

需要整本电子书，联系我QQ：3411317522

Broadview
www.broadview.com.cn

程序员刷题宝典！编程能力提升秘笈！

程序员代码面试指南

IT名企算法与数据结构题目最优解

左程云 著

精选IT名企真实代码面试题，全面覆盖算法与数据结构题型



Coding Interview Guide

需要整本电子书，联系我QQ：3411317522

中国工业出版社
CHINA INDUSTRY PUBLISHING HOUSE
www.cip.com.cn

需要整本电子书，联系我QQ：3411317522

程序员代码面试指南

IT名企算法与数据结构题目最优解

左程云 著

电子工业出版社

Publishing House of Electronics Industry

需要整本电子书，联系我QQ：3411317522

需要整本电子书，联系我QQ：3411317522

内 容 简 介

这是一本程序员面试宝典！书中对IT名企代码面试各类题目的最优解进行了总结，并提供了相关代码实现。针对当前程序员面试缺乏权威题目汇总这一痛点，本书选取将近200道真实出现过的经典代码面试题，帮助广大程序员的面试准备做到万无一失。“刷”完本书后，你就是“题王”！

本书采用题目+解答的方式组织内容，并把面试题类型相近或者解法相近的题目尽量放在一起，读者在学习本书时很容易看出面试题解法之间的联系，使知识的学习避免碎片化。书中将所有的面试题从难到易依次分为“将、校、尉、士”四个档次，方便读者有针对性地选择“刷”题。本书所收录的所有面试题都给出了最优解讲解和代码实现，并且提供了一些普通解法和最优解法的运行时间对比，让读者真切地感受到最优解的魅力！

本书中的题目全面且经典，更重要的是，书中收录了大量独家题目和最优解分析，这些内容源自笔者多年来“死磕自己”的深入思考。

码农们，你们做好准备在IT名企的面试中脱颖而出、一举成名了吗？这本书就是你应该拥有的“神兵利器”。当然，对需要提升算法和数据结构等方面能力的程序员而言，本书的价值也是显而易见的。

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

程序员代码面试指南：IT名企算法与数据结构题目最优解 / 左程云著. —北京：电子工业出版社，2015.9

ISBN 978-7-121-27011-6

I. ①程… II. ①左… 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(2015)第 198018 号

策划编辑：牛 勇

责任编辑：李利健

印 刷：三河市双峰印刷装订有限公司

装 订：三河市双峰印刷装订有限公司

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：33.25 字数：658.9 千字

版 次：2015 年 9 月第 1 版

印 次：2015 年 9 月第 1 次印刷

定 价：79.00 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 zltts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：(010) 88258888。

需要整本电子书，联系我QQ：3411317522

需要整本电子书，联系我QQ：3411317522

献给左军和谢桂兰

需要整本电子书，联系我QQ：3411317522

特别说明

1. 本书所有题目的代码都为 Java 实现，但这并不会妨碍其他语言使用者的阅读。这是因为笔者在实现每一道题目时，都尽最大努力回避与 Java 语言特性相关的写法出现，而且尽量遵循大多数编程语言共有的写法习惯。所以，将本书中的 Java 实现改写成其他语言的实现是非常容易的。

2. 在 Java 中，如果想得到字符串 `str` 第 i 个位置的字符，需用如下方式：

```
char p = str.charAt(i);
```

本书提供的函数中有大量参数为字符串类型的函数，但如上所示的方式并不符合大多数读者的阅读习惯。为了让代码更加易读，笔者都在这样的函数中把字符串类型的参数转换成 `char` 类型数组的变量来使用，例如：

```
char[] charArr = str.toCharArray();
```

此时得到字符串 `str` 第 i 个位置的字符，可以用如下方式：

```
char p = charArr[i];
```

在本书中，发生如上转换行为的函数在估算额外空间复杂度的时候，笔者并没有把 `charArr` 的空间计算在内，这是因为如果不转换成 `char` 数组，而是选择直接使用原参数 `str`，也是完全可以的，之所以选择转换，仅仅是为了让读者更容易读懂代码；是否进行转换对算法的逻辑没有任何影响，所以不把 `charArr` 的空间算作必须使用的额外空间。

另外，本书涉及的程序源代码可以在 <http://www.broadview.com.cn/27011> 中下载。

推荐序 1

2015 年春节，因为公司业务的快速发展，我们开始寻觅优秀的笔试面试算法讲师。几经周折，找到了当时在举办线下算法分享的程云，认真地听他讲了一堂课，当时就认定他就是我们要找的人。

我听过很多国内顶尖 ACM 选手的算法分享，但是每一次听完以后总觉得我和那些人永远隔着一个断裂带，算法对我来说遥不可及，而程云讲解算法的时候总能从最小的切口讲起，由浅入深，环环相扣，不知不觉引你走向算法的核心精髓，那种醍醐灌顶的感觉能激发大家学习算法的热情，并一直推着我们前进。

这几年 IT 技术蓬勃发展，日新月异，对技术人才的需求日益增长，程序员招聘市场也如火如荼。在有限的三五轮面试中，国外流行让面试者编程解决某些数据结构和算法的题目，通过观察面试者编码的熟练程度、思考的速度和深度来衡量面试者的能力和潜力。国内以百度、阿里、腾讯为首的互联网企业也都逐步开始采用算法面试来筛选人才。

程云出于对算法的热爱，长期泡在 `careercup`、`leetcode` 等笔试面试网站上，编码解决各种最新的笔试面试编程题，对各种笔试面试编程题的解题技巧了如指掌。

算法面试普及后，传统的数据结构和算法课本讲得太过基础，又远离求职需求，国内也逐渐出现迎合求职需求的笔试面试工具书，这些书籍有些过于应试，纯粹以通过面试为导向，程云的书和那些书相比，题目更前沿，讲解更注重思考思路和代码的实践技巧，对每个题目都深挖最优解，同时根据自己在线下讲课学员们的反馈，对每个编程考题的解题反复修改，让思路更清晰。

这本书不仅可以作为面试代码指南，还可以作为学生课后的辅助练习，“刷”题 5 年，悉数总结都沉淀在这本书里，相信读者跟着他的引导从头到尾逐一攻克一定会有所收获。

叶向宇
牛客网 CEO

推荐序 2

初次遇进程云是在 2014 年 8 月，当时我在上一家公司工作刚好满 4 年，也是在那时我开始想换个环境，寻找新机会，就试着投了一家公司，结果第一次面试遇到算法题就被淘汰了。后来又面试过其他一些国内互联网公司，也总是卡在算法上。其实，之前我曾经自己在家抱着《算法导论》“啃”了几章，花了 1 个月的业余时间看了前 5 章，后面就没再继续坚持下去。看过的人都知道，虽然很有用，但实在很难“啃”。

单调地看书很枯燥，于是想到去网上找志同道合的人一起研究，就开始“逛”算法论坛。很巧的是，在某个论坛的算法板块看到一个帖子，说是在周末有算法交流班，当时我立即报名，周日的名额已满，我是很幸运地“替补”上去的。

还记得第一次交流是在程云租的房子里，小小的客厅里放了一张沙发、两排椅子和一张桌子，桌上放着笔记本电脑和一台大电视，前面还挂着白板。第一次算法交流就在这样的环境里开始了。

程云讲起题来犹如行云流水，我们听得更是酣畅淋漓，第一次听完就爱上了……当然，我说的是他的讲述。

相信大家都有过这样的经历，面对一道算法题，苦思冥想了半天，还是不知道怎么解，感觉很沮丧。如果这时突然有人把解题思路和方法以及代码都告诉你了，是不是感觉豁然开朗，心情舒畅了？这样的情景一天出现一次就可以让人感觉很开心，而如果一天连续出现二十次，那将会是什么感觉？一个字：爽！

程云把每一道题都讲解得清晰透彻，有的题目难以理解、思路诡异，他就会不厌其烦地反复讲解，用形象的方式展现复杂的逻辑，直到大家都听懂为止。给人的感觉可以说是高潮迭起，一波又一波。

后来进行第二次交流时，我带来最好的朋友一起参加。之后的交流中，我和朋友都毫不犹豫地报名参加。交流的内容涉及经典算法的高难度题目，也有一些小巧玲珑的技巧题。难题难得让人叹服，巧题巧得让人玩味。

对想去国外大公司就职的程序员来说，算法题这一关是必不可少的。程云讲述的题目

需要整本电子书，联系我QQ：3411317522

程序员代码面试指南：IT 名企算法与数据结构题目最优解

是他 5 年“刷”题的经验积累而成的，其实只要掌握题目的解题思路 and 思想，就足以应付国内互联网公司程序员职位的算法面试题。不过，要想去国外的大公司，比如 Google、Facebook 之类的，还是要研究得透彻一些才行。

另外，除应付面试之外，还有很重要的一点，甚至是更重要的一点，就是本书可以帮助我们打开思路，因为很多算法题的解法是需要逆向思维的，需要跳出原有的固定思维模式，当思维模式被打开之后，你会发现原有的事物现在看起来会有不同的看法，因为角度变了。不过这只能自己体会。

后来才知道，程云举办算法交流是为写书做准备。用他的话说：“会做题不算什么，比我“刷”题多的人我也能找出一大堆，但能给人讲明白就不容易了。”于是我后来又变成了程云在写这本书期间的试读者。

在此书还未上市之前，就能听到作者面对面地逐一讲解每一道题，真是非常难得且宝贵的经历。

如果你和我一样，对数据结构有个大概的了解，很想快速掌握算法题的解法技巧，那么这本书一定适合你！

祝每一位勤奋努力的程序员都能拿到自己满意的职位！

周宝鑫
一个程序员

需要整本电子书，联系我QQ：3411317522

自序

我能出书挺意外的。

在 6 年前的某一天，虽然我早就知道想进入那些大公司要靠“刷”代码面试题来练习编写代码的能力。可是这一天却不止如此，我突然有了心情去看代码面试题长什么样子，于是收集了代码面试的题目，越深入，我越有一种恐慌的感觉，因为感觉自己什么都不太在行，对一个归并排序（Merge sort）写出完整的代码都感觉挺费劲的，面对这个冯·诺伊曼发明的排序算法，我真有底气说自己是计算机专业的学生吗？这种打击并没有持续太久，因为爱耍小聪明的人总会特别自信。我决定开始认真面对“刷”题这件事，但那时我根本不知道我即将面对什么，更不要谈有写书的念头。

我把课余时间利用起来，心想：不就是“刷”题吗？别人能写出来，咱也能写出来。起初的心态是我不服，我就想告诉自己能行。过程虐心是肯定的，经常半夜因为看到一个复杂度特别低的算法自己真的不能理解而沮丧地睡不着觉。当时觉得找不到什么资料能彻底让我明白，书上讲得太粗浅，网上的太散乱，代码写得看不懂。起初我“刷”题的时候无数次地想放弃，因为觉得这些都是什么玩意儿！我为什么放着好好的日子不过，去找这种罪受？可是我又不甘心，虽然我不懂很多解法，但是它们真的很有意思。

我将能买到的所有相关书籍上的所有题目全都研究了一遍，不管是中文的还是英文的，我都硬着头皮“啃”。写完每道题后，我都和书上的方法进行反复对比。“啃”完了五六本书之后，距离我刚开始“刷”题已经过去 16 个月了。写书？别逗了，才刚看完。

“年轻人总会找借口说这个东西不是我感兴趣的，所以我做不好是应该的。但他们没有注意的是，你面对的事情中感兴趣的事情总是少数，这就使得大多数时候你做事情的态度总是很懈怠、很消极，这使你变成了一个懈怠的人。当你真正面对自己感兴趣的東西时，你发现你已经攥不紧拳头了。”时常想起本科时的毕业设计指导老师——高鹏义老师说的这句话。说得对！对一个东西，如果你没有透彻研究过，不要轻易说它不精彩。这不是博爱，而是对自己认真。

“刷”题代码达到 4 万行的时候，我基本上成了国内外所有热门“刷”题网站的日常用户，此时我确认了一件事情，今天的代码面试指导真的处在一个很初级的阶段，这种不健全是

全方面的。

例如：

- 经常看到一篇文章前后的语境是割裂的，作者经常根据之前的一个优良解法提出更好的优化方式，但整篇文章都不提之前的解法是什么。这就导致初学者根本无法看懂；
- 几乎所有的书籍都忽略例子带来的引导作用，甚至还有不少书籍在阐述一个解法的时候只写伪代码，这就使得读者在看懂意思和自己真正能写出代码之间其实还有很多的路要走；
- 代码面试题的特点是“多”、“杂”、“难”，从着手开始学习到最终达到自己想要的效果之间，自己对自己的评估根本无从谈起。“慢慢练吧，学海无涯”成为主要的心态，这就难免会产生怀疑的情绪；
- 看见一道新的面试题时还是会无从下手，因为之前的学习无法做到举一反三，对自己做过的题目缺乏总结和归纳。

难道“刷”题真的只适合聪明人玩？我不这么看，既然大多数内容处在有待商榷的地步，那我就去学习原论文吧。

当时一个人在国外，记得在初冬的一个下午，“刷”题已经两年之久，快吃晚饭的时候，我突然想起自己忘了吃午饭，就冲出家门去觅食。站在 7-11 门前的广场上，我拿着 1.5 美元的热狗和 75 美分的咖啡，微温的阳光撒在眼睛里，远远地望着即将消失的一天。我停下来，把咖啡放在斑驳的石头台子上，手里的热狗挺好看，香肠和洋葱都挺新鲜，清冷的空气吹过来，却让我的心绪更乱。旧金山的天空五彩斑斓，让漂泊者头晕目眩。哭得跟个鬼似的我除了想家，哪里敢想自己会出书呢？

当我意识到在网上很难搜到新鲜的题目时，我已经换了两家公司，反复实现了 600 多道题目，写了差不多 10 万行代码。原来只是为了找份工作而“刷”题这一初心早就忘了，变成了兴趣并坚持了这么久，我自己也感到意外。更奇怪的是，我已经完全乐在其中，同时交流欲望越来越强，时常和同事们展开这方面的讨论。发现很多书上的解法不是最优，很多题目其实和同事们讨论的做法更好，发现高手特别多，但好像都懒得动笔。

有一天，我看到自己写的题目，想到自己那些抓心挠肝的日子，突然觉得要不出书吧？我已经离不开这种感觉了，如果这不是真爱，那什么才是呢？

这不是一个励志的故事，是一个爱“刷”题的人决定把很多最优解讲出来，就这么简单。

左程云

2015 年 7 月 20 日

目 录

第 1 章 栈和队列 1

 设计一个有 getMin 功能的栈（士 ★☆☆☆）1

 由两个栈组成的队列（尉 ★★☆☆）5

 如何仅用递归函数和栈操作逆序一个栈（尉 ★★☆☆）8

 猫狗队列（士 ★☆☆☆）10

 用一个栈实现另一个栈的排序（士 ★☆☆☆）13

 用栈来求解汉诺塔问题（校 ★★★☆）14

 生成窗口最大值数组（尉 ★★☆☆）19

 构造数组的 MaxTree（校 ★★★☆）22

 求最大子矩阵的大小（校 ★★★☆）26

 最大值减去最小值小于或等于 num 的子数组数量（校 ★★★☆）31

第 2 章 链表问题 34

 打印两个有序链表的公共部分（士 ★☆☆☆）34

 在单链表和双链表中删除倒数第 K 个节点（士 ★☆☆☆）35

 删除链表的中间节点和 a/b 处的节点（士 ★☆☆☆）38

 反转单向和双向链表（士 ★☆☆☆）40

 反转部分单向链表（士 ★☆☆☆）42

 环形单链表的约瑟夫问题（原问题：士 ★☆☆☆ 进阶：校 ★★★☆）43

 判断一个链表是否为回文结构（普通解法 士 ★☆☆☆）
 （进阶解法 尉 ★★☆☆）48

 将单向链表按某值划分成左边小、中间相等、右边大的形式（尉 ★★☆☆）52

 复制含有随机指针节点的链表（尉 ★★☆☆）56

 两个单链表生成相加链表（士 ★☆☆☆）59

两个单链表相交的一系列问题（将 ★★★★★）	62
将单链表的每 K 个节点之间逆序（尉 ★★☆☆）	68
删除无序单链表中值重复出现的节点（士 ★☆☆☆）	71
在单链表中删除指定值的节点（士 ★☆☆☆）	73
将搜索二叉树转换成双向链表（尉 ★★☆☆）	74
单链表的选择排序（士 ★☆☆☆）	79
一种怪异的节点删除方式（士 ★☆☆☆）	81
向有序的环形单链表中插入新节点（士 ★☆☆☆）	82
合并两个有序的单链表（士 ★☆☆☆）	84
按照左右半区的方式重新组合单链表（士 ★☆☆☆）	86
 第 3 章 二叉树问题	 88
分别用递归和非递归方式实现二叉树先序、中序和后序遍历（校 ★★★★★）	88
打印二叉树的边界节点（尉 ★★☆☆）	95
如何较为直观地打印二叉树（尉 ★★☆☆）	100
二叉树的序列化和反序列化（士 ★☆☆☆）	103
遍历二叉树的神级方法（将 ★★★★★）	107
在二叉树中找到累加和为指定值的最长路径长度（尉 ★★☆☆）	115
找到二叉树中的最大搜索二叉子树（尉 ★★☆☆）	117
找到二叉树中符合搜索二叉树条件的最大拓扑结构（校 ★★★★★）	119
二叉树的按层打印与 ZigZag 打印（尉 ★★☆☆）	129
调整搜索二叉树中两个错误的节点（原问题：尉 ★★☆☆）	
（进阶问题：将 ★★★★★）	134
判断 t_1 树是否包含 t_2 树全部的拓扑结构（士 ★☆☆☆）	140
判断 t_1 树中是否有与 t_2 树拓扑结构完全相同的子树（校 ★★★★★）	141
判断二叉树是否为平衡二叉树（士 ★☆☆☆）	144
根据后序数组重建搜索二叉树（士 ★☆☆☆）	145
判断一棵二叉树是否为搜索二叉树和完全二叉树（士 ★☆☆☆）	147
通过有序数组生成平衡搜索二叉树（士 ★☆☆☆）	150
在二叉树中找到一个节点的后继节点（尉 ★★☆☆）	151
在二叉树中找到两个节点的最近公共祖先（原问题：士 ★☆☆☆）	
（进阶问题：尉 ★★☆☆ 再进阶问题：校 ★★★★★）	153

Tarjan 算法与并查集解决二叉树节点间最近公共祖先的批量查询问题	
(校 ★★★☆)	159
二叉树节点间的最大距离问题 (尉 ★★★☆)	169
先序、中序和后序数组两两结合重构二叉树 (先序与中序结合 士 ★☆☆☆)	
(中序与后序结合 士 ★☆☆☆ 先序与后序结合 尉 ★★★☆)	171
通过先序和中序数组生成后序数组 (士 ★☆☆☆)	174
统计和生成所有不同的二叉树 (尉 ★★★☆)	175
统计完全二叉树的节点数 (尉 ★★★☆)	178
第 4 章 递归和动态规划	181
斐波那契系列问题的递归和动态规划 (将 ★★★★★)	181
矩阵的最小路径和 (尉 ★★★☆)	187
换钱的最少货币数 (尉 ★★★☆)	191
换钱的方法数 (尉 ★★★☆)	196
最长递增子序列 (校 ★★★★★)	202
汉诺塔问题 (校 ★★★★★)	206
最长公共子序列问题 (尉 ★★★☆)	210
最长公共子串问题 (校 ★★★★★)	213
最小编辑代价 (校 ★★★★★)	217
字符串的交错组成 (校 ★★★★★)	220
龙与地下城游戏问题 (尉 ★★★☆)	223
数字字符串转换为字母组合的种数 (尉 ★★★☆)	225
表达式得到期望结果的组成种数 (校 ★★★★★)	228
排成一条线的纸牌博弈问题 (尉 ★★★☆)	233
跳跃游戏 (士 ★☆☆☆)	235
数组中的最长连续序列 (尉 ★★★☆)	236
N 皇后问题 (校 ★★★★★)	238
第 5 章 字符串问题	242
判断两个字符串是否互为变形词 (士 ★☆☆☆)	242
字符串中数字子串的求和 (士 ★☆☆☆)	243
去掉字符串中连续出现 k 个 0 的子串 (士 ★☆☆☆)	245
判断两个字符串是否互为旋转词 (士 ★☆☆☆)	247

将整数字符串转成整数值（尉 ★★☆☆）	248
替换字符串中连续出现的指定字符串（士 ★☆☆☆）	251
字符串的统计字符串（士 ★☆☆☆）	253
判断字符串数组中是否所有的字符都只出现过一次 （按要求 1 实现的方法 士 ★☆☆☆）	255
（按要求 2 实现的方法 尉 ★★☆☆）	258
在有序但含有空的数组中查找字符串（尉 ★★☆☆）	260
字符串的调整与替换（士 ★☆☆☆）	262
翻转字符串（士 ★☆☆☆）	266
数组中两个字符串的最小距离（尉 ★★☆☆）	269
添加最少字符使字符串整体都是回文字符串（校 ★★★★★）	273
括号字符串的有效性和最长有效长度（原问题 士 ★☆☆☆） （补充问题 尉 ★★☆☆）	276
公式字符串求值（校 ★★★★★）	278
0 左边必有 1 的二进制字符串数量（校 ★★★★★）	281
拼接所有字符串产生字典顺序最小的大写字母串（校 ★★★★★）	284
找到字符串的最长无重复字符串（尉 ★★☆☆）	286
找到被指的新类型字符（士 ★☆☆☆）	288
最小包含子串的长度（校 ★★★★★）	292
回文最少分割数（尉 ★★★★★）	294
字符串匹配问题（校 ★★★★★）	299
字典树（前缀树）的实现（尉 ★★☆☆）	

第 6 章 大数据和空间限制..... 303

认识布隆过滤器（尉 ★★☆☆）	303
只用 2GB 内存在 20 亿个整数中找到出现次数最多的数（士 ★☆☆☆）	308
40 亿个非负整数中找到没出现的数（尉 ★★☆☆）	309
找到 100 亿个 URL 中重复的 URL 以及搜索词汇的 top K 问题（士 ★☆☆☆）	311
40 亿个非负整数中找到出现两次的数和所有数的中位数（尉 ★★☆☆）	312
一致性哈希算法的基本原理（尉 ★★☆☆）	313

第 7 章 位运算..... 317

不用额外变量交换两个整数的值（士 ★☆☆☆）	317
------------------------------	-----

不用任何比较判断找出两个数中较大的数（校 ★★★☆）	318
只用位运算不用算术运算实现整数的加减乘除运算（尉 ★★☆☆）	319
整数的二进制表达中有多少个 1（尉 ★★☆☆）	325
在其他数都出现偶数次的数组中找到出现奇数次的数（尉 ★★☆☆）	327
在其他数都出现 k 次的数组中找到只出现一次的数（尉 ★★☆☆）	329
第 8 章 数组和矩阵问题	331
转圈打印矩阵（士 ★☆☆☆）	331
将正方形矩阵顺时针转动 90° （士 ★☆☆☆）	333
“之”字形打印矩阵（士 ★☆☆☆）	335
找到无序数组中最小的 k 个数（ $O(N\log k)$ 的方法 尉 ★★☆☆） （ $O(N)$ 的方法 将 ★★★★★）	336
需要排序的最短子数组长度（士 ★☆☆☆）	342
在数组中找到出现次数大于 N/K 的数（校 ★★☆☆）	343
在行列都排好序的矩阵中找数（士 ★☆☆☆）	347
最长的可整合子数组的长度（尉 ★★☆☆）	349
不重复打印排序数组中相加和为给定值的所有二元组和三元组 （尉 ★★☆☆）	351
未排序正数数组中累加和为给定值的最长子数组长度（尉 ★★☆☆）	354
未排序数组中累加和为给定值的最长子数组系列问题（尉 ★★☆☆）	355
未排序数组中累加和小于或等于给定值的最长子数组长度（校 ★★☆☆）	358
计算数组的小和（校 ★★☆☆）	361
自然数数组的排序（士 ★☆☆☆）	364
奇数下标都是奇数或者偶数下标都是偶数（士 ★☆☆☆）	366
子数组的最大累加和问题（士 ★☆☆☆）	367
子矩阵的最大累加和问题（尉 ★★☆☆）	368
在数组中找到一个局部最小的位置（尉 ★★☆☆）	371
数组中子数组的最大累乘积（尉 ★★☆☆）	373
打印 N 个数组整体最大的 Top K （尉 ★★☆☆）	374
边界都是 1 的最大正方形大小（尉 ★★☆☆）	377
不包含本位置值的累乘数组（士 ★☆☆☆）	380
数组的 partition 调整（士 ★☆☆☆）	382
求最短通路值（尉 ★★☆☆）	384

数组中未出现的最小正整数（尉 ★★★☆☆）	386
数组排序之后相邻数的最大差值（尉 ★★★☆☆）	388
第 9 章 其他题目	390
从 5 随机到 7 随机及其扩展（原问题 尉 ★★★☆☆ 补充问题 尉 ★★★☆☆） （进阶问题 校 ★★★★★）	390
一行代码求两个数的最大公约数（士 ★★★☆☆）	394
有关阶乘的两个问题（原问题 尉 ★★★☆☆ 进阶问题 校 ★★★★★）	395
判断一个点是否在矩形内部（尉 ★★★☆☆）	398
判断一个点是否在三角形内部（尉 ★★★☆☆）	399
折纸问题（尉 ★★★☆☆）	402
蓄水池算法（尉 ★★★☆☆）	404
设计有 setAll 功能的哈希表（士 ★☆☆☆☆）	406
最大的 leftMax 与 rightMax 之差的绝对值（校 ★★★★★）	408
设计可以变更的缓存结构（尉 ★★★☆☆）	410
设计 RandomPool 结构（尉 ★★★☆☆）	414
调整 $[0,x]$ 区间上的数出现的概率（士 ★☆☆☆☆）	416
路径数组变为统计数组（校 ★★★★★）	417
正数数组的最小不可组成和（尉 ★★★☆☆）	422
一种字符串和数字的对应关系（校 ★★★★★）	426
1 到 n 中 1 出现的次数（校 ★★★★★）	429
从 N 个数中等概率打印 M 个数（士 ★☆☆☆☆）	431
判断一个数是否是回文数（士 ★☆☆☆☆）	433
在有序旋转数组中找到最小值（尉 ★★★☆☆）	434
在有序旋转数组中找到一个数（尉 ★★★☆☆）	436
数字的英文表达和中文表达（校 ★★★★★）	439
分糖果问题（校 ★★★★★）	444
一种消息接收并打印的结构设计（尉 ★★★☆☆）	448
设计一个没有扩容负担的堆结构（将 ★★★★★）	451
随时找到数据流的中位数（将 ★★★★★）	462
在两个长度相等的排序数组中找到上中位数（尉 ★★★☆☆）	465
在两个排序数组中找到第 K 小的数（将 ★★★★★）	468
两个有序数组间相加和的 TOP K 问题（尉 ★★★☆☆）	471

出现次数的 TOP K 问题（原问题 尉 ★★☆☆ 进阶问题 校 ★★★★★）	474
Manacher 算法（将 ★★★★★）	483
KMP 算法（将 ★★★★★）	491
丢棋子问题（校 ★★★★★）	498
画匠问题（校 ★★★★★）	505
邮局选址问题（校 ★★★★★）	509

第 1 章

栈和队列

设计一个有 getMin 功能的栈

【题目】

实现一个特殊的栈，在实现栈的基本功能的基础上，再实现返回栈中最小元素的操作。

【要求】

1. pop、push、getMin 操作的时间复杂度都是 $O(1)$ 。
2. 设计的栈类型可以使用现成的栈结构。

【难度】

士 ★☆☆☆

【解答】

在设计上我们使用两个栈，一个栈用来保存当前栈中的元素，其功能和一个正常的栈没有区别，这个栈记为 `stackData`；另一个栈用于保存每一步的最小值，这个栈记为 `stackMin`。具体的实现方式有两种。

第一种设计方案如下。

- 压入数据规则

假设当前数据为 `newNum`，先将其压入 `stackData`。然后判断 `stackMin` 是否为空：

- 如果为空，则 newNum 也压入 stackMin。
- 如果不为空，则比较 newNum 和 stackMin 的栈顶元素中哪一个更小：
- 如果 newNum 更小或两者相等，则 newNum 也压入 stackMin；
- 如果 stackMin 中栈顶元素小，则 stackMin 不压入任何内容。

举例：依次压入 3、4、5、1、2、1 的过程中，stackData 和 stackMin 的变化如图 1-1 所示。

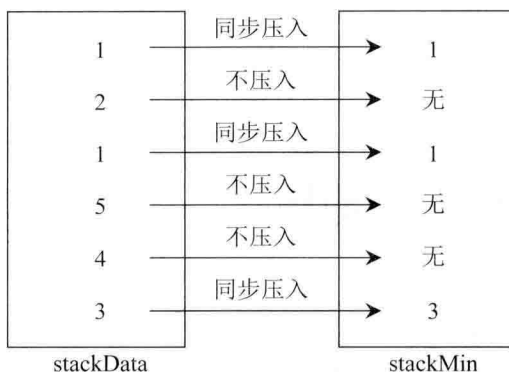


图 1-1

• 弹出数据规则

先在 stackData 中弹出栈顶元素，记为 value。然后比较当前 stackMin 的栈顶元素和 value 哪一个更小。

通过上文提到的压入规则可知，stackMin 中存在的元素是从栈底到栈顶逐渐变小的，stackMin 栈顶的元素既是 stackMin 栈的最小值，也是当前 stackData 栈的最小值。所以不会出现 value 比 stackMin 的栈顶元素更小的情况，value 只可能大于或等于 stackMin 的栈顶元素。

当 value 等于 stackMin 的栈顶元素时，stackMin 弹出栈顶元素；当 value 大于 stackMin 的栈顶元素时，stackMin 不弹出栈顶元素；返回 value。

很明显可以看出，压入与弹出规则是对应的。

• 查询当前栈中的最小值操作

由上文的压入数据规则和弹出数据规则可知，stackMin 始终记录着 stackData 中的最小值，所以，stackMin 的栈顶元素始终是当前 stackData 中的最小值。

方案一的代码实现如 `MyStack1` 类所示：

```
public class MyStack1 {
    private Stack<Integer> stackData;
    private Stack<Integer> stackMin;

    public MyStack1() {
        this.stackData = new Stack<Integer>();
        this.stackMin = new Stack<Integer>();
    }

    public void push(int newNum) {
        if (this.stackMin.isEmpty()) {
            this.stackMin.push(newNum);
        } else if (newNum <= this.getmin()) {
            this.stackMin.push(newNum);
        }
        this.stackData.push(newNum);
    }

    public int pop() {
        if (this.stackData.isEmpty()) {
            throw new RuntimeException("Your stack is empty.");
        }
        int value = this.stackData.pop();
        if (value == this.getmin()) {
            this.stackMin.pop();
        }
        return value;
    }

    public int getmin() {
        if (this.stackMin.isEmpty()) {
            throw new RuntimeException("Your stack is empty.");
        }
        return this.stackMin.peek();
    }
}
```

第二种设计方案如下。

- 压入数据规则

假设当前数据为 `newNum`，先将其压入 `stackData`。然后判断 `stackMin` 是否为空。

如果为空，则 `newNum` 也压入 `stackMin`；如果不为空，则比较 `newNum` 和 `stackMin` 的栈顶元素中哪一个更小：

如果 `newNum` 更小或两者相等，则 `newNum` 也压入 `stackMin`；如果 `stackMin` 中栈顶元素小，则把 `stackMin` 的栈顶元素重复压入 `stackMin`，即在栈顶元素上再压入一个栈顶

元素。

举例：依次压入 3、4、5、1、2、1 的过程中，stockData 和 stackMin 的变化如图 1-2 所示。

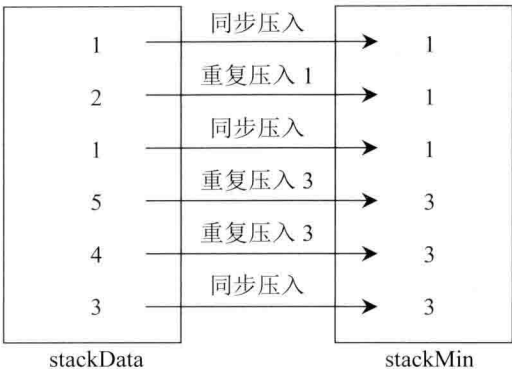


图 1-2

- 弹出数据规则

在 stackData 中弹出数据，弹出的数据记为 value；弹出 stackMin 中的栈顶；返回 value。很明显可以看出，压入与弹出规则是对应的。

- 查询当前栈中的最小值操作

由上文的压入数据规则和弹出数据规则可知，stackMin 始终记录着 stackData 中的最小值，所以 stackMin 的栈顶元素始终是当前 stackData 中的最小值。

方案二的代码实现如 MyStack2 类所示：

```
public class MyStack2 {
    private Stack<Integer> stackData;
    private Stack<Integer> stackMin;

    public MyStack2() {
        this.stackData = new Stack<Integer>();
        this.stackMin = new Stack<Integer>();
    }

    public void push(int newNum) {
        if (this.stackMin.isEmpty()) {
            this.stackMin.push(newNum);
        } else if (newNum < this.getmin()) {
            this.stackMin.push(newNum);
        } else {
            // ...
        }
    }
}
```

```
        int newMin = this.stackMin.peek();
        this.stackMin.push(newMin);
    }
    this.stackData.push(newNum);
}

public int pop() {
    if (this.stackData.isEmpty()) {
        throw new RuntimeException("Your stack is empty.");
    }
    this.stackMin.pop();
    return this.stackData.pop();
}

public int getmin() {
    if (this.stackMin.isEmpty()) {
        throw new RuntimeException("Your stack is empty.");
    }
    return this.stackMin.peek();
}
}
```

【点评】

方案一和方案二其实都是用 `stackMin` 栈保存着 `stackData` 每一步的最小值。共同点是所有操作的时间复杂度都为 $O(1)$ 、空间复杂度都为 $O(n)$ 。区别是：方案一中 `stackMin` 压入时稍省空间，但是弹出操作稍费时间；方案二中 `stackMin` 压入时稍费空间，但是弹出操作稍省时间。

由两个栈组成的队列

【题目】

编写一个类，用两个栈实现队列，支持队列的基本操作（`add`、`poll`、`peek`）。

【难度】

尉 ★★☆☆

【解答】

栈的特点是先进后出，而队列的特点是先进先出。我们用两个栈正好能把顺序反过来实现类似队列的操作。

具体实现上是一个栈作为压入栈，在压入数据时只往这个栈中压入，记为 `stackPush`；另一个栈只作为弹出栈，在弹出数据时只从这个栈弹出，记为 `stackPop`。

因为数据压入栈的时候，顺序是先进后出的。那么只要把 `stackPush` 的数据再压入 `stackPop` 中，顺序就变回来了。例如，将 1~5 依次压入 `stackPush`，那么从 `stackPush` 的栈顶到栈底为 5~1，此时依次再将 5~1 倒入 `stackPop`，那么从 `stackPop` 的栈顶到栈底就变成了 1~5。再从 `stackPop` 弹出时，顺序就像队列一样，如图 1-3 所示。

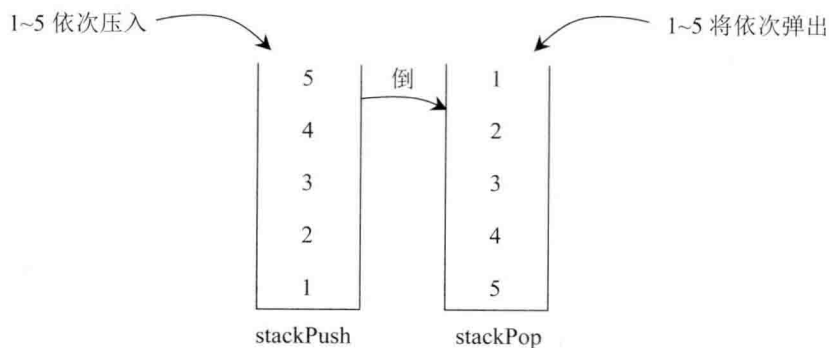


图 1-3

听起来虽然简单，实际上必须做到以下两点。

1. 如果 `stackPush` 要往 `stackPop` 中压入数据，那么必须一次性把 `stackPush` 中的数据全部压入。
 2. 如果 `stackPop` 不为空，`stackPush` 绝对不能向 `stackPop` 中压入数据。
- 违反了以上两点都会发生错误。

违反 1 的情况举例：1~5 依次压入 `stackPush`，`stackPush` 的栈顶到栈底为 5~1，从 `stackPush` 压入 `stackPop` 时，只将 5 和 4 压入了 `stackPop`，`stackPush` 还剩下 1、2、3 没有压入。此时如果用户想进行弹出操作，那么 4 将最先弹出，与预想的队列顺序就不一致。

违反 2 的情况举例：1~5 依次压入 `stackPush`，`stackPush` 将所有数据压入了 `stackPop`，此时从 `stackPop` 的栈顶到栈底就变成了 1~5。此时又有 6~10 依次压入 `stackPush`，`stackPop` 不为空，`stackPush` 不能向其中压入数据。如果违反 2 压入了 `stackPop`，从 `stackPop` 的栈顶到栈底就变成了 6~10、1~5。那么此时如果用户想进行弹出操作，6 将最先弹出，与预想的队列顺序就不一致。

上面介绍了压入数据的注意事项。那么这个压入数据的操作在何时发生呢？

这个选择的时机可以有很多，调用 `add`、`poll` 和 `peek` 三种方法中的任何一种时发生“压”入数据的行为都是可以的。只要满足如上提到的两点，就不会出错。

本书的实现是在调用 `poll` 和 `peek` 方法进行压入数据的过程。

具体实现请参看如下的 `TwoStacksQueue` 类：

```
public class TwoStacksQueue {
    public Stack<Integer> stackPush;
    public Stack<Integer> stackPop;

    public TwoStacksQueue() {
        stackPush = new Stack<Integer>();
        stackPop = new Stack<Integer>();
    }

    public void add(int pushInt) {
        stackPush.push(pushInt);
    }

    public int poll() {
        if (stackPop.empty() && stackPush.empty()) {
            throw new RuntimeException("Queue is empty!");
        } else if (stackPop.empty()) {
            while (!stackPush.empty()) {
                stackPop.push(stackPush.pop());
            }
        }
        return stackPop.pop();
    }

    public int peek() {
        if (stackPop.empty() && stackPush.empty()) {
            throw new RuntimeException("Queue is empty!");
        } else if (stackPop.empty()) {
            while (!stackPush.empty()) {
                stackPop.push(stackPush.pop());
            }
        }
        return stackPop.peek();
    }
}
```

如何仅用递归函数和栈操作逆序一个栈

【题目】

一个栈依次压入 1、2、3、4、5，那么从栈顶到栈底分别为 5、4、3、2、1。将这个栈转置后，从栈顶到栈底为 1、2、3、4、5，也就是实现栈中元素的逆序，但是只能用递归函数来实现，不能用其他数据结构。

【难度】

尉 ★★☆☆

【解答】

本题考查栈的操作和递归函数的设计，我们需要设计出两个递归函数。

递归函数一：将栈 `stack` 的栈底元素返回并移除。

具体过程就是如下代码中的 `getAndRemoveLastElement` 方法。

```
public static int getAndRemoveLastElement(Stack<Integer> stack) {  
    int result = stack.pop();  
    if (stack.isEmpty()) {  
        return result;  
    } else {  
        int last = getAndRemoveLastElement(stack);  
        stack.push(result);  
        return last;  
    }  
}
```

如果从 `stack` 的栈顶到栈底依次为 3、2、1，这个函数的具体过程如图 1-4 所示。

递归函数二：逆序一个栈，就是题目要求实现的方法，具体过程就是如下代码中的 `reverse` 方法。该方法使用了上面提到的 `getAndRemoveLastElement` 方法。

```
public static void reverse(Stack<Integer> stack) {  
    if (stack.isEmpty()) {  
        return;  
    }  
    int i = getAndRemoveLastElement(stack);  
    reverse(stack);  
    stack.push(i);  
}
```

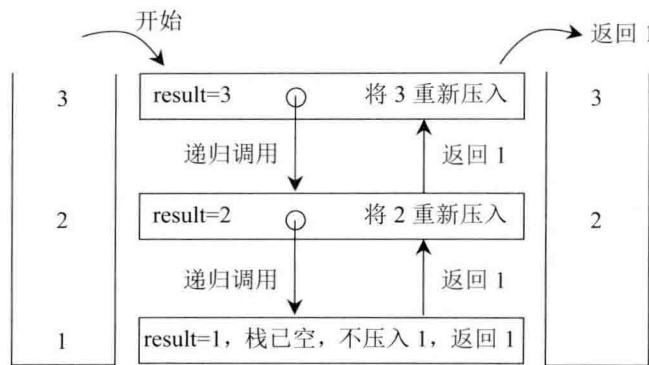


图 1-4

如果从 stack 的栈顶到栈底依次为 3、2、1，reverse 函数的具体过程如图 1-5 所示。

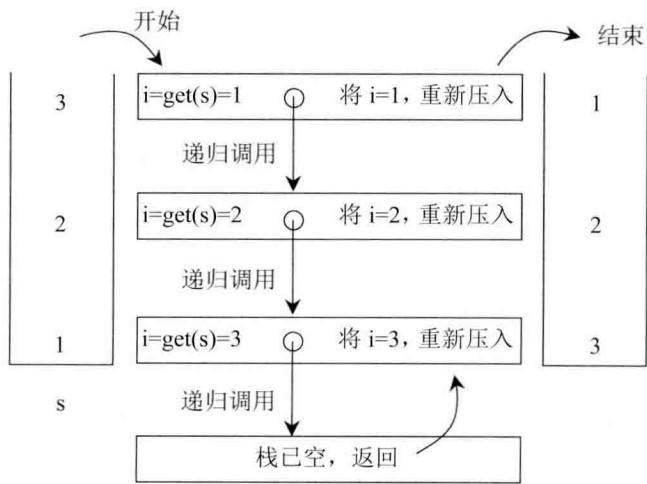


图 1-5

getAndRemoveLastElement 方法在图中简单表示为 get 方法，表示移除并返回当前栈底元素。

猫狗队列

【题目】

宠物、狗和猫的种类如下：

```
public class Pet {
    private String type;

    public Pet(String type) {
        this.type = type;
    }

    public String getPetType() {
        return this.type;
    }
}

public class Dog extends Pet {
    public Dog() {
        super("dog");
    }
}

public class Cat extends Pet {
    public Cat() {
        super("cat");
    }
}
```

实现一种狗猫队列的结构，要求如下：

- 用户可以调用 add 方法将 cat 类或 dog 类的实例放入队列中；
- 用户可以调用 pollAll 方法，将队列中所有的实例按照进队列的先后顺序依次弹出；
- 用户可以调用 pollDog 方法，将队列中 dog 类的实例按照进队列的先后顺序依次弹出；
- 用户可以调用 pollCat 方法，将队列中 cat 类的实例按照进队列的先后顺序依次弹出；
- 用户可以调用 isEmpty 方法，检查队列中是否还有 dog 或 cat 的实例；
- 用户可以调用 isDogEmpty 方法，检查队列中是否有 dog 类的实例；
- 用户可以调用 isCatEmpty 方法，检查队列中是否有 cat 类的实例。

【难度】

士 ★☆☆☆

【解答】

本题考查实现特殊数据结构的能力以及针对特殊功能的算法设计能力。

本题为开放类型的面试题，希望读者能有自己的实现，在这里列出几种常见的设计错误：

- cat 队列只放 cat 实例，dog 队列只放 dog 实例，再用一个总队列放所有的实例。
错误原因：cat、dog 以及总队列的更新问题。
- 用哈希表，key 表示一个 cat 实例或 dog 实例，value 表示这个实例进队列的次序。
错误原因：不能支持一个实例多次进队列的功能需求，因为哈希表的 key 只能对应一个 value 值。
- 将用户原有的 cat 或 dog 类改写，加一个计数项来表示某一个实例进队列的时间。
错误原因：不能擅自改变用户的类结构。

本题实现将不同的实例盖上时间戳的方法，但是又不能改变用户本身的类，所以定义一个新的类，具体实现请参看如下的 PetEnterQueue 类。

```
public class PetEnterQueue {
    private Pet pet;
    private long count;

    public PetEnterQueue(Pet pet, long count) {
        this.pet = pet;
        this.count = count;
    }

    public Pet getPet() {
        return this.pet;
    }

    public long getCount() {
        return this.count;
    }

    public String getEnterPetType() {
        return this.pet.getPetType();
    }
}
```

PetEnterQueue 类在构造时，pet 是用户原有的实例，count 就是这个实例的时间戳。

我们实现的队列其实是 `PetEnterQueue` 类的实例。大体说来，首先有一个不断累加的数项，用来表示实例进队列的时间；同时有两个队列，一个是只放 `dog` 类实例的队列 `dogQ`，另一个是只放 `cat` 类实例的队列 `catQ`。

在加入实例时，如果实例是 `dog`，就盖上时间戳，生成对应的 `PetEnterQueue` 类的实例，然后放入 `dogQ`；如果实例是 `cat`，就盖上时间戳，生成对应的 `PetEnterQueue` 类的实例，然后放入 `catQ`。具体过程请参看如下 `DogCatQueue` 类的 `add` 方法。

只想弹出 `dog` 类的实例时，从 `dogQ` 里不断弹出即可，具体过程请参看如下 `DogCatQueue` 类的 `pollDog` 方法。

只想弹出 `cat` 类的实例时，从 `catQ` 里不断弹出即可，具体过程请参看如下 `DogCatQueue` 类的 `pollCat` 方法。

想按实际顺序弹出实例时，因为 `dogQ` 的队列头表示所有 `dog` 实例中最早进队列的实例，同时 `catQ` 的队列头表示所有的 `cat` 实例中最早进队列的实例。则比较这两个队列头的时间戳，谁更早，就弹出谁。具体过程请参看如下 `DogCatQueue` 类的 `pollAll` 方法。

`DogCatQueue` 类的整体代码如下：

```
public class DogCatQueue {
    private Queue<PetEnterQueue> dogQ;
    private Queue<PetEnterQueue> catQ;
    private long count;

    public DogCatQueue() {
        this.dogQ = new LinkedList<PetEnterQueue>();
        this.catQ = new LinkedList<PetEnterQueue>();
        this.count = 0;
    }

    public void add(Pet pet) {
        if (pet.getPetType().equals("dog")) {
            this.dogQ.add(new PetEnterQueue(pet, this.count++));
        } else if (pet.getPetType().equals("cat")) {
            this.catQ.add(new PetEnterQueue(pet, this.count++));
        } else {
            throw new RuntimeException("err, not dog or cat");
        }
    }

    public Pet pollAll() {
        if (!this.dogQ.isEmpty() && !this.catQ.isEmpty()) {
            if (this.dogQ.peek().getCount() < this.catQ.peek().GetCount()) {
                return this.dogQ.poll().getPet();
            } else {
                return this.catQ.poll().getPet();
            }
        }
    }
}
```

```
        }
    } else if (!this.dogQ.isEmpty()) {
        return this.dogQ.poll().getPet();
    } else if (!this.catQ.isEmpty()) {
        return this.catQ.poll().getPet();
    } else {
        throw new RuntimeException("err, queue is empty!");
    }
}

public Dog pollDog() {
    if (!this.isDogQueueEmpty()) {
        return (Dog) this.dogQ.poll().getPet();
    } else {
        throw new RuntimeException("Dog queue is empty!");
    }
}

public Cat pollCat() {
    if (!this.isCatQueueEmpty()) {
        return (Cat) this.catQ.poll().getPet();
    } else {
        throw new RuntimeException("Cat queue is empty!");
    }
}

public boolean isEmpty() {
    return this.dogQ.isEmpty() && this.catQ.isEmpty();
}

public boolean isDogQueueEmpty() {
    return this.dogQ.isEmpty();
}

public boolean isCatQueueEmpty() {
    return this.catQ.isEmpty();
}
}
```

用一个栈实现另一个栈的排序

【题目】

一个栈中元素的类型为整型，现在想将该栈从顶到底按从大到小的顺序排序，只许申请一个栈。除此之外，可以申请新的变量，但不能申请额外的数据结构。如何完成排序？

【难度】

士 ★☆☆☆

【解答】

将要排序的栈记为 `stack`，申请的辅助栈记为 `help`。在 `stack` 上执行 `pop` 操作，弹出的元素记为 `cur`。

- 如果 `cur` 小于或等于 `help` 的栈顶元素，则将 `cur` 直接压入 `help`；
- 如果 `cur` 大于 `help` 的栈顶元素，则将 `help` 的元素逐一弹出，逐一压入 `stack`，直到 `cur` 小于或等于 `help` 的栈顶元素，再将 `cur` 压入 `help`。

一直执行以上操作，直到 `stack` 中的全部元素都压入到 `help`。最后将 `help` 中的所有元素逐一压入 `stack`，即完成排序。

```
public static void sortStackByStack(Stack<Integer> stack) {  
    Stack<Integer> help = new Stack<Integer>();  
    while (!stack.isEmpty()) {  
        int cur = stack.pop();  
        while (!help.isEmpty() && help.peek() > cur) {  
            stack.push(help.pop());  
        }  
        help.push(cur);  
    }  
    while (!help.isEmpty()) {  
        stack.push(help.pop());  
    }  
}
```

用栈来求解汉诺塔问题

【题目】

汉诺塔问题比较经典，这里修改一下游戏规则：现在限制不能从最左侧的塔直接移动到最右侧，也不能从最右侧直接移动到最左侧，而是必须经过中间。求当塔有 N 层的时候，打印最优移动过程和最优移动总步数。

例如，当塔数为两层时，最上层的塔记为 1，最下层的塔记为 2，则打印：

```
Move 1 from left to mid  
Move 1 from mid to right  
Move 2 from left to mid
```

```
Move 1 from right to mid  
Move 1 from mid to left  
Move 2 from mid to right  
Move 1 from left to mid  
Move 1 from mid to right  
It will move 8 steps.
```

注意：关于汉诺塔游戏的更多讨论，将在本书递归与动态规划的章节中继续。

【要求】

用以下两种方法解决。

- 方法一：递归的方法；
- 方法二：非递归的方法，用栈来模拟汉诺塔的三个塔。

【难度】

校 ★★★☆

【解答】

方法一：递归的方法。

首先，如果只剩最上层的塔需要移动，则有如下处理：

1. 如果希望从“左”移到“中”，打印“Move 1 from left to mid”。
2. 如果希望从“中”移到“左”，打印“Move 1 from mid to left”。
3. 如果希望从“中”移到“右”，打印“Move 1 from mid to right”。
4. 如果希望从“右”移到“中”，打印“Move 1 from right to mid”。
5. 如果希望从“左”移到“右”，打印“Move 1 from left to mid”和“Move 1 from mid to right”。
6. 如果希望从“右”移到“左”，打印“Move 1 from right to mid”和“Move 1 from mid to left”。

以上过程就是递归的终止条件，也就是只剩上层塔时的打印过程。

接下来，我们分析剩下多层塔的情况。

如果剩下 N 层塔，从最上到最下依次为 $1 \sim N$ ，则有如下判断：

1. 如果剩下的 N 层塔都在“左”，希望全部移到“中”，则有三个步骤。
 - 1) 将 $1 \sim N-1$ 层塔先全部从“左”移到“右”，明显交给递归过程。
 - 2) 将第 N 层塔从“左”移到“中”。

3) 再将 1~N-1 层塔全部从“右”移到“中”，明显交给递归过程。

2. 如果把剩下的 N 层塔从“中”移到“左”，从“中”移到“右”，从“右”移到“中”，过程与情况 1 同理，一样是分解为三步，在此不再详述。

3. 如果剩下的 N 层塔都在“左”，希望全部移到“右”，则有五个步骤。

1) 将 1~N-1 层塔先全部从“左”移到“右”，明显交给递归过程。

2) 将第 N 层塔从“左”移到“中”。

3) 将 1~N-1 层塔全部从“右”移到“左”，明显交给递归过程。

4) 将第 N 层塔从“中”移到“右”。

5) 最后将 1~N-1 层塔全部从“左”移到“右”，明显交给递归过程。

4. 如果剩下的 N 层塔都在“右”，希望全部移到“左”，过程与情况 3 同理，一样是分解为五步，在此不再详述。

以上递归过程经过逻辑化简之后的代码请参看如下代码中的 `hanoiProblem1` 方法。

```
public int hanoiProblem1(int num, String left, String mid,
                        String right) {
    if (num < 1) {
        return 0;
    }
    return process(num, left, mid, right, left, right);
}

public int process(int num, String left, String mid, String right,
                  String from, String to) {
    if (num == 1) {
        if (from.equals(mid) || to.equals(mid)) {
            System.out.println("Move 1 from " + from + " to " + to);
            return 1;
        } else {
            System.out.println("Move 1 from " + from + " to " + mid);
            System.out.println("Move 1 from " + mid + " to " + to);
            return 2;
        }
    }
    if (from.equals(mid) || to.equals(mid)) {
        String another = (from.equals(left) || to.equals(left)) ? right :
left;

        int part1 = process(num - 1, left, mid, right, from, another);
        int part2 = 1;
        System.out.println("Move " + num + " from " + from + " to " + to);
        int part3 = process(num - 1, left, mid, right, another, to);
        return part1 + part2 + part3;
    } else {
        int part1 = process(num - 1, left, mid, right, from, to);
        int part2 = 1;
```

```
        System.out.println("Move " + num + " from " + from + " to " + mid);
        int part3 = process(num - 1, left, mid, right, to, from);
        int part4 = 1;
        System.out.println("Move " + num + " from " + mid + " to " + to);
        int part5 = process(num - 1, left, mid, right, from, to);
        return part1 + part2 + part3 + part4 + part5;
    }
}
```

方法二：非递归的方法——用栈来模拟整个过程。

修改后的汉诺塔问题不能让任何塔从“左”直接移动到“右”，也不能从“右”直接移动到“左”，而是要经过中间。也就是说，实际动作只有4个：“左”到“中”、“中”到“左”、“中”到“右”、“右”到“中”。

现在我们把左、中、右三个地点抽象成栈，依次记为LS、MS和RS。最初所有的塔都在LS上。那么如上4个动作就可以看作是：某一个栈（from）把栈顶元素弹出，然后压入到另一个栈里（to），作为这一个栈（to）的栈顶。

例如，如果是7层塔，在最初时所有的塔都在LS上，LS从栈顶到栈底就依次是1~7，如果现在发生了“左”到“中”的动作，这个动作对应的操作是LS栈将栈顶元素1弹出，然后1压入到MS栈中，成为MS的栈顶。其他的操作同理。

一个动作能发生的先决条件是不违反小压大的原则。

from栈弹出的元素num如果想压入到to栈中，那么num的值必须小于当前to栈的栈顶。

还有一个原则不是很明显，但也是非常重要的，叫相邻不可逆原则，解释如下：

1. 我们把四个动作依次定义为：L->M、M->L、M->R和R->M。
2. 很明显，L->M和M->L过程互为逆过程，M->R和R->M互为逆过程。

3. 在修改后的汉诺塔游戏中，如果想走出最少步数，那么任何两个相邻的动作都不是互为逆过程的。举个例子：如果上一步的动作是L->M，那么这一步绝不可能是M->L，直观地解释为：你上一步把一个栈顶数从“左”移动到“中”，这一步为什么又要移回去呢？这必然不是取得最小步数的走法。同理，M->R动作和R->M动作也不可能相邻发生。

有了小压大和相邻不可逆原则后，可以推导出两个十分有用的结论——非递归的方法核心结论：

1. 游戏的第一个动作一定是L->M，这是显而易见的。
2. 在走出最少步数过程中的任何时刻，四个动作中只有一个动作不违反小压大和相邻不可逆原则，另外三个动作一定都会违反。

对于结论 2，现在进行简单的证明。

因为游戏的第一个动作已经确定是 $L \rightarrow M$ ，则以后的每一步都会有前一步的动作。

假设前一步的动作是 $L \rightarrow M$ ：

1. 根据小压大原则， $L \rightarrow M$ 的动作不会重复发生。
2. 根据相邻不可逆原则， $M \rightarrow L$ 的动作也不该发生。
3. 根据小压大原则， $M \rightarrow R$ 和 $R \rightarrow M$ 只会有一个达标。

假设前一步的动作是 $M \rightarrow L$ ：

1. 根据小压大原则， $M \rightarrow L$ 的动作不会重复发生。
2. 根据相邻不可逆原则， $L \rightarrow M$ 的动作也不该发生。
3. 根据小压大原则， $M \rightarrow R$ 和 $R \rightarrow M$ 只会有一个达标。

假设前一步的动作是 $M \rightarrow R$ ：

1. 根据小压大原则， $M \rightarrow R$ 的动作不会重复发生。
2. 根据相邻不可逆原则， $R \rightarrow M$ 的动作也不该发生。
3. 根据小压大原则， $L \rightarrow M$ 和 $M \rightarrow L$ 只会有一个达标。

假设前一步的动作是 $R \rightarrow M$ ：

1. 根据小压大原则， $R \rightarrow M$ 的动作不会重复发生。
2. 根据相邻不可逆原则， $M \rightarrow R$ 的动作也不该发生。
3. 根据小压大原则， $L \rightarrow M$ 和 $M \rightarrow L$ 只会有一个达标。

综上所述，每一步只会有一个动作达标。那么只要每走一步都根据这两个原则考查所有的动作就可以，哪个动作达标就走哪个动作，反正每次都只有一个动作满足要求，按顺序走下来即可。

非递归的具体过程请参看如下代码中的 `hanoiProblem2` 方法。

```
public enum Action {
    No, LToM, MToL, MToR, RToM
}

public int hanoiProblem2(int num, String left, String mid, String right) {
    Stack<Integer> lS = new Stack<Integer>();
    Stack<Integer> mS = new Stack<Integer>();
    Stack<Integer> rS = new Stack<Integer>();
    lS.push(Integer.MAX_VALUE);
    mS.push(Integer.MAX_VALUE);
    rS.push(Integer.MAX_VALUE);
    for (int i = num; i > 0; i--) {
        lS.push(i);
    }
}
```

```

Action[] record = { Action.No };
int step = 0;
while (rS.size() != num + 1) {
    step += fStackTotStack(record, Action.MToL, Action.LToM, lS, mS,
                           left, mid);
    step += fStackTotStack(record, Action.LToM, Action.MToL, mS, lS,
                           mid, left);
    step += fStackTotStack(record, Action.RToM, Action.MToR, mS, rS,
                           mid, right);
    step += fStackTotStack(record, Action.MToR, Action.RToM, rS, mS,
                           right, mid);
}
return step;
}

public static int fStackTotStack(Action[] record, Action preNoAct,
                                Action nowAct, Stack<Integer> fStack, Stack<Integer> tStack,
                                String from, String to) {
    if (record[0] != preNoAct && fStack.peek() < tStack.peek()) {
        tStack.push(fStack.pop());
        System.out.println("Move " + tStack.peek() + " from " + from + "
to " + to);
        record[0] = nowAct;
        return 1;
    }
    return 0;
}
}

```

生成窗口最大值数组

【题目】

有一个整型数组 `arr` 和一个大小为 `w` 的窗口从数组的最左边滑到最右边，窗口每次向右边滑一个位置。

例如，数组为[4,3,5,4,3,3,6,7]，窗口大小为 3 时：

[4 3 5]	4 3 3 6 7	窗口中最大值为 5
4 [3 5 4]	3 3 6 7	窗口中最大值为 5
4 3 [5 4 3]	3 6 7	窗口中最大值为 5
4 3 5 [4 3 3]	6 7	窗口中最大值为 4
4 3 5 4 [3 3 6]	7	窗口中最大值为 6
4 3 5 4 3 [3 6 7]		窗口中最大值为 7

如果数组长度为 n ，窗口大小为 w ，则一共产生 $n-w+1$ 个窗口的最大值。

请实现一个函数。

- 输入：整型数组 arr ，窗口大小为 w 。
- 输出：一个长度为 $n-w+1$ 的数组 res ， $res[i]$ 表示每一种窗口状态下的最大值。

以本题为例，结果应该返回 $\{5,5,5,4,6,7\}$ 。

【难度】

尉 ★★☆☆

【解答】

如果数组长度为 N ，窗口大小为 w ，如果做出时间复杂度 $O(N \times w)$ 的解法是不能让面试官满意的，本题要求面试者想出时间复杂度 $O(N)$ 的实现。

本题的关键在于利用双端队列来实现窗口最大值的更新。首先生成双端队列 $qmax$ ， $qmax$ 中存放数组 arr 中的下标。

假设遍历到 $arr[i]$ ， $qmax$ 的放入规则为：

1. 如果 $qmax$ 为空，直接把下标 i 放进 $qmax$ ，放入过程结束。
2. 如果 $qmax$ 不为空，取出当前 $qmax$ 队尾存放的下标，假设为 j 。
 - 1) 如果 $arr[j] > arr[i]$ ，直接把下标 i 放进 $qmax$ 的队尾，放入过程结束。
 - 2) 如果 $arr[j] \leq arr[i]$ ，把 j 从 $qmax$ 中弹出，继续 $qmax$ 的放入规则。

假设遍历到 $arr[i]$ ， $qmax$ 的弹出规则为：

如果 $qmax$ 队头的下标等于 $i-w$ ，说明当前 $qmax$ 队头的下标已过期，弹出当前对头的下标即可。

根据如上的放入和弹出规则， $qmax$ 便成了一个维护窗口为 w 的子数组的最大值更新的结构。下面举例说明题目给出的例子。

1. 开始时 $qmax$ 为空， $qmax = \{\}$
2. 遍历到 $arr[0] = 4$ ，将下标 0 放入 $qmax$ ， $qmax = \{0\}$ 。
3. 遍历到 $arr[1] = 3$ ，当前 $qmax$ 的队尾下标为 0，又有 $arr[0] > arr[1]$ ，所以将下标 1 放入 $qmax$ 的尾部， $qmax = \{0, 1\}$ 。
4. 遍历到 $arr[2] = 5$ ，当前 $qmax$ 的队尾下标为 1，又有 $arr[1] \leq arr[2]$ ，所以将下标 1 从 $qmax$ 的尾部弹出， $qmax$ 变为 $\{0\}$ 。当前 $qmax$ 的队尾下标为 0，又有 $arr[0] \leq arr[2]$ ，所以将下标 0 从 $qmax$ 尾部弹出， $qmax$ 变为 $\{\}$ 。将下标 2 放入 $qmax$ ， $qmax = \{2\}$ 。此时已经遍历到下标 2 的位置，窗口 $arr[0..2]$ 出现，当前 $qmax$ 队头的下标为 2，所以窗口 $arr[0..2]$

的最大值为 $\text{arr}[2]$ （即 5）。

5. 遍历到 $\text{arr}[3]=4$ ，当前 qmax 的队尾下标为 2，又有 $\text{arr}[2]>\text{arr}[3]$ ，所以将下标 3 放入 qmax 尾部， $\text{qmax}=\{2,3\}$ 。窗口 $\text{arr}[1..3]$ 出现，当前 qmax 队头的下标为 2，这个下标还没有过期，所以窗口 $\text{arr}[1..3]$ 的最大值为 $\text{arr}[2]$ （即 5）。

6. 遍历到 $\text{arr}[4]=3$ ，当前 qmax 的队尾下标为 3，又有 $\text{arr}[3]>\text{arr}[4]$ ，所以将下标 4 放入 qmax 尾部， $\text{qmax}=\{2,3,4\}$ 。窗口 $\text{arr}[2..4]$ 出现，当前 qmax 队头的下标为 2，这个下标还没有过期，所以窗口 $\text{arr}[2..4]$ 的最大值为 $\text{arr}[2]$ （即 5）。

7. 遍历到 $\text{arr}[5]=3$ ，当前 qmax 的队尾下标为 4，又有 $\text{arr}[4]\leq\text{arr}[5]$ ，所以将下标 4 从 qmax 的尾部弹出， qmax 变为 $\{2,3\}$ 。当前 qmax 的队尾下标为 3，又有 $\text{arr}[3]>\text{arr}[5]$ ，所以将下标 5 放入 qmax 尾部， $\text{qmax}=\{2,3,5\}$ 。窗口 $\text{arr}[3..5]$ 出现，当前 qmax 队头的下标为 2，这个下标已经过期，所以从 qmax 的头部弹出， qmax 变为 $\{3,5\}$ 。当前 qmax 队头的下标为 3，这个下标没有过期，所以窗口 $\text{arr}[3..5]$ 的最大值为 $\text{arr}[3]$ （即 4）。

8. 遍历到 $\text{arr}[6]=6$ ，当前 qmax 的队尾下标为 5，又有 $\text{arr}[5]\leq\text{arr}[6]$ ，所以将下标 5 从 qmax 的尾部弹出， qmax 变为 $\{3\}$ 。当前 qmax 的队尾下标为 3，又有 $\text{arr}[3]\leq\text{arr}[6]$ ，所以将下标 3 从 qmax 的尾部弹出， qmax 变为 $\{\}$ 。将下标 6 放入 qmax ， $\text{qmax}=\{6\}$ 。窗口 $\text{arr}[4..6]$ 出现，当前 qmax 队头的下标为 6，这个下标没有过期，所以窗口 $\text{arr}[4..6]$ 的最大值为 $\text{arr}[6]$ （即 6）。

9. 遍历到 $\text{arr}[7]=7$ ，当前 qmax 的队尾下标为 6，又有 $\text{arr}[6]\leq\text{arr}[7]$ ，所以将下标 6 从 qmax 的尾部弹出， qmax 变为 $\{\}$ 。将下标 7 放入 qmax ， $\text{qmax}=\{7\}$ 。窗口 $\text{arr}[5..7]$ 出现，当前 qmax 队头的下标为 7，这个下标没有过期，所以窗口 $\text{arr}[5..7]$ 的最大值为 $\text{arr}[7]$ （即 7）。

10. 依次出现的窗口最大值为 $[5,5,5,4,6,7]$ ，在遍历过程中收集起来，最后返回即可。

上述过程中，每个下标值最多进 qmax 一次，出 qmax 一次。所以遍历的过程中进出双端队列的操作是时间复杂度为 $O(N)$ ，整体的时间复杂度也为 $O(N)$ 。具体过程参看如下代码中的 `getMaxWindow` 方法。

```
public int[] getMaxWindow(int[] arr, int w) {
    if (arr == null || w < 1 || arr.length < w) {
        return null;
    }
    LinkedList<Integer> qmax = new LinkedList<Integer>();
    int[] res = new int[arr.length - w + 1];
    int index = 0;
    for (int i = 0; i < arr.length; i++) {
        while (!qmax.isEmpty() && arr[qmax.peekLast()] <= arr[i]) {

```

```
        qmax.pollLast();
    }
    qmax.addLast(i);
    if (qmax.peekFirst() == i - w) {
        qmax.pollFirst();
    }
    if (i >= w - 1) {
        res[index++] = arr[qmax.peekFirst()];
    }
}
return res;
}
```

构造数组的 MaxTree

【题目】

定义二叉树节点如下：

```
public class Node {
    public int value;
    public Node left;
    public Node right;

    public Node(int data) {
        this.value = data;
    }
}
```

一个数组的 MaxTree 定义如下。

- 数组必须没有重复元素。
- MaxTree 是一棵二叉树，数组的每一个值对应一个二叉树节点。
- 包括 MaxTree 树在内且在其中的每一棵子树上，值最大的节点都是树的头。

给定一个没有重复元素的数组 `arr`，写出生成这个数组的 MaxTree 的函数，要求如果数组长度为 N ，则时间复杂度为 $O(N)$ 、额外空间复杂度为 $O(N)$ 。

【难度】

校 ★★★★★

【解答】

下面举例说明如何在满足时间和空间复杂度的要求下生成 MaxTree。

```
arr = {3, 4, 5, 1, 2}
```

3 的左边第一个比 3 大的数：无

3 的右边第一个比 3 大的数：4

4 的左边第一个比 4 大的数：无

4 的右边第一个比 4 大的数：5

5 的左边第一个比 5 大的数：无

5 的右边第一个比 5 大的数：无

1 的左边第一个比 1 大的数：5

1 的右边第一个比 1 大的数：2

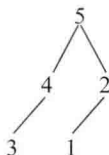
2 的左边第一个比 2 大的数：5

2 的右边第一个比 2 大的数：无

以下列原则来建立这棵树：

- 每一个数的父节点是它左边第一个比它大的数和它右边第一个比它大的数中，较小的那个。
- 如果一个数左边没有比它大的数，右边也没有。也就是说，这个数是整个数组的最大值，那么这个数是 MaxTree 的头节点。

那么 3, 4, 5, 1, 2 的 MaxTree 如下：



为什么通过这个方法能够正确地生成 MaxTree 呢？我们需要给出证明，证明分为如下两步。

1. 通过这个方法，所有的数能生成一棵树，这棵树可能不是二叉树，但肯定是一棵树，而不是多棵树（森林）。

我们知道，在数组中的所有数都不同，而一个较小的数肯定会以一个比自己大的数作为父节点，那么最终所有的数向上找都会找到数组中的最大值，所以它们会有一个共同的头。证明完毕。

2. 通过这个方法，所有的数最多都只有两个孩子。也就是说，这棵树可以用二叉树表示，而不需要多叉树。

要想证明这个问题，只需证明任何一个数在单独一侧，孩子数量都不可能超过 1 个即可。

假设 a 这个数在单独一侧有 2 个孩子，不妨设在右侧。假设这两个孩子一个是 k1，另一个是 k2，即

...a...k1...k2...

因为 a 是 $k1$ 和 $k2$ 的父，所以 $a > k1$, $a > k2$ 。根据题意， $k1$ 和 $k2$ 不相等，所以 $k1$ 和 $k2$ 可以分出大小，先假设 $k1$ 是较小的， $k2$ 是较大的：

那么 $k1$ 可能会以 $k2$ 为父节点，而绝对不会以 a 为父节点，因为根据我们的方法，每一个数的父节点是它左边第一个比它大的数和它右边第一个比它大的数中较小的那个，又有 $a > k2$ 。

再假设 $k2$ 是较小的， $k1$ 是较大的：

那么 $k2$ 可能会以 $k1$ 为父节点，也绝对不会以 a 为父节点，因为根据我们的方法， $k1$ 才可能是 $k2$ 左边第一个遇到的比 $k2$ 大的数，而绝对不会轮到 a 。

总之， $k1$ 和 $k2$ 肯定有一个不是 a 的孩子。

所以，任何一个数的单独一侧，其孩子数量都不可能超过 1 个，最多只会有 1 个。进而我们知道，任何一个数最多会有 2 个孩子，而不会有更多。

证明完毕。

以上证明了该方法是有效的，那么如何尽可能快地找到每一个数左右两边第一个比它大的数呢？利用栈。

找每个数左边第一个比它大的数，从左到右遍历每个数，栈中保持递减序列，新来的数不停地利用 Pop 出栈顶，直到栈顶比新数大或没有数。

以 $[3, 1, 2]$ 为例，首先 3 入栈，接下来 1 比 3 小，无须 pop 出 3，1 入栈，并且确定了 1 往左第一个比它大的数为 3。接下来 2 比 1 大，1 出栈，2 比 3 小，2 入栈，并且确定了 2 往左第一个比它大的数为 3。

用同样的方法可以求得每个数往右第一个比它大的数。

具体请参看如下代码中的 getMaxTree 方法。

```
public Node getMaxTree(int[] arr) {
    Node[] nArr = new Node[arr.length];
    for (int i = 0; i != arr.length; i++) {
        nArr[i] = new Node(arr[i]);
    }
    Stack<Node> stack = new Stack<Node>();
    HashMap<Node, Node> lBigMap = new HashMap<Node, Node>();
    HashMap<Node, Node> rBigMap = new HashMap<Node, Node>();
    for (int i = 0; i != nArr.length; i++) {
        Node curNode = nArr[i];
        while ((!stack.isEmpty()) && stack.peek().value < curNode.value) {
            popStackSetMap(stack, lBigMap);
        }
        stack.push(curNode);
    }
}
```

```
while (!stack.isEmpty()) {
    popStackSetMap(stack, lBigMap);
}
for (int i = nArr.length - 1; i != -1; i--) {
    Node curNode = nArr[i];
    while ((!stack.isEmpty()) && stack.peek().value < curNode.value) {
        popStackSetMap(stack, rBigMap);
    }
    stack.push(curNode);
}
while (!stack.isEmpty()) {
    popStackSetMap(stack, rBigMap);
}
Node head = null;
for (int i = 0; i != nArr.length; i++) {
    Node curNode = nArr[i];
    Node left = lBigMap.get(curNode);
    Node right = rBigMap.get(curNode);
    if (left == null && right == null) {
        head = curNode;
    } else if (left == null) {
        if (right.left == null) {
            right.left = curNode;
        } else {
            right.right = curNode;
        }
    } else if (right == null) {
        if (left.left == null) {
            left.left = curNode;
        } else {
            left.right = curNode;
        }
    } else {
        Node parent = left.value < right.value ? left : right;
        if (parent.left == null) {
            parent.left = curNode;
        } else {
            parent.right = curNode;
        }
    }
}
return head;
}

public void popStackSetMap(Stack<Node> stack, HashMap<Node, Node> map) {
    Node popNode = stack.pop();
    if (stack.isEmpty()) {
        map.put(popNode, null);
    } else {
        map.put(popNode, stack.peek());
    }
}
```

求最大子矩阵的大小

【题目】

给定一个整型矩阵 `map`，其中的值只有 0 和 1 两种，求其中全是 1 的所有矩形区域中，最大的矩形区域为 1 的数量。

例如：

```
1  1  1  0
```

其中，最大的矩形区域有 3 个 1，所以返回 3。

再如：

```
1  0  1  1
1  1  1  1
1  1  1  0
```

其中，最大的矩形区域有 6 个 1，所以返回 6。

【难度】

校 ★★★☆

【解答】

如果矩阵的大小为 $O(N \times M)$ ，本题可以做到时间复杂度为 $O(N \times M)$ 。解法的具体过程为：

1. 矩阵的行数为 N ，以每一行做切割，统计以当前行作为底的情况下，每个位置往上的 1 的数量。使用高度数组 `height` 来表示。

例如：

```
map =  1  0  1  1
       1  1  1  1
       1  1  1  0
```

以第 1 行做切割后，`height={1,0,1,1}`，`height[j]` 表示目前的底上（第 1 行）， j 位置往上（包括 j 位置）有多少连续的 1。

以第 2 行做切割后，`height={2,1,2,2}`，注意到从第一行到第二行，`height` 数组的更新是十分方便的，即 `height[j] = map[i][j]==0 ? 0 : height[j]+1`。

以第 3 行做切割后，`height={3,2,3,0}`。

2. 对于每一次切割，都利用更新后的 `height` 数组来求出以每一行为底的情况下，最大

的矩形是什么。那么这么多次切割中，最大的那个矩形就是我们想要的。

整个过程就是如下代码中的 `maxRecSize` 方法。步骤 2 的实现是如下代码中的 `maxRecFromBottom` 方法。

下面重点介绍一下步骤 2 如何快速地实现，这也是这道题最重要的部分，如果 `height` 数组的长度为 M ，那么求解步骤 2 的过程可以做到时间复杂度为 $O(M)$ 。

对于 `height` 数组，读者可以理解为一个直方图，比如 $\{3, 2, 3, 0\}$ ，其实就是如图 1-6 所示的直方图。

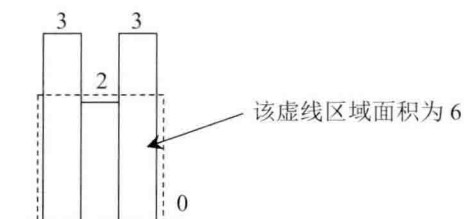


图 1-6

也就是说，步骤 2 的实质是在一个大的直方图中求最大矩形的面积。如果我们能够求出以每一根柱子扩展出去的最大矩形，那么其中最大的矩形就是我们想找的。比如：

- 第 1 根高度为 3 的柱子向左无法扩展，它的右边是 2，比 3 小，所以向右也无法扩展，则以第 1 根柱子为高度的矩形面积就是 $3 \times 1 = 3$ ；
- 第 2 根高度为 2 的柱子向左可以扩 1 个距离，因为它的左边是 3，比 2 大；右边的柱子也是 3，所以向右也可以扩 1 个距离，则以第 2 根柱子为高度的矩形面积就是 $2 \times 3 = 6$ ；
- 第 3 根高度为 3 的柱子向左没法扩展，向右也没法扩展，则以第 3 根柱子为高度的矩形面积就是 $3 \times 1 = 3$ ；
- 第 4 根高度为 0 的柱子向左没法扩展，向右也没法扩展，则以第 4 根柱子为高度的矩形面积就是 $0 \times 1 = 0$ ；

所以，当前直方图中最大的矩形面积就是 6，也就是图 1-6 中虚线框住的部分。

考查每一根柱子最大能扩多大，这个行为的实质就是找到柱子左边刚比它小的柱子位置在哪里，以及右边刚比它小的柱子位置在哪里。这个过程怎么计算最快呢？用栈。

为了方便表述，我们以 `height = {3, 4, 5, 4, 3, 6}` 为例说明如何根据 `height` 数组求其中的最大矩形。具体过程如下：

1. 生成一个栈，记为 `stack`，从左到右遍历 `height` 数组，每遍历一个位置，都会把位

置压进 stack 中。

2. 遍历到 height 的 0 位置，height[0]=3，此时 stack 为空，直接将位置 0 压入栈中，此时 stack 从栈顶到栈底为{0}。

3. 遍历到 height 的 1 位置，height[1]=4，此时 stack 的栈顶为位置 0，值为 height[0]=3，又有 height[1]>height[0]，那么将位置 1 直接压入 stack。这一步体现了遍历过程中的一个关键逻辑：只有当前 i 位置的值 height[i] 大于当前栈顶位置所代表的值(height[stack.peek()])，则 i 位置才可以压入 stack。

所以可以知道，stack 中从栈顶到栈底的位置所代表的值是依次递减，并且无重复值，此时 stack 从栈顶到栈底为{1,0}。

4. 遍历到 height 的 2 位置，height[2]=5，与步骤 3 的情况完全一样，所以直接将位置 2 压入 stack，此时 stack 从栈顶到栈底为{2,1,0}。

5. 遍历到 height 的 3 位置，height[3]=4，此时 stack 的栈顶为位置 2，值为 height[2]=5，又有 height[3]<height[2]。此时又出现了一个遍历过程中的关键逻辑，即如果当前 i 位置的值 height[i] 小于或等于当前栈顶位置所代表的值(height[stack.peek()])，则把栈中存的位置不断弹出，直到某一个栈顶所代表的值小于 height[i]，再把位置 i 压入，并在这期间做如下处理：

1) 假设当前弹出的栈顶位置记为位置 j ，弹出栈顶之后，新的栈顶记为 k 。然后我们开始考虑位置 j 的柱子向右和向左最远能扩到哪里。

2) 对位置 j 的柱子来说，向右最远能扩到哪里呢？

如果 height[j]>height[i]，那么 $i-1$ 位置就是向右能扩到的最远位置。因为 j 之所以被弹出，就是因为遇到了第一个比位置 j 值小的位置。

如果 height[j]==height[i]，那么 $i-1$ 位置不一定是向右能扩到的最远位置，只是起码能扩到的位置。那怎么办呢？

可以肯定的是，在这种情况下， i 位置的柱子向左必然也可以扩到 j 位置。也就是说， j 位置的柱子扩出来的最大矩形和 i 位置的柱子扩出来的最大矩形是同一个。

所以，此时可以不再计算 j 位置的柱子能扩出来的最大矩形，因为位置 i 肯定要压入到栈中，那就等位置 i 弹出的时候再说。

3) 对位置 j 的柱子来说，向左最远能扩到哪里呢？

肯定是 $k+1$ 位置。首先，height[k+1..j-1] 之间不可能有小于或等于 height[k] 的值，否则 k 位置早从栈里弹出了。

然后因为在栈里 k 位置和 j 位置原本是相邻的，并且从栈顶到栈底的位置所代表的值是依次递减并且无重复值，所以在 height[k+1..j-1] 之间不可能有大于或等于 height[k]，同时

又小于或等于 $\text{height}[j]$ 的，因为如果有这样的值， k 和 j 在栈中就不可能相邻。

所以， $\text{height}[k+1..j-1]$ 之间的值必然是既大于 $\text{height}[k]$ ，又大于 $\text{height}[j]$ 的，所以 j 位置的柱子向左最远可以扩到 $k+1$ 位置。

4) 综上所述， j 位置的柱子能扩出来的最大矩形为 $(i-k-1)*\text{height}[j]$ 。

以例子来说明：

① $i=3$ ， $\text{height}[3]=4$ ，此时 stack 的栈顶为位置 2，值为 $\text{height}[2]=5$ ，故 $\text{height}[3]<=\text{height}[2]$ ，所以位置 2 被弹出 ($j=2$)，当前栈顶变为 1 ($k=1$)。位置 2 的柱子扩出来的最大矩形面积为 $(3-1-1)*5=5$ 。

② $i=3$ ， $\text{height}[3]=4$ ，此时 stack 的栈顶为位置 1，值为 $\text{height}[1]=4$ ，故 $\text{height}[3]<=\text{height}[1]$ ，所以位置 1 被弹出 ($j=1$)，当前栈顶变为 0 ($k=0$)。位置 1 的柱子扩出来的最大矩形面积为 $(3-0-1)*4=8$ ，这个值实际上是不对的 (偏小)，但在位置 3 被弹出的时候是能够重新正确计算得到的。

③ $i=3$ ， $\text{height}[3]=4$ ，此时 stack 的栈顶为位置 0，值为 $\text{height}[0]=3$ ，这时 $\text{height}[3]<=\text{height}[2]$ ，所以位置 0 不弹出。

④ 将位置 3 压入 stack ， stack 从栈顶到栈底为 $\{3,0\}$ 。

6. 遍历到 height 的 4 位置， $\text{height}[4]=3$ 。与步骤 5 的情况类似，以下是弹出过程：

1) $i=4$ ， $\text{height}[4]=3$ ，此时 stack 的栈顶为位置 3，值为 $\text{height}[3]=4$ ，故 $\text{height}[4]<=\text{height}[3]$ ，所以位置 3 被弹出 ($j=3$)，当前栈顶变为 0 ($k=0$)。位置 3 的柱子扩出来的最大矩形面积为 $(4-0-1)*4=12$ 。这个最大面积也是位置 1 的柱子扩出来的最大矩形面积，在位置 1 被弹出时，这个矩形其实没有找到，但在位置 3 这里找到了。

2) $i=4$ ， $\text{height}[4]=3$ ，此时 stack 的栈顶为位置 0，值为 $\text{height}[0]=3$ ，故 $\text{height}[4]<=\text{height}[0]$ ，所以位置 0 被弹出 ($j=0$)，当前没有了栈顶元素，此时可以认为 $k=-1$ 。位置 0 的柱子扩出来的最大矩形面积为 $(4-(-1)-1)*3=12$ 这个值实际上是不对的 (偏小)，但在位置 4 被弹出时是能够重新正确计算得到的。

3) 栈已经为空，所以将位置 4 压入 stack ，此时从栈顶到栈底为 $\{4\}$ 。

7. 遍历到 height 的 5 位置， $\text{height}[5]=6$ ，情况和步骤 3 类似，直接压入位置 5，此时从栈顶到栈底为 $\{5,4\}$ 。

8. 遍历结束后， stack 中仍有位置没有经历扩的过程，从栈顶到栈底为 $\{5,4\}$ 。此时因为 height 数组再往右不能扩出去，所以认为 $i=\text{height.length}=6$ 且越界之后的值极小，然后开始弹出留在栈中的位置：

1) $i=6$ ， $\text{height}[6]$ 极小，此时 stack 的栈顶为位置 5，值为 $\text{height}[5]=6$ ，故

$height[6] \leq height[5]$ ，所以位置 6 被弹出 ($j=6$)，当前栈顶变为 4 ($k=4$)。位置 5 的柱子扩出来的最大矩形面积为 $(6-4-1)*6=6$ 。

2) $i=6$ ， $height[6]$ 极小，此时 $stack$ 的栈顶为位置 4，值为 $height[4]=3$ ，故 $height[6] \leq height[4]$ ，所以位置 4 被弹出 ($j=4$)，栈空了，此时可以认为 $k=-1$ 。位置 4 的柱子扩出来的最大矩形面积为 $(6-(-1)-1)*3=18$ 。这个最大面积也是位置 0 的柱子扩出来的最大矩形面积，在位置 0 被弹出的时候，这个矩形其实没有找到，但在位置 4 这里找到了。

3) 栈已经空了，过程结束。

9. 整个过程结束，所有找到的最大矩形面积中 18 是最大的，所以返回 18。

研究以上 9 个步骤时我们发现，任何一个位置都仅仅进出栈 1 次，所以时间复杂度为 $O(M)$ 。既然每做一次切割处理的时间复杂度为 $O(M)$ ，一共做 N 次，则总的时间复杂度为 $O(N \times M)$ 。

全部过程参看如下代码中的 `maxRecSize` 方法。9 个步骤的详细过程参看代码中的 `maxRecFromBottom` 方法。

```
public int maxRecSize(int[][] map) {
    if (map == null || map.length == 0 || map[0].length == 0) {
        return 0;
    }
    int maxArea = 0;
    int[] height = new int[map[0].length];
    for (int i = 0; i < map.length; i++) {
        for (int j = 0; j < map[0].length; j++) {
            height[j] = map[i][j] == 0 ? 0 : height[j] + 1;
        }
        maxArea = Math.max(maxRecFromBottom(height), maxArea);
    }
    return maxArea;
}

public int maxRecFromBottom(int[] height) {
    if (height == null || height.length == 0) {
        return 0;
    }
    int maxArea = 0;
    Stack<Integer> stack = new Stack<Integer>();
    for (int i = 0; i < height.length; i++) {
        while (!stack.isEmpty() && height[i] <= height[stack.peek()]) {
            int j = stack.pop();
            int k = stack.isEmpty() ? -1 : stack.peek();
            int curArea = (i - k - 1) * height[j];
            maxArea = Math.max(maxArea, curArea);
        }
    }
}
```

```
        stack.push(i);
    }
    while (!stack.isEmpty()) {
        int j = stack.pop();
        int k = stack.isEmpty() ? -1 : stack.peek();
        int curArea = (height.length - k - 1) * height[j];
        maxArea = Math.max(maxArea, curArea);
    }
    return maxArea;
}
```

最大值减去最小值小于或等于 num 的子数组数量

【题目】

给定数组 arr 和整数 num，共返回有多少个子数组满足如下情况：

$\max(\text{arr}[i..j]) - \min(\text{arr}[i..j]) \leq \text{num}$

$\max(\text{arr}[i..j])$ 表示子数组 arr[i..j]中的最大值， $\min(\text{arr}[i..j])$ 表示子数组 arr[i..j]中的最小值。

【要求】

如果数组长度为 N，请实现时间复杂度为 $O(N)$ 的解法。

【难度】

校 ★★★☆

【解答】

首先介绍普通的解法，找到 arr 的所有子数组，一共有 $O(N^2)$ 个，然后对每一个子数组做遍历找到其中的最小值和最大值，这个过程时间复杂度为 $O(N)$ ，然后看看这个子数组是否满足条件。统计所有满足的子数组数量即可。普通解法容易实现，但是时间复杂度为 $O(N^3)$ ，本书不再详述。最优解可以做到时间复杂度 $O(N)$ ，额外空间复杂度 $O(N)$ ，在阅读下面的分析过程之前，请读者先阅读本章“生成窗口最大值数组”问题，本题所使用到的双端队列结构与解决“生成窗口最大值数组”问题中的双端队列结构含义基本一致。

生成两个双端队列 qmax 和 qmin。当子数组为 arr[i..j] 时，qmax 维护了窗口子数组 arr[i..j] 的最大值更新的结构，qmin 维护了窗口子数组 arr[i..j] 的最小值更新的结构。当子数组 arr[i..j] 向右扩一个位置变成 arr[i..j+1] 时，qmax 和 qmin 结构可以在 $O(1)$ 的时间内更新，并且可以

在 $O(1)$ 的时间内得到 $\text{arr}[i..j+1]$ 的最大值和最小值。当子数组 $\text{arr}[i..j]$ 向左缩一个位置变成 $\text{arr}[i+1..j]$ 时， qmax 和 qmin 结构依然可以在 $O(1)$ 的时间内更新，并且在 $O(1)$ 的时间内得到 $\text{arr}[i+1..j]$ 的最大值和最小值。

通过分析题目满足的条件，可以得到如下两个结论：

- 如果子数组 $\text{arr}[i..j]$ 满足条件，即 $\max(\text{arr}[i..j]) - \min(\text{arr}[i..j]) \leq \text{num}$ ，那么 $\text{arr}[i..j]$ 中的每一个子数组，即 $\text{arr}[k..l] (i \leq k \leq l \leq j)$ 都满足条件。我们以子数组 $\text{arr}[i..j-1]$ 为例说明， $\text{arr}[i..j-1]$ 最大值只可能小于或等于 $\text{arr}[i..j]$ 的最大值， $\text{arr}[i..j-1]$ 最小值只可能大于或等于 $\text{arr}[i..j]$ 的最小值，所以 $\text{arr}[i..j-1]$ 必然满足条件。同理， $\text{arr}[i..j]$ 中的每一个子数组都满足条件。
- 如果子数组 $\text{arr}[i..j]$ 不满足条件，那么所有包含 $\text{arr}[i..j]$ 的子数组，即 $\text{arr}[k..l] (k \leq i \leq j \leq l)$ 都不满足条件。证明过程同第一个结论。

根据双端队列 qmax 和 qmin 的结构性质，以及如上两个结论，设计整个过程如下：

1. 生成两个双端队列 qmax 和 qmin ，含义如上文所说。生成两个整型变量 i 和 j ，表示子数组的范围，即 $\text{arr}[i..j]$ 。生成整型变量 res ，表示所有满足条件的子数组数量。

2. 令 j 不断向右移动 ($j++$)，表示 $\text{arr}[i..j]$ 一直向右扩大，并不断更新 qmax 和 qmin 结构，保证 qmax 和 qmin 始终维持动态窗口最大值和最小值的更新结构。一旦出现 $\text{arr}[i..j]$ 不满足条件的情况， j 向右扩的过程停止，此时 $\text{arr}[i..j-1]$ 、 $\text{arr}[i..j-2]$ 、 $\text{arr}[i..j-3]$ 、...、 $\text{arr}[i..i]$ 一定都是满足条件的。也就是说，所有必须以 $\text{arr}[i]$ 作为第一个元素的子数组，满足条件的数量为 $j-i$ 个。于是令 $\text{res} += j-i$ 。

3. 当进行完步骤 2，令 i 向右移动一个位置，并对 qmax 和 qmin 做出相应的更新， qmax 和 qmin 从原来的 $\text{arr}[i..j]$ 窗口变成 $\text{arr}[i+1..j]$ 窗口的最大值和最小值的更新结构。然后重复步骤 2，也就是求所有必须以 $\text{arr}[i+1]$ 作为第一个元素的子数组中，满足条件的数量有多少个。

4. 根据步骤 2 和步骤 3，依次求出以 $\text{arr}[0]$ 、 $\text{arr}[1]$ 、...、 $\text{arr}[N-1]$ 作为第一个元素的子数组中满足条件的数量分别有多少个，累加起来的数量就是最终的结果。

上述过程中，所有的下标值最多进 qmax 和 qmin 一次，出 qmax 和 qmin 一次。 i 和 j 的值也不断增加，并且从来不减小。所以整个过程的时间复杂度为 $O(N)$ 。

最优解全部实现请参看如下代码中的 `getNum` 方法。

```
public int getNum(int[] arr, int num) {  
    if (arr == null || arr.length == 0) {  
        return 0;  
    }  
}
```

```
LinkedList<Integer> qmin = new LinkedList<Integer>();
LinkedList<Integer> qmax = new LinkedList<Integer>();
int i = 0;
int j = 0;
int res = 0;
while (i < arr.length) {
    while (j < arr.length) {
        while (!qmin.isEmpty() && arr[qmin.peekLast()] >= arr[j]) {
            qmin.pollLast();
        }
        qmin.addLast(j);
        while (!qmax.isEmpty() && arr[qmax.peekLast()] <= arr[j]) {
            qmax.pollLast();
        }
        qmax.addLast(j);
        if (arr[qmax.getFirst()] - arr[qmin.getFirst()] > num) {
            break;
        }
        j++;
    }
    if (qmin.peekFirst() == i) {
        qmin.pollFirst();
    }
    if (qmax.peekFirst() == i) {
        qmax.pollFirst();
    }
    res += j - i;
    i++;
}
return res;
}
```

第 2 章

链表问题

打印两个有序链表的公共部分

【题目】

给定两个有序链表的头指针 head1 和 head2，打印两个链表的公共部分。

【难度】

士 ★☆☆☆

【解答】

本题难度很低，因为是有序链表，所以从两个链表的头开始进行如下判断：

- 如果 head1 的值小于 head2，则 head1 往下移动。
- 如果 head2 的值小于 head1，则 head2 往下移动。
- 如果 head1 的值与 head2 的值相等，则打印这个值，然后 head1 与 head2 都往下移动。
- head1 或 head2 有任何一个移动到 null，整个过程停止。

具体过程参看如下代码中的 printCommonPart 方法。

```
public class Node {  
    public int value;  
    public Node next;  
    public Node(int data) {  
        this.value = data;  
    }  
}
```

```
}  
  
public void printCommonPart(Node head1, Node head2) {  
    System.out.print("Common Part: ");  
    while (head1 != null && head2 != null) {  
        if (head1.value < head2.value) {  
            head1 = head1.next;  
        } else if (head1.value > head2.value) {  
            head2 = head2.next;  
        } else {  
            System.out.print(head1.value + " ");  
            head1 = head1.next;  
            head2 = head2.next;  
        }  
    }  
    System.out.println();  
}
```

在单链表和双链表中删除倒数第 K 个节点

【题目】

分别实现两个函数，一个可以删除单链表中倒数第 K 个节点，另一个可以删除双链表中倒数第 K 个节点。

【要求】

如果链表长度为 N ，时间复杂度达到 $O(N)$ ，额外空间复杂度达到 $O(1)$ 。

【难度】

士 ★☆☆☆

【解答】

本题较为简单，实现方式也是多种多样的，本书提供一种方法供读者参考。

先来看看单链表如何调整。如果链表为空或者 K 值小于 1，这种情况下，参数是无效的，直接返回即可。除此之外，让链表从头开始走到尾，每移动一步，就让 K 的值减 1。

链表：1->2->3， $K=4$ ，链表根本不存在倒数第 4 个节点。

走到的节点：1->2->3

K 变化为：3 2 1

链表：1->2->3， $K=3$ ，链表倒数第 3 个节点是 1 节点。

走到的节点：1->2->3

K 变化为：2 1 0

链表：1->2->3， $K=2$ ，链表倒数第 2 个节点是 2 节点。

走到的节点：1->2->3

K 变化为：1 0 -1

由以上三种情况可知，让链表从头开始走到尾，每移动一步，就让 K 值减 1，当链表走到结尾时，如果 K 值大于 0，说明不用调整链表，因为链表根本没有倒数第 K 个节点，此时将原链表直接返回即可；如果 K 值等于 0，说明链表倒数第 K 个节点就是头节点，此时直接返回 `head.next`，也就是原链表的第二个节点，让第二个节点作为链表的头返回即可，相当于删除头节点；接下来，说明一下如果 K 值小于 0，该如何处理。

先明确一点，如果要删除链表的头节点之后的某个节点，实际上需要找到要删除节点的前一个节点，比如：1->2->3，如果想删除节点 2，则需要找到节点 1，然后把节点 1 连到节点 3 上（1->3），以此来达到删除节点 2 的目的。

如果 K 值小于 0，如何找到要删除节点的前一个节点呢？方法如下：

1. 重新从头节点开始走，每移动一步，就让 K 的值加 1。
2. 当 K 等于 0 时，移动停止，移动到的节点就是要删除节点的前一个节点。

这样做是非常好理解的，因为如果链表长度为 N ，要删除倒数第 K 个节点，很明显，倒数第 K 个节点的前一个节点就是第 $N-K$ 个节点。在第一次遍历后， K 的值变为 $K-N$ 。第二次遍历时， K 的值不断加 1，加到 0 就停止遍历，第二次遍历当然会停到第 $N-K$ 个节点的位置。

具体过程请参看如下代码中的 `removeLastKthNode` 方法。

```
public class Node {
    public int value;
    public Node next;

    public Node(int data) {
        this.value = data;
    }
}

public Node removeLastKthNode(Node head, int lastKth) {
    if (head == null || lastKth < 1) {
        return head;
    }
    Node cur = head;
```

```
while (cur != null) {
    lastKth--;
    cur = cur.next;
}
if (lastKth == 0) {
    head = head.next;
}
if (lastKth < 0) {
    cur = head;
    while (++lastKth != 0) {
        cur = cur.next;
    }
    cur.next = cur.next.next;
}
return head;
}
```

对于双链表的调整，几乎与单链表的处理方式一样，注意 last 指针的重连即可。具体过程请参看如下代码中的 removeLastKthNode 方法。

```
public class DoubleNode {
    public int value;
    public DoubleNode last;
    public DoubleNode next;

    public DoubleNode(int data) {
        this.value = data;
    }
}

public DoubleNode removeLastKthNode(DoubleNode head, int lastKth) {
    if (head == null || lastKth < 1) {
        return head;
    }
    DoubleNode cur = head;
    while (cur != null) {
        lastKth--;
        cur = cur.next;
    }
    if (lastKth == 0) {
        head = head.next;
        head.last = null;
    }
    if (lastKth < 0) {
        cur = head;
        while (++lastKth != 0) {
            cur = cur.next;
        }
        DoubleNode newNext = cur.next.next;
        cur.next = newNext;
    }
}
```

```
        if (newNext != null) {
            newNext.last = cur;
        }
    }
    return head;
}
```

删除链表的中间节点和 a/b 处的节点

【题目】

给定链表的头节点 head，实现删除链表的中间节点的函数。

例如：

不删除任何节点；

1->2，删除节点 1；

1->2->3，删除节点 2；

1->2->3->4，删除节点 2；

1->2->3->4->5，删除节点 3；

进阶：

给定链表的头节点 head、整数 a 和 b，实现删除位于 a/b 处节点的函数。

例如：

链表：1->2->3->4->5，假设 a/b 的值为 r。

如果 r 等于 0，不删除任何节点；

如果 r 在区间(0, 1/5]上，删除节点 1；

如果 r 在区间(1/5, 2/5]上，删除节点 2；

如果 r 在区间(2/5, 3/5]上，删除节点 3；

如果 r 在区间(3/5, 4/5]上，删除节点 4；

如果 r 在区间(4/5, 1]上，删除节点 5；

如果 r 大于 1，不删除任何节点。

【难度】

士 ★☆☆☆

【解答】

先来分析原问题，如果链表为空或者长度为 1，不需要调整，则直接返回；如果链表的长度为 2，将头节点删除即可；当链表长度到达 3，应该删除第 2 个节点；当链表长度为 4，应该删除第 2 个节点；当链表长度为 5，应该删除第 3 个节点……也就是链表长度每增加 2(3,5,7...)，要删除的节点就后移一个节点。删除节点的问题在之前的题目中我们已经讨论过，如果要删除一个节点，则需要找到待删除节点的前一个节点。

具体过程请参看如下代码中的 removeMidNode 方法。

```
public class Node {
    public int value;
    public Node next;

    public Node(int data) {
        this.value = data;
    }
}

public Node removeMidNode(Node head) {
    if (head == null || head.next == null) {
        return head;
    }
    if (head.next.next == null) {
        return head.next;
    }
    Node pre = head;
    Node cur = head.next.next;
    while (cur.next != null && cur.next.next != null) {
        pre = pre.next;
        cur = cur.next.next;
    }
    pre.next = pre.next.next;
    return head;
}
```

接下来讨论进阶问题，首先需要解决的问题是，如何根据链表的长度 n ，以及 a 与 b 的值决定该删除的节点是哪一个节点呢？根据如下方法：

先计算 $\text{double } r = ((\text{double}) (a * n)) / ((\text{double}) b)$ 的值，然后 r 向上取整之后的整数值代表该删除的节点是第几个节点。

下面举几个例子来验证一下：

如果链表长度为 7， $a=5$ ， $b=7$ 。

$r = (7*5)/7 = 5.0$ ，向上取整后为 5，所以应该删除第 5 个节点。

如果链表长度为 7， $a=5$ ， $b=6$ 。

$r = (7*5)/6 = 5.8333\dots$ ，向上取整后为 6，所以应该删除第 6 个节点。

如果链表长度为 7， $a=1$ ， $b=6$ 。

$r = (7*1)/6 = 1.1666\dots$ ，向上取整后为 2，所以应该删除第 2 个节点。

知道该删除第几个节点之后，接下来找到需要删除节点的前一个节点即可。具体过程请参看如下代码中的 `removeByRatio` 方法。

```
public Node removeByRatio(Node head, int a, int b) {
    if (a < 1 || a > b) {
        return head;
    }
    int n = 0;
    Node cur = head;
    while (cur != null) {
        n++;
        cur = cur.next;
    }
    n = (int) Math.ceil(((double) (a * n)) / (double) b);
    if (n == 1) {
        head = head.next;
    }
    if (n > 1) {
        cur = head;
        while (--n != 1) {
            cur = cur.next;
        }
        cur.next = cur.next.next;
    }
    return head;
}
```

反转单向和双向链表

【题目】

分别实现反转单向链表和反转双向链表的函数。

【要求】

如果链表长度为 N ，时间复杂度要求为 $O(N)$ ，额外空间复杂度要求为 $O(1)$ 。

【难度】

士 ★☆☆☆

【解答】

本题比较简单，读者做到代码一次成型，运行不出错即可。

反转单向链表的函数如下，函数返回反转之后链表新的头节点：

```
public class Node {
    public int value;
    public Node next;
    public Node(int data) {
        this.value = data;
    }
}

public Node reverseList(Node head) {
    Node pre = null;
    Node next = null;
    while (head != null) {
        next = head.next;
        head.next = pre;
        pre = head;
        head = next;
    }
    return pre;
}
```

反转双向链表的函数如下，函数返回反转之后链表新的头节点：

```
public DoubleNode {
    public int value;
    public DoubleNode last;
    public DoubleNode next;
    public DoubleNode(int data) {
        this.value = data;
    }
}

public DoubleNode reverseList(DoubleNode head) {
    DoubleNode pre = null;
    DoubleNode next = null;
    while (head != null) {
        next = head.next;
        head.next = pre;
        head.last = next;
        pre = head;
        head = next;
    }
    return pre;
}
```