编程之美——微软技术面试心得

内容介绍：

《编 程之美——微软技术面试心得》是微软亚洲研究院技术创新组研发主管邹欣老师继《移山之道——VSTS 软件开发指南》后的最新力作。他带领其他几位同事和实 习生经过 9 个月的时间完成了这本书。本书收集了大约 60 道微软技术面试题，这些问题妙趣横生，其解答别出心裁，还穿插了面试者的各种小故事。它传达给读 者：微软重视什么样的能力，需要什么样的人才。但它更深层的意义在于引导读者思考，提倡一种发现问题、解决问题的思维方式，充分挖掘编程的乐趣，展示编程 之美。

第 1 章 游戏之乐——游戏中碰到的题目... 1

1.1 让 CPU 占用率曲线听你指挥... 3

1.2 中国象棋将帅问题... 13

1.3 一摞烙饼的排序... 20

1.4 买书问题... 30

1.5 快速找出故障机器... 35

1.6 饮料供货... 40

1.7 光影切割问题... 45

1.8 小飞的电梯调度算法... 50

1.9 高效率地安排见面会... 54

1.10 双线程高效下载... 59

1.11 NIM（1）一排石头的游戏... 64

1.12 NIM（2）“拈”游戏分析... 67

1.13 NIM（3）两堆石头的游戏... 72

1.14 连连看游戏设计... 86

1.15 构造数独... 91

1.16 24 点游戏... 99

1.17 俄罗斯方块游戏... 108

1.18 挖雷游戏... 116

第 2 章 数字之魅——数字中的技巧... 117

2.1 求二进制数中 1 的个数... 119

2.2 不要被阶乘吓倒... 125

2.3 寻找发帖“水王”... 129

2.4 1 的数目... 132

2.5 寻找最大的 K 个数... 139

2.6 精确表达浮点数... 147

2.7 最大公约数问题... 150

2.8 找符合条件的整数... 155

2.9 斐波那契（Fibonacci）数列... 160

2.10 寻找数组中的最大值和最小值... 166

2.11 寻找最近点对... 171

2.12 快速寻找满足条件的两个数... 178

2.13 子数组的最大乘积... 182

2.14 求数组的子数组之和的最大值... 185

2.15 子数组之和的最大值（二维）... 192

2.16 求数组中最长递增子序列... 198

2.17 数组循环移位... 204

2.18 数组分割... 207

2.19 区间重合判断... 211

2.20 程序理解和时间分析... 215

2.21 只考加法的面试题... 217

第 3 章 结构之法——字符串及链表的探索... 219

3.1 字符串移位包含的问题... 221

3.2 电话号码对应英语单词... 224

3.3 计算字符串的相似度... 230

3.4 从无头单链表中删除节点... 234

3.5 最短摘要的生成... 237

3.6 编程判断两个链表是否相交... 241

3.7 队列中取最大值操作问题... 244

3.8 求二叉树中节点的最大距离... 250

3.9 重建二叉树... 256

3.10 分层遍历二叉树... 262

3.11 程序改错... 268

第 4 章 数学之趣——数学游戏的乐趣... 273

4.1 金刚坐飞机问题... 275

4.2 瓷砖覆盖地板... 279

4.3 买票找零... 282

4.4 点是否在三角形内... 286

4.5 磁带文件存放优化... 291

4.6 桶中取黑白球... 294

4.7 蚂蚁爬杆... 299

4.8 三角形测试用例... 303

4.9 数独知多少... 307

4.10 数字哑谜和回文... 315

4.11 挖雷游戏的概率... 322

编程之美 - 序

一位应聘者 (interviewee) 在我面前写下了这样的几行程序：

```
while (true) {

  if (busy)  i++;
else
}
```

然后就陷入了沉思，良久，她问道：**那 else 怎么办？怎么能让电脑不做事情？**

我说：对呀，怎么才能让电脑闲下来？你平时上课，玩电脑的时候有没有想过？这样吧，你可以上网查查资料。

她很快地在搜索引擎中输入“50% CPU 占用率”等关键字，但是搜索并没有返回什么有用的结果。

在她忙着搜索的时候，我又看了一遍她的简历，从简历上可以看到她的成绩不错，她学习了很多程序设计语言，也研究过“设计模式”、“架构”、“ SOA”等，她对 Windows、Linux 也很熟悉。我的面试问题是：“如何写一个短小的程序，让 Windows 的任务管理器显示 CPU 占用率保持为 50%？”这位应聘者尝试了一些方法，但是始终没有写出一个完整的程序。 面试的时间到了，她看起来比较遗憾，我也一样，因为我还有一系列的 后续问题没有机会问她：

· 如何能通过命令行参数，让 CPU 的使用率在保持任意位置， 如 90%？

· 如何能让 CPU 的使用率表现为一条正弦曲线？

· 如果你的电脑是双核（dual-core），那你的程序会有什么样的结果？为什么？

作为面试者，我最希望看到应聘者能给出独具匠心的回答，这样我也能从中学到一些“妙招”。遗憾的是看到“妙招”的时候并不多。 自从 2005 年回到微软亚洲研究院担任开发经理一职以来，我面试过不少应聘者，也为微软校园招聘出过考题，做过员工和实习生的培训。我也了解到不少同学认 为软件开发的工作没意思，是“IT 民工”、“软件蓝领”。我和其他同事也听到一些抱怨，说一些高校计算机科学的教育只停留在原理，而忽视了对原理和技术的理解和运用。

写程序真的没有意思？为什么许多微软的员工乐此不疲？我和一些喜欢编程的员工和实习生编了这本书，这本书想通过分析微软面试中经常出现的题目，来展 示编程的乐趣。编程的乐趣在于探索，而不是在于背答案。面试的过程就是展现分析能力、探索能力的过程，在面试中展现的巧妙的思路，简明的算法，严谨的数学 分析就是我们这本书要谈的“编程之美”。

还有不少同学问：“你们是不是有面试题库？” 言下之意是每个应聘者都是从“库”中随机抽出一道题目，如果答对了，就中了，如果答错了，就 bye-bye 了。书中的一些关于面试的问答，能回答这样一些疑惑。

本书的题目，一部分来源于各位作者平时自己的实践，例如有一次一位应聘者滔滔不绝地讲述自己如何在某大型项目中进行 CPU 压力测试，听上去水分不 少，我一边听一边琢磨“怎么才能考察是否真正懂了 CPU，任务调度。。。。。”，后来就有了上面提到的“CPU 使用率”的面试题。有些题目在网上流传较 广，但是能得到正解的不多， 我们在书中加上了详细的分析，提出了一些扩展问题。有些题目在一些教科书和专业书籍中有更深入的分析和解答，读者可以参考。

书中的大多数题目都能在四十五分钟左右内解决，这也是微软一次技术面试的时间。本书不是一个“答案汇编”，很多题目并没有给出完整的答案，有些题目还有更多的问题要读者去解答。这是这本书和其他书籍不一样的地方。面试不是闭卷考试，如果大家都背好了“井盖为什么是圆的”的答案来面试，但是却不会变通，那结果肯定是令人失望的。

为了方便读者评估自己的水平，我们还按照每道题目的难度制定了相应的“星级”：

- 一颗星：不用查阅资料，在 20 分钟内完成
- 两颗星：可以在 40 分钟内完成
- 三颗星：需要查阅一些资料，在 60 分钟左右完成

由于每个人的专业背景，经历，兴趣不一样，这种“星级”仅仅是一种参考。

我们的水平非常有限，书中的题目并不能代表程序设计各个方面的最新进展，虽然经过几轮审核，不少解法都可能有漏洞或错误，希望广大读者能给我们指正。我们计划在微软亚洲研究院的门户网站(www.msra.cn) 上开辟专栏和读者交流 – 初学者和高手都非常欢迎！

本书的内容分为下面几个部分：

- 玩电脑：电脑上的游戏是给人玩的，CPU 也可以让人‘玩’。这一部分的题目从游戏和作者平时遇到得的有趣问题出发，展现一些并不为人重视的问题，并且加以分析和总结。希望其中化繁为简的有趣思路能够对读者解决其它复杂问题有所帮助。

- 字符串和常用数据结构问题：对字符以及常用数据结构的处理几乎是每个程序中必然会涉及到的问题，这一部分汇集了常用的对字符串以及链表、队列以及树等操作的题目。

- 数字问题：编程的过程实际上就是和数字以及字符的打交道的过程。如何提高掌控这些数字和字符的能力对提高编程能力至关重要。这一部分收集了一些好玩的对数字进行处理的题目。

- 数学问题：书中还列了一些不需要写具体程序的数学问题，但是其中显示的原理和解决问题的思路对于提高思维能力还是很重要的，我们把它们单独列出。

· 关于笔试，面试，职业选择的一些问答：微软的各种技术职位，招聘流程是很多学生所关心的，因此我们把一些相关的介绍和讨论也收录了进来。

我们希望《编程之美》的读者是：

1. 计算机系、软件学院或相关专业的大学生、研究生，可以把这本书当作一个习题集。

2. 面临求职面试的 IT 从业人员，不妨把这本书当作“面试真题”，演练一下。

3. 编程爱好者，可以平时随便翻翻，重温数学和编程技能，开拓思路，享受思考的乐趣。

《编程之美》由下面几位作者协同完成，如果把这本书比作一个软件项目，它有下面的各个阶段，每个阶段有不同的目标和角色：

1. 构想阶段：邹欣

2. 计划阶段：邹欣，刘铁锋，莫瑜。

3. 实现阶段/里程碑（一）：上述全部人员，加上李东，张晓，陈远，高霖 （负责封面设计）。

4. 实现阶段/里程碑（二）： 上述全部人员，加上梁举，胡睿。

5. 稳定阶段：上述全部人员，加上博文视点的编辑们。

6. 发布阶段：邹欣，刘铁锋，和博文视点的编辑们。

这本书从 2007 年 2 月开始构思，到 2007 年 11 月底交出完整的第一稿，花费的时间比每一位作者预想的要长得多，一方面是大家都有日常的工作和学 习任务要完成；更重要的是，美的创造和提炼，是一个漫长和痛苦的过程。要把“编程之美”表达出来，不是一件容易的事，需要创造力、想象力和持久的艰苦劳 作。就像沈向洋博士经常讲的一句话——Nothing replaces hard work。

这本书的各位作者，都是利用自己的业余时间参与这个项目，他们的创造力、热情、执着和专业精神让这本书从一个模糊的构想变成了现实。 通过这次合作，我从他们那里学到了很多，借此机会，对所有参与这个项目的同仁们说一声： 谢谢！

在本书编写过程中，作者们得到了许多微软亚洲研究院的同事和实习生的帮助，具体请参见“致谢”。

我们希望书中展现的题目和分析，能像海滩上美丽的石子和漂亮的贝壳那样，反映出造化之美，编程之美。

编程之美 - 推荐序

我在卡内基梅隆大学毕业找工作的时候，经常和其他同学一起交流面试的经验。当时“闻面色变”的公司有微软，研究所有 DEC 的 SRC。每次有同学去微软或 SRC 面试回来都被其他人追问有没有什么有趣的面试题。我也是那时第一次听说下水道井盖为什么是圆的。

我 自己申请微软美国研究院时被面试了两天，见了 15 个人，感觉压力很大。 至今还记得被一位面试官不断追问我论文中一个算法的收敛性的热烈讨论。在微软工作的十几年中，我自己也面试了非常多的新员工。特别在微软亚洲研究院的九 年，经常感觉很多刚刚毕业的优秀学生基础很好，但面试的准备不足。我非常欣慰地看到邹欣工程师和微软亚洲研究院其他同事们努力编写了这本好书，和大家一起 分享微软的面试心得和编程技巧。相信更多的同学会因此成为“笔霸”，“面霸”，甚至“offer 霸”。

程 序很美妙，虽然很难写。程序要想写的好，需要学好一定的基础知识，包括编程语言，数据结构和算法。程序写的好的人通常都有慎密的逻辑思维能力和良好的数理 基础。还需要熟悉编程环境和编程工具。古人说“见文如见人”。我觉得程序同样也能反映出一个人的功力和风格。好的程序读来非常赏心悦目。我以前常问的一道 面试题是“写一段自己觉得写过的最好的程序”。

编 程很艰苦，但是很有趣。本书的作者们从游戏中遇到的编程问题谈起，介绍了数字和字符串中的很多技巧，探索了数据结构的窍门，还发掘了数学游戏的乐趣。我希 望读者在阅读本书是能找到编程的快乐，欣赏到编程之美。本书适合计算机学院、软件学院、信息学院高年级本科生、研究生作为软件开发的参考教材。也是程序员 继续进修的优秀阅读材料。更是每位申请微软公司和其他公司软件工程师面试的必读秘笈。

人类的生活因为优秀的程序员和美妙的程序而变得更加美好。

沈向洋

微软公司杰出工程师

微软公司全球资深副总裁

2008 年春节于香港

编程之美 - 潘爱民点评

潘爱民倾力推荐《编程之美——微软技术面试心得》

我很早知道邹欣计划要写这样一本书，也能够预计到这本书定会广受欢迎，因为它符合当前大量求职人员的需求，毕竟于他们而言，谁不想知道微软亚洲研究院在招 人时候问些什么问题呢。另一方面，把考察软件技术人员专业知识和相应技能的各种手段加以归纳和整理，这本身也是对业界的贡献，所以，我相信，一旦这本书如 计划般完成，其对业界的影响将是深远的。

在我的面试经历中，通过一些具体的程序问题来考察人，往往是最有效的，即使是一些人所皆知的问题，也往 往能够挖掘出被面试者的亮点或弱点，原因在于，每个问题都有

不同层次的解答之辞，面试者总是可以刨根究底地问下去。我们在看一段程序的时候，思路固然重 要，细节也是不可忽视的，比如整数是否越界、指针是否为空，等等。这些细节可以用于考察基本功，毫无疑问，基本功不扎实的人通常很难得到面试者的青睐。

当拿到这本书的样稿时，我迫不及待地放下手头工作，阅读起来。有些题目的内容会引起强烈的共鸣，尤其是 那些自己非常熟悉并且又深知解答的题目；也有一些题目让我异常惊诧，原来除了我所知道的解答思路之外，还有更好的解答以及更深层次的原因。还有一些题目是 从来没想到过的。阅读过程是一次愉快的享受，也是脑细胞持续活跃的过程。

充满好奇心的人们总是能从生活的点点滴滴中想到或找到各种优化的余地，比如说，楼宇中的电梯常常显得很 “傻”(微软研究院所在的希格玛大厦的电梯是一个典型的例子)，更智能或更有效的调度策略完全有可能；近距离内的交通灯联动可以有效地提高行车效率。程序 员在玩电脑游戏的时候常常会想着怎么自动完成一些过程，比如说，本书中提到的俄罗斯方块游戏中如何有效地旋转和移动可快速地消除积木块、24 点游戏如何自 动求解、推箱子游戏如何自动求解，扫地雷游戏如何自动完成，等等。实际上，这些自然的疑问正是训练程序能力的好来源，本书采录了不少此类题目。因此，阅读 本书可以满足很多人的好奇心，这也正是我自己的体会。

尽管作者在前言中声称“虽经过几轮审核，不少解法仍可能有漏洞或错误”，但事实上，在绝大多数题目的讲 解中，作者已经由浅入深地把问题分析透了，而且，作者也为读者指出了进一步思考这些题目的方向。不同背景的人在看到这些题目的时候，可能会有不同的解法， 甚至完全不同的思路。举例而言，邹欣曾经问过我如何控制 CPU 占用率曲线的问题，我当时的直觉是，直接截取 Taskmgr 调用的相关 API 函数，从而达到 随意控制 CPU 占用率曲线的目的。显然这不是规范的做法，本书的分析揭示了这个问题背后的本质道理以及考问要点。另一种情况，即使有的问题你深知其理，但 看过本书仍然很有收获。例如，在斐波那契数列问题中，我知道直接递归法的缺陷，也知道如何简化成迭代法来改进效率，还会推导通项公式，但是，书中的细致讲 解仍然让我对这个问题有了更进一步的认识。这是本书的深度所在，如果读者更加在意所选题目背后的深层次道理，相信书中的讲解不会让你失望。

除了趣味性以外，本书中的题目讲解之中也融入了大量专业知识。这使得本书可以作为计算机数据结构课程或 算法课程的辅助参考书。比如，有些问题的解答涉及到贪心算法或动态规划方法，算法的复杂度分析更是无处不在。数据结构教科书中介绍的链表(list)、队 列、hash 表和二叉树等常用数据结构也多有提及。因此，对于正在学习数据结构或算法课程的学生来说，本书中的问题正是对课程中所学知识的一次检阅，通过 本书他们可以看到这些知识是如何用于解决实际问题的。从我自己的教学经验来看，这样的题解分析有助于提高学生的学习兴趣。另一方面，阅读本书也需要有必要 的计算机算法和程序设计知识作为基础，否则阅读的效果会大打折扣。

我大致了解本书的成书过程，从策划阶段到题目收集，再到成稿和改稿，我能体会到邹欣和他的写作团队倾注 了大量的精力来写作这本书。他们尽了最大的努力来编写这本书，无论是原创的题目，还是传统的题目，他们都努力把题目分析透彻并提供扩展思考的余地。邹欣在 发送样稿给我的信中说道：“Our goal is to ship a top quality book. I can'

t say "world class", but definitely "best in China" level.”以我阅读这本书的体会来讲，他们做到了这一点。我相信，这本书的出版会符合我当初的预期，它会影响到很多人。

潘爱民

2008 年 2 月

编程之美 - 孟岩点评

这是一本让人着迷的书！

从我得到样书的那一刻起，在每天的闲暇时间阅读和思索这本书中的题目就成了我的一个新习惯。虽然网络上 早就流传着不少微软面试题，坊间也不乏一些程序员面试类的图书，但是像这样集中展示高水平编程面试题目，并且以启发性方式对这些题目予以权威解答的图书， 这还是第一本。对于那些正在准备面试的同学来说，这本书毫无疑问是宝贵的学习资料。而在我看来，即使是对于已经工作的程序员来说，这本书也是非常值得用心 阅读的。

实话实说，对于算法和数学类谜题的意义和价值，在程序员社群里长期以来就存在很大的争议。CSDN 上每 隔一段时间就会有人讨论“算法真的重要吗”或者“数学真的重要吗”这样的问题。很多人对此都持质疑甚至是否定态度，他们认为，对于企业来说，是软件产品而 不是具体的程序创造了价值，而创造成功的软件产品是一个庞大而复杂的系统工程，优质的算法和程序在其中的作用是有限的，相反，对平台和系统的理解、对领域 知识和规则的掌握、软件质量的控制、产品设计、架构的选择和设计、平台和工具选型，以至于团队管理和有效沟通，对于软件工程师来说是更为重要的技能和素 质。相比之下，算法和数学只要基础扎实就可以了，在实践当中反而不如上面那些要素显得重要，更没有必要在类似智力测试般的面试谜题和奇技淫巧上花费太多心 思了。

这样的看法，当然有很有价值的方面。当代的软件工程师，确实需要建立更全面的知识技能体系和系统思维， 但是以此来否定和贬低算法和数学基本功的重要性，否定面试谜题的意义和价值，则又属于只见树木不见森林。事实上，这些谜题考察和锻炼的，并不是算法和数学 的“奇技淫巧”，而是扎实、严密和具有创造性的思考能力，面对问题有条不紊的分析能力，和不断深入、刨根问底的精神。毫无疑问，这些素质，都是软件工程师 身上最宝贵的东西。

本书就是对这一问题的有力证明。请翻开这本书，随便挑选几个问题，认真思考，尝试解答，再看看作者的思 路，在其启发下更进一步思考，尝试给出更多更好的解决方案，甚至更进一步，提出书中都没有提出的问题，把问题想透，把程序实现出来，验证自己的想法。毫无 疑问这样的阅读方法是相当吃功夫的，但通过这样的方式，不但能够最大程度地获得本书的价值，也能够实实在在提高读者的基本功、思考力和创造力。毫无疑问， 这些能力的意义要远比任何具体知识和技能的获得都更重要。对系统与平台的理解也好，对领域知识的掌握也好，产品的设计、架构的选择也好，所有这些算法之外 的技能，不都需要强

悍的思考能力的支撑才能获得吗？事实上，在这个知识开放共享的时代，头脑和思维才是唯一核心的竞争力，从这个意义上讲，这本书是直接面 向核心竞争力发展的，其意义何其大也！

这本书另一个必须要提的意义，就是它的“美”。真正的程序员都领略过程序之美，那些简洁有力的代码，精 巧严密的构思，高效直接的解决方案，美得令人窒息。可惜，在日益工业化和利益驱动的 IT 中，这种美已经是越来越少见的东西。但我想每一个真正热爱编程的 人，都渴望欣赏这种美，渴望在思考过程中一次次“Aha！”式的愉悦。这本书就能够最有效地满足我们欣赏编程之美的渴望，题目的美，思考过程的美，解答的 美，延伸思考的美，这种美的感觉，对于真正的程序员来说，本身就是一种精神享受。

虽然这本书表面上是指导面试的，但是依我来看，已经工作的职业程序员更有必要好好读读这本书。毕竟世界很大，不是每个人都要去微软或者谷歌，而这本书的意义绝不仅仅在面试，更重要的是在编程之美，思考之美。

编程之美 - 创作后记

转眼一年过去了，《编程之美》就要出版了。

回想大家一起合作的日子，我自己学到了很多东西。当初，看到邹欣老师的倡议

，觉得创作这样的一本书还 是挺有意义的，并且抱着跟其他优秀同事学习的想法，有幸成为了一位“作者”。其实，我觉得把我的名字挂在书上，有点尸位素餐的感觉。在编写《编程之美》的 过程中，我贡献甚微。相反，伴随着我从学生到软件开发工程师的角色转变，其他作者无形中给我上了一堂课。

最开始，我们先收集题目，大家一起讨论改进各个问题的解法，然后去掉一些面试时并不合适的问题，手头也有了不少草稿。我一度认为，一切进展得都很“顺利”，应该很快就“大功告成”了。但很快地，我发现这些草稿，离“书”的要求还很远。

后来，事实也证明后面的工作才是最重要的。其实我是一个表达很不好的人，写作文对我来说从来都是一件头痛的事情(写这篇文章的时候也不例外)。刚开始，我 总是三言两语就写完自己的想法，有时甚至是不成熟的想法，然后就以为差不多了。相反，其他同事在这方面就做得很好，更懂得如何去把一个问题描述清楚，写得 有条理、通俗易懂。

如何进行版本维护？如何保证写作风格尽可能统一？如何把一件看似简单的事情做好？这些问题，我压根就没有考虑过。

类似地，在软件开发方面， 可能也有不少跟我一样的微软新员工，觉得自己大大小小的程序也写了一些，软件开发也就是写写几行程序，没太大问题。以前自己写程序的时候，可以随意地按自 己喜欢的方式做，变量名也不会考虑命名，函数的接口可以不用考虑，反

正整个程序都在自己心里。但这就有点像是我们自己随意记录草稿和笔记，除非只是给自己看，不考虑会有其他人阅读。不然，这草稿和笔记别人根本看不明白。这样想来，做软件开发和写一本书还真是有很多相似之处。写书，希望相关的读者能读懂；而 写软件也希望能让用户用的明白。一个团队合作开发软件，就像一伙人一起写一本书。因为你的代码需要维护，因为多人合作开发，而不是“单兵作战”如何设计 和定义模块接口，管理维护代码？我们再也难做到整个程序都在自己的掌控之中。还有，如何写可靠安全的代码呢…….这些是软件开发中经常碰到的问题，也是面试时应聘者容易忽略的问题。

当年，我被面试[JZ1] 的时候，曾碰到一个问题：完成下面归并排序函数，将有序数组 arrX 和 arrY 归并排序的结果存到 arrZ 数组中。

bool MergeSort(int* arrX, int nX, int* arrY, int nY, int* arrZ, int nZ)

在面试房间的白板上，当时我很快就把整个程序写完了，然后开始想后面有些什么难题呢？但，面试官简单的几句话就让我无言以对——“你有没有考虑数组指针 arrX 和 arrY 是否为空？它们的分配空间是否可能重叠？”那时，我还认为这些太苛刻了吧。

但在实际软件开发的时候，正是这些细节，可能会带来潜在的 bug。以前，会觉得一个出错的可能性为百万分之一的程序尚可忍受，但在类似搜索引擎这种每天被大量用户使用的软件系统上，这样的错误却是无法容忍的。

对我个人来说，参与创作 《编程之美》，给了我与其他同事合作及一起学习的机会。虽然读者们不能像我一样亲历写书过程中受到的启发，但相信读者们也可以从书中或难或易的问题里得到 一些启发。一本书的出版，就像一个新的软件系统的发布和上线。出版(发布)前的心情总是很复杂，既兴奋却多少也有些担心。兴奋，是因为我们的努力可能会带 给读者帮助，希望大家能从中得到一些启发，即使在面试的时候并不一定会碰到这些具体的问题；担心，则当然是害怕由于我们的疏忽，时间和水平的有限而存在潜 在的 bug，希望读者们多指正。

Chen Yuan

我 应该算是最早知道将要编写《编程之美——微软技术面试指南》这本书的少数几个人之一。那时邹欣老师正在对《移山之道——VSTS 开发指南》进行最后的润 色，而我还在学校里上研究生课程，生平第一次接受正统的计算机专业教育。当邹老师问我要不要参与编写时，作为一名自诩的“文学青年”而不是“计算机高 手”，我毫不犹豫地答应了。

我 本科读的是航空学院，在大二时闲得无聊抱着玩的心态才开始真正自学编程的，然后凭着热情和兴趣就一头扎了进来。但是，我心里一直有种隐隐的痛，我可以熟练 使用 ASP.NET、AJAX 很快地做出一个网站来，却对一些基本的数据结构、算法一知半解。唯一一次认真去读《数据结构》那本书还是保研机试前一夜临时 抱佛脚，通宵看了排序、树、图之类常考的重点。虽然最后考出来成绩不错，但自己斤两多少，自己最清楚。所以实际上我对许多公司偏重算法的面试一直以来都抱 有一种畏惧感和神秘感，而且非常仰慕那些受过

ACM、ICPC 训练过的同学，尤其是那些能很快分析出问题复杂度的人。

但 是毕竟我不是科班出身，而且只在学校里面做过一些简单的网站项目，这让我在很长一段时间里都抱有一种误解，即认为工程能力和算法解题能力是不相干的两回 事，佐证就在于有些人可以很轻松地解出一些算法题却无法用 C#写一个真正可用的软件；而像我一样的人可以轻车熟路写出一个“看上去很美”的 CMS 系统，但 面对一些课本上的算法题时却手足无措。而且更要命的在于，简单的网站做多了，我逐渐认为做工程不需要所谓的算法，算法好只能让人拿到更高的课程分数或是竞 赛奖项，而在计算机科学这一非常讲究实践的领域中，只有良好的工程能力才有办法真正实现某个项目。于是，在很长一段时间里，我对那些能通过解出很难的算法 题拿到很好的 offer 的人都比较嗤之以鼻，并对那些公司的招聘标准感到疑惑不解——明明是我更能干活，实践经验和能力上更强，凭什么不要我而是他们呢？

我 觉得我最大的幸运在于，随后的一些经历让我很快走出了这个误区。在本科的最后一个学期，我幸运地获得了一个前往微软亚洲研究院实习的机会（面试时考了我一 道智力题而不是算法题 J）。在实习过程中，我才“真正”地做了一个软件项目，并且通过和其他实习生的交流，“耳濡目染”地看到了许多现实中的研究性软件的 开发过程，这些经历带给了我许多前所未有的体验。在现实的软件开发中你会看到各种形式各异的需求，比如在一定数量的帖子中找出发帖最多的“水王”，在这之 前我开发过的网站最多也不过几千条记录，所以我即使用最简单的遍历也能很快实现这一功能，但是当你面对的是十万甚至百万级别的现实数据时，问题就从最基本 的“实现”变成了“更快更高效地实现”了！令我感到汗颜的是，我往往只能用效率最低的复杂度实现类似的功能，而面对如何更优雅更高效地实现它时，我常常感 到力不从心。

这 些经历让我逐渐意识到，我所沾沾自喜的工程实践能力实际上只是一种“实现”的能力，而在解决现实世界的实际问题时，更需要的是一种“优美的实现”，因为只 有在可接受的时间或空间约束条件下的实现才是真正能解决问题的答案。而如何找到所谓的“优美的实现”，一个人的算法能力在这里就起到了决定性的作用。算法 实际上是对现实问题的抽象，因为现实问题是复杂的，我们可以把它抽象成模型。寻找合适的数据结构表示问题模型，并通过分析，寻找到对应的解决算法，这种抽 丝剥茧的思维方式将会使得开发者事半功倍。那句著名的“软件 == 算法 + 数据结构”并非空穴来风，我也从这些经历中逐渐理解了微软等公司的招聘标准实际上没有错，因为他们需要找的是能真正通过分析来解决实际问题的人。如果把工 程实践能力比作一辆车的轮子，那只能说明这辆车具有了移动的能力，而让这辆车能又快又稳地运行，则需要算法分析能力这台强劲的发动机驱动，这两种能力是相 辅相成的。

我 觉得自己更大的幸运在于，在我逐渐明白了这些道理后，参与创作了《编程之美》这本书。编书的过程也是我自己动手解里面一道道有趣题目的过程，期间我对一个 个优美、巧妙的解法拍案叫绝，在遇到难题或想不通的时候，就通过与其他编者一起讨论解决，这些经历都让我不断体会到“解法之美”和“问题之美”。《编程之 美》里的许多题目实际上都来源于现实项目中所遇到的具体问题，它们或是实际问题的简化，或是改头换面以其他有趣的场景表示出来。但是万变不离其宗，通过把 问题抽象化，并运用算法分析寻找解决方案将是解题的利器。这种思考方式也是我们希望通过本书传递给读者们的。祝大家能在阅读的过程

中体会到“美”的无处不 在。

Ju Liang

在很久以后才意识到 BOP 原来是“Beauty Of Programming”的缩写——在我设置了 outlook 里 bop puzzle 目录接收 bop 组的邮件很久以后。

BOP，《编程之美——微软技术面试心得》，虽然标着“面试心得”有些落俗，但或许会让更多的人在看到副书名时受到较强的阅读刺激（我是倾向于“编程之美”这一书名的）。

接触 BOP 于 2007 年 8 月——来微软入职一个月后。而在大约一年前，也在为找工作做准备：写简历，在网上看笔试面试题，也包括面经，似乎也在图书馆的新书阅览室里读过一本简历/面试相关的书（后来也证明，这些确有帮助）。

2007 年 7 月入职，然后是很多的 training。邹欣是其中一个 Engineering training 的 coach。一次，他在邮件中给了一个有趣的 Stone Quiz，后来偶给了一个数学解，就幸运地来到 BOP 创作小组。

当 时 BOP 的题库都基本定了，初步的解答也有。接下来需要做的是 review，包括解法的验证、给出新的算法、文字语言润饰、代码规范、标点符号、字体大小 颜色等等。题目的状态从 active（待修阅）到 peer review（我们修阅后）到 editor review（出版社修阅后）再到 active，如此多轮往返，直到大家都满意。这本书不在我们的 commitment 之内，没有分配常规的工作时间，所 以很多的时候大家会用周末开会，或者晚上在中餐馆小聚，谈下各自的进度，或者中午在日餐馆，一起 review 一组题目。在这个过程中，也很是享受，能分享 到别人算法的美妙，自己也会在细节处求精。（后来因为项目很紧，没能更多的投入，有很多歉意）

在面试中，我多次被问到，为什么选择计算机。源于兴趣——而这又多是缘于数学。虽然大概小学就喜爱数学的，但真正窥见数学其美是在高中图书馆的书堆里读了《趣味数论[JZ2] 》，书里也是列举了很多有趣的数论题目，从多个角度给出了优美的解答，用通俗简单的语言。很多年过去了，后来虽没有学数学专业，却仍是记得那本书，以致即使对于学文的人，我也会推荐他们读这本关于数论的书。

现在，对于学计算机的年轻人，我会向他们推荐《编程之美》，对于非学计算机的年轻人，我也会推荐《编程之美》，相信书中用通俗简单的语言解说的优美思想也会吸引他们的兴趣，让他们受益。

希望《编程之美》能让更多的人进入程序世界，感受这个世界中引人入胜的美。

Rui Hu

很久以前就听说邹欣老师要 出一本微软亚洲研究院关于编程艺术的书，但是自己很晚才加入到这个人才济济的编写团队中来，也因此错过了许多的故事。究其原因，大概有两条：

一是因为邹欣 老师当年是自己的面试官，而且当年他给我出的第二个题目我当时并没有得到很好的结果，想起这个，心里总是有些忐忑；二是自认为对于编程的认识并不能说有多么深刻精辟，自己和印象中的那种大师风范好像还相差非常远，怕在众多的武林高手面前献丑丢脸。因此也就犹豫到今年 9 月份才真正加入这个队伍做一点事情， “Better later than never”，后来发现，能够有幸加入到这样一个团队做这么一件有意义的事情，机会很难得。

自认为不是程序设计天才， 但对于那些传奇的程序设计大师境界，“虽不能至，心向往之”。自己看过不少关于计算机和程序设计方面的书籍，面试过一些程序员，也在实际的工作中遇到过很 多的编程问题，对于程序设计有了自己的一些体会。程序设计其实本质上是一个知识和能力综合应用的过程。要编写出好的程序来，基础知识很重要，如果基本的数 据结构和经典的算法都不知道，很难编出很好的程序来；但是当你有了一定的基础，基本了解了常用的数据结构和算法以后，想象力和思维方式就更为关键，正如爱 因斯坦所说：“想象力比知识更重要”。知道什么时候在什么样的问题上采用什么样的算法和数据结构，非常不容易；如果还能够将一些常用的算法和数据结构针对 特定的问题做一些优化和修改，甚至创造出一些新的数据结构和算法，就更为难得。

计算机方面的书籍很多，讲 述基本算法和常用数据结构的书也不少，《编程之美》的创作思想和一般的编程书籍不大一样，全书并不是给大家讲解一些计算机和程序设计的理论知识，而是通过 分析讲解实际生活中的一些问题，来启迪大家的思路，让大家体会程序设计的思维方式。这本书的独特之处就在于，它更强调的是描述程序设计的思维方式，分析的 是将实际问题抽象为计算机程序设计问题，并找到最优算法的过程，并且花了很多精力来比较各个算法的优劣，并分析各个算法的复杂度。

参与创作这本书，通过和邹欣老师以及其他作者之间的交流和讨论过程，我接触到了很多以前从未碰到的新奇的问题，学到了很多精巧的解法，更重要的是，开阔了自己的思路，也认识到了程序设计科学和艺术的博大精深，需要用一辈子去学习、提高。我[JZ3] 希望，读者朋友们也能通过这本书得到自己的收获。

Tie Feng

实在很难相信，我这个不容易坚持持续做完且做好一件事情的人，竟然坚持花了近一年的时间和大家一起创作出了一本书。

历经重重修改、审阅，这本《编程之美》的修改已经接近了尾声。

这一年的时间，也正是我到微软工作的第一年，也经历了从一个求职者、软件工程师到面试官的角色转变。而这个过程，也伴随了整本书的出版过程。

作为一个曾经的求职者，当 时自己能够做的，就是搜集和整理能够在网络上搜刮到的所有题目。算法题、智力题以及各种面经。把各种题目做到让自己条件反射为止。而这本书的创作过程之 初，也同样如此。把自己经历过的、网络上看到的、同事们讨论过的题目统统都拿过来，进行分类和研究，作为撰写书籍的原材料。虽然美中不足的是有些题目的分类略微牵强，但是，从分类的结果来看，也基本上代表了微软会经常 probe 面试者的几个方面： problem solving, coding skills, algorithm analysis skills。

而作为一个软件工程师，编 写程序是本职工作。每个新人必受的打击，就是 code review。在这个 review 的过程中，你能够体会到多年的编程经验会体现在什么地方。你会发现就算是短短的几十行代码，review 之后，可能一行 都不能用。的确如此，当你的产品发布出去之后，你希望它能够稳定使用，不出任何问题（当然，这是理想情况）。你会发现需要考虑的，实在有太多的问题。

在 release 了第一个 project 之后，回过头来再看看同事们的解答，以及中间附上的代码，就能够清楚地从自己写的代码中看出编程的功力。自己工程的实践、同事们在 code review 过程中对代码提出的种种质疑，以及处理种种程序在实际运行中碰到的问题，也为我积累了更多的思考，并且会更加了解应该通过题目给读者传递怎样 的信息。

而当自己开始面试 candidate 的时候，还是有更多的收获。在面试的高峰期，一周之内大概会有 3 个电话面试，3 个 on site 面试。在 HR 统计的结果中，我手上的通过率不到 20%。在面试的过程中，我也会直接拿书稿中的题目来挑战面试者，希望能够看到有精彩的解法。同时，也是对 题目的一个有益的补充。不仅如此，怎样的面试题目才能挑选出一个合格的软件工程师？反过来，换到读者的角度，这个问题应该是，微软到底会出怎样的题目来甄 别求职者？

以我个人的经验来看，分析问题的方法更加重要。这也是在书中努力想传达给读者的重要信息。

从分析的套路上来说，作者们也都是通过先提供一个简单的方法，然后试图找出一个更好的办法，这样不断地挑战自己的智力来分析题目。

而这个过程，也正是面试的过程中，面试官和求职者的互动过程。这也是每一道题目想教会给读者的一个套路。

正所谓“无招胜有招”“万变不离其宗”。题目只仅仅是一个表象，面试的过程中希望看到的却是面试者真正的实力。

所以，也真心盼望读者能够在阅读本书的过程中，体会到面试官到底想 probe 出些什么 skills。

在本书的创作过程中，从邹欣老师身上学到了不少东西。也让我明白了一个道理：不怕慢，就怕站。方向确定后，只要天天有行动，最终一定能够有结果。计划一定是要以执行为依托的。

限于作者们的水平，每一道题目的解答绝非尽善尽美，甚至还会有潜在的 bug。也切盼能够得到读者的反馈。

编程之美 - 让 CPU 占用率曲线听你指挥

问题

写一个程序，让用户来决定 Windows 任务管理器（Task Manager）的 CPU 占用率。程序越精简越好，计算机语言不限。例如，可以实现下面三种情况：

1. CPU 的占用率固定在 50%，为一条直线；
2. CPU 的占用率为一条直线，但是具体占用率由命令行参数决定（参数范围 1~ 100）；
3. CPU 的占用率状态是一个正弦曲线。

分析与解法

有一名学生写了如下的代码：

```
while (true)
{
   if (busy)
      i++;
   else
}
```

然后她就陷入了苦苦思索：else 干什么呢？怎么才能让电脑不做事情呢？CPU 使用率为 0 的时候，到底是什么东西在用 CPU？另一名学生花了很多时间构想如何“深入内核，以控制 CPU 占用率”——可是事情真的有这么复杂么？

MSRA TTG（Microsoft Research Asia, Technology Transfer Group）的一些实习生写了各种解法，他们写的简单程序可以达到如图 1-1 所示的效果。

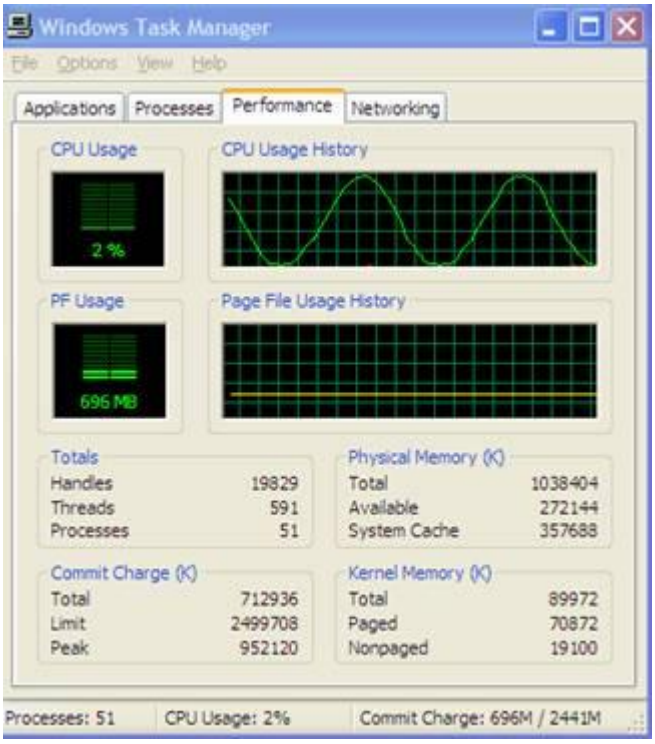


图 1-1 编码控制 CPU 占用率呈现正弦曲线形态

看来这并不 是不可能完成的任务。让我们仔细地回想一下写程序时曾经碰到的问题，如果我们不小心写了一个死循环，CPU 占用率就会跳到最高，并且一直保持 100%。我 们也可以打开任务管理器，实际观测一下它是怎样变动的。凭肉眼观察，它大约是 1 秒钟更新一次。一般情况下，CPU 使用率会很低。但是，当用户运行一个程 序，执行一些复杂操作的时候，CPU 的使用率会急剧升高。当用户晃动鼠标时，CPU 的使用率也有小幅度的变化。

那当任务管 理器报告 CPU 使用率为 0 的时候，谁在使用 CPU 呢？通过任务管理器的“进程（Process）”一栏可以看到，System Idle Process 占用了 CPU 空闲的时间——这时候大家该回忆起在“操作系统原理”这门课上学到的一些知识了吧。系统中有那么多进程，它们什么时候能“闲下 来”呢？答案很简单，这些程序或者在等待用户的输入，或者在等待某些事件的发生（WaitForSingleObject()），或者进入休眠状态（通过 Sleep()来实现）。

在任务 管理器的一个刷新周期内，CPU 忙（执行应用程序）的时间和刷新周期总时间的比率，就是 CPU 的占用率，也就是说，任务管理器中显示的是每个刷新周期内 CPU 占用率的统计平均值。因此，我们写一个程序，让它在任务管理器的刷新期间内一会儿忙，一会儿闲，然后通过调节忙/闲的比例，就可以控制任务管理器中 显示的 CPU 占用率。

【解法一】简单的解法

步骤 1 要操纵 CPU 的 usage 曲线，就需要使 CPU 在一段时间内（根据 Task Manager 的采样率）跑 busy 和 idle 两个不同的 loop，从而通过不同的时间 比例，来获得调节 CPU Usage 的效果。

步骤 2 Busy loop 可以通过执行空循环来实现，idle 可以通过 Sleep()来实现。

问题的关键在于如何控制两个 loop 的时间，方法有二：

Sleep 一段时间，然后以 for 循环 n 次，估算 n 的值。

那么对于一个空循环 for(i = 0; i < n; i++);又该如何来估算这个最合适的 n 值呢？我们都知道 CPU 执行的是机器指令，而最接近于机器指令的语言是汇编语言，所以我们可以先把这个空循环简单地写成如下汇编代码后再进行分析：

```
loop:
mov dx i        ;将 i 置入 dx 寄存器
inc dx          ;将 dx 寄存器加 1
mov i dx        ;将 dx 中的值赋回 i
cmp i n         ;比较 i 和 n
jl loop         ;i 小于 n 时则重复循环
```

假设这段代 码要运行的 CPU 是 P4 2.4Ghz（2.4 * 10 的 9 次方个时钟周期每秒）。现代 CPU 每个时钟周期可以执行两条以上的代码，那么我们就取平均值两条，于是让（2 400 000 000 * 2）/5=960 000 000（循环/秒），也就是说 CPU 1 秒钟可以运行这个空循环 960 000 000 次。不过我们还是不能简单地将 n = 60 000 000，然后 Sleep(1000)了事。如果我们让 CPU 工作 1 秒钟，然后休息 1 秒钟，波形很有可能就是锯齿状的——先达到一个峰值（大于>50%），然后跌到一个很低的占用率。

我们尝试着降低两个数量级，令 n = 9 600 000，而睡眠时间相应改为 10 毫秒（Sleep(10)）。用 10 毫秒是因为它不大也不小，比较接近 Windows 的调度时间片。如果选得太小（比如 1 毫秒），则会造成线程频繁地被唤醒和挂起，无形中又增加了内核时间的不确定性影响。最后我们可以得到如下代码：

代码清单 1-1

```
int main()
{
   for(;;)
   {
      for(int i = 0; i < 9600000; i++);
      Sleep(10);
   }
   return 0;
```

```
}
```

在不断调整 9 600 000 的参数后，我们就可以在一台指定的机器上获得一条大致稳定的 50% CPU 占用率直线。

使用这种方法要注意两点影响：

1. 尽量减少 sleep/awake 的频率，如果频繁发生，影响则会很大，因为此时优先级更高的操作系统内核调度程序会占用很多 CPU 运算时间。
2. 尽量不要调用 system call（比如 I/O 这些 privilege instruction），因为它也会导致很多不可控的内核运行时间。

该方法的缺点也很明显：不能适应机器差异性。一旦换了一个 CPU，我们又得重新估算 n 值。有没有办法动态地了解 CPU 的运算能力，然后自动调节忙/闲的时间比呢？请看下一个解法。

【解法二】使用 GetTickCount()和 Sleep()

我们知道 GetTickCount()可以得到“系统启动到现在”的毫秒值，最多能够统计到 49.7 天。另外，利用 Sleep()函数，最多也只能精确到 1 毫秒。因此，可以在“毫秒”这个量级做操作和比较。具体如下：

利用 GetTickCount()来实现 busy loop 的循环，用 Sleep()实现 idle loop。伪代码如下：

代码清单 1-2

```
int busyTime = 10; //10 ms

int idleTime = busyTime; //same ratio will lead to 50% cpu usage

Int64 startTime = 0;

while (true)

{

startTime = GetTickCount();

// busy loop 的循环

while ((GetTickCount() - startTime) <= busyTime) ;

    //idle loop

    Sleep(idleTime);

}
```

这两种解法都是假设目前系统上只有当前程序在运行，但实际上，操作系统中有很多程序都会在不同时间执行各种各样的任务，如果此刻其他进程使用了 10% 的 CPU，那我们的程序应该只能使用 40%的 CPU（而不是机械地占用 50%），这样可达到 50%的效果。

怎么做呢？

我们得知道“当前 CPU 占用率是多少”，这就要用到另一个工具来帮忙——Perfmon.exe。

Perfmon 是从 Windows NT 开始就包含在 Windows 服务器和台式机操作系统的管理工具组中的专业监视工具之一（如图 1-2 所示）。Perfmon 可监视各类系统计数器，获取 有关操作系统、应用程序和硬件的统计数字。Perfmon 的用法相当直接，只要选择您所要监视的对象（比如：处理器、RAM 或硬盘），然后选择所要监视的 计数器（比如监视物理磁盘对象时的平均队列长度）即可。还可以选择所要监视的实例，比如面对一台多 CPU 服务器时，可以选择监视特定的处理器。

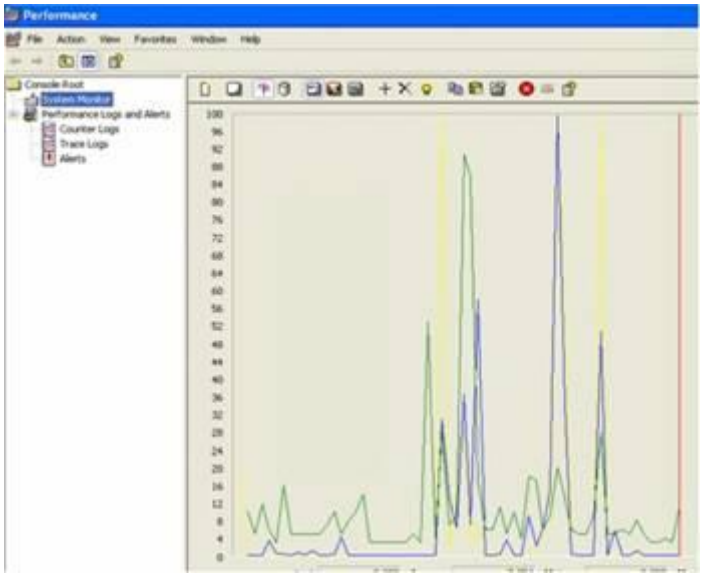


图 1-2 系统监视器（Perfmon）

我们可以写程序来查询 Perfmon 的值，Microsoft .Net Framework 提供了 PerformanceCounter()这一类型，从而可以方便地拿到当前各种计算机性能数据，包括 CPU 的使用率。例如下面这个程序——

【解法三】能动态适应的解法

代码清单 1-3

```
//C# code

static void MakeUsage(float level)

{
```

```
PerformanceCounter p = new PerformanceCounter("Processor", "% Proc
  essor Time", "_Total");
    while (true)
    {
       if (p.NextValue() > level)
          System.Threading.Thread.Sleep(10);
    }
}
```

可以看到，上面的解法能方便地处理各种 CPU 使用率参数。这个程序可以解答前面提到的问题 2。

有了前面的积累，我们应该可以让任务管理器画出优美的正弦曲线了，见下面的代码。

【解法四】正弦曲线

代码清单 1-4

```
//C++ code  to make task manager generate sine graph
#include "Windows.h"
#include "stdlib.h"
#include "math.h"
const double SPLIT = 0.01;
const int COUNT = 200;
const double PI = 3.14159265;
const int INTERVAL = 300;
int _tmain(int argc, _TCHAR* argv[])
{
   DWORD busySpan[COUNT];  //array of busy times
   DWORD idleSpan[COUNT];  //array of idle times
   int half = INTERVAL / 2;
```

```
    double radian = 0.0;
    for(int i = 0; i < COUNT; i++)
    {
        busySpan[i] = (DWORD)(half + (sin(PI * radian) * half));
        idleSpan[i] = INTERVAL - busySpan[i];
        radian += SPLIT;
    }
    DWORD startTime = 0;
    int j = 0;
    while (true)
    {
        j = j % COUNT;
        startTime = GetTickCount();
        while ((GetTickCount() - startTime) <= busySpan[j]) ;
        Sleep(idleSpan[j]);
        j++;
    }
    return 0;
}
```

讨论

如果机器是多 CPU，上面的程序会出现什么结果？如何在多个 CPU 时显示同样的状态？例如，在双核的机器上，如果让一个单线程的程序死循环，能让两个 CPU 的使用率达到 50%的水平么？为什么？

多 CPU 的问题首先需要获得系统的 CPU 信息。可以使用 GetProcessorInfo()获得多处理器的信息,然后指定进程在哪一个处理器上运行。其中指定运行使用的是 SetThreadAffinityMask()函数。

另外，还可以使用 RDTSC 指令获取当前 CPU 核心运行周期数。

在 x86 平台上定义函数：

```
inline __int64 GetCPUTickCount()
{
    __asm
    {
       rdtsc;
    }
}
```

在 x64 平台上定义：

```
#define GetCPUTickCount() __rdtsc()
```

使用 CallNtPowerInformation API 得到 CPU 频率，从而将周期数转化为毫秒数，例如：

代码清单 1-5

```
_PROCESSOR_POWER_INFORMATION info;
CallNTPowerInformation(11,  //query processor power information
    NULL,              //no input buffer
    0,                //input buffer size is zero
    &info,             //output buffer
    Sizeof(info));     //outbuf size
    __int64 t_begin = GetCPUTickCount();
    //do something
    __int64 t_end = GetCPUTickCount();
    double millisec = ((double)t_end –
       (double)t_begin)/(double)info.CurrentMhz;
```

RDTSC 指令读取当前 CPU 的周期数，在多 CPU 系统中，这个周期数在不同的 CPU 之间基数不同，频率也有可能不同。用从两个不同的 CPU 得到的周期数作计算会得 出没有意义的

值。如果线程在运行中被调度到了不同的 CPU，就会出现上述情况。可用 SetThreadAffinityMask 避免线程迁移。另外， CPU 的频率会随系统供电及负荷情况有所调整。

总结

能帮助你了解当前线程/进程/系统效能的 API 大致有以下这些：

1. Sleep()——这个方法能让当前线程“停”下来。
2. WaitForSingleObject()——自己停下来，等待某个事件发生
3. GetTickCount()——有人把 Tick 翻译成“嘀嗒”，很形象。
4. QueryPerformanceFrequency()、QueryPerformanceCounter()——让你访问到精度更高的 CPU 数据。
5. timeGetSystemTime()——是另一个得到高精度时间的方法。
6. PerformanceCounter——效能计数器。
7. GetProcessorInfo()/SetThreadAffinityMask()。遇到多核的问题怎么办呢？这两个方法能够帮你更好地控制 CPU。
8. GetCPUTickCount()。想拿到 CPU 核心运行周期数吗？用用这个方法吧。

编程之美 - 中国象棋将帅问题

下过中国象棋的朋友都知道，双方的“将”和“帅”相隔遥远，并且它们不能照面。在象棋残局中，许多高手能利用这一规则走出精妙的杀招。假设棋盘上只有“将”和“帅”二子（如图 1-3 所示）（为了下面叙述方便，我们约定用 A 表示“将”，B 表示“帅”）：

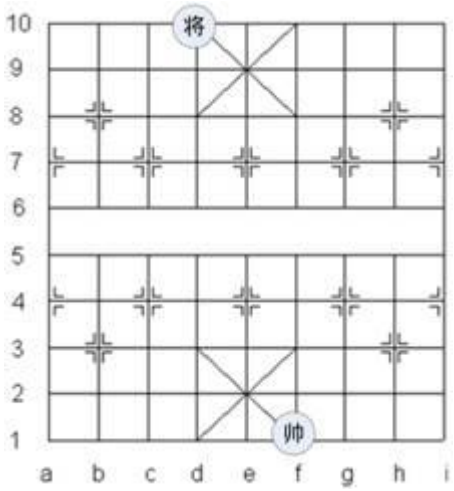


图 1-3

A、B 二子被限制在己方 3×3 的格子里运动。例如,在如上的表格里，A 被正方形{d_{10}，f_{10}，d_8，f_8}包围，而 B 被正方形{d_3，f_3，d_1，f_1}包围。每一步，A、B 分别可以横向或纵向移动一格，但不能沿对角线移动。另外，A 不能面对 B，也就是说，A 和 B 不能处于同一纵向直线上（比如 A 在 d_{10} 的位置，那么 B 就不能在 d_1、d_2 以及 d_3）。

请写出一个程序，输出 A、B 所有合法位置。要求在代码中只能使用一个变量。

分析与解法

问题的本身并不复杂，只要把所有 A、B 互相排斥的条件列举出来就可以完成本题的要求。由于本题要求只能使用一个变量，所以必须首先想清楚在写代码的时候，有哪些信息需要存储，并且尽量高效率地存储信息。稍微思考一下，可以知道这个程序的大体框架是：

遍历 A 的位置

遍历 B 的位置

判断 A、B 的位置组合是否满足要求。

如果满足，则输出。

因此，需要存储的是 A、B 的位置信息，并且每次循环都要更新。为了能够进行判断，首先需要创建一个逻辑的坐标系统，以便检测 A 何时会面对 B。这里我们想到的方法是用 1~9 的数字，按照行优先的顺序来表示每个格点的位置（如图 1-4 所示）。这样，只需要用模余运算就可以得到当前的列号，从而判断 A、B 是否互斥。

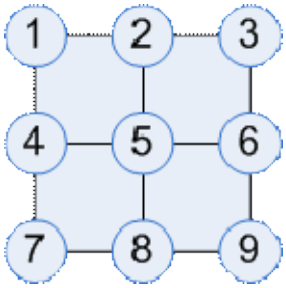


图 1-4 用 1~9 的数字表示 A、B 的坐标

第二，题目要求只用一个变量，但是我们却要存储 A 和 B 两个子的位置信息，该怎么办呢？

可以先把已知变量类型列举一下，然后做些分析。

对于 bool 类型，估计没有办法做任何扩展了，因为它只能表示 true 和 false 两个值；而 byte 或者 int 类型，它们能够表达的信息则更多。事实上，对本题来说，每个子都只需要 9 个数字就可以表达它的全部位置。

一个 8 位的 byte 类型能够表达 2^8=256 个值，所以用它来表示 A、B 的位置信息绰绰有余，因此可以把这个字节的变量（设为 b）分成两部分。用前面的 4 bit 表示 A 的位置，用后面的 4 bit 表示 B 的位置，那么 4 个 bit 可以表示 16 个数，这已经足够了。

问题在于：如何使用 bit 级的运算将数据从这一 byte 变量的左边和右边分别存入和读出。

下面是做法：

n 将 byte b（10100101）的右边 4 bit（0101）设为 n（0011）：

首先清除 b 右边的 bits，同时保持左边的 bits:

11110000（LMASK）

& 10100101（b）

10100000

然后将上一步得到的结果与 n 做或运算

10100000（LMASK & b）

^ 00000011（n）

10100011

n 将 byte b（10100101）左边的 4 bit（1010）设为 n（0011）：

首先，清除 b 左边的 bits，同时保持右边的 bits:

00001111（RMASK）

& 10100101（b）

00000101

现在，把 n 移动到 byte 数据的左边

n << 4 = 00110000

然后对以上两步得到的结果做或运算，从而得到最终结果。

00000101（RMASK & b）

^ 00110000（n << 4）

00110101

n 得到 byte 数据的右边 4 bits 或左边 4 bits（e.g. 10100101 中的 1010 以及 0101）：

清除 b 左边的 bits，同时保持右边的 bits

00001111（RMASK）

& 10100101（b）

00000101

清除 b 的右边的 bits，同时保持左边的 bits

11110000（LMASK）

& 10100101（b）

10100000

将结果右移 4 bits

10100000 >> 4 = 00000101

最后的挑战是如何在不声明其他变量约束的前提下创建一个 for 循环。可以重复利用 1 byte 的存储单元，把它作为循环计数器并用前面提到的存取和读入技术进行操作。还可以用宏来抽象化代码，例如：

for (LSET(b, 1); LGET(b) <= GRIDW * GRIDW; LSET(b, (LGET(b) + 1)))

【解法一】

代码清单 1-6

```
#define HALF_BITS_LENGTH 4

// 这个值是记忆存储单元长度的一半，在这道题里是 4bit

#define FULLMASK 255

// 这个数字表示一个全部 bit 的 mask，在二进制表示中，它是 11111111。

#define LMASK (FULLMASK << HALF_BITS_LENGTH)

// 这个宏表示左 bits 的 mask，在二进制表示中，它是 11110000。

#define RMASK (FULLMASK >> HALF_BITS_LENGTH)

// 这个数字表示右 bits 的 mask，在二进制表示中，它表示 00001111。

#define RSET(b, n) (b = ((LMASK & b) ^ n))

// 这个宏，将 b 的右边设置成 n

#define LSET(b, n) (b = ((RMASK & b) ^ (n << HALF_BITS_LENGTH)))

// 这个宏，将 b 的左边设置成 n

#define RGET(b) (RMASK & b)

// 这个宏得到 b 的右边的值

#define LGET(b) ((LMASK & b) >> HALF_BITS_LENGTH)

// 这个宏得到 b 的左边的值

#define GRIDW 3
```

```
// 这个数字表示将帅移动范围的行宽度。

#include <stdio.h>

#define HALF_BITS_LENGTH 4

#define FULLMASK 255

#define LMASK (FULLMASK << HALF_BITS_LENGTH)

#define RMASK (FULLMASK >> HALF_BITS_LENGTH)

#define RSET(b, n) (b = ((LMASK & b) ^ n))

#define LSET(b, n) (b = ((RMASK & b) ^ (n << HALF_BITS_LENGT
H)))

#define RGET(b) (RMASK & b)

#define LGET(b) ((LMASK & b) >> HALF_BITS_LENGTH)

#define GRIDW 3

int main()

{

   unsigned char b;

   for(LSET(b, 1); LGET(b) <= GRIDW * GRIDW; LSET(b, (LGET(b) +
1)))

      for(RSET(b, 1); RGET(b) <= GRIDW * GRIDW; RSET(b, (RGET(b)
+ 1)))

         if(LGET(b) % GRIDW != RGET(b) % GRIDW)

            printf("A = %d, B = %d\n", LGET(b), RGET(b));

return 0;

}
```

【输出】

格子的位置用 N 来表示，N = 1，2，...，8，9，依照行优先的顺序，如图 1-5 所示：

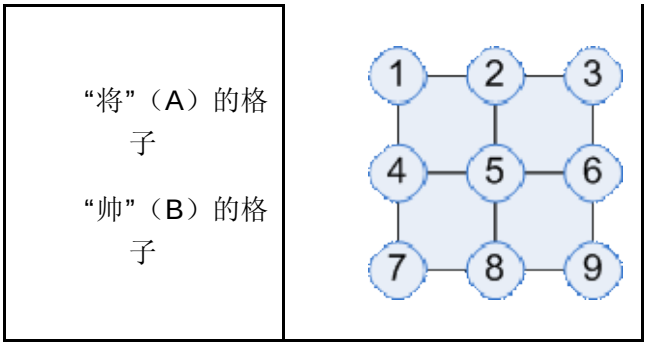


图 1-5

A = 1, B = 2	A = 4, B = 2	A = 7, B = 2
A = 1, B = 3	A = 4, B = 3	A = 7, B = 3
A = 1, B = 5	A = 4, B = 5	A = 7, B = 5
A = 1, B = 6	A = 4, B = 6	A = 7, B = 6
A = 1, B = 8	A = 4, B = 8	A = 7, B = 8
A = 1, B = 9	A = 4, B = 9	A = 7, B = 9
A = 2, B = 1	A = 5, B = 1	A = 8, B = 1
A = 2, B = 3	A = 5, B = 3	A = 8, B = 3
A = 2, B = 4	A = 5, B = 4	A = 8, B = 4
A = 2, B = 6	A = 5, B = 6	A = 8, B = 6
A = 2, B = 7	A = 5, B = 7	A = 8, B = 7
A = 2, B = 9	A = 5, B = 9	A = 8, B = 9
A = 3, B = 1	A = 6, B = 1	A = 9, B = 1
A = 3, B = 2	A = 6, B = 2	A = 9, B = 2
A = 3, B = 4	A = 6, B = 4	A = 9, B = 4
A = 3, B = 5	A = 6, B = 5	A = 9, B = 5
A = 3, B = 7	A = 6, B = 7	A = 9, B = 7
A = 3, B = 8	A = 6, B = 8	A = 9, B = 8

考虑了这么多因素，总算得到了本题的一个解法，但是 MSRA 里却有人说，下面的一小段代码也能达到同样的目的：

```
BYTE i = 81;
while(i--)
{
    if(i / 9 % 3 == i % 9 % 3)
        continue;
    printf("A = %d, B = %d\n", i / 9 + 1, i % 9 + 1);
}
```

但是很快又有另一个人说他的解法才是效率最高的：

代码清单 1-7

```
struct {
    unsigned char a:4;
    unsigned char b:4;
} i;
for(i.a = 1; i.a <= 9; i.a++)
for(i.b = 1; i.b <= 9; i.b++)
    if(i.a % 3 == i.b % 3)
        printf("A = %d, B = %d\n", i.a, i.b);
```

编程之美 - 一摞烙饼的排序

星期五的晚上，一帮同事在希格玛大厦附近的“硬盘酒吧”多喝了几杯。程序员多喝了几杯之后谈什么呢？自然是算法问题。有个同事说：

“我以前在餐馆打工，顾客经常点非常多的烙饼。店里的饼大小不一，我习惯在到达顾客饭桌前，把一摞饼按照大小次序摆好——小的在上面，大的在下面。由于我一只手托着盘

子，只好用另一只手，一次抓住最上面的几块饼，把它们上下颠倒个个儿，反复几次之后，这摞烙饼就排好序了。

我后来想，这实际上是个有趣的排序问题：假设有 n 块大小不一的烙饼，那最少要翻几次，才能达到最后大小有序的结果呢？

你能否写出一个程序，对于 n 块大小不一的烙饼，输出最优化的翻饼过程呢？

分析与解法

这个排序问题非常有意思，首先我们要弄清楚解决问题的关键操作——“单手每次抓几块饼，全部颠倒”。

具体参看图 1-6：

图 1-6 烙饼的翻转过程

每次我们只能选择最上方的一堆饼，一起翻转。而不能一张张地直接抽出来，然后进行插入，也不能交换任意两块饼子。这说明基本的排序办法都不太好用。那么怎么把这 n 个烙饼排好序呢？

由于每次操作都是针对最上面的饼，如果最底层的饼已经排序，那我们只用处理上面的 n-1 个烙饼。这样，我们可以再简化为 n-2、n-3，直到最上面的两个饼排好序。

【解法一】

我们用图 1-7 演示一下，为了把最大的烙饼摆在最下面，我们先把最上面的烙饼和最大的烙饼之间的烙饼翻转（1~4 之间），这样，最大的烙饼就在最上面了。接着，我们把所有烙饼翻转（4~5 之间），最大的烙饼就摆在最下面了。

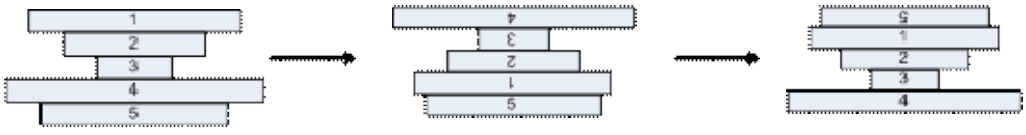


图 1-7 两次翻转烙饼，调整最大的烙饼到最底端

之后，我们对上面 n-1、n-2 个饼重复这个过程就可以了。

那么，我们一共需要多少次翻转才能把这些烙饼给翻转过来呢？

首先，经过两次翻转可以把最大的烙饼翻转到最下面。因此，最多需要把上面的 n-1 个烙饼依次翻转两次。那么，我们至多需要 2（n-1）次翻转就可以把所有烙饼排好序（因为第二小的烙饼排好的时候，最小的烙饼已经在最上面了）。

这样看来，单手翻转的想法是肯定可以实现的。我们进一步想想怎么减少翻转烙饼的次数吧。

怎样才能通过程序来搜索到一个最优的方案呢？

首先，通过每次找出最大的烙饼进行翻转是一个可行的解决方案。那么，这个方案是最好的一个吗？考虑这样一种情况，假如这堆烙饼中有好几个不同的部分相对有序，凭直觉来猜想，我们可以先把小一些的烙饼进行翻转，让其有序。这样会比每次翻转最大的烙饼要更快。

既然如此，有类似的方案可以达到目的吗？比如说，考虑每次翻转的时候，把两个本来应该相邻在烙饼尽可能地换到一起。这样，当所有的烙饼都换到一起之后，实际上就是完成排序了。（从这个意义上来说，每次翻最大烙饼的方案实质上就是每次把最大的和次大的交换到一起。）

在这样的基础之上，本能的一个想法就是穷举。只要穷举出所有可能的交换方案，那么，我们一定能够找到一个最优的方案。

沿着这个思路去考虑，我们自然就会使用动态规划或者递归的方法来进行实现了。可以从不同的翻转策略开始，比如说第一次先翻最小的，然后递归把所有的可能全部翻转一遍。这样，最终肯定是可以找到一个解的。

但是，既然是递归就一定有退出的条件。在这个过程中，第一个退出的条件肯定是所有的烙饼已经排好序。那么，有其他的吗？如果大家仔细想想就会发现到，既然 2（n-1）是一个最多的翻转次数。如果在算法中，需要翻转的次数多于 2（n-1），那么，我们就应该放弃这个翻转算法，直接退出。这样，就能够减少翻转的次数。

从另外一个层面上来说，既然这是一个排序问题。我们也应该利用到排序的信息来进行处理。同样，在翻转的过程中，我们可以看看当前的烙饼数组的排序情况如何，然后利用这些信息来帮助减少翻转次数的判断过程。

下面是在前面讨论的基础之上形成的一个粗略的搜索最优方案的程序：

代码清单 1-8

```
#include <stdio.h>

/*************************************************************
*************/

//

// 烙饼排序实现

//

/*************************************************************
*************/
```

```
class CPrefixSorting
{
public:
    CPrefixSorting()
    {
        m_nCakeCnt = 0;
        m_nMaxSwap = 0;
    }
    //
    // 计算烙饼翻转信息
    // @param
    // pCakeArray   存储烙饼索引数组
    // nCakeCnt 烙饼个数
    //
    void Run(int* pCakeArray, int nCakeCnt)
    {
        Init(pCakeArray, nCakeCnt);
        m_nSearch = 0;
        Search(0);
    }
    //
    // 输出烙饼具体翻转的次数
    //
void Output()
```

```
{
    for(int i = 0; i < m_nMaxSwap; i++)
    {
        printf("%d ", m_arrSwap[i]);
    }

    printf("\n |Search Times| : %d\n", m_nSearch);
    printf("Total Swap times = %d\n", m_nMaxSwap);
}
private:
    //
    // 初始化数组信息
    // @param
    // pCakeArray   存储烙饼索引数组
    // nCakeCnt 烙饼个数
    //
    void Init(int* pCakeArray, int nCakeCnt)
    {
        Assert(pCakeArray != NULL);
        Assert(nCakeCnt > 0);
        m_nCakeCnt = n;
    // 初始化烙饼数组
        m_CakeArray = new int[m_nCakeCnt];
        Assert(m_CakeArray != NULL);
```

```
    for(int i = 0; i < m_nCakeCnt; i++)
    {
        m_CakeArray[i] = pCakeArray[i];
    }
    // 设置最多交换次数信息
    m_nMaxSwap = UpBound(m_nCakeCnt);
    // 初始化交换结果数组
    m_SwapArray = new int[m_nMaxSwap];
    Assert(m_SwapArray != NULL);
    // 初始化中间交换结果信息
    m_ReverseCakeArray = new int[m_nCakeCnt];
    for(i = 0; i < m_nCakeCnt; i++)
    {
        m_ReverseCakeArray[i] = m_CakeArray[i];
    }
    m_ReverseCakeArraySwap = new int[m_nMaxSwap];
}



//
// 寻找当前翻转的上界
//
//
int UpBound(int nCakeCnt)
```

```
{
    return nCakeCnt*2;
}
//
// 寻找当前翻转的下界
//
//
int LowerBound(int* pCakeArray, int nCakeCnt)
{
    int t, ret = 0;
    // 根据当前数组的排序信息情况来判断最少需要交换多少次
    for(int i = 1; i < nCakeCnt; i++)
    {
        // 判断位置相邻的两个烙饼，是否为尺寸排序上相邻的
        t = pCakeArray[i] - pCakeArray[i-1];
        if((t == 1) || (t == -1))
        {
        }
        else
        {
            ret++;
        }
    }
    return ret;
```

```
}
// 排序的主函数
void Search(int step)
{
    int i, nEstimate;
    m_nSearch++;
    // 估算这次搜索所需要的最小交换次数
    nEstimate = LowerBound(m_ReverseCakeArray, m_nCakeCnt);
    if(step + nEstimate > m_nMaxSwap)
        return;
    // 如果已经排好序，即翻转完成，输出结果
    if(IsSorted(m_ReverseCakeArray, m_nCakeCnt))
    {
        if(step < m_nMaxSwap)
        {
            m_nMaxSwap = step;
            for(i = 0; i < m_nMaxSwap; i++)
                m_arrSwap[i] = m_ReverseCakeArraySwap[i];
        }
        return;
    }
    // 递归进行翻转
    for(i = 1; i < m_nCakeCnt; i++)
    {
```

```
            Revert(0, i);
            m_ReverseCakeArraySwap[step] = i;
            Search(step + 1);
            Revert(0, i);
        }
    }
    //
    // true : 已经排好序
    // false : 未排序
    //
    bool IsSorted(int* pCakeArray, int nCakeCnt)
    {
        for(int i = 1; i < nCakeCnt; i++)
        {
            if(pCakeArray[i-1] > pCakeArray[i])
            {
                return false;
            }
        }
        return true;
    }
    //
    // 翻转烙饼信息
    //
```

```
        void Revert(int nBegin, int nEnd)
        {
        Assert(nEnd > nBegin);
           int i, j, t;
           // 翻转烙饼信息
           for(i = nBegin, j = nEnd; i < j; i++, j--)
           {
              t = m_ReverseCakeArray[i];
              m_ReverseCakeArray[i] = m_ReverseCakeArray[j];
              m_ReverseCakeArray[j] = t;
           }
        }
    private:
        int* m_CakeArray;   // 烙饼信息数组
        int m_nCakeCnt;     // 烙饼个数
        int m_nMaxSwap;     // 最多交换次数。根据前面的推断，这里最多为 m_nCakeCnt * 2
        int* m_SwapArray;   // 交换结果数组
        int* m_ReverseCakeArray;        // 当前翻转烙饼信息数组
        int* m_ReverseCakeArraySwap;    // 当前翻转烙饼交换结果数组
        int m_nSearch;                  // 当前搜索次数信息
    };
```

当烙饼不多的时候，我们已经可以很快地找出最优的翻转方案。

我们还有办法来对这个程序进行优化，使得能更快地找出最优方案吗？

我们已 经知道怎么构造一个可行的翻转方案，所以最优的方案肯定不会比这个差。这个就是我们程序中的上界（UpperBound），就是说，我们感兴趣的最优方案 最差也就是我

们刚构造出来的方案了。如果我们能够找到一个更好的构造方案，我们的搜索空间就会继续缩小，因为我们一开始就设 m_nMinSwap 为 UpperBound，而程序中有一个剪枝：

```
nEstimate = LowerBound(m_tArr, m_n);

if(step + nEstimate > m_nMinSwap)

   return;
```

m_nMinSwap 越小，那么这个剪枝条件就越容易满足，更多的情况就不需要再去搜索。当然，程序也就能更快地找出最优方案。

仔细分析上 面的剪枝条件，在到达 m_tArr 状态之前，我们已经翻转了 step 次，nEstimate 是在当前这个状态我们至少还要翻转多少次才能成功的次数。如果 step+nEstimate 大于 m_nMinSwap，也就说明从当前状态继续下去，m_nMinSwap 次我们也不能排好所有烙饼。那么，当然就没有 必要再继续了。因为继续下去得到的方案不可能比我们已经找到的好。

显然，如果 nEstimate 越大，剪枝条件越容易被满足。而这正是我们希望的。

结合上面两点，我们希望 UpperBound 越小越好，而下界（LowerBound）越大越好。假设如果有神仙指点，你只要告诉神仙你当前的状态，他就能告诉你最少需要多少次翻转。这样的话，我们可以花费 O（N^2）的时间得到最优的方案。但是，现实中，没有这样的神仙。我们只能尽可能地减小 UpperBound，增加 LowerBound，从而减少需要搜索的空间。

利用上面的程序，做一个简单的比较。

对于一个输入，10 个烙饼，从上到下，烙饼半径分别为 3，2，1，6，5，4，9，8，7，0。对应上面程序的输入为：

10

3 2 1 6 5 4 9 8 7 0

如果 LowerBound 在任何状态都为 0，也就是我们太懒了，不想考虑那么多。当然任意状态下，你至少需要 0 次翻转才能排好序。这样，上面的程序 Search 函数被调用了 575 225 200 次。

但是如果把 LowerBound 稍微改进一下（如上面程序中所计算的方法估计），程序则只需要调用 172 126 次 Search 函数便可以得到最优方案：

6

4 8 6 8 4 9

程序中的下界是怎么估计出来的呢？

每一次翻转我们最多使得一个烙饼与大小跟它相邻的烙饼排到一起。如果当前状态 n 个烙饼中，有 m 对相邻的烙饼它们的半径不相邻，那么我们至少需要 m 次才能排好序。

从上面的例子，大家都会发现改进上界和下界，好处可不少。我想不用多说，大家肯定想继续优化上界和下界的估计吧。

除了上界和下界的改进，还有什么办法可以提高搜索效率吗？如果我们翻了若干次之后，又回到一个已经出现过的状态，我们还值得继续从这个状态开始搜索吗？我们怎样去检测一个状态是否出现过呢？

读者也许不相信，比尔盖茨在上大学的时候也研究过这个问题，并且发表过论文。你不妨跟盖茨的结果比比吧。

扩展问题

3. 有一些服务员会把上面的一摞饼子放在自己头顶上（放心，他们都戴着洁白的帽子），然后再处理其他饼子，在这个条件下，我们的算法能有什么改进？

4. 事实上，饭店的师傅经常把烙饼烤得一面非常焦，另一面则是金黄色。这时，服务员还得考虑让烙饼大小有序，并且金黄色的一面都要向上。这样要怎么翻呢？

5. 有一次师傅烙了三个饼，一个两面都焦了，一个两面都是金黄色，一个一面是焦的，一面是金黄色，我把它们摞一起，只能看到最上面一个饼的上面，发现是焦的，问最上面这个饼的另一面是焦的概率是多少？

6. 每次翻烙饼的时候，上面的若干个烙饼会被翻转。如果我们希望在排序过程中，翻转烙饼的总个数最少，结果会如何呢？

7. 对于任意次序的 n 个饼的排列，我们可以研究把它们完全排序需要大致多少次翻转，目前的研究成果是：

（1） 目前找到的最大的下界是 15n/14，就是说，如果有 100 个烙饼，那么我们至少需要 15×100/14= 108 次翻转才能把烙饼翻好——而且具体如何翻还不知道。

（2） 目前找到的最小的上界是（5n + 5）/3，对于 100 个烙饼，这个上界是 169。

（3） 任意次序的 n 个烙饼反转排序所需的最小反转次数被称为第 n 个烙饼数，现在找到的烙饼数为：

N	1	2	3	4	5	6	7	8	9	10	
P_n	0	1	3	4	5	7	8	9	10	11	

第 14 个烙饼数 P_{14} 还没有找到，读者朋友们，能否在吃烙饼之余考虑一下这个问题？

1.4 编程之美 - 买书问题

在 节假日的时候，书店一般都会做促销活动。由于《哈利波特》系列相当畅销，店长决定通过促销活动来回馈读者。在销售的《哈利波特》平装本系列中，一共有五 卷，用编号 0，1，2，3，4 来表示。假设每一卷单独销售均需要 8 欧元。如果读者一次购买不同的两卷，就可以扣除 5%的费用，三卷则更多。假设具体折扣的情况如下：

本数	折扣
2	5%
3	10%
4	20%
5	25%

在一份订单中，根据购买的卷数以及本书，就会出现可以应用不同折扣规则的情况。但是，一本书只会应用一个折扣规则。比如，读者一共买了两本卷一，一本卷二。那么，可以享受到 5%的折扣。另外一本卷一则不能享受折扣。如果有多种折扣，希望能够计算出的总额尽可能的低。

要求根据这样的需求，设计出算法，能够计算出读者所购买一批书的最低价格。

分析与解法

怎么购买比较省钱呢？第一个感觉，当然是先优先考虑最大折扣，然后再次之。

这的确是一个有效的办法。那么这个算法是不是最省钱的呢？如果我们要买两本第一卷，两本第二卷，两本第三卷，一本第四卷，一本第五卷。

按照优先考 虑最大折扣的策略，我们先买各卷各一本，花去 5×8×（1-25%）=30，再买第一，二，三卷，花去 3×8×（1-10%）=21.6，总共 51.6 欧 元。但是如果我们换一个策略，先买第一、二、三、四卷，然后再买第一、二、三、五卷，那么总共花去 2×（4×8×（1-20%）=51.2 欧元。

从上面我们可以发现，这些折扣有一定的规律。在同样是买 8 本书的情况下，选择分别按照 3 本和 5 本，以及按照两个 4 本的付法，费用会不一样。也就是说，针对这个问题试图用贪心策略行不通。

【解法一】

那么针对这种问题的解法，动态规划是一个不错的选择。那么，不妨试一下动态规划的方法。

首先，在使用动态规划之前，得考虑怎么表达购买中间出现的状态。假设我们用 X_n 来表示购买第 n 卷书籍的数量。如果要买 X_1 本第一卷，X_2 本第二卷，X_3 本第三卷，X_4 本第四卷，X_5 本第五卷，那么我们可以用（X_1，X_2，X_3，X_4，X_5）表示我们要买的书，而 F（X_1，X_2，X_3，X_4，X_5）表示我们要买这些书需要的最少花费。

如果我们要买 X_3 本第一卷，X_2 本第二卷，X_1 本第三卷，X_4 本第四卷，X_5 本第五卷呢？是否需要用 F（X_3，X_2，X_1，X_4，X_5）来表示呢？其实不难看出，因为各卷的价格一样，需要的最少花费仍然等于 F（X_1，X_2，X_3，X_4，X_5）。也就是说，F（X_1，X_2，X_3，X_4，X_5）等价于 F（X_3，X_2，X_1，X_4，X_5）。因此我们没有必要区分不同的卷。那么对于所有跟（X_1，X_2，X_3，X_4，X_5）等价的情况，我们用什么来表示呢？F（X_1，X_2，X_3，X_4，X_5）还是 F（X_1，X_2，X_3，X_5，X_4）？还是……

根据排列组合的规则，最多有 5!种可选择的表示方法，我们可以选择一种特别的表示（Y_1，Y_2，Y_3，Y_4，Y_5）（其中，Y_n 用来表示购买第 n 卷书籍的数量，Y_1，Y_2，Y_3，Y_4，Y_5 是 X_1，X_2，X_3，X_4，X_5 的重新排列，满足 $Y_1>=Y_2>=Y_3>=Y_4>=Y_5$），我们称它为所有跟（X_1，X_2，X_3，X_4，X_5）等价的情况的“最小表示”。

接下来，就需要考虑怎么把一个大问题转化为小一点的问题。

假定要买的书为（Y_1，Y_2，Y_3，Y_4，Y_5）。如果第一次考虑为 5 本不同卷付钱（当然这里需要保证 $Y_5>=1$），那么剩下还要再付钱的书集合为（Y_1-1，Y_2-1，Y_3-1，Y_4-1，Y_5-1）。如果第一次考虑买 4 本不同卷（$Y_4>=1$），那么我们接下来还需要买的书集合为（Y_1-1，Y_2-1，Y_3-1，Y_4-1，Y_5）。

根据题意，只要是不同卷的书组合起来就能享受折扣，至于具体是哪几卷，并不要求。因此，就可以不用考虑其他可能比如（Y_1-1，Y_2-1，Y_3-1，Y_4，Y_5-1）。

其他同理，根据如上的推理可以得到状态转移方程：

F（Y_1，Y_2，Y_3，Y_4，Y_5）

=

0 if（$Y_1 = Y_2 = Y_3 = Y_4 = Y_5 = 0$）

= min {

5 × 8 ×（1 – 25%）+ F（Y_1-1，Y_2-1，Y_3-1，Y_4-1，Y_5-1），if（Y_5 >= 1）

4 × 8 ×（1 – 20%）+ F（Y_1-1，Y_2-1，Y_3-1，Y_4-1，Y_5），if（Y_4 >= 1）

3 × 8 ×（1 – 10%）+ F（Y_1-1，Y_2-1，Y_3-1，Y_4，Y_5），if（Y_3 >= 1）

2 × 8 ×（1 – 5%）+ F（Y_1-1，Y_2-1，Y_3，Y_4，Y_5），if（Y_2 >= 1）

8 + F（Y_1-1，Y_2，Y_3，Y_4，Y_5）if（Y_1 >= 1）

}

状态转化之后得到的（Y_1-1，Y_2-1，Y_3-1，Y_4-1，Y_5）等可能不是“最小表示”，我们需要把它们转化为对应的最小表示。比如：

```
F（2，2，2，2，2）

= min {

        5 × 8 ×（1 – 25%）+ F（1，1，1，1，1），

        4 × 8 ×（1 – 20%）+ F（1，1，1，1，2），          /* 这里不是最小表示 */

        3 × 8 ×（1 – 10%）+ F（1，1，1，2，2），

        2 × 8 ×（1 – 5%） + F（1，1，2，2，2），

        8 + F（1，2，2，2，2）

 }

= min {

        5 × 8 ×（1 – 25%）+ F（1，1，1，1，1），

        4 × 8 ×（1 – 20%）+ F（2，1，1，1，1），          /* 转换为最小表示 */

        3 × 8 ×（1 – 10%）+ F（2，2，1，1，1），

        2 × 8 ×（1 – 5%）+ F（2，2，2，1，1），

        8 + F（2，2，2，2，1）

 }
```

从上面的表示公式中可以看出，整个动态规划的算法需要耗费 $O(Y_1 \times Y_2 \times Y_3 \times Y_4 \times Y_5)$ 的空间来保存状态的值，所需要的时间复杂度也为 $O(Y_1 \times Y_2 \times Y_3 \times Y_4 \times Y_5)$。

【解法二】

对于上面的算法，时间复杂度仍然很大。那么，对于这个问题还有没有别的办法呢？贪心策略是否有办法成功呢？

该贪心策略会在买 5+3 本的时候出错。因为

$5 \times 8 \times (1\text{-}25\%) + 3 \times 8 \times (1\text{-}10\%) > 4 \times 8 \times (1\text{-}20\%) + 4 \times 8 \times (1\text{-}20\%)$。实际上就是说，在购买 8 本书的情况下：$5 \times 25\% + 3 \times 10\% < 8 \times 20\%$。

回过头对比一下：

在小于 5 本的情况下，直接按折扣买就好了：

2	5%
3	10%
4	20%
5	25%

那么如果大于 5 本呢。

首先，对于大于 5 本的情况，我们不应该考虑按一本付钱的情况，因为没有折扣。

本数	可能的分解本数	对应的折扣
6	= 4+2	4×20%+2×5% = 1.1
	= 3+3	3×10%+3×10% = 0.6
	= 2+2+2	6×5% = 0.3

注 明：上面的分解并不适用于所有情况。我们仅考虑的是理论上可能的最大分法，对于不能进行分解的情况，比如购买 6 本卷一，没有办法享受折扣，因此其折扣将小 于上述的讨论情况。同样地，对于购买 5 本卷一，1 本卷二的情况，最多只能享受一种折扣，其折扣也会小于上述的分解情况。对于以下的分解情况，同样适用。

本数	可能的分解本数	对应的折扣
7	= 5+2	5×25%+2×5% = 1.35
	= 4+3	4×20%+3×10% = 1.1
8	= 5+3	5×25%+3×10% = 1.55
	= 4+4	4×20%+4×20% = 1.6
	= 3+3+2	6×10%+2×5% = 0.7
	= 2+2+2+2	8×5% = 0.4
9	= 5+4	5×25% + 4×20% = 2.05

= 5+2+2 5×25%+4×5% = 1.4
5

= 4+3+2 4×20%+3×10%+2×
5% = 1.2

= 3+3+3 9×10% = 0.9

10 = 5+5 10×25% = 2.5

= 4+4+2 8×20%+2×5% = 1.7

= 4+3+3 4×20%+6×10% = 1.
4

= 2+2+2+2+2 10×5% = 0.5

由于折扣的规则仅针对 2 到 5 本的情况。对于大于 10 本的情况，我们可以提出如下的一个假设：

假设在分解的过程中，可以找到如下的一种分法：可以把 10 本以上的书籍分成小于 10 的多组（X_{11}，X_{12}，X_{13}，X_{14}，X_{15}），（X_{21}，X_{22}，X_{23}，X_{24}，X_{25}）…（X_{n1}，X_{n2}，X_{n3}，X_{n4}，X_{n5}），并且使得只要把每组的最优解相加，就可以得到全局的最优解。

那么，对于大于 10 的情况，都可以分解为小于 10 的情况。

假设这样的分法存在，那么就可以按照小于 10 的情况考虑求解。如果要买的书为（Y_1，Y_2，Y_3，Y_4，Y_5）（其中 $Y_1>=Y_2>=Y_3>=Y_4>=Y_5$），贪心策略建议我们买 Y_5 本五卷的，Y_4-Y_5 本四卷，Y_3-Y_4 本三卷，Y_2-Y_3 本两卷和 Y_1-Y_2 本一卷。由于贪心策略的反例，我们把 K 本五卷和 K 本三卷重新组合成 2×K 本四卷（K = min{Y5, Y3-Y4}）。结果就是我们买 Y5-K 本五卷的，Y_4-Y_5 本四卷，Y_3-Y_4-K 本三卷，Y_2-Y_3 本两卷和 Y_1-Y_2 本一卷（K = min{Y_5, Y_3-Y_4}）。比如我们要买 3 本第一卷，2 本第二卷，6 本第三卷，1 本第四卷和 7 本第五卷，像前面所说的，我们要买的书可以用（7，6，3，2，1）表示。新的经过调整的贪心策略告诉我们应该买三本四卷，三本两卷和一本一卷。

那么这种分法是正确的吗？有办法证明或者找到反例吗？读者可以直接和我们联系

编程之美 - 饮料供货

在微软亚洲研究院上班，大家早上来的第一件事是干啥呢？查看邮件？No，是去水房拿饮料：酸奶，豆浆，绿茶、王老吉、咖啡、可口可乐……（当然，还是有很多同事把拿饮料当做第二件事）。

管理水房的阿姨们每天都会准备很多的饮料给大家，为了提高服务质量，她们会统计大家对每种饮料的满意度。一段时间后，阿姨们已经有了大批的数据。某天早上，当实习生小

飞第一个冲进水房并一次拿了五瓶酸奶、四瓶王老吉、三瓶鲜橙多时，阿姨们逮住了他，要他帮忙。

从阿姨们统计的数据中，小飞可以知道大家对每一种饮料的满意度。阿姨们还告诉小飞，STC（Smart Tea Corp.）负责给研究院供应饮料，每天总量为 V。STC 很神奇，他们提供的每种饮料之单个容量都是 2 的方幂，比如王老吉，都是 2^3=8 升的，可乐都是 2^5=32 升的。当然 STC 的存货也是有限的，这会是每种饮料购买量的上限。统计数据中用饮料名字、容量、数量、满意度描述每一种饮料。

那么，小飞如何完成这个任务，求出保证最大满意度的购买量呢？

分析与解法

【解法一】

我们先把这个问题“数学化”一下吧。

假设 STC 共提供 n 种饮料，用（S_i、V_i、C_i、H_i、B_i）（对应的是饮料名字、容量、可能的最大数量、满意度、实际购买量）来表示第 i 种饮料（i = 0, 1,..., n-1），其中可能的最大数量指如果仅买某种饮料的最大可能数量，比如对于第 i 中饮料 $C_i=V/V_i$。

基于如上公式：

饮料总容量为$\sum_{i=0}^{n-1}(V_i * B_i)$；

总满意度为$\sum_{i=0}^{n-1}(H_i * B_i)$；

那么题目的要求就是，在满足条件$\sum_{i=0}^{n-1}(V_i * B_i)$=V 的基础上，求解 max{$\sum_{i=0}^{n-1}(H_i * B_i)$}。

对于求最优化的问题，我们来看看动态规划能否解决。用 Opt（V', i）表示从第 i, i+1, i+2, ..., n-1, n 种饮料中，算出总量为 V'的方案中满意度之和的最大值。

因此，Opt（V, n）就是我们要求的值。

那么，我们可以列出如下的推导公式：Opt （V', i） = max { k* H_i + Opt（V' - V_i * k, i-1）}（k = 0, 1, ..., C_i, i =0, 1, ..., n-1）。

即：最优化的结果 = 选择第 k 种饮料×满意度+减去第 k 种饮料×容量的最优化结果根据这样的推导公式，我们列出如下的初始边界条件：

Opt（0, n）= 0，即容量为 0 的情况下，最优化结果为 0。

Opt（x, n）= -INF（x != 0）（－INF 为负无穷大），即在容量不为 0 的情况下，把最优化结果设为负无穷大，并把它作为初值。

那么，根据以上的推导公式，就不难列出动态规划求解代码，如下所示：

代码清单 1-9

```
int Cal(int V, int type)
{
    opt[0][T] = 0;// 边界条件
    for(int i = 1; i <= V; i++)// 边界条件
    {
        opt[i][T] = -INF;
    }
    for(int j = T - 1; j >= 0; j--)
    {
        for(int i = 0; i <= V; i++)
        {
            opt[i][j] = -INF;
            for(int k = 0; k <= C[j]; k++)      // 遍历第 j 种饮料选取数量 k
            {
                if(i <= k * V[j])
                {
                    break;
                }
                int x = opt[i - k * V[j]][j + 1];
                if(x != -INF)
                {
                    x += H[j] * k;
                    if(x > opt[i][j])
                    {
```

```
                    opt[i][j] = x;
                }
            }
        }
    }
    return opt[V][0];
}
```

在上面的算法中，空间复杂度为 0（V*N），时间复杂度约为 0（V*N*Max（C_i））。

因为我们只需要得到最大的满意度，则计算 opt[i][j]的时候不需要 opt[i][j+2]，只需要 opt[i][j]和 opt[i][j+1]，所以空间复杂度可以降为 0（v）。

【解法二】

应用上面的 动态规划法可以得到结果，那么是否有可能进一步地提高效率呢？我们知道动态规划算法的一个变形是备忘录法，备忘录法也是用一个表格来保存已解决的子问题的 答案，并通过记忆化搜索来避免计算一些不可能到达的状态。具体的实现方法是为每个子问题建立一个记录项。初始化时，该纪录项存入一个特殊的值，表示该子问 题尚未求解。在求解的过程中，对每个待求解的子问题，首先查看其相应的纪录项。若记录项中存储的是初始化时存入的特殊值，则表示该子问题是第一次遇到，此 时计算出该子问题的解，并保存在其相应的记录项中。若记录项中存储的已不是初始化时存入的初始值，则表示该子问题已经被计算过，其相应的记录项中存储的是 该子问题的解答。此时只需要从记录项中取出该子问题的解答即可。

因此，我们可以应用备忘录法来进一步提高算法的效率。

代码清单 1-10

```
int[V + 1][T + 1] opt;      // 子问题的记录项表，假设从 i 到 T 种饮料中，
                            // 找出容量总和为 V'的一个方案，快乐指数最多能够达到
                            // opt（V'，i，T-1），存储于 opt[V'][i]，
                            // 初始化时 opt 中存储值为-1，表示该子问题尚未求解。
int Cal(int V, int type)
{
```

```
if(type == T)
{
    if(V == 0)
        return 0;
    else
        return -INF;
}
if(V < 0)
    return -INF;
else if(V == 0)
    return 0;
else if(opt[V][type] != -1)
    return opt[V][type];    // 该子问题已求解，则直接返回子问题的解；
                            // 子问题尚未求解，则求解该子问题
int ret = -INF;
for(int i = 0; i <= C[type]; i++)
{
    int temp = Cal(V – i * C[type], type + 1);
    if(temp != -INF)
    {
        temp += H[type] * i;
        if(temp > ret)
        ret = temp;
    }
```

```
    }
    return opt[V][type] = ret;
}
```

【解法三】

请注意这个题目的限制条件，看看它能否给我们一些特殊的提示。

我们把信息重新整理一下，按饮料的容量（单位为 L）排序：

Volume	TotalCount	Happiness
20L	TC_00	H_00
20L	TC_01	H_01
…	…	…
21L	TC_10	H_10
…	…	…
2000L	TC_M0	H_M0
…	…	…

假设最大容量为 2 000L。一开始，如果 V%（21）非零，那么，我们肯定需要购买 20L 容量的饮料，至少一瓶。在这里可以使用贪心规则，购买快乐指数最高的一瓶。除去这个，我们只要再购买总量（V-20） L 的饮料就可以了。这时，如果我们要购买 21L 容量的饮料怎么办呢？除了 21L 容量里面快乐指数最高的，我们还应该考虑，两个容量为 20L 的饮料组合的情 况。其实我们可以把剩下的容量为 20L 的饮料之快乐指数从大到小排列，并用最大的两个快乐指数组合出一个新的“容量为 2L ”的饮料。不断地这样使用贪心原 则，即得解。这是不是就简单了很多呢？

编程之美 - NIM“拈”游戏分析

问题

有 N 块石头和两个玩家 A 和 B，玩家 A 先将石头分成若干堆，然后按照 BABA……的顺序不断轮流取石头，能将剩下的石头一次取光的玩家获胜。每次取石头时，每个玩家只能从若干堆石头中任选一堆，取这一堆石头中任意数目（大于 1）个石头。

请问：玩家 A 有必胜策略吗？要怎么分配和取石头才能保证自己有把握取胜？

解法与分析

据说，该游戏起源于中国，英文名字叫做“NIM”，是由广东话“拈”（取物之意）音译而来，经由当年到美洲打工的华人流传出去，这个游戏一个常见的变种是将十二枚硬币分三列排成 [3，4，5] 再开始玩 。我们这里讨论的是一般意义上的“拈”游戏。

言归正传，在面试者咄咄逼人的目光下，你要如何着手解决这个问题？

在面试中，面试者考察的重点不是“what”——能否记住某道题目的解法，某件历史事件发生的确切年代，C++语言中关于类的继承的某个规则的分支等。面试者很想知道的是“how”——应聘者是如何思考和学习的。

所以，应聘者得展现自己的思路。解答这类问题应从最基本的特例开始分析。我们用 N 表示石头的堆数，M 表示总的石头数目。

当 N=1 时，即只有一堆石头——显然无论你放多少石头，你的对手都能一次全拿光，你不能这样摆。

当 N=2 时，即有两堆石头，最简单的情况是每堆石头中各有一个石子（1，1）——先让对手拿，无论怎样你都可以获胜。我们把这种在双方理性走法下，你一定能够赢的局面叫作安全局面。

当 N = 2，M > 2 时，既然（1, 1）是安全局面，那么（1, X）都不是安全局面，因为对手只要经过一次转换，就能把（1, X）变成（1, 1），然后该你走，你就输了。既然（1, X）不安全，那么（2, 2）如何？经过分析，（2，2）是安全的，因为它不能一步变成（1，1）这样的安全局面。这样我们似乎可以推理（3, 3）、（4, 4），一直到（X, X）都是安全局面。

于是我们初步总结，如果石头的数目是偶数，就把它们分为两堆，每堆有同样多的数目。这样无论对手如何取，你只要保证你取之后是安全局面（X, X），你就能赢。

好，如果石头数目是奇数个呢？

当 M=3 的时候，有两种情况，（2, 1）、（1, 1, 1），这两种情况都会是先拿者赢。

当 M=5 的时候，和 M=3 类似。无论你怎么摆，都会是先拿者赢。

若 M=7 呢？情况多起来了，头有些晕了，好像也是先拿者赢。

我们在这里得到一个很重要的阶段性结论：

当摆放方法为（1, 1,…, 1）的时候，如果 1 的个数是奇数个，则先拿者赢；如果 1 的个数是偶数个，则先拿者必输。

当摆放方法为（1, 1,…, 1, X）（多个 1，加上一个大于 1 的 X）的时候，先拿者必赢。因为：

如果 1 有奇数个，先拿者可以从（X）这一堆中一次拿走 X-1 个，剩下偶数个 1——接下来动手的人必输。

如果有偶数个 1，加上一个 X，先拿者可以一次把 X 都拿光，剩下偶数个 1——接下来动手的人也必输。

当然，游戏是两 个人玩的，还有其他的各种摆法，例如当 M = 9 的时候，我们可以摆为（2, 3, 4）、（1, 4, 4）、（1, 2, 6），等等，这么多堆石头，它们既互相独立，又互相牵制，那如何分析得出致胜策略呢？关键是找到在这一系列变化过程中有没有一个特性始终决定着输赢。这个 时候，就得考验一下真功夫了，我们要想想大学一年级数理逻辑课上学的异或（XOR）运算。异或运算规则如下：

XOR（0，0）= 0

XOR（1，0）= 1

XOR（1，1）= 0

首先我们看整个游戏过程，我们从 N 堆石头（M_1, M_2, …, M_n）开始，双方斗智斗勇，石头一直递减到全部为零（0, 0,…, 0）。

当 M 为偶数的时候，我们的取胜策略是把 M 分成相同的两份，这样就能取胜。

开始：（M_1, M_1） 它们异或的结果是 XOR（M_1, M_1）= 0

中途：（M_1, M_2） 对手无论怎样从这堆石头中取，XOR（M_1, M_2）!= 0

我方：（M_2, M_2） 我方还是把两堆变相等。XOR（M_2, M_2）= 0

…

最后：（M_2, M_2） 我方取胜

类似的，若 M 为奇数，我们把石头分成（1, 1, …,1）奇数堆的时候，XOR（1, 1,…,1）[奇数个] !=0。而这时候，对方可以取走一整堆，XOR（1, 1,…, 1）[偶数个]=0，如此下去，我方必输。

我们推广到 M 为奇数，但是每堆石头的数目不限于 1 的情况，看看 XOR 值的规律：

开始：（M_1, M_2, …, M_n） XOR（M_1, M_2, … M_n）=?

中途：（M_1’, M_2’, … M_n’） XOR（M_1’, M_2’, … M_n’）=?

最后：（0, 0, …, 0） XOR（0，0，… 0）=0

不幸的是，可以看出，当有奇数个石头时，无论你如何分堆，XOR（M_1, M_2, … M_n）总是不等于 0！因为必然会有奇数堆有奇数个石头（二进制表示最低位为 1），异或的结果最低位肯定为 1。 [结论 1]

再不幸的是，还可以证明，当 XOR（M_1, M_2, … M_n）!= 0 时，我们总是只需要改变一个 M_i 的值，就可以让 XOR（M_1, M_2, …M_i’, … M_n）= 0。 [结论 2]

更不幸的是，又可以证明，当 XOR（M_1, M_2, … M_n）= 0 时，对任何一个 M 值的改变（取走石头），都会让 XOR（M_1, M_2, …M_i’, … M_n）！= 0。 [结论 3]

有了这三个“不幸”的结论，我们不得不承认，当 M 为奇数时，无论怎样分堆，总是先

动手的人赢。

还不信？那我们试试看：当 M=9，随机分堆为（1，2，6）

开始：（1，2，6）

```
                1=0 0 1
2=0 1 0
6=1 1 0
XOR=1 0 1       即 XOR（1，2，6）!=0
```

B 先手：（1, 2, 3），即从第三堆取走三个，得到（1，2，3）

```
                1=0 0 1
2=0 1 0
3=0 1 1
XOR=0 0 0       所以，XOR（1，2，3）=0
```

A 方：（1, 2, 2）XOR（1, 2, 2）!=0。

B 方：（0, 2, 2）XOR （0, 2, 2）=0

……A 方继续顽抗……

B 方最后：（0, 0, 0），XOR（0, 0, 0）= 0

好了，通过以上的分析，我们不但知道了这类问题的答案，还知道了游戏的规律，以及如何才能赢。XOR，这个我们很早就学过的运算，在这里帮了大忙。我们应该对 XOR 说 Orz 才对！

有兴趣的读者可以写一个程序，返回当输入为（M_1, M_2, …, M_n）的时候，到底如何取石头，才能有赢的可能。比如，当输入为（3, 4, 5）的时候，程序返回（1, 4, 5）——这样就转败为胜了！

扩展问题

8. 1．如果规定相反，取光所有石头的人输，又该如何控制局面？

9. 2．如果每次可以挑选任意 K 堆，并从中任意取石头，又该如何找到必胜策略呢？

编程之美 - 读书笔记 - 中国象棋将帅问题

千呼万唤始出来，在跳票了快一个月之后，虽然明知道书里还有不少错误没改过来（附了一整页的勘误），但是感觉已经不能等下一版了。赶快去书店买回来，吃完 饭躺床上舒舒服服地看。大致翻看之后，总体感觉是书中的内容没有“脱离群众”，很多都是我们平时生活、工作中经常能遇到的。题目不见得难，基本上

给一本 《算法导论》和足够的时间，大多数人都能解决其中的问题。但注意副标题--“微软技术面试心得”，这就给这本书定下一个基调：面对这些我们并不陌生、也并 非特别困难的问题，在有限的时间里，（可能）比较紧张的心情之下，如何充分发挥自己分析问题和解决问题的能力，如何正确且漂亮地解决问题才是关键。我想， 在平时学习的时候或许我们左手《算法导论》，右手《编程之美》效果会更好一些。

中国象棋将帅问题由于比较简单，所以我们暂时不用请出《算法导论》。该问题的具体描述是：（根据中国象棋的基本原则）在只有双的将帅棋盘上，找出所有双方 可以落子的位置（将帅不能碰面），但只能使用一个变量。直觉上我们想到，只要遍历将帅所有可能的位置，去除将帅冲突的位置即可。可见，剩下的问题就在于如 何使用一个变量来做二重循环的遍历。书中解法一给出的方法是将一个 Byte 变量拆成两个用，前一半代表“帅”可以走的位置，后一个变量代表“将”可以走的 位置（事先已经将“将”和“帅”可以走的 3*3 的位置进行了编号），利用位操作即可获得两个计数器的功能。书中的解法三采用结构体来解决一个变量遍历二重 循环的问题，思想上换汤不换药。真正有趣的是解法二，它的代码如下：

```
      int var = 81;
 while( var-- )
 {
  if( var / 9 % 3 == var % 9 % 3 )//发生冲突
   continue;
  else
printf(/** 打印可行的位置 **/);
 }
```

当看到这个解法的时候，我心里有一些感慨。在前几个月，我一直未 MSRA 面试没通过而恼火。但看到这个解法之后，我觉得我确实还要再努力一些才行。短短几 行，体现了简约之美，仅看看这个就值回钱了（开玩笑）。虽然可能有牛人说这没什么了不起，但我觉得如果我在面试这个问题的时候能写下这样的代码，我会很有 成就感。在大多数时候我们无需知道希尔排序的时间复杂度的一点几次方是怎么算出来的，也无需去证明一个最优化问题是否满足“拟阵”的条件，我们只需要在这 样一个“简单”的问题上做得漂亮，就够了。

回过头来分析这个解法。“将”和“帅”各在自己的 3*3 的格子间里面走动，我们共需要验证 9*9=81 种位置关系，这也是 i=81 的由来。此外我们要明白 i/9 和 i%9 的含义。我们知道，整数 i 可以由部两分组成，即 var=(var/9)*9+var%9 ，其中 var<n。我们注意到，在 i 从 81 到 0 变化的过程中，var%9 的变化相当于内层循环，var/9 的变话相对于内层循环。这样，作者就妙地 用一个变量 i 同时获得了两个变量的数值。

简单即是美，相对于解法一的大段代码，我更希望我以后再面试中写出解法二。

其实这个问题还可以进行一些扩展，即如何利用一个变量达到三重循环的效果。也就是说，如果给定下面的循环：

```
    int counter = 0;
 for( int i = 0; i < 5; i++ )
for( int j = 0; j < 4; j++ )
  for( int k = 0; k < 3; k++ )
  {
   System.out.println("counter="+counter+"\t, i="+i+", j="+j+", k="+k);
   counter++;
  }
```

其结果如下：

```
counter=0 , i=0, j=0, k=0
counter=1 , i=0, j=0, k=1
counter=2 , i=0, j=0, k=2
counter=3 , i=0, j=1, k=0
counter=4 , i=0, j=1, k=1
  ....中间略
  counter=59  , i=4, j=3, k=2
```

问题是（1）我们如何用一个打印出相同的结果？（2）如果是 N 重循环呢？

面对第一个问题，实际上就是对原始的中国象棋将帅问题进行了一个扩展，即在棋盘上添加一个“王”，其行走规则和将帅 一样。于是棋盘变成了三国争霸:-) ，将帅王可以走动的格子数分别为 3、4、5，它们之间的互斥条件可以按需要设定。

这时，就需要只用一个变量遍历一个三重循环。直观的方法是像方法一那样把一个 4 字节的 INT 拆开来用。我这里只关注方法二。

只用一个变量解决扩展的中国象棋将帅问题，我们的代码应该是如下的样子：

```
int var = 3*4*5;
while( var-- )
{
 if( /** 冲突条件 **/ )//发生冲突
  continue;
 else
    printf(/** 打印可行的位置 **/);
}
```

在冲突条件中，我们需要知道 var 取得某个特定的值（即第 var+1 次循环）的时候的 i，j，k 分别是多少（这样我们才能判定将帅位置是否冲突）

从上例的结果中我们可以看到，counter 的值（即当前的循环次数）和三元组（i，j，k）是一一对应的，越是外层的循环变化越慢，他们满足什么关系呢？

k 的取值最好确定，我们都知道是 var%3。

在原始的将帅问题中我们知道,j 的值应该是 var/3，但是由于 j 上面还有一层循环，就需要做些调整，变成 var/3%4

最外层循环 i 的值则为(var/(3*4))%5.

即：k=var%3 //其下没有循环了
j=var/3 //其下有几个循环长度为 3 的循环
i=var/(3*4). //其下有几个循环长度为 3*4 的循环

于是 4 重循环的公式我们也可以轻松得出：

```
for( int i = 0; i < 5; i++ )
  for( int j = 0; j < 4; j++ )
    for( int k = 0; k < 3; k++ )
      for( int p = 0; p < 3; p++ )
```

p=var%2 //其下没有循环了
k=var/2 //其下有几个循环长度为 2 的循环
j=var/(2*3)) //其下有几个循环长度为 2*3 的循环
i=var/(2*3*4)//其下有几个循环长度 2*3*4 的循环

下面就是一个变量实现三重循环

```
int var = 2*3*4*5;
while( var-- > 0){
 System.out.println("var="+var+" , i="+((var/(2*3*4))%5)+
                                  ", j ="+((var/(2*3))%4)+",
                                    k="+((var/2)%3)+",
                                    p="+var%2);
}
```

结果是：

```
var=119 , i=4, j=3, k=2, p=1
var=118 , i=4, j=3, k=2, p=0
var=117 , i=4, j=3, k=1, p=1
...中间略
var=5 , i=0, j=0, k=2, p=1
```

```
var=4，i=0，j=0，k=2，p=0
var=3，i=0，j=0，k=1，p=1
var=2，i=0，j=0，k=1，p=0
var=1，i=0，j=0，k=0，p=1
var=0，i=0，j=0，k=0，p=0
```

N 重循环原理也是一样，就不再赘述了。

PS:看到最后一例的结果是不是与《算法导论》中平摊分析一章的二进制计数器很像？只不过这里进制不一样而已:-)

[勘误: P19 代码清单 1-7 的第七行,应该改为 if(i.a%3 != i.b%3)]

谨以此文与大家共勉 2008/04/05

作者：薛笛 EMail:jxuedi#gmail.com

编程之美 - 读书笔记 - 一摞烙饼的排序问题

早在一年前，当时我的一个很牛的胖师兄受邀参加 Google 中国的面试，一开始问他考什么问题他就用签了保密协议打发我们。但当最后他得知无缘 Google 的时候，终于打开话匣子，跟我们这些小字辈滔滔不绝地传授了一些“面经”。我记得其中就有一道题就是这个一摞烙饼问题，还有一道概率题在我面 试 MS RA 的时候也被问到，可恨我当时没在意，后来面试吃了亏。不过如此的巧合说明微软和 Google 面试题库相同？抑或是两个互为竞争对手的公司选择的 标准惊人的一致？只可惜 Google 今年没来哈尔滨招聘，我没法证实答案到底是 A 还是 B。但如果是后者，我相信明年参加校园招聘的朋友还是有必要把这些经 典的问题搞清楚。只可惜我当时对微软的面试不甚了解导致准备不足，而《编程之美》又出得晚了半年，否则说不定....。算了，还是关注问题本身吧。

言归正传，一摞烙饼问题其实是一个很有意思的问题，它的描述是让一摞随机顺序的烙饼通过单手翻转的方式进行排序，以达到这摞烙饼由小到大顺序放置在盘子上 的目的，其特点是每次翻转都会导致第一个烙饼到所要反转的那个烙饼之间的顺序变为逆序。我们的目的是求出次数最少的翻转方案以及翻转次数，很显然这是一个 最优化问题，我们本能想到的是动态规划、贪心以及分支限界三种方法。

书中给出的递归算法或称之为分支限界法（即遍历+剪枝=分支限界）秉承了递归算法传统的简单、明了，但效率偏低的特点。这个问题的实质，我们在每一次反转 之前其实是需要做出一种选择,这种选择必须能够导致全局最优解。递归算法就是递归的构建所有解(实际是一颗搜索树），并在遍历过程中不断刷新 LowerBound 和 UpperBound，以及当前

的最优解（剪枝），并最终找到一个最整体优解。在这种策略下，提高算法的效率只能寄希望于剪枝方法 的改进。但是这种方法显然不是多项式时间的，有没有多项式时间的算法呢？

书中 P22 页提到动态规划，但最后却给出了解决最优化问题普遍适用但效率可能是最差的递归方法。这不禁让人疑惑：这也不美啊！？如果我们能证明该问题满足 动态规划或贪心算法的使用条件，解决问题的时间复杂度将会降到多项式时间甚至 N^2。但书中提到动态规划却最终没有使用，又没有讲明原因，我觉得是一种疏 失（应该不算错误）。那我们就来想一下为什么没有动态规划或贪心算法的原因。

我们知道动态规划方法是一种自底向上的获取问题最优解的方法，它采用子问题的最优解来构造全局最优解。利用动态规划求解的问题需要满足两个条件：即（1） 最优子结构 （2）子结构具有重叠性。条件（1）使我们可以利用子问题的最优解来构造全局最优解，而条件（2）是我们在计算过程中可以利用子结构的重叠性来减少运算次 数。此外，《算法导论》上还以有向图的无权最短路径和无权最长路径为例提出条件（3）子问题必须独立。

首先我们假定烙饼问题存在优化子结构。假如我们有 N 个烙饼，把他们以其半径由小到大进行编号。优化子结构告诉我们对于 i 个烙饼，我们只需要先排列前(i- 1)个，然后再将第 i 个归位；或先排列第 2 到 i 个，最后将第一个归位；又或是找到一个位置 k[i<=k<j] 像矩阵乘法加括号那样，使得我们 先排列前 k 个，再排列后 j-k 个，最后再将二者合并，以找到一个最佳翻转策略等等...

根据动态规划算法的计算过程，我们需要一个 N*N 矩阵 M，其中 M[i][j]表示将编号 i 至编号 j 的烙饼排序所需要的翻转次数。但我们真的能从 M[0] [0..j-1]和 M[1][j+1]，或与 M[i][j]同行同列的值来计算 M[i][j]吗？如果能，我们就能获得多项式时间的算法。

我们来看书中给出的例子：(顶端)3，2，1，6，5，4，9，8，7，0(底端)，我们最终的目标是计算 M[0][9]。
这里我们以计算 M[0][4]为例，计算的矩阵我已经在下面给出：

```
  0 1  2  3  4  5  6  7  8  9
  ----------------------
0|0 1 (1){1}[? ]
1|  0  1 (1){1}
2|    0  1 (1)
3|       0  0
4|          0
  -----------------
```

实际上如果我们要向将 0-4 号烙饼(注意：烙饼编号也等同于其半径)排为正序(中间有其他烙饼也没关系)，按照程序给出的结果， 我们需要进行 3 次翻转，分别为[2,5,9](即分别翻转队列中第二(从零开始)、五、九个烙饼，这里的数字不是烙饼的编号)：

```
  [1]  [2]  [3]  6   5  [4]  9  8  7  [0]
```

[4] 5 6 [3] [2] [1] 9 8 7 [0]
[0] 7 8 9 [1] [2] [3] 6 5 [4]

我们知道，该矩阵中每一个数的背后都隐含着一个烙饼的排列，例如 M[0][4]就应该对应 0,7,8,9,1,2,3,6,5,4

所以，每一个 M[i][j]的选取都蕴含着其子排列的顺序的变化。

在计算 M[i][j]的时候，我们需要计算 i-j 号饼的全部划分(不包括全部为 1 的划分)所能构成的翻转结构，并取其翻转 次数最少的哪一个最为 M[i][j]的最终值。例如，我们在计算 M[0][4]的时候，需要查看：

/**先将 0 和 1-4 号分别排序，最后将二者合并为有序所需要的翻转次数*/
M[0][0],M[1][4]

/** 同上 */
M[0][1],M[2][4]

/** 同上 */
M[0][2],M[3][4]

/** 同上 */
M[0][3],M[4][4]

/* 先将 0、1、2、3-4 号分别排序，最后将 4 者合并为有序所需要的翻转次数.
* 注意这里又包含将 4 个分组再次进行划分的问题!
*/
M[0][0],M[1][1],M[2][2],M[3][4]
.....//中间略
M[0][3],M[4][4]

如果再加上运算过程中我们可以淘汰超过最大反转次数的方案（剪枝？），我们完成全部的运算，所经历的运算过程的时间复杂度已经不是多项式时间的，而是和先前所说的递归方法已没什么两样。

造成这种现象的原因是：某个子问题的最优解不一定是整体的最优解，所以我们在处理整个问题的时候，需要遍历所有可能的子问题，并计算它到整体问题所消耗的代价，才能最终作出有利于整体问题的选择。

所以我们一开始的假设，即烙饼问题有优化子结构的假设是错误的。因此我们不能用动态规划，同理也不能用贪心算法。

但说到每一步的“选择”问题，我记得算法导论上有一个叫做“A*”的算法，它的思想是在进行每一步选择的时候都“推算”最终可能需要的代价，并选择当前代 价最小的分支进行遍

历。这个“推算”的结果可能不会是最终的代价，而只是作为分支选择的依据。如果谁有兴趣就做一下吧 :-)

好累，就到这里吧，休息，休息一会！

《编程之美》之我的勘误：

[1]P23 和 P25 中的变量 m_arrSwap 应该是 m_SwapArray,因为 m_arrSwap 从被定义过(难道程序的编译错误也没改过来？不能理解:-( )

[2]P24 中 upBound()方法中应改为 return (nCakeCnt-1)*2，当然这个上界不改也行，只是我想这里应该和前面说的相对应

[3]Search()方法的剪枝部分，if(step+nElimate > m_nMaxSwap)有误，因为 nElimate 可能为 0，所以当 step 等于 m_nMaxSwap 时候 会造成下面的 m_reverseCakeArraySwap[step] = i;的数组越界。所以应该改为：
if(step + nEstimate > m_nMaxSwap || step >= m_nMaxSwap)。这个错误运行时也应该查出来啊，再次不理解。

[4]修改完[2]和[3]之后，P28 页的例子的运行次数应该是 141787 次,而如果[2]不修改的话运行次数应该为 164872

以下是修改后的 Java 代码，其中所有些数组长度的变量在 Java 里没用，不过为了保持原文的一致性，这里就没把它们删掉：

```
public class CakeTuneProblem {

    int[] m_cakeArray; // 烙饼信息数组
    int m_nCakeCnt;    // 烙饼个数
    int m_nMaxSwap;    // 最多交换次数,最多为 2*(m_nCakeCnt-1)
    int[] m_swapArray; // 交换结果数组
    int[] m_reverseCakeArray; // 当前翻转烙饼信息数组
    int[] m_reverseCakeArraySwap; // 当前翻转烙饼交换信息数组
    int m_nSearch; //当前搜索次数信息

    public CakeTuneProblem(){
        m_nCakeCnt = 0;
        m_nMaxSwap = 0;
    }
```

```
    void run(int[] pCakeArray){
        init(pCakeArray);
        m_nSearch = 0;
        search(0);
    }

    void output(){
        for(int i = 0; i < m_nMaxSwap; i++)
            System.out.print(m_swapArray[i]+",");

        System.out.println(" Search Times:"+m_nSearch);
        System.out.println("Total Swap Times:"+m_nMaxSwap);
    }

    void init(int[] pCakeArray){
        assert( pCakeArray != null);
        assert( pCakeArray.length > 0);

        m_nCakeCnt = pCakeArray.length;

        m_cakeArray = new int[m_nCakeCnt];

        for(int i = 0; i < m_nCakeCnt; i++)
            m_cakeArray[i] = pCakeArray[i];

        m_nMaxSwap = upperBound(m_nCakeCnt);

        m_swapArray = new int[m_nMaxSwap];
```

```
    m_reverseCakeArray = new int[m_nCakeCnt];
    for(int i = 0; i < m_nCakeCnt; i++)
        m_reverseCakeArray[i] = m_cakeArray[i];

    m_reverseCakeArraySwap = new int[m_nMaxSwap];
  }


int lowerBound(int[] pCakeArray)
  {
    int t, ret = 0;

      for(int i = 1; i < pCakeArray.length; i++){
       t = pCakeArray[i] - pCakeArray[i-1];
        if( (t==1) || (t == -1)){
        }
        else{
         ret ++;
        }
    }
    return ret;
  }


 int upperBound(int nCakeCnt){
    //return (nCakeCnt-1)*2;
    return (nCakeCnt)*2;
  }
```

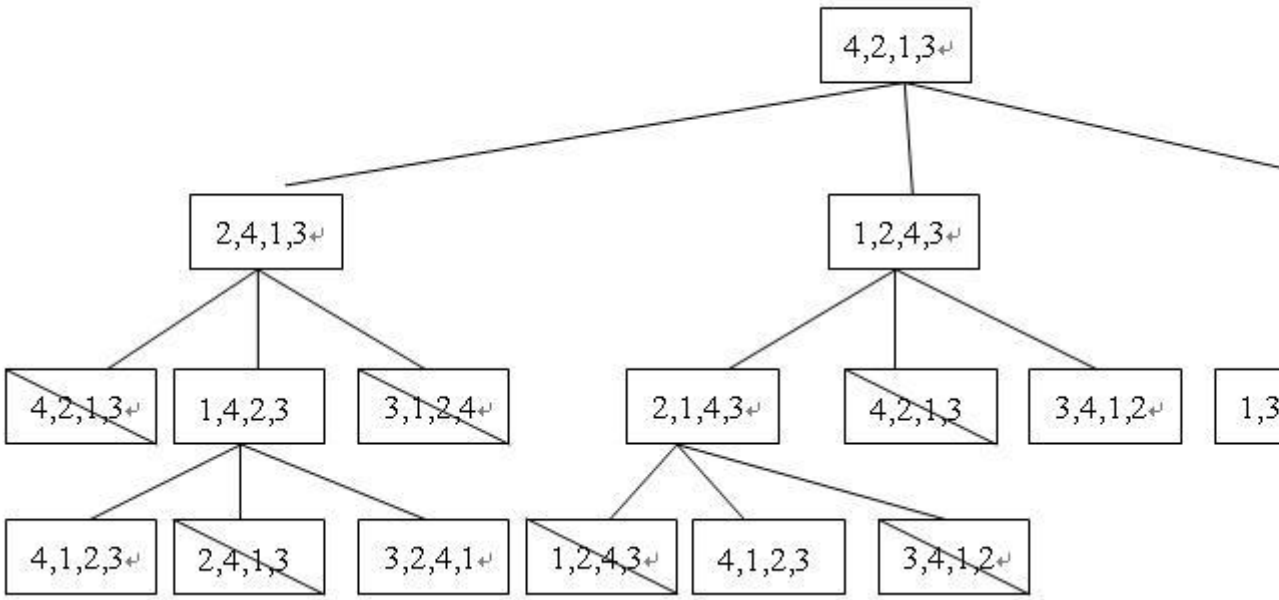
```
    void search(int step){
        int i, nEstimate;
        m_nSearch++;

        //剪枝条件
        nEstimate = lowerBound(m_reverseCakeArray);
        if( step + nEstimate > m_nMaxSwap || step >= m_nMaxSwap)
            return;

        if( isSorted( m_reverseCakeArray)){
            if( step < m_nMaxSwap){
                m_nMaxSwap = step;
                for(i = 0; i < m_nMaxSwap; i++)
                    m_swapArray[i] = m_reverseCakeArraySwap[i];
            }
            return;
        }

        for(i = 1; i < m_nCakeCnt; i++){
            revert(0,i);
            m_reverseCakeArraySwap[step] = i;
            search(step+1);
            revert(0,i);
        }

    }

    boolean isSorted(int[] pCakeArray)
    {
        for(int i = 1; i < pCakeArray.length; i++)
```

```
        if( pCakeArray[i-1] > pCakeArray[i])
            return false;
    return true;
  }

  void revert(int nBegin, int nEnd)
   {
    assert( nEnd > nBegin);
    int t = 0;
    for( int i=nBegin, j=nEnd; i<j; i++,j--)
      {
       t = m_reverseCakeArray[i];
       m_reverseCakeArray[i] = m_reverseCakeArray[j];
       m_reverseCakeArray[j] = t;
    }
  }


   public static void main(String[] args) {
    CakeTuneProblem problem = new CakeTuneProblem();
    //int[] cakeArray = {3,2,1,6,5,4,9,8,7,0};
      int[] cakeArray = {3,2,1,6,5,4};
    problem.run(cakeArray);
    problem.output();
  }

}
```

编程之美 - 读书笔记 - 烙饼问题与搜索树收藏

前 面已经写了一些关于烙饼问题的简单分析，但因为那天太累有些意犹未尽，今天再充实一些内容那这个问题研究透。我想，通过这篇文章，我们就可以把这一类问题 搞懂。再遇到优化问题，如果我们想不到别的办法，就可以采用搜索树算法来解决，至少我们不至于拿不出解决方案。前面我们已经知道，关于一摞烙饼的排序问题 我们可以采用递归的方式来完成。其间我们要做的是尽量调整 UpperBound 和 LowerBound，已减少运算次数。对于这种方法，在算法课中我们应该称之为：Tree Searching Strategy。即整个解空间为一棵搜索树，我们按照一定的策略遍历解空间，并寻找最优解。一旦找到比当前最优解更好的解，就用它替换当前最优解，并用它来进行“剪枝”操作来加速求解过程。

书中给出的解法就是采用深度优先的方式来遍历这棵搜索树，例如要排序[4,2,1,3]，最大反转次数不应该超过(4-1)*2=6 次，所以搜索树的深度也不应大于 6，搜索树如下图所示：



这里只列到第三层，其中被画斜线的方块由于和上层的某一节点的状态重复而无需再扩展下去（即便扩展也不可能比有相同状态的上层节点的代价少）。我们可以看到在右子树中的一个分支，只需要用 3 次反转即可完成，我们的目标是如何更为快速有效的找到这一分支。直观上我们可以看到：基本的搜索方法要先

从左子树开始，所以要找到本例最佳的方案的代价是很高的（利用书中的算法需要查找 292 次）。

既然要遍历搜索树，就有广度优先和深度优先之分，可以分别用栈和队列来实现（当然也可以用递归的方法）。那么如何能更有效地解决问题呢？我们主要考虑一下几种方法：

（1） 爬山法

该方法是在深度优先的搜索过程中使用贪心方法确定搜索方向，它实际上是一种深度优先搜索策略。爬山法采用启发式侧读来排序节点的扩展顺序，其关键点就在于测度函数 f(n)的定义。我们来看一下如何为上例定制代价函数 f(n)，以快速找到右子树中最好的那个分支（很像贪心算法，呵呵）。

我们看到在[1,2,4,3]中，[1,2,3]已经相对有序，而[4]位与他们之间，要想另整体有序，需要 4 次反转；而[3,1,2,4]中，由于[4]已经就位，剩下的数变成了长度为 3 的子队列，而子队列中[1,2]有序，令其全体有序只需要 2 次反转。

所以我们的代价函数应该如下定义：

1 从当前状态的最后一个饼开始搜索，如果该饼在其应该在的位置（中间断开不算），则跳过；
2 自后向前的搜索过程中，如果碰到两个数不相邻的情况，就+1
这样我们就可以在本例中迅速找到最优分枝。因为在树的第一层

f(2,4,1,3)=3，f(1,2,4,3)=2，f(3,1,2,4)=1，所以我们选择[3,1,2,4]那一枝，而在[3,1,2,4]的下一层：

f(1,3,2,4)=2，f(2,1,3,4)=1，f(4,2,1,3)=2，所以我们又找到了最佳的路径。

上面方法看似不错，但是数字比较多的时候呢？我们来看书中给出的 10 个数的例子：

[3,2,1,6,5,4,9,8,7,0]，程序给出的最佳翻转序列为{ 4,8,6,8,4,9}（从 0 开始算起）

那么，对于搜索树的第一层，按照上面的算法我计算的结果如下：

```
f(2,3,1,6,5,4,9,8,7,0)=4
   f(1,2,3,6,5,4,9,8,7,0)=3
   f(6,1,2,3,5,4,9,8,7,0)=4
   f(5,6,1,2,3,4,9,8,7,0)=3
   f(4,5,6,1,2,3,9,8,7,0)=3
   f(9,4,5,6,1,2,3,8,7,0)=4
   f(8,9,4,5,6,1,2,3,7,0)=4
f(7,8,9,4,5,6,1,2,3,0)=3
f(0,7,8,9,4,5,6,1,2,3)=3
```

我们看到有 4 个分支的结果和最佳结果相同，也就是说，我们目前的代价函数还不够“一击致命”，但是这已经比书中的结果要好一些，起码我们能更快地找到最佳方案，这使得我们在此后的剪枝过程更加高效。

爬山法的伪代码如下：

```
1 构造由根组成的单元素栈 S
2 IF Top(s)是目标节点 THEN 停止；
3 Pop(s);
4 S 的子节点按照启发式测度，由小到大的顺序压入 S
5 IF 栈空 Then 失败
  Else 返回 2
```

如果有时间我会把爬山法解决的烙饼问题贴在后面。

（2） Best-First 搜索策略

最佳优先搜索策略结合了深度优先和广度优先二者的优点，它采取的策略是根据评价函数，在目前产生的所有节点中选择具有最小代价值的节点进行扩展。该策略具有全局优化的观念，而爬山法则只具有局部优化的能力。具体用小根堆来实现搜索树就可以了，这里不再赘述。

（3） A*算法

如果我们把下棋比喻成解决问题，则爬山法和 Best-First 算法就是两个只能“看”未来一步棋的玩家。而 A*算法则至少能够“看”到未来的两步棋。

我们知道，搜索树的每一个节点的代价 f*(n)=g(n)+h*(n)。其中，g(n)为从根节点到节点 n 的代价，这个值我们是可求的；h*(n)则是从 n 节点到目标节点的代价，这个值我们是无法实际算出的，只能进行估计。我们可以用下一层节点代价的最小者来替代 h*(n)，这也就是“看”了两步棋。可以证明，如果 A*算法找到了一个解，那它一定是优化解。A*算法的描述如下：

1. 使用 BestFirst 搜索树
2. 按照上面所述对下层点 n 进行计算获得 f*(n)的估计值 f(n)，并取其最小者进行扩展。
3. 若找到目标节点，则算法停止，返回优化解

总结：归根到底，烙饼问题之所以难于在多项式时间内解决的关键就在于我们无法为搜索树中的每一条边设定一个合理的权值。在这里，每条边的权值都是 1， 因为从上一个状态节点到下一个状态节点之需要一次翻转。所以我们不能简单地把每个节点的代价定义为翻转次数，而应该根据其距离最终解的接近程度来给出一个 数值，而这恰恰就是该问题的难点。但是无论上面哪一种方法，都需要我们确定搜索树各个边的代价是多少，然后才能进行要么广度优先、要么深度优先、要么 A*算法的估计代价。所以，在给出一个合理的代价之前，我们所有的努力都只能是帮忙“加速”，而无法真正在多项式时间内解决问题。

《编程之美》读书笔记(四)：卖书折扣问题的贪心解法

每 次看完《编程之美》中的问题，想要亲自演算一下或深入思考的时候，都觉得时间过得很快，动辄一两个小时，如果再把代码敲一遍的话，需要的时间可能更长，真 是搞不懂通过微软面试的那些家伙的脑袋到底什么构造，书的序言中提到他们每次面试 45 分钟，还要写出程序？！在我看来，如果是控制 CPU 曲线或是中国象棋 问题或许还有可能，如果是买书折扣问题，我觉得真的是不太容易，尤其是如果当面试者钻进本题的贪心解法而不是动态规划算法的思路之后，因为我写这篇文章前 前后后大概用了 5 个小时

:-(。不过我想只要是学习就不是浪费时间，今天上网看到微软的校园招聘网站又有更新，等我把这本书看完，就投简历过去试一试 :-) 。

1 问题描述及分析

买书折扣问题的描述是，某出版社的《哈里波特》系列共有 5 卷，每本单卖都是 8 块钱，如果读者一次购买不同的 k（k>=2）卷，就可以享受不同的折扣优惠，如下所示：

本数	折扣
2	5%
3	10%
4	20%
5	25%

表1-1 折扣表

问题是如果给定一个订单，如何计算出最大的折扣数？

书中给出的动态规划解法这里就不再赘述了。不过，里面有两个问题需要单独关注一下：(1)如果订单描述为（X_1，X_2，X_3，X_4，X_5），其中 X_1-X_5 为所订数的数量，其所在位置为卷的编号，即第一卷 X_1 本，第二卷 X_2 本，...，第 5 卷 X_5 本；如果订单为：（X_3，X_2，X_4，X_5，X_1），则表示第一卷 X_3 本，第二卷 X_2 本，...，第五卷 X_1 本。我们可以很容易的看到，由于每本书的价格相同，所以折扣的多少仅仅在于如何选取而不在于究竟取那一卷书。因此，上面两个订单的最大折扣数是相同的，这也使得我们可以使用统一的方法（Y1，Y2，Y3，Y4，Y5），其中 Y1≥Y1≥Y1≥Y1≥Y5，来表示一个订单。作者将每本书的定价设为相同的是为了简化问题，因为如果每本书的定价也不同，则问题就会变得更加复杂，这时我们就不能仅仅考虑选几本书，还要考虑选哪几本。(2)书中说对于一次选择 4 至 2 本书的情况，（以 3 本书为例）只需考虑 F（Y_1-1，Y_2-1，Y_3-1，Y_4，Y_5）的情况就可以了（注意，F 为订单总价计算函数），并说“**这样选择能够保证在选择当前折扣的情况下，剩下的书的种类最多，它比其它组合都好**”。 我觉得这个结论并不显然，下一步可选的书的种类多就能证明这个子问题比别的子问题更好？这点我不敢苟同，须知该问题的解决是一个多步选择的过程，所以要得 到这个结论就需要严格证明。如果这个条件不成立，那么书中给出的递归式也就不成立，即不能证明优化子结构性质成立。所以，对于如此重要的细节，书中应该给 出严格证明而不是一句话带过。

刚 开始拿到问题的时候就条件反射地想能不能用贪心算法，即每次都尽量按最大折扣来取书。书中给出一个反例：所给的订单是（2，2，2，1，1），按照贪心算 法，我们的选择方式是 5+3，其折扣为 5*0.25+3*0.10=1.55；而如果采用 4+4 的模式，则折扣数为 2*4*0.20=1.6。这显然违背 了贪心规

则。所以书中在解法二中采用了另外一种分析，作者计算了订单中书的数量在[1-10]区间内，各种不同的选取方法所能获得的最大折扣数：

本数	可能的分解本数	对应的折扣
对于 2-5 本，直接按折扣购买	2	5%
	3	10%
	4	20%
	5	25%
6	=4+2	0.9
	=3+3	0.6
	=2+2+2	0.3
7	=5+2	1.35
	=4+3	1.1
8	=5+3	5*25%+3*10%=1.55
	=4+4	4*20%+4*20%=1.6
	=3+3+2	0.7
	=2+2+2+2	0.4
9	=5+4	2.05
	=5+2+2	1.45
	=4+3+2	1.2
	=3+3+3	0.9
10	=5+5	2.5
	=4+4+2	1.7
	=4+3+3	1.4
	=2+2+2+2+2	0.5

表 1-2 折扣计算表

看 上面给出的折扣计算表，我们可以简单分析一下。实际上这个反例产生的原因是 3 本书的折扣数量与 4 本书的折扣数量差距过大造成的。实际上只要我们将题目稍微 改动一下，将三本书的折扣改成 15%，得到表 1-2 所示的折扣，我们就会发现贪心算法奇迹般地生效了！（为清楚起见，我把修改前和修改后的

折扣计算结果放 到了一张表中）那么如果一开始作者给出的就是三本书 15%的折扣的话，从表中就可以看出使用原始的贪心算法所得到的结果是对的，就可能会连带得出贪心算法 有效的结论？！**我有点后怕！**

很显然，如果可以应用贪心算法的话，贪心选择不应该局限于某一种折扣数量的设定。而如果《哈利波特》不是 5 卷而是 M 卷，折扣表扩大的话，就很可能会产生更多的反例情况。

本数	可能的分解本数	原始的折扣	新的折扣
对于 2-5 本，直接按折扣购买	2	5%	5%
	3	10%	15%
	4	20%	20%
	5	25%	25%
6	=4+2	0.9	0.9
	=3+3	0.6	0.9
	=2+2+2	0.3	0.3
7	=5+2	1.35	1.35
	=4+3	1.1	1.25
8	=5+3	5*25%+3*10%=1.55	5*25%+3*15%=1.7
	=4+4	4*20%+4*20%=1.6	1.6
	=3+3+2	0.7	3*15%*2+2*10%=1.0
	=2+2+2+2	0.4	0.4
9	=5+4	2.05	2.05
	=5+2+2	1.45	1.45
	=4+3+2	1.2	0.8+0.45+0.1=1.35
	=3+3+3	0.9	0.45*3=1.35
10	=5+5	2.5	2.5
	=4+4+2	1.7	1.7
	=4+3+3	1.4	1.7
	=2+2+2+2+2	0.5	0.5

表 1-3 与贪心策略相符合的折扣表

于 是作者想把多余 10 本的订单分成若干个小于 10 的订单组，并把每组的最大折扣相加，以得到全局最优解，关于这一点我将在下面进行说明。作者在最后讲述了修 改贪心策略的方法，不过我不是太理解，而且其中还有一些错误，最重要的是作者最终也没能证明其正确性，我就不再深究了。

2 贪心算法是否适用的分析

贪 心算法的适用有两个必要条件，即优化子结构和贪心选择性。第一个性质由于已经证明可以适用动态规划算法，所以优化子结构性质显然成立（假如书中的动态规划 递归式成立的话）。现需要证明其贪心选择性，即如何“贪心”的进行选择。显然每次都查找最大的折扣数进行处理的贪心方法是行不通的，那么是贪心方法真的不 行还是我们“贪”的不正确呢？我们下面就来分析。

贪 心选择性的含义是，一个全局最优解可以通过局部最优选择来达到，换句话说，当考虑作何选择时，我们只考虑对当前问题最佳的选择而不考虑子问题的结果。贪心 算法所做的当前选择可能要依赖于已经做出的所有选择，但不依赖于有待于做出的选择或子问题的解。所以，贪心策略通常是自顶向下地，一个一个地做出贪心选 择，不断地将给定的问题归约为更小的问题。当然，在此之前我们必须证明在每一步所做的贪心选择最终能产生一个全局最优解，这也正是本题的关键所在。

书中解法二给出的分组的思想是可以借鉴的，但是显然犯了方向性的错误。因为贪心算法的关键在于“**选择**”， 即从当前的状态来“贪心地”从多种子状态中选择一个“当前的”最优解进行下去，并由其贪心选择性而使最终的解刚好是最优解。既然书中最后采用的还是（经修 改后的）贪心算法，就不应该把整体的问题分成若干组来执行。这是因为贪心算法的上一次选择与下一次选择之间是带有连续性的，并不能够将它们拆开处理；而如 果要拆开处理之后再将结果相加，就还需要证明拆分是使得结果最优的拆分。所以我们的目标最终还是应该定为寻找如何“贪”！其实作者的意图是对的，但实际目 标应该定为**限制本次选择对于此后选择的影响，或使本次选择的影响仅限于下一次选择，**这也是表格 1-1 只需要计算到 10 的原因（两次选择最多选择 10 本书），但由于下一次选择的影响可能会影响下下次的选择，所以我们不能硬性将这些选择拆开然后再相加，而只能一次一次地完成选择。

2.1 特殊情况

首先，让我们分析一下图 2-1 左图所 示的订单，很显然，蓝框所框住的 15 本书都应该按照 5 本书的折扣处理。我们之所以“敢”这么做的原因是这次的处理不会影响到下一次的“选择”。即如果蓝框 的范围再向上扩大一层的话，一次选择之后就会使得最后两列书的数量为 0，并导致下次选择时最多只有 3 本书的折扣可以选择，相当于间接地选择了“5+3”模 式。而如果蓝框的范围如图 2-2(左)所示的话，这两次处理我们还可以有“4+4”模式可供选择，而这才是最佳选择。所以，当我们的选择不会使列数减少的 时候，我们可以放心大胆地选择利润最大的那选择法；同样在图 2-1 右图（注意：左右两图并无关联，是两个不同

的订单），由于第五列已经没有了，在蓝框范围内的书籍我们仍然应该按照左图的做法处理，只是这次我们的折扣是 4 本书的。

再细想一下，如果我们不选择折扣最大的（即书的数量最多的）那个选择，按照**书中给出的动态规划子问题的定义中的取法：如果取 4 本书是从前四列取，取 3 本书从前三列取，依次类推，我们权且称其为“最小取法”（与书中的“最小表示”相呼应）**，我们可以知道，这一次不选择折扣最大只是将折扣最大的选择“推后”了。因为贪心算法每一步的选择都应该是当前状况下的最优选择，即我们在每一步都应该尽可能多获得折扣，而不应该将折扣“推后”而导致我们本次的选择并非最佳选择。换句话说，**我们应该保证下一次选择的书的数量不应该比本次选择的数量多**，即我们可以允许两次的选择是 5+3 或 4+4，但绝不能允许 3+5 出现。作者给出的表 1-1 也印证了这一点，我们可以看到，**表中所有的拆分都是降序排列的，即下一次可选择的最大数量不会超过本次选择的书的数量**。

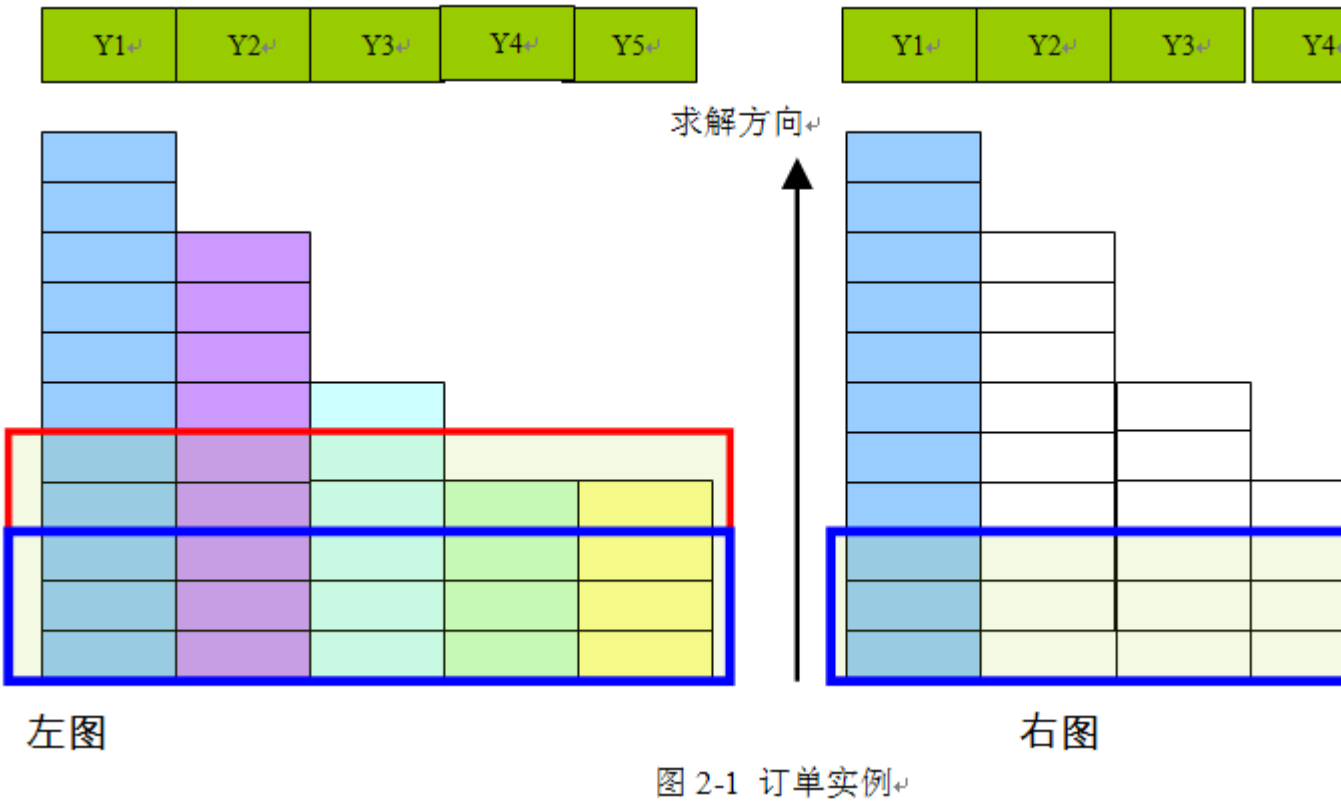


图 2-1 订单实例

例如图 2-2 左图中，如果我们本次选择（最底层）只选择了 3 本书，那么上一段定义的**最小取法**，我们得到的子问题应该是 F（Y_1-1，Y_2-1，Y_3-1，Y_4，Y_5），于是下一次我们仍然可以选择 5 本书，大于本次所选的三本书，这是不应该发生的。而如果我们选择 5 本书，下一次只有 3 本书可以选择，这显然是可以的，但并不是最优的；如果我们选择 4+4 呢？显然不违反我们刚才定下的规则，而且折扣最大。下面我们的问题就是如何对原始的贪心定义稍作修改，使其能够帮我们挑出最大折扣的选法。

在处理掉一些特殊情况之后，我们的订单变为图 2-2 左图所示的情况。而且前面我们定义：**本次选择的书的数量不应该小于下一次可选择的书的最大数量（其中，书的取法按照书中动态规划方法给出的选法来进行）**。 也就是说，我们不应该把最大的折扣选择“推后”执行，而使得本次选择不是当前看起来“最优的”。其次，这种选取方式也使得本次选择的影响仅限于下一次（即 图中红色框中的范围），而不会“扩散”到下一次选择之后的选择，例如下下次选择。同时，下下次选择也只受下次选择的影响，由此我们成功地阻止了本次选择对 整体带来的影响。

2.2 修改的贪心算法

对于图 2-2 左图所示的情况，在满足上述条件的前提下，我们目前有 5 和 4 两种选择（当然，如果书的卷数为 M>5 的话，可能的选择也许会更多），我们如何在不引发贪心反例的情况下获得最大折扣呢？我的答案是“**查表**”。此时，我们应该查阅表 1-2，找到其中折扣数最大的那个最为本次的选择。我们可以看到，因为 5+3<4+4，所以我们本次应该选择 4 本书。我们还需要验证，下一次选择我们还需要选择 4 本书。此时状况如图 2-2 右图所示，我们的选择只有 4+3，因为如果本次我们选择 3 本书，下一次我们还可以选择 4 本书，这与我们的规则不符。所以我们的贪心选择是正确的。其余的书籍我们可以按照上面提到的两种情况加以处理即可。

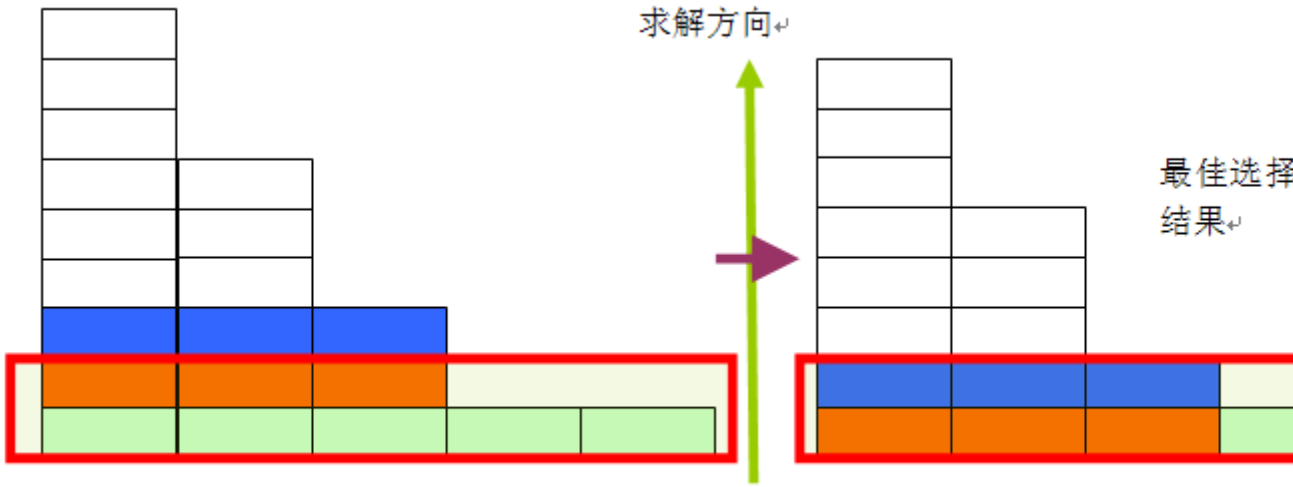


图 2-2 第一步处理之后的订单（20 本书）

经过修改后，贪心算法的空间复杂度变为计算如 1-2 所示的表格的代价，而修改后的贪心算法的时间复杂度就很直观，应该为我们做出选择的次数（包括一次查表操作），也就是 O（Y1）。

3 总结

本问题的修改后的贪心算法的规则是：

1 本次选择的书的数量不应该小于下一次可选择的书的最大数量（其中，书的取法按照书中动态规划方法给出的选法来进行）。

2 每一次选择之前都应该查表，选择其中使得近两次折扣数最大的那个作为本次选择（因为我们使得本次选择的影响被控制在两次选择的范围内）。

而图 2-1 左图中蓝框内的情况最终也可以归结为后面所分析的状况中的一种。即因为下一次只有 5 可以选择（少于五本将违反规则 1），所以本次就选择 5 本书。此外，当书的卷数变为 M(M>5)的时候，表 1-2 需要的计算的数量也会相应变大。但实 际上表 1-2 可以简化，例如我们看 8 本书的情况， 3+3+2 和 2+2+2+2 其实都不用考虑，因为按照图 2-2 左图所示，因为规则 1 的存在使我们不可能选择 3 本书。即便是换一个例子，如果我们本次选择 3 本书、下一次最多也只能选 3 本书的话，我们就回到了处理图 2-2 左图蓝框中的情况当中。最后，如果书的卷数为 M，则我们的表格的行数只需要为 2M 即可，**因为我们一次选择的影响仅限于本次和下次，所涉及的书的数量不会超过 2M 个**。

4《编程之美》本问题勘误

[1] P33，本数为 6 的 4+2 的折扣应该是 4*20%+2*5%=0.9 而不是 1.1

[2] P34，最后一段，第 2-3 行说原始的“**贪心策略建议我们买 Y5 本卷五，Y4-Y5 本卷四，Y3-Y4 本卷三，Y2-Y3 本卷二和 Y1-Y2 本卷一**”，对于订单（2，2，2，1，1），其中前三卷每卷两本，后两卷每卷一本，贪心的规则应该是 5+3，即第一次选择应该每卷拿一本才对。**而文中说第四卷拿 Y4-Y5=0 本？**所以这段文字我认为不妥。而且对于这一段后面的内容我也不是太明白，我个人觉得有修改一下的必要。

[3] 同样是 P34 最后一段，最后三行。虽然我对这一段的意思不太了解，但是仍然可以看出一个明显的错误，那就是最后三行所举的例子提到“要买 3 本第一卷，2 本第二卷....”，而在最后却说“经调整后的贪心策略告诉我们应该买三本四卷，三本两卷....”，前面说第二卷一共就两本，后面却说要买 3 本第二卷，真不知道这个贪心策略是怎么“告诉的”。我建议，反正由于书的价格导致订单上的书到底第几卷不重要，就不如干脆不用第几卷，改用第几列更为合适。

编程之美 - 读书笔记 - 饮料供应问题收藏

新一篇: php.net 所有中文资料疑被全部被删除 | **旧一篇**: 谷歌今年在中国新招 200 人 应届毕业生占 50%

从买书那天算起，到今天已经过了半个多月。这段时间说短不短，如果是一本 300 多页的小说的话，我大概一天就能搞定（我的记录是一天一千多页《大唐双龙传》），但是到现在《编程之美》我只看了不到 50 页。 虽然我不是天天看，

但是一旦我看了一个问题之后，我就希望能够把这个问题在算法层面分析透，这份专注是我以前看《算法导论》或者上算法课的时候所不曾体会 到的。究其原因，主要还是纯粹的理论流于枯燥，纯粹的应用不免肤浅，而这本书的定位刚刚好，既能够以应用带动算法的学习，又能够避免过于说教的风格。

尽 管初衷不错，但我仍觉拿捏得尺度有待商榷。因为编程应该既严谨又灵活，严谨的思考保证了程序运行的稳定，而灵活的思维则为创新提供了条件。而“编程之美 ” 给我的感觉是灵活有余严谨不足。我觉得既然是有关算法的书，那么在一些关键点上的证明就不可或缺，例如在我看的六个问题中就出现两个贪心算法没证明其贪心 选择性，或讨论的不够，其中“买书问题”的贪心选择还是有问题的。当然，我们不应该奢求这本定位于“面试题集”的书能够写出如“算法导论”般严格的证明。 但至少应该给出具体思路，以证明想法是有依据而非直观感觉（因为我们的直观感觉在很多时候是不准确的，就比如我的读书笔记四种分析买书问题，如果把三本书 的折扣改成 15%， 那我们就有可能被得到的折扣表格所误导而采用原始的贪心算法）；最可怕的是鉴于构建反例实际是一件非常困难的事情，所以我们从给出的例子中可能也无法看出 端倪。此外，在看书过程中所遇到的各种错误的数量也是屡创新高。听说马上就要出第二版了，真是为后面的内容担心，难道最后又要附一页勘误表？

虽然问题多多，但由于瑕不掩瑜，我们仍然能够从书中获得足够多的微软人的智慧，哪怕是促使我们重新温习一遍算法基础也好，希望第四版（ 如果有的话:-) ）能看到一个完美的“编程之美”。

1 问题描述及分析

本题所说的问题是微软每天为员工提供各种不同的饮料，如可乐，酸奶，豆浆，咖啡，绿茶……..（待遇不错啊，呵呵），饮料 i 的单位容量为 V_i，其中每种饮料单个容量都是 2 的方幂，员工对饮料 i 的满意度为 H_i，冰柜的总容量为 V（每天必须装满），问题是如何组合现有的各种饮料，使总的满意度最高。

还是说一下我的第一印象，很显然这是一道最优化问题，但很容易想到这道题和线性规划的描述很符合，但是由于解线性规划的单纯型方法比较复杂，这里就不再讨论了。其次，回想一下经典的 0-1 背包问题，和本问题也有些相似，所不同的是 0-1 背包问题中，每件物品只能拿一次，而这里同一种饮料能拿多瓶；此外，原问题中每天供应的总量 V 是必须达到的（否则会有员工投诉？），所以不能够像 0-1 背包问题那样有让背包装不满的情况。这个条件实际上改变了我们对于最优解的搜索策略，因为容量为 V 的装饮料的冰柜每天早晨都必须是满的，所以即便有另一个使满意度最高但冰柜不满的组合我们也是不能选的。

其实我们可以稍微改变一下本题的条件，忽略原问题中的每种饮料单个容量都是 2 的方幂的条件，并允许冰柜不满的情况下求最大满意度的组合，希望可以使原问题的解决更富有一般性。

2 算法分析

2.1 动态规划算法

没有悬念，优化问题就用动态规划、贪心算法、分支限界轮番上阵就好了。设 Opt（V’，i）表示从 i 到 n-1 种饮料中，Ci 为第 i 种饮料可能的最大数量，算出总量为 V’的方案中满意度之和的最大值。那么递归式就应该是：

Opt（V’，i）=max{ k * H_i+Opt（V’-V_i * k，i+1）}（k=0，1，2…，Ci，i=0，1，2…，n-1）

这里我觉得需要说明给出的饮料组合最终可以组合出 V。

2.2 贪心算法

书中的贪心解法似曾相识，把信息按照饮料的容量排序（其中设我们有 n^0 种容量为 2^0 的饮料）：

Volume	TotalCount	Happiness
2^0	TC_0_0	H_0_0
…	…	…
2^0	TC_0_1	H_0_n_0
…	…	…
2^1	TC_1_0	H_0_0
…		…
2^M	TC_M_0	H_M_0
…		

然后按照下面的顺序进行贪心选择：

（1） 饮料总量为 1，从容量为 2^0 的饮料中选出快乐指数最大的。

（2） 饮料总量为 2，从容量为 2^1 的饮料中选出快乐指数最大的（设为 H_1），与容量为 2^0 的饮料中快乐指数最大的（设为 H_0），比较 H_1 和 2* H_0，取出其中最大者为当前最佳选择

（3） 继续进行下去，直到求出 Opt（V，0）

粗看一下，有些似曾相识。我们在买书问题的时候也曾面临将买书计划拆分，然后查表进行贪心选择。然而买书问题每次选择至多选 M 本书，而且每次选择只影响下一次选择，所以只需要把 2M 进行有限的拆分即可。

而本题则不尽相同，对于某种容量 V’（以 V’=11 为例）来说，有两个问题：

1. 首先，我们需要察看所有拆分的可能性，找出其中最大者作为本次贪心选择的结果。其中，由于“每种饮料的单位容量都是 2 的方幂”，所以拆分结果仅考虑用小于 V’的 2 的方幂来进行组合，即（**计算式 1**）11=8+2+1，4+4+2=1，4+2+2+2+1，….，1+1+…+1。可以看到，对于 V’我们至少可以“拆”出 V’种组合（或许更多），即便我们把每次的计算结果用表格保存起来，我们的查找次数也至少是Ω（1+2+…+V’=V’^{2}），空间复杂度也很高，并没有如数中说“简单很多”。而且，如果取消“每种饮料的单位容量都是 2 的方幂”的限制，拆分的结果就会变成（**计算式 2**）11=10+1，9+2，…，5+5，5+5+1，5+4+2，…..，导致查找次数进一步增加。

2. 其次，让我们回到贪心选择的定义上来。由于贪心选择性，贪心算法的过程是每次选择都选取当前看来最好的结果，使得当达到最终状态时的结果刚好是最优解。而我们再看 V’=11 的例子，假设最优解是 4+4+2+1，则我们曾经计算过的 8 就不是最优解的一部分，这就和贪心算法的精神不符，所以这个方法其实还是动态规划。

3 总结

动 态规划解法很好，贪心算法有待商榷。其实这是正常的，因为通常情况下使用贪心算法的难点在于证明贪心选择性的存在，鉴于这本书的定位，我们不能苛责更多。 但是这个问题和上一个问题（买书折扣问题）的贪心算法我觉得都过于草率。如果想让这本书成为经典，“微软面试”的噱头是远远不够的，精益求精的态度至关重 要，有时候真的没必要把 Deadline 设置的那么紧迫，偶尔跳票追求质量也是理所应当，看看人家暴雪的“星际争霸 2”跳票这么多年就知道了…….

4 《编程之美》本问题勘误

[1] P41，第七段，“ Opt（V’，i）表示从第 i，i+1，…，n-1，n 种饮料…”，应该把 n 去掉，因为 i 的取值范围是[0，n-1]。

[2] P41，第八段，“ Opt（V，n）”应该改为 Opt（V，0）。

[3] P41，推导公式，应改为 max{ … +Opt(V’-Vi * k, i+1)}（…，i=n-2，n-1，……，0）

[4] P42，代码清单 1-9，for (int i=0; i <= V; i++)，应改为 for (int i=1; i <= V; i++)，因为 Opt[v][n]的第一行都是 0，就不用计算了。

[5] P42，代码清单 1-9，“ if (i<=k*V[j])”应该改为 if (i < k*V[j])，因为显然等号情况应该保留

[6] P42，最后一段，应改为计算 Opt[i][j]时只需要 Opt[k][j+1](0<=k<=V)这一列，所以不必列出整个矩阵，而只需要两列即可。

[7] P44，解法三种的表的第三行，TC0_1，应该改为 TC0_n0

编程之美 - 读书笔记 - 连连看游戏设计收藏

新一篇: 不出十年，纯粹意义上的软件开发即将死亡 | 旧一篇: 火狐 3 本月 17 日发布 与 IE 8 设计理念完全不同

作者：薛笛 联系方式：jxuedi(Gmail 邮箱--@gmail.com)

看着本书主页上长长的勘误表，我真的替能拿到第一版第二次印刷的朋友们开心，相信在经过调整之后阅读效果会更好。同时，本书的作者和编辑没有匆忙推出第二 版也是一种很负责任的行为，花多一点的修改酝酿，会让我对第二版产生更多的期待，这也是一本书成为经典的必经之路。虽然我在上一篇书评中比较激烈的批评了 书中算法论证不够严密的缺点，但是那是因为我没看到 1.11-1.13 中 描述的拈石头游戏问题，这三节看过之后心里唯有“爽快”二字，要是所有章节都能达到这种质量，我觉得就没有任何遗憾了。但是所有章节都这样写可能会令很多 人感到枯燥，毕竟众口难调，而且有些问题的解决也确实不需要细到这种程度；此外，这本书是多个作者合著，风格上确实很难做到统一。不过这些难题还是留给作 者头痛好了，我还是开开心心写书评。

1 问题描述及分析

连 连看游戏是一种很流行的小游戏，记得在小时候去游戏厅玩街机的时候就有两台专门的连连看的机器（当然当时不叫这个名字），一个是连麻将牌、另一个是连水果 图片。当时的麻将牌分好几层，相邻层的牌还可以连，看得人眼花缭乱。真是佩服能玩这游戏的人，我还是打“拳皇”比较擅长☺。书中给出的 Microsoft Link-up 界面很漂亮，不过似乎图形只有一层，而且数量也不是很多，降低了不少难度，连我也能玩了☺。

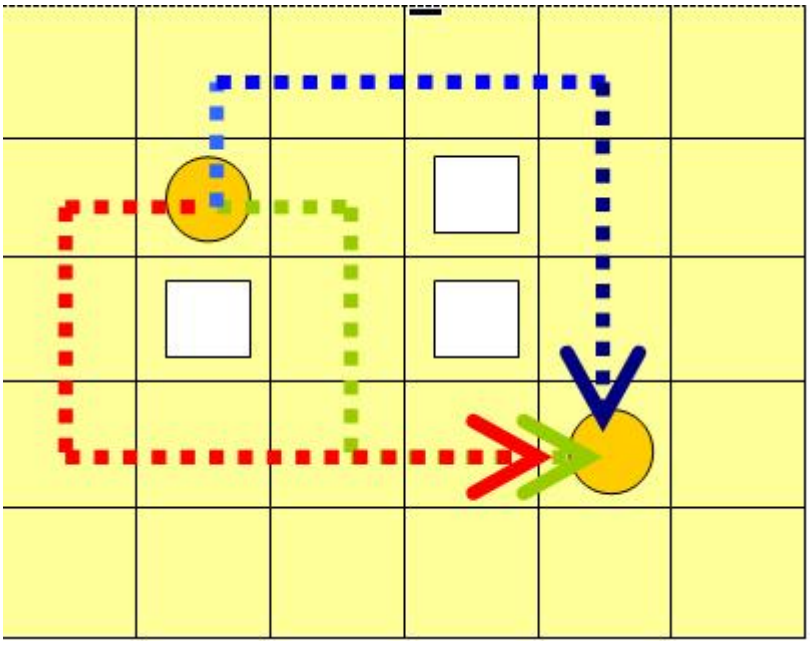


图 1 广度优先路径搜索

书中给出的解法利用了广度优先搜索算法，本质上是一种建立搜索树然后剪枝的策略。具体过程如图 1 所示：目标是要找到从左上角的圆形图形（设为图形 1）到右下角的圆形图形（设为图形 2）之间不超过 3 个弯的最短路径。首先在图形 1 所在位置向四个方向进行直线查找，如果在某一方向上找到图形 2 则结束。否则，记录下所有这四个方向上的空白格子，并在每一个空白格子的其他 3 个方向上进行直线搜索。如果仍未找到图形 2 则再次记录下查找过程中沿途记录下来的空白格子（要去掉已经走过的空白格子），仍然在 3 个方向上进行搜索。若找到则返回结果，若还未找到则说明两个图形之间不存在不超过 3 个弯的路径。这个算法很直观，就是**在有限的转弯次数内不断构建出更大的路径网络**，看看能否“覆盖”到被连接的图形，但似乎还可以进一步提高效率，下面就给出我的一些想法。

2 相同图形对之间的最短路径查找算法

以图 2 为例，我们的目标是找到图形 1 和图形 2 之间的最短路径，我们从图形 1 出发进行查找动作。由于题目要求最短路径不能超过 3 个弯，并且我们知道：“直线路径长度≤带有一个弯的路径长度≤带有两个弯的路径长度”，所以下面我们将分三种情况进行讨论。

为了加速程序运行，这里还引入了图 3 所示的两个辅助数据结构，其中每一个方块中都带有一个二进制位，用来表示在该位置上的格子是否是空格子，0 代表是空格子，1 代表有图形存在。虽然左右两组数据结构的内容是一样的，但是其组织方式有所不同：左边的二进制位是按列组织的，即一列的二进制位保存在一个 int 或者 long 型的变量中，多出来的位用 0 填满；右边的结构是按行组织的，其他与左边结构类似。这两个结构具体的功能后面用到的时候会详细介绍。

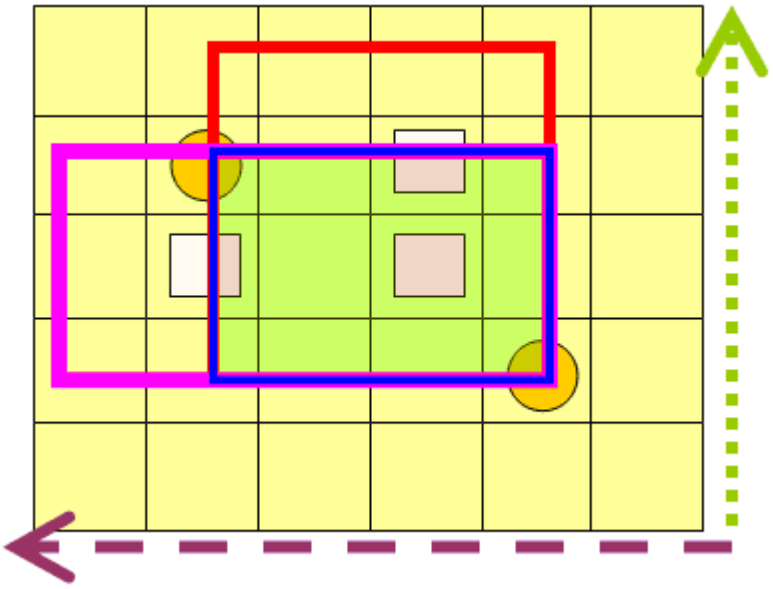


图 2 改进的路径查找方法示意图

首先，应该判断是否存在图形 1 到图形 2 之间的直线路径。这一点我们可以通过他们的位置信息直接得到，例如图形 1 的位置是[2,2]，图形 2 的位置是[4,5]，很显然二者不会在一条直线上，所以直接跳过这一步。但如果二者在一条直线上的话，还需要判断他们之间是否有障碍物存在。例如，如果图形 2 在[5,2]的话，我们可以断定二者在同一直线上，但我们还需要检查在二者的直线路径上是否还有其他图形存在。因为两个图形同在第二列上，于是我们取图 3 左边结构的第二列变量 C2（这里假设是一个 8 位的 Byte 变量），然后做 val=C2&**0 0110**000，注意其中的两个 1 是用来测试在图 2 的第二列上是否有非空的格子。如果 val=0 则说明两个图形间存在直线路径，否则说明二者之间不存在直线路径。这一过程的时间复杂度为 O(1)。

第二步，查找二者之间是否存在转弯一次的路径。从图中我们可以清楚地看到，这样的路径只有两条，他们组成了一个矩形，图 3 中蓝色的矩形就是两条路径的组合而成。我们可以用上面的方法分别来测试这两条路径上是否有障碍存在，如果没有障碍则作为最短路径返回，否则就要进行第三步。这一步的时间复杂度也是 O(1)。

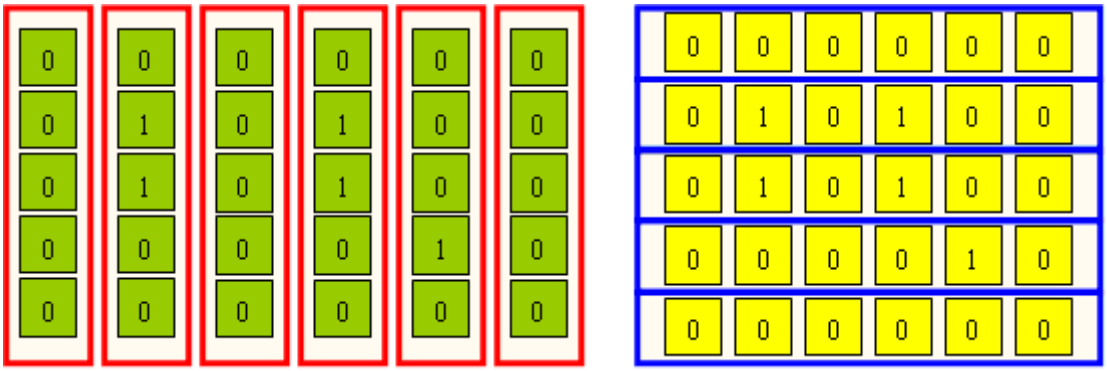


图 3 辅助数据结构

第三步，查找两个图形间是否存在转弯三次的路径。首先我们用书中给出的方法在蓝色矩形内部进行路径查找，但是限制只在“向右”和“向下”两个方向上进行操作。如果没找到所需路径，则需要再次扩大搜索范围。如图 2 所示，在这一步我们需要把搜索方向在“向

上”和“向左”两个方向上扩展，所获得的路径分别用红色框和粉色框表示，这个动作类似于把蓝色框向上和向左拉伸。对于获得的路径上的 3 条线段，我们只需用第一步所提到的方法来查找是否存在障碍即可，若 3 条线段都没有障碍则返回该路径作为最短路径；否则继续向两个方向扩展搜索范围直到搜索范围到达格子的边界，若仍未找到则认为两个图形间没有符合条件的最短路径。可以看出，由于不需要对同时对 4 个方向上所有空闲格子进行广度优先搜索操作，第三步的时间复杂度应该低于书中所给出的算法。

3 扩展问题的研究和修改意见

1.13.1 扩展问题 1 的研究

扩展问题 1 说的是(1)是否可以通过维护**任意两个格子之间的最短路**径来实现快速搜索。(2)在每次消去两个格子之后，自然更新要更新需要维护的数据，这样的思路有哪些优缺点，如何实现。

粗 略想来，由于用户每次只能消除一对图形，即只会用到一个最短路径，但由于实现并不知道用户会选择哪一对图形，所以需要事先计算出所有可能的最短路径并保存 起来。此外，采用这种方法的话似乎每次用户消去一对相同图像之后都需要重新计算出当前所有可能被连接的相同图形之间最短路径，这是因为当某些图像被消去之 后可能会产生很多新路径，而我们又不能确定这些空出来的格子到底能够影响哪些路径，所以就只好都重新计算一遍。其缺点很明显就是每次消去图形动作之后重新 计算所有可能的最短路径所需要消耗的时间；而该方法的优点则是可以很快地判断两个相同图形之间是否存在满足条件的最短路径。

如 果用户很厉害，每次都能选中可以消除的图形对，那么用这种方法浪费的时间就会相当可观，毕竟用户未选中的其他可以连接的图形对之间的最短路径都被浪费掉 了；而如果用户很差劲，每轮选择的次数都远远大于当前可能的连接数量时，该方法就会比书中正文提到的方法高效。但这种情况是比较少的，因为在整个游戏中用 户主要是会用眼睛“找”而不是频繁的用鼠标去“试”。所以总的来看，维护所有最短路径的方法的效率相对比较低。

1.23.2 修改意见

（一）书中给出了这个游戏几个关键点是：怎样求出相同图像之间的最短路径（路径的转弯数最少，经过的格子数最少，书中规定路径不能超过两个弯）。然而让人迷惑的是，在“分析与解答”的开始部分数中提到了用**自动机模型**来描述游戏设计，当时我想难道本章会有像字符串匹配问题那样利用自动机进行问题求解的方法吗？但是在后面却没有看到任何应用自动机理论来解决问题或进行算法优化的内容。虽然游戏设计时常常会用自动机来描述游戏的状态转换，但书中本章的主旨是寻找相同图像之间不超过 3 个弯的最短路径，并没有将自动机的能力应用到算法中以加速寻径过程；退一步说，如果只是讲游戏设计的通常做法的话，那又和本章的内容有什么联系？所以还是建议去掉这一段以免造成误解。

（二）关于扩展问题(1)的题干我想应该不是“任意两个格子”之间的最短路径，因为维护这样的路径是没有意义的。而应该是“任意两个带有相同图像的格子”之间的最短路径

才对。

（三） 老实说虽然书中附上源码无可厚非，但是以我的看书经验来说，我通常只是会找到关键部分详细看，很少能够把所有代码完整的看完，尤其是书中又给出很多不同语 言写成的代码又增加了不少阅读困难。我总是觉得像《编程之美》这类的书应该主要讲问题分析，讲实现算法，关键代码用类似算法导论那样的比较详细的伪代码列 在书里即可，这样才不会把重要的部分淹没在细节中。工程上的实现细节打个包用网站下载的方式提供也不错，即节省了书里的空间，也可以让用户在开发环境里读 代码提高阅读质量。还是那句话，风格统一是一件好事，我关注的是问题本身，而不希望一会 C#一会又 Python，类 C 语 言的伪代码就好很阿，大家都看得懂。用不用代码，用多少代码，用什么样的代码都是值得拿捏的，既不能回避困难的问题的实现细节、也不能在某些问题上给出过 多不必要的干扰因素，这样才能更有利于对原始问题的理解。以上是我的一家之言，仅供参考，如有不妥帖之处您一笑而过就好。

4 后记

答 辩完成，终于顺利毕业了，恭喜一下自己。在上班之前还有一些时间可以挥霍，记忆中在高考之后就没再拥有如此多的休闲时光。这段时间哈尔滨白天气很热，不适 合做户外活动，所以除了和同学朋友老师喝酒之外，就是希望能把这本书看完，写更多的书评与大家分享。同时也希望获得在算法方面更多更广泛的交流，毕竟不同 的思维才造就了丰富多彩的世界。“三人行必有吾师矣”，夫子的话还是要听阿。联络方式放在开头了，希望收到有相同兴趣的朋友的交流意见，大家共同进步。

最后，希望表达对四川地震灾区人民的祝福，借用总书记的话：再大的困难也难不倒英雄的中国人民！祝福四川，祝福中国！