Beauty of Programming

Broadview
www.broadview.com.cn 博文视点

编程之美

——微软技术面试心得

《编程之美》小组 著

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

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。当时令求职者“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，回来的时候都会被其他同学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”这一问题。

我自己申请加入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。至今还记得有一位面试者不断追问我论文中一个算法的收敛性时，我们进行了热烈讨论。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”。

程序虽然很难写，却很美妙。要想把程序写好，需要学好一定的基础知识，包括编程语言、数据结构和算法。程序写得好的人通常都有缜密的逻辑思维能力和良好的数理基础，而且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。我以前常出的一道面试题是“展示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时能找到编程的快乐，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之职的面试必读秘笈。

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

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呢？”

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。

当她忙着搜索的时候，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 的占用率为 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 的使用率保持在任意位置，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 CPU）的，那么你的程序会有什么样的结果？为什么？

自从 2005 年回到微软亚洲研究院后，我面试过不少应聘者，作为面试者，我最希望看到应聘者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。

我也为微软校园招聘出过考题，走访过不少软件学院，还为员工和实习生做过培训。我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上，忽视了对原理和技术的理解和运用。

写程序真的没有意思吗？为什么许多微软的员工和软件业界的牛人乐此不疲？我和一些喜欢编程的同事和实习生创作编写了这本书，我们希望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，在面试中展现出来的巧妙的思路、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”。

有时候会有同学问：“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。书中有一些关于面试的问答，我想它们可以回答这样一些疑惑。

本书的题目，一部分源自各位作者平时想出来的，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 的压力测试，听上去水分不少，我一边听一边琢磨“怎样才能考察一个人是否真正懂了 CPU，任务调度……”，后来就有了上面提到的“CPU 使用率”的面试题。有些题目来自于平时的实践和讨论，比如一些和游戏相关的题目。有些题目是随手拈来，比如我看到朋友的博客上有一道面试题，自己做了一下，发现自己的解法并不是最优的，但是，倒是可以作为一个面试题的题目，第 1 章的“程序理解和时间分析”就是这么得来的。书中有些题目在网上流传较广，但是网上流传的解法并不是正解，我们在书中加上了详细的分析，并提出了一些扩展问题。还有一些题目在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 45 分钟内解决，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 分钟内完成。

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。

作者们水平有限，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，不少解法仍可能有漏洞或错误，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（*www.msra.cn/bop*）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。

- **游戏之乐**：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思路能够对读者解决其他复杂问题有所帮助。
- **数字之魅**：编程的过程实际上就是和数字及字符打交道的过程。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。
- **结构之法**：对字符及常用数据结构的处理几乎是每个程序必然会涉及的问题，这一部分汇集了对常用的字符串、链表、队列，以及树等进行操作的题目。
- **数学之趣**：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来。
- **关于笔试、面试、职业选择的一些问答**：微软的面经，各种技术职位的介绍是很多学生所关心的内容，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. 大学计算机系、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. 面临求职笔试、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，平时可以随便翻翻，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，如果把这本书的写作比作一个软件项目，它有下面的各个阶段，每个阶段则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的，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，我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，作者们得到了微软亚洲研究院的许多同事的帮助，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。

邹欣

2007 年 11 月于北京

致谢

《编程之美》这本书从构思、编写到最后的出版，得到了许多同事和朋友的帮助。在此，作者们要特别感谢以下人士。

微软亚洲研究院的多位同事热情地与我们分享了他们觉得有意思的题目，他们分别是：邓科峰、宋京民、宋江云、刘晓辉、赵爽、李劲宇、李愈胜和 Matt Scott。

感谢微软亚洲研究院技术创新组的同事梁潇、殷秋丰，他们认真审阅了所有的题目和解答，找出了不少 bug[1]。技术创新组的另外几位优秀工程师李愈胜、魏颢、赵婧还帮助我们解决了书中的几个难题。

感谢研究院的同事、著名技术作家潘爱民对我们的鼓励，他审阅了全部稿件，并且提出了不少意见。

本书的封面和插图都出自研究院的实习生高霖之手，他在 10 余个构图都被否定的情况下，坚持不懈，最后拿出了“九连环”的封面设计，得到作者和出版方的一致认同。

在本书写作的过程中，作者们各自的“老板”——杨晓松、姚麒、田江森和刘激扬都给予了不少支持，在此特表示感谢。

作者们的“老板的老板”，研究院前任院长，微软公司资深副总裁沈向洋博士，现任院长洪小文博士对本书一直很关心和支持。沈向洋博士在百忙之中还亲自为本书写了序。

微软亚洲研究院市场部的金俊女士、葛瑜女士对本书的推广提供了很大帮助。

[1] 本书残留的 bug 都是作者们的责任。

负责 *www.msra.cn* 网站的徐鹏、马小宁、黄贤俊为本书设计了专栏。

感谢博文视点编辑团队。感谢在本书写作前期与我们合作过的编辑方舟，在写作后期参与合作的编辑徐定翔和李滟波。特别感谢自始至终和作者们一起工作的博文视点编辑周筠、杨绣国。他们和作者们一同构思，耐心修改，没有他们的不懈努力，以及细致的编辑和推广工作，就没有《编程之美》的成功上市。

面试杂谈

背景

每年从金秋九月起，校园里的广告栏中、BBS 上的招聘信息就逐渐多了起来。小飞是一名普通高校的应届计算机专业硕士毕业生，他勤奋好学，成绩中上，爱好广泛。看到身边的同学都在准备精美的简历，参加各种各样的招聘会，笔试、面试，他也坐不住了。他在 BBS 上看了各式各样的“面经”，也挤过招聘会上的人潮，长叹：“行路难，行路难，好工作，今安在？”

小飞从网上了解到了有关招聘的各种术语，他整理了一个列表：

名词	解释
面经	面试的经历。
默拒	投了简历，进行了面试，但是公司从此再也没有消息，询问也不回答。
Offer	公司给学生发的入职邀请。
群殴	通常指一群人一起参加面试，一般以多对多的形式同时进行，最后总是会有人被不幸淘汰，这一过程就叫做“群殴”。
听霸	凡校内招聘演讲会都出席旁听的。
投霸	凡公司招人都投简历的。
笔霸	凡投出简历都能得到笔试机会的。
面霸	凡参加笔试都有面试通知的。
巨无霸	在招聘过程屡屡被拒、机会全无的，江湖人称“巨无霸”!
霸王面	“霸王面”指没有获得面试资格，却主动找用人单位，要求面试的人，源自吃饭不给钱的“霸王餐”，即“没机会面，创造机会也要面”。

小飞获得了一个在微软亚洲研究院实习的机会，在工作中认识了一位有丰富招

聘经验的研发经理。他对经理进行了非正式的采访，希望能得到第一手的“内幕”消息。下面就是小飞整理出来的问答。小飞的问题用 Q 来标注，经理的回答用 A 标注。

典型面试

备注：在本文中，应聘者（英文为：candidate，interviewee）指应聘公司职位的学生或其他社会人士；面试者（英文为：interviewer）指公司里进行招聘和面试的人员。

Q：经理，您好。我就开门见山，能否分享一下当年您第一次去面试的故事？

A：好，大学毕业后，我进入了学校“产业办”开的公司。有一天，一家美国公司（我们姑且叫它 H 公司）来招人，这是我的第一次面试。那个公司的代表和我寒暄之后，递给我一道题目，题目大意是“写一个函数，返回一个数组中所有元素被第一个元素除的结果”。我当时还问了一些问题，以确保理解无误，所谓 clarification 是也。那位面试者简单地解释了一下，然后就在电脑上敲敲打打，也不理我了。我想这也不难，如何能显示我的功力呢？于是我就把循环倒着写 `for (i=n; i>=0; i--)`，因为我当时看到一本 Unix 书上是这么写的。

代码大概是这样的：

```
void DivArray(int * pArray, int size)
{
     for (int i = size-1; i >= 0; i--)
     {
         pArray[i] /= pArray[0];
     }
}
```

写完之后，他看了看就问我，你为什么要这么写循环？如果不这么写可以么？我说，也可以呀。他问了两遍，如果正着写循环会出现什么问题。我想，能有啥问题？就把循环正着写。噢，原来陷阱在这里！你知道这个陷阱是什么吗？

Q：让我想一想，知道了，如果循环从数组的第一个元素开始，并且不用其他变量的话，在循环的第一步，第一个元素就变成了 1，然后再用它去除以其他元素，就不符合题目要求了。

A：对，同时还有另一个陷阱——看看你是否会检查除数为零的情况，以及对参数的检查，等等。

Q：这不是很简单么？一会儿就写完了。

A: 面试题大多数不难，但是通过观察应聘者写程序的实际过程，面试者可以看出应聘者的思维、分析、编程能力。面试者一般还会有后面几招留着。比如，如果你要测试刚才写的这个函数，你的测试用例有多少？或者改变一些条件，能否做得出来？

Q：很多人说，面试是一个不公平的游戏，因为信息不对称。比如：面试者知道问题的答案，而应聘者不知道，面试者知道今年公司要招几个人，而应聘者不知道。

A：但是，应聘者手头有几个 Offer，面试者也不知道。应聘者是否喜欢公司提供的职位和薪酬，面试者也不知道。一方面，应聘者在“求”职，另一方面，面试者也在“求”才。面试也是一个增进双方互相了解的有效途径。

既然扯到了“信息不对称”，我再讲一个我的故事，当年 H 公司来我校面试的时候，我对 H 公司的了解仅限于 H 公司捐赠给我们计算机系的一个有些过时的小型机系统。我想，这个 H 公司是不是还有一些新东西？那时候还没有互联网，于是我就托人借了几本原版的 Byte 杂志来看，那是很厚的一本杂志，非常多的广告，看了半天，夹在杂志中的小广告掉了一地。我只看到杂志对 H 公司新出的一个桌面管理软件“NewWave”的评价，我琢磨了半天，大概搞懂了这是一个什么东西，市场上还有什么竞争对手，等等。

过了两天，面试开始了，对方端坐在沙发里问“你对我们 H 公司有何了解？”我先说了 H 公司的小型机系统，然后说，“By the way，我还了解了 NewWave”。于是我把看到的东西复述了一下。没想到对方坐直了身子，说这个 NewWave 就是他曾经领导的项目。于是我就根据杂志上的描述问，“您怎么看某某竞争产品？”他很兴奋地跟我谈了 NewWave 是如何的领先，等等。后来我们又聊了不少相关的东西。

最后所有人面试结束之后，我们的领导说，美方觉得我很突出，知道不少东西，包括 NewWave，口语也很好。领导就要求我给所有人都介绍一下 NewWave，我只好把看到的东西又复述了一次。不久，H 公司过来面试的另一个经理不解地对我们领导说：“为什么你们这么多人知道 NewWave？”

前不久，我在面试的时候问一位同学，“你对微软亚洲研究院有什么了解？”他说，“没啥了解，昨天打电话叫我来面试，我就来了……”对于这样的同学，

信息的确是非常不对称，那他吃亏也是难免的了。还有一位在面试中发挥得很不好的同学跟我说，他特地没有做任何准备，因为他想显示他的“raw talent”……

Q：关于 Test（测试）的职位，有没有一些典型的题目呢？

A：有哇，典型的题目如给你一支笔，让你说说你如何测试——据说要测试 12 个方面；再比如判断一个三角形的特性（直角、钝角、锐角、等腰）——据说有 20 多个测试用例，这是要考察大家思考问题的全面程度和逻辑分析能力（测试用例见 4.8 节“三角形测试用例”）。

Q：网上有些非常流行的问题，都号称是从大公司流传出去的，是真的么？

A：对，是有一些题目比较常见，例如“下水道的井盖为什么是圆的”，还有一个问题一度非常流行，据说早期应聘 PM（Program Manager 程序经理）职位的应聘者大多曾碰到这个题目：

房间里有三盏灯，屋外有三个开关，分别控制这三盏灯，只有进入房间，才能看到哪一个电灯是亮的。请问如何只进入房间一次，就能指明哪一个开关控制哪一个灯？

传说在晚上，微软一些会议室的灯忽明忽灭，那就是一些还没有搞懂的同事们在实地钻研。

Q：我大概了解了 Dev/PM/Test 这三种工作的典型面试题，那么这些题目的答案别人都知道了，还怎么面试呢？

A：对，会有不少题目流传出去，这本来无妨。但是一些人知道答案之后，就开始背诵，或者原封不动地拿它去面试应聘者，忘了“知道答案”和“能做一个好员工”的关系。知道了题目的答案，就能做一个好的开发人员、项目经理，或者销售经理么？一个极端的情况会是：公司里每一个人都知道哪盏灯是由哪一个开关控制的，如何测试三角形的类别等，但是这个公司真能从此开发出更好的软件么？

一句话：关键不在于答案，而在于思考问题的方法，这也是我们没有“题库”的原因。

研发职位的选择

Q：微软及很多其他软件公司都有不少研发职位，名称不尽相同，而且还是缩写，能不能讲解一下？

A：不少同学对微软公司的各种研发职位（Discipline）并不太了解，我们在面试进行到一半的时候，经常发现一个应聘者其实更适合做其他类型的工作。当然这时我们可以调换面试的方向，但是对应聘者来说总不是一件好事。我刚好在 BBS 上看到了一篇文章，这篇文章从个人的角度出发，非正式地讲了 R&D 各个方向的特点，虽然并非完全正确，介绍也不一定全面，但是我们不妨看看。

aR：Assistant Researcher，助理研究员，也可以叫研究员助理，主要在“R&D”的“R”这一端，工作是读论文，提想法，被否决后再提想法（如此反复 N 次），赶在截止时间之前提交论文。aR 的想法得到初步验证之后，还要跟其他部门推销自己的想法，争取把想法变成产品。aR 的乐趣是能在一个领域中深入研究，发表论文，申请专利，每个专利申请（无论是否批准）都能让自己得一块黑色立方体石头（如图 1 所示）。好多人的桌面上堆了不少石头，好像他们没什么苦恼。aR 有时做的事情和 RSDE 差不多。aR 以后会成长为 Associate Researcher（副研究员）、Researcher（研究员）、高级研究员，等等。总之，最后就成了大家小时候特别梦想做的“科学家”。

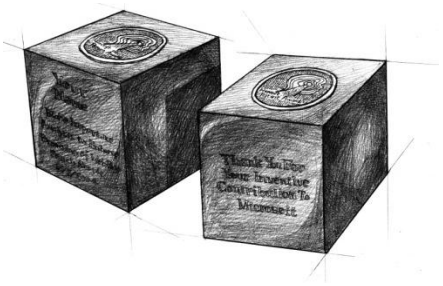

图 1　申请专利得到的石头

Dev：正式的名称叫 SDE（Software Development Engineer），这个职位和 aR 相对，是在“R&D”的“D”这一端。他们在一个产品团队中，按照严格的流程开发产品。MS 的一个产品发布之后，所有成员会得到一小块铁皮（学名叫“Ship-it Award”，如图 2 所示），上面写着产品的名字和发布日期。资深的 Dev 会收集到不少，他们会认真地把这些小铁皮整齐地贴起来，摆在办公桌最高的位置上。Dev 的乐不少，这里就不列举了。但是苦也有不少，比如产品的周期有时非常长，过程定义得非常完备（有时不免觉得太完备了）；比如要维护老版本；比如要用比较成熟的技术，而不是用最时髦的东西来开发产品。另外，Dev 要负责一个或几个

模块，这些模块不一定和最终用户打交道，未必是整个产品的核心模块。做一个好的 Dev 要生活在代码中，对代码和平台的各种细节要非常熟悉，掌握非常底层的技术，有些人以此为乐，有些人则未必。Dev 的职业发展道路很多，如果只想钻研技术，不乐意做很多管理工作，Dev 可以成为非常高级的工程师，直到杰出工程师（Distinguished Engineer）。当然，Dev 也可以成长为开发主管（Dev Lead）、开发总经理（Dev Manager），等等。

图 2　Dev 得到的小铁片 Ship-it

Test：正式名称是 Software Development Engineer in Test（SDET），简称为 Test 或 SDET（读作 S-DET）。这个职位看似没有 Dev 和 aR 酷，但是很有前途，首先中国的同学由于种种原因（不了解，看不起，做不来）不太愿意做这种工作，因此，公司找人非常急迫，相对容易进入。这一职位所谓的苦（也反映了一些人的偏见和误解）从传统意义上说，SDET 得等着上家（PM/Dev）给你东西，你才能“测试”。然而，现代软件工程要求 TEST 从项目一开始就积极参与项目的规划，了解客户需求，制定测试计划，设计测试架构，实现测试自动化，等等。事实上这些都是开发的工作，所以他们叫 SDE in Test。而且 SDET 能更深入地了解产品的各个模块是如何合作，如何在实际情况下被用户使用的。从代码之外理解程序，这是测试之乐。那种“产品发布前一个星期让测试人员来测一下”的情况在微软是不会发生的。那些只会用鼠标点击测试，然后报告 bug 的人员叫 Software Test Engineer（STE），这样的事一般会外包给别的公司。用足球比赛作比喻，Test 就是最后一道防线，如果你没有防守好 bug，bug 就会跑

到顾客那里去，因此 Test 工作非常重要。Test 的职业发展和 Dev 类似，一直到有专门管 Test 工作的副总裁（VP）。

PM：这恐怕是外界误解最多的行当，简而言之，Program Manager（程序经理）做的是开发和测试之外的所有事情。有些同学会问“我写程序都不用测试，那么除了开发和测试之外还有什么事儿?”在公司里开发商业软件可没有那么简单，比如有 10 个 Dev 和 5 个 Test 要在一起开发下一个版本的 MSN Messenger，那我们到底要做多长时间才能完成？什么事情先做，什么事情后做？项目进行到一半的时候，领导说我们改名叫 Live Messenger 吧，那这一改名意味着什么？如何调整进度？最后还剩下两个月的时候，看起来我们的确完不成全部任务，那要怎么办？你又不是 Dev 和 Test 的老板，他们凭什么听你的呢？这也是 PM 的苦。PM 的乐看起来在于，他们可以全盘掌控一个产品，广泛了解一个行业，和用户打交道，代表团队出席各种会议，在公司内部的曝光度也比较高。Dev/Test/PM 在产品开发中各负其责，互相协助，为共同的目标努力。产品成功发布之后，他们都会得到 Ship-it 小铁片儿。

RSDE：好了，我们最后看看 RSDE（Research SDE），这是微软亚洲研究院一个比较特殊的队伍。RSDE 的乐趣在于可以接触到各种最新的研究成果，并用它来解决挑战性的问题。RSDE 的苦在于项目都是 V0.1 版，而且做得成功的项目大多数会转化（Transfer）到产品组中，由别人推向市场。RSDE 在和研究部门合作的时候，就要负起 aR 和 PM（甚至 Test）的责任。刚开始，RSDE 既没有 R 的黑石头，又没有 D 的 Ship-it 小铁片。RSDE 参与的项目有比较大的风险，经常会不如预期，或者会失败（这也是科学研究的特点）。项目失败后，RSDE 掩埋了项目的尸体，擦干自己的血迹，又得找新的领域和新的项目。RSDE 还有“创新”的任务，这个词人人都会说，但是要做出来就不是那么容易了，全世界有这么多人在琢磨计算机，你能在什么地方做得比其他任何人都更进一步呢？这也是 RSDE 的乐趣吧。有些同学能力很强，兴趣广泛，但是一时也拿不准自己要深入研究哪一个领域，这时不妨来做 RSDE。做得好的 RSDE，他们的工作成果推进了研究，又走向了市场，这样就既可以拿到黑石头，又可以拿到 Ship-it 小铁片儿。我个人认为能有机会做 RSDE 是很令人自豪的事情，相当于参军当上了特种兵，很好，很强大。

Q：*看起来真是眼花缭乱……*

A：总之，每类职位都很重要，都有存在的理由，都有不错的发展前景，都有自己的苦和乐。微软很大，微软中国研发集团（CRD）内部有很多不同的机构和部

门，这也意味着有许多机会，让有能力的同学尝试 aR、Dev、Test、RSDE、PM 的职位。

求职攻略之笔试答疑

微软中国每年都会举行几次技术笔试，2006 年的笔试结束后，主持笔试的经理回答了学生提出的很多问题，小飞把这些问答整理如下（下文的“我们”指的是策划并批改试卷的技术人员）。

Q：笔试的难度是不是有些太难了？

A：从分数看，参加笔试的同学普遍得分较低，这说明不少同学大大低估了试题的难度，或者说低估了我们对答案的期望。一言以蔽之，我们希望看到接近“职业”水平的答案。

Q：为什么有些人笔试得了负分呢？

A：这是因为我们对选择题采用了“不做得零分，做错倒扣分”的判卷策略。公司的大部分同事们认为倒扣分是比较有效的甄别方法。而且我们尽量避免非常偏僻的知识点和有争议的答案。

Q：你们是不是只选取了其中一些卷子判分？

A：我们对大多数的卷子全部判分，每个部门都会抽调不少工程师加班判卷，同学们写的每一行文字都会被看到，对于一些很难读通的程序，我们还会一起分析，不会因为一眼看不懂就给个 0 分。对于单项题答得非常好的同学，我们会特别标记。像这样的无绝对标准答案的试卷，判卷是相当累人的活儿。至于是否全部判分，会不会把所有分数都全部告知考生，这由各个部门决定。

Q：笔试题目全是英语，这究竟是考英语还是考技术？为什么不用中文出题呢？

A：微软公司的工作语言是英语，公司在中国的各个部门（研究院，工程院等）都是如此。我们注意在考卷中不用很生僻的词汇，以免影响同学们的发挥。在有些题目中，我们还增加了一些注释，并且有一些小题目注明可以用中文回答。有些考生英语写得不错，起承转合，很像 GRE/TOEFL 的作文，可惜只有结构，实质内容不多，得分也不多。

Q：笔试的题量为什么这么大？很多人根本没有足够的时间做完！

A：每次开发新的软件，我们的时间也不够，这就是做软件项目的特点。 我们看到很多同学有些大题一个字也没有写，感到很可惜。其实，如果时间安排得当，至少应该每一道题试着回答一些基本问题。我们的很多监考人员也会提示大家注意时间分配。况且，如何在有压力的情况下最有效地分配时间，这也是一个人非常重要的能力。

Q：我觉得我回答得不错，每道题目都差不多做出来了，为什么分数很低？

A：有必要解释一下，我们的评分标准可能和学校里不一样。比如说有一道程序改错题，正确的解法要纠正 5 个错误。我们的评分标准是：

如果 5 个错误全部改正，满分。

如果找到 4 个错误，只能得一半分。

如果只找到 3 个错误，得 1/3 分。

如果只找到 2 个错误，得 1/4 分。

我们的评分标准要拉开“满分”和其他“差不多”的答案的距离。如果你每一道题目都“差不多”，那你的总分将是全部分数的一半以下。

Q：我会 C#、VB.NET，为什么微软的笔试偏偏要求用 C 语言答题？

A：对于微软的工程师来说，C 语言是基本功。

Q：为什么我投一个技术支持的职位也要用这么难的题来折磨我？

A：因为投同一个位置的人太多了。大家的简历都很优秀，所以只好用笔试来进行一次筛选。

Q：考题包罗万象，甚至包括我不熟悉的知识领域，难道微软需要的是“全才”吗？

A：我们的考试是想考察在实战中的基本知识和基本技能。考试不是万能的，笔试总分很高的同学，也有在面试中表现得很不如人意的。如果有人在某些题目中有优异的表现，即使总分不高，我们也会考虑的。

Q：我申请的职位比较特别，自己的专长没有能够显露，通过这样的一个考试不能真实反映出个人特点，有什么办法呢？

A：这一点我们同意。我们考试的主要目的是把所有考生中的优秀学生选出，并安排他们进入下一轮。至于在某一方面有专长的同学，他们应该直接和有关部门联系，或者我们的有关部门应该直接联系这些同学，例如在某些研究领域发表过高水平文章的同学。

Q：笔试通不通过是不是还有些运气成分在里面？

A：当然有，大家也都知道，一次笔试不能够反映一个人完全的、真实的水平。同学们寒窗十多年，经历了无数闭卷考试，作为一个过来人，我觉得职业生涯和人生不是一次两小时的闭卷考试能决定的，希望这样的笔试是大家人生中倒数第几次的闭卷考试之一。人生是更加开阔、充满更多变数的开卷考试。不管是开卷、闭卷，都是一分耕耘，一分收获。

求职攻略之决胜面试

经历了笔试、电话面试之后，许多同学接到了微软公司的邀请——来公司进行面对面的考察。

Q：既然微软这么重视实际的能力，每一个人都会经过几轮面试的考察，在学校时的学习成绩是否就不重要了？

A：也不一定。同样，关键不是在于静态的成绩，而是通过成绩了解成绩取得的过程，了解一个人的特质。曾经有一个面试者详细询问了一个应聘者在学校里的各种表现，最后在面试报告中写道：“我详细询问了她从中学到大学、研究生的情况，她在学校里没有一科的成绩是非常拔尖的，也没有太坏的成绩。她从来没有做过出格的事情，如逃课、自己写一些程序、打工等。我在她身上看不到对卓越的追求，也没有看到她有实现自身价值的想法……所以我认为本公司不应该雇用她。”

Q：虽然我没什么想法，但我觉得微软太有名了，我也不用多想了，我就是要进这样的公司，你叫我干什么都可以！

A：我们恰恰不太需要没什么想法的人，这也许和企业文化有一些关系。在中国一些企业的文化中，往往是领导安排你做什么，你就做什么。在微软，我们认为每个人都是独立的个体，我们希望雇员能够“在其位，谋其事”，同时能考虑到自己三五年后的发展，并且能自己制定计划去实现事业目标，这是公司的文化。

Q：面试的时候要穿什么衣服？

A：在没有特别规定的情况下，穿你觉得舒服的衣服就行。我们看到不少应聘者穿着明显不舒服的西装来面试，这样不会给自己加分，当然也不会减分。但是自己太不舒服，会影响发挥。

Q：不舒服没关系，只要你们公司觉得舒服，我就舒服。

A：我们刚刚说过，微软更看重的是“你”是否觉得舒服，“你”要做什么，以及“你”有什么创意。

Q：有没有在面试中作弊的呢？

A：说起来，还真有。有一天，我在微软外面的一个中餐馆吃晚饭，这个餐馆很小，大家坐得比较挤，我不得不听到邻座的高谈阔论。原来是一个刚刚在微软面试过的学生在和几个同学聚餐，他很兴奋地谈着当天面试的经历——

“他问了那个在链表中找回路的问题了么？”

“问了，我假装思考了一下，稍稍试了试别的解法，然后就把你说的那个解法讲了出来……”

对于这种人，我们内部叫“Poser”——摆姿势的人。如果你在面试时恰好被问到了一道知道答案的题目，你可以向面试者提出来。摆姿势的话，万一被戳破，会比较难堪。既然你已经花了时间了解解法，不妨和面试者深入地探讨一下。

Q：大家发表在 BBS 上的面经，公司看不看？

A：公司的一些员工也在看，有一次，HR 在某 BBS 上看到一篇很详细的面经，文笔生动，此文章从他看到 HR JJ 的那一刻写起，直到做了什么题目、怎么做的、说了什么话、最后如何走出了公司大门他都做了详细记录。从描述上看，我们很容易就能推断出这是哪一位应聘者。他似乎发挥得很不错，可惜他忘了在开始面试的时候，HR JJ 给他讲的，他也签了自己大名的保密协定。对于这样的同学，我们只能遗憾地放弃了。

Q：整个面试过程中我觉得自己答得很不错了，面试者指出的问题我大部分都能回答出来，为什么我还是没有通过？

A：一个原因是有比你更厉害的应聘者，另一个大家容易忽略的原因是，应聘者和面试者对于“不错”的定义是不一样的（参见对笔试问题的回答）。

对于在校学生，觉得自己写的程序，涂涂改改，大概逻辑能通过就行了，面试者指出的问题能答出来一些就行了。但是对于将来的公司员工，我们要考察：程序设计的思路如何？编程风格如何？细节是否考虑到？程序是否有内存泄露？是否采用了最优算法？是否能对程序进行修改以满足不断变化的需求？是否能举一反三？

另外，除了专业技巧，我们在面试中还会考察应聘者的职业技巧（professional skills，也有人称为 soft skills）。这个人的交流能力、合作能力如何，对自己的评价和期望是什么？在有压力的情况下，能否发挥水平？是否追求卓越？这些“非技术”的因素相当重要。

Q：很多有名的企业面试只要求谈谈就可以了，为什么微软一定要写代码？

A：我们的绝大部分工作，都通过代码而来，很大一部分的问题，也是由代码所导致的。所以我们不能不重视写代码这件事。当然有很多其他工作不需要写代码，但这不在我们的讨论范围内。

有一次我在过道上碰到一个同事陪着一个应聘者走出大楼，这位应聘者边走边侃侃而谈。后来我问这位同事详情。他说，“这位先生表达能力不错，但是当我叫他写一个小程序的时候，他死活不动手。他说在以前的工作中，如果要写代码，从 MSDN 上拷贝一些下来就行了。我和他僵持了一会儿之后，只好说，那你要是不写，我们就没什么可谈的了。所以后面的面试都没有必要了，我直接送他出了门。”

有一次我收到我们开发总经理的邮件，上面强调了面试的时候一定要让应聘者动手写代码等，这时对面的一位同事不好意思地说，他今天碰到的应聘者是以前朋友的朋友。两人聊了很长时间的闲话，后来他不好意思叫他写代码，时间也不够了，于是就写了一些反馈，说这人看起来还行。没想到开发总经理眼尖，把这个问题揪出来了。

Q：市场上有很多号称宝典的面试书籍，这些的确是外企用的面试题目么？我看到一本，就像是网络上流传的各种面经的汇编，好像没有太大的价值。

A：我觉得最好的技术面试“宝典”，就是讲算法和数据结构的经典著作。微软亚洲研究院的工程师们在长期的面试过程中，也收集了一些有意思的面试题目，叫《编程之美》，听说马上就要出版了。

Q：太好了！这本书里面一定有无数的源代码供学生们钻研吧？

A：其实，大部分题目都不需要连篇累牍的程序来解决，聪明的解法通常是非常简明的。药灵丸不大，棋妙子无多，程序也是这样，许多题目的核心算法就是寥寥几行。这可以说是编程之美的一种表现形式。我们面试就是要寻找能体会到编程之美的人。

另外，我们的这一番对话应该给微软的技术面试做了相当的“去神秘化”（demystified）的工作。我还要提醒同学们要“去粉丝化”——不要像极品粉丝

追逐明星那样，如果明星不能满足自己见一面的要求（或者其他要求），就觉得天旋地转，痛不欲生。如果你经过努力，仍然没有进入微软公司，你并非一无是处，天也不会塌下来。微软公司不过是很多软件公司中的一个，它要寻找“合适”它条件的员工，这个公司不合适你，还有下一个，或者干脆你自己开创一个吧。

Q：技术面试还有什么特别的诀窍么？

A：微软全球资深副总裁，亚洲研究院的前任院长沈向洋博士经常讲的一句话是“Nothing replaces hard work”，既然同学们知道技术面试不外乎就是这些类型的题目，那大家就自己动手做一遍好了。 如果实在做不出来，可以学习《编程之美》或其他书上详细的讲解。

Q：我自己解答问题太慢了，能把《编程之美》书上的解法背下来，这也是一种捷径吧？

A：有时要小心这样的“捷径”。我想起以前考大学的一件事儿。当时有一本很厚的英语标准化考试模拟题，不少同学都买来做。另一位同学从学长那里得了一本做过的书，我们在做题的时候，他说：“我不用做了，我已经有答案了，我平时看看答案就行了，一样的。”结果高考的时候，他的英语考得很不好。

所以，对于认为只要买了一本《编程之美》，或者其他宝典，就好像得到了入职捷径的同学，我要提醒一下：小心这样的捷径！纸上得来终觉浅，绝知此事要躬行。

小飞的总结

结束了和研发经理的几次对话之后，小飞陷入了深思。他发现面试并不一定是用难题、偏题来考倒人，笔试和面试考察的都是自己在编程、解决问题、与人合作等方面的全面能力。运气和背好的答案并不能帮助他解决所有的问题。微软公司花费很多人力物力来寻找合适的人才[1]，那自己如何能展现能力，让伯乐相中？他做了如下的总结。

1. 知己知彼。知己，就是要了解自己的能力、兴趣、职业发展方向；知彼，就是要了解公司的文化、战略方向和择才标准。
2. 笔试就是基础，用扎实的理解和考虑完备的解答来征服阅卷者。

[1] 要更多地了解微软，特别是微软亚洲研究院的方方面面，请访问研究院网站：*www.msra.cn* 和博客网站：*http://blog.sina.com.cn/msra*。

3. 面试就是探讨，用缜密的代码和严密的分析赢得未来同事的尊重。思考问题的方法比结果重要，面试者会更加在乎你解决问题的思考过程。

4. 你的工作就是最好的面试，不要把时间花在寻找捷径和背诵答案上，要通过实际的工作和产品来体现自己的水平。

千里之行，始于足下，要想在入职竞争中脱颖而出，自己得先下苦功夫，在平时就要用职业的标准来要求自己。他相信，只要自己付出了足够的努力，就会有收获——“长风破浪会有时，直挂云帆济沧海”。

目　录

CONTENTS

第 2 章

数字之魅

—— 数 字 中 的 技 巧

面试是双方平等交流的过程，有时候分不清谁在面试谁。

这一章收集了一些好玩的对数字及数组进行处理的题目。编程的过程实际上就是和数字打交道的过程。很多庞大的应用，例如搜索引擎查询并返回搜索结果的过程，就可以看作是对众多数组和数组中大量的数字（如：Page Rank，Page Id）进行计算、比较的过程。我们一方面不断地说要处理“海量数据”，另一方面同学们在程序课上定义数组的时候会写 int array[10], int array[100]往往就觉得“技止此耳”，“我掌握了”！如果数组的元素个数是百万、千万级，你的算法还有效率么？

有些题目看似简单，但是我们在面试中发现，有很多应聘者不能正确地写出“冒泡排序”或 “二分查找”，所以还是要从简单的题目出发。能不能把简单的问题完完全全地解决，没有任何 bug？

有读者会问——

那这些题目我都背好了，再来面试，行么？

当然行。比如“求数组的子数组之和的最大值”（见正文之“2.14”节）这道题目，正确的解法只有不到 10 行代码。你当然可以背好了再来面试。不过面试者肯定会问一些扩展问题，像“如果数组首尾相连，怎么办”，“如果要求数组的子数组乘积的最大值”等。不能举一反三的同学，可能会比较难过。你只有真正掌握了这些内容，才能应付自如。

2.1 ★★★ 求二进制数中 1 的个数

对于一个字节（8bit）的无符号整型变量，求其二进制表示中“1”的个数，要求算法的执行效率尽可能地高。

分析与解法

大多数读者都会有这样的反应：这个题目也太简单了吧，解法似乎也相当地单一，不会有太多的曲折分析，或者峰回路转之处。那么面试者到底能用这个题目考察我们什么呢？事实上，在编写程序的过程中，根据实际应用的不同，对存储空间或效率的要求也不一样。比如在 PC 上的程序编写与在嵌入式设备上的程序编写就有很大的差别。我们可以仔细思索一下如何才能使效率尽可能地“高”。

解法一

可以举一个八位的二进制例子来进行分析。对于二进制操作，我们知道，除以一个 2，原来的数字将会减少一个 0。如果除的过程中有余，那么就表示当前位置有一个 1。

以 10100010 为例：

第一次除以 2 时，商为 1010001，余为 0。

第二次除以 2 时，商为 101000，余为 1。

因此，可以考虑利用整型数据除法的特点，通过相除和判断余数的值来进行分析。于是有了如代码清单 2-1 所示的代码。

代码清单 2-1

```
int Count(BYTE v)
{
    int num = 0;
    while(v)
    {
        if(v % 2 == 1)
        {
            num++;
        }
        v = v/ 2;
    }
    return num;
}
```

解法二：使用位操作

前面的代码看起来比较复杂。我们知道，向右移位操作同样也可以达到相除的目的。唯一不同之处在于，移位之后如何来判断是否有 1 存在。对于这个问题，再来看一个八位的数字：10100001。

在向右移位的过程中，我们会把最后一位直接丢弃。因此，需要判断最后一位是否为 1，而“与”操作可以达到目的。可以把这个八位的数字与 00000001 进行“与”操作。如果结果为 1，则表示当前八位数的最后一位为 1，否则为 0。如代码清单 2-2 所示：

代码清单 2-2

```
int Count(BYTE v)
{
    int num = 0;
    while(v)
    {
        num += v & 0x01;
        v >>= 1;
    }
    return num;
}
```

解法三

位操作比除、余操作的效率高了很多。但是，即使采用位操作，时间复杂度仍为 $O(\log_2 v)$，$\log_2 v$ 为二进制数的位数。那么，还能不能再降低一些复杂度呢？如果有办法让算法的复杂度只与“1”的个数有关，复杂度不就能进一步降低了吗？

同样用 10100001 来举例。如果只考虑和 1 的个数相关，那么，我们是否能够在每次判断中，仅与 1 来进行判断呢？

为了简化这个问题，我们考虑只有一个 1 的情况。例如：01000000。

如何判断给定的二进制数里面有且仅有一个 1 呢？可以通过判断这个数是否是 2 的整数次幂来实现。另外，如果只和这一个“1”进行判断，如何设计操作呢？我们知道的是，如果进行这个操作，结果为 0 或为 1，就可以得到结论。

如果希望操作后的结果为 0，01000000 可以和 00111111 进行“与”操作。

这样，要进行的操作就是 01000000 &（01000000 – 00000001）= 01000000 & 00111111 = 0。

因此就有了解法三的代码清单 2-3：

代码清单 2-3

```
int Count(BYTE v)
```

```
{
    int num = 0;
    while(v)
    {
        v &= (v-1);
        num++;
    }
    return num;
}
```

解法四：使用分支操作

解法三的复杂度降低到 $O(M)$，其中 M 是 v 中 1 的个数，可能会有人已经很满足了，只用计算 1 的位数，这样应该够快了吧。然而，我们说既然只有八位数据，索性直接把 0~255 的情况都罗列出来，并使用分支操作，可以得到答案，如代码清单 2-4 所示：

代码清单 2-4

```
int Count(BYTE v)
{
    int num = 0;
    switch (v)
    {
        case 0x0:
            num = 0;
            break;
        case 0x1:
        case 0x2:
        case 0x4:
        case 0x8:
        case 0x10:
        case 0x20:
        case 0x40:
        case 0x80:
            num = 1;
            break;
        case 0x3:
        case 0x6:
        case 0xc:
        case 0x18:
        case 0x30:
        case 0x60:
        case 0xc0:
            num = 2;
            break;
            //...
     }
     return num;
}
```

解法四看似很直接，但实际执行效率可能会低于解法二和解法三，因为分支语句的执行情况要看具体字节的值，如果 *a* =0，那自然在第 1 个 case 就得出了答案，但是，如果 *a* =255，则要在最后一个 case 才得出答案，即在进行了 255 次比较操作之后！

看来，解法四不可取！但是解法四提供了一个思路，就是采用空间换时间的方法，罗列并直接给出值。如果需要快速地得到结果，可以利用空间或利用已知结论。这就好比已经知道计算 1+2+ … +*N* 的公式，在程序实现中就可以利用公式得到结论。

最后，得到解法五：算法中不须要进行任何的比较便可直接返回答案，这个解法在时间复杂度上应该能够让人高山仰止了。

解法五：查表法

代码如清单 2-5 所示。

代码清单 2-5

```
/* 预定义的结果表 */
int countTable[256] =
{
     0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3,
        3, 4, 3, 4, 4, 5, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, 2, 3, 3,
        4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4,
        3, 4, 4, 5, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 2, 3, 3, 4, 3,
        4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6,
        6, 7, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, 2, 3, 3, 4, 3, 4, 4,
        5, 3, 4, 4, 5, 4, 5, 5, 6, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
        3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 2, 3, 3, 4, 3, 4, 4, 5, 3,
        4, 4, 5, 4, 5, 5, 6, 3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 3, 4,
        4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 4, 5, 5, 6, 5, 6, 6, 7, 5, 6, 6,
        7, 6, 7, 7, 8
};
int Count(BYTE v))
{
     //check parameter
     return countTable[v];
}
```

这是个典型的空间换时间的算法，把 0~255 中“1”的个数直接存储在数组中，*v* 作为数组的下标，countTable[*v*]就是 *v* 中“1”的个数。算法的时间复杂度仅为 *O*（1）。

在一个须要频繁使用这个算法的应用中，通过“空间换时间”来获取高的时间效率是一个常用的方法，具体的算法还应针对不同应用进行优化。

扩展问题

1. 如果变量是 32 位的 DWORD，你会使用上述的哪一个算法，或者改进哪一个算法？
2. 另一个相关的问题，给定两个正整数（二进制形式表示）*A* 和 *B*，问把 *A* 变为 *B* 需要改变多少位（bit）？也就是说，整数 *A* 和 *B* 的二进制表示中有多少位是不同的？

读者反馈

对于这个小小的问题，我们找出来四个解法，是否就已经“叹为观止”，“技止此耳”？其实不然，有些读者给出了更好的解法。其中之一转载于博文视点的官方博客上：

http://blog.csdn.net/bvbook/archive/2008/04/15/2292823.aspx

大家可以仔细分析，考虑一下各个算法的优劣以及使用的场景。

2.2 ★★ 不要被阶乘吓倒

阶乘（Factorial）是个很有意思的函数，但是不少人都比较怕它，我们来看看两个与阶乘相关的问题。

1. 给定一个整数 N，那么 N 的阶乘 $N!$ 末尾有多少个 0 呢？例如：$N=10$，$N!=3\,628\,800$，$N!$ 的末尾有两个 0。
2. 求 $N!$ 的二进制表示中最低位 1 的位置。

分析与解法

有些人碰到这样的题目会想：是不是要完整计算出 $N!$ 的值？如果溢出怎么办？事实上，如果我们从“哪些数相乘能得到 10”这个角度来考虑，问题就变得简单了。

首先考虑，如果 $N! = K \times 10^M$，且 K 不能被 10 整除，那么 $N!$ 末尾有 M 个 0。再考虑对 $N!$ 进行质因数分解，$N! = (2^X) \times (3^Y) \times (5^Z) \cdots$，由于 $10 = 2 \times 5$，所以 M 只跟 X 和 Z 相关，每一对 2 和 5 相乘可以得到一个 10，于是 $M = \min(X, Z)$。不难看出 X 大于等于 Z，因为能被 2 整除的数出现的频率比能被 5 整除的数高得多，所以把公式简化为 $M = Z$。

根据上面的分析，只要计算出 Z 的值，就可以得到 $N!$ 末尾 0 的个数。

问题 1 的解法一

要计算 Z，最直接的方法，就是计算 i（i =1, 2, …, N）的因式分解中 5 的指数，然后求和，如代码清单 2-6 所示：

代码清单 2-6

```
ret = 0;
for(i = 1; i <= N; i++)
{
    j = i;
    while(j % 5 ==0)
    {
        ret++;
        j /= 5;
    }
}
```

问题 1 的解法二

公式：$Z = [N/5] + [N/5^2] + [N/5^3] + \cdots$（不用担心这会是一个无穷的运算，因为总存在一个 K，使得 $5^K > N$，$[N/5^K]=0$。）

公式中，$[N/5]$表示不大于 N 的数中 5 的倍数贡献一个 5，$[N/5^2]$表示不大于 N 的数中 5^2 的倍数再贡献一个 5……代码如下：

```
ret = 0;
while(N)
{
    ret += N / 5;
    N /= 5;
```

```
}
```

问题 2 要求的是 $N!$ 的二进制表示中最低位 1 的位置。给定一个整数 N，求 $N!$ 二进制表示的最低位 1 在第几位？例如：给定 $N = 3$，$N! = 6$，那么 $N!$ 的二进制表示（1010）的最低位 1 在第二位。

为了得到更好的解法，首先要对题目进行一下转化。

首先来看一个二进制数除以 2 的计算过程和结果是怎样的。

把一个二进制数除以 2，实际过程如下。

判断最后一个二进制位是否为 0，若为 0，则将此二进制数右移一位，即为商值（为什么）；反之，若为 1，则说明这个二进制数是奇数，无法被 2 整除（这又是为什么）。

所以，这个问题实际上等同于求 $N!$ 含有质因数 2 的个数。即答案等于 $N!$ 含有质因数 2 的个数加 1。

问题 2 的解法一

由于 $N!$ 中含有质因数 2 的个数，等于$[N/2]+[N/4]+[N/8]+[N/16]+\cdots$[1]，

根据上述分析，得到具体算法，如代码清单 2-7 所示：

代码清单 2-7

```
int lowestOne(int N)
{
    int Ret = 0;
    while(N)
    {
        N >>= 1;
        Ret += N;
    }
    return Ret;
}
```

问题 2 的解法二

$N!$ 含有质因数 2 的个数，还等于 N 减去 N 的二进制表示中 1 的数目。我们还可以通过这个规律来求解。

[1] 这个规律请读者自己证明（提示：$[N/k]$等于 $1, 2, 3, \cdots, N$ 中能被 k 整除的数的个数）。

下面对这个规律进行举例说明，假设 N = 11011（二进制表示，下列 01 串均为整数的二进制表示），那么 $N!$中含有质因数 2 的个数为$[N/2]+[N/4]+[N/8]+[N/16]+\cdots$

即：
$$
\begin{aligned}
&1101 + 110 + 11 + 1\\
&=（1000 + 100 + 1）\\
&\quad +（100 + 10）\\
&\quad +（10 + 1）\\
&\quad + 1\\
&=（1000 + 100+ 10 + 1）+（100 + 10 + 1）+ 1\\
&= 1111 + 111 + 1\\
&=（10000 -1）+（1000 - 1）+（10-1）+（1-1）\\
&= 11011-（N \text{ 二进制表示中 1 的个数}）
\end{aligned}
$$

小结

任意一个长度为 m 的二进制数 N 可以表示为 $N = b[1] + b[2] * 2 + b[3] * 2^2 + \cdots + b[m] * 2^{(m-1)}$，其中 $b[i]$ 表示此二进制数第 i 位上的数字（1 或 0）。所以，若最低位 $b[1]$为 1，则说明 N 为奇数；反之为偶数，将其除以 2，即等于将整个二进制数向低位移一位。

相关题目

给定整数 n，判断它是否为 2 的方幂（解答提示：n>0&&（(n&（n-1））==0））。

2.3 ★ 寻找发帖“水王”

Tango 是微软亚洲研究院的一个试验项目。研究院的员工和实习生们都很喜欢在 Tango 上面交流灌水。传说，Tango 有一大“水王”，他不但喜欢发帖，还会回复其他 ID 发的每个帖子。坊间风闻该“水王”发帖数目超过了帖子总数的一半。如果你有一个当前论坛上所有帖子（包括回帖）的列表，其中帖子作者的 ID 也在表中，你能快速找出这个传说中的 Tango 水王吗？

分析与解法

首先想到的是一个最直接的方法，我们可以对所有 ID 进行排序。然后再扫描一遍排好序的 ID 列表，统计各个 ID 出现的次数。如果某个 ID 出现的次数超过总数的一半，那么就输出这个 ID。这个算法的时间复杂度为 $O(N * \log_2 N + N)$。

如果 ID 列表已经是有序的，还需要扫描一遍整个列表来统计各个 ID 出现的次数吗？

如果一个 ID 出现的次数超过总数 N 的一半。那么，无论水王的 ID 是什么，这个有序的 ID 列表中的第 $N/2$ 项（从 0 开始编号）一定会是这个 ID（读者可以试着证明一下）。省去重新扫描一遍列表，可以节省一点算法耗费的时间。如果能够迅速定位到列表的某一项（比如使用数组来存储列表），除去排序的时间复杂度，后处理需要的时间为 $O(1)$。

但上面两种方法都需要先对 ID 列表进行排序，时间复杂度方面没有本质的改进。能否避免排序呢？

如果每次删除两个不同的 ID（不管是否包含“水王”的 ID），那么，在剩下的 ID 列表中，“水王”ID 出现的次数仍然超过总数的一半。看到这一点之后，就可以通过不断重复这个过程，把 ID 列表中的 ID 总数降低（转化为更小的问题），从而得到问题的答案。新的思路，避免了排序这个耗时的步骤，总的时间复杂度只有 $O(N)$，且只需要常数的额外内存。伪代码如清单 2-8 所示：

代码清单 2-8

```
Type Find(Type* ID, int N)
{
    Type candidate;
    int nTimes, i;
    for(i = nTimes = 0; i < N; i++)
    {
        if(nTimes == 0)
        {
            candidate = ID[i], nTimes = 1;
        }
        else
        {
            if(candidate == ID[i])
                nTimes++;
            else
                nTimes--;
        }
    }
    return candidate;
}
```

在这个题目中，有一个计算机科学中很普遍的思想，就是如何把一个问题转化为规模较小的若干个问题。分治、递推和贪心等都是基于这样的思路。在转化过程中，小的问题跟原问题本质上一致。这样，我们可以通过同样的方式将小问题转化为更小的问题。因此，转化过程是很重要的。像上面这个题目，我们保证了问题的解在小问题中仍然具有与原问题相同的性质：水王的 ID 在 ID 列表中的次数超过一半。转化本身计算的效率越高，转化之后问题规模缩小得越快，则整体算法的时间复杂度越低。

扩展问题

随着 Tango 的发展，管理员发现，“超级水王”没有了。统计结果表明，有 3 个发帖很多的 ID，他们的发帖数目都超过了帖子总数目 N 的 1/4。你能从发帖 ID 列表中快速找出他们的 ID 吗？

2.4 ★★★ 1 的数目

给定一个十进制正整数 N，写下从 1 开始，到 N 的所有整数，然后数一下其中出现的所有“1”的个数。

例如：

N= 2，写下 1，2。这样只出现了 1 个“1”。

N= 12，我们会写下 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12。这样，1 的个数是 5。

问题是：

1. 写一个函数 f（N），返回 1 到 N 之间出现的“1”的个数，比如 f（12）=5；
2. 满足条件“f（N）= N”的最大的 N 是多少？

分析与解法

问题 1 的解法一

这个问题看上去并不是一个困难的问题，因为不需要太多的思考，我想大家都能找到一个最简单的方法来计算 $f(N)$，那就是从 1 开始遍历到 N，将其中每一个数中含有“1”的个数加起来，自然就得到了从 1 到 N 所有“1”的个数的和。写成程序如代码清单 2-9 所示：

代码清单 2-9

```
ULONGLONG Count1InAInteger(ULONGLONG n)
{
    ULONGLONG iNum = 0;
    while(n != 0)
    {
        iNum += (n % 10 == 1) ? 1 : 0;
        n /= 10;
    }

    return iNum;
}

ULONGLONG f(ULONGLONG n)
{
    ULONGLONG iCount = 0;
    for (ULONGLONG i = 1; i <= n; i++)
    {
        iCount += Count1InAInteger(i);
    }

    return iCount;
}
```

这个方法很简单，只要学过一点编程知识的人都能想到，实现也很简单，容易理解。但是这个算法的致命问题是效率，它的时间复杂度是

$$O(N) \times \text{计算一个整数数字里面“1”的个数的复杂度} = O(N * \log_2 N)$$

如果给定的 N 比较大，则需要很长的运算时间才能得到计算结果。比如在笔者的机器上，如果给定 N=100 000 000，则算出 $f(N)$ 大概需要 40 秒的时间，计算时间会随着 N 的增大而以超过线性的速度增长。

看起来要计算从 1 到 N 的数字中所有 1 的和，至少也得遍历 1 到 N 之间所有的数字才能得到。那么能不能找到快一点的方法来解决这个问题呢？要提高效率，

必须摈弃这种遍历 1 到 N 所有数字来计算 $f(N)$ 的方法，而应采用另外的思路来解决这个问题。

问题 1 的解法二

仔细分析这个问题，给定了 N，似乎就可以通过分析“小于 N 的数在每一位上可能出现 1 的次数”之和来得到这个结果。让我们来分析一下对于一个特定的 N，如何得到一个规律来分析在每一位上所有出现 1 的可能性，并求和得到最后的 $f(N)$。

先从一些简单的情况开始观察，看看能不能总结出什么规律。

先看 1 位数的情况。

如果 $N = 3$，那么从 1 到 3 的所有数字：1、2、3，只有个位数字上可能出现 1，而且只出现 1 次，进一步可以发现如果 N 是个位数，如果 $N>=1$，那么 $f(N)$ 都等于 1，如果 $N=0$，则 $f(N)$ 为 0。

再看 2 位数的情况。

如果 $N=13$，那么从 1 到 13 的所有数字：1、2、3、4、5、6、7、8、9、10、11、12、13，个位和十位的数字上都可能有 1，我们可以将它们分开来考虑，个位出现 1 的次数有两次：1 和 11，十位出现 1 的次数有 4 次：10、11、12 和 13，所以 $f(N)=2+4=6$。要注意的是 11 这个数字在十位和个位都出现了 1，但是 11 恰好在个位为 1 和十位为 1 中被计算了两次，所以不用特殊处理，是对的。再考虑 $N=23$ 的情况，它和 $N=13$ 有点不同，十位出现 1 的次数为 10 次，从 10 到 19，个位出现 1 的次数为 1、11 和 21，所以 $f(N)=3+10=13$。通过对两位数进行分析，我们发现，个位数出现 1 的次数不仅和个位数字有关，还和十位数有关：如果 N 的个位数大于等于 1，则个位出现 1 的次数为十位数的数字加 1；如果 N 的个位数为 0，则个位出现 1 的次数等于十位数的数字。而十位数上出现 1 的次数不仅和十位数有关，还和个位数有关：如果十位数字等于 1，则十位数上出现 1 的次数为个位数的数字加 1；如果十位数大于 1，则十位数上出现 1 的次数为 10。

```
f(13) = 个位出现 1 的个数 + 十位出现 1 的个数 = 2 + 4 = 6;
f(23) = 个位出现 1 的个数 + 十位出现 1 的个数 = 3 + 10 = 13;
f(33) = 个位出现 1 的个数 + 十位出现 1 的个数 = 4 + 10 = 14;
…
f(93) = 个位出现 1 的个数 + 十位出现 1 的个数 = 10 + 10 = 20;
```

接着分析 3 位数。

如果 $N = 123$：

个位出现 1 的个数为 13：1, 11, 21, …, 91, 101, 111, 121

十位出现 1 的个数为 20：10～19, 110～119

百位出现 1 的个数为 24：100～123

f（23）= 个位出现 1 的个数 + 十位出现 1 的个数 + 百位出现 1 的次数 = 13 + 20 + 24 = 57；

同理我们可以再分析 4 位数、5 位数。读者朋友们可以写一写，总结一下各种情况有什么不同。

根据上面的一些尝试，下面我们推导出一般情况下，从 *N* 得到 *f*（*N*）的计算方法。

假设 *N*=*abcde*，这里 *a*、*b*、*c*、*d*、*e* 分别是十进制数 *N* 的各个数位上的数字。如果要计算百位上出现 1 的次数，它将会受到三个因素的影响：百位上的数字，百位以下（低位）的数字，百位（更高位）以上的数字。

如果百位上的数字为 0，则可以知道，百位上可能出现 1 的次数由更高位决定，比如 12 013，则可以知道百位出现 1 的情况可能是 100～199，1 100～1 199，2 100～2 199，…，11 100~11 199，一共有 1 200 个。也就是由更高位数字（12）决定，并且等于更高位数字（12）×当前位数（100）。

如果百位上的数字为 1，则可以知道，百位上可能出现 1 的次数不仅受更高位影响，还受低位影响，也就是由更高位和低位共同决定。例如对于 12 113，受更高位影响，百位出现 1 的情况是 100～199，1 100～1 199，2 100～2 199，…，11 100~11 199，一共 1 200 个，和上面第一种情况一样，等于更高位数字（12）×当前位数（100）。但是它还受低位影响，百位出现 1 的情况是 12 100～12 113，一共 114 个，等于低位数字（113）+1。

如果百位上数字大于 1（即为 2~9），则百位上可能出现 1 的次数也仅由更高位决定，比如 12 213，则百位出现 1 的可能性为：100～199，1 100～1 199，2 100～2 199，…，11 100～11 199，12 100～12 199，一共有 1 300 个，并且等于更高位数字+1（12+1）×当前位数（100）。

通过上面的归纳和总结，我们可以写出如代码清单 2-10 所示的更高效算法来计算 *f*（*N*）：

代码清单 2-10

```
LONGLONG Sum1s(ULONGLONG n)
{
```

```
    ULONGLONG iCount = 0;

    ULONGLONG iFactor = 1;

    ULONGLONG iLowerNum = 0;
    ULONGLONG iCurrNum = 0;
    ULONGLONG iHigherNum = 0;

    while(n / iFactor != 0)
    {
        iLowerNum = n - (n / iFactor) * iFactor;
        iCurrNum = (n / iFactor) % 10;
        iHigherNum = n / (iFactor * 10);

        switch(iCurrNum)
        {
        case 0:
            iCount += iHigherNum * iFactor;
            break;
        case 1:
            iCount += iHigherNum * iFactor + iLowerNum + 1;
            break;
        default:
            iCount += (iHigherNum + 1) * iFactor;
            break;
        }

        iFactor *= 10;
    }

    return iCount;
}
```

这个方法只要分析 *N* 就可以得到 *f*（*N*），避开了从 1 到 *N* 的遍历，输入长度为 *Len* 的数字 *N* 的时间复杂度为 *O*（*Len*），即为 *O*（ln（*N*）/ln（10）+1）。在笔者的计算机上，计算 *N*=100 000 000，相对于第一种方法的 40 秒时间，这种算法不到 1 毫秒就可以返回结果，速度至少提高了 40 000 倍。

问题 2 的解法

要确定最大的数 *N*，满足 *f*（*N*）=*N*。我们通过简单的分析可以知道（仿照上面给出的方法来分析）：

9 以下为：1 个；
99 以下为：1×10+10×1=20 个；
999 以下为：1×100+10×20=300 个；
9 999 以下为：1×1 000+10×300=4 000 个；
…
999 999 999 以下为：900 000 000 个；

9 999 999 999 以下为： 10 000 000 000 个。

容易从上面的式子归纳出：$f(10n-1)=n*10n-1$。通过这个递推式，很容易看到，当 $n = 10^{10}-1$ 时，$f(n)$ 的值大于 n，所以我们可以猜想，当 n 大于某一个数 N 时，$f(n)$ 会始终比 n 大，也就是说，最大满足条件在 0～N 之间，亦即 N 是最大满足条件 $f(n) = n$ 的一个上界。如果能估计出这个 N，那么只要让 n 从 N 往 0 递减，每个分别检查是否有 $f(n) = n$，第一个满足条件的数就是我们要求的整数。

因此，问题转化为如何证明上界 N 确实存在，并估计出这个上界 N。

证明满足条件 $f(n) = n$ 的数存在一个上界

用类似数学归纳法的思路来推理这个问题。很容易得到下面这些结论（读者朋友可以自己试着列举验证一下）：

当 n 增加 10 时，$f(n)$ 至少增加 1；

当 n 增加 100 时，$f(n)$ 至少增加 20；

当 n 增加 1 000 时，$f(n)$ 至少增加 300；

当 n 增加 10 000 时，$f(n)$ 至少增加 4 000；

……

当 n 增加 10^k 时，$f(n)$ 至少增加 $k*10^{k-1}$。

把 n 按十进制展开，$n=a*10^k+b*10^{k-1}+\cdots$，则由上可得，

$f(n)=f(0+a*10^k+b*10^{k-1}+\cdots\cdots)>a*k*10^{k-1}+b*(k-1)*10^{k-2}$

这里把 $a*10^k$ 看作在初值 0 上作 a 次的 10^k 的增量，$b*10^{k-1}$ 为再作 b 次的 10^{k-1} 的增量，重复使用上面关于自变量的 10^k 增加量的归纳结果。

又，

$n=a*10^k+b*10^{k-1}+\cdots<a*10^k+(b+1)*10^{k-1}$

如果 $a*k*10^{k-1}+b*(k-1)*10^{k-2}>=a*10^k+(b+1)*10^{k-1}$ 的话，那么 $f(n)$ 必然大于 n。而要使不等式 $a*k*10^{k-1}+b*(k-1)*10^{k-2}>=a*10^k+(b+1)*10^{k-1}$ 成立，k 需要满足条件：$k>=10+(b+10)/(b+10*a)$。显然，当 $k > 11$，或者说 n 的整数位数大于等于 12 时，$f(n)>n$ 恒成立。因此，我们求得一个满足条件的上界 $N=10^{11}$。

计算这个最大数 n

令 $N = 10^{11}-1=99\,999\,999\,999$，让 n 从 N 往 0 递减，每个分别检查是否有 $f(n) = n$，第一满足条件的就是我们要求的整数。很容易解出 $n = 1\,111\,111\,110$ 是满足 $f(n) = n$ 的最大整数。

扩展问题

对于其他进制表达方式，也可以试一试，看看有什么规律。例如二进制：

$f(1) = 1$

$f(10) = 10$（因为 01, 10 有两个 1）

$f(11) = 100$（因为 01, 10, 11 有四个 1）

读者朋友可以模仿我们的分析方法，给出相应的解答。

2.5 ★★★ 寻找最大的 K 个数

在面试中，有下面的问答。

问：有很多个无序的数，我们姑且假定它们各不相等，怎么选出其中最大的若干个数呢？

答：可以这样写：int array[100] ……

问：好，如果有更多的元素呢？

答：那可以改为：int array[1000] ……

问：如果我们有很多元素，例如 1 亿个浮点数，怎么办？

答：个，十，百，千，万……那可以写：float array [100 000 000] ……

问：这样的程序能编译运行么？

答：嗯……我从来没写过这么多的 0 ……

分析与解法

解法一

当学生们信笔写下 float array [10000000]，他们往往没有想到这个数据结构要如何在电脑上实现，是从当前程序的栈（Stack）中分配，还是堆（Heap），还是电脑的内存也许放不下这么大的东西？

我们先假设元素的数量不大，例如在几千个左右，在这种情况下，那我们就排序一下吧。在这里，快速排序或堆排序都是不错的选择，他们的平均时间复杂度都是 $O(N * \log_2 N)$。然后取出前 K 个，$O(K)$。总时间复杂度 $O(N * \log_2 N) + O(K) = O(N * \log_2 N)$。

你一定注意到了，当 K=1 时，上面的算法也是 $O(N * \log_2 N)$ 的复杂度，而显然我们可以通过 N–1 次的比较和交换得到结果。上面的算法把整个数组都进行了排序，而原题目只要求最大的 K 个数，并不需要前 K 个数有序，也不需要后 N–K 个数有序。

怎么能够避免做后 N–K 个数的排序呢？我们需要部分排序的算法，选择排序和交换排序都是不错的选择。把 N 个数中的前 K 个数排序出来，复杂度是 $O(N * K)$。

哪一个更好呢？$O(N * \log_2 N)$ 还是 $O(N * K)$？这取决于 K 的大小，这是你需要在面试者那里弄清楚的问题。在 K（$K <= \log_2 N$）较小的情况下，可以选择部分排序。

在下一个解法中，我们会通过避免对前 K 个数排序来得到更好的性能。

解法二

回忆一下快速排序，快排中的每一步，都是将待排数据分做两组，其中一组数据的任何一个数都比另一组中的任何一个大，然后再对两组分别做类似的操作，然后继续下去……

在本问题中，假设 N 个数存储在数组 S 中，我们从数组 S 中随机找出一个元素 X，把数组分为两部分 S_a 和 S_b。S_a 中的元素大于等于 X，S_b 中的元素小于 X。

这时，有两种可能性：

1. S_a 中元素的个数小于 K，S_a 中所有的数和 S_b 中最大的 $K-|S_a|$ 个元素（$|S_a|$ 指 S_a 中元素的个数）就是数组 S 中最大的 K 个数；
2. S_a 中元素的个数大于或等于 K，则需要返回 S_a 中最大的 K 个元素。

这样递归下去，不断把问题分解成更小的问题，平均时间复杂度 $O(N * \log_2 K)$。伪代码如清单 2-11 所示：

代码清单 2-11

```
Kbig(S, k):
    if(k <= 0):
        return []       // 返回空数组
    if(length S <= k):
        return S
    (Sa, Sb) = Partition(S)
    return Kbig(Sa, k).Append(Kbig(Sb, k - length Sa)

Partition(S):
    Sa = []             // 初始化为空数组
    Sb = []             // 初始化为空数组
    Swap(s[1], S[Random()%length S])   // 随机选择一个数作为分组标准，以
                                       // 避免特殊数据下的算法退化，也可
                                       // 以通过对整个数据进行洗牌预处理
                                       // 实现这个目的
    p = S[1]
    for i in [2: length S]:
        S[i] > p ? Sa.Append(S[i]) : Sb.Append(S[i])
                        // 将 p 加入较小的组，可以避免分组失败，也使分组
                        // 更均匀，提高效率
length Sa < length Sb ? Sa.Append(p) : Sb.Append(p)
return (Sa, Sb)
```

解法三

寻找 N 个数中最大的 K 个数，本质上就是寻找最大的 K 个数中最小的那个，也就是第 K 大的数。可以使用二分搜索的策略来寻找 N 个数中的第 K 大的数。对于一个给定的数 p，可以在 $O(N)$ 的时间复杂度内找出所有不小于 p 的数。假如 N 个数中最大的数为 V_{max}，最小的数为 V_{min}，那么这 N 个数中的第 K 大数一定在区间 $[V_{min}, V_{max}]$之间。那么，可以在这个区间内二分搜索 N 个数中的第 K 大数 p。伪代码如清单 2-12 所示：

代码清单 2-12

```
while(Vmax - Vmin > delta)
{
    Vmid = Vmin + (Vmax - Vmin) * 0.5;
    if(f(arr, N, Vmid) >= K)
        Vmin = Vmid;
```

```
    else
        Vmax = Vmid;
}
```

伪代码中 f（arr, N, V_{mid}）返回数组 $arr[0,\cdots, N-1]$中大于等于 V_{mid} 的数的个数。

上述伪代码中，delta 的取值要比所有 N 个数中的任意两个不相等的元素差值之最小值小。如果所有元素都是整数，delta 可以取值 0.5。循环运行之后，得到一个区间（V_{min}, V_{max}），这个区间仅包含一个元素（或者多个相等的元素）。这个元素就是第 K 大的元素。整个算法的时间复杂度为 O（$N * \log_2$（$|V_{max} - V_{min}|$ /delta））。由于 delta 的取值要比所有 N 个数中的任意两个不相等的元素差值之最小值小，因此时间复杂度跟数据分布相关。在数据分布平均的情况下，时间复杂度为 $O(N * \log_2(N))$。

在整数的情况下，可以从另一个角度来看这个算法。假设所有整数的大小都在$[0, 2^{m-1}]$之间，也就是说，所有整数在二进制中都可以用 m bit 来表示（从低位到高位，分别用 0, 1, …, $m-1$ 标记）。我们可以先考察在二进制位的第（$m-1$）位，将 N 个整数按该位为 1 或 0 分成两个部分。也就是将整数分成取值为$[0, 2^{m-1}-1]$和$[2^{m-1}, 2^m-1]$两个区间。前一个区间中的整数第$(m-1)$位为 0，后一个区间中的整数第$(m-1)$位为 1。如果该位为 1 的整数个数 A 大于等于 K，那么，在所有该位为 1 的整数中继续寻找最大的 K 个。否则，在该位为 0 的整数中寻找最大的 $K-A$ 个。接着考虑二进制位第（$m-2$）位，依次类推。思路跟上面的浮点数的情况本质上一样。

对于上面两个方法，我们都需要遍历一遍整个集合，统计在该集合中大于等于某一个数的整数有多少个。不须要做随机访问操作，如果全部数据不能载入内存，可以每次都遍历一遍文件。经过统计，更新解所在的区间之后，再遍历一次文件，把在新的区间中的元素存入新的文件。下一次操作的时候，不再须要遍历全部的元素。每次须要两次文件遍历，最坏情况下，总共需要遍历文件的次数为 $2 * \log_2(|V_{max} - V_{min}|/\text{delta})$。由于每次更新解所在区间之后，元素数目会减少。在所有元素能够全部载入内存之后，就可以不再通过读写文件的方式来操作了。

此外，寻找 N 个数中的第 K 大的数，是一个经典问题。理论上，这个问题存在线性算法。不过这个线性算法的常数项比较大，在实际应用中效果有时并不好。

解法四

我们已经得到了三个解法，不过这三个解法有个共同的地方，就是需要对数据访问多次，那么就有下一个问题，如果 N 很大呢，100 亿？（更多的情况下，是面

试者问你这个问题。）这个时候数据不能全部装入内存（不过也很难说，谁知道以后会不会 1T 内存比 1 斤白菜还便宜），所以要求尽可能少地遍历所有数据。

不妨设 $N > K$，前 K 个数中的最大 K 个数是一个退化的情况，所有 K 个数就是最大的 K 个数。如果考虑第 $K+1$ 个数 X 呢？如果 X 比最大的 K 个数中的最小的数 Y 小，那么最大的 K 个数还是保持不变。如果 X 比 Y 大，那么最大的 K 个数应该去掉 Y，而包含 X。如果用一个数组来存储最大的 K 个数，每新加入一个数 X，就扫描一遍数组，得到数组中最小的数 Y。用 X 替代 Y，或者保持原数组不变。这样的方法，所耗费的时间为 O（$N * K$）。

进一步，可以用容量为 K 的最小堆来存储最大的 K 个数。最小堆的堆顶元素就是最大 K 个数中最小的一个。每次新考虑一个数 X，如果 X 比堆顶的元素 Y 小，则不需要改变原来的堆，因为这个元素比最大的 K 个数小。如果 X 比堆顶元素大，那么用 X 替换堆顶的元素 Y。在 X 替换堆顶元素 Y 之后，X 可能破坏最小堆的结构（每个结点都比它的父亲结点大），需要更新堆来维持堆的性质。更新过程花费的时间复杂度为 O（$\log_2 K$）。

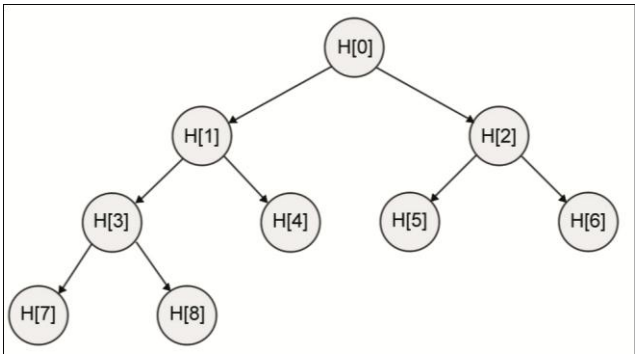

图 2-1

图 2-1 是一个堆，用一个数组 h[]表示。每个元素 $h[i]$，它的父亲结点是 $h[i/2]$，儿子结点是 $h[2 * i + 1]$和 $h[2 * i + 2]$。重新考虑一个数 X，需要进行的更新操作伪代码如清单 2-13 所示：

代码清单 2-13

```
if(X > h[0])
{
    h[0] = X;
    p = 0;
    while(p < K)
    {
        q = 2 * p + 1;
        if(q >= K)
            break;
        if((q < K - 1) && (h[q + 1] < h[q]))
```

```
            q = q + 1;
        if(h[q] < h[p])
        {
            t = h[p];
            h[p] = h[q];
            h[q] = t;
            p = q;
        }
        else
            break;
    }
}
```

因此，算法只需要扫描所有的数据一次，时间复杂度为 O（N * $\log_2 K$）。这实际上是部分执行了堆排序的算法。在空间方面，由于这个算法只扫描所有的数据一次，因此我们只须要存储一个容量为 K 的堆。大多数情况下，堆可以全部载入内存。如果 K 仍然很大，我们可以尝试先找最大的 K' 个元素，然后找第 K' +1 个到第 2 * K' 个元素，依次类推（其中容量 K' 的堆可以完全载入内存）。不过这样，我们须要扫描所有数据 ceil[2]（K/K' ）次。

解法五

上面类快速排序的方法平均时间复杂度是线性的。能否有确定的线性算法呢？是否可以通过改进计数排序、基数排序等来得到一个更高效的算法呢？答案是肯定的。但算法的适用范围会受到一定的限制。

如果所有 N 个数都是正整数，且它们的取值范围不太大，可以考虑申请空间，记录每个整数出现的次数，然后再从大到小取最大的 K 个。比如，所有整数都在（0, MAXN）区间中的话，利用一个数组 count[MAXN]来记录每个整数出现的个数（count[i]表示整数 i 在所有整数中出现的个数）。我们只须要扫描一遍就可以得到 count 数组。然后，寻找第 K 大的元素，如代码清单 2-14 所示：

代码清单 2-14

```
for(sumCount = 0, v = MAXN - 1; v >= 0; v--)
{
    sumCount += count[v];
    if(sumCount >= K)
        break;
}
return v;
```

[2] ceil（ceiling，天花板之意）表示大于等于一个浮点数的最小整数。

极端情况下，如果 N 个整数各不相同，我们甚至只需要一个 bit 来存储这个整数是否存在。

实际情况下，并不一定能保证所有元素都是正整数，且取值范围不太大。上面的方法仍然可以推广适用。如果 N 个数中最大的数为 V_{max}，最小的数为 V_{min}，我们可以把这个区间$[V_{min}, V_{max}]$分成 M 块，每个小区间的跨度为 d =（$V_{max} - V_{min}$）/M，即 $[V_{min}, V_{min}+d]$, $[V_{min} + d, V_{min} + 2d]$……然后，扫描一遍所有元素，统计各个小区间中的元素个数，跟上面方法类似地，我们可以知道第 K 大的元素在哪一个小区间。然后，再对那个小区间继续进行分块处理。这个方法介于解法三和类计数排序方法之间，不能保证线性。跟解法三类似地，时间复杂度为 O（(N+M）* $\log_2 M$（|V_{max} − V_{min}|/delta））。遍历文件的次数为 2 * $\log_2 M$（|V_{max} − V_{min}|/delta）。当然，我们须要找一个尽量大的 M，但 M 取值要受内存限制。

在这道题中，我们根据 K 和 N 的相对大小，设计了不同的算法。在实际面试中，如果一个面试者能针对一个问题，说出多种不同的方法，并且分析它们各自适用的情况，那一定会给人留下深刻印象。

注：本题目的解答中用到了多种排序算法，这些算法在大部分的算法书籍中都有讲解。掌握排序算法对工作也会很有帮助。

扩展问题

1. 如果须要找出 N 个数中最大的 K 个不同的浮点数呢？比如，含有 10 个浮点数的数组（1.5, 1.5, 2.5, 2.5, 3.5, 3.5, 5, 0, −1.5, 3.5）中最大的 3 个不同的浮点数是（5, 3.5, 2.5）。
2. 如果是找第 k 到 m（0 < k < = m < = n）大的数呢？
3. 在搜索引擎中，网络上的每个网页都有“权威性”权重，如 page rank。如果我们须要寻找权重最大的 K 个网页，而网页的权重会不断地更新，那么算法要如何变动以达到快速更新（incremental update）并及时返回权重最大的 K 个网页？

   提示：堆排序？当每一个网页权重更新的时候，更新堆。还有更好的方法吗？

4. 在实际应用中，还有一个“精确度”的问题。我们可能并不须要返回严格意义上的最大的 K 个元素，在边界位置允许出现一些误差。当用户输入一个 query 的时候，对于每一个文档 d 来说，它跟这个 query 之间都有一个相关性

衡量权重 f（query, d）。搜索引擎须要返回给用户的就是相关性权重最大的 K 个网页。如果每页 10 个网页，用户不会关心第 1000 页开外搜索结果的“精确度”，稍有误差是可以接受的。比如我们可以返回相关性第 10 001 大的网页，而不是第 9999 大的。在这种情况下，算法该如何改进才能更快更有效率呢？网页的数目可能大到一台机器无法容纳得下，这时怎么办呢？

提示：归并排序？如果每台机器都返回最相关的 K 个文档，那么所有机器上最相关 K 个文档的并集肯定包含全集中最相关的 K 个文档。由于边界情况并不需要非常精确，如果每台机器返回最好的 K' 个文档，那么 K' 应该如何取值，以达到我们返回最相关的 90%*K 个文档是完全精确的，或者最终返回的最相关的 K 个文档精确度超过 90%（最相关的 K 个文档中 90%以上在全集中相关性的确排在前 K），或者最终返回的最相关的 K 个文档最差的相关性排序没有超出 110%*K。

5. 如第 4 点所说，对于每个文档 d，相对于不同的关键字 $q_1, q_2, \cdots, q_m$，分别有相关性权重 $f(d, q_1)$，$f(d, q_2)$，…，$f(d, q_m)$。如果用户输入关键字 q_i 之后，我们已经获得了最相关的 K 个文档，而已知关键字 q_j 跟关键字 q_i 相似，文档跟这两个关键字的权重大小比较靠近，那么关键字 q_i 的最相关的 K 个文档，对寻找 q_j 最相关的 K 个文档有没有帮助呢？

2.6 ★ 精确表达浮点数

在计算机中，使用 float 或 double 来存储小数是不能得到精确值的。如果你希望得到精确计算结果，最好是用分数形式来表示小数。有限小数或者无限循环小数都可以转化为分数。比如：

0.9 = 9/10

0.333（3）= 1/3（括号中的数字表示是循环节）

当然一个小数可以用好几种分数形式来表示。如：

0.333（3）= 1/3 = 3/9

给定一个有限小数或无限循环小数，你能否以分母最小的分数形式来返回这个小数呢？如果输入为循环小数，循环节用括号标记出来。下面是一些可能的输入数据，如 0.3、0.30、0.3（000）、0.3333（3333）、……

分析与解法

拿到这样一个问题，我们往往会从最简单的情况入手，因为所有的小数都可以分解成一个整数和一个纯小数之和，不妨只考虑大于 0、小于 1 的纯小数，且暂时不考虑分子和分母的约分，先设法将其表示为分数形式，然后再进行约分。题目中输入的小数，要么为有限小数 $X=0.a_1a_2\cdots a_n$，要么为无限循环小数 $X=0.a_1a_2\cdots a_n(b_1b_2\cdots b_m)$，$X$ 表示式中的字母 $a_1a_2\cdots a_n$，$b_1b_2\cdots b_m$ 都是 0~9 的数字，括号部分 $(b_1b_2\cdots b_m)$ 表示循环节，我们需要处理的就是以上两种情况。

对于有限小数 $X=0.a_1a_2\cdots a_n$ 来说，这个问题比较简单，X 就等于 $(a_1a_2\cdots a_n)/10^n$。

对于无限循环小数 $X=0.a_1a_2\cdots a_n(b_1b_2\cdots b_m)$ 来说，其复杂部分在于小数点后同时有非循环部分和循环部分，我们可以做如下的转换：

$X = 0.a_1a_2\cdots a_n(b_1b_2\cdots b_m)$

$\Rightarrow 10^n * X = a_1a_2\cdots a_n.(b_1b_2\cdots b_m)$

$\Rightarrow 10^n * X = a_1a_2\cdots a_n+0.(b_1b_2\cdots b_m)$

$\Rightarrow X = (a_1a_2\cdots a_n+0.(b_1b_2\cdots b_m))/10^n$

对于整数部分 $a_1a_2\cdots a_n$，不需要做额外处理，只需要把小数部分转化为分数形式再加上这个整数即可。对于后面的无限循环部分，可以采用如下方式进行处理：

令 $Y=0.(b_1b_2\cdots b_m)$，那么

$10^m * Y = b_1b_2\cdots b_m.(b_1b_2\cdots b_m)$

$\Rightarrow 10^m * Y = b_1b_2\cdots b_m+0.(b_1b_2\cdots b_m)$

$\Rightarrow 10^m * Y - Y = b_1b_2\cdots b_m$

$\Rightarrow Y = b_1b_2\cdots b_m/(10^m-1)$

将 Y 代入前面的 X 的等式可得：

$X = (a_1a_2\cdots a_n+Y)/10^n$

$= (a_1a_2\cdots a_n + b_1b_2\cdots b_m/(10^m-1))/10^n$

$= ((a_1a_2\cdots a_n) * (10^m-1) + (b_1b_2\cdots b_m))/((10^m-1) * 10^n)$

至此，便可以得到任意一个有限小数或无限循环小数的分数表示，但是此时分母未必是最简的，接下来的任务就是让分母最小，即对分子和分母进行约分，这个

相对比较简单。对于任意一个分数 *A*/*B*，可以简化为（*A* /Gcd（*A*, *B*））/（*B* /Gcd（*A*, *B*）），其中 Gcd 函数为求 *A* 和 *B* 的最大公约数，这就涉及本书中的算法（2.7 节“最大公约数问题”），其中有很巧妙的解法，请读者阅读具体的章节，这里就不再赘述。

综上所述，先求得小数的分数表示方式，再对其分子分母进行约分，便能够得到分母最小的分数表现形式。

例如，对于小数 0.3（33），根据上述方法，可以转化为分数：

0.3（33）

=（3 *（10^2−1）+ 33）/（（10^2−1）*10）

=（3*99+33）/990

= 1 / 3

对于小数 0. 285 714（285 714），我们也可以算出：

0. 285 714（285 714）

= （285 714 *（10^6−1） + 285 714） / （（10^6−1）*10^6）

= （285 714*999 999 +285 714） / 999 999 000 000

= 285 714 / 999 999

= 2/7

2.7 ★★ 最大公约数问题

写一个程序，求两个正整数的最大公约数。如果两个正整数都很大，有什么简单的算法吗？

分析与解法

求最大公约数是一个很基本的问题。早在公元前 300 年左右，欧几里得就在他的著作《几何原本》中给出了高效的解法——辗转相除法。辗转相除法使用到的原理很聪明也很简单，假设用 $f(x, y)$ 表示 x，y 的最大公约数，取 $k = x/y$，$b = x\%y$，则 $x = ky + b$，如果一个数能够同时整除 x 和 y，则必能同时整除 b 和 y；而能够同时整除 b 和 y 的数也必能同时整除 x 和 y，即 x 和 y 的公约数与 b 和 y 的公约数是相同的，其最大公约数也是相同的，则有 $f(x, y) = f(y, y \% x)$（$y > 0$），如此便可把原问题转化为求两个更小数的最大公约数，直到其中一个数为 0，剩下的另外一个数就是两者最大的公约数。辗转相除法更详细的证明可以在很多的初等数论相关书籍中找到，或者读者也可以试着证明一下。

示例如下：

$f(42, 30) = f(30, 12) = f(12, 6) = f(6, 0) = 6$

解法一

最简单的实现，就是直接用代码来实现辗转相除法。从上面的描述中，我们知道，利用递归就能够很轻松地把这个问题完成。

具体代码如下：

```
int gcd(int x, int y)
{
    return (!y)?x:gcd(y, x%y);
}
```

解法二

在解法一中，我们用到了取模运算。但对于大整数而言，取模运算（其中用到除法）是非常昂贵的开销，将成为整个算法的瓶颈。有没有办法能够不用取模运算呢？

采用类似前面辗转相除法的分析，如果一个数能够同时整除 x 和 y，则必能同时整除 $x–y$ 和 y；而能够同时整除 $x–y$ 和 y 的数也必能同时整除 x 和 y，即 x 和 y 的公约数与 $x–y$ 和 y 的公约数是相同的，其最大公约数也是相同的，即 $f(x, y) = f(x–y, y)$，那么就可以不再需要进行大整数的取模运算，而转换成简单得多的大整数的减法。

在实际操作中，如果 $x<y$，可以先交换（x, y）（因为（x, y）=（y, x）），从而避免求一个正数和一个负数的最大公约数情况的出现。一直迭代下去，直到其中一个数为 0。

示例如下：

f（42, 30）=f（30, 12）=f（12, 18）=f（18, 12）=f（12, 6）=f（6, 6）=f（6, 0）= 6

解法二的具体代码如清单 2-15 所示。

代码清单 2-15

```
BigInt gcd(BigInt x, BigInt y)
{
    if(x < y)
        return gcd(y, x);
    if(y == 0)
        return x;
    else
        return gcd(x - y, y);
}
```

代码中 BigInt 是读者自己实现的一个大整数类（所谓大整数当然可以是成百上千位），那么就要求读者重载该大整数类中的减法运算符“-”，关于大整数的具体实现这里不再赘述，若读者只是想验证该算法的正确性，完全可使用系统内建的 int 型来测试。

这个算法，免去了大整数除法的繁琐，但是同样也有不足之处。最大的瓶颈就是迭代的次数比之前的算法多了不少，如果遇到（10 000 000 000 000, 1）这类情况，就会相当地令人郁闷了。

解法三

解法一的问题在于计算复杂的大整数除法运算，而解法二虽然将大整数的除法运算转换成了减法运算，降低了计算的复杂度，但它的问题在于减法的迭代次数太多，那么能否结合解法一和解法二从而使其成为一个最佳的算法呢？答案是肯定的。

从分析公约数的特点入手。

对于 y 和 x 来说，如果 $y=k * y_1$，$x=k * x_1$。那么有 f（y, x）= $k * f$（y_1, x_1）。

另外，如果 $x = p * x_1$，假设 p 是素数，并且 $y \% p != 0$（即 y 不能被 p 整除），那么 f（x, y）= f（$p * x_1, y$）= f（x_1, y）。

注意到以上两点之后，我们就可以利用这两点对算法进行改进。

最简单的方法是，我们知道，2 是一个素数，同时对于二进制表示的大整数而言，可以很容易地将除以 2 和乘以 2 的运算转换成移位运算，从而避免大整数除法，由此就可以利用 2 这个数字来进行分析。

取 $p = 2$

若 x, y 均为偶数，$f(x, y) = 2 * f(x/2, y/2) = 2 * f(x>>1, y>>1)$

若 x 为偶数，y 为奇数，$f(x, y) = f(x/2, y) = f(x>>1, y)$

若 x 为奇数，y 为偶数，$f(x, y) = f(x, y/2) = f(x, y>>1)$

若 x, y 均为奇数，$f(x, y) = f(y, x - y)$，

那么在 $f(x, y) = f(y, x - y)$ 之后，$(x - y)$ 是一个偶数，下一步一定会有除以 2 的操作。

因此，最坏情况下的时间复杂度是 $O(\log_2(\max(x, y)))$。

考虑如下的情况：

$$
\begin{aligned}
f(42, 30) &= f(101010_2, 11110_2) \\
&= 2 * f(10101_2, 1111_2) \\
&= 2 * f(1111_2, 110_2) \\
&= 2 * f(1111_2, 11_2) \\
&= 2 * f(1100_2, 11_2) \\
&= 2 * f(11_2, 11_2) \\
&= 2 * f(0_2, 11_2) \\
&= 2 * 11_2 \\
&= 6
\end{aligned}
$$

根据上面的规律，具体代码实现如清单 2-16 所示。

代码清单 2-16

```
BigInt gcd(BigInt x, BigInt y)
{
    if(x < y)
        return gcd(y, x);
    if(y == 0)
        return x;
    else
    {
        if(IsEven(x))
```

```
        {
            if(IsEven(y))
                return (gcd(x >> 1, y >> 1) << 1);
            else
                return gcd(x >> 1, y);
        }
        else
        {
            if(IsEven(y))
                return gcd(x, y >> 1);
            else
                return gcd(y, x - y);
        }
    }
}
```

BigInt 见解法二中的解释，IsEven（BigInt *x*）函数检查 *x* 是否为偶数，如果 *x* 为偶数，则返回 true，否则返回 false。

解法三很巧妙地利用移位运算和减法运算，避开了大整数除法，提高了算法的效率。程序员常常将移位运算作为一种技巧来使用，最常见的就是通过左移或右移来实现乘以 2 或除以 2 的操作。其实移位的用处远不止于此，如求一个整数的二进制表示中 1 的个数问题（见本书 2.1 节“求二进制数中 1 的个数”）和逆转一个整数的二进制表示问题等，往往让人拍案叫绝。

2.8 ★★ 找符合条件的整数

任意给定一个正整数 N，求一个最小的正整数 M（$M>1$），使得 $N*M$ 的十进制表示形式里只含有 1 和 0。

最初的解法

看了题目要求之后，我们的第一想法就是从小到大枚举 M 的取值，然后再计算 $N * M$，最后判断它们的乘积是否只含有 1 和 0。大体的思路可以用下面的伪代码来实现：

```
for(M = 2; ; M++)
{
    product = N * M;
    if(HasOnlyOneAndZero(product))
        output N, M, Product, and return;
}
```

但是问题很快就出现了，什么时候应该终止循环呢？这个循环会终止吗？即使能终止，也许这个循环仍须要耗费太多的时间，比如 $N = 99$ 时，$M = 1\ 122\ 334\ 455\ 667\ 789$，$N * M = 111\ 111\ 111\ 111\ 111\ 111$。

分析与解法

题目中的直接做法显然不是一个令人满意的方法。还有没有其他的方法呢？答案是肯定的。

可以做一个问题的转化。由于问题中要求 $N * M$ 的十进制表示形式里只含有 1 和 0，所以 $N * M$ 与 M 相比有明显的特征。我们不妨尝试去搜索它们的乘积 $N * M$，这样在某些情况下需要搜索的空间要小很多。另外，搜索 $N * M$，而不去搜索 M，其实有一个更加重要的原因，就是当 M 很大时，特别是当 M 大于 2^{32} 时，某些机器就可能没法表示 M 了，我们就得自己实现高精度大整数类。但是考虑 $N * M$ 的特点，可以只需要存储 $N * M$ 的十进制表示中“1”的位置，这样就可以大大缩小为表示 $N * M$ 所需要的空间，从而使程序能处理数值很大的情况。因此，考虑到程序的推广性，选择了以 $N * M$ 为目标进行计算。

换句话说，就是把问题从“求一个最小的正整数 M，使得 $N * M$ 的十进制表示形式里只含有 1 和 0”变成求一个最小的正整数 X，使得 X 的十进制表示形式里只含有 1 和 0，并且 X 被 N 整除。

我们先来看一下 X 的取值，X 从小到大有如下的取值：1、10、11、100、101、110、111、1000、1001、1010、1011、1100、1101、1110、1111、10000、……

如果直接对 X 进行循环，就是先检查 X=1 是否可以整除 N，再检查 X=10，然后检查 X=11，接着检查 X=100…（就像遍历二进制整数一样遍历 X 的各个取值）。但是这样处理还是比较慢，如果 X 的最终结果有 K 位，则要循环搜索 2^K 次。由于我们的目标是寻找最小的 X，使得 $X \bmod N = 0$，我们只要记录 $\bmod N = i (0 <= i < N)$ 的最小 X 就可以了。这样通过避免一些不必要的循环，可以达到加速算法的目的。那么如何避免不必要的循环呢？先来看一个例子。

设 N=3，X=1，再引入一个变量 j，$j=X \% N$。直接遍历 X，计算中间结果如表 2-1 所示。

表 2-1

Num	1	2	3	4	5	6	7
N	3	3	3	3	3	3	3
X	1	10	11	100	101	110	111
J	1	1	2	1	2	2	0

表 2-1 计算 110 % 3 是多余的。原因是 1 和 10 对 3 的余数相同，所以 101 和 110 对 3 的余数相同，那么只需要判断 101 是否可以整除 3 就可以了，而不用判断 110 是否能整除 3。并且，如果 X 的最低 3 位是 110，那么可以通过将 101 替换 110 得到一个符合条件的更小正整数。因此，对于 mod N 同余的数，只需要记录最小的一个。

有些读者可能会问，当 X 循环到 110 时，我怎么知道 1 和 10 对 3 的余数相同呢？其实，X=1，X=10 是否能整除 3，在 X 循环到 110 时都已经计算过了，只要在计算 X=1，X=10 时，保留 X % N 的结果，就可以在 X=110 时作出判断，从而避免计算 X % N。

以上的例子阐明了在计算中保留 X 除以 N 的余数信息可以避免不必要的计算。下面给出更加形式化的论述。

假设已经遍历了 X 的十进制表示有 K 位时的所有情况，而且也搜索了 $X=10^K$ 的情况，设 $10^K \% N = a$。现在要搜索 X 有 K+1 位的情况，即 $X=10^K+Y$，$(0<Y<10^K)$。如果用最简单的方法，搜索空间（Y 的取值）将有 2^K-1 个数据。但是如果对这个空间进行分解，即把 Y 按照其对 N 的余数分类，我们的搜索空间将被分成 $N-1$ 个子空间。对于每个子空间，其实只须要判断其中最小的元素加上 10^K 是否能被 N 整除即可，而没有必要判断这个子空间里所有元素加上 10^K 是否能被 N 整除。这样搜索的空间就从 2^K-1 维压缩到了 $N-1$ 维。但是这种压缩有一个前提，就是在前面的计算中已经保留了余数信息，并且把 Y 的搜索空间进行了分解。所谓分解，其实，从

技术上讲，就是对于“X 模 N”的各种可能结果，保留一个对应的已经出现了的最小的 X（即建立一个长度为 N 的“余数信息数组”，这个数组的第 i 位保留已经出现的最小的模 N 为 i 的 X）。

那么现在的问题就是如何维护这个“余数信息数组”了。假设已经有了 X 的十进制表示有 K 位时的所有余数信息。也有了 $X=10^K$ 的余数信息。现在我们要搜索 X 有 $K+1$ 位的情况，也即 $X=10^K+Y$，（$0<Y<10^K$）时，X 除以 N 的余数情况。由于已经有了对 Y 的按除 N 的余数进行的空间分解情况，即 $Y<10^K$ 的余数信息数组。我们只须要将 $10^K\%N$ 的结果与余数信息数组里非空的元素相加，再去模 N，看看会不会出现新的余数即可。如果出现，就在余数信息数组的相应位置增添对应的 X。这一步只须要 N 次循环。

综上所述，假设最终的结果 X 有 K 位，那么直接遍历 X，须要循环 2^K 次，而按照我们保留余数信息避免不必要的循环的方法，最多只需要（K–1）*N 步。可以看出，当最终结果比较大时，保留余数信息的算法具有明显的优势。

下面是这个算法的伪代码：BigInt[i]表示模 N 等于 i 的十进制表示形式里只含 1 和 0 的最小整数。由于 BigInt[i]可能很大，又因为它只有 0 和 1，所以，只须要记下 1 的位置即可。比如，整数 1001，记为（0, 3）= 10^0+10^3。即 BigInt 的每个元素是一个变长数组，对于模 N 等于 i 的最小 X，BigInt 的每个元素将存储最小 X 在十进制中表示“1”的位置。我们的目标就是求 BigInt[0]，如代码清单 2-17 所示。

代码清单 2-17

```
// 初始化
    for(i = 0; i < N; i++)
        BigInt[i].clear();
    BigInt[1].push_back(0);

    int NoUpdate = 0;
    for(i=1,j=10%N; ; i++,j=(j*10)%N)
    {
        bool flag = false;
        if(BigInt[j].size() == 0)
        {
            flag = true;
            // BigInt[j] = 10^i, (10^i % N = j)
            BigInt[j].clear();
            BigInt[j].push_back(i);
        }
        for(k = 1; k < N; k++)
        {
            if((BigInt[k].size() > 0)
                && (i > BigInt[k][BigInt[k].size() - 1])
                && (BigInt[(k + j) % N].size() == 0))
            {
```

```
                // BigInt[(k + j) % N] = 10^i + BigInt[k]
                flag = true;
                BigInt[(k + j) % N] = BigInt[k];
                BigInt[(k + j) % N].push_back(i);
            }
        }
        if(flag == false) NoUpdate++;
        else NoUpdate=0;
        // 如果经过一个循环节都没能对 BigInt 进行更新，就是无解，跳出。
        // 或者 BigInt[0] != NULL，已经找到解，也跳出。
        if(NoUpdate == N || BigInt[0].size() > 0)
            break;
    }
    if(BigInt[0].size() == 0)
    {
        // M not exist
    }
    else
    {
        // Find N * M = BigInt[0]
    }
```

在上面的实现中，循环节取值 N（其实循环节小于等于 N，循环节就是最小的 c，使得 $10^c \bmod N = 1$）。

这个算法其实部分借鉴了动态规划算法的思想。在动态规划算法的经典实例“最短路径问题”中，当处理中间结点时，只须要得到从起点到中间结点的最短路径，而不需要中间结点到那个端点的所有路径信息。“只保留模 N 的结果相同的 X 中最小的一个”的方法，与此思路相似。

扩展问题

1. 对于任意的 N，一定存在 M，使得 $N * M$ 的乘积的十进制表示只有 0 和 1 吗？
2. 怎样找出满足题目要求的 N 和 M，使得 $N * M < 2^{16}$，且 $N+M$ 最大？

2.9 ★★ 斐波那契（Fibonacci）数列

斐波那契数列是一个非常美丽、和谐的数列，有人说它起源于一对繁殖力惊人、基因非常优秀的兔子，也有人说远古时期的鹦鹉螺就知道这个规律。

这个数列可以用排成螺旋状的一系列正方形来形象地说明。刚开始，我们把两个边长为 1 的正方形排列在一起（如图 2-2 所示）：

图 2-2

然后我们依次在图形中边长较长的一边接上一个新的正方形（如图 2-3、图 2-4 所示）：

图 2-3

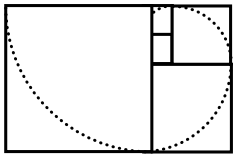

图 2-4

按此顺序依次累加，并在新的正方形中加入以边长为半径的圆弧，我们就会得到美丽的曲线。

每一个学理工科的学生都知道斐波那契数列，斐波那契数列由如下递推关系式定义：

$$F(n)=\begin{cases}0 & \text{if } n=0;\\ 1 & \text{if } n=1;\\ F(n-1)+F(n-2) & \text{if } n>1。\end{cases}$$

每一个上过算法课的同学都能用递归的方法求解斐波那契数列的第 $n+1$ 项的值，即 $F(n)$。

代码清单 2-18

```
int Fibonacci(int n)
{
    if(n <= 0)
    {
```

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

我们的问题是：有没有更加优化的解法？

分析与解法

技术面试的一个常见问题是，对于一个常见的算法，能否进一步优化？这个时候，平时喜欢超越课本思考问题的同学，就有施展才华的机会了。

解法一：递推关系式的优化

上面一页写出的算法是根据递推关系式的定义直接得出的，它在计算 $F[n]$时，须要计算从 $F[2]$到 $F[n-1]$每一项的值，这样简单的递归式存在着很多的重复计算，如求 $F[5]=F[4]+F[3]$，在求 $F[4]$的时候也需要求一次 $F[3]$的大小，等等。请问这个算法的时间复杂度是多少？

那么如何减少这样的重复计算呢？可以用一个数组储存所有已计算过的项。这样便可以达到用空间换取时间的目的。在这种情况下，时间复杂度为 $O(N)$，而空间复杂度也为 $O(N)$。

那么有更快的算法吗？

解法二：求解通项公式

如果我们知道一个数列的通项公式，使用公式来计算会更加容易。能不能把这个函数的递推公式计算出来？

由递推公式 $F(n)=F(n-1)+F(n-2)$，知道 $F(n)$ 的特征方程为：

$x^2=x+1$

有根：$X_{1,2}=\dfrac{1\pm\sqrt{5}}{2}$

所以存在 A，B 使得：

$$F(n)=A\times\left(\frac{1+\sqrt{5}}{2}\right)^n+B\times\left(\frac{1-\sqrt{5}}{2}\right)^n$$

代入 $F(0)=0$，$F(1)=1$，解得 $A=\dfrac{\sqrt{5}}{5}$，$B=-\dfrac{\sqrt{5}}{5}$，即

$$F(n)=\frac{\sqrt{5}}{5}\times\left(\frac{1+\sqrt{5}}{2}\right)^n-\frac{\sqrt{5}}{5}\times\left(\frac{1-\sqrt{5}}{2}\right)^n$$

通过公式，我们可以在 O（1）的时间内得到 F（n）。但公式中引入了无理数，所以不能保证结果的精度。

解法三：分治策略

注意到 Fibonacci 数列是二阶递推数列，所以存在一个 2 * 2 的矩阵 A，使得：

$$(F_n \quad F_{n-1}) = (F_{n-1} \quad F_{n-2}) * A \tag{1}$$

求解，可得：

$$A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

由（1）式我们有：

$$(F_n \quad F_{n-1}) = (F_{n-1} \quad F_{n-2}) * A = (F_{n-2} \quad F_{n-3}) * A^2 = \cdots = (F_1 \quad F_0) * A^{n-1}$$

剩下的问题就是求解矩阵 A 的方幂。

$A^n = A * A * \cdots * A$。最直接的解法就是通过 $n-1$ 次乘法得到结果。但是当 n 很大时，比如 1 000 000 或 1 000 000 000，这个算法的效率就不能接受了。当然，你马上会说，在这个情况下，F_n 在整数里早就溢出了，但如果须要求解的是 F_n 对某个素数的余数呢？这个算法会是非常有用和高效的。

我们注意到：

$A^{x+y} = A^x * A^y$；

$A^{x*2} = A^{x+x} =$（A^x）^{2}；

用二进制方式表示 n：

$n = a_k * 2^k + a_{k-1} * 2^{k-1} + \cdots + a_1 * 2 + a_0$（其中 a_i=0 或 1，i=0, 1, ⋯, k）；

$$A^n = A^{a_k * 2^k + a_{k-1} * 2^{k-1} \ldots + a_0} = \left(A^{2^k}\right)^{a_k} * \left(A^{2^{k-1}}\right)^{a_{k-1}} * \cdots * \left(A^{2^1}\right)^{a_1} * A^{a_0}$$

如果能够得到 A^{2^i}（i=1, 2, ⋯, k）的值，就可以再经过 $\log_2 n$ 次乘法得到 A^n。

而这显然容易通过递推得到：

$$A^{2^i} = \left(A^{2^{i-1}}\right)^2$$

举个例子：

$$A^5 = A^{1*2^2+0*2^1+1*2^0} = A^{2^2} * A^{2^0}$$

求解：$A^{2^0} = A$

$$A^{2^1} = \left(A^{2^0}\right)^2$$

$$A^{2^2} = \left(A^{2^1}\right)^2$$

具体的代码如清单 2-19 所示。

代码清单 2-19

```
Class Matrix;                          // 假设我们已经有了实现乘法操作的矩阵类
                                       // 求解 m 的 n 次方
Matrix MatrixPow(const Matrix& m, int n)
{
    Matrix result = Matrix::Identity;       // 赋初值为单位矩阵
    Matrix tmp = m;
    for(; n; n >>= 1)
    {
        if (n & 1)
            result *= tmp;
        tmp *= tmp;
    }
}
int Fibonacci(int n)
{
    Matrix an = MatrixPow(A, n - 1);        // A 的值就是上面求解出来的
    return F1* an(0, 0) + F0 * an(1, 0);    // 返回 Fn
}
```

整个算法的时间复杂度是 O（$\log_2 n$）。

扩展问题

假设 A（0）= 1, A（1）= 2, A（2）= 2。对于 n>2，都有 A（k）=A（k–1）+ A（k–2）+A（k–3）。

1. 对于任何一个给定的 n，如何计算出 A（n）？
2. 对于 n 非常大的情况，如 n=2^{60} 的时候，如何计算 A(n)mod M（M < 100000）呢？

2.10 ★★ 寻找数组中的最大值和最小值

数组是最简单的一种数据结构。我们经常碰到的一个基本问题，就是寻找整个数组中最大的数，或者最小的数。这时，我们都会扫描一遍数组，把最大（最小）的数找出来。如果我们需要同时找出最大和最小的数呢？

对于一个由 N 个整数组成的数组，需要比较多少次才能把最大和最小的数找出来呢？

分析与解法

解法一

可以把寻找数组中的最大值和最小值看成是两个独立的问题，我们只要分别求出数组的最大值和最小值即可解决问题。最直接的做法是先扫描一遍数组，找出最大的数以及最小的数。这样，我们需要比较 2 * *N* 次才能找出最大的数和最小的数。

能否在这两个看似独立的问题之间建立关联，从而减少比较的次数呢？

解法二

一般情况下，最大的数和最小的数不会是同一个数（除非 *N*=1，或者所有整数都是一样的大小）。所以，我们希望先把数组分成两部分，然后再从这两部分中分别找出最大的数和最小的数。

首先按顺序将数组中相邻的两个数分在同一组（这只是概念上的分组，无须做任何实际操作）。若数组为｛5, 6, 8, 3, 7, 9｝（如图 2-5 所示）：

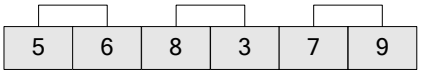

图 2-5 数组示意图

接着比较同一组中奇数位数字和偶数位数字，将较大的数放在偶数位上，较小的数放在奇数位上。经过 *N*/2 次比较的预处理后，较大的数都放到了偶数位置上，较小的数则放到了奇数位置上，如图 2-6 所示：

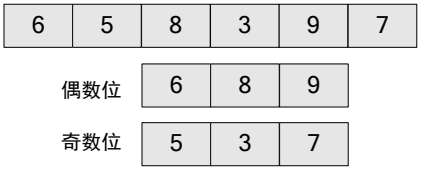

图 2-6

最后，我们从奇偶数位上分别求出 Max=9，Min=3，各需要比较 *N*/2 次。整个算法共需要比较 1.5 * *N* 次。

解法三

解法二已经将比较次数降低到了 1.5 * *N* 次，但它破坏了原数组，如何能够不破坏原数组呢？解法二是事先将两两分组中较小和较大的数调整了顺序，从而破坏了数组，如果可以在遍历的过程中进行比较，而不需要对数组中的元素进行调换，就可以不用破坏原数组了。首先仍然按顺序将数组中相邻的两个数分在同一组（这只是概念上的分组，无须做任何实际操作）。然后可以利用两个变量 Max 和 Min 来存储当前的最大值和最小值。同一组的两个数比较之后，不再调整顺序，而是将其中较小的数与当前 Min 作比较，如果该数小于当前 Min 则更新 Min。同理，将其中较大的数与当前 Max 作比较，如果该数大于当前 Max，则更新 Max。如此反复比较，直到遍历完整个数组。Min 和 Max 分别被初始化为数组第一和第二个数中的小者和大者。仍设原数组为｛5, 6, 8, 3, 7, 9｝，则 Min 和 Max 分别被初始化为 6 和 5。比较过程如图 2-7 所示：

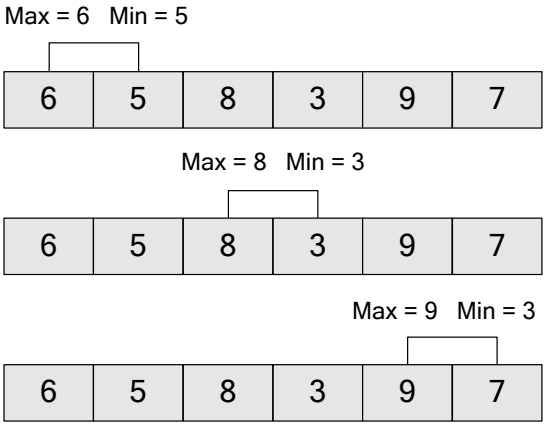

图 2-7 查找比较示意图

最后，Max=9，Min=3。但是时间复杂度并未降低，整个过程的比较次数仍为 1.5 * *N* 次。

解法四

分治思想是算法中很常用的一种技巧。在 *N* 个数中求最小值 Min 和最大值 Max，我们只需分别求出前后 *N*/2 个数的 Min 和 Max，然后取较小的 Min，较大的 Max 即可（只需较大的数和较大的数比较，较小的数和较小的数比较，两次就可以了）。

假设我们要求 arr[1, 2, … , *n*]数组的最大数和最小数，上述算法的伪代码如清单 2-20 所示：

代码清单 2-20

```
(max, min) Search(arr, b, e)
{
    if(e - b <= 1)
    {
        if(arr[b] < arr[e])
            return (arr[e], arr[b]);
        else
            return (arr[b], arr[e]);
    }
    (maxL, minL) = Search(arr, b, b + (e - b) / 2);
    (maxR, minR) = Search(arr, b + (e - b) / 2 + 1, e);
    if(maxL > maxR)
        maxV = maxL;
    else
        maxV = maxR;
    if(minL < minR)
        minV = minL;
    else
        minV = minR;
    return (maxV, minV);
}
```

如果用 $f(N)$ 表示这个算法对于 N 个数的情况需要比较的次数。我们可以得到：

$$f(2)=1$$

$$\begin{aligned} f(N) &= 2*f(N/2)+2 \\ &= 2\ *(2*f(N/2^2)+2)+2 \\ &= 2^2*f(N/2^2)+2^2+2 \\ &\cdots \\ &= 2^{(\log_2 N)-1}*f(N/2^{(\log_2 N)-1})+2^{\log_2 N-1}+\cdots+2^2+2 \\ &= \frac{N}{2}*f(2)+2^{(\log_2 N)-1}+\cdots+2^2+2 \\ &= \frac{N}{2}*f(2)+\frac{2*(1-2^{(\log_2 N)-1})}{1-2} \\ &= \frac{N}{2}+\frac{2*(1-\frac{N}{2})}{1-2} \\ &= 1.5N-2 \end{aligned}$$

所以说即使采用分治法，总的比较次数仍然没有减少。

扩展问题

如果需要找出 N 个数组中的第二大数，需要比较多少次呢？是否可以使用类似的分治思想来降低比较的次数呢？

2.11 ★★★ 寻找最近点对

给定平面上 N 个点的坐标，找出距离最近的两个点（如图 2-8 所示）。

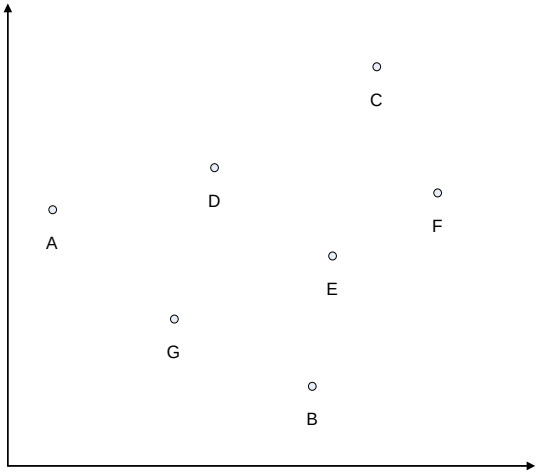

图 2-8 七个平面上的点

分析与解法

初看这个问题，会觉得不太容易解答，没有什么头绪，在面试的时候，怎么办？我们不妨先看看一维的情况：在一个包含 N 个数的数组中，如何快速找出 N 个数中两两差值的最小值？一维的情况相当于所有的点都在一条直线上。虽然是一个退化的情况，但还是能从中得到一些启发。

解法一

数组中总共包含 N 个数，我们把它们两两之间的差值都求出来，那样就不难得出最小的差值了。这样一个直接的想法，时间复杂度为 $O(N^2)$。伪代码如清单 2-21 所示：

代码清单 2-21

```
double MinDifference(double arr[], int n)
{
    if(n < 2)
    {
        return 0;
    }
    double fMinDiff = fabs(arr[0] - arr[1]);
    for(int i = 0; i < n; ++i)
        for(int j = i + 1; j <n; ++j)
        {
            double tmp = fabs(arr[i] - arr[j]);
            if(fMinDiff > tmp)
            {
                fMinDiff = tmp;
            }
        }
    return fMinDiff;
}
```

如果扩展到二维的情况，那就相当于枚举任意两个点，然后再记录下距离最近的点对。时间复杂度也是 $O(N^2)$。这还是一个很直接的想法，能否继续改进呢？

解法二

如果数组有序，找出最小的差值就很容易了。可以用 $O(N * \log_2 N)$ 的算法进行排序（快速排序、堆排序、归并排序等）。排序完成后，找最小差值只需要 $O(N)$ 的时间，总时间复杂度是 $O(N * \log_2 N)$。

代码清单 2-22

```
double MinDifference(double arr[], int n)
{
    if(n < 2)
    {
        return 0;
    }
    // Sort array arr[]
    Sort(arr, arr + n);

    double fMinDiff = arr[1] - arr[0];
    for(int i = 2; i < n; ++i)
    {
        double tmp = arr[i] - arr[i - 1];
        if(fMinDiff > tmp)
        {
            fMinDiff = tmp;
        }
    }
    return fMinDiff;
}
```

在一维情况下，时间复杂度改进了不少。但是这个方法不能推广到二维的情况，因为距离最近的点对不能保证是映射到某条直线之后紧靠着的两个点。如图 2-9 所示，点 *A* 和 *C* 的距离最近，但它们在 X 轴上的投影点却不是相邻的。

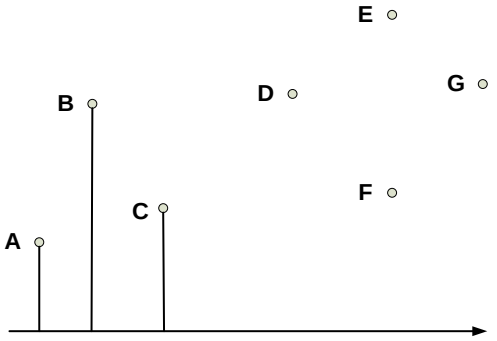

图 2-9 二维投影示意图

解法三

还有什么想法呢？如果我们用数组的中间值 *k* 把数组分成 Left、Right 两部分，小于 *k* 的数为 Left 部分，其他的为 Right 部分，那么这个最小差值要么来自 Left 部分，要么来自 Right 部分，要么是 Left 中最大数和 Right 中最小数的差值。在这里，我们其实借用了分治思想。时间复杂度仍然为 $O(N * \log_2 N)$。

这个方法中的分治思想也可以扩展到二维的情况。

根据水平方向的坐标把平面上的 N 个点分成两部分 Left 和 Right。跟以往一样，我们希望这两个部分点数的个数差不多。假设分别求出了 Left 和 Right 两个部分中距离最近的点对之最短距离为 MinDist（Left）和 MinDist（Right），还有一种情况我们没有考虑，那就是点对中一个点来自于 Left 部分，另一个点来自于 Right 部分。最直接的想法，那就是穷举 Left 和 Right 两个部分之间的点对，这样的点对很多，最多可能有 $N * N/4$ 对。显然，穷举所有 Left 和 Right 之间的点对是不好的做法。是否可以只考虑有可能成为最近点对的候选点对呢？由于我们已经知道 Left 和 Right 两个部分中的最近点对距离分别为 MinDist（Left）和 MinDist（Right），如果 Left 和 Right 之间的点对距离超过 MDist = MinValue（MinDist（Left），MinDist（Right）），我们则对它们并不感兴趣，因为这些点对不可能是最近点对。

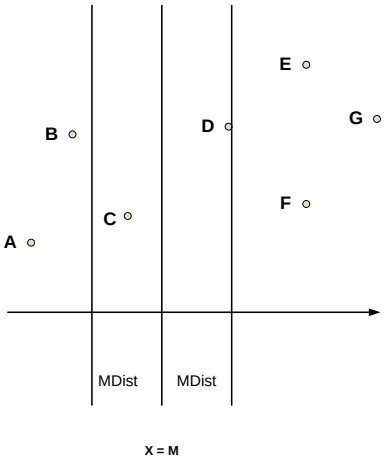

图 2-10 二维点分布示意图

如图 2-10 所示，通过直线 $x = M$ 将所有的点分成 $x < M$ 和 $x > M$ 两部分，在分别求出两部分的最近点对之后，只需要考虑点对 CD。因为其他点对 AD、BD、CE、CF、CG 等都不可能成为最近点对。也就是说，只要考虑从 $x = M$ – Mdist 到 $x = M$ + MDist 之间这个带状区域内的最小点对，然后再跟 MDist 比较就可以了。在计算带状区域的最小点对时，可以按 Y 坐标，对带状区域内的顶点进行排序。如果一个点对的距离小于 MDist，那么它们一定在一个 MDist *（2 * Mdist）的区域内（如图 2-11 所示）。

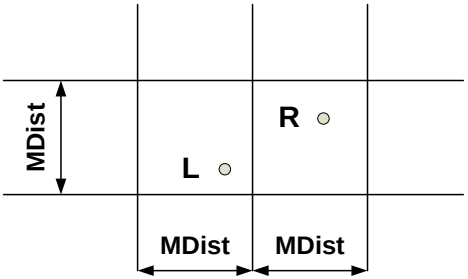

图 2-11 区域示意图

而在左右两个 Mdist * MDist 正方形区域内，最多都只能含有 4 个点。如果超过 4 个点，则这个正方形区域内至少存在一个点对的距离小于 Mdist，这跟 $x < M$ 和 $x > M$ 两个部分的最近点对距离分别是 MinDist（Left）和 MinDist（Right）矛盾。

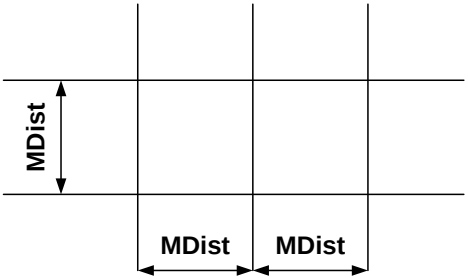

图 2-12 区域示意图

因此，一个 MDist *（2 * Mdist）的区域内最多有 8 个点（如图 2-12 所示）。对于任意一个带状区域内的顶点，只要考察它与按 Y 坐标排序且紧接着的 7 个点之间的距离就可以了。根据这个特点，我们可用 $O(N)$ 时间完成带状区域最近点对的查找。在这一步，需要注意的是：我们可以用归并排序法将带状区域的点按 Y 坐标排序。归并排序的过程与计算最近点对的算法结合在一起。

整个算法的时间复杂度 $f(N)$ 的递归表达式为:

$F(2) = 1$

$F(3) = 3$

$F(N) = 2 * f(N/2) + O(N)\ (N > 2)$

可以计算出 $f(N) = N * \log_2 N$。也就是说，我们用 $O(N * \log_2 N)$ 的时间可以完成最近点对问题。

扩展问题

1. 如果给定一个数组 arr[0, …, *N*−1]，要求找出相邻两个数的最大差值。对于数 *X* 和 *Y*，如果不存在其他数组中的数在[*X*, *Y*]区间内，则我们称 *X* 和 *Y* 是相邻的。

   我们也可以使用上面的方法来解决这个扩展问题。不过，这个扩展问题还有一个更漂亮的解法。如果在这个数组 arr 中，最大的数为 arr.MaxValue，而最小的数为 arr.MinValue，那么根据抽屉原理，相邻两个数的最大差值一定不小于 delta = （arr.MaxValue – arr.MinValue）/（*N* – 1）。

   证明：假设任意相邻两个数的差值都小于 delta，而数组 arr 从小到大排好序后得到数组 sorted_arr, sorted_arr[0] < sorted_arr[1] < … < sorted_arr[*N*−1]，则 sorted_arr[1] – sorted_arr[0] < delta, sorted_arr[2] – sorted_arr[1] < delta, …, sorted_arr[*N*−1] – sorted_arr[*N*−2] < delta，有 arr.MaxValue – arr.MinValue = sorted_arr[*N*−1] – sorted_arr[0] = （sorted_arr[*N*−1] – sorted_arr[*N*−2]）+ … +（sorted_arr[1] – sorted_arr[0]）< delta *（*N*− 1）= arr.MaxValue − arr.MinValue。那么它与假设矛盾。

   通过上面的分析，我们知道最大的差值大于等于 delta。因此，我们忽略小于 delta 的差值，因为这些差值都不可能是答案。一个方法就是我们把区间[arr.MinValue, arr.MaxValue]分成 *N* 个桶：[arr.MinValue, arr.MinValue], [arr.MinValue + delta], [arr.MinValue + delta, arr.MinValue + delta * 2], …, [arr.MaxValue − delta, arr.MaxValue]。综合上面的分析，我们知道最大的差值应该出现在不同的桶之间（除非 delta = 0），因为处于同一个桶的两个数的差值不超过 delta，我们可以忽略不计。既然最大的差值处于两个不同的桶之间，那么它们就等于某个桶的最小值与前一个非空桶的最大值的差值。

   根据上面的分析，可以申请 *O*（*N*）大小的空间来存储每一个桶的最大值和最小值，然后再扫描一遍所有的桶就可以得到整个数组的最大差值。整个算法的空间复杂度和时间复杂度均为 *O*（*N*）。

2. 如果给定的是平面上的 *N* 个点，如何寻找距离最远的两个点呢？

2.12 ★★ 快速寻找满足条件的两个数

能否快速找出一个数组中的两个数字，让这两个数字之和等于一个给定的数字，为了简化起见，我们假设这个数组中肯定存在至少一组符合要求的解。

分析与解法

这个题目不是很难，也很容易理解。但是要得出高效率的解法，还是需要一番思考的。

解法一

刚看到这个题目，很容易想到的解法就是穷举：从数组中任意取出两个数字，计算两者之和是否为给定的数字。

显然其时间复杂度为 N（N−1）/2 即 O（N^2）。这个算法很简单，写起来也很容易，但是效率不高。一般在程序设计里面，要尽可能降低算法的时间和空间复杂度，所以需要继续寻找效率更高的解法。

解法二

求两个数字之和，假设给定的和为 Sum。一个变通的思路，就是对数组中的每个数字 arr[i]都判别 Sum−arr[i]是否在数组中。这样，就变通成为一个查找的算法。

在一个无序数组中查找一个数的复杂度是 O（N），对于每个数字 arr[i]，都需要查找对应的 Sum−arr[i]在不在数组中，很容易得到时间复杂度还是 O（N^2）。这和最原始的方法相比没有改进。但是如果能够提高查找的效率，就能够提高整个算法的效率。怎样提高查找的效率呢？

学过编程的人都知道，提高查找效率通常可以先将要查找的数组排序，然后用二分查找等方法进行查找，就可以将原来 O（N）的查找时间缩短到 O（$\log_2 N$）。这样对于每个 arr[i]，都要花 O（$\log_2 N$）去查找对应的 Sum−arr[i]在不在数组中，总的时间复杂度降低为 $N\log_2 N$。当然将长度为 N 的数组进行排序本身也需要 O（$N\log_2 N$）的时间，好在只须要排序一次就够了，所以总的时间复杂度依然是 O（$N\log_2 N$）。这样，就改进了最原始的方法。

到这里，有的读者可能会更进一步地想，先排序再二分查找固然可以将时间从 O（N^2）缩短到 O（$\log_2 N$），但是还有更快的查找方法：hash 表。因为给定一个数字，根据 hash 映射查找另一个数字是否在数组中，只需用 O（1）时间。这样的话，总体的算法复杂度可以降低到 O（N），但这种方法需要额外增加 O（N）的 hash 表存储空间。在有的情况下，用空间换时间也并不失为一个好方法。

解法三

还可以换个角度来考虑这个问题，假设已经有了这个数组的任意两个元素之和的有序数组（长为 N^2）。那么利用二分查找法，只需用 $O(2\log_2 N)$ 就可以解决这个问题。当然不太可能去计算这个有序数组，因为它需要 $O(N^2)$ 的时间。但这个思考仍启发我们，可以直接对两个数字的和进行一个有序的遍历，从而降低算法的时间复杂度。

首先对数组进行排序，时间复杂度为（$N\log_2 N$）。

然后令 $i=0$，$j=n-1$，看 arr[i] + arr[j]是否等于 Sum，如果是，则结束。如果小于 Sum，则 $i=i+1$；如果大于 Sum，则 $j=j-1$。这样只需要在排好序的数组上遍历一次，就可以得到最后的结果，时间复杂度为 $O(N)$。两步加起来总的时间复杂度 $O(N\log_2 N)$，下面这个程序就利用了这个思想，如代码清单 2-23 所示。

代码清单 2-23 伪代码

```
for(i = 0, j = n - 1; i < j; )
    if(arr[i] + arr[j] == Sum)
        return (i, j);
    else if(arr[i] + arr[j] < Sum)
        i++;
    else
        j--;
return (-1, -1) ;
```

它的时间复杂度是 $O(N)$。

扩展问题

注意我们题目有一个重要的前提，就是数组中肯定存在这样的一对数字。考虑下面的扩展问题。

1. 如果把这个问题中的“两个数字”改成“三个数字”或“任意个数字”时，你的解是什么呢？
2. 如果完全相等的一对数字对找不到，能否找出和最接近的解？
3. 把上面的两个题目综合起来，就得到这样一个题目：给定一个数 N 和一组数字集合 S，求 S 中和最接近 N 的子集。想继续钻研下去的读者，可以看一看专业书籍中关于 NP，NP-Complete 的描述。

面试中很多题目都是给定一个数组，要求返回两个下标的（比如找两个元素，或者找一个子数组）。而相应比较高效的解法，则是先排序，然后在一个循环体里利用两个变量进行反向的遍历，并且这两个变量遍历的方向是不变的，从而保证遍历算法的时间复杂度是 $O(N)$。以后读者再遇到类似的问题，也可以考虑利用两个下标进行遍历。

2.13 ★★ 子数组的最大乘积

给定一个长度为 N 的整数数组，只允许用乘法，不能用除法，计算任意（N–1）个数的组合乘积中最大的一组，并写出算法的时间复杂度。

我们把所有可能的（N–1）个数的组合找出来，分别计算它们的乘积，并比较大小。由于总共有 N 个（N–1）个数的组合，总的时间复杂度为 $O(N^2)$，但显然这不是最好的解法。

分析与解法

解法一

在计算机科学中，时间和空间往往是一对矛盾体，不过，这里有一个优化的折中方法。可以通过“空间换时间”或“时间换空间”的策略来达到优化某一方面的效果。在这里，是否可以通过“空间换时间”来降低时间复杂度呢？

计算（N−1）个数的组合乘积，假设第 i 个（0≤i≤N−1）元素被排除在乘积之外（如图 2-13 所示）。

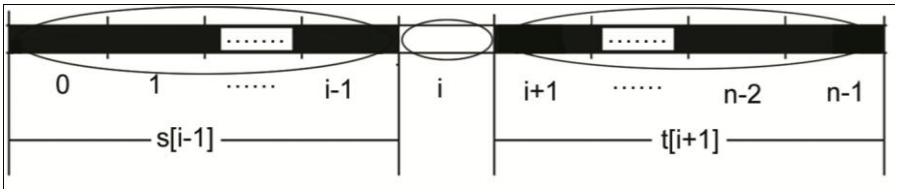

图 2-13 组合示意图

设 *array*[]为初始数组，*s*[*i*]表示数组前 i 个元素的乘积$s[i]=\prod_{1}^{i} array[i-1]$，其中 1≤$i$≤$N$，*s*[0] = 1（边界条件），那么 *s*[*i*]= *s*[*i*−1]×*array*[*i*−1]，其中 i = 1, 2, …, N−1, N；

设 *t*[*i*]表示数组后（N−i）个元素的乘积$t[i]=\prod_{i}^{n} array[i]$，其中 1≤$i$≤$N$，*t*[$N$+1]=1（边界条件），那么 *t*[*i*]=*t*[*i*+1]×*array*[*i*]，其中 i=1, 2, …, N−1, N；

那么设 *p*[*i*]为数组除第 i 个元素外，其他 N−1 个元素的乘积，即有：

p[*i*]=*s*[*i*−1]×*t*[*i*+1]。

由于只需要从头至尾，和从尾至头扫描数组两次即可得到数组 *s*[]和 *t*[]，进而线性时间可以得到 *p*[]。所以，很容易就可以得到 *p*[]的最大值（只需遍历 *p*[]一次）。总的时间复杂度等于计算数组 *s*[]、*t*[]、*p*[]的时间复杂度加上查找 *p*[]最大值的时间复杂度等于 O（N）。

解法二

其实，还可以通过分析，进一步减少解答问题的计算量。假设 N 个整数的乘积为 P，针对 P 的正负性进行如下分析（其中，A_{N-1} 表示 N−1 个数的组合，P_{N-1} 表示 N−1 个数的组合的乘积）。

1. P 为 0
   那么，数组中至少包含有一个 0。假设除去一个 0 之外，其他 N–1 个数的乘积为 Q，根据 Q 的正负性进行讨论：
   Q 为 0
   说明数组中至少有两个 0，那么 N–1 个数的乘积只能为 0，返回 0；
   Q 为正数
   返回 Q，因为如果以 0 替换此时 A_{N-1} 中的任一个数，所得到的 P_{N-1} 为 0，必然小于 Q；
   Q 为负数
   如果以 0 替换此时 A_{N-1} 中的任一个数，所得到的 P_{N-1} 为 0，大于 Q，乘积最大值为 0。
2. P 为负数
   根据“负负得正”的乘法性质，自然想到从 N 个整数中去掉一个负数，使得 P_{N-1} 为一个正数。而要使这个正数最大，这个被去掉的负数的绝对值必须是数组中最小的。我们只需要扫描一遍数组，把绝对值最小的负数给去掉就可以了。
3. P 为正数
   类似 P 为负数的情况，应该去掉一个绝对值最小的正数值，这样得到的 P_{N-1} 就是最大的。

上面的解法采用了直接求 N 个整数的乘积 P，进而判断 P 的正负性的办法，但是直接求乘积在编译环境下往往会有溢出的危险（这也就是本题要求不使用除法的潜在用意☺），事实上可做一个小的转变，不需要直接求乘积，而是求出数组中正数（+）、负数（–）和 0 的个数，从而判断 P 的正负性，其余部分与以上面的解法相同。

在时间复杂度方面，由于只需要遍历数组一次，在遍历数组的同时就可得到数组中正数（+）、负数（–）和 0 的个数，以及数组中绝对值最小的正数和负数，时间复杂度为 O（N）。

2.14 ★★★ 求数组的子数组之和的最大值

一个有 N 个整数元素的一维数组（$A[0], A[1], \cdots, A[n-2], A[n-1]$），这个数组当然有很多子数组，那么子数组之和的最大值是什么呢？

这是一道看似简单，实际上也挺简单，但是却难倒了不少学生的题目。这也是很多公司面试的题目，微软亚洲研究院曾在 2006 年的笔试题中出过这道题，只有 20%的人能够写出正确的解法，能做到最优 $O(N)$ 解法的同学非常少。事实上这个题目及相关解答在网上都能够找到，不过散布于网上的几个版本似乎都有不正确的地方，我们在这里总结一下。

分析与解法

解法一

我们先明确题意。

1. 题目说的子数组，是连续的。
2. 题目只需要求和，并不需要返回子数组的具体位置。
3. 数组的元素是整数，所以数组可能包含有正整数、零、负整数。

举几个例子：

数组：[1，−2，3，5，−3，2]应返回：8

数组：[0，−2，3，5，−1，2]应返回：9

数组：[−9，−2，−3，−5，−3]应返回：−2，这也是最大子数组的和。

这几个典型的输入能帮助我们测试算法的逻辑。在写具体算法前列出各种可能输入，也可以让应聘者有机会和面试者交流，明确题目的要求。例如：如果数组中全部是负数，怎么办？是返回 0，还是最大的负数？这是面试和闭卷考试不一样的地方，要抓住机会交流。

了解了题意之后，我们试验最直接的方法，记 Sum[i, …, j]为数组 A 中第 i 个元素到第 j 个元素的和（其中 0 <= i <= j < n），遍历所有可能的 Sum[i, …, j]，那么时间复杂度为 O（N^3）：

代码清单 2-24

```
int MaxSum(int* A, int n)
{
    int maximum = -INF;
    int sum;
    for(int i = 0; i < n; i++)
    {
        for(int j = i; j < n; j++)
        {
            for(int k = i; k <= j; k++)
            {
                sum += A[k];
            }
            if(sum > maximum)
                maximum = sum;
        }
    }
    return maximum;
```

```
}
```

如果注意到 Sum[i, …, j] = Sum[i, …, j−1] + A[j]，则可以将算法中的最后一个 `for` 循环省略，避免重复计算，从而使得算法得以改进，改进后的算法如下，这时复杂度为 O（N^2）：

代码清单 2-25

```
int MaxSum(int* A, int n)
{
    int maximum = -INF;
    int sum;
    for(int i = 0; i < n; i++)
    {
        sum = 0;
        for(int j = i; j < n; j++)
        {
            sum += A[j];
            if(sum > maximum)
                maximum = sum;
        }
    }
    return maximum;
}
```

能继续优化吗?

解法二

如果将所给数组（A[0], …, A[n−1]）分为长度相等的两段数组（A[0], …, A[n/2−1]）和（A[n/2], …, A[n−1]），分别求出这两段数组各自的最大子段和，则原数组（A[0], …, A[n−1]）的最大子段和为以下三种情况的最大值：

1. （A[0], …, A[n−1]）的最大子段和与（A[0], …, A[n/2−1]）的最大子段和相同；
2. （A[0], …, A[n−1]）的最大子段和与（A[n/2], …, A[n−1]）的最大子段和相同；
3. （A[0], …, A[n−1]）的最大子段跨过其中间两个元素 A[n/2−1]到 A[n/2]。

第 1 和 2 两种情况事实上是问题规模减半的相同子问题，可以通过递归求得。

至于第 3 种情况，我们只要找到以 A[n/2−1]结尾的和最大的一段数组和 s_1=（A[i], …, A[n/2−1]）（0<=i<n/2−1）A[]和以 A[n/2]开始和最大的一段和 s_2=（A[n/2], …, A[j]）（n/2< =j < n）。那么第 3 种情况的最大值为 s_1+s_2 = A[i]+…+A[n/2−1] + A[n/2] +…+A[j]，只须要对原数组进行一次遍历即可。

其实这是一种分治算法，每个问题都可分解成为两个问题规模减半的子问题，再加上一次遍历算法。该分治算法的时间复杂度满足典型的分治算法递归式，总的时间复杂度为 $T(N) = O(N * \log_2 N)$。

解法三

解法二中的分治算法已经将时间复杂度从 $O(N^2)$ 降到了 $O(N * \log_2 N)$，应该说是一个不错的改进，但是否还可以进一步将时间复杂度降低呢？答案是肯定的，从分治算法中得到提示：可以考虑数组的第一个元素 $A[0]$，以及最大的一段数组（$A[i], \cdots, A[j]$）跟 $A[0]$之间的关系，有以下几种情况：

1. 当 $0 = i = j$ 时，元素 $A[0]$本身构成和最大的一段；
2. 当 $0 = i < j$ 时，和最大的一段以 $A[0]$开始；
3. 当 $0 < i$ 时，元素 $A[0]$跟和最大的一段没有关系。

从上面三种情况可以看出，可以将一个大问题（N 个元素数组）转化为一个较小的问题（$n-1$ 个元素的数组）。假设已经知道（$A[1], \cdots, A[n-1]$）中和最大的一段之和为 $All[1]$，并且已经知道（$A[1], \cdots, A[n-1]$）中包含 $A[1]$的和最大的一段的和为 $Start[1]$。那么，根据上述分析的三种情况，不难看出（$A[0], \cdots, A[n-1]$）中问题的解 $All[0]$是三种情况的最大值 $\max\{A[0], A[0]+Start[1], All[1]\}$。通过这样的分析，可以看出这个问题符合无后效性，可以使用动态规划的方法来解决。

代码清单 2-26

```
int max(int x, int y)                    // 返回 x,y 两者中的较大值
{
    return (x > y) ? x : y;
}

int MaxSum(int* A, int n)
{
    Start[n - 1] = A[n - 1];
    All[n - 1] = A[n - 1];
    for(i = n - 2; i >= 0; i--)          // 从数组末尾往前遍历，直到数组首
    {
        Start[i] = max(A[i], A[i] + Start[i + 1]);
        All[i] = max(Start[i], All[i + 1]);
    }
    return All[0];                       // 遍历完数组，All[0]中存放着结果
}
```

新方法的时间复杂度已经降到 $O(N)$ 了。

但一个新的问题出现了：我们又额外申请了两个数组 *All*[]、*Start*[]，能否在空间方面也节省一点呢？

观察这两个递推式：

```
Start[i] = max{A[i], Start[i+1] + A[i]}
All[i] = max{Start[i], All[i+1]}
```

第一个递推式：*Start*[*i*] = max{*A*[*i*], *Start*[*i*+1] + *A*[*i*]}。如果 *Start*[*i*+1] < 0，则 *Start*[*i*] = *A*[*i*]。而且，在这两个递推式中，其实都只须要用两个变量就可以了。*Start*[*k*+1]只有在计算 *Start*[*k*]时使用，而 *All*[*k*+1]也只有在计算 *All*[*k*]时使用。所以程序可以进一步改进一下，只需 *O*（1）的空间就足够了。

代码清单 2-27

```
int max(int x, int y)
{
    return (x > y) ? x : y;
}                              // 用于比较 x 和 y 的大小，返回 x 和 y 中的较大者

int MaxSum(int* A, int n)
{
// 要做参数检查
    nStart = A[n - 1];
    nAll = A[n - 1];
    for(i = n-2; i >= 0; i--)
    {
        nStart = max(A[i], nStart + A[i]);
        nAll = max(nStart, nAll);
    }
    return nAll;
}
```

改进的算法不仅节省了空间，而且只有寥寥几行，却达到了很高的效率，是不是很美呢？

我们还可以换一个写法：

代码清单 2-28

```
int MaxSum(int* A, int n)
{
                                       // 要做输入参数检查
    nStart = A[n - 1];
    nAll = A[n - 1];
    for(i = n - 2; i >= 0; i--)
    {
        if(nStart < 0)
            nStart = 0;                // 数组全部是负数，如何？
        nStart += A[i];
        if(nStart > nAll)
            nAll = nStart;
```

```
    }
    return nAll;
}
```

扩展问题

1. 如果数组（$A[0], \cdots, A[n-1]$）首尾相邻，也就是我们允许找到一段数字（$A[i], \cdots, A[n-1], A[0], \cdots, A[j]$），请使其和最大，怎么办？

   可以把问题的解分为两种情况。

   （1）解没有跨过 $A[n-1]$到 $A[0]$（原问题）。

   （2）解跨过 $A[n-1]$到 $A[0]$。

   对于第二种情况，只要找到从 $A[0]$开始和最大的一段（$A[0], \cdots, A[j]$）（$0<=j<n$），以及以 $A[n-1]$结尾的和最大的一段（$A[i], \cdots, A[n-1]$）（$0<=i<n$），那么，第二种情况中，和的最大值 M_2 为：

   $M\_2 = A[i] + \cdots + A[n-1] + A[0] + \cdots + A[j]$

   如果 $i <= j$，则

   $M\_2 = A[0] + \cdots + A[n-1]$

   否则

   $M\_2 = A[0] + \cdots + A[j] + A[i] + \cdots + A[n-1]$

   最后，再取两种情况的最大值就可以了，求解跨过 $A[n-1]$到 $A[0]$的情况只需要遍历数组一次，故总的时间复杂度为 $O(N)+O(N)=O(N)$。

2. 如果题目要求同时返回最大子数组的位置，算法应如何改变？还能保持 $O(N)$的时间复杂度么？

2.15 ★★★ 子数组之和的最大值（二维）

我们在前面分析了一维数组的子数组之和最大值的问题，那么如果是二维数组又该如何分析呢？

分析与解法

最直接的方法，当然就是枚举每一个矩形区域，然后再求这个矩形区域中元素的和。

解法一

代码清单 2-29

```
int max(int x, int y)
{
    return (x > y) ? x : y;  // 用于比较 x 和 y 的大小，返回 x 和 y 中的较大者
}

// @parameters
// A, 二维数组
// n, 行数
// m, 列数
int MaxSum(int* A, int n, int m)
{
    maximum = -INF;
    for(i_min = 1; i_min <= n; i_min++)
        for(i_max = i_min; i_max <= n; i_max++)
            for(j_min = 1; j_min <= m; j_min++)
                for(j_max = j_min; j_max <= m; j_max++)
                    maximum = max(maximum, Sum(i_min, i_max, j_min,
                      j_max));
    return maximum;
}
```

采用这种方法的时间复杂度为 $O(N^2 * M^2 * \text{Sum 的时间复杂度})$。上面 Sum 函数是以（i_min, j_min）、（i_min, j_max）、（i_max, j_min）、（i_max, j_max）为顶点的矩形区域（图 2-14 的灰色部分）中元素之和。求矩形区域中元素之和若仍采用最直接的遍历，时间复杂度就太大了。

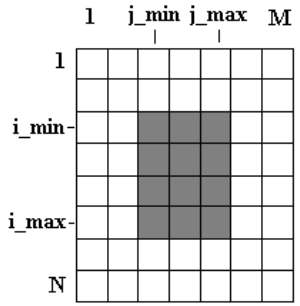

图 2-14 二维最大值示意图

考虑到区域的和需要被频繁计算，或许我们可以做一些预处理，并把计算结果存下来，以达到“空间换时间”的目的。事实上，的确可以这样做。通过“部分和”的 $O(N*M)$ 预处理，可以在 $O(1)$ 时间内计算出任意一个区域的和。

关于“部分和”，先看一维数组（$A[1], \cdots, A[n]$），如果事先记录下 PS[i]：

PS[0] = 0，边界值

PS[i] = PS[$i-1$] + $A[i]$ = $A[1] + \cdots + A[i]$（$0 < i <= n$）

那么，数组中任意一段（$A[i], \cdots, A[j]$）的元素之和等于 PS[j]−PS[$i-1$]。

类似地，在二维情况下，定义“部分和”PS[i][j]等于以（1, 1），（i, 1），（1, j），（i, j）为顶点的矩形区域的元素之和。

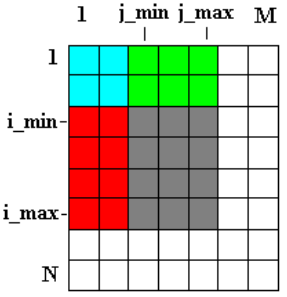

图 2-15 二维最大值示意图

通过图 2-15 也可以看出，以（i_min, j_min），（i_min, j_max），（i_max, j_min），（i_max, j_max）为顶点的矩形区域的元素之和，等于 PS[i_max][j_max]−PS[i_min−1][j_max]−PS[i_max][j_min−1] + PS[i_min−1][j_min−1]。也就是在已知“部分和”的基础上可以用 $O(1)$ 时间算出任意矩形区域的元素之和。

万事俱备，只欠“部分和”。怎么快速预处理以得到所有“部分和”呢？

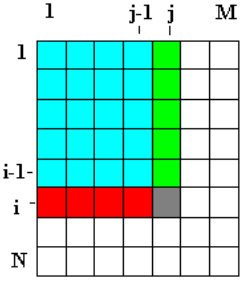

图 2-16 二维最大值示意图

观察图 2-16 不难看出，在更小“部分和”的基础上，也能以 O（1）时间得到新的“部分和”。图 2-16 中 PS[i][j] = PS[i−1][j] + PS[i][j−1]−PS[i−1][j−1] + B[i][j]，其中 B[i][j]为矩阵中第 i 行第 j 列的元素（下标从 1 开始）。因此，O（N * M）的时间就足够预处理并得到所有部分和：

```
for(i = 0; i <=n; i++)
     PS[i][0] = 0;      // 边界值
for(j = 0; j <= M; j++)
     PS[0][j] = 0;      // 边界值
for(i = 1; i <= n; i++)
     for(j = 1; j <= M; j++)
          PS[i][j] = PS[i - 1][j] + PS[i][j - 1] - PS[i - 1][j - 1] + B[i][j];
```

综上所述，我们得到了一个时间复杂度为 O（N^2*M^2）的解法。

解法二

是否还可以找到更快的方法呢？前面我们发现一维的解答可以线性完成。如果我们能把问题从二维转化为一维，或许可以再改进一下。

假设已经确定了矩形区域的上下边界，比如知道矩形区域的上下边界分别是第 a 行和第 c 行，现在要确定左右边界。

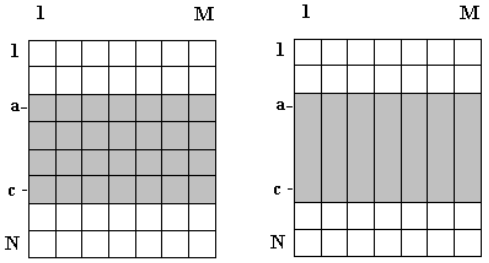

图 2-17 二维示意图

如图 2-17 所示，其实这个问题就是一维的，可以把每一列中第 a 行和第 c 行之间的元素看成一个整体。即求数组（BC[1], …, BC[M]）中和最大的一段，其中 BC[i]=B[a][i]+…+B[c][i]。

这样，我们枚举矩形上下边界，然后再用一维情况下的方法确定左右边界，就可以得到二维问题的解。新方法的时间复杂度为 O（N^2 * M）。

代码清单 2-30

```
// @parameters
// A, 二维数组
```

```
// n, 行数
// m, 列数
int MaxSum(int* A, int n, int m)
{
    maximum = -INF;
    for(a = 1; a <= n; a++)
        for(c = a; c <= n; c++)
        {
            Start = BC(a, c, m);
            All = BC(a, c, m);
            for(i = m-1; i >= 1; i--)
            {
                if(Start < 0)
                    Start = 0;
                Start += BC(a, c, i);
                if(Start > All)
                    All = Start;
            }
            if(All > maximum)
                maximum = All;
        }
    return maximum;
}
```

BC（*a*, *c*, *i*），表示在第 *a* 行和第 *c* 行之间的第 *i* 列的所有元素的和，显然可以通过“部分和”在 *O*（1）时间内计算出来，它等于 PS[*c*][*i*]−PS[*a*−1][*i*]−PS[*c*][*i*−1]+PS[*a*−1] [*i*−1]。

当然，也可以枚举左右边界，再用一维情况下的方法确定上下边界，它们本质上是一样的。至此，这个问题只需 *O*（*N* * *M* * min（*N*, *M*））的时间就可以解决了。

扩展问题

1. 如果二维数组也是首尾相连，像一条首尾相连的带子，算法会如何改变？

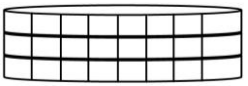

图 2-18

2. 在上面的基础上，如果这个二维数组的上下也相连，就像一个游泳圈（如图 2-18 所示），算法应该怎样修改？
3. 在三维数组 *C*[*i*][*j*] [*k*]（1 <= *i* <=*N*, 1 <= *j* <= *M*, 1 <= *k* <= *L*）中找出一块长方体，使得和最大。
4. 如果再将问题扩展到四维的情况又如何呢？请读者根据以上的分析，想出一个优化解法。

还有一个扩展问题……算了，下次吧。☺

2.16 ★★★ 求数组中最长递增子序列

写一个时间复杂度尽可能低的程序，求一个一维数组（*N* 个元素）中的最长递增子序列的长度。

例如：在序列 1, −1, 2, −3, 4, −5, 6, −7 中，其最长的递增子序列为 1, 2, 4, 6。

分析与解法

根据题目的要求，求一维数组中的最长递增子序列，也就是找一个标号的序列 $b[0], b[1], \cdots, b[m]$（$0 <= b[0]<b[1]<\cdots<b[m] < N$），使得 $array[b[0]] < array[b[1]] < \cdots < array[b[m]]$。

解法一

根据无后效性的定义我们知道，将各阶段按照一定的次序排列好之后，对于某个给定的阶段状态来说，它以前各阶段的状态无法直接影响它未来的决策，而只能间接地通过当前的这个状态来影响。换句话说，每个状态都是过去历史的一个完整总结。

同样地，仍以序列 1, −1, 2,−3, 4, −5, 6, −7 为例，我们在找到 4 之后，并不关心 4 之前的两个值具体是怎样，因为它对找到 6 并没有直接影响。因此，这个问题满足无后效性，可以使用动态规划来解决。

可以通过数字的规律来分析目标串：1, −1, 2, −3, 4, −5, 6, −7。

使用 i 来表示当前遍历的位置：

当 $i = 1$ 时，显然，最长的递增序列为（1），序列长度为 1。

当 $i = 2$ 时，由于$-1 < 1$。因此，必须丢弃第一个值然后重新建立串。当前的递增序列为（−1），长度为 1。

当 $i =3$ 时，由于 $2 > 1$，$2 > -1$。因此，最长的递增序列为（1, 2），（−1, 2），长度为 2。在这里，2 前面是 1 还是−1 对求出后面的递增序列没有直接影响。

依次类推之后，可以得出如下的结论。

假设在目标数组 $array[]$的前 i 个元素中，最长递增子序列的长度为 $LIS[i]$。那么，

$$LIS[i+1] = \max\{1, LIS[k] + 1\}, array[i+1] > array[k], \text{for any } k <= i$$

即如果 $array[i+1]$大于 $array[k]$，那么第 $i+1$ 个元素可以接在 $LIS[k]$长的子序列后面构成一个更长的子序列。与此同时 $array[i+1]$ 本身至少可以构成一个长度为 1 的子序列。

根据上面的分析，就可以得到代码清单 2-31：

代码清单 2-31 C#代码

```
int LIS(int[] array)
{
    int[] LIS = new int[array.Length];
    for(int i = 0; i < array.Length; i++)
    {
        LIS[i] = 1;                              // 初始化默认的长度
        for(int j = 0; j < i; j++)               // 前面最长的序列
        {
            if(array[i] > array[j] && LIS[j] + 1 > LIS[i])
            {
                LIS[i] = LIS[j] + 1;
            }
        }
    }
    return Max(LIS);                             // 取 LIS 的最大值
}
```

这种方法的时间复杂度为 $O(N^2+N) = O(N^2)$。

解法二

显然，$O(N^2)$的算法只是一个比较基本的解法，我们须要想想看是否能够进一步提高效率。在前面的分析中，当考察第 i+1 个元素的时候，我们是不考虑前面 i 个元素的分布情况的。现在我们从另一个角度分析，即当考察第 i+1 个元素的时候考虑前面 i 个元素的情况。

对于前面 i 个元素的任何一个递增子序列，如果这个子序列的最大的元素比 *array*[i+1]小，那么就可以将 *array*[i+1]加在这个子序列后面，构成一个新的递增子序列。

比如当 i=4 的时候，目标序列为：1, −1, 2, −3, 4, −5, 6, −7 最长递增序列为：(1, 2)，(−1, 2)。

那么，只要 4 > 2，就可以把 4 直接增加到前面的子序列中形成一个新的递增子序列。

因此，我们希望找到前 i 个元素中的一个递增子序列，使得这个递增子序列的最大的元素比 *array*[i+1]小，且长度尽量地长。这样将 *array*[i+1]加在该递增子序列后，便可找到以 *array*[i+1]为最大元素的最长递增子序列。

仍然假设在数组的前 i 个元素中，以 *array*[i]为最大元素的最长递增子序列的长度为 *LIS*[i]。

同时，假设：

长度为 1 的递增子序列最大元素的最小值为 MaxV[1]；

长度为 2 的递增子序列最大元素的最小值为 MaxV[2]；

……

长度为 *LIS*[*i*]的递增子序列最大元素的最小值为 MaxV[*LIS*[*i*]]。

假如维护了这些值，那么，在算法中就可以利用相关的信息来减少判断的次数。

具体算法实现如代码清单 2-32 所示。

代码清单 2-32 C#代码

```
int LIS(int[] array)
{
    // 记录数组中的递增序列信息
    int[] MaxV = new int[array.Length + 1];

    MaxV[1] = array[0];              // 数组中的第一值，边界值
    MaxV[0] = Min(array) - 1;        // 数组中最小值，边界值
    int[] LIS = new int[array.Length];

    // 初始化最长递增序列的信息
    for(int i = 0; i < LIS.Length; i++)
    {
        LIS[i] = 1;
    }

    int nMaxLIS = 1;                 // 数组最长递增子序列的长度

    for(int i = 1; i < array.Length; i++)
    {
        // 遍历历史最长递增序列信息
        int j;
        for(j = nMaxLIS; j >= 0; j--)
        {
            if(array[i] > MaxV[j])
            {
                LIS[i] = j + 1;
                break;
            }
        }

        // 如果当前最长序列大于最长递增序列长度，更新最长信息
        if(LIS[i] > nMaxLIS)
        {
            nMaxLIS = LIS[i];
            MaxV[LIS[i]] = array[i];
        }
        else if (MaxV[j] < array[i] && array[i] < MaxV[j + 1])
        {
```

```
                MaxV[j + 1] = array[i];
            }
        }

        return nMaxLIS;
}
```

由于上述解法中的穷举遍历，时间复杂度仍然为 $O(N^2)$。

解法三

解法二的结果似乎仍然不能让人满意。我们是否把递增序列中间的关系全部挖掘出来了呢？再分析一下临时存储下来的最长递增序列信息。

在递增序列中，如果 $i<j$，那么就会有 MaxV[i] < MaxV[j]。如果出现 MaxV[j]<MaxV[i]的情况，则跟定义矛盾，为什么？

因此，根据这样单调递增的关系，可以将上面方法中的穷举部分进行如下修改：

```
for(j = LIS[i-1]; j >= 1; j--)
{
    if(array[i] > MaxV[j])
    {
        LIS[i] = j + 1;
        break;
    }
}
```

如果把上述的查询部分利用二分搜索进行加速，那么就可以把时间复杂度降为 $O(N * \log_2 N)$。

小结

从上面的分析中可以看出我们先提出一个最直接（或者说最简单）的解法，然后从这个最简单解法来看是否有提升的空间，进而一步一步地挖掘解法中的潜力，从而减少解法的时间复杂度。

在实际的面试中，这样的方法同样有效。因为面试者更加看中的是应聘者是否有解决问题的思路，不会因为最后没有达到最优算法而简单地给予否定。应聘者也可以先提出简单的办法，以此投石问路，看看面试者是否会有进一步的提示。

2.17 ★ 数组循环移位

设计一个算法，把一个含有 N 个元素的数组循环右移 K 位，要求时间复杂度为 $O(N)$，且只允许使用两个附加变量。

不合题意的解法如下：

我们先试验简单的办法，可以每次将数组中的元素右移一位，循环 K 次。*abcd*1234→4*abcd*123→34*abcd*12→234*abcd*1→1234*abcd*。伪代码如清单 2-33 所示。

代码清单 2-33

```
RightShift(int* arr, int N, int K)
{
    while(K--)
    {
        int t = arr[N - 1];
        for(int i = N - 1; i > 0; i --)
            arr[i] = arr[i - 1];
        arr[0] = t;
    }
}
```

虽然这个算法可以实现数组的循环右移，但是算法复杂度为 $O(K*N)$，不符合题目的要求，须要继续往下探索。

分析与解法

假如数组为 *abcd*1234，循环右移 4 位的话，我们希望到达的状态是 1234*abcd*。不妨设 *K* 是一个非负的整数，当 *K* 为负整数的时候，右移 *K* 位，相当于左移（–*K*）位。左移和右移在本质上是一样的。

解法一

大家开始可能会有这样的潜在假设，*K*<*N*。事实上，很多时候也的确是这样的。但严格来说，我们不能用这样的“惯性思维”来思考问题。尤其在编程的时候，全面地考虑问题是很重要的，*K* 可能是一个远大于 *N* 的整数，在这个时候，上面的解法是需要改进的。

仔细观察循环右移的特点，不难发现：每个元素右移 *N* 位后都会回到自己的位置上。因此，如果 *K* > *N*，右移 *K*–*N* 之后的数组序列跟右移 *K* 位的结果是一样的。进而可得出一条通用的规律：右移 *K* 位之后的情形，跟右移 *K*′ = *K* % *N* 位之后的情形一样，如代码清单 2-34 所示。

代码清单 2-34

```
RightShift(int* arr, int N, int K)
{
    K %= N;
    while(K--)
    {
        int t = arr[N - 1];
        for(int i = N - 1; i > 0; i --)
            arr[i] = arr[i - 1];
        arr[0] = t;
    }
}
```

可见，增加考虑循环右移的特点之后，算法复杂度降为 $O(N^2)$，这跟 *K* 无关，与题目的要求又接近了一步。但时间复杂度还不够低，接下来让我们继续挖掘循环右移前后，数组之间的关联。

解法二

假设原数组序列为 *abcd*1234，要求变换成的数组序列为 1234*abcd*，即循环右移了 4 位。比较之后，不难看出，其中有两段的顺序是不变的：1234 和 *abcd*，可

把这两段看成两个整体。右移 *K* 位的过程就是把数组的两部分交换一下。变换的过程通过以下步骤完成：

1. 逆序排列 *abcd*：*abcd*1234 → *dcba*1234；
2. 逆序排列 1234：*dcba*1234 → *dcba*4321；
3. 全部逆序：*dcba*4321 → 1234*abcd*。

伪代码可以参考清单 2-35。

代码清单 2-35

```
Reverse(int* arr, int b, int e)
{
    for(; b < e; b++, e--)
    {
        int temp = arr[e];
        arr[e] = arr[b];
        arr[b] = temp;
    }
}

RightShift(int* arr, int N, int k)
{
    K %= N;
    Reverse(arr, 0, N - K - 1);
    Reverse(arr, N - K, N - 1);
    Reverse(arr, 0, N - 1);
}
```

这样，我们就可以在线性时间内实现右移操作了。

2.18 ★★ 数组分割

有一个没有排序、元素个数为 $2n$ 的正整数数组，要求：如何能把这个数组分割为元素个数为 n 的两个数组，并使两个子数组的和最接近？

分析与解法

从题目中可以分析出，题目的本质就是要从 $2n$ 个整数中找出 n 个，使得它们的和尽可能地靠近所有整数之和的一半。

解法一

看到这个题目后，一个直观的想法是：

先将数组的所有元素排序为 $a_1 < a_2 < \cdots < a_{2N}$。

将它们划分为两个子数组 $S_1 = [a_1, a_3, a_5, \cdots, a_{2N-1}]$和 $S_2 = [a_2, a_4, a_6, \cdots, a_{2N}]$。

从 S_1 和 S_2 中找出一对数进行交换，使得 SUM（S_1）和 SUM（S_2）之间的值尽可能的小，直到找不到可对换的。这种想法的缺陷是得到的 S_1 和 S_2 并不是最优的。

解法二

假设 $2n$ 个整数之和为 SUM。从 $2n$ 个整数中找出 n 个元素的和，不管如何接近 SUM/2，同样都会存在大于 SUM/2 或者小于 SUM/2 的情况。当求解这个问题时，大于或小于 SUM/2 没有本质的区别。因此，可以只考虑小于等于 SUM/2 的情况。

比较直观地来说，可以用动态规划来解决这个问题。具体分析如下。

可以把任务分成 $2N$ 步，第 k 步的定义是前 k 个元素中任意 i 个元素的和，所有可能的取值之集合为 S_k（只考虑取值小于等于 SUM/2 的情况）。

然后将第 k 步拆分成两个小步。即首先得到前 k–1 个元素中，任意 i 个元素，总共能有多少种取值，设这个取值集合为 $S_{k-1} = \{v_i\}$。第二步就是令 $S_k = S_{k-1} \cup \{v_i + arr[k]\}$，即可完成第 k 步。

伪代码的实现如清单 2-36。

代码清单 2-36

```
定义：Heap[i]表示存储从 arr 中取 i 个数所能产生的和之集合的堆。
初始化：Heap[0]只有一个元素 0。Heap[i], i〉0 没有元素。
for(k = 1; k <= 2 * n; k++)
{
    i_max = min(k - 1, n - 1);
    for(i = i_max; i >= 0; i--)
    {
        for each v in Heap[i]
            insert(v + arr[k], Heap[i + 1]);
    }
```

```
}
```

这个代码实际执行 insert 次数至多是 2^{N-1} 次，因此，时间复杂度为 O（2^N）。

既然算法的时间复杂度是 N 的指数级，因此在 N 很大时，效率很低。我们不得不考虑设计一种时间复杂度是 N 的多项式函数的方法。考虑的出发点是，是否有另一种拆分第 k 步的方法。

解法三

解法二的拆分方法需要遍历 $S_{k-1}=\{v_i\}$ 的元素，由于 $S_{k-1}=\{v_i\}$ 的元素个数随着 k 的增大而增大，所以导致了解法二的效率低下。能不能设计一个算法使得第 k 步所花费的时间与 k 无关呢？

我们不妨倒过来想，原来是给定 $S_{k-1}=\{v_i\}$，求 S_k。那我们能不能给定 S_k 的可能值 v 和 arr[k]，去寻找 v–arr[k]是否在 $S_{k-1}=\{v_i\}$ 中呢？由于 S_k 可能值的集合的大小与 k 无关，所以这样设计的动态规划算法其第 k 步的时间复杂度与 k 无关。

代码如清单 2-37 所示。

代码清单 2-37

```
定义：isOK[i][v]表示是否可以找到 i 个数，使得它们之和等于 v
初始化 isOK[0][0] = true;
     isOK[i][v] = false(i > 0, v > 0)

for(k = 1; k <= 2 * n; k++)
{
     for(i = 1; (i <= k && i <= n); i++)
          for(v = 1; v <= Sum / 2; v++)
               if(v >= arr[k] && isOK[i - 1][v - arr[k]])
                    isOK[i][v] = true;
}
```

利用如上的算法，时间复杂度将为 O（$N^2 * \mathrm{Sum}$）。

讨论

虽然解法三对于 N 是多项式时间算法，但当 SUM 很大而 N 很小时，比如对于 $arr=\{10^9, 10^8, 10^8+1, 10^9+1\}$，这个解法是很不合适的，所以我们应该根据具体的问题选用合适的方法。

扩展问题

如果数组中有负数，怎么办？

2.19 ★★ 区间重合判断

给定一个源区间[x, y]（$y \geq x$）和 N 个无序的目标区间$[x_1, y_1][x_2, y_2][x_3, y_3] \cdots [x_n, y_n]$，判断源区间[x, y]是不是在目标区间内（也即$[x, y] \in \bigcup_{i=1}^{n}[x_i, y_i]$是否成立）？

例如：给定源区间[1, 6]和一组无序的目标区间[2, 3][1, 2][3, 9]，即可认为区间[1, 6]在区间[2, 3][1, 2][3, 9]内（因为目标区间实际上是[1, 9]）。

分析与解法

解法一

问题的本质在于对目标区间的处理。一个比较直接的思路即将源区间[x, y]（$y \geqslant x$）和 N 个无序的目标区间[x_1, y_1][x_2, y_2][x_3, y_3]…[x_n, y_n]逐个投影到坐标轴上，只考察源区间未被覆盖的部分。如果所有的目标区间全部投影完毕，仍然有源区间没有被覆盖，那么源区间就不在目标区间之内。

仍以 [1, 6]和[2, 3][1, 2][3, 9]为例，考察[1, 6]是否在[2, 3][1, 2][3, 9]内：

源区间为[1, 6]，那么最初未被覆盖的部分为{[1, 6]}（如图 2-19 所示），将按顺序考察目标区间[2, 3][1, 2][3, 9]。

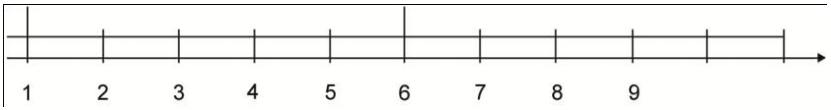

图 2-19

将目标区间[2, 3]投影到坐标轴，那么未被覆盖的部分{[1, 6]}将变为{[1, 2], [3, 6]}（如图 2-20 所示）。

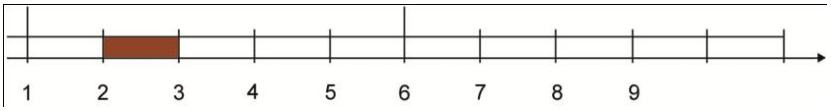

图 2-20

将目标区间[1, 2]投影到坐标轴，那么未被覆盖的部分{[1, 2], [3, 6]}将变为{[3, 6]}（如图 2-21 所示）。

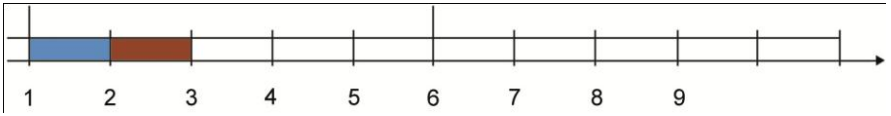

图 2-21

将目标区间[3, 9]投影到坐标轴，那么未被覆盖的部分{[3, 6]}将变为{ø}，即可说明[1, 6]是在[2, 3][1, 2][3, 9]内（如图 2-22 所示）。

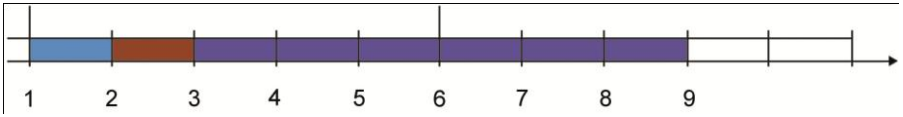

图 2-22

由以上步骤可看出，每次操作，尚未被覆盖的区间数组大小最多增加 1（当然可能减少），而每投影一个新的目标区间，计算有哪些源区间数组被覆盖需要 $O(\log_2 N)$ 的时间复杂度，但是更新尚未被覆盖的区间数组需要 $O(N)$ 的时间复杂度，所以总的时间复杂度为 $O(N^2)$。

这个解法的时间复杂度高，而且如果要对 k 组源区间进行查询，那么时间复杂度会增大单次查询的 k 倍。有没有更好的解法，使得单次查询的时间复杂度降低，并且使 k 次查询的时间复杂度小于单次查询的时间复杂度的 $1/k$ 倍呢？

解法二

一种值得尝试并已经在本书中多次运用的思路是，对现有的数组进行一些预处理（如合并、排序等），将无序的目标区间合并成几个有序的区间，这样就可以进行区间的比较。

因此，问题就变成了如何将这些无序的数组转化为一个目标区间。

首先可以做一次数据初始化的工作。由于目标区间数组是无序的，因此可以对其进行合并操作，使其变得有序。

即先将目标区间数组按 X 轴坐标从小到大排序（排序时可采用快速排序等），如 [2, 3][1, 2] [3, 9] → [1, 2] [2, 3] [3,9]；接着扫描排序后的目标区间数组，将这些区间合并成若干个互不相交的区间，如[1, 2] [2, 3] [3, 9] → [1, 9]。

然后在数据初始化的基础上，运用二分查找（为什么？）来判定源区间[x, y]是否被合并后的这些互不相交的区间中的某一个包含。如[1, 6]被[1, 9]包含，则可说明[1, 6]在[2, 3][1, 2][3, 9]内。

这种思路相对简单，时间复杂度计算如下：

排序的时间复杂度：$O(N * \log_2 N)$（N 为目标区间的个数）；

合并的时间复杂度：$O(N)$；

单次查找的时间复杂度：$\log_2 N$；

所以总的时间复杂度为 $O(N*\log_2 N)+O(N)+O(k*\log_2 N)=O(N*\log_2 N+k*\log_2 N)$，$k$ 为查询的次数，合并目标区间数组的初始化数据操作只须要进行一次。

这样不仅单次查询的时间复杂度降低了，而且对于 $k>>N$ 的情况，处理起来也会方便很多。

总结

解法一采用利用目标区间来分割源区间的方法，会增加存储空间；解法二采用合并的方法，既简单又节省了空间。

扩展问题

如何处理二维空间的覆盖问题？例如在 Windows 桌面上有若干窗口，如何判断某一窗口是否完全被其他窗口覆盖？

2.20 ★★ 程序理解和时间分析

很多同学自己会写不少程序，但是往往看不懂别人写的程序。碰到程序需要理解时，都到电脑上去试验，或者用单步跟踪的办法来调试。如果程序运行的时间很长，那我们要等电脑运行几天几夜么？用人脑行不行呢？在面试的时候，面试者也会考一考应聘者对程序的理解能力，下面就是一个这样的题目。

不用电脑的帮助，回答下面的问题（如代码清单 2-38 所示）。

代码清单 2-38 C#代码

```
using System;
using System.Collections.Generic;
using System.Text;

namespace FindTheNumber
{
    class Program
    {
        static void Main(string[] args)
        {
            int [] rg =
            {2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,
            18,19,20,21,22,23,24,25,26,27,28,29,30,31};

            for(Int64 i = 1; i < Int64.MaxValue; i++)
            {
                int hit = 0;
                int hit1 = -1;
                int hit2 = -1;
                for (int j = 0; (j < rg.Length) && (hit <= 2); j++)
                {
                    if((i % rg[j]) != 0)
                    {
                        hit++;
                        if(hit == 1)
                        {
                            hit1 = j;
                        }
                        else if (hit == 2)
                        {
                            hit2 = j;
                        }
                        else
                            break;
```

```
                    }
                }

                if((hit == 2) && (hit1 + 1 == hit2))
                {
                    Console.WriteLine("found {0}", i);
                }

            }
        }
    }
}
```

问题 1：这个程序要找的是符合什么条件的数？

问题 2：这样的数存在么？符合这一条件的最小数是什么？

问题 3：在电脑上运行这一程序，你估计多长时间才能输出第一个结果？时间精确到分钟（电脑：单核 CPU 2.0G Hz，内存和硬盘等资源充足）。

这道题目没有分析，也没有答案。读者得靠自己的力量来搞定。

2.21 ★★ 只考加法的面试题

看了这么多题目，有人不禁会想，这些题目都太难了！有没有容易的？这里有一题，只用到加法，大家别嫌题目简单，不妨试试看。

我们知道：

1+2 = 3；
4+5 = 9；
2+3+4 = 9。

等式的左边都是两个以上连续的自然数相加，那么是不是所有的整数都可以写成这样的形式呢？稍微考虑一下，我们发现，4、8 等数并不能写成这样的形式。

问题 1：写一个程序，对于一个 64 位正整数，输出它所有可能的连续自然数（两个以上）之和的算式[3]。

问题 2：大家在测试上面程序的过程中，肯定会注意到有一些数字不能表达为一系列连续的自然之和，例如 32 好像就找不到。那么，这样的数字有什么规律呢？能否证明你的结论？

问题 3：在 64 位正整数范围内，子序列数目最多的数是哪一个？这个问题要用程序蛮力搜索，恐怕要运行很长时间，能否用数学知识推导出来？

[3] 当然，在写这个程序的时候，可以用各种运算，不限于加法。