

# 进军硅谷：程序员面试揭秘

陈东锋 著

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

## 内 容 简 介

本书介绍了如何在硅谷求职,以及剖析了具有代表性的 150 道热门硅谷公司的面试题,从面试技巧、基础知识、解题思路和效率优化等方面总结面试和解题规律。全书分为四部分共 19 章,包含出国工作途径、IT 求职准备等,以及常见数据结构、算法、大数据、系统设计和面向对象语言等方面的题目和解题思路,并提炼出解题的 5 个步骤:复述/提问、举例、观察、编码和测试。本书精选出的面试题是硅谷热门公司的高频题,可以用来作为面试前的练习。对于每道题,本书尽可能给出多种解法,对日常工作中遇到的问题也有一定启发性。

本书适合正在应聘程序员相关职位的就业人员阅读和参考,特别是打算寻求美国 IT 公司职位并想通过技术移民实现美国梦的程序员。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。  
版权所有,侵权必究。

### 图书在版编目(CIP)数据

进军硅谷:程序员面试揭秘 / 陈东锋著. —北京:电子工业出版社, 2014.3

ISBN 978-7-121-22566-6

I. ①进… II. ①陈… III. ①程序设计—工程技术人员—资格考试—自学参考资料  
IV. ①TP311.1

中国版本图书馆 CIP 数据核字(2014)第 038400 号

策划编辑:符隆美

责任编辑:徐津平

印 刷:北京中新伟业印刷有限公司

装 订:北京中新伟业印刷有限公司

出版发行:电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本:787×980 1/16 印张:20 字数:288 千字

印 次:2014 年 3 月第 1 次印刷

定 价:55.00 元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010) 88254888。

质量投诉请发邮件至 [zltz@phei.com.cn](mailto:zltz@phei.com.cn), 盗版侵权举报请发邮件至 [dbqq@phei.com.cn](mailto:dbqq@phei.com.cn)。

服务热线:(010) 88258888。

献给我的妻子 Emily 和女儿 Ella。

To my wife Emily and our daughter Ella.

# 前言

随着越来越多 IT 工程师寻找国外工作机会，介绍和总结国外热门 IT 公司面试过程及面试内容的需求尤为迫切。美国最新移民改革 CIR 方案更倾向于技术移民，这将使得今后会有更多国内程序员去美国工作。笔者亲身参与了国内和美国一些热门 IT 公司的面试，同时也作为面试官面试过不少人，熟知海内外 IT 公司招聘流程和面试方式。通常来说，去美国 IT 公司工作有三种途径。

- 直接申请美国公司职位，拿 H1B 签证工作。不少热门 IT 公司直接在国内招人，比如 Facebook、Twitter、Microsoft、Google 等。越来越多的程序员选择这条路，一方面是因为美国签证放宽了，另一方面是硅谷公司面试并没有比国内公司难多少。
- 在国内的跨国公司工作一年后，内部转组到美国的分部，使用 L1 签证。例如，从微软中国转至微软西雅图总部工作。
- 申请攻读美国学校的计算机科学硕士或博士学位，毕业后再找工作，即由 F1 签证转为 H1B 签证。

这三种途径都需要成功通过公司技术面试。热门 IT 企业的面试方式大致相同：1~2 轮电话面试，通过之后，又有 4~5 轮的现场面谈。其中 80% 的面试是技术面试，每

轮技术面试大约 45 分钟，扣除双方自我介绍和提问时间，花在技术面试的时间大约为 30 分钟。由于技术面试时间的限制，面试的题目一般不会太难，比大学生编程比赛（ACM）的题目简单很多，但是，面试者需要一些编程面试技巧，以及对算法、数据结构熟练掌握才能在限定时间内完成。这对要求在白板上写程序和无 Bug（Bug free）的公司来说尤其重要，比如 Facebook。

在编程面试过程中，光有解法却写不出来代码是行不通的，这只会让面试官觉得你只会夸夸其谈，不会编程而已。在编程面试里，切记“**让代码说话**”这条准则。在本书面试题相关的章节中，笔者贴出了面试题的全部代码，是为了更多时候让代码来说话。针对每道面试题，我们通常会有如下步骤。

- **复述/提问**：用自己的话复述面试官的题目，以免偏题。面试官给出的面试题并非一开始就很明确，需要多次问答来确定题意、边界条件、时间和数据结构限制等。
- **举例**：可以与提问同步进行，主要用来确认输入和输出结果。
- **观察**：通过举例来总结规律，思考可能使用到的结构和算法，然后设计一种你认为最优的算法。
- **编码**：和面试官沟通你的算法之后，开始在白板编码。
- **测试**：使用个别例子，把你的代码测试一遍。

在以上 5 个步骤里，看时间是否充裕，有些步骤可以省略。比如，如果面试官已经把问题说得很清楚了，那么复述可以省略。在本书当中，笔者也会按照这 5 个步骤的解题技巧来阐述面试题的解题方案。

笔者根据自身作为面试官的多年经历，并收集了网上众多的热门 IT 公司面试题目，精选了 150 道题来代表当前热门和高频的面试题。本书内容覆盖了基础的数据结构：数组、链表、树、堆栈、字符串等，以及高频率出现的算法，如动态规划、俩指针、排列组合、优先遍历等。本书的内容分为以下四个部分。

- 硅谷求职和面试：硅谷公司文化、技术移民、简历、面试和录用谈判。
- 常见数据结构：数组、链表、树和图、堆栈、字符串。
- 算法：动态规划、俩指针、优先遍历、哈希、排列组合。
- 杂项：系统设计、海量数据分析、面向对象设计、数学和位操作。

此外，附录还提供了数据结构和算法总结以及海量数据分析，以供读者快速查阅。

本书含有以下几个特点。

- 本书是市面上第一本介绍硅谷求职和技术移民美国的书。
- 精选出的面试题是硅谷各家热门公司的高频题，极其具有代表性。
- 总结了常见数据结构的对应算法，提炼出一套解题规律。对于类似题目，有着强烈的借鉴意义。
- 本书提供了完整的可运行的源代码。
- 对于每道题，本书尽可能给出多种解法，对我们在日常工作中遇到问题时有一定启发性。

虽然本书大部分的代码是用 Java 编写，但很容易转化为 C++/.NET 代码，因此，本书也适合 C++/.NET 程序员阅读。

由于本人水平有限，书中的题目并不能完全代表当前热门公司的编程面试的各个方面，虽然经过多轮审核，不少解法依然可能有漏洞或者错误，希望广大读者能给予指正。我已经搭建了一个关于程序员出国工作的网站“i 码工”(<http://www.imagong.com>)和读者交流。

在本书的写作过程中，我得到了很多朋友、同事的帮忙，包括汪纯子、周泽勇、俞明辉、吴盛萱、杨超、尹杭锋和于东东等。感谢他们帮忙校对文字、审核代码。同时，感谢电子工业出版社的工作人员，尤其是符隆美的帮助。感谢她大到全书的架构，小到文字的推敲，都给予了我极大的帮助，从而使本书的质量有了极大的提升。

最后，我要衷心地感谢我的妻子徐淼。感谢她在过去几年中对我的理解和支持，为我营造了一个温馨而浪漫的家，让我能够心无旁骛地写书。谨以此书献给她以及我们的女儿 Ella。

陈东锋

2013 年 10 月于上海张江

# 目 录

## 第一部分 硅谷求职

第 1 章 硅谷公司 .....	3
1.1 硅谷简介 .....	3
1.2 传奇旗帜 .....	7
1.2.1 微软 .....	8
1.2.2 谷歌 .....	9
1.2.3 亚马逊 .....	10
1.2.4 T witter .....	12
1.2.5 Ep ic .....	12
1.3 技术移民 .....	13
1.3.1 签证和绿卡 .....	14
1.3.2 税率和生活 .....	16
第 2 章 求职准备 .....	19
2.1 职位选择 .....	21



2.2 公司选择 .....	22
2.3 人际关系 .....	24
2.4 求职渠道 .....	27
<b>第 3 章 简历 .....</b>	<b>29</b>
3.1 简历特点 .....	30
3.2 简历结构 .....	33
3.3 简历优化 .....	35
<b>第 4 章 面试 .....</b>	<b>39</b>
4.1 面试流程 .....	40
4.2 编程面试 .....	42
4.3 注意事项 .....	43
<b>第 5 章 聘书与职业发展 .....</b>	<b>47</b>
5.1 聘书 .....	48
5.1.1 聘书要素 .....	48
5.1.2 决策因子 .....	49
5.1.3 薪酬谈判 .....	52
5.1.4 接受、延期或婉拒 .....	54
5.2 职业发展 .....	55

## 第二部分 数据结构

<b>第 6 章 数组 .....</b>	<b>59</b>
面试题 1：两数之和 I ☆☆ .....	59
面试题 2：两数之和 II ☆☆☆ .....	61



面试题 3：两数之和 III ☆☆☆☆	62
面试题 4：数组旋转 ☆☆☆	64
面试题 5：最大下标距离 ☆☆☆☆	65
面试题 6：重叠区间个数 ☆☆	67
面试题 7：插入区间 ☆☆☆	69
面试题 8：合并区间 ☆☆☆☆	71
面试题 9：数组配对 ☆☆☆	72
面试题 10：数位重组 ☆☆☆	73
面试题 11：产生随机数 ☆☆	75
面试题 12：Top K I ☆☆☆	76
面试题 13：Top K II ☆☆☆☆	79
面试题 14：两数组第 k 个值 ☆☆☆☆☆	80
面试题 15：两数组中值 ☆☆☆☆☆	82
面试题 16：旋转数组最小值 ☆☆☆	84
面试题 17：旋转数组搜索 ☆☆☆	85
面试题 18：首个正数 ☆☆☆☆	86
面试题 19：合并有序数组 ☆☆	88
面试题 20：三角形 ☆☆	89
面试题 21：二维数组搜索 ☆☆☆	90
面试题 22：区间搜索 ☆☆☆☆	92
面试题 23：插入位置 ☆☆	94
面试题 24：矩阵清零 ☆☆☆	95
面试题 25：螺旋矩阵 ☆☆☆☆	98
第 7 章 链表	101
面试题 26：合并链表 ☆☆	102

面试题 27: 环的长度 ☆☆☆	103
面试题 28: 反转链表 ☆☆	105
面试题 29: 分组反转链表 ☆☆☆☆	109
面试题 30: 两数相加 ☆☆☆	110
面试题 31: 链表分区 ☆☆☆	112
面试题 32: 链表去重 ☆	114
<b>第 8 章 树</b>	<b>117</b>
面试题 33: 二叉搜索树转为双向链表 ☆☆☆☆	118
面试题 34: 最小公共祖先 I ☆☆	120
面试题 35: 最小公共祖先 II ☆☆☆	121
面试题 36: 最小公共祖先 III ☆☆☆☆	124
面试题 37: 最小公共祖先 IV ☆☆☆☆	125
面试题 38: 路径和 I ☆☆	128
面试题 39: 路径和 II ☆☆☆☆	129
面试题 40: 平衡二叉树 ☆☆☆☆	131
面试题 41: 树的镜像 ☆☆	132
面试题 42: 中序下个节点 ☆☆☆	134
面试题 43: 二叉搜索树近值 ☆☆☆	135
面试题 44: 二叉搜索树 KNN ☆☆☆☆	136
面试题 45: 实现二叉搜索树迭代器 ☆☆☆☆	138
面试题 46: 充实横向指针 ☆☆☆	140
面试题 47: 恢复二叉搜索树 ☆☆☆☆	142
面试题 48: 按层遍历二叉树 ☆☆☆	144
面试题 49: 二叉树最大路径和 ☆☆☆☆	145

第 9 章 字符串 .....	148
面试题 50: 字符判重 ☆☆☆ .....	148
面试题 51: 产生括号 ☆☆☆☆ .....	150
面试题 52: 提取单词 I ☆☆☆☆ .....	151
面试题 53: 提取单词 II ☆☆☆☆ .....	153
面试题 54: 字符交替 ☆☆☆ .....	154
面试题 55: 字符串相乘 ☆☆☆☆ .....	155
面试题 56: 数字验证 ☆☆☆ .....	157
面试题 57: 字符串转为十进制数 ☆☆ .....	160
面试题 58: 提取 IP 地址 ☆☆☆ .....	161
面试题 59: 正则匹配 ☆☆☆☆☆ .....	163

### 第三部分 算法

第 10 章 俩指针 .....	167
面试题 60: 有序数组去重 ☆ .....	167
面试题 61: 三数之和 ☆☆☆ .....	169
面试题 62: 股票买卖 ☆☆ .....	171
面试题 63: 三色排序 ☆☆☆☆ .....	172
面试题 64: 蛙跳 ☆☆☆ .....	174
面试题 65: 容器盛水 I ☆☆☆ .....	176
面试题 66: 容器盛水 II ☆☆☆☆ .....	177
面试题 67: 数组分水岭 ☆☆☆ .....	179
第 11 章 动态规划 .....	181
面试题 68: 最长递增子序列 ☆☆☆☆ .....	182

面试题 69: 最小化数组乘积 ☆☆☆☆	183
面试题 70: 股票买卖 II ☆☆☆☆	185
面试题 71: 数组最大和 ☆☆☆	186
面试题 72: 二维数组最小路径和 ☆☆☆	187
面试题 73: 三角形最小路径 ☆☆☆	188
面试题 74: 爬楼梯 ☆☆	189
面试题 75: 迷宫路径数 ☆☆	190
面试题 76: 刷房子 ☆☆☆	192
面试题 77: 数字解码 ☆☆☆	193
面试题 78: 子串个数 ☆☆☆☆	194
面试题 79: 编辑距离 ☆☆☆☆	196
面试题 80: 交替字符串 ☆☆☆☆☆	197
面试题 81: 最长回文子串 ☆☆☆☆☆	198
面试题 82: 回文分割 ☆☆☆☆	199
面试题 83: 最大公共子串 ☆☆☆☆	201
面试题 84: 字符串洗牌 ☆☆☆☆☆	202
<b>第 12 章 优先遍历</b>	<b>205</b>
面试题 85: 填充图像 ☆☆☆☆	205
面试题 86: 封闭区间个数 ☆☆☆☆	206
面试题 87: 填充封闭区间 ☆☆☆☆☆	208
面试题 88: 单词查找 ☆☆☆	210
面试题 89: 单词变换 ☆☆☆☆	211
面试题 90: 单词替换规则 ☆☆☆☆	213
面试题 91: 有向图遍历 ☆☆☆☆	215

第 13 章 哈希 .....	217
面试题 92: 最长连续序列 ☆☆☆☆ .....	217
面试题 93: 变位词 ☆☆☆ .....	218
面试题 94: 最长不同字符的子串 ☆☆☆☆ .....	220
面试题 95: 最小字符窗口 ☆☆☆☆ .....	221
面试题 96: 单词拼接 ☆☆☆☆☆ .....	223
面试题 97: 常数时间插入删除查找 ☆☆☆ .....	224
面试题 98: 对数时间范围查询 ☆☆☆☆ .....	225
面试题 99: 实现 LRU 缓存 ☆☆☆☆ .....	226
面试题 100: 经过最多点的直线 ☆☆☆ .....	229
第 14 章 堆栈 .....	232
面试题 101: 局部最大值 ☆☆☆ .....	232
面试题 102: 数据流最大值 ☆☆☆☆ .....	234
面试题 103: 最大四方形 ☆☆☆☆☆ .....	235
面试题 104: 合并多个有序链表 ☆☆☆☆ .....	239
面试题 105: 产生逆波兰式 ☆☆☆ .....	240
面试题 106: 逆波兰式计算 ☆☆☆ .....	241
面试题 107: 简化文件路径 ☆☆☆ .....	243
面试题 108: 括号验证 ☆☆ .....	244
面试题 109: 最长有效括号 ☆☆☆ .....	245
面试题 110: 设计 Min 栈 ☆☆☆☆ .....	247
面试题 111: 中序遍历 ☆☆☆ .....	248
面试题 112: 打印路径 ☆☆☆☆ .....	249
面试题 113: 二叉搜索树两点之和 ☆☆☆☆ .....	251
面试题 114: 矩阵 Top K ☆☆☆☆ .....	253

第 15 章 排列组合 .....	256
面试题 115: 翻译手机号码 ☆☆☆ .....	256
面试题 116: 数组签名 ☆☆☆☆ .....	258
面试题 117: 组合和 ☆☆☆ .....	259
面试题 118: 子集合 ☆☆☆ .....	262
面试题 119: 全排列 ☆☆☆ .....	264
面试题 120: 下一个排列 ☆☆☆☆☆ .....	266
面试题 121: N 皇后 ☆☆☆☆ .....	268

## 第四部分 综合面试题

第 16 章 数学 .....	273
面试题 122: Fibonacci 数 ☆ .....	273
面试题 123: 求幂 ☆☆☆ .....	274
面试题 124: 求开方 ☆☆☆☆ .....	275
面试题 125: 随机数产生器 ☆☆☆☆☆ .....	276
面试题 126: 找出明星 ☆☆☆ .....	277
面试题 127: 聚合数 ☆☆☆ .....	278
面试题 128: 根据概率分布产生随机数 ☆☆☆☆ .....	279
面试题 129: 随机采样 ☆☆☆ .....	280
面试题 130: 数组元素乘积 ☆☆☆ .....	281
面试题 131: 访问计数 ☆☆☆ .....	282
第 17 章 位操作 .....	283
面试题 132: isPowerOf2() ☆☆ .....	283
面试题 133: isPowerOf4() ☆☆☆☆ .....	284

面试题 134: 两数相除 ☆☆☆☆	284
面试题 135: 不用加减乘除做加法 ☆☆☆	285
面试题 136: 实现 BitSet 类 ☆☆☆	286
面试题 137: 爬楼梯 II ☆☆☆	287
面试题 138: 只出现一次的数字 ☆☆	288
第 18 章 面向对象	289
面试题 139: 实现迭代器 peek() ☆☆☆	289
面试题 140: 实现复杂的迭代器 ☆☆☆☆	290
面试题 141: 实现 BlockingQueue ☆☆☆	292
面试题 142: Java 字节码编入 ☆☆	293
面试题 143: 依赖注入 ☆☆	294
第 19 章 杂项	295
面试题 144: 垃圾回收机制 ☆☆☆	295
面试题 145: 程序崩溃 ☆☆☆☆	296
面试题 146: 实现任意读 ☆☆☆☆	297
面试题 147: 实现读一行 ☆☆☆	298
面试题 148: 统计电话号码个数 ☆☆☆	299
面试题 149: 海量数据高频词 ☆☆☆	300
面试题 150: 多台机器的中值 ☆☆☆☆	300
附录 A 数据结构与算法	302
附录 B 海量数据结构	303



---

## 第一部分 硅谷求职

---



# 第 1 章

## 硅谷公司

硅谷是世界 IT 中心，汇聚了顶级的电子、芯片、软件、互联网等 IT 公司。本章将介绍硅谷的地理位置和发展历程，简述五家具有代表性的硅谷公司的文化和特点，最后描述国内的软件工程师寻找美国工作的三种途径及技术移民美国的方法。

### 1.1 硅谷简介

硅谷（Silicon Valley）指美国加利福尼亚州的旧金山以南，沿着 101 公路从门罗公园、帕拉托、山景城、太阳谷到硅谷的中心圣克拉拉，再经坎贝尔直达圣何赛的这条狭长地带。硅谷这个词最早见于 1971 年 1 月 11 日《每周商业》报纸关于电子新闻的标题。之所以名字当中有一个“硅”字，是因为当地企业多数是从事加工制造高浓度硅的半导体行业和电脑工业，而“谷”字则是从圣克拉拉谷中得到的。

硅谷气候宜人，四季如春。庞大的信息产业结合完美的气候和优美的环境聚集了世

界顶级的高科技人才，包括数以万计的中国优秀科技人才。有人说，北京和硅谷的区别并不在于天空的颜色，而在于一边的年轻人大多谈论的是买房、买车，被催着结婚生子，可另一边的年轻人谈论的则是改变世界的梦想以及如何让自己一辈子没白活。你能想象十几年前睡在硅谷大学大道 165 号车库地板上那几个 20 岁出头的小伙子吗？他们后来分别创办了 Youtube、LinkedIn 和 Yelp。

上千家高科技公司的总部设在硅谷（见图 1），近 40 家被福布斯收录在世界 500 强排名中。硅谷暴富的故事吸引一批又一批年轻人来硅谷创业。成功的创业公司总是具有这样的传奇色彩：不用考虑你的社会阶层、教育、种族、国籍或其他的各种因素，只要你是人才，便能出人头地。这就是硅谷的核心精神：只要有智慧、野心和伟大的想法，任何人都可以筹集资金开公司。

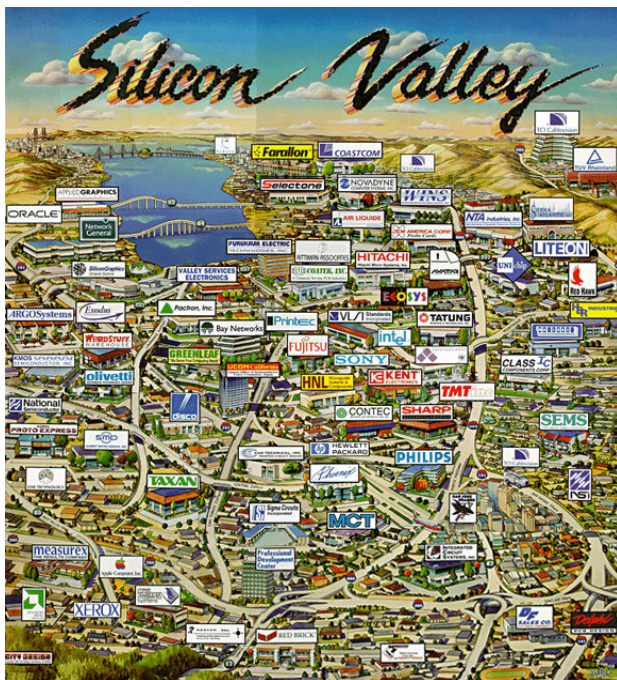


图 1 硅谷公司地图

从 20 世纪 60 年代起，硅谷这块狩猎的沼泽地飞速发展，在短短几十年后成为高科技工业中心，创造了巨大的物质财富，取得了引以为傲的非凡成就。当硅谷取得巨大成功之后，不少人试图在美国其他地区复制硅谷，但均以失败告终。硅谷之所以成为世界 IT 中心，是因为其特定的时代背景。硅谷取得成功的原因很多，而以下几个因素对硅谷的发展至关重要。

一、**以技术推动行业发展的时代背景**。这种时代发展主要表现在三波技术飞跃上，而这三波均起源于硅谷。第一波，晶体管的发明带动了半导体行业快速发展。1956 年，晶体管的发明人威廉·肖克利 (William Shockley) 在斯坦福大学南边的山景城创立了“肖克利半导体实验室”。来自该半导体实验室的八位工程师创办了仙童 (Fairchild) 半导体公司，他们被称为“八叛逆”。“八叛逆”里的诺伊斯和摩尔后来创办了英特尔 (Intel) 公司。在仙童工作过的人中，斯波克后来成为国民半导体公司的 CEO，另一位桑德斯则创办了 AMD 公司。第二波，软件产业兴起。除了半导体工业，硅谷同样以软件产业著称。施乐公司在 Palo Alto 的研究中心在 OOP (面向对象的编程)、GUI (图形界面)、以太网和激光打印机等领域都有开创性的贡献。现今的许多著名企业都得益于施乐公司的研究，例如苹果和微软先后将 GUI 用于各自的操作系统。第三波，20 世纪 90 年代末互联网服务兴起以及新世纪的移动互联网跳跃式的发展，带动了全世界互联网发展，加快信息流通，方便世界各地的人们交流。

二、**斯坦福大学以及斯坦福工业园**。二战结束后，美国返回大学的学生骤增。为满足财务需求，同时给毕业生提供就业机会，斯坦福大学开辟工业园，允许高新技术公司租用此地作为办公区。最早入驻的公司是 1930 年由斯坦福毕业生创办的瓦里安公司 (Varian Associates)。工业园同时为民用技术的初创企业提供风险资本。惠普公司是最成功的例子之一。在 20 世纪 90 年代中期，斯坦福工业园逐渐成为技术中心。作为世界十大名校之一的斯坦福大学，源源不断地给 IT 公司输送人才，同时也培养了大量的创业家。

三、**活跃和宽容的社会环境**。硅谷的许多人来自美国东部和西欧，他们为摆脱墨守成规的文化和官僚主义的束缚，被加州的特殊机会吸引而来。正如一位风险投资家所说：“东部乃是大公司的地盘，壁垒森严，个人很难立足其间。而加州则是前线，从经济、社会和组织形式来看都没有固定的模式，而且最重要的是它真正重视个人的价值”。近年来，来自世界各地的不少年轻人在硅谷圆了自己的创业梦。尽管失败者占多数，但整个社会对创业失败者有着比其他地方更多的包容。

四、**良好的自然条件以及完善的基础设施**。正如我们前面谈到的，整个旧金山湾有优越的地理条件，阳光明媚、气候舒适，有“天然空调”之称。北加州原有的高质量生活，吸引人们来到此地生活工作。便利的交通、快捷的通信、世界一流的大学，斯坦福大学、加州大学伯克利分校等提供了有力的科技后援——优良、丰富又具流动性的高科技人才。这一切营造出了良好的科技和商业环境。

五、**发达的资本市场**。美国高科技企业的创业与风险投资的关系很大。苹果电脑公司 1976 年创办时，投资企业家马克库拉投资 9 万美元，借贷 25 万美元，占 30% 股份，从而推动了苹果电脑的发展，进而使革命性的个人电脑成为新兴产业。1982 年 Adobe 公司创建时得到了著名风险投资公司 H&Q 的支持，后者获得了百倍的利润回报；Adobe 公司在成长壮大后，又与 H&Q 合资成立了新的风险投资公司，支持高科技企业的建立，得到了丰厚的回报。上述两家公司的建立都起源于独创性的科研成果，而风险投资使成果能及时转化、占领市场，并分别形成个人电脑和电子化出版业这两个新兴产业。没有风险投资的参与，仅靠一人或数人的有限财力，很难使公司快速发展。在美国有许多专业的、高素质的投资家，他们有着丰富的投资经验，一项独创性的技术很容易吸引到大量的资金投入，所以在美国经常可以听到高科技人员一夜之间成为亿万富翁的消息。

## 1.2 传奇旗帜

**Myth flag**，译为“**传奇旗帜**”，分别是八家公司的首字母组合：**M**icrosoft（微软）、**Y**ahoo（雅虎）、**T**witter（推特）、**H**ortonworks（一家新兴的云计算公司）、**F**acebook（脸书）、**L**inkedIn（最大职业社交网站）、**A**mazo**n**（亚马逊）和 **G**oo**g**le（谷歌）。这八家公司是硅谷的软件和互联网公司的典型（从严格意义上来说，亚马逊和微软不属于硅谷公司，虽然它们在硅谷有分部，但总部在西雅图）。其中，微软代表了传统软件公司，此外还有甲骨文、Teradata、Adobe 和苹果公司等；雅虎和谷歌是老牌互联网公司的代表，而 Facebook 和 Twitter 则属于新兴互联网公司，发迹于 Web 2.0 兴起。而最具有活力和爆发力的公司是新一代云计算和移动互联网公司，它们中大部分还在创业阶段，还未上市，Hortonworks 就是其代表之一。

不同的公司有不同的企业文化。评判一家公司企业的标准有很多，其中之一就是组织结构图。图 2 是一张来自网络有趣的组织结构图，里面包括了亚马逊、谷歌、Facebook、微软、苹果、甲骨文这六家公司的组织结构图。从图中我们可以看出亚马逊有着严格的等级制度；谷歌也有清晰的等级，但是部门之间相互交错；Facebook 就像是一张分布式对等网络；微软则是各自“占山为王”，拥有浓厚的办公室政治文化，类似国企；苹果则是一个人说了算，属于个人独裁，围绕着乔布斯或新 CEO 库克转；最具讽刺意义的是最后的甲骨文，法务部门远远大于工程部门，随时准备向竞争对手提起诉讼，以及应诉。

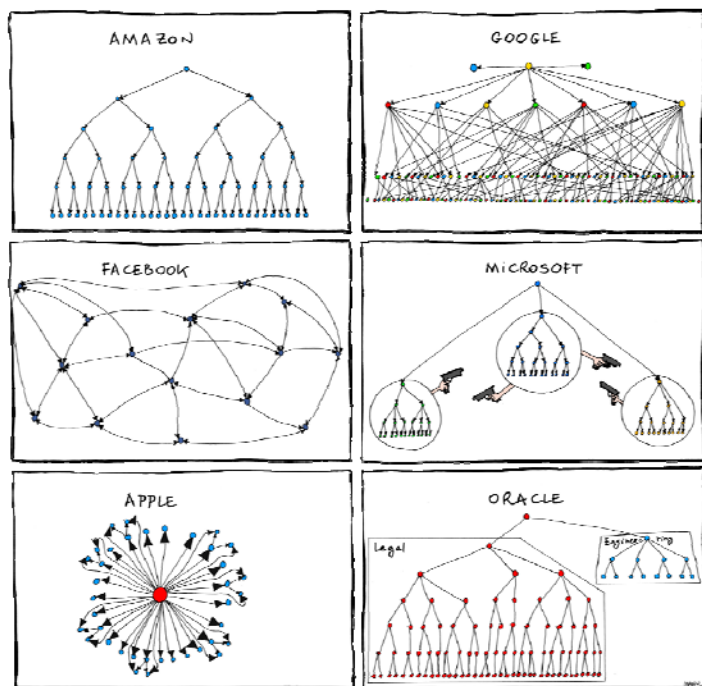


图2 六家公司架构与文化

接下来，我们从上千家软件和互联网公司中挑出五家（这五家拥有大量来自中国的员工），重点从工程师角度阐述公司以及公司的文化。

### 1.2.1 微软

微软在 PC 行业一直是霸主，特别是在新旧世纪交替时，Windows 操作系统正处于顶峰，它在 PC 操作系统市场的份额超过 90%。然而，在互联网和移动互联网快速发展的十几年里，微软未能成功转型，或者说未能成功在互联网和移动互联网开拓市场，被互联网公司谷歌等抢去了风头，还被华尔街看衰。这一点从股票上也能体现出来：在 2000 年 1 月 1 日鲍尔默接替比尔·盖茨成为微软 CEO 时，微软股价为 58.719 美元，而市值



则为 6163 亿美元；然而，笔者在写本书时，微软股价为 33.27 美元，市值为 2771.4 亿美元，较当时下降 55%。迫于多方压力，今年鲍尔默宣布一年后将辞任 CEO 职位，并开始物色新 CEO 人选。

尽管如此，微软仍然是一家非常赚钱的公司。2008 年微软很好地应对了全球金融危机，强劲的营收和多样化的产品使微软能够在金融危机期间，在信贷极度紧缩的情况下获得足够多的现金。微软仍然保留着强大的技术和产品研发能力。令人惊讶的是，微软实际上是首家开发平板电脑的公司。2002 年时，微软曾开发了一款未获得成功的平板电脑 Tablet PC。此外，微软的 Pocket PC 产品线同样也是最早的智能手机之一，微软早在 2000 年就推出了这一产品。但是由于微软试图在手机上复制与 Windows PC 类似的体验，因此未能像 iPhone 一样对手机体验进行重大变革，从而未能取得成功。

微软为员工提供着相当不错的薪水及福利，总部拥有近 2000 名来自中国的员工。微软职位最高的华裔是毕业于复旦大学的陆奇，他负责 Office 软件和必应在线搜索引擎，在他的帮助下，必应培养了一大批中国籍的管理层。相比谷歌和 Facebook，微软录用门槛较低，是值得一试的公司。但有一点需要注意的是：微软每年采用末位淘汰的制度，在工作表现（performance）上得低分的员工将被迫辞职。因此，除了做好自己的工作以外，还需要和上司保持良好的关系。

### 1.2.2 谷歌

谷歌是一家很酷并且富有创造力的公司，除了称霸搜索领域，它还拥有一个神秘实验室——Google X。该实验室可以脱离其核心业务而进行自由的产品研发。从无人驾驶汽车到谷歌气球，再到谷歌眼镜，一项项酷味十足的创造都是 Google X 实验室的杰作。谷歌技术驱动型的文化的确有助于员工尽展所能。除了 Google X 实验室外，谷歌还形成了著名的“20%自由时间”规则，即允许员工把工作日 20%的时间用于与本职工作无关的项目。谷歌本身就像一个创新引擎，通过创始人奖金、同事评优及 20%的自由时间

等，驱动员工不断创新。App Engine 论坛、Google X 实验室及其他的开源项目等扮演着创新的练兵场。正是从这架创新机器里诞生了诸如 Gmail、Chrome 等具有战略意义的重要产品，而它们也都来自底层员工的创新。

据最新报道，“20%自由时间”规则已名存实亡。一个从谷歌离职的高官抱怨现在的谷歌是一家专注于广告的公司，而不是原来的不断驱动员工创新的技术公司。他认为昔日的谷歌靠内容赚钱，就像电视台一样，通过制作最好的电视剧，吸引最大的广告收入，但是今天的谷歌看起来似乎把更多的精力放在了广告本身上。

尽管如此，谷歌仍然连续多年被评为年度最佳雇主，其奢华的工作环境举世公认：舒适宽敞的员工工位、人性化的休息室、完备的游乐场、常年运转的健身房、瑜伽室、按摩室、沐浴室，等等。大楼每一层一般都会设置一个巨大的休息室，里面提供按摩椅、桌上足球、台球、游戏机。另外，用心排列的零食陈列架囊括了所有主流零食，上面摆满了可乐、雪碧、苹果汁、橙汁等时下最火的饮料，冰箱里还有各式各样的冰激凌、酸奶等。员工按摩可以随时预约，而餐饮常常是五星级酒店大厨的手艺。最关键的是**所有这一切都是免费的**。最近，谷歌又提供了一项新福利：如果员工不幸去世，其配偶还能在未来的10年里享受到去世员工的半数薪酬；他们的未成年子女还能每月收到1000美元的生活费直至子女到19岁成年。除了半数薪酬，配偶还能获得去世员工的股权授予。

但是被谷歌录用除了需要拥有过硬的技术实力以外，还需要运气，因为这里的每个岗位都竞争太激烈了。

### 1.2.3 亚马逊

亚马逊看重的是创意人才，它并不只是一家电子商务公司。亚马逊员工背景多样，有些是自由派艺术家、大学学者、摇滚音乐家，还有些是职业滑冰选手、赛车选手等。亚马逊的“未来利润分享制”，把所有员工包括仓库员工、公司职员及行政经理、最高主管，全部纳入到公司的认股专项计划中。

亚马逊上市十几年以来，一直处于不盈利或微盈利状态，其根本原因是零售的利润太低。这也导致了亚马逊工程师的福利在顶级 IT 公司中较为一般。我们先谈一谈亚马逊对于工程师来说的优势。

- **与时俱进的技术。**在软件开发环境、开发框架、网络监控、软件部署等技术上，强于传统的软件公司。因为全公司的软件开发都基于网络，并采用大量的开源技术，所以对于新入职的员工来说，容易上手，对于老员工来说，容易跳槽。
- **良好的业内名声。**业内都认为亚马逊的工程师技术过硬，当他们去面试谷歌、Facebook 和 Twitter 等福利更好的公司时，均有面试的机会。
- **开放的公司文化。**对于工程师来说，在亚马逊没有其他大公司的办公室政治，工程师拥有自主权，有活做，但活不重；内部转组相对容易，而且没有微软的末位淘汰机制，公司基本上不裁员。

当然也有不好的地方。

- **“臭名昭著”的 on-call 制度。**on-call 意味着你需要一天 24 小时随时待命，准备解决可能出现的问题。因为亚马逊的零售及云服务都是 7×24 运转，而且在中国、日本、欧洲的零售分部也由美国支持，所以几乎所有的开发组都有 on-call 制度。让研发工程师处于随时待命的状态，部分原因是亚马逊没有一支完善的运维团队。仓库一旦出现问题，那边工人会直接通过 on-call 跟研发工程师联系，这样效率非常高，而且节省了运维的人力成本，但是给研发工程师们带来了巨大压力，工程师们任何时候都有被呼叫的可能。假设你在半夜 2 点被叫醒，一直处理到 6 点才修复问题，并且第二天你还得正常上班，无论怎么想都是一件可怕的事。
- **不合理的绿卡制度。**对于新入职两年内的低级别的工程师，公司不给办绿卡（接下来一章会解释绿卡的重要性）。其他大公司，比如微软、谷歌和 Facebook 等，入职第一天公司即可协助员工办理绿卡。

- **工程师晋升缓慢。**特别是在传统电子商务部门,从 SDE I 到 SDE II 要等待 4 年。部分原因是管理混乱,管理人员变动过快,导致底层工程师的职称评定被拖后。
- **一般水平的员工福利。**除了基本工资,对比其他公司亚马逊有些福利没有,有些福利则较差,比如医疗保险比同城的微软就相差很多。

尽管亚马逊拥有让人诟病的 on-call 制度,但它还是很适合刚毕业的学生,或者是单身工程师。况且,如果你来自国内的互联网公司,比如百度、淘宝或腾讯,对于这种 on-call 制度你将表示毫无压力。

#### 1.2.4 T witter

Twitter 是一家媒体公司,同时还是一家科技公司。Twitter 的 CEO Dick 表示 Twitter 是一家媒体领域的技术公司,但它卖的不是技术,它的核心业务是广告。在最近的一项调查中, Twitter 给出的薪水在行业内排名最高,达到人均 11.5 万美元,同时它在赠与股票方面也很大方,高级工程师通常每年拿到的股票价格与薪水持平。

Twitter 的公司文化和谷歌类似,有各种标准的小福利,如免费午餐、晚餐,每人一台 MacBook Pro,弹性工作时间以及换团队和项目比较容易,还有令人满意的工作环境。公司用心对待老员工,不用虚空的荣誉头衔和奖励,而是选择把老员工们提升到有实权的岗位。Twitter 正在实现成为一个由工程技术人才驱动的企业。有媒体报道称,在 2011 年的人才竞争中, Twitter 每离开 1 人后,公司从其他公司雇佣近 11 人,新进人数是离开人数的 11 倍,这表明 Twitter 具有独特的吸引力。

#### 1.2.5 Epic

Epic 在 IT 行业并不出名,但它一直是医疗软件排行第一的软件公司。Epic 的名声不大,其中一个原因是它还没上市,另一个原因是它的总部在美国中部威斯康辛州的麦迪逊郊区。本书挑选 Epic 作为美国普通软件公司的典型,原因是它的地理位置较偏,

薪水较其他行业高，但福利一般，老板抠门。

Epic 在 Verona，麦迪逊的郊区，位置较为偏僻。附近餐馆少，员工很难找到外出吃饭的地方。据离职员工称，程序员的年终奖也少，股票基本上没有。公司代缴的医疗保险部分常常是从员工工资里扣除。

Epic 不是传统软件开发公司，而是一家提供软件服务的公司，这使得 Epic 不是很重视技术，而是把客户关系放在第一位。从而影响了 Epic 在技术上的创新。另外，由于 Epic 不是一家上市公司，老板有绝对的权威，他的喜好决定了 Epic 发展的走向。很多人把 Epic 作为职业发展中的一个跳板，锻炼几年后就跳槽到西部的纯软件开发公司了。

## 1.3 技术移民

对于国内程序员来说，通常有三种途径去美国工作。

**1. 进入跨国公司工作一年以上，然后内部转组到美国的部门。**例如，进入微软中国工作一年后，申请转组到美国总部工作，若能批准，则申请 L-1 签证去美国工作。

**2. 直接应聘美国公司。**这其实没有想象的那么难。当前不少美国 IT 公司直接从国内招人，比如谷歌、Facebook 等。这些企业看中的是国内程序员扎实的计算机基础以及丰富的项目经验。拿到美国公司的聘书（offer）之后，申请的是 H1B 签证去美国工作。不过，按照目前 H1B 申请与批准情况来看，H1B 名额很快就用完了；每年 4 月 1 日开始提交申请，而要到 10 月 1 日才能合法工作，这要看公司能否等你半年。当然，很多大企业愿意等。如果你拿不到名额，有些公司可以安排你先去别的国家工作，等到下一年申请到名额了，再派你去美国。

**3. 去美国留学，然后找工作。**先持 F1 学生签证去美国读书，毕业后，找到了工作，转为 H1B 签证。笔者就是走这条路径：在中国科技大学读计算机硕士期间，考 GRE 和托福并申请美国的学校，毕业后拿到全额奖学金去北卡罗来纳州立大学攻读计算机科学博士；四年后，毕业去微软工作。这也是很多在美工作的中国程序员走的路。其实，如果你对研究不感兴趣的话，完全可以申请硕士，而不是博士，也不用全额奖学金。留学已经不难了，只要你有一定的经济基础。但这种途径的坏处是花费时间长，代价也很大。

在美国工作的中国籍程序员经历的过程大致相同：找到一份工作，申请工作签证；在工作过程中，申请绿卡；等绿卡批准之后，再过 5 年才能申请美国国籍。在这个章节，我们将介绍美国的工作签证、绿卡和生活情况。

### 1.3.1 签证和绿卡

如果你不是美国人，也不是美国绿卡持有者（即永久居民），你进入美国工作之前得申请签证。如果申请人希望以非移民身份在美国短期工作，根据美国移民法的规定，应根据工作类型申请相应签证。大部分短期工作签证都要求准备雇佣申请人的美国雇主提交申请批件，并获得美国公民及移民事务处（USCIS）的批准，然后才能申请工作签证。

工作签证主要有三种：H1B、L-1 和 O1。O1 签证颁发给各行业的杰出人才，一般工程师拿不到。在这里，我们主要介绍 H1B 和 L-1。

**H1B（专业人士）：**如果申请人希望去美国从事预先安排的专业技术工作，则应申请 H1B 签证。此类签证要求申请人在准备就职的专业领域拥有学士以上学位（或同等学力）。USCIS 会审查决定申请人被雇用后将从事的工作是否构成专业技能工作，申请人是否符合该专业技能的要求。申请人的雇主应向美国劳工部提交劳工情况申请表，包括雇佣双方之间签订的合同条款。H1B 申请人的家属可以申请 H-4 签证。如果你是持有 H1B 签证的主申请人，则你的配偶或未婚子女（21 岁以下）将获发 H-4 签证以便陪同

赴美，但是，家属不得在美国工作。

**L-1（公司内部调职者）：**如果你是一家跨国公司的雇员，被临时派往美国的上级机构、子公司或附属单位工作，则需申请 L-1 签证。此处所述跨国公司可以是一家美国或外国企业。要获得 L-1 签证，申请人必须属于管理层、高管层或是拥有专业知识的人员，并且被公司派往美国担任此类职务，但赴美前后的职位头衔不必完全相同。此外，在申请赴美工作的前三年中，申请人须在该国际公司的美国境外机构连续工作满一年。只有在申请人准备就职的美国公司或附属机构获得 USCIS 的申请批件后，才能申请 L-1 签证。如果你是持有有效的 L 类签证的主申请人，则你的配偶或未婚子女（21 岁以下）将获发 L-2 签证。根据最近出台的法律规定，你的配偶也有机会申请就职许可。

由于工作签证有种种限制，比如 H1B 签证年限为 6 年，6 年后你必须离境一年以上才能再进入美国，L-1 只能为一家公司工作，换了公司就得重新申请，等等，一般中国程序员都会在工作过程中申请绿卡。绿卡持有者除了出入境方便以外，还容易换工作，可选雇主范围也会更广，因为有海量的小公司和与政府项目相关的外包公司不愿意或者不方便为外国人申请工作签证。

美国移民法将职业绿卡申请分为五大类，分别为第一优先（EB1，杰出研究人员和学者移民），第二优先（EB2，高等技术移民），第三优先（EB3，普通技术移民），第四优先（EB4，特殊类移民），和第五优先（EB5，投资移民）。劳工移民的申请类别不同，拿到绿卡的时间也会不同。劳工移民第三优先（EB3）是最慢的一种。一般来说，程序员会申请 EB2 或 EB3，特别突出的人才可以尝试一下 EB1。目前，EB2 从申请到批准的时间大概是 4~6 年，EB3 需要 5~8 年，而 EB1 等待时间最短，基本上一年之内获批。由于这几年 IT 行业好转，大量中国和印度的程序员涌进美国，绿卡排期也越来越长。有人说，申请美国绿卡是一条看不见尽头的漫漫长路，犹如坐监，俗称“移民监”。

美国总统奥巴马对媒体称 2014 年他的工作重心是移民改革，他将积极推动新移民法案 CIR 顺利通过国会两院，并最终使之成为法律。新移民改革方案主要倾向于给滞留美国的非法移民绿卡，同时开放高科技移民，以保持美国的全球竞争力。对于高科技职业移民来说，绿卡申请的等待时间大大缩短，同时每年工作签证 H1B 配额翻倍或者 3 倍以上。最直接的影响是大量高科技人员将移民美国，包含中国的顶级程序员。

除了职业移民，还有亲属移民。美国移民法允许与美国公民或美国永久居民有亲属关系的外国公民移民到美国，这就是美国亲属移民。美国亲属移民分为两种：无配额限制亲属移民和配额限制亲属移民。亲属移民视各类别的不同，从申请到签证获批，等候的时间从六个月到十二年不等。无配额限制亲属移民等待时间较短，只有美国公民的直系亲属才能申请。他们包括美国公民的配偶、美国公民的 21 岁以下未婚子女、美国公民的配偶同其前夫或前妻所生的 18 岁以下未婚子女，以及美国公民合法领养的 16 岁以下未婚子女和 21 岁以上美国公民的父母。

### 1.3.2 税率和生活

在美国有句谚语：“世上只有两件事你逃不过：一件是死亡，另一件就是缴税。”美国税种繁多，税率复杂，也有类似国内“起征点”的宽免额。但和中国不同的是，这个“起征点”是不固定的，一般来说，大概是年收入在 7500 美元以下的免征。不过在美国有正常收入的人，都会超过这个数字，所以才会说“逃不过缴税”——几乎每个有收入的人都必须缴税。为保证中低收入家庭和个人的生活水平不会因纳税过多而下降，美国税法还规定了各种扣除、免税收入和退税制度。

这里简单列举一下几个主要的税种。

**1. Federal Income Tax (联邦税)：**由联邦政府征收，同时也是美国联邦政府收入的最大来源。联邦税有两个特点。第一，计算方式是阶梯式的，收入越高边际税率越高，从 15%到 38%不等；第二，家庭结构（比如有无配偶、有无子女）极大影响最终税率，



单身的赋税最重。

**2. State Income Tax (州税):** 由所在地的州政府征收。有些州没有州税，比如西雅图所在的华盛顿州。大部分州均有州税，税率从 7% 到 11% 不等。

**3. Sale Tax (消费税):** 在你购买商品时，商家代收的税，不会从你的薪水中扣除。商店展示的商品价格都是税前的价格。绝大部分州都有消费税，税率和州税类似。个别州没有消费税，比如俄勒冈州。因为华盛顿州有 9.5% 的消费税，所以生活在西雅图的微软员工在购买大件的时候，常常驱车三个小时去没有州税的波特兰购买。

**4. Social Security Tax (社保基金):** 美国政府用于养活贫困线以下的家庭、残疾人 and 无退休金的老人等。社保基金按照固定税率征收，税率为 6.2%，征税基数是薪资扣除商业医保剩下的部分。它的征收额有上限，目前最高征收税基是年收入 10 万美元。

**5. Medicare Tax (社会医疗保险):** 主要用于医院支付无医保人员的费用。这个税也按固定税率征收，税率为 1.5%，但没有上限。

此外，还有商业保险税和房产税。商业保险税一般由雇主代付，不会从你的薪水中扣除。房产税税率每个州都不一样，大概在 1% 左右。报税的方式和国内不同，在美国是以家庭为单位报税，而在国内是以个人为单位报税，而且国内没有退税制度。在美国，你会看到很多已婚妇女做家庭主妇，一方面是因为以家庭为报税单位，如果夫妻双方均工作，则赋税更重，她们税后的薪水常常只够请保姆的钱，还不如自己带小孩；另一方面，她们认为自己带小孩对小孩的教育更有利，而且她们的生活更自由。

总体来说，对同等收入人群，美国税率比中国税率高。假设同是年薪 60 万人民币，在国内税率大概是 28%，而在美国税率为 38%。虽然美国个税比中国高，但是同等水平的程序员在美国的生活质量却远超中国。这涉及收入与物价对比。大件商品，比如汽车、化妆品、名牌衣服等，在美国的售价即便换成人民币也比国内便宜。硅谷房价一直很高，

在美国也能排上前几名，我们以北京和硅谷的房价为例，北京的百度程序员，年薪 20 万人民币，不吃不喝买一套普通住房需要 15 年以上。而在硅谷，刚毕业的硕士生能拿到 9 万美元年薪，不吃不喝买一个普通独栋别墅需要 10 年左右。是的，你没看错——是独栋别墅！

# 第 2 章

## 求职准备

通常来说，寻求美国程序员的职位必须做好前期准备，比如，你打算做什么工作？开发、产品设计还是测试？打算应聘哪些类型的公司？大公司还是创业型公司？需要花多长时间做好技术面试的准备？从哪里获取更多的信息？如何准备简历？每家公司的面试风格是什么样子？等等。

首先，我们看看 2013 年拿到 Facebook 聘书的国内程序员贴在网络上的求职历程。

1 月 12 日：通过朋友推荐，收到 Facebook 猎头的邮件

1 月 26 日：预约了第一次电话面试的时间

1 月 29 日：一个老美问了一个简单的问题和一个 DFS 算法题，很快搞定

1 月 30 日：收到邮件说通过了，约了第二次电话面试的时间

1 月 31 日：第二个电话面试的面试官是一个中国人，也问了两个编程题目

2 月 5 日：收到反馈说让我去硅谷现场面试，可是在春节假期无法办签证

2 月 25 日：Facebook 发了邀请函，我赶紧去预约美国大使馆面签

2月27日：顺利通过美国签证面试，面试官居然没看我的邀请函

3月3日：说去美国现场面试后可能赶不上今年 H1B 签证的申请，所以改成 Skype 面试

3月5日：半夜，我进行了4轮编程面试

3月6日：一大早 Facebook 就发信来说收到两个面试官反馈，说对我评价很好

3月8日：我拿到 offer

3月11日：律师开始跟我联系，让我准备 H1B 材料

3月15日：我把所有的材料发给律师

3月23日：律师效率很高，LCA 被批准

3月28日：H1B petition 提交，据说要抽签

4月11日：律师给我发邮件，说 H1B 被批准

5月20日：收到 Fedex 快递，律师给我寄的申请材料和 I797B

5月23日：收到邮件说要对我进行背景调查

6月14日：背景调查通过

7月8日：故意挑女儿一岁生日时，一个 H1B 带两个 H4，去上海领事馆面签

9月4日：收到邮件让我去中信银行取护照，终于一切都定了。就等着公司提供搬家服务，32岁的我即将带着老婆孩子去美国开始新生活

从上面的例子，我们可以看出：1) 确定求职目标，即 Facebook 的大公司及软件开发职位；2) 通过已有的人际关系，找到内部推荐；3) 准备好技术面试；4) 有了聘书之后，赶在4月1日之前申请 H1B 名额；5) 有了 H1B 名额之后，即可去美国大使馆或领事馆面试签证。在这个章节，我们先介绍常见的硅谷职位、如何建立求职人际关系及获取求职信息的渠道。在接下来的几个章节，我们将陆续介绍如何撰写简历、如何准备面试及如何与招聘你的公司谈判，等等。

## 2.1 职位选择

谈起软件行业的职位，大家自然就会想到软件开发工程师，其实硅谷的职位远不止这个。只要我们在招聘网站搜索公司名字，就可以看到大量的不同性质的职位。与开发相关的有研发工程师、测试工程师、运维工程师、产品经理、前端工程师、数据科学家、架构师、基础框架工程师、用户体验工程师，等等。本书会重点介绍大量招聘外国人的职位，如研发工程师、测试工程师、产品经理和数据科学家。

### 研发工程师（SDE）

这是最普通的职位，也是需求量最大的职位。这需要你精通一门面向对象语言，比如 Java 或 C++，最好还能掌握一门脚本语言，比如 Python 或 Shell，加上若干个项目经验，此时你就能胜任初级的软件开发工程师的工作。一般来说，研发工程师更具有自由度和自主性，比如能够设计新产品特性、提交并修复 BUG 等，但晋升竞争激烈，因为每个团队都有大量的研发工程师，想脱颖而出并非易事。

### 测试工程师（QA）

在有些公司，比如微软，测试工程师又称为 SDET，即测试开发工程师。这是很多人会忽视的职位，也有很多人会误解，认为这种职位没什么技术含量。其实在一些公司里，测试工程师编写代码的量不会小于研发工程师。测试工程师与研发工程师相比，缺少自由度，通常他们的工作与项目捆绑在一起。但是，由于测试工程师进入门槛较低，并且晋升快，所以适合做事细腻的女性工程师。

### 产品经理（PM）

如果你的英文较好，而且在产品方面有敏锐的洞察力，那么产品经理也会是一种很

好的选择。产品经理是市场和开发之间的桥梁，通常需要把客户和用户需求转化为产品文档，并兼顾产品的营运。产品经理在上班时候不是在开会，就是在去开会的路上，因为他们需要和开发、测试、用户体验等团队协调，甚至和客户见面。产品经理并非都是科班出身，他们来自各行各业。在不少 IT 公司里，公司 CEO 就是最大的产品经理。

### 数据科学家 (Data Scientist)

伴随着大数据的出现，对这个职位的需求量也越来越大，目前成为硅谷最热门的职业之一，薪水相比研发工程师只多不少。数据科学家就是采用科学方法，运用数据挖掘工具寻找新的数据洞察的工程师，通常需要有大数据处理、机器学习和数据分析的经验。它的准入门槛较高，但如果你拥有计算机博士或者统计学博士学位，则可以试一试。

## 2.2 公司选择

你打算去大公司，中小公司，还是创业公司？这也是每次跳槽时大家都会去考虑的问题。在硅谷虽然有很多名头响亮的大公司，但更多的还是锐意进取的小公司，并且每天都在诞生新的创意公司、新的小公司。当技术、资金、理念和创新有大量机会摩擦碰撞时，当有这么一种机制帮助你不断实践梦想而不必承担失败的后果时，并且大量人才聚集在硅谷时，创业就变得自然而然了。

在大公司工作的好处大家都知道，待遇优厚、稳定性高、绿卡申请便捷，等等。而在创业公司 (startup) 工作的好处则是，可以更全面地接触系统内各个方面的技术，更直接地观察公司的运作，与同事的关系也更简单融洽。不管是大公司还是小公司，你**必须在面试之前和对方的人力资源部门确认 H1B 政策**，即公司是否支持外国人在美国合法工作（可以先把绿卡政策放在一边，等到你有多个聘书之后，再问也不迟）。

小公司更缺人才，他们更会不惜代价地引进一个人才。另外，小公司的面试难度也

相对较低，毕竟申请的人也较少。最后，如果小公司成功被收购甚至上市，那么你的股票收入很可能会远大于工资所得，而且你更容易进入公司高层。因此，不妨也试试小公司，至少可以作为一个跳板。一旦进入了这个圈子，工作机会也就慢慢铺开了，用不了多久猎头们就会盯上你，成天鼓动你跳槽。

所有的创业公司都值得去吗？一位斯坦福大学商学院的教授给刚毕业的学生的建议是，第一份工作应该选在一家势头不错的中型公司，因为这样的公司更有可能取得成功。他认为，一家成功的企业会让你获得更有价值的东西，即行业洞察力。

同时，他不推荐初始的创业公司。他认为，大部分的创业公司都会以失败告终，这意味着加入它们的风险与获得的回报将不成正比。这个理念对于投资来说也很重要，以最小的风险获得最大的回报。他通过与 10 到 15 家的顶级风投交流后，得出了一份将会快速发展并值得加盟的企业名单。他推荐这些科技企业时只考虑公司的成长性，他认为这比你获得的收入、职位头衔都要重要。这些企业的年收入在 2000 万到 3 亿美元之间，它们增长迅速并且在可预期的未来会保持这种增长势头。

以下是一部分被推荐的中型、快速增长公司的列表。

- Acronis: 对于保存在物理媒介、虚拟媒介及云端的数据进行保护和灾难恢复管理的公司。<http://www.acronis.com>
- Arista Networks: 大型数据中心与计算业云端网络解决方案提供商。  
<http://www.artistanetworks.com>
- Boku: 全球移动支付平台。<http://www.boku.com>
- Box: 云存储服务提供商。<https://www.box.com>
- Cloudera: 基于 Apache-Hadoop 的软件开发商。提供基于 Hadoop 及其他 Apache 平台的服务、培训及资格证明授权。<http://www.cloudera.com>
- Dropbox: 云存储服务提供商。<https://www.dropbox.com>
- Evernote: 笔记记录与整理应用开发商。<https://evernote.com>

- Palantir Technologies：专注于数据问题的软件提供商，又是一家大数据公司。  
<https://www.palantir.com>
- Quantcast Corp：网络评测分析公司。<https://www.quantcast.com>
- Spotify：音乐搜索，分享与即时播放应用开放商。<https://www.spotify.com>
- Square：信用卡移动支付服务。<https://squareup.com>
- Yousendit：企业协作服务提供商。<http://yousendit.com>

## 2.3 人际关系

内部推荐和个人引荐是找工作的最好的两种方式。不仅公司喜欢招推荐来的人，自己也会觉得这样可以找到适合自己技能和兴趣的工作，而且推荐人一般也能得到物质奖励。找推荐人需要在平时就注意建立人际关系网。

人际关系网不是到需要的时候才开始建立的。当要为新工作动用关系的时候，才和对方建立关系，则为时已晚。关系的建立是长久之事，敞开心扉并乐于助人可以让你快速地建立起庞大的关系网，而这个网也会随着你展现出的对于别人生活的价值而进一步加深。关系网是在你用不着它的时候建立的，和你建立关系网的人可能是你的校友、同事、朋友，也可能是专业的猎头。

平时在和同事、朋友相处的过程中，要保持真诚，并能够积极帮助他人。那些一直施惠于人而不考虑回报的人，到最后才会有一大群对其心生感激的人。在关系网的建立上，特别要注意不能过于关注对方的一点，以免显得过于功利。

随着社交网络的广泛使用，在找工作的过程中，也需要充分利用 Twitter、Facebook 和 LinkedIn 这些“社交工具”。但是，我们需要认识到的是它们的功用大不相同。Facebook 和 LinkedIn 可以帮助维护现有关系。一般来说，你不会和陌生人在 Facebook 上聊天。



即使聊，你也不会想着以此来影响自己的职业发展。Twitter 则不仅限于朋友圈的交流，同时也能大大促进交友圈的扩大。

下面就介绍一下如何有效利用这些社交渠道。

- **LinkedIn**，可以用于朋友或职场间的交流。先从你的校友和好朋友开始，把你认识或者不认识的在美国工作的校友都加入到你的关系网中。一位风险资本家就鼓励与自己合作过的企业家们“把每一个你见到的人都加为好友，而且要快，不能等”。用 LinkedIn 交流有一个好处，就是你可以为别人写推荐信，同时也可以请别人为自己写。搜索一下与自己兴趣相关的小组并参与到其中的讨论，那些负责招聘的人肯定喜欢在那里出没。LinkedIn 也是猎头出没的地方，有针对性地把大公司的猎头加入到你的关系网中。
- **Facebook**，由于在生活社交领域做得极为出色，所以很多人都忽略了它的职场功能。Facebook 是服务于社会的，这一点对于职场社交来说显然很有价值。当你需要咨询别人或向他人求助的时候，上 Facebook。通常你所要做的就是更改一下状态标签而已。
- **Twitter**，可以成为你与他人互相联系、开展对话的一种极为有效的工具。大多数没有利用好 Twitter 的人是因为自己没有花时间和上面。要是你觉得自己可以投入些精力在账户维护上，那么我建议你开一个 Twitter 账户，然后把自己职业方面的想法或者有趣的文章贴上去。如果可以持续性地发帖，你就可以开始寻求关注了。关注你感兴趣的与行业相关的人士，他们也会关注你。在你的 E-mail 签名栏、Facebook 和 LinkedIn 里都放上你的 Twitter 链接。找到你很想见面的那些人的 Twitter，针对他/她的 Tweet 发表自己的观点，以期和他们建立联系。在 Twitter、Facebook 和 LinkedIn 必须使用英文，一方面让别人更好地了解你，另一方面也可以锻炼你的英文写作能力。
- **校友**，引荐你的人也可能是你的校友，因此，你要尽可能从师兄师姐手里获取在美国工作的校友花名册。硅谷的华人每年会按照学校举行校友聚会，所以很容易

获取你所在学校的通讯录。你可以根据花名册上面的联系方式，分别加上他们的 LinkedIn、Facebook 和 Twitter 账户。

有了这些关系网之后，接下来的一步是通告天下，告之所有朋友你的未来计划。你将会发现，不少素未谋面的朋友会突然来找你，他们会主动帮你推荐一些公司和职位。如果你有广为人知的才能和可靠的技术，只需要让认识你的人知道你要找什么工作，大家都会乐意帮忙的。如果你有使用像 Facebook 和 Twitter 这样的社交网络，仅仅在上面发条消息没准儿就能帮助你联系到梦想的公司。你也可以更直接一点，问问朋友们有谁在相关公司工作。说不定你在谷歌工作的朋友认识一些微软的员工。各种方法不妨都试一试。

相中了一家公司，却找不到门路怎么办？答案是要学会宣传自己。找到那家公司的员工并和他们交朋友。这一点可以通过 LinkedIn 或者 Twitter 完成。如果他们需要帮助，想办法帮助他们。把自己和他们的圈子联系起来。当然不要做得过火，没人喜欢被盯梢。

与此同时，新开一个技术博客，涉及一些你感兴趣的科技内容，积极参与 GitHub，并尽可能**参与开源项目**。下载一些你感兴趣的软件和工具，看看是不是可以做进一步的改进。一旦发现有漏洞，马上向原先的开发者报告。上述方法可以让你在招聘者找到你之前就展示出自己的技能。很多招聘者会通过线上的个人资料寻找工作人选，而这些就是让他们来找你的好办法。有了这些，你就可以把 **GitHub 账号** 贴到你的简历中（在下一章，也会谈到 GitHub 在简历里的分量）。

不过，在与对方公司员工交往之初先不要寄上你的简历，以免显得过于功利。招聘者很清楚收到的是不是群发邮件。所以，重在质量，而非数量。比起泛泛而谈，一封专为某家公司量身定制的邮件会让你走得更远（在第 3 章，将谈到如何量身定制简历）。

要尽可能和最相关的招聘人员取得联系。如果找不到负责你心仪职位的招聘人员，也要说明自己的兴趣所在，再请别人帮忙转交简历。不过更好的是能明确知道那位招聘人员的姓名或是具体的职位名称。

## 2.4 求职渠道

除了前期建立校友、同事、朋友甚至猎头的关系网，你还得了解每个公司的薪资水平、绿卡政策，等等。

在求职之初，应该定一个 20 到 40 家目标公司的列表，结合你的关系网，找到相关的朋友和猎头，然后与他们进行积极沟通。如果找不到内部人推荐，则需要在公司网站上注册，经常查看、投递公司的新职位。

以下是几个查找薪水与公司信息的网站。

**www.h1bwage.com** H1B 薪水查询网站，可以查看公司给外国人办理 H1B 签证时提供的薪水。这个数字不含奖金和股票。

**www.glassdoor.com** 含有针对各个公司的常见面试问题及公司评价。

**www.careercup.com** 主要是技术面试题。

此外，还有一些常用的查找职位的网站。

**Indeed.com** 职位搜索聚合网站，将来自众多企业网站和招聘网站的信息综合在一起，信息比较全面。但也没有包含全部信息，有的小公司或者小网站上的信息就没有被收录。

**Linkedin.com** 根据笔者的亲身经历，LinkedIn 提供的高端职位较多，容易获取猎头提供的最新职位。如何充分利用 LinkedIn 是一门艺术。网站上有职位搜索功能，搜到的职位可以直接申请，也可以通过点击直达目标公司的网站提交简历。加入各种与职业发展相关的群组，每个群组均有人会发布新的职位信息。完善自己的简历和技能标签，

以便猎头找上门来。

**Monster.com** 老牌综合性求职网站，类似国内的 51job.com。在 LinkedIn 冲击下，现在是日趋衰落了，来自猎头而不是公司发布的招聘信息比例越来越高，而且职位大部分是合同工。尽管如此，你还得充分利用 Monster，因为在上面投递简历比较方便，而且只要定期更新简历，就有机会被猎头看中。笔者的第一份工作就来自 Monster，更新简历的时候被微软的猎头看中。综合性求职网站还有 careerbuilder.com，hotjobs.com。

**Dice.com** IT 行业招聘网站，网站功能类似 Monster.com，会有些中小型公司的招聘职位。此外，还有一些 IT 垂直类的招聘网站：[www.computerjobs.com](http://www.computerjobs.com)，[www.computerwork.com](http://www.computerwork.com)。

**Craigslist.com** 用来查看本地小公司的招聘信息。

# 第 3 章

## 简历

猎头或人事部门通常只会在每份简历上停留 20 秒钟，并在这短短 20 秒钟内，决定你的简历是否继续转交给技术经理，由技术经理或招聘经理决定是否开启面试流程。因此，一份有吸引力的简历是你入职的敲门砖。

我们先看看硅谷公司的人事部门是怎样阅读简历的。以下是一个列表，每个选项之前是加权分值。

- (+15 分) 如果简历中提及和职位相符的技能超过 5 次以上。
- (+8 分) 如果简历中提及和职位相符的技能 3 次到 5 次。
- (+4 分) 如果简历中提及和职位相符的技能 1 次到 2 次。
- (+4 分) 求职信 (Cover Letter) 提到了招聘人员或招聘公司。
- (+2 分) 简历含有求职信。
- (-10 分) 没有提到和职位描述相关的技能。
- (-15 分) 没有大学学位。

技术经理又是如何阅读候选者的简历的呢?

- (+11 分) 为开源软件贡献过代码。
- (+8 分) 有一个分享技术知识的博客。
- (+7 分) 编程竞赛参与者。
- (+7 分) 使用 LaTeX 生成的简历。
- (+7 分) 在 Google 和 Microsoft 实习过。
- (+6 分) 掌握脚本语言 (Python/Perl/Ruby)。
- (+5 分) 知道 3 种以上的编程语言。
- (+5 分) 拥有和职位相似的经验。
- (+4 分) 创过业、开过公司。
- (+4 分) 拥有 GitHub 账户。
- (+0 分) 有奖学金。
- (-1 分) 有博士头衔。
- (-2 分) 没有求职信。
- (-2 分) 在简历中说自己懂 Word/Excel。
- (-2 分) 在简历中有拼写和语法错误。
- (-3 分) 简历的文字字号太小。
- (-4 分) 所有的编程经验只是在学校中。

从中我们可以看出简历内容及简历结构都很重要。在接下来的几个章节中将阐述哪些要写、哪些不应该写在简历上。

### 3.1 简历特点

一份强大的简历应该能在短短几秒钟之内抓住阅读者的眼球。简历里的每一行都应

该有助于加固雇主想聘用你的决心。既然如此，为什么候选者还要把他那些含糊或无法证明的内容列在简历里呢？每份简历只需要珍贵的几行，简明扼要地把你的长处展现出来。在提交简历前，审视每一行并且问自己，为什么它能帮助我获得面试机会？如果你给不出一个理由，那这行就不该出现在简历上。一份良好的简历至少包含如下特点。

### ➤ 有针对性

在用打字机的年代，编辑一份简历是一个耗费体力的过程，求职者经常需要影印 200 份，并把相同的简历发到每家公司。优质的、有针对性的简历无疑能帮你加分。随着简历编辑成本的降低，定制高水准的简历也变得越来越易行。在这个竞争日益残酷的时代，这一点额外的工作或许不能让你的简历脱颖而出，但至少能让它登得上台面。千万要记住，**申请不同公司要使用不同的简历和求职信。**

必须在简历适当的位置提及应聘的公司的职位所要求的技能和经验，这可以使 HR 觉得你就是他们要找的人。撰写有针对性的简历并不难，通过公司网站的职位信息，以及公司团队人员的博客，了解一下他们需要有什么样经验的人，应聘者需要掌握哪些技能，或是什么样性格的人。你也可以问问自己，这个职位上的人会遇到哪些问题？如果是你又该如何解决？即使你还没有能力全部解决面临的问题，也要向 HR 表明你有解决问题的技能。

### ➤ 量化成果

有没有见过一个广告说“我们利润中的一部分捐赠给了慈善机构”？这种说法是很讨巧的。因为那一部分可能只是 0.0001%，但这说法又完全没错。作为面试官的我，当看到一份简历提到“提高原有系统性能”或者“降低服务延迟”时，我就有点纳闷：为什么你不告诉我究竟是多大幅度？

量化你的工作就能向雇主展示你能带来多大的影响，使你的陈述更直观，也更有意义。如果你表示完成这些量化的改进，从而降低了公司的成本或者提高了效率，雇主就

会想聘请你。对于程序员来说，用更技术化的术语量化成果则更具冲击力，比如缩短了 30 秒的延迟，修复了 10 个致命的漏洞，把算法复杂度从  $O(n^2)$  下降为  $O(n \log n)$ 。下面是一个例子，修改前为“fixed several bugs for our CRM system”，我们添加了修改 bug 之后为整个业务系统节省的成本比例，如“fixed three biggest bugs for our CRM system, leading to a 25 percent reduction in customer support calls”。

但是，也不能过多使用数字，因为你的成就可能在一个同行工程师面前令人印象深刻，但技术水平略低的人力资源 HR 可能才是审查你简历的那个人。要确保你的简历令每个人都印象深刻。

### ➤ 结果导向

如果简历读起来太像一份工作描述，很可能就是做错了。**简历应该能突出你做了什么，而不是你应该做什么。**在量化成果的基础上，加上结果导向，就像强有力的组合拳。每家公司都想要一个“能把事情做完”的员工。尽量避免使用这些词：贡献、参与和协助。因为这些词表明你曾经专注在一些职责上而非完成的事情上。毕竟，很多在微软工作的人都可以说“在微软 Windows 系统上我有贡献”，但这又说明了什么呢？

### ➤ 干净、专业和简洁

很多 HR 会因为一个错字扔掉你的简历。他们有这么多的简历要浏览，为什么要浪费时间在沟通能力差的人身上呢？因此，一定要从如下方面仔细检查简历。

- **简练。**在简历里，避免大块的文本。通常来说人们讨厌阅读大段文字，一般会跳过该段落。简历应该是一项项集合，一项大约含有一到两行的文字。
- **拼写。**避免拼写错误。通过从后往前阅读简历的方式来检查拼写错误。
- **语法。**可以使用 Microsoft Word 的语法检查器，但不要完全依赖于这个工具。找个母语为英语的朋友或者有强大的语法和拼写能力的人审查你的简历。
- **普通字体。**使用标准的字体，如 Times New Roman 或 Arial 字体，使用的字号不



小于 10 磅。Comic Sans 字体是绝对不能接受的。

- **一致性。**可以使用逗号或分号分隔列表中的项目，但必须一致。用句号结束每一个项。确保格式一致，比如加粗、下画线、斜体等。你用什么样的格式往往并不重要，重要的是它们应该保持一致。
- **避免使用第一人称。**避免使用“我”、“本人”或“我自己”。在整个简历中，除了客观的语句，尽量使用第三人称。

## 3.2 简历结构

通常投递简历时，会附上一封求职信（cover letter），简明扼要地阐述你满足这个职位的要求和胜任这份工作的理由及对这份工作的渴望。

标准简历按时间倒序阐述工作经历、教育背景等。一份标准简历至少包含工作经验和教育背景，也可以包括求职目标、技术技能、奖项和荣誉等。选择包括哪些部分取决于你的技能、背景和应聘的职位。

### ➤ 求职目标（Objective）

求职目标应该简明扼要，比如“PHP 软件开发工程师”。其实求职目标不是必填项，只有当它增加了重要的信息才使用，比如更换工作角色。例如，如果你以前做产品管理，但现在希望将工作重点转移到搜索营销角色上时，那么求职目标可能对指定的招聘人员有价值。但是，如果申请一个 PHP 职位而且你之前的职位也做 PHP 开发，则并不需要指定。

### ➤ 要点（Highlight）

把一些你认为招聘公司会看中的要点列在简历里。针对不同职位，写不同要点。例

如，应聘数据挖掘和数据分析相关的职位，你的要点可能是以下几点。

- Work experience in Microsoft and IBM.
- Extensive programming and system design experience, and solid background in data structure and algorithms.
- Experience with data mining, data analyst, and recommendation systems.

#### ➤ 工作经历 (Work Experience)

对于大多数应聘者，工作经验应该是简历中最重要的一部分。工作经验的介绍至少包含职位头衔、公司名称、地理位置和就业时间。如果你是在一家有很多产品的大公司工作，如微软或亚马逊，则可以列出你的团队。对于最近的工作经历应该有一到两行，每行大约四项。每项应着眼于你的成就（结果导向）而不是你的责任，应该尽可能用数字支持（量化成果）。如果你是应届毕业生，则可以把实习经历或者课程项目作为工作经历。如果你参与过开源项目，不管大小，都可以把链接贴到简历中，这会让人眼前一亮。

#### ➤ 教育 (Education)

工作经验通常比教育更重要。如果你是非应届生，虽然必须包括教育，但是这部分应该短，把更多的空间给工作经验。很多只有两三年工作经验的应聘者也只是简单列出他们的专业和学位，而使用大量篇幅介绍他们的工作项目。如果你是应届生，教育这栏则是你的重点，用大篇幅介绍你的教育背景，至少包含主修/辅修学位、课程作业、论文题目、GPA、参赛记录和获奖情况等。如果你的 GPA 不高，最好还是不要写在简历上了。

#### ➤ 技能 (Skills)

本栏应列出会应用的软件、编程语言或其他特定的技能。为了避免冗长，使用简洁

的列表方式。需要注意的是，不应该列出那些“显而易见”的技能，如会使用微软 Office。同样，熟悉 Windows 和 Mac 也不必写，除非可以列出一些不太明显的项，如 Linux。对于程序员来说，对编程语言需要注明哪些是精通，哪些是熟悉，哪些只是了解而已。根据面试官的亲身经历，同时精通 5 种以上的编程语言是不太可能的。如果你真的精通很多种编程语言，那就请列出应聘职位所需求的 3 种语言吧。

➤ 奖项和荣誉 (Awards and Honors)

如果有奖励或荣誉，可以在工作经验或教育栏里列出。如果想强调奖项，并且简历中还有空间，你也可以专门列出一个奖励专栏。需要明确的是这些奖励能帮助你同其他候选人区分开。无论使用哪种方式，你应该列出日期、获得该奖项的原因以及难度。如果可以量化获奖难度，那是最好不过了。

3.3 简历优化

一旦完成简历初稿，就需要进行一遍又一遍的优化。哪些用词可以改进？简历会不会太长？我的简历有没有特色？

➤ 简历用词

如表 1 中的单词可以让你的简历看起来更专业，也更准确。

表 1

项目相关	技术相关	管理相关
Approved	Architected	Assigned
Catalogued	Assembled	Attained

续表

项目相关	技术相关	管理相关
Classified	Built	Chaired
Compiled	Coded	Contracted
Dispatched	Designed	Consolidated
Implemented	Developed	Coordinated
Monitored	Devised	Delegated
Prepared	Engineered	Developed
Processed	Fabricated	Directed
Purchased	Initiated	Enhanced
Recorded	Maintained	Evaluated
Reorganized	Operated	Executed
Retrieved	Overhauled	Forced
Screened	Programmed	Improved
Specified	Redesigned	Increased
Tabulated	Reduced	Led
Validated	Remodeled	Organized
	Repaired	Planned
	Solved	Prioritized
	Trained	Produced
	Upgraded	Scheduled
	Utilized	Strengthened
		Supervised

### ➤ 不应包含的内容

对于在美国或加拿大的职位，简历不应该包括种族、宗教、性取向、婚姻状况，或其他任何与歧视有关的信息。照片也不应该包含在简历里，因为照片包含歧视有关的信息。招聘者讨厌这些额外信息，因为这些信息可能导致公司被人以种族歧视等方面提起诉讼。

### ➤ 简历不宜过长

简历不宜过长，正如前面章节所说招聘人员只有 20 秒钟时间读每份简历。招聘人员无法精读每一行简历去掘地三尺寻找最相关的人选。审查简历的过程更像是跳读而不是精读。那么，简历应该多长？在美国，两页的简历应该是合理的。更短的通常是更好的，比如一页。一页纸的简历迫使你选择性地填写，只包括最好的东西。简历内容过长，是因为价值一般的内容混合了进来。招聘者就会把你作为一般的候选人，而不是最好的候选人。

### ➤ 个性化简历

大多数的应聘者最大胆的也只是用彩纸打印自己的简历而已，而另一些人则会做一些更创新的尝试。例如，有一位应聘者就将自己的简历贴在一个巨大的弹力球上交给了谷歌，而另一位应聘者则将自己的简历印在了一块蛋糕上交给了 Facebook。虽然这不能帮他们拿到聘书，但可以肯定的是他们的简历都被看过了。这些不走寻常路的求职者不仅展现出了非凡的创意，而且还秀出了热情。在某些例子中，他们还能表现出自己是否懂这家公司。

最近一位美国设计师 Er ic G andhi 制作了一份别具一格的简历，看上去好像是 Google 的搜索结果（见图 3）。关键词是“富有创造力、勤奋、有天赋的优秀设计师”，搜索结果的第一个链接就是他自己，Eric Gandh i。他后来被著名的气象频道（Weather Channel）录用。



图3 个性化简历

但切记：这种剑走偏锋的求职方式很有可能遭到不喜欢这种方式的公司或招聘者厌烦。他们可不希望在“一本正经”的工作环境中见到这种“哗众取宠”的人。

# 第 4 章

## 面试

硅谷公司的程序员面试，也并非高不可攀，绝大部分还是考那些基本功，比如数据结构和算法、对面向对象编程的理解、语言的特性、领域内的专业知识等，跟国内的面试差不太多，甚至有的比国内公司还简单。笔者的一位朋友拿到谷歌总部的程序员 offer 之后，他想转到国内的谷歌，被要求加一轮谷歌中国的面试。很不幸的是额外一轮的面试失败了，他只能留在谷歌总部工作。

针对你的目标公司清单，从多方渠道获取每家公司的公司文化、面试流程、招聘渠道、面试方式、题目范围等方面的信息。不管面试有多难，其实都是可以准备的，就看你功夫下够了没有，**成功的窍门就是勤于练习**。要在美国工作，当然英语听、说、读、写也要足够顺畅流利，但并非要很好才行。一般对于技术人员的语言要求不算太高，但如果你的英语还有障碍，那还是要努力提高，否则你的职业发展会出现瓶颈，因为没人愿意跟交流困难的人共事。仅仅是口语，是可以在短时间内提升的，比如盲听网络视频、复述名人演讲、模仿美剧中的语音语调、参加口语培训班等。

时下 IT 求职市场火爆，竞争相当激烈，对软件工程师的要求高于以往。如果你是应届生，正在寻找第一份工作，你会感到十分紧张，压力很大。尤其是你还打算出国工作，去一个陌生的国家重新开始。企业和人一样都有鲜明的个性，如果不能与遇到的人相处好，那么你也不会和为你提供工作的企业相处好，反之亦然。为了确保自己完全适合一家公司，你需要提一些有意义的问题，寻找在薪酬以外的其他价值。比如，工作节奏快不快？领导如何？从现在开始它会不会在一年内成为一家快速成长的创业公司？大型知名公司的工作是否乏味？换句话说，公司在面试你的同时，你也要面试公司。

面试还有运气成分，这导致再充分的准备也无法确保你能获得梦寐以求的工作。笔者的建议是把你想要获得的工作按优先级进行排序，当你进行最后一次面试时，结局将如你所愿，因为面试是个积累经验的过程，通常面了几次后表现会更加成熟。换句话说，把你不太想去的公司的面试放在前面作为练习，积累经验后再开始面试你想去的公司。

## 4.1 面试流程

我们以某人的 Facebook 面试流程为例，透视硅谷公司是如何面试程序员的。

### 第一轮电话面试

输入一个字符串数组，有些字串是同位词（Anagrams），分组输出这些 Anagrams。

### 第二轮电话面试

实现两个长的数字串相乘。

### 现场面试（onsite）

一共四个面试官，每场面试 45 分钟，中途除了 40 分钟午饭时间之外，没有其他任何休息时间。

第一个面试官，编程题目（Coding Question）。给定一个函数，可从文件中读取固定大小；实现函数，根据指定的大小读取文件。



第二个面试官，设计题目 (Design Question)。一位经理加上一位旁听者。主要问 switch infrastructure 相关的东西，算是泛泛而谈。

第三个面试官，行为考察 (Behavior Question)。主要问：为什么要跳槽？为什么要来 Facebook？在你做过的所有项目中最得意的是什么？等等。

第四个面试官，编程题目 (Coding Question)。又有一个旁听者。实现一个固定容量的 pipe。于是悲剧发生了，事后忍不住免冠徒跣以头抢地。忘了这个是要考虑循环数组的。写完后，在面试官的冷眼下惶恐修补，但是还是超时了。因为“可恶”的 Facebook 文化，一定要留出时间来问问题。

加上电话面试总共 6 轮，其中 4 轮是编程面试，1 轮是技术设计，剩下 1 轮是行为考察。目前在硅谷程序员面试里，编程面试占绝大比例，并逐步取代基于项目经验的面试。这有很多原因，其中最大的原因之一是之前很多印度人求职时简历作假，项目作假，如果公司光是考核项目，很难甄别候选者，所以不管三七二十一，都要求候选者在白板上写程序。还有个原因是不少硅谷公司认为只要招到足够聪明的人就行，项目经验可以慢慢培养。我们会在下一个章节介绍如何准备编程面试。

当然，项目经验也很重要，几乎每轮有 5 到 10 分钟时间来讨论你的项目经验。如果我们从面试官的角度来考察候选者，我们会在意候选者的经验以及解决问题的能力。

### ➤ 经验

面试中，我们会问这些问题：你遇到最难的问题是什么？是如何解决的？你是怎么设计这个系统的？是怎么调试和测试你的程序的？你是怎么做性能调优的？什么样的代码是好的代码？等等。重要的是你对知识的运用和驾驭，对做过的事情的反思和总结。最好是能让面试官从你的经验中受益。

### ➤ 解决问题的能力

我们会出很难的题目，难到我们自己也没有解决方案，所以我们想从候选者那里得

到的不是那个解题的答案，而是解题的思路和方法。在面试官看来，解难题的过程更重要，通过解题过程来考察应聘者的思路、运用知识的广度和深度、应聘者是否有经验、沟通是否顺畅等。当然，最终是要找到题目的答案。通常从面试官的角度，他们会考察应聘者如下几个方面。

- 应聘者在解算法题时会不会分解或简化这个难题？
- 应聘者在解算法题时会不会使用一些基础知识，如数据结构和基础算法？
- 在讨论的过程中应聘者有没有钻研能力？进一步优化解决方案？
- 应聘者是否有畏难情绪？是否能承受一定压力？
- 应聘者是否有和我们交流的能力？如果以后成为同事，能否较快融入现有的团队里？

## 4.2 编程面试

每轮编程面试有 45 分钟，扣除双方自我介绍和提问的时间，花在编程上的时间大约为 30 分钟。由于受到面试时间的限制，面试的题目不会太难，比大学生编程比赛（ACM）题目简单很多，但是，需要一些编程面试技巧及对算法、数据结构的熟练掌握才能在限定时间内完成。这对要求在白板上写程序和代码无 Bug（Bug Free）的公司来说尤其重要，比如 Facebook。而习惯使用 IDE 来获取函数或校正拼写的应聘者应该特别注意。

在编程的面试中，光有解法却写不出来代码是行不通的，这会让面试官觉得你只会夸夸其谈，却不会编程。在编程面试里，切记“让代码说话”这条准则。针对每道面试题，通常应该进行如下步骤。

- **复述/提问：**用自己的话复述面试官的题目，以免偏题。面试官给出的面试题并

非一开始就很明确，需要多次提问来确定题意、边界条件、时间和数据结构限制等。

- **举例**：可以和提问同步进行，主要用来确认输入和输出结果。
- **观察**：通过举例来总结规律，思考可能使用到的结构和算法，然后设计一种最优算法。
- **编码**：和面试官沟通算法之后，开始在白板上编码。
- **测试**：使用个别例子，把代码测试一遍。

在以上5个步骤里，视时间充裕情况，有些步骤可以省略。比如，如果面试官已经把问题说得很清楚了，那么复述可以省略。在把代码交给面试官之前，一定先要自己检查一遍代码。当遇到难题时不要慌张，也许面试官考察的正是你的抗压能力，冷静下来，把常见的数据结构和算法往里套；实在没有头绪时，可以**要求面试官给些提示**（很多人不知道这一点，这也是硅谷公司和国内公司面试的不同之一），积极和面试官沟通你遇到的问题。当遇到见过的题目时，千万不要沾沾自喜，马上背出答案，这只能适得其反；你得再次确认题目是否和自己记忆中的一模一样，即便一模一样，你也得**假装思考几分钟**，然后把思路 and 方案告诉面试官。

编程面试所需的准备时间视个人的基础不同而不同，从2个月到1年不等。平时做项目时就试着减少对IDE的依赖，时刻记录工作过程中遇到的难点及攻克方案。对于如何准备编程面试，笔者提供的建议是：**练习、练习还是练习**。在本书第二部分和第三部分，笔者贴出了面试题的全部代码，更多时候是想让代码来“说话”。

## 4.3 注意事项

技术能力和项目经验需要靠一段时间的学习和积累。但是，面试的现场表现的确可以在短时间内准备和提高。在这里，我们分享一些面试的技巧和注意事项。

### ➤ 准备持久战

如果是电话面试，请准备好耳塞，这样便可以解放双手；确保网络畅通，方便在线写代码。如果是现场面试，请保证前一天充足睡眠，吃饱早餐（注意：美国的牛奶比国内浓，避免喝过多牛奶导致身体不适），提前 30 分钟到公司。咖啡或茶可以提神，午饭不宜过饱，以免下午犯困；在每轮面试间隔，争取几分钟的休息时间。

### ➤ 保持自信

走进面试房间之前，检查一下自己的衣着打扮。深吸一口气保持镇定，通过自我暗示，默默地告诉自己能行。走进房间，主动给面试官打个招呼，给一个简短有力的握手，留下一个自信的第一印象。交流时，定时保持眼神接触，这是我们中国人所缺乏的，毕竟两国文化不一样。

### ➤ 表达顺畅

不要觉得自己的英文说得不好，怕说错，说话很小声；有时候又因为紧张，越说越快。因为你说话声音很小、有口音，加上语速太快，所以面试官就很难听明白你要说什么。因此，说话的时候声音一定要足够大，让对方能听得很清楚。同时，你可以主动放慢自己的语速，也可以要求对方适当放慢他们的语速。不用担心你的英语不够好，只要能确保对方能理解你的意思就行了。在大部分硅谷的 IT 公司里，本身就拥有大量外籍员工，比如大量来自东欧和亚洲的母语非英语的工程师。他们可以理解你的英语表达并且愿意减慢语速。如果语言交流还有障碍，可以借用文字，在白板或白纸上写下关键的问题和方案。

### ➤ 确认双方对问题的理解一致

在 4.2 节里，我们谈到编程面试的第一步为**复述/提问**，其实这对所有面试都适用。硅谷的公司面试往往是使用工作里面碰到的实际问题，在此基础上做一定的简化，改成

可以在 30 分钟之内解决的问题。问题本身很有可能是似是而非的。如果你没有跟面试的同事确认你们俩对问题的理解是否一样，那很容易发生的情况是你按照自己的理解花 30 分钟写出了程序，然后面试官说“这不是我要的答案啊”。比较稳妥的方法是你主动针对问题提出一些讨论，比如用几个简单而有代表性的例子把问题的输入和期待的输出都描述清楚，双方都同意这一组输入输出，然后再动手解决，这也自动地给你一组测试例子了。

### ➤ 确认对方理解你的方案

笔者见过不少例子是面试者用很复杂的算法（往往需要一篇论文来证明其正确性）来得到一定的算法复杂度优化。大部分的工程师并没有那么专注于学术，面试官很可能看不明白。而面试者本人也很难在 10 分钟内把一篇论文的正确性说清楚。这样的结果使别人不可能很快地同意你的解决方案，而在 45 分钟的面试里面，面试官就很可能不会赞成录取你。因此，如果一个算法不能在 5 分钟之内解释清楚，就不要在面试时用这个算法。

### ➤ 互动交流

把面试官当成你的同事，而不是监考老师。在整个面试的过程中，技术题目答案的优劣大概只占一半的权重，同样重要的是看你能否从头到尾解决一个实际问题。你得在短时间内做出权衡：理解问题的本质、约束条件、时间和空间的取舍、面试时间限制和工作量的取舍及程序的易读性。而获取这些权衡，必须要做到主动沟通，从面试官那里得到帮助。最优秀的面试者往往把自己沉浸在与面试官共同工作的状态，主动分析问题，甚至对问题提出异议，一起讨论，就解决方案达成一致，然后挽起袖子写出一个漂亮的程序，再根据讨论的例子对程序进行测试，最后提出对自己的解决方案的看法及下一步可改进的地方。这样走下来，即便写出来的程序不是最优的，而你在整个过程里表现出来的经验和解决问题的能力都会让人刮目相看。

### ➤ 抓住提问环节

在面试最后都有 5 分钟左右的提问时间，**千万别浪费提问的机会**。在面试过程中，应该表现出就像已经获得工作邀请一样。这非常有趣，因为不仅是企业在面试你，你也在面试企业。果断地提问一方面体现出你对这份工作的渴望，另一方面也方便你以后挑选公司。泛泛的提问可能很难获得你想要的答案，所以除了问一些有关团队和技术的问题以外，还可以提出一些有关公司如何运营的细节问题，特别是针对创业公司。以下就是笔者曾经提出的对笔者后来有用的问题，看看哪一条适合你：

- 公司的决策速度如何？
- 工程师在测试一项产品提议时一般会采取何种措施？
- 团队间是如何协作的？
- 对新人是否有正式的指导计划？
- 新人能从公司学到多少东西？
- 会不会提供管理培训或尝试新事物的机会？

# 第 5 章

## 聘书与职业发展

经过一系列紧张激烈的面试，恭喜你拿到一家甚至多家硅谷 IT 公司的聘书 (offer)。那么，接下来的事情对你来说将是一个幸福的“烦恼”：我要不要接受这家公司的 offer？我应该选择哪一家公司呢？

当拿到聘书时，不要立即接受或者拒绝，可以礼貌地说：“我非常开心能有这个机会。请给我一段时间让我认真思考一下，再给您答复。非常感谢！”一般来说 HR 会给你一周到一个月的时间考虑，因为他们也知道你会同时面试好几家公司。接不接受 offer，取决于很多因素，比如待遇、职业发展、工作幸福感等。不少人选择去工资最高的公司，又或者是不好意思拒绝对方的盛情邀请，而去了一家 HR 比较主动的公司，不幸的是，在入职一段时间后你才发现当前这家公司并不是最适合自己的。要知道频繁跳槽对你的职业发展并不利。

## 5.1 聘书

如何评价一份聘书（offer）呢？它包括了薪水、奖金、股票、假期、医保等，而且还含有工作内容和职责。当你做出选择时，你必须考虑自己的职业方向、公司文化、未来的工作伙伴，甚至会考虑到自己的家人。然而，更加棘手的是，你不可能知道所有的事情，比如到底要工作多少个小时？每年能有多少加薪？等等。此外，当你阅读聘书时，还可能需要考虑与职业发展相关的问题，比如这份工作对于你的职业发展是否有利？是否会为你的简历增色？是否能帮助自己在职业道路上提升？以及会考虑到与薪资待遇相关的问题，比如他们付你多少钱？其他待遇有哪些（比如医疗、股票等）？与你自身幸福感相关的问题，比如你会喜欢这份工作吗？和同事会相处愉快吗？工作地点是你理想的吗？

在接下来的章节，笔者会帮助你分析聘书并且告诉你做出决策时需要考虑哪些因素，以便你为自己作出正确选择。

### 5.1.1 聘书要素

除了基本薪水以外，硅谷 IT 公司的聘书常常包括股票（stock）、股权（option）、签字费（sign-on）等。如何在不同的指标间进行比较和衡量呢？给每样都贴一个价格标签，然后与你计划在这家公司的年数相除。比如，假设亚马逊报价 10 万美元年薪加 2 万元签字费，微软为 5 千美元的签字费加上 10.5 万年薪。这两家公司哪家待遇好呢？这取决于你打算在这家公司干多久。如果两年后就离开，那么亚马逊更好。换句话说，在一家公司干的时间越长，一次性支付的津贴就越显得不重要。

要看透聘书的薪资待遇，就要看聘书中列出的所有项目，包括那些没有书面记录的



项目。下面列举的项目，拿到越多越好。和国内 IT 公司不同的是，硅谷 IT 公司也会给普通级别的程序员搬家费、股票（创业公司是股权）、签字费等，而不只是给新入职的高级工程师或管理层。

表 2

核心要素	其他福利
基本薪水	年终奖
签字费	年度加薪
搬家费	员工购股计划
优先认股权	401K 计划
无偿配股	医疗保险
休假	免费午餐、食品、健身房等

表 2 中的一些因素，诸如年终奖和每年的加薪幅度，不是很容易就能明确知晓，因为公司都不太愿意透露这方面的信息。但是如果你能找到一位内部员工打探一下的话，也许能了解一般标准是怎样的，好一些的情况又是怎样的。

在比较聘书时，一定要把工作地考虑进去。正如我们在前面章节提到的税率问题，各州个人所得税、消费税不一样，各地消费水平也不一样，你需要比较工作地点的生活成本。

5.1.2 决策因子

在考虑薪酬的同时，还需要考虑以下的决策因素。

➤ 绿卡政策

作为外国人，你没法忽视雇主的绿卡政策，除非只打算出国工作几年，攒些经历，

然后回国。如果希望以后在美国长久生活，接收 offer 前先问问 HR：公司对绿卡的申请是不是很支持？一般要多长时间才能开始？有些公司需要员工工作两年之后才开始协助员工申请绿卡，而有些公司在员工入职之时就开始办理。

### ➤ 个人目标

你5年以后想做什么？是开一家自己的公司？还是加入一家创业公司？又或者是想在某一个领域做世界一流的技术专家？还是想在一家大公司里面做管理或是做一个深耕技术的工程师？回答这个问题很重要，因为无论薪水多少、加薪速度多快，我们都会觉得太少。最终的成就感和幸福感往往来自于个人成长和个人目标的实现。找到一家能给自己的成长提供相应机会的公司，如果这家公司还能实现自己的个人目标，那就更好了。

如果你以后打算创业，那么加入有发展前景的创业公司更合适。如果你打算老老实实地在公司里爬梯子，那么可以向你未来上司或 HR 询问一下公司为员工提供的职业路线，不管是技术路线，还是管理路线。如果做得好，三年之后公司会提供什么样的职业道路？公司会为你的职业规划提供什么样的培训和支持？是否支持员工到大学里面在职攻读 MBA？是否可以在公司内改变职业路线，比如从技术换管理？跨部门调动是否有很高的门槛？等等。

如果你是行业新人，那么可能会更看重行业培训。通常，大公司比中小公司，有着更为严格的培训体系。比如谷歌，它会给每一位新员工进行一个为期两周的入职培训。通过这些课程，新员工可以了解谷歌作为一家公司的运作情况，并且深入了解自己本职工作的需求。除了新员工培训以外，有些公司还会提供一些进修课程，这些课程会安排在公司内部或是当地的大学进行。

### ➤ 晋升空间

几乎每家公司都会对工程师分等级，但是在聘书上不会写出你的等级。只有主动询问 HR，才能知道自己的等级。此外，你还得了解，整个公司的工程师等级是怎样定义

和划分的？如果有多份聘书，不同公司的等级在数字上不一样，那么如何换算不同公司的等级？技术团队如何做业绩评估？晋升的过程是怎么样的？是谁决定我的业绩和晋升？等等这些问题。

如果你想得到快速提升，那么加入成长型公司（或团队）是一个非常好的选择。成长型公司意味着要招入新人，当然这些新人要有人领导，这个领导者或许就是你。

此外，你还得了解公司内部的提拔机制。一些公司习惯从公司内部提拔优秀员工，而另一些则喜欢从外部吸纳高管人员。比如 Intel 就是一个常常从内部员工中选拔管理人员的公司。而在早期的谷歌，很多管理者都是外聘的。

### ➤ 公司文化和前景

快速增长的公司会有更多的学习和晋升机会，但淘汰率也更高；已经成型的大公司不太会因为业绩问题而淘汰个人，但更有可能解散整个部门。在专注产品的公司工作，更容易学到产品开发的知识，这样的公司更重视能把握整个开发流程的人才，而那些只希望在一个领域钻研的人则不一定做得开心；专注技术的公司则相反。工程师文化很强的公司会把工程师看成创造机会的原动力，所以聚集了很多具有很强自主性的顶尖人才，这些的公司对个人创造力和主动性要求也高；而销售文化很强的公司则往往把工程师看作成本，工资能压则压，工程师往往只需要跟进销售团队的反馈意见，不要求很强的主动性。

作为外国人，还有个因素不可忽视，那就是**工作的稳定性**。如果你所在的团队不稳定，或者整个公司在裁员中，那么这种情况对你申请绿卡会很不利，同时也会对你的生活构成重大威胁。

### ➤ 幸福感

没有幸福感的员工会减少工作时间，工作效率降低并且不久就会辞职。在接受一份

觉得可能会让自己不开心的工作之前，好好想想自己是否能够应付。接受工作之前，要想清楚的是，什么能让你快乐？是和你一起工作的人吗？还是因为这份工作能激发你的才智，获得成就感？又或是你认为这份工作很有意义可以影响人们的生活？等等。

其实通过一个细节也能看出在工作中你是否有幸福感，通过一个周末的休息，看看周一的你是精神百倍，还是无精打采、厌倦上班。充足时间的休息可以缓解身体疲劳，但没法解决厌烦的工作情绪。

你的顶头上司和你的同事也会影响你的幸福感。你也希望看到公司能有一个氛围能帮助刚从中国到来的新员工适应语言、工作、生活和文化的改变。你和上司的关系会在最大程度上决定你是否喜欢这份工作。**请务必和自己的未来老板进行一次谈话**，问问他这样的问题：在公司如何做算得上成功？很多朋友入职是因为公司和项目的吸引力，而离职是因为他们和上司关系不和。

当你做决策时，HR 并不能完全帮得上忙，毕竟两者对公司的看法角度不同，因此，你还得找公司里跟自己背景相似的员工聊一下。如果有校友或朋友在那个公司工作，一定要找他问问情况。如果没有，可以跟 HR 回信说：在做决定之前，能不能跟公司里的一个中国工程师聊一下？我想了解他们来美国的过程和工作生活的情况。一般 HR 是很乐意为你找这样的工程师。只有跟背景相似的员工聊天，才能看到公司更真实的一面。

### 5.1.3 薪酬谈判

在美国，很多交易是明码标价的，没有讨价还价的余地，但是薪酬谈判是个例外。假如拿到了 offer，在谈具体工作事项前，我们首先要解决一个大问题——薪酬谈判。很多人不愿意谈，或表现出这并不重要的样子。这很好，但是你必须准备好这种谈话，因为这是你在开始正式工作前唯一一次讨论薪酬的机会，一旦签订了合同，你的所得就会基于你的表现，除了晋升的时候，你不会有太多重新谈判的机会。

在面试的过程中，如果他们提问到你想要的薪酬或股权，你需要知道的是，这不是一个随便问问的问题，不要在这样的交谈中随意抛出你想要的数字。你可以告诉他们，自己尚未想好确切的数字，但可以告诉他们你希望获得的薪酬水准是多少。

### ➤ 永远不要接受第一次报价

薪酬谈判是多次来回的过程。HR 都是谈判高手，特别是大公司的 HR，他们每周都要负责好多个新人入职。HR 第一次给出的数字通常是留有余地的，而第二次的数字可能是 HR 跟管理团队讨论之后的结果。可以讨价还价的 offer 要素是基本薪水、签字费、搬家费、股票或股权，其他的要素，比如假期、401K 等几乎没有谈判余地。其中基本薪水和搬家费余地很小，因为这些数字和你的级别有关，而且公司不想改变现有的制度，以免激起老员工的不满情绪。不要小瞧签字费和股票，每年你从股票获得的钱可能比你的薪水还高。笔者还见过 Facebook 给新毕业的博士发十万美元的签字费的情况。

### ➤ 最好有备选方案

如果能拿多个 offer，你也就能大概知道自己值多少钱了。不少财大气粗的公司，比如谷歌和 Facebook，能提供其他竞争对手相同的薪水给你。有了多个 offer，在谈判过程中你就站到了主导地位。如果只拿到了一个 offer，还有其他公司没有答复，可以跟其他公司说你有一个待决定（pending）的 offer，让他们加快速度；而跟给你 offer 的公司说你需要一些时间决定，希望对方能够理解和体谅。在这个过程中，HR 很可能会敦促你快速做决定，你需要诚实告之当前进展。

### ➤ 做好调研

如果你了解整个行业的薪资水平和公司给予相似员工的待遇情况，你便可以对提出多少要求做到心中有数。浏览一下我们在第 2 章求职渠道中提到的薪资网站，比如 [www.glassdoor.com](http://www.glassdoor.com)，对薪资待遇做一番调查。

### ➤ 适可而止

公平性是大家都愿意接受的一个讨论基础。从公司角度来说，如果你的待遇低于市场待遇，那你会觉得没有被公平对待，很可能会很快离开公司，这对公司也不是一件好事情。从个人角度来说，一般抱着“不吃亏就好”的心态，不要想着从中得到比别的类似背景的工程师更好的待遇。待遇总是跟期望值挂钩的，如果你真的要到了更好的待遇，公司对你的期望值也会更高。

讨价还价的过程本身很容易让人过于专注在薪酬数字上从而忽视了对其他因素的考虑。在讨价还价的过程中要保持适可而止的态度，以免公司收回 offer。笔者的一个朋友拿到了一家创业公司的 offer 之后，讨价还价了许久，激怒了该公司的 CEO，最后公司把 offer 收回了。

#### 5.1.4 接受、延期或婉拒

在硅谷绝大部分 IT 公司的 offer 是 at-will employment，即雇佣双方的任何一方都可以随时随地无条件地解除雇佣的协议。类似国内有工作期限的合同，这种合同并不是给全职的正式员工，而是给在美国称之为“合同工”（contract）的人。当雇员决定离开公司时，一般应在离开前的两周要礼貌性地通知公司。公司如果需要解除雇佣协议，一般应提供两周或更多的补偿，这取决于双方谈判的结果。如果你**接受**了 offer，要时刻与对方 HR 和你未来上司保持联系，并表示如果公司有什么变动，请及时告诉你，因为公司可以随时解除雇佣关系。

当你还在面试其他公司而当前这家公司提供的这个职位又不是你最想要的，或者其他什么原因，这时你需要向对方申请**延期**做决定。在给你 offer 的时候，公司一般都会设个最后期限。其实公司有充分的理由设定最后期限，因为他们不能把位子留给你的同时还去面试其他人选，也不可能把下决定的时间拖得太长。如果你需要延期，就和招聘者实话实说，向对方解释你需要延期的原因，你和其他公司的接触进行到了哪个阶段以

及你什么时候可以做出决定等。

**婉拒**并不意味着断绝关系。其实更应该看做是“改期再约”。你是喜欢这家公司的，所以千辛万苦完成了所有的招聘程序；他们也是喜欢你的，所以才给你发出了聘书。你们之间的这层关系以后可能用得上。因此，婉拒对方的时候，你应该尽可能通过你们一直以来采用的沟通方式同对方接触。也就是说，如果那个招聘负责人总是给你打电话，那你就要和他致电沟通。换言之，如果你一直是和经理电子邮件往来的，那你就要第一时间给那位经理写邮件说明。不论是写简短的邮件还是打电话，你都应该找那个与你经常联系的人沟通。

## 5.2 职业发展

除了深耕技术和做好自己的工作以外，笔者还有如下与职业发展相关的建议。

### ➤ 表现专业

讨论问题时对事不对人。积极汇报工作进展，多给上司和组里成员发送 E-mail，留下书面证据。一方面让上司知道你的工作，另一方面也让其他人知道你在干活。口语是我们的劣势，那 E-mail 就是我们最好的交流方式。积极回复群里邮件，不是说让你多揽活干，而是让大家知道你能干这个活，前提是你了解同事所做的工作。

### ➤ 摸清评价体系

你必须了解是谁评价你的工作表现 (performance) 及评价体系的要素是什么，做到有的放矢。和他搞好关系，随时让他知道你的进度。比如，有些公司是经理评价你的工作表现，但是你主要跟组长干活。在这种情况下，你要想尽办法让经理知道你的进展，千万不要把所有的宝押在组长上面，以为他会把你的表现如实上报给经理。

### ➤ 迎合上司

做技术的程序员多少有些傲气，很多时候你觉得一件事情应该这么做，但是老板却不这么认为。迎合上司比固执己见更重要。上司都是孤独的，他需要有人迎合、认同他，他也会提拔和他意见一致的人。

### ➤ 寻找导师

找到一位职业导师，比如直接上司或者资深工程师，他可以让你少走弯路，更快融入团队。如果有机会，你也可以做别人的导师，扩展你的人脉。

### ➤ 抓住机会

通常来说，如果你为一个快速发展的项目干活，那么你得到提升的机会就越大。因此，必要时候，看准了就换组、换项目。如果关键岗位突然出现空缺，而你又有八成把握胜任这个职位，在这个时候你应该毫不犹豫挺身而出，积极竞聘，而不是等到你有了百分之百的能力时再竞聘。你要知道很多高官都是在岗位上把能力锻炼出来的。如果你一直在走技术路线可能很难被提升，因为做策略的工作通常比做工程的工作更容易得到提升，由于策略的工作直接影响公司收入，从而能引起公司上上下下所有人的关注。



---

## 第二部分 数据结构

---

在第二部分和第三部分章节里，我们重点阐述编程面试题所涉及的数据结构，以及常用的算法。

在本书中，每道面试题的标题后面均附上了星号标示的难易度，即一颗星代表容易，三颗星代表中等，五颗星代表非常难。



# 第 6 章

## 数组

数组是一种最常见的数据结构，并且在众多面试题中占有很大的比例。有关数组的面试题解法繁多，但通常归纳为如下几种：排序、俩指针、动态规划、排列组合等。对于二维数组的遍历，还可以考虑深度优先遍历或者广度优先遍历的方法。

### 面试题 1：两数之和 I ☆☆

给定一个整型数组，是否能找出其中的两个数使其和为某个指定的值？

提问

应聘者：输入数组是有序的吗？

面试官：你可以假定是无序的。

### 举例

输入数组为{1, 5, 7, 3}以及指定的目标值为 10, 我们可以从中找出两个数 3 和 7, 和为 10。

一种非常直观的办法就是使用两个循环, 从数组里提取一个数, 然后在该数之后的部分找出另外一个数, 计算这两个数之和, 看看是否等于指定的值。这种暴力破解的方法显然不是面试官想要的。那么, 能否降低暴力破解  $O(n^2)$  的时间复杂度呢? 可以尝试先把该数组排序, 排序之后, 从首尾两端移动, 一次移动一端的指针, 直至相遇或者找出两个数的和为指定的值为止。假设当前首尾指针分别为  $i$  和  $j$ , 其中  $i < j$ , 如果  $A[i]$  与  $A[j]$  之和大于指定的值, 那么要找的两个数一定在  $j$  的左侧, 因此, 尾指针  $j$  往前移动一步。如果  $A[i]$  与  $A[j]$  之和小于指定的值, 那么要找的两个数一定在  $i$  的右侧, 所以首指针  $i$  往后移动一步。

如何证明如上的解法是正确的? 反证法。假设  $A[i]$  与  $A[j]$  之和大于指定的值, 并且有个  $k$ ,  $k$  大于  $j$ , 使得  $A[i]+A[k]$  为指定的值。由于  $A[j] < A[k]$ , 并且  $A[i]$  与  $A[j]$  之和大于指定的值, 那么  $A[i]+A[k]$  必定大于指定的值, 所以不可能有  $k$ , 而且  $k$  在  $j$  的后面, 使得  $A[i]+A[k]$  为指定的值。

---

```
boolean hasSum(int[] A, int target){
    boolean res=false;
    if(A==null || A.length<2) return res;
    Arrays.sort(A);
    int i=0, j=A.length-1;
    while(i<j){
        if(A[i]+A[j]== target){
            res=true;
            break;
        }elseif(A[i]+A[j] > target){
            //目标值过小, 则向前移动尾部指针, 减小两数和
            j--;
        }else{
            i++;
        }
    }
    return res;
}
```

---

---

```
        //目标值过大，则向后移动首部指针，增加两数和
        i++;
    }
}
return res;
}
```

---

通过排序，使得时间复杂度降至  $O(n\log n)$ 。在 `while` 循环里，至多扫描一遍数组就可以得出结果，所以最终程序的时间复杂度为  $O(n\log n)$ ， $n$  为数组的长度。能否继续降低时间复杂度呢？如果允许使用额外的存储空间，那么答案是可以，详见“两数之和 II”。

## 面试题 2：两数之和 II ☆☆☆

给定一个整型的数组，找出其中的两个数使其和为某个指定的值，并返回这两个数的下标（数组下标是从 0 开始）。假设数组元素的值各不相同，则要求时间复杂度为  $O(n)$ ， $n$  为数组的长度。

### 观察

扫描一遍数组即需要  $O(n)$  的时间，如何让查找时间是常数时间呢？可以考虑使用哈希函数。

先扫描一遍数组，把键值 `<value, index>` 存入哈希表；第二次扫描的时候，检查 `target` 与当前的差值是否在哈希表中。如果遇到 `target` 与当前元素的差值为当前元素的大小，怎么办？根据题意，数组元素的值各不相同，不可能出现第二个值为当前元素的元素，因此出现这种情况，不应返回当前的下标。

因为哈希表的查找是常数时间，所以程序的时间复杂度还是  $O(n)$ ，而空间复杂度为哈希表大小，即数组大小  $O(n)$ 。

---

```
int[] twoSum(int[] A, int target){
    int[] res = {-1,-1};
    if(A==null || A.length<2) return res;
    HashMap<Integer, Integer> hm = new HashMap<Integer, Integer>();
    for(int i=0; i<A.length; i++){
        //扫描一遍，存储值与下标
        hm.put(A[i],i);
    }
    for(int i=0; i<A.length; i++){
        if(hm.containsKey(target-A[i])&& target != 2*A[i]){
            //获取结果的两个下标
            res[0]= i;
            res[1]= hm.get(target-A[i]);
            break;
        }
    }
    return res;
}
```

---

### 扩展问题

如果数组可能出现相同值的元素，那么上述算法还能正确解决吗？

## 面试题 3：两数之和 III ☆☆☆☆

设计一个类，包含如下两个成员函数：

Save(int input)

插入一个整数到一个整数集合里。

Test(int target)

检验是否存在两个数和为输入值。如果存在这两个数，则返回 true；否则返回 false。

## 提问

问题：在这个整数集合中是否允许有相同值的元素？

面试官：允许。

这道面试题是“两数之和 II”的变体，但必须注意的是整数集合里的数是可能有重复的。如果 Test 函数的输入值 target 为集合里某个数的两倍，我们该如何判断呢？在“两数之和 II”的哈希表里存放<值，下标>，而在这里就需要改变为记录每个数出现的次数，在哈希表里存放<值，个数>。如果某数两倍为 target 值，那么只有该数出现两次或两次以上，才能返回 true。

---

```
public class TwoSum {
    HashMap<Integer, Integer> hm =new HashMap<Integer, Integer>();
    public void save(int input){
        int originalCount = 0;
        if(hm.containsKey(input)){
            //判断是否已经存在，如果存在，则读取个数
            originalCount = hm.get(input);
        }
        hm.put(input,originalCount + 1);
    }

    public boolean test(int target){
        Iterator<Integer> it = hm.keySet().iterator();
        while(it.hasNext()){
            int val = it.next();
            if(hm.containsKey(target - val)){
                //需要判断一下是否有这种特殊情况
                boolean isDouble = target == val * 2;
                if(!isDouble && hm.get(val) == 1)
                    return false;
            }
        }
        return true;
    }
}
```

---

## 面试题 4：数组旋转 ☆☆☆

返回将一维数组向右旋转  $k$  个位置的结果。比如，一维数组  $\{1,2,3,4,5\}$ ， $k=2$  时，返回结果是  $\{4,5,1,2,3\}$ 。要求常数级空间复杂度，允许修改原有数组。

### 观察

如果允许额外分配线性空间，那么可以错位复制原有数组的元素。如果允许修改原有数组，那么我们可以通过三次反转数组来实现数组旋转，不需要申请额外空间，并且每次反转时间为  $O(n)$ ，从而实现线性的时间复杂度和常数级的空间复杂度。

三次反转数组：第一次反转整个数组；第二次反转数组的前  $k$  个数；第三次反转数组剩下的数。例如，

一维数组  $\{1,2,3,4,5\}$ ， $k=2$

第一次反转：5,4,3,2,1

第二次反转：4,5,3,2,1

第三次反转：4,5,1,2,3 即为最终结果。

---

```
int[] rotateK(int[] A, int k){
    if(A==null || k>=A.length) return A;
    reverse(A, 0, A.length-1); //反转整个数组
    reverse(A, 0, k-1); //反转前 k 个数
    reverse(A, k, A.length-1); //反转剩下的数
    return A;
}
```

//辅助函数，反转从 start 到 end 的数

```
void reverse(int[] A, int start, int end){
    while(start<end){
```

---



---

```
        //交换 A[start]和 A[end]两个数
        inttemp = A[start];
        A[start]= A[end];
        A[end]= temp;
        start++;
        end--;
    }
}
```

---

## 面试题 5：最大下标距离 ☆☆☆☆

给定一个整型数组，找出最大下标距离  $j-i$ ，当且仅当  $A[i]<A[j]$  和  $i<j$ 。

### 观察

直观的方案是对每个元素，从其后找出比其大的元素，并计算两者下标的差值，取差值中的最大值。该方案时间复杂度为  $O(n^2)$ ， $n$  为数组长度。那么有没有更快的解决方法？

我们可以试试排序。为了记录元素下标，使用一个哈希表存储下标，如果数组允许有相同元素，那么哈希表的值为一个数组；将数组排序之后，使用俩指针方法，一个指针从头部开始扫描，另一个指针从尾部开始，计算俩指针的下标差值，直至俩指针相遇。记录下标和两端开始扫描的时间复杂度为  $O(n)$ ，所以时间复杂度为排序的复杂度  $O(n\log n)$ ，而空间复杂度为  $O(n)$ ，即哈希表的大小。那么是否还有更快的线性时间的方法呢？

假设输入数组为  $\{5,3,4,0,1,4,1\}$ ，比较第一个元素 5 和第二个元素 3，如果后面的元素比 5 大，那么我们需要记录 5，因为 5 可能用得上。如果后面的元素介于 3 和 5 之间，则说明 3 能用得上。继续比较 3 和 4，如果后面的数比 4 大，我们只要记录 3 即可，因为 3 比 4 小，并且 3 在 4 前面，所以 4 是无用的。如果后面的数比 4 小，则 4 不可能用

得上。通过上面的观察，我们需要记录从第一个元素开始的下降序列，然后使用一个指针从尾部开始逆向扫描，求出下标的最大距离。对于数组{5,3,4,0,1,4,1}，下降序列是{5,3,0}，初始化  $i=3, j=6$ ， $j$  从尾部开始扫描。

第一步：初始化  $i$  和  $j$ ， $i=3, j=6, A[i]=0 < A[j]=1, \text{maxDistance}=3$

第二步： $i$  往前移动， $i=1, j=6, A[i]=3 > A[j]=1$

第三步： $j$  往前移动， $i=1, j=5, A[i]=3 < A[j]=4, \text{maxDistance}=4$

第四步： $i$  往前移动， $i=0, j=5, A[i]=5 > A[j]=4$

第五步： $j$  往前移动， $i=0, j=4, A[i]=5 > A[j]=1$

第六步： $j$  往前移动， $i=0, j=3, A[i]=5 > A[j]=0$

第七步： $j$  往前移动， $i=0, j=2, A[i]=5 > A[j]=4$

第八步： $j$  往前移动， $i=0, j=1, A[i]=5 > A[j]=3$

第九步： $i$  往前移动， $i=-1$ ，结束程序

如何证明我们的观察是正确的？反证法。假设存在最大下标距离的两个下标  $i$  和  $j$ ， $i < j$ ， $A[i] < A[j]$ ，并且  $A[i]$  不在从第一个元素开始的下降序列里。我们一定还能找到一个  $i'$ ，使得  $i' < i$ ， $A[i'] < A[i]$ ，这时候最大下标距离应该是  $j-i'$ ，而不是  $j-i$ 。

---

```
int maxIndexDistance(int A[]){
    if(A==null || A.length<2) return 0;
    boolean inDescSeq[] = new boolean[A.length];
    int min=A[0], n=A.length;
    inDescSeq[0]=true;
    for(int i=1; i<n; i++){
        if(A[i]<min){
            //做下降序列的标记
            inDescSeq[i]=true;
            min=A[i];
        }
    }
    int maxDist=0, i=n-1, j=n-1;
```

---

---

```
        while(i>=0){
            if(inDescSeq[i]==false){
                i--; 倒序找出下一个下降序列的元素
                continue;
            }
            while((A[j]<=A[i])&&(j>i))
                j--; 从后往前移动直至找到符合的元素
            if((j-i)>maxDist){
                maxDist=j-i;
            }
            i--;
        }
        return maxDist;
    }
}
```

---

以上总共做了三次线性扫描，每次扫描的时间复杂度为  $O(n)$ ，因此，整个算法的时间复杂度为  $O(n)$ 。

## 面试题 6：重叠区间个数 ☆☆

给定多个可能重叠的区间，找出重叠区间的个数。区间定义如下：

---

```
class Interval{
    int start;  //起点
    int end;    //止点
    Interval(int a, int b){
        start=a;
        end=b;
    }
}
```

---

### 举例

输入: [1, 5], [10, 15], [5, 10], [20, 30]

输出: 3

本题对面试者考察的是对复杂数据结构排序的能力。假如输入四个区间: [start1, end1], [start2, end2], [start3, end3], [start4, end4], 我们不区分区间点的类型, 而是根据区间点的大小排序, 得到 start1-start2-start3-end1-end2-end3-start4-end4。然后扫描上面结果, 当遇到起点时, 重叠个数加一, 并且记录重叠个数的最大值; 否则当遇到止点时, 重叠个数减一。

首先, 要定义区间的类, 实现 Comparable 接口, 含有起点与止点的值和类型, 还要重写用于排序的 compareTo 函数。

---

```
class Point implements Comparable<Point>{
    int value; //数值
    int type; //点的类型, 0 为起点, 1 为止点
    Point(int v, int t){
        value = v;
        type = t;
    }
    //还需要实现 compareTo 函数, 以便排序
    public int compareTo(Point p){
        if(this.value == p.value){
            return 0;
        }else if(this.value > p.value){
            return 1;
        }else{
            return -1;
        }
    }
}
```

---

其次, 区间转换为点, 并将点排序, 然后统计重叠的个数。

---

```
int getOverlappingCount(Interval[] A){
    int max=0, count=1;
    if(A==null || A.length==0) return max;
    Point[] points = new Point[A.length*2];
    for(int i=0; i<A.length; i++){ //转为可排序的点
        points[2*i] = new Point(A[i].start, 0);
        points[2*i+1] = new Point(A[i].end, 1);
    }
    Collections.sort(points); //排序
    for(int i=0; i<points.length; i++){
        if(points[i].type==0){
            count++; //起点
            max = Math.max(max, count);
        }else{
            count--;
        }
    }
    return max;
}
```

---

此算法时间主要消耗在对点的排序上，因此，时间复杂度为  $O(n\log n)$ 。

## 面试题 7：插入区间 ☆☆☆

给定一个没有重叠的区间序列，现插入一个新的区间到该序列中，要求维持没有重叠的情况。假设该序列存放的区间是有序的。区间定义如题目“重叠区间个数”。

### 举例

假定我们已有两个不重叠而且排好序的区间： $[1, 5]$ ,  $[6, 10]$ ，现插入一个新的区间  $[4, 6]$ ，那么最后结果是  $[1, 10]$ 。

这道题的关键是找出与插入区间重叠的区间（可能有多个），然后合并成一个新的区间。对于其他非重叠的部分，直接加入结果集合就可以了。如何合并两个有重叠的区间呢？例如， $[1, 5]$  和  $[4, 6]$ ，因为  $4 \leq 5 \leq 6$ ，所以这两个区间是重叠的，那么合并后新区间的起点取双方起点的最小值 1，止点取双方止点的最大值 6，因此新的区间为  $[1, 6]$ 。

---

```
ArrayList<Interval> insert(Interval[] intervals, Interval newInt){
    ArrayList<Interval> res = new ArrayList<Interval>();
    if(intervals==null){
        //输入是 null，那么直接返回新区间
        res.add(newInt);
        return res;
    }
    int i=0, n=intervals.length;
    while(i<n&& newInt.start> intervals[i].end){
        //加入非重叠的区间
        res.add(intervals[i++]);
    }
    while(i<n&& newInt.end >= intervals[i].start){
        //合并重叠的区间到新的区间
        newInt.end=Math.max(newInt.end, intervals[i].end);
        newInt.start=Math.min(newInt.start, intervals[i].start);
        i++;
    }
    //加上合并后的新区间
    res.add(newInt);
    //加上剩余的部分，如果有的话
    while(i<n) res.add(intervals[i++]);
    return res;
}
```

---

## 面试题 8：合并区间 ☆☆☆☆

给定一个区间集合，合并里面重叠的区间，并返回新的不含重叠区间的集合。例如，输入：[1, 5], [6, 10], [4, 6], [15, 20]。输出：[1, 10], [15, 20]。

首先，为了方便合并区间，先对集合进行排序。然后，初始化结果集。最后，扫描原始集合。如果原始集合的区间和结果集的最后一个区间重叠，那么合并这两个区间，并更新结果集；如果不重叠，则将此区间加入结果集。

---

```
ArrayList<Interval> merge(ArrayList<Interval> intervals) {
    ArrayList<Interval> res = new ArrayList<Interval>();
    if(intervals == null || intervals.isEmpty()) return res;
    Comparator<Interval> comparator = new Comparator<Interval>() {
        @Override
        public int compare(Interval a, Interval b) {
            if(a.start < b.start) return -1;
            else if (a.start > b.start) return 1;
            else
                //start值相同，还需要比较 end
                if(a.end < b.end) return -1;
                else if (a.end > b.end) return 1;
                else return 0;
        }
    };
    //调用 java 排序函数
    Collections.sort(intervals, comparator);
    for (int i = 0; i < intervals.size(); i++) {
        Interval current = intervals.get(i);
        if(res.isEmpty()) {
            //初始化结果集
```

---

---

```

        res.add(current);
    } else {
        Interval last = res.get(res.size() - 1);
        if (last.end >= current.start) {
            // 合并并更新结果集
            last.end = Math.max(last.end, current.end);
        } else {
            // 若没有重叠，则直接加入结果集合集里
            res.add(current);
        }
    }
}

return res;
}

```

---

## 面试题 9：数组配对 ☆☆☆

给定  $N$  个整数， $N$  为偶数，是否能找到  $N/2$  对，使得每对和能被  $k$  整除。  
注意：每个元素只能出现在一个配对中。

### 观察

如果某个数能被  $k$  整除，那么该数除以  $k$  的余数为零。假设  $a$  和  $b$  之和能被  $k$  整除，即  $(a+b)\%k=0$ ，如果  $a\%k=z$ ，那么当  $z$  不等于 0 时， $b\%k=k-z$ ；当  $z$  等于 0 时， $b\%k=0$ 。通过上面的推导，我们需要记录这个数组除以  $k$  之后的余数情况。然后从余数里找出配对。

使用额外一位数组保存余数情况，数组大小为  $k$ ， $A[i]$  记录余数为  $i$  的元素的个数。有了这个余数个数的数组之后，我们需要考虑如下边界情况。

- 1) 被  $k$  整除的个数是否为偶数？如果是奇数，那么就不可能找到和被  $k$  整除。



2) 如果  $k$  为偶数, 那么余数为  $k/2$  的个数是否为偶数?

3) 余数为  $i$  和余数为  $k-i$  的个数是否相同?

---

```
boolean checkPairable(int[] nums, int k) {
    if (k <= 0) return false;
    //额外数组保存余数情况
    int[] counts = new int[k];
    for (int num : nums) {
        //初始化余数数组
        counts[num%k]++;
    }
    //整除 k 的个数若不是偶数, 返回 false
    if (counts[0] % 2 != 0) return false;
    //k 是偶数时, 查看余数 k/2 的个数是否为偶数
    if (k % 2 == 0) {
        if(counts[k/2] %2 != 0) return false;
    }
    for (int i = 1; i <= k/2; i++) {
        //余数配对
        if(counts[i] != counts[k-i]) return false;
    }
    return true;
}
```

---

## 面试题 10: 数位重组 ☆☆☆

给定两个数组表示的整数, 比如  $x=1234=\{1, 2, 3, 4\}$ ,  $y=2410=\{2, 4, 1, 0\}$ , 返回第一个整数的重组后的值最接近第二个整数, 并且大于第二个整数。假设两个整数的数组大小相同, 并且肯定能找出符合条件的数。输入  $\{1,2,3,4\}$  和  $\{2,4,1,0\}$ , 返回  $\{2,4,1,3\}$ 。

从  $y$  的高位开始，挑一个和该位数字相等或更大的数字，直至到达数组的尾部。为了方便从  $x$  里挑出一个与  $y$  对应的相等或更大的数字，我们需要：

- 1) 将  $x$  数字进行排序。
- 2) 使用一个一维数组记录  $x$  数字的使用情况。

如果从  $x$  挑出来的是相等的数字，那么我们对剩下数位重复上述操作；否则，如果挑出来的是大于  $y$  的数字，后面的数位只要复制递增未使用的  $x$  数位就可以了。例如， $x = \{1, 2, 3, 5\}$ ,  $y = \{2, 4, 1, 0\}$ :

第一步：从  $x$  挑出 2，即  $y$  的首位数值

第二步：从  $x$  挑出 5，因为最接近 4 且大于等于 4 的值为 5

第三步：因为 5 大于 4，只需复制  $x$  未使用的数字，即  $\{2, 5, 1, 3\}$

---

```
int[] getClosestBigger(int[] x, int[] y){
    int len=x.length;
    int[] res = new int[len];
    //对 x 排序，生成递增序列
    Arrays.sort(x);
    int i=0, k=0;
    boolean used[] = new boolean[len];
    for(int j=0; j<len; j++){
        i=0;
        while(i<len&& (used[i] || x[i]<y[j])) i++;
        //挑出了一个等于或者大于的数字
        res[k++]=x[i];
        used[i]=true;
        if(i==len-1) break;
        if(x[i]>y[j]){
            //大于情况，复制未使用的递增序列
            for(i=0; i<len; i++){
                if(!used[i]) res[k++] = x[i];
            }
        }
    }
}
```

---

---

```
        }  
        break;  
    }  
}  
return res;  
}
```

---

## 面试题 11：产生随机数 ☆☆

给出一个整型数组，以及数组的最大值 max 和最小值 min，写一个函数产生一个在 [min,max] 范围内的随机数，该随机数并不在数组里。假设总能找到该随机数。

### 观察

分成两步。首先产生从 min 到 max 之间的随机数，然后判断该数是否已经存在数组里，如果已经存在了，则重新挑选随机数，否则，返回该数。如果顺序查找，则判断该数是否存在复杂度是  $O(n)$  时间。在理想的随机数产生器下，最坏情况是选出的  $n$  个随机数均存在数组中，第  $n+1$  个随机数才符合条件，因此，整个算法复杂度是  $O(n^2)$ 。能否进一步提升效率呢？能提升的部分只能是在判重的时候，利用二分法从  $O(n)$  降至  $O(\log n)$ 。

解题思路：在 [min,max] 范围内产生一个随机数，判断该数是否在数组里。如果是，则重复上面过程，即产生另一个随机数；如果不在数组里，则返回该值。为了提高查找效率，我们可以先对数组进行排序，然后使用二分查找，可以把时间复杂度从  $O(n^2)$  降至  $O(n \lg n)$ 。

---

```
void numInRange(int[] intList, int min, int max) {  
    Arrays.sort(intList); // 对输入数组进行排序  
    int range = max - min + 1;  
    Random rand = new Random();  
}
```

---

---

```
int result = rand.nextInt(range) + min;
//二分查找，直至该随机数不在数组里
while(Arrays.binarySearch(intList, result) >= 0){
    result = rand.nextInt(range) + min;
}
System.out.println(result);
}
```

---

## 面试题 12: Top K I ☆☆☆

求一维数组中最小的 k 个数。

### 提问

在给出答案之前，需要和面试官沟通，明确如下几点。

- 机器内存是否能容纳整个数组？
- 最小的 k 个数是否要求有序？
- 能否修改输入数组？
- 时间复杂度的要求：是  $O(n\log n)$ ，还是  $O(n)$ ？

### 方案一：排序

先把数组从小到大进行排序，取前 k 个数。时间复杂度为  $O(n\log n)$ 。如果数组过大，机器内存无法同时容纳整个数组，则需要使用外部排序。

### 方案二：使用堆

- 1) 创建一个最小堆，初始化大小为 k；堆顶为堆的最大元素。
- 2) 扫描一遍数组，往最小堆插入数据，如果堆的元素个数已经达到 k，那么新元素

需要和堆顶比较，如果小于堆顶，则移除堆顶，插入新元素。

3) 最终我们得到  $k$  个最小的元素。时间复杂度为  $O(n\log k)$ 。

### 方案三：快排分区函数

快排的分区函数：选择一个数，把数组的数分为两部分，把比选中的数小或者相等的数移到数组的左边，把比选中的数大的数移到数组的右边，返回分区后的选中数所在的下标。

---

```
int partition(int[] data, int start, int end){
    if(start>end) return -1;
    int index = start; //可以随机选择pivot，不一定是第一个元素
    //第一次交换
    int tmp = data[index];
    data[index] = data[end];
    data[end] = tmp;
    for(int i=start; i<end; i++){
        if(data[i]>= data[end]){
            if(i!=index){
                //第二次交换
                tmp = data[index];
                data[index] = data[i];
                data[i] = tmp;
            }
            index++;
        }
    }
    //第三次交换
    tmp = data[end];
    data[end] = data[index];
    data[index] = tmp;
    return index;
}
```

---

快排的分区函数广泛运用在排序和海量数据挑选当中，笔者希望面试者能做到 5 分钟之内默写出来。

对数组调用分区函数之后，如果返回的小标是  $k-1$ ，那么数组左边的  $k$  个数（数组小标从 0 到  $k-1$  时，有  $k$  个元素）就是最小的  $k$  个数（这  $k$  个数不一定是排了序的）。问题转化为不停调用 `partition` 函数，直至返回的下标为  $k-1$ 。这种方法的平均时间复杂度为  $O(n)$ ，因为它并未实现真正的快速排序算法。

---

```
void getTopK(int[] data, int k){
    int start = 0, end = data.length - 1;
    int index = partition(data, start, end);
    while(index != k - 1){
        if(index > k - 1){
            //从 index 前面重新找
            end = index - 1;
            index = partition(data, start, end);
        }else{
            //从 index 后面重新找
            start = index + 1;
            index = partition(data, start, end);
        }
    }
    for(int i = 0; i < k; i++){
        System.out.println(data[i] + "\t");
    }
}
```

---

### 扩展问题

一个直角坐标系上有  $N$  个点，找出离原点最近的  $k$  个点，请设计数据结构并计算时间复杂度。

如何把上述扩展问题转化为“Top K”问题，使它的时间复杂度为  $O(n)$ ？这取决于我们如何设计数据结构。为这些点设计一个数据结构，并含有到原点的距离。

---

```
class Point{
    double x;
    double y;
    double distance; //到原点的距离
    Point(double x, double y){
        this.x = x;
        this.y = y;
        distance = Math.sqrt(x*x+y*y);
    }
}
```

---

对于这些点的集合，调用分区函数（排序依据是点的 `distance` 大小），如果返回的小标是 `k-1`，那么前 `k` 个点就是离原点最近的 `k` 个点，把问题转化为“Top K”，因此，时间复杂度平均为  $O(n)$ 。

## 面试题 13：Top K II ☆☆☆☆

给定一维整型数组和一个整数 `k`，找出和不少于 `k` 的数目最少的子数组。

我们依然可以利用 `partition` 函数，把数组一分为二，并把此问题转化为“Top K”的问题。不过此时，我们需要对 `partition` 函数进行修改：1) 对数组进行降序排列，即大的数放在数组左边，小的数放在数组右边；2) 分区的同时计算左边数的和。

有了新的 `partition` 函数之后，我们可以循环调用该函数。把左半部分的和同 `k` 比较，如果大于等于 `k`，那么找到符合条件的分区，但不确定左半部分的子数组的元素个数是否是最少的，所以我们还需要继续调用 `partition` 函数；如果左半部分的和比 `k` 小，则相应更新 `k` 的值， $k = k - \text{左半部分的和}$ ，因为下次调用 `partition` 函数时，忽略左半部分的元素了。

---

```
void getTopKII(int[] data, int k){
    int start=0, end=data.length-1;
    int last=-1; //记录上次的子数组和不小于 k 的尾部元素下标
    //记录分区后的左半部分的和, java 无法传参, 我们使用 ArrayList 来代替
    ArrayList<Integer> currSum = new ArrayList<Integer>();
    int index = partition(data, start, end, currSum);
    while(index >= 0){
        if(currSum.get(0) >= k) {
            //记录当前下标, 因为我们找到了从 0 到 index 子数组
            //并且满足和不小于 k 的条件。
            last=index;
            //从 index 前面重新找
            end=index-1;
            index=partition(data, start, end, currSum);
        }else{
            //从 index 后面重新找
            start=index+1;
            //因为更新了 start, 忽略了新 start 之前的元素,
            //所以我们要相应调整 k
            k=k - currSum.get(0);
            index=partition(data, start, end, currSum);
        }
    }
    for(int i=0; i<= last; i++){
        System.out.println(data[i]+"\\t");
    }
}
```

---

## 面试题 14: 两数组第 k 个值 ☆☆☆☆☆

两个有序数组 A 和 B, 分别拥有 m 和 n 的长度, 求其合并后的第 k 个值。



考虑这个问题开始时会觉得很简单，但随着我们思考的深入，会觉得难以达到面试官的要求。让我们从简单的解法开始。

### 方案一：归并排序

把数组 A 和 B 分别看作排序数组的左半部分和右半部分，然后进行归并操作，从两个数组的头部开始取出第 k 个值。只移动元素较小的数组，总共移动了 k 步，因此，时间复杂度为  $O(k)$ 。

### 方案二：归并和二分查找

与其按顺序挑出下一个排序的元素，我们不如考虑通过二分查找，同时在 A 和 B 里找，时间复杂度为  $O(\log(m+n))$ 。感兴趣的读者可以试写一下实现代码。

### 方案三：二分操作

能否只在一个数组里使用二分查找呢？通过观察我们知道：1) 不需要得到前 k 个数，只要第 k 个数；2) 使用二分查找 A 时，不需要在 B 使用二分查找，而是通过各自的下标来计算该元素在合并之后的位置是否等于 k-1（因为数组下标从 0 开始，所以第 k 个元素的下标是 k-1）。

对于  $A[i]$  和  $B[j]$ ，我们得到了在他们之前的  $i+j$  个元素，如果包含这两个元素本身，那么我们有  $i+j+2$  个元素。假设  $\max(A[i-1], B[j-1]) < B[j] < A[i]$ ，我们无法判断  $A[i]$  是否为合并后的第  $i+j+2$  个元素，因为当前无法确定  $B[j+1]$  和  $A[i]$  的关系；但是，可以肯定  $B[j]$  是第  $i+j+1$  个元素，为什么？因为在他们之前有  $i+j$  个元素比  $A[i]$  和  $B[j]$  小，那么第  $i+j+1$  个小的元素必定在  $A[i]$  和  $B[j]$  之间产生，而  $A[i]$  大于  $B[j]$ ， $B[j]$  就是第  $i+j+1$  个小的元素。

回到问题本身，我们需要保证两个等式：

1) 分别在数组 A 和 B 找出 i 和 j, 使得  $i+j+1=k$ 。

2) 满足  $\max(A[i-1], B[j-1]) < B[j] < A[i]$ 。我们不能保证能找到第二个不等式, 因为不能保证 A 和 B 的元素是犬牙交错的。如果找不到相应的 A[i], 则说明 A 数组的元素过小, 这时候在满足第一个等式的情况下, 如果  $j \geq 0$ , 则取 B[j] 的值; 否则取 A[i-1] 的值。

---

```

int findKthSortedArrays(int[] A, int[] B, int k) {
    int m = A.length, n=B.length;
    if(m>n) return findKthSortedArrays(B, A, k);
    int left = 0, right = m;
    while(left < right) {
        //二分查找
        int mid = left + (right - left) / 2, j = k-1 - mid;
        if(j >= n || A[mid] < B[j]) {
            //小标 i 尽可能往前移动, 找到比 B[j] 大的第一个 A[i]
            left = mid + 1;
        } else {
            right = mid;
        }
    }
    int Aminus1 = left - 1 >= 0 ? A[left - 1] : Integer.MIN_VALUE;
    int Bj = k - 1 - left >= 0 ? B[k - 1 - left] : Integer.MIN_VALUE;
    //取 B[j] 和 A[i-1] 中的一个
    return Math.max(Aminus1, Bj);
}
    
```

---

由于我们只对较短的数组进行二分查找, 所以时间复杂度为  $O(\log(\min(m,n)))$ 。

## 面试题 15: 两数组中值 ☆☆☆☆☆

两个有序数组 A 和 B, 分别拥有 m 和 n 的长度, 求其合并后的中值。

中值的定义：有序数组长度为  $n$ ，如果  $n$  为奇数，则返回第  $n/2+1$  个元素；如果  $n$  为偶数，则返回第  $n/2$  个元素和  $n/2+1$  个元素的平均值。

对于长度为  $m$  的  $A$  和长度为  $n$  的  $B$ ，合并后长度为  $n+m$ 。假设  $k=(n+m-1)/2 + 1$ ，如果  $n+m$  为奇数，则中值为第  $k$  个元素；如果  $n+m$  是偶数，则中值为第  $k$  个元素和第  $k+1$  个元素的平均值。中值的问题可以转化为上一道题“两数组第  $k$  个值”。对于偶数情况，当我们找到第  $k$  个值之后，不需要再次调用 `findKthSortedArrays()`，而是从  $A[i]$  和  $B[j+1]$  中挑选一个较小的数，即为第  $k+1$  个数。

---

```
public double findMedianSortedArrays(int A[], int B[]) {
    if((A == null || A.length == 0) && (B == null || B.length == 0))
        return 0;
    int m=A.length, n=B.length;
    int k=(n+m-1)/2 + 1;
    return findKthSortedArrays (A, B, k);
}

double findKthSortedArrays (int[] A, int[] B, int k) {
    int m = A.length, n=B.length;
    if(m>n)return findKthSortedArrays(B, A, k);
    int left = 0, right = m;
    while(left < right) {
        int mid = left + (right - left) / 2, j = k-1 - mid;
        if(j >= n || A[mid] < B[j]) {
            left=mid + 1;
        }
        else {
            right=mid;
        }
    }
    int Aminus1 = left - 1 >= 0? A[left - 1] : Integer.MIN_VALUE;
    int Bj = k -1- left >= 0? B[k -1- left] : Integer.MIN_VALUE;
    int valK = Math.max(Aminus1,Bj);
    if( (n + m) % 2 == 1)
        return valK;
    //如果两个数组元素的总数是偶数，则我们取第 k 和第 k+1 个元素的均值
```

---

---

```
int Bjplus1 = k - left >= n? Integer.MAX_VALUE: B[k - left];
intAi = left >= m? Integer.MAX_VALUE: A[left];
return((double) (valK + Math.min(Bjplus1,Ai))) / 2.0;
}
```

---

## 面试题 16：旋转数组最小值 ☆☆☆

输入一个递增排序数组的一个旋转，输出旋转数组的最小值。

### 举例

数组{3,4,5,6,1,2}为{1,2,3,4,5,6}的一个旋转，该数组的最小值为 1。

即便是旋转数组，只要**局部有序**，就可以使用**二分查找**。我们把数组一分为二之后，需要判断排序部分在左半部分还是右半部分。当数组出现有相同值的元素时，比如，{1,1,1, 1,0,0,1}或者{1,0,0,1,1,1,1}，我们无法区分哪个部分是已经排序的，这时候只能把首部指针向前移动一位。每次查找时，需要提取最小值。

---

```
int getMinOfRotation(int A[]){
    int left=0, right=A.length-1, mid, min=A[left];
    while(left<right){
        mid=left+(right-left)/2;
        min = Math.min(A[left], min);
        if(A[mid]==A[left] && A[mid]==A[right]) {
            //不能确定最小值在哪一部分，向前移动一格
            left++;
        }else if(A[mid]>= A[left]){
            //在右半部分查找
            left=mid+1;
            min=Math.min(A[left], min);
        }else{
            //在左半部分查找

```

---

---

```
        min= Math.min(A[mid], min);
        right=mid-1;
    }
}
return min;
}
```

---

## 面试题 17：旋转数组搜索 ☆☆☆

给定一个有序数组的旋转和一个目标值，返回目标值在该数组的下标，如果不存在，那么返回-1。假设该数组没有重复的值。

同前一道题“旋转数组最小值”，只要是局部有序，哪怕是旋转的，我们依然能用二分查找法。通过中间元素和左右两端元素比较，判断哪一部分是有序的。

1) 如果左半部分是有序的，目标值落在有序区间里，那么移动右指针，因为无序的部分要么比中间元素大，要么比左端元素小；如果目标值不在有序区间里，则移动左指针。

2) 如果右半部分是有序的，目标值落在有序区间里，那么移动左指针，抛弃左半部分；如果目标值不在有序区间里，抛弃右半部分，则移动右指针。

---

```
int searchRotatedArray(int A[], int target) {
    int left = 0;
    int right = A.length - 1;
    while (left <= right) {
        int mid = left + (right-left) / 2;
        if(A[mid] == target) return mid;
        //左半部分是有序的
        if(A[left] <= A[mid]) {
            if(A[left] <= target && target < A[mid])
                right = mid - 1;
        }
    }
}
```

---

---

```
        else
            leftmid + 1;
    } else {
        // 右半部分是有序的
        if (A[mid] < target && target <= A[right])
            leftmid + 1;
        else
            rightmid - 1;
    }
}
return -1;
}
```

---

### 扩展问题

如果数组含有相同值的元素，上面的解法还有效吗？

我们还可以使用二分查找法。唯一的不同在于：数组含有相同值的元素，在左端元素、中间元素和右端元素值相等的情况下，无法判断出哪一部分是有序的。在这种情况下，只能把左指针向前移动一步。因此，在最坏的情况下时间复杂度是  $O(n)$ 。

## 面试题 18：首个正数 ☆☆☆☆

给定一个无序整型数组，找出第一个不在数组里的正整数。要求时间复杂度  $O(n)$ ，空间复杂度为  $O(1)$ 。

### 举例

输入数组 {5, 3, -1, 1}，返回 2。

## 观察

在上述例子中，返回的不是0，也不是4，而是2，其中2比数组长度4小。根据上面观察，返回结果肯定在1和数组的长度+1之间。那么存在  $O(n)$  算法吗？扫描一遍就已经是  $O(n)$  时间了，这自然而然地让我们想到哈希算法，但是哈希会导致额外的存储空间，这与题目要求的常数时间不符。那么试一试修改原有数组呢？

借用哈希算法的思路，使用数组下标来存储相应的值，比如第  $k$  个元素（对应的下标是  $k-1$ ）存储数字  $k$ ，也就说  $A[k-1]=k$ 。对于大于数组长度的数字或者小于1的数字，怎么处理？抛弃，因为他们对我们找第一个不在数组里的整数没有帮助。一旦有了新数组，就从头开始扫描，遇到第一个  $A[k-1]$  不等于  $k$  时，输出  $k$ 。如果没有遇到，那结果只能是数组长度的下一个数。比如数组  $\{1, 2, 3, 4\}$ ，返回5。

---

```
int firstMissingPositive(int A[]) {
    int n=A.length;
    for(int i=0; i<n; ++i) {
        if(A[i] > 0 && A[i] <= n) {
            if(A[i]-1 != i && A[A[i]-1] != A[i]) {
                //交换相应的值
                int temp = A[A[i]-1];
                A[A[i]-1] = A[i];
                A[temp] = A[i];
                //交换之后，还需要对新的 A[i] 值进行判断
                i--;
            }
        }
    }
    for(int i=0; i<n; ++i)
        //输出第一个不匹配的数字
        if(A[i]-1 != i) return i+1;
    //如果没找到，则输出长度+1 的数
    return n+1;
}
```

---

## 面试题 19：合并有序数组 ☆☆

给定两个有序数组 A 和 B，把 B 合并到 A 里，保持结果有序。假设 A 有足够的空间容纳 B 的元素。初始时，A 和 B 的元素各有 m 和 n 个。

数组插入新元素没有链表那么方便，因为插入一个新元素会导致插入位置后面的所有元素的移动。为了避免大量的移动，通常的办法就是从后到前插入元素，即反向插入和移动元素。

如何映射合并后的新数组下标和原有两个数组各自的下标呢？我们以原有数组 A 和 B 的最后一个元素为例，A[m-1]和 B[n-1]，新数组的最后一个元素的下标应该是 m+n-1，这个元素要么来自 A[m-1]，要么来自 B[n-1]。被选中数组的下标相应减去 1，准备下一轮的挑选。

---

```
void mergeTwoSortedArrays(int A[], int m, int B[], int n) {
    while(n>0){
        if(m==0 || A[m-1] < B[n-1])
            //如果取完了原有数组 A，或者当前 A 的元素小于当前 B 的元素
            //那么我们选取当前 B 的元素
            A[n+m-1] = B[--n];
        else
            A[n+m-1] = A[--m];
    }
}
```

---



## 面试题 20：三角形 ☆☆

给定一个正整数  $n$ ，产生 Pascal 三角形的前  $n$  行。

### 举例

比如，当  $n=5$  时，返回：

```
1
1,1
1,2,1
1,3,3,1
1,4,6,4,1
```

### 观察

在第二行以后，每行的首尾都是 1，中间的元素值是上一行相邻两个元素之和。比如，当  $n=5$  时， $A[4][0]=1$ ， $A[4][4]=1$ ， $A[4][1]=A[3][0]+A[3][1]$ ， $A[4][2]=A[3][1]+A[3][2]$ ， $A[4][3]=A[3][2]+A[3][3]$ 。

由此，我们可以通过前一行相邻的两个元素来计算当前行的值。

---

```
ArrayList<ArrayList<Integer>>pascalTriangle(int n) {
    ArrayList<ArrayList<Integer>>res =
        new ArrayList<ArrayList<Integer>>();
    if(n < 1) return res;
    //初始化第一行
    ArrayList<Integer>row1 = new ArrayList<Integer>();
    row1.add(1);
    res.add(row1);
}
```

---

---

```
        if(n==1)return res;
        for(int i = 1; i < n; ++i){
            ArrayList<Integer> row = new ArrayList<Integer>();
            row.add(1);
            for(int j = 1; j < i; ++j){
                //获取上一行的相邻两个元素之和
                row.add(
                    res.get(i-1).get(j-1)+res.get(i-1).get(j));
            }
            row.add(1);
            res.add(row);
        }
        return res;
    }
}
```

---

### 扩展问题

给定一个正整数  $n$ ，产生 Pascal 三角形的第  $n$  行。要求：只能使用  $O(n)$  的空间。比如，当  $n=5$  时，返回：[1, 4, 6, 4, 1]。

我们类似地可以通过已有的结果来计算新的一行，只不过题目要求只能利用  $O(n)$  的空间。由于不能使用额外空间，对输出结果进行修改，也就是说一边计算一边修改当前元素的值。

## 面试题 21：二维数组搜索 ☆☆☆

设计一个算法对  $m \times n$  矩阵进行搜索，这个矩阵拥有如下属性。

- 每行的数都是从左到右排序好的；
- 每行的首数大于上行的尾数。

### 举例

现有这样的 3×4 矩阵：

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

在矩阵查找 7，程序应返回 true。

对于排好序的数组我们考虑使用二分查找法，以获取  $O(\log n)$  的查找效率。符合上述两个属性的二维数组其实可以看作一维有序数组，在这个一维数组中，元素被分成  $m$  段，每段固定长度。接着，我们如何产生从二维到一维的映射规则呢？以上述题目为例，如果把二维数组展开之后， $\text{matrix}[1][2] = 7$  对应一维的下标就是  $1*4+2 = 6$ 。

假设映射到一维数组  $A$ ， $A$  有  $m*n$  个元素，那么  $A[k] = \text{matrix}[i][j]$ ，当  $k=i*n+j$ ，即  $i = k/n, j=k\%n$ 。有了映射关系之后，我们可以对展开之后的一维数组进行二分查找，时间复杂度为  $O(\log(m*n))$ 。

---

```
boolean searchMatrix(int[][] matrix, int target) {
    int m = matrix.length;
    int n = matrix[0].length;
    int low = 0;
    int high = m * n - 1; // 总共有 m*n 个元素
    while (low <= high) {
        int mid = (low + high) / 2;
        if (matrix[mid/n][mid%n] == target)
            // 找到此元素，返回 true
            return true;
        elseif (matrix[mid/n][mid%n] < target)
            // 在右半部分继续找
            low = mid + 1;
    }
}
```

---

---

```
        elseif (matrix[mid/n][mid%n] > target)
            //在左半部分继续找
            highmid = 1;
        }
        return false;
    }
}
```

---

## 面试题 22：区间搜索 ☆☆☆☆

给定一个有序数组（该数组元素可能含有相同值）和一个目标值，找出目标值的起止下标，如果数组不含有该目标值，则返回[-1,-1]。要求时间复杂度为  $O(\log(n))$ ， $n$  为数组长度。

### 举例

假定数组为 {1, 3, 4, 4, 5}，目标值为 4，那么返回[2, 3]。

### 观察

第一次接触这道题时，一个直观想法是使用二分查找，但是，不清楚如何使用二分查找来找出起止下标。假如目标值不在有序数组里，这种情况和普通二分查找一样，元素找不到，返回[-1,1]。如果目标值在数组里，而且可能有多个目标值，那么如何找到第一个和最后一个目标值的下标？

题目的要求和普通的二分查找法有些不同，并不是说找到目标值，返回其所在下标就可以了。因为数组存在重复的元素，目标值可能多次出现。分两步走：第一步，为了找到起点，先找到比目标值小但最靠近目标值的元素，该元素往后一位的元素值等于目标值，则就是我们要找的起点；第二步，同样的道理，如果起点能找到，那么为了找止点，先找到比目标值大但最靠近目标值的元素，该元素的前一个元素就是我们要找的止点。

---

```
int[] searchRange(int A[], int target) {
    int[] range = {-1, -1};
    int lower = 0, upper = A.length, mid;
    if (A[upper-1] < target)
        return range;
    // 搜索起点
    while (lower < upper) {
        mid = (lower + upper) / 2;
        if (A[mid] < target)
            // 尽可能往目标值靠近
            lower = mid + 1;
        else
            // 不使用 mid-1, 是因为要保留可能存在目标值的元素
            upper = mid;
    }
    // 如果目标值没找到, 则返回 [-1, -1]
    if (A[lower] != target)
        return range;
    range[0] = lower;
    // 搜索止点
    upper = A.length;
    while (lower < upper) {
        mid = (lower + upper) / 2;
        if (A[mid] > target)
            // 找出最靠近目标值但又比目标值大的元素
            upper = mid;
        else
            lower = mid + 1;
    }
    range[1] = upper - 1;
    return range;
}
```

---

该算法使用了两次二分查找法, 因此, 算法复杂度依然是  $O(\log n)$ 。

## 面试题 23：插入位置 ☆☆

给定一个有序数组和一个目标值，返回目标值的下标，如果目标值不在数组里，那么返回其插入数组后的下标，插入之后数组依然保持有序。假定数组不存在重复元素。

### 提问

应聘者：这道题也太简单了吧？

面试官：怎么说？

应聘者：从头到尾扫描一遍，找出第一个大于或等于目标值的元素。

面试官：这种算法复杂度是多少？

应聘者：嗯，是  $O(n)$ 。

面试官：有没有更快的办法？

### 举例

数组  $[1,2,3,4]$ ，目标值 3，则返回下标 2。

又如，数组  $[1,2,4,5]$ ，目标值 3，则返回下标 2。

### 观察

和上一道题一样，直观想法是使用二分查找法，但是不清楚如何修改二分查找来找出下标。假如目标值在有序数组里，这种情况和普通二分查找一样，返回当前下标即可。如果目标值不在数组里，那么如何找到插入下标？

通过二分查找找出目标值，如果能找到，则返回当前下标；如果不能，则返回二分查找区间开始 low，结束 high，还是两者中间 mid？我们知道循环调用二分查找时，low

总是小于等于 high，循环结束后 low 大于 high，low 记录了第一个大于目标值的元素的下标，即为答案。

---

```
int searchInsert(int A[], int target) {
    int low=0, high=A.length-1, mid;
    while(low <= high) {
        mid = (low+high)/2;
        if(A[mid] == target)
            return mid;
        else if (A[mid] > target)
            high=mid-1;
        else
            //往目标值靠近
            low=mid+1;
    }
    //返回第一个大于目标值的元素的下标
    return low;
}
```

---

上述解法是二分查找法，其算法复杂度为  $O(\log n)$ ，比从头到尾扫描要快。

## 面试题 24：矩阵清零 ☆☆☆

给定一个二维的  $m \times n$  矩阵，如果某个元素为 0，那么将其所在行和列的所有元素设为 0。不允许使用额外空间。

### 举例

左边为输入矩阵，右边为输出矩阵：

1	0	3	4	→	0	0	0	0
5	6	0	8		0	0	0	0
9	10	11	12		9	0	0	12

由于不允许使用额外空间，只能考虑矩阵本身，从里面挑出一行和一列来记录哪些行和列需要清零。

首先找出第一 0 元素，根据题目要求该元素所在的行和列需要清零，因此我们可以利用该元素所在的行和列记录需要清零的行和列。如果我们找到了第一个 0 元素，并且记下了该元素的下标，那么再进行一次扫描，记录需要清零的行和列。使用第一个 0 元素所在的行来记录需要清零的列，而使用其所在的列来记录需要清零的行。最后，根据第一个 0 元素所在的行和列记录情况，决定当前元素是否设置为零。

以上述矩阵为例，找出第一个 0 元素，即 `matrix[0][1]`，记录该元素的所在的行( $X=0$ )，和所在的列 ( $Y=1$ )。继续扫描所有矩阵元素，记录需要清零的行和列。

1 0		<u>0</u>	4
5 0	-	0 8	
9 10 11			12

最后，再从头扫描一遍矩阵，根据 X 行和 Y 列记录结果，将对应的元素设置为 0。

0 0 0			0
0 0 0			0
9 0 0 12			



---

```
void setZeroes(int[][] matrix){
    int m=matrix.length;
    if(m==0) return;
    int n=matrix[0].length;
    int x=-1, y=-1;
    for(int i=0; i<m; i++){
        for(int j=0; j<n; j++){
            if(matrix[i][j]==0){
                if(x==&& y==-1){
                    //找到第一个 0 元素
                    x=i;
                    y=j;
                }else{
                    //在第一个 0 元素所在行和列设置 0
                    matrix[x][j]=0;
                    matrix[i][y]=0;
                }
            }
        }
    }
    //如果找不到 0 元素，则直接返回
    if(x==-1 || y==-1) return;
    for(int i=0; i<m; i++){
        for(int j=0; j<n; j++){
            if(matrix[x][j]==0 || matrix[i][y]==0){
                //清零
                matrix[i][j]=0;
            }
        }
    }
}
```

---

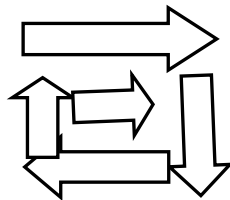
## 面试题 25：螺旋矩阵 ☆☆☆☆

给定一个  $m \times n$  矩阵，按螺旋顺序返回所有元素。

### 举例

有一个  $3 \times 3$  矩阵：

1 2		3
4 5 6		
7 8 9		



返回[1,2,3,6,9,8,7,4,5]

### 观察

从左到右、从上到下、从右到左和从下到上四个方向构建了一个循环。

针对这四个方向遍历，分别设置水平方向和垂直方向遍历的边界。每个方向遍历之后，需要更新相应的边界。当完成从左到右遍历时，因为已经完成当前上边界所在行的遍历，所以垂直方向的上边界往下移动一位，并且需要和下边界比较判断是否越界，如果越界，则退出循环；当完成从上到下遍历时，因为已经完成当前右边界所在列的遍历，所以水平方向的右边界往前移动一位，并且需要和左边界比较判断是否越界，如果越界，则退出循环；当完成从右到左遍历时，因为已经完成当前下边界所在行的遍历，所以垂直方向的下边界往上移动一位，并且需要和上边界比较判断是否越界，如果越界，则退出循环；当完成从下到上遍历时，因为已经完成当前左边界所在列的遍历，所以水平方

向的左边界往后移动一位，并且需要和右边界比较判断是否越界，如果越界，则退出循环。

---

```
ArrayList<Integer> spiralOrder(int[][] matrix) {
    ArrayList<Integer> result = new ArrayList<Integer>();
    if(matrix.length==0) return result;
    int beginX = 0, endX = matrix[0].length - 1; //水平方向
    int beginY = 0, endY = matrix.length - 1; //垂直方向
    while(true) {
        //外围从左到右
        for(int i = beginX; i <= endX; ++i)
            result.add(matrix[beginY][i]);
        if(++beginY > endY) break; //判断是否越界
        // 从上到下
        for(int i = beginY; i <= endY; ++i)
            result.add(matrix[i][endX]);
        if(beginX > --endX) break; //判断是否越界
        // 从右到左
        for(int i = endX; i >= beginX; --i)
            result.add(matrix[endY][i]);
        if(beginY > --endY) break; //判断是否越界
        // 从下到上
        for(int i = endY; i >= beginY; --i)
            result.add(matrix[i][beginX]);
        if(++beginX > endX) break; //判断是否越界
    }
    return result;
}
```

---

### 扩展问题

给定一个正整数  $n$ ，产生一个含有 1 到  $n^2$  数的螺旋矩阵。例如， $n=3$ ，返回矩阵：

1 2 3		
8 9 4		
7 6 5		

因为矩阵是  $N \times N$  的正方形，我们只需要两个边界：**begin**（合并了左边界和上边界），**end**（合并了右边界和下边界），而且不需要每次单个方向遍历完成时更新边界值，而在每次循环结束时，更新边界值即可。

# 第 7 章

## 链表

链表的面试题注意事项：

- 1) 链表的面试题相对简单，80%以上的题目可以采用递归和俩指针的方法来解决。
- 2) 当出现链表头部可能是空、被修改或被删除的情况下，我们往往需要使用一个 dummy 节点，dummy.next 指向头部，最终只要返回 dummy.next 即可。

除非特别说明，在本书中我们采用如下的链表结构：

---

```
class ListNode{
    int val;
    ListNode next;
    ListNode(int val) {
        this.val = val;
        this.next = null;
    }
}
```

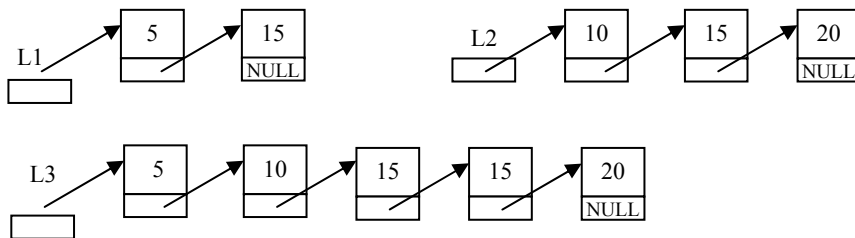
---

## 面试题 26：合并链表 ☆☆

合并两个有序链表，返回合并后的有序链表头。不允许利用额外的线性存储空间。

### 举例

两个有序链表 L1 和 L2 合并后得到有序链表 L3：



介于两个输入链表可能为空，所以我们采用 dummy 节点，使得 dummy.next 指向新链表的第一个节点。初始化时，使用两个指针指向两个输入链表的表头，在这两个指针不为空的情况下，取值较小的节点放入新链表的尾部，直至两个指针至少有一个为空。然后，将链表中不为空的部分（如果有的话）放到新链表的尾部。

```
ListNode mergeTwoLists(ListNode l1, ListNode l2) {  
    //创建 dummy 节点  
    ListNode dummy = new ListNode(0);  
    ListNode cur = dummy;  
    while (l1 != null && l2 != null) {  
        //取较小值的节点  
        if(l1.val <= l2.val) {  
            cur.next = l1;  
            l1 = l1.next;  
        }
```

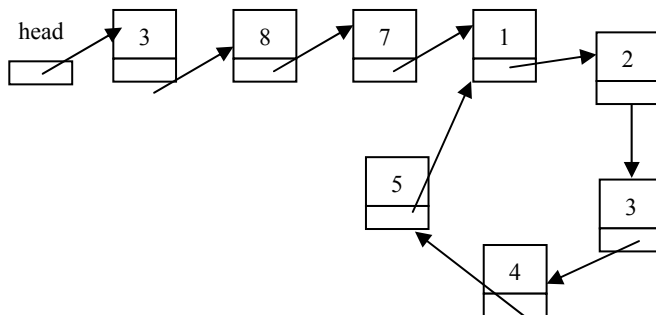
```
        }else {
            cur.next= l2;
            l2= l2.next;
        }
        cur= cur.next;
    }
    //将剩下的部分放置在合并后链表的尾部
    if (l1 != null) {
        cur.next= l1;
    }else if (l2 != null) {
        cur.next= l2;
    }
    return dummy.next;
}
```

## 面试题 27：环的长度 ☆☆☆

给出一个单向链表的头指针，如果有环，则返回环的长度，否则返回 0。

### 举例

如下的单向链表，其环的长度为 5。



### 观察

首先，我们需要判断链表是否有环。使用俩指针，一快一慢，如果快指针已经到达了尾部，而且他们没有相遇，那么可以判断链表无环。如果他们相遇，如何计算环的长度呢？

我们可以试试快指针移动速度是慢指针的 2 倍，初始化时慢指针指向链表头，快指针指向慢指针后两个节点。快与慢俩指针相遇时，快指针遍历 7→2→4→1，慢指针遍历 9→8→7→1，在节点 1 相遇。下一步，因为节点 1 肯定在环内，所以我们从这个节点 1 开始往前走，转了一圈肯定能回来，环的长度也很容易计算出来。

---

```
int getCircleLength(ListNode head){
    ListNode slow = head;
    if(slow==null || slow.next == null) return 0;
    ListNode fast = slow.next.next;
    //使用一快一慢指针
    while(fast!=null&& fast.next!=null){
        if(slow==fast)return getLength(slow);
        slow = slow.next;
        //快指针一次移动两步
        fast = fast.next.next;
    }
    return 0;
}

int getLength(ListNode node){
    int length = 1;
    ListNode curr = node;
    while(curr.next!= node){
        //转一圈，并且计算长度
        length++;
        curr = curr.next;
    }
    return length;
}
```

---



### 扩展问题

如何计算环的起点？快与慢指针相遇的节点就是环的起点吗？

快与慢指针相遇的节点一定在环内，但它不一定是环的第一个节点。我们有两种思路。

1) 哈希表：使用哈希表记录从表头开始遍历的节点，如果遇到已经存在的节点，即为环的起点。

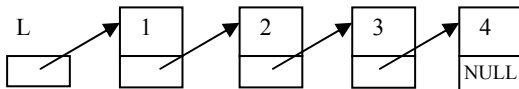
2) 将问题转化为两单向链表的交点：我们把快与慢指针相遇的节点记录为 P 点，在环上把 next 指针指向 P 点切开，这样我们有两个单向链表头，分别是原来链表头和 P 点，此问题转为求两单向链表的交点。

## 面试题 28：反转链表 ☆☆

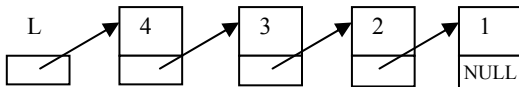
递归方法来实现反转链表。

### 举例

有一链表，含有节点 1 到 4：



其反转链表为：

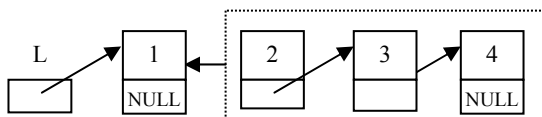


这道题是很多 IT 公司的必考题。我们可以通过递归和非递归方式来解答。

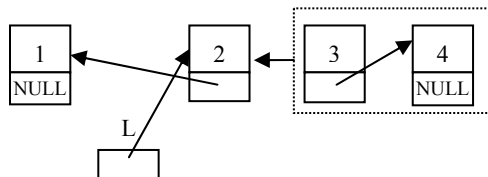
## 递归

在递归函数内除了递归调用函数以外，还需做两件事：1) 反转指针的指向；2) 返回新的链表头。以上述链表为例，

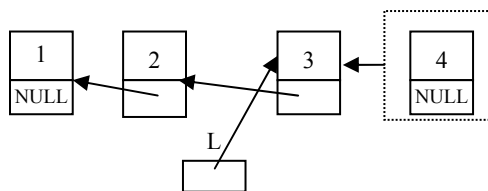
第一步：



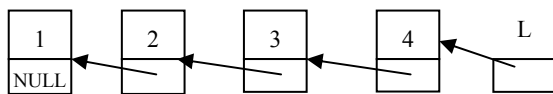
第二步：



第三步：



第四步：



---

```
ListNode reverseList(ListNode head, ListNode newHead){
    if(head == null || head.next == null){
        //递归结束条件判断
```

---

---

```
        newHead = head;
        return head;
    }
    //递归调用
    ListNode pre = reverseList(head.next, newHead);
    pre.next = head; //反转指针转向
    head.next = null;
    return head; //返回新链表头
}
```

---

### 非递归

使用 dummy 节点，简化当表头为空或者只含有单个节点的判断。初始化时，dummy.next 指向 head，并从第二个节点开始遍历，把遍历过的节点依次插入在 dummy 节点后。最后，返回 dummy.next 即可。

---

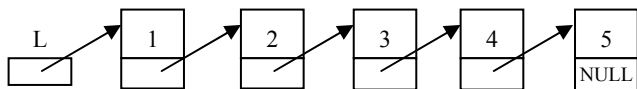
```
ListNode reverseList(ListNode head){
    ListNodedummy = new ListNode(0);
    dummy.next = head;
    if(head==null) return head;
    ListNode curr = head.next; //从第二个节点开始遍历
    head.next = null; //链表尾节点的 next 设为 null
    while(curr!= null){
        //依次插入遍历过的节点
        ListNodenext = curr.next;
        curr.next = dummy.next;
        dummy.next = curr;
        curr = next;
    }
    return dummy.next;
}
```

---

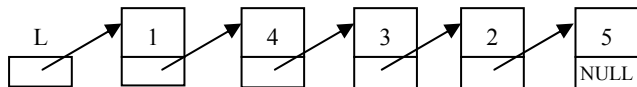
### 扩展问题

反转链表，从第 m 个元素到第 n 个元素。m 和 n 符合如下条件， $0 < m < n < \text{链表长度}$ 。

例如，输入：



当  $m=2$  和  $n=4$  时，输出：



其实该问题是要求在某段区间进行链表转置。为了更好地处理表头和第  $m$  个节点的关系，我们引入 **dummy** 节点，此外，还需记录第  $m-1$  个节点。从第  $m$  个节点开始遍历至第  $n$  个节点，已经将遍历过的节点插入在第  $m-1$  个节点后，并保证第  $m$  个节点的 **next** 指向遍历节点的 **next**，以避免链表断裂。

---

```

ListNode reverseBetween(ListNode head, int m, int n) {
    ListNodedummy = new ListNode(0);
    dummy.next = head;
    ListNodemNode = new ListNode(0);
    ListNode pre = dummy, curr = head, next;
    for (int i = 1; i <= n; ++i) {
        if (i == m) mNode = curr; //记录第 m 个节点
        if (i < m) pre = pre.next; //记录第 m-1 个节点
        next = curr.next;
        if (i > m && i <= n) {
            mNode.next = next; //以避免链表断裂
            curr.next = pre.next;
            pre.next = curr;
        }
        curr = next;
    }
    return dummy.next;
}

```

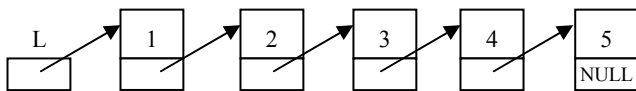
---

## 面试题 29：分组反转链表 ☆☆☆☆

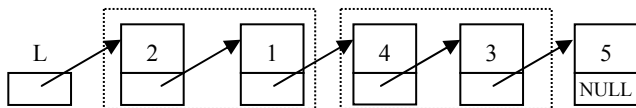
反转一个链表中的每  $K$  个节点，如果链表长度不能被  $K$  整除，则最后余数节点保持不变。

### 举例

给定一个链表：



当  $K=2$  时，返回：



如果熟悉链表转置，这道题不算难。首先，对链表进行分组，表头到第  $K$  个节点为第一组，第  $K+1$  个节点至第  $2K$  个节点为第二组，依次类推。我们可以发现分组的规律：当  $i\%k=1$  时，从第  $i$  个节点到第  $i+k-1$  个节点为一组。

---

```
ListNode reverseKGroup(ListNode head, int k) {  
    if (head == null || k <= 1) return head;  
    ListNode dummy = new ListNode(0);  
    dummy.next = head;  
    //记录分组区间的开始节点  
    ListNode pre = dummy, cross = head;  
    int count = 0;  
    while (cross != null) {
```

---

---

```
        count++;
        if(count % k == 0) {
            //分组区间结束节点
            prereverse(pre, cross.next);
            crosspre.next;
        } else {
            crosscross.next;
        }
    }
    returndummy.next;
}
```

---

其次对组内的节点进行转置。为了方便转置，我们借鉴了 dummy 节点用途，传入了这个分组区间的前一个节点和后一个节点。

---

```
ListNode reverse(ListNode pre, ListNode next) {
    ListNoddlast = pre.next;
    ListNodecur = last.next;
    while(cur != next) {
        last.next cur.next;
        cur.next pre.next;
        //在区间头部插入遍历的节点
        pre.next cur;
        cur last.next;
    }
    //返回区间转置前的第一个节点，作为下一个区间的前一个节点
    returllast;
}
```

---

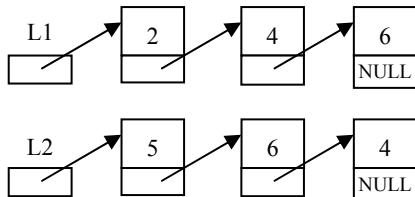
## 面试题 30：两数相加 ☆☆☆

给出两个正整数，每个整数由一个链表表示。其中链表的每个节点保存数的数位，这些数位反序存在在链表中，即高位在最后，低位在最前。返回一个链表代表这两个数

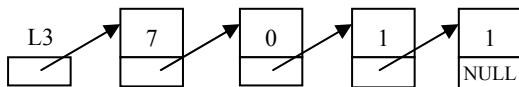
之和。

### 举例

输入链表 L1 和 L2，分别代表 642 和 465：



输出链表 L3，代表两数之和 1107：



我们注意到每个节点代表一个数位，节点的值在 0 到 9 之间，两个节点的和可能超过 9，需要有个变量记录可能的进位。由于数位反序存储在链表之中，从两个链表头开始顺序相加即可。同时注意到两个输入链表的长度可能不一样，如何相加一个链表的节点而另一个链表没有剩余节点呢？

这里有个巧妙的办法：只要链表还有未处理的节点，或者只要还有进位（即进位不为 0），我们就进行节点相加；对于当前指针指向空节点时，我们使用 0 代表其值，并且使用 NULL 代表其下一轮迭代节点。

---

```
public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
    ListNode iter1 = l1, iter2 = l2;
    ListNode list = null, tail = null;
    int carry = 0;
    //只要有未处理的节点或者进位不为 0，循环迭代
    while (iter1 != null || iter2 != null || carry != 0) {
        int num1 = iter1 == null ? 0 : iter1.val; //使用 0 代表 null 值
```

---

---

```
int num2 = iter2 == null ? 0 : iter2.val;
int sum = num1 + num2 + carry;
carry = sum / 10; //求出进位
sum = sum % 10;
if(list == null) {
    //记录新的链表头
    list = new ListNode(sum);
    tail = list;
} else {
    tail.next = new ListNode(sum);
    tail = tail.next;
}
//可以使用 null 代表 null 的下一轮迭代
iter1 = iter1 == null ? null : iter1.next;
iter2 = iter2 == null ? null : iter2.next;
}
return list;
}
```

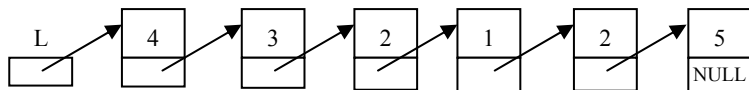
---

## 面试题 31：链表分区 ☆☆☆

给定一个链表和一个值  $x$ ，把链表一分为二，即将小于  $x$  的所有节点放在大于或等于  $x$  的节点前，并保存节点的原有相对顺序。

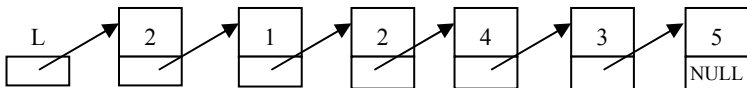
### 举例

输入链表：



当  $x=3$  时，输出：





根据  $x$  把链表一分为二，前半部分是小于  $x$  的节点，后半部分是大于或等于  $x$  的节点。为了便于处理，我们使用两个 **dummy** 节点，分别指向两部分链表的表头。从头开始遍历原始链表，当遇到小于  $x$  的节点时，将其附加上半部分的尾部；当遇到大于或等于  $x$  的节点时，将其附加上后半部分的尾部。遍历完成时，利用第二个 **dummy** 指向的下一个节点，把前半部分和后半部分连接起来，程序返回第一个 **dummy** 节点指向的下一个节点即为新链表头。

---

```
public ListNode partitionLinkedList(ListNode head, int x) {
    ListNode dummy = new ListNode(0); //第一部分的链表头
    ListNode pivot = new ListNode(x); //第二部分的链表头
    ListNode first = dummy, second = pivot, curr = head;
    while (curr != null) {
        ListNode next = curr.next; //保存下一个节点，避免中断遍历
        if (curr.val < x) {
            //放置小于 x 的节点
            first.next = curr;
            first = curr;
        } else {
            //放置大于等于 x 的节点
            second.next = curr;
            second = curr;
            second.next = null;
        }
        curr = next;
    }
    //前半部分同后半部分连接起来
    first.next = pivot.next;
    return dummy.next;
}
```

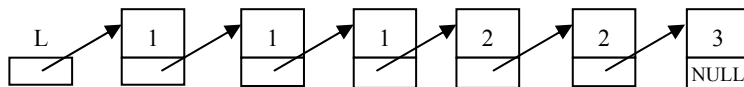
---

## 面试题 32：链表去重 ☆

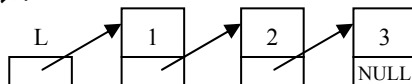
输入一个有序链表，删除重复值，确保每个值只出现一次。

### 举例

输入一个有序链表：



输出非重复的有序链表：



使用两个指针：第一个指针从头扫描原始链表，而第二个指针用于产生新的链表。第二个指针随着第一个指针移动而移动。当第二个指针遇到相同节点时，不移动，其 next 指向第一个指针的 next，也就是说忽略了相同节点；若遇到不同值的节点时，随着第一个指针移动。

```
public ListNode deleteDuplicates(ListNode head) {  
    if(head==null) return head;  
    ListNode prev=head; //保存非重复元素的列表  
    ListNode curr=head.next;  
    //开始扫描原有链表  
    while(curr!=null){  
        //如果值不同，两个指针同时向后移动，prev 沿着 curr 移动  
        //如果有相同的值，只移动 curr。  
        if(curr.val!=prev.val) prev=prev.next;  
        curr=curr.next;  
    }  
}
```

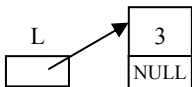
---

```
        prev.next=curr;
    }
    return head;
}
```

---

### 扩展问题

给出一个有序链表，删除所有重复元素，只保留不同值的节点。以上述输入链表为例，此时输出：



在结果的链表里，删除了所有重复元素，因此我们需要做些修改：1) 链表头可能会被删除，我们将使用 **dummy** 节点，这样会让删除操作更直观。2) 如果当前元素没有重复的，则加入结果的列表里。如何判断当前元素没有重复呢？使用一个指针记录当前元素第一次出现的节点，然后第二个指针遍历具有相同元素的节点，遍历完之后，比较第一个指针和第二个指针指向的节点，如果他们指向同一个节点，则意味着该元素没有重复，因为元素链表是有序的，拥有相同元素的节点是紧挨着的。

---

```
public ListNode deleteDuplicates2(ListNode head) {
    ListNodedummy = new ListNode(0);
    ListNode tail=dummy; //用来保存结果
    ListNode pre=head, curr=head;
    while(curr != null && curr.next != null){
        //移动当前指针到重复元素的最后一个
        while(curr.next!= null && curr.val == curr.next.val)
            curr= curr.next;
        if(pre==curr){
            //如果当前元素没有重复的，那么加入当前元素
            tail.next=pre;
            tail=tail.next;
        }
        pre=curr.next;
    }
}
```

---

---

```
        curr=curr.next;
    }
    tail.next=curr;//加入最后一个元素
    returndummy.next;
}
```

---

# 第 8 章

## 树

关于树的面试题注意事项：

1) 45 分钟的编程面试不会出现过于复杂的树结构，比如 B 树、B+树、红黑树等，但是应聘者应该去了解这些复杂的树结构。

2) 编程面试也难得出现关于图的题目，因为很难在短时间内完成编码。

3) 树的面试题通常可以使用深度优先遍历，广度优先遍历、栈，以及递归的方法来解决。

4) 二叉搜索树，即二叉查找树，是面试中常见的考察对象，也经常被要求和哈希表做比较，比如比较两者优劣和应用场景等。

除非特别说明，本书中对树的节点定义如下：

---

```
class TreeNode{  
    int val;  
    TreeNode left;
```

---

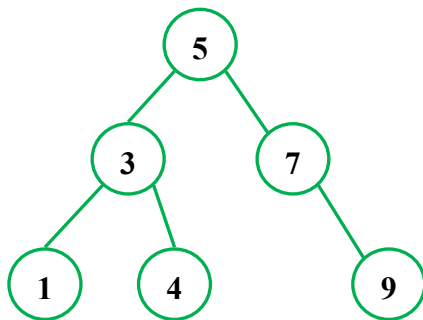
```
TreeNode right;  
TreeNode(int value){  
    val = value;  
    left = null;  
    right = null;  
}
```

### 面试题 33：二叉搜索树转为双向链表 ☆☆☆☆

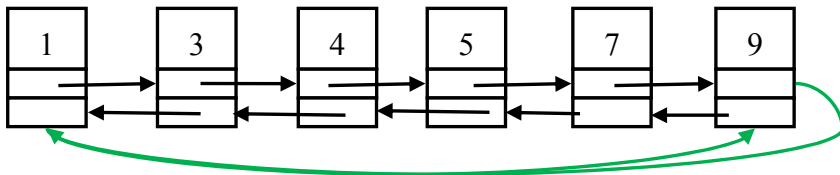
将一棵二叉搜索树转化为一个有序的循环双向链表。

#### 举例

输入一颗二叉搜索树：



输出一个有序的循环双向链表：



## 观察

如果二叉搜索树转为有序数组而不是链表，那么只需要中序遍历二叉搜索树就可以了。因此，通过中序遍历很容易充实双向链表指向后一节点的指针。如何获取指向前一个节点的指针呢？中序遍历时，我们需要记录节点的前一个指针，即中序遍历的前一个节点。

修改中序遍历过程，以获取双向链表节点的两个指针：

1) 记录头指针以便返回新的链表。

2) 每次有新的节点加入时，头指针需要和该节点建立相互指向关系，以保证链表的首尾相接。

3) 为了充实双向链表指向后一节点的指针，我们还需要一个 prev 指针记录当前的位置。这个 prev 指针随着节点遍历移动，作为下一轮遍历节点的前一个节点。

---

```
TreeNode treeToDoublyList(TreeNode root) {
    TreeNode prev = null;
    TreeNode head = null;
    treeToDoublyList(root, prev, head);
    return head;
}
//实现函数
void treeToDoublyList(TreeNode p, TreeNode prev, TreeNode head) {
    if(!p) return;
    treeToDoublyList(p.left, prev, head);
    // 当前节点的左节点指向前一个节点
    p.left = prev;
    if(prev)
        prev.right = p; // 前一个节点的右节点指向当前节点
    else
        head = p; // 如果 prev 未被初始化，那么它是链表的头
    p.right = p.right;
}
```

---

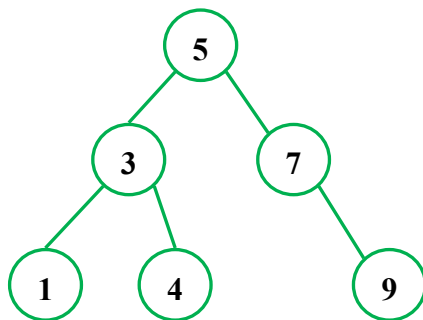
```
//将首尾相接，建立循环关系
head.left = p;
p.right= head;
//进入下一个节点之前，更新 prev 节点
prev= p;
treeToDoublyList(right,prev, head);
}
```

## 面试题 34：最小公共祖先 I ☆☆

给定两个节点，求它们在一棵二叉搜索树中的最小公共祖先。

### 举例

一棵二叉搜索树：



节点 1 和节点 9 的最小公共祖先是树根节点 5；而节点 1 和节点 4 的最小公共祖先是节点 3。

### 观察

二叉搜索树特点是节点的值大于其左子树的值，而小于其右子树的值。因此两个节



点的最小公共祖先的值一定介于这两个节点的值之间。

利用二叉搜索树的特点,通过前序遍历二叉树来判断最小公共祖先出现在左子树、右子树,还是节点本身。如何证明通过前序遍历找出最小公共祖先是正确的?我们利用反证法: $r$ 是通过前序遍历算出的 $p$ 和 $q$ 的最小公共祖先( $p$ 和 $q$ 分别在 $r$ 的左右子树里,即 $p < r < q$ ),假设存在 $r'$ 是 $p$ 和 $q$ 的最小公共祖先,而且 $r'$ 比 $r$ 更靠近 $p$ 和 $q$ ,则意味着 $r'$ 在 $r$ 的子树中,而且 $p$ 和 $q$ 分别在 $r'$ 的左右子树里。假设 $r'$ 在 $r$ 的左子树里,那么 $p < r' < q < r$ ,这与前面的 $q > r$ 矛盾,所以不可能出现有比 $r$ 更靠近 $p$ 和 $q$ 的最小公共祖先。

---

```
TreeNode LCA(TreeNode root, TreeNode p, TreeNode q){
    if(root==null || p==null || q==null) return null;
    if(root.val > p.val && root.val > q.val){
        //最小公共祖先在左子树
        return LCA(root.left, p, q);
    }else if(root.val < p.val && root.val < q.val){
        //最小公共祖先在右子树
        return LCA(root.right, p, q);
    }else{
        //返回节点本身
        return root;
    }
}
```

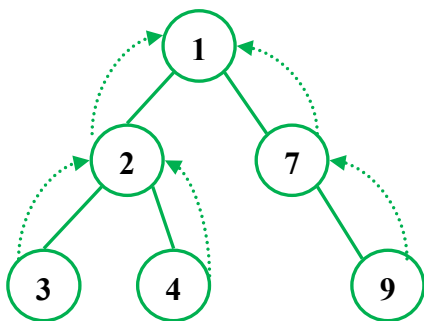
---

## 面试题 35: 最小公共祖先 II ☆☆☆

给定两个节点,求它们在一棵二叉树中的最小公共祖先。每个节点除了有左右节点以外,还有一个指向其父节点的指针。

### 举例

一棵二叉搜索树：



节点 3 和节点 9 的最小公共祖先是树根节点 1；而节点 2 和节点 4 的最小公共祖先是节点 2。

### 观察

根据每个节点由指向其父节点的指针获取从节点到树根的路径。有了两条路径之后，求出这两条路径的第一个相同的节点，即为最小公共祖先，解法类似于求两链表的相交点。

首先，判断在二叉树里是否存在这两个节点，如果有则返回这两个节点在树里对应的节点。有了这两个节点之后，根据其指向父节点的指针，求出到根节点的路径。类似于求两链表的相交点。算出两条路径的长度，如果长度不一致，砍掉长度较长的路径的前面部分，以保证两条路径长度相同，然后同步走，同时比较两边的值，值相同的即为最小公共祖先。还有一种办法是使用哈希表，把一条路径的所有节点放入哈希表，然后扫描第二条路径，如果能在哈希表中找到当前节点，则该节点为最小公共祖先。

---

```
TreeNode LCA2(TreeNode root, TreeNode p, TreeNode q){  
    if(root==null || p==null || q==null) return null;  
    // 用 pq.left pq.right 分别存储 p 与 q 在二叉树中对应的节点
```

---

---

```
    TreeNode pq = new TreeNode(0);
//    找出 p 和 q 在二叉树对应的节点
    help(root, p, q, pq);
    TreeNode pp = pq.left, qq = pq.right;
//    如果二叉树没有 p 或 q, 则返回 null
    if( pp==null || qq==null) return null
    int lenp = 0, lenq=0;
    TreeNode up1=pp, up2=qq;
//    计算各种到顶的长度
    while(up1 != root){ up1=up1.parent; lenp++;}
    while(up2 != root){ up2=up2.parent; lenq++;}
up1=pp;
up2=qq;
//    借鉴求两链表相交节点的做法, 先同步到长度相同的情况
while(lenp>lenq){
    up1=up1.parent;
    lenp--;
}
while(lenq>lenp){
    up2=up2.parent;
    lenq--;
}
//    往上爬, 直到它们相遇, 相遇的点就是最小公共祖先
while(up1!=up2){
    up1=up1.parent;
    up2=up2.parent;
}
return    up1;
}

void help(TreeNode root, TreeNode p, TreeNode q, TreeNode pq){
    if(root==null)    return;
//    遍历二叉树, 找出 p 和 q
    if(root == p)    pq.left=root;
    if(root == q)    pq.right=root;
    if(pq.left==null || pq.right==null){
        help(root.left, p, q, pq);
```

---

---

```
        help(root.right, p, q, pq);  
    }  
}
```

---

## 面试题 36：最小公共祖先 III ☆☆☆☆

给定两个节点，求它们在一个二叉树中的最小公共祖先。

### 提问

应聘者：输入的二叉树是二叉搜索树吗？

面试官：不是。

应聘者：给定的两个节点在输入的二叉树里？

面试官：是的。

应聘者：在给定的两个节点 A 和 B 中，其中节点 A 在另一个节点 B 的子树里，那么我可以理解为它们的最小公共祖先为 B。

面试官：对的。

第一次被问到这个问题时，可能没什么头绪。遍历二叉树通常有三种方式：前序、中序和后序。其实，还有两种：前后序和中后序，分别结合两种遍历的方式。比如，前后序，在递归函数体内，先处理当前节点，然后对其左节点和右节点分别递归调用函数，最后结合其左右子树的结果以及当前节点的处理结果，返回最终结果。

在本题中，我们先判断当前节点是否为两个给定节点 p 和 q 之一，如果是，直接返回当前节点，即为最小公共祖先。否则，递归对其左右节点调用函数，然后根据左右子树的结果，返回恰当的值：如果左右子树返回的最小公共祖先均不为空，说明 p 和 q 分别在当前节点的两边，则返回当前节点；如果一个不为空，则返回不为空的结果；如果左右子树返回均为空的结果，则函数最后也返回空。

```
TreeNode LCA3(TreeNode root, TreeNode p, TreeNode q) {  
    if (root == null || p == null || q == null) return null;  
    if (root == p || root == q) return root; //返回当前节点  
    TreeNode left = LCA3(root.left, p, q);  
    TreeNode right = LCA3(root.right, p, q);  
    //p 和 q 分别在两边，则返回当前节点  
    if (left != null && right != null) return root;  
    //如果 p 或 q 在单边，则返回单边节点  
    return left == null ? right : left;  
}
```

## 面试题 37：最小公共祖先 IV ☆☆☆☆

给定两个节点，求它们在一个普通树中的最小公共祖先。

### 观察

借鉴“最小公共祖先 II”解题思路，通过某种遍历方式分别求出从两个节点到树根的路径。当获取两条路径之和，把问题转为求两条链表的相交点。在没有指向父节点指针的情况下，如何获取到树根的路径呢？

当前的任务是求出节点到树根的路径。我们采用广度优先遍历的方法找到节点，其好处是更方便记录节点和子节点的关系。使用一张哈希表记录每个节点的父节点。当我们找到给定的两个节点时，通过哈希表回溯找出到树根的路径。

先定义树的节点：

```
class TreeNode{  
    int val;  
    ArrayList<TreeNode> children;  
}
```

整个流程是这样的：首先，通过广度优先遍历找出节点并且通过一张哈希表记录每个节点的父节点。然后，回溯哈希表的内容，求出到树根的路径。最后，比较两条路径，求出第一个相同的节点，即所求的最小公共祖先。

---

```

TreeNode LCA4(TreeNode root, TreeNode p, TreeNode q) {
    if(root == null || p == null || q==null ) return null;
    Queue<TreeNode> currLevel = new LinkedList<TreeNode>();
    Queue<TreeNode> nextLevel = new LinkedList<TreeNode>();
    currLevel.offer(root);
    //使用队列记录到树根的路径
    Queue<TreeNode> pp = new LinkedList<TreeNode>();
    Queue<TreeNode> qq = new LinkedList<TreeNode>();
    //使用哈希表存储每个节点的父节点
    HashMap<TreeNode, TreeNode> backTracking =
        new HashMap<TreeNode, TreeNode>();
    //通过非递归方式，广度优先遍历整棵树
    while(!currLevel.isEmpty()){
        while(!currLevel.isEmpty()){
            nextLevel = new LinkedList<TreeNode>();
            TreeNode node = currLevel.poll();
            for(TreeNode child: node.children){
                //记录父节点
                backTracking.put(child, node);
                if(child == p){
                    //求出从 p 到树根的路径
                    addParent(pp, backTracking);
                }else if(child == q){
                    //求出从 q 到树根的路径
                    addParent(qq, backTracking);
                }
            }
            if(!pp.isEmpty() && !qq.isEmpty()){
                //求出两条路径的第一个交点
                return LCA(pp, qq);
            }else{
                nextLevel.offer(child);
            }
        }
    }
}

```

---

---

```
        }
    }
    currLevel = nextLevel;
}
return null;
}

void addParent(Queue<TreeNode> que, TreeNode q,
    HashMap<TreeNode, TreeNode> bt){
    TreeNode parent = bt.get(q);
    while(parent != null){
        //回溯读取哈希表
        que.addFirst(parent);
        parent = bt.get(parent);
    }
}

TreeNode getLCA(Queue<TreeNode> pp, Queue<TreeNode> qq){
    TreeNode result = null;
    while(!pp.isEmpty() && !qq.isEmpty()){
        //从树根开始比较
        TreeNode pParent = pp.poll();
        TreeNode qParent = qq.poll();
        if(pParent == qParent){
            //每次记录相同的值
            result = pParent;
        }else{
            break;
        }
    }
    //最后一个相同的值即为最小公共祖先
    return result;
}
```

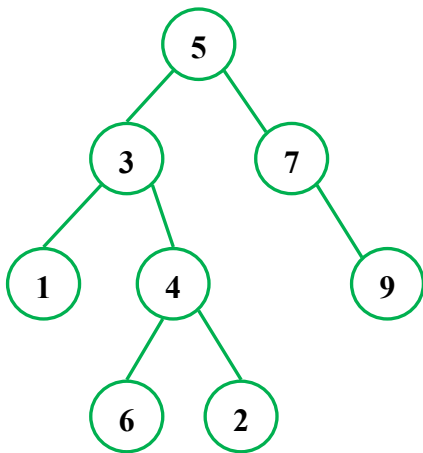
---

## 面试题 38：路径和 I ☆☆

给定一棵二叉树和一个值，判断是否存在从根到叶子节点的路径和等于给定的值。

### 举例

给出一棵二叉树和一个值 18：



我们能找出从根到叶子的路径：5 → 3 → 4 → 6，其和为 18。

### 观察

采用递归方式，如果当前节点是非叶子节点，那么递归判断其子节点开始的路径和是否有新的值，即目标值减去当前节点值后的数。

采用递归会让这个问题变得很简单。从树根开始判断：1) 如果当前是叶子节点，



而且节点值等于剩余和，那么程序返回真；2) 如果当前节点为空，则返回 False；3) 除了以上两种情况，返回递归调用左右子节点的结果或值，并且传入剩余和扣除当前节点值后的值。

---

```
public boolean hasPathSum(TreeNode root, int sum) {
    if (root == null) // 节点为空
        return false;
    if (root.left == null && root.right == null && root.val == sum)
        // 叶节点并且当前节点等于剩余的和
        return true;
    else
        // 递归调用左右子节点
        return hasPathSum(root.left, sum - root.val) ||
            hasPathSum(root.right, sum - root.val);
}
```

---

## 面试题 39：路径和 II ☆☆☆☆

输入一棵二叉树和一个整数，打印出二叉树中节点值的和为输入整数的所有路径。该路径可以从任意节点到叶节点。

与上一道“路径和”不同的是路径可以从任意节点到叶节点，而不仅仅是从根节点到叶节点。通过修改传入路径和的参数值，看似行不通了。为了求出任意节点到叶节点的路径和，我们需要做如下调整：

- 1) 前序遍历二叉树，记录从根节点到叶节点的路径；
- 2) 有了上述路径后，从叶节点开始累计路径和，同时与给定的值比较，如果相等，则输出该路径；
- 3) 如果遇到非叶节点，则递归调用其左右子节点。

---

```
void findPathSum2(TreeNode root, int sum){
    ArrayList<Integer> path = new ArrayList<Integer>();
    ArrayList<ArrayList<Integer>> res =
        new ArrayList<ArrayList<Integer>>();
    //调用递归函数
    findPathSumHelper(root, sum, path, res, 0);
}

void findPathSumHelper(TreeNode root, int sum, ArrayList<Integer> path,
    ArrayList<ArrayList<Integer>> res, int level) {
    if (root == null) return;
    path.add(root.data); //将节点加入路径
    if(root.left == null & root.right == null){
        //叶节点
        int tmp = root;
        for (int i = level; i >= 0; i--){
            tmp = path.get(i);
            if (tmp == 0) copyList(buffer, i, level, res);
        }
    }else{
        //递归调用左右子节点
        findSum(head.left, sum, path, level + 1);
        findSum(head.right, sum, path, level + 1);
    }
    //路径回退
    path.remove(path.size()-1);
}

//复制路径
private void copyList(ArrayList<Integer> path, int from, int end,
    ArrayList<ArrayList<Integer>> res){
    ArrayList<Integer> subPath = new ArrayList<Integer>();
    for(int i=from; i<=end; i++){
        subPath.add(path.get(i));
    }
    res.add(subPath);
}
}
```

---

## 面试题 40：平衡二叉树 ☆☆☆☆

给出一棵二叉树，判断其是否为平衡二叉树。

### 观察

先看看平衡二叉树的定义。平衡二叉树或者是一棵空树，或者是具有下列性质的二叉树。它的左子树和右子树都是平衡二叉树，且左子树和右子树高度之差的绝对值不超过1。看来递归判断左右子树是否是平衡二叉树就可以了，但是，有没有更好的办法呢？

根据平衡二叉树的定义，我们很容易想出一个直观的办法：递归判断左右子树是否是平衡二叉树，每次判断均比较其左右子树的高度。这种直观的办法导致了大量的重复计算。其实，我们可以从两方面进行优化：一方面，判断子树是否为平衡二叉树时，返回子树的高度，避免计算父节点为根的二叉树高度时，重复计算子树的高度。另一方面，当子树确定不是平衡二叉树时，返回-1，此时，我们认为这棵二叉树不是平衡二叉树。

---

```
public boolean isBalanced(TreeNode root) {
    //返回值为-1，即非平衡二叉树
    return (getHeight(root) != -1);
}

//求出子树高度和是否为平衡二叉树的信息
private int getHeight(TreeNode root) {
    if(root == null) return 0;
    int leftHeight = getHeight(root.left);
    if(leftHeight == -1) return -1;
    int rightHeight = getHeight(root.right);
    if(rightHeight == -1) return -1;
    if(Math.abs(leftHeight-rightHeight) > 1)
        return -1;
```

---

```
return l + Math.max(leftHeight, rightHeight);
```

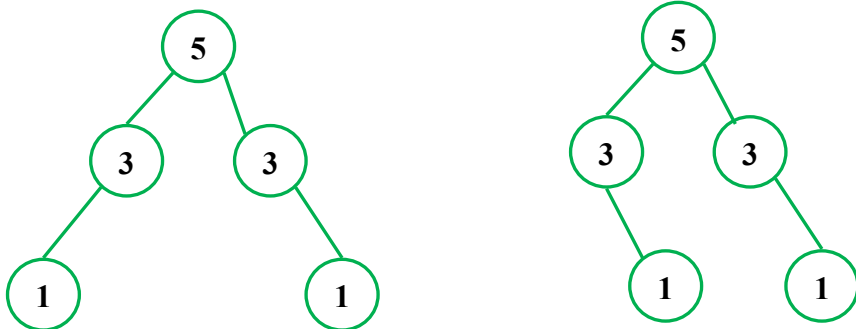
```
}
```

## 面试题 41：树的镜像 ☆☆

给出一棵二叉树，使用非递归的方法判断是否为其自身的镜像。

### 举例

例如，左边的树是其自身的镜像，而右边不是。



### 观察

我们发现只要能保证根节点下的左子树的每一层是右子树的镜像，就可以说明此树是其自身的镜像。

我们利用两个队列来分别记录左右子树每一层的节点。在广度优先遍历左右子树时，为了方便比较左右两边的节点，我们也把空节点加入到队列中。针对两边镜像节点的情况，必须处理如下不同情况：

- 1) 如果两边节点均为空，我们将忽略这种情况；

2) 如果一边为空而另一边不为空, 则程序返回 false;

3) 如果两者均不为空, 但值不相同, 则程序返回 false;

4) 除了以上的三种情况, 第四种情况为把左边的节点的子节点按照从左到右顺序加入左边的队列尾部, 而把右边的节点的子节点按照从右到左顺序加入右边的队列尾部。其理由是方便我们用从头部到尾部的顺序比较两个队列是否为镜像关系。

在广度优先遍历完成之后, 我们还需判断两个队列是否均为空, 只有均为空才返回真。

---

```
boolean isSymmetric(TreeNode root) {
    if(root == null) return true;
    //使用两个队列分别记录左右子树的每一层节点
    Queue<TreeNode> left = new LinkedList<TreeNode>();
    Queue<TreeNode> right = new LinkedList<TreeNode>();
    left.add(root.left);
    right.add(root.right);
    while(!left.isEmpty() && !right.isEmpty()) {
        //将两个队列的头部出列
        TreeNode l = left.poll();
        TreeNode r = right.poll();
        if(l == null && r == null) continue; //忽略两者为空的情况
        if(l == null || r == null) return false; //一方为空, 返回 false
        if(l.val != r.val) return false;
        left.add(l.left); left.add(l.right);
        right.add(r.right); right.add(r.left); //反过来插入
    }
    //遍历完成之后, 比较两队列的元素个数
    if(left.isEmpty() && right.isEmpty()) return true;
    else return false;
}
```

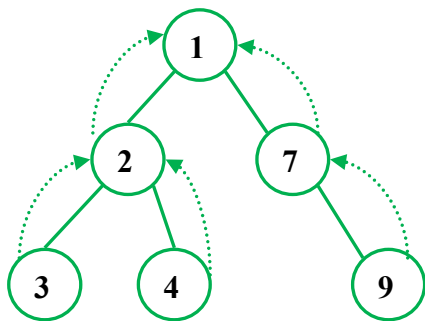
---

## 面试题 42：中序下个节点 ☆☆☆

给出一棵二叉树里的一个节点，求其中序遍历时的下一个节点。在这棵二叉树里，每个节点均有指向其父节点的指针。

### 举例

如下面的一棵二叉树，2 的中序遍历下一个节点是 4，而 4 的下一个节点是 1。



中序遍历的过程是首先遍历左子树，然后访问根结点，最后遍历右子树。因此，根据定义，我们需要判断节点类型。

1) 如果有右子树，那么其中序遍历的下一个节点是右子树的最左节点。比如节点 2 有右子树，那么其下一个节点是节点 4（因为节点 4 没有左子树，所以节点 4 为右子树的最左节点）。

2) 如果没有右子树，我们需要区分两种情况。

a) 如果节点是其父节点的左节点，那么其中序遍历的下一个节点即为其父节点；

b) 如果节点是其父节点的右节点，那么向上查看其父节点，直到找出节点是父节点的左节点为止，并返回该父节点。例如，节点 4 是节点 2 的右节点，向上找到节点 2，而节点 2 是节点 1 的左节点，所以返回节点 1。

---

```
public TreeNode inorderSucc(TreeNode e){
    if(e==null) return null;
    if(e.right!=null){
        //如果有右节点，那么中序下一个节点肯定在其右节点的左子树里
        return getLeftMost(e.right);
    }else{
        //如果没有右节点，向上查看其父节点
        //找出第一个节点是其父节点的左节点为止
        TreeNode p = e.parent;
        while(p!=null && p.left!=e){
            e=p;
            p=e.parent;
        }
        return p;
    }
}

//找出最左节点
TreeNode getLeftMost(TreeNode e){
    TreeNode res = e;
    while(res.left!=null)    res=res.left;
    return    res;
}
```

---

## 面试题 43：二叉搜索树近值 ☆☆☆

给定一颗二叉搜索树和一个值，找出和给定的值最接近的节点。

与从二叉搜索树中找出某个值类似，不同的是找出一个近似的值。分别计算三个差

值的绝对值：目标值与当前节点差值，与左子树返回节点的差值，与右子树返回节点的差值。取三个差值绝对值最小的节点，作为返回节点。

---

```
TreeNode findClosestBST(TreeNode root, int val){
    if(root==null) return null;
    // 当前节点值为目标值，不用继续找了，直接返回当前点
    if(root.val == val) return root;
    //递归调用函数，找出当前节点左右子树的近值
    TreeNode left = findClosestBST (root.left, val);
    TreeNode right = findClosestBST (root.right, val);
    int diffLeft = left==null?
        Integer.MAX_VALUE: Math.abs(val-left.val);
    int diffRight = right==null?
        Integer.MAX_VALUE: Math.abs(val-right.val);
    int diff = Math.abs(val-root.val);
    //取三个差值绝对值最小的值
    int min=Math.min(diff, Math.min(diffLeft,diffRight));
    //返回拥有差值绝对值最小的节点
    if(min==diff)
        return root;
    else if(min == diffLeft)
        return left;
    else return right;
}
```

---

## 面试题 44：二叉搜索树 KNN ☆☆☆☆

给定一棵二叉搜索树和一个值，找出和给定的值最接近的 K 个值。

与上一道题“二叉搜索树近值”类似，找出和目标值最接近的值，不同的是需要找出 K 个，而不是一个。我们知道，中序遍历二叉搜索树便可到达一个有序序列。通过这个序列，我们可以先填充 K 个数至一个堆里。当堆的元素个数等于 K 时，再比较两个



差值的绝对值。目标值与当前节点差值的绝对值以及与堆顶元素差值的绝对值。取较小的一个，放入堆里。最后，返回含有 K 个差值的堆。

---

```
void findKNN(TreeNode root, int k, int target){
    Comparator<Integer> comparator = new Comparator<Integer>(){
        public int compare(Integer a, Integer b){
            //让绝对值最大放至堆顶
            int abs1 = Math.abs(a);
            int abs2 = Math.abs(b);
            if(abs2 > abs1) return 1;
            else if(abs2 < abs1) return -1;
            else return 0;
        }
    };
    PriorityQueue<Integer> heap =
        new PriorityQueue<Integer>(k, comparator);
    //调用实现函数
    findKNN (root, k, target, heap);
    //输出这 K 个值
    Iterator<Integer> it = heap.iterator();
    while(it.hasNext()){
        System.out.println(it.next()+target);
    }
}

void findKNN(TreeNode root, int k, int target,
    PriorityQueue<Integer> heap){
    if(root==null) return;
    //中序遍历，也就是按排序顺序找出 k 个数。
    findKNN(root.left, k, target, heap);
    if(heap.size()<k) heap.add(root.val-target);
    else{
        //计算两个差值的绝对值，取其较小者
        int diff=Math.abs(root.val-target);
        int maxdiff = Math.abs(heap.peek());
        if(maxdiff>diff){
            //替换堆顶元素。
        }
    }
}
```

---

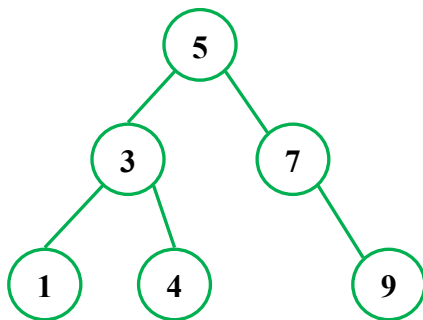
```
        heap.poll();  
        heap.add(root.val-target);  
    }  
}  
findKNN(root.right, k, target, heap);  
}
```

## 面试题 45：实现二叉搜索树迭代器 ☆☆☆☆

实现二叉搜索树迭代器类，至少实现 `hasNext()` 和 `next()` 函数。

### 举例

如下面的一棵二叉搜索树，如果已经获取的节点是 3，那么调用 `next()` 函数，返回为 4；再次调用，返回 5。



### 提问

应聘者：`next()` 返回的是中序遍历的下一个节点吗？

面试官：你可以这么认为。

应聘者：如果这样的话，把中序遍历的结果保存在一个数组中，然后使用一个变量

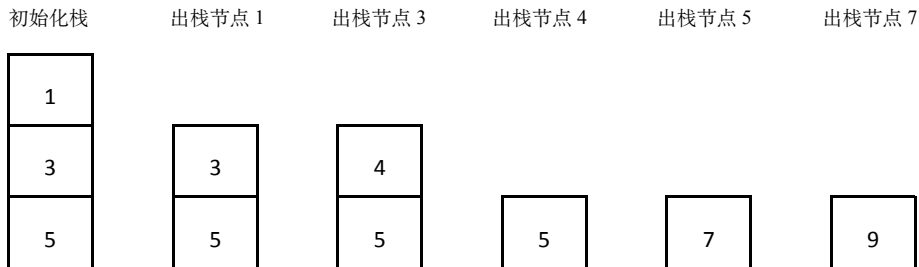
标示当前的位置，可以吗？

面试官：这是一种可行的方案，不过，你需要的这个辅助数组的大小是多少？

应聘者：二叉搜索树元素的个数。

面试官：能不能使用更少的辅助空间？

我们可以考虑使用一个栈，栈顶代表将在下一个返回的节点，栈里保存着从根节点到栈顶节点的路径，但这条路径不包含已经出栈的节点。如何初始化栈呢？根据中序遍历的顺序，第一个元素是最左节点，我们把从根节点到最左节点路径上的节点压入栈。如果让最小值出栈，而且栈顶节点有右子树，那么我们必须同时把其到其右子树的最左节点路径上的节点压入栈，正如我们初始化时，对根节点的操作一样。如果栈顶节点没有右子树，那么下一个返回的节点即为其父节点。根据如上规则，对例子中的二叉搜索树进行操作：



辅助栈存储了从根节点到叶子节点的路径节点，也就是大小是树的高度，因此，所需空间是  $O(\log n)$ ，比  $O(n)$  要小。

---

```
class BSTIterator implements Iterator<Integer>{
    private Stack<TreeNode> stack = new Stack<TreeNode>();
    BSTIterator(TreeNode root) {
        //初始化，压入最左节点，栈头是最小值
        pushLeft(root);
    }
    public boolean hasNext() {
        //以栈是否为空作为标准
    }
}
```

---

---

```
        return stack.isEmpty();
    }

    public int next() {
        TreeNode x = stack.pop();
        //找出栈头的中序遍历的下一个节点
        pushLeft(x.right);
        return x.val;
    }

    private void pushLeft(TreeNode x) {
        //压入最左节点
        while(x != null) {
            stack.push(x);
            x = x.left;
        }
    }
}
```

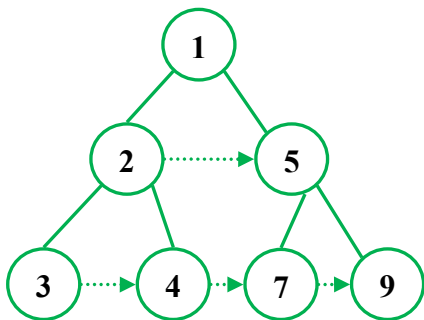
---

## 面试题 46：充实横向指针 ☆☆☆

给定一棵二叉树，每个节点除了有左右节点以外，还有 next 节点。初始的时候，next 为 null，现在要求充实 next，使其指向其右边的节点。如果没有右边节点，那么设置为 null。要求使用常数级空间。给出的二叉树是完全二叉树，即每个节点要么没有子节点，要么有两个，而且所有叶节点都在同一层上。

### 举例

如下面的一棵完全二叉树，充实横向指针后的结果：



因为是一棵完全二叉树，如果有左节点，那么肯定有右节点，我们可以把左右子节点连接起来。但是，如何连接不属于同一个父节点的临近节点呢？我们采取如下措施。

1) 按层遍历完全二叉树，记录每一层的第一个需要遍历的节点。通过观察，我们注意到，下一层的第一个节点是本层第一个节点的左子树。

2) 把属于同一个父节点的左右子节点连接起来。

3) 连接不属于同一父节点的临近节点，即跨子树的连接：我们可以通过父节点的 next 节点，获取另一个父节点，那么该节点的左节点就是临近节点。

---

```
public void connectTree(TreeLinkNode root) {
    TreeLinkNode currLevel = root;
    while(currLevel != null) {
        TreeLinkNode across = currLevel;
        while(across != null) {
            //先把左右子树连起来
            if(across.left != null) {
                across.left.next = across.right;
            }
            //连接同一层的下一个兄弟
            if(across.right != null && across.next != null) {
                across.right.next = across.next.left;
            }
        }
        currLevel = across.next;
    }
}
```

---

---

```
        }  
        //在同一层向后走一步  
        acrossacross.next;  
    }  
    //转到下一层  
    currLevel = currLevel.left;  
}
```

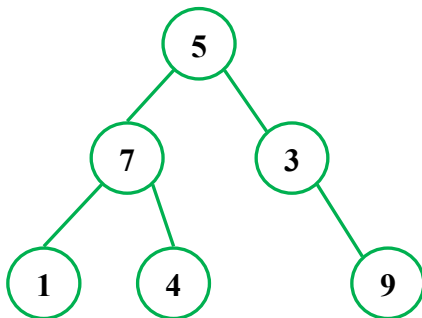
---

## 面试题 47：恢复二叉搜索树 ☆☆☆☆

二叉搜索树里其中有两个节点被调换，要求将它们找出来并恢复二叉搜索树。只允许使用常数级空间。

### 举例

如下面的二叉搜索树中节点 3 和节点 7 被调换了。



### 观察

如果我们能找出被调换的两个节点，恢复这棵二叉树只需要交换两节点值就可以，不用进行删除和插入操作，这样会简单很多。如果有两个节点被调换了，那么意味着中

序遍历二叉搜索树之后，有两个值违反了有序的规则。

一种简单的方法是中序遍历二叉搜索树后，把节点放入一个数组里，然后挑出两个违反有序规则的节点。这样做需要  $O(n)$  的存储空间。其实不需要存储全部的节点，只要记录两个节点即可。我们在中序遍历时，记录第一次违反有序规则的两个节点，第一个被调换的节点肯定是第一个违反有序规则的节点，但是我们不能确定第二个节点是不是被调换的另一个节点。继续寻找，如果第二次发现违反有序规则的两个节点，那么第二个节点就是待查找的被调换的另一个节点。如果没有第二次，那么第一次发现的第二个节点即为被调换的另一个节点。

在递归函数里，为了判断两节点是否有序，我们还需引入一个变量，用来记录前一个节点。这个变量和当前遍历时的节点比较，如果比当前节点值大，那么就违反了有序（升序）原则。

---

```
void recoverTree (TreeNode root, TreeNode n1, TreeNode n2,
                TreeNode prev){
    if(root == null) return;
    //中序遍历，按有序序列
    recoverTree(root.left,n1,n2,prev);
    //发现违反有序规则的两个数
    if(prev!=null && prev.val > root.val) {
        n2 = root; //第二次违反时，才是真正的 n2
        //第一个被调换的节点肯定是第一个违反有序规则的数。
        if(n1==null) n1 = prev;
    }
    prev = root;
    recoverTree(root.right,n1,n2,prev);
}

void recoverTree(TreeNode root) {
    TreeNode n1=null, n2=null, prev=null;
    recoverTree(root,n1,n2,prev);
    if(n1!=null && n2 != null){
        //交换两者值
```

---

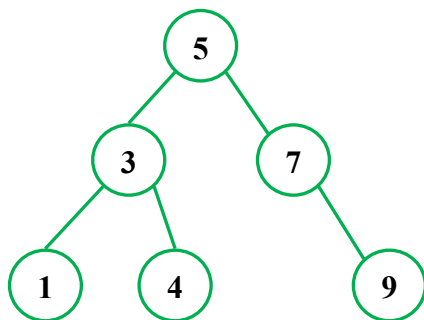
```
        int tmp = n2.val;  
        n2.val = n1.val;  
        n1.val = tmp;  
    }  
}
```

## 面试题 48：按层遍历二叉树 ☆☆☆

输入一棵二叉树，Z 字形按层遍历二叉树节点。

### 举例

如下面的二叉树，Z 字形按层遍历结果如右所示：



Z 字形按层遍历结果：

```
[  
    [5],  
    [7, 3],  
    [1, 4, 9]  
]
```

我们利用两个队列分别保存当前这一层的节点和下一层的所有节点。当遍历某层节点时，插入下一层要遍历的节点。此外，还有个方向标，它决定插入下一层节点时是放在队尾，还是队列头部。若是从左到右，则放在队尾；若是从右到左，则放在队列头部。每层遍历完成时，转向方向标。

```
public ArrayList<ArrayList<Integer>> zigzagLevelOrder(TreeNode root) {  
    ArrayList<ArrayList<Integer>> res =  
        new ArrayList<ArrayList<Integer>> ();  
}
```



---

```
        if(root==null) return res;
        //记录当前这一层所有节点
        LinkedList<TreeNode>currLevel = new LinkedList<TreeNode>();
        currLevel.add(root);
        booleanleftToRight = true;
        while(currLevel.size() > 0) {
            ArrayList<Integer>currList = new ArrayList<Integer>();
            LinkedList<TreeNode>nextLevel =
                new LinkedList<TreeNode> ();
            while(currLevel.size() > 0){
                TreeNodede = currLevel.poll();
                //如果有左右节点，那么加入下一层队列中
                if(de.left != null) nextLevel.add(de.left);
                if(de.right != null) nextLevel.add(de.right);
                if(leftToRight){
                    currList.add(de.val);
                }else{
                    //把右边节点放到前面
                    currList.add(0,de.val);
                }
            }
            res.add(currList);
            currLevel = nextLevel;
            leftToRight = !leftToRight; //转换顺序
        }
        returnres;
    }
}
```

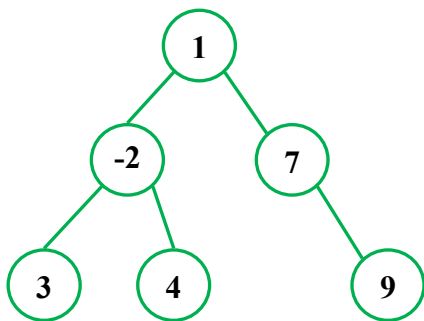
---

## 面试题 49：二叉树最大路径和 ☆☆☆☆

给定一棵二叉树，求最大路径和。路径可以以任意节点开始，到任意节点结束。

### 举例

在如下的二叉树中，最大路径和是  $4-2+1+7+9=19$ ，而不是  $1+7+9=17$ 。



对于每个节点，我们需要比较三个值。

- 1) 左节点为结尾的路径和+当前节点值；
- 2) 右节点为结尾的路径和+当前节点值；
- 3) 当前节点值。

取这三个值的最大值作为以当前节点为结尾的最大路径和返回。同时返回以当前节点为根的子树的最大路径和。如何计算子树的最大路径和呢？我们还需要比较四个值。

- 1) 以当前节点为结尾的最大路径和；
- 2) 左节点为结尾的路径和+当前节点值+右节点为结尾的路径和；
- 3) 左子树的最大路径和；
- 4) 右子树的最大路径和；

取其四值中的最大值作为子树的最大路径和。程序最后返回的也是子树最大路径和。

---

```
public int maxPathSum(TreeNode root) {
    ArrayList<Integer>maxSum = new ArrayList<Integer>(1);
    maxSum.add(Integer.MIN_VALUE);
    //ArrayList可以用来传参，作为子树最大路径和
    getMaxSum(root,maxSum);
    returnmaxSum.get(0);
}

public int getMaxSum(TreeNode root,ArrayList<Integer> maxSum){
    if(root==null) return0;
    intleftSum=0,rightSum=0;
    //左右子树各返回两个结果：以子树为结尾的最大路径和，
    //以及子树下所有路径的最大值
    leftSum= getMaxSum(root.left,maxSum);
    rightSum= getMaxSum(root.right,maxSum);
    //比较三者：自身，自身+左子树最大路径和，自身+右子树最大路径和
    intcurSum = Math.max (root.val,
        Math.max (root.val+leftSum, root. val+rightSum));
    maxSum.set(0,Math.max(maxSum.get(0),
        Math.max(curSum,root. val+leftSum+rightSum)));
    returncurSum;
}
```

---

# 第 9 章

## 字符串

字符串面试题是做文字处理的公司经常出的题目，比如 Google 和 Twitter，而且题目范围较窄。

字符串面试题注意事项：

- 1) 这类题目通常可以借用动态规划、哈希算法和广度优先遍历等解决。
- 2) 重点关注的题目类型有字符串回文、括号匹配、逆波兰式和切词等。

### 面试题 50：字符判重 ☆☆☆

输入字符串，判断该串是否不含有重复字符。

## 提问

应聘者：字符的编码是 UTF-8 还是 ASCII 码？

面试官：如果是 UTF-8，你会怎么处理？如果是 ASCII 码呢？

应聘者：两者都可以使用哈希表来记录字符的出现情况。但是如果是 ASCII 码，一个长度为 256 的布尔数组会节省许多空间。

面试官：那你就写个 ASCII 码字符的判重代码吧。

总共有 256 个 ASCII 字符，因此，我们使用长度为 256 的布尔数组记录每个字符的出现情况。从头到尾遍历一遍字符，如果在布尔数组里已经存在了，则说明有重复。

---

```
boolean isUniqueChars(String str) {
    boolean[] char_set = new boolean[256];
    for (int i = 0; i < str.length(); i++) {
        int val = str.charAt(i);
        if(char_set[val]) return false; //找到重复字符
        char_set[val] = true;
    }
    return true;
}
```

---

## 扩展问题

如果仅仅含有 a-z 字符，如何处理？

字符串只含有 a-z 字符，也就是说 26 个不同字符，那么我们可以通过 32 位的整型数来记录每个字符的出现情况。

---

```
boolean isUniqueChars2(String str) {
    int checker = 0;
    str = str.toLowerCase();
    for (int i = 0; i < str.length(); ++i) {
        int val = str.charAt(i) - 'a';
        if((checker & (1 << val)) > 0) return false; //找到重复字符
    }
}
```

---

---

```
        checked = (1 << val);  
    }  
    return true;  
}
```

---

## 面试题 51：产生括号 ☆☆☆☆

打印出所有 N 对合理的括号组合。

### 举例

当 N=3 时，输出：000, 0(0), (0)0, ((0))

直观地想到使用递归函数产生 N 对合理的括号。在递归调用的过程中，我们需要使用三个变量分别记录当前位置、左括号剩余量和右括号剩余量。初始化时，当前位置为 0，左括号和右括号均为 N。在往组合填充括号时，判断：

- 1) 如果没有剩余的左括号和右括号可以使用，则把当前结果作为合理的括号组合返回；
- 2) 如果左括号还有剩余时，在当前位置插入左括号，递归调用函数填充下一个位置；
- 3) 如果右括号剩余量大于左括号，说明需要匹配左括号，以保证合理的组合，则在当前位置插入右括号，递归调用函数填充下一个位置。

---

```
void printPar(int l, int r, char[] str, int count) {  
    if (l < 0 || r < 0) return; // 终止不合法的括号  
    if (l == 0 && r == 0) {  
        // 找到一个  
        System.out.println(str);  
    } else {  
        if (l > 0) {
```

---

---

```
        //左括号名额还有剩余，则插入左括号。
        str[count] = '(';
        printPar(l + 1, r, str, count + 1);
    }
    if(r > l) {
        //需要匹配左括号，故插入右括号
        str[count] = ')';
        printPar(l, r - 1, str, count + 1);
    }
}

void printPar(int n) {
    char[] str = new char[n*2];
    printPar(n, n, str, 0);
}
```

---

## 面试题 52：提取单词 I ☆☆☆☆

根据字典，从一个抹去空格的字符串里提取出全部单词组合，并且拼接成正常的句子。

### 举例

输入一个字符串：“thisisanexample”，程序输出：“this is an example”。

这是 Google 和 Twitter 的高频题。分词是他们在日常工作中常用的必备技能。首先，找出从字符串第一个字符开始组成的第一个在字典里的单词；如果能找到，那么对剩下的字符串递归调用分词函数，返回所有可能的单词组合，并将之前识别的单词拼接起来，组成正常的句子。为了避免对子串重复调用分词函数，我们使用一个哈希表来记录已经做好分词的子串，加快计算速度。

---

```
ArrayList<String> tokenizeString(String input, HashSet<String> d,
    HashMap<String, ArrayList<String>> memorized){
    ArrayList<String> res = new ArrayList<String>();
    if(input==null || input.length()==0) return res;
    //从已保存的结果中找出提取后的单词，加快提取速度
    if(memorized.containsKey(input))
        return memorized.get(input);
    if(d.contains(input))
        res.add(input); //如果整个串就是一个单词，将其加入结果
    int len = input.length();
    for(int i=1; i<len ; i++){
        String prefix = input.substring(0,i);
        if(d.contains(prefix)){
            //如果这个前缀是一个单词，递归调用函数
            for(String segSuffix:
                tokenizeString (input.substring(i,len),d,memorized)){
                if(segSuffix!=null){
                    //如果后面部分也能提取出单词，将其加入结果中
                    res.add(prefix+segSuffix);
                }
            }
        }
    }
    memorized.put(input, res);
    return res;
}
```

---

### 扩展问题

如果被抹去空格的字符串里，字符的顺序也被打乱了，那么，如何恢复原串呢？

首先为字典里所有单词建立一棵前缀树，同时为输入串的每个字符建立一张哈希表，记录每个字符出现的次数。然后根据前缀树扫描输入字符串，找出单词集合覆盖输入串的所有字符。最后，对单词集合进行排列，每个排列在单词间隔插入空格组成一个字符串。



## 面试题 53：提取单词 II ☆☆☆☆

根据字典，从一个抹去空格的字符串里提取出全部单词组合，求出组合里最大单词个数。

与上一道题目不同的是只需要返回可分割的最大单词数量。通常来说，**求最大数量，而不是求具体方案，都可以使用动态规划的方法**。我们在第三部分算法的动态规划章节将详细介绍动态规划的用法。

一维整型数组  $dp$  记录中间状态。 $dp[i]$ 代表从 0 到  $i$  的子串含有的最大单词数，程序最后返回  $dp[n-1]$ ， $n$  为字符串的长度。 $dp$  计算的等式为：

$dp[j] = \max(dp[j], dp[i+1])$ ，其中  $0 \leq i < j < n$ ，而且  $dp[i] > 0$ ，从第  $i+1$  到第  $j$  个字符组成的字符串是一个单词。

---

```
int segString(String s, HashSet<String> d){
    if(s==null || s.length()==0) return 0;
    int dp[] = new int[s.length()]; //初始化为零
    //记录从 0 到 i 的子串是否为字典里的单词
    for(int i=0; i<s.length();i++){
        String sub = s.substring(0, i+1);
        if(d.contains(sub)){
            dp[i] = 1; //如果整个串是一个单词，则为 1
        }
    }
    for(int i=0; i<s.length()-1; i++){
        for(int j=i+1; j<s.length(); j++){
            if(dp[i]>0){
                //如果 i 之前的能分割成单词
                //那么考虑 i+1 到 j 的情况
                String sub=s.substring(i+1,j+1);
```

---

---

```
        if(d.contains(sub)){
            //如果 i+1 到 j 也是一个单词
            //那么更新从 0 到 j 的最大单词数
            dp[j]=Math.max(dp[j], dp[i]+1);
        }
    }
}
return dp[s.length()-1];
}
```

---

## 面试题 54：字符交替 ☆☆☆

给出两个字符串，打印出所有由这两个字符串的字符交替组成的字符串。要求维持原有字符的相对顺序。

### 举例

输入“AB”和“CD”，打印：

ABCD  
ACBD  
ACDB  
CABD  
CADB  
CDAB

根据两个输入字符串 s1 和 s2 的剩余字符情况，我们必须处理如下几种组合：

1) s1 和 s2 均无字符可用，则返回空串

2) 其中一个无字符可用, 比如 s1 剩余字符为空, 那么把 s2 剩余字符附在已得到的字符组合后, 并输出结果。

3) 其他情况, 或者我们从 s1 取一个字符, 或者我们从 s2 取一个字符, 保存已得到的字符组合, 然后递归调用函数。

---

```
void printInterleavings(String s1,String s2){
    printInterleavings(s1,s2,"");
}

void printInterleavings(String s1,String s2,String soFar){
    if((s1==null||s1.length()==0) && (s2==null||s2.length()==0))
        return;
    if(s1==null || s1.length()==0){
        //s1 为空, 输出结果加上 s2
        System.out.println( soFar + s2 );
        return;
    }
    if(s2==null || s2.length()==0){
        //s2 为空, 输出结果加上 s1
        System.out.print(soFar + s1);
        return;
    }
    //加上 s1 的一个字符
    printInterleavings(s1.substring(1), s2, soFar + s1.charAt(0));
    //加上 s2 的一个字符
    printInterleavings(s1, s2.substring(1), soFar + s2.charAt(0));
}
}
```

---

## 面试题 55: 字符串相乘 ☆☆☆☆

给出两个字符串表示的正整数, 求其乘积, 返回其乘积的字符串表示。

### 举例

字符串“123”和“456”分别代表整数 123 和 456，其乘积的字符串表示为：“56088”。

假设两个字符串 num1 和 num2 的长度分别为 N 和 M，那么其代表整数的乘积最多有 N+M 位。因此，我们需要创建长度为 N+M 的一维数组 num 保存乘积。暂时不考虑进位，我们试一试将两个正整数 123 和 456 相乘：

$$\begin{array}{r}
 \phantom{000}1 \phantom{00}2 \phantom{00}3 \\
 \times \phantom{00}4 \phantom{00}5 \phantom{00}6 \\
 \hline
 \phantom{000}6 \phantom{00}12 \phantom{00}18 \\
 \phantom{00}5 \phantom{00}10 \phantom{00}15 \\
 \phantom{00}4 \phantom{00}8 \phantom{00}12 \\
 \hline
 4 \phantom{00}13 \phantom{00}28 \phantom{00}27 \phantom{00}18
 \end{array}$$

由此获得序列：4，13，28，27，18。最后，为了保证每个数位在 0~9 之间，我们把进位往前累加，得到：5，6，0，8，8。为了方便操作，我们把最小位数的乘积放在数组的最后一位，那么  $\text{num}[i+j+1] = \sum \text{num1}[i] * \text{num2}[j]$ ，其中  $0 \leq i < N, 0 \leq j < M$ 。

---

```

public String multiply(String num1, String num2) {
    int len1 = num1.length(), len2 = num2.length();
    int[] num = new int[len1 + len2];
    int n = num.length;
    for (int i = len1 - 1; i >= 0; i--) {
        for (int j = len2 - 1; j >= 0; j--) {
            // 不考虑进位，模拟乘法
            num[i+j+1] = (num1.charAt(i) - '0') *
                        (num2.charAt(j) - '0');
        }
    }
}
    
```

---

---

```
//确保每个数位大小在 0~9 之间
int carry = 0;
for (int i = n-1; i >= 0; i--) {
    num[i] += carry;
    carry = num[i]/10;
    num[i] = num[i]%10;
}
String result = "";
boolean firstNonzero = false;
//将数组转化为字符串，如果第一位是 0，去掉。
for (int i = 0; i < n; i++){
    //找出第一个非 0 的数字
    if(!firstNonzero && num[i] == 0)
        continue;
    else
        result += num[i];
        firstNonzero = true;
}
if(result.equals("")) return "0";
return result;
}
```

---

## 面试题 56：数字验证 ☆☆☆

验证一个输入字符串是否为合法的数。

### 举例

输入与输出：

“10”=> true

“10”=> true

“10.1”=> true

“xyz”=> false

“ 1a”=>false

“2.”=> false

“2e10”=> true

只需考虑正负符号、小数点、指数 e 符号、数字这四种字符，除了它们之外的字符，程序返回 false。针对这些符号分别处理。

- 正负符号：只能出现在第一个字符或者 e 后面，不能出现在最后一个字符。
- 小数点：字符串不能只含有小数点，也不能只含正负符号和小数点；小数点不能在 e 或者小数点后。
- e：e 不能出现在第一个字符，也不能在最后一个字符；e 前面不能没有数字，也不能有 e。

---

```

public boolean isNumber(String s) {
    boolean res = false;
    if(s==null)return res;
    s = s.trim(); //抹掉前后端的空格
    if(s.equals(""))return res;
    //需要考虑特殊情况：符号、点、指数 E、数字
    boolean hasSign=false, hasDot=false;
    boolean hasExp=false, hasDigit=false;
    s = s.toLowerCase(); //转为小写字母，方便处理指数情况
    int len = s.length();
    for(int i=0; i<s.length(); i++){
        char c=s.charAt(i);
        if(!isValid(c))return false;
        switch(c){
            case '+':
            case '-':
            case '.':
            case 'e':
            case 'E':
                if((i!=0&&s.charAt(i-1)!='e'))

```

---

---

```
        i==len-1){
            //不在第一个或者e后面;在最后一个字符
            return true;
        }else{
            hasSignre;
        }
        break;
    case:
        if(len==1|| (len==2 && hasSign) ||
            hasExp || hasDot){
            //只有一个字符情况;只有符号和点;在e和点之后
            return true;
        }else{
            hasDate;
        }
        break;
    case:
        if(i!=0==len-1 || !hasDigit || hasExp){
            //出现在第一个或最后一个字符;前面没有数字;前面有e了
            return true;
        }else{
            hasExp;
        }
        break;
    default:
        hasDigitue;
        break;
    }
}
return true;
}

public boolean isValid(char c){
    //判断是否在合法字符里:正负符号、小数点、e、数字
    if(c=='+' || c=='-' || c=='.' || c=='e' ||
        ( c>='0' && c<='9')){
        return true;
    }
}
```

---

---

```
        }else{  
            retufalse;  
        }  
    }  
}
```

---

## 面试题 57：字符串转为十进制数 ☆☆

实现 atoi 函数：将字符串转为十进制整数。

### 提问

应聘者：字符串里的字符可能是任意字符吗？

面试官：如果是，怎么办？

应聘者：如何处理非数字的字符？

面试官：你可以忽略这种字符。

应聘者：int 类型是 32 位的吗？

面试官：是的。

应聘者：如果溢出，怎么办？

面试官：返回最大值或者最小值。

这道题题意晦涩，需要不停和面试官沟通，理解对方的考核重点。本道题考察重点 1)判断正负数;2)判断是否溢出。32 位整数最大值为 2147483647,最小值为-2147483648;如果当前结果大于 214748364, 并且还有下一个字符, 则说明已经溢出; 如果当前结果等于 214748364, 并且下一个数字大于 7, 也归为溢出情况。

---

```
int atoi(String str) {  
    if(str== null || str.length() == 0) return 0;  
    booleanisNeg = false;  
    str= str.trim();
```

---



---

```
//获取整数符号
intp = 0;
if(str.charAt(p)=='-'){
    isNeg true;
    p++;
}elseif(str.charAt(p) == '+'){
    isNeg false;
    p++;
}
//把字符串转为整数
int num = 0;
char c;
while (p<str.length()) {
    c = str.charAt(p);
    if(c < '0' || c > '9') break;
    //判断是否溢出
    if (num == 214748364 && ( c - '0' ) > 7 ) ||
        (num > 214748364) ) {
        return isNeg ?
            Integer.MAX_VALUE: Integer. MIN_VALUE;
    }
    num 10*num + (c - '0');
    ++p;
}
return (!isNeg ) ? num : -num;
}
```

---

## 面试题 58：提取 IP 地址 ☆☆☆

给定一个只含数字的字符串，返回所有合法的 ip 地址。

### 举例

输入“10112”，输出：“1.0.1.12”，“1.0.11.2”，“10.1.1.2”。

对输入字符串进行分段处理，每段组成的数不能超过 255。将 ip 地址分为 4 段，每段有三种情况：1) 只取一个数字；2) 取两个数字，第一个数不为 0；3) 取三个数字，第一个数字不为 0，而且三个数字组成的数小于 256。

在上一个章节中，处理二叉树的面试题时，我们大量使用递归函数，分别对左右子树递归调用。其实在处理字符串以及字符排列组合的题目里，问题变得更复杂，每个阶段可能有多种情况，不单单是两个分支的情况，这时候，通常需要采用在循环体内调用递归函数的方法。在本题解法中，是在 for 语句内调用递归函数。

---

```
ArrayList<String> restoreIpAddresses(String s) {
    ArrayList<String> res = new ArrayList<String>();
    String ip = "";
    restoreIpAddresses(s, 0, 0, ip, res);
    return res;
}

void restoreIpAddresses(String s, int start, int part, String ip,
    ArrayList<String> res) {
    //如果数字过多，则忽略
    if(s.length() - start > (4-part)*3) return;
    //忽略剩余数字过少情况
    if(s.length() - start < (4-part)) return;
    //找到一个合法的 ip
    if(start == s.length() && part == 4) {
        //把最后的一个.抹去
        res.add(ip.substring(0, ip.length()-1));
        return;
    }
    int num = 0;
    for(int i = start; i < Math.min(start + 3, s.length()); i++) {
```

---

---

```
//每个位置有三种可能
num = num*10 + (s.charAt(i)-'0');
if (num<=255)
    //每段 ip 地址的数字都小于 256
    ip+=s[i];
    restoreIpAddresses(i+1, part+1, ip+'.', res);
}
//如果第一个是 0 的话，只考虑 0 的情况，不需要考虑 0x 组成的地址
if (num==0) break;
}
}
```

---

## 面试题 59：正则匹配 ☆☆☆☆☆

实现函数 `boolean isMatch(String s, String p)`，`s` 为目标字符串，`p` 为匹配模式。该函数支持 `'.'` 和 `'*'` 的正则匹配，其中 `'.'` 匹配任意单个字符，`'*'` 匹配零个或多个前一个字符。

### 举例

`isMatch("a", "a") => true`

`isMatch("aa", "a") => false`

`isMatch("a", "aa") => false`

`isMatch("aa", "a*") => true`

`isMatch("ab", ". *") => true`

`isMatch("ab", "a*bc*") => true`

使用两个下标 `si` 和 `pi` 记录当前匹配的位置。我们对如下情况逐个分析。

1) 如果已经遍历完了 `p`，则要求 `s` 也被遍历完成；

2) 如果  $pi$  的下一个字符不为 $*$ ，则匹配  $pi$  和  $si$  的当前字符；如果匹配成功，则递归调用函数继续匹配  $pi+1$  和  $si+1$ ；

3) 如果  $pi$  的下一个字符为 $*$ ，则有两条分支：

a) 跳过 $*$ 符号，递归调用函数匹配  $pi+2$  和  $si$ ；

b) 不跳过 $*$ 符号，继续匹配  $pi$  和  $si+1$ 。

4) 最后，还需比较  $pi+2$  和  $si$  的情况，因为如果  $si$  到头了， $'*'$ 也是能匹配空串的。

---

```
boolean isMatch(String s, String p){
    if(s== null ) return p==null;
    if(p == null) return false;
    return isMatch(s, p, 0, 0);
}

boolean isMatch(String s, String p, int i, int j){
    //p 到头了，看看 s 是否到头
    if(j== p.length()) return i == s.length();
    //如果后一个字符不是*情况
    if(j == p.length() - 1 || p.charAt(j + 1) != '*'){
        if(i== s.length()) return false;
        //转到下一个字符
        return p.charAt(j) == '.' || s.charAt(i) == p.charAt(j)
            && isMatch(s, p, ++i, ++j);
    }
    while(i < s.length() && (p.charAt(j) == '.' ||
        s.charAt(i) == p.charAt(j))){
        //跳过*字符，如果匹配，返回 true，否则，继续比较
        if(isMatch(s, p, i, j + 2)) return true;
        i++;
    }
    //跳过*字符，方便检查 s 到头情况。
    return isMatch(s, p, i, j + 2);
}
```

---

---

## 第三部分 算法

---



# 第 10 章

## 俩指针

在比较数组或链表元素时，通常需要两个指针遍历整个数组或链表。这两个指针或者在数组的同一端做同向移动，或者在数组的两端做相向移动，而且可能移动速度不一致。通过两个指针解决比较、移动、挑选数组或者链表元素的方法，我们称之为“俩指针”。

### 面试题 60：有序数组去重 ☆

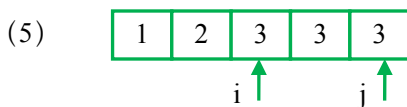
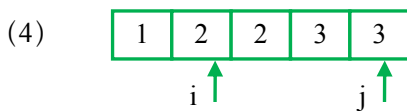
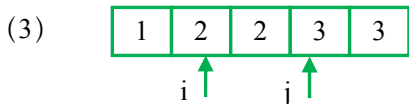
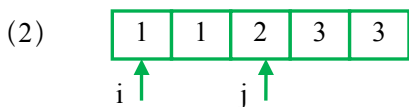
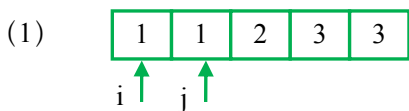
给出一个有序数组，就地移除重复元素，保持每个元素只出现一次并返回新的数组长度。

#### 举例

输入数组  $A=[1, 1, 2, 3, 3]$ ，输出长度 3， $A$  为  $[1, 2, 3]$ 。

使用两个下标  $i$  和  $j$ ，其中  $i$  记录新数组最后一个元素的位置， $j$  遍历整个数组，通过  $j$  移动来移动  $i$ 。当  $j$  和  $i$  指向不同值的元素时，把  $j$  指向的值拷贝至  $i$  指向的元素，然后将  $i$  往后移动一位。当  $j$  和  $i$  指向元素值相同时，不对  $i$  做操作，只移动  $j$ ，以保证  $i$  遍历过的元素值各不相同。

以数组  $A$  为例，



---

```
int removeDuplicates(int A[]) {
    inti=0; //第一个指针
    intj;   //第二个指针
    intn = A.length;
    if(n<=1) return n;
    for(j=1;j<n;j++) {
        //j 在 i 前面走
        if(A[j] != A[i]) { //如果两者值不同，则在 i+1 位置保存 A[j] 的值
            A[++i]=A[j];
        }
    }
    returni+1; //返回下标+1 的长度。
}
```

---

不需要进行删除元素的操作，进而避免每次删除带来移动后续元素的开销。因此，



算法复杂度为  $O(n)$ ，而且不需要额外分配线性空间。

### 扩展问题

如果允许重复元素最多出现两次呢？

放宽把  $j$  指向的值拷贝至  $i$  指向的元素的条件：除了当  $j$  和  $i$  指向不同值的情况，我们还加上  $j$  指向的值不等于  $i-1$  指向的值的值的情况。

---

```
int removeDuplicatesII(int A[]) {
    int n = A.length;
    if(n <= 2) return n;
    int i = 1; // 第一个指针
    int j;     // 第二个指针
    for(j = 2; j < n; ++j) {
        //复制俩指针的值不等，或者第二个指针值不等于第一个指针的前一个值
        if(A[j] == A[i] && A[j] == A[i - 1])
            A[i-1] = A[j];
    }
    return i + 1;
}
```

---

## 面试题 61：三数之和 ☆☆☆

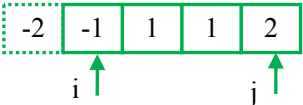
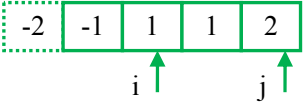
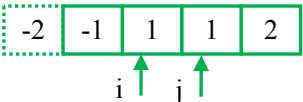
给出一个整型数组，找出所有三个元素的组合，其组合之和等于 0。要求在结果集合里不含有重复的组合。

### 举例

数组  $A=[-2, 1, -1, 2, 1]$ ，三数之和为零的有： $\{-2, 1, 1\}$ 。

直观的方法是挑选第一个元素，然后在其后挑选第二个元素，再从除了已经挑选出的两个元素之外挑出第三个元素，判断三者之和是否为零。这种方法的时间复杂度为  $O(n^3)$ ，肯定不是面试官想要的。

我们可以考虑先将数组排序，从头依次挑选出第一个元素，在其后选两个元素。为了充分利用有序的属性，使用分别在首尾部的俩指针来挑选后两个元素。

- (1)  挑选-2 作为第一个元素。俩指针在首尾两端，其和小于 2，移动首端。
- (2)  俩指针和为 3，大于 2，移动尾端。
- (3)  和为 2，输出结果。俩指针相遇，故该循环结束。

---

```
public ArrayList<ArrayList<Integer>> threeSum(int[] num) {
    ArrayList<ArrayList<Integer>> resSet =
        new ArrayList<ArrayList<Integer>>();
    if(num.length < 3) return resSet;
    Arrays.sort(num);  //对输入数组进行排序
    for(int i=0; i<num.length-2; ++i) {
        if(num[i] > 0) break; //第一个数如果大于 0，那就没必要进行下去
        if(i>0 && num[i]==num[i-1]) continue; //去除重复的结果
        // 一个指针在首部，另一个指针在尾部，相向移动
        int start=i+1, end=num.length-1;
        while(start<end) {
            int sum = num[i] + num[start] + num[end];
            if(sum < 0) {
                start++; 和过小，移动首部指针
            } else if (sum > 0) {
                end--; 和过大，移动尾部指针
            }
        }
    }
}
```

---

---

```
        else { //找到一个结果，并记录
            ArrayList<Integer> res = new ArrayList<Integer>(3);
            res.add(num[i]);
            res.add(num[start]);
            res.add(num[end]);
            resSet.add(res);
            // 忽略可能相同的结果；
            do start++; } while (start<end &&
                                num[start]==num [start-1]);
            do { end--; } while (start<end && num[end]==num[end+1]);
        }
    }
}

return resSet;
}
```

---

在程序的最后，为了避免出现重复的结果，我们忽略相同元素的计算。程序时间复杂度为  $O(n^2)$ ，主要花在移动首尾指针上。

## 面试题 62：股票买卖 ☆☆

给出一个数组，第  $i$  个元素代表第  $i$  天的估价。假设最多允许进行一次买卖，求可能的最大利润。

### 观察

这道题可以转化为：求出  $\max(A[j]-A[i])$ ，当  $j>i$  时。

使用俩指针：一个扫描整个数组，另一个记录已扫描元素中的最小值。在第一个指针扫描数组时，一边移动另一个指针，一边更新最大利润。此算法只需扫描数组一遍，故算法复杂度为  $O(n)$ 。

---

```
public int maxProfit(int[] prices) {  
    if(prices.length==0)    return 0;  
    int profit = 0 ;  
    int min=prices[0]; //使用第一个指针记录最小值  
    for(int i = 1; i < prices.length;i++){  
        //第二个指针扫描整个数组  
        min = Math.min(min,prices[i]);  
        profit = Math.max(profit,prices[i]-min);    //更新最大利润  
    }  
    return profit;  
}
```

---

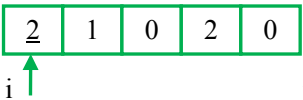
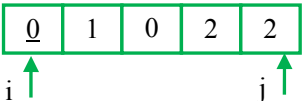
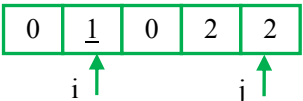
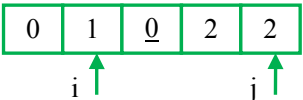
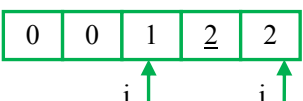
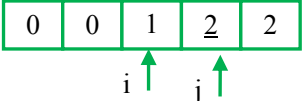
## 面试题 63：三色排序 ☆☆☆☆

输入一个整型数组，每个元素值在 0 到 2 之间，其中 0, 1, 2 分别代表红色，白色和蓝色。现要求对数组进行排序，相同颜色在一起，而且白色在红色之后，蓝色又在白色之后。要求时间复杂度为  $O(n)$ 。

直观的方法是对数组进行排序，比如使用快速排序，时间复杂度为  $O(n\log n)$ ，但是这种解法并未充分利用元素值在有限范围的属性。

第二种方法是计数排序，先扫描一遍分别记录 0、1 和 2 的个数，然后在对数组进行赋值。总共需要扫描整个数组两边，比快速排序快了不少，但是还是没有利用元素值只有 3 种的属性。有没有只扫描一遍就能解决的方法？

当我们扫描数组的时候，使用首尾俩指针：首指针标示红色与白色边界，尾指针标示白色和蓝色边界。比如，我们有数组  $A=\{2, 1, 0, 2, 0\}$ ：

- (1)  初始化时，首指针指向第一个元素，尾指针指向最后一个元素的后一个位置，并且从首指针开始扫描
- (2)  当前元素为 2，将尾指针往前移动，并交换当前元素与尾指针的值。
- (3)  交换后，当前元素为 0，与首指针值交换，首指针向前移动一格。
- (4)  当前元素为 1，不做处理，继续扫描下一个元素。
- (5)  当前元素为 0，与首指针值交换，首指针向前移动一格。
- (6)  当前元素为 2，将尾指针往前移动，并交换当前元素与尾指针的值。当前元素与尾指针相遇，扫描结束。

扫描一遍，实时更新三种颜色的两个边界，就能完成三色排序。

---

```

Int[] sortColors(int A[]) {
    //使用俩指针，首指针记录红色边界，尾指针记录蓝色边界
    int p0 = 0, p2 = A.length;
    for(int i = 0; i < p2; ++i) {
        // 把红色放在首部
        if(A[i] == 0) {
            int tmp = A[p0];
            A[p0] = A[i];
            A[i] = tmp;
            p0++;
        }
    }
    // 把蓝色放在尾部
    for(int i = p2 - 1; i >= p0; --i) {
        if(A[i] == 2) {
            int tmp = A[p2 - 1];
            A[p2 - 1] = A[i];
            A[i] = tmp;
            p2--;
        }
    }
}

```

---

---

```
        }  
        // 把蓝色放在尾部，这样白色自然就在中间了  
        else (A[i] == 2) {  
            p2--;  
            tmp = A[p2];  
            A[p2] = A[i];  
            A[i] = tmp;  
            i--;  
        }  
    }  
    return A;  
}
```

---

## 面试题 64：蛙跳 ☆☆☆

给出一维非负元素的数组，每个元素代表从该元素位置能跳的最远距离。假设初始位置在第一个元素，现根据输入数组判断是否能跳到数组的末尾。

### 举例

数组 {2, 1, 3, 1, 1} => true

数组 {3, 2, 1, 0, 1} => false，卡在第四个元素。

同向移动的俩指针：第一个指针扫描当前值，第二个指针记录当前最远的目的地。扫描的同时，实时更新当前最远目的地。如果当前最远目的地能到达数组尾端，程序返回 true。如果当前最远目的地不能到达尾端，而且又不能向前移动的时候，我们认为不可能到达终点，程序返回 false。

---

```
boolean canJump(int A[]) {  
    if (A.length <= 1) return true;
```

---

---

```
//使用俩指针，一个扫描当前值，另一个记录当前最远目的地
int i, currMax = 0;
for(i=0 ; i < A.length - 1; ++i) {
    //如果当前最远目的地不能向后移动的话，我们认为没有希望到达终点
    if(A[i] == 0 && currMax < i+1) return false;
    if(A[i] + i > currMax && A[i]>0) {
        //移动当前最远目的地
        currMax = A[i];
        //如果能到达终点，则返回 true
        if(currMax >= A.length-1) return true;
    }
}
return false;
}
```

---

### 扩展问题

假设输入数组能满足达到数组末尾的条件，求出最少条数。例如，输入数组{2, 1, 3, 1, 1}，则输出 2。

我们考虑使用贪心算法。有两个变量分别记录上一轮和本轮的最远目的地。当且仅当上一轮最远目的地更新的时候，我们实施一跳。什么时候更新上一轮的值呢？在扫描的过程中，如果当前位置越过了上一轮的最远目的地，那么我们需要更新上一轮的值。

---

```
int jump(int A[]) {
    int result = 0;
    //使用俩指针，分别记录上一轮和本轮的最远目的地
    int last = 0, curr = 0;
    for(int i = 0; i < A.length; ++i) {
        if(i > last) {
            //如果当前指针越过了上轮最远目的地
            //那么更新上轮指针，并且实施一跳
            last = curr;
            result++;
        }
    }
}
```

---

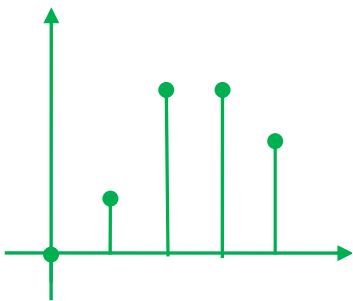
```
    }  
    //保存本轮的最远目的地的值。  
    curr=Math.max(curr, i+A[i]);  
}  
return result;  
}
```

## 面试题 65：容器盛水 I ☆☆☆

给出  $n$  个非负整数： $a_1, a_2 \dots a_n$ ，每个元素代表平面上的一个点 $(i, a_i)$ ，然后将这些点 $(i, a_i)$ 和  $x$  坐标上的点 $(i, 0)$ 连成一条垂直于  $x$  轴的线段。找出两条线段，连同  $x$  轴组成的容器所容纳的水最多。

### 举例

输入数组  $A=\{0, 1, 3, 3, 2\}$ ， $A[2]$ 和  $A[4]$ 连同  $X$  轴组成的容器所容纳的水最多，为 4。



- $A[0]$ 与其他轴组成容器大小均为 0
- $A[1]$ 与  $A[2]$ 、 $A[3]$ 、 $A[4]$ 所组成容器大小分别为：1, 2, 3
- $A[2]$ 与  $A[3]$ 、 $A[4]$ 组成容器大小分别为：3, 4
- $A[3]$ 与  $A[4]$ 组成容器大小为 2

我们考虑使用首尾相向移动的俩指针。先计算这两个指针指向元素与  $X$  轴围成容器的大小，同时更新当前最大值。然后移动首尾俩指针，直至它们相遇。那么，如何移动首尾指针呢？移动指向较短线段的指针，即如果首指针指向较小值，那么首指针往后移动一位；如果尾指针指向较小值，那么尾指针往前移动一位。如何证明最大值会出现在



该移动方法中？答案是用反证法。当前首尾指针分别是  $i$  和  $j$ ，其中  $A[i] < A[j]$ ，那么移动  $i$ 。假设有个  $k$ ，在  $i$  和  $j$  之间， $A[k]$  和  $A[i]$  组成容器所容纳的水最多，那么势必  $A[k] > A[j]$ 。然而， $A[i]$  和  $A[k]$  组成容器的容量为  $(k-i) \cdot \min(A[i], A[k])$ ，即  $(k-i) \cdot A[i]$ ，小于  $A[i]$  与  $A[j]$  所组成的容器大小： $(j-i) \cdot A[i]$ 。

---

```
int maxArea(int A[]) {
    //俩指针，在首尾，相向移动
    int i = 0, j = A.length - 1;
    int result = 0;
    while(i < j) {
        int area = Math.min(A[i], A[j]) * (j - i); //求当前面积
        result = Math.max(result, area); //更新最大值
        if(A[i] < A[j]) //移动较短线段的指针
            i++;
        else
            j--;
    }
    return result;
}
```

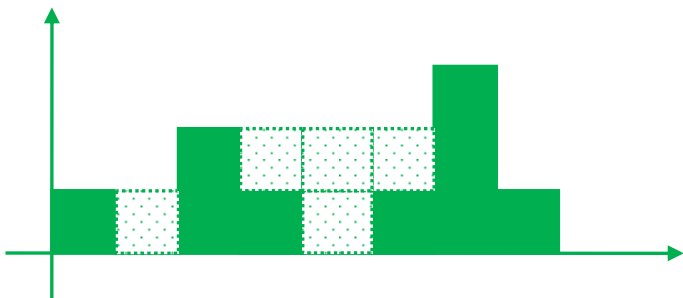
---

## 面试题 66：容器盛水 II ☆☆☆☆

给出一个非负的数组，每个元素代表从  $x$  轴到该元素值并且宽度为 1 的柱子，求雨后能盛多少水。

### 举例

输入数组  $A=\{1, 0, 2, 1, 0, 1, 3, 1\}$ ，输出 4。



为了便于计算，我们对每根柱子单独计算能盛多少水，最后累积每根柱子上的盛水量，即为所有的盛水量。每根柱子能盛的水并不是自身能决定的，而是取决于其左右两边的柱子。我们先记录最高的柱子，把数组一分为二，分别计算最高柱子左右两边的盛水量。

以最高柱子的左边部分为例，从首端开始扫描，并使用第二个指针记录已扫描部分中的最高柱子。如果历史最高柱子（即第二个指针指向的值）比当前柱子高，这说明当前柱子能盛水，所构建容器由历史最高柱子、当前柱子以及最高柱子这三根柱子紧挨着组成，其所盛水的量为：历史最高柱子-当前柱子高度。如果历史最高柱子矮于当前柱子，则更新历史最高柱子为当前柱子。

---

```
int maxAreaII(int A[]) {  
    int n = A.length, result = 0;  
    int highest = 0; //第一个指针记录最高的柱子  
    for(int i = 0; i < n; ++i) {  
        if(A[i] > A[highest])  
            highest = i;  
    }  
    int second = 0; //第二个指针记录之前扫描的最高柱子  
    //从首端开始扫描，求左半部分的水  
    for(int i = 0; i < highest; ++i) {  
        if(second > A[i])  
            //求出这三根柱子紧挨着所能盛的水: second, i, highest
```

---

---

```
                resultsecond = A[i];
            else
                second = A[i];
        }
        second = 0; //从末端开始倒序扫描，求右半部分的水
        for (int i = n - 1; i > highest; --i) {
            if(second > A[i])
                resultsecond = A[i];
            else
                second = A[i];
        }
        return result;
    }
}
```

---

## 面试题 67：数组分水岭 ☆☆☆

在一维数组中，找出一个点，使得其所有左边的数字均小于等于它，所有右边的数字都大于等于它。要求在线性时间内返回这个点所在的下标。

### 提问

应聘者：是不是一定能找出这个点？

面试官：不一定。

应聘者：如果找不到这个点，如何处理？

面试官：返回-1。

### 举例

输入 A={1, 0, 1, 0, 1, 2, 3}，返回下标 4 或 5。

首先，从左到右扫描一遍数组，通过一个辅助布尔数组记录哪些大于等于其之前所

有元素的元素；其次，从右到左第二遍扫描数组，如果其后所有元素大于等于当前元素，而且在第一次遍历时当前元素大于等于之前的所有元素，那么程序返回其下标。

---

```
int getMagnitudePole(int[] A){
    if(A == null || A.length == 0)    return -1;
    boolean[] isCurrMax = new boolean[A.length];
    int first = A[0]; //第一个指针记录当前最大值
    for(int i = 0; i < A.length; i++) {
        if(A[i] >= first) {
            first = A[i];
            //如果比当前最大值还大，那么认为它大于之前的所有元素
            isCurrMax[i] = true;
        }
    }
    int second = A[A.length-1]; //第二个指数记录当前最小值
    for(int i = A.length - 1; i >= 0; i--) {
        if(A[i] <= second) {
            second = A[i];
            //不小于左边元素，又不大于右边元素，返回该元素下标
            if(isCurrMax[i])    return i;
        }
    }
    return -1;
}
```

---

# 第 11 章

## 动态规划

动态规划的方法是将一个问题化解为多个子问题，当某个给定子问题的解已经算出时，将其记忆化存储，以便下次解决同一个子问题时直接给出答案，避免重复计算。动态规划适用于有重叠子问题和最优子结构性质的问题，通常能够把指数级的算法复杂度下降为多项式的算法复杂度。

动态规划面试题的注意事项：

- 1) 需要一个辅助空间，记录子问题的解决结果。这种辅助空间通常是三维以下的数组。
- 2) 明确各个子问题之间的关系，以避免重复计算。
- 3) 通常对数组和字符串的高难度面试题非常有效。当遇到此类的面试题而又毫无头绪时，动态规划是首要选择。

## 面试题 68：最长递增子序列 ☆☆☆☆

给出一个数组，找出最少元素，使得将其删除之后，剩下的元素是递增有序的。

### 观察

删除最少的元素，保证剩下的元素是递增有序的。换一句话说，找出最长的递增有序序列。

找出最长递增有序序列有多种方法。在这里，我们除了需要额外一维数组 `dp` 记录以及每个元素为结尾的最长子序列的长度以外，还需要一个哈希表，用来保存在最长子序列的元素。`dp[i]`代表以输入数组 `A[i]`为结尾的最长子序列长度，`dp[i]`通过如下方式计算：

$$dp[i] = \begin{cases} 1, & \text{初始化时每个元素构成一个序列} \\ \max(dp[i], dp[j]+1), & \text{其中 } j < i, \text{ 并且 } A[i] \geq A[j] \end{cases}$$

最后，通过回溯哈希表的方法，把需要删除的元素剔除。

---

```
ArrayList<Integer> minDelete(int[] A) {
    ArrayList<Integer> res = new ArrayList<Integer>();
    //通过倒序追踪记录最长的递增子序列
    HashMap<Integer,Integer> bt= new HashMap<Integer,Integer>();
    int[] dp= new int[A.length]; //记录每个元素为结尾的最长子序列长度
    int maxCount=0;
    int end=0; //最长递增子序列的最后一个元素
    for(int i=0; i<A.length; i++){
        dp[i] = 1;
        for(int j=0; j<i; j++){
```

---

---

```
        if (A[i] >= A[j]) {
            dp[i] = Math.max(dp[i], dp[j] + 1);
            if (maxCount < dp[i]) {
                maxCount = dp[i];
                bt.put(i, j);
                end = i;
            }
        }
    }
}

int k = A.length - 1;
while (k >= 0) {
    // 加入需要被删除的元素
    while (k > end) { res.add(A[k]); k--; }
    k--;
    if (bt.containsKey(end)) {
        end = bt.get(end); // 求出 LIS 的上一个元素
    } else end = -1; // LIS 到头了
}
return res;
}
```

---

## 面试题 69：最小化数组乘积 ☆☆☆☆

给出两个数组 A 和 B，两个数组大小分别为 m 和 n，其中  $m < n$ 。现要求将  $(n-m)$  个 0 插入 A，使得  $A*B$  的值最小，求其最小乘积。

### 举例

两个数组  $A = \{1, -1\}$ ， $B = \{1, 2, 3, 4\}$ ，如果把两个零插入数组 A 中间，得到  $A' = \{1, 0, 0, 1\}$ ，使得数组乘积最小，即为 -3。

一个二维数组 dp 记录 A 和 B 不同位置组合的最小乘积， $dp[i][j]$  代表  $A[0...i]$  与  $B[0...j]$

组合的最小乘积。当  $i=0, j=0$  时, 只有一种组合, 无法插入 0, 即  $A[0]*B[0]$ ; 当  $i=0$  时,  $j$  可以从 0 到  $n-m$ , 也就是  $n-m$  个 0, 加上  $B[j]$  本身与  $A[0]$  进行组合; 当  $i>0$  时, 因为我们只能往较短数组  $A$  插入 0, 所以我们要么累加  $A[i]*B[j]+dp[i-1][j-1]$ , 要么累加  $0*B[j]+dp[i][j-1]$ , 取两者较小值。

$$dp[i] = \begin{cases} A[i]*B[j], & \text{当 } i=0, j=0 \text{ 时} \\ \min(A[i]*B[j], dp[i][j-1]), & \text{当 } i=0, 0<j\leq n-m \text{ 时} \\ \min(A[i]*B[j]+dp[i-1][j-1], dp[i][j-1]), & \text{当 } 0<i\leq j\leq i+n-m \text{ 时} \end{cases}$$

---

```
int minProduct(int A[], int B[]) {
    int m=A.length, n=B.length;
    int[][] dp = new int[m][n]; //记录每种组合的最小乘积
    for(int i = 0; i < m; i++) {
        for(int j = i; j < i + (n - m) + 1; j++) {
            if(j<i) {
                //要么使用 A[i]*B[j], 要么使用 0*B[j]
                if(i) dp[i][j] = Math.min(A[i] * B[j] +
                    dp[i - 1][j - 1], dp[i][j - 1]);
                else dp[i][j] = Math.min(A[i] * B[j], dp[i][j - 1]);
            } else {
                if(i==0){
                    dp[i][j] = A[i] * B[j];
                } else {
                    dp[i][j] = A[i] * B[j] + dp[i-1][j-1];
                }
            }
        }
    }
    return dp[m-1][n-1];
}
```

---



## 面试题 70：股票买卖 II ☆☆☆☆

给出一个数组，第  $i$  个元素代表第  $i$  天的股票，求最大交易利润。允许最多交易两次。

一维数组 `currProfit` 记录截止当日最大利润：

`currProfit[i]=max(currProfit[i-1],prices[i]-low)`

以上值通过从头到尾扫描一遍数组即可获得。此外，我们还需要一个一维数组 `futureProfit` 计算当日以后的交易最大利润：

`futureProfit[i]=max(futureProfit[i+1], high-prices[i])`

以上最大利润通过从尾到头逆向扫描数组获得。合计之前计算的利润，求出最大利润：

`maxProfit = max(maxProfit,currProfit[i]+futureProfit[i])`

---

```
int maxProfitII(int[] prices) {
    int len = prices.length;
    if(len==0) return 0;
    int[] currProfit = new int[len];
    int[] futureProfit = new int[len];
    int low=prices[0];
    int maxProfit = 0;
    //记录截止当日最大利润
    for(int i = 1; i<len; ++i) {
        low=Math.min(low,prices[i]);
        currProfit[i]=Math.max(currProfit[i-1],prices[i]-low);
    }
    int high=prices[len-1];
    // 计算该日以后的交易最大利润，合计之前计算的利润，求出最大利润
```

---

---

```
        for(int i = len-1; i>=0; --i) {
            high=Math.max(high, prices[i]);
            if(i<len-1) {
                futureProfit[i]=Math.max(futureProfit[i+1],
                                         high-prices[i]);
            }
            maxProfit=Math.max (maxProfit,currProfit[i]+
                               futureProfit [i]);
        }
        return maxProfit;
    }
}
```

---

## 面试题 71：数组最大和 ☆☆☆

输入一个数组 A，求其连续子数组的最大和。

### 举例

输入数组 A={-3, 1, -3, 4, -1, 2, 1}，连续子数组{4, -1, 2, 1}有最大和 6。

直观的想法是使用一维数组记录以每个元素为结尾的子序列的最大和，然后再扫描一遍这个一维数组，以获取最大值。我们确实需要一位数组来记录每个位置的最大和吗？不需要，因为我们不需要返回有最大和的子序列。使用一个变量记录以当前位置结束的连续子序列的最大和：

```
currMax = Math.max(A[i], currMax + A[i])
```

另一个变量记录历史最大和 max，currmax 更新时，max 也随之更新。

---

```
public int maxSubArray(int[] A) {
    if(A==null || A.length==0) return 0;
    int currMax = A[0], max = A[0];
}
```

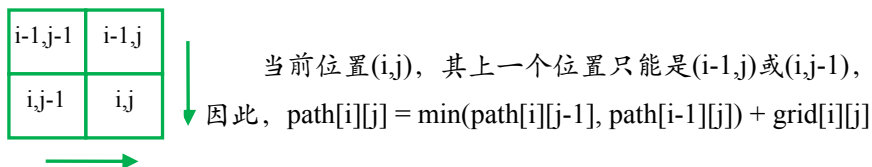
---

```
for (int i = 1; i < A.length; i++) {  
    //以当前位置结束的子序列之和  
    currMax = Math.max(A[i], currMax + A[i]);  
    max = Math.max(max, currMax); //保留最大值  
}  
return max;  
}
```

## 面试题 72：二维数组最小路径和 ☆☆☆

给出一个二维  $m \times n$  矩阵 `grid`，含有非负整数。找出一条路径从最左上角到右下角，使之经过元素之和最小。假定只能向右或向下移动。

二维数组 `path` 记录以当前位置结束的路径的最小和，比如，`path[i][j]` 代表从 `grid[0][0]` 开始到 `grid[i][j]` 结束的最小路径和。



程序最后返回 `path[m-1][n-1]`，即从最左上角到右下角的最小路径和。

```
int minPathSum(int[][] grid) {  
    int m = grid.length, n = grid[0].length;  
    int[][] path = new int[m][n];  
    path[0][0] = grid[0][0];  
    for (int i = 1; i < m; i++) { //左边界  
        path[i][0] = path[i-1][0] + grid[i][0];  
    }  
    for (int j = 1; j < n; j++) { //顶部边界  
        path[0][j] = path[0][j-1] + grid[0][j];  
    }  
}
```

```

    }
    for (int i = 1; i < m; i++) {
        for(int j = 1; j < n; j++) {
            //计算从左边过来的路径和从上面下来的路径最小值。
            path[i][j] = Math.min(path[i][j-1] + grid[i][j],
                                   path[i-1][j] + grid[i][j]);
        }
    }
    return path[m-1][n-1];
}
    
```

## 面试题 73：三角形最小路径 ☆☆☆

输入一个三角形，找出一条从顶部到底部的最小路径和。你只能往下移动相邻的节点。

举例

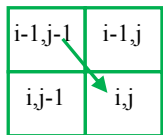
```

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

如右图，带有下列划线的路径：

$2 \rightarrow 3 \rightarrow 5 \rightarrow 3$ ，拥有最小路径和 13。

同上一道题“二维数组最小路径和”类似，一个二维数组 `path` 记录以当前位置结束的路径的最小和，比如，`path[i][j]` 代表从顶部到当前位置结束的最小路径和。



当前位置  $(i, j)$ ，其上一个位置只能是  $(i-1, j)$  或  $(i-1, j-1)$ ，因此， $path[i][j] = \min(path[i-1][j-1], path[i-1][j]) + grid[i][j]$

有没有更节省空间的做法呢？比如仅使用一维数组。把三角形倒过来计算，在第一行就充实这个一维数组，以后计算每一行时，利用上一行相邻位置的结果，取其中较小值，更新当前位置的值。

---

```
public int minimumTotal(ArrayList<ArrayList<Integer>> triangle) {
    ArrayList<Integer>path = new ArrayList<Integer>();
    for(int i=triangle.size()-1;i>=0;i--){
        for(int j=0;j<=i;j++){
            if(i==triangle.size()-1)
                //初始化
                path.add(triangle.get(i).get(j));
            else{
                //相邻元素最小和
                path.set(j,triangle.get(i).get(j)+
                    Math.min(path.get(j),path.get(j+1)));
            }
        }
    }
    return path.get(0);
}
```

---

## 面试题 74：爬楼梯 ☆☆

假设有  $N$  个阶梯的楼梯，每次你只能爬上一个或两个台阶。计算出有多少种不同方法爬到顶部。

相对于其他动态规划的题目，这道题是最简单的。一维数组  $dp$  记录每个台阶的方法数， $dp[i] = dp[i-1] + dp[i-2]$ 。初始化时， $dp[0] = dp[1] = 1$ 。

---

```
int climbStairs(int n) {
    if(n == 0 || n == 1) return 1;
    int[] dp = new int[n+1]; //考虑到数组下标从 0 开始
```

---

```
dp[0] = 1; dp[1] = 1;
for(int i = 2; i < n+1; i++) {
    dp[i] = dp[i-1] + dp[i-2]; //是前两个台阶的方法数之和
}
return dp[n];
}
```

## 面试题 75：迷宫路径数 ☆☆

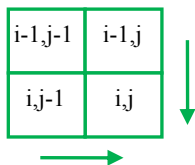
给出一个  $m \times n$  矩阵，求从左上角到右下角的总路径数，只能向右、向下移动。

### 举例

一个  $3 \times 4$  矩阵，从左上角到右下角的总路径数为 10。

1	1 1 1		
1	2 3 4		
1 3		6	10

同上一道题“二维数组最小路径和”类似，一个二维数组 `path` 记录以当前位置结束的路径的最小和，比如，`path[i][j]` 代表从顶部到当前位置结束的最小路径和。



当前位置  $(i, j)$ ，其上一个位置只能是  $(i-1, j)$  或  $(i, j-1)$ ，因此， $path[i][j] = path[i-1][j-1] + path[i-1][j]$

由于我们可以利用上一轮计算的结果（即向下箭头），把二维辅助空间下降为一维辅助空间。


```
public int uniquePaths(int m, int n){
    int[] dp = new int[n];
    //初始化顶部
    for(int i = 0; i < n; i++)    dp[i] = 1;
    //遍历每个格子
    for(int i = 1; i < m; i++){
        for(int j = 1; j < n; j++) {
            //上面+左边的路径数
            dp[j] = dp[j] + dp[j - 1];
        }
    }
    return dp[n - 1];
}
```

### 扩展问题

如果矩阵有障碍物呢？输入的矩阵元素里 1 代表有障碍物，0 代表没有。

如下面的输入矩阵，其路径数为 3：

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



1	0	0	0
1	1	1	0
1	2	3	3

这时候不能简单地相加上面和左边的值，而需要判断当前位置是否有障碍物，如果有，需要清零。

```
public int uniquePathsII(int[][] grid) {
    int m = grid.length;
    int n = grid[0].length;
    int[] dp = new int[n];
    //初始化顶部
    if(grid[0][0] == 0)    dp[0] = 1;
```

---

```

        for(int i = 1; i < n; i++) dp[i] = grid[i][j] == 1? 0: dp[i-1];
    for(int i = 1; i < m; i++) {
        for(int j = 0; j < n; j++) {
            if(j==0){
                //最左侧的元素
                dp[i][j]=grid[i][j] == 1? 0: dp[j];
            }
            else{
                //如果可通行，累加上面和左边的路径数
                dp[i][j] = grid[i][j] == 1 ? 0 : dp[j + 1] + dp[j];
            }
        }
    }
    return dp[n-1];
}

```

---

## 面试题 76：刷房子 ☆☆☆

在一条街上，给 H 个房子刷墙，要求每个房子刷一种颜色，相邻房子不能刷同一种颜色。每种颜色成本不同，求最小刷墙成本。

一个二维数组 dp 记录每个房子刷每一种颜色的最小成本，比如，dp[i][j]代表第 i+1 个房子刷第 j+1 种颜色的最小刷墙成本：

$d[i][j] = \min(d[i-1][k]) + \text{color}[j]$ ，其中  $0 \leq k < \text{color.length}$ ,  $k \neq j$ 。

最后，找出最后一栋房子刷各种颜色之后的最小值，即为整条街的最小刷墙成本。

---

```

int minCost(int h, int[] color){
    if(h==0 || color.length==0) return 0;
    int min=Integer.MAX_VALUE;
    int[][] d= new int[h][color.length];
    for(int i=0; i<h; i++){
        for(int j=0; j<color.length; j++){

```

---



---

```
        if(i==0)d[i][j] = color[j]; //初始化第一个房子
        else{
            min=Integer.MAX_VALUE;
            //找出在刷第 j 种颜色时, 最小代价
            for(int k=0; k<color.length;k++){
                if(k==j)continue;
                min= Math.min(min, d[i-1][k]);
            }
            d[i][j]= min+color[j];
        }
    }
}
min=Integer.MAX_VALUE;
//找出最后一栋房子刷各种颜色之后的最小值
for(int j=0; j<color.length; j++){
    min= Math.min(min, d[h-1][j]);
}
return min;
}
```

---

## 面试题 77：数字解码 ☆☆☆

给只含有 A 到 Z 字符的字符串加密, 加密规则为: ‘A’ → 1, ‘B’ → 2, ..., ‘Z’ → 26。现有一个已经加过密的密文 (仅含有数字), 求其解密的方法个数。

### 举例

现有密文“26”, 它可以解密为“Z”或者“BF”, 两种解密方法。

一维辅助空间 `dp` 记录每个位置解密方法数。为了更直观计算解密方法数, 我们使用倒序扫描输入串。

$$dp[i] = \begin{cases} 0, & \text{当 } s[i]='0' \text{ 时} \\ dp[i+1]+dp[i+2], & \text{当和后面字符组成数字小于等于 26 时} \\ dp[i+1], & \text{当上面条件不成立时} \end{cases}$$

最后，程序返回 `dp[0]` 代表整个输入串的解密方法数。

```
int numDecodings(String s) {
    if(s.length()==0) return 0;
    int len = s.length();
    int[] dp = new int[len+1];
    dp[len]= 1;
    for(int i=len-1; i >= 0; i--) {
        if(s.charAt(i)=='0')
            dp[i]=0; // 当遇到一个 0 时，没有办法破解。
        else
            dp[i]=dp[i+1];
        // 如果和后面的字符组成的数小于等于 26，
        // 那么认为当前两个字符是一种解码方案，故加上 dp[i+2]
        if(i+1<s.length() &&
            (s.charAt(i)=='1' ||
             (s.charAt(i)=='2' && s.charAt(i+1) <='6'))){
            dp[i]+=dp[i+2];
        }
    }
    return dp[0];
}
```

## 面试题 78：子串个数 ☆☆☆☆

输入两个字符串 `S` 和 `T`，求出 `T` 在 `S` 的子序列个数。一个字符串的子序列是从该串删除多个字符（也可能不删除任何字符）而且保留字符相对顺序不变的子串。

## 举例

S=“Examp~~p~~ple”, T=“Example”, T 在 S 的子序列的个数是 2。

使用一个二维辅助空间 dp 记录中间状态。dp[i][j]代表从开始到第 j 个 T 字符组成字符串在从 S 开始到第 i 个 S 字符组成字符串里的子序列个数。

$$dp[i][j] = \begin{cases} dp[i-1][j] + dp[i-1][j-1], & \text{当 S 的第 i 个字符与 T 的第 j 个字符相等} \\ dp[i-1][j], & \text{当上述条件不成立时} \end{cases}$$

我们申请的空间是(sLen+1)\*(tLen+1)，而不是 sLen\*tLen，原因是方便处理空串的情况。在初始化时，dp[i][0]赋值为 1，其中  $0 \leq i \leq sLen$ ，说明空串也属于字符串的一个子序列。

---

```
int numDistinct(String S, String T) {
    int sLen = S.length(), tLen = T.length();
    if(sLen == 0 || tLen == 0 || tLen > sLen ) return 0;
    int[][] dp = new int[sLen+1][tLen+1];
    //初始化边界条件，
    for(int i=0; i<=sLen; i++) dp[i][0] = 1;
    for(int i=1; i<=sLen; i++){
        for(int j=1; j<=tLen; j++){
            if(S.charAt(i-1)==T.charAt(j-1)){
                //如果 S 的第 i 个字符与 T 的第 j 个字符相等
                //需要加上 dp[i-1][j-1]结果
                dp[i][j]=dp[i-1][j] + dp[i-1][j-1];
            }else{
                dp[i][j]=dp[i-1][j];
            }
        }
    }
    return dp[sLen][tLen];
}
```

---

## 面试题 79：编辑距离 ☆☆☆☆

输入两个单词 word1 和 word2，求出从 word1 转换为 word2 的最少步骤。每个转换操作算一步。转换操作限定于：删除一个字符，插入一个字符和替换一个字符。

只是求出最少的步骤，而不是需要具体的转换过程，因此，我们考虑动态规划的方法。二维辅助空间的元素  $dp[i][j]$  代表从  $word1[0...i-1]$  转换为  $word2[0...j-1]$  的最少步骤。

$$dp[i][j] = \begin{cases} dp[i-1][j-1], & \text{当 } word1[i-1] \text{ 字符与 } word2[j-1] \text{ 字符相等} \\ 1 + \min(dp[i-1][j-1], dp[i-1][j], dp[i][j-1]), & \text{当上述条件不成立时,} \\ & \text{取修改, 删除 } word1 \text{ 字符, 删除 } word2 \text{ 字符的最小值} \end{cases}$$

在初始化边界时，特别需要注意空串的时候，当一方是空串时，最少步骤即为另一方的字符个数。

---

```
int editDistance(String word1, String word2) {
    int len1=word1.length(), len2=word2.length();
    if(len1==0)return len2;
    if(len2==0)return len1;
    int dp[][] = new int[len1+1][len2+1];
    //初始化边界，只有 word1 的字符
    for(int i=0;i<=len1;i++)    dp[i][0]=i;
    //初始化边界，只有 word2 的字符
    for(int j=0;j<=len2;j++)    dp[0][j]=j;
    for(int i=1;i<=len1;i++) {
        for(int j=1;j<=len2;j++) {
            if(word1.charAt(i-1)==word2.charAt(j-1))
                dp[i][j]=dp[i-1][j-1];
            else
                dp[i][j]=1+Math.min(dp[i-1][j-1], 修改 //
```

---

---

```
Math.min(dp[i-1][j], dp[i][j-1])); //删除
    }
}
}
return dp[len1][len2];
}
```

---

## 面试题 80：交替字符串 ☆☆☆☆☆

输入三个字符串 s1、s2 和 s3，判断 s3 是否由 s1 和 s2 的字符组成，字符保持原有的相对顺序。

### 举例

假设 s1="abc", s2="def", 如果 s3="adbecf", 返回 true; 如果 s3="abcdee", 返回 false。

二维布尔辅助空间的元素  $dp[i][j]$  代表  $s3[0...i+j-1]$  是否由  $s2[0...i-1]$  和  $s1[0...j-1]$  的字符组成。如果 s2 当前字符（即  $s2[i-1]$ ）等于 s3 当前字符（即  $s3[i+j-1]$ ），而且  $dp[i-1][j]$  为真，那么可以取 s2 当前字符而忽略 s1 的情况， $dp[i][j]$  返回真。如果 s1 当前字符等于 s3 当前字符，并且  $dp[i][j-1]$  为真，那么可以取 s1 当前字符而忽略 s2 的情况， $dp[i][j]$  返回真。其他情况， $dp[i][j]$  返回假。

在初始化边界时，我们认为空串可以由空串组成，因此  $dp[0][0]$  赋值为 true。

---

```
public boolean isInterleave(String s1, String s2, String s3) {
    int m = s1.length(), n = s2.length(), s = s3.length();
    //如果长度不一致，s3 不可能由 s1 和 s2 组成
    if(m + n != s) return false;
    boolean[][] dp = new boolean[n + 1][m + 1];
    dp[0][0] = true;
    for(int i = 0; i < n + 1; i++){
        for(int j = 0; j < m + 1; j++) {
```

---

---

```

        if(dp[i][j] (i - 1 >= 0 && dp[i - 1][j] == true &&
        //取 s2 字符
        s2.charAt(i - 1) == s3.charAt(i + j - 1)) ||
        ( j - 1 >= 0 && dp[i][j - 1] == true && //取 s1 字符
        s1.charAt(j - 1) == s3.charAt(i + j - 1)))
            dp[i][j]=true;
        else
            dp[i][j]=false;
    }
}
return dp[n][m];
}

```

---

## 面试题 81：最长回文子串 ☆☆☆☆☆

输入一字符串 S，找出其最长回文子串(plindromic substring)。

二维布尔数组的元素  $dp[i][j]$  记录从  $s[i]$  到  $s[j]$  组成的子串是否为回文。 $dp[i][j]$  计算如下：

$$dp[i][j] = \begin{cases} dp[i+1][j-1], & \text{当 } s[i] \text{ 与 } s[j] \text{ 相等时} \\ \text{false}, & \text{当 } s[i] \text{ 与 } s[j] \text{ 不相等时} \end{cases}$$

与以前初始化边界不同，这次初始化对角线：我们认为单个字符是回文，如果是紧挨着的两个相同字符也是回文。

---

```

String longestPalindrome(String s) {
    int n = s.length(), longestBegin = 0, maxLen = 1;
    boolean dp[][] = new boolean[n][n];
    //单个字符是回文
    for (int i = 0; i < n; i++) dp[i][i] = true;

```

---

---

```
//紧挨着的两个相同字符也是回文
for (int i = 0; i < n-1; i++) {
    if(s.charAt(i) == s.charAt(i+1)) {
        dp[i][i+1]=true;
        longestBegin=i;
        maxlen=2;
    }
}
for (int i=n-3; i >= 0; i--) {
    for(int j =i+2; j < n; j++) {
        if(s.charAt(i) == s.charAt(j) && dp[i+1][j-1]) {
            //如果两端字符相同，那取决于内部子串的情况
            dp[i][j]=true;
            longestBegin=i;
            maxlen=j-i+1;
        }
    }
}
//截取回文
return s.substring(longestBegin, longestBegin+maxlen);
}
```

---

动态规划的算法复杂度为  $O(n^2)$ ，还有一种是将每个位置作为中心点遍历的 Manacher 方法 ([http://en.wikipedia.org/wiki/Longest\\_palindromic\\_substring](http://en.wikipedia.org/wiki/Longest_palindromic_substring))，可以达到  $O(n)$ 。面试的时候可以提一下 Manacher 方法，但很多面试官并不知道有这种算法。

## 面试题 82：回文分割 ☆☆☆☆

给出一个字符串  $s$ ，将  $s$  进行分割，使得每个子串均为回文。求最少回文分割数。

### 举例

输入  $s = \text{"abb"}$ ，可能的分割结果有： $[\text{"a"}, \text{"b"}, \text{"b"}]$ ， $[\text{"a"}, \text{"bb"}]$ ，那么其中最少回文分割数为 1，即 $[\text{"a"}, \text{"bb"}]$ 。

一维数组的元素  $dp[i]$  记录从  $s[i]$  到字符串结尾的回文分割数。此外，我们还需要记录子串的回文情况， $isPa[i][j]$  判断从  $s[i]$  到  $s[j]$  的子串是否为回文。 $isPa[i][j]$  计算如下：

$$isPa[i][j] = \begin{cases} isPa[i+1][j-1], & \text{当 } s[i] \text{ 与 } s[j] \text{ 相等时} \\ false, & \text{当 } s[i] \text{ 与 } s[j] \text{ 不相等时} \end{cases}$$

初始化  $dp$  时，我们认为每个字符都是一个回文，所以  $dp[i]$  等于  $len-i+1$ ，即  $i$  到末尾的长度。当  $isPa[i][j]$  为  $true$ ，即  $s[i]$  到  $s[j]$  的子串为回文，那么  $dp[i]$  取  $dp[i]$  与  $dp[j+1]+1$  的较小值。

---

```
int minPalinCut(String s){
    int len = s.length();
    int dp[] = new int[len+1];
    boolean isPa[][] = new boolean[len][len];
    //初始化从 i 到结束的字符串的回文分割数
    for(int i = 0; i <= len; i++){
        dp[i] = len-i+1;
    }
    for(int i = len-1; i >= 0; i--){
        for(int j = i; j < len; j++){
            //判断从 i 到 j 是否为回文，如果是，更新最小分隔数
            if(s.charAt(i) == s.charAt(j) && (j-i<2 || isPa[i+1][j-1])){
                isPa[i][j] = true;
                //dp取 dp[i] 与 dp[j+1]+1 的较小值
                dp[i] = Math.min(dp[i], dp[j+1]+1);
            }
        }
    }
    return dp[0];
}
```

---



## 面试题 83：最大公共子串 ☆☆☆☆

给出两个字符串 s1 和 s2，返回其最大的公共子串。

二维整型数组的元素  $dp[i][j]$  记录的是从 s1 的开始字符到 s1 的第 i 个字符组成的字符串，和从 s2 的开始字符到第 j 个字符组成的字符串，这两个字符串的最大公共子串长度。 $dp[i][j]$  计算如下：

$$dp[i][j] = \begin{cases} dp[i-1][j-1]+1, & \text{当 } s[i] \text{ 与 } s[j] \text{ 相等时} \\ 0, & \text{当 } s[i] \text{ 与 } s[j] \text{ 不相等时} \end{cases}$$

在计算  $dp[i][j]$  的过程中，记录了公共子串的最大长度，同时也记录了在当前最大长度下的公共子串。

---

```
public String LCS(String s1, String s2){
    String res="";
    if(s1==null || s1.length()==0 || s2==null || s2.length()==0)
        return res;
    int max=0, m=s1.length(), n=s2.length();
    int[][] dp = new int[m][n];
    //计算到 s1 的第 i 个字符和 s2 的第 j 个字符为止的最大公共子串长度
    for(int i=0; i<m; i++){
        for(int j=0; j<n; j++){
            if(s1.charAt(i)==s2.charAt(j)){
                if(i==0 || j==0) dp[i][j]=1; //边界情况
                else{
                    dp[i][j] = dp[i-1][j-1]+1; //加上当前长度
                }
                //记录最大长度和子串
                if(dp[i][j]>max){
```

---

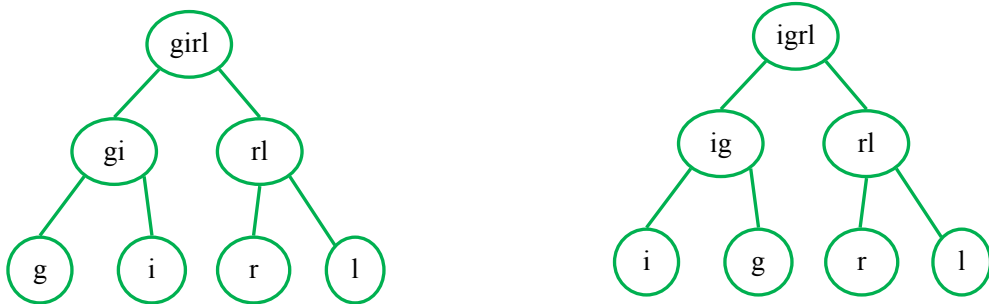
```
        max=dp[i][j];  
        res = s1.substring(i-dp[i][j]+1, i+1);  
    }  
    }  
    }  
    return res;  
}
```

## 面试题 84：字符串洗牌 ☆☆☆☆☆

给定两个相同长度的字符串 s1 和 s2，判断 s2 是否为 s1 翻洗后的字符串。我们使用二叉树来存储字符串，每个节点一分为二，子节点为非空的子串。翻转非叶节点的两个子节点后，组成新的字符串，我们称为翻洗后的字符串。

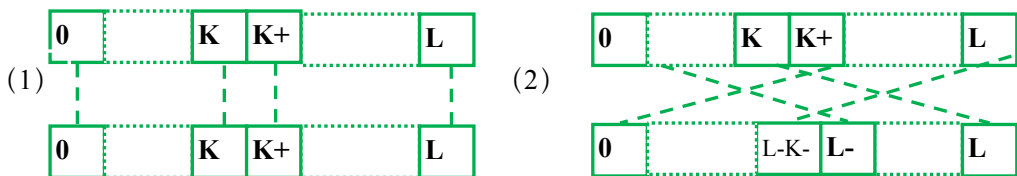
### 举例

字符串“girl”的二叉树如下图的左图所示，翻转“gi”后我们得到“igrl”的二叉树如下面的右图所示。



使用三维布尔数组记录子问题的结果。这是我们到目前为止使用维度最高的辅助空

间。 $dp[i][j][l]$ 代表从字符  $s1[i]$  开始和从字符  $s2[j]$  开始长度各为  $l+1$  的两个子串是否互为翻洗后的字符串。首先初始化长度为 1 的情况, 即  $dp[i][j][0]$ , 仅当  $s[i]=s[j]$  时,  $dp[i][j][0]$  为 true。对于长度大于 1 的情况, 即  $dp[i][j][l]$ , 如果我们能找到  $k, 0 \leq k < l$ , 使得  $s1[i \dots i+k]$  与  $s2[j \dots j+k]$  互为翻洗后的字符串, 并且  $s1[i+k+1 \dots l]$  与  $s2[j+k+1 \dots l]$  互为翻洗后的字符串, 那么可以认为  $s1[i \dots i+l]$  与  $s2[j \dots j+l]$  互为翻洗后的字符串, 即  $dp[i][j][l] = dp[i][j][k] \ \&\& \ dp[i+k+1][j+k+1][l-1-k]$ 。或者, 如果  $s1[i \dots i+k]$  与  $s2[l-k \dots l]$  互为翻洗后的字符串, 并且  $s1[i+k+1 \dots l]$  与  $s2[j \dots j+l-k-1]$  互为翻洗后的字符串, 那么可以认为  $s1[i \dots i+l]$  与  $s2[j \dots j+l]$  互为翻洗后的字符串, 即  $dp[i][j][l] = dp[i][j+l-k][k] \ \&\& \ dp[i+k+1][j][l-1-k]$ , 如下图所示:



```
boolean isScramble(String s1, String s2) {
    int n = s1.length();
    if(s2.length() != n) return false;
    boolean dp[][][] = new boolean[n][n][n];
    //初始化长度为 1 的情况
    for(int i=0; i<n; i++)
        for(int j=0; j<n; j++)
            dp[i][j][0] = s1.charAt(i) == s2.charAt(j);
    //计算长度为 l 的情况
    for(int l=1; l<n; l++) {
        for(int i=0; i+l<n; i++) {
            for(int j=0; j+l<n; j++) {
                for(int k=0; k<l; k++) {
                    ((dp[i][j][k] &&
                     dp[i+k+1][j+k+1][l-1-k])
                     || (dp[i][j+l-k][k] &&
                        dp[i+k+1][j][l-1-k])) {
                        dp[i][j][l] = true;
                    }
                }
            }
        }
    }
}
```

---

```
                break;
            //只要找出符合条件的一种情况即可
        }
    }
}

//返回长度为 n 的情况
return dp[0][0][n-1];
}
```

---

# 第 12 章

## 优先遍历

深度优先遍历 (Depth-First-Search) 是沿着树的深度遍历树的节点，尽可能深的搜索树的分支。当节点  $v$  的所有边都被探寻过，搜索将回溯到发现节点  $v$  的那条边的起始节点。整个进程反复进行直到所有节点都被访问为止。广度优先遍历 (Breadth-First-Search) 是从根节点开始，沿着树的宽度遍历树的节点。如果所有节点均被访问，则算法中止。优先遍历经常用在树和图遍历、二维数组保存的图像处理、字符串替换等。

### 面试题 85：填充图像 ☆☆☆☆

假设输入二维数组代表一张位图，元素的不同整数值代表不同颜色，先输入位置  $(x,y)$ ，同时用一个新的颜色替换原有颜色，并且向周边（上下前后四个方向）替换颜色，直到遇上新的颜色为止，以形成一个单色的区域。

对于每个位置做同样的操作，我们可以采用深度优先遍历或者广度优先遍历。对于每个位置，当遇上当前颜色和新颜色不一致时，替换为新颜色，然后遍历其四个方向上的邻居。当遇上当前颜色和新颜色一致时，不做任何操作，直接返回。

---

```
void bucketfill (ArrayList<ArrayList<Integer>> image, int newcolor, int
x, int y){
    int m=image.size(), n=image.get(0).size();
    int oldcolor=image.get(x).get(y);
    if(oldcolor==newcolor) return;//新旧一样
    image.get(x).set(y,newcolor); //设置新颜色
    //向左遍历
    if(x>0 && image.get(x-1).get(y)==oldcolor){
        bucketfill(image,newcolor,x-1,y);
    }
    //向上遍历
    if(y>0 && image.get(x).get(y-1)==oldcolor){
        bucketfill(image,newcolor,x,y-1);
    }
    //向右遍历
    if(x<m-1 && image.get(x+1).get(y)==oldcolor){
        bucketfill(image,newcolor,x+1,y);
    }
    //向下遍历
    if(y<n-1 && image.get(x).get(y+1)==oldcolor){
        bucketfill(image,newcolor,x,y+1);
    }
}
```

---

## 面试题 86：封闭区间个数 ☆☆☆☆

给出一个二维的 0 和 1 矩阵，判断有多少个全是 0 的封闭区域。

找出第一个 0 元素，从四个方向做深度优先遍历；当遇到 0 时，在该位置标记为已访问状态；从第一个 0 元素的四个方向遍历完之后，认为完成了第一个封闭区域遍历，则统计数字加一；接着找下一个未被访问的 0 元素，重复如上过程。

---

```
int countZeroArea(int[][] grid){
    int m=grid.length, n=grid[0].length, count=0;
    int visited[][] = new int[m][n];
    for(int i=0; i< m; i++){
        for(int j=0; j < n; j++){
            //找出一个未访问的 0 元素
            if(visited[i][j]==0 && grid[i][j] == 0){
                DFS(grid, i, j, visited);
                count++;
            }
        }
    }
    return count;
}

private void DFS(int[][] grid, int x, int y, int[][] visited){
    if(visited[x][y]!= 0) return;
    visited[x][y]= 1;
    //向上
    if(x>0 && visited[x-1][y] == 0 && grid[x-1][y] == 0)
        DFS(grid, x-1, y, visited);
    //向下
    if(x<grid.length-1 && grid[x+1][y] == 0 && visited[x+1][y] == 0)
        DFS(grid, x+1, y, visited);
    //向左
    if(y>0 && grid[x][y-1] == 0 && visited[x][y-1] == 0)
        DFS(grid, x, y-1, visited);
    //向右
    if(y<grid[0].length-1 && grid[x][y+1] == 0 && visited[x][y+1] ==
0)
        DFS(grid, x, y+1, visited);
}
```

---


## 面试题 87：填充封闭区间 ☆☆☆☆☆

给出一个二维的画板，含有字符‘X’和‘O’，要求替换被‘X’包围的‘O’为‘X’。

### 举例

输入如下左边二维画板，替换被 X 包围的 O 后，如下右图所示：

O	X	X	X
X	O	O	X
O	X	O	X
O	X	X	X



O	X	X	X
X	X	X	X
O	X	X	X
O	X	X	X

我们可以换种思路，与其求出被 X 包围的 O，还不如求那些没有被 X 包围的 O，然后对这种类型的 O 做特殊的标记。最后，重新扫描一遍画板，遇到 O 替换成 X，而遇到特殊标记时，替换为 O。

通过观察我们知道，在外围的 O 是不可能被替换的，因为它们不会被 X 包围。因此，先找出外围的 O，然后从当前位置开始向右、向下、向左、向上四个方向进行深度优先遍历，遇到 X 时停止，而遇到 O 时，做特殊标示并继续遍历。

```
void fillRegions(char[][] board) {  
    int m = board.length, n = board[0].length;  
    // 顶部的 O 开始遍历  
    for(int j = 0; j < n; ++j)
```



---

```
        if(board[0][j] == 'O')
            DFS(board,0,j);
//右侧的O开始遍历
for(int i = 1; i < n; ++i)
    if(board[i][m-1] == 'O')
        DFS(board,i,m - 1);
//底部的O开始遍历
for(int j = 0; j < m - 1; ++j)
    if(board[n-1][j] == 'O')
        DFS(board,n - 1, j);
//左侧的O开始遍历
for(int i = 1; i < n - 1; ++i)
    if(board[i][0] == 'O')
        DFS(board,i,0);
// 把内部O替换为X, 外部#替换成O
for(int i=0; i<m; i++)
    for(int j=0; j<n; j++)
        if(board[i][j] == 'O')
            board[i][j] = 'X';
        else if(board[i][j] == '#')
            board[i][j] = 'O';
}

private void DFS(char[][] board, int i, int j) {
    board[i][j] = '#';
    if(i - 1 >= 0 && board[i-1][j] == 'O')
        DFS(board,i - 1, j); //向上
    if(j + 1 < board[0].length && board[i][j+1] == 'O')
        DFS(board,i, j + 1); //向右
    if(i + 1 < board.length && board[i+1][j] == 'O')
        DFS(board,i + 1, j); //向下
    if(j - 1 >= 0 && board[i][j-1] == 'O')
        DFS(board,i, j - 1); //向左
}
```

---

## 面试题 88：单词查找 ☆☆☆

输入一个含有字符的二维画板和一个单词，判断这个单词是否在画板里。单词由画板相邻的字符组成（上下相邻或左右相邻）。每个位置的字符只能用一次。

### 举例

输入如下二维画板，单词“DOG”和“DOGGY”均在画板里。

D O		G
A Y		G
C D		B

对画板的每个位置进行深度优先遍历，查找单词的字符。每个位置有四个方向：向上、向下、向左和向右。此外，我们还需要一个二维布尔数组记录每个位置的访问状态。

---

```
public boolean findWord(char[][] board, String word) {
    if(board == null || board.length == 0 || board[0].length == 0)
        return false;
    //二维布尔数组记录每个位置的访问状态
    boolean[][] visited = new boolean[board.length][board[0].length];
    for(int i = 0; i < board.length; i++)
        for(int j = 0; j < board[i].length; j++)
            //从每个格子开始找
            if(existDfs(board, j, word, visited))
                return true;
    return false;
}
```

---

```
boolean existDfs(char[][] board, int i, int j, String word, boolean[][] visited){
    if(word.length() == 0 || word.length() == 1 && word.charAt(0) == board[i][j]){
        return true;
    }
    //比较当前第一个字符
    if(word.charAt(0) != board[i][j]) return false;
    visited[i][j] = true;
    //截取第一个字符之后的子串
    String subWord = word.substring(1);
    if(i + 1 < board.length && !visited[i + 1][j]) //向下
        if(existDfs(board, i + 1, j, subWord, visited)) return true;
    if(i - 1 >= 0 && !visited[i - 1][j]) //向上
        if(existDfs(board, i - 1, j, subWord, visited)) return true;
    if(j + 1 < board[i].length && !visited[i][j + 1]) //向右
        if(existDfs(board, i, j + 1, subWord, visited)) return true;
    if(j - 1 >= 0 && !visited[i][j - 1]) //向左
        if(existDfs(board, i, j - 1, subWord, visited)) return true;
    visited[i][j] = false; //重置
    return false;
}
```

## 面试题 89：单词变换 ☆☆☆☆

给出两个英文单词，和一个字典，求从第一个单词到第二个单词的最少转换路径长度。要求一次转换只能改变一个字母，而且每次转换后的单词都需要出现在字典里。如果不存在转换路径，则返回 0。

### 举例

两个单词“hit”和“cog”，字典为[“hot”，“dot”，“dog”，“lot”，“log”]，那么从“hit”到“cog”的最少转换路径为：“hit”→“hot”→“dot”→“dog”→“cog”，即最小转换长度为 5。

为了求最短转换路径，在这里我们使用广度优先遍历：使用两个队列，分别记录当前遍历的节点，和下一轮需要遍历的节点（即本层节点的下一层子节点）。遍历时，如果遇到目标单词，那么直接返回遍历的层数。

如何获取每个节点的下一层子节点呢？即如何获取单词的一次转换/变体后的单词呢？单词的每个字符都可以替换成其他 25 个字符，选出那些只在字典里出现的变体。

---

```

int wordLadder(String start, String end, HashSet<String> dict){
    Queue<String>currQue = new LinkedList<String>();
    HashSet<String>used = new HashSet<String>();
    //加上初始单词
    used.add(start);
    currQue.add(start);
    int count=1;
    if(start.equals(end))return count; //两者相等的情况
    while(currQue.size()> 0){
        Queue<String>nextQue = new LinkedList<String>();
        count++;
        while(currQue.size()> 0){
            String word = currQue.poll();
            //获取单词的所有变体
            for(String transform:
                getTransform(word, used, dict, end)){
                if(transform.equals(end)){
                    //找到最终变体
                    return count;
                }
            }
            //加入下一轮的单词
            nextQue.add(transform);
        }
        //复制下一轮的结果
        currQue = nextQue;
    }
    return 0;
}

```

---

```
ArrayList<String> getTransform(String word, HashSet<String> used,
                                HashSet<String> dict, String end){
    char[] letters = word.toCharArray();
    ArrayList<String> res = new ArrayList<String>();
    for(int i=0; i< letters.length; i++){
        //每个字母做替换
        char original = letters[i];
        for(char c='a'; c <='z'; c++){
            if(original==c){
                letters[i]=c; //替换字符
                String trans = new String(letters);
                if(trans.equals(end))
                    (!used.contains(trans) && dict.contains(trans)){
                        //如果变为第二个单词，
                        //或者它在字典里，但又没使用过，则加入
                        used.add(trans);
                        res.add(trans);
                    }
            }
        }
        //恢复原来字符
        letters[i]=original;
    }
    return res;
}
```

## 面试题 90：单词替换规则 ☆☆☆☆

输入字符串和一组变换规则，输出所有通过变换规则之后的字符串。

### 举例

输入字符串“face”，变换规则如：‘a’→‘@’，‘e’→‘3’，‘e’→‘E’等。输出：“f@c3”，

“fac3”, “facE”等。

与前一道题“单词变换”类似,从字符串的第一个字母开始,获取每个字母的变体,加上字母本身(即不做变换的情况)做深度优先遍历,直至最后一个字母。为了避免重复输出结果,需要通过一个不含重复元素的集合做去重。

---

```
void genStringByRule(String s,
    HashMap<Character, ArrayList<Character>> hm){
    HashSet<String> set = new HashSet<String>();
    set.add(s); //不输出自身
    genStringByRule(s, hm, 0, set, "");
}

void genStringByRule(String s,
    HashMap<Character, ArrayList<Character>> hm,
    int start, HashSet<String> set, String op){
    if(start==s.length()){
        //找到一个结果
        if(!set.contains(op)){
            //如果不重复,则输出
            System.out.println(op);
            set.add(op);
        }
        return;
    }
    ArrayList<Character> mutations = hm.get(s.charAt(start));
    if(mutations!=null){
        for(Character c: mutations){
            //递归调用每种字符变换
            if(c!=null)genStringByRule(s, hm, start+1, set, op+c);
        }
    }
    //还要加上保留当前字符的情况
    genStringByRule(s, hm, start+1, set, op+s.charAt(start));
}
```

---

## 面试题 91：有向图遍历 ☆☆☆☆

给出一个有向图，从 A 节点走到 B 节点，正好走 N 步，有多少种走法？走过的节点可以重复走。

假设有向图的节点定义如下：

```
class GraphNode{
    int val;
    ArrayList<GraphNode> nexts;
}
```

在这里我们使用广度优先遍历：使用两个队列，分别记录当前遍历的节点，和下一轮需要遍历的节点（即本层节点的下一层子节点）。从 A 节点开始遍历，每遍历完一层时，步数加一；如果遍历遇到了 B 而且步数恰好为 N，那么走法数相应加一。遍历时并未记录节点的访问情况，这样可以满足走过的节点可以重复走。

---

```
int getGraphPaths(GraphNode A, GraphNode B, int N){
    Queue<GraphNode> currQueue = new LinkedList<GraphNode>();
    currQueue.add(A);
    int count=0, steps = 0;
    while(currQueue.size()> 0 && steps <= N){
        Queue<GraphNode> nextQueue = new LinkedList<GraphNode>();
        steps++;
        while(currQueue.size()> 0){
            GraphNode node = currQueue.poll();
            //获取下一个节点
            for(GraphNode next: node.nexts){
                if(next==B && steps == N){
                    //找到一条路径
                    count++;
                }
            }
        }
    }
}
```

---

---

```
                //同一轮可能含有重复节点
                nextQue.add(next);
            }
        }
        //复制下一轮的结果
        currQue = nextQue;
    }
    return count;
}
```

---



# 第 13 章

## 哈希

哈希表通常用在要求常数级时间查找，以及字符或数字去重的解题方法中。

### 面试题 92：最长连续序列 ☆☆☆☆

给出一个无序的整型数组，找出最长连续元素序列的长度。时间复杂度要求在线性时间内。

#### 举例

输入 {8, 1, 9, 3, 2, 4}，那么其最长连续序列是 {1, 2, 3, 4}，即输出长度 4。

使用一个哈希表记录数组每个整数所在的连续序列的长度，初始化时为 1。从输入数组的第一个元素开始扫描，插入哈希表后，判断其前一个数（即比该数小一的数）和后一个数（即比该数大一的数）是否已经存在哈希表内了。如果存在，更新连续序列的

左右两端的数对应哈希表的值，即所在序列的长度值。由于哈希表的查找和插入操作都是  $O(1)$ ，所以算法的复杂度即为扫描整个数组的时间，即为  $O(n)$ 。

---

```
public int findLongestConsequence(int[] A) {
    HashMap<Integer,Integer> map = new HashMap<Integer, Integer>();
    intmax = 1;
    for(int i : A) {
        ifmap.containsKey(i)) continue; //忽略重复值
        map.put(i);
        ifmap.containsKey(i - 1)) { //连接左边元素
            maxMath.max(max, merge(map, i-1, i));
        }
        ifmap.containsKey(i + 1)) { //连接右边元素
            maxMath.max(max, merge(map, i, i+1));
        }
    }
    returnmax;
}

private int merge(HashMap<Integer, Integer> map, int left, int right) {
    int upper = right + map.get(right) - 1; //最右边数
    int lower = left - map.get(left) + 1; //最左边数
    int len = upper - lower + 1; //连续的长度
    map.put(upper,len); //更新右边界
    map.put(lower,len); //更新左边界
    returnlen;
}
```

---

## 面试题 93：变位词 ☆☆☆

给出一组字符串，按组返回拥有相同变位词（anagram）的字符串。

如何判断两个单词有相同变位词？我们可以对单词中的字母排序。如果排序后的单

词是一样，那么我们可以断定这两个单词有相同的变位词。首先，求出每个单词的变位词(anagram)，以变位词作为键插入哈希表，值为一个链表。然后，把该单词附在这个链表末端。最后，遍历哈希表的值，输出链表长度大于 1 的字符串。

---

```
ArrayList<ArrayList<String>> anagrams(ArrayList<String> strs){
    ArrayList<ArrayList<String>> res =
        new ArrayList<ArrayList<String>> ();
    HashMap<String,ArrayList<String>> hm =
        new HashMap<String, ArrayList<String>>();
    for(String str: strs){
        char[] letters = str.toCharArray();
        //排序求出 anagram
        Arrays.sort(letters);
        String anagram = new String(letters);
        if(hm.containsKey(anagram)){
            //如果已经存在，插入到列表里
            ArrayList<String> list = hm.get(anagram);
            list.add(str);
            hm.put(anagram,list);
        }else{
            //新建一项
            ArrayList<String> list = new ArrayList<String>();
            list.add(str);
            hm.put(anagram,list);
        }
    }
    for(ArrayList<String> list: hm.values()){
        //只统计元素个数大于 1 的组
        if(list.size() > 1) res.add(list);
    }
    return res;
}
```

---

## 面试题 94：最长不同字符的子串 ☆☆☆☆

输入一个字符串，求不含有重复字母的最长子串的长度。

### 举例

输入字符串“aaaa”，其不含有重复字母的最长子串为“a”，输出长度为 1。

一般来说，哈希表的键值记录字母和该字母出现的次数。但这次光是记录出现的次数还不够，需要记录字母出现的位置。此外，为了记录长度，我们还需要最长子串的开始位置。每当扫描字符串的字符时，如果该字符已经出现过，而且出现的位置在开始位置之后，更新开始位置。然后，更新哈希表的值，并且保存最长子串的长度。

---

```
int lengthOfLongestSubstring(String s) {
    int n=s.length(), start=0, maxLen =0;
    //使用哈希表记录字符最新出现的位置
    HashMap<Character, Integer> hm = new HashMap<Character, Integer> ();
    for(int i=0; i<n; i++){
        char c = s.charAt(i);
        if(hm.containsKey(c)&& hm.get(c) >= start){
            //更新出现的位置
            start=hm.get(c) +1;
        }
        hm.put(c,i);
        maxLen= Math.max(maxLen, i-start+1);
    }
    return maxLen;
}
```

---

## 面试题 95：最小字符窗口 ☆☆☆☆

给出两个字符串 S 和 T，求 S 的最小子串（或者最小字符窗口），该子串包含所有 T 的字符。要求在线性时间内完成。如果没有符合要求的子串，则返回“”。如果有多个满足条件的最小子串，则返回第一个。

### 举例

输入 S=“AFEGCABC”，和 T=“FACE”，则输出最小子串“AFEGC”。

由于目标串 T 可能含有重复字符，因此使用一个哈希表记录目标串的字符和字符出现的次数。此外，我们至少需要三个变量：已匹配的总字符数，最小字符窗口的开始位置，已匹配的字符和次数。当已匹配的总字符数大于目标串的字符数时，我们尽可能把窗口的开始位置往后移动，使得窗口尽可能小，但是在移动开始位置时必须满足如下条件：当前字符不在目标串 T 里，或者已匹配的字符数大于目标串里相应字符出现的次数。

---

```
String minWindow(String s, String t){
    if(t==null || t.length() == 0) return "";
    int sLen = s.length(), tLen = t.length();
    //记录目标字符串中每个不同字符的个数
    HashMap<Character, Integer> toFind =
        new HashMap<Character, Integer>();
    for(int i=0; i < tLen; i++){
        char c = t.charAt(i);
        if(toFind.containsKey(c)){
            toFind.put(c, toFind.get(c)+1);
        }else{
            toFind.put(c, 1);
        }
    }
    //记录在原串的字符个数
    HashMap<Character, Integer> hasFound =
```

---

---

```

        new HashMap<Character, Integer>());
    int count=0, start=0, minLen=Integer.MAX_VALUE, minLenStart=-1;
    for(int i=0; i<sLen; i++){
        char ch = s.charAt(i);
        // 忽略不在目标串里的字符
        if(!toFind.containsKey(ch)) continue;
        if (hasFound.containsKey(ch)) {
            hasFound.put(ch, hasFound.get(ch)+1);
        } else {
            hasFound.put(ch, 1);
        }
        if (hasFound.get(ch) >= toFind.get(ch)) {
            count++; // 统计已有的字符个数
        }
        if (count >= tLen) {
            char c = s.charAt(start);
            while (hasFound.containsKey(c) && false ||
                    hasFound.get(c) > toFind.get(c)) {
                if (hasFound.containsKey(c))
                    hasFound.put(c, hasFound.get(c)-1);
                start++; // 在满足这两个的条件下, 尽可能往右移动
                c = s.charAt(start);
            }
            int len = i - start + 1;
            if (len < minLen) { // 保存最小窗口
                minLen = len;
                minLenStart = start;
            }
        }
    }
    if (minLenStart >= 0)
        return s.substring(minLenStart, minLenStart+minLen);
    else return "";
}

```

---

## 面试题 96：单词拼接 ☆☆☆☆☆

输入字符串 S 和列表 L，L 含有一组长度相同的单词。找出所有子串的开始下标，该子串由 L 的所有单词拼接而成，而且没有夹杂其他字符。

### 举例

输入 S="barfoothefoobarman"，和 L={"foo", "bar"}，输出下标 {0, 9}，找到两个符合要求的子串 "barfoo" 和 "foobar"。

与上一道题“最小字符窗口”类似，我们需要一个哈希表记录 L 里每个单词的出现次数。从原串 S 的第一个字符开始，取一定长度的子串。其实这个长度我们可以通过计算获得：L 的元素个数乘以 L 里的单词长度。这样就可以把问题转化为：给定一个字符串，判断该串是否仅仅包含了 L 的所有单词。

将子串拆成多个长度和 L 单词长度一致的单词，然后根据哈希表来匹配子串的单词，如果单词匹配成功，则匹配数加一。如果最后的匹配数等同于 L 的元素个数，那么我们认定该串仅仅包含了 L 的所有单词，而且把该串的开始位置放入结果集中，继续考察 S 的下一个字符开始的子串情况。

---

```
public ArrayList<Integer> findSubstring(String S, String[] L) {
    ArrayList<Integer> res = new ArrayList<Integer>();
    int tokenLength = L[0].length();
    int tokenCount = L.length;
    int length = tokenCount * tokenLength;
    //记录数组单词的情况
    HashMap<String,Integer> toFind = new HashMap<String, Integer>();
    for(String str : L) {
        if(!toFind.containsKey(str))
```

---

---

```

        toFind.put(str);
    } else {
        toFind.put(str, toFind.get(str) + 1);
    }
}
for(int i = 0; i < S.length() - length + 1; i++) {
    //先取出相同长度的字符串
    String str = S.substring(i, i + length);
    int count=0;
    HashMap<String,Integer> leftMap =
        new HashMap<String, Integer>( toFind);
    for(int j = 0; j < tokenCount; j++) {
        //读取每个单词
        String token = str.substring(j * tokenLength,
                                     (j + 1) * tokenLength);
        if(!leftMap.containsKey(token))
            break; //不在目标数组 T 里
        else {
            //对应单词的频率减一，获取的单词量加一
            leftMap.put(token, leftMap.get(token)-1);
            if(leftMap.get(token) >= 0) count++;
            else break;
        }
    }
    if(count== tokenCount){
        res.add(i);
    }
}
return res;
}
}

```

---

## 面试题 97：常数时间插入删除查找 ☆☆☆

设计一数据结构，使插入、删除、搜索、随机访问都是常数时间。



单是哈希表或者数组的数据结构，均无法满足题目要求。哈希表做不到随机访问是常数级，而数组的数据结构无法做到在常数时间搜索数组元素。因此，我们可以考虑结合两者，使得数组 A 保存元素，而哈希表 HM 的键存储数组元素，其值为元素在数组的下标。

**插入：**把元素 value 放在数组末端。假设 i 为新元素的下标，那么  $HM[value] = i$ 。

**删除：**使用数组的最后一个元素来替换待删除的元素，通过哈希表获取待删除的位置，替换成最后一个元素，同时更新替换后元素的下标。最后，将数组大小减少一，并移除待删除元素在哈希表的键值。

**搜索：**搜索哈希表，判断是否含有此键。

**随机访问：**获取随机值，返回数组的下标为随机值的元素。

## 面试题 98：对数时间范围查询 ☆☆☆☆

设计一个键与值数据结构，至少实现如下两个查找功能：查找单个 key，以及支持在给出 key 范围内查找。

数组在存储和管理有序元素时不是很有效，因此，我们可以考虑使用二叉搜索树 (BST)。哈希表记录键-值对，而且二叉搜索树只保存键。当查找单个键时，我们只要在哈希表查找该键，如果存在，则返回其对应的值，这样可以在  $O(1)$  时间内完成。当查找给定 key 范围内的值时，先在二叉搜索树找出范围内的 key 值列表，再从哈希表里找出那些 key 对应的值的列表。

## 面试题 99：实现 LRU 缓存 ☆☆☆☆

实现 LRU 缓存类，至少包含 get 和 put 成员函数。get 和 put 算是对元素的一次访问。

设计思路：使用一个双向队列（含头尾空元素）保存元素，并记录元素的访问顺序；再利用一个哈希表，使得查找元素更便捷。

首先，设计元素类，包含了键与值以及双向指针。

---

```
class Item<K,V>{
    Kkey;
    Vvalue;
    Itemprev, next;
    Item(K key, V value){ this.key = key;
        this.value = value;prev=null; next=null;}
    Item(){key=null;value=null; prev=null; next=null;}
}
```

---

其次，设计 LRUCache 类，包含了队列和哈希表。

---

```
class LRUCache{
    Itemm_start, m_end;
    intm_size, curr_size;
    HashMap<K,Item>m_hm;
    LRUCache(intsize){
        //初始化
        m_size= size;
        curr_size=0;
        m_start=newItem();
        m_end=new Item();
        //首尾相接的空队列
```

---

---

```
        m_start.next=m_end;
        m_end.prev= m_start;
        m_hm= new HashMap<K,Item>();
    }
}
```

---

然后，实现 `get()` 方法：先判断哈希表是否存在该 `key`，如果存在，我们还不能直接返回对应的 `value`，因为 `get` 也算是对元素的一次访问，我们需要把该元素移动到双向队列的头部。

---

```
V get(K key) {
    if(m_hm.containsKey()){
        //移动元素至队列头部
        moveToHead(m_hm.get(key));
        return m_hm.get(key).value;
    }else{
        return null;
    }
}

void moveToHead(Item item){
    Item cross = m_start;
    //找出元素的前一个节点
    while(cross.next!=null && cross.next!=m_end &&
        cross.next.key!=item.key)
        cross= cross.next;
    if(cross.next!=null && cross.next.key == item.key){
        //连接该元素的前后节点
        Item next= item.next;
        cross.next= next;
        next.prev= cross;
        //insert to head
        insertHead(item);
    }
}

void insertHead(Item item){
    //插入到队列头部
```

---

---

```
        Item next = m_start.next;
        m_start.next = item;
        item.next = next;
        item.prev = m_start;
        next.prev = item;
    }
```

---

最后，实现 put()方法。

---

```
void put(K key, V value){
    Item item = new Item(key, value);
    if(m_hm.contains(key)){
        //如果已经存在了，则直接移动到队列头部
        moveToHead(item);
    }else{
        if(curr_size < m_size){
            //插入新元素
            insertHead(item);
            curr_size++;
        }else{
            //队列满了，删除队尾，然后再插入新元素
            removeItem();
            insertHead(item);
        }
    }
    m_hm.put(key, item);
}

void removeItem(){
    //删除队尾的元素
    Item item = m_end.prev;
    if(item != m_start){
        Item prev = item.prev;
        prev.next = m_end;
        m_end.prev = prev;
        m_hm.remove(item.key);
    }
}
```

---

如果面试官能接受的话，我们还可以继承现成的实现 LRU 的 Java 类 LinkedHashMap。

## 面试题 100：经过最多点的直线 ☆☆☆

给出一个二维的图，图上有很多点，找出一条穿过最多点的直线。

解题思路：两点可组成一条直线，把图上的每两个点组成的直线计算出来，然后聚合，把相同直线聚合在一起，返回穿过最多点的直线。

首先，设计直线。直线包含了斜率，以及与 x 轴相交的点。在 Java 类中，判断两个对象相等，必须重写 hashCode 和 equals 两个函数。如果斜率相等，以及与 X 轴相交的点相等，我们认为两个直线是相同的。

---

```
public class Line {
    double epsilon = .0001;
    double slope;
    double intercept;
    boolean infiniteSlope = false;
    public Line(Point p, Point q) {
        if(Math.abs(p.x - q.x) > epsilon) {
            //对于不和 Y 轴平行的直线，计算斜率和 x 轴的交点
            slope = (p.y - q.y) / (p.x - q.x);
            intercept = p.y - slope * p.x;
        } else {
            //与 Y 轴平行的直线
            infiniteSlope = true;
            intercept = p.x;
        }
    }
    public boolean isEqual(double a, double b) {
```

---

---

```

        return(Math.abs(a - b) < epsilon);
    }
    @Override
    public int hashCode() {
        //根据斜率和交点计算哈希值
        int sl = (int)(slope * 1000);
        int in = (int)(intercept * 1000);
        return sl | in;
    }
    @Override
    public boolean equals(Object o) {
        Line l = (Line) o;
        if(isEqual(l.slope, slope) && isEqual(l.intercept, intercept)
            && (l.infiniteSlope == l.infiniteSlope)) {
            //如果斜率相同，与x轴交点相同，那么我们就认为这两条直线相同
            return true;
        }
        return false;
    }
}

```

---

其次，使用哈希表记录两点组成的直线的出现次数。出现次数最多的线，即为穿过最多点的直线。

---

```

Line findBestLine(Point[] points) {
    Line bestLine = null;
    //记录每条直线的出现次数
    HashMap<Line, Integer> lineCount = new HashMap<Line, Integer>();
    for(int i = 0; i < points.length; i++) {
        for(int j = i + 1; j < points.length; j++) {
            Line line = new Line(points[i], points[j]);
            //更新或插入每条线出现的次数
            if(!lineCount.containsKey(line)) {
                lineCount.put(line, 1);
            }
            lineCount.put(line, lineCount.get(line) + 1);
        }
    }
}

```

---

---

```
        if (bestLine == null ||
            lineCount.get(line) > lineCount.get(bestLine)) {
            //保存出现次数最多的直线
            bestLine = line;
        }
    }
    return bestLine;
}
```

---

# 第 14 章

## 堆栈

通常我们也会使用堆、栈以及队列来解决字符串变换问题，数据流的中值和最大值，以及二叉树的遍历相关的面试题。

### 面试题 101：局部最大值 ☆☆☆

输入一个长数组  $A$ ，以及窗口大小  $w$ ，输出一个数组  $B$ ， $B[i]$ 代表移动窗口  $A[i] \dots A[i+w-1]$ 的最大值。

#### 举例

输入  $A = \{1, 3, -1, -3, 5, 3, 6, 7\}$ ，窗口大小  $w=3$ ，输出如下：



移动窗口	最大值
[1 3 -1] -3 5 3 6 7	3
1 [3 -1 -3] 5 3 6 7	3
1 3 [-1 -3 5] 3 6 7	5
1 3 -1 [-3 5 3] 6 7	5
1 3 -1 -3 [5 3 6] 7	6
1 3 -1 -3 5 [3 6 7]	7

解题思路：使用一个队列，队列的头部保存当前最大值。Java 的 `PriorityQueue` 既可以当 `Queue` 使用，也可以当堆来使用，但是默认情况是小的元素排在前面，大的元素排在后面。我们需要设置自己的排列顺序。在移动窗口时，如果队列的头元素不在窗口内，我们将移除队列的头元素，直至它在移动窗口内。

```
class Pair{
    int val;
    int index;
    Pair(int v, int i){
        val = v;
        index = i;
    }
}

void maxSlidingWindow(int A[], int w, int B[]) {
    int n = A.length;
    Comparator< Pair > comparator = new Comparator< Pair >() {
        @Override
        public int compare(Pair n1, Pair n2) {
            // 设定排序规则，大的在前，小的在后
            if (n1.val < n2.val)
                return 1;
            else if (n1.val > n2.val)
                return -1;
            else if (n1.index > n2.index)
                return 1;
            else
                return -1;
        }
    };
}
```

---

```

        else if (n1.index < n2.index)
            return
        else
            return
    }
};
//队列头部保存最大值
PriorityQueue<Pair>Q = new PriorityQueue<Pair>(w, comparator);
for (int i = 0; i < w; i++)
    Q.offer(new Pair(A[i], i)); //初始化前 w 个
for (int i = w; i < n; i++) {
    Pair p = Q.peek();
    B[i-w] = p.val;
    while (p.index <= i-w) {
        //移除不在窗口内的队头
        Q.poll();
        p = Q.peek();
    }
    Q.push(new Pair(A[i], i));
}
B[n-w] = Q.peek().val;
}

```

---

## 面试题 102：数据流最大值 ☆☆☆☆

某一刻开始从无限的整数流(infinite integer streaming)取最大值。

起初看起来觉得无从下手。其实我们可以问面试官：如何获取这些无限的整数流？从某一刻开始保存这些数据流的时候，使用一个队列来记录最大值：每当加入一个整数时，从队列尾部开始扫描，移除比当前值小或相等的元素（因为用不上），这样队列头保存当前最大值。因此，某一刻开始的最大值，可能在常数时间内获得。

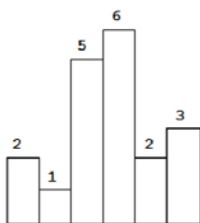
```
ArrayList<Integer> list= new ArrayList<Integer>();  
void add(int num){  
    for(int i=list.size()-1; i>=0; i--){  
        //列表尾部开始扫描，移除比当前值小或相等的元素  
        if(num>=list.get(i)){  
            list.remove(i);  
        }  
    }  
    list.add(num);  
}  
int getMax(){  
    //取队列头部元素  
    return list.size()>0 ? list.get(0): Integer.MIN_VALUE;  
}
```

## 面试题 103：最大四边形 ☆☆☆☆☆

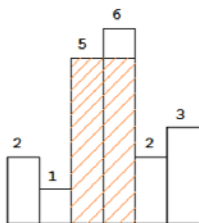
输入 N 个非负整数，分别代表宽度为 1 高度为整数值的柱子，现从柱状图里找出最大四边形的面积。

### 举例

输入数组 {2, 1, 5, 6, 2, 3}，组成的柱状图如下左图所示，而其最大四边形面积为 10，如下右图所示：



宽度为 1，高度为元素值的柱状图



阴影构成的正方形最大，其面积为 10

### 观察

较矮的柱子和其他柱子才能组成一个四方形。比较两个相邻的柱子：如果后面柱子比前面的矮，例子中的 2 和 1 相邻，1 比 2 矮，柱子 2 在以后计算中已经没什么用，因为后面的柱子不会和 2 组成一个四方形了，这时候可以计算柱子 2 和之前柱子组成的四方的最大面积；如果后面柱子比前面的高，例子中的 1 和 5 相邻，5 比 1 高，这时候，这两根柱子在以后可以和其他柱子组成四方形。

通过观察我们知道，如果遇到上升的柱子，可以继续往下走；如果遇到下降的柱子，可以计算之前的四方的最大面积，因为之前的柱子在今后已经没什么用了。

首先，通过一个栈记录上升的柱子，如果遇到下降的柱子，可以开始计算栈顶和之前柱子构建的四方的面积。栈保存的是柱子的下标，而不是柱子的高度，目的是方便计算四方的宽度。遇到上升的柱子，把该柱子对应的下标压入栈。

最后，扫描完成时，如果栈还有元素，说明该柱子到结尾可以组成一个四方形，即宽度为  $N - \text{bars.peek}() - 1$ ， $\text{bars.peek}()$  为柱子所在的下标。

---

```
int maxHistogramArea(int[] height){
    int n=height.length, max=0;
    Stack<Integer>bars = new Stack<Integer>();
    bars.add(-1); //为了方便计算相对距离
    for(int i=0; i<n; i++){
        int prev = bars.peek();
        if(prev<0 || height[i] >= height[prev]){
            //压入上升序列的下标
            bars.add(i);
        }else{
            //一旦下降了，开始计算柱子的面积
            prev=bars.pop();
            max=Math.max(max, height[prev]*(i-bars.peek()-1));
            i--; 还不能确定当前元素是否在上升序列里
        }
    }
}
```

---

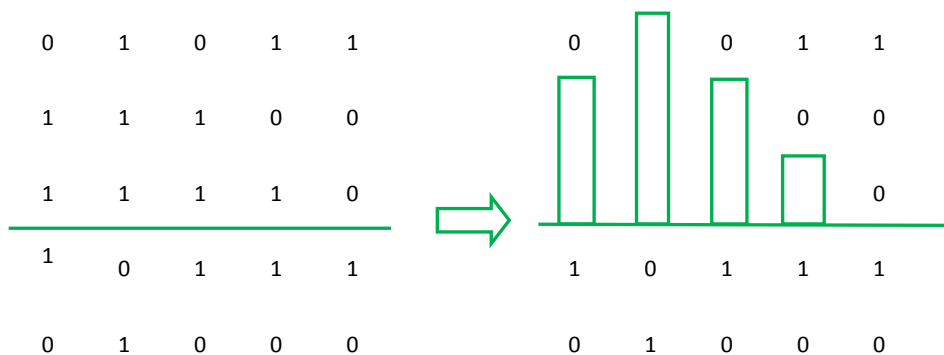
```
}  
//计算还未计算的上升序列的面积  
while(bars.peek() != -1) {  
    int prev = bars.pop();  
    max = Math.max(max, height[prev] * (n - bars.peek() - 1));  
}  
return max;  
}
```

### 扩展问题

给出二维 0 和 1 矩阵，找出最大包含 1 的四方形面积。

假如所有 1 可以在矩阵最后一行连成一个柱状图，那么我们可以把问题转化为“最大四方形”的最大面积。换一个思路来看待这个问题，我们可以逐行扫描，每行求出最大四边形的面积。例如，如下图所示，我们将第三行转为柱状图：

接下来问题转为如何求出每行的柱状图：按列扫描，遇到 0 时，柱子断了，遇到 1 时，柱子高度累加 1。



```
int maximalRectangle(char[][] matrix) {  
    int n = matrix.length;  
    if (n == 0) return 0;  
    int m = matrix[0].length;
```

---

```

        Stack<Integer>bars = new Stack<Integer>();
//记录每个元素上1的高度
        int[][]height= new int[m][n];
        intmax=0;
        for(intj=0; j<n; j++){
            intt=0;
            for(inti=0; i<m; i++){
                if(matrix[i][j]!='1') h++;
                else h=0如果当前元素为0,那就断了
                height[i][j]=h;
            }
        }
//同前一道题“最大正方形”计算最大长方形面积
        bars.add(-1);
        for(inti=0; i<m; i++){
            for(intj=0; j<n; j++){
                intprev = bars.peek();
                if(prev<0 height[i][j] >= height[i][prev]){
                    //压入上升序列的下标
                    bars.add(j);
                }else{
                    prevbars.pop();
                    maxMath.max(max,
                        height[i][prev]* (j-bars. peek()-1));
                    j--;
                }
            }
            //计算还未计算的上升序列的面积
            while(bars.peek() !=-1){
                intprev = bars.pop();
                maxMath.max(max, height[i][prev]*(n-bars. peek()-1));
            }
        }
        returmax;
    }
}

```

---

## 面试题 104：合并多个有序链表 ☆☆☆☆

合并 K 个有序链表，并分析其复杂度。

这道题有很多种解法。最直观的是合并第一个和第二个链表，然后将结果合并到第三个链表，以此类推。假设链表平均长度为 N，那么这 K-1 次合并的复杂度为  $O(NK^2)$ 。

这 K 个链表本身就是有序的，如果我们能从 K 个链表头选出最小值，那么移动最小值的表头，我们就可以按升序获取节点，并形成一个新链表。如何快速从 K 个元素中提取最小值呢？可以考虑使用最小堆。在一个大小为 K 的最小堆里求最小值为常数级时间，插入时间为  $\log K$ ，因此，使用最小堆的算法复杂度降至为  $O(NK\log K)$ 。

---

```
public ListNode mergeKLists(ArrayList<ListNode> lists) {
    if (lists == null || lists.isEmpty())    return null;
    Comparator<ListNode> comparator = new Comparator<ListNode>() {
        @Override
        public int compare(ListNode n1, ListNode n2) {
            //从小到大排列
            if (n1.val < n2.val)
                return -1;
            else if (n1.val > n2.val)
                return 1;
            else
                return 0;
        }
    };
    //建立一个堆，升序排列
    PriorityQueue<ListNode> heap =
        new PriorityQueue<ListNode> (lists.size(), comparator);
    ListNode head = null, cur = null;
    //往堆加入每个链表的表头
```

---

---

```
        for (int i = 0; i < lists.size(); i++) {
            ListNode node = lists.get(i);
            if (node != null)
                heap.add(lists.get(i));
        }
        while (!heap.isEmpty()) {
            if (head == null) { // 找出第一个节点
                head = heap.poll();
                cur = head;
            }
            if (cur.next != null)
                heap.add(cur.next);

            else {
                ListNode newNode = heap.poll();
                cur.next = newNode; // 把输出链表连接起来
                cur = newNode;
            }
            if (cur.next != null)
                heap.add(cur.next);
        }
        return head;
    }
}
```

---

## 面试题 105：产生逆波兰式 ☆☆☆

将逆波兰式转化为中序表达式。

### 举例

输入{"5", "8", "4", "/", "+"}, 输出“(5 + (8 / 4))”。

扫描输入字符数组，如果遇到数字，直接加入栈；如果遇到运算符，第一个操作数为栈顶元素，第二个操作数为中间结果，将三者连接起来形成新的字符串作为新的中间



结果。如果第一次遇到运算符时，无中间结果，可考虑出栈两次，第一次出栈的数作为中间结果。

---

```
String genRPNotation(char[] input){
    int n = input.length;
    if(n==0) return "";
    String result = "";
    Stack<Character> stack = new Stack<Character>();
    for(int i=0; i<n; i++){
        //区分符号和数字
        if(input[i] == '/' || input[i] == '+'
        || input[i] == '*' || input[i] == '-'){
            if(result=="")
                //第一次遇到运算符时两次出栈
                result = stack.pop()+stack.pop();
            char first = stack.pop();
            result = "("+first+input[i]+result+")";
        }else{
            //如果是数字，直接压入栈
            stack.push(input[i]);
        }
    }
    return result;
}
```

---

## 面试题 106：逆波兰式计算 ☆☆☆

给出逆波兰表达式，输出其计算结果。

举例

`{"4", "1", "+", "2", "*"}`  $\rightarrow ((4 + 1) * 2) \rightarrow 10$

`{"5", "8", "4", "/", "+"}`  $\rightarrow (5 + (8 / 4)) \rightarrow 7$

扫描输入字符数组，如果遇到数字字符，将其转化为数字之后直接压入栈；如果遇到运算符，第一个操作数为栈顶元素，第二个操作数为中间结果，计算此运算式的结果作为新的中间结果。如果第一次遇到运算符时，无中间结果，可考虑出栈两次，第一次出栈的数作为中间结果。

---

```
double calculateRPNotation(char[] input){
    int n = input.length;
    if(n==0) return 0.0;
    double result=0.0;
    boolean firstSign=true;
    Stack<Integer> stack = new Stack<Integer>();
    for(int i=0; i<n; i++){
        //区分符号和数字
        if(input[i] == '/' || input[i] == '+'
           || input[i] == '*' || input[i] == '-'){
            if(firstSign){
                result=(double)stack.pop(); //保留精度
                firstSign=false;
            }
            if(!first) first = stack.pop();
            switch(input[i]){
                //分别处理各种符号
                case '/': result= first/result; break;
                case '+': result = first+result; break;
                case '-': result = first-result; break;
                case '*': result = first*result; break;
                default:
                    break;
            }
        }else{
            //压入数字
            stack.push(input[i]-'0');
        }
    }
}
```

---

---

```
    }  
    return result;  
}
```

---

## 面试题 107：简化文件路径 ☆☆☆

给出一个 linux 文件路径，输出其简化结果。

### 举例

“/home/” → “/home”

“/a/./b/../../c/” → “/c”

### 观察

‘.’代表当前位置，‘..’代表返回上一级目录。如果已经在根目录，还要返回上一级目录怎么办？还是在根目录。如何处理连续多个斜杠？比如“/home//foo/”，这种情况，我们忽略多余的斜杠，返回“/home/foo”。

首先，将输入字符串以斜杠拆分成数组。从头扫描该数组，忽略空串、“.”，如果遇到“..”，弹出栈顶；其他情况，压入栈。扫描结束后，如果栈是空的，返回根目录。如果栈还有元素，栈里的元素连同斜杠组成简化后的路径，其中先出栈元素放在路径的后端。

---

```
public String simplifyPath(String path) {  
    if (path == null || path.length() == 0) return "/";  
    Stack<String> stack = new Stack<String>();  
    String[] splits = path.trim().split("/"); //以/拆分成数组  
    for (String s : splits) {  
        if (s == null || s.length() == 0 || s.equals(".")) {  
            // 忽略
```

---

---

```
        else if (s.equals("..")) {
            // 弹出栈顶
            if(stack.size() > 0) stack.pop();
        } else {
            // 其他情况加入栈
            stack.push(s);
        }
    }
    //空则返回/
    if(stack.isEmpty()) return "/";
    StringBuffer buf = new StringBuffer();
    while (!stack.isEmpty()) {
        //栈顶元素后置
        buf.insert(0,stack.pop());
        buf.insert(0,"/");
    }
    return buf.toString();
}
```

---

## 面试题 108：括号验证 ☆☆

给出一个字符串，只含括号：‘(’, ‘)’, ‘{’, ‘}’, ‘[’, ‘]’。判断该输入串是否合法。

### 举例

合法的字符串有“(”, “(){}”等。不合法的有“(}”, “[)”等。

使用一个栈记录尚未配对的括号。从头扫描字符串，当遇到能和栈头配对的括号时，栈头出栈，其他情况均压入栈。哈希表记录配对括号可以让程序更简洁。扫描完成时，如果栈为空，则说明所有括号配对成功，是合法的括号字符串。

---

```
public boolean isBracketValid(String s) {
    //使用哈希表记录配对情况
    HashMap<Character,Character> map =
        new HashMap<Character, Character>();
    map.put('[',']');
    map.put('(',')');
    map.put('{','}');
    //记录尚未配对的括号
    Stack<Character>stack = new Stack<Character>();
    for(int i = 0; i < s.length(); i++) {
        if(stack.isEmpty() && map.get(stack.peek()) == s.charAt(i))
            //配对成功, 栈头出栈
            stack.pop();
        else
            stack.add(s.charAt(i));
    }
    if(stack.isEmpty())
        //所有括号配对成功
        return true;
    else
        return false;
}
```

---

## 面试题 109：最长有效括号 ☆☆☆

输入只含圆括号的字符串，找出最长合法括号子串的长度。

### 举例

“(0” → 2, 最长子串为“(0”

“)00)” → 4, 最长子串为“(00)”

我们也是利用栈记录尚未配对的情况，但是，只是记录尚未配对的括号的意义不是

很大。考虑到计算合法括号子串的长度，我们得记录尚未配对的左括号的下标。如果配对成功，则栈顶出栈；如果栈还有元素，那么有效配对长度为当前位置减去栈顶元素的下标；如果栈为空，我们还需要变量记录有效配对的开始下标，就是说从开始位置到当前位置都是有效的括号字符串。如何设置有效配对的开始下标呢？当配对没成功时，也就是右括号过多时，开始小标可以设为当前位置的下一个位置，即有效配对的可能的开始下标。

---

```
public int longestValidParentheses(String s) {
    int maxLen = 0;
    //记录尚未配对的左括号所在的下标
    Stack<Integer> stack = new Stack<Integer>();
    int left = 0; //记录有效配对的开始下标
    for(int i=0; i < s.length(); i++){
        if(s.charAt(i) == '('){
            stack.push(i);
        }else{
            if(!stack.empty()) left = i+1; //下个位置才可能是有效配对的下标
            else{
                stack.pop();
                //有效配对的开始位置视栈是否为空而定
                if(stack.empty()){
                    maxLen = Math.max(maxLen, i-left+1);
                }else{
                    //非空，则长度取决于栈顶的下标
                    maxLen = Math.max(maxLen, i-stack.peek());
                }
            }
        }
    }
    return maxLen;
}
```

---

## 面试题 110：设计 Min 栈 ☆☆☆☆

设计一个栈的数据结构，使得 min、push 以及 pop 的时间复杂度都是  $O(1)$ 。

### 观察

对于普通的栈，push 和 pop 都是  $O(1)$  的时间，但是求最小值，则需要扫描栈内的所有元素，时间复杂度为  $O(n)$ 。这就说明需要一个辅助的空间来记录最小值，也许是一个变量，也许是另一个栈，也许是数组、哈希表等等。

我们注意到最小值将伴随进出栈的操作而改变，常数空间满足不了这种情况，而需要一个线性的空间，比如说新栈来存储最小值，使得记录的最小值跟随着原有栈顶元素的变化而改变。当新元素入栈时，同时将当前最小值压入辅助栈。如何求出当前最小值呢？辅助栈的栈顶一直是最小值，只要把新元素同它比较，取较小值者作为当前新的最小值。

元素出栈时，辅助栈的栈顶也出栈，这样辅助栈在每次出入站时均保留当前的最小值。如果求栈元素的最小值，只要查看辅助栈的栈顶元素即可。

---

```
class MinStack{
    Stack<Integer>s;
    Stack<Integer>s2; //辅助栈
    MinStack(){
        s = new Stack<Integer>();
        s2 = new Stack<Integer>();
    }
    public void push(int val){
        s.push(val);
```

---

---

```

        //同时往 s2 压入最小值
        if(s2.empty()|| val < s2.peek()) s2.push(val);
        else s2.push(s2.peek());
    }
    public int min() throws EmptyStackException{
        //预防 s2 为空, 抛出异常
        if(s2.empty()) throw new EmptyStackException("");
        return s2.peek();
    }
    public int pop() throws EmptyStackException {
        //预防 s 或 s2 为空, 抛出异常
        if(s2.empty()|| s.empty()) throw new EmptyStackException("");
        s2.pop();
        return s.pop();
    }
}

```

---

## 面试题 111：中序遍历 ☆☆☆

使用非递归的方法中序遍历顺序打印出一个二叉树节点。

我们可以考虑利用一个栈来模仿递归的过程。中序遍历时，先访问左子树，然后是根节点，最后是右子树。初始化时，将从根节点到最左节点路径上的所有节点压入栈，栈顶即为最左节点，打印输出栈顶元素，然后将其右节点子树下到最左节点路径压入栈，模仿递归的过程。

---

```

void inOrderIteratively(TreeNode root){
    if(root== null) return;
    Stack<TreeNode>s = new Stack<TreeNode>();
    pushLeft(s,root); //模仿递归的过程, 把当前最左节点放在栈顶
    while(!s.isEmpty()){
        TreeNode node = s.pop();

```

---



---

```
        //打印输出栈顶
        System.out.print(node.value+"");
        //递归对其右子树进行类似操作
        pushLeft(snode.right);
    }
}
//压入子树的根节点到最左节点的路径
void pushLeft(Stack<TreeNode> s, TreeNode node){
    while(node!= null){
        s.push(node);
        node= node.left;
    }
}
```

---

## 面试题 112：打印路径 ☆☆☆☆

非递归从根节点到叶节点的所有路径。

如果采用递归方式，前序遍历整棵树，同时记录路径，当遇到叶节点时，输出路径。如果采用非递归的方式，则需要一个栈或队列模仿递归过程。初始化时，将树根节点压入栈。查看栈顶节点，如果节点含有左子树，将其左节点压入栈；如果有右子树，什么时候处理？等到处理完了左子树之后，再考虑其右子树。以什么样方式判断遍历完成左子树呢？

如果节点不含有子树，即为叶节点，根据题意，打印输出从根节点到该节点的路径，即倒序输出栈内元素。假设当前栈顶节点为 A，A 为叶节点，同时 A 为 B 的左节点，那么遍历完了 A 节点，A 节点出栈，则已经遍历完成了 B 的左子树，这时候可以开始遍历 B 的右子树（此时 B 还在栈中）。如果 B 的右子树遍历完了，将 B 弹出栈，查看下一个栈顶节点，即 B 的父节点。由此看来，遍历 B 的右子树要么在 B 的左节点出栈时，

要么 B 的左节点为空时。

---

```
void printPathIteratively(TreeNode root){
    Stack<TreeNode> s = new Stack<TreeNode>();
    if(root==null) return;
    s.push(root);
    while(!s.isEmpty()){
        TreeNode node = s.peek();
        if(node.left==null && node.right==null){
            //到了叶节点
            Stack<TreeNode> s2= new Stack<TreeNode>();
            String str=""; //初始化路径
            while(!s.isEmpty()){
                TreeNode p = s.pop();
                str= p.val+" "+str;
                //使用 s2 保存栈中的元素
                s2.push(p);
            }
            System.out.println(str) //输出从根到叶节点
            while(!s2.isEmpty()) s2.push(s2.pop());
            TreeNode curr= s.pop();
            //前序遍历二叉树
            while(!s.isEmpty()){
                TreeNode parent=s.peek();
                //遍历完了左子树，开始遍历右节点
                if(curr==parent.left&& parent.right!=null){
                    s.push(parent.right);
                    break;
                }else{
                    curr=s.pop();
                }
            }
        }
        }else if(node.left!=null){
            //前序遍历
            s.push(node.left);
        }else if(node.left==null){
```

---

---

```
        s.push(node.right);  
    }  
    }  
}
```

---

## 面试题 113：二叉搜索树两点之和 ☆☆☆☆

给出一棵二叉搜索树和一个值  $X$ ，找出两个节点，使之和为  $X$ 。假设各个节点值不同，同时要求辅助空间大小为  $O(\log N)$ ， $N$  为节点数。

### 观察

回顾第一道面试题“两数之和”：先把该数组排序，排序之后，从首尾两端移动，一次移动一端的指针，直至相遇或者找出两个数和为指定的值为止。中序遍历二叉搜索树之和即可得到有序序列，那么我们可以考虑使用两个栈代替两个指针，记录两端的值。

和一维排序数组类似，需要记录两端的指针，这时候使用两个栈，分别记录最左左子树路径 `leftStack`，和最右右子树路径 `rightStack`，即保存前半部分较小值，以及后半部分的较大值。这两个栈符合题目要求，即大小为树的高度 ( $\log N$ )。

如果两个栈 `leftStack` 和 `rightStack` 的栈顶元素之和大于目标值  $X$ ，则说明 `rightStack` 栈顶过大，将其出栈，找出仅次于原栈顶的节点并且将其压入栈。如果栈顶元素之和小于目标值  $X$ ，则说明 `leftStack` 栈顶元素过小，将其出栈，找出其中序遍历的下一个节点压入栈。如果栈顶元素之后等于目标值  $X$ ，则输出这两个节点。

---

```
public void twoNodeSum(TreeNode root, int x) {  
    Stack leftStack = new Stack<TreeNode>();  
    Stack rightStack = new Stack<TreeNode>();  
    TreeNode scan = root;  
    //使用 leftStack 保存最小值部分
```

---

---

```

        while(scan != null) {
            leftStack.add(scan);
            scan = scan.left;
        }
        scan = root;
        //使用 rightStack 保存最大值部分
        while(scan != null) {
            rightStack.add(scan);
            scan = scan.right;
        }
        do {
            if (leftStack.peek().item + rightStack.peek().item > x) {
                //栈顶元素和过大，找出 BST 的次大值
                TreeNode rightElem = rightStack.pop();
                if (rightElem.left != null) {
                    scan = rightElem.left;
                    while(scan != null) {
                        rightStack.add(scan);
                        scan = scan.right;
                    }
                }
            } else if (leftStack.peek().item +
                rightStack.peek().item < x) {
                //俩栈顶和过小，找出 BST 的次小值
                TreeNode leftElem = leftStack.pop();
                if (leftElem.right != null) {
                    scan = leftElem.right;
                    while(scan != null) {
                        leftStack.add(scan);
                        scan = scan.left;
                    }
                }
            } else {
                //找到了
                System.out.println(leftStack.peek().item, "
                    rightStack.peek().item);
            }
        } while (true);
    }
}

```

---

```
        return;  
    }  
    while (leftStack.peek().item < rightStack.peek().item);  
    System.out.println("nofound");  
}
```

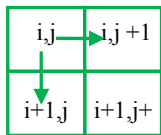
## 面试题 114：矩阵 Top K ☆☆☆☆

给出一个矩阵，各行各列均有序，求第 K 个最小值。

由于每行的首个数并非一定大于上一行的尾数，因此，我们不能像第 21 道题“二维数组搜索”一样利用二分查找法来找出第 K 个值。

参考第 104 道题“合并多个有序链表”：假设矩阵是  $M \times N$  大小，我们可以看做是 M 个或者 N 个有序链表，把题目转化为求这个 M 个或 N 个有序链表合并后的第 K 个值。假设 M 较 N 小，如果我们能从 M 个链表头选出最小值，那么移动最小值的表头，就可以按升序获取节点，并形成一个新链表。使用一个大小为 M 的最小堆求第 K 个值，因此，算法复杂度为  $O(K \log(\min(M, N)))$ 。

通过合并 M 个有序链表求第 K 个值的方法并没有有效利用行列均为有序的情况。如果 K 远小于 M 和 N，我们可以通过利用行列均有属性的属性，把算法复杂度降至  $O(K \log K)$ 。



假设当前最小堆的堆顶是最小值，其为位置  $(i, j)$ ，出堆之后，入堆的元素是其右边的元素，以及其下面的元素，因为从已知行列有序的顺序只有这两个元素在出堆元素之后。

首先，设计排序的单元，即矩阵里的元素。

---

```

class Cell implements Comparable<Cell>{
    int val, x, y;
    Cell(int val, int x, int y){
        //初始化坐标和值
        this.val = val;
        this.x = x;
        this.y = y;
    }
    public int compareTo(Cell c){
        //用来排序, 构建最小堆
        if(this.val > c.val) return 1;
        else if(this.val < c.val) return -1;
        else return 0;
    }
    //重写 equals 和 hashCode 函数, 用来判断是否为同一个元素
    public boolean equals(Object o){
        Cell c = (Cell)o;
        if(this.val == c.val && this.x == c.x && this.y == c.y){
            return true;
        }
        return false;
    }
    public int hashCode(){
        return this.val;
    }
}
    
```

---

其次, 建立一个最小堆。从左上角元素开始入堆。当堆顶出堆之后, 入堆的元素是其右边的元素以及其下面的元素。第 K 个出堆的元素, 即为第 K 个小的元素。

---

```

int getKthInSortedMatrix(int[][] matrix, int k) {
    int m=matrix.length, n=matrix[0].length;
    //建立一个堆, 升序排列
    PriorityQueue<Cell> heap = new PriorityQueue<Cell>();
    Cell cell = new Cell(matrix[0][0], 0, 0);
    heap.add(cell);
    int result = 0, count=0;
    
```

---

---

```
while(!heap.isEmpty()){
    Cell smallest = heap.poll();
    count++;
    if(count== k) return smallest.val;
    int x = smallest.x, y=smallest.y;
    if(x < m-1){
        Cell down = new Cell(matrix[x+1][ y], x+1, y);
        if(!heap.contains(down)) heap.add(down);
    }
    if(y < n-1) {
        Cell right = new Cell(matrix[x][y+1], x, y+1);
        if(!heap.contains(right)) heap.add(right);
    }
}
return result;
}
```

---

# 第 15 章

## 排列组合

排列组合通常用在字符串或序列的排列和组合当中，其特点是固定的解法和统一的代码风格。通常有两种方法：第一种是类似动态规划的分期摊还(amortized)的方式，即保存中间结果，依次附上新元素，产生新的中间结果；第二种是递归法，通常是在递归函数里，使用 for 语句，遍历所有排列或组合的可能，然后在 for 语句内调用递归函数。

### 面试题 115：翻译手机号码 ☆☆☆

输入由数字组成的字符串，输出数字代表字母的所有组合。数字代表的字母参见手机 9 键英文输入键盘。

#### 举例

输入 “23”，输出：“ad”，“ae”，“af”，“bd”，“be”，“bf”，“cd”，“ce”，“cf”。



手机 9 键英文输入键盘如下图所示。通常面试官会给出数字和字符的映射关系，或者直接掏出手机展示 9 键英文输入键盘给应聘者看，所以不需要特意记忆这些数字对应的字母。一般来说，数字 0 和 1 所代表的字母为空串，而从 2 到 9 的数字，分别代表字母“abc”，“def”，“ghi”，“jkl”，“mno”，“pqrs”，“tuv”，“wxyz”。



在这里，我们使用排列组合的第一种方法：分期摊还(amortized)的方法。从数字字符串第一个数字开始扫描，记录之前组合结果集，求出当前数字对应的多个字母，将每个映射的字母依附至结果集中的每个组合中，产生新的结果集。

扫描至数字结尾时，结果集即为所求的所有组合。

---

```
public ArrayList<String> letterCombinations(String digits) {
    ArrayList<String> res = new ArrayList<String>();
    res.add("");
    if(digits==null || digits.length()==0) return res;
    //数字对应的字母
    String[] maps = {"", "", "abc", "def", "ghi", "jkl", "mno",
                    "pqrs", "tuv", "wxyz"};
    for(int i=0; i<digits.length(); i++){
        String s = maps[digits.charAt(i)-'0']; //获取映射字符串
        ArrayList<String> temp = new ArrayList<String>();
        for(int j=0; j<s.length(); j++){
            //把每个对应的字母加入到每个结果集里
```

---

---

```
        for (int k=0; k<res.size(); k++){
            tmp.add(res.get(k)+s.charAt(j));
        }
        //保存最新结果集
        res=tmp;
    }
    return res;
}
```

---

## 面试题 116：数组签名 ☆☆☆☆

给出一个含有 1 到 N 的整型数组的签名，求出其所有可能的字典最小的排列 (lexicographically smallest permutation)，其中  $N \leq 9$ 。签名是这样计算的：比较相邻元素的大小，如果后者比前者大，输出 I，反之，输出 D。

### 举例

输入数组的签名“DDIIDI”，排列{3, 2, 1, 6, 7, 4, 5}就是其中一个输出。

含有 1 到 N 的数组长度为 N，其签名长度一定为 N-1。由于数字仅用一次，我们需要记录每个数字的使用情况。输出签名里的字符要么是 D，要么是 I。D 代表下降趋向，遇到 D 时，如果前一个数字为 i，那么只能从 1 到 i-1 里挑一个数字；I 代表上升趋向，遇到 I 时，当前数字只能从 i+1 到 N 里挑出。

使用递归的方法，程序的结构是典型的 for 语句内调用递归函数。

---

```
boolean computePerm(String signature, int n, int level,
    StringBuffer sb, boolean[] used){
    if (level==n-1){
        //遍历完了 signature
        System.out.println(sb.toString());
    }
```

---

---

```
        return true;
    }
    //获取前一个数字
    int i = sb.charAt(sb.length()-1)-'0';
    if(signature.charAt(level)=='I'){
        //找个最接近前一个数字，但又比它大的数字
        for(int j=i+1; j<=n; j++){
            if(used[j]) continue; //不能重复使用数字
            sb.append(j);
            used[j]=true;
            perm(signature, level+1, sb, used);
            sb.setLength(sb.length()-1); //恢复使用前的状态
            used[j]=false;
        }
    }else{
        //找个尽可能比前一个数字小的数字
        for(int j=1; j<i; j++){
            if(used[j]) continue;
            sb.append(j);
            used[j]=true;
            perm(signature, level+1, sb, used);
            sb.setLength(sb.length()-1); //恢复使用前的状态
            used[j]=false;
        }
    }
    return false;
}
```

---

## 面试题 117：组合和 ☆☆☆

给出一组正整数序列和一个目标值，求出所有可能的组合，使得组合里所有元素和为目标值，其中组合里的整数从输入序列里获取。要求：1) 每

个组合里的元素按照升序排列；2) 输出组合里不含有重复的组合；3) 从输入序列挑选出的整数，可以多次在组合里出现。

### 举例

输入整数序列{2, 3, 4, 7}，目标值为 7，那么输出的组合有：{7}, {2, 2, 3}, {3, 4}。

为了让输出元素按照升序排列，我们先对输入的数列进行排序。同样，我们使用递归的方法，程序的结构也是典型的 for 语句内调用递归函数。特别地，我们需要注意以下几点：

- 1) 记录输出组合的中间结果，每加入或移除一个数，则改变中间结果。
- 2) 记录剩余数之和，只有该值为 0 时，组合才是有效的。
- 3) 记录当前位置。因为序列里的数可以重复使用，所以下一次递归时，还可以从当前位置开始，这将体现在递归函数的参数里。

---

```
public ArrayList<ArrayList<Integer>> combinationSum(int[] candidates,
    int target) {
    if (candidates.length==0 || target <=0) return null;
    ArrayList<ArrayList<Integer>> resSet =
        new ArrayList <ArrayList <Integer>>();
    Arrays.sort(candidates); //排序保证结果是有序的
    helper(candidates, 0, target, new ArrayList<Integer>(), resSet);
    return resSet;
}

private void helper(int[] candidates, int start, int target,
    ArrayList<Integer> path,
    ArrayList<ArrayList<Integer>> resSet) {
    if(target< 0) return;
    if (target == 0) {
        //找到一个结果
        ArrayList<Integer>result = new ArrayList<Integer>(path);
        resSet.add(result);
    }
}
```

---

---

```
    } else {
        for(int i= start; i<candidates.length; ++i) {
            if (candidates[i]>target) break; //后面更大的数不可能满足条件
            path.add(candidates[i]);
            helper(candidates, i, target-candidates[i], path, resSet);
            path.remove(path.size()-1); //重置
        }
    }
}
```

---

### 扩展问题

如果序列可能有相同的元素，并且每个元素最多只能使用一次，如何处理？

比如，输入整数序列{2, 3, 4, 7}，目标值为 7，那么输出的组合没有了{2,2,3}，只有{7}和{3, 4}。这里有两个变化：1) 每个元素最多只能使用一次，下次递归是不能从当前位置开始的，而是从当前位置的下一个开始；2) 由于序列含有相等的元素，哪怕每个元素最多使用一次，也可能出现重复的组合，所以，为了避免输出重复组合，只取第一个相同元素加入组合里。这两个变化在程序里使用黑体标明。

---

```
private void helper(int[] candidates, int start, int target,
    ArrayList<Integer> path,
    ArrayList<ArrayList<Integer>> resSet) {
    if(target< 0) return;
    if(target == 0) {
        //找到一个结果
        ArrayList<Integer> result = new ArrayList<Integer>(path);
        resSet.add(result);
    } else {
        for(int i= start; i<candidates.length; ++i) {
            if (candidates[i]>target) break; //后面更大的数不可能满足条件
            //避免结果集重复，只取第一个相同值加入结果中
            if(i != start && candidates[i] == candidates[i-1])
                continue;
            path.add(candidates[i]);
        }
    }
}
```

---

---

```

        helper(candidates, i+1,
               target-candidates[i], path, resSet);
        path.remove(path.size()-1); //重置
    }
}

```

---

## 面试题 118：子集合 ☆☆☆

输入一个含有不同数字的序列，输出所有可能的集合（含空集）。要求：  
1) 集合里元素有序排列；2) 输出结果不含有重复集合。

### 举例

输入序列 {3, 1, 2}，输出集合有：{}，{1}，{2}，{3}，{1,2}，{1,3}，{2,3}，{1,2,3}。

为了让输出集合有序，先对输入序列进行排序。我们先考虑排列组合的第一种方法：分期摊还(amortized)的方法。初始化时，结果集合含有一个空集。当扫描数列时，保留原有集合，同时将当前扫描的元素插入现有的所有集合中，从而形成新的集合。

---

```

ArrayList<ArrayList<Integer>> subsets(int[] S){
    Arrays.sort(S); //对数列排序
    ArrayList<ArrayList<Integer>> resSet =
        new ArrayList<ArrayList<Integer>>();
    resSet.add(new ArrayList<Integer>()); //初始化加入空集
    for(int i=0; i< S.length; i++){
        ArrayList<ArrayList<Integer>> tmp =
            new ArrayList<ArrayList<Integer>>();
        for(ArrayList<Integer> list: resSet){
            tmp.add(list); //保留原有结果
            ArrayList<Integer> clone = new ArrayList<Integer>(list);
            clone.add(S[i]); //在原有结果里加入新元素
        }
        resSet.addAll(tmp);
    }
    return resSet;
}

```

---

---

```
        tmp.add(clone);
    }
    resSet tmp;
}
return resSet;
}
```

---

第二种方法是使用递归的方法，程序的结构是典型的 for 语句内调用递归函数。需要特别注意以下几点：

- 1) 保留中间任意结果，而不是遍历完成数列才保存结果。
- 2) 利用完了数列的数之和，需要回退，恢复至使用前的状态。

---

```
ArrayList<ArrayList<Integer>> subsets(int[] S){
    Arrays.sort(S);
    ArrayList<ArrayList<Integer>> resSet =
        new ArrayList<ArrayList<Integer>>();
    helper(S,0, new ArrayList<Integer>(), resSet);
    return resSet;
}

void helper(int[] S, int start, ArrayList<Integer> subset,
    ArrayList<ArrayList<Integer>> resSet){
    if(start> S.length) return;
    //保留任意中间结果
    ArrayList<Integer> clone = new ArrayList<Integer>(subset);
    resSet.add(clone);
    for(int i=start; i< S.length; i++){
        subset.add(S[i])//加入新元素，并递归调用下一个元素
        helper(S,i+1, subset, resSet);
        subset.remove(subset.size()-1)//回退
    }
}
```

---

### 扩展问题

如果序列含有重复元素，又要保证输出结果不含重复集合，如何处理？

类似的方法，为了避免输出重复组合，只取第一个相同元素加入组合里（在程序里使用黑体标明）。

---

```
void helper(int[] S, int start, ArrayList<Integer> subset,
            ArrayList<ArrayList<Integer>> resSet) {
    if(start > S.length) return;
    //保留任意中间结果
    ArrayList<Integer> clone = new ArrayList<Integer>(subset);
    resSet.add(clone);
    for(int i = start; i < S.length; i++) {
        //避免结果集重复，只取第一个相同值加入结果中
        if(i != start && S[i] == S[i-1]) continue;
        subset.add(S[i]); //加入新元素，并递归调用下一个元素
        helper(S, i+1, subset, resSet);
        subset.remove(subset.size()-1);
    }
}
```

---

## 面试题 119：全排列 ☆☆☆

输入一个不含相同数字的序列，输出所有可能的排列。

### 举例

输入序列 {1, 2, 3}，返回：{1, 2, 3}，{1, 3, 2}，{2, 1, 3}，{2, 3, 1}，{3, 1, 2}，{3, 2, 1}。

与上一道题“子集合”类似，使用递归方法，程序的结构是典型的 for 语句内调用



递归函数。不同的是，必须等到所有数字均在集合里才能输出。为了记录每个数字的使用情况，我们还需要辅助数组记录它们的使用情况。

---

```
ArrayList<ArrayList<Integer>> permute(int[] S){
    Arrays.sort(S);
    ArrayList<ArrayList<Integer>> resSet =
        new ArrayList<ArrayList<Integer>>();
    boolean used[] = new boolean[S.length]; //记录元素的使用情况
    helper(S, new ArrayList<Integer>(), resSet, used);
    return resSet;
}

void helper(int[] S, ArrayList<Integer> subset,
    ArrayList<ArrayList<Integer>> resSet, boolean[] used){
    if(subset.size() == S.length){
        //输出结果集
        ArrayList<Integer> clone = new ArrayList<Integer>(subset);
        resSet.add(clone);
        return;
    }
    for(int i=0; i< S.length; i++){
        if(used[i]) continue; //不能重复使用
        subset.add(S[i]); //加入新元素，并递归调用下一个元素
        used[i] = true;
        helper(S, subset, resSet, used);
        subset.remove(subset.size()-1); //还原
        used[i] = false;
    }
}
```

---

### 扩展问题

如果数列里含有重复的数，上述方法能正确运行吗？

比如，输入序列{1, 1, 2}，输出{1, 1, 2}，{1, 2, 1}和{2, 1, 1}。我们在挑选数字的时

候，除了考虑当前使用情况外，还需判断前一个相同元素的使用情况。如果前一个相同元素还未被使用，那么当前元素也不应该被使用，确保输出唯一。

---

```
void helper(int[] S, ArrayList<Integer> subset,
            ArrayList<ArrayList<Integer>> resSet, boolean[] used){
    if(subset.size() == S.length){
        ArrayList<Integer> clone = new ArrayList<Integer>(subset);
        resSet.add(clone);
        return;
    }
    for(int i=0; i< S.length; i++){
        //除了考虑当前使用情况外，还需判断是否为重复元素的第一个
        if(used[i] || (i>0 && !used[i-1] && S[i] == S[i-1])) continue;
        subset.add(S[i]) //加入新元素，并递归调用下一个元素
        used[i] = true;
        helper(S, subset, resSet, used);
        subset.remove(subset.size()-1) //还原
        used[i] = false;
    }
}
```

---

## 面试题 120：下一个排列 ☆☆☆☆☆

给出一个数字序列，求出它的下一个排列，即其在字典顺序的下一个变换 (lexicographically next greater permutation)。如果找不到下一个排列，则返回字典顺序里最小的变换。

### 举例

{1, 2, 3} → {1, 3, 2}

{3, 2, 1} → {1, 2, 3}

$\{3, 1, 2\} \rightarrow \{3, 2, 1\}$

先考虑一个特例，即找不到下一个排序，比如序列 $\{3, 2, 1\}$ ，这种数列的特点是倒序排列，返回的结果必须是正序顺序。通过观察例子，我们知道先找出最后一个峰值（即先上升后下降的转折点），找出从该峰值到结尾的数中比峰值前一个元素大而且最接近该元素的数，该数与峰值前一个元素交换，然后按从小到大的顺序对峰值后的部分（含峰值所在位置）排序，比如 $\{1, 3, 5, 4, 2\}$ ，最后一个峰值是 5，将 4 和 3 交换，然后对 5、3 和 2 排序，最后结果为 $\{1, 4, 2, 3, 5\}$ 。还有一种特例，找不到峰值，比如输入序列是正序排列，比如序列 $\{1, 2, 3\}$ ，那么交换最后两个元素即可，即得到 $\{1, 3, 2\}$ 。

---

```
public void nextPermutation(int[] num) {
    int n = num.length;
    int pivot = -1; //记录最后一个从下降序列到上升序列的转折点
    for (int i = n - 1; i > 0; i--) {
        if (num[i] > num[i - 1]) {
            pivot = i - 1;
            break;
        }
    }
    if (pivot >= 0) {
        for (int i = n - 1; i > pivot; i--) {
            if (num[i] > num[pivot]) {
                //交换和 pivot 最接近的又大于它的数
                int tmp = num[pivot];
                num[pivot] = num[i];
                num[i] = tmp;
                //交换之后，升序排列，尽可能小
                Arrays.sort(num, pivot + 1, n);
                return;
            }
        }
    }
    //如果是倒序，那么其下一个排序是递增排序
    Arrays.sort(num);
}
```

---

---

```
    }  
}
```

---

## 面试题 121: N 皇后 ☆☆☆☆

有  $N \times N$  大小的棋盘，放置  $N$  个皇后，使得任何一个皇后都无法直接吃掉其他的皇后，即任意两个皇后都不能处于同一条横行、纵行或斜线上。

这是一道经典面试题，有很多种解法。最直观的莫过于排列组合的方法。 $N$  个皇后放置在  $N \times N$  棋盘上，必定每行一个皇后，我们将每行中皇后的位置用 1 到  $N$  的数字来表示，总共有  $N$  行，这样就是 1 到  $N$  的数列排列。这种排列只满足了任意两个皇后不能同处在一行或一列上，并不能保证它们不会在一条斜线上。

在加入数字至排列前，判断该数字是否和排列里的所有数字在“斜线”上：如果两个数字在斜线上，那么两数之差的绝对值等于其下标差的绝对值。

---

```
public void NQueen(int n){  
    ArrayList<Integer> sol = new ArrayList<Integer>();  
    boolean used[] = new boolean[n+1]; //下标从 1 开始算  
    //1 到 n 的全排列  
    NQueen(n, 0, sol, used);  
}  
  
public void NQueen(int n, int start, ArrayList<Integer> sol,  
    boolean used[]){  
    if(start == n){  
        //找到一个方案  
        System.out.println(sol.toString());  
        return;  
    }  
  
    for(int i=1; i<=n; i++){  
        if(used[i]) continue; //不能重复使用
```

---

---

```
        if(!checkBoard(sol,i)) continue; //满足对角线限制
        sol.add(i);
        used[i]=true;
        NQueen(sol,start+1, sol, used); //找一下个皇后
        sol.remove(sol.size()-1);
        used[i]=false;
    }
}

public boolean checkBoard(ArrayList<Integer> sol, int q){
    for(int i=0; i< sol.size(); i++){
        //判断当前的 q 是否和之前所有皇后在对角线上
        if(Math.abs(i-sol.size())== Math.abs(sol.get(i)-q)){
            return false;
        }
    }
    return true;
}
```

---



---

## 第四部分 综合面试题

---

最后一部分面试题覆盖了数学运算、位操作、面向对象概念、系统设计以及海量数据的处理。





# 第 16 章

## 数学

与数学相关的面试题，我们经常遇到的是开方运算、幂运算、随机数产生器和随机抽样等。

### 面试题 122：Fibonacci 数 ☆

用非递归方法实现输出第  $n$  个 Fibonacci 数。

根据 Fibonacci 数的定义： $f(n) = f(n-1) + f(n-2)$ ，我们需要两个变量记录前两个数。初始化时，这两个数均为 1，从第三个 Fibonacci 数开始循环计算。

---

```
int fib(int n){
    if(n<=2) return 1;
    int a=1, b=1, i=3, res=0;
    while(i<=n){
        res = a+b; //第 i 个 Fibonacci 数
```

---

---

```
    a=b;
    b=res;    //相应更新前两个数
    i++;
}
return res;
}
```

---

### 扩展问题

递归实现输出第  $n$  个 Fabonacci 数。请说明如何进一步做优化。

通过观察我们知道递归求出前两个数时,存在多次的重复计算。为了避免重复计算,我们使用空间换时间的概念,使用一个哈希表记录前  $n-1$  个 Fabonacci 数的值。例如,在求  $i$  个 Fabonacci 数时,递归调用函数计算第  $i-1$  个 Fabonacci 值,其实在之前我们已经算出了该值,只要把该值存入哈希表里即可。

---

```
int fibII(int n, HashMap<Integer, Integer> hm){
    if(n<=2) return 1;
    if(hm.containsKey(n)){//避免重复计算
        return hm.get(n);
    }
    int res=fibII(n-1, hm)+fibII(n-2,hm); //递归调用
    hm.put(n, res);
    return res;
}
```

---

## 面试题 123: 求幂 ☆☆☆

实现  $\text{pow}(x,n)$ , 并分析时间复杂度。

直观的办法是连续  $n$  个  $x$  相乘, 求出最后的乘积。这种方法的时间复杂度为  $O(n)$ 。

还有更快的算法吗？我们可以把  $n$  个  $x$ ，分为相同个数的两部分，即  $n/2$  个  $x$ ，再把每个部分进一步拆分成两个相同的部分，以此类推。以  $n$  个  $x$  相乘的积为树根，建立一个平衡二叉树，同一层节点的值相同，也就是说整棵树不同值的个数就是树高，因此，时间复杂度为  $O(\log n)$ 。

由于  $n$  不可能总是偶数，如果  $n$  是正奇数，那么  $n$  个  $x$  的乘积等于两部分的乘积再乘以  $x$ 。如果  $n$  是负的奇数，那么  $n$  个  $x$  的乘积等于两部分的乘积再乘以  $1/x$ 。

---

```
double pow(double x, int n) {
    if (n == 0) return 1.0;
    //计算一半的值，可以重复利用
    double half = pow(x, n / 2);
    if (n % 2 == 0)
        return half * half;
    elseif (n > 0)
        return half * half * x;
    else
        return half * half / x; //负数情况
}
```

---

## 面试题 124：求开方 ☆☆☆☆

实现 sqrt 函数。

使用二分查找法比顺序查找要快很多。为了避免死循环，设置一个精度值。

---

```
double sqrt(double a){
    if(a==0 || a==1) return 0;
    double precision = 1.0e-7, start=0, end=0;
    if(a<1) end = 1;
    while(end-start > precision){ //二分查找
        double mid= (start+end)/2;
```

---

---

```
        if(mid <= a / mid) return mid; //找到当前值
        elseif(mid > a / mid) end = mid; //抛弃大的部分
        else start = mid; //抛弃小的部分
    }
    return (start+end)/2;
}
```

---

此外，还可以利用数学公式，具体参考 wikipedia 的求平方根词条：[http://en.wikipedia.org/wiki/Methods\\_of\\_computing\\_square\\_roots](http://en.wikipedia.org/wiki/Methods_of_computing_square_roots)。下面这段程序实现了 Babylonian Method。如果你不了解这些公式，背下其中一个。

---

```
public double sqrt(double d){
    if(d==0.0) return 0;
    double root = d/2;
    double tolerance = 1.0e-7; //精度
    do{
        root = (root+d/root)/2;
    }while(Math.abs(root*root - d) > tolerance);
    return root;
}
```

---

## 面试题 125：随机数产生器 ☆☆☆☆☆

给出一个随机数产生器 Rand7()，随机产生 1 到 7 的整数，根据该随机数产生器产生 1 到 10 的随机数。

调用一次 Rand7()还不足以产生 1 到 10 的随机数，我们可以考虑调用两次，将第一次和第二次可能的结果进行组合，如下表所示：

	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7
2	8		9		10	1	12
3	15	16	17	18	19	20	21
4	22	23	24	25	26	27	28
5	29	30	31	32	33	34	35
6	36	37	38	39	40	41	
7	42						

两次调用产生的组合有 49 种情况，如果我们抛弃 41 到 49 的组合，那么剩下有 40 种组合。这 40 种组合，我们可以理解为 1 到 10 的 10 个随机数中每个随机数出现了 4 次，即采用除以 10 取余数的做法，以至于能在同等概率下产生随机数。根据以上二维坐标定位取余数的做法，我们可以扩展至产生任意的随机数上。

---

```

int rand10() {
    int row, col, idx;
    do {
        row= rand7();
        col= rand7(); //调用两次
        idx= col + (row-1)*7; //计算组合
    } while (idx > 40); //只取能在同等概率下产生随机数的组合
    return 1 + (idx-1)%10;
}

```

---

## 面试题 126：找出明星 ☆☆☆

给出一个函数 knows(A,B)，如果 A 认识 B，则返回 true，否则返回 false。明星定义为他不认识所有的人，但所有的人认识他。现在给出 N 个人，在线性时间内找出所有明星。

这  $N$  个人里面最多只有一个明星。为什么呢？如果有两个明星，那么这两个人应该彼此认识，这和明星的定义冲突。借用俩指针的做法，有一头一尾的两个指针指向第 1 到  $N$  个人。假设这两个指针为  $A$  和  $B$ ，调用 `knows` 函数，如果  $A$  认识  $B$ ，返回 `true`，这说明  $A$  不可能是明星，把  $A$  向后移动一位。如果函数返回 `false`，那么  $B$  也不可能是明星， $B$  往前移动一位。对于最后剩下的人，我们并不确定其是否为明星，那么我们从头到尾再调用 `knows` 函数，如果所有人都认识他，那么他就是场上的明星。如果你是球迷，而且了解淘汰赛机制，那么你可以轻而易举地解决这道题。

## 面试题 127：聚合数 ☆☆☆

给出一个正整数，判断其是否为聚合数。如果一个数是聚合数，那么这个数满足：把这个数的数位拆成若干部分，后面的部分是其前面两部分的和。

### 举例

112358 是聚合数，因为我们把它拆成 1, 1, 2, 3, 5, 8，其中  $1+1=2$ ， $1+2=3$ ， $2+3=5$ ， $3+5=8$ 。122436 也是聚合数，拆成三部分 12, 24, 36，其中  $12+24=36$ 。

当我们没什么头绪的时候，往往暴力破解是最简单有效的办法。先将输入数进行数位分割，提取出两个数，算出两者之和，即第三个数，然后累加第二个数和第三个数，得到第四个数，以此类推，直到所有数组成字符串长度相等或大于输入数的原串，然后比较拼接后的结果和原串。如果不相等，重新分割输入数的数位。

---

```
boolean isAggregatedNumber(String num) {
    for (int i = 1; i <= num.length() / 2; i++) {
        for (int j = i+1; j <= num.length() / 2 + 1; j++) {
            //提取 num[0...i-1] 和 num[i...j-1] 的两个数
            (ifMatch(i, j, num)) return true;
        }
    }
}
```

---

---

```
    }
    return false;
}

boolean isMatch(int i, int j, String num) {
    String first = num.substring(0, i);
    String second = num.substring(i, j);
    StringBuffer sb = new StringBuffer();
    sb.append(first);
    sb.append(second);
    while (sb.length() < num.length()) {
        // 根据前两个数算出第三个数，然后更新前两个数
        // 直至累积长度大于或等于 num 长度为止
        String third = "";
        try {
            third = Integer.toString(Integer.parseInt(first) +
                                     Integer.parseInt(second));
        } catch (Exception e) {
            return false;
        }
        sb.append(third);
        first = second;
        second = third;
    }
    return (num.equals(sb.toString()));
}
```

---

## 面试题 128：根据概率分布产生随机数 ☆☆☆☆

根据输入概率密度（即一维整型数组），产生随机数。例如，输入一维整型数组  $w$ ，长度为  $L$ ，则返回一个随机数，假设为  $i$ ，那么  $i$  在  $0$  与  $L-1$  之间，而且出现  $i$  的概率是  $w[i]/\text{sum}(w)$ 。

首先, 根据概率密度, 需要计算每个输出随机数的概率分布。假设输入  $w=\{1, 2, 3, 2, 4\}$ , 累加  $w$  元素得到,  $P=\{1, 3, 6, 8, 12\}$ 。然后, 调用随机函数产生 0 到 12-1 的随机数, 看看该随机数落在哪个区间。假设产生的随机数为 10, 那么它落在  $[8, 12)$  区间, 则返回下标 4。如果是 6, 它落在  $[6, 8)$  区间, 则返回 3。

---

```
int randWithWeights(int[] w){
    int n = w.length;
    int dist[] = new int[n];
    dist[0] = w[0];
    for(int i=1; i< n; i++){
        //计算概率分布
        dist[i] = w[i] + dist[i-1];
    }
    Random random = new Random();
    //随机产生 0 到 dist[n-1]-1 的数字
    int rand = random.nextInt(dist[n-1]);
    for(int i=0; i< n; i++){
        //看看落在哪个区间
        if(rand < dist[i]) return i;
    }
    return 0;
}
```

---

## 面试题 129: 随机采样 ☆☆☆

随机从  $N$  个数中取出  $K$  个数。假设  $N$  非常大。

其实这道题就是在一个数据流里随机采样, 取出  $K$  个数。首先, 把  $K$  个数填满, 填满后, 我们再考虑替换结果中的某个数。何时替换? 如果需要替换, 替换谁?

其次, 根据数据流当前的个数, 假设为  $i$ , 那么在  $[0, i)$  区间产生一个随机数  $index$ 。



如果 index 比 K 小，则把结果的第 index+1 个数替换为数据流的当前元素。

---

```
int[] pickK(int[] A, int k){
    int buf= new int[k];
    for(int i=0; i< A.length; i++){
        if(i<k) buf[i]=A[i];
        else{
            Random rand= new Random(i);
            //随机产生 0 到 i-1 数字
            int index = rand.nextInt();
            //如果在 k 以内，替换原有的值
            if(index<k) buf[index]=A[i];
        }
    }
    return buf;
}
```

---

## 面试题 130：数组元素乘积 ☆☆☆

输入一个数组 A，输出一个数组 B，使得 B[k]等于除了 A[k]以外，A 的其他所有元素的乘积。假设不能使用除法。

假设能使用除法，先求出 A 的所有元素之积，为了获取 B[k]，把乘积除以对应的 A[k]即可。我们可以把乘积的元素分为两部分，一部分在 A[k]的左边，另一部分在 A[k]的右边。对于每个 B[k]，先从头开始扫描，求出第一部分的乘积；然后，从后端倒着扫描，求出后半部分的乘积，两部分相乘即为所求的结果。注意每一部分的乘积不应包含元素本身。

---

```
int[] getProduct(int[] A){
    int[] B = new int[A.length];
    for(int i=0; i<A.length; i++){
```

---

---

```
        //计算 i 之前所有元素之积
        if (i==0) B[i]=1;
        else B[i]=B[i-1]*A[i-1];
    }
    int right=A[A.length-1];
    for(int i=A.length-2; i>=0; i--){
        //乘上右边的元素
        B[i]= B[i]*right;
        right=A[i]*right;
    }
    return B;
}
```

---

## 面试题 131：访问计数 ☆☆☆

设计一个网页访问计数器，可以方便统计每秒、每分、每小时的访问量。

以秒为单位，通过数组 `clicksBySec` 记录从凌晨开始到现在的访问总数。数组的长度为一天内的秒数，即  $24*60*60 = 86400$ 。比如，我们求出第 10000 秒的访问量，即 `clicksPerSec[10000] - clicksPerSec[9999]`，在常数时间内可以获取。如果想获取第  $i$  分钟的访问量，即为 `clicksPerSec[i*60] - clicksPerSec[(i-1)*60]`；如果想获取第 5 个小时的访问量，即为 `clicksPerSec[5*3600] - clicksPerSec[4*3600]`。

# 第 17 章

## 位操作

与位操作相关的题目有去重、求异、不得使用加、减、乘、除做运算等。

### 面试题 132: isPowerOf2() ☆☆

输入一个整数  $n$ ，判断  $n$  是否为 2 的幂。

我们知道满足  $n \& (n-1)$  等于 0 的条件， $n$  或者是 2 的幂，或者是 0。因此，我们只需要把 0 排除。

---

```
boolean isPowerOf2(int n){  
    return ((n!=0) && (n&(n-1) ==0));  
}
```

---

## 面试题 133: isPowerOf4() ☆☆☆☆

输入一个 32 位整数，判断是否为 4 的幂。

有了判断是否为 2 的幂的条件，我们还需进一步过滤，使得它也是 4 的幂。通过观察，我们知道在所有 2 的幂中，奇数位含有 1 的均不是 4 的幂。

---

```
boolean isPowerOf4(int n) {  
    return n!=0 && (n & 0xAAAAAAAA)==0 && (n & (n-1)) ==0;  
}
```

---

## 面试题 134: 两数相除 ☆☆☆☆

不用乘、除、求余操作，返回两整数相除结果，结果也是整数。

为了方便起见，暂不考虑两数的符号，假设都是正整数，最直观的方法是：不停用被除数减去除数，看看能减多少次，而被除数还大于等于 0。能否加快相减的步骤呢？假设除数是 2，相除的商就是被除数二进制表示向右移动一位。假设被除数是 a，除数是 b，因为不知道 a 除以 b 的商，所以只能从 b, 2b, 4b, 8b...这种序列一个个尝试，从 a 扣除那些尝试的值。如果 a 大于序列的数，那么从 a 扣除该值，并且最终结果是商加上对应的二进制位为 1 的数，然后尝试序列下一个数。如果 a 小于序列的数，那么从头再来，直至 a 小于 b。

---

```
int divide(int dividend, int divisor) {  
    long a = Math.abs((long)dividend); //保留符号避免溢出
```

---

---

```
longb = Math.abs((long)divisor);
longresult = 0;
while(a >= b) {
    long = b;
    int i=0;
    while(a>=c){
        a = a-c;
        //加上对应的二进制位为 1 的数
        result+= 1<<i; //1, 2, 4, 8, 16...
        i++;
        //序列的下一个数
        c = c << 1; //b, 2b, 4b, 8b...
    }
}
if((dividend>0 && divisor <0) || (dividend <0 && divisor >0)){
    //除数和被除数其中一个为负数
    result = -result;
}
return(int)result;
}
```

---

## 面试题 135：不用加减乘除做加法 ☆☆☆

求两个整数之和，要求在函数内不得使用加、减、乘、除四则运算。

异或保留相加之和的结果，而与操作可以记录进位。为了方便下一轮计算，我们把进位向左移动一位，把中间结果和进位当成新的两个数，循环操作直至进位为 0。

---

```
int add(int a, int b){
    int sum, carry;
    do{
        //使用异或和与保留进位
```

---

---

```
        sum= a^b;
        carry= (a&b) <<1; //向左移动一位
        a= sum;
        b= carry;
    }while(b!=0);
    return a;
}
```

---

## 面试题 136：实现 BitSet 类 ☆☆☆

实现 BitSet 类，至少包含 get(), set(), clear() 函数。

使用长整型数组保存位的数据，数组长度在 BitSet 类初始化时确定。假设数组长度为 len，一个长整型有 64 位，所以整个数组可以表示 64\*len 个位。实现 get、set、clear 函数时，先映射到数组的哪个元素（即先除以 64），然后通过余数判断在这个整数的哪个位上。

---

```
class BitSet{
    long[] bits; //使用长整型数组
    BitSet(int numBits) {
        //一个长整型是 64 位，可以表示 64 个数
        int numLongs = numBits >>> 6;
        if (numBits & 0x3F) != 0) {
            //如果还有余数
            numLongs++;
        }
        bits= new long[numLongs];
    }

    private BitSet(long[] bits) {
        this.bits= bits;
    }

    boolean get(int index) {
```

---

---

```
//先除以 64，找到是哪个整数里，通过余数判断是在哪个位
return (bits[index >>> 6] & 1L << (index & 0x3F)) != 0L;
}

voidset(int index) {
    //先找到哪个整数，然后设位
    bits[index>>> 6] |= 1L << (index & 0x3F);
}

voidclear(int index) {
    //清 0
    bits[index>>> 6] &= ~(1L << (index & 0x3F));
}
}
```

---

## 面试题 137：爬楼梯 II ☆☆☆

给出楼层数  $N$ ，求最快达到顶楼的方法。假设每个单步可以移动  $2^k$  个楼层， $k$  为任意整数。

借用贪心算法，每次跳跃的时候尽可能接近楼顶，这样我们可以把问题转化为  $n$  的二进制表示，每个设置位对应的是  $k$  的值，即每次单步移动  $2^k$  个楼层。

---

```
void climbStairs(int n){
    for(int i=0; i<32 && n>0; i++){
        //设置位对应的 k 值
        if((n1) > 0) System.out.print("k="+i+",");
        >>=1;
    }
}
```

---

## 面试题 138：只出现一次的数字 ☆☆

求出数组中只出现一次的一个数字，而其他数字出现了两次。

我们注意到两个相同的数异或之后结果为 0。如果我们异或数组所有的元素，那么所有出现两次的数字异或后为 0，最后的结果就是只出现一次的数字。

如果只出现一次的数字有两个呢？假设这两个数是 A 和 B，那么我们异或数组所有的元素之后，得到 C，C 为  $A \oplus B$ 。从 C 中找出一个设置位，在这个位上 A 和 B 是不同的，以这个位的值把数组一分为二，为 1 的分配到一组，为 0 的分配到另一个组，A 和 B 肯定不是在一个组。这样，在同一个组内，只有一个数字出现了一次，其他数字出现了两次。最后，我们对两个组分别进行异或操作，每组各得一个数，即找出了这两个只出现一次的数。



# 第 18 章

## 面向对象

### 面试题 139：实现迭代器 peek() ☆☆☆

实现含有 peek()函数的 Iterator 类。

初始化的时候，先把第一个元素提取出来保存在私有变量里，这样 peek()函数直接返回改私有变量的值即可。重写 hasNext()和 next()。next()函数返回当前保存在私有变量的第一个元素，同时求出下一个元素来替换私有变量的值。

---

```
public class Peekable<T> {
    public Peekable(Iterator <T> iterator) {
        _iterator = iterator;
        getTop(); // 提前当前元素
    }

    public boolean hasNext() {
```

---

---

```
        return _top != null;
    }

    public T next() {
        //空就抛出异常
        if(_top == null) throw new NoSuchElementException();
        _currentTop = _top;
        //找出下一个元素
        getTop();
        return _currentTop;
    }

    public T peek() {
        return _top;
    }

    private void getTop() {
        _top = null;
        //先取出第一个元素
        if(_iterator.hasNext()) {
            _top = _iterator.next();
        }
    }

    private Iterator<T> _iterator;
    private T _top;
}
```

---

## 面试题 140：实现复杂的迭代器 ☆☆☆☆

实现一种迭代器处理复合型的数据结构，至少包含 hasNext() 和 next() 两个函数。

复合型的数据结构定义如下：

---

```
public interface Data<T> {
    public boolean isCollection(); //是否为一个集合
    // 如果是集合返回集合，如果是单个元素，则返回 null
    public Collection<Data<T>> getCollection();
    // 如果是单个元素则返回该值，如果是集合，则返回 null
    public T getElement();
}
```

---

初始化时，将输入集合扁平展开之后放入一个列表里，即列表由单个元素组成。去嵌套化之后，next()和 hasNext()函数与普通 Iterator 类类似，需要记录当前位置，并与列表长度比较。

---

```
public class ComplexIterator<T> implements Iterator<T> {
    private int currIndex; //记录当前位置
    private ArrayList<T> flatColl; //扁平展开之后的集合
    public ComplexIterator(Collection<Data<T>> c) {
        currIndex=0;
        flatColl=new ArrayList<T>();
        flatElements(c);
    }
    //去除嵌套
    private void flatElements(Collection<Data<T>> c) {
        for(Data<T> item : c) {
            if(item.isCollection())
                //若是集合，递归调用
                flatElements(item.getCollection());
            else{
                flatColl.add(item.getElement());
            }
        }
    }
    public T next() {
        if(null==flatColl|| currIndex >= flatColl.size())
            throw NoSuchElementException();
    }
}
```

---

---

```

        T=flatColl.get(currIndex);
        currIndex++;
        return;
    }

    public boolean hasNext() {
        return null!=flatColl&&flatColl.size()> 0&&
            currIndex< flatColl.size();
    }
}

```

---

## 面试题 141：实现 BlockingQueue ☆☆☆

实现可支持多线程的 BlockingQueue 类。

对于存、取操作，均需加上锁。使用双端队列来存放元素。当存放元素时，如果队列满了，则需要等待直至队列有空余；取出元素时，如果队列是空的，那么等待直至队列含有新元素为止。

---

```

public class BlockingQueue<T> {
    private LinkedList<T> list; //双端队列
    private int limit; //队列长度的限制
    public BlockingQueue(int limit) {
        this.limit=limit;
        list=new LinkedList<T>();
    }

    public synchronized void put(T t) throws InterruptedException
    {
        while(list.size()==limit){
            try{
                list.wait(); 队列满了，等待其他线程唤醒
            } catch (InterruptedException e) {

```

---

---

```
        //
    }
}

list.add(t);
if(list.size()>0)    list.notifyAll(); //唤醒读取线程
}

public synchronized T get() throws InterruptedException
{
    while(list.size()==0){
        try{
            list.wait(); //队列为空，等待
        } catch (InterruptedException e) {
            //
        }
    }
    T=t=list.pop();
    list.notifyAll(); //通知由于队列满了而等待的线程
    return t;
}
}
```

---

## 面试题 142：Java 字节码编入 ☆☆

请解释 Java 字节码编入(Bytecode Instrumentation)。

Java 字节码植入编入，简称 BCI，是指在调用函数前后插入字节码的一种技术，以改变程序运行结果。这个过程通常发生在 Java 类加载的过程中。

## 面试题 143：依赖注入 ☆☆

你是如何理解注入(Dependency Injection)?

依赖注入是一种设计模式，也被称为控制反转模式(Inversion of Control，简称 IoC)，以消减程序的耦合问题。依赖注入是指一种对象配置方式，即对象的成员变量由外部实体来设置，以达到松耦合的目的。例如，一个类成员变量含有其他类的对象，即通过组合方式达到“has a”的关系，可以说这个类依赖于其他类。我们可以通过如下操作实现减少该类对其他类的依赖：把成员变量的初始化移到构造函数，甚至是构造函数的输入参数里。

# 第 19 章

## 杂项

### 面试题 144：垃圾回收机制 ☆☆☆

请解释你所了解的几种内存垃圾回收机制。

随着 Java 虚拟机的发展，垃圾回收机制也伴随着发生了变化。在这里，我们列举了几个有代表性的内存垃圾回收机制。

**1. 引用计数法 (Reference Counting)：**使用引用计数器来区分存活对象和不再使用的对象。每个对象对应一个引用计数器。当每一次创建一个对象并赋与一个变量时，引用计数器设置为 1。当对象被赋与任意变量时，引用计数器每次加 1，当对象出了作用域后，或该对象丢弃不再使用时，引用计数器减 1，一旦引用计数器为 0，对象就满足了垃圾收集的条件。基于引用计数器的垃圾收集器运行较快，不会长时间中断程序执行，适宜地实时运行的程序，但引用计数器增加了程序执行的开销。

**2. 标记和清除 (Mark and Sweep):** 为了解决引用计数法的问题而提出, 它使用了根集的概念。垃圾收集器从根集开始扫描, 识别出哪些对象可达, 哪些对象不可达, 并用某种方式标记可达对象, 定时回收不可达的对象。根集定义为正在执行的 Java 程序可以访问的引用变量的集合 (包括局部变量、参数、类变量), 程序可以使用引用变量访问对象的属性和调用对象的方法。

**3. 压缩算法 (Compacting):** 为了解决堆碎片问题, 对标记和清除做了修改。在清除的过程中, 算法将所有的对象移到堆的一端, 堆的另一端就变成了一个相邻的空闲内存区, 收集器会对它移动的所有对象的所有引用进行更新, 使得这些引用在新的位置能识别原来的对象。

**4. 复制算法 (Coping):** 开始时把堆分成一个对象面和多个空闲面, 程序从对象面为对象分配空间, 当对象满了, 垃圾收集器就从根集中扫描活动对象, 并将每个活动对象复制到空闲面, 这样空闲面变成了对象面, 原来的对象面变成了空闲面, 程序会在新的对象面中分配内存。这种算法解决了压缩算法移动对象的开销, 但空间利用率下降了。

## 面试题 145: 程序崩溃 ☆☆☆☆

什么原因导致测试用例会在同一代码上跑有时能通过, 有时却失败了?

硬件、软件、程序设计和编码都可能导致有时候测试用例通不过, 通常来说有以下原因:

- 多线程下的资源竞争(Race Conditions)。
- 使用共享内存空间。
- 使用外部变量, 而这个变量可能被别的进程修改。



- 依赖外部类库，在加载时失败。
- 依赖其他进程的输出。
- 网络连接错误。
- 内存泄露。
- 程序使用随机数，随机数导致测试用例失败。

## 面试题 146：实现任意读 ☆☆☆☆

给一个一次只能读 4KB 的 C++ 函数 `int read4096(byte *buff)`，要求实现能读任意字节数的 `int read(byte *buff, int n_bytes)`。

假设需要读的字节个数为 `n_bytes`，那么 `n_bytes` 除以 4096 的商，即为调用 `read4096()` 的次数。如果它们的余数大于 0，则还需要再次调用 `read4096()`，将剩下的字节内容拷贝到之前的结果中。

---

```
int read(char* buf, int n){
    int num = n / 4096; //读的次数
    int remain = n % 4096;
    int total = 0;
    while(num--){
        int num_read = read4096(buf);
        if(num_read == 0) break; //读取出错
        buf += num_read;
        total += num_read;
    }
    //读取出错的情况返回已读数量
    if(total != n - remain) return total;
    //最后一次读取剩下的字节
    char readbuf[4096];
    int num_read = read4096(readbuf);
```

---

---

```
int num_copy = min(num_read, remain);
memcpy(buf, readbuf, num_copy); //把剩下的字节内容拷贝到 buf
total += num_copy;
//返回最后读取的字节大小
return total;
}
```

---

## 面试题 147：实现读一行 ☆☆☆

根据给出的一个 C++ 函数 `char* read4096()`，写一个 `char* readLine()` 的函数，满足：

1. 当遇到换行和文件结尾时返回
2. `readLine()` 可能被多次调用，确保其正确性
3. 返回的字节长度可能大于 4096

循环调用 `read4096()`，对于读取的字符逐个扫描，如果遇到文件结尾符或者换行符时，则终止循环。最后，拼接每次读取的字符，并作为结果返回。

---

```
char* readLine() {
    char buffer[MAX_SIZE+1];
    char tmpbuf[4096];
    int size=4096;
    int total=0;
    while(true) {
        tmpbuf = read4096();
        if(tmpbuf=='\0') { // 到达文件尾部，返回
            buffer[total] = '\0';
            break;
        }
        for(int i=0; i<4096; i++) {
            //对于读取的字符逐个扫描，
            //遇到文件尾部或者换行符时返回
        }
    }
}
```

---

---

```
        if(tmpbuf[i]=='\0' || tmpbuf[i] =='\n'){
            size=i+1;
            break;
        }
    }
    //把临时空间内容导出来
    memcpy(buffer+total, tmpbuf, size);
    total    +=size;
    //    到文件尾部或者换行，退出循环读取 4096 个字符
    if(size<4096){
        buffer[total]='\0';
        break;
    }
}
return buffer;
}
```

---

## 面试题 148：统计电话号码个数 ☆☆☆

给出 10 亿个手机号码，统计不重复的号码的个数。

如果使用字符串存储号码，大概需要 110GB 内存，才能容纳这 10 亿个手机号码，所以使用计数数组不可行。我们可以考虑使用 BitMap 来标记相应的号码是否出现（不需要统计每个号码出现的次数）。即便如此，单单使用一个 BitMap，内存还是容纳不了上亿的号码。这时候，我们考虑使用二次寻址，即双层桶概念，分区统计，把统计的中间结果存入硬盘。例如，分成一万个分区，标记时，只要导入对应的分区，这样大概需要 60MB 内存。最后，遍历整个 bitmap，统计位为 1 的个数。

更多与海量数据处理相关的数据结构参加附件 B “海量数据结构”。

## 面试题 149：海量数据高频词 ☆☆☆

给出 1GB 大小的文件，每行是一个词，内存限制在 1MB 以内，要求返回频率最高的 100 个词。

先统计每个词出现的次数，然后使用一个堆，维护频率最高的 100 个词。

1) 统计每个词出现的次数。如果内存无法容纳这么多单词，利用分布式哈希表，进行分区统计。例如，分开放置到  $n$  个文件，使用映射  $\text{hash}(\text{word}) \% n$ ，使用哈希函数记录每个单词出现的次数，即 key 为单词，value 为次数。

2) 有了这些次数之后，使用一个堆，维护频率最高的 100 个词，把  $n$  个统计结果文件从头扫描一遍。最后，在堆里的单词就是高频词。

## 面试题 150：多台机器的中值 ☆☆☆☆

求存放在  $N$  台机器上海量数字的中值。

借用快速排序的 partition 思想，把问题转化为对这  $N$  台机器的数字进行分区，找出第  $k$  ( $k=\text{total}/2$ ) 个数（假设总的数字个数为 total 个）。

1) 挑选某个机器作为协调员，向其他  $N-1$  台机器发送统计数字个数的请求。加上自身的数字个数，计算出总个数，假设为 total。

2) 协调员从自身抽取某个数字作为标杆 (pivot)，或者从其他机器挑选（如果本机没有进入下一轮计算的数字）。把这个数字发送给所有机器，让所有机器返回比这个数

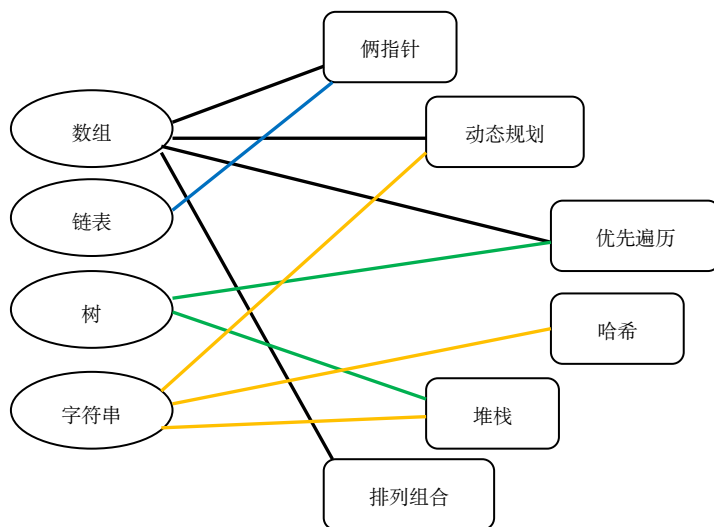
字小或相等的数字个数。累加各台机器返回的结果，假设为  $m$ 。如果  $m$  等于  $k$ ，则返回  $\text{pivot}$ ；如果  $m$  小于  $k$ ，让各台机器丢弃小于等于  $\text{pivot}$  的数（即这些数不进入下一轮计算），更新  $k$  的值， $k=k-m$ ，并重复第二步；如果  $m$  大于  $k$ ，让各台机器丢弃大于  $\text{pivot}$  的数（即不进入下一轮计算），重复以上过程。

3) 重复第二步，直至找出  $m$  等于  $k$  为止。

算法分析：假设这些海量数据较为均匀地分布在每台机器上，即每台有  $O(\text{total}/N)$  个数。平均而言，如果是第  $i$  轮计算，每台机器会进行  $O(\text{total}/(N \cdot 2^i))$  比较，共有  $O(\log(\text{total}/N))$  轮计算。此外，每轮通信和统计每台机器中间结果需要  $O(N)$  时间。总的时间复杂度为  $O(\text{total}/N + N \cdot \log(\text{total}/N))$ ，一般来说  $N$  远小于  $\log(\text{total})$ ，所以最终为  $O(\text{total}/N)$ ，即每台机器平均存储量的线性级别时间。

# 附录 A

## 数据结构与算法



附录 B

海量数据结构

数据结构	应用场景	示 例
哈希表	要求所有键值对放入内存；查找可在常数时间内完成。	<ul style="list-style-type: none"><li>• 提取某日访问百度次数最多的 IP</li><li>• 统计不同电话号码的个数</li></ul>
堆	插入和堆调整需要 $O(\log N)$ 时间， $N$ 为堆元素的个数，而获取堆顶元素只需要常数时间。	<ul style="list-style-type: none"><li>• 求出海量数据前 <math>K</math> 大的数</li><li>• 求出海量数据流的中位数</li></ul>
BitMap	通常记录整数出现情况，用来快速查找、数字判重、删除元素等。	<ul style="list-style-type: none"><li>• 统计不同电话号码的个数</li><li>• 2.5 亿个整数中查出不重复的整数的个数</li></ul>
双层桶	两次寻址方式以节省内存，通常用在求第 $K$ 大、中位数和数字判重。	<ul style="list-style-type: none"><li>• 5 亿个整数找出中位数</li><li>• 海量数据的第 <math>K</math> 大的值</li></ul>
反向索引 (Inverted Index)	通过单词—文档，属性—实体建索引，方便后续查找。	<ul style="list-style-type: none"><li>• 基于关键词的搜索</li><li>• 搜索框输入的自动补全</li></ul>

续表

数据结构	应用场景	示 例
外排	借用硬盘空间实现海量数据的排序。	<ul style="list-style-type: none"><li>• 1GB 大小的文件，每行是一个词，内存 1MB，返回频率最高的 100 个词</li></ul>
前缀树 (Trie)	为集合内所有单词建立前缀树。	<ul style="list-style-type: none"><li>• 求出热门的查询字符串</li><li>• 求重复率高的词</li></ul>
MapReduce	分布式处理，将数据交给不同机器去处理，划分数据，然后归约结果。	<ul style="list-style-type: none"><li>• 海量日志分析</li><li>• 数据挖掘</li><li>• 智能推荐系统</li></ul>