程序员面试
白皮书

AN ULTIMATE GUIDE TO CODING INTERVIEWS

逸超 虾米 笑笑 董飞 著 Ruthia 封面

目录

内容提要

本书是程序员和 IT 从业人员的面试求职指南。本书遵从大多数面试参考图书的组织方式，结合实例，按照常见的数据结构、算法以及计算机基础知识进行章节划分。每一章的“知识要点”部分介绍章节涉及的相关知识点，回顾重要的基础知识点；“模式识别”部分给出一些例题，帮助大家总结解决相关问题的常见方法，并且通过分析问题中的关键信息，教授读者如何从题目中分析题型和解题方法。程序员面试是对于面试者计算机知识的全面检测，因此，本书设有专门的章节覆盖了网络、操作系统、编译器、算法和数据结构等各个领域的知识。

本书作者来自硅谷一线的 IT 公司，书中包含了作者亲身的经历和体验，书中收集的题目部分来自互联网上分享的面试经验、在线编程网站 leetcode，以及一些著名的面试参考资料。本书适合想要从事正规的程序员、架构师以及相关 IT 公司的专业人士和学生参考，尤其适合那些以一线 IT 外企或互联网公司为求职目标的读者阅读。

作者简介

逸超

竞赛保送国内 Top2 大学本科，毕业后前往加州大学攻读硕士学位，期间获得全额奖学金。毕业时获得十多个知名科技公司 offer，现在在硅谷就职于顶尖科技公司，从事软件开发工作。擅长总结面试方法，乐于帮助朋友准备面试。

虾米

国内 TOP2 高校毕业，赴美硕士毕业后跨专业应聘程序员工作，有一套自己的面试方法学。现于硅谷市值最高的公司之一任软件工程师，业余时间开设面试讲座，在圈内小有名气。

笑笑

国内 top2 大学本科,UCSD 计算机硕士。现苹果公司软件工程师。曾拿到多个硅谷顶级软件公司的 offer，面试经验相当丰富。

董飞

本科南开大学，硕士杜克大学计算机系毕业。在攻读硕士期间，从事跟 Hadoop 大数据相关的研究项目，在 VLDB，SOCC 顶尖数据库大会发表过论文， 先后在创业公司酷迅，百度基础架构组，Amazon 云计算部门，Linkedin 担任高级工程师，负责过垂直搜索引擎，百度云计算平台研发，广告系统和在线教育平台的架构。在大数据领域业界研究多年，涉及 Hadoop 调优，分布式框架，Data Pipeline，实时系统。目前在线教育创业公司 Coursera 从事数据工程师工作。

在多年工作中，除了对技术的不懈追求，也积累了大量的面试经验，拿到北美热门公司 10+ offer。在 Linkedin 期间，也积极参与面试过近 200 人，全公司前三的面试官，乐于分享并帮助很多人成功求职，实现目标。

前言

我的故事，你的故事

这是一本我希望自己在大学就能看到的一本书。在我读大学的时候，跟很多同学一样也走过不少弯路：艰难地啃着计算机必修课，被里面的指针和对象搞的晕头转向，对 i++和++i 区别死活不理解，为操作系统的调度策略而抓狂，很难对专业书产生兴趣，只是想着去应付考试和学分。当我们毕业后，虽然有了计算机学位和实际工作经历，但其实还是不知道最想得到什么。我当时心中有个情节，就是希望能去最好的公司，比如 Google，因为听说那里面都是最聪明的人，有着最好的待遇和福利，做着最有影响力的事情。当我鼓起勇气去尝试的时候，我失败了，并且要进入 6 个月的冷藏期（失去在一段时期内继续面试的机会）。而我发现当时的面试表现是那么的稚嫩和糟糕，我开始懊恼，开始反省，开始重新准备，这样才有了我下一个面试机会和其他的 Offer。当然如果一切可以重新再来，我可以更有自信更好地发挥我的能力，也许当时的理想公司就会给我 Offer。

有些朋友会认为只要能混过面试，拿到工作之后有的是时间可以继续学习。这话听起来有道理，但我在这里给出一个反例：我记得多年前第一份实习的任务是对某个大型应用程序进行性能测试。我不知道如何创建一个用户界面，然后随意定义文本字段、菜单和按钮。我

不知道如何用线程来思考，我错误调用整个缓存和线程池。我也不知道如何去做代码维护，并且没有单元测试和编写基本的文档，最后我还是写了几千行的 Java 代码，这是个无法维护的巨大的类。而如果我在工作之前多一些积累和看一些代码，或许不会那么尴尬。

也有些同学说面试中只会考没有用的算法，这跟工作没有任何直接关系。我同意工作中大多数情况是不会用到复杂的算法的，但如果你没有过硬的基本功，在面对一些新情况的时候，你就很难举一反三，灵活运用了。我记得我在第一家公司第一个项目是在一个新城市中增加新的排序选项来选择上市的所有租房。这是一个紧急任务，上司希望我尽快熟悉代码库，我当时也顶着压力，做到了一周上线。不久后，我就在总裁面前演示：我看着他点击了某个区的房源，选择了新的排序选项，结果花了几分钟去加载页面。我之前也验证过应该没问题，平常只需要几秒钟的时间。我当时满头大汗，真是搞砸了。那天晚上，我思考了很久才想通。我选择的新代码做两个数据库调用需要遍历其中的每一个，它需要的 n * log n 比较的次数，而对于那个区域，其中大约有 n =1000 个房源，那么大约要 2 万次数据库调用才能完成一个页面加载。当然，知道了原因，优化起来就简单了，通过调用缓冲，把数据切成更小块，做数据量的控制，最后性能提高了 100 倍，所以说系统优化是离不开算法和扎实计算机基本功的。

其实每个工程师都讨厌 Bug、代码不整洁、性能太差、用户界面不人性化等，这些都是一些技术细节，是可以慢慢体会和提高，总能找到答案去改进的。但在工作中，我应该学习和使用什么样的技术？为什么要自动化测试？如何搭建一个产品，看起来比较靠谱？我怎么去选择一份工作？如果我在一家大公司工作，如何跳槽到创业公司？我如何谈判取得更多的薪金或奖金？什么是股票激励？这些问题倒是更棘手，我也会在第 1 章给出一些介绍和辅助工具。

回看我走过的一路艰辛，我尝试反思学到了什么，我发现其中大部分经验都来自痛苦的反复试验的结果。当我意识到成千上万的面试者或者 IT 开发人员正在经历同样的试错，犯同样的错误， 我觉得应该做一些更有意义的事情：这本书就是一个工具。诚然，有些教训只能从在自己的错误中学习，但我希望本书能够帮助你从别人的经验中获得通向成功的捷径。

现状

俗话说凡事要“顺势而为”，找工作亦是如此。现如今，借助手机网络带宽的快速提升以及移动互联网的概念，原先在 PC 平台才能开展的服务一下被冠上了“移动”二字。在手机上使用服务和计算机上使用服务，在本质上并没有太大区别，但关键在于手机更具有“私密性”和“便携性”，大大增加了客户粘性和使用时间，进而会对服务提供商产生更大的依赖性。

在中国，互联网公司中百度、腾讯、阿里巴巴“三架马车”分别把持搜索、社交游戏和电商平台三大主战场，互相竞争。而硅谷更是百花齐放：Apple、Google、Microsoft 致力于打造自己的生态圈，完成硬件、软件、服务的闭环；而 Facebook、Twitter、LinkedIn 等为代表的社交平台也迅速通过巨大的用户群体完成“圈地运动”，将管辖区域内的用户导向各个实体商户；更有 Uber、Airbnb 等新兴公司，致力于通过移动互联网思维改变人们的实际生活。

相比于 2000 年左右的互联网泡沫，这次的计算机高潮来势更为凶猛：不单单提供资讯、门户、电邮等虚拟线上服务，而是直接破坏性地侵入传统行业，以更高的效率改变原有行业。这就是为什么许多新兴科技企业号称是技术公司，但实际上提供着传统行业的服务。当前的趋势也会逼迫着传统公司作出改变，引入更多计算机人才，利用云计算、机器学习等新手段与新兴公司竞争。最简单的例子如 Wal-Mart Labs，它以一个科技智囊的角色隶属于 Wal-Mart，通过计算机技术分析，优化 Wal-Mart 的营运效率。金融、银行、地产、石油、制造、电子硬件等各个行业也纷纷引入计算机技术，大大创造了从业人员的工作机会。

这样的趋势对你我有什么影响？人才需求的极速扩张意味着找工作难度降低，并且待遇也是水涨船高。举例来说，在美国，硕士毕业加入 Apple、Google、Facebook 等公司起薪至少 10 万美元，外加股票期权。更不要说加入最火的创业公司，三四年后一旦上市就可以提前退休，或者把工作作为兴趣。在国内，阿里巴巴上市也造就了成百上千个千万富翁，即使是上市前一两年刚加入的新人，也拿到了 100 万人民币左右的股权。

如果说这些职位难度太高，对于你遥不可及，或者需要很长的准备时间，那也许对于上面的文字你只会一笑而过。但是，如果告诉你

通过正确的方式，做好面试准备，上面所说的职位触手可及，是不是听起来更有吸引力了呢？事实就是，由于软件工程师的职业特性、面试要求及局限性，以及市场需求等因素，程序员求职是一种比较具有应试性、相对容易找到门道的简单职业道路。不乏相关或不那么相关专业的毕业生通过半年到一年的努力拿到理想的 IT 行业的相关工作的事例，由此可见，挑战不在于“能”和“不能”，而是如何通过正确的方法、迅速地赶上潮流分一杯羹。

目的

本书的目的并不在于代替课本教材，系统性地讲授计算机技术，而是作为一本工具书，创建一个实际的、可操作的面试方法论教程，提供一条快速熟悉技术面试题目的捷径，并且针对不同类型的题目，归纳总结解题方法。

程序员面试是对于面试者计算机知识的全面检测，因此，关于计算机诸如网络、操作系统、编译器、算法和数据结构等各个领域的系统性学习不可或缺。但是考虑到面试的局限性，诸如时间限制，面试官对于面试者的熟悉程度等，在白板（或者白纸）上写程序解决一些算法问题成为面试官较为青睐的方法之一。由于该面试方法比较机械，相对容易准备，也最适合总结一些方法论，所以本书的目的就在于传授白板写代码的准备技巧，帮助大家通过面试。在本书中，我们将会遇到的题目、技术，都是来自于我们实际面试过的一些炙手可热的硅谷公司和我们自己作为面试官的一些心得和宝贵经验。正如参加 GRE 考试一样，关于考试技巧的书籍并不能让一个完全不懂英语的人通过考试，但是可以使得英语基础合格的人如虎添翼，大大增加通过考试的几率。这就是本书的全部意义所在。

特色

市面上关于程序员面试的参考书也不少，但是我们认为这些书的关键问题在于它们大多是教你“怎么做”，但很少涉及“为什么这么做”。于是，读者往往会觉得书中的解法十分精妙，但是在面试的时候完全想不起来用哪种方法解决问题。其根本原因在于，这些参考书

代替你做了最关键的一步：判断用什么方法解决当前的问题。

本书遵从大多数面试参考书的构成方式，结合实例，按照常见的数据结构、算法以及计算机基础知识进行章节划分，但是，本书着眼于如何进行“模式识别”，通过分析为什么这个问题被划分到这个章节，来教大家如何判断实际面试问题的类型，并且顺水推舟地得出解决问题的方法。每一章的“知识要点”介绍章节涉及的相关知识点，回顾通常出现在教材中的重点内容；“模式识别”给出一些实例，帮助大家总结解决相关问题的常见方法，并且通过分析问题中的关键信息，教授大家如何从题目中得到关于题型分类以及解题方法的蜘丝马迹。

本书收集的题目部分来自互联网上分享的面试经验、在线编程网站 leetcode，以及面试参考资料 Cracking the Coding Interview 和 Element of Programming Interview。如果你认真准备过面试，可能会对题目有似曾相识的感觉。但本书的关键不是教会你做一道题目，而是教会分析题目、解决题目的方法，从而学会解一类题目。具体的题目不是关键，从题目到方法的思维过程，是本书努力想传达的重点；最后，“工具箱”部分给出该章涉及的 C++/Java 类，它们的常见函数及使用方法，还提供一些标准库函数，以及相关参考资料或扩展阅读。

如果你有志于投身到滚滚的 IT 互联网浪潮中，无论你是一个学生，还是初级程序员，不论你以后是定位于架构师还是项目经理，你都需要一块敲门砖，那么这本书就是为你量身定做的。作为有着在国内外创业公司和一线公司经历的过来人，我们希望本书能给你职业生涯上添砖加瓦，帮助大家到达理想的彼岸。

最后，我们特别感谢 Ruthia 百忙之中抽空帮助设计封面。正如封面中锤子所象征的那样，每个人都需要付出持之以恒的努力才能获得成绩，在此与读者共勉。

由于水平有限，编写过程中难免产生疏漏。如果在阅读本书的过程中你有任何建议和问题，请联系我们：UltimateGuideToCodingInterview@gmail.com。

第1章　简历、面试和 Offer

整个招聘流程主要如下：申请某个公司你感兴趣的职位，投出你的简历。HR 从简历库中筛选符合要求的简历，安排面试。面试主要分为电话面试和现场面试，如果面试表现优秀，HR 会同你讨论待遇、福利、起始时间等具体信息。针对上述流程，让我们一一分析其中的关键环节。

1.1 简历

简历是求职的第一关，尽管简历不能决定最后的录取结果，但是一份结构清晰、内容充实且具有针对性的简历，可以给你带来宝贵的面试机会。本节将从格式、内容安排和描述技巧等方面介绍如何生成一份对 HR 有吸引力的简历。可以根据不同的职位描述适当更改简历的某些内容，但不建议准备太多份区别过大的简历，以免自己投递的时候产生错误。如果你真的需要这么多简历以应对不同职位，则应当考虑下自己的优势在哪些方面，适当减少求职目标。**注意，下文的描述注重在北美求职的简历结构，其中大部分要点具有普适性，但是如果在国内求职，有些地方可能需要根据实际要求进行更改。**

1.1.1 格式

除非你想面试用户体验或者设计相关的职位，否则简历的格式尽量以简介清晰为主。标题及名字等重要信息可以适当加粗或者增大字体，其他文字描述尽量统一字号。字体选定一种即可，不建议用不同的字体去突出不同的内容：不同的字体会显得版面杂乱，并且对排版造成困难。英文简历 Calibri 是比较适宜的字体，中文可以用宋体。简历可以用 Word、Latex、Pages 或者 InDesign 制作，但是最后务必导出成 PDF，确保在不同的平台上，对方看到的格式一致。

1.1.2 内容安排

简历是一个提供信息、展示自己的平台。对于刚毕业、没有什么工作经验的人来说，简历一般控制在一页为宜，如果是 PhD，需要适当列举一些相关的文章发表信息，那么简历可以扩展至两页。这里就给一个示例，它在一页纸的长度中精炼地描述了求职者的教育背景、

实习经历、技能和专业背景，如图 1-1 所示。

图 1-1 一页纸的简历

简历的第一要务是让对方能够认识并联系上你，所以名字、邮件、电话、地址等基本信息缺一不可。特别的，对于在美国本地求职的人而言，邮编很重要，因为网上的申请系统往往会根据邮编将申请者按地域划分。对于很多公司而言，他们会优先考虑本地的申请者。所以当你申请的公司在当地有总部/分部时，务必写上本地的邮编。

Anakin Skywalker

CONTACT INFORMATION	School of Computer Science, Carnegie Mellon University 5000 Forbes Avenue Pittsburgh, PA 15213	Tel: (XXX) XXX-XXXX E-mail: xxx@andrew.cmu.edu Website: http://www.website.com

EDUCATION

Carnegie Mellon University, Pittsburgh, PA
Master of Science in Information Technology August 2013 – December 2014 (expected)
- Specialization: XXX, GPA: XXX
- Coursework: Multimedia Database and Data mining, Machine Learning with Large Datasets, Distributed Systems, Web Application Development, Operating Systems

XXX University, Beijing, China (PR)
Bachelor of Science September 2009 – July 2013
- Major: Computer Science

PROFESSIONAL EXPERIENCE

XXX Inc., Palo Alto, CA
Software Engineer Intern May 2012 – August 2012
- Benchmarked XXX performance by implementing a XXX with XXX.
- Profiled and explored ways to improve performance for XXX. Reduced the processing latencies by XX% through XXX and YYY.

XXX Inc., Beijing, China (PR)
Software Engineer Intern March 2010 – June 2010
- Designed and implemented a XXX system to automatically XXX...
- Built a XXX based recommendation system to automatically recommendation ads to customers with XXX and YYY. This system achieves a XXX precision and XXX recall.
- Improved a XXX algorithm to automatically filter noise information from extracted webpages and improved the precision of this algorithm by 50% through replacing XXX module with YYY module.

XXX Inc., Shenzhen, China (PR)
Software Engineer Intern August 2008 – November 2009
- Designed and implemented a XXX system, which consisted of XXX and YYY. Experiments showed that this system was able to XXX.

SKILLS C, C++, Java, Python, .NET, Matlab, Bash, SQL, Linux, Hadoop.

HONORS AND AWARDS

XXX Scholarship, XXX University. 2013
YYY Award, XXX University. 2012

这里有个小技巧，由于绝大部分科技公司都在加州，特别是北加州硅谷附近，所以如果有朋友在加州的话不妨写朋友的地址。这样做的确可以提高被选中面试的几率，甚至有些小公司可能会跳过一轮电话面试直接邀请你去公司面试。现在绝大部分面试过程都是通过电话或邮件确定，除了最后给你发 Offer，招聘的中间部分都不会给你提供地址寄送任何文件。如果公司决定给你 Offer，你也完全有机会和 HR 确认/更新你的实际地址。这样做的缺点是，可能去公司面试的时候你需要自己安排机票住宿，如果真到了这一步，权且就当花钱买个机会了。

基本信息之外，对于刚毕业的学生而言最重要的信息包括学校、专业和学位。学习成绩对于大部分公司而言只是加分项，如果 GPA 低于 3.3，可以考虑不包括成绩。当然，如果 GPA 是你的亮点之一，那也完全可以包含此信息。在这方面，Oracle 是出了名的名校控，如果你高 GPA、名校出身，基本上半只脚已经踏入了 Oracle。相对而言，其他公司并没有这样的要求。此外，对学生而言可以列举一些在校学过的相关课程。只需要列举课程名即可，内容数量以 5 项左右为宜，课程名尽量选择大家熟知的，比如算法、数据结构、操作系统等，如果是研究生课程，可以加上“高级（advanced）”关键字。

简历中另一个十分关键的内容是之前做过的项目描述，包括工作经历、实习经历或者课程项目经历，这些部分的目的在于展示你具有相关经验，具有较强的技术实力，能够加入团队一起完成一个复杂的项目。这里，我们主要介绍如何选择合适的项目，在描述技巧部分，会进一步介绍如何描述使得你的项目更有趣。就项目内容而言，你需要优先选择和职位描述相关的项目，相关性包括：需要相似的知识，需要使用相同的软件或开发环境，需要类似的编程语言或编程模式，需要实现类似的功能等等。这些项目最能体现你的价值，使得你从一堆申请者中脱颖而出。如果没有相似的经历，那么你可以列举一些比较复杂的项目，突出你的综合能力。这些可以包括：毕业设计、课程的学期作业、网上参与合作的开源项目等。列举的目的主要是突出你的技术水准优秀，具有与他人合作的能力。就项目种类而言，你需要优先选择业界的工作实习经历，只要它们和你所求职位的描述大体在同一个行业即可。毕竟，有实际的工作经验可以传达下列信息：你对业界有一定的了解，熟悉业界的开发模式和开发周期，能够适应公司的工作环境。

简历的最后部分可以用来列举你的技能，包括熟悉的编程语言、开发环境、技术强项等。这部分的目的在于让 HR 能够从简历库中匹配到你的信息。通常，每个职位都有一些技术要求，HR 会通过查询关键字，从简历库中选择匹配程度高的简历。这部分可以大大增加简历被匹配上的可能性。列举的原则是，并不需要特别熟悉，只要实际用过就可以在此列举。注意，如果你有幸被安排了面试，一定要回到这个部分，确保你所列举的部分至少都能回答一些基本的问题，千万不要给人做假的感觉。往往，HR 并不一定了解你项目部分的描述是否与职位要求一致，因此，你这部分所列举的技能需要尽量用业界标准的语言，列举名词即可。

1.1.3 描述技巧

描述的技巧主要体现在项目描述方面。项目描述主要突出你做了什么，实现了什么样的目的。项目名称一般需要让读者大致了解你做了什么，然后以如下模版，“通过……开发方式（或者技术），做了……，最终实现了……的结果”，描述项目的具体内容。就英文简历而言，一般以过去时为主，以动词开头，描述你做过了什么，实现了什么目的。举例如下：

Software Engineering Internship，XX Company，6/1/2014 – 9/1/2014

Interned with the server team.

Implemented a distributed access control algorithm in C++, which improved login time by around 50%.

英文简历常见的另一个问题是如何翻译专有名词，比如在国内大学获得的种种荣誉等。此时，一定要参考网络资料，确保读者的理解和你想要表达的意思一致。甚至可以用一句话简单描述这是怎样的荣誉，或者用百分比表示只有 top 的学生才能获得该项荣誉。举例如下：

Excellent Student Award（top 3%）

Granted to recognize overall outstanding performance.

另一个很好学习途径，就是去 LinkedIn 上看一些优秀人士的个人主页，特别是你想去的公司的工程师的背景。一般都会很清晰地构建他的目标和过去经历，不需要照搬内容，但可以给你启发，当你去投这些公司时，你就知道什么样的简历更容易被他们欣赏。

1.2 面试

根据面试的不同对象，在招聘过程中你可能需要面对 HR、技术面试官和老板。针对不同的角色，你应该准备不同的面试方式。具体分析如图 1-2 所示。[1]

面试官	他想要知道	可以向其询问
人力资源	你是谁？你的职业兴趣是什么？	公司的整体信息和组织结构；开放的职位
工程经理	你做过什么项目？具备什么技能？ 对什么项目感兴趣？	团队的职责是什么？团队目前和将来会做什么项目？团队在找什么样的人？
产品经理	对公司产品的观点/反馈/建议。如果让你设计一款产品，你会怎么做？	公司的下一步产品是什么？公司面临的关键性挑战是什么？公司中的工程师如何与产品经理交互？
午餐面试者	你之前或当前公司/学校的团队怎么样？你喜欢他们吗？为什么？你怎样融入公司的文化？	公司中的团队是怎么样的？他们有哪些团队构建活动？
工程师	如果你能够提出合理的清晰的问题，如果你能够有效地表达你解决问题的规划，如果你最终能够解决这个问题——那么，请展示你的代码	一般的工作流程是怎样的？公司使用的技术栈是什么？
架构师	你是否能够以可扩展的方式解决问题？你	公司使用的技术栈是什么？公司如何

面试官	他想要知道	可以向其询问
	是否能够认清系统设计中的关键权衡？	使用这些技术来解决现实世界的问题？

图 1-2 面试须知

[1] 这个表格主要是针对美国求职而言的。中国的程序员并不一定是这种形式，这里只是希望读者可以参考一下北美的面试流程。

1.2.1 HR

HR 是你与公司的连接点。通常，HR 负责安排协调面试，主要通过邮件联系。HR 也有可能会直接打电话联系你，目的是了解你的基本情况，包括身份、毕业/离职时间等。HR 通常会介绍职位要求和公司的基本情况，并且在面试当天接待你，了解你是否有其他面试安排或者其他公司的 Offer。此外，HR 还负责面试你的沟通能力，向老板反馈性格方面与团队的契合度以及对公司感兴趣的程度。因此，每次与 HR 的沟通也需要热情、职业。适合向 HR 了解的信息包括：公司的整体氛围、面试的流程安排、最近公司人员流动情况等。

1.2.2 技术面试官

技术面试官主要负责衡量你的技术水平，以及判断你是否符合职位要求。总体而言，对科技公司，技术面试官的意见最为重要。技术面试包括电话面试和现场面试，前者主要偏向概念性的问答，也包括通过协作网站直接写代码等。现场面试通常包括白板写代码，解决一个算法问题或者设计问题等。本书的主要目的就是帮助你通过这轮面试。

一些面试的小技巧如下：一定要先沟通，明确自己了解题意，不要过分考虑或者欠考虑。首先可以给出一个比较容易想到、但并不是最优的解决方案，再逐步优化。在思考的时候也要把思路讲出来，哪怕不是很成熟的方案。一旦遇到困难，可以先自己设法解决，如果五分钟没有思路，可以向面试官求助。适当的提示并不会影响你面试的最终结果。当开始写程序的时候，尽量注意语法格式、变量命名等，避免写伪代码，越接近真实代码越好。写完以后自己检查下有没有明显的错误，可以列举几个简单的测试数据，与面试官一起检验一下整

个运行过程。

面试是一个合作解决问题的过程，沟通一定是面试的关键：需要通过沟通展示你的逻辑性、理解能力和表达能力。在面试的最后，通常对方会给你提问的机会，你可以问的问题包括：团队平时使用什么样的技术，通常的工作压力和工作时间，公司最让人兴奋的地方；在当前职位工作了多少年，面试官之前的工作经历与现在相比有什么异同等。

面试考察的基本功，包括以下方面：

- 程序风格：能正确使用缩进，括号要对齐，变量名可以起的有意义；
- 编码习惯：异常检查，边界处理；
- 沟通：让面试官时刻明白你的意图，不要闭着眼睛不停地写。因为你的算法未必对。对了你也未必写得出来。中间稍微有点问题，你就失败了。对于面试官来说，他根本不知道你的解题进行到哪一步了；
- 测试：主动写出合理的测试用例（Test case），一些常见的用例，如 null 检查。一般你没写的话，面试官会让你写，但如果你主动写出来，说明你有好的习惯，容易加分。

技术面试的流程通常如下，可供参考。

当你拿到一个具体问题时，可以按照以下流程回答：

1．明确题意：通过与面试官交流明确需要解答的问题。这部分主要为了让自己放松心态，并且给面试官留下你具有良好团队意识和交流能力的印象。

2．描述大体思路：描述你打算用什么算法，什么数据结构。主要是为了让面试官了解你的思维过程，如果你给出的解答与他想要的答案偏差太多，可以及时纠正。同时，描述思路也给了你自己思考的机会。

3．实现算法：先处理边界条件。对于重要的算法模块，加一些注释或者与面试官进行交流。目的是让面试官始终了解你在做什么，算法框架是什么。

4．跑一个测试：用一个测试用例走一遍你写的程序。目的在于和面试官一起确保你的算法是有效的，可以在过程中及时发现并纠正自己的错误。同时，给面试官留下你有写单元测试（unit test）习惯的良好印象。

5. 描述算法复杂度，回答面试官的问题。

1.2.3 老板

团队的老板通常最后一个出场面试，或者陪同面试者一起吃午饭。老板负责收集整理所有人的反馈，并且决定是否发Offer。通常而言，老板可能不会问过于技术的问题，而是侧重考察你的协作沟通能力。老板的问题可能包括：如何面对工作中的难题/压力，你之前做过的项目，为什么适应这个职位等等。与老板沟通需要表现出你对他们团队的热情，并且在回答中尽量体现自己为什么适合这个职位。面试是一个相互的过程，通过与老板的面试，你需要了解这些问题：团队的成员构成，一般情况下项目如何分配，老板对你的期望，老板对团队在公司中发展的一些展望等等。

另外，准备一些常见的行为问题：比如你有没有过失败的经历，如果你老板给你不喜欢的任务怎么办，你想象中要成为什么样的人。这里一方面可以结合自身精力，另一方面多关注公司的介绍页面，包括公司创始人背景、企业文化、招聘的要求。这些都可以提前做好功课，尽量体现出来你的激情、负责、勤奋等优秀品质。

1.3 Offer

如果走到这一步，那么祝贺你，你成功了！在这一步，你需要一些谈判技巧，为自己争取更多的利益。首先，你要做的是与 HR 核对信息，包括你的地址、入职时间等。

通常，HR 会简单介绍你的待遇福利，当你确认无误后，HR 会生成正式文档让你签名。在这个阶段，你可以让 HR 解释 Offer 条约中你不理解的部分，并且协商你的待遇。协商的最常见方式是，当你有其他公司的 Offer，你希望最想去的公司能够 match 其他公司的最高值。注意，在这个阶段，HR 是与你站在一条战线上：HR 也不希望你轻易地拒绝他们的 Offer。因此，你完全有理由提出你自己的要求。一般在你有其他公司 Offer 的情况下，HR 都能争取到一些更多的利益。从争取难度而言，入职时的签字奖金最容易争取，股票和基准工资则比较难有提升。当然，工资待遇是重要的一方面，但在你决定是否接受 Offer 的时候，综合考虑公司的发展前景、团队在公司中的地

位、老板与你交流时你的感受、团队氛围等也是必不可少的因素。

对于美国的绝大部分公司，Offer 上都不会写雇佣时间，这意味着双方都可以随时终止合同。通常情况下，大公司不会轻易裁员，哪怕裁员也会有一定的补助。另一方面，这也说明你可以随时离职，甚至在入职之前，也即毁约。一般来说，不建议这种做法：更合理的做法是尝试与 HR 沟通，告知对方自己还有其他的面试/Offer，需要推迟一段时间做决定。如果实在万不得已，有其他更好的选择，你需要尽早与 HR 沟通，希望对方理解。一定不要拖到最后告诉对方自己不去了，这样的做法很不职业，也不礼貌。

当你接受 Offer 之后，可以向老板要一些材料，自己先准备一下，以便工作开始的时候能够更快上手。一般新到一个公司都会有数周甚至数月的上手时间，团队会专门有人帮助你了解他们的项目。新的旅程就此开始！但这仅仅是开始，未来也许是更大的挑战，能不能融入团队中，能不能抵抗住压力，工作内容是否符合自己的兴趣，这些都是未知数。所以我们说没有绝对正确的选择，只要用你的才华和汗水付出才有实际意义！

1.4 常见问题

问题 1：如何知道一些靠谱的公司？

首先，了解一下“牛人”都选择去哪些公司。如果公司名气不大，可以去流量排名上去看他处于什么地位，公司有没有上市，它的融资规模，还可以从 Linkedin 看它的员工是否优秀。

在硅谷，大家非常热情地谈创业谈机会，我们也通过自己的一些观察和积累，看到了不少最近几年才涌现出来的热门创业公司。给大家一个列表，这个是华尔街网站的全世界创业公司融资规模评选（http://graphics.wsj.com/billion-dollar-club/ ）。它本来的标题是“Billion Startup Club”（十亿美金创业公司俱乐部），不到一年的时间，截至 2015 年 1 月 17 日，现在的排名和规模已经发生了很大的变化，如图 1-3 所示。

第一，估值在 10Billlon（百亿美金）的公司达到了 7 家，而一年前一家都没有。第二，第一名是中国人家喻户晓的小米，第三，前 20 名中，绝大多数（八成）在美国，在加州，在硅谷，在旧金山！比如 Uber、Airbnb、Dropbox、Pinterest。第四，里面也有不少以相似模式取得成功的公司，如 Flipkart 就是印度市场的淘宝，Uber 与

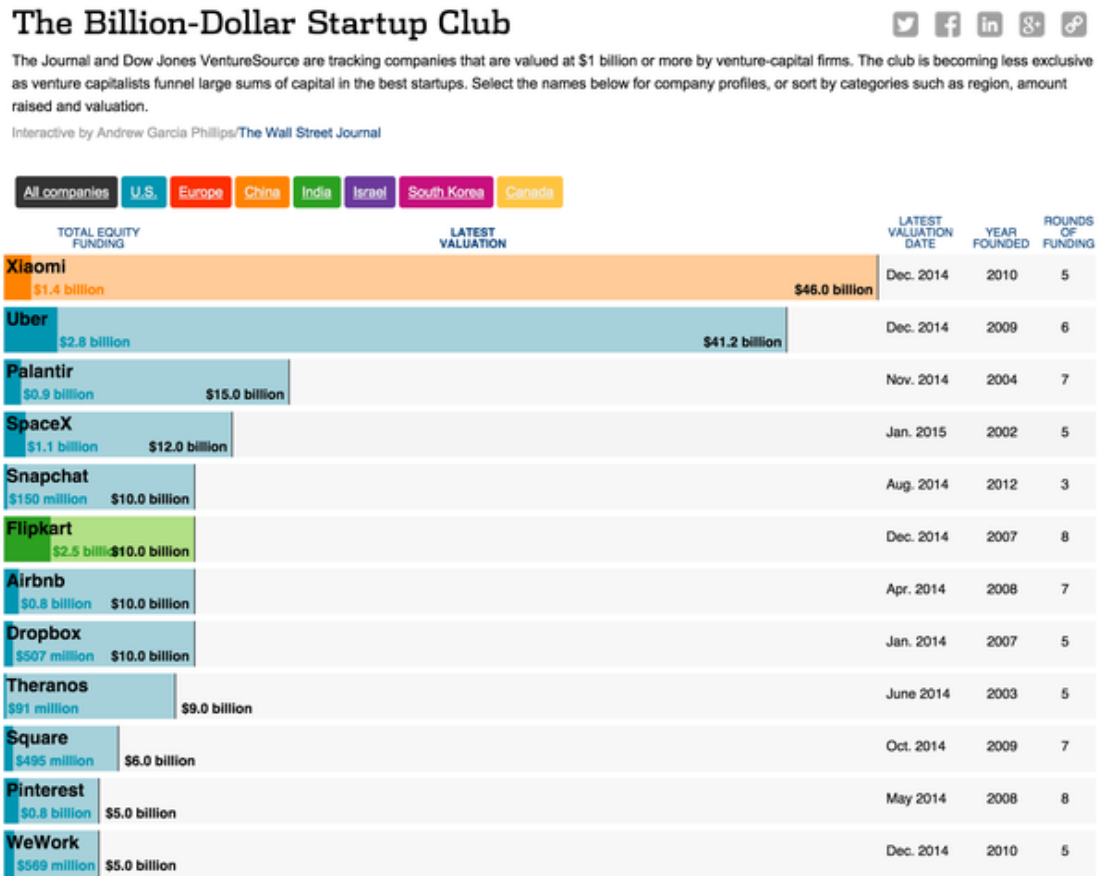
Airbnb 都是共享经济的范畴。所以大家还是可以在移动（Uber）、大数据（Palantir）、消费级互联网、通信（Snapchat）、支付（Square）、O2O App 里面寻找大的机会。

图 1-3 十亿美元创业公司俱乐部

问题 2：硅谷的 Startup 有什么技术方案？

分类介绍如下，如图 1-4 所示。

编程语言：Python、Scala、Swift for iOS、C/C++、Java 等主流开发语言；



前端：Play、Video.js、Jade、HTML5 等；

框架容器：Docker、Mesos、Vagrant 等；

日志监控：Datadog、Sumologic、Akka、Kafka 等；

后台数据处理：Hive、Scalding、EMR、Big Query 等；

图 1-4 硅谷常用技术矩阵

虚拟机：EC2 等云服务；

服务器：Nginx 等；

配置工具：NPM、Zookeeper、Puppet、Gradle 等；

信号通知：SQS 等；

代码托管：Github、SVN 等版本控制工具；

数据存储：Cassandra、MySQL、S3、Redshift 等；

内部工具：Phabricator、Slack、Google Docs、RelateIQ、Jira 等协作工具。

以 Coursera 为例，作为创业公司，Coursera 力图保持敏捷和高效。从技术上来说，所有的内容都是在基于 AWS 开发，可以想象随意启动云端服务，做一些实验。公司大致分成产品组、架构组和数据分析组。因为公司比较新，所以没有什么历史遗留迁移的问题。大家大胆地使用 Scala 作为主要编程语言，采用 Python 作为脚本控制，比如产品组就是提供课程产品，里面大量使用 Play Framework、Javascript 的 Backbone 作为控制中枢。而架构组主要是维护底层存储、通用服务、性能和稳定性。笔者所在的数据组，一部分是对商业产品、核心增长指标做监控、挖掘和改进；另一部分是搭建数据仓库，完善与各个部门的无缝数据流动，也用到很多技术；例如使用 Scalding 编写 MapReduce 程序，也有人做 AB testing 框架、推荐系统，尽可能用最少人力做有影响力的事情。其实除了开源世界，

Coursera 也积极使用第三方的产品，比如 Sumologic 做日志错误分析，Redshift 作为大数据分析平台，Slack 做内部通信。而所有的这些的目标就是想解放生产力，把重心放到用户体验、产品开发和迭代上去。

问题 3：什么时候才知道准备好面试？

可以从如下几个方面来衡量。

算法是否过关，是否能写出递归和动规；

Coding 是否过关，是否能在编译器中写出 Bug Free；

Design 是否过关，是否能给出 Tradeoff；

项目经历整理，能够流利说出架构、难点、自己的贡献；

加分项：Github、Blog、参与 Open Source。

问题 4：如何拿到美国工作签证？

要来美国工作，一般是要求具有 H1B（工作签证）的身份，获得该身份需要有雇主向劳工局提出申请（sponsorship）。根据现在的形式，每年的名额都是一抢而空，那么这就需要抽签决定。在 4 月 1 日之前提出申请，4 月 1 日之后开始抽签，如果是在美国获取硕士以上的学位，可以有优先级，抽取概率更高，而根据今年情况，普通的抽中几率是 50%。即便没有抽中，如果是拥有美国的硕士学位，可以使用 OPT 照样工作，并且可以省社会安全税（Social Security Tax）。而如果是海外的学位，只能等来年再抽签。

像 Google、Facebook 这种全球性公司，他们也会提供其他国家办公室的机会，工作一年后再通过 L1 或者 H1B 继续到美国工作。另外如果你抽中了 H1B，那么如果以后再跳槽，则可以通过办理转职（transfer）沿用之前的名额，而不需要再次抽签。H1B 每 3 年可以续一次，最多 6 年。如果 H1B 到期时已经申请绿卡，那么还是可以延长 H1B 的有效时间，直至绿卡生效。

问题 5：我不是算法大牛，不是 ACM 队员，听说 Google，Facebook 有很多牛人才能进，那我怎么能拿到好 Offer？

首先要有信心，算法不是想象那么难。原因有以下两个方面：

第一，因为面试常见的算法就那么几种。只要你努力去总结归类相似题目，只做很少的题，就可以举一反三掌握很多的题。不要盲目关注数字。做题质量非常重要。标准是：你做过的题目，让你再做一次，你就能“完美解决”。

第二，即使你知道一道题的解法，你未必能写好。因为你可能每

次写出来的程序都很随性，这样会漏洞百出。程序员是一个非常讲究严谨性的职业，如果你在总结题目的时候能够找到这些题目的模板，把模板提炼好，碰到类似的题目，可以一边写模板，一边想想怎么在模板上做一点简单的改动。这样既节省时间又保证不会出错。

问题 6：从其他专业转做计算机专业的该怎么准备？

1. 让自己更专业。例如，你的简历只能放和计算机有关的东西，其他东西再牛也不能发挥用处（如学生会主席）。如果简历还是很空，就多去做项目。实在没项目，就把非计算机专业的项目改写得更接近。

2. 在某一方面达到工程师实力。临时转行时间短，找一个容易入手的准备，如果你以前做的事情跟数据有关，就申请数据分析师（Data Analyst）。如果你本来就会一些基本技术，可以做前端，用 JavaScript、HTML 和 CSS，去真正做一个自己博客。如果对产品感兴趣，就玩 Django、Ruby on Rails 这样的网站框架，了解一个网站是怎么搭建起来的。如果对移动开发感兴趣，就写几个在 iOS 或者 Android 上的 App。这样做的好处是，简历不空，如果问到，可以驾轻就熟。如果不相关，坦诚相见，说由于我是转行的，在这方面不熟悉，可以尝试一下。也可以直接告诉面试官，我是转行的，我对计算机很感兴趣，做了这么这么几个项目。于是面试官不会问你难题。而实际上你早就准备好了，超出面试官预期，会得到一个面试高分！

问题 7：面试时候如何表现自己体现沟通能力？

首先你要站在面试官的角度思考问题。面试官要招你进去当同事，你希望同事怎么样？

你可以反复和面试官交流自己的想法，得到面试官认可以后再动手写。可以讲讲你是怎样想到这个思路的。从而展现你的沟通能力。记住，你并不是要说服他接受你的想法，而是要把你的想法解释给他听。面试官提出质疑的时候，第一，不要觉得面试官什么都不懂，怎么这都不知道（其实他只是看一下你是否真的懂）；第二，面试官比你经验丰富得多，很有可能就是你犯错了，赶紧想想是不是真的有问题。

问题 8：面试中出了 Bug 怎么办？

避免 Bug 很重要，这个需要我们平时不断地练习，按照上述的方法准备，还是可以避免一些“坑”的。但碰巧你可能不在状态，写出了 Bug 被面试官指出，是不是就挂了呢？

首先别担心，出 Bug 很正常，也许面试官来面试你之前正在 Debug。

衡量一个程序员能力的标准，并不是他能想出多牛的算法，而是程序员在遇到问题的时候分析和解决问题的能力。而出 Bug 的时候，正是展现你是否是一个合格程序员的时候！

Debug 的流程如下所示：

1．通过测试用例定位 Bug 所在位置；

2．不要立即修改代码，重新梳理逻辑。因为很有可能还有其他 Bug；

3．走完所有逻辑之后，心里有数怎么改了，再动手开始改；

4．用测试用例再走一次新的代码；

5．在整个过程中，不停地告诉面试官你在做什么（在不影响正常写程序的情况下）。

这样，成功排解 Bug，不但不会减分，还会因为你优秀的 Debug 能力和与此同时展现出来的沟通能力而加分。

问题 9：如何做出最后选择 Offer？

在考虑 Offer 之前，先对公司做个研究，比如这家公司是什么规模，产品是什么，Glassdoor 员工如何评价的，你的职位你喜欢吗？这就跟选学校一样，如果选错了，也是需要走很多弯路。我们个人的参考是首先这家公司是上升期的，产品是否有爱，团队是否比较强，能否学到东西。对公司分类，例如 Hortonworks 这种是纯技术性的，面向企业级的，可能没多少人知道，而 Uber 是大众消费性，很多朋友都用过。现在的热点是移动互联网，大家也可以多考虑这一块。

如果上市的公司，会给限制性股票，分 3～4 年行使期权，创业公司一般给期权，不同就是限制性股票是白送的，不需要自己掏腰包，期权需要自己买入，不同时期价格不同，但股票交的税非常高，有些期权是长期避税的。最后也要考虑你的兴趣和对风险的承受能力，如果去大公司做个螺丝钉，实现共产主义生活也无可厚非。去小公司压力大，成长快。但也要做好失败的准备，看看当年 Zynga 教训。

了解了面试的基本过程，让我们回到正题，按章节梳理技术面试的要点。

1.5 工具箱

本小节列出求职、面试和考虑 Offer 的整个过程中，可以参考的一些有用的网站和资源。

1. 求职

Glassdoor

http://www.glassdoor.com

LinkedIn

http://www.linkedin.com/jobs

Indeed

http://www.indeed.com/

CareerBuilder

http://www.careerbuilder.com/

Monster

http://www.monster.com/

2. 代码

A Short List of DevOps Tools

http://newrelic.com/devops/toolset

TopCoder

http://www.topcoder.com/

Google CodeJam

https://code.google.com/codejam

CodeChef

http://www.codechef.com/

HackerRank

https://www.hackerrank.com/

ACM ICPC

http://icpc.baylor.edu/

hackathon.io

http://www.hackathon.io

Hacker League

https://www.hackerleague.org

Hackathon Hero

http://www.hackathonhero.com/

3. 行业趋势

Y Combinator

https://news.ycombinator.com/jobs

AngelList

https://angel.co/jobs

Reddit

https://www.reddit.com/r/Jobbit

StackOverflow

http://careers.stackoverflow.com/

The Distributed Developer Stack Field Guide

http://sites.oreilly.com/odewahn/dds-field-guide/

4. 代码质量

Continuous Integration

http://martinfowler.com/tags/continuous%20integration.html

Java Code Review Checklist

http://java.dzone.com/articles/java-code-review-checklist

Stop More Bugs with our Code Review Checklist

http://blog.fogcreek.com/increase-defect-detection-with-our-code-review-checklist-example/

Embedded System Code Review Checklist

http://users.ece.cmu.edu/~koopman/pubs/code_review_checklist_v1_00.pdf

Typesafe Project & Developer Guidelines

https://gist.github.com/jboner/3864996

List of tools for static code analysis

http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis

Security Source Code Analysis Tools

https://www.owasp.org/index.php/Source_Code_Analysis_Tools

5. 团队合作开发

Slack

https://slack.com/

Google Apps (GMail, GChat)

https://www.google.com/work/apps/business/

IRC

http://en.wikipedia.org/wiki/Internet_Relay_Chat

Yammer

https://www.yammer.com/

Asana

https://asana.com/

Microsoft Project

http://products.office.com/en-us/project/project-and-portfolio-management-software

JIRA

https://www.atlassian.com/software/jira/

bugzilla

https://www.bugzilla.org/

GitHub Issues

https://github.com/blog/831-issues-2-0-the-next-generation

Confluence

https://www.atlassian.com/software/confluence

DropBox

https://www.dropbox.com

Google Drive

https://www.google.com/drive/

Zoho

https://www.zoho.com

RelateIQ

https://www.relateiq.com/

BambooHR

http://www.bamboohr.com/

6. 薪水

Crunchbase

Wealthfront Startup Salary & Equity Compensation

https://www.wealthfront.com/tools/startup-salary-equity-compensation

AngelList Salary and Equity Data

https://angel.co/salaries

Salary.com
http://www.salary.com/
PayScale Salary Calculator
http://www.payscale.com/salary-calculator
CareerBuilder Salary Calculator
http://salary.careerbuilder.com/
Wolfram Alpha Salary Comparison
http://www.wolframalpha.com/examples/SalariesWages.html
PayScale Cost of Living Calculator
http://www.payscale.com/cost-of-living-calculator
CNN Money Cost of Living Calculator
http://money.cnn.com/calculator/pf/cost-of-living/
An Introduction to Stock & Options for the Tech Entrepreneur or Startup Employee
http://www.scribd.com/doc/55945011/An-Introduction-to-Stock-Options-for-the-Tech-Entrepreneur-or-Startup-Employee
Startup Equity for Employees
http://www.payne.org/index.php/Startup_Equity_For_Employees
An Engineer's Guide to Stock Options
http://blog.alexmaccaw.com/an-engineers-guide-to-stock-options
The Impact of Dilution
https://blog.wealthfront.com/impact-of-dilution/
Term Sheet: Liquidation Preference
http://www.feld.com/archives/2005/01/term-sheet-liquidation-preference.html
I have a job Offer at a startup, am I getting a good deal?
http://venturehacks.com/articles/job-Offer
The 14 Crucial Questions About Stock Options
https://blog.wealthfront.com/stock-options-14-crucial-questions/
How much are startup options worth?

http://www.danshapiro.com/blog/2010/11/how-much-are-startup-options-worth/

The Equity Equation

http://paulgraham.com/equity.html

How I negotiated my startup compensation

https://keen.io/blog/29904565692/how-i-negotiated-my-startup-compensation

第 2 章 数组和字符串

2.1 知识要点

2.1.1 数组

数组（array）是常见的数据结构之一，用于存储一系列相同类型的数据。从底层实现的角度来看，所谓的“开辟一个数组”相当于系统为你提供了一段连续的内存区间用于存取数据。数组名就是一个指针，指向这段内存的起始地址。通过数组的类型，编译器知道在访问下一个元素的时候需要在内存中后移多少个字节。在 C/C++中，标准的数组可以通过在栈（Stack）上分配空间，或者通过先声明指针，然后用 new 关键字（或者 C 中的 malloc 函数），在堆（Heap）上动态地分配空间。举例如下：

```
int array[arraySize];    // 在栈上定义长度为 arraySize 的整型数组
int *array = new int[arraySize];    // 在堆上定义长度为 arraySize 的整型数组
```

使用完后需要释放内存：

delete[] array;注意，在旧的 C 编译器中，不能在栈上定义一个长度不确定的数组，即只能定义如下：

int array[10];新的 C 编译器没有这一限制。但是如果数组长度不定，则不能初始化数组：

int array[arraySize] = {0};   // 把不定长度的数组初始化为零，编译报错访问数组时要防止数组越界。所谓的数组越界，就是访问了不属于该数组的内存。之前提到，数组名就是一个指针，指向这段数组内存的起始地址。但这并不阻碍你通过下标（index）或

者移动指针的方式访问超过数组大小的内存区域。这样可能改写程序其他部分的数据，或者尝试写只读区域的内存，造成程序奔溃。在写循环或者传递数组的时候需要特别注意这点。

栈和堆的区别是一个常见的概念性问题，具体分析请见 2.3 节“工具箱”给出的参考资料。

我们常常需要二维数组存储一个矩阵。在 C/C++中，二维数组的使用方法简介如下：

在栈上创建：

int array[M][N];传递给子函数：

```
void func(int arr[M][N]){    // M 可以省略，但 N 必须存在，以便编译器确定移动内存的间距
  ...
}
```

在堆上创建：

```
int **array = new int*[M];    // 或者 (int**)malloc( M * sizeof(int*) );
for( int i = 0; i < M; i++)
     array [i] = new int[N];    // 或者 (int*)malloc( N * sizeof(int) );
```

传递给子函数：

```
void func( int **arr, int M, int N ){
   ...
}
```

使用完后需要释放内存：

```
for( int i = 0; i < M; i++)
    delete[] array[i];
delete[] array;
```

数组可以通过下标随机访问元素，所以在修改、读取某个元素的时候效率很高，具有 O(1)的时间复杂度。在插入、删除的时候需要移动后面的元素，所以平均时间复杂度 O(n)。通常，数组这个数据结构不是很适合增减元素。如果你想要的操作需要大量在随机位置增减元素，可以考虑其他的数据结构。在 C++中，标准库提供了 vector 容器，其本质上相当于一个长度可变的动态数组。2.3 节“工具箱”提供关于 vector 的常见使用方法。

2.1.2 哈希表

哈希表（Hash Table）几乎是最为重要的数据结构，主要用于基于“键(key)”的查找，存储的基本元素是键-值对(key-value pair)。

逻辑上，数组可以作为哈希表的一个特例：键是一个非负整数。注意，通常哈希表会假设键是数据的唯一标识，相同的键默认表示同一个基本存储元素。

哈希表的本质是当使用者提供一个键，根据哈希表自身定义的哈希函数（Hash Function），映射出一个下标，根据这个下标决定需要把当前的元素存储在什么位置。在一些合理的假设情况下，查找一个元素的平均时间复杂度是 O(1)，插入一个元素的平摊（amortized）时间复杂度是 O(1)。

当对于不同的键，哈希函数提供相同的存储地址时，哈希表就遇到了所谓的冲突（collision）。解决冲突的方式有链接法（chaining）和开放地址法（Open Addressing）两种。简单来说，链接法相当于利用辅助数据结构（比如链表），将哈希函数映射出相同地址的那些元素链接起来。而开放地址法是指以某种持续的哈希方式继续哈希，直到产生的下标对应尚未被使用的存储地址，然后把当前元素存储在这个地址里。

通常，链接法实现相对简便，但是可能需要附加空间，并且利用当前空间的效率不如开放地址法高。开放地址法更需要合理设计的连续哈希函数，但是可以获得更好的空间使用效率。需要注意的是，过于频繁的冲突会降低哈希表的搜索效率，此时需要哈希表的扩张。关于哈希表更深入的讨论请参见 2.3 节“工具箱”提供的参考资料。

C++标准库中提供 map 容器，可以插入、删除、查找键-值对，底层以平衡二叉搜索树的方式实现，根据键进行了排序。严格来说，map 并不是一个哈希表，原因是查找时间从 O(1)变为了 O(log n)，但是好处在于能够根据键值，顺序地输出元素，对于某些应用场景可能更为合适。在 C++11 中，标准库添加了 unordered_map，更符合哈希表的传统定义，平均查找时间 O(1)。2.3 节“工具箱”提供了 map 和 unordered_map 的常见使用方法。

2.1.3 String

在 C 语言中，字符串指的是一个以‘\0’结尾的 char 数组。关于字符串的函数通常需要传入一个字符型指针。然而，在 C++中，String 是一个类，并且可以通过调用类函数实现判断字符串长度等等操作。关于 C/C++中字符串的常见操作参见 2.3 节“工具箱”

2.2 模式识别

2.2.1 使用哈希表

当遇到某些题目需要统计元素集中一个元素出现的次数，应该直觉反应使用哈希表，即使用 std::unordered_map 或 std::map：键是元素，值是出现的次数。特别地，有一些题目仅仅需要判断元素出现与否（相当于判断值是 0 还是 1），可以用 bitvector，即 bitset，利用一个 bit 来表示当前的下标是否有值。

例题 1 Determine if all characters of a string are unique.

判断一个字符串所有的字符是否都是唯一的。

解题分析：这道题的关键在于“unique”。*一般来说，一旦出现“unique”，就落入使用哈希表或者 bitset 来判断元素出现与否的范畴。*进一步地，考虑如何建立键-值对：如果运用哈希表，我们可以直接用字符作为键，出现的次数作为值。特别地，由于“unique”只需要判断哈希表中是否已经存在当前键,所以我们可以通过 insert 函数的返回值做出相应的判断。

如果运用 bitset，我们需要建立字符到整数下标的映射关系。通常，字符都是 255 个 ASCII 编码之一，所以可以利用 ASCII 索引作为其整数下标的映射：a 对应 97，b 对应 98 等等。此时，通常可以与面试官进行沟通，作出字符串中仅含有 a-z、A-Z 的合理假设，这样可以缩小 bitset 的空间需求。

复杂度分析：哈希表和 bitset 做法都需要扫描整个字符串，每次插入操作时间复杂度 O(1)，假设字符串长度为 n，则平均时间复杂度都是 O(n)。空间上，每个合法字符都有可能出现，假设字符集大小为 m，则平均空间是 O(m)。哈希表的数据结构需要占用更多空间，所以 bitset 是更合理的数据结构。

参考解答：

```
bool isUnique(string input) {
    bitset<256> hashMap;
    for (int i = 0; i < input.length(); i++) {
        if (hashMap[(int)input[i]]) {
            return false;
        }
        hashMap[(int)input[i]] = 1;
    }
```

```
    return true;
}
```

例题 2 Given two strings, determine if they are permutations of each other.

给定两个字符串，判断它们是否是彼此可置换的。

解题分析：对于这道题目，我们需要找到两个字符串之间的共同点，即通过某种映射，使得所有置换得到相同的结果。考虑到置换的特性：无论如何变化，每个字符出现的次数一定相同。*一旦需要统计一个元素集中元素出现的次数，我们就应该想到哈希表。*于是，对于每个字符串，我们通过统计每个字符出现的次数把字符串映射成一个哈希表，最后比较两个哈希表是否相同。事实上，这种所谓的映射，其本身也是一个哈希的过程 （只不过哈希的结果是一个哈希表）：我们可以根据哈希的结果判断一个字符串集合中有多少字符串属于同一个置换。针对这道题目，我们还可以利用一些小技巧提前进行快速判断：如果两个字符串的长度不同，那它们一定不是一个置换。

复杂度分析：哈希表需要扫描整个字符串，每次插入操作时间复杂度 O(1)，假设字符串的长度为 n，则平均时间复杂度都是 O(n)。最后比较两个 hash 是否相同，每个合法字符都有可能出现，假设字符集大小为 m，则需要的时间复杂度是 O(m)，故总的时间复杂度 O(m+n)。空间上，平均空间是 O(m)。

其他解法：还有一个比较有意思的解法是分别对每个字符串中的字符按照 ASCII 编码顺序进行排序。如果是一个置换，那么排序完的两个字符串应该相等。该解法利用了置换的传递性：如果 A 可以通过置换变成 C，而 B 也可以通过置换变成 C，那么 A 一定能通过置换变成 B。这里我们只是通过对字符串中的字符排序，找出了一个比较容易实现的公共置换。这样做的时间复杂度是 O(n log n)，空间复杂度是 O(n)。

参考解答：

```
bool isPermutation(string stringA, string stringB) {
    if (stringA.length() != stringB.length()) {
        return false;
    }

    unordered_map<char, int> hashMapA;
    unordered_map<char, int> hashMapB;
    for (int i = 0; i < stringA.length(); i++) {
        hashMapA[stringA[i]]++;
```

```
        hashMapB[stringB[i]]++;
    }

    if (hashMapA.size() != hashMapB.size()) {
        return false;
    }

    unordered_map<char, int>::iterator it;
    for (it = hashMapA.begin(); it != hashMapA.end(); it++) {
        if (it->second != hashMapB[it->first]) {
            return false;
        }
    }
    return true;
}
```

例题3 Given a newspaper and message as two strings, check if the message can be composed using letters in the newspaper.

给定两个字符串 newspaper 和 message，检查是否能够使用 newspaper 中的字母来组成 message。

解题分析：在什么情况下 newspaper 中出现的字符能够组成 message？首先，message 中用到的字符必须出现在 newspaper 中。其次，message 中任意字符出现的次数一定少于其在 newspaper 中出现的次数。*一旦需要统计一个元素集中元素出现的次数，我们就应该想到哈希表。*键是字符，值是字符在 newspaper 中出现的次数。那么，如果 message 中用到的字符没有出现在哈希表中，则构成失败；如果 message 用到了某个哈希表中出现的字符，那我们可以将值减少，表示“消耗”了一个字符。如果最终某个值变成了负值，那么 message 中该字符出现的次数多于其在 newspaper 中出现的次数，构成失败。

复杂度分析：假设 message 长度为 m ，newspaper 长度为 n，我们的算法需要 hash 整个 message 和整个 newspaper，故时间复杂度为 O(m+n)。假设字符集大小为 c，则平均空间是 O(c)。

参考解答：

```
bool canCompose(string newspaper, string message) {
    unordered_map<char, int> hashMap;
    int i;
    if (newspaper.length() < message.length()) {
        return false;
    }
```

```
    for (i = 0; i < newspaper.length(); i++) {
        hashMap[newspaper[i]]++;
    }

    for (i = 0; i < message.length(); i++) {
        if (hashMap.count(message[i]) == 0) {
            return false;
        }
        if (--hashMap[message[i]] < 0) {
            return false;
        }
    }

    return true;
}
```

2.2.2 利用哈希表实现动态规划的思想

当处理当前节点需要依赖于之前的部分结果时，可以考虑使用哈希表记录之前的处理结果。其本质类似于动态规划（Dynamic Programming），利用哈希表以O(1)的时间复杂度利用之前的结果。

例题4 Find a pair of two elements in an array, whose sum is a given target number. Assume only one qualified pair of numbers existed in the array, return the index of these numbers (e.g. returns (i, j), smaller index in the front).

从一个数组中找出一对元素，其和是一个给定的目标数字。假设数组中只存在一个符合要求的数值对，返回这些数值的下标（例如，返回(i, j)，较小的下标在前面）。

解题分析：对于数组中的某个下标i，如何判断它是否属于符合条件的两个数字之一？最直观的方法是再次扫描数组，判断target - array[i]是否存在于数组中。这样做的时间复杂度是O(n^2)，效率不高的原因是没有保存之前的处理结果，每次都在做重复的工作。尽管效率不高，但是通过最直观的方法，*我们*发现处理当前节点需要依赖于之前的部分结果。如何保存之前的处理结果？可以使用哈希表。既然我们需要回答“target - array[i]是否存在于数组中”，那不妨把值作为键，通过询问哈希表是否含有所需要的键来得到我们需要的回答。最终，根据题目，我们需要返回下标，那么把下标作为哈希表

的值也是非常自然了。

复杂度分析：我们可以先对数组中的每个元素进行上述哈希处理，然后再从头至尾扫描数组，判断对应的另一个数是否存在于数组中，时间复杂度 O(n + n)。事实上，我们可以利用动态规划的思想，仅仅利用已经处理的部分解：哈希表只存储前驱节点的信息。对于当前节点，判断前驱中是否含有对应值。当处理完当前节点，把当前节点加入哈希表，作为已经处理的部分解。这样，时间复杂度可以进一步减少为 O(n)。

参考解答：

```
vector<int> addsToTarget(vector<int> &numbers, int target) {
    unordered_map<int, int> numToIndex;
    vector<int> vi(2);
    for (auto it = numbers.begin(); it != numbers.end(); it++) {
        if (numToIndex.count(target - *it)) {
            vi[0] = numToIndex[target - *it] + 1;
            vi[1] = (int)(it - numbers.begin()) + 1;
            return vi;
        }
        numToIndex[*it] = (int)(it - numbers.begin());
    }
}
```

例题 5 Get the length of the longest consecutive elements sequence in an array. For example, given [31, 6, 32, 1, 3, 2],the longest consecutive elements sequence is [1, 2, 3]. Return its length: 3.

获取一个数组中最长的连续的元素序列。例如，给定了[31, 6, 32, 1, 3, 2]，最长的连续的元素序列是[1, 2, 3]。返回其长度 3。

解题分析：如何判断当前节点 i 是否属于一个序列？如果 array[i] - 1 存在于数组中，那么 array[i]就可以作为后继加入序列。类似地，如果 array[i] + 1 存在于数组中，那么 array[i]就可以作为前驱加入序列。*我们发现处理当前节点需要依赖于之前的部分结果：判断 array[i] - 1，array[i] + 1 是否存在于数组中。如何保存之前的处理结果？可以使用哈希表。*很显然，键对应于数值。但对于这个问题，value 并不是那么明显，需要进一步分析。

一般而言，键用于快速判断哈希表中有没有我们需要的元素，值提供我们需要的结果。在这题中，我们期望获得怎样的部分解呢？我们需要知道现在已经构成的序列是怎样的。由于序列是一系列连续整

数，所以只要序列的最小值以及最大值，就能唯一确定序列。而所谓的“作为后继加入序列”，“作为前驱加入序列”，无非就是更新最大最小值。所以哈希表的值可以是一个记录最大/最小值的结构，用以描述当前节点参与构成的最长序列。

复杂度分析：根据上述算法，我们只要扫描一遍整个数组就能获得结果，时间复杂度 O(n)，附加空间复杂度 O(n)。

参考解答：

```
struct Bound {
    int high;
    int low;

    Bound(int h = 0, int l = 0) {
        high = h;
        low  = l;
    }
};
int lengthOfLongestConsecutiveSequence(vector<int> &num) {
    unordered_map<int, Bound> table;

    int local;
    int maxLen = 0;

    for (int i = 0; i < num.size(); i++) {
        if (table.count(num[i])) {
            continue;
        }

        local = num[i];

        int low = local, high = local;

        if (table.count(local - 1)) {
            low = table[local - 1].low;
        }

        if (table.count(local + 1)) {
            high = table[local + 1].high;
        }

        table[low].high = table[local].high = high;
        table[high].low = table[local].low = low;
```

```
        if (high - low + 1 > maxLen) {
            maxLen = high - low + 1;
        }
    }

    return maxLen;
}
```

2.2.3 String 相关问题的处理技巧

通常，纯粹的字符串操作难度不大，但是实现起来可能比较麻烦，边界情况（Edge Case）比较多。例如，把字符串变成数字，把数字变成字符串等等。这时候需要与面试官进行沟通，明确他们期望的细节要求，再开始写代码。同时，可以利用一些子函数，使得代码结构更加清晰。考虑到时间限制，有的时候面试官会让你略去一些过于细节的实现。此外，不妨看看经典字符串算法，比如判断子串等，这也是比较常见的面试题。

例题 6 Given input -> "I have 36 books, 40 pens2."; output -> "I evah 36 skoob, 40 2snep." (Suppose punctuation mark may only have period or comma)

给定输入是"I have 36 books, 40 pens2."；要求输出"I evah 36 skoob, 40 2snep."（假设标点符号只能是点号或逗号）。

解题分析：题意比较简单：把每个以空格或符号为间隔的单词逆向输出，如果遇到纯数字，则不改变顺序。自然而然地，每次处理分为两个步骤：（1）判断是否需要逆向 （2）逆向当前单词。这样就可以分为两个子函数：一个负责判断，另一个负责逆向。然后进行分段处理。另外，请注意如果括号中的假设没有给出，可以与面试官试探去要这个假设。

参考解答：

```
bool isPunctuationOrSpace(char *character) {
    return *character == ' ' || *character == ',' || *character == '.';
}

bool isNumber(char *character) {
    return *character >= '0' && *character <= '9';
}
```

```
bool needReverse(char *sentence, int *offset) {
    int length = (int)strlen(sentence);
    bool needReverse = false;
    *offset = 0;
    while (!isPunctuationOrSpace(sentence + (*offset)) && (*offset) < lengt
h) {
        if (!isNumber(sentence + (*offset))) {
            needReverse = true;
        }
        (*offset)++;
    }
    return needReverse;
}

void reverseWord(char *word, int length) {
    int i = 0, j = length - 1;
    while (i < j) {
        swap(*(word + i), *(word + j));
        i++;
        j--;
    }
}

void reverseSentence(char *sentence) {
    int length = (int)strlen(sentence);
    int offset;
    for (int i = 0; i < length;) {
        if (needReverse(sentence + i, &offset)) {
            reverseWord(sentence + i, offset);
        }
        i += (offset + 1);
    }
}
```

例题7 Replace space in the string with“%20”.E.g.given"Big mac". Return“Big%20mac”.

将字符串中的空格替换为“%20”。例如，给定“Big mac”，返回“Big%20mac”。

例题8 Replace“%20”substring with space.E.g.given“Big%20mac”, return“Big mac”.

将字符串中的“%20”替换为空格。例如，给定“Big%20mac”，返回“Big mac”。

解题分析：对于要求原处（in-place）的删除或修改，可以用两

个 int 变量分别记录新下标与原下标，不断地将原下标所指的数据写到新下标中。

这里有个小规律：如果改动后字符串长度增大，则先计算新字符串的长度，再从后往前对新字符串进行赋值；反之，则先从前往后顺序赋值，再计算长度。

2.3 工具箱

1. 栈和堆（操作系统意义上的，而不是数据结构意义的）

栈主要是指由操作系统自动管理的内存空间。当进入一个函数，操作系统会为该函数中的局部变量分配存储空间。事实上，系统会分配一个内存块，叠加在当前的栈上，并且利用指针指向前一个内存块的地址。函数的局部变量就存储在当前的内存块上。当该函数返回时，系统“弹出”内存块，并且根据指针回到前一个内存块。所以，栈总是以后进先出（LIFO）的方式工作。通常，在栈上分配的空间不需要用户操心。

堆是用来存储动态分配变量的空间。对于堆而言，并没有像栈那样后进先出的规则，程序员可以选择随时分配或回收内存。这就意味着需要程序员自己用命令回收内存，否则会产生内存泄漏（memory leak）。在 C/C++中，程序员需要调用 free/delete 来释放动态分配的内存。在 Java、Objective-C（with Automatic Reference Count）中，语言本身引入垃圾回收和计数规则帮助用户决定在什么时候自动释放内存。

2. Vector

vector 可以用运算符[]直接访问元素。简要介绍如下常见函数（更多信息请参考 http://www.cplusplus.com/reference/vector/vector/）：

```
size_type size() const;    // Returns the number of elements in the vector.
void push_back (const value_type& val);    // Adds a new element at the end of the vector. The content of val is copied (or moved) to the new element.
```

注意：如果放入一个类的实例（instance），会调用复制构造函数（copy constructor）

```
void pop_back();    // Removes the last element in the vector, effectively reducing the container size by one.
```

注意：如果弹出的是一个类的实例，pop_back 会调用类的析构函

数（deconstructor）。

```
iterator insert (iterator position, const value_type& val);    // inserting new elements before the element at the specified position, effectively increasing the container size by the number of elements inserted.
```

示例：

```
vector::iterator it = v.begin();
it = v.insert ( it , 200 );
insert 函数有一些变式，可以用来插入多个数据，请见 http://www.cplusplus.com/reference/vector/vector/insert/
iterator erase (iterator position);
iterator erase (iterator first, iterator last);    // Removes from the vector either a single element (position) or a range of elements ([first,last]).
```

注意：erase 总是返回下一个有效的 iterator。此外，在利用 for 循环删除的时候，需要额外小心 iterator 后移的条件。通常的 for 循环可能导致越界：例如 for(it = v.begin(); it != v.end(); it++)，当删除最后一个元素时，it 实际上已经成为 v.end()，如果继续执行 it++，则会产生越界。可以利用下述代码进行删除：

```
        for (vector<int>::iterator it = v.begin(); it != v.end(); ) {
          if (condition) {
              it = v.erase(it);
          } else {
              ++it;
          }
      }
```

3. 哈希表

关于哈希表的理论分析，请见参考资料：

《算法导论》（Introduction to Algorithms，2nd Edition），Thomas H. Corman, Charles E.Leiserson，Ronald L.Rivest, Clifford Stein，第 11 章，哈希表。

4. Map / unordered_map

map 可以用运算符[]直接访问元素。简要介绍如下常见函数（更多信息请参考 http://www.cplusplus.com/reference/map/map/）：

```
   pair<iterator,bool> insert (const value_type& val);    // Extends the container by inserting new element, effectively increasing the container size. Returns a pair, using Boolean to indicate if inserted successfully or not
```

示例：

```
std::pair<std::map<char,int>::iterator,bool> ret;
ret = mymap.insert ( std::pair<char,int>('z',500) );
if (ret.second==false) {
    std::cout << "element 'z' already existed";
    std::cout << " with a value of " << ret.first->second << '\n';
}
void erase (iterator position);
size_type erase (const key_type& k);
void erase (iterator first, iterator last);    // Removes from the map
container either a single element or a range of elements ([first,last)).
```

示例：

```
it=mymap.find('b');
mymap.erase (it);    // erasing by iterator
mymap.erase ('c');    // erasing by key
it=mymap.find ('e');
mymap.erase ( it, mymap.end() );    // erasing by range
iterator find (const key_type& k);    // Searches the container for an eleme
nt with a key equivalent to k and returns an iterator to it if found, otherw
ise it returns an iterator to map::end.
size_type count (const key_type& k) const;    // Searches the container for
elements with a key equivalent to k and returns the number of matches.
```

5. C语言中的字符串相关的常见函数

相关函数通常定义在string.h中，简要介绍如下（更多信息参考[http://www. cplusplus.com/reference/cstring/](http://www.cplusplus.com/reference/cstring/)）：

```
char *strcpy ( char *destination, const char *source );    //copy
source string to destination string
```

示例：

```
char str1[] = "Sample string";
char str2[SAMPLE_STRING_SIZE];
strcpy (str2,str1);
char *strcat ( char *destination, const char *source );    //Appends a copy
of the source string to the destination string.
```

示例：

```
char str[STRING_SIZE] = "string is ";
strcat(str, "concatinated");    //str becomes "string is concatinated"
```

int strcmp (const char *str1, const char* str2);

返 回 值

说 明

0

str1，str2 每个字符都相同

< 0

str1，str2 不相等且第一个不同字符 str1 小于 str2

> 0

str1，str2 不相等且第一个不同字符 str1 大于 str2

```
char *strchr ( const char *str, int character );    // Returns a pointer to the first occurrence of character in the C string str (NULL if not found).
```

示例：

```
char string[STRING_LENGTH] = "This is a string";
int firstOccurPos = (int)(strchr(string, 'i') - string);    // firstOccurPos equals to 2 (here we assume string contains character 'i')
char *strrchr ( char *str, int character );    // Returns a pointer to the last occurrence of character in the C string str.
char *strstr (char *str1, const char *str2 );    // Returns a pointer to the first occurrence of str2 in str1, or a NULL pointer if str2 is not part of str1.
size_t strlen ( const char *str );    // Returns the length of the C string str.
```

示例：

```
char string[STRING_LENGTH] = "This is a string";
int length = strlen(string);    // length equals to 16
double atof (const char *str);    // convert char string to a double
int atoi (const char *str);    // convert char string to an int
```

6. C++ String 类的常见函数

由于 String 类重载了+、<、>、=、==等运算符，故复制、比较是否相等、附加子串等都可以用运算符直接实现。简要介绍其他常见函数如下（更多信息请参考 http://www.cplusplus.com/reference/string/string/）：

```
size_t find (const string& str, size_t pos = 0) const;    // Searches the string for the first occurrence of the str, returns index
string substr (size_t pos = 0, size_t len = npos) const; // Returns a newly constructed string object with its value initialized to a copy of a substring starting at pos with length len.
```

示例：

```
string str("This is a string");
string subStr = str.substr(10,6);    // subStr equals to "string", with length 6
string &erase (size_t pos = 0, size_t len = npos);  // erase characters from pos with length len
```

示例：

```
string str("This is a string");
str.erase(0,10);    // str becomes "string", with length 6 after erasure
size_t length();    // Returns the length of the string, in terms of bytes.
```

示例：

```
string str ("This is a string");
int length = str.length();    // length equals to 16
```

7. String Matching 的常见算法

简单介绍两种比较易于实现的字符串比较算法，下述算法假设在长度为 n 的母串中匹配长度为 m 的子串。更详细的解释请见（http://en.wikipedia.org/wiki/String_searching_algorithm）。

Brute-Force 算法： 顺序遍历母串，将每个字符作为匹配的起始字符，判断是否匹配子串。时间复杂度为 O(mn)。

Rabin-Karp 算法：将每一个匹配子串映射为一个哈希值。例如，将子串看做一个多进制数，比较它的值与母串中相同长度子串的哈希值，如果相同，再细致地按字符确认字符串是否确实相同。顺序计算母串哈希值的过程中，使用增量计算的方法：扣除最高位的哈希值，增加最低位的哈希值。因此能在平均情况下做到 O(m+n)。

示例：

为描述简单，假设字符串只含有十进制数字，母串为"1234"，待匹配子串为"23"，定义 hash 函数把字符串转成整数值。

首先计算子串 hash：值为 23。

然后计算母串前两个字符的 hash：值为 12，与子串不符合，需要后移。此时我们扣除最高位"1"的 hash，增加新的最低位"3"的 hash，得到新的 hash 值 23，与子串相符。

通过按字符比较发现的确匹配成功，可以返回母串匹配上的下标。

伪代码：

```
int RabinKarp(string s[1..n], string pattern[1..m])
hpattern = hash(pattern[1..m]);  hs = hash(s[1..m]);
for i from 1 to n-m+1
if hs = hpattern
    if s[i..i+m-1] = pattern[1..m]
return i
hs = hash(s[i+1..i+m])
      return not found
```

第 3 章 链表

3.1 知识要点

链表（Linked List）是一种常见的线性数据结构。对于单向链表（Singly Linked list），每个节点有一个 next 指针指向后一个节点，还有一个成员变量用以存储数值；对于双向链表（Doubly Linked List），还有一个 prev 指针指向前一个节点。与数组类似，搜索链表需要 O(n)的时间复杂度，但是链表不能通过常数时间（O(1)）读取第 k 个数据。链表的优势在于能够以较高的效率在任意位置插入或删除一个节点。

除特别说明外，本文提到的链表特指单向链表。在 C++中，标准库函数中 List 实现了一个双向链表。

3.2 模式识别

3.2.1 链表的基本操作

凡是修改单向链表的操作，只需考虑：

（1）哪个节点的 next 指针会受到影响，则需要修正该指针。

（2）如果待删除节点是动态开辟的内存空间，则需要释放这部分空间（C/C++）。

毕竟，一个链表节点，无非是包含 value 和 next 这两个成员变量的数据结构而已。对于双向链表，类似的，则只需额外考虑谁的 prev 指针会受到影响。

举例如下：

```
void delNode(ListNode *prev) {
  ListNode *curr = prev->next;
  prev->next = curr->next;    // 删除 curr 节点只会使 prev 节点的 next 受到影响
  delete curr;    // 清理 trash 指针
}
```

注意，操作链表时务必注意边界条件：curr == head, curr == tail 或者 curr == NULL。

3.2.2 哑节点

链表操作时利用哑节点（Dummy Node）是一个非常好用的技巧：只要涉及操作 head 节点，不妨创建 dummy node: ListNode *dummy = new ListNode(0); dummy->next = head; 这使得操作 head 节点与操作其他节点无异。

例题 1 Given a linked list and a value x, write a function to reorder this list such that all nodes less than x come before the nodes greater than or equal to x.

给定一个链表和一个值 x，编写一个函数，对该链表重新排序，以便所有小于 x 的节点都出现在大于或等于 x 的节点的前面。

解题分析：将链表分成两部分，但这两部分的头节点连是不是 NULL 都不确定。*当涉及对头节点的操作，我们不妨考虑创建哑节点。*这样做的好处是：我们总是可以创建两个哑节点，用 dummy->next 作为真正头节点的引用，然后在此基础上 append，这样就不用处理头节点是否为空的边界条件了。

参考解答：

```
ListNode *reorderList(ListNode *head, int x) {
    ListNode *newHead = NULL;
    ListNode *aDummy = new ListNode(0);
    ListNode *aCurr = aDummy;
    ListNode *bDummy = new ListNode(0);
    ListNode *bCurr = bDummy;

    while(head) {
        ListNode *next = head->next;
        head->next = NULL;
        if( head->val < x ) {
            aCurr->next = head;
            aCurr = head;
        } else {
            bCurr->next = head;
            bCurr = head;
        }
        head = next;
    }
    aCurr->next = bDummy->next;
    newHead  = aDummy->next;
```

```
    delete aDummy;
    delete bDummy;

    return newHead;
}
```

3.2.3 Runner 和 Chaser

对于寻找链表的某个特定位置的问题，不妨用两个指针变量 runner 与 chaser (ListNode*runner = head,* chaser = head)，以不同的速度遍历该链表，以找到目标位置。在验证算法正确性时，可以用几个简单的小测试用例（例如长度为 4 和 5 的链表）。注意，通常需要分别选取链表长度为奇数和偶数的测试用例，以验证算法在一般情况下的正确性。

例题 2 Given a linked list, write a function to return the middle point of that list. 给定一个链表，编写一个函数以返回该链表的中间点。

解题分析：*由于题目涉及在链表中寻找特定位置，我们用两个指针变量以不同的速度遍历该链表。*其中，runner 以两倍速前进，chaser 一倍速。这样，当 runner 到达尾节点时，chaser 即为所求解。在实际实现的时候，还需要注意链表的越界问题。

参考解答：

```
ListNode *midpoint( ListNode *head) {
    ListNode *chaser = head, *runner = head;
    if (head == NULL)
        return NULL;
    while (runner->next && runner->next->next ) {
        chaser = chaser->next;
        runner = runner->next->next;
    }
    return chaser;
}
```

例题 3 Find the kth to last element of a singly linked list. For example, if given a list 1->2->3->4, and k equals to 2, the function should return element 2.

找到一个单向链表中，距离最后一个元素为 k 的那个元素。例如，如果给定一个链表 1->2->3->4，并且 k 等于 2，那么，该函数应该返

回元素 2。

解题分析：与例题 2 类似，*由于题目涉及在链表中寻找特定位置，我们用两个指针变量遍历该链表。*只是在本例中，runner 与 chaser 以相同倍速前进，但 runner 提前 k 步出发。

参考解答：

```
ListNode *findkthtoLast(ListNode *head,int k){
    ListNode *runner = head;
    ListNode *chaser = head;

    if (head == NULL || k < 0)
     return NULL;

    for(int i = 0; i < k; i++)
        runner = runner->next;

    if( runner == NULL )
        return NULL;
    while( runner->next != NULL ) {
        chaser = chaser->next;
        runner = runner->next;
    }
    return chaser;
}
```

例题 4 Given a linked list that may contain a circle, write a function to return the node at the beginning of that circle. Return NULL if such list doesn't contain a circle.

给定一个可能包含一个环的链表，编写一个函数来返回环开始的节点。如果该链表不包含环，返回 NULL。

解题分析：*寻找某个特定位置，用* runner 和 chaser。这里的技巧是，如果 runner 以两倍速度前进，chaser 以一倍速前进，在有环的情况下，runner 与 chaser 一定能在某点相遇。相遇后，再让 chaser 从头节点出发再次追赶 runner，此时两者都以一倍速前进，可以证明，第二次相遇的节点即为环的开始位置。

参考解答：

```
ListNode* findLoop(ListNode* head) {
    if (head == NULL)
        return NULL;

    ListNode *slow = head;
    ListNode *fast = head;
```

```
    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast)
            break;
    }

    if (fast == NULL || fast->next == NULL)
        return NULL;

    slow = head;
    while (slow != fast) {
        slow = slow->next;
        fast = fast->next;
    }

    return slow;
}
```

例题5 Given a list, rotate the list to the right by k places, where k is non-negative. e.g: 1->2->3->4->5, k = 2; after rotation: 4->5->1->2->3

给定一个链表，向右旋转链表k个位置，其中k是一个非负数。例如，1->2->3->4->5, k = 2; 旋转之后得到：4->5->1->2->3。

解题分析：假设链表长度为len，新的链表的尾节点是原链表中第len-k个节点。则我们的runner指针需要前进len-k-1次，到达的位置即为新链表中的尾节点。同时，下一个节点即为新链表的头结点。

参考解答：

```
ListNode *rotateRight(ListNode *head, int k) {
    int len = 1;
    ListNode *cur = head;

    if(head == NULL)
        return head;

    while(cur->next){
        len++;
        cur = cur->next;
    }    // find last element in the list and calculate list length

    k = k%len;
```

```
    if(k == 0)
        return head;
    cur->next = head;  // tail of the original list should link to the origin
al head

    // use runner pointer to find the new tail
    cur = head;
    for(int i = 0; i < len - k - 1; i++)
        cur = cur->next;

    ListNode *const newTail = cur;
    ListNode *const newHead = cur->next;

    newTail ->next = NULL;

    return newHead;
};
```

3.2.4 遍历并处理节点

在遍历链表时，注意每次循环内只处理一个或一对节点，否则很容易出现重复处理的问题。

例题 6 Reverse the linked list and return the new head.

将链表逆转并且返回新的链表头。

解题分析：循环遍历链表，每次*只处理当前指针的* next 变量，由此实现链表的逆转。

参考解答：

```
ListNode *reverseLinkedList( ListNode *head ) {
    ListNode *prev = NULL;
    ListNode *next = NULL;
    while(head) {
        next = head->next;
        head->next = prev;

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

3.2.5 交换节点的问题

如果需要交换两个节点的位置，则对这两个节点的前驱节点，它们的 next 指针会受到影响；这两个节点本身的 next 指针，也会受到影响。但是，如下过程总是可以实现交换：

（1）先交换两个前驱节点的 next 指针的值。

（2）再交换这两个节点的 next 指针的值。

无论这两个节点的相对位置和绝对位置如何，以上的处理方式总是成立。

例题 7 Given a linked list, swap every two adjacent nodes and return its head.

给定一个链表，交换每两个相邻的节点并返回其链表头。

解题分析：依照上述交换规则给出解答如下。

参考解答：

```
ListNode *swapPairs(ListNode *head) {
    if(head == NULL)
        return head;

    ListNode *dummy = new ListNode(0);
    dummy->next = head;
    ListNode *prev = dummy;
    ListNode *node1 = head, *node2 = head->next;
    ListNode *newHead = NULL;

    while(node1 && node1->next) {
        node2 = node1->next;

        // swap the "next" of prev nodes;
        ListNode *tmp1 = prev->next;    // = node1
        prev->next = node1->next;    // = node2
        node1->next = tmp1;    // = node1

        // swap the "next" of current nodes;
        ListNode *tmp2 = node1->next;    // = node1
        node1->next = node2->next;
        node2->next = tmp2;    // = node1

        prev = node1;
        node1 = prev->next;
    }
```

```
    newHead = dummy->next;
    delete dummy;
    return newHead;
}
```

根据 comment 优化之后，循环代码段可以缩写为：

```
    while(node1 && node1->next) {
      node2 = node1->next;

      // swap the "next" of prev nodes;
      prev->next = node1->next;    // = node2

      // swap the "next" of current nodes;
      node1->next = node2->next;
      node2->next = node1

      prev = node1;
      node1 = prev->next;
   }
```

但应当注意到，就算不优化，代码仍然是有效的，充其量只是多用了几个临时变量而已。

3.2.6 同时操作两个链表

遇到同时处理两个链表的问题，循环的条件一般可以用 while（list1 && list2），当循环跳出后，再处理剩下非空的链表。这相当于：边界情况特殊处理，常规情况常规处理。

例题 8 Given two sorted linked lists, write a function to merge these two lists, and return a new list which is sorted.

给定两个有序的链表，编写一个函数来合并这两个链表，并且返回一个新的有序链表。

解题分析：这是一个典型的需要同时处理两个链表的问题，*可以先处理常规情况*（两个链表都有剩下节点），*再处理边界情况*（其中一个链表已经遍历完毕）。在处理常规情况的时候，我们将当前两个链表中较小的那个节点放入新的链表。

参考解答：

```
ListNode *mergeTwoLists(ListNode *l1, ListNode *l2) {
    ListNode *dummy = new ListNode(0);
    ListNode *cur = dummy;
    ListNode *newHead = NULL;

    while(l1&&l2) {
        if(l1->val <= l2->val){
            cur->next = l1;
            l1 = l1->next;
        } else {
            cur->next = l2;
            l2 = l2->next;
        }
        cur = cur->next;
    }

    if(l1)
        cur->next = l1;
    else
        cur->next = l2;

    newHead = dummy->next;
    delete dummy;
    return newHead;
}
```

3.2.7 倒序处理

如果对靠前节点的处理必须在靠后节点之后，即类似于倒序访问的问题，可以用递归（recursion），或者等效地，用栈来解决。

例题 9 Traverse the linked list reversely.

倒序地访问链表。

解题分析：倒序访问问题，我们利用递归处理。

参考解答：

```
void reversedTraverse(ListNode *head) {
    if (head == NULL)
        return;
    traverse(head->next);
    visit(head);
```

```
}
```

例题 10 Given two linked lists, each element of the lists is a integer. Write a function to return a new list, which is the “sum” of the given two lists.

Part a. Given input (7->1->6) + (5->9->2), output 2->1->9.
Part b. Given input (6->1->7) + (2->9->5), output 9->1->2.
给定两个链表，链表的每个元素都是一个整数。编写一个函数来返回一个新的链表，它是给定的两个链表的“和”。
Part a. 给定输入(7->1->6) + (5->9->2)，输出 2->1->9，即 617+295=912
Part b. 给定输入(6->1->7) + (2->9->5)，输出 9->1->2，即 617+295=912

解题分析：对于 a，靠前节点的解不依赖靠后节点，因此顺序遍历求解即可。对于 b，*靠前节点的解依赖于靠后节点（进位），因此必须用递归或栈处理。*并且，子问题返回的结果，可以是一个自定义的结构（进位 + sub-list）。当然，对于问题 b，也可以通过逆向列表之后再用 a 的解法求解。同时，注意到该题还是处理两个链表的问题，所以*可以先处理常规情况*（两个链表都有剩下节点），*再处理边界情况*（其中一个链表已经遍历完毕）。

参考解答：

a.

```
ListNode * addList(ListNode *l1, ListNode *l2) {
    int carry = 0;
    ListNode *dummy = new ListNode(0);
    ListNode *curr = dummy;
    ListNode *newHead = NULL;

    while(l1 && l2) {
        int sum = l1->value + l2->value + add_on;
        add_on = sum/10;
        curr->next = new ListNode(sum % 10);
        curr = curr->next;
        l1 = l1->next;
        l2 = l2->next;
    }

    ListNode *rest = l1 ? l1 : l2;

    while(rest) {
```

```
        int sum = rest->value + add_on;
        carry = sum/10;
        curr->next = new ListNode(sum % 10);
        curr = curr->next;
        rest = rest->next;
    }

    if (add_on)
        curr->next = new ListNode(carry);

    newHead = dummy->next;
    delete dummy;
    return newHead;
}
```

b.

```
struct ListWithCarry {
    ListNode *head;
    int Carry;
};

int getLen(ListNode *head) {
    int len = 0;
    while(head) {
        len++;
        head = head->next;
    }
    return len;
}

ListNode *padList(ListNode *head, int diff){
    ListNode *dummy = new ListNode(0);
    ListNode *curr = dummy;
    ListNode *newHead = NULL;
    for(int i = 0; i < diff; i++) {
        curr->next = new ListNode(0);
        curr = curr->next;
    }
    curr->next = head;
    newHead = dummy->next;
    delete dummy;
    return newHead;
}

ListWithCarry addListImpl(ListNode *l1, ListNode *l2) {
```

```
    ListWithCarry res;
    res.head = NULL;
    res.Carry = 0;
    if (!l1 && !l2)
        return res;
    int sum = 0;

    ListWithCarry subList = addListImpl(l1->next, l2->next);

    if(l1)
        sum += l1->value;
    if(l2)
        sum += l2->value;
    sum += subList.Carry;

    res.Carry = sum/10;
    res.head->val = sum%10;
    res.head->next = subList.head;

    return res;
}

ListNode *addList(ListNode *l1, ListNode *l2) {
    int len1 = getLen(l1);
    int len2 = getLen(l2);
    if (len1 > len2) {
        l2 = padList(l2, len1 - len2);
    } else if (len1 < len2){
        l1 = padList(l1, len2 - len1);
    }

    ListWithCarry resStruct = addListImpl(l1, l2);
    ListNode *dummy = new ListNode(0);
    ListNode *newHead = NULL;
    if(resStruct.Carry) {
        dummy->next = new ListNode(resStruct.Carry);
        dummy->next->next = resStruct.head;
    } else {
        dummy->next = resStruct.head;
    }
    newHead = dummy->next;
    delete dummy;
    return newHead;
}
```

3.3 工具箱

对于利用链表解决问题，而非解构链表的问题，可以考虑使用标准库。

对 C++,双链表(doubly linked list)的实现类是 std::list<T>.

常用 iterator：begin()、end()、rbegin()、rend()

常用函数：

```
empty(), size(), push_back(T value), pop_back(T value);
erase(iterator pos), insert(iterator pos, T value);
```

对于 Java，双链表的实现类是 LinkedList<E>

常用函数：

```
add(E e), add(int index, E element), remove(int index),
addAll(Collection<? Extends E> c),  get(int index),
```

第 4 章 栈和队列

4.1 知识要点

4.1.1 栈

栈（stack）是一种数据结构，可以实现后进先出（Last In First Out, LIFO）。通常情况下，我们可以用栈作为辅助，实现深度优先算法（Depth First Search, DFS），或者将递归转为 while 循环。在本书第 5 章中，可以看到这样的实例。

事实上，递归本身相当于把函数本身一层一层地加到操作系统的内存栈上，所以利用栈数据结构去实现递归也是非常自然的：入栈操作相当于递归调用自身，出栈操作相当于递归返回。入栈操作的对象相当于需要被解决的问题，出栈对象相当于已经解决的子问题，通过共享的状态变量或返回值把子问题的结果传递出来。

最基本的栈至少包括入栈（push）和出栈（pop），前者将一个元素放入栈内，后者将最后放入栈的元素弹出。在 C++中，标准模版库提供封装好的栈类，具体使用方法见 4.3“工具箱”。

4.1.2 队列

与栈对称，队列（Queue）帮助实现先进先出（First in first out,FIFO),我们可以用 Queue 作为辅助,实现广度优先算法(Breadth first search，BFS)。在第 5 章可以看到这样的实例。队列还可以作为 buffer，构建一个生产者－消费者模型：生产者把新的元素加到队尾，消费者从队头读取元素。在有两个线程同时读取同一个队列时，需要考虑同步（synchronization)，具体问题在第 12 章中讨论。

事实上，栈与队列可以视作封装好的链表，只是限制了访问和插入的自由。因此适用栈或队列的情境，也可以考虑使用更为强大的链表。

4.2 模式识别

4.2.1 通过栈实现特殊顺序的读取

由于栈具有 LIFO 的特性，如需实现任何特定顺序的读取操作，往往可以借助两个栈互相“倾倒”来实现特定顺序。事实上，在很多情况下，栈并不是实现这种读取顺序的最佳数据结构。但作为面试问题，往往面试官会很明确地说明用栈实现。此时，我们就应该立刻想到利用另一个栈作为辅助。

例题 1 Design a stack with push(), pop() and max() in O(1) time.

设计一个栈，带有 push()、pop()和 max()方法，其时间复杂度为 O(1)。

解题分析：显然，在 push 的时候记录当前的最大值，只需要在插入当前值的时候比较是否比现在的最大值大，如果是，更新最大值即可。问题在于如何恰当处理 pop 的情况：直观想法是，当需要出栈的元素是当前最大值，则需要“回溯”到前一个最大值。*考虑到栈具有* LIFO 的特性，那么与之匹配地，最大值追踪方式也需要具有相同特性：不妨用另一个栈追踪最大值。注意一个细节，当最大值重复入栈时，我们的“最大值栈”也需要重复记录该值。否则，在弹出第一个重复最大值的时候，我们就可能在最大值栈中丢失相应的信息。

复杂度分析：时间复杂度符合题目要求为O(1)。空间复杂度最坏情况附加的栈中需要存储每个元素，故额外使用O(n)空间。

参考解答：

```
class stackWithMax {
private:
stack<int> valueStack;
stack<int> maxStack;

public:
    void push(int);
    int pop();
    int max();
};

void stackWithMax::push(int value) {
    if (maxStack.empty() || maxStack.top() <= value) {
        maxStack.push(value);
    }
    valueStack.push(value);
}

int stackWithMax::pop() {
    int value = valueStack.top();
    valueStack.pop();
    if (value == maxStack.top()) {
        maxStack.pop();
    }
    return value;
}

int stackWithMax::max() {
    return maxStack.top();
}
```

例题2 Given a stack structure, use it to implement a queue.

给定一个栈结构，使用它来实现一个队列。

解题分析：这是一道比较常见的题目。栈的输出顺序是LIFO，队列的输出顺序是 FIFO，考虑到*如果想利用栈实现特定顺序的读取操作，往往可以借助两个栈互相“倾倒”来实现特定顺序*：当一个栈中的元素倾倒到另一个栈中，则原栈最后出栈的元素最先出栈，相当于实现了FIFO。

复杂度分析：出栈由于多了倾倒的过程，故时间复杂度降为O(n)。空间复杂度不变，因为并没有重复存储元素。

参考解答：

```
class Queue {
private:
    stack<int> inputStack;
    stack<int> outputStack;
public:
    void enqueue(int);
    int dequeue();
};

void Queue::enqueue(int value) {
    inputStack.push(value);
}

int Queue::dequeue() {
    int value;
    if (!outputStack.empty()) {
        value = outputStack.top();
        outputStack.pop();
        return value;
    }
    while (!inputStack.empty()) {
        outputStack.push(inputStack.top());
        inputStack.pop();
    }
    value = outputStack.top();
    outputStack.pop();
    return value;
}
```

例题 3 How to sort a stack in ascending order (i.e. pop in ascending order) with another stack?

如何按照升序，使用另外一个栈来排序一个栈？例如，按照升序来 pop。

解题分析：本题明确要求利用两个栈实现一种特定的输出顺序。由于栈限制了输出的顺序，所以要调整栈内部的元素顺序比较困难。但从另一个角度想，*我们也许可以在入栈的时候就控制插入的位置，使得栈内的元素都是有序的*，即只要保证新栈的元素一定是有序的即可。同时，考虑到栈具有 LIFO 的输出性质，*将一个栈中的元素倾倒到另一个栈，再倾倒回来，元素之间保持原有顺序。*

于是我们可以构建如下算法：假设有 Stack A 和 Stack B，Stack A 中的元素没有顺序。我们需要把 Stack A 中的数据有序地加入 Stack B 中。每当我们从 Stack A 中取出一个元素，当该元素不符合 Stack B 的当前排列顺序时，我们倾倒 Stack B 的元素，直到该元素可以按顺序入栈。然后，我们恢复之前倾倒的元素。根据栈“将一个栈中的元素倾倒到另一个栈，再倾倒回来，元素之间保持原有顺序”的性质，特别地，我们可以将 Stack A 看成性质中的“另一个栈”。

复杂度分析：由于调整一个元素的顺序可能要求将之前的 n 个元素来回倾倒，故时间复杂度 O(n^2)。

参考解答：

```
stack<int> sort(stack<int> &input) {
    stack<int> output;
    while (!input.empty()) {
        int value = input.top();
        input.pop();
        while (!output.empty() && output.top() < value) {
            input.push(output.top());
            output.pop();
        }
        output.push(value);
    }
    return output;
}
```

4.2.2 “Save for later”问题

有一类问题有这样的特性：当前节点的解依赖后驱节点。也就是说，对于某个当前节点，如果不能获知后驱节点，就无法得到有意义的解。这类问题可以通过栈（或等同于栈的若干个临时变量）解决：先将当前节点入栈，然后看其后继节点的值，直到其依赖的所有节点都完备时，再从栈中弹出该节点求解。某些时候，甚至需要反复这个过程：将当前节点的计算结果再次入栈，直到其依赖的后继节点完备。

例题 4 Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is a valid parentheses string. For example, “(([]))” is valid, but “()” or “((” is not.

给定一个字符串，它只包含如下的字符：'('、')'、'{'、'}'、

'['和']',判断输入的字符串是否是一个有效的圆括号字符串。例如，“(([]))”是有效的，而“()”和“((”则不是。

解题分析：比较明显的是，为了判断输入是否有效，我们需要从头到尾地扫描整个字符串。问题在于，对于某个字符，我们并不能立刻判断是否有效，因为*当前节点的解依赖后驱节点*。这种情况下，我们可以尝试使用栈作为辅助结构：需要先将当前节点入栈，然后看其后继节点的值，直到其依赖的所有节点都完备时，再从栈中弹出该元素求解。具体地，当当前节点是一个左括号，我们就把它入栈，直到读到其所依赖的右括号，再从栈中弹出该节点。如果当前节点是右括号，则栈顶必须是与之相依赖的左括号，否则输入无效。当扫描完整个字符串，栈应该为空，否则输入无效。

复杂度分析：时间复杂度为O(n)。辅助栈最多有n个数据入栈，故额外空间复杂度为O(n)。

参考解答：

```
bool isLeftParentheses (char input) {
    return input == '(' || input == '[' || input == '{';
}

bool isMatchParentheses(char left, char right) {
    switch (left) {
        case '(':
            return right == ')';
        case '[':
            return right == ']';
        case '{':
        return right == '}';
    }
    return false;
}

bool isValidParentheses(string input) {
    stack<char> parenthesesStack;
    for (int i = 0; i < input.length(); i++) {
        if (isLeftParentheses(input[i])) {
            parenthesesStack.push(input[i]);
        } else {
            if (parenthesesStack.empty() || !isMatchParentheses(parenthesesS
tack.top(), input[i])) {
                return false;
            }
```

```
            parenthesesStack.pop();
        }
    }
    return parenthesesStack.empty();
}
```

4.2.3 用栈解决自上而下结构的问题

所谓的自上而下（Top-Down）结构，从逻辑理解的角度来看，实际上就是一种树形结构，从顶层出发，逐渐向下扩散，例如二叉树的遍历问题。 在实际运算的时候，我们先解决子问题，再利用子问题的结果解决当前问题。从算法角度而言，就是利用递归。用递归解决自上而下结构的问题，详见第 8 章。

由于栈的 LIFO 特性，可以利用栈数据结构消除递归。递归通常用函数调用自身实现，在调用的时候系统会分配额外的空间，并且需要用栈指针记录函数返回的位置，故额外开销（overhead）比较大。但在实际工作或面试中，用栈或者用递归的区别标准不大，按照自己擅长的方式做就可以。在使用栈的时候，每个子问题应当被看成是同样类型的对象（object），将该对象按照自上而下的方向入栈。然后通过 while 循环，调用栈的 pop()函数弹出栈顶元素并访问，直至栈清空。这样，后入栈的子问题会优先被弹出，相当于实现了递归。

例题 5 Given a binary tree, implement the In-Order Traversal using a stack.

给定一个二叉树，使用栈实现中序遍历。

解题分析：中序遍历（In-Order Traversal）的规则是：（1）先遍历左子树（2）访问当前节点（3）遍历右子树。我们先逻辑上重现整个遍历过程，对于某个上层节点，先向左下降到下一层，解决子问题，回到当前节点，向右下降到下一层。很显然，*遍历子树的过程是一个自上而下结构：从顶层出发，逐渐向下扩散。*实际计算时，我们先解决子问题（遍历左子树），再利用子问题的结果解决当前问题（访问当前节点，转向右子树）。如果用栈作为辅助结构消除递归，由于子问题是遍历左子树，所以我们将左孩子以自上而下的顺序放入栈，每次弹出时，都意味着子问题已经解决好了，需要解决当前问题，即访问当前节点，再转向右子树。

复杂度分析：该算法不重复地扫描所有节点，故时间复杂度为 O(n)。

参考解答：

```
void inOrderTraversal(TreeNode *root) {
    if(root == NULL)
        return;
    stack<TreeNode *> st;
    while(!st.empty() || root ) {
        if(root) {
            st.push(root);
            root = root->left;
        } else {
            root = st.top();
            st.pop();
            visit( root );
            root = root->right;
        }
    }
}
```

例题 6 Solve the Hanoi Tower (http://mathworld.wolfram.com/ TowerofHanoi.html) problem without recursion

不使用递归求解汉诺塔问题。

解题分析：在面试中，比较普遍的一个技巧是*从最基本的情况出发，根据题意推倒整个计算流程。*这样做的好处是：（1）确保自己正确地理解了题目（2）从简单的情况出发，找找解题思路。*该方法特别适用于递归，动态规划等题目类型。*在这里就可以尝试一下：

（1）假设有 1 个盘子

直接把盘子从 left 移动到 right，需要 1 步完成。

（2）假设有 2 个盘子

将盘子 1 从 left 移动到 middle。

将盘子 2 从 left 移动到 right。

将盘子 1 从 middle 移动到 right。

可以发现，这里我们解了一个子问题：把 n-1 个盘子移动到 middle，然后把第 n 个盘子移动到 left，最后把 n-1 个盘子从 middle 移动到 left。这里限于篇幅不列举三个盘子的情况，但强烈建议你根据上面子问题的描述方式重现三个盘子的情况。

根据上述分析，*汉诺塔（Hanoi Tower）的游戏过程是一个自上而下结构：从顶层（移动 n 个盘子）出发，逐渐向下（移动 n-1 个盘子）扩散。*实际计算时，我们先解决子问题（移动 n-1 个盘子到 middle），

再利用子问题的结果解决当前问题（移动最后一个盘子到 right，移动 n-1 个盘子到 right）。

如果用栈作为辅助结构消除递归，由于子问题是“从 A 柱通过 B 柱移动 n 个盘子到 C 柱”，所以我们可以构造含有 source、midpoint、destination 及盘子个数信息的 object，并把它加入栈中。每次弹出时，需要解决当前问题，即把“从 A 柱通过 C 柱移动 n-1 个盘子到 B 柱”，然后“从 A 移动 1 个盘子到 C”，最后“从 B 柱通过 A 柱移动 n-1 个盘子到 C 柱”。注意栈的 LIFO 特性，我们需要的出栈顺序如上所述，实际入栈顺序与之相反。

复杂度分析：可以用数学归纳法证明复杂度为 O(2^n-1)：

（1）1 个盘子的时候需要 2^1 - 1 步。

（2）根据（1），我们知道移动 n-1 个盘子需要 2^(n-1)-1 步。而移动 n 个盘子需要做：把 n-1 个盘子移动到 middle，然后把第 n 个盘子移动到 left，最后把 n-1 个盘子从 middle 移动到 left。即 2 * (2^(n-1)-1)+1= 2^n-1。

（3）由数学归纳法可知，对于任意 n，完成 Hanoi Tower 需要 2^n-1 步。

参考解答：

```
enum Tower { Zero, Left, Mid, Right };

class HanoiObj {
public:
    Tower src, tmp, dest;
    int num, index;
    HanoiObj(Tower s, Tower t, Tower d, int n) : src(s), tmp(t), dest(d), nu
m(n) {
        if (n == 1) {
            index = 1;
        }
    }
    HanoiObj(Tower s, Tower d, int i) : src(s), dest(d), index(i) {
        num = 1; tmp = Zero;
    }
};

void move(Tower src, Tower dest, int index) {
    cout << "Move Disk #" << index << " from Tower " << src << " to Tower " <
< dest << endl;
}
```

```
void TOH(int N) {
    stack<HanoiObj *> Hstack;
    Tower s, t, d;
    int n;
    Hstack.push(new HanoiObj(Left, Mid, Right, N));
    while (!Hstack.empty()) {
        HanoiObj *tmpObj = Hstack.top();
        Hstack.pop();
        s = tmpObj->src;
        t = tmpObj->tmp;
        d = tmpObj->dest;
        n = tmpObj->num;
        if (n < 1) {
            continue;
        }
        if (n == 1) {
            move(s, d, tmpObj->index);
        } else {
// because Stack is LIFO, the push sequence is symmetric to the way we expec
t it to pop
            Hstack.push(new HanoiObj(t, s, d, n - 1));
            Hstack.push(new HanoiObj(s, d, n)); // basically, equivalent to c
alling move function
            Hstack.push(new HanoiObj(s, d, t, n - 1));
        }
        delete tmpObj;
    }
}
```

4.3 工具箱

1. 栈

栈在C++标准模版库（Standard Template Library, STL）中实现，使用时需要include<stack>。简要介绍如下常见函数（更多信息请参考http://www.cplusplus.com/reference/stack/stack/）：

```
bool empty() const;    // Returns whether the stack is empty: i.e. whether its size is zero.
void push (const value_type& val);    // Inserts a new element at the top of the stack. The content of this new element is initialized to a copy of val.
```

示例：

```
stack<int> myStack;
myStack.push(10);    // stack has one element, the value of which is 10
```

注意：如果放入一个类的实例（instance），会调用复制构造函数（copy constructor）。

```
void pop();    // Removes the element on top of the stack, effectively reducing its size by one.
```
示例：

```
stack myStack;
myStack.push(10);
myStack.push(20);    // stack now has two elements, the value of which is 20, 10
myStack.pop();    // stack now has one element, the value of which is 10
value_type& top();    // Returns a reference to the top element in the stack
```

示例：

```
stack<int> myStack;
myStack.push(10);
myStack.push(20);
int value = myStack.top();    // value equals to 20
```

2. 队列

队列在C++标准模版库（Standard Template Library，STL）中实现，使用时需要include<queue>。简要介绍如下常见函数（更多信息请参考http://www.cplusplus.com/reference/queue/queue/）：

```
bool empty() const;    // Returns whether the queue is empty
void push (const value_type& val);    // Inserts a new element at the end of the queue. The content of this new element is initialized to val.
```

示例：

```
queue<int> myQueue;
myQueue.push(10);    // queue has one element, the value of which is 10
```

注意：如果放入一个类的实例（instance），会调用复制构造函数（copy constructor）。

```
void pop();    // Removes the oldest element in the queue,
effectively reducing its size by one.
```
示例：

```
queue myQueue;
myQueue.push(10);
myQueue.push(20);
int value = myQueue.front();    // value equals to 10
value_type& front();    // Returns a reference to the oldest element in the
queue.
```

示例：

```
queue<int> myQueue;
myQueue.push(10);
myQueue.push(20);       // queue now has two elements, the value of which is
 10, 20
myQueue.pop();     // queue now has one element, the value of which is 20
```

第 5 章 树和图

5.1 知识要点

5.1.1 树

树的概念

树（tree）是一种能够分层存储数据的重要数据结构，树中的每个元素称为树的节点，每个节点有若干个指针指向子节点。从节点的角度来看，树是由唯一的起始节点引出的节点集合。这个起始结点称为根（root）。树中节点的子树数目称为节点的度（degree）。在面试中，关于树的面试问题非常常见，尤其是关于二叉树（Binary Tree）、二叉搜索树（Binary Search Tree，BST）的问题。

所谓的二叉树，是指对于树中的每个节点而言，至多有左右两个子节点，即任意节点的度小于等于 2。而广义的树则没有如上限制。二叉树是最常见的树形结构。二分查找树是二叉树的一种特例，对于二分查找树的任意节点，该节点存储的数值一定比左子树的所有节点的值大，比右子树的所有节点的值小（与之完全对称的情况也是有效的：即该节点存储的数值一定比左子树的所有节点的值小，比右子树的所有节点的值大）。

基于这个特性，二分查找树通常被用于维护有序数据。二分查找树查找、删除、插入的效率都会高于一般的线性数据结构。事实上，对于二分查找树的操作相当于执行二分搜索，其执行效率与树的高度（depth）有关，检索任意数据的比较次数不会多于树的高度。这里需要引入高度的概念：对一棵树而言，从根节点到某个节点的路径长度称为该节点的层数（level），根节点为第 0 层，非根节点的层数是其父节点的层数加 1。树的高度定义为该树中层数最大的叶节点的层数加 1，即相当于从根节点到叶节点的最长路径加 1。由此，对于 n 个数据，二分查找树应该以尽可能小的高度存储所有数据。由于二叉树第 L 层至多可以存储 2^L 个节点，故树的高度应在 log n 量级，因此，二分查找树的搜索效率为 O(log n)。

直观上看，尽可能地把二分查找树的每一层“塞满”数据可以使得搜索效率最高，但考虑到每次插入删除都需要维护二分查找树的性质，要实现这点并不容易。特别地，当二分查找树退化为一个由小到大排列的单链表时（每个节点只有右孩子），其搜索效率变为 O(n)。为了解决这样的问题，人们引入平衡二叉树的概念。所谓平衡二叉树，是指一棵树的左右两个子树的高度差的绝对值不超过 1，并且左右两个子树都是一棵平衡二叉树。通过恰当的构造与调整，平衡二叉树能够保证每次插入删除之后都保持平衡性。平衡二叉树的具体实现算法包括 AVL 算法和红黑算法等。由于平衡二叉树的实现比较复杂，故一般面试官只会问些概念性的问题。关于平衡二叉树的具体算法，请参考 5.3 节“工具箱”给出的参考资料。

树型的概念

满二叉树（Full Binary Tree）：如果一棵二叉树的任何结点，或者是叶节点，或者左右子树都存在，则这棵二叉树称作满二叉树。

完全二叉树（Complete Binary Tree）：如果一棵二叉树最多只有最下面的两层节点度数可以小于 2，并且最下面一层的节点都集中在该层最左边的连续位置上，则此二叉树称作完全二叉树。

二叉树的遍历

二叉树的常见操作包括树的遍历，即以一种特定的规律访问树中的所有节点。常见的遍历方式包括：

前序遍历（Pre-Order Traversal）：访问根结点；按前序遍历左子树；按前序遍历右子树。

中序遍历（In-Order Traversal）：按中序遍历左子树；访问根

结点；按中序遍历右子树。特别地，对于二分查找树而言，中序遍历可以获得一个由小到大或者由大到小的有序序列。

后续遍历（Post-Order Traversal）：按后序遍历左子树；按后序遍历右子树；访问根结点。

以上三种遍历方式都是深度优先搜索算法（Depth-First Search）。深度优先算法最自然的实现方式是通过递归实现，事实上，大部分和树相关的面试问题都可以优先考虑递归。此外，另一个值得注意的要点是：深度优先的算法往往都可以使用栈数据结构将递归化为非递归实现。这里利用了栈先进后出的特性，其数据的进出顺序与递归顺序一致（请见第 4 章）。上述遍历的具体实现请见 5.3“工具箱”。

层次遍历（Level Traversal）：首先访问第 0 层，也就是根结点所在的层；当第 i 层的所有结点访问完之后，再从左至右依次访问第 i+1 层的各个结点。层次遍历属于广度优先搜索算法（Breadth-First Search）。广度优先算法往往通过队列数据结构实现。层次遍历的具体实现请见 5.3“工具箱”。

5.1.2 字典树

字典树（Trie 或 Prefix tree）是一个 26 叉树，用于在一个集合中检索一个字符串，或者字符串前缀。字典树的每个节点有一个指针数组代表其所有子树，其本质上是一个哈希表，因为子树所在的位置（index）本身，就代表了节点对应的字母。节点与每个兄弟具有相同的前缀，这就是 trie 也称为 prefix tree 的原因。

假设我们要存储如下名字和年龄：

Amy 12

Ann 18

Bob 30

则构成的字典树如下：

```
.                 root: level 0
a---------b       level 1
|           |
m---n    o        level 2
|     |    |
y     n    b       level 3
|     |    |
12  18   30       level 4
```

由于 Amy 和 Ann 共享前缀 a，故第二个字母 m 和 n 构成兄弟关系。

字典树以及字典树节点的原型：

```
class TrieNode {
    private:
        T mContent;
        vector<TrieNode*> mChildren;
    public:
        Node();
        ~Node();
        friend class Trie;
};
class Trie {
public:
    Trie();
    ~Trie();
    void addWord(string s);
    bool searchWord(string s);
    void deleteWord(string s);
private:
    TrieNode* root;
};
```

字典树的基本函数如下：

（1）void addWord(string key, int value);

添加一个键-值对。添加时从根节点出发，如果在第 i 层找到了字符串的第 i 个字母，则沿该节点方向下降一层（注意，如果下一层存储的是数据，则视为没有找到）。否则，将第 i 个字母作为新的兄弟插入到第 i 层。将键插入完成后，再插入值节点。

（2）bool searchWord(string key, int &value);

查找某个键是否存在，并返回值。从根节点出发，在第 i 层寻找字符串中第 i 个字母是否存在。如果是，沿着该节点方向下降一层；否则，返回 false。

（3）void deleteWord(string key)

删除一个键-值对。删除时从底层向上删除节点，直到遇到第一个有兄弟的节点（说明该节点向上都是与其他节点共享的前缀），删除该节点。

5.1.3 堆与优先队列

通常所说的堆（Heap）是指二叉堆，从结构上说是完全二叉树，从实现上说一般用数组。以数组的下标建立父子节点关系：对于下标为 i 的节点，其父节点为(int)i/2，其左子节点为 2i，右子节点为 2i+1。堆最重要的性质是，它满足部分有序（Partial Order）：最大（小）堆的父节点一定大于等于（小于等于）当前节点，且堆顶元素一定是当前所有元素的最大（小）值。

堆算法的核心在于插入、删除算法如何保持堆的性质（以下讨论均以最大堆为例）：

下移（shift-down）操作：下移是堆算法的核心。对于最大值堆而言，对于某个节点的下移操作相当于比较当前节点与其左右子节点的相对大小。如果当前节点小于其子节点，则将当前节点与其左右子节点中较大的子节点对换，直至操作无法进行（即当前节点大于其左右子节点）。

建堆：假设堆数组长度为 n，建堆过程如下。

```
for i ← n/2 downto 1
    do shift-down(A,i)
```

插入：将新元素插入堆的末尾，并且与父节点进行比较，如果新节点的值大于父节点，则与之交换，即上移（shift-up），直至操作无法进行。

弹出堆顶元素：弹出堆顶元素（假设记为 A[1]，堆尾元素记为 A[n]）并维护堆性质的过程如下。

```
output = A[1]
exchange A[1] <-> A[n]
heap size -= 1
shift-down(A,1)
return output
```

值得注意的是，堆的插入操作逐层上移，耗时 O(log(n))，与二叉搜索树的插入相同。但建堆通过下移所有非叶子节点（下标 n/2 至 1）实现，耗时 O(n)，小于 BST 的 O(nlog(n))。关于堆的具体理论讨

论请见 5.3“工具箱”。

通过上述描述，不难发现堆其实就是一个优先队列。对于 C++，标准模版库中的 priority_queue 是堆的一种具体实现。具体使用方法请参考 5.3“工具箱”。

5.1.4 图

图（Graph）是节点集合的一个拓扑结构，节点之间通过边相连。图分为有向图和无向图。有向图的边具有指向性，即 AB 仅表示由 A 到 B 的路径，但并不意味着 B 可以连到 A。与之对应的，无向图的每条边都表示一条双向路径。

图的数据表示方式也分为两种，即邻接表（Adjacency List）和邻接矩阵（Adjacency Matrix）。对于节点 A，A 的邻接表将与 A 之间相连的所有节点以链表的形势存储起来，节点 A 为链表的头节点。这样，对于有 V 个节点的图而言，邻接表表示法包含 V 个链表。因此，链接表需要的空间复杂度为 O(V+E)。邻接表适用于边数不多的稀疏图。但是，如果要确定图中边（u, v）是否存在，则只能在节点 u 对应的邻接表中以 O(E)复杂度线性搜索。

对于有 V 个节点的图而言，邻接矩阵用 V*V 的二维矩阵形式表示一个图。矩阵中的元素 Aij表示节点 i 到节点 j 之间是否直接有边相连。若有，则 Aij数值为该边的权值，否则 Aij数值为 0。特别地，对于无向图，由于边的双向性，其邻接矩阵的转置矩阵为其本身。邻接矩阵的空间复杂度为 O(V^2)，适用于边较为密集的图。邻接矩阵在检索两个节点之间是否有边相连这样的需求上，具有优势。

5.1.5 图的遍历

图的遍历（Graph Transversal）类似于树的遍历（事实上，树可以看成是图的一个特例），也分为广度优先搜索和深度优先搜索。算法描述如下：

广度优先

对于某个节点，广度优先会先访问其所有邻近节点，再访问其他节点。即，对于任意节点，算法首先发现距离为 d 的节点，当所有距离为 d 的节点都被访问后，算法才会访问距离为 d+1 的节点。广度优先算法将每个节点着色为白、灰或黑，白色表示未被发现，灰色表示

被发现，黑色表示已访问。算法利用先进先出队列来管理所有灰色节点。一句话总结，广度优先算法先访问当前节点，一旦发现未被访问的邻近节点，推入队列，以待访问。

《算法导论》第 22 章关于图的基本算法部分给出了广度优先的伪代码实现，引用如下：

```
BFS(G, s)
For each vertex u exept s
    Do Color[u] = WHITE
        Distance[u] = MAX
        Parent[u] = NIL
Color[s] = GRAY
Distance[s] = 0
Parent[s] = NIL
Enqueue(Q, s)
While Q not empty
    Do u = Dequeue(Q)
        For each v is the neighbor of u
            Do if Color[v] == WHITE
                Color[v] = GRAY
                Distance[v] = Distance[u] + 1
                Parent[v] = u
                Enqueue(Q, v)
            Color[u] = BLACK
```

深度优先

深度优先算法尽可能“深”地搜索一个图。对于某个节点 v，如果它有未搜索的边，则沿着这条边继续搜索下去，直到该路径无法发现新的节点，回溯回节点 v，继续搜索它的下一条边。深度优先算法也通过着色标记节点，白色表示未被发现，灰色表示被发现，黑色表示已访问。算法通过递归实现先进后出。一句话总结，深度优先算法一旦发现没被访问过的邻近节点，则立刻递归访问它，直到所有邻近节点都被访问过了，最后访问自己。

《算法导论》第 22 章关于图的基本算法部分给出了深度优先的伪代码实现，引用如下：

```
DFS(G)
For each vertex v in G
    Do Color[v] = WHITE
    Parent[v] = NIL
For each vertex v in G
    DFS_Visit(v)
```

```
DFS_Visit(u)
Color[u] = GRAY
For each v is the neighbor of u
    If Color[v] == WHITE
        Parent[v] = u
        DFS_Visit(v)
Color[u] = BLACK
```

5.1.6 单源最短路径问题

对于每条边都有一个权值的图来说，单源最短路径问题是指从某个节点出发，到其他节点的最短距离。该问题的常见算法有 Bellman-Ford 和 Dijkstra 算法。前者适用于一般情况（包括存在负权值的情况，但不存在从源点可达的负权值回路），后者仅适用于均为非负权值边的情况。Dijkstra 的运行时间可以小于 Bellman-Ford。本小节重点介绍 Dijkstra 算法。

特别地，如果每条边权值相同（无权图），由于从源开始访问图遇到节点的最小深度就等于到该节点的最短路径，因此 Priority Queue 就退化成 Queue，Dijkstra 算法就退化成 BFS。

Dijkstra 的核心在于，构造一个节点集合 S，对于 S 中的每一个节点，源点到该节点的最短距离已经确定。进一步地，对于不在 S 中的节点，我们总是选择其中到源点最近的节点，将它加入 S，并且更新其邻近节点到源点的距离。算法实现时需要依赖优先队列。一句话总结，Dijkstra 算法利用贪心的思想，在剩下的节点中选取离源点最近的那个加入集合，并且更新其邻近节点到源点的距离，直至所有节点都被加入集合。关于 Dijkstra 算法的正确性分析，可以使用数学归纳法证明，详见《算法导论》第 24 章单源最短路径。这里给出伪代码如下：

```
DIJKSTRA(G, s)
S = EMPTY
Insert all vertexes into Q
While Q is not empty
    u = Q.top
    S.insert(u)
    For each v is the neighbor of u
        If d[v] > d[u] + weight(u, v)
            d[v] = d[u] + weight(u, v)
            parent[v] = u
```

5.1.7 任意两点之间的最短距离

另一个关于图常见的算法是，如何获得任意两点之间的最短距离（All-pairs shortest paths）。直观的想法是，可以对于每个节点运行 Dijkstra 算法，该方法可行，但更适合的算法是 Floyd-Warshall 算法。

Floyd 算法的核心是动态规划，利用二维矩阵存储 i，j 之间的最短距离，矩阵的初始值为 i，j 之间的权值，如果 i，j 不直接相连，则值为正无穷。动态规划的递归式为：d(k)*ij*= min(d(k-1)*ij*, d(k-1)*ik*+ d(k-1)*kj*) (1<= k <= n)。直观上理解，对于第 k 次更新，我们比较从 i 到 j 只经过节点编号小于 k 的中间节点(d(k-1)*ij*)，和从 i 到 k，从 k 到 j 的距离之和(d(k-1)*ik*+ d(k-1)*kj*)。Floyd 算法的复杂度是 O(n^3)。关于 Floyd 算法的理论分析，请见《算法导论》第 25 章所有结点对的最短路径问题。这里给出伪代码如下：

```
FLOYD(G)
Distance(0) = Weight(G)
For k = 1 to n
    For i = 1 to n
        For j = 1 to n

    Distance(k)ij= min(Distance (k-1)ij, Distance(k-1)ik+ Distance(k-1)kj)

Return Distance(n)
```

5.2 模式识别

5.2.1 利用分而治之（D&C）策略判断树、图的性质

对于树和图的性质，一般全局解依赖于局部解。通常可以用 DFS 来判断子问题的解，然后综合得到当前的全局结论。

值得注意的是，当我们传递节点指针的时候，其实其代表的不只是这个节点本身，而是指对整个子树、子图进行操作。只要每次递归的操作对象的结构一致，我们就可以选择分而治之（事实上对于树和图总是如此，因为子图和子树仍然是图和树结构）。实现函数递归的步骤是：首先设置函数出口，就此类问题而言，递归出口往往是 node == NULL。其次，在构造递归的时候，不妨将递归调用自身的部分视

为黑盒，并想象它能够完整解决子问题。以二叉树的中序遍历为例，函数的实现为：

```
void InOrderTraversal(TreeNode *root) {
    if (root == NULL) {
        return;
    }
    InOrderTraversal(root->left);
    root->print();
    InOrderTraversal(root->right);
}
```

想象递归调用的部分InOrderTraversal(root->left)/InOrderTraversal(root->right)能够完整地中序遍历一棵子树，那么根据中序遍历“按中序遍历左子树；访问根结点；按中序遍历右子树”的定义，写出上述实现就显得很自然了。

例题 1 Determine if a binary tree is a balanced tree.

判断一个二叉树是否是一个平衡树。

解题分析：回顾二叉树是否平衡的定义：一颗二叉树是平衡的，当且仅当左右两个子树的高度差的绝对值不超过 1，并且左右两个子树都是一棵平衡二叉树。同时，注意特例，空树是一棵平衡二叉树。*子树平衡是全局问题的一部分，因此可以通过子问题的结果来推断全局的结果：判断高度差的绝对值不超过 1，以及递归左右子树都为平衡二叉树。*

计算高度的子函数也可以用递归来实现。首先，考虑递归的出口：当 node 为空的时候，高度应该为 0。其次，如果 node 不为空，那么这棵树的高度应该为左右子树中的高度较高者加 1。

参考解答：

```
int level(TreeNode *root){
    if(root ==  NULL)
        return 0;
    return max(level(root->left), level(root->right)) + 1;
}
bool isBalanced(TreeNode* root){
    if(root ==  NULL)
       return true;
    int factor = abs(level(root->left) - level(root->right));
    return factor < 2 && isBalanced(root->right) && isBalanced(root->left);
}
```

复杂度分析：对于这个实现，isBalanced 函数对于每个节点只运行一次。然而，level 函数会进行很多重复的计算：每次进入 isBalanced 函数，调用 level 会遍历一遍所有子节点。原因在于 level 函数没有记忆性，当输入为同一个节点时，level 函数会再次进行完整的计算。

一种改进方式是，可以考虑利用动态规划的思想，将 TreeNode 指针作为键，高度作为值，一旦发现节点已经被计算过，直接返回结果，这样，level 函数对每个节点只计算一次。另一种更为巧妙的方法是，isBalanced 返回当前节点的高度，用-1 表示树不平衡。将计算结果自底向上地传递，并且确保每个节点只被计算一次，复杂度为 O(n)。

```
#define INBALANCE (-1)
int isBalancedHelper(TreeNode *root) {
    if (root == NULL) {
       return 0;
    }
    int leftHeight = isBalance(root->left);
    if (leftHeight == INBALANCE) {
        return INBALANCE;
    }
    int rightHeight = isBalance(root->right);
    if (rightHeight == INBALANCE) {
        return INBALANCE;
    }
    if (abs(leftHeight - rightHeight) > 1) {
        return INBALANCE;
    }
    return max(leftHeight, rightHeight) + 1;    // return height
}

bool isBalancedTree(TreeNode *root) {
    if(isBalancedHelper(root) == INBALANCE)
        return false;
    else
        return true;
}
```

例题 2 Check if a binary tree is a Binary Search Tree (BST), you may assume there are no elements having the same value.

检查一个二叉树是否是一个二叉搜索树，你可以假设不会有元素拥有相同的值。

解题分析： 首先考虑 BST 的定义：对于二分查找树的任意节点，该节点存储的数值一定比左子树的所有节点的值大，比右子树的所有节点的值小。并且中序遍历能够获得从小到大排列的有序数组。

根据 BST 的中序遍历特性，非常直观的想法是：可以通过中序遍历将这棵二叉树线性化，然后遍历数据，考察数组是否有序。然而，这样的方法并不是最优的，原因在于我们至少遍历整棵树，而不是一旦发现该树不是 BST 立刻返回结果。此外，遍历完成后还需要再遍历一遍额外的数组空间以判断是否有序，故该方法整体额外开销较大。

*事实上，问题的定义符合采用 D&C 的条件：子树是 BST 的是原问题的一部分；对于递归调用的函数而言，节点代表的树与子节点代表的树是同样的结构。*考虑到对于 BST 的判断，我们需要左子树均小于当前节点，右子树均大于当前节点，故可以将当前节点作为最小或最大值传入。为了函数接口的统一，一个小技巧是：可以同时传入最小/最大值，并且将初始值设为 INT_MIN，INT_MAX，这样，其左子树所有节点的值必须在 INT_MIN 及根节点的值之间，其右子树所有节点的值必须在根节点的值以及 INT_MAX 之间。从递归的角度看，不难得出递归方式是：判断 root->value > min && root->value < max（root->value 为 INT_MIN 或 INT_MAX 的情况需要特殊处理），以及递归左右子树都为 BST。同时，注意特例，空树是一棵 BST，该特例可以最为递归的出口。

复杂度分析：根据解题分析的描述，算法需要遍历所有节点，故时间复杂度为 O(n)。

参考解答：

```
bool helper(TreeNode *root, int min, int max){
    if(!root) return true;
    if((root->val < max || (root->val == INT_MAX && root->right == NULL)) &&
        (root->val > min || (root->val == INT_MIN && root->left == NULL)) &&
        helper(root->left, min, root->val) &&
        helper(root->right, root->val, max))
            return true;
    return false;
}

bool isValidBST(TreeNode *root) {
    return helper(root, INT_MIN, INT_MAX);
}
```

例题 3 Tree1 and Tree2 are both binary trees nodes having value, determine if Tree2 is a subtree of Tree1.

Tree1 和 Tree2 都是节点拥有值的二叉树，判断 Tree2 是否是 Tree1 的子树。

解题分析： 根据题意，比较容易想到的是：我们需要实现一个 matchTree 辅助函数，用以判断根节点为 root1 和 root2 的两棵树是否完全相等。*判断两棵树相等的定义符合采用 D&C 的条件：原问题的答案取决于左子树右子树都相等这两个子问题。*对于递归的实现，出口为 root1 == NULL && root2 == NULL，递归函数需要判断当前节点是否相等，左子树是否相等和右子树是否相等。

其次，考虑在什么情况下我们需要调用 matchTree：当 Tree1 的当前节点与 Tree2 的根节点相等时，我们有兴趣调用 matchTree 判断以 Tree1 当前节点为根的子树是否与 Tree2 相等。那么如果不相等怎么办？我们同样可以利用递归的思想，将当前节点的左子树或右子树是否包含 Tree2 作为子问题，通过递归调用自身获得结果。

复杂度分析：假设 Tree1 有 M 个节点，Tree2 有 N 个节点。最坏情况下，对于 Tree1 的每个节点，都需要调用 matchTree 函数，并且 matchTree 在基本遍历完 Tree2 后才能返回结果。此时，时间复杂度为 O(MN)。

参考解答：

```
bool subTree(TreeNode *root1, TreeNode *root2){
    if (root2 == NULL) {
        return true;
    }
    if (root1 == NULL) { //we have exhauste the root1 already
        return false;
    }
    if (root1->val == root2->val) {
        if (matchTree(root1, root2)) {
            return true;
        }
    }
    return isSubTree(root1->left, root2) || isSubTree(root1->right, root2);
}

bool matchTree(TreeNode *root1, TreeNode *root2){
    if (root1 == NULL && root2 == NULL) {
        return true;
    }
```

```
    if (root1 == NULL || root2 == NULL) {
        return false;
    }
    if (root1->val != root2->val) {
        return false;
    }
    return matchTree(root1->left, root2->left) &&
        matchTree(root1->right, root2->right);
}
```

例题 4 Compute the depth of a binary tree.

计算一个二叉树的深度。

解题分析： 回顾树的高度定义：从根节点到某个节点的路径长度称为该节点的层数（level），根节点为第 0 层，非根节点的层数是其父节点的层数加 1。树的高度定义为该树中层数最大的叶节点的层数加 1。*判断树的高度符合 D&C 的 条件：对于某个节点，其高度为左子树和右子树高度的较大者加 1，即原问题依赖于两个子问题。*对于递归的实现，出口为传入节点为空，此时应该返回高度 0。

复杂度分析：该算法遍历树的所有节点，故复杂度为 O(n)。

参考解答：

```
int treeDepth(TreeNode *node) {
    if (node == NULL)
        return 0;
    else
        return max(treeDepth(node->left), treeDepth(node->right)) + 1;
}
```

5.2.2 树的路径问题

有一类关于树的问题是，要求找出一条满足特定条件的路径。对于这类问题，通常都是传入一个 vector 记录当前走过的路径（为尽可能模版化，统一记为 path），传入 path 的时候可以是引用，可以是值，具体见下面的分析。还需要传入另一个 vector 引用记录所有符合条件的 path（为尽可能模版化，统一记为 result）。注意，result 可以用引用或指针形式，相当于一个全局变量，或者就开辟一个独立于函数的成员变量。由于 path 通常是 vector<int> ，那么 result 就是 vector<vector<int>>。当然，那个特定条件也是函数的一个输入变量。

在解答此类问题的时候，通常都采用 DFS 来访问，利用回溯思想，

直到无法继续访问再返回。值得注意的是，如果 path 本身是以引用（reference）的形式传入，那么需要在返回之前消除之前所做的影响（回溯）。因为传引用（Pass by reference）相当于把 path 也看做全局变量，对 path 的任何操作都会影响其他递归状态，而传值（pass by value）则不会。传引用的好处是可以减小空间开销。在本书第 8 章中，我们会进一步介绍递归和回溯。

例题 5 Get all the paths (always starts from the root) in a binary tree, whose sum would be equal to given value.

获取一个二叉树中的所有路径，总是从根开始，路径之和应该等于一个给定的值。

解题分析：*正如上文的叙述，寻找满足条件的路径，用 path 记录当前走过的路径，用 result 记录所有符合条件的 path，用 DFS 进行探索。*对于递归函数，传入的节点相当于当前子树的根：当传入节点为空时，说明我们走完了当前的 path，直接返回，即达到递归函数的出口。由于根节点必须要出现在 path 中，所以我们先将当前节点 push 到 path 中去。此时，如果当前节点的数值等于 sum，说明找到了一个可行解，故把该递归状态下的 path 加入到 answer 中。进一步，我们需要调用函数自身解决左子树和右子树的子问题。由于当前节点已经加入 path，那么自然传递给子问题的 sum 变为 sum - root->val，根的左右孩子成为了子问题的根。

复杂度分析：时间上，上述解法遍历每个节点，故时间复杂度为 O(n)。空间上，如果 path 以引用的方式传入，则额外空间为 O(n)；如果 path 以值的方式传入，则在递归函数的底层会有 2^h 个 path 拷贝，h 为树的高度。故复杂度为 O(2^h*n)。

参考解答：

这里对 path 用传值的方法。

```
void pathSumHelper(vector<int> path, vector<vector<int>> &result, TreeNode *
root, int sum){
    if(root == NULL)
        return;
    path.push_back(root->val);
    if(root->val == sum){
        answer.push_back(path);
    }
    pathSumHelper(path,answer,root->left, sum - root->val);
    pathSumHelper(path,answer,root->right, sum - root->val);
}
```

```
vector<vector<int> > pathSum(TreeNode *root, int sum) {
    vector<int> path;
    vector<vector<int>> result;
    pathSumHelper(path, result, root, sum);
    return result;
}
```

例题 6 Get all the paths (always starts from the root and ends at leaf) in a binary tree, whose sum would be equal to given value.

获取一个二叉树中的所有路径，总是从根和叶子的末端开始，路径之和应该等于一个给定的值。

解题分析：本题的处理办法基本与上题完全一致，唯一的区别在于现在要求一定是从根到叶子的路径（root-to-leaf paths），故当 path 和为给定值的时候，还需要判断当前节点是否是叶节点。

复杂度分析：时间、空间复杂度均为 O(n)。

参考解答：

这里对 path 用传引用的方法。

```
void pathSumHelper(vector<int> path, vector<vector<int>> &result, TreeNode *
root, int sum){
    if(root == NULL)
        return;
    path.push_back(root->val);
    if(root->left == NULL && root->right == NULL && root->val == sum){
        result.push_back(path);
    }
    pathSumHelper(path,answer,root->left, sum - root->val);
    pathSumHelper(path,answer,root->right, sum - root->val);
    path.pop_back();
}
vector<vector<int> > pathSum(TreeNode *root, int sum) {
    vector<int> path;
    vector<vector<int>> result;
    pathSumHelper(path, result, root, sum);
    return result;
}
```

例题 7 Get all the paths (from any node to any other node with deeper level) in a binary tree, whose sum would be equal to given value.

获取一个二叉树中的所有路径，总任何一个节点到另一个更深层

级的节点，路径之和应该等于一个给定的值。

解题分析： 经过前两道题目的分析，本题的大体思路应该已经比较容易想到了：*用 path 记录当前走过的路径，用 answer 记录所有符合条件的 path，利用 DFS 进行左子树和右子树的探索。*本题的特别之处在于，所求的 path 并不一定需要从 root 开始，即之前的题目都是知道固定的起始点，寻找终点，而本题起始点不定。进一步考虑 path 中存放的路径，该路径是由 root 到当前节点经过的所有节点，且当前节点一定是 path 的终点。那么，不妨换一个角度思考：以当前节点为终点，是否存在一个起始节点，使得路径上的节点数字和为给定值。这样，对于每个 path 数组，应该从数组尾（当前节点，即终点)反向搜索,寻求起始节点。一旦找到,则将该段 path 加入 answer。

复杂度分析：对于处于第 i 层的节点，从终点往根节点反向搜索需要的复杂度为 i。对于二叉树，第 i 层有 2^i 个节点，故复杂度为 i*2^i。对于深度为 d 的二叉树，整体复杂度为：

1*2^1+2*2^2+…+d*2^d = 2(d - 1)*2^d + 2 又 d = logn，故整体时间复杂度为 O(nlogn)。

参考解答：

```
void pathSumHelper(vector<int> path, vector<vector<int>> &answer, TreeNode *
root, int sum){
    if(root == NULL)
        return;
    path.push_back(root->val);
    int tempSum = sum;
    vector<int> partialPath;
    for (int i = (int)path.size() - 1; i >= 0; i--) {
        tempSum -= path[i];
        partialPath.push_back(path[i]);
        if (tempSum == 0) {
            answer.push_back(partialPath);
        }
    }
    pathSumHelper(path, answer,root->left, sum);
    pathSumHelper(path, answer,root->right, sum);
}
vector<vector<int> > pathSum(TreeNode *root, int sum) {
    vector<int> path;
    vector<vector<int> > answer;
    pathSumHelper(path, answer,root,sum);
    return answer;
}
```

5.2.3 树和其他数据结构的相互转换

这类题目要求将树的结构转化成其他数据结构，例如链表、数组等，或者反之，从数组等结构构成一棵树。前者通常是通过树的遍历，合并局部解来得到全局解，而后者则可以利用 D&C 的策略，递归将数据结构的两部分分别转换成子树，再合并。

例题 8 Covert a binary tree to linked lists. Each linked list is correspondent to all the nodes at the same level.

将一个二叉树转换为链表。每个链表都和相同层级的所有节点对应。

解题分析：*本题相当于层次遍历，当遍历完一层时将构成的链表加入链表集合*。本题的核心在于如何判断某层已经遍历完成。回顾层次遍历，我们将节点逐次放入优先队列，然后一次弹出，并将左右子节点放入队列。我们可以引入一个特别的哑节点，用来作为层与层之间的分隔符。每当发现当前节点是哑节点，则说明当前层已经遍历完毕，也意味着下一层的所有节点都已经进入队列。此时，立刻再推送一个哑节点入队，作为下一层的分隔符。

复杂度分析：算法需要层次遍历树，故时间复杂度为 O(n)。同时，需要额外的 O(n)空间将树中的元素存储到链表。

参考解答：

```
bool isDummyNode(TreeNode *node)
{
    return (node->left == node);
}

vector<list<TreeNode *>> linkedListsFromTree(TreeNode *root)
{
    vector<list<TreeNode *>> result;
    list<TreeNode *> levelList;
    queue<TreeNode *> nodeQueue;
    TreeNode *currentNode;
    TreeNode dummyNode;
    dummyNode.left = &dummyNode;


    if (!root) {
        return answer;
    }
```

```
    nodeQueue.push(&dummyNode);
    nodeQueue.push(root);

    while (!nodeQueue.empty()) {
        currentNode = nodeQueue.front();
        if (isDummyNode(currentNode)) {
            if (!levelList.empty()) {
                answer.push_back(levelList);
                levelList.clear();
            }
            nodeQueue.pop();
            if (nodeQueue.empty()) {
                break;
            } else {
                nodeQueue.push(&dummyNode);
            }
        } else {
            levelList.push_back(currentNode);
            if (currentNode->left) {
                nodeQueue.push(currentNode->left);
            }
            if (currentNode->right) {
                nodeQueue.push(currentNode->right);
            }
            nodeQueue.pop();
        }
    }

    return result;
}
```

例题9 Convert a binary tree to a linked list without additional space.

将一个二叉树转换为一个链表而不需要额外的空间。

解题分析：*将树转换成链表，用DFS将左右子树分别转换成链表，再合并这两个局部解。*在这里，局部解可以用节点指针表示，因为链表头节点的指针就可以代表整个链表（正如二叉树的根就可以代表整个二叉树一样）。在合并局部解时，注意处理边界条件。

复杂度分析：算法需要遍历树，故时间复杂度为O(n)。

参考解答：

```
TreeNode *convertToList(TreeNode *root){
```

```
    if(!root)   return NULL;

    TreeNode *leftTail = convertToList (root->left);
    TreeNode *rightTail = convertToList (root->right);

    if(leftTail)    leftTail->right = root->right;
    if(root->left)  root->right = root->left;
    root->left = NULL;

    if(rightTail)   return rightTail;
    else if(leftTail)   return leftTail;
    else    return root;
}
```

例题 10 Convert a sorted array(increasing order) to a balanced BST.

将一个有序的数组（递增顺序）转换为一个平衡的 BST。

解题分析：*将数据结构分成左右两部分，分别构造两个二叉树，再用中间数与这两个局部解合并得到当前的全局解。*因为数组已经排序好，因此这个二叉树一定满足 BST 的条件（左边的任何数一定小于中间数，小于右边的任何数），在选择分界点的时候选择中点，那么也就能满足平衡条件。

复杂度分析：算法遍历整个数组，故时间复杂度为 O(n)，需要额外 O(n)空间存储树的节点。

参考解答：

```
TreeNode* helper(vector<int>num, int first, int last){
    if(first > last){
        return NULL;
    }
    if (first == last) {
        TreeNode* parent = new TreeNode(num[first]);
        return parent;
    }
    int mid = (first + last) / 2;
    TreeNode *leftchild = helper(num, first, mid - 1);
    TreeNode *rightchild = helper(num, mid + 1, last);
    TreeNode *parent = new TreeNode(num[mid]);
    parent->left = leftchild;
    parent->right = rightchild;
    return parent;
}
```

```
TreeNode *sortedArrayToBST(vector<int> &num) {
    if(num.size() == 0)
        return NULL;
    if(num.size() == 1){
        TreeNode* parent = new TreeNode(num[0]);
        return parent;
    }
    int first = 0;
    int last = (int)num.size() - 1;
    return helper(num, first, last);
}
```

5.2.4 寻找特定节点

此类题目通常会传入一个当前节点，要求找到与此节点具有一定关系的特定节点：例如前驱、后继、左/右兄弟等。

对于这类题目，首先可以了解一下常见特定节点的定义及性质。在存在指向父节点指针的情况下，通常可以由当前节点出发，向上倒推解决。如果节点没有父节点指针，一般需要从根节点出发向下搜索，搜索的过程就是 DFS。

例题 11 In-order traverse a binary tree with parent links, find the next node to visit given a specific node.

中序遍历带有父链接的一个二叉树，找到给定节点的下一个节点。

解题分析：*根据中序遍历的性质，我们可以分几种情况来考虑目标节点与给定节点的关系*：（1）如果该节点有右子树，那么，中序后继节点就是右子树中最左的节点。（2）如果没有右子树，那么考虑该节点与其父节点的关系：如果它是父节点的左孩子，那么，父节点就是它的后继。(3)如果它是父节点的右孩子，那么我们可以向上倒推，直到某个节点（或者不存在这样的节点，返回空指针）是其父节点的左孩子。

复杂度分析：最坏情况下，考虑一棵树只有右孩子，而输入恰好是最右节点。在这个情况下，我们需要向上倒推遍历所有节点，此时复杂度为 O(n)。平均情况下，复杂度为 O(h)，h 为树的高度。

参考解答：

```
TreeNode *leftMostNode(TreeNode *node)
{
    if (!node) {
        return NULL;
    }
    while (node->left) {
        node = node->left;
    }
    return node;
}

bool isLeftChild(TreeNode *node, TreeNode *parent)
{
    return (parent->left == node);
}
TreeNode *inOrderSuccessor(TreeNode *node)
{
    if (!node) {
        return NULL;
    }
    if (node->right) {
        return leftMostNode(node->right);
    }
    TreeNode *parent = node->parent;
    while (parent && !isLeftChild(node, parent)) {
        node = parent;
        parent = node->parent;
    }
    return parent;
}
```

例题 12 In-order traverse a binary tree without parent links, find the next node to visit given a specific node.

中序遍历不带父链接的一个二叉树，找到给定节点的下一个节点。

解题分析：对于该节点有右子树的情况，由于我们不需要利用父节点信息倒推，故搜索过程与上题一致：中序后继节点就是右子树中最左的节点。*在其他情况下，我们需要从根部开始搜索。*对于根节点，我们应该如何判断继续搜索左子树还是右子树？对于 BST，中序遍历的后继节点就是值比当前节点大的所有节点中最小的那个。因此，一旦根节点大于当前节点，我们存储当前节点，并且往数值减小的方向搜索（左子树）；一旦根节点小于当前节点，我们继续往数值增大的

方向搜索（右子树）。这样，当算法执行完成，我们存储的最后一个节点一定恰好大于给定节点，即是给定节点的中序遍历后继。

复杂度分析：算法复杂度同上题：平均情况下，复杂度为 O(h)，h 为树的高度。

参考解答：

```
TreeNode *inOrderSuccessor(TreeNode *node, TreeNode *root)
{
    if (!node) {
        return NULL;
    }
    if (node->right) {
        return leftMostNode(node->right);
    }

    TreeNode *successor = NULL;
    while (root) {
        if (root->val > node->val) {
            successor = root;
            root = root->left;
        } else {
            root = root->right;
        }
    }
    return successor;
}
```

例题 13 In a binary tree without parent links, find the closest common ancestor.

在不带父链接的一个二叉树中，找到最近的共同祖先。

解题分析：如果有指向父节点的指针，那么就一直上溯，直到找到一样的父节点。*如果没有指向父节点的指针，那么从根节点开始用 DFS 选择性的进行搜索*：如果这两个节点都在某个节点的左子树中，那么解一定在此节点的左子树中；类似的，如果两个节点都在某个节点的右子树中，那么解一定在此节点的右子树中。换言之，所求的解一定将给定的两个节点分别分割在左右子树中。

那么，我们可以用一个辅助函数来判断一个节点是否隶属于子树。在主函数中，从根开始判断两个节点是否处于同一边的子树：如果是，那么所求的解一定属于该子树，我们可以沿子树方向再往下走一层；如果不是，那么当前根就是答案。

复杂度分析：假设树高度为 h，对于第 i 层，判断两个节点是否

分别处于不同子树需要搜索 2^(h-i)次。整体复杂度为 1+2+2^2 + … + 2^h，即 O(2^(h+1))。

参考解答：

```
TreeNode *commonAncestor(TreeNode *root, TreeNode *p, TreeNode *q){
    if (covers(root->left, p) && covers(root->left, q)) {
        return commonAncestor(root->left, q, p);
    }
    if (covers(root->right, p) && covers(root->right, q)) {
        return commonAncestor(root->right, q, p);
    }
    return root;
}

bool covers(TreeNode *root, TreeNode *p){
    if (root == NULL) {
        return false;
    }
    if (root == p) {
        return true;
    }
    return covers(root->left, p) || covers(root->right, p);
}
```

例题 14 Find the immediate right neighbor of the given node, with parent links given, but without root node.

找到给定的节点的直接的右邻居，给定了父链接，但没有给定根节点。

解题分析：直接的右邻居（Immediate Right Neighbor）定义为在给定节点右侧且与给定节点在同一层。*由于给定了父指针，故可以利用父指针向上倒推。*根据定义，如果当前节点是父节点的右孩子，我们继续倒推，直到当前节点是某个父节点的左孩子，并且该父节点存在右子树。这样，我们立即进入该子树，找到与给定节点处于相同层的最左节点即可。

复杂度分析：倒推需要复杂度 O(h)，下降进入子树寻找与给定节点处于相同层的最左节点也需要复杂度 O(h)，故整体复杂度为 O(h)，h 为树高。

参考解答：

```
TreeNode *descend(TreeNode *node, int level)
{
    while (node && level > 0) {
        if (node->left) {
            node = node->left;
        } else if (node->right) {
            node = node->right;
        } else {
            node = NULL;
        }
        level--;
    }
    return node;
}

TreeNode *findRightNeighbor(TreeNode *node)
{
    int level = 0;
    TreeNode *parent = node->parent;
    if (!node) {
        return NULL;
    }

    while (parent != NULL) {
        if (isLeftChild(node, parent) && parent->right) {
            break;
        }
        level++;
        node = parent;
        parent = node->parent;
    }

    if (parent == NULL) {
        return NULL;
    } else {
        return descend(parent->right, level);
    }
}
```

5.2.5 图的访问

关于图的问题一般有两类。一类是 5.1“知识要点”中提到的关于图的基本问题，例如图的遍历、最短路径、可达性等；另一类是将问题转化成图，再通过图的遍历解决问题。第二类问题有一定的难度，但也有一些规律可循：如果题目有一个起始点和一个终止点，可以考虑看成图的最短路径问题。

例题 15 Given two words (start and end), and a dictionary of words, find the length of shortest sequences of words from sta*rt to end, such that at each step only one letter is changed.*

e.g: start word: hat stop word: dog
dictionary{cat, dot, cot, fat}
sequence: hat->cat->cot->dot->dog
给定了两个单词，start 和 end，还有单词的一个字典，找到从 start 到 end 的最短的单词序列，并且在每一步中只有一个字母发生变化。例如：起始单词：hat 结束单词：dog
dictionary{cat, dot, cot, fat}
sequence: hat->cat->cot->dot->dog

解题分析： 这是一道比较有技巧性的题目。由于有一个起始点，一个终止点，需要求一个最短路径，故可以尝试用图来解决：我们可以将所有的单词看做一个图的节点，如果一个单词变换其中的一个字母就能变成另外一个在字典中的单词，那么我们就可以用一个有向箭头从原来的单词指向变换后的单词。那么，*最短的变换次数就转化为从一个单词所在的节点到另一个单词所在的节点的最短路径。*

然而，第一个难点在于如何构造图，并且建立两个节点之间的联系。比较容易想到的节点联系方式是：对于某个字典中的单词，遍历整个字典，判断其他单词与该单词是否只有一个字母不同，这样做的时间复杂度是 O(n*1)，1 是单词长度。另一个比较巧妙的做法是，对于给定单词的每个字母，用 26 个字母逐个全部替换一遍，并判断字典中是否有替换后的单词。由于判断是否存在可以用哈希表，时间复杂度为常数，故这样做的时间复杂度是 O(26*l)。通常情况下，可以假定 n 大于 26，则第二种方法更好。

在确定了如何构造图的基础上，我们考虑如何进行 BFS 寻找最短

路径。我们在实现 BFS 的时候可以用一个队列作为辅助：将当前能到达的所有节点放进去，然后放一个空节点作为层次的分割；然后取出队首，将队首所指的节点都放入队尾；如果队首为空节点，可知这是层次的分割符，则路径长度加 1。注意，题目中最短的要求使得我们的路径中不能有环，换言之，一旦加入过队列的节点，不能再次入队。

复杂度分析：对于每个节点构图需要的时间复杂度是 O(26*l*)，*BFS 不重复地至多走过 n 个节点，故算法复杂度为 O(26*l*n)。

参考解答：

```
int ladderLength(string start, string end, unordered_set<string> &dict) {
    int count = 1;    // used to store the answer;
    unordered_set<string> check;    // contains all of the words that is in t
he path, no duplicate in the path
    queue<string> myque;    // used to level traverse
    myque.push(start);
    myque.push("");    // used to determine which level it is
    string word;
    string mid;    // used to intermediate result
    while(!myque.empty()) {
        word = myque.front();
        myque.pop();
        if(word == end){
            return count;
        }
        if(word.size() == 0 && !myque.empty()) {
            count++;
            myque.push("");
        }
        else if(!myque.empty()) {
            for(int i = 0; i < word.size(); i++) {
                mid = word;
                for(int j = 'a'; j < 'z'; j++){
                    mid[i] = (char)j;
                    if(check.find(mid) == check.end() && dict.find (mid) != d
ict.end()) {
                        myque.push(mid);
                        check.insert(mid);
                    }
                }
            }
        }
    }
return 0;}
```

5.3 工具箱

1．二叉树类

一个最基本的二叉树类可以如下定义：

```
class TreeNode {
public:
     TreeNode *left;
     TreeNode *right;
     TreeNode *parent;
     int val;
};

class BinaryTree {
public:
    BinaryTree(int rootValue);
    ~BinaryTree();
    bool insertNodeWithValue(int value);
    bool deleteNodeWithValue(int value);
    void printTree();
private:
    TreeNode *root;
};
```

2．平衡二叉树

关于平衡二叉树的理论讨论请见：

《算法导论》(Introduction to Algorithms, 2nd Edition), Thomas H. Corman, Charles E.Leiserson, Ronald L.Rivest, Clifford Stein，第 13 章，红黑树

3．二叉树的遍历

DFS 遍历：树的中序遍历、前序遍历和后序遍历。

```


void preOrderTraversal(TreeNode *root) {
    if (!root) {
        return;
    }
    visit(root);
    preOrderTraversal(root->left);
    preOrderTraversal(root->right);
}
```

```
void inOrderTraversal(TreeNode *root) {
    if (!root) {
        return;
    }
    inOrderTraversal(root->right);
    visit(root);
    inOrderTraversal(root->left);
}

void postOrderTraversal(TreeNode *root) {
    if (!root) {
        return;
    }
    postOrderTraversal(root->left);
    postOrderTraversal(root->right);
    visit(root);
}
```

BFS 遍历

```
void levelTraversal(TreeNode *root)
{
    queue<TreeNode *> nodeQueue;
    TreeNode *currentNode;
    if (!root) {
        return;
    }
    nodeQueue.push(root);
    while (!nodeQueue.empty()) {
        currentNode = nodeQueue.front();
        processNode(currentNode);
        if (currentNode->left) {
            nodeQueue.push(currentNode->left);
        }
        if (currentNode->right) {
            nodeQueue.push(currentNode->right);
        }
        nodeQueue.pop();
    }
}
```

4. 字典树类

可以如下定义一个字典树:

```
class TrieNode {
private:
    char mContent;
    bool mMarker;
    vector<TrieNode *> mChildren;
public:
    TrieNode() {
        mContent = ' '; mMarker = false;
    }
    ~TrieNode() {
    }
    friend class Trie;
};
class Trie {
public:
    Trie();
    ~Trie();
    void addWord(string s);
    bool searchWord(string s);
    void deleteWord(string s);
private:
    TrieNode *root;
};
```

5. 堆的重要函数

《算法导论》(Introduction to Algorithms, 2nd Edition), Thomas H. Corman, Charles E.Leiserson, Ronald L.Rivest, Clifford Stein，第 6 章，堆排序。

6. priority_queue

priority_queue 在 C++标准模版库(Standard Template Library，STL）中实现，使用时需要 include<priority_queue>。简要介绍如下常见函数（更多信息请参考 http://www.cplusplus.com/reference/queue/priority_queue/）：

```
bool empty() const;    // Returns whether the priority queue is empty: i.e.
whether its size is zero.
void push (const value_type& val);    // Inserts a new element in the
priority queue, effectively increasing its size by one.
```

示例：

```
priority_queue myPriorityQueue;
myPriorityQueue.push(20);
myPriorityQueue.push(50);
myPriorityQueue.push(30);    // queue contains 3 elements, will be accessed
in the order of 50, 30, 20

void pop();    // Removes the element on top of the priority queue, effectiv
ely reducing its size by one.
const value_type& top() const;    // Returns a constant reference to the
top element in the priority queue.
```

示例：

```
priority_queue<int> myPriorityQueue;
myPriorityQueue.push(30);
myPriorityQueue.push(10);
myPriorityQueue.push(50);
myPriorityQueue.push(40);

cout << "Popping out elements...";
while (!myPriorityQueue.empty()) {
    cout << ' ' << myPriorityQueue.top();
    myPriorityQueue.pop();
}
cout << endl;        // output: Popping out elements... 50 40 30 10
```

第 6 章 位操作

6.1 知识要点

对于网络、操作系统、嵌入式系统等相关职位的面试，位运算也是常见的题目类型之一。所谓的位运算，是指按二进制进行的运算。常见运算包括求反、与运算、或运算、异或运算及位移。关于位运算的真值表，请参考 6.3“工具箱”给出的链接。

在 C/C++中，基本的位运算符总结如下，其中运算符优先级为从上到下递减，且<<和>>优先级相同，如表 6-1 所示。

表 6-1 位运算符概览

操 作 符

功 能

用 法

~

位求反

~var

<<

左移运算（相当于乘法）

var << position

>>

右移运算（相当于除法）

var >> position

&

位与

var1 & var2

^

位异或

var1 ^ var2

|

位或

var1 | var2

需要注意的是,位运算符只能用在带符号或无符号的char、short、int 与 long 类型上。在实际应用中，建议用 unsigned 整型操作数，以免带符号操作数因为不同机器导致的结果不同：无符号数左移/右移默认移入的新位是 0。对于符号数，当最高位是 1（代表负数）时，有的机器认为右移移入的新位是 1。此外，复杂的位运算建议都用括号强制计算顺序，而不是依赖于优先级，这样做可以增加可读性并避免错误。

用十六进制（hex）定义一个变量如下所示：

unsigned short value = 0xFFFF;等价于二进制(binary)定义：

unsigned short value = 0b1111111111111111;等价于十进制定义：

unsigned short value = 65535;

6.2 模式识别

6.2.1 基本的位操作

最基本的操作包括获取位、设置位和清除位。获取位可以利用&1：&(0x1 << pos)；设置位可以利用|1: | (0x1 << pos) ；清除位可以利用&0: &(～(0x1 << pos))。判断某位是否相同用^：(A & (0x1 << pos)) ^ (B & (0x1 << pos))。

例题 1 Determine the number of bits required to convert integer A to integer B

确定将一个整数 A 转换为整数 B 所需要的位数。

解题分析：通常情况下，当需要比较某个位是否相同时，需要用位异或。如果 A 和 B 中某位不相同，则位异或得 1。所以，*题目等价为 A 异或 B，所得结果里有几位是 1。*统计有几位是 1 可以通过反复 Get lowest bit 和右移 1 位（除 2）实现。另一个常用技巧是 *n &=(n-1)相当于 clear 最低的一位 1*。事实上，可以用一个哈希表在 O(1)时间内得到一个整数中有多少个比特为 1，具体见 6.3 小节“工具箱”。

复杂度分析：假设整型有 n 位，则复杂度为 O(n)。

参考解答：

```
int numberOfDifferentBits(int A, int B) {
    int diff = A ^ B;
    int count = 0;
    while (diff > 0) {
        count += diff & (0x1);
        diff = diff >> 1;
/* or do the following:
        diff &= (diff - 1); // clear lowest 1 in diff
        count++;
*/
    }
    return count;
}
```

例题 2 Given an array of integers, every element appears twice except for one. Please write a function to find that single one.

给定整数的一个数组，每个元素都出现了两次，只有一个元素例

外。请编写一个函数找到那个单个的元素。

解题分析：当遇到某些题目需要统计一个元素集中元素出现的次数，应该直觉反应使用哈希表，key 是元素，value 是出现的次数。扫描整个数组建立哈希表，再次扫描表看哪个元素出现了一次。这样做的时间复杂度为 O(n+n)。

事实上，能在面试中给出这样的解答已经足够好，而且这种解法具有普适性，应该首先想到。对于这道特殊的题目，能不能做的更好？我们考虑两个数如果相等，二进制表示有什么特点？很明显，当然是二进制表示每位比特都相等。能否通过某种二进制操作把两个相同整数变成常数？答案是异或：*相同整数异或得 0*。如何把这个性质利用到本题？如果我们异或所有得元素，则出现两次的数都相互抵消，最后留下的就是单独的那个。

复杂度分析：扫描数组一次，复杂度为 O(n)。

参考解答：

```
int singleValue(vector<int> array) {
    int value = 0;
    for (int i = 0; i < array.size(); i++) {
        value ^= array[i];
    }
    return value;
}
```

例题 3 Explain what the following code does: (n & (n-1)) == 0

说明如下的代码做什么事情：(n & (n-1)) == 0

解题分析：之前我们看到 n &= (n-1)相当于 clear 最低的一位 1，我们用这个方法统计一个整数的二进制表示中有多少个 1。那么 (n & (n-1)) = 0 意味着该整数的二进制表示中只有一个 1，即它是 2 的次方。事实上，如果不能立刻看出结果，不妨尝试从 1 开始列举 n 的值，看看有什么规律。根据观察到的结果再来倒推表达式的含义。

6.2.2 位掩码

对于这类题目，需要做的全部工作，就是选择合适的位掩码（bit mask），然后与给定的二进制数进行基本位操作。而掩码，通常可以通过对～0，1 进行基本操作和加减法得到。例如，我们要构造一个第

i 到第 j 位为 0，其他位为 1 的位掩码，则可以对～0 进行左移操作获得形如 111…0000 的掩码，再对～0 进行右移操作，获得形如 000…111 的掩码，最后通过位或（此处相当于相加）得到最终的位掩码。

在寻求得到一个特定的掩码时，还是利用最基本的获取位、设置位或清除位得到所需掩码的形态。另外，应当尽可能避免直接出现常数，比如使用 32-i 这样的情况（这里默认想要操作一个 32 位的整型），而应当定义一个意义明确的宏，以提高可读性：#define INT_BIT_LENTH (32)。

例题 4 Given N and M, 32bit integers, how to set i to j bits (bit position as 1,2,3,…32) of N as the value of bits in M.

For example, N = 00000000000000000000000001111011, M = 00000000001000000011000000011000, i = 10, j = 20, then the result should be: 00000000001000000011000001111011

给定了 32 位整数 N 和 M,如何将 N 的 i 到 j 位(位位置是 1,2,3,…32）设置为和 M 中相同位的值。例如，N = 00000000000000000000000001111011, M = 00000000001000000011000000011000, i = 10, j = 20；那么，结果应该是 00000000001000000011000001111011

解题分析： 我们首先根据题意重现需要做的操作：（1）我们需要从 M 中 get 第 i 到第 j 个位（2）我们需要 clear N 中第 i 到第 j 个位（3）我们需要 set N 中第 i 到第 j 个位。

对于（1），根据基本操作，get bit 需要&1，所以与 M 进行操作的位掩码在第 i 到第 j 位应当为 1，其他位为 0。对于（2），根据基本操作，clear bit 需要&0，所以与 N 进行操作的位掩码在第 i 到第 j 位应当为 0，其他位为 1。注意，这个位掩码刚好是前一项操作位掩码的位反运算。对于（3），我们只需要将（1），（2）的操作结果进行位或即可。构造所需位掩码的过程如前所述，对～0 进行基本操作和加减法即可。

参考解答：

```
#define INT_BIT_LENGTH (32)
void setBits(unsigned int &N, unsigned int M, int i, int j) {
    unsigned int max = ~0;
    unsigned int bitMask = (max << (INT_BIT_LENGTH - i) |
                            max >> j); //11..100..011..1
    N = (M & (~bitMask)) | (N & bitMask);
}
```

例题 5 Swap the neighboring odd and even bits in a integer (bit position as 1,2,3,…32). 将一个整数中相邻的奇数位和偶数位交换（位位置是 1,2,3,…32）

解题分析：同样的，我们首先根据题意重现我们需要做的操作：（1）我们需要 get 奇数位，根据基本操作易得需要与形如 1010…10 的位掩码做位与。（2）get 偶数位，根据基本操作易得需要与形如 0101…01 的位掩码做位与。（3）swap 意味着将（1）中得到的结果右移一位，与（2）中得到的结果左移一位，然后做位或操作。由此可见，*对于需要进行比特操作的题目，对题目要求进行分步，然后选择合适的*位掩码，最后与给定二进制数进行基本位操作都是解题的关键。

参考解答：

```
int swapBits(int input) {
    return ((0xaaaaaaaa & input) >> 1) | ((0x55555555 & input) << 1);
}
```

6.3 工具箱

1．位运算的定义

参考链接：

http://en.wikipedia.org/wiki/Bitwise_operation 给出的位运算定义。

2．关于位运算的深入讨论请参考链接：

http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetTable

第 7 章 面向对象的设计

7.1 知识要点

7.1.1 设计题解答要领

对于初级程序员的面试，最难的部分可能就是所谓的设计题。设计题可以分成两个类别：系统架构设计和利用面向对象编程原理进行程序设计。前者所涉及的技术往往包括数据库、并发处理和分布式系统等等，对于经验要求和知识要求比较高。我们先来看看系统设计的面试流程。

1．题目描述。

往往非常简单，如：设计一个 XX 系统。或者：你有没有用过 XXX，给你看一下它的界面和功能，你来设计一个。

2．阐述题意。

面试者需向面试官询问系统的具体要求。如，需要什么功能，需要承受的流量大小，是否需要考虑可靠性、容错性等等。

3．面试者提供一个初步的系统设计。

4．面试官对初步的系统中提出一些后续的问题：如果要加某个功能怎么办，如果流量大了怎么办，如何考虑一致性，如果机器挂了怎么办。

5．面试者根据面试官的后续问题逐步完善系统设计。

6．完成面试。

总体特点是以交流为主，以画图和代码为辅。

根据我们面试别人和参与面试的经验，先从面试官的角度给出一些考量标准：

适应变化的需求（Adapt to the changing requirements）。

设计干净、优美，考虑周到的系统（Produce a system that is clean, elegant, well thought）。

解释为何这么实现（Explain why you choose this implementation）。

对自己的能力水平很熟练（Be familiar with your experience level to make decisions）。

在一些高层结构和复杂性方面有设计（Answer in high level

of scale and complexity)。

按照评分体系的化，分成下面 4 个等级

Scoring

Candidate

Criteria

1.0

Bad

No sense of requirement, no scoping

2.0

Pool

Limited knowledge, common sense

3.0

Good

Reasonable Solution, explain clearly

4.0

Great

Out of expectation, well thoughtful, tradeoff

其实大家大可不必追求完美，在真正的面试中，没有人能对答如流，往往面试官也会给出善意的提示，就算你没回答某个子问题，在面试后的评价中也会综合衡量，跟其他的面试者比较，最终打出一个分数。我认为很多人在 2 到 3 分左右，当然我们目标是尽量在 3 分以上。

7.1.2 模拟面试

下面我就来做个模拟面试（Mock Interview），拿一道很经典的 TinyURL 举例：

TinyURL 是说给你一个长 URL，通过某一种编码，你把它压缩成 5 个字母（数字）长度的 code，然后当给出短链接，要能还原出来原来的 URL。举个例子：

原始 URL：http://www.zhihu.com /abc?user=12345

TinyURL: http://t.cn/xcef0

当访问 TinyURL，会自动跳转到原始 URL。

下面我就按 4 个等级分别给出解答。

等级 1：最直接想到的是用一个数据库，把原始 URL 存入数据库中，ID 就是自增主键，避免重复。当访问 TinyURL 时候就去数据库查询，当查询成功就返回原始 URL，没有就返回空。这是不是很简单？

想到这还是太容易了，虽然这是个正确的方法，但没考虑全面。比如这里面编码是用数字，没有很好利用 5 个字节的表示空间，然后在性能方面，每次都是查询数据库这样效率不高。那么我们看下面的方法。

等级 1.5：上面使用数字能表达的 URL 最多就是 0－99999，即 10 万条，那么使用 Base 64 编码，所谓 Base 64 是有 64 个不同字符构成，比如 a 到 z，A 到 Z，0 到 9，再加上 2 个特殊符号，比如_、-，这样能表示的就是 64^5 = 2^30，大约 10 亿的大小。这样就极大提高了空间容量。

然后为了提高性能，可以加上缓存，缓存大小是有限的，我们如何让一些不常用的查询从缓存中替换出去，这就涉及缓存淘汰算法，常用的 LRU（Least Recent Use）和 LFU（Least Frequent Use），一个是最近最少访问，一个是最不频繁访问，我们在这里使用 LFU 就比较合适。

到了这里，设计还是不够好，一个是性能还是可能出现瓶颈，当我们在使用单个数据库，如果查询依然很多，到每秒 1 万次以上怎么办？使用的是哪种数据库，如果数据库当机怎么办？

等级 2：使用 key-value 存储的数据库，所谓的 key-value 对就是 NoSQL 的一种经典应用，传统数据库为了支持事务、外键，有很多需要考虑。但 key-value 型数据库，就减少了这么多限制，对这种典型通过某个 key 查询 value 的，就直接用这个合适，它的吞吐量可以比传统 MySQL 有 10 倍以上的提升。另外一个是关于如何快速生成短 code，有一个 md5 的算法，它是用在单项加密上，最直接地把一段信息加密成 128bit 大小，如果我们算一下，128bits 是 16 个 bytes，这样就超过了题目要求的 5 个，那么我们可以做一个尾部截取，但这样又会带来 hash 重合的问题。大家可以想想有没有更好的解决方法。

等级 3：下面可以在考虑分布式，可靠性的问题。刚才说了一台机器可能导致挂了就失去服务和数据。那我们可以用多台机器，通过 sharding 的方式，就是把某个 URL 取一个 hash 值再%N，这样就把这个请求分配到某台具体机器上，然后在这台机器上进行读取或者存储。

另外我们要注意一种分配不均的情况，你可以想象也许机器 1 就是非常热门，它的负载比其他的机器高很多，但总体负载又不高，如果我们简单的扩容，一方面浪费了机器，涉及到迁移成本，并且也不见得解决真正 1 号机负载高的问题。最好的做法是在 1 号机的旁边再放置一台同样规模和数据的机器，这个叫热等待（Hot Standby），通过流量的导流，实现负载均衡。对于可靠性，你想象某台机器当机，它如何做恢复呢？首先你要保证数据是有备份的，也许再另一个数据库中，然后当那台机器重启后，可以先载入上次备份的地方，也叫快照（snapshot），然后再把日志中应该重新做的操作继续做一遍，这样就保证了数据的一致性。

等级 4：其实到这一步，更多的是一些后续问题，比如你怎么解决生成全局唯一的 token 号，这里面可以用 zookeeper 作为工具，在集群中协作保证生成唯一 ID，具体算法可以参考 Paxos。此外，还可能会问如何防止 TinyURL 被爬取，刚才我们看到的例子如果是按正常顺序，就是 1、2、3……9、a、b，这样很容易被机器抓取，有个简单办法，内部你做一个乱序对应表，比如 z 对应 1，w 对应 2，b 对应 34，这样就比较难猜到你的生成规则。还可能接着问如何限制用户访问，当某个用户每分钟到达一定次数就禁止访问，可以使用一些计数器针对用户 cookie 做统计，当一分钟内到达阈值就触发屏蔽。还可能问如何实现 URL 的跳转服务，当你没有经验，可能就说自己根据 HTTP 协议做一个 Web Server，但其实不必自己造轮子，现在 Apache\Nginx 等主流服务器早就自带重定向模块，你只需要打开这个模块配置一下就可。

上面说的详细解答就是想给大家一个直观认识，遇到这种面试题在不同的层次是答道什么深度，其实这些都是可以训练出来，大家不必太担心灵活性。在面试过程中，面试官一般不会对初级程序员提出这方面的问题。因此，面向对象编程原理（OOP）是设计题中需要重点准备的。下面就对一些 OOP 方面的设计题，多一些更为基本的补充解释。

7.1.3 抽象、面向对象和解耦（Decoupling）

通常，关于 OOP，面试官会让面试者设计一个程序框架，该程序能够实现一些特定的功能。比如，如何实现一个音乐播放器，如何设计一个车库管理程序等等。对于此类问题，设计的关键过程一般包括

抽象（abstraction）、设计对象（object）和设计合理的层次/接口（decoupling）。这里，我们举一个例子简单说明这些过程分别需要做些什么。7.2“模式识别”部分给出更为具体和完整的实例。

例：设计一个音乐播放器，能够管理专辑，播放歌曲。

抽象

抽象的意思是能够根据设计要求，得出程序运行的逻辑框架以及定义需要的功能。简单来说，就是构想自己是该程序的使用者，列举实现程序某个功能需要哪些步骤。比如对于我们所说的例子，播放一首歌的运行流程可能包括：添加歌曲到音乐库，添加歌曲到播放列表，播放，删除等。对于某些系统，系统可以处于多个不同状态，每个状态的功能不同，这就需要构造有限状态机。关于有限状态机的一些参考资料请参见 7.3“工具箱”。

在抽象的过程中，往往需要和面试官进行沟通，确认需要实现什么功能。在抽象结束后，所得到的实体就应该成为一个个对象，而功能就是对象的函数接口。比如，在本例中，涉及的实体有播放器、专辑、歌曲、播放列表。函数包括添加歌曲、删除歌曲、播放、停止、暂停等等。之后，我们就要根据抽象的结果，进行对象设计，并且把所需的功能以函数接口的形式分配给不同的对象。

设计对象

通常而言，对于抽象出的每个实体，我们都应该构造一个类去描述它。对象之间的关系可能是继承，或者包含，具体分析请见 7.1.4 节。对于这里所需要实现的音乐播放器，我们就可以构造播放器、专辑、歌曲、播放列表这几个类。

设计接口

接口用于与用户进行交互，以及对象之间的交互。设计接口的核心在于明确每个对象需要实现什么功能，当上层对象调用下层对象的接口时，只需要提供相应的参数，就能够期待获得对应的结果。这样，上层对象不需要知道对方如何获得结果，下层对象也不需要知道对方拿到结果会进行什么样的操作。如此，通过设计恰当的接口我们就实现了解耦：让程序具有逻辑上的层次，每一层的对象实现特定的功能，对象可以独立地更改实现功能的方法，而不会影响上层和下层。

在设计接口的时候，我们可能需要添加一些不那么明显的辅助类，使得程序更具有层次。比如说，考虑添加歌曲这个功能，用户会通过调用播放器的添加歌曲接口，传入一首歌。当音乐库变得很复杂的时

候，我们需要判断诸如该歌曲是否已经存在，应当用怎样的数据库存储数据等等一系列问题。如果都由播放器对象来处理这些问题，会使得播放器这一层变得过为臃肿，降低了代码的可读性和可维护性。所以我们可以引入另一个类，歌曲管理器，用来处理数据相关的操作。这样，用户通过播放器接口添加歌曲，播放器调用歌曲管理器的添加接口将新歌写到数据库，歌曲管理器负责打开数据库，判断数据库中是否存在重复的歌曲，写入数据等等。当之后出于某些原因需要更换存储方式的时候，只要更改歌曲管理器的代码即可。同时，如果发现播放器中有重复的歌曲，也只需要查看歌曲管理器的实现过程有什么漏洞，而不是漫无目的地找 bug，这样也提升了程序的可维护性。

同时，在确定接口功能的时候，尽量把功能的具体实现向下层“推”。比如，播放一首歌曲可以包括打开文件，访问文件等。当用户调用播放器接口要求播放一首歌时，我们可以让播放器打开文件，也可以让播放器调用歌曲对象的播放接口，由歌曲对象负责打开文件等操作。这样做的好处在于，用户可能在任何时候播放一首新的歌曲，播放器只需要有一个指针指向当前播放的歌曲对象，如果用户需要播放新的歌曲，播放器只需要：调用当前歌曲的停止接口，设置新的当前播放，调用新歌曲的播放接口即可，而不需要在播放器层反复地进行文件操作。这样，播放器层负责切换歌曲的逻辑，而歌曲层负责具体的播放操作，两者相互独立。这样的好处在于，当以后用户需要更复杂的切歌方式，只需改变播放器的实现即可；当存在更好的音乐编解码，只需要改变歌曲的播放操作即可。我们成功地将切换和播放解耦。

通常，接口的函数定义可以遵从如下模式：函数参数包括输入参数和输出参数，返回值为错误代码，或者是标示操作是否成功的布尔变量。函数需要对参数的有效性进行验证。输出参数通常传入地址，函数内可以将结果直接写到地址中。比如说，创建一个播放列表可以如下定义：

```
ErrorResult createPlaylist(Vector< Song *> inSongsArray, Playlist
*outPlaylist)
```

当 ErrorResult 为 0 时表示创建成功，outPlaylist 指向该播放列表。

经过上述例子，我们做出如下总结：

- 可以依照抽象，设计对象，设计接口的流程进行思考。
- 针对接口而不是实现来进行编程：不同对象之间保持接

口一致，调用接口时不需要基于接口内的实现方式。

在之后的部分，我们会给出一些常见的设计模式。但是，不要拘泥于具体的模式实现方式，我们更应该注重每种模式的意图是什么。

7.1.4 继承/组合/参数化类型

在面向对象中最常用的两种代码复用技术就是继承和组合。在设计对象的时候，“Is-A”表示一种继承关系。比如，班长“Is-A”学生，那么，学生就是基类，班长就是派生类。在确定了派生关系之后，我们需要分析什么是基类变量（Base Class Variable）什么是子类变量（Sub Class Variable），并由此确定基类和派生类之间的联系。而“Has-A”表示一种从属关系，这就是组合。比如，班长“Has-A”眼镜，那就可以解释为班长实例中拥有一个眼镜实例变量（Instance Variable）。在具体实现的时候，班长类中定义一个眼镜的基类指针。在生成班长实例的时候，同时生成一个眼镜实例，利用眼镜的基类指针指向这个实例。任何关于眼镜的操作函数都可以利用这个基类指针实现多态（polymorphism）。注意，多态是 OOP 相关的一个重要概念，也是面试常考的概念之一。关于多态的解释请见 7.3“工具箱”。

在通常情况下，我们更偏向于“Has-A”的设计模式。因为该模式减少了两个实例之间的相关性。对于继承的使用，通常情况下我们会定义一个虚基类，由此派生出多个不同的实例类。在业界的程序开发中，多重继承并不常见，Java 甚至不允许从多个父类同时继承，产生一个子类。

此外，我们还要提及参数化类型。参数化类型，或者说模版类也是一种有效的代码复用技术。在 C++的标准模版库中大量应用了这种方式。例如，在定义一个 List<String>的变量时，List 被另一个类型 String 所参数化。

设计模式着重于代码的复用，所以在选择复用技术上，有必要看看上述三种复用技术的优劣。

继承

- 通过继承方式，子类能够非常方便地改写父类方法，同时保留部分父类方法，可以说是能够最快速地达到代码复用的方法。
- 继承是在静态编译时候就定义了，所以无法在运行时刻改写父类方法。

- 因为子类没有改写父类方法的话，就相当于依赖了父类这个方法的实现细节，被认为破坏封装性。
- 并且如果父类接口定义需要更改时，子类也需要更改响应接口。

组合

- 对象组合通过获得其他对象引用而在运行时刻动态定义。
- 组合要求对象遵守彼此约定，进而要求更仔细地定义接口，而这些接口并不妨碍你将一个对象和另外一个对象一起使用。
- 对象只能够通过接口来访问，所以我们并没有破坏封装性。
- 而且只要抽象类型一致，对象是可以被替换的。
- 使用组合方式，我们可以将类层次限制在比较小的范围内，不容易产生类的爆炸。
- 相对于继承来说，组合可能需要编写更多的代码。

参数化类型

- 参数化类型方式是基于接口的编程，在一定程度上消除了类型给程序设计语言带来的限制。
- 相对于组合方式来说，缺少的是动态修改能力。
- 因为参数化类型本身就不是面向对象语言的一个特征，所以在面向对象的设计模式里面，没有一种模式是与参数化类型相关的。
- 实践上我们方面是可以使用参数化类型来编写某种模式的。

总结

- 对象组合技术允许你在运行时刻改变被组合的行为，但是它存在间接性，相对来说比较低效。
- 继承允许你提供操作的缺省实现，通过子类来重定义这些操作，但是不能够在运行时改变。
- 参数化允许你改变所使用的类型，同样不能够在运行时改变。

7.1.5 设计模式

所谓的设计模式是指人们在开发软件的过程中，对于一些普适需求而总结的设计模版。根据模式目的可以分为三类：

- 创建型(Creational)：创建型模式与对象的创建相关。
- 结构型（Structural)：结构型模式处理类或者是对象的组合。
- 行为型（Behavioral)：行为型模式对类或者是对象怎样交互和怎样分配职责进行描述。

下面我们对每种类型进行介绍。具体的模式请见7.3“工具箱”。值得提醒的是，在面试或工作中不可盲目相信设计模式。设计模式更多地只是提供一些思路，能够直接套用设计模式的情况并不多，更多的时候是对现成设计模式的改进和组合。所以对于设计模式的学习更多应该着眼于模式的意图，而不是模式的具体实现方法。

1．创建型

一个类的创建型模式使用继承改变被实例化的类，而一个对象的创建型模式将实例化委托给另外一个对象。 在这些模式中，有两种不断出现的主旋律：

- 将该系统使用哪些具体的类封装起来。
- 隐藏的实例是如何被创建和存储的。

总而言之，效果就是用户创建对象的结果是得到一个基类指针，用户通过基类指针调用继承类的方法。用户不需要知道在使用哪些继承类。

单例模式

意图：单例模式（Singleton Pattern）是一种常见的设计模式。其目的在于保证一个类仅仅有一个实例并且提供一个访问它的全局访问点。

这个模式主要的对比对象就是全局变量。相对于全局变量，单例有下面这些好处：

- 全局变量不能够保证只有一个实例。
- 某些情况下面，我们需要稍微计算才能够初始化这个单例。全局变量也行但是不自然。
- C++下面没有保证全局变量的初始化顺序。

比如，在我们之前说的音乐播放器设计中，我们引入了歌曲管理

器实现数据的存储。歌曲管理器在整个程序中应当实例化一次，其他所有关于数据的操作都应该在这个实例上进行。所以，歌曲管理器应该应用单例模式。实现单例模式的关键在于利用静态变量（Static Variable），通过判断静态变量是否已经初始化来确定该类是否已经实例化。此外，还需要把构造函数设为私有函数，通过公共接口 getSharedInstance 进行调用。我们举例如下：

```
// Example for singleton pattern
// class definition
class MySingleton {
private:
// Private Constructor
    MySingleton();
// Stop the compiler generating methods of copy the object
    MySingleton(const MySingleton &copy);    // Not Implemented
    MySingleton &operator=(const MySingleton &copy);    // Not Implemented
    static MySingleton *m_pInstance;
public:
    static MySingleton *getSharedInstance() {
        if (!m_pInstance) {
            m_pInstance = new MySingleton;
        }
        return m_pInstance;
    }
};
// in the source file
MySingleton *MySingleton::m_pInstance = NULL;
```

注意，本例中的实现方式针对非多线程的情况。如果有多个线程想要同时调用 getSharedInstance 函数，则需要用 mutex 保护下列代码：

```
        pthread_mutex_lock(&mutex);
        if (!m_pInstance) {
            m_pInstance = new MySingleton;
        }
        pthread_mutex_unlock(&mutex);
```

工厂模式

意图：抽象类需要创建一个对象时，让子类决定实例化哪一个类。

所谓的工厂模式（Factory Pattern），就是指定义一个创建对象的接口，但让实现这个接口的类来决定实例化哪个类。通常，接口提供传入参数，用以决定实例化什么类。工厂模式常见于工具包和框架中，当需要生成一系列类似的子类时，可以考虑使用工厂模式。举例

如下：

```
// class for factory pattern
enum ImageType{
    GIF,
    JPEG
};

class ImageReader {
    // implementation for image reader base class
};

class GIFReader : public ImageReader {
    // implementation for GIF reader derived class
};

class JPEGReader : public ImageReader {
    // implementation for JPEG reader derived class
};

class ImageReaderFactory {
public:
    static ImageReader *imageReaderFactoryMethod(ImageType imageType) {
        ImageReader *product = NULL;
        switch (imageType) {
            case GIF:
                product = new GIFReader();
            case JPEG:
                product = new JPEGReader();
                //...
        }
        return product;
    }
};
```

2. 结构型

类的结构型模式采用继承机制来组合接口。对象的结构型模式不是对接口进行组合，而是描述如何对一些对象进行组合，从而实现新功能。

适配器

意图：适配器（Adapter）将一个类的接口转化为客户希望的另外一个接口。

假设 A 实现了 Foo()接口，但是 B 希望 A 同样实现一个 Bar()接

口，事实上 Foo()基本实现了 Bar()接口功能。Adapter 模式就是设计一个新类 C，C 提供 Bar()接口，但实现的方式是内部调用 A 的 Foo()。

在实现层面上可以通过继承和组合两种方式达到目的：C 可以继承 A，或者 C 把 A 作为自己的成员变量。两者孰优孰劣需要视情况而定。

3．行为型

行为型模式涉及算法和对象之间职责的分配。行为型模式不仅描述对象或者类的功能行为，还描述它们之间的通信模式。 这些模式刻画了在运行时难以追踪的控制流，它们将你的注意从控制流转移到对象之间的联系上来。

观察者

意图：观察者模式（observer）定义对象之间的依赖关系，当一个对象状态发生改变，所有依赖这个对象的对象都会被通知并且进行更新。

被观察的对象需要能够动态地增删观察者对象，这就要求观察者提供一个公共接口，如 Update()。然后每个观察者实例注册到被观察对象里面去，在被观察对象状态更新时候能够遍历所有注册观察者并且调用 Update()。

至于观察者和被观察之间是采用 push 还是 pull 模式完全取决于应用。对于观察这件事情来说的话，我们还可以引入方面（Aspect）这样一个概念，在注册观察者的时候不仅仅只是一个观察者对象，还包括一个 Aspect 参数，可以以此告诉被观察者仅在发生某些变化时通过调用 Update()通知我。

状态

意图：状态模式（state）允许一个对象在其内部状态改变时改变它的行为。

这里状态模式意图是，对于实例 A，当 A 的状态改变时，将 A 可能改变的行为封装成为一个类 S（有多少种可能的状态就有多少个 S 的子类，比如 S1、S2、S3 等）。当 A 的状态转换时，在 A 内部切换 S 的实例。从 A 的用户角度来看，A 的接口不变，但 A 的行为因 A 的状态改变而改变，这是因为行为的具体实现由 S 完成。

7.2 模式识别

例题 1 Design a parking lot using object-oriented principles.

使用面向对象的原理，设计一个停车库。

解题分析：我们尝试使用抽象→设计对象→设计接口的流程。

抽象

在这一步，我们重现现实中车库的工作原理。通常，车库可能有不同的层，每层都有若干个车位，汽车可以停在车位上。车库的主要功能在于实现车辆入库和车辆出库，根据汽车在车库中停留的时间收费。

设计对象

经过抽象分析，我们发现了车位、层次、车库、汽车这些实体。对于每个实体，我们都应该构造一个类去描述它。我们考虑各个实体之间的从属，继承关系：很明显，车库拥有不同层次，每个层次拥有一些车位。因此，车库、层次、车位属于“Has-A”的关系。考虑我们可能希望车库是一个全局都可以访问的变量，而且程序中应该只有一个车库实例，所以我们可以利用单例模式，把车库作为一个单例。进一步考虑汽车实体，现实中，有各种类型的汽车，不同类型的车辆对车位的要求也不一样。然而，汽车具有一些共同属性，比如车长、车宽等，特别地，对于本例，每辆车都需要记录停车的状态。所以，我们可以考虑从一个汽车基类派生出不同类型的汽车派生类。

设计接口

车库需要与用户进行交互，因此应该提供车辆入库和车辆出库的接口。车辆入库时，需要从最底层依次向上寻找可用的车位，因此，寻找车位应该一个是由层次提供的接口，返回一个车位。车库把这个车位提供给一个汽车实例，并且标记车位不可用。当车辆出库时，我们需要汽车提供它停车位置的信息（因此，车辆需要提供接口返回它停车的位置），车库需要计算停车费，并且标记车位可用。

参考解答：

```
// define some constants
enum ErrorCode {
    NO_ERROR,
    ERROR
};

enum SpotType {
    COMPACT,
    SUV,
    RESERVED
};
#define NO_PARKING (-1)

class Spot {
public:
    bool     available;
    SpotType type;
};

class Vehicle {
private:
    int     length;
    int     width;
    bool    parked;
    Spot    *spot;
public:
    // omit some setters / getters
    // virtual function here because subclasses will have different behavior
    virtual SpotType getRequiredSpotType() = 0;
    // no need for virtual functions here because subclasses will have the s
ame "behavior"
    bool isParked();
    void parkVehicle(Spot *s);  // park at spot S;
    Spot *removeVehicle();      // move the vehicle away, return parked spot
};

//every type of vehicle has default value of length and width;
class Motor:public Vehicle{};
class Car:public Vehicle{};
class SUV:public Vehicle{};
```

```
class Level {
private:
    vector<Spot> spots;
public:
    Spot *findASpot(Vehicle *v);    // find an available spot for a vehicle
                                                    // return NULL if all spots
 are taken
};

class ParkingLot {
private:
    vector<Level> levels;
    static ParkingLot *pInstance;
    unordered_map<Vehicle *, time_t> parkingInfo;

    ParkingLot();
    // Stop the compiler generating methods of copy the object
    ParkingLot(const ParkingLot &copy);    // Not Implemented
    ParkingLot& operator = (const ParkingLot & copy);    // Not Implemented

    time_t getCurrentTime();
    double calculateFee(Vehicle *v);

public:
    static ParkingLot *getInstance();
    // NOTE: vehicleEnter and leave is not thread safe!
    ErrorCode vehicleEnter(Vehicle *v);
    ErrorCode vehicleLeave(Vehicle *v, double *fee);
};
ErrorCode ParkingLot::vehicleEnter(Vehicle *v) {
    Spot *spot = NULL;
    for (int i = 0; i < levels.size();i++) {
        spot = levels[i].findASpot(v);
        if (spot)
            break;
    }
    if (!spot) {
        return ERROR;
    }
    v->parkVehicle(spot);
    spot->available = false;
    parkingInfo[v] = getCurrentTime();
    return NO_ERROR;
}
```

```
ErrorCode ParkingLot::vehicleLeave(Vehicle *v, double *fee) {
    *fee = 0;
    if (!v->isParked()) {
        return ERROR;
    }
    Spot *spot = v->removeVehicle();
    spot->available = true;
    *fee = calculateFee(v);
    parkingInfo.erase(v);
    return NO_ERROR;
}
```

进一步讨论：

我们提供的代码并不是线程安全的，当多个线程同时调用 enter 和 leave 的时候，可能造成停车状态的不一致，需要通过加锁解决。其次，查找车位的时候我们实现了最简单的线性查找。事实上，我们可以用一个队列记录可用的车位，每次只需要弹出一个即可。那对于不同的车位类型怎么处理？我们可以用多个队列记录可用的车位，每个队列对应一个车型。

例题 2 Design an elevator bank for a building, with multiple elevators

为一栋建筑设计一个电梯间，带有多部电梯。

解题分析：我们尝试使用抽象→设计对象→设计接口的流程。

抽象

在这一步，我们重现现实中电梯间的工作原理：电梯间拥有多部电梯，当有用户需要乘坐电梯时，电梯间分配一个最优的电梯去用户所需的楼层。当电梯需要维修时，电梯间可以停止某部电梯。

设计对象

经过抽象分析，我们发现了电梯间和电梯这两个实体。对于每个实体，我们都应该构造一个类去描述它。我们考虑各个实体之间的从属关系：很明显，电梯间拥有一些电梯。因此，电梯间和电梯属于“Has-A”的关系。对于一部电梯，它应该具有当前的运行状态，包括停止在某一层、向上运行和向下运行。电梯间可以根据当前各个电梯的位置和运行情况，选择最优的电梯去满足用户需求，可以采用下述选择逻辑，计算一个“距离分”，距离分越低，则优先级越高：

（1）当电梯处于停止状态，距离分等于当前楼层和需求楼层的差。

（2）当电梯处于上升状态，如果需求楼层高于当前楼层，则距离分等于当前楼层和需求楼层的差，否则不考虑当前电梯。

（3）当电梯处于下降状态，如果需求楼层低于当前楼层，则距离分等于当前楼层和需求楼层的差，否则不考虑当前电梯。

（4）算法开始时先随机选择一部电梯，当存在更好的电梯选择时，替代当前的候选电梯，这样确保至少会有一部电梯响应（在少数情况下这样做不一定是最优的，可以设想更好的算法解决这个问题）。

设计接口

用户所有的请求都应该与电梯间进行交互。电梯间再选择最恰当的电梯，去满足用户的请求。电梯也需要提供接口，以供电梯间设置一个请求。同时，电梯间依赖于电梯的当前位置和运行状况实现选择算法，所以电梯应该提供 getter 函数。此外，电梯间还应该提供启用/停止某台电梯的接口，以供管理员进行维护。至于电梯间选择电梯的算法函数等等，用户不需要知道这些信息，故应该作为私有函数。

多线程

电梯需要在后台不停地自动运行，同时还需要响应电梯间设置的用户请求。因此，我们需要考虑多线程。我们可以用主线程响应用户请求，同时建立另一个线程模拟电梯的运行过程。注意，由于使用了多线程，所有的线程共享变量都需要用锁保护起来，避免产生竞争（Racing Condition）。同时，考虑到打印调试信息也可能产生竞争，所以应该用一个特定的打印线程处理所有输出。在参考解答中，我们给出在 Mac OS 下利用 dispatch_queue 的实现方式。

参考解答：

```
// define some constants
dispatch_queue_t printingQueue;
enum ElevatorState {
    UP,           // moving up
    DOWN,    // moving down
    STAND    // stay at a level, waiting for request
};

class Elevator {
private:
    ElevatorState state;
    int id;
    int currentLevel;
    int maxLevel;
    int minLevel;
```

```
    vector<int> requests;    // floor requests from low level to high level
    pthread_mutex_t lock;
    pthread_t runThread;    // runLoop is called on this thread
    void mutexLock();       // utility methods to lock and unlock
    void mutexUnlock();
    bool needStop();        // check if need to stop on current floor
    void updateState();    // switch to STAND if no request pending.
                                    // go DOWN if currently going UP, and no
further higher level requests
                                    // go UP if currently going DOWN, and no
further lower level requests

    void move();    // move one floor up/down. Open door if needed. Update st
ate
    void openDoor();    // open door and close door, delete request from arra
y
    void runLoop();    // thread safe operation, main logic for elevator oper
ation (move, open door)
    static void *elevatorProc(void *parameter);
public:
    Elevator(int id, int minLevel, int maxLevel);
    ~Elevator();
    ErrorCode addRequest(int newRequest);    // thread safe operation, calle
r adds a new stop request
    void startOperation();    // thread safe operation, start elevatorProc t
o activate elevator
    void stopOperation();    // thread safe operation, stop elevatorProc to
deactivate elevator
    ElevatorState getState();    // thread safe operation, get elevator stat
e
    int getCurrentLevel();    // thread safe operation, get elevator floor
#pragma -mark Test Methods
    void printRequests() {
        // caller should have the data lock
        dispatch_async(printingQueue, ^{
            cout << "Elevator " << id << " current requests: ";
            for (int i = 0 ; i < requests.size(); i++) {
                cout << requests[i] << ' ';
            }
            cout << endl;
        });
    }
};
```

```
#pragma -mark Private Methods

bool Elevator::needStop() {
    for (int i = 0; i < requests.size(); i++) {
        if (requests[i] == currentLevel) {
            return true;
        }
    }
    return false;
}

void Elevator::updateState() {
    switch (state) {
        case UP:
            if (requests.size() == 0) {
                state = STAND;
            } else {
                if (requests.back() < currentLevel) {
                    state = DOWN;
                }
            }
            break;
        case DOWN:
            if (requests.size() == 0) {
                state = STAND;
            } else {
                if (requests.front() > currentLevel) {
                    state = UP;
                }
            }
            break;
        default:
            break;
    }
}

void Elevator::move() {
    switch (state) {
        case UP:
            if (needStop()) {
                openDoor();
                // flip state
                updateState();
            } else {
```

```
                dispatch_async(printingQueue, ^{
                    cout << "Elevator " << id << " move from " << currentLevel
                    << " to " << currentLevel + 1 << endl;
                });
                currentLevel++;
            }
            break;
        case DOWN:
            if (needStop()) {
                openDoor();
                // flip state
                updateState();
            } else {
                dispatch_async(printingQueue, ^{
                    cout << "Elevator " << id << " move from " << currentLevel
                    << " to " << currentLevel - 1 << endl;
                });
                currentLevel--;
            }
            break;
        default:
            break;
    }
}

void Elevator::openDoor() {
    dispatch_async(printingQueue, ^{
        cout << "Elevator " << id << " arriving at " << currentLevel << endl;
        cout << "Elevator " << id << " door open" << endl;
    });
    for (vector<int>::iterator it = requests.begin(); it != requests.end();
 ) {
        if (*it == currentLevel) {
            it = requests.erase(it);
        } else {
            ++it;
        }
    }
    dispatch_async(printingQueue, ^{
        cout << "Elevator " << id << " door close" << endl;
    });
    printRequests();
}
```

```
void Elevator::runLoop() {
    mutexLock();
    switch (state) {
        case UP:
        case DOWN:
            move();
            break;
        default:
            break;
    }
    mutexUnlock();
}

void *Elevator::elevatorProc(void *parameter) {
    Elevator *elevator = (Elevator *)parameter;
    while (1) {
        elevator->runLoop();
        usleep(4000000);
    }
    return NULL;
}

#pragma -mark Public Methods

Elevator::Elevator(int inID, int inMinLevel, int inMaxLevel) {
    state = STAND;
    id = inID;
    maxLevel = inMaxLevel;
    minLevel = inMinLevel;
    currentLevel = inMinLevel;
    pthread_mutex_init(&lock, NULL);
}

Elevator::~Elevator() {
    stopOperation();
    pthread_mutex_destroy(&lock);
}

void Elevator::mutexLock() {
    pthread_mutex_lock(&lock);
}

void Elevator::mutexUnlock() {
    pthread_mutex_unlock(&lock);
```

```
}

void Elevator::startOperation() {
    mutexLock();
    pthread_create(&runThread, NULL, elevatorProc, this);
    mutexUnlock();
}

void Elevator::stopOperation() {
    mutexLock();
    if (runThread) {
        pthread_cancel(runThread);
    }
    mutexUnlock();
}

ErrorCode Elevator::addRequest(int newRequest) {
    mutexLock();
    if (newRequest > maxLevel || newRequest < minLevel) {
        cerr << "Invalid request " << newRequest << endl;
        return ERROR;
    }
    if (requests.size() == 0) {
        requests.push_back(newRequest);
        if (newRequest > currentLevel) {
            dispatch_async(printingQueue, ^{
                cout << "Elevator " << id << " moving up" << endl;
            });
            state = UP;
        } else if (newRequest == currentLevel) {
            state = STAND;
            openDoor();
        } else {
            dispatch_async(printingQueue, ^{
                cout << "Elevator " << id << " moving down" << endl;
            });
            state = DOWN;
        }
        goto Done;
    }

    for (vector<int>::iterator i = requests.begin(); i != requests.end(); i+
+) {
        if (newRequest == *i) {
```

```
            goto Done;
        }
        if(*i > newRequest) {
            requests.insert(i, newRequest);
            goto Done;
        }
    }
    requests.push_back(newRequest);
Done:
    printRequests();
    mutexUnlock();
    return NO_ERROR;
}

ElevatorState Elevator::getState() {
    mutexLock();
    ElevatorState currentState = state;
    mutexUnlock();
    return currentState;
}

int Elevator::getCurrentLevel() {
    mutexLock();
    int level = currentLevel;
    mutexUnlock();
    return level;
}

class ElevatorBank {
private:
    int numberOfElevators;
    vector<Elevator> elevators;
    int calculateScore(int index, int level);
    bool isBetterNewCandidate(int currentCandidateIndex, int newCandidateInd
ex, int level);
    int selectAnElevator(int level);
public:
    ElevatorBank(int numberOfElevators, int minFloor, int maxFloor);
    ~ElevatorBank();
    ErrorCode startAnElevator(int index);   // index start from 0
    ErrorCode stopAnElevator(int index);    // index start from 0
    ErrorCode setARequest(int level);
};
```

```
#pragma -mark Private Methods

int ElevatorBank::calculateScore(int index, int level) {
    if (elevators[index].getState() == STAND) {
        return abs(elevators[index].getCurrentLevel() - level);
    }
    if (elevators[index].getCurrentLevel() > level
        && elevators[index].getState() == DOWN) {
        return elevators[index].getCurrentLevel() - level;
    }
    if (elevators[index].getCurrentLevel() < level
        && elevators[index].getState() == UP) {
        return level - elevators[index].getCurrentLevel();
    }
    return INT_MAX;
}

bool ElevatorBank::isBetterNewCandidate(int currentCandidateIndex, int newC
andidateIndex, int level) {
    return calculateScore(currentCandidateIndex, level) > calculateScore(new
CandidateIndex, level);
}
int ElevatorBank::selectAnElevator(int level) {
    int selectedIndex = 0;
    for (int i = 1; i < numberOfElevators; i++) {
        if (isBetterNewCandidate(selectedIndex, i, level)) {
            selectedIndex = i;
        }
    }
    return selectedIndex;
}

#pragma -mark Public Methods

ElevatorBank::ElevatorBank(int inNumberOfElevators, int minFloor, int maxFl
oor) {
    numberOfElevators = inNumberOfElevators;
    for (int i = 0; i < inNumberOfElevators; i++) {
        elevators.push_back(Elevator(i, minFloor, maxFloor));
    }
}

ElevatorBank::~ElevatorBank() {
    elevators.clear();
```

```
}

ErrorCode ElevatorBank::startAnElevator(int index) {
    if (index > numberOfElevators || index < 0) {
        cerr << "Index " << index << " invalid!" << endl;
        return ERROR;
    }
    elevators[index].startOperation();
    return NO_ERROR;
}

ErrorCode ElevatorBank::stopAnElevator(int index) {
    if (index > numberOfElevators || index < 0) {
        cerr << "Index " << index << " invalid!" << endl;
        return ERROR;
    }
    elevators[index].stopOperation();
    return NO_ERROR;
}

ErrorCode ElevatorBank::setARequest(int level) {
    int index = selectAnElevator(level);
    return elevators[index].addRequest(level);
}

int main() {
    printingQueue = dispatch_queue_create("elevatorBank.printingQueue", NUL
L);
    ElevatorBank elevatorBank = ElevatorBank(2, 1, 10);
    for (int i = 0; i < 2; i++) {
        elevatorBank.startAnElevator(i);
    }
    int request;
    while (cin >> request) {
        elevatorBank.setARequest(request);
    };
    return 0;
}
```

7.3 工具箱

1. 有限状态机
   http://en.wikipedia.org/wiki/Finite-state_machine

2．多态

在 C++中，最常见的多态指的是用基类指针指向一个派生类的实例，当用该指针调用一个基类中的虚函数时，实际调用的是派生类的函数实现，而不是基类函数。如果该指针指向另一个派生类实例，则调用另一个派生类的函数实现。因此，比如工厂模式返回一个实例，上层函数不需要知道实例来自哪个派生类，只需要用一个基类指针指向它，就可以直接获得需要的行为。从编译的角度来看，函数的调用地址并不是在编译阶段静态决定的，而是在运行阶段，动态地决定函数的调用地址。

多态是通过虚函数表实现的。当基类中用 virtual 关键字定义函数时，系统自动分配一个指针，指向该类的虚函数表。虚函数表中存储的是函数指针。在生成派生类的时候，会将派生类中对应的函数的地址写到虚函数表。之后，当利用基类指针调用函数时，先通过虚函数表指针找到对应的虚函数表，再通过表内存储的函数指针调用对应函数。由于函数指针指向派生类的实现，因此函数行为自然也就是派生类中定义的行为了。

3．创建型设计模式补充

Builder

意图：将一个复杂对象构建过程和元素表示分离。

假设我们需要创建一个复杂对象，而这个复杂对象是由很多元素构成的。这些元素的组合逻辑可能非常复杂，但是逻辑组合和创建这些元素是无关的，独立于这些元素本身的。

那么我们可以将元素的组合逻辑和元素构建进行分离，元素构建我们单独放在 Builder 这样一个类里面，而元素的组合逻辑通过 Director 来指导，Director 内部包含 Builder 对象。创建对象是通过 Director 来负责组合逻辑部分的，Director 内部调用 Builder 来创建元素并且组装起来。最终通过 Builder 的 GetResult 来获得最终复杂对象。

4．结构型设计模式补充

Bridge

意图：将抽象部分和具体实现相分离，使得它们之间可以独立变化。

一个很简单的例子就是类 Shape，有个方法 Draw[抽象]和 DrawLine[具体]和 DrawText[具体]，而 Square 和 SquareText 继承

于 Shape 实现 Draw()这个方法，Square 调用 DrawLine()，而 SquareText 调用 DrawLine()+DrawText()。而且假设 DrawLine 和 DrawText 分别有 LinuxDrawLine、LinuxDrawText 和 Win32DrawLine 和 Win32DrawText。如果我们简单地使用子类来实现的话，比如构造 LinuxSquare、LinuxSquareText、Win32Square 和 Win32SquareText，那么很快就会类爆炸。

事实上我们没有必要在 Shape 这个类层面跟进变化，即通过继承 Shape 类实现跨平台，而只需要在实现底层跟进变化。为此我们定义一套接口，如例子中的 DrawLine 和 DrawText，然后在 Linux 和 Win32 下实现一个这样接口实例（如称为跨平台 GDI），最终 Shape 内部持有这个 GDI 对象，Shape 的 DrawLine 和 DrawText 只是调用 GDI 的接口而已。这样，我们把 Shape 及其子类的 DrawLine 和 DrawText 功能 Bridge 到 GDI，GDI 可以通过工厂模式在不同平台下实现不同的实例。

例子中 Shape 成为了完全抽象的部分，具体实现完全交给 GDI 类，若以后需要增加更多的平台支持，开发者也不需要添加更多的 Shape 子类，只需要扩展 GDI 即可。总之，当抽象部分和具体实现部分需要独立开来的时候，就可以使用 Bridge 模式。

Composite

意图：将对象组合成为树形以表示层级结构，对于叶子和非叶子节点对象使用需要有一致性。

Composite 模式强调在这种层级结构下，叶子和非叶子节点需要一致对待，所以关键是需要定义一个抽象类，作为叶节点的子节点。然后对于叶子节点操作没有特殊之处，而对于非叶子节点操作不仅仅需要操作自身，还要操作所管理的子节点。 至于遍历子节点和处理顺序是由应用决定的，在 Composite 模式里面并不做具体规定。

Decorator

意图：动态地给对象添加一些额外职责，通过组合而非继承方式完成。

给对象添加一些额外职责，例如增加新的方法，很容易会考虑使用子类方式来实现。使用子类方式实现很快但是却不通用，考虑一个抽象类 X，子类有 SubX1、SubX2 等。现在需要为 X 提供一个附加方法 echo，如果用继承的方式添加，那么需要为每个子类都实现 echo 方法，并且代码往往是重复的。我们可以考虑 Decorator 模式，定义一个新类，使其持有指向 X 基类的指针，并且新类只需要单独实现

echo 方法，而其他方法直接利用 X 基类指针通过多态调用即可。

值得注意的是，装饰出来的对象必须包含被装饰对象的所有接口。所以很明显这里存在一个问题，那就是 X 一定不能够有过多的方法，不然 Echo 类里面需要把 X 方法全部转发一次（理论上说 Echo 类可以仅转发 X 的部分方法，但 Decorator 默认需要转发被装饰类的全部方法）。

Facade

意图：为子系统的一组接口提供一个一致的界面。

编译器是一个非常好的例子。对于编译器来说，有非常多的子系统包括词法语法解析、语义检查、中间代码生成、代码优化，以及代码生成这些逻辑部件。但是对于大多数用户来说，不关心这些子系统，而只是关心编译这一个过程。

所以我们可以提供 Compiler 的类，里面只有很简单的方法比如 Compile()，让用户直接使用 Compile()这个接口。 一方面用户使用起来简单，另外一方面子系统和用户界面耦合性也降低了。

Facade 模式对于大部分用户都是满足需求的。对于少部分不能够满足需求的用户，可以让他们绕过 Facade 模式提供的界面，直接控制子系统即可。例如，GCC 提供了很多特殊优化选项来让高级用户来指定，而不是仅仅指定-O2 这样的选项。

Proxy

意图：为其他对象提供一种代理以控制对这个对象的访问。

通常使用 Proxy 模式是想针对原本要访问的对象做一些手脚，以达到一定的目的，包括访问权限设置，访问速度优化，或者是加入一些自己特有的逻辑。至于实现方式上，不管是继承还是组合都行，可能代价稍微有些不同，视情况而定。但是偏向组合方式，因为对于 Proxy 而言，完全可以定义一套新的访问接口。

Adapter、Decorator 以及 Proxy 之间比较相近，虽然说意图上差别很大，但是在实践中，三者都是通过引用对象增加一个新类来完成的，但是这个新类在生成接口方面有点差别：

- Adapter 模式的接口一定要和对接的接口相同。
- Decorator 模式的接口一定要包含原有接口，通常来说还要添加新接口。
- Proxy 模式完全可以重新定义一套新的接口。

5．行为型设计模式补充

Chain of Responsibility

意图：将对象连成一条链并沿着链传递某个请求，直到有某个对象处理它为止。

大部分情况下连接起来的对象本身就存在一定的层次结构关系，少数情况下这些连接起来的对象是内部构造的。 职责链通常与Composite模式一起使用,一个构件的父构件可以作为它的后继节点。许多类库使用职责链模式来处理事件，比如在UI部分，View本来就是相互嵌套的，一个View对象可能存在Parent View对象。如果某个UI不能够处理事件的话，那么完全可以交给Parent View来完成事件处理，以此类推。

Command

意图：将一个请求封装成为一个对象。

Command模式可以说是回调机制（Callback）的一个面向对象的替代品。对于回调函数来说需要传递一个上下文参数（context），同时内部附带一些逻辑。将上下文参数以及逻辑包装起来，就是一个Command对象。 Command对象接口可以非常简单，如只有Execute/UnExecute，但是使用Command对象来管理请求之后，就可以非常方便地实现命令的复用、排队、重做、撤销、事务等。

Iterator

意图：提供一种方法顺序访问一个聚合对象中的各个元素，但是又不需要暴露该对象内部表示。

将遍历机制与聚合对象表示分离,使得我们可以定义不同的迭代器来实现不同的迭代策略，而无需在聚合对象接口上面列举他们。一个健壮的迭代器,应该保证在聚合对象上面插入和删除操作不会干扰遍历，同时不需要复制这个聚合对象。一种实现方式就是在聚合对象上面注册某个迭代器，一旦聚合对象发生改变的话，需要调整迭代器内部的状态。

Template Method

意图:定义一个操作里面算法的骨架,而将一些步骤延迟到子类。

假设父类A里面有抽象方法Step1()、Step2(),默认方法Step3()。并且A提供一个操作X(),分别依次使用Step1()、Step2()、Step3()。对于A的子类，通过实现自己的Step1()、Step2()（选择性地实现Step3()），提供属于子类的X具体操作。 这里操作X()就是算法的骨架，子类需要复写其中部分step，但不改变X的执行流程。

很重要的一点是模板方法必须指明哪些操作是钩子操作（可以被重定义的，比如 Step3），以及哪些操作是抽象操作（必须被重定义，比如 Step1 和 Step2）。要有效地重用一个抽象类，子类编写者必须明确了解哪些操作设计为有待重定义的。

第 8 章 递归和动态规划

8.1 知识要点

递归（Recursion）和动态规划（Dynamic Programming，DP）是算法类问题中的难点所在。算法的核心在于找到状态转移方程，即如何通过子问题解决原问题。在本节，我们先介绍递规和 DP 的普适特性；再通过 8.2“模式识别”，从题目的关键字出发，判断什么样的题目适合用递归和 DP，并且总结算法模版。

8.1.1 构建从子问题到最终目标的方法

递归和动态规划能解决的问题都有一个特性：原问题（problem）可以分解成若干个子问题（sub-problem），只有先解决了子问题才能进一步解决原问题。子问题的解决方式形式上与原问题一致。从题目描述来看，可以提示我们尝试用递归、DP 解决的关键词有：compute nth element (value, sum, max, etc.)（求第 n 个元素、值、和、最大值等等），return all the paths（返回所有的路径），return all the combinations（返回所有的组合），return all the solutions（返回所有的解）…

既然动态规划与递归都能解决相同类型的问题，那么 DP 和递归有什么不同？最大的区别在于，DP 存储子问题的结果，当子问题已经被计算过，直接返回结果。因此，当需要重复计算子问题时，DP 的时间效率高很多，但需要额外的空间。

特别地，具有聚合属性的问题（Aggregate），例如在所有组合中寻找符合特定条件的特解（如二叉树求一条从根节点到叶节点和为定值的路径，或第 n 个元素），或最优解（包括最值），或总和，或数量的问题（其实看一下 SQL 里的聚合函数（Aggregate Function）就明白了）。因为这些问题只需要一个聚合的或者特殊的结果，而不是所

有满足条件的集合，所以它们具有很强的收敛性质。这类问题往往也可以用 DP 来解决。

本章将问题处理的每一个最小的元素/步骤，称为节点，就好比一维/二维/三维数组中的一个 element，或者每一次递归中独立解决的那个元操作。我们把节点空间“两端收敛”的问题，归结为收敛结构；将节点空间“发散”的问题，归结为发散结构。形象地说，收敛问题是由若干个子问题共同决定当前状态，即状态的总数逐渐“收敛”，例如斐波那契数列问题（前两个节点决定当前节点）。发散问题是当前状态会衍生出多个下一状态，例如遍历已知根节点的二叉树（下一层的状态以指数形式增加）。抽象地说，能够在多项式时间内解决的问题，是收敛问题（P 类问题），不能在多项式内解决的问题（如阶乘级或指数级），是发散问题（NP 类问题）。这里定义“收敛”和“发散”是为了方便本章节描述和区分这两类问题，并非是公认的准则。

8.1.2 递归的空间与时间成本

对系统层面上说，操作系统是利用函数栈来实现递归，每次操作可视为栈里的一个对象。递归的时间成本随递归深度 n（单条路径中递归调用的次数）成指数增长；空间复杂度为 O(n)。

8.1.3 自底向上与自顶向下

从子问题解决原问题，无非是两种方法，自底向上（Bottom-Up）与自顶向下（Top-Down），形式上前者对应迭代，利用循环将结果存在数组里，从数组起始位置向后计算；后者对应递归，即利用函数调用自身实现，如果不存储上一个状态的解，则为递归，否则就是 DP。举个斐波那契数列（0,1,1,2,3,5…）的例子：

（1）自底向上

```
int array[n] = {0};
array[1] = 1;
for (int i = 2; i < n; i++)
    array[i] = array[i-1] + array[i-2];
```

这里，为了说明方法，采用数组存储结果，空间复杂度为 O(n)。事实上，额外空间可以进一步缩小到 O(1)：利用几个变量记录之前的状态即可。由于我们记录了子问题的解，故给出的方法就是 DP。事实

上，自底向上的方式往往都是通过动态规划实现。

（2）自顶向下

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

为计算 Fibonacci 的第 n 个元素，我们先自顶向下地计算子问题：第 n-1 个元素和第 n-2 个元素。由于没有存储子问题的运算结果，我们给出的方法是递归。然而，Fibonacci(n-1)与 Fibonacci(n-2)包含很多重复的子问题，所以 DP 效率更高。如果用一个全局数组，将子问题的解存储到数组的对应位置，在重复计算的时候直接读取计算结果，那么就是 DP 的解法。

我们再次强调：动态规划的核心在于，如果在一个问题的解决方案中，子问题被重复计算，那么就可以利用记录中间结果，达到用空间换取时间的目的。

以下如不额外说明，动态规划特指迭代形式的自底向上的动态规划，并将自顶向下、递归形式、在递归过程中用哈希表记录中间计算结果的 DP，称作 Memorization。

Memorization 的一般形式是：建立以 input 为键，以 output 为 value 的哈希表：

```
T func(N node, HashTable<N, T>& cache) {
    If (cache.contains(node)) {
        return cache.get(node);
    }
    …
    T sub_res = func(next_node, cache);
    …
    T res = G( sub_res … );  //当前子问题的解，依赖于更小的子问题(s)
    cache.set(node, res);
    return res;
}
```

从解决问题的角度来说，用自底向上的 DP，固然通常可以节省递归本身的空间开销，但有很多缺点和局限：较难理解，边界条件较难处理，只适用于问题的节点空间是离散的整数空间，必须一步步邻接、连续地计算（无论是不是每一个节点的结果都会被用到）。而

Memorization 则灵活得多：可以从递归形式轻易修改得到，也更符合普遍的思维过程，并且没有上面说的这些局限，子问题只有在被需要的时候才会被计算。尤其是在某些情况下，不仅需要 aggregate 的结果，还需要获得 achieve 这个结果的路径，这时候就算用自底向上的 DP，也需要记录 prev 节点，最后需要递归回溯得到路径，那么节省递归空间开销的优势，也荡然无存了。

尽管有上述局限，自底向上的 DP，仍然可以作为很好的思维和编程训练，熟练之后写法也很简洁精妙，本章也会举很多例子，但完全不用过于痴迷这类技巧性较强的解法，否则就是买椟还珠了。毕竟，从面试的角度讲，在这么短时间内，期望面试者用这类 DP 解法写出一个完整程序，并不是普遍能做到的。

8.1.4 算法策略

以下回顾一些利用到 DP 思想的经典算法策略：

分而治之（Divide and Conquer）：这里只谈狭义的 D&C，即将问题分成几个部分，每一部分相互独立，互不重叠，假定每个部分都可以得到解决来进行递归调用，合并每一部分的结果。例如 Merge Sort Quick Sort（Merge Sort 的 divide 容易，但 Conquer/Merge 复杂，Quick Sort 的 divide 复杂，但 Conquer/Merge 容易）。

动态规划（Dynamic Programming）：尽可能不重复计算每个子问题，而是将计算结果存储下来，以确定后驱问题的解。与贪婪算法的区别是，会记录下所有可能通向全局最优解的局部解，以便在计算后驱问题时综合考虑多个前驱问题的解。

贪婪算法（Greedy Algorithm）：只做出当下最优的判断，并且以此为基础进行下一步计算。当前判断最优时，不考虑对全局/未来的影响，所以从全局来说并不能保证总是最优。贪婪算法每次更新当前的最优解。如 Dijkstra 算法就是贪婪算法的实例之一。

回溯（Backtracking）：一种暴力（穷举）的深度优先搜索法：搜索，直到节点空间的尽头，然后再返回到上次的节点，再往其他方向深度搜索。树或图的 DFS 是回溯的实例之一。

8.2 模式识别

8.2.1 用动态规划（自底向上）解决收敛结构问题

具有强收敛性/聚合属性的问题（如前所述，是指关于特解，或最值，或总和，或数量的问题），都可以用整数坐标映射所有节点，且当前节点的解只依赖于前驱节点（无论是顺序还是倒序）。那么，这类问题往往可以用 DP 解决。解决的关键是建立子问题的解之间的递推关系：

```
f(n) = G[f(n-1), f(n-2), … , f(1)] 或
f(i, j) = G[f(i-1, j-1), f(i, j -1), f(i-1,j)]
```

其中 G[ ]表示子问题到原问题的映射关系，例如对于斐波那契数列，有递推式：

f(n) = G[f(n-1), f(n-2)] = f(n-1) + f(n-2)解决这类问题的时候，可以把上述递推关系写在手边，这样做非常有利于理清算法实现的思路。实际实现算法时，往往以问题的一端为循环开端，另一端为循环终止条件，将当前节点的解（或往往是，以当前节点为末节点的问题的解，抑或是以当前两个坐标为输入的问题的解）用 DP 表（即数组）记录下来（如果当前节点只由之前紧接的若干个节点决定，那么用若干个变量也够了），数组的下标即为子问题的输入变量，也就是递推关系中的函数参数，只是把 f(i,j)表示成 array[i][j]而已。

如果问题除了要计算动归终点的数值以外，还需要记录具体的到达路径，则可记录每个节点的前驱节点（prev[n]）或前驱路径（vector<vector<int>> prev），然后用终点出发通过回溯处理成路径。这时候记录的前驱们都是经过了 DP 的剪枝，每一条路径都是符合条件的正确路径。

注意，如果出现类似于“所有解”，“所有路径”等关键词，则用自上而下方法更为直接。之后我们也会给出例子。

例题 1 Suppose we have a ladder which has n steps. Each time you can either climb 1 or 2 steps. Please write a function to calculate how many distinct ways that can you climb to the top?

假设有一个楼梯共有 n 步。你每次可以爬 1 步或 2 步。请编写一个函数来计算，有多少种不同的方法可以爬到顶。

解题分析：*本问题描述了一个数量问题，属于前述的强收敛*（聚合）性问题，可以用 DP。*DP 的核心在于递推关系：*当前节点的值可以由前驱走一步到达，或者前前驱走两步到达，即 CountOfWays(n) = CountOfWays(n－1) + CountOfWays(n-2)；由于当前节点只与紧邻的两个节点决定，所以只需要 2 个临时变量来表示前驱节点的解即可，而不用 DP 表，因为更老的解我们不需要关心。在实现时，往往边界条件直接用 if…then return value 的形式，成为递归的出口。

参考解答：

```
int climbStairs(int n) {
    if(n <= 1 ) return 1;
    if(n == 2)  return 2;

    int p = 1, q = 2, curr;
    for(int i = 3; i <= n; ++i ){
        curr = p + q;
        p = q;
        q = curr;
    }
    return curr;
}
```

例题 2 Compute the nth prime 计算第 n 个素数。

解题分析：在数学理论中，存在埃拉托斯特尼筛法（sieve of Eratosthenes),用来解决素数判定问题。如果面试中碰到这个问题，面试官绝对不是希望获得那样的解答，因为这里考察的是编程技能，而不是高深的数学知识。

*第 n 个素数仍然是一个特解问题，我们需要考虑 DP。*事实上，素数的定义隐含了一个递推关系，即如果 n 是素数，那么 n 不能被 1～n-1 的所有素数整除（事实上，可以优化为 1～sqrt(n)），即当前节点与之前的所有节点有关:

```
Prime(n) = G ( Prime(n-1), Prime(n-2), Prime(n-3) … Prime(1));
Prime(1) = 2
```

G( )表示不能整除的关系。

参考解答：

```
int GetNthPrime( int n) {
    list<int> primes (1, 2);    // init list: length 1, value 2 (first prime)
    int number = 3;
    while(primes.size() != n) {
        bool isPrime = true;
        for(auto it = primes.begin(); it != primes.end() && (*it)*(*it) <= nu
mber; it++) {
            if(number % (*it) == 0)
                isPrime = false;
        }

        if (isPrime) {
            primes.push_back(number);
        }
        number += 2;    // even numbers greater than 2 are not prime
    }
    return *(primes.rbegin());
}
```

例题 3 Given a string and a dictionary of words, please write a function to determine if that string can be completely segmented into several words, where every word appears in the given dictionary.

给定一个字符串和包含一些单词的一个字典，编写一个函数来判断该字符串是否能够完全由几个单词组成，其中每个单词都存在于给定的字典之中。

解题分析：这个问题属于在所有组合中，*寻求一个特解以满足一定的条件。对于这样的判定性问题，它具有很强的聚合性，可以用 DP。*

DP 的关键在于寻找递推关系：考虑 string 的前 n 个字符，那么对于第 i 个字符(i [0, n))，如果[0,i）可以由一个或多个单词组成（前驱节点，可以直接读取 DP table 中计算过的结果），并且[i, n）是一个单词（这一段我们不能去借助局部解，因为目前为止，我们只有[0,1)，[0,2) … [0, n-1)的结果，需要利用字典判断]，那么，[0, n]的结果就是 true。

参考解答：

```
bool wordBreak(string s, unordered_set<string> &dict) {
    int begin = 0, end = 0;
    string word;
    bool words[s.size()+1] = {0};
    words[0] = true;
    for( int i = 1; i < s.size() + 1; i++) {
        words[i] = false;
    }
    for( end = 0; end < s.size(); end++) {
        for( begin = 0; begin <= end; begin++) {
            if( words[begin] && dict.find( s.substr(begin, end-begin+1) )
                != dict.end()) {
                words[end + 1] = true;
                break;
            }
        }
    }
    return words[s.size()];
}
```

例题4 Given a string s, we can partition s such that every segment is a palindrome (e.g, ‘abba’ is a palindrome, ‘a’ is a palindrome, ‘ab’ is not). Please write a function to return the minimum cuts needed for a palindrome partitioning, given string s.

给定一个字符串 s，我们可以将 s 切割使得切割后的每一段都是一个回文。例如，‘abba’是回文，‘a’也是回文，但‘ab’不是回文。请编写一个函数，针对给定的字符串，返一个回文分割所需的最小的分隔次数。

解题分析：本题*是最小值问题，有很强的聚合性，用 DP。*

本题可以分为两个部分：判断字串是不是回文（palindrome），如何切割原字符串。

对于问题的第一部分，*实际上也是一个递归问题*，递推关系是：

isPalindrome(i , j) = (value(i) == value(j)) AND (isPalindrome(i+1, j-1) OR j - i <= 1) ，i 和 j 分别表示 substring 的首坐标和尾坐标。

注意 i 的前驱坐标是 i+1，j 的前驱坐标是 j-1，所以在具体处理时，前者是倒序遍历，后者是顺序遍历，其实也是利用了自底向上，自底向上*的处理方向总是与直观理解的方向相反，这样可以确保每次*

都能调用已经计算过的结果。

再来考虑 cut 的问题，因为首坐标的遍历顺序是倒序，因此可以将 minCut(i)定义为：将原字符串最末字符到第 i 个字符视为字串，依题意处理这个字串需要的最少 cut 数量。所谓的“最少”，同样是一个 DP 问题，递归式如下（请深刻理解该递归式，所有 DP 相关的最大/最小问题都可以总结为类似的递归式）：

minCut(i) = min(minCut(j+1)+1), for i <= j < n, and substring(i , j) is palindrome。

直观上说，如果 substring(i , j)是回文，那么一种分割方式就是将 i 到 j 视为一个字串，j+1 到字符串按照 minCut(j+1)的方式分割。这样，minCut(i)需要在 minCut(j+1)的基础上再分割一次(substring(i , j))。最终，最外层的最小值符号确保保存 minCut(i)是所有这些分割中最优的一个。

注意在算法开始的时候，我们需要对于最小值初始化。通常，我们将其初始化为最坏的解：本题中，最坏情况就是每一个字符都需要分割的情况。这样，随着算法的进行，该值就会被逐步优化。

参考解答：

```
int minCut(string s) {
    if(s.empty())   return 0;
    vector<vector<bool> > palin(s.size(), vector<bool>(s.size(), false));
    vector<int> minCut(s.size()+1,0);
    for(int i = 0; i <= s.size(); i++)
        minCut[i] = s.size() - i - 1;

    for(int i = s.size() - 1; i >= 0; i--) {
        for(int j = i; j < s.size();++j) {
            if(s[i] == s[j] && ( j - i <= 1 || palin[i+1][j-1] ) ) {
                palin[i][j] = true;
                minCut[i] = min(minCut[j+1]+1, minCut[i]);
            }
        }
    }

    return minCut[0];
}
```

特别地，具有简单“聚合”性质的问题，如最值或者求和问题，往往可以进一步优化 DP 表的空间。应该更确切地说，如果只在乎紧邻的前一个的局部解，而不在乎前几个局部解的问题，就可以接受每

次在计算当前解的时候，替换掉那个最优解。毕竟，我们只在乎最值，而不在乎次最值（假定在乎次最值，那么就需要两个变量，以此类推）；只在乎总和，而不在乎总和的一部分（所以只要放心地叠加上去就行了，覆盖过去的总和）。其实最简单的例子就是求一个数组内的数的最值/总和：我们自左向右遍历数组，存储当前的最值或者和。所存储的恰恰就是从下标 0 到当前下标的计算结果，是原问题的一个子问题。在这个过程中，我们只需要一个全局变量就够了，因为总是可以把过去的局部最优解替换掉。

对于 DP 表的优化仍然是一个比较复杂的优化，不用强求，建议先解决问题，再寻求简化。用一个数组而不是若干个变量解决问题绝不会导致你面试不通过。

例题 5 How many paths are there for a robot to go from (0,0) to (x,y), supposing it can only move down and move right.

假设一个机器人只能够向下和向右移动，它从（0,0）移动到（x,y）有多少条路径。

解题分析：这是一个具有收敛性的数量问题，需要在 x 轴和 y 轴两个维度上使用二维 DP。

递推关系：Paths(i, j) = Paths(i+1, j) + Paths(i, j+1); i 和 j 分别表示起点的横纵坐标。DP table 可以是一个二维数组，这样的解答在实际面试中已经足够好。但事实上，根据上述 DP table 的优化理论，我们至少可以选取一个维度来化简，因为我们只关心紧接的那一行/列的结果，而不在乎之前的行/列的结果，我们可以将新的结果叠加上去，这样可以仅用一个 array 作为 DP table，详见解答。

参考解答：

```
int uniquePaths(int m, int n) {
    int ways[n] = {0};

    ways[n-1] = 1;
    for(int i = m - 1; i >= 0; --i)
        for(int j = n - 2; j >= 0; --j )
            ways[j] += ways[j+1];
    return ways[0];
}
```

例题 6 Given a value N, this N means we need to make

change for N cents, and we have infinite supply of each of S = { S1, S2, .. , Sm} valued coins, how many ways can we make the change?

给定一个值 N，这个 N 表示我们需要换成 N 个美分的零钱，并且我们拥有可以无限供应的、面值为 S = { S1, S2, .. , Sm} 的硬币，求问找零的方式有多少种？

解题分析：本问题以及其他类似描述的换零钱问题，*都属于典型的分布在二维整数空间上的计数问题，用 DP。*

重要的是理解如下递推关系：

对于第 j 种 coin，无非是选择和不选择使用两种可能（i 是目标钱数，j 是当前 coin 的下标）。

ways(i, j) = ways(i-s(j), j) + ways(i, j-1); i [0,N], j [1,m]注意到在 j 这个维度上，只在意紧邻的上一步的结果，而不在意过去几步的结果，最后仍然是求和，因此上一步的局部解也只是当前解求和过程当中的一部分，完全可以覆盖，仅用一个变量表示，因此 DP table 可以简化为 i 方向一维空间。

参考解答：

```
int countWays(vector<int> S, int m, int n) {
    vector<int> table(n+1, 0);
    table[0] = 1;
    for(int i = 1; i <= n; i++)
        for(int j = 0; j < m; j++)
            table[i] += (i-s[j] >= 0) ? table[i-S[j]] : 0;
    return table[n];
}
```

通过解答可以发现，本题其实和之前例题 1 的走梯子问题是同样类型的问题。请对比加深理解。

例题 7 Given a value N, this N means we need to make change for N cents, and we have infinite supply of each of S = { S1, S2, .. , Sm} valued coins. Please implement a function which gets the minimal number of coins for N cents.

给定一个值 N，这个 N 表示我们需要换成 N 个美分的零钱，并且我们拥有可以无限供应的、面值为 S = { S1, S2, .. , Sm} 的硬币。请实现一个函数，它可以求得组合为 N 美分的最少的硬币数目。

解题分析：本题也是换零钱（Coin Change）类型的问题，与上题的区别在于，本题是一个建立在二维整数空间上的*最值问题*，用 DP。

递推关系：对于第 j 种 coin，还是选择和不选择使用两种可能，唯一的区别在于对于每种选择，需要再取一个最小值以记录最优结果：

minNum(i, j) = min(minNum(i-s(j), j) + 1, minNum(i, j-1)); i [0,N], j [1,m]

与例题 6 类似，在 j 这个维度上，我们只在乎最小值，而上一步的局部解只是候选中的一个，可以被覆盖。因此可以简化为一维空间。

参考解答：

```
int minNum(vector<int> S, int m, int n) {
    vector<int> table(n+1, INT::MAX);
    table[0] = 0;
    for(int i = 1; i <= n; i++)
        for(int j = 0; j < m; j++) {
            if( i >= s[j] && table[i] > table[i-s[j] )
                table[i] = table[i-s[j]] + 1;
    }
    return table[n];
}
```

8.2.2 最长子序列类型的问题

“最长子序列”问题（即有限空间内，满足一定条件的最长顺序子序列），本身具有很强的聚合性，可以以如下方式解答：用 DP 表来记录以当前节点为末节点的序列的解（至少固定问题的一端，因此不是以“当前节点或之前节点”为末节点）的解，并根据递推关系，由问题空间的起点到达问题空间的终点。

例题 8 Find the longest increasing subsequence in an integer array. E.g, for array {1, 3, 2, 4}, return 3.

在一个整数数组中，找到最长的递增的子序列。例如，对于数组{1, 3, 2, 4}，返回 3。

解题分析：*用 DP* 表来记录以当前节点为末节点的序列的最大长度，其数值取决于当前节点之前的所有节点：如果当前节点对应的数组数值大于之前的某个节点，那么可以将当前节点对应的数组数值 append 在该节点的最长序列之后。最终，我们在 DP 表中将当前节点的结果更新为所有可能解的最大值。递推关系如下：

maxLength(i) = max{ maxLength(k), k = 0~i-1 and array[i] > array[k] } + 1;另外，如果需要输出最长序列，那么无非就是对于每个节点额外记录一个 index，该 index 是以当前节点为末节点的最

长序列中，前驱元素在数组中的下标。

参考解答：

```
int longestIncreasingSubsequence( int arr[], int n) {
    vector<int> maxLength(n, 1);
    int global_max = 0;
    for (int i = 0; i < n; i++)
        for(int j = 0; j < i; j++)
            if ( arr[i] > arr[j] && maxLength[j] + 1 > maxLength[i] )
                maxLength[i] = maxLength[j] + 1;

    for(int i = 0; i < n; i++)
        if ( global_max < maxLength[i] )
            global_max = maxLength[i];
    return global_max;
}
```

例题 9 Given the heights and weight of each person in the circus, compute the largest possible number of people in tower. (each person has to be both shorter and lighter than the person below him/her)

现在有一个马戏团，我们获得了马戏团中每个人的身高体重。马戏团要表演“人塔”，要求在上层的人比下面的人更轻并且更矮，求“人塔”的最大高度。

解题分析：将 People 以 Height 进行排序，在高度相同的情况下，按照 Weight 排序。这样，把问题划归为以 Weight 为基准的 LIS（Longest Increasing Subsequence）问题（因为选出的序列自然满足高度递增），这样就与例题 8 无甚区别。

例题 10 Find the contiguous subarray within an array (containing at least one number) which has the largest sum.

在一个数组中找出一个连续的子数组，它至少包含 1 个数字，并且所有数字的和最大。

解题分析：只观察以当前节点为末节点可能的最大 sum，并记录一个 global sum。对于当前节点，需要判断加入对应数组元素能否使得 sum 变大。递推公式如下：

sum[i] = max(sum[i-1] + A[i], A[i])因为需要连续的子数组，故计算当前的最大 sum，只在乎前一次计算的结果，因此用一个变量每次覆盖即可（正如我们之前关于简化 DP 空间的描述）。

参考解答：

```
int maxSubArray(int A[], int n) {
    if(n <= 0)  return 0;
    int max_sum = A[0], sum = A[0];
    for(int i = 1; i < n; i++){
        sum = max(sum + A[i], A[i]);
        if(sum > max_sum)
            max_sum = sum;
    }
    return max_sum;
}
```

例题 11 Suppose you are traveling along a circular route. On that route, we have N gas stations for you, where the amount of gas at station i is gas[i]. Suppose the size of the gas tank on your car is unlimited. To travel from station i to its next neighbor will cost you cost[i] of gas. Initially, your car has an empty tank, but you can begin your travel at any of the gas stations. Please return the smallest starting gas station's index if you can travel around the circuit once, otherwise return -1.

假设你要遍历一个环形路径。在这条路径上，有 N 个加油站供你使用，而加油站 i 的汽油量为 gas[i]。假设你的汽车的油箱的大小是无限的。要从加油站 i 到其邻近的下一个加油站，将会用掉 cost[i] 的汽油。一开始你的汽车的油箱是空的，但是，你可以从任何一个加油站开始遍历。如果能够遍历环形路径一次的话，返回所需的开始加油站的最小索引，否则的话，返回-1。

解题分析：这是一个比较难理解的问题，但其本质上还是选择一个序列，只不过这个序列是环形的。事实上，可以考虑对问题进行如下操作：对于第 i 个加油站，它能够给车子提供的净动力为 array[i] = gas[i] - cost[i]。问题转化为，找到一个起始位置 index，将 array 依此向左 shift，即 index->0（index 对应新的数组下标 0），index+1->1…，使得对于任意 0 <= i < n，满足序列和 subSum(0, i) 大于 0。

首先，考虑什么情况下有解。经过上述转换，很明显有解的情况对应于 sum（array）大于等于 0。

那么，剩下的问题是在有解的情况下，如何选择一个正确的起始点。类似于一般序列问题，考虑将当前节点作为序列的末节点。如果

从记录的开始节点(index)起,到当前节点的过程中,一旦出现 subSum 小于 0,那么从开始节点到当前节点的所有节点都不能作为开始节点,因为在过程中一定会出现 subSum 小于 0 的情况，否则累计的结果不会为负。那么，开始点至少是 index+1。另一方面，可以证明，如果从记录的开始点出发可以走到第 n 个加油站，即 subSum(index, n)大于 0，那么该开始点一定能走完全程。

参考解答：

```
int canCompleteCircuit(vector<int> &gas, vector<int> &cost) {
    int size = gas.size();
    int subSum = 0, sum = 0;
    int array[gas.size()];
    int index = 0;
    for(int i = 0; i < size; i++){
        array[i] = gas[i] - cost[i];
        sum += array[i];
    }
    if (sum < 0)
        return -1;
    for(int i = 0; i < size; i++) {
        subSum += array[i];
        if(subSum < 0) {
            subSum = 0;
            index = i + 1;
        }
    }
    return index;
}
```

例题 12 Please write a function to calculate the Longest Common Subsequence (LCS) given two strings. LCS for input Sequences “ABCDGH” and “AEDFHR” is “ADH” of length 3.LCS for input Sequences “AGGTAB” and “GXTXAYB” is “GTAB” of length 4.

请编写一个函数来计算给定的两个字符串的最长公共子序列,即 LCS。输入序列“ABCDGH”和“AEDFHR”的话，LCS 是“ADH”，长度为 3；输入是“AGGTAB”和“GXTXAYB”的话，LCS 是“GTAB”，长度为 4。

解题分析：同时遍历两个序列，*考虑以当前两个节点为末节点的序列的*公共子序列长度。如果其对应的字符相等，那么可以使得 LCS 长度+1，即 append 当前字符；否则，保留较优的结果。递推关系如下：

```
Length(i,j) = (str1[i-1] == str2[j-1]) ? Length(i-1, j-1) + 1 :  Max { Length(i,j-1), Length(i-1,j) }
```

参考解答：

```
int lcs( string str1, string str2) {
    vector<vector<int>> length( str1.size()+1, vector<int>(str2. size()+1);
    for(int i = 0; i < str1.size(); i++) {
        for(int j = 0; j < str2.size(); j++) {
            if(i == 0 || j == 0)
                length[i][j] = 0;
            else if (str1[i-1] == str2[j-1])
                length[i][j] = length[i-1][j-1] + 1;
            else
                length[i][j] = max(length[i-1][j], length[i][j-1]);
        }
    }
    return length[str1.size()][str2.size()];
}
```

特别地，如果当前节点的解，既依赖于前驱问题的解，又依赖于后驱问题的解，但这两部分又互相独立，则可以分别自左开始 DP，计算从最左节点到当前节点的结果；自右开始 DP，计算从最右节点到当前节点的结果；再用同一个 DP Table 来合并解。具体举例如下。

例题 13 Given an integer array, each element is a stock price on day i (i is the index). Design an algorithm to find the maximum profit. You may complete at most two transactions(one buy, one sell).

给定一个整数数组，每个元素都是 i 这一天的一个股票价格，i 是索引。设计一个算法来求解最大收益。你最多可以完成两次交易，一次买入，一次卖出。

解题分析：假设在 i 位置买入，j 位置卖出，那么对于 i，j 之间的某个节点，如何计算其利润？可以分成两部分计算：*从 i 到当前节点的利润，这部分只和前驱问题有关；从当前节点到 j 的利润，这部分只依赖于后驱问题。并且这两部分相互独立，可以把结果叠加在 DP Table 上。*直观上说，相当于在当天卖出又立刻买进，相当于增加了两次虚拟操作。因此，可以以当前节点为分界线，第一次 DP 自左向右，只计算到当前为止可获得的最大收益（即从之前某天买入，当前卖出的最大收益）；第二次 DP 自右向左，计算从当前开始可获得的最大收益（即从当前买入，之后某天卖出的最大收益）。两部分收益之

和即为总收益。

参考解答：

```
int maxProfit(vector<int> &prices) {
    if(prices.empty())
        return 0;

    int n = prices.size();
    vector<int> dp(n, 0);

    // left scan
    int minPrice = prices[0];
    for(int i = 0; i < n; ++i) {
        dp[i] = prices[i] - minPrice;
        if(prices[i] < minPrice)
            minPrice = prices[i];
    }

    int globalProfit = 0;

    // right scan
    int maxPrice = prices[n-1];
    for(int i = n-1;i >= 0;--i) {
        if(prices[i] > maxPrice)
            maxPrice = prices[i];
        dp[i] += maxPrice - prices[i];
        globalProfit = max(globalProfit, dp[i]);
    }

    return globalProfit;
}
```

例题 14 Given an array of integers, write a function to replace each element with the product of all elements other than that element.

给定一个整数数组，编写一个函数来替换每个元素，使用除了该元素之外的所有元素的乘积来替换。

解题分析：*当前节点的解，既和左边的元素有关，又与右边的元素有关，两者相互独立，可以用双向 DP。*左遍历 DP 计算积累到目前为止的乘积，右遍历 DP 计算从目前开始到最后的乘积。

参考解答：

```
void replaceWithProducts(int elements[], int n) {
    int product = 1;
    vector<int> table(n,1);
    for(int  i = 0; i < n; i++){
        table[i] = product;
        product *= elements[i];
    }

    product = 1;
    for(i = n-1; i >=0 ; i--){
        elements[i] = table[i] * product;
        product *= elements[i];
    }
}
```

例题 15 You are given an array of n non-negative integers. Each value means the height of a histogram. Suppose you are pouring water onto them, what is the maximum water it can hold between a left bar and a right bar (no seperation)?

给定 n 个非负的整数的一个数组。每个值表示一个柱形的高度。假设你向柱形上倾注水，那么左柱之间和右柱之间储水量的最大值是多少？

解题分析：*当前节点的解，取决于左右两边的海拔高度*，可以自左向右遍历加数组，求出左侧海拔高度，再从右向左遍历加数组，求出右侧海拔高度：当前节点的储水量，等于左侧最高海拔与右侧最高海拔的较小值减去当前节点的海拔。

参考解答：

```
int trap(int A[], int n) {
    if(n <= 0)  return 0;
    vector<int> dp(n,0);

    int left_max = 0, right_max = 0,water = 0;

    for(int i = 0; i < n; i++) {
        dp[i] = left_max;
        if(A[i] > left_max)
            left_max = A[i];
        }

    for(int i = n -1; i >= 0; i--) {
        if(min(right_max,dp[i]) > A[i])
```

```
            water += min(right_max,dp[i]) - A[i];

        if(A[i] > right_max)
            right_max = A[i];
    }
    return water;
}
```

8.2.3 用 Memorization（自顶向下）解决收敛结构问题

Memorization 是自顶向下形式的动态规划，并且受到的制约更少（之前在 8.1“知识要点”部分已有讨论），自然也可以用来解决前述的问题（但空间上可能效率不及自底向上形式的 DP）。

Memorization 的核心在于，在原有递归框架下，存储子问题的计算结果，在重复计算子问题时返回已经计算的值。

值得注意的是，这里所谓的“重复计算子问题”，在自顶向下结构中必须与前驱节点无关，因为子问题并不知道原问题是如何到达当前节点的。举例来说，求二叉树从根节点到叶节点的权值最大路径，对于当前节点到叶节点的路径与之前如何到达当前节点没有关系，只要计算当前节点到叶节点的路径，就一定是重复的计算，可以直接返回结果。作为反例，在一个字母矩阵中寻找词典中的单词，当前路径能否构成单词，不仅与之后走的过程有关，也与之前的过程有关。因此，从当前节点出发，哪怕走过相同的路径，也不能看成是重复计算的子问题。在回溯部分我们会进一步讲解。

例题 16 Given a set of boxes, each one has a square bottom and height of 1. Please write a function to return the tallest stack of these boxes. The constraint is that a box can be put on top only when its square bottom is restrictively smaller.

给定一组盒子，每个盒子都有一个方形的底子且高度为 1。请编写一个函数，返回这些盒子的最高的堆。限制条件是，只有当一个盒子的方形底严格来说较小的时候，它在可以放到堆的顶部。

解题分析：放置在当前盒子上的盒子序列是否满足条件，*与当前盒子下面已经放好的盒子*（前驱条件）无关，只与将要放在当前盒子上的盒子序列有关（后驱节点），并且在计算后驱条件时，以每个盒子为底的最长序列必然会被反复计算。因此*可以将盒子作为*

Memorization 的 key，以满足条件的最长序列作为 Memorization 的 value。

参考解答：

```
vector<Box> createStackDP( Box boxes[], const int num, Box bottom, unordered
_map< Box, vector<Box> >& stackCache) {
    vector<Box> max_stack;
    int max_height = 0;
    vector<Box> new_stack;

    // memorization
    if( stackCache.count( bottom ) > 0 )
        return stackCache[ bottom ];
    else {
        for( int i = 0; i < num; i++ ) {
            if( Box[i].canBeAbove( bottom ) ) {
                // solve subproblem
                new_stack = createStackDP( boxes, num, Box[i], stackCache );
            }
            if( new_stack.size() > max_height ) {
                max_height = new_stack.size();
                max_stack = new_stack;
            }
        }
    }

    max_stack.insert( max_stack.begin(), bottom );
    stackCache[ bottom ] = max_stack;
    return max_stack;
}
```

例题 17（例题 3） Given a string and a dictionary of words, please write a function to add space into the string, such that the string can be completely segmented into several words, where every word appears in the given dictionary.

给定一个字符串和包含一些单词的一个字典，编写一个函数来给字符串添加空格，使得字符串能够完全由几个单词组成，并且其中每个单词都存在于给定的字典之中。

解题分析：本题事实上是例题 3 的另一种描述，为的是让你了解自上而下一样可以解决用自底向上解决的问题。考虑使用 Memorization 的理由是：*要求满足特定条件的所有解，当前节点以后的拆分方法是否满足条件，与当前节点之前的拆分方法无关。并且在*

*计算子问题时，会有子串的拆分被重复计算。*因此可以用 substring 作为 Memorization 的 key，拆分后的结果集合作为 Memorization 的 value。

参考解答：

```
vector<string> wordBreak(string s, unordered_set<string> &dict, unordered_m
ap<string,vector<string> >& cache) {
    // memorization
    if(cache.count(s))
        return cache[s];

    vector<string> vs;

    if(s.empty()) {
        vs.push_back(string());
        return vs;
    }

    for(int len = 1; len <= s.size(); ++len ) {
        string prefix = s.substr(0,len);
        if(dict.count(prefix) > 0) {
            string suffix = s.substr(len);
            // solve subproblem
            vector<string> segments = wordBreak(suffix,dict,cache);
            for(int i = 0; i < segments.size(); ++i) {
                if(segments[i].empty())
                    vs.push_back(prefix);
                else
                    vs.push_back(prefix + " " + segments[i]);
            }
        }
    }

    cache[s] = vs;
    return vs;
}
vector<string> wordBreak(string s, unordered_set<string> &dict) {
    if(s.empty())
        return vector<string>();
    unordered_map<string,vector<string> > cache;
    return wordBreak(s,dict,cache);
}
```

8.2.4 用回溯法（自上而下）解决发散结构问题

对于发散性问题（例如“所有组合”，“全部解”），可以选取其问题空间“收敛”的一端作为起点，沿着节点发散的方向（或者说，当前节点的多种选择）进行递归，直到（a）当前节点“不合法”或（b）当前节点发散方向搜索完毕，才会 return。举例来说，考虑树的遍历：根节点方向就是“收敛”的一端，节点发散的方向就是子节点。对于某个树的节点，其孩子就是当前决策的多种选择。当达到叶节点是，其孩子为 NULL，即达到“不合法”的边界条件。回溯法的核心在于选择哪些方向/决策，才是最合理，不重复的。所谓“剪枝”（pruning），就是指：只选择尽可能少的、可能到达“胜利条件”的方向，而不是搜索当前节点的所有发散方向。这样，可能将幂指数级的复杂度降低到阶乘级。

值得注意的是，invalid 前的最末节点未必意味着胜利（不是所有的问题走通就算满足条件），胜利的节点也未必代表不需要继续走下去（比如寻找到一个单词之后，继续走下去可能能找到以这个单词为前缀的另一个单词）。因此我们强烈推荐将 invalid 的判定与胜利条件的判定总是分开，即使在某些题目中它们是一致的。当然，如果经过充分剪枝之后，所有搜索只会沿着“正确”的方向行进，那么当前节点“不合法”往往也就意味着胜利条件。

如果需要记录决策的路径，可以用 vector<int> &path 沿着搜索的方向记录，在满足胜利条件时记录当前 path（通常是将 path 存入 vector<vector<int>> &paths）。

注意，我们传入的 path 是引用形式，属于全局变量。Backtracking（回溯）本身隐含的含义是，在访问完这个节点返回时，需要恢复原本的状态（即回到该节点），以访问其他路径。具体实现时，意味着需要:

（1）在 return 前，删除 path 中的当前节点。

（2）如果搜索的方向有出现环路的可能，那么可以使用 bool [] 或 unordered_map 来记录该节点是否已被使用，在访问时以及 return 前维护。

如果以传值形式传入 path，由于 path 成了局部变量，故在某些情况下不需要显式回溯，相当于把状态复制给了子问题。可能有人觉得这样做比较直观，但其缺点是需要额外的空间。

回溯法的典型模板如下所示：

```
void backtracking( P node, vector<P> &path, vector<vector<P> >&paths ){
       if(!node )  // invalid node
           return;
       path.push_back(node);
       bool success =  ;  // condition for success
       if( success )
           paths.push_back( vector<P>(path.begin(),path.end()) ); // don't r
eturn here

       for( P next: all directions )
           backtracking( next, path, paths );
       path.pop_back();
       return;
}
```

例题 18 Given n pairs of parentheses, generate all valid combinations of parentheses. E.g. if n = 2, you should return ()(), (())

给定 n 对括号，生成括号的所有合法组合。例如，如果 n=2，应该返回()()和(())。

解题分析：*由于题目要求找出所有解，故属于发散性的 DP，用*回溯法。核心在于：*对于当前节点，也哪些可用的选择。*对本题而言：

如果所剩的左括号大于 0,那么继续添加左括号一定是一种决策。

如果所剩的左括号少于右括号，那么补充右括号也一定是一种决策。

应该按照决策的选择方向进行回溯。

参考解答：

```
void parenthesesCombination(int leftRem, int rightRem, string &path, vector<
string> &paths ){
   if(leftRem < 0 || rightRem < 0)
       return;

   if(leftRem > 0) {
       // make choice
       path.push_back('(');
       parenthesesCombination(leftRem-1,rightRem,path,paths);
       // backtracking
       path.pop_back();
 }
```

```
    if(leftRem < rightRem) {
        // make choice
        path.push_back(')');
        rightRem -= 1;
        if(rightRem == 0)
            paths.push_back(path);    // winning
        parenthesesCombination (leftRem, rightRem, path, paths);
        // backtracking
        path.pop_back();
    }
}
vector<string> generateParenthesis(int n) {
    vector<string> res;
    if(n <= 0)
        return res;
    string path;
    parenthesesCombination(n, n, path, res);

    return res;
}
```

例题 19 Please write a function to find all ways to place n queens on an n×n chessboard such that no two queens attack each other.

请编写一个函数，找到在一个 n×n 棋盘上放置 n 个皇后的所有方法，要求是两个皇后无法彼此攻击。

解题方向：本题是经典的八皇后问题。*由于需要找到所有解，属于发散性问题*。对于每一行，我们枚举可以将当前的皇后放在哪一列，所有目前为止可行的列都可以作为选择进行 DP，即每次选择方向时都经过充分剪枝，使得每一步都朝着胜利条件前进。这样，最后得到的解一定是需要的解。

复杂度分析：回溯总共 n 步，每次供选择的方向为 n。经过剪枝之后，可以认为复杂度小于 n!。

参考解答：

```
bool checkValid( int row1, int col1,int *rowCol ) {
    for( int row2 = row1 - 1; row2 >= 0; row2--) {
        if( rowCol[row2] == col1 )
            return false;
        if( abs(row1 - row2) == abs( rowCol[row2] - col1 ) )
            return false;
    }
```

```
    return true;
}

void placeQ( int row, int rowCol[], vector<int*>& res ) {
    if (row == GRID_SIZE) {
        //winning
        int p[GRID_SIZE];
        for( int i =0; i < GRID_SIZE; i++)
            p[i] = rowCol[i];
        res.push_back(p);
        return;
    }

    int col = 0;
    for(col = 0; col < GRID_SIZE; col++) {
        if( checkValid(row, col, rowCol) ) {
            rowCol[row] = col;
            placeQ( row+1, rowCol, res);
            // because we rewrite rowCol[row] everytime,
 // so backtracking is inferred here
        }
    }
}
```

例题 20 Given a collection of integers that might contain duplicates, return all possible subsets.

给定整数的一个集合，其中可能包含重复的值，返回其所有可能的子集。

解题分析：如果不存在重复，那么对于当前节点，分为选取当前元素和不选取当前元素两条路径。

当存在重复时，那么对于当前节点，也可以分为选取当前元素和不选取当前元素。但是，如果没有选择当前元素，那么也一定不能选择后驱节点中与当前元素重复的任何元素，否则会产生完全重复的路径。

至于如果选取了当前元素，那么之后这个支线内部的重复问题，支线自然会解决，不需要关心。原因在于，*子问题和原问题相互独立*，子问题不需要关心如何来到当前节点。这样既保证了对于 set{1,2,2}，subset{1, 2}只被选择一次，也保证了{1,2,2}会被选择在内。

参考解答：

```
void subsetsWithDup(int index, const vector<int> &S, vector<int> &path, vect
or<vector<int> > &paths) {
    if(index == S.size())
        return;
    for(int i = index; i < S.size(); i++) {
        if( i != index && S[i] == S[i-1])
            continue;
        path.push_back(S[i]);
        paths.push_back(path);
        subsetsWithDup(i+1, S, path, paths);
        path.pop_back();
    }
}

vector<vector<int> > subsetsWithDup(vector<int> &S) {
    vector<vector<int>> paths;
    vector<int> path;
    paths.push_back(path);
    sort(S.begin(), S.end());
    subsetsWithDup( 0, S, path, paths );
    return paths;
}
```

例题21 Given a collection of numbers that might contain duplicates, return all possible unique permutations.

给定数字的一个集合，可能包含有重复的值，返回所有可能的、唯一的组合。

解题分析：选择后驱元素中的一个与当前节点交换，然后再将后面的节点作为子问题考虑。由于有重复元素，而重复元素对当前节点的影响是相同的，因此应该去重：把相同元素的替换为同一个回溯方向/选择来处理，一旦发现是已经处理过的相同元素，则直接跳过。

参考解答：

```
void permuteHelper(int index, vector<int> &num, vector<vector <int>> &paths
 ) {
    if(index > num.size()) {
        return;
    }
    if(index == num.size()) {
        paths.push_back(num);
    }

    unordered_set<int> used;
```

```
    for(int I = index; I < num.size(); I++ ) {
        // handle duplicates
        if(used.count(num[I]))
            continue;
        // make choice
        swap(num,index,I);
        permuteHelper(index+1,num,paths);
        // backtracking
        swap(num,index,I);
        used.insert(num[I]);
    }
}
vector<vector<int>> permuteUnique(vector<int> &num) {
    vector<vector<int>> paths;
    permuteHelper(0,num,paths);
    return paths;
}
```

例题 22 Solve a boggle game(http://en.wikipedia.org/wiki/Boggle), with a dictionary given as unordered_set.

求解一个猜词游戏(http://en.wikipedia.org/wiki/Boggle)，给定一个 unordered_set 作为字典。

解题分析：

（1）猜词游戏（boggle game）实际上就是寻找有限路径，判断路径能否组成单词。从 matrix 的每一个 slot 出发进行回溯，直到超过 matrix 的边缘，或者当前 slot 已经被访问过（环路可能），或者路径超过了最长单词的长度，就这 3 种 invalid 条件。胜利条件是当前的路径组成一个词典中的单词，但胜利条件之后应当继续回溯（统一地，对任何回溯问题，在处理上胜利条件之后都应该继续回溯，只不过很多时候胜利节点就是末节点，因此回溯下去也会因为 invalid 条件返回而已）。

（2）考虑对这个问题剪枝，可以将 dictionary 改写成 prefix tree，这样才每一步查询时，可以根据当前路径序列是否是一个有效的 prefix，增加一个 invalid 的条件，大幅提高搜索的效率。Prefix tree 的 DFS，可以与问题的回溯同步进行，matrix 单条搜索路径中的节点，对应 prefix tree 单条搜索路径中的节点。

利用回溯的另一种情景是：就算是处理收敛结构问题，如果无论从哪一端出发，都避免不了“（部分）当前节点的解依赖后驱节点”

（也就是说，当前节点，如果不能获知后驱节点，就无法得到有意义的解)的情况,那么可以也用回溯解决(请参考本书第 4 章中的叙述)。

8.2.5 用 D&C 策略解决独立子问题

如果能将问题由几个孤立但类似的部分组成,则可以优先选择使用 D&C 策略：将问题分割解决，再合并结果。特别地，如果期望将问题的复杂度由 O(n)进一步降低到 O(logn)，一般总是可以联想到使用 D&C 策略，将问题分割而治。

例题 23 Implement pow(x, n).

实现 pow(x, n)。

解题分析：由于 pow(x, n)相当于 n 个 x 相乘，*左半边乘积与右半边相互独立且类似，所以可以用 D&C 策略进行二分*。注意，二分之后需要处理 n 大于 0，等于 0，小于 0 等情况。

参考解答：

```
double pow(double x, int n) {
    if (n == 0)
        return 1.0;
    if( abs(x) < numeric_limits<double>::epsilon() )
        return 0.0;
    double half = pow(x,n/2);
    if(n%2 == 0)
        return half*half;
    else if(n > 0)
        return half*half*x;
    else
        return half*half/x;
}
```

例题 24 Evaluate an infix notation(without parentheses). E.g 4+3*2 returns 10

计算一个不包含括号的算术式。例如，4+3*2 返回 10。

解题分析：对于一个不包含括号的算术式，如果看到*或\，则应该立刻计算。如果看到+或-，则应该解决加减号之后表达式的子问题（devide），并且把子问题的结果叠加到当前结果（conquer）。在这种情况下，由于当前节点已经读入了字符（加号或减号），我们需要吐出这个字符，目的是将其作为正负号传给子问题。在读取数字的时候，我们可以用 sscanf，关于 sscanf 的详细介绍请见

http://www.cplusplus.com/reference/cstdio/sscanf/ 。

参考解答：

```
double readNum( char * &str) {
    double num;
    sscanf( str, "%lf", &num );
    str++;
    while( isdigit(*str) || *str == '.' )
        str++;
    return num;
}

double evaluate( char *str) {
    if( *str == '\0' )
        return 0.0;

    double res = 1.0;
    char op = '*';

    while( op == '*' || op == '/' ) {
        if( op == '*' )
            res *= readNum(str);
        else
            res /= readNum(str);
        op = *str++;
    }

    return res + evaluate( --str);
}
```

第 9 章 排序和搜索

9.1 知识要点

9.1.1 常见的内排序算法

所谓的内排序是指所有的数据已经读入内存，在内存中进行排序的算法。排序过程中不需要对磁盘进行读写。同时，内排序也一般假定所有用到的辅助空间也可以直接存在于内存中。与之对应地，另一类排序称作外排序，即内存中无法保存全部数据，需要进行磁盘访问，每次读入部分数据到内存进行排序。我们在 9.1.2“常见的外排序算

法”中讨论该类问题。

1．合并排序

合并排序（Merge Sort）是一种典型的排序算法，应用“分而治之(divide and conquer)”的算法思路,将线性数据结构(如 array、vector 或 list）分为两个部分，对两部分分别进行排序，排序完成后，再将各自排序好的两个部分合并还原成一个有序结构。由于合并排序不依赖于随机读写，因此具有很强的普适性，适用于链表等数据结构。算法的时间复杂度为 O(nlogn),如果是处理数组需要额外 O(n）空间，处理链表只需要 O（1）空间。算法实现如下：

```
void merge_sort( int array[], int helper[], int left, int right){
    if( left >= right )
        return;

    // divide and conquer: array will be divided into left part and right par
t
    // both parts will be sorted by the calling merge_sort
    int mid = right - (right - left) / 2;
    merge_sort( array, helper, left, mid );
    merge_sort( array, helper, mid + 1, right);

    // now we merge two parts into one
    int helperLeft = left;
    int helperRight = mid + 1;
    int curr = left;
    for(int i = left; i <= right; i++)
        helper[i] = array[i];
    while( helperLeft <= mid && helperRight <= right ){
        if( helper[helperLeft] <= helper[helperRight] )
            array[curr++] = helper[helperLeft++];
        else
            array[curr++] = helper[helperRight++];
    }

    // left part has some large elements remaining. Put them into the right s
ide
    while( helperLeft <= mid )
        array[curr++] = helper[helperLeft++];
}
```

当递归调用 merge_sort 返回时，array 的左右两部分已经分别由子函数排序完成，我们利用 helper 数组暂存 array 中的数值，再利用两个 while 循环完成合并。helper 数组的左右半边就是两个排

序完的队列，第一个 while 循环相当于比较队列头，将较小的元素取出放入 array，最后使得 array 的左半边由小到大排序完成。第二个 while 循环负责扫尾，把 helper 左半边剩余元素复制入 array 中。注意，此时我们不需要对 helper 右半边做类似操作，因为即使右半边有剩余元素，它们也已经处于 array 中恰当的位置。

关于合并排序更多理论方面的讨论，请见 9.3“工具箱”。

2. 快速排序

快速排序（Quick Sort）是最为常用的排序算法，C++自带的排序算法的实现就是快速排序。该算法以其高效性，简洁性，被评为 20 世纪十大算法之一（虽然合并排序与堆排序的时间复杂度量级相同，但一般还是比快速排序慢常数倍）。快速排序的算法核心与合并排序类似，也采用“分而治之”的想法：随机选定一个元素作为轴值，利用该轴值将数组分为左右两部分，左边元素都比轴值小，右边元素都比轴值大，但它们不是完全排序的。在此基础上，分别对左右两部分递归调用快速排序，使得左右部分完全排序。算法的平均时间复杂度是 O(nlogn)，在最坏情况下为 O(n^2)，额外空间复杂度为 O(logn)。算法实现如下：

```
int partition( int array[], int left, int right ) {
    int pivot = array[right];
    while( left != right ){
        while( array[left] < pivot && left < right)
            left++;
        if (left < right) {
            swap( array[left], array[right--]);
        }
        while( array[right] > pivot && left < right)
            right--;
        if( left < right )
            swap( array[left++], array[right]);
    }
    //array[left] = pivot;
    return left;
}
void qSort( int array[], int left, int right ){
    if( left >=right )
        return;
    int index = partition( array, left, right);
    qSort(array, left, index - 1);
    qSort(array, index + 1, right);
}
```

partition 函数先选定数组 right 下标所指的元素作为轴值，用 pivot 变量存储该元素值。然后，右移 left，即从左向右扫描数组，直到发现某个元素大于轴值或者扫描完成。如果某个元素大于轴值，则将该元素与轴值交换。该操作特性在于：保证交换后轴值左侧的元素都比轴值小。再次，左移 right，即从右向左扫描数组，直到发现某个元素小于轴值或者扫描完成。如果某个元素小于轴值，则将该元素与轴值交换。该操作特性在于：保证交换后轴值右侧的元素都比轴值大。重复上述过程直到 left 和 right 相遇，最终相遇的位置即为轴值所在位置。由于上述操作的特性，最终轴值左侧的元素都比轴值小，轴值右侧的元素都比轴值大。

关于快速排序的更多理论讨论请见 9.3“工具箱”。C++标准模版库提供函数 sort，实现快速排序的功能：sort(iterator first, iterator last, Compare comp); // can also be pointers here。

3. 堆排序

堆排序（Heap Sort）利用了我们在第 5 章中提到的堆作为逻辑存储结构，将输入 array 变成一个最大值堆。然后，我们反复进行堆的弹出操作。回顾之前所述的弹出过程：将堆顶元素与堆末元素交换，堆的大小减一，向下移动新的堆顶以维护堆的性质。事实上，该操作等价于每次将剩余的最大元素移动到数组的最右边，重复这样的操作最终就能获得由小到大排序的数组。初次建堆的时间复杂度为O(n)，删除堆顶元素并维护堆的性质需要 O(logn)，这样的操作一共进行 n 次，故最终时间复杂度为 O(nlogn)。我们不需要利用额外空间，故空间复杂度 O(1)。具体实现如下：

```
void heapSort(int array[], int size)  {
    Heapify(array, size);
    for (int i = 0; i < size - 1; i++)
       popHeap(array);
}
```

Heapify 和 popHeap 的实现参考本书第 5 章。

4. 桶排序和基数排序

桶排序（Bucket Sort）和基数排序（Radix Sort）不需要进行数据之间的两两比较，但是需要事先知道数组的一些具体情况。特别地，桶排序适用于知道待排序数组大小范围的情况。其特性在于将数据根据其大小，放入合适的“桶（容器）”中，再依次从桶中取出，形成有序序列。具体实现如下：

```
void BucketSort(int array[], int n, int max)
```

```
{
    // array of length n, all records in the range of [0,max)
    int tempArray[n];
    int i;
    for (i = 0; i < n; i++)
        tempArray[i] = array[i];

    int count[max];    // buckets
    memset(count, 0, max * sizeof(int));
    for (i = 0; i < n; i++)    // put elements into the buckets
        count[array[i]]++;
    for (i = 1; i < max; i++)
        count[i] = count[i-1] + count [i];  // count[i] saves the starting in
dex (in array) of value i+1

    // for value tempArray[i], the last index should be count[tempArray[i]]-
1
    for (i = n-1; i >= 0; i--)
        array[--count[tempArray[i]]] = tempArray[i];
}
```

该实现总的时间代价为 O(max+n)，适用于 max 相对 n 较小的情况。空间复杂度也为 O(max+n)，用以记录原始数组和桶计数。

桶排序只适合 max 很小的情况，如果数据范围很大，可以将一个记录的值即排序码拆分为多个部分来进行比较，即使用基数排序。基数排序相当于将数据看作一个个有限进制数，按照由高位到低位（适用于字典序），或者由低位到高位（适用于数值序）进行排序。排序具体过程如下：对于每一位，利用桶排序进行分类，在维持相对顺序的前提下进行下一步排序，直到遍历所有位。该算法复杂度为 O(k*n)，k 为位数（或者字符串长度）。直观上，基数排序进行了 k 次桶排序。具体实现如下：

```
void RadixSort(int Array[], int n, int digits, int radix)
{
    // n is the length of the array
    // digits is the number of digits
    int  *TempArray = new int[n];
    int *count = new int[radix]; // radix buckets
    int i, j, k;
    int Radix = 1; // radix modulo, used to get the ith digit of Array[j]
    // for ith digit
    for (i = 1; i <= digits; i++)  {
        for (j = 0; j < radix; j++)
            count[j] = 0;           // initialize counter
```

```
        for (j = 0; j < n; j++)     {
            // put elements into buckets
            k = (Array[j] / Radix) % radix;  // get a digit
            count[k]++;
        }

        for (j = 1; j < radix; j++) {
            // count elements in the buckets
            count[j] = count[j-1] + count[j];
        }
        // bucket sort
        for (j = n-1; j >= 0; j--)  {
            k = (Array[j] / Radix ) % radix;
            count[k]--;
            TempArray[count[k]] = Array[j];
        }
        for (j = 0; j < n; j++) {
            // copy data back to array
            Array[j] = TempArray[j];
        }
        Radix *= radix;      // get the next digit
    }
}
```

与其他排序方式相比，桶排序和基数排序不需要交换或比较，它更像是通过逐级的分类来把元素排序好。

9.1.2 常见的外排序算法

外排序算法的核心思路在于把文件分块读到内存，在内存中对每块文件依次进行排序，最后合并排序后的各块数据，依次按顺序写回文件。外排序需要进行多次磁盘读写，因此执行效率往往低于内排序，时间主要花费于磁盘读写上。我们给出外排序的算法步骤如下：

假设文件需要分成 k 块读入，需要从小到大进行排序。

（1）依次读入每个文件块，在内存中对当前文件块进行排序（应用恰当的内排序算法）。此时，每块文件相当于一个由小到大排列的有序队列。

（2）在内存中建立一个最小值堆，读入每块文件的队列头。

（3）弹出堆顶元素，如果元素来自第 i 块，则从第 i 块文件中补充一个元素到最小值堆。弹出的元素暂存至临时数组。

（4）当临时数组存满时，将数组写至磁盘，并清空数组内容。

（5）重复过程（3）、（4），直至所有文件块读取完毕。

9.1.3 快速选择算法

快速选择算法（quick selection algorithm）能够在平均O(n)时间内从一个无序数组中返回第k大的元素。算法实际上利用了快速排序的思想，将数组依照一个轴值分割成两个部分，左边元素都比轴值小，右边元素都比轴值大。由于轴值下标已知，则可以判断所求元素落在数组的哪一部分，并在那一部分继续进行上述操作，直至找到该元素。与快排不同，由于快速选择算法只在乎所求元素所在的那一部分，所以时间复杂度是 O(n)。关于算法复杂度的理论分析请见9.3“工具箱”给出的参考资料。我们给出算法实现如下：

```
int partition( int array[], int left, int right ) {
    int pivot = array[right];
    while( left != right ){
        while( array[left] < pivot && left < right)
            left++;
        if (left < right) {
            swap( array[left], array[right--]);
        }
        while( array[right] > pivot && left < right)
            right--;
        if( left < right )
            swap( array[left++], array[right]);
    }
    return left;
}

int quick_select(int array[], int left, int right, int k)
{
    if ( left >= right )
        return array[left];
    int index = partition(array, left, right);
    int size = index - left + 1;
    if ( size == k )
        return array[left + k - 1]; // the pivot is the kth largest element
    else if ( size > k )
        return quick_select(array, left, index - 1, k);
    else
        return quick_select(array, index + 1, right , k - size);
}
```

例题 1 Get the k largest elements in an array with O(n) expected time, they don’t need to be sorted.

一个乱序数组中，如何在 O（n）时间复杂度内找出前 K 大的元素们，并且取出的 K 个元素不需要排序。

解题分析：实际上和快速排序的应用场景是一致的，先找到第 k 大的元素，再将数组重新整理，找出比第 k 大的元素小的所有元素。

例题 2 There are n points on a 2D plan, find the k points that are closest to origin (x= 0, y= 0).

一个 2D 平面上有 n 个点，找到距离原点（x= 0, y= 0）最近的 k 个点。

解题分析：在这里已知点的数量，因此 k 个点到原点的距离构成了大小确定的静态数组，应该对这个数组使用快速选择算法。

9.1.4 二分查找

对于已排序的有序线性容器而言（比如数组、vector），二分查找（Binary search）几乎总是最优的搜索方案。二分查找将容器等分为两部分，再根据中间节点与待搜索数据的相对大小关系，进一步搜索其中某一部分。二分查找的算法复杂度为 O(logn)，算法复杂度的具体分析请见 9.3“工具箱”给出的参考资料。算法实现如下：

```
int binarySearch(int *array, int left, int right, int value) {
    if (left > right) {
        // value not found
        return -1;
    }

    int mid = right - (right - left) / 2;
    if (array[mid] == value) {
        return mid;
    } else if (array[mid] < value) {
        return binarySearch(array, mid + 1, right, value);
    } else {
        return binarySearch(array, left, mid - 1, value);
    }
}
```

对于局部有序的数据，也可以根据其局部有序的特性，尽可能地利用逼近、剪枝，使用二分查找的变种进行搜索。

9.2 模式识别

9.2.1 动态数据结构的维护

维护动态数据（Data Stream）的最大值、最小值或中位数，可以考虑使用堆。如果是动态数据求最大的 k 个元素，因为元素总数量不确定，不能使用快速选择算法，这种情况下也应该用堆解决。

如果需要一个动态插入/删除的有序数据结构，那么可以使用二叉搜索树，因为它天生就是一个动态的有序数组，并且支持检索。

例题 3 Merge k sorted linked lists to be one sorted list.

将 k 个排序后的链表，合并为一个排序的链表。

解题分析：*可以将 k 个 sorted list 想象成 k 个有序数据流，互相竞争插入到结果序列，因此可以考虑使用一个最小值堆维护动态数据*：将每个队头的元素加入一个堆，然后从堆中依次弹出最小数据。如果数据属于第 i 个 list，则该 list 补充一个元素到堆。重复上述过程直到所有元素排序完成。事实上，该算法就是外排序算法的一个具体实现，区别仅仅在于这里略去了文件的读写操作。请仔细类比 9.1.2 小节提到的外排序算法，进一步加深理解。

复杂度分析：时间上，我们需要维护一个大小为 k 的最小值堆，每次维护复杂度 O(logk)，由于一共有 n 个数据，每个数据都会加入最小值堆，故总体时间复杂度 O(nlogk)。由于每个 list 至少有一个数据，故 n 一定大于等于 k。相比于完全无序的 n 个数据排序（所需时间 O(nlogn)），我们的算法将复杂度降至 O(nlogk)。原因在于数据是部分有序的。事实上，在通常面试中，如果数据已经部分有序，我们理应能够实现时间复杂度优于为 O(nlogn)的算法。

空间上，我们需要大小为 k 的最小值堆。同时，在不允许破坏原有链表的情况下，我们需要额外 O(n)的空间构建新链表，故总体空间复杂度为 O(n+k)。如果可以直接修改原有数据的 next 指针，则总体空间复杂度即为 O(k)。至于是否能够破坏原始数据，需要与面试官进行沟通。

参考解答：

```
struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};
struct cmp {
    bool operator() (const ListNode *a, const ListNode *b)
    {
        if (a->val < b->val)
            return false;
        else
            return true;
    }
};

ListNode *mergeKLists(vector<ListNode *> &lists) {
    priority_queue<ListNode *, vector<ListNode *>, cmp> heap;
    for (int i = 0; i < lists.size(); i++) {
        if (lists[i]) {
            // for the corner case: [{}]
            heap.push(lists[i]);
        }
    }
    ListNode *prevNode = NULL;
    ListNode *head = NULL;
    ListNode *curNode = NULL;
    while (!heap.empty()) {
        curNode = heap.top();
        heap.pop();
        if (head == NULL) {
            head = curNode;
        }
        if (curNode->next) {
            heap.push(curNode->next);
        }
        if (prevNode) {
            prevNode->next = curNode;
        }
        prevNode = curNode;
    }

    return head;
}
```

例题 4 Having a stream of integers, design a data structure to support easy look up the number of values less than or equal to a given value.

有一组整数，设计一个数据结构，支持很容易地查找出小于或等于一个给定值的整数的数目。

解题分析：*需要一个动态的数据结构，支持快速检索，并且满足一定的有序性，这样才能维护这个特殊属性（给定值在这个有序结构中的相对位置），使用二叉搜索树。*注意，没有其他数据结构例如哈希表的辅助，堆无法支持给定 key 的快速检索。

参考解答：

```
class Node{
    private:
        Node*trackhelp(Node*rt,int x);
    public:
        Node *left;
        Node *right;
        int key;
        int left_cnt;
        void track(int x);
        int getRank(int x);
        Node(int key,Node*l = NULL,Node*r =NULL,int cnt = 0):key(key),left
(l),right(r),left_cnt(cnt)
         {};
};

Node *Node::trackhelp(Node*rt, int x){
    if( rt == NULL) return new Node(x);
    if( x <= rt->key ){
        rt->left = trackhelp(rt->left,x);
        rt->left_cnt++;
    } else {
        rt->right = trackhelp(rt->right,x);
    }
    return rt;
}

void Node::track( int x){
     trackhelp(this,x);
}

int Node::getRank(int x){
    if(this == NULL) return -1;
```

```
    if(x == key)
       return left_cnt;
    if(x < key)
      return left->getRank(x);
    if(x > key){
        if(right->getRank(x) == -1) return -1;
        return left_cnt + 1 + right->getRank(x);
    }
}
```

9.2.2 对于有序/部分有序容器的搜索，用二分查找

例题 5 Find i in a given array that arr[i] == i.

在一个给定的数组中，找出符合 arr[i] == i 的 i。

解题分析：本题最直观的想法是线性扫描整个数组，逐一检查元素值与下标是否相同。这样做的时间复杂度是 O(n)，但是很明显的缺陷在于这种做法完全没有利用到数组是有序的特性。由于题目中出现了关键字“sorted”，结合 9.1.4 小节中提到的规律：*对于有序线性容器的搜索，二分查找或其变种基本上是解题的最佳方法*，我们需要尝试设计一种类似于二分查找的算法。通常，我们可以尝试自己举个实例，便于发现普遍规律性：

Index

0

1

2

3

4

5

6

7

8

Value

-7

-2

0

3

7

9

10

12

13

在此例中，A[3] = 3。同时，不难发现一个规律：A[3]左侧的数据满足 value < index，A[3]右侧的数据满足 value > index。反而言之，如果当前数据满足 value < index，则需要搜索的数据必然在其右侧，如果当前数据满足 value > index，则需要搜索的数据必然在其左侧。这样，实际上就可以利用二分查找：比较当前数据值和其下标，选择恰当的半边进行下一步搜索。

复杂度分析：时间复杂度同二分查找，为 O(logn)。

参考解答：

```
int indexSearch(int *array, int left, int right) {
    if (left > right) {
        // value not found
        return -1;
    }

    int mid = right - (right - left) / 2;
    if (array[mid] == mid) {
        return mid;
    } else if (array[mid] < mid) {
        return indexSearch(array, mid + 1, right);
    } else {
        return indexSearch(array, left, mid - 1);
    }
}
```

例题6 An array is sorted without duplicates. However, someone mysteriously shifted all the elements in this array (e.g. 1,2,3,4,5 -> 5,1,2,3,4). Implement a function to find an element in such array (return -1 if no such element).

一个数组经过排序而没有重复元素。然而，某人神秘地移动了数组中所有的元素，例如1,2,3,4,5 -> 5,1,2,3,4。实现一个函数，在这样一个数组中找出一个元素。如果没有这样的元素，返回-1。

解题分析：如果是在完全有序的数组中进行搜索，最优的解法无疑是二分查找。本题中，平移操作后的数组不再是完全有序了，因此我们不能直接应用二分查找。回顾二分查找算法：二分查找将容器等分为两部分，再根据中间节点与待搜索数据的相对大小关系，进一步搜索其中某一部分。算法本质在于，通过数据之间的相互比较，每次我们只需要搜索容器的某一半边。事实上，*对于本题而言，既然数据具备局部有序的特性，如果通过适当的条件判断，每次能够减半搜索范围，那么我们同样可以达到二分查找的效果*。关键问题在于：如何通过数据之间的相互比较，确定需要检索的那一半数据？

首先，我们可以通过一个实例观察平移后的数组有什么特性：假设数组 0,1,2,3,4,5,6,7,8,9,10 通过平移变为7,8,9,10,0,1,2,3,4,5,6，要求搜索4。根据二分查找的算法，我们将4与容器的中间元素进行比较，由此确定需要继续搜索的半边。本例中,中间元素为1,即将数组切割为7,8,9,10,0,1与1,2,3,4,5,6。不难发现，每次切割都保证至少有一半数组是完全有序的，并且，我

们可以通过比较切割后两半数组各自的头尾数据大小，确定哪一半是完全有序的。

现在，我们进一步考虑如何减半搜索范围。由于至少有一半数组是有序的（并且我们知道数据的范围），那么可能有两种情况：

（1）待搜索元素落在有序的那一半（即大于最左元素且小于最右元素）；

（2）待搜索元素不在有序的那一半。

对于情况（1），我们只需要搜索那半边即可。对于情况（2），我们只需搜索另一半边即可。这样，无论出现哪种情况，我们都可以减半搜索范围。

总结一下我们的算法：

（1）通过中间元素将数组划分为两个半边；

（2）通过比较切割后两半数组各自的头尾数据大小，确定哪一半是完全有序的；

（3）判断待搜索元素落在哪个半边，减半搜索范围。

复杂度分析：由于我们每次能够减半搜索范围，故时间复杂度与二分查找相同，为O(logn)。

参考解答：

```
int searchInRotatedArrayHelper(int array[], int left, int right, int target)
 {
   if (left > right) {
       // exit: target not found
       return -1;
   }

   int mid = right - (right - left) / 2;
   if (array[mid] == target) {
       // exit: target found
       return mid;
   }

   if (array[left] <= array[mid]) {
       // left half is completely sorted
       if (target >= array[left] && target <= array[mid]) {
           // target in the sorted part
           return searchInRotatedArrayHelper(array, left, mid - 1, target);
       } else {
           // target not in the sorted part
           return searchInRotatedArrayHelper(array, mid + 1, right, target);
```

```
        }
    } else {
        // right half is completely sorted
        if (target >= array[mid] && target <= array[right]) {
            // target in the sorted part
            return searchInRotatedArrayHelper(array, mid + 1, right, target);
        } else {
            // target not in the sorted part
            return searchInRotatedArrayHelper(array, left, mid - 1, target);
        }
    }
}
int searchInRotatedArray(int array[], int n, int target) {
    return searchInRotatedArrayHelper(array, 0, n - 1, target);
}
```

例题 7 Given a sorted array of integers with duplicates.Implement a function to get the start and end position of a given value.

给定一个排序的整数数组，且没有重复值。实现一个函数，获取一个给定的值的开始和结束位置。

解题分析：*对于完全排序数组的搜索问题，首先应该想到二分查找：我们可以通过二分查找找到相应的元素。*其次，题目要求返回该元素的起始和终止位置。那么，我们可以基于二分查找返回的结果，向左向右依次做线性扩展，即查看下一个元素是否依然符合条件。这样做可以得到正确的结果,但是在最坏情况下,该算法复杂度为O(n)。例如，数组为 1,1,1,1,1，给定的元素也为 1。那么，在我们做线性扩展的时候，我们会遍历数组中的每一个元素。

如何效率更高地找到元素的起始和终止位置？考虑到数组是完全排序的，*即被目标值分割的左右半边仍然分别有序，满足局部有序的特征，于是我们可以进一步继续做二分查找*：即对左右两个区间分别继续搜索目标元素，在这个过程中，更新目标值出现的最左位置和最右位置。这样，我们可以以O(logn)的复杂度快速获得起始和终止位置。

复杂度分析：始终在做二分查找，没有线性查找，因此平均时间复杂度是O(logn)。

参考解答：

```
void searchRangeHelper(int array[], int left, int right, int target, int &be
gin, int &end) {
    if (left > right) {
        return;
    }

    int mid = right - (right - left) / 2;
    if (array[mid] == target) {
        if (mid < begin || begin == -1) {
            begin = mid;
        }
        if (mid > end) {
            end = mid;
        }
        searchRangeHelper(array, left, mid - 1, target, begin, end);
        searchRangeHelper(array, mid + 1, right, target, begin, end);
    }
    else if (array[mid] < target) {
        searchRangeHelper(array, mid + 1, right, target, begin, end);
    }
    else {
        searchRangeHelper(array, left, mid - 1, target, begin, end);
    }
}

vector<int> searchRange(int A[], int n, int target) {
    int begin = -1, end = -1;
    searchRangeHelper(A, 0, n - 1, target, begin, end);
    vector<int> ans;
    ans.push_back(begin);
    ans.push_back(end);
    return ans;
}
```

例题 8 Check if an element is in a M x N matrix, each row and column of which is sorted.

检查一个元素是否在一个 M x N 矩形中，矩形的每一行和每一列都是排序的。

解题分析：首先我们可以构造一个矩阵。

1	5	10	20
2	6	11	30
7	9	12	40

8　　15　　31　　41

如果要在上述矩阵中找到 9，应该如何计算？最简单的方法显然是遍历每行每列，这样的时间复杂度是 O(n^2)，而且完全没有利用到矩阵已经部分有序的特性。

进一步观察矩阵，任何元素都将矩阵划分为 4 个部分：

Ⅰ

Ⅱ

Ⅲ

Ⅳ

根据矩阵的特性，同行同列中元素的大小关系已知，并且Ⅰ区的所有数据都比当前元素小，Ⅳ区的所有数据都比当前元素大。Ⅱ和Ⅲ两区数据与当前元素没有明确的相对大小关系。因此，我们的每次操作必须保证没有Ⅱ区或Ⅲ区，即从右上角或者左下角开始搜索。不妨假设从右上角（20）开始搜索：

（1）比较 20 与 9，左移。

（2）比较 10 与 9，左移。

（3）比较 5 与 9，下移。

（4）比较 6 与 9，下移。

（5）找到 9。

不难发现，每次当前元素大于待搜索元素，我们左移，否则下移。

对于有序容器的搜索，能不能用二分查找？对于本例，我们不能使用二分查找及其变种。原因是：二分查找的关键在于，当前元素将容器分为两个部分，并且通过比较当前元素和待搜索元素的大小，我们能够确定两者的相对位置关系，进而缩小搜索范围。*但是对于本例，当前元素和待搜索元素大小关系并不能确定两者相对位置。*

复杂度分析：假设矩阵有 M 行 N 列，则我们至多下移 M 次，左移 N 次，即算法复杂度为 O(M+N)。

参考解答：

```
bool isElementInMatrix(int **matrix, int M, int N, int target) {
    int row = 0;
    int column = N - 1;
    while (row < M && column >= 0) {
        if (matrix[row][column] == target) {
            return true;
        } else if (matrix[row][column] < target) {
            row++;
        } else {
            column--;
        }
    }
    return false;
}
```

例题 9 Implement sqrt(x), which returns the square root of value x.

实现 sqrt(x)，它返回值 x 的平方根。

解题分析：首先我们需要明确开根号的性质：

（1）负数无效。

（2）若 x 为 0，则返回 0。

（3）若 x 属于（0,1），则 sqrt(x)属于（x,1）。

（4）若 x 为 1，则返回 1。

（5）若 x 大于 1，则 sqrt(x)属于（1,x）。

此外，若 x ＞ y，则 sqrt(x) ＞ sqrt(y)。情况 1，2，4 可以作为特例，而对于通常情况（情况 3，情况 5），我们发现如下两个特性：

（1）解落在已知区间。

（2）存在相对大小关系。

进一步地，我们可以将 sqrt(x)的所有“候选数”看成是分布在有限区间上的有序数列，对于每个元素，我们通过平方操作比较与待搜索数 x 的相对大小关系。很明显，这就是二分查找的思想。

复杂度分析：由于我们利用了二分查找的思想，故复杂度为 O(log(x/precision))。

参考解答：

```
Double mySqrtHelper(double x, double lowBound, double highBound) {
    double precision = 0.00001;
    double sqrt = lowBound / 2 + highBound / 2;
    if (abs(sqrt * sqrt - x) <  precision) {
        return sqrt;
    } else if (sqrt * sqrt - x > 0) {
        return mySqrtHelper(x, lowBound, sqrt);
    } else {
        return mySqrtHelper(x, sqrt, highBound);
    }
}

double mySqrt(double x) {
    if (x < 0)
        return ERROR;
    if (x == 0) {
        return 0;
    }
    if (x == 1) {
        return 1;
    }
    if (x < 1) {
        return mySqrtHelper(x, x, 1);
    } else {
        return mySqrtHelper(x, 1, x);
    }
}
```

例题10 Find the median of the elements of two sorted arrays.

找出两个排序的数组的元素的中间位置元素。

解题分析：对于含有 n 个数的数组，若 n 为奇数，则中位数 array[n/2+1]，若 n 为偶数，则中位数为 (array[n/2] + array[n/2+1]) / 2。所以我们当前的目标是设计一种算法能够返回第 k 大的元素。对于查找第 k 大的数，最先想到的应该是快速选择算法，但是该算法并没有利用数组已经有序的特性，故时间复杂度为 O(m+n)。*与二分查找相似，快速选择算法的核心在于快速缩小搜索范围。*

_对于本例，我们应该如何缩小搜索范围呢？假设数组分别记为 A，B。当前需要搜索第 k 大的数，于是我们可以考虑从数组 A 中取出

前 m 个元素，从数组 B 中取出 k-m 个元素。由于数组 A，B 分别排序，则 A[m]大于从数组 A 中取出的其他所有元素，B[k-m] 大于数组 B 中取出的其他所有元素。此时，尽管取出元素之间的相对大小关系不确定，但 A[m]与 B[k-m]的较大者一定是这 k 个元素中最大的。那么，较小的那个元素一定不是第 k 大的，它至多是第 k-1 大的：因为它小于其他未被取出的所有元素，并且小于取出的 k 个元素中最大的那个。为叙述方便，假设 A[m]是较小的那个元素。那么，我们可以进一步说，A[1], A[2]…A[m-1]也一定不是第 k 大的元素，因为它们小于 A[m]，而 A[m]至多是第 k-1 大的。因此，我们可以把较小元素所在数组中选出的所有元素统统排除，并且相应地减少 k 值。这样，我们就完成了一次范围缩小。特别地，我们可以选取 m=k/2。

复杂度分析：每次缩小范围之后 k 值基本上折半，故时间复杂度为 O(logn)。

参考解答：

```
double helper(int A[], int m, int B[], int n, int k){
    // find the kth largest element
    if(m > n)
        return helper(B, n, A, m, k);//make sure that the second one is the b
igger array;
    if(m == 0)
        return B[k - 1];
    if(k == 1){
        return min(A[0], B[0]);
    }
    int pa = min(k / 2, m); // assign k / 2 to each of the array and cut the
smaller one
    int pb = k - pa;
    if (A[pa-1] <= B[pb-1])
        return helper(A + pa, m - pa, B, n, k - pa);
    return helper(A, m, B + pb, n - pb, k - pb);
}
double findMedianSortedArrays(int A[], int m, int B[], int n) {
    int total = m + n;
    if(total % 2 == 0){
        return (helper(A, m, B, n, total / 2) + helper(A, m, B, n, total / 2
+ 1)) / 2;
    }
    return helper(A, m, B, n, total / 2 + 1);
}
```

例题 11 Given a server that has requests coming in. Design a data structure such that you can fetch the count of the number requests in the last second, minute and hour.

给定一个有入向请求的服务器。设计一种数据结构，以便可以获取最后一秒钟、一分钟和一小时的请求的数目。

解题分析：对于这个问题，比较容易想到的是对于每个请求，*我们都分配一个当时的时间戳*（timestamp），并且将请求根据时间戳排序，则时间戳构成一个完全有序的序列。这样，当我们要搜索一分钟前的请求时，可以用二分查找快速定位。其次，如果我们找到了对应的*请求*，如何最快速地知道从这个请求至现在一共发生了多少次其他请求？我们可以采用计数的方式，对于每个请求，分配一个计数，用以表示这是第几个请求。我们只需要用当前计数减去某个请求的计数，就可以知道从那个请求至现在一共发生了多少次其他请求。

复杂度分析：我们应用二分查找寻找某个特定时间的请求，算法涉及的时间复杂度为 O(logn)。

参考解答：

注意，我们简化了越界的处理，采用 long int 并且默认整形数据不会越界。读者可以进一步考虑如何处理越界。

```
#include <time.h>
#include <sys/time.h>
long now() {
    struct timeval time;
    gettimeofday(&time, NULL);
    return (time.tv_sec * 1000000 + time.tv_usec);
}
class HitCounter {
private:
    deque<pair<long, int>> hits;
    long last_count = 0;
    const int second = 1000000;
    const int minute = 60 * second;
    const int hour = 60 * minute;
    void prune() {
        auto old = upper_bound(hits.begin(), hits.end(), make_pair(now() - 1
 * hour, -1));
        if (old != hits.end()) {
            hits.erase(hits.begin(), old);
        }
    }
```

```
public:
    void hit() {
        hits.push_back(make_pair(now(), ++last_count));
        prune();
    }

    long hitsInLastSecond() {
        auto before = lower_bound(hits.begin(), hits.end(), make_pair(now()
- 1 * second, -1));
        if (before == hits.end()) { return 0; }
        return last_count - before->second + 1;
    }

    long hitsInLastMinute() {
        auto before = lower_bound(hits.begin(), hits.end(), make_pair(now()
- 1 * minute, -1));
        if (before == hits.end()) { return 0; }
        return last_count - before->second + 1;
    }

    long hitsInLastHour() {
        auto before = lower_bound(hits.begin(), hits.end(), make_pair(now()
- 1 * hour, -1));
        if (before == hits.end()) { return 0; }
        return last_count - before->second + 1;
    }
};
```

9.2.3 数据范围有限、离散的排序问题

数据范围有限、离散（或存在大量重复数据，即密集数据）的排序问题，一般可以使用桶排序。对于有限位数的数据（如string, vector<int>, int），可以利用基数排序进行数值序或词典序排序。

例题 12 Sort a large number of people by their ages.

根据年龄对一大批人排序。

解题分析：*人的寿命是有限的，即数据都处于[0,150]。在数据最大值已知的情况下，通常***桶排序***效率最高，为O(n)。*对于本例，由于数据为十进制数，并且至多 3 位。故我们还可以考虑用基数排序，时间复杂度与桶排序近似。

复杂度分析：同桶排序，为O(n)。

参考解答：请参考桶排序的代码实现。

例题13 Reorder an array of strings so that anagrams appear together.

对一个字符串的数组重新排序，以便相同字母异序词相邻。

解题分析：*这并不是一个完全排序的问题，而只要求分类，并且类别一定是有限个，选择用***桶排序***，类别数量等于***桶的***数量。*对一般的桶排序，每一个不同的值就对应一个桶，而这里则是所有相同字母异序词（anagram）对应一个桶，因此需要找到一个哈希函数，要求是所有相同字母异序词的哈希值相同，对应同一个桶。一种做法是将字符串排序后的结果作为字符串的哈希值。

复杂度分析：假定字符串平均长度为k，数量为n，那么平均复杂度为O(klogk*n)。

参考解答：

```
void reOrderWithAnagrams( vector<string> strs ) {
    unordered_map<string, list<string>> buckets;
    for (auto it = strs.begin(); it != strs.end(); it++) {
       string curr_str = *it;
       sort(curr_str.begin(), curr_str.end());
       buckets[curr_str].push_back(*it); // Add this string to corresponding
 bucket
    }
    // Post processing, reorder the strings with the bucket
    int index = 0;
    for (auto it = buckets.begin(); it != buckets.end(); ++it) {
        list<string> &anagrams = it->second;
        for (auto local_it = anagrams.begin(); local_it != anagrams.end(); +
+local_it){
            strs[index] = *local_it;
            index++;
        }
    }
    return;
}
```

9.2.4 Scalability & Memory Limits 问题

对这类问题一般采用 Divide & Conquer 策略，即对问题进行预处理，将问题的输入进行分割、归类（sorting），放入相应的桶（单机上的某一块 Chunk，或者分布式系统中的一台单机），再对每个桶进行后期处理，最后合并结果。

整个过程中应该用到哈希函数：对于 Memory Limits 问题，一般可以直接利用哈希函数建立对象到索引的直接映射；对 Scalability 问题，一般可以用哈希表来记录对象与存储该对象的机器之间的映射，在该机器上进一步做映射以获得索引。

例题 14 A library is trying to build up a smart computer-aided look up system: user may input a list of key words, and the system shall provide all books that contain these words. How to implement such query? (A library may have millions of books)

一家图书馆试图构建一个智能化的计算机辅助查找系统：用户只需要输入一个关键词列表，系统将提供包含这些关键词的所有图书。如何实现这样的查询呢？图书馆可能有数以百万计的图书。

解题分析：对于文章中单词的检索，一般采用倒排索引（Inverted index），其本质就是单词到文档集合的一个哈希，例如：

```
apple -> doc 1, doc 2
banana -> doc 2
```

对于文档集合，可以进一步利用各种集合操作，例如交集、并集等，获得对应的结果。如果题目中书本数目不大，并且出现的单词总个数不多，那么可以直接将倒排索引建立在一部机器上。在查询时，对于用户给定的每个单词，通过哈希表查找出对应的文档集合，最后对所有集合求交集，即可实现查询功能。但是，当文档/单词量巨大时，如何实现可扩展性？*我们需要以一定的规律将哈希表分配到多台机器上，即完成对哈希表的拆分和分块存储。*例如，machine1 可以负责首字母为 a 的所有单词，machine2 可以负责首字母为 b 的单词，以此类推。事实上，这个过程相当于引入另一层 hash 进行数据分流：建立对象与存储该对象的机器之间的映射。

例题 15 Design a scalable social network system and support looking up connection between two users.

设计一个可扩展的社交网络系统，并且支持查找两个用户之间的

联系。

解题分析：与上题类似，首先我们考虑数据量不大的情况，即所用用户数据存储在同一部机器上。要查询两个用户之间的联系无非就是广度优先搜索（请回顾第 5 章）。进一步，如果数据量很大，那么我们需要用多台机器存储对象。不妨采用与上题类似的思路：引入另一层哈希进行数据分流，建立对象与存储该对象的机器之间的映射。通常用户都有一个唯一的 ID，我们可以利用 ID 的前几位数字，决定将该用户数据存放在哪台机器上。这样，广度搜索的查询流程变为：

（1）获得用户的好友 ID 列表。

（2）根据 ID 获得机器。

（3）访问机器获得对应的用户节点。

（4）进行下一步递归。

应用该方法我们需要注意一下问题：

（1）访问另一台机器的成本可能比较高，因此应该尽量把该机器上的所有相关数据一次性读取完毕。

（2）广度搜索需要记录某个节点是否已经访问过以避免循环访问，我们可以另外开辟一个哈希表记录已经访问的用户节点。

（3）注意合理地简化问题：如果两个用户之间需要多于 3 重关系才能联系上，我们可以把他们归类为“陌生人”。那么，我们可以将关系链限制在 3 重关系以内，以减少搜索成本。

例题 16 Suppose you are given an extremely large set of URLs,~ billion for example. How do you de-duplicate(remove duplicates)?

假设给定很大的一个 URL 的集合，例如，数量达到十亿级别。如何去除重复的内容？

解题分析：对于本题，我们还是套用惯用的模版：

1. 先考虑小数据量的情况使用哈希表。

2. 数据量很大使用多台机器，引入另一层 hash 进行数据分流。

分流方式可以根据机器数量，采用 URL 的前若干个字符进行数据到机器号的映射。

9.3 工具箱

正文 1. 合并排序

关于合并排序的理论分析，请见参考资料：

《算法导论》(Introduction to Algorithms，2nd Edition)，Thomas H. Corman, Charles E.Leiserson, Ronald L.Rivest, Clifford Stein，第 2 章，算法入门，2.3.1 分治法。

2. 快速排序

关于快速排序的理论分析，请见参考资料：

《算法导论》（Introduction to Algorithms, 2nd Edition），Thomas H. Corman, Charles E.Leiserson, Ronald L.Rivest, Clifford Stein，第 7 章，快速排序。

3. 快速选择算法（quick selection algorithm）

关于快速选择算法的理论分析，请见参考资料：

《算法导论》(Introduction to Algorithms，2nd Edition)，Thomas H. Corman, Charles E.Leiserson, Ronald L.Rivest, Clifford Stein，第 9 章，中位数和顺序统计学。

4. 二分查找（binary search）

关于二分查找的理论分析，请见参考资料：

《算法导论》(Introduction to Algorithms, 2nd Edition)，Thomas H. Corman, Charles E.Leiserson, Ronald L.Rivest, Clifford Stein, 第 12 章，二叉查找树。

5. 动态数据结构的维护

本章涉及的排序与搜索通常可以由如下数据结构实现，我们在此做一下比较。

二叉查找树（Binary Search Tree，BST)：可以看成一个动态的有序数组。建立 BST（对应 std::map ）通常可以满足有序要求，将检索对象作为树节点的 key，将检索内容作为树节点的 value。利用 BST 本身的性质，在插入/删除时都能够维持其有序的特性。此外，BST 的节点可以方便地记录其子树的信息，并在节点间传递全局信息，方便检索。当 BST 反复插入/删除时，可能导致树的不平衡，影响检索效率。此时，应该使用更为高级的平衡二叉查找树或者红黑树来维护动态数据。

堆：能够很方便地维护动态数据的最大值、最小值或中位数。priority_queue 相比简单的队列更适用于需要优先级的情境。与队列类似，该结构适合 push 和 pop，但不方便更新和检索（仅仅部分有序）。

链表：适合快速的更新、插入和删除，但不方便检索。可以注意

到链表同时也可以是队列的底层实现方式，因此适合使用队列的情境，一般也可以使用更为强大的链表。

哈希表：为了实现更快速的非排序检索(O(1))，如 unordered_map 以及 array、set、bitset<int>。

6. 倒排索引（Inverted index）

http://en.wikipedia.org/wiki/Inverted_index

第 10 章 测试

10.1 知识要点

在面试软件开发的过程中，面试官可能也会询问关于软件开发流程以及测试方法相关的问题。在大多数互联网公司，许多部门不一定配有专门的 QA（Quality Assurance），在这种情况下，程序员需要对自己开发的模块和系统进行测试。另一方面，程序员在开发过程中测试自己的程序也是非常好的习惯，这样可以确保开发效率。基于上述原因，面试软件开发职位但遇到测试相关的问题并不少见。本章节总结了一些常见的测试相关问题及解题方法，并且在 10.3“工具箱”部分列举了测试相关的常见概念。如果需要面试测试相关的专门职位，建议查阅更多相关资料。

10.1.1 测试现实世界的物体、软件或函数

三者并无本质的差别，问题的核心均在于：测试对象在不同的输入下，能否实现预计的功能，提供恰当的输出。 一般情况下，总是需要考虑以下几个方面，以全面测试对象对于不同类型输入的“效果”。

1. 常规情况（Normal cases）

输入不同类型的合法数据，主要用以判断对象的功能性：在给定输入的情况下能否给出期望的输出，由此判断功能的实现是否正确。比如，测试银行账户的转账功能：假设账户中有 1000 元，可以输入 100、2000 等并判断余额及转出钱数是否符合期望。

2. 极端情况（Extreme cases）

测试一些边界条件或极端情况。所谓的极端情况包括多用户或多线程情况下频繁地访问/更新数据。比如，继续测试银行账户的转账功能：假设账户中有 1000 元，可以测试边界条件，取出 1000 元等。或者测试极端情况，假设用户开了多个页面，并在每个页面上几乎同时都尝试转出 1000 元，或者用户通过 ATM 机和手机 APP 同时进行转账操作等。

3. 非法情况（Invalid case）

主要测试用户输入非法数据时系统不会崩溃，并且能够给出恰当的反馈。比如，测试银行账户的转账功能：当用户输入大于账户余额的数字时，或者当接收人账户错误时，系统能否给出错误提示等。

10.1.2 故障排除

另一大类的常见问题是给出一个有问题的测试现象，让面试者判断问题出现在哪里。对于这类问题，首先考虑测试对象由生成，到运行，到产生最终结果的完整流程，其次判断每一步执行了什么，需要依赖哪些参数，该步骤的异常是否会导致最终的测试现象，并且考虑如何验证自己的判断。 例如，测试用户无法访问你开发的网站。首先考虑主要流程，简述如下：用户连接到网络，发送 HTTP 请求到网站，网站发送数据包给用户，用户浏览器显示页面。在此例中，每一步都有可能导致无法访问网站的情况，具体描述如下。

（1）用户连接到网络：这一步用户需要获得有效的 IP，获取访问互联网的权限。需要依赖用户的网卡是否工作正常，是否能够被分配到有效的 IP，是否能够从路由器或者服务器获得互联网访问权限等等。检验方式可以是：可以打开终端用 ping 命令，尝试建立与大型网站的连接；或者直接用浏览器尝试访问其他大型网站。如果不能建立与其他网站的连接，则网络接入有问题。

（2）发送 HTTP 请求到网站：用户首先会通过 DNS 获取服务器地址，然后发送 HTTP 请求到对应的 IP。需要依赖用户能否正确获取网站 IP 地址。检验方式可以是：在用户端利用抓包软件，例如 WireShark、tcpdump 等，观察是否有 HTTP 请求发送到网站服务器。如果没有发送 HTTP 请求或目的地 IP 有问题，则 DNS 可能有错。

（3）网站发送数据包给用户：这一步需要网站接收到 HTTP 请求，并且将对应数据传回给用户。需要依赖网站能否收到 HTTP 请求以及对于 HTTP 请求的处理是否正确。检验方式可以是：在服务器端通过

日志判断是否有新用户接入，接入请求的处理是否正确，以及发送给用户的数据是什么。如果网站没有收到请求，则服务器端的网络可能有问题。如果服务器无法处理 HTTP 请求或抛出异常，则服务器的实现可能有问题。

（4）用户浏览器显示页面：这一步需要用户接收到网站发回的数据，浏览器解析数据并显示页面。需要依赖于用户能否收到数据，以及收到的数据是否能够被浏览器正确解析及显示。检验方式可以是：在用户端利用抓包软件，观察是否有来自服务器的数据。一般来说，如果用户用的是商用浏览器，即能够正确解析数据。故如果能收到服务器数据但是不能正常显示，我们可以认为服务器的数据有问题。

10.2 模式识别

例题 1 How do you test a login system (with user ID and password)?

如何测试一个登录系统。登录时系统需要用户 ID 和密码。

解题分析：本题属于“测试现实世界的软件”，我们按照 10.1 节给出的模版进行分析：

1．常规情况

正常情况下，用户输入正确的用户名和密码，系统就必须允许用户访问。如果用户名或密码有误，系统应该拒绝用户访问。这是最基本的功能，应该归类于正常情况的测试范畴。具体测试时，可以利用脚本生成许多用户，然后利用用户名和密码测试登录。

2．极端情况

在极端情况下，需要考虑某用户同时登录的情况，例如，用户打开了多个网页几乎同时尝试登录，或者用户从网页或手机 App 同时尝试登录。系统需要考虑：从安全性而言，是否能够允许用户多处登录。如果不允许，应该采用什么机制避免这样的情况。同时，在极端情况下，系统必须考虑用户多次尝试密码，但无法成功的情况。系统必须阻止类似行为，防止黑客采用暴力破解的方式获取用户密码。

3．非法情况

用户名可能是邮箱形式，如果用户输入格式不对，可以提示用户输入邮箱。此外，还应该考虑用户恶意输入超长字符串的情况，这样可能造成栈溢出，对后台产生影响。其次，应当检测用户名密码是以何种方式传到服务器端的。这里需要考虑用户账号密码的安全性，还

需要考虑到如果用户名直接以明文方式内嵌在 URL 中跳转到对应页面，用户可能特意输入一个带有特殊意义的用户名，使得服务器在解析 URL 时跳转到一个不应该出现的页面。

例题 2 How do you test a pen?

如何测试一支钢笔

解题分析：本题属于测试现实世界的物体，我们按照 10.1.1 小节给出的模版进行分析。

进一步的信息

本题和之前的问题有一点区别：笔的范围太过宽泛，不同笔的功能也不一样。因此，在开始解答之前要与面试官进行沟通，获得更多的信息。例如，询问这是什么类型的笔，面对的用户是谁，有没有特殊的适用场合等等。假设我们需要测试一只普通的圆珠笔。

1．常规情况

普通测试包括外观，墨水颜色是否纯正，各个部件是否完整等。可以在纸上测试书写效果，有没有漏墨，出墨不匀等情况。写字的手感是否顺滑，字迹能否很快变干也是影响用户体验的关键因素。

2．极端情况

从正常使用的角度而言，极端情况包括用户能否把所有的墨水顺利写完。其他使用的极限情况包括：圆珠笔能否在高温或低温环境下正常工作，比如温度低于零度时，是否还能正常书写；或者用户意外将笔从高处跌落，笔尖着地后还能否正常书写等。

例题 3 How do you test your implementation for “int binary_search(vector<int> array, int value)”, which returns index of the value in array or -1 if not found?

如何测试你的“int binary_search(vector<int> array, int value)”实现，它返回该值的索引，如果没有找到的话返回-1。

解题分析：本题属于“测试现实世界的一个函数”，我们按照 10.1.1 小节给出的模版进行分析：

1．常规情况

对于函数或软件的测试，通常采用脚本测试的方法。特别地，对于比较常见的函数功能，例如本题的二叉搜索，可以尝试寻找开源的函数与自己的实现进行比较。或者实现一个简单的线性搜索，比较查询结果是否一致。有了验证结果的方法，则可以编写脚本生成随机数据，同时运行需要测试的函数以及对照函数，比较运行结果是否相同。

正常测试情况包括从数组中找出某个数据，以及测试数组中不存在该数据的情况。

2．极端情况

极端情况包括输入数组为空，或输入数组很大的情况。

3．非法情况

需要考虑数组中含有重复元素是否是有效的情况。二叉搜索在这样的情况下返回的元素下标不一定是该元素第一次出现的下标（参见第 9 章例题 6）。

例题 4 You implemented a mobile web browser, but it crashes sometimes when browsing. How to debug it?

你实现了一款移动 Web 浏览器，但是，有时候浏览的时候，它会崩溃。如何修复它？

解题分析：该问题属于故障排除，我们按照 10.1.1 小节给出的模版进行分析。首先考虑主要流程，简述如下：用户打开应用，连接到网页，接收数据，用 JavaScript 及 Flash 等插件解析网页，显示内容。由于并非每次浏览都会崩溃，因此可以假设打开应用，连接到网页，接收数据不是问题的关键。于是我们进一步考虑解析网页，显示内容这两步执行了什么，需要依赖哪些参数。

（1）解析网页

这一步需要分析服务器端传入的 HTML 数据，并且调用 JavaScript、Flash 等插件显示网页的动态部分。该过程依赖于 HTML 解释器及各个插件是否工作正常，如果缺少插件并没有恰当地处理异常，可能导致应用崩溃。检验的方法是需要研究崩溃现象是否和访问的具体网页有关，如果每次访问某个网页都会崩溃，则需要利用其他浏览器进一步研究解析该网页的过程中调用到了哪些模块，然后采用排除法确定待测试浏览器的哪个模块出现了问题。

（2）显示内容

浏览网页的最后一步是将数据，图片和动态元素显示在页面上。考虑到这是手机端的浏览器，移动平台本身的计算资源比较有限，因此，内存可能成为瓶颈，并导致应用崩溃。该情况较容易出现在同时打开多个页面的情况。检验的方式可以是：同时访问多个大型门户网站，观察是否会造成应用崩溃。解决方式可以是：仅将用户正在浏览的页面内容读入内存，其他页面缓存至硬盘。并且可以降低图片分辨率，以减少内存消耗。

10.3 工具箱

1. GNU 调试器命令

（1）Compile with GDB support: gcc -g prog.c -o prog.x;

（2）Start debugging: gdb prog.x , (gdb) run;

（3）Handle breakpoints: (gdb) break prog.c: 6 or (gdb) break my_func; if, continue, step, next; delete;

（4）Watch variable: (gdb) print my_var, (gdb) watch my_var, (gdb) x (address of var);

（5）Other utilities: (gdb) backtrace, (gdb) finish.

2. 测试的方法

AB 测试

AB 测试是一种对比测试方案。测试人员对于不同用户随机生成两种方案，例如，某些用户看到的网页按钮是圆形的，其他用户看到的网页按钮是方形的。通过用户对于不同测试方案的反应，来决定最终部署哪种方案。具体请参考：

http://en.wikipedia.org/wiki/A/B_testing

黑箱测试

黑箱测试（Black Box Testing）主要用于测试程序的功能，而不是内部结构或运作。测试者秩序知道输入以及对应的输出，就可以生成测试数据。黑箱测试的目的在于快速检测程序的功能性。特别地，黑箱测试还应该包括非法的输入数据，以确保程序不会崩溃。

白箱测试

与黑箱测试相对，白箱测试（White Box Testing）主要用于测试程序的内部结构或运作。测试人员需要从程序设计的角度生成测试案例：输入测试数据并验证程序按照既定的流程执行。

3. 工业界测试流程

单元测试（Unit Test）

优良的软件设计强调模块化，即模块之间通过 API 进行交互，每个模块负责实现相对独立的功能。单元测试的目的在于对于每个模块设计相应的测试数据，用以检验模块的功能。通常，单元测试采用黑箱测试，通过运行脚本完成。测试人员将测试数据输入脚本，将输出结果与期望的输出数据进行比较。单元测试不仅仅可以用于新模块的开发，还可以用于对于已有模块的更新、维护。对于模块的每次更改

都应该运行相应的单元测试以确保功能的完整性。

Alpha 测试

Alpha 测试通常是阶段性开发完成后开始进行。主要是面向内部开发人员，在模拟环境中输入模拟的数据进行测试，以验证系统符合使用者以及设计者的需求。

Beta 测试

当 Alpha 阶段完成后，可以进入由公众参与的 Beta 测试阶段。Beta 测试通常使用真实的运行环境，并且使用实际数据进行测试，以确认系统效率。测试的主要目的在于进一步测试及完善功能。

第 11 章 网络

11.1 知识要点

计算机网络是计算机科学的一个重要组成部分。特别是在当前互联网、移动互联网的浪潮之下，几乎所有的服务都需要通过网络实现，并且后台的分布式数据库也通过网络进行互联。因此，了解一些计算机网络相关的概念不仅仅有利于技术面试，也可以对整体行业有更好的理解。

对于初级的程序员面试而言，除非是网络相关的职位面试，否则面试的问题基本停留于一些常见概念。例如，TCP 和 UDP 的区别，TCP 的基本功能，网络的层次等。对于路由器生产商，例如 Cisco、Arista、华为等，需要着重准备路由相关的问题；对于即时通信的职位，由于该应用对于性能要求较高，故基本需要用 C 语言实现，对应地，你需要着重准备 socket programing；对于互联网公司，通常公司的开发环境会抽象网络层的具体实现，并提供 API，故你只需要对于网络有一个基本概念即可。

11.1.1 网络分层

计算机之间的交互模型通常是指 Open Systems Interconnection Model （OSI），该模型将网络通信系统抽象成了七层，简要说明如下，具体请参考工具箱给出的链接：

分层

数据单位
功 能
示例协议
应用层
数据
端到端的高层级 API
HTTP、FTP、SMTP
表现层
网络服务和应用之间的数据传输,包括编码/解码、加密/解密、压缩/解压缩
ASCII、JPEG、H264
会话层
在两个节点之间建立、管理和终止通信
RPC、PAP
传输层
段
网络上两个点之间可靠的数据段传送，包括分段、确认和复用
TCP、UDP
网络层
包
结构化和管理一个多节点网络，包括寻址、路由和流量控制
lPv4、IPv6、BGP
数据链路层
位/帧
通过物理层连接的两个节点之间的可靠的数据帧传输
IEEE 802.2
物理层
位
一个物理媒介上的原始位数据流的调制、传送和接收
Ethemet、Wi-Fi、Bluetooth、fiber
###

11.1.2 路由

从用户角度来看，路由（Routing）是指将数据从一个用户终端，通过网络节点（例如路由器，交换机等），发送到另一个用户节点的过程。理论上说，对于一个拥有多个节点的拓扑网络而言，路由是指在 Network Layer（OSI Model 的第三层），将数据包（data packet）从一个节点以最优路径发送到目标节点的实现方法。其核心包括：如何获得邻近节点的信息，如何估计链路质量，如何寻址，如何构建网络拓扑结构等。通过路由器之间的路由协议（routing protocol），可以实现两个网络节点之间信息（包括网络域名、邻近节点、链路质量等）的交换和散布，通过不断重复该过程，每个节点都会获得足够多关于所在网络的拓扑信息。当有数据包需要传送时，路由器再通过路由算法（routing algorithm）计算传递当前数据包的最优路径，并把数据包发送给下一个邻近节点。许多路由算法基于图理论，实现了最小生成树、最短路径等经典的拓扑算法。关于路由算法的进一步讨论，请参考 11.3“工具箱”中提到的参考教材。

网络中，所谓的地址是指 IP 地址，IPv4 规定利用 32 位作为 IP 地址。但随着网络设备的增多，IPv4 已经不能满足人们的需求，故互联网逐渐向 IPv6 进行演进，IPv6 利用 128 位作为 IP 地址。

事实上，直观而言，网络路由的过程就相当于传统意义上的邮包寄送，IP 地址可以类比为邮政编码，路由器就相当于邮局，通过目的地邮政编码与邮局系统中的递送路径进行比较，由此确定下一步应该把当前包裹传递到哪里。

11.1.3 常用网络统计指标

衡量网络质量通常有下面两个指标：

1. 带宽/速率（Bandwidth/Rate）

带宽是指一个网络节点能以多快的速度将数据接收/发送出去，单位是 bits per second（bit/s）。对于对实时性要求不高的数据，例如下载等，带宽是影响用户体验的主要因素。两个终端节点之间的带宽由路径中所有节点的最小带宽决定。同时，终端的数据发送速度不应该超过当前的上载带宽，否则会对网络造成压力导致拥堵（congestion）。

2. One-way Delay / Round Trip Time (RTT)

One-way Delay 用以衡量网络的延迟。假设在时间点 A 从一个节点发送数据到另一个节点，目的地节点在时间点 B 收到数据，则两个时间点之差即为 One-way Delay。类似地，RTT 则是数据完成一个 Round Trip 回到始发节点的时间差，一般 RTT 可以近似估计为 One-way Delay 的两倍。对于网络会议、IP 电话等，延迟是影响用户体验的主要因素。延迟可能是由于网络中某个节点处理数据速度慢，突然有大规模数据需要传输，或者某条链路不断重传数据造成的。延迟与带宽有一定的相关性，但没有必然联系：可以类比某个路口，假设每秒可以有一辆车通过该路口，但现在突然来了 100 辆车，路口的通过效率并没有变化（即带宽不变），但每辆车通过路口的等待时间却变长了（延迟增加）。

11.1.4 TCP vs. UDP

在传输层，数据流被分块传输。最常见的传输协议是 TCP 和 UDP。两种协议的区别、优劣，通常是网络相关面试的重点。建议着重准备。

1. 传输控制协议（Transmission Control Protocol，TCP）

可靠协议

TCP 是一种可靠的传输控制协议，即在网络条件正常的情况下，TCP 协议能够保证接收端收到所有数据，并且接收到的数据顺序与发送端一致。TCP 通过在发送端给每个数据包分配单调递增的序列号，以及在接收端发送 ACK（acknowledgement）实现可靠传输。每个发送的数据包都包含序列号，当接收端收到数据包时，会发送 ACK 告诉发送端当前自己期待的下一个序列号是多少。例如，发送端分别发送了序列号为 99、100、101、102 的 4 个数据包，接收端收到数据包 99 后，会发送 ACK100，意味着接收端期待下一个数据包编号 100。如果由于某些原因，数据包 100 没有到达接收端，但数据包 101、102 到达了，那么接收端会继续发送 ACK100。当发送端发现当前发送的数据包编号超过了 100，但接收端仍然期望收到 100，那么发送端就会重新发送数据包 100。如果接收端收到了重新发送的数据包 100，那么接收端会回复 ACK103，继续进行剩下的数据传输，并且把数据包 99、100、101、102 按顺序传递给上一层。

流控制（flow control）

TCP 使用了端到端流控制以避免发送端发送数据过快导致接收端

无法处理。TCP 采用了滑动窗口（sliding window）实现流量控制。接收端通过 ACK 告诉发送端自己还能够接收多少数据，发送端不能发送超过该值的数据量。当接收端返回的窗口大小为 0 时，发送端停止发送数据，直到窗口大小被更新。由于 ACK 是由发送端发送的数据触发的，可能接收端窗口已经打开，但是由于发送端已经停止发送，故接收端没有机会通过 ACK 告知发送端新的窗口大小，在这种情况下会造成死锁。在实际实现中，发送端会设置一个定时器，如果定时器到期，发送端会尝试发送小数据包，以触发接收端的 ACK。

堵塞控制（congestion control）

为了控制传输速度防止堵塞网络，并且在网络容量允许的范围内尽可能多地传输数据，TCP 引入堵塞控制，用以判断当前的网络负荷，并且调整传输速率。TCP 通常采用加性增加、乘性减少（additive increase multiplicative decrease）的阻塞避免机制，即如果按时收到对应的 ACK，则下一次传输速率线性增加，否则则视为发生了网络堵塞，下一次传输的比特数折半。所谓的“按时”基于 RTT：发送端会估计 RTT，并且期望当数据包发送以后，在 RTT 时间内收到对应的 ACK。现代 TCP 需要分别实现慢启动、阻塞避免、快速重传和快速恢复，以达到最高的效率。具体请参考 11.3“工具箱”给出的资料。

2. 用户报文协议（User Datagram Protocol，UDP）

相比于 TCP，UDP 简单许多：连接建立时不需要经过类似于 TCP 的三次握手，只需要知道接收端的 IP 和端口，发送端就可以直接发送数据。同时，UDP 也没有 ACK，流控制和堵塞控制，故 UDP 本身不能保证传输是可靠的。由于 UDP 本身只负责把数据传输到目的地，故可扩展性比较强。有些应用可以实现基于 UDP 的特定算法，使得传输效率高于 TCP。例如，当发生丢包时，TCP 会重传该数据包，但该操作增加了传输延时。对于某些实时性要求较高的应用，可能继续传输新的数据更为重要，故基于 UDP 的传输方式可以更好地满足该要求。

通常而言，如果需要满足可靠性、有序接收、自适应带宽等要求，应该优先考虑 TCP，因为其协议本身确保了这点。如果对实时性要求较高，或者应用需要特定的网络传输特性，则可以实现基于 UDP 的传输协议。往往，这样的协议需要实现堵塞控制、流控制、重传等机制，故通常情况下都可以直接采用 TCP 以减小开发成本。

11.2 模式识别

例题 1 What happens after you typed a URL in your browser and pressed return key?

当你在浏览器中输入一个 URL 并按下回车键后，会发生什么事情？

解题分析：如果要连到远程服务器，首先需要知道服务器的 IP 地址和端口。其次需要发送接入请求到服务器，服务器返回响应数据。因此，如何寻址和如何建立链接是本题的关键。本题属于知识性问题，没有太多的解题技巧，直接给出解答如下：

参考解答：

（1）进行寻址：如果在浏览器缓存中存有 URL 的对应 IP，则直接查询其 IP；否则，访问 DNS（Domain Name System）进行寻址（Domain Name Resolution）。

（2）DNS 或者 URL cache 返回网页服务器的 IP 地址。

（3）浏览器与网页服务器通过三次握手建立 TCP 连接。由于是网页浏览服务，故浏览器连接到服务器的 80 端口。

（4）浏览器与服务器建立 HTTP 会话（session），接收来自服务器的 HTTP 数据。

（5）浏览器解析 HTTP 数据，在本地窗口内渲染并显示网页。

（6）当浏览器页面被关闭时，终止 HTTP 会话并关闭链接。

例题 2 If you are designing a reliable UDP, what should you do?

如果你要设计一个可靠的 UDP，该怎么做？

解题分析：通常，所谓的可靠都是指接收端能够将收到的数据情况反馈给发送端。由于我们已经知道一种可靠的传输协议，TCP，故可靠的 UDP 的设计完全可以参考 TCP 的设计方式，引入 ACK，flow control，congestion control 等模块。模块的实现可以直接模仿 TCP，也可以通过和面试官的沟通进一步确定需求。可靠的 UDP 的核心在于反馈机制，这里给出几个可能的实现方式。

参考解答：

由于可靠性要求在接收端能够恢复数据包的顺序，故发送端每个数据包都需要有序列号。现在着重讨论反馈机制：

（1）最朴素的 ACK 方式：发送端每发送一个数据包，都需要接收端返回 ACK，一旦超时，发送端重新发送数据包，直到该数据包被

接收端 ACK。该方法效率不高，因为之后的所有数据包都被当前数据包 block，并且每次返回 ACK 增加了 overhead。

（2）Block/bit map ACK：发送端发送一批数据包，例如 32 个，编号 0～31。接收端发回的 ACK 中用 32bits（4bytes）的 bit map 表示收到了哪些数据包，发送端再一次性重发所有未被收到的数据包。该方法能够更加充分地利用带宽，在发送端一次性传输更多的数据。但缺点是在发送端接收端都需要更深的 buffer，暂存正在传输的所有数据。

（3）ACK last packet：发送端可以在发送最后一个数据包时要求接收端反馈 ACK，并重发丢失的数据包。这样做的好处可以减少由 ACK 造成的 data overhead，但需要通过 buffer 暂存数据。

事实上，可以结合方法 2 和方法 3，在每一批数据包的最后一个置位 request ACK flag，要求接收端返回 bit map ACK。更进一步地，可以根据丢包率及延迟，估计网络状况，动态地调整 bit map 的大小：在网络状况好的情况下，用更大的 bit map，即同时发送更多数据。否则，减小发送数据量。事实上，这种对于网络状况的自适应也相当于实现了 congestion control。

例题 3 For real-time video conference application, how do you choose between TCP and UDP?

对于实时视频会议应用，如何在 TCP 和 UDP 之间做出选择？

解题分析：本题的关键在于比较 TCP 和 UDP 的特点，并且根据实时视频会议这个特定的应用场景进行选择。在 11.1.4 节提到过，TCP 的重传机制会增加延迟，所以不适用于当前场景。其次，视频音频编码本身可以容忍数据出错甚至数据丢失。因此，并不需要采用 TCP 进行可靠的数据传输。当某一视频帧出现丢包时，可以直接跳过这一帧或者继续播放上一帧。再次，一旦出现网络堵塞的状况，发送端应该主动丢弃一部分数据。原因是，即使这些视频帧发送到了接收端，也可能已经“过期”了，不会被解码显示。采用自己设计的 UDP 更便于实现对数据包的控制。然而，即使使用 UDP，也需要实现 TCP 的某些模块：比如需要 flow control 和 congestion control 来判断接收端的播放情况和网络情况，并且也需要反馈机制判断接收端的接收状况。尽管对于当前场景我们不需要 ACK 每个数据包，但是接收端可以反馈当前收到的最新完整视频帧的序号。这样，如果一旦发生丢包，发送端可以以接收端收到的最新视频帧为基础，压缩后继的视频。

11.3 工具箱

1. OSI 模型

http://en.wikipedia.org/wiki/OSI_model

2. Socket 编程

http://beej.us/guide/bgnet/

3. 网络技术参考书

Computer Networks, Fifth Edition: A Systems Approach, Larry L. Peterson, Bruce S. Davie

4. TCP

（1）RFC 793, Transmission Control Protocol, http://tools.ietf.org/html/rfc793

（2）RFC 5681, TCP Congestion Control, http://tools.ietf.org/html/rfc5681

（3）Three-way handshake (http://en.wikipedia.org/wiki/Transmission_Control_Protocol)

建立连接

为了建立连接，TCP 使用三次握手的机制。在客户端试图连接一个服务器之前，服务器必须首先开始监听为该连接而打开的端口：这称为一次被动打开。一旦建立了被动打开，客户端可以激活一次主动打开。要建立连接，会发生 3 次(或 3 步)握手：

1. SYN：客户端向服务器发送一个 SYN 来执行主动打开。客户端将段的序列号设置为一个随机值 A。

2. SYN-ACK：作为响应. 服务器回复一个 SYN-ACK。确认号设置为比接收到的序列号增加 1，例如 A+1；并且服务器为包选择的序列号，是另一个随机数 B。

3. ACK：最后，客户端向服务器发送回一个 ACK。序列号设置为接收到的确认值，例如 A+1，并且确认号设置为比接收到的序列号增加 1，如 B+1。

此时，客户端和服务器已经接受到了连接的一个确认号。步骤 1 和步骤 2 建立了一方的连接参数（序列号），并且连接得到了确认。步骤 2 和步骤 3 建立了另一方的连接参数（序列号），并且连接得到了确认。通过这些步骤，建立了一个完整的双向通信。

5. UDP

RFC 768, User Datagram Protocol, http://tools.ietf.org/html/rfc768

第 12 章 计算机底层知识

12.1 知识要点

12.1.1 进程 vs.线程

进程（process）与线程（thread）最大的区别是：进程拥有自己的地址空间，某进程内的线程对于其他进程不可见，即进程 A 不能通过传地址的方式直接读写进程 B 的存储区域。进程之间的通信需要通过进程间通信（Inter-Process Communication，IPC）。与之相对的，同一进程的各线程间之间可以直接通过传递地址或全局变量的方式传递信息。

此外，进程作为操作系统中拥有资源和独立调度的基本单位，可以拥有多个线程。通常操作系统中运行的一个程序就对应一个进程。在同一进程中，线程的切换不会引起进程切换。在不同进程中进行线程切换，如从一个进程内的线程切换到另一个进程中的线程时，会引起进程切换。相比进程切换，线程切换的开销要小很多。线程于进程相互结合能够提高系统的运行效率。

线程可以分为两类：

一类是用户级线程（user level thread）。对于这类线程，有关线程管理的所有工作都由应用程序完成，内核意识不到线程的存在。在应用程序启动后，操作系统分配给该程序一个进程号，以及其对应的内存空间等资源。应用程序通常先在一个线程中运行，该线程称为主线程。在其运行的某个时刻，可以通过调用线程库中的函数创建一个在相同进程中运行的新线程。 用户级线程的好处是非常高效，不需要进入内核空间，但并发效率不高。

另一类是内核级线程（kernel level thread）。对于这类线程，有关线程管理的所有工作由内核完成，应用程序没有进行线程管理的代码，只能调用内核线程的接口。内核维护进程及其内部的每个线程，调度也由内核基于线程架构完成。内核级线程的好处是，内核可以将

不同线程更好地分配到不同的 CPU，以实现真正的并行计算。

事实上，在现代操作系统中，往往使用组合方式实现多线程，即线程创建完全在用户空间中完成，并且一个应用程序中的多个用户级线程被映射到一些内核级线程上，相当于是一种折中方案。

12.1.2 上下文切换

对于单核单线程 CPU 而言，在某一时刻只能执行一条 CPU 指令。上下文切换（Context Switch）是一种将 CPU 资源从一个进程分配给另一个进程的机制。从用户角度看，计算机能够并行运行多个进程，这恰恰是操作系统通过快速上下文切换造成的结果。在切换的过程中，操作系统需要先存储当前进程的状态（包括内存空间的指针，当前执行完的指令等等），再读入下一个进程的状态，然后执行此进程。

12.1.3 系统调用

系统调用（System Call）是程序向系统内核请求服务的方式。可以包括硬件相关的服务（例如，访问硬盘等），或者创建新进程，调度其他进程等。系统调用是程序和操作系统之间的重要接口。

12.1.4 Semaphore/Mutex

当用户创立多个线程/进程时，如果不同线程/进程同时读写相同的内容，则可能造成读写错误，或者数据不一致。此时，需要通过加锁的方式，控制核心区域（critical section）的访问权限。对于 semaphore 而言，在初始化变量的时候可以控制允许多少个线程/进程同时访问一个核心区域，其他的线程/进程会被堵塞，直到有人解锁。Mutex 相当于只允许一个线程/进程访问的 semaphore。此外，根据实际需要，人们还实现了一种读写锁（read-write lock），它允许同时存在多个读取者（reader），但任何时候至多只有一个写入者（writer），且不能与读取者共存。

12.1.5 死锁

在引入锁的同时，我们遇到了一个新的问题：死锁（Deadlock）。

死锁是指两个或多个线程/进程之间相互阻塞，以至于任何一个都不能继续运行，因此也不能解锁其他线程/进程。例如，线程 A 占有 lock A，并且尝试获取 lock B；而线程 2 占有 lock B，尝试获取 lock A。此时，两者相互阻塞，都无法继续运行。

产生死锁的 4 个条件概括如下（只有当 4 个条件同时满足时才会产生死锁）：

（1）Mutual Exclusion - Only one process may use a resource at a time

（2）Hold-and-Wait - Process holds resource while waiting for another

（3）No Preemption - Can’t take a resource away from a process

（4）Circular Wait - The waiting processes form a cycle

12.1.6 生产者消费者

生产者消费者模型是一种常见的通信模型：生产者和消费者共享一个数据管道，生产者将数据写入 buffer，消费者从另一头读取数据。对于数据管道，需要考虑为空和溢出的情况。同时，通常还需要将这部分共享内存用 mutex 加锁。在只有一个生产者一个消费者的情况下，可以设计无锁队列（lockless queue），线程安全地直接读写数据。

12.1.7 进程间通信

在介绍进程的时候，我们提起过一个进程不能直接读写另一个进程的数据，两者之间的通信需要通过进程间通信进行。进程通信的方式通常遵从生产者消费者模型，需要实现数据交换和同步两大功能。

（1）共享内存（Shared-memory）+ semaphore

不同进程通过读写操作系统中特殊的共享内存进行数据交换，进程之间用 semaphore 实现同步。

（2）信息传递（Message passing）

进程在操作系统内部注册一个端口，并且监测有没有数据，其他进程直接写数据到该端口。该通信方式更加接近于网络通信方式。事

实上，网络通信也是一种 IPC，只是进程分布在不同机器上而已。

12.1.8 逻辑地址/物理地址/虚拟内存

所谓的逻辑地址，是指计算机用户（例如程序开发者）看到的地址。例如，当创建一个长度为 100 的整型数组时，操作系统返回一个逻辑上的连续空间：指针指向数组第一个元素的内存地址。由于整型元素的大小为 4 个字节，故第二个元素的地址时起始地址加 4，以此类推。事实上，逻辑地址并不一定是元素存储的真实地址，即数组元素的物理地址（在内存条中所处的位置），物理地址并不是连续的，只不过操作系统通过地址映射，将逻辑地址映射成连续的，这样更符合人们的直观思维。

另一个重要概念是虚拟内存。操作系统读写内存的速度可以比读写磁盘的速度快几个量级。但是，内存价格也相对较高，不能大规模扩展。于是，操作系统可以将部分不太常用的数据移出内存，存放到价格相对较低的磁盘缓存，以实现内存扩展。操作系统还可以通过算法预测哪部分存储到磁盘缓存的数据需要进行读写，提前把这部分数据读回内存。虚拟内存空间相对磁盘而言要小很多，因此，即使搜索虚拟内存空间也比直接搜索磁盘要快。唯一慢于磁盘的可能是，内存、虚拟内存中都没有所需要的数据，最终还需要从硬盘中直接读取。这就是为什么内存和虚拟内存中需要存储会被重复读写的数据，否则就失去了缓存的意义。

现代计算机中有一个专门的转译缓冲区（Translation Lookaside Buffer，TLB），用来实现虚拟地址到物理地址的快速转换。

与内存/虚拟内存相关的还有以下两个概念：

（1）Resident Set

当一个进程在运行的时候，操作系统不会一次性加载进程的所有数据到内存，只会加载一部分正在用，以及预期要用的数据。其他数据可能存储在虚拟内存，交换区和硬盘文件系统上。被加载到内存的部分就是 resident set。

（2）Thrashing

由于 resident set 包含预期要用的数据，理想情况下，进程运行过程中用到的数据都会逐步加载进 resident set。但事实往往并非如此：每当需要的内存页面（page）不在 resident set 中时，操作系统必须从虚拟内存或硬盘中读数据，这个过程被称为内存页面错

误（page faults）。当操作系统需要花费大量时间去处理页面错误的情况就是 thrashing。

12.1.9 文件系统

UNIX 风格的文件系统利用树形结构管理文件。每个节点有多个指针，指向下一层节点或者文件的磁盘存储位置。文件节点还附有文件的操作信息（metadata），包括修改时间、访问权限等。

用户的访问权限通过访问控制表（Access Control List）和能力表（Capability List）实现。前者从文件角度出发，标注了每个用户可以对该文件进行何种操作。后者从用户角度出发，标注了某用户可以以什么权限操作哪些文件。

UNIX 的文件权限分为读、写和执行，用户组分为文件拥有者、组和所有用户。可以通过命令对三组用户分别设置权限。

12.1.10 实时 vs.分时操作系统

操作系统可以分为实时操作系统（Real-Time System），和分时操作系统（Sharing Time System）。通常计算机采用的是分时，即多个进程/用户之间共享 CPU，从形势上实现多任务。各个用户/进程之间的调度并非精准度特别高，如果一个进程被锁住，可以给它分配更多的时间。而实时操作系统则不同，软件和硬件必须遵从严格的 deadline，超过时限的进程可能直接被终止。在这样的操作系统中，每次加锁都需要仔细考虑。

12.1.11 编译器

对于高级语言来说，代码需要通过编译才能够运行。编译通过编译器（Compiler）实现，是一个将程序源代码转换成二进制机器码的过程。计算机可以直接执行二进制代码。在编译的过程中，编译器需要进行词法分析（Lexical Analysis）、解析（Parsing）和过渡代码生成（Intermediate Code Generation）。编译器的好坏可以直接影响最终代码的执行效率。

版权信息

版权信息
书名：程序员面试白皮书
ISBN：978-7-115-40184-7

看完了

如果您对本书内容有疑问，可发邮件至contact@epubit.com.cn，会有编辑或作译者协助答疑。也可访问异步社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@epubit.com.cn。

在这里可以找到我们：

- 微博：@人邮异步社区
- QQ 群：436746675