

剑指

Offer

名企面试官精讲
典型编程题

何海涛 ◎著



```
public class Solution {
    public int Power(int base, int exponent) {
        if(exponent < 0) {
            return 1 / Power(base, -exponent);
        }
        if(exponent == 0) {
            return 1;
        }
        if(exponent == 1) {
            return base;
        }
        double result = 1.0;
        for(int i = 0; i < exponent; i++) {
            result *= base;
        }
        return result;
    }

    public double PowerWithUnsignedExponent(double base, unsigned int exponent) {
        double result = 1.0;
        for(unsigned int i = 0; i < exponent; i++) {
            result *= base;
        }
        return result;
    }

    bool equal(double num1, double num2) {
        if((num1 - num2) > 0.000001)
            return true;
        else
            return false;
    }

    double PowerWithSignedExponent(double base, int exponent) {
        if(exponent == 0)
            return 1;
        if(exponent == 1)
            return base;

        double result = PowerWithUnsignedExponent(base, exponent);
        result *= result;
        if(exponent % 2 == 1)
            result *= base;
        return result;
    }
}
```

剑指Offer

名企面试官精讲典型编程题

(第2版)

何海涛 著

电子工业出版社
Publishing House of Electronics Industry
北京•BEIJING

内容简介

本书前身曾在全球范围内发行过英文版。这一版本在前版基础上进一步精选和增补试题，结合作者近年来在美国从事开发工作的实践经验及思考积累，使全书更加融会贯通、广泛适用。本书剖析了 80 道典型的编程题，系统整理基础知识、代码质量、解题思路、优化效率和综合能力这 5 个面试要点。全书共分 7 章，主要包括面试的流程，讨论面试每一环节需要注意的问题；面试需要的基础知识，从编程语言、数据结构及算法三方面总结程序员面试知识点；高质量的代码，讨论影响代码质量的 3 个要素（规范性、完整性和鲁棒性），强调高质量代码除完成基本功能外，还能考虑特殊情况并对非法输入进行合理处理；解决面试题的思路，总结编程面试中解决难题的有效思考模式，如在面试中遇到复杂难题，应聘者可利用画图、举例和分解这 3 种方法将其化繁为简，先形成清晰思路，再动手编程；优化时间和空间效率，读者将学会优化时间效率及用空间换时间的常用算法，从而在面试中找到最优解；面试中的各项能力，总结应聘者如何充分表现学习和沟通能力，并通过具体面试题讨论如何培养知识迁移、抽象建模和发散思维能力；两个面试案例，总结哪些面试举动是不良行为，而哪些表现又是面试官所期待的行为。

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

剑指 Offer：名企面试官精讲典型编程题 / 何海涛著. —2 版. —北京：电子工业出版社，2017.5

ISBN 978-7-121-31092-8

I. ①剑… II. ①何… III. ①程序设计—资格考试—习题集 IV. ①TP311.1-44

中国版本图书馆 CIP 数据核字 (2017) 第 053903 号

策划编辑：张春雨

责任编辑：徐津平

特约编辑：赵树刚

印 刷：三河市良远印务有限公司

装 订：三河市良远印务有限公司

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：22 字数：423 千字

版 次：2011 年 11 月第 1 版

2017 年 5 月第 2 版

印 次：2017 年 5 月第 1 次印刷

定 价：65.00 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, 88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819, faq@phei.com.cn。

第 2 版序言

时间总是在不经意间流逝，我们也在人生的旅途上不断前行，转眼间我在微软的美国总部工作近两年了。生活总给我们带来新的挑战，同时也有新的惊喜。这两年在陌生的国度里用着不太流利的英语和各种肤色的人交流，体验着世界的多元化。这两年加班、熬过夜，也为进展不顺的项目焦头烂额过。在微软 Office 新产品发布那天，也自豪过，忍不住在朋友圈里和大家分享自己的喜悦和兴奋。2015 年 4 月，我和素云又一次迎来了一个小生命。之后的日子虽然辛苦，但每当看着呼呼、阳阳两兄弟天真灿烂的笑容时，我的心里只有无限的幸福。

西雅图是一个 IT 氛围很浓的地方，这里是微软和亚马逊的总部所在地，Google、Facebook 等很多知名公司都在这里有研发中心。一群程序员聚在一起，总会谈到谁去这家公司面试了，谁拿到了那家公司的 Offer。这让我有机会从多个角度去理解编程面试，也更加深入地思考怎样刷题才会更加有效。我的这些理解、思考都融入《剑指 Offer——名企面试官精讲典型编程题》这本书的第 2 版里。

这次再版在第 1 版的基础上增加了新的面试题，涵盖了新的知识点。第 2 版新增了 2.4.3 节和 2.4.4 节，分别讨论回溯法、动态规划和贪婪算法。正则表达式是编程面试时经常出现的内容，本次新增了两个正则表达式匹配的问题（详见面试题 19 和面试题 20）。

这次新增的内容有些是原有内容的延伸。比如原书的面试题 35 要求找出字符串中第一个只出现一次的字符 [在第 2 版中为面试题 50（题目一）]。这次新增的面试题 50（题目二）把要求改为从一个字符流中找出第一个只出现一次的字符。再比如，在原书的面试题 23 [在第 2 版中为面试题 32（题

目一)] 中讨论了如何把二叉树按层打印到一行里，这次新增了两个按层打印二叉树的面试题：面试题 32（题目二）要求把二叉树的每一层单独打印到一行；面试题 32（题目三）要求按之字形顺序打印二叉树。

计算机领域的知识更新很快，编程面试题也需要推陈出新。本书的参考代码以 C++为主，这次再版根据 C++新的标准在内容上进行了一些调整。例如，原书的面试题 48 要求用 C++实现不能继承的类。由于在 C++11 中引入了关键字 final，那么用 C++实现不能继承的类已经变得非常容易。因此，这次再版时用新的面试题替代了它。

自本书出版以来，收到了很多读者的反馈，让我受益匪浅。例如，面试题 20 “表示数值的字符串”根据 GitHub 用户 cooljacket 的意见做了修改。在此对所有提出反馈、建议的读者表示衷心的感谢。

本书所有源代码（包含单元测试用例）都分享在 GitHub 上，欢迎读者对本书及 GitHub 上的代码提出意见。如果发现代码中存在问题，或者发现还有更好的解法，则欢迎读者递交代码。本书所有源代码均以 BSD 许可证开源，欢迎大家共同参与，一起提高代码的质量。

通过读者的 E-mail，我很高兴地得知《剑指 Offer——名企面试官精讲典型编程题》一书陪伴很多读者找到了心仪的工作，拿到了满意的 Offer。实际上，这本书不仅仅是一本关于求职面试的工具书，同时还是一本关于编程的技术书。书中用大量的篇幅讨论数据结构和算法，讨论如何才能写出高质量的代码。这些技能在面试的时候有用，在平时的开发工作中同样有用。希望本书能陪伴更多的读者在职场中成长。

何海涛

2016 年 12 月 7 日深夜于美国雷德蒙德

推荐序一

海涛 2008 年在我的团队做过软件开发工程师。他是一名很细心的员工，对面试这个话题很感兴趣，经常和我及其他员工讨论，积累了很多面试方面的技巧和经验。他曾跟我提过想要写本有关面试的书，如今他把书写出来了！他是一个有目标、有耐心和持久力的人。

我在微软做了很多年的面试官，后面 7 年多作为把关面试官，也面试了很多应聘者。应聘者要想做好面试，确实应把面试当作一门技巧来学习，更重要的是要提高自身的能力。我遇到很多应聘者可能自身能力也不差，但因为不懂得怎样回答提问，不能很好地发挥。也有很多刚走出校园的应聘者也学过数据结构和算法分析，可是在处理具体问题时不能用学过的知识来有效地解决。这些朋友读读海涛的这本书，会受益匪浅，在面试中的发挥也会有很大提高。这本书也可以作为很好的教学补充资料，让学生不仅学到书本知识，也学到解决问题的方法。

在我汇报的员工中有面试发挥很好但工作平平的，也有面试一般但工作优秀的。对于追求职业发展的人来说，通过面试只是迈过一道门槛而不是目的，真正的较量是在入职后的成长。就像学钓鱼，你可能在有经验的垂钓者的指导下能钓到几条鱼，但如果没学到垂钓的真谛，离开了指导者，你可能就很难钓到很多鱼。我希望读这本书的朋友不要只学一些技巧来应付面试，而是通过学习如何解决面试中的难题来提高自己的编程和解决问题的能力，进而提高自信心，在职场中迅速成长。

徐鹏阳 (Pung Xu)

Principal Development Manager, Search Technology Center Asia
Microsoft

推荐序二

I had the privilege of working with Harry at Microsoft. His background and industry experience are a great asset in learning about the process and techniques of technical interviews. Harry shares practical information about what to expect in a technical interview that goes beyond the core engineering skills. An interview is more than a skills assessment. It is the chance for you and a prospective employer to gauge whether there is a mutual fit. Harry includes reminders about the key factors that can determine a successful interview as well as success in your new job.

Harry takes you through a set of interview questions to share his insight into the key aspects of the question. By understanding these questions, you can learn how to approach any question more effectively. The basics of languages, algorithms and data structures are discussed as well as questions that explore how to write robust solutions after breaking down problems into manageable pieces. Harry also includes examples to focus on modeling and creative problem solving.

The skills that Harry teaches for problem solving can help you with your next interview and in your next job. Understanding better the key problem solving techniques that are analyzed in an interview can help you get the first job after university or make your next career move.

Matt Gibbs
Direct of Development, Asia Research & Development
Microsoft Corporation

前言

自 2011 年 9 月以来，我的面试题博客（<http://zhedahht.blog.163.com/>）点击率上升很快，累计点击量超过 70 万次，并且平均每天还会增加约 3000 次点击。每年随着秋季新学期的开始，新一轮招聘高峰也即将来到。这不禁让我想起几年前自己找工作的情形。那个时候的我，也是在网络的各个角落搜索面试经验，尽可能多地搜集各家公司的面试题。

当时网上的面试经验还很零散，应聘者如果想系统地搜集面试题，则需要付出很大的努力。于是我萌生了一个念头，在博客上系统地搜集、整理有代表性的面试题，这样可以极大地方便后来人。经过一段时间的准备，我于 2007 年 2 月在网易博客上发表了第一篇关于编程面试题的博文。

在之后的日子里，我陆续发表了 60 余篇关于面试题的博文。随着博文数目的增加，我也逐渐意识到一篇篇博文仍然是零散的。一篇博文只是单纯地分析一道面试题，但对解题思路缺乏系统性的梳理。于是，2010 年 10 月，我有了把博文整理成一本书的想法。经过努力，这本书终于和读者见面了。

本书内容

全书分为 7 章，各章的主要内容如下：

第 1 章介绍面试的流程。通常整个面试过程可以分为电话面试、共享桌面远程面试和现场面试 3 个阶段，每轮面试又可以分为行为面试、技术面试和应聘者提问 3 个环节。本章详细讨论了面试中每个环节需要注意的问题。其中，1.3.2 节深入讨论了技术面试中的 5 个要素，是全书的大纲，

接下来的第2~6章将逐一讨论每个要点。

第2章梳理应聘者在接受技术面试时需要用到的基础知识。本章从编程语言、数据结构及算法3个方面总结了程序员面试的知识点。

第3章讨论应聘者在面试时写出高质量代码的3个要点。通常面试官除了期待应聘者写出的代码能够完成基本的功能，还能应对特殊情况并对非法输入进行合理的处理。读完这一章，读者将学会如何从规范性、完整性和鲁棒性3个方面提高代码的质量。

第4章总结在编程面试中解决难题的常用思路。如果在面试过程中遇到复杂的难题，那么应聘者最好在写代码之前形成清晰的思路。读者在读完这一章之后，将学会如何用画图、举例和分解这3种思路来解决问题。

第5章介绍如何优化代码的时间效率和空间效率。如果一个问题有多种解法，那么面试官总是期待应聘者能找到最优的解法。读完这一章，读者将学会优化时间效率及用空间换时间的常用算法。

第6章总结面试中的各项能力。在面试过程中，面试官会一直关注应聘者的学习能力和沟通能力。除此之外，有些面试官还喜欢考查应聘者的知识迁移能力、抽象建模能力和发散思维能力。读完这一章，读者将学会如何培养和运用这些能力。

第7章是两个面试案例。在这两个案例中，读者将看到应聘者在面试过程中的哪些举动是不好的行为，而哪些表现又是面试官所期待的行为。衷心地希望应聘者能在面试时少犯甚至不犯错误，完美地表现出自己的综合素质，最终拿到心仪的Offer。

本书特色

正如前面提到的那样，本书的原型是我多年来陆陆续续发表的几十篇博文，但这本书也仅仅是这些博文的总和，它在博文的基础上添加了如下内容：

本书试图以面试官的视角来剖析面试题。本书前6章的第一节都是“面试官谈面试”，收录了分布在不同IT企业（或者IT部门）的面试官对代码质量、应聘者如何形成及表达解题思路等方面的理解。在本书中穿插着几十条“面试小提示”，是我作为面试官给应聘者在面试方法、技巧方面的建议。在第7章的案例中，“面试官心理”揭示了面试官在听到应聘者不同回

答时的心理活动。应聘者如果能了解面试官的心理活动，则无疑能在面试时更好地表现自己。

本书总结了解决面试难题的常用方法，而不仅仅是解决一道道零散的题目。在仔细分析、解决了几十道典型的面试题之后，我发现，其实是一些通用的方法可以在面试的时候帮助我们解题的。举个例子，如果面试的时候遇到的题目很难，那么我们可以试着把一个大的、复杂的问题分解成若干小的、简单的子问题，然后递归地去解决这些子问题。再比如，我们可以用数组实现一个简单的哈希表解决一系列与字符串相关的面试题。在详细分析了一道面试题之后，很多章节都会在“相关题目”中列举同类型的面试题，并在“举一反三”中总结解决这一类型题目的方法和要点。

本书收集的面试题都是各大公司的编程面试题，极具实战意义。包括 Google、微软在内的知名 IT 企业在招聘的时候都非常重视应聘者的编程能力，编程技术面试也是整个面试流程中最为重要的环节。本书选取的题目都是被各大公司面试官反复采用的编程题。如果读者一开始觉得书中的有些题目比较难，那也正常，没有必要感到气馁，因为像 Google、微软、阿里巴巴、腾讯这样的大企业的面试本身就不简单。读者逐步掌握了书中总结的解题方法之后，编程能力和分析复杂问题的能力将会得到很大的提升，再去大公司面试将会轻松很多。

本书附带提供了**80**道编程题的完整的源代码，其中包含每道题的测试用例。很多面试官在应聘者写完程序之后，都会要求应聘者自己想一些测试用例来测试自己的代码，而一些没有实际项目开发经验的应聘者不知道如何进行单元测试。相信读者在读完本书后就会知道如何从基本功能测试、边界值测试、性能测试等方面去设计测试用例，从而提高编写高质量代码的能力。

本书体例

在本书的正文中间或者章节的末尾穿插了不少特殊体例。这些体例或用来给应聘者提出建议，或用来总结解题方法，希望能够引起读者的注意。



面试小提示：

本条目是从面试官的角度给应聘者提出的建议，或者希望应聘者能够注意到的细节。



源代码：

读者将在本条目看到一个指向 GitHub 的链接，可以到对应的网页上浏览代码。同时，读者也可以把代码下载到本地，用 Visual Studio 打开 CodingInterviewChinese2.sln 文件阅读或者调试代码。



测试用例：

本条目列举应聘者在面试时可以用来测试代码是否完整、鲁棒的单元测试用例。通常本书从基本功能、边界值、无效的输入等方面测试代码的完整性和鲁棒性，针对在时间效率或者空间效率方面有要求的面试题还包含性能测试的测试用例。



本题考点：

本条目总结面试官采用一道面试题的考查要点。



相关题目：

本条目列举一些和详细分析的面试题相关或者类似的面试题。



举一反三：

本条目从解决面试例题中提炼出常用的解题方法。这些解题方法能够应用到解决其他同类型的问题中去，达到举一反三的目的。



面试官心理：

在第 7 章的面试案例中，本条目用来模拟面试官听到应聘者的回答之后的心理活动。

关于遗漏的问题

由于时间仓促，再加上笔者的能力有限，书中难免会有一些遗漏。今后一旦发现遗漏的问题，我将第一时间在博客（<http://zhedahht.blog.163.com/>）上公布勘误信息。读者如果发现任何问题或者有任何建议，那么也请在博客上留言、评论，或者通过电子邮件（zhedahht@hotmail.com）和我联系。

致谢

在写博客及把博文整理成书的过程中，我得到了很多人的帮助。没有他们，也就没有这本书。因此，我想在这里对他们诚挚地说一声：谢谢！

首先我要感谢个人博客上的读者。网友的鼓励让我在博客上的写作从2007年2月开始坚持到了现在。也正是由于网友的鼓励，我最终下定决心把博文整理成一本书。

在本书的写作过程中，我得到了很多同学、同事的帮助，包括 Autodesk 的马凌洲、刘景勇、王海波、蓝诚，支付宝的殷焰，百度的张珺、张晓禹，英特尔的尹彦，交通银行的朱麟，淘宝的尧敏，微软的陈黎明、田超，英伟达的吴斌，SAP 的何幸杰和华为的韩伟东（在书稿写作阶段他还在盛大工作）。感谢他们和大家分享了对编程面试的理解和思考。同时还要感谢 GlaxoSmithKline Investment 的 Recruitment & HRIS Manager 蔡咏来（也是2008年把我招进微软的 HR）和大家分享了微软所推崇的 STAR 简历模型。还要感谢在微软期间我的两个老板徐鹏阳和 Matt Gibbs，他们都是在微软有十几年面试经验的资深面试官，对面试有着深刻的理解。感谢二位在百忙之中抽时间为本书写序，为本书增色不少。

我同样要感谢现在思科的老板 Min Lu 及 TQSG 上海团队的同事王荔、赵斌和朱波对我的理解。他们在我写作期间替我分担了大量的工作，让我能够集中更多的精力来写书。

感谢电子工业出版社的工作人员，他们大到全书的构架，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。

本书还得到了很多朋友的支持和帮助，限于篇幅，虽然不能在此一一说出他们的名字，但我一样对他们心存感激。

最后，我要衷心地感谢我的爱人刘素云。感谢她在过去一年中对我的理解和支持，为我营造了一个温馨而又浪漫的家，让我能够心无旁骛地写书。我无以为谢，谨以此书献给她及我们的孩子。

何海涛

2017年2月于上海三泾南宅

轻松注册成为博文视点社区用户（www.broadview.com.cn），您即可享受以下服务。

- **下载资源：**本书所提供的示例代码及资源文件均可在【下载资源】处下载。
- **提交勘误：**您对书中内容的修改意见可在【提交勘误】处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用 来抵扣相应金额）。
- **与我们交流：**在页面下方【读者评论】处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/31092>

二维码：



目 录

第 1 章 面试的流程	1
1.1 面试官谈面试	1
1.2 面试的 3 种形式	2
1.2.1 电话面试	2
1.2.2 共享桌面远程面试	3
1.2.3 现场面试	4
1.3 面试的 3 个环节	5
1.3.1 行为面试环节	5
1.3.2 技术面试环节	10
1.3.3 应聘者提问环节	17
1.4 本章小结	18
第 2 章 面试需要的基础知识	21
2.1 面试官谈基础知识	21
2.2 编程语言	22
2.2.1 C++	23

面试题 1：赋值运算符函数	25
2.2.2 C#.....	28
面试题 2：实现 Singleton 模式	32
2.3 数据结构.....	37
2.3.1 数组	37
面试题 3：数组中重复的数字	39
面试题 4：二维数组中的查找	44
2.3.2 字符串	48
面试题 5：替换空格	51
2.3.3 链表	56
面试题 6：从尾到头打印链表	58
2.3.4 树	60
面试题 7：重建二叉树	62
面试题 8：二叉树的下一个节点	65
2.3.5 栈和队列	68
面试题 9：用两个栈实现队列	68
2.4 算法和数据操作.....	72
2.4.1 递归和循环	73
面试题 10：斐波那契数列	74
2.4.2 查找和排序	79
面试题 11：旋转数组的最小数字	82
2.4.3 回溯法	88
面试题 12：矩阵中的路径	89
面试题 13：机器人的运动范围	92
2.4.4 动态规划与贪婪算法	94

面试题 14: 剪绳子	96
2.4.5 位运算	99
面试题 15: 二进制中 1 的个数	100
2.5 本章小结	104
 第 3 章 高质量的代码.....	105
3.1 面试官谈代码质量	105
3.2 代码的规范性	106
3.3 代码的完整性	107
面试题 16: 数值的整数次方	110
面试题 17: 打印从 1 到最大的 n 位数	114
面试题 18: 删除链表的节点	119
面试题 19: 正则表达式匹配	124
面试题 20: 表示数值的字符串	127
面试题 21: 调整数组顺序使奇数位于偶数前面	129
3.4 代码的鲁棒性	133
面试题 22: 链表中倒数第 k 个节点	134
面试题 23: 链表中环的入口节点	139
面试题 24: 反转链表	142
面试题 25: 合并两个排序的链表	145
面试题 26: 树的子结构	148
3.5 本章小结	152
 第 4 章 解决面试题的思路	155
4.1 面试官谈面试思路	155
4.2 画图让抽象问题形象化	156

面试题 27：二叉树的镜像	157
面试题 28：对称的二叉树	159
面试题 29：顺时针打印矩阵	161
4.3 举例让抽象问题具体化.....	165
面试题 30：包含 min 函数的栈	165
面试题 31：栈的压入、弹出序列	168
面试题 32：从上到下打印二叉树	171
面试题 33：二叉搜索树的后序遍历序列	179
面试题 34：二叉树中和为某一值的路径	182
4.4 分解让复杂问题简单化.....	186
面试题 35：复杂链表的复制	187
面试题 36：二叉搜索树与双向链表	191
面试题 37：序列化二叉树	194
面试题 38：字符串的排列	197
4.5 本章小结.....	201
第 5 章 优化时间和空间效率	203
5.1 面试官谈效率.....	203
5.2 时间效率.....	204
面试题 39：数组中出现次数超过一半的数字.....	205
面试题 40：最小的 k 个数.....	209
面试题 41：数据流中的中位数	214
面试题 42：连续子数组的最大和	218
面试题 43：1~n 整数中 1 出现的次数	221
面试题 44：数字序列中某一位的数字	225

面试题 45：把数组排成最小的数	227
面试题 46：把数字翻译成字符串	231
面试题 47：礼物的最大价值	233
面试题 48：最长不含重复字符的子字符串	236
5.3 时间效率与空间效率的平衡.....	239
面试题 49：丑数	240
面试题 50：第一个只出现一次的字符	243
面试题 51：数组中的逆序对	249
面试题 52：两个链表的第一个公共节点	253
5.4 本章小结.....	256
第 6 章 面试中的各项能力.....	257
6.1 面试官谈能力	257
6.2 沟通能力和学习能力	258
6.3 知识迁移能力	261
面试题 53：在排序数组中查找数字	263
面试题 54：二叉搜索树的第 k 大节点	239
面试题 55：二叉树的深度	271
面试题 56：数组中数字出现的次数	275
面试题 57：和为 s 的数字	280
面试题 58：翻转字符串	284
面试题 59：队列的最大值	288
6.4 抽象建模能力	294
面试题 60： n 个骰子的点数	294
面试题 61：扑克牌中的顺子	298

面试题 62：圆圈中最后剩下的数字	300
面试题 63：股票的最大利润	304
6.5 发散思维能力	306
面试题 64：求 $1+2+\cdots+n$	307
面试题 65：不用加减乘除做加法	310
面试题 66：构建乘积数组	312
6.6 本章小结	314
第 7 章 两个面试案例	317
7.1 案例一：（面试题 67）把字符串转换成整数	318
7.2 案例二：（面试题 68）树中两个节点的最低公共祖先	326

第1章

面试的流程

1.1 面试官谈面试

“对于初级程序员，我一般会偏向考查算法和数据结构，看应聘者的基本功；对于高级程序员，我会多关注专业技能和项目经验。”

——何幸杰（SAP，高级工程师）

“应聘者要事先做好准备，对公司近况、项目情况有所了解，对所应聘的工作很有热情。另外，应聘者还要准备好合适的问题问面试官。”

——韩伟东（盛大，高级研究员）

“应聘者在面试过程中首先需要放松，不要过于紧张，这有助于后面解决问题时开拓思路。其次不要急于编写代码，应该先了解清楚所要解决的问题。这时候最好先和面试官多做沟通，然后开始做一些整体的设计和规划，这有助于编写高质量和高可读性的代码。写完代码后不要马上提交，最好自己检查并借助一些测试用例来测试几遍代码，找出可能出现的错误。”

——尧敏（淘宝，资深经理）

“‘神马’都是浮云，应聘技术岗位就是要踏实写程序。”

——田超（微软，SDE II）

1.2 面试的3种形式

如果应聘者能够通过公司的简历筛选环节，那恭喜他取得了阶段性的成功。但要想拿到心仪的Offer，应聘者还有更长的路要走。大部分公司的面试都是从电话面试开始的。通过电话面试之后，有些公司还会有一两轮远程面试。面试官让应聘者共享自己的桌面，远程观察应聘者编写及调试代码的过程。如果前面的面试都很顺利，应聘者就会收到现场面试的邀请信，请他去公司接受面对面的面试。整个面试的流程我们可以用图1.1表示。

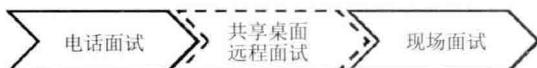


图1.1 面试的形式和流程

注：只有少数公司有共享桌面远程面试环节。

1.2.1 电话面试

顾名思义，电话面试是面试官以打电话的形式考查应聘者。有些面试官会先和应聘者预约好电话面试的时间，而还有些面试官却喜欢搞突然袭击，一个电话打过去就开始面试。为了应付这种突然袭击，建议应聘者在投出简历之后的一两个星期之内，要保证手机电池能至少连续通话一小时。另外，应聘者不要长时间待在很嘈杂的地方。如果应聘者身在闹市的时候突然接到面试电话，那么双方就有可能因为听不清对方而倍感尴尬。

电话面试和现场面试最大的区别就是应聘者和面试官是见不到对方的，因此双方的沟通只能依靠声音。没有了肢体语言、面部表情，应聘者清楚地表达自己想法的难度就比现场面试时要大很多，特别是在解释复杂算法的时候。应聘者在电话面试的时候应尽可能用形象化的语言把细节说清楚。例如，在现场面试的时候，应聘者如果想说一棵二叉树的结构，则可以用笔在白纸上画出来，就一目了然了。但在电话面试的时候，应聘者就需要把二叉树中有哪些节点，每个节点的左子节点是什么、右子节点是什么都说得很清楚，只有这样面试官才能准确地理解应聘者的思路。

很多外企在电话面试时都会加上英语面试的环节，甚至有些公司全部

面试都会用英语进行。电话面试时应聘者只能听到面试官的声音而看不到他的口型，这对应聘者的听力提出了更高的要求。如果应聘者在面试的时候没有听清楚或者听懂面试官的问题，则千万不要不懂装懂、答非所问，这是面试的大忌。当不确定面试官的问题的时候，应聘者一定要大胆地向面试官多提问，直到弄清楚面试官的意图为止。



面试小提示：

应聘者在电话面试的时候应尽可能用形象的语言把细节说清楚。

如果在英语面试时没有听清或没有听懂面试官的问题，则应聘者要敢于说 Pardon。

1.2.2 共享桌面远程面试

共享桌面远程面试（Phone-Screen Interview）是指利用一些共享桌面的软件（如微软的 Skype、思科的 WebEx 等），应聘者把自己电脑的桌面共享给远程的面试官。这样两个人虽然没有坐在一起，但面试官却能通过共享桌面观看应聘者编程和调试的过程。目前只有为数不多的几家大公司会在邀请应聘者到公司参加现场面试之前，先进行一两轮共享桌面远程面试。

这种形式的面试，面试官最关心的是应聘者的编程习惯及调试能力。通常面试官会认可应聘者下列几种编程习惯：

- **思考清楚再开始编码。**应聘者不要一听到题目就匆忙打开编程软件如 Visual Studio 开始敲代码，因为在没有形成清晰的思路之前写出的代码通常会漏洞百出。这些漏洞被面试官发现之后，应聘者容易慌张，这个时候再修改代码也会越改越乱，最终导致面试的结果不理想。更好的策略是应聘者应先想清楚解决问题的思路，如算法的时间、空间复杂度各是什么，有哪些特殊情况需要处理等，然后再动手编写代码。
- **良好的代码命名和缩进对齐习惯。**一目了然的变量和函数名，加以合理的缩进和括号对齐，会让面试官觉得应聘者有参与大型项目的开发经验。

- 能够进行单元测试。通常面试官出的题目都是要求写函数解决某一问题，如果应聘者能够在定义函数之后，立即对该函数进行全面的单元测试，那就相当于向面试官证明了自己有着专业的软件开发经验。如果应聘者先写单元测试用例，再写解决问题的函数，那么我相信面试官定会对你刮目相看，因为能做到测试在前、开发在后的程序员实在是太稀缺了，他会毫不犹豫地抛出橄榄枝。

通常我们在写代码的时候都会遇到问题。当应聘者运行代码发现结果不对之后的表现，也是面试官关注的重点，因为应聘者此时的反应、采取的措施都能体现出他的调试功底。如果应聘者能够熟练地设置断点、单步跟踪、查看内存、分析调用栈，就能很快发现问题的根源并最终解决问题，那么面试官将会觉得他的开发经验很丰富。调试能力是在书本上学不到的，只有通过大量的软件开发实践才能积累出调试技巧。当面试官发现一个应聘者的调试功底很扎实的时候，他在写面试报告的时候是不会吝啬赞美之词的。



面试小提示：

在共享桌面远程面试过程中，面试官最关心的是应聘者的编程习惯及调试能力。

1.2.3 现场面试

在通过电话面试和共享桌面远程面试之后，应聘者不久就会收到E-mail，邀请他去公司参加现场面试（Onsite Interview）。

在去公司参加现场面试之前，应聘者应做好以下几点准备：

- 规划好路线并估算出行时间。应聘者要事先估算在路上需要花费多长时间，并预留半小时左右的缓冲时间以应对堵车等意外情况。如果面试迟到，那至少印象分会大打折扣。
- 准备好得体的衣服。IT公司通常衣着比较随意，应聘者通常没有必要穿着正装，一般舒服干净的衣服都可以。
- 注意面试邀请函里的面试流程。如果面试有好几轮，时间也很长，那么你在面试过程中可能会觉得疲劳且思维变得迟钝。比如微软对

技术职位通常有 5 轮面试，连续几小时处在高压的面试之中，人难免会变得精疲力竭。因此，应聘者可以带一些提神的饮料或者食品，在两轮面试之间提神醒脑。

- 准备几个问题。每一轮面试的最后，面试官都会让应聘者问几个问题，应聘者可以提前准备好问题。

现场面试是整个面试流程中的重头戏。由于是坐在面试官的对面，应聘者的一举一动都看在面试官的眼里。面试官通过应聘者的语言和行动考查他的沟通能力、学习能力、编程能力等综合实力。本书接下来的章节将详细讨论各种能力。

1.3 面试的 3 个环节

通常面试官会把每一轮面试分为 3 个环节（如图 1.2 所示）：首先是行为面试，面试官参照简历了解应聘者的过往经验；其次是技术面试，这一环节很有可能会要求应聘者现场写代码；最后一个环节是应聘者问几个自己最感兴趣的问题。下面将详细讨论面试的这 3 个环节。

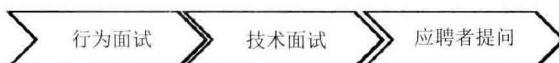


图 1.2 面试的 3 个环节

1.3.1 行为面试环节

面试开始的 5~10 分钟通常是行为面试的时间。在行为面试这个环节里，面试官会注意应聘者的性格特点，深入地了解简历中列举的项目经历。由于这一环节一般不会问技术难题，因此也是一个暖场的过程，应聘者可以利用这几分钟调整自己的情绪，进入面试的状态。

不少面试官会让应聘者做一个简短的自我介绍。由于面试官手中拿着应聘者的简历，而那里有应聘者的详细信息，因此此时的自我介绍不用花很多时间，用 30 秒到 1 分钟的时间介绍自己的主要学习、工作经历即可。

如果面试官对你的某一段经历或者参与的某一个项目很感兴趣，那么他会有针对性地提几个问题详细了解。

1. 应聘者的项目经验

应聘者自我介绍之后，面试官接着会对照应聘者的简历去详细了解他感兴趣的项目。应聘者在准备简历的时候，建议用如图1.3所示的STAR模型描述自己经历过的每一个项目。



图1.3 简历中描述项目的STAR模型

- **Situation:** 简短的项目背景。比如项目的规模，开发的软件的功能、目标用户等。
- **Task:** 自己完成的任务。这个要写详细，要让面试官对自己的工作一目了然。在用词上要注意区分“参与”和“负责”：如果只是加入某一个开发团队写了几行代码就用“负责”，那就很危险。面试官看到简历上应聘者“负责”了某个项目，他可能就会问项目的总体框架设计、核心算法、团队合作等问题。这些问题对于只是简单“参与”的人来说，是很难回答的，会让面试官认为你不诚实，印象分会减去很多。
- **Action:** 为完成任务自己做了哪些工作，是怎么做的。这里可以详细介绍。做系统设计的，可以介绍系统架构的特点；做软件开发的，可以写基于什么工具在哪个平台下应用了哪些技术；做软件测试的，可以写是手工测试还是自动化测试、是白盒测试还是黑盒测试等。
- **Result:** 自己的贡献。这方面的信息可以写得具体些，最好能用数字加以说明。如果是参与功能开发，则可以说按时完成了多少功能；

如果做优化，则可以说性能提高的百分比是多少；如果是维护，则可以说修改了多少个 Bug。

举个例子，笔者用下面一段话介绍自己在微软 Winforms 项目组的经历：

Winforms 是微软.NET 中的一个成熟的 UI 平台（**Situation**）。本人的工作是在添加少量新功能之外主要负责维护已有的功能（**Task**）。新的功能主要是让 Winforms 的控件风格和 Vista、Windows 7 的风格保持一致。在维护方面，对于较难的问题，我用 WinDbg 等工具进行调试（**Action**）。在过去两年中，我共修改了超过 200 个 Bug（**Result**）。

如果在应聘者的简历中上述 4 类信息还不够清晰，则面试官可能会追问相关的问题。除此之外，面试官针对项目经验最常问的问题包括如下几个类型：

- 你在该项目中碰到的最大问题是什么，你是怎么解决的？
- 从这个项目中你学到了什么？
- 什么时候会和其他团队成员（包括开发人员、测试人员、设计人员、项目经理等）有什么样的冲突，你们是怎么解决冲突的？

应聘者在准备简历的时候，针对每一个项目经历都应提前做好相应的准备。只有准备充分，应聘者在行为面试环节才可以表现得游刃有余。



面试小提示：

在介绍项目经验（包括在简历上介绍和面试时口头介绍）时，应聘者不必详述项目的背景，而要突出介绍自己完成的工作及取得的成绩。

2. 应聘者掌握的技能

除应聘者参与过的项目之外，面试官对应聘者掌握的技能也很感兴趣，他有可能针对简历上提到的技能提出问题。和描述项目时要注意“参与”和“负责”一样，描述技能掌握程度时也要注意“了解”、“熟悉”和“精通”的区别。

“了解”指对某项技术只是上过课或者看过书，但没有做过实际的项目。通常不建议在简历中列出只是肤浅地了解一点的技能，除非这项技术应聘的职位的确需要。比如某学生读本科的时候学过《计算机图形学》这门课

程，但一直没有开发过与图形绘制相关的项目，那就只能算是了解。如果他去应聘 Autodesk 公司，那么他可以在简历上提一下他了解图形学。Autodesk 是一家开发三维设计软件的公司，有很多职位或多或少会与图形学有关系，那么了解图形学的总比完全不了解的要适合一些。但如果他是去应聘 Oracle，那就没有必要提这一点了，因为开发数据库系统的 Oracle 公司大部分职位与图形学没有什么关系。

简历中我们描述技能的掌握程度大部分应该是“熟悉”。如果我们在实际项目中使用某项技术已经有较长的时间，通过查阅相关的文档可以独立解决大部分问题，那么我们就熟悉它了。对应届毕业生而言，他毕业设计所用到的技能可以用“熟悉”；对已经工作过的，在项目开发过程中所用到的技能，也可以用“熟悉”。

如果我们对一项技术使用得得心应手，在项目开发过程中，当同学或同事向我们请教这个领域的问题时，我们都有信心也有能力解决，这个时候我们就可以说自己精通了这项技术。应聘者不要试图在简历中把自己修饰成“高人”而轻易使用“精通”，除非自己能够很轻松地回答这个领域里的绝大多数问题，否则就会适得其反。通常如果应聘者在简历中说自己精通某项技术，面试官就会对他有很高的期望值，因此会挑一些比较难的问题来问。这也是越装高手就越容易露馅的原因。曾经碰到一个在简历中说自己精通 C++ 的应聘者，连成员变量的初始化顺序这样的问题都被问得一头雾水，那最终的结果也就可想而知了。

3. 回答“为什么跳槽”

在面试已经有工作经验的应聘者的时候，面试官总喜欢问为什么打算跳槽。每个人都有自己的跳槽动机和原因，因此面试官也不会期待一个标准答案。面试官只是想通过这个问题来了解应聘者的性格，因此应聘者可以大胆地根据自己的真实想法来回答这个问题。但是，应聘者也不要想要说什么就说什么，以免给面试官留下负面的印象。

在回答这个问题时不要抱怨，也不要流露出负面的情绪。负面的情绪通常是能够传染的，当应聘者总是在抱怨的时候，面试官就会担心如果把他招进来，那么他将成为团队负面情绪的传染源，从而影响整个团队的士气。应聘者应尽量避免以下 4 个原因：

- 老板太苛刻。如果面试官就是当前招聘的职位的老板，那么当他听

到应聘者抱怨现在的老板苛刻时，他肯定会想要是把这个人招进来，接下来他就会抱怨我也苛刻了。

- 同事太难相处。如果应聘者说他周围有很多很难相处的同事，则面试官很有可能会觉得这个人本身就很难相处。
- 加班太频繁。对于大部分 IT 企业来说，加班是家常便饭。如果正在面试的公司也需要经常加班，那等于应聘者说他不想进这家公司。
- 工资太低。现在的工资太低的确是大部分人跳槽的真实原因，但不建议在面试的时候对面试官抱怨。面试的目的是拿到 Offer，我们要尽量给面试官留下好印象。现在假设你是面试官，有两个人来面试：一个人一开口就说现在工资太低了，希望新工作能加多少多少工资；另一个说我只管努力干活，工资公司看着给，相信公司不会亏待勤奋的员工。你更喜欢哪个？这里不是说工资不重要，但我们要清楚面试不是谈工资的时候。等完成技术面试之后谈 Offer 的时候，再和 HR 谈工资也不迟。通过面试之后我们就掌握主动权了，想怎么谈就怎么谈，如果工资真的开高了，那么 HR 会和你很客气地商量。

笔者在面试的时候通常给出的答案是：现在的工作做了一段时间，已经没有太多的激情了，因此希望寻找一份更有挑战的工作。然后具体论述为什么有些厌倦现在的职位，以及面试的职位我为什么会有兴趣。笔者自己跳过几次槽，第一次从 Autodesk 跳槽到微软，第二次从微软跳槽到思科，后来又从思科回到了微软。从面试的结果来看，这样的回答都让面试官很满意，最终也都拿到了 Offer。

当时在微软面试被问到为什么要跳槽时，笔者的回答是：我在 Autodesk 开发的软件 Civil 3D 是一款面向土木行业设计软件。如果我想在现在的职位上得到提升，就必须加强土木行业的学习，可我对诸如计算土方量、道路设计等没有太多兴趣，因此出来寻找机会。

在微软工作两年半之后去思科面试的时候，笔者的回答是：我在微软的主要工作是开发和维护.NET 的 UI 平台 Winforms。由于 Winforms 已经非常成熟，不需要添加多少新功能，因此我的大部分工作是维护和修改 Bug。两年下来，调试的能力得到了很大的提高，但长期如此，自己的软件开发和设计能力将不能得到提高，因此想出来寻找可以设计和开发系统的职位。

同时，我在过去几年里的工作都是开发桌面软件，对网络了解甚少，因此希望下一个工作能与网络相关。众所周知，思科是一家网络公司，这里的软件和系统或多或少都离不开网络，因此我对思科的职位很感兴趣。

1.3.2 技术面试环节

面试官在通过简历及行为面试大致了解应聘者的背景之后，接下来就要开始技术面试了。一轮一小时的面试，通常技术面试会占据40~50分钟。这是面试的重头戏，对面试的结果起决定性作用。虽然不同公司不同面试官的背景、性格各不相同，但总体来说他们都会关注应聘者的5种素质：扎实的基础知识、能写高质量的代码、分析问题时思路清晰、能优化时间效率和空间效率，以及学习沟通等各方面的能力（如图1.4所示）。

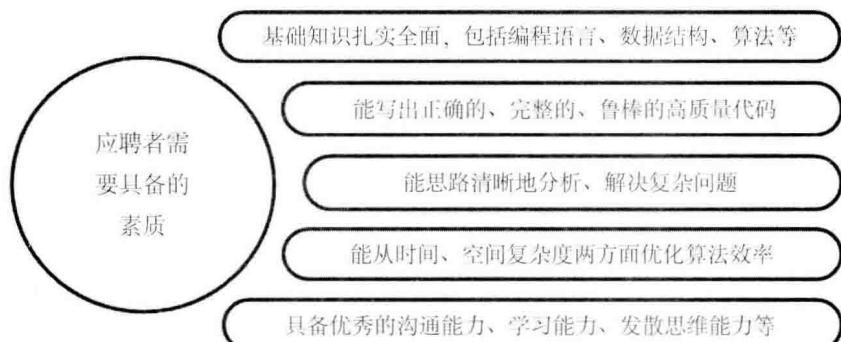


图1.4 应聘者需要具备的素质

应聘者在面试之前需要做足准备，对编程语言、数据结构和算法等基础知识有全面的了解。面试的时候如果遇到简单的问题，则应聘者一定要注重细节，写出完整、鲁棒的代码。如果遇到复杂的问题，则应聘者可以通过画图、举具体例子分析和分解复杂问题等方法先厘清思路再动手编程。除此之外，应聘者还应该不断优化时间效率和空间效率，力求找到最优的解法。在面试过程中，应聘者还应该主动提问，以弄清楚题目要求，表现自己的沟通能力。当面试官前后问的两个问题有相关性的时候，尽量把解决前面问题的思路迁移到后面的问题中去，展示自己良好的学习能力。如果能做到这几点，那么通过面试获得心仪的职位将是水到渠成的事情。

1. 扎实的基础知识

扎实的基本功是成为优秀程序员的前提条件，因此面试官首要关注的应聘者素质就是是否具备扎实的基础知识。通常基本功在编程面试环节体现在3个方面：编程语言、数据结构和算法。

首先，每个程序员至少要掌握一两门编程语言。面试官从应聘者在面试过程中写的代码及跟进的提问中能看出其编程语言掌握的熟练程度。以很多公司面试要求的C++举例。如果写的函数需要传入一个指针，则面试官可能会问是否需要为该指针加上const、把const加在指针不同的位置是否有区别；如果写的函数需要传入的参数是一个复杂类型的实例，则面试官可能会问传入值参数和传入引用参数有什么区别、什么时候需要为传入的引用参数加上const。

其次，数据结构通常是编程面试过程中考查的重点。在参加面试之前，应聘者需要熟练掌握链表、树、栈、队列和哈希表等数据结构，以及它们的操作。如果我们留意各大公司的面试题，就会发现与链表和二叉树相关的问题是很多面试官喜欢问的问题。这方面的问题看似比较简单，但要真正掌握也不容易，特别适合在这么短的面试时间内检验应聘者的基本功。如果应聘者事先对链表的插入和删除节点了如指掌，对二叉树的各种遍历方法的循环和递归写法都烂熟于胸，那么真正到了面试的时候也就游刃有余了。

最后，大部分公司都会注重考查查找、排序等算法。应聘者可以在了解各种查找和排序算法的基础上，重点掌握二分查找、归并排序和快速排序，因为很多面试题都只是这些算法的变体而已。比如面试题11“旋转数组的最小数字”和面试题53“在排序数组中查找数字”的本质是考查二分查找，而面试题51“数组中的逆序对”实际上是考查归并排序。少数对算法很重视的公司如谷歌或者百度，还会要求应聘者熟练掌握动态规划和贪婪算法。如果应聘者对动态规划算法很熟悉，那么他就能很轻松地解决面试题14“剪绳子”。

在本书的第2章“面试需要的基础知识”中，我们将详细介绍应聘者需要熟练掌握的基础知识。

2. 高质量的代码

只有注重质量的程序员，才能写出鲁棒、稳定的大型软件。在面试过程中，面试官总会格外关注边界条件、特殊输入等看似细枝末节但实则至

关重要的地方，以考查应聘者是否注重代码质量。很多时候，面试官发现应聘者写出来的代码只能完成最基本的功能，一旦输入特殊的边界条件参数，就会错误百出甚至程序崩溃。

总有些应聘者很困惑：面试的时候觉得题目很简单，感觉自己都做出来了，可最后为什么被拒了呢？面试被拒有很多种可能，比如面试官认为你性格不适合、态度不够诚恳等。但在技术面试过程中，这些都不是最重要的。技术面试的面试官一般都是程序员，程序员通常没有那么多想法。他们只认一个理：题目做对、做完整了，就让你通过面试；否则失败。所以遇到简单题目却被拒的情况，应聘者应认真反思在思路或者代码中存在哪些漏洞。

以微软面试开发工程师时最常用的一个问题为例：把一个字符串转换成整数。这个题目很简单，很多人都能在3分钟之内写出如下不到10行的代码：

```
int StrToInt(char* string)
{
    int number = 0;
    while(*string != 0)
    {
        number = number * 10 + *string - '0';
        ++string;
    }

    return number;
}
```

看了上面的代码，你是不是觉得微软面试很容易？如果你真的这么想，那你可能又要被拒了。

通常越是简单的问题，面试官的期望值就会越高。如果题目很简单，面试官就会期待应聘者能够很完整地解决问题，除完成基本功能之外，还要考虑到边界条件、错误处理等各个方面。比如这道题，面试官不仅仅期待你能完成把字符串转换成整数这个最起码的要求，而且希望你能考虑到各种特殊的输入。面试官至少会期待应聘者能够在不需要提示的情况下，考虑到输入的字符串中有非数字字符和正负号，要考虑到最大的正整数和最小的负整数以及溢出。同时面试官还期待应聘者能够考虑到当输入的字符串不能转换成整数时，应该如何做错误处理。当把这个问题的方方面面都考虑到的时候，我们就不会再认为这道题简单了。

除问题考虑不全面之外，还有一个面试官不能容忍的错误就是程序不

够鲁棒。以前面的那段代码为例，只要输入一个空指针，程序立即崩溃。这样的代码如果加入软件当中，那么将是灾难。因此，当面试官看到代码中对空指针没有判断并加以特殊处理的时候，通常他连往下看的兴趣都没有。

当然，不是所有与鲁棒性相关的问题都和前面的代码那样明显。再举一个很多人被面试过的问题：求链表中的倒数第 k 个节点。有不少人在面试之前在网上看过这个题目，因此知道思路是用两个指针，第一个指针先走 $k-1$ 步，然后两个指针一起走。当第一个指针走到尾节点的时候，第二个指针指向的就是倒数第 k 个节点。于是他大笔一挥，写下了下面的代码：

```
ListNode* FindKthToTail(ListNode* pListHead, unsigned int k)
{
    if(pListHead == nullptr)
        return nullptr;

    ListNode *pAhead = pListHead;
    ListNode *pBehind = nullptr;

    for(unsigned int i = 0; i < k - 1; ++ i)
    {
        pAhead = pAhead->m_pNext;
    }

    pBehind = pListHead;

    while(pAhead->m_pNext != nullptr)
    {
        pAhead = pAhead->m_pNext;
        pBehind = pBehind->m_pNext;
    }

    return pBehind;
}
```

写完之后，应聘者看到自己已经判断了输入的指针是不是空指针并进行了特殊处理，于是以为这次面试必定能顺利通过。可是他没有想到的是这段代码中仍然有很严重的问题：当链表中的节点总数小于 k 的时候，程序还是会崩溃；另外，当输入的 k 为 0 时，同样也会引起程序崩溃。因此，几天之后他收到的仍然不是 Offer 而是拒信。

要想很好地解决前面的问题，最好的办法是在动手写代码之前想好测试用例。只有把各种可能的输入事先都想好了，才能在写代码的时候把各种情况都进行相应的处理。写完代码之后，也不要立刻给面试官检查，而是先在心里默默地运行。当输入之前想好的所有测试用例都能得到合理的

输出时，再把代码交给面试官。做到了这一步，通过面试拿到 Offer 就是顺理成章的事情了。

在本书的第3章“高质量的代码”中，我们将详细讨论提高代码质量的方法。



面试小提示：

面试官除了希望应聘者的代码能够完成基本的功能，还会关注应聘者是否考虑了边界条件、特殊输入（如 nullptr 指针、空字符串等）及错误处理。

3. 清晰的思路

只有思路清晰，应聘者才有可能在面试过程中解决复杂的问题。有时候面试官会有意出一些比较复杂的问题，以考查应聘者能否在短时间内形成清晰的思路并解决问题。对于确实很复杂的问题，面试官甚至不期待应聘者能在面试不到一小时的时间里给出完整的答案，他更看重的可能还是应聘者是否有清晰的思路。面试官通常不喜欢应聘者在没有形成清晰思路之前就草率地开始写代码，这样写出来的代码容易逻辑混乱、错误百出。

应聘者可以用几个简单的方法帮助自己形成清晰的思路。首先，举几个简单的具体例子让自己理解问题。当我们一眼看不出问题中隐藏的规律的时候，可以试着用一两个具体的例子模拟操作的过程，这样说不定就能通过具体的例子找到抽象的规律。其次，可以试着用图形表示抽象的数据结构。像分析与链表、二叉树相关的题目，我们都可以画出它们的结构来简化题目。最后，可以试着把复杂的问题分解成若干简单的子问题，再一一解决。很多基于递归的思路，包括分治法和动态规划，都是把复杂的问题分解成一个或者多个简单的子问题。

比如把二叉搜索树转换成排序的双向链表这个问题就很复杂。遇到这个问题，我们不妨先画出一两棵具体的二叉搜索树，直观地感受二叉搜索树和排序的双向链表有哪些联系。如果一下子找不出转换的规律，则可以把整棵二叉树看成3个部分：根节点、左子树和右子树。当我们递归地把转换左右子树这两个子问题解决之后，再把转换左右子树得到的链表和根节点链接起来，整个问题也就解决了（详见面试题36“二叉搜索树与双向链表”）。

在本书的第4章“解决面试题的思路”中，我们将详细讨论遇到复杂问题时如何采用画图、举例和分解问题等方法帮助我们解决问题。



面试小提示：

如果在面试的时候遇到难题，我们有3种办法分析、解决复杂的问题：画图能使抽象问题形象化，举例使抽象问题具体化，分解使复杂问题简单化。

4. 优化效率的能力

优秀的程序员对时间和内存的消耗锱铢必较，他们很有激情地不断优化自己的代码。当面试官出的题目有多种解法的时候，通常他会期待应聘者最终能够找到最优解。当面试官提示还有更好的解法的时候，应聘者不能放弃思考，而应该努力寻找在时间消耗或者空间消耗上可以优化的地方。

要想优化时间或者空间效率，首先要知道如何分析效率。即使同一种算法，用不同方法实现的效率可能也会大不相同，我们要能够分析出算法及其代码实现的效率。例如，求斐波那契数列，很多人喜欢用递归公式 $f(n)=f(n-1)+f(n-2)$ 求解。如果分析它的递归调用树，我们就会发现有大量的计算是重复的，时间复杂度以 n 的指数增加。但如果我们在先求 $f(1)$ 和 $f(2)$ ，再根据 $f(1)$ 和 $f(2)$ 求出 $f(3)$ ，接下来根据 $f(2)$ 和 $f(3)$ 求出 $f(4)$ ，并以此类推用一个循环求出 $f(n)$ ，这种计算方法的时间效率就只有 $O(n)$ ，比前面递归的方法要好得多。

要想优化代码的效率，我们还要熟知各种数据结构的优缺点，并能选择合适的数据结构解决问题。我们在数组中根据下标可以用 $O(1)$ 时间完成查找。数组的这个特征可以实现用简单的哈希表解决很多问题，如面试题50“第一个只出现一次的字符”。为了解决面试题40“最小的 k 个数”，我们需要一个数据容器来存储 k 个数字。在这个数据容器中，我们希望能够快速地找到最大值，并且能快速地替换其中的数字。经过权衡，我们发现二叉树如最大堆或者红黑树都是实现这个数据容器的不错选择。

要想优化代码的效率，我们也要熟练掌握常用的算法。面试中最常用的算法是查找和排序。如果从头到尾顺序扫描一个数组，那么我们需要 $O(n)$

时间才能完成查找操作。但如果数组是排序的，应用二分查找算法就能把时间复杂度降低到 $O(\log n)$ （详见面试题 11 “旋转数组的最小数字”和面试题 53 “在排序数组中查找数字”）。排序算法除了能够给数组排序，还能用来解决其他问题。比如快速排序算法中的 Partition 函数能够用来在 n 个数里查找第 k 大的数字，从而解决面试题 39 “数组中出现次数超过一半的数字”和面试题 40 “最小的 k 个数”。归并排序算法能够实现在 $O(n \log n)$ 时间统计 n 个数字中的逆序对数目（详见面试题 51 “数组中的逆序对”）。

在本书的第 5 章“优化时间和空间效率”中，我们将详细讨论如何从时间效率和空间效率两方面进行优化。

5. 优秀的综合能力

在面试过程中，应聘者除了展示自己的编程能力和技术功底，还需要展示自己的软技能（Soft Skills），诸如自己的沟通能力和学习能力。随着软件系统的规模越来越大，软件开发已经告别了单打独斗的年代，程序员与他人的沟通变得越来越重要。在面试过程中，面试官会观察应聘者在介绍项目经验或者算法思路时是否观点明确、逻辑清晰，并以此判断其沟通能力的强弱。另外，面试官也会从应聘者说话的神态和语气来判断他是否有团队合作的意识。通常面试官不会喜欢高傲或者轻视合作者的人。

IT 行业知识更新很快，因此程序员只有具备很好的学习能力才能跟上知识更替的步伐。通常面试官有两种办法考查应聘者的学习能力。第一种方法是询问应聘者最近在看什么书、从中学到了哪些新技术。面试官可以用这个问题了解应聘者的学习愿望和学习能力。第二种方法是抛出一个新概念，接下来他会观察应聘者能不能在较短的时间内理解这个新概念并解决相关的问题。比如面试官要求应聘者计算第 1500 个丑数。很多人都没有听说过丑数这个概念。这时候面试官就会观察应聘者面对丑数这个新概念时，能不能经过提问、思考、再提问的过程，最终找出丑数的规律，从而找到解决方案（详见面试题 49 “丑数”）。

知识迁移能力是一种特殊的学习能力。如果我们能够把已经掌握的知识迁移到其他领域，那么学习新技术或者解决新问题就会变得容易。面试官经常会先问一个简单的问题，再问一个很复杂但和前面的简单问题相关的问题。这时候面试官期待应聘者能够从简单问题中得到启示，从而找到解决复杂问题的窍门。比如面试官先要求应聘者写一个函数求斐波那契数

列，再问一个青蛙跳台阶的问题：一只青蛙一次可以跳上 1 级台阶，也可以跳上 2 级台阶。请问这只青蛙跳上 n 级台阶总共有多少种跳法。应聘者如果具有较强的知识迁移能力，就能分析出青蛙跳台阶问题实质上只是斐波那契数列的一个应用（详见面试题 10 “斐波那契数列”）。

还有不少面试官喜欢考查应聘者的抽象建模能力和发散思维能力。面试官从日常生活中提炼出问题，比如面试题 61 “扑克牌的顺子”，考查应聘者能不能把问题抽象出来用合理的数据结构表示，并找到其中的规律解决这个问题。面试官也可以限制应聘者不得使用常规方法，这要求应聘者具备创新精神，能够打开思路从多角度去分析、解决问题。比如在面试题 65 “不用加减乘除做加法”中，面试官期待应聘者能够打开思路，用位运算实现整数的加法。

我们将在本书的第 6 章“面试中的各项能力”中用具体的面试题详细讨论上述能力在面试中的重要作用。

1.3.3 应聘者提问环节

在结束面试前的 5~10 分钟，面试官会给应聘者机会问几个问题，应聘者的问题质量对面试的结果也有一定的影响。有些人的沟通能力很强，马上就能想到有意思的问题。但对于大多数人而言，在经受了面试官将近一小时的拷问之后可能已经精疲力竭，再迅速想出几个问题难度很大。因此建议应聘者不妨在面试之前做些功课，为每一轮面试准备 2~3 个问题，这样到提问环节的时候就游刃有余了。

面试官让应聘者问几个问题，主要是想了解他最关心的问题有哪些，因此应聘者至少要问一两个问题，否则面试官就会觉得你对我们公司、职位等都不感兴趣，那你来面试做什么？但也不是什么问题都可以在这个时候问。如果问题问得比较合适，则对应聘者来说是个加分的好机会；但如果问的问题不太合适，则面试官对他的印象就会大打折扣。

有些问题是不适合在技术面试这个环节里问的。首先，不要问和自己的职位没有关系的问题，比如问“公司未来五年的发展战略是什么”。如果应聘的职位是 CTO，而面试官是 CEO，那么这倒是个合适的问题。如果应聘的只是一线开发的职位，那这个问题离我们就太远了，与我们的切身利益没有多少关系。另外，坐在对面的面试官很有可能也只是一个在一线开发的程序员，他该怎么回答这个关系公司发展战略的问题呢？

其次，不要问薪水。技术面试不是谈薪水的时候，要谈工资要等通过面试之后和HR谈。而且这会让面试官觉得你最关心的问题就是薪水，给面试官留下的印象也不好。

再次，不要立即打听面试结果，比如问“您觉得我能拿到Offer吗”之类的问题。现在大部分公司的面试都有好几轮，最终决定应聘者能不能通过面试，是要把所有面试官的评价综合起来的。问这个问题相当于自问，因为问了面试官也不可能告诉应聘者结果，还会让面试官觉得你没有自我评估的能力。

最后，推荐问的问题是与应聘的职位或者项目相关的问题。如果这种类型的问题问得很到位，那么面试官就会觉得你对应聘的职位很有兴趣。不过要问好这种类型的问题也不容易，因为首先要对应聘的职位或者项目的背景有一定的了解。我们可以从两方面去了解相关的信息：一是面试前做足功课，到网上去搜集一些相关的信息，做到对公司成立时间、主要业务、职位要求等都了然于胸；二是面试过程中留心面试官说过的话。有不少面试官在面试之前会简单介绍与招聘职位相关的项目，其中会包含从其他渠道无法得到的信息，比如项目进展情况等。应聘者可以从中找出一两个点，然后向面试官提问。

下面的例子是笔者去思科面试时问的几个问题。一个面试官介绍项目时说这次招聘是项目组第一次在中国招人，目前这个项目所有人员都在美国总部。这轮面试笔者最后问的问题是：这个项目所有的老员工都在美国，那怎么对中国这一批新员工进行培训？中国的新员工有没有机会去美国总部学习？最后一轮面试是老板面试，她介绍说正在招聘的项目组负责开发一个测试系统，思科用它来测试供应商生产的网络设备。这轮面试笔者问的几个问题是：这个组是做系统测试的，那这个组的人员是不是也要参与网络设备的测试？是不是需要学习与硬件测试相关的知识？因为我们测试的对象是网络设备，那么这个职位对网络硬件的掌握程度有没有要求？

1.4 本章小结

本章重点介绍了面试的流程。通常面试是从电话面试开始的。接下来可能有一两轮共享桌面远程面试，面试官通过桌面共享软件远程考查应聘

者的编程和调试能力。如果应聘者的表现足够优秀，那么公司将邀请他到公司去接受现场面试。

一般每一轮面试都有 3 个环节。首先是行为面试环节，面试官在这一环节中对照简历询问应聘者的项目经验和掌握的技能。接下来就是技术面试环节，这是面试的重头戏。在这一环节里，面试官除了关注应聘者的编程能力和技术功底，还会注意考查他的沟通能力和学习能力。在面试的最后，通常面试官会留几分钟时间让应聘者问几个他感兴趣的问题。

1.3.2 节是全书的大纲，介绍了面试官关注应聘者 5 个方面的素质：基础知识是否扎实、能否写出高质量的代码、思路是否清晰、是否有优化效率的能力，以及包括学习能力、沟通能力在内的综合素质是否优秀。在接下来的第 2~6 章中，我们将一一深入探讨这 5 个方面的素质。

第2章

面试需要的基础知识

2.1

面试官谈基础知识

“C++的基础知识，如面向对象的特性、构造函数、析构函数、动态绑定等，能够反映出应聘者是否善于把握问题本质，有没有耐心深入一个问题。另外还有常用的设计模式、UML图等，这些都能体现应聘者是否有软件工程方面的经验。”

——王海波（Autodesk，软件工程师）

“对基础知识的考查我特别重视C++中对内存的使用管理。我觉得内存管理是C++程序员特别要注意的，因为内存的使用和管理会影响程序的效率和稳定性。”

——蓝诚（Autodesk，软件工程师）

“基础知识反映了一个人的基本能力和基础素质，是以后工作中最核心的能力要求。我一般考查：(1) 数据结构和算法；(2) 编程能力；(3) 部分数学知识，如概率；(4) 问题的分析和推理能力。”

——张晓禹（百度，技术经理）

“我比较重视四块基础知识：(1) 编程基本功（特别喜欢字符串处理这一类的问题）；(2) 并发控制；(3) 算法、复杂度；(4) 语言的基本概念。”

——张珺（百度，高级软件工程师）

“我会考查编程基础、计算机系统基础知识、算法及设计能力。这些是成为一个软件工程师的最基本的要求，这些方面表现出色的人，我们一般认为是有发展潜力的。”

——韩伟东（盛大，高级研究员）

“(1) 对 OS 的理解程度。这些知识对于工作中常遇到的内存管理、文件操作、程序性能、多线程、程序安全等有重要帮助。对于 OS 理解比较深入的人对于偏底层的工作上手一般比较快。(2) 对于一门编程语言的掌握程度。一个热爱编程的人应该会对某种语言有比较深入的了解。通常这样的人对于新的编程语言上手也比较快，而且理解比较深入。(3) 常用的算法和数据结构。不了解这些的程序员基本只能写写‘Hello World’。”

——陈黎明（微软，SDE II）

2.2 编程语言

程序员写代码总是基于某一种编程语言，因此技术面试的时候都会直接或者间接涉及至少一种编程语言。在面试过程中，面试官要么直接问语言的语法，要么让应聘者用一种编程语言写代码解决一个问题，通过写出的代码来判断应聘者对他使用的语言的掌握程度。现在流行的编程语言很多，不同公司开发用的语言也不尽相同。做底层开发比如经常写驱动的人更习惯用 C，Linux 下有很多程序员用 C++ 开发应用程序，基于 Windows 的 C# 项目已经越来越多，跨平台开发的程序员则可能更喜欢 Java，随着苹果 iPad、iPhone 的热销已经有很多程序员投向了 Objective C 的阵营，同时还有很多人喜欢用脚本语言如 Perl、Python 开发短小精致的小应用软件。

因此，不同公司面试的时候对编程语言的要求也有所不同。每一种编程语言都可以写出一本大部头的书籍，本书限于篇幅不可能面面俱到。本书中所有代码都用 C/C++/C#实现，下面简要介绍一些 C++/C#常见的面试题。

2.2.1 C++

国内绝大部分高校都开设 C++的课程，因此绝大部分程序员都学过 C++，于是 C++成了各公司面试的首选编程语言。包括 Autodesk 在内的很多公司在面试的时候会有大量的 C++的语法题，其他公司虽然不直接面试 C++的语法，但面试题要求用 C++实现算法。因此，总的来说，应聘者不管去什么公司求职，都应该在一定程度上掌握 C++。

通常语言面试有 3 种类型。第一种题型是面试官直接询问应聘者对 C++概念的理解。这种类型的问题，面试官特别喜欢了解应聘者对 C++关键字的理解程度。例如，在 C++中，有哪 4 个与类型转换相关的关键字？这些关键字各有什么特点，应该在什么场合下使用？

在这种类型的题目中，`sizeof` 是经常被问到的一个概念。比如下面的面试片段，就反复出现在各公司的技术面试中。

面试官：定义一个空的类型，里面没有任何成员变量和成员函数。对该类型求 `sizeof`，得到的结果是多少？

应聘者：答案是 1。

面试官：为什么不是 0？

应聘者：空类型的实例中不包含任何信息，本来求 `sizeof` 应该是 0，但是当我们声明该类型的实例的时候，它必须在内存中占有一定的空间，否则无法使用这些实例。至于占用多少内存，由编译器决定。在 Visual Studio 中，每个空类型的实例占用 1 字节的空间。

面试官：如果在该类型中添加一个构造函数和析构函数，再对该类型求 `sizeof`，得到的结果又是多少？

应聘者：和前面一样，还是 1。调用构造函数和析构函数只需要知道函数的地址即可，而这些函数的地址只与类型相关，而与类型的实例无关，编译器也不会因为这两个函数而在实例内添加任何额外的信息。

面试官：那如果把析构函数标记为虚函数呢？

应聘者：C++的编译器一旦发现一个类型中有虚函数，就会为该类型生成虚函数表，并在该类型的每一个实例中添加一个指向虚函数表的指针。在32位的机器上，一个指针占4字节的空间，因此求sizeof得到4；如果是64位的机器，则一个指针占8字节的空间，因此求sizeof得到8。

第二种题型就是面试官拿出事先准备好的代码，让应聘者分析代码的运行结果。这种题型选择的代码通常包含比较复杂微妙的语言特性，这要求应聘者对C++考点有着透彻的理解。即使应聘者对考点有一点点模糊，那么最终他得到的结果和实际运行的结果可能就会差距甚远。

比如，面试官递给应聘者一张有如下代码的A4打印纸要求他分析编译运行的结果，并提供3个选项：A. 编译错误；B. 编译成功，运行时程序崩溃；C. 编译运行正常，输出10。

```
class A
{
private:
    int value;

public:
    A(int n) { value = n; }
    A(A other) { value = other.value; }

    void Print() { std::cout << value << std::endl; }
};

int _tmain(int argc, _TCHAR* argv[])
{
    A a = 10;
    A b = a;
    b.Print();

    return 0;
}
```

在上述代码中，复制构造函数A(A other)传入的参数是A的一个实例。由于是传值参数，我们把形参复制到实参会调用复制构造函数。因此，如果允许复制构造函数传值，就会在复制构造函数内调用复制构造函数，就会形成永无休止的递归调用从而导致栈溢出。因此，C++的标准不允许复制构造函数传值参数，在Visual Studio和GCC中，都将编译出错。要解决这个问题，我们可以把构造函数修改为A(const A& other)，也就是把传值参数改成常量引用。

第三种题型就是要求应聘者写代码定义一个类型或者实现类型中的成员函数。让应聘者写代码的难度自然比让应聘者分析代码要高不少，因

为能想明白的未必就能写得清楚。很多考查 C++语法的代码题重点考查构造函数、析构函数及运算符重载，比如面试题 1 “赋值运算符函数”就是一个例子。

为了让大家能顺利地通过 C++面试，更重要的是能更好地学习掌握 C++ 这门编程语言，这里推荐几本 C++ 的书，大家可以根据自己的具体情况选择阅读的顺序。

- 《Effective C++》：这本书很适合在面试之前突击 C++。这本书列举了使用 C++ 经常出现的问题及解决这些问题的技巧。该书中提到的问题也是面试官很喜欢问的问题。
- 《C++ Primer》：读完这本书，就会对 C++ 的语法有全面的了解。
- 《深度探索 C++ 对象模型》：这本书有助于我们深入了解 C++ 对象的内部。读懂这本书后，很多 C++ 难题，比如前面的 sizeof 的问题、虚函数的调用机制等，都会变得很容易。
- 《The C++ Programming Language》：如果想全面深入掌握 C++，那么没有哪本书比这本书更适合的了。

面试题 1：赋值运算符函数

题目：如下为类型 CMyString 的声明，请为该类型添加赋值运算符函数。

```
class CMyString
{
public:
    CMyString(char* pData = nullptr);
    CMyString(const CMyString& str);
    ~CMyString(void);

private:
    char* m_pData;
};
```

当面试官要求应聘者定义一个赋值运算符函数时，他会在检查应聘者写出的代码时关注如下几点：

- 是否把返回值的类型声明为该类型的引用，并在函数结束前返回实例自身的引用 (*this)。只有返回一个引用，才可以允许连续赋值。否则，如果函数的返回值是 void，则应用该赋值运算符将不能进行连续赋值。假设有 3 个 CMyString 的对象：str1、str2 和 str3，在程

序中语句 str1=str2=str3 将不能通过编译。

- 是否把传入的参数的类型声明为常量引用。如果传入的参数不是引用而是实例，那么从形参到实参会调用一次复制构造函数。把参数声明为引用可以避免这样的无谓消耗，能提高代码的效率。同时，我们在赋值运算符函数内不会改变传入的实例的状态，因此应该为传入的引用参数加上 `const` 关键字。
- 是否释放实例自身已有的内存。如果我们忘记在分配新内存之前释放自身已有的空间，则程序将出现内存泄漏。
- 判断传入的参数和当前的实例 (`*this`) 是不是同一个实例。如果是同一个，则不进行赋值操作，直接返回。如果事先不判断就进行赋值，那么在释放实例自身内存的时候就会导致严重的问题：当`*this` 和传入的参数是同一个实例时，一旦释放了自身的内存，传入的参数的内存也同时被释放了，因此再也找不到需要赋值的内容了。

❖ 经典的解法，适用于初级程序员

当我们完整地考虑了上述 4 个方面之后，可以写出如下的代码：

```
CMyString& CMyString::operator =(const CMyString &str)
{
    if(this == &str)
        return *this;

    delete []m_pData;
    m_pData = nullptr;

    m_pData = new char[strlen(str.m_pData) + 1];
    strcpy(m_pData, str.m_pData);

    return *this;
}
```

这是一般 C++教材上提供的参考代码。如果接受面试的是应届毕业生或者 C++初级程序员，能全面地考虑到前面 4 点并完整地写出代码，那么面试官可能会让他通过这轮面试。但如果面试的是 C++高级程序员，则面试官可能会提出更高的要求。

❖ 考虑异常安全性的解法，高级程序员必备

在前面的函数中，我们在分配内存之前先用 `delete` 释放了实例 `m_pData`

的内存。如果此时内存不足导致 new char 抛出异常，则 m_pData 将是一个空指针，这样非常容易导致程序崩溃。也就是说，一旦在赋值运算符函数内部抛出一个异常，CMyString 的实例不再保持有效的状态，这就违背了异常安全性（Exception Safety）原则。

要想在赋值运算符函数中实现异常安全性，我们有两种方法。一种简单的办法是我们先用 new 分配新内容，再用 delete 释放已有的内容。这样只在分配内容成功之后再释放原来的内容，也就是当分配内存失败时我们能确保 CMyString 的实例不会被修改。我们还有一种更好的办法，即先创建一个临时实例，再交换临时实例和原来的实例。下面是这种思路的参考代码：

```
CMyString& CMyString::operator =(const CMyString &str)
{
    if(this != &str)
    {
        CMyString strTemp(str);

        char* pTemp = strTemp.m_pData;
        strTemp.m_pData = m_pData;
        m_pData = pTemp;
    }

    return *this;
}
```

在这个函数中，我们先创建一个临时实例 strTemp，接着把 strTemp.m_pData 和实例自身的 m_pData 进行交换。由于 strTemp 是一个局部变量，但程序运行到 if 的外面时也就出了该变量的作用域，就会自动调用 strTemp 的析构函数，把 strTemp.m_pData 所指向的内存释放掉。由于 strTemp.m_pData 指向的内存就是实例之前 m_pData 的内存，这就相当于自动调用析构函数释放实例的内存。

在新的代码中，我们在 CMyString 的构造函数里用 new 分配内存。如果由于内存不足抛出诸如 bad_alloc 等异常，但我们还没有修改原来实例的状态，因此实例的状态还是有效的，这也就保证了异常安全性。

如果应聘者在面试的时候能够考虑到这个层面，面试官就会觉得他对代码的异常安全性有很深的理解，那么他自然也就能通过这轮面试了。



源代码：

[本题完整的源代码](#)：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/01_AssignmentOperator



测试用例：

- 把一个 CMyString 的实例赋值给另外一个实例。
- 把一个 CMyString 的实例赋值给它自己。
- 连续赋值。



本题考点：

- 考查应聘者对 C++基础语法的理解，如运算符函数、常量引用等。
- 考查应聘者对内存泄漏的理解。
- 对于高级 C++程序员，面试官还将考查应聘者对代码异常安全性的理解。

2.2.2 C#

C#是微软在推出新的开发平台.NET时同步推出的编程语言。由于 Windows至今仍然是用户最多的操作系统，而.NET又是微软近年来力推的开发平台，因此C#无论是在桌面软件还是在网络应用的开发上都有着广泛的应用，所以我们也不难理解为什么现在很多基于Windows系统开发的公司都会要求应聘者掌握C#。

C#可以看成一门以C++为基础发展起来的托管语言，因此它的很多关键字甚至语法都和C++类似。对于一个学习过C++编程的程序员而言，他用不了多长时间的学习就能用C#来开发软件。然而我们也要清醒地认识到，虽然学习C#与C++相同或者类似的部分很容易，但要掌握并区分两者不同的地方却不是一件很容易的事情。面试官总是喜欢深究我们模棱两可的地方以考查我们是不是真的理解了，因此我们要着重注意C#与C++不同的语法特点。下面的面试片段就是一个例子。

面试官：在C++中可以用struct和class来定义类型。这两种类型有什么区别？

应聘者：如果没有标明成员函数或者成员变量的访问权限级别，那么在 struct 中默认的是 public，而在 class 中默认的是 private。

面试官：那在 C# 中呢？

应聘者：C# 和 C++ 不一样。在 C# 中如果没有标明成员函数或者成员变量的访问权限级别，则在 struct 和 class 中都是 private 的。struct 和 class 的区别是 struct 定义的是值类型，值类型的实例在栈上分配内存；而 class 定义的是引用类型，引用类型的实例在堆上分配内存。

和 C++一样，在 C# 中，每个类型中都有构造函数。但和 C++ 不同的是，我们在 C# 中可以为类型定义一个 Finalizer 和 Dispose 方法以释放资源。Finalizer 方法虽然写法与 C++ 的析构函数看起来一样，都是~后面跟类型名字，但与 C++ 析构函数的调用时机是确定的不同，C# 的 Finalizer 是在运行时（CLR）进行垃圾回收时才会被调用的，它的调用时机是由运行时决定的，因此对程序员来说是不确定的。另外，在 C# 中可以为类型定义一个特殊的构造函数：静态构造函数。这个函数的特点是在类型第一次被使用之前由运行时自动调用，而且保证只调用一次。关于静态构造函数，我们有很多有意思的面试题，比如运行下面的 C# 代码，输出的结果是什么？

```
class A
{
    public A(string text)
    {
        Console.WriteLine(text);
    }
}

class B
{
    static A a1 = new A("a1");
    A a2 = new A("a2");

    static B()
    {
        a1 = new A("a3");
    }

    public B()
    {
        a2 = new A("a4");
    }
}

class Program
{
    static void Main(string[] args)
```

```

    {
        B b = new B();
    }
}

```

在调用类型 B 的代码之前先执行 B 的静态构造函数。静态构造函数先初始化类型的静态变量，再执行函数体内的语句。因此，先打印 a1 再打印 a3。接下来执行 B b = new B()，即调用 B 的普通构造函数。构造函数先初始化成员变量，再执行函数体内的语句，因此先后打印出 a2、a4。因此，运行上面的代码，得到的结果将是打印出 4 行，分别是 a1、a3、a2、a4。

我们除了要关注 C# 和 C++ 不同的知识点，还要格外关注 C# 一些特有的功能，比如反射、应用程序域（AppDomain）等。这些概念还相互关联，要花很多时间学习研究才能透彻地理解它们。下面就是一段关于反射和应用程序域的代码，运行它得到的结果是什么？

```

[Serializable]
internal class A : MarshalByRefObject
{
    public static int Number;

    public void SetNumber(int value)
    {
        Number = value;
    }
}

[Serializable]
internal class B
{
    public static int Number;

    public void SetNumber(int value)
    {
        Number = value;
    }
}

class Program
{
    static void Main(string[] args)
    {
        String assamby = Assembly.GetEntryAssembly().FullName;
        AppDomain domain = AppDomain.CreateDomain("NewDomain");

        A.Number = 10;
        String nameOfA = typeof(A).FullName;
        A a = domain.CreateInstanceAndUnwrap(assamby, nameOfA) as A;
        a.SetNumber(20);
        Console.WriteLine("Number in class A is {0}", A.Number);
    }
}

```

```

        B.Number = 10;
        String nameOfB = typeof(B).FullName;
        B b = domain.CreateInstanceAndUnwrap(assambly, nameOfB) as B;
        b.SetNumber(20);
        Console.WriteLine("Number in class B is {0}", B.Number);
    }
}

```

上述 C# 代码先创建一个名为 NewDomain 的应用程序域，并在该域中利用反射机制创建类型 A 的一个实例和类型 B 的一个实例。我们注意到类型 A 继承自 MarshalByRefObject，而 B 不是。虽然这两个类型的结构一样，但由于基类不同而导致在跨越应用程序域的边界时表现出的行为将大不相同。

先考虑 A 的情况。由于 A 继承自 MarshalByRefObject，那么 a 实际上只是在默认的域中的一个代理实例（Proxy），它指向位于 NewDomain 域中的 A 的一个实例。当调用 a 的方法 SetNumber 时，是在 NewDomain 域中调用该方法的，它将修改 NewDomain 域中静态变量 A.Number 的值并设为 20。由于静态变量在每个应用程序域中都有一份独立的拷贝，修改 NewDomain 域中的静态变量 A.Number 对默认域中的静态变量 A.Number 没有任何影响。由于 Console.WriteLine 是在默认的应用程序域中输出 A.Number，因此输出仍然是 10。

接着讨论 B。由于 B 只是从 Object 继承而来的类型，它的实例穿越应用程序域的边界时，将会完整地复制实例。因此，在上述代码中，我们尽管试图在 NewDomain 域中生成 B 的实例，但会把实例 b 复制到默认的应用程序域。此时调用方法 b.SetNumber 也是在默认的应用程序域上进行，它将修改默认的域上的 B.Number 并设为 20。再在默认的域上调用 Console.WriteLine 时，它将输出 20。

下面推荐两本与 C# 相关的书籍，以方便大家应对 C# 面试并学习好 C#。

- **《Professional C#》：**这本书最大的特点是在附录中有几章专门写给已经有其他语言（如 VB、C++ 和 Java）经验的程序员，它详细讲述了 C# 和其他语言的区别，看了这几章之后就不会把 C# 和之前掌握的语言相混淆。
- Jeffrey Richter 的 **《CLR Via C#》：**该书不仅深入地介绍了 C# 语言，

同时对 CLR 及.NET 进行了全面的剖析。如果能够读懂这本书，那么我们就能深入理解装箱卸箱、垃圾回收、反射等概念，知其然的同时也能知其所以然，通过与 C#相关的面试自然也就不难了。

面试题 2：实现 Singleton 模式

题目：设计一个类，我们只能生成该类的一个实例。

只能生成一个实例的类是实现了 Singleton（单例）模式的类型。由于设计模式在面向对象程序设计中起着举足轻重的作用，在面试过程中很多公司都喜欢问一些与设计模式相关的问题。在常用的模式中，Singleton 是唯一一个能够用短短几十行代码完整实现的模式。因此，写一个 Singleton 的类型是一个很常见的面试题。

❖ 不好的解法一：只适用于单线程环境

由于要求只能生成一个实例，因此我们必须把构造函数设为私有函数以禁止他人创建实例。我们可以定义一个静态的实例，在需要的时候创建该实例。下面定义类型 Singleton1 就是基于这个思路的实现：

```
public sealed class Singleton1
{
    private Singleton1()
    {
    }

    private static Singleton1 instance = null;
    public static Singleton1 Instance
    {
        get
        {
            if (instance == null)
                instance = new Singleton1();

            return instance;
        }
    }
}
```

上述代码在 Singleton1 的静态属性 Instance 中，只有在 instance 为 null 的时候才创建一个实例以避免重复创建。同时我们把构造函数定义为私有函数，这样就能确保只创建一个实例。

❖ 不好的解法二：虽然在多线程环境中能工作，但效率不高

解法一中的代码在单线程的时候工作正常，但在多线程的情况下就有问题了。设想如果两个线程同时运行到判断 `instance` 是否为 `null` 的 `if` 语句，并且 `instance` 的确没有创建时，那么两个线程都会创建一个实例，此时类型 `Singleton1` 就不再满足单例模式的要求了。为了保证在多线程环境下我们还是只能得到类型的一个实例，需要加上一个同步锁。把 `Singleton1` 稍作修改得到了如下代码：

```
public sealed class Singleton2
{
    private Singleton2()
    {
    }

    private static readonly object syncObj = new object();

    private static Singleton2 instance = null;
    public static Singleton2 Instance
    {
        get
        {
            lock (syncObj)
            {
                if (instance == null)
                    instance = new Singleton2();
            }
            return instance;
        }
    }
}
```

我们还是假设有两个线程同时想创建一个实例。由于在一个时刻只有一个线程能得到同步锁，当第一个线程加上锁时，第二个线程只能等待。当第一个线程发现实例还没有创建时，它创建出一个实例。接着第一个线程释放同步锁，此时第二个线程可以加上同步锁，并运行接下来的代码。这时候由于实例已经被第一个线程创建出来了，第二个线程就不会重复创建实例了，这样就保证了我们在多线程环境中也只能得到一个实例。

但是类型 `Singleton2` 还不是很完美。我们每次通过属性 `Instance` 得到 `Singleton2` 的实例，都会试图加上一个同步锁，而加锁是一个非常耗时的操作，在没有必要的时候我们应该尽量避免。

❖ 可行的解法：加同步锁前后两次判断实例是否已存在

我们只是在实例还没有创建之前需要加锁操作，以保证只有一个线程创建出实例。而当实例已经创建之后，我们已经不需要再执行加锁操作了。于是我们可以把解法二中的代码再做进一步的改进：

```
public sealed class Singleton3
{
    private Singleton3()
    {
    }

    private static object syncObj = new object();

    private static Singleton3 instance = null;
    public static Singleton3 Instance
    {
        get
        {
            if (instance == null)
            {
                lock (syncObj)
                {
                    if (instance == null)
                        instance = new Singleton3();
                }
            }
            return instance;
        }
    }
}
```

Singleton3 中只有当 `instance` 为 `null` 即没有创建时，需要加锁操作。当 `instance` 已经创建出来之后，则无须加锁。因为只在第一次的时候 `instance` 为 `null`，因此只在第一次试图创建实例的时候需要加锁。这样 Singleton3 的时间效率比 Singleton2 要好很多。

Singleton3 用加锁机制来确保在多线程环境下只创建一个实例，并且用两个 `if` 判断来提高效率。这样的代码实现起来比较复杂，容易出错，我们还有更加优秀的解法。

❖ 强烈推荐的解法一：利用静态构造函数

C#的语法中有一个函数能够确保只调用一次，那就是静态构造函数，我们可以利用 C#的这个特性实现单例模式。

```

public sealed class Singleton4
{
    private Singleton4()
    {
    }

    private static Singleton4 instance = new Singleton4();
    public static Singleton4 Instance
    {
        get
        {
            return instance;
        }
    }
}

```

Singleton4 的实现代码非常简洁。我们在初始化静态变量 instance 的时候创建一个实例。由于 C#是在调用静态构造函数时初始化静态变量，.NET 运行时能够确保只调用一次静态构造函数，这样我们就能够保证只初始化一次 instance。

C#中调用静态构造函数的时机不是由程序员掌控的，而是当.NET 运行时发现第一次使用一个类型的时候自动调用该类型的静态构造函数。因此在 Singleton4 中，实例 instance 并不是在第一次调用属性 Singleton4.Instance 的时候被创建的，而是在第一次用到 Singleton4 的时候就会被创建。假设我们在 Singleton4 中添加一个静态方法，调用该静态函数是不需要创建一个实例的，但如果按照 Singleton4 的方式实现单例模式，则仍然会过早地创建实例，从而降低内存的使用效率。

❖ 强烈推荐的解法二：实现按需创建实例

最后一个实现 Singleton5 则很好地解决了 Singleton4 中的实例创建时机过早的问题。

```

public sealed class Singleton5
{
    Singleton5()
    {
    }

    public static Singleton5 Instance
    {
        get
        {
            return Nested.instance;
        }
    }
}

```

```

class Nested
{
    static Nested()
    {
    }

    internal static readonly Singleton5 instance = new Singleton5();
}

```

在上述 Singleton5 的代码中，我们在内部定义了一个私有类型 Nested。当第一次用到这个嵌套类型的时候，会调用静态构造函数创建 Singleton5 的实例 instance。类型 Nested 只在属性 Singleton5.Instance 中被用到，由于其私有属性，他人无法使用 Nested 类型。因此，当我们第一次试图通过属性 Singleton5.Instance 得到 Singleton5 的实例时，会自动调用 Nested 的静态构造函数创建实例 instance。如果我们不调用属性 Singleton5.Instance，就不会触发.NET 运行时调用 Nested，也不会创建实例，这样就真正做到了按需创建。

❖ 解法比较

在前面的 5 种实现单例模式的方法中，第一种方法在多线程环境中不能正常工作，第二种模式虽然能在多线程环境中正常工作，但时间效率很低，都不是面试官期待的解法。在第三种方法中，我们通过两次判断一次加锁确保在多线程环境中能高效率地工作。第四种方法利用 C# 的静态构造函数的特性，确保只创建一个实例。第五种方法利用私有嵌套类型的特性，做到只在真正需要的时候才会创建实例，提高空间使用效率。如果在面试中给出第四种或者第五种解法，则毫无疑问会得到面试官的青睐。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/02_Singleton



本题考点：

- 考查应聘者对单例（Singleton）模式的理解。

- 考查应聘者对 C# 基础语法的理解，如静态构造函数等。
- 考查应聘者对多线程编程的理解。



本题扩展：

在前面的代码中，5 种单例模式的实现把类型标记为 sealed，表示它们不能作为其他类型的基类。现在我们要求定义一个表示总统的类型 President，可以从该类型继承出 FrenchPresident 和 AmericanPresident 等类型。这些派生类型都只能产生一个实例。请问该如何设计实现这些类型？

2.3

数据结构

数据结构一直是技术面试的重点，大多数面试题都是围绕着数组、字符串、链表、树、栈及队列这几种常见的数据结构展开的，因此每一个应聘者都要熟练掌握这几种数据结构。

数组和字符串是两种最基本的数据结构，它们用连续内存分别存储数字和字符。链表和树是面试中出现频率最高的数据结构。由于操作链表和树需要操作大量的指针，应聘者在解决相关问题的时候一定要留意代码的鲁棒性，否则容易出现程序崩溃的问题。栈是一个与递归紧密相关的数据结构，同样队列也与广度优先遍历算法紧密相关，深刻理解这两种数据结构能帮助我们解决很多算法问题。

2.3.1 数组

数组可以说是最简单的一种数据结构，它占据一块连续的内存并按照顺序存储数据。创建数组时，我们需要首先指定数组的容量大小，然后根据大小分配内存。即使我们只在数组中存储一个数字，也需要为所有的数据预先分配内存。因此数组的空间效率不是很好，经常会有空闲的区域没有得到充分利用。

由于数组中的内存是连续的，于是可以根据下标在 $O(1)$ 时间读/写任何元素，因此它的时间效率是很高的。我们可以根据数组时间效率高的优点，用数组来实现简单的哈希表：把数组的下标设为哈希表的键值（Key），而

把数组中的每一个数字设为哈希表的值（Value），这样每一个下标及数组中该下标对应的数字就组成了一个“键值-值”的配对。有了这样的哈希表，我们就可以在 $O(1)$ 时间内实现查找，从而快速、高效地解决很多问题。面试题 50 “第一个只出现一次的字符”就是一个很好的例子。

为了解决数组空间效率不高的问题，人们又设计实现了多种动态数组，比如 C++ 的 STL 中的 vector。为了避免浪费，我们先为数组开辟较小的空间，然后往数组中添加数据。当数据的数目超过数组的容量时，我们再重新分配一块更大的空间（STL 的 vector 每次扩充容量时，新的容量都是前一次的两倍），把之前的数据复制到新的数组中，再把之前的内存释放，这样就能减少内存的浪费。但我们也注意到每一次扩充数组容量时都有大量的额外操作，这对时间性能有负面影响，因此使用动态数组时要尽量减少改变数组容量大小的次数。

在 C/C++ 中，数组和指针是既相互关联又有区别的两个概念。当我们声明一个数组时，其数组的名字也是一个指针，该指针指向数组的第一个元素。我们可以用一个指针来访问数组。但值得注意的是，C/C++ 没有记录数组的大小，因此在用指针访问数组中的元素时，程序员要确保没有超出数组的边界。下面通过一个例子来了解数组和指针的区别。运行下面的代码，请问输出是什么？

```
int GetSize(int data[])
{
    return sizeof(data);
}

int _tmain(int argc, _TCHAR* argv[])
{
    int data1[] = {1, 2, 3, 4, 5};
    int size1 = sizeof(data1);

    int* data2 = data1;
    int size2 = sizeof(data2);

    int size3 = GetSize(data1);

    printf("%d, %d, %d", size1, size2, size3);
}
```

答案是输出“20, 4, 4”。data1 是一个数组，`sizeof(data1)` 是求数组的大小。这个数组包含 5 个整数，每个整数占 4 字节，因此共占用 20 字节。`data2` 声明为指针，尽管它指向了数组 `data1` 的第一个数字，但它的本质仍然是一个指针。在 32 位系统上，对任意指针求 `sizeof`，得到的结果都是 4。在 C/C++

中，当数组作为函数的参数进行传递时，数组就自动退化为同类型的指针。因此，尽管函数 GetSize 的参数 data 被声明为数组，但它会退化为指针，size3 的结果仍然是 4。

面试题 3：数组中重复的数字

题目一：找出数组中重复的数字。

在一个长度为 n 的数组里的所有数字都在 $0 \sim n-1$ 的范围内。数组中某些数字是重复的，但不知道有几个数字重复了，也不知道每个数字重复了几次。请找出数组中任意一个重复的数字。例如，如果输入长度为 7 的数组 {2, 3, 1, 0, 2, 5, 3}，那么对应的输出是重复的数字 2 或者 3。

解决这个问题的一个简单的方法是先把输入的数组排序。从排序的数组中找出重复的数字是一件很容易的事情，只需要从头到尾扫描排序后的数组就可以了。排序一个长度为 n 的数组需要 $O(n \log n)$ 的时间。

还可以利用哈希表来解决这个问题。从头到尾按顺序扫描数组的每个数字，每扫描到一个数字的时候，都可以用 $O(1)$ 的时间来判断哈希表里是否已经包含了该数字。如果哈希表里还没有这个数字，就把它加入哈希表。如果哈希表里已经存在该数字，就找到一个重复的数字。这个算法的时间复杂度是 $O(n)$ ，但它提高时间效率是以一个大小为 $O(n)$ 的哈希表为代价的。我们再看看有没有空间复杂度是 $O(1)$ 的算法。

我们注意到数组中的数字都在 $0 \sim n-1$ 的范围内。如果这个数组中没有重复的数字，那么当数组排序之后数字 i 将出现在下标为 i 的位置。由于数组中有重复的数字，有些位置可能存在多个数字，同时有些位置可能没有数字。

现在让我们重排这个数组。从头到尾依次扫描这个数组中的每个数字。当扫描到下标为 i 的数字时，首先比较这个数字（用 m 表示）是不是等于 i 。如果是，则接着扫描下一个数字；如果不是，则再拿它和第 m 个数字进行比较。如果它和第 m 个数字相等，就找到了一个重复的数字（该数字在下标为 i 和 m 的位置都出现了）；如果它和第 m 个数字不相等，就把第 i 个数字和第 m 个数字交换，把 m 放到属于它的位置。接下来再重复这个比较、交换的过程，直到我们发现一个重复的数字。

以数组 {2, 3, 1, 0, 2, 5, 3} 为例来分析找到重复数字的步骤。数组的第 0

个数字（从0开始计数，和数组的下标保持一致）是2，与它的下标不相等，于是把它和下标为2的数字1交换。交换之后的数组是{1, 3, 2, 0, 2, 5, 3}。此时第0个数字是1，仍然与它的下标不相等，继续把它和下标为1的数字3交换，得到数组{3, 1, 2, 0, 2, 5, 3}。接下来继续交换第0个数字3和第3个数字0，得到数组{0, 1, 2, 3, 2, 5, 3}。此时第0个数字的数值为0，接着扫描下一个数字。在接下来的几个数字中，下标为1、2、3的3个数字分别为1、2、3，它们的下标和数值都分别相等，因此不需要执行任何操作。接下来扫描到下标为4的数字2。由于它的数值与它的下标不相等，再比较它和下标为2的数字。注意到此时数组中下标为2的数字也是2，也就是数字2在下标为2和下标为4的两个位置都出现了，因此找到一个重复的数字。

上述思路可以用如下代码实现：

```
bool duplicate(int numbers[], int length, int* duplication)
{
    if(numbers == nullptr || length <= 0)
    {
        return false;
    }

    for(int i = 0; i < length; ++i)
    {
        if(numbers[i] < 0 || numbers[i] > length - 1)
            return false;
    }

    for(int i = 0; i < length; ++i)
    {
        while(numbers[i] != i)
        {
            if(numbers[i] == numbers[numbers[i]])
            {
                *duplication = numbers[i];
                return true;
            }

            // swap numbers[i] and numbers[numbers[i]]
            int temp = numbers[i];
            numbers[i] = numbers[temp];
            numbers[temp] = temp;
        }
    }

    return false;
}
```

在上述代码中，找到的重复数字通过参数 `duplication` 传给函数的调用

者，而函数的返回值表示数组中是否有重复的数字。当输入的数组中存在重复的数字时，返回 true；否则返回 false。

代码中尽管有一个两重循环，但每个数字最多只要交换两次就能找到属于它自己的位置，因此总的时间复杂度是 $O(n)$ 。另外，所有的操作步骤都是在输入数组上进行的，不需要额外分配内存，因此空间复杂度为 $O(1)$ 。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/03_01_DuplicationInArray



测试用例：

- 长度为 n 的数组里包含一个或多个重复的数字。
- 数组中不包含重复的数字。
- 无效输入测试用例（输入空指针；长度为 n 的数组中包含 $0 \sim n-1$ 之外的数字）。



本题考点：

- 考查应聘者对一维数组的理解及编程能力。一维数组在内存中占据连续的空间，因此我们可以根据下标定位对应的元素。
- 考查应聘者分析问题的能力。当应聘者发现问题比较复杂时，能不能通过具体的例子找出其中的规律，是能否解决这个问题的关键所在。

题目二：不修改数组找出重复的数字。

在一个长度为 $n+1$ 的数组里的所有数字都在 $1 \sim n$ 的范围内，所以数组中至少有一个数字是重复的。请找出数组中任意一个重复的数字，但不能修改输入的数组。例如，如果输入长度为 8 的数组 {2, 3, 5, 4, 3, 2, 6, 7}，那么对应的输出是重复的数字 2 或者 3。

这一题看起来和上面的面试题类似。由于题目要求不能修改输入的数组，我们可以创建一个长度为 $n+1$ 的辅助数组，然后逐一把原数组的每个数字复制到辅助数组。如果原数组中被复制的数字是 m ，则把它复制到辅助数组中下标为 m 的位置。这样很容易就能发现哪个数字是重复的。由于需要创建一个数组，该方案需要 $O(n)$ 的辅助空间。

接下来我们尝试避免使用 $O(n)$ 的辅助空间。为什么数组中会有重复的数字？假如没有重复的数字，那么在从 $1 \sim n$ 的范围里只有 n 个数字。由于数组里包含超过 n 个数字，所以一定包含了重复的数字。看起来在某范围里数字的个数对解决这个问题很重要。

我们把从 $1 \sim n$ 的数字从中间的数字 m 分为两部分，前面一半为 $1 \sim m$ ，后面一半为 $m+1 \sim n$ 。如果 $1 \sim m$ 的数字的数目超过 m ，那么这一半的区间里一定包含重复的数字；否则，另一半 $m+1 \sim n$ 的区间里一定包含重复的数字。我们可以继续把包含重复数字的区间一分为二，直到找到一个重复的数字。这个过程和二分查找算法很类似，只是多了一步统计区间里数字的数目。

我们以长度为 8 的数组 {2, 3, 5, 4, 3, 2, 6, 7} 为例分析查找的过程。根据题目要求，这个长度为 8 的所有数字都在 $1 \sim 7$ 的范围内。中间的数字 4 把 $1 \sim 7$ 的范围分为两段，一段是 $1 \sim 4$ ，另一段是 $5 \sim 7$ 。接下来我们统计 $1 \sim 4$ 这 4 个数字在数组中出现的次数，它们一共出现了 5 次，因此这 4 个数字中一定有重复的数字。

接下来我们再把 $1 \sim 4$ 的范围一分为二，一段是 1、2 两个数字，另一段是 3、4 两个数字。数字 1 或者 2 在数组中一共出现了两次。我们再统计数字 3 或者 4 在数组中出现的次数，它们一共出现了三次。这意味着 3、4 两个数字中一定有一个重复了。我们再分别统计这两个数字在数组中出现的次数。接着我们发现数字 3 出现了两次，是一个重复的数字。

上述思路可以用如下代码实现：

```
int getDuplication(const int* numbers, int length)
{
    if(numbers == nullptr || length <= 0)
        return -1;

    int start = 1;
    int end = length - 1;
    while(end >= start)
    {
        int middle = ((end - start) >> 1) + start;
```

```

int count = countRange(numbers, length, start, middle);
if(end == start)
{
    if(count > 1)
        return start;
    else
        break;
}

if(count > (middle - start + 1))
    end = middle;
else
    start = middle + 1;
}
return -1;
}

int countRange(const int* numbers, int length, int start, int end)
{
    if(numbers == nullptr)
        return 0;

    int count = 0;
    for(int i = 0; i < length; i++)
        if(numbers[i] >= start && numbers[i] <= end)
            ++count;
    return count;
}

```

上述代码按照二分查找的思路，如果输入长度为 n 的数组，那么函数 `countRange` 将被调用 $O(\log n)$ 次，每次需要 $O(n)$ 的时间，因此总的时间复杂度是 $O(n \log n)$ ，空间复杂度为 $O(1)$ 。和最前面提到的需要 $O(n)$ 的辅助空间的算法相比，这种算法相当于以时间换空间。

需要指出的是，这种算法不能保证找出所有重复的数字。例如，该算法不能找出数组 {2, 3, 5, 4, 3, 2, 6, 7} 中重复的数字 2。这是因为在 1~2 的范围里有 1 和 2 两个数字，这个范围的数字也出现 2 次，此时我们用该算法不能确定是每个数字各出现一次还是某个数字出现了两次。

从上述分析中我们可以看出，如果面试官提出不同的功能要求（找出任意一个重复的数字、找出所有重复的数字）或者性能要求（时间效率优先、空间效率优先），那么我们最终选取的算法也将不同。这也说明在面试中和面试官交流的重要性，我们一定要在动手写代码之前弄清楚面试官的需求。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/03_02_DuplicationInArrayNoEdit



测试用例：

- 长度为 n 的数组里包含一个或多个重复的数字。
- 数组中不包含重复的数字。
- 无效输入测试用例（输入空指针）。



本题考点：

- 考查应聘者对一维数组的理解及编程能力。一维数组在内存中占据连续的空间，因此我们可以根据下标定位对应的元素。
- 考查应聘者对二分查找算法的理解，并能快速、正确地实现二分查找算法的代码。
- 考查应聘者的沟通能力。应聘者只有具备良好的沟通能力，才能充分了解面试官的需求，从而有针对性地选择算法解决问题。

面试题 4：二维数组中的查找

题目：在一个二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样一个二维数组和一个整数，判断数组中是否含有该整数。

例如下面的二维数组就是每行、每列都递增排序。如果在这个数组中查找数字 7，则返回 true；如果查找数字 5，由于数组不含有该数字，则返回 false。

1	2	8	9
2	4	9	12
4	7	10	13
6	8	11	15

在分析这个问题的时候，很多应聘者都会把二维数组画成矩形，然后从数组中选取一个数字，分3种情况来分析查找的过程。当数组中选取的数字刚好和要查找的数字相等时，就结束查找过程。如果选取的数字小于要查找的数字，那么根据数组排序的规则，要查找的数字应该在当前选取位置的右边或者下边，如图2.1(a)所示。同样，如果选取的数字大于要查找的数字，那么要查找的数字应该在当前选取位置的上边或者左边，如图2.1(b)所示。

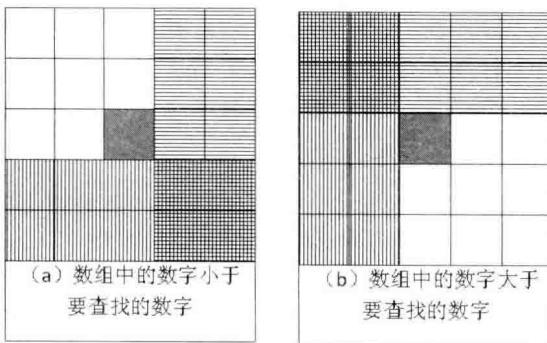


图2.1 二维数组中的查找

注：在数组中间选择一个数（深色方格），根据它的大小判断要查找的数字可能出现的区域（阴影部分）。

在上面的分析中，由于要查找的数字相对于当前选取的位置有可能在两个区域中出现，而且这两个区域还有重叠，这问题看起来就复杂了，于是很多人就卡在这里束手无策了。

当我们需要解决一个复杂的问题时，一个很有效的办法就是从一个具体的问题入手，通过分析简单具体的例子，试图寻找普遍的规律。针对这个问题，我们不妨也从一个具体的例子入手。下面我们将以在题目中给出的数组中查找数字7为例来一步步分析查找的过程。

前面我们之所以遇到难题，是因为我们在二维数组的中间选取一个数字来和要查找的数字进行比较，这就导致下一次要查找的是两个相互重叠的区域。如果我们从数组的一个角上选取数字来和要查找的数字进行比较，那么情况会不会变简单呢？

首先我们选取数组右上角的数字9。由于9大于7，并且9还是第4列的第一个（也是最小的）数字，因此7不可能出现在数字9所在的列。于

是我们把这一列从需要考虑的区域内剔除，之后只需要分析剩下的3列，如图2.2(a)所示。在剩下的矩阵中，位于右上角的数字是8。同样8大于7，因此8所在的列我们也可以剔除。接下来我们只要分析剩下的两列即可，如图2.2(b)所示。

在由剩余的两列组成的数组中，数字2位于数组的右上角。2小于7，那么要查找的7可能在2的右边，也可能在2的下边。在前面的步骤中，我们已经发现2右边的列都已经被剔除了，也就是说7不可能出现在2的右边，因此7只可能出现在2的下边。于是我们把数字2所在的行也剔除，只分析剩下的三行两列数字，如图2.2(c)所示。在剩下的数字中，数字4位于右上角，和前面一样，我们把数字4所在的行也删除，最后剩下两行两列数字，如图2.2(d)所示。

在剩下的两行两列4个数字中，位于右上角的刚好就是我们要查找的数字7，于是查找过程就可以结束了。

1	2	8	9
2	4	9	12
4	7	10	13
6	8	11	15

(a) 9大于7，下一次只需要在9的左边区域查找

1	2	8	9
2	4	9	12
4	7	10	13
6	8	11	15

(b) 8大于7，下一次只需要在8的左边区域查找

1	2	8	9
2	4	9	12
4	7	10	13
6	8	11	15

(c) 2小于7，下一次只需要在2的下边区域查找

1	2	8	9
2	4	9	12
4	7	10	13
6	8	11	15

(d) 4小于7，下一次只需要在4的下边区域查找

图2.2 在二维数组中查找7的步骤

注：矩阵中加阴影的区域是下一步查找的范围。

总结上述查找的过程，我们发现如下规律：首先选取数组中右上角的

数字。如果该数字等于要查找的数字，则查找过程结束；如果该数字大于要查找的数字，则剔除这个数字所在的列；如果该数字小于要查找的数字，则剔除这个数字所在的行。也就是说，如果要查找的数字不在数组的右上角，则每一次都在数组的查找范围内剔除一行或者一列，这样每一步都可以缩小查找的范围，直到找到要查找的数字，或者查找范围为空。

把整个查找过程分析清楚之后，我们再写代码就不是一件很难的事情了。下面是上述思路对应的参考代码：

```
bool Find(int* matrix, int rows, int columns, int number)
{
    bool found = false;

    if(matrix != nullptr && rows > 0 && columns > 0)
    {
        int row = 0;
        int column = columns - 1;
        while(row < rows && column >= 0)
        {
            if(matrix[row * columns + column] == number)
            {
                found = true;
                break;
            }
            else if(matrix[row * columns + column] > number)
                -- column;
            else
                ++ row;
        }
    }

    return found;
}
```

在前面的分析中，我们都选取数组查找范围内的右上角数字。同样，我们也可以选取左下角的数字。感兴趣的读者不妨自己分析一下每次都选取左下角数字的查找过程。但我们不能选择左上角数字或者右下角数字。以左上角数字为例，最初数字 1 位于初始数组的左上角，由于 1 小于 7，那么 7 应该位于 1 的右边或者下边。此时我们既不能从查找范围内剔除 1 所在的行，也不能剔除 1 所在的列，这样我们就无法缩小查找的范围。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/04_FindInPartiallySortedMatrix



测试用例：

- 二维数组中包含查找的数字(查找的数字是数组中的最大值和最小值；查找的数字介于数组中的最大值和最小值之间)。
- 二维数组中没有查找的数字(查找的数字大于数组中的最大值；查找的数字小于数组中的最小值；查找的数字在数组的最大值和最小值之间但数组中没有这个数字)。
- 特殊输入测试(输入空指针)。



本题考点：

- 考查应聘者对二维数组的理解及编程能力。二维数组在内存中占据连续的空间。在内存中从上到下存储各行元素，在同一行中按照从左到右的顺序存储。因此我们可以根据行号和列号计算出相对于数组首地址的偏移量，从而找到对应的元素。
- 考查应聘者分析问题的能力。当应聘者发现问题比较复杂时，能不能通过具体的例子找出其中的规律，是能否解决这个问题的关键所在。这个题目只要从一个具体的二维数组的右上角开始分析，就能找到查找的规律，从而找到解决问题的突破口。

2.3.2 字符串

字符串是由若干字符组成的序列。由于字符串在编程时使用的频率非常高，为了优化，很多语言都对字符串做了特殊的规定。下面分别讨论C/C++和C#中字符串的特性。

C/C++中每个字符串都以字符'\0'作为结尾，这样我们就能很方便地找到字符串的最后尾部。但由于这个特点，每个字符串中都有一个额外字符的开销，稍不留神就会造成字符串的越界。比如下面的代码：

```
char str[10];
strcpy(str, "0123456789");
```

我们先声明一个长度为 10 的字符数组，然后把字符串"0123456789"复制到数组中。"0123456789"这个字符串看起来只有 10 个字符，但实际上它的末尾还有一个'\0'字符，因此它的实际长度为 11 字节。要正确地复制该字符串，至少需要一个长度为 11 字节的数组。

为了节省内存，C/C++ 把常量字符串放到单独的一个内存区域。当几个指针赋值给相同的常量字符串时，它们实际上会指向相同的内存地址。但用常量内存初始化数组，情况却有所不同。下面通过一个面试题来学习这一知识点。运行下面的代码，得到的结果是什么？

```
int _tmain(int argc, _TCHAR* argv[])
{
    char str1[] = "hello world";
    char str2[] = "hello world";

    char* str3 = "hello world";
    char* str4 = "hello world";

    if(str1 == str2)
        printf("str1 and str2 are same.\n");
    else
        printf("str1 and str2 are not same.\n");

    if(str3 == str4)
        printf("str3 and str4 are same.\n");
    else
        printf("str3 and str4 are not same.\n");

    return 0;
}
```

str1 和 str2 是两个字符串数组，我们会为它们分配两个长度为 12 字节的空间，并把"hello world"的内容分别复制到数组中去。这是两个初始地址不同的数组，因此 str1 和 str2 的值也不相同，所以输出的第一行是"str1 and str2 are not same"。

str3 和 str4 是两个指针，我们无须为它们分配内存以存储字符串的内容，而只需要把它们指向"hello world"在内存中的地址就可以了。由于"hello world"是常量字符串，它在内存中只有一个拷贝，因此 str3 和 str4 指向的是同一个地址。所以比较 str3 和 str4 的值得到的结果是相同的，输出的第二行是"str3 and str4 are same"。

在 C# 中，封装字符串的类型 System.String 有一个非常特殊的性质：String 的内容是不能改变的。一旦试图改变 String 的内容，就会产生一个新的实例。请看下面的 C# 代码：

```
String str = "hello";
str.ToUpper();
str.Insert(0, " WORLD");
```

虽然我们对 str 执行了 ToUpper 和 Insert 两个操作，但操作的结果都是生成一个新的 String 实例并在返回值中返回，str 本身的内容都不会发生改变，因此最终 str 的值仍然是"hello"。由此可见，如果试图改变 String 的内容，则改变之后的值只能通过返回值得到。用 String 进行连续多次修改，每一次修改都会产生一个临时对象，这样开销太大会影响效率。为此，C# 定义了一个新的与字符串相关的类型 StringBuilder，它能容纳修改后的结果。因此，如果要连续多次修改字符串内容，用 StringBuilder 是更好的选择。

和修改 String 的内容类似，如果我们试图把一个常量字符串赋值给一个 String 实例，那么也不是把 String 的内容改成赋值的字符串，而是生成一个新的 String 实例。请看下面的代码：

```
class Program
{
    internal static void ValueOrReference(Type type)
    {
        String result = "The type " + type.Name;

        if (type.IsValueType)
            Console.WriteLine(result + " is a value type.");
        else
            Console.WriteLine(result + " is a reference type.");
    }

    internal static void ModifyString(String text)
    {
        text = "world";
    }

    static void Main(string[] args)
    {
        String text = "hello";

        ValueOrReference(text.GetType());
        ModifyString(text);

        Console.WriteLine(text);
    }
}
```

在上面的代码中，我们先判断 String 是值类型还是引用类型。类型 String 的定义是 public sealed class String {...}。既然是 class，那么 String 自然就是引用类型。接下来在方法 ModifyString 里，对 text 赋值一个新的字符串。

我们要记得 `text` 的内容是不能被修改的。此时会先生成一个新的内容是 "world" 的 `String` 实例，然后把 `text` 指向这个新的实例。由于参数 `text` 没有加 `ref` 或者 `out`，出了方法 `ModifyString` 之后，`text` 还是指向原来的字符串，因此输出仍然是 "hello"。要想实现了函数之后 `text` 变成 "world" 的效果，我们必须把参数 `text` 标记 `ref` 或者 `out`。

面试题 5：替换空格

题目：请实现一个函数，把字符串中的每个空格替换成 "%20"。例如，输入 "We are happy."，则输出 "We%20are%20happy."。

在网络编程中，如果 URL 参数中含有特殊字符，如空格、 '#' 等，则可能导致服务器端无法获得正确的参数值。我们需要将这些特殊符号转换成服务器可以识别的字符。转换的规则是在 '%' 后面跟上 ASCII 码的两位十六进制的表示。比如空格的 ASCII 码是 32，即十六进制的 0x20，因此空格被替换成 "%20"。再比如 '#' 的 ASCII 码为 35，即十六进制的 0x23，它在 URL 中被替换为 "%23"。

看到这个题目，我们首先应该想到的是原来一个空格字符，替换之后变成 '%'、'2' 和 '0' 这 3 个字符，因此字符串会变长。如果是在原来的字符串上进行替换，就有可能覆盖修改在该字符串后面的内存。如果是创建新的字符串并在新的字符串上进行替换，那么我们可以自己分配足够多的内存。由于有两种不同的解决方案，我们应该向面试官问清楚，让他明确告诉我们他的需求。假设面试官让我们在原来的字符串上进行替换，并且保证输入的字符串后面有足够的空余内存。

❖ 时间复杂度为 $O(n^2)$ 的解法，不足以拿到 Offer

现在我们考虑怎么执行替换操作。最直观的做法是从头到尾扫描字符串，每次碰到空格字符的时候进行替换。由于是把 1 个字符替换成 3 个字符，我们必须要把空格后面所有的字符都后移 2 字节，否则就有两个字符被覆盖了。

举个例子，我们从头到尾把 "We are happy." 中的每个空格替换成 "%20"。为了形象起见，我们可以用一个表格来表示字符串，表格中的每个格子表示一个字符，如图 2.3 (a) 所示。

(a)	<table border="1"><tr><td>W</td><td>e</td><td></td><td>a</td><td>r</td><td>e</td><td></td><td>h</td><td>a</td><td>p</td><td>p</td><td>y</td><td>.</td><td>\0</td><td></td><td></td><td></td><td></td><td></td></tr></table>	W	e		a	r	e		h	a	p	p	y	.	\0					
W	e		a	r	e		h	a	p	p	y	.	\0							
(b)	<table border="1"><tr><td>W</td><td>e</td><td>%</td><td>2</td><td>0</td><td>a</td><td>r</td><td>e</td><td></td><td>h</td><td>a</td><td>p</td><td>p</td><td>y</td><td>.</td><td>\0</td><td></td><td></td><td></td></tr></table>	W	e	%	2	0	a	r	e		h	a	p	p	y	.	\0			
W	e	%	2	0	a	r	e		h	a	p	p	y	.	\0					
(c)	<table border="1"><tr><td>W</td><td>e</td><td>%</td><td>2</td><td>0</td><td>a</td><td>r</td><td>e</td><td>%</td><td>2</td><td>0</td><td>h</td><td>a</td><td>p</td><td>p</td><td>y</td><td>.</td><td>\0</td><td></td></tr></table>	W	e	%	2	0	a	r	e	%	2	0	h	a	p	p	y	.	\0	
W	e	%	2	0	a	r	e	%	2	0	h	a	p	p	y	.	\0			

图 2.3 从前往后把字符串中的空格替换成'%'20'的过程

注：(a) 字符串"We are happy."。(b) 把字符串中的第一个空格替换成'%'20'。灰色背景表示需要移动的字符。(c) 把字符串中的第二个空格替换成'%'20'。浅灰色背景表示需要移动一次的字符，深灰色背景表示需要移动两次的字符。

我们替换第一个空格，这个字符串变成图 2.3 (b) 中的内容，表格中灰色背景的格子表示需要进行移动的区域。接着我们替换第二个空格，替换之后的内容如图 2.3 (c) 所示。同时，我们注意到用深灰色背景标注的“happy”部分被移动了两次。

假设字符串的长度是 n 。对每个空格字符，需要移动后面 $O(n)$ 个字符，因此对于含有 $O(n)$ 个空格字符的字符串而言，总的时间效率是 $O(n^2)$ 。

当我们把这种思路阐述给面试官后，他不会就此满意，他将让我们寻找更快的方法。在前面的分析中，我们发现数组中很多字符都移动了很多次，能不能减少移动次数呢？答案是肯定的。我们换一种思路，把从前向后替换改成从后向前替换。

❖ 时间复杂度为 $O(n)$ 的解法，搞定 Offer 就靠它了

我们可以先遍历一次字符串，这样就能统计出字符串中空格的总数，并可以由此计算出替换之后的字符串的总长度。每替换一个空格，长度增加 2，因此替换以后字符串的长度等于原来的长度加上 2 乘以空格数目。我们还是以前面的字符串"We are happy."为例。"We are happy."这个字符串的长度是 14（包括结尾符号'\0'），里面有两个空格，因此替换之后字符串的长度是 18。

我们从字符串的后面开始复制和替换。首先准备两个指针： P_1 和 P_2 。 P_1 指向原始字符串的末尾，而 P_2 指向替换之后的字符串的末尾，如图 2.4 (a) 所示。接下来我们向前移动指针 P_1 ，逐个把它指向的字符复制到 P_2 指向的位置，直到碰到第一个空格为止。此时字符串如图 2.4 (b) 所示，灰

色背景的区域是进行了字符复制（移动）的区域。碰到第一个空格之后，把 P_1 向前移动 1 格，在 P_2 之前插入字符串 "%20"。由于 "%20" 的长度为 3，同时也要把 P_2 向前移动 3 格，如图 2.4 (c) 所示。

我们接着向前复制，直到碰到第二个空格，如图 2.4 (d) 所示。和上一次一样，我们再把 P_1 向前移动 1 格，并把 P_2 向前移动 3 格插入 "%20"，如图 2.4 (e) 所示。此时 P_1 和 P_2 指向同一位置，表明所有空格都已经替换完毕。

从上面的分析中我们可以看出，所有的字符都只复制（移动）一次，因此这个算法的时间效率是 $O(n)$ ，比第一个思路要快。

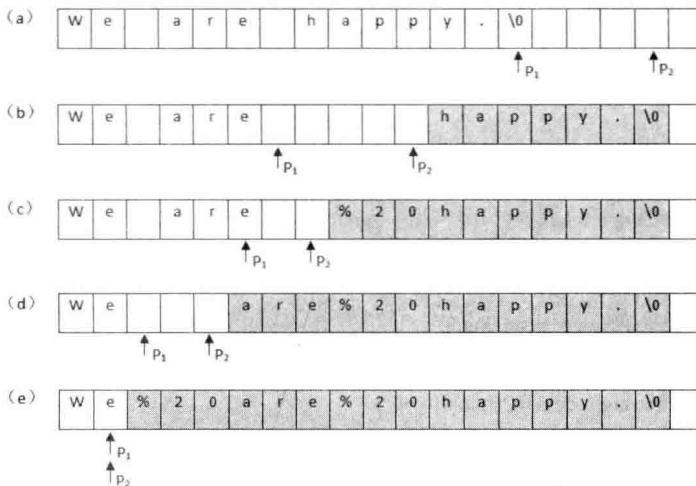


图 2.4 从后往前把字符串中的空格替换成 "%20" 的过程

注：图中带有阴影的区域表示被移动的字符。（a）把第一个指针指向字符串的末尾，把第二个指针指向替换之后的字符串的末尾。（b）依次复制字符串的内容，直至第一个指针碰到第一个空格。（c）把第一个空格替换成 "%20"，把第一个指针向前移动 1 格，把第二个指针向前移动 3 格。（d）依次向前复制字符串中的字符，直至碰到空格。（e）替换字符串中的倒数第二个空格，把第一个指针向前移动 1 格，把第二个指针向前移动 3 格。

在面试过程中，我们也可以和前面的分析一样画一两个示意图解释自己的思路，这样既能帮助我们厘清思路，也能使我们和面试官的交流变得更加高效。在面试官肯定我们的思路之后，就可以开始写代码了。下面是参考代码：

```

/*length 为字符数组 string 的总容量*/
void ReplaceBlank(char string[], int length)
{
    if(string == nullptr || length <= 0)

        return;

    /*originalLength 为字符串 string 的实际长度*/
    int originalLength = 0;
    int numberOfBlank = 0;
    int i = 0;
    while(string[i] != '\0')
    {
        ++ originalLength;

        if(string[i] == ' ')
            ++ numberOfBlank;

        ++ i;
    }

    /*newLength 为把空格替换成 "%20" 之后的长度*/
    int newLength = originalLength + numberOfBlank * 2;
    if(newLength > length)
        return;

    int indexOfOriginal = originalLength;
    int indexOfNew = newLength;
    while(indexOfOriginal >= 0 && indexOfNew > indexOfOriginal)
    {
        if(string[indexOfOriginal] == ' ')
        {
            string[indexOfNew--] = '0';
            string[indexOfNew--] = '2';
            string[indexOfNew--] = '%';
        }
        else
        {
            string[indexOfNew--] = string[indexOfOriginal];
        }

        -- indexOfOriginal;
    }
}

```



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/05_ReplaceSpaces



测试用例：

- 输入的字符串中包含空格（空格位于字符串的最前面；空格位于字符串的最后面；空格位于字符串的中间；字符串中有连续多个空格）。
- 输入的字符串中没有空格。
- 特殊输入测试（字符串是一个 nullptr 指针；字符串是一个空字符串；字符串只有一个空格字符；字符串中有连续多个空格）。



本题考点：

- 考查应聘者对字符串的编程能力。
- 考查应聘者分析时间效率的能力。我们要能清晰地分析出两种不同方法的时间效率各是多少。
- 考查应聘者对内存覆盖是否有高度的警惕。在分析得知字符串会变长之后，我们能够意识到潜在的问题，并主动和面试官沟通以寻找问题的解决方案。
- 考查应聘者的思维能力。在从前到后替换的思路被面试官否定之后，我们能迅速想到从后往前替换的方法，这是解决此题的关键。



相关题目：

有两个排序的数组 A1 和 A2，内存 A1 的末尾有足够的空余空间容纳 A2。请实现一个函数，把 A2 中的所有数字插入 A1 中，并且所有的数字是排序的。

和前面的例题一样，很多人首先想到的办法是在 A1 中从头到尾复制数字，但这样就会出现多次复制一个数字的情况。更好的办法是从尾到头比较 A1 和 A2 中的数字，并把较大的数字复制到 A1 中的合适位置。



举一反三：

在合并两个数组（包括字符串）时，如果从前往后复制每个数字（或字符）则需要重复移动数字（或字符）多次，那么我们可以考虑从后往前复制，这样就能减少移动的次数，从而提高效率。

2.3.3 链表

链表应该是面试时被提及最频繁的数据结构。链表的结构很简单，它由指针把若干个节点连接成链状结构。链表的创建、插入节点、删除节点等操作都只需要20行左右的代码就能实现，其代码量比较适合面试。而像哈希表、有向图等复杂数据结构，实现它们的一个操作需要的代码量都较大，很难在几十分钟的面试中完成。另外，由于链表是一种动态的数据结构，其需要对指针进行操作，因此应聘者需要有较好的编程功底才能写出完整的操作链表的代码。而且链表这种数据结构很灵活，面试官可以用链表来设计具有挑战性的面试题。基于上述几个原因，很多面试官都特别青睐与链表相关的题目。

我们说链表是一种动态数据结构，是因为在创建链表时，无须知道链表的长度。当插入一个节点时，我们只需要为新节点分配内存，然后调整指针的指向来确保新节点被链接到链表当中。内存分配不是在创建链表时一次性完成的，而是每添加一个节点分配一次内存。由于没有闲置的内存，链表的空间效率比数组高。如果单向链表的节点定义如下：

```
struct ListNode
{
    int         m_nValue;
    ListNode*  m_pNext;
};
```

那么往该链表的末尾添加一个节点的C++代码如下：

```
void AddToTail(ListNode** pHead, int value)
{
    ListNode* pNew = new ListNode();
    pNew->m_nValue = value;
    pNew->m_pNext = nullptr;

    if(*pHead == nullptr)
    {
        *pHead = pNew;
    }
    else
    {
        ListNode* pNode = *pHead;

        while(pNode->m_pNext != nullptr)
            pNode = pNode->m_pNext;

        pNode->m_pNext = pNew;
    }
}
```

在上面的代码中，我们要特别注意函数的第一个参数 pHead 是一个指向指针的指针。当我们往一个空链表中插入一个节点时，新插入的节点就是链表的头指针。由于此时会改动头指针，因此必须把 pHead 参数设为指向指针的指针，否则出了这个函数 pHead 仍然是一个空指针。

由于链表中的内存不是一次性分配的，因而我们无法保证链表的内存和数组一样是连续的。因此，如果想在链表中找到它的第 i 个节点，那么我们只能从头节点开始，沿着指向下一个节点的指针遍历链表，它的时间效率为 $O(n)$ 。而在数组中，我们可以根据下标在 $O(1)$ 时间内找到第 i 个元素。下面是在链表中找到第一个含有某值的节点并删除该节点的代码：

```
void RemoveNode(ListNode** pHead, int value)
{
    if(pHead == nullptr || *pHead == nullptr)
        return;

    ListNode* pToBeDeleted = nullptr;
    if((*pHead)->m_nValue == value)
    {
        pToBeDeleted = *pHead;
        *pHead = (*pHead)->m_pNext;
    }
    else
    {
        ListNode* pNode = *pHead;
        while(pNode->m_pNext != nullptr
              && pNode->m_pNext->m_nValue != value)
            pNode = pNode->m_pNext;

        if(pNode->m_pNext != nullptr && pNode->m_pNext->m_nValue == value)
        {
            pToBeDeleted = pNode->m_pNext;
            pNode->m_pNext = pNode->m_pNext->m_pNext;
        }
    }

    if(pToBeDeleted != nullptr)
    {
        delete pToBeDeleted;
        pToBeDeleted = nullptr;
    }
}
```

除了简单的单向链表经常被设计为面试题（详见面试题 6 “从尾到头打印链表”、面试题 18 “删除链表的节点”、面试题 22 “链表中倒数第 k 个节点”、面试题 24 “反转链表”、面试题 25 “合并两个排序的链表”、面试题 52 “两个链表的第一个公共节点” 等），链表的其他形式同样也备受面试官的青睐。

- 把链表的末尾节点的指针指向头节点，从而形成一个环形链表（详见面试题62“圆圈中最后剩下的数字”）。
- 链表中的节点中除了有指向下一个节点的指针，还有指向前一个节点的指针。这就是双向链表（详见面试题36“二叉搜索树与双向链表”）。
- 链表中的节点中除了有指向下一个节点的指针，还有指向任意节点的指针。这就是复杂链表（详见面试题35“复杂链表的复制”）。

面试题6：从尾到头打印链表

题目：输入一个链表的头节点，从尾到头反过来打印出每个节点的值。
链表节点定义如下：

```
struct ListNode
{
    int         m_nKey;
    ListNode*  m_pNext;
};
```

看到这道题后，很多人的第一反应是从头到尾输出将会比较简单，于是我们很自然地想到把链表中链接节点的指针反转过来，改变链表的方向，然后就可以从头到尾输出了。但该方法会改变原来链表的结构。是否允许在打印链表的时候修改链表的结构？这取决于面试官的要求，因此在面试的时候我们要询问清楚面试官的要求。



面试小提示：

在面试中，如果我们打算修改输入的数据，则最好先问面试官是不是允许修改。

通常打印是一个只读操作，我们不希望打印时修改内容。假设面试官也要求这个题目不能改变链表的结构。

接下来我们想到解决这个问题肯定要遍历链表。遍历的顺序是从头到尾，可输出的顺序却是从尾到头。也就是说，第一个遍历到的节点最后一个输出，而最后一个遍历到的节点第一个输出。这就是典型的“后进先出”，我们可以用栈实现这种顺序。每经过一个节点的时候，把该节点放到一个栈中。当遍历完整个链表后，再从栈顶开始逐个输出节点的值，此时输出

的节点的顺序已经反转过来了。这种思路的实现代码如下：

```
void PrintListReversingly_Iteratively(ListNode* pHead)
{
    std::stack<ListNode*> nodes;

    ListNode* pNode = pHead;
    while(pNode != nullptr)
    {
        nodes.push(pNode);
        pNode = pNode->m_pNext;
    }

    while(!nodes.empty())
    {
        pNode = nodes.top();
        printf("%d\t", pNode->m_nValue);
        nodes.pop();
    }
}
```

既然想到了用栈来实现这个函数，而递归在本质上就是一个栈结构，于是很自然地又想到了用递归来实现。要实现反过来输出链表，我们每访问到一个节点的时候，先递归输出它后面的节点，再输出该节点自身，这样链表的输出结果就反过来了。

基于这样的思路，不难写出如下代码：

```
void PrintListReversingly_Recursively(ListNode* pHead)
{
    if(pHead != nullptr)
    {
        if (pHead->m_pNext != nullptr)
        {
            PrintListReversingly_Recursively(pHead->m_pNext);
        }

        printf("%d\t", pHead->m_nValue);
    }
}
```

上面的基于递归的代码看起来很简洁，但有一个问题：当链表非常长的时候，就会导致函数调用的层级很深，从而有可能导致函数调用栈溢出。显然用栈基于循环实现的代码的鲁棒性要好一些。更多关于循环和递归的讨论，详见本书的 2.4.1 节。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/06_PrintListInReversedOrder



测试用例：

- 功能测试（输入的链表有多个节点；输入的链表只有一个节点）。
- 特殊输入测试（输入的链表头节点指针为 nullptr）。



本题考点：

- 考查应聘者对单向链表的理解和编程能力。
- 考查应聘者对循环、递归和栈 3 个相互关联的概念的理解。

2.3.4 树

树是一种在实际编程中经常遇到的数据结构。它的逻辑很简单：除根节点之外每个节点只有一个父节点，根节点没有父节点；除叶节点之外所有节点都有一个或多个子节点，叶节点没有子节点。父节点和子节点之间用指针链接。由于树的操作会涉及大量的指针，因此与树有关的面试题都不太容易。当面试官想考查应聘者在有复杂指针操作的情况下写代码的能力时，他往往会想到用与树有关的面试题。

面试的时候提到的树，大部分是二叉树。所谓二叉树是树的一种特殊结构，在二叉树中每个节点最多只能有两个子节点。在二叉树中最重要的操作莫过于遍历，即按照某一顺序访问树中的所有节点。通常树有如下几种遍历方式。

- **前序遍历：**先访问根节点，再访问左子节点，最后访问右子节点。
图 2.5 中的二叉树的前序遍历的顺序是 10、6、4、8、14、12、16。
- **中序遍历：**先访问左子节点，再访问根节点，最后访问右子节点。
图 2.5 中的二叉树的中序遍历的顺序是 4、6、8、10、12、14、16。
- **后序遍历：**先访问左子节点，再访问右子节点，最后访问根节点。
图 2.5 中的二叉树的后序遍历的顺序是 4、8、6、12、16、14、10。

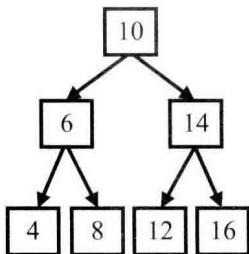


图 2.5 一个二叉树的例子

这 3 种遍历都有递归和循环两种不同的实现方法，每种遍历的递归实现都比循环实现要简洁很多。很多面试官喜欢直接或间接考查遍历（详见面试题 26 “树的子结构”、面试题 34 “二叉树中和为某一值的路径”、面试题 55 “二叉树的深度”）的具体代码实现，面试题 7 “重建二叉树”、面试题 33 “二叉搜索树的后序遍历序列”也是考查对遍历特点的理解，因此应聘者应该对这 3 种遍历的 6 种实现方法都了如指掌。

- 宽度优先遍历：先访问树的第一层节点，再访问树的第二层节点……一直到访问到最下面一层节点。在同一层节点中，以从左到右的顺序依次访问。我们可以对包括二叉树在内的所有树进行宽度优先遍历。图 2.5 中的二叉树的宽度优先遍历的顺序是 10、6、14、4、8、12、16。

面试题 32 “从上到下打印二叉树”就是考查宽度优先遍历算法的题目。

二叉树有很多特例，二叉搜索树就是其中之一。在二叉搜索树中，左子节点总是小于或等于根节点，而右子节点总是大于或等于根节点。图 2.5 中的二叉树就是一棵二叉搜索树。我们可以平均在 $O(\log n)$ 的时间内根据数值在二叉搜索树中找到一个节点。二叉搜索树的面试题有很多，如面试题 36 “二叉搜索树与双向链表”、面试题 68 “树中两个节点的最低公共祖先”。

二叉树的另外两个特例是堆和红黑树。堆分为最大堆和最小堆。在最大堆中根节点的值最大，在最小堆中根节点的值最小。有很多需要快速找到最大值或者最小值的问题都可以用堆来解决。红黑树是把树中的节点定义为红、黑两种颜色，并通过规则确保从根节点到叶节点的最长路径的长度不超过最短路径的两倍。在 C++ 的 STL 中，`set`、`multiset`、`map`、`multimap` 等数据结构都是基于红黑树实现的。与堆和红黑树相关的面试题，请参考面试题 40 “最小的 k 个数”。

面试题 7：重建二叉树

题目：输入某二叉树的前序遍历和中序遍历的结果，请重建该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。例如，输入前序遍历序列{1, 2, 4, 7, 3, 5, 6, 8}和中序遍历序列{4, 7, 2, 1, 5, 3, 8, 6}，则重建如图2.6所示的二叉树并输出它的头节点。二叉树节点的定义如下：

```
struct BinaryTreeNode
{
    int m_nValue;
    BinaryTreeNode* m_pLeft;
    BinaryTreeNode* m_pRight;
};
```

在二叉树的前序遍历序列中，第一个数字总是树的根节点的值。但在中序遍历序列中，根节点的值在序列的中间，左子树的节点的值位于根节点的值的左边，而右子树的节点的值位于根节点的值的右边。因此我们需要扫描中序遍历序列，才能找到根节点的值。

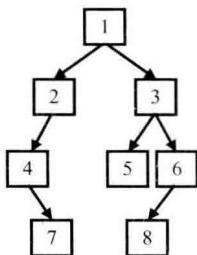


图2.6 根据前序遍历序列{1, 2, 4, 7, 3, 5, 6, 8}和中序遍历序列{4, 7, 2, 1, 5, 3, 8, 6}重建的二叉树

如图2.7所示，前序遍历序列的第一个数字1就是根节点的值。扫描中序遍历序列，就能确定根节点的值的位置。根据中序遍历的特点，在根节点的值1前面的3个数字都是左子树节点的值，位于1后面的数字都是右子树节点的值。

由于在中序遍历序列中，有3个数字是左子树节点的值，因此左子树共有3个左子节点。同样，在前序遍历序列中，根节点后面的3个数字就是3个左子树节点的值，再后面的所有数字都是右子树节点的值。这样我们就在前序遍历和中序遍历两个序列中分别找到了左、右子树对应的子序列。

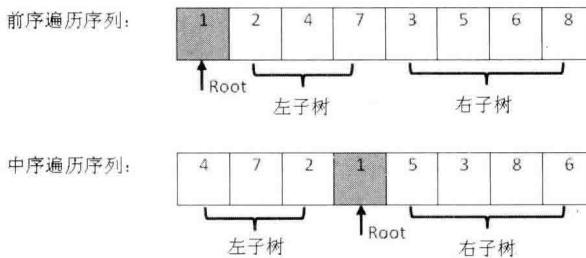


图 2.7 在二叉树的前序遍历和中序遍历序列中确定根节点的值、左子树节点的值和右子树节点的值

既然我们已经分别找到了左、右子树的前序遍历序列和中序遍历序列，我们可以用同样的方法分别构建左、右子树。也就是说，接下来的事情可以用递归的方法去完成。

在想清楚如何在前序遍历和中序遍历序列中确定左、右子树的子序列之后，我们可以写出如下的递归代码：

```
BinaryTreeNode* Construct(int* preorder, int* inorder, int length)
{
    if(preorder == nullptr || inorder == nullptr || length <= 0)
        return nullptr;

    return ConstructCore(preorder, preorder + length - 1,
                        inorder, inorder + length - 1);
}

BinaryTreeNode* ConstructCore
(
    int* startPreorder, int* endPreorder,
    int* startInorder, int* endInorder
)
{
    // 前序遍历序列的第一个数字是根节点的值
    int rootValue = startPreorder[0];
    BinaryTreeNode* root = new BinaryTreeNode();
    root->m_nValue = rootValue;
    root->m_pLeft = root->m_pRight = nullptr;

    if(startPreorder == endPreorder)
    {
        if(startInorder == endInorder
           && *startPreorder == *startInorder)
            return root;
        else
            throw std::exception("Invalid input.");
    }
}
```

```

// 在中序遍历序列中找到根节点的值
int* rootInorder = startInorder;
while(rootInorder <= endInorder && *rootInorder != rootValue)
    ++rootInorder;

if(rootInorder == endInorder && *rootInorder != rootValue)
    throw std::exception("Invalid input.");

int leftLength = rootInorder - startInorder;
int* leftPreorderEnd = startPreorder + leftLength;
if(leftLength > 0)
{
    // 构建左子树
    root->m_pLeft = ConstructCore(startPreorder + 1,
                                    leftPreorderEnd, startInorder, rootInorder - 1);
}
if(leftLength < endPreorder - startPreorder)
{
    // 构建右子树
    root->m_pRight = ConstructCore(leftPreorderEnd + 1,
                                    endPreorder, rootInorder + 1, endInorder);
}

return root;
}

```

在函数 `ConstructCore` 中，我们先根据前序遍历序列的第一个数字创建根节点，接下来在中序遍历序列中找到根节点的位置，这样就能确定左、右子树节点的数量。在前序遍历和中序遍历序列中划分了左、右子树节点的值之后，我们就可以递归地调用函数 `ConstructCore` 去分别构建它的左、右子树。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/07_ConstructBinaryTree



测试用例：

- 普通二叉树（完全二叉树；不完全二叉树）。
- 特殊二叉树（所有节点都没有右子节点的二叉树；所有节点都没有左子节点的二叉树；只有一个节点的二叉树）。

- 特殊输入测试（二叉树的根节点指针为 `nullptr`; 输入的前序遍历序列和中序遍历序列不匹配）。



本题考点：

- 考查应聘者对二叉树的前序遍历和中序遍历的理解程度。只有对二叉树的不同遍历算法有了深刻的理解，应聘者才有可能在遍历序列中划分出左、右子树对应的子序列。
- 考查应聘者分析复杂问题的能力。我们把构建二叉树的大问题分解成构建左、右子树的两个小问题。我们发现小问题和大问题在本质上是一致的，因此可以用递归的方式解决。更多关于分解复杂问题的讨论，请参考本书的 4.4 节。

面试题 8：二叉树的下一个节点

题目：给定一棵二叉树和其中的一个节点，如何找出中序遍历序列的下一个节点？树中的节点除了有两个分别指向左、右子节点的指针，还有一个指向父节点的指针。

在图 2.8 中的二叉树的中序遍历序列是 {d, b, h, e, i, a, f, c, g}。我们将以这棵树为例来分析如何找出二叉树的下一个节点。

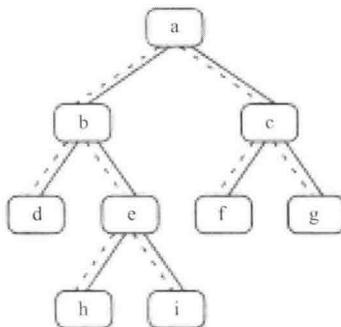


图 2.8 一棵有 9 个节点的二叉树。树中从父节点指向子节点的指针用实线表示，从子节点指向父节点的指针用虚线表示

如果一个节点有右子树，那么它的下一个节点就是它的右子树中的最左子节点。也就是说，从右子节点出发一直沿着指向左子节点的指针，我

们就能找到它的下一个节点。例如，图2.8中节点b的下一个节点是h，节点a的下一个节点是f。

接着我们分析一个节点没有右子树的情形。如果节点是它父节点的左子节点，那么它的下一个节点就是它的父节点。例如，图2.8中节点d的下一个节点是b，节点f的下一个节点是c。

如果一个节点既没有右子树，并且它还是它父节点的右子节点，那么这种情形就比较复杂。我们可以沿着指向父节点的指针一直向上遍历，直到找到一个是它父节点的左子节点的节点。如果这样的节点存在，那么这个节点的父节点就是我们要找的下一个节点。

为了找到图2.8中节点i的下一个节点，我们沿着指向父节点的指针向上遍历，先到达节点e。由于节点e是它父节点b的右节点，我们继续向上遍历到达节点b。节点b是它父节点a的左子节点，因此节点b的父节点a就是节点i的下一个节点。

找出节点g的下一个节点的步骤类似。我们先沿着指向父节点的指针到达节点c。由于节点c是它父节点a的右子节点，我们继续向上遍历到达节点a。由于节点a是树的根节点，它没有父节点，因此节点g没有下一个节点。

我们用如下的C++代码从二叉树中找出一个节点的下一个节点：

```
BinaryTreeNode* GetNext(BinaryTreeNode* pNode)
{
    if(pNode == nullptr)
        return nullptr;

    BinaryTreeNode* pNext = nullptr;
    if(pNode->m_pRight != nullptr)
    {
        BinaryTreeNode* pRight = pNode->m_pRight;
        while(pRight->m_pLeft != nullptr)
            pRight = pRight->m_pLeft;

        pNext = pRight;
    }
    else if(pNode->m_pParent != nullptr)
    {
        BinaryTreeNode* pCurrent = pNode;
        BinaryTreeNode* pParent = pNode->m_pParent;
        while(pParent != nullptr && pCurrent == pParent->m_pRight)
        {
            pCurrent = pParent;
        }
    }
}
```

```

    pParent = pParent->m_pParent;
}

pNext = pParent;
}

return pNext;
}

```



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/08_NextNodeInBinaryTrees



测试用例：

- 普通二叉树（完全二叉树；不完全二叉树）。
- 特殊二叉树（所有节点都没有右子节点的二叉树；所有节点都没有左子节点的二叉树；只有一个节点的二叉树；二叉树的根节点指针为 `nullptr`）。
- 不同位置的节点的下一个节点（下一个节点为当前节点的右子节点、右子树的最左子节点、父节点、跨层的父节点等；当前节点没有下一个节点）。



本题考点：

- 考查应聘者对二叉树中序遍历的理解程度。只有对二叉树的遍历算法有了深刻的理解，应聘者才有可能准确找出每个节点的中序遍历的下一个节点。
- 考查应聘者分析复杂问题的能力。应聘者只有画出二叉树的结构图、通过具体的例子找出中序遍历下一个节点的规律，才有可能设计出可行的算法。关于画图和举例解决复杂问题的讨论，请参考本书的 4.2 和 4.3 节。

2.3.5 栈和队列

栈是一个非常常见的数据结构，它在计算机领域被广泛应用，比如操作系统会给每个线程创建一个栈用来存储函数调用时各个函数的参数、返回地址及临时变量等。栈的特点是后进先出，即最后被压入（push）栈的元素会第一个被弹出（pop）。在面试题31“栈的压入、弹出序列”中，我们再详细分析进栈和出栈序列的特点。

通常栈是一个不考虑排序的数据结构，我们需要 $O(n)$ 时间才能找到栈中最大或者最小的元素。如果想要在 $O(1)$ 时间内得到栈的最大值或者最小值，则需要对栈做特殊的设计，详见面试题30“包含 min 函数的栈”。

队列是另外一种很重要的数据结构。和栈不同的是，队列的特点是先进先出，即第一个进入队列的元素将会第一个出来。在2.3.4节介绍的树的宽度优先遍历算法中，我们在遍历某一层树的节点时，把节点的子节点放到一个队列里，以备下一层节点的遍历。详细的代码参见面试题32“从上到下打印二叉树”。

栈和队列虽然是特点针锋相对的两个数据结构，但有意思的是它们却相互联系。请看面试题9“用两个栈实现队列”，同时读者也可以考虑如何用两个队列实现栈。

面试题9：用两个栈实现队列

题目：用两个栈实现一个队列。队列的声明如下，请实现它的两个函数 `appendTail` 和 `deleteHead`，分别完成在队列尾部插入节点和在队列头部删除节点的功能。

```
template <typename T> class CQueue
{
public:
    CQueue(void);
    ~CQueue(void);

    void appendTail(const T& node);
    T deleteHead();

private:
    stack<T> stack1;
    stack<T> stack2;
};
```

从上述队列的声明中可以看出，一个队列包含了两个栈 stack1 和 stack2，因此这道题的意图是要求我们操作这两个“先进后出”的栈实现一个“先进先出”的队列 CQueue。

我们通过一个具体的例子来分析往该队列插入和删除元素的过程。首先插入一个元素 a，不妨先把它插入 stack1，此时 stack1 中的元素有{a}，stack2 为空。再压入两个元素 b 和 c，还是插入 stack1，此时 stack1 中的元素有{a, b, c}，其中 c 位于栈顶，而 stack2 仍然是空的，如图 2.9 (a) 所示。

这时候我们试着从队列中删除一个元素。按照队列先入先出的规则，由于 a 比 b、c 先插入队列中，最先被删除的元素应该是 a。元素 a 存储在 stack1 中，但并不在栈顶上，因此不能直接进行删除。注意到 stack2 一直没有被使用过，现在是让 stack2 发挥作用的时候了。如果我们把 stack1 中的元素逐个弹出并压入 stack2，则元素在 stack2 中的顺序正好和原来在 stack1 中的顺序相反。因此经过 3 次弹出 stack1 和压入 stack2 的操作之后，stack1 为空，而 stack2 中的元素是{c,b,a}，这时候就可以弹出 stack2 的栈顶 a 了。此时的 stack1 为空，而 stack2 的元素为{c,b}，其中 b 在栈顶，如图 2.9 (b) 所示。

如果我们还想继续删除队列的头部应该怎么办呢？剩下的两个元素是 b 和 c，b 比 c 早进入队列，因此 b 应该先删除。而此时 b 恰好又在栈顶上，因此直接弹出 stack2 的栈顶即可。在这次弹出操作之后，stack1 仍然为空，而 stack2 中的元素为{c}，如图 2.9 (c) 所示。

从上面的分析中我们可以总结出删除一个元素的步骤：当 stack2 不为空时，在 stack2 中的栈顶元素是最先进入队列的元素，可以弹出。当 stack2 为空时，我们把 stack1 中的元素逐个弹出并压入 stack2。由于先进入队列的元素被压到 stack1 的底端，经过弹出和压入操作之后就处于 stack2 的顶端，又可以直接弹出。

接下来再插入一个元素 d。我们还是把它压入 stack1，如图 2.9 (d) 所示，这样会不会有问题呢？我们考虑下一次删除队列的头部 stack2 不为空，直接弹出它的栈顶元素 c，如图 2.9 (e) 所示。而 c 的确比 d 先进入队列，应该在 d 之前从队列中删除，因此不会出现任何矛盾。

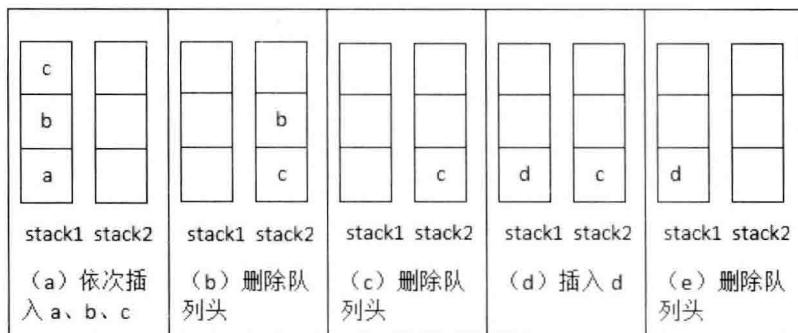


图 2.9 用两个栈模拟一个队列的操作

总结完每一次在队列中插入和删除操作的过程之后，我们就可以开始动手写代码了。参考代码如下：

```
template<typename T> void CQueue<T>::appendTail(const T& element)
{
    stack1.push(element);
}

template<typename T> T CQueue<T>::deleteHead()
{
    if(stack2.size() <= 0)
    {
        while(stack1.size() > 0)
        {
            T& data = stack1.top();
            stack1.pop();
            stack2.push(data);
        }
    }

    if(stack2.size() == 0)
        throw new exception("queue is empty");

    T head = stack2.top();
    stack2.pop();

    return head;
}
```



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/09_QueueWithTwoStacks



测试用例：

- 往空的队列里添加、删除元素。
- 往非空的队列里添加、删除元素。
- 连续删除元素直至队列为空。



本题考点：

- 考查应聘者对栈和队列的理解。
- 考查应聘者写与模板相关的代码的能力。
- 考查应聘者分析复杂问题的能力。本题解法的代码虽然只有二十几行，但形成正确的思路却不容易。应聘者能否通过具体的例子分析问题，通过画图的手段把抽象的问题形象化，从而解决这个相对复杂的问题，是能否顺利通过面试的关键。



相关题目：

用两个队列实现一个栈。

我们通过一系列栈的压入和弹出操作来分析用两个队列模拟一个栈的过程。如图 2.10 (a) 所示，我们先往栈内压入一个元素 a。由于两个队列现在都是空的，我们可以选择把 a 插入两个队列的任意一个。我们不妨把 a 插入 queue1。接下来继续往栈内压入 b、c 两个元素，我们把它们都插入 queue1。这个时候 queue1 包含 3 个元素 a、b 和 c，其中 a 位于队列的头部，c 位于队列的尾部。

现在我们考虑从栈内弹出一个元素。根据栈的后入先出原则，最后被压入栈的 c 应该最先被弹出。由于 c 位于 queue1 的尾部，而我们每次只能从队列的头部删除元素，因此我们可以先从 queue1 中依次删除元素 a、b 并插入 queue2，再从 queue1 中删除元素 c。这就相当于从栈中弹出元素 c 了，如图 2.10 (b) 所示。我们可以用同样的方法从栈内弹出元素 b，如图 2.10 (c) 所示。

接下来我们考虑往栈内压入一个元素 d。此时 queue1 已经有一个元素，我们就把 d 插入 queue1 的尾部，如图 2.10 (d) 所示。如果我们再从栈内弹出一个元素，那么此时被弹出的应该是最后被压入的 d。由于 d 位于 queue1 的尾部，我们只能先从头删除 queue1 的元素并插入 queue2，直到在 queue1 中遇到 d 再直接把它删除，如图 2.10 (e) 所示。

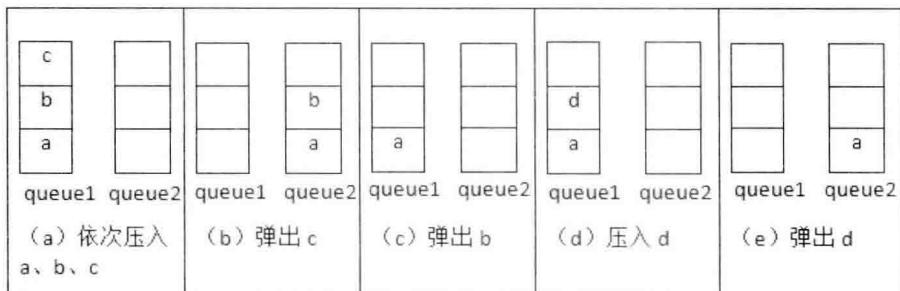


图 2.10 用两个队列模拟一个栈的操作

2.4 算法和数据操作

和数据结构一样，考查算法的面试题也备受面试官的青睐。有很多算法都可以用递归和循环两种不同的方式实现。通常基于递归的实现方法代码会比较简洁，但性能不如基于循环的实现方法。在面试的时候，我们可以根据题目的特点，甚至可以和面试官讨论选择合适的方法编程。

通常排序和查找是面试时考查算法的重点。在准备面试的时候，我们应该重点掌握二分查找、归并排序和快速排序，做到能随时正确、完整地写出它们的代码。

如果面试题要求在二维数组（可能具体表现为迷宫或者棋盘等）上搜索路径，那么我们可以尝试用回溯法。通常回溯法很适合用递归的代码实现。只有当面试官限定不可以用递归实现的时候，我们再考虑用栈来模拟递归的过程。

如果面试题是求某个问题的最优解，并且该问题可以分为多个子问题，那么我们可以尝试用动态规划。在用自上而下的递归思路去分析动态规划问题的时候，我们会发现子问题之间存在重叠的更小的子问题。为了避免

不必要的重复计算，我们用自下而上的循环代码来实现，也就是把子问题的最优解先算出来并用数组（一般是一维或者二维数组）保存下来，接下来基于子问题的解计算大问题的解。

如果我们告诉面试官动态规划的思路之后，面试官还在提醒说在分解子问题的时候是不是存在某个特殊的选择，如果采用这个特殊的选择将一定能得到最优解，那么，通常面试官这样的提示意味着该面试题可能适用于贪婪算法。当然，面试官也会要求应聘者证明贪婪选择的确最终能够得到最优解。

位运算可以看成一类特殊的算法，它是把数字表示成二进制之后对 0 和 1 的操作。由于位运算的对象为二进制数字，所以不是很直观，但掌握它也不难，因为总共只有与、或、异或、左移和右移 5 种位运算。

2.4.1 递归和循环

如果我们要重复地多次计算相同的问题，则通常可以选择用递归或者循环两种不同的方法。递归是在一个函数的内部调用这个函数自身。而循环则是通过设置计算的初始值及终止条件，在一个范围内重复运算。比如求 $1+2+\cdots+n$ ，我们可以用递归或者循环两种方式求出结果。对应的代码如下：

```
int AddFrom1ToN_Recursive(int n)
{
    return n <= 0 ? 0 : n + AddFrom1ToN_Recursive(n - 1);
}

int AddFrom1ToN_Iterative(int n)
{
    int result = 0;
    for(int i = 1; i <= n; ++ i)
        result += i;

    return result;
}
```

通常递归的代码会比较简洁。在上面的例子中，递归的代码只有一条语句，而循环则需要 4 条语句。在树的前序、中序、后序遍历算法的代码中，递归的实现明显要比循环简单得多。在面试的时候，如果面试官没有特别的要求，则应聘者可以尽量多采用递归的方法编程。



面试小提示：

通常基于递归实现的代码比基于循环实现的代码要简洁很多，更加容易实现。如果面试官没有特殊要求，则应聘者可以优先采用递归的方法编程。

递归虽然有简洁的优点，但它同时也有显著的缺点。递归由于是函数调用自身，而函数调用是有时间和空间的消耗的：每一次函数调用，都需要在内存栈中分配空间以保存参数、返回地址及临时变量，而且往栈里压入数据和弹出数据都需要时间。这就不难理解上述的例子中递归实现的效率不如循环。

另外，递归中有可能很多计算都是重复的，从而对性能带来很大的负面影响。递归的本质是把一个问题分解成两个或者多个小问题。如果多个小问题存在相互重叠的部分，就存在重复的计算。在面试题10“斐波那契数列”及面试题60“ n 个骰子的点数”中，我们将详细地分析递归和循环的性能区别。

通常应用动态规划解决问题时我们都是用递归的思路分析问题，但由于递归分解的子问题中存在大量的重复，因此我们总是用自下而上的循环来实现代码。我们将在面试题14“剪绳子”、面试题47“礼物的最大价值”及面试题48“最长不含重复字符的子字符串”中详细讨论如何用递归分析问题并基于循环写代码。

除效率之外，递归还有可能引起更严重的问题：调用栈溢出。在前面的分析中提到需要为每一次函数调用在内存栈中分配空间，而每个进程的栈的容量是有限的。当递归调用的层级太多时，就会超出栈的容量，从而导致调用栈溢出。在上述例子中，如果输入的参数比较小，如10，则它们都能返回结果55。但如果输入的参数很大，如5000，那么递归代码在运行的时候就会出错，但运行循环的代码能得到正确的结果12502500。

面试题10：斐波那契数列

题目一：求斐波那契数列的第 n 项。

写一个函数，输入 n ，求斐波那契(Fibonacci)数列的第 n 项。斐波那契数列的定义如下：

$$f(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ f(n-1) + f(n-2) & n > 1 \end{cases}$$

❖ 效率很低的解法，挑剔的面试官不会喜欢

很多 C 语言教科书在讲述递归函数的时候，都会用斐波那契数列作为例子，因此很多应聘者对这道题的递归解法都很熟悉。他们看到这道题的时候心中会忍不住一阵窃喜，因为他们能很快写出如下代码：

```
long long Fibonacci(unsigned int n)
{
    if(n <= 0)
        return 0;
    if(n == 1)
        return 1;
    return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

我们的教科书上反复用这个问题来讲解递归函数，并不能说明递归的解法最适合这道题目。面试官会提示我们上述递归的解法有很严重的效率问题并要求我们分析原因。

我们以求解 $f(10)$ 为例来分析递归的求解过程。想求得 $f(10)$ ，需要先求得 $f(9)$ 和 $f(8)$ 。同样，想求得 $f(9)$ ，需要先求得 $f(8)$ 和 $f(7)$ ……我们可以用树形结构来表示这种依赖关系，如图 2.11 所示。

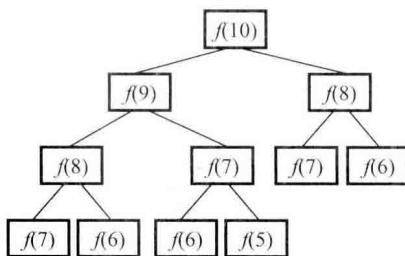


图 2.11 基于递归求斐波那契数列的第 10 项的调用过程

我们不难发现，在这棵树中有很多节点是重复的，而且重复的节点数会随着 n 的增大而急剧增加，这意味着计算量会随着 n 的增大而急剧增大。事实上，用递归方法计算的时间复杂度是以 n 的指数的方式递增的。读者不妨求斐波那契数列的第 100 项试试，感受一下这样递归会慢到什么程度。

❖ 面试官期待的实用解法

其实改进的方法并不复杂。上述递归代码之所以慢，是因为重复的计算太多，我们只要想办法避免重复计算就行了。比如我们可以把已经得到的数列中间项保存起来，在下次需要计算的时候我们先查找一下，如果前面已经计算过就不用再重复计算了。

更简单的办法是从下往上计算，首先根据 $f(0)$ 和 $f(1)$ 算出 $f(2)$ ，再根据 $f(1)$ 和 $f(2)$ 算出 $f(3)$ ……以此类推就可以算出第 n 项了。很容易理解，这种思路的时间复杂度是 $O(n)$ 。实现代码如下：

```
long long Fibonacci(unsigned n)
{
    int result[2] = {0, 1};
    if(n < 2)
        return result[n];

    long long fibNMinusOne = 1;
    long long fibNMinusTwo = 0;
    long long fibN = 0;
    for(unsigned int i = 2; i <= n; ++ i)
    {
        fibN = fibNMinusOne + fibNMinusTwo;
        fibNMinusTwo = fibNMinusOne;
        fibNMinusOne = fibN;
    }

    return fibN;
}
```

❖ 时间复杂度 $O(\log n)$ 但不够实用的解法

通常面试到这里也就差不多了，尽管我们还有比这更快的 $O(\log n)$ 算法。由于这种算法需要用到一个很生僻的数学公式，因此很少有面试官会要求我们掌握。不过以防不时之需，我们还是简要介绍一下这种算法。

在介绍这种方法之前，我们先介绍一个数学公式：

$$\begin{bmatrix} f(n) & f(n-1) \\ f(n-1) & f(n-2) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1}$$

这个公式用数学归纳法不难证明，感兴趣的读者不妨自己证明一下。

有了这个公式，我们只需要求得矩阵 $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1}$ 即可得到 $f(n)$ 。现在的问题转

换为如何求矩阵 $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ 的乘方。如果只是简单地从0开始循环， n 次方需要

n 次运算，则其时间复杂度仍然是 $O(n)$ ，并不比前面的方法快。但我们可以考虑乘方的如下性质：

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & n \text{ 为偶数} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & n \text{ 为奇数} \end{cases}$$

从上面的公式中我们可以看出，我们想求得 n 次方，就要先求得 $n/2$ 次方，再把 $n/2$ 次方的结果平方一下即可。这可以用递归的思路实现。

由于很少有面试官要求编程实现这种思路，本书中不再列出完整的代码，感兴趣的读者请参考附带的源代码。不过这种基于递归用 $O(\log n)$ 的时间求得 n 次方的算法却值得我们重视。我们在面试题16“数值的整数次方”中再详细讨论这种算法。

❖ 解法比较

用不同的方法求解斐波那契数列的时间效率大不相同。第一种基于递归的解法虽然直观但时间效率很低，在实际软件开发中不会用这种方法，也不可能得到面试官的青睐。第二种方法把递归的算法用循环实现，极大地提高了时间效率。第三种方法把求斐波那契数列转换成求矩阵的乘方，是一种很有创意的算法。虽然我们可以用 $O(\log n)$ 求得矩阵的 n 次方，但由于隐含的时间常数较大，很少会有软件采用这种算法。另外，实现这种解法的代码也很复杂，不太适合面试。因此第三种方法不是一种实用的算法，不过应聘者可以用它来展示自己的知识面。

除了面试官直接要求编程实现斐波那契数列，还有不少面试题可以看成斐波那契数列的应用。

题目二：青蛙跳台阶问题。

一只青蛙一次可以跳上1级台阶，也可以跳上2级台阶。求该青蛙跳上一个 n 级的台阶总共有多少种跳法。

首先我们考虑最简单的情况。如果只有1级台阶，那显然只有一种跳法。如果有2级台阶，那就有两种跳法：一种是分两次跳，每次跳1级；另一种就是一次跳2级。

接着我们再来讨论一般情况。我们把 n 级台阶时的跳法看成 n 的函数，记为 $f(n)$ 。当 $n > 2$ 时，第一次跳的时候就有两种不同的选择：一是第一次只跳 1 级，此时跳法数目等于后面剩下的 $n-1$ 级台阶的跳法数目，即为 $f(n-1)$ ；二是第一次跳 2 级，此时跳法数目等于后面剩下的 $n-2$ 级台阶的跳法数目，即为 $f(n-2)$ 。因此， n 级台阶的不同跳法的总数 $f(n) = f(n-1) + f(n-2)$ 。分析到这里，我们不难看出这实际上就是斐波那契数列了。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/10_Fibonacci



测试用例：

- 功能测试（如输入 3、5、10 等）。
- 边界值测试（如输入 0、1、2）。
- 性能测试（输入较大的数字，如 40、50、100 等）。



本题考点：

- 考查应聘者对递归、循环的理解及编码能力。
- 考查应聘者对时间复杂度的分析能力。
- 如果面试官采用的是青蛙跳台阶的问题，那么同时还在考查应聘者的数学建模能力。



本题扩展：

在青蛙跳台阶的问题中，如果把条件改成：一只青蛙一次可以跳上 1 级台阶，也可以跳上 2 级……它也可以跳上 n 级，此时该青蛙跳上一个 n 级的台阶总共有多少种跳法？我们用数学归纳法可以证明 $f(n) = 2^{n-1}$ 。



相关题目：

我们可以用 2×1 （图 2.12 的左边）的小矩形横着或者竖着去覆盖更大的矩形。请问用 8 个 2×1 的小矩形无重叠地覆盖一个 2×8 的大矩形（图 2.12 的右边），总共有多少种方法？



图 2.12 一个 2×1 的矩形和 2×8 的矩形

我们先把 2×8 的覆盖方法记为 $f(8)$ 。用第一个 2×1 的小矩形去覆盖大矩形的最左边时有两种选择：竖着放或者横着放。当竖着放的时候，右边还剩下 2×7 的区域，这种情形下的覆盖方法记为 $f(7)$ 。接下来考虑横着放的情况。当 2×1 的小矩形横着放在左上角的时候，左下角必须和横着放一个 2×1 的小矩形，而在右边还剩下 2×6 的区域，这种情形下的覆盖方法记为 $f(6)$ ，因此 $f(8)=f(7)+f(6)$ 。此时我们可以看出，这仍然是斐波那契数列。

2.4.2 查找和排序

查找和排序都是在程序设计中经常用到的算法。查找相对而言较为简单，不外乎顺序查找、二分查找、哈希表查找和二叉排序树查找。在面试的时候，不管是用循环还是用递归，面试官都期待应聘者能够信手拈来写出完整正确的二分查找代码，否则可能连继续面试的兴趣都没有。面试题 11 “旋转数组的最小数字” 和面试题 53 “在排序数组中查找数字” 都可以用二分查找算法解决。



面试小提示：

如果面试题是要求在排序的数组（或者部分排序的数组）中查找一个数字或者统计某个数字出现的次数，那么我们都可以尝试用二分查找算法。

哈希表和二叉排序树查找的重点在于考查对应的数据结构而不是算法。哈希表最主要的优点是我们利用它能够在 $O(1)$ 时间内查找某一元素，是效率最高的查找方式；但其缺点是需要额外的空间来实现哈希表。面试题 50 “第一个只出现一次的字符” 就是用哈希表的特性来实现高效查找的。

与二叉排序树查找算法对应的数据结构是二叉搜索树，我们将在面试题33“二叉搜索树的后序遍历序列”和面试题36“二叉搜索树与双向链表”中详细介绍二叉搜索树的特点。

排序比查找要复杂一些。面试官会经常要求应聘者比较插入排序、冒泡排序、归并排序、快速排序等不同算法的优劣。强烈建议应聘者在准备面试的时候，一定要对各种排序算法的特点烂熟于胸，能够从额外空间消耗、平均时间复杂度和最差时间复杂度等方面去比较它们的优缺点。需要特别强调的是，很多公司的面试官喜欢在面试环节要求应聘者写出快速排序的代码。应聘者不妨自己写一个快速排序的函数并用各种数据进行测试。当测试都通过之后，再和经典的实现进行比较，看看有什么区别。

实现快速排序算法的关键在于先在数组中选择一个数字，接下来把数组中的数字分为两部分，比选择的数字小的数字移到数组的左边，比选择的数字大的数字移到数组的右边。这个函数可以如下实现：

```
int Partition(int data[], int length, int start, int end)
{
    if(data == nullptr || length <= 0 || start < 0 || end >= length)
        throw new std::exception("Invalid Parameters");

    int index = RandomInRange(start, end);
    Swap(&data[index], &data[end]);

    int small = start - 1;
    for(index = start; index < end; ++ index)
    {
        if(data[index] < data[end])
        {
            ++ small;
            if(small != index)
                Swap(&data[index], &data[small]);
        }
    }

    ++ small;
    Swap(&data[small], &data[end]);

    return small;
}
```

函数RandomInRange用来生成一个在start和end之间的随机数，函数Swap的作用是用来交换两个数字。接下来我们可以用递归的思路分别对每次选中的数字的左右两边排序。下面就是递归实现快速排序的参考代码：

```
void QuickSort(int data[], int length, int start, int end)
{
```

```

if(start == end)
    return;

int index = Partition(data, length, start, end);
if(index > start)
    QuickSort(data, length, start, index - 1);
if(index < end)
    QuickSort(data, length, index + 1, end);
}

```

对一个长度为 n 的数组排序，只需把 `start` 设为 0、把 `end` 设为 $n-1$ ，调用函数 `QuickSort` 即可。

在前面的代码中，函数 `Partition` 除了可以用在快速排序算法中，还可以用来实现在长度为 n 的数组中查找第 k 大的数字。面试题 39 “数组中出现次数超过一半的数字” 和面试题 40 “最小的 k 个数” 都可以用这个函数来解决。

不同的排序算法适用的场合也不尽相同。快速排序虽然总体的平均效率是最好的，但也不是任何时候都是最优的算法。比如数组本身已经排好序了，而每一轮排序的时候都以最后一个数字作为比较的标准，此时快速排序的效率只有 $O(n^2)$ 。因此，在这种场合快速排序就不是最优的算法。在面试的时候，如果面试官要求实现一个排序算法，那么应聘者一定要问清楚这个排序应用的环境是什么、有哪些约束条件，在得到足够多的信息之后再选择最合适的排序算法。下面来看一个面试的片段。

面试官：请实现一个排序算法，要求时间效率为 $O(n)$ 。

应聘者：对什么数字进行排序，有多少个数字？

面试官：我们想对公司所有员工的年龄排序。我们公司总共有几万名员工。

应聘者：也就是说数字的大小是在一个较小的范围之内的，对吧？

面试官：嗯，是的。

应聘者：可以使用辅助空间吗？

面试官：看你用多少辅助内存。只允许使用常量大小辅助空间，不得超过 $O(n)$ 。

在面试的时候应聘者不要怕问面试官问题，只有多提问，应聘者才有可能明了面试官的意图。在上面的例子中，该应聘者通过几个问题就弄清楚了需排序的数字在一个较小的范围内，并且可以用辅助内存。知道了这

些限制条件，就不难写出如下代码了：

```
void SortAges(int ages[], int length)
{
    if(ages == nullptr || length <= 0)
        return;

    const int oldestAge = 99;
    int timesOfAge[oldestAge + 1];

    for(int i = 0; i <= oldestAge; ++ i)
        timesOfAge[i] = 0;

    for(int i = 0; i < length; ++ i)
    {
        int age = ages[i];
        if(age < 0 || age > oldestAge)
            throw new std::exception("age out of range.");

        ++ timesOfAge[age];
    }

    int index = 0;
    for(int i = 0; i <= oldestAge; ++ i)
    {
        for(int j = 0; j < timesOfAge[i]; ++ j)
        {
            ages[index] = i;
            ++ index;
        }
    }
}
```

公司员工的年龄有一个范围。在上面的代码中，允许的范围是 0~99岁。数组 `timesOfAge` 用来统计每个年龄出现的次数。某个年龄出现了多少次，就在数组 `ages` 里设置几次该年龄，这就相当于给数组 `ages` 排序了。该方法用长度 100 的整数数组作为辅助空间换来了 $O(n)$ 的时间效率。

面试题 11：旋转数组的最小数字

题目：把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。输入一个递增排序的数组的一个旋转，输出旋转数组的最小元素。例如，数组 {3, 4, 5, 1, 2} 为 {1, 2, 3, 4, 5} 的一个旋转，该数组的最小值为 1。

这道题最直观的解法并不难，从头到尾遍历数组一次，我们就能找出最小的元素。这种思路的时间复杂度显然是 $O(n)$ 。但是这种思路没有利用输入的旋转数组的特性，肯定达不到面试官的要求。

我们注意到旋转之后的数组实际上可以划分为两个排序的子数组，而且前面子数组的元素都大于或者等于后面子数组的元素。我们还注意到最小的元素刚好是这两个子数组的分界线。在排序的数组中我们可以用二分查找法实现 $O(\log n)$ 的查找。本题给出的数组在一定程度上是排序的，因此我们可以试着用二分查找法的思路来寻找这个最小的元素。

和二分查找法一样，我们用两个指针分别指向数组的第一个元素和最后一个元素。按照题目中旋转的规则，第一个元素应该是大于或者等于最后一个元素的（这其实不完全对，还有特例，后面再加以讨论）。

接着我们可以找到数组中间的元素。如果该中间元素位于前面的递增子数组，那么它应该大于或者等于第一个指针指向的元素。此时数组中最小的元素应该位于该中间元素的后面。我们可以把第一个指针指向该中间元素，这样可以缩小寻找的范围。移动之后的第一个指针仍然位于前面的递增子数组。

同样，如果中间元素位于后面的递增子数组，那么它应该小于或者等于第二个指针指向的元素。此时该数组中最小的元素应该位于该中间元素的前面。我们可以把第二个指针指向该中间元素，这样也可以缩小寻找的范围。移动之后的第二个指针仍然位于后面的递增子数组。

不管是移动第一个指针还是第二个指针，查找范围都会缩小到原来的一半。接下来我们再用更新之后的两个指针重复做新一轮的查找。

按照上述思路，第一个指针总是指向前面递增数组的元素，而第二个指针总是指向后面递增数组的元素。最终第一个指针将指向前面子数组的最后一个元素，而第二个指针会指向后面子数组的第一个元素。也就是它们最终会指向两个相邻的元素，而第二个指针指向的刚好是最小的元素。这就是循环结束的条件。

以前面的数组 {3, 4, 5, 1, 2} 为例，我们先把第一个指针指向第 0 个元素，把第二个指针指向第 4 个元素，如图 2.13 (a) 所示。位于两个指针中间（在数组中的下标是 2）的数字是 5，它大于第一个指针指向的数字。因此中间数字 5 一定位于第一个递增子数组，并且最小的数字一定位于它的后面。因此我们可以移动第一个指针，让它指向数组的中间，如图 2.13 (b) 所示。

此时位于这两个指针中间（在数组中的下标是 3）的数字是 1，它小于第二个指针指向的数字。因此这个中间数字 1 一定位于第二个递增子数组，

并且最小的数字一定位于它的前面或者它自己就是最小的数字。因此我们可以移动第二个指针，让它指向两个指针中间的元素，即下标为3的元素，如图2.13(c)所示。

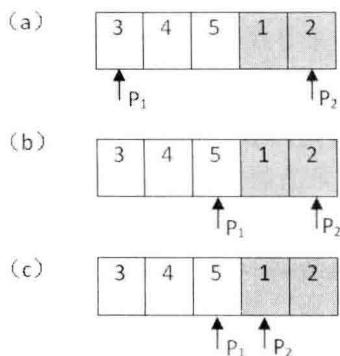


图2.13 在数组{3, 4, 5, 1, 2}中查找最小值的过程

注：旋转数组中包含两个递增排序的子数组，有阴影的是第二个子数组。(a) 把 P_1 指向数组的第一个数字， P_2 指向数组的最后一个数字。由于 P_1 和 P_2 中间的数字 5 大于 P_1 指向的数字，中间的数字在第一个子数组中。下一步把 P_1 指向中间的数字。(b) P_1 和 P_2 中间的数字 1 小于 P_2 指向的数字，中间的数字在第二个子数组中。下一步把 P_2 指向中间的数字。(c) P_1 和 P_2 指向两个相邻的数字，则 P_2 指向的是数组中的最小数字。

此时两个指针的距离是 1，表明第一个指针已经指向第一个递增子数组的末尾，而第二个指针指向第二个递增子数组的开头。第二个子数组的第一个数字就是最小的数字，因此第二个指针指向的数字就是我们查找的结果。

基于这个思路，我们可以写出如下代码：

```
int Min(int* numbers, int length)
{
    if(numbers == nullptr || length <= 0)
        throw new std::exception("Invalid parameters");

    int index1 = 0;
    int index2 = length - 1;
    int indexMid = index1;
    while(numbers[index1] >= numbers[index2])
    {
        if(index2 - index1 == 1)
        {
            indexMid = index2;
        }
    }
}
```

```

        break;
    }

    indexMid = (index1 + index2) / 2;
    if(numbers[indexMid] >= numbers[index1])
        index1 = indexMid;
    else if(numbers[indexMid] <= numbers[index2])
        index2 = indexMid;
}

return numbers[indexMid];
}

```

前面我们提到，在旋转数组中，由于是把递增排序数组前面的若干个数字搬到数组的后面，因此第一个数字总是大于或者等于最后一个数字。但按照定义还有一个特例：如果把排序数组的前面的 0 个元素搬到最后面，即排序数组本身，这仍然是数组的一个旋转，我们的代码需要支持这种情况。此时，数组中的第一个数字就是最小的数字，可以直接返回。这就是在上面的代码中，把 `indexMid` 初始化为 `index1` 的原因。一旦发现数组中第一个数字小于最后一个数字，表明该数组是排序的，就可以直接返回第一个数字。

上述代码是否就完美了呢？面试官会告诉我们其实不然。他将提示我们再仔细分析下标为 `index1` 和 `index2` (`index1` 和 `index2` 分别与图 2.13 中 P_1 和 P_2 相对应) 的两个数相同的情况。在前面的代码中，当这两个数相同，并且它们中间的数字 (`indexMid` 指向的数字) 也相同时，我们把 `indexMid` 赋值给 `index1`，也就是认为此时最小的数字位于中间数字的后面。是不是一定这样？

我们再来看一个例子。数组 {1, 0, 1, 1, 1} 和数组 {1, 1, 1, 0, 1} 都可以看成递增排序数组 {0, 1, 1, 1, 1} 的旋转，图 2.14 分别画出它们由最小数字分隔开的两个子数组。



图 2.14 数组{0, 1, 1, 1, 1}的两个旋转{1, 0, 1, 1, 1}和{1, 1, 1, 0, 1}

注：在这两个数组中，第一个数字、最后一个数字和中间数字都是 1，我们无法确定中间的数字 1 是属于第一个递增子数组还是属于第二个递增子数组。第二个递增子数组用灰色背景表示。

在这两种情况中，第一个指针和第二个指针指向的数字都是1，并且两个指针中间的数字也是1，这3个数字相同。在第一种情况下，中间数字（下标为2）位于后面的子数组；在第二种情况下，中间数字（下标为2）位于前面的子数组。因此，当两个指针指向的数字及它们中间的数字三者相同的时候，我们无法判断中间的数字是位于前面的子数组还是后面的子数组，也就无法移动两个指针来缩小查找的范围。此时，我们不得不采用顺序查找的方法。

在把问题分析清楚形成清晰的思路之后，我们就可以把前面的代码修改为：

```
int Min(int* numbers, int length)
{
    if(numbers == nullptr || length <= 0)
        throw new std::exception("Invalid parameters");

    int index1 = 0;
    int index2 = length - 1;
    int indexMid = index1;
    while(numbers[index1] >= numbers[index2])
    {
        if(index2 - index1 == 1)
        {
            indexMid = index2;
            break;
        }

        indexMid = (index1 + index2) / 2;

        // 如果下标为 index1、index2 和 indexMid 指向的三个数字相等，
        // 则只能顺序查找
        if(numbers[index1] == numbers[index2]
           && numbers[indexMid] == numbers[index1])
            return MinInOrder(numbers, index1, index2);

        if(numbers[indexMid] >= numbers[index1])
            index1 = indexMid;
        else if(numbers[indexMid] <= numbers[index2])
            index2 = indexMid;
    }

    return numbers[indexMid];
}

int MinInOrder(int* numbers, int index1, int index2)
{
    int result = numbers[index1];
    for(int i = index1 + 1; i <= index2; ++i)
    {
        if(result > numbers[i])
```

```
        result = numbers[i];  
    }  
  
    return result;  
}
```



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/11_MinNumberInRotatedArray



测试用例：

- 功能测试（输入的数组是升序排序数组的一个旋转，数组中有重复数字或者没有重复数字）。
- 边界值测试（输入的数组是一个升序排序的数组，只包含一个数字的数组）。
- 特殊输入测试（输入 nullptr 指针）。



本题考点：

- 考查应聘者对二分查找的理解。本题变换了二分查找的条件，输入的数组不是排序的，而是排序数组的一个旋转。这要求我们对二分查找的过程有深刻的理解。
- 考查应聘者的沟通能力和学习能力。本题面试官提出了一个新的概念：数组的旋转。我们要在很短的时间内学习、理解这个新概念。在面试过程中，如果面试官提出新的概念，那么我们可以主动和面试官沟通，多问几个问题，把概念弄清楚。
- 考查应聘者思维的全面性。排序数组本身是数组旋转的一个特例。另外，我们要考虑到数组中有相同数字的特例。如果不能很好地处理这些特例，就很难写出让面试官满意的完美代码。

2.4.3 回溯法

回溯法可以看成蛮力法的升级版，它从解决问题每一步的所有可能选项里系统地选择出一个可行的解决方案。回溯法非常适合由多个步骤组成的问题，并且每个步骤都有多个选项。当我们在某一步选择了其中一个选项时，就进入下一步，然后又面临新的选项。我们就这样重复选择，直至到达最终的状态。

用回溯法解决的问题的所有选项可以形象地用树状结构表示。在某一步有 n 个可能的选项，那么该步骤可以看成是树状结构中的一个节点，每个选项看成树中节点连接线，经过这些连接线到达该节点的 n 个子节点。树的叶节点对应着终结状态。如果在叶节点的状态满足题目的约束条件，那么我们找到了一个可行的解决方案。

如果在叶节点的状态不满足约束条件，那么只好回溯到它的上一个节点再尝试其他的选项。如果上一个节点所有可能的选项都已经试过，并且不能到达满足约束条件的终结状态，则再次回溯到上一个节点。如果所有节点的所有选项都已经尝试过仍然不能到达满足约束条件的终结状态，则该问题无解。

接下来以面试题 12 为例，分析用回溯法解决问题的过程。由于矩阵的第一行第二个字母'b'和路径"bfce"的第一个字符相等，我们就从这里开始分析。根据题目的要求，我们此时有 3 个选项，分别是向左到达字母'a'、向右到达字母't'、向下到达字母'f'。我们先尝试选项'a'，由于此时不可能得到路径"bfce"，因此不得不回到节点'b'尝试下一个选项't'。同样，经过节点't'也不可能得到路径"bfce"，因此再次回到节点'b'尝试下一个选项'f'。

在节点'f'我们也有 3 个选项，向左、向右都能到达字母'c'、向下到达字母'd'。我们先选择向左到达字母'c'，此时只有一个选择，即向下到达字母'j'。由于此时的路径为"bfcj"，不满足题目的约束条件，我们只好回到上一个节点左边的节点'c'。注意到左边的节点'c'的所有选项都已经尝试过，因此只好再回溯到上一个节点't'并尝试它的下一个选项，即向右到达节点'c'。

在右边的节点'c'我们有两个选择，即向右到达节点's'，或者向下到达节点'e'。由于经过节点's'不可能找到满足条件的路径，我们再选择节点'e'，此时路径上的字母刚好组成字符串"bfce"，满足题目的约束条件，因此我们找到了符合要求的解决方案，如图 2.15 所示。

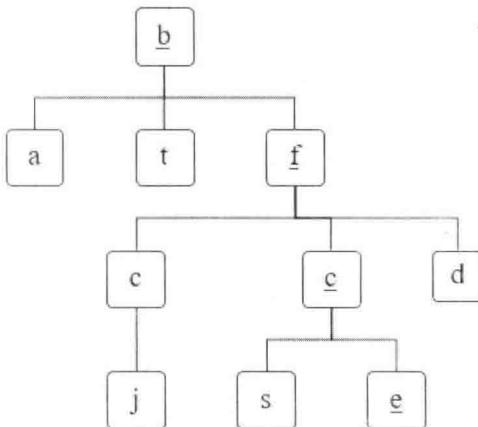


图 2.15 面试题 12 从字母 b 开始的选项组成的树状结构

通常回溯法算法适合用递归实现代码。当我们到达某一个节点时，尝试所有可能的选项并在满足条件的前提下递归地抵达下一个节点。

面试题 12：矩阵中的路径

题目：请设计一个函数，用来判断在一个矩阵中是否存在一条包含某字符串所有字符的路径。路径可以从矩阵中的任意一格开始，每一步可以在矩阵中向左、右、上、下移动一格。如果一条路径经过了矩阵的某一格，那么该路径不能再次进入该格子。例如，在下面的 3×4 的矩阵中包含一条字符串“bfce”的路径（路径中的字母用下画线标出）。但矩阵中不包含字符串“abfb”的路径，因为字符串的第一个字符 b 占据了矩阵中的第一行第二个格子之后，路径不能再次进入这个格子。

a b t g

c f c s

j d e h

这是一个可以用回溯法解决的典型题。首先，在矩阵中任选一个格子作为路径的起点。假设矩阵中某个格子的字符为 ch，并且这个格子将对应于路径上的第 i 个字符。如果路径上的第 i 个字符不是 ch，那么这个格子不可能处在路径上的第 i 个位置。如果路径上的第 i 个字符正好是 ch，那么到相邻的格子寻找路径上的第 i+1 个字符。除矩阵边界上的格子之外，其他格

子都有4个相邻的格子。重复这个过程，直到路径上的所有字符都在矩阵中找到相应的位置。

由于回溯法的递归特性，路径可以被看成一个栈。当在矩阵中定位了路径中前n个字符的位置之后，在与第n个字符对应的格子的周围都没有找到第n+1个字符，这时候只好在路径上回到第n-1个字符，重新定位第n个字符。

由于路径不能重复进入矩阵的格子，所以还需要定义和字符矩阵大小一样的布尔值矩阵，用来标识路径是否已经进入了每个格子。

下面的代码实现了这个回溯算法：

```
bool hasPath(char* matrix, int rows, int cols, char* str)
{
    if(matrix == nullptr || rows < 1 || cols < 1 || str == nullptr)
        return false;

    bool *visited = new bool[rows * cols];
    memset(visited, 0, rows * cols);

    int pathLength = 0;
    for(int row = 0; row < rows; ++row)
    {
        for(int col = 0; col < cols; ++col)
        {
            if(hasPathCore(matrix, rows, cols, row, col, str,
                           pathLength, visited))
            {
                return true;
            }
        }
    }

    delete[] visited;
    return false;
}

bool hasPathCore(const char* matrix, int rows, int cols, int row,
                 int col, const char* str, int& pathLength, bool* visited)
{
    if(str[pathLength] == '\0')
        return true;

    bool hasPath = false;
    if(row >= 0 && row < rows && col >= 0 && col < cols
       && matrix[row * cols + col] == str[pathLength]
       && !visited[row * cols + col])
    {
        ++pathLength;
        visited[row * cols + col] = true;
        hasPath = hasPathCore(matrix, rows, cols, row, col, str,
                              pathLength, visited);
        visited[row * cols + col] = false;
    }
    return hasPath;
}
```

```

visited[row * cols + col] = true;

hasPath = hasPathCore(matrix, rows, cols, row, col - 1,
                      str, pathLength, visited)
    || hasPathCore(matrix, rows, cols, row - 1, col,
                   str, pathLength, visited)
    || hasPathCore(matrix, rows, cols, row, col + 1,
                   str, pathLength, visited)
    || hasPathCore(matrix, rows, cols, row + 1, col,
                   str, pathLength, visited);

if(!hasPath)
{
    --pathLength;
    visited[row * cols + col] = false;
}

return hasPath;
}

```

当矩阵中坐标为 (row, col) 的格子和路径字符串中下标为 $pathLength$ 的字符一样时，从 4 个相邻的格子 $(row, col-1)$ 、 $(row-1, col)$ 、 $(row, col+1)$ 和 $(row+1, col)$ 中去定位路径字符串中下标为 $pathLength+1$ 的字符。

如果 4 个相邻的格子都没有匹配字符串中下标为 $pathLength+1$ 的字符，则表明当前路径字符串中下标为 $pathLength$ 的字符在矩阵中的定位不正确，我们需要回到前一个字符 ($pathLength-1$)，然后重新定位。

一直重复这个过程，直到路径字符串上的所有字符都在矩阵中找到合适的位置（此时 $str[pathLength] == '\0'$ ）。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/12_StringPathInMatrix



测试用例：

- 功能测试（在多行多列的矩阵中存在或者不存在路径）。
- 边界值测试（矩阵只有一行或者只有一列；矩阵和路径中的所有字母都是相同的）。
- 特殊输入测试（输入 `nullptr` 指针）。



本题考点：

- 考查应聘者对回溯法的理解。通常在二维矩阵上找路径这类问题都可以应用回溯法解决。
- 考查应聘者对数组的编程能力。我们一般都把矩阵看成一个二维的数组。只有对数组的特性充分了解，才有可能快速、正确地实现回溯法的代码。

面试题 13：机器人的运动范围

题目：地上有一个 m 行 n 列的方格。一个机器人从坐标 $(0, 0)$ 的格子开始移动，它每次可以向左、右、上、下移动一格，但不能进入行坐标和列坐标的数位之和大于 k 的格子。例如，当 k 为 18 时，机器人能够进入方格 $(35, 37)$ ，因为 $3+5+3+7=18$ 。但它不能进入方格 $(35, 38)$ ，因为 $3+5+3+8=19$ 。请问该机器人能够到达多少个格子？

和前面的题目类似，这个方格也可以看作一个 $m \times n$ 的矩阵。同样，在这个矩阵中，除边界上的格子之外，其他格子都有 4 个相邻的格子。

机器人从坐标 $(0, 0)$ 开始移动。当它准备进入坐标为 (i, j) 的格子时，通过检查坐标的数位和来判断机器人是否能够进入。如果机器人能够进入坐标为 (i, j) 的格子，则再判断它能否进入 4 个相邻的格子 $(i, j-1)$ 、 $(i-1, j)$ 、 $(i, j+1)$ 和 $(i+1, j)$ 。因此，我们可以用如下的代码来实现回溯算法：

```

int movingCount(int threshold, int rows, int cols)
{
    if(threshold < 0 || rows <= 0 || cols <= 0)
        return 0;

    bool *visited = new bool[rows * cols];
    for(int i = 0; i < rows * cols; ++i)
        visited[i] = false;

    int count = movingCountCore(threshold, rows, cols,
                                0, 0, visited);

    delete[] visited;
    return count;
}

int movingCountCore(int threshold, int rows, int cols, int row,
                    int col, bool* visited){

```

```

int count = 0;
if(check(threshold, rows, cols, row, col, visited))
{
    visited[row * cols + col] = true;

    count = 1 + movingCountCore(threshold, rows, cols,
                                row - 1, col, visited)
            + movingCountCore(threshold, rows, cols,
                                row, col - 1, visited)
            + movingCountCore(threshold, rows, cols,
                                row + 1, col, visited)
            + movingCountCore(threshold, rows, cols,
                                row, col + 1, visited);
}

return count;
}

```

下面的函数 check 用来判断机器人能否进入坐标为(row, col)的方格，而函数 getDigitSum 用来得到一个数字的数位之和。

```

bool check(int threshold, int rows, int cols, int row, int col,
           bool* visited)
{
    if(row >= 0 && row < rows && col >= 0 && col < cols
       && getDigitSum(row) + getDigitSum(col) <= threshold
       && !visited[row* cols + col])
        return true;

    return false;
}

int getDigitSum(int number)
{
    int sum = 0;
    while(number > 0)
    {
        sum += number % 10;
        number /= 10;
    }

    return sum;
}

```



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/13_RobotMove



测试用例：

- 功能测试（方格为多行多列； k 为正数）。
- 边界值测试（方格只有一行或者只有一列； k 等于 0）。
- 特殊输入测试（ k 为负数）。



本题考点：

- 考查应聘者对回溯法的理解。通常物体或者人在二维方格运动这类问题都可以用回溯法解决。
- 考查应聘者对数组编程的能力。我们一般都把矩阵看成一个二维的数组。只有对数组的特性充分了解，才有可能快速、正确地实现回溯法的代码编写。

2.4.4 动态规划与贪婪算法

动态规划现在是编程面试中的热门话题。如果面试题是求一个问题的最优解（通常是求最大值或者最小值），而且该问题能够分解成若干个子问题，并且子问题之间还有重叠的更小的子问题，就可以考虑用动态规划来解决这个问题。

我们在应用动态规划之前要分析能否把大问题分解成小问题，分解后的每个小问题也存在最优解。如果把小问题的最优解组合起来能够得到整个问题的最优解，那么我们可以应用动态规划解决这个问题。

例如在面试题 14 中，我们如何把长度为 n 的绳子剪成若干段，使得得到的各段长度的乘积最大。这个问题的目标是求剪出的各段绳子长度的乘积最大值，也就是求一个问题的最优解——这是可以应用动态规划求解的问题的第一个特点。

我们把长度为 n 的绳子剪成若干段后得到的乘积最大值定义为函数 $f(n)$ 。假设我们把第一刀剪在长度为 i ($0 < i < n$) 的位置，于是把绳子剪成了长度分别为 i 和 $n-i$ 的两段。我们要想得到整个问题的最优解 $f(n)$ ，那么要同样用最优化的方法把长度为 i 和 $n-i$ 的两段分别剪成若干段，使得它们各自剪出的每段绳子的长度乘积最大。也就是说整体问题的最优解是依赖各

个子问题的最优解——这是可以应用动态规划求解的问题的第二个特点。

我们把大问题分解成若干个小问题，这些小问题之间还有相互重叠的更小的子问题——这是可以应用动态规划求解的问题的第三个特点。还是以面试题 14 为例，假设绳子最初的长度为 10，我们可以把绳子剪成长度分别为 4 和 6 的两段，也就是 $f(4)$ 和 $f(6)$ 都是 $f(10)$ 的子问题。接下来分别求解这两个子问题。我们可以把长度为 4 的绳子剪成均为 2 的两段，即 $f(2)$ 是 $f(4)$ 的子问题。同样，我们也可以把长度为 6 的绳子剪成长度分别为 2 和 4 的两段，即 $f(2)$ 和 $f(4)$ 都是 $f(6)$ 的子问题。我们注意到 $f(2)$ 是 $f(4)$ 和 $f(6)$ 公共的更小的子问题。

由于子问题在分解大问题的过程中重复出现，为了避免重复求解子问题，我们可以用从下往上的顺序先计算小问题的最优解并存储下来，再以此为基础求取大问题的最优解。从上往下分析问题，从下往上求解问题，这是可以应用动态规划求解的问题的第四个特点。在应用动态规划解决问题的时候，我们总是从解决最小问题开始，并把已经解决的子问题的最优解存储下来（大部分面试题都是存储在一维或者二维数组里），并把子问题的最优解组合起来逐步解决大的问题。

在应用动态规划的时候，我们每一步都可能面临若干个选择。在求解面试题 14 时，我们在剪第一刀的时候就有 $n-1$ 个选择。我们可以剪在长度为 $1, 2, \dots, n-1$ 的任意位置。由于我们事先不知道剪在哪个位置是最优的解法，只好把所有的可能都尝试一遍，然后比较得出最优的剪法。如果用数学的语言来表示，这就是 $f(n)=\max(f(i) \times f(n-i))$ ，其中 $0 < i < n$ 。

贪婪算法和动态规划不一样。当我们应用贪婪算法解决问题的时候，每一步都可以做出一个贪婪的选择，基于这个选择，我们确定能够得到最优解。还是以面试题 14 “剪绳子”为例，如果绳子的长度大于 5，则每次都剪出一段长度为 3 的绳子。如果剩下的绳子的长度仍然大于 5，则接着剪出一段长度为 3 的绳子。接下来重复这个步骤，直到剩下的绳子的长度小于 5。剪出一段长度为 3 的绳子，就是我们在每一步做出的贪婪选择。为什么这样的贪婪选择能得到最优解？这是我们应用贪婪算法时都需要问的问题，需要用数学方式来证明贪婪选择是正确的。

面试题 14：剪绳子

题目：给你一根长度为 n 的绳子，请把绳子剪成 m 段 (m, n 都是整数， $n > 1$ 并且 $m > 1$)，每段绳子的长度记为 $k[0], k[1], \dots, k[m]$ 。请问 $k[0] \times k[1] \times \dots \times k[m]$ 可能的最大乘积是多少？例如，当绳子的长度是 8 时，我们把它剪成长度分别为 2、3、3 的三段，此时得到的最大乘积是 18。

我们有两种不同的方法解决这个问题。先用常规的需要 $O(n^2)$ 时间和 $O(n)$ 空间的动态规划的思路，接着用只需要 $O(1)$ 时间和空间的贪心算法来分析解决这个问题。

❖ 动态规划

首先定义函数 $f(n)$ 为把长度为 n 的绳子剪成若干段后各段长度乘积的最大值。在剪第一刀的时候，我们有 $n-1$ 种可能的选择，也就是剪出来的第一段绳子的可能长度分别为 $1, 2, \dots, n-1$ 。因此 $f(n) = \max(f(i) \times f(n-i))$ ，其中 $0 < i < n$ 。

这是一个从上至下的递归公式。由于递归会有很多重复的子问题，从而有大量不必要的重复计算。一个更好的办法是按照从下而上的顺序计算，也就是说我们先得到 $f(2), f(3)$ ，再得到 $f(4), f(5)$ ，直到得到 $f(n)$ 。

当绳子的长度为 2 时，只可能剪成长度都为 1 的两段，因此 $f(2)=1$ 。当绳子的长度为 3 时，可能把绳子剪成长度分别为 1 和 2 的两段或者长度都为 1 的三段，由于 $1 \times 2 > 1 \times 1 \times 1$ ，因此 $f(3)=2$ 。

下面是这一思路对应的参考代码：

```
int maxProductAfterCutting_solution1(int length)
{
    if(length < 2)
        return 0;
    if(length == 2)
        return 1;
    if(length == 3)
        return 2;

    int* products = new int[length + 1];
    products[0] = 0;
    products[1] = 1;
    products[2] = 2;
    products[3] = 3;

    int max = 0;
```

```

for(int i = 4; i <= length; ++i)
{
    max = 0;
    for(int j = 1; j <= i / 2; ++j)
    {
        int product = products[j] * products[i - j];
        if(max < product)
            max = product;

        products[i] = max;
    }
}

max = products[length];
delete[] products;

return max;
}

```

在上述代码中，子问题的最优解存储在数组 `products` 里。数组中第 i 个元素表示把长度为 i 的绳子剪成若干段之后各段长度乘积的最大值，即 $f(i)$ 。我们注意到代码中第一个 `for` 循环变量 i 是顺序递增的，这意味着计算顺序是自下而上的。因此在求 $f(i)$ 之前，对于每一个 j ($0 < j < i$) 而言， $f(j)$ 都已经求解出来了，并且结果保存在 `products[j]` 里。为了求解 $f(i)$ ，我们需要求出所有可能的 $f(j) \times f(i-j)$ 并比较得出它们的最大值。这就是代码中第二个 `for` 循环的功能。

◆ 贪婪算法

如果我们按照如下的策略来剪绳子，则得到的各段绳子的长度的乘积将最大：当 $n \geq 5$ 时，我们尽可能多地剪长度为 3 的绳子；当剩下的绳子长度为 4 时，把绳子剪成两段长度为 2 的绳子。这种思路对应的参考代码如下：

```

int maxProductAfterCutting_solution2(int length)
{
    if(length < 2)
        return 0;
    if(length == 2)
        return 1;
    if(length == 3)
        return 2;

    // 尽可能多地剪去长度为 3 的绳子段
    int timesOf3 = length / 3;

    // 当绳子最后剩下的长度为 4 的时候，不能再剪去长度为 3 的绳子段。

```

```

// 此时更好的方法是把绳子剪成长度为2的两段，因为2×2>3×1
if(length - timesOf3 * 3 == 1)
    timesOf3 -= 1;

int timesOf2 = (length - timesOf3 * 3) / 2;

return (int)(pow(3, timesOf3)) * (int)(pow(2, timesOf2));
}

```

接下来我们证明这种思路的正确性。首先，当 $n \geq 5$ 的时候，我们可以证明 $2(n-2) > n$ 并且 $3(n-3) > n$ 。也就是说，当绳子剩下的长度大于或者等于 5 的时候，我们就把它剪成长度为 3 或者 2 的绳子段。另外，当 $n \geq 5$ 时， $3(n-3) \geq 2(n-2)$ ，因此我们应该尽可能地多剪长度为 3 的绳子段。

前面证明的前提是 $n \geq 5$ 。那么当绳子的长度为 4 呢？在长度为 4 的绳子上剪一刀，有两种可能的结果：剪成长度分别为 1 和 3 的两根绳子，或者两根长度都为 2 的绳子。注意到 $2 \times 2 > 1 \times 3$ ，同时 $2 \times 2 = 4$ ，也就是说，当绳子长度为 4 时其实没有必要剪，只是题目的要求是至少要剪一刀。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/14_CuttingRope



测试用例：

- 功能测试（绳子的初始长度大于 5）。
- 边界值测试（绳子的初始长度分别为 0、1、2、3、4）。



本题考点：

- 考查应聘者的抽象建模能力。应聘者需要把一个具体的场景抽象成一个能够用动态规划或者贪婪算法解决的模型。更多关于抽象建模能力的讨论请参考本书 6.4 节。
- 考查应聘者对动态规划和贪婪算法的理解。能够灵活运用动态规划解决问题的关键是具备从上到下分析问题、从下到上解决问题的能力，而灵活运用贪婪算法则需要扎实的数学基本功。

2.4.5 位运算

位运算是把数字用二进制表示之后，对每一位上 0 或者 1 的运算。二进制及其位运算是现代计算机学科的基石，很多底层的技术都离不开位运算，因此与位运算相关的题目也经常出现在面试中。我们在日常生活中习惯了十进制，很多人看到二进制及位运算都觉得很难适应。

理解位运算的第一步是理解二进制。二进制是指数字的每一位都是 0 或者 1。比如十进制的 2 转换成二进制之后是 10，而十进制的 10 转换成二进制之后是 1010。在程序员圈子里有一则流传了很久的笑话，说世界上有 10 种人，一种人知道二进制，而另一种人不知道二进制……

除了二进制，我们还可以把数字表示成其他进制，比如表示时间分秒的六十进制等。针对不太熟悉的进制，已经出现了不少很有意思的面试题。比如：

在微软产品 Excel 中，用 A 表示第 1 列，B 表示第 2 列……Z 表示第 26 列，AA 表示第 27 列，AB 表示第 28 列……以此类推。请写出一个函数，输入用字母表示的列号编码，输出它是第几列。

这是一道很新颖的关于进制的题目，其本质是把十进制数字用 A~Z 表示成二十六进制。如果想到这一点，那么解决这个问题就不难了。

其实二进制的位运算并不是很难掌握，因为位运算总共只有 5 种运算：与、或、异或、左移和右移。与、或和异或运算的规律我们可以用表 2.1 总结如下。

表 2.1 与、或、异或的运算规律

与 (&)	$0 \& 0 = 0$	$1 \& 0 = 0$	$0 \& 1 = 0$	$1 \& 1 = 1$
或 ()	$0 0 = 0$	$1 0 = 1$	$0 1 = 1$	$1 1 = 1$
异或 (^)	$0 ^ 0 = 0$	$1 ^ 0 = 1$	$0 ^ 1 = 1$	$1 ^ 1 = 0$

左移运算符 $m << n$ 表示把 m 左移 n 位。在左移 n 位的时候，最左边的 n 位将被丢弃，同时在最右边补上 n 个 0。比如：

$00001010 << 2 = 00101000$

$10001010 << 3 = 01010000$

右移运算符 $m >> n$ 表示把 m 右移 n 位。在右移 n 位的时候，最右边的