m[c] < 0) return c;
8         }
9         return 0;
10    }
11 }
```

CPP

我们也可以使用位操作Bit Manipulation来做，利用异或的性质，相同位返回0，这样相同的字符都抵消了，剩下的就是后加的那个字符，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     char findTheDifference(string s, string t) {
4         char res = 0;
5         for (char c : s) res ^= c;
6         for (char c : t) res ^= c;
7         return res;
8     }
9 }
```

CPP

我们也可以直接用加和减，相同的字符一减一加也抵消了，剩下的就是后加的那个字符，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     char findTheDifference(string s, string t) {
4         char res = 0;
5         for (char c : s) res -= c;
6         for (char c : t) res += c;
7         return res;
8     }
9 };

```

下面这种方法是史蒂芬大神提出来的，利用了STL的accumulate函数，实际上是上面解法二的改写，一行就写完了真是丧心病狂啊，参见代码如下：

解法4：

```

1 class Solution {
2 public:
3     char findTheDifference(string s, string t) {
4         return accumulate(begin(s), end(s) += t), 0, bit_xor<int>());
5     }
6 };

```

390. 淘汰游戏

There is a list of sorted integers from 1 to n. Starting from left to right, remove the first number and every other number afterward until you reach the end of the list.

Repeat the previous step again, but this time from right to left, remove the right most number and every other number from the remaining numbers.

We keep repeating the steps again, alternating left to right and right to left, until a single number remains.

Find the last number that remains starting with a list of length n.

Example:

Input:
n = 9,
1 2 3 4 5 6 7 8 9
2 4 6 8
2 6
6

Output:

6

这道题是LeetCode第二次编程比赛的题，然而博主并没有做出来，博主用的方法是那种最笨的方法，用一个数组把n个数组都存起来，然后根据循环的奇偶来决定是从左还是从右删除，结果不幸超时TLE了。后来通过想大神请教和上网搜索，发现这道题用递归来很简单，我们用一个bool型变量left2right，为true表示从左往右，为false表示从右往左遍历。当n为1时，不论从左往右还是从右往左都返回1。如果n大于1，且是从左往右的话，我们返回2倍的对n/2的从右往左的遍历；如果是从右往左的话，稍稍麻烦一些，我们肯定还是要对n/2调用递归函数的，但是要分奇偶情况，如果n为奇数，返回2倍的对n/2的从左往右的遍历的值；如果n为偶数，2倍的对n/2的从左往右的遍历的值，再减去1。具体这样的原因，楼主还在研究中，也不是太清楚：

解法1：

```

1 class Solution {
2 public:
3     int lastRemaining(int n) {
4         return help(n, true);
5     }
6     int help(int n, bool left2right) {
7         if (n == 1) return 1;
8         if (left2right) {
9             return 2 * help(n / 2, false);
10        } else {
11            return 2 * help(n / 2, true) - 1 + n % 2;
12        }
13    }
14 };

```

下面这种方法相当的叼，一行就搞定了简直丧心病狂啊。第一次从左往右删除的时候，奇数都被删掉了，剩下的都是偶数。如果我们对所有数都除以2，那么得到一个1到n/2的新数列。下一次我们从右往左删出，那么返回的结果应该是调用递归的结果lastRemaining(n / 2)在数组1到n/2之间的镜像。何为镜像，比如1, 2, 3, 4这个数字，2的镜像就是3, 1的镜像是4，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int lastRemaining(int n) {
4         return n == 1 ? 1 : 2 * (1 + n / 2 - lastRemaining(n / 2));
5     }
6 };

```

下面这种迭代的解法是我请教另一位大神的方法，个人感觉也非常叼，膜拜大神中。我们先来看两个简单的例子：

```

n = 8
1 2 3 4 5 6 7 8
      2       4       6       8
          2           6
              6

```

```

n = 7
1 2 3 4 5 6 7
      2       4       6
          4

```

如果我们仔细观察，我们可以发现从左往右删的时候，每次都是删掉第一个数字，而从右往左删的时候，则有可能删掉第一个或者第二个数字，而且每删一次，数字之间的距离会变为之前的两倍。我们要做的是每次记录当前数组的第一个数字，而且我们再通过观察可以看出，从右往左删时，如果剩下的数字个数是偶数个时，删掉的是第二个数字；如果是奇数个的时候，删掉的是第一个数字。总结出了上述规律，就可以写出代码如下：

解法3：

```

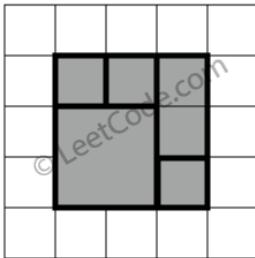
1 class Solution {
2 public:
3     int lastRemaining(int n) {
4         int base = 1, res = 1;
5         while (base * 2 <= n) {
6             res += base;
7             base *= 2;
8             if (base * 2 > n) break;
9             if ((n / base) % 2 == 1) res += base;
10            base *= 2;
11        }
12        return res;
13    }
14 };

```

391. 完美矩形

Given N axis-aligned rectangles where N > 0, determine if they all together form an exact cover of a rectangular region.

Each rectangle is represented as a bottom-left point and a top-right point. For example, a unit square is represented as [1,1,2,2]. (coordinate of bottom-left point is (1, 1) and top-right point is (2, 2)).



这道题是LeetCode第二周编程比赛的压轴题目，然而我并没有做出来，我想了两种方法都无法通过OJ的大数据集合，第一种方法是对于每一个矩形，我将其拆分为多个面积为1的单位矩形，然后以其左下方的点为标记，用一个哈希表建立每一个单位矩形和遍历到的矩形的映射，因为每个单位矩形只能属于一个矩形，否则就会有重叠，我感觉这种思路应该没错，但是由于把每一个遍历到的矩形拆分为单位矩形再建立映射很费时间，尤其是当矩形很大的时候，TLE就很正常了，后来我试的第二种方法是对于遍历到的每个矩形都和其他所有矩形检测一遍是否重叠，这种方法也是毫无悬念的TLE。

博主能力有限，只能去论坛中找各位大神的解法，发现下面两种方法比较fancy，也比较好理解。首先来看第一种方法，这种方法的设计思路很巧妙，利用了mask，也就是位操作Bit Manipulation的一些技巧，下面这张图来自这个帖子：



所有的矩形的四个顶点只会有下面蓝，绿，红三种情况，其中蓝表示该顶点周围没有其他矩形，T型的绿点表示两个矩形并排相邻，红点表示四个矩形相邻，那么在一个完美矩形中，蓝色的点只能有四个，这是个很重要的判断条件。我们再来看矩形的四个顶点，我们按照左下，左上，右上，右下的顺序来给顶点标号为1, 2, 4, 8，为啥不是1, 2, 3, 4呢，我们注意它们的二进制1(0001), 2(0010), 4(0100), 8(1000)，这样便于我们与或的操作，我们还需要知道的一个判定条件是，当一个点是某一个矩形的左下顶点时，这个点就不能是其他矩形的左下顶点了，这个条件对于这四种顶点都要成立，那么对于每一个点，如果它是某个矩形的四个顶点之一，我们记录下来，如果在别的矩形中它又是相同的顶点，那么直接返回false即可，这样就体现了我们标记为1, 2, 4, 8的好处，我们可以按位检查1。如果每个点的属性没有冲突，那么我们来验证每个点的mask是否合理，通过上面的分析，我们知道每个点只能是蓝，绿，红三种情况的一种，其中蓝的情况是mask的四位中只有一个1，分别就是1(0001), 2(0010), 4(0100), 8(1000)，而且蓝点只能有四个；那么对于T型的绿点，mask的四位中有两个1，那么就有六种情况，分别是12(1100), 10(1010), 9(1001), 6(0110), 5(0101), 3(0011)；而对于红点，mask的四位都是1，只有一种情况15(1111)，那么我们

可以通过直接找mask是1, 2, 4, 8的个数，也可以间接通过找不是绿点和红点的个数，看是否是四个。最后一个判定条件是每个矩形面积累加和要等于最后的大矩形的面积，那么大矩形的面积我们通过计算最小左下点和最大右上点来计算出来即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool isRectangleCover(vector<vector<int>>& rectangles) {
4         unordered_map<string, int> m;
5         int min_x = INT_MAX, min_y = INT_MAX, max_x = INT_MIN, max_y = INT_MIN, area = 0,
6         cnt = 0;
7         for (auto rect : rectangles) {
8             min_x = min(min_x, rect[0]);
9             min_y = min(min_y, rect[1]);
10            max_x = max(max_x, rect[2]);
11            max_y = max(max_y, rect[3]);
12            area += (rect[2] - rect[0]) * (rect[3] - rect[1]);
13            if (!isValid(m, to_string(rect[0]) + "_" + to_string(rect[1]), 1)) return
14 false; // bottom-left
15            if (!isValid(m, to_string(rect[0]) + "_" + to_string(rect[3]), 2)) return
16 false; // top-left
17            if (!isValid(m, to_string(rect[2]) + "_" + to_string(rect[3]), 4)) return
18 false; // top-right
19            if (!isValid(m, to_string(rect[2]) + "_" + to_string(rect[1]), 8)) return
20 false; // bottom-right
21        }
22        for (auto it = m.begin(); it != m.end(); ++it) {
23            int t = it->second;
24            if (t != 15 && t != 12 && t != 10 && t != 9 && t != 6 && t != 5 && t != 3) {
25                ++cnt;
26            }
27        }
28        return cnt == 4 && area == (max_x - min_x) * (max_y - min_y);
29    }
30    bool isValid(unordered_map<string, int>& m, string corner, int type) {
31        int& val = m[corner];
32        if (val & type) return false;
33        val |= type;
34        return true;
35    }
36};

```

下面这种方法也相当的巧妙，提出这种算法的大神细心的发现了每种点的规律，每个绿点其实都是两个顶点的重合，每个红点都是四个顶点的重合，而每个蓝点只有一个顶点，有了这条神奇的性质就不用再去判断“每个点最多只能是一个矩形的左下，左上，右上，或右下顶点”这条性质了，我们直接用一个set，对于遍历到的任意一个顶点，如果set中已经存在了，则删去这个点，如果没有就加上，这样最后会把绿点和红点都滤去，剩下的都是蓝点，我们只要看蓝点的个数是否为四个，再加上检测每个矩形面积累加和要等于最后的大矩形的面积即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool isRectangleCover(vector<vector<int>>& rectangles) {
4         unordered_set<string> st;
5         int min_x = INT_MAX, min_y = INT_MAX, max_x = INT_MIN, max_y = INT_MIN, area = 0;
6         for (auto rect : rectangles) {
7             min_x = min(min_x, rect[0]);
8             min_y = min(min_y, rect[1]);
9             max_x = max(max_x, rect[2]);
10            max_y = max(max_y, rect[3]);
11            area += (rect[2] - rect[0]) * (rect[3] - rect[1]);
12            string s1 = to_string(rect[0]) + "_" + to_string(rect[1]); // bottom-left
13            string s2 = to_string(rect[0]) + "_" + to_string(rect[3]); // top-left
14            string s3 = to_string(rect[2]) + "_" + to_string(rect[3]); // top-right
15            string s4 = to_string(rect[2]) + "_" + to_string(rect[1]); // bottom-right
16            if (st.count(s1)) st.erase(s1);
17            else st.insert(s1);
18            if (st.count(s2)) st.erase(s2);
19            else st.insert(s2);
20            if (st.count(s3)) st.erase(s3);
21            else st.insert(s3);
22            if (st.count(s4)) st.erase(s4);
23            else st.insert(s4);
24        }
25        string t1 = to_string(min_x) + "_" + to_string(min_y);
26        string t2 = to_string(min_x) + "_" + to_string(max_y);
27        string t3 = to_string(max_x) + "_" + to_string(max_y);
28        string t4 = to_string(max_x) + "_" + to_string(min_y);
29        if (!st.count(t1) || !st.count(t2) || !st.count(t3) || !st.count(t4) || st.size()
30 != 4) return false;
31        return area == (max_x - min_x) * (max_y - min_y);
32    }
33};

```

392. 是子序列

Given a string s and a string t, check if s is subsequence of t.

You may assume that there is only lower case English letters in both s and t. t is potentially a very long (length $\approx 500,000$) string, and s is a short string (≤ 100).

A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, "ace" is a subsequence of "abcde" while "aec" is not).

Example 1:

s = "abc", t = "ahbgdc"

Return true.

这道题算比较简单的一种，我们可以用两个指针分别指向字符串s和t，然后如果字符相等，则i和j自增1，反之只有j自增1，最后看i是否等于s的长度，等于说明s已经遍历完了，而且字符都有在t中出现过，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool isSubsequence(string s, string t) {
4         if (s.empty()) return true;
5         int i = 0, j = 0;
6         while (i < s.size() && j < t.size()) {
7             if (s[i] == t[j]) {
8                 ++i; ++j;
9             } else {
10                 ++j;
11             }
12         }
13         return i == s.size();
14     }
15 };

```

下面这种写法稍稍简洁了一些，但是思路并没有什么不同，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool isSubsequence(string s, string t) {
4         if (s.empty()) return true;
5         int i = 0, j = 0;
6         while (i < s.size() && j < t.size()) {
7             if (s[i] == t[j]) ++i;
8             ++j;
9         }
10         return i == s.size();
11     }
12 };

```

393. 编码验证

A character in UTF8 can be from 1 to 4 bytes long, subjected to the following rules:

For 1-byte character, the first bit is a 0, followed by its unicode code.

For n-bytes character, the first n-bits are all one's, the n+1 bit is 0, followed by n-1 bytes with most significant 2 bits being 10.

This is how the UTF-8 encoding would work:

Char. number range (hexadecimal)	UTF-8 octet sequence (binary)
0000 0000-0000 007F	0xxxxxxxx
0000 0080-0000 07FF	110xxxxx 10xxxxxx
0000 0800-0000 FFFF	1110xxxx 10xxxxxx 10xxxxxx
0001 0000-0010 FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

Given an array of integers representing the data, return whether it is a valid utf-8 encoding.

Note:

The input is an array of integers. Only the least significant 8 bits of each integer is used to store the data. This means each integer represents only 1 byte of data.

这道题考察我们UTF-8编码，这种互联网所采用的通用的编码格式的产生是为了解决ASCII只能表示英文字符的局限性，和统一Unicode的实现方式。下面这段摘自维基百科UTF-8编码：

对于UTF-8编码中的任意字节B，如果B的第一位为0，则B独立的表示一个字符(ASCII码)；
 如果B的第一位为1，第二位为0，则B为一个多字节字符中的一个字节(非ASCII字符)；
 如果B的前两位为1，第三位为0，则B为两个字节表示的字符中的第一个字节；
 如果B的前三位为1，第四位为0，则B为三个字节表示的字符中的第一个字节；
 如果B的前四位为1，第五位为0，则B为四个字节表示的字符中的第一个字节；
 因此，对UTF-8编码中的任意字节，根据第一位，可判断是否为ASCII字符；根据前二位，可判断该字节是否为一个字符编码的第一个字节；根据前四位（如果前两位均为1），可确定该字节为字符编码的第一个字节，并且可判断对应的字符由几个字节表示；
 根据前五位（如果前四位为1），可判断编码是否有错误或数据传输过程中是否有错误。

那么根据上面的描述，我们可以先来判断第一位，如果是0的话，则说明是ASCII码，我们直接跳过，判断方法是只要比二进制数10000000小的数第一位肯定是0，然后我们来处理第一位是1的情况，由于第一位的1只是个标识符，后面连续跟的1的个数才是表示后面的字节的个数，我们可以统一从第一位开始连续1的个数，然后减去1就是后面的字节的个数，我的办法是如果该数字大于等于128，则表示第一位是1，然后减去128，如果得到的数大于等于64，则表示第二位是1，依次类推就可以得到连续的个数，我们要注意10000000这个数是不合法的，遇到了直接返回false。我们得到了cnt的个数，只要验证后面的字节是否是以10开头的数即可，验证方法也很简单，只要这个数在10000000 ~ 10111111范围之间，则一定是10开头的，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     bool validUtf8(vector<int>& data) {
4         for (int i = 0; i < data.size(); ++i) {
5             if (data[i] < 0b10000000) {
6                 continue;
7             } else {
8                 int cnt = 0, val = data[i];
9                 for (int j = 7; j >= 1; --j) {
10                     if (val >= pow(2, j)) ++cnt;
11                     else break;
12                     val -= pow(2, j);
13                 }
14                 if (cnt == 1) return false;
15                 for (int j = i + 1; j < i + cnt; ++j) {
16                     if (data[j] > 0b10111111 || data[j] < 0b10000000) return false;
17                 }
18                 i += cnt - 1;
19             }
20         }
21         return true;
22     }
23 }
```

CPP

在论坛里看到了一种非常简洁的方法，大神就是大神啊，这种方法也是要记连续1的个数，如果是标识字节，先将其向右平移五位，如果得到110，则说明后面跟了一个字节，否则向右平移四位，如果得到1110，则说明后面跟了两个字节，否则向右平移三位，如果得到11110，则说明后面跟了三个字节，否则向右平移七位，如果为1的话，说明是10000000这种情况，不能当标识字节，直接返回false。在非标识字节中，向右平移六位，如果得到的不是10，则说明不是以10开头的，直接返回false，否则cnt自减1，成功完成遍历返回true，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool validUtf8(vector<int>& data) {
4         int cnt = 0;
5         for (int d : data) {
6             if (cnt == 0) {
7                 if ((d >> 5) == 0b110) cnt = 1;
8                 else if ((d >> 4) == 0b1110) cnt = 2;
9                 else if ((d >> 3) == 0b11110) cnt = 3;
10                else if (d >> 7) return false;
11            } else {
12                if ((d >> 6) != 0b10) return false;
13                --cnt;
14            }
15        }
16        return cnt == 0;
17    }
18 };

```

394. 解码字符串

Given an encoded string, return it's decoded string.

The encoding rule is: k[encoded_string], where the encoded_string inside the square brackets is being repeated exactly k times. Note that k is guaranteed to be a positive integer.

You may assume that the input string is always valid; No extra white spaces, square brackets are well-formed, etc.

Furthermore, you may assume that the original data does not contain any digits and that digits are only for those repeat numbers, k. For example, there won't be input like 3a or 2[4].

Examples:

s = "3[a]2[bc]", return "aaabcbc".

这道题让我们把一个按一定规则编码后的字符串解码成其原来的模样，编码的方法很简单，就是把重复的字符串放在一个中括号里，把重复的次数放在中括号的前面，注意中括号里面有可能会嵌套中括号，这题可以用递归和迭代两种方法来解，我们首先来看递归的解法，我们把一个中括号中的所有内容看做一个整体，一次递归函数返回一对中括号中解码后的字符串。给定的编码字符串实际上只有四种字符，数字，字母，左中括号，和右中括号。那么我们开始用一个变量i从0开始遍历到字符串的末尾，由于左中括号都是跟在数字后面，所以我们首先遇到的字符只能是数字或者字母，如果是字母，我们直接存入结果中，如果是数字，我们循环读入所有的数字，并正确转换，那么下一位非数字的字符一定是左中括号，我们指针右移跳过左中括号，对之后的内容调用递归函数求解，注意我们循环的停止条件是遍历到末尾和遇到右中括号，由于递归调用的函数返回了子中括号里解码后的字符串，而我们之前把次数也已经求出来了，那么循环添加到结果中即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     string decodeString(string s) {
4         int i = 0;
5         return decode(s, i);
6     }
7     string decode(string s, int& i) {
8         string res = "";
9         int n = s.size();
10        while (i < n && s[i] != ']') {
11            if (s[i] < '0' || s[i] > '9') {
12                res += s[i++];
13            } else {
14                int cnt = 0;
15                while (i < n && s[i] >= '0' && s[i] <= '9') {
16                    cnt = cnt * 10 + s[i++]- '0';
17                }
18                ++i;
19                string t = decode(s, i);
20                ++i;
21                while (cnt-- > 0) {
22                    res += t;
23                }
24            }
25        }
26        return res;
27    }
28};

```

我们也可以用迭代的方法写出来，当然需要用stack来辅助运算，我们用两个stack，一个用来保存个数，一个用来保存字符串，我们遍历输入字符串，如果遇到数字，我们更新计数变量cnt；如果遇到左中括号，我们把当前cnt压入数字栈中，把当前t压入字符串栈中；如果遇到右中括号时，我们取出数字栈中顶元素，存入变量k，然后给字符串栈的顶元素循环加上k个t字符串，然后取出顶元素存入字符串t中；如果遇到字母，我们直接加入字符串t中即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     string decodeString(string s) {
4         string res = "", t = "";
5         stack<int> s_num;
6         stack<string> s_str;
7         int cnt = 0;
8         for (int i = 0; i < s.size(); ++i) {
9             if (s[i] >= '0' && s[i] <= '9') {
10                 cnt = 10 * cnt + s[i] - '0';
11             } else if (s[i] == '[') {
12                 s_num.push(cnt);
13                 s_str.push(t);
14                 cnt = 0; t.clear();
15             } else if (s[i] == ']') {
16                 int k = s_num.top(); s_num.pop();
17                 for (int j = 0; j < k; ++j) s_str.top() += t;
18                 t = s_str.top(); s_str.pop();
19             } else {
20                 t += s[i];
21             }
22         }
23         return s_str.empty() ? t : s_str.top();
24     }
25 };

```

395. 至少有K个重复字符的最长子字符串

Find the length of the longest substring T of a given string (consists of lowercase letters only) such that every character in T appears no less than k times.

Example 1:

Input:

s = "aaabb", k = 3

Output:

3

The longest substring is "aaa", as 'a' is repeated 3 times.

这道题给了我们一个字符串s和一个正整数k，让我们求一个最大子字符串并且每个字符必须至少出现k次。作为LeetCode第三次编程比赛的压轴题目，博主再一次没有做出来，虽然难度标识只是Medium。后来在网上膜拜学习了大神们的解法，发现我当时的没做出来的主要是卡在了如何快速的判断某一个字符串是否所有的元素都已经满足了至少出现k次这个条件，虽然我也用哈希表建立了字符和其出现次数之间的映射，但是如果每一次都要遍历哈希表中的所有字符看其出现次数是否大于k，未免有些不高效。而用mask就很好的解决了这个问题，由于字母只有26个，而整型mask有32位，足够用了，每一位代表一个字母，如果为1，表示该字母不够k次，如果为0就表示已经出现了k次，这种思路真是太聪明了，隐约记得这种用法在之前的题目中也用过，但是博主并不能举一反三(沮丧脸:O，还得继续努力啊。我们遍历字符串，对于每一个字符，我们都将其视为起点，然后遍历到末尾，我们增加哈希表中字母的出现次数，如果其小于k，我们将mask的对应位改为1，如果大于等于k，将mask对应位改为0。然后看mask是否为0，是的话就更新res结果，然后把当前满足要求的子字符串的起始位置j保存到max_idx中，等内层循环结束后，将外层循环变量i赋值为max_idx+1，继续循环直至结束，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     int longestSubstring(string s, int k) {
4         int res = 0, i = 0, n = s.size();
5         while (i + k <= n) {
6             int m[26] = {0}, mask = 0, max_idx = i;
7             for (int j = i; j < n; ++j) {
8                 int t = s[j] - 'a';
9                 ++m[t];
10                if (m[t] < k) mask |= (1 << t);
11                else mask &= (~(1 << t));
12                if (mask == 0) {
13                    res = max(res, j - i + 1);
14                    max_idx = j;
15                }
16            }
17            i = max_idx + 1;
18        }
19        return res;
20    }
21 };

```

下面这种写法是上面的解法的递归写法，看起来简洁了不少，但是个人感觉比较难想，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int longestSubstring(string s, int k) {
4         int n = s.size(), max_idx = 0, res = 0;
5         int m[128] = {0};
6         bool ok = true;
7         for (char c : s) ++m[c];
8         for (int i = 0; i < n; ++i) {
9             if (m[s[i]] < k) {
10                 res = max(res, longestSubstring(s.substr(max_idx, i - max_idx), k));
11                 ok = false;
12                 max_idx = i + 1;
13             }
14         }
15         return ok ? n : max(res, longestSubstring(s.substr(max_idx, n - max_idx), k));
16     }
17 };

```

396. 旋转函数

Given an array of integers A and let n to be its length.

Assume Bk to be an array obtained by rotating the array A k positions clock-wise, we define a "rotation function" F on A as follow:

$$F(k) = 0 * B_k[0] + 1 * B_k[1] + \dots + (n-1) * B_k[n-1].$$

Calculate the maximum value of F(0), F(1), ..., F(n-1).

Note:

n is guaranteed to be less than 105.

Example:

$$A = [4, 3, 2, 6]$$

$$F(0) = (0 * 4) + (1 * 3) + (2 * 2) + (3 * 6) = 0 + 3 + 4 + 18 = 25$$

$$F(1) = (0 * 6) + (1 * 4) + (2 * 3) + (3 * 2) = 0 + 4 + 6 + 6 = 16$$

$$F(2) = (0 * 2) + (1 * 6) + (2 * 4) + (3 * 3) = 0 + 6 + 8 + 9 = 23$$

$$F(3) = (0 * 3) + (1 * 2) + (2 * 6) + (3 * 4) = 0 + 2 + 12 + 12 = 26$$

So the maximum value of F(0), F(1), F(2), F(3) is F(3) = 26.

这道题是LeetCode第四次比赛的第一道题，博主第一道题就没有做出来，博主写了个O(n^2)的方法并不能通过OJ的大数据集合，后来网上看大家的解法都是很好的找到了规律，可以在O(n)时间内完成。现在想想找规律的能力真的挺重要，比如之前那道 Elimination Game也靠找规律，而用傻方法肯定超时，然后博主发现自己脑子不够活，很难想到正确的方法，说出来全是泪啊 T.T。好了，来解题吧，我们为了找规律，先把具体的数字抽象为A,B,C,D，那么我们可以得到：

$$F(0) = 0A + 1B + 2C + 3D$$

$$F(1) = 0D + 1A + 2B + 3C$$

$$F(2) = 0C + 1D + 2A + 3B$$

$$F(3) = 0B + 1C + 2D + 3A$$

那么，我们通过仔细观察，我们可以得出下面的规律：

$$F(1) = F(0) + \text{sum} - 4D$$

$$F(2) = F(1) + \text{sum} - 4C$$

$$F(3) = F(2) + \text{sum} - 4B$$

那么我们就找到规律了， $F(i) = F(i-1) + \text{sum} - n * A[n-i]$ ，可以写出代码如下：

```

1 class Solution {
2 public:
3     int maxRotateFunction(vector<int>& A) {
4         int t = 0, sum = 0, n = A.size();
5         for (int i = 0; i < n; ++i) {
6             sum += A[i];
7             t += i * A[i];
8         }
9         int res = t;
10        for (int i = 1; i < n; ++i) {
11            t = t + sum - n * A[n - i];
12            res = max(res, t);
13        }
14        return res;
15    }
16 };

```

397. 整数替换

Given a positive integer n and you can do operations as follow:

If n is even, replace n with $n/2$.

If n is odd, you can replace n with either $n + 1$ or $n - 1$.

What is the minimum number of replacements needed for n to become 1?

Example 1:

Input:

8

Output:

3

Explanation:

8 -> 4 -> 2 -> 1

这道题给了我们一个整数 n , 然后让我们通过变换变为1, 如果 n 是偶数, 我们变为 $n/2$, 如果是奇数, 我们可以变为 $n+1$ 或 $n-1$, 让我们求变为1的最少步骤。那么一看道题的要求, 就会感觉应该用递归很合适, 我们直接按照规则写出递归即可, 注意由于有 $n+1$ 的操作, 所以当 n 为INT_MAX的时候, 就有可能溢出, 所以我们可以先将 n 转为长整型, 然后再进行运算, 参见代码如下:

解法1:

```

1 class Solution {
2 public:
3     int integerReplacement(int n) {
4         if (n == 1) return 0;
5         if (n % 2 == 0) return 1 + integerReplacement(n / 2);
6         else {
7             long long t = n;
8             return 2 + min(integerReplacement((t + 1) / 2), integerReplacement((t - 1) /
9             2));
10        }
11    }
12 };

```

我们也可以使用迭代的解法，那么这里就有小技巧了，当n为奇数的时候，我们什么时候应该加1，什么时候应该减1呢，通过观察来说，除了3和7意外，所有加1就变成4的倍数的奇数，适合加1运算，比如15：

15 -> 16 -> 8 -> 4 -> 2 -> 1

15 -> 14 -> 7 -> 6 -> 3 -> 2 -> 1

对于7来说，加1和减1的结果相同，我们可以不用管，对于3来说，减1的步骤小，所以我们需要去掉这种情况。那么我们如何知道某个数字加1后是否是4的倍数呢，我们可以用个小技巧，由于我们之前判定其是奇数了，那么最右边一位肯定是1，如果其右边第二位也是1的话，那么进行加1运算，进位后右边肯定会出现两个0，则一定是4的倍数，搞定。如果之前判定是偶数，那么除以2即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int integerReplacement(int n) {
4         long long t = n;
5         int cnt = 0;
6         while (t > 1) {
7             ++cnt;
8             if (t & 1) {
9                 if ((t & 2) && (t != 3)) ++t;
10                else --t;
11            } else {
12                t >>= 1;
13            }
14        }
15        return cnt;
16    }
17 };

```

CPP

398. 随机拾取序列

Given an array of integers with possible duplicates, randomly output the index of a given target number. You can assume that the given target number must exist in the array.

Note:

The array size can be very large. Solution that uses too much extra space will not pass the judge.

这道题指明了我们不能用太多的空间，那么省空间的随机方法只有水塘抽样Reservoir Sampling了，LeetCode之前有过两道需要用这种方法的题目Shuffle an Array和Linked List Random Node。那么如果了解了水塘抽样，这道题就不算一道难题了，我们定义两个变量，计数器cnt和返回结果res，我们遍历整个数组，如果数组的值不等于target，直接跳过；如果等于target，计数器加1，然后我们在[0,cnt)范围内随机生成一个数字，如果这个数字是0，我们将res赋值为i即可，参见代码如下：

```

1 class Solution {
2 public:
3     Solution(vector<int> nums): v(nums) {}
4
5     int pick(int target) {
6         int cnt = 0, res = -1;
7         for (int i = 0; i < v.size(); ++i) {
8             if (v[i] != target) continue;
9             ++cnt;
10            if (rand() % cnt == 0) res = i;
11        }
12        return res;
13    }
14 private:
15     vector<int> v;
16 };

```

399. 求除法表达式的值

Equations are given in the format $A / B = k$, where A and B are variables represented as strings, and k is a real number (floating point number). Given some queries, return the answers. If the answer does not exist, return -1.0.

Example:

Given $a / b = 2.0$, $b / c = 3.0$.
 queries are: $a / c = ?$, $b / a = ?$, $a / e = ?$, $a / a = ?$, $x / x = ?$.
 return [6.0, 0.5, -1.0, 1.0, -1.0].

The input is: `vector<pair<string, string>> equations, vector<double>& values, vector<pair<string, string>> queries`, where `equations.size() == values.size()`, and the values are positive. This represents the equations. Return `vector<double>`.

According to the example above:

```

equations = [ ["a", "b"], ["b", "c"] ],
values = [2.0, 3.0],
queries = [ ["a", "c"], ["b", "a"], ["a", "e"], ["a", "a"], ["x", "x"] ].

```

这道题作为第四次编程比赛的压轴题，感觉还是挺有难度的，个人感觉难度应该设为hard比较合理。这道题已知条件中给了一些除法等式，然后给了另外一些除法等式，问我们能不能根据已知条件求出结果，不能的用-1表示。问题本身是很简单的数学问题，但是写代码来自动实现就需要我们用正确的数据结构和算法，通过观察题目中的例子，我们可以看出如果需要分析的除法式的除数和被除数如果其中任意一个没有在已知条件中出现过，那么返回结果-1，所以我们在分析已知条件的时候，可以使用set来记录所有出现过的字符串，然后我们在分析其他除法式的时候，可以使用递归来做。通过分析得出，不能直接由已知条件得到的情况主要有下面三种：

- 1) 已知: $a / b = 2$, $b / c = 3$, 求 a / c
- 2) 已知: $a / c = 2$, $b / c = 3$, 求 a / b
- 3) 已知: $a / b = 2$, $a / c = 3$, 求 b / c

在递归函数中，我们有一个需要分析的除法表达式，我们遍历所有的已知条件，如果跟某一个已知表达式相等，直接返回结果，或者跟某一个已知表达式正好相反，那么返回已知表达式结果的倒数即可。如果都没有的话，那么就需要间接寻找了，我们需要一个vector来记录已经访问过的表达式，我们先看待求表达式的被除数和当前遍历到的已知表达式的被除数是否相同如果相同，那么就是上面的第一种情况，我们就可以把待求表达式的被输出换成已知表达式的除数，比如要求 a/c 就换成了求 b/c ，而求 b/c 的过程就可以调用递归函数来求解，结果要乘以 a/b 的值。如果算出来是正数我们直接返回，如果是非正数说明没有找到。对于上面的第一种情况，如果我们要求 c/a ，那么上面的方法就没法开始查找，所以我们同时也要看待求表达式的除数和当前遍历到的已知表达式的被除数是否相同，后面的处理方法都相同，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     vector<double> calcEquation(vector<pair<string, string>> equations, vector<double>&
4 values, vector<pair<string, string>> queries) {
5         vector<double> res(queries.size(), -1);
6         set<string> s;
7         for (auto a : equations) {
8             s.insert(a.first);
9             s.insert(a.second);
10        }
11        for (int i = 0; i < queries.size(); ++i) {
12            vector<string> query{queries[i].first, queries[i].second};
13            if (s.count(query[0]) && s.count(query[1])) {
14                vector<int> v;
15                res[i] = helper(equations, values, query, v);
16            }
17        }
18        return res;
19    }
20    double helper(vector<pair<string, string>> equations, vector<double>& values,
21 vector<string> query, vector<int>& v) {
22        for (int i = 0; i < equations.size(); ++i) {
23            if (equations[i].first == query[0] && equations[i].second == query[1]) return
values[i];
25            if (equations[i].first == query[1] && equations[i].second == query[0]) return
26 1.0 / values[i];
27        }
28        for (int i = 0; i < equations.size(); ++i) {
29            if (find(v.begin(), v.end(), i) == v.end() && equations[i].first == query[0]) {
30                v.push_back(i);
31                double t = values[i] * helper(equations, values, {equations[i].second,
32 query[1]}, v);
33                if (t > 0) return t;
34                else v.pop_back();
35            }
36            if (find(v.begin(), v.end(), i) == v.end() && equations[i].second == query[0])
37        {
38                v.push_back(i);
39                double t = helper(equations, values, {equations[i].first, query[1]}, v) /
40 values[i];
41                if (t > 0) return t;
42                else v.pop_back();
43            }
44        }
45        return -1.0;
46    }
47 };

```

此题还有迭代的写法，用邻接列表的表示方法建立了一个图，然后进行bfs搜索，需要用queue来辅助运算，参见代码如下：

解法2:

```

1 class Solution {
2 public:
3     vector<double> calcEquation(vector<pair<string, string>> equations, vector<double>&
4     values, vector<pair<string, string>> queries) {
5         vector<double> res;
6         unordered_map<string, unordered_map<string, double>> g;
7         for (int i = 0; i < equations.size(); ++i) {
8             g[equations[i].first].emplace(equations[i].second, values[i]);
9             g[equations[i].first].emplace(equations[i].first, 1.0);
10            g[equations[i].second].emplace(equations[i].first, 1.0 / values[i]);
11            g[equations[i].second].emplace(equations[i].second, 1.0);
12        }
13        for (auto query : queries) {
14            if (!g.count(query.first) || !g.count(query.second)) res.push_back(-1.0);
15            else {
16                queue<pair<string, double>> q;
17                unordered_set<string> used{query.first};
18                bool find = false;
19                q.push({query.first, 1.0});
20                while (!q.empty() && !find) {
21                    queue<pair<string, double>> next;
22                    while (!q.empty() && !find) {
23                        pair<string, double> t = q.front(); q.pop();
24                        if (t.first == query.second) {
25                            find = true;
26                            res.push_back(t.second);
27                            break;
28                        }
29                        for (auto a : g[t.first]) {
30                            if (!used.count(a.first)) {
31                                a.second *= t.second;
32                                next.push(a);
33                                used.insert(a.first);
34                            }
35                        }
36                    }
37                    q = next;
38                }
39                if (!find) res.push_back(-1.0);
40            }
41        }
42        return res;
43    }
44};

```

400. 第N位

Find the nth digit of the infinite integer sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...

Note:

n is positive and will fit within the range of a 32-bit signed integer (n < 231).

Example 1:

Input:

3

Output:

3

这道题还是蛮有创意的一道题，是说自然数序列看成一个长字符串，问我们第N位上的数字是什么。那么这道题的关键就是要找出第N位所在的数字，然后可以把数字转为字符串，这样直接可以访问任何一位。那么我们首先来分析自然数序列和其位数的关系，前九个数都是1位的，然后10到99总共90个数字都是两位的，100到999这900个数都是三位的，那么这就很有规律了，我们可以定义个变量cnt，初始化为9，然后每次循环扩大10倍，再用一个变量len记录当前循环区间数字的位数，另外再需要一个变量start用来记录当前循环区间的第一个数字，我们n每次循环都减去len*cnt（区间总位数），当n落到某一个确定的区间里了，那么(n-1)/len就是目标数字在该区间里的坐标，加上start就是得到了目标数字，然后我们将目标数字start转为字符串，(n-1)%len就是所要求的目标位，最后别忘了考虑int溢出问题，我们干脆把所有变量都申请为长整型的好了，参见代码如下：

```
1 class Solution {
2 public:
3     int findNthDigit(int n) {
4         long long len = 1, cnt = 9, start = 1;
5         while (n > len * cnt) {
6             n -= len * cnt;
7             ++len;
8             cnt *= 10;
9             start *= 10;
10        }
11        start += (n - 1) / len;
12        string t = to_string(start);
13        return t[(n - 1) % len] - '0';
14    }
15};
```

401. 二进制表

A binary watch has 4 LEDs on the top which represent the hours (0-11), and the 6 LEDs on the bottom represent the minutes (0-59).

Each LED represents a zero or one, with the least significant bit on the right.

For example, the above binary watch reads "3:25".

Given a non-negative integer n which represents the number of LEDs that are currently on, return all possible times the watch could represent.

Example:

Input: n = 1
Return: ["1:00", "2:00", "4:00", "8:00", "0:01", "0:02", "0:04", "0:08", "0:16", "0:32"]

这道题考察我们二进制表，说实话，博主对二进制表无感，感觉除了装b没啥其他的作用，谁会看个时间还要算半天啊，但是这并不影响我们做题，我们首先来看一种写法很简洁的解法，这种解法利用到了bitset这个类，可以将任意进制数转为二进制，而且又用到了count函数，用来统计1的个数。那么时针从0遍历到11，分针从0遍历到59，然后我们把时针的数组左移6位加上分针的数值，然后统计1的个数，即为亮灯的个数，我们遍历所有的情况，当其等于num的时候，存入结果res中，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<string> readBinaryWatch(int num) {
4         vector<string> res;
5         for (int h = 0; h < 12; ++h) {
6             for (int m = 0; m < 60; ++m) {
7                 if (bitset<10>((h << 6) + m).count() == num) {
8                     res.push_back(to_string(h) + (m < 10 ? ":0" : ":") + to_string(m));
9                 }
10            }
11        }
12        return res;
13    }
14 };

```

上面的方法之所以那么简洁是因为用了bitset这个类，如果我们不用这个类，那么应该怎么做呢？这个灯亮问题的本质其实就是在n个数字中取出k个，那么就跟之前的那道Combinations一样，我们可以借鉴那道题的解法，那么思路是，如果总共要取num个，我们在小时集合里取i个，算出和，然后在分钟集合里去num-i个求和，如果两个都符合题意，那么加入结果中即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<string> readBinaryWatch(int num) {
4         vector<string> res;
5         vector<int> hour{8, 4, 2, 1}, minute{32, 16, 8, 4, 2, 1};
6         for (int i = 0; i <= num; ++i) {
7             vector<int> hours = generate(hour, i);
8             vector<int> minutes = generate(minute, num - i);
9             for (int h : hours) {
10                 if (h > 11) continue;
11                 for (int m : minutes) {
12                     if (m > 59) continue;
13                     res.push_back(to_string(h) + (m < 10 ? ":0" : ":") + to_string(m));
14                 }
15             }
16         }
17         return res;
18     }
19     vector<int> generate(vector<int>& nums, int cnt) {
20         vector<int> res;
21         helper(nums, cnt, 0, 0, res);
22         return res;
23     }
24     void helper(vector<int>& nums, int cnt, int pos, int out, vector<int>& res) {
25         if (cnt == 0) {
26             res.push_back(out);
27             return;
28         }
29         for (int i = pos; i < nums.size(); ++i) {
30             helper(nums, cnt - 1, i + 1, out + nums[i], res);
31         }
32     }
33 };

```

下面这种方法就比较搞笑了，是博主在没法想出上面两种方法的情况下万般无奈使用的，你个二进制表再叼也就72种情况，全给你列出来，然后采用跟上面那种解法相同的思路，时针集合取k个，分针集合取num-k个，然后存入结果中即可，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     vector<string> readBinaryWatch(int num) {
4         vector<vector<int>> hours{{0},{1,2,4,8},{3,5,9,6,10},{7,11}};
5         vector<vector<int>> minutes{{0},{1,2,4,8,16,32},
6 {3,5,9,17,33,6,10,18,34,12,20,36,24,40,48},
7 {7,11,19,35,13,21,37,25,41,49,14,22,38,26,42,50,28,44,52,56},
8 {15,23,39,27,43,51,29,45,53,57,30,46,54,58},{31,47,55,59}};
9         vector<string> res;
10        for (int k = 0; k <= num; ++k) {
11            int t = num - k;
12            if (k > 3 || t > 5) continue;
13            for (int i = 0; i < hours[k].size(); ++i) {
14                for (int j = 0; j < minutes[t].size(); ++j) {
15                    string str = minutes[t][j] < 10 ? "0" + to_string(minutes[t][j]) :
16 to_string(minutes[t][j]);
17                    res.push_back(to_string(hours[k][i]) + ":" + str);
18                }
19            }
20        }
21        return res;
22    }
23 };

```

402. 去掉K位数字

Given a non-negative integer num represented as a string, remove k digits from the number so that the new number is the smallest possible.

Note:

The length of num is less than 10002 and will be $\geq k$.
The given num does not contain any leading zero.

这道题让我们将给定的数字去掉k位，要使得留下来的数字最小，这题跟LeetCode上之前那道Create Maximum Number有些类似，可以借鉴其中的思路，如果n是num的长度，我们要去除k个，那么需要剩下n-k个，我们开始遍历给定数字num的每一位，对于当前遍历到的数字c，进行如下while循环，如果res不为空，且k大于0，且res的最后一位大于c，那么我们应该将res的最后一位移去，且k自减1。当跳出while循环后，我们将c加入res中，最后我们将res的大小重设为n-k。根据题目中的描述，可能会出现"0200"这样不符合要求的情况，所以我们用一个while循环来去掉前面的所有0，然后返回时判断是否为空，为空则返回"0"，参见代码如下：

```

1 class Solution {
2 public:
3     string removeKdigits(string num, int k) {
4         string res = "";
5         int n = num.size(), keep = n - k;
6         for (char c : num) {
7             while (k && res.size() && res.back() > c) {
8                 res.pop_back();
9                 --k;
10            }
11            res.push_back(c);
12        }
13        res.resize(keep);
14        while (!res.empty() && res[0] == '0') res.erase(res.begin());
15        return res.empty() ? "0" : res;
16    }
17 };

```

403. 青蛙过河

A frog is crossing a river. The river is divided into x units and at each unit there may or may not exist a stone. The frog can jump on a stone, but it must not jump into the water.

Given a list of stones' positions (in units) in sorted ascending order, determine if the frog is able to cross the river by landing on the last stone. Initially, the frog is on the first stone and assume the first jump must be 1 unit.

If the frog's last jump was k units, then its next jump must be either $k - 1$, k , or $k + 1$ units. Note that the frog can only jump in the forward direction.

Note:

The number of stones is ≥ 2 and is $< 1,100$.

Each stone's position will be a non-negative integer $< 2^{31}$.

The first stone's position is always 0.

终于等到青蛙过河问题了，一颗赛艇。题目中说青蛙如果上一次跳了 k 距离，那么下一次只能跳 $k-1$, k , 或 $k+1$ 的距离，那么青蛙跳到某个石头上可能有多种跳法，由于这道题只是让我们判断青蛙是否能跳到最后一个石头上，并没有让我们返回所有的路径，这样就降低了一些难度。我们可以用递归来做，我们维护一个哈希表，建立青蛙在 pos 位置和拥有 $jump$ 跳跃能力时是否能跳到对岸。为了能用一个变量同时表示 pos 和 $jump$ ，我们可以将 $jump$ 左移很多位并或上 pos ，由于题目中对于位置大小有限制，所以不会产生冲突。我们还是首先判断 pos 是否已经到最后一个石头了，是的话直接返回true；然后看当前这种情况是否已经出现在哈希表中，是的话直接从哈希表中取结果。如果没有，我们就遍历余下的所有石头，对于遍历到的石头，我们计算到当前石头的距离 $dist$ ，如果距离小于 $jump-1$ ，我们接着遍历下一块石头；如果 $dist$ 大于 $jump+1$ ，说明无法跳到下一块石头， $m[key]$ 赋值为false，并返回false；如果在青蛙能跳到的范围内，我们调用递归函数，以新位置 i 为 pos ，距离 $dist$ 为 $jump$ ，如果返回true了，我们给 $m[key]$ 赋值为true，并返回true。如果结束遍历我们给 $m[key]$ 赋值为false，并返回false，参加代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool canCross(vector<int>& stones) {
4         unordered_map<int, bool> m;
5         return helper(stones, 0, 0, m);
6     }
7     bool helper(vector<int>& stones, int pos, int jump, unordered_map<int, bool>& m) {
8         int n = stones.size(), key = pos | jump << 11;
9         if (pos >= n - 1) return true;
10        if (m.count(key)) return m[key];
11        for (int i = pos + 1; i < n; ++i) {
12            int dist = stones[i] - stones[pos];
13            if (dist < jump - 1) continue;
14            if (dist > jump + 1) return m[key] = false;
15            if (helper(stones, i, dist, m)) return m[key] = true;
16        }
17        return m[key] = false;
18    }
19 };

```

我们也可以用迭代的方法来解，用一个哈希表来建立每个石头和在该位置上能跳的距离之间的映射，建立一个一维dp数组，其中 $dp[i]$ 表示在位置为*i*的石头青蛙的弹跳力(只有青蛙能跳到该石头上， $dp[i]$ 才大于0)，由于题目中规定了第一个石头上青蛙跳的距离必须是1，为了跟后面的统一，我们对青蛙在第一块石头上的弹跳力初始化为0(虽然为0，但是由于题目上说青蛙最远能到其弹跳力+1的距离，所以仍然可以到达第二块石头)。我们用变量k表示当前石头，然后开始遍历剩余的石头，对于遍历到的石头*i*，我们来找到刚好能跳到*i*上的石头k，如果*i*和*k*的距离大于青蛙在*k*上的弹跳力+1，则说明青蛙在*k*上到不了*i*，则*k*自增1。我们从*k*遍历到*i*，如果青蛙能从中间某个石头上跳到*i*上，我们更新石头*i*上的弹跳力和最大弹跳力。这样当循环完成后，我们只要检查最后一个石头上青蛙的最大弹跳力是否大于0即可，参见代码如下：

解法2：

```

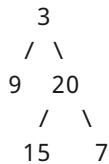
1 class Solution {
2 public:
3     bool canCross(vector<int>& stones) {
4         unordered_map<int, unordered_set<int>> m;
5         vector<int> dp(stones.size(), 0);
6         m[0].insert(0);
7         int k = 0;
8         for (int i = 1; i < stones.size(); ++i) {
9             while (dp[k] + 1 < stones[i] - stones[k]) ++k;
10            for (int j = k; j < i; ++j) {
11                int t = stones[i] - stones[j];
12                if (m[j].count(t - 1) || m[j].count(t) || m[j].count(t + 1)) {
13                    m[i].insert(t);
14                    dp[i] = max(dp[i], t);
15                }
16            }
17        }
18        return dp.back() > 0;
19    }
20 };

```

404. 左子叶之和

Find the sum of all left leaves in a given binary tree.

Example:



There are two left leaves in the binary tree, with values 9 and 15 respectively. Return 24.

这道题让我们求一棵二叉树的所有左子叶的和，那么看到这道题我们知道这肯定是考二叉树的遍历问题，那么最简洁的写法肯定是用递归，由于我们只需要累加左子叶之和，那么我们在进入递归函数的时候需要知道当前结点是否是左子节点，如果是左子节点，而且该左子节点再没有子节点了说明其是左子叶，那么我们将其值加入结果res中，我们用一个bool型的变量，如果为true说明当前结点是左子节点，若为false则说明是右子节点，不做特殊处理，整个来说就是个递归的先序遍历的写法，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int sumOfLeftLeaves(TreeNode* root) {
4         if (!root || (!root->left && !root->right)) return 0;
5         int res = 0;
6         helper(root->left, true, res);
7         helper(root->right, false, res);
8         return res;
9     }
10    void helper(TreeNode* node, bool left, int& res) {
11        if (!node) return;
12        if (!node->left && !node->right && left) res += node->val;
13        helper(node->left, true, res);
14        helper(node->right, false, res);
15    }
16 }
  
```

CPP

我们还可以写的更简洁一些，不需要写其他的函数，直接在原函数中检查当前节点的左子节点是否是左子叶，如果是的话，则返回左子叶的值加上对当前结点的右子节点调用递归的结果；如果不是的话，我们对左右子节点分别调用递归函数，返回二者之和，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int sumOfLeftLeaves(TreeNode* root) {
4         if (!root) return 0;
5         if (root->left && !root->left->left && !root->left->right) {
6             return root->left->val + sumOfLeftLeaves(root->right);
7         }
8         return sumOfLeftLeaves(root->left) + sumOfLeftLeaves(root->right);
9     }
10 }
  
```

CPP

我们也可以使用迭代来解，因为这道题的本质是遍历二叉树，所以我们可以用层序遍历的迭代写法，利用queue来辅助，注意对左子叶的判断和处理，参见代码如下：

解法3：

```
1 class Solution {
2 public:
3     int sumOfLeftLeaves(TreeNode* root) {
4         if (!root || (!root->left && !root->right)) return 0;
5         int res = 0;
6         queue<TreeNode*> q;
7         q.push(root);
8         while (!q.empty()) {
9             TreeNode *t = q.front(); q.pop();
10            if (t->left && !t->left->left && !t->left->right) res += t->left->val;
11            if (t->left) q.push(t->left);
12            if (t->right) q.push(t->right);
13        }
14        return res;
15    }
16};
```

我们也可以用stack来辅助，对比上面的解法，我们发现几乎一模一样，只是把queue换成了stack，但实际上遍历的顺序不同，这种方法是先序遍历的迭代写法，参见代码如下：

解法4：

```
1 class Solution {
2 public:
3     int sumOfLeftLeaves(TreeNode* root) {
4         if (!root || (!root->left && !root->right)) return 0;
5         int res = 0;
6         stack<TreeNode*> s;
7         s.push(root);
8         while (!s.empty()) {
9             TreeNode *t = s.top(); s.pop();
10            if (t->left && !t->left->left && !t->left->right) res += t->left->val;
11            if (t->left) s.push(t->left);
12            if (t->right) s.push(t->right);
13        }
14        return res;
15    }
16};
```

405. 数字转为十六进制

Given an integer, write an algorithm to convert it to hexadecimal. For negative integer, two's complement method is used.

Note:

All letters in hexadecimal (a-f) must be in lowercase.

The hexadecimal string must not contain extra leading 0s. If the number is zero, it is represented by a single zero character '0'; otherwise, the first character in the hexadecimal string will not be the zero character.

The given number is guaranteed to fit within the range of a 32-bit signed integer.

You must not use any method provided by the library which converts/formats the number to hex directly.

这道题给了我们一个数字，让我们转化为十六进制，抛开题目，我们应该都会把一个十进制数转为十六进制数，比如50，转为十六进制数，我们先对50除以16，商3余2，那么转为十六进制数就是32。所以我们就按照这个思路来写代码，由于输入数字的大小限制为int型，我们对于负数的处理方法是用其补码来运算，那么数字范围就是0到UINT_MAX，即为 16^{8-1} ，那么最高位就是 16^7 ，我们首先除以这个数字，如果商大于等于10，我们用字母代替，否则就是用数字代替，然后对其余数进行同样的处理，一直到当前数字为0停止，最后我们还要补齐末尾的0，方法根据n的值，比-1大多少就补多少个0。由于题目中说明了最高位不能有多余的0，所以我们将起始0移除，如果res为空了，我们就返回0即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     string toHex(int num) {
4         string res = "";
5         vector<string> v{"a", "b", "c", "d", "e", "f"};
6         int n = 7;
7         unsigned int x = num;
8         if (num < 0) x = UINT_MAX + num + 1;
9         while (x > 0) {
10             int t = pow(16, n);
11             int d = x / t;
12             if (d >= 10) res += v[d - 10];
13             else if (d >= 0) res += to_string(d);
14             x %= t;
15             --n;
16         }
17         while (n-- >= 0) res += to_string(0);
18         while (!res.empty() && res[0] == '0') res.erase(res.begin());
19         return res.empty() ? "0" : res;
20     }
21 };

```

CPP

上述方法稍稍复杂一些，我们来看一种更简洁的方法，我们采取位操作的思路，每次取出最右边四位，如果其大于等于10，找到对应的字母加入结果，反之则将对应的数字加入结果，然后num像右平移四位，循环停止的条件是num为0，或者是已经循环了7次，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     string toHex(int num) {
4         string res = "";
5         for (int i = 0; num && i < 8; ++i) {
6             int t = num & 0xf;
7             if (t >= 10) res = char('a' + t - 10) + res;
8             else res = char('0' + t) + res;
9             num >>= 4;
10        }
11        return res.empty() ? "0" : res;
12    }
13 };

```

下面这种写法更加简洁一些，虽然思路跟解法二并没有什么区别，但是我们把要转换的十六进制的数字字母都放在一个字符串中，按位置直接取就可以了，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     string toHex(int num) {
4         string res = "", str = "0123456789abcdef";
5         int cnt = 0;
6         while (num != 0 && cnt++ < 8) {
7             res = str[(num & 0xf)] + res;
8             num >>= 4;
9         }
10        return res.empty() ? "0" : res;
11    }
12 };

```

406. 根据高度重建队列

Suppose you have a random list of people standing in a queue. Each person is described by a pair of integers(h, k), where h is the height of the person and k is the number of people in front of this person who have a height greater than or equal to h . Write an algorithm to reconstruct the queue.

Note:

The number of people is less than 1,100.

Example

Input:

`[[7,0], [4,4], [7,1], [5,0], [6,1], [5,2]]`

Output:

`[[5,0], [7,0], [5,2], [6,1], [4,4], [7,1]]`

这道题给了我们一个队列，队列中的每个元素是一个pair，分别为身高和前面身高不低于当前身高的人的个数，让我们重新排列队列，使得每个pair的第二个参数都满足题意。首先我们来看一种超级简洁的方法，不得不膜拜想出这种解法的大神。首先我们给队列先排个序，按照身高高的排前面，如果身高相同，则第二个数小的排前面。然后我们新建一个空的数组，遍历之前排好序的数组，然后根据每个元素的第二个数字，将其插入到res数组中对应的位置，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     vector<pair<int, int>> reconstructQueue(vector<pair<int, int>>& people) {
4         sort(people.begin(), people.end(), [](const pair<int, int>& a, const pair<int,
5 int>& b) {
6             return a.first > b.first || (a.first == b.first && a.second < b.second);
7         });
8         vector<pair<int, int>> res;
9         for (auto a : people) {
10            res.insert(res.begin() + a.second, a);
11        }
12        return res;
13    }
14};

```

CPP

上面那种方法是简洁，但是用到了额外空间，我们来看一种不使用额外空间的解法，这种方法没有使用vector自带的insert或者erase函数，而是通过一个变量cnt和k的关系来将元素向前移动到正确位置，移动到方法是通过每次跟前面的元素交换位置，使用题目中给的例子来演示过程：

[[7,0], [4,4], [7,1], [5,0], [6,1], [5,2]]

排序后：

[[7,0], [7,1], [6,1], [5,0], [5,2], [4,4]]

交换顺序：

[[7,0], [6,1], [7,1], [5,0], [5,2], [4,4]]

[[5,0], [7,0], [6,1], [7,1], [5,2], [4,4]]

[[5,0], [7,0], [5,2], [6,1], [7,1], [4,4]]

[[5,0], [7,0], [5,2], [6,1], [4,4], [7,1]]

解法2：

```

1 class Solution {
2 public:
3     vector<pair<int, int>> reconstructQueue(vector<pair<int, int>>& people) {
4         sort(people.begin(), people.end(), [](<const pair<int, int>& a, <const pair<int,
5 int>& b) {
6             return a.first > b.first || (a.first == b.first && a.second < b.second);
7         });
8         for (int i = 1; i < people.size(); ++i) {
9             int cnt = 0;
10            for (int j = 0; j < i; ++j) {
11                if (cnt == people[i].second) {
12                    pair<int, int> t = people[i];
13                    for (int k = i - 1; k >= j; --k) {
14                        people[k + 1] = people[k];
15                    }
16                    people[j] = t;
17                    break;
18                }
19                if (people[j].first >= people[i].first) ++cnt;
20            }
21        }
22        return people;
23    }
24};

```

下面这种解法跟解法一很相似，只不过没有使用额外空间，而是直接把位置不对的元素从原数组中删除，直接加入到正确的位置上，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     vector<pair<int, int>> reconstructQueue(vector<pair<int, int>>& people) {
4         sort(people.begin(), people.end(), [](<const pair<int, int> &a, <const pair<int, int>
5 &b) {
6             return a.first > b.first || (a.first == b.first && a.second < b.second);
7         });
8         for (int i = 0; i < people.size(); i++) {
9             auto p = people[i];
10            if (p.second != i) {
11                people.erase(people.begin() + i);
12                people.insert(people.begin() + p.second, p);
13            }
14        }
15        return people;
16    }
17};

```

407. 收集雨水之二

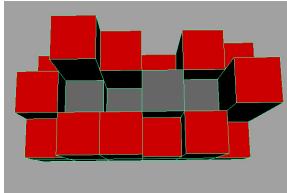
Given an $m \times n$ matrix of positive integers representing the height of each unit cell in a 2D elevation map, compute the volume of water it is able to trap after raining.

Note:

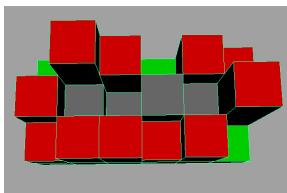
Both m and n are less than 110. The height of each unit cell is greater than 0 and is less than 20,000.

这道题是之前那道Trapping Rain Water的拓展，由2D变3D了，感觉很叼。但其实解法跟之前的完全不同了，之前那道题由于是二维的，我们可以用双指针来做，而这道三维的，我们需要用BFS来做，解法思路很巧妙，下面我们就以题目中的例子来进行分析讲解，多图预警，手机流量党慎入：

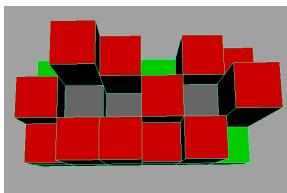
首先我们应该能分析出，能装水的底面肯定不能在边界上，因为边界上的点无法封闭，那么所有边界上的点都可以加入queue，当作BFS的启动点，同时我们需要一个二维数组来标记访问过的点，访问过的点我们用红色来表示，那么如下图所示：



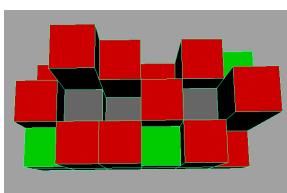
我们再想想，怎么样可以成功的装进去水呢，是不是周围的高度都应该比当前的高度高，形成一个凹槽才能装水，而且装水量取决于周围最小的那个高度，有点像木桶原理的感觉，那么为了模拟这种方法，我们采用模拟海平面上升的方法来做，我们维护一个海平面高度mx，初始化为最小值，从1开始往上升，那么我们BFS遍历的时候就需要从高度最小的格子开始遍历，那么我们的queue就不能使用普通队列了，而是使用优先级队列，将高度小的放在队首，最先取出，这样我们就可以遍历高度为1的三个格子，用绿色标记出来了，如下图所示：



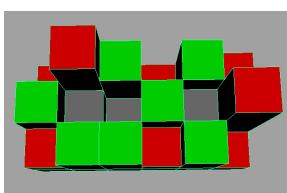
如上图所示，向周围BFS搜索的条件是不能越界，且周围格子未被访问，那么可以看出上面的第一个和最后一个绿格子无法进一步搜索，只有第一行中间那个绿格子可以搜索，其周围有一个灰格子未被访问过，将其加入优先队列queue中，然后标记为红色，如下图所示：



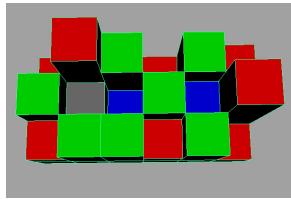
那么优先队列queue中高度为1的格子遍历完了，此时海平面上升1，变为2，此时我们遍历优先队列queue中高度为2的格子，有3个，如下图绿色标记所示：



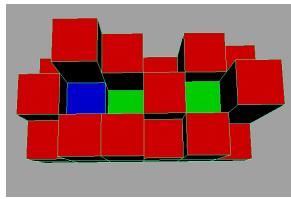
我们发现这三个绿格子周围的格子均已被访问过了，所以不做任何操作，海平面继续上升，变为4，遍历所有高度为4的格子，如下图绿色标记所示：



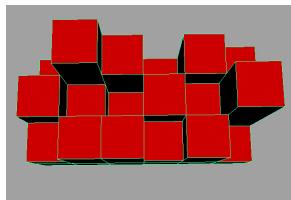
由于我们没有特别声明高度相同的格子在优先队列queue中的顺序，所以应该是随机的，其实谁先遍历到都一样，对结果没啥影响，我们就假设第一行的两个绿格子先遍历到，那么那么周围各有一个灰格子可以遍历，这两个灰格子比海平面低了，可以存水了，把存水量算出来加入结果res中，如下图所示：



上图中这两个遍历到的蓝格子会被加入优先队列queue中，由于它们的高度小，所以下一次从优先队列queue中取格子时，它们会被优先遍历到，那么左边的那个蓝格子进行BFS搜索，就会遍历到其左边的那个灰格子，由于其高度小于海平面，也可以存水，将存水量算出来加入结果res中，如下图所示：



等两个绿格子遍历结束了，它们会被标记为红色，蓝格子遍历会先被标记红色，然后加入优先队列queue中，由于其周围格子全变成红色了，所有不会有任何操作，如下图所示：



此时所有的格子都标记为红色了，海平面继续上升，继续遍历完优先队列queue中的格子，不过已经不会对结果有任何影响了，因为所有的格子都已经访问过了，此时等循环结束后返回res即可，参见代码如下：

```

1 class Solution {
2 public:
3     int trapRainWater(vector<vector<int>>& heightMap) {
4         if (heightMap.empty()) return 0;
5         int m = heightMap.size(), n = heightMap[0].size(), res = 0, mx = INT_MIN;
6         priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>> q;
7         vector<vector<bool>> visited(m, vector<bool>(n, false));
8         vector<vector<int>> dir{{0,-1},{-1,0},{0,1},{1,0}};
9         for (int i = 0; i < m; ++i) {
10             for (int j = 0; j < n; ++j) {
11                 if (i == 0 || i == m - 1 || j == 0 || j == n - 1) {
12                     q.push({heightMap[i][j], i * n + j});
13                     visited[i][j] = true;
14                 }
15             }
16         }
17         while (!q.empty()) {
18             auto t = q.top(); q.pop();
19             int h = t.first, r = t.second / n, c = t.second % n;
20             mx = max(mx, h);
21             for (int i = 0; i < dir.size(); ++i) {
22                 int x = r + dir[i][0], y = c + dir[i][1];
23                 if (x < 0 || x >= m || y < 0 || y >= n || visited[x][y]) continue;
24                 visited[x][y] = true;
25                 if (heightMap[x][y] < mx) res += mx - heightMap[x][y];
26                 q.push({heightMap[x][y], x * n + y});
27             }
28         }
29         return res;
30     }
31 };

```

408. 验证单词缩写

Given a non-empty string s and an abbreviation abbr, return whether the string matches with the given abbreviation.

A string such as "word" contains only the following valid abbreviations:

["word", "1ord", "w1rd", "wo1d", "wor1", "2rd", "w2d", "wo2", "1o1d", "1or1", "w1r1", "1o2", "2r1", "3d", "w3", "4"]

Notice that only the above abbreviations are valid abbreviations of the string "word". Any other string is not a valid abbreviation of "word".

Note:

Assume s contains only lowercase letters and abbr contains only lowercase letters and digits.

这道题让我们验证单词缩写，关于单词缩写LeetCode上还有两道相类似的题目Unique Word Abbreviation和Generalized Abbreviation。这道题给了我们一个单词和一个缩写形式，让我们验证这个缩写形式是否是正确的，由于题目中限定了单词中只有小写字母和数字，所以我们只要对这两种情况分别处理即可。我们使用双指针分别指向两个单词的开头，循环的条件是两个指针都没有到各自的末尾，如果指向缩写单词的指针指的是一个数字的话，如果当前数字是0，返回false，因为数字不能以0开头，然后我们要把该数字整体取出来，所以我们用一个while循环将数字整体取出来，然后指向原单词的指针也要对应的向后移动这么多位数。如果指向缩写单词的指针指的是一个字母的话，那么我们只要比两个指针指向的字母是否相同，不同则返回false，相同则两个指针均向后移动一位，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool validWordAbbreviation(string word, string abbr) {
4         int i = 0, j = 0, m = word.size(), n = abbr.size();
5         while (i < m && j < n) {
6             if (abbr[j] >= '0' && abbr[j] <= '9') {
7                 if (abbr[j] == '0') return false;
8                 int val = 0;
9                 while (j < n && abbr[j] >= '0' && abbr[j] <= '9') {
10                     val = val * 10 + abbr[j++]- '0';
11                 }
12                 i += val;
13             } else {
14                 if (word[i++] != abbr[j++]) return false;
15             }
16         }
17         return i == m && j == n;
18     }
19 };

```

下面这种方法和上面的方法稍有不同，这里是用了一个for循环来遍历缩写单词的所有字符，然后用一个指针p来指向与其对应的原单词的位置，然后cnt表示当前读取查出来的数字，如果读取的是数字，我们先排除首位是0的情况，然后cnt做累加；如果读取的是字母，那么指针p向后移动cnt位，如果p到超过范围了，或者p指向的字符和当前遍历到的缩写单词的字符不相等，则返回false，反之则给cnt置零继续循环，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool validWordAbbreviation(string word, string abbr) {
4         int m = word.size(), n = abbr.size(), p = 0, cnt = 0;
5         for (int i = 0; i < abbr.size(); ++i) {
6             if (abbr[i] >= '0' && abbr[i] <= '9') {
7                 if (cnt == 0 && abbr[i] == '0') return false;
8                 cnt = 10 * cnt + abbr[i] - '0';
9             } else {
10                 p += cnt;
11                 if (p >= m || word[p++] != abbr[i]) return false;
12                 cnt = 0;
13             }
14         }
15         return p + cnt == m;
16     }
17 };

```

409. 最长回文串

Given a string which consists of lowercase or uppercase letters, find the length of the longest palindromes that can be built with those letters.

This is case sensitive, for example "Aa" is not considered a palindrome here.

Note:

Assume the length of given string will not exceed 1,010.

Example:

Input:

"abccccdd"

Output:

7

这又是一道关于回文字符串的问题，LeetCode上关于回文串的题有十来道呢，也算一个比较重要的知识点。但是这道题确实不算一道难题，给了我们一个字符串，让我们找出可以组成的最长的回文串的长度，由于字符顺序可以打乱，所以问题就转化为了求偶数个字符的个数，我们了解回文串的都知道，回文串主要有两种形式，一个是左右完全对称的，比如noon，还有一种是以中间字符为中心，左右对称，比如bob，level等，那么我们统计出来所有偶数个字符的出现总和，然后如果有奇数个字符的话，我们取取出其最大偶数，然后最后结果加1即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int longestPalindrome(string s) {
4         int res = 0;
5         bool mid = false;
6         unordered_map<char, int> m;
7         for (char c : s) ++m[c];
8         for (auto it = m.begin(); it != m.end(); ++it) {
9             res += it->second;
10            if (it->second % 2 == 1) {
11                res -= 1;
12                mid = true;
13            }
14        }
15        return mid ? res + 1 : res;
16    }
17 };

```

CPP

上面那种方法是通过哈希表来建立字符串和其出现次数的映射，这里我们可以换一种思路，来找出所有奇数个的字符，我们采用的方法是使用一个set集合，如果遍历到的字符不在set中，那么就将其加入set，如果已经在set里了，就将其从set中删去，这样遍历完成后set中就是所有出现个数是奇数个的字符了，那么我们最后只要用s的长度减去0和set长度减一之间的较大值即可，为啥这样呢，我们想，如果没有出现个数是奇数个的字符，那么t的长度就是0，减1成了-1，那么s的长度只要减去0即可；如果有奇数个的字符，那么字符个数减1，就是不能组成回文串的字符，因为回文串最多允许一个不成对出现的字符，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int longestPalindrome(string s) {
4         unordered_set<char> t;
5         for (char c : s) {
6             if (!t.count(c)) t.insert(c);
7             else t.erase(c);
8         }
9         return s.size() - max(0, (int)t.size() - 1);
10    }
11 };

```

最后这种方法利用到了STL中的count函数，就是找字符串中某个字符出现的个数，那么我们和1相与，就可以知道该个数是奇数还是偶数了，返回的写法和上面那种方法相同，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int longestPalindrome(string s) {
4         int odds = 0;
5         for (char c = 'A'; c <= 'z'; ++c) {
6             odds += count(s.begin(), s.end(), c) & 1;
7         }
8         return s.size() - max(0, odds - 1);
9     }
10 };

```

410. 分割数组的最大值

Given an array which consists of non-negative integers and an integer m, you can split the array into m non-empty continuous subarrays. Write an algorithm to minimize the largest sum among these m subarrays.

Note:

Given m satisfies the following constraint: $1 \leq m \leq \text{length}(\text{nums}) \leq 14,000$.

Examples:

Input:

nums = [7,2,5,10,8]

m = 2

Output:

18

这道题给了我们一个非负数的数组nums和一个整数m，让我们把数组分割成m个非空的连续子数组，让我们最小化m个子数组中的最大值。开始以为要用博弈论中的最小最大化算法，可是想了半天发现并不会做，于是后面决定采用无脑暴力破解，在nums中取出所有的m个子数组的情况都找一遍最大值，为了加快求子数组和的运算，还建立了累计和数组，可以还是TLE了，所以博主就没有办法了，只能上网参考大神们的解法，发现大家普遍使用了二分搜索法来做，感觉特别巧妙，原来二分搜索法还能这么用，厉害了我的哥。我们首先来分析，如果m和数组nums的个数相等，那么每个数组都是一个子数组，所以返回nums中最大的数字即可，如果m为1，那么整个nums数组就是一个子数组，返回nums所有数字之和，所以对于其他有效的m值，返回的值必定在上面两个值之间，所以我们可以用二分搜索法来做。我们用一个例子来分析， $\text{nums} = [1, 2, 3, 4, 5]$, $m = 3$ ，我们将left设为数组中的最大值5，right设为数字之和15，然后我们算出中间数为10，我们接下来要做的是找出和最大且小于等于10的子数组的个数，[1, 2, 3, 4], [5]，可以看到我们无法分为3组，说明mid偏大，所以我们让right=mid，然后我们再次进行二分查找哦啊，算出mid=7，再次找出和最大且小于等于7的子数组的个数，[1, 2, 3], [4], [5]，我们成功的找出了三组，说明mid还可以

进一步降低，我们让right=mid，然后我们再次进行二分查找哦啊，算出mid=6，再次找出和最大且小于等于6的子数组的个数，[1,2,3], [4], [5]，我们成功的找出了三组，我们尝试着继续降低mid，我们让right=mid，然后我们再次进行二分查找哦啊，算出mid=5，再次找出和最大且小于等于5的子数组的个数，[1,2], [3], [4], [5]，发现有4组，此时我们的mid太小了，应该增大mid，我们让left=mid+1，此时left=6，right=5，循环退出了，我们返回left即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int splitArray(vector<int>& nums, int m) {
4         long long left = 0, right = 0;
5         for (int i = 0; i < nums.size(); ++i) {
6             left = max((int)left, nums[i]);
7             right += nums[i];
8         }
9         while (left < right) {
10            long long mid = left + (right - left) / 2;
11            if (can_split(nums, m, mid)) right = mid;
12            else left = mid + 1;
13        }
14        return left;
15    }
16    bool can_split(vector<int>& nums, int m, int sum) {
17        int cnt = 1, curSum = 0;
18        for (int i = 0; i < nums.size(); ++i) {
19            curSum += nums[i];
20            if (curSum > sum) {
21                curSum = nums[i];
22                ++cnt;
23                if (cnt > m) return false;
24            }
25        }
26        return true;
27    }
28 };

```

上面的解法相对来说比较难想，在热心网友perthblank的提醒下，我们再来看一种DP的解法，相对来说，这种方法应该更容易理解一些。我们建立一个二维数组dp，其中 $dp[i][j]$ 表示将数组中前 j 个数字分成 i 组所能得到的最小的各个子数组中最大值，初始化为整型最大值，如果无法分为 i 组，那么还是保持为整型最大值。为了能快速的算出子数组之和，我们还是要建立累计和数组，难点就是在于要求递推公式了。我们来分析，如果前 j 个数字要分成 i 组，那么 i 的范围是什么，由于只有 j 个数字，如果每个数字都是单独的一组，那么最多有 j 组；如果将整个数组看为一个整体，那么最少有1组，所以 i 的范围是 $[1, j]$ ，所以我们要遍历这中间所有的情况，假如中间任意一个位置 k ， $dp[i-1][k]$ 表示数组中前 k 个数字分成 $i-1$ 组所能得到的最小的各个子数组中最大值，而 $sums[j]-sums[k]$ 就是后面的数字之和，我们取二者之间的较大值，然后和 $dp[i][j]$ 原有值进行对比，更新 $dp[i][j]$ 为二者之中的较小值，这样 k 在 $[1, j]$ 的范围内扫过一遍， $dp[i][j]$ 就能更新到最小值，我们最终返回 $dp[m][n]$ 即可，博主认为这道题所用的思想应该是之前那道题Reverse Pairs中解法二中总结的分割重现关系(Partition Recurrence Relation)，由此看来很多问题的本质都是一样，但是披上华丽的外衣，难免会让人有些眼花缭乱了，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int splitArray(vector<int>& nums, int m) {
4         int n = nums.size();
5         vector<int> sums(n + 1, 0);
6         vector<vector<int>> dp(m + 1, vector<int>(n + 1, INT_MAX));
7         dp[0][0] = 0;
8         for (int i = 1; i <= n; ++i) {
9             sums[i] = sums[i - 1] + nums[i - 1];
10        }
11        for (int i = 1; i <= m; ++i) {
12            for (int j = 1; j <= n; ++j) {
13                for (int k = i - 1; k < j; ++k) {
14                    int val = max(dp[i - 1][k], sums[j] - sums[k]);
15                    dp[i][j] = min(dp[i][j], val);
16                }
17            }
18        }
19        return dp[m][n];
20    }
21 };

```

411. 最短的独一无二的单词缩写

A string such as "word" contains the following abbreviations:

["word", "1ord", "w1rd", "wo1d", "wor1", "2rd", "w2d", "wo2", "1o1d", "1or1", "w1r1", "1o2", "2r1", "3d", "w3", "4"]

Given a target string and a set of strings in a dictionary, find an abbreviation of this target string with the smallest possible length such that it does not conflict with abbreviations of the strings in the dictionary.

Each number or letter in the abbreviation is considered length = 1. For example, the abbreviation "a32bc" has length = 4.

Note:

In the case of multiple answers as shown in the second example below, you may return any one of them.

Assume length of target string = m, and dictionary size = n. You may assume that m ≤ 21, n ≤ 1000, and log2(n) + m ≤ 20.

这道题实际上是之前那两道Valid Word Abbreviation和Generalized Abbreviation的合体，我们的思路其实很简单，首先找出target的所有的单词缩写的形式，然后按照长度来排序，小的排前面，我们用优先队列来自动排序，里面存一个pair，保存单词缩写及其长度，然后我们从最短的单词缩写开始，跟dictionary中所有的单词一一进行验证，利用Valid Word Abbreviation中的方法，看其是否是合法的单词的缩写，如果是，说明有冲突，直接break，进行下一个单词缩写的验证，参见代码如下：

```

1 class Solution {
2 public:
3     string minAbbreviation(string target, vector<string>& dictionary) {
4         if (dictionary.empty()) return to_string((int)target.size());
5         priority_queue<pair<int, string>, vector<pair<int, string>>, greater<pair<int,
6         string>>> q;
7         q = generate(target);
8         while (!q.empty()) {
9             auto t = q.top(); q.pop();
10            bool no_conflict = true;
11            for (string word : dictionary) {
12                if (valid(word, t.second)) {
13                    no_conflict = false;
14                    break;
15                }
16            }
17            if (no_conflict) return t.second;
18        }
19        return "";
20    }
21    priority_queue<pair<int, string>, vector<pair<int, string>>, greater<pair<int,
22    string>>> generate(string target) {
23        priority_queue<pair<int, string>, vector<pair<int, string>>, greater<pair<int,
24        string>>> res;
25        for (int i = 0; i < pow(2, target.size()); ++i) {
26            string out = "";
27            int cnt = 0, size = 0;
28            for (int j = 0; j < target.size(); ++j) {
29                if ((i >> j) & 1) ++cnt;
30                else {
31                    if (cnt != 0) {
32                        out += to_string(cnt);
33                        cnt = 0;
34                        ++size;
35                    }
36                    out += target[j];
37                    ++size;
38                }
39            }
40            if (cnt > 0) {
41                out += to_string(cnt);
42                ++size;
43            }
44            res.push({size, out});
45        }
46        return res;
47    }
48    bool valid(string word, string abbr) {
49        int m = word.size(), n = abbr.size(), p = 0, cnt = 0;
50        for (int i = 0; i < abbr.size(); ++i) {
51            if (abbr[i] >= '0' && abbr[i] <= '9') {
52                if (cnt == 0 && abbr[i] == '0') return false;
53                cnt = 10 * cnt + abbr[i] - '0';
54            } else {
55                p += cnt;
56                if (p >= m || word[p++] != abbr[i]) return false;
57                cnt = 0;
58            }
59        }
60    }
61 }
```

```

        return p + cnt == m;
    }
};

```

412. 嘶嘶嗡嗡

Write a program that outputs the string representation of numbers from 1 to n.

But for multiples of three it should output "Fizz" instead of the number and for the multiples of five output "Buzz". For numbers which are multiples of both three and five output "FizzBuzz".

Example:

n = 15,

Return:

```
[
    "1",
    "2",
    "Fizz",
    "4",
    "Buzz",
    "Fizz",
    "7",
    "8",
    "Fizz",
    "Buzz",
    "11",
    "Fizz",
    "13",
    "14",
    "FizzBuzz"
]
```

这道题真心没有什么可讲的，就是分情况处理就行了。

```

1 class Solution {
2 public:
3     vector<string> fizzBuzz(int n) {
4         vector<string> res;
5         for (int i = 1; i <= n; ++i) {
6             if (i % 15 == 0) res.push_back("FizzBuzz");
7             else if (i % 3 == 0) res.push_back("Fizz");
8             else if (i % 5 == 0) res.push_back("Buzz");
9             else res.push_back(to_string(i));
10        }
11        return res;
12    }
13 };

```

CPP

413. 算数切片

A sequence of number is called arithmetic if it consists of at least three elements and if the difference between any two consecutive elements is the same.

For example, these are arithmetic sequence:

```
1, 3, 5, 7, 9
7, 7, 7, 7
3, -1, -5, -9
```

The following sequence is not arithmetic.

```
1, 1, 2, 5, 7
```

A zero-indexed array A consisting of N numbers is given. A slice of that array is any pair of integers (P, Q) such that $0 \leq P < Q < N$.

A slice (P, Q) of array A is called arithmetic if the sequence:

$A[P], A[p + 1], \dots, A[Q - 1], A[Q]$ is arithmetic. In particular, this means that $P + 1 < Q$.

The function should return the number of arithmetic slices in the array A.

这道题让我们算一种算数切片，说白了就是找等差数列，限定了等差数列的长度至少为3，那么 $[1, 2, 3, 4]$ 含有3个长度至少为3的算数切片，我们再来看 $[1, 2, 3, 4, 5]$ 有多少个呢：

```
len = 3: [1,2,3], [2,3,4], [3,4,5]
len = 4: [1,2,3,4], [2,3,4,5]
len = 5: [1,2,3,4,5]
```

那么我们可以找出递推式，长度为n的等差数列中含有长度至少为3的算数切片的个数为 $(n-1)(n-2)/2$ ，那么题目就变成了找原数组中等差数列的长度，然后带入公式去算个数即可，参见代码如下：

解法1:

```
1 class Solution {
2 public:
3     int numberOfArithmeticSlices(vector<int>& A) {
4         int res = 0, len = 2, n = A.size();
5         for (int i = 2; i < n; ++i) {
6             if (A[i] - A[i - 1] == A[i - 1] - A[i - 2]) {
7                 ++len;
8             } else {
9                 if (len > 2) res += (len - 1) * (len - 2) * 0.5;
10                len = 2;
11            }
12        }
13        if (len > 2) res += (len - 1) * (len - 2) * 0.5;
14        return res;
15    }
16};
```

CPP

我们还可以用DP来做，定义一个一维dp数组，其中 $dp[i]$ 表示，到i位置为止的算数切片的个数，那么我们从第三个数字开始遍历，如果当前数字和之前两个数字构成算数切片，那么我们更新 $dp[i]$ 为 $dp[i-1]+1$ ，然后res累加上 $dp[i]$ 的值即可：

解法2:

```

1 class Solution {
2 public:
3     int numberOfArithmeticSlices(vector<int>& A) {
4         int res = 0, n = A.size();
5         vector<int> dp(n, 0);
6         for (int i = 2; i < n; ++i) {
7             if (A[i] - A[i - 1] == A[i - 1] - A[i - 2]) {
8                 dp[i] = dp[i - 1] + 1;
9             }
10            res += dp[i];
11        }
12        return res;
13    }
14 };

```

我们还可以进一步优化空间，用一个变量来代替上面的数组，原理都一样，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int numberOfArithmeticSlices(vector<int>& A) {
4         int res = 0, cur = 0;
5         for (int i = 2; i < A.size(); ++i) {
6             if (A[i] - A[i - 1] == A[i - 1] - A[i - 2]) {
7                 cur += 1;
8                 res += cur;
9             } else {
10                 cur = 0;
11             }
12         }
13         return res;
14     }
15 };

```

414. 第三大的数

Given a non-empty array of integers, return the third maximum number in this array. If it does not exist, return the maximum number. The time complexity must be in O(n).

Example 1:

Input: [3, 2, 1]

Output: 1

Explanation: The third maximum is 1.

这道题让我们求数组中第三大的数，如果不存在的话那么就返回最大的数，题目中说明了这里的第三大不能和第二大相同，必须是严格的小于，而并非小于等于。这道题并不是很难，如果知道怎么求第二大的数，那么求第三大的数的思路都是一样的。那么我们用三个变量first, second, third来分别保存第一大，第二大，和第三大的数，然后我们遍历数组，如果遍历到的数字大于当前第一大的数first，那么三个变量各自错位赋值，如果当前数字大于second，小于first，那么就更新second和third，如果当前数字大于third，小于second，那就只更新third，注意这里有个坑，就是初始化要用长整型long的最小值，否则当数组中有INT_MIN存在时，程序就不知道该返回INT_MIN还是最大值first了，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     int thirdMax(vector<int>& nums) {
4         long first = LONG_MIN, second = LONG_MIN, third = LONG_MIN;
5         for (int num : nums) {
6             if (num > first) {
7                 third = second;
8                 second = first;
9                 first = num;
10            } else if (num > second && num < first) {
11                third = second;
12                second = num;
13            } else if (num > third && num < second) {
14                third = num;
15            }
16        }
17        return (third == LONG_MIN || third == second) ? first : third;
18    }
19 };

```

CPP

下面这种方法的时间复杂度是 $O(nlgn)$, 不符合题目要求, 纯粹是拓宽下思路哈, 利用了set的自动排序和自动去重复项的特性, 很好的解决了问题, 对于遍历到的数字, 加入set中, 重复项就自动去掉了, 如果此时set大小大于3个了, 那么我们把set的第一个元素去掉, 也就是将第四大的数字去掉, 那么就可以看出set始终维护的是最大的三个不同的数字, 最后遍历结束后, 我们看set的大小是否为3, 是的话就返回首元素, 不是的话就返回尾元素, 参见代码如下:

解法2:

```

1 class Solution {
2 public:
3     int thirdMax(vector<int>& nums) {
4         set<int> s;
5         for (int num : nums) {
6             s.insert(num);
7             if (s.size() > 3) {
8                 s.erase(s.begin());
9             }
10        }
11        return s.size() == 3 ? *s.begin() : *s.rbegin();
12    }
13 };

```

CPP

415. 字符串相加

Given two non-negative numbers num1 and num2 represented as string, return the sum of num1 and num2.

Note:

The length of both num1 and num2 is < 5100 .

Both num1 and num2 contains only digits 0-9.

Both num1 and num2 does not contain any leading zero.

You must not use any built-in BigInteger library or convert the inputs to integer directly.

这道题让我们求两个字符串的相加，之前LeetCode出过几道类似的题目，比如二进制数相加，还有链表相加，或是字符串加1，基本思路很类似，都是一位一位相加，然后算和算进位，最后根据进位情况看需不需要补一个高位，难度不大，参见代码如下：

```

1 class Solution {
2     public:
3         string addStrings(string num1, string num2) {
4             string res = "";
5             int m = num1.size(), n = num2.size(), i = m - 1, j = n - 1, carry = 0;
6             while (i >= 0 || j >= 0) {
7                 int a = i >= 0 ? num1[i--] - '0' : 0;
8                 int b = j >= 0 ? num2[j--] - '0' : 0;
9                 int sum = a + b + carry;
10                res.insert(res.begin(), sum % 10 + '0');
11                carry = sum / 10;
12            }
13            return carry ? "1" + res : res;
14        }
15    };

```

416. 相同子集和分割

Given a non-empty array containing only positive integers, find if the array can be partitioned into two subsets such that the sum of elements in both subsets is equal.

Note:

Both the array size and each of the array element will not exceed 100.

Example 1:

Input: [1, 5, 11, 5]

Output: true

Explanation: The array can be partitioned as [1, 5, 5] and [11].

这道题给了我们一个数组，问我们这个数组能不能分成两个非空子集合，使得两个子集合的元素之和相同。那么我们想，原数组所有数字和一定是偶数，不然根本无法拆成两个和相同的子集合，那么我们只需要算出原数组的数字之和，然后除以2，就是我们的target，那么问题就转换为能不能找到一个非空子集合，使得其数字之和为target。开始我想的是遍历所有子集合，算和，但是这种方法无法通过OJ的大数据集合。于是乎，动态规划 Dynamic Programming 就是我们的不二之选。我们定义一个一维的dp数组，其中dp[i]表示数字i是否是原数组的任意个子集合之和，那么我们最后只需要返回dp[target]就行了。我们初始化dp[0]为true，由于题目中限制了所有数字为正数，那么我们就不用担心会出现和为0或者负数的情况。那么关键问题就是要找出状态转移方程了，我们需要遍历原数组中的数字，对于遍历到的每个数字nums[i]，我们需要更新dp数组，要更新[nums[i], target]之间的值，那么对于这个区间中的任意一个数字j，如果dp[j - nums[i]]为true的话，那么dp[j]就一定为true，于是状态转移方程如下：

$dp[j] = dp[j] \mid\mid dp[j - nums[i]] \quad (nums[i] \leq j \leq target)$

有了状态转移方程，那么我们就可以写出代码了，这里需要特别注意的是，第二个for循环一定要从target遍历到nums[i]，而不能反过来，想想为什么呢？因为如果我们从nums[i]遍历到target的话，假如nums[i]=1的话，那么[1, target]中所有的dp值都是true，因为dp[0]是true，dp[1]会或上dp[0]，为true，dp[2]会或上dp[1]，为true，依此类推，完全使我们的dp数组失效了，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool canPartition(vector<int>& nums) {
4         int sum = accumulate(nums.begin(), nums.end(), 0), target = sum >> 1;
5         if (sum & 1) return false;
6         vector<bool> dp(target + 1, false);
7         dp[0] = true;
8         for (int num : nums) {
9             for (int i = target; i >= num; --i) {
10                 dp[i] = dp[i] || dp[i - num];
11             }
12         }
13         return dp[target];
14     }
15 };

```

这道题还可以用bitset来做，感觉也十分的巧妙，bisets的大小设为5001，为啥呢，因为题目中说了数组的长度和每个数字的大小都不会超过100，那么最大的和为10000，那么一半就是5000，前面再加上个0，就是5001了。我们初始化把最低位赋值为1，然后我们算出数组之和，然后我们遍历数字，对于遍历到的数字num，我们把bits向左平移num位，然后再或上原来的bits，这样所有的可能出现的和位置上都为1。举个例子来说吧，比如对于数组[2,3]来说，初始化bits为1，然后对于数字2，bits变为101，我们可以看出来bits[2]标记为了1，然后遍历到3，bits变为了101101，我们看到bits[5],bits[3],bits[2]都分别为1了，正好代表了可能的和2，3，5，这样我们遍历玩整个数组后，去看bits[sum >> 1]是否为1即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool canPartition(vector<int>& nums) {
4         bitset<5001> bits(1);
5         int sum = accumulate(nums.begin(), nums.end(), 0);
6         for (int num : nums) bits |= bits << num;
7         return (sum % 2 == 0) && bits[sum >> 1];
8     }
9 };

```

417. 太平洋大西洋水流

Given an $m \times n$ matrix of non-negative integers representing the height of each unit cell in a continent, the "Pacific ocean" touches the left and top edges of the matrix and the "Atlantic ocean" touches the right and bottom edges.

Water can only flow in four directions (up, down, left, or right) from a cell to another one with height equal or lower.

Find the list of grid coordinates where water can flow to both the Pacific and Atlantic ocean.

Note:

The order of returned grid coordinates does not matter.

Both m and n are less than 150.

Example:

Given the following 5x5 matrix:

Pacific	~	~	~	~	~	
~	1	2	2	3	(5)	*
~	3	2	3	(4)	(4)	*
~	2	4	(5)	3	1	*
~	(6)	(7)	1	4	5	*
~	(5)	1	1	2	4	*
*	*	*	*	*	*	Atlantic

Return:

`[[0, 4], [1, 3], [1, 4], [2, 2], [3, 0], [3, 1], [4, 0]]` (positions with parentheses in above matrix).

这道题给了我们一个二维数组，说是数组的左边和上边是太平洋，右边和下边是大西洋，假设水能从高处向低处流，问我们所有能流向两大洋的点的集合。刚开始我们没有理解题意，以为加括号的点是一条路径，连通两大洋的，但是看来看去感觉也不太对，后来终于明白了，是每一个点单独都路径来通向两大洋。那么就是典型的搜索问题，那么我最开始想的是对于每个点来搜索是否能到达边缘，只不过搜索的目标点不在是一个单点，而是所有的边缘点，找这种思路写出的代码无法通过OJ大数据集，那么我们就要想办法来优化我们的代码，优化的方法跟之前那道Surrounded Regions很类似，都是换一个方向考虑问题，既然从每个点像中间扩散会TLE，那么我们从边缘当作起点开始遍历搜索，然后标记能到达的点为true，分别标记出pacific和atlantic能到达的点，那么最终能返回的点就是二者均为true的点。我们可以先用DFS来遍历二维数组，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<pair<int, int>> pacificAtlantic(vector<vector<int>>& matrix) {
4         if (matrix.empty() || matrix[0].empty()) return {};
5         vector<pair<int, int>> res;
6         int m = matrix.size(), n = matrix[0].size();
7         vector<vector<bool>> pacific(m, vector<bool>(n, false));
8         vector<vector<bool>> atlantic(m, vector<bool>(n, false));
9         for (int i = 0; i < m; ++i) {
10             dfs(matrix, pacific, INT_MIN, i, 0);
11             dfs(matrix, atlantic, INT_MIN, i, n - 1);
12         }
13         for (int i = 0; i < n; ++i) {
14             dfs(matrix, pacific, INT_MIN, 0, i);
15             dfs(matrix, atlantic, INT_MIN, m - 1, i);
16         }
17         for (int i = 0; i < m; ++i) {
18             for (int j = 0; j < n; ++j) {
19                 if (pacific[i][j] && atlantic[i][j]) {
20                     res.push_back({i, j});
21                 }
22             }
23         }
24     }
25     return res;
26 }
27 void dfs(vector<vector<int>>& matrix, vector<vector<bool>>& visited, int pre, int i,
28 int j) {
29     int m = matrix.size(), n = matrix[0].size();
30     if (i < 0 || i >= m || j < 0 || j >= n || visited[i][j] || matrix[i][j] < pre)
31     return;
32     visited[i][j] = true;
33     dfs(matrix, visited, matrix[i][j], i + 1, j);
34     dfs(matrix, visited, matrix[i][j], i - 1, j);
35     dfs(matrix, visited, matrix[i][j], i, j + 1);
36     dfs(matrix, visited, matrix[i][j], i, j - 1);
37 }
38 };

```

那么BFS的解法也可以做，用queue来辅助，开始把边上的点分别存入queue中，然后对应的map标记true，然后开始BFS遍历，遍历结束后还是找pacific和atlantic均标记为true的点加入res中返回即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<pair<int, int>> pacificAtlantic(vector<vector<int>>& matrix) {
4         if (matrix.empty() || matrix[0].empty()) return {};
5         vector<pair<int, int>> res;
6         int m = matrix.size(), n = matrix[0].size();
7         queue<pair<int, int>> q1, q2;
8         vector<vector<bool>> pacific(m, vector<bool>(n, false)), atlantic = pacific;
9         for (int i = 0; i < m; ++i) {
10             q1.push({i, 0});
11             q2.push({i, n - 1});
12             pacific[i][0] = true;
13             atlantic[i][n - 1] = true;
14         }
15         for (int i = 0; i < n; ++i) {
16             q1.push({0, i});
17             q2.push({m - 1, i});
18             pacific[0][i] = true;
19             atlantic[m - 1][i] = true;
20         }
21         bfs(matrix, pacific, q1);
22         bfs(matrix, atlantic, q2);
23         for (int i = 0; i < m; ++i) {
24             for (int j = 0; j < n; ++j) {
25                 if (pacific[i][j] && atlantic[i][j]) {
26                     res.push_back({i, j});
27                 }
28             }
29         }
30     }
31     return res;
32 }
33 void bfs(vector<vector<int>>& matrix, vector<vector<bool>>& visited, queue<pair<int, int>>& q) {
34     int m = matrix.size(), n = matrix[0].size();
35     vector<vector<int>> dirs{{0,-1},{-1,0},{0,1},{1,0}};
36     while (!q.empty()) {
37         auto t = q.front(); q.pop();
38         for (auto dir : dirs) {
39             int x = t.first + dir[0], y = t.second + dir[1];
40             if (x < 0 || x >= m || y < 0 || y >= n || visited[x][y] || matrix[x][y] <
41 matrix[t.first][t.second]) continue;
42             visited[x][y] = true;
43             q.push({x, y});
44         }
45     }
46 };

```

418. 调整屏幕上的句子

Given a $rows \times cols$ screen and a sentence represented by a list of words, find how many times the given sentence can be fitted on the screen.

Note:

A word cannot be split into two lines.
The order of words in the sentence must remain unchanged.
Two consecutive words in a line must be separated by a single space.
Total words in the sentence won't exceed 100.
Length of each word won't exceed 10.
 $1 \leq rows, cols \leq 20,000$.

这道题给我们了一个句子，由若干个单词组成，然后给我们了一个空白屏幕区域，让我们填充单词，前提是单词和单词之间需要一个空格隔开，而且单词不能断开，如果当前行剩余位置放下不下某个单词，则必须将该单词整个移动到下一行。我刚开始想的是便利句子，每个单词分别处理，但是这种做法很不高效，因为有可能屏幕的宽度特别大，而单词可能就一两个，那么我们这样遍历的话就太浪费时间了，应该直接用宽度除以句子加上空格的长度之和，可以快速的得到能装下的个数。下面这种方法设计的很巧妙，思路是用start变量来记录下能装下的句子的总长度，最后除以一个句子的长度，就可以得到个数。而句子的总长度的求法时要在每个单词后面加上一个空格(包括最后一个单词)，我们遍历屏幕的每一行，然后每次start都加上宽度，然后看all[start%len]是否为空格，是的话就start加1，这样做的好处是可以处理末尾是没有空格的情况，比如宽度为1，只有一个单词a，那么我们都知道是这样放的a，start变为1，len是2，all[start%len]是空格，所以start自增1，变成2，这样我们用start/len就知道能放下几个了。对于all[start%len]不为空格的情况，如果all[(start-1)%len]也不为空格，那么start就自减1，进行while循环，直至其为空格为止。大家可以自己带例子尝试，个人觉得想出此方法的人真是太聪明了：

解法1：

```

1 class Solution {
2 public:
3     int wordsTyping(vector<string>& sentence, int rows, int cols) {
4         string all = "";
5         for (string word : sentence) all += (word + " ");
6         int start = 0, len = all.size();
7         for (int i = 0; i < rows; ++i) {
8             start += cols;
9             if (all[start % len] == ' ') {
10                 ++start;
11             } else {
12                 while (start > 0 && all[(start - 1) % len] != ' ') {
13                     --start;
14                 }
15             }
16         }
17         return start / len;
18     }
19 };

```

CPP

下面这种方法也是很棒，同样也需要统计加空格的句子总长度，然后遍历每一行，初始化colsRemaining为cols，然后还需要一个变量idx，来记录当前单词的位置，如果colsRemaining大于0，就进行while循环，如果当前单词的长度小于等于colsRemaining，说明可以放下该单词，那么就减去该单词的长度就是剩余的空间，然后如果此时colsRemaining仍然大于0，则减去空格的长度1，然后idx自增1，如果idx此时超过单词个数的范围了，说明一整句可以放下，那么就有可能出现宽度远大于句子长度的情况，所以我们加上之前放好的一句之外，还要加上colsRemaining/len的个数，然后colsRemaining%len是剩余的位置，此时idx重置为0，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int wordsTyping(vector<string>& sentence, int rows, int cols) {
4         string all = "";
5         for (string word : sentence) all += (word + " ");
6         int res = 0, idx = 0, n = sentence.size(), len = all.size();
7         for (int i = 0; i < rows; ++i) {
8             int colsRemaining = cols;
9             while (colsRemaining > 0) {
10                 if (sentence[idx].size() <= colsRemaining) {
11                     colsRemaining -= sentence[idx].size();
12                     if (colsRemaining > 0) colsRemaining -= 1;
13                     if (++idx >= n) {
14                         res += (1 + colsRemaining / len);
15                         colsRemaining %= len;
16                         idx = 0;
17                     }
18                 } else {
19                     break;
20                 }
21             }
22         }
23         return res;
24     }
25 };

```

419. 平板上的战船

Given an 2D board, count how many different battleships are in it. The battleships are represented with 'X's, empty slots are represented with '.'s. You may assume the following rules:

You receive a valid board, made of only battleships or empty slots.

Battleships can only be placed horizontally or vertically. In other words, they can only be made of the shape $1 \times N$ (1 row, N columns) or $N \times 1$ (N rows, 1 column), where N can be of any size. At least one horizontal or vertical cell separates between two battleships - there are no adjacent battleships.

Example:

X..X

...X

...X

In the above board there are 2 battleships.

Invalid Example:

...X

XXXX

...X

This is not a valid board - as battleships will always have a cell separating between them.

Your algorithm should not modify the value of the board.

这道题好像之前在地里面见过，忘了是哪家公司的面试题了，现在被LeetCode收录了，感觉现在LeetCode更新越来越快了，感觉要成为第一大题库了，赞一个👍。这道题让我们求战舰的个数，所谓的战舰就是只能是一行或者一列，不能有拐弯。这道题降低了难度的做法是限定了不会有相邻的两个战舰的存在，有了这一点限制，那么我们只需要遍历一次二维数组就行了，只要找

出战舰的起始点。所谓的战舰起始点，就是为X的点，而且该点的上方和左边的点不能为X，所以我们只要找出所有满足这个条件的点即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int countBattleships(vector<vector<char>>& board) {
4         if (board.empty() || board[0].empty()) return 0;
5         int res = 0, m = board.size(), n = board[0].size();
6         for (int i = 0; i < m; ++i) {
7             for (int j = 0; j < n; ++j) {
8                 if (board[i][j] == '.') || (i > 0 && board[i - 1][j] == 'X') || (j > 0 &&
9                     board[i][j - 1] == 'X')) continue;
10                ++res;
11            }
12        }
13        return res;
14    }
};
```

CPP

然而我自己在做的时候并没有注意到题目中限制了两艘战舰不能相邻的情况，我加上了处理方法，首先我算出来了所有连续X的区域的个数，方法跟之前那道Number of Islands一样，稍有不同的是，我分别记录下来每一个连续区域的i和j，把所有的点的横纵坐标分别或了起来，这样做好处是如果是在一条直线上的战舰，那么所有点肯定是要么横坐标都相同，要么纵坐标都相同，所以最后我们检测如果横纵坐标的累积或都跟之前的i和j不同的话，那么一定不是题目中定义的战舰，那么我们就不累加结果res，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int countBattleships(vector<vector<char>>& board) {
4         if (board.empty() || board[0].empty()) return 0;
5         int m = board.size(), n = board[0].size(), res = 0;
6         vector<vector<bool>> visited(m, vector<bool>(n, false));
7         for (int i = 0; i < m; ++i) {
8             for (int j = 0; j < n; ++j) {
9                 if (board[i][j] == 'X' && !visited[i][j]) {
10                     int vertical = 0, horizontal = 0;
11                     dfs(board, visited, vertical, horizontal, i, j);
12                     if (vertical == i || horizontal == j) ++res;
13                 }
14             }
15         }
16         return res;
17     }
18     void dfs(vector<vector<char>>& board, vector<vector<bool>>& visited, int& vertical,
19     int& horizontal, int i, int j) {
20         int m = board.size(), n = board[0].size();
21         if (i < 0 || i >= m || j < 0 || j >= n || visited[i][j] || board[i][j] == '.')
22             return;
23         vertical |= i; horizontal |= j;
24         visited[i][j] = true;
25         dfs(board, visited, vertical, horizontal, i - 1, j);
26         dfs(board, visited, vertical, horizontal, i + 1, j);
27         dfs(board, visited, vertical, horizontal, i, j - 1);
28         dfs(board, visited, vertical, horizontal, i, j + 1);
29     }
30 };

```

既然DFS能实现，那么BFS就应该没啥问题，这里完全按题目的要求，默认两个战舰不会相邻，并没有添加解法二中的过滤条件，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int countBattleships(vector<vector<char>>& board) {
4         if (board.empty() || board[0].empty()) return 0;
5         int res = 0, m = board.size(), n = board[0].size();
6         vector<vector<bool>> visited(m, vector<bool>(n, false));
7         vector<vector<int>> dirs{{0,-1},{-1,0},{0,1},{1,0}};
8         for (int i = 0; i < m; ++i) {
9             for (int j = 0; j < n; ++j) {
10                 if (board[i][j] == 'X' && !visited[i][j]) {
11                     ++res;
12                     queue<pair<int, int>> q;
13                     q.push({i, j});
14                     while (!q.empty()) {
15                         auto t = q.front(); q.pop();
16                         visited[t.first][t.second] = true;
17                         for (auto dir : dirs) {
18                             int x = t.first + dir[0], y = t.second + dir[1];
19                             if (x < 0 || x >= m || y < 0 || y >= n || visited[x][y] ||
20                                 board[x][y] == '.') continue;
21                             q.push({x, y});
22                         }
23                     }
24                 }
25             }
26         }
27         return res;
28     }
29 };

```

420. 密码强度检查器

A password is considered strong if below conditions are all met:

It has at least 6 characters and at most 20 characters.

It must contain at least one lowercase letter, at least one uppercase letter, and at least one digit.

It must NOT contain three repeating characters in a row ("...aaa..." is weak, but "...aa...a..." is strong, assuming other conditions are met).

Write a function `strongPasswordChecker(s)`, that takes a string `s` as input, and return the MINIMUM change required to make `s` a strong password. If `s` is already strong, return 0.

Insertion, deletion or replace of any one character are all considered as one change.

这道题给了我们一个密码串，让我们判断其需要多少步修改能变成一个强密码串，然后给定了强密码串的条件，长度为6到20之间，必须含有至少一个小写字母，大写字母，数字，而且不能有连续三个相同的字符，给了我们三种修改方法，任意一个位置加入字符，删除字符，或者是置换任意一个字符，让我们修改最小的次数变成强密码串。这道题定义为Hard真是名副其实，博主光是看大神的帖子都看了好久，这里主要是参考了大神fun4LeetCode的帖子，个人感觉这个算是讲的十分清楚的了，这里就照搬过来吧。首先我们来看非强密码串主要有的三个问题：

1. 长度问题，当长度小于6的时候，我们要通过插入字符来补充长度，当长度超过20时，我们要删除字符。
2. 缺失字符或数字，当我们缺少大写，小写和数字的时候，我们可以通过插入字符或者替换字符的方式来补全。
3. 重复字符，这个也是本题最大的难点，因为插入，删除，或者置换都可以解决重复字符的问题，比如有一个字符串"aaaaa"，我们可以用一次置换，比如换掉中间的字符'a'；或者两次插入字符，在第二个a和第四个a后面分别插入一个非a字符；或者可以删除3个a来解决重复字符的问题。由于题目要求我们要用最少的步骤，那么显而易见置换是最高效的去重复字符的方法。

我们通过举例观察可以知道这三种情况并不是相互独立的，一个操作有时候可以解决多个问题，比如字符串"aaa1a"，我们在第二个a后面增加一个'B'，变为"aaBa1a"，这样同时解决了三个问题，即增加了长度，又补充了缺失的大写字母，又去掉了重复，所以我们的目标就是尽可能的找出这种能解决多种问题的操作。由于情况三(重复字符)可以用三种操作来解决，所以我们分别来看能同时解决情况一和情况三，跟同时解决情况二和情况三的操作。对于同时解决情况一和情况二的操作如果原密码串长度小于6会有重叠出现，所以我们要分情况讨论：

当密码串长度小于6时，情况一和情况二的操作步骤可以完全覆盖情况三，这个不难理解，因为这种情况下重复字符个数的范围为[3,5]，如果有三个重复字符，那么增加三个字符的操作可以同时解决重复字符问题("aaa" -> "a1BCaa"；如果有四个重复字符，那么增加二个字符的操作也可以解决重复问题("aaaa" -> "aa1Baa")；如果有五个重复字符，那么增加和置换操作也同时解决重复问题("aaaaa" -> "aa1aaB")。所以我们就专心看最少多少步能同时解决情况一和情况二，首先我们计算出当前密码串需要补几个字符才能到6，补充字符的方法只能用插入字符操作，而插入字符操作也可以解决情况二，所以当情况二的缺失种类数小于等于diff时，我们不用再增加操作，当diff不能完全覆盖缺失种类个数时，我们还应加上二者的差值。

当密码串长度大于等于6个的时候，这种情况就比较复杂了，由于目前字符串的长度只可能超标不可能不达标，所以我们尽量不要用插入字符操作，因为这有可能会使长度超过限制。由于长度的不确定性，所以可能会有大量的重复字符，那么解决情况三就变得很重要了，由于前面的分析，替换字符是最高效的解法，但是这种方法没法解决情况一，因为长度超标了的话，再怎么替换字符，也不会让长度减少，但是我们也不能无脑删除字符，这样不一定能保证是最少步骤，所以在解决情况三的时候还要综合考虑到情况一，这里用到了一个trick（很膜拜大神能想的出来），对于重复字符个数k大于等于3的情况，我们并不是直接将其删除到2个，而是先将其删除到最近的(3m+2)个，那么如果k正好被3整除，那么我们直接变为k-1，如果k除以3余1，那么变为k-2。这样做的好处是3m+2个重复字符可以最高效的用替换m个字符来去除重复。那么下面我们来看具体的步骤，首先我们算出超过20个的个数over，我们先把over加到结果res中，因为无论如何这over个删除操作都是要做的。如果没超过，over就是0，用变量left表示解决重复字符最少需要替换的个数，初始化为0。然后我们遍历之前统计字符出现个数的数组，如果某个字符出现个数大于等于3，且此时over大于0，那么我们将个数减为最近的3m+2个，over也对应的减少，注意，一旦over小于等于0，不要再进行删除操作。如果所有重复个数都减为3m+2了，但是over仍大于0，那么我们还要进一步的进行删除操作，这回每次直接删除3m个，直到over小于等于0为止，剩下的如果还有重复个数大于3的字符，我们算出置换字符需要的个数直接加到left中即可，最后我们比较left和missing，取其中较大值加入结果res中即可，参见代码如下：

```

1 class Solution {
2 public:
3     int strongPasswordChecker(string s) {
4         int res = 0, n = s.size(), lower = 1, upper = 1, digit = 1;
5         vector<int> v(n, 0);
6         for (int i = 0; i < n;) {
7             if (s[i] >= 'a' && s[i] <= 'z') lower = 0;
8             if (s[i] >= 'A' && s[i] <= 'Z') upper = 0;
9             if (s[i] >= '0' && s[i] <= '9') digit = 0;
10            int j = i;
11            while (i < n && s[i] == s[j]) ++i;
12            v[j] = i - j;
13        }
14        int missing = (lower + upper + digit);
15        if (n < 6) {
16            int diff = 6 - n;
17            res += diff + max(0, missing - diff);
18        } else {
19            int over = max(n - 20, 0), left = 0;
20            res += over;
21            for (int k = 1; k < 3; ++k) {
22                for (int i = 0; i < n && over > 0; ++i) {
23                    if (v[i] < 3 || v[i] % 3 != (k - 1)) continue;
24                    v[i] -= k;
25                    over -= k;
26                }
27            }
28            for (int i = 0; i < n; ++i) {
29                if (v[i] >= 3 && over > 0) {
30                    int need = v[i] - 2;
31                    v[i] -= over;
32                    over -= need;
33                }
34                if (v[i] >= 3) left += v[i] / 3;
35            }
36            res += max(missing, left);
37        }
38        return res;
39    }
40 };

```

421. 数组中异或值最大的两个数字

Given a non-empty array of numbers, $a_0, a_1, a_2, \dots, a_{n-1}$, where $0 \leq a_i < 2^{31}$.

Find the maximum result of $a_i \text{ XOR } a_j$, where $0 \leq i, j < n$.

Could you do this in $O(n)$ runtime?

Example:

Input: [3, 10, 5, 25, 2, 8]

Output: 28

Explanation: The maximum result is $5 \wedge 25 = 28$.

这道题是一道典型的位操作Bit Manipulation的题目，我开始以为异或值最大的两个数一定包括数组的最大值，但是OJ给了另一个例子{10, 23, 20, 18, 28}，这个数组的异或最大值是10和20异或，得到30。那么只能另辟蹊径，正确的做法是按位遍历，题目中给定了数字的返回不会超过231，那么最多只能有32位，我们用一个从左往右的mask，用来提取数字的前缀，然后将其都存入HashSet中，我们用一个变量t，用来验证当前位为1再或上之前结果res，看结果和HashSet中的前缀异或之后在不在HashSet中，这里用到了一个性质，若 $a \wedge b = c$ ，那么 $a = b \wedge c$ ，因为t是我们要验证的当前最大值，所以我们遍历HashSet中的数时，和t异或后的结果仍在HashSet中，说明两个前缀可以异或出t的值，所以我们更新res为t，继续遍历，如果上述讲解不容易理解，那么建议自己带个例子一步一步试试，并把每次循环中HashSet中所有的数字都打印出来，基本应该就能理解了，算了，还是博主带着大家来看题目中给的例子吧：

3	10	5	25	2	8
11	1010	101	11001	10	1000

我们观察这些数字最大的为25，其二进制最高位在 $i=4$ 时为1，那么我们的循环[31, 5]之间是取不到任何数字的，所以不会对结果res有任何影响。

当 $i=4$ 时，我们此时mask为前28位为'1'的二进制数，跟除25以外的任何数相'与'，都会得到0。然后跟25的二进制数10101相'与'，得到二进制数10000，存入HashSet中，那么此时HashSet中就有0和16两个数字。此时我们的t为结果res（此时为0）'或'上二进制数10000，得到二进制数10000。然后我们遍历HashSet，由于HashSet是无序的，所以我们会取出0和16中的其中一个，如果prefix取出的是0，那么 $t=16$ '异或'上0，还等于16，而16是在HashSet中存在的，所以此时结果res更新为16，然后break掉遍历HashSet的循环。实际上prefix先取16的话也一样，那么 $t=16$ '异或'上16，等于0，而0是在HashSet中存在的，所以此时结果res更新为16，然后break掉遍历HashSet的循环。

3	10	5	25	2	8
11	1010	101	11001	10	1000

当 $i=3$ 时，我们此时mask为前29位为'1'的二进制数，如上所示，跟数字3, 5, 2中任何一个相'与'，都会得到0。然后跟10的二进制数1010，或跟8的二进制数1000相'与'，都会得到二进制数1000，即8。跟25的二进制数11001相'与'，会得到二进数11000，即24，存入HashSet中，那么此时HashSet中就有0, 8, 和24三个数字。此时我们的t为结果res（此时为16）'或'上二进制数1000，得到二进制数11000，即24。此时遍历HashSet中的数，当prefix取出0，那么 $t=24$ '异或'上0，还等于24，而24是在HashSet中存在的，所以此时结果res更新为24，然后break掉遍历HashSet的循环。大家可以尝试其他的数，当prefix取出24，其实也可以更新结果res为24的。但是8就不行啦，因为HashSet中没有16。不过无所谓了，我们只要有一个能更新结果res就可以了。

3	10	5	25	2	8
11	1010	101	11001	10	1000

当 $i=2$ 时，我们此时mask为前30位为'1'的二进制数，如上所示，跟3的二进制数11相'与'，会得到二进制数0，即0。然后跟10的二进制数1010相'与'，会得到二进制数1000，即8。然后跟5的二进制数101相'与'，会得到二进制数100，即4。然后跟25的二进制数11001相'与'，会得到二进制数11000，即24。跟数字2和8相'与'，分别会得到0和8，跟前面重复了。所以最终HashSet中就有0, 4, 8, 和24这四个数字。此时我们的t为结果res（此时为24）'或'上二进制数100，得到二进制数11100，即28。那么就要验证结果res能否取到28。我们遍历HashSet，当prefix取出0，那么 $t=28$ '异或'上0，还等于28，但是HashSet中没有28，所以不行。当prefix取出4，那么 $t=28$ '异或'上二进制数100，等于24，在HashSet中存在，Bingo！结果res更新为28。其他的数可以不用试了。

3	10	5	25	2	8
11	1010	101	11001	10	1000

当 $i=1$ 时，我们此时mask为前31位为'1'的二进制数，如上所示，每个数与mask相'与'后，我们HashSet中会有2, 4, 8, 10, 24这五个数。此时我们的t为结果res（此时为28）'或'上二进制数10，得到二进制数11110，即30。那么就要验证结果res能否取到30。我们遍历HashSet，当prefix取出2，那么 $t=30$ '异或'上2，等于28，但是HashSet中没有28，所以不行。当prefix取出4，那么 $t=30$ '异或'上4，等于26，但是HashSet中没有26，所以不行。当prefix取出8，那么 $t=30$ '异或'上8，等于22，但是HashSet中没有22，所以不行。当prefix取出10，那么 $t=30$ '异或'上10，等于20，但是HashSet中没有20，所以不行。当prefix取出24，那么 $t=30$ '异或'上24，等于6，但是HashSet中没有6，所以不行。遍历完了HashSet所有的数，结果res没有被更新，还是28。

3	10	5	25	2	8
11	1010	101	11001	10	1000

当 $i=0$ 时，我们此时mask为前32位为'1'的二进制数，如上所示，每个数与mask相'与'后，我们HashSet中会有2, 3, 5, 8, 10, 25这六个数。此时我们的t为结果res（此时为28）'或'上二进制数1，得到二进制数11101，即29。那么就要验证结果res能否取到29。取出HashSet中每一个数字来验证，跟上面的验证方法相同，这里博主偷懒就不写了，最终可以发现，结果res无法被更新，还是28，所以最终的结果就是28。

综上所述，我们来分析一下这道题的核心。我们希望用二进制来拼出结果的数，最终结果28的二进制数为11100，里面有三个'1'，我们来找一下都是谁贡献了这三个'1'? 在 $i=4$ 时，数字25贡献了最高位的'1'，在 $i=3$ 时，数字25贡献了次高位的'1'，在 $i=2$ 时，数字5贡献了第三位的'1'。而一旦某个数贡献了'1'，那么之后在需要贡献'1'的时候，此数就可以再继续贡献'1'。而一旦有两个数贡献了'1'后，那么之后的'1'就基本上只跟这两个数有关了，其他数字有'1'也贡献不出来。验证方法里使用了前面提到的性质， $a \wedge b = t$ ，如果t是所求结果话，我们可以先假定一个t，然后验证，如果 $a \wedge t = b$ 成立，说明该t可以通过a和b'异或'得到。参见代码如下：

```

1 class Solution {
2 public:
3     int findMaximumXOR(vector<int>& nums) {
4         int res = 0, mask = 0;
5         for (int i = 31; i >= 0; --i) {
6             mask |= (1 << i);
7             unordered_set<int> s;
8             for (int num : nums) {
9                 s.insert(num & mask);
10            }
11            int t = res | (1 << i);
12            for (int prefix : s) {
13                if (s.count(t ^ prefix)) {
14                    res = t;
15                    break;
16                }
17            }
18        }
19        return res;
20    }
21 };

```

CPP

422. 验证单词平方

Given a sequence of words, check whether it forms a valid word square.

A sequence of words forms a valid word square if the k th row and column read the exact same string, where $0 \leq k < \max(\text{numRows}, \text{numColumns})$.

Note:

The number of words given is at least 1 and does not exceed 500.
Word length will be at least 1 and does not exceed 500.
Each word contains only lowercase English alphabet a-z.

这道题给了我们一个二维数组，每行每列都是一个单词，需要满足第 k 行的单词和第 k 列的单词要相等，这里不要求每一个单词的长度都一样，只要对应位置的单词一样即可。那么这里实际上也就是一个遍历二维数组，然后验证对应位上的字符是否相等问题，由于各行的单词长度不一定相等，所以我们在找对应位置的字符时，要先判断是否越界，即对应位置是否有字符存在，遇到不符合要求的地方直接返回false，全部遍历结束后返回true，参见代码如下：

```

1 class Solution {
2 public:
3     bool validWordSquare(vector<string>& words) {
4         if (words.empty()) return true;
5         if (words.size() != words[0].size()) return false;
6         for (int i = 0; i < words.size(); ++i) {
7             for (int j = 0; j < words[i].size(); ++j) {
8                 if (j >= words.size() || i >= words[j].size() || words[i][j] != words[j]
9 [i]) {
10                     return false;
11                 }
12             }
13         }
14         return true;
15     }
16 };

```

423. 从英文中重建数字

Given a non-empty string containing an out-of-order English representation of digits 0-9, output the digits in ascending order.

Note:

Input contains only lowercase English letters.

Input is guaranteed to be valid and can be transformed to its original digits. That means invalid inputs such as "abc" or "zerone" are not permitted.

Input length is less than 50,000.

这道题给了我们一串英文字符串，是由表示数字的英文单词组成的，不过字符顺序是打乱的，让我们重建出数字。那么这道题的思路是先要统计出各个字符出现的次数，然后算出每个单词出现的次数，然后就可以重建了。由于题目中限定了输入的字符串一定是有效的，那么不会出现无法成功重建的情况，这里需要用个trick。我们仔细观察这些表示数字的单词"zero", "one", "two", "three", "four", "five", "six", "seven", "eight", "nine"，我们可以发现有些的单词的字符是独一无二的，比如z，只出现在zero中，还有w, u, x, g这四个单词，分别只出现在two, four, six, eight中，那么这五个数字的个数就可以被确定了，由于含有o的单词有zero, two, four, one，其中前三个都被确定了，那么one的个数也就知道了；由于含有h的单词有eight, three，其中eight个数已知，那么three的个数就知道了；由于含有f的单词有four, five，其中four个数已知，那么five的个数就知道了；由于含有s的单词有six, seven，其中six个数已知，那么seven的个数就知道了；由于含有i的单词有six, eight, five, nine，其中前三个都被确定了，那么nine的个数就知道了，知道了这些问题就变的容易多了，我们按这个顺序"zero", "two", "four", "six", "eight", "one", "three", "five", "seven", "nine"就能找出所有的个数了，参见代码如下：

解法1

```

1 class Solution {
2 public:
3     string originalDigits(string s) {
4         string res = "";
5         vector<string> words{"zero", "two", "four", "six", "eight", "one", "three", "five",
6 "seven", "nine"};
7         vector<int> nums{0, 2, 4, 6, 8, 1, 3, 5, 7, 9}, counts(26, 0);
8         vector<char> chars{'z', 'w', 'u', 'x', 'g', 'o', 'h', 'f', 's', 'i'};
9         for (char c : s) ++counts[c - 'a'];
10        for (int i = 0; i < 10; ++i) {
11            int cnt = counts[chars[i] - 'a'];
12            for (int j = 0; j < words[i].size(); ++j) {
13                counts[words[i][j] - 'a'] -= cnt;
14            }
15            while (cnt--) res += (nums[i] + '0');
16        }
17        sort(res.begin(), res.end());
18        return res;
19    }
20 };

```

另外我们也可以用更加简洁易懂的方法来快速的找出各个数字的个数，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     string originalDigits(string s) {
4         string res = "";
5         vector<int> counts(128, 0), nums(10, 0);
6         for (char c : s) ++counts[c];
7         nums[0] = counts['z'];
8         nums[2] = counts['w'];
9         nums[4] = counts['u'];
10        nums[6] = counts['x'];
11        nums[8] = counts['g'];
12        nums[1] = counts['o'] - nums[0] - nums[2] - nums[4];
13        nums[3] = counts['h'] - nums[8];
14        nums[5] = counts['f'] - nums[4];
15        nums[7] = counts['s'] - nums[6];
16        nums[9] = counts['i'] - nums[6] - nums[8] - nums[5];
17        for (int i = 0; i < nums.size(); ++i) {
18            for (int j = 0; j < nums[i]; ++j) {
19                res += (i + '0');
20            }
21        }
22        return res;
23    }
24 };

```

424. 最长重复字符置换

Given a string that consists of only uppercase English letters, you can replace any letter in the string with another letter at most k times. Find the length of a longest substring containing all repeating letters you can get after performing the above operations.

Note:

Both the string's length and k will not exceed 104.

Example 1:

Input:

s = "ABAB", k = 2

Output:

4

这道题给我们了一个字符串，说我们有k次随意置换任意字符的机会，让我们找出最长的重复字符的字符串。这道题跟之前那道 Longest Substring with At Most K Distinct Characters 很像，都需要用滑动窗口 Sliding Window 来解。我们首先来想，如果没有k的限制，让我们求把字符串变成只有一个字符重复的字符串需要的最小置换次数，那么就是字符串的总长度减去出现次数最多的字符的个数。如果加上k的限制，我们其实就是要满足(子字符串的长度减去出现次数最多的字符个数)≤k的最大子字符串长度即可，搞清了这一点，我们也就应该知道怎么用滑动窗口来解了吧。我们用一个变量start记录滑动窗口左边界，初始化为0，然后我们遍历字符串，每次累加出现字符的个数，然后更新出现最多字符的个数，然后我们判断当前滑动窗口是否满足之前说的那个条件，如果不满足，我们就把滑动窗口左边界向右移动一个，并注意去掉的字符要在counts里减一，直到满足条件，我们更新结果res即可，参见代码如下：

```
1 class Solution {
2 public:
3     int characterReplacement(string s, int k) {
4         int res = 0, maxCnt = 0, start = 0;
5         vector<int> counts(26, 0);
6         for (int i = 0; i < s.size(); ++i) {
7             maxCnt = max(maxCnt, ++counts[s[i] - 'A']);
8             while (i - start + 1 - maxCnt > k) {
9                 --counts[s[start] - 'A'];
10                ++start;
11            }
12            res = max(res, i - start + 1);
13        }
14        return res;
15    }
16};
```

425. 单词平方

Given a set of words (without duplicates), find all word squares you can build from them.

A sequence of words forms a valid word square if the kth row and column read the exact same string, where $0 \leq k < \max(\text{numRows}, \text{numColumns})$.

For example, the word sequence `["ball", "area", "lead", "lady"]` forms a word square because each word reads the same both horizontally and vertically.

```
b a l l
a r e a
l e a d
l a d y
```

Note:

There are at least 1 and at most 1000 words.

All words will have the exact same length.

Word length is at least 1 and at most 5.

Each word contains only lowercase English alphabet a-z.

这道题是之前那道Valid Word Square的延伸，由于要求出所有满足要求的单词平方，所以难度大大的增加了，不要幻想着可以利用之前那题的解法来暴力破解，OJ不会答应的。那么根据以往的经验，对于这种要打印出所有情况的题的解法大多都是用递归来解，那么这题的关键是根据前缀来找单词，我们如果能利用合适的数据结构来建立前缀跟单词之间的映射，使得我们能快速的通过前缀来判断某个单词是否存在，这是解题的关键。对于建立这种映射，这里主要有两种方法，一种是利用哈希表来建立前缀和所有包含此前缀单词的集合之前的映射，第二种方法是建立前缀树Trie，顾名思义，前缀树专门就是为这种问题设计的。那么我们首先来看第一种方法，用哈希表来建立映射的方法，我们就是取出每个单词的所有前缀，然后将该单词加入该前缀对应的集合中去，然后我们建立一个空的 $n \times n$ 的char矩阵，其中n为单词的长度，我们的目标就是来把这个矩阵填满，我们从0开始遍历，我们先取出长度为0的前缀，即空字符串，由于我们在建立映射的时候，空字符串也和每个单词的集合建立了映射，然后我们遍历这个集合，用遍历到的单词的i位置字符，填充矩阵 $\text{mat}[i][i]$ ，然后j从i+1出开始遍历，对应填充矩阵 $\text{mat}[i][j]$ 和 $\text{mat}[j][i]$ ，然后我们根据第j行填充得到的前缀，在哈希表中查看有没单词，如果没有，就break掉，如果有，则继续填充下一个位置。最后如果 $j == n$ 了，说明第0行和第0列都被填好了，我们再调用递归函数，开始填充第一行和第一列，依次类推，直至填充完成，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<vector<string>> wordSquares(vector<string>& words) {
4         vector<vector<string>> res;
5         unordered_map<string, set<string>> m;
6         int n = words[0].size();
7         for (string word : words) {
8             for (int i = 0; i < n; ++i) {
9                 string key = word.substr(0, i);
10                m[key].insert(word);
11            }
12        }
13        vector<vector<char>> mat(n, vector<char>(n));
14        helper(0, n, mat, m, res);
15        return res;
16    }
17    void helper(int i, int n, vector<vector<char>>& mat, unordered_map<string,
18 set<string>>& m, vector<vector<string>>& res) {
19        if (i == n) {
20            vector<string> out;
21            for (int j = 0; j < n; ++j) out.push_back(string(mat[j].begin(),
22 mat[j].end()));
23            res.push_back(out);
24            return;
25        }
26        string key = string(mat[i].begin(), mat[i].begin() + i);
27        for (string str : m[key]) {
28            mat[i][i] = str[i];
29            int j = i + 1;
30            for (; j < n; ++j) {
31                mat[i][j] = str[j];
32                mat[j][i] = str[j];
33                if (!m.count(string(mat[j].begin(), mat[j].begin() + i + 1))) break;
34            }
35            if (j == n) helper(i + 1, n, mat, m, res);
36        }
37    }
38};

```

下面来看建立前缀树Trie的方法，这种方法的难点是看能不能熟练的写出Trie的定义，还有构建过程，以及后面在递归函数中，如果利用前缀树来快速查找单词的前缀，总之，这道题是前缀树的一种经典的应用，能白板写出来就说明基本上已经掌握了前缀树了，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     struct TrieNode {
4         vector<int> indexs;
5         vector<TrieNode*> children;
6         TrieNode(): children(26, nullptr) {}
7     };
8     TrieNode* buildTrie(vector<string>& words) {
9         TrieNode *root = new TrieNode();
10        for (int i = 0; i < words.size(); ++i) {
11            TrieNode *t = root;
12            for (int j = 0; j < words[i].size(); ++j) {
13                if (!t->children[words[i][j] - 'a']) {
14                    t->children[words[i][j] - 'a'] = new TrieNode();
15                }
16                t = t->children[words[i][j] - 'a'];
17                t->indexs.push_back(i);
18            }
19        }
20        return root;
21    }
22    vector<vector<string>> wordSquares(vector<string>& words) {
23        TrieNode *root = buildTrie(words);
24        vector<string> out(words[0].size());
25        vector<vector<string>> res;
26        for (string word : words) {
27            out[0] = word;
28            helper(words, 1, root, out, res);
29        }
30        return res;
31    }
32    void helper(vector<string>& words, int level, TrieNode* root, vector<string>& out,
33    vector<vector<string>>& res) {
34        if (level >= words[0].size()) {
35            res.push_back(out);
36            return;
37        }
38        string str = "";
39        for (int i = 0; i < level; ++i) {
40            str += out[i][level];
41        }
42        TrieNode *t = root;
43        for (int i = 0; i < str.size(); ++i) {
44            if (!t->children[str[i] - 'a']) return;
45            t = t->children[str[i] - 'a'];
46        }
47        for (int idx : t->indexs) {
48            out[level] = words[idx];
49            helper(words, level + 1, root, out, res);
50        }
51    }
52};

```

426. Convert Binary Search Tree to Sorted Doubly Linked List

```

1 class Solution {
2 public:
3     Node* treeToDoublyList(Node* root) {
4         if(!root) return root;
5         Node* cur = root;
6         stack<Node*> stk;
7         Node* head = NULL;
8         Node* prev = NULL;
9         while(cur || !stk.empty()){
10             if(cur){
11                 stk.push(cur);
12                 cur = cur->left;
13             }else{
14                 cur = stk.top();
15                 stk.pop();
16                 if(!head) head = cur;
17                 if(prev){
18                     prev->right = cur;
19                     cur->left = prev;
20                 }
21                 prev = cur;
22                 cur = cur->right;
23             }
24         }
25         head->left = prev;
26         prev->right = head;
27         return head;
28     }
29 };

```

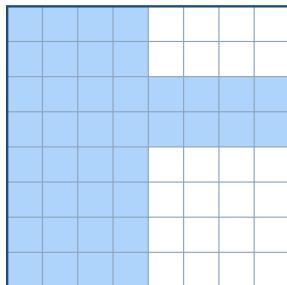
427. 建立四叉树

我们想要使用一棵四叉树来储存一个 $N \times N$ 的布尔值网络。网络中每一格的值只会是真或假。树的根结点代表整个网络。对于每个结点，它将被分等成四个孩子结点直到这个区域内的值都是相同的。

每个结点还有另外两个布尔变量：`isLeaf` 和 `val`。`isLeaf` 当这个节点是一个叶子结点时为真。`val` 变量储存叶子结点所代表的区域的值。

你的任务是使用一个四叉树表示给定的网络。下面的例子将有助于你理解这个问题：

给定下面这个 8×8 网络，我们将这样建立一个对应的四叉树：



由上文的定义，它能被这样分割：



```

1  /*
2   // Definition for a QuadTree node.
3   class Node {
4     public:
5       bool val;
6       bool isLeaf;
7       Node* topLeft;
8       Node* topRight;
9       Node* bottomLeft;
10      Node* bottomRight;
11
12      Node() {}
13
14     Node(bool _val, bool _isLeaf, Node* _topLeft, Node* _topRight, Node* _bottomLeft, Node*
15     _bottomRight) {
16       val = _val;
17       isLeaf = _isLeaf;
18       topLeft = _topLeft;
19       topRight = _topRight;
20       bottomLeft = _bottomLeft;
21       bottomRight = _bottomRight;
22     }
23   };
24 */
25 class Solution {
26 public:
27   Node* construct(vector<vector<int>>& grid) {
28     return construct(grid, 0, 0, grid.size());
29   }
30
31   Node* construct(const vector<vector<int>>& grid, int x, int y, int n) {
32     if (n == 0) return nullptr;
33     bool all_zeros = true;
34     bool all_ones = true;
35     for (int i = y; i < y + n; ++i)
36       for (int j = x; j < x + n; ++j)
37         if (grid[i][j] == 0)
38           all_ones = false;
39         else
40           all_zeros = false;
41     if (all_zeros || all_ones)
42       return new Node(all_ones, true, nullptr, nullptr, nullptr);
43
44     return new Node(true, false,
45                   construct(grid, x, y, n/2), // topLeft
46                   construct(grid, x + n/2, y, n/2), // topRight
47                   construct(grid, x, y + n/2, n/2), // bottomLeft
48                   construct(grid, x + n/2, y + n/2, n/2)); // bottomRight
49   }
50 };

```

428. Serialize and Deserialize N-ary Tree

```

1 // This function stores the given N-ary tree in a file pointed by fp
2 void serialize(Node *root, FILE *fp)
3 {
4     // Base case
5     if (root == NULL) return;
6
7     // Else, store current node and recur for its children
8     fprintf(fp, "%c ", root->key);
9     for (int i = 0; i < N && root->child[i]; i++)
10        serialize(root->child[i], fp);
11
12    // Store marker at the end of children
13    fprintf(fp, "%c ", MARKER);
14 }
15
16 // This function constructs N-ary tree from a file pointed by 'fp'.
17 // This function returns 0 to indicate that the next item is a valid
18 // tree key. Else returns 0
19 int deSerialize(Node *&root, FILE *fp)
20 {
21     // Read next item from file. If there are no more items or next
22     // item is marker, then return 1 to indicate same
23     char val;
24     if ( !fscanf(fp, "%c ", &val) || val == MARKER )
25         return 1;
26
27     // Else create node with this item and recur for children
28     root = newNode(val);
29     for (int i = 0; i < N; i++)
30         if (deSerialize(root->child[i], fp))
31             break;
32
33     // Finally return 0 for successful finish
34     return 0;
35 }
```

429. N叉树的层序遍历

给定一个 N 叉树，返回其节点值的层序遍历。（即从左到右，逐层遍历）。

```

1  /*
2   // Definition for a Node.
3   class Node {
4   public:
5   int val = NULL;
6   vector<Node*> children;
7
8   Node() {}
9
10  Node(int _val, vector<Node*> _children) {
11      val = _val;
12      children = _children;
13  }
14 }
15 */
16 class Solution {
17 public:
18     vector<vector<int>> levelOrder(Node* root) {
19         vector<vector<int>> res;
20         if(!root) return res;
21         queue<Node*> q;
22         q.push(root);
23
24         while(!q.empty()){
25             vector<int> tmp;
26             int n=q.size();
27             for(int i=0;i<n;++i){
28                 Node* t=q.front();q.pop();
29                 tmp.push_back(t->val);
30                 for(int j=0;j<t->children.size();++j){
31                     q.push(t->children[j]);
32                 }
33             }
34             res.push_back(tmp);
35         }
36         return res;
37     }
38 };

```

430. 扁平化多级双向链表

您将获得一个双向链表，除了下一个和前一个指针之外，它还有一个子指针，可能指向单独的双向链表。这些子列表可能有一个或多个自己的子项，依此类推，生成多级数据结构，如下面的示例所示。

扁平化列表，使所有结点出现在单级双链表中。您将获得列表第一级的头部。

示例：

输入：

```

1---2---3---4---5---6--NULL
 |
7---8---9---10--NULL
 |
11--12--NULL

```

输出：

```
1-2-3-7-8-11-12-9-10-4-5-6--NULL
```

```
1 class Solution {
2 public:
3     bool dfs(Node *head, vector<Node *> &path){
4         if(head == NULL){
5             return false;
6         }
7
7         path.push_back(head);
8         if(head->child){
9             dfs(head->child, path);
10        }
11        dfs(head->next, path);
12
13        return true;
14    }
15
16
17    Node* flatten(Node* head) {
18        Node * node = head;
19        vector<Node *> path;
20        dfs(head, path);
21
22        if(!head){
23            return NULL;
24        }
25
26        for(int i = 1;i < path.size(); ++i){
27            node->next = path[i];
28            node->child = NULL;
29            path[i]->prev = node;
30            node = path[i];
31        }
32
33        node->next = NULL;
34        node->child = NULL;
35
36        return head;
37    }
38
39    /*
40    Node* getTail(Node* head) { /* 获取尾节点 */
41        Node* now = head;
42        while (now->next) {
43            now = now->next;
44        }
45        return now;
46    }
47
48    Node* flatten(Node* head) {
49        Node* now = head;
50        while (now) {
51            if (now->child) {
52                Node* subHead = flatten(now->child); /* 递归获取展开后的子序列的头节点 */
53                Node* subTail = getTail(subHead); /* 获取子序列的尾节点 */
54                Node* nowNext = now->next;
55                now->child = nullptr; /* 断开儿子节点 */
56                /* 将当前节点与子序列的头部进行链接 */
57                now->next = subHead;
58                subHead->prev = now;
59                /* 若当前节点存在后驱，则要将当前节点的后驱与子序列的尾部链接 */
60            }
61        }
62    }
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
```

```
60     if (nowNext) {  
61         nowNext->prev = subTail;  
62         subTail->next = nowNext;  
63     }  
64     now = now->next;  
65 }  
66 return head;  
67 }  
68 */  
69 };  
70 };
```

431. Encode N-ary Tree to Binary Tree

```

1  class Codec {
2  public:
3
4      // Encodes an n-ary tree to a binary tree.
5      TreeNode* encode(Node* root) {
6          if (root == nullptr) {
7              return nullptr;
8          }
9          auto node = new TreeNode(root->val);
10         if (!root->children.empty()) {
11             node->right = encodeHelper(root->children[0], root, 0);
12         }
13         return node;
14     }
15
16     // Decodes your binary tree to an n-ary tree.
17     Node* decode(TreeNode* root) {
18         if (root == nullptr) {
19             return nullptr;
20         }
21         vector<Node*> children;
22         auto node = new Node(root->val, children);
23         decodeHelper(root->right, node);
24         return node;
25     }
26
27 private:
28     TreeNode *encodeHelper(Node *root, Node *parent, int index) {
29         if (root == nullptr) {
30             return nullptr;
31         }
32         auto node = new TreeNode(root->val);
33         if (index + 1 < parent->children.size()) {
34             node->left = encodeHelper(parent->children[index + 1], parent, index + 1);
35         }
36         if (!root->children.empty()) {
37             node->right = encodeHelper(root->children[0], root, 0);
38         }
39         return node;
40     }
41
42     void decodeHelper(TreeNode* root, Node* parent) {
43         if (!root) {
44             return;
45         }
46         vector<Node*> children;
47         auto node = new Node(root->val, children);
48         decodeHelper(root->right, node);
49         parent->children.push_back(node);
50         decodeHelper(root->left, parent);
51     }
52 };

```

432. 全O(1)的数据结构

Implement a data structure supporting the following operations:

Inc(Key) - Inserts a new key with value 1. Or increments an existing key by 1. Key is guaranteed to be a non-empty string.

Dec(Key) - If Key's value is 1, remove it from the data structure. Otherwise decrements an existing key by 1. If the key does not exist, this function does nothing. Key is guaranteed to be a non-empty string.

GetMaxKey() - Returns one of the keys with maximal value. If no element exists, return an empty string "".

GetMinKey() - Returns one of the keys with minimal value. If no element exists, return an empty string "".

Challenge: Perform all these in O(1) time complexity.

这道题让我们实现一个全是O(1)复杂度的数据结构，包括了增加key，减少key，获取最大key，获取最小key，这几个函数。由于需要常数级的时间复杂度，我们首先第一反应就是要用哈希表来做，不仅如此，我们肯定还需要用list来保存所有的key，那么哈希表就是建立key和list中位置迭代器之间的映射，这不由得令人想到了之前那道LRU Cache，也是用了类似的方法来解，但是感觉此题还要更加复杂一些。由于每个key还要对应一个次数，所以list中不能只放key，而且相同的次数可能会对应多个key值，所以我们用unordered_set来保存次数相同的所有key值，我们建立一个Bucket的结构体来保存次数val，和保存key值的集合keys。解题思路主要参考了网友ivancjw的帖子，数据结构参考了史蒂芬大神的帖子，思路是，我们建立一个次数分层的结构，次数多的在顶层，每一层放相同次数的key值，例如下面这个例子：

```
"A": 4, "B": 4, "C": 2, "D": 1
```

那么用我们设计的结构保存出来就是：

```
row0: val = 4, keys = {"A", "B"}  
row1: val = 2, keys = {"C"}  
row2: val = 1, keys = {"D"}
```

好，我们现在来分析如何实现inc函数，我们来想，如果我们插入一个新的key，跟我们插入一个已经存在的key，情况是完全不一样的，那么我们就需要分情况来讨论：

- 如果我们插入一个新的key，那么由于该key没有出现过，所以加入后次数一定为1，那么就有两种情况了，如果list中没有val为1的这一行，那么我们需要插入该行，如果已经有了val为1的这行，我们直接将key加入集合keys中即可。
- 如果我们插入了一个已存在的key，那么由于个数增加了1个，所以该key值肯定不能在当前行继续待下去了，要往上升职啊，那么这里就有两种情况了，如果该key要升职到的那行不存在，我们需要手动添加那一行；如果那一行存在，我们之间将key加入集合keys中，记得都要将原来行中的key值删掉。

下面我们再来看dec函数如何实现，其实理解了上面的inc函数，那么dec函数也就没什么难度了：

- 如果我们要删除的key不存在，那么直接返回即可。
- 如果我们要删除的key存在，那么我们看其val值是否为1，如果为1的话，那么直接在keys中删除该key即可，然后还需要判断如果该key是集合中的唯一一个，那么该行也需要删除。如果key的次数val不为1的话，我们要考虑降级问题，跟之前的升职很类似，如果要降级的行不存在，我们手动添加上，如果存在，则直接将key值添加到keys集合中即可。

当我们搞懂了inc和dec的实现方法，那么getMaxKey()和getMinKey()简直就是福利啊，不要太简单啊，直接返回首层和尾层的key值即可，参见代码如下：

```

1 class AllOne {
2 public:
3     /** Initialize your data structure here. */
4     AllOne() {}
5
6     /** Inserts a new key <Key> with value 1. Or increments an existing key by 1. */
7     void inc(string key) {
8         if (!m.count(key)) {
9             if (buckets.empty() || buckets.back().val != 1) {
10                 auto newBucket = buckets.insert(buckets.end(), {1, {key}});
11                 m[key] = newBucket;
12             } else {
13                 auto newBucket = --buckets.end();
14                 newBucket->keys.insert(key);
15                 m[key] = newBucket;
16             }
17         } else {
18             auto curBucket = m[key], lastBucket = (--m[key]);
19             if (lastBucket == buckets.end() || lastBucket->val != curBucket->val + 1) {
20                 auto newBucket = buckets.insert(curBucket, {curBucket->val + 1, {key}});
21                 m[key] = newBucket;
22             } else {
23                 lastBucket->keys.insert(key);
24                 m[key] = lastBucket;
25             }
26             curBucket->keys.erase(key);
27             if (curBucket->keys.empty()) buckets.erase(curBucket);
28         }
29     }
30
31     /** Decrements an existing key by 1. If Key's value is 1, remove it from the data
32     structure. */
33     void dec(string key) {
34         if (!m.count(key)) return;
35         auto curBucket = m[key];
36         if (curBucket->val == 1) {
37             curBucket->keys.erase(key);
38             if (curBucket->keys.empty()) buckets.erase(curBucket);
39             m.erase(key);
40             return;
41         }
42         auto nextBucket = ++m[key];
43         if (nextBucket == buckets.end() || nextBucket->val != curBucket->val - 1) {
44             auto newBucket = buckets.insert(nextBucket, {curBucket->val - 1, {key}});
45             m[key] = newBucket;
46         } else {
47             nextBucket->keys.insert(key);
48             m[key] = nextBucket;
49         }
50         curBucket->keys.erase(key);
51         if (curBucket->keys.empty()) buckets.erase(curBucket);
52     }
53
54     /** Returns one of the keys with maximal value. */
55     string getMaxKey() {
56         return buckets.empty() ? "" : *(buckets.begin()->keys.begin());
57     }
58
59     /** Returns one of the keys with Minimal value. */

```

```

60     string getMinKey() {
61         return buckets.empty() ? "" : *(buckets.rbegin()->keys.begin());
62     }
63 private:
64     struct Bucket { int val; unordered_set<string> keys; };
65     list<Bucket> buckets;
66     unordered_map<string, list<Bucket>::iterator> m;
67 };

```

433. 最小基因变化

A gene string can be represented by an 8-character long string, with choices from "A", "C", "G", "T".

Suppose we need to investigate about a mutation (mutation from "start" to "end"), where ONE mutation is defined as ONE single character changed in the gene string.

For example, "AACCGGTT" -> "AACCGGTA" is 1 mutation.

Also, there is a given gene "bank", which records all the valid gene mutations. A gene must be in the bank to make it a valid gene string.

Now, given 3 things - start, end, bank, your task is to determine what is the minimum number of mutations needed to mutate from "start" to "end". If there is no such a mutation, return -1.

Note:

Starting point is assumed to be valid, so it might not be included in the bank.
If multiple mutations are needed, all mutations during in the sequence must be valid.
You may assume start and end string is not the same.

这道题跟之前的Word Ladder完全是一道题啊，换个故事就直接来啊，越来越走心了啊。不过博主做的时候并没有想起来是之前一样的题，而是先按照脑海里第一个浮现出的思路做的，发现也通过OJ了。博主使用的一种BFS的搜索，先建立bank数组的距离场，这里距离就是两个字符串之间不同字符的个数。然后以start字符串为起点，向周围距离为1的点扩散，采用BFS搜索，每扩散一层，level自加1，当扩散到end字符串时，返回当前level即可。注意我们要把start字符串也加入bank中，而且此时我们也知道start的坐标位置，bank的最后一个位置，然后在简历距离场的时候，调用一个count子函数，用来统计输入的两个字符串之间不同字符的个数，注意dist[i][j]和dist[j][i]是相同，所以我们只用算一次就行了。然后我们进行BFS搜索，用一个visited集合来保存遍历过的字符串，注意检测距离的时候，dist[i][j]和dist[j][i]只要有一个是1，就可以了，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int minMutation(string start, string end, vector<string>& bank) {
4         if (bank.empty()) return -1;
5         bank.push_back(start);
6         int res = 0, n = bank.size();
7         queue<int> q{{n - 1}};
8         vector<vector<int>> dist(n, vector<int>(n, 0));
9         for (int i = 0; i < n; ++i) {
10             for (int j = i + 1; j < n; ++j) {
11                 dist[i][j] = count(bank[i], bank[j]);
12             }
13         }
14         unordered_set<int> visited;
15         while (!q.empty()) {
16             int len = q.size();
17             ++res;
18             for (int i = 0; i < len; ++i) {
19                 int t = q.front(); q.pop();
20                 visited.insert(t);
21                 for (int j = 0; j < n; ++j) {
22                     if ((dist[t][j] != 1 && dist[j][t] != 1) || visited.count(j)) continue;
23                     if (bank[j] == end) return res;
24                     q.push(j);
25                 }
26             }
27         }
28         return -1;
29     }
30     int count(string word1, string word2) {
31         int cnt = 0, n = word1.size();
32         for (int i = 0; i < n; ++i) {
33             if (word1[i] != word2[i]) ++cnt;
34         }
35         return cnt;
36     }
37 };

```

下面这种解法跟之前的那道Word Ladder是一样的，也是用的BFS搜索。跟上面的解法不同之处在于，对于遍历到的字符串，我们不再有距离场，而是对于每个字符，我们都尝试将其换为一个新的字符，每次只换一个，这样会得到一个新的字符串，如果这个字符串在bank中存在，说明这样变换是合法的，加入visited集合和queue中等待下一次遍历，记得在下次置换字符的时候要将之前的还原。我们在queue中取字符串出来遍历的时候，先检测其是否和end相等，相等的话返回level，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int minMutation(string start, string end, vector<string>& bank) {
4         if (bank.empty()) return -1;
5         vector<char> gens{'A','C','G','T'};
6         unordered_set<string> s{bank.begin(), bank.end()};
7         unordered_set<string> visited;
8         queue<string> q{{start}};
9         int level = 0;
10        while (!q.empty()) {
11            int len = q.size();
12            for (int i = 0; i < len; ++i) {
13                string t = q.front(); q.pop();
14                if (t == end) return level;
15                for (int j = 0; j < t.size(); ++j) {
16                    char old = t[j];
17                    for (char c : gens) {
18                        t[j] = c;
19                        if (s.count(t) && !visited.count(t)) {
20                            visited.insert(t);
21                            q.push(t);
22                        }
23                    }
24                    t[j] = old;
25                }
26            }
27            ++level;
28        }
29        return -1;
30    }
31 };

```

博主一直想找种递归的解法，于是在论坛上找到了这个帖子，是Java版的递归写法，博主将其改写成C++版本，但是无法通过OJ，百思不得其解啊，明明一模一样啊，连变量名都起的一样，为啥Java版的就是对的，博主的这个改写版就不对呢，各位看官大神们帮忙解答一下呀~

解法3：

```

1 class Solution {
2 public:
3     int minMutation(string start, string end, vector<string>& bank) {
4         vector<bool> explored(bank.size(), false);
5         if (bank.empty()) return -1;
6         return minMutation(explored, start, end, bank);
7     }
8     bool minMutation(vector<bool>& explored, string start, string end, vector<string>&
9 bank) {
10     if (start == end) return 0;
11     int step = bank.size() + 1;
12     for (int i = 0; i < bank.size(); ++i) {
13         if (diffOne(start, bank[i]) && explored[i]) {
14             explored[i] = true;
15             int temp = minMutation(explored, bank[i], end, bank);
16             if (temp != -1) {
17                 step = min(step, temp);
18             }
19             explored[i] = false;
20         }
21     }
22     return step == bank.size() + 1 ? -1 : 1 + step;
23 }
24 bool diffOne(string& s1, string& s2) {
25     int count = 0;
26     for (int i = 0; i < s1.size(); ++i) {
27         if (s1[i] != s2[i]) ++count;
28         if (count >= 2) return false;
29     }
30     return count == 1;
31 }
32 };

```

434. 字符串中的分段数量

Count the number of segments in a string, where a segment is defined to be a contiguous sequence of non-space characters.

Please note that the string does not contain any non-printable characters.

Example:

Input: "Hello, my name is John"

Output: 5

这道题跟之前那道Reverse Words in a String有些类似，不过比那题要简单一些，因为不用翻转单词，只要统计出单词的数量即可。那么我们的做法是遍历字符串，遇到空格直接跳过，如果不是空格，则计数器加1，然后用个while循环找到下一个空格的位置，这样就遍历完了一个单词，再重复上面的操作直至结束，就能得到正确结果：

解法1：

```

1 class Solution {
2 public:
3     int countSegments(string s) {
4         int res = 0, n = s.size();
5         for (int i = 0; i < n; ++i) {
6             if (s[i] == ' ') continue;
7             ++res;
8             while (i < n && s[i] != ' ') ++i;
9         }
10        return res;
11    }
12 };

```

下面这种方法是统计单词开头的第一个字符，因为每个单词的第一个字符前面一个字符一定是空格，利用这个特性也可以统计单词的个数：

解法2：

```

1 class Solution {
2 public:
3     int countSegments(string s) {
4         int res = 0;
5         for (int i = 0; i < s.size(); ++i) {
6             if (s[i] != ' ' && (i == 0 || s[i - 1] == ' ')) {
7                 ++res;
8             }
9         }
10        return res;
11    }
12 };

```

下面这种方法用到了C++的字符串流操作，利用getline函数取出每两个空格符之间的字符串，由于多个空格符可能连在一起，所以有可能取出空字符串，我们要判断一下，如果取出的是非空字符串我们才累加计数器，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int countSegments(string s) {
4         int res = 0;
5         istringstream is(s);
6         string t = "";
7         while (getline(is, t, ' ')) {
8             if (t.empty()) continue;
9             ++res;
10        }
11        return res;
12    }
13 };

```

435. 非重叠区间

Given a collection of intervals, find the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping.

Note:

You may assume the interval's end point is always bigger than its start point.
Intervals like [1,2] and [2,3] have borders "touching" but they don't overlap each other.

这道题给了我们一堆区间，让我们求需要至少移除多少个区间才能使剩下的区间没有重叠，那么我们首先要给区间排序，根据每个区间的start来做升序排序，然后我们开始要查找重叠区间，判断方法是看如果前一个区间的end大于后一个区间的start，那么一定是重复区间，此时我们结果res自增1，我们需要删除一个，那么此时我们究竟该删哪一个呢，为了保证我们总体去掉的区间数最小，我们去掉那个end值较大的区间，而在代码中，我们并没有真正的删掉某一个区间，而是用一个变量last指向一个需要比较的区间，我们将last指向end值较小的那个区间；如果两个区间没有重叠，那么此时last指向当前区间，继续进行下一次遍历，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     int eraseOverlapIntervals(vector<Interval>& intervals) {
4         int res = 0, n = intervals.size(), last = 0;
5         sort(intervals.begin(), intervals.end(), [] (Interval& a, Interval& b){return
6             a.start < b.start;});
7         for (int i = 1; i < n; ++i) {
8             if (intervals[i].start < intervals[last].end) {
9                 ++res;
10                if (intervals[i].end < intervals[last].end) last = i;
11            } else {
12                last = i;
13            }
14        }
15        return res;
16    }
};
```

CPP

我们也可以对上面代码进行简化，主要利用三元操作符来代替if从句，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     int eraseOverlapIntervals(vector<Interval>& intervals) {
4         if (intervals.empty()) return 0;
5         sort(intervals.begin(), intervals.end(), [] (Interval& a, Interval& b){return
6             a.start < b.start;});
7         int res = 0, n = intervals.size(), endLast = intervals[0].end;
8         for (int i = 1; i < n; ++i) {
9             int t = endLast > intervals[i].start ? 1 : 0;
10            endLast = t == 1 ? min(endLast, intervals[i].end) : intervals[i].end;
11            res += t;
12        }
13        return res;
14    }
};
```

CPP

436. 找右区间

Given a set of intervals, for each of the interval i , check if there exists an interval j whose start point is bigger than or equal to the end point of the interval i , which can be called that j is on the "right" of i .

For any interval i , you need to store the minimum interval j 's index, which means that the interval j has the minimum start point to build the "right" relationship for interval i . If the interval j doesn't exist, store -1 for the interval i . Finally, you need output the stored value of each interval as an array.

Note:

You may assume the interval's end point is always bigger than its start point.
You may assume none of these intervals have the same start point.

这道题给了我们一堆区间，让我们找每个区间的最近右区间，要保证右区间的start要大于等于当前区间的end，由于区间的顺序不能变，所以我们不能给区间排序，我们需要建立区间的start和该区间位置之间的映射，由于题目中限定了每个区间的start都不同，所以不用担心一对多的情况出现。然后我们把所有的区间的start都放到一个数组中，并对这个数组进行降序排序，那么start值大的就在数组前面。然后我们遍历区间集合，对于每个区间，我们在数组中找第一个小于当前区间的end值的位置，如果数组中第一个数就小于当前区间的end，那么说明该区间不存在右区间，结果res中加入-1；如果找到了第一个小于当前区间end的位置，那么往前推一个就是第一个大于等于当前区间end的start，我们在哈希表中找到该区间的坐标加入结果res中即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<int> findRightInterval(vector<Interval>& intervals) {
4         vector<int> res, v;
5         unordered_map<int, int> m;
6         for (int i = 0; i < intervals.size(); ++i) {
7             m[intervals[i].start] = i;
8             v.push_back(intervals[i].start);
9         }
10        sort(v.begin(), v.end(), greater<int>());
11        for (auto a : intervals) {
12            int i = 0;
13            for (; i < v.size(); ++i) {
14                if (v[i] < a.end) break;
15            }
16            res.push_back((i > 0) ? m[v[i - 1]] : -1);
17        }
18        return res;
19    }
20 };

```

CPP

上面的解法可以进一步化简，我们可以利用STL的lower_bound函数来找第一个不小于目标值的位置，这样也可以达到我们的目标，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<int> findRightInterval(vector<Interval>& intervals) {
4         vector<int> res;
5         map<int, int> m;
6         for (int i = 0; i < intervals.size(); ++i) {
7             m[intervals[i].start] = i;
8         }
9         for (auto a : intervals) {
10             auto it = m.lower_bound(a.end);
11             if (it == m.end()) res.push_back(-1);
12             else res.push_back(it->second);
13         }
14     }
15     return res;
16 }
16 };

```

437. 二叉树的路径和之三

You are given a binary tree in which each node contains an integer value.

Find the number of paths that sum to a given value.

The path does not need to start or end at the root or a leaf, but it must go downwards (traveling only from parent nodes to child nodes).

The tree has no more than 1,000 nodes and the values are in the range -1,000,000 to 1,000,000.

Example:

root = [10,5,-3,3,2,null,11,3,-2,null,1], sum = 8

```

      10
     /   \
    5    -3
   / \     \
  3  2    11
 / \   \
3  -2   1

```

Return 3. The paths that sum to 8 are:

1. 5 -> 3
2. 5 -> 2 -> 1
3. -3 -> 11

这道题让我们求二叉树的路径的和等于一个给定值，说明了这条路径不必要从根节点开始，可以是中间的任意一段，而且二叉树的节点值也是有正有负。那么我们可以用递归来做，相当于先序遍历二叉树，对于每一个节点都有记录了一条从根节点到当前节点到路径，同时用一个变量curSum记录路径节点总和，然后我们看curSum和sum是否相等，相等的话结果res加1，不等的话我们来继续查看子路径和有没有满足题意的，做法就是每次去掉一个节点，看路径和是否等于给定值，注意最后必须留一个节点，不能全去掉了，因为如果全去掉了，路径之和为0，而如果假如给定值刚好为0的话就会有问题，整体来说不算一道很难的题，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     int pathSum(TreeNode* root, int sum) {
4         int res = 0;
5         vector<TreeNode*> out;
6         helper(root, sum, 0, out, res);
7         return res;
8     }
9     void helper(TreeNode* node, int sum, int curSum, vector<TreeNode*>& out, int& res) {
10        if (!node) return;
11        curSum += node->val;
12        out.push_back(node);
13        if (curSum == sum) ++res;
14        int t = curSum;
15        for (int i = 0; i < out.size() - 1; ++i) {
16            t -= out[i]->val;
17            if (t == sum) ++res;
18        }
19        helper(node->left, sum, curSum, out, res);
20        helper(node->right, sum, curSum, out, res);
21        out.pop_back();
22    }
23 };

```

我们还可以对上面的方法进行一些优化，来去掉一些不必要的计算，我们可以用哈希表来建立所有的前缀路径之和跟其个数之间的映射，然后看子路径之和有没有等于给定值的，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int pathSum(TreeNode* root, int sum) {
4         unordered_map<int, int> m;
5         m[0] = 1;
6         return helper(root, sum, 0, m);
7     }
8     int helper(TreeNode* node, int sum, int curSum, unordered_map<int, int>& m) {
9        if (!node) return 0;
10       curSum += node->val;
11       int res = m[curSum - sum];
12       ++m[curSum];
13       res += helper(node->left, sum, curSum, m) + helper(node->right, sum, curSum, m);
14       --m[curSum];
15       return res;
16    }
17 };

```

下面这种方法非常的简洁，也是利用了前序遍历，对于每个遍历到的节点进行处理，维护一个变量pre来记录之前路径之和，然后cur为pre加上当前节点值，如果cur等于sum，那么返回结果时要加1，然后对当前节点的左右子节点调用递归函数求解，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int pathSum(TreeNode* root, int sum) {
4         if (!root) return 0;
5         return sumUp(root, 0, sum) + pathSum(root->left, sum) + pathSum(root->right, sum);
6     }
7     int sumUp(TreeNode* node, int pre, int& sum) {
8         if (!node) return 0;
9         int cur = pre + node->val;
10        return (cur == sum) + sumUp(node->left, cur, sum) + sumUp(node->right, cur, sum);
11    }
12 };

```

438. 找出字符串中所有的变位词

Given a string s and a non-empty string p, find all the start indices of p's anagrams in s.

Strings consists of lowercase English letters only and the length of both strings s and p will not be larger than 20,100.

The order of output does not matter.

Example 1:

Input:

s: "cbaebabacd" p: "abc"

Output:

[0, 6]

这道题给了我们两个字符串s和p，让我们在s中找字符串p的所有变位次的位置，所谓变位次就是字符种类个数均相同但是顺序可以不同的两个词，那么我们肯定首先就要统计字符串p中字符出现的次数，然后从s的开头开始，每次找p字符串长度个字符，来验证字符个数是否相同，如果不相同出现了直接break，如果一直都相同了，则将起始位置加入结果res中，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     vector<int> findAnagrams(string s, string p) {
4         if (s.empty()) return {};
5         vector<int> res, cnt(128, 0);
6         int ns = s.size(), np = p.size(), i = 0;
7         for (char c : p) ++cnt[c];
8         while (i < ns) {
9             bool success = true;
10            vector<int> tmp = cnt;
11            for (int j = i; j < i + np; ++j) {
12                if (--tmp[s[j]] < 0) {
13                    success = false;
14                    break;
15                }
16            }
17            if (success) {
18                res.push_back(i);
19            }
20            ++i;
21        }
22        return res;
23    }
24 };

```

我们可以将上述代码写的更加简洁一些，用两个哈希表，分别记录p的字符个数，和s中前p字符串长度的字符个数，然后比较，如果两者相同，则将0加入结果res中，然后开始遍历s中剩余的字符，每次右边加入一个新的字符，然后去掉左边的一个旧的字符，每次再比较两个哈希表是否相同即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<int> findAnagrams(string s, string p) {
4         if (s.empty()) return {};
5         vector<int> res, m1(256, 0), m2(256, 0);
6         for (int i = 0; i < p.size(); ++i) {
7             ++m1[s[i]]; ++m2[p[i]];
8         }
9         if (m1 == m2) res.push_back(0);
10        for (int i = p.size(); i < s.size(); ++i) {
11            ++m1[s[i]];
12            --m1[s[i - p.size()]];
13            if (m1 == m2) res.push_back(i - p.size() + 1);
14        }
15        return res;
16    }
17 };

```

下面这种利用滑动窗口Sliding Window的方法也比较巧妙，首先统计字符串p的字符个数，然后用两个变量left和right表示滑动窗口的左右边界，用变量cnt表示字符串p中需要匹配的字符个数，然后开始循环，如果右端界的字符已经在哈希表中了，说明该字符在p中有出现，则cnt自减1，然后哈希表中该字符个数自减1，右端界自加1，如果此时cnt减为0了，说明p中的字符都匹配上了，那么将此时左端界加入结果res中。如果此时right和left的差为p的长度，说明此时应该去掉最左边的一个字符，我们看如

果该字符在哈希表中的个数大于等于0，说明该字符是p中的字符，为啥呢，因为上面我们有让每个字符自减1，如果不是p中的字符，那么在哈希表中个数应该为0，自减1后就为-1，所以这样就知道该字符是否属于p，如果我们去掉了属于p的一个字符，cnt自增1，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     vector<int> findAnagrams(string s, string p) {
4         if (s.empty()) return {};
5         vector<int> res, m(256, 0);
6         int left = 0, right = 0, cnt = p.size(), n = s.size();
7         for (char c : p) ++m[c];
8         while (right < n) {
9             if (m[s[right++]]-- >= 1) --cnt;
10            if (cnt == 0) res.push_back(left);
11            if (right - left == p.size() && m[s[left++]]++ >= 0) ++cnt;
12        }
13        return res;
14    }
15 };

```

CPP

439. 三元表达式解析器

Given a string representing arbitrarily nested ternary expressions, calculate the result of the expression. You can always assume that the given expression is valid and only consists of digits 0-9, ?, :, T and F (T and F represent True and False respectively).

Note:

The length of the given string is ≤ 10000 .

Each number will contain only one digit.

The conditional expressions group right-to-left (as usual in most languages).

The condition will always be either T or F. That is, the condition will never be a digit.

The result of the expression will always evaluate to either a digit 0-9, T or F.

这道题让我们解析一个三元表达式，我们通过分析题目中的例子可以知道，如果有多个三元表达式嵌套的情况出现，那么我们的做法是从右边开始找到第一个问号，然后先处理这个三元表达式，然后再一步一步向左推，这也符合程序是从右向左执行的特点。那么我最先想到的方法是用一个stack来记录所有问号的位置，然后根据此问号的位置，取出当前的三元表达式，调用一个eval函数来分析得到结果，能这样做的原因是题目中限定了三元表达式每一部分只有一个字符，而且需要分析的三元表达式是合法的，然后我们把分析后的结果和前后两段拼接成一个新的字符串，继续处理之前一个问号，这样当所有问号处理完成后，所剩的一个字符就是答案，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     string parseTernary(string expression) {
4         string res = expression;
5         stack<int> s;
6         for (int i = 0; i < expression.size(); ++i) {
7             if (expression[i] == '?') s.push(i);
8         }
9         while (!s.empty()) {
10             int t = s.top(); s.pop();
11             res = res.substr(0, t - 1) + eval(res.substr(t - 1, 5)) + res.substr(t + 4);
12         }
13         return res;
14     }
15     string eval(string str) {
16         if (str.size() != 5) return "";
17         return str[0] == 'T' ? str.substr(2, 1) : str.substr(4);
18     }
19 };

```

下面这种方法也是利用栈stack的思想，但是不同之处在于不是存问号的位置，而是存所有的字符，将原数组从后往前遍历，将遍历到的字符都压入栈中，我们检测如果栈首元素是问号，说明我们当前遍历到的字符是T或F，然后我们移除问号，再取出第一部分，再移除冒号，再取出第二部分，我们根据当前字符来判断是放哪一部分进栈，这样遍历完成后，所有问号都处理完了，剩下的栈顶元素即为所求：

解法2：

```

1 class Solution {
2 public:
3     string parseTernary(string expression) {
4         stack<char> s;
5         for (int i = expression.size() - 1; i >= 0; --i) {
6             char c = expression[i];
7             if (!s.empty() && s.top() == '?') {
8                 s.pop();
9                 char first = s.top(); s.pop();
10                s.pop();
11                char second = s.top(); s.pop();
12                s.push(c == 'T' ? first : second);
13            } else {
14                s.push(c);
15            }
16        }
17        return string(1, s.top());
18    }
19 }
20 
```

下面这种方法更加简洁，没有用到栈，但是用到了STL的内置函数find_last_of，用于查找字符串中最后一个相同字符串出现的位置，这里我们找最后一个问号出现的位置，刚好就是最右边的问号，我们进行跟解法一类似的处理，拼接字符串，循环处理，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     string parseTernary(string expression) {
4         string res = expression;
5         while (res.size() > 1) {
6             int i = res.find_last_of("?");
7             res = res.substr(0, i - 1) + string(1, res[i - 1] == 'T' ? res[i + 1] : res[i +
8 3]) + res.substr(i + 4);
9         }
10        return res;
11    }
12 };

```

440. 字典顺序的第K小数字

Given integers n and k, find the lexicographically k-th smallest integer in the range from 1 to n.

Note: 1 ≤ k ≤ n ≤ 109.

Example:

Input:

n: 13 k: 2

Output:

10

这道题是之前那道Lexicographical Numbers的延伸，之前让按字典顺序打印数组，而这道题让我们快速定位某一个位置，那么我们就不能像之前那道题一样，一个一个的遍历，这样无法通过OJ，这也是这道题被定为Hard的原因。那么我们得找出能够快速定位的方法，我们如果仔细观察字典顺序的数组，我们可以发现，其实这是个十叉树Denary Tree，就是每个节点的子节点可以有十个，比如数字1的子节点就是10到19，数字10的子节点可以是100到109，但是由于n大小的限制，构成的并不是一个满十叉树。我们分析题目中给的例子可以知道，数字1的子节点有4个(10,11,12,13)，而后面的数字2到9都没有子节点，那么这道题实际上就变成了一个先序遍历十叉树的问题，那么难点就变成了如何计算出每个节点的子节点的个数，我们不停的用k减去子节点的个数，当k减到0的时候，当前位置的数字即为所求。现在我们来看如何求子节点个数，比如数字1和数字2，我们要求按字典遍历顺序从1到2需要经过多少个数字，首先把1本身这一个数字加到step中，然后我们把范围扩大十倍，范围变成10到20之前，但是由于我们要考虑n的大小，由于n为13，所以只有4个子节点，这样我们就知道从数字1遍历到数字2需要经过5个数字，然后我们看step是否小于等于k，如果是，我们cur自增1，k减去step；如果不是，说明要求的数字在子节点中，我们此时cur乘以10，k自减1，以此类推，直到k为0推出循环，此时cur即为所求：

```

1 class Solution {
2 public:
3     int findKthNumber(int n, int k) {
4         int cur = 1;
5         --k;
6         while (k > 0) {
7             long long step = 0, first = cur, last = cur + 1;
8             while (first <= n) {
9                 step += min((long long)n + 1, last) - first;
10                first *= 10;
11                last *= 10;
12            }
13            if (step <= k) {
14                ++cur;
15                k -= step;
16            } else {
17                cur *= 10;
18                --k;
19            }
20        }
21        return cur;
22    }
23 };

```

441. 排列硬币

You have a total of n coins that you want to form in a staircase shape, where every k -th row must have exactly k coins.

Given n , find the total number of full staircase rows that can be formed.

n is a non-negative integer and fits within the range of a 32-bit signed integer.

这道题给了我们 n 个硬币，让我们按一定规律排列，第一行放1个，第二行放2个，以此类推，问我们有多少行能放满。通过分析题目中的例子可以得知最后一行只有两种情况，放满和没放满。由于是按等差数列排放的，我们可以快速计算出前 i 行的硬币总数。我们先来看一种 $O(n)$ 的方法，非常简单粗暴，就是从第一行开始，一行一行的从 n 中减去，如果此时剩余的硬币没法满足下一行需要的硬币数了，我们之间返回当前行数即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int arrangeCoins(int n) {
4         int cur = 1, rem = n - 1;
5         while (rem >= cur + 1) {
6             ++cur;
7             rem -= cur;
8         }
9         return n == 0 ? 0 : cur;
10    }
11 };

```

再来看一种 $O(\lg n)$ 的方法，用到了二分搜索法，我们搜索前 i 行之和刚好大于 n 的临界点，这样我们减一个就是能排满的行数，注意我们计算前 i 行之和有可能会整型溢出，所以我们需要将变量都定义成长整型，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int arrangeCoins(int n) {
4         if (n <= 1) return n;
5         long low = 1, high = n;
6         while (low < high) {
7             long mid = low + (high - low) / 2;
8             if (mid * (mid + 1) / 2 <= n) low = mid + 1;
9             else high = mid;
10        }
11        return low - 1;
12    }
13 };

```

再来看一种数学解法O(1)，充分利用了等差数列的性质，我们建立等式， $n = (1 + x) * x / 2$ ，我们用一元二次方程的求根公式可以得到 $x = (-1 + \sqrt{8 * n + 1}) / 2$ ，然后取整后就是能填满的行数，一行搞定简直丧心病狂啊：

解法3：

```

1 class Solution {
2 public:
3     int arrangeCoins(int n) {
4         return (int)((-1 + sqrt(1 + 8 * (long)n)) / 2);
5     }
6 };

```

442. 找出数组中所有重复项

Given an array of integers, $1 \leq a[i] \leq n$ ($n = \text{size of array}$), some elements appear twice and others appear once.

Find all the elements that appear twice in this array.

Could you do it without extra space and in $O(n)$ runtime?

这道题给了我们一个数组，数组中的数字可能出现一次或两次，让我们找出所有出现两次的数字，由于之前做过一道类似的题目 Find the Duplicate Number，所以不是完全无从下手。这类问题的一个重要条件就是 $1 \leq a[i] \leq n$ ($n = \text{size of array}$)，不然很难在 $O(1)$ 空间和 $O(n)$ 时间内完成。首先来看一种正负替换的方法，这类问题的核心是就是找 $\text{nums}[i]$ 和 $\text{nums}[\text{nums}[i] - 1]$ 的关系，我们的做法是，对于每个 $\text{nums}[i]$ ，我们将其对应的 $\text{nums}[\text{nums}[i] - 1]$ 取相反数，如果其已经是负数了，说明之前存在过，我们将其加入结果 res 中即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<int> findDuplicates(vector<int>& nums) {
4         vector<int> res;
5         for (int i = 0; i < nums.size(); ++i) {
6             int idx = abs(nums[i]) - 1;
7             if (nums[idx] < 0) res.push_back(idx + 1);
8             nums[idx] = -nums[idx];
9         }
10        return res;
11    }
12 };

```

下面这种方法是将`nums[i]`置换到其对应的位置`nums[nums[i]-1]`上去，比如对于没有重复项的正确的顺序应该是[1, 2, 3, 4, 5, 6, 7, 8]，而我们现在却是[4,3,2,7,8,2,3,1]，我们需要把数字移动到正确的位置上去，比如第一个4就应该和7先交换个位置，以此类推，最后得到的顺序应该是[1, 2, 3, 4, 3, 2, 7, 8]，我们最后在对应位置检验，如果`nums[i]`和`i+1`不等，那么我们将`nums[i]`存入结果`res`中即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<int> findDuplicates(vector<int>& nums) {
4         vector<int> res;
5         for (int i = 0; i < nums.size(); ++i) {
6             if (nums[i] != nums[nums[i] - 1]) {
7                 swap(nums[i], nums[nums[i] - 1]);
8                 --i;
9             }
10        }
11        for (int i = 0; i < nums.size(); ++i) {
12            if (nums[i] != i + 1) res.push_back(nums[i]);
13        }
14        return res;
15    }
16 };

```

下面这种方法是在`nums[nums[i]-1]`位置累加数组长度n，注意`nums[i]-1`有可能越界，所以我们需要对n取余，最后要找出两次的数只需要看`nums[i]`的值是否大于`2n`即可，最后遍历完`nums[i]`数组为[12, 19, 18, 15, 8, 2, 11, 9]，我们发现有两个数字19和18大于`2n`，那么就可以通过`i+1`来得到正确的结果2和3了，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     vector<int> findDuplicates(vector<int>& nums) {
4         vector<int> res;
5         int n = nums.size();
6         for (int i = 0; i < n; ++i) {
7             nums[(nums[i] - 1) % n] += n;
8         }
9         for (int i = 0; i < n; ++i) {
10             if (nums[i] > 2 * n) res.push_back(i + 1);
11         }
12     return res;
13 }
14 };

```

443. 字符串压缩

Given an array of characters, compress it in-place.

The length after compression must always be smaller than or equal to the original array.

Every element of the array should be a character (not int) of length 1.

After you are done modifying the input array in-place, return the new length of the array.

Follow up:

Could you solve it using only O(1) extra space?

这道题给了我们一个字符串，让我们进行压缩，即相同的字符统计出个数，显示在该字符之后，根据例子分析不难理解题意。这道题要求我们进行in place操作，即不使用额外空间，最后让我们返回修改后的新数组的长度。我们首先想，数组的字符不一定是有顺序的，如果我们用Map来建立字符和出现次数之间的映射，不管是用HashMap还是TreeMap，一定无法保证原有的顺序。所以不能用Map，而我们有需要统计个数，那么双指针就是不二之选啦。既然双指针，其中一个指针指向重复字符串的第一个，然后另一个指针向后遍历并计数，就能得到重复的个数。我们仔细研究例子3，可以发现，当个数是两位数的时候，比如12，这里是将12拆分成1和2，然后存入数组的。那么比较简便的提取出各个位上的数字的办法就是转为字符串进行遍历。另外，由于我们需要对原数组进行修改，则需要一个指针cur来标记下一个可以修改的位置，那么最终cur的值就是新数组的长度，直接返回即可。

具体来看代码，我们用i和j表示双指针，开始循环后，我们用j来找重复的字符串的个数，用一个while循环，最终j指向的是第一个和i指向字符不同的地方，此时我们需要先将i位置的字符写进chars中，然后我们判断j是否比i正好大一个，因为只有一个字符的话，后面是不用加个数的，所以直接跳过。否则我们将重复个数转为字符串，然后提取出来修改chars数组即可，注意每次需要将i赋值为j，从而开始下一个字符的统计，参见代码如下：

```

1 class Solution {
2 public:
3     int compress(vector<char>& chars) {
4         int n = chars.size(), cur = 0;
5         for (int i = 0, j = 0; i < n; i = j) {
6             while (j < n && chars[j] == chars[i]) ++j;
7             chars[cur++] = chars[i];
8             if (j - i == 1) continue;
9             for (char c : to_string(j - i)) chars[cur++] = c;
10        }
11        return cur;
12    }
13 };

```

444. 序列重建

Check whether the original sequence org can be uniquely reconstructed from the sequences in seqs. The org sequence is a permutation of the integers from 1 to n, with $1 \leq n \leq 104$. Reconstruction means building a shortest common supersequence of the sequences in seqs (i.e., a shortest sequence so that all sequences in seqs are subsequences of it). Determine whether there is only one sequence that can be reconstructed from seqs and it is the org sequence.

这道题给了我们一个序列org，又给我们了一些子序列seqs，问这些子序列能否唯一的重建出原序列。能唯一重建的意思就是任意两个数字的顺序必须是一致的，不能说在一个子序列中1在4的后面，但是在另一个子序列中1在4的前面，这样就不是唯一的了。还有一点就是，子序列seqs中不能出现其他的数字，就是说必须都是原序列中的数字。那么我们可以用了一个一维数组pos来记录org中每个数字对应的位置，然后用一个flags数字来标记当前数字和其前面一个数字是否和org中的顺序一致，用cnt来标记还需要验证顺序的数字的个数，初始化cnt为n-1，因为n个数字只需要验证n-1对顺序即可，然后我们先遍历一遍org，将每个数字的位置信息存入pos中，然后再遍历子序列中的每一个数字，还是要先判断数字是否越界，然后我们取出当前数字cur，和其前一位置上的数字pre，如果在org中，pre在cur之后，那么直接返回false。否则我们看如果cur的顺序没被验证过，而且pre是在cur的前一个，那么标记cur已验证，且cnt自减1，最后如果cnt为0了，说明所有顺序被成功验证了，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool sequenceReconstruction(vector<int>& org, vector<vector<int>>& seqs) {
4         if (seqs.empty()) return false;
5         int n = org.size(), cnt = n - 1;
6         vector<int> pos(n + 1, 0), flags(n + 1, 0);
7         bool existed = false;
8         for (int i = 0; i < n; ++i) pos[org[i]] = i;
9         for (auto& seq : seqs) {
10             for (int i = 0; i < seq.size(); ++i) {
11                 existed = true;
12                 if (seq[i] <= 0 || seq[i] > n) return false;
13                 if (i == 0) continue;
14                 int pre = seq[i - 1], cur = seq[i];
15                 if (pos[pre] >= pos[cur]) return false;
16                 if (flags[cur] == 0 && pos[pre] + 1 == pos[cur]) {
17                     flags[cur] = 1; --cnt;
18                 }
19             }
20         }
21         return cnt == 0 && existed;
22     }
23 };

```

下面这种方法跟上面的方法大同小异，用两个哈希表来代替了上面的数组和变量，其中m为数字与其位置之间的映射，pre为当前数字和其前一个位置的数字在org中的位置之间的映射。跟上面的方法的不同点在于，当遍历到某一个数字的时候，我们看当前数字是否在pre中有映射，如果没有的话，我们建立该映射，注意如果是第一个位置的数字的话，其前面数字设为-1。如果该映射存在的话，我们对比前一位数字在org中的位置和当前的映射值的大小，取其中较大值。最后我们遍历一遍org，看每个数字的映射值是否是前一个数字的位置，如果有不是的返回false，全部验证成功返回true，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool sequenceReconstruction(vector<int>& org, vector<vector<int>>& seqs) {
4         unordered_map<int, int> m, pre;
5         for (int i = 0; i < org.size(); ++i) m[org[i]] = i;
6         for (auto& seq : seqs) {
7             for (int i = 0; i < seq.size(); ++i) {
8                 if (!m.count(seq[i])) return false;
9                 if (i > 0 && m[seq[i - 1]] >= m[seq[i]]) return false;
10                if (!pre.count(seq[i])) {
11                    pre[seq[i]] = (i > 0) ? m[seq[i - 1]] : -1;
12                } else {
13                    pre[seq[i]] = max(pre[seq[i]], (i > 0) ? m[seq[i - 1]] : -1);
14                }
15            }
16        }
17        for (int i = 0; i < org.size(); ++i) {
18            if (pre[org[i]] != i - 1) return false;
19        }
20        return true;
21    }
22};

```

445. 两个数字相加之二

You are given two linked lists representing two non-negative numbers. The most significant digit comes first and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.

You may assume the two numbers do not contain any leading zero, except the number 0 itself.

Follow up:

What if you cannot modify the input lists? In other words, reversing the lists is not allowed.

Example:

Input: (7 -> 2 -> 4 -> 3) + (5 -> 6 -> 4)

Output: 7 -> 8 -> 0 -> 7

这道题是之前那道Add Two Numbers的拓展，我们可以看到这道题的最高位在链表首位置，如果我们给链表翻转一下的话就跟之前的题目一样了，这里我们来看一些不修改链表顺序的方法。由于加法需要从最低位开始运算，而最低位在链表末尾，链表只能从前往后遍历，没法取到前面的元素，那怎么办呢？我们可以利用栈来保存所有的元素，然后利用栈的后进先出的特点就可以从后往前取数字了，我们首先遍历两个链表，将所有数字分别压入两个栈s1和s2中，我们建立一个值为0的res节点，然后开始循环，如果栈不为空，则将栈顶数字加入sum中，然后将res节点值赋为sum%10，然后新建一个进位节点head，赋值为sum/10，如果没有进位，那么就是0，然后我们head后面连上res，将res指向head，这样循环退出后，我们只要看res的值是否为0，为0返回res->next，不为0则返回res即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
4         stack<int> s1, s2;
5         while (l1) {
6             s1.push(l1->val);
7             l1 = l1->next;
8         }
9         while (l2) {
10            s2.push(l2->val);
11            l2 = l2->next;
12        }
13        int sum = 0;
14        ListNode *res = new ListNode(0);
15        while (!s1.empty() || !s2.empty()) {
16            if (!s1.empty()) {sum += s1.top(); s1.pop();}
17            if (!s2.empty()) {sum += s2.top(); s2.pop();}
18            res->val = sum % 10;
19            ListNode *head = new ListNode(sum / 10);
20            head->next = res;
21            res = head;
22            sum /= 10;
23        }
24        return res->val == 0 ? res->next : res;
25    }
26 };

```

CPP

下面这种方法使用递归来实现的，我们知道递归其实也是用栈来保存每一个状态，那么也就可以实现从后往前取数字，我们首先统计出两个链表长度，然后根据长度来调用递归函数，需要传一个参数差值，递归函数参数中的l1链表长度长于l2，在递归函数中，我们建立一个节点res，如果差值不为0，节点值为l1的值，如果为0，那么就是l1和l2的和，然后在根据差值分别调用递归函

数求出节点post，然后要处理进位，如果post的值大于9，那么对10取余，且res的值自增1，然后把pos连到res后面，返回res，最后回到原函数中，我们仍要处理进位情况，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
4         int n1 = getLength(l1), n2 = getLength(l2);
5         ListNode *head = new ListNode(1);
6         head->next = (n1 > n2) ? helper(l1, l2, n1 - n2) : helper(l2, l1, n2 - n1);
7         if (head->next->val > 9) {
8             head->next->val %= 10;
9             return head;
10        }
11        return head->next;
12    }
13    int getLength(ListNode* head) {
14        int cnt = 0;
15        while (head) {
16            ++cnt;
17            head = head->next;
18        }
19        return cnt;
20    }
21    ListNode* helper(ListNode* l1, ListNode* l2, int diff) {
22        if (!l1) return NULL;
23        ListNode *res = (diff == 0) ? new ListNode(l1->val + l2->val) : new ListNode(l1->val);
24        ListNode *post = (diff == 0) ? helper(l1->next, l2->next, 0) : helper(l1->next, l2,
25 diff - 1);
26        if (post && post->val > 9) {
27            post->val %= 10;
28            ++res->val;
29        }
30        res->next = post;
31        return res;
32    }
33};

```

下面这种方法借鉴了Plus One Linked List中的解法三，在处理加1问题时，我们需要找出右起第一个不等于9的位置，然后此位置值自增1，之后的全部赋为0。这里我们同样要先算出两个链表的长度，我们把其中较长的放在l1，然后我们算出两个链表长度差diff。如果diff大于0，我们用l1的值新建节点，并连在cur节点后(cur节点初始化时指向dummy节点)。并且如果l1的值不等于9，那么right节点也指向这个新建的节点，然后cur和l1都分别后移一位，diff自减1。当diff为0后，我们循环遍历，将此时l1和l2的值加起来放入变量val中，如果val大于9，那么val对10取余，right节点自增1，将right后面节点全赋值为0。在cur节点后新建节点，节点值为更新后的val，如果val的值不等于9，那么right节点也指向这个新建的节点，然后cur，l1和l2都分别后移一位。最后我们看dummy节点值若为1，返回dummy节点，如果是0，则返回dummy的下一个节点。

解法3：

```

1 class Solution {
2 public:
3     ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
4         int n1 = getLength(l1), n2 = getLength(l2), diff = abs(n1 - n2);
5         if (n1 < n2) swap(l1, l2);
6         ListNode *dummy = new ListNode(0), *cur = dummy, *right = cur;
7         while (diff > 0) {
8             cur->next = new ListNode(l1->val);
9             if (l1->val != 9) right = cur->next;
10            cur = cur->next;
11            l1 = l1->next;
12            --diff;
13        }
14        while (l1) {
15            int val = l1->val + l2->val;
16            if (val > 9) {
17                val %= 10;
18                ++right->val;
19                while (right->next) {
20                    right->next->val = 0;
21                    right = right->next;
22                }
23                right = cur;
24            }
25            cur->next = new ListNode(val);
26            if (val != 9) right = cur->next;
27            cur = cur->next;
28            l1 = l1->next;
29            l2 = l2->next;
30        }
31        return (dummy->val == 1) ? dummy : dummy->next;
32    }
33    int getLength(ListNode* head) {
34        int cnt = 0;
35        while (head) {
36            ++cnt;
37            head = head->next;
38        }
39        return cnt;
40    }
41 };
42

```

446. 算数切片之二 - 子序列

A sequence of numbers is called arithmetic if it consists of at least three elements and if the difference between any two consecutive elements is the same.

For example, these are arithmetic sequences:

1, 3, 5, 7, 9
7, 7, 7, 7
3, -1, -5, -9

The following sequence is not arithmetic.

1, 1, 2, 5, 7

A zero-indexed array A consisting of N numbers is given. A subsequence slice of that array is any sequence of integers (P_0, P_1, \dots, P_k) such that $0 \leq P_0 < P_1 < \dots < P_k < N$.

A subsequence slice (P_0, P_1, \dots, P_k) of array A is called arithmetic if the sequence $A[P_0], A[P_1], \dots, A[P_{k-1}], A[P_k]$ is arithmetic. In particular, this means that $k \geq 2$.

The function should return the number of arithmetic subsequence slices in the array A.

The input contains N integers. Every integer is in the range of -231 and 231-1 and $0 \leq N \leq 1000$. The output is guaranteed to be less than 231-1.

这道题是之前那道Arithmetic Slices的延伸，那道题比较简单是因为要求等差数列是连续的，而这道题让我们求是等差数列的子序列，可以跳过某些数字，不一定非得连续，那么难度就加大了，但还是需要用动态规划Dynamic Programming来做。知道用DP来做是一回事，真正能做出来又是另一回事。刷题的最终目的不是背题，而是训练思维方式，如何从完全没思路，变为好像有点思路，慢慢推理演变找出正确解，就像顺着藏宝图的丝丝线索最终发现了宝藏一样，是无比的令人激动和富有成就感的过程。死记硬背的话，只要题目稍稍变形一下就完蛋。这就是为啥博主喜欢看fun4LeetCode大神的帖子，虽然看范佛力扣大神的帖子像读论文，但是有推理过程，看完让人神清气爽，茶饭不思，燃鹅遇到同类型的还是会，还是要再看。好，博主皮一下就行了，下面来顺着大神的帖子来讲吧。

好，既然决定要用DP了，那么首先就要确定dp数组的定义了，刚开始我们可能会考虑使用个一维的dp数组，然后 $dp[i]$ 定义为范围为[0, i]的子数组中等差数列的个数。定义的很简单，OK，但是基于这种定义的递归式却十分的难想。我们想对于(0, i)之间的任意位置j，如何让 $dp[i]$ 和 $dp[j]$ 产生关联呢？是不是只有 $A[i]$ 和 $A[j]$ 的差值diff，跟 $A[j]$ 之前等差数列的差值相同，才会有关系，所以差值diff是一个很重要的隐藏信息Hidden Information，我们必须要在dp的定义中考虑进去。所以一维dp数组是罩不住的，必须升维，但是用二维dp数组的话，差值diff那一维的范围又是个问题，数字的范围是整型数，所以差值的范围也很大，为了节省空间，我们建立一个一维数组dp，数组里的元素不是数字，而是放一个HashMap，建立等差数列的差值和当前位置之前差值相同的数字个数之间的映射。我们遍历数组中的所有数字，对于当前遍历到的数字，又从开头遍历到当前数字，计算两个数字之差diff，如果越界了不做任何处理，如果没越界，我们让 $dp[i]$ 中diff的差值映射自增1，因为此时 $A[i]$ 前面有相差为diff的 $A[j]$ ，所以映射值要加1。然后我们看 $dp[j]$ 中是否有diff的映射，如果有的话，说明此时相差为diff的数字至少有三个了，已经能构成题目要求的等差数列了，将 $dp[j][diff]$ 加入结果res中，然后再更新 $dp[i][diff]$ ，这样等遍历完数组，res即为所求。

我们用题目中给的例子数组 [2, 4, 6, 8, 10] 来看，因为2之前没有数字了，所以我们从4开始，遍历前面的数字，是2，二者差值为2，那么在 $dp[1]$ 的HashMap就可以建立 2→1 的映射，表示4之前有1个差值为2的数字，即数字2。那么现在i=2指向6了，遍历前面的数字，第一个数是2，二者相差4，那么在 $dp[2]$ 的HashMap就可以建立 4→1 的映射，第二个数是4，二者相差2，那么先在 $dp[2]$ 的HashMap建立 2→1 的映射，由于 $dp[1]$ 的HashMap中也有差值为2的映射，2→1，那么说明此时至少有三个数字差值相同，即这里的 [2 4 6]，我们将 $dp[1]$ 中的映射值加入结果res中，然后当前 $dp[2]$ 中的映射值加上 $dp[1]$ 中的映射值。这应该不难理解，比如当i=3指向数字8时，j=2指向数字6，那么二者差值为2，此时先在 $dp[3]$ 建立 2→1 的映射，由于 $dp[2]$ 中有 2→2 的映射，那么加上数字8其实新增了两个等差数列 [2,4,6,8] 和 [4,6,8]，所以结果res加上的值就是 $dp[j][diff]$ ，即2，并且 $dp[i][diff]$ 也需要加上这个值，才能使得 $dp[3]$ 中的映射变为 2→3，后面数字10的处理情况也相同，这里就不多赘述了，最终的各个位置的映射关系如下所示：

```

2      4      6      8      10
 2->1  4->1  6->1  8->1
 2->2  4->1  6->1
 2->3  4->2
 2->4

```

最终累计出来的结果是跟上面红色的数字相关，分别对应着如下的等差数列：

2->2: [2,4,6]

2->3: [2,4,6,8] [4,6,8]

4->2: [2,6,10]

2->4: [2,4,6,8,10] [4,6,8,10] [6,8,10]

```

1 class Solution {
2 public:
3     int numberOfArithmeticSlices(vector<int>& A) {
4         int res = 0, n = A.size();
5         vector<unordered_map<int, int>> dp(n);
6         for (int i = 0; i < n; ++i) {
7             for (int j = 0; j < i; ++j) {
8                 long delta = (long)A[i] - A[j];
9                 if (delta > INT_MAX || delta < INT_MIN) continue;
10                int diff = (int)delta;
11                ++dp[i][diff];
12                if (dp[j].count(diff)) {
13                    res += dp[j][diff];
14                    dp[i][diff] += dp[j][diff];
15                }
16            }
17        }
18        return res;
19    }
20 };

```

CPP

447. 回旋镖的数量

Given n points in the plane that are all pairwise distinct, a "boomerang" is a tuple of points (i, j, k) such that the distance between i and j equals the distance between i and k (the order of the tuple matters).

Find the number of boomerangs. You may assume that n will be at most 500 and coordinates of points are all in the range $[-10000, 10000]$ (inclusive).

这道题定义了一种类似回旋镖形状的三元组结构，要求第一个点和第二个点之间的距离跟第一个点和第三个点之间的距离相等。现在给了我们 n 个点，让我们找出回旋镖的个数。那么我们想，如果我们有一个点a，还有两个点b和c，如果ab和ac之间的距离相等，那么就有两种排列方法abc和acb；如果有三个点b, c, d都分别和a之间的距离相等，那么有六种排列方法，abc, acb, acd, adc, abd, adb，那么是怎么算出来的呢，很简单，如果有 n 个点和a距离相等，那么排列方式为 $n(n-1)$ ，这属于最简单的排列组合问题了，我大天朝中学生都会做的。那么我们问题就变成了遍历所有点，让每个点都做一次点a，然后遍历其他所有点，统计和a距离相等的点有多少个，然后分别带入 $n(n-1)$ 计算结果并累加到res中，只有当n大于等于2时，res值才会真正增加，参见代码如下：

```

1 class Solution {
2 public:
3     int numberOfBoomerangs(vector<pair<int, int>>& points) {
4         int res = 0;
5         for (int i = 0; i < points.size(); ++i) {
6             unordered_map<int, int> m;
7             for (int j = 0; j < points.size(); ++j) {
8                 int a = points[i].first - points[j].first;
9                 int b = points[i].second - points[j].second;
10                ++m[a * a + b * b];
11            }
12            for (auto it = m.begin(); it != m.end(); ++it) {
13                res += it->second * (it->second - 1);
14            }
15        }
16        return res;
17    }
18 };

```

448. 找出数组中所有消失的数字

Given an array of integers where $1 \leq a[i] \leq n$ ($n = \text{size of array}$), some elements appear twice and others appear once.

Find all the elements of $[1, n]$ inclusive that do not appear in this array.

Could you do it without extra space and in $O(n)$ runtime? You may assume the returned list does not count as extra space.

这道题让我们找出数组中所有消失的数，跟之前那道Find All Duplicates in an Array极其类似，那道题让找出所有重复的数字，这道题让找不存在的数，这类问题的一个重要条件就是 $1 \leq a[i] \leq n$ ($n = \text{size of array}$)，不然很难在 $O(1)$ 空间和 $O(n)$ 时间内完成。三种解法也跟之前题目的解法极其类似。首先来看第一种解法，这种解法的思路是，对于每个数字 $\text{nums}[i]$ ，如果其对应的 $\text{nums}[\text{nums}[i] - 1]$ 是正数，我们就赋值为其相反数，如果已经是负数了，就不变了，那么最后我们只要把留下的整数对应的位置加入结果 res 中即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<int> findDisappearedNumbers(vector<int>& nums) {
4         vector<int> res;
5         for (int i = 0; i < nums.size(); ++i) {
6             int idx = abs(nums[i]) - 1;
7             nums[idx] = (nums[idx] > 0) ? -nums[idx] : nums[idx];
8         }
9         for (int i = 0; i < nums.size(); ++i) {
10            if (nums[i] > 0) {
11                res.push_back(i + 1);
12            }
13        }
14        return res;
15    }
16 };

```

第二种方法是将`nums[i]`置换到其对应的位置`nums[nums[i]-1]`上去，比如对于没有缺失项的正确的顺序应该是`[1, 2, 3, 4, 5, 6, 7, 8]`，而我们现在却是`[4,3,2,7,8,2,3,1]`，我们需要把数字移动到正确的位置上去，比如第一个4就应该和7先交换个位置，以此类推，最后得到的顺序应该是`[1, 2, 3, 4, 3, 2, 7, 8]`，我们最后在对应位置检验，如果`nums[i]`和`i+1`不等，那么我们将`i+1`存入结果`res`中即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<int> findDisappearedNumbers(vector<int>& nums) {
4         vector<int> res;
5         for (int i = 0; i < nums.size(); ++i) {
6             if (nums[i] != nums[nums[i] - 1]) {
7                 swap(nums[i], nums[nums[i] - 1]);
8                 --i;
9             }
10        }
11        for (int i = 0; i < nums.size(); ++i) {
12            if (nums[i] != i + 1) {
13                res.push_back(i + 1);
14            }
15        }
16        return res;
17    }
18 };

```

CPP

下面这种方法是在`nums[nums[i]-1]`位置累加数组长度`n`，注意`nums[i]-1`有可能越界，所以我们需要对`n`取余，最后要找出缺失的数只需要看`nums[i]`的值是否小于等于`n`即可，最后遍历完`nums[i]`数组为`[12, 19, 18, 15, 8, 2, 11, 9]`，我们发现有两个数字8和2小于等于`n`，那么就可以通过`i+1`来得到正确的结果5和6了，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     vector<int> findDisappearedNumbers(vector<int>& nums) {
4         vector<int> res;
5         int n = nums.size();
6         for (int i = 0; i < n; ++i) {
7             nums[(nums[i] - 1) % n] += n;
8         }
9         for (int i = 0; i < n; ++i) {
10            if (nums[i] <= n) {
11                res.push_back(i + 1);
12            }
13        }
14        return res;
15    }
16 };

```

CPP

449. 二叉搜索树的序列化和去序列化

Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be reconstructed later in the same or another computer environment.

Design an algorithm to serialize and deserialize a binary search tree. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary search tree can be serialized to a string and this string can be deserialized to the original tree structure.

The encoded string should be as compact as possible.

Note: Do not use class member/global/static variables to store states. Your serialize and deserialize algorithms should be stateless.

这道题让我们对二叉搜索树序列化和去序列化，跟之前那道Serialize and Deserialize Binary Tree极其相似，虽然题目中说编码成的字符串要尽可能的紧凑，但是我们并没有发现跟之前那题有何不同，而且也没有看到能够利用BST性质的方法，姑且就按照之前题目的解法来写吧：

解法1：

```

1  class Codec {
2  public:
3
4      // Encodes a tree to a single string.
5      string serialize(TreeNode* root) {
6          ostringstream os;
7          serialize(root, os);
8          return os.str();
9      }
10
11     // Decodes your encoded data to tree.
12     TreeNode* deserialize(string data) {
13         istringstream is(data);
14         return deserialize(is);
15     }
16
17     void serialize(TreeNode* root, ostringstream& os) {
18         if (!root) os << "# ";
19         else {
20             os << root->val << " ";
21             serialize(root->left, os);
22             serialize(root->right, os);
23         }
24     }
25
26     TreeNode* deserialize(istringstream& is) {
27         string val = "";
28         is >> val;
29         if (val == "#") return NULL;
30         TreeNode* node = new TreeNode(stoi(val));
31         node->left = deserialize(is);
32         node->right = deserialize(is);
33         return node;
34     }
35 };

```

CPP

另一种方法是层序遍历的非递归解法，这种方法略微复杂一些，我们需要借助queue来做，本质是BFS算法，也不是很难理解，就是BFS算法的常规套路稍作修改即可，参见代码如下：

解法2：

```

1  class Codec {
2  public:
3
4      // Encodes a tree to a single string.
5      string serialize(TreeNode* root) {
6          if (!root) return "";
7          ostringstream os;
8          queue<TreeNode*> q;
9          q.push(root);
10         while (!q.empty()) {
11             TreeNode *t = q.front(); q.pop();
12             if (t) {
13                 os << t->val << " ";
14                 q.push(t->left);
15                 q.push(t->right);
16             } else {
17                 os << "# ";
18             }
19         }
20         return os.str();
21     }
22
23     // Decodes your encoded data to tree.
24     TreeNode* deserialize(string data) {
25         if (data.empty()) return NULL;
26         istringstream is(data);
27         queue<TreeNode*> q;
28         string val = "";
29         is >> val;
30         TreeNode *res = new TreeNode(stoi(val)), *cur = res;
31         q.push(cur);
32         while (!q.empty()) {
33             TreeNode *t = q.front(); q.pop();
34             if (!(is >> val)) break;
35             if (val != "#") {
36                 cur = new TreeNode(stoi(val));
37                 q.push(cur);
38                 t->left = cur;
39             }
40             if (!(is >> val)) break;
41             if (val != "#") {
42                 cur = new TreeNode(stoi(val));
43                 q.push(cur);
44                 t->right = cur;
45             }
46         }
47         return res;
48     }
49 };

```

450. 删除二叉搜索树中的节点

Given a root node reference of a BST and a key, delete the node with the given key in the BST. Return the root node reference (possibly updated) of the BST.

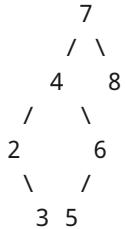
Basically, the deletion can be divided into two stages:

Search for a node to remove.

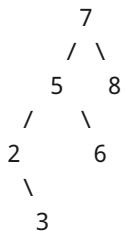
If the node is found, delete the node.

Note: Time complexity should be O(height of tree).

这道题让我们删除二叉搜索树中的一个节点，这道题的难点在于删除完节点并补上那个节点的位置后还应该是一棵二叉搜索树。被删除掉的节点位置，不一定是由其的左右子节点补上，比如下面这棵树：



如果我们要删除节点4，那么应该将节点5补到4的位置，这样才能保证还是BST，那么结果是如下这棵树：



我们先来看一种递归的解法，首先判断根节点是否为空。由于BST的左<根<右的性质，使得我们可以快速定位到要删除的节点，我们对于当前节点值不等于key的情况，根据大小关系对其左右子节点分别调用递归函数。若当前节点就是要删除的节点，我们首先判断是否有一个子节点不存在，那么我们就将root指向另一个节点，如果左右子节点都不存在，那么root就赋值为空了，也正确。难点就在于处理左右子节点都存在的情况，我们需要在右子树找到最小值，即右子树中最左下方的节点，然后将该最小值赋值给root，然后再在右子树中调用递归函数来删除这个值最小的节点，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     TreeNode* deleteNode(TreeNode* root, int key) {
4         if (!root) return NULL;
5         if (root->val > key) {
6             root->left = deleteNode(root->left, key);
7         } else if (root->val < key) {
8             root->right = deleteNode(root->right, key);
9         } else {
10            if (!root->left || !root->right) {
11                root = (root->left) ? root->left : root->right;
12            } else {
13                TreeNode *cur = root->right;
14                while (cur->left) cur = cur->left;
15                root->val = cur->val;
16                root->right = deleteNode(root->right, cur->val);
17            }
18        }
19        return root;
20    }
21 };

```

下面我们来看迭代的写法，还是通过BST的性质来快速定位要删除的节点，如果没找到直接返回空。遍历的过程要记录上一个位置的节点pre，如果pre不存在，说明要删除的是根节点，如果要删除的节点在pre的左子树中，那么pre的左子节点连上删除后的节点，反之pre的右子节点连上删除后的节点。在删除函数中，如果左右子节点都不存在，那么返回空；如果有一个不存在，那么我们返回那个存在的；难点还是在于处理左右子节点都存在的情况，还是要找到需要删除节点的右子树中的最小值，然后把最小值赋值给要删除节点，然后就是要处理最小值可能存在的右子树的连接问题，如果要删除节点的右子节点没有左子节点了的话，那么最小值的右子树直接连到要删除节点的右子节点上即可(因为此时原本要删除的节点的值已经被最小值替换了，所以现在其实是要删掉最小值节点)。否则我们就把最小值节点的右子树连到其父节点的左子节点上。文字表述确实比较绕，请大家自行带例子一步一步观察就会很清晰明了了，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     TreeNode* deleteNode(TreeNode* root, int key) {
4         TreeNode *cur = root, *pre = NULL;
5         while (cur) {
6             if (cur->val == key) break;
7             pre = cur;
8             if (cur->val > key) cur = cur->left;
9             else cur = cur->right;
10        }
11        if (!cur) return root;
12        if (!pre) return del(cur);
13        if (pre->left && pre->left->val == key) pre->left = del(cur);
14        else pre->right = del(cur);
15        return root;
16    }
17    TreeNode* del(TreeNode* node) {
18        if (!node->left && !node->right) return NULL;
19        if (!node->left || !node->right) {
20            return (node->left) ? node->left : node->right;
21        }
22        TreeNode *pre = node, *cur = node->right;
23        while (cur->left) {
24            pre = cur;
25            cur = cur->left;
26        }
27        node->val = cur->val;
28        (pre == node ? node->right : pre->left) = cur->right;
29        return node;
30    }
31 };

```

下面来看一种对于二叉树通用的解法，适用于所有二叉树，所以并没有利用BST的性质，而是遍历了所有的节点，然后删掉和key值相同的节点，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     TreeNode* deleteNode(TreeNode* root, int key) {
4         if (!root) return NULL;
5         if (root->val == key) {
6             if (!root->right) return root->left;
7             else {
8                 TreeNode *cur = root->right;
9                 while (cur->left) cur = cur->left;
10                swap(root->val, cur->val);
11            }
12        }
13        root->left = deleteNode(root->left, key);
14        root->right = deleteNode(root->right, key);
15        return root;
16    }
17 };

```

451. 根据字符出现频率排序

Given a string, sort it in decreasing order based on the frequency of characters.

Example 1:

Input:
"tree"

Output:
"eert"

这道题让我们给一个字符串按照字符出现的频率来排序，那么毫无疑问肯定要先统计出每个字符出现的个数，那么之后怎么做呢？我们可以利用优先队列的自动排序的特点，把个数和字符组成pair放到优先队列里排好序后，再取出来组成结果res即可，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     string frequencySort(string s) {
4         string res = "";
5         priority_queue<pair<int, char>> q;
6         unordered_map<char, int> m;
7         for (char c : s) ++m[c];
8         for (auto a : m) q.push({a.second, a.first});
9         while (!q.empty()) {
10             auto t = q.top(); q.pop();
11             res.append(t.first, t.second);
12         }
13         return res;
14     }
15 }
```

CPP

我们也可以使用STL自带的sort来做，关键就在于重写comparator，由于需要使用外部变量，记得中括号中放入&，然后我们将频率大的返回，注意一定还要处理频率相等的情况，要不然两个频率相等的字符可能穿插着出现在结果res中，这样是不对的。参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     string frequencySort(string s) {
4         unordered_map<char, int> m;
5         for (char c : s) ++m[c];
6         sort(s.begin(), s.end(), [&](char& a, char& b){
7             return m[a] > m[b] || (m[a] == m[b] && a < b);
8         });
9         return s;
10    }
11 }
12 }
```

CPP

我们也可以不使用优先队列，而是建立一个字符串数组，因为某个字符的出现次数不可能超过s的长度，所以我们将每个字符根据其出现次数放入数组中的对应位置，那么最后我们只要从后往前遍历数组所有位置，将不为空的位置的字符串加入结果res中即可，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     string frequencySort(string s) {
4         string res = "";
5         vector<string> v(s.size() + 1, "");
6         unordered_map<char, int> m;
7         for (char c : s) ++m[c];
8         for (auto& a : m) {
9             v[a.second].append(a.second, a.first);
10        }
11        for (int i = s.size(); i > 0; --i) {
12            if (!v[i].empty()) {
13                res.append(v[i]);
14            }
15        }
16        return res;
17    }
18 };

```

CPP

452. 最少数量的箭引爆气球

There are a number of spherical balloons spread in two-dimensional space. For each balloon, provided input is the start and end coordinates of the horizontal diameter. Since it's horizontal, y-coordinates don't matter and hence the x-coordinates of start and end of the diameter suffice. Start is always smaller than end. There will be at most 104 balloons.

An arrow can be shot up exactly vertically from different points along the x-axis. A balloon with x_{start} and x_{end} bursts by an arrow shot at x if $x_{start} \leq x \leq x_{end}$. There is no limit to the number of arrows that can be shot. An arrow once shot keeps travelling up infinitely. The problem is to find the minimum number of arrows that must be shot to burst all balloons.

这道题给了我们一堆大小不等的气球，用区间范围来表示气球的大小，可能会有重叠区间。然后我们用最少的箭数来将所有的气球打爆。那么这道题是典型的用贪婪算法来做的题，因为局部最优解就等于全局最优解，我们首先给区间排序，我们不用特意去写排序比较函数，因为默认的对于pair的排序，就是按第一个数字升序排列，如果第一个数字相同，那么按第二个数字升序排列，这个就是我们需要的顺序，所以直接用即可。然后我们将res初始化为1，因为气球数量不为0，所以怎么也得先来一发啊，然后这一箭能覆盖的最远位置就是第一个气球的结束点，用变量end来表示。然后我们开始遍历剩下的气球，如果当前气球的开始点小于等于end，说明跟之前的气球有重合，之前那一箭也可以照顾到当前的气球，此时我们要更新end的位置，end更新为两个气球结束点之间较小的那个，这也是当前气球和之前气球的重合点，然后继续看后面的气球；如果某个气球的起始点大于end了，说明前面的箭无法覆盖到当前的气球，那么就得再来一发，既然又来了一发，那么我们此时就要把end设为当前气球的结束点了，这样贪婪算法遍历结束后就能得到最少的箭数了，参见代码如下：

```

1 class Solution {
2 public:
3     int findMinArrowShots(vector<pair<int, int>>& points) {
4         if (points.empty()) return 0;
5         sort(points.begin(), points.end());
6         int res = 1, end = points[0].second;
7         for (int i = 1; i < points.size(); ++i) {
8             if (points[i].first <= end) {
9                 end = min(end, points[i].second);
10            } else {
11                ++res;
12                end = points[i].second;
13            }
14        }
15        return res;
16    }
17 };

```

453. 最少移动次数使数组元素相等

Given a non-empty integer array of size n, find the minimum number of moves required to make all array elements equal, where a move is incrementing n - 1 elements by 1.

Example:

Input:

[1,2,3]

Output:

3

这道题给了我们一个长度为n的数组，说是我们每次可以对n-1个数字同时加1，问最少需要多少次这样的操作才能让数组中所有的数字相等。那么我们想，为了快速的缩小差距，该选择哪些数字加1呢，不难看出每次需要给除了数组最大值的所有数字加1，这样能快速的到达平衡状态。但是这道题如果我们老老实实的每次找出最大值，然后给其他数字加1，再判断是否平衡，思路是正确，但是OJ不答应。正确的解法相当的巧妙，需要换一个角度来看问题，其实给n-1个数字加1，效果等同于给那个未被选中的数字减1，比如数组[1, 2, 3]，给除去最大值的其他数字加1，变为[2, 3, 3]，我们全体减1，并不影响数字间相对差异，变为[1, 2, 2]，这个结果其实就是原始数组的最大值3自减1，那么问题也可能转化为，将所有数字都减小到最小值，这样难度就大大降低了，我们只要先找到最小值，然后累加每个数跟最小值之间的差值即可，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     int minMoves(vector<int>& nums) {
4         int mn = INT_MAX, res = 0;
5         for (int num : nums) mn = min(mn, num);
6         for (int num : nums) res += num - mn;
7         return res;
8     }
9 };

```

我们也可以求出数组的数字之和sum，然后用sum减去最小值和数组长度的乘积，也能得到答案：

解法2:

```

1 class Solution {
2 public:
3     int minMoves(vector<int>& nums) {
4         int mn = INT_MAX, sum = 0, res = 0;
5         for (int num : nums) {
6             mn = min(mn, num);
7             sum += num;
8         }
9         return sum - mn * nums.size();
10    }
11 };

```

454. 四数之和之二

Given four lists A, B, C, D of integer values, compute how many tuples (i, j, k, l) there are such that A[i] + B[j] + C[k] + D[l] is zero.

To make problem a bit easier, all A, B, C, D have same length of N where $0 \leq N \leq 500$. All integers are in the range of -228 to 228 - 1 and the result is guaranteed to be at most 231 - 1.

Example:

Input:

```

A = [ 1, 2]
B = [-2,-1]
C = [-1, 2]
D = [ 0, 2]

```

Output:

2

这道题是之前那道4Sum的延伸，让我们在四个数组中各取一个数字，使其和为0。那么笨的方法就是遍历所有的情况，时间复杂度为O(n⁴)。但是我们想想既然Two Sum那道都能将时间复杂度缩小一倍，那么这道题我们使用哈希表是否也能将时间复杂度降到O(n²)呢？答案是肯定的，我们如果把A和B的两两之和都求出来，在哈希表中建立两数之和跟其出现次数之间的映射，那么我们再遍历C和D中任意两个数之和，我们只要看哈希表存不存在这两数之和的相反数就行了，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int fourSumCount(vector<int>& A, vector<int>& B, vector<int>& C, vector<int>& D) {
4         int res = 0;
5         unordered_map<int, int> m;
6         for (int i = 0; i < A.size(); ++i) {
7             for (int j = 0; j < B.size(); ++j) {
8                 ++m[A[i] + B[j]];
9             }
10        }
11        for (int i = 0; i < C.size(); ++i) {
12            for (int j = 0; j < D.size(); ++j) {
13                int target = -1 * (C[i] + D[j]);
14                res += m[target];
15            }
16        }
17        return res;
18    }
19 };

```

这种方法用了两个哈希表分别记录AB和CB的两两之和出现次数，然后遍历其中一个哈希表，并在另一个哈希表中找和的相反数出现的次数，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int fourSumCount(vector<int>& A, vector<int>& B, vector<int>& C, vector<int>& D) {
4         int res = 0, n = A.size();
5         unordered_map<int, int> m1, m2;
6         for (int i = 0; i < n; ++i) {
7             for (int j = 0; j < n; ++j) {
8                 ++m1[A[i] + B[j]];
9                 ++m2[C[i] + D[j]];
10            }
11        }
12        for (auto a : m1) res += a.second * m2[-a.first];
13        return res;
14    }
15 };

```

455. 分点心

Assume you are an awesome parent and want to give your children some cookies. But, you should give each child at most one cookie. Each child i has a greed factor g_i , which is the minimum size of a cookie that the child will be content with; and each cookie j has a size s_j . If $s_j \geq g_i$, we can assign the cookie j to the child i , and the child i will be content. Your goal is to maximize the number of your content children and output the maximum number.

Note:

You may assume the greed factor is always positive.

You cannot assign more than one cookie to one child.

这道题给了我们一堆cookie，每个cookie的大小不同，还有一堆小朋友，每个小朋友的胃口也不同的，问我们当前的cookie最多能满足几个小朋友。这是典型的利用贪婪算法的题目，我们可以首先对两个数组进行排序，让小的在前面。然后我们先拿最小的cookie给胃口最小的小朋友，看能否满足，能的话，我们结果res自加1，然后再拿下一个cookie去满足下一位小朋友；如果当前cookie不能满足当前小朋友，那么我们就用下一块稍大一点的cookie去尝试满足当前的小朋友。当cookie发完了或者小朋友没有了我们停止遍历，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int findContentChildren(vector<int>& g, vector<int>& s) {
4         int res = 0, p = 0;
5         sort(g.begin(), g.end());
6         sort(s.begin(), s.end());
7         for (int i = 0; i < s.size(); ++i) {
8             if (s[i] >= g[p]) {
9                 ++res;
10                ++p;
11                if (p >= g.size()) break;
12            }
13        }
14        return res;
15    }
16 };

```

CPP

我们可以对上述代码进行精简，我们用变量j既可以表示小朋友数组的坐标，同时又可以表示已满足的小朋友的个数，因为只有满足了当前的小朋友，才会去满足下一个胃口较大的小朋友，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int findContentChildren(vector<int>& g, vector<int>& s) {
4         int j = 0;
5         sort(g.begin(), g.end());
6         sort(s.begin(), s.end());
7         for (int i = 0; i < s.size() && j < g.size(); ++i) {
8             if (s[i] >= g[j]) ++j;
9         }
10        return j;
11    }
12 };

```

CPP

456. 132模式

Given a sequence of n integers a_1, a_2, \dots, a_n , a 132 pattern is a subsequence a_i, a_j, a_k such that $i < j < k$ and $a_i < a_k < a_j$. Design an algorithm that takes a list of n numbers as input and checks whether there is a 132 pattern in the list.

Note: n will be less than 15,000.

Example 1:

Input: [1, 2, 3, 4]

Output: False

这道题给我们了一个数组，让我们找到132的模式，就是第一个数小于第二第三个数，且第三个数小于第二个数。那么我们就按顺序来找这三个数，首先我们来找第一个数，这个数需要最小，那么我们如果发现当前数字大于等于后面一个数字，我们就往下继续遍历，直到当前数字小于下一个数字停止。然后我们找第二个数字，这个数字需要最大，那么如果我们发现当前数字小于等于下一个数字就继续遍历，直到当前数字大于下一个数字停止。最后就找第三个数字，我们验证这个数字是否在之前两个数字的中间，如果没有找到，我们就从第二个数字的后面一个位置继续开始重新找这三个数字，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool find132pattern(vector<int>& nums) {
4         if (nums.size() <= 2) return false;
5         int n = nums.size(), i = 0, j = 0, k = 0;
6         while (i < n) {
7             while (i < n - 1 && nums[i] >= nums[i + 1]) ++i;
8             j = i + 1;
9             while (j < n - 1 && nums[j] <= nums[j + 1]) ++j;
10            k = j + 1;
11            while (k < n) {
12                if (nums[k] > nums[i] && nums[k] < nums[j]) return true;
13                ++k;
14            }
15            i = j + 1;
16        }
17        return false;
18    }
19 }
```

CPP

下面这种方法利用来栈来做，既简洁又高效，思路是我们维护一个栈和一个变量third，其中third就是第三个数字，也是pattern 132中的2，栈里面按顺序放所有大于third的数字，也是pattern 132中的3，那么我们在遍历的时候，如果当前数字小于third，即pattern 132中的1找到了，我们直接返回true即可，因为已经找到了，注意我们应该从后往前遍历数组。如果当前数字大于栈顶元素，那么我们按顺序将栈顶数字取出，赋值给third，然后将该数字压入栈，这样保证了栈里的元素仍然都是大于third的，我们想要的顺序依旧存在，进一步来说，栈里存放的都是可以维持 $second > third$ 的 $second$ 值，其中的任何一个值都是大于当前的third值，如果有更大的值进来，那就等于形成了一个更优的 $second > third$ 的这样一个组合，并且这时弹出的third值比以前的third值更大，为什么要保证third值更大，因为这样才可以更容易的满足当前的值first比third值小这个条件，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool find132pattern(vector<int>& nums) {
4         int third = INT_MIN;
5         stack<int> s;
6         for (int i = nums.size() - 1; i >= 0; --i) {
7             if (nums[i] < third) return true;
8             else while (!s.empty() && nums[i] > s.top()) {
9                 third = s.top(); s.pop();
10            }
11            s.push(nums[i]);
12        }
13        return false;
14    }
15 };

```

457. 环形数组循环

You are given an array of positive and negative integers. If a number n at an index is positive, then move forward n steps. Conversely, if it's negative ($-n$), move backward n steps. Assume the first element of the array is forward next to the last element, and the last element is backward next to the first element. Determine if there is a loop in this array. A loop starts and ends at a particular index with more than 1 element along the loop. The loop must be "forward" or "backward".

Example 1: Given the array [2, -1, 1, 2, 2], there is a loop, from index 0 \rightarrow 2 \rightarrow 3 \rightarrow 0.

Example 2: Given the array [-1, 2], there is no loop.

Note: The given array is guaranteed to contain no element "0".

Can you do it in $O(n)$ time complexity and $O(1)$ space complexity?

说实话，这道题描述的并不是很清楚，比如题目中有句话说循环必须是forward或是backward的，如果不给例子说明，不太容易能get到point。所谓的循环必须是一个方向的就是说不能跳到一个数，再反方向跳回来，这不算一个loop。比如[1, -1]就不是一个loop，而[1, 1]是一个正确的loop。看到论坛中一半的帖子都是各种需要clarify和不理解test case就感觉很好笑~博主也成功踩坑了。弄清楚了题意后来考虑如何做，由于从一个位置只能跳到一个别的位置，而不是像图那样一个点可以到多个位置，所以这里我们就可以根据坐标建立一对一的映射，一旦某个达到的坐标已经有映射了，说明环存在，当然我们还需要进行一系列条件判断。首先我们需要一个visited数组，来记录访问过的数字，然后我们遍历原数组，如果当前数字已经访问过了，直接跳过，否则就以当前位置坐标为起始点开始查找，进行while循环，将当前位置在visited数组中标记true，然后计算下一个位置，计算方法是当前位置坐标加上对应的数字，由于是循环数组，所以结果可能会超出数组的长度，所以我们要对数组长度取余。当然上面的数字也可能是负数，加完以后可能也是负数，所以光取余还不够，还得再补上一个 n ，使其变为正数。此时我们判断，如果next和cur相等，说明此时是一个数字的循环，不符合题意，再有就是检查二者的方向，数字是正数表示forward，若是负数表示backward，在一个loop中必须同正或同负，我们只要让二者相乘，如果结果是负数的话，说明方向不同，直接break掉。此时如果next已经有映射了，说明我们找到了合法的loop，返回true，否则建立一个这样的映射，继续循环，参见代码如下：

```

1 class Solution {
2 public:
3     bool circularArrayLoop(vector<int>& nums) {
4         unordered_map<int, int> m;
5         int n = nums.size();
6         vector<bool> visited(n, false);
7         for (int i = 0; i < n; ++i) {
8             if (visited[i]) continue;
9             int cur = i;
10            while (true) {
11                visited[cur] = true;
12                int next = (cur + nums[cur]) % n;
13                if (next < 0) next += n;
14                if (next == cur || nums[next] * nums[cur] < 0) break;
15                if (m.count(next)) return true;
16                m[cur] = next;
17                cur = next;
18            }
19        }
20        return false;
21    }
22 };

```

458. 可怜的猪

There are 1000 buckets, one and only one of them contains poison, the rest are filled with water. They all look the same. If a pig drinks that poison it will die within 15 minutes. What is the minimum amount of pigs you need to figure out which bucket contains the poison within one hour.

Answer this question, and write an algorithm for the follow-up general case.

Follow-up:

If there are n buckets and a pig drinking poison will die within m minutes, how many pigs (x) you need to figure out the "poison" bucket within p minutes? There is exact one bucket with poison.

这道题博主拿到以后并木有什么头绪，可是明明标的是Easy，打击甚大，于是去论坛上大神们的解法，感觉这道题应该算是一道Brain Teaser的题，对问题的分析能力要求很高。那么我们来一步一步从最简单的情况来分析吧，假设只有1只猪，只有15分钟，那么我们能测几个水桶呢？很显然是两个，因为只能测一次的话，让猪去随便喝一桶，如果毒死了，就是喝的那桶，反之则是另一桶。好，那么如果有两只猪呢，能测几桶？怎么喝呢，两只猪一猪喝一桶，再同时喝一桶，剩下一桶谁也不喝，那么如果两只猪都毒死了，说明是共同喝的那桶有毒，如果某个猪毒死了，说明该猪喝的那桶有毒，如果都没事，说明是谁也没喝的那桶。那么我们应该看出规律了吧，没错，三猪能测8桶，其实就是2的指数倍。

如果只能测一次的话，实际上相当一个一维数组，而如果能测两次的话，情况就不一样了，我们就可以重复利用猪了。比如还是两只猪，能测两次，功能测几个桶，答案可以测9桶，为啥，我们组个二维数组：

1 2 3

4 5 6

7 8 9

如果我们让第一头猪第一次喝1, 2, 3桶，第二次喝4, 5, 6桶，而让第二头猪第一次喝1, 4, 7桶，第二次喝2, 5, 8桶，我们可以根据猪的死亡情况来确定是哪一桶的问题，实际上就把猪被毒死的那个节点当作了二维数组的横纵坐标来定位毒桶的位置，巧妙吧~更巧妙的是，如果再增加一头猪，实际上是给数组增加一个维度，变成了一个三维数组，那么三只猪，测两次，可以测27桶，叼不叼。这道题让我们求最少用多少猪来测，那么就是求数组的维度，我们知道了数组的总个数，所以要尽量增加数组的长宽，尽量减少维度。这里，数组的长宽其实都是测试的次数+1，所以我们首先要确定能测的次数，通过总测试时间除以毒发时间，再加上1就是测试次数。有了数组长宽m，那么如果有x只猪，能测的桶数为m的x次方，现在我们给定了桶数N，要求x，就log一下就行，然后用个换底公式，就可以求出x的值了，参见代码如下：

```
1 class Solution {
2     public:
3         int poorPigs(int buckets, int minutesToDie, int minutesToTest) {
4             return ceil(log(buckets) / log(minutesToTest / minutesToDie + 1));
5         }
6     };

```

CPP

459. 重复子字符串模式

Given a non-empty string check if it can be constructed by taking a substring of it and appending multiple copies of the substring together. You may assume the given string consists of lowercase English letters only and its length will not exceed 10000.

Example 1:

Input: "abab"

Output: True

这道题给了我们一个字符串，问其是否能拆成n个重复的子串。那么既然能拆分成多个子串，那么每个子串的长度肯定不能大于原字符串长度的一半，那么我们可以从原字符串长度的一半遍历到1，如果当前长度能被总长度整除，说明可以分成若干个子字符串，我们将这些子字符串拼接起来看跟原字符串是否相等。如果拆完了都不相等，返回false。

解法1:

```

1 class Solution {
2 public:
3     bool repeatedSubstringPattern(string str) {
4         int n = str.size();
5         for (int i = n / 2; i >= 1; --i) {
6             if (n % i == 0) {
7                 int c = n / i;
8                 string t = "";
9                 for (int j = 0; j < c; ++j) {
10                     t += str.substr(0, i);
11                 }
12                 if (t == str) return true;
13             }
14         }
15         return false;
16     }
17 };

```

下面这种方法是参考的网上的这个帖子，原作者说是用的KMP算法，LeetCode之前也有一道应用KMP算法来解的题Shortest Palindrome，但是感觉那道题才是KMP算法。这道题也称为KMP算法感觉怪怪的(关于KMP的详细介绍请参见从头到尾彻底理解KMP，也可以看博主自己写的一篇KMP Algorithm 字符串匹配算法KMP小结)，KMP算法中的next数组是找当前位置的最大相同前缀后缀的个数，而这道题维护的一位数组dp[i]表示，到位置i-1为止的重复字符串的字符个数，不包括被重复的那个字符串，什么意思呢，我们举个例子，比如"abcabc"的dp数组为[0 0 0 1 2 3]，dp数组长度要比原字符串长度多一个。那么我们看最后一个位置数字为3，就表示重复的字符串的字符数有3个。如果是"abcababcabc"，那么dp数组为[0 0 0 0 1 2 3 4 5 6]，我们发现最后一个数字为6，那么表示重复的字符串为"abcabc"，有6个字符。那么怎么通过最后一个数字来知道原字符串是否由重复的子字符串组成的呢，首先当然是最后一个数字不能为0，而且还要满足 $dp[n] \% (n - dp[n]) == 0$ 才行，因为 $n - dp[n]$ 是一个子字符串的长度，那么重复字符串的长度和肯定是一个子字符串的整数倍，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool repeatedSubstringPattern(string str) {
4         int i = 1, j = 0, n = str.size();
5         vector<int> dp(n + 1, 0);
6         while (i < n) {
7             if (str[i] == str[j]) dp[++i] = ++j;
8             else if (j == 0) ++i;
9             else j = dp[j];
10        }
11        return dp[n] && (dp[n] % (n - dp[n]) == 0);
12    }
13 };

```

460. 最近最不常用页面置换缓存器

Design and implement a data structure for Least Frequently Used (LFU) cache. It should support the following operations: get and put.

get(key) - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.

put(key, value) - Set or insert the value if the key is not already present. When the cache reaches its capacity, it should invalidate the least frequently used item before inserting a new item. For the purpose of this problem, when there is a tie (i.e., two or more keys that have the same frequency), the least recently used key would be evicted.

Follow up:

Could you do both operations in O(1) time complexity?

这道题是让我们实现最近不常用页面置换算法LFU (Least Frequently Used)，之前我们做过一道类似的题LRU Cache，让我们求最近最少使用页面置换算法LRU (Least Recently Used)。两种算法虽然名字看起来很相似，但是其实是不同的。顾名思义，LRU 算法是首先淘汰最长时间未被使用的页面，而LFU是先淘汰一定时间内被访问次数最少的页面。光说无凭，举个例子来看看，比如说我们的cache的大小为3，然后我们按顺序存入 5, 4, 5, 4, 5, 7，这时候cache刚好被装满了，因为put进去之前存在的数不会占用额外地方。那么此时我们想再put进去一个8，如果使用LRU算法，应该将4删除，因为4最久未被使用，而如果使用LFU 算法，则应该删除7，因为7被使用的次数最少，只使用了一次。相信这个简单的例子可以大概说明二者的区别。

这道题比之前那道LRU的题目还要麻烦一些，因为那道题只要用个list把数字按时间顺序存入，链表底部的位置总是最久未被使用的，每次删除底部的值即可。而这道题不一样，由于需要删除最少次数的数字，那么我们必须要统计每一个key出现的次数，所以我们用一个哈希表m来记录当前数据{key, value}和其出现次数之间的映射，这样还不够，为了方便操作，我们需要把相同频率的key都放到一个list中，那么需要另一个哈希表freq来建立频率和一个里面所有key都是当前频率的list之间的映射。由于题目中要我们在O(1)的时间内完成操作了，为了快速的定位freq中key的位置，我们再用一个哈希表iter来建立key和freq中key的位置之间的映射。最后当然我们还需要两个变量cap和minFreq，分别来保存cache的大小，和当前最小的频率。

为了更好的讲解思路，我们还是用例子来说明吧，我们假设cache的大小为2，假设我们已经按顺序put进去5, 4，那么来看一下内部的数据是怎么保存的，由于value的值并不是很重要，为了不影响key和frequency，我们采用value#来标记：

m:

5 -> {value5, 1}

4 -> {value4, 1}

freq:

1 -> {5, 4}

iter:

4 -> list.begin() + 1

5 -> list.begin()

这应该不是很难理解，m中5对应的频率为1，4对应的频率为1，然后freq中频率为1的有4和5。iter中是key所在freq中对应链表中的位置的iterator。然后我们的下一步操作是get(5)，下面是get需要做的步骤：

1. 如果m中不存在5，那么返回-1
2. 从freq中频率为1的list中将5删除
3. 将m中5对应的frequency值自增1
4. 将5保存到freq中频率为2的list的末尾
5. 在iter中保存5在freq中频率为2的list中的位置
6. 如果freq中频率为minFreq的list为空，minFreq自增1
7. 返回m中5对应的value值

经过这些步骤后，我们再来看下此时内部数据的值：

m:

5 -> {value5, 2}

4 -> {value4, 1}

freq:

1 -> {4}

2 -> {5}

iter:

4 -> list.begin()

5 -> list.begin()

这应该不是很难理解，m中5对应的频率为2，4对应的频率为1，然后freq中频率为1的只有4，频率为2的只有5。iter中是key所在freq中对应链表中的位置的iterator。然后我们下一步操作是要put进去一个7，下面是put需要做的步骤：

1. 如果调用get(7)返回的结果不是-1，那么在将m中7对应的value更新为当前value，并返回
2. 如果此时m的大小大于了cap，即超过了cache的容量，则：
 - a) 在m中移除minFreq对应的list的首元素的纪录，即移除4 -> {value4, 1}
 - b) 在iter中清除4对应的纪录，即移除4 -> list.begin()
 - c) 在freq中移除minFreq对应的list的首元素，即移除4
3. 在m中建立7的映射，即 7 -> {value7, 1}
4. 在freq中频率为1的list末尾加上7
5. 在iter中保存7在freq中频率为1的list中的位置
6. minFreq重置为1

经过这些步骤后，我们再来看下此时内部数据的值：

m:

```
5 -> {value5, 2}  
7 -> {value7, 1}
```

freq:

```
1 -> {7}  
2 -> {5}
```

iter:

```
7 -> list.begin()  
5 -> list.begin()
```

参见代码如下：

```

1 class LFUCache {
2 public:
3     LFUCache(int capacity) {
4         cap = capacity;
5     }
6
7     int get(int key) {
8         if (m.count(key) == 0) return -1;
9         freq[m[key].second].erase(iter[key]);
10        ++m[key].second;
11        freq[m[key].second].push_back(key);
12        iter[key] = --freq[m[key].second].end();
13        if (freq[minFreq].size() == 0) ++minFreq;
14        return m[key].first;
15    }
16
17    void put(int key, int value) {
18        if (cap <= 0) return;
19        if (get(key) != -1) {
20            m[key].first = value;
21            return;
22        }
23        if (m.size() >= cap) {
24            m.erase(freq[minFreq].front());
25            iter.erase(freq[minFreq].front());
26            freq[minFreq].pop_front();
27        }
28        m[key] = {value, 1};
29        freq[1].push_back(key);
30        iter[key] = --freq[1].end();
31        minFreq = 1;
32    }
33
34 private:
35     int cap, minFreq;
36     unordered_map<int, pair<int, int>> m;
37     unordered_map<int, list<int>> freq;
38     unordered_map<int, list<int>::iterator> iter;
39 };

```

461. 汉明距离

The Hamming distance between two integers is the number of positions at which the corresponding bits are different.

Given two integers x and y , calculate the Hamming distance.

Note:

$0 \leq x, y < 2^{31}$.

这道题让我求两个数字之间的汉明距离，题目中解释的很清楚了，两个数字之间的汉明距离就是其二进制数对应位不同的个数，那么最直接了当的做法就是按位分别取出两个数对应位上的数并异或，我们知道异或的性质上相同的为0，不同的为1，我们只要把为1的情况累加起来就是汉明距离了，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int hammingDistance(int x, int y) {
4         int res = 0;
5         for (int i = 0; i < 32; ++i) {
6             if ((x & (1 << i)) ^ (y & (1 << i))) {
7                 ++res;
8             }
9         }
10        return res;
11    }
12 };

```

我们可以对上面的代码进行优化，我们可以一开始直接将两个数字异或起来，然后我们遍历异或结果的每一位，统计为1的个数，也能达到同样的效果，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int hammingDistance(int x, int y) {
4         int res = 0, exc = x ^ y;
5         for (int i = 0; i < 32; ++i) {
6             res += (exc >> i) & 1;
7         }
8         return res;
9     }
10 };

```

经过副博主@fantasywindy的提醒，上面的遍历每一位的方法并不高效，还可以进一步优化，假如数为num, num & (num - 1)可以快速地移除最右边的bit 1，一直循环到num为0，总的循环数就是num中bit 1的个数。参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int hammingDistance(int x, int y) {
4         int res = 0, exc = x ^ y;
5         while (exc) {
6             ++res;
7             exc &= (exc - 1);
8         }
9         return res;
10    }
11 };

```

我们再来看一种递归的写法，非常的简洁，递归终止的条件是当两个数异或为0时，表明此时两个数完全相同，我们返回0，否则我们返回异或和对2取余加上对x/2和y/2调用递归的结果。异或和对2取余相当于检查最低位是否相同，而对x/2和y/2调用递归相当于将x和y分别向右移动一位，这样每一位都可以比较到，也能得到正确结果，参见代码如下：

解法4：

```

1 class Solution {
2 public:
3     int hammingDistance(int x, int y) {
4         if ((x ^ y) == 0) return 0;
5         return (x ^ y) % 2 + hammingDistance(x / 2, y / 2);
6     }
7 };

```

462. 最少移动次数使数组元素相等之二

Given a non-empty integer array, find the minimum number of moves required to make all array elements equal, where a move is incrementing a selected element by 1 or decrementing a selected element by 1.

You may assume the array's length is at most 10,000

这道题是之前那道Minimum Moves to Equal Array Elements的拓展，现在我们可以每次对任意一个数字加1或者减1，让我们用最少的次数让数组所有值相等。一般来说这种题目是不能用暴力方法算出所有情况，因为OJ一般是不会答应的。那么这道题是否像上面一道题一样，有巧妙的方法呢？答案是肯定的。下面这种解法实际上利用了之前一道题Best Meeting Point的思想，是不感觉很amazing，看似完全不相干的两道题，居然有着某种内部联系。我们首先给数组排序，那么我们最终需要变成的相等的数字就是中间的数，如果数组有奇数个，那么就是最中间的那个数字；如果是偶数个，那么就是中间两个数的区间中的任意一个数字。而两端的数字变成中间的一个数字需要的步数实际上就是两端数字的距离，讲到这里发现是不是就和这道题Best Meeting Point的思路是一样了。那么我们就两对两对的累加它们的差值就可以了，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int minMoves2(vector<int>& nums) {
4         int res = 0, i = 0, j = (int)nums.size() - 1;
5         sort(nums.begin(), nums.end());
6         while (i < j) {
7             res += nums[j--] - nums[i++];
8         }
9         return res;
10    }
11 };

```

既然有了上面的分析，我们知道实际上最后相等的数字就是数组的最中间的那个数字，那么我们在给数组排序后，直接利用坐标定位到中间的数字，然后算数组中每个数组与其的差的绝对值累加即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int minMoves2(vector<int>& nums) {
4         sort(nums.begin(), nums.end());
5         int res = 0, mid = nums[nums.size() / 2];
6         for (int num : nums) {
7             res += abs(num - mid);
8         }
9         return res;
10    }
11 };

```

上面的两种方法都给整个数组排序了，时间复杂度是O(nlgn)，其实我们并不需要给所有的数字排序，我们只关系最中间的数字，那么这个stl中自带的函数nth_element就可以完美的发挥其作用了，我们只要给出我们想要数字的位置，它就能在O(n)的时间内返回正确的数字，然后算数组中每个数组与其的差的绝对值累加即可，参见代码如下：

解法3：

```
1 class Solution {
2 public:
3     int minMoves2(vector<int>& nums) {
4         int res = 0, n = nums.size(), mid = n / 2;
5         nth_element(nums.begin(), nums.begin() + mid, nums.end());
6         for (int i = 0; i < n; ++i) {
7             res += abs(nums[i] - nums[mid]);
8         }
9         return res;
10    }
11};
```

CPP

下面这种方法是改进版的暴力破解法，它遍历了所有的数字，让每个数字都当作最后相等的值，然后算法出来总步数，每次和res比较，留下较小的。而这种方法叨就叨在它在O(1)的时间内完成了步数统计，那么这样整个遍历下来也只是O(n)的时间，不过由于还是要给数组排序，所以整体的时间复杂度是O(nlgn)，这已经能保证可以通过OJ啦。那么我们来看看如何快速计算总步数，首先我们给数组排序，我们假设中间某个位置有个数字k，那么此时数组就是：nums[0], nums[1], ..., k, ..., nums[n - 1]，如果i为数字k在数组中的坐标，那么有 $k = \text{nums}[i]$ ，那么总步数为：

$$\begin{aligned} Y &= k - \text{nums}[0] + k - \text{nums}[1] + \dots + k - \text{nums}[i - 1] + \text{nums}[i] - k + \text{nums}[i + 1] - k + \dots + \\ &\quad \text{nums}[n - 1] - k \\ &= i * k - (\text{nums}[0] + \text{nums}[1] + \dots + \text{nums}[i - 1]) + (\text{nums}[i] + \text{nums}[i + 1] + \dots + \text{nums}[n - 1]) - (n - i) * k \\ &= 2 * i * k - n * k + \text{sum} - 2 * \text{curSum} \end{aligned}$$

那么我们只要算出sum和curSum就可以快速得到总步数了，数组之和可以通过遍历数组计算出来，curSum可以在遍历的过程中累加，那么我们就可以算出总步数，然后每次更新结果res了，参见代码如下：

解法4：

```
1 class Solution {
2 public:
3     int minMoves2(vector<int>& nums) {
4         sort(nums.begin(), nums.end());
5         long long sum = accumulate(nums.begin(), nums.end(), 0);
6         long long res = LONG_MAX, curSum = 0;
7         int n = nums.size();
8         for (int i = 0; i < n; ++i) {
9             long long k = nums[i];
10            curSum += k;
11            res = min(res, 2 * k * (i + 1) - n * k + sum - 2 * curSum);
12        }
13        return res;
14    }
15};
```

CPP

463. 岛屿周长

You are given a map in form of a two-dimensional integer grid where 1 represents land and 0 represents water. Grid cells are connected horizontally/vertically (not diagonally). The grid is completely surrounded by water, and there is exactly one island (i.e., one or more connected land cells). The island doesn't have "lakes" (water inside that isn't connected to the water around the island). One cell is a square with side length 1. The grid is rectangular, width and height don't exceed 100. Determine the perimeter of the island.

这道题给了我们一个格子图，若干连在一起的格子形成了一个小岛，规定了图中只有一个相连的岛，且岛中没有湖，让我们求岛的周长。我们知道一个格子有四条边，但是当两个格子相邻，周围为6，若某个格子四周都有格子，那么这个格子一条边都不算在周长里。那么我们怎么统计出岛的周长呢？第一种方法，我们对于每个格子的四条边分别来处理，首先看左边的边，只有当左边的边处于第一个位置或者当前格子的左面没有岛格子的时候，左边的边计入周长。其他三条边的分析情况都跟左边的边相似，这里就不多叙述了，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     int islandPerimeter(vector<vector<int>>& grid) {
4         if (grid.empty() || grid[0].empty()) return 0;
5         int m = grid.size(), n = grid[0].size(), res = 0;
6         for (int i = 0; i < m; ++i) {
7             for (int j = 0; j < n; ++j) {
8                 if (grid[i][j] == 0) continue;
9                 if (j == 0 || grid[i][j - 1] == 0) ++res;
10                if (i == 0 || grid[i - 1][j] == 0) ++res;
11                if (j == n - 1 || grid[i][j + 1] == 0) ++res;
12                if (i == m - 1 || grid[i + 1][j] == 0) ++res;
13            }
14        }
15        return res;
16    }
17};
```

CPP

下面这种方法对于每个岛屿格子先默认加上四条边，然后检查其左面和上面是否有岛屿格子，有的话分别减去两条边，这样也能得到正确的结果，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     int islandPerimeter(vector<vector<int>>& grid) {
4         if (grid.empty() || grid[0].empty()) return 0;
5         int res = 0, m = grid.size(), n = grid[0].size();
6         for (int i = 0; i < m; ++i) {
7             for (int j = 0; j < n; ++j) {
8                 if (grid[i][j] == 0) continue;
9                 res += 4;
10                if (i > 0 && grid[i - 1][j] == 1) res -= 2;
11                if (j > 0 && grid[i][j - 1] == 1) res -= 2;
12            }
13        }
14        return res;
15    }
16};
```

CPP

464. 我能赢吗

In the "100 game," two players take turns adding, to a running total, any integer from 1..10. The player who first causes the running total to reach or exceed 100 wins.

What if we change the game so that players cannot re-use integers?

这道题给了我们一堆数字，然后两个人，每人每次选一个数字，看数字总数谁先到给定值，有点像之前那道Nim Game，但是比那题难度大。我刚开始想肯定说用递归啊，结果写完发现TLE了，后来发现我们必须要优化效率，使用HashMap来记录已经计算过的结果。我们首先来看如果给定的数字范围大于等于目标值的话，直接返回true。如果给定的数字总和小于目标值的话，说明谁也没法赢，返回false。然后我们进入递归函数，首先我们查找当前情况是否在哈希表中存在，有的话直接返回即可。我们使用一个整型数按位来记录数组中的某个数字是否使用过，我们遍历所有数字，将该数字对应的mask算出来，如果其和used相与为0的话，说明该数字没有使用过，我们看如果此时的目标值小于等于当前数字，说明已经赢了，或者我们调用递归函数，如果返回false，说明也是第一个人赢了。为啥呢，因为当我们已经选过数字了，此时就该对第二个人调用递归函数，只有他的结果是false，我们才能赢，所以此时我们标记true，返回true。如果遍历完所有数字，我们标记false，返回false，参见代码如下：

```

1 class Solution {
2 public:
3     bool canIWin(int maxChoosableInteger, int desiredTotal) {
4         if (maxChoosableInteger >= desiredTotal) return true;
5         if (maxChoosableInteger * (maxChoosableInteger + 1) / 2 < desiredTotal) return
6             false;
7         unordered_map<int, bool> m;
8         return canWin(maxChoosableInteger, desiredTotal, 0, m);
9     }
10    bool canWin(int length, int total, int used, unordered_map<int, bool>& m) {
11        if (m.count(used)) return m[used];
12        for (int i = 0; i < length; ++i) {
13            int cur = (1 << i);
14            if ((cur & used) == 0) {
15                if (total <= i + 1 || !canWin(length, total - (i + 1), cur | used, m)) {
16                    m[used] = true;
17                    return true;
18                }
19            }
20        }
21        m[used] = false;
22        return false;
23    }
24 };

```

465. 最优账户平衡

A group of friends went on holiday and sometimes lent each other money. For example, Alice paid for Bill's lunch for 10. Then later Chris gave Alice 5 for a taxi ride. We can model each transaction as a tuple (x, y, z) which means person x gave person y $\$z$. Assuming Alice, Bill, and Chris are person 0, 1, and 2 respectively (0, 1, 2 are the person's ID), the transactions can be represented as $[[0, 1, 10], [2, 0, 5]]$.

Given a list of transactions between a group of people, return the minimum number of transactions required to settle the debt.

Note:

A transaction will be given as a tuple (x, y, z) . Note that $x \neq y$ and $z > 0$. Person's IDs may not be linear, e.g. we could have the persons 0, 1, 2 or we could also have the persons 0, 2, 6.

这道题给了一堆某人欠某人多少钱这样的账单，问我们经过优化后最少还剩几个。其实就相当于一堆人出去玩，某些人可能帮另一些人垫付过花费，最后结算总花费的时候可能你欠着别人的钱，其他人可能也欠你的欠。我们需要找出简单的方法把所有欠账都还清就行了。这道题的思路跟之前那道Evaluate Division有些像，都需要对一组数据颠倒顺序处理。我们使用一个哈希表来建立每个人和其账户的映射，其中账户若为正数，说明其他人欠你钱；如果账户为负数，说明你欠别人钱。我们对于每份账单，前面的人就在哈希表中减去钱数，后面的人在哈希表中加上钱数。这样我们每个人就都有一个账户了，然后我们接下来要做的就是合并账户，看最少需要多少次汇款。我们先统计出账户值不为0的人数，因为如果为0了，表明你既不欠别人钱，别人也不欠你钱，如果不为0，我们把钱数放入一个数组accnt中，然后调用递归函数。在递归函数中，我们初始化结果res为整型最大值，然后我们跳过为0的账户，然后我们开始遍历之后的账户，如果当前账户和之前账户的钱数正负不同的话，我们将前一个账户的钱数加到当前账户上，这很好理解，比如前一个账户钱数是-5，表示张三欠了别人五块钱，当前账户钱数是5，表示某人欠了李四五块钱，那么张三给李四五块，这两人的账户就都清零了。然后我们调用递归函数，此时从当前改变过的账户开始找，num表示当前的转账数，需要加1，然后我们用这个递归函数返回的结果来更新res，后面别忘了复原当前账户的值。遍历结束后，我们看res的值如果还是整型的最大值，说明没有改变过，我们返回num，否则返回res即可，参见代码如下：

```

1 class Solution {
2 public:
3     int minTransfers(vector<vector<int>>& transactions) {
4         unordered_map<int, int> m;
5         for (auto t : transactions) {
6             m[t[0]] -= t[2];
7             m[t[1]] += t[2];
8         }
9         vector<int> accnt(m.size());
10        int cnt = 0;
11        for (auto a : m) {
12            if (a.second != 0) accnt[cnt++] = a.second;
13        }
14        return helper(accnt, 0, cnt, 0);
15    }
16    int helper(vector<int>& accnt, int start, int n, int num) {
17        int res = INT_MAX;
18        while (start < n && accnt[start] == 0) ++start;
19        for (int i = start + 1; i < n; ++i) {
20            if ((accnt[i] < 0 && accnt[start] > 0) || (accnt[i] > 0 && accnt[start] < 0)) {
21                accnt[i] += accnt[start];
22                res = min(res, helper(accnt, start + 1, n, num + 1));
23                accnt[i] -= accnt[start];
24            }
25        }
26        return res == INT_MAX ? num : res;
27    }
28 };

```

466. 计数重复个数

Define $S = [s, n]$ as the string S which consists of n connected strings s . For example, $["abc", 3] = "abcabcabc"$.

On the other hand, we define that string s_1 can be obtained from string s_2 if we can remove some characters from s_2 such that it becomes s_1 . For example, “abc” can be obtained from “abdbec” based on our definition, but it can not be obtained from “acbbe”.

You are given two non-empty strings s_1 and s_2 (each at most 100 characters long) and two integers $0 \leq n_1 \leq 10^6$ and $1 \leq n_2 \leq 10^6$. Now consider the strings S_1 and S_2 , where $S_1=[s_1, n_1]$ and $S_2=[s_2, n_2]$. Find the maximum integer M such that $[S_2, M]$ can be obtained from S_1 .

这道题放了好久才写，主要是因为这道题难度确实不小，光是分析研究网上大神们的帖子就搞了好久，就是现在也不能说完全理解了这道题，哎，将就着写吧，有不足的地方欢迎指正。主要参考了网上大神1z1124631x的帖子，还有大神aaaeee0的帖子。这道题的Java版本的brute force可以通过OJ，但是C++的就不行了，我们需要使用重复模式来优化我们的方法，我们知道：

如果s2在S1中出现了N次，那么S2肯定在S1中出现了N/n2次，注意这里的大写表示字符串加上重复次数组成的大字符串。

所以我们得出结论，我们只要算出s2出现的次数，然后除以n2，就可以得出S2出现的次数了。

那么问题就是我们表示重复，我们遍历s1字符串n1次，表示每个s1字符串为一段，对于每段，我们有：

1. 出现在该段的s2字符串的累计出现次数

2. 一个nextIndex，其中s2[nextIndex]表示在下一段s1中你所要寻找的s2中的一个字符。(比如说s1="abc"，s2="bac"，由于第一个s1中只能匹配上s2中的b，那么只有在下一段s1中才能继续匹配s2中的a，所以nextIndex=1，即a在s2中的位置为1；同理，比如s1="abca"，s2="bac"，第一个s1可以匹配上s2中的ba，那么后面的c只能在下一段s1中匹配上，那么nextIndex=2，即c在s2中的位置为2)

表示重复关键就在于nextIndex，比如对于下面这个例子：

Input:

```
s1="abacb", n1=6
s2="bcaa", n2=1
```

Return:

3

j ----->	1	2	3	0	1	2	3	0	
S1 ----->	abacb		abacb		abacb		abacb		abacb

repeatCount ----->	0		1		1		2		2		3
--------------------	---	--	---	--	---	--	---	--	---	--	---

nextIndex ----->	2		1		2		1		2		1
------------------	---	--	---	--	---	--	---	--	---	--	---

nextIndex的范围从0到s2.size()-1，根据鸽巢原理(又称抽屉原理)，你一定会找到相同的两个nextIndex在遍历s1段s2.size()+1次之后。在上面的例子中，重复的nextIndex出现在第三段，和第一段一样都为2，那么重复的pattern就找到了，是第二段和第三段中的aabc，而且从第四段开始，每两段就有一个aabc，现在我们的目标就是统计出整个S1中有多少个s2。

由于pattern占用了两段，所以interval为2，我们然后看整个S1中有多少个这样的两段，repeat = (n1 - start) / interval。start表示pattern的起始段数，之前的不是pattern，然后我们算在整个S1中有多少个pattern出现，patternCnt = (repeatCnt[k] - repeatCnt[start]) * repeat，注意这里的repeatCnt[k] - repeatCnt[start]表示一个pattern中有多少个字符串s2，个人感觉一般来说都是1个。然后我们算出剩下的非pattern的字符串里能包含几个s2，remainCnt = repeatCnt[start + (n1 - start) % interval]，然后我们把patternCnt + remainCnt之和算出来除以n2就是需要的结果啦。如果pattern未曾出现，那么我们直接用repeatCnt[n1] / n2也能得到正确的结果，参见代码如下：

```

1 class Solution {
2 public:
3     int getMaxRepetitions(string s1, int n1, string s2, int n2) {
4         vector<int> repeatCnt(n1 + 1, 0);
5         vector<int> nextIdx(n1 + 1, 0);
6         int j = 0, cnt = 0;
7         for (int k = 1; k <= n1; ++k) {
8             for (int i = 0; i < s1.size(); ++i) {
9                 if (s1[i] == s2[j]) {
10                     ++j;
11                     if (j == s2.size()) {
12                         j = 0;
13                         ++cnt;
14                     }
15                 }
16             }
17             repeatCnt[k] = cnt;
18             nextIdx[k] = j;
19             for (int start = 0; start < k; ++start) {
20                 if (nextIdx[start] == j) {
21                     int interval = k - start;
22                     int repeat = (n1 - start) / interval;
23                     int patternCnt = (repeatCnt[k] - repeatCnt[start]) * repeat;
24                     int remainCnt = repeatCnt[start + (n1 - start) % interval];
25                     return (patternCnt + remainCnt) / n2;
26                 }
27             }
28         }
29         return repeatCnt[n1] / n2;
30     }
31 };

```

467. 封装字符串中的独特子字符串

Consider the string s to be the infinite wraparound string of "abcdefghijklmnopqrstuvwxyz", so s will look like this: "...zabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcd....".

Now we have another string p . Your job is to find out how many unique non-empty substrings of p are present in s . In particular, your input is the string p and you need to output the number of different non-empty substrings of p in the string s .

Note: p consists of only lowercase English letters and the size of p might be over 10000.

这道题说有一个无限长的封装字符串，然后又给了我们另一个字符串 p ，问我们 p 有多少非空子字符串在封装字符串中。我们通过观察题目中的例子可以发现，由于封装字符串是26个字符按顺序无限循环组成的，那么满足题意的 p 的子字符串要么是单一的字符，要么是按字母顺序的子字符串。这道题遍历 p 的所有子字符串会TLE，因为如果 p 很大的话，子字符串很多，会有大量的满足题意的重复子字符串，必须要用到trick，而所谓技巧就是一般来说你想不到的方法。我们看 $abcd$ 这个字符串，以 d 结尾的子字符串有 $abcd$, bcd , cd , d ，那么我们可以发现 bcd 或者 cd 这些以 d 结尾的字符串的子字符串都包含在 $abcd$ 中，那么我们知道以某个字符结束的最大字符串包含其他以该字符结束的字符串的所有子字符串，说起来很拗口，但是理解了我上面举的例子就行。那么题目就可以转换为分别求出以每个字符(a-z)为结束字符的最长连续字符串就行了，我们用一个数组 cnt 记录下来，最后在求出数组 cnt 的所有数字之和就是我们要的结果啦，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     int findSubstringInWraproundString(string p) {
4         vector<int> cnt(26, 0);
5         int len = 0;
6         for (int i = 0; i < p.size(); ++i) {
7             if (i > 0 && (p[i] == p[i - 1] + 1 || p[i - 1] - p[i] == 25)) {
8                 ++len;
9             } else {
10                 len = 1;
11             }
12             cnt[p[i] - 'a'] = max(cnt[p[i] - 'a'], len);
13         }
14         return accumulate(cnt.begin(), cnt.end(), 0);
15     }
16 };

```

下面这种方法跟上面的基本一样，就是在更新每个最大长度时，把差值累加到结果中，这跟最后统一加上最大值的效果一样，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int findSubstringInWraproundString(string p) {
4         vector<int> cnt(26, 0);
5         int res = 0, len = 0;
6         for (int i = 0; i < p.size(); ++i) {
7             int cur = p[i] - 'a';
8             if (i > 0 && p[i - 1] != (cur + 26 - 1) % 26 + 'a') len = 0;
9             if (++len > cnt[cur]) {
10                 res += len - cnt[cur];
11                 cnt[cur] = len;
12             }
13         }
14         return res;
15     }
16 };

```

[468. 验证IP地址](#)

In this problem, your job to write a function to check whether a input string is a valid IPv4 address or IPv6 address or neither.

IPv4 addresses are canonically represented in dot-decimal notation, which consists of four decimal numbers, each ranging from 0 to 255, separated by dots ("."), e.g.,172.16.254.1;

Besides, you need to keep in mind that leading zeros in the IPv4 is illegal. For example, the address 172.16.254.01 is illegal.

IPv6 addresses are represented as eight groups of four hexadecimal digits, each group representing 16 bits. The groups are separated by colons (":"). For example, the address 2001:0db8:85a3:0000:0000:8a2e:0370:7334 is a legal one. Also, we could omit some leading zeros among four hexadecimal digits and some low-case characters in the address to upper-case ones, so 2001:db8:85a3:0:0:8A2E:0370:7334 is also a valid IPv6 address(Omit leading zeros and using upper cases).

However, we don't replace a consecutive group of zero value with a single empty group using two consecutive colons (::) to pursue simplicity. For example, 2001:0db8:85a3::8A2E:0370:7334 is an invalid IPv6 address.

Besides, you need to keep in mind that extra leading zeros in the IPv6 is also illegal. For example, the address 02001:0db8:85a3:0000:0000:8a2e:0370:7334 is also illegal.

这道题让我们验证两种IP地址，LeetCode之前有一道关于IPv4的题Restore IP Addresses，给我们了一个字符串，让我们通过在中间加点来找出所有正确的IP地址，这道题给了我们中间加点或者冒号的字符串，让我们验证其是否是正确的IPv4或者IPv6，感觉要稍稍复杂一些。那么我们只有分别来验证了，那么我们怎么样能快速的区别是IPv4还是IPv6呢，当然是通过中间的点或者冒号啦，所以我们首先在字符串中找冒号(当然你想找点也可以)，如果字符串中没有冒号，那么我们来验证其是否是IPv4，如果有冒号，我们就来验证其是否是IPv6。

首先对于IPv4，我们使用getline函数来截取两个点之间的字符串，我们还需要一个计数器cnt来记录我们已经截取了多少段，如果cnt大于4了，说明超过了4段，说明是不是正确的地址。如果取出的字符串为空，说明两个点连在一起了，也不对。再有就是如果字符串长度大于1，且第一个字符是0，也不对。由于IPv4的地址在0到255之间，所以如果字符串长度大于3，也不正确。下面我们检查每一个字符，如果有不是数字的字符，返回Neither。最后我们再把字符串转为数字，如果不在0到255之间就是非法的。最后的最后，我们要保证cnt正好为4，而且最后一个字符不能是点，统统满足以上条件才是正确的IPv4地址。

然后对于IPv6，我们也使用getline函数来截取两个冒号之间的字符串，我们同样需要计数器cnt来记录我们已经截取了多少段，如果cnt大于8了，说明超过了8段，说明是不是正确的地址。如果取出的字符串为空，说明两个冒号连在一起了，也不对。面我们检查每一个字符，正确的字符应该是0到9之间的数字，或者a到f，或A到F之间的字符，如果出现了其他字符，返回Neither。最后的最后，我们要保证cnt正好为8，而且最后一个字符不能是冒号，统统满足以上条件才是正确的IPv6地址。

```

1 class Solution {
2 public:
3     string validIPAddress(string IP) {
4         istringstream is(IP);
5         string t = "";
6         int cnt = 0;
7         if (IP.find(':') == string::npos) { // Check IPv4
8             while (getline(is, t, '.')) {
9                 ++cnt;
10                if (cnt > 4 || t.empty() || (t.size() > 1 && t[0] == '0') || t.size() > 3)
11                    return "Neither";
12                for (char c : t) {
13                    if (c < '0' || c > '9') return "Neither";
14                }
15                int val = stoi(t);
16                if (val < 0 || val > 255) return "Neither";
17            }
18            return (cnt == 4 && IP.back() != '.') ? "IPv4" : "Neither";
19        } else { // Check IPv6
20            while (getline(is, t, ':')) {
21                ++cnt;
22                if (cnt > 8 || t.empty() || t.size() > 4) return "Neither";
23                for (char c : t) {
24                    if (!(c >= '0' && c <= '9') && !(c >= 'a' && c <= 'f') && !(c >= 'A' &&
25 c <= 'F')) return "Neither";
26                }
27            }
28            return (cnt == 8 && IP.back() != ':') ? "IPv6" : "Neither";
29        }
30    }
31 };

```

469. 凸多边形

Given a list of points that form a polygon when joined sequentially, find if this polygon is convex (Convex polygon definition).

Note:

There are at least 3 and at most 10,000 points.

Coordinates are in the range -10,000 to 10,000.

You may assume the polygon formed by given points is always a simple polygon (Simple polygon definition). In other words, we ensure that exactly two edges intersect at each vertex, and that edges otherwise don't intersect each other.

这道题让我们判断一个多边形是否为凸多边形，我想关于凸多边形的性质，我大天朝的初中几何就应该有所涉猎啦吧，忘了的去面壁。就是所有的顶点角都不大于180度。那么我们该如何快速验证这一个特点呢，学过计算机图形学或者是图像处理的课应该对计算法线normal并不陌生吧，计算的curve的法向量是非常重要的手段，一段连续曲线可以离散看成许多离散点组成，而相邻的三个点就是最基本的单位，我们可以算由三个点组成的一小段曲线的法线方向，而凸多边形的每个三个相邻点的法向量方向都应该相同，要同正，要同负。那么我们只要遍历每个点，然后取出其周围的两个点计算法线方向，然后跟之前的方向对比，如果不一样，直接返回false。这里我们要特别注意法向量为0的情况，如果某一个点的法向量算出来为0，那么正确的pre就会被覆盖为0，后面再遇到相反的法向就无法检测出来，所以我们对计算出来法向量为0的情况直接跳过即可，参见代码如下：

```

1 class Solution {
2 public:
3     bool isConvex(vector<vector<int>>& points) {
4         long long n = points.size(), pre = 0, cur = 0;
5         for (int i = 0; i < n; ++i) {
6             int dx1 = points[(i + 1) % n][0] - points[i][0];
7             int dx2 = points[(i + 2) % n][0] - points[i][0];
8             int dy1 = points[(i + 1) % n][1] - points[i][1];
9             int dy2 = points[(i + 2) % n][1] - points[i][1];
10            cur = dx1 * dy2 - dx2 * dy1;
11            if (cur != 0) {
12                if (cur * pre < 0) return false;
13                else pre = cur;
14            }
15        }
16        return true;
17    }
18 };

```

470. 用 Rand7() 实现 Rand10()

已有方法 rand7 可生成 1 到 7 范围内的均匀随机整数，试写一个方法 rand10 生成 1 到 10 范围内的均匀随机整数。

不要使用系统的 Math.random() 方法。

```

1 class Solution {
2 public:
3     int rand10() {
4         auto x = (rand7()-1)*7 + rand7();
5         return x<=40 ? x%10 + 1 : rand10();
6     }
7 };

```

471. 最短长度编码字符串

Given a non-empty string, encode the string such that its encoded length is the shortest.

The encoding rule is: k[encoded_string], where the encoded_string inside the square brackets is being repeated exactly k times.

Note:

k will be a positive integer and encoded string will not be empty or have extra space.
You may assume that the input string contains only lowercase English letters. The string's length is at most 160.

If an encoding process does not make the string shorter, then do not encode it. If there are several solutions, return any of them is fine.

这道题让我们压缩字符串，把相同的字符串用中括号括起来，然后在前面加上出现的次数，感觉还是一道相当有难度的题呢。参考了网上大神的帖子才弄懂该怎么做，这道题还是应该用DP来做。我们建立一个二维的DP数组，其中dp[i][j]表示s在[i, j]范围内的字符串的缩写形式(如果缩写形式长度大于子字符串，那么还是保留子字符串)，那么如果s字符串的长度是n，最终我们需要的结果就保存在dp[0][n-1]中，然后我们需要遍历s的所有子字符串，对于任意一段子字符串[i, j]，我们\\我们以中间任意位置k来拆分成两段，比较dp[i][k]加上dp[k+1][j]的总长度和dp[i][j]的长度，将长度较小的字符串赋给dp[i][j]，然后我们要做的就是在s中取出[i, j]范围内的子字符串t进行合并。合并的方法是我们在取出的字符串t后面再加上一个t，然后在这里面寻找子字符串t的第二个起始位置，如果第二个起始位置小于t的长度的话，说明t包含重复字符串，举个例子吧，比如 t = "abab"，那么 t+t =

"abababab"，我们在里面找第二个t出现的位置为2，小于t的长度4，说明t中有重复出现，重复的个数为t.size0/pos = 2个，那么我们就要把重复的地方放入中括号中，注意中括号里不能直接放这个子字符串，而是应该从dp中取出对应位置的字符串，因为重复的部分有可能已经写成缩写形式了，比如题目中的例子5。再看一个例子，如果t = "abc"，那么t+t = "abcabc"，我们在里面找第二个t出现的位置为3，等于t的长度3，说明t中没有重复出现，那么replace就还是t。然后我们比较我们得到的replace和dp[i][j]中的字符串长度，把长度较小的赋给dp[i][j]即可，时间复杂度为O(n3)，空间复杂度为O(n2)，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     string encode(string s) {
4         int n = s.size();
5         vector<vector<string>> dp(n, vector<string>(n, ""));
6         for (int step = 1; step <= n; ++step) {
7             for (int i = 0; i + step - 1 < n; ++i) {
8                 int j = i + step - 1;
9                 dp[i][j] = s.substr(i, step);
10                for (int k = i; k < j; ++k) {
11                    string left = dp[i][k], right = dp[k + 1][j];
12                    if (left.size() + right.size() < dp[i][j].size()) {
13                        dp[i][j] = left + right;
14                    }
15                }
16                string t = s.substr(i, j - i + 1), replace = "";
17                auto pos = (t + t).find(t, 1);
18                if (pos >= t.size()) replace = t;
19                else replace = to_string(t.size() / pos) + '[' + dp[i][i + pos - 1] + ']';
20                if (replace.size() < dp[i][j].size()) dp[i][j] = replace;
21            }
22        }
23        return dp[0][n - 1];
24    }
25}

```

根据热心网友iffalse的留言，我们可以优化上面的方法。如果t是重复的，是不是就不需要再看left.size0 + right.size0 < dp[i][j].size0了。例如t是abcababcababcabc，最终肯定是5[abc]，不需要再看3[abc]+abcabc或者abcabc+3[abc]。对于一个本身就重复的字符串，最小的长度肯定是n[REPEATED]，不会是某个left+right。所以应该把k的那个循环放在t和replace那部分代码的后面。这样的确提高了一些运算效率的，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     string encode(string s) {
4         int n = s.size();
5         vector<vector<string>> dp(n, vector<string>(n, ""));
6         for (int step = 1; step <= n; ++step) {
7             for (int i = 0; i + step - 1 < n; ++i) {
8                 int j = i + step - 1;
9                 dp[i][j] = s.substr(i, step);
10                string t = s.substr(i, j - i + 1), replace = "";
11                auto pos = (t + t).find(t, 1);
12                if (pos < t.size()) {
13                    replace = to_string(t.size() / pos) + "[" + dp[i][i + pos - 1] + "]";
14                    if (replace.size() < dp[i][j].size()) dp[i][j] = replace;
15                    continue;
16                }
17                for (int k = i; k < j; ++k) {
18                    string left = dp[i][k], right = dp[k + 1][j];
19                    if (left.size() + right.size() < dp[i][j].size()) {
20                        dp[i][j] = left + right;
21                    }
22                }
23            }
24        }
25        return dp[0][n - 1];
26    }
27 };

```

472. 连接的单词

Given a list of words (without duplicates), please write a program that returns all concatenated words in the given list of words.

A concatenated word is defined as a string that is comprised entirely of at least two shorter words in the given array.

Example:

Input: ["cat", "cats", "catsdogcats", "dog", "dogcatsdog", "hippopotamuses", "rat", "ratcatdogcat"]

Output: ["catsdogcats", "dogcatsdog", "ratcatdogcat"]

这道题给了一个由单词组成的数组，某些单词是可能由其他的单词组成的，让我们找出所有这样的单词。这道题跟之前那道Word Break十分类似，我们可以对每一个单词都调用之前那题的方法，我们首先把所有单词都放到一个unordered_set中，这样可以快速找到某个单词是否在数组中存在。对于当前要判断的单词，我们先将其从set中删去，然后调用之前的Word Break的解法，具体讲解可以参见之前的帖子。如果是可以拆分，那么我们就存入结果res中，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<string> findAllConcatenatedWordsInADict(vector<string>& words) {
4         if (words.size() <= 2) return {};
5         vector<string> res;
6         unordered_set<string> dict(words.begin(), words.end());
7         for (string word : words) {
8             dict.erase(word);
9             int len = word.size();
10            if (len == 0) continue;
11            vector<bool> v(len + 1, false);
12            v[0] = true;
13            for (int i = 0; i < len + 1; ++i) {
14                for (int j = 0; j < i; ++j) {
15                    if (v[j] && dict.count(word.substr(j, i - j))) {
16                        v[i] = true;
17                        break;
18                    }
19                }
20            }
21            if (v.back()) res.push_back(word);
22            dict.insert(word);
23        }
24        return res;
25    }
26};

```

下面这种方法跟上面的方法很类似，不同的是判断每个单词的时候不用将其移除set，而是在判断的过程中加了判断，使其不会判断单词本身是否在集合set中存在，而且由于对单词中子字符串的遍历顺序不同，加了一些优化在里面，使得其运算速度更快一些，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<string> findAllConcatenatedWordsInADict(vector<string>& words) {
4         vector<string> res;
5         unordered_set<string> dict(words.begin(), words.end());
6         for (string word : words) {
7             int n = word.size();
8             if (n == 0) continue;
9             vector<bool> dp(n + 1, false);
10            dp[0] = true;
11            for (int i = 0; i < n; ++i) {
12                if (!dp[i]) continue;
13                for (int j = i + 1; j <= n; ++j) {
14                    if (j - i < n && dict.count(word.substr(i, j - i))) {
15                        dp[j] = true;
16                    }
17                }
18                if (dp[n]) {res.push_back(word); break;}
19            }
20        }
21        return res;
22    }
23};

```

下面这种方法是递归的写法，其中递归函数中的cnt表示有其他单词组成的个数，至少得由其他两个单词组成才符合题意，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     vector<string> findAllConcatenatedWordsInADict(vector<string>& words) {
4         vector<string> res;
5         unordered_set<string> dict(words.begin(), words.end());
6         for (string word : words) {
7             if (word.empty()) continue;
8             if (helper(word, dict, 0, 0)) {
9                 res.push_back(word);
10            }
11        }
12        return res;
13    }
14    bool helper(string& word, unordered_set<string>& dict, int pos, int cnt) {
15        if (pos >= word.size() && cnt >= 2) return true;
16        for (int i = 1; i <= (int)word.size() - pos; ++i) {
17            string t = word.substr(pos, i);
18            if (dict.count(t) && helper(word, dict, pos + i, cnt + 1)) {
19                return true;
20            }
21        }
22        return false;
23    }
24 };

```

473. 火柴棍组成正方形

Remember the story of Little Match Girl? By now, you know exactly what matchsticks the little match girl has, please find out a way you can make one square by using up all those matchsticks. You should not break any stick, but you can link them up, and each matchstick must be used exactly one time.

Your input will be several matchsticks the girl has, represented with their stick length. Your output will either be true or false, to represent whether you could make one square using all the matchsticks the little match girl has.

Example 1:

Input: [1,1,2,2,2]
Output: true

我已经服了LeetCode了，连卖火柴的小女孩也能改编成题目，还能不能愉快的玩耍了，坐等灰姑娘，丑小鸭的改编题了。好了，言归正传，这道题让我们用数组中的数字来摆出一个正方形。跟之前有道题Partition Equal Subset Sum有点像，那道题问我们能不能将一个数组分成和相等的两个子数组，而这道题实际上是让我们将一个数组分成四个和相等的子数组。我一开始尝试着用那题的解法来做，首先来判断数组之和是否是4的倍数，然后还是找能否分成和相等的两个子数组，但是在遍历的时候加上判断如果数组中某一个数字大于一条边的长度时返回false。最后我们同时检查dp数组中一条边长度位置上的值跟两倍多一条边长度位置上的值是否为true，这种方法不幸TLE了。所以只能上论坛求助各路大神了，发现了可以用优化过的递归来解，递归的方法基本上等于brute force，但是C++版本的直接递归没法通过OJ，而是要先给数组从大到小的顺序排序，这样大的数字先加，如果超过target了，就直接跳过了后面的再次调用递归的操作，效率会提高不少，所以会通过OJ。下面来看代码，我们建立一个

长度为4的数组sums来保存每个边的长度和，我们希望每条边都等于target，数组总和的四分之一。然后我们遍历sums中的每条边，我们判断如果加上数组中的当前数字大于target，那么我们跳过，如果没有，我们就加上这个数字，然后对数组中下一个位置调用递归，如果返回为真，我们返回true，否则我们再从sums中对应位置将这个数字减去继续循环，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool makesquare(vector<int>& nums) {
4         if (nums.empty() || nums.size() < 4) return false;
5         int sum = accumulate(nums.begin(), nums.end(), 0);
6         if (sum % 4 != 0) return false;
7         vector<int> sums(4, 0);
8         sort(nums.rbegin(), nums.rend());
9         return helper(nums, sums, 0, sum / 4);
10    }
11    bool helper(vector<int>& nums, vector<int>& sums, int pos, int target) {
12        if (pos >= nums.size()) {
13            return sums[0] == target && sums[1] == target && sums[2] == target;
14        }
15        for (int i = 0; i < 4; ++i) {
16            if (sums[i] + nums[pos] > target) continue;
17            sums[i] += nums[pos];
18            if (helper(nums, sums, pos + 1, target)) return true;
19            sums[i] -= nums[pos];
20        }
21        return false;
22    }
23 };

```

其实这题还有迭代的方法，很巧妙的利用到了位操作的特性，前面的基本求和跟判断还是一样，然后建立一个变量all，初始化为 $(1 << n) - 1$ ，这是什么意思呢，all其实是一个mask，数组中有多少个数字，all就有多少个1，表示全选所有的数字，然后变量target表示一条边的长度。我们建立两个一位向量masks和validHalf，其中masks保存和target相等的几个数字位置的mask，validHalf保存某个mask是否是总和的一半。然后我们从0遍历到all，实际上就是遍历所有子数组，然后我们根据mask来计算出子数组的和，注意这里用了15，而不是32，因为题目中说了数组元素个数不会超过15个。我们算出的子数组之和如果等于一条边的长度target，我们遍历masks数组中其他等于target的子数组，如果两个mask相与不为0，说明有公用的数字，直接跳过；否则将两个mask或起来，说明我们当前选的数字之和为数组总和的一半，更新validHalf的对应位置，然后我们通过all取出所有剩下的数组，并在validHalf里查找，如果也为true，说明我们成功的找到了四条边，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool makesquare(vector<int>& nums) {
4         if (nums.empty() || nums.size() < 4) return false;
5         int sum = accumulate(nums.begin(), nums.end(), 0);
6         if (sum % 4 != 0) return false;
7         int n = nums.size(), all = (1 << n) - 1, target = sum / 4;
8         vector<int> masks, validHalf(1 << n, false);
9         for (int i = 0; i <= all; ++i) {
10             int curSum = 0;
11             for (int j = 0; j <= 15; ++j) {
12                 if ((i >> j) & 1) curSum += nums[j];
13             }
14             if (curSum == target) {
15                 for (int mask : masks) {
16                     if ((mask & i) != 0) continue;
17                     int half = mask | i;
18                     validHalf[half] = true;
19                     if (validHalf[all ^ half]) return true;
20                 }
21                 masks.push_back(i);
22             }
23         }
24         return false;
25     }
26 };

```

474. 一和零

In the computer world, use restricted resource you have to generate maximum benefit is what we always want to pursue.

For now, suppose you are a dominator of m 0s and n 1s respectively. On the other hand, there is an array with strings consisting of only 0s and 1s.

Now your task is to find the maximum number of strings that you can form with given m 0s and n 1s. Each 0 and 1 can be used at most once.

Note:

The given numbers of 0s and 1s will both not exceed 100
The size of given string array won't exceed 600.

这道题是一道典型的应用DP来解的题，如果我们看到这种求总数，而不是列出所有情况的题，十有八九都是用DP来解，重中之重就是在于找出递推式。如果你第一反应没有想到用DP来做，想得是用贪心算法来做，比如先给字符串数组排个序，让长度小的字符串在前面，然后遍历每个字符串，遇到0或者1就将对应的m和n的值减小，这种方法在有的时候是不对的，比如对于{"11", "01", "10"}, m=2, n=2这个例子，我们将遍历完"11"的时候，把1用完了，那么对于后面两个字符串就没法处理了，而其实正确的答案是应该组成后面两个字符串才对。所以我们需要建立一个二维的DP数组，其中dp[i][j]表示有i个0和j个1时能组成的最多字符串的个数，而对于当前遍历到的字符串，我们统计出其中0和1的个数为zeros和ones，然后dp[i - zeros][j - ones]表示当前的i和j减去zeros和ones之前能拼成字符串的个数，那么加上当前的zeros和ones就是当前dp[i][j]可以达到的个数，我们跟其原有数值对比取较大值即可，所以递推式如下：

```
dp[i][j] = max(dp[i][j], dp[i - zeros][j - ones] + 1);
```

有了递推式，我们就可以很容易的写出代码如下：

```

1 class Solution {
2 public:
3     int findMaxForm(vector<string>& strs, int m, int n) {
4         vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
5         for (string str : strs) {
6             int zeros = 0, ones = 0;
7             for (char c : str) (c == '0') ? ++zeros : ++ones;
8             for (int i = m; i >= zeros; --i) {
9                 for (int j = n; j >= ones; --j) {
10                     dp[i][j] = max(dp[i][j], dp[i - zeros][j - ones] + 1);
11                 }
12             }
13         }
14         return dp[m][n];
15     }
16 };

```

475. 加热器

Winter is coming! Your first job during the contest is to design a standard heater with fixed warm radius to warm all the houses.

Now, you are given positions of houses and heaters on a horizontal line, find out minimum radius of heaters so that all houses could be covered by those heaters.

So, your input will be the positions of houses and heaters separately, and your expected output will be the minimum radius standard of heaters.

这道题是一道蛮有意思的题目，首先我们看题目中的例子，不管是houses还是heaters数组都是有序的，所以我们也需要给输入的这两个数组先排序，以免其为乱序。我们就拿第二个例子来分析，我们的目标是houses中的每一个数字都要被cover到，那么我们就遍历houses数组，对每一个数组的数字，我们在heaters中找能包含这个数字的左右范围，然后看离左右两边谁近取谁的值，如果某个house位置比heaters中最小的数字还小，那么肯定要用最小的heater去cover，反之如果比最大的数字还大，就用最大的数字去cover。对于每个数字算出的半径，我们要取其中最大的值。通过上面的分析，我们就不难写出代码了，我们在heater中两个数一组进行检查，如果后面一个数和当前house位置差的绝对值小于等于前面一个数和当前house位置差的绝对值，那么我们继续遍历下一个位置的数。跳出循环的条件是遍历到heater中最后一个数，或者上面的小于等于不成立，此时heater中的值和当前house位置的差的绝对值就是能cover当前house的最小半径，我们更新结果res即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int findRadius(vector<int>& houses, vector<int>& heaters) {
4         int n = heaters.size(), j = 0, res = 0;
5         sort(houses.begin(), houses.end());
6         sort(heaters.begin(), heaters.end());
7         for (int i = 0; i < houses.size(); ++i) {
8             int cur = houses[i];
9             while (j < n - 1 && abs(heaters[j + 1] - cur) <= abs(heaters[j] - cur)) {
10                 ++j;
11             }
12             res = max(res, abs(heaters[j] - cur));
13         }
14         return res;
15     }
16 };

```

还是上面的思路，我们可以用二分查找法来快速找到第一个大于等于当前house位置的数，如果这个数存在，那么我们可以算出其和house的差值，并且如果这个数不是heater的首数字，我们可以算出house和前面一个数的差值，这两个数中取较小的为cover当前house的最小半径，然后我们每次更新结果res即可，参见代码如下：

解法2：

```
1 class Solution {
2     public:
3         int findRadius(vector<int>& houses, vector<int>& heaters) {
4             int res = 0, n = heaters.size();
5             sort(heaters.begin(), heaters.end());
6             for (int house : houses) {
7                 int left = 0, right = n;
8                 while (left < right) {
9                     int mid = left + (right - left) / 2;
10                    if (heaters[mid] < house) left = mid + 1;
11                    else right = mid;
12                }
13                int dist1 = (right == n) ? INT_MAX : heaters[right] - house;
14                int dist2 = (right == 0) ? INT_MAX : house - heaters[right - 1];
15                res = max(res, min(dist1, dist2));
16            }
17            return res;
18        }
19    };
}
```

我们可以用STL中的lower_bound来代替二分查找的代码来快速找到第一个大于等于目标值的位置，其余部分均和上面方法相同，参见代码如下：

解法3：

```
1 class Solution {
2     public:
3         int findRadius(vector<int>& houses, vector<int>& heaters) {
4             int res = 0;
5             sort(heaters.begin(), heaters.end());
6             for (int house : houses) {
7                 auto pos = lower_bound(heaters.begin(), heaters.end(), house);
8                 int dist1 = (pos == heaters.end()) ? INT_MAX : *pos - house;
9                 int dist2 = (pos == heaters.begin()) ? INT_MAX : house - *(--pos);
10                res = max(res, min(dist1, dist2));
11            }
12            return res;
13        }
14    };
}
```

476. 补数

Given a positive integer, output its complement number. The complement strategy is to flip the bits of its binary representation.

这道题给了我们一个数，让我们求补数。通过分析题目汇总的例子，我们知道需要做的就是每个位翻转一下就行了，但是翻转的起始位置上从最高位的1开始的，前面的0是不能被翻转的，所以我们从高往低遍历，如果遇到第一个1了后，我们的flag就赋值为true，然后就可以进行翻转了，翻转的方法就是对应位异或一个1即可，参见代码如下：

解法1:

```
1 class Solution {
2 public:
3     int findComplement(int num) {
4         bool start = false;
5         for (int i = 31; i >= 0; --i) {
6             if (num & (1 << i)) start = true;
7             if (start) num ^= (1 << i);
8         }
9         return num;
10    }
11 };

```

CPP

由于位操作里面的取反符号~本身就可以翻转位，但是如果直接对num取反的话就是每一位都翻转了，而最高位1之前的0是不能翻转的，所以我们只要用一个mask来标记最高位1前面的所有0的位置，然后对mask取反后，与上对num取反的结果即可，参见代码如下：

解法2:

```
1 class Solution {
2 public:
3     int findComplement(int num) {
4         int mask = INT_MAX;
5         while (mask & num) mask <<= 1;
6         return ~mask & ~num;
7     }
8 };

```

CPP

再来看一种迭代的写法，一行搞定碉堡了，思路就是每次都右移一位，并根据最低位的值先进行翻转，如果当前值小于等于1了，就不用再调用递归函数了，参见代码如下：

解法3:

```
1 class Solution {
2 public:
3     int findComplement(int num) {
4         return (1 - num % 2) + 2 * (num <= 1 ? 0 : findComplement(num / 2));
5     }
6 };

```

CPP

477. 全部汉明距离

The Hamming distance between two integers is the number of positions at which the corresponding bits are different.

Now your job is to find the total Hamming distance between all pairs of the given numbers.

Example:

Input: 4, 14, 2

Output: 6

这道题是之前那道Hamming Distance的拓展，由于有之前那道题的经验，我们知道需要用异或来求每个位上的情况，那么我们需要来找出某种规律来，比如我们看下面这个例子，4, 14, 2和1：

```
4:      0 1 0 0
14:     1 1 1 0
2:      0 0 1 0
1:      0 0 0 1
```

我们先看最后一列，有三个0和一个1，那么它们之间相互的汉明距离就是3，即1和其他三个0分别的距离累加，然后在看第三列，累加汉明距离为4，因为每个1都会跟两个0产生两个汉明距离，同理第二列也是4，第一列是3。我们仔细观察累计汉明距离和0跟1的个数，我们可以发现其实就是0的个数乘以1的个数，发现了这个重要的规律，那么整道题就迎刃而解了，只要统计出每一位的1的个数即可，参见代码如下：

```
1 class Solution {
2 public:
3     int totalHammingDistance(vector<int>& nums) {
4         int res = 0, n = nums.size();
5         for (int i = 0; i < 32; ++i) {
6             int cnt = 0;
7             for (int num : nums) {
8                 if (num & (1 << i)) ++cnt;
9             }
10            res += cnt * (n - cnt);
11        }
12        return res;
13    }
14};
```

CPP

478. 在圆内随机生成点

给定圆的半径和圆心的x、y坐标，写一个在圆中产生均匀随机点的函数 randPoint。

```
1 class Solution {
2 public:
3     Solution(double radius, double x_center, double y_center): distribution(0, 1),
4 R(radius), X(x_center), Y(y_center) {}
5
6     vector<double> randPoint() {
7         double theta = distribution(generator) * 2 * PI;
8         double r = sqrt(distribution(generator)) * R;
9         return {X + r * cos(theta), Y + r * sin(theta)};
10    }
11
12 private:
13     const double PI = 3.1415926535897932384626;
14     std::default_random_engine generator;
15     std::uniform_real_distribution<double> distribution;
16     const double R;
17     const double X;
18     const double Y;
19};
```

CPP

479. 最大回文串乘积

Find the largest palindrome made from the product of two n-digit numbers.

Since the result could be very large, you should return the largest palindrome mod 1337.

Example:

Input: 2

Output: 987

Explanation: $99 \times 91 = 9009$, $9009 \% 1337 = 987$

这道题给我们一个数字n，问两个n位数的乘积组成的大回文数是多少，返回的结果对1337取余。博主刚开始用暴力搜索做，遍历所有的数字组合，求乘积，再来判断是否是回文数，最终TLE了，只能换一种思路来做。论坛上的这种思路真心吓啊，博主感觉这题绝比不该Easy啊。首先我们还是要确定出n位数的范围，最大值upper，可以取到，最小值lower，不能取到。然后我们遍历这区间的所有数字，对于每个遍历到的数字，我们用当前数字当作回文数的前半段，将其翻转一下拼接到后面，此时组成一个回文数，这里用到了一个规律，当n>1时，两个n位数乘积的最大回文数一定是2n位的。下面就要来验证这个回文数能否由两个n位数相乘的来，我们还是遍历区间中的数，从upper开始遍历，但此时结束位置不是lower，而是当前数的平方大于回文数，因为我们遍历的是相乘得到回文数的两个数中的较大数，一旦超过这个范围，就变成较小数了，就重复计算了。比如对于回文数9009，其是由99和91组成的，其较大数的范围是[99,95]，所以当遍历到94时，另一个数至少需要是95，而这种情况在之前已经验证过了。当回文数能整除较大数时，说明是成立的，直接对1337取余返回即可，参见代码如下：

```

1 class Solution {
2 public:
3     int largestPalindrome(int n) {
4         int upper = pow(10, n) - 1, lower = upper / 10;
5         for (int i = upper; i > lower; --i) {
6             string t = to_string(i);
7             long p = stol(t + string(t.rbegin(), t.rend()));
8             for (long j = upper; j * j > p; --j) {
9                 if (p % j == 0) return p % 1337;
10            }
11        }
12        return 9;
13    }
14 }
```

CPP

480. 滑动窗口中位数

Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.

Examples:

[2,3,4] , the median is 3

[2,3] , the median is $(2 + 3) / 2 = 2.5$

Given an array nums, there is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position. Your job is to output the median array for each window in the original array.

这道题给了我们一个数组，还是滑动窗口的大小，让我们求滑动窗口的中位数。我想起来之前也有一道滑动窗口的题Sliding Window Maximum，于是想套用那道题的方法，可以用deque怎么也做不出，因为求中位数并不是像求最大值那样只操作deque的首尾元素。后来看到了史蒂芬大神的方法，原来是要用一个multiset集合，和一个指向最中间元素的iterator。我们首先将数组的前k个数组加入集合中，由于multiset自带排序功能，所以我们通过k/2能快速的找到指向最中间的数字的迭代器mid，如果k为奇数，那么mid指向的数字就是中位数；如果k为偶数，那么mid指向的数跟前面那个数求平均值就是中位数。当我们添加新的数字到集合中，multiset会根据新数字的大小加到正确的位置，然后我们看如果这个新加入的数字比之前的mid指向的数小，那么中位数肯定被拉低了，所以mid往前移动一个，再看如果要删掉的数小于等于mid指向的数(注意这里加等号是因为要删的数可能就是mid指向的数)，则mid向后移动一个。然后我们将滑动窗口最左边的数删掉，我们不能直接根据值来用erase来删数字，因为这样有可能删掉多个相同的数字，而是应该用lower_bound来找到第一个不小于目标值的数，通过iterator来删掉确定的一个数字，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<double> medianSlidingWindow(vector<int>& nums, int k) {
4         vector<double> res;
5         multiset<double> ms(nums.begin(), nums.begin() + k);
6         auto mid = next(ms.begin(), k / 2);
7         for (int i = k; ; ++i) {
8             res.push_back((*mid + *prev(mid, 1 - k % 2)) / 2);
9             if (i == nums.size()) return res;
10            ms.insert(nums[i]);
11            if (nums[i] < *mid) --mid;
12            if (nums[i - k] <= *mid) ++mid;
13            ms.erase(ms.lower_bound(nums[i - k]));
14        }
15    }
16 };

```

CPP

上面的方法用到了很多STL内置的函数，比如next、lower_bound啥的，下面我们来看一种不使用这些函数的解法。这种解法跟Find Median from Data Stream那题的解法很类似，都是维护了small和large两个堆，分别保存有序数组的左半段和右半段的数字，保持small的长度大于等于large的长度。我们开始遍历数组nums，如果i>=k，说明此时滑动窗口已经满k个了，再滑动就要删掉最左值了，我们分别在small和large中查找最左值，有的话就删掉。然后处理增加数字的情况（分两种情况：1.如果small的长度小于large的长度，再看如果large是空或者新加的数小于等于large的首元素，我们把此数加入small中。否则就把large的首元素移出并加入small中，然后把新数字加入large。2.如果small的长度大于large，再看如果新数字大于small的尾元素，那么新数字加入large中，否则就把small的尾元素移出并加入large中，把新数字加入small中）。最后我们再计算中位数并加入结果res中，根据k的奇偶性来分别处理，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<double> medianSlidingWindow(vector<int>& nums, int k) {
4         vector<double> res;
5         multiset<int> small, large;
6         for (int i = 0; i < nums.size(); ++i) {
7             if (i >= k) {
8                 if (small.count(nums[i - k])) small.erase(small.find(nums[i - k]));
9                 else if (large.count(nums[i - k])) large.erase(large.find(nums[i - k]));
10            }
11            if (small.size() <= large.size()) {
12                if (large.empty() || nums[i] <= *large.begin()) small.insert(nums[i]);
13                else {
14                    small.insert(*large.begin());
15                    large.erase(large.begin());
16                    large.insert(nums[i]);
17                }
18            } else {
19                if (nums[i] >= *small.rbegin()) large.insert(nums[i]);
20                else {
21                    large.insert(*small.rbegin());
22                    small.erase(--small.end());
23                    small.insert(nums[i]);
24                }
25            }
26            if (i >= (k - 1)) {
27                if (k % 2) res.push_back(*small.rbegin());
28                else res.push_back((double)*small.rbegin() + *large.begin() / 2);
29            }
30        }
31        return res;
32    }
33 };

```

481. 神奇字符串

A magical string S consists of only '1' and '2' and obeys the following rules:

The string S is magical because concatenating the number of contiguous occurrences of characters '1' and '2' generates the string S itself.

The first few elements of string S is the following: S = "122112122122112112....."

If we group the consecutive '1's and '2's in S, it will be:

1 22 11 2 1 22 1 22 11 2 11 22

and the occurrences of '1's or '2's in each group are:

1 2 2 1 1 2 1 2 2 1 2 2

You can see that the occurrence sequence above is the S itself.

Given an integer N as input, return the number of '1's in the first N number in the magical string S.

Note: N will not exceed 100,000.

这道题介绍了一种神奇字符串，只由1和2组成，通过计数1组和2组的个数，又能生成相同的字符串。而让我们求前n个数字中1的个数说白了其实就是在我们按规律生成这个神奇字符串，只有生成了字符串的前n个字符，才能统计出1的个数。其实这道题的难点就是在于找到规律来生成字符串，这里我们就直接说规律了，因为博主也没有自己找到，都是看了网上大神们的解法。根据第三个数字2开始往后生成数字，此时生成两个1，然后根据第四个数字1，生成一个2，再根据第五个数字1，生成一个1，以此类推，生成的数字1或2可能通过异或3来交替生成，在生成的过程中同时统计1的个数即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int magicalString(int n) {
4         if (n <= 0) return 0;
5         if (n <= 3) return 1;
6         int res = 1, head = 2, tail = 3, num = 1;
7         vector<int> v{1, 2, 2};
8         while (tail < n) {
9             for (int i = 0; i < v[head]; ++i) {
10                 v.push_back(num);
11                 if (num == 1 && tail < n) ++res;
12                 ++tail;
13             }
14             num ^= 3;
15             ++head;
16         }
17         return res;
18     }
19 };

```

CPP

下面这种解法的思路跟上面一样，但是写法上面大大的简洁了，感觉很呀！

解法2：

```

1 class Solution {
2 public:
3     int magicalString(int n) {
4         string s = "122";
5         int i = 2;
6         while (s.size() < n) {
7             s += string(s[i++] - '0', s.back() ^ 3);
8         }
9         return count(s.begin(), s.begin() + n, '1');
10    }
11 };

```

CPP

482. 注册码格式化

Now you are given a string S, which represents a software license key which we would like to format. The string S is composed of alphanumerical characters and dashes. The dashes split the alphanumerical characters within the string into groups. (i.e. if there are M dashes, the string is split into M+1 groups). The dashes in the given string are possibly misplaced.

We want each group of characters to be of length K (except for possibly the first group, which could be shorter, but still must contain at least one character). To satisfy this requirement, we will reinsert dashes. Additionally, all the lower case letters in the string must be converted to upper case.

So, you are given a non-empty string S, representing a license key to format, and an integer K. And you need to return the license key formatted according to the description above.

这道题让我们对注册码进行格式化，正确的注册码的格式是每四个字符后面跟一个短杠，每一部分的长度为K，第一部分长度可以小于K，另外，字母必须是大写的。那么由于第一部分可以不为K，那么我们可以反过来想，我们从S的尾部往前遍历，把字符加入结果res，每K个后面加一个短杠，那么最后遍历完再把res翻转一下即可，注意翻转之前要把结尾的短杠去掉(如果有的话)，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     string licenseKeyFormatting(string S, int K) {
4         string res = "";
5         int cnt = 0, n = S.size();
6         for (int i = n - 1; i >= 0; --i) {
7             char c = S[i];
8             if (c == '-') continue;
9             if (c >= 'a' && c <= 'z') c -= 32;
10            res.push_back(c);
11            if (++cnt % K == 0) res.push_back('-');
12        }
13        if (!res.empty() && res.back() == '-') res.pop_back();
14        return string(res.rbegin(), res.rend());
15    }
16};
```

上面代码可以进一步精简到下面这种，我们用到了自带函数toupper，把字母转为大写格式，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     string licenseKeyFormatting(string S, int K) {
4         string res = "";
5         for (int i = (int)S.size() - 1; i >= 0; --i) {
6             if (S[i] != '-') {
7                 ((res.size() % (K + 1) - K) ? res : res += '-') += toupper(S[i]);
8             }
9         }
10        return string(res.rbegin(), res.rend());
11    }
12};
```

483. 最小的好基数

For an integer n, we call k>=2 a good base of n, if all digits of n base k are 1.

Now given a string representing n, you should return the smallest good base of n in string format.

Example 1:

Input: "13"

Output: "3"

Explanation: 13 base 3 is 111.

这道题让我们求最小的好基数，定义了一个大于等于2的基数k，如果可以把数字n转化为各位都是1的数，那么就称这个基数k是好基数。通过看题目中的三个例子，应该大致可以理解题意了吧。如果我们用k表示基数，m表示转为全1数字的位数，那么数字n就可以拆分为：

$$n = 1 + k + k^2 + k^3 + \dots + k^{(m-1)}$$

这是一个等比数列，中学数学的内容吧，利用求和公式可以表示为 $n = (k^m - 1) / (k - 1)$ 。我们的目标是求最小的k，那么仔细观察这个式子，在n恒定的情况下，k越小则m却大，那么就是说上面的等式越长越好。下面我们来分析m的取值范围，题目中给了n的范围，是[3, 10^18]。那么由于k至少为2，n至少为3，那么肯定至少有两项，则m>=2。那么m的上限该如何求？其实也不难，想要m最大，那么k就要最小，k最小是2，那么m最大只能为log2(n + 1)，数字n用二进制表示的时候可拆分出的项最多。但这道题要求变换后的数各位都是1，那么我们看题目中最后一个例子，可以发现，当k=n-1时，一定能变成11，所以实在找不到更小的情况下就返回n-1。

下面我们来确定k的范围，由于k至少为2，那么我们可以根据下面这个不等式来求k的上限：

$$n = 1 + k + k^2 + k^3 + \dots + k^{(m-1)} > k^{(m-1)}$$

解出 $k < n^{(1 / (m-1))}$ ，其实我们也可以可以通过 $n < k^m - 1$ 来求出k的准确的下限，但由于是二分查找法，下限直接使用2也没啥问题。分析到这里，那么解法应该已经跃然纸上了，我们遍历所有可能的m值，然后利用二分查找法来确定k的值，对每一个k值，我们通过联合m值算出总和sum，然后跟n来对比即可，参见代码如下：

```

1 class Solution {
2 public:
3     string smallestGoodBase(string n) {
4         long long num = stol(n);
5         for (int i = log(num + 1) / log(2); i >= 2; --i) {
6             long long left = 2, right = pow(num, 1.0 / (i - 1)) + 1;
7             while (left < right) {
8                 long long mid = left + (right - left) / 2, sum = 0;
9                 for (int j = 0; j < i; ++j) {
10                     sum = sum * mid + 1;
11                 }
12                 if (sum == num) return to_string(mid);
13                 else if (sum < num) left = mid + 1;
14                 else right = mid;
15             }
16         }
17         return to_string(num - 1);
18     }
19 };

```

CPP

484. 找全排列

By now, you are given a secret signature consisting of character 'D' and 'I'. 'D' represents a decreasing relationship between two numbers, 'I' represents an increasing relationship between two numbers. And our secret signature was constructed by a special integer array, which contains uniquely all the different number from 1 to n (n is the length of the secret signature plus 1). For example, the secret signature "DI" can be constructed by array [2,1,3] or [3,1,2], but won't be constructed by array [3,2,4] or [2,1,3,4], which are both illegal constructing special string that can't represent the "DI" secret signature.

On the other hand, now your job is to find the lexicographically smallest permutation of [1, 2, ... n] could refer to the given secret signature in the input

这道题给了我们一个由D和I两个字符组成的字符串，分别表示对应位置的升序和降序，要我们根据这个字符串生成对应的数字字符串。由于受名字中的permutation的影响，感觉做法应该是找出所有的全排列然后逐个数字验证，这种方法十有八九无法通过OJ。其实这题用贪心算法最为简单，我们来看一个例子：

D D I I D I

1 2 3 4 5 6 7

3 2 1 4 6 5 7

我们不难看出，只有D对应的位置附近的数字才需要变换，而且变换方法就是倒置一下字符串，我们要做的就是通过D的位置来确定需要倒置的子字符串的起始位置和长度即可。通过观察，我们需要记录D的起始位置i，还有D的连续个数k，那么我们只需要在数组中倒置[i, i+k]之间的数字即可，根据上述思路可以写出代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<int> findPermutation(string s) {
4         int n = s.size(), cnt = 0;
5         vector<int> res(n + 1);
6         for (int i = 0; i < n + 1; ++i) res[i] = i + 1;
7         for (int i = 0; i < n; ++i) {
8             if (s[i] == 'D') {
9                 int j = i;
10                while (s[i] == 'D' && i < n) ++i;
11                reverse(res.begin() + j, res.begin() + i + 1);
12                --i;
13            } else {
14                cnt = 0;
15            }
16        }
17        return res;
18    }
19 }
```

CPP

下面这种方法没有用到数组倒置，而是根据情况来往结果res中加入正确顺序的数字，我们遍历s字符串，遇到D直接跳过，遇到I进行处理，我们每次先记录下结果res的长度size，然后从i+1的位置开始往size遍历，将数字加入结果res中即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<int> findPermutation(string s) {
4         vector<int> res;
5         for (int i = 0; i < s.size() + 1; ++i) {
6             if (i == s.size() || s[i] == 'I') {
7                 int size = res.size();
8                 for (int j = i + 1; j > size; --j) {
9                     res.push_back(j);
10                }
11            }
12        }
13        return res;
14    }
15 };

```

485. 最大连续1的个数

Given a binary array, find the maximum number of consecutive 1s in this array.

Example 1:

Input: [1,1,0,1,1,1]

Output: 3

Explanation: The first two digits or the last three digits are consecutive 1s.

The maximum number of consecutive 1s is 3.

这道题让我们求最大连续1的个数，不是一道难题。我们可以遍历一遍数组，用一个计数器cnt来统计1的个数，方法是如果当前数字为0，那么cnt重置为0，如果不是0，cnt自增1，然后每次更新结果res即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int findMaxConsecutiveOnes(vector<int>& nums) {
4         int res = 0, cnt = 0;
5         for (int num : nums) {
6             cnt = (num == 0) ? 0 : cnt + 1;
7             res = max(res, cnt);
8         }
9         return res;
10    }
11 };

```

由于是个二进制数组，所以数组中的数字只能是0或1，那么连续1的和跟个数相等，所以我们可以计算和，通过加上num，再乘以num来计算，如果当前数字是0，那么sum就被重置为0，还是要更新结果res，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int findMaxConsecutiveOnes(vector<int>& nums) {
4         int res = 0, sum = 0;
5         for (int num : nums) {
6             sum = (sum + num) * num;
7             res = max(res, sum);
8         }
9         return res;
10    }
11 };

```

486. 预测赢家

Given an array of scores that are non-negative integers. Player 1 picks one of the numbers from either end of the array followed by the player 2 and then player 1 and so on. Each time a player picks a number, that number will not be available for the next player. This continues until all the scores have been chosen. The player with the maximum score wins.

Given an array of scores, predict whether player 1 is the winner. You can assume each player plays to maximize his score.

这道题给了一个小游戏，有一个数组，两个玩家轮流取数，说明了只能从开头或结尾取，问我们第一个玩家能赢吗。这道题我想到了应该是用Minimax来做，由于之前有过一道这样的题Guess Number Higher or Lower II，所以依稀记得应该要用递归的方法，而且当前玩家赢返回true的条件就是递归调用下一个玩家输返回false。我们需要一个变量来标记当前是第几个玩家，还需要两个变量来分别记录两个玩家的当前数字和，在递归函数里面，如果当前数组为空了，我们直接比较两个玩家的当前得分即可，如果数组中只有一个数字了，我们根据玩家标识来将这个数字加给某个玩家并进行比较总得分。如果数组有多个数字，我们分别生成两个新数组，一个是去掉首元素，一个是去掉尾元素，然后根据玩家标识分别调用不同的递归，只要下一个玩家两种情况下任意一种返回false了，那么当前玩家就可以赢了，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool PredictTheWinner(vector<int>& nums) {
4         return canWin(nums, 0, 0, 1);
5     }
6     bool canWin(vector<int> nums, int sum1, int sum2, int player) {
7         if (nums.empty()) return sum1 >= sum2;
8         if (nums.size() == 1) {
9             if (player == 1) return sum1 + nums[0] >= sum2;
10            else if (player == 2) return sum2 + nums[0] > sum1;
11        }
12        vector<int> va = vector<int>(nums.begin() + 1, nums.end());
13        vector<int> vb = vector<int>(nums.begin(), nums.end() - 1);
14        if (player == 1) {
15            return !canWin(va, sum1 + nums[0], sum2, 2) || !canWin(vb, sum1 + nums.back(),
16            sum2, 2);
17        } else if (player == 2) {
18            return !canWin(va, sum1, sum2 + nums[0], 1) || !canWin(vb, sum1, sum2 +
19            nums.back(), 1);
20        }
21    }
22};

```

我们还可以使用DP加Minimax的方法来做，先来看递归的写法，十分的简洁。DP数组的作用是保存中间结果，再次遇到相同情况时直接返回不用再次计算，提高了运算效率：

解法2：

```
1 class Solution {
2 public:
3     bool PredictTheWinner(vector<int>& nums) {
4         int n = nums.size();
5         vector<vector<int>> dp(n, vector<int>(n, -1));
6         return canWin(nums, 0, n - 1, dp) >= 0;
7     }
8     int canWin(vector<int>& nums, int s, int e, vector<vector<int>>& dp) {
9         if (dp[s][e] == -1) {
10             dp[s][e] = (s == e) ? nums[s] : max(nums[s] - canWin(nums, s + 1, e, dp),
11                                         nums[e] - canWin(nums, s, e - 1, dp));
12         }
13         return dp[s][e];
14     }
};
```

CPP

下面这种方法是DP加Minimax的递归写法，要注意的是DP的更新顺序，跟以往不太一样，这种更新方法是按区间来更新的，感觉之前好像没有遇到过这种更新的方法，还蛮特别的：

解法3：

```
1 class Solution {
2 public:
3     bool PredictTheWinner(vector<int>& nums) {
4         int n = nums.size();
5         vector<vector<int>> dp(n, vector<int>(n, 0));
6         for (int i = 0; i < n; ++i) dp[i][i] = nums[i];
7         for (int len = 1; len < n; ++len) {
8             for (int i = 0, j = len; j < n; ++i, ++j) {
9                 dp[i][j] = max(nums[i] - dp[i + 1][j], nums[j] - dp[i][j - 1]);
10            }
11        }
12        return dp[0][n - 1] >= 0;
13    }
};
```

CPP

487. 最大连续1的个数之二

Given a binary array, find the maximum number of consecutive 1s in this array if you can flip at most one 0.

Example 1:

Input: [1,0,1,1,0]

Output: 4

Explanation: Flip the first zero will get the the maximum number of consecutive 1s.

After flipping, the maximum number of consecutive 1s is 4.

这道题在之前那道题Max Consecutive Ones的基础上加了一个条件，说我们有一次将0翻转成1的机会，问此时最大连续1的个数，再看看follow up中的说明，很明显是让我们只遍历一次数组，那我们想，肯定需要用一个变量cnt来记录连续1的个数吧，那么当遇到了0的时候怎么处理呢，因为我们有一次0变1的机会，所以我们遇到0了还是要累加cnt，然后我们此时需要用另外一个变量cur来保存当前cnt的值，然后cnt重置为0，以便于让cnt一直用来统计纯连续1的个数，然后我们每次都用用cnt+cur来更新结果res，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int findMaxConsecutiveOnes(vector<int>& nums) {
4         int res = 0, cur = 0, cnt = 0;
5         for (int num : nums) {
6             ++cnt;
7             if (num == 0) {
8                 cur = cnt;
9                 cnt = 0;
10            }
11            res = max(res, cnt + cur);
12        }
13        return res;
14    }
15 };

```

CPP

上面的方法有局限性，如果题目中说能翻转k次怎么办呢，我们最好用一个通解来处理这类问题。我们可以维护一个窗口 $[left,right]$ 来容纳至少 k 个0。我们遇到了0，就累加zero的个数，然后判断如果此时0的个数大于 k ，那么我们右移左边界left，如果移除掉的 $nums[left]$ 为0，那么我们zero自减1。如果不大于 k ，那么我们用窗口中数字的个数来更新res，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int findMaxConsecutiveOnes(vector<int>& nums) {
4         int res = 0, zero = 0, left = 0, k = 1;
5         for (int right = 0; right < nums.size(); ++right) {
6             if (nums[right] == 0) ++zero;
7             while (zero > k) {
8                 if (nums[left++] == 0) --zero;
9             }
10            res = max(res, right - left + 1);
11        }
12        return res;
13    }
14 };

```

CPP

上面那种方法对于follow up中的情况无法使用，因为 $nums[left]$ 需要访问之前的数字。我们可以将遇到的0的位置全都保存下来，这样我们需要移动left的时候就知道移到哪里了：

解法3：

```

1 class Solution {
2 public:
3     int findMaxConsecutiveOnes(vector<int>& nums) {
4         int res = 0, left = 0, k = 1;
5         queue<int> q;
6         for (int right = 0; right < nums.size(); ++right) {
7             if (nums[right] == 0) q.push(right);
8             if (q.size() > k) {
9                 left = q.front() + 1; q.pop();
10            }
11            res = max(res, right - left + 1);
12        }
13        return res;
14    }
15 };

```

488. 祖玛游戏

Think about Zuma Game. You have a row of balls on the table, colored red(R), yellow(Y), blue(B), green(G), and white(W). You also have several balls in your hand.

Each time, you may choose a ball in your hand, and insert it into the row (including the leftmost place and rightmost place). Then, if there is a group of 3 or more balls in the same color touching, remove these balls. Keep doing this until no more balls can be removed.

Find the minimal balls you have to insert to remove all the balls on the table. If you cannot remove all the balls, output -1.

这道题说的就是著名的祖玛游戏了，让我想起了以前玩过的泡泡龙，也是一种祖玛游戏，在QQ上也有泡泡龙的游戏，还可以使用各种道具害其他玩家，相当有趣。那么这道题是一种简化版的祖玛游戏，只是一个一维数组，而且通过限定桌面上的球不超过20个，手里的球不超过5个来降低难度，貌似是在暗示我们可以用暴力搜索法来做。这道题比较使用递归的方法来做，通过遍历所有可能的情况来找出最优解，题目希望我们用最少的球来消掉桌上所有的球，如果不能完全消掉，返回-1。我们使用哈希表来统计手中每种球的个数，然后我们遍历桌上的球，我们找连续相同球的个数，在没有可以消除的情况下，连续的个数只能是1个或2个，然后我们用3减去连续个数，就是我们需要补充的球数以使其可以被消除，我们在哈希表中减去需要使用掉的球数，然后将消掉的球移除，对新的字符串调用递归，如果可以成功消除，会返回一个结果，该结果加上之前需要的球数用来更新结果res，注意调用完递归要恢复哈希表的状态。还有就是在刚进入递归函数时，我们要检测字符串，去除连续3个相同球的情况，这个去除函数也是个递归函数，写起来很简洁，但是很强大，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     int findMinStep(string board, string hand) {
4         int res = INT_MAX;
5         unordered_map<char, int> m;
6         for (char c : hand) ++m[c];
7         res = helper(board, m);
8         return res == INT_MAX ? -1 : res;
9     }
10    int helper(string board, unordered_map<char, int>& m) {
11        board = removeConsecutive(board);
12        if (board.empty()) return 0;
13        int cnt = INT_MAX, j = 0;
14        for (int i = 0; i <= board.size(); ++i) {
15            if (i < board.size() && board[i] == board[j]) continue;
16            int need = 3 - (i - j);
17            if (m[board[j]] >= need) {
18                m[board[j]] -= need;
19                int t = helper(board.substr(0, j) + board.substr(i), m);
20                if (t != INT_MAX) cnt = min(cnt, t + need);
21                m[board[j]] += need;
22            }
23            j = i;
24        }
25        return cnt;
26    }
27    string removeConsecutive(string board) {
28        for (int i = 0, j = 0; i <= board.size(); ++i) {
29            if (i < board.size() && board[i] == board[j]) continue;
30            if (i - j >= 3) return removeConsecutive(board.substr(0, j) + board.substr(i));
31            else j = i;
32        }
33        return board;
34    }
35 };

```

下面这种解法也是递归解法，但是思路和上面略有不同，这里我们不使用哈希表，而是使用一个集合，我们遍历手中的所有小球，如果某个小球已经在集合中存在了，说明我们已经处理过该小球了，直接跳过，否则就将该小球加入集合中。然后我们遍历桌上的小球，寻找和当前手中小球一样的位置，然后将手中小球加入当前位置，调用去除重复3个小球的函数，如果此时字符串为0了，说明当前桌上小球已经完全消掉了，返回1，因为我们此时只使用了一个小球；否则就将手中的当前小球去掉，对新的桌面和剩余手中的小球调用递归，如果得到的结果不是-1，我们用此结果加1来更新结果res，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     int findMinStep(string board, string hand) {
4         int res = INT_MAX;
5         unordered_set<char> s;
6         for (int i = 0; i < hand.size(); ++i) {
7             if (s.count(hand[i])) continue;
8             s.insert(hand[i]);
9             for (int j = 0; j < board.size(); ++j) {
10                 if (board[j] != hand[i]) continue;
11                 string newBoard = board, newHand = hand;
12                 newBoard.insert(j, 1, hand[i]);
13                 newBoard = removeConsecutive(newBoard);
14                 if (newBoard.size() == 0) return 1;
15                 newHand.erase(i, 1);
16                 int cnt = findMinStep(newBoard, newHand);
17                 if (cnt != -1) res = min(res, cnt + 1);
18             }
19         }
20         return res == INT_MAX ? -1 : res;
21     }
22     string removeConsecutive(string board) {
23         for (int i = 0, j = 0; i <= board.size(); ++i) {
24             if (i < board.size() && board[i] == board[j]) continue;
25             if (i - j >= 3) return removeConsecutive(board.substr(0, j) + board.substr(i));
26             else j = i;
27         }
28         return board;
29     }
30 };
```

489. Robot Room Cleaner

```
1 class Solution {
2 public:
3     void cleanRoom(Robot& robot) {
4         unordered_set<string> visited;
5         vector<pair<int, int>> dirs = {{-1, 0}, {0, -1}, {1, 0}, {0, 1}};
6         int d = 0; // 0: up, 1: left, 2: down, 3: right
7         dfs(robot, visited, d, 1, 3, dirs);
8     }
9     void dfs(Robot& robot, unordered_set<string>& visited, int d, int i, int j,
10    vector<pair<int, int>>& dirs) {
11        string cur_pos = to_string(i) + "," + to_string(j);
12        if (visited.count(cur_pos))
13            return;
14        visited.insert(cur_pos);
15        robot.clean();
16        for (int k = 0; k < 4; k++) {
17            if (robot.move()) {
18                int next_i = i + dirs[d].first;
19                int next_j = j + dirs[d].second;
20                dfs(robot, visited, d, next_i, next_j, dirs);
21                // go back to starting position!
22                robot.turnLeft();
23                robot.turnLeft();
24                robot.move();
25                robot.turnRight();
26                robot.turnRight();
27            } // if
28            // turn to next direction
29            robot.turnLeft();
30            d = (d + 1) % 4;
31        }
32    }
};
```

490. 迷宫

There is a ball in a maze with empty spaces and walls. The ball can go through empty spaces by rolling up, down, left or right, but it won't stop rolling until hitting a wall. When the ball stops, it could choose the next direction.

Given the ball's start position, the destination and the maze, determine whether the ball could stop at the destination.

The maze is represented by a binary 2D array. 1 means the wall and 0 means the empty space. You may assume that the borders of the maze are all walls. The start and destination coordinates are represented by row and column indexes.

Example 1

Input 1: a maze represented by a 2D array

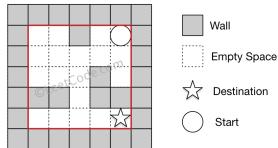
```
0 0 1 0 0
0 0 0 0 0
0 0 0 1 0
1 1 0 1 1
0 0 0 0 0
```

Input 2: start coordinate (rowStart, colStart) = (0, 4)

Input 3: destination coordinate (rowDest, colDest) = (4, 4)

Output: true

Explanation: One possible way is : left -> down -> left -> down -> right -> down -> right.



这道题让我们遍历迷宫，但是与以往不同的是，这次迷宫是有一个滚动的小球，这样就不是每次只走一步了，而是朝某一个方向一直滚，直到遇到墙或者边缘才停下来，我记得貌似之前在手机上玩过类似的游戏。那么其实还是要用DFS或者BFS来解，只不过需要做一些修改。先来看DFS的解法，我们用DFS的同时最好能用上优化，即记录中间的结果，这样可以避免重复运算，提高效率。我们用二维数组dp来保存中间结果，然后用maze数组本身通过将0改为-1来记录某个点是否被访问过，这道题的难点是在于处理一直滚的情况，其实也不难，只要我们有了方向，只要一直在那个方向上往前走，每次判读是否越界了或者是否遇到墙了即可，然后对于新位置继续调用递归函数，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<vector<int>> dirs{{0,-1},{-1,0},{0,1},{1,0}};
4     bool hasPath(vector<vector<int>>& maze, vector<int>& start, vector<int>& destination) {
5         if (maze.empty() || maze[0].empty()) return true;
6         int m = maze.size(), n = maze[0].size();
7         vector<vector<int>> dp(m, vector<int>(n, -1));
8         return helper(maze, dp, start[0], start[1], destination[0], destination[1]);
9     }
10    bool helper(vector<vector<int>>& maze, vector<vector<int>>& dp, int i, int j, int di,
11    int dj) {
12        if (i == di && j == dj) return true;
13        if (dp[i][j] != -1) return dp[i][j];
14        bool res = false;
15        int m = maze.size(), n = maze[0].size();
16        maze[i][j] = -1;
17        for (auto dir : dirs) {
18            int x = i, y = j;
19            while (x >= 0 && x < m && y >= 0 && y < n && maze[x][y] != 1) {
20                x += dir[0]; y += dir[1];
21            }
22            x -= dir[0]; y -= dir[1];
23            if (maze[x][y] != -1) {
24                res |= helper(maze, dp, x, y, di, dj);
25            }
26        }
27        dp[i][j] = res;
28        return res;
29    }
30};

```

同样的道理，对于BFS的实现需要用到队列queue，在对于一直滚的处理跟上面相同，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool hasPath(vector<vector<int>>& maze, vector<int>& start, vector<int>& destination) {
4         if (maze.empty() || maze[0].empty()) return true;
5         int m = maze.size(), n = maze[0].size();
6         vector<vector<bool>> visited(m, vector<bool>(n, false));
7         vector<vector<int>> dirs{{0,-1},{-1,0},{0,1},{1,0}};
8         queue<pair<int, int>> q;
9         q.push({start[0], start[1]});
10        visited[start[0]][start[1]] = true;
11        while (!q.empty()) {
12            auto t = q.front(); q.pop();
13            if (t.first == destination[0] && t.second == destination[1]) return true;
14            for (auto dir : dirs) {
15                int x = t.first, y = t.second;
16                while (x >= 0 && x < m && y >= 0 && y < n && maze[x][y] == 0) {
17                    x += dir[0]; y += dir[1];
18                }
19                x -= dir[0]; y -= dir[1];
20                if (!visited[x][y]) {
21                    visited[x][y] = true;
22                    q.push({x, y});
23                }
24            }
25        }
26        return false;
27    }
28};

```

491. 递增子序列

Given an integer array, your task is to find all the different possible increasing subsequences of the given array, and the length of an increasing subsequence should be at least 2 .

Example:

Input: [4, 6, 7, 7]
Output: [[4, 6], [4, 7], [4, 6, 7], [4, 6, 7, 7], [6, 7], [6, 7, 7], [7, 7], [4, 7, 7]]

这道题让我们找出所有的递增子序列，那么我们应该不难想到，这题肯定是要先找出所有的子序列，从中找出递增的。找出所有的子序列的题我们之前也接触过Subsets和Subsets II，那两题不同之处在于数组中有没有重复项。而這道题明显是有重复项的，所以需要用到Subsets II中的解法。我们首先来看一种迭代的解法，对于重复项的处理，最偷懒的方法是使用set，利用其自动去处重复项的机制，然后最后返回时再转回vector即可。由于是找递增序列，所以我们需要对递归函数做一些修改，首先题目中说明了递归序列数字至少两个，所以只有当当前子序列个数大于等于2时，才加入结果。然后就是要递增，如果之前的数字大于当前的数字，那么跳过这种情况，继续循环，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     vector<vector<int>> findSubsequences(vector<int>& nums) {
4         set<vector<int>> res;
5         vector<int> out;
6         helper(nums, 0, out, res);
7         return vector<vector<int>>(res.begin(), res.end());
8     }
9     void helper(vector<int>& nums, int start, vector<int>& out, set<vector<int>>& res) {
10        if (out.size() >= 2) res.insert(out);
11        for (int i = start; i < nums.size(); ++i) {
12            if (!out.empty() && out.back() > nums[i]) continue;
13            out.push_back(nums[i]);
14            helper(nums, i + 1, out, res);
15            out.pop_back();
16        }
17    }
18 };

```

我们也可以在递归中进行去重处理，方法是用一个set保存中间过程的数字，如果当前的数字在之前出现过了，就直接跳过这种情况即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<vector<int>> findSubsequences(vector<int>& nums) {
4         vector<vector<int>> res;
5         vector<int> out;
6         helper(nums, 0, out, res);
7         return res;
8     }
9     void helper(vector<int>& nums, int start, vector<int>& out, vector<vector<int>>& res) {
10        if (out.size() >= 2) res.push_back(out);
11        unordered_set<int> st;
12        for (int i = start; i < nums.size(); ++i) {
13            if (!out.empty() && out.back() > nums[i] || st.count(nums[i])) continue;
14            out.push_back(nums[i]);
15            st.insert(nums[i]);
16            helper(nums, i + 1, out, res);
17            out.pop_back();
18        }
19    }
20 };

```

下面我们来看迭代的解法，还是老套路，先看偷懒的方法，用set来去处重复。对于递归的处理方法跟之前相同，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     vector<vector<int>> findSubsequences(vector<int>& nums) {
4         set<vector<int>> res;
5         vector<vector<int>> cur(1);
6         for (int i = 0; i < nums.size(); ++i) {
7             int n = cur.size();
8             for (int j = 0; j < n; ++j) {
9                 if (!cur[j].empty() && cur[j].back() > nums[i]) continue;
10                cur.push_back(cur[j]);
11                cur.back().push_back(nums[i]);
12                if (cur.back().size() >= 2) res.insert(cur.back());
13            }
14        }
15        return vector<vector<int>>(res.begin(), res.end());
16    }
17 };

```

我们来看不用set的方法，使用一个哈希表来建立每个数字对应的遍历起始位置，默认都是0，然后在遍历的时候先取出原有值当作遍历起始点，然后更新为当前位置，如果某个数字之前出现过，那么取出的原有值就不是0，而是之前那个数的出现位置，这样就不会产生重复了，如果不太好理解的话就带个简单的实例去试试吧，参见代码如下：

解法4：

```

1 class Solution {
2 public:
3     vector<vector<int>> findSubsequences(vector<int>& nums) {
4         vector<vector<int>> res, cur(1);
5         unordered_map<int, int> m;
6         for (int i = 0; i < nums.size(); ++i) {
7             int n = cur.size();
8             int start = m[nums[i]];
9             m[nums[i]] = n;
10            for (int j = start; j < n; ++j) {
11                if (!cur[j].empty() && cur[j].back() > nums[i]) continue;
12                cur.push_back(cur[j]);
13                cur.back().push_back(nums[i]);
14                if (cur.back().size() >= 2) res.push_back(cur.back());
15            }
16        }
17        return res;
18    }
19 };

```

492. 构建矩形

For a web developer, it is very important to know how to design a web page's size. So, given a specific rectangular web page's area, your job by now is to design a rectangular web page, whose length L and width W satisfy the following requirements:

1. The area of the rectangular web page you designed must equal to the given target area.
 2. The width W should not be larger than the length L, which means $L \geq W$.
 3. The difference between length L and width W should be as small as possible.
- You need to output the length L and the width W of the web page you designed in sequence.

这道题让我们根据面积来求出矩形的长和宽，要求长和宽的差距尽量的小，那么就是说越接近正方形越好。那么我们肯定是先来判断一下是不是正方形，对面积开方，如果得到的不是整数，说明不是正方形。那么我们取最近的一个整数，看此时能不能整除，如果不可以，就自减1，再看能否整除。最坏的情况就是面积是质数，最后减到了1，那么返回结果即可，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     vector<int> constructRectangle(int area) {
4         int r = sqrt(area);
5         while (area % r != 0) --r;
6         return {area / r, r};
7     }
8 };
```

CPP

如果我们不想用开方运算sqrt的话，那就从1开始，看能不能整除，循环的终止条件是看平方值是否小于等于面积，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     vector<int> constructRectangle(int area) {
4         int r = 1;
5         for (int i = 1; i * i <= area; ++i) {
6             if (area % i == 0) r = i;
7         }
8         return {area / r, r};
9     }
10 };
```

CPP

493. 翻转对

Given an array `nums`, we call (i, j) an important reverse pair if $i < j$ and $\text{nums}[i] > 2 * \text{nums}[j]$.

You need to return the number of important reverse pairs in the given array.

Example1:

Input: [1,3,2,3,1]
Output: 2

这道题第一次做的时候就感觉和Count of Smaller Numbers After Self很像，开始也是用的二分查找法来建立有序数组，[LintCode](#)上也有一道非常类似的题Reverse Pairs 翻转对，虽然那道题没有加2倍的限制条件，但是思路都是一样的。于是我就开始稍稍改了一下，二分搜索的时候是搜索 $\text{nums}[i]/2.0$ ，加入数组的时候是加入原数num，我记得当时是可以通过OJ的，但是后来OJ变的严格起来了，什么二分搜索啊，BST啊之类的解法统统弄死，据目前来看，侥幸存活下来的方法就只有BIT和MergeSort这两种方法了。对于这类问题的本质的分析，大神fun4LeetCode的帖子讲的非常好，这里也借鉴一下大神的讲解来帮助理解吧。

对于这类数组找数字之间的关系的题，一种很好的解题思路就是拆分数组来解决子问题，就是把大问题拆成小问题，把小问题一一解决来，大问题的答案也就出来了，这么说起来是不是有点像DP的感觉，但是不同的是，博主感觉此类问题很难找到DP的递推公式吧。[fun4LeetCode](#)大神归纳了两种拆分方法，一种叫顺序重现关系(Sequential Recurrence Relation)，用式子表示是 $T(i, j) = T(i, j - 1) + C$ 。这里的C就是处理最后一个数字的子问题，那么用文字来描述就是“已知翻转对的第二个数字为 $\text{nums}[j]$ ，在子数组 $\text{nums}[i, j - 1]$ 中找翻转对的第一个数字”，这里翻转对的两个条件中的顺序条件已经满足，就只需要找比 $2 * \text{nums}[j]$ 大的数

即可。当然最森破的方法就是线性扫描，但这样整个时间复杂度就会上升到令人发指的 $O(n^2)$ ，这又怎么能逃过连Binary Search都不放过的OJ的魔爪。由于二分搜索和BST等方法已经被OJ阉割，所以我们只能用树状数组Binary Indexed Tree来做了。关于BIT，我之前有篇博客Range Sum Query - Mutable 应该讲的比较清楚了，如果弄懂了那篇博客，我们对BIT的机制也应该有个基本的了解，由于BIT的存储方式不是将数组中的数字对应的一一存入，而是有的对应存入，有的是存若干个数字之和，其设计初衷之一就是要在 $O(lgn)$ 的时间复杂度下完成求和运算。那么我们该如何利用这一特性呢，这跟这道题又有什么关系呢，别着急，博主会慢慢解释。

首先我们应该确定一个遍历的方向，这里博主推荐从后往前遍历数组，这样做的好处是对于当前遍历到的数字，在已遍历过的数字中找小于当前数字的一半($nums[i]/2.0$)的数字，这样的遍历方向也能跟上面的顺序重现关系的定义式统一起来。当然如果你想强行从前往后遍历，也不是不行，那么就需要在已遍历的数字中找大于当前数字的二倍($nums[i]*2$)的数字就行了。由于我们要在之前遍历过的数字中找符合条件的数字，怎么样利用BIT的特性来快速的找到是这种解法的最大难点。我们需要将之前遍历过的数字存入BIT中，怎么存是难点。由于之前那篇博客我们知道BIT用update函数来存数，需要提供要存入的位置和要存入的数字这两个参数，那么这里难道我们就按照数字在原数组中的位置存入BIT吗，这样做毫无意义！我们要存的是该数字在有序数组中的位置，而且存入的也不是该数字本身，而是该数字出现的次数1。我们用题目中的第一个例子来说明，我们先给数组排序，得到：

1 1 2 3 3

对于每一个数字我们要确定其在BIT中的位置，由于有重复数字的存在，那么每个数字对应的位置就是其最后出现的位置，而且因为BIT是从1开始的，并不是像一般的数组那样从0开始，那么有如下对应关系：

1→2, 2→3, 3→5

那么当我们遇到数字1了，就update(2,1)，遇到数字2了，就update(3,1)，遇到数字3了，就update(5,1)。我们之前解释了并不把数字本身存入BIT，而是将其对应的位置存入BIT，真正存入的数字是1，这样方便累加，而且由于1是固定的，在下面的代码中就不用将1当作函数的参数了。这样我们知道了如果存入数字，那么我们在遍历到新数字时，为了得到符合要求的数字的个数，需利用BIT的getSum函数。getSum函数需要提供一个位置参数，可以返回该位置之前的所有数之和。同理，我们提供的参数既不是当前遍历到的数字本身，也不是其在原数组中的位置，而是该数字的一半($nums[i]/2.0$)在有序数组中的正确位置，可以用lower_bound函数来找第一个不小于目标值的位置，当然我们也可以自己写个二分搜索的子函数来代替lower_bound函数。比如我们当前遍历到的数字是3，那么我们在有序数组中找1.5的位置，返回是2，此时我们在BIT中用getSum来返回位置2之前的数字之和，返回几就表示有几个小于1.5的数字。讲到这里基本上这种解法的核心内容都讲完了，如果你还是一头雾水，那么就是博主的表述能力的问题了(沮丧脸:O。那么博主只能建议你带实例一步一步去试，看看每一步操作后BIT中的结果是啥，下面就列出这些内容：

update(2,1) -> BIT: 0 0 1 0 1 0

update(5,1) -> BIT: 0 0 1 0 1 1

update(3,1) -> BIT: 0 0 1 1 2 1

update(5,1) -> BIT: 0 0 1 1 2 2

update(2,1) -> BIT: 0 0 2 1 3 2

解法1:

```

1 class Solution {
2 public:
3     int reversePairs(vector<int>& nums) {
4         int res = 0, n = nums.size();
5         vector<int> v = nums, bit(n + 1);
6         sort(v.begin(), v.end());
7         unordered_map<int, int> m;
8         for (int i = 0; i < n; ++i) m[v[i]] = i + 1;
9         for (int i = n - 1; i >= 0; --i) {
10             res += getSum(lower_bound(v.begin(), v.end(), nums[i] / 2.0) - v.begin(), bit);
11             update(m[nums[i]], bit);
12         }
13         return res;
14     }
15     int getSum(int i, vector<int>& bit) {
16         int sum = 0;
17         while (i > 0) {
18             sum += bit[i];
19             i -= (i & -i);
20         }
21         return sum;
22     }
23     void update(int i, vector<int>& bit) {
24         while (i < bit.size()) {
25             bit[i] += 1;
26             i += (i & -i);
27         }
28     }
29 };

```

fun4LeetCode大神归纳的第二种方法叫做分割重现关系(Partition Recurrence Relation)，用式子表示是 $T(i, j) = T(i, m) + T(m+1, j) + C$ 。这里的C就是处理合并两个部分的子问题，那么用文字来描述就是“已知翻转对的两个数字分别在子数组 $nums[i, m]$ 和 $nums[m+1, j]$ 之中，求满足要求的翻转对的个数”，这里翻转对的两个条件中的顺序条件已经满足，就只需要找到满足大小关系的数对即可。这里两个数字都是不确定的，如果用暴力搜索肯定会被OJ唾弃。但是如果两个子数组是有序的，那么我们可以用双指针的方法在线性时间内就可以统计出符合题意的翻转对的个数。要想办法产生有序的子数组，那么这就和MergeSort的核心思想完美匹配了。我们知道混合排序就是不断的将数组对半拆分成子数组，拆到最小的数组后开始排序，然后一层一层的返回，最后原数组也是有序的了。这里我们在混合排序的递归函数中，对于有序的两个子数组进行统计翻转对的个数，然后再逐层返回，这就完美的实现了上述的分割重现关系的思想。整个的写法非常的简洁，实在是太叼了。博主的直觉表明，fun4LeetCode大神肯定是国人，不要问我为什么，因为这么强的肯定是中国，哈~

解法2：

```

1 class Solution {
2 public:
3     int reversePairs(vector<int>& nums) {
4         return mergeSort(nums, 0, nums.size() - 1);
5     }
6     int mergeSort(vector<int>& nums, int left, int right) {
7         if (left >= right) return 0;
8         int mid = left + (right - left) / 2;
9         int res = mergeSort(nums, left, mid) + mergeSort(nums, mid + 1, right);
10        for (int i = left, j = mid + 1; i <= mid; ++i) {
11            while (j <= right && nums[i] / 2.0 > nums[j]) ++j;
12            res += j - (mid + 1);
13        }
14        sort(nums.begin() + left, nums.begin() + right + 1);
15        return res;
16    }
17 };

```

494. 目标和

You are given a list of non-negative integers, a_1, a_2, \dots, a_n , and a target, S . Now you have 2 symbols + and -. For each integer, you should choose one from + and - as its new symbol.

Find out how many ways to assign symbols to make sum of integers equal to target S .

这道题给了我们一个数组，和一个目标值，让我们给数组中每个数字加上正号或负号，然后求和要和目标值相等，求有多少种不同的情况。那么对于这种求多种情况的问题，我最想到的方法使用递归来做。我们从第一个数字，调用递归函数，在递归函数中，分别对目标值进行加上当前数字调用递归，和减去当前数字调用递归，这样会涵盖所有情况，并且当所有数字遍历完成后，我们看若目标值为0了，则结果res自增1，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     int findTargetSumWays(vector<int>& nums, int S) {
4         int res = 0;
5         helper(nums, S, 0, res);
6         return res;
7     }
8     void helper(vector<int>& nums, int S, int start, int& res) {
9         if (start >= nums.size()) {
10             if (S == 0) ++res;
11             return;
12         }
13         helper(nums, S - nums[start], start + 1, res);
14         helper(nums, S + nums[start], start + 1, res);
15     }
16 };

```

我们对上面的递归方法进行优化，使用dp数组来记录中间值，这样可以避免重复运算，参见代码如下：

解法2:

```

1 class Solution {
2 public:
3     int findTargetSumWays(vector<int>& nums, int S) {
4         vector<unordered_map<int, int>> dp(nums.size());
5         return helper(nums, S, 0, dp);
6     }
7     int helper(vector<int>& nums, int sum, int start, vector<unordered_map<int, int>>& dp)
8     {
9         if (start == nums.size()) return sum == 0;
10        if (dp[start].count(sum)) return dp[start][sum];
11        int cnt1 = helper(nums, sum - nums[start], start + 1, dp);
12        int cnt2 = helper(nums, sum + nums[start], start + 1, dp);
13        return dp[start][sum] = cnt1 + cnt2;
14    }
15 }

```

我们也可以使用迭代的方法来解，还是要用dp数组，其中 $dp[i][j]$ 表示到第*i*-1个数字且和为*j*的情况总数，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int findTargetSumWays(vector<int>& nums, int S) {
4         int n = nums.size();
5         vector<unordered_map<int, int>> dp(n + 1);
6         dp[0][0] = 1;
7         for (int i = 0; i < n; ++i) {
8             for (auto &a : dp[i]) {
9                 int sum = a.first, cnt = a.second;
10                dp[i + 1][sum + nums[i]] += cnt;
11                dp[i + 1][sum - nums[i]] += cnt;
12            }
13        }
14        return dp[n][S];
15    }
16 }

```

我们也可以对上面的方法进行空间上的优化，只用一个哈希表，而不是用一个数组的哈希表，我们在遍历数组中的每一个数字时，新建一个哈希表，我们在遍历原哈希表中的项时更新这个新建的哈希表，最后把新建的哈希表整个赋值和原哈希表，参见代码如下：

解法4：

```

1 class Solution {
2 public:
3     int findTargetSumWays(vector<int>& nums, int S) {
4         unordered_map<int, int> dp;
5         dp[0] = 1;
6         for (int num : nums) {
7             unordered_map<int, int> t;
8             for (auto a : dp) {
9                 int sum = a.first, cnt = a.second;
10                t[sum + num] += cnt;
11                t[sum - num] += cnt;
12            }
13            dp = t;
14        }
15        return dp[S];
16    }
17 };

```

495. 提莫攻击

In LLP world, there is a hero called Teemo and his attacking can make his enemy Ashe be in poisoned condition. Now, given the Teemo's attacking ascending time series towards Ashe and the poisoning time duration per Teemo's attacking, you need to output the total time that Ashe is in poisoned condition.

You may assume that Teemo attacks at the very beginning of a specific time point, and makes Ashe be in poisoned condition immediately.

Example 1:

Input: [1,4], 2

Output: 4

Explanation: At time point 1, Teemo starts attacking Ashe and makes Ashe be poisoned immediately.

This poisoned status will last 2 seconds until the end of time point 2.

And at time point 4, Teemo attacks Ashe again, and causes Ashe to be in poisoned status for another 2 seconds.

So you finally need to output 4.

LeetCode果然花样百出，连提莫都搬上题目了，那个草丛里乱种蘑菇的小提莫，那个“团战可以输提莫必须死”的提莫？？可以，服了，坐等女枪女警轮子妈的题目了~好了，不闲扯了，其实这道题蛮简单的，感觉不能算一道medium的题，就直接使用贪心算法，比较相邻两个时间点的时间差，如果小于duration，就加上这个差，如果大于或等于，就加上duration即可，参见代码如下：

```

1 class Solution {
2 public:
3     int findPoisonedDuration(vector<int>& timeSeries, int duration) {
4         if (timeSeries.empty()) return 0;
5         int res = 0, n = timeSeries.size();
6         for (int i = 1; i < n; ++i) {
7             int diff = timeSeries[i] - timeSeries[i - 1];
8             res += (diff < duration) ? diff : duration;
9         }
10        return res + duration;
11    }
12 };

```

496. 下一个较大的元素之一

You are given two arrays (without duplicates) `nums1` and `nums2` where `nums1`'s elements are subset of `nums2`. Find all the next greater numbers for `nums1`'s elements in the corresponding places of `nums2`.

The Next Greater Number of a number `x` in `nums1` is the first greater number to its right in `nums2`. If it does not exist, output -1 for this number.

这道题给了我们一个数组，又给了该数组的一个子集合，让我们求集合中每个数字在原数组中右边第一个较大的数字。参考题目中给的例子，题意不难理解，既然这次难度标识为Easy，想必不是一道太难的题。二话不说，先上无脑暴力搜索，遍历子集合中的每一个数字，然后在原数组中找到这个数字，然后向右遍历，找到第一个大于该数字的数即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<int> nextGreaterElement(vector<int>& findNums, vector<int>& nums) {
4         vector<int> res(findNums.size());
5         for (int i = 0; i < findNums.size(); ++i) {
6             int j = 0, k = 0;
7             for (; j < nums.size(); ++j) {
8                 if (nums[j] == findNums[i]) break;
9             }
10            for (k = j + 1; k < nums.size(); ++k) {
11                if (nums[k] > nums[j]) {
12                    res[i] = nums[k];
13                    break;
14                }
15            }
16            if (k == nums.size()) res[i] = -1;
17        }
18        return res;
19    }
20 };

```

CPP

我们来对上面的方法稍做优化，我们用哈希表先来建立每个数字和其坐标位置之间的映射，那么我们在遍历子集合中的数字时，就能直接定位到该数字在原数组中的位置，然后再往右边遍历寻找较大数即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<int> nextGreaterElement(vector<int>& findNums, vector<int>& nums) {
4         vector<int> res(findNums.size());
5         unordered_map<int, int> m;
6         for (int i = 0; i < nums.size(); ++i) {
7             m[nums[i]] = i;
8         }
9         for (int i = 0; i < findNums.size(); ++i) {
10            res[i] = -1;
11            int start = m[findNums[i]];
12            for (int j = start + 1; j < nums.size(); ++j) {
13                if (nums[j] > findNums[i]) {
14                    res[i] = nums[j];
15                    break;
16                }
17            }
18        }
19        return res;
20    }
21 };

```

下面这种方法使用了哈希表和栈，但是这里的哈希表和上面的不一样，这里是建立每个数字和其右边第一个较大数之间的映射，没有的话就是-1。我们遍历原数组中的所有数字，如果此时栈不为空，且栈顶元素小于当前数字，说明当前数字就是栈顶元素的右边第一个较大数，那么建立二者的映射，并且去除当前栈顶元素，最后将当前遍历到的数字压入栈。当所有数字都建立了映射，那么最后我们可以直接通过哈希表快速的找到子集合中数字的右边较大值，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     vector<int> nextGreaterElement(vector<int>& findNums, vector<int>& nums) {
4         vector<int> res;
5         stack<int> st;
6         unordered_map<int, int> m;
7         for (int num : nums) {
8             while (!st.empty() && st.top() < num) {
9                 m[st.top()] = num; st.pop();
10            }
11            st.push(num);
12        }
13        for (int num : findNums) {
14            res.push_back(m.count(num) ? m[num] : -1);
15        }
16        return res;
17    }
18 };

```

497. 非重叠矩形中的随机点

给定一个非重叠轴对齐矩形的列表 `rects`，写一个函数 `pick` 随机均匀地选取矩形覆盖的空间中的整数点。

```

1 class Solution {
2 public:
3     vector<int> v;
4     vector<vector<int>> rects;
5
6     int area(vector<int>& r) {
7         return (r[2] - r[0] + 1) * (r[3] - r[1] + 1);
8     }
9
10    Solution(vector<vector<int>> _) {
11        rects = _;
12        for (auto& r : rects) {
13            v.push_back(area(r) + (v.empty() ? 0 : v.back()));
14        }
15    }
16
17    vector<int> pick() {
18        int rnd = rand() % v.back();
19        auto it = upper_bound(v.begin(), v.end(), rnd);
20        int idx = it - v.begin();
21
22        // pick a random point in rect[idx]
23        auto r = rects[idx];
24        return {
25            rand() % (r[2] - r[0] + 1) + r[0],
26            rand() % (r[3] - r[1] + 1) + r[1]
27        };
28    }
29}

```

498. 对角线遍历

Given a matrix of $M \times N$ elements (M rows, N columns), return all elements of the matrix in diagonal order as shown in the below image.

这道题给了我们一个 $m \times n$ 大小的数组，让我们进行对角线遍历，先向右上，然后左下，再右上，以此类推直至遍历完整个数组。题目中的例子和图示也能很好的帮我们理解。由于移动的方向不再是水平或竖直方向，而是对角线方向，那么每移动一次，横纵坐标都要变化，向右上移动的话要坐标加上 $[1, 1]$ ，向左下移动的话要坐标加上 $[1, -1]$ ，那么难点在于我们如何处理越界情况，越界后遍历的方向怎么变换。向右上和左下两个对角线方向遍历的时候都会有越界的可能，但是除了左下角和右上角的位置越界需要改变两个坐标之外，其余的越界只需要改变一个。那么我们就先判断要同时改变两个坐标的越界情况，即在右上角和左下角的位置。如果在右上角位置还要往右上走时，那么要移动到它下面的位置的，那么如果 col 超过了 $n-1$ 的范围，那么 col 重置为 $n-1$ ，并且 row 自增2，然后改变遍历的方向。同理如果 row 超过了 $m-1$ 的范围，那么 row 重置为 $m-1$ ，并且 col 自增2，然后改变遍历的方向。然后我们再来判断一般的越界情况，如果 row 小于0，那么 row 重置0，然后改变遍历的方向。同理如果 col 小于0，那么 col 重置0，然后改变遍历的方向。参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     vector<int> findDiagonalOrder(vector<vector<int>>& matrix) {
4         if (matrix.empty() || matrix[0].empty()) return {};
5         int m = matrix.size(), n = matrix[0].size(), r = 0, c = 0, k = 0;
6         vector<int> res(m * n);
7         vector<vector<int>> dirs{{{-1, 1}, {1, -1}}};
8         for (int i = 0; i < m * n; ++i) {
9             res[i] = matrix[r][c];
10            r += dirs[k][0];
11            c += dirs[k][1];
12            if (r >= m) {r = m - 1; c += 2; k = 1 - k;}
13            if (c >= n) {c = n - 1; r += 2; k = 1 - k;}
14            if (r < 0) {r = 0; k = 1 - k;}
15            if (c < 0) {c = 0; k = 1 - k;}
16        }
17        return res;
18    }
19 };

```

下面这种方法跟上面的方法思路相同，不过写法有些不同，这里根据横纵左边之和的奇偶性来判断遍历的方向，然后对于越界情况再单独处理即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<int> findDiagonalOrder(vector<vector<int>>& matrix) {
4         if (matrix.empty() || matrix[0].empty()) return {};
5         int m = matrix.size(), n = matrix[0].size(), r = 0, c = 0;
6         vector<int> res(m * n);
7         for (int i = 0; i < m * n; ++i) {
8             res[i] = matrix[r][c];
9             if ((r + c) % 2 == 0) {
10                 if (c == n - 1) {++r;}
11                 else if (r == 0) {++c;}
12                 else {--r; ++c;}
13             } else {
14                 if (r == m - 1) {++c;}
15                 else if (c == 0) {++r;}
16                 else {++r; --c;}
17             }
18         }
19         return res;
20     }
21 };

```

下面这种方法是按遍历方向来按规律往结果res中添加数字的，比如题目中的那个例子，那么添加的顺序如下：

[0,0] -> [0,1],[1,0] -> [2,0],[1,1],[0,2] -> [1,2],[2,1] -> [2,2]

根据遍历的方向不同共分为五层，关键就是确定每一层的坐标范围，其中下边界 $low = \max(0, i - n + 1)$ ，这样可以保证下边界不会小于0，而上边界 $high = \min(i, m - 1)$ ，这样也保证了上边界不会大于m-1，如果是偶数层，则从上边界往下边界遍历，反之如果是奇数层，则从下边界往上边界遍历，注意从matrix中取数字的坐标，，参见代码如下：

解法3:

```

1 class Solution {
2 public:
3     vector<int> findDiagonalOrder(vector<vector<int>>& matrix) {
4         if (matrix.empty() || matrix[0].empty()) return {};
5         int m = matrix.size(), n = matrix[0].size(), k = 0;
6         vector<int> res(m * n);
7         for (int i = 0; i < m + n - 1; ++i) {
8             int low = max(0, i - n + 1), high = min(i, m - 1);
9             if (i % 2 == 0) {
10                 for (int j = high; j >= low; --j) {
11                     res[k++] = matrix[j][i - j];
12                 }
13             } else {
14                 for (int j = low; j <= high; ++j) {
15                     res[k++] = matrix[j][i - j];
16                 }
17             }
18         }
19         return res;
20     }
21 };

```

CPP

下面这种方法就有一点暴力搜索的感觉，不像上面一种精确计算每一层的坐标范围，这种方法是利用对角线上的数字的横纵坐标之和恒定这一特性来搜索的，然后把和为特定值的数字加入结果res中，参见代码如下：

解法4:

```

1 class Solution {
2 public:
3     vector<int> findDiagonalOrder(vector<vector<int>>& matrix) {
4         if (matrix.empty() || matrix[0].empty()) return {};
5         int m = matrix.size(), n = matrix[0].size(), k = 0;
6         vector<int> res;
7         for (int k = 0; k < m + n - 1; ++k) {
8             int delta = 1 - 2 * (k % 2 == 0);
9             int ii = (m - 1) * (k % 2 == 0);
10            int jj = (n - 1) * (k % 2 == 0);
11            for (int i = ii; i >= 0 && i < m; i += delta) {
12                for (int j = jj; j >= 0 && j < n; j += delta) {
13                    if (i + j == k) {
14                        res.push_back(matrix[i][j]);
15                    }
16                }
17            }
18        }
19        return res;
20    }
21 };

```

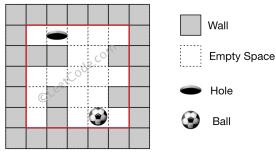
CPP

499. 迷宫之三

There is a ball in a maze with empty spaces and walls. The ball can go through empty spaces by rolling up (u), down (d), left (l) or right (r), but it won't stop rolling until hitting a wall. When the ball stops, it could choose the next direction. There is also a hole in this maze. The ball will drop into the hole if it rolls on to the hole.

Given the ball position, the hole position and the maze, find out how the ball could drop into the hole by moving the shortest distance. The distance is defined by the number of empty spaces traveled by the ball from the start position (excluded) to the hole (included). Output the moving directions by using 'u', 'd', 'l' and 'r'. Since there could be several different shortest ways, you should output the lexicographically smallest way. If the ball cannot reach the hole, output "impossible".

The maze is represented by a binary 2D array. 1 means the wall and 0 means the empty space. You may assume that the borders of the maze are all walls. The ball and the hole coordinates are represented by row and column indexes.



这道题在之前的两道The Maze II和The Maze的基础上又做了些改变，在路径中间放了个陷阱，让球在最小步数内滚到陷阱之中，此时返回的并不是最小步数，而是滚动的方向，用u, r, d, l这四个字母来分别表示上右下左，而且在步数相等的情况下，让我们返回按字母排序小的答案。相对于迷宫二那题来说，难度是增加了一些，但我们还是可以借鉴之前那道题的思路，我们还是需要用一个二维数组dists，其中dists[i][j]表示到达(i,j)这个位置时需要的最小步数，我们都初始化为整型最大值，在后在遍历的过程中不断用较小值来更新每个位置的步数值。我们还需要用一个哈希表来建立每个位置跟滚到该位置的方向字符串之间的映射，这里我们用一个trick，将二维坐标转(i,j)为一个数字i*n+j，这实际上就是把二维数组拉成一维数组的操作，matlab中很常见的操作。还有需要注意的是，一滚到底的操作需要稍作修改，之前我们都是一直滚到墙里面或者界外才停止，然后做退一步处理，就是小球能滚到的位置，这里我们滚的时候要判断陷阱，如果滚到了陷阱，那么我们也停下来，注意这时候不需要做后退一步处理。然后我们还是比较当前步数是否小于dists中的原有步数，小于的话就更新dists，然后更新哈希表中的映射方向字符串，然后对于不是陷阱的点，我们加入队列queue中继续滚。另一点跟迷宫二不同的之处在于，这里还要处理另一种情况，就是当最小步数相等的时候，并且新的滚法的方向字符串的字母顺序要小于原有的字符串的时候，我们也需要更新哈希表的映射，并且判断是否需要加入队列queue中，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     string findShortestWay(vector<vector<int>>& maze, vector<int>& ball, vector<int>& hole)
4     {
5         int m = maze.size(), n = maze[0].size();
6         vector<vector<int>> dists(m, vector<int>(n, INT_MAX));
7         vector<vector<int>> dirs{{0,-1},{-1,0},{0,1},{1,0}};
8         vector<char> way{'l','u','r','d'};
9         queue<pair<int, int>> q;
10        unordered_map<int, string> u;
11        dists[ball[0]][ball[1]] = 0;
12        q.push({ball[0], ball[1]});
13        while (!q.empty()) {
14            auto t = q.front(); q.pop();
15            for (int i = 0; i < 4; ++i) {
16                int x = t.first, y = t.second, dist = dists[x][y];
17                string path = u[x * n + y];
18                while (x >= 0 && x < m && y >= 0 && y < n && maze[x][y] == 0 && (x != hole[0] || y != hole[1])) {
19                    x += dirs[i][0]; y += dirs[i][1]; ++dist;
20                }
21                if (x != hole[0] || y != hole[1]) {
22                    x -= dirs[i][0]; y -= dirs[i][1]; --dist;
23                }
24                path.push_back(way[i]);
25                if (dists[x][y] > dist) {
26                    dists[x][y] = dist;
27                    u[x * n + y] = path;
28                    if (x != hole[0] || y != hole[1]) q.push({x, y});
29                } else if (dists[x][y] == dist && u[x * n + y].compare(path) > 0) {
30                    u[x * n + y] = path;
31                    if (x != hole[0] || y != hole[1]) q.push({x, y});
32                }
33            }
34        }
35        string res = u[hole[0] * n + hole[1]];
36        return res.empty() ? "impossible" : res;
37    }
};

```

下面这种写法是DFS的解法，可以看出来思路基本上跟上面的解法没有啥区别，写法上稍有不同，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<vector<int>> dirs{{0,-1},{-1,0},{0,1},{1,0}};
4     vector<char> way{'l','u','r','d'};
5     string findShortestWay(vector<vector<int>>& maze, vector<int>& ball, vector<int>& hole)
6     {
7         int m = maze.size(), n = maze[0].size();
8         vector<vector<int>> dists(m, vector<int>(n, INT_MAX));
9         unordered_map<int, string> u;
10        dists[ball[0]][ball[1]] = 0;
11        helper(maze, ball[0], ball[1], hole, dists, u);
12        string res = u[hole[0] * n + hole[1]];
13        return res.empty() ? "impossible" : res;
14    }
15    void helper(vector<vector<int>>& maze, int i, int j, vector<int>& hole,
16    vector<vector<int>>& dists, unordered_map<int, string>& u) {
17        if (i == hole[0] && j == hole[1]) return;
18        int m = maze.size(), n = maze[0].size();
19        for (int k = 0; k < 4; ++k) {
20            int x = i, y = j, dist = dists[x][y];
21            string path = u[x * n + y];
22            while (x >= 0 && x < m && y >= 0 && y < n && maze[x][y] == 0 && (x != hole[0]
23 || y != hole[1])) {
24                x += dirs[k][0]; y += dirs[k][1]; ++dist;
25            }
26            if (x != hole[0] || y != hole[1]) {
27                x -= dirs[k][0]; y -= dirs[k][1]; --dist;
28            }
29            path.push_back(way[k]);
30            if (dists[x][y] > dist) {
31                dists[x][y] = dist;
32                u[x * n + y] = path;
33                helper(maze, x, y, hole, dists, u);
34            } else if (dists[x][y] == dist && u[x * n + y].compare(path) > 0) {
35                u[x * n + y] = path;
36                helper(maze, x, y, hole, dists, u);
37            }
38        }
39    }
40 };

```

500. 键盘行

Given a List of words, return the words that can be typed using letters of alphabet on only one row's of American keyboard like the image below.

这道题给了我们一些单词，问哪些单词可以由键盘上的一行中的键符打出来，难度其实并不大。首先我们把键盘的三行字符分别保存到三个set中，然后遍历每个单词中的每个字符，分别看当前字符是否在三个集合中，如果在，对应的标识变量变为1，我们统计三个标识变量之和就知道有几个集合参与其中了，参见代码如下：

```

1 class Solution {
2 public:
3     vector<string> findWords(vector<string>& words) {
4         vector<string> res;
5         unordered_set<char> row1{'q','w','e','r','t','y','u','i','o','p'};
6         unordered_set<char> row2{'a','s','d','f','g','h','j','k','l'};
7         unordered_set<char> row3{'z','x','c','v','b','n','m'};
8         for (string word : words) {
9             int one = 0, two = 0, three = 0;
10            for (char c : word) {
11                if (c < 'a') c += 32;
12                if (row1.count(c)) one = 1;
13                if (row2.count(c)) two = 1;
14                if (row3.count(c)) three = 1;
15                if (one + two + three > 1) break;
16            }
17            if (one + two + three == 1) res.push_back(word);
18        }
19        return res;
20    }
21 };

```

501. 找二分搜索数的众数

Given a binary search tree (BST) with duplicates, find all the mode(s) (the most frequently occurred element) in the given BST.

Assume a BST is defined as follows:

The left subtree of a node contains only nodes with keys less than or equal to the node's key.
The right subtree of a node contains only nodes with keys greater than or equal to the node's key.

Both the left and right subtrees must also be binary search trees.

这道题让我们求二分搜索树中的众数，这里定义的二分搜索树中左根右结点之间的关系是小于等于的，有些题目中是严格小于的，所以一定要看清题目要求。所谓的众数就是出现次数最多的数字，可以有多个，那么这道题比较直接的思路就是利用一个哈希表来记录数字和其出现次数之前的映射，然后维护一个变量mx来记录当前最多的次数值，这样在遍历完树之后，根据这个mx值就能把对应的元素找出来。那么用这种方法的话就不需要用到二分搜索树的性质了，随意一种遍历方式都可以，下面我们来看递归的中序遍历的解法如下：

解法1：

```

1 class Solution {
2 public:
3     vector<int> findMode(TreeNode* root) {
4         vector<int> res;
5         int mx = 0;
6         unordered_map<int, int> m;
7         inorder(root, m, mx);
8         for (auto a : m) {
9             if (a.second == mx) {
10                 res.push_back(a.first);
11             }
12         }
13         return res;
14     }
15     void inorder(TreeNode* node, unordered_map<int, int>& m, int& mx) {
16         if (!node) return;
17         inorder(node->left, m, mx);
18         mx = max(mx, ++m[node->val]);
19         inorder(node->right, m, mx);
20     }
21 };

```

下面这种解法是上面方法的迭代形式，也是用的中序遍历的方法，有兴趣的童鞋可以实现其他的遍历方法：

解法2：

```

1 class Solution {
2 public:
3     vector<int> findMode(TreeNode* root) {
4         if (!root) return {};
5         vector<int> res;
6         TreeNode *p = root;
7         stack<TreeNode*> s;
8         unordered_map<int, int> m;
9         int mx = 0;
10        while (!s.empty() || p) {
11            while (p) {
12                s.push(p);
13                p = p->left;
14            }
15            p = s.top(); s.pop();
16            mx = max(mx, ++m[p->val]);
17            p = p->right;
18        }
19        for (auto a : m) {
20            if (a.second == mx) {
21                res.push_back(a.first);
22            }
23        }
24        return res;
25    }
26};

```

题目中的follow up说了让我们不用除了递归中的隐含栈之外的额外空间，那么我们就不能用哈希表了，不过这也不难，由于是二分搜索树，那么我们中序遍历出来的结果就是有序的，这样我们只要比较前后两个元素是否相等，就等统计出现某个元素出现的次数，因为相同的元素肯定是都在一起的。我们需要一个结点变量pre来记录上一个遍历到的结点，然后mx还是记录最大的次数，cnt来计数当前元素出现的个数，我们在中序遍历的时候，如果pre不为空，说明当前不是第一个结点，我们和之前一个结点

值比较，如果相等，cnt自增1，如果不等，cnt重置1。如果此时cnt大于了mx，那么我们清空结果res，并把当前结点值加入结果res，如果cnt等于mx，那我们直接将当前结点值加入结果res，然后mx赋值为cnt。最后我们要把pre更新为当前结点，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     vector<int> findMode(TreeNode* root) {
4         vector<int> res;
5         int mx = 0, cnt = 1;
6         TreeNode *pre = NULL;
7         inorder(root, pre, cnt, mx, res);
8         return res;
9     }
10    void inorder(TreeNode* node, TreeNode*& pre, int& cnt, int& mx, vector<int>& res) {
11        if (!node) return;
12        inorder(node->left, pre, cnt, mx, res);
13        if (pre) {
14            cnt = (node->val == pre->val) ? cnt + 1 : 1;
15        }
16        if (cnt >= mx) {
17            if (cnt > mx) res.clear();
18            res.push_back(node->val);
19            mx = cnt;
20        }
21        pre = node;
22        inorder(node->right, pre, cnt, mx, res);
23    }
24 };

```

下面这种方法是上面解法的迭代写法，思路基本相同，可以参考上面的讲解，参见代码如下：

解法4：

```

1 class Solution {
2 public:
3     vector<int> findMode(TreeNode* root) {
4         if (!root) return {};
5         vector<int> res;
6         TreeNode *p = root, *pre = NULL;
7         stack<TreeNode*> s;
8         int mx = 0, cnt = 1;
9         while (!s.empty() || p) {
10             while (p) {
11                 s.push(p);
12                 p = p->left;
13             }
14             p = s.top(); s.pop();
15             if (pre) {
16                 cnt = (p->val == pre->val) ? cnt + 1 : 1;
17             }
18             if (cnt >= mx) {
19                 if (cnt > mx) res.clear();
20                 res.push_back(p->val);
21                 mx = cnt;
22             }
23             pre = p;
24             p = p->right;
25         }
26         return res;
27     }
28 };

```

502. IPO 上市

Suppose LeetCode will start its IPO soon. In order to sell a good price of its shares to Venture Capital, LeetCode would like to work on some projects to increase its capital before the IPO. Since it has limited resources, it can only finish at most k distinct projects before the IPO. Help LeetCode design the best way to maximize its total capital after finishing at most k distinct projects.

You are given several projects. For each project i, it has a pure profit P_i and a minimum capital of C_i is needed to start the corresponding project. Initially, you have W capital. When you finish a project, you will obtain its pure profit and the profit will be added to your total capital.

To sum up, pick a list of at most k distinct projects from given projects to maximize your final capital, and output your final maximized capital.

这道题上来就让人眼前一亮，剑指上市，每个创业公司都有一个上市的梦想吧，博主认为照现在这种发展趋势，感觉上市并非遥不可及，资瓷一下。这道题说初始时我们的资本为0，可以交易k次，并且给了我们提供了交易所需的资本和所能获得的利润，让我们求怎样选择k次交易，使我们最终的资本最大。虽然题目中给我们的资本数组是有序的，但是OJ里的test case肯定不都是有序的，还有就是不一定需要资本大的交易利润就多，该遍历的时候还得遍历。我们可以用贪婪算法来解，每一次都选择资本范围内最大利润的进行交易，那么我们首先应该建立资本和利润对，然后根据资本的大小进行排序，然后我们根据自己当前的资本，用二分搜索法在有序数组中找第一个大于当前资本的交易的位置，然后往前退一步就是最后一个不大于当前资本的交易，然后向前遍历，找到利润最大的那个的进行交易，把利润加入资本W中，然后将这个交易对删除，这样我们就可以保证在进行k次交易后，我们的总资本最大，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     int findMaximizedCapital(int k, int W, vector<int>& Profits, vector<int>& Capital) {
4         vector<pair<int, int>> v;
5         for (int i = 0; i < Capital.size(); ++i) {
6             v.push_back({Capital[i], Profits[i]});
7         }
8         sort(v.begin(), v.end());
9         for (int i = 0; i < k; ++i) {
10             int left = 0, right = v.size(), mx = 0, idx = 0;
11             while (left < right) {
12                 int mid = left + (right - left) / 2;
13                 if (v[mid].first <= W) left = mid + 1;
14                 else right = mid;
15             }
16             for (int j = right - 1; j >= 0; --j) {
17                 if (mx < v[j].second) {
18                     mx = v[j].second;
19                     idx = j;
20                 }
21             }
22             W += mx;
23             v.erase(v.begin() + idx);
24         }
25         return W;
26     }
27 };

```

看论坛上的大神们都比较喜欢用一些可以自动排序的数据结构来做，比如我们可以使用一个最大堆和一个最小堆，把资本利润对放在最小堆中，这样需要资本小的交易就在队首，然后从队首按顺序取出资本小的交易，如果所需资本不大于当前所拥有的资本，那么就把利润资本存入最大堆中，注意这里资本和利润要翻个，因为我们希望把利润最大的交易放在队首，便于取出，这样也能实现我们的目的，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int findMaximizedCapital(int k, int W, vector<int>& Profits, vector<int>& Capital) {
4         priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>> minH;
5         priority_queue<pair<int, int>, vector<pair<int, int>>, less<pair<int, int>> maxH;
6         for (int i = 0; i < Capital.size(); ++i) {
7             minH.push({Capital[i], Profits[i]});
8         }
9         for (int i = 0; i < k; ++i) {
10             while (!minH.empty() && minH.top().first <= W) {
11                 auto t = minH.top(); minH.pop();
12                 maxH.push({t.second, t.first});
13             }
14             if (maxH.empty()) break;
15             W += maxH.top().first; maxH.pop();
16         }
17         return W;
18     }
19 };

```

下面这种方法跟上面的解法思路完全一样，就是数据结构有些变化，我们用multiset来模拟最小堆，然后最大堆还是用优先队列来实现，不过是需要存利润值就行了，不需要存对应的资本了，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int findMaximizedCapital(int k, int W, vector<int>& Profits, vector<int>& Capital) {
4         priority_queue<int> q;
5         multiset<pair<int, int>> s;
6         for (int i = 0; i < Capital.size(); ++i) {
7             s.insert({Capital[i], Profits[i]});
8         }
9         for (int i = 0; i < k; ++i) {
10             for (auto it = s.begin(); it != s.end(); ++it) {
11                 if (it->first > W) break;
12                 q.push(it->second);
13                 s.erase(it);
14             }
15             if (q.empty()) break;
16             W += q.top(); q.pop();
17         }
18         return W;
19     }
20 };

```

503. 下一个较大的元素之二

Greater Number for every element. The Next Greater Number of a number x is the first greater number to its traversing-order next in the array, which means you could search circularly to find its next greater number. If it doesn't exist, output -1 for this number.

Example 1:

Input: [1,2,1]
Output: [2,-1,2]

这道题是之前那道Next Greater Element I的拓展，不同的是，此时数组是一个循环数组，就是说某一个元素的下一个较大值可以在其前面，那么对于循环数组的遍历，为了使下标不超过数组的长度，我们需要对n取余，下面先来看暴力破解的方法，遍历每一个数字，然后对于每一个遍历到的数字，遍历所有其他数字，注意不是遍历到数组末尾，而是通过循环数组遍历其前一个数字，遇到较大值则存入结果res中，并break，再进行下一个数字的遍历，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<int> nextGreaterElements(vector<int>& nums) {
4         int n = nums.size();
5         vector<int> res(n, -1);
6         for (int i = 0; i < n; ++i) {
7             for (int j = i + 1; j < i + n; ++j) {
8                 if (nums[j % n] > nums[i]) {
9                     res[i] = nums[j % n];
10                    break;
11                }
12            }
13        }
14        return res;
15    }
16 };

```

我们可以使用栈来进行优化上面的算法，我们遍历两倍的数组，然后还是坐标i对n取余，取出数字，如果此时栈不为空，且栈顶元素小于当前数字，说明当前数字就是栈顶元素的右边第一个较大数，那么建立二者的映射，并且去除当前栈顶元素，最后如果i小于n，则把i压入栈。因为res的长度必须是n，超过n的部分我们只是为了给之前栈中的数字找较大值，所以不能压入栈，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<int> nextGreaterElements(vector<int>& nums) {
4         int n = nums.size();
5         vector<int> res(n, -1);
6         stack<int> st;
7         for (int i = 0; i < 2 * n; ++i) {
8             int num = nums[i % n];
9             while (!st.empty() && nums[st.top()] < num) {
10                 res[st.top()] = num; st.pop();
11             }
12             if (i < n) st.push(i);
13         }
14         return res;
15     }
16 };

```

504. 基数七

Given an integer, return its base 7 string representation.

Example 1:

Input: 100
Output: "202"

这道题给了我们一个数，让我们转为七进制的数，而且这个可正可负。那么我们想如果给一个十进制的100，怎么转为七进制。我会先用100除以49，商2余2。在除以7，商0余2，于是就得到七进制的202。其实我们还可以反过来算，先用100除以7，商14余2，然后用14除以7，商2余0，再用2除以7，商0余2，这样也可以得到202。这种方法更适合于代码实现，要注意的是，我们要处理好负数的情况，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     string convertToBase7(int num) {
4         if (num == 0) return "0";
5         string res = "";
6         bool positive = num > 0;
7         while (num != 0) {
8             res = to_string(abs(num % 7)) + res;
9             num /= 7;
10        }
11        return positive ? res : "-" + res;
12    }
13 };

```

CPP

上面的思路也可以写成迭代方式，非常的简洁，仅要三行就搞定了，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     string convertToBase7(int num) {
4         if (num < 0) return "-" + convertToBase7(-num);
5         if (num < 7) return to_string(num);
6         return convertToBase7(num / 7) + to_string(num % 7);
7     }
8 };
9

```

CPP

[505. 迷宫之二](#)

There is a ball in a maze with empty spaces and walls. The ball can go through empty spaces by rolling up, down, left or right, but it won't stop rolling until hitting a wall. When the ball stops, it could choose the next direction.

Given the ball's start position, the destination and the maze, find the shortest distance for the ball to stop at the destination. The distance is defined by the number of empty spaces traveled by the ball from the start position (excluded) to the destination (included). If the ball cannot stop at the destination, return -1.

The maze is represented by a binary 2D array. 1 means the wall and 0 means the empty space. You may assume that the borders of the maze are all walls. The start and destination coordinates are represented by row and column indexes.

这道题是之前那道The Maze的拓展，那道题只让我们判断能不能在终点位置停下，而这道题让我们求出到达终点的最少步数。其实本质都是一样的，难点还是在于对于一滚到底的实现方法，唯一不同的是，这里我们用一个二维数组dists，其中dists[i][j]表示到达(i,j)这个位置时需要的最小步数，我们都初始化为整型最大值，在后在遍历的过程中不断用较小值来更新每个位置的步数值，最后我们来看终点位置的步数值，如果还是整型最大值的话，说明没法在终点处停下来，返回-1，否则就返回步数值。注意在压入栈的时候，我们对x和y进行了判断，只有当其不是终点的时候才压入栈，这样是做了优化，因为如果小球已经滚到终点了，我们就不要让它再滚了，就不把终点位置压入栈，免得它还滚，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int shortestDistance(vector<vector<int>>& maze, vector<int>& start, vector<int>&
4 destination) {
5         int m = maze.size(), n = maze[0].size();
6         vector<vector<int>> dists(m, vector<int>(n, INT_MAX));
7         vector<vector<int>> dirs{{0,-1},{-1,0},{0,1},{1,0}};
8         queue<pair<int, int>> q;
9         q.push({start[0], start[1]});
10        dists[start[0]][start[1]] = 0;
11        while (!q.empty()) {
12            auto t = q.front(); q.pop();
13            for (auto d : dirs) {
14                int x = t.first, y = t.second, dist = dists[t.first][t.second];
15                while (x >= 0 && x < m && y >= 0 && y < n && maze[x][y] == 0) {
16                    x += d[0];
17                    y += d[1];
18                    ++dist;
19                }
20                x -= d[0];
21                y -= d[1];
22                --dist;
23                if (dists[x][y] > dist) {
24                    dists[x][y] = dist;
25                    if (x != destination[0] || y != destination[1]) q.push({x, y});
26                }
27            }
28        }
29        int res = dists[destination[0]][destination[1]];
30        return (res == INT_MAX) ? -1 : res;
31    }
32};

```

下面这种写法是DFS的解法，可以看出来思路基本上跟上面的解法没有啥区别，写法上稍有不同，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<vector<int>> dirs{{0,-1},{-1,0},{0,1},{1,0}};
4     int shortestDistance(vector<vector<int>>& maze, vector<int>& start, vector<int>&
5 destination) {
6         int m = maze.size(), n = maze[0].size();
7         vector<vector<int>> dists(m, vector<int>(n, INT_MAX));
8         dists[start[0]][start[1]] = 0;
9         helper(maze, start[0], start[1], destination, dists);
10        int res = dists[destination[0]][destination[1]];
11        return (res == INT_MAX) ? -1 : res;
12    }
13    void helper(vector<vector<int>>& maze, int i, int j, vector<int>& destination,
14    vector<vector<int>>& dists) {
15        if (i == destination[0] && j == destination[1]) return;
16        int m = maze.size(), n = maze[0].size();
17        for (auto d : dirs) {
18            int x = i, y = j, dist = dists[x][y];
19            while (x >= 0 && x < m && y >= 0 && y < n && maze[x][y] == 0) {
20                x += d[0];
21                y += d[1];
22                ++dist;
23            }
24            x -= d[0];
25            y -= d[1];
26            --dist;
27            if (dists[x][y] > dist) {
28                dists[x][y] = dist;
29                helper(maze, x, y, destination, dists);
30            }
31        }
32    }
33};

```

506. 相对排名

Given scores of N athletes, find their relative ranks and the people with the top three highest scores, who will be awarded medals: "Gold Medal", "Silver Medal" and "Bronze Medal".

这道题给了我们一组分数，让我们求相对排名，前三名分别是金银铜牌，后面的就是名次数，不是一道难题，我们可以利用堆来排序，建立一个优先队列，把分数和其坐标位置放入队列中，会自动按其分数高低排序，然后我们从顶端开始一个一个取出数据，由于保存了其在原数组的位置，我们可以直接将其存到结果res中正确的位置，用一个变量cnt来记录名词，前三名给奖牌，后面就是名次数，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     vector<string> findRelativeRanks(vector<int>& nums) {
4         int n = nums.size(), cnt = 1;
5         vector<string> res(n, "");
6         priority_queue<pair<int, int>> q;
7         for (int i = 0; i < n; ++i) {
8             q.push({nums[i], i});
9         }
10        for (int i = 0; i < n; ++i) {
11            int idx = q.top().second; q.pop();
12            if (cnt == 1) res[idx] = "Gold Medal";
13            else if (cnt == 2) res[idx] = "Silver Medal";
14            else if (cnt == 3) res[idx] = "Bronze Medal";
15            else res[idx] = to_string(cnt);
16            ++cnt;
17        }
18        return res;
19    }
20 };

```

下面这种方法思路和上面一样，不过数据结构用的不同，这里利用map的自动排序的功能，不过map是升序排列的，所以我们遍历的时候就要从最后面开始遍历，最后一个金牌，然后往前依次是银牌，铜牌，名次等，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<string> findRelativeRanks(vector<int>& nums) {
4         int n = nums.size(), cnt = 1;
5         vector<string> res(n, "");
6         map<int, int> m;
7         for (int i = 0; i < n; ++i) {
8             m[nums[i]] = i;
9         }
10        for (auto it = m.rbegin(); it != m.rend(); ++it) {
11            if (cnt == 1) res[it->second] = "Gold Medal";
12            else if (cnt == 2) res[it->second] = "Silver Medal";
13            else if (cnt == 3) res[it->second] = "Bronze Medal";
14            else res[it->second] = to_string(cnt);
15            ++cnt;
16        }
17        return res;
18    }
19 };

```

下面这种方法没用什么炫的数据结构，就是数组，建立一个坐标数组，不过排序的时候比较的不是坐标，而是该坐标位置上对应的数字，后面的处理方法和之前的并没有什么不同，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     vector<string> findRelativeRanks(vector<int>& nums) {
4         int n = nums.size();
5         vector<int> idx(n);
6         vector<string> res(n, "");
7         for (int i = 0; i < n; ++i) idx[i] = i;
8         sort(idx.begin(), idx.end(), [&](int a, int b){return nums[a] > nums[b];});
9         for (int i = 0; i < n; ++i) {
10             if (i == 0) res[idx[i]] = "Gold Medal";
11             else if (i == 1) res[idx[i]] = "Silver Medal";
12             else if (i == 2) res[idx[i]] = "Bronze Medal";
13             else res[idx[i]] = to_string(i + 1);
14         }
15         return res;
16     }
17 };

```

507. 完美数字

We define the Perfect Number is a positive integer that is equal to the sum of all its positive divisors except itself.

Now, given an integer n, write a function that returns true when it is a perfect number and false when it is not.

Example:

Input: 28

Output: True

Explanation: $28 = 1 + 2 + 4 + 7 + 14$

这道题让我们判断给定数字是否为完美数字，并给出完美数字的定义，就是一个整数等于除其自身之外的所有因子之和。那么由于不能包含自身，所以n必定大于1。其实这道题跟之前的判断质数的题蛮类似的，都是要找因子。由于1肯定是因子，可以提前加上，那么我们找其他因子的范围是 $[2, \sqrt{n}]$ 。我们遍历这之间所有的数字，如果可以被n整除，那么我们把i和num/i加上，对于n如果是平方数的话，那么我们此时相同的因子加来两次，所以我们要减掉一次。还有就是在遍历的过程中如果累积和sum大于n了，直接返回false即可。在循环结束后，我们看sum是否和num相等，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     bool checkPerfectNumber(int num) {
4         if (num == 1) return false;
5         int sum = 1;
6         for (int i = 2; i * i <= num; ++i) {
7             if (num % i == 0) sum += (i + num / i);
8             if (i * i == num) sum -= i;
9             if (sum > num) return false;
10        }
11        return sum == num;
12    }
13 };

```

下面这种方法叫的不行，在给定的n的范围内其实只有五个符合要求的完美数字，于是就有这种枚举的解法，那么套用一句诸葛亮的名言就是，我从未见过如此厚颜无耻之解法。哈哈，开个玩笑。写这篇博客的时候，国足正和伊朗进行十二强赛，上半场0比0，希望国足下半场能进球，好运好运，不忘初心，方得始终~

解法2：

```
1 class Solution {
2     public:
3         bool checkPerfectNumber(int num) {
4             return num==6 || num==28 || num==496 || num==8128 || num==33550336;
5         }
6     };

```

CPP

508. 出现频率最高的子树和

Given the root of a tree, you are asked to find the most frequent subtree sum. The subtree sum of a node is defined as the sum of all the node values formed by the subtree rooted at that node (including the node itself). So what is the most frequent subtree sum value? If there is a tie, return all the values with the highest frequency in any order.

Examples 1

Input:

```
5
/
2   -3
return [2, -3, 4], since all the values happen only once, return all of them in any order.
```

这道题给了我们一个二叉树，让我们求出现频率最高的子树之和，求树的结点和并不是很难，就是遍历所有结点累加起来即可。那么这道题的暴力解法就是遍历每个结点，对于每个结点都看作子树的根结点，然后再遍历子树所有结点求和，这样也许可以通过OJ，但是绝对不是最好的方法。我们想下子树有何特点，必须是要有叶结点，单独的一个叶结点也可以当作是子树，那么子树是从下往上构建的，这种特点很适合使用后序遍历，我们使用一个哈希表来建立子树和跟其出现频率的映射，用一个变量cnt来记录当前最多的次数，递归函数返回的是以当前结点为根结点的子树结点值之和，然后在递归函数中，我们先对当前结点的左右子结点调用递归函数，然后加上当前结点值，然后更新对应的哈希表中的值，然后看此时哈希表中的值是否大于等于cnt，大于的话首先要清空res，等于的话不用，然后将sum值加入结果res中即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<int> findFrequentTreeSum(TreeNode* root) {
4         vector<int> res;
5         unordered_map<int, int> m;
6         int cnt = 0;
7         postorder(root, m, cnt, res);
8         return res;
9     }
10    int postorder(TreeNode* node, unordered_map<int, int>& m, int& cnt, vector<int>& res) {
11        if (!node) return 0;
12        int left = postorder(node->left, m, cnt, res);
13        int right = postorder(node->right, m, cnt, res);
14        int sum = left + right + node->val;
15        ++m[sum];
16        if (m[sum] >= cnt) {
17            if (m[sum] > cnt) res.clear();
18            res.push_back(sum);
19            cnt = m[sum];
20        }
21        return sum;
22    }
23 };

```

下面这种解法跟上面的基本一样，就是没有在递归函数中更新结果res，更是利用cnt，最后再更新res，这样做能略微高效一些，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<int> findFrequentTreeSum(TreeNode* root) {
4         vector<int> res;
5         unordered_map<int, int> m;
6         int cnt = 0;
7         postorder(root, m, cnt);
8         for (auto a : m) {
9             if (a.second == cnt) res.push_back(a.first);
10        }
11        return res;
12    }
13    int postorder(TreeNode* node, unordered_map<int, int>& m, int& cnt) {
14        if (!node) return 0;
15        int left = postorder(node->left, m, cnt);
16        int right = postorder(node->right, m, cnt);
17        int sum = left + right + node->val;
18        cnt = max(cnt, ++m[sum]);
19        return sum;
20    }
21 };

```

开始我还在想能不能利用后序遍历的迭代形式来解，后来想了半天发现不太容易实现，因为博主无法想出有效的机制来保存左子树结点之和，而计算完对应的右子树结点之和后要用到对应的左子树结点之和，才能继续往上算。可能博主不够smart，有大神如果知道如何用迭代的形式来解，请一定要留言告知博主啊，多谢啦～

509. 寻找最左下树结点的值

Given a binary tree, find the leftmost value in the last row of the tree.

这道题让我们求二叉树的最左下树结点的值，也就是最后一行左数第一个值，那么我首先想的是用先序遍历来做，我们维护一个最大深度和该深度的结点值，由于先序遍历遍历的顺序是根-左右，所以每一行最左边的结点肯定最先遍历到，那么由于是新一行，那么当前深度肯定比之前的最大深度大，所以我们可以更新最大深度为当前深度，结点值res为当前结点值，这样在遍历到该行其他结点时就不会更新结果res了，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     int findBottomLeftValue(TreeNode* root) {
4         if (!root) return 0;
5         int max_depth = 1, res = root->val;
6         helper(root, 1, max_depth, res);
7         return res;
8     }
9     void helper(TreeNode* node, int depth, int& max_depth, int& res) {
10        if (!node) return;
11        if (depth > max_depth) {
12            max_depth = depth;
13            res = node->val;
14        }
15        helper(node->left, depth + 1, max_depth, res);
16        helper(node->right, depth + 1, max_depth, res);
17    }
18};
```

CPP

其实这道题用层序遍历更直接一些，因为层序遍历时遍历完当前行所有结点之后才去下一行，那么我们再遍历每行第一个结点时更新结果res即可，根本不用维护最大深度了，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     int findBottomLeftValue(TreeNode* root) {
4         if (!root) return 0;
5         int res = 0;
6         queue<TreeNode*> q;
7         q.push(root);
8         while (!q.empty()) {
9             int n = q.size();
10            for (int i = 0; i < n; ++i) {
11                TreeNode *t = q.front(); q.pop();
12                if (i == 0) res = t->val;
13                if (t->left) q.push(t->left);
14                if (t->right) q.push(t->right);
15            }
16        }
17        return res;
18    }
19};
```

CPP

510. 自由之路

In the video game Fallout 4, the quest "Road to Freedom" requires players to reach a metal dial called the "Freedom Trail Ring", and use the dial to spell a specific keyword in order to open the door.

Given a string ring, which represents the code engraved on the outer ring and another string key, which represents the keyword needs to be spelled. You need to find the minimum number of steps in order to spell all the characters in the keyword.

Initially, the first character of the ring is aligned at 12:00 direction. You need to spell all the characters in the string key one by one by rotating the ring clockwise or anticlockwise to make each character of the string key aligned at 12:00 direction and then by pressing the center button.

At the stage of rotating the ring to spell the key character $key[i]$:

You can rotate the ring clockwise or anticlockwise one place, which counts as 1 step. The final purpose of the rotation is to align one of the string ring's characters at the 12:00 direction, where this character must equal to the character $key[i]$.

If the character $key[i]$ has been aligned at the 12:00 direction, you need to press the center button to spell, which also counts as 1 step. After the pressing, you could begin to spell the next character in the key (next stage), otherwise, you've finished all the spelling.

这道题是关于辐射4这款游戏出的，博主虽然没有玩过这款游戏，但是知道确实有些游戏中需要破解一些谜题才能继续通关，像博主很早以前玩过的恐龙危机啊，生化危机啊啥的，都有一些机关需要破解，博主大部分都要靠看攻略来通关哈哈。这道题讲的是一种叫做自由之路的机关，我们需要将密码字符串都转出来，让我们求最短的转动步数。博主最先尝试的用贪婪算法来做，就是每一步都选最短的转法，但是OJ中总有些test case会引诱贪婪算法得出错误的结果，因为全局最优解不一定都是局部最优解，而贪婪算法一直都是在累加局部最优解，这也是为啥DP解法这么叼的原因。贪婪算法好想好实现，但是不一定能得到正确的结果。DP解法难想不好写，但往往才是正确的解法，这也算一个trade off吧。这道题可以用DP来解，难点还是写递推公式，博主在充分研究网上大神们的帖子后尝试着自己理理思路，如果有不正确或者不足的地方，也请各位不吝赐教。此题需要使用一个二维数组dp，其中 $dp[i][j]$ 表示转动从i位置开始的key串所需要的最少步数(这里不包括spell的步数，因为spell可以在最后统一加上)，此时表盘的12点位置是ring中的第j个字符。不得不佩服这样的设计的确很巧妙，我们可以从key的末尾往前推，这样 $dp[0][0]$ 就是我们所需要的结果，因为此时是从key的开头开始转动，而且表盘此时的12点位置也是ring的第一个字符。现在我们来看如何找出递推公式，对于 $dp[i][j]$ ，我们知道此时要将 $key[i]$ 转动到12点的位置，而此时表盘的12点位置是ring[j]，我们有两种旋转的方式，顺时针和逆时针，我们的目标肯定是要求最小的转动步数，而顺时针和逆时针的转动次数之和刚好为ring的长度n，这样我们求出来一个方向的次数，就可以迅速得到反方向的转动次数。为了将此时表盘上12点位置上的ring[j]转动到 $key[i]$ ，我们要将表盘转动一整圈，当转到 $key[i]$ 的位置时，我们计算出转动步数diff，然后计算出反向转动步数，并取二者较小值为整个转动步数step，此时我们更新 $dp[i][j]$ ，更新对比值为 $step + dp[i+1][k]$ ，这个也不难理解，因为key的前一个字符 $key[i+1]$ 的转动情况suppose已经计算好了，那么 $dp[i+1][k]$ 就是当时表盘12点位置上ring[k]的情况的最短步数，step就是从ring[k]转到ring[j]的步数，也就是 $key[i]$ 转到ring[j]的步数，用语言来描述就是，从key的i位置开始转动并且此时表盘12点位置为ring[j]的最小步数($dp[i][j]$)就等价于将ring[k]转动到12点位置的步数(step)加上从key的i+1位置开始转动并且ring[k]已经在表盘12点位置上的最小步数($dp[i+1][k]$)之和。突然发现这不就是之前那道Reverse Pairs中解法一中归纳的顺序重现关系的思路吗，都做了总结，可换个马甲就又不认识了，泪目中。。。。

解法1：

```

1 class Solution {
2 public:
3     int findRotateSteps(string ring, string key) {
4         int n = ring.size(), m = key.size();
5         vector<vector<int>> dp(m + 1, vector<int>(n));
6         for (int i = m - 1; i >= 0; --i) {
7             for (int j = 0; j < n; ++j) {
8                 dp[i][j] = INT_MAX;
9                 for (int k = 0; k < n; ++k) {
10                     if (ring[k] == key[i]) {
11                         int diff = abs(j - k);
12                         int step = min(diff, n - diff);
13                         dp[i][j] = min(dp[i][j], step + dp[i + 1][k]);
14                     }
15                 }
16             }
17         }
18         return dp[0][0] + m;
19     }
20 };

```

下面这种解法是用DFS来解的，我们需要做优化，也就是用memo数组来保存已经计算过的结果，否则大量的重复运算是无法通过OJ的。其实这里的memo数组也起到了跟上面解法中的dp数组相类似的作用，还有就是要注意数组v的作用，记录了每个字母在ring中的出现位置，由于ring中可能有重复字符，而且麻烦的情况是当前位置向两个方向分别转动相同的步数会分别到达两个相同的字符，这也是贪婪算法会失效的一个重要原因，而且也是上面的解法在找到ring[k] == key[i]并处理完之后不break的原因，因为后面还有可能找到。上面的迭代解法中使用到的变量i和j可以直接访问到，而在递归的写法中必须要把位置变量x和y当作参数导入进去，这样才能更新正确的地方，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int findRotateSteps(string ring, string key) {
4         int n = ring.size(), m = key.size();
5         vector<vector<int>> v(26);
6         vector<vector<int>> memo(n, vector<int>(m));
7         for (int i = 0; i < n; ++i) v[ring[i] - 'a'].push_back(i);
8         return helper(ring, key, 0, 0, v, memo);
9     }
10    int helper(string ring, string key, int x, int y, vector<vector<int>>&v,
11    vector<vector<int>>& memo) {
12        if (y == key.size()) return 0;
13        if (memo[x][y]) return memo[x][y];
14        int res = INT_MAX, n = ring.size();
15        for (int k : v[key[y] - 'a']) {
16            int diff = abs(x - k);
17            int step = min(diff, n - diff);
18            res = min(res, step + helper(ring, key, k, y + 1, v, memo));
19        }
20        return memo[x][y] = res + 1;
21    }
22};

```

511. 找树每行最大的结点值

You need to find the largest value in each row of a binary tree.

这道题让我们找二叉树每行的最大的结点值，那么实际上最直接的方法就是用层序遍历，然后在每一层中找到最大值，加入结果res中即可，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     vector<int> largestValues(TreeNode* root) {
4         if (!root) return {};
5         vector<int> res;
6         queue<TreeNode*> q;
7         q.push(root);
8         while (!q.empty()) {
9             int n = q.size(), mx = INT_MIN;
10            for (int i = 0; i < n; ++i) {
11                TreeNode *t = q.front(); q.pop();
12                mx = max(mx, t->val);
13                if (t->left) q.push(t->left);
14                if (t->right) q.push(t->right);
15            }
16            res.push_back(mx);
17        }
18        return res;
19    }
20};
```

如果我们想用迭代的方法来解，可以用先序遍历，这样的话就需要维护一个深度变量depth，来记录当前结点的深度，如果当前深度大于结果res的长度，说明这个新一层，我们将当前结点值加入结果res中，如果不大于res的长度的话，我们用当前结点值和结果res中对应深度的那个结点值相比较，取较大值赋给结果res中的对应深度位置，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     vector<int> largestValues(TreeNode* root) {
4         if (!root) return {};
5         vector<int> res;
6         helper(root, 1, res);
7         return res;
8     }
9     void helper(TreeNode* root, int depth, vector<int>& res) {
10        if (depth > res.size()) res.push_back(root->val);
11        else res[depth - 1] = max(res[depth - 1], root->val);
12        if (root->left) helper(root->left, depth + 1, res);
13        if (root->right) helper(root->right, depth + 1, res);
14    }
15}
16};
```

512. 最长回文子序列

Given a string s, find the longest palindromic subsequence's length in s. You may assume that the maximum length of s is 1000.

这道题给了我们一个字符串，让我们求最大的回文子序列，子序列和子字符串不同，不需要连续。而关于回文串的题之前也做了不少，处理方法上就是老老实实的两两比较吧。像这种有关极值的问题，最应该优先考虑的就是贪婪算法和动态规划，这道题显然使用DP更加合适。我们建立一个二维的DP数组，其中 $dp[i][j]$ 表示 $[i, j]$ 区间内的字符串的最长回文子序列，那么对于递推公式我们分析一下，如果 $s[i]==s[j]$ ，那么 i 和 j 就可以增加2个回文串的长度，我们知道中间 $dp[i+1][j-1]$ 的值，那么其加上2就是 $dp[i][j]$ 的值。如果 $s[i] != s[j]$ ，那么我们可以去掉 i 或 j 其中的一个字符，然后比较两种情况下所剩的字符串谁 dp 值大，就赋给 $dp[i][j]$ ，那么递推公式如下：

```

    /  dp[i + 1][j - 1] + 2                                if (s[i] == s[j])
dp[i][j] =
\  max(dp[i + 1][j], dp[i][j - 1])                      if (s[i] != s[j])

```

解法1：

```
1 class Solution {
2 public:
3     int longestPalindromeSubseq(string s) {
4         int n = s.size();
5         vector<vector<int>> dp(n, vector<int>(n));
6         for (int i = n - 1; i >= 0; --i) {
7             dp[i][i] = 1;
8             for (int j = i + 1; j < n; ++j) {
9                 if (s[i] == s[j]) {
10                     dp[i][j] = dp[i + 1][j - 1] + 2;
11                 } else {
12                     dp[i][j] = max(dp[i + 1][j], dp[i][j - 1]);
13                 }
14             }
15         }
16         return dp[0][n - 1];
17     }
18 }
```

CPP

我们可以对空间进行优化，只用一个一维的 dp 数组，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int longestPalindromeSubseq(string s) {
4         int n = s.size(), res = 0;
5         vector<int> dp(n, 1);
6         for (int i = n - 1; i >= 0; --i) {
7             int len = 0;
8             for (int j = i + 1; j < n; ++j) {
9                 int t = dp[j];
10                if (s[i] == s[j]) {
11                    dp[j] = len + 2;
12                }
13                len = max(len, t);
14            }
15        }
16        for (int num : dp) res = max(res, num);
17        return res;
18    }
19 };

```

下面是递归形式的解法，memo数组这里起到了一个缓存已经计算过了的结果，这样能提高运算效率，使其不会TLE，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int longestPalindromeSubseq(string s) {
4         int n = s.size();
5         vector<vector<int>> memo(n, vector<int>(n, -1));
6         return helper(s, 0, n - 1, memo);
7     }
8     int helper(string& s, int i, int j, vector<vector<int>>& memo) {
9         if (memo[i][j] != -1) return memo[i][j];
10        if (i > j) return 0;
11        if (i == j) return 1;
12        if (s[i] == s[j]) {
13            memo[i][j] = helper(s, i + 1, j - 1, memo) + 2;
14        } else {
15            memo[i][j] = max(helper(s, i + 1, j, memo), helper(s, i, j - 1, memo));
16        }
17        return memo[i][j];
18    }
19 };

```

513. 超级洗衣机

You have n super washing machines on a line. Initially, each washing machine has some dresses or is empty.

For each move, you could choose any m ($1 \leq m \leq n$) washing machines, and pass one dress of each washing machine to one of its adjacent washing machines at the same time .

Given an integer array representing the number of dresses in each washing machine from left to right on the line, you should find the minimum number of moves to make all the washing machines have the same number of dresses. If it is not possible to do it, return -1.

这道题给了我们一堆工藤新一，噢不，是滚筒洗衣机。我们有许多洗衣机，每个洗衣机里的衣服数不同，每个洗衣机每次只允许向相邻的洗衣机转移一件衣服，问要多少次才能使所有洗衣机的衣服数相等。注意这里的一次移动是说所有洗衣机都可以移动一件衣服到其相邻的洗衣机。这道题的代码量其实不多，难点是在于解题思路，难的是对问题的等价转换等。博主也没有做出这道题，博主想到了要先验证衣服总数是否能整除洗衣机的数量，然后计算出每个洗衣机最终应该放的衣服数，返回跟初始状态衣服数之差的最大值，但这种解法是不对的，无法通过这个test case [0, 0, 11, 5]，最终每个洗衣机会留4件衣服，我想的那方法会返回7，然后正确答案是8。想想也是，如果这么是这么简单的思路，这题怎么会标记为Hard呢，还是图样图森破啊。这里直接参照大神Chidong的帖子来做，我们就用上面那个例子，有四个洗衣机，装的衣服数为[0, 0, 11, 5]，最终的状态会变为[4, 4, 4, 4]，那么我们将二者做差，得到[-4, -4, 7, 1]，这里负数表示当前洗衣机还需要的衣服数，正数表示当前洗衣机多余的衣服数。我们要做的是要将这个差值数组每一项都变为0，对于第一个洗衣机来说，需要四件衣服可以从第二个洗衣机获得，那么就可以把-4移给二号洗衣机，那么差值数组变为[0, -8, 7, 1]，此时二号洗衣机需要八件衣服，那么至少需要移动8次。然后二号洗衣机把这八件衣服从三号洗衣机处获得，那么差值数组变为[0, 0, -1, 1]，此时三号洗衣机还缺1件，就从四号洗衣机处获得，此时差值数组成功变为了[0, 0, 0, 0]，成功。那么移动的最大次数就是差值数组中出现的绝对值最大的数字，8次，参见代码如下：

```

1 class Solution {
2 public:
3     int findMinMoves(vector<int>& machines) {
4         int sum = accumulate(machines.begin(), machines.end(), 0);
5         if (sum % machines.size() != 0) return -1;
6         int res = 0, cnt = 0, avg = sum / machines.size();
7         for (int m : machines) {
8             cnt += m - avg;
9             res = max(res, max(abs(cnt), m - avg));
10        }
11    return res;
12 }
13 };

```

CPP

514. 硬币找零之二

You are given coins of different denominations and a total amount of money. Write a function to compute the number of combinations that make up that amount. You may assume that you have infinite number of each kind of coin.

Note: You can assume that

```

0 <= amount <= 5000
1 <= coin <= 5000
the number of coins is less than 500
the answer is guaranteed to fit into signed 32-bit integer

```

Example 1:

Input: amount = 5, coins = [1, 2, 5]

Output: 4

Explanation: there are four ways to make up the amount:

5=5

5=2+2+1

5=2+1+1+1

5=1+1+1+1+1

这道题是之前那道Coin Change的拓展，那道题问我们最少能用多少个硬币组成给定的钱数，而这道题问的是组成给定钱数总共有多少种不同的方法。那么我们还是要使用DP来做，首先我们来考虑最简单的情况，如果只有一个硬币的话，那么给定钱数的组成方式就最多有1种，就看此钱数能否整除该硬币值。那么当有两个硬币的话，那么组成某个钱数的方式就可能有多种，比如可能由每种硬币单独来组成，或者是两种硬币同时来组成。那么我们怎么量化呢，比如我们有两个硬币[1, 2]，钱数为5，那么钱数的5的组成方法是可以看作两部分组成，一种是由硬币1单独组成，那么仅有一种情况(1+1+1+1+1)；另一种是由1和2共同组成，说明我们的组成方法中至少需要由一个2，所以此时我们先取出一个硬币2，那么我们只要拼出钱数为3即可，这个3还是可以用硬币1和2来拼，所以就相当于求由硬币[1, 2]组成的钱数为3的总方法。是不是不太好理解，多想想。那么我们需要一个二维的dp数组，其中 $dp[i][j]$ 表示用前*i*个硬币组成钱数为*j*的不同组合方法，那么怎么算才不会重复，也不会漏掉呢？我们采用的方法是一个硬币一个硬币的增加，每增加一个硬币，都从1遍历到amount，对于遍历到的当前钱数*j*，组成方法就是不加上当前硬币的频发 $dp[i-1][j]$ ，还要加上，去掉当前硬币值的钱数的组成方法，当然钱数*j*要大于当前硬币值，那么我们的递推公式也在上面的分析中得到了：

```
dp[i][j] = dp[i - 1][j] + (j >= coins[i - 1] ? dp[i][j - coins[i - 1]] : 0)
```

注意我们要初始化每行的第一个位置为0，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     int change(int amount, vector<int>& coins) {
4         vector<vector<int>> dp(coins.size() + 1, vector<int>(amount + 1, 0));
5         dp[0][0] = 1;
6         for (int i = 1; i <= coins.size(); ++i) {
7             dp[i][0] = 1;
8             for (int j = 1; j <= amount; ++j) {
9                 dp[i][j] = dp[i - 1][j] + (j >= coins[i - 1] ? dp[i][j - coins[i - 1]] : 0);
10            }
11        }
12        return dp[coins.size()][amount];
13    }
14 }
```

CPP

我们可以对空间进行优化，我们发现 $dp[i][j]$ 仅仅依赖于 $dp[i - 1][j]$ 和 $dp[i][j - coins[i - 1]]$ 这两项，那么我们可以使用一个一维dp数组来代替，此时的 $dp[i]$ 表示组成钱数*i*的不同方法。其实最开始的时候，博主就想着用一维的dp数组来写，但是博主开始想的方法是把里面两个for循环调换了一个位置，结果计算的种类数要大于正确答案，所以一定要注意for循环的顺序不能搞反，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     int change(int amount, vector<int>& coins) {
4         vector<int> dp(amount + 1, 0);
5         dp[0] = 1;
6         for (int coin : coins) {
7             for (int i = coin; i <= amount; ++i) {
8                 dp[i] += dp[i - coin];
9             }
10        }
11        return dp[amount];
12    }
13 }
```

CPP

在CareerCup中，有一道极其相似的题9.8 Represent N Cents 美分的组成，书里面用的是那种递归的方法，博主想将其解法直接搬到这道题里，但是失败了，博主发现使用那种的递归的解法必须要有值为1的硬币存在，这点无法在这道题里满足。你以为这样博主就没有办法了吗？当然有，博主加了判断，当用到最后一个硬币时，我们判断当前还剩点钱数是否能整除这个硬币，不能的话就返回0，否则返回1。还有就是用二维数组的memo会TLE，所以博主换成了map，就可以通过啦~

解法3：

```

1 class Solution {
2 public:
3     int change(int amount, vector<int>& coins) {
4         if (amount == 0) return 1;
5         if (coins.empty()) return 0;
6         map<pair<int, int>, int> memo;
7         return helper(amount, coins, 0, memo);
8     }
9     int helper(int amount, vector<int>& coins, int idx, map<pair<int, int>, int>& memo) {
10        if (amount == 0) return 1;
11        else if (idx >= coins.size()) return 0;
12        else if (idx == coins.size() - 1) return amount % coins[idx] == 0;
13        if (memo.count({amount, idx})) return memo[{amount, idx}];
14        int val = coins[idx], res = 0;
15        for (int i = 0; i * val <= amount; ++i) {
16            int rem = amount - i * val;
17            res += helper(rem, coins, idx + 1, memo);
18        }
19        memo[{amount, idx}] = res;
20    }
21 };

```

515. 检测大写格式

Given a word, you need to judge whether the usage of capitals in it is right or not.

We define the usage of capitals in a word to be right when one of the following cases holds:

All letters in this word are capitals, like "USA".

All letters in this word are not capitals, like "leetcode".

Only the first letter in this word is capital if it has more than one letter, like "Google".

Otherwise, we define that this word doesn't use capitals in a right way.

这道题给了我们一个单词，让我们检测大写格式是否正确，规定了三种正确方式，要么都是大写或小写，要么首字母大写，其他情况都不正确。那么我们要做的就是统计出单词中所有大写字母的个数cnt，再来判断是否属于这三种情况，如果cnt为0，说明都是小写，正确；如果cnt和单词长度相等，说明都是大写，正确；如果cnt为1，且首字母为大写，正确，其他情况均返回false，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool detectCapitalUse(string word) {
4         int cnt = 0, n = word.size();
5         for (int i = 0; i < n; ++i) {
6             if (word[i] <= 'Z') ++cnt;
7         }
8         return cnt == 0 || cnt == n || (cnt == 1 && word[0] <= 'Z');
9     }
10 };

```

下面这种方法利用了STL的内置方法count_if，根据条件来计数，这样使得code非常简洁，两行就搞定了，丧心病狂啊~

解法2：

```

1 class Solution {
2 public:
3     bool detectCapitalUse(string word) {
4         int cnt = count_if(word.begin(), word.end(), [] (char c){return c <= 'Z';});
5         return cnt == 0 || cnt == word.size() || (cnt == 1 && word[0] <= 'Z');
6     }
7 };

```

516. 最长非共同子序列之一

Given a group of two strings, you need to find the longest uncommon subsequence of this group of two strings. The longest uncommon subsequence is defined as the longest subsequence of one of these strings and this subsequence should not be any subsequence of the other strings.

A subsequence is a sequence that can be derived from one sequence by deleting some characters without changing the order of the remaining elements. Trivially, any string is a subsequence of itself and an empty string is a subsequence of any string.

The input will be two strings, and the output needs to be the length of the longest uncommon subsequence. If the longest uncommon subsequence doesn't exist, return -1

这道题是在4月1号出的，有人说这是愚人节最好的礼物，哈哈~这道题只是为了后面那道题做铺垫，不过这题确实简单啊，两个字符串的情况很少，如果两个字符串相等，那么一定没有非共同子序列，反之，如果两个字符串不等，那么较长的那个字符串就是最长非共同子序列，参见代码如下：

```

1 class Solution {
2 public:
3     int findLUSlength(string a, string b) {
4         return a == b ? -1 : max(a.size(), b.size());
5     }
6 };

```

517. 最长非共同子序列之二

Given a list of strings, you need to find the longest uncommon subsequence among them. The longest uncommon subsequence is defined as the longest subsequence of one of these strings and this subsequence should not be any subsequence of the other strings.

A subsequence is a sequence that can be derived from one sequence by deleting some characters without changing the order of the remaining elements. Trivially, any string is a subsequence of itself and an empty string is a subsequence of any string.

The input will be a list of strings, and the output needs to be the length of the longest uncommon subsequence. If the longest uncommon subsequence doesn't exist, return -1.

这道题是之前那道Longest Uncommon Subsequence I的拓展，那道题因为只有两个字符串为大家所不屑。那么这道题有多个字符串，这次大家满足了吧。令我吃惊的是，这次的OJ异常的大度，连暴力搜索的解法也让过，那么还等什么，无脑暴力破解吧。遍历所有的字符串，对于每个遍历到的字符串，再和所有的其他的字符串比较，看是不是某一个字符串的子序列，如果都不是的话，那么当前字符串就是一个非共同子序列，用其长度来更新结果res，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int findLUSlength(vector<string>& strs) {
4         int res = -1, j = 0, n = strs.size();
5         for (int i = 0; i < n; ++i) {
6             for (j = 0; j < n; ++j) {
7                 if (i == j) continue;
8                 if (checkSubs(strs[i], strs[j])) break;
9             }
10            if (j == n) res = max(res, (int)strs[i].size());
11        }
12        return res;
13    }
14    int checkSubs(string subs, string str) {
15        int i = 0;
16        for (char c : str) {
17            if (c == subs[i]) ++i;
18            if (i == subs.size()) break;
19        }
20        return i == subs.size();
21    }
22};

```

CPP

下面这种解法使用一些博主能想到的优化手段，首先我们给字符串按长度来排序，将长度大的放到前面，这样我们如果找到了非共同子序列，那么直接返回其长度即可，因为当前找到的肯定是最长的。然后我们用一个集合来记录已经遍历过的字符串，利用集合的去重复特性，这样在有大量的重复字符串的时候可以提高效率，然后我们开始遍历字符串，对于当前遍历到的字符串，我们和集合中的所有字符串相比，看其是否是某个的子序列，如果都不是，说明当前的就是最长的非共同子序列。注意如果当前的字符串是集合中某个字符串的子序列，那么直接break出来，不用再和其他的比较了，这样在集合中有大量的字符串时可以提高效率，最后别忘了将遍历过的字符串加入集合中，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int findLUSlength(vector<string>& strs) {
4         int n = strs.size();
5         unordered_set<string> s;
6         sort(strs.begin(), strs.end(), [] (string a, string b) {
7             if (a.size() == b.size()) return a > b;
8             return a.size() > b.size();
9         });
10        for (int i = 0; i < n; ++i) {
11            if (i == n - 1 || strs[i] != strs[i + 1]) {
12                bool found = true;
13                for (auto a : s) {
14                    int j = 0;
15                    for (char c : a) {
16                        if (c == strs[i][j]) ++j;
17                        if (j == strs[i].size()) break;
18                    }
19                    if (j == strs[i].size()) {
20                        found = false;
21                        break;
22                    }
23                }
24                if (found) return strs[i].size();
25            }
26            s.insert(strs[i]);
27        }
28        return -1;
29    }
30 };

```

518. 连续的子数组之和

Given a list of non-negative numbers and a target integer k, write a function to check if the array has a continuous subarray of size at least 2 that sums up to the multiple of k, that is, sums up to $n*k$ where n is also an integer.

Example 1:

Input: [23, 2, 4, 6, 7], k=6

Output: True

这道题给了我们一个数组和一个数字k，让我们求是否存在这样的一个连续的子数组，该子数组的数组之和可以整除k。遇到除法问题，我们肯定不能忘了除数为0的情况等处理。还有就是我们如何能快速的遍历所有的子数组，并且求和，我们肯定不能完全的暴力破解，这样OJ肯定不答应。我们需要适当的优化，如果是刷题老司机的话，遇到这种求子数组或者子矩阵之和的题，应该不难想到要建立累加和数组或者累加和矩阵来做。没错，这道题也得这么做，我们要遍历所有的子数组，然后利用累加和来快速求和。在得到每个子数组之和时，我们先和k比较，如果相同直接返回true，否则再判断，若k不为0，且sum能整除k，同样返回true，最后遍历结束返回false，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool checkSubarraySum(vector<int>& nums, int k) {
4         for (int i = 0; i < nums.size(); ++i) {
5             int sum = nums[i];
6             for (int j = i + 1; j < nums.size(); ++j) {
7                 sum += nums[j];
8                 if (sum == k) return true;
9                 if (k != 0 && sum % k == 0) return true;
10            }
11        }
12        return false;
13    }
14 };

```

下面这种方法用了些技巧，那就是，若数字a和b分别除以数字c，若得到的余数相同，那么(a-b)必定能够整除c。这里就不证明了，博主也不会证明。明白了这条定理，那么我们用一个集合set来保存所有出现过的余数，如果当前的累加和除以k得到的余数在set中已经存在了，那么说明之前必定有一段子数组和可以整除k。需要注意的是k为0的情况，由于无法取余，我们就把当前累加和放入set中。还有就是题目要求子数组至少需要两个数字，那么我们需要一个变量pre来记录之前的和，我们每次存入set中的是pre，而不是当前的累积和，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool checkSubarraySum(vector<int>& nums, int k) {
4         int n = nums.size(), sum = 0, pre = 0;
5         unordered_set<int> st;
6         for (int i = 0; i < n; ++i) {
7             sum += nums[i];
8             int t = (k == 0) ? sum : (sum % k);
9             if (st.count(t)) return true;
10            st.insert(pre);
11            pre = t;
12        }
13        return false;
14    }
15 };

```

既然set可以做，一般来说用哈希表也可以做，这里我们建立余数和当前位置之间的映射，由于有了位置信息，我们就不需要pre变量了，之前用保存的坐标和当前位置i比较判断就可以了，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     bool checkSubarraySum(vector<int>& nums, int k) {
4         int n = nums.size(), sum = 0;
5         unordered_map<int, int> m{{0,-1}};
6         for (int i = 0; i < n; ++i) {
7             sum += nums[i];
8             int t = (k == 0) ? sum : (sum % k);
9             if (m.count(t)) {
10                 if (i - m[t] > 1) return true;
11             } else m[t] = i;
12         }
13         return false;
14     }
15 };

```

519. 优美排列

Suppose you have N integers from 1 to N. We define a beautiful arrangement as an array that is constructed by these N numbers successfully if one of the following is true for the ith position ($1 \leq i \leq N$) in this array:

The number at the ith position is divisible by i.
i is divisible by the number at the ith position.

Now given N, how many beautiful arrangements can you construct?

这道题给了我们1到N，总共N个正数，然后定义了一种优美排列方式，对于该排列中的所有数，如果数字可以整除下标，或者下标可以整除数字，那么我们就是优美排列，让我们求出所有优美排列的个数。那么对于求种类个数，或者是求所有情况，这种问题通常要用递归来做，递归简直是暴力的不能再暴力的方法了。而递归方法等难点在于写递归函数，如何确定终止条件，还有for循环中变量的起始位置如何确定。那么这里我们需要一个visited数组来记录数字是否已经访问过，因为优美排列中不能有重复数字。我们用变量pos来标记已经生成的数字的个数，如果大于N了，说明已经找到了一组排列，结果res自增1。在for循环中，i应该从1开始，因为我们遍历1到N中的所有数字，如果该数字未被使用过，且满足和坐标之间的整除关系，那么我们标记该数字已被访问过，再调用下一个位置的递归函数，之后不要忘记了恢复初始状态，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int countArrangement(int N) {
4         int res = 0;
5         vector<int> visited(N + 1, 0);
6         helper(N, visited, 1, res);
7         return res;
8     }
9     void helper(int N, vector<int>& visited, int pos, int& res) {
10        if (pos > N) {
11            ++res;
12            return;
13        }
14        for (int i = 1; i <= N; ++i) {
15            if (visited[i] == 0 && (i % pos == 0 || pos % i == 0)) {
16                visited[i] = 1;
17                helper(N, visited, pos + 1, res);
18                visited[i] = 0;
19            }
20        }
21    }
22 };

```

上面的解法在N=4时产生的优美序列如下：

```

1 2 3 4
1 4 3 2
2 1 3 4
2 4 3 1
3 2 1 4
3 4 1 2
4 1 3 2
4 2 3 1

```

通过看上面的分析，是不是觉得这道题的本质其实是求全排列，然后在所有全排列中筛选出符合题意的排列。那么求全排列的另一种经典解法就是交换数组中任意两个数字的位置，来形成新的排列，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int countArrangement(int N) {
4         vector<int> nums(N);
5         for (int i = 0; i < N; ++i) nums[i] = i + 1;
6         return helper(N, nums);
7     }
8     int helper(int n, vector<int>& nums) {
9         if (n <= 0) return 1;
10        int res = 0;
11        for (int i = 0; i < n; ++i) {
12            if (n % nums[i] == 0 || nums[i] % n == 0) {
13                swap(nums[i], nums[n - 1]);
14                res += helper(n - 1, nums);
15                swap(nums[i], nums[n - 1]);
16            }
17        }
18        return res;
19    }
20 };

```

上面的解法在N=4时产生的优美序列如下：

```

2 4 3 1
4 2 3 1
3 4 1 2
4 1 3 2
1 4 3 2
3 2 1 4
2 1 3 4
1 2 3 4

```

520. 单词缩写

Given an array of n distinct non-empty strings, you need to generate minimal possible abbreviations for every word following rules below.

Begin with the first character and then the number of characters abbreviated, which followed by the last character.

If there are any conflict, that is more than one words share the same abbreviation, a longer prefix is used instead of only the first character until making the map from word to abbreviation become unique. In other words, a final abbreviation cannot map to more than one original words.

If the abbreviation doesn't make the word shorter, then keep it as original.

这道题让我们求单词的缩写形式，就是首尾字母加上中间字符的个数组成的新字符串，但是要求是不能有重复的缩写字符串，而且说明如果缩写字符串的长度并没有减小的话就保留原来的字符串，比如god，缩写成g1d也没啥用，所以仍是god。博主刚开始在研究题目中给的例子的时候有些疑惑，虽然知道internal和interval的缩写形式都是i6l，会冲突，博主刚开始不明白的是，为什么不能一个是i6l，一个是in5l，这样不就不冲突了么，而题目中的缩写形式居然都是原字符串。后来才搞清楚题目原来是说只要有冲突的都不能用，而internal和interval是典型的死杠上的一对，i6l, in5l, int4l, inte3l, inter2l, 统统冲突，而再往后的缩写长度就和原字符串一样了，所以二者就都保留了原样。理解了题意就好办了，由于每个单词的缩写形式中数字前面的字母个数不一定相同，所以我们用一个pre数组来记录每个单词缩写形式开头字母的长度，初始化都为1，然后先求出所有单词pre为1的缩写形式，再来进行冲突处理。我们遍历每一个缩写字符串，进行while循环，新建一个集合set，然后遍历其他所有字符

串，所有发现冲突字符串，就把冲突字符串的坐标存入集合中，如果没有冲突，那么集合为空，直接break掉，如果由冲突，那么还要把当前遍历的位置i加入结合中，然后遍历集合中所有的位置，对其调用缩写函数，此时pre对应的值自增1，直到没有冲突存在为止，参见代码如下：

```

1 class Solution {
2 public:
3     vector<string> wordsAbbreviation(vector<string>& dict) {
4         int n = dict.size();
5         vector<string> res(n);
6         vector<int> pre(n, 1);
7         for (int i = 0; i < n; ++i) {
8             res[i] = abbreviate(dict[i], pre[i]);
9         }
10        for (int i = 0; i < n; ++i) {
11            while (true) {
12                set<int> s;
13                for (int j = i + 1; j < n; ++j) {
14                    if (res[j] == res[i]) s.insert(j);
15                }
16                if (s.empty()) break;
17                s.insert(i);
18                for (auto a : s) {
19                    res[a] = abbreviate(dict[a], ++pre[a]);
20                }
21            }
22        }
23        return res;
24    }
25    string abbreviate(string s, int k) {
26        return (k >= s.size() - 2) ? s : s.substr(0, k) + to_string(s.size() - k - 1) +
27        s.back();
28    }
};

```

521. 扫雷游戏

Let's play the minesweeper game (Wikipedia, online game)!

You are given a 2D char matrix representing the game board. 'M' represents an unrevealed mine, 'E' represents an unrevealed empty square, 'B' represents a revealed blank square that has no adjacent (above, below, left, right, and all 4 diagonals) mines, digit ('1' to '8') represents how many mines are adjacent to this revealed square, and finally 'X' represents a revealed mine.

Now given the next click position (row and column indices) among all the unrevealed squares ('M' or 'E'), return the board after revealing this position according to the following rules:

If a mine ('M') is revealed, then the game is over - change it to 'X'.

If an empty square ('E') with no adjacent mines is revealed, then change it to revealed blank ('B') and all of its adjacent unrevealed squares should be revealed recursively.

If an empty square ('E') with at least one adjacent mine is revealed, then change it to a digit ('1' to '8') representing the number of adjacent mines.

Return the board when no more squares will be revealed.

这道题就是经典的扫雷游戏啦，经典到不能再经典，从Win98开始，附件中始终存在的游戏，和纸牌、红心大战、空当接龙一起称为四大天王，曾经消耗了博主太多的时间。小时侯一直不太会玩扫雷，就是瞎点，完全不根据数字分析，每次点几下就炸了，就觉得这个游戏好无聊。后来长大了一些，慢慢的了解了游戏的玩法，才发现这个游戏果然很经典，就像破解数学难题一样，充

满了挑战与乐趣。花样百出的LeetCode这次把扫雷出成题，让博主借机回忆了一把小时候，不错不错，那么来做题吧。题目中图文并茂，相信就算是没玩过扫雷的也能弄懂了，而且规则也说的比较详尽了，那么我们相对应的做法也就明了了。对于当前需要点击的点，我们先判断是不是雷，是的话直接标记X返回即可。如果不是的话，我们就数该点周围的雷个数，如果周围有雷，则当前点变为雷的个数并返回。如果没有的话，我们再对周围所有的点调用递归函数再点击即可。参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<vector<char>> updateBoard(vector<vector<char>>& board, vector<int>& click) {
4         if (board.empty() || board[0].empty()) return {};
5         int m = board.size(), n = board[0].size(), row = click[0], col = click[1], cnt = 0;
6         if (board[row][col] == 'M') {
7             board[row][col] = 'X';
8         } else {
9             for (int i = -1; i < 2; ++i) {
10                 for (int j = -1; j < 2; ++j) {
11                     int x = row + i, y = col + j;
12                     if (x < 0 || x >= m || y < 0 || y >= n) continue;
13                     if (board[x][y] == 'M') ++cnt;
14                 }
15             }
16             if (cnt > 0) {
17                 board[row][col] = cnt + '0';
18             } else {
19                 board[row][col] = 'B';
20                 for (int i = -1; i < 2; ++i) {
21                     for (int j = -1; j < 2; ++j) {
22                         int x = row + i, y = col + j;
23                         if (x < 0 || x >= m || y < 0 || y >= n) continue;
24                         if (board[x][y] == 'E') {
25                             vector<int> nextPos{x, y};
26                             updateBoard(board, nextPos);
27                         }
28                     }
29                 }
30             }
31         }
32     }
33     return board;
34 }

```

CPP

下面这种解法跟上面的解法思路基本一样，写法更简洁了一些。可以看出上面的解法中的那两个for循环出现了两次，这样显得代码比较冗余，一般来说对于重复代码是要抽离成函数的，但那样还要多加个函数，也麻烦。我们可以根据第一次找周围雷个数的时候，若此时cnt个数为0并且标识是E的位置记录下来，那么如果最后雷个数确实为0了的话，我们直接遍历我们保存下来为E的位置调用递归函数即可，就不用再写两个for循环了，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<vector<char>> updateBoard(vector<vector<char>>& board, vector<int>& click) {
4         if (board.empty() || board[0].empty()) return {};
5         int m = board.size(), n = board[0].size(), row = click[0], col = click[1], cnt = 0;
6         if (board[row][col] == 'M') {
7             board[row][col] = 'X';
8         } else {
9             vector<vector<int>> neighbors;
10            for (int i = -1; i < 2; ++i) {
11                for (int j = -1; j < 2; ++j) {
12                    int x = row + i, y = col + j;
13                    if (x < 0 || x >= m || y < 0 || y >= n) continue;
14                    if (board[x][y] == 'M') ++cnt;
15                    else if (cnt == 0 && board[x][y] == 'E') neighbors.push_back({x, y});
16                }
17            }
18            if (cnt > 0) {
19                board[row][col] = cnt + '0';
20            } else {
21                for (auto a : neighbors) {
22                    board[a[0]][a[1]] = 'B';
23                    updateBoard(board, a);
24                }
25            }
26        }
27        return board;
28    }
29};

```

下面这种方法是上面方法的迭代写法，用queue来存储之后要遍历的位置，这样就不用递归调用函数了，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     vector<vector<char>> updateBoard(vector<vector<char>>& board, vector<int>& click) {
4         if (board.empty() || board[0].empty()) return {};
5         int m = board.size(), n = board[0].size();
6         queue<pair<int, int>> q({{click[0], click[1]}});
7         while (!q.empty()) {
8             int row = q.front().first, col = q.front().second, cnt = 0; q.pop();
9             vector<pair<int, int>> neighbors;
10            if (board[row][col] == 'M') board[row][col] = 'X';
11            else {
12                for (int i = -1; i < 2; ++i) {
13                    for (int j = -1; j < 2; ++j) {
14                        int x = row + i, y = col + j;
15                        if (x < 0 || x >= m || y < 0 || y >= n) continue;
16                        if (board[x][y] == 'M') ++cnt;
17                        else if (cnt == 0 && board[x][y] == 'E') neighbors.push_back({x,
18 y});
19                    }
20                }
21            }
22            if (cnt > 0) board[row][col] = cnt + '0';
23            else {
24                for (auto a : neighbors) {
25                    board[a.first][a.second] = 'B';
26                    q.push(a);
27                }
28            }
29        }
30        return board;
31    }
32 };

```

522. 二叉搜索树的最小绝对差

Given a binary search tree with non-negative values, find the minimum absolute difference between values of any two nodes.

Example:

Input:

```

1
 \
 3
 /
2

```

Output:

```
1
```

这道题给了我们一棵二叉搜索树，让我们求任意个节点值之间的最小绝对差。由于BST的左<根<右的性质可知，如果按照中序遍历会得到一个有序数组，那么最小绝对差肯定在相邻的两个节点值之间产生。所以我们的做法就是对BST进行中序遍历，然后当前节点值和之前节点值求绝对差并更新结果res。这里需要注意的就是在处理第一个节点值时，由于其没有前节点，所以不能求绝对差。这里我们用变量pre来表示前节点值，这里由于题目中说明了所以节点值不为负数，所以我们给pre初始化-1，这样我们就知道pre是否存在。如果没有题目中的这个非负条件，那么就不能用int变量来，必须要用指针，通过来判断是否为指向空来判断前结点是否存在。还好这里简化了问题，用-1就能搞定了，这里我们先来看中序遍历的递归写法，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     int getMinimumDifference(TreeNode* root) {
4         int res = INT_MAX, pre = -1;
5         inorder(root, pre, res);
6         return res;
7     }
8     void inorder(TreeNode* root, int& pre, int& res) {
9         if (!root) return;
10        inorder(root->left, pre, res);
11        if (pre != -1) res = min(res, root->val - pre);
12        pre = root->val;
13        inorder(root->right, pre, res);
14    }
15 };

```

CPP

其实我们也不必非要用中序遍历不可，用先序遍历同样可以利用到BST的性质，我们带两个变量low和high来分别表示上下界，初始化为int的极值，然后我们在递归函数中，分别用上下界和当前节点值的绝对差来更新结果res，参见代码如下：

解法2:

```

1 class Solution {
2 public:
3     int getMinimumDifference(TreeNode* root) {
4         int res = INT_MAX;
5         helper(root, INT_MIN, INT_MAX, res);
6         return res;
7     }
8     void helper(TreeNode* root, int low, int high, int& res) {
9         if (!root) return;
10        if (low != INT_MIN) res = min(res, root->val - low);
11        if (high != INT_MAX) res = min(res, high - root->val);
12        helper(root->left, low, root->val, res);
13        helper(root->right, root->val, high, res);
14    }
15 };

```

CPP

下面这种方法是解法一的迭代的写法，思路跟之前的解法没有什么区别，参见代码如下：

解法3:

```

1 class Solution {
2 public:
3     int getMinimumDifference(TreeNode* root) {
4         int res = INT_MAX, pre = -1;
5         stack<TreeNode*> st;
6         TreeNode *p = root;
7         while (p || !st.empty()) {
8             while (p) {
9                 st.push(p);
10                p = p->left;
11            }
12            p = st.top(); st.pop();
13            if (pre != -1) res = min(res, p->val - pre);
14            pre = p->val;
15            p = p->right;
16        }
17        return res;
18    }
19 };

```

523. 孤独的像素之一

Given a picture consisting of black and white pixels, find the number of black lonely pixels.

The picture is represented by a 2D char array consisting of 'B' and 'W', which means black and white pixels respectively.

A black lonely pixel is character 'B' that located at a specific position where the same row and same column don't have any other black pixels.

Example:

Input:

```
[['W', 'W', 'B'],
 ['W', 'B', 'W'],
 ['B', 'W', 'W']]
```

Output: 3

这道题定义了一种孤独的黑像素，就是该黑像素所在的行和列中没有其他的黑像素，让我们找出所有的这样的像素。那么既然对于每个黑像素都需要查找其所在的行和列，为了避免重复查找，我们可以统一的扫描一次，将各行各列的黑像素的个数都统计出来，然后再扫描所有的黑像素一次，看其是否是该行该列唯一的存在，是的话就累加计数器即可，参见代码如下：

```

1 class Solution {
2 public:
3     int findLonelyPixel(vector<vector<char>>& picture) {
4         if (picture.empty() || picture[0].empty()) return 0;
5         int m = picture.size(), n = picture[0].size(), res = 0;
6         vector<int> rowCnt(m, 0), colCnt(n, 0);
7         for (int i = 0; i < m; ++i) {
8             for (int j = 0; j < n; ++j) {
9                 if (picture[i][j] == 'B') {
10                     ++rowCnt[i];
11                     ++colCnt[j];
12                 }
13             }
14         }
15         for (int i = 0; i < m; ++i) {
16             for (int j = 0; j < n; ++j) {
17                 if (picture[i][j] == 'B') {
18                     if (rowCnt[i] == 1 && colCnt[j] == 1) {
19                         ++res;
20                     }
21                 }
22             }
23         }
24     }
25     return res;
26 }

```

524. 数组中差为K的数对

Given an array of integers and an integer k, you need to find the number of unique k-diff pairs in the array. Here a k-diff pair is defined as an integer pair (i, j), where i and j are both numbers in the array and their absolute difference is k.

Example 1:

Input: [3, 1, 4, 1, 5], k = 2
Output: 2

这道题给了我们一个含有重复数字的无序数组，还有一个整数k，让我们找出有多少对不重复的数对(i, j)使得i和j的差刚好为k。由于k有可能为0，而只有含有至少两个相同的数字才能形成数对，那么就是说我们需要统计数组中每个数字的个数。我们可以建立每个数字和其出现次数之间的映射，然后遍历哈希表中的数字，如果k为0且该数字出现的次数大于1，则结果res自增1；如果k不为0，且用当前数字加上k后得到的新数字也在数组中存在，则结果res自增1，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int findPairs(vector<int>& nums, int k) {
4         int res = 0, n = nums.size();
5         unordered_map<int, int> m;
6         for (int num : nums) ++m[num];
7         for (auto a : m) {
8             if (k == 0 && a.second > 1) ++res;
9             if (k > 0 && m.count(a.first + k)) ++res;
10        }
11        return res;
12    }
13 };

```

下面这种方法没有使用哈希表，而是使用了双指针，需要给数组排序，节省了空间的同时牺牲了时间。我们遍历排序后的数组，然后在当前数字之后找第一个和当前数之差不小于k的数字，若这个数字和当前数字之差正好为k，那么结果res自增1，然后遍历后面的数字去掉重复数字，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int findPairs(vector<int>& nums, int k) {
4         int res = 0, n = nums.size(), j = 0;
5         sort(nums.begin(), nums.end());
6         for (int i = 0; i < n; ++i) {
7             int j = max(j, i + 1);
8             while (j < n && (long)nums[j] - nums[i] < k) ++j;
9             if (j < n && (long)nums[j] - nums[i] == k) ++res;
10            while (i < n - 1 && nums[i] == nums[i + 1]) ++i;
11        }
12        return res;
13    }
14 };

```

525. 孤独的像素之二

Given a picture consisting of black and white pixels, and a positive integer N, find the number of black pixels located at some specific row R and column C that align with all the following rules:

Row R and column C both contain exactly N black pixels.

For all rows that have a black pixel at column C, they should be exactly the same as row R
The picture is represented by a 2D char array consisting of 'B' and 'W', which means black and white pixels respectively.

这道题是之前那道Lonely Pixel I的拓展，我开始以为这次要考虑到对角线的情况，可是这次题目却完全换了一种玩法。给了一个整数N，说对于均含有N个黑像素的某行某列，如果该列中所有的黑像素所在的行都相同的话，该列的所有黑像素均为孤独的像素，让我们统计所有的这样的孤独的像素的个数。那么跟之前那题类似，我们还是要统计每一行每一列的黑像素的个数，而且由于条件二中要比较各行之间是否相等，如果一个字符一个字符的比较写起来比较麻烦，我们可以用个trick，把每行的字符连起来，形成一个字符串，然后直接比较两个字符串是否相等会简单很多。然后我们遍历每一行和每一列，如果某行和某列的黑像素刚好均为N，我们遍历该列的所有黑像素，如果其所在行均相等，则说明该列的所有黑像素均为孤独的像素，将个数加入结果res中，然后将该行的黑像素统计个数清零，以免重复运算，这样我们就可以求出所有的孤独的像素了，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int findBlackPixel(vector<vector<char>>& picture, int N) {
4         if (picture.empty() || picture[0].empty()) return 0;
5         int m = picture.size(), n = picture[0].size(), res = 0, k = 0;
6         vector<int> rowCnt(m, 0), colCnt(n, 0);
7         vector<string> rows(m, "");
8         for (int i = 0; i < m; ++i) {
9             for (int j = 0; j < n; ++j) {
10                 rows[i].push_back(picture[i][j]);
11                 if (picture[i][j] == 'B') {
12                     ++rowCnt[i];
13                     ++colCnt[j];
14                 }
15             }
16         }
17         for (int i = 0; i < m; ++i) {
18             for (int j = 0; j < n; ++j) {
19                 if (rowCnt[i] == N && colCnt[j] == N) {
20                     for (k = 0; k < m; ++k) {
21                         if (picture[k][j] == 'B') {
22                             if (rows[i] != rows[k]) break;
23                         }
24                     }
25                     if (k == m) {
26                         res += colCnt[j];
27                         colCnt[j] = 0;
28                     }
29                 }
30             }
31         }
32         return res;
33     }
34 };

```

CPP

看到论坛中的比较流行的解法是用哈希表来做的，建立黑像素出现个数为N的行和其出现次数之间的映射，然后我们就只需要统计每列的黑像素的个数，然后我们遍历哈希表，找到出现次数刚好为N的行，说明矩阵中有N个相同的该行，而且该行中的黑像素的个数也刚好为N个，那么第二个条件就已经满足了，我们只要再满足第一个条件就行了，我们在找黑像素为N个的列就行了，有几列就加几个N即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int findBlackPixel(vector<vector<char>>& picture, int N) {
4         if (picture.empty() || picture[0].empty()) return 0;
5         int m = picture.size(), n = picture[0].size(), res = 0;
6         vector<int> colCnt(n, 0);
7         unordered_map<string, int> u;
8         for (int i = 0; i < m; ++i) {
9             int cnt = 0;
10            for (int j = 0; j < n; ++j) {
11                if (picture[i][j] == 'B') {
12                    ++colCnt[j];
13                    ++cnt;
14                }
15            }
16            if (cnt == N) ++u[string(picture[i].begin(), picture[i].end())];
17        }
18        for (auto a : u) {
19            if (a.second != N) continue;
20            for (int i = 0; i < n; ++i) {
21                res += (a.first[i] == 'B' && colCnt[i] == N) ? N : 0;
22            }
23        }
24        return res;
25    }
26 };

```

526. 设计精简URL地址

How would you design a URL shortening service that is similar to TinyURL?

Background:

TinyURL is a URL shortening service where you enter a URL such as <https://leetcode.com/problems/design-tinyurl> and it returns a short URL such as <http://tinyurl.com/4e9iAk>.

Requirements:

For instance, "<http://tinyurl.com/4e9iAk>" is the tiny url for the page "<https://leetcode.com/problems/design-tinyurl>". The identifier (the highlighted part) can be any string with 6 alphanumeric characters containing 0-9, a-z, A-Z. Each shortened URL must be unique; that is, no two different URLs can be shortened to the same URL.

这道系统设计的题跟之前的算法还是不一样的，代码只是其中的一部分，估计大部分还是要跟面试官侃大山，博主也不太熟悉这类题目，还是照着ztlevi大神的帖子来写吧。

S: Scenario 场景

长URL和短URL的相互转换

N: Need 需求

- QPS (Queires Per Second) 每秒查询数
 - 日活用户: 100M
 - 每日人均使用量: (写) long2short 0.1, (读) short2long 1
 - 每日请求量: 写 10M, 读 100M
 - QPS: 一天共有86400秒, 约100K。写 100, 读 1K
 - 峰值QPS: 写 200, 读 2K

(千级的量可以用一个单SSD的MySQL机器来处理)

- Storage 存储

- 每天10M个新映射 (长URL到短URL)
 - 一个映射大约占100B的大小
- 每天1GB, 1TB大约能扛三年

对于这种系统来说，存储不是问题。只有像Netflix那样的系统可能会有存储问题。通过SN分析，我们对系统有了一个大框架印象，这个系统可以使用单SSD机器来实现。

A: API 接口

只有一种类型的服务: URLService

- Core (Business Logic) Layer
- Class: URLService
- Interface:
- URLService.encode(string long_url)
- URLService.decode(stirng short_url)
- Web Layer
- REST API:
 - GET: /{short_url}, return a http redirect response (301)
 - POST: goo.gl method - google shorten URL

Request Body: {url=longUrl} e.g. {"longUrl": "http://www.google.com/"}

Return OK(200), short_url is included in the data

K: Data Access 数据访问

Step 1: Pick a storage structure 选择一个存储结构

- SQL VS NoSQL?

- 需要支持事务Transactions吗? NoSQL不支持事务Transactions。
- 需要Rich SQL Query吗? NoSQL不支持SQL那么多的Query。
- 需要高效开发吗? 大多数的网络框架对SQL的支持性非常好, 意味着系统不需要太多的代码。
- 需要AUTO_INCREMENT ID吗? NoSQL不支持这个, 仅有一个全局唯一的Object_id。
- 需要高QPS吗? NoSQL有高性能。比如Memcached的QPS可达到百万级, MongoDB可达万级, MySQL只有千级。
- 系统的可伸缩性Scalability有多高? SQL需要开发者写代码去伸缩Scale, 而NoSQL自带该功能 (Sharding, replica)。

- Answer 回答:

- 不需要 -> NoSQL
- 不需要 -> NoSQL
- 无所谓, 因为只有很少的代码 -> NoSQL
- 算法需要AUTO_INCREMENT ID -> SQL
- 写 200, 读 2K, 不高 -> SQL
- 不高 -> SQL

- System Algorithm 系统算法

- Hash 函数

```
long_url => md5/sha1
```

- md5将一个字符串转为128位, 通常用16个字节的十六进制来表示:

```
http://site.douban.com/chuan -> c93a360dc7f3eb093ab6e304db516653
```

- sha1将字符串转为160位, 通常用20个字节的十六进制来表示:

```
http://site.douban.com/chuan -> dff85871a72c73c3eae09e39ffe97aea63047094
```

这两个算法使得哈希值是随机分布的, 但是冲突Conflicts无法避免。任何哈希算法都无法避免冲突问题。

- 优点: 简单。我们用转换字符串的前6个字符

- 缺点: 冲突

解决方法 1. 使用 (long_url + timestamp) 作为哈希函数的关键字Key。2. 当冲突时, 重新生成哈希值 (生成的值不同因为时间戳改变了)。

总之, 当urls的个数超过十亿个, 可能会有大量的冲突使得系统不高效。

- base62

将short_url用62 base标记。6位可以表示 62^6 57 billion。

每个short_url表示一个十进制数，可以当作SQL数据库中的AUTO_INCREMENT ID。

```
1 class URLService {
2     public:
3         URLService() {
4             COUNTER = 1;
5             elements = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
6         }
7
8         string longToShort(string url) {
9             string short_url = base10ToBase62(COUNTER);
10            long2short[url] = COUNTER;
11            short2long[COUNTER] = url;
12            ++COUNTER;
13            return "http://tiny.url/" + short_url;
14        }
15
16        string shortToLong(string url) {
17            string prefix = "http://tiny.url/";
18            url = url.substr(prefix.size());
19            int n = base62ToBase10(url);
20            return short2long[n];
21        }
22
23        int base62ToBase10(string s) {
24            int n = 0;
25            for (int i = 0; i < s.size(); ++i) {
26                n = n * 62 + convert(s[i]);
27            }
28            return n;
29        }
30
31        int convert(char c) {
32            if (c >= '0' && c <= '9') {
33                return c - '0';
34            } else if (c >= 'a' && c <= 'z') {
35                return c - 'a' + 10;
36            } else if (c >= 'A' && c <= 'Z') {
37                return c - 'A' + 36;
38            }
39            return -1;
40        }
41
42        string base10ToBase62(int n) {
43            string str = "";
44            while (n != 0) {
45                str.insert(str.begin(), elements[n % 62]);
46                n /= 62;
47            }
48            while (str.size() != 6) {
49                str.insert(str.begin(), '0');
50            }
51            return str;
52        }
53
54    private:
55        unordered_map<string, int> long2short;
56        unordered_map<int, string> short2long;
57        int COUNTER;
58        string elements;
59    };
}
```

Step 2: Database Schema 数据库概要

一个表 (id, long_url) 。id是主键，通过long_url排序。基本的系统架构为：

Browser <-> Web <-> Core <-> DB

O: Optimize 优化

如何提高响应速度？

- 在网络服务器和数据库之间提高响应速度

使用Memcached来提高响应速度。当获得long_url时，先在缓存中搜索。我们可以把90%的读请求放在缓存当中。

- 在网络服务器和用户浏览器之间提高响应速度

不同的地区使用不同的网络服务器和缓存服务器。所有的地区共享一个数据库用来匹配用户到最近的网络服务器（通过DNS），当他们不在缓存中的时候。

如果我们需要多于一台的MySQL机器？

- 问题：

- 缓存用完了
- 越来越多的请求
- 越来越多的缓存丢失

- 解决方案：

- 垂直切分 Vertical Sharding
- 水平切分 Horizontal Sharding

最好的方式是水平切分。当前的表结构是 (id, long_url)，哪列可以当作切分关键字。

一个简单的方法是id模块切分。

现在有另一个问题：如何能使多个机器共享一个全局的AUTO_INCREMENT ID？

两种方法：1. 多使用一个机器去维护id。2. 使用zookeeper。都很操蛋。

所以，我们不适用AUTO_INCREMENT ID

好处是将切分关键字当作short_url的第一个字节。

另一种方法是用统一的哈希将循环分成62份。有多少份并没有啥关系，因为可能并没有62台机器（可能有360或其他的）。每台机器都是为循环的一部分的服务负责。

```
write long_url -> hash(long_url)%62 -> put long_url to the specific machine according to hash value -> generate short_url on this machine -> return short_url
```

```
short_url request -> get the sharding key (first byte of the short_url) -> search in the corresponding machine based on sharding key -> return long_url
```

每当我们增加一台新机器，将最多使用的机器的一半范围放到新的机器中。

更多优化

将中文服务器放在中国，美国的服务器放在美国。使用地理信息当作切分关键字，例如，0是中国的网站，1是美国的网站。

527. 编码和解码精简URL地址

TinyURL is a URL shortening service where you enter a URL such as <https://leetcode.com/problems/design-tinyurl> and it returns a short URL such as <http://tinyurl.com/4e9iAk>.

Design the encode and decode methods for the TinyURL service. There is no restriction on how your encode/decode algorithm should work. You just need to ensure that a URL can be encoded to a tiny URL and the tiny URL can be decoded to the original URL.

这道题让我们编码和解码精简URL地址，这其实很有用，因为有的链接地址特别的长，就很烦，如果能精简成固定的长度，就很清爽。最简单的一种编码就是用个计数器，当前是第几个存入的url就编码成几，然后解码的时候也能根据数字来找到原来的url，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3
4     // Encodes a URL to a shortened URL.
5     string encode(string longUrl) {
6         url.push_back(longUrl);
7         return "http://tinyurl.com/" + to_string(url.size() - 1);
8     }
9
10    // Decodes a shortened URL to its original URL.
11    string decode(string shortUrl) {
12        auto pos = shortUrl.find_last_of("/");
13        return url[stoi(shortUrl.substr(pos + 1))];
14    }
15
16 private:
17     vector<string> url;
18 };

```

CPP

上面这种方法虽然简单，但是缺点却很多，首先，如果接收到多次同一url地址，仍然会当做不同的url来处理。当然这个缺点可以通过将vector换成哈希表，每次先查找url是否已经存在。虽然这个缺点可以克服掉，但是由于是用计数器编码，那么当前服务器存了多少url就暴露出来了，也许会有安全隐患。而且计数器编码另一个缺点就是数字会不断的增大，那么编码的长度也就不是确定的了。而题目中明确推荐了使用六位随机字符串来编码，那么我们只要在所有大小写字母和数字中随机产生6个字符就可以了，我们用哈希表建立6位字符串和url之间的映射，如果随机生成的字符之前已经存在了，我们就继续随机生成新的字符串，直到生成了之前没有的字符串为止。下面的代码中使用了两个哈希表，目的是为了建立六位随机字符串和url之间的相互映射，这样进来大量的相同url时，就不用生成新的随机字符串了。当然，不加这个功能也能通过OJ，这道题的OJ基本上是形同虚设，两个函数分别直接返回参数字符串也能通过OJ，囧~

解法2：

```

1 class Solution {
2 public:
3     Solution() {
4         dict = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
5         short2long.clear();
6         long2short.clear();
7         srand(time(NULL));
8     }
9
10    // Encodes a URL to a shortened URL.
11    string encode(string longUrl) {
12        if (long2short.count(longUrl)) {
13            return "http://tinyurl.com/" + long2short[longUrl];
14        }
15        int idx = 0;
16        string randStr;
17        for (int i = 0; i < 6; ++i) randStr.push_back(dict[rand() % 62]);
18        while (short2long.count(randStr)) {
19            randStr[idx] = dict[rand() % 62];
20            idx = (idx + 1) % 5;
21        }
22        short2long[randStr] = longUrl;
23        long2short[longUrl] = randStr;
24        return "http://tinyurl.com/" + randStr;
25    }
26
27    // Decodes a shortened URL to its original URL.
28    string decode(string shortUrl) {
29        string randStr = shortUrl.substr(shortUrl.find_last_of("/") + 1);
30        return short2long.count(randStr) ? short2long[randStr] : shortUrl;
31    }
32
33 private:
34     unordered_map<string, string> short2long, long2short;
35     string dict;
36 };

```

528. 从字符串创建二叉树

You need to construct a binary tree from a string consisting of parenthesis and integers.

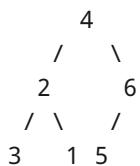
The whole input represents a binary tree. It contains an integer followed by zero, one or two pairs of parenthesis. The integer represents the root's value and a pair of parenthesis contains a child binary tree with the same structure.

You always start to construct the left child node of the parent first if it exists.

Example:

Input: "4(2(3)(1))(6(5))"

Output: return the tree root node representing the following tree:



这道题让我们根据一个字符串来创建一个二叉树，其中结点与其左右子树是用括号隔开，每个括号中又是数字后面跟括号的模式，这种模型就很有递归的感觉，所以我们当然可以使用递归来实现。首先我们要做的是先找出根结点值，我们找第一个左括号的位置，如果找不到，说明当前字符串都是数字，直接转化为整型，然后新建结点返回即可。否则的话从当前位置开始遍历，因为当前位置是一个左括号，我们的目标是找到与之对应的右括号的位置，但是由于中间还会遇到左右括号，所以我们需要用一个变量cnt来记录左括号的个数，如果遇到左括号，cnt自增1，如果遇到右括号，cnt自减1，这样当某个时刻cnt为0的时候，我们就确定了一个完整的子树的位置，那么问题来了，这个子树到底是左子树还是右子树呢，我们需要一个辅助变量start，当最开始找到第一个左括号的位置时，将start赋值为该位置，那么当cnt为0时，如果start还是原来的位置，说明这个是左子树，我们对其进行递归函数，注意此时更新start的位置，这样就能区分左右子树了，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     TreeNode* str2tree(string s) {
4         if (s.empty()) return NULL;
5         auto found = s.find('(');
6         int val = (found == string::npos) ? stoi(s) : stoi(s.substr(0, found));
7         TreeNode *cur = new TreeNode(val);
8         if (found == string::npos) return cur;
9         int start = found, cnt = 0;
10        for (int i = start; i < s.size(); ++i) {
11            if (s[i] == '(') ++cnt;
12            else if (s[i] == ')') --cnt;
13            if (cnt == 0 && start == found) {
14                cur->left = str2tree(s.substr(start + 1, i - start - 1));
15                start = i + 1;
16            } else if (cnt == 0) {
17                cur->right = str2tree(s.substr(start + 1, i - start - 1));
18            }
19        }
20        return cur;
21    }
22 };

```

CPP

下面这种解法使用迭代来实现，借助栈stack来实现。遍历字符串s，用变量j记录当前位置i，然后看当前遍历到的字符是什么，如果遇到的是左括号，什么也不做继续遍历；如果遇到的是数字或者负号，那么我们将连续的数字都找出来，然后转为整型并新建结点，此时我们看stack中是否有结点，如果有的话，当前结点就是栈顶结点的子结点，如果栈顶结点没有左子结点，那么此结点就是其左子结点，反之则为其右子结点。之后要将此结点压入栈中。如果我们遍历到的是右括号，说明栈顶元素的子结点已经处理完了，将其移除栈，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     TreeNode* str2tree(string s) {
4         if (s.empty()) return NULL;
5         stack<TreeNode*> st;
6         for (int i = 0; i < s.size(); ++i) {
7             int j = i;
8             if (s[i] == ')') st.pop();
9             else if ((s[i] >= '0' && s[i] <= '9') || s[i] == '-') {
10                 while (i + 1 < s.size() && s[i + 1] >= '0' && s[i + 1] <= '9') ++i;
11                 TreeNode *cur = new TreeNode(stoi(s.substr(j, i - j + 1)));
12                 if (!st.empty()) {
13                     TreeNode *t = st.top();
14                     if (!t->left) t->left = cur;
15                     else t->right = cur;
16                 }
17                 st.push(cur);
18             }
19         }
20         return st.top();
21     }
22 };

```

529. 复数相乘

Given two strings representing two complex numbers.

You need to return a string representing their multiplication. Note $i^2 = -1$ according to the definition.

Example 1:

Input: "1+1i", "1+1i"

Output: "0+2i"

Explanation: $(1 + i) * (1 + i) = 1 + i$

这道题让我们求复数的乘法，有关复数的知识最早还是在本科的复变函数中接触到的，难起来还真是难。但是这里只是最简单的乘法，只要利用好定义 $i^2=-1$ 就可以解题，而且这道题的另一个考察点其实是对字符的处理，我们需要把字符串中的实部和虚部分离开并进行运算，那么我们可以用STL中自带的`find_last_of`函数来找到加号的位置，然后分别拆出实部虚部，进行运算后再变回字符串，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     string complexNumberMultiply(string a, string b) {
4         int n1 = a.size(), n2 = b.size();
5         auto p1 = a.find_last_of("+"), p2 = b.find_last_of("+");
6         int a1 = stoi(a.substr(0, p1)), b1 = stoi(b.substr(0, p2));
7         int a2 = stoi(a.substr(p1 + 1, n1 - p1 - 2));
8         int b2 = stoi(b.substr(p2 + 1, n2 - p2 - 2));
9         int r1 = a1 * b1 - a2 * b2, r2 = a1 * b2 + a2 * b1;
10        return to_string(r1) + "+" + to_string(r2) + "i";
11    }
12 };

```

下面这种方法利用到了字符串流类`istringstream`来读入字符串，直接将实部虚部读入`int`变量中，注意中间也要把加号读入`char`变量中，然后再进行运算即可，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     string complexNumberMultiply(string a, string b) {
4         istringstream is1(a), is2(b);
5         int a1, a2, b1, b2, r1, r2;
6         char plus;
7         is1 >> a1 >> plus >> a2;
8         is2 >> b1 >> plus >> b2;
9         r1 = a1 * b1 - a2 * b2, r2 = a1 * b2 + a2 * b1;
10        return to_string(r1) + "+" + to_string(r2) + "i";
11    }
12};
```

CPP

下面这种解法实际上是C语言的解法，用到了`sscanf`这个读入字符串的函数，需要把`string`转为`cost char*`型，然后标明读入的方式和类型，再进行运算即可，参见代码如下：

解法3：

```
1 class Solution {
2 public:
3     string complexNumberMultiply(string a, string b) {
4         int a1, a2, b1, b2, r1, r2;
5         sscanf(a.c_str(), "%d+%di", &a1, &a2);
6         sscanf(b.c_str(), "%d+%di", &b1, &b2);
7         r1 = a1 * b1 - a2 * b2, r2 = a1 * b2 + a2 * b1;
8         return to_string(r1) + "+" + to_string(r2) + "i";
9     }
10};
```

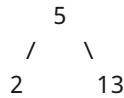
CPP

530. 将二叉搜索树BST转为较大树

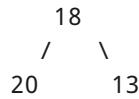
Given a Binary Search Tree (BST), convert it to a Greater Tree such that every key of the original BST is changed to the original key plus sum of all keys greater than the original key in BST.

Example:

Input: The root of a Binary Search Tree like this:



Output: The root of a Greater Tree like this:



这道题让我们将二叉搜索树转为较大树，通过题目汇总的例子可以明白，是把每个结点值加上所有比它大的结点值总和当作新的结点值。仔细观察题目中的例子可以发现，2变成了20，而20是所有结点之和，因为2是最小结点值，要加上其他所有结点值，所以肯定就是所有结点值之和。5变成了18，是通过20减去2得来的，而13还是13，是由20减去7得来的，而7是2和5之和。我开

始想的方法是先求出所有结点值之和，然后开始中序遍历数组，同时用变量sum来记录累加和，根据上面分析的规律来更新所有的数组。但是通过看论坛，发现还有更巧妙的方法，不用先求出的所有结点值之和，而是巧妙的将中序遍历左根右的顺序逆过来，变成右根左的顺序，这样就可以反向计算累加和sum，同时更新结点值，叼的不行，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     TreeNode* convertBST(TreeNode* root) {
4         int sum = 0;
5         helper(root, sum);
6         return root;
7     }
8     void helper(TreeNode*& node, int& sum) {
9         if (!node) return;
10        helper(node->right, sum);
11        node->val += sum;
12        sum = node->val;
13        helper(node->left, sum);
14    }
15 }

```

CPP

下面这种方法写的更加简洁一些，没有写其他递归函数，而是把自身写成了可以递归调用的函数，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     TreeNode* convertBST(TreeNode* root) {
4         if (!root) return NULL;
5         convertBST(root->right);
6         root->val += sum;
7         sum = root->val;
8         convertBST(root->left);
9         return root;
10    }
11
12 private:
13     int sum = 0;
14 }

```

CPP

下面这种解法是迭代的写法，因为中序遍历有递归和迭代两种写法，逆中序遍历同样也可以写成迭代的形式，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     TreeNode* convertBST(TreeNode* root) {
4         if (!root) return NULL;
5         int sum = 0;
6         stack<TreeNode*> st;
7         TreeNode *p = root;
8         while (p || !st.empty()) {
9             while (p) {
10                 st.push(p);
11                 p = p->right;
12             }
13             p = st.top(); st.pop();
14             p->val += sum;
15             sum = p->val;
16             p = p->left;
17         }
18         return root;
19     }
20 };

```

531. 最短时间差

Given a list of 24-hour clock time points in "Hour:Minutes" format, find the minimum minutes difference between any two time points in the list.

Example 1:

Input: ["23:59","00:00"]

Output: 1

这道题给了我们一系列无序的时间点，让我们求最短的两个时间点之间的差值。那么最简单直接的办法就是给数组排序，这样时间点小的就在前面了，然后我们分别把小时和分钟提取出来，计算差值，注意唯一的特殊情况就是第一个和末尾的时间点进行比较，第一个时间点需要加上24小时再做差值，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     int findMinDifference(vector<string>& timePoints) {
4         int res = INT_MAX, n = timePoints.size(), diff = 0;
5         sort(timePoints.begin(), timePoints.end());
6         for (int i = 0; i < n; ++i) {
7             string t1 = timePoints[i], t2 = timePoints[(i + 1) % n];
8             int h1 = (t1[0] - '0') * 10 + t1[1] - '0';
9             int m1 = (t1[3] - '0') * 10 + t1[4] - '0';
10            int h2 = (t2[0] - '0') * 10 + t2[1] - '0';
11            int m2 = (t2[3] - '0') * 10 + t2[4] - '0';
12            diff = (h2 - h1) * 60 + (m2 - m1);
13            if (i == n - 1) diff += 24 * 60;
14            res = min(res, diff);
15        }
16        return res;
17    }
18 };

```

下面这种写法跟上面的大体思路一样，写法上略有不同，是在一开始就把小时和分钟数提取出来并计算总分钟数存入一个新数组，然后再对新数组进行排序，再计算两两之差，最后还是要处理首尾之差，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     int findMinDifference(vector<string>& timePoints) {
4         int res = INT_MAX, n = timePoints.size();
5         vector<int> nums;
6         for (string str : timePoints) {
7             int h = stoi(str.substr(0, 2)), m = stoi(str.substr(3));
8             nums.push_back(h * 60 + m);
9         }
10        sort(nums.begin(), nums.end());
11        for (int i = 1; i < n; ++i) {
12            res = min(res, nums[i] - nums[i - 1]);
13        }
14        return min(res, 1440 + nums[0] - nums.back());
15    }
16};
```

CPP

上面两种方法的时间复杂度都是 $O(nlgn)$ ，我们来看一种 $O(n)$ 时间复杂度的方法，由于时间点并不是无穷多个，而是只有1440个，所以我们建立一个大小为1440的数组来标记某个时间点是否出现过，如果之前已经出现过，说明有两个相同的时间点，直接返回0即可；若没有，将当前时间点标记为出现过。我们还需要一些辅助变量，pre表示之前遍历到的时间点，first表示按顺序排的第一个时间点，last表示按顺序排的最后一个时间点，然后我们再遍历这个mask数组，如果当前时间点出现过，再看如果first不为初始值的话，说明pre已经被更新过了，我们用当前时间点减去pre来更新结果res，然后再分别更新first，last，和pre即可，参见代码如下：

解法3：

```
1 class Solution {
2 public:
3     int findMinDifference(vector<string>& timePoints) {
4         int res = INT_MAX, pre = 0, first = INT_MAX, last = INT_MIN;
5         vector<int> mask(1440, 0);
6         for (string str : timePoints) {
7             int h = stoi(str.substr(0, 2)), m = stoi(str.substr(3));
8             if (mask[h * 60 + m] == 1) return 0;
9             mask[h * 60 + m] = 1;
10        }
11        for (int i = 0; i < 1440; ++i) {
12            if (mask[i] == 1) {
13                if (first != INT_MAX) {
14                    res = min(res, i - pre);
15                }
16                first = min(first, i);
17                last = max(last, i);
18                pre = i;
19            }
20        }
21        return min(res, 1440 + first - last);
22    }
23};
```

CPP

532. 有序数组中的单独元素

Given a sorted array consisting of only integers where every element appears twice except for one element which appears once. Find this single element that appears only once.

Example 1:

Input: [1,1,2,3,3,4,4,8,8]

Output: 2

这道题给我们了一个有序数组，说是所有的元素都出现了两次，除了一个元素，让我们找到这个元素。如果没有时间复杂度的限制，我们可以用多种方法来做，最straightforward的解法就是用个双指针，每次检验两个，就能找出落单的。也可以像Single Number里的方法那样，将所有数字亦或起来，相同的数字都会亦或成0，剩下就是那个落单的数字。那么由于有了时间复杂度的限制，需要为 $O(\log n)$ ，而数组又是有序的，不难想到要用二分搜索法来做。二分搜索法的难点在于折半了以后，如何判断将要去哪个分支继续搜索，而这道题确实判断条件不明显，比如下面两个例子：

1 1 2 2 3

1 2 2 3 3

这两个例子初始化的时候 $left=0$, $right=4$ 一样， mid 算出来也一样为2，但是他们要去的方向不同，如何区分出来呢？仔细观察我们可以发现，如果当前数字出现两次的话，我们可以通过数组的长度跟当前位置的关系，计算出右边和当前数字不同的数字的总个数，如果是偶数个，说明落单数在左半边，反之则在右半边。有了这个规律就可以写代码了，为啥我们直接就能跟 $mid+1$ 比呢，不怕越界吗？当然不会，因为 $left$ 如何跟 $right$ 相等，就不会进入循环，所以 mid 一定会比 $right$ 小，一定会有 $mid+1$ 存在。当然 mid 是有可能为0的，所以此时当 mid 和 $mid+1$ 的数字不等时，我们直接返回 mid 的数字就可以了，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     int singleNonDuplicate(vector<int>& nums) {
4         int left = 0, right = nums.size() - 1, n = nums.size();
5         while (left < right) {
6             int mid = left + (right - left) / 2;
7             if (nums[mid] == nums[mid + 1]) {
8                 if ((n - 1 - mid) % 2 == 1) right = mid;
9                 else left = mid + 1;
10            } else {
11                if (mid == 0 || nums[mid] != nums[mid - 1]) return nums[mid];
12                if ((n - 1 - mid) % 2 == 0) right = mid;
13                else left = mid + 1;
14            }
15        }
16        return nums[left];
17    }
18 }
```

CPP

下面这种解法是对上面的分支进行合并，使得代码非常的简洁。使用到了亦或1这个小技巧，为什么要亦或1呢，原来我们可以将坐标两两归为一对，比如0和1, 2和3, 4和5等等。而亦或1可以直接找到你的小伙伴，比如对于2，亦或1就是3，对于3，亦或1就是2。如果你和你的小伙伴相等了，说明落单数在右边，如果不等，说明在左边，这方法，太叼了有木有，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int singleNonDuplicate(vector<int>& nums) {
4         int left = 0, right = nums.size() - 1;
5         while (left < right) {
6             int mid = left + (right - left) / 2;
7             if (nums[mid] == nums[mid ^ 1]) left = mid + 1;
8             else right = mid;
9         }
10        return nums[left];
11    }
12 };

```

下面这种解法其实跟上面的方法其实有些类似，虽然没有亦或1，但是将right缩小了一倍，但是在比较的时候，是比较mid*2和mid*2+1的关系的，这样还是能正确的比较原本应该相等的两个小伙伴的值的，其实核心思路和上面一样，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int singleNonDuplicate(vector<int>& nums) {
4         int left = 0, right = nums.size() / 2;
5         while (left < right) {
6             int mid = left + (right - left) / 2;
7             if (nums[mid * 2] == nums[mid * 2 + 1]) left = mid + 1;
8             else right = mid;
9         }
10        return nums[left * 2];
11    }
12 };

```

下面这种方法其实跟解法二很像，没有用亦或1，但是对mid进行了处理，强制使其成为小伙伴对儿中的第一个位置，然后跟另一个小伙伴比较大小，参见代码如下：

解法4：

```

1 class Solution {
2 public:
3     int singleNonDuplicate(vector<int>& nums) {
4         int left = 0, right = nums.size() - 1;
5         while (left < right) {
6             int mid = left + (right - left) / 2;
7             if (mid % 2 == 1) --mid;
8             if (nums[mid] == nums[mid + 1]) left = mid + 2;
9             else right = mid;
10        }
11        return nums[left];
12    }
13 };

```

533. 翻转字符串之二

Given a string and an integer k, you need to reverse the first k characters for every $2k$ characters counting from the start of the string. If there are less than k characters left, reverse all of them. If there are less than $2k$ but greater than or equal to k characters, then reverse the first k characters and left the other as original.

Example:

Input: s = "abcdefg", k = 2
Output: "bacdfeg"

这道题是之前那道题Reverse String的拓展，同样是翻转字符串，但是这里是每隔 k 隔字符，翻转 k 个字符，最后如果不够 k 个了的话，剩几个就翻转几个。比较直接的方法就是先用 n / k 算出来原字符串s能分成几个长度为 k 的字符串，然后开始遍历这些字符串，遇到 2 的倍数就翻转，翻转的时候注意考虑下是否已经到s末尾了，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     string reverseStr(string s, int k) {
4         int n = s.size(), cnt = n / k;
5         for (int i = 0; i <= cnt; ++i) {
6             if (i % 2 == 0) {
7                 if (i * k + k < n) {
8                     reverse(s.begin() + i * k, s.begin() + i * k + k);
9                 } else {
10                     reverse(s.begin() + i * k, s.end());
11                 }
12             }
13         }
14         return s;
15     }
16 }
```

CPP

在论坛里又发现了写法更为简洁的方法，就是每 $2k$ 个字符来遍历原字符串s，然后进行翻转，翻转的结尾位置是取 $i+k$ 和末尾位置之间的较小值，感觉很叼，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     string reverseStr(string s, int k) {
4         for (int i = 0; i < s.size(); i += 2 * k) {
5             reverse(s.begin() + i, min(s.begin() + i + k, s.end()));
6         }
7         return s;
8     }
9 }
```

CPP

534. 零一矩阵

Given a matrix consists of 0 and 1, find the distance of the nearest 0 for each cell.

The distance between two adjacent cells is 1.

这道题给了我们一个只有0和1的矩阵，让我们求每一个1到离其最近的0的距离，其实也就是求一个距离场，而求距离场那么BFS将是不二之选。刚看到此题时，我以为这跟之前那道Shortest Distance from All Buildings是一样的，从每一个0开始遍历，不停的更新每一个1的距离，但是这样写下来TLE了。后来我又改变思路，从每一个1开始BFS，找到最近的0，结果还是TLE，气死人。后来逛论坛发现思路是对的，就是写法上可以进一步优化，我们可以首先遍历一次矩阵，将值为0的点都存入queue，将值为1的点改为INT_MAX。之前像什么遍历迷宫啊，起点只有一个，而这道题所有为0的点都是起点，这想法，呵！然后开始BFS遍历，从queue中取出一个数字，遍历其周围四个点，如果越界或者周围点的值小于等于当前值，则直接跳过。因为周围点的距离更小的话，就没有更新的必要，否则将周围点的值更新为当前值加1，然后把周围点的坐标加入queue，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<vector<int>> updateMatrix(vector<vector<int>>& matrix) {
4         int m = matrix.size(), n = matrix[0].size();
5         vector<vector<int>> dirs{{0,-1},{-1,0},{0,1},{1,0}};
6         queue<pair<int, int>> q;
7         for (int i = 0; i < m; ++i) {
8             for (int j = 0; j < n; ++j) {
9                 if (matrix[i][j] == 0) q.push({i, j});
10                else matrix[i][j] = INT_MAX;
11            }
12        }
13        while (!q.empty()) {
14            auto t = q.front(); q.pop();
15            for (auto dir : dirs) {
16                int x = t.first + dir[0], y = t.second + dir[1];
17                if (x < 0 || x >= m || y < 0 || y >= n ||
18                    matrix[x][y] <= matrix[t.first][t.second]) continue;
19                matrix[x][y] = matrix[t.first][t.second] + 1;
20                q.push({x, y});
21            }
22        }
23        return matrix;
24    }
25 };

```

CPP

下面这种解法是参考的qswawrq大神的帖子，他想出了一种二次扫描的解法，从而不用使用BFS了。这种解法也相当的巧妙，我们首先建立一个和matrix大小相等的矩阵res，初始化为很大的值，这里我们用INT_MAX-1，为甚么要减1呢，后面再说。然后我们遍历matrix矩阵，当遇到为0的位置，我们将结果res矩阵的对应位置也设为0，这make sense吧，就不多说了。然后就是这个解法的精髓了，如果不是0的地方，我们在第一次扫描的时候，比较其左边和上边的位置，取其中较小的值，再加上1，来更新结果res中的对应位置。这里就明白了为啥我们要初始化为INT_MAX-1了吧，因为这里要加1，如果初始化为INT_MAX就会整型溢出，不过放心，由于是取较小值，res[i][j]永远不会取到INT_MAX，所以不会有再加1溢出的风险。第一次遍历我们比较了左和上的方向，那么我们第二次遍历就要比较右和下的方向，注意两种情况下我们不需要比较，一种是当值为0时，还有一种是当值为1时，这两种情况下值都不可能再变小了，所以没有更新的必要，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<vector<int>> updateMatrix(vector<vector<int>>& matrix) {
4         int m = matrix.size(), n = matrix[0].size();
5         vector<vector<int>> res(m, vector<int>(n, INT_MAX - 1));
6         for (int i = 0; i < m; ++i) {
7             for (int j = 0; j < n; ++j) {
8                 if (matrix[i][j] == 0) res[i][j] = 0;
9                 else {
10                     if (i > 0) res[i][j] = min(res[i][j], res[i - 1][j] + 1);
11                     if (j > 0) res[i][j] = min(res[i][j], res[i][j - 1] + 1);
12                 }
13             }
14         }
15         for (int i = m - 1; i >= 0; --i) {
16             for (int j = n - 1; j >= 0; --j) {
17                 if (res[i][j] != 0 && res[i][j] != 1) {
18                     if (i < m - 1) res[i][j] = min(res[i][j], res[i + 1][j] + 1);
19                     if (j < n - 1) res[i][j] = min(res[i][j], res[i][j + 1] + 1);
20                 }
21             }
22         }
23         return res;
24     }
25 };

```

在史蒂芬大神的帖子中，他提出了一种变型的方法，没有再区分左上右下，而是每次都跟左边相比，但是需要每次把矩阵旋转90度。他用python写的解法异常的简洁，貌似python中可以一行代码进行矩阵旋转，但是貌似C++没有这么叼，矩阵旋转写起来还是需要两个for循环，写出来估计也不短，这里就不写了，有兴趣的童鞋可以自己试试写一下，可以贴到留言板上哈~

535. 二叉树的直径

Given a binary tree, you need to compute the length of the diameter of the tree. The diameter of a binary tree is the length of the longest path between any two nodes in a tree. This path may or may not pass through the root.

这道题让我们求二叉树的直径，并告诉了我们直径就是两点之间的最远距离，根据题目中的例子也不难理解题意。我们再来仔细观察例子中的那两个最长路径[4,2,1,3] 和 [5,2,1,3]，我们转换一种角度来看，是不是其实就是根结点1的左右两个子树的深度之和再加1呢。那么我们只要对每一个结点求出其左右子树深度之和，再加上1就可以更新结果res了。为了减少重复计算，我们用哈希表建立每个结点和其深度之间的映射，这样某个结点的深度之前计算过了，就不用再次计算了，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int diameterOfBinaryTree(TreeNode* root) {
4         if (!root) return 0;
5         int res = getHeight(root->left) + getHeight(root->right);
6         return max(res, max(diameterOfBinaryTree(root->left), diameterOfBinaryTree(root-
7 >right)));
8     }
9     int getHeight(TreeNode* node) {
10        if (!node) return 0;
11        if (m.count(node)) return m[node];
12        int h = 1 + max(getHeight(node->left), getHeight(node->right));
13        m[node] = h;
14    }
15
16 private:
17     unordered_map<TreeNode*, int> m;
18 };

```

上面的方法貌似有两个递归函数，其实我们只需要用一个递归函数就可以了，我们再求深度的递归函数中顺便就把直径算出来了，而且貌似不用进行优化也能通过OJ，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int diameterOfBinaryTree(TreeNode* root) {
4         int res = 0;
5         maxDepth(root, res);
6         return res;
7     }
8     int maxDepth(TreeNode* node, int& res) {
9         if (!node) return 0;
10        int left = maxDepth(node->left, res);
11        int right = maxDepth(node->right, res);
12        res = max(res, left + right);
13        return max(left, right) + 1;
14    }
15 };

```

虽说不用进行优化也能通过OJ，但是毕竟还是优化一下好一点啊，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int diameterOfBinaryTree(TreeNode* root) {
4         int res = 0;
5         maxDepth(root, res);
6         return res;
7     }
8     int maxDepth(TreeNode* node, int& res) {
9         if (!node) return 0;
10        if (m.count(node)) return m[node];
11        int left = maxDepth(node->left, res);
12        int right = maxDepth(node->right, res);
13        res = max(res, left + right);
14        return m[node] = (max(left, right) + 1);
15    }
16
17 private:
18     unordered_map<TreeNode*, int> m;
19 };

```

536. 输出比赛匹配对

During the NBA playoffs, we always arrange the rather strong team to play with the rather weak team, like make the rank 1 team play with the rank nth team, which is a good strategy to make the contest more interesting. Now, you're given n teams, you need to output their final contest matches in the form of a string.

The n teams are given in the form of positive integers from 1 to n, which represents their initial rank. (Rank 1 is the strongest team and Rank n is the weakest team.) We'll use parentheses('(', ')') and commas(',') to represent the contest team pairing - parentheses('(', ')') for pairing and commas(',') for partition. During the pairing process in each round, you always need to follow the strategy of making the rather strong one pair with the rather weak one.

这道题讲的是NBA的季后赛对战顺序，对于一个看了十几年NBA的老粉来说，再熟悉不过了。这种对战顺序是为了避免强强之间过早对决，从而失去比赛的公平性，跟欧冠欧联那种八强就开始随机抽签匹配有本质上的区别。NBA的这种比赛机制基本弱队很难翻身，假如你是拿到最后一张季后赛门票进的，那么一上来就干联盟第一，肯定凶多吉少，很有可能就被横扫了。但是偶尔也会出现黑八的情况，但都是极其少见的，毕竟像勇士这么叼的球队毕竟不多。好了，不闲扯了，来做题吧。我们就拿NBA这种八个球队的情况来分析吧，八支球队的排名是按常规赛胜率来排的：

1 2 3 4 5 6 7 8

因为是最强和最弱来对决，其次是次强与次弱对决，以此类推可得到：

1-8 2-7 3-6 4-5

那么接下来呢，还是最强与最弱，次强与次弱这种关系：

(1-8 4-5) (2-7 3-6)

最后胜者争夺冠军

((1-8 4-5) (2-7 3-6))

这样分析是不是就清楚了呢，由于n限定了是2的次方数，那么就是可以一直对半分的，比如开始有n队，第一拆分为n/2对匹配，然后再对半拆，就是n/2/2，直到拆到n为1停止，而且每次都是首与末配对，次首与次末配对，这样搞清楚了规律，代码应该就不难写了吧，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     string findContestMatch(int n) {
4         vector<string> v;
5         for (int i = 1; i <= n; ++i) v.push_back(to_string(i));
6         while (n > 1) {
7             for (int i = 0; i < n / 2; ++i) {
8                 v[i] = "(" + v[i] + "," + v[n - i - 1] + ")";
9             }
10            n /= 2;
11        }
12        return v[0];
13    }
14 }
```

CPP

下面这种方法是递归的写法，解题思路跟上面没有区别，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     string findContestMatch(int n) {
4         vector<string> v;
5         for (int i = 1; i <= n; ++i) v.push_back(to_string(i));
6         helper(n, v);
7         return v[0];
8     }
9     void helper(int n, vector<string>& v) {
10        if (n == 1) return;
11        for (int i = 0; i < n; ++i) {
12            v[i] = "(" + v[i] + "," + v[n - i - 1] + ")";
13        }
14        helper(n / 2, v);
15    }
16 };

```

537. 二叉树的边界

Given a binary tree, return the values of its boundary in anti-clockwise direction starting from root. Boundary includes left boundary, leaves, and right boundary in order without duplicate nodes.

Left boundary is defined as the path from root to the left-most node. Right boundary is defined as the path from root to the right-most node. If the root doesn't have left subtree or right subtree, then the root itself is left boundary or right boundary. Note this definition only applies to the input binary tree, and not applies to any subtrees.

The left-most node is defined as a leaf node you could reach when you always firstly travel to the left subtree if exists. If not, travel to the right subtree. Repeat until you reach a leaf node.

The right-most node is also defined by the same way with left and right exchanged.

这道题给了我们一棵二叉树，让我们以逆时针的顺序来输出树的边界，按顺序分别为左边界，叶结点和右边界。题目中给的例子也能让我们很清晰的明白哪些算是边界上的结点。那么最直接的方法就是分别按顺序求出左边界结点，叶结点，和右边界结点。那么如何求的，对于树的操作肯定是用递归最简洁啊，所以我们可以写分别三个递归函数来分别求左边界结点，叶结点，和右边界结点。首先我们先要处理根结点的情况，当根结点没有左右子结点时，其也是一个叶结点，那么我们一开始就将其加入结果res中，那么再计算叶结点的时候又会再加入一次，这样不对。所以我们判断如果根结点至少有一个子结点，我们才提前将其加入结果res中。然后再来看求左边界结点的函数，如果当前结点不存在，或者没有子结点，我们直接返回。否则就把当前结点值加入结果res中，然后看如果左子结点存在，就对其调用递归函数，反之如果左子结点不存在，那么对右子结点调用递归函数。而对于求右边界结点的函数就反过来了，如果右子结点存在，就对其调用递归函数，反之如果右子结点不存在，就对左子结点调用递归函数，注意在调用递归函数之后才将结点值加入结果res，因为我们是需要按逆时针的顺序输出。最后就来看求叶结点的函数，没什么可说的，就是看没有子结点存在了就加入结果res，然后对左右子结点分别调用递归即可，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     vector<int> boundaryOfBinaryTree(TreeNode* root) {
4         if (!root) return {};
5         vector<int> res;
6         if (root->left || root->right) res.push_back(root->val);
7         leftBoundary(root->left, res);
8         leaves(root, res);
9         rightBoundary(root->right, res);
10        return res;
11    }
12    void leftBoundary(TreeNode* node, vector<int>& res) {
13        if (!node || (!node->left && !node->right)) return;
14        res.push_back(node->val);
15        if (!node->left) leftBoundary(node->right, res);
16        else leftBoundary(node->left, res);
17    }
18    void rightBoundary(TreeNode* node, vector<int>& res) {
19        if (!node || (!node->left && !node->right)) return;
20        if (!node->right) rightBoundary(node->left, res);
21        else rightBoundary(node->right, res);
22        res.push_back(node->val);
23    }
24    void leaves(TreeNode* node, vector<int>& res) {
25        if (!node) return;
26        if (!node->left && !node->right) {
27            res.push_back(node->val);
28        }
29        leaves(node->left, res);
30        leaves(node->right, res);
31    }
32 };

```

下面这种方法把上面三种不同的递归揉合到了一个递归中，并用bool型变量来标记当前是求左边界结点还是求右边界结点，同时还有加入叶结点到结果res中的功能。如果左边界标记为true，那么将结点值加入结果res中，下面就是调用对左右结点调用递归函数了。根据上面的解题思路我们知道，如果是求左边界结点，优先调用左子结点，当左子结点不存在时再调右子结点，而对于求右边界结点，优先调用右子结点，当右子结点不存在时再调用左子结点。综上考虑，在对左子结点调用递归函数时，左边界标识设为leftbd && node->left，而对右子结点调用递归的左边界标识设为leftbd && !node->left，这样左子结点存在就会被优先调用。而右边界结点的情况就正好相反，调用左子结点的右边界标识为rightbd && !node->right，调用右子结点的右边界标识为rightbd && node->right，这样就保证了右子结点存在就会被优先调用，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<int> boundaryOfBinaryTree(TreeNode* root) {
4         if (!root) return {};
5         vector<int> res{root->val};
6         helper(root->left, true, false, res);
7         helper(root->right, false, true, res);
8         return res;
9     }
10    void helper(TreeNode* node, bool leftbd, bool rightbd, vector<int>& res) {
11        if (!node) return;
12        if (!node->left && !node->right) {
13            res.push_back(node->val);
14            return;
15        }
16        if (leftbd) res.push_back(node->val);
17        helper(node->left, leftbd && node->left, rightbd && !node->right, res);
18        helper(node->right, leftbd && !node->left, rightbd && node->right, res);
19        if (rightbd) res.push_back(node->val);
20    }
21 };

```

下面这种解法实际上时解法一的迭代形式，整体思路基本一样，只是没有再用递归的写法，而是均采用while的迭代写法，注意在求右边界结点时迭代写法很难直接写出逆时针的顺序，我们可以先反过来保存，最后再调个顺序即可，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     vector<int> boundaryOfBinaryTree(TreeNode* root) {
4         if (!root) return {};
5         vector<int> res, right;
6         TreeNode *l = root->left, *r = root->right, *p = root;
7         if (root->left || root->right) res.push_back(root->val);
8         while (l && (l->left || l->right)) {
9             res.push_back(l->val);
10            if (l->left) l = l->left;
11            else l = l->right;
12        }
13        stack<TreeNode*> st;
14        while (p || !st.empty()) {
15            if (p) {
16                st.push(p);
17                if (!p->left && !p->right) res.push_back(p->val);
18                p = p->left;
19            } else {
20                p = st.top(); st.pop();
21                p = p->right;
22            }
23        }
24        while (r && (r->left || r->right)) {
25            right.push_back(r->val);
26            if (r->right) r = r->right;
27            else r = r->left;
28        }
29        res.insert(res.end(), right.rbegin(), right.rend());
30        return res;
31    }
32 };

```

538. 移除盒子

Given several boxes with different colors represented by different positive numbers. You may experience several rounds to remove boxes until there is no box left. Each time you can choose some continuous boxes with the same color (composed of k boxes, $k \geq 1$), remove them and get k^k points.
Find the maximum points you can get.

刚开始看这道题的时候，感觉跟之前那道Zuma Game很像，于是就写了一个暴力破解的方法，结果TLE了。无奈之下只好上网搜大神们的解法，又看了fun4LeetCode大神写的帖子，之前那道Reverse Pairs就是参考的fun4LeetCode大神的帖子，惊为天人，这次又是这般精彩，大神请收下我的膝盖。那么下面的解法就大部分参考fun4LeetCode大神的帖子来讲解吧。在之前帖子Reverse Pairs的讲解中，大神归纳了两种重现模式，我们这里也试着看能不能套用上。对于这种看来看去都没思路的题来说，抽象建模的能力就非常的重要了。对于题目中的具体场景啊，具体代表的东西我们都可忽略不看，这样能帮助我们接近问题的本质，这道题的本质就是一个数组，我们每次消去一个或多个数字，并获得相应的分数，让我们求最高能获得的分数。而之前那道Zuma Game也是给了一个数组，让我们往某个位置加数字，使得相同的数字至少有3个才能消除，二者是不是很像呢，但是其实解法却差别很大。那道题之所以暴力破解没有问题是因为数组的长度和给定的数字个数都有限制，而且都是相对较小的数，那么即便遍历所有情况也不会有太大的计算量。而这道题就不一样了，既然不能暴力破解，那么对于这种玩数组和子数组的题，刷题老司机们都会优先考虑用DP来做吧。既然要玩子数组，肯定要限定子数组的范围，那么至少应该是个二维的dp数组，其中 $dp[i][j]$ 表示在子数组 $[i, j]$ 范围内所能得到的最高的分数，那么最后我们返回 $dp[0][n-1]$ 就是要求的结果。

那么对于 $dp[i][j]$ 我们想，如果我们移除 $boxes[i]$ 这个数字，那么总得分应该是 $1 + dp[i+1][j]$ ，但是通过分析题目中的例子，能够获得高积分的trick是，移除某个或某几个数字后，如果能使得原本不连续的相同数字变的连续是更好的，因为同时移除的数字越多，那么所得的积分就越高。那么假如在 $[i, j]$ 中间有个位置 m ，使得 $boxes[i]$ 和 $boxes[m]$ 相等，那么我们就应该只是移除 $boxes[i]$ 这个数字，而是还应该考虑直接移除 $[i+1, m-1]$ 区间上的数，使得 $boxes[i]$ 和 $boxes[m]$ 直接相邻，那么我们获得的积分就是 $dp[i+1][m-1]$ ，那么我们剩余了什么， $boxes[i]$ 和 $boxes[m, j]$ 区间的数，此时我们无法处理子数组 $[m, j]$ ，因为我们有些信息没有包括在我们的 dp 数组中，此类的题目归纳为不自己包含的子问题，其解法依赖于一些子问题以外的信息。这类问题通常没有定义好的重现关系，所以不太容易递归求解。为了解决这类问题，我们需要修改问题的定义，使得其包含一些外部信息，从而变成自包含子问题。

那么对于这道题来说，无法处理 $boxes[m, j]$ 区间是因为其缺少了关键信息，我们不知道 $boxes[m]$ 左边相同数字的个数 k ，只有知道了这个信息，那么 m 的位置才有意义，所以我们的 dp 数组应该是一个三维数组 $dp[i][j][k]$ ，表示区间 $[i, j]$ 中能获得的最大积分，当 $boxes[i]$ 左边有 k 个数字跟其相等，那么我们的目标就是要求 $dp[0][n-1][0]$ 了，而且我们也能推出 $dp[i][i][k] = (1+k) * (1+k)$ 这个等式。那么我们来推导重现关系，对于 $dp[i][j][k]$ ，如果我们移除 $boxes[i]$ ，那么我们得到 $(1+k)*(1+k) + dp[i+1][j][0]$ 。对于上面提到的那种情况，当某个位置 m ，有 $boxes[i] == boxes[m]$ 时，我们也应该考虑先移除 $[i+1, m-1]$ 这部分，我们得到积分 $dp[i+1][m-1][0]$ ，然后再处理剩下的部分，得到积分 $dp[m][j][k+1]$ ，这里 k 加1点原因是，移除了中间的部分后，原本和 $boxes[m]$ 不相邻的 $boxes[i]$ 现在相邻了，又因为二者值相同，所以 k 应该加1，因为 k 的定义就是左边相等的数字的个数。讲到这里，那么DP方法最难的递推公式也就得到了，那么代码就不难写了，需要注意的是，这里的C++的写法不能用vector来表示三维数组，好像是内存限制超出，只能用C语言的写法，由于C语言数组的定义需要初始化大小，而题目中说了数组长度不会超100，所以我们就用100来初始化，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     int removeBoxes(vector<int>& boxes) {
4         int n = boxes.size();
5         int dp[100][100][100] = {0};
6         return helper(boxes, 0, n - 1, 0, dp);
7     }
8     int helper(vector<int>& boxes, int i, int j, int k, int dp[100][100][100]) {
9         if (j < i) return 0;
10        if (dp[i][j][k] > 0) return dp[i][j][k];
11        int res = (1 + k) * (1 + k) + helper(boxes, i + 1, j, 0, dp);
12        for (int m = i + 1; m <= j; ++m) {
13            if (boxes[m] == boxes[i]) {
14                res = max(res, helper(boxes, i + 1, m - 1, 0, dp) + helper(boxes, m, j, k +
15                1, dp));
16            }
17        }
18        return dp[i][j][k] = res;
19    }
};
```

CPP

下面这种写法是上面解法的迭代方式，但是却有一些不同，这里我们需要对 dp 数组的部分值做一些初始化，将每个数字的所有 k 值的情况的积分都先算出来，然后在整体更新三维 dp 数组的时候也很有意思，并不是按照原有的顺序更新，而是块更新，先更新 $dp[1][0][k], dp[2][1][k], dp[3][2][k]....$ ，再更新 $dp[2][0][k], dp[3][1][k], dp[4][2][k]....$ ，再更新 $dp[3][0][k], dp[4][1][k], dp[5][2][k]....$ ，之前好像也有一道是这样区域更新的题，但是博主想不起来是哪一道了，以后想起来了再来补充吧，参见代码如下：

解法2:

```

1 class Solution {
2 public:
3     int removeBoxes(vector<int>& boxes) {
4         int n = boxes.size();
5         int dp[n][n][n] = {0};
6         for (int i = 0; i < n; ++i) {
7             for (int k = 0; k <= i; ++k) {
8                 dp[i][i][k] = (1 + k) * (1 + k);
9             }
10        }
11        for (int t = 1; t < n; ++t) {
12            for (int j = t; j < n; ++j) {
13                int i = j - t;
14                for (int k = 0; k <= i; ++k) {
15                    int res = (1 + k) * (1 + k) + dp[i + 1][j][0];
16                    for (int m = i + 1; m <= j; ++m) {
17                        if (boxes[m] == boxes[i]) {
18                            res = max(res, dp[i + 1][m - 1][0] + dp[m][j][k + 1]);
19                        }
20                    }
21                    dp[i][j][k] = res;
22                }
23            }
24        }
25        return n == 0 ? 0 : dp[0][n - 1][0];
26    }
27 };

```

539. 朋友圈

There are N students in a class. Some of them are friends, while some are not. Their friendship is transitive in nature. For example, if A is a direct friend of B, and B is a direct friend of C, then A is an indirect friend of C. And we defined a friend circle is a group of students who are direct or indirect friends.

Given a N*N matrix M representing the friend relationship between students in the class. If $M[i][j] = 1$, then the ithand jth students are direct friends with each other, otherwise not. And you have to output the total number of friend circles among all the students.

这道题让我们求朋友圈的个数，题目中对于朋友圈的定义是可以传递的，比如A和B是好友，B和C是好友，那么即使A和C不是好友，那么他们三人也属于一个朋友圈。那么比较直接的解法就是DFS搜索，对于某个人，遍历其好友，然后再遍历其好友的好友，那么我们就能把属于同一个朋友圈的人都遍历一遍，我们同时标记出已经遍历过的人，然后累积朋友圈的个数，再去对于没有遍历到的人找其朋友圈的人，这样就能求出个数。其实这道题的本质是之前那道题Number of Connected Components in an Undirected Graph，其实许多题目的本质都是一样的，就是看我们有没有一双慧眼能将它们识别出来：

解法1：

```

1 class Solution {
2 public:
3     int findCircleNum(vector<vector<int>>& M) {
4         int n = M.size(), res = 0;
5         vector<bool> visited(n, false);
6         for (int i = 0; i < n; ++i) {
7             if (visited[i]) continue;
8             helper(M, i, visited);
9             ++res;
10        }
11        return res;
12    }
13    void helper(vector<vector<int>>& M, int k, vector<bool>& visited) {
14        visited[k] = true;
15        for (int i = 0; i < M.size(); ++i) {
16            if (!M[k][i] || visited[i]) continue;
17            helper(M, i, visited);
18        }
19    }
20 };

```

我们也可以用BFS来遍历朋友圈中的所有人，解题思路和上面大同小异，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int findCircleNum(vector<vector<int>>& M) {
4         int n = M.size(), res = 0;
5         vector<bool> visited(n, false);
6         queue<int> q;
7         for (int i = 0; i < n; ++i) {
8             if (visited[i]) continue;
9             q.push(i);
10            while (!q.empty()) {
11                int t = q.front(); q.pop();
12                visited[t] = true;
13                for (int j = 0; j < n; ++j) {
14                    if (!M[t][j] || visited[j]) continue;
15                    q.push(j);
16                }
17            }
18            ++res;
19        }
20        return res;
21    }
22 };

```

下面这种解法叫联合查找Union Find，也是一种很经典的解题思路，在之前的两道题Graph Valid Tree和Number of Connected Components in an Undirected Graph中也有过应用，核心思想是初始时给每一个对象都赋上不同的标签，然后对于属于同一类的对象，在root中查找其标签，如果不同，那么将其中一个对象的标签赋值给另一个对象，注意root数组中的数字跟数字的坐标是有很大关系的，root存的是属于同一组的另一个对象的坐标，这样通过getRoot函数可以使同一个组的对象返回相同的值，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int findCircleNum(vector<vector<int>>& M) {
4         int n = M.size(), res = n;
5         vector<int> root(n);
6         for (int i = 0; i < n; ++i) root[i] = i;
7         for (int i = 0; i < n; ++i) {
8             for (int j = i + 1; j < n; ++j) {
9                 if (M[i][j] == 1) {
10                     int p1 = getRoot(root, i);
11                     int p2 = getRoot(root, j);
12                     if (p1 != p2) {
13                         --res;
14                         root[p2] = p1;
15                     }
16                 }
17             }
18         }
19         return res;
20     }
21     int getRoot(vector<int>& root, int i) {
22         while (i != root[i]) {
23             root[i] = root[root[i]];
24             i = root[i];
25         }
26         return i;
27     }
28 };

```

540. 分割数组成和相同的子数组

Given an array with n integers, you need to find if there are triplets (i, j, k) which satisfies following conditions:

$0 < i, i + 1 < j, j + 1 < k < n - 1$

Sum of subarrays $(0, i - 1), (i + 1, j - 1), (j + 1, k - 1)$ and $(k + 1, n - 1)$ should be equal.
where we define that subarray (L, R) represents a slice of the original array starting from the element indexed L to the element indexed R .

这道题给了我们一个数组，让我们找出三个位置，使得数组被分为四段，使得每段之和相等，问存不存在这样的三个位置，注意三个位置上的数字不属于任何一段。刚开始博主觉得这题貌似跟之前那道Partition Equal Subset Sum很像，所以在想能不能用DP来做，可是想了半天不知道DP该如何定义，更别说推导递推公式了。于是就尝试了建立累加和数组，并搜索所有的可能组合，进行暴力破解，结果却TLE了。说明OJ不接受时间复杂度为三次方的解法，那么就要想办法来优化了，博主只好上网学习大神们的解法，发现大神们的解法果然巧妙，只是改变了一个查找顺序，就轻易的将时间复杂度降到了平方级，碉堡了有木有。思路是这样的，因为我们需要找三个位置 i, j, k ，如果我们按正常的顺序来暴力搜索，那么就会遍历所有的情况，其实大部分的情况都是不符合题意的，会有大量的无用的运算。而如果我们换一个角度，先搜索 j 的位置，那么 i 和 k 的位置就可以固定在一个小的范围内了，而且可以在 j 的循环里面同时进行，这样就少嵌套了一个循环，所以时间复杂度会降一维度。确定 j 的范围应该左右各留3个数字，因为四段均不能为空，而且分割位上的数字不能算入四段。再确定了 j 的位置后， i 和 k 的位置就能分别确定了，我们要做的是先遍历 i 的所有可能位置，然后遍历所有的拆分情况，如果拆出的两段和相等，则把这个相等的值加入一个集合中，然后再遍历 k 的所有情况，同样遍历所有的拆分情况，如果拆出两段和相等，再看这个相等的和是否在集合中，如果存在，说明拆出的四段都可以相同，那么返回true即可，否则当遍历结束了，返回false。唉，为啥自己就想不到呢，估计这就是和大神之间的区别吧，泪目中。

解法1：

```

1 class Solution {
2 public:
3     bool splitArray(vector<int>& nums) {
4         if (nums.size() < 7) return false;
5         int n = nums.size();
6         vector<int> sums = nums;
7         for (int i = 1; i < n; ++i) {
8             sums[i] = sums[i - 1] + nums[i];
9         }
10        for (int j = 3; j < n - 3; ++j) {
11            unordered_set<int> s;
12            for (int i = 1; i < j - 1; ++i) {
13                if (sums[i - 1] == (sums[j - 1] - sums[i])) {
14                    s.insert(sums[i - 1]);
15                }
16            }
17            for (int k = j + 1; k < n - 1; ++k) {
18                int s3 = sums[k - 1] - sums[j], s4 = sums[n - 1] - sums[k];
19                if (s3 == s4 && s.count(s3)) return true;
20            }
21        }
22        return false;
23    }
24};

```

下面这种解法是递归的暴力破解写法，刚开始博主还纳闷了，为啥博主之前写的迭代形式的暴力破解过不了OJ，而这个递归版本的确能通过呢，仔细研究了一下，发现这种解法有两个地方做了优化。第一个优化是在for循环里面，如果i不等于1，且当前数字和之前数字均为0，那么跳过这个位置，因为加上0也不会对target有任何影响，那为什么要加上i不等于1的判断呢，因为输入数组如果是七个0，那么实际上应该返回true的，而如果没有i!=1这个条件限制，后面的代码均不会得到执行，那么就直接返回false了，是不对的。第二个优化的地方是在递归函数里面，只有当curSum等于target了，才进一步调用递归函数，这样就相当于做了剪枝处理，减少了大量的不必要的运算，这可能就是其可以通过OJ的原因吧，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool splitArray(vector<int>& nums) {
4         if (nums.size() < 7) return false;
5         int n = nums.size(), target = 0;
6         int sum = accumulate(nums.begin(), nums.end(), 0);
7         for (int i = 1; i < n - 5; ++i) {
8             if (i != 1 && nums[i] == 0 && nums[i - 1] == 0) continue;
9             target += nums[i - 1];
10            if (helper(nums, target, sum - target - nums[i], i + 1, 1)) {
11                return true;
12            }
13        }
14        return false;
15    }
16    bool helper(vector<int>& nums, int target, int sum, int start, int cnt) {
17        if (cnt == 3) return sum == target;
18        int curSum = 0, n = nums.size();
19        for (int i = start + 1; i < n + 2 * cnt - 5; ++i) {
20            curSum += nums[i - 1];
21            if (curSum == target && helper(nums, target, sum - curSum - nums[i], i + 1, cnt
22 + 1)) {
23                return true;
24            }
25        }
26        return false;
27    }
28};

```

基于上面递归的优化方法的启发，博主将两个优化方法加到了之前写的迭代的暴力破解法上，就能通过OJ了，perfect!

解法3：

```

1 class Solution {
2 public:
3     bool splitArray(vector<int>& nums) {
4         int n = nums.size();
5         vector<int> sums = nums;
6         for (int i = 1; i < n; ++i) {
7             sums[i] = sums[i - 1] + nums[i];
8         }
9         for (int i = 1; i <= n - 5; ++i) {
10            if (i != 1 && nums[i] == 0 && nums[i - 1] == 0) continue;
11            for (int j = i + 2; j <= n - 3; ++j) {
12                if (sums[i - 1] != (sums[j - 1] - sums[i])) continue;
13                for (int k = j + 2; k <= n - 1; ++k) {
14                    int sum3 = sums[k - 1] - sums[j];
15                    int sum4 = sums[n - 1] - sums[k];
16                    if (sum3 == sum4 && sum3 == sums[i - 1]) {
17                        return true;
18                    }
19                }
20            }
21        }
22        return false;
23    }
24};

```

541. 二叉树最长连续序列之二

Given a binary tree, you need to find the length of Longest Consecutive Path in Binary Tree.

Especially, this path can be either increasing or decreasing. For example, [1,2,3,4] and [4,3,2,1] are both considered valid, but the path [1,2,4,3] is not valid. On the other hand, the path can be in the child-Parent-child order, where not necessarily be parent-child order.

这道题是之前那道Binary Tree Longest Consecutive Sequence的拓展，那道题只让从父结点到子结点这种顺序来找最长连续序列，而这道题没有这个顺序限制，我们可以任意的拐弯，这样能找到最长的递增或者递减的路径。这道题利用回溯的思想比较容易，因为当一个结点没有子结点时，它只需要跟其父结点进行比较，这种情况最容易处理，而且一旦叶结点处理完了，我们可以一层一层的回溯，直到回到根结点，然后再遍历的过程中不断的更新结果res即可。由于题目中说了要么是递增，要么是递减，我们不能一会递增一会递减，所以我们递增递减的情况都要统计，只是最后取最长的路径。所以我们要知道每一个结点的最长递增和递减路径的长度，当然是从叶结点算起，这样才方便往根结点回溯。当某个结点比其父结点值大1的话，说明这条路径是递增的，那么当我们知道其左右子结点各自的递增路径长度，那么当前结点的递增路径长度就是左右子结点递增路径长度中的较大值加上1，同理如果是递减路径，那么当前结点的递减路径长度就是左右子结点递减路径长度中的较大值加上1，通过这种方式我们可以更新每个结点的递增递减路径长度。在回溯的过程中，一旦我们知道了某个结点的左右子结点的最长递增递减路径长度，那么我们可以算出当前结点的最长连续序列的长度，要么是左子结点的递增路径跟右子结点的递减路径之和加1，要么是左子结点的递减路径跟右子结点的递增路径之和加1，二者中取较大值即可，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     int longestConsecutive(TreeNode* root) {
4         int res = 0;
5         helper(root, root, res);
6         return res;
7     }
8     pair<int, int> helper(TreeNode* node, TreeNode* parent, int& res) {
9         if (!node) return {0, 0};
10        auto left = helper(node->left, node, res);
11        auto right = helper(node->right, node, res);
12        res = max(res, left.first + right.second + 1);
13        res = max(res, left.second + right.first + 1);
14        int inc = 0, dec = 0;
15        if (node->val == parent->val + 1) {
16            inc = max(left.first, right.first) + 1;
17        } else if (node->val - 1 == parent->val) {
18            dec = max(left.second, right.second) + 1;
19        }
20        return {inc, dec};
21    }
22 };

```

CPP

上面的方法把所有内容都写到了一个递归函数中，看起来有些臃肿。而下面这种方法分了两个递归来写，相对来说简洁一些。因为每个结点的最长连续序列长度等于其最长递增路径长度跟最长递减路径之和加1，然后分别对其左右子结点调用递归函数，取三者最大值，相当于对二叉树进行了先序遍历，参见代码如下：

解法2:

```

1 class Solution {
2 public:
3     int longestConsecutive(TreeNode* root) {
4         if (!root) return 0;
5         int res = helper(root, 1) + helper(root, -1) + 1;
6         return max(res, max(longestConsecutive(root->left), longestConsecutive(root-
7 >right)));
8     }
9     int helper(TreeNode* node, int diff) {
10        if (!node) return 0;
11        int left = 0, right = 0;
12        if (node->left && node->val - node->left->val == diff) {
13            left = 1 + helper(node->left, diff);
14        }
15        if (node->right && node->val - node->right->val == diff) {
16            right = 1 + helper(node->right, diff);
17        }
18        return max(left, right);
19    }
20};

```

542. 学生出勤记录之一

You are given a string representing an attendance record for a student. The record only contains the following three characters:

'A' : Absent.
'L' : Late.
'P' : Present.

A student could be rewarded if his attendance record doesn't contain more than one 'A' (absent) or more than two continuous 'L' (late).

You need to return whether the student could be rewarded according to his attendance record.

这道题让我们判断学生的出勤率是否是优秀，判断标准是不能缺勤两次和不能连续迟到三次，那么最直接的方法就是分别记录缺勤和连续迟到的次数，如果当前遇到缺勤，那么缺勤计数器自增1，如果此时次数大于1了，说明已经不是优秀了，直接返回false，否则连续迟到计数器清零。如果当前遇到迟到，那么连续迟到计数器自增1，如果此时连续迟到计数器大于1了，说明已经不是优秀了，直接返回false。如果遇到正常出勤了，那么连续迟到计数器清零，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool checkRecord(string s) {
4         int cntA = 0, cntL = 0;
5         for (char c : s) {
6             if (c == 'A') {
7                 if (++cntA > 1) return false;
8                 cntL = 0;
9             } else if (c == 'L') {
10                 if (++cntL > 2) return false;
11             } else {
12                 cntL = 0;
13             }
14         }
15         return true;
16     }
17 };

```

那么这种方法利用到了string的查找函数，由于本题的逻辑并不复杂，所以我们可以直接对字符串进行操作，利用STL提供的find函数，方法是同时满足下面两个条件就是优秀，第一个条件是找不到A，或者正着找A和逆着找A在同一个位置(说明只有一个A)；第二个条件是找不到LLL，说明不能连续迟到三次，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool checkRecord(string s) {
4         return (s.find("A") == string::npos || s.find("A") == s.rfind("A")) && s.find("LLL")
5 == string::npos;
6     }
7 };

```

再来看使用正则匹配来做的解法，我们找出不合题意的情况，然后取反即可，正则匹配式是A.*A|LLL，其中.*表示有零个或者多个，那么A.*A就是至少有两A的情况，LLL是三个连续的迟到，|表示两个是或的关系，只要能匹配出任意一种情况，就会返回false，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     bool checkRecord(string s) {
4         return !regex_search(s, regex("A.*A|LLL"));
5     }
6 };

```

543. 最优分隔

Given a list of positive integers, the adjacent integers will perform the float division. For example, [2,3,4] -> 2 / 3 / 4.

However, you can add any number of parenthesis at any position to change the priority of operations. You should find out how to add parenthesis to get the maximum result, and return the corresponding expression in string format. Your expression should NOT contain redundant parenthesis.

这道题给了我们一个数组，让我们确定除法的顺序，从而得到值最大的运算顺序，并且不能加多余的括号。刚开始博主没看清题，以为是要返回最大的值，就直接写了个递归的暴力搜索的方法，结果发现是要返回带括号的字符串，尝试的修改了一下，觉得挺麻烦。于是直接放弃抵抗，上网参考大神们的解法，结果大吃一惊，这题原来还可以这么解，完全是数学上的知识啊，太tricky了。数组中n个数字，如果不加括号就是：

$x_1 / x_2 / x_3 / \dots / x_n$

那么我们如何加括号使得其值最大呢，那么就是将 x_2 后面的除数都变成乘数，比如只有三个数字的情况 $a / b / c$ ，如果我们在后两个数上加上括号 $a / (b / c)$ ，实际上就是 $a / b * c$ 。而且b永远只能当除数，a也永远只能当被除数。同理， x_1 只能当被除数， x_2 只能当除数，但是 x_3 之后的数，只要我们都将其变为乘数，那么得到的值肯定是最大的，所以就只有一种加括号的方式，即：

$x_1 / (x_2 / x_3 / \dots / x_n)$

这样的话就完全不用递归了，这道题就变成了一个简单的字符串操作的题目了，这思路，博主服了，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     string optimalDivision(vector<int>& nums) {
4         if (nums.empty()) return "";
5         string res = to_string(nums[0]);
6         if (nums.size() == 1) return res;
7         if (nums.size() == 2) return res + "/" + to_string(nums[1]);
8         res += "/(" + to_string(nums[1]);
9         for (int i = 2; i < nums.size(); ++i) {
10             res += "/" + to_string(nums[i]);
11         }
12         return res + ")";
13     }
14 }
```

CPP

下面这种解法的思路和上面基本相同，就是写法上略有不同，直接看代码吧：

解法2：

```
1 class Solution {
2 public:
3     string optimalDivision(vector<int>& nums) {
4         string res = "";
5         int n = nums.size();
6         for (int i = 0; i < n; ++i) {
7             if (i > 0) res += "/";
8             if (i == 1 && n > 2) res += "(";
9             res += to_string(nums[i]);
10            if (i == n - 1 && n > 2) res += ")";
11        }
12        return res;
13    }
14 }
```

CPP

544. 砖头墙壁

There is a brick wall in front of you. The wall is rectangular and has several rows of bricks. The bricks have the same height but different width. You want to draw a vertical line from the top to the bottom and cross the leastbricks.

The brick wall is represented by a list of rows. Each row is a list of integers representing the width of each brick in this row from left to right.

If your line go through the edge of a brick, then the brick is not considered as crossed. You need to find out how to draw the line to cross the least bricks and return the number of crossed bricks.

You cannot draw a line just along one of the two vertical edges of the wall, in which case the line will obviously cross no bricks.

question 554

这道题给了我们一个砖头墙壁，上面由不同的长度的砖头组成，让我们选个地方从上往下把墙劈开，使得被劈开的砖头个数最少，前提是不能从墙壁的两边劈，这样没有什么意义。我们使用一个哈希表来建立每一个断点的长度和其出现频率之间的映射，这样只要我们从断点频率出现最多的地方劈墙，损坏的板砖一定最少，参见代码如下：

```
1 class Solution {
2 public:
3     int leastBricks(vector<vector<int>>& wall) {
4         int mx = 0;
5         unordered_map<int, int> m;
6         for (auto a : wall) {
7             int sum = 0;
8             for (int i = 0; i < a.size() - 1; ++i) {
9                 sum += a[i];
10                ++m[sum];
11                mx = max(mx, m[sum]);
12            }
13        }
14        return wall.size() - mx;
15    }
16};
```

CPP

545. 分割串联字符串

Given a list of strings, you could concatenate these strings together into a loop, where for each string you could choose to reverse it or not. Among all the possible loops, you need to find the lexicographically biggest string after cutting the loop, which will make the looped string into a regular one.

Specifically, to find the lexicographically biggest string, you need to experience two phases:

Concatenate all the strings into a loop, where you can reverse some strings or not and connect them in the same order as given.

Cut and make one breakpoint in any place of the loop, which will make the looped string into a regular one starting from the character at the cutpoint.

And your job is to find the lexicographically biggest one among all the possible regular strings.

这道题给了我们一些字符串，让我们将其连接起来，连接的时候对于每个字符串我们可以选择翻转或者不翻转，在行程的大字符串上找一个位置cut掉，将该位置当作首字符，前面的字符串移动到末尾去，问怎么cut能使字符串的字母顺序大。刚开始博主想，既然要让最终字符串字母顺序最大，那么每一个字符串当然要尽可能的大了，所以如果其翻转字符串的字母顺序大的话，就要对字符串进行翻转。然后在组成的字符串中找最大的字符进行cut，然而这种思路不一定能得到正确的结果。比如字符串数组["lc", "love", "ydc"]，如果按照博主之前的思路得到的字符串应该为"ydclclove"，但正确结果应该是"ylclovecd"。我们可出来正确的答案中cut位置所在的字符串ydc，虽然cdy小于ydc，但还是翻转了。但是其他的字符都是按照字母顺序来确定要不要翻转的，那么我们可以得出这样的结论，只有cut所在的字符串的翻转可能不按规律。那么我们如何确定cut位置呢，其实没有太好的办法，只能遍历每一个字母。我们首先来根据字母顺序确定要不要翻转每一个字符串，将字母顺序大的连成一个字符串，然后遍历每一个字符串，在每一个字符串中遍历每一个位置，将当前遍历的字符串后面的所有字符串跟前面所有字符串先连起来，存入mid中，然后取出当前遍历的字符串中当前遍历的位置及其后面的字符取出，连上mid，然后再连上当前位置前面的字符，然后跟结果res比较，取较大者存入结果res。这里我们可以进行小优化，如果cut位置的字母大于等于结果res的首字母，我们才进行对比更新。注意我们在遍历每个字符串时，要将其翻转字符串的每一位也遍历了，这样才能涵盖所有的情况，参见代码如下：

```

1 class Solution {
2 public:
3     string splitLoopedString(vector<string>& strs) {
4         if (strs.empty()) return "";
5         string s = "", res = "a";
6         int n = strs.size(), cur = 0;
7         for (string str : strs) {
8             string t = string(str.rbegin(), str.rend());
9             s += str > t ? str : t;
10        }
11        for (int i = 0; i < n; ++i) {
12            string t1 = strs[i], t2 = string(t1.rbegin(), t1.rend());
13            string mid = s.substr(cur + t1.size()) + s.substr(0, cur);
14            for (int j = 0; j < strs[i].size(); ++j) {
15                if (t1[j] >= res[0]) res = max(res, t1.substr(j) + mid + t1.substr(0, j));
16                if (t2[j] >= res[0]) res = max(res, t2.substr(j) + mid + t2.substr(0, j));
17            }
18            cur += strs[i].size();
19        }
20        return res;
21    }
22 };

```

CPP

546. 下一个较大的元素之三

Given a positive 32-bit integer n, you need to find the smallest 32-bit integer which has exactly the same digits existing in the integer n and is greater in value than n. If no such positive 32-bit integer exists, you need to return -1.

这道题给了我们一个数字，让我们对各个位数重新排序，求出刚好比给定数字大的一种排序，如果不存在就返回-1。这道题给的例子的数字都比较简单，我们来看一个复杂的，比如12443322，这个数字的重排序结果应该为13222344，如果我们仔细观察的话会发现数字变大的原因是左数第二位的2变成了3，细心的童鞋会更进一步的发现后面的数字由降序变为了升序，这也不难理解，因为我们要求刚好比给定数字大的排序方式。那么我们再观察下原数字，看看2是怎么确定的，我们发现，如果从后往前看的话，2是第一个小于其右边位数的数字，因为如果是个纯降序排列的数字，做任何改变都不会使数字变大，直接返回-1。知道了找出转折点的方法，再来看如何确定2和谁交换，这里2并没有跟4换位，而是跟3换了，那么如何确定的3？其实也是从后往前遍历，找到第一个大于2的数字交换，然后把转折点之后的数字按升序排列就是最终的结果了。最后记得为防止越界要转为长整数型，然后根据结果判断是否要返回-1即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int nextGreaterElement(int n) {
4         string str = to_string(n);
5         int len = str.size(), i = len - 1;
6         for (; i > 0; --i) {
7             if (str[i] > str[i - 1]) break;
8         }
9         if (i == 0) return -1;
10        for (int j = len - 1; j >= i; --j) {
11            if (str[j] > str[i - 1]) {
12                swap(str[j], str[i - 1]);
13                break;
14            }
15        }
16        sort(str.begin() + i, str.end());
17        long long res = stoll(str);
18        return res > INT_MAX ? -1 : res;
19    }
20 };

```

下面这种解法博主感觉有些耍赖了，用到了STL的内置函数next_permutation，该函数实现的就是这样一个功能，找下一个全排序，刚好比当前的值大，贴上来权当好玩：

解法2：

```

1 class Solution {
2 public:
3     int nextGreaterElement(int n) {
4         string str = to_string(n);
5         next_permutation(str.begin(), str.end());
6         long long res = stoll(str);
7         return (res > INT_MAX || res <= n) ? -1 : res;
8     }
9 };

```

547. 翻转字符串中的单词之三

Given a string, you need to reverse the order of characters in each word within a sentence while still preserving whitespace and initial word order.

这道题让我们翻转字符串中的每个单词，感觉整体难度要比之前两道Reverse Words in a String II和Reverse Words in a String要小一些，由于题目中说明了没有多余空格，使得难度进一步的降低了。首先我们来看使用字符流处理类stringstream来做的方法，相当简单，就是按顺序读入每个单词进行翻转即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     string reverseWords(string s) {
4         string res = "", t = "";
5         istringstream is(s);
6         while (is >> t) {
7             reverse(t.begin(), t.end());
8             res += t + " ";
9         }
10        res.pop_back();
11        return res;
12    }
13 };

```

下面我们来看不使用字符流处理类，也不使用STL内置的reverse函数的方法，那么就是用两个指针，分别指向每个单词的开头和结尾位置，确定了单词的首尾位置后，再用两个指针对单词进行首尾交换即可，有点像验证回文字符串的方法，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     string reverseWords(string s) {
4         int start = 0, end = 0, n = s.size();
5         while (start < n && end < n) {
6             while (end < n && s[end] != ' ') ++end;
7             for (int i = start, j = end - 1; i < j; ++i, --j) {
8                 swap(s[i], s[j]);
9             }
10            start = ++end;
11        }
12        return s;
13    }
14 };

```

548. 四叉树交集

四叉树是一种树数据，其中每个结点恰好有四个子结点：topLeft、topRight、bottomLeft 和 bottomRight。四叉树通常被用来划分一个二维空间，递归地将其细分为四个象限或区域。

我们希望在四叉树中存储 True/False 信息。四叉树用来表示 $N * N$ 的布尔网格。对于每个结点，它将被等分成四个孩子结点直到这个区域内的值都是相同的。每个节点都有另外两个布尔属性：isLeaf 和 val。当这个节点是一个叶子结点时 isLeaf 为真。val 变量储存叶子结点所代表的区域的值。

```

1 class Solution {
2 public:
3     Node* intersect(Node* quadTree1, Node* quadTree2) {
4         if(quadTree1->isLeaf && quadTree1->val) return quadTree1;
5         if(quadTree2->isLeaf && quadTree2->val) return quadTree2;
6         if(quadTree1->isLeaf && !quadTree1->val) return quadTree2;
7         if(quadTree2->isLeaf && !quadTree2->val) return quadTree1;
8
9         auto tl = intersect(quadTree1->topLeft, quadTree2->topLeft);
10        auto tr = intersect(quadTree1->topRight, quadTree2->topRight);
11        auto bl = intersect(quadTree1->bottomLeft, quadTree2->bottomLeft);
12        auto br = intersect(quadTree1->bottomRight, quadTree2->bottomRight);
13
14        if(tl->val == tr->val && tl->val == bl->val && tl->val == br->val && tl->isLeaf &&
15 tr->isLeaf && bl->isLeaf && br->isLeaf)
16            return new Node(tl->val, true, nullptr, nullptr, nullptr, nullptr);
17        else
18            return new Node(false, false, tl, tr, bl, br);
19    }
20};

```

549. N叉树的最大深度

给定一个 N 叉树，找到其最大深度。

最大深度是指从根节点到最远叶子节点的最长路径上的节点总数。

```

1 class Solution {
2 public:
3     int maxDepth(Node* root) {
4         if (!root) {
5             return 0;
6         }
7
8         int max_depth = 0;
9         for (auto child : root->children) {
10             max_depth = std::max(max_depth, maxDepth(child));
11         }
12
13         return 1 + max_depth;
14     }
15 };

```

550. K 子数组和为K

Given an array of integers and an integer k, you need to find the total number of continuous subarrays whose sum equals to k.

这道题给了我们一个数组，让我们求和为k的连续子数组的个数，博主最开始看到这道题想着肯定要建立累加和数组啊，然后遍历累加和数组的每个数字，首先看其是否为k，是的话结果res自增1，然后再加个往前的循环，这样可以快速求出所有的子数组之和，看是否为k，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int subarraySum(vector<int>& nums, int k) {
4         int res = 0, n = nums.size();
5         vector<int> sums = nums;
6         for (int i = 1; i < n; ++i) {
7             sums[i] = sums[i - 1] + nums[i];
8         }
9         for (int i = 0; i < n; ++i) {
10            if (sums[i] == k) ++res;
11            for (int j = i + 1; j < n; ++j) {
12                if (sums[j] - sums[i] == k) ++res;
13            }
14        }
15        return res;
16    }
17 };

```

上面的求累加和的方法其实并没有提高程序的执行效率，跟下面这种暴力搜索的解法并没有什么不同，博主很惊奇OJ居然这么大度，让这种解法也能通过，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int subarraySum(vector<int>& nums, int k) {
4         int res = 0, n = nums.size();
5         for (int i = 0; i < n; ++i) {
6             int sum = nums[i];
7             if (sum == k) ++res;
8             for (int j = i + 1; j < n; ++j) {
9                 sum += nums[j];
10                if (sum == k) ++res;
11            }
12        }
13        return res;
14    }
15 };

```

论坛上大家比较推崇的其实是这种解法，用一个哈希表来建立连续子数组之和跟其出现次数之间的映射，初始化要加入{0,1}这对映射，这是为啥呢，因为我们的解题思路是遍历数组中的数字，用sum来记录到当前位置的累加和，我们建立哈希表的目的是为了让我们可以快速的查找sum-k是否存在，即是否有连续子数组的和为sum-k，如果存在的话，那么和为k的子数组一定也存在，这样当sum刚好为k的时候，那么数组从起始到当前位置的这段子数组的和就是k，满足题意，如果哈希表中事先没有m[0]项的话，这个符合题意的结果就无法累加到结果res中，这就是初始化的用途。上面讲解的内容顺带着也把for循环中的内容解释了，这里就不多阐述了，有疑问的童鞋请在评论区留言哈，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int subarraySum(vector<int>& nums, int k) {
4         int res = 0, sum = 0, n = nums.size();
5         unordered_map<int, int> m{{0, 1}};
6         for (int i = 0; i < n; ++i) {
7             sum += nums[i];
8             res += m[sum - k];
9             ++m[sum];
10        }
11    return res;
12 }
13 };

```

551. 数组分割之一

Given an array of $2n$ integers, your task is to group these integers into n pairs of integer, say $(a_1, b_1), (a_2, b_2), \dots, (a_n, b_n)$ which makes sum of $\min(a_i, b_i)$ for all i from 1 to n as large as possible.

这道题让我们分割数组，两两一对，让每对中较小的数的和最大。这题难度不大，用贪婪算法就可以了。由于我们要最大化每对中的较小值之和，那么肯定是每对中两个数字大小越接近越好，因为如果差距过大，而我们只取较小的数字，那么大数字就浪费掉了。明白了这一点，我们只需要给数组排个序，然后按顺序的每两个就是一对，我们取出每对中的第一个数即为较小值累加起来即可，参见代码如下：

```

1 class Solution {
2 public:
3     int arrayPairSum(vector<int>& nums) {
4         int res = 0, n = nums.size();
5         sort(nums.begin(), nums.end());
6         for (int i = 0; i < n; i += 2) {
7             res += nums[i];
8         }
9         return res;
10    }
11 };

```

552. 矩阵中最长的连续1

Given a 01 matrix M, find the longest line of consecutive one in the matrix. The line could be horizontal, vertical, diagonal or anti-diagonal.

这道题给了我们一个二维矩阵，让我们求矩阵中最长的连续1，连续方向任意，可以是水平，竖直，对角线或者逆对角线均可。那么最直接最暴力的方法就是四个方向分别来统计最长的连续1，其中水平方向和竖直方向都比较容易，就是逐行逐列的扫描，使用一个计数器，如果当前位置是1，则计数器自增1，并且更新结果res，否则计数器清零。对于对角线和逆对角线需要进行些坐标转换，对于一个 $m \times n$ 的矩阵，对角线和逆对角线的排数都是 $m+n-1$ 个，难点在于我们要确定每一排上的数字的坐标，如果i是从0到 $m+n-1$ 之间遍历，j是在i到0之间遍历，那么对角线的数字的坐标就为 $(i-j, j)$ ，逆对角线的坐标就为 $(m-1-i+j, j)$ ，这是博主千辛万苦试出来的TT，如果能直接记住，效果肯定棒！那么有了坐标转换，求对角线和逆对角线的连续1也就不是啥难事了，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int longestLine(vector<vector<int>>& M) {
4         if (M.empty() || M[0].empty()) return 0;
5         int res = 0, m = M.size(), n = M[0].size();
6         for (int i = 0; i < m; ++i) { // Check horizontal
7             int cnt = 0;
8             for (int j = 0; j < n; ++j) {
9                 if (M[i][j] == 1) res = max(res, ++cnt);
10                else cnt = 0;
11            }
12        }
13        for (int j = 0; j < n; ++j) {
14            int cnt = 0;
15            for (int i = 0; i < m; ++i) { // Check vertical
16                if (M[i][j] == 1) res = max(res, ++cnt);
17                else cnt = 0;
18            }
19        }
20        for (int i = 0; i < m + n - 1; ++i) {
21            int cnt1 = 0, cnt2 = 0;
22            for (int j = i; j >= 0; --j) {
23                if (i - j < m && j < n) { // Check diagonal
24                    if (M[i - j][j] == 1) res = max(res, ++cnt1);
25                    else cnt1 = 0;
26                }
27                int t = m - 1 - i + j;
28                if (t >= 0 && t < m && j < n) { // Check anti-diagonal
29                    if (M[t][j] == 1) res = max(res, ++cnt2);
30                    else cnt2 = 0;
31                }
32            }
33        }
34        return res;
35    }
36 };

```

如果上面的解法的坐标转换不好想的话，我们也可以考虑用DP解法来做，我们建立一个三维dp数组，其中 $dp[i][j][k]$ 表示从开头遍历到数字 $nums[i][j]$ 为止，第 k 种情况的连续1的个数， k 的值为0, 1, 2, 3，分别对应水平，竖直，对角线和逆对角线这四种情况。之后就是更新dp数组的过程了，如果如果数字为0的情况直接跳过，然后水平方向就加上前一个的dp值，竖直方向加上上面一个数字的dp值，对角线方向就加上右上方数字的dp值，逆对角线就加上左上方数字的dp值，然后每个值都用来更新结果res，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int longestLine(vector<vector<int>>& M) {
4         if (M.empty() || M[0].empty()) return 0;
5         int m = M.size(), n = M[0].size(), res = 0;
6         vector<vector<vector<int>>> dp(m, vector<vector<int>>(n, vector<int>(4)));
7         for (int i = 0; i < m; ++i) {
8             for (int j = 0; j < n; ++j) {
9                 if (M[i][j] == 0) continue;
10                for (int k = 0; k < 4; ++k) dp[i][j][k] = 1;
11                if (j > 0) dp[i][j][0] += dp[i][j - 1][0]; // horizontal
12                if (i > 0) dp[i][j][1] += dp[i - 1][j][1]; // vertical
13                if (i > 0 && j < n - 1) dp[i][j][2] += dp[i - 1][j + 1][2]; // diagonal
14                if (i > 0 && j > 0) dp[i][j][3] += dp[i - 1][j - 1][3]; // anti-diagonal
15                res = max(res, max(dp[i][j][0], dp[i][j][1]));
16                res = max(res, max(dp[i][j][2], dp[i][j][3]));
17            }
18        }
19        return res;
20    }
21 };

```

下面我们来优化空间复杂度，用一种类似于DFS的思路来解决问题，我们在遍历到为1的点时，对其水平方向，竖直方向，对角线方向和逆对角线方向分别不停遍历，直到越界或者遇到为0的数字，同时用计数器来累计1的个数，这样就可以用来更新结果res了，就不用把每个中间结果都保存下来了，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int longestLine(vector<vector<int>>& M) {
4         if (M.empty() || M[0].empty()) return 0;
5         int m = M.size(), n = M[0].size(), res = 0;
6         vector<vector<int>> dirs{{1,0},{0,1},{-1,-1},{-1,1}};
7         for (int i = 0; i < m; ++i) {
8             for (int j = 0; j < n; ++j) {
9                 if (M[i][j] == 0) continue;
10                for (int k = 0; k < 4; ++k) {
11                    int cnt = 0, x = i, y = j;
12                    while (x >= 0 && x < m && y >= 0 && y < n && M[x][y] == 1) {
13                        x += dirs[k][0];
14                        y += dirs[k][1];
15                        ++cnt;
16                    }
17                    res = max(res, cnt);
18                }
19            }
20        }
21        return res;
22    }
23 };

```

553. 二叉树的坡度

Given a binary tree, return the tilt of the whole tree.

The tilt of a tree node is defined as the absolute difference between the sum of all left subtree node values and the sum of all right subtree node values. Null node has tilt 0.

The tilt of the whole tree is defined as the sum of all nodes' tilt.

这道题让我们求二叉树的坡度，某个结点的坡度的定义为该结点的左子树之和与右子树之和的差的绝对值，这道题让我们求所有结点的坡度之和。我开始的想法就是老老实实的按定义去做，用先序遍历，对于每个遍历到的结点，先计算坡度，根据定义就是左子树之和与右子树之和的差的绝对值，然后返回的是当前结点的tilt加上对其左右子结点调用求坡度的递归函数即可。其中求子树之和用另外一个函数来求，也是用先序遍历来求结点之和，为了避免重复运算，这里用哈希表来保存已经算过的结点，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     unordered_map<TreeNode*, int> m;
4     int findTilt(TreeNode* root) {
5         if (!root) return 0;
6         int tilt = abs(getSum(root->left, m) - getSum(root->right, m));
7         return tilt + findTilt(root->left) + findTilt(root->right);
8     }
9     int getSum(TreeNode* node, unordered_map<TreeNode*, int>& m) {
10        if (!node) return 0;
11        if (m.count(node)) return m[node];
12        m[node] = getSum(node->left, m) + getSum(node->right, m) + node->val;
13    }
14};
```

CPP

但是在论坛中看了大神们的帖子后，发现这道题最好的解法应该是用后序遍历来做，因为后序遍历的顺序是左-右-根，那么就会从叶结点开始处理，这样我们就能很方便的计算结点的累加和，同时也可以很容易的根据子树和来计算tilt，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     int findTilt(TreeNode* root) {
4         int res = 0;
5         postorder(root, res);
6         return res;
7     }
8     int postorder(TreeNode* node, int& res) {
9         if (!node) return 0;
10        int leftSum = postorder(node->left, res);
11        int rightSum = postorder(node->right, res);
12        res += abs(leftSum - rightSum);
13        return leftSum + rightSum + node->val;
14    }
15};
```

CPP

554. 寻找最近的回文串

Given an integer n, find the closest integer (not including itself), which is a palindrome.

The 'closest' is defined as absolute difference minimized between two integers.

这道题给了我们一个数字，让我们找到其最近的回文数，而且说明了这个最近的回文数不能是其本身。比如如果给你个131，那么就需要返回121。而且返回的回文数可能位数还不同，比如当n为100的时候，我们就应该返回99，或者给了我们99时，需要返回101。那么实际上最近回文数是有范围的，比如说n为三位数，那么其最近回文数的范围在[99, 1001]之间，这样我们就可以根据给定数字的位数来确定出两个边界值，要和其他生成的回文数进行比较，取绝对差最小的。

下面我们来看如何求一般情况下的最近回文数，我们知道回文数就是左半边和右半边互为翻转，奇数情况下中间还有个单独的值。那么如何将一个不是回文数的数变成回文数呢，我们有两种选择，要么改变左半边，要么改变右半边。由于我们希望和原数绝对差最小，肯定是改变低位上的数比较好，所以我们改变右半边，那么改变的情况又分为两种，一种是原数本来就是回文数，这种情况下，我们需要改变中间的那个数字，要么增加1，要么减小1，比如121，可以变成111和131。另一种情况是原数不是回文数，我们只需要改变右半边就行了，比如123，变成121。那么其实这三种情况可以总结起来，分别相当于对中间的2进行了-1,+1,+0操作，那么我们就可以用一个-1到1的for循环一起处理了，先取出包括中间数的左半边，比如123就取出12，1234也取出12，然后就要根据左半边生成右半边，为了同时处理奇数和偶数的情况，我们使用一个小tricky，在反转复制左半边的时候，我们给rbegin()加上len&1，当奇数时，len&1为1，这样就不会复制中间数了；为偶数时，len&1为0，这就整个翻转复制了左半边。我们把每次生成的回文串转为数字后加入到一个集合set中，把之前的两个边界值也同样加进去，最后我们在五个candidates中找出和原数绝对差最小的那个返回，记得别忘了在集合中删除原数，因为如果原数时回文的话，i=0时就把自己也加入了集合了，参见代码如下：

```

1 | class Solution {
2 | public:
3 |     string nearestPalindromic(string n) {
4 |         long len = n.size(), num = stol(n), res, minDiff = LONG_MAX;
5 |         unordered_set<long> s;
6 |         s.insert(pow(10, len) + 1);
7 |         s.insert(pow(10, len - 1) - 1);
8 |         long prefix = stol(n.substr(0, (len + 1) / 2));
9 |         for (long i = -1; i <= 1; ++i) {
10 |             string pre = to_string(prefix + i);
11 |             string str = pre + string(pre.rbegin() + (len & 1), pre.rend());
12 |             s.insert(stol(str));
13 |         }
14 |         s.erase(num);
15 |         for (auto a : s) {
16 |             long diff = abs(a - num);
17 |             if (diff < minDiff) {
18 |                 minDiff = diff;
19 |                 res = a;
20 |             } else if (diff == minDiff) {
21 |                 res = min(res, a);
22 |             }
23 |         }
24 |         return to_string(res);
25 |     }
26 | };

```

CPP

555. 数组嵌套

A zero-indexed array A consisting of N different integers is given. The array contains all integers in the range [0, N - 1].

Sets S[K] for 0 <= K < N are defined as follows:

S[K] = { A[K], A[A[K]], A[A[A[K]]], ... }.

Sets S[K] are finite for each K and should NOT contain duplicates.

Write a function that given an array A consisting of N integers, return the size of the largest set S[K] for this array.

这道题让我们找嵌套数组的最大个数，给的数组总共有n个数字，范围均在[0, n-1]之间，题目中也把嵌套数组的生成解释的很清楚了，其实就是值变成坐标，得到的数值再变坐标。那么实际上当循环出现的时候，嵌套数组的长度也不能再增加了，而出现的这个相同的数一定是嵌套数组的首元素，博主刚开始没有想清楚这一点，以为出现重复数字的地方可能是嵌套数组中间的某个位置，于是用个set将生成的嵌套数组存入，然后每次查找新生成的数组是否已经存在。而且还要以原数组中每个数字当作嵌套数组的起始数字都算一遍，结果当然是TLE了。其实对于遍历过的数字，我们不用再将其当作开头来计算了，而是只对于未遍历过的数字当作嵌套数组的开头数字，不过在进行嵌套运算的时候，并不考虑中间的数字是否已经访问过，而是只要找到和起始位置相同的数字位置，然后更新结果res，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int arrayNesting(vector<int>& nums) {
4         int n = nums.size(), res = INT_MIN;
5         vector<bool> visited(n, false);
6         for (int i = 0; i < nums.size(); ++i) {
7             if (visited[nums[i]]) continue;
8             res = max(res, helper(nums, i, visited));
9         }
10        return res;
11    }
12    int helper(vector<int>& nums, int start, vector<bool>& visited) {
13        int i = start, cnt = 0;
14        while (cnt == 0 || i != start) {
15            visited[i] = true;
16            i = nums[i];
17            ++cnt;
18        }
19        return cnt;
20    }
21 };

```

CPP

下面这种方法写法上更简洁一些，思路完全一样，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int arrayNesting(vector<int>& nums) {
4         int n = nums.size(), res = INT_MIN;
5         vector<bool> visited(n, false);
6         for (int i = 0; i < n; ++i) {
7             if (visited[nums[i]]) continue;
8             int cnt = 0, j = i;
9             while(cnt == 0 || j != i) {
10                 visited[j] = true;
11                 j = nums[j];
12                 ++cnt;
13             }
14             res = max(res, cnt);
15         }
16         return res;
17     }
18 };

```

下面这种解法是网友@edyyy提醒博主的，我们可以优化解法二的空间，我们并不需要专门的数组来记录数组是否被遍历过，而是在遍历的过程中，将其交换到其应该出现的位置上，因为如果某个数出现在正确的位置上，那么它一定无法组成嵌套数组，这样就相当于我们标记了其已经访问过了，思路确实很赞啊，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int arrayNesting(vector<int>& nums) {
4         int n = nums.size(), res = 0;
5         for (int i = 0; i < n; ++i) {
6             int cnt = 1;
7             while (nums[i] != i && nums[i] != nums[nums[i]]) {
8                 swap(nums[i], nums[nums[i]]);
9                 ++cnt;
10            }
11            res = max(res, cnt);
12        }
13        return res;
14    }
15 };

```

556. 重塑矩阵

In MATLAB, there is a very useful function called 'reshape', which can reshape a matrix into a new one with different size but keep its original data.

You're given a matrix represented by a two-dimensional array, and two positive integers r and c representing the row number and column number of the wanted reshaped matrix, respectively.

The reshaped matrix need to be filled with all the elements of the original matrix in the same row-traversing order as they were.

If the 'reshape' operation with given parameters is possible and legal, output the new reshaped matrix; Otherwise, output the original matrix.

这道题让我们实现矩阵大小的重塑，也就是实现Matlab中的reshape函数，博主也经常使用matlab，对这个函数还是比较的熟悉的。对于这种二维数组大小重新分配的问题的关键就是对应位置的坐标转换，最直接的办法就是先把原数组拉直，变成一条直线，然后再组成新的数组。所以这道题我们先判断给定数组是否能重塑成给定的大小，就是看两者的元素总数是否相同，直接行数乘以列数即可，然后我们新建一个目标大小的数组，并开始遍历，对于每个位置，我们先转为拉直后的一维坐标，然后在算出在原数组中的对应位置赋值过来即可，参见代码如下：

解法1：

```
1 class Solution {
2     public:
3         vector<vector<int>> matrixReshape(vector<vector<int>>& nums, int r, int c) {
4             int m = nums.size(), n = nums[0].size();
5             if (m * n != r * c) return nums;
6             vector<vector<int>> res(r, vector<int>(c));
7             for (int i = 0; i < r; ++i) {
8                 for (int j = 0; j < c; ++j) {
9                     int k = i * c + j;
10                    res[i][j] = nums[k / n][k % n];
11                }
12            }
13            return res;
14        }
15    };

```

CPP

下面这种方法整体思路和上面没啥区别，但是只使用了一个循环，直接就是遍历拉直后的一维数组的坐标，然后分别转换为两个二维数组的坐标进行赋值，参见代码如下：

解法2：

```
1 class Solution {
2     public:
3         vector<vector<int>> matrixReshape(vector<vector<int>>& nums, int r, int c) {
4             int m = nums.size(), n = nums[0].size();
5             if (m * n != r * c) return nums;
6             vector<vector<int>> res(r, vector<int>(c));
7             for (int i = 0; i < r * c; ++i) {
8                 res[i / c][i % c] = nums[i / n][i % n];
9             }
10            return res;
11        }
12    };

```

CPP

557. 字符串中的全排列

Given two strings s1 and s2, write a function to return true if s2 contains the permutation of s1. In other words, one of the first string's permutations is the substring of the second string.

这道题给了两个字符串s1和s2，问我们s1的全排列的字符串任意一个是否为s2的字串。虽然题目中有全排列的关键字，但是跟之前的全排列的题目的解法并不一样，如果受思维定势影响比较深的话，很容易遍历s1所有全排列的情况，然后检测其是否为s2的子串，这种解法是非常不高效的，估计OJ不会答应。这道题的正确做法应该是使用滑动窗口Sliding Window的思想来做，可以使用两个哈希表来做，或者是使用一个哈希表配上双指针来做。我们先来看使用两个哈希表来做的情况，我们先来分别统计s1和s2中前n1个字符串中各个字符出现的次数，其中n1为字符串s1的长度，这样如果二者字符出现次数的情况完全相同，说明s1和s2中前n1的字符互为全排列关系，那么符合题意了，直接返回true。如果不是的话，那么我们遍历s2之后的字符，对于遍历到的字符，对应的次数加1，由于窗口的大小限定为了n1，所以每在窗口右侧加一个新字符的同时就要在窗口左侧去掉一个字符，每次都比较一下两个哈希表的情况，如果相等，说明存在，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     bool checkInclusion(string s1, string s2) {
4         int n1 = s1.size(), n2 = s2.size();
5         vector<int> m1(128), m2(128);
6         for (int i = 0; i < n1; ++i) {
7             ++m1[s1[i]];
8         }
9         if (m1 == m2) return true;
10        for (int i = n1; i < n2; ++i) {
11            ++m2[s2[i]];
12            --m2[s2[i - n1]];
13            if (m1 == m2) return true;
14        }
15        return false;
16    }
17 };
18

```

CPP

下面这种解法是利用一个哈希表加上双指针，我们还是先统计s1中字符的出现次数，然后遍历s2中的字符，对于每个遍历到的字符，我们在哈希表中对应的字符次数减1，如果次数次数小于0了，说明该字符在s1中不曾出现，或是出现的次数超过了s1中的对应的字符出现次数，那么我们此时移动滑动窗口的左边界，对于移除的字符串，哈希表中对应的次数要加1，如果此时次数不为0，说明该字符不在s1中，继续向右移，直到更新后的次数为0停止，此时到达的字符是在s1中的。如果次数大于等于0了，我们看此时窗口大小是否为s1的长度，若二者相等，由于此时窗口中的字符都是在s1中存在的字符，而且对应的次数都为0了，说明窗口中的字符串和s1互为全排列，返回true即可，参见代码如下：

解法2:

```

1 class Solution {
2 public:
3     bool checkInclusion(string s1, string s2) {
4         int n1 = s1.size(), n2 = s2.size(), left = 0;
5         vector<int> m(128);
6         for (char c : s1) ++m[c];
7         for (int right = 0; right < n2; ++right) {
8             if (--m[s2[right]] < 0) {
9                 while (++m[s2[left++]] != 0) {}
10            } else if (right - left + 1 == n1) return true;
11        }
12        return n1 == 0;
13    }
14 };

```

CPP

下面这种解法也是用一个哈希表外加双指针来做的，跟上面的解法思路大体相同，写法有些不同，不变的还是统计s1中字符出现的次数，不一样的是我们用一个变量cnt来表示还需要匹配的s1中的字符的个数，初始化为s1的长度，然后遍历s2中的字符，如果该字符在哈希表中存在，说明匹配上了，cnt自减1，哈希表中的次数也应该自减1，然后如果cnt减为0了，说明s1的字符都匹配上了，如果此时窗口的大小正好为s1的长度，那么说明找到了s1的全排列，返回true，否则说明窗口过大，里面有一些非s1中的字符，我们将左边界右移，同时将移除的字符串在哈希表中的次数自增1，如果增加后的次数大于0了，说明该字符是s1中的字符，我们将其移除了，那么cnt就要自增1，参见代码如下：

解法3:

```

1 class Solution {
2 public:
3     bool checkInclusion(string s1, string s2) {
4         int n1 = s1.size(), n2 = s2.size(), cnt = n1, left = 0;
5         vector<int> m(128);
6         for (char c : s1) ++m[c];
7         for (int right = 0; right < n2; ++right) {
8             if (m[s2[right]]-- > 0) --cnt;
9             while (cnt == 0) {
10                 if (right - left + 1 == n1) return true;
11                 if (++m[s2[left++]] > 0) ++cnt;
12             }
13         }
14         return false;
15     }
16 };

```

558. 最大化休假日

LeetCode wants to give one of its best employees the option to travel among N cities to collect algorithm problems. But all work and no play makes Jack a dull boy, you could take vacations in some particular cities and weeks. Your job is to schedule the traveling to maximize the number of vacation days you could take, but there are certain rules and restrictions you need to follow.

Rules and restrictions:

You can only travel among N cities, represented by indexes from 0 to N-1. Initially, you are in the city indexed 0 on Monday.

The cities are connected by flights. The flights are represented as a N*N matrix (not necessary symmetrical), called flights representing the airline status from the city i to the city j. If there is no flight from the city i to the city j, flights[i][j] = 0; Otherwise, flights[i][j] = 1. Also, flights[i][i] = 0 for all i.

You totally have K weeks (each week has 7 days) to travel. You can only take flights at most once per day and can only take flights on each week's Monday morning. Since flight time is so short, we don't consider the impact of flight time.

For each city, you can only have restricted vacation days in different weeks, given an N*K matrix called days representing this relationship. For the value of days[i][j], it represents the maximum days you could take vacation in the city i in the week j.

You're given the flights matrix and days matrix, and you need to output the maximum vacation days you could take during K weeks.

这道题给了我们一个NxN的数组，表示城市i是否有飞机直达城市j，又给了我们一个NxK的数组days，表示在第j周能在城市i休假的天数，让我们找出一个行程能使我们休假的天数最大化。开始尝试写了个递归的暴力破解法，结果TLE了。其实这道题比较适合用DP来解，我们建立一个二维DP数组，其中dp[i][j]表示目前是第j周，并且在此时在城市i，总共已经获得休假的总日子数。我们采取从后往前更新的方式(不要问我为什么，因为从前往后更新的写法要复杂一些)，我们从第k周开始往第一周遍历，那么最后结果都累加在了dp[i][0]中，i的范围是[0, n-1]，找出其中的最大值就是我们能休息的最大假期数了。难点就在于找递推式了，我们想dp[i][j]表示的是当前是第j周并在城市i已经获得的休假总日子数，那么上一个状态，也就是j+1周(因为我们是从后往前更新)，跟当前状态有何联系，上一周我们可能还在城市i，也可能在其他城市p，那么在其他城市p的条件是，城市p有直飞城市i的飞机，那么我们可以用上一个状态的值dp[p][j+1]来更新当前值dp[i][j]，还要注意的是我们要从倒数第二周开始更新，因为倒数第一周没有上一个状态，还有就是每个状态dp[i][j]都初始化赋为days[i][j]来更新，这样一旦没有任何城市可以直飞当前城市，起码我们还可以享受当前城市的假期，最后要做的就是想上面所说在dp[i][0]中找最大值，下面的代码是把这一步融合到了for循环中，所以加上了一堆判断条件，我们也可以在dp数组整个更新结束之后再来找最大值，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int maxVacationDays(vector<vector<int>>& flights, vector<vector<int>>& days) {
4         int n = flights.size(), k = days[0].size(), res = 0;
5         vector<vector<int>> dp(n, vector<int>(k, 0));
6         for (int j = k - 1; j >= 0; --j) {
7             for (int i = 0; i < n; ++i) {
8                 dp[i][j] = days[i][j];
9                 for (int p = 0; p < n; ++p) {
10                     if ((i == p || flights[i][p]) && j < k - 1) {
11                         dp[i][j] = max(dp[i][j], dp[p][j + 1] + days[i][j]);
12                     }
13                     if (j == 0 && (i == 0 || flights[0][i])) res = max(res, dp[i][0]);
14                 }
15             }
16         }
17         return res;
18     }
19 };

```

下面这种方法优化了空间复杂度，只用了一个一维的DP数组，其中dp[i]表示在当前周，在城市i时已经获得的最大假期数，并且除了第一个数初始化为0，其余均初始化为整型最小值，然后我们从第一周往后遍历，我们新建一个临时数组t，初始化为整型最小值，然后遍历每一个城市，对于每一个城市，我们遍历其他所有城市，看是否有飞机能直达当前城市，或者就是当前的城市，我们用dp[p] + days[i][j]来更新dp[i]，当每个城市都遍历完了之后，我们将t整个赋值给dp，然后进行下一周的更新，最后只要在dp数组中找出最大值返回即可，这种写法不但省空间，而且也相对简洁一些，很赞啊~

解法2：

```

1 class Solution {
2 public:
3     int maxVacationDays(vector<vector<int>>& flights, vector<vector<int>>& days) {
4         int n = flights.size(), k = days[0].size();
5         vector<int> dp(n, INT_MIN);
6         dp[0] = 0;
7         for (int j = 0; j < k; ++j) {
8             vector<int> t(n, INT_MIN);
9             for (int i = 0; i < n; ++i) {
10                 for (int p = 0; p < n; ++p) {
11                     if (i == p || flights[p][i]) {
12                         t[i] = max(t[i], dp[p] + days[i][j]);
13                     }
14                 }
15             }
16             dp = t;
17         }
18         return *max_element(dp.begin(), dp.end());
19     }
20 };

```

之前提到了递归的DFS会TLE，但是如果我们使用一个memo数组来保存中间计算结果，就能省去大量的重复计算，并且能够通过OJ，解题思想跟解法一非常的类似，参见代码如下：

解法3：

```
1 class Solution {
2 public:
3     int maxVacationDays(vector<vector<int>>& flights, vector<vector<int>>& days) {
4         int n = flights.size(), k = days[0].size();
5         vector<vector<int>> memo(n, vector<int>(k, 0));
6         return helper(flights, days, 0, 0, memo);
7     }
8     int helper(vector<vector<int>>& flights, vector<vector<int>>& days, int city, int day,
9     vector<vector<int>>& memo) {
10        int n = flights.size(), k = days[0].size(), res = 0;
11        if (day == k) return 0;
12        if (memo[city][day] > 0) return memo[city][day];
13        for (int i = 0; i < n; ++i) {
14            if (i == city || flights[city][i] == 1) {
15                res = max(res, days[i][day] + helper(flights, days, i, day + 1, memo));
16            }
17        }
18        return memo[city][day] = res;
19    }
};
```

559. Median Employee Salary

The Employee table holds all employees. The employee table has three columns: Employee Id, Company Name, and Salary.

Id	Company	Salary
1	A	2341
2	A	341
3	A	15
4	A	15314
5	A	451
6	A	513
7	B	15
8	B	13
9	B	1154
10	B	1345
11	B	1221
12	B	234
13	C	2345
14	C	2645
15	C	2645
16	C	2652
17	C	65

Write a SQL query to find the median salary of each company. Bonus points if you can solve it without using any built-in SQL functions.

Id	Company	Salary
5	A	451
6	A	513
12	B	234
9	B	1154
14	C	2645

```

1 | SELECT
2 | Employee.Id, Employee.Company, Employee.Salary
3 | FROM
4 | Employee,
5 | Employee alias
6 | WHERE
7 | Employee.Company = alias.Company
8 | GROUP BY Employee.Company , Employee.Salary
9 | HAVING SUM(CASE
10 |     WHEN Employee.Salary = alias.Salary THEN 1
11 |     ELSE 0
12 | END) >= ABS(SUM(SIGN(Employee.Salary - alias.Salary)))
13 | ORDER BY Employee.Id
14 |

```

SQL

560. Managers with at Least 5 Direct Reports

The Employee table holds all employees including their managers. Every employee has an Id, and there is also a column for the manager Id.

Id	Name	Department	ManagerId
101	John	A	null
102	Dan	A	101
103	James	A	101
104	Amy	A	101
105	Anne	A	101
106	Ron	B	101

Given the Employee table, write a SQL query that finds out managers with at least 5 direct report. For the above table, your SQL query should return:

+-----+
Name
+-----+
John
+-----+

```

1 | SELECT
2 | Name
3 | FROM
4 | Employee AS t1 JOIN
5 | (SELECT
6 | ManagerId
7 | FROM
8 | Employee
9 | GROUP BY ManagerId
10 | HAVING COUNT(ManagerId) >= 5) AS t2
11 | ON t1.Id = t2.ManagerId
12 |
;
```

SQL

561. Find Median Given Frequency of Numbers

```

1 | select FORMAT(avg(n.Number),4)*1.0 as median
2 | from Numbers n left join
3 | (
4 |   select Number, @prev := @count as prevNumber, (@count := @count + Frequency) as
5 |   countNumber
6 |   from Numbers,
7 |   (select @count := 0, @prev := 0, @total := (select sum(Frequency) from Numbers)) temp
8 |   order by Number
9 |   ) n2
10 | on n.Number = n2.Number
11 | where
12 |   (prevNumber < floor((@total+1)/2) and countNumber >= floor((@total+1)/2))
or
13 |   (prevNumber < floor((@total+2)/2) and countNumber >= floor((@total+2)/2))
;
```

SQL

562. 另一个树的子树

Given two non-empty binary trees s and t , check whether tree t has exactly the same structure and node values with a subtree of s . A subtree of s is a tree consists of a node in s and all of this node's descendants. The tree scould also be considered as a subtree of itself.

这道题让我们求一个数是否是另一个树的子树，从题目中的第二个例子中可以看出，子树必须是从叶结点开始的，中间某个部分的不能算是子树，那么我们转换一下思路，是不是从 s 的某个结点开始，跟 t 的所有结构都一样，那么问题就转换成了判断两棵树是否相同，也就是Same Tree的问题了，这点想通了其实代码就很好写了，用递归来写十分的简洁，我们先从 s 的根结点开始，跟 t 比较，如果两棵树完全相同，那么返回true，否则就分别对 s 的左子结点和右子结点调用递归再次来判断是否相同，只要有一个返回true了，就表示可以找得到。

解法1：

```
1 class Solution {
2 public:
3     bool isSubtree(TreeNode* s, TreeNode* t) {
4         if (!s) return false;
5         if (isSame(s, t)) return true;
6         return isSubtree(s->left, t) || isSubtree(s->right, t);
7     }
8     bool isSame(TreeNode* s, TreeNode* t) {
9         if (!s && !t) return true;
10        if (!s || !t) return false;
11        if (s->val != t->val) return false;
12        return isSame(s->left, t->left) && isSame(s->right, t->right);
13    }
14};
```

CPP

下面这道题的解法用到了之前那道Serialize and Deserialize Binary Tree的解法，思路是对 s 和 t 两棵树分别进行序列化，各生成一个字符串，如果 t 的字符串是 s 的子串的话，就说明 t 是 s 的子树，但是需要注意的是，为了避免出现[12],[2]，这种情况，虽然2也是12的子串，但是[2]却不是[12]的子树，所以我们再序列化的时候要特殊处理一下，就是在每个结点值前面都加上一个字符，比如','，来分隔开，那么[12]序列化后就是",12,"，而[2]序列化之后就是",2,"，这样就可以完美的解决之前的问题了，参见代码如下：

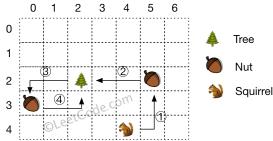
解法2：

```
1 class Solution {
2 public:
3     bool isSubtree(TreeNode* s, TreeNode* t) {
4         ostringstream os1, os2;
5         serialize(s, os1);
6         serialize(t, os2);
7         return os1.str().find(os2.str()) != string::npos;
8     }
9     void serialize(TreeNode* node, ostringstream& os) {
10        if (!node) os << ",#";
11        else {
12            os << "," << node->val;
13            serialize(node->left, os);
14            serialize(node->right, os);
15        }
16    }
17};
```

CPP

563. 松鼠模拟

There's a tree, a squirrel, and several nuts. Positions are represented by the cells in a 2D grid. Your goal is to find the minimal distance for the squirrel to collect all the nuts and put them under the tree one by one. The squirrel can only take at most one nut at one time and can move in four directions - up, down, left and right, to the adjacent cell. The distance is represented by the number of moves.



这道题是关于可爱的小松鼠的题目，不由得让人想起来冰河世纪里面的那只对栗子执着追求的原始松鼠。每天在校园里也能见到抱着栗子啃的小家伙，有的挺个大白肚皮，吃的巨肥，完全没有天敌啊。本题说有一只小松鼠，一堆在不同位置的栗子，还有一棵树，小松鼠目标是把所有的栗子都运到树的位置，问怎样的顺序可以使用最少的步数。那么我们这么想，如果小松鼠本身就在树的位置，那么把所有的栗子运回树的步数就是一个定值，为每个栗子距树的距离总和乘以2。那么只有当小松鼠不在树的位置时候，它首先要走到一个栗子的位置，然后再去树那儿。而且一旦小松鼠到了树那，再出发，之后的步数就是定值了。所以关键就在于决定小松鼠首先去哪个栗子那。博主最开始犯了一个这道题很容易犯的一个错误，就是在选起始栗子的时候的判定条件是松鼠到该栗子的距离加上该栗子到树的距离之和最小当作判定条件，其实这样是不对的。举个简单的反例，比如此时有两个栗子A和B，小松鼠到栗子A的距离为2，栗子A到树的距离为1，小松鼠到栗子B的距离为2，栗子B到树的距离为2。那么按照博主之前的选择方法，会选先去栗子A，因为小松鼠到栗子A再到树的距离之和为3，小于先去栗子B再去树的距离之和(为4)。然而小松鼠先去栗子A的话，总距离就是7，而如果先去栗子B的话，总距离为6，这就说明之前的判定条件不对。那么正确思路应该是，假设小松鼠最先应该去栗子i，那么我们假设栗子i到树的距离为x，小松鼠到栗子i的距离为y，那么如果小松鼠不去栗子i，累加步数就是 $2x$ ，如果小松鼠去栗子i，累加步数就是 $x+y$ ，我们希望 $x+y$ 尽可能的小于 $2x$ ，那么就是 y 尽可能小于 x ，即 $x-y$ 越大越好。这样我们遍历每个栗子，找出 $x-y$ 最大的那个，让小松鼠先去捡就好了。话说萌萌的小松鼠真是很可爱，希望这些小萌物们远离马路，不要随便过马路，真是太危险了。。。

```

1 class Solution {
2 public:
3     int minDistance(int height, int width, vector<int>& tree, vector<int>& squirrel,
4     vector<vector<int>>& nuts) {
5         int res = 0, mxDiff = INT_MIN, idx = 0;
6         for (auto nut : nuts) {
7             int dist = abs(tree[0] - nut[0]) + abs(tree[1] - nut[1]);
8             res += 2 * dist;
9             mxDiff = max(mxDiff, dist - abs(squirrel[0] - nut[0]) - abs(squirrel[1] -
10 nut[1]));
11         }
12         return res - mxDiff;
13     }
14 };

```

CPP

564. Winning Candidate

Table: Candidate

id	Name
1	A
2	B
3	C
4	D
5	E

Table: Vote

id	CandidateId
1	2
2	4
3	3
4	2
5	5

id is the auto-increment primary key,

CandidateId is the id appeared in Candidate table.

Write a sql to find the name of the winning candidate, the above example will return the winner B.

Name
B

```

1 | SELECT
2 |   name AS 'Name'
3 | FROM
4 |   Candidate
5 |     JOIN
6 |   (SELECT
7 |     Candidateid
8 |   FROM
9 |     Vote
10|   GROUP BY Candidateid
11|   ORDER BY COUNT(*) DESC
12|   LIMIT 1) AS winner
13| WHERE
14| Candidate.id = winner.Candidateid;
```

SQL

565. 分糖果

Given an integer array with even length, where different numbers in this array represent different kinds of candies. Each number means one candy of the corresponding kind. You need to distribute these candies equally in number to brother and sister. Return the maximum number of kinds of candies the sister could gain.

这道题给我们一堆糖，每种糖的个数不定，分给两个人，让我们求其中一个人能拿到的最大的糖的种类数。那么我们想，如果总共有n个糖，平均分给两个人，每人得到 $n/2$ 块糖，那么能拿到的最大的糖的种类数也就是 $n/2$ 种，不可能再多，只可能再少。那么我们要做的就是统计出总共的糖的种类数，如果糖的种类数小于 $n/2$ ，说明拿不到 $n/2$ 种糖，最多能拿到的种类数数就是当前糖的总种类数，明白了这点就很容易了，我们利用集合set的自动去重复特性来求出糖的种类数，然后跟 $n/2$ 比较，取二者之中的较小值返回即可，参加代码如下：

解法1：

```
1 class Solution {
2 public:
3     int distributeCandies(vector<int>& candies) {
4         unordered_set<int> s;
5         for (int candy : candies) s.insert(candy);
6         return min(s.size(), candies.size() / 2);
7     }
8 };
```

CPP

下面这种方法写的不行，直接用把上面的解法浓缩为了一行，有种显摆的感觉：

解法2：

```
1 class Solution {
2 public:
3     int distributeCandies(vector<int>& candies) {
4         return min(unordered_set<int>(candies.begin(), candies.end()).size(), candies.size()
5 / 2);
6     }
7 };
```

CPP

566. 出界的路径

There is an m by n grid with a ball. Given the start coordinate (i, j) of the ball, you can move the ball to adjacent cell or cross the grid boundary in four directions (up, down, left, right). However, you can at most move N times. Find out the number of paths to move the ball out of grid boundary. The answer may be very large, return it after mod $10^9 + 7$.

这道题给了我们一个二维的数组，某个位置放个足球，每次可以在上下左右四个方向中任意移动一步，总共可以移动N步，问我们总共能有多少种移动方法能把足球移除边界，由于结果可能是个巨大的数，所以让我们对一个大数取余。那么我们知道对于这种结果很大的数如果用递归解法很容易爆栈，所以最好考虑使用DP来解。那么我们使用一个三维的DP数组，其中 $dp[k][i][j]$ 表示总共走k步，从 (i,j) 位置走出边界的总路径数。那么我们来找递推式，对于 $dp[k][i][j]$ ，走k步出边界的总路径数等于其周围四个位置的走 $k-1$ 步出边界的总路径数之和，如果周围某个位置已经出边界了，那么就直接加上1，否则就在dp数组中找出该值，这样整个更新下来，我们就能得出每一个位置走任意步数的出界路径数了，最后只要返回 $dp[N][i][j]$ 就是所求结果了，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int findPaths(int m, int n, int N, int i, int j) {
4         vector<vector<vector<int>>> dp(N + 1, vector<vector<int>>(m, vector<int>(n, 0)));
5         for (int k = 1; k <= N; ++k) {
6             for (int x = 0; x < m; ++x) {
7                 for (int y = 0; y < n; ++y) {
8                     long long v1 = (x == 0) ? 1 : dp[k - 1][x - 1][y];
9                     long long v2 = (x == m - 1) ? 1 : dp[k - 1][x + 1][y];
10                    long long v3 = (y == 0) ? 1 : dp[k - 1][x][y - 1];
11                    long long v4 = (y == n - 1) ? 1 : dp[k - 1][x][y + 1];
12                    dp[k][x][y] = (v1 + v2 + v3 + v4) % 1000000007;
13                }
14            }
15        }
16        return dp[N][i][j];
17    }
18};

```

下面这种方法虽然也是用的DP解法，但是DP数组的定义和上面的不一样，这种解法相当于使用了BFS搜索，以(i,j)为起始点，其中 $dp[k][x][y]$ 表示用了k步，进入(x,y)位置的路径数，由于 $dp[k][x][y]$ 只依赖于 $dp[k-1][x][y]$ ，所以我们可以用一个二维dp数组来代替，初始化 $dp[i][j]$ 为1，总共N步，进行N次循环，每次都新建一个 $m \times n$ 大小的临时数组t，然后就是对于遍历到的每个位置，都遍历其四个相邻位置，如果相邻位置越界了，那么我们用当前位置的dp值更新结果res，因为此时dp值的意义就是从(i,j)到越界位置的路径数。如果没有，我们将当前位置的dp值赋给t数组的对应位置，这样在遍历完所有的位置时，将数组t整个赋值给dp，然后进入下一步的循环，参加代码如下：

解法2：

```

1 class Solution {
2 public:
3     int findPaths(int m, int n, int N, int i, int j) {
4         int res = 0;
5         vector<vector<int>> dp(m, vector<int>(n, 0));
6         dp[i][j] = 1;
7         vector<vector<int>> dirs{{0,-1},{-1,0},{0,1},{1,0}};
8         for (int k = 0; k < N; ++k) {
9             vector<vector<int>> t(m, vector<int>(n, 0));
10            for (int r = 0; r < m; ++r) {
11                for (int c = 0; c < n; ++c) {
12                    for (auto dir : dirs) {
13                        int x = r + dir[0], y = c + dir[1];
14                        if (x < 0 || x >= m || y < 0 || y >= n) {
15                            res = (res + dp[r][c]) % 1000000007;
16                        } else {
17                            t[x][y] = (t[x][y] + dp[r][c]) % 1000000007;
18                        }
19                    }
20                }
21            }
22            dp = t;
23        }
24        return res;
25    }
26};

```

567. Employee Bonus

Select all employee's name and bonus whose bonus is < 1000.

Table:Employee

```
+-----+-----+-----+
| empId | name   | supervisor| salary |
+-----+-----+-----+
| 1     | John    | 3          | 1000   |
| 2     | Dan     | 3          | 2000   |
| 3     | Brad    | null       | 4000   |
| 4     | Thomas  | 3          | 4000   |
+-----+-----+-----+
```

empId is the primary key column for this table.

Table: Bonus

```
+-----+
| empId | bonus |
+-----+
| 2      | 500   |
| 4      | 2000  |
+-----+
```

empId is the primary key column for this table.

```
1 | SELECT
2 |   Employee.name, Bonus.bonus
3 | FROM
4 |   Employee
5 |     LEFT OUTER JOIN
6 |     Bonus ON Employee.empid = Bonus.empid
7 | ;
```

SQL

568. Get Highest Answer Rate Question

Get the highest answer rate question from a table survey_log with these columns: uid, action, question_id, answer_id, q_num, timestamp.

uid means user id; action has these kind of values: "show", "answer", "skip"; answer_id is not null when action column is "answer", while is null for "show" and "skip"; q_num is the numeral order of the question in current session.

Write a sql query to identify the question which has the highest answer rate.

```

1  SELECT question_id AS survey_log
2  FROM
3  (
4    SELECT question_id,
5      SUM(case when action="answer" THEN 1 ELSE 0 END) AS num_answer,
6      SUM(case when action="show" THEN 1 ELSE 0 END) AS num_show,
7      FROM survey_log
8      GROUP BY question_id
9    ) AS tbl
10 ORDER BY (num_answer / num_show) DESC
11 LIMIT 1
12
13 SELECT
14   question_id AS 'survey_log'
15   FROM
16   survey_log
17   GROUP BY question_id
18   ORDER BY COUNT(answer_id) / COUNT(IF(action = 'show', 1, 0)) DESC
19   LIMIT 1;

```

569. Find Cumulative Salary of an Employee

The Employee table holds the salary information in a year.

Write a SQL to get the cumulative sum of an employee's salary over a period of 3 months but exclude the most recent month.

The result should be displayed by 'Id' ascending, and then by 'Month' descending.

```

1  SELECT
2    E1.id,
3    E1.month,
4    (IFNULL(E1.salary, 0) + IFNULL(E2.salary, 0) + IFNULL(E3.salary, 0)) AS Salary
5  FROM
6  (SELECT
7    id, MAX(month) AS month
8    FROM
9    Employee
10   GROUP BY id
11   HAVING COUNT(*) > 1) AS maxmonth
12  LEFT JOIN
13  Employee E1 ON (maxmonth.id = E1.id
14                  AND maxmonth.month > E1.month)
15  LEFT JOIN
16  Employee E2 ON (E2.id = E1.id
17                  AND E2.month = E1.month - 1)
18  LEFT JOIN
19  Employee E3 ON (E3.id = E1.id
20                  AND E3.month = E1.month - 2)
21  ORDER BY id ASC, month DESC
22 ;

```

570. Count Student Number in Departments

A university uses 2 data tables, student and department, to store data about its students and the departments associated with each major.

Write a query to print the respective department name and number of students majoring in each department for all departments in the department table (even ones with no current students).

Sort your results by descending number of students; if two or more departments have the same number of students, then sort those departments alphabetically by department name.

The student is described as follow:

Column Name	Type
student_id	Integer
student_name	String
gender	Character
dept_id	Integer

where student_id is the student's ID number, student_name is the student's name, gender is their gender, and dept_id is the department ID associated with their declared major.

And the department table is described as below:

Column Name	Type
dept_id	Integer
dept_name	String

where dept_id is the department's ID number and dept_name is the department name.

```

1 SELECT
2 dept_name, COUNT(student_id) AS student_number
3 FROM
4 department
5 LEFT OUTER JOIN
6 student ON department.dept_id = student.dept_id
7 GROUP BY department.dept_name
8 ORDER BY student_number DESC , department.dept_name
9 ;

```

SQL

571. 最短无序连续子数组

Given an integer array, you need to find one continuous subarray that if you only sort this subarray in ascending order, then the whole array will be sorted in ascending order, too.

You need to find the shortest such subarray and output its length.

这道题给了我们一个数组，让我们求最短的无序连续子数组，根据题目中的例子也不难分析出来是让我们找出数组中的无序的部分。那么我最开始的想法就是要确定无序子数组的起始和结束位置，这样就能知道子数组的长度了。所以我们用一个变量start来记录起始位置，然后我们开始遍历数组，当我们发现某个数字比其前面的数字要小的时候，说明此时数组不再有序，所以我们要将此数字向前移动，移到其应该在的地方，我们用另一个变量j来记录移动到的位置，然后我们考虑要不要用这个位置来更新start的值，当start还是初始值-1时，肯定要更新，因为这是出现的第一个无序的地方，还有就是如果当前位置小于start也要更新，这说明此时的无序数组比之前的更长了。我们举个例子来说明，比如数组[1,3,5,4,2]，第一个无序的地方是数字4，它移动到的正确位置是坐标2，此时start更新为2，然后下一个无序的地方是数字2，它的正确位置是坐标1，所以此时start应更新为1，这样每次用i - start + 1来更新结果res时才能得到正确的结果，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int findUnsortedSubarray(vector<int>& nums) {
4         int res = 0, start = -1, n = nums.size();
5         for (int i = 1; i < n; ++i) {
6             if (nums[i] < nums[i - 1]) {
7                 int j = i;
8                 while (j > 0 && nums[j] < nums[j - 1]) {
9                     swap(nums[j], nums[j - 1]);
10                --j;
11            }
12            if (start == -1 || start > j) start = j;
13            res = max(res, i - start + 1);
14        }
15    }
16    return res;
17 }
18 };

```

下面这种方法是用了一个辅助数组，我们新一个跟原数组一摸一样的数组，然后排序。从数组起始位置开始，两个数组相互比较，当对应位置数字不同的时候停止，同理再从末尾开始，对应位置上比较，也是遇到不同的数字时停止，这样中间一段就是最短无序连续子数组了，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int findUnsortedSubarray(vector<int>& nums) {
4         int n = nums.size(), i = 0, j = n - 1;
5         vector<int> t = nums;
6         sort(t.begin(), t.end());
7         while (i < n && nums[i] == t[i]) ++i;
8         while (j > i && nums[j] == t[j]) --j;
9         return j - i + 1;
10    }
11 };

```

下面这种方法很叼啊，是O(n)的时间复杂度加上O(1)的空间复杂度，博主觉得这实际上是对上面的那种方法进行空间上的优化的结果，用两个变量mx和mn来代替上面的有序数组，我们仔细来分析发现，最小值mn初始化为数组的最后一个数字，最大值mx初始化为了第一个数字，然后我们从第二个数字开始遍历，mx和nums[i]之间取较大值赋值给mx，然后比较此时mx和nums[i]之间的大小关系，如果mx大于nums[i]，就把i赋值给end，那么我们想如果第一个数字小于第二个，mx就会赋值为第二个数字，这时候mx和nums[i]就相等了，不进行任何操作，这make sense，因为说明此时是有序的。mn和nums[n-1-i]之间取较小值赋给mn，然后比较此时mn和nums[n-1-i]之间的大小关系，如果mn小于nums[n-1-i]，就把n-1-i赋值给start，那么什么时候会进行赋值呢，是当倒数第二个数字大于最后一个数字，这样mn还是最后一个数字，而nums[n-1-i]就会大于mn，这样我们更新start。我们可以看出start是不断往前走的，end是不断往后走的，整个遍历完成后，start和end就分别指向了最短无序连续子数组的起始和结束位置，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int findUnsortedSubarray(vector<int>& nums) {
4         int n = nums.size(), start = -1, end = -2;
5         int mn = nums[n - 1], mx = nums[0];
6         for (int i = 1; i < n; ++i) {
7             mx = max(mx, nums[i]);
8             mn = min(mn, nums[n - 1 - i]);
9             if (mx > nums[i]) end = i;
10            if (mn < nums[n - 1 - i]) start = n - 1 - i;
11        }
12        return end - start + 1;
13    }
14 };

```

572. 结束进程

Given n processes, each process has a unique PID (process id) and its PPID (parent process id).

Each process only has one parent process, but may have one or more children processes. This is just like a tree structure. Only one process has PPID that is 0, which means this process has no parent process. All the PIDs will be distinct positive integers.

We use two lists of integers to represent a list of processes, where the first list contains PID for each process and the second list contains the corresponding PPID.

Now given the two lists, and a PID representing a process you want to kill, return a list of PIDs of processes that will be killed in the end. You should assume that when a process is killed, all its children processes will be killed. No order is required for the final answer.

这道题让我们结束进程，一直不想翻译成杀死进程，感觉进程很可怜的样子，还要被杀死。题目给了我们两个数组，一个是进程的数组，还有一个是进程数组中的每个进程的父进程组成的数组。题目中说结束了某一个进程，其所有的子进程都需要结束，由于一个进程可能有多个子进程，所以我们首先要理清父子进程的关系。所以我们使用一个哈希表，建立进程和其所有子进程之间的映射，然后我们首先把要结束的进程放入一个队列queue中，然后while循环，每次取出一个进程，将其加入结果res中，然后遍历其所有子进程，将所有子进程都排入队列中，这样我们就能结束所有相关的进程，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<int> killProcess(vector<int>& pid, vector<int>& ppid, int kill) {
4         vector<int> res;
5         queue<int> q{{kill}};
6         unordered_map<int, vector<int>> m;
7         for (int i = 0; i < pid.size(); ++i) {
8             m[ppid[i]].push_back(pid[i]);
9         }
10        while (!q.empty()) {
11            int t = q.front(); q.pop();
12            res.push_back(t);
13            for (int p : m[t]) {
14                q.push(p);
15            }
16        }
17        return res;
18    }
19 };

```

我们也可以使用递归的写法，思路都一样，只不过用递归函数来代替队列，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<int> killProcess(vector<int>& pid, vector<int>& ppid, int kill) {
4         vector<int> res;
5         unordered_map<int, vector<int>> m;
6         for (int i = 0; i < pid.size(); ++i) {
7             m[ppid[i]].push_back(pid[i]);
8         }
9         helper(kill, m, res);
10        return res;
11    }
12    void helper(int kill, unordered_map<int, vector<int>>& m, vector<int>& res) {
13        res.push_back(kill);
14        for (int p : m[kill]) {
15            helper(p, m, res);
16        }
17    }
18 };

```

573. 两个字符串的删除操作

Given two words word1 and word2, find the minimum number of steps required to make word1 and word2 the same, where in each step you can delete one character in either string.

这道题给了我们两个单词，问我们最少需要多少步可以让两个单词相等，每一步我们可以在任意一个单词中删掉一个字符。那么我们分析怎么能让步数最少呢，是不是知道两个单词最长的相同子序列的长度，并乘以2，被两个单词的长度之和减，就是最少步数了。其实这道题就转换成求Longest Common Subsequence最长相同子序列的问题，令博主意外的是，LeetCode中竟然没有这道题，这与包含万物的LeetCode的作风不符啊。不过没事，有这道题也行啊，对于这种玩字符串，并且是求极值的问题，十有八九都是用dp来解的，曾经有网友问博主，如何确定什么时候用greedy，什么时候用dp？其实博主也不太清楚，感觉dp要更tricky一些，而且出现的概率大，所以博主一般会先考虑dp，如果实在想不出递推公式，那么就想想greedy能做不。如果有大神知道更好的区分方法，请一定留言告知博主啊，多谢！那么决定了用dp来做，就定义一个二维的dp数组，其中 $dp[i][j]$ 表示word1的前*i*个字符和word2的前*j*个字符组成的两个单词的最长公共子序列的长度。下面来看递推式 $dp[i][j]$ 怎么求，首先来考虑

$dp[i][j]$ 和 $dp[i-1][j-1]$ 之间的关系，我们可以发现，如果当前的两个字符相等，那么 $dp[i][j] = dp[i-1][j-1] + 1$ ，这不难理解吧，因为最长相同子序列又多了一个相同的字符，所以长度加1。由于我们dp数组的大小定义的是 $(n1+1) \times (n2+1)$ ，所以我们比较的是 $word1[i-1]$ 和 $word2[j-1]$ 。那么我们想如果这两个字符不相等呢，难道我们直接将 $dp[i-1][j-1]$ 赋值给 $dp[i][j]$ 吗，当然不是，我们还要错位相比嘛，比如就拿题目中的例子来说，“sea”和“eat”，当我们比较第一个字符，发现’s’和’e’不相等，下一步就要错位比较啊，比较sea中第一个’s’和eat中的’a’，sea中的’e’跟eat中的第一个’e’相比，这样我们的 $dp[i][j]$ 就要取 $dp[i-1][j]$ 跟 $dp[i][j-1]$ 中的较大值了，最后我们求出了最大共同子序列的长度，就能直接算出最小步数了，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int minDistance(string word1, string word2) {
4         int n1 = word1.size(), n2 = word2.size();
5         vector<vector<int>> dp(n1 + 1, vector<int>(n2 + 1, 0));
6         for (int i = 1; i <= n1; ++i) {
7             for (int j = 1; j <= n2; ++j) {
8                 if (word1[i - 1] == word2[j - 1]) {
9                     dp[i][j] = dp[i - 1][j - 1] + 1;
10                } else {
11                    dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
12                }
13            }
14        }
15        return n1 + n2 - 2 * dp[n1][n2];
16    }
17};

```

CPP

下面这种方法也是用的dp，但是和上面的dp思路不太一样，这种算法是跟之前那道Edit Distance相同的思路。那道题问我们一个单词通过多少步修改可以得到另一个单词，其实 $word2$ 删除一个字符，和跟在 $word1$ 对应的地方加上那个要删除的字符，达到的效果是一样的，并不影响最终的步骤数，所以这道题完全可以按照那道题的解法来做，一点都不需要变动，定义一个二维的dp数组，其中 $dp[i][j]$ 表示 $word1$ 的前*i*个字符和 $word2$ 的前*j*个字符组成的两个单词，能使其变相同的最小的步数，讲解可以参看那篇帖子，参见代码入下：

解法2：

```

1 class Solution {
2 public:
3     int minDistance(string word1, string word2) {
4         int n1 = word1.size(), n2 = word2.size();
5         vector<vector<int>> dp(n1 + 1, vector<int>(n2 + 1, 0));
6         for (int i = 0; i <= n1; ++i) dp[i][0] = i;
7         for (int j = 0; j <= n2; ++j) dp[0][j] = j;
8         for (int i = 1; i <= n1; ++i) {
9             for (int j = 1; j <= n2; ++j) {
10                 if (word1[i - 1] == word2[j - 1]) {
11                     dp[i][j] = dp[i - 1][j - 1];
12                 } else {
13                     dp[i][j] = 1 + min(dp[i - 1][j], dp[i][j - 1]);
14                 }
15             }
16         }
17         return dp[n1][n2];
18     }
19 };

```

CPP

下面这种方法是解法二的递归写法，用的优化的dfs的方法，用memo数组来保存中间计算结果，以避免大量的重复计算，参见代码如下：

解法3：

```
1 | class Solution {
2 | public:
3 |     int minDistance(string word1, string word2) {
4 |         int n1 = word1.size(), n2 = word2.size();
5 |         vector<vector<int>> memo(n1 + 1, vector<int>(n2 + 1, 0));
6 |         return helper(word1, word2, 0, 0, memo);
7 |     }
8 |     int helper(string word1, string word2, int p1, int p2, vector<vector<int>>& memo) {
9 |         if (memo[p1][p2] != 0) return memo[p1][p2];
10 |         int n1 = word1.size(), n2 = word2.size();
11 |         if (p1 == n1 || p2 == n2) return n1 - p1 + n2 - p2;
12 |         if (word1[p1] == word2[p2]) {
13 |             memo[p1][p2] = helper(word1, word2, p1 + 1, p2 + 1, memo);
14 |         } else {
15 |             memo[p1][p2] = 1 + min(helper(word1, word2, p1 + 1, p2, memo), helper(word1,
16 | word2, p1, p2 + 1, memo));
17 |         }
18 |         return memo[p1][p2];
19 |     }
};
```

CPP

574. Find Customer Referee

Given a table customer holding customers information and the referee.

+-----+-----+-----+
id name referee_id
+-----+-----+-----+
1 Will NULL
2 Jane NULL
3 Alex 2
4 Bill NULL
5 Zack 1
6 Mark 2
+-----+-----+-----+

Write a query to return the list of customers NOT referred by the person with id '2'.

```
1 | SELECT name FROM customer WHERE referee_id <> 2 OR referee_id IS NULL;
```

SQL

575. Investments in 2016

Write a query to print the sum of all total investment values in 2016 (TIV_2016), to a scale of 2 decimal places, for all policy holders who meet the following criteria:

Have the same TIV_2015 value as one or more other policyholders.

Are not located in the same city as any other policyholder (i.e.: the (latitude, longitude) attribute pairs must be unique).

Input Format:

The insurance table is described as follows:

Column Name	Type
PID	INTEGER(11)
TIV_2015	NUMERIC(15,2)
TIV_2016	NUMERIC(15,2)
LAT	NUMERIC(5,2)
LON	NUMERIC(5,2)

where PID is the policyholder's policy ID, TIV_2015 is the total investment value in 2015, TIV_2016 is the total investment value in 2016, LAT is the latitude of the policy holder's city, and LON is the longitude of the policy holder's city.

```

1 | SELECT
2 | SUM(insurance.TIV_2016) AS TIV_2016
3 | FROM
4 | insurance
5 | WHERE
6 | insurance.TIV_2015 IN
7 | (
8 |   SELECT
9 |     TIV_2015
10 |   FROM
11 |   insurance
12 |   GROUP BY TIV_2015
13 |   HAVING COUNT(*) > 1
14 |
15 | AND CONCAT(LAT, LON) IN
16 | (
17 |   SELECT
18 |     CONCAT(LAT, LON)
19 |   FROM
20 |   insurance
21 |   GROUP BY LAT , LON
22 |   HAVING COUNT(*) = 1
23 |
24 | ;

```

576. Customer Placing the Largest Number of Orders

Query the customer_number from the orders table for the customer who has placed the largest number of orders.

It is guaranteed that exactly one customer will have placed more orders than any other customer.

The orders table is defined as follows:

Column	Type
order_number (PK)	int
customer_number	int
order_date	date
required_date	date
shipped_date	date
status	char(15)
comment	char(200)

```

1 | SELECT
2 | customer_number
3 | FROM
4 | orders
5 | GROUP BY customer_number
6 | ORDER BY COUNT(*) DESC
7 | LIMIT 1
8 |

```

SQL

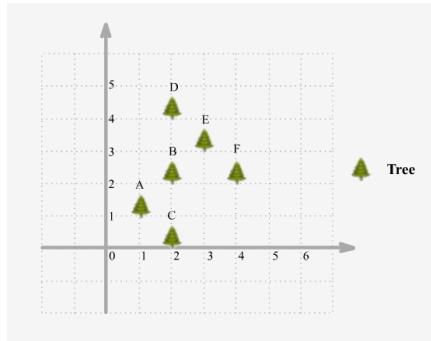
577. 竖立栅栏

There are some trees, where each tree is represented by (x,y) coordinate in a two-dimensional garden. Your job is to fence the entire garden using the minimum length of rope as it is expensive. The garden is well fenced only if all the trees are enclosed. Your task is to help find the coordinates of trees which are exactly located on the fence perimeter.

这道题给了我们一些树，每个树都有其特定的坐标，让我们用最少的栅栏将其全部包住，让我们找出在栅栏边上的树。其实这道题是凸包问题，就是平面上给了一堆点，让我们找出一个多边形，正好包括了所有的点。凸包问题的算法有很多，常见的有八种，参见wiki上的这个帖子。我们来看一种比较常见的算法，卷包裹Gift wrapping算法，又叫Jarvis march算法。这种算法的核心像一种卷包裹的操作，比如说我们把每个点当成墙上的钉子，然后我们有一个皮筋，我们直接将皮筋撑的老大，然后套在所有钉子上松手，其自动形成的形状就是要求的凸包，也是凸多边形。脑海中有没有产生这个画面？撑起皮筋的边缘点就是我们要求的关键的结点，形象的图文讲解可以参见这个帖子。

我们的目标是找到这些点，做法是先找到一个边缘点，然后按一个方向转一圈，找到所有的边缘点，当再次找到起始的边缘点时结束循环。起始点的选择方法是找横坐标最小的点，即最左边的点，如果有多个横坐标相同且最小的点也没有关系，其中任意一个都可以当作起始点。因为某个点的横坐标或纵坐标任意一个是最小或最大时，该点一定是边缘上的点。我们把这个起始点标记为first，其坐标标记为firstIdx，然后我们建立一个当前点遍历cur，初始化为first，当前点坐标curIdx，初始化为firstIdx。然后我们进行循环，我们目标是找到下一个边缘点next，初始化其为数组中的第一个点，在例子1中，起始点也是数组中的第一个点，这样cur和next重合了，不过没有关系，这是个初始化而已。好，现在两个点已经确定了，我们还需要一个点，这样三个点就可以利用叉积来求出向量间的夹角，从而根据正负来判断是否为边缘点。第三个点的选择就从数组中的第二个点开始遍历，如果遍历到了cur点，直接跳过。然后此时我们算三个点之间的叉积Cross Product，不太了解叉积的菊苣们可以google一些帖子看一看，简单的来说，就是比如有三个点A, B和C，那么叉积就是求和向量BA, BC都垂直的一个向量，等于两个向量的长度乘以夹角的正弦值。在之前那道Convex Polygon中，我们就是根据叉积来判断是否是凸多边形，要保持凸多边形，那么每三个点的叉积要同正或同负。这有什么用呢，别急，一会再说。先来说之前的cur和next重合了的情况，根据叉积的计算方法，只要有两个点重合，那么叉积就为0。比如当cur和next都是A, points[i]是B时，cross是0，此时我们判断如果points[i]到cur的距离大于

next到cur的距离的话，将next移动到points[i]。为啥要判断距离呢，我们假设现在有种情况，cur是D，next是E，points[i]是F，此时的cross算出是0，而且FD的距离大于ED的距离，则将next移动到F点，是正确的。但假如此cur是D，next是F，points[i]是E，此时cross算出来也是0，但是ED的距离小于FD的距离，所以不用讲next移动到E，这也make sense。



好，还有两种情况也需要移动next，一种是当next点和cur点相同的时候直接移动next到points[i]，其实这种情况已经在上面的分析中cover了，所以这个判断有没有都一样，有的话能省几步距离的计算吧。还有一种情况是cross大于0的时候，要找凸多边形，cross必须同正负，如果我们设定cross大于0移动next，那么就是逆时针来找边缘点。当我们算出来了next后，如果不存在三点共线的情况，我们可以直接将next存入结果res中，但是有共线点的话，我们只能遍历所有的点，再次计算cross，如果为0的话，说明共线，则加入结果res中。在大神的帖子中用的是Set可以自动取出重复，C++版本的应该使用指针的Point，这样才能让set的插入函数work，不加指针的话就不能用set了，那只能手动去重复了，写个去重复的子函数来filter一下吧，参见代码如下：

```

1 class Solution {
2 public:
3     vector<Point> outerTrees(vector<Point>& points) {
4         vector<Point> res;
5         Point first = points[0];
6         int firstIdx = 0, n = points.size();
7         for (int i = 1; i < n; ++i) {
8             if (points[i].x < first.x) {
9                 first = points[i];
10                firstIdx = i;
11            }
12        }
13        res.push_back(first);
14        Point cur = first;
15        int curIdx = firstIdx;
16        while (true) {
17            Point next = points[0];
18            int nextIdx = 0;
19            for (int i = 1; i < n; ++i) {
20                if (i == curIdx) continue;
21                int cross = crossProduct(cur, points[i], next);
22                if (nextIdx == curIdx || cross > 0 || (cross == 0 && dist(points[i], cur) >
23 dist(next, cur))) {
24                    next = points[i];
25                    nextIdx = i;
26                }
27            }
28            for (int i = 0; i < n; ++i) {
29                if (i == curIdx) continue;
30                int cross = crossProduct(cur, points[i], next);
31                if (cross == 0) {
32                    if (check(res, points[i])) res.push_back(points[i]);
33                }
34            }
35            cur = next;
36            curIdx = nextIdx;
37            if (curIdx == firstIdx) break;
38        }
39        return res;
40    }
41    int crossProduct(Point A, Point B, Point C) {
42        int BAx = A.x - B.x;
43        int BAy = A.y - B.y;
44        int BCx = C.x - B.x;
45        int BCy = C.y - B.y;
46        return BAx * BCy - BAy * BCx;
47    }
48    int dist(Point A, Point B) {
49        return (A.x - B.x) * (A.x - B.x) + (A.y - B.y) * (A.y - B.y);
50    }
51    bool check(vector<Point>& res, Point p) {
52        for (Point r : res) {
53            if (r.x == p.x && r.y == p.y) return false;
54        }
55        return true;
56    }
57};

```

578. 设计内存文件系统

Design an in-memory file system to simulate the following functions:

ls: Given a path in string format. If it is a file path, return a list that only contains this file's name. If it is a directory path, return the list of file and directory names in this directory. Your output (file and directory names together) should be in lexicographic order.

mkdir: Given a directory path that does not exist, you should make a new directory according to the path. If the middle directories in the path don't exist either, you should create them as well. This function has void return type.

addContentToFile: Given a file path and file content in string format. If the file doesn't exist, you need to create that file containing given content. If the file already exists, you need to append given content to original content. This function has void return type.

readContentFromFile: Given a file path, return its content in string format.

这道题让我们设计一个内存文件系统，实现显示当前文件，创建文件，添加内容到文件，读取文件内容等功能，感觉像是模拟一个terminal的一些命令。这道题比较tricky的地方是ls这个命令，题目中的例子其实不能很好的展示出ls的要求，其对文件和文件夹的处理方式是不同的。由于这里面的文件没有后缀，所以最后一个字符串有可能是文件，也有可能是文件夹。比如a/b/c，那么最后的c有可能是文件夹，也有可能好是文件，如果c是文件夹的话，ls命令要输出文件夹c中的所有文件和文件夹，而当c是文件的话，只需要输出文件c即可。另外需要注意的是在创建文件夹的时候，路径上没有的文件夹都要创建出来，还有就是在给文件添加内容时，路径中没有的文件夹都要创建出来。论坛上这道题的高票解法都新建了一个自定义类，但是博主一般不喜欢用自定义类来解题，而且感觉那些使用了自定义类的解法并没有更简洁易懂，所以这里博主就不创建自定义类了，而是使用两个哈希表来做，其中dirs建立了路径和其对应的包含所有文件和文件夹的集合之间的映射，files建立了文件的路径跟其内容之间的映射。

最开始时将根目录"/"放入dirs中，然后看ls的实现方法，如果该路径存在于files中，说明最后一个字符串是文件，那么我们将文件名拿出来返回即可，如果不存在，说明最后一个字符串是文件夹，那么我们到dirs中取出该文件夹内所有的东西返回即可。再来看mkdir函数，我们的处理方法就是根据"/"来分隔分隔字符串，如果是Java，那么直接用String自带的split函数就好了，但是C++没有Java那么多自带函数，所以只能借助字符串流类来处理，处理方法就是将每一层的路径分离出来，然后将该层的文件或者文件夹加入对应的集合中，注意的地方就是处理根目录时，要先加上"/"，其他情况都是后加。下面来看addContentToFile函数，首先分离出路径和文件名，如果路径为空，说明是根目录，需要加上"/"，然后看这个路径是否已经在dirs中存在，如果不存在，调用mkdir来创建该路径，然后把文件加入该路径对应的集合中，再把内容加入该文件路径的映射中。最后的读取文件内容就相当简单了，直接在files中返回即可，参见代码如下：

```

1 class FileSystem {
2 public:
3     FileSystem() {
4         dirs["/"];
5     }
6
7     vector<string> ls(string path) {
8         if (files.count(path)) {
9             int idx = path.find_last_of('/');
10            return {path.substr(idx + 1)};
11        }
12        auto t = dirs[path];
13        return vector<string>(t.begin(), t.end());
14    }
15
16    void mkdir(string path) {
17        istringstream is(path);
18        string t = "", dir = "";
19        while (getline(is, t, '/')) {
20            if (t.empty()) continue;
21            if (dir.empty()) dir += "/";
22            dirs[dir].insert(t);
23            if (dir.size() > 1) dir += "/";
24            dir += t;
25        }
26    }
27
28    void addContentToFile(string filePath, string content) {
29        int idx = filePath.find_last_of('/');
30        string dir = filePath.substr(0, idx);
31        string file = filePath.substr(idx + 1);
32        if (dir.empty()) dir = "/";
33        if (!dirs.count(dir)) mkdir(dir);
34        dirs[dir].insert(file);
35        files[filePath].append(content);
36    }
37
38    string readContentFromFile(string filePath) {
39        return files[filePath];
40    }
41
42 private:
43     unordered_map<string, set<string>> dirs;
44     unordered_map<string, string> files;
45 };

```

579. N-ary Tree Preorder Traversal

Given an n-ary tree, return the preorder traversal of its nodes' values.

```
1 class Solution {
2 public:
3     vector<int> preorder(Node* root) {
4         /*
5             if(!root)
6                 return vector<int>();
7             vector<int> res;
8             res.push_back(root->val);
9             for(int i(0);i<root->children.size();i++){
10                 vector<int> cur=preorder(root->children[i]);
11                 if(cur.size()>0)
12                     res.insert(res.end(),cur.begin(),cur.end());
13             }
14             return res;
15         */
16
17         vector<int> v;
18         stack<Node*> s;
19         if (!root) return v;
20         if (root->children.size() == 0) {
21             v.push_back(root->val);
22             return v;
23         }
24         s.push(root);
25         while (s.size() > 0) {
26             Node* c = s.top();
27             s.pop();
28             v.push_back(c->val);
29             for (int i = c->children.size() - 1; c != nullptr && i > -1; i--) {
30                 s.push(c->children[i]);
31             }
32         }
33         return v;
34     }
35 }
```

580. N-ary Tree Postorder Transversal

Given an n-ary tree, return the postorder traversal of its nodes' values.

```

1 class Solution {
2 public:
3     vector<int> postorder(Node* root) {
4         if(root==NULL) return {};
5         vector<int> res;
6         stack<Node*> stk;
7         stk.push(root);
8         while(!stk.empty())
9         {
10             Node* temp=stk.top();
11             stk.pop();
12             for(int i=0;i<temp->children.size();i++)
13                 stk.push(temp->children[i]);
14             res.push_back(temp->val);
15         }
16         reverse(res.begin(), res.end());
17         return res;
18     }
19 };

```

581. 标签验证器

Given a string representing a code snippet, you need to implement a tag validator to parse the code and return whether it is valid. A code snippet is valid if all the following rules hold:

The code must be wrapped in a valid closed tag. Otherwise, the code is invalid.

A closed tag (not necessarily valid) has exactly the following format :

<TAG_NAME>TAG_CONTENT</TAG_NAME>. Among them, <TAG_NAME> is the start tag, and </TAG_NAME> is the end tag. The TAG_NAME in start and end tags should be the same. A closed tag is valid if and only if the TAG_NAME and TAG_CONTENT are valid.

A valid TAG_NAME only contain upper-case letters, and has length in range [1,9]. Otherwise, the TAG_NAME is invalid.

A valid TAG_CONTENT may contain other valid closed tags, cdata and any characters (see note1) EXCEPT unmatched <, unmatched start and end tag, and unmatched or closed tags with invalid TAG_NAME. Otherwise, the TAG_CONTENT is invalid.

A start tag is unmatched if no end tag exists with the same TAG_NAME, and vice versa. However, you also need to consider the issue of unbalanced when tags are nested.

A < is unmatched if you cannot find a subsequent >. And when you find a < or </, all the subsequent characters until the next > should be parsed as TAG_NAME (not necessarily valid).

The cdata has the following format : <![CDATA[CDATA_CONTENT]]>. The range of CDATA_CONTENT is defined as the characters between <![CDATA[and the first subsequent]]>.

CDATA_CONTENT may contain any characters. The function of cdata is to forbid the validator to parse CDATA_CONTENT, so even it has some characters that can be parsed as tag (no matter valid or invalid), you should treat it as regular characters.

由于博主的修修补补使得博主自己的代码看起来不 elegant，所以博主借鉴了论坛上 dengzhizhang 大神的解法，果然还是用 find 函数写起来简洁，而不是用 while 函数来一个一个的找。下面的代码结构还是比较清晰的，首先对于这种成对匹配的问题肯定是要用栈 stack 的，就像之前的匹配括号的问题。那么我们来遍历 code 字符串，首先是判断，如果当前遍历到的字符非首字符，并且栈为空，那么直接返回 false。虽然只是短短的一句但其实非常的重要，这句就排除了很多错误情况，比如开头结尾不是标签的情况，以及没有标签的情况，和开头的标签在中间就闭合了情况等等，非常 powerful 的一句判断。然后我们来处理包含 CDATA 的情况，当然是要先匹配到 "<![CDATA["，然后我们用 find 来找结束标志 "]>"，如果没找到，直接返回 false，找到了点话就继续遍历，顺便把当前遍历的位置移到结束标志符的最后一位上。

如果我们只匹配到了 "</ "，说明是个结束标签，那么我们用 find 来找到右尖括号 '>'，如果没找到直接返回 false，找到了就把 tag 到内容提出来，然后看此时的 stack，如果 stack 为空，或者栈顶元素不等于 tag，直接返回 false，否则就将栈顶元素取出。

如果我们只匹配到了 "< "，说明是个起始标签，还是要找右尖括号，如果找不到，或者标签的长度为 0，或者大于 9 了，直接返回 true。然后遍历标签的每一位，如果不全是大些字母，返回 false，否则就把 tag 压入栈。那么你可能会有疑问，为啥在处理结束标签时，没有这些额外的判断呢，因为结束标签要和栈顶元素比较，栈里的标签肯定都是合法的，所以如果结束标签不合法，那么肯定不相等，也就直接返回 false 了。最后我们看栈是否为空，如果不为空，说明有未封闭的标签，返回 false。参见代码如下：

```

1 class Solution {
2 public:
3     bool isValid(string code) {
4         stack<string> st;
5         for (int i = 0; i < code.size(); ++i) {
6             if (i > 0 && st.empty()) return false;
7             if (code.substr(i, 9) == "<![CDATA[") {
8                 int j = i + 9;
9                 i = code.find("]]>", j);
10                if (i < 0) return false;
11                i += 2;
12            } else if (code.substr(i, 2) == "</") {
13                int j = i + 2;
14                i = code.find(">", j);
15                if (i < 0) return false;
16                string tag = code.substr(j, i - j);
17                if (st.empty() || st.top() != tag) return false;
18                st.pop();
19            } else if (code.substr(i, 1) == "<") {
20                int j = i + 1;
21                i = code.find(">", j);
22                if (i < 0 || i == j || i - j > 9) return false;
23                for (int k = j; k < i; ++k) {
24                    if (code[k] < 'A' || code[k] > 'Z') return false;
25                }
26                string tag = code.substr(j, i - j);
27                st.push(tag);
28            }
29        }
30        return st.empty();
31    }
32 };

```

CPP

这道题只是判断 html 里最简单的标签，并没有加上 js 和 css 的东西，就已经是个 Hard 的题目了。论坛看到有大神们用正则匹配来做，代码太简洁了，但是博主看不太懂那个正则表达式，囧~ 所以没有把那种解法贴上来，大家可以看看 zqfan 大神的帖子，如果哪位看官大神看懂了，请给博主讲一下，博主可以贴上来供大家参考，多谢~

582. 分数加减法

Given a string representing an expression of fraction addition and subtraction, you need to return the calculation result in string format. The final result should be irreducible fraction. If your final result is an integer, say 2, you need to change it to the format of fraction that has denominator 1. So in this case, 2 should be converted to 2/1.

这道题让我们做分数的加减法，给了我们一个分数加减法式子的字符串，然我们算出结果，结果当然还是用分数表示了。那么其实这道题主要就是字符串的拆分处理，再加上一点中学的数学运算的知识就可以了。这里我们使用字符流处理类来做，每次按顺序读入一个数字，一个字符，和另一个数字。分别代表了分子，除号，分母。我们初始化分子为0，分母为1，这样就可以进行任何加减法了。中学数学告诉我们必须将分母变为同一个数，分子才能相加，为了简便，我们不求最小公倍数，而是直接乘上另一个数的分母，然后相加。不过得到的结果需要化简一下，我们求出分子分母的最大公约数，记得要取绝对值，然后分子分母分别除以这个最大公约数就是最后的结果了，参见代码如下：

```

1 class Solution {
2 public:
3     string fractionAddition(string expression) {
4         istringstream is(expression);
5         int num = 0, dem = 0, A = 0, B = 1;
6         char c;
7         while (is >> num >> c >> dem) {
8             A = A * dem + num * B;
9             B *= dem;
10            int g = abs(gcd(A, B));
11            A /= g;
12            B /= g;
13        }
14        return to_string(A) + "/" + to_string(B);
15    }
16    int gcd(int a, int b) {
17        return (b == 0) ? a : gcd(b, a % b);
18    }
19 };

```

CPP

583. 验证正方形

Given the coordinates of four points in 2D space, return whether the four points could construct a square.

The coordinate (x,y) of a point is represented by an integer array with two integers.

这道题给了我们四个点，让我们验证这四个点是否能组成一个正方形，刚开始博主考虑的方法是想判断四个角是否是直角，但即便四个角都是直角，也不能说明一定就是正方形，还有可能是矩形。还得判断各边是否相等。其实我们可以仅通过边的关系的来判断是否是正方形，根据初中几何的知识我们知道正方形的四条边相等，两条对角线相等，满足这两个条件的四边形一定是正方形。那么这样就好办了，我们只需要对四个点，两两之间算距离，如果计算出某两个点之间距离为0，说明两点重合了，直接返回false，如果不为0，那么我们就建立距离和其出现次数之间的映射，最后如果我们只得到了两个不同的距离长度，那么就说明是正方形了，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool validSquare(vector<int>& p1, vector<int>& p2, vector<int>& p3, vector<int>& p4) {
4         unordered_map<int, int> m;
5         vector<vector<int>> v{p1, p2, p3, p4};
6         for (int i = 0; i < 4; ++i) {
7             for (int j = i + 1; j < 4; ++j) {
8                 int x1 = v[i][0], y1 = v[i][1], x2 = v[j][0], y2 = v[j][1];
9                 int dist = (x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2);
10                if (dist == 0) return false;
11                ++m[dist];
12            }
13        }
14        return m.size() == 2;
15    }
16 };

```

我们其实不用建立映射，直接用个集合set来放距离就行了，如果最后集合中不存在0，且里面只有两个数的时候，说明是正方形，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool validSquare(vector<int>& p1, vector<int>& p2, vector<int>& p3, vector<int>& p4) {
4         unordered_set<int> s{d(p1, p2), d(p1, p3), d(p1, p4), d(p2, p3), d(p2, p4), d(p3,
5 p4)};
6         return !s.count(0) && s.size() == 2;
7     }
8     int d(vector<int>& p1, vector<int>& p2) {
9         return (p1[0] - p2[0]) * (p1[0] - p2[0]) + (p1[1] - p2[1]) * (p1[1] - p2[1]);
10    }
11 };

```

584. 最长和谐子序列

We define a harmonious array is an array where the difference between its maximum value and its minimum value is exactly 1.

Now, given an integer array, you need to find the length of its longest harmonious subsequence among all its possible subsequences.

Example 1:

Input: [1,3,2,2,5,2,3,7]

Output: 5

Explanation: The longest harmonious subsequence is [3,2,2,2,3].

这道题给了我们一个数组，让我们找出最长的和谐子序列，关于和谐子序列就是序列中数组的最大最小差值均为1。由于这里只是让我们求长度，并不需要返回具体的子序列。所以我们可以对数组进行排序，那么实际上我们只要找出来相差为1的两个数的总共出现个数就是一个和谐子序列的长度了。明白了这一点，我们就可以建立一个数字和其出现次数之间的映射，利用map的自动排序的特性，那么我们遍历map的时候就是从小往大开始遍历，我们从第二个映射对开始遍历，每次跟其前面的映射对比较，如果二者的数字刚好差1，那么就把二个数字的出现的次数相加并更新结果res即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int findLHS(vector<int>& nums) {
4         if (nums.empty()) return 0;
5         int res = 0;
6         map<int, int> m;
7         for (int num : nums) ++m[num];
8         for (auto it = next(m.begin()); it != m.end(); ++it) {
9             auto pre = prev(it);
10            if (it->first == pre->first + 1) {
11                res = max(res, it->second + pre->second);
12            }
13        }
14        return res;
15    }
16 };

```

其实我们并不用向上面那种解法那样用next和prev来移动迭代器，我们遍历每个数字的时候，只需在map中查找该数字加1是否存在，存在就更新结果res，这样更简单一些，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int findLHS(vector<int>& nums) {
4         int res = 0;
5         map<int, int> m;
6         for (int num : nums) ++m[num];
7         for (auto a : m) {
8             if (m.count(a.first + 1)) {
9                 res = max(res, m[a.first] + m[a.first + 1]);
10            }
11        }
12        return res;
13    }
14 };

```

585. Big Countries

There is a table World

name	continent	area	population	gdp
Afghanistan	Asia	652230	25500100	20343000
Albania	Europe	28748	2831741	12960000
Algeria	Africa	2381741	37100000	188681000
Andorra	Europe	468	78115	3712000
Angola	Africa	1246700	20609294	100990000

A country is big if it has an area of bigger than 3 million square km or a population of more than 25 million.

Write a SQL solution to output big countries' name, population and area.

```

1 | SELECT name, population, area
2 | FROM World
3 | WHERE area > 3000000
4 |
5 | UNION
6 |
7 | SELECT name, population, area
8 | FROM World
9 | WHERE population > 25000000

```

586. Classes More Than 5 Students

There is a table courses with columns: student and class

Please list out all classes which have more than or equal to 5 students.

For example, the table:

student	class
A	Math
B	English
C	Math
D	Biology
E	Math
F	Computer
G	Math
H	Math
I	Math

```

1 | select class from courses group by class having count(distinct student) >= 5

```

587. Friend Requests I: Overall Acceptance Rate

In social network like Facebook or Twitter, people send friend requests and accept others' requests as well. Now given two tables as below:

Table: friend_request

sender_id	send_to_id	request_date
1	2	2016_06-01
1	3	2016_06-01
1	4	2016_06-01
2	3	2016_06-02
3	4	2016-06-09

Table: request_accepted

requester_id	accepter_id	accept_date
1	2	2016_06-03
1	3	2016-06-08
2	3	2016-06-08
3	4	2016-06-09
3	4	2016-06-10

Write a query to find the overall acceptance rate of requests rounded to 2 decimals, which is the number of acceptance divide the number of requests.

```

1 select
2 round(
3     ifnull(
4         (select count(*) from (select distinct requester_id, accepter_id from
5 request_accepted) as A)
6         /
7         (select count(*) from (select distinct sender_id, send_to_id from
8 friend_request) as B),
8         0)
9     , 2) as accept_rate;

```

588. 范围相加之二

Given an $m * n$ matrix M initialized with all 0's and several update operations.

Operations are represented by a 2D array, and each operation is represented by an array with two positive integers a and b , which means $M[i][j]$ should be added by one for all $0 \leq i < a$ and $0 \leq j < b$.

You need to count and return the number of maximum integers in the matrix after performing all the operations.

这道题看起来像是之前那道Range Addition的拓展，但是感觉实际上更简单一些。每次在ops中给定我们一个横纵坐标，将这个子矩形范围内的数字全部自增1，让我们求最大数字的个数。原数组初始化均为0，那么如果ops为空，没有任何操作，那么直接返回 $m*n$ 即可，我们可以用一个优先队列来保存最大数字矩阵的横纵坐标，我们可以通过举些例子发现，只有最小数字组成的边界中的数字才会被每次更新，所以我们想让最小的数字到队首，更优先队列的排序机制是大的数字在队首，所以我们对其取相反数，这样我们最后取出两个队列的队首数字相乘即为结果，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     int maxCount(int m, int n, vector<vector<int>>& ops) {
4         if (ops.empty() || ops[0].empty()) return m * n;
5         priority_queue<int> r, c;
6         for (auto op : ops) {
7             r.push(-op[0]);
8             c.push(-op[1]);
9         }
10        return r.top() * c.top();
11    }
12 };

```

我们可以对空间进行优化，不使用优先队列，而是每次用ops中的值来更新m和n，取其中较小值，这样遍历完成后，m和n就是最大数矩阵的边界了，参见代码如下：

解法2:

```

1 class Solution {
2 public:
3     int maxCount(int m, int n, vector<vector<int>>& ops) {
4         for (auto op : ops) {
5             m = min(m, op[0]);
6             n = min(n, op[1]);
7         }
8         return m * n;
9     }
10 };

```

589. 两个表单的最小坐标和

Suppose Andy and Doris want to choose a restaurant for dinner, and they both have a list of favorite restaurants represented by strings.

You need to help them find out their common interest with the least list index sum. If there is a choice tie between answers, output all of them with no order requirement. You could assume there always exists an answer.

这道题给了我们两个字符串数组，让我们找到坐标位置之和最小的相同的字符串。那么对于这种数组项和其坐标之间关系的题，最先考虑到的就是要建立数据与其位置坐标之间的映射。我们建立list1的值和坐标的之间的映射，然后遍历list2，如果当前遍历到的字符串在list1中也出现了，那么我们计算两个的坐标之和，如果跟我们维护的最小坐标和mn相同，那么将这个字符串加入结果res中，如果比mn小，那么mn更新为这个较小值，然后将结果res清空并加入这个字符串，参见代码如下：

```

1 class Solution {
2 public:
3     vector<string> findRestaurant(vector<string>& list1, vector<string>& list2) {
4         vector<string> res;
5         unordered_map<string, int> m;
6         int mn = INT_MAX, n1 = list1.size(), n2 = list2.size();
7         for (int i = 0; i < n1; ++i) m[list1[i]] = i;
8         for (int i = 0; i < n2; ++i) {
9             if (m.count(list2[i])) {
10                 int sum = i + m[list2[i]];
11                 if (sum == mn) res.push_back(list2[i]);
12                 else if (sum < mn) {
13                     mn = sum;
14                     res = {list2[i]};
15                 }
16             }
17         }
18         return res;
19     }
20 };

```

590. 非负整数不包括连续的1

Given a positive integer n, find the number of non-negative integers less than or equal to n, whose binary representations do NOT contain consecutive ones.

这道题给了我们一个数字，让我们求不大于这个数字的所有数字中，其二进制的表示形式中没有连续1的个数。根据题目中的例子也不难理解题意。我们首先来考虑二进制的情况，对于1来说，有0和1两种，对于11来说，有00, 01, 10，三种情况，那么有没有规律可寻呢，其实是有，我们可以参见这个帖子，这样我们就可以通过DP的方法求出长度为k的二进制数的无连续1的数

字个数。由于题目给我们的并不是一个二进制数的长度，而是一个二进制数，比如100，如果我们按长度为3的情况计算无连续1点个数个数，就会多计算101这种情况。所以我们的目标是要将大于num的情况去掉。下面从头来分析代码，首先我们要把十进制数转为二进制数，将二进制数存在一个字符串中，并统计字符串的长度。然后我们利用这个帖子中的方法，计算该字符串长度的二进制数所有无连续1的数字个数，然后我们从倒数第二个字符开始往前遍历这个二进制数字符串，如果当前字符和后面一个位置的字符均为1，说明我们并没有多计算任何情况，不明白的可以带例子来看。如果当前字符和后一个位置的字符均为0，说明我们有多计算一些情况，就像之前举的100这个例子，我们就多算了101这种情况。我们怎么确定多了多少种情况呢，假如给我们的数字是8，二进制为1000，我们首先按长度为4算出所有情况，共8种。仔细观察我们十进制转为二进制字符串的写法，发现转换结果跟真实的二进制数翻转了一下，所以我们的t为"0001"，那么我们从倒数第二位开始往前遍历，到i=1时，发现有两个连续的0出现，那么i=1这个位置上能出现1的次数，就到one数组中去找，那么我们减去1，减去的就是0101这种情况，再往前遍历，i=0时，又发现两个连续0，那么i=0这个位置上能出1的次数也到one数组中去找，我们再减去1，减去的是1001这种情况，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int findIntegers(int num) {
4         int cnt = 0, n = num;
5         string t = "";
6         while (n > 0) {
7             ++cnt;
8             t += (n & 1) ? "1" : "0";
9             n >>= 1;
10        }
11        vector<int> zero(cnt), one(cnt);
12        zero[0] = 1; one[0] = 1;
13        for (int i = 1; i < cnt; ++i) {
14            zero[i] = zero[i - 1] + one[i - 1];
15            one[i] = zero[i - 1];
16        }
17        int res = zero[cnt - 1] + one[cnt - 1];
18        for (int i = cnt - 2; i >= 0; --i) {
19            if (t[i] == '1' && t[i + 1] == '1') break;
20            if (t[i] == '0' && t[i + 1] == '0') res -= one[i];
21        }
22        return res;
23    }
24 };

```

CPP

下面这种解法其实蛮有意思的，其实长度为k的二进制数字符串没有连续的1的个数是一个斐波那契数列f(k)。比如当k=5时，二进制数的范围是00000-11111，我们可以将其分为两个部分，00000-01111和10000-10111，因为任何大于11000的数字都是不成立的，因为有开头已经有了两个连续1。而我们发现其实00000-01111就是f(4)，而10000-10111就是f(3)，所以f(5) = f(4) + f(3)，这就是一个斐波那契数列啦。那么我们要做的首先就是建立一个这个数组，方便之后直接查值。我们从给定数字的最高位开始遍历，如果某一位是1，后面有k位，就加上f(k)，因为如果我们把当前位变成0，那么后面k位就可以直接从斐波那契数列中取值了。然后标记pre为1，再往下遍历，如果遇到0位，则pre标记为0。如果当前位是1，pre也是1，那么直接返回结果。最后循环退出后我们要加上数字本身这种情况，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int findIntegers(int num) {
4         int res = 0, k = 31, pre = 0;
5         vector<int> f(32, 0);
6         f[0] = 1; f[1] = 2;
7         for (int i = 2; i < 31; ++i) {
8             f[i] = f[i - 2] + f[i - 1];
9         }
10        while (k >= 0) {
11            if (num & (1 << k)) {
12                res += f[k];
13                if (pre) return res;
14                pre = 1;
15            } else pre = 0;
16            --k;
17        }
18        return res + 1;
19    }
20 };

```

591. Human Traffic of Stadium

X city built a new stadium, each day many people visit it and the stats are saved as these columns: id, date, people

Please write a query to display the records which have 3 or more consecutive rows and the amount of people more than 100(inclusive).

```

1 select distinct t1.*
2 from stadium t1, stadium t2, stadium t3
3 where t1.people >= 100 and t2.people >= 100 and t3.people >= 100
4 and
5 (
6     (t1.id - t2.id = 1 and t1.id - t3.id = 2 and t2.id - t3.id = 1) -- t1, t2, t3
7     or
8     (t2.id - t1.id = 1 and t2.id - t3.id = 2 and t1.id - t3.id = 1) -- t2, t1, t3
9     or
10    (t3.id - t2.id = 1 and t2.id - t1.id = 1 and t3.id - t1.id = 2) -- t3, t2, t1
11 )
12 order by t1.id
13 ;

```

592. Friend Requests II: Who Has Most Friend?

In social network like Facebook or Twitter, people send friend requests and accept others' requests as well.

Table request_accepted holds the data of friend acceptance, while requester_id and accepter_id both are the id of a person.

requester_id	accepter_id	accept_date
1	2	2016-06-03
1	3	2016-06-08
2	3	2016-06-08
3	4	2016-06-09

Write a query to find the the people who has most friends and the most friends number. For the sample data above, the result is:

id	num
3	3

```

1 select ids as id, cnt as num
2 from
3 (
4 select ids, count(*) as cnt
5 from
6 (
7 select requester_id as ids from request_accepted
8 union all
9 select accepter_id from request_accepted
10 ) as tbl1
11 group by ids
12 ) as tbl2
13 order by cnt desc
14 limit 1
15 ;

```

SQL

593. Consecutive Available Seats

Several friends at a cinema ticket office would like to reserve consecutive available seats. Can you help to query all the consecutive available seats order by the seat_id using the following cinema table?

seat_id	free
1	1
2	0
3	1
4	1
5	1

Your query should return the following result for the sample case above.

seat_id
3
4
5

```

1 | select distinct a.seat_id
2 | from cinema a join cinema b
3 | on abs(a.seat_id - b.seat_id) = 1
4 | and a.free = true and b.free = true
5 | order by a.seat_id
6 |

```

594. 设计压缩字符串的迭代器

Design and implement a data structure for a compressed string iterator. It should support the following operations: next and hasNext.

The given compressed string will be in the form of each letter followed by a positive integer representing the number of this letter existing in the original uncompressed string.

next() - if the original string still has uncompressed characters, return the next letter;
Otherwise return a white space.

hasNext() - Judge whether there is any letter needs to be uncompressed.

Note:

Please remember to RESET your class variables declared in StringIterator, as static/class variables are persisted across multiple test cases. Please see here for more details.

这道题给了我们一个压缩字符串，就是每个字符后面跟上其出现的次数，这里就算只出现一次，后面还是要加上1，那么其实如果当字符串很好有连续字符的时候，压缩字符串反而要比原字符串长。不过这题的重点不在于压缩字符串本身，而是让我们设计一个压缩字符串的迭代器，那么实际上是要我们根据压缩字符串来输出原字符串中的所有字符。那么我们关键就是要取出每个字符和其出现的次数，每当调用一次next，次数减1，如果减到0了，我们就要取出下一个字符和其出现的次数。我们要用个私有变量s来保存原字符串，然后用个变量i来记录当前遍历到的位置，变量c为当前处理的字符，变量cnt为字符c的当前次数。变量i的初始化为0，指向第一个字符，我们在hasNext()函数中，现将s[i]存入c，然后i自增1，然后我们用while循环取出所有的数字，存入cnt中。在next()函数中，如果hasNext()返回true，那么cnt就自减1，返回c；如果hasNext()返回false，那么字节返回空字符。在hasNext()函数中首先判断cnt的值，如果大于0，直接返回true，参见代码如下：

解法1：

```

1 class StringIterator {
2 public:
3     StringIterator(string compressedString) {
4         s = compressedString;
5         n = s.size();
6         i = 0;
7         cnt = 0;
8         c = ' ';
9     }
10
11    char next() {
12        if (hasNext()) {
13            --cnt;
14            return c;
15        }
16        return ' ';
17    }
18
19    bool hasNext() {
20        if (cnt > 0) return true;
21        if (i >= n) return false;
22        c = s[i++];
23        while (i < n && s[i] >= '0' && s[i] <= '9') {
24            cnt = cnt * 10 + s[i++] - '0';
25        }
26        return true;
27    }
28
29 private:
30     string s;
31     int n, i, cnt;
32     char c;
33 };

```

我们可以用C++中的字符流类来处理字符串，写法非常的简洁，可以少定义一些变量，在hasNext()函数中，如果cnt为0了，那么我们用字符流类直接读出下一个字符和次数，然后看是否能读出大于0的次数来返回真假值，参见代码如下：

解法2：

```

1 class StringIterator {
2 public:
3     StringIterator(string compressedString) {
4         is = istringstream(compressedString);
5         cnt = 0;
6         c = ' ';
7     }
8
9     char next() {
10        if (hasNext()) {
11            --cnt;
12            return c;
13        }
14        return ' ';
15    }
16
17    bool hasNext() {
18        if (cnt == 0) {
19            is >> c >> cnt;
20        }
21        return cnt > 0;
22    }
23
24 private:
25     istringstream is;
26     int cnt;
27     char c;
28 };

```

下面这种解法还是用字符流类，和上面方法不同的地方是，在构建函数中完成了所有字符和次数的拆分，然后字符和其次数组成一个pair，加入一个队列queue中，这样我们每次处理的时候就直接去queue中取值就行了，这样hasNext()函数就变得非常简洁，只需要判断队列queue是否为空即可，参见代码如下：

解法3：

```

1 class StringIterator {
2 public:
3     StringIterator(string compressedString) {
4         istringstream is(compressedString);
5         int cnt = 0;
6         char c = ' ';
7         while (is >> c >> cnt) {
8             q.push({c, cnt});
9         }
10    }
11
12    char next() {
13        if (hasNext()) {
14            auto &t = q.front();
15            if (--t.second == 0) q.pop();
16            return t.first;
17        }
18        return ' ';
19    }
20
21    bool hasNext() {
22        return !q.empty();
23    }
24
25 private:
26     queue<pair<char, int>> q;
27 };

```

595. 可以放置花

Suppose you have a long flowerbed in which some of the plots are planted and some are not. However, flowers cannot be planted in adjacent plots - they would compete for water and both would die.

Given a flowerbed (represented as an array containing 0 and 1, where 0 means empty and 1 means not empty), and a number n, return if n new flowers can be planted in it without violating the no-adjacent-flowers rule.

这道题给了我们一个01数组，其中1表示已经放了花，0表示可以放花的位置，但是有个限制条件是不能有相邻的花。那么我们来看如果是一些简单的例子，如果有3个连续的零，000，能放几盆花呢，其实是要取决约左右的位置的，如果是10001，那么只能放1盆，如果左右是边界的花，那么就能放两盆，101，所以如果我们想通过计算连续0的个数，然后直接算出能放花的个数，就必须要对边界进行处理，处理方法是如果首位置是0，那么前面再加上个0，如果末位置是0，就在最后面再加上个0。这样处理之后我们就默认连续0的左右两边都是1了，这样如果有k个连续0，那么就可以通过 $(k-1)/2$ 来快速计算出能放的花的数量，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool canPlaceFlowers(vector<int>& flowerbed, int n) {
4         if (flowerbed.empty()) return false;
5         if (flowerbed[0] == 0) flowerbed.insert(flowerbed.begin(), 0);
6         if (flowerbed.back() == 0) flowerbed.push_back(0);
7         int len = flowerbed.size(), cnt = 0, sum = 0;
8         for (int i = 0; i <= len; ++i) {
9             if (i < len && flowerbed[i] == 0) ++cnt;
10            else {
11                sum += (cnt - 1) / 2;
12                cnt = 0;
13            }
14        }
15        return sum >= n;
16    }
17 };

```

我们也可以直接通过修改flowerbed的值来做，我们遍历花床，如果某个位置为0，我们就看其前面一个和后面一个位置的值，注意处理首位置和末位置的情况，如果pre和next均为0，那么说明当前位置可以放花，我们修改flowerbed的值，并且n自减1，最后看n是否小于等于0，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool canPlaceFlowers(vector<int>& flowerbed, int n) {
4         for (int i = 0; i < flowerbed.size(); ++i) {
5             if (n == 0) return true;
6             if (flowerbed[i] == 0) {
7                 int next = (i == flowerbed.size() - 1 ? 0 : flowerbed[i + 1]);
8                 int pre = (i == 0 ? 0 : flowerbed[i - 1]);
9                 if (next + pre == 0) {
10                     flowerbed[i] = 1;
11                     --n;
12                 }
13             }
14         }
15         return n <= 0;
16     }
17 };

```

下面这种方法跟上面的方法类似，为了不特殊处理首末位置，直接先在首尾各加了一个0，然后就三个三个的来遍历，如果找到了三个连续的0，那么n自减1，i自增1，这样相当于i一下向后跨了两步，可以自行带例子检验，最后还是看n是否小于等于0，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     bool canPlaceFlowers(vector<int>& flowerbed, int n) {
4         flowerbed.insert(flowerbed.begin(), 0);
5         flowerbed.push_back(0);
6         for (int i = 1; i < flowerbed.size() - 1; ++i) {
7             if (n == 0) return true;
8             if (flowerbed[i - 1] + flowerbed[i] + flowerbed[i + 1] == 0) {
9                 --n;
10                ++i;
11            }
12        }
13        return n <= 0;
14    }
15 };

```

596. 根据二叉树创建字符串

You need to construct a string consists of parenthesis and integers from a binary tree with the preorder traversing way.

The null node needs to be represented by empty parenthesis pair "()". And you need to omit all the empty parenthesis pairs that don't affect the one-to-one mapping relationship between the string and the original binary tree.

这道题给我们了一个二叉树，让我们创建对应的字符串，之前有一道正好反过来的题Construct Binary Tree from String。对于二叉树的处理，递归肯定是王道啊。想想如何来实现递归函数，我们观察到题目中的例子，发现如果左子结点为空，右子结点不为空时，需要在父结点后加上个空括号，而右子结点如果不存在，或者左右子结点都不存在就不需要这么做。那我们在递归函数中，如果当前结点不存在，直接返回，然后要在当前结点值前面加上左括号，然后判断，如果左子结点不存在，而右子结点存在的话，要在结果res后加上个空括号，然后分别对左右子结点调用递归函数，调用完之后要加上右括号，形成封闭的括号。由于最外面一层的括号不需要，所以我们再返回最终结果之前要去掉首尾的括号，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     string tree2str(TreeNode* t) {
4         if (!t) return "";
5         string res = "";
6         helper(t, res);
7         return string(res.begin() + 1, res.end() - 1);
8     }
9     void helper(TreeNode* t, string& res) {
10         if (!t) return;
11         res += "(" + to_string(t->val);
12         if (!t->left && t->right) res += "()";
13         helper(t->left, res);
14         helper(t->right, res);
15         res += ")";
16     }
17 };

```

下面来看一种不用额外函数的递归写法，这种做法是一开始调用递归函数求出左右子结点的返回字符串，如果左右结果串均为空，则直接返回当前结点值；如果左子结果串为空，那么返回当前结果res，加上一个空括号，再加上放在括号中的右子结果串；如果右子结果串为空，那么发返回当前结果res，加上放在括号中的左子结果串；如果左右子结果串都存在，那么返回当前

结果，加上分别放在括号中的左右子结果串，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     string tree2str(TreeNode* t) {
4         if (!t) return "";
5         string res = to_string(t->val);
6         string left = tree2str(t->left), right = tree2str(t->right);
7         if (left == "" && right == "") return res;
8         if (left == "") return res + "(" + right + ")";
9         if (right == "") return res + "(" + left + ")";
10        return res + "(" + left + ")" + "(" + right + ")";
11    }
12};
```

CPP

下面这种解法更加简洁，由热心网友edyyy提供，思路和上面解法相同，参见代码如下：

解法3：

```
1 class Solution {
2 public:
3     string tree2str(TreeNode* t) {
4         if (!t) return "";
5         string res = to_string(t->val);
6         if (!t->left && !t->right) return res;
7         res += "(" + tree2str(t->left) + ")";
8         if (t->right) res += "(" + tree2str(t->right) + ")";
9         return res;
10    }
11};
```

CPP

597. Sales Person

Description

Given three tables: salesperson, company, orders.

Output all the names in the table salesperson, who didn't have sales to company 'RED'.

```
1 SELECT
2 s.name
3 FROM
4 salesperson s
5 WHERE
6 s.sales_id NOT IN (SELECT
7                 o.sales_id
8                 FROM
9                 orders o
10                LEFT JOIN
11                company c ON o.com_id = c.com_id
12                WHERE
13                c.name = 'RED')
14;
```

SQL

598. Tree Node

Given a table tree, id is identifier of the tree node and p_id is its parent node's id.

id	p_id
1	null
2	1
3	1
4	2
5	2

Each node in the tree can be one of three types:

Leaf: if the node is a leaf node.

Root: if the node is the root of the tree.

Inner: If the node is neither a leaf node nor a root node.

Write a query to print the node id and the type of the node. Sort your output by the node id.

The result for the above sample is:

id	Type
1	Root
2	Inner
3	Leaf
4	Leaf
5	Leaf

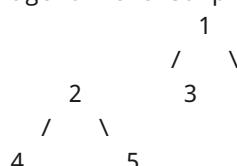
Explanation

Node '1' is root node, because its parent node is NULL and it has child node '2' and '3'.

Node '2' is inner node, because it has parent node '1' and child node '4' and '5'.

Node '3', '4' and '5' is Leaf node, because they have parent node and they don't have child node.

And here is the image of the sample tree as below:



Note

If there is only one node on the tree, you only need to output its root attributes.

```

1 SELECT
2 atree.id,
3 IF(ISNULL(atree.p_id),
4   'Root',
5   IF(atree.id IN (SELECT p_id FROM tree), 'Inner', 'Leaf')) Type
6 FROM
7 tree atree
8 ORDER BY atree.id
  
```

SQL

599. 在系统中寻找重复文件

Given a list of directory info including directory path, and all the files with contents in this directory, you need to find out all the groups of duplicate files in the file system in terms of their paths.

A group of duplicate files consists of at least two files that have exactly the same content.

A single directory info string in the input list has the following format:

```
"root/d1/d2/.../dm f1.txt(f1_content) f2.txt(f2_content) ... fn.txt(fn_content)"
```

It means there are n files (f1.txt, f2.txt ... fn.txt with content f1_content, f2_content ... fn_content, respectively) in directory root/d1/d2/.../dm. Note that n >= 1 and m >= 0. If m = 0, it means the directory is just the root directory.

The output is a list of group of duplicate file paths. For each group, it contains all the file paths of the files that have the same content. A file path is a string that has the following format:

```
"directory_path/file_name.txt"
```

LeetCode的主页又改版了，放了一些五颜六色的按钮上去了，博主个人觉得风格不太搭，还是比较喜欢之前深沉低调的风格，不过也许看久了就习惯了。来看题吧，这道题给了我们一堆字符串数组，每个字符串中包含了文件路径，文件名称和内容，让我们找到重复的文件，这里只要文件内容相同即可，不用管文件名是否相同，而且返回结果中要带上文件的路径。博主个人感觉这实际上应该算是字符串操作的题目，因为思路上并不是很难想，就是要处理字符串，把路径，文件名，和文件内容从一个字符串中拆出来，我们这里建立一个文件内容和文件路径加文件名组成的数组的映射，因为会有多个文件有相同的内容，所以我们要用数组。然后把分离出的路径和文件名拼接到一起，最后我们只要看哪些映射的数组元素个数多于1个的，就说明有重复文件，我们把整个数组加入结果res中，参见代码如下：

```
1 class Solution {
2 public:
3     vector<vector<string>> findDuplicate(vector<string>& paths) {
4         vector<vector<string>> res;
5         unordered_map<string, vector<string>> m;
6         for (string path : paths) {
7             istringstream is(path);
8             string pre = "", t = "";
9             is >> pre;
10            while (is >> t) {
11                int idx = t.find_last_of('(');
12                string dir = pre + "/" + t.substr(0, idx);
13                string content = t.substr(idx + 1, t.size() - idx - 2);
14                m[content].push_back(dir);
15            }
16        }
17        for (auto a : m) {
18            if (a.second.size() > 1) res.push_back(a.second);
19        }
20        return res;
21    }
22};
```

CPP

600. Triangle Judgement

A pupil Tim gets homework to identify whether three line segments could possibly form a triangle.

However, this assignment is very heavy because there are hundreds of records to calculate.

Could you help Tim by writing a query to judge whether these three sides can form a triangle, assuming table triangle holds the length of the three sides x, y and z.

x	y	z
13	15	30
10	20	15

```

1 | SELECT
2 | x,
3 | y,
4 | z,
5 | CASE
6 | WHEN x + y > z AND x + z > y AND y + z > x THEN 'Yes'
7 | ELSE 'No'
8 | END AS 'triangle'
9 | FROM
10| triangle
11|

```

SQL

601. 合法的三角形个数

Given an array consists of non-negative integers, your task is to count the number of triplets chosen from the array that can make triangles if we take them as side lengths of a triangle.

这道题给了我们一堆数字，问我们能组成多少个正确的三角形，我们初中就知道三角形的性质，任意两条边之和要大于第三边。那么问题其实就变成了找出所有这样的三个数字，使得任意两个数字之和都大于第三个数字。那么可以转变一下，三个数字中如果较小的两个数字之和大于第三个数字，那么任意两个数字之和都大于第三个数字，这很好证明，因为第三个数字是最大的，所以它加上任意一个数肯定大于另一个数。这样，我们就先要给数组排序，博主最先尝试了暴力破解法，结果TLE了(不要吐槽博主哈，博主就是喜欢霸王硬上弓~)，后来优化的方法是先确定前两个数，将这两个数之和sum作为目标值，然后用二分查找法来快速确定第一个小于目标值的数，这种情况属于博主之前的博客LeetCode Binary Search Summary 二分搜索法小结中总结的第二类的变形，我们找到这个临界值，那么这之前一直到j的位置之间的数都满足题意，直接加起来即可，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     int triangleNumber(vector<int>& nums) {
4         int res = 0, n = nums.size();
5         sort(nums.begin(), nums.end());
6         for (int i = 0; i < n; ++i) {
7             for (int j = i + 1; j < n; ++j) {
8                 int sum = nums[i] + nums[j], left = j + 1, right = n;
9                 while (left < right) {
10                     int mid = left + (right - left) / 2;
11                     if (nums[mid] < sum) left = mid + 1;
12                     else right = mid;
13                 }
14                 res += right - 1 - j;
15             }
16         }
17         return res;
18     }
19 };

```

其实还有更进一步优化的方法，用的是博主之前那篇3Sum Smaller里面的解法二，明明博主以前都总结过，换个题目情景就又没想到，看来博主的举一反三能力还是有所欠缺啊。没办法，只能继续刻意练习了。这种方法能将时间复杂度优化到O(n²)，感觉很叼了。思路是排序之后，从数字末尾开始往前遍历，将left指向首数字，将right之前遍历到的数字的前面一个数字，然后如果left小于right就进行循环，循环里面判断如果left指向的数加上right指向的数大于当前的数字的话，那么right到left之间的数字都可以组成三角形，这是为啥呢，相当于此时确定了i和right的位置，可以将left向右移到right的位置，中间经过的数都大于left指向的数，所以都能组成三角形，就说这思路叼不叼！加完之后，right自减一，即向左移动一位。如果left和right指向的数字之和不大于nums[i]，那么left自增1，即向右移动一位，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int triangleNumber(vector<int>& nums) {
4         int res = 0, n = nums.size();
5         sort(nums.begin(), nums.end());
6         for (int i = n - 1; i >= 2; --i) {
7             int left = 0, right = i - 1;
8             while (left < right) {
9                 if (nums[left] + nums[right] > nums[i]) {
10                     res += right - left;
11                     --right;
12                 } else {
13                     ++left;
14                 }
15             }
16         }
17         return res;
18     }
19 };

```

602. Shortest Distance in a Plane

Table point_2d holds the coordinates (x,y) of some unique points (more than two) in a plane.
Write a query to find the shortest distance between these points rounded to 2 decimals.

x	y
-1	-1
0	0
-1	-2

The shortest distance is 1.00 from point (-1,-1) to (-1,2). So the output should be:

shortest
1.00

```
1 | SELECT
2 | ROUND(SQRT(MIN((POW(p1.x - p2.x, 2) + POW(p1.y - p2.y, 2)))),2) AS shortest
3 | FROM
4 | point_2d p1
5 | JOIN
6 | point_2d p2 ON (p1.x <= p2.x AND p1.y < p2.y)
7 | OR (p1.x <= p2.x AND p1.y > p2.y)
8 | OR (p1.x < p2.x AND p1.y = p2.y)
9 | ;
```

SQL

603. Shortest Distance in a Line

Table point holds the x coordinate of some points on x-axis in a plane, which are all integers.
Write a query to find the shortest distance between two points in these points.

x
-1
0
2

The shortest distance is '1' obviously, which is from point '-1' to '0'. So the output is as below:

shortest
1

```
1 | SELECT
2 | MIN(ABS(p1.x - p2.x)) AS shortest
3 | FROM
4 | point p1
5 | JOIN
6 | point p2 ON p1.x != p2.x
7 | ;
```

SQL

604. Second Degree Follower

```
1 | SELECT f1.follower, COUNT(DISTINCT f2.follower) AS num
2 | FROM follow f1
3 | JOIN follow f2 ON f1.follower = f2.followee
4 | GROUP BY f1.follower
```

SQL

605. Average Salary: Departments VS Company

Given two tables as below, write a query to display the comparison result (higher/lower/same) of the average salary of employees in a department to the company's average salary.

Table: salary

id	employee_id	amount	pay_date
1	1	9000	2017-03-31
2	2	6000	2017-03-31
3	3	10000	2017-03-31
4	1	7000	2017-02-28
5	2	6000	2017-02-28
6	3	8000	2017-02-28

The employee_id column refers to the employee_id in the following table employee.

employee_id	department_id
1	1
2	2
3	2

```

1 | select department_salary.pay_month, department_id,
2 | case
3 | when department_avg>company_avg then 'higher'
4 | when department_avg<company_avg then 'lower'
5 | else 'same'
6 | end as comparison
7 | from
8 |
9 | select department_id, avg(amount) as department_avg, date_format(pay_date, '%Y-%m') as
10 | pay_month
11 | from salary join employee on salary.employee_id = employee.employee_id
12 | group by department_id, pay_month
13 | ) as department_salary
14 | join
15 |
16 | select avg(amount) as company_avg, date_format(pay_date, '%Y-%m') as pay_month from
17 | salary group by date_format(pay_date, '%Y-%m')
18 | ) as company_salary
on department_salary.pay_month = company_salary.pay_month
;
```

SQL

606. 字符串中增加粗标签

Given a string s and a list of strings dict, you need to add a closed pair of bold tag and to wrap the substrings in s that exist in dict. If two such substrings overlap, you need to wrap them together by only one pair of closed bold tag. Also, if two substrings wrapped by bold tags are consecutive, you need to combine them.

这道题给我们了一个字符串，还有一个字典，让我们把字符串中在字典中的单词加粗，注意如果两个单词有交集或者相接，就放到同一个加粗标签中。博主刚开始的想法是，既然需要匹配字符串，那么就上KMP大法，然后得到每个单词在字符串匹配的区间位置，然后再合并区间，再在合并后的区间两头加标签。但是一看题目难度，Medium，中等难度的题不至于要祭出KMP大法吧，于是去网上扫了一眼众神们的解法，发现大多都是暴力匹配啊，既然OJ能过去，那么就一起暴力吧。这题参考的是高神shawngao的解法，高神可是集了一千五百多个赞的男人，叼到飞起！思路是建一个和字符串s等长的bold布尔型数组，表示如果该字符在单词里面就为true，那么最后我们就可以根据bold数组的真假值来添加标签了。我们遍历字符串s中的每一个字符，把遍历到的每一个字符当作起始位置，我们都匹配一遍字典中的所有单词，如果能匹配上，我们就用i + len来更新end，len是当前单词的长度，end表示字典中的单词在字符串s中结束的位置，那么如果i小于end，bold[i]就要赋值为true了。最后我们更新完bold数组了，就再遍历一遍字符串s，如果bold[i]为false，直接将s[i]加入结果res中；如果bold[i]为true，那么我们用while循环来找出所有连续为true的个数，然后在左右两端加上标签，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     string addBoldTag(string s, vector<string>& dict) {
4         string res = "";
5         int n = s.size(), end = 0;
6         vector<bool> bold(n, false);
7         for (int i = 0; i < n; ++i) {
8             for (string word : dict) {
9                 int len = word.size();
10                if (i + len <= n && s.substr(i, len) == word) {
11                    end = max(end, i + len);
12                }
13            }
14            bold[i] = end > i;
15        }
16        for (int i = 0; i < n; ++i) {
17            if (!bold[i]) {
18                res.push_back(s[i]);
19                continue;
20            }
21            int j = i;
22            while (j < n && bold[j]) ++j;
23            res += "<b>" + s.substr(i, j - i) + "</b>";
24            i = j - 1;
25        }
26        return res;
27    }
28 };

```

CPP

这道题跟之后的那道Bold Words in String是一模一样的题，那么解法当然是可以互通的了，这里我们把那道题中解法二也贴过来吧，由于解法一和解法二实在是太相似了，就贴一个吧，具体讲解可以参见Bold Words in String这篇帖子，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     string addBoldTag(string s, vector<string>& dict) {
4         string res = "";
5         int n = s.size();
6         unordered_set<int> bold;
7         for (string word : dict) {
8             int len = word.size();
9             for (int i = 0; i <= n - len; ++i) {
10                 if (s[i] == word[0] && s.substr(i, len) == word) {
11                     for (int j = i; j < i + len; ++j) bold.insert(j);
12                 }
13             }
14         }
15         for (int i = 0; i < n; ++i) {
16             if (bold.count(i) && !bold.count(i - 1)) res += "<b>";
17             res += s[i];
18             if (bold.count(i) && !bold.count(i + 1)) res += "</b>";
19         }
20         return res;
21     }
22 };

```

607. 合并二叉树

Given two binary trees and imagine that when you put one of them to cover the other, some nodes of the two trees are overlapped while the others are not.

You need to merge them into a new binary tree. The merge rule is that if two nodes overlap, then sum node values up as the new value of the merged node. Otherwise, the NOT null node will be used as the node of new tree.

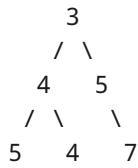
Example 1:

Input:



Output:

Merged tree:



这道题给了我们两个二叉树，让我们合并成一个，规则是，都存在的结点，就将结点值加起来，否则空的位置就由另一个树的结点来代替。那么根据过往经验，处理二叉树问题的神器就是递归，那么我们来看递归函数如何去写。根据题目中的规则，我们知道如果要处理的相同位置上的两个结点都不存在的话，直接返回即可，如果t1存在，t2不存在，那么我们就以t1的结点值建立一个新结点，然后分别对t1的左右子结点和空结点调用递归函数，反之，如果t1不存在，t2存在，那么我们就以t2的结点值建立一个新结点，然后分别对t2的左右子结点和空结点调用递归函数。如果t1和t2都存在，那么我们就以t1和t2的结点值之和建立一个新结点，然后分别对t1的左右子结点和t2的左右子结点调用递归函数，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     TreeNode* mergeTrees(TreeNode* t1, TreeNode* t2) {
4         TreeNode *res = NULL;
5         helper(t1, t2, res);
6         return res;
7     }
8     void helper(TreeNode* t1, TreeNode* t2, TreeNode*& res) {
9         if (!t1 && !t2) return;
10        else if (t1 && !t2) {
11            res = new TreeNode(t1->val);
12            helper(t1->left, NULL, res->left);
13            helper(t1->right, NULL, res->right);
14        } else if (!t1 && t2) {
15            res = new TreeNode(t2->val);
16            helper(NULL, t2->left, res->left);
17            helper(NULL, t2->right, res->right);
18        } else {
19            res = new TreeNode(t1->val + t2->val);
20            helper(t1->left, t2->left, res->left);
21            helper(t1->right, t2->right, res->right);
22        }
23    }
24};

```

其实远不用写的像上面那么复杂，我们连额外的函数都不用写，直接递归调用给定的函数即可，我们首先判断，如果t1不存在，则直接返回t2，反之，如果t2不存在，则直接返回t1。如果上面两种情况都不满足，那么以t1和t2的结点值之和建立新结点t，然后对t1和t2的左子结点调用递归并赋给t的左子结点，再对t1和t2的右子结点调用递归并赋给t的右子结点，返回t结点即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     TreeNode* mergeTrees(TreeNode* t1, TreeNode* t2) {
4         if (!t1) return t2;
5         if (!t2) return t1;
6         TreeNode *t = new TreeNode(t1->val + t2->val);
7         t->left = mergeTrees(t1->left, t2->left);
8         t->right = mergeTrees(t1->right, t2->right);
9         return t;
10    }
11};

```

608. Students Report By Geography

A U.S graduate school has students from Asia, Europe and America. The students' location information are stored in table student as below.

name	continent
Jack	America
Pascal	Europe
Xi	Asia
Jane	America

Pivot the continent column in this table so that each name is sorted alphabetically and displayed underneath its corresponding continent. The output headers should be America, Asia and Europe respectively. It is guaranteed that the student number from America is no less than either Asia or Europe.

For the sample input, the output is:

America	Asia	Europe
Jack	Xi	Pascal
Jane		

```

1 | SELECT
2 | America, Asia, Europe
3 | FROM
4 | (SELECT @as:=0, @am:=0, @eu:=0) t,
5 | (SELECT
6 |   @as:=@as + 1 AS asid, name AS Asia
7 |   FROM
8 |   student
9 |   WHERE
10 |   continent = 'Asia'
11 |   ORDER BY Asia) AS t1
12 | RIGHT JOIN
13 | (SELECT
14 |   @am:=@am + 1 AS amid, name AS America
15 |   FROM
16 |   student
17 |   WHERE
18 |   continent = 'America'
19 |   ORDER BY America) AS t2 ON asid = amid
20 | LEFT JOIN
21 | (SELECT
22 |   @eu:=@eu + 1 AS euid, name AS Europe
23 |   FROM
24 |   student
25 |   WHERE
26 |   continent = 'Europe'
27 |   ORDER BY Europe) AS t3 ON amid = euid
28 |
;
```

SQL

609. Biggest Single Number

Table number contains many numbers in column num including duplicated ones.
Can you write a SQL query to find the biggest number, which only appears once.

```
+---+
|num|
+---+
| 8 |
| 8 |
| 3 |
| 3 |
| 1 |
| 4 |
| 5 |
| 6 |
```

For the sample data above, your query should return the following result:

```
+---+
|num|
+---+
| 6 |
```

```
1 SELECT
2 MAX(num) AS num
3 FROM
4 (SELECT
5   num
6   FROM
7   number
8   GROUP BY num
9   HAVING COUNT(num) = 1) AS t
10 ;
```

SQL

610. Not Boring Movies

X city opened a new cinema, many people would like to go to this cinema. The cinema also gives out a poster indicating the movies' ratings and descriptions.
Please write a SQL query to output movies with an odd numbered ID and a description that is not 'boring'. Order the result by rating.

For example, table cinema:

```
+-----+-----+-----+-----+
| id    | movie     | description | rating   |
+-----+-----+-----+-----+
| 1     | War       | great 3D   | 8.9      |
| 2     | Science   | fiction    | 8.5      |
| 3     | irish     | boring     | 6.2      |
| 4     | Ice song  | Fantasy   | 8.6      |
| 5     | House card| Interesting| 9.1      |
+-----+-----+-----+-----+
```

For the example above, the output should be:

```
+-----+-----+-----+-----+
| id    | movie     | description | rating   |
+-----+-----+-----+-----+
| 5     | House card| Interesting| 9.1      |
| 1     | War       | great 3D   | 8.9      |
+-----+-----+-----+-----+
```

```
1 | select *
2 | from cinema
3 | where mod(id, 2) = 1 and description != 'boring'
4 | order by rating DESC
5 | ;
```

611. 任务行程表

Given a char array representing tasks CPU need to do. It contains capital letters A to Z where different letters represent different tasks. Tasks could be done without original order. Each task could be done in one interval. For each interval, CPU could finish one task or just be idle.

However, there is a non-negative cooling interval n that means between two same tasks, there must be at least n intervals that CPU are doing different tasks or just be idle.

You need to return the least number of intervals the CPU will take to finish all the given tasks.

这道题让我们安排CPU的任务，规定在两个相同任务之间至少隔n个时间点。说实话，刚开始博主并没有完全理解题目的意思，后来看了大神们的解法才悟出个道理来。下面这种解法参考了大神fatalme的帖子，由于题目中规定了两个相同任务之间至少隔n个时间点，那么我们首先应该处理的出现次数最多的那个任务，先确定好这些高频任务，然后再来安排那些低频任务。如果任务F的出现频率最高，为k次，那么我们用n个空位将每两个F分隔开，然后我们按顺序加入其他低频的任务，来看一个例子：

AAAABBEEFFGG 3

我们发现任务A出现了4次，频率最高，于是我们在每个A中间加入三个空位，如下：

A---A---A---A

AB--AB--AB--A (加入B)

ABE-ABE-AB--A (加入E)

ABEFABE-ABF-A (加入F，每次尽可能填满或者是均匀填充)

ABEFABEGABFGA (加入G)

再来看一个例子：

ACCCEE 2

我们发现任务C和E都出现了三次，那么我们就将CE看作一个整体，在中间加入一个位置即可：

CE-CE-CE

CEACE-CE (加入A)

注意最后面那个idle不能省略，不然就不满足相同两个任务之间要隔2个时间点了。

这道题好在没有让我们输出任务安排结果，而只是问所需的时间总长，那么我们就想个方法来快速计算出所需时间总长即可。我们仔细观察上面两个例子可以发现，都分成了 $(mx - 1)$ 块，再加上最后面的字母，其中 mx 为最大出现次数。比如例子1中，A出现了4次，所以有A---模块出现了3次，再加上最后的A，每个模块的长度为4。例子2中，CE-出现了2次，再加上最后的CE，每个模块长度为3。我们可以发现，模块的次数为任务最大次数减1，模块的长度为 $n+1$ ，最后加上的字母个数为出现次数最多的任务，可能有多个并列。这样三个部分都搞清楚了，写起来就不难了，我们统计每个大写字母出现的次数，然后排序，这样出现次数最多的字母就到了末尾，然后我们向前遍历，找出出现次数一样多的任务个数，就可以迅速求出总时间长了，下面这段代码可能最不好理解的可能就是最后一句了，那么我们特别来讲解一下。先看括号中的第二部分，前面分析说了 mx 是出现的最大次数， $mx - 1$ 是可以分为的块数， $n+1$ 是每块中的个数，而后面的 $25 - i$ 是还需要补全的个数，用之前的例子来说明：

AAAABBEEFFGG 3

A出现了4次，最多， $mx=4$ ，那么可以分为 $mx - 1 = 3$ 块，如下：

A---A---A---

每块有 $n+1=4$ 个，最后还要加上末尾的一个A，也就是 $25 - 24 = 1$ 个任务，最终结果为13：

ABEFABEGABFGA

再来看另一个例子：

ACCCEE 2

C和E都出现了3次，最多， $mx=3$ ，那么可以分为 $mx - 1 = 2$ 块，如下：

CE-CE-

每块有 $n+1=3$ 个，最后还要加上末尾的一个CE，也就是 $25 - 23 = 2$ 个任务，最终结果为8：

CEACE-CE

好，那么此时你可能会有疑问，为啥还要跟原任务个数len相比，取较大值呢？我们再来看一个例子：

AAABBB 0

A和B都出现了3次，最多， $mx=3$ ，那么可以分为 $mx-1=2$ 块，如下：

ABAB

每块有 $n+1=1$ 个？你会发现有问题，这里明明每块有两个啊，为啥这里算出来 $n+1=1$ 呢，因为给的 $n=0$ ，这有没有矛盾呢，没有！因为 n 表示相同的任务间需要间隔的个数，那么既然这里为0了，说明相同的任务可以放在一起，这里就没有任何限制了，我们只需要执行完所有的任务就可以了，所以我们最终的返回结果一定不能小于任务的总个数len的，这就是要对比取较大值的原因了。

参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int leastInterval(vector<char>& tasks, int n) {
4         vector<int> cnt(26, 0);
5         for (char task : tasks) {
6             ++cnt[task - 'A'];
7         }
8         sort(cnt.begin(), cnt.end());
9         int i = 25, mx = cnt[25], len = tasks.size();
10        while (i >= 0 && cnt[i] == mx) --i;
11        return max(len, (mx - 1) * (n + 1) + 25 - i);
12    }
13 };
14 
```

CPP

下面这种解法是根据大神jinzhou的帖子，优点是代码更容易读懂，而且变量命名很reasonable，前半部分都是一样的，求出最多的次数mx，还有同时出现mx次的不同任务的个数mxCnt。这个解法的思想是先算出所有空出来的位置，然后计算出所有需要填入的task的个数，如果超出了空位的个数，就需要最后再补上相应的个数。注意这里如果有多个任务出现次数相同，那么将其整体放一起，就像上面的第二个例子中的CE一样，那么此时每个part中的空位个数就是 $n - (mxCnt - 1)$ ，那么空位的总数就是part的总数乘以每个part中空位的个数了，那么我们此时除去已经放入part中的，还剩下的task的个数就是task的总个数减去 $mx * mxCnt$ ，然后此时和之前求出的空位数相比较，如果空位数要大于剩余的task数，那么则说明还需补充多余的空位，否则就直接返回task的总数即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int leastInterval(vector<char>& tasks, int n) {
4         int mx = 0, mxCnt = 0;
5         vector<int> cnt(26, 0);
6         for (char task : tasks) {
7             ++cnt[task - 'A'];
8             if (mx == cnt[task - 'A']) {
9                 ++mxCnt;
10            } else if (mx < cnt[task - 'A']) {
11                mx = cnt[task - 'A'];
12                mxCnt = 1;
13            }
14        }
15        int partCnt = mx + 1;
16        int partLen = n - (mxCnt - 1);
17        int emptySlots = partCnt * partLen;
18        int taskLeft = tasks.size() - mx * mxCnt;
19        int idles = max(0, emptySlots - taskLeft);
20        return tasks.size() + idles;
21    }
22 };

```

下面这种解法是参考的大神alexander的解法，思路是建立一个优先队列，然后把统计好的个数都存入优先队列中，那么大的次数会在队列的前面。这题还是要分块，每块能装 $n+1$ 个任务，装任务是从优先队列中取，每个任务取一个，装到一个临时数组中，然后遍历取出的任务，对于每个任务，将其哈希表映射的次数减1，如果减1后，次数仍大于0，则将此任务次数再次排入队列中，遍历完后如果队列不为空，说明该块全部被填满，则结果加上 $n+1$ 。我们之前在队列中取任务是用个变量cnt来记录取出任务的个数，我们想取出 $n+1$ 个，如果队列中任务数少于 $n+1$ 个，那就用cnt来记录真实取出的个数，当队列为空时，就加上cnt的个数，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int leastInterval(vector<char>& tasks, int n) {
4         int res = 0, cycle = n + 1;
5         unordered_map<char, int> m;
6         priority_queue<int> q;
7         for (char c : tasks) ++m[c];
8         for (auto a : m) q.push(a.second);
9         while (!q.empty()) {
10             int cnt = 0;
11             vector<int> t;
12             for (int i = 0; i < cycle; ++i) {
13                 if (!q.empty()) {
14                     t.push_back(q.top()); q.pop();
15                     ++cnt;
16                 }
17             }
18             for (int d : t) {
19                 if (--d > 0) q.push(d);
20             }
21             res += q.empty() ? cnt : cycle;
22         }
23         return res;
24     }
25 };

```

612. Design circular queue

Design your implementation of the circular queue. The circular queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called "Ring Buffer".

One of the benefits of the circular queue is that we can make use of the spaces in front of the queue. In a normal queue, once the queue becomes full, we cannot insert the next element even if there is a space in front of the queue. But using the circular queue, we can use the space to store new values.

Your implementation should support following operations:

`MyCircularQueue(k)`: Constructor, set the size of the queue to be k.

`Front`: Get the front item from the queue. If the queue is empty, return -1.

`Rear`: Get the last item from the queue. If the queue is empty, return -1.

`enQueue(value)`: Insert an element into the circular queue. Return true if the operation is successful.

`deQueue()`: Delete an element from the circular queue. Return true if the operation is successful.

`isEmpty()`: Checks whether the circular queue is empty or not.

`isFull()`: Checks whether the circular queue is full or not.

```
1 class MyCircularQueue {
2     private:
3         class Node {
4             public:
5                 int val;
6                 Node* next;
7                 Node() {}
8             };
9             Node *front, *rear;
10
11 public:
12     /** Initialize your data structure here. Set the size of the queue to be k. */
13     MyCircularQueue(int k) {
14         Node* root = new Node();
15         Node* cur = root;
16         front = root;
17         rear = nullptr;
18         for(int i = 0; i < k - 1; i++)
19         {
20             Node* temp = new Node();
21             cur -> next = temp;
22             cur = temp;
23         }
24         cur -> next = root;
25     }
26
27     /** Insert an element into the circular queue. Return true if the operation is
28     successful. */
29     bool enqueue(int value) {
30         if (isFull()) return false;
31         if (isEmpty())
32             rear = front;
33         else
34             rear = rear -> next;
35         rear -> val = value;
36         return true;
37     }
38
39     /** Delete an element from the circular queue. Return true if the operation is
40     successful. */
41     bool dequeue() {
42         if (isEmpty()) return false;
43         if (front == rear)
44             rear = nullptr;
45         else
46             front = front -> next;
47         return true;
48     }
49
50     /** Get the front item from the queue. */
51     int Front() {
52         return isEmpty() ? -1 : front -> val;
53     }
54
55     /** Get the last item from the queue. */
56     int Rear() {
57         return isEmpty() ? -1 : rear -> val;
58     }
59 }
```

```

60  /** Checks whether the circular queue is empty or not. */
61  bool isEmpty() {
62      return rear == nullptr;
63  }
64
65  /** Checks whether the circular queue is full or not. */
66  bool isFull() {
67      if(isEmpty()) return false;
68      return rear -> next == front;
69  }
}:
```

613. 二叉树中增加一行

Given the root of a binary tree, then value v and depth d, you need to add a row of nodes with value v at the given depth d. The root node is at depth 1.

The adding rule is: given a positive integer depth d, for each NOT null tree nodes N in depth d-1, create two tree nodes with value v as N's left subtree root and right subtree root. And N's original left subtree should be the left subtree of the new left subtree root, its original right subtree should be the right subtree of the new right subtree root. If depth d is 1 that means there is no depth d-1 at all, then create a tree node with value v as the new root of the whole original tree, and the original tree is the new root's left subtree.

这道题让我们给二叉树增加一行，给了我们需要增加的值，还有需要增加的位置深度，题目中给的例子也比较能清晰的说明问题。但是漏了一种情况，那就是当d=1时，这该怎么加？这时候就需要替换根结点了。其他情况的处理方法都一样，这里博主第一映像觉得应该用层序遍历来做，没遍历完一层，d自减1，我们探测，当d==1时，那么我们需要对于当前层的每一个结点，首先用临时变量保存其原有的左右子结点，然后新建值为v的左右子结点，将原有的左子结点连到新建的左子结点的左子结点上，将原有的右子结点连到新建的右子结点的右子结点，是不是很绕-.-|||。如果d不为1，那么就是层序遍历原有的排入队列操作，记得当检测到d为0时，直接返回，因为添加操作已经完成，没有必要遍历完剩下的结点，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     TreeNode* addOneRow(TreeNode* root, int v, int d) {
4         if (!root) return NULL;
5         if (d == 1) {
6             TreeNode *newRoot = new TreeNode(v);
7             newRoot->left = root;
8             return newRoot;
9         }
10        queue<TreeNode*> q{{root}};
11        while (!q.empty()) {
12            if (--d == 0) return root;
13            int n = q.size();
14            for (int i = 0; i < n; ++i) {
15                auto t = q.front(); q.pop();
16                if (d == 1) {
17                    TreeNode *left = t->left;
18                    TreeNode *right = t->right;
19                    t->left = new TreeNode(v);
20                    t->right = new TreeNode(v);
21                    t->left->left = left;
22                    t->right->right = right;
23                } else {
24                    if (t->left) q.push(t->left);
25                    if (t->right) q.push(t->right);
26                }
27            }
28        }
29        return root;
30    }
31 };

```

虽然博主一贯的理念是二叉树问题肯定首选递归来解，但是这道题博主刚开始以为递归没法解，结果看了大神们的帖子，才发现自己还是图样图森破，难道二叉树的问题皆可递归？反正这道题是可以的，而且写法so简洁，乍一看上去，会有疑问，题目中明明d的范围是从1开始的，为何要考虑d为0的情况，后来读懂了整个解法后，才为原作者的聪慧叹服。这里d的0和1，其实相当于一种flag，如果d为1的话，那么将root连到新建的结点的左子结点上；反之如果d为0，那么将root连到新建的结点的右子结点上，然后返回新建的结点。如果root存在且d大于1的话，那么对root的左子结点调用递归函数，注意此时若d的值正好为2，那么我们就不能直接减1，而是根据左右子结点的情况分别赋值1和0，这样才能起到flag的作用嘛，叼的飞起，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     TreeNode* addOneRow(TreeNode* root, int v, int d) {
4         if (d == 0 || d == 1) {
5             TreeNode *newRoot = new TreeNode(v);
6             (d ? newRoot->left : newRoot->right) = root;
7             return newRoot;
8         }
9         if (root && d > 1) {
10             root->left = addOneRow(root->left, v, d > 2 ? d - 1 : 1);
11             root->right = addOneRow(root->right, v, d > 2 ? d - 1 : 0);
12         }
13         return root;
14     }
15 };

```

614. 数组中的最大距离

Given m arrays, and each array is sorted in ascending order. Now you can pick up two integers from two different arrays (each array picks one) and calculate the distance. We define the distance between two integers a and b to be their absolute difference $|a-b|$. Your task is to find the maximum distance.

这道题给我们了一些数组，每个数组都是有序的，让我们从不同的数组中各取一个数字，使得这两个数字的差的绝对值最大，让我们求这个最大值。那么我们想，既然数组都是有序的，那么差的绝对值最大的两个数字肯定是分别位于数组的首和尾，注意题目中说要从不同的数组中取数，那么即使某个数组的首尾差距很大，也不行。博主最先考虑的是用堆来做，一个最大堆，一个最小堆，最大堆存每个数组的尾元素，最小堆存每个数组的首元素，由于最大的数字和最小的数字有可能来自于同一个数组，所以我们在堆中存数字的时候还要存入当前数字所在的数组的序号，最后我们其实要分别在最大堆和最小堆中各取两个数字，如果最大的数字和最小的数字不在一个数组，那么直接返回二者的绝对差即可，如果在的话，那么要返回第二大数字和最小数字绝对差跟最大数字和第二小数字绝对差中的较大值，参见代码如下：

解法1:

```
1 class Solution {
2 public:
3     int maxDistance(vector<vector<int>>& arrays) {
4         priority_queue<pair<int, int>, mn;
5         for (int i = 0; i < arrays.size(); ++i) {
6             mn.push({-arrays[i][0], i});
7             mx.push({arrays[i].back(), i});
8         }
9         auto a1 = mx.top(); mx.pop();
10        auto b1 = mn.top(); mn.pop();
11        if (a1.second != b1.second) return a1.first + b1.first;
12        return max(a1.first + mn.top().first, mx.top().first + b1.first);
13    }
14};
```

CPP

下面这种方法还是很不错的，并没有用到堆，而是用两个变量start和end分别表示当前遍历过的数组中最小的首元素，和最大的尾元素，那么每当我们遍历到一个新的数组时，只需计算新数组尾元素和start绝对差，跟end和新数组首元素的绝对差，取二者之间的较大值来更新结果res即可，参见代码如下：

解法2:

```
1 class Solution {
2 public:
3     int maxDistance(vector<vector<int>>& arrays) {
4         int res = 0, start = arrays[0][0], end = arrays[0].back();
5         for (int i = 1; i < arrays.size(); ++i) {
6             res = max(res, max(abs(arrays[i].back() - start), abs(end - arrays[i][0])));
7             start = min(start, arrays[i][0]);
8             end = max(end, arrays[i].back());
9         }
10        return res;
11    }
12};
```

CPP

615. 最小因数分解

Given a positive integer a, find the smallest positive integer b whose multiplication of each digit equals to a.

If there is no answer or the answer is not fit in 32-bit signed integer, then return 0.

这道题给了我们一个数字，让我们进行因数分解，让我们找出因数组成的最小的数字。从题目中的例子可以看出，分解出的因数一定是个位数字，即范围是[2, 9]。那我们就可以从大到小开始找因数，首先查找9是否是因数，是要能整除a，就是其因数，如果是的话，就加入到结果res的开头，a自除以9，我们用while循环查找9，直到取出所有的9，然后取8, 7, 6...以此类推，如果a能成功的被分解的话，最后a的值应该为1，如果a值大于1，说明无法被分解，返回true。最后还要看我们结果res字符转为整型是否越界，越界的话还是返回0，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     int smallestFactorization(int a) {
4         if (a == 1) return 1;
5         string res = "";
6         for (int k = 9; k >= 2; --k) {
7             while (a % k == 0) {
8                 res = to_string(k) + res;
9                 a /= k;
10            }
11        }
12        if (a > 1) return 0;
13        long long num = stoll(res);
14        return num > INT_MAX ? 0 : num;
15    }
16};
```

CPP

下面这种方法跟上面解法思路很像，只是结果res没有用字符串，而是直接用的长整型，我们每次在更新完res的结果后，判断一次是否越整型的界，越了就直接返回0，其他部分和上面没有什么区别，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     int smallestFactorization(int a) {
4         if (a < 10) return a;
5         long long res = 0, cnt = 1;
6         for (int i = 9; i >= 2; --i) {
7             while (a % i == 0) {
8                 res += cnt * i;
9                 if (res > INT_MAX) return 0;
10                a /= i;
11                cnt *= 10;
12            }
13        }
14        return (a == 1) ? res : 0;
15    }
16};
```

CPP

616. 三个数字的最大乘积

Given an integer array, find three numbers whose product is maximum and output the maximum product.

这道题博主刚开始看的时候，心想直接排序，然后最后三个数字相乘不就完了，心想不会这么Easy吧，果然被OJ无情打脸，没有考虑到负数和0的情况。这道题给了数组的范围，至少三个，那么如果是三个的话，就无所谓了，直接相乘返回即可，但是如果超过了3个，而且有负数存在的话，情况就可能不一样，我们来考虑几种情况，如果全是负数，三个负数相乘还是负数，为了让负数最大，那么其绝对值就该最小，而负数排序后绝对值小的都在末尾，所以是末尾三个数字相乘，这个跟全是正数的情况一样。那么重点在于前半段是负数，后半段是正数，那么最好的情况肯定是两个最小的负数相乘得到一个正数，然后跟一个最大的正数相乘，这样得到的肯定是最大的数，所以我们让前两个数相乘，再和数组的最后一个数字相乘，就可以得到这种情况下的最大的乘积。实际上我们并不用分情况讨论数组的正负，只要把这两种情况的乘积都算出来，比较二者取较大值，就能涵盖所有的情况，从而得到正确的结果，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     int maximumProduct(vector<int>& nums) {
4         int n = nums.size();
5         sort(nums.begin(), nums.end());
6         int p = nums[0] * nums[1] * nums[n - 1];
7         return max(p, nums[n - 1] * nums[n - 2] * nums[n - 3]);
8     }
9 }
```

CPP

下面这种方法由网友hello_world00提供，找出3个最大的数 || 找出一个最大的和两个最小的，相乘对比也能得到结果，而且是O(n)的时间复杂度，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     int maximumProduct(vector<int>& nums) {
4         int mx1 = INT_MIN, mx2 = INT_MIN, mx3 = INT_MIN;
5         int mn1 = INT_MAX, mn2 = INT_MAX;
6         for (int num : nums) {
7             if (num > mx1) {
8                 mx3 = mx2; mx2 = mx1; mx1 = num;
9             } else if (num > mx2) {
10                 mx3 = mx2; mx2 = num;
11             } else if (num > mx3) {
12                 mx3 = num;
13             }
14             if (num < mn1) {
15                 mn2 = mn1; mn1 = num;
16             } else if (num < mn2) {
17                 mn2 = num;
18             }
19         }
20         return max(mx1 * mx2 * mx3, mx1 * mn1 * mn2);
21     }
22 }
```

CPP

617. K个翻转对数组

Given two integers n and k, find how many different arrays consist of numbers from 1 to n such that there are exactly k inverse pairs.

We define an inverse pair as following: For ith and jth element in the array, if $i < j$ and $a[i] > a[j]$ then it's an inverse pair; Otherwise, it's not.

Since the answer may very large, the answer should be modulo 10⁹ + 7.

这道题给了我们1到n总共n个数字，让我们任意排列数组的顺序，使其刚好存在k个翻转对，所谓的翻转对，就是位置在前面的数字值大，而且题目中表明了结果会很大很大，要我们对一个很大的数字取余。对于这种结果巨大的题目，劝君放弃暴力破解或者是无脑递归，想都不用想，那么最先应该考虑的就是DP的解法了。我们需要一个二维的DP数组，其中 $dp[i][j]$ 表示1到i的数字中有j个翻转对的排列总数，那么我们要求的就是 $dp[n][k]$ 了，即1到n的数字中有k个翻转对的排列总数。现在难点就是要求递推公式了。我们想如果我们已经知道 $dp[n][k]$ 了，怎么求 $dp[n+1][k]$ ，先来看 $dp[n+1][k]$ 的含义，是1到n+1点数字中有k个翻转对的个数，那么实际上在1到n的数字中的某个位置加上了n+1这个数，为了简单起见，我们先让n=4，那么实际上相当于要在某个位置加上5，那么加5的位置就有如下几种情况：

xxxx5

xxx5x

xx5xx

x5xxx

5xxxx

这里xxxx表示1到4的任意排列，那么第一种情况xxxx5不会增加任何新的翻转对，因为xxxx中没有比5大的数字，而xxx5x会新增加1个翻转对，xx5xx，x5xxx，5xxxx分别会增加2，3，4个翻转对。那么xxxx5就相当于 $dp[n][k]$ ，即 $dp[4][k]$ ，那么依次往前类推，就是 $dp[n][k-1]$ ， $dp[n][k-2] \dots dp[n][k-n]$ ，这样我们就可以得出 $dp[n+1][k]$ 的求法了：

$$dp[n+1][k] = dp[n][k] + dp[n][k-1] + \dots + dp[n][k-n]$$

那么 $dp[n][k]$ 的求法也就一目了然了：

$$dp[n][k] = dp[n-1][k] + dp[n-1][k-1] + \dots + dp[n-1][k-n+1]$$

那么我们就可以写出代码如下了：

解法1：

```

1 class Solution {
2 public:
3     int kInversePairs(int n, int k) {
4         int M = 1000000007;
5         vector<vector<int>> dp(n + 1, vector<int>(k + 1, 0));
6         dp[0][0] = 1;
7         for (int i = 0; i <= n; ++i) {
8             for (int j = 0; j < i; ++j) {
9                 for (int m = 0; m <= k; ++m) {
10                     if (m - j >= 0 && m - j <= k) {
11                         dp[i][m] = (dp[i][m] + dp[i - 1][m - j]) % M;
12                     }
13                 }
14             }
15         }
16         return dp[n][k];
17     }
18 };

```

我们可以对上面的解法进行时间上的优化，还是来看我们的递推公式：

$$dp[n][k] = dp[n - 1][k] + dp[n - 1][k-1] + \dots + dp[n - 1][k - n + 1]$$

我们可以用k+1代替k，得到：

$$dp[n][k+1] = dp[n - 1][k+1] + dp[n - 1][k] + \dots + dp[n - 1][k + 1 - n + 1]$$

用第二个等式减去第一个等式可以得到：

$$dp[n][k+1] = dp[n][k] + dp[n - 1][k+1] - dp[n - 1][k - n + 1]$$

将k+1换回成k，可以得到：

$$dp[n][k] = dp[n][k-1] + dp[n - 1][k] - dp[n - 1][k - n]$$

我们可以发现当k>=n的时候，最后一项的数组坐标才能为非负数，从而最后一项才有值，所以我们再更新的时候只需要判断一下k和n的关系，如果k>=n的话，就要减去最后一项，这种递推式算起来更高效，减少了一个循环，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int kInversePairs(int n, int k) {
4         int M = 1000000007;
5         vector<vector<int>> dp(n + 1, vector<int>(k + 1, 0));
6         dp[0][0] = 1;
7         for (int i = 1; i <= n; ++i) {
8             dp[i][0] = 1;
9             for (int j = 1; j <= k; ++j) {
10                 dp[i][j] = (dp[i - 1][j] + dp[i][j - 1]) % M;
11                 if (j >= i) {
12                     dp[i][j] = (dp[i][j] - dp[i - 1][j - i] + M) % M;
13                 }
14             }
15         }
16         return dp[n][k];
17     }
18 };

```

618. 课程清单之三

There are n different online courses numbered from 1 to n . Each course has some duration(course length) and closed on d th day. A course should be taken continuously for t days and must be finished before or on the d th day. You will start at the 1st day.

Given n online courses represented by pairs (t, d) , your task is to find the maximal number of courses that can be taken.

这道题给了我们许多课程，每个课程有两个参数，第一个是课程的持续时间，第二个是课程的最晚结束日期，让我们求最多能上多少门课。博主尝试了递归的暴力破解，TLE了。这道题给的提示是用贪婪算法，那么我们首先给课程排个序，按照结束时间的顺序来排序，我们维护一个当前的时间，初始化为0，再建立一个优先数组，然后我们遍历每个课程，对于每一个遍历到的课程，当前时间加上该课程的持续时间，然后将该持续时间放入优先数组中，然后我们判断如果当前时间大于课程的结束时间，说明这门课程无法被完成，我们并不是直接减去当前课程的持续时间，而是取出优先数组的顶元素，即用时最长的一门课，这也 make sense，因为我们的目标是尽可能的多上课，既然非要去掉一门课，那肯定是去掉耗时最长的课，这样省下来的时间说不定能多上几门课呢，最后返回优先队列中元素的个数就是能完成的课程总数啦，参见代码如下：

```

1 class Solution {
2 public:
3     int scheduleCourse(vector<vector<int>>& courses) {
4         int curTime = 0;
5         priority_queue<int> q;
6         sort(courses.begin(), courses.end(), [] (vector<int>& a, vector<int>& b) {return
7             a[1] < b[1];});
8         for (auto course : courses) {
9             curTime += course[0];
10            q.push(course[0]);
11            if (curTime > course[1]) {
12                curTime -= q.top(); q.pop();
13            }
14        }
15        return q.size();
16    }
17 };

```

619. 设计Excel表格求和公式

Your task is to design the basic function of Excel and implement the function of sum formula. Specifically, you need to implement the following functions:

`Excel(int H, char W):` This is the constructor. The inputs represents the height and width of the Excel form. His a positive integer, range from 1 to 26. It represents the height. W is a character range from 'A' to 'Z'. It represents that the width is the number of characters from 'A' to W. The Excel form content is represented by a height * width 2D integer array C, it should be initialized to zero. You should assume that the first row of C starts from 1, and the first column of C starts from 'A'.

`void Set(int row, char column, int val):` Change the value at C(row, column) to be val.

`int Get(int row, char column):` Return the value at C(row, column).

`int Sum(int row, char column, List of Strings : numbers):` This function calculate and set the value at C(row, column), where the value should be the sum of cells represented by numbers. This function return the sum result at C(row, column). This sum formula should exist until this cell is overlapped by another value or another sum formula.

numbers is a list of strings that each string represent a cell or a range of cells. If the string represent a single cell, then it has the following format : ColRow. For example, "F7" represents the cell at (7, F).

If the string represent a range of cells, then it has the following format : ColRow1:ColRow2. The range will always be a rectangle, and ColRow1 represent the position of the top-left cell, and ColRow2 represents the position of the bottom-right cell.

这道题让我们设计Excel表格的求和公式，Excel表格想必大家都用过，还是比较熟悉的，这里让我们对单元格进行求和运算。由于这道题里要求二维数组的局部和，而且又会经常更新数组的值，博主第一反应觉得应该用之前那题Range Sum Query 2D - Mutable中的树状数组来做，结果哼哼哧哧的写完后，发现下面这个test case没通过：

```
["Excel", "sum", "set", "get"]
[[3, "C"], [1, "A", ["A2"]], [2, "A", 1], [1, "A"]]
Expected:
[null, 0, null, 1]
```

仔细分析一下发现，这个case先把A2的值赋给了A1，此时A1和A2都是0，然后给A2赋值为1，求A1的值。大家的第一印象肯定是觉得A1还是0啊，其实在Excel中，相当于已经把A1和A2关联起来了，只要A2点值发生了改变，A1的值也会跟着变，所以A1的值此时也为1。而树状数组的主要功能的优化区域和的计算速度，并没有建立关联的步骤，难怪不能通过OJ呢。这道题标记为Hard还是有道理的，我们要模拟出Excel表中的这种关联方式，这里参考的是yupinglu大神的帖子，首先我们肯定需要一个二维数组mat来保存数据，然后需要一个map来建立单元格和区域和之间的映射，这里的区域和就是sum函数中的字符串数组表示的内容，可参见题目中的例子，有可能单个单元格或者多个。

我们来看set函数，如果我们改变了某个单元格的内容，那么如果作为结果单元格，那么对应的链接就会断开。比如我们有三个单元格A1, B1, C1，我们设置的关联是A1 + B1 = C1，那么我们改变A1和B1的值都是OK的，C1的值会自动更新。但如果我们改变了C1的值，那么这个关联就不复存在了，Excel中也是这样的。所以我们在改变某个单元格的时候，要将其的关联删除。

我们再来看get函数，我们在获取某个单元格的值的时候，一定要先看其有没有和其他单元格关联，如果说有的话，要重新计算一下关联，有可能关联的单元格的值已经发生改变了，那么当前作为结果单元格的值也需要改变；如果该单元格没有任何关联，那么就直接从数组mat中取值即可。

最后看本题的难点sum函数，要根据关联格求出结果格的值，首先这个字符串数组可能有多个字符串，每个字符串有两个可能，一种是单个的单元格，一种是两个单元格中间用冒号隔开。那么我们需要分情况讨论，区别这两种情况的方法就是看冒号是否存在，如果不存在，就说明只有一个单元格，我们将其数字和字母都提取出来，调用get函数，将该位置的值加入结果res中；如果冒号存在，我们根据冒号的位置，分别将两个单元格的字母和数字提取出来，然后遍历这两个单元格之间所有的单元格，调用get函数并将返回值加入结果res中。这个遍历相加的过程可能可以用树状数组来优化，但由于这不是此题的考察重点，所以直接遍历就OK。最后别忘了建立目标单元格和区域字符串数组之间的映射，并返回结果res即可。

```

1 class Excel {
2 public:
3     Excel(int H, char W) {
4         m.clear();
5         mat.resize(H, vector<int>(W - 'A', 0));
6     }
7
8     void set(int r, char c, int v) {
9         if (m.count({r, c})) m.erase({r, c});
10        mat[r - 1][c - 'A'] = v;
11    }
12
13    int get(int r, char c) {
14        if (m.count({r, c})) return sum(r, c, m[{r, c}]);
15        return mat[r - 1][c - 'A'];
16    }
17
18    int sum(int r, char c, vector<string> strs) {
19        int res = 0;
20        for (string str : strs) {
21            auto found = str.find_last_of(":");
22            if (found == string::npos) {
23                char y = str[0];
24                int x = stoi(str.substr(1));
25                res += get(x, y);
26            } else {
27                int x1 = stoi(str.substr(1, (int)found - 1)), y1 = str[0] - 'A';
28                int x2 = stoi(str.substr(found + 2)), y2 = str[found + 1] - 'A';
29                for (int i = x1; i <= x2; ++i) {
30                    for (int j = y1; j <= y2; ++j) {
31                        res += get(i, j + 'A');
32                    }
33                }
34            }
35        }
36        m[{r, c}] = strs;
37        return res;
38    }
39
40 private:
41     vector<vector<int>> mat;
42     map<pair<int, char>, vector<string>> m;
43 };

```

620. 最小的范围

You have k lists of sorted integers in ascending order. Find the smallest range that includes at least one number from each of the k lists.

We define the range $[a, b]$ is smaller than range $[c, d]$ if $b-a < d-c$ or $a < c$ if $b-a == d-c$.

这道题给了我们一些数组，都是排好序的，让我们求一个最小的范围，使得这个范围内至少会包括每个数组中的一个数字。虽然每个数组都是有序的，但是考虑到他们之间的数字差距可能很大，所以我们最好还是合并成一个数组统一处理比较好，但是合并成一个大数组还需要保留其原属数组的序号，所以我们大数组中存pair对，同时保存数字和原数组的序号。然后我们重新按照数字大小进行排序，这样我们的问题实际上就转换成了求一个最小窗口，使其能够同时包括所有数组中的至少一个数字。这不就变成了那道Minimum Window Substring。所以说啊，这些题目都是换汤不换药的，总能变成我们见过的类型。我们用两个指针left和right来确定滑动窗口的范围，我们还要用一个哈希表来建立每个数组与其数组中数字出现的个数之间的映射，变量cnt表

示当前窗口中的数字覆盖了几个数组，diff为窗口的大小，我们让right向右滑动，然后判断如果right指向的数字所在数组没有被覆盖到，cnt自增1，然后哈希表中对应的数组出现次数自增1，然后我们循环判断如果cnt此时为k(数组的个数)且left不大于right，那么我们用当前窗口的范围来更新结果，然后此时我们想缩小窗口，通过将left向右移，移动之前需要减小哈希表中的映射值，因为我们去除了数字，如果此时映射值为0了，说明我们有个数组无法覆盖到了，cnt就要自减1。这样遍历后我们就能得到最小的范围了，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<int> smallestRange(vector<vector<int>>& nums) {
4         vector<int> res;
5         vector<pair<int, int>> v;
6         unordered_map<int, int> m;
7         for (int i = 0; i < nums.size(); ++i) {
8             for (int num : nums[i]) {
9                 v.push_back({num, i});
10            }
11        }
12        sort(v.begin(), v.end());
13        int left = 0, n = v.size(), k = nums.size(), cnt = 0, diff = INT_MAX;
14        for (int right = 0; right < n; ++right) {
15            if (m[v[right].second] == 0) ++cnt;
16            ++m[v[right].second];
17            while (cnt == k && left <= right) {
18                if (diff > v[right].first - v[left].first) {
19                    diff = v[right].first - v[left].first;
20                    res = {v[left].first, v[right].first};
21                }
22                if (--m[v[left].second] == 0) --cnt;
23                ++left;
24            }
25        }
26        return res;
27    }
28 };

```

CPP

这道题还有一种使用priority_queue来做的，优先队列默认情况是最大堆，但是这道题我们需要使用最小堆，我们可以重新写一下comparator就行了。解题的主要思路很上面的解法很相似，只是具体的数据结构的使用上略有不同，这curMax表示当前遇到的最大数字，用一个idx数组表示每个list中遍历到的位置，然后就是我们的优先队列了，里面放一个pair，是数字和其所属list组成的对儿。然后我们遍历所有的list，将每个list的首元素和该list序号组成pair放入队列中，然后idx数组中每个位置都赋值为1，因为0的位置已经放入队列了，所以指针向后移一个位置，还要更新当前最大值curMax。此时我们的queue中是每个list各有一个数字，由于是最小堆，所以最小的数字就在队首，再加上最大值curMax，就可以初始化结果res了。然后我们进行循环，注意这里循环的条件不是队列不为空，而是当某个list的数字遍历完了就结束循环，因为我们的范围要cover每个list至少一个数字。所以我们的while循环条件即是队首数字所在的list的遍历位置小于该list的总个数，在循环中，取出队首数字所在的list序号t，然后将该list中下一个位置的数字和该list序号t组成pair，加入队列中，然后用这个数字更新curMax，同时idx中t对应的位置也自增1。现在来更新结果res，如果结果res中两数之差大于curMax和队首数字之差，则我们更新结果res，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<int> smallestRange(vector<vector<int>>& nums) {
4         int curMax = INT_MIN, n = nums.size();
5         vector<int> idx(n, 0);
6         auto cmp = [] (pair<int, int>& a, pair<int, int>& b) {return a.first > b.first;};
7         priority_queue<pair<int, int>, vector<pair<int, int>>, decltype(cmp) > q(cmp);
8         for (int i = 0; i < n; ++i) {
9             q.push({nums[i][0], i});
10            idx[i] = 1;
11            curMax = max(curMax, nums[i][0]);
12        }
13        vector<int> res{q.top().first, curMax};
14        while (idx[q.top().second] < nums[q.top().second].size()) {
15            int t = q.top().second; q.pop();
16            q.push({nums[t][idx[t]], t});
17            curMax = max(curMax, nums[t][idx[t]]);
18            ++idx[t];
19            if (res[1] - res[0] > curMax - q.top().first) {
20                res = {q.top().first, curMax};
21            }
22        }
23        return res;
24    }
25 };

```

621. 平方数之和

Given a non-negative integer c, your task is to decide whether there're two integers a and b such that $a^2 + b^2 = c$.

这道题让我们求一个数是否能由平方数之和组成，刚开始博主没仔细看题，没有看到必须要是两个平方数之和，博主以为任意一个就可以。所以写了个带优化的递归解法，楼主已经不是上来就无脑暴力破解的辣个青葱骚年了，直接带优化。可是居然对14返回false，难道14不等于 $1+4+9$ 吗，结果仔细一看，必须要两个平方数之和。好吧，那么递归都省了，直接判断两次就行了。我们可以从c的平方根，注意即使c不是平方数，也会返回一个整型数。然后我们判断如果 i^2 等于c，说明c就是个平方数，只要再凑个0，就是两个平方数之和，返回true；如果不等于的话，那么算出差值 $c - i^2$ ，如果这个差值也是平方数的话，返回true。遍历结束后返回false，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool judgeSquareSum(int c) {
4         for (int i = sqrt(c); i >= 0; --i) {
5             if (i * i == c) return true;
6             int d = c - i * i, t = sqrt(d);
7             if (t * t == d) return true;
8         }
9         return false;
10    }
11 };

```

下面这种方法用到了集合set，从0遍历到c的平方根，对于每个 i^2 ，都加入集合set中，然后计算 $c - i^2$ ，如果这个差值也在集合set中，返回true，遍历结束返回false，参见代码如下：

解法2:

```

1 class Solution {
2 public:
3     bool judgeSquareSum(int c) {
4         unordered_set<int> s;
5         for (int i = 0; i <= sqrt(c); ++i) {
6             s.insert(i * i);
7             if (s.count(c - i * i)) return true;
8         }
9         return false;
10    }
11 };

```

CPP

上面两种方法都不是很高效率，来看下面这种高效的解法。论坛上有人称之为二分解法，但是博主怎么觉得不是呢，虽然样子很像，但是并没有折半的操作啊。这里用a和b代表了左右两个范围，分别为0和c的平方根，然后while循环遍历，如果 $a^2 + b^2$ 刚好等于c，那么返回true；如果小于c，则a增大1；反之如果大于c，则b自减1，参见代码如下：

解法3:

```

1 class Solution {
2 public:
3     bool judgeSquareSum(int c) {
4         int a = 0, b = sqrt(c);
5         while (a <= b) {
6             if (a * a + b * b == c) return true;
7             else if (a * a + b * b < c) ++a;
8             else --b;
9         }
10        return false;
11    }
12 };

```

CPP

622. 找数组的错排

In combinatorial mathematics, a derangement is a permutation of the elements of a set, such that no element appears in its original position.

There's originally an array consisting of n integers from 1 to n in ascending order, you need to find the number of derangement it can generate.

Also, since the answer may be very large, you should return the output mod 109 + 7.

这道题给了我们一个数组，让我们求其错排的个数，所谓错排就是1到n中的每个数字都不在其原有的位置上，全部打乱了，问能有多少种错排的方式。博主注意到了这道题又让对一个很大的数取余，而且每次那个很大的数都是 $10^9 + 7$ ，为啥大家都偏爱这个数呢，有啥特别之处吗？根据博主之前的经验，这种结果很大很大的题十有八九都是用dp来做的，那么就建一个一维的dp数组吧，其中 $dp[i]$ 表示1到i中的错位排列的个数。那么难点就是找递推公式啦，先从最简单的情况来看：

$n = 1$ 时有 0 种错排

$n = 2$ 时有 1 种错排 [2, 1]

$n = 3$ 时有 2 种错排 [3, 1, 2], [2, 3, 1]

然后博主就在想知道了 $dp[2]$ ，能求出 $dp[3]$ 吗，又在考虑是不是算加入数字3的情况的个数。结果左看右看发现没有啥特别的规律，又在想是不是有啥隐含的信息需要挖掘，还是没想出来。于是看了一眼标签，发现是Math，我的天，难道又是小学奥数的题？挣扎了半天最后还是放弃了，上网去搜大神们的解法。其实这道题是组合数学种的错排问题，是有专门的递归公式的。

我们来想 $n = 4$ 时该怎么求，我们假设把4排在了第k位，这里我们就让 $k = 3$ 吧，那么我们就把4放到了3的位置，变成了：

x x 4 x

我们看被4占了位置的3，应该放到哪里，这里分两种情况，如果3放到了4的位置，那么有：

x x 4 3

那么此时4和3的位置都确定了，实际上只用排1和2了，那么就相当于只排1和2，就是 $dp[2]$ 的值，是已知的。那么再来看第二种情况，3不在4的位置，那么此时我们把4去掉的话，就又变成了：

x x x

这里3不能放在第3个x的位置，在去掉4之前，这里是移动4之前的4的位置，那么实际上这又变成了排1, 2, 3的情况了，就是 $dp[3]$ 的值。

再回到最开始我们选k的时候，我们当时选了 $k = 3$ ，其实 k 可以等于1, 2, 3，也就是有三种情况，所以 $dp[4] = 3 * (dp[3] + dp[2])$ 。

那么递推公式也就出来了：

$dp[i] = (i - 1) * (dp[i - 1] + dp[i - 2])$

有了递推公式，代码就不难写了吧，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     int findDerangement(int n) {
4         if (n < 2) return 0;
5         vector<long long> dp(n + 1, 0);
6         dp[1] = 0; dp[2] = 1;
7         for (int i = 3; i <= n; ++i) {
8             dp[i] = (dp[i - 1] + dp[i - 2]) * (i - 1) % 1000000007;
9         }
10        return dp[n];
11    }
12};
```

CPP

下面这种解法精简了空间，因为当前值只跟前两个值有关系，所以没必要保留整个数组，只用两个变量来记录前两个值，并每次更新一下就好了，参见代码如下：

解法2:

```

1 class Solution {
2 public:
3     int findDerangement(int n) {
4         long long a = 0, b = 1, res = 1;
5         for (int i = 3; i <= n; ++i) {
6             res = (i - 1) * (a + b) % 1000000007;
7             a = b;
8             b = res;
9         }
10        return (n == 1) ? 0 : res;
11    }
12 };

```

CPP

下面这种方法是对之前的递推公式进行了推导变形，使其只跟前一个数有关，具体的推导步骤是这样的：

我们假设 $e[i] = dp[i] - i * dp[i - 1]$

递推公式为： $dp[i] = (i - 1) * (dp[i - 1] + dp[i - 2])$

将递推公式带入假设，得到：

$e[i] = -dp[i - 1] + (n - 1) * dp[i - 2] = -e[i - 1]$

从而得到 $e[i] = (-1)^n$

那么带回假设公式，可得： $dp[i] = i * dp[i - 1] + (-1)^n$

根据这个新的递推公式，可以写出代码如下：

解法3:

```

1 class Solution {
2 public:
3     int findDerangement(int n) {
4         long long res = 1;
5         for (int i = 1; i <= n; ++i) {
6             res = (i * res + (i % 2 == 0 ? 1 : -1)) % 1000000007;
7         }
8         return res;
9     }
10 };

```

CPP

623. 设计日志存储系统

You are given several logs that each log contains a unique id and timestamp. Timestamp is a string that has the following format: Year:Month:Day:Hour:Minute:Second, for example, 2017:01:01:23:59:59. All domains are zero-padded decimal numbers.

Design a log storage system to implement the following functions:

`void Put(int id, string timestamp)`: Given a log's unique id and timestamp, store the log in your storage system.

`int[] Retrieve(String start, String end, String granularity)`: Return the id of logs whose timestamps are within the range from start to end. Start and end all have the same format as timestamp. However, granularity means the time level for consideration. For example, start = "2017:01:01:23:59:59", end = "2017:01:02:23:59:59", granularity = "Day", it means that we need to find the logs within the range from Jan. 1st 2017 to Jan. 2nd 2017.

这道题让我们设计一个日志存储系统，给了日志的生成时间和日志编号，日志的生成时间是精确到秒的，然后我们主要需要完成一个retrieve函数，这个函数会给出一个起始时间，结束时间，还有一个granularity精确度，可以精确到任意的年月日时分秒，可以分析下题目中的例子，应该不难理解。我们首先需要一个数据结构来存储每个日志的编号和时间戳，那么这里我们就用一个数组，里面存pair，这样就能存下日志的数据了。然后由于我们要用到精确度，所以我们用一个units数组来列出所有可能的精确度了。下面就是本题的难点了，如何能正确的在时间范围内取出日志。由于精确度的存在，比如精确度是Day，那么我们就不关心后面的时分秒是多少了，只需要比到天就行了。判断是否在给定的时间范围内的方法也很简单，看其是否大于起始时间，且小于结束时间，我们甚至可以直接用字符串相比较，不用换成秒啥的太麻烦。所以我们可以根据时间精度确定要比的子字符串的位置，然后直接相比就行了。所以我们需要一个indices数组，来对应我们的units数组，记录下每个时间精度下取出的字符的个数。然后在retrieve函数中，遍历所有的日志，快速的根据时间精度取出对应的时间戳并且和起始结束时间相比，在其之间的就把序号加入结果res即可，参见代码如下：

```

1  class LogSystem {
2  public:
3      LogSystem() {
4          units = {"Year", "Month", "Day", "Hour", "Minute", "Second"};
5          indices = {4, 7, 10, 13, 16, 19};
6      }
7
8      void put(int id, string timestamp) {
9          timestamps.push_back({id, timestamp});
10     }
11
12     vector<int> retrieve(string s, string e, string gra) {
13         vector<int> res;
14         int idx = indices[find(units.begin(), units.end(), gra) - units.begin()];
15         for (auto p : timestamps) {
16             string t = p.second;
17             if (t.substr(0, idx).compare(s.substr(0, idx)) >= 0 && t.substr(0,
18                 idx).compare(e.substr(0, idx)) <= 0) {
19                 res.push_back(p.first);
20             }
21         }
22         return res;
23     }
24
25 private:
26     vector<pair<int, string>> timestamps;
27     vector<string> units;
28     vector<int> indices;
29 };

```

CPP

624. 函数的独家时间

Given the running logs of n functions that are executed in a nonpreemptive single threaded CPU, find the exclusive time of these functions.

Each function has a unique id, start from 0 to n-1. A function may be called recursively or by another function.

A log is a string has this format : function_id:start_or_end:timestamp. For example, "0:start:0" means function 0 starts from the very beginning of time 0. "0:end:0" means function 0 ends to the very end of time 0.

Exclusive time of a function is defined as the time spent within this function, the time spent by calling other functions should not be considered as this function's exclusive time. You should return the exclusive time of each function sorted by their function id.

这道题让我们函数的独家运行的时间，没错，exclusive就是要翻译成独家，要让每个函数都成为码农的独家记忆～哈～根据题目中给的例子，我们可以看出来，当一个函数start了之后，并不需要必须有end，可以直接被另一个程序start的时候强行关闭。而且，在某个时间点上调用end时，也不需要前面非得调用start，可以直接在某个时间点来个end，这样也算执行了1秒，得+1秒～咳咳，本站禁“苟”，请勿轻易吟诗。博主自以为了解了这个题的逻辑，自己写了一个，结果跪在了下面这个test case：

```
2
["0:start:0", "0:start:2", "0:end:5", "1:start:7", "1:end:7", "0:end:8"]
```

Expected:

```
[8, 1]
```

这个结果很confusing啊，你想啊，函数0运行了时间点0, 1, 2, 3, 4, 5, 8，共7秒，函数1运行了时间点7，共1秒，为啥答案不是[7, 1]而是[8, 1]呢？

根据分析网上大神们的解法，貌似时间点6还是函数0在执行。这是为啥呢，说明博主之前的理解有误，当函数0在时间点2时再次开启时，前面那个函数0应该没有被强制关闭，所以现在实际上有两个函数0在执行，所以当我们在时间点5关掉一个函数0时，还有另一个函数0在跑，所以时间点6还是函数0的，还得给函数0续1秒。这样才能解释的通这个case啊。这样的话用栈stack就比较合适了，函数开启了就压入栈，结束了就出栈，不会有函数被漏掉。这样的我们可以遍历每个log，然后把三部分分开，函数idx，类型type，时间点time。如果此时栈不空，说明之前肯定有函数在跑，那么不管当前时start还是end，之前函数时间都得增加，增加的值为time - preTime，这里的preTime是上一个时间点。然后我们更新preTime为当前时间点time。然后我们判断log的类型，如果是start，我们将当前函数压入栈；如果是end，那么我们将栈顶元素取出，对其加1秒，并且preTime也要加1秒，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<int> exclusiveTime(int n, vector<string>& logs) {
4         vector<int> res(n, 0);
5         stack<int> st;
6         int preTime = 0;
7         for (string log : logs) {
8             int found1 = log.find(":");
9             int found2 = log.find_last_of(":");
10            int idx = stoi(log.substr(0, found1));
11            string type = log.substr(found1 + 1, found2 - found1 - 1);
12            int time = stoi(log.substr(found2 + 1));
13            if (!st.empty()) {
14                res[st.top()] += time - preTime;
15            }
16            preTime = time;
17            if (type == "start") st.push(idx);
18            else {
19                auto t = st.top(); st.pop();
20                ++res[t];
21                ++preTime;
22            }
23        }
24        return res;
25    }
26 };

```

下面这种方法比较叼的地方是在于使用了C语言的sscanf函数来一步读取了三个变量，注意这里面的"[^:]"，表示copy所有字符，直到遇到'：'，这样就能把中间的start或者end拷到type中去了。而且接下来的写法跟上面也不太相同，这里先判断了type的类型，如果是start，那么再看如果栈不为空，那么栈顶函数加上时间差，这个上面讲过了，然后将当前函数压入栈；如果是end，那么栈顶元素加上时间差，还要再加1秒，这个在上面也提到了加了1秒的事，然后再将栈顶元素出栈。最后更新preTime为当前时间点。讲解中加了这么多秒，博主已经尽力了。

解法2：

```

1 class Solution {
2 public:
3     vector<int> exclusiveTime(int n, vector<string>& logs) {
4         vector<int> res(n, 0);
5         stack<int> st;
6         int preTime = 0, idx = 0, time = 0;
7         char type[10];
8         for (string log : logs) {
9             sscanf(log.c_str(), "%d:[^:]%d", &idx, type, &time);
10            if (type[0] == 's') {
11                if (!st.empty()) {
12                    res[st.top()] += time - preTime;
13                }
14                st.push(idx);
15            } else {
16                res[st.top()] += ++time - preTime;
17                st.pop();
18            }
19            preTime = time;
20        }
21        return res;
22    }
23 };

```

625. 二叉树的层平均值

Given a non-empty binary tree, return the average value of the nodes on each level in the form of an array.

Example 1:

Input:

```

3
 / \
9  20
 /   \
15   7

```

Output: [3, 14.5, 11]

这道题让我们求一个二叉树每层的平均值，那么一看就是要进行层序遍历了，直接上queue啊，如果熟悉层序遍历的方法，那么这题就没有什么难度了，直接将每层的值累计加起来，除以该层的结点个数，存入结果res中即可，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     vector<double> averageOfLevels(TreeNode* root) {
4         if (!root) return {};
5         vector<double> res;
6         queue<TreeNode*> q{{root}};
7         while (!q.empty()) {
8             int n = q.size();
9             double sum = 0;
10            for (int i = 0; i < n; ++i) {
11                TreeNode *t = q.front(); q.pop();
12                sum += t->val;
13                if (t->left) q.push(t->left);
14                if (t->right) q.push(t->right);
15            }
16            res.push_back(sum / n);
17        }
18        return res;
19    }
20 };

```

下面这种方法虽然是利用的递归形式的先序遍历，但是其根据判断当前层数level跟结果res中已经初始化的层数之间的关系对比，能把当前结点值累计到正确的位置，而且该层的结点数也自增1，这样我们分别求了两个数组，一个数组保存了每行的所有结点值，另一个保存了每行结点的个数，这样对应位相除就是我们要求的结果了，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<double> averageOfLevels(TreeNode* root) {
4         vector<double> res, cnt;
5         helper(root, 0, cnt, res);
6         for (int i = 0; i < res.size(); ++i) {
7             res[i] /= cnt[i];
8         }
9         return res;
10    }
11    void helper(TreeNode* node, int level, vector<double>& cnt, vector<double>& res) {
12        if (!node) return;
13        if (res.size() <= level) {
14            res.push_back(0);
15            cnt.push_back(0);
16        }
17        res[level] += node->val;
18        ++cnt[level];
19        helper(node->left, level + 1, cnt, res);
20        helper(node->right, level + 1, cnt, res);
21    }
22 };

```

626. 购物优惠

In LeetCode Store, there are some kinds of items to sell. Each item has a price.

However, there are some special offers, and a special offer consists of one or more different kinds of items with a sale price.

You are given the each item's price, a set of special offers, and the number we need to buy for each item. The job is to output the lowest price you have to pay for exactly certain items as given, where you could make optimal use of the special offers.

Each special offer is represented in the form of an array, the last number represents the price you need to pay for this special offer, other numbers represents how many specific items you could get if you buy this offer.

You could use any of special offers as many times as you want.

这道题说有一些商品，各自有不同的价格，然后给我们了一些优惠券，可以在优惠的价格买各种商品若干个，要求我们每个商品要买特定的个数，问我们使用优惠券能少花多少钱，注意优惠券可以重复使用，而且商品不能多买。那么我们可以先求出不使用任何商品需要花的钱数作为结果res的初始值，然后我们遍历每一个coupon，定义一个变量isValid表示当前coupon可以使用，然后遍历每一个商品，如果某个商品需要的个数小于coupon中提供的个数，说明当前coupon不可用，isValid标记为false。如果遍历完了发现isValid还为true的话，表明该coupon可用，我们可以更新结果res，对剩余的needs调用递归并且加上使用该coupon需要付的钱数。最后别忘了恢复needs的状态，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int shoppingOffers(vector<int>& price, vector<vector<int>>& special, vector<int>&
4 needs) {
5         int res = 0, n = price.size();
6         for (int i = 0; i < n; ++i) {
7             res += price[i] * needs[i];
8         }
9         for (auto offer : special) {
10             bool isValid = true;
11             for (int j = 0; j < n; ++j) {
12                 if (needs[j] - offer[j] < 0) isValid = false;
13                 needs[j] -= offer[j];
14             }
15             if (isValid) {
16                 res = min(res, shoppingOffers(price, special, needs) + offer.back());
17             }
18             for (int j = 0; j < n; ++j) {
19                 needs[j] += offer[j];
20             }
21         }
22     return res;
23 }
};
```

CPP

下面这种解法也是递归的写法，总的来说思路跟上面没有啥差别，应该不难理解，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int shoppingOffers(vector<int>& price, vector<vector<int>>& special, vector<int>&
4 needs) {
5         int res = inner_product(price.begin(), price.end(), needs.begin(), 0);
6         for (auto offer : special) {
7             vector<int> r = helper(offer, needs);
8             if (r.empty()) continue;
9             res = min(res, shoppingOffers(price, special, r) + offer.back());
10        }
11        return res;
12    }
13    vector<int> helper(vector<int>& offer, vector<int>& needs) {
14        vector<int> r(needs.size(), 0);
15        for (int i = 0; i < needs.size(); ++i) {
16            if (offer[i] > needs[i]) return {};
17            r[i] = needs[i] - offer[i];
18        }
19        return r;
20    }
21};

```

627. 解码方法之二

A message containing letters from A-Z is being encoded to numbers using the following mapping way:

```

'A' -> 1
'B' -> 2
...
'Z' -> 26

```

Beyond that, now the encoded string can also contain the character '*', which can be treated as one of the numbers from 1 to 9.

Given the encoded message containing digits and the character '*', return the total number of ways to decode it.

Also, since the answer may be very large, you should return the output mod 109 + 7.

这道解码的题是之前那道Decode Ways的拓展，难度提高了不少，引入了星号，可以代表1到9之间的任意数字，是不是有点外卡匹配的感觉。有了星号以后，整个题就变得异常的复杂，所以结果才让我们对一个很大的数求余，避免溢出。这道题的难点就是要分情况种类太多，一定要全部理通顺才行。我们还是用DP来做，建立一个一维dp数组，其中 $dp[i]$ 表示前*i*个字符的解码方法等个数，长度为字符串的长度加1。将 $dp[0]$ 初始化为1，然后我们判断，如果字符串第一个字符是0，那么直接返回0，如果是*，则 $dp[1]$ 初始化为9，否则初始化为1。下面就来计算一般情况下的 $dp[i]$ 了，我们从*i*=2开始遍历，由于要分的情况种类太多，我们先选一个大分支，就是当前遍历到的字符 $s[i-1]$ ，只有三种情况，要么是0，要么是1到9的数字，要么是星号。我们一个一个来分析：

首先来看 $s[i-1]$ 为0的情况，这种情况相对来说比较简单，因为0不能单独拆开，只能跟前面的数字一起，而且前面的数字只能是1或2，其他的直接返回0即可。那么当前面的数字是1或2的时候， $dp[i]$ 的种类数就跟 $dp[i-2]$ 相等，可以参见之前那道Decode Ways的讲解，因为后两数无法单独拆分开，就无法产生新的解码方法，所以只保持住原来的拆分数量就不错了；如果前面的数是星号的时候，那么前面的数可以为1或者2，这样就相等于两倍的 $dp[i-2]$ ；如果前面的数也为0，直接返回0即可。

再来看 $s[i-1]$ 为1到9之间的数字的情况，首先搞清楚当前数字是可以单独拆分出来的，那么 $dp[i]$ 至少是等于 $dp[i-1]$ 的，不会拖后腿，还要看其能不能和前面的数字组成两位数进一步增加解码方法。那么就要分情况讨论前面一个数字的种类，如果当前数字可以跟前面的数字组成一个小于等于26的两位数的话， $dp[i]$ 还需要加上 $dp[i-2]$ ；如果前面的数字为星号的话，那么要看当前的数字是否小于等于6，如果是小于等于6，那么前面的数字就可以是1或者2了，此时 $dp[i]$ 需要加上两倍的 $dp[i-2]$ ，如果大于6，那么前面的数字只能是1，所以 $dp[i]$ 只能加上 $dp[i-2]$ 。

最后来看 $s[i-1]$ 为星号的情况，如果当前数字为星号，那么就创造9种可以单独拆分的方法，所以那么 $dp[i]$ 至少是等于9倍的 $dp[i-1]$ ，还要看其能不能和前面的数字组成两位数进一步增加解码方法。那么就要分情况讨论前面一个数字的种类，如果前面的数字是1，那么当前的9种情况都可以跟前面的数字组成两位数，所以 $dp[i]$ 需要加上9倍的 $dp[i-2]$ ；如果前面的数字是2，那么只有小于等于6的6种情况都可以跟前面的数字组成两位数，所以 $dp[i]$ 需要加上6倍的 $dp[i-2]$ ；如果前面的数字是星号，那么就是上面两种情况的总和， $dp[i]$ 需要加上15倍的 $dp[i-2]$ 。

每次算完 $dp[i]$ 别忘了对超大数取余，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int numDecodings(string s) {
4         int n = s.size(), M = 1e9 + 7;
5         vector<long> dp(n + 1, 0);
6         dp[0] = 1;
7         if (s[0] == '0') return 0;
8         dp[1] = (s[0] == '*') ? 9 : 1;
9         for (int i = 2; i <= n; ++i) {
10             if (s[i - 1] == '0') {
11                 if (s[i - 2] == '1' || s[i - 2] == '2') {
12                     dp[i] += dp[i - 2];
13                 } else if (s[i - 2] == '*') {
14                     dp[i] += 2 * dp[i - 2];
15                 } else {
16                     return 0;
17                 }
18             } else if (s[i - 1] >= '1' && s[i - 1] <= '9') {
19                 dp[i] += dp[i - 1];
20                 if (s[i - 2] == '1' || (s[i - 2] == '2' && s[i - 1] <= '6')) {
21                     dp[i] += dp[i - 2];
22                 } else if (s[i - 2] == '*') {
23                     dp[i] += (s[i - 1] <= '6') ? (2 * dp[i - 2]) : dp[i - 2];
24                 }
25             } else { // s[i - 1] == '*'
26                 dp[i] += 9 * dp[i - 1];
27                 if (s[i - 2] == '1') dp[i] += 9 * dp[i - 2];
28                 else if (s[i - 2] == '2') dp[i] += 6 * dp[i - 2];
29                 else if (s[i - 2] == '*') dp[i] += 15 * dp[i - 2];
30             }
31             dp[i] %= M;
32         }
33         return dp[n];
34     }
35 };

```

下面这种解法是论坛上排名最高的解法，常数级的空间复杂度，写法非常简洁，思路也巨牛逼，博主是无论如何也想不出来的，只能继续当搬运工了。这里定义了一系列的变量 $e0, e1, e2, f0, f1, f2$ 。其中：

$e0$ 表示当前可以获得的解码的次数，当前数字可以为任意数（也就是上面解法中的 $dp[i]$ ）

$e1$ 表示当前可以获得的解码的次数，当前数字为1

$e2$ 表示当前可以获得的解码的次数，当前数字为2

$f0, f1, f2$ 分别为处理完当前字符c的 $e0, e1, e2$ 的值

那么下面我们来进行分类讨论，当c为星号的时候， $f0$ 的值就是 $9*e0 + 9*e1 + 6*e2$ ，这个应该不难理解了，可以参考上面解法中的讲解，这里的 $e0$ 就相当于 $dp[i-1]$ ， $e1$ 和 $e2$ 相当于两种不同情况的 $dp[i-2]$ ，此时 $f1$ 和 $f2$ 都赋值为 $e0$ ，因为要和后面的数字组成两位数的话，不会增加新的解码方法，所以解码总数跟之前的一样，为 $e0$ ，即 $dp[i-1]$ 。

当c不为星号的时候，如果c不为0，则 $f0$ 首先应该加上 $e0$ 。然后不管c为何值， $e1$ 都需要加上，总能和前面的1组成两位数；如果c小于等于6，可以和前面的2组成两位数，可以加上 $e2$ 。然后我们更新 $f1$ 和 $f2$ ，如果c为1，则 $f1$ 为 $e0$ ；如果c为2，则 $f2$ 为 $e0$ 。

最后别忘了将 $f0, f1, f2$ 赋值给 $e0, e1, e2$ ，其中 $f0$ 需要对超大数取余，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int numDecodings(string s) {
4         long e0 = 1, e1 = 0, e2 = 0, f0, f1, f2, M = 1e9 + 7;
5         for (char c : s) {
6             if (c == '*') {
7                 f0 = 9 * e0 + 9 * e1 + 6 * e2;
8                 f1 = e0;
9                 f2 = e0;
10            } else {
11                f0 = (c > '0') * e0 + e1 + (c <= '6') * e2;
12                f1 = (c == '1') * e0;
13                f2 = (c == '2') * e0;
14            }
15            e0 = f0 % M;
16            e1 = f1;
17            e2 = f2;
18        }
19        return e0;
20    }
21 };

```

下面这解法由热心网友edyyyy提供，在解法二的基础上去掉了两个变量，节省了行数，很符合博主的极简风格，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int numDecodings(string s) {
4         long e0 = 1, e1 = 0, e2 = 0, f0 = 0, M = 1e9 + 7;
5         for (char c : s) {
6             if (c == '*') {
7                 f0 = 9 * e0 + 9 * e1 + 6 * e2;
8                 e1 = e0;
9                 e2 = e0;
10            } else {
11                f0 = (c > '0') * e0 + e1 + (c <= '6') * e2;
12                e1 = (c == '1') * e0;
13                e2 = (c == '2') * e0;
14            }
15            e0 = f0 % M;
16        }
17        return e0;
18    }
19 };

```

628. 解方程

Solve a given equation and return the value of x in the form of string "x=#value". The equation contains only '+', '-' operation, the variable x and its coefficient.

If there is no solution for the equation, return "No solution".

If there are infinite solutions for the equation, return "Infinite solutions".

If there is exactly one solution for the equation, we ensure that the value of x is an integer.

Example 1:

Input: "x+5-3+x=6+x-2"

Output: "x=2"

这道题给了我们一个用字符串表示的方程式，让我们求出x的解，根据例子可知，还包括x有无穷多个解和x没有解的情况。解一元一次方程没什么难度，难点在于处理字符串，如何将x的系数合并起来，将常数合并起来，化简成 $ax=b$ 的形式来求解。博主最开始的思路是先找到等号，然后左右两部分分开处理。由于要化成 $ax=b$ 的格式，所以左半部分对于x的系数都是加，右半部分对于x的系数都是减。同理，左半部分对于常数是减，右半部分对于常数是加。

那么我们就开始处理字符串了，我们定义一个符号变量sign，初始化为1，数字变量num，初始化为-1，后面会提到为啥不能初始化为0。我们遍历每一个字符，如果遇到了符号位，我们看num的值，如果num是-1的话，说明是初始值，没有更新过，我们将其赋值为0；反之，如果不是-1，说明num已经更新过了，我们乘上当前的正负符号值sign。这是为了区分"-3"和"3+3"这两种情况，遇到-3种的符号时，我们还不需要加到b中，所以num此时必须为0，而遇到3+3中的加号时，此时num已经为3了，我们要把前面的3加到b中。

遇到数字的时候，我们还是要看num的值，如果是初始值，那么就将其赋值为0，然后计算数字的时候要先给num乘10，再加上当前的数字。这样做的原因是常数不一定都是个位数字，有可能是两位数或者三位数，这样做才能正确的读入数字。我们在遇到数字的时候并不更新a或者b，我们只在遇到符号位或者x的时候才更新。这样如果最后一位是数字的话就会产生问题，所以我们要在字符串的末尾加上一个+号，这样确保了末尾数字会被处理。

遇到x的时候比较tricky，因为可能是x, 0x, -x这几种情况，我们还是首先要看num的值是否为初始值-1，如果是的话，那么就可能是x或-x这种情况，我们此时将num赋值为sign；如果num不是-1，说明num已经被更新了，可能是0x, -3x等等，所以我们要将num赋值为num*sign。这里应该就明白了为啥不能将num初始化为0了，因为一旦初始化为0了，就没法区分x和0x这两种情况了。

那么我们算完了a和b，得到了 $ax=b$ 的等式，下面的步骤就很简单了，只要分情况讨论得出正确的返回结果即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     string solveEquation(string equation) {
4         int a = 0, b = 0;
5         auto found = equation.find("=");
6         helper(equation.substr(0, found), true, a, b);
7         helper(equation.substr(found + 1), false, a, b);
8         if (a == 0 && a == b) return "Infinite solutions";
9         if (a == 0 && a != b) return "No solution";
10        return "x=" + to_string(b / a);
11    }
12    void helper(string e, bool isLeft, int& a, int& b) {
13        int sign = 1, num = -1;
14        e += "+";
15        for (int i = 0; i < e.size(); ++i) {
16            if (e[i] == '-' || e[i] == '+') {
17                num = (num == -1) ? 0 : (num * sign);
18                b += isLeft ? -num : num;
19                num = -1;
20                sign = (e[i] == '+') ? 1 : -1;
21            } else if (e[i] >= '0' && e[i] <= '9') {
22                if (num == -1) num = 0;
23                num = num * 10 + e[i] - '0';
24            } else if (e[i] == 'x') {
25                num = (num == -1) ? sign : (num * sign);
26                a += isLeft ? num : -num;
27                num = -1;
28            }
29        }
30    }
31 };

```

下面这种解法也很不错，也是求 $ax=b$ 等式中的a和b，但是没有根据等号拆分为左右两部分，而是用一个变量sign来控制是对a和b加还是减，这跟上面解法中的的sign不一样。这里没有专门管正负的变量，而是通过双指针，指向数字的范围，这个数字可以是x的系数，也可以是常量，可以带着正负号，然后通过stoi函数将字符串直接转为整型数，然后乘以sign加到a或b中。变量j会指向数字或者符号，当i大于j时，我们就提取出范围内的数字。当我们遇到x的时候，跟上面一样，要处理+x, -x, 0x的情况，我们看前一位的字符，如果是符号，那么我们直接给a加上符号值；如果是数字，就用上面的办法提取出数字乘以sign加到a中。如果遇到了等号，那么先处理前面的数字(如果有的话)，然后flip sign。最后循环结束后，还要考虑最后一位是数字的情况，要加到b中。后面分情况讨论就不多说了，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     string solveEquation(string equation) {
4         int n = equation.size(), a = 0, b = 0, sign = 1, j = 0;
5         for (int i = 0; i < n; ++i) {
6             if (equation[i] == '+' || equation[i] == '-') {
7                 if (i > j) b += stoi(equation.substr(j, i - j)) * sign;
8                 j = i;
9             } else if (equation[i] == 'x') {
10                 if (i == j || equation[i - 1] == '+') {
11                     a += sign;
12                 } else if (equation[i - 1] == '-') {
13                     a -= sign;
14                 } else {
15                     a += stoi(equation.substr(j, i - j)) * sign;
16                 }
17                 j = i + 1;
18             } else if (equation[i] == '=') {
19                 if (i > j) b += stoi(equation.substr(j, i - j)) * sign;
20                 sign = -1;
21                 j = i + 1;
22             }
23         }
24         if (j < n) b += stoi(equation.substr(j)) * sign;
25         if (a == 0 && a == b) return "Infinite solutions";
26         if (a == 0 && a != b) return "No solution";
27         return "x=" + to_string(-b / a);
28     }
29 };

```

629. 设计循环双端队列

设计实现双端队列。

你的实现需要支持以下操作：

`MyCircularDeque(k)`: 构造函数，双端队列的大小为k。

`insertFront()`: 将一个元素添加到双端队列头部。如果操作成功返回 true。

`insertLast()`: 将一个元素添加到双端队列尾部。如果操作成功返回 true。

`deleteFront()`: 从双端队列头部删除一个元素。如果操作成功返回 true。

`deleteLast()`: 从双端队列尾部删除一个元素。如果操作成功返回 true。

`getFront()`: 从双端队列头部获得一个元素。如果双端队列为空，返回 -1。

`getRear()`: 获得双端队列的最后一个元素。如果双端队列为空，返回 -1。

`isEmpty()`: 检查双端队列是否为空。

`isFull()`: 检查双端队列是否满了。

```
1 class MyCircularDeque {
2     private:
3         vector<int> buffer;
4         int cnt;
5         int k;
6         int front;
7         int rear;
8     public:
9         /** Initialize your data structure here. Set the size of the deque to be k. */
10        MyCircularDeque(int k): buffer(k, 0), cnt(0), k(k), front(k - 1), rear(0) {
11        }
12
13        /** Adds an item at the front of Deque. Return true if the operation is successful. */
14        bool insertFront(int value) {
15            if (cnt == k) {
16                return false;
17            }
18            buffer[front] = value;
19            front = (front - 1 + k) % k;
20            ++cnt;
21
22            return true;
23        }
24
25        /** Adds an item at the rear of Deque. Return true if the operation is successful. */
26        bool insertLast(int value) {
27            if (cnt == k) {
28                return false;
29            }
30            buffer[rear] = value;
31            rear = (rear + 1) % k;
32            ++cnt;
33
34            return true;
35        }
36
37        /** Deletes an item from the front of Deque. Return true if the operation is
38        successful. */
39        bool deleteFront() {
40            if (cnt == 0) {
41                return false;
42            }
43            front = (front + 1) % k;
44            --cnt;
45
46            return true;
47        }
48
49        /** Deletes an item from the rear of Deque. Return true if the operation is successful.
50    */
51        bool deleteLast() {
52            if (cnt == 0) {
53                return false;
54            }
55            rear = (rear - 1 + k) % k;
56            --cnt;
57
58            return true;
59        }
```

```
60
61     /** Get the front item from the deque. */
62     int getFront() {
63         if (cnt == 0) {
64             return -1;
65         }
66         return buffer[(front + 1) % k];
67     }
68
69     /** Get the last item from the deque. */
70     int getRear() {
71         if (cnt == 0) {
72             return -1;
73         }
74         return buffer[(rear - 1 + k) % k];
75     }
76
77     /** Checks whether the circular deque is empty or not. */
78     bool isEmpty() {
79         return cnt == 0;
80     }
81
82     /** Checks whether the circular deque is full or not. */
83     bool isFull() {
84         return cnt == k;
85     }
86};
```

630. 设计搜索自动补全系统

Design a search autocomplete system for a search engine. Users may input a sentence (at least one word and end with a special character '#'). For each character they type except '#', you need to return the top 3 historical hot sentences that have prefix the same as the part of sentence already typed. Here are the specific rules:

The hot degree for a sentence is defined as the number of times a user typed the exactly same sentence before.

The returned top 3 hot sentences should be sorted by hot degree (The first is the hottest one). If several sentences have the same degree of hot, you need to use ASCII-code order (smaller one appears first).

If less than 3 hot sentences exist, then just return as many as you can.

When the input is a special character, it means the sentence ends, and in this case, you need to return an empty list.

Your job is to implement the following functions:

The constructor function:

`AutocompleteSystem(String[] sentences, int[] times)`: This is the constructor. The input is historical data. Sentences is a string array consists of previously typed sentences. Times is the corresponding times a sentence has been typed. Your system should record these historical data.

Now, the user wants to input a new sentence. The following function will provide the next character the user types:

`List<String> input(char c)`: The input c is the next character typed by the user. The character will only be lower-case letters ('a' to 'z'), blank space (' ') or a special character ('\#'). Also, the previously typed sentence should be recorded in your system. The output will be the top 3 historical hot sentences that have prefix the same as the part of sentence already typed.

这道题让我们实现一个简单的搜索自动补全系统，我们用谷歌或者百度进行搜索时，会有这样的体验，输入些单词，搜索框会弹出一些以你输入为开头的一些完整的句子供你选择，这就是一种搜索自动补全系统。根据题目的要求，补全的句子是按之前出现的频率排列的，高频率的出现在最上面，如果频率相同，就按字母顺序来显示。输入规则是每次输入一个字符，然后返回自动补全的句子，如果遇到井字符，表示完整句子结束。那么我们肯定需要一个哈希map，建立句子和其出现频率的映射，还需要一个字符串data，用来保存之前输入过的字符。在构造函数中，给了我们一些句子，和其出现的次数，那么我们就直接将其加入哈希map，然后data初始化为空字符串。在input函数中，我们首先判读输入字符是否为井字符，如果是的话，那么表明当前的data字符串已经是一个完整的句子，在哈希表中次数加1，并且data清空，返回空集。否则的话我们将当前字符加入data字符串中，现在就要找出包含data前缀的前三高频句子了，我们使用优先队列来做，设计的思路是，始终用优先队列保存频率最高的三个句子，那么我们就应该把频率低的或者是字母顺序大的放在队首，以便随时可以移出队列，所以应该是个最小堆，队列里放句子和其出现频率的pair，并且根据其频率大小进行排序，所以我们要重写优先队列的comparator。然后我们遍历哈希表中的所有句子，我们首先要验证当前data字符串是否是其前缀，没啥好的方法，就逐个字符比较，用标识符matched，初始化为true，如果发现不匹配，则matched标记为false，并break掉。然后判断如果matched为true的话，说明data字符串是前缀，那么就把这个pair加入优先队列中，如果此时队列中的元素大于三个，那把队首元素移除，因为我们设计的是最小堆，所以频率小的句子会被先移除。然后就是将优先队列的元素加到结果res中，由于先出队列的是频率小的句子，所以要加到结果res的末尾，参见代码如下：

```

1 class AutocompleteSystem {
2 public:
3     AutocompleteSystem(vector<string> sentences, vector<int> times) {
4         for (int i = 0; i < sentences.size(); ++i) {
5             freq[sentences[i]] += times[i];
6         }
7         data = "";
8     }
9
10    vector<string> input(char c) {
11        if (c == '#') {
12            ++freq[data];
13            data = "";
14            return {};
15        }
16        data.push_back(c);
17        auto cmp = [] (pair<string, int>& a, pair<string, int>& b) {
18            return a.second > b.second || (a.second == b.second && a.first < b.first);
19        };
20        priority_queue<pair<string, int>, vector<pair<string, int>>, decltype(cmp) >
21        q(cmp);
22        for (auto f : freq) {
23            bool matched = true;
24            for (int i = 0; i < data.size(); ++i) {
25                if (data[i] != f.first[i]) {
26                    matched = false;
27                    break;
28                }
29            }
30            if (matched) {
31                q.push(f);
32                if (q.size() > 3) q.pop();
33            }
34        }
35        vector<string> res(q.size());
36        for (int i = q.size() - 1; i >= 0; --i) {
37            res[i] = q.top().first; q.pop();
38        }
39        return res;
40    }
41
42 private:
43     unordered_map<string, int> freq;
44     string data;
45 };

```

631. 子数组的最大平均值

Given an array consisting of n integers, find the contiguous subarray of given length k that has the maximum average value. And you need to output the maximum average value.

这道题给了我们一个数组nums，还有一个数字k，让我们找长度为k且平均值最大的子数组。由于子数组必须是连续的，所以我们不能给数组排序。那么怎么办呢，在博主印象中，计算子数组之和的常用方法应该是建立累加数组，然后我们可以快速计算出任意一个长度为k的子数组，用来更新结果res，从而得到最大的那个，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     double findMaxAverage(vector<int>& nums, int k) {
4         int n = nums.size();
5         vector<int> sums = nums;
6         for (int i = 1; i < n; ++i) {
7             sums[i] = sums[i - 1] + nums[i];
8         }
9         double mx = sums[k - 1];
10        for (int i = k; i < n; ++i) {
11            mx = max(mx, (double)sums[i] - sums[i - k]);
12        }
13        return mx / k;
14    }
15 };

```

由于这道题子数组的长度k是确定的，所以我们其实没有必要建立整个累加数组，而是先算出前k个数字的和，然后就像维护一个滑动窗口一样，将窗口向右移动一位，即加上一个右边的数字，减去一个左边的数字，就等同于加上右边数字减去左边数字的差值，然后每次更新结果res即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     double findMaxAverage(vector<int>& nums, int k) {
4         double sum = accumulate(nums.begin(), nums.begin() + k, 0), res = sum;
5         for (int i = k; i < nums.size(); ++i) {
6             sum += nums[i] - nums[i - k];
7             res = max(res, sum);
8         }
9         return res / k;
10    }
11 };

```

632. 子数组的最大平均值之二

Given an array consisting of n integers, find the contiguous subarray whose length is greater than or equal to k that has the maximum average value. And you need to output the maximum average value.

这道题是之前那道Maximum Average Subarray I的拓展，那道题说是要找长度为k的子数组的最大平均值，而这道题要找长度大于等于k的子数组的最大平均值。加了个大于k的条件，那么情况就复杂很多了，之前只要遍历所有长度为k的子数组就行了，现在还要包括所有长度大于k的子数组。我们首先来看brute force的方法，就是遍历所有的长度大于等于k的子数组，并计算平均值并更新结果res。那么我们先建立累加和数组sums，结果res初始化为前k个数字的平均值，然后让i从k+1个数字开始遍历，那么此时的sums[i]就是前k+1个数组组成的子数组之和，我们用其平均数来更新结果res，然后要做的就是从开头开始去掉数字，直到子数组剩余k个数字为止，然后用其平均值来更新结果res，通过这种方法，我们就遍历了所有长度大于等于k的子数组。这里需要注意的一点是，更新结果res的步骤不能写成 $res = \min(res, t / (i + 1))$ 这种形式，会TLE，必须要在if中判断 $t > res * (i + 1)$ 才能accept，写成 $t / (i + 1) > res$ 也不行，必须要用乘法，这也说明了计算机不喜欢算除法吧，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     double findMaxAverage(vector<int>& nums, int k) {
4         int n = nums.size();
5         vector<int> sums = nums;
6         for (int i = 1; i < n; ++i) {
7             sums[i] = sums[i - 1] + nums[i];
8         }
9         double res = (double)sums[k - 1] / k;
10        for (int i = k; i < n; ++i) {
11            double t = sums[i];
12            if (t > res * (i + 1)) res = t / (i + 1);
13            for (int j = i - k; j >= 0; --j) {
14                t = sums[i] - sums[j];
15                if (t > res * (i - j)) res = t / (i - j);
16            }
17        }
18        return res;
19    }
20 };

```

我们再来看一种O(n²)时间复杂度的方法，这里对上面的解法进行了空间上的优化，并没有长度为n数组，而是使用了preSum和sum两个变量来代替，preSum初始化为前k个数字之和，sum初始化为preSum，结果res初始化为前k个数字的平均值，然后从第k+1个数字开始遍历，首先preSum加上这个数字，sum更新为preSum，然后此时用当前k+1个数字的平均值来更新结果res。和上面的方法一样，我们还是要从开头开始去掉数字，直到子数组剩余k个数字为止，然后用其平均值来更新结果res，那么每次就用sum减去nums[j]，就可以不断的缩小子数组的长度了，用当前平均值更新结果res，注意还是要用乘法来判断大小，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     double findMaxAverage(vector<int>& nums, int k) {
4         double preSum = accumulate(nums.begin(), nums.begin() + k, 0);
5         double sum = preSum, res = preSum / k;
6         for (int i = k; i < nums.size(); ++i) {
7             preSum += nums[i];
8             sum = preSum;
9             if (sum > res * (i + 1)) res = sum / (i + 1);
10            for (int j = 0; j <= i - k; ++j) {
11                sum -= nums[j];
12                if (sum > res * (i - j)) res = sum / (i - j);
13            }
14        }
15        return res;
16    }
17 };

```

下面来看一种优化时间复杂度到 $O(n \lg(\max - \min))$ 的解法，其中 \max 和 \min 分别是数组中的最大值和最小值，是利用了二分搜索法，博主之前写了一篇LeetCode Binary Search Summary 二分搜索法小结的博客，这里的二分法应该是小结的第四类，也是最难的那一类，因为判断折半的方向是一个子函数，这里我们没有用子函数，而是写到了一起，可以抽出来成为一个子函数，这一类的特点就是不再是简单的大小比较，而是需要一些复杂的操作来确定折半方向。这里主要借鉴了蔡文森特大神的帖子，所求的最大平均值一定是介于原数组的最大值和最小值之间，所以我们的目标是用二分法来快速的在这个范围内找到我们要求的最大平均值，初始化 $left$ 为原数组的最小值， $right$ 为原数组的最大值，然后 mid 就是 $left$ 和 $right$ 的中间值，难点就在于如何得到 mid 和要求的最大平均值之间的大小关系，从而判断折半方向。我们想，如果我们已经算出来了这个最大平均值 $maxAvg$ ，那么对于任意一个长度大于等于 k 的数组，如果让每个数字都减去 $maxAvg$ ，那么得到的累加差值一定是小于等于0的，这个不难理解，比如下面这个数组：

```
[1, 2, 3, 4] k = 2
```

我们一眼就可以看出来最大平均值 $maxAvg = 3.5$ ，所以任何一个长度大于等于2的子数组每个数字都减去 $maxAvg$ 的差值累加起来都小于等于0，只有产生这个最大平均值的子数组[3, 4]，算出来才正好等于0，其他都是小于0的。那么我们可以根据这个特点来确定折半方向，我们通过 $left$ 和 $right$ 值算出来的 mid ，可以看作是 $maxAvg$ 的一个candidate，所以我们就让数组中的每一个数字都减去 mid ，然后算差值的累加和，一旦发现累加和大于0了，那么说明我们 mid 比 $maxAvg$ 小，这样就可以判断方向了。

我们建立一个累加和数组 $sums$ ，然后求出原数组中最小值赋给 $left$ ，最大值赋给 $right$ ，题目中说了误差是 $1e-5$ ，所以我们的循环条件就是 $right$ 比 $left$ 大 $1e-5$ ，然后我们算出来 mid ，定义一个 $minSum$ 初始化为0，布尔型变量 $check$ ，初始化为 $false$ 。然后开始遍历数组，先更新累加和数组 $sums$ ，注意这个累加和数组不是原始数字的累加，而是它们和 mid 相减的差值累加。我们的目标是找长度大于等于 k 的子数组的平均值大于 mid ，由于我们每个数组都减去了 mid ，那么就转换为找长度大于等于 k 的子数组的差累积值大于0。我们建立差值累加数组的意义就在于通过 $sums[i] - sums[j]$ 来快速算出 j 和 i 位置中间数字之和，那么我们只要 j 和 i 中间正好差 k 个数字即可，然后 $minSum$ 就是用来保存 j 位置之前的子数组差累积的最小值，所以当 $i >= k$ 时，我们用 $sums[i - k]$ 来更新 $minSum$ ，这里的 $i - k$ 就是 j 的位置，然后判断如果 $sums[i] - minSum > 0$ 了，说明我们找到了一段长度大于等于 k 的子数组平均值大于 mid 了，就可以更新 $left$ 为 mid 了，我们标记 $check$ 为 $true$ ，并退出循环。在for循环外面，当 $check$ 为 $true$ 的时候， $left$ 更新为 mid ，否则 $right$ 更新为 mid ，参见代码如下：

解法3：

```
1 class Solution {
2 public:
3     double findMaxAverage(vector<int>& nums, int k) {
4         int n = nums.size();
5         vector<double> sums(n + 1, 0);
6         double left = *min_element(nums.begin(), nums.end());
7         double right = *max_element(nums.begin(), nums.end());
8         while (right - left > 1e-5) {
9             double minSum = 0, mid = left + (right - left) / 2;
10            bool check = false;
11            for (int i = 1; i <= n; ++i) {
12                sums[i] = sums[i - 1] + nums[i - 1] - mid;
13                if (i >= k) {
14                    minSum = min(minSum, sums[i - k]);
15                }
16                if (i >= k && sums[i] > minSum) {check = true; break;}
17            }
18            if (check) left = mid;
19            else right = mid;
20        }
21        return left;
22    }
23 }
```

CPP

下面这种解法对上面的方法优化了空间复杂度，使用 $preSum$ 和 sum 来代替数组，思路和上面完全一样，可以参加上面的讲解，注意这里我们的第二个if中是判断 $i >= k - 1$ ，而上面的方法是判断 $i >= k$ ，这是因为上面的 $sums$ 数组初始化了 $n + 1$ 个元素，注意坐标的转换，而第一个if中 $i >= k$ 不变是因为 j 和 i 之间就差了 k 个，所以不需要考虑坐标的转换，参见代码如下：

解法4：

CPP

```

1 class Solution {
2 public:
3     double findMaxAverage(vector<int>& nums, int k) {
4         double left = *min_element(nums.begin(), nums.end());
5         double right = *max_element(nums.begin(), nums.end());
6         while (right - left > 1e-5) {
7             double minSum = 0, sum = 0, preSum = 0, mid = left + (right - left) / 2;
8             bool check = false;
9             for (int i = 0; i < nums.size(); ++i) {
10                 sum += nums[i] - mid;
11                 if (i >= k) {
12                     preSum += nums[i - k] - mid;
13                     minSum = min(minSum, preSum);
14                 }
15                 if (i >= k - 1 && sum > minSum) {check = true; break;}
16             }
17             if (check) left = mid;
18             else right = mid;
19         }
20         return left;
21     }
22 };

```

633. 设置不匹配

The set S originally contains numbers from 1 to n. But unfortunately, due to the data error, one of the numbers in the set got duplicated to another number in the set, which results in repetition of one number and loss of another number.

Given an array nums representing the data status of this set after the error. Your task is to firstly find the number occurs twice and then find the number that is missing. Return them in the form of an array.

这道题给了我们一个长度为n的数组，说里面的数字是从1到n，但是有一个数字重复出现了一次，从而造成了另一个数字的缺失，让我们找出重复的数字和缺失的数字。那么最直接的一种解法就是统计每个数字出现的次数了，然后再遍历次数数组，如果某个数字出现了两次就是重复数，如果出现了0次，就是缺失数，参见代码如下：

解法1：

CPP

```

1 class Solution {
2 public:
3     vector<int> findErrorNums(vector<int>& nums) {
4         vector<int> res(2, 0), cnt(nums.size(), 0);
5         for (int num : nums) ++cnt[num - 1];
6         for (int i = 0; i < cnt.size(); ++i) {
7             if (res[0] != 0 && res[1] != 0) return res;
8             if (cnt[i] == 2) res[0] = i + 1;
9             else if (cnt[i] == 0) res[1] = i + 1;
10        }
11    return res;
12 }
13 };

```

我们来看一种更省空间的解法，这种解法思路相当巧妙，遍历每个数字，然后将其应该出现的位置上的数字变为其相反数，这样如果我们再变为其相反数之前已经成负数了，说明该数字是重复数，将其将入结果res中，然后再遍历原数组，如果某个位置上的数字为正数，说明该位置对应的数字没有出现过，加入res中即可，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     vector<int> findErrorNums(vector<int>& nums) {
4         vector<int> res(2, -1);
5         for (int i : nums) {
6             if (nums[abs(i) - 1] < 0) res[0] = abs(i);
7             else nums[abs(i) - 1] *= -1;
8         }
9         for (int i = 0; i < nums.size(); ++i) {
10            if (nums[i] > 0) res[1] = i + 1;
11        }
12     return res;
13 }
14 };
```

CPP

下面这种方法也很赞，首先我们把乱序的数字放到其正确的位置上，用while循环来不停的放，直到该数字在正确的位置上，那么一旦数组有序了，我们只要从头遍历就能直接找到重复的数字，然后缺失的数字同样也就知道了，参见代码如下：

解法3：

```
1 class Solution {
2 public:
3     vector<int> findErrorNums(vector<int>& nums) {
4         for (int i = 0; i < nums.size(); ++i) {
5             while (nums[i] != nums[nums[i] - 1]) swap(nums[i], nums[nums[i] - 1]);
6         }
7         for (int i = 0; i < nums.size(); ++i) {
8             if (nums[i] != i + 1) return {nums[i], i + 1};
9         }
10    }
11 };
```

CPP

634. 链对的最大长度

You are given n pairs of numbers. In every pair, the first number is always smaller than the second number.

Now, we define a pair (c, d) can follow another pair (a, b) if and only if b < c. Chain of pairs can be formed in this fashion.

Given a set of pairs, find the length longest chain which can be formed. You needn't use up all the given pairs. You can select pairs in any order.

这道题给了我们一些链对，规定了如果后面链对的首元素大于前链对的末元素，那么这两个链对就可以链起来，问我们最大能链多少个。那么我们想，由于规定了链对的首元素一定小于尾元素，我们需要比较的是某个链表的首元素和另一个链表的尾元素之间的关系，如果整个链对数组是无序的，那么就很麻烦，所以我们需要做的是首先对链对数组进行排序，按链对的尾元素进行排序，小的放前面。这样我们就可以利用Greedy算法进行求解了。我们可以用一个栈，先将第一个链对压入栈，然后对于后面遍历到的每一个链对，我们看其首元素是否大于栈顶链对的尾元素，如果大于的话，就将当前链对压入栈，这样最后我们返回栈中元素的个数即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int findLongestChain(vector<vector<int>>& pairs) {
4         stack<vector<int>> st;
5         sort(pairs.begin(), pairs.end(), [] (vector<int>& a, vector<int>& b) {
6             return a[1] < b[1];
7         });
8         for (auto pair : pairs) {
9             if (st.empty()) st.push(pair);
10            else {
11                auto t = st.top();
12                if (pair[0] > t[1]) st.push(pair);
13            }
14        }
15        return st.size();
16    }
17 };

```

CPP

我们可以对上面解法的空间进行优化，并不需要用栈来记录最长链上的每一个链对。而是用一个变量end来记录当前比较到的尾元素的值，初始化为最小值，然后遍历的时候，如果当前链对的首元素大于end，那么结果res自增1，end更新为当前链对的尾元素，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int findLongestChain(vector<vector<int>>& pairs) {
4         int res = 0, end = INT_MIN;
5         sort(pairs.begin(), pairs.end(), [] (vector<int>& a, vector<int>& b) {
6             return a[1] < b[1];
7         });
8         for (auto pair : pairs) {
9             if (pair[0] > end) {
10                 ++res;
11                 end = pair[1];
12             }
13         }
14         return res;
15     }
16 };

```

CPP

635. 回文子字符串

Given a string, your task is to count how many palindromic substrings in this string.

The substrings with different start indexes or end indexes are counted as different substrings even they consist of same characters.

这道题给了我们一个字符串，让我们计算有多少个回文子字符串。博主看到这个题，下意识的想着应该是用DP来做，哼哼哧哧写了半天，修修补补，终于通过了，但是博主写的DP不是最简便的方法，略显复杂，这里就不贴了。还是直接讲解大神们的解法好了。其实这道题也可以用递归来做，而且思路非常的简单粗暴。就是以字符串中的每一个字符都当作回文串中间的位置，然

后向两边扩散，每当成功匹配两个左右两个字符，结果res自增1，然后再比较下一对。注意回文字符串有奇数和偶数两种形式，如果是奇数长度，那么i位置就是中间那个字符的位置，所以我们左右两遍都从i开始遍历；如果是偶数长度的，那么i是最中间两个字符的左边那个，右边那个就是i+1，这样就能cover所有的情况啦，而且都是不同的回文字字符串，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int countSubstrings(string s) {
4         if (s.empty()) return 0;
5         int n = s.size(), res = 0;
6         for (int i = 0; i < n; ++i) {
7             helper(s, i, i, res);
8             helper(s, i, i + 1, res);
9         }
10        return res;
11    }
12    void helper(string s, int i, int j, int& res) {
13        while (i >= 0 && j < s.size() && s[i] == s[j]) {
14            --i; ++j; ++res;
15        }
16    }
17 };

```

在刚开始的时候博主提到了自己写的DP的方法比较复杂，为什么呢，因为博主的 $dp[i][j]$ 定义的是范围 $[i, j]$ 之间的子字符串的个数，这样我们其实还需要一个二维数组来记录子字符串 $[i, j]$ 是否是回文串，那么我们直接就将 $dp[i][j]$ 定义成子字符串 $[i, j]$ 是否是回文串就行了，然后我们i从n-1往0遍历，j从i往n-1遍历，然后我们看 $s[i]$ 和 $s[j]$ 是否相等，这时候我们需要留意一下，有了 $s[i]$ 和 $s[j]$ 相等这个条件后，i和j的位置关系很重要，如果i和j相等了，那么 $dp[i][j]$ 肯定是true；如果i和j是相邻的，那么 $dp[i][j]$ 也是true；如果i和j中间只有一个字符，那么 $dp[i][j]$ 还是true；如果中间有多余一个字符存在，那么我们需要看 $dp[i+1][j-1]$ 是否为true，若为true，那么 $dp[i][j]$ 就是true。赋值 $dp[i][j]$ 后，如果其为true，结果res自增1，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int countSubstrings(string s) {
4         int n = s.size(), res = 0;
5         vector<vector<bool>> dp(n, vector<bool>(n, false));
6         for (int i = n - 1; i >= 0; --i) {
7             for (int j = i; j < n; ++j) {
8                 dp[i][j] = (s[i] == s[j]) && (j - i <= 2 || dp[i + 1][j - 1]);
9                 if (dp[i][j]) ++res;
10            }
11        }
12        return res;
13    }
14 };

```

636. 替换单词

In English, we have a concept called root, which can be followed by some other words to form another longer word - let's call this word successor. For example, the root an, followed by other, which can form another word another.

Now, given a dictionary consisting of many roots and a sentence. You need to replace all the successor in the sentence with the root forming it. If a successor has many roots can form it, replace it with the root with the shortest length.

You need to output the sentence after the replacement.

这道题给了我们一个前缀字典，又给了一个句子，让我们将句子中较长的单词换成其前缀(如果在前缀字典中存在的话)。我们对于句子中的一个长单词如何找前缀呢，是不是可以根据第一个字母来快速定位呢，比如cattle这个单词的首字母是c，那么我们在前缀字典中找所有开头是c的前缀，为了方便查找，我们将首字母相同的前缀都放到同一个数组中，总共需要26个数组，所以我们可以定义一个二维数组来装这些前缀。还有，我们希望短前缀在长前缀的前面，因为题目中要求用最短的前缀来替换单词，所以我们可以先按单词的长度来给所有的前缀排序，然后再依次加入对应的数组中，这样就可以保证短的前缀在前面。

下面我们要来遍历句子中的每一个单词了，由于C++中没有split函数，所以我们就采用字符串流来提取每一个单词，对于遍历到的单词，我们根据其首字母查找对应数组中所有以该首字母开始的前缀，然后直接用substr函数来提取单词中和前缀长度相同的子字符串来跟前缀比较，如果二者相等，说明可以用前缀来替换单词，然后break掉for循环。别忘了单词之前还要加上空格，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     string replaceWords(vector<string>& dict, string sentence) {
4         string res = "", t = "";
5         vector<vector<string>> v(26);
6         istringstream is(sentence);
7         sort(dict.begin(), dict.end(), [] (string &a, string &b) {return a.size() <
8             b.size();});
9         for (string word : dict) {
10             v[word[0] - 'a'].push_back(word);
11         }
12         while (is >> t) {
13             for (string word : v[t[0] - 'a']) {
14                 if (t.substr(0, word.size()) == word) {
15                     t = word;
16                     break;
17                 }
18             }
19             res += t + " ";
20         }
21         res.pop_back();
22         return res;
23     }
24 };

```

CPP

你以为想出了上面的解法，这道题就算做完了？？Naive!!! 这道题最好的解法其实是用前缀树(Trie / Prefix Tree)来做，关于前缀树使用之前有一道很好的入门题Implement Trie (Prefix Tree)。了解了前缀树的原理机制，那么我们就可以发现这道题其实很适合前缀树的特点。我们要做的就是把所有的前缀都放到前缀树里面，而且在前缀的最后一个结点的地方将标志isWord设为true，表示从根节点到当前结点是一个前缀，然后我们在遍历单词中的每一个字母，我们都在前缀树查找，如果当前字母对应的结点的标志isWord是true，我们就返回这个前缀，如果当前字母对应的结点在前缀树中不存在，我们就返回原单词，这样就能完美的解决问题了。所以啊，以后遇到了有关前缀或者类似的问题，一定不要忘了前缀树这个神器哟~

解法2：

CPP

```

1 class Solution {
2 public:
3     class TrieNode {
4 public:
5         bool isWord;
6         TrieNode *child[26];
7         TrieNode(): isWord(false) {
8             for (auto &a : child) a = NULL;
9         }
10    };
11
12    string replaceWords(vector<string>& dict, string sentence) {
13        string res = "", t = "";
14        istringstream is(sentence);
15        TrieNode *root = new TrieNode();
16        for (string word : dict) {
17            insert(root, word);
18        }
19        while (is >> t) {
20            if (!res.empty()) res += " ";
21            res += findPrefix(root, t);
22        }
23        return res;
24    }
25
26    void insert(TrieNode* node, string word) {
27        for (char c : word) {
28            if (!node->child[c - 'a']) node->child[c - 'a'] = new TrieNode();
29            node = node->child[c - 'a'];
30        }
31        node->isWord = true;
32    }
33
34    string findPrefix(TrieNode* node, string word) {
35        string cur = "";
36        for (char c : word) {
37            if (!node->child[c - 'a']) break;
38            cur.push_back(c);
39            node = node->child[c - 'a'];
40            if (node->isWord) return cur;
41        }
42        return word;
43    }
44};

```

637. 刀塔二参议院

In the world of Dota2, there are two parties: the Radiant and the Dire.

The Dota2 senate consists of senators coming from two parties. Now the senate wants to make a decision about a change in the Dota2 game. The voting for this change is a round-based procedure. In each round, each senator can exercise one of the two rights:

Ban one senator's right:

A senator can make another senator lose all his rights in this and all the following rounds.

Announce the victory:

If this senator found the senators who still have rights to vote are all from the same party, he can announce the victory and make the decision about the change in the game.

Given a string representing each senator's party belonging. The character 'R' and 'D' represent the Radiant party and the Dire party respectively. Then if there are n senators, the size of the given string will be n.

The round-based procedure starts from the first senator to the last senator in the given order. This procedure will last until the end of voting. All the senators who have lost their rights will be skipped during the procedure.

Suppose every senator is smart enough and will play the best strategy for his own party, you need to predict which party will finally announce the victory and make the change in the Dota2 game. The output should be Radiant or Dire.

该来的总会来！！！自从上次LeetCode拿提莫出题Teemo Attacking后，我就知道刀塔早晚也难逃魔掌，这道题直接就搞起了刀塔二。不过话说如果你是从魔兽3无缝过渡到刀塔，那么应该熟悉了两个阵营的叫法，近卫和天灾。刀塔二里面不知道搞什么鬼，改成了光辉和梦魇，不管了，反正跟这道题的解法没什么关系。这道题模拟了刀塔类游戏开始之前的BP过程，两个阵营按顺序Ban掉对方的英雄，看最后谁剩下来了，就返回哪个阵营。那么博主能想到的简单暴力的方法就是先统计所有R和D的个数，然后从头开始遍历，如果遇到了R，就扫描之后所有的位置，然后还要扫描R前面的位置，这就要用到数组的环形遍历的知识了，其实就是坐标对总长度取余，使其不会越界，如果我们找到了下一个D，就将其标记为B，然后对应的计数器cntR自减1。对于D也是同样处理，我们的while循环的条件是cntR和cntD都要大于0，当有一个等于0了的话，那么推出循环，返回那个不为0的阵营即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     string predictPartyVictory(string senate) {
4         int n = senate.size(), cntR = 0, cntD = 0;
5         for (char c : senate) {
6             c == 'R' ? ++cntR : ++cntD;
7         }
8         if (cntR == 0) return "Dire";
9         if (cntD == 0) return "Radiant";
10        while (cntR > 0 && cntD > 0) {
11            for (int i = 0; i < n; ++i) {
12                if (senate[i] == 'R') {
13                    for (int j = i + 1; j < i + n; ++j) {
14                        if (senate[j % n] == 'D') {
15                            senate[j % n] = 'B';
16                            --cntD;
17                            break;
18                        }
19                    }
20                } else if (senate[i] == 'D') {
21                    for (int j = i + 1; j < i + n; ++j) {
22                        if (senate[j % n] == 'R') {
23                            senate[j % n] = 'B';
24                            --cntR;
25                            break;
26                        }
27                    }
28                }
29            }
30        }
31        return cntR != 0 ? "Radiant" : "Dire";
32    }
33 };

```

上面的暴力搜索的方法略显复杂，我们其实有更好的方法来做，我们可以用两个队列queue，把各自阵营的位置存入不同的队列里面，然后进行循环，每次从两个队列各取一个位置出来，看其大小关系，小的那个说明在前面，就可以把后面的那个Ban掉，所以我们要把小的那个位置要加回队列里面，但是不能直接加原位置，因为下一轮才能再轮到他来Ban，所以我们要加上一个n，再排入队列。这样当某个队列为空时，推出循环，我们返回不为空的那个阵营，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     string predictPartyVictory(string senate) {
4         int n = senate.size();
5         queue<int> q1, q2;
6         for (int i = 0; i < n; ++i) {
7             (senate[i] == 'R') ? q1.push(i) : q2.push(i);
8         }
9         while (!q1.empty() && !q2.empty()) {
10             int i = q1.front(); q1.pop();
11             int j = q2.front(); q2.pop();
12             (i < j) ? q1.push(i + n) : q2.push(j + n);
13         }
14         return (q1.size() > q2.size()) ? "Radiant" : "Dire";
15     }
16 };

```

638. 两键的键盘

Initially on a notepad only one character 'A' is present. You can perform two operations on this notepad for each step:

Copy All: You can copy all the characters present on the notepad (partial copy is not allowed).
Paste: You can paste the characters which are copied last time.

Given a number n. You have to get exactly n 'A' on the notepad by performing the minimum number of steps permitted. Output the minimum number of steps to get n 'A'.

这道题只给了我们两个按键，如果只能选择两个按键，那么博主一定会要复制和粘贴，此二键在手，天下我有！！！果然，这道题就是给了我们复制和粘贴这两个按键，然后给了我们了一个A，我们的目标时利用这两个键来打印出n个A，注意复制的时候时全部复制，不能选择部分来复制，然后复制和粘贴都算操作步骤，问我们打印出n个A需要多少步操作。对于这种有明显的递推特征的题，我们要有隐约的感觉，一定要尝试递归和DP。递归解法一般接近于暴力搜索，但是有时候是可以优化的，从而能够通过OJ。而一旦递归不行的话，那么一般来说DP这个大杀器都能解的。还有一点，对于这种题，找规律最重要，DP要找出递推公式，而如果无法发现内在的联系，那么递推公式就比较难写出来了。所以，我们需要从简单的例子开始分析，试图找规律：

当n = 1时，已经有一个A了，我们不需要其他操作，返回0

当n = 2时，我们需要复制一次，粘贴一次，返回2

当n = 3时，我们需要复制一次，粘贴两次，返回3

当n = 4时，这里有两种做法，一种是我们需要复制一次，粘贴三次，共4步，另一种是先复制一次，粘贴一次，得到AA，然后再复制一次，粘贴一次，得到AAAA，两种方法都是返回4

当n = 5时，我们需要复制一次，粘贴四次，返回5

当n = 6时，我们需要复制一次，粘贴两次，得到AAA，再复制一次，粘贴一次，得到AAAAAA，共5步，返回5

通过分析上面这6个简单的例子，我想我们已经可以总结出一些规律了，首先对于任意一个n(除了1以外)，我们最差的情况就是用n步，不会再多于n步，但是有可能是会小于n步的，比如n=6时，就只用了5步，仔细分析一下，发现时先拼成了AAA，再复制粘贴成了AAAAAA。那么什么情况下可以利用这种方法来减少步骤呢，分析发现，小模块的长度必须要能整除n，这样才能拆分。对于n=6，我们其实还可先拼出AA，然后再复制一次，粘贴两次，得到的还是5。分析到这里，我想解题的思路应该比较清晰了，我们要找出n的所有因子，然后这个因子可以当作模块的个数，我们再算出模块的长度n/i，调用递归，加上模块的个数i来更新结果res即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int minSteps(int n) {
4         if (n == 1) return 0;
5         int res = n;
6         for (int i = n - 1; i > 1; --i) {
7             if (n % i == 0) {
8                 res = min(res, minSteps(n / i) + i);
9             }
10        }
11        return res;
12    }
13 };

```

CPP

下面这种方法是用DP来做的，我们可以看出来，其实就是上面递归解法的迭代形式，思路没有任何区别，参见代码如下：

解法2:

```

1 class Solution {
2 public:
3     int minSteps(int n) {
4         vector<int> dp(n + 1, 0);
5         for (int i = 2; i <= n; ++i) {
6             dp[i] = i;
7             for (int j = i - 1; j > 1; --j) {
8                 if (i % j == 0) {
9                     dp[i] = min(dp[i], dp[j] + i / j);
10                }
11            }
12        }
13        return dp[n];
14    }
15 };

```

CPP

下面我们来看一种省空间的方法，我们不需要记录每一个中间值，而是通过改变n的值来实时累加结果res，参见代码如下：

解法3:

```

1 class Solution {
2 public:
3     int minSteps(int n) {
4         int res = 0;
5         for (int i = 2; i <= n; ++i) {
6             while (n % i == 0) {
7                 res += i;
8                 n /= i;
9             }
10        }
11        return res;
12    }
13 };

```

CPP

639. 四键的键盘

Imagine you have a special keyboard with the following keys:

Key 1: (A): Print one 'A' on screen.

Key 2: (Ctrl-A): Select the whole screen.

Key 3: (Ctrl-C): Copy selection to buffer.

Key 4: (Ctrl-V): Print buffer on screen appending it after what has already been printed.

Now, you can only press the keyboard for N times (with the above four keys), find out the maximum numbers of 'A' you can print on screen.

这道题给了我们四个操作，分别是打印A，全选，复制，粘贴。每个操作都算一个步骤，给了我们一个数字N，问我们N个操作最多能输出多个A。我们可以分析题目中的例子可以发现，N步最少都能打印N个A出来，因为我们可以每步都是打印A。那么能超过N的情况肯定就是使用了复制粘贴，这里由于全选和复制要占用两步，所以能增加A的个数的操作其实只有N-2步，那么我们如

何确定打印几个A，剩下都是粘贴呢，其实是个trade off，A打印的太多或太少，都不会得到最大结果，所以打印A和粘贴的次数要接近，最简单的方法就是遍历所有的情况然后取最大值，打印A的次数在[1, N-3]之间，粘贴的次数为N-2-i，加上打印出的部分，就是N-1-i了，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     int maxA(int N) {
4         int res = N;
5         for (int i = 1; i < N - 2; ++i) {
6             res = max(res, maxA(i) * (N - 1 - i));
7         }
8         return res;
9     }
10 };
```

CPP

这道题也可以用DP来做，我们用一个一维数组dp，其中dp[i]表示步骤总数为i时，能打印出的最多A的个数，初始化为N+1个，然后我们来想递推公式怎么求。对于dp[i]来说，求法其实跟上面的方法一样，还是要遍历所有打印A的个数，然后乘以粘贴的次数加1，用来更新dp[i]，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     int maxA(int N) {
4         vector<int> dp(N + 1, 0);
5         for (int i = 0; i <= N; ++i) {
6             dp[i] = i;
7             for (int j = 1; j < i - 2; ++j) {
8                 dp[i] = max(dp[i], dp[j] * (i - j - 1));
9             }
10        }
11        return dp[N];
12    }
13 };
```

CPP

640. 寻找重复树

Given a binary tree, return all duplicate subtrees. For each kind of duplicate subtrees, you only need to return the root node of any one of them.

Two trees are duplicate if they have the same structure with same node values.

这道题让我们寻找重复树，博主开始的思路是遍历每个结点，将结点值相同的结点放到一起，如果再遇到相同的结点值，则调用一个判断是否是相同树的子函数，但是这样会有大量的重复运算，会TLE。后来去网上看大神们的解法，发现果然是很叼啊，用到了后序遍历，还有数组序列化，并且建立序列化跟其出现次数的映射，这样如果我们得到某个结点的序列化字符串，而该字符串正好出现的次数为1，说明之前已经有一个重复树了，我们将当前结点存入结果res，这样保证了多个重复树只会存入一个结点，参见代码如下：

```

1 class Solution {
2 public:
3     vector<TreeNode*> findDuplicateSubtrees(TreeNode* root) {
4         vector<TreeNode*> res;
5         unordered_map<string, int> m;
6         helper(root, m, res);
7         return res;
8     }
9     string helper(TreeNode* node, unordered_map<string, int>& m, vector<TreeNode*>& res) {
10        if (!node) return "#";
11        string str = to_string(node->val) + "," + helper(node->left, m, res) + "," +
12        helper(node->right, m, res);
13        if (m[str] == 1) res.push_back(node);
14        ++m[str];
15        return str;
16    }
};

```

641. 两数之和之四 - 输入是二叉搜索树

Given a Binary Search Tree and a target number, return true if there exist two elements in the BST such that their sum is equal to the given target.

这道题又是一道2sum的变种题，博主一直强调，平生不识TwoSum，刷尽LeetCode也枉然！只要是两数之和的题，一定要记得用哈希表来做，这道题只不过是把数组变成了一棵二叉树而已，换汤不换药，我们遍历二叉树就行，然后用一个哈希set，在递归函数函数中，如果node为空，返回false。如果k减去当前结点值在哈希set中存在，直接返回true；否则就将当前结点值加入哈希set，然后对左右子结点分别调用递归函数并且或起来返回即可，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     bool findTarget(TreeNode* root, int k) {
4         if (!root) return false;
5         unordered_set<int> s;
6         return helper(root, k, s);
7     }
8     bool helper(TreeNode* node, int k, unordered_set<int>& s) {
9         if (!node) return false;
10        if (s.count(k - node->val)) return true;
11        s.insert(node->val);
12        return helper(node->left, k, s) || helper(node->right, k, s);
13    }
14 };

```

我们也可以用层序遍历来做，这样就是迭代的写法了，但是利用哈希表的精髓还是没变的，参见代码如下：

解法2:

```

1 class Solution {
2 public:
3     bool findTarget(TreeNode* root, int k) {
4         if (!root) return false;
5         unordered_set<int> s;
6         queue<TreeNode*> q{{root}};
7         while (!q.empty()) {
8             auto t = q.front(); q.pop();
9             if (s.count(k - t->val)) return true;
10            s.insert(t->val);
11            if (t->left) q.push(t->left);
12            if (t->right) q.push(t->right);
13        }
14        return false;
15    }
16 };

```

642. 最大二叉树

Given an integer array with no duplicates. A maximum tree building on this array is defined as follow:

The root is the maximum number in the array.

The left subtree is the maximum tree constructed from left part subarray divided by the maximum number.

The right subtree is the maximum tree constructed from right part subarray divided by the maximum number.

这道题给了我们一个数组，让我们创建一个最大二叉树，创建规则是数组中的最大值为根结点，然后分隔出的左右部分再分别创建最大二叉树。那么明眼人一看就知道这是分治法啊，果断上递归啊。首先就是要先找出数组中的最大值，由于数组是无序的，所以没啥好的办法，就直接遍历吧，找到了最大值，就创建一个结点，然后将左右两个子数组提取出来，分别调用递归函数并将结果连到该结点上，最后将结点返回即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     TreeNode* constructMaximumBinaryTree(vector<int>& nums) {
4         if (nums.empty()) return NULL;
5         int mx = INT_MIN, mx_idx = 0;
6         for (int i = 0; i < nums.size(); ++i) {
7             if (mx < nums[i]) {
8                 mx = nums[i];
9                 mx_idx = i;
10            }
11        }
12        TreeNode *node = new TreeNode(mx);
13        vector<int> leftArr = vector<int>(nums.begin(), nums.begin() + mx_idx);
14        vector<int> rightArr = vector<int>(nums.begin() + mx_idx + 1, nums.end());
15        node->left = constructMaximumBinaryTree(leftArr);
16        node->right = constructMaximumBinaryTree(rightArr);
17        return node;
18    }
19 };

```

下面这种方法也是递归的解法，和上面的解法稍有不同的是不必提取子数组，而是用两个变量来指定子数组的范围，其他部分均和上面的解法相同，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     TreeNode* constructMaximumBinaryTree(vector<int>& nums) {
4         if (nums.empty()) return NULL;
5         return helper(nums, 0, nums.size() - 1);
6     }
7     TreeNode* helper(vector<int>& nums, int left, int right) {
8         if (left > right) return NULL;
9         int mid = left;
10        for (int i = left + 1; i <= right; ++i) {
11            if (nums[i] > nums[mid]) {
12                mid = i;
13            }
14        }
15        TreeNode *node = new TreeNode(nums[mid]);
16        node->left = helper(nums, left, mid - 1);
17        node->right = helper(nums, mid + 1, right);
18        return node;
19    }
20};

```

CPP

下面这种解法是论坛上的高分解法，使用到了一个辅助数组v来让保持降序。我们遍历数组，对于每个遍历到的数字，创建一个结点，然后进行循环，如果数组v不空，且末尾结点值小于当前数字，那么将末尾结点连到当前结点的左子结点，并且移除数组中的末尾结点，这样可以保证子结点都会小于父结点。循环结束后，如果此时数组v仍不为空，说明结点值很大，那么将当前结点连到数组末尾结点的右子结点上。之后别忘了将当前结点加入数组v中，最后返回数组v的首结点即可，如果不太容易理解的话，就把题目中的例子带入一步一步运行看一下吧，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     TreeNode* constructMaximumBinaryTree(vector<int>& nums) {
4         vector<TreeNode*> v;
5         for (int num : nums) {
6             TreeNode *cur = new TreeNode(num);
7             while (!v.empty() && v.back()->val < num) {
8                 cur->left = v.back();
9                 v.pop_back();
10            }
11            if (!v.empty()) {
12                v.back()->right = cur;
13            }
14            v.push_back(cur);
15        }
16        return v.front();
17    }
18};

```

CPP

643. 打印二叉树

Print a binary tree in an $m \times n$ 2D string array following these rules:

The row number m should be equal to the height of the given binary tree.

The column number n should always be an odd number.

The root node's value (in string format) should be put in the exactly middle of the first row it can be put. The column and the row where the root node belongs will separate the rest space into two parts (left-bottom part and right-bottom part). You should print the left subtree in the left-bottom part and print the right subtree in the right-bottom part. The left-bottom part and the right-bottom part should have the same size. Even if one subtree is none while the other is not, you don't need to print anything for the none subtree but still need to leave the space as large as that for the other subtree. However, if two subtrees are none, then you don't need to leave space for both of them.

Each unused space should contain an empty string "".

Print the subtrees following the same rules.

这道题给了我们一棵二叉树，让我们以数组的形式打印出来。数组每一行的宽度是二叉树的最底层所能有的最多结点数，存在的结点需要填入到正确的位置上。那么这道题我们就应该首先要确定返回数组的宽度，由于宽度跟数组的深度有关，所以我们首先应该算出二叉树的最大深度，直接写一个子函数返回这个最大深度，从而计算出宽度。下面就是要遍历二叉树从而在数组中加入结点值。我们先来看第一行，由于根结点只有一个，所以第一行只需要插入一个数字，不管这一行多少个位置，我们都是在最中间的位置插入结点值。下面来看第二行，我们仔细观察可以发现，如果我们将这一行分为左右两部分，那么插入的位置还是在每一部分的中间位置，这样我们只要能确定分成的部分的左右边界位置，就知道插入结点的位置了，所以应该是使用分治法的思路。在递归函数中，如果当前node不存在或者当前深度超过了最大深度直接返回，否则就给中间位置赋值为结点值，然后对于左子结点，范围是左边界到中间位置，调用递归函数，注意当前深度加1；同理对于右子结点，范围是中间位置加1到右边界，调用递归函数，注意当前深度加1，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<vector<string>> printTree(TreeNode* root) {
4         int h = getHeight(root), w = pow(2, h) - 1;
5         vector<vector<string>> res(h, vector<string>(w, ""));
6         helper(root, 0, w - 1, 0, h, res);
7         return res;
8     }
9     void helper(TreeNode* node, int i, int j, int curH, int height, vector<vector<string>>&
10    res) {
11         if (!node || curH == height) return;
12         res[curH][(i + j) / 2] = to_string(node->val);
13         helper(node->left, i, (i + j) / 2, curH + 1, height, res);
14         helper(node->right, (i + j) / 2 + 1, j, curH + 1, height, res);
15     }
16     int getHeight(TreeNode* node) {
17         if (!node) return 0;
18         return 1 + max(getHeight(node->left), getHeight(node->right));
19     }
20 };

```

CPP

下面这种方法是层序遍历二叉树，使用了两个辅助队列来做，思路都一样，只不过是迭代的写法而已，关键还是在于左右边界的处理上，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<vector<string>> printTree(TreeNode* root) {
4         int h = getHeight(root), w = pow(2, h) - 1, curH = -1;
5         vector<vector<string>> res(h, vector<string>(w, ""));
6         queue<TreeNode*> q{{root}};
7         queue<pair<int, int>> idxQ{{{0, w - 1}}};
8         while (!q.empty()) {
9             int n = q.size();
10            ++curH;
11            for (int i = 0; i < n; ++i) {
12                auto t = q.front(); q.pop();
13                auto idx = idxQ.front(); idxQ.pop();
14                if (!t) continue;
15                int left = idx.first, right = idx.second;
16                int mid = left + (right - left) / 2;
17                res[curH][mid] = to_string(t->val);
18                q.push(t->left);
19                q.push(t->right);
20                idxQ.push({left, mid});
21                idxQ.push({mid + 1, right});
22            }
23        }
24        return res;
25    }
26    int getHeight(TreeNode* node) {
27        if (!node) return 0;
28        return 1 + max(getHeight(node->left), getHeight(node->right));
29    }
30 };

```

644. 硬币路径

Given an array A (index starts at 1) consisting of N integers: A₁, A₂, ..., A_N and an integer B. The integer B denotes that from any place (suppose the index is i) in the array A, you can jump to any one of the place in the array A indexed i+1, i+2, ..., i+B if this place can be jumped to. Also, if you step on the index i, you have to pay A_i coins. If A_i is -1, it means you can't jump to the place indexed i in the array.

Now, you start from the place indexed 1 in the array A, and your aim is to reach the place indexed N using the minimum coins. You need to return the path of indexes (starting from 1 to N) in the array you should take to get to the place indexed N using minimum coins.

If there are multiple paths with the same cost, return the lexicographically smallest such path.

If it's not possible to reach the place indexed N then you need to return an empty array.

这道题给了我们一个数组A，又给了我们一个整数B，表示能走的最大步数，数组上的每个数字都是cost值，如果到达某个位置，就要加上该位置上的数字，其实位置是在第一个数字上，目标是到达末尾位置，我们需要让总cost值最小，并输入路径，如果cost相同的话，输出字母顺序小的那个路径。还有就是如果数组上的某个位置为-1的话，表示到达该位置后不能再下一个位置，而且数组末位置不能为-1。博主最开始写了一个递归的解法，结果MLE了，看来这道题对内存使用的管控极为苛刻。所以我们不能将所有的候选路径都存在内存中，而是应该建立祖先数组，即数组上每个位置放其父结点的位置，有点像联合查找Union Find中的root数组，再最后根据这个祖先数组来找出正确的路径。由于需要找出cost最小的路径，所以我们可以考虑用dp数组，其中dp[i]表示从开头到位置i的最小cost值，但是如果我们从后往前跳，那么dp[i]就是从末尾到位置i的最小cost值。

我们首先判断数组A的末尾数字是否为-1，是的话直接返回空集。否则就新建结果res数组，dp数组，和pos数组，其中dp数组都初始化为整型最大值，pos数组都初始化为-1。然后将dp数组的最后一个数字赋值为数组A的尾元素。因为我们要从后往前跳，那我们从后往前遍历，如果遇到数字-1，说明不能往前跳了，直接continue继续循环，然后对于每个遍历到的数字，我们都要遍历其上一步可能的位置的dp[j]值来更新当前dp[i]值，由于限制了步数B，所以最多能到i+B，为了防止越界，要取i+B和n-1中的较小值为界限，如果上一步dp[j]值为INT_MAX，说明上一个位置无法跳过来，直接continue，否则看上一个位置dp[j]值加上当前cost值A[i]，如果小于dp[i]，说明dp[i]需要更新，并且建立祖先数组的映射pos[i] = j。最后在循环结束后，我们判断dp[0]的值，如果是INT_MAX，说明没有跳到首位置，直接返回空集，否则我们就通过pos数组来取路径。我们从前前往后遍历pos数组来取位置，直到遇到-1停止。另外要说明的就是，这种从后往前遍历的模式得到的路径一定是字母顺序最小的，zestypanda大神的帖子中有证明，不过博主没太看懂-.-|||，可以带这个例子尝试：

```
A = [0, 0, 0], B = 2
```

上面这个例子得到的结果是[1, 2, 3]，是字母顺序最小的路径，而相同的cost路径[1, 3]，就不是字母顺序最小的路径，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     vector<int> cheapestJump(vector<int>& A, int B) {
4         if (A.back() == -1) return {};
5         int n = A.size();
6         vector<int> res, dp(n, INT_MAX), pos(n, -1);
7         dp[n - 1] = A[n - 1];
8         for (int i = n - 2; i >= 0; --i) {
9             if (A[i] == -1) continue;
10            for (int j = i + 1; j <= min(i + B, n - 1); ++j) {
11                if (dp[j] == INT_MAX) continue;
12                if (A[i] + dp[j] < dp[i]) {
13                    dp[i] = A[i] + dp[j];
14                    pos[i] = j;
15                }
16            }
17        }
18        if (dp[0] == INT_MAX) return res;
19        for (int cur = 0; cur != -1; cur = pos[cur]) {
20            res.push_back(cur + 1);
21        }
22        return res;
23    }
24};
```

下面这种方法是正向遍历的解法，正向跳的话就需要另一个数组len，len[i]表示从开头到达位置i的路径的长度，如果两个路径的cost相同，那么一定是路径长度大的字母顺序小，可以参见例子 A = [0, 0, 0], B = 2。

具体的写法就不讲了，跟上面十分类似，参考上面的讲解，需要注意的就是更新的判定条件中多了一个t == dp[i] && len[i] < len[j] + 1，就是判断当cost相同时，我们取长度大路径当作结果保存。还有就是最后查找路径时要从末尾往前遍历，只要遇到-1时停止，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<int> cheapestJump(vector<int>& A, int B) {
4         if (A.back() == -1) return {};
5         int n = A.size();
6         vector<int> res, dp(n, INT_MAX), pos(n, -1), len(n, 0);
7         dp[0] = 0;
8         for (int i = 0; i < n; ++i) {
9             if (A[i] == -1) continue;
10            for (int j = max(0, i - B); j < i; ++j) {
11                if (dp[j] == INT_MAX) continue;
12                int t = A[i] + dp[j];
13                if (t < dp[i] || (t == dp[i] && len[i] < len[j] + 1)) {
14                    dp[i] = t;
15                    pos[i] = j;
16                    len[i] = len[j] + 1;
17                }
18            }
19        }
20        if (dp[n - 1] == INT_MAX) return res;
21        for (int cur = n - 1; cur != -1; cur = pos[cur]) {
22            res.insert(res.begin(), cur + 1);
23        }
24        return res;
25    }
26};

```

CPP

645. 判断路线绕圈

Initially, there is a Robot at position (0, 0). Given a sequence of its moves, judge if this robot makes a circle, which means it moves back to the original place.

The move sequence is represented by a string. And each move is represent by a character. The valid robot moves are R (Right), L(Left), U (Up) and D (down). The output should be true or false representing whether the robot makes a circle.

这道题让我们判断一个路径是否绕圈，就是说有多少个U，就得对应多少个D。同理，L和R的个数也得相等。这不就是之前那道 Valid Parentheses 的变种么，这次博主终于举一反三了！这比括号那题还要简单，因为括号至少还有三种，这里就水平和竖直两种。比较简单的方法就是使用两个计数器，如果是U，则cnt1自增1；如果是D，cnt1自减1。同理，如果是L，则cnt1自增1；如果是R，cnt1自减1。最后只要看cnt1和cnt2是否同时为0即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool judgeCircle(string moves) {
4         int cnt1 = 0, cnt2 = 0;
5         for (char move : moves) {
6             if (move == 'U') ++cnt1;
7             else if (move == 'D') --cnt1;
8             else if (move == 'L') ++cnt2;
9             else if (move == 'R') --cnt2;
10        }
11        return cnt1 == 0 && cnt2 == 0;
12    }
13 };

```

下面这种解法使用了哈希表来建立字符和其出现的次数之间的映射，最后直接比较对应的字符出现的次数是否相等即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool judgeCircle(string moves) {
4         unordered_map<char, int> m;
5         for (char c : moves) ++m[c];
6         return m['L'] == m['R'] && m['U'] == m['D'];
7     }
8 };

```

646. 寻找K个最近元素

Given a sorted array, two integers k and x, find the k closest elements to x in the array. The result should also be sorted in ascending order. If there is a tie, the smaller elements are always preferred.

这道题给我们了一个数组，还有两个变量k和x。让我们找数组中离x最近的k个元素，而且说明了数组是有序的，如果两个数字距离x相等的话，取较小的那个。从给定的例子可以分析出x不一定是数组中的数字，我们想，由于数组是有序的，所以最后返回的k个元素也一定是有序的，那么其实就是返回了原数组的一个长度为k的子数组，转化一下，实际上相当于在长度为n的数组中去掉n-k个数字，而且去掉的顺序肯定是从两头开始去，因为距离x最远的数字肯定在首尾出现。那么问题就变的明朗了，我们每次比较首尾两个数字跟x的距离，将距离大的那个数字删除，直到剩余的数组长度为k为止，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<int> findClosestElements(vector<int>& arr, int k, int x) {
4         vector<int> res = arr;
5         while (res.size() > k) {
6             int first = 0, last = res.size() - 1;
7             if (x - res.front() <= res.back() - x) {
8                 res.pop_back();
9             } else {
10                 res.erase(res.begin());
11             }
12         }
13         return res;
14     }
15 };

```

下面这种解法是论坛上的高分解法，用到了二分搜索法。其实博主最开始用的方法并不是帖子中的这两个方法，虽然也是用的二分搜索法，但博主搜的是第一个不小于x的数，然后同时向左右两个方向遍历，每次取和x距离最小的数加入结果res中，直到取满k个为止。但是下面这种方法更加巧妙一些，二分法的判定条件做了一些改变，就可以直接找到要返回的k的数字的子数组的起始位置，感觉非常的神奇。每次比较的是mid位置和x的距离跟mid+k跟x的距离，以这两者的大小关系来确定二分法折半的方向，最后找到最近距离子数组的起始位置，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<int> findClosestElements(vector<int>& arr, int k, int x) {
4         int left = 0, right = arr.size() - k;
5         while (left < right) {
6             int mid = left + (right - left) / 2;
7             if (x - arr[mid] > arr[mid + k] - x) left = mid + 1;
8             else right = mid;
9         }
10        return vector<int>(arr.begin() + left, arr.begin() + left + k);
11    }
12 };

```

647. 将数组分割成连续子序列

You are given an integer array sorted in ascending order (may contain duplicates), you need to split them into several subsequences, where each subsequences consist of at least 3 consecutive integers. Return whether you can make such a split.

博主第一眼看到这题，心想，我去，这不就是打牌么，什么挖坑，拐3，红桃4啊，3个起连，有时候排组合的好，就不用划单儿。这道题让我们将数组分割成多个连续递增的子序列，注意这里可能会产生歧义，实际上应该是分割成一个或多个连续递增的子序列，因为[1,2,3,4,5]也是正确的解。这道题就用贪心解法就可以了，我们使用两个哈希表map，第一个map用来建立数字和其出现次数之间的映射freq，第二个用来建立可以加在某个连续子序列后的数字及其可以出现的次数之间的映射need。对于第二个map，举个例子来说，就是假如有个连，[1,2,3]，那么后面可以加上4，所以就建立4的映射。这样我们首先遍历一遍数组，统计每个数字出现的频率，然后我们开始遍历数组，对于每个遍历到的数字，首先看其当前出现的次数，如果为0，则继续循环；如果need中存在这个数字的非0映射，那么表示当前的数字可以加到某个连的末尾，我们将当前数字的映射值自减1，然后将下一个连续数字的映射值加1，因为当[1,2,3]连上4后变成[1,2,3,4]之后，就可以连上5了；如果不能连到其他子序列后面，我们来看其是否可以成为新的子序列的起点，可以通过看后面两个数字的映射值是否大于0，都大于0的话，说明可以组成3连儿，于

是将后面两个数字的映射值都自减1，还有由于组成了3连儿，在need中将末尾的下一位数字的映射值自增1；如果上面情况都不满足，说明该数字是单牌，只能划单儿，直接返回false。最后别忘了将当前数字的freq映射值自减1。退出for循环后返回true，参见代码如下：

```

1 class Solution {
2 public:
3     bool isPossible(vector<int>& nums) {
4         unordered_map<int, int> freq, need;
5         for (int num : nums) ++freq[num];
6         for (int num : nums) {
7             if (freq[num] == 0) continue;
8             else if (need[num] > 0) {
9                 --need[num];
10                ++need[num + 1];
11            } else if (freq[num + 1] > 0 && freq[num + 2] > 0) {
12                --freq[num + 1];
13                --freq[num + 2];
14                ++need[num + 3];
15            } else return false;
16            --freq[num];
17        }
18        return true;
19    }
20 };

```

CPP

648. 移除9

Start from integer 1, remove any integer that contains 9 such as 9, 19, 29...

So now, you will have a new integer sequence: 1, 2, 3, 4, 5, 6, 7, 8, 10, 11, ...

Given a positive integer n, you need to return the n-th integer after removing. Note that 1 will be the first integer.

这道题让我们移除所有包含数字9的数字，然后得到一个新的数列，给我们一个数字n，让我们求在这个新的数组中第n个数字。我们多写些数字来看看：

0, 1, 2, 3, 4, 5, 6, 7, 8 (移除了9)

10, 11, 12, 13, 14, 15, 16, 17, 18 (移除了19)

.....

80, 81, 82, 83, 84, 85, 86, 87, 88 (移除了89)

(移除了 90 - 99)

100, 101, 102, 103, 104, 105, 106, 107, 108 (移除了109)

我们可以发现，8的下一位就是10了，18的下一位是20，88的下一位是100，实际上这就是九进制的数字的规律，那么这道题就变成了将十进制数n转为九进制数，这个就没啥难度了，就每次对9取余，然后乘以base，n每次自除以9，base每次扩大10倍，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     int newInteger(int n) {
4         int res = 0, base = 1;
5         while (n > 0) {
6             res += n % 9 * base;
7             n /= 9;
8             base *= 10;
9         }
10        return res;
11    }
12};
```

我们也可以写的更简洁一些，不用base变量，将结果res先当作字符串来处理，最后再转回整型数，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     int newInteger(int n) {
4         string res = "";
5         while (n > 0) {
6             res = to_string(n % 9) + res;
7             n /= 9;
8         }
9         return stoi(res);
10    }
11};
```

将十进制数转为九进制只能算Easy的题目，既然这道题标记了Hard，我们就不应该只满足于此。因为数字9是个特例，可以用上面的巧妙的解法，但如果要移除1到8中间的任意一个呢？上面的方法就不好使了，我们还是要来看看通用的解法。又来读fun4LeetCode大神的paper了，这次大神收着写的，不算太长，还是可以好好读一读的，首先我们知道，不管是移出那个数字，新数组中的第n个数字的值m，都是要大于n本身的，我们将多出的数的个数用f(1, m)表示，则有：

$$m - f(1, m) = n$$

要求m的话，我们就要先求出 $f(1, m)$ 的值，然后加上n的值，就能得到m了。这道题无法直接求出m的值，而是采用一种迭代逼近的方法来算m。最开始的时候，我们让m为n，先求 $f(1, n)$ 的值，比如说结果为k，然后我们再算 $f(1, n + k)$ 的值，用得到的结果 k' 来更新k，再带入算 $f(1, n + k')$ ，直到 $k == f(1, n + k')$ 为止，那么此时的 $n + k$ 就是我们要求的m。

下面来看我们如何计算 $f(1, m)$ ，我们当然不可能遍历所有的数字，一位一位来查看有没有要移除的数字了，太不高效了。我们再来看看开头列举的前99个数字中移除9后剩下的数字，统计一下，总共去掉了19个包含9的数字。那我们想，如果前99个数字中要移除所有包含2的数字，会去掉多少个？其实还是19个，我们发现，前99个数字，不论去掉哪个数字，都会去掉19个数字。这是一个很重要的发现，我们再来看看这19个数是怎么分布的，首先每10个数都一定会包含一个要移除的数，比如要移除的是9，每10个数都会有一个9出现，而在90几那一行，10个数都会包含9，所以都要移除，那么我们可以总结出规律，非移除数开头的其他9行，各移除1个，移除数开头的10个都要移除，所以就有 $10+9=19$ 个。好，那么这是前99个数的情况，那么前999个数又是什么情况呢？其实很类似，非移除数开头的9行各有19个，移除数开头的有 10×19 个，所以整个就是 19×19 个，所以19这个基数很重要。

好，下面来看看各位上的数字a跟要移除数d之间的关系。有三种关系，分别是小于，等于，大于：

1) 当 $a < d$ 时，比如说我们要移除的数字是6，那么a就是1到5中的数，我们知道，每10个数中只含有一个6，所以就要移除a个6就行了，如果a在百位上，就是a * 19个，然后再加上下一位上移除的值，用等式来写就是：

$$T(1, m) = a_i * (10^i - 9^i) + T(1, m \% 10^i)$$

2) 当 $a = d$ 时，那么a此时为6，如果a是十位上的数，那么前面[1, 59]中的5个6要先移除掉，然后此时下一位有多少个数移除多个数，还要加上1。比如m如果是63，那么60, 61, 62, 63这四个数要移除，怎么算的，通过 $m \% 10 + 1$ 来计算，所以整个用等式来写就是：

$$T(1, m) = a_i * (10^i - 9^i) + m \% 10^i + 1$$

3) 当 $a > d$ 时，比如此时a为8，要移除的数字还是6，那么[60, 69]这10个数都要移除，那么实际上还要再移除7个6，分别是[1,9], [10,19], [21,29], [31,39], [41,49], [51,59], [71,79] 这7个区间中的6，那么是怎么算的，通过 $a - 1$ 来算，实际上是情况1的值再加上 10^i 个数，用等式来写就是：

$$T(1, m) = (a_i - 1) * (10^i - 9^i) + 10^i + T(1, m \% 10^i) = a_i * (10^i - 9^i) + 9^i + T(1, m \% 10^i)$$

参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int newInteger(int n) {
4         long d = 9, pre = 0, cur = 0;
5         while (true) {
6             pre = cur;
7             cur = helper(n + cur, d);
8             if (cur == pre) break;
9         }
10        return n + cur;
11    }
12    long helper(long m, long d) {
13        long res = 0, p = 1, q = 1;
14        for (long i = m; i >= 10; i /= 10) {
15            p *= 10;
16            q *= 9;
17        }
18        for (long i = m; i >= d; i %= p, p /= 10, q /= 9) {
19            long a = i / p;
20            res += a * (p - q);
21            if (a == d) {
22                res += i % p + 1; break;
23            } else if (a > d) {
24                res += q;
25            }
26        }
27        return res;
28    }
29}

```

649. 图片平滑器

Given a 2D integer matrix M representing the gray scale of an image, you need to design a smoother to make the gray scale of each cell becomes the average gray scale (rounding down) of all the 8 surrounding cells and itself. If a cell has less than 8 surrounding cells, then use as many as you can.

这道题让我们给一个图片进行平滑处理，博主其实还是有一些图像处理的背景的，一般来说都是用算子来跟图片进行卷积，但是由于这道题只是个Easy的题目，我们直接用土办法就能解了，就直接对于每一个点统计其周围点的个数，然后累加像素值，做个除法就行了，注意边界情况的处理，参见代码如下：

```

1 class Solution {
2 public:
3     vector<vector<int>> imageSmoother(vector<vector<int>>& M) {
4         if (M.empty() || M[0].empty()) return {};
5         int m = M.size(), n = M[0].size();
6         vector<vector<int>> res = M, dirs{{0,-1},{-1,-1},{-1,0},{-1,1},{0,1},{1,1},{1,0},
7             {1,-1}};
8         for (int i = 0; i < m; ++i) {
9             for (int j = 0; j < n; ++j) {
10                 int cnt = M[i][j], all = 1;
11                 for (auto dir : dirs) {
12                     int x = i + dir[0], y = j + dir[1];
13                     if (x < 0 || x >= m || y < 0 || y >= n) continue;
14                     ++all;
15                     cnt += M[x][y];
16                 }
17                 res[i][j] = cnt / all;
18             }
19         }
20         return res;
21     }
22 };

```

650. 二叉树的最大宽度

Given a binary tree, write a function to get the maximum width of the given tree. The width of a tree is the maximum width among all levels. The binary tree has the same structure as a full binary tree, but some nodes are null.

The width of one level is defined as the length between the end-nodes (the leftmost and right most non-null nodes in the level, where the null nodes between the end-nodes are also counted into the length calculation).

这道题让我们求二叉树的最大宽度，根据题目中的描述可知，这里的最大宽度不是满树的时候的最大宽度，如果是那样的话，肯定是最后一层的结点数最多。这里的最大宽度应该是两个存在的结点中间可容纳的总的结点个数，中间的结点可以为空。那么其实只要我们知道了每一层中最左边和最右边的结点的位置，我们就可以算出这一层的宽度了。所以这道题的关键就是要记录每一层中最左边结点的位置，我们知道对于一棵完美二叉树，如果根结点是深度1，那么每一层的结点数就是 $2^n - 1$ ，那么每个结点的位置就是 $[1, 2^n - 1]$ 中的一个，假设某个结点的位置是1，那么其左右子结点的位置可以直接算出来，为 $2^h \times i$ 和 $2^h \times i + 1$ ，可以自行带例子检验。由于之前说过，我们需要保存每一层的最左结点的位置，那么我们使用一个数组start，由于数组是从0开始的，我们就姑且认定根结点的深度为0，不影响结果。我们从根结点进入，深度为0，位置为1，进入递归函数。

首先判断，如果当前结点为空，那么直接返回，然后判断如果当前深度大于start数组的长度，说明当前到了新的一层的最左结点，我们将当前位置存入start数组中。然后我们用 $idx - start[h] + 1$ 来更新结果res。这里 idx 是当前结点的位置， $start[h]$ 是当前层最左结点的位置。然后对左右子结点分别调用递归函数，注意左右子结点的位置可以直接计算出来，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int widthOfBinaryTree(TreeNode* root) {
4         int res = 0;
5         vector<int> start;
6         helper(root, 0, 1, start, res);
7         return res;
8     }
9     void helper(TreeNode* node, int h, int idx, vector<int>& start, int& res) {
10        if (!node) return;
11        if (h >= start.size()) start.push_back(idx);
12        res = max(res, idx - start[h] + 1);
13        helper(node->left, h + 1, idx * 2, start, res);
14        helper(node->right, h + 1, idx * 2 + 1, start, res);
15    }
16 };

```

下面这种解法还是递归，比上面的解法稍微简洁一些，没有用结果res变量，而是递归函数直接返回最大宽度了，但是解题思路没有啥区别，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int widthOfBinaryTree(TreeNode* root) {
4         vector<int> start;
5         return helper(root, 0, 1, start);
6     }
7     int helper(TreeNode* node, int h, int idx, vector<int>& start) {
8         if (!node) return 0;
9         if (h >= start.size()) start.push_back(idx);
10        return max({idx - start[h] + 1, helper(node->left, h + 1, idx * 2, start),
11 helper(node->right, h + 1, idx * 2 + 1, start)});
12    }
13 };

```

下面这个解法用的是层序遍历，迭代的方法来写的，注意这里使用了队列queue来辅助运算，queue里存的是一个pair，结点和其当前位置，在进入新一层的循环时，首先将首结点的位置保存出来当作最左位置，然后对于遍历到的结点，都更新右结点的位置，遍历一层的结点后来计算宽度更新结果res，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int widthOfBinaryTree(TreeNode* root) {
4         if (!root) return 0;
5         int res = 0;
6         queue<pair<TreeNode*, int>> q;
7         q.push({root, 1});
8         while (!q.empty()) {
9             int left = q.front().second, right = left, n = q.size();
10            for (int i = 0; i < n; ++i) {
11                auto t = q.front().first;
12                right = q.front().second; q.pop();
13                if (t->left) q.push({t->left, right * 2});
14                if (t->right) q.push({t->right, right * 2 + 1});
15            }
16            res = max(res, right - left + 1);
17        }
18        return res;
19    }
20 };

```

651. 划分等价树

Given a binary tree with n nodes, your task is to check if it's possible to partition the tree to two trees which have the equal sum of values after removing exactly one edge on the original tree.

这道题让我们划分等价树，就是说当移除一条边后，被分成的两棵树的结点之和需要相等。那么通过观察题目中的例子我们可以发现，如果我们将每个结点的结点值变成其所有子结点的结点值之和再加上当前的结点值，那么对于例子1来说，根结点的结点值就变成了30，断开位置的结点就变成了15，那么我们就可以发现其实只要断开位置的结点值是根结点值的一半，就存在等价划分。所以这道题的难点就是更新每个结点的结点值，我们可以使用递归来做。博主最开始使用的是`unordered_set`，把更新后的每个结点值都存入集合中，但是对于test case: [0, 1, -1] 会fail，仔细分析下这个case，发现更新后的根结点值还是0，而且0已经被存入集合了，而0除以2还是0，在集合中存在，会返回true，但其实这棵树是不能等价划分的。0的情况确实比较特殊，所以我们使用`unordered_map`，建立更新后的结点值和其出现次数之间的映射，这样只有map中0的个数大于1的时候，才返回true。这样完美的避开了根结点为0的陷阱，Perfect！参见代码如下：

```

1 class Solution {
2 public:
3     bool checkEqualTree(TreeNode* root) {
4         unordered_map<int, int> m;
5         int sum = helper(root, m);
6         if (sum == 0) return m[0] > 1;
7         return sum % 2 == 0 && m.count(sum / 2);
8     }
9     int helper(TreeNode* node, unordered_map<int, int>& m) {
10        if (!node) return 0;
11        int cur = node->val + helper(node->left, m) + helper(node->right, m);
12        ++m[cur];
13        return cur;
14    }
15 };

```

652. 奇怪的打印机

There is a strange printer with the following two special requirements:

The printer can only print a sequence of the same character each time.

At each turn, the printer can print new characters starting from and ending at any places, and will cover the original existing characters.

Given a string consists of lower English letters only, your job is to count the minimum number of turns the printer needed in order to print it.

这道题说有一种奇怪的打印机每次只能打印一排相同的字符，然后可以在任意起点和终点位置之间打印新的字符，用来覆盖原有的字符。现在给了我们一个新的字符串，问我们需要几次可以正确的打印出来。题目中给了两个非常简单的例子，主要是帮助我们理解的。博主最开始想的方法是一种类似贪婪算法，先是找出出现次数最多的字符，然后算需要多少次变换能将所有其他字符都变成那个出现最多次的字符，结果fail了。然后又试了一种类似剥洋葱的方法，从首尾都分别找连续相同的字符，如果首尾字符相同，则两部分一起移去，否则就移去连续相同个数多的子序列，这种基于贪婪算法的解法还是fail了，所以这道题是典型的只能动态规划Dynamic Programming，而不能用贪婪算法Greedy Algorithm的题。这道题的解题思路跟之前那道Remove Boxes很相似，博主在那个帖子中做了详细的讲解，是根据fun4leetcode大神的帖子写的，大神的思路对解这道题也相当有帮助。其实这道题并没有之前那道Remove Boxes难，移除盒子的题有隐含的条件需要加到重现关系中，大大地增加了题目的难度，非常地难想出来，这道题没有隐含条件都是个Hard题，那道题妥妥应该是Super Hard。

好，话不多说，来分析这道题吧。思考的线索和思路很重要，不理解核心精髓，当背题侠是没用的，稍微变个形式又不会了，博主就经常是这样的-.-!!!。既然说了要用DP来做，先整个二维dp数组呗，其中 $dp[i][j]$ 表示打印出字符串 $[i, j]$ 范围内字符的最小步数，难点就是找递推公式啦。遇到乍看去没啥思路的题，博主一般会先从简单的例子开始，看能不能分析出规律，从而找到解题的线索。首先如果只有一个字符，比如字符串是"a"的话，那么直接一次打印出来就行了。如果字符串是"ab"的话，那么我们要么先打印出"aa"，再改成"ab"，或者先打印出"bb"，再改成"ab"。同理，如果字符串是"abc"的话，就需要三次打印。那么一个很明显的特征是，如果没有重复的字符，打印的次数就是字符的个数。燃鹅这题的难点就是要处理有相同字符的情况，比如字符串是"aba"的时候，我们先打"aaa"的话，两步就搞定了，如果先打"bbb"的话，就需要三步。我们再来看一个字符串"abcb"，我们知道需要需要三步，我们看如果把这个字符串分成两个部分"ab"和"bc"，它们分别的步数是1和2，加起来的3是整个的步数。而对于字符串"abba"，如果分成"ab"和"ba"，它们分别的步数也是1和2，但是总步数却是2。这是因为分出的"ab"和"ba"中的最后一个字符相同。对于字符串"abbac"，因为位置0上的a和位置3上的a相同，那么整个字符串的步数相当于"bb"和"ac"的步数之和，为3。那么分析到这，是不是有点眉目了？我们关心的是字符相等的地方，对于 $[i, j]$ 范围的字符，我们从 $i+1$ 位置上的字符开始遍历到 j ，如果和 i 位置上的字符相等，我们就以此位置为界，将 $[i+1, j]$ 范围内的字符拆为两个部分，将二者的dp值加起来，和原dp值相比，取较小的那个。所以我们的递推式如下：

```
dp[i][j] = min(dp[i][j], dp[i + 1][k - 1] + dp[k][j])      (s[k] == s[i] and i + 1 <= k <= j)
要注意一些初始化的值，dp[i][i]是1，因为一个字符嘛，打印1次，还是就是在遍历k之前，dp[i][j]初始化为 1 + dp[i + 1][j]，为啥呢，可以看成在[i + 1, j]的范围上多加了一个s[i]字符，最坏的情况就是加上的是一个不曾出现过的字符，步数顶多加1步，注意我们的i是从后往前遍历的，当然你可以从前往后遍历，参数对应好就行了，参见代码如下：
```

解法1：

```

1 class Solution {
2 public:
3     int strangePrinter(string s) {
4         int n = s.size();
5         vector<vector<int>> dp(n, vector<int>(n, 0));
6         for (int i = n - 1; i >= 0; --i) {
7             for (int j = i; j < n; ++j) {
8                 dp[i][j] = (i == j) ? 1 : (1 + dp[i + 1][j]);
9                 for (int k = i + 1; k <= j; ++k) {
10                     if (s[k] == s[i]) dp[i][j] = min(dp[i][j], dp[i + 1][k - 1] + dp[k]
11 [j]);
12                 }
13             }
14         }
15         return (n == 0) ? 0 : dp[0][n - 1];
16     }
17 };

```

理解了上面的DP的方法，那么也可以用递归的形式来写，记忆数组memo就相当于dp数组，整个思路完全一样，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int strangePrinter(string s) {
4         int n = s.size();
5         vector<vector<int>> memo(n, vector<int>(n, 0));
6         return helper(s, 0, n - 1, memo);
7     }
8     int helper(string s, int i, int j, vector<vector<int>>& memo) {
9         if (i > j) return 0;
10        if (memo[i][j]) return memo[i][j];
11        memo[i][j] = helper(s, i + 1, j, memo) + 1;
12        for (int k = i + 1; k <= j; ++k) {
13            if (s[k] == s[i]) {
14                memo[i][j] = min(memo[i][j], helper(s, i + 1, k - 1, memo) + helper(s, k,
15 j, memo));
16            }
17        }
18        return memo[i][j];
19    }
20 };

```

653. 非递减数列

Given an array with n integers, your task is to check if it could become non-decreasing by modifying at most 1 element.

We define an array is non-decreasing if $\text{array}[i] \leq \text{array}[i + 1]$ holds for every i ($1 \leq i < n$).

这道题给了我们一个数组，说我们最多有1次修改某个数字的机会，问能不能将数组变为非递减数组。题目中给的例子太少，不能覆盖所有情况，我们再来看下面三个例子：

4, 2, 3

-1, 4, 2, 3

2, 3, 3, 2, 4

我们通过分析上面三个例子可以发现，当我们发现后面的数字小于前面的数字产生冲突后，有时候需要修改前面较大的数字（比如前两个例子需要修改4），有时候却要修改后面较小的那个数字（比如前第三个例子需要修改2），那么有什么内在规律吗？是有的，判断修改那个数字其实跟再前面一个数的大小有关系，首先如果再前面的数不存在，比如例子1, 4前面没有数字了，我们直接修改前面的数字为当前的数字2即可。而当再前面的数字存在，并且小于当前数时，比如例子2, -1小于2，我们还是需要修改前面的数字4为当前数字2；如果再前面的数大于当前数，比如例子3, 3大于2，我们需要修改当前数2为前面的数3。这是修改的情况，由于我们只有一次修改的机会，所以用一个变量cnt，初始化为1，修改数字后cnt自减1，当下次再需要修改时，如果cnt已经为0了，直接返回false。遍历结束后返回true，参见代码如下：

```

1 class Solution {
2 public:
3     bool checkPossibility(vector<int>& nums) {
4         int cnt = 1, n = nums.size();
5         for (int i = 1; i < n; ++i) {
6             if (nums[i] < nums[i - 1]) {
7                 if (cnt == 0) return false;
8                 if (i == 1 || nums[i] >= nums[i - 2]) nums[i - 1] = nums[i];
9                 else nums[i] = nums[i - 1];
10                --cnt;
11            }
12        }
13        return true;
14    }
15 };

```

CPP

654. 二叉树的路径和之四

If the depth of a tree is smaller than 5, then this tree can be represented by a list of three-digits integers.

For each integer in this list:

The hundreds digit represents the depth D of this node, $1 \leq D \leq 4$.

The tens digit represents the position P of this node in the level it belongs to, $1 \leq P \leq 8$.

The position is the same as that in a full binary tree.

The units digit represents the value V of this node, $0 \leq V \leq 9$.

Given a list of ascending three-digits integers representing a binary with the depth smaller than 5. You need to return the sum of all paths from the root towards the leaves.

这道题还是让我们求二叉树的路径之和，但是跟之前不同的是，树的存储方式比较特别，并没有专门的结点，而是使用一个三位数字来存的，百位数是该结点的深度，十位上是该结点在某一层中的位置，个位数是该结点的结点值。为了求路径之和，我们肯定还是需要遍历树，但是由于没有树结点，所以我们可以用其他的数据结构代替。比如我们可以将每个结点的位置信息和结点值分离开，然后建立两者之间的映射。比如我们可以将百位数和十位数当作key，将个位数当作value，建立映射。由于题目中说了数组是有序的，所以首元素就是根结点，然后我们进行先序遍历即可。在递归函数中，我们先将深度和位置拆分出来，然后算

出左右子结点的深度和位置的两位数，我们还要维护一个变量cur，用来保存当前路径之和。如果当前结点的左右子结点不存在，说明此时cur已经是一条完整的路径之和了，加到结果res中，直接返回。否则就是对存在的左右子结点调用递归函数即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int pathSum(vector<int>& nums) {
4         if (nums.empty()) return 0;
5         int res = 0;
6         unordered_map<int, int> m;
7         for (int num : nums) {
8             m[num / 10] = num % 10;
9         }
10        helper(nums[0] / 10, m, 0, res);
11        return res;
12    }
13    void helper(int num, unordered_map<int, int>& m, int cur, int& res) {
14        int level = num / 10, pos = num % 10;
15        int left = (level + 1) * 10 + 2 * pos - 1, right = left + 1;
16        cur += m[num];
17        if (!m.count(left) && !m.count(right)) {
18            res += cur;
19            return;
20        }
21        if (m.count(left)) helper(left, m, cur, res);
22        if (m.count(right)) helper(right, m, cur, res);
23    }
24 };

```

下面这种方法是迭代的形式，我们使用的层序遍历，与先序遍历不同的是，我们不能维护一个当前路径之和的变量，这样会重复计算结点值，而是在遍历每一层的结点时，加上其父结点的值，如果某一个结点没有子结点了，才将累加起来的结点值加到结果res中，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int pathSum(vector<int>& nums) {
4         if (nums.empty()) return 0;
5         int res = 0, cur = 0;
6         unordered_map<int, int> m;
7         queue<int> q{{nums[0] / 10}};
8         for (int num : nums) {
9             m[num / 10] = num % 10;
10        }
11        while (!q.empty()) {
12            int t = q.front(); q.pop();
13            int level = t / 10, pos = t % 10;
14            int left = (level + 1) * 10 + 2 * pos - 1, right = left + 1;
15            if (!m.count(left) && !m.count(right)) {
16                res += m[t];
17            }
18            if (m.count(left)) {
19                m[left] += m[t];
20                q.push(left);
21            }
22            if (m.count(right)) {
23                m[right] += m[t];
24                q.push(right);
25            }
26        }
27        return res;
28    }
29 };

```

655. 优美排列之二

Given two integers n and k, you need to construct a list which contains n different positive integers ranging from 1 to n and obeys the following requirement:

Suppose this list is [a₁, a₂, a₃, ..., a_n], then the list [|a₁ - a₂|, |a₂ - a₃|, |a₃ - a₄|, ..., |a_{n-1} - a_n|] has exactly k distinct integers.

If there are multiple answers, print any of them.

这道题虽然也叫优美排列，但是貌似跟之前那道Beautiful Arrangement的关系不太大。这道题给我们了一个数字n和一个数字k，让找出一种排列方式，使得1到n组成的数组中相邻两个数的差的绝对值正好有k种。给了k和n的关系为 $k < n$ 。那么我们首先来考虑，是否这种条件关系下，是否已定存在这种优美排列呢，我们用一个例子来分析，比如说当 $n=8$ ，我们有数组：

```
1, 2, 3, 4, 5, 6, 7, 8
```

当我们这样有序排列的话，相邻两数的差的绝对值为1。我们想差的绝对值最大能为多少，应该是把1和8放到一起，为7。那么为了尽可能的产生不同的差的绝对值，我们在8后面需要放一个小数字，比如2，这样会产生差的绝对值6，同理，后面再跟一个大数，比如7，产生差的绝对值5，以此类推，我们得到下列数组：

```
1, 8, 2, 7, 3, 6, 4, 5
```

其差的绝对值为：7, 6, 5, 4, 3, 2, 1

共有7种，所以我们知道k最大为 $n-1$ ，所以这样的排列一定会存在。我们的策略是，先按照这种最小最大数相邻的方法排列，没排一个，k自减1，当k减到1的时候，后面的排列方法只要按照生序的方法排列，就不会产生不同的差的绝对值，这种算法的时间复杂度是 $O(n)$ ，属于比较高效的那种。我们使用两个指针，初始时分别指向1和n，然后分别从i和j取数加入结果res，每取一个数字k自减1，直到k减到1的时候，开始按升序取后面的数字，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     vector<int> constructArray(int n, int k) {
4         vector<int> res;
5         int i = 1, j = n;
6         while (i <= j) {
7             if (k > 1) res.push_back(k-- % 2 ? i++ : j--);
8             else res.push_back(i++);
9         }
10        return res;
11    }
12};
```

CPP

下面这种方法是把上面的if...else的语句用三元操作符合并成了一句，看起来更加简洁了一些。

解法2：

```
1 class Solution {
2 public:
3     vector<int> constructArray(int n, int k) {
4         vector<int> res;
5         int i = 1, j = n;
6         while (i <= j) {
7             res.push_back(k > 1 ? (k-- % 2 ? i++ : j--) : i++);
8         }
9         return res;
10    }
11};
```

CPP

656. 乘法表中的第K小的数字

Nearly every one have used the Multiplication Table. But could you find out the k-th smallest number quickly from the multiplication table?

Given the height m and the length n of a m * n Multiplication Table, and a positive integer k, you need to return the k-th smallest number in this table.

这道题跟之前那道Kth Smallest Element in a Sorted Matrix没有什么太大的区别，这里的乘法表也是各行各列分别有序的。那么之前帖子里的方法都可以拿来参考。之前帖子中的解法一在这道题中无法通过OJ，维护一个大小为k的优先队列实在是没有利用到这道题乘法表的特点，但是后两种解法都是可以的。为了快速定位出第K小的数字，我们采用二分搜索法，由于是有序矩阵，那么左上角的数字一定是最小的，而右下角的数字一定是最大的，所以这个是我们搜索的范围，然后我们算出中间数字mid，由于矩阵中不同行之间的元素并不是严格有序的，所以我们要在每一行都查找一下mid，由于乘法表每行都是连续数字1, 2, 3...乘以当前行号（从1开始计数），所以我们甚至不需要在每行中使用二分查找，而是直接定位出位置。具体做法是，先比较mid和该行最后一个数字的大小，最后一数字是n * i, i是行数，n是该行数字的个数，如果mid大的话，直接将该行所有数字的个数加入cnt，否则的话加上mid / i，比如当前行是2, 4, 6, 8, 10，如果我们要查找小于7的个数，那么就是7除以2，得3，就是有三个数小于7，直接加入cnt即可。这样我们就可以快速算出矩阵中所有小于mid的个数，根据cnt和k的大小关系，来更新我们的范围，循环推出后，left就是第K小的数字，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int findKthNumber(int m, int n, int k) {
4         int left = 1, right = m * n;
5         while (left < right) {
6             int mid = left + (right - left) / 2, cnt = 0;
7             for (int i = 1; i <= m; ++i) {
8                 cnt += (mid > n * i) ? n : (mid / i);
9             }
10            if (cnt < k) left = mid + 1;
11            else right = mid;
12        }
13        return right;
14    }
15 };

```

CPP

下面这种解法在统计小于mid的数字个数的方法上有些不同，并不是逐行来统计，而是从左下角的数字开始统计，如果该数字小于mid，说明该数字及上方所有数字都小于mid，cnt加上i个，然后向右移动一位继续比较。如果当前数字小于mid了，那么向上移动一位，直到横纵方向有一个越界停止，其他部分都和上面的解法相同，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int findKthNumber(int m, int n, int k) {
4         int left = 1, right = m * n;
5         while (left < right) {
6             int mid = left + (right - left) / 2, cnt = 0, i = m, j = 1;
7             while (i >= 1 && j <= n) {
8                 if (i * j <= mid) {
9                     cnt += i;
10                ++j;
11            } else {
12                --i;
13            }
14        }
15        if (cnt < k) left = mid + 1;
16        else right = mid;
17    }
18    return right;
19 }
20 };

```

下面这种解法由网友bamboo提供，是对解法二的优化，再快一点，使用除法来快速定位新的j值，然后迅速算出当前行的小于mid的数的个数，然后快速更新i的值，这比之前那种一次只加1或减1的方法要高效许多，感觉像是解法一和解法二的混合体，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int findKthNumber(int m, int n, int k) {
4         int left = 1, right = m * n;
5         while (left < right) {
6             int mid = left + (right - left) / 2, cnt = 0, i = m, j = 1;
7             while (i >= 1 && j <= n) {
8                 int t = j;
9                 j = (mid > n * i) ? n + 1 : (mid / i + 1);
10                cnt += (j - t) * i;
11                i = mid / j;
12            }
13            if (cnt < k) left = mid + 1;
14            else right = mid;
15        }
16        return right;
17    }
18 };

```

657. 修剪一棵二叉搜索树

Given a binary search tree and the lowest and highest boundaries as L and R, trim the tree so that all its elements lies in [L, R] ($R \geq L$). You might need to change the root of the tree, so the result should return the new root of the trimmed binary search tree.

这道题让我们修剪一棵二叉搜索树，给了个边界范围[L, R]，所有不在这个范围内的结点应该被移除掉，但是仍需要保留二叉搜索树的性质，即左<根<右，有时候是小于等于。博主最开始的想法是先遍历一遍二叉树，将在返回内的结点值都放到一个数组后，遍历结束后再根据数组重建一棵二叉搜索树。这种方法会在某些test case上fail掉，可能会改变原来的二叉搜索树的结构，所以

我们只能换一种思路。正确方法其实应该是在遍历的过程中就修改二叉树，移除不合题意的结点。当然对于二叉树的题，十有八九都是要用递归来解的。首先判断如果root为空，那么直接返回空即可。然后就是要看根结点是否在范围内，如果根结点值小于L，那么返回对其右子结点调用递归函数的值；如果根结点大于R，那么返回对其左子结点调用递归函数的值。如果根结点在范围内，将其左子结点更新为对其左子结点调用递归函数的返回值，同样，将其右子结点更新为对其右子结点调用递归函数的返回值。最后返回root即可，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     TreeNode* trimBST(TreeNode* root, int L, int R) {
4         if (!root) return NULL;
5         if (root->val < L) return trimBST(root->right, L, R);
6         if (root->val > R) return trimBST(root->left, L, R);
7         root->left = trimBST(root->left, L, R);
8         root->right = trimBST(root->right, L, R);
9         return root;
10    }
11 };

```

CPP

下面这种方法是迭代的写法，虽然树的题一般都是用递归来写，简洁又美观。但是我们也可以强行用while来代替递归，比如下面这种写法：

解法2：

```
1 class Solution {
2 public:
3     TreeNode* trimBST(TreeNode* root, int L, int R) {
4         if (!root) return NULL;
5         while (root->val < L || root->val > R) {
6             root = (root->val < L) ? root->right : root->left;
7         }
8         TreeNode *cur = root;
9         while (cur) {
10             while (cur->left && cur->left->val < L) {
11                 cur->left = cur->left->right;
12             }
13             cur = cur->left;
14         }
15         cur = root;
16         while (cur) {
17             while (cur->right && cur->right->val > R) {
18                 cur->right = cur->right->left;
19             }
20             cur = cur->right;
21         }
22         return root;
23     }
24 };
25 
```

CPP

658. 大置换

Given a non-negative integer, you could swap two digits at most once to get the maximum valued number. Return the maximum valued number you could get.

这道题给了我们一个数字，我们有一次机会可以置换该数字中的任意两位，让我们返回置换后的最大值，当然如果当前数字就是最大值，我们也可以选择不置换，直接返回原数。那么最简单粗暴的方法当然就是将所有可能的置换都进行一遍，然后更新结果 res，取其中较大的数字，这样一定会得到置换后的最大数字，这里使用了整型数和字符串之间的相互转换，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int maximumSwap(int num) {
4         string str = to_string(num);
5         int res = num, n = str.size();
6         for (int i = 0; i < n; ++i) {
7             for (int j = i + 1; j < n; ++j) {
8                 swap(str[i], str[j]);
9                 res = max(res, stoi(str));
10                swap(str[i], str[j]);
11            }
12        }
13        return res;
14    }
15 };

```

CPP

下面这种解法是一种更优解，思路是这样的，由于我们希望置换后的数字最大，那么肯定最好的高位上的小数字和低位上的大数字进行置换，比如题目汇总的例子1。而如果高位上的都是大数字，像例子2那样，很有可能就不需要置换。所以我们需要找到每个数字右边的最大数字(包括其自身)，这样我们再从高位像低位遍历，如果某一位上的数字小于其右边的最大数字，说明需要调换，由于最大数字可能不止出现一次，我们希望能跟较低位的数字置换，这样置换后的数字最大，所以我们就从低位向高位遍历来找那个最大的数字，找到后进行调换即可。比如对于1993这个数：

1 9 9 3

9 9 9 3 (back数组)

9 9 1 3

我们建立好back数组后，从头遍历原数字，发现1比9小，于是从末尾往前找9，找到后一置换，就得到了9913。

解法2：

```

1 class Solution {
2 public:
3     int maximumSwap(int num) {
4         string res = to_string(num), back = res;
5         for (int i = back.size() - 2; i >= 0; --i) {
6             back[i] = max(back[i], back[i + 1]);
7         }
8         for (int i = 0; i < res.size(); ++i) {
9             if (res[i] == back[i]) continue;
10            for (int j = res.size() - 1; j > i; --j) {
11                if (res[j] == back[i]) {
12                    swap(res[i], res[j]);
13                    return stoi(res);
14                }
15            }
16        }
17        return stoi(res);
18    }
19 };

```

下面这种解法建了十个桶，分别代表数字0到9，每个桶存该数字出现的最后一个位置，也就是低位。这样我们从开头开始遍历数字上的每位上的数字，然后从大桶开始遍历，如果该大桶的数字对应的位置大于当前数字的位置，说明低位的数字要大于当前高位上的数字，那么直接交换这两个数字返回即可，其实核心思路跟上面的解法蛮像的，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int maximumSwap(int num) {
4         string str = to_string(num);
5         vector<int> buckets(10, 0);
6         for (int i = 0; i < str.size(); ++i) {
7             buckets[str[i] - '0'] = i;
8         }
9         for (int i = 0; i < str.size(); ++i) {
10            for (int k = 9; k > str[i] - '0'; --k) {
11                if (buckets[k] <= i) continue;
12                swap(str[i], str[buckets[k]]);
13                return stoi(str);
14            }
15        }
16        return num;
17    }
18 };

```

659. 二叉树中第二小的结点

Given a non-empty special binary tree consisting of nodes with the non-negative value, where each node in this tree has exactly two or zero sub-node. If the node has two sub-nodes, then this node's value is the smaller value among its two sub-nodes.

Given such a binary tree, you need to output the second minimum value in the set made of all the nodes' value in the whole tree.

If no such second minimum value exists, output -1 instead.

这道题让我们找二叉树中的第二小的结点值，并且给该二叉树做了一些限制，比如对于任意一个结点，要么其没有子结点，要么就同时有两个子结点，而且父结点值是子结点值中较小的那个，当然两个子结点值可以相等。那么直接上暴力搜索呗，根据该树的附加条件可知，根结点一定是最小的结点值first，那么我们只要找出第二小的值second即可，初始化为整型的最大值。然后对根结点调用递归函数，将first和second当作参数传进去即可。在递归函数中，如果当前结点为空，直接返回，若当前结点值不等于first，说明其肯定比first要大，然后我们看其是否比second小，小的话就更新second，然后对当前结点的左右子结点分别调用递归函数即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int findSecondMinimumValue(TreeNode* root) {
4         int first = root->val, second = INT_MAX;
5         helper(root, first, second);
6         return (second == first || second == INT_MAX) ? -1 : second;
7     }
8     void helper(TreeNode* node, int& first, int& second) {
9         if (!node) return;
10        if (node->val != first && node->val < second) {
11            second = node->val;
12        }
13        helper(node->left, first, second);
14        helper(node->right, first, second);
15    }
16 };

```

CPP

下面这种方法也是用递归来做的，不过现在递归函数有了返回值，在递归函数中，还是先判断当前结点是否为空，为空直接返回-1。然后就是看当前结点是否等于first，不等于直接返回当前结点值。如果等于，我们对其左右子结点分别调用递归函数，分别得到left和right。如果left和right其中有一个为-1了，我们取其中的较大值；如果left和right都不为-1，我们取其中的较小值返回即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int findSecondMinimumValue(TreeNode* root) {
4         return helper(root, root->val);
5     }
6     int helper(TreeNode* node, int first) {
7         if (!node) return -1;
8         if (node->val != first) return node->val;
9         int left = helper(node->left, first), right = helper(node->right, first);
10        return (left == -1 || right == -1) ? max(left, right) : min(left, right);
11    }
12 };

```

CPP

下面这种递归方法更加简洁了，没有再使用专门的递归函数helper，而是对当前根结点判断其左子树是否存在，不存在就返回-1。题目中说了是非空树，所以根结点一定存在。然后我们比较如果左子结点值等于根结点值，我们则对其左子结点调用递归函数；否则left就等于其左子结点值。再比较如果右子结点值等于根结点值，则对其右子结点调用递归函数；否则right就等于其右子结点值。最后我们还是看如果left和right其中有一个为-1了，我们取其中的较大值；如果left和right都不为-1，我们取其中的较小值返回即可，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int findSecondMinimumValue(TreeNode* root) {
4         if (!root->left) return -1;
5         int left = (root->left->val == root->val) ? findSecondMinimumValue(root->left) :
6             root->left->val;
7         int right = (root->right->val == root->val) ? findSecondMinimumValue(root->right) :
8             root->right->val;
9         return (left == -1 || right == -1) ? max(left, right) : min(left, right);
10    }
11 };

```

整了三种递归的解法，来看一种迭代的解法吧，用的是层序遍历，但还是用的解法一种的不停更新second的方法，参见代码如下：

解法4：

```

1 class Solution {
2 public:
3     int findSecondMinimumValue(TreeNode* root) {
4         int first = root->val, second = INT_MAX;
5         queue<TreeNode*> q{{root}};
6         while (!q.empty()) {
7             auto t = q.front(); q.pop();
8             if (t->val != first && t->val < second) {
9                 second = t->val;
10            }
11            if (t->left) q.push(t->left);
12            if (t->right) q.push(t->right);
13        }
14        return (second == first || second == INT_MAX) ? -1 : second;
15    }
16 };

```

660. 灯泡开关之二

There is a room with n lights which are turned on initially and 4 buttons on the wall. After performing exactly m unknown operations towards buttons, you need to return how many different kinds of status of the n lights could be.

Suppose n lights are labeled as number $[1, 2, 3 \dots, n]$, function of these 4 buttons are given below:

- Flip all the lights.
- Flip lights with even numbers.
- Flip lights with odd numbers.
- Flip lights with $(3k + 1)$ numbers, $k = 0, 1, 2, \dots$

这道题是之前那道Bulb Switcher的拓展，但是关灯的方式改变了。现在有四种关灯方法，全关，关偶数灯，关奇数灯，关 $3k+1$ 的灯。现在给我们n盏灯，允许m步操作，问我们总共能组成多少种不同的状态。博主开始想，题目没有让列出所有的情况，而只是让返回总个数。那么博主觉得应该不能用递归的暴力破解来做，一般都是用DP来做啊。可是想了半天也没想出递推公式，只得作罢。只好去参考大神们的做法，发现这道题的结果并不会是一个超大数，最多情况只有8种。转念一想，也是，如果结果是一个超大数，一般都会对一个超大数 $10e7$ 来取余，而这道题并没有，所以是一个很大的hint，只不过博主没有get到。博主应该多列几种情况的，说不定就能找出规律。下面先来看一种暴力解法，首先我们先做一个小小的优化，我们来分析四种情况：

第一种情况：1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, ...

第二种情况：1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, ...

第三种情况：1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, ...

第四种情况：1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, ...

通过观察上面的数组，我们可以发现以6个为1组，都是重复的pattern，那么实际上我们可以把重复的pattern去掉而且并不会影响结果。如果n大于6，我们则对其取余再加上6，新的n跟使用原来的n会得到同样的结果，但这样降低了我们的计算量。

下面我们先来生成n个1，这里1表示灯亮，0表示灯灭，然后我们需要一个set来记录已经存在的状态，用一个queue来辅助我们的BFS运算。我们需要循环m次，因为要操作m次，每次开始循环之前，先统计出此时queue中数字的个数len，然后进行len次循环，这就像二叉树中的层序遍历，必须上一层的结点全部遍历完了才能进入下一层，当然，在每一层开始前，我们都需要情况集合s，这样每个操作之间才不会互相干扰。然后在每层的数字循环中，我们取出队首状态，然后分别调用四种方法，突然感觉，这很像迷宫遍历问题，上下左右四个方向，周围四个状态算出来，我们将不再集合set中的状态加入queue和集合set。当m次操作遍历完成后，队列queue中状态的个数即为所求，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     int flipLights(int n, int m) {
4         n == (n <= 6) ? n : (n % 6 + 6);
5         int start = (1 << n) - 1;
6         unordered_set<int> s;
7         queue<int> q{{start}};
8         for (int i = 0; i < m; ++i) {
9             int len = q.size();
10            s.clear();
11            for (int k = 0; k < len; ++k) {
12                int t = q.front(); q.pop();
13                vector<int> next{flipAll(t, n), flipEven(t, n), flipOdd(t, n), flip3k1(t,
14 n)};
15                for (int num : next) {
16                    if (s.count(num)) continue;
17                    q.push(num);
18                    s.insert(num);
19                }
20            }
21        }
22        return q.size();
23    }
24
25    int flipAll(int t, int n) {
26        int x = (1 << n) - 1;
27        return t ^ x;
28    }
29
30    int flipEven(int t, int n) {
31        for (int i = 0; i < n; i += 2) {
32            t ^= (1 << i);
33        }
34        return t;
35    }
36
37    int flipOdd(int t, int n) {
38        for (int i = 1; i < n; i += 2) {
39            t ^= (1 << i);
40        }
41        return t;
42    }
43
44    int flip3k1(int t, int n) {
45        for (int i = 0; i < n; i += 3) {
46            t ^= (1 << i);
47        }
48        return t;
49    }
};
```

上面那个方法虽然正确，但是有些复杂了，由于这道题最多只有8中情况，所以很适合分情况来讨论：

- 当m和n其中有任意一个数是0时，返回1
- 当n = 1时

只有两种情况，0和1

- 当n = 2时，

这时候要看m的次数，如果m = 1，那么有三种状态 00, 01, 10

当m > 1时，那么有四种状态，00, 01, 10, 11

- 当m = 1时，

此时n至少为3，那么我们有四种状态，000, 010, 101, 011

- 当m = 2时，

此时n至少为3，我们有七种状态：111, 101, 010, 100, 000, 001, 110

- 当m > 2时，

此时n至少为3，我们有八种状态：111, 101, 010, 100, 000, 001, 110, 011

解法2：

```
1 class Solution {
2 public:
3     int flipLights(int n, int m) {
4         if (n == 0 || m == 0) return 1;
5         if (n == 1) return 2;
6         if (n == 2) return m == 1 ? 3 : 4;
7         if (m == 1) return 4;
8         return m == 2 ? 7 : 8;
9     }
10};
```

CPP

下面这种简洁到变态的方法是史蒂芬大神观察规律得到的，他自己也在帖子中说不清为啥这样可以，但是就是呀，贴上来纯属娱乐吧~

解法3：

```
1 class Solution {
2 public:
3     int flipLights(int n, int m) {
4         n = min(n, 3);
5         return min(1 << n, 1 + m * n);
6     }
7};
```

CPP

661. 最长递增序列的个数

Given an unsorted array of integers, find the number of longest increasing subsequence.

这道题给了我们一个数组，让我们求最长递增序列的个数，题目中的两个例子也很好的说明了问题。那么对于这种求个数的问题，直觉告诉我们应该要使用DP来做。其实这道题在设计DP数组的时候有个坑，如果我们将dp[i]定义为到i位置的最长子序列的个数的话，那么递推公式不好找。但是如果我们将dp[i]定义为以nums[i]为结尾的递推序列的个数的话，再配上这些递推序列的长度，将会比较容易的发现递推关系。这里我们用len[i]表示以nums[i]为结尾的递推序列的长度，用cnt[i]表示以nums[i]为结尾的递推序列的个数，初始化都赋值为1，只要有数字，那么至少都是1。然后我们遍历数组，对于每个遍历到的数字nums[i]，我们再遍历其之前的所有数字nums[j]，当nums[i]小于等于nums[j]时，不做任何处理，因为不是递增序列。反之，则判断len[i]和len[j]的关系，如果len[i]等于len[j] + 1，说明nums[i]这个数字可以加在以nums[j]结尾的递增序列后面，并且以nums[j]结尾的递增序列个数可以直接加到以nums[i]结尾的递增序列个数上。如果len[i]小于len[j] + 1，说明我们找到了一条长度更长的递增序列，那么我们此时将len[i]更新为len[j] + 1，并且原本的递增序列都不能用了，直接用cnt[j]来代替。我们在更新完len[i]和cnt[i]之后，要更新mx和res，如果mx等于len[i]，则把cnt[i]加到res之上；如果mx小于len[i]，则更新mx为len[i]，更新结果res为cnt[i]，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int findNumberofLIS(vector<int>& nums) {
4         int res = 0, mx = 0, n = nums.size();
5         vector<int> len(n, 1), cnt(n, 1);
6         for (int i = 0; i < n; ++i) {
7             for (int j = 0; j < i; ++j) {
8                 if (nums[i] <= nums[j]) continue;
9                 if (len[i] == len[j] + 1) cnt[i] += cnt[j];
10                else if (len[i] < len[j] + 1) {
11                    len[i] = len[j] + 1;
12                    cnt[i] = cnt[j];
13                }
14            }
15            if (mx == len[i]) res += cnt[i];
16            else if (mx < len[i]) {
17                mx = len[i];
18                res = cnt[i];
19            }
20        }
21        return res;
22    }
23 };

```

CPP

下面这种方法跟上面的解法基本一样，就是把更新结果res放在了遍历完数组之后，我们利用mx来找到所有的cnt[i]，累加到结果res上，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int findNumber0fLIS(vector<int>& nums) {
4         int res = 0, mx = 0, n = nums.size();
5         vector<int> len(n, 1), cnt(n, 1);
6         for (int i = 0; i < n; ++i) {
7             for (int j = 0; j < i; ++j) {
8                 if (nums[i] <= nums[j]) continue;
9                 if (len[i] == len[j] + 1) cnt[i] += cnt[j];
10                else if (len[i] < len[j] + 1) {
11                    len[i] = len[j] + 1;
12                    cnt[i] = cnt[j];
13                }
14            }
15            mx = max(mx, len[i]);
16        }
17        for (int i = 0; i < n; ++i) {
18            if (mx == len[i]) res += cnt[i];
19        }
20        return res;
21    }
22 };

```

662. 最长连续递增序列

Given an unsorted array of integers, find the length of longest continuous increasing subsequence.

这道题让我们求一个数组的最长连续递增序列，由于有了连续这个条件，跟之前那道Number of Longest Increasing Subsequence比起来，其实难度就降低了很多。我们可以使用一个计数器，如果遇到大的数字，计数器自增1；如果是一个小的数字，则计数器重置为1。我们用一个变量cur来表示前一个数字，初始化为整型最大值，当前遍历到的数字num就和cur比较就行了，每次用cnt来更新结果res，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     int findLength0fLCIS(vector<int>& nums) {
4         int res = 0, cnt = 0, cur = INT_MAX;
5         for (int num : nums) {
6             if (num > cur) ++cnt;
7             else cnt = 1;
8             res = max(res, cnt);
9             cur = num;
10        }
11        return res;
12    }
13 };

```

下面这种方法的思路和上面的解法一样，每次都和前面一个数字来比较，注意处理无法取到钱一个数字的情况，参见代码如下：

解法2:

```

1 class Solution {
2 public:
3     int findLengthOfLCIS(vector<int>& nums) {
4         int res = 0, cnt = 0, n = nums.size();
5         for (int i = 0; i < n; ++i) {
6             if (i == 0 || nums[i - 1] < nums[i]) res = max(res, ++cnt);
7             else cnt = 1;
8         }
9         return res;
10    }
11 };

```

663. 为高尔夫赛事砍树

You are asked to cut off trees in a forest for a golf event. The forest is represented as a non-negative 2D map, in this map:

0 represents the obstacle can't be reached.
1 represents the ground can be walked through.

The place with number bigger than 1 represents a tree can be walked through, and this positive number represents the tree's height.

You are asked to cut off all the trees in this forest in the order of tree's height - always cut off the tree with lowest height first. And after cutting, the original place has the tree will become a grass (value 1).

You will start from the point (0, 0) and you should output the minimum steps you need to walk to cut off all the trees. If you can't cut off all the trees, output -1 in that situation.

You are guaranteed that no two trees have the same height and there is at least one tree needs to be cut off.

这道题让我们砍掉所有高度大于1的树，而且要求是按顺序从低到高来砍，那么本质上还是要求任意两点之间的最短距离啊。对于这种类似迷宫遍历求最短路径的题，BFS是不二之选啊。那么这道题就对高度相邻的两棵树之间调用一个BFS，所以我们可以把BFS的内容放倒子函数helper中来使用。那么我们首先就要将所有的树从低到高进行排序，我们遍历原二维矩阵，将每棵树的高度及其横纵坐标取出来，组成一个三元组，然后放到vector中，之后用sort对数组进行排序，因为sort默认是以第一个数字排序，这也是为啥我们要把高度放在第一个位置。然后我们就遍历我们的trees数组，我们的起始位置是(0, 0)，终点位置是从trees数组中取出的树的位置，然后调用BFS的helper函数，这个BFS的子函数就是很基本的写法，没啥过多需要讲解的地方，会返回最短路径的值，如果无法到达目标点，就返回-1。所以我们先检查，如果helper函数返回-1了，那么我们就直接返回-1，否则就将cnt加到结果res中。然后更新我们的起始点为当前树的位置，然后循环取下一棵树即可，参见代码如下：

```

1 class Solution {
2 public:
3     int cutOffTree(vector<vector<int>>& forest) {
4         int m = forest.size(), n = forest[0].size(), res = 0, row = 0, col = 0;
5         vector<vector<int>> trees;
6         for (int i = 0; i < m; ++i) {
7             for (int j = 0; j < n; ++j) {
8                 if (forest[i][j] > 1) trees.push_back({forest[i][j], i, j});
9             }
10        }
11        sort(trees.begin(), trees.end());
12        for (int i = 0; i < trees.size(); ++i) {
13            int cnt = helper(forest, row, col, trees[i][1], trees[i][2]);
14            if (cnt == -1) return -1;
15            res += cnt;
16            row = trees[i][1];
17            col = trees[i][2];
18        }
19        return res;
20    }
21    int helper(vector<vector<int>>& forest, int row, int col, int treeRow, int treeCol) {
22        if (row == treeRow && col == treeCol) return 0;
23        int m = forest.size(), n = forest[0].size(), cnt = 0;
24        queue<pair<int, int>> q{{row, col}};
25        vector<vector<bool>> visited(m, vector<bool>(n, false));
26        vector<vector<int>> dirs{{-1, 0}, {0, 1}, {1, 0}, {0, -1}};
27        while (!q.empty()) {
28            ++cnt;
29            for (int i = q.size() - 1; i >= 0; --i) {
30                auto t = q.front(); q.pop();
31                for (auto dir : dirs) {
32                    int x = t.first + dir[0], y = t.second + dir[1];
33                    if (x < 0 || x >= m || y < 0 || y >= n || visited[x][y] || forest[x][y]
34 == 0) continue;
35                    if (x == treeRow && y == treeCol) return cnt;
36                    visited[x][y] = true;
37                    q.push({x, y});
38                }
39            }
40        }
41        return -1;
42    }
43 };

```

664. 实现神奇字典

Implement a magic directory with buildDict, and search methods.

For the method buildDict, you'll be given a list of non-repetitive words to build a dictionary.

For the method search, you'll be given a word, and judge whether if you modify exactly one character into another character in this word, the modified word is in the dictionary you just built.

这道题让我们设计一种神奇字典的数据结构，里面有一些单词，实现的功能是当我们搜索一个单词，只有存在和这个单词只有一个位置上的字符不相同的才能返回true，否则就返回false，注意完全相同也是返回false，必须要有两个字符不同。博主首先想到了One Edit Distance那道题，只不过这道题的两个单词之间长度必须相等。所以只需检测和要搜索单词长度一样的单词即

可，所以我们用的数据结构就是根据单词的长度来分，把长度相同相同的单词放到一起，这样就可以减少搜索量。那么对于和要搜索单词进行比较的单词，由于已经保证了长度相等，我们直接进行逐个字符比较即可，用cnt表示不同字符的个数，初始化为0。如果当前遍历到的字符相等，则continue；如果当前遍历到的字符不相同，并且此时cnt已经为1了，则break，否则cnt就自增1。退出循环后，我们检测是否所有字符都比较完了且cnt为1，是的话则返回true，否则就是跟下一个词比较。如果所有词都比较完了，则返回false，参见代码如下：

解法1：

CPP

```

1  class MagicDictionary {
2  public:
3      /** Initialize your data structure here. */
4      MagicDictionary() {}
5
6      /** Build a dictionary through a list of words */
7      void buildDict(vector<string> dict) {
8          for (string word : dict) {
9              m[word.size()].push_back(word);
10         }
11     }
12
13     /** Returns if there is any word in the trie that equals to the given word after
14     modifying exactly one character */
15     bool search(string word) {
16         for (string str : m[word.size()]) {
17             int cnt = 0, i = 0;
18             for (; i < word.size(); ++i) {
19                 if (word[i] == str[i]) continue;
20                 if (word[i] != str[i] && cnt == 1) break;
21                 ++cnt;
22             }
23             if (i == word.size() && cnt == 1) return true;
24         }
25         return false;
26     }
27
28 private:
29     unordered_map<int, vector<string>> m;
30 };

```

下面这种解法实际上是用到了前缀树中的search的思路，但是我们又没有整个用到prefix tree，博主感觉那样写法略复杂，其实我们只需要借鉴一下search方法就行了。我们首先将所有的单词都放到一个集合中，然后在search函数中，我们遍历要搜索的单词的每个字符，然后把每个字符都用a-z中的字符替换一下，形成一个新词，当然遇到本身要跳过。然后在集合中看是否存在，存在的话就返回true。记得换完一圈字符后要换回去，不然就不满足只改变一个字符的条件了，参见代码如下：

解法2：

```

1 class MagicDictionary {
2 public:
3     /** Initialize your data structure here. */
4     MagicDictionary() {}
5
6     /** Build a dictionary through a list of words */
7     void buildDict(vector<string> dict) {
8         for (string word : dict) s.insert(word);
9     }
10
11    /** Returns if there is any word in the trie that equals to the given word after
12     modifying exactly one character */
13    bool search(string word) {
14        for (int i = 0; i < word.size(); ++i) {
15            char t = word[i];
16            for (char c = 'a'; c <= 'z'; ++c) {
17                if (c == t) continue;
18                word[i] = c;
19                if (s.count(word)) return true;
20            }
21            word[i] = t;
22        }
23        return false;
24    }
25
26 private:
27     unordered_set<string> s;
28 };

```

665. 映射配对之和

Implement a MapSum class with insert, and sum methods.

For the method insert, you'll be given a pair of (string, integer). The string represents the key and the integer represents the value. If the key already existed, then the original key-value pair will be overridden to the new one.

For the method sum, you'll be given a string representing the prefix, and you need to return the sum of all the pairs' value whose key starts with the prefix.

这道题让我们实现一个MapSum类，里面有两个方法，insert和sum，其中insert就是插入一个键值对，而sum方法比较特别，是在找一个前缀，需要将所有有此前缀的单词的值累加起来返回。看到这种玩前缀的题，照理来说是要用前缀树来做的。但是博主一般想偷懒，不想新写一个结构或类，于是就使用map来代替前缀树啦。博主开始想到的方法是建立前缀和一个pair之间的映射，这里的pair的第一个值表示该词的值，第二个值表示将该词作为前缀的所有词的累加值，那么我们的sum函数就异常的简单了，直接将pair中的两个值相加即可。关键就是要在insert中把数据结构建好，构建的方法也不难，首先我们suppose原本这个key是有值的，我们更新的时候只需要加上它的差值即可，就算key不存在默认就是0，算差值也没问题。然后我们将first值更新为val，然后就是遍历其所有的前缀了，给每个前缀的second都加上diff即可，参见代码如下：

解法1：

```

1 class MapSum {
2 public:
3     /** Initialize your data structure here. */
4     MapSum() {}
5
6     void insert(string key, int val) {
7         int diff = val - m[key].first, n = key.size();
8         m[key].first = val;
9         for (int i = n - 1; i > 0; --i) {
10             m[key.substr(0, i)].second += diff;
11         }
12     }
13
14     int sum(string prefix) {
15         return m[prefix].first + m[prefix].second;
16     }
17
18 private:
19     unordered_map<string, pair<int, int>> m;
20 };

```

下面这种方法是论坛上投票最高的方法，感觉很叼，用的是带排序的map，insert就是把单词加入map。在map里会按照字母顺序自动排序，然后在sum函数里，我们根据prefix来用二分查找快速定位到第一个不小于prefix的位置，然后向后遍历，向后遍历的都是以prefix为前缀的单词，如果我们发现某个单词不是以prefix为前缀了，直接break；否则就累加其val值，参见代码如下：

解法2：

```

1 class MapSum {
2 public:
3     /** Initialize your data structure here. */
4     MapSum() {}
5
6     void insert(string key, int val) {
7         m[key] = val;
8     }
9
10    int sum(string prefix) {
11        int res = 0, n = prefix.size();
12        for (auto it = m.lower_bound(prefix); it != m.end(); ++it) {
13            if (it->first.substr(0, n) != prefix) break;
14            res += it->second;
15        }
16        return res;
17    }
18
19 private:
20     map<string, int> m;
21 };
22

```

666. 验证括号字符串

Given a string containing only three types of characters: '(', ')' and '*', write a function to check whether this string is valid. We define the validity of a string by these rules:

Any left parenthesis '(' must have a corresponding right parenthesis ')'.
 Any right parenthesis ')' must have a corresponding left parenthesis '('.
 Left parenthesis '(' must go before the corresponding right parenthesis ')'.
 '*' could be treated as a single right parenthesis ')' or a single left parenthesis '(' or an empty string.
 An empty string is also valid.

这道题让我们验证括号字符串，跟之前那道Valid Parentheses有些类似。不同之处在于这道题不只有小括号，还存在星号，星号可以当左括号，右括号，或空来使用，问我们能不能得到一个合法的括号字符串。那么我们想，如果不存在星号，那么这题是不是异常的简单，我们甚至连stack都可以不用，直接用一个变量，遇到左括号，自增1，遇到右括号，如果此时计数器已经为0了，直接返回false，否则自减1，一旦计数器出现了负数，立即返回false，最后还要看变量是否为0即可。但是由于星号的存在，这道题就变的复杂了，由于星号可以当括号用，所以当遇到右括号时，就算此时变量为0，也可以用星号来当左括号使用。那么星号什么时候都能当括号来用吗，我们来看两个例子)和*(，在第一种情况下，星号可以当左括号来用，而在第二种情况下，无论星号当左括号，右括号，还是空，(都是不对的。当然这种情况只限于星号和左括号之间的位置关系，而只要星号在右括号前面，就一定可以消掉右括号。那么我们使用两个stack，分别存放左括号和星号的位置，遍历字符串，当遇到星号时，压入星号栈star，当遇到左括号时，压入左括号栈left，当遇到右括号时，此时如果left和star均为空时，直接返回false；如果left不为空，则pop一个左括号来抵消当前的右括号；否则从star中取出一个星号当作左括号来抵消右括号。当循环结束后，我们希望left中没有多余的左括号，就算有，我们可以尝试着用星号来抵消，当star和left均不为空时，进行循环，如果left的栈顶左括号的位置在star的栈顶星号的右边，那么就组成了*(模式，直接返回false；否则就说明星号可以抵消左括号，各自pop一个元素。最终退出循环后我们看left中是否还有多余的左括号，没有就返回true，否则false，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool checkValidString(string s) {
4         stack<int> left, star;
5         for (int i = 0; i < s.size(); ++i) {
6             if (s[i] == '*') star.push(i);
7             else if (s[i] == '(') left.push(i);
8             else {
9                 if (left.empty() && star.empty()) return false;
10                if (!left.empty()) left.pop();
11                else star.pop();
12            }
13        }
14        while (!left.empty() && !star.empty()) {
15            if (left.top() > star.top()) return false;
16            left.pop(); star.pop();
17        }
18        return left.empty();
19    }
20 }
```

CPP

如果你觉得上面的解法逻辑稍稍复杂了一些，我们来看一种逻辑无比简单的解法。既然星号可以当左括号和右括号，那么我们就正反各遍历一次，正向遍历的时候，我们把星号都当成左括号，此时用经典的验证括号的方法，即遇左括号计数器加1，遇右括号则自减1，如果中间某个时刻计数器小于0了，直接返回false。如果最终计数器等于0了，我们直接返回true，因为此时我们把星号都当作了左括号，可以跟所有的右括号抵消。而此时就算计数器大于0了，我们暂时不能返回false，因为有可能多余的左括号是星号变的，星号也可以表示空，所以有可能不多，我们还需要反向遍历一下，哦不，是反向遍历一下，这是我们将所有的星号当作右括号，遇右括号计数器加1，遇左括号则自减1，如果中间某个时刻计数器小于0了，直接返回false。遍历结束后直接返回true，这是为啥呢？此时计数器有两种情况，要么为0，要么大于0。为0不用说，肯定是true，为啥大于0也是true呢？因为之前

正向遍历的时候，我们的左括号多了，我们之前说过了，多余的左括号可能是星号变的，也可能是本身就多的左括号。本身就多的左括号这种情况会在反向遍历时被检测出来，如果没有检测出来，说明多余的左括号一定是星号变的。而这些星号在反向遍历时又变做了右括号，最终导致了右括号有剩余，所以当这些星号都当作空的时候，左右括号都是对应的，即是合法的。你可能会有疑问，右括号本身不会多么，其实不会的，如果多的话，会在正向遍历中被检测出来，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool checkValidString(string s) {
4         int left = 0, right = 0, n = s.size();
5         for (int i = 0; i < n; ++i) {
6             if (s[i] == '(' || s[i] == '*') ++left;
7             else --left;
8             if (left < 0) return false;
9         }
10        if (left == 0) return true;
11        for (int i = n - 1; i >= 0; --i) {
12            if (s[i] == ')' || s[i] == '*') ++right;
13            else --right;
14            if (right < 0) return false;
15        }
16        return true;
17    }
18 };

```

CPP

下面这种方法是用递归来写的，思路特别的简单直接，感觉应该属于暴力破解法。使用了变量cnt来记录左括号的个数，变量start表示当前开始遍历的位置，那么在递归函数中，首先判断如果cnt小于0，直接返回false。否则进行从start开始的遍历，如果当前字符为左括号，cnt自增1；如果为右括号，若cnt此时小于等于0，返回false，否则cnt自减1；如果为星号，我们同时递归三种情况，分别是当星号为空，左括号，或右括号，只要有一种情况返回true，那么就是true了。如果循环退出后，若cnt为0，返回true，否则false，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     bool checkValidString(string s) {
4         return helper(s, 0, 0);
5     }
6     bool helper(string s, int start, int cnt) {
7         if (cnt < 0) return false;
8         for (int i = start; i < s.size(); ++i) {
9             if (s[i] == '(') {
10                 ++cnt;
11             } else if (s[i] == ')') {
12                 if (cnt <= 0) return false;
13                 --cnt;
14             } else {
15                 return helper(s, i + 1, cnt) || helper(s, i + 1, cnt + 1) || helper(s, i +
16                 1, cnt - 1);
17             }
18         }
19         return cnt == 0;
20     }
21 };

```

CPP

下面这种解法是论坛上排第一的解法，感觉思路清新脱俗，博主研究了好久，参考了网友的留言才稍稍弄懂了一些，这里维护了两个变量low和high，其中low表示在有左括号的情况下，将星号当作右括号时左括号的个数(这样做的原因是尽量不多增加右括号的个数)，而high表示将星号当作左括号时左括号的个数。是不是很绕，没办法。那么当high小于0时，说明就算把星号都当作左括号了，还是不够抵消右括号，返回false。而当low大于0时，说明左括号的个数太多了，没有足够多的右括号来抵消，返回false。那么开始遍历字符串，当遇到左括号时，low和high都自增1；当遇到右括号时，只有当low大于0时，low才自减1，保证了low不会小于0，而high直接自减1；当遇到星号时，只有当low大于0时，low才自减1，保证了low不会小于0，而high直接自增1，因为high把星号当作左括号。当此时high小于0，说明右括号太多，返回false。当循环退出后，我们看low是否为0，参见代码如下：

解法4：

```

1 class Solution {
2 public:
3     bool checkValidString(string s) {
4         int low = 0, high = 0;
5         for (char c : s) {
6             if (c == '(') {
7                 ++low; ++high;
8             } else if (c == ')') {
9                 if (low > 0) --low;
10                --high;
11            } else {
12                if (low > 0) --low;
13                ++high;
14            }
15            if (high < 0) return false;
16        }
17        return low == 0;
18    }
19 };

```

CPP

667. 二十四点游戏

You have 4 cards each containing a number from 1 to 9. You need to judge whether they could operated through *, /, +, -, (,)to get the value of 24.

这道题就是经典的24点游戏了，记得小时候经常玩这个游戏，就是每个人发四张牌，看谁最快能算出24，这完全是脑力大比拼啊，不是拼的牌技。玩的多了，就会摸出一些套路来，比如尽量去凑2和12，3和8，4和6等等，但是对于一些特殊的case，比如[1, 5, 5, 5]这种，正确的解法是 $5 * (5 - 1 / 5)$ ，一般人都会去试加减乘，和能整除的除法，而像这种带小数的确实很难想到，但是程序计算就没问题，可以遍历所有的情况，这也是这道题的实际意义所在吧。那么既然是要遍历所有的情况，我们应该隐约感觉到应该是要使用递归来做的。我们想，任意的两个数字之间都可能进行加减乘除，其中加法和乘法对于两个数字的前后顺序没有影响，但是减法和除法是有影响的，而且做除法的时候还要另外保证除数不能为零。我们要遍历任意两个数字，然后对于这两个数字，尝试各种加减乘除后得到一个新数字，将这个新数字加到原数组中，记得原来的两个数要移除掉，然后调用递归函数进行计算，我们可以发现每次调用递归函数后，数组都减少一个数字，那么当减少到只剩一个数字了，就是最后的计算结果，所以我们在递归函数开始时判断，如果数组只有一个数字，且为24，说明可以算出24，结果res赋值为true返回。这里我们的结果res是一个全局的变量，如果已经为true了，就没必要再进行运算了，所以第一行应该是判断结果res，为true就直接返回了。我们遍历任意两个数字，分别用p和q来取出，然后进行两者各种加减乘除的运算，将结果保存进数组临时数组t，记得要判断除数不为零。然后将原数组nums中的p和q移除，遍历临时数组t中的数字，将其加入数组nums，然后调用递归函数，记得完成后要移除数字，恢复状态，这是递归解法很重要的一点。最后还要把p和q再加回原数组nums，这也是还原状态，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool judgePoint24(vector<int>& nums) {
4         bool res = false;
5         double eps = 0.001;
6         vector<double> arr(nums.begin(), nums.end());
7         helper(arr, eps, res);
8         return res;
9     }
10    void helper(vector<double>& nums, double eps, bool& res) {
11        if (res) return;
12        if (nums.size() == 1) {
13            if (abs(nums[0] - 24) < eps) res = true;
14            return;
15        }
16        for (int i = 0; i < nums.size(); ++i) {
17            for (int j = 0; j < i; ++j) {
18                double p = nums[i], q = nums[j];
19                vector<double> t{p + q, p - q, q - p, p * q};
20                if (p > eps) t.push_back(q / p);
21                if (q > eps) t.push_back(p / q);
22                nums.erase(nums.begin() + i);
23                nums.erase(nums.begin() + j);
24                for (double d : t) {
25                    nums.push_back(d);
26                    helper(nums, eps, res);
27                    nums.pop_back();
28                }
29                nums.insert(nums.begin() + j, q);
30                nums.insert(nums.begin() + i, p);
31            }
32        }
33    }
34 };

```

来看一种很不同的递归写法，这里将加减乘除操作符放到了一个数组ops中。并且没有用全局变量res，而是让递归函数带有bool型返回值。在递归函数中，还是要先看nums数组的长度，如果为1了，说明已经计算完成，直接看结果是否为0就行了。然后遍历任意两个数字，注意这里的i和j都分别从0到了数组长度，而上面解法的j是从0到i，这是因为上面解法将 $p - q$, $q - p$, q / p , $q \cdot p$ 都分别列出来了，而这里仅仅是 $nums[i] - nums[j]$, $nums[i] / nums[j]$ ，所以i和j要交换位置，但是为了避免加法和乘法的重复计算，我们可以做个判断，还有别忘记了除数不为零的判断，i和j不能相同的判断。我们建立一个临时数组t，将非i和j位置的数字都加入t，然后遍历操作符数组ops，每次取出一个操作符，然后将 $nums[i]$ 和 $nums[j]$ 的计算结果加入t，调用递归函数，如果递归函数返回true了，那么就直接返回true。否则移除刚加入的结果，还原t的状态，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool judgePoint24(vector<int>& nums) {
4         double eps = 0.001;
5         vector<char> ops{'+', '-', '*', '/'};
6         vector<double> arr(nums.begin(), nums.end());
7         return helper(arr, ops, eps);
8     }
9     bool helper(vector<double>& nums, vector<char>& ops, double eps) {
10        if (nums.size() == 1) return abs(nums[0] - 24) < eps;
11        for (int i = 0; i < nums.size(); ++i) {
12            for (int j = 0; j < nums.size(); ++j) {
13                if (i == j) continue;
14                vector<double> t;
15                for (int k = 0; k < nums.size(); ++k) {
16                    if (k != i && k != j) t.push_back(nums[k]);
17                }
18                for (char op : ops) {
19                    if ((op == '+' || op == '*') && i > j) continue;
20                    if (op == '/' && nums[j] < eps) continue;
21                    switch(op) {
22                        case '+': t.push_back(nums[i] + nums[j]); break;
23                        case '-': t.push_back(nums[i] - nums[j]); break;
24                        case '*': t.push_back(nums[i] * nums[j]); break;
25                        case '/': t.push_back(nums[i] / nums[j]); break;
26                    }
27                    if (helper(t, ops, eps)) return true;
28                    t.pop_back();
29                }
30            }
31        }
32        return false;
33    }
34 };

```

668. 验证回文字符串之二

Given a non-empty string s, you may delete at most one character. Judge whether you can make it a palindrome.

这道题是之前那道Valid Palindrome的拓展，还是让我们验证回复字符串，但是区别是这道题的字符串中只含有小写字母，而且这道题允许删除一个字符，那么当遇到不匹配的时候，我们到底是删除左边的字符，还是右边的字符呢，我们的做法是两种情况都要算一遍，只要有一种能返回true，那么结果就返回true。我们可以写一个子函数来判断字符串中的某一个范围内的子字符串是否为回文串，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool validPalindrome(string s) {
4         int left = 0, right = s.size() - 1;
5         while (left < right) {
6             if (s[left] != s[right]) return isValid(s, left, right - 1) || isValid(s, left
7 + 1, right);
8             ++left; --right;
9         }
10        return true;
11    }
12    bool isValid(string s, int left, int right) {
13        while (left < right) {
14            if (s[left] != s[right]) return false;
15            ++left; --right;
16        }
17        return true;
18    }
19 };

```

下面这种写法跟上面的解法思路一样，只不过没有写额外的函数，还是要遍历两种情况，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool validPalindrome(string s) {
4         int left = 0, right = s.size() - 1;
5         while (left < right) {
6             if (s[left] == s[right]) {
7                 ++left; --right;
8             } else {
9                 int l = left, r = right - 1;
10                while (l < r) {
11                    if (s[l] != s[r]) break;
12                    ++l; --r;
13                    if (l >= r) return true;
14                }
15                ++left;
16                while (left < right) {
17                    if (s[left] != s[right]) return false;
18                    ++left; --right;
19                }
20            }
21        }
22        return true;
23    }
24 };

```

669. 下一个最近时间点

Given a time represented in the format "HH:MM", form the next closest time by reusing the current digits. There is no limit on how many times a digit can be reused.

You may assume the given input string is always valid. For example, "01:34", "12:09" are all valid. "1:34", "12:9" are all invalid.

这道题给了我们一个时间点，让我们求最近的下一个时间点，规定了不能产生新的数字，当这个时间点超过零点时，就当第二天的时间。为了找到下一个时间点，我们肯定是从分钟开始换数字，而且换的数字要是存在的数字，那么我们最先要做的就是统计当前时间点中的数字，由于可能有重复数字的存在，我们把数字都存入集合set中，这样可以去除重复数字，并且可以排序，然后再转为vector。下面就从低位分钟开始换数字了，如果低位分钟上的数字已经是最大的数字了，那么说明要转过一轮了，就要把低位分钟上的数字换成最小的那个数字；否则就将低位分钟上的数字换成下一个数字。然后再看高位分钟上的数字，同理，如果高位分钟上的数字已经是最大的数字，或则下一个数字大于5，那么直接换成最小值；否则就将高位分钟上的数字换成下一个数字。对于小时位上的数字也是同理，对于小时低位上的数字情况比较复杂，当小时高位不为2的时候，低位可以是任意数字，而当高位为2时，低位需要小于等于3。对于小时高位，其必须要小于等于2，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     string nextClosestTime(string time) {
4         string res = time;
5         set<int> s{time[0], time[1], time[3], time[4]};
6         string str(s.begin(), s.end());
7         for (int i = res.size() - 1; i >= 0; --i) {
8             if (res[i] == ':') continue;
9             int pos = str.find(res[i]);
10            if (pos == str.size() - 1) {
11                res[i] = str[0];
12            } else {
13                char next = str[pos + 1];
14                if (i == 4) {
15                    res[i] = next;
16                    return res;
17                } else if (i == 3 && next <= '5') {
18                    res[i] = next;
19                    return res;
20                } else if (i == 1 && (res[0] != '2' || (res[0] == '2' && next <= '3'))) {
21                    res[i] = next;
22                    return res;
23                } else if (i == 0 && next <= '2') {
24                    res[i] = next;
25                    return res;
26                }
27                res[i] = str[0];
28            }
29        }
30        return res;
31    }
32 };

```

CPP

下面这种方法的写法比较简洁，实际上用了暴力搜索，由于按分钟算的话，一天只有1440分钟，也就是1440个时间点，我们可以从当前时间点开始，遍历一天的时间，也就是接下来的1440个时间点，得到一个新的整型时间点后，我们按位来更新结果res，对于每个更新的数字字符，看其是否在原时间点字符中存在，如果不存在，直接break，然后开始遍历下一个时间点，如果四个数字都成功存在了，那么将当前时间点中间夹上冒号返回即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     string nextClosestTime(string time) {
4         string res = "0000";
5         vector<int> v{600, 60, 10, 1};
6         int found = time.find(":");
7         int cur = stoi(time.substr(0, found)) * 60 + stoi(time.substr(found + 1));
8         for (int i = 1, d = 0; i <= 1440; ++i) {
9             int next = (cur + i) % 1440;
10            for (d = 0; d < 4; ++d) {
11                res[d] = '0' + next / v[d];
12                next %= v[d];
13                if (time.find(res[d]) == string::npos) break;
14            }
15            if (d >= 4) break;
16        }
17        return res.substr(0, 2) + ":" + res.substr(2);
18    }
19 };

```

670. 棒球游戏

You're now a baseball game point recorder.

Given a list of strings, each string can be one of the 4 following types:

"Integer (one round's score)": Directly represents the number of points you get in this round.

"+" (one round's score): Represents that the points you get in this round are the sum of the last two valid round's points.

"D" (one round's score): Represents that the points you get in this round are the doubled data of the last valid round's points.

"C" (an operation, which isn't a round's score): Represents the last valid round's points you get were invalid and should be removed.

Each round's operation is permanent and could have an impact on the round before and the round after.

You need to return the sum of the points you could get in all the rounds.

这道题不是一道难题，直接按照题目的描述来分情况处理即可，博主开始在取数组的最后一个数和倒数第二个数的时候还做了数组为空检测，但是貌似这道题默认输入都是合法的，不会存在上一轮不存在还要取值的情况，那就不用检测啦，代码就更加的简洁啦：

```

1 class Solution {
2 public:
3     int calPoints(vector<string>& ops) {
4         vector<int> v;
5         for (string op : ops) {
6             if (op == "+") {
7                 v.push_back(v.back() + v[v.size() - 2]);
8             } else if (op == "D") {
9                 v.push_back(2 * v.back());
10            } else if (op == "C") {
11                v.pop_back();
12            } else {
13                v.push_back(stoi(op));
14            }
15        }
16        return accumulate(v.begin(), v.end(), 0);
17    }
18 };

```

671. K个空槽

There is a garden with N slots. In each slot, there is a flower. The N flowers will bloom one by one in N days. In each day, there will be exactly one flower blooming and it will be in the status of blooming since then.

Given an array flowers consists of number from 1 to N. Each number in the array represents the place where the flower will open in that day.

For example, flowers[i] = x means that the unique flower that blooms at day i will be at position x, where i and x will be in the range from 1 to N.

Also given an integer k, you need to output in which day there exists two flowers in the status of blooming, and also the number of flowers between them is k and these flowers are not blooming.

If there isn't such day, output -1.

这道题给了我们这样一个场景，说是花园里有N个空槽，可以放花，每天放一朵开着的花，而且一旦放了就会一直开下去。不是按顺序放花，而是给了我们一个数组flowers，其中flowers[i] = x表示第i天放的花会在位置x。其实题目这里有误，数组是从0开始的，而天数和位置都是从1开始的，所以正确的应该是第i+1天放的花会在位置x。然后给了我们一个整数k，让我们判断是否正好有两朵盛开的花中间有k个空槽，如果有，返回当前天数，否则返回-1。博主刚开始想的是先用暴力破解来做，用一个状态数组，如果该位置有花为1，无花为0，然后每增加一朵花，就遍历一下状态数组，找有没有连续k个0，结果TLE了。这说明，应该等所有花都放好了，再来找才行，但是这样仅用0和1的状态数组是不行的，我们得换个形式。

我们用一个days数组，其中days[i] = t表示在i+1位置上会在第t天放上花，那么如果days数组为[1 3 2]，就表示第一个位置会在第一天放上花，第二个位置在第三天放上花，第三个位置在第二天放上花。我们想，在之前的状态数组中，0表示没放花，1表示放了花，而days数组中的数字表示放花的天数，那么就是说数字大的就是花放的时间晚，那么在当前时间i，所有大于i的是不是也就是可以看作是没放花呢，这样问题就迎刃而解了，我们来找一个k+2大小的子数组，除了首尾两个数字，中间的k个数字都要大于首尾两个数字即可，那么首尾两个数字中较大的数就是当前的天数。left和right是这个大小为k+2的窗口，初始化时left为0，right为k+1，然后i从0开始遍历，这里循环的条件是right小于n，当窗口的右边界越界后，循环自然需要停止。如果当days[i]小于days[left]，或者days[i]小于等于days[right]的时候，有两种情况，一种是i在[left, right]范围内，说明窗口中有数字小于边界数字，这不满足我们之前限定的条件，至于days[i]为何可以等于days[right]，是因为当i遍历到right到位置时，说明中间的数字都是大于左右边界数的，此时我们要用左右边界中较大的那个数字更新结果res。不管i是否等于right，只要进了这个if

条件，说明当前窗口要么是不合题意，要么是遍历完了，我们此时要重新给left和right赋值，其中left赋值为i，right赋值为k+1+i，还是大小为k+2的窗口，继续检测。最后我们看结果res，如果还是INT_MAX，说明无法找到，返回-1即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int kEmptySlots(vector<int>& flowers, int k) {
4         int res = INT_MAX, left = 0, right = k + 1, n = flowers.size();
5         vector<int> days(n, 0);
6         for (int i = 0; i < n; ++i) days[flowers[i] - 1] = i + 1;
7         for (int i = 0; right < n; ++i) {
8             if (days[i] < days[left] || days[i] <= days[right]) {
9                 if (i == right) res = min(res, max(days[left], days[right]));
10                left = i;
11                right = k + 1 + i;
12            }
13        }
14     return (res == INT_MAX) ? -1 : res;
15 }
16 };

```

下面这种方法用到了TreeSet来做，利用其自动排序的特点，然后用lower_bound和upper_bound进行快速的二分查找。题目中的 $flowers[i] = x$ 表示第*i*+1天放的花会在位置x。所以我们遍历flowers数组，其实就是按照时间顺序进行的。我们取出当前需要放置的位置cur，然后在集合set中查找第一个大于cur的数字，如果存在的话，说明两者中间点位置都没有放花，而如果中间正好有k个空位的话，那么当前天数就即为所求。这是当cur为左边界的情况，同样，我们可以把cur当右边界来检测，在集合set中查找第一个小于cur的数字，如果二者中间有k个空位，也返回当前天数。需要注意的是，C和Java中的upper_bound和higher是相同作用的，但是lower_bound和lower却不太一样。C中的lower_bound找的是第一个不小于目标值的数字，所以可能会返回和目标值相同或者大于目标值的数字。只要这个数字不是第一个数字，然后我们往前退一位，就是要求的第一个小于目标值的数字，这相当于Java中的lower函数，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int kEmptySlots(vector<int>& flowers, int k) {
4         set<int> s;
5         for (int i = 0; i < flowers.size(); ++i) {
6             int cur = flowers[i];
7             auto it = s.upper_bound(cur);
8             if (it != s.end() && *it - cur == k + 1) {
9                 return i + 1;
10            }
11            it = s.lower_bound(cur);
12            if (it != s.begin() && cur - *(--it) == k + 1) {
13                return i + 1;
14            }
15            s.insert(cur);
16        }
17     return -1;
18 }
19 };

```

672. 冗余的连接

In this problem, a tree is an undirected graph that is connected and has no cycles.

The given input is a graph that started as a tree with N nodes (with distinct values 1, 2, ..., N), with one additional edge added. The added edge has two different vertices chosen from 1 to N, and was not an edge that already existed.

The resulting graph is given as a 2D-array of edges. Each element of edges is a pair [u, v] with u < v, that represents an undirected edge connecting nodes u and v.

Return an edge that can be removed so that the resulting graph is a tree of N nodes. If there are multiple answers, return the answer that occurs last in the given 2D-array. The answer edge [u, v] should be in the same format, with u < v.

这道题给我们了一个无向图，让我们删掉组成环的最后一条边，其实这道题跟之前那道Graph Valid Tree基本没什么区别，三种解法都基本相同。博主觉得老题稍微变一下就是一道新题，而onsite遇到原题的概率很小，大多情况下都会稍稍变一下，所以举一反三的能力真的很重要，要完全吃透一道题也不太容易，需要多下功夫。我们首先来看递归的解法，这种解法的思路是，每加入一条边，就进行环检测，一旦发现了环，就返回当前边。对于无向图，我们还是用邻接表来保存，建立每个结点和其所有邻接点的映射，由于两个结点之间不算有环，所以我们要避免这种情况 $1 \rightarrow \{2\}$, $2 \rightarrow \{1\}$ 的死循环，所以我们用一个变量pre记录上一次递归的结点，比如上一次遍历的是结点1，那么在遍历结点2的邻接表时，就不会再次进入结点1了，这样有效的避免了死循环，使其能返回正确的结果，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     vector<int> findRedundantConnection(vector<vector<int>>& edges) {
4         unordered_map<int, unordered_set<int>> m;
5         for (auto edge : edges) {
6             if (hasCycle(edge[0], edge[1], m, -1)) return edge;
7             m[edge[0]].insert(edge[1]);
8             m[edge[1]].insert(edge[0]);
9         }
10        return {};
11    }
12    bool hasCycle(int cur, int target, unordered_map<int, unordered_set<int>>& m, int pre)
13    {
14        if (m[cur].count(target)) return true;
15        for (int num : m[cur]) {
16            if (num == pre) continue;
17            if (hasCycle(num, target, m, cur)) return true;
18        }
19        return false;
20    }
21 };

```

CPP

既然递归能做，一般来说迭代也木有问题。但是由于BFS的遍历机制和DFS不同，所以没法采用利用变量pre来避免上面所说的死循环(不是很确定，可能是博主没想出来，有做出来的请在评论区贴上代码)，所以我们采用一个集合来记录遍历过的结点，如果该结点已经遍历过了，那么直接跳过即可，否则我们就把该结点加入queue和集合，继续循环，参见代码如下：

解法2:

```

1 class Solution {
2 public:
3     vector<int> findRedundantConnection(vector<vector<int>>& edges) {
4         unordered_map<int, unordered_set<int>> m;
5         for (auto edge : edges) {
6             queue<int> q{{edge[0]}};
7             unordered_set<int> s{{edge[0]}};
8             while (!q.empty()) {
9                 auto t = q.front(); q.pop();
10                if (m[t].count(edge[1])) return edge;
11                for (int num : m[t]) {
12                    if (s.count(num)) continue;
13                    q.push(num);
14                    s.insert(num);
15                }
16            }
17            m[edge[0]].insert(edge[1]);
18            m[edge[1]].insert(edge[0]);
19        }
20        return {};
21    }
22 };

```

其实这道题最好的解法使用Union Find来做，论坛上清一色的都是用这种解法来做的，像博主用DFS和BFS这么清新脱俗的方法还真不多:) 其实Union Find的核心思想并不是很难理解，首先我们建立一个长度为(n+1)的数组root，由于这道题并没有明确的说明n是多少，只是说了输入的二维数组的长度不超过1000，那么n绝对不会超过2000，我们加1的原因是由于结点值是从1开始的，而数组是从0开始的，我们懒得转换了，就多加一位得了。我们将这个数组都初始化为-1，有些人喜欢初始化为1，都可以。开始表示每个结点都是一个单独的组，所谓的Union Find就是要让结点之间建立关联，比如若root[1] = 2，就表示结点1和结点2是相连的，root[2] = 3表示结点2和结点3是相连的，如果我们此时新加一条边[1, 3]的话，我们通过root[1]得到2，再通过root[2]得到3，说明结点1有另一条路径能到结点3，这样就说明环是存在的；如果没有这条路径，那么我们要将结点1和结点3关联起来，让root[1] = 3即可，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     vector<int> findRedundantConnection(vector<vector<int>>& edges) {
4         vector<int> root(2001, -1);
5         for (auto edge : edges) {
6             int x = find(root, edge[0]), y = find(root, edge[1]);
7             if (x == y) return edge;
8             root[x] = y;
9         }
10        return {};
11    }
12    int find(vector<int>& root, int i) {
13        while (root[i] != -1) {
14            i = root[i];
15        }
16        return i;
17    }
18 };

```

673. 重复字符串匹配

Given two strings A and B, find the minimum number of times A has to be repeated such that B is a substring of it. If no such solution, return -1.

For example, with A = "abcd" and B = "cdabcdab".

Return 3, because by repeating A three times ("abcdabcdabcd"), B is a substring of it; and B is not a substring of A repeated two times ("abcdabcd").

Note:

The length of A and B will be between 1 and 10000.

这道题让我们用字符串B来匹配字符串A，问字符串A需要重复几次，如果无法匹配，则返回-1。那么B要能成为A的字符串，那么A的长度肯定要大于等于B，所以当A的长度小于B的时候，我们可以先进行重复A，直到A的长度大于等于B，并且累计次数cnt。那么此时我们用find来找，看B是否存在A中，如果存在直接返回cnt。如果不存在，我们再加上一个A，再来找，这样可以处理这种情况A="abc", B="cab"，如果此时还找不到，说明无法匹配，返回-1，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     int repeatedStringMatch(string A, string B) {
4         int n1 = A.size(), n2 = B.size(), cnt = 1;
5         string t = A;
6         while (t.size() < n2) {
7             t += A;
8             ++cnt;
9         }
10        if (t.find(B) != string::npos) return cnt;
11        t += A;
12        return (t.find(B) != string::npos) ? cnt + 1 : -1;
13    }
14};
```

CPP

下面这种解法就更简洁了，思路和上面的一样，都是每次给t增加一个字符串A，我们其实可以直接算出最多需要增加的个数，即B的长度除以A的长度再加上2，当B小于A的时候，那么可能需要两个A，所以i就是小于等于2，这样我们每次在t中找B，如果找到了，就返回i，没找到，就增加一个A，循环推出后返回-1即可，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     int repeatedStringMatch(string A, string B) {
4         string t = A;
5         for (int i = 1; i <= B.size() / A.size() + 2; ++i) {
6             if (t.find(B) != string::npos) return i;
7             t += A;
8         }
9         return -1;
10    }
11};
```

CPP

下面这种解法还是由热心网友edyyy提供，没有用到字符串自带的find函数，而是逐个字符进行比较，循环字符串A中的所有字符，然后用个变量j，初始化为0，用来循环字符串B中的字符，每个字符和A中对应的字符进行比较，此时从A中取字符就要把A当作一个循环数组来处理，用(i+j)%m来取字符，还要确保j小于n，避免越界，如果字符匹配的话，j自增1。while 循环结束

后，如果j等于n了，说明B中所有的字符都成功匹配了，那么我们来计算A的重复次数，通过 $(i+j-1)/m + 1$ 来得到，注意i+j要减1再除以m，之后再加上一次。因为当i+j正好等于m时，说明此时不用重复A，那么 $(i+j-1)/m + 1$ 还是等于1，当i+j>m的时候，需要重复A了， $(i+j-1)/m + 1$ 也可以得到正确的结构，参见代码如下：

解法3：

```
1 class Solution {
2 public:
3     int repeatedStringMatch(string A, string B) {
4         int m = A.size(), n = B.size();
5         for (int i = 0; i < m; ++i) {
6             int j = 0;
7             while (j < n && A[(i + j) % m] == B[j]) ++j;
8             if (j == n) return (i + j - 1) / m + 1;
9         }
10    return -1;
11 }
12 };
```

674. 最长相同值路径

Given a binary tree, find the length of the longest path where each node in the path has the same value. This path may or may not pass through the root.

Note: The length of path between two nodes is represented by the number of edges between them.

这道题让我们求最长的相同值路径，跟之前那道Count Univalued Subtrees十分的类似，解法也很类似。对于这种树的路径问题，递归是不二之选。在递归函数中，我们首先对其左右子结点调用递归函数，得到其左右子树的最大相同值路径，下面就要来看当前结点和其左右子结点之间的关系了，如果其左子结点存在且和当前节点值相同，则left自增1，否则left重置0；同理，如果其右子结点存在且和当前节点值相同，则right自增1，否则right重置0。然后用left+right来更新结果res。而调用当前节点值的函数只能返回left和right中的较大值，因为如果还要跟父节点组path，就只能在左右子节点中选一条path，当然选值大的那个了，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     int longestUnivaluedPath(TreeNode* root) {
4         if (!root) return 0;
5         int res = 0;
6         helper(root, root, res);
7         return res;
8     }
9     int helper(TreeNode* node, TreeNode* parent, int& res) {
10        if (!node) return 0;
11        int left = helper(node->left, node, res);
12        int right = helper(node->right, node, res);
13        left = (node->left && node->val == node->left->val) ? left + 1 : 0;
14        right = (node->right && node->val == node->right->val) ? right + 1 : 0;
15        res = max(res, left + right);
16        return max(left, right);
17    }
18 };
```

下面这种解法使用了两个递归函数，使得写法更加简洁了，首先还是先判断root是否为空，是的话返回0。然后对左右子节点分别调用当前函数，取其中较大值保存到变量sub中，表示左右子树中最长的相同值路径，然后就是要跟当前树的最长相同值路径比较，计算方法是对左右子结点调用一个helper函数，并把当前结点值传进去，把返回值加起来和sub比较，去较大值返回。在helper函数里，当当前结点为空，或者当前节点值不等于父结点值的话，返回0。否则结返回对左右子结点分别调用helper递归函数中的较大值加1，我们发现这种写法跟求树的最大深度很像，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int longestUnivalPath(TreeNode* root) {
4         if (!root) return 0;
5         int sub = max(longestUnivalPath(root->left), longestUnivalPath(root->right));
6         return max(sub, helper(root->left, root->val) + helper(root->right, root->val));
7     }
8     int helper(TreeNode* node, int parent) {
9         if (!node || node->val != parent) return 0;
10        return 1 + max(helper(node->left, node->val), helper(node->right, node->val));
11    }
12 };

```

675. 棋盘上骑士的可能性

On an NxN chessboard, a knight starts at the r-th row and c-th column and attempts to make exactly K moves. The rows and columns are 0 indexed, so the top-left square is (0, 0), and the bottom-right square is (N-1, N-1).

A chess knight has 8 possible moves it can make, as illustrated below. Each move is two squares in a cardinal direction, then one square in an orthogonal direction.

Each time the knight is to move, it chooses one of eight possible moves uniformly at random (even if the piece would go off the chessboard) and moves there.

The knight continues moving until it has made exactly K moves or has moved off the chessboard. Return the probability that the knight remains on the board after it has stopped moving.

这道题给了我们一个大小为NxN国际象棋棋盘，上面有个骑士，相当于我们中国象棋中的马，能走‘日’字，给了我们一个起始位置，然后说允许我们走K步，问走完K步之后还能留在棋盘上的概率是多少。那么要求概率，我们必须要先分别求出分子和分母，其中分子是走完K步还在棋盘上的走法，分母是没有限制条件的总共的走法。那么分母最好算，每步走有8种跳法，那么K步就是8的K次方种了。关键是要求出分子，博主开始向的方法是以给定位置为起始点，然后进行BFS，每步遍历8种情况，遇到在棋盘上的就计数器加1，结果TLE了。上论坛看大家的解法，结果发现都是换了一个角度来解决问题的，并不很关心骑士的起始位置，而是把棋盘上所有位置上经过K步还留在棋盘上的走法总和都算出来，那么最后直接返回需要的值即可。跟之前那道Out of Boundary Paths没啥本质上的区别，又是换了一个马甲就不会了系列。还是要用DP来做，我们可以用三维DP数组，也可以用二维DP数组来做，这里为了省空间，我们就用二维DP数组来做，其中dp[i][j]表示在棋盘(i, j)位置上走完当前步骤还留在棋盘上的走法总和，初始化为1，我们其实将步骤这个维度当成了时间维度在不停更新。好，下面我们先写出8种‘日’字走法的位置的坐标，就像之前遍历迷宫上下左右四个方向坐标一样，这不过这次位置变了而已。然后我们一步一步来遍历，每一步都需要完整遍历一遍棋盘的每个位置，新建一个临时数组t，大小和dp数组相同，但是初始化为0，然后对于遍历到的棋盘上的每一个格子，我们都遍历8中解法，如果新的位置不在棋盘上了，直接跳过，否则就加上上一步中的dp数组中对应的值，遍历完棋盘后，将dp数组更新为这个临时数组t，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     double knightProbability(int N, int K, int r, int c) {
4         if (K == 0) return 1;
5         vector<vector<double>> dp(N, vector<double>(N, 1));
6         vector<vector<int>> dirs{{-1,-2},{-2,-1},{-2,1},{-1,2},{1,2},{2,1},{2,-1},{1,-2}};
7         for (int m = 0; m < K; ++m) {
8             vector<vector<double>> t(N, vector<double>(N, 0));
9             for (int i = 0; i < N; ++i) {
10                 for (int j = 0; j < N; ++j) {
11                     for (auto dir : dirs) {
12                         int x = i + dir[0], y = j + dir[1];
13                         if (x < 0 || x >= N || y < 0 || y >= N) continue;
14                         t[i][j] += dp[x][y];
15                     }
16                 }
17             }
18             dp = t;
19         }
20         return dp[r][c] / pow(8, K);
21     }
22 };

```

我们也可以使用有memo数组优化的递归来做，避免重复运算，从而能通过OJ。递归下的memo数组其实就是迭代下的dp数组，这里我们用三维的数组，初始化为0。在递归函数中，如果k为0了，说明已经走了k步，返回 1。如果memo[k][r][c]不为0，说明这种情况之前已经计算过，直接返回。然后遍历8种走法，计算新的位置，如果不在棋盘上就跳过；然后更新memo[k][r][c]，使其加上对新位置调用递归的返回值，注意此时带入k-1和新的位置，退出循环后返回memo[k][r][c]即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<vector<int>> dirs{{-1,-2},{-2,-1},{-2,1},{-1,2},{1,2},{2,1},{2,-1},{1,-2}};
4     double knightProbability(int N, int K, int r, int c) {
5         vector<vector<vector<double>>> memo(K + 1, vector<vector<double>>(N, vector<double>(N, 0.0)));
6         return helper(memo, N, K, r, c) / pow(8, K);
7     }
8     double helper(vector<vector<vector<double>>>& memo, int N, int k, int r, int c) {
9         if (k == 0) return 1.0;
10        if (memo[k][r][c] != 0.0) return memo[k][r][c];
11        for (auto dir : dirs) {
12            int x = r + dir[0], y = c + dir[1];
13            if (x < 0 || x >= N || y < 0 || y >= N) continue;
14            memo[k][r][c] += helper(memo, N, k - 1, x, y);
15        }
16        return memo[k][r][c];
17    }
18 };

```

676. 三个非重叠子数组的最大和

In a given array nums of positive integers, find three non-overlapping subarrays with maximum sum.

Each subarray will be of size k, and we want to maximize the sum of all 3*k entries.

Return the result as a list of indices representing the starting position of each interval (0-indexed). If there are multiple answers, return the lexicographically smallest one.

这道题给了我们一个只包含正数的数组，让我们找三个长度为k的不重叠的子数组，使得所有子数组的数字之和最大。首先我们应该明确的是，暴力搜索在这道题上基本不太可能，因为遍历一个子数组的复杂度是平方级，遍历三个还得六次方啊，看OJ不削你～那么我们只能另辟蹊径，对于这种求子数组和有关的题目时，一般都需要建立累加和数组，为啥呢，因为累加和数组可以快速的求出任意长度的子数组之和，当然也能快速的求出长度为k的子数组之和。因为这道题只让我们找出三个子数组，那么我们可以先确定中间那个子数组的位置，这样左右两边的子数组的位置范围就缩小了，中间子数组的起点不能是从开头到结尾整个区间，必须要在首尾各留出k个位置给其他两个数组。一旦中间子数组的起始位置确定了，那么其和就能通过累加和数组快速确定。那么现在就要在左右两边的区间内分别找出和最大的子数组，遍历所有的子数组显然不是很高效，如何快速求出呢，这里我们需要使用动态规划Dynamic Programming的思想来维护两个DP数组left和right，其中：

left[i]表示在区间[0, i]范围内长度为k且和最大的子数组的起始位置

right[i]表示在区间[i, n - 1]范围内长度为k且和最大的子数组的起始位置

这两个dp数组各需要一个for循环来更新，left数组都初始化为0，前k个数字没办法，肯定起点都是0，变量total初始化为前k个数字之和，然后从第k+1个数字开始，每次向前取k个，利用累加和数组sums快速算出数字之和，跟total比较，如果大于total的话，那么更新total和left数组当前位置值，否则的话left数组的当前值就赋值为前一位的值。同理对right数组的更新也类似，total初始化为最后k个数字之和，然后从前一个数字向前遍历，如果大于total，更新total和right数组的当前位置，否则right数组的当前值就赋值为后一位的值。一旦left数组和right数组都更新好了，那么就可以遍历中间子数组的起始位置了，然后我们可以通过left和right数组快速定位出左边和右边的最大子数组的起始位置，并快速计算出这三个子数组的所有数字之和，用来更新全局最大值mx，如果mx被更新了的话，记录此时的三个子数组的起始位置到结果res中，参见代码如下：

```

1 class Solution {
2 public:
3     vector<int> maxSumOfThreeSubarrays(vector<int>& nums, int k) {
4         int n = nums.size(), mx = INT_MIN;
5         vector<int> sums{0}, res, left(n, 0), right(n, n - k);
6         for (int num : nums) sums.push_back(sums.back() + num);
7         for (int i = k, total = sums[k] - sums[0]; i < n; ++i) {
8             if (sums[i + 1] - sums[i + 1 - k] > total) {
9                 left[i] = i + 1 - k;
10                total = sums[i + 1] - sums[i + 1 - k];
11            } else {
12                left[i] = left[i - 1];
13            }
14        }
15        for (int i = n - 1 - k, total = sums[n] - sums[n - k]; i >= 0; --i) {
16            if (sums[i + k] - sums[i] >= total) {
17                right[i] = i;
18                total = sums[i + k] - sums[i];
19            } else {
20                right[i] = right[i + 1];
21            }
22        }
23        for (int i = k; i <= n - 2 * k; ++i) {
24            int l = left[i - 1], r = right[i + k];
25            int total = (sums[i + k] - sums[i]) + (sums[l + k] - sums[l]) + (sums[r + k] -
26 sums[r]);
27            if (mx < total) {
28                mx = total;
29                res = {l, i, r};
30            }
31        }
32        return res;
33    }
34 };

```

677. 员工重要度

You are given a data structure of employee information, which includes the employee's unique id, his importance value and his direct subordinates' id.

For example, employee 1 is the leader of employee 2, and employee 2 is the leader of employee 3. They have importance value 15, 10 and 5, respectively. Then employee 1 has a data structure like [1, 15, [2]], and employee 2 has [2, 10, [3]], and employee 3 has [3, 5, []]. Note that although employee 3 is also a subordinate of employee 1, the relationship is not direct.

Now given the employee information of a company, and an employee id, you need to return the total importance value of this employee and all his subordinates.

这道题定义了一种员工类，有id，重要度，和direct report的员工，让我们求某个员工的总重要度。我们要明白的是就算某个员工不直接向你汇报工作，而是向你手下人汇报，这个人的重要度也会算进你的总重要度中。这其实就是之前那道Nested List Weight Sum的变化形式，我们可以用DFS来做。首先我们想，为了快速的通过id来定位到员工类，需要建立一个id和员工类的映射，然后我们根据给定的员工id来算其重要度。计算方法当然是其本身的重要度加上其所有手下人的总重要度，对于手下人，还要累加其手下人的总重要度，为了不重复计算某个员工的重要度，我们建立一个集合，将遍历过的员工id放到集合中，这样一旦我们遍历到集合中有的员工，直接返回0即可；否则就将该员工id加入集合中，然后建立一个结果res变量，加上当前员工的重要度，然后遍历其所有手下，对其每个手下人调用递归函数加到res上，最后返回res即可，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     int getImportance(vector<Employee*> employees, int id) {
4         unordered_set<int> s;
5         unordered_map<int, Employee*> m;
6         for (auto e : employees) m[e->id] = e;
7         return helper(id, m, s);
8     }
9     int helper(int id, unordered_map<int, Employee*>& m, unordered_set<int>& s) {
10        if (s.count(id)) return 0;
11        s.insert(id);
12        int res = m[id]->importance;
13        for (int num : m[id]->subordinates) {
14            res += helper(num, m, s);
15        }
16        return res;
17    }
18 };

```

CPP

我们也可以用BFS来做，使用一个queue来辅助运算，开始将给定员工id放入，然后当queue不为空进行循环，每次取出队首员工，如果已经访问过了，直接跳过，否则加入集合中，然后累加上当前员工的复杂度到结果res，然后将其所有手下人加入队列等待遍历，参见代码如下：

解法2:

```

1 class Solution {
2 public:
3     int getImportance(vector<Employee*> employees, int id) {
4         int res = 0;
5         queue<int> q{{id}};
6         unordered_set<int> s;
7         unordered_map<int, Employee*> m;
8         for (auto e : employees) m[e->id] = e;
9         while (!q.empty()) {
10            auto t = q.front(); q.pop();
11            if (s.count(t)) continue;
12            s.insert(t);
13            res += m[t]->importance;
14            for (int num : m[t]->subordinates) {
15                q.push(num);
16            }
17        }
18        return res;
19    }
20 };

```

CPP

678. 贴片拼单词

We are given N different types of stickers. Each sticker has a lowercase English word on it.

You would like to spell out the given target string by cutting individual letters from your collection of stickers and rearranging them.

You can use each sticker more than once if you want, and you have infinite quantities of each sticker.

What is the minimum number of stickers that you need to spell out the target? If the task is impossible, return -1.

这道题给了我们N个贴片，每个贴片上有一个小写字母的单词，给了我们一个目标单词target，让我们通过剪下贴片单词上的字母来拼出目标值，每个贴片都有无数个，问我们最少用几个贴片能拼出目标值target，如果不能拼出来的话，就返回-1。这道题博主最开始尝试用贪婪算法，结果发现不行，有网友留言提示说是多重背包问题，然后去论坛上看大神们的解法，果然都是用DP做的，之前曾有网友推荐过一个“背包九讲”的帖子，大概扫过几眼，真是叼到飞起啊，博主希望有时间也能总结一下。先来看这道题吧，既然是用DP来做，那么就需要用dp数组了，我们使用一个一维的dp数组，其中dp[i]表示组成第i个子集合所需要的最少的sticker的个数，注意这里是子集合，而不是子串。长度为n的字符串共有 2^n 个子集合，比如字符串"ab"，就有4个子集合，分别是 "", "a", "b", "ab"。字符串"abc"就有8个子集合，如果我们用0到7来分别对应其子集合，就有：

复制代码

	abc	subset
0	000	" "
1	001	c
2	010	b
3	011	bc
4	100	a
5	101	ac
6	110	bb
7	111	abc

复制代码

我们可以看到0到7的二进制数的每一位上的0和1就相当于mask，是1的话就选上该位对应的字母，000就表示都不选，就是空集，111就表示都选，就是abc，那么只要我们将所有子集合的dp值都算出来，最后返回dp数组的最后一个位置上的数字，就是和目标子串相等的子集合啦。我们以下面这个简单的例子来讲解：

```
stickers = ["ab", "bc", "ac"], target = "abc"
```

之前说了abc的共有8个子集合，所以dp数组的长度为8，除了dp[0]初始化为0之外，其余的都初始化为INT_MAX，然后我们开始逐个更新dp数组的值，我们的目标是从sticker中取出字符，来拼出子集合，所以如果当前遍历到的dp值仍为INT_MAX的话，说明该子集合无法被拼出来，自然我们也无法再其基础上去拼别的子集合了，直接跳过。否则我们就来遍历所有的sticker，让变量cur等于i，说明此时是在第i个子集合的基础上去reach其他的子集合，我们遍历当前sticker的每一个字母，对于sticker的每一个字母，我们都扫描一遍target的所有字符，如果target里有这个字符，且该字符未出现在当前cur位置的子集合中，则将该字符加入子集合中。什么意思呢，比如当前我们的cur是3，二进制为011，对应的子集合是"bc"，此时如果我们遍历到的sticker是"ab"，那么遇到"a"时，我们知道target中是有"a"的，然后我们看"bc"中包不包括"a"，判断方法是看 $(cur >> k) \& 1$ 是否为0，这可以乍看上去不太好理解，其实不难想，因为我们的子集合是跟二进制对应的，"bc"就对应着011，第一个0就表示"a"缺失，所以我们想看哪个字符，就提取出该字符对应的二进制位，提取方法就是 $cur >> k$ ，表示cur向右移动k位，懂位操作Bit Manipulation的童鞋应该不难理解，提出出来的值与上1就知道该位是0还是1了，如果是0，表示缺失，那么把该位变为1，通过 $cur |= 1 << k$ 来实现，那么此时我们的cur就变位7，二进制为111，对应的子集合是"abc"，更新此时的dp[cur]为 dp[cur] 和 dp[i] + 1 中的较大值即可，最后循环结束后，如果"abc"对应的dp值还是INT_MAX，就返回-1，否则返回其dp值，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int minStickers(vector<string>& stickers, string target) {
4         int n = target.size(), m = 1 << n;
5         vector<int> dp(m, INT_MAX);
6         dp[0] = 0;
7         for (int i = 0; i < m; ++i) {
8             if (dp[i] == INT_MAX) continue;
9             for (string &sticker : stickers) {
10                 int cur = i;
11                 for (char c : sticker) {
12                     for (int k = 0; k < n; ++k) {
13                         if (target[k] == c && !((cur >> k) & 1)) {
14                             cur |= 1 << k;
15                             break;
16                         }
17                     }
18                 }
19                 dp[cur] = min(dp[cur], dp[i] + 1);
20             }
21         }
22         return dp[m - 1] == INT_MAX ? -1 : dp[m - 1];
23     }
24 };

```

下面这种解法是带记忆数组memo的递归解法，可以看作是DP解法的递归形式，核心思想都一样。只不过dp数组换成了哈希Map，建立子集合跟最小使用的sticker的个数之间的映射，初始化空集为0，我们首先统计每个sticker的各个字母出现的频率，放到对应的二维数组freq中，然后就是调用递归函数。在递归函数中，首先判断，如果target已经在memo中，直接返回其值。否则我们开始计算，首先统计出此时的target字符串的各个字母出现次数，然后我们遍历统计所有sticker中各个字母出现次数的数组freq，如果target字符串的第一个字母不在当前sticker中，我们直接跳过，注意递归函数中的target字符串不是原始的字符串，我们创建一个临时字符串t，然后我们遍历target字符串中存在的字符，如果target中的某字符存在的个数多于sticker中对应的字符，那么将多余的部分存在字符串t中，表示当前sticker无法拼出的字符，交给下一个递归函数来处理，我们看再次调用递归函数的结果ans，如果不为-1，说明可以拼出剩余的那些字符，那么此时我们的res更新为ans+1，循环退出后，此时我们的res就应该是组成当前递归函数中的target串的最少贴片数，更新dp[target]值，如果res是INT_MAX，说明无法拼出，更新为-1，否则更新为res，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int minStickers(vector<string>& stickers, string target) {
4         int N = stickers.size();
5         vector<vector<int>> freq(N, vector<int>(26, 0));
6         unordered_map<string, int> memo;
7         memo[""] = 0;
8         for (int i = 0; i < N; ++i) {
9             for (char c : stickers[i]) ++freq[i][c - 'a'];
10        }
11        return helper(freq, target, memo);
12    }
13    int helper(vector<vector<int>>& freq, string target, unordered_map<string, int>& memo)
14    {
15        if (memo.count(target)) return memo[target];
16        int res = INT_MAX, N = freq.size();
17        vector<int> cnt(26, 0);
18        for (char c : target) ++cnt[c - 'a'];
19        for (int i = 0; i < N; ++i) {
20            if (freq[i][target[0] - 'a'] == 0) continue;
21            string t = "";
22            for (int j = 0; j < 26; ++j) {
23                if (cnt[j] - freq[i][j] > 0) t += string(cnt[j] - freq[i][j], 'a' + j);
24            }
25            int ans = helper(freq, t, memo);
26            if (ans != -1) res = min(res, ans + 1);
27        }
28        memo[target] = (res == INT_MAX) ? -1 : res;
29        return memo[target];
30    }
31 };

```

679. 前K个高频词

Given a non-empty list of words, return the k most frequent elements.

Your answer should be sorted by frequency from highest to lowest. If two words have the same frequency, then the word with the lower alphabetical order comes first.

这道题让我们求前K个高频词，跟之前那道题Top K Frequent Elements极其类似，换了个数据类型就又是一道新题。唯一的不同就是之前那道题对于出现频率相同的数字，没有顺序要求。而这道题对于出现频率相同的单词，需要按照字母顺序来排。但是解法都一样，还是用最大堆和桶排序的方法。首先来看最大堆的方法，思路是先建立每个单词和其出现次数之间的映射，然后把单词和频率的pair放进最大堆，如果没有相同频率的单词排序要求，我们完全可以让频率当作pair的第一项，这样priority_queue默认是以pair的第一项为key进行从大到小的排序，而当第一项相等时，又会以第二项由大到小进行排序，这样就与题目要求的相同频率的单词要按字母顺序排列不相符，当然我们可以在存入结果res时对相同频率的词进行重新排序处理，也可以对priority_queue的排序机制进行自定义，这里我们采用第二种方法，我们自定义排序机制，我们让a.second > b.second，让小频率的词在第一位，然后当a.second == b.second时，我们让a.first < b.first，这是让字母顺序大的排在前面（这里博主需要强调一点的是，priority_queue的排序机制的写法和vector的sort的排序机制的写法正好顺序相反，同样的写法，用在sort里面就是频率小的在前面，不信的话可以自己试一下）。定义好最小堆后，我们首先统计单词的出现频率，然后组成pair排序最小堆之中，我们只保存k个pair，超过了就把队首的pair移除队列，最后我们把单词放入结果res中即可，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     vector<string> topKFrequent(vector<string>& words, int k) {
4         vector<string> res(k);
5         unordered_map<string, int> freq;
6         auto cmp = [] (pair<string, int>& a, pair<string, int>& b) {
7             return a.second > b.second || (a.second == b.second && a.first < b.first);
8         };
9         priority_queue<pair<string, int>, vector<pair<string, int>>, decltype(cmp) >
10        q(cmp);
11         for (auto word : words) ++freq[word];
12         for (auto f : freq) {
13             q.push(f);
14             if (q.size() > k) q.pop();
15         }
16         for (int i = res.size() - 1; i >= 0; --i) {
17             res[i] = q.top().first; q.pop();
18         }
19         return res;
20     }
21 };

```

下面这种解法还是一种堆排序的思路，这里我们用map，来建立次数和出现该次数所有单词的集合set之间的映射，这里也利用了set能自动排序的特性，当然我们还是需要首先建立每个单词和其出现次数的映射，然后将其组成pair放入map中，map是从小到大排序的，这样我们从最后面取pair，就是次数最大的，每次取出一层中所有的单词，如果此时的k大于该层的单词个数，就将整层的单词加入结果res中，否则就取前K个就行了，取完要更新K值，如果K小于等于0了，就break掉，返回结果res即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<string> topKFrequent(vector<string>& words, int k) {
4         vector<string> res;
5         unordered_map<string, int> freq;
6         map<int, set<string>> m;
7         for (string word : words) ++freq[word];
8         for (auto a : freq) {
9             m[a.second].insert(a.first);
10        }
11        for (auto it = m.rbegin(); it != m.rend(); ++it) {
12            if (k <= 0) break;
13            auto t = it->second;
14            vector<string> v(t.begin(), t.end());
15            if (k >= t.size()) {
16                res.insert(res.end(), v.begin(), v.end());
17            } else {
18                res.insert(res.end(), v.begin(), v.begin() + k);
19            }
20            k -= t.size();
21        }
22        return res;
23    }
24};

```

下面这种解法是一种桶排序的思路，我们根据出现次数建立多个bucket，桶的个数不会超过单词的个数，在每个桶中，我们对单词按字符顺序进行排序。我们可以用个数组来表示桶，每一层中放一个集合，利用set的自动排序的功能，使其能按字母顺序排列。我们还是需要首先建立每个单词和其出现次数的映射，然后将其组成pair放入map种，map是从小到大排序的，这样我们倒序遍历所有的桶，这样取pair，就是次数最大的，每次取出一层中所有的单词，如果此时的k大于该层的单词个数，就将整层的单词加入结果res中，否则就取前K个就行了，取完要更新K值，如果K小于等于0了，就break掉，返回结果res即可，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     vector<string> topKFrequent(vector<string>& words, int k) {
4         vector<string> res;
5         unordered_map<string, int> freq;
6         vector<set<string>> v(words.size() + 1, set<string>());
7         for (string word : words) ++freq[word];
8         for (auto a : freq) {
9             v[a.second].insert(a.first);
10        }
11        for (int i = v.size() - 1; i >= 0; --i) {
12            if (k <= 0) break;
13            vector<string> t(v[i].begin(), v[i].end());
14            if (k >= t.size()) {
15                res.insert(res.end(), t.begin(), t.end());
16            } else {
17                res.insert(res.end(), t.begin(), t.begin() + k);
18            }
19            k -= t.size();
20        }
21        return res;
22    }
23 };

```

680. 有交替位的二进制数

Given a positive integer, check whether it has alternating bits: namely, if two adjacent bits will always have different values.

这道题让我们判断一个二进制数的1和0是否是交替出现的，博主开始也想到啥简便方法，就一位一位来检测呗，用个变量bit来记录上一个位置的值，初始化为-1，然后我们用‘与’1的方法来获取最低位的值，如果是1，那么当此时bit已经是1的话，说明两个1相邻了，返回false，否则bit赋值为1。同理，如果是0，那么当此时bit已经是0的话，说明两个0相邻了，返回false，否则bit赋值为0。判断完别忘了将n向右移动一位。如果while循环退出了，返回true，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool hasAlternatingBits(int n) {
4         int bit = -1;
5         while (n > 0) {
6             if (n & 1 == 1) {
7                 if (bit == 1) return false;
8                 bit = 1;
9             } else {
10                 if (bit == 0) return false;
11                 bit = 0;
12             }
13             n >>= 1;
14         }
15         return true;
16     }
17 };

```

下面这种解法写的更加简洁了，我们不需要用if条件来判断，而是可以通过‘亦或’1的方式来将0和1互换，当然我们也可以通过 $d = 1 - d$ 来达到同样的效果，但还是写成‘亦或’1比较呀，while循环的条件是最低位等于d，而d不停的在0和1之间切换，n每次也向右平移一位，这样能交替检测0和1，循环退出后，如果n为0，则返回true，反之则返回false，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool hasAlternatingBits(int n) {
4         int d = n & 1;
5         while ((n & 1) == d) {
6             d ^= 1;
7             n >>= 1;
8         }
9         return n == 0;
10    }
11 };

```

下面这种解法就十分的巧妙了，利用了0和1的交替的特性，进行错位相加，从而组成全1的二进制数，然后再用一个检测全1的二进制数的trick，就是‘与’上加1后的数，因为全1的二进制数加1，就会进一位，并且除了最高位，其余位都是0，跟原数相‘与’就会得0，所以我们可以这样判断。比如n是10101，那么 $n >> 1$ 就是1010，二者相加就是11111，再加1就是100000，二者相‘与’就是0，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     bool hasAlternatingBits(int n) {
4         return ((n + (n >> 1) + 1) & (n + (n >> 1))) == 0;
5     }
6 };

```

下面这种解法也很巧妙，先将n右移两位，再和原来的n亦或，得到的新n其实就是除了最高位，其余都是0的数，然后再和自身减1的数相‘与’，如果是0就返回true，反之false。比如n是10101，那么 $n / 4$ 是101，二者相‘亦或’，得到10000，此时再减1，为1111，二者相‘与’得0，参见代码如下：

解法4：

CPP

```

1 class Solution {
2 public:
3     bool hasAlternatingBits(int n) {
4         return ((n ^= n / 4) & (n - 1)) == 0;
5     }
6 };

```

681. 不同岛屿的个数

Given a non-empty 2D array grid of 0's and 1's, an island is a group of 1's (representing land) connected 4-directionally (horizontal or vertical.) You may assume all four edges of the grid are surrounded by water.

Count the number of distinct islands. An island is considered to be the same as another if and only if one island can be translated (and not rotated or reflected) to equal the other.

这道题让我们求不同岛屿的个数，是之前那道Number of Islands的拓展，这道题的难点是如何去判断两个岛屿是否是不同的岛屿。首先1的个数肯定是要相同，但是1的个数相同不能保证一定是相同的岛屿，比如例子2中的那两个岛屿的就不相同，就是说两个相同的岛屿通过平移可以完全重合，但是不能旋转。那么我们如何来判断呢，我们发现可以通过相对位置坐标来判断，比如我们使用岛屿的最左上角的1当作基点，那么基点左边的点就是(0,-1)，右边的点就是(0,1)，上边的点就是(-1,0)，下面的点就是(1,0)。那么例子1中的两个岛屿都可以表示为[(0,0), (0,1), (1,0), (1,1)]，点的顺序是基点-右边点-下边点-右下点。通过这样就可以判断两个岛屿是否相同了，下面这种解法我们没有用数组来存，而是encode成了字符串，比如这四个点的数组就存为"0_0_0_1_1_0_1_1_"，然后把字符串存入集合unordered_set中，利用其自动去重复的特性，就可以得到不同的岛屿的数量啦，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<vector<int>> dirs{{0,-1},{-1,0},{0,1},{1,0}};
4     int numDistinctIslands(vector<vector<int>>& grid) {
5         int m = grid.size(), n = grid[0].size();
6         unordered_set<string> res;
7         vector<vector<bool>> visited(m, vector<bool>(n, false));
8         for (int i = 0; i < m; ++i) {
9             for (int j = 0; j < n; ++j) {
10                 if (grid[i][j] == 1 && !visited[i][j]) {
11                     set<string> s;
12                     helper(grid, i, j, i, j, visited, s);
13                     string t = "";
14                     for (auto str : s) t += str + "_";
15                     res.insert(t);
16                 }
17             }
18         }
19         return res.size();
20     }
21     void helper(vector<vector<int>>& grid, int x0, int y0, int i, int j,
22     vector<vector<bool>>& visited, set<string>& s) {
23         int m = grid.size(), n = grid[0].size();
24         visited[i][j] = true;
25         for (auto dir : dirs) {
26             int x = i + dir[0], y = j + dir[1];
27             if (x < 0 || x >= m || y < 0 || y >= n || grid[x][y] == 0 || visited[x][y])
28                 continue;
29             string str = to_string(x - x0) + "_" + to_string(y - y0);
30             s.insert(str);
31             helper(grid, x0, y0, x, y, visited, s);
32         }
33     }
34 };

```

当然我们也可以不encode字符串，直接将相对坐标存入数组中，然后把整个数组放到集合set中，还是会去掉相同的数组，而且这种解法直接在grid数组上标记访问过的位置，写起来更加简洁了，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<vector<int>> dirs{{0,-1},{-1,0},{0,1},{1,0}};
4     int numDistinctIslands(vector<vector<int>>& grid) {
5         int m = grid.size(), n = grid[0].size();
6         set<vector<pair<int, int>>> res;
7         for (int i = 0; i < m; ++i) {
8             for (int j = 0; j < n; ++j) {
9                 if (grid[i][j] != 1) continue;
10                vector<pair<int, int>> v;
11                helper(grid, i, j, i, j, v);
12                res.insert(v);
13            }
14        }
15        return res.size();
16    }
17    void helper(vector<vector<int>>& grid, int x0, int y0, int i, int j, vector<pair<int,
18 int>>& v) {
19        int m = grid.size(), n = grid[0].size();
20        if (i < 0 || i >= m || j < 0 || j >= n || grid[i][j] <= 0) return;
21        grid[i][j] *= -1;
22        v.push_back({i - x0, j - y0});
23        for (auto dir : dirs) {
24            helper(grid, x0, y0, i + dir[0], j + dir[1], v);
25        }
26    }
};

```

既然递归DFS可以，那么迭代的BFS就坐不住了，其实思路没什么区别，这种类似迷宫遍历的题都是一个套路，整体框架都很像，细枝末节需要改改就行了，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     vector<vector<int>> dirs{{0,-1},{-1,0},{0,1},{1,0}};
4     int numDistinctIslands(vector<vector<int>>& grid) {
5         int m = grid.size(), n = grid[0].size();
6         set<vector<pair<int, int>>> res;
7         for (int i = 0; i < m; ++i) {
8             for (int j = 0; j < n; ++j) {
9                 if (grid[i][j] != 1) continue;
10                vector<pair<int, int>> v;
11                queue<pair<int, int>> q{{{i, j}}};
12                grid[i][j] *= -1;
13                while (!q.empty()) {
14                    auto t = q.front(); q.pop();
15                    for (auto dir : dirs) {
16                        int x = t.first + dir[0], y = t.second + dir[1];
17                        if (x < 0 || x >= m || y < 0 || y >= n || grid[x][y] <= 0)
18                            continue;
19                        q.push({x, y});
20                        grid[x][y] *= -1;
21                        v.push_back({x - i, y - j});
22                    }
23                }
24                res.insert(v);
25            }
26        }
27        return res.size();
28    }
29 };

```

682. 岛的最大面积

Given a non-empty 2D array grid of 0's and 1's, an island is a group of 1's (representing land) connected 4-directionally (horizontal or vertical.) You may assume all four edges of the grid are surrounded by water.

Find the maximum area of an island in the given 2D array. (If there is no island, the maximum area is 0.)

这道题跟之前的那两道Number of Islands和Number of Distinct Islands是同一个类型的，只不过这次需要统计出每个岛的大小，再来更新结果res。先用递归来做，遍历grid，当遇到为1的点，我们调用递归函数，在递归函数中，我们首先判断i和j是否越界，还有grid[i][j]是否为1，我们没有用visited数组，而是直接修改了grid数组，遍历过的标记为-1。如果合法，那么cnt自增1，并且更新结果res，然后对其周围四个相邻位置分别调用递归函数即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<vector<int>> dirs{{0,-1},{-1,0},{0,1},{1,0}};
4     int maxAreaOfIsland(vector<vector<int>>& grid) {
5         int m = grid.size(), n = grid[0].size(), res = 0;
6         for (int i = 0; i < m; ++i) {
7             for (int j = 0; j < n; ++j) {
8                 if (grid[i][j] != 1) continue;
9                 int cnt = 0;
10                helper(grid, i, j, cnt, res);
11            }
12        }
13        return res;
14    }
15    void helper(vector<vector<int>>& grid, int i, int j, int& cnt, int& res) {
16        int m = grid.size(), n = grid[0].size();
17        if (i < 0 || i >= m || j < 0 || j >= n || grid[i][j] <= 0) return;
18        res = max(res, ++cnt);
19        grid[i][j] *= -1;
20        for (auto dir : dirs) {
21            helper(grid, i + dir[0], j + dir[1], cnt, res);
22        }
23    }
24};

```

下面是迭代的写法，BFS遍历，使用queue来辅助运算，思路没啥太大区别，都是套路，都是模版，往里套就行了，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<vector<int>> dirs{{0,-1},{-1,0},{0,1},{1,0}};
4     int maxAreaOfIsland(vector<vector<int>>& grid) {
5         int m = grid.size(), n = grid[0].size(), res = 0;
6         for (int i = 0; i < m; ++i) {
7             for (int j = 0; j < n; ++j) {
8                 if (grid[i][j] != 1) continue;
9                 int cnt = 0;
10                queue<pair<int, int>> q{{{i, j}}};
11                grid[i][j] *= -1;
12                while (!q.empty()) {
13                    auto t = q.front(); q.pop();
14                    res = max(res, ++cnt);
15                    for (auto dir : dirs) {
16                        int x = t.first + dir[0], y = t.second + dir[1];
17                        if (x < 0 || x >= m || y < 0 || y >= n || grid[x][y] <= 0)
18                            continue;
19                        grid[x][y] *= -1;
20                        q.push({x, y});
21                    }
22                }
23            }
24        }
25        return res;
26    }
27};

```

683. 统计二进制子字符串

Give a string s , count the number of non-empty (contiguous) substrings that have the same number of 0's and 1's, and all the 0's and all the 1's in these substrings are grouped consecutively.

Substrings that occur multiple times are counted the number of times they occur.

这道题给了我们一个二进制字符串，然后我们统计具有相同0和1的个数，且0和1各自都群组在一起(即0和1不能交替出现)的子字符串的个数，题目中的两个例子也很能说明问题。那么我们来分析题目中的第一个例子00110011，符合要求的子字符串要求0和1同时出现，那么当第一个1出现的时候，前面由于前面有两个0，所以肯定能组成01，再遇到下一个1时，此时1有2个，0有2个，能组成0011，下一个遇到0时，此时0的个数重置为1，而1的个数有两个，所以一定有10，同理，下一个还为0，就会有1100存在，之后的也是这样分析。那么我们可以发现我们只要分别统计0和1的个数，而且如果当前遇到的是1，那么只要之前统计的0的个数大于当前1的个数，就一定有一个对应的子字符串，而一旦前一个数字和当前的数字不一样的时候，那么当前数字的计数要重置为1。所以我们遍历元数组，如果是第一个数字，那么对应的ones或zeros自增1。然后进行分情况讨论，如果当前数字是1，然后判断如果前面的数字也是1，则ones自增1，否则ones重置为1。如果此时zeros大于ones，res自增1。反之同理，如果当前数字是0，然后判断如果前面的数字也是0，则zeros自增1，否则zeros重置为1。如果此时ones大于zeros，res自增1。参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int countBinarySubstrings(string s) {
4         int zeros = 0, ones = 0, res = 0;
5         for (int i = 0; i < s.size(); ++i) {
6             if (i == 0) {
7                 (s[i] == '1') ? ++ones : ++zeros;
8             } else {
9                 if (s[i] == '1') {
10                     ones = (s[i - 1] == '1') ? ones + 1 : 1;
11                     if (zeros >= ones) ++res;
12                 } else if (s[i] == '0') {
13                     zeros = (s[i - 1] == '0') ? zeros + 1 : 1;
14                     if (ones >= zeros) ++res;
15                 }
16             }
17         }
18         return res;
19     }
20 };

```

CPP

下面这种方法更加简洁了，不用具体的分0和1的情况来讨论了，而是直接用了pre和cur两个变量，其中pre初始化为0，cur初始化为1，然后从第二个数字开始遍历，如果当前数字和前面的数字相同，则cur自增1，否则pre赋值为cur，cur重置1。然后判断如果pre大于等于cur，res自增1。其实核心思想跟上面的方法一样，只不过pre和cur可以在0和1之间切换，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int countBinarySubstrings(string s) {
4         int res = 0, pre = 0, cur = 1, n = s.size();
5         for (int i = 1; i < n; ++i) {
6             if (s[i] == s[i - 1]) ++cur;
7             else {
8                 pre = cur;
9                 cur = 1;
10            }
11            if (pre >= cur) ++res;
12        }
13        return res;
14    }
15 };

```

684. 数组的度

Given a non-empty array of non-negative integers `nums`, the degree of this array is defined as the maximum frequency of any one of its elements.

Your task is to find the smallest possible length of a (contiguous) subarray of `nums`, that has the same degree as `nums`.

这道题给了我们一个数组，定义数组的度为某个或某些数字出现最多的次数，要我们找最短的子数组使其和原数组拥有相同的度。那么我们肯定需要统计每个数字出现的次数，就要用哈希表来建立每个数字和其出现次数之间的映射。由于我们要求包含原度的最小长度的子数组，那么最好的情况就是子数组的首位数字都是统计度的数字，即出现最多的数字。那么我们肯定要知道该数字的第一次出现的位置和最后一次出现的位置，由于我们开始不知道哪些数字会出现最多次，所以我们统计所有数字的首尾出现位置，那么我们再用一个哈希表，建立每个数字和其首尾出现的位置。我们用变量`degree`来表示数组的度。好，现在我们遍历原数组，累加当前数字出现的次数，当某个数字是第一次出现，那么我们用当前位置的来更新该数字出现的首尾位置，否则只更新尾位置。每遍历一个数，我们都更新一下`degree`。当遍历完成后，我们已经有了数组的度，还有每个数字首尾出现的位置，下面就来找出现次数为`degree`的数组，然后计算其首尾位置差加1就是`candidate`数组的长度，由于出现次数为`degree`的数字不一定只有一个，我们遍历所有的，找出其中最小的即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int findShortestSubArray(vector<int>& nums) {
4         int n = nums.size(), res = INT_MAX, degree = 0;
5         unordered_map<int, int> m;
6         unordered_map<int, pair<int, int>> pos;
7         for (int i = 0; i < nums.size(); ++i) {
8             if (++m[nums[i]] == 1) {
9                 pos[nums[i]] = {i, i};
10            } else {
11                pos[nums[i]].second = i;
12            }
13            degree = max(degree, m[nums[i]]);
14        }
15        for (auto a : m) {
16            if (degree == a.second) {
17                res = min(res, pos[a.first].second - pos[a.first].first + 1);
18            }
19        }
20        return res;
21    }
22 };

```

下面这种方法只用了一次遍历，思路跟上面的解法很相似，还是要建立数字出现次数的哈希表，还有就是建立每个数字和其第一次出现位置之间的映射，那么我们当前遍历的位置其实可以看作是尾位置，还是可以计算子数组的长度的。我们遍历数组，累加当前数字出现的次数，如果某个数字是第一次出现，建立该数字和当前位置的映射，如果当前数字的出现次数等于degree时，当前位置为尾位置，首位置在startIdx中取的，二者做差加1来更新结果res；如果当前数字的出现次数大于degree，说明之前的结果代表的数字不是出现最多的，直接将结果res更新为当前数字的首尾差加1的长度，然后degree也更新为当前数字出现的次数。参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int findShortestSubArray(vector<int>& nums) {
4         int n = nums.size(), res = INT_MAX, degree = 0;
5         unordered_map<int, int> m, startIdx;
6         for (int i = 0; i < n; ++i) {
7             ++m[nums[i]];
8             if (!startIdx.count(nums[i])) startIdx[nums[i]] = i;
9             if (m[nums[i]] == degree) {
10                 res = min(res, i - startIdx[nums[i]] + 1);
11             } else if (m[nums[i]] > degree) {
12                 res = i - startIdx[nums[i]] + 1;
13                 degree = m[nums[i]];
14             }
15         }
16         return res;
17     }
18 };

```

685. 分割K个等和的子集

Given an array of integers nums and a positive integer k, find whether it's possible to divide this array into knon-empty subsets whose sums are all equal.

这道题给了我们一个数组nums和一个数字k，问我们该数字能不能分成k个非空子集合，使得每个子集合的和相同。给了k的范围是[1,16]，而且数组中的数字都是正数。这跟之前那道Partition Equal Subset Sum很类似，但是那道题只让分成两个子集合，所以问题可以转换为是否存在和为整个数组和的一半的子集合，可以用dp来做。但是这道题让求k个和相同的，感觉无法用dp来做，因为就算找出了一个，其余的也需要验证。这道题我们可以用递归来做，首先我们还是求出数组的所有数字之和sum，首先判断sum是否能整除k，不能整除的话直接返回false。然后需要一个visited数组来记录哪些数组已经被选中了，然后调用递归函数，我们的目标是组k个子集合，是的每个子集合之和为target = sum/k。我们还需要变量start，表示从数组的某个位置开始查找，curSum为当前子集合之和，在递归函数中，如果k=1，说明此时只需要组一个子集合，那么当前的就是了，直接返回true。如果curSum等于target了，那么我们再次调用递归，此时传入k-1，start和curSum都重置为0，因为我们当前又找到了一个和为target的子集合，要开始继续找下一个。否则的话就从start开始遍历数组，如果当前数字已经访问过了则直接跳过，否则标记为已访问。然后调用递归函数，k保持不变，因为还在累加当前的子集合，start传入i+1，curSum传入curSum+nums[i]，因为要累加当前的数字，如果递归函数返回true了，则直接返回true。否则就将当前数字重置为未访问的状态继续遍历，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     bool canPartitionKSubsets(vector<int>& nums, int k) {
4         int sum = accumulate(nums.begin(), nums.end(), 0);
5         if (sum % k != 0) return false;
6         vector<bool> visited(nums.size(), false);
7         return helper(nums, k, sum / k, 0, 0, visited);
8     }
9     bool helper(vector<int>& nums, int k, int target, int start, int curSum, vector<bool>&
10    visited) {
11         if (k == 1) return true;
12         if (curSum == target) return helper(nums, k - 1, target, 0, 0, visited);
13         for (int i = start; i < nums.size(); ++i) {
14             if (visited[i]) continue;
15             visited[i] = true;
16             if (helper(nums, k, target, i + 1, curSum + nums[i], visited)) return true;
17             visited[i] = false;
18         }
19         return false;
20     }
21 };

```

CPP

下面这种方法也挺巧妙的，思路是建立长度为k的数组v，只有当v里面所有的数字都是target的时候，才能返回true。我们还需要给数组排个序，由于题目中限制了全是正数，所以数字累加只会增大不会减小，一旦累加超过了target，这个子集合是无法再变小的，所以就不能加入这个数。实际上相当于贪婪算法，由于题目中数组数字为正的限制，有解的话就可以用贪婪算法得到。我们用一个变量idx表示当前遍历的数字，排序后，我们从末尾大的数字开始累加，我们遍历数组v，当前位置加上nums[idx]，如果超过了target，我们掉过继续到下一个位置，否则就调用递归，此时的idx为idx-1，表示之前那个数字已经成功加入数组v了，我们尝试着加下一个数字。如果递归返回false了，我们就将nums[idx]从数组v中对应的位置减去，还原状态，然后继续下一个位置。如果某个递归中idx等于-1了，表明所有的数字已经遍历完了，此时我们检查数组v中k个数字是否都为target，是的话返回true，否则返回false，参见代码如下

解法2：

```

1 class Solution {
2 public:
3     bool canPartitionKSubsets(vector<int>& nums, int k) {
4         int sum = accumulate(nums.begin(), nums.end(), 0);
5         if (sum % k != 0) return false;
6         vector<int> v(k, 0);
7         sort(nums.begin(), nums.end());
8         return helper(nums, sum / k, v, (int)nums.size() - 1);
9     }
10    bool helper(vector<int>& nums, int target, vector<int>& v, int idx) {
11        if (idx == -1) {
12            for (int t : v) {
13                if (t != target) return false;
14            }
15            return true;
16        }
17        int num = nums[idx];
18        for (int i = 0; i < v.size(); ++i) {
19            if (v[i] + num > target) continue;
20            v[i] += num;
21            if (helper(nums, target, v, idx - 1)) return true;
22            v[i] -= num;
23        }
24        return false;
25    }
26};

```

686. 下落的方块

On an infinite number line (x -axis), we drop given squares in the order they are given.

The i -th square dropped ($\text{positions}[i] = (\text{left}, \text{side_length})$) is a square with the left-most point being $\text{positions}[i][0]$ and sidelength $\text{positions}[i][1]$.

The square is dropped with the bottom edge parallel to the number line, and from a higher height than all currently landed squares. We wait for each square to stick before dropping the next.

The squares are infinitely sticky on their bottom edge, and will remain fixed to any positive length surface they touch (either the number line or another square). Squares dropped adjacent to each other will not stick together prematurely.

Return a list ans of heights. Each height $\text{ans}[i]$ represents the current highest height of any square we have dropped, after dropping squares represented by $\text{positions}[0], \text{positions}[1], \dots, \text{positions}[i]$.

这道题不就是经典的俄罗斯方块么？！只不过是简化版的，我们只有方块下落，没有其他那些奇形怪状的东西，这样简化了难度，不过方块的大小不是固定的，有可能很大，但是不管方块再大，只要有一点点部分搭在其他方块上面，整个方块都会在上面，并不会掉下来，让我们求每落下一个方块后的最大高度。我们知道返回的是每落下一个方块后当前场景中的最大高度，那么返回的数组的长度就应该和落下方块的个数相同。所以我们可以建立一个heights数组，其中heights[i]表示第i块方块落下后所在的高度，那么第i块方块落下后场景的最大高度就是[0, i]区间内的最大值。那么我们在求出heights数组后，只要不停返回[0, i]区间内的最大值即可。继续来看，这道题的难点就是方块重叠的情况，我们先来想，如果各个方块不重叠，那么heights[i]的高度就是每个方块自身的高度。一旦重叠了，就得在已有的基础上再加上自身的高度。那么我们可以采用brute force的思想，对于每一个下落的方块，我们都去看和后面将要落下的方块有没有重叠，有的话，和后面将要落下的方块的位置相比较，取二者中较大值为后面要落下的方块位置高度heights[j]。判断两个方块是否重叠的方法是如果方块2的左边界小于方块1的右边界，并且方块2点右边界大于方块1点左边界。就拿题目中的例子1来举例吧，第一个下落的方块的范围是[1, 3]，长度为2，则

heights[0]=2，然后我们看其和第二个方块[2, 5]是否重叠，发现是重叠的，则heights[1]更新为2，再看第三个方块[6, 7]，不重叠，不更新。然后第二个方块落下，此时累加高度，则heights[1]=5，再看第三个方块，不重叠，不更新。然后第三个方块落下，heights[2]=1。此时我们heights数组更新好了，然后我们开始从头遍历，维护一个当前最大值curMax，每次将[0, i]中最大值加入结果res即可，参见代码如下：

解法1：

```

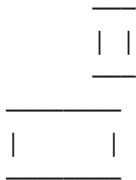
1 class Solution {
2 public:
3     vector<int> fallingSquares(vector<pair<int, int>>& positions) {
4         int n = positions.size(), cur = 0;
5         vector<int> heights(n), res;
6         for (int i = 0; i < n; ++i) {
7             int len = positions[i].second, left = positions[i].first, right = left + len;
8             heights[i] += len;
9             for (int j = i + 1; j < n; ++j) {
10                 int l = positions[j].first, r = l + positions[j].second;
11                 if (l < right && r > left) {
12                     heights[j] = max(heights[j], heights[i]);
13                 }
14             }
15         }
16         for (int h : heights) {
17             cur = max(cur, h);
18             res.push_back(cur);
19         }
20         return res;
21     }
22 };

```

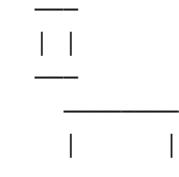
CPP

我们来看一种时间复杂度为 $O(n\log n)$ 的解法，这种解法将每一个不同高度的区间都存到了一个HashMap中，然后每当有新的方块落下的时候，可以使用二分法来快速定位到可能发生重叠的区间的位置，如果有重叠的话，将原区间再根据高度拆成多个小区间，并且一直维护一个当前出现过的最高值，并加入结果res中。我们的HashMap的映射是建立pair和int之间的映射，由于HashMap是有自动排序的功能，默认的是使用pair中第一个元素，正好就是每个方块的起始位置。然后我们遍历每个下落的方块，建立一个临时的二维数组t，用来保存拆分后的小区间。然后我们取出当前方块的起始终止位置start和end，然后我们希望在HashMap中找第一个不大于当前方块起始位置的区间，在Java中我们可以使用floorKey()函数，但是在C++中，我们只有lower_bound()和upper_bound()可以用，分别表示找第一个不大于目标值，和第一个大于目标值的区间，那么我们为了找到第一个不大于当前起始位置的区间，可以先用upper_bound()来找到第一个大于起始位置的区间，然后向前移动一个，就是第一个不大于的了。注意向前移动操作有前提条件，就是upper_bound()返回的位置不能是首位置，否则无法前移，还有就是如果前一个区间的结束为止小于等于当前区间的起始位置，说明两个区间没有重叠，我们再移回来。

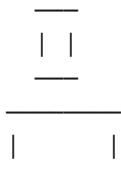
下面就要进行拆分区间的操作了，我们用一个while循环，循环条件是it存在，并且it指向区间的起始位置小于当前区间的结束位置。由于之前的操作确定了这两个区间一定会有重叠，那么重叠的方式就有一下四种（上方为当前区间，下方为it区间）：



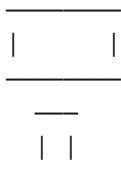
如上图所示，如果当前区间（上方）的起始位置大于it指向区间（下方）的起始位置，说明红色那段区间需要被拆分出来，我们将其拆分出来并存入数组t中。



如上图所示，如果当前区间（上方）的结束位置小于it指向区间（下方）的结束位置，说明洋红色那段区间需要被拆分出来，我们将其拆分出来并存入数组t中。



如上图所示，红色区间和洋红色区间都需要拆分出来，我们将其拆分出来并存入数组t中。



如上图所示，底层方块被完全覆盖了，没有区间需要被拆分出来。

我们用底层it指向的区间的高度来更新h，这里的h表示当前方块下落后的基础高度，然后我们将底层it指向的区间删除，因为我们已经将没有被覆盖的区间拆分出来并存入数组t中了。注意erase()函数返回的是被删除区间的下一个位置，这样使得我们的while函数能继续判断，直到it区间和当前区间不再重叠位置。退出while循环后，我们需要将当前下落方块的区间加入HashMap中，高度为基础高度h加上自身高度len。接下来就把数组t中拆分出来的小区间都加入到HashMap中，然后用当前用h+len来更新curMax，表示当前场景最大高度，加入结果res中，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<int> fallingSquares(vector<pair<int, int>>& positions) {
4         vector<int> res;
5         map<pair<int, int>, int> m;
6         int curMax = 0;
7         for (auto &pos : positions) {
8             vector<vector<int>> t;
9             int len = pos.second, start = pos.first, end = start + len, h = 0;
10            auto it = m.upper_bound({start, start});
11            if (it != m.begin() && (--it)->first.second <= start) ++it;
12            while (it != m.end() && it->first.first < end) {
13                if (start > it->first.first) t.push_back({it->first.first, start, it-
14                >second});
15                if (end < it->first.second) t.push_back({end, it->first.second, it-
16                >second});
17                h = max(h, it->second);
18                it = m.erase(it);
19            }
20            m[{start, end}] = h + len;
21            for (auto &a : t) m[{a[0], a[1]}] = a[2];
22            curMax = max(curMax, h + len);
23            res.push_back(curMax);
24        }
25        return res;
    }
};


```

687. 二叉搜索树中的搜索

给定二叉搜索树（BST）的根节点和一个值。你需要在BST中找到节点值等于给定值的节点。返回以该节点为根的子树。如果节点不存在，则返回 NULL。

```

1 class Solution {
2 public:
3     TreeNode* searchBST(TreeNode* root, int val) {
4         if (root == NULL) { return NULL; }
5         if (root->val == val) { return root; }
6         //Recursion
7         if (root->val < val) { return searchBST(root->right, val); }
8         if (root->val > val) { return searchBST(root->left, val); }
9     }
10 };

```

688. 二叉搜索树中的插入操作

给定二叉搜索树（BST）的根节点和要插入树中的值，将值插入二叉搜索树。返回插入后二叉搜索树的根节点。保证原始二叉搜索树中不存在新值。

注意，可能存在多种有效的插入方式，只要树在插入后仍保持为二叉搜索树即可。你可以返回任意有效的结果。

```

1 class Solution {
2 public:
3     TreeNode* insertIntoBST(TreeNode* root, int val) {
4         TreeNode *cur = root, *node = new TreeNode(val);
5         while (cur) {
6             if (cur->val > val) {
7                 if (cur->left) {
8                     cur = cur->left;
9                 } else {
10                     cur->left = node;
11                     break;
12                 }
13             } else {
14                 if (cur->right) {
15                     cur = cur->right;
16                 } else {
17                     cur->right = node;
18                     break;
19                 }
20             }
21         }
22         return root ? root : node;
23     }
24 };

```

689. Search in a Sorted Array of Unknown Size

```

1 class ArrayReader;
2
3 class Solution {
4 public:
5     int search(const ArrayReader& reader, int target) {
6         int left = 0, right = 19999;
7         while (left <= right) {
8             auto mid = left + (right-left) / 2;
9             auto response = reader.get(mid);
10            if (response > target) {
11                right = mid - 1;
12            } else if (response < target) {
13                left = mid + 1;
14            } else {
15                return mid;
16            }
17        }
18        return -1;
19    }
20};

```

690. 数据流中的第K大元素

设计一个找到数据流中第K大元素的类（class）。注意是排序后的第K大元素，不是第K个不同的元素。

你的 KthLargest 类需要一个同时接收整数 k 和整数数组nums 的构造器，它包含数据流中的初始元素。每次调用 KthLargest.add，返回当前数据流中第K大的元素。

```

1 class KthLargest {
2 public:
3     KthLargest(int k, vector<int> nums) :
4         k_(k) {
5             for (const auto& num : nums) {
6                 add(num);
7             }
8         }
9
10    int add(int val) {
11        min_heap_.emplace(val);
12        if (min_heap_.size() > k_) {
13            min_heap_.pop();
14        }
15        return min_heap_.top();
16    }
17
18 private:
19     const int k_;
20     priority_queue<int, vector<int>, greater<int>> min_heap_;
21 };

```

691. 二分查找

给定一个 n 个元素有序的（升序）整型数组 nums 和一个目标值 target，写一个函数搜索 nums 中的 target，如果目标值存在返回下标，否则返回 -1。

```

1 class Solution {
2 public:
3     int search(vector<int>& nums, int target) {
4         int l = 0, r = nums.size() - 1;
5         while (l <= r) {
6             int m = (l + r) / 2;
7             if (nums[m] == target) {
8                 return m;
9             } else if (nums[m] > target) {
10                 r = m - 1;
11             } else {
12                 l = m + 1;
13             }
14         }
15         return -1;
16     }
17 };

```

692. 设计哈希集合

不使用任何内建的哈希表库设计一个哈希集合

具体地说，你的设计应该包含以下的功能

`add(value)`: 向哈希集合中插入一个值。

`contains(value)` : 返回哈希集合中是否存在这个值。

`remove(value)`: 将给定值从哈希集合中删除。如果哈希集合中没有这个值，什么也不做。

```

1 class MyHashSet {
2 public:
3     /** Initialize your data structure here. */
4     MyHashSet() {
5
6     }
7
8     void add(int key) {
9         data[key] = true;
10    }
11
12    void remove(int key) {
13        data[key] = false;
14    }
15
16    /** Returns true if this set contains the specified element */
17    bool contains(int key) {
18        return data[key];
19    }
20 private:
21     bool data[1000001] = {false};
22 };

```

693. 设计哈希映射

不使用任何内建的哈希表库设计一个哈希映射

具体地说，你的设计应该包含以下的功能

`put(key, value)`: 向哈希映射中插入(键, 值)的数值对。如果键对应的值已经存在，更新这个值。

`get(key)`: 返回给定的键所对应的值，如果映射中不包含这个键，返回-1。

`remove(key)`: 如果映射中存在这个键，删除这个数值对。

```

1 class MyHashMap {
2 public:
3 private:
4     int elements[1000001];
5 public:
6     MyHashMap() {
7         memset(elements, -1, 1000001);
8     }
9
10    void put(int key, int value) {
11        elements[key] = value;
12    }
13
14    int get(int key) {
15        return elements[key];
16    }
17
18    void remove(int key) {
19        elements[key] = -1;
20    }
21 };

```

694. 设计链表

设计链表的实现。您可以选择使用单链表或双链表。单链表中的节点应该具有两个属性：`val` 和 `next`。`val` 是当前节点的值，`next` 是指向下一个节点的指针/引用。如果要使用双向链表，则还需要一个属性 `prev` 以指示链表中的上一个节点。假设链表中的所有节点都是 0-index 的。

在链表类中实现这些功能：

`get(index)`: 获取链表中第 `index` 个节点的值。如果索引无效，则返回 -1。

`addAtHead(val)`: 在链表的第一个元素之前添加一个值为 `val` 的节点。插入后，新节点将成为链表的第一个节点。

`addAtTail(val)`: 将值为 `val` 的节点追加到链表的最后一个元素。

`addAtIndex(index, val)`: 在链表中的第 `index` 个节点之前添加值为 `val` 的节点。如果 `index` 等于链表的长度，则该节点将附加到链表的末尾。如果 `index` 大于链表长度，则不会插入节点。

`deleteAtIndex(index)`: 如果索引 `index` 有效，则删除链表中的第 `index` 个节点。

```
1 class MyLinkedList {
2     private:
3         struct node{
4             int val;
5             node *next;
6             // node(int x) : val(x), next(nullptr) {}
7         };
8         node *head;
9         node *tail;
10        int size;
11    public:
12        /** Initialize your data structure here. */
13        MyLinkedList() {
14            head = nullptr;
15            tail = nullptr;
16            size = 0;
17        }
18
19        /** Get the value of the index-th node in the linked list. If the index is invalid,
20        return -1. */
21        int get(int index) {
22            if(index >= size|| index < 0) return -1;
23            node *cur = new node;
24            cur = head;
25            // while(cur!=nullptr){
26            //     cout<<" no. "<<cur->val;
27            //     cur = cur->next;
28            // }
29            for(int i = 0;i<index;++i){
30                cur = cur->next;
31            }
32
33            return cur->val;
34        }
35
36
37        /** Add a node of value val before the first element of the linked list. After the
38        insertion, the new node will be the first node of the linked list. */
39        void addAtHead(int val) {
40            node *temp = new node();
41            temp->val = val;
42            temp->next = head;
43            head = temp;
44            if(size==0) tail = temp;
45            ++size;
46        }
47
48        /** Append a node of value val to the last element of the linked list. */
49        void addAtTail(int val) {
50            node *temp = new node();
51            temp->val = val;
52            if(size==0) {
53                tail = temp;
54                head = temp;
55            }
56            tail->next = temp;
57            tail = temp;
58            ++size;
59        }
}
```

```
60
61     /** Add a node of value val before the index-th node in the linked list. If index
62      equals to the length of linked list, the node will be appended to the end of linked list.
63      If index is greater than the length, the node will not be inserted. */
64      void addAtIndex(int index, int val) {
65          if(index>size || index < 0) return;
66          else if(index == 0) {
67              addAtHead(val);
68              return;
69          }
70          else if(index == size) {
71              addAtTail(val);
72              return;
73          }
74          node *cur = new node();
75          cur = head;
76          for(int i = 0;i<index-1;++i){
77              cur = cur->next;
78          }
79          node *temp = new node();
80          temp->val = val;
81          temp->next = cur->next;
82          cur->next = temp;
83          ++size;
84      }
85
86     /** Delete the index-th node in the linked list, if the index is valid. */
87     void deleteAtIndex(int index) {
88         if(index>=size || index < 0) return;
89         else if(index == 0){
90             head = head->next;
91             --size;
92             return;
93         }
94         node *cur = new node();
95         cur = head;
96         for(int i = 0;i<index-1;++i){
97             cur = cur->next;
98         }
99         cur->next= cur->next->next;
100        if(index == size-1) tail = cur;
101        --size;
102    }
103};
```

695. Insert into cyclic sorted list

```
1 class Solution {
2 public:
3     Node* insert(Node* head, int insertVal) {
4         if (head == nullptr) {
5             auto node = new Node(insertVal, nullptr);
6             node->next = node;
7             return node;
8         }
9         auto curr = head;
10        while (true) {
11            if (curr->val < curr->next->val) {
12                if (curr->val <= insertVal &&
13                    insertVal <= curr->next->val) {
14                    insertAfter(curr, insertVal);
15                    break;
16                }
17            } else if (curr->val > curr->next->val) {
18                if (curr->val <= insertVal ||
19                    insertVal <= curr->next->val) {
20                    insertAfter(curr, insertVal);
21                    break;
22                }
23            } else {
24                if (curr->next == head) {
25                    insertAfter(curr, insertVal);
26                    break;
27                }
28            }
29            curr = curr->next;
30        }
31        return head;
32    }
33
34 private:
35     void insertAfter(Node *node, int val) {
36         node->next = new Node(val, node->next);
37     }
38 };
```

696. 转换成小写字母

实现函数 ToLowerCase(), 该函数接收一个字符串参数 str, 并将该字符串中的大写字母转换成小写字母, 之后返回新的字符串。

```

1 class Solution {
2 public:
3     string toLowerCase(string str) {
4         for(int i = 0; i < str.length(); i++) {
5             str[i] = str[i] >= 'A' && str[i] <= 'Z' ? str[i] + 32: str[i];
6         }
7
8         return str;
9
10    /*
11        for (int i=0; i<str.length(); i++)
12        {
13            if (isupper(str[i]))
14                str[i]=tolower(str[i]);
15        }
16        return str;
17    */
18    }
19 };

```

697. 黑名单中的随机数

给定一个包含 $[0, n]$ 中独特的整数的黑名单 B , 写一个函数从 $[0, n]$ 中返回一个不在 B 中的随机整数。

对它进行优化使其尽量少调用系统方法 `Math.random()`。

```

1 class Solution {
2 private:
3     int upperBound;
4     vector<int> blackCnt;
5
6 public:
7     Solution(int N, vector<int> blacklist) : upperBound(N), blackCnt(blacklist)  {
8         upperBound -= (int)blackCnt.size();
9         sort(blackCnt.begin(), blackCnt.end());
10        for(int i = 1; i < blackCnt.size(); ++i)
11            blackCnt[i] -= i;
12    }
13
14     int pick() {
15         int num = rand() % upperBound;
16         int sta = 0, end = blackCnt.size();
17         while(sta < end) {
18             int mid = sta + (end - sta) / 2;
19             if(blackCnt[mid] <= num)
20                 sta = mid + 1;
21             else
22                 end = mid;
23         }
24         return num + sta;
25     }
26 };

```

698. 不同岛屿的个数之二

Given a non-empty 2D array grid of 0's and 1's, an island is a group of 1's (representing land) connected 4-directionally (horizontal or vertical.) You may assume all four edges of the grid are surrounded by water.

Count the number of distinct islands. An island is considered to be the same as another if they have the same shape, or have the same shape after rotation (90, 180, or 270 degrees only) or reflection (left/right direction or up/down direction).

```

1 class Solution {
2 public:
3     int numDistinctIslands2(vector<vector<int>>& grid) {
4         unordered_set<vector<pair<int, int>>, VectorHash> islands;
5         for (int i = 0; i < grid.size(); ++i) {
6             for (int j = 0; j < grid[i].size(); ++j) {
7                 if (grid[i][j] == 1) {
8                     vector<pair<int, int>> island;
9                     if (dfs(i, j, &grid, &island)) {
10                         islands.emplace(normalize(island));
11                     }
12                 }
13             }
14         }
15         return islands.size();
16     }
17
18 private:
19     struct VectorHash {
20         size_t operator()(const std::vector<pair<int, int>>& v) const {
21             size_t seed = 0;
22             for (const auto& i : v) {
23                 seed ^= std::hash<int>{}(i.first) + 0x9e3779b9 + (seed<<6) + (seed>>2);
24                 seed ^= std::hash<int>{}(i.second) + 0x9e3779b9 + (seed<<6) + (seed>>2);
25             }
26             return seed;
27         }
28     };
29
30     bool dfs(const int i, const int j,
31              vector<vector<int>> *grid, vector<pair<int, int>> *island) {
32
33         static const vector<pair<int, int>> directions{{1, 0}, {-1, 0},
34                                         {0, 1}, {0, -1}};
35
36         if (i < 0 || i >= grid->size() ||
37             j < 0 || j >= (*grid)[0].size() ||
38             (*grid)[i][j] <= 0) {
39             return false;
40         }
41         (*grid)[i][j] *= -1;
42         island->emplace_back(i, j);
43         for (const auto& direction : directions) {
44             dfs(i + direction.first, j + direction.second, grid, island);
45         }
46         return true;
47     }
48
49     vector<pair<int,int>> normalize(const vector<pair<int, int>>& island) {
50         vector<vector<pair<int,int>>> shapes(8);
51         for (const auto& p : island) {
52             int x, y;
53             tie(x, y) = p;
54             vector<pair<int, int>> rotations_and_reflections{{ x, y}, { x, -y}, {-x, y},
55             {-x, -y},
56             { y, x}, { y, -x}, {-y, x}, {-y, -x}};
57             for (int i = 0; i < rotations_and_reflections.size(); ++i) {
58                 shapes[i].emplace_back(rotations_and_reflections[i]);
59             }
}

```

```

60 }
61     for (auto& shape : shapes) {
62         sort(shape.begin(), shape.end());
63         const auto origin = shape.front();
64         for (auto& p : shape) {
65             p = {p.first - origin.first,
66                  p.second - origin.second};
67         }
68     }
69     return *min_element(shapes.begin(), shapes.end());
70 }

```

699. 两个字符串的最小ASCII删除和

Given two strings s1, s2, find the lowest ASCII sum of deleted characters to make two strings equal.

这道题给了我们两个字符串，让我们删除一些字符使得两个字符串相等，我们希望删除的字符的ASCII码最小。这道题跟之前那道Delete Operation for Two Strings极其类似，那道题让求删除的最少的字符数，这道题换成了ASCII码值。其实很多大厂的面试就是这种改动，虽然很少出原题，但是这种小范围的改动却是很经常的，所以当背题侠是没有用的，必须要完全掌握了解题思想，并能举一反三才是最重要的。看到这种玩字符串又是求极值的题，想都不要想直接上DP，我们建立一个二维数组dp，其中 $dp[i][j]$ 表示字符串s1的前*i*个字符和字符串s2的前*j*个字符变相等所要删除的字符的最小ASCII码累加值。那么我们可以先初始化边缘，即有一个字符串为空的话，那么另一个字符串有多少字符就要删多少字符，才能变空字符串。所以我们初始化 $dp[0][j]$ 和 $dp[i][0]$ ，计算方法就是上一个dp值加上对应位置的字符，有点像计算累加数组的方法，由于字符就是用ASCII表示的，所以我们不用转int，直接累加就可以。这里我们把 $dp[i][0]$ 的计算放入大的循环中计算，是为了少写一个for循环。好，现在我们来看递推公式，需要遍历这个二维数组的每一个位置即 $dp[i][j]$ ，当对应位置的字符相等时， $s1[i-1] == s2[j-1]$ ，(注意由于dp数组的i和j是从1开始的，所以字符串中要减1)，那么我们直接赋值为上一个状态的dp值，即 $dp[i-1][j-1]$ ，因为已经匹配上了，不用删除字符。如果 $s1[i-1] != s2[j-1]$ ，那么就有两种情况，我们可以删除 $s1[i-1]$ 的字符，且加上被删除的字符的ASCII码到上一个状态的dp值中，即 $dp[i-1][j] + s1[i-1]$ ，或者删除 $s2[j-1]$ 的字符，且加上被删除的字符的ASCII码到上一个状态的dp值中，即 $dp[i][j-1] + s2[j-1]$ 。这不难理解吧，比如sea和eat，当首字符s和e失配了，那么有两种情况，要么删掉s，用ea和eat继续匹配，或者删掉e，用sea和at继续匹配，记住删掉的字符一定要累加到dp值中才行，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int minimumDeleteSum(string s1, string s2) {
4         int m = s1.size(), n = s2.size();
5         vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
6         for (int j = 1; j <= n; ++j) dp[0][j] = dp[0][j - 1] + s2[j - 1];
7         for (int i = 1; i <= m; ++i) {
8             dp[i][0] = dp[i - 1][0] + s1[i - 1];
9             for (int j = 1; j <= n; ++j) {
10                 dp[i][j] = (s1[i - 1] == s2[j - 1]) ? dp[i - 1][j - 1] : min(dp[i - 1][j] +
11 s1[i - 1], dp[i][j - 1] + s2[j - 1]);
12             }
13         }
14         return dp[m][n];
15     }
}

```

CPP

我们也可以优化空间复杂度，使用一个一维数组dp，其中 $dp[i]$ 表示字符串s1和字符串s2的前*i*个字符变相等所要删除的字符的最小ASCII码累加值。刚开始还是要初始化 $dp[j]$ ，这里用变量t1和t2保存上一个状态的值，并不断更新。如果面试官没有特别的要求，还是用二维dp数组吧，毕竟逻辑更清晰一些，一维的容易写错~

解法2：

```

1 class Solution {
2 public:
3     int minimumDeleteSum(string s1, string s2) {
4         int m = s1.size(), n = s2.size();
5         vector<int> dp(n + 1, 0);
6         for (int j = 1; j <= n; ++j) dp[j] = dp[j - 1] + s2[j - 1];
7         for (int i = 1; i <= m; ++i) {
8             int t1 = dp[0];
9             dp[0] += s1[i - 1];
10            for (int j = 1; j <= n; ++j) {
11                int t2 = dp[j];
12                dp[j] = (s1[i - 1] == s2[j - 1]) ? t1 : min(dp[j] + s1[i - 1], dp[j - 1] +
13 s2[j - 1]);
14                t1 = t2;
15            }
16        }
17        return dp[n];
18    }
19 };

```

CPP

下面这种方法思路也很巧妙，反其道而行之，相当于先计算了字符串s1和s2的最大相同子序列，在这道题中就是最大相同子序列的ASCII码值，然后用s1和s2的所有字符之和减去这个最大ASCII码值的两倍，就是要删除的字符的最小ASCII码值了。那么还是建立二维数组dp，其中dp[i][j]表示字符串s1的前i个字符和字符串s2点前j个字符中的最大相同子序列的ASCII值。然后我们遍历所有的位置，当对应位置的字符相等时， $s1[i-1] == s2[j-1]$ ，那么我们直接赋值为上一个状态的dp值加上这个相同的字符，即 $dp[i-1][j-1] + s1[i-1]$ ，注意这里跟解法一不同之处，因为dp的定义不同，所以写法不同。如果 $s1[i-1] != s2[j-1]$ ，那么就有两种情况，我们可以删除s[i-1]的字符，即 $dp[i-1][j]$ ，或者删除s[j-1]的字符，即 $dp[i][j-1]$ ，取二者中最大值赋给 $dp[i][j]$ 。最后分别算出s1和s2的累加值，减去两倍的dp最大值即可，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int minimumDeleteSum(string s1, string s2) {
4         int m = s1.size(), n = s2.size();
5         vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
6         for (int i = 1; i <= m; ++i) {
7             for (int j = 1; j <= n; ++j) {
8                 if (s1[i - 1] == s2[j - 1]) dp[i][j] = dp[i - 1][j - 1] + s1[i - 1];
9                 else dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
10            }
11        }
12        int sum1 = accumulate(s1.begin(), s1.end(), 0);
13        int sum2 = accumulate(s2.begin(), s2.end(), 0);
14        return sum1 + sum2 - 2 * dp[m][n];
15    }
16 };

```

CPP

700. 子数组乘积小于K

Your are given an array of positive integers nums.

Count and print the number of (contiguous) subarrays where the product of all the elements in the subarray is less than k.

这道题给了我们一个数组和一个数字K，让我们求子数组且满足乘积小于K的个数。既然是子数组，那么必须是连续的，所以肯定不能给数组排序了，这道题好在限定了输入数字都是正数，能稍稍好做一点。博主刚开始用的是暴力搜索的方法来做的，就是遍历所有的子数组算乘积和K比较，两个for循环就行了，但是OJ不答应。于是上网搜大神们的解法，思路很赞。相当于是一种滑动窗口的解法，维护一个数字乘积刚好小于k的滑动窗口，用变量left来记录其左边界的位置，右边界i就是当前遍历到的位置。遍历原数组，用prod乘上当前遍历到的数字，然后进行while循环，如果prod大于等于k，则滑动窗口的左边界需要向右移动一位，删除最左边的数字，那么少了一个数字，乘积就会改变，所以用prod除以最左边的数字，然后左边右移一位，即left自增1。当我们确定了窗口的大小后，就可以统计子数组的个数了，就是窗口的大小。为啥呢，比如[5 2 6]这个窗口，k还是100，右边界刚滑到6这个位置，这个窗口的大小就是包含6的子数组乘积小于k的个数，即[6], [2 6], [5 2 6]，正好是3个。所以窗口每次向右增加一个数字，然后左边去掉需要去掉的数字后，窗口的大小就是新的子数组的个数，每次加到结果res中即可，参见代码如下：

```

1 class Solution {
2 public:
3     int numSubarrayProductLessThanK(vector<int>& nums, int k) {
4         if (k <= 1) return 0;
5         int res = 0, prod = 1, left = 0;
6         for (int i = 0; i < nums.size(); ++i) {
7             prod *= nums[i];
8             while (prod >= k) prod /= nums[left++];
9             res += i - left + 1;
10        }
11    }
12 }
13 };

```

CPP

701. 买股票的最佳时间含交易费

Your are given an array of integers prices, for which the i -th element is the price of a given stock on day i ; and a non-negative integer fee representing a transaction fee.

You may complete as many transactions as you like, but you need to pay the transaction fee for each transaction. You may not buy more than 1 share of a stock at a time (ie. you must sell the stock share before you buy again.)

Return the maximum profit you can make.

又是一道股票交易的题，之前已经有过类似的五道题了，fun4LeetCode大神的帖子做了amazing的归纳总结，有时间的话博主也写个总结。这道题跟Best Time to Buy and Sell Stock II其实最像，但是由于那道题没有交易费的限制，所以我们就无脑贪婪就可以了，见到利润就往上加。但是这道题有了交易费，所以当卖出的利润小于交易费的时候，我们就不应该卖了，不然亏了。所以这道题还是得用动态规划来做，按照fun4LeetCode大神的理论，本质其实是个三维dp数组，由于第三维只有两种情况，卖出和保留，而且第二维交易的次数在这道题中没有限制，所以我们用两个一维数组就可以了，sold[i]表示第*i*天卖掉股票此时的最大利润，hold[i]表示第*i*天保留手里的股票此时的最大利润。那么我们来分析递推公式，在第*i*天，如果我们要卖掉手中的股票，那么此时我们的总利润应该是前一天手里有股票的利润(不然没股票卖啊)，加上此时的卖出价格，减去交易费得到的利润总值，跟前一天卖出的利润相比，取其中较大值，如果前一天卖出的利润较大，那么我们就前一天卖了，不留到今天了。然后来看如果第*i*天不卖的利润，就是昨天股票卖了的利润然后今天再买入股票，得减去今天的价格，得到的值和昨天股票保留时的利润相比，取其中的较大值，如果昨天保留股票的利润大，那么我们就继续保留到今天，所以递推时可以得到：

```

sold[i] = max(sold[i - 1], hold[i - 1] + prices[i] - fee);

hold[i] = max(hold[i - 1], sold[i - 1] - prices[i]);

```

参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int maxProfit(vector<int>& prices, int fee) {
4         vector<int> sold(prices.size(), 0), hold = sold;
5         hold[0] = -prices[0];
6         for (int i = 1; i < prices.size(); ++i) {
7             sold[i] = max(sold[i - 1], hold[i - 1] + prices[i] - fee);
8             hold[i] = max(hold[i - 1], sold[i - 1] - prices[i]);
9         }
10        return sold.back();
11    }
12 };

```

我们发现不管是卖出还是保留，第*i*天到利润只跟第*i*-1天有关系，所以我们可以优化空间，用两个变量来表示当前的卖出和保留的利润，更新方法和上面的基本相同，就是开始要保存sold的值，不然sold先更新后，再更新hold时就没能使用更新前的值了，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int maxProfit(vector<int>& prices, int fee) {
4         int sold = 0, hold = -prices[0];
5         for (int price : prices) {
6             int t = sold;
7             sold = max(sold, hold + price - fee);
8             hold = max(hold, t - price);
9         }
10        return sold;
11    }
12 };

```

702. 范围模块

A Range Module is a module that tracks ranges of numbers. Your task is to design and implement the following interfaces in an efficient manner.

`addRange(int left, int right)` Adds the half-open interval [left, right), tracking every real number in that interval. Adding an interval that partially overlaps with currently tracked numbers should add any numbers in the interval [left, right) that are not already tracked.

`queryRange(int left, int right)` Returns true if and only if every real number in the interval [left, right) is currently being tracked.

`removeRange(int left, int right)` Stops tracking every real number currently being tracked in the interval [left, right).

这道题让我们实现一个RangeModule的类，里面有三个功能函数，分别好似插入范围，查找范围，删除范围，题目中的例子给的也很明确，基本不会引起什么歧义。其实不管范围也好，区间也好，都是一回事，跟之前的区间的题目Insert Interval和Merge Intervals没有什么不同。这里的插入范围函数的实现方法跟之前那道Insert Interval的解法一样，直接抄过来就好。然后对于查找范围函数，由于题目中说只要有数字未被包括，就返回false。那么我们反过来想，只有当某个范围完全覆盖了这个要查找的范围才会返回true，所以我们可以遍历所有的范围，然后看是否有一个范围完全覆盖了要查找的范围，有的话返回true，循环结束后返回false。最后来看删除范围函数，其实现方法跟插入范围函数很类似，但又有少许不同。首先我们还是新建一个数组res存结果，然后遍历已有的范围，如果当前范围的结束位置小于等于要删除的范围的起始位置，由于题目中的范围定义是左

开右闭，那么说明没有重叠，加入结果res，并且用变量cur自增1来记录当前位置。如果当前范围的起始位置大于等于要删除的范围的结束位置，说明咩有重叠，加入结果res。否则就是有重叠的情况，这里跟插入范围有所不同的是，插入范围只需要加入一个范围，而删除范围操作有可能将原来的大范围break成为两个小的范围，所以我们用一个临时数组t来存break掉后的小范围。如果当前范围的起始位置小于要删除的范围的起始位置left，说明此时一定有一段范围留下来了，即从当前范围的起始位置到要删除的范围的起始位置left，将这段范围加入临时数组t，同理，如果当前范围的结束位置大于要删除的范围的结束位置right，将这段范围加入临时数组t。最后将数组t加入结果res中的cur位置即可，参见代码如下：

解法1：

```
1 class RangeModule {
2 public:
3     RangeModule() {}
4
5     void addRange(int left, int right) {
6         vector<pair<int, int>> res;
7         int n = v.size(), cur = 0;
8         for (int i = 0; i < n; ++i) {
9             if (v[i].second < left) {
10                 res.push_back(v[i]);
11                 ++cur;
12             } else if (v[i].first > right) {
13                 res.push_back(v[i]);
14             } else {
15                 left = min(left, v[i].first);
16                 right = max(right, v[i].second);
17             }
18         }
19         res.insert(res.begin() + cur, {left, right});
20         v = res;
21     }
22
23     bool queryRange(int left, int right) {
24         for (auto a : v) {
25             if (a.first <= left && a.second >= right) return true;
26         }
27         return false;
28     }
29
30     void removeRange(int left, int right) {
31         vector<pair<int, int>> res, t;
32         int n = v.size(), cur = 0;
33         for (int i = 0; i < n; ++i) {
34             if (v[i].second <= left) {
35                 res.push_back(v[i]);
36                 ++cur;
37             } else if (v[i].first >= right) {
38                 res.push_back(v[i]);
39             } else {
40                 if (v[i].first < left) {
41                     t.push_back({v[i].first, left});
42                 }
43                 if (v[i].second > right) {
44                     t.push_back({right, v[i].second});
45                 }
46             }
47         }
48         res.insert(res.begin() + cur, t.begin(), t.end());
49         v = res;
50     }
51
52 private:
53     vector<pair<int, int>> v;
54 }
```

下面来看一种优化了时间复杂度的解法，我们使用TreeMap来建立范围的起始位置和结束位置之间的映射，利用了TreeMap的自动排序功能，其会根据起始位置从小到大进行排序。既然是有序的，我们就可以利用二分法来快速进行查找了。

在加入范围函数中，我们首先用upper_bound函数来查找第一个大于left的位置，标记为l，再用upper_bound函数来查找第一个大于right的位置，标记为r。我们其实是想找第一个不大于left和right的位置的，由于C++没有floorKey这个函数，所以我们只能用upper_bound找大于left和right的位置，然后再往前移一个。如果l不是TreeMap中的第一个位置，且前面一个范围的结束位置小于left，说明和前一个范围没有交集，那么还是回到当前这个范围吧。如果此时l和r指向同一个位置，说明当前要加入的范围没有跟其他任何一个范围有交集，所以我们直接返回即可，不需要其他任何操作。否则的话我们将left和l指向范围的起始位置中的较小值赋给i，将right和r指向的前一个位置的结束位置的较大值赋给j，然后将l和r之间的范围都删除掉（注意这里r自增了1，是因为之前先自减了1），然后将i和j返回即可。返回后我们建立起这个映射即可。

在查找范围函数中，我们先用upper_bound找出第一个大于left位置的范围it，然后看如果it不是第一个范围，并且如果其前面的一个范围的结束位置大于等于right，说明已经完全包括这个要查找的范围，因为前一个范围的起始位置小于left，且结束位置大于等于right，直接返回true。

在删除范围函数中，查找重叠范围的方法跟加入范围函数中的操作一样，所以抽出来放到了find函数中，由于删除的范围有可能完全覆盖了原有范围，也有可能只是部分覆盖，将一个大的范围拆成了一个或者两个范围。所以我们判断，如果left大于覆盖范围的起始位置，那么将这段建立映射，同理，如果覆盖范围的结束位置大于right，同样建立这段的映射，参见代码如下：

解法2：

```

1 class RangeModule {
2 public:
3     RangeModule() {}
4
5     void addRange(int left, int right) {
6         auto x = find(left, right);
7         m[x.first] = x.second;
8     }
9
10    bool queryRange(int left, int right) {
11        auto it = m.upper_bound(left);
12        return it != m.begin() && (--it)->second >= right;
13    }
14
15    void removeRange(int left, int right) {
16        auto x = find(left, right);
17        if (left > x.first) m[x.first] = left;
18        if (x.second > right) m[right] = x.second;
19    }
20
21 private:
22     map<int, int> m;
23
24     pair<int, int> find(int left, int right) {
25         auto l = m.upper_bound(left), r = m.upper_bound(right);
26         if (l != m.begin() && (--l)->second < left) ++l;
27         if (l == r) return {left, right};
28         int i = min(left, l->first), j = max(right, (--r)->second);
29         m.erase(l, ++r);
30         return {i, j};
31     }
32 };

```

703. 最大栈

Design a max stack that supports push, pop, top, peekMax and popMax.

```
push(x) -- Push element x onto stack.  
pop() -- Remove the element on top of the stack and return it.  
top() -- Get the element on the top.  
peekMax() -- Retrieve the maximum element in the stack.  
popMax() -- Retrieve the maximum element in the stack, and remove it. If you find more than one  
maximum elements, only remove the top-most one.
```

这道题让我们实现一个最大栈，包含一般栈的功能，但是还新加了两个功能peekMax()和popMax()，随时随地可以查看和返回最大值。之前有一道很类似的题Min Stack，所以我们借鉴那道题的解法，使用两个栈来模拟，s1为普通的栈，用来保存所有的数字，而s2为最大栈，用来保存出现的最大的数字。

在push()函数中，我们先来看s2，如果s2为空，或者s2的栈顶元素小于等于x，将x压入s2中。因为s2保存的是目前为止最大的数字，所以一旦新数字大于等于栈顶元素，说明遇到更大的数字了，压入栈。然后将数组压入s1，s1保存所有的数字，所以都得压入栈。

在pop()函数中，当s2的栈顶元素和s1的栈顶元素相同时，我们要移除s2的栈顶元素，因为一个数字不在s1中了，就不能在s2中。然后取出s1的栈顶元素，并移除s1，返回即可。

在top()函数中，直接返回s1的top()函数即可。

在peekMax()函数中，直接返回s2的top()函数即可。

在popMax()函数中，先将s2的栈顶元素保存到一个变量mx中，然后我们要在s1中删除这个元素，由于栈无法直接定位元素，所以我们用一个临时栈t，将s1的出栈元素保存到临时栈t中，当s1的栈顶元素和s2的栈顶元素相同时退出while循环，此时我们在s1中找到了s2的栈顶元素，分别将s1和s2的栈顶元素移除，然后要做的是将临时栈t中的元素加回s1中，注意此时容易犯的一个错误是，没有同时更新s2，所以我们直接调用push()函数即可，参见代码如下：

解法1：

```

1 class MaxStack {
2 public:
3     /** initialize your data structure here. */
4     MaxStack() {}
5
6     void push(int x) {
7         if (s2.empty() || s2.top() <= x) s2.push(x);
8         s1.push(x);
9     }
10
11    int pop() {
12        if (!s2.empty() && s2.top() == s1.top()) s2.pop();
13        int t = s1.top(); s1.pop();
14        return t;
15    }
16
17    int top() {
18        return s1.top();
19    }
20
21    int peekMax() {
22        return s2.top();
23    }
24
25    int popMax() {
26        int mx = s2.top();
27        stack<int> t;
28        while (s1.top() != s2.top()) {
29            t.push(s1.top()); s1.pop();
30        }
31        s1.pop(); s2.pop();
32        while (!t.empty()) {
33            push(t.top()); t.pop();
34        }
35        return mx;
36    }
37
38 private:
39     stack<int> s1, s2;
40 };

```

下面这种解法没有利用一般的stack，而是建立一种较为复杂的数据结构，首先用一个list链表来保存所有的数字，然后建立一个数字和包含所有相同的数字的位置iterator的向量容器的映射map。

在push()函数中，把新数字加到list表头，然后把数字x的位置iterator加到数字映射的向量容器的末尾。

在pop()函数中，先得到表头数字，然后把该数字对应的iterator向量容器的末尾元素删掉，如果此时向量容器为空了，将这个映射直接删除，移除表头数字，返回该数字即可。

在top()函数中，直接返回表头数字即可。

在peekMax()函数中，因为map是按key值自动排序的，直接尾映射的key值即可。

在popMax()函数中，首先保存尾映射的key值，也就是最大值到变量x中，然后在其对应的向量容器的末尾取出其在list中的iterator。然后删除该向量容器的尾元素，如果此时向量容器为空了，将这个映射直接删除。根据之前取出的iterator，在list中删除对应的数字，返回x即可，参见代码如下：

解法2：

```

1  class MaxStack {
2  public:
3      /** initialize your data structure here. */
4      MaxStack() {}
5
6      void push(int x) {
7          v.insert(v.begin(), x);
8          m[x].push_back(v.begin());
9      }
10
11     int pop() {
12         int x = *v.begin();
13         m[x].pop_back();
14         if (m[x].empty()) m.erase(x);
15         v.erase(v.begin());
16         return x;
17     }
18
19     int top() {
20         return *v.begin();
21     }
22
23     int peekMax() {
24         return m.rbegin()->first;
25     }
26
27     int popMax() {
28         int x = m.rbegin()->first;
29         auto it = m[x].back();
30         m[x].pop_back();
31         if (m[x].empty()) m.erase(x);
32         v.erase(it);
33         return x;
34     }
35
36 private:
37     list<int> v;
38     map<int, vector<list<int>::iterator>> m;
39 };

```

704. 一位和两位字符

We have two special characters. The first character can be represented by one bit 0. The second character can be represented by two bits (10 or 11).

Now given a string represented by several bits. Return whether the last character must be a one-bit character or not. The given string will always end with a zero.

这道题说有两种特殊的字符，一种是两位字符，只能是二进制的11和10，另一种是单个位字符，只能是二进制的0。现在给了我们一个只包含0和1的数组，问我们能否将其正确的分割，使得最后一个字符是个单个位字符。这道题可以使用贪心算法来做，因为两种字符互不干扰，只要我们遍历到了数字1，那么其必定是两位字符，所以后面一位也得跟着，而遍历到了数字0，那么就必定是单个位字符。所以我们可以用一个变量i来记录当前遍历到的位置，如果遇到了0，那么i自增1，如果遇到了1，那么i自增2，我们循环的条件是*i < n-1*，即留出最后一位，所以当循环退出后，当i正好停留在n-1上，说明最后一位是单独分割开的，因为题目中限定了最后一位一定是0，所以没必要再判断了，参见代码如下：

解法1:

```
1 class Solution {
2 public:
3     bool isOneBitCharacter(vector<int>& bits) {
4         int n = bits.size(), i = 0;
5         while (i < n - 1) {
6             if (bits[i] == 0) ++i;
7             else i += 2;
8         }
9         return i == n - 1;
10    }
11};
```

CPP

下面这种解法写的更加简洁了，直接用一行代替了if..else..语句，相当巧妙，当bits[i]为0时，i还是相当于自增了1，当bits[i]为1时，i相当于自增了2，最后还是在循环跳出后检测i是否为n-1，参见代码如下：

解法2:

```
1 class Solution {
2 public:
3     bool isOneBitCharacter(vector<int>& bits) {
4         int n = bits.size(), i = 0;
5         while (i < n - 1) {
6             i += bits[i] + 1;
7         }
8         return i == n - 1;
9     }
10};
```

CPP

下面我们来看递归解法，用的是回溯的思想，首先判断如果bits为空了，直接返回false，因为题目初始给的bits是非空的，在调用递归函数中为空了说明最后一位跟倒数第二位组成了个两位字符，所以不合题意返回false。再判断如果bits大小为1了，那么返回这个数字是否为0，其实直接返回true也行，因为题目中说了最后一个数字一定是0。然后我们新建一个数组t，如果bits的首元素为0，则我们的t赋值为去掉首元素的bits数组；如果bits的首元素是1，则我们的t赋之为去掉前两个元素的bits数组，然后返回调用递归函数的结果即可，参见代码如下

解法3:

```
1 class Solution {
2 public:
3     bool isOneBitCharacter(vector<int>& bits) {
4         if (bits.empty()) return false;
5         if (bits.size() == 1) return bits[0] == 0;
6         vector<int> t;
7         if (bits[0] == 0) {
8             t = vector<int>(bits.begin() + 1, bits.end());
9         } else if (bits[0] == 1) {
10            t = vector<int>(bits.begin() + 2, bits.end());
11        }
12        return isOneBitCharacter(t);
13    }
14};
```

CPP

下面这种解法也是用的递归，递归函数用的不是原函数，这样可以只用位置变量idx来遍历，而不用新建数组t，初始时idx传入0，在递归函数中，如果idx为n了，相当于上面解法中的bits数组为空了情况，返回false；如果idx为n-1，返回true；如果bits[idx]为0，则返回调用递归函数的结果，此时idx加上1；如果bits[idx]为1，则返回调用递归函数的结果，此时idx加上2，参见代码如下：

解法4：

```
1 class Solution {
2 public:
3     bool isOneBitCharacter(vector<int>& bits) {
4         return helper(bits, 0);
5     }
6     bool helper(vector<int>& bits, int idx) {
7         int n = bits.size();
8         if (idx == n) return false;
9         if (idx == n - 1) return bits[idx] == 0;
10        if (bits[idx] == 0) return helper(bits, idx + 1);
11        return helper(bits, idx + 2);
12    }
13};
```

CPP

705. 最长的重复子数组

Given two integer arrays A and B, return the maximum length of an subarray that appears in both arrays.

这道题给了我们两个数组A和B，让我们返回连个数组的最长重复子数组。那么如果我们将数组换成字符串，实际这道题就是求Longest Common Substring的问题了，而貌似LeetCode上并没有这种明显的要求最长相同子串的题，注意需要跟最长子序列Longest Common Subsequence区分开，关于最长子序列会在follow up中讨论。好，先来看这道题，对于这种求极值的问题，DP是不二之选，我们使用一个二维的DP数组，其中dp[i][j]表示数组A的前i个数字和数组B的前j个数字的最长子数组的长度，如果dp[i][j]不为0，则A中第i个数组和B中第j个数字必须相等，比对于这两个数组[1, 2, 2]和[3, 1, 2]，我们的dp数组为：

```
3 1 2
1 0 1 0
2 0 0 2
2 0 0 1
```

我们注意观察，dp值不为0的地方，都是当A[i] == B[j]的地方，而且还要加上左上方的dp值，即dp[i-1][j-1]，所以当前的dp[i][j]就等于dp[i-1][j-1] + 1，而一旦A[i] != B[j]时，直接赋值为0，不用多想，因为子数组是要连续的，一旦不匹配了，就不能再增加长度了。我们每次算出一个dp值，都要用来更新结果res，这样就能得到最长相同子数组的长度了，参见代码如下：

```

1 class Solution {
2 public:
3     int findLength(vector<int>& A, vector<int>& B) {
4         int res = 0;
5         vector<vector<int>> dp(A.size() + 1, vector<int>(B.size() + 1, 0));
6         for (int i = 1; i < dp.size(); ++i) {
7             for (int j = 1; j < dp[i].size(); ++j) {
8                 dp[i][j] = (A[i - 1] == B[j - 1]) ? dp[i - 1][j - 1] + 1 : 0;
9                 res = max(res, dp[i][j]);
10            }
11        }
12        return res;
13    }
14 };

```

706. 找第K小的数对儿距离

Given an integer array, return the k-th smallest distance among all the pairs. The distance of a pair (A, B) is defined as the absolute difference between A and B.

这道题给了我们一个数组，让我们找第k小的数对儿距离，数对儿距离就是任意两个数字之间的绝对值差。那么我们先来考虑最暴力的解法，是不是就是遍历任意两个数字，算出其绝对值差，然后将所有距离排序，取第k小的就行了。But，OJ摇着头说图样图森破。但是我们可以在纯暴力搜索的基础上做些优化，从而让OJ说YES。那么下面这种利用了桶排序的解法就是一种很好的优化，题目中给了数字的大小范围，不会超过一百万，所以我们就建立一百万个桶，然后还是遍历任意两个数字，将计算出的距离放到对应的桶中，这里桶不是存的具体距离，而是该距离出现的次数，桶本身的位置就是距离，所以我们才建立了一百万个桶。然后我们就可以从0开始遍历到一百万了，这样保证了我们先处理小距离，如果某个距离的出现次数大于等于k了，那么我们返回这个距离，否则就用k减去这个距离的出现次数，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int smallestDistancePair(vector<int>& nums, int k) {
4         int n = nums.size(), N = 1000000;
5         vector<int> cnt(N, 0);
6         for (int i = 0; i < n; ++i) {
7             for (int j = i + 1; j < n; ++j) {
8                 ++cnt[abs(nums[i] - nums[j])];
9             }
10        }
11        for (int i = 0; i < N; ++i) {
12            if (cnt[i] >= k) return i;
13            k -= cnt[i];
14        }
15        return -1;
16    }
17 };
18

```

上面的解法虽然逃脱了OJ的魔掌，但也仅仅是险过，并不高效。我们来看一种基于二分搜索的解法。这道题使用的二分搜索法是博主归纳总结帖LeetCode Binary Search Summary 二分搜索法小结中的第四种，即二分法的判定条件不是简单的大小关系，而是可以抽离出子函数的情况，下面我们来看具体怎么弄。我们的目标是快速定位出第k小的距离，那么很适合用二分法来快速的缩小查找范围，然而最大的难点就是如何找到判定依据来折半查找，即如果确定搜索目标是在左半边还是右半边。做过Kth Smallest Element in a Sorted Matrix和Kth Smallest Number in Multiplication Table这两道题的同学应该对这种搜索方式并不陌生。核心思想是二分确定一个中间数，然后找到所有小于等于这个中间数的距离个数，用其跟k比较来确定折半的方向。具体的操作是，我们首先要给数组排序，二分搜索的起始left为0，结束位置right为最大距离，即排序后的数字最后一个元素减去首元素。然后进入while循环，算出中间值mid，此外我们还需要两个变量cnt和start，其中cnt是记录小于等于mid的距离个数，start是较小数字的位置，均初始化为0，然后我们遍历整个数组，先进行while循环，如果start未越界，并且当前数字减去start指向的数组之差大于mid，说明此时距离太大了，我们增加减数大小，通过将start右移一个，那么while循环退出后，就有 $i - start$ 个距离小于等于mid，将其加入cnt中，举个栗子来说：

```
1   2   3   3   5
start           i
mid = 2
```

如果start在位置0，i在位置3，那么以nums[i]为较大数可以产生三个($i - start$)小于等于mid的距离，[1 3]，[2 3]，[3 3]，这样当i遍历完所有的数字后，所有小于等于mid的距离的个数就求出来了，即cnt。然后我们跟k比较，如果其小于k，那么left赋值为mid+1，反之，则right赋值为mid。最终返回right或left均可，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     int smallestDistancePair(vector<int>& nums, int k) {
4         sort(nums.begin(), nums.end());
5         int n = nums.size(), left = 0, right = nums.back() - nums[0];
6         while (left < right) {
7             int mid = left + (right - left) / 2, cnt = 0, start = 0;
8             for (int i = 0; i < n; ++i) {
9                 while (start < n && nums[i] - nums[start] > mid) ++start;
10                cnt += i - start;
11            }
12            if (cnt < k) left = mid + 1;
13            else right = mid;
14        }
15        return right;
16    }
17};
```

CPP

707. 字典中的最长单词

Given a list of strings words representing an English Dictionary, find the longest word in words that can be built one character at a time by other words in words. If there is more than one possible answer, return the longest word with the smallest lexicographical order.

If there is no answer, return the empty string.

这道题给了我们一个字典，是个字符串数组，然后问我们从单个字符开始拼，最长能组成啥单词，注意中间生成的字符串也要在字典中存在，而且当组成的单词长度相等时，返回字母顺序小的那个。好，看到这么多前缀一样多字符串，是不是很容易想到用前缀树来做，其实我们并不需要真正的建立前缀树结点，可以借鉴查找的思想来做。那么为了快速的查找某个单词是否在字典中存在，我们将所有单词放到哈希集合中，在查找的时候，可以采用BFS或者DFS都行。先来看BFS的做法，使用一个queue来辅助，我们先把所有长度为1的单词找出排入queue中，当作种子选手，然后我们进行循环，每次从队首取一个元素出来，如果其

长度大于我们维护的最大值mxLen，则更新mxLen和结果res，如果正好相等，也要更新结果res，取字母顺序小的那个。然后我们试着增加长度，做法就是遍历26个字母，将每个字母都加到单词后面，然后看是否在字典中存在，存在的话，就加入queue中等待下一次遍历，完了以后记得要恢复状态，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     string longestWord(vector<string>& words) {
4         string res = "";
5         int mxLen = 0;
6         unordered_set<string> s(words.begin(), words.end());
7         queue<string> q;
8         for (string word : words) {
9             if (word.size() == 1) q.push(word);
10        }
11        while (!q.empty()) {
12            string t = q.front(); q.pop();
13            if (t.size() > mxLen) {
14                mxLen = t.size();
15                res = t;
16            } else if (t.size() == mxLen) {
17                res = min(res, t);
18            }
19            for (char c = 'a'; c <= 'z'; ++c) {
20                t.push_back(c);
21                if (s.count(t)) q.push(t);
22                t.pop_back();
23            }
24        }
25        return res;
26    }
27 };

```

CPP

下面来看递归的解法，前面都一样，不同在于直接对长度为1的单词调用递归函数，在递归中，还是先判断单词和mxLen关系来更新结果res，然后就是遍历所有字符，加到单词后面，如果在集合中存在，调用递归函数，结束后恢复状态，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     string longestWord(vector<string>& words) {
4         string res = "";
5         int mxLen = 0;
6         unordered_set<string> s(words.begin(), words.end());for (string word : words) {
7             if (word.size() == 1) helper(s, word, mxLen, res);
8         }
9         return res;
10    }
11    void helper(unordered_set<string>& s, string word, int& mxLen, string& res) {
12        if (word.size() > mxLen) {
13            mxLen = word.size();
14            res = word;
15        } else if (word.size() == mxLen) {
16            res = min(res, word);
17        }
18        for (char c = 'a'; c <= 'z'; ++c) {
19            word.push_back(c);
20            if (s.count(word)) helper(s, word, mxLen, res);
21            word.pop_back();
22        }
23    }
24 };

```

下面这种解法是论坛上的高分解法，其实我们只要给数组排个序，就可以使用贪心算法来做了，并不需要什么DFS或BFS这么复杂。首先建立一个空的哈希set，然后我们直接遍历排序后的字典，对于当前的单词，如果当前单词长度为1，或者该单词去掉最后一个字母后在集合中存在，这也不难理解，长度为1，说明是起始单词，不需要的多余的判断，否则的话就要判断其去掉最后一个字母后的单词是否在集合中存在，存在的话，才说明其中间单词都存在，因为此时是从短单词向长单词遍历，只要符合要求的才会加入集合，所以一旦其去掉尾字母的单词存在的话，那么其之前所有的中间情况都会在集合中存在。我们更新结果res时，要判断当前单词长度是否大于结果res的长度，因为排序过后，默认先更新的字母顺序小的单词，所有只有当当前单词长度大，才更新结果res，之后别忘了把当前单词加入集合中，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     string longestWord(vector<string>& words) {
4         string res = "";
5         unordered_set<string> s;
6         sort(words.begin(), words.end());
7         for (string word : words) {
8             if (word.size() == 1 || s.count(word.substr(0, word.size() - 1))) {
9                 res = (word.size() > res.size()) ? word : res;
10                s.insert(word);
11            }
12        }
13        return res;
14    }
15 };

```

708. 账户合并

Given a list accounts, each element accounts[i] is a list of strings, where the first element accounts[i][0] is a name, and the rest of the elements are emails representing emails of the account.

Now, we would like to merge these accounts. Two accounts definitely belong to the same person if there is some email that is common to both accounts. Note that even if two accounts have the same name, they may belong to different people as people could have the same name. A person can have any number of accounts initially, but all of their accounts definitely have the same name.

After merging the accounts, return the accounts in the following format: the first element of each account is the name, and the rest of the elements are emails in sorted order. The accounts themselves can be returned in any order.

这道题给了我们一堆人名和邮箱，一个名字可能有多个邮箱，但是一个邮箱只属于一个人，让我们把同一个人的邮箱都合并到一起，名字相同不一定是同一个人，只有当两个名字有相同的邮箱，才能确定是同一个人，题目中的例子很好说明了这个问题，输入有三个John，最后合并之后就只有两个了。这道题博主最开始尝试使用贪婪算法来做，结果发现对于下面这个例子不适用：

```
["John", "a@gmail.com", "b@gmail.com"]
```

```
["John", "c@gmail.com", "d@gmail.com"]
```

```
["John", "a@gmail.com", "c@gmail.com"]
```

我们可以看到其实这三个John是同一个人，但是贪婪算法遍历完前两个John，还是认为其是两个不同的人，当遍历第三个John时，就直接加到第一个John中了，而没有同时把第二个John加进来，也可能博主写的是假的贪婪算法，反正不管了，还是参考大神们的解法吧。这个归组类的问题，最典型的就是岛屿问题(例如Number of Islands II)，很适合使用Union Find来做，LeetCode中有很多道可以使用这个方法来做的题，比如Friend Circles, Graph Valid Tree, Number of Connected Components in an Undirected Graph, 和Redundant Connection等等。都是要用一个root数组，每个点开始初始化为不同的值，如果两个点属于相同的组，就将其中一个点的root值赋值为另一个点的位置，这样只要是相同组里的两点，通过find函数得到相同的值。在这里，由于邮件是字符串不是数字，所以root可以用哈希map来代替，我们还需要一个哈希映射owner，建立每个邮箱和其所有者姓名之前的映射，另外用一个哈希映射来建立用户和其所有的邮箱之间的映射，也就是合并后的结果。

首先我们遍历每个账户和其中的所有邮箱，先将每个邮箱的root映射为其自身，然后将owner赋值为用户名。然后开始另一个循环，遍历每一个账号，首先对帐号的第一个邮箱调用find函数，得到其父串p，然后遍历之后的邮箱，对每个遍历到的邮箱先调用find函数，将其父串的root值赋值为p，这样做相当于将相同账号内的所有邮箱都链接起来了。我们下来要做的就是再次遍历每个账户内的所有邮箱，先对该邮箱调用find函数，找到父串，然后将该邮箱加入该父串映射的集合汇总，这样就我们就完成了合并。最后只需要将集合转为字符串数组，加入结果res中，通过owner映射找到父串的用户名，加入字符串数组的首位置，参见代码如下：

解法1:

```
1 class Solution {
2 public:
3     vector<vector<string>> accountsMerge(vector<vector<string>>& accounts) {
4         vector<vector<string>> res;
5         unordered_map<string, string> root;
6         unordered_map<string, string> owner;
7         unordered_map<string, set<string>> m;
8         for (auto account : accounts) {
9             for (int i = 1; i < account.size(); ++i) {
10                 root[account[i]] = account[i];
11                 owner[account[i]] = account[0];
12             }
13         }
14         for (auto account : accounts) {
15             string p = find(account[1], root);
16             for (int i = 2; i < account.size(); ++i) {
17                 root[find(account[i], root)] = p;
18             }
19         }
20         for (auto account : accounts) {
21             for (int i = 1; i < account.size(); ++i) {
22                 m[find(account[i], root)].insert(account[i]);
23             }
24         }
25         for (auto a : m) {
26             vector<string> v(a.second.begin(), a.second.end());
27             v.insert(v.begin(), owner[a.first]);
28             res.push_back(v);
29         }
30         return res;
31     }
32     string find(string s, unordered_map<string, string>& root) {
33         return root[s] == s ? s : find(root[s], root);
34     }
35 };
```

下面这种方法是使用BFS来解的，建立了每个邮箱和其所有出现的账户数组之间的映射，比如还是这个例子：

```
["John", "a@gmail.com", "b@gmail.com"]
```

```
["John", "c@gmail.com", "d@gmail.com"]
```

```
["John", "a@gmail.com", "c@gmail.com"]
```

那么建立的映射就是：

```
"a@gmail.com" -> [0, 2]
```

```
"b@gmail.com" -> [0]
```

```
"c@gmail.com" -> [1, 2]
```

```
"d@gmail.com" -> [1]
```

然后我们还需要一个visited数组，来标记某个账户是否已经被遍历过，0表示为未访问，1表示已访问。在建立好哈希map之后，我们遍历所有的账户，如果账户未被访问过，将其加入队列queue，新建一个集合set，此时进行队列不为空的while循环，取出队首账户，将该账户标记已访问1，此时将该账户的所有邮箱取出来放入数组mails中，然后遍历mails中的每一个邮箱，将遍历到的邮箱加入集合set中，根据映射来找到该邮箱所属的所有账户，如果该账户未访问，则加入队列中并标记已访问。当while循环结束后，当前账户的所有合并后的邮箱都保存在集合set中，将其转为字符串数组，并且加上用户名在首位置，最后加入结果res中即可，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     vector<vector<string>> accountsMerge(vector<vector<string>>& accounts) {
4         vector<vector<string>> res;
5         int n = accounts.size();
6         unordered_map<string, vector<int>> m;
7         vector<int> visited(n, 0);
8         for (int i = 0; i < n; ++i) {
9             for (int j = 1; j < accounts[i].size(); ++j) {
10                 m[accounts[i][j]].push_back(i);
11             }
12         }
13         for (int i = 0; i < n; ++i) {
14             if (visited[i] != 0) continue;
15             queue<int> q{{i}};
16             set<string> s;
17             while (!q.empty()) {
18                 int t = q.front(); q.pop();
19                 visited[t] = 1;
20                 vector<string> mails(accounts[t].begin() + 1, accounts[t].end());
21                 for (string mail : mails) {
22                     s.insert(mail);
23                     for (int user : m[mail]) {
24                         if (visited[user] != 0) continue;
25                         q.push(user);
26                         visited[user] = 1;
27                     }
28                 }
29             }
30             vector<string> out(s.begin(), s.end());
31             out.insert(out.begin(), accounts[i][0]);
32             res.push_back(out);
33         }
34         return res;
35     }
36 };
```

709. 移除注释

Given a C++ program, remove comments from it. The program source is an array where `source[i]` is the i -th line of the source code. This represents the result of splitting the original source code string by the newline character `\n`.

In C++, there are two types of comments, line comments, and block comments.

The string `//` denotes a line comment, which represents that it and rest of the characters to the right of it in the same line should be ignored.

The string `/*` denotes a block comment, which represents that all characters until the next (non-overlapping) occurrence of `*/` should be ignored. (Here, occurrences happen in reading order: line by line from left to right.) To be clear, the string `/*` does not yet end the block comment, as the ending would be overlapping the beginning.

The first effective comment takes precedence over others: if the string `//` occurs in a block comment, it is ignored. Similarly, if the string `/*` occurs in a line or block comment, it is also ignored.

If a certain line of code is empty after removing comments, you must not output that line: each string in the answer list will be non-empty.

There will be no control characters, single quote, or double quote characters. For example, `source = "string s = /* Not a comment. */;"` will not be a test case. (Also, nothing else such as defines or macros will interfere with the comments.)

It is guaranteed that every open block comment will eventually be closed, so `/*` outside of a line or block comment always starts a new comment.

Finally, implicit newline characters can be deleted by block comments. Please see the examples below for details.

After removing the comments from the source code, return the source code in the same format.

这道题让我们移除代码中的注释部分，就是写代码中经常遇到的两种注释，单行注释和多行注释，也可以叫块注释，当然最重要就是要找到这两种注释的起始标识符`"//"`和`"/"`，注意它们两者之间存在覆盖的关系，谁在前面谁**work**，比如`"//abc/"`，那么此时后面的块注释起始符被忽略掉，同样`"/abc//"`，后面的单行注释起始符也不起作用，所以两者之间的前后顺序很重要。博主刚开始想的方法是用**string**的**find**函数来分别找`"//"`和`"/"`的起始位置，如果不存在就返回-1，但是需要分多种情况来处理，其是否存在，还有二者的前后顺序，处理起来比较麻烦。起始我们可以直接按字符来一个一个处理，由于块注释是多行注释，所以一旦之前有了块注释的起始符，当前行的处理方式就有所不同了，所以我们需要一个变量**blocked**来记录当前是否为块注释状态，初始化为**false**。建立空字符**out**，用来保存去除注释后的字符。然后我们遍历整个代码的每一行，遍历每一行中的每一个字符，如果当前字符是最后一个字符了，说明不会再有注释了，将当前字符加入**out**中，否则取出当前位置和下一个位置的两个字符，如果其正好是`"/"`，说明之后的部分都是块注释了，我们将**blocked**赋值为**true**，然后指针向后移动一个，明明两个字符啊，为啥只移动一个呢，因为另一个可以在**for**循环中的`++i`移动；如果当前两个字符正好是`"//"`，说明当前行之后都是注释，我们并不**care**后面有啥，所以可以直接**break**掉当前行；如果不是，说明当前字符是代码，将其加入**out**中。好，下面来看**blocked**为**true**的情况，说明之后的内容都是块注释的内容，我们唯一关心的是有没有结束符`"/"`，所以还是先做判断，如果当前不是最后一个字符，说明至少还有两个字符，然后取出两个字符，如果正好是块注释结束符，那么我们将标识重置为**false**，指针要后移动一个。当前行遍历完后，如果**out**不为空，且**blocked**为**false**，则将**out**存入结果**res**中，参见代码如下：

```

1 class Solution {
2 public:
3     vector<string> removeComments(vector<string>& source) {
4         vector<string> res;
5         bool blocked = false;
6         string out = "";
7         for (string line : source) {
8             for (int i = 0; i < line.size(); ++i) {
9                 if (!blocked) {
10                     if (i == line.size() - 1) out += line[i];
11                 else {
12                     string t = line.substr(i, 2);
13                     if (t == "/*") blocked = true, ++i;
14                     else if (t == "*/") break;
15                     else out += line[i];
16                 }
17             } else {
18                 if (i < line.size() - 1) {
19                     string t = line.substr(i, 2);
20                     if (t == "*/") blocked = false, ++i;
21                 }
22             }
23         }
24         if (!out.empty() && !blocked) {
25             res.push_back(out);
26             out = "";
27         }
28     }
29     return res;
30 }
31 };

```

710. 糖果消消乐

This question is about implementing a basic elimination algorithm for Candy Crush.

Given a 2D integer array board representing the grid of candy, different positive integers board[i][j] represent different types of candies. A value of board[i][j] = 0 represents that the cell at position (i, j) is empty. The given board represents the state of the game following the player's move. Now, you need to restore the board to a stable state by crushing candies according to the following rules:

If three or more candies of the same type are adjacent vertically or horizontally, "crush" them all at the same time - these positions become empty.

After crushing all candies simultaneously, if an empty space on the board has candies on top of itself, then these candies will drop until they hit a candy or bottom at the same time. (No new candies will drop outside the top boundary.)

After the above steps, there may exist more candies that can be crushed. If so, you need to repeat the above steps.

If there does not exist more candies that can be crushed (ie. the board is stable), then return the current board.

You need to perform the above rules until the board becomes stable, then return the current board.

这道题就是糖果消消乐，博主刚开始做的时候，没有看清楚题意，以为就像游戏中的那样，每次只能点击一个地方，然后消除后糖果落下，这样会导致一个问题，就是原本其他可以消除的地方在糖果落下后可能就没有了，所以博主在想点击的顺序肯定会影响最终的stable的状态，可是题目怎么没有要求返回所剩糖果最少的状态？后来发现，其实这道题一次消除table中所有可消除的

糖果，然后才下落，形成新的table，这样消除后得到的结果就是统一的了，这样也大大的降低了难度。下面就来看如何找到要消除的糖果，可能有人会想像之前的岛屿的题目一样找连通区域，可是这道题的有限制条件，只有横向或纵向相同的糖果数达到三个才能消除，并不是所有的连通区域都能消除，所以找连通区域不是一个好办法。最好的办法其实是每个糖果单独检查其是否能被消除，然后把所有能被删除的糖果都标记出来统一删除，然后在下落糖果，然后再次查找，直到无法找出能够消除的糖果时达到稳定状态。好，那么我们用一个数组来保存可以被消除的糖果的位置坐标，判断某个位置上的糖果能否被消除的方法就是检查其横向和纵向的最大相同糖果的个数，只要有一个方向达到三个了，当前糖果就可以被消除。所以我们对当前糖果的上下左右四个方向进行查看，用四个变量x0, x1, y0, y1，其中x0表示上方相同的糖果的最大位置，x1表示下方相同糖果的最大位置，y0表示左边相同糖果的最大位置，y1表示右边相同糖果的最大位置，均初始化为当前糖果的位置，然后使用while循环向每个方向遍历，注意我们并不需要遍历到头，而是只要遍历三个糖果就行了，因为一旦查到了三个相同的，就说明当前的糖果已经可以消除了，没必要再往下查了。查的过程还要注意处理越界情况，好，我们得到了上下左右的最大的位置，分别让相同方向的做差，如果水平和竖直方向任意一个大于3了，就说明可以消除，将坐标加入数组del中。注意这里一定要大于3，是因为当发现不相等退出while循环时，坐标值已经改变了，所以已经多加了或者减了一个，所以差值要大于3。遍历完成后，如果数组del为空，说明已经stable了，直接break掉，否则将要消除的糖果位置都标记为0，然后进行下落处理。下落处理实际上是把数组中的0都移动到开头，那么就从数组的末尾开始遍历，用一个变量t先指向末尾，然后当遇到非0的数，就将其和t位置上的数置换，然后t自减1，这样t一路减下来都是非0的数，而0都被置换到数组开头了，参见代码如下：

```

1 class Solution {
2 public:
3     vector<vector<int>> candyCrush(vector<vector<int>>& board) {
4         int m = board.size(), n = board[0].size();
5         while (true) {
6             vector<pair<int, int>> del;
7             for (int i = 0; i < m; ++i) {
8                 for (int j = 0; j < n; ++j) {
9                     if (board[i][j] == 0) continue;
10                    int x0 = i, x1 = i, y0 = j, y1 = j;
11                    while (x0 >= 0 && x0 > i - 3 && board[x0][j] == board[i][j]) --x0;
12                    while (x1 < m && x1 < i + 3 && board[x1][j] == board[i][j]) ++x1;
13                    while (y0 >= 0 && y0 > j - 3 && board[i][y0] == board[i][j]) --y0;
14                    while (y1 < n && y1 < j + 3 && board[i][y1] == board[i][j]) ++y1;
15                    if (x1 - x0 > 3 || y1 - y0 > 3) del.push_back({i, j});
16                }
17            }
18            if (del.empty()) break;
19            for (auto a : del) board[a.first][a.second] = 0;
20            for (int j = 0; j < n; ++j) {
21                int t = m - 1;
22                for (int i = m - 1; i >= 0; --i) {
23                    if (board[i][j]) swap(board[t--][j], board[i][j]);
24                }
25            }
26        }
27        return board;
28    }
29}

```

711. 寻找中枢点

Given an array of integers nums, write a method that returns the "pivot" index of this array.

We define the pivot index as the index where the sum of the numbers to the left of the index is equal to the sum of the numbers to the right of the index.

If no such index exists, we should return -1. If there are multiple pivot indexes, you should return the left-most pivot index.

这道题给了我们一个数组，让我们求一个中枢点，使得该位置左右两边的子数组之和相等。这道题难度不大，直接按题意去搜索就行了，因为中枢点可能出现的位置就是数组上的位置，所以我们搜索一遍就可以找出来，我们先求出数组的总和，然后维护一个当前数组之和curSum，然后对于遍历到的位置，用总和减去当前数字，看得到的结果是否是curSum的两倍，是的话，那么当前位置就是中枢点，返回即可；否则就将当前数字加到curSum中继续遍历，遍历结束后还没返回，说明没有中枢点，返回-1即可，参见代码如下：

```

1 class Solution {
2 public:
3     int pivotIndex(vector<int>& nums) {
4         int sum = accumulate(nums.begin(), nums.end(), 0);
5         int curSum = 0, n = nums.size();
6         for (int i = 0; i < n; ++i) {
7             if (sum - nums[i] == 2 * curSum) return i;
8             curSum += nums[i];
9         }
10    return -1;
11 }
12 };

```

CPP

712. 拆分链表成部分

Given a (singly) linked list with head node root, write a function to split the linked list into k consecutive linked list "parts".

The length of each part should be as equal as possible: no two parts should have a size differing by more than 1. This may lead to some parts being null.

The parts should be in order of occurrence in the input list, and parts occurring earlier should always have a size greater than or equal parts occurring later.

Return a List of ListNode's representing the linked list parts that are formed.

Examples 1->2->3->4, k = 5 // 5 equal parts [[1], [2], [3], [4], null]

这道题给我们一个链表和一个正数k，让我们分割链表成k部分，尽可能的平均分割，如果结点不够了，就用空结点，比如例子1中的。如果无法平均分，那么多余的结点就按顺序放在子链表中，如例子2中所示。我们要知道每个部分结点的个数，才能将整个链表断开成子链表，所以我们首先要统计链表中结点的总个数，然后除以k，得到的商avg就是能分成的部分个数，余数ext就是包含有多余的结点的子链表的个数。我们开始for循环，循环的结束条件是i小于k且root存在，要生成k个子链表，在循环中，先把头结点加入结果res中对应的位置，然后就要遍历该子链表的结点个数了，首先每个子链表都一定包含有avg个结点，这是之前除法得到的商，然后还要有没有多余结点，如果i小于ext，就说明当前子链表还得有一个多余结点，然后我们将指针向后移动一个，注意我们这里的j是从1开始，我们希望移动到子链表的最后一个结点上，而不是移动到下一个子链表的首结点，因为我们要断开链表。我们新建一个临时结点t指向下一个结点，也就是下一个子链表的首结点，然后将链表断开，再将root指向临时结点t，这样就完成了断开链表的操作，参见代码如下：

```

1 class Solution {
2 public:
3     vector<ListNode*> splitListToParts(ListNode* root, int k) {
4         vector<ListNode*> res(k);
5         int len = 0;
6         for (ListNode *t = root; t; t = t->next) ++len;
7         int avg = len / k, ext = len % k;
8         for (int i = 0; i < k && root; ++i) {
9             res[i] = root;
10            for (int j = 1; j < avg + (i < ext); ++j) {
11                root = root->next;
12            }
13            ListNode *t = root->next;
14            root->next = NULL;
15            root = t;
16        }
17        return res;
18    }
19 };

```

713. 原子的个数

Given a chemical formula (given as a string), return the count of each atom.

An atomic element always starts with an uppercase character, then zero or more lowercase letters, representing the name.

1 or more digits representing the count of that element may follow if the count is greater than 1. If the count is 1, no digits will follow. For example, H2O and H2O2 are possible, but H1O2 is impossible.

Two formulas concatenated together produce another formula. For example, H2O2He3Mg4 is also a formula.

A formula placed in parentheses, and a count (optionally added) is also a formula. For example, (H2O2) and (H2O2)3 are formulas.

Given a formula, output the count of all elements as a string in the following form: the first name (in sorted order), followed by its count (if that count is more than 1), followed by the second name (in sorted order), followed by its count (if that count is more than 1), and so on.

这道题给了我们一个化学式，让我们数其中原子的个数。比如水是H₂O，里面有2个氢原子，一个氧原子，返回还是H₂O。例子2给的是氢氧化镁（哈哈，想不到这么多年过去了，高中化学还没有完全还给老师，呀，暴露年龄了呢|||---），里面有一个镁原子，氧原子和氢原子各两个，我们返回H₂MgO₂，可以看到元素是按字母顺序排列的，这道题就是纯粹玩字符串，不需要任何的化学知识。再看第三个例子K₄(ON(SO₃)₂)₂，就算你不认识里面的钾，硫，氮，氧等元素，也不影响做题，这个例子的返回是K₄N₂O₁₄S₄，钾原子有4个，氮原子有2个，氧原子有14个，是 $3 \times 2 \times 2 + 2 = 14$ 得来的，硫原子有4个，是 $2 \times 2 = 4$ 得来的。那么我们可以发现规律，先统计括号里的原子个数，然后如果括号外面有数字，那么括号里每个原子的个数乘以外面的数字即可，然后在外层若还有数字，那么就继续乘这个数字，这种带有嵌套形式的字符串，比较适合用递归来做。我们最终的目的是统计每个原子的数量，所以我们只要建立了每个元素和其出现次数的映射，就可以生成返回的字符串了，由于需要按元素的字母顺序排列，所以我们使用TreeMap来建立映射。我们使用一个变量pos，来记录我们遍历的位置，这是个全局的变量，在递归函数参数中需要设置引用。我们遍历的时候，需要分三种情况讨论，分别是遇到左括号，右括号，和其他。我们一个个来看：

如果当前是左括号，那么我们pos先自增1，跳过括号位置，然后我们可以调用递归函数，来处理这个括号中包括的所有内容，外加上后面的数字，比如Mg(OH)₂，在pos=2处遇到左括号，调用完递归函数后pos指向了最后一个字符的后一位，即pos=7。而在K₄(ON(SO₃)₂)₂中，如果是遇到中间的那个左括号pos=5时，调用完递归函数后pos指向了第二个右括号，即pos=11。递归函数返回了中间部分所有原子跟其个数之间的映射，我们直接将其都加入到当前的映射中即可。

如果当前是右括号，说明一个完整的括号已经遍历完了，我们需要取出其后面的数字，如果括号存在，那么后面一定会跟数字，否则不需要括号。所以我们先让pos自增1，跳过括号的位置，然后用个变量i记录当前位置，再进行while循环，找出第一个非数字的位置，那么中间就都是数字啦，用substr将其提取出来，并转为整数，然后遍历当前的映射对，每个值都乘以这个倍数即可，然后返回。

如果当前是字母，那么需要将元素名提取出来了，题目中说了元素名只有第一个字母是大写，后面如果说有的话，都是小写字母。所以我们用个while循环找到第一个非小写字母的位置，用substr取出中间的字符串，即元素名。由于元素名后也可能跟数字，所以在用个while循环，来找之后第一个非数字的位置，用substr提取出数字字符串。当然也可能元素名后没有数字，提取出来的数字字符串就是空的，我们加的时候判断一下，如果为空就只加1，否则就加上转化后的整数，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     string countOfAtoms(string formula) {
4         string res = "";
5         int pos = 0;
6         map<string, int> m = parse(formula, pos);
7         for (auto a : m) {
8             res += a.first + (a.second == 1 ? "" : to_string(a.second));
9         }
10        return res;
11    }
12    map<string, int> parse(string& str, int& pos) {
13        map<string, int> res;
14        while (pos < str.size()) {
15            if (str[pos] == '(') {
16                ++pos;
17                for (auto a : parse(str, pos)) res[a.first] += a.second;
18            } else if (str[pos] == ')') {
19                int i = ++pos;
20                while (pos < str.size() && isdigit(str[pos])) ++pos;
21                int multiple = stoi(str.substr(i, pos - i));
22                for (auto a : res) a.second *= multiple;
23                return res;
24            } else {
25                int i = pos++;
26                while (pos < str.size() && islower(str[pos])) ++pos;
27                string elem = str.substr(i, pos - i);
28                i = pos;
29                while (pos < str.size() && isdigit(str[pos])) ++pos;
30                string cnt = str.substr(i, pos - i);
31                res[elem] += cnt.empty() ? 1 : stoi(cnt);
32            }
33        }
34        return res;
35    }
36};

```

下面这种解法是迭代形式，根据上面的递归解法改写而来。使用栈来代替递归函数，本身之上基本没有任何区别。需要注意的是，在遇到左括号时，我们将当前映射集cur加入了栈，这里用了个自带的move函数，表示将cur中所有的映射对移出并加入栈，之后cur就为空了。还有就是在处理右括号时，算出了倍数后，我们把当前的映射值乘以倍数后加到栈顶映射集中，然后用栈顶映射集来更新cur，并移除栈顶元素，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     string countOfAtoms(string formula) {
4         string res = "";
5         stack<map<string, int>> st;
6         map<string, int> cur;
7         int n = formula.size(), pos = 0;
8         while (pos < n) {
9             if (formula[pos] == '(') {
10                 ++pos;
11                 st.push(move(cur));
12             } else if (formula[pos] == ')') {
13                 int i = ++pos;
14                 while (pos < n && isdigit(formula[pos])) ++pos;
15                 int multiple = stoi(formula.substr(i, pos - i));
16                 for (auto a : cur) st.top()[a.first] += a.second * multiple;
17                 cur = move(st.top());
18                 st.pop();
19             } else {
20                 int i = pos++;
21                 while (pos < n && islower(formula[pos])) ++pos;
22                 string elem = formula.substr(i, pos - i);
23                 i = pos;
24                 while (pos < n && isdigit(formula[pos])) ++pos;
25                 string cnt = formula.substr(i, pos - i);
26                 cur[elem] += cnt.empty() ? 1 : stoi(cnt);
27             }
28         }
29         for (auto a : cur) {
30             res += a.first + (a.second == 1 ? "" : to_string(a.second));
31         }
32         return res;
33     }
34 };

```

714. 最小窗口序列

Given strings S and T, find the minimum (contiguous) substring W of S, so that T is a subsequence of W.

If there is no such window in S that covers all characters in T, return the empty string "". If there are multiple such minimum-length windows, return the one with the left-most starting index.

这道题给了我们两个字符串S和T，让我们找出S的一个长度最短子串W，使得T是W的子序列，如果长度相同，取起始位置靠前的。清楚子串和子序列的区别，那么题意就不难理解，题目中给的例子也很好的解释了题意。我们经过研究可以发现，返回的子串的起始字母和T的起始字母一定相同，这样才能保证最短。那么你肯定会想先试试暴力搜索吧，以S中每个T的起始字母为起点，均开始搜索字符串T，然后维护一个子串长度的最小值。如果是这种思路，那么还是趁早打消念头吧，博主已经替你试过了，OJ不依。原因也不难想，假如S中有大量的连续b，并且如果T也很长的话，这种算法实在是不高效啊。根据博主多年经验，这种玩字符串且还是Hard的题，十有八九都是要用动态规划Dynamic Programming来做的，那么就直接往DP上去想吧。DP的第一步就是设计dp数组，像这种两个字符串的题，一般都是一个二维数组，想想该怎么定义。确定一个子串的两个关键要素是起始位置和长度，那么我们的dp值到底应该是定起始位置还是长度呢？That is a question！仔细想一想，其实起始位置是长度的基础，因为我们一旦知道了起始位置，那么当前位置减去起始位置，就是长度了，所以我们dp值定为起始位置。那么 $dp[i][j]$ 表示范围S中前i个字符包含范围T中前j个字符的子串的起始位置，注意这里的包含是子序列包含关系。然后就是确定长度了，有时候会使用字符串的原长度，有时候会多加1，看个人习惯吧，这里博主长度多加了个1。

OK，下面就是重中之重啦，求递推式。一般来说， $dp[i][j]$ 的值是依赖于之前已经求出的dp值的，在递归形式的解法中，dp数组也可以看作是记忆数组，从而省去了大量的重复计算，这也是dp解法凌驾于暴力搜索之上的主要原因。牛B的方法总是最难想出来的，dp的递推式就是其中之一。在脑子一片浆糊的情况下，博主的建议是从最简单的例子开始分析，比如 $S = "b"$, $T = "b"$ ，那么我们就有 $dp[1][1] = 0$ ，因为S中的起始位置为0，长度为1的子串可以包含T。如果当 $S = "d"$, $T = "b"$ ，那么我们有 $dp[1][1] = -1$ ，因为我们的dp数组初始化均为-1，表示未匹配或者无法匹配。下面来看一个稍稍复杂些的例子， $S = "dbd"$, $T = "bd"$ ，我们的dp数组是：

\emptyset	b	d
\emptyset	?	?
d	?	-1
b	?	1
d	?	1

这里的问号是边界，我们还不知道如何初给边界赋值，我们看到，为-1的地方是对应的字母不相等的地方。我们首先要明确的是 $dp[i][j]$ 中的j不能大于i，因为T的长度不能大于S的长度，所以j大于i的 $dp[i][j]$ 一定都是-1的。再来看为1的几个位置，首先是 $dp[2][1] = 1$ ，这里表示db包含b的子串起始位置为1，make sense！然后是 $dp[3][1] = 1$ ，这里表示dbd包含b的子串起始位置为1，没错！然后是 $dp[3][2] = 1$ ，这里表示dbd包含bd的起始位置为1，all right！那么我们可以观察出，当 $S[i] == T[j]$ 的时候，实际上起始位置和 $dp[i - 1][j - 1]$ 是一样的，比如dbd包含bd的起始位置和db包含b的起始位置一样，所以可以继承过来。那么当 $S[i] != T[j]$ 的时候，怎么搞？其实是和 $dp[i - 1][j]$ 是一样的，比如dbd包含b的起始位置和db包含b的起始位置是一样的。

嗯，这就是递推式的核心了，下面再来看边界怎么赋值，由于j比如小于等于i，所以第一行的第二个位置往后一定都是-1，我们只需要给第一列赋值即可。通过前面的分析，我们知道了当 $S[i] == T[j]$ 时，我们取的是左上角的dp值，表示当前字母在S中的位置，由于我们dp数组提前加过1，所以第一列的数只要赋值为当前行数即可。最终的dp数组如下：

\emptyset	b	d
\emptyset	0	-1
d	1	-1
b	2	1
d	3	1

为了使代码更加简洁，我们在遍历完每一行，检测如果 $dp[i][n]$ 不为-1，说明T已经被完全包含了，且当前的位置跟起始位置都知道了，我们计算出长度来更新一个全局最小值minLen，同时更新最小值对应的起始位置start，最后取出这个全局最短子串，如果没有找到返回空串即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     string minWindow(string S, string T) {
4         int m = S.size(), n = T.size(), start = -1, minLen = INT_MAX;
5         vector<vector<int>> dp(m + 1, vector<int>(n + 1, -1));
6         for (int i = 0; i <= m; ++i) dp[i][0] = i;
7         for (int i = 1; i <= m; ++i) {
8             for (int j = 1; j <= min(i, n); ++j) {
9                 dp[i][j] = (S[i - 1] == T[j - 1]) ? dp[i - 1][j - 1] : dp[i - 1][j];
10            }
11            if (dp[i][n] != -1) {
12                int len = i - dp[i][n];
13                if (minLen > len) {
14                    minLen = len;
15                    start = dp[i][n];
16                }
17            }
18        }
19        return (start != -1) ? S.substr(start, minLen) : "";
20    }
21 };

```

论坛上的danzhutest大神提出了一种双指针的解法，其实这是优化过的暴力搜索的方法，而且居然beat了100%，给跪了好嘛？！而且这双指针的跳跃方式犹如舞蹈般美妙绝伦，比那粗鄙的暴力搜索双指针不知道高到哪里去了？！举个栗子来说吧，比如当 S = "bbbbddde", T = "bde"时，我们知道暴力搜索的双指针在S和T的第一个b匹配上之后，就开始检测S之后的字符能否包含T之后的所有字符，当匹配结束后，S的指针就会跳到第二个b开始匹配，由于有大量的重复b出现，所以每一个b都要遍历一遍，会达到平方级的复杂度，会被OJ无情拒绝。而下面这种修改后的算法会跳过所有重复的b，使得效率大大提升，具体是这么做的，当第一次匹配成功后，我们的双指针往前走，找到那个刚好包含T中字符的位置，比如开始指针 i = 0 时，指向S中的第一个b，指针 j = 0 时指向T中的第一个b，然后开始匹配T，当 i = 6, j = 2 时，此时完全包含了T。暴力搜索解法中此时i会回到1继续找，而这里，我们通过向前再次匹配T，会在 i = 3, j = 0 处停下，然后继续向后找，这样S中重复的b就会被跳过，从而达到线性的复杂度。旋转，跳跃，我闭着眼，尘嚣看不见，你沉醉了没？博主已经沉醉在这双指针之舞中了……。

解法2：

```

1 class Solution {
2 public:
3     string minWindow(string S, string T) {
4         int m = S.size(), n = T.size(), start = -1, minLen = INT_MAX, i = 0, j = 0;
5         while (i < m) {
6             if (S[i] == T[j]) {
7                 if (++j == n) {
8                     int end = i + 1;
9                     while (--j >= 0) {
10                         while (S[i--] != T[j]);
11                     }
12                     ++i; ++j;
13                     if (end - i < minLen) {
14                         minLen = end - i;
15                         start = i;
16                     }
17                 }
18             }
19             ++i;
20         }
21         return (start != -1) ? S.substr(start, minLen) : "";
22     }
23 };

```

715. 自整除数字

A self-dividing number is a number that is divisible by every digit it contains.

For example, 128 is a self-dividing number because $128 \% 1 == 0$, $128 \% 2 == 0$, and $128 \% 8 == 0$.

Also, a self-dividing number is not allowed to contain the digit zero.

Given a lower and upper number bound, output a list of every possible self dividing number, including the bounds if possible.

这道题让我们找一个给定范围内的所有的自整除数字，所谓的自整除数字就是该数字可以整除其每一个位上的数字。既然这道题是Easy类，那么一般来说不需要用tricky的方法，直接暴力搜索就行了，遍历区间内的所有数字，然后调用子函数判断其是否是自整除数，是的话就加入结果res中。在子函数中，我们先把数字转为字符串，然后遍历每个字符，只要其为0，或者num无法整除该位上的数字，就返回false，循环结束后返回true，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<int> selfDividingNumbers(int left, int right) {
4         vector<int> res;
5         for (int i = left; i <= right; ++i) {
6             if (check(i)) res.push_back(i);
7         }
8         return res;
9     }
10    bool check(int num) {
11        string str = to_string(num);
12        for (char c : str) {
13            if (c == '0' || num % (c - '0')) return false;
14        }
15        return true;
16    }
17 };

```

我们可以不用子函数，直接在大的for循环中加上一个for循环进行判断即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<int> selfDividingNumbers(int left, int right) {
4         vector<int> res;
5         for (int i = left, n = 0; i <= right; ++i) {
6             for (n = i; n > 0; n /= 10) {
7                 if (n % 10 == 0 || i % (n % 10) != 0) break;
8             }
9             if (n == 0) res.push_back(i);
10        }
11        return res;
12    }
13 };

```

716. 我的日历之一

Implement a MyCalendar class to store your events. A new event can be added if adding the event will not cause a double booking.

Your class will have the method, book(int start, int end). Formally, this represents a booking on the half open interval [start, end), the range of real numbers x such that start <= x < end.

A double booking happens when two events have some non-empty intersection (ie., there is some time that is common to both events.)

For each call to the method MyCalendar.book, return true if the event can be added to the calendar successfully without causing a double booking. Otherwise, return false and do not add the event to the calendar.

Your class will be called like this: MyCalendar cal = new MyCalendar(); MyCalendar.book(start, end)

这道题让我们设计一个我的日历类，里面有一个book函数，需要给定一个起始时间和结束时间，与Google Calendar不同的是，我们的事件事件上不能重叠，实际上这道题的本质就是检查区间是否重叠。那么我们可以暴力搜索，对于每一个将要加入的区间，我们都和已经已经存在的区间进行比较，看是否有重复。而新加入的区间和当前区间产生重复的情况有两种，一种是新加入区间的前半段重复，并且，另一种是新加入区间的后半段重复。比如当前区间如果是[3, 8)，那么第一种情况下新加入区间就是[6, 9)，那么触发条件就是当前区间的起始时间小于等于新加入区间的起始时间，并且结束时间大于新加入区间的结束时间。第二种情况下新加入区间就是[2,5)，那么触发条件就是当前区间的起始时间大于等于新加入区间的起始时间，并且起始时间小于新加入区间的结束时间。这两种情况均返回false，否则就将新区间加入数组，并返回true即可，参见代码如下：

解法1：

```
1 class MyCalendar {
2 public:
3     MyCalendar() {}
4
5     bool book(int start, int end) {
6         for (auto a : cal) {
7             if (a.first <= start && a.second > start) return false;
8             if (a.first >= start && a.first < end) return false;
9         }
10        cal.push_back({start, end});
11        return true;
12    }
13
14 private:
15     vector<pair<int, int>> cal;
16 };
CPP
```

下面这种方法将上面方法的两个if判断融合成了一个，我们来观察两个区间的起始和结束位置的关系发现，如果两个区间的起始时间中的较大值小于结束区间的较小值，那么就有重合，返回false。比如 [3, 8) 和 [6, 9)，3和6中的较大值6，小于8和9中的较小值8，有重叠。再比如[3, 8) 和 [2, 5)，3和2中的较大值3，就小于8和5中的较小值5，有重叠。而对于[3, 8) 和 [9, 10)，3和9中的较大值9，不小于8和10中的较小值8，所以没有重叠，参见代码如下：

解法2：

```
1 class MyCalendar {
2 public:
3     MyCalendar() {}
4
5     bool book(int start, int end) {
6         for (auto a : cal) {
7             if (max(a.first, start) < min(a.second, end)) return false;
8         }
9         cal.push_back({start, end});
10        return true;
11    }
12
13 private:
14     vector<pair<int, int>> cal;
15 };
CPP
```

上面两种解法都是线性搜索，我们起始可以优化搜索时间，如果我们的区间是有序的话。所以我们用一个map来建立起始时间和结束时间的映射，map会按照起始时间进行自动排序。然后对于新进来的区间，我们在已有区间中查找第一个不小于新入区间的起始时间的区间，如果这个区间存在的话，说明新入区间的起始时间小于等于当前区间，也就是解法一中的第二个if情况，当前

区间起始时间小于新入区间结束时间的话返回false。我们还要跟前面一个区间进行查重叠操作，那么判断如果当前区间不是第一个区间的话，就找到前一个区间，此时是解法一中第一个if情况，并且如果前一个区间的结束时间大于新入区间的起始时间的话，返回false。否则就建立新的映射，返回true即可，参见代码如下：

解法3：

```
1 class MyCalendar {
2 public:
3     MyCalendar() {}
4
5     bool book(int start, int end) {
6         auto it = cal.lower_bound(start);
7         if (it != cal.end() && it->first < end) return false;
8         if (it != cal.begin() && prev(it)->second > start) return false;
9         cal[start] = end;
10        return true;
11    }
12
13 private:
14     map<int, int> cal;
15 }
```

717. 计数不同的回文子序列的个数

Given a string S, find the number of different non-empty palindromic subsequences in S, and return that number modulo $10^9 + 7$.

A subsequence of a string S is obtained by deleting 0 or more characters from S.

A sequence is palindromic if it is equal to the sequence reversed.

Two sequences A_1, A_2, ... and B_1, B_2, ... are different if there is some i for which A_i != B_i.

这道题给了给了我们一个字符串，让我们求出所有的非空回文子序列的个数，虽然这题限制了字符只有四种，但是我们还是按一般的情况来解吧，可以有26个字母。然后说最终结果要对一个很大的数字取余，这就暗示了结果会是一个很大的值，那么对于这种问题一般都是用DP或者是带记忆数组memo的递归来解，二者的本质其实是一样的。我们先来看带记忆数组memo的递归解法，这种解法的思路是一层一层剥洋葱，比如"bccb"，按照字母来剥，先剥字母b，确定最外层"b _ _ b"，这会产生两个回文子序列"b"和"bb"，然后递归进中间的部分，把中间的回文子序列个数算出来加到结果res中，然后开始剥字母c，找到最外层"cc"，此时会产生两个回文子序列"c"和"cc"，然后由于中间没有字符串了，所以递归返回0，按照这种方法就可以算出所有的回文子序列了。

我们建立一个二维数组chars，外层长度为26，里面放一个空数组。这是为了统计每个字母在原字符串中出现的位置，然后定义一个二维记忆数组memo，其中memo[i][j]表示第i个字符到第j个字符之间的子字符串中的回文子序列的个数，初始化均为0。然后我们遍历字符串S，将每个字符的位置加入其对应的数组中，比如对于"bccb"，那么有：

b -> {0, 3}

c -> {1, 2}

然后在[0, n]的范围内调用递归函数，在递归函数中，首先判断如果start大于等于end，返回0。如果当前位置在memo的值大于0，说明当前情况已经计算过了，直接返回memo数组中的值。否则进行所有字母的遍历，如果某个字母对应的数组中没有值，说明该字母不曾在字符串中出现，跳过。然后我们在字母数组中查找第一个不小于start的位置，查找第一个小于end的位置，当前循环中，start为0，end为4，当前处理字母b，我们的new_start指向0，new_end指向3，如果当前new_start指向了end()，或者其指向的位置大于end，说明当前范围内没有字母b，直接跳过，否则结果res自增1，因为此时new_start存在，至少有个单个的字母b，也可以当作回文子序列，然后看new_start和new_end如果不相同，说明两者各指向了不同的b，此时res应自增1，因为又增加了一个新的回文子序列"bb"，下面就是对中间部分调用递归函数了，把返回值加到结果res中。此时字母b就处理完了，现在处理字母c，此时的start还是0，end还是4，new_start指向1，new_end指向2，跟上面的分析相同，new_start在范围内，结果自增1，因为加上了"c"，然后new_start和new_end不同，结果res再自增1，因为加上了"cc"，中间没有字符了，调用递归的结果是0，for循环结束，我们将memo[start][end]的值对超大数取余，将该值返回即可，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     int countPalindromicSubsequences(string S) {
4         int n = S.size();
5         vector<vector<int>> chars(26, vector<int>());
6         vector<vector<int>> memo(n + 1, vector<int>(n + 1, 0));
7         for (int i = 0; i < n; ++i) {
8             chars[S[i] - 'a'].push_back(i);
9         }
10        return helper(S, chars, 0, n, memo);
11    }
12    int helper(string S, vector<vector<int>>& chars, int start, int end,
13    vector<vector<int>>& memo) {
14        if (start >= end) return 0;
15        if (memo[start][end] > 0) return memo[start][end];
16        long res = 0;
17        for (int i = 0; i < 26; ++i) {
18            if (chars[i].empty()) continue;
19            auto new_start = lower_bound(chars[i].begin(), chars[i].end(), start);
20            auto new_end = lower_bound(chars[i].begin(), chars[i].end(), end) - 1;
21            if (new_start == chars[i].end() || *new_start >= end) continue;
22            ++res;
23            if (new_start != new_end) ++res;
24            res += helper(S, chars, *new_start + 1, *new_end, memo);
25        }
26        memo[start][end] = res % int(1e9 + 7);
27        return memo[start][end];
28    }
};
```

我们再来看一种迭代的写法，使用一个二维的dp数组，其中 $dp[i][j]$ 表示子字符串 $[i, j]$ 中的不同回文子序列的个数，我们初始化 $dp[i][i]$ 为1，因为任意一个单个字符就是一个回文子序列，其余均为0。这里的更新顺序不是正向，也不是逆向，而是斜着更新，对于“bccb”的例子，其最终dp数组如下，我们可以看到其更新顺序分别是红-绿-蓝-橙。

```

b c c b
b 1 2 3 6
c 0 1 2 3
c 0 0 1 2
b 0 0 0 1

```

这样更新的好处是，更新当前位置时，其左，下，和左下位置的dp值均已存在，而当前位置的dp值需要用到这三个位置的dp值。我们观察上面的dp数组，可以发现当 $S[i] \neq S[j]$ 的时候， $dp[i][j] = dp[i][j - 1] + dp[i + 1][j] - dp[i + 1][j - 1]$ ，即当前的dp值等于左边值加下边值减去左下值，因为算左边值的时候包括了左下的所有情况，而算下边值的时候也包括了左下值的所有情况，那么左下值就多算了一遍，所以要减去。而当 $S[i] = S[j]$ 的时候，情况就比较复杂了，需要分情况讨论，因为我们不知道中间还有几个和 $S[i]$ 相等的值。举个简单的例子，比如“aba”和“aaa”，当 $i = 0, j = 2$ 的时候，两个字符串均有 $S[i] == S[j]$ ，此时二者都新增两个子序列“a”和“aa”，但是“aba”中间的“b”就可以加到结果res中，而“aaa”中的“a”就不能加了，因为和外层的单独“a”重复了。我们的目标就要找到中间重复的“a”。所以我们让 $left = i + 1, right = j - 1$ ，然后对left进行while循环，如果 $left <= right$ ，且 $S[left] != S[i]$ 的时候，left向右移动一个；同理，对right进行while循环，如果 $left <= right$ ，且 $S[right] != S[i]$ 的时候，right向左移动一个。这样最终left和right值就有三种情况：

1. 当 $left > right$ 时，说明中间没有和 $S[i]$ 相同的字母了，就是“aba”这种情况，那么就有 $dp[i][j] = dp[i + 1][j - 1] * 2 + 2$ ，其中 $dp[i + 1][j - 1]$ 是中间部分的回文子序列个数，为啥要乘2呢，因为中间的所有子序列可以单独存在，也可以再外面裹上字母a，所以是成对出现的，要乘2。加2的原因是外层的“a”和“aa”也要统计上。
2. 当 $left = right$ 时，说明中间只有一个和 $S[i]$ 相同的字母，就是“aaa”这种情况，那么有 $dp[i][j] = dp[i + 1][j - 1] * 2 + 1$ ，其中乘2的部分跟上面的原因相同，加1的原因是单个字母“a”的情况已经在中间部分算过了，外层就只能再加上一个“aa”了。
3. 当 $left < right$ 时，说明中间至少有两个和 $S[i]$ 相同的字母，就是“aabaa”这种情况，那么有 $dp[i][j] = dp[i + 1][j - 1] * 2 - dp[left + 1][right - 1]$ ，其中乘2的部分跟上面的原因相同，要减去left和right中间部分的子序列个数的原因是其被计算了两遍，要将多余的减掉。

参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int countPalindromicSubsequences(string S) {
4         int n = S.size(), M = 1e9 + 7;
5         vector<vector<int>> dp(n, vector<int>(n, 0));
6         for (int i = 0; i < n; ++i) dp[i][i] = 1;
7         for (int len = 1; len < n; ++len) {
8             for (int i = 0; i < n - len; ++i) {
9                 int j = i + len;
10                if (S[i] == S[j]) {
11                    int left = i + 1, right = j - 1;
12                    while (left <= right && S[left] != S[i]) ++left;
13                    while (left <= right && S[right] != S[i]) --right;
14                    if (left > right) {
15                        dp[i][j] = dp[i + 1][j - 1] * 2 + 2;
16                    } else if (left == right) {
17                        dp[i][j] = dp[i + 1][j - 1] * 2 + 1;
18                    } else {
19                        dp[i][j] = dp[i + 1][j - 1] * 2 - dp[left + 1][right - 1];
20                    }
21                } else {
22                    dp[i][j] = dp[i][j - 1] + dp[i + 1][j] - dp[i + 1][j - 1];
23                }
24                dp[i][j] = (dp[i][j] < 0) ? dp[i][j] + M : dp[i][j] % M;
25            }
26        }
27        return dp[0][n - 1];
28    }
29};

```

718. 我的日历之二

Implement a MyCalendarTwo class to store your events. A new event can be added if adding the event will not cause a triple booking.

Your class will have one method, book(int start, int end). Formally, this represents a booking on the half open interval [start, end), the range of real numbers x such that start \leq x $<$ end.

A triple booking happens when three events have some non-empty intersection (ie., there is some time that is common to all 3 events.)

For each call to the method MyCalendar.book, return true if the event can be added to the calendar successfully without causing a triple booking. Otherwise, return false and do not add the event to the calendar.

Your class will be called like this: MyCalendar cal = new MyCalendar(); MyCalendar.book(start, end)

这道题是My Calendar I的拓展，之前那道题说是不能有任何的重叠区间，而这道题说最多容忍两个重叠区域，注意是重叠区域，不是事件。比如事件A, B, C互不重叠，但是有一个事件D，和这三个事件都重叠，这样是可以的，因为重叠的区域最多只有两个。所以关键还是要知道具体的重叠区域，如果两个事件重叠，那么重叠区域就是它们的交集，求交集的方法是两个区间的起始时间中的较大值，到结束时间中的较小值。那么我们可以用一个集合来专门存重叠区间，再用一个集合来存完整的区间，那么我们的思路就是，先遍历专门存重叠区间的集合，因为能在这里出现的区间，都已经是出现两次了，如果当前新的区间跟重叠区间有交集的话，说明此时三个事件重叠了，直接返回false。如果当前区间跟重叠区间没有交集的话，那么再来遍历完整区间的集合，如果有交集的话，那么应该算出重叠区间并且加入放重叠区间的集合中。最后记得将新区间加入完整区间的集合中，参见代码如下：

解法1：

```
1 class MyCalendarTwo {
2 public:
3     MyCalendarTwo() {}
4
5     bool book(int start, int end) {
6         for (auto a : s2) {
7             if (start >= a.second || end <= a.first) continue;
8             else return false;
9         }
10        for (auto a : s1) {
11            if (start >= a.second || end <= a.first) continue;
12            else s2.insert({max(start, a.first), min(end, a.second)});
13        }
14        s1.insert({start, end});
15        return true;
16    }
17
18 private:
19     set<pair<int, int>> s1, s2;
20 }
```

CPP

下面这种方法相当的巧妙，我们建立一个时间点和次数之间的映射，规定遇到起始时间点，次数加1，遇到结束时间点，次数减1。那么我们首先更改新的起始时间start和结束时间end的映射，start对应值增1，end对应值减1。然后定义一个变量cnt，来统计当前的次数。我们使用treemap具有自动排序的功能，所以我们遍历的时候就是按时间顺序的，最先遍历到的一定是一个起始时间，所以加上其映射值，一定是个正数。那么我们想，如果此时只有一个区间，就是刚加进来的区间的话，那么首先肯定遍历到start，那么cnt此时加1，然后就会遍历到end，那么此时cnt减1，最后下来cnt为0，没有重叠。还是用具体数字来说吧，我们现在假设treemap中已经加入了一个区间[3, 5)了，那么我们就有下面的映射：

3 -> 1

5 -> -1

假如我们此时要加入的区间为[6, 8)的话，那么在遍历到6的时候，前面经过3和5，分别加1减1，那么cnt又重置为0了，而后面的6和8也是分别加1减1，还是0。那么加入我们新加入的区间为[3, 8]时，那么此时的映射为：

3 -> 2

5 -> -1

8 -> -1

那么我们最先遍历到3，cnt为2，没有超过3，我们知道此时有两个事件有重叠，是允许的。然后遍历5和8，分别减去1，最终又变成0了，始终cnt没有超过2，所以是符合题意的。如果此时我们再加入一个新的区间[1, 4)，那么此时的映射为：

1 -> 1

3 -> 2

4 -> -1

5 -> -1

8 -> -1

那么我们先遍历到1，cnt为1，然后遍历到3，此时cnt为3了，那么我们就知道有三个事件有重叠区间了，所以这个新区间是不能加入的，那么我们要还原其start和end做的操作，把start的映射值减1，end的映射值加1，然后返回false。否则没有三个事件有共同重叠区间的话，返回true即可，参见代码如下：

解法2：

```

1 class MyCalendarTwo {
2 public:
3     MyCalendarTwo() {}
4
5     bool book(int start, int end) {
6         ++freq[start];
7         --freq[end];
8         int cnt = 0;
9         for (auto f : freq) {
10             cnt += f.second;
11             if (cnt == 3) {
12                 --freq[start];
13                 ++freq[end];
14                 return false;
15             }
16         }
17         return true;
18     }
19
20 private:
21     map<int, int> freq;
22 };
23

```

719. 我的日历之三

Implement a MyCalendarThree class to store your events. A new event can always be added.

Your class will have one method, book(int start, int end). Formally, this represents a booking on the half open interval [start, end), the range of real numbers x such that start \leq x < end.

A K-booking happens when K events have some non-empty intersection (ie., there is some time that is common to all K events.)

For each call to the method MyCalendar.book, return an integer K representing the largest integer such that there exists a K-booking in the calendar.

Your class will be called like this: MyCalendarThree cal = new MyCalendarThree();
MyCalendarThree.book(start, end)

这道题是之前那两道题My Calendar II, My Calendar I的拓展，论坛上有人说这题不应该算是Hard类的，但实际上如果没有之前那两道题做铺垫，直接上这道其实还是还蛮有难度的。这道题博主在做完之前那道，再做这道一下子就做出来了，因为用的就是之前那道My Calendar II的解法二，具体的讲解可以参见那道题，反正博主写完那道题再来做这道题就是秒解啊，参见代码如下：

```

1 class MyCalendarThree {
2 public:
3     MyCalendarThree() {}
4
5     int book(int start, int end) {
6         ++freq[start];
7         --freq[end];
8         int cnt = 0, mx = 0;
9         for (auto f : freq) {
10            cnt += f.second;
11            mx = max(mx, cnt);
12        }
13        return mx;
14    }
15
16 private:
17     map<int, int> freq;
18 };

```

720. 洪水填充

An image is represented by a 2-D array of integers, each integer representing the pixel value of the image (from 0 to 65535).

Given a coordinate (sr , sc) representing the starting pixel (row and column) of the flood fill, and a pixel value $newColor$, "flood fill" the image.

To perform a "flood fill", consider the starting pixel, plus any pixels connected 4-directionally to the starting pixel of the same color as the starting pixel, plus any pixels connected 4-directionally to those pixels (also with the same color as the starting pixel), and so on. Replace the color of all of the aforementioned pixels with the $newColor$.

At the end, return the modified image.

这道题给了我们一个用二维数组表示的图像，不同的数字代表不同的颜色，给了我们一个起始点坐标，还有一个新的颜色，让我们把起始点的颜色以及其相邻的同样的颜色都换成新的颜色。那么实际上就是一个找相同区间的题，我们可以用BFS或者DFS来做。先来看BFS的解法，我们使用一个队列queue来辅助，首先将给定点放入队列中，然后进行while循环，条件是queue不为空，然后进行类似层序遍历的方法，取出队首元素，将其赋值为新的颜色，然后遍历周围四个点，如果不越界，且周围的颜色跟起始颜色相同的话，将位置加入队列中，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<vector<int>> floodFill(vector<vector<int>>& image, int sr, int sc, int newColor)
4     {
5         int m = image.size(), n = image[0].size(), color = image[sr][sc];
6         vector<vector<int>> res = image;
7         vector<vector<int>> dirs{{0,-1},{-1,0},{0,1},{1,0}};
8         queue<pair<int, int>> q{{{sr, sc}}};
9         while (!q.empty()) {
10             int len = q.size();
11             for (int i = 0; i < len; ++i) {
12                 auto t = q.front(); q.pop();
13                 res[t.first][t.second] = newColor;
14                 for (auto dir : dirs) {
15                     int x = t.first + dir[0], y = t.second + dir[1];
16                     if (x < 0 || x >= m || y < 0 || y >= n || res[x][y] != color) continue;
17                     q.push({x, y});
18                 }
19             }
20         }
21         return res;
22     }
23 };

```

DFS的写法相对简洁一些，首先判断如果给定位置的颜色跟新的颜色相同的话，直接返回，否则就对给定位置调用递归函数。在递归函数中，如果越界或者当前颜色跟起始颜色不同，直接返回。否则就给当前位置赋上新的颜色，然后对周围四个点继续调用递归函数，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<vector<int>> floodFill(vector<vector<int>>& image, int sr, int sc, int newColor)
4     {
5         if (image[sr][sc] == newColor) return image;
6         helper(image, sr, sc, image[sr][sc], newColor);
7         return image;
8     }
9     void helper(vector<vector<int>>& image, int i, int j, int color, int newColor) {
10         int m = image.size(), n = image[0].size();
11         if (i < 0 || i >= m || j < 0 || j >= n || image[i][j] != color) return;
12         image[i][j] = newColor;
13         helper(image, i + 1, j, color, newColor);
14         helper(image, i, j + 1, color, newColor);
15         helper(image, i - 1, j, color, newColor);
16         helper(image, i, j - 1, color, newColor);
17     }
18 };

```

721. 句子相似度

Given two sentences words1, words2 (each represented as an array of strings), and a list of similar word pairs pairs, determine if two sentences are similar.

For example, "great acting skills" and "fine drama talent" are similar, if the similar word pairs are pairs = [["great", "fine"], ["acting","drama"], ["skills","talent"]].

Note that the similarity relation is not transitive. For example, if "great" and "fine" are similar, and "fine" and "good" are similar, "great" and "good" are not necessarily similar.

However, similarity is symmetric. For example, "great" and "fine" being similar is the same as "fine" and "great" being similar.

Also, a word is always similar with itself. For example, the sentences words1 = ["great"], words2 = ["great"], pairs = [] are similar, even though there are no specified similar word pairs.

Finally, sentences can only be similar if they have the same number of words. So a sentence like words1 = ["great"] can never be similar to words2 = ["doubleplus","good"].

Note:

The length of words1 and words2 will not exceed 1000.

The length of pairs will not exceed 2000.

The length of each pairs[i] will be 2.

The length of each words[i] and pairs[i][j] will be in the range [1, 20].

这道题给了我们两个句子，问这两个句子是否是相似的。判定的条件是两个句子的单词数要相同，而且每两个对应的单词要是相似度，这里会给出一些相似的单词对，这里说明了单词对的相似具有互逆性但是没有传递性。看到这里博主似乎已经看到了Follow up了，加上传递性就是一个很好的拓展。那么这里没有传递性，就使得问题变得很容易了，我们只要建立一个单词和其所有相似单词的集合的映射就可以了，比如说如果great和fine类似，且great和good类似，那么就有下面这个映射：

```
great -> {fine, good}
```

所以我们在逐个检验两个句子中对应的单词时就可以直接去映射中找，注意有可能遇到的单词对时反过来的，比如fine和great，所以我们两个单词都要带到映射中去查找，只要有一个能查找到，就说明是相似的，反之，如果两个都没查找到，说明不相似，直接返回false，参见代码如下：

```
1 class Solution {
2 public:
3     bool areSentencesSimilar(vector<string>& words1, vector<string>& words2,
4     vector<pair<string, string>> pairs) {
5         if (words1.size() != words2.size()) return false;
6         unordered_map<string, unordered_set<string>> m;
7         for (auto pair : pairs) {
8             m[pair.first].insert(pair.second);
9         }
10        for (int i = 0; i < words1.size(); ++i) {
11            if (words1[i] == words2[i]) continue;
12            if (!m[words1[i]].count(words2[i]) && !m[words2[i]].count(words1[i])) return
13 false;
14        }
15        return true;
16    }
};
```

CPP

We are given an array asteroids of integers representing asteroids in a row.

For each asteroid, the absolute value represents its size, and the sign represents its direction (positive meaning right, negative meaning left). Each asteroid moves at the same speed.

Find out the state of the asteroids after all collisions. If two asteroids meet, the smaller one will explode. If both are the same size, both will explode. Two asteroids moving in the same direction will never meet.

这道题用一个数组来模拟行星碰撞，正数代表行星向右移动，负数表示向左移动，绝对值大小表示行星的质量，如果两个相邻的行星相向移动会碰撞，质量大的行星会完好无损的保存，质量小的就会灰飞烟灭。那么博主最开始想的方法就是按照题目要求来一个一个的处理，我们先把给定的数组放到结果res中，然后进行while循环，如果此时结果res中的数字个数小于等于1个，直接返回即可，没有可碰撞的了。否则我们建立一个临时数组t，把结果res中的首元素放到t中，然后从第二个数字开始遍历结果res，如果此时t为空了，或者当前数字大于0而t数组最后一个数字小于0（此时两个行星向相反方向运动，不会相撞），或者两个数字的符号相同（此时两个行星向同一个方向运动，不会相撞），这三种情况下都把当前数字res[i]加到数组t中；那么剩下的情况就是两个行星相向运动了，如果两个数字相加等于0，则说明两个行星质量相同，且相向运动，则一起消失，我们将数组t中最后一个数字移除；如果当前数字小于0，且两个数字相加小于0，那么此时相撞后会留下质量大的行星，我们将数组t的最后一个数字赋值为res[i]即可。for循环之和，如果数组t和结果res的大小相等，说明此时状态已经稳定了，我们直接break，否则就把数组t赋值给结果res并继续循环，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     vector<int> asteroidCollision(vector<int>& asteroids) {
4         vector<int> res = asteroids;
5         while (true) {
6             if (res.size() <= 1) return res;
7             vector<int> t{res[0]};
8             for (int i = 1; i < res.size(); ++i) {
9                 if (t.empty() || (res[i] > 0 && t.back() < 0) || res[i] * t.back() > 0) {
10                     t.push_back(res[i]);
11                 } else if (res[i] + t.back() == 0) {
12                     t.pop_back();
13                 } else if (res[i] < 0 && res[i] + t.back() < 0) {
14                     t.back() = res[i];
15                 }
16             }
17             if (t.size() == res.size()) break;
18             else res = t;
19         }
20         return res;
21     }
22 };

```

CPP

实际上我们可以写的更加简洁一些，我们遍历所有的数字，如果当前数字是正数的话，我们直接加入结果res；否则我们遇到的都是负数，如果结果res为空，或者结果res的最后一个数字小于0（此时两个行星同时向左运动），直接将当前数字加入结果res；如果结果res的最后一个数字（此时为正数）小于当前数字的绝对值，说明碰撞后消失了，那么我们将i自减一个，然后将res最后一个数字移除，这样下次遍历的时候还是这个质量大的行星。如果两个质量相等，那么直接移除res最后一个数字，此时两个行星都消失了，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     vector<int> asteroidCollision(vector<int>& asteroids) {
4         vector<int> res;
5         for (int i = 0; i < asteroids.size(); ++i) {
6             if (asteroids[i] > 0) {
7                 res.push_back(asteroids[i]);
8             } else if (res.empty() || res.back() < 0) {
9                 res.push_back(asteroids[i]);
10            } else if (res.back() <= -asteroids[i]) {
11                if (res.back() < -asteroids[i]) --i;
12                res.pop_back();
13            }
14        }
15        return res;
16    }
17 };

```

723. 句子相似度之二

Given two sentences words1, words2 (each represented as an array of strings), and a list of similar word pairs pairs, determine if two sentences are similar.

For example, words1 = ["great", "acting", "skills"] and words2 = ["fine", "drama", "talent"] are similar, if the similar word pairs are pairs = [["great", "good"], ["fine", "good"], ["acting", "drama"], ["skills", "talent"]].

Note that the similarity relation is transitive. For example, if "great" and "good" are similar, and "fine" and "good" are similar, then "great" and "fine" are similar.

Similarity is also symmetric. For example, "great" and "fine" being similar is the same as "fine" and "great" being similar.

Also, a word is always similar with itself. For example, the sentences words1 = ["great"], words2 = ["great"], pairs = [] are similar, even though there are no specified similar word pairs.

Finally, sentences can only be similar if they have the same number of words. So a sentence like words1 = ["great"] can never be similar to words2 = ["doubleplus", "good"].

这道题是之前那道Sentence Similarity的拓展，那道题说单词之间不可传递，于是乎这道题就变成可以传递了，那么难度就增加了。不过没有关系，还是用我们的经典老三样来解，BFS，DFS，和Union Find。我们先来看BFS的解法，其实这道题的本质是无向连通图的问题，那么首先要做就是建立这个连通图的数据结构，对于每个结点来说，我们要记录所有和其相连的结点，所以我们建立每个结点和其所有相连结点集合之间的映射，比如对于这三个相似对(a, b), (b, c), 和(c, d)，我们有如下的映射关系：

```
a -> {b}
b -> {a, c}
c -> {b, d}
d -> {c}
```

那么如果我们要验证a和d是否相似，就需要用到传递关系，a只能找到b，b可以找到a, c，为了不陷入死循环，我们将访问过的结点加入一个集合visited，那么此时b只能去，c只能去d，那么说明a和d是相似的了。那么我们用for循环来比较对应位置上的两个单词，如果二者相同，那么直接跳过去比较接下来的。否则就建一个访问即可visited，建一个队列queue，然后把words1中的单词放入queue，建一个布尔型变量succ，标记是否找到，然后就是传统的BFS遍历的写法了，从队列中取元素，如果和其相连的结点中有words2中的对应单词，标记succ为true，并break掉。否则就将取出的结点加入队列queue，并且遍历其所有相连结点，将其中未访问过的结点加入队列queue继续循环，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     bool areSentencesSimilarTwo(vector<string>& words1, vector<string>& words2,
4     vector<pair<string, string>> pairs) {
5         if (words1.size() != words2.size()) return false;
6         unordered_map<string, unordered_set<string>> m;
7         for (auto pair : pairs) {
8             m[pair.first].insert(pair.second);
9             m[pair.second].insert(pair.first);
10        }
11        for (int i = 0; i < words1.size(); ++i) {
12            if (words1[i] == words2[i]) continue;
13            unordered_set<string> visited;
14            queue<string> q{{words1[i]}};
15            bool succ = false;
16            while (!q.empty()) {
17                auto t = q.front(); q.pop();
18                if (m[t].count(words2[i])) {
19                    succ = true; break;
20                }
21                visited.insert(t);
22                for (auto a : m[t]) {
23                    if (!visited.count(a)) q.push(a);
24                }
25            }
26            if (!succ) return false;
27        }
28        return true;
29    }
};
```

CPP

下面来看递归的写法，解题思路跟上面的完全一样，把主要操作都放到了一个递归函数中来写，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     bool areSentencesSimilarTwo(vector<string>& words1, vector<string>& words2,
4     vector<pair<string, string>> pairs) {
5         if (words1.size() != words2.size()) return false;
6         unordered_map<string, unordered_set<string>> m;
7         for (auto pair : pairs) {
8             m[pair.first].insert(pair.second);
9             m[pair.second].insert(pair.first);
10        }
11        for (int i = 0; i < words1.size(); ++i) {
12            unordered_set<string> visited;
13            if (!helper(m, words1[i], words2[i], visited)) return false;
14        }
15        return true;
16    }
17    bool helper(unordered_map<string, unordered_set<string>>& m, string& cur, string&
18 target, unordered_set<string>& visited) {
19        if (cur == target) return true;
20        visited.insert(cur);
21        for (string word : m[cur]) {
22            if (!visited.count(word) && helper(m, word, target, visited)) return true;
23        }
24        return false;
25    }
26};

```

下面这种解法就是碉堡了的联合查找Union Find了，这种解法的核心是一个getRoot函数，如果两个元素属于同一个群组的话，调用getRoot函数会返回相同的值。主要分为两部，第一步是建立群组关系，suppose开始时每一个元素都是独立的个体，各自属于不同的群组。然后对于每一个给定的关系对，我们对两个单词分别调用getRoot函数，找到二者的祖先结点，如果从未建立过联系的话，那么二者的祖先结点时不同的，此时就要建立二者的关系。等所有的关系都建立好了以后，第二步就是验证两个任意的元素是否属于同一个群组，就只需要比较二者的祖先结点都否相同啦。是不是有点深度学习的赶脚，先建立模型training，然后再test。哈哈，博主乱扯的，二者并没有什么联系。我们保存群组关系的数据结构，有时用数组，有时用哈希map，看输入的数据类型吧，如果输入元素的整型数的话，用root数组就可以了，如果是像本题这种的字符串的话，需要用哈希表来建立映射，建立每一个结点和其祖先结点的映射。注意这里的祖先结点不一定是最终祖先结点，而最终祖先结点的映射一定是最重祖先结点，所以我们的getRoot函数的设计思路就是要找到最终祖先结点，那么就是当结点和其映射结点相同时返回，否则继续循环，可以递归写，也可以迭代写，这无所谓。注意这里第一行判空是相当于初始化，这个操作可以在外面写，就是要让初始时每个元素属于不同的群组，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     bool areSentencesSimilarTwo(vector<string>& words1, vector<string>& words2,
4     vector<pair<string, string>> pairs) {
5         if (words1.size() != words2.size()) return false;
6         unordered_map<string, string> m;
7         for (auto pair : pairs) {
8             string x = getRoot(pair.first, m), y = getRoot(pair.second, m);
9             if (x != y) m[x] = y;
10        }
11        for (int i = 0; i < words1.size(); ++i) {
12            if (getRoot(words1[i], m) != getRoot(words2[i], m)) return false;
13        }
14        return true;
15    }
16    string getRoot(string word, unordered_map<string, string>& m) {
17        if (!m.count(word)) m[word] = word;
18        return word == m[word] ? word : getRoot(m[word], m);
19    }
};

```

724. 单调递增数字

Given a non-negative integer N, find the largest number that is less than or equal to N with monotone increasing digits.

(Recall that an integer has monotone increasing digits if and only if each pair of adjacent digits x and y satisfy x <= y.)

这道题给了我们一个非负数，让我们求一个数字小于等于给定数字，且该数字各位上的数字是单调递增的。那么我们就来分析题目中给的几个例子吧，首先如果是10的话，我们知道1大于0，所以不是单调自增的，那么返回的数就是9。第二个例子是1234，各位上已经满足单调自增的条件了，返回原数即可。第三个例子是332，我们发现最后一位2小于之前的3，那么此时我们将前面位减1，先变成322，再往前看，还是小于前面的3，那么我们再将前面位减1，就变成了222，此时222不是最大的单调递增数，我们可以将后面两位变成9，于是乎就有了299，小于给定的332，符合题意。如果给定的数字是232，那么就会得到229，我们可以发现规律，要找到从后往前遍历的最后一个值升高的位置，让前一位减1，并把当前位以及后面的所有位都变成9，就可以得到最大的单调递增数啦。

我们用j表示最后一个值升高的位置，具体来说应该是其前一位的值大，初始化为总位数n，然后从后往前遍历，因为每次要和前一位比较，为防止越界，应遍历到第二个数停止，如果当前位大于等于前一位，符合单调递增，直接跳过；否则就将前一位自减1，j赋值为当前位i，循环结束后，从j位到末尾的位数都改为9即可，参见代码如下：

```

1 class Solution {
2 public:
3     int monotoneIncreasingDigits(int N) {
4         string str = to_string(N);
5         int n = str.size(), j = n;
6         for (int i = n - 1; i > 0; --i) {
7             if (str[i] >= str[i - 1]) continue;
8             --str[i - 1];
9             j = i;
10        }
11        for (int i = j; i < n; ++i) {
12            str[i] = '9';
13        }
14        return stoi(str);
15    }
16 };

```

725. 日常温度

Given a list of daily temperatures, produce a list that, for each day in the input, tells you how many days you would have to wait until a warmer temperature. If there is no future day for which this is possible, put 0 instead.

For example, given the list temperatures = [73, 74, 75, 71, 69, 72, 76, 73], your output should be [1, 1, 4, 2, 1, 1, 0, 0].

Note: The length of temperatures will be in the range [1, 30000]. Each temperature will be an integer in the range [30, 100].

这道题给了我们一个数组，让我们找下一个比当前数字大的数字的距离，我们研究一下题目中给的例子，发现数组是无序的，所以没法用二分法快速定位下一个大的数字，那么最先考虑的方法就是暴力搜索了，写起来没有什么难度，但是OJ并不答应。实际上这道题应该使用递减栈Descending Stack来做，栈里只有递减元素，思路是这样的，我们遍历数组，如果栈不空，且当前数字大于栈顶元素，那么如果直接入栈的话就不是递减栈了，所以我们取出栈顶元素，那么由于当前数字大于栈顶元素的数字，而且一定是第一个大于栈顶元素的数，那么我们直接求出下标差就是二者的距离了，然后继续看新的栈顶元素，直到当前数字小于等于栈顶元素停止，然后将数字入栈，这样就可以一直保持递减栈，且每个数字和第一个大于它的数的距离也可以算出来了，参见代码如下：

```

1 class Solution {
2 public:
3     vector<int> dailyTemperatures(vector<int>& temperatures) {
4         int n = temperatures.size();
5         vector<int> res(n, 0);
6         stack<int> st;
7         for (int i = 0; i < temperatures.size(); ++i) {
8             while (!st.empty() && temperatures[i] > temperatures[st.top()]) {
9                 auto t = st.top(); st.pop();
10                res[t] = i - t;
11            }
12            st.push(i);
13        }
14        return res;
15    }
16 };

```

726. 删数与赚取

Given an array `nums` of integers, you can perform operations on the array.

In each operation, you pick any `nums[i]` and delete it to earn `nums[i]` points. After, you must delete every element equal to `nums[i] - 1` or `nums[i] + 1`.

You start with 0 points. Return the maximum number of points you can earn by applying such operations.

博主浪了整整一个圣诞假期，现在也该收收心了，2018了，今年对于博主是很关键的一年，有太多的事情要做，各种小目标需要完成，还有梦想去追逐，又要开始努力啦~在博主停更的这一周半的时间内，收到了网友们的私信和留言催更，请大家放心，2018年博主会继续坚持下去，继续追赶进度，虽然一直都没有完全追上--|||，照LeetCode这出题速度，今年题号有望突破一千大关啊，感觉碉堡了有木有，一起为了幸福而奋斗吧~

好了，来做题吧。这道题给了我们一个数组，每次让我们删除一个数字，删除的数字本身变为了积分累积，并且要同时移除之前数的加1和减1的数，但此时移除的数字不累计积分，让我们求最多能获得多少积分。博主最开始尝试的方法是积分大小来排列，先删除大的数字，但是不对。于是乎，博主发现相同的数字可以同时删除，于是就是建立了每个数字和其出现次数之间的映射，然后放到优先队列里，重写排序方式comparator为数字乘以其出现次数，先移除能产生最大积分的数字，可是还是不对。其实这道题跟之前那道House Robber的本质是一样的，那道题小偷不能偷相邻的房子，这道题相邻的数字不能累加积分，是不是一个道理？那么对于每一个数字，我们都有两个选择，拿或者不拿。如果我们拿了当前的数字，我们就不能拿之前的数字（如果我们从小往大遍历就不需要考虑后面的数字），那么当前的积分就是不拿前面的数字的积分加上当前数字之和。如果我们不拿当前的数字，那么对于前面的数字我们既可以拿也可以不拿，于是当前的积分就是拿前面的数字的积分和不拿前面数字的积分中的较大值。这里我们用take和skip分别表示拿与不拿上一个数字，takei和skipi分别表示拿与不拿当前数字，每次更新完当前的takei和skipi时，也要更新take和skip，为下一个数字做准备，最后只要返回take和skip中的较大值即可，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     int deleteAndEarn(vector<int>& nums) {
4         vector<int> sums(10001, 0);
5         int take = 0, skip = 0;
6         for (int num : nums) sums[num] += num;
7         for (int i = 0; i < 10001; ++i) {
8             int takei = skip + sums[i];
9             int skipi = max(skip, take);
10            take = takei; skip = skipi;
11        }
12        return max(skip, take);
13    }
14 }
```

CPP

下面这种解法直接使用sums数组来更新，而没有使用额外的变量。上面解法中没有讲解这个sums数组，这里的sums实际上相当于建立了数字和其总积分的映射，这里的总积分的计算方法是由数字乘以其出现次数得来的。由于题目中说了每个数字不会超过10000，所以sums的长度可以初始化为10001，然后遍历原数组，将遇到的数字都累加到该数字在数组中的位置上。然后从sums数组的第三个数字开始遍历，更新方法跟上面解法的思路很类似，当前的sums[i]值就等于前一个值sums[i-1]和前两个值sums[i-2]加上当前的sums[i]值中的较大值，其实思想就是在不拿当前数的积分，跟不拿前一个数的积分加上当前的积分之和，取二者中的较大值更新当前值sums[i]，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int deleteAndEarn(vector<int>& nums) {
4         vector<int> sums(10001, 0);
5         for (int num : nums) sums[num] += num;
6         for (int i = 2; i < 10001; ++i) {
7             sums[i] = max(sums[i - 1], sums[i - 2] + sums[i]);
8         }
9         return sums[10000];
10    }
11 };

```

727. 捡樱桃

In a $N \times N$ grid representing a field of cherries, each cell is one of three possible integers.

0 means the cell is empty, so you can pass through;

1 means the cell contains a cherry, that you can pick up and pass through;

-1 means the cell contains a thorn that blocks your way.

Your task is to collect maximum number of cherries possible by following the rules below:

Starting at the position $(0, 0)$ and reaching $(N-1, N-1)$ by moving right or down through valid path cells (cells with value 0 or 1);

After reaching $(N-1, N-1)$, returning to $(0, 0)$ by moving left or up through valid path cells;

When passing through a path cell containing a cherry, you pick it up and the cell becomes an empty cell (0);

If there is no valid path between $(0, 0)$ and $(N-1, N-1)$, then no cherries can be collected.

这道题给了我们一个二维数组，每个数字只有三个数字，-1，0，和1，其中-1表示障碍物不能通过，1表示有樱桃并可以通过，0表示没有樱桃并可以通过，并设定左上角为起点，右下角为终点，让我们从起点走到终点，再从终点返回起点，求最多能捡的樱桃的个数，限定起点和终点都没有障碍物。博主开始想的是就用dp来做呗，先从起点走到终点，求最多能捡多个樱桃，然后将捡起樱桃后将grid值变为0，然后再走一遍，把两次得到的樱桃数相加即可，但是类似贪婪算法的dp解法却跪在了下面这个case：

```
1 1 1 1 0 0 0
0 0 0 1 0 0 0
0 0 0 1 0 0 1
1 0 0 1 0 0 0
0 0 0 1 0 0 0
0 0 0 1 0 0 0
0 0 0 1 1 1 1
```

[复制代码](#)

我们可以看出，红色的轨迹是第一次dp解法走过的路径，共拿到了13个樱桃，但是回到起点的话，剩下的两个樱桃无论如何也不可能同时拿到，只能拿到1颗，所以总共只能捡到14颗樱桃，而实际上所有的樱桃都可以捡到，需要换个走法的话，比如下面这种走法：

[复制代码](#)

```
1 1 1 1 0 0 0
0 0 0 1 0 0 0
0 0 0 1 0 0 1
1 0 0 1 0 0 0
0 0 0 1 0 0 0
0 0 0 1 0 0 0
0 0 0 1 1 1 1
```

[复制代码](#)

红色为从起点到终点的走法，共拿到9颗樱桃，回去走蓝色的路径，可拿到6颗樱桃，所以总共15颗都能收入囊中。那这是怎么回事，原因出在了我们的dp递推式的设计上，博主之前设计式，当前位置的樱桃数跟上边和左边的樱桃数有关，取二者的较大值，如果只是从起点到终点走单程的话，这种设计是没有问题的，可以拿到最多的樱桃，但如果是round trip的话，那么就不行了。这里参考的还是fun4LeetCode大神的帖子，范佛利特扣德大神的帖子每次讲解都写的巨详细，总是让博主有种读paper的感觉。博主就挑选部分来讲讲，完整版可以自己去读一读大神的亲笔～

最开始时博主定义的 $dp[i][j]$ 为单程的，即到达 (i, j) 位置能捡到的最大樱桃数，即：

$T(i, j) = grid[i][j] + \max\{ T(i-1, j), T(i, j-1) \}$

但是定义单程就得改变grid的值，再进行一次dp计算时，就会陷入之前例子中的陷阱。所以我们的 $dp[i][j]$ 还是需要定义为round trip的，即到达 (i, j) 位置并返回起点时能捡到的最大樱桃数，但是新的问题就来了，樱桃只有一个，只能捡一次，去程捡了，返程就不能再捡了，如何才能避免重复计算呢？我们只有 i 和 j 是不够的，其只能定义去程的位置，我们还需要 p, g ，（不是pgone哈哈），来定义返程的位置，那么重现关系Recurrence Relations就变成了 $T(i, j, p, g)$ ，我们有分别两种方式离开 (i, j) 和 (p, g) ，我们suppose时从终点往起点遍历，那么就有4种情况：

Case 1: $(0, 0) \Rightarrow (i-1, j) \Rightarrow (i, j); (p, q) \Rightarrow (p-1, q) \Rightarrow (0, 0)$
Case 2: $(0, 0) \Rightarrow (i-1, j) \Rightarrow (i, j); (p, q) \Rightarrow (p, q-1) \Rightarrow (0, 0)$
Case 3: $(0, 0) \Rightarrow (i, j-1) \Rightarrow (i, j); (p, q) \Rightarrow (p-1, q) \Rightarrow (0, 0)$
Case 4: $(0, 0) \Rightarrow (i, j-1) \Rightarrow (i, j); (p, q) \Rightarrow (p, q-1) \Rightarrow (0, 0)$

根据定义，我们有：

Case 1 is equivalent to $T(i-1, j, p-1, q) + grid[i][j] + grid[p][q];$
Case 2 is equivalent to $T(i-1, j, p, q-1) + grid[i][j] + grid[p][q];$
Case 3 is equivalent to $T(i, j-1, p-1, q) + grid[i][j] + grid[p][q];$
Case 4 is equivalent to $T(i, j-1, p, q-1) + grid[i][j] + grid[p][q];$

因此，我们的重现关系可以写作：

$T(i, j, p, q) = grid[i][j] + grid[p][q] + \max\{ T(i-1, j, p-1, q), T(i-1, j, p, q-1), T(i, j-1, p-1, q), T(i, j-1, p, q-1) \}$

为了避免重复计算，我们希望 $grid[i][j]$ 和 $grid[p][q]$ 不出现在 $T(i-1, j, p-1, q), T(i-1, j, p, q-1), T(i, j-1, p-1, q)$ 和 $T(i, j-1, p, q-1)$ 中的任意一个上。显而易见的是 (i, j) 不会出现在 $(0, 0) \Rightarrow (i-1, j)$ 或 $(0, 0) \Rightarrow (i, j-1)$ 的路径上，同理， (p, q) 也不会出现在 $(p-1, q) \Rightarrow (0, 0)$ 或 $(p, q-1) \Rightarrow (0, 0)$

的路径上。因此，我们需要保证 (i, j) 不会出现在 $(p-1, q) \Rightarrow (0, 0)$ 或 $(p, q-1) \Rightarrow (0, 0)$ 的路径上，同时 (p, g) 不会出现在 $(0, 0) \Rightarrow (i-1, j)$ 或 $(0, 0) \Rightarrow (i, j-1)$ 的路径上，怎么做呢？

我们观察到 $(0, 0) \Rightarrow (i-1, j)$ 和 $(0, 0) \Rightarrow (i, j-1)$ 的所有点都在矩形 $[0, 0, i, j]$ 中（除了右下角点 (i, j) 点），所以只要 (p, g) 不在矩形 $[0, 0, i, j]$ 中就行了，注意 (p, g) 和 (i, j) 是有可能重合的，这种情况特殊处理一下就行了。同理， (i, j) 也不能在矩形 $[0, 0, p, g]$ 中，那么以下三个条件中需要满足一个：

```
i < p && j > q
i == p && j == q
i > p && j < q
```

为了满足上述条件，我们希望当 i 或 p 增加的时候， j 或 q 减小，那么我们可以有这个等式：

```
k = i + j = p + q
```

其中 k 为从起点开始走的步数，所以我们可以用 $T(k, i, p)$ 来代替 $T(i, j, p, g)$ ，那么我们的重现关系式就变成了：

$T(k, i, p) = grid[i][k-i] + grid[p][k-p] + \max\{T(k-1, i-1, p-1), T(k-1, i-1, p), T(k-1, i, p-1), T(k-1, i, p)\}.$

当 $i == p$ 时， $grid[i][k-i]$ 和 $grid[p][k-p]$ 就相等了，此时只能加一个。我们注意到 i, j, p, q 的范围是 $[0, n]$ ，意味着 k 只能在范围 $[0, 2n - 1]$ 中， 初始化时 $T(0, 0, 0) = grid[0][0]$ 。我们这里的重现关系 T 虽然是三维的，但是我们可以用二维dp数组来实现，因为第 k 步的值只依赖于第 $k-1$ 步的情况，参见代码如下：

```
1 class Solution {
2 public:
3     int cherryPickup(vector<vector<int>>& grid) {
4         int n = grid.size(), mx = 2 * n - 1;
5         vector<vector<int>> dp(n, vector<int>(n, -1));
6         dp[0][0] = grid[0][0];
7         for (int k = 1; k < mx; ++k) {
8             for (int i = n - 1; i >= 0; --i) {
9                 for (int p = n - 1; p >= 0; --p) {
10                     int j = k - i, q = k - p;
11                     if (j < 0 || j >= n || q < 0 || q >= n || grid[i][j] < 0 || grid[p][q]
12 < 0) {
13                         dp[i][p] = -1;
14                         continue;
15                     }
16                     if (i > 0) dp[i][p] = max(dp[i][p], dp[i - 1][p]);
17                     if (p > 0) dp[i][p] = max(dp[i][p], dp[i][p - 1]);
18                     if (i > 0 && p > 0) dp[i][p] = max(dp[i][p], dp[i - 1][p - 1]);
19                     if (dp[i][p] >= 0) dp[i][p] += grid[i][j] + (i != p ? grid[p][q] : 0);
20                 }
21             }
22         }
23         return max(dp[n - 1][n - 1], 0);
24     }
};
```

CPP

728. 二叉树中最近的叶结点

Given a binary tree where every node has a unique value, and a target key k , find the value of the nearest leaf node to target k in the tree.

Here, nearest to a leaf means the least number of edges travelled on the binary tree to reach any leaf of the tree. Also, a node is called a leaf if it has no children.

In the following examples, the input tree is represented in flattened form row by row. The actual root tree given will be a `TreeNode` object.

这道题让我们找二叉树中最近的叶结点，叶结点就是最底端没有子结点的那个。我们观察题目中的例子3，发现结点2的最近叶结点是其右边的那个结点3，那么传统的二叉树的遍历只能去找其子结点中的叶结点，像这种同一层水平的结点该怎么弄呢？我们知道树的本质就是一种无向图，但是树只提供了父结点到子结点的连接，反过来就不行了，所以只要我们建立了反向连接，就可以用BFS来找最近的叶结点了。明白了这一点后，我们就先来做反向连接吧，用一个哈希map，建立子结点与其父结点之间的映射，其实我们不用做完所有的反向连接，而是做到要求的结点k就行了，因为结点k的子结点可以直接访问，不需要再反过来查找。我们用DFS来遍历结点，并做反向连接，直到遇到结点k时，将其返回。此时我们得到了结点k，并且做好了结点k上面所有结点的反向连接，那么就可以用BFS来找最近的叶结点了，将结点k加入队列queue和已访问集合visited中，然后开始循环，每次取出队首元素，如果是叶结点，说明已经找到了最近叶结点，直接返回；如果左子结点存在，并且不在visited集合中，那么先将其加入集合，然后再加入队列，同理，如果右子结点存在，并且不在visited集合中，那么先将其加入集合，然后再加入队列；再来看其父结点，如果不在visited集合中，那么先将其加入集合，然后再加入队列。因为题目中说了一定会有结点k，所以在循环内部就可以直接返回了，不会有退出循环的可能，但是为表尊重，我们最后还是加上return -1吧，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int findClosestLeaf(TreeNode* root, int k) {
4         unordered_map<TreeNode*, TreeNode*> back;
5         TreeNode *kNode = find(root, k, back);
6         queue<TreeNode*> q{{kNode}};
7         unordered_set<TreeNode*> visited{{kNode}};
8         while (!q.empty()) {
9             TreeNode *t = q.front(); q.pop();
10            if (!t->left && !t->right) return t->val;
11            if (t->left && !visited.count(t->left)) {
12                visited.insert(t->left);
13                q.push(t->left);
14            }
15            if (t->right && !visited.count(t->right)) {
16                visited.insert(t->right);
17                q.push(t->right);
18            }
19            if (back.count(t) && !visited.count(back[t])) {
20                visited.insert(back[t]);
21                q.push(back[t]);
22            }
23        }
24        return -1;
25    }
26    TreeNode* find(TreeNode* node, int k, unordered_map<TreeNode*, TreeNode*>& back) {
27        if (node->val == k) return node;
28        if (node->left) {
29            back[node->left] = node;
30            TreeNode *left = find(node->left, k, back);
31            if (left) return left;
32        }
33        if (node->right) {
34            back[node->right] = node;
35            TreeNode *right = find(node->right, k, back);
36            if (right) return right;
37        }
38        return NULL;
39    }
40 };

```

CPP

下面这种解法也挺巧妙的，虽然没有像上面的解法那样建立所有父结点的反向连接，但是这种解法直接提前算出来了所有父结点到结点k的距离，就比如说例子3中，结点k的父结点只有一个，即为结点1，那么算出其和结点k的距离为1，即建立结点1和距离1之间的映射，另外建立结点k和0之间的映射，这样便于从结点k开始像叶结点统计距离。接下来，我们维护一个最小值mn，表示结点k到叶结点的最小距离，还有结果res，指向那个最小距离的叶结点。下面就开始再次遍历二叉树了，如果当前结点为空，直接返回。否则先在哈希map中看当前结点是否有映射值，有的话就取出来（如果有，则说明当前结点可能k或者其父结点），如果当前结点是叶结点了，那么我们要用当前距离cur和最小距离mn比较，如果cur更小的话，就将mn更新为cur，将结果res更新为当前结点。否则就对其左右子结点调用递归函数，注意cur要加1，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int findClosestLeaf(TreeNode* root, int k) {
4         int res = -1, mn = INT_MAX;
5         unordered_map<int, int> m;
6         m[k] = 0;
7         find(root, k, m);
8         helper(root, -1, m, mn, res);
9         return res;
10    }
11    int find(TreeNode* node, int k, unordered_map<int, int>& m) {
12        if (!node) return -1;
13        if (node->val == k) return 1;
14        int r = find(node->left, k, m);
15        if (r != -1) {
16            m[node->val] = r;
17            return r + 1;
18        }
19        r = find(node->right, k, m);
20        if (r != -1) {
21            m[node->val] = r;
22            return r + 1;
23        }
24        return -1;
25    }
26    void helper(TreeNode* node, int cur, unordered_map<int, int>& m, int& mn, int& res) {
27        if (!node) return;
28        if (m.count(node->val)) cur = m[node->val];
29        if (!node->left && !node->right) {
30            if (mn > cur) {
31                mn = cur;
32                res = node->val;
33            }
34        }
35        helper(node->left, cur + 1, m, mn, res);
36        helper(node->right, cur + 1, m, mn, res);
37    }
38 };

```

729. 网络延迟时间

There are N network nodes, labelled 1 to N.

Given times, a list of travel times as directed edges times[i] = (u, v, w), where u is the source node, v is the target node, and w is the time it takes for a signal to travel from source to target.

Now, we send a signal from a certain node K. How long will it take for all nodes to receive the signal? If it is impossible, return -1

这道题给了我们一些有向边，又给了一个结点K，问至少需要多少时间才能从K到达任何一个结点。这实际上是一个有向图求最短路径的问题，我们求出K点到每一个点到最短路径，然后取其中最大的一个就是需要的时间了。可以想成从结点K开始有水流向周围扩散，当水流到达最远的一个结点时，那么其他所有的结点一定已经流过水了。最短路径的常用解法有迪杰克斯特拉算法Dijkstra Algorithm, 弗洛伊德算法Floyd-Warshall Algorithm, 和贝尔曼福特算法Bellman-Ford Algorithm, 其中, Floyd算法是多源最短路径，即求任意点到任意点到最短路径，而Dijkstra算法和Bellman-Ford算法是单源最短路径，即单个点到任意点到最短路径。这里因为起点只有一个K，所以使用单源最短路径就行了。这三种算法还有一点不同，就是Dijkstra算法处理有向权重图时，权重必须为正，而另外两种可以处理负权重有向图，但是不能出现负环，所谓负环，就是权重均为负的环。为啥呢，这里要先引入松弛操作Relaxtion，这是这三个算法的核心思想，当有对边(u, v)是结点u到结点v，如果 $dist(v) > dist(u) + w(u, v)$ ，那么 $dist(v)$ 就可以被更新，这是所有这些的算法的核心操作。Dijkstra算法是以起点为中心，向外层层扩展，直到扩展到终点为止。根据这特性，用BFS来实现时再好不过了，注意while循环里的第一层for循环，这保证了每一层的结点先被处理完，才会进入进入下一层，这种特性在用BFS遍历迷宫统计步数的时候很重要。对于每一个结点，我们都跟其周围的结点进行Relaxtion操作，从而更新周围结点的距离值。为了防止重复比较，我们需要使用visited数组来记录已访问过的结点，最后我们在所有的最小路径中选最大的返回，注意，如果结果res为INT_MAX，说明有些结点是无法到达的，返回-1。普通的实现方法的时间复杂度为O(V²)，基于优先队列的实现方法的时间复杂度为O(E + VlogV)，其中V和E分别为结点和边的个数，这里多说一句，Dijkstra算法这种类贪心算法的机制，使得其无法处理有负权重的最短距离，还好这道题的权重都是正数，参见代码如下：

解法1:

```

1 class Solution {
2 public:
3     int networkDelayTime(vector<vector<int>>& times, int N, int K) {
4         int res = 0;
5         vector<vector<int>> edges(101, vector<int>(101, -1));
6         queue<int> q{{K}};
7         vector<int> dist(N + 1, INT_MAX);
8         dist[K] = 0;
9         for (auto e : times) edges[e[0]][e[1]] = e[2];
10        while (!q.empty()) {
11            unordered_set<int> visited;
12            for (int i = q.size(); i > 0; --i) {
13                int u = q.front(); q.pop();
14                for (int v = 1; v <= 100; ++v) {
15                    if (edges[u][v] != -1 && dist[u] + edges[u][v] < dist[v]) {
16                        if (!visited.count(v)) {
17                            visited.insert(v);
18                            q.push(v);
19                        }
20                        dist[v] = dist[u] + edges[u][v];
21                    }
22                }
23            }
24        }
25        for (int i = 1; i <= N; ++i) {
26            res = max(res, dist[i]);
27        }
28        return res == INT_MAX ? -1 : res;
29    }
30 };

```

下面来看基于Bellman-Ford算法的解法，时间复杂度是 $O(VE)$ ， V 和 E 分别是结点和边的个数。这种算法是基于DP来求全局最优解，原理是对图进行 $V - 1$ 次松弛操作，这里的 V 是所有结点的个数（为啥是 $V-1$ 次呢，因为最短路径最多只有 $V-1$ 条边，所以只需循环 $V-1$ 次），在重复计算中，使得每个结点的距离被不停的更新，直到获得最小的距离，这种设计方法融合了暴力搜索之美，写法简洁又不失优雅。之前提到了，Bellman-Ford算法可以处理负权重的情况，但是不能有负环存在，一般形式的写法中最后一部分是检测负环的，如果存在负环则报错。不能有负环原因是，每转一圈，权重和都在减小，可以无限转，那么最后的最小距离都是负无穷，无意义了。没有负环的话， $V-1$ 次循环后各点的最小距离应该已经收敛了，所以在检测负环时，就再循环一次，如果最小距离还能更新的话，就说明存在负环。这道题由于不存在负权重，所以就不检测了，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int networkDelayTime(vector<vector<int>>& times, int N, int K) {
4         int res = 0;
5         vector<int> dist(N + 1, INT_MAX);
6         dist[K] = 0;
7         for (int i = 1; i < N; ++i) {
8             for (auto e : times) {
9                 int u = e[0], v = e[1], w = e[2];
10                if (dist[u] != INT_MAX && dist[v] > dist[u] + w) {
11                    dist[v] = dist[u] + w;
12                }
13            }
14        }
15        for (int i = 1; i <= N; ++i) {
16            res = max(res, dist[i]);
17        }
18        return res == INT_MAX ? -1 : res;
19    }
20 };

```

730. 找比目标值大的最小字母

Given a list of sorted characters letters containing only lowercase letters, and given a target letter target, find the smallest element in the list that is larger than the given target.

Latters also wrap around. For example, if the target is target = 'z' and letters = ['a', 'b'], the answer is 'a'.

这道题给了我们一堆有序的字母，然后又给了我们一个target字母，让我们求字母数组中第一个大于target的字母，数组是循环的，如果没有，那就返回第一个字母。像这种在有序数组中找数字，二分法简直不要太适合啊。题目中说了数组至少有两个元素，那么我们首先用数组的尾元素来跟target比较，如果target大于等于尾元素的话，直接返回数组的首元素即可。否则就利用二分法来做，这里是查找第一个大于目标值的数组，博主之前做过二分法的总结，参见这个帖子[LeetCode Binary Search Summary](#) 二分搜索法小结，参见代码如下：

*解法1:

[[source, cpp, lineenums]]

```

class Solution {
public:
    char nextGreatestLetter(vector<char>& letters, char target) {
        if (target >= letters.back()) return letters[0];
        int n = letters.size(), left = 0, right = n;
        while (left < right) {
            int mid = left + (right - left) / 2;
            if (letters[mid] <= target) left = mid + 1;
            else right = mid;
        }
        return letters[right];
    }
};

```

我们也可以用STL自带的upper_bound函数来做，这个就是找第一个大于目标值的数字，如果返回end0，说明没找到，返回首元素即可，参见代码如下：

解法2：

```
1 class Solution {
2 public:
3     char nextGreatestLetter(vector<char>& letters, char target) {
4         auto it = upper_bound(letters.begin(), letters.end(), target);
5         return it == letters.end() ? *letters.begin() : *it;
6     }
7 };

```

CPP

731. 前后缀搜索

Given many words, words[i] has weight i.

Design a class WordFilter that supports one function, WordFilter.f(String prefix, String suffix). It will return the word with given prefix and suffix with maximum weight. If no word exists, return -1.

这道题给了我们一些单词，让我们通过输入单词的前缀和后缀来查找单词的位置。单词的位置就是其权重值，如果给定的前后缀能对应到不只一个单词，那么返回最大的权重。首先，一个单词如果长度为n的话，那么其就有n个前缀，比如对于单词apple，其前缀即为"a", "ap", "app", "appl", "apple"，同理，后缀也有n个。那么其组成的情况就有n²个，所以最简单的方法就是把这n²个前后缀组成一个字符串，和当前权重建立映射。如果后面的单词有相同的前后缀，直接用后面的大权重来覆盖之前的权重即可。为了将前后缀encode成一个字符串，我们可以在中间加上一个非字母字符，比如'#'，然后在查找的时候，我们先拼出“前缀#后缀”字符串，直接去哈希map中找即可，这种解法的WordFilter函数时间复杂度为O(NL²)，其中N是单词个数、L是单词长度。f函数时间复杂度为O(1)，空间复杂度为O(NL²)，适合需要大量查找的情况下使用，参见代码如下：

解法1：

```
1 class WordFilter {
2 public:
3     WordFilter(vector<string> words) {
4         for (int k = 0; k < words.size(); ++k) {
5             for (int i = 0; i <= words[k].size(); ++i) {
6                 for (int j = 0; j <= words[k].size(); ++j) {
7                     m[words[k].substr(0, i) + "#" + words[k].substr(words[k].size() - j)] =
8                         k;
9                 }
10            }
11        }
12    }
13
14    int f(string prefix, string suffix) {
15        return (m.count(prefix + "#" + suffix)) ? m[prefix + "#" + suffix] : -1;
16    }
17
18 private:
19     unordered_map<string, int> m;
20 };

```

CPP

如果我们希望节省一些空间的话，可以使用下面的方法。使用两个哈希map，一个建立所有前缀和权重数组之间的映射，另一个建立所有后缀和权重数组之间的映射。在WordFilter函数中，我们遍历每个单词，然后先遍历其所有前缀，将遍历到的前缀的映射数组中加入当前权重，同理再遍历其所有后缀，将遍历到的后缀的映射数组中加入当前权重。在搜索函数f中，首先判断，如果前缀或后缀不存在的话，直接返回-1。否则我们分别把前缀和后缀的权重数组取出来，然后用两个指针i和j，分别指向数组的

最后一个位置。当i和j不小于0时进行循环，如果两者的权重相等，直接返回，如果前缀的权重数组值大，则j自减1，反之i自减1，这种解法的WordFilter函数时间复杂度为O(NL)，其中N是单词个数，L是单词长度。f函数时间复杂度为O(N)，空间复杂度为O(NL)，参见代码如下：

解法2：

```

1 class WordFilter {
2 public:
3     WordFilter(vector<string> words) {
4         for (int k = 0; k < words.size(); ++k) {
5             for (int i = 0; i <= words[k].size(); ++i) {
6                 mp[words[k].substr(0, i)].push_back(k);
7             }
8             for (int i = 0; i <= words[k].size(); ++i) {
9                 ms[words[k].substr(words[k].size() - i)].push_back(k);
10            }
11        }
12    }
13
14    int f(string prefix, string suffix) {
15        if (!mp.count(prefix) || !ms.count(suffix)) return -1;
16        vector<int> pre = mp[prefix], suf = ms[suffix];
17        int i = pre.size() - 1, j = suf.size() - 1;
18        while (i >= 0 && j >= 0) {
19            if (pre[i] < suf[j]) --j;
20            else if (pre[i] > suf[j]) --i;
21            else return pre[i];
22        }
23        return -1;
24    }
25
26 private:
27     unordered_map<string, vector<int>> mp, ms;
28 };

```

732. 爬楼梯的最小损失

On a staircase, the i -th step has some non-negative cost $\text{cost}[i]$ assigned (0 indexed).

Once you pay the cost, you can either climb one or two steps. You need to find minimum cost to reach the top of the floor, and you can either start from the step with index 0, or the step with index 1.

这道题应该算是之前那道Climbing Stairs的拓展，这里不是求步数，而是每个台阶上都有一个cost，让我们求爬到顶端的最小cost是多少。换汤不换药，还是用动态规划Dynamic Programming来做。这里我们定义一个一维的dp数组，其中 $\text{dp}[i]$ 表示爬到第*i*层的最小cost，然后我们来想 $\text{dp}[i]$ 如何推导。我们来思考一下如何才能到第*i*层呢？是不是只有两种可能性，一个是从第*i*-2层上直接跳上来，一个是从第*i*-1层上跳上来。不会再有别的方法，所以我们的 $\text{dp}[i]$ 只和前两层有关系，所以可以写做如下：

```
dp[i] = min(dp[i- 2] + cost[i - 2], dp[i - 1] + cost[i - 1])
```

最后我们返回最后一个数字 $\text{dp}[n]$ 即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int minCostClimbingStairs(vector<int>& cost) {
4         int n = cost.size();
5         vector<int> dp(n + 1, 0);
6         for (int i = 2; i < n + 1; ++i) {
7             dp[i] = min(dp[i - 2] + cost[i - 2], dp[i - 1] + cost[i - 1]);
8         }
9         return dp.back();
10    }
11 };

```

再来看一种DP的解法，跟上面的解法很相近，不同在于dp数组长度为n，其中dp[i]表示到第i+1层的最小cost，分别初始化dp[0]和dp[1]为cost[0]和cost[1]。然后从i=2处开始遍历，此时我们的更新思路是，要爬当前的台阶，肯定需要加上当前的cost[i]，那么我们还是要从前一层或者前两层的台阶上跳上来，那么我们选择dp值小的那个，所以递归式如下：

```
dp[i] = cost[i] + min(dp[i - 1], dp[i - 2])
```

最后我们在最后两个dp值中选择一个较小的返回即可，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int minCostClimbingStairs(vector<int>& cost) {
4         int n = cost.size();
5         vector<int> dp(n, 0);
6         dp[0] = cost[0]; dp[1] = cost[1];
7         for (int i = 2; i < n; ++i) {
8             dp[i] = cost[i] + min(dp[i - 1], dp[i - 2]);
9         }
10        return min(dp[n - 1], dp[n - 2]);
11    }
12 };

```

我们可以对空间复杂度进行优化，通过前面的分析我们可以发现，当前的dp值仅仅依赖前面两个的值，所以我们不必把整个dp数组都记录下来，只需用两个变量a和b来记录前两个值，然后不停的用新得到的值来覆盖它们就好了。我们初始化a和b均为0，然后遍历cost数组，首先将a和b中较小值加上num放入临时变量t中，然后把b赋给a，把t赋给b即可，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int minCostClimbingStairs(vector<int>& cost) {
4         int a = 0, b = 0;
5         for (int num : cost) {
6             int t = min(a, b) + num;
7             a = b;
8             b = t;
9         }
10        return min(a, b);
11    }
12 };

```

我们还可以用递归来写，需要优化计算量，即用哈希map来保存已经算过了台阶，用的还是dp的思想，参见代码如下：

解法4：

```
1 class Solution {
2 public:
3     int minCostClimbingStairs(vector<int>& cost) {
4         unordered_map<int, int> memo;
5         return helper(cost, cost.size(), memo);
6     }
7     int helper(vector<int>& cost, int i, unordered_map<int, int>& memo) {
8         if (memo.count(i)) return memo[i];
9         if (i <= 1) return memo[i] = cost[i];
10        return memo[i] = (i == cost.size() ? 0 : cost[i]) + min(helper(cost, i - 1, memo),
11 helper(cost, i - 2, memo));
12    }
13 }
```

CPP

733. 至少是其他数字两倍的最大数

In a given integer array nums, there is always exactly one largest element.

Find whether the largest element in the array is at least twice as much as every other number in the array.

If it is, return the index of the largest element, otherwise return -1.

这道题让我们找一个至少是其他数字两倍的最大数字，那么我们想，首先明确的一点是这个要求的数字一定是数组中的最大数字，因为其是其他所有的数字的至少两倍。然后就是，如果该数字是数组中第二大的数字至少两倍的话，那么它一定是其他所有数字的至少两倍，所以我们可以遍历一次数组分别求出最大数字和第二大数字，然后判断一下最大数字是否是第二大数字的两倍即可，注意这里我们判断两倍的方法并不是直接相除，为了避免除以零的情况，我们采用减法，参见代码如下：

解法1：

```
1 class Solution {
2 public:
3     int dominantIndex(vector<int>& nums) {
4         int mx = INT_MIN, secondMx = INT_MIN, mxId = 0;
5         for (int i = 0; i < nums.size(); ++i) {
6             if (nums[i] > mx) {
7                 secondMx = mx;
8                 mx = nums[i];
9                 mxId = i;
10            } else if (nums[i] > secondMx) {
11                secondMx = nums[i];
12            }
13        }
14        return (mx - secondMx >= secondMx) ? mxId : -1;
15    }
16 }
17 }
```

CPP

当然我们也可以使用更straightforward的方法，首先遍历一遍数组找出最大数字，然后再遍历一遍数组，验证这个数字是否是其他数字的至少两倍，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int dominantIndex(vector<int>& nums) {
4         int mx = INT_MIN, mxId = 0;
5         for (int i = 0; i < nums.size(); ++i) {
6             if (mx < nums[i]) {
7                 mx = nums[i];
8                 mxId = i;
9             }
10        }
11        for (int num : nums) {
12            if (mx != num && mx - num < num) return -1;
13        }
14        return mxId;
15    }
16 };

```

734. 最短完整的单词

Find the minimum length word from a given dictionary words, which has all the letters from the string licensePlate. Such a word is said to complete the given string licensePlate

Here, for letters we ignore case. For example, "P" on the licensePlate still matches "p" on the word.

It is guaranteed an answer exists. If there are multiple answers, return the one that occurs first in the array.

The license plate might have the same letter occurring multiple times. For example, given a licensePlate of "PP", the word "pair" does not complete the licensePlate, but the word "supper" does.

这道题给了我们一个车牌号，还有一些单词，让我们找出包含这个车牌号中所有字母的第一个最短的单词。车牌中的字母有大小写之分，但是单词只是由小写单词组成的，所以需要把车牌号中的所有大写字母都转为小写的，转换方法很简单，ASCII码加上32即可。我们建立车牌中各个字母和其出现的次数之间的映射，同时记录所有字母的个数total，然后遍历所有的单词，对于每个单词都要单独处理，我们遍历单词中所有的字母，如果其在车牌中也出现了，则对应字母的映射减1，同时还需匹配的字母数cnt也自减1，最后遍历字母完成后，如果cnt为0（说明车牌中所有的字母都在单词中出现了），并且结果res为空或长度大于当前单词word的话，更新结果即可，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     string shortestCompletingWord(string licensePlate, vector<string>& words) {
4         string res = "";
5         int total = 0;
6         unordered_map<char, int> freq;
7         for (char c : licensePlate) {
8             if (c >= 'a' && c <= 'z') {++freq[c]; ++total;}
9             else if (c >= 'A' && c <= 'Z') {++freq[c + 32]; ++total;}
10        }
11        for (string word : words) {
12            int cnt = total;
13            unordered_map<char, int> t = freq;
14            for (char c : word) {
15                if (--t[c] >= 0) --cnt;
16            }
17            if (cnt == 0 && (res.empty() || res.size() > word.size())) {
18                res = word;
19            }
20        }
21        return res;
22    }
23 };

```

如果这道题的单词是按长度排序的话，那么上面的方法就不是很高效率了，因为其会强制遍历完所有的单词。所以我们考虑给单词排序，博主这里用了TreeMap这个数据结构建立单词长度和包含所有该长度单词的数组之间的映射，其会自动按照单词长度来排序。然后还使用了一个chars数组来记录车牌中的所有字母，这样就可以方便的统计出字母总个数。我们从单词长度等于字母总个数的映射开始遍历，先检验该长度的所有单词。这里检验方法跟上面略有不同，但都大同小异，用一个bool型变量succ，初始化为true，然后建立一个字母及其出现次数的映射，先遍历单词，统计各个字母出现的次数。然后就遍历chars数组，如果chars中某个字母不在单词中，那么succ赋值为false，然后break掉。最后我们看succ，如果仍为true，直接返回当前单词word，之后的单词就不用再检验了，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     string shortestCompletingWord(string licensePlate, vector<string>& words) {
4         map<int, vector<string>> m;
5         vector<char> chars;
6         for (string word : words) {
7             m[word.size()].push_back(word);
8         }
9         for (char c : licensePlate) {
10            if (c >= 'a' && c <= 'z') chars.push_back(c);
11            else if (c >= 'A' && c <= 'Z') chars.push_back(c + 32);
12        }
13        for (auto a : m) {
14            if (a.first < chars.size()) continue;
15            for (string word : a.second) {
16                bool succ = true;
17                unordered_map<char, int> freq;
18                for (char c : word) ++freq[c];
19                for (char c : chars) {
20                    if (--freq[c] < 0) {succ = false; break;}
21                }
22                if (succ) return word;
23            }
24        }
25        return "";
26    }
27 };

```

735. 边角矩形的数量

Given a grid where each entry is only 0 or 1, find the number of corner rectangles.

A corner rectangle is 4 distinct 1s on the grid that form an axis-aligned rectangle. Note that only the corners need to have the value 1. Also, all four 1s used must be distinct.

这道题给了我们一个由0和1组成的二维数组，这里定义了一种边角矩形，其四个顶点均为1，让我们求这个二维数组中有多少个不同的边角矩形。那么最简单直接的方法就是暴力破解啦，我们遍历所有的子矩形，并且检验其四个顶点是否为1即可。先确定左上顶点，每个顶点都可以当作左上顶点，所以需要两个for循环，然后我们直接跳过非1的左上顶点，接下来就是要确定右上顶点和左下顶点了，先用一个for循环确定左下顶点的位置，同理，如果左下顶点为0，直接跳过。再用一个for循环确定右上顶点的位置，如果右上顶点位置也确定了，那么此时四个顶点中确定了三个，右下顶点的位置也就确定了，此时如果右上和右下顶点均为1，则结果res自增1，参见代码如下：

解法1：

```

1 class Solution {
2 public:
3     int countCornerRectangles(vector<vector<int>>& grid) {
4         int m = grid.size(), n = grid[0].size(), res = 0;
5         for (int i = 0; i < m; ++i) {
6             for (int j = 0; j < n; ++j) {
7                 if (grid[i][j] == 0) continue;
8                 for (int h = 1; h < m - i; ++h) {
9                     if (grid[i + h][j] == 0) continue;
10                    for (int w = 1; w < n - j; ++w) {
11                        if (grid[i][j + w] == 1 && grid[i + h][j + w] == 1) ++res;
12                    }
13                }
14            }
15        }
16        return res;
17    }
18 };

```

我们来看一种优化了时间复杂度的方法，这种方法的原理是两行同时遍历，如果两行中相同列位置的值都为1，则计数器cnt自增1，那么最后就相当于有了($cnt - 1$)个相邻的格子，问题就转化为了求 $cnt - 1$ 个相邻的格子能组成多少个矩形，就变成了初中数学问题了，共有 $cnt * (cnt - 1) / 2$ 个，参见代码如下：

解法2：

```

1 class Solution {
2 public:
3     int countCornerRectangles(vector<vector<int>>& grid) {
4         int m = grid.size(), n = grid[0].size(), res = 0;
5         for (int i = 0; i < m; ++i) {
6             for (int j = i + 1; j < m; ++j) {
7                 int cnt = 0;
8                 for (int k = 0; k < n; ++k) {
9                     if (grid[i][k] == 1 && grid[j][k] == 1) ++cnt;
10                }
11                res += cnt * (cnt - 1) / 2;
12            }
13        }
14        return res;
15    }
16 };

```

下面这种解法由热心网友edyyy提供，最大亮点是将解法二的beat 65%提高到了beat 97%，速度杠杠的，要飞起来了的节奏。在遍历前一行的时候，将所有为1的位置存入到了一个数组ones中，然后在遍历其他行时，直接检测ones数组中的那些位置是否为1，这样省去了检查一些之前行为0的步骤，提高了运行速度，但是也牺牲了一些空间，比如需要ones数组，算是个trade off吧，参见代码如下：

解法3：

```

1 class Solution {
2 public:
3     int countCornerRectangles(vector<vector<int>>& grid) {
4         int m = grid.size(), n = grid[0].size(), res = 0;
5         for (int i = 0; i < m - 1; i++) {
6             vector<int> ones;
7             for (int k = 0; k < n; k++) if (grid[i][k]) ones.push_back(k);
8             for (int j = i + 1; j < m; j++) {
9                 int cnt = 0;
10                for (int l = 0; l < ones.size(); l++) {
11                    if (grid[j][ones[l]]) cnt++;
12                }
13                res += cnt * (cnt - 1) / 2;
14            }
15        }
16        return res;
17    }
18 };

```

736. 数组中的第k个最大元素

在未排序的数组中找到第 k 个最大的元素。请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。

示例 1:

输入: [3,2,1,5,6,4] 和 $k = 2$

输出: 5

示例 2:

输入: [3,2,3,1,2,4,5,5,6] 和 $k = 4$

输出: 4

说明:

你可以假设 k 总是有效的，且 $1 \leq k \leq$ 数组的长度。

解法1:

```

1 class Solution {
2 public:
3     int findKthLargest(vector<int>& nums, int k) {
4         if(k<=0 || k>nums.size()) return -1;
5         return bfprt(nums, 0, nums.size()-1, k);
6     }
7
8     int partition(vector<int>& nums, int left, int right, int pos){
9         swap(nums[left], nums[pos]);
10        int pivot = nums[left];
11        int l = left + 1, r = right;
12        while(l <= r){
13            if(nums[l] < pivot && nums[r] > pivot){
14                swap(nums[r--], nums[l++]);
15            }
16            if(nums[l] >= pivot)
17                ++l;
18            if(nums[r] <= pivot)
19                --r;
20        }
21
22        swap(nums[left], nums[r]);
23        return r;
24    }
25
26    int bfprt(vector<int>& nums, int left, int right, int k){
27        if(right - left + 1 <= 5){
28            sort(nums.begin()+left, nums.begin()+right+1);
29            return nums[right-k+1];
30        }
31
32        int t = left;
33        int count = (right - left + 1) / 5;
34        for(int i=0; i<count; ++i){
35            sort(nums.begin()+left+i*5, nums.begin()+left+(i+1)*5);
36            swap(nums[t], nums[left+i*5+2]);
37            ++t;
38        }
39        --t;
40
41        int pos = (left + t) / 2;
42        bfprt(nums, left, t, pos-left+1);
43
44        int m = partition(nums, left, right, pos);
45        if(m-left+1 == k)
46            return nums[m];
47        else if(m-left+1 < k)
48            return bfprt(nums, m+1, right, k-(m-left+1));
49        else
50            return bfprt(nums, left, m-1, k);
51    }
52};

```

解法2：

```
1 class Solution {
2 public:
3     int findKthLargest(vector<int>& nums, int k) {
4         if(k<=0 || k>nums.size()) return -1;
5
6         buildMinHeap(nums, k);
7         for(int i=k; i<nums.size(); ++i){
8             if(nums[i] > nums[0]){
9                 swap(nums[i], nums[0]);
10                modifyMinHeap(nums, k, 0);
11            }
12        }
13        return nums[0];
14    }
15    void modifyMinHeap(vector<int>& nums, int k, int i){
16        int left = 2*i+1, right = 2*i+2;
17        int small = i;
18        if(left<k && nums[small]>nums[left])
19            small = left;
20        if(right<k && nums[small]>nums[right])
21            small = right;
22
23        if(small != i){
24            swap(nums[small], nums[i]);
25            modifyMinHeap(nums, k, small);
26        }
27    }
28
29    void buildMinHeap(vector<int>& nums, int k){
30        for(int i=k-1; i>=0; --i)
31            modifyMinHeap(nums, k, i);
32    }
33};
```

解法3：

```

1 class Solution {
2 public:
3     int findKthLargest(vector<int>& nums, int k) {
4         if(k > nums.size() || k<=0) return -1;
5
6         int left = 0, right = nums.size() - 1;
7         while(true){
8             int pos = partition(nums, left, right);
9             if(pos == k-1) return nums[pos];
10            else if(pos > k-1) right = pos-1;
11            else left = pos + 1;
12        }
13    }
14
15    int partition(vector<int>& nums, int left, int right){
16        int pivot = nums[left];
17        int l = left + 1, r = right;
18        while(l <= r){
19            if(nums[l] < pivot && nums[r] > pivot){
20                swap(nums[r--], nums[l++]);
21            }
22            if(nums[l] >= pivot)
23                ++l;
24            if(nums[r] <= pivot)
25                --r;
26        }
27
28        swap(nums[left], nums[r]);
29        return r;
30    }
31 };

```

解法4:

```

1 class Solution {
2 public:
3     int findKthLargest(vector<int>& nums, int k) {
4         if(k<=0 || k>nums.size()) return -1;
5         priority_queue<int> q(nums.begin(), nums.end());
6         for(int i=0; i<k-1; ++i)
7             q.pop();
8
9         return q.top();
10    }
11 };

```

解法5:

```

1 class Solution {
2 public:
3     int findKthLargest(vector<int>& nums, int k) {
4         if(k<=0 || k>nums.size()) return -1;
5         sort(nums.begin(), nums.end());
6         return nums[nums.size() - k];
7     }
8 };

```

2018/11/7

Fighting for dream !

最后更新时间 2018-11-07 18:37:59 CST