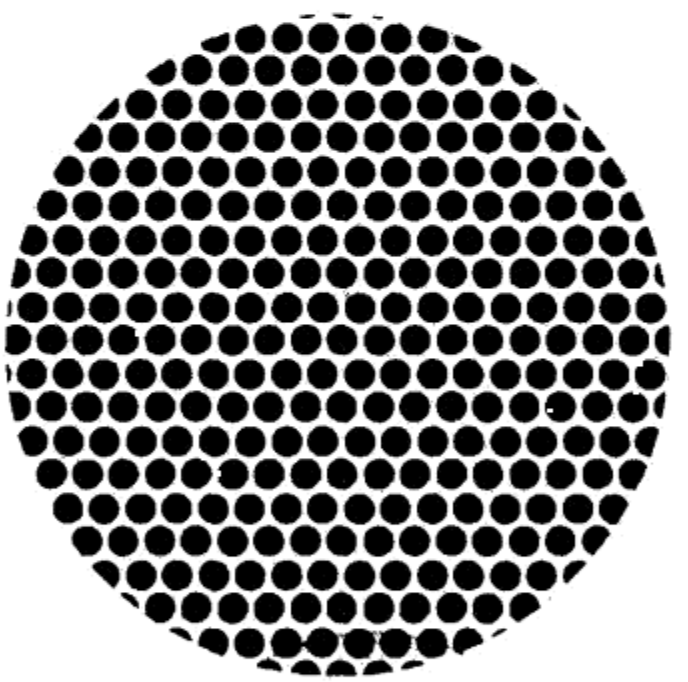
从无有 到无穷

算法之道

Ex Nihilo:Algorithmica Logos

Second Edition

邹恒明 著



机械工业出版社
China Machine Press

本书追求的目标是算法背后的逻辑，是一本启示书，而不是一本包罗万象的算法大全。因此，本书甄选了那些最能展现算法思想、战略和精华，并能够有效训练算法思维的内容。本书将算法的讨论分为五篇：算法基础篇、算法设计篇、算法分析篇、经典算法篇、难解与无解篇。每篇分别讨论算法的一个方面：基础、设计、分析、经典和难解问题。第 2 版还对进程调度问题、跳转表问题、概率分析应用、遗传算法等方面进行了论述。

本书既可以作为大学本科或研究生的算法教材或参考书，也可以作为对算法有兴趣的读者提升认知深度的读物。

图书在版编目（CIP）数据

算法之道 / 邹恒明著. —2 版. —北京：机械工业出版社，2012.1

ISBN 978-7-111-37050-5

Ⅰ. 算…　Ⅱ. 邹…　Ⅲ. 电子计算机-算法理论　Ⅳ. TP301.6

中国版本图书馆 CIP 数据核字（2012）第 002416 号

机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码　100037）

责任编辑：盛思源

北京京北印刷有限公司印刷

2012 年 4 月第 2 版第 1 次印刷

186mm×240mm · 21.5 印张

标准书号：ISBN 978-7-111-37050-5

定价：59.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88378991；88361066

购书热线：（010）68326294；88379649；68995259

投稿热线：（010）88379604

读者信箱：hzjsj@hzbook.com

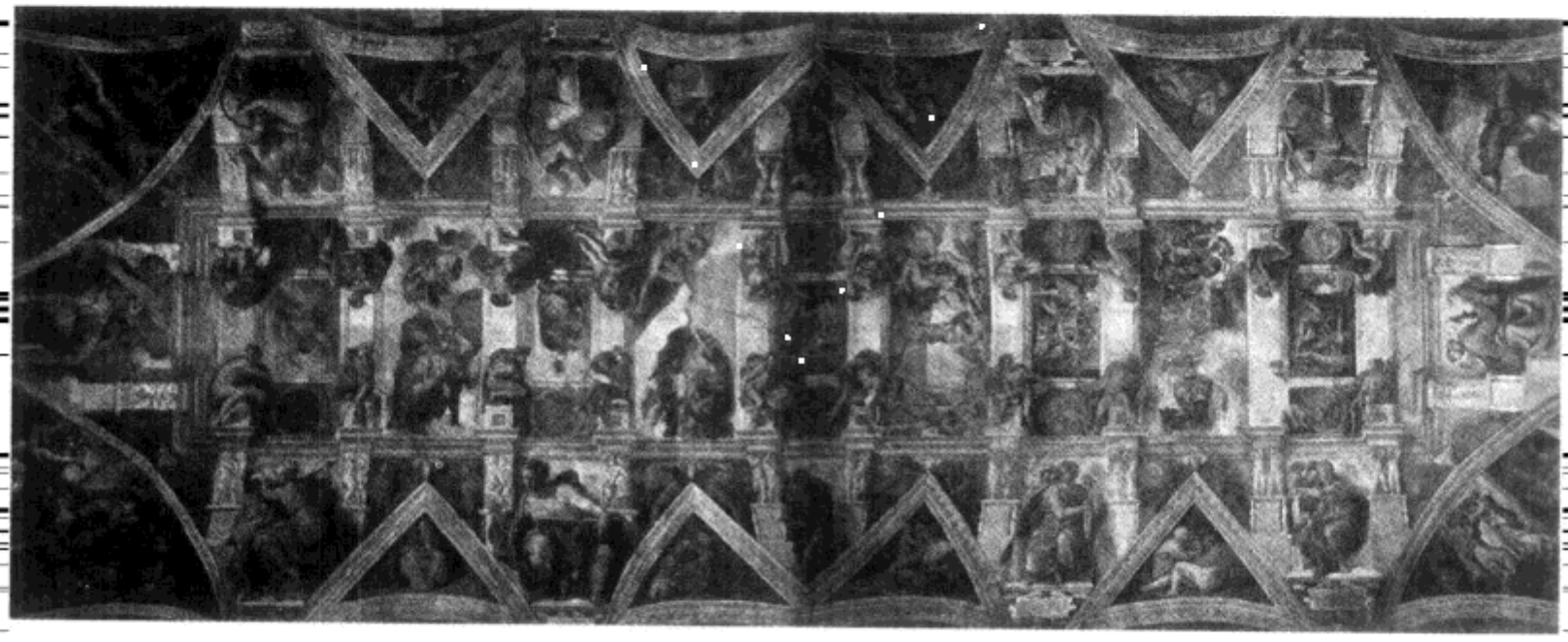
求于至简，归于永恒

In Pursuit of Absolute Simplicity

谨以此书献给夫人蔷蔷，女儿雨洁、雨蓉、雨恒、雨宜。

To my wife Lily and daughter Charissa, Elizabeth, Grace, Ida.

前　言



对于神创论者来说，这是不可怀疑的事实。但对于进化论者来说，6 天创造一切根本就不可能。

作为一本算法书，我们当然不打算加入神创论和进化论者的永无休止的争论当中去。我们关心的是这么一个问题：圣经上为什么给出的是 6 天，而不是其他的时间长度。不管是神创论者还是进化论者，弄清楚 6 这个数字的来历很可能会对己方的观点有所帮助。在这 6 天里，神将他的创作方程式重复了 6 次，每天 1 次。对于全能的神来说，他完全可以在 1 天、1 秒或者任何他所愿意的时间长度里创造天地万物，但却为什么是不多不少的 6 天呢？而不管圣经上的“1 天”是多长，这个问题都是值得讨论的。

我们知道，任何一个自然数的约数中都有 1 和它本身，而所有小于它本身的因数叫做这个自然数的真约数。例如，6 的所有真约数是 1、2、3；数字 8 的真约数是 1、2、4。如果一个数的真约数之和等于这个自然数本身，则这个自然数就称为完全数，或者完美数。例如，6=1 + 2 + 3，因此 6 是完美数；而 8≠1+2+4，因此 8 不是完美数。因此，神 6 天创造世界，暗示着该创造是完美的！

以完美数来昭示创造的完美，似乎合情合理。但问题是，完美数只有 6 这一个数吗？如果不是，为什么不使用其他的完美数呢？答案是，完美数虽然不只有 6 这一个，但确实数量稀少。一直到现在（2009 年 6 月），数学家们探索了 2 600 年，并且现代数学家们还借助了超级计算机的帮助，但也仅仅找到了 47 个完美数。其中第 1 个完美数是 6，接下来的 4 个完美数分别是：28、496、8 128、33 550 336。而第 47 个完美数有 25 956 377 个数位（注意，是数位，不是数值），它的数值为：$2^{43\,112\,608} \times (2^{43\,112\,609} - 1)$。

完美数的稀少昭示着达到完美的难度，而神选择 6 天来创造天地万有也许是因为 6 是最小的完美数，即创造天地万有对于神来说是轻而易举的一件事情……

完美与算法

完美数由于其各种神秘属性（真约数之和等于自身只是其中的一种性质）而受到了特殊的关注。但到底哪些数是完美数则不是一件容易判断的事情。显然，按照完美数的定义，判断一个数是否是完美数的不二法则是找出它的所有真约数，然后求和看看其是否等于自身。而这种方法效率太过低下，因为这意味着因式分解，而这是十分困难的（本书后面将会讨论到这个问题）。

如果判断一个数是否是完美数就已经非常困难，那么要找出所有的完美数则更是一个难上加难的任务。因为这就意味着将所有的数进行上面描述的判断验证：因式分解。这似乎是人类不可能完成的任务。即使用世界上超快的计算机来进行计算，情况也不会有任何数量级的改善。

显然，我们需要新的解决方案，而不是发明或使用新的计算工具！研究这样的解决方案就可以归结到算法的范畴里，因为如何高效地解决问题正是算法要研究的核心课题。

有意思的是，判断和搜索完美数是算法的研究范畴，而算法本身的追求却也是“完美”（见图 2）。

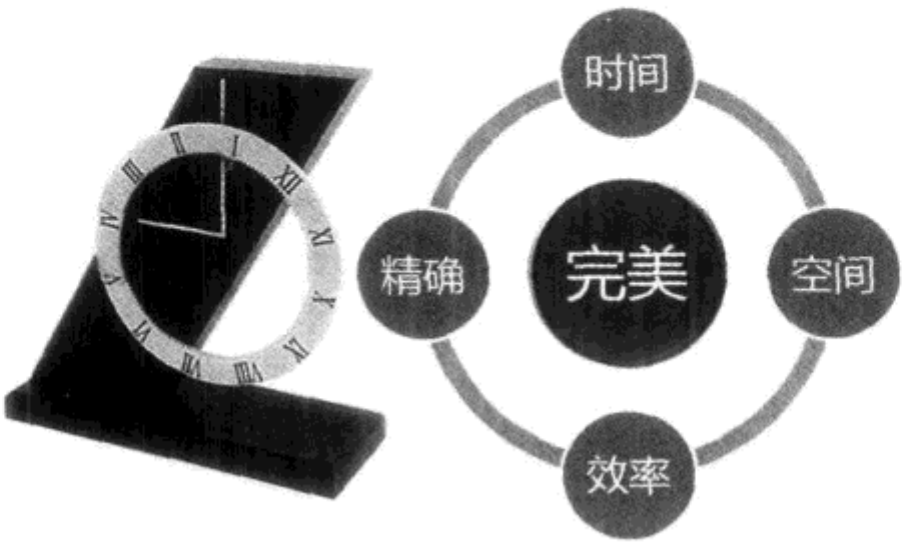


图 2 算法所追求的理想就是“完美”

算法无处不在

如果你觉得算法只是用来研究解决找出完美数之类的“漫无边际的问题”，那就大错特错了。

也许算法这个名词听上去很抽象，让人联想不到任何具体的物体。也许你会觉得算法与自己的生活并无太多关系，它只不过存在于那些闲得无聊的数学家或计算机专业人士的脑海中。

但事实真是这样吗？当然不是。如果我们告诉你算法就是解决各种问题的方法，你就不会觉得它太抽象，与生活无关了吧。事实是，算法无处不在。每个人每天都在使用不同的算法来活出自己的人生，比如你去食堂买饭会选择一个较短的队列，而有人则可能选择一个推进速度更快的队列。每天起床后，你可能先读一会儿书，再去吃早饭；另外一个人则可能先去吃早饭，然后再看书。所有这些行为都是算法或算法一部分的体现。也许运行这些算法并不在你的思想意识里，也许你并不知道算法在帮助自己的生活，但它确实存在。这些算法也许没有经过精心设计，没有经过仔细分析，但它还是算法！

2009 年 7 月 23 日下午，我在游览云南省大理市的蝴蝶泉时由于泉水边的石头很滑，在用泉水洗手时（导游金花说用该泉水洗手会带来好运）不慎滑落到蝴蝶泉水（见图 3）里面全身湿透。（据说一天至多只会有一个人滑落到泉里，可见我的运气不错！看来“蝴蝶泉边好梳妆”的歌词也许应该改为“蝴蝶泉里好冲凉”。）泉水冰冷透凉，而大理的气温又低。这样，我就面临一个是否更换全身衣服的选择。问题是，旅游团需要马上赶去登游船游览洱海风光，而若找地方或者回旅店换衣服就将赶不上游船。

如何处理这件事情就是一个算法问题：是先上游船再在船上找地方换衣服，还是找个地方换衣服而放弃游览苍山洱海。显然不同的算法有着不同的收益和代价。如果能够在游船上找到合适的地方更换衣服，则采用先上游船再换衣服的算法为佳；否则就是放弃游览的算法更好，因为如果冻病了显然就不划算了。最后，我选择了在游船上更换衣服的算法：在游船上找到了一个贵宾室更衣。

图 3　在蝴蝶泉水下洗个手也会涉及算法

算法由问题驱动

算法的发现总是由相关的问题驱动的。拿排序来说，因为生活中到处都充满次序，每个人都要接受自己在某个次序里的位置。比如，各种排名、评优、民意调查等，最后的结果都体现为一个次序！看来，“没有次序无以成方圆”并不是空穴来风！而谈到排序用的方法，人们很自然地想到了插入法。因为这种朴素的算法和人的思维方式非常类似：它就是人们打牌时整理手中扑克牌的算法。

但是随着数据量的增多，插入排序的效率缺陷迅速变为人们无法容忍的缺点。于是人们发明了归并排序、堆排序、快速排序等，这些排序的方法大大改善了速度，但是人们却并不满足于此，因此又发明了效率更高的线性排序。表 1 给出的是各种排序算法平均情况下的效率比较：最上面一行的数字代表输入的规模，如 10 表示一共有 10 个数据项，1M 表示一共有 100 万个数据项。其他格子里面的数据为相应算法在相应输入规模下完成排序所需要的时间，单位为毫秒。所有输入数据为随机产生。

表 1　部分排序算法的时间效率比较　（单位：毫秒）

排 序 算 法	10	100	1k	10k	100k	1M
冒泡排序	0.000 276	0.005 643	0.545	61	8 174	549 432
选择排序	0.000 237	0.006 438	0.488	47	4 717	478 694
插入排序	0.000 258	0.008 619	0.764	56	5 145	515 621
希尔排序/增量 3	0.000 522	0.003 372	0.036	0.518	4.152	61
堆排序	0.000 45	0.002 991	0.041	0.531	6.506	79

（续）

排序算法	10	100	1k	10k	100k	1M
归并排序	0.000 723	0.006 225	0.066	0.561	5.48	70
快速排序	0.000 291	0.003 051	0.030	0.311	3.634	39
基数排序/进制 100	0.005 181	0.021	0.165	1.65	11.428	117
基数排序/进制 1 000	0.016 134	0.026	0.139	1.264	8.394	89

注：1. 算法运行环境为 Intel 酷睿 2 双核 E8400，3.0GHz, Windows 7x64.

2. 本表数据由作者所授“数据结构”课的胡嘉斌同学测试所得.

一个个新的算法都是为了解决前面算法遗留的问题而产生的。从表 1 里的数字可以看出，一般来说，随着新的算法出现，排序效率在不断提高。不过，虽然每个算法似乎解决了前面算法的遗留问题，但新的问题也会被有意或无意地引入。例如，线性排序虽然将排序的时间复杂性降低到线性级，但各种前提条件极大地限制了其应用范围。也许这就是算法永远也不能或不会停止发展的一个原因吧。

算法是计算机的灵魂

因为人不是全能的，一个时刻只能做一件事情，所以做事情就要有一个步骤。由于算法要满足人的这种特性，因此它通常表示为一个做事情的行为序列。因此，从一般意义上说，算法就是求解问题的步骤。由于计算机的计算操作完全是一步一步进行，因此算法的上述性质用于计算机是再合适不过了，可以说算法弥漫在计算机的一切行为上。如果说操作系统是计算机的心智，那么算法就是计算机的灵魂。

理解灵魂当然不是一件容易的事情，由于它高度抽象与简洁，许多学生都望而却步。先看一个纸牌魔术（见图 4）:

1）任选一位观众将一副扑克牌充分洗好。

2）背对观众，请观众随机抽出一张牌，记住牌面，然后将这张牌放回整副牌的最上面。

3）接过牌后，洗牌几次。洗牌的时候保持最上面一张牌不动。

4）对观众说：“我来教你魔法，只要吹一口气，就能把刚才你抽的牌吹到任意位置上”。

5）请观众说出一个数字，比如说 10，然后一边吹气，一边想着刚才说的数字 10。

6）在吹完气后，请观众一张一张地将上面的牌取出放在桌上。

7）到第 10 张时，将牌翻开，发现并不是其原来抽的牌。

8）接回整副牌，并把上一个步骤里取出堆放在桌上的牌收起，仍放在整副牌的最上面。

9）然后洗牌几次，洗牌的时候保持上面放回来的那堆牌不动。

10）从观众手上拿回刚才翻开的那张牌，插入最上面 9 个位置中的任意一个。

11）对观众说：“你刚才不是在想着那个数字的时候吹的气，而是在吹气的时候想着那

个数字，而这是完全不同的两回事。我现在演示如何吹气。”对着牌吹一口气。

12）请观众从上到下数牌，到第 10 张时翻开。

13）这张翻开的牌就是观众一开始抽的那张牌。

读者看明白了上面的这个魔术了吗？这里面隐藏着一个算法。如果看懂了就可以在朋友面前一显身手了。当然，如果没有看懂也没有什么关系。算法本来就不是轻易让人看懂的嘛。

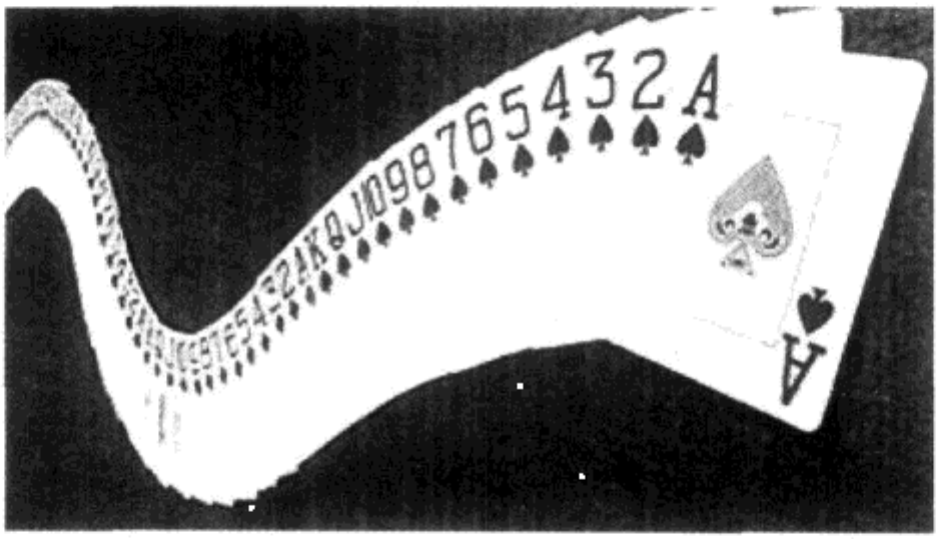


图 4　算法无处不在，就连扑克魔术都有其背后支持的算法

对于一些吹毛求疵的人来说，也许会说这个纸牌魔术不是算法。至少这与我们研究算法的人打交道的常见算法不太一样。这没有什么关系，来看下面的一段伪代码：

```
PARTITION(A, p, q)        //A是一个实数数组，p、q是该数组的上下限
x=A[p];                   //A[p]被选中
i=p;
for( j=p+ 1; j <=q; j++) {
     if(A[ j] <=x) {
          i=i+ 1;
          temp = A[i];        //以下三行交换A[i] 和A[j]的内容
          A[i] =A[j];
          A[ j ]=temp;
     }
}
temp = A[i];                  //以下三行交换A[i] 和A[p]的内容
A[i] =A[p];
A[p]=temp;
return i;
```

读者能看出来这个伪代码程序片段完成的是什么功能吗？

要分析一个算法，似乎就更难了。读者能看出下面的 C 程序片段里面“laugh++”语句执行了多少次吗？

```
for (i=1; i<=n; i*=2)
```

```
                for (j=1; j<=i; j++)
        laugh++;
```

如果这些问题读者都能回答，那么恭喜你。看来算法分析对于读者来说将是件很容易的事情，不过可能也不一定。如果你回答不出这些问题，不用担心，因为回答诸如此类的问题就是本书的目的。当然了，本书回答的远不止这么几个简单问题，而是会阐述更重要的算法精髓：算法思想、战略和分析！

本书内容安排

本书追求的目标是算法背后的逻辑。因此，它不可能是一本包罗万象的算法大全，而是一本启示书。因此，本书甄选了那些最能够展现算法思想、战略和精华，并能够有效训练算法思维的内容。本书的选材遵循的规则是：书中选取的每个算法都在某个方面具有独特性，能够彰显算法的精髓。

本书将算法的讨论分为五篇：算法基础篇、算法设计篇、算法分析篇、经典算法篇、难解与无解篇。每篇分别讨论算法的一大方面：基础、设计、分析、经典和难解问题。如图 5 所示。

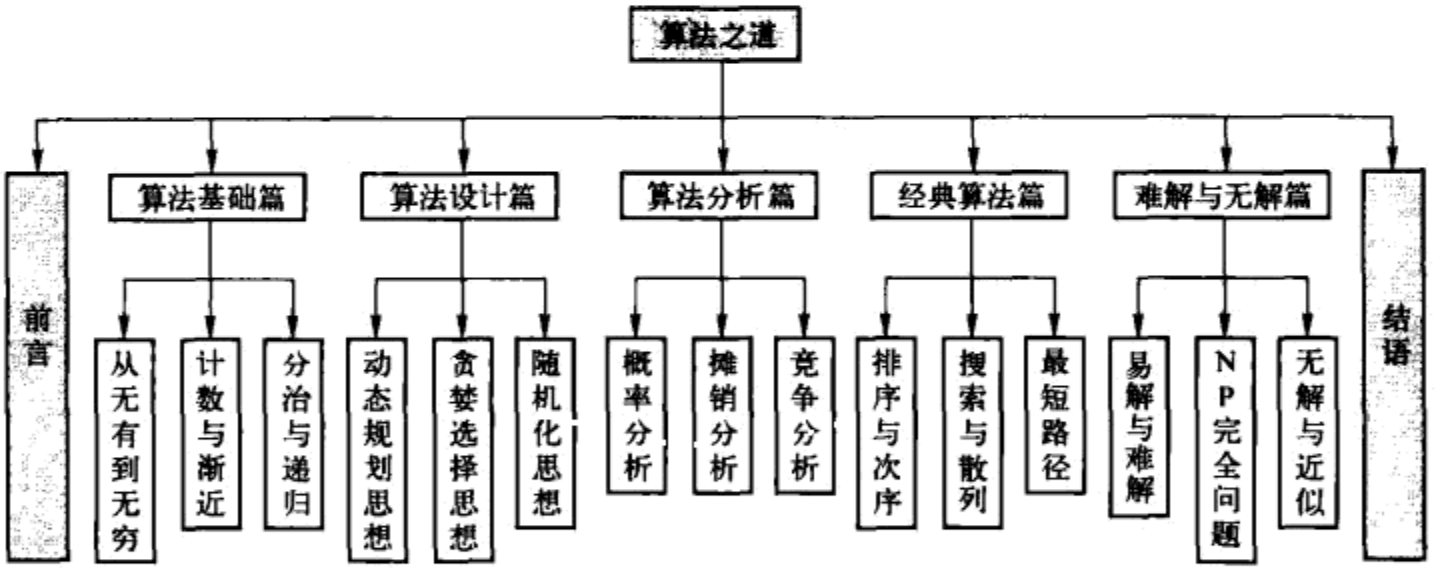


图 5　本书内容框架

本书第一篇是算法基础篇，讨论基本的算法设计思想与算法分析方法和手段，包括第 1~3 章。第 1 章讨论从无有到无穷、什么是算法、算法的表示、算法之魂、算法与计算机的关系、算法的范畴和为什么学习算法。第 2 章讨论算法的正确性、时空效率和时空特性分析、计数分析方法、算法设计、渐近表示与分析。第 3 章阐述算法设计的最基本战略：分治与递归。具体内容包括分而治之为上策、分治策略中的递归、求解递归表达式、乘法及乘方运算、矩阵乘法、斐波那契数的计算、VLSI 布线和多项式乘法。

第二篇为算法设计篇，讨论算法中常见且重要的战略或思想，包括第 4~6 章。第 4 章讨论什么是动态规划、流水线问题、最长公共子序列问题、最优二叉搜索树问题、记忆递归

法、最优子结构、重叠子问题、动态规划与静态规划之间的关系。第 5 章介绍人在生活中潜意识地遵守的一种行为：贪婪。具体内容包括什么是贪婪、贪婪选择属性、背包问题、教室规划（排课）问题、最小生成树问题、霍夫曼编码问题、标准分治、动态规划和贪婪策略的比较。第 6 章讨论人生及算法中无处不在的随机，内容包括为什么要随机化、随机的平方、拉斯维加斯算法、蒙特卡罗算法、素性测试、矩阵乘积验证器、随机化最小生成树算法。

第三篇为算法分析篇，主要介绍计数分析之上的算法分析的另外三种重要手段：概率分析、摊销分析和竞争分析，包括第 7~9 章。第 7 章包括一切都在概率中、什么是概率分析、梦幻情人的代价、概率分析结果的有效性、正确概率分析的保障、梦幻情人的概率、随机排列问题和从无穷到无有。第 8 章讨论什么是摊销分析、摊销分析与数据结构、摊销分析的方法、聚类分析、会计分析、势能分析、动态表扩张及其摊销。第 9 章讨论竞争无处不在、在线算法和离线算法、竞争力、健忘对手和优良对手、线性表更新问题、前置移动算法及其竞争分析、聚类问题及其竞争分析、竞争分析与普通算法分析的比较。

第四篇是经典算法篇，包括第 10~12 章。第 10 章包括插入排序、折半插入排序、归并排序、快速排序、随机化快速排序、排序的下限、线性排序、求最大值、求最小值、求中值、任意次序选择。第 11 章包括搜索问题定义、顺序搜索、折半搜索、常数搜索、散列搜索、散列函数选择、散列冲突的解决、随机化散列、全域散列和完美散列。第 12 章包括单源单点最短路径、单源多点最短路径、多源多点最短路径。具体算法则包括穷举搜索算法、Dijkstra 算法、Bellman-Ford 算法、Floyd-Warshall 算法和 Johnson 算法。

第五篇是难解与无解篇，包括第 13~15 章。第 13 章讨论易解与难解、决策和优化、P 和 NP、确定性与非确定性。第 14 章讨论 NP 完全问题、多项式时间规约、如何证明一个问题 S 是 NP 完全、库克定理、3-SAT 问题、整数规划问题、独立集问题、汉密尔顿回路问题、弱 NP 完全、强 NP 完全和中 NP 完全。第 15 章包括意志的胜利、难解问题、不可解问题、程序终结的判断、难解之题的求解、不可决定问题、难解之题的求解、智能穷举、近似算法、本地搜索、回溯策略、分支限界、贪婪近似策略、启发式搜索策略、模拟退火算法和基因／遗传算法。

本书以《创世纪》的 6 天为起点，寓意算法也有创始的一刻，将人类算法体系中优美而有代表性的内容囊括书中，最后以算法之道的随想作为结尾，构成了逻辑上一气呵成，思想上韵味深长的算法知识体系。

本书的特点

我相信，写书的目的是对读者有所启示（enlightenment），而不是用一大堆的公式或繁琐的推导来烦死或吓倒读者。虽然在一定的时候也会迫不得已地使用复杂的公式，但不必到处都引用复杂繁琐的表述，甚至将简单的东西也以繁琐的式子来表达。复杂的式子或能给部分人带来一丝莫名其妙的快感，但对于大多数读者来说，也许就是“装腔作势罢了，真可憎恶”。基于此种认识，本书将以简洁的方式来表示复杂的概念。

在我读小学的时候，有一天在街上听到一个人吆喝：“快来看，快来看开膛破肚表演！”有人问：“怎么开膛破肚？”卖艺的人说：“用刀将活人的胸膛破开，将里面的内脏拿出来给大家看，然后再缝上。”开膛破肚？这可是难得一见的事情。于是我按照卖艺人的指引进入一个很小的黑屋里，里面已经挤进了一些人，站在屋子四周。屋中间的床架上躺着一个上身赤裸的年轻人，旁边则站着一个手拿尖刀的“屠夫”。“屠夫”先叫每个观众交了一笔钱，然后开始了他的开膛破肚表演。

只见“屠夫”将尖刀举起，对着躺在床架上的年轻人喊道：“你要钱还是要命？”年轻人很坚定地回答：“要钱！”“屠夫”连喊了三遍，得到的回答都是“要钱”。于是“屠夫”将尖刀快速砍下，刺进了年轻人的肚脐，血从尖刀的四周往外渗出。然后“屠夫”对着四周的观众喊道：“你们看这个人要钱不要命，你们给他一些钱吧。”很多人看到这个流血的场面，出于同情或害怕又向外掏钱。

令人遗憾的是，“屠夫”并没有遵守许诺将年轻人的腹部或者胸膛划开，也没有将脏器拿出来给大家看……

这是江湖杂耍的一些小伎俩，但本书对算法进行的“开膛破肚”的分析却是实打实的！这也是本书最显著的特点：**对算法的分析深入到以往没有深入的境界**。本书更关注的是算法后面的逻辑脉络，强调一个算法为什么会出现，又为什么会是现在呈现的样子。通过挖掘算法背后的思维过程，本书淋漓尽致地展现了算法的精妙绝伦。

本书的第二个特点是**结构紧凑**：摒弃臃肿繁琐的内容堆砌，通过逻辑关系将算法各部分内容进行递进演绎，形成一个层次感强、由表入里的有机体。

本书的第三个特点是**全新的角度**：也许讲的是同样的算法、相似的问题,但是本书采纳的是完全不同的角度，这种独特的视角能把我们对算法的理解带到新的高度。

本书的第四个特点是**新颖的结构**：不同一般的章节组织使条理更为清晰，内容上也包含了不少新的概念和理念。

本书的最后一个特点是**写作风格轻松活泼**：以讲故事的形式将概念和算法的精华娓娓道来，由浅入深，非常易于理解和消化。

上述这些特点赋予了本书与一般算法书或其他科技书籍的巨大不同（见图 6）。

图 6 本书的特点

当然，本书也没有走另一个极端：过分强调语言的生动而忽视了严谨性。恰恰相反，本书力求兼顾这两个看似矛盾的方面。在书中我们看不到繁多的数学公式，取而代之的是精确的文字叙述。我认为，这种用严谨的语言代替数学形式化的方法更容易被读者接受。因为读者需要知道的，通常是蕴涵在各种公式或算法伪代码（或程序代码）背后的思想，而正是这些思想促成了所有精巧的算法。

本书几乎没有讲述数据结构的内容，也不会讨论算法的程序设计实现，而是集中精力论述算法本身的特点。虽然有的人认为算法与数据结构和程序设计关系密切，但是毕竟算法可以独立于它们而单独存在。事实上，早在计算机出现之前，算法就已经存在了。从相对的角度来看，算法是抽象的，数据结构与程序设计是具体的（当然从绝对意义上看，数据结构与程序设计也是很抽象的）。而越抽象就越具有普遍意义，也只有在比较抽象的层面上学习算法，才能看透算法的精妙。

当然，本书的讨论也不可能完全脱离数据结构或程序，有时候也会提到它们，有时候甚至直接给出某个算法的具体程序实现片段，但所有对数据结构和程序的论及皆点到为止，以有利于对算法的掌握和体现算法的实现为目的。而数据结构和程序设计的精妙讲解就留给“数据结构”和“程序设计”课程来完成吧。

此外，本书使用的伪代码采用类似于 C/C++的结构，每个算法的表示均以展示算法本身的思路为最高目标，并不对表示的细节进行优化，以使得逻辑清晰，方便绝大多数读者理解。

本书的使用方法

本书既可以作为大学本科或研究生的算法教材或参考书，也可以作为对算法有兴趣的读者提升认知深度的读物。如果作为教材使用，建议课程为 4 个学分，如果学时限制只有 3 个学分，建议将摊销分析（第 8 章）、竞争分析（第 9 章）和无解与近似（第 15 章）这三章内容跳过。建议课堂讲解顺序按书中安排进行，因为本书内容是按照逻辑演绎顺序环环相扣的。按这种顺序讲解条理清晰、逻辑明朗、前后连贯，学生比较容易接受。

对于一般读者，可以将本书作为一种算法思维的修养书来看，按照自己的时间和计划自行安排。或者休闲时翻看此书，斟酌算法，品味精妙，这不也是一件美事吗？

本书隐含 7 个悖论。如果读者能够发现这些悖论并找出答案，那么恭喜你！因为你有一双发现真理的眼睛。不，应该说，你有一颗发现真理的心！如果读者没有发现这些悖论，也没关系，因为能够发现这些悖论的人毕竟不多，更不用说寻找到答案了。而不管发现与否，这些悖论都不会妨碍读者对本书内容的理解。因为如果你发现了这些悖论，它们施加的是正面影响：读者对算法的理解深度会大大增加！而如果没有发现这些悖论，则对读者来说，这些悖论相当于不存在，自然也就无法对读者施加任何正面或负面的影响了。

另外，本书隐含 7 个重要的算法奥秘，但能否察觉就看个人的理解了（见图 7）。

图 7　本书隐含着 7 个悖论和 7 个奥秘

逻辑演绎、生活归纳、趣味交织，入木三分地揭示算法的奥妙；新的角度、新的分析、新的境界，耳目一新地阐述算法的精华。就让我们即刻开始精彩纷呈的算法之旅，Let the Funs Begin!

第五篇 难解与无解篇

PART ONE

第一篇　算法基础篇

PDG

CHAPTER 1

第 1 章　从无有到无穷

在第一类弗里德曼宇宙模型中，第四维——时间，正如空间一样，在范围上是有限的。它如一根具有两个端点或边界的线。因此时间具有终结，而且它也有一个开端。事实上，在宇宙具有我们观测到的物质总量的情形下，由爱因斯坦方程得出的所有解中，都有一个非常重要的特征：在过去某一时刻（大约 137 亿年以前）相邻星系之间的距离必须为零。换言之，整个宇宙被挤压在零尺度的单独一点，就像一个半径为零的球。那时，宇宙的密度和时空曲率都为无限大。它是我们称做大爆炸的时刻。

——摘自史蒂文·霍金《时间简史》

这个零尺度的单独一点被物理学家称做“原点”。它的另一个名字是奇异点（singularity）。但是零尺度是什么意思呢？霍金曾解释过：零尺度就是不占空间。那么不占空间是什么意思呢？也许读者猜出来了：没有（nothing）！即虚无。实际上，物理学家们普遍认为在原点之外没有空间，空间也是大爆炸后的产物。

也就是说，宇宙是从无到有的，用希腊文来说就是 Ex Nihilo（见图 1-1）。

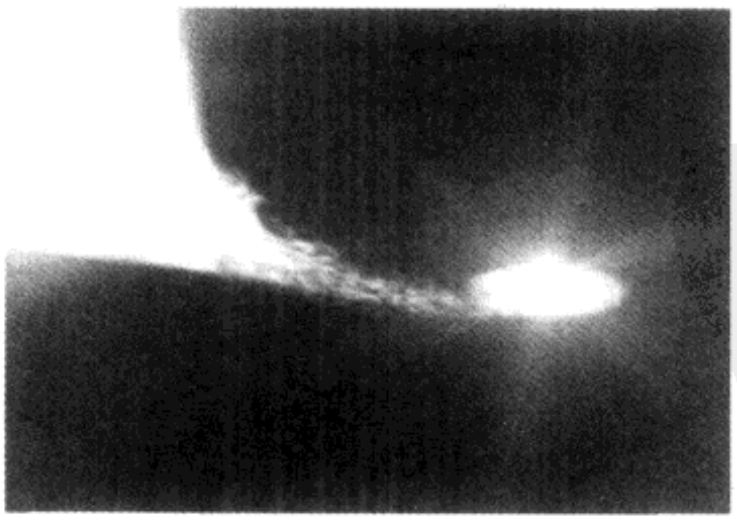


图 1-1　Ex Nihilo：宇宙从无到有的一刹那

整个宇宙从无到有对一般人来说都很难理解，而这个原点是谁或者如何放在那里也是众说纷纭。不过，这不是本书准备要讨论的问题。本书关心的是算法，而算法具有一个与宇宙

起源类似的性质：从无到有。不过这个从无到有却有着非同一般或者说更加丰富的意义，下面将详细分析。

1.1 意念与现实

先看一个例子。给你一个无限容积的罐子和无限个球，球从 1 开始连续编号。

在差 1 分钟到零点时：将标号为 1～10 的 10 个球放进罐子，然后将 10 号球从罐子拿出。

在差 1/2 分钟到零点时：将标号为 11～20 的 10 个球放进罐子，然后将 20 号球从罐子拿出。

在差 1/4 分钟到零点时：将标号为 21～30 的 10 个球放进罐子，然后将 30 号球从罐子拿出。

……

就这样将游戏进行下去。假定放球和取球不占时间，请问，当时钟指向零点时，罐子里还剩多少个球？

这个答案似乎很直接：无限个球！这是因为所有编号不是 $10n$（$n \geq 1$）的球在放进去罐子里后就不会再拿出来；而在零点之前这种放球、取球的次数是无限的。因此，罐子里面的球数在零点时将是无数个。

但是你很确信这个答案吗？

现在来让我们改变拿球的方式，将每次拿 10、20、30、…号球分别变为拿 1、2、3、…号球，即第 x 次拿球，所拿出来的球的编号是 x。结果又会怎样呢？

这个时候，神奇的事情发生了。这个罐子里面的球数将为 0。

我们来看，对于任意一个球，设其编号为 n，则在差 $(1/2)^{n-1}$ 分钟到零点时该球将被取出。也就是说，对于任意球 n，在零点时它都不在罐子里。因此，零点时罐子里球的个数为 0。

对于有些人来说，这个答案似乎不可接受。但又确实找不到驳斥的办法。你能找出来吗？

也许这个答案是合理的，因为拿球顺序的变化使得算法发生了变化，即我们实际上讨论的是两个算法。可仔细一想又觉得不对，因为两个算法都是每次放进 10 个球，拿出 1 个球，即从根本上说，这是两个一样的算法，怎么会有截然相反的结果呢（见图 1-2）？

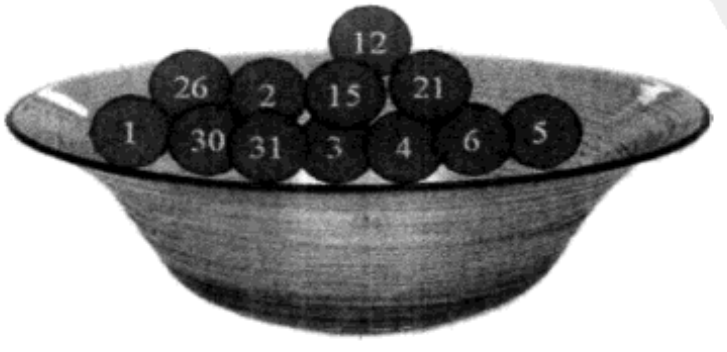


图 1-2 到底剩多少个球？不同的拿球顺序有不同的结果

如果我们再次改变试验中拿球的方式，将拿某个特定标号的球改为取出任意标号的球，即

在差 1 分钟到零点时：将标号为 1～10 的 10 个球放进罐子，然后从罐子任意拿出一球。

在差 1/2 分钟到零点时：将标号为 11～20 的 10 个球放进罐子，然后从罐子任意拿出一球。

在差 1/4 分钟到零点时：将标号为 21～30 的 10 个球放进罐子，然后从罐子任意拿出一球。

……

这种拿球方式又将产生何种结果呢？

答案仍然是无有，即 0（本书将在第 1 章对这个问题进行正面解析）。

太不可思议了吧！这三个本质相同的算法怎么有如此匪夷所思的结果呢？如果非要说这三个算法有什么不同，那就是拿球时的标号不同。

难道是，标号的不同使最后球的数量发生了变化？

没错。就是这个标号对结果产生了深远影响。从某种意义上说，标号是虚的，它只存在于我们的想象中，但确实对现实结果产生了影响，即我们的思维使算法发生了变化。或许从另一个角度来看，这个问题就是：无有就是无穷，无穷就是无有。它们之间也许根本没有什么不同，它们的不同只存在于人们的想象或者意念中。也许这是为什么无穷的符号∞是由两个 0 连接而成的，从左右两面看都是无有，而从中间看则是无穷，如图 1-3 所示。

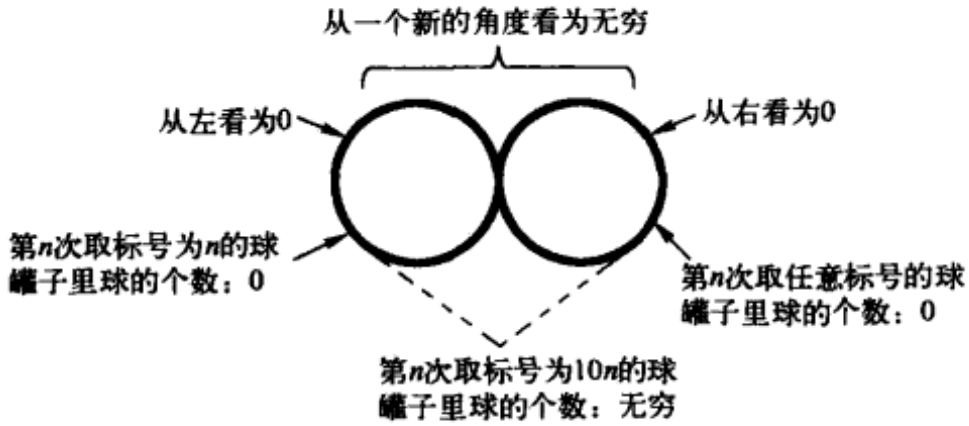


图 1-3　无有和无穷的区别也许只存在于人类的思维中

从这个意义上说，算法是一种思维方式（algorithmic thinking），或者说一种哲学。而本书就是从算法思维的角度出发，阐述算法的灵魂。

1.2　什么是算法

究竟什么是算法呢？顾名思义，算法就是计算的办法或法则。这里的计算指的当然不只是加、减、乘、除等算术运算，而是广义的做任何事情的计算，而办法和法则意味着使用它就可以解决需要的问题。

算法的历史可以追溯到 9 世纪的古波斯。最初它仅表示“阿拉伯数字的运算法则”。后来，它被赋予更一般的含义，即所谓的一组确定的、有效的、有限的解决问题的步骤。这是算法的最初定义，注意，这个定义里面没有包括“正确”。

推动算法传播的是生活在美索不达美亚的 Al Khwarizmi 于 9 世纪一本以阿拉姆语（Aramaic）著述的教科书。该书列举了加、减、乘、除、求平方根和计算圆周率数值的方

法。这些步骤的特点是：**简单、没有歧义、机械、有效和正确——这就是算法**。注意，这个定义加上了“正确”这个词。几百年后，当十进制计数法在欧洲被广泛使用时，“算法”（algorithm）这个单词被人们创造出来以纪念 Al Khwarizmi 先生。

由上面提到的定义可推知，算法作为解决问题的方法，它必须具备以下特点：

- 确定性，即无歧义，能让人照着执行。
- 可行性，算法中的运算都是基本的，理论上能够由人用纸和笔完成。
- 有限性，在**有限输入**下，算法必须能在有限步骤内实现**有限输出**。

此外，算法必须有输出、计算的结果，通常还有至少一个输入量。这是因为算法用以解决的问题的描述均包括输入和输出。例如，排序问题可以描述如下：

输入：

数列：$a_1, a_2, \cdots, a_n$

输出：

排列：$a_1', a_2', \cdots, a_n'$，其中 $a_1' \leqslant a_2' \leqslant \cdots \leqslant a_n'$

举例：

输入序列：　8 2 4 9 3 6

输出序列：　2 3 4 6 8 9

这里需要注意的是，有时候，算法的输入和输出并不是显而易见的。例如，计算圆周率的输入是什么？是圆吗？哪个圆？任意圆？再如，下面这个纸牌游戏的输出又是什么呢？

1）任选一位观众将一副扑克牌充分洗好。

2）背对观众，请观众随机抽出一张牌，记住牌面，然后将这张牌放回到整副牌的最上面。

3）接过牌后，洗牌几次。洗牌的时候保持最上面一张牌不动。

4）对观众说：“我来教你魔法，只要吹一口气，就能把刚才你抽的牌吹到任意位置上。”

5）请观众说出一个数字，比如说 10，然后一边吹气，一边想着刚才说的数字 10。

6）在吹完气后，请观众一张一张地将上面的牌取出放在桌上。

7）到第 10 张时，将牌翻开，发现并不是其原来抽的牌。

8）接回整副牌，并把上一个步骤里取出堆放在桌上的牌收起，仍放在整副牌的最上面。

9）然后洗牌几次，洗牌的时候保持上面放回来的那堆牌不动。

10）从观众手上拿回刚才翻开的那张牌，插入到最上面 9 个位置中的任意一个。

11）对观众说：“你刚才不是在想着那个数字的时候吹的气，而是在吹气的时候想着那个数字，而这是完全不同的两回事。我现在演示如何吹气。”对着牌吹一口气。

12）请观众从上到下数牌，到第 10 张时翻开。

13）这张翻开的牌就是观众一开始抽的那张牌。

也许我们并不能一下子说出这个纸牌游戏的输出，但却是可以看到其背后隐藏着一个确定的算法。从这一点上看，在算法的定义上，我们没有必要吹毛求疵。太过严格的定义并不一定符合实际。事实上，算法的定义可以有多种，从不同角度看，获得的定义各不相同。

- 从哲学角度看：算法是解决一个问题的抽象行为序列。
- 从技术层面上看：算法是一个计算过程，它接受一些输入，并产生某些输出。

- 从抽象层面上看：算法是一个将输入转化为输出的计算步骤序列。
- 从宏观层面上看：算法是解决一个精确定义的计算问题的工具。

到底使用哪一个定义就要看具体情况和个人喜好。选择的原则是这种定义能够帮助我们更好地理解、设计和分析算法。事实上，算法并不一定要“**简单、没有歧义、机械、有效和正确**”，**很多复杂的、有歧义的，甚至是错误的东西也有可能是算法**。在多数时候，我们不在这个定义上纠缠是非，因为这种区分对算法设计与分析并无太大影响。

本书强调的是算法的不变式：必须能够让人一步一步地照着执行。

1.3 算法的表示

算法如何表示呢？由于算法由一系列步骤组成，那么任何一个步骤序列从广义上看，都可以当做是一个算法。例如，下面的步骤就可以看做是一个算法：

1）起床。
2）吃早点。
3）上早自习。
4）上课。
5）吃午饭。
6）上课。
7）吃晚饭。
8）上晚自习。
9）睡觉。

这是用自然语言表示的一个学生每天运行的通用算法。下面是一个将末端递归转换为循环的算法，我们也可用英语来表示它：

1）Copy the function's type signature.
2）Identify any needed 'loop variables' by inspecting the call to the helper function (if it exists).
3）Write initialization code to mirror the call to the helper function.
4）Identify termination condition(s) and return values by copying the base case behavior.
5）Write loop body by copying the inductive step.

除了自然语言，我们也可用计算机语言（即程序设计语言）来表示算法。实际上，由于计算机程序给出的是一个一个步骤的执行序列，因此计算机程序都是算法，或者说都是算法的一种表示。例如，下面的计算机程序片段就表示的是一个计算阶乘的算法：

```
int factorial(int n)
{
if (n==0) return 1;
else return n*factorial(n-1);
```

```
}
```

不过，虽然自然语言和计算机程序设计语言都可用来表示算法，但这些表示都存在缺陷。自然语言容易产生歧义，从而导致对算法的表示和分析错误。另外，用自然语言表示算法时要求细心，稍有不慎，就会叙述错误。例如，在排序时，人们一不留心，就很容易将排好序的数据称做一个递增或递减序列，这种称法是十分不准确的，甚至有可能是错误的。例如，如果一个序列里面有数值相同的数据项，则排好序的结果不能满足递增或递减的属性。这个时候的正确称呼应该是非递增或非递减序列。

如果说用自然语言表示算法太过一般，那么计算机程序设计语言又过于具体，导致不必要的操心和工作量，且理解起来有困难，尤其是对于没有学习过程序设计的人来说。因此，这两种表示对于算法的设计和分析来说都不是特别适合。

我们需要的是一种既不必花太大力气写出来，又具有精确性的表示方法。这种方法就是计算机界最喜欢的伪代码表示法，例如下面的归并排序算法就是以伪代码表示的：

```
MERGE-SORT A[1..n]
1. 如果 n=1，结束排序。
2. 递归对子序列 A[1..[n/2]] 和 A[[n/2]+1..n]归并排序。
3. 合并已排好序的两个子序列。
```

上述表示既精确，又容易理解，也不会陷入机械的程序设计语言的细节泥潭里。

当然了，算法的表示还可以有很多。例如，我们还可以用流程图甚至形式语言描述一个算法。不过，这些描述方式均存在要么琐细、要么难以理解等问题而没有获得广泛使用。

1.4 算法之魂

算法是解决问题的办法或法则。但解决一个问题不一定只有一种办法，不同的办法之间便有了好坏之分。对于解决同一个问题的不同算法，我们如何比较它们的好坏呢？

能够比较的东西当然很多：模块性、正确性、可维护性、功能性、健壮性、友好性、简易性、可扩展性、可靠性等，但这些并不是算法设计与分析中最为关心的问题。因为，这些因素虽然重要，却不是算法本身的独特属性。实际上，它们更加像是人类附加在算法上的外部属性，因为它们通常依赖于使用或实现算法的人员的其他方面素质：理解力、表述力、编程水平、数据结构的运用与设计技巧等。

那么算法关心的是什么呢？或者说算法的核心或灵魂是什么呢？也许读者已经猜出来了：速度。算法分析最为关心的是算法的速度！也就是其解决问题的速度。因为速度往往是区分可行和不可行方案的分水岭。例如，一个让人等上很多年才能运行结束的算法，就是再正确，也不会令我们满意。从实际意义上看，这种正确的算法和不正确并无太大的本质区别。

除此之外，算法之关心速度也许还有一个心理原因：速度能带给人快感（见图 1-4）。

读者有没有飙过车？如果你飙过车，那么你就会知道速度能够给人一种难以名状的快

感。如果没有飙过车，那么世界上第一个飙车的人为我们提供了鲜活的证据。而这个世界上第一个飙车的人恐怕是多数人不曾想到的：第二次世界大战时期的德国元首阿道夫·希特勒。

图 1-4　高效的算法就像飙车一样带给人莫名的快感

1943 年春，在苏联刺骨寒冷、冰天雪地的环境折磨下，德国的战争优势逐渐逆转。不，更加准确地说，是形势急转直下。在不利战况报告一个接一个的情况下，德国元首希特勒的心情也如冬天一般变得阴沉起来，胸中的烦躁就像莫斯科战场上的厚厚白雪，挥之不化。为了排遣胸中的焦虑，希特勒拿出了绝活：飙车。

当然，白天是不能飙车的。因为繁华的柏林街头民众很多，飙车将严重威胁市民生命，而作为元首，“当然对人民的生命爱惜有加”。再说，德国的高层领导集团也不会让希特勒冒这种无谓的风险。于是，希特勒只能在深更半夜起床，坐上自己的奔驰吉普车，命令司机将油门踩到底，且不准放松。汽车在 100 多千米的时速下飞奔，零部件都嘎嘎作响，好像要散架一样。司机紧张得几乎精神崩溃。要知道，20 世纪 40 年代汽车的正常时速为 20～30 千米/小时，且路况也远不如今天的高速公路这样好。因此，那时的飙车对车体是一种极限挑战。

希特勒在高速奔驰的汽车上发泄了烦闷，但司机却由于高度的紧张而导致神经衰弱，不得不被撤换。就这样，希特勒撤换了一个又一个司机，才能够不断享受飙车的体验。希特勒需要的就是两个字：速度。因为这带给了他无限的快感。也许就是这种希特勒唯一还能享受的快感让他的身体又支撑了两年多！

速度也是算法所追求的。实际上，“速度”就是算法之魂。不过，我们追求算法的速度不是像希特勒那样需要发泄个人胸中的焦虑，而是要替人类发泄烦躁！如果一个算法在你的改进下，突然效率提高了成百上千倍，则当你坐在计算机前，看到结果瞬时出现时所获得的快感不亚于希特勒飙车体验的快感！

1.5　如何比较速度

比较算法的好坏就是比较算法的速度。需要注意的是，这里的速度是一个抽象概念，指的是算法计算所需要的步骤，而不是具体的多少小时、多少分钟等。

那么如何计算一个算法运行时所需要的步骤呢？来看一个具体例子。

```
INSERTION-SORT (A, n)                    //A[1..n]为一个数组
for (j =2;i<=n;j++) {
      key=A[j];
      i=j-1;
      while(i>0 and A[i]>key) {
        A[i+1] =A[i];
        i=i-1;
      }
      A[i+1] = key;
}
```

这个算法需要多少个步骤才能运行结束呢？从代码可以看出，外循环是 n-1 次，在每次外循环下，需要执行 3 条赋值语句和一个循环语句。而这个内循环要执行多少次呢？这个一下子看不出来，因为在没有给出具体数据值的情况下，$A[i]$ >key 这个条件能够多少次满足无法判断。不过，这没有关系，我们可以看看这个循环最多执行多少次，这样我们可以获得一个该算法步骤数的上限。从 i 的变化可以看出，这个内循环最多 j-1 次。那么该内循环的总执行次数最多就是 $1+2+\cdots+(n-1)=(n-1)n/2$。

因此，整个 INSERTION-SORT（插入排序）的步骤数最多是：

$$(n-1)\times 3+(n-1)n/2\times 2=n^2+2n-3$$

虽然这个算法我们可以通过蛮力计数获得结果，但这不是我们推荐的办法。因为这种方法在算法较为复杂的时候就非常不好用了。并且，我们算出的还只是一个上限，而不是一个准确的结果。那么在复杂的算法中，如何计算它的步骤呢？如何获得准确或者说平均的结果呢？这就是本书要论述的一个重要主题！

1.6 算法与计算机的关系

计算机作为按照给定指令对数据进行操作的机器，其运行过程是一系列离散的步骤，而这正好符合算法的定义。换个角度看，算法的三个性质适用于计算机是再合适不过了：计算机程序不就是确定的、有效的和有限的吗？事实上，不管我们喜欢不喜欢，计算机的每条指令的执行都是某个算法的体现！只不过这个算法也许复杂，也许简单，也许经过精心设计，也许是信手拈来，甚至只是潜意识思维的一种表示。

由此可见，计算机与算法有着不可分割的关系。可以说，没有算法，就没有计算机。或者说计算机无法独立于算法而存在。从这个层面上看，算法就是计算机的灵魂！就像操作系统就是计算机的心智一样。一个计算机行业的人员如果不了解算法，那就没有真正了解计算机。

但是，算法却不一定要依赖于计算机才能存在。前面已经说过，算法可以是抽象的，用来实现算法的实体可以是人，而不一定要是计算机。事实上，我们在表述、设计和分析算法时，并不希望我们的理解和分析依赖于某个具体的计算机或体系结构。我们更为关注的是抽象层面上的算法效率，因为这种分析才是“永恒”的，不会因计算机的发展而过时。

但是，在多数时候，算法又确实是通过计算机实现，因为很多算法对于人来说过于复杂，计算的工作量太大且常常重复，对于人脑来说实在是难以胜任。因此，我们在设计、分析算法的时候，又经常会考虑其在计算机上实现时的各种隐含意义和外延。

由此可见，离开了算法，计算机就毫无意义；而离开了计算机，算法的实际作用就要大打折扣。从这个意义上说，算法与计算机是一种相互交织，水乳交融的关系。本书在讨论算法的时候总是将计算机的使用考虑在内。毕竟，我们是搞计算机的嘛！

1.7　算法的范畴

算法的范畴依角度的不同而有很大的不同。从广义上看，一切皆算法。我们每天的行为都是一个离散的序列，因此都可以说是在遵循着某个算法，只不过这种遵守不一定是有意识的。但这些并不是本书要讨论的主要问题，对于本书来说，算法的范围主要局限在解决各种科学和工程问题的层次上，而不涉及人文、社会学或经济学领域。

算法课程的范畴一般包括两方面内容：算法设计与算法分析。前者指根据实际问题制定出有效的算法；后者即是对算法的各种性质进行定性或定量分析，从而能够择优选择某种算法。

从更高的层次上看，算法还是一种思维方式。因此，算法课程除了覆盖设计与分析外，还应该训练人的算法思维能力，从而更加清晰地理解这个世界甚至人生。

1.8　为什么学习算法

到目前为止，我们简要论述了算法是什么、算法之魂、算法和计算机的关系以及算法思维。读者应该体会到算法的重要性。但仅仅是因为算法重要就要学习它吗？世界上有很多重要的东西，难道我们都要学吗？即使是计算机专业的学生，不学算法也照样可以编程写软件。那么，我们为什么要学习算法呢？

当然，我们有成千个理由要学，但这里仅给出几个。

首先，算法是计算机的灵魂。前面已经说过，计算机不能独立于算法而存在，或者说独立于算法的计算机其存在价值要大打折扣。一个程序要完成一个任务，其背后肯定要涉及算法的设计。实际上，程序就是算法的实现，或者说程序是算法的外在体现。学好了算法，就能够设计出更加有效的软件，以最有效的方式完成更为复杂的功能。

其次，算法是数学机械化的一部分，能够帮助我们解决复杂的计算问题，其中有的问题就存在于我们的日常生活中。前面讲过，算法无处不在。实际上，人是躲避不了算法的，每天的日常生活都会涉及算法。例如，如何分配自己的时间才能最有效地完成学习或工作任务就会牵扯到算法。没有算法知识的人，分配的时候多半会源于自发、非科学的处理方法，难以达到高效。

再次，算法作为一种思想，能锻炼我们的思维，使思维变得更清晰、更有逻辑。算法是对事物本质的数学抽象，看似深奥，却体现着点点滴滴的朴素思想。虽然真理未必只有一

个，但是当你掌握了其中的一个，你就掌握了全部，这就像是 NP 完全问题一样。因此，学会算法的思想，其意义不仅仅在算法本身，也会对日后的学习生活产生深远的影响。

算法还能帮助人们理解什么是可行的，什么是不可行的。

不过最重要的理由并不是上面给出的那些，而是算法本身真的很有意思，很有趣味。当你真的沉浸到算法里的时候，其速度、其构思都会让你觉得精妙绝伦，有一种不可言喻的美感和快感。Donald Knuth 曾经说过，程序就是蓝色的诗。如果这是真的，那这首诗的灵魂，或者诗魂，就是其背后的算法。难道窥探计算机的灵魂不会令人激动与兴奋吗？

当然，并不是所有的人都会有这样的感觉。算法的那份优雅与精巧虽然吸引人，却也令很多人望而生畏。事实证明，对很多人来说，学习算法是一件很痛苦的事情。不过我希望阅读本书对读者来说是一件舒心之事。只要你将本书从头读到尾，就会体验到不同寻常的乐趣！

思考题

1. 本章放球、拿球的两个方式到底是一个算法还是两个算法？你怎么看？
2. 你为什么要学习算法？因为它是必修课吗？还是别的什么原因？
3. 如果一个自然数的全部真约数之和等于该自然数本身，则该自然数被称为完全数。你能设计一个完全数判断的算法吗？你能估算出它的效率吗？
4. 请列出一些你能想出来的完全数的属性。
5. 你平时的吃饭、穿衣与算法有什么关系吗？
6. 你认为算法必须是“简单、没有歧义、机械、有效和正确”吗？给出你的理由。
7. 算法从不同的层面上看可以有不同的定义，你最欣赏哪一层的定义？
8. 练习本章给出的纸牌游戏，熟练后演示给你的朋友看。
9. 试从算法的角度阐述无有与无穷的关系。
10. 平时与社会上的各色人等的相处与算法有关系吗？请详细说明。
11. 从无有到无穷隐示算法无所不包，另外一种观点认为万物唯数。这二者之间有何关系？

CHAPTER 2

第 2 章　计数与渐近

据说在 200 多年前，在德国乡村的一所小学里，一个变态的老师总是要求学生不停地做整数加法计算。他的嗜好就是让小学生们将一长串整数加起来，自己就可以在旁边休息，看自己的书。这一天又如法炮制，他布置了一道将 1、2、3、…、100 求和的加法作业后，就打开了一本大部头的书自顾自地看了起来。但他刚看了不到一行字，就有一个学生说他算出答案了。老师头也不抬地说，再去算，不料这个学生却站着不动。于是老师变得不耐烦了，说再去做，你的答案肯定是错误的。可学生还是站在老师面前不动。

老师被激怒了，他不相信任何一个小学生能够在几秒钟内将 1+2+3+…+100 计算出来。于是他一把抢过学生的答案，正欲发作时，却发现学生计算的结果是 5 050。老师怔住了。原来，这个学生不是一个一个自然数地往上加，而是将这 100 个数分为 50 对再进行乘法运算：$1+100=101, 2+99=101, 3+98=101$ 等，共 50 对，而结果则是 $50\times101=5\,050$。

用这种简便算法给出这道题正确结果的学生不是别人，就是德国数学家高斯。

有人认为上述故事纯属虚构，但故事是否虚构并不是本书要研究的问题，本书所关心的是算法对于解决问题的重要性。一个问题，如果采用了合适的算法，其解决的速度将大大提高。就像乡村老师布置的级数求和题，如果直接使用逐个数相加的办法，非常耗时。但如果像高斯一样，通过使用很巧妙的计算方法，则在几秒钟内就解决了。这就是算法和它的魅力。

2.1　算法的分析

也许你会感觉到，高斯的计算方法优于那种逐个数加下去的计算方法，但是，你能说出其优越的理由吗？下面我们就来仔细分析。

如果逐个数进行相加，需要相加 99 次。用高斯的办法，先算出 50 对数的加法，然后再进行一次乘法就可以得出。一共需要 50 次加法和 1 次乘法，即 51 次运算。这比起 99 次运算要少了 48 次运算。这里需要注意的是，虽然 50 对数的加法结果一样，但不能将 50 次加法看做一次加法。这是因为，至少在潜意识里还是需要进行加法的，只不过你一眼看出来它

们是一样的罢了。如果用计算机来说，则确实需要进行 50 次加法。

高斯的算法为优还有一个前提条件，就是乘法运算和加法运算在难度上与时间上是一样的。至少，一次乘法运算比 49 次加法运算要快！如果不是这样，则我们的分析就要打折扣了。但真的是这样吗？如果你学过了计算机的组成与体系结构，也许能够找出这个答案。

现在我们知道，高斯的算法为优不是我们想当然拍脑袋决定的，而是经过了分析后获得的结果，为判断算法的效率而对其进行的此种分析就是算法分析。

但是效率分析并不是算法分析的唯一目的。第 1 章说过，虽然算法追求的目标是速度，但算法必须首先正确才有存在的意义。

因此，设计算法时，或者对多个算法进行比较时，就要分析它们的正确性和时间效率。这种对算法进行解剖而获得其正确性和时间效率的操作就是算法分析。不过，正确性和时间分析并不是算法分析的唯一任务。如果两个算法的时间效率一样，我们就要对算法实现所使用的空间进行比较，空间使用较少的为优。

图 2-1 算法分析的三个主要方面

有时候，两个算法的时间、空间效率都可能相同或相似，这时候就要分析算法的其他属性，如稳定性、健壮性、实现难度等，并以此来判断到底应该选择哪一个算法。因此，算法分析可以分为以下三个方面，如图 2-1 所示。

- 正确性分析
- 时空效率分析
- 时空特性分析

2.1.1 正确性分析

毫无疑问，一个算法必须正确才有存在的意义。当然，有的人认为不正确的算法也有其存在的价值，但这是一个哲学问题，本书暂不予讨论。

但什么样的算法是正确的呢？

正确的第一个要素是能够终结，注意这里是“能够终结”，而不是“必须终结”。能够终结意味着如果输入或输出有限，则算法将在有限步骤内终结；而如果输入或输出无限，则算法也就无需终结。例如，计算圆周率 π 的算法在正常情况下就不应该终结，除非我们告诉这个算法仅计算到小数点后的某特定位。

算法正确的第二个要素是能够得出合理的结果，即这个结果能够与实际世界相符合，或者至少不相矛盾。在一般情况下，一个算法运行的结果是否正确比较容易判断。例如，排序算法是否正确，可以通过检查排好序后的数据是否真是按照非递增或非递减排列来判断。但也存在某些情况，正确性的判断并不直接。还是用 π 作例子，计算 π 的有效位的算法是否正确就不是一件简单的事情。因为 π 的正确值由无限位构成，任何计算出有限位的算法都不正确，但计算无限位的算法则无法终结，也就不可能计算出最后的结果。

2.1.2 时空效率分析

前面说过，速度是算法之魂。因此，一个算法只有正确性并无太多意义。一个让人等上几百年的算法，就是再正确，也不会令我们满意。因此，算法的速度非常重要，甚至在广义上可以判为正确性的一部分。需要注意的是，这里的速度是一个抽象概念，指的是算法计算所需要的步骤，而不是具体的多少小时、多少分钟等。

除了速度外，一个算法在实现的时候需要占用的空间也是一个考虑的因素。本书前面已经说过，算法在很多时候是在计算机上实现的，而在计算机上实现就需要占用空间，这里的空间指的是内存，可以是物理内存，也可以是虚拟内存，但不包括磁盘，因为磁盘便宜使得其不在我们的考虑范围内，占用空间少的优点不只是空间节省。也许大家知道，占用内存少的程序通常能运行得更快，即空间的节省有可能转化为时间的节省。因此，在其他因素相同的情况下，节省空间的算法更优。

事实上，时间和空间是可以相互转换的。在物理学领域如此，在计算机领域也是如此。

当然了，如果我们只是讨论抽象的算法，并不管它的实现，则空间的讨论就是多余的。但是一个算法如果不要实现，讨论它又有什么意义呢？恐怕其唯一的意义在于哲学上了。

2.1.3 时空特性分析

除了正确性分析和时空效率分析外，有时候我们还需要进行时空特性分析，例如稳定性、健壮性、实现难易性等。一个算法在实现主要目的的同时，还能实现一些附加目的，那么这种算法就比只实现主要目的的算法更优。例如，在排序时，有的算法是稳定的，即值相等的数据其相对位置保持不变；有的是不稳定的，即相等数据的相对位置在排序的过程中有可能发生变化。那么，稳定的属性就是一个附加属性。

另外，算法还应该考虑到实现的难易性问题。有的算法在抽象上非常精妙，但具体实现起来则可能存在诸如难以理解、难以在计算机上表示、编程困难等问题。这样的算法就不如那种容易理解、容易表示、容易编程的算法好。

有时候，我们会因为算法展现出的某种人性特质而喜欢上它，例如我们后面将要讲到的快速排序，即使运气很差，但只要给它一点机会，就能实现很好的效率。这样的算法由于有着我们人所追求的“坚韧不拔”的精神而获得人们的喜欢。当然，这不是我们喜欢快速排序的唯一理由。我们将在第 10 章中详细论述。

2.2 计数：算法分析的核心

前一节已经论述了算法分析的重要性及算法分析的三个方面，那如何进行算法分析呢？具体来说，我们怎么知道一个算法正确呢？怎么知道一个算法的步骤数是多少呢？怎么知道一个算法是否在完成主要目的之外有附加收益呢？

正确性的判断当然要依赖于证明了，大家在中学和大学应该已经学过许多的证明方法。

附加收益则依赖于敏锐的观察，这个靠长期的训练。而对于算法的时空效率分析所需要的技术则是本书第 1 章已经论及过的——计数，即算法的时空效率分析的基石就是**计数**（见图 2-2）。

图 2-2 计数是算法分析的核心

计数？谁不会啊。我们从小就学会了计数，难道算法分析原来就如此简单？

如果你觉得简单，那我们来看看前言中提到的例子：

```
for (i=1; i<=n; i*=2)
        for (j=1; j<=i; j++)
            laugh++;
```

在上述 C 程序片段（即算法在计算机语言上的表示）里，唯一的实际工作是 laugh++，而这个语句执行的次数就是该算法的步骤数。那么这个步骤数是多少呢？或者说上述程序片段中的 laugh++语句一共执行多少次呢？

乍一看似乎很简单，不就两重循环嘛。只需要将内外两重循环的执行次数相乘即可。但问题是内外两重循环的各自次数是多少呢？内层循环的次数显然是 i，但外层循环的次数是多少呢？是 n 吗？显然不是。$n/2$ 吗？也不正确。$n/3$？$n/4$？这个时候才发现，要数一下变量 i 的取值次数还真不容易。

2.3 算法设计

算法设计就是针对一个问题，设计一个解决方案。由前面算法分析的结果可知，在设计算法时必须注重下面几个因素：

- 必须正确；
- 步骤尽可能少；
- 实现尽可能简单；
- 空间占用尽可能少；
- 根据用户要求提供其他附加收益。

下面就以数性测试为例对算法设计进行简单的入门介绍。

所谓的数性测试就是对一个数的性质进行测试。在数论里，数可以按照其性质而进行分类：偶数、奇数、素数、斐波那契数、三角数、孪生数、完美数等。给定任意一个自然数 N，进行数性测试就是判断 N 是否是素数，是否是斐波那契数，是否是完美数，是否是三角数或孪生数等。不要认为进行数性测试是一种抽象的脑部锻炼，其实进行数性测试有着重要的实际意义。例如，素性测试在计算机及信息安全领域有着极为重要的意义。而测试一个数是否为合数、完美数、三角数、孪生数和其他一些性质的数则能够提升我们对宇宙及自身的认识和理解。

下面我们就来看一下数性测试中最为受人关注的一种测试：素性测试。

给你一个自然数 N，判断 N 是否是素数。

这是个看上去很简单的问题。例如，如果 N 是 53，我们马上可以断定它是一个素数。但真的简单吗？如果 N=2 396 345 897 613，你是否能很快判断其是否为素数呢？恐怕不太容易，但如果用计算机来解决这个问题是否会容易些呢？答案是不一定。

最直接判断素数的办法是将所有大于 1 但小于该数的整数作为潜在因子，一个一个地检查。如果其中某个数确实是因子，则该数不是素数。如果所有潜在因子都被证明不是真正的因子，则该数就是素数。当然，这种判断需要检查 N–2 个数。

但真的需要测试这么多个数吗？

我们注意到，如果 N 真的能被分解为两个大于 1 的因子之积，则至少其中一个因子将小于等于 $\sqrt{N}$ 。因此，我们实际上只需要检查从 2～$\sqrt{N}$ 的数即可。这样我们将算法的时间成本降低了一个平方根级，这是很大的一个改善！不过这种改善还是不够的。因为它仍然是一个指数级算法（后面我们将讨论到指数级算法）!

因此，使用直接的素数判断方法——这里采用的是因式分解法，效率低下。所以，要想提高素数判定的效率，必须抛开因式分解这条路。本书将在第 6 章再回到素性测试问题上来。

2.4　算法效率表示

现在，我们已经知道算法分析的关键是效率分析，那么如何表示一个算法的效率呢？

最直截了当的是用算法在计算机上运行一遍所需要的时间来表示。这似乎也是非常合理的一种办法：一个算法运行所需要的时间不就是它的效率吗？这个想法不错，但在实际操作中存在问题：不同的计算机速度不同，这里的速度包括相对速度和绝对速度，其采用的指令集结构、缓存策略等技术细节是不同的。这些不同将影响到一个算法在计算机上具体实现的运行时间需求，从而导致对算法的分析被计算机体系结构和操作系统等所影响与干扰。即在不同的指令集结构上运行同一个算法其时间将不一样。那么以哪台计算机上运行的时间为准呢？

当然，我们也可以用算法在转换为计算机程序后所执行的指令条数来表示其效率，这种方法似乎非常精确，也不受具体体系结构的影响。但仔细分析发现，这种分析方法将受到编程技术的影响，而且分析过程复杂。除了一些非常简单的算法外，难以实现。

因此，我们需要寻找一种简便易行的、与具体机器和程序员编程技术无关的算法效率度量方法，即把对机器、编译器、程序设计员的依赖拿开，纯粹描述的是算法本身的时空效率性质。这就像我们讨论过的素性测试，第一个算法由于要检查 N–2 个数，所以我们说该算法的时间效率是 N–2；而第二个算法的效率则是 $\sqrt{N}$ –2。检查 N–2 和 $\sqrt{N}$ –2 个数要多长时间则不在我们的考虑中，换一个说法，我们假定对测试任意一个数的时间成本为固定值。这样就完全屏蔽了不同机器所可能带来的影响。

诚然，我们在设计算法时当然应该考虑计算机的特异性，并对这种特异性大加利用，从而进一步提升算法在实现时的运行效率，但这种考虑不应该是算法设计的主要考虑。因为，机器总是在变的，而算法本身的时空效率特性，或者说算法的灵魂才是重要的。

2.5 渐近分析

在分析两个解决同一个问题的算法中，如果一个算法比另一个算法多一个步骤，我们可能不会认为这两个算法的效率有何不同。如果多两个步骤，也是如此。但是多 100 个步骤呢？也许这两个算法的效率就有区别了？就像 2.3 节讨论的两个测试素性的算法，一个 N–2，一个 $\sqrt{N}$ –2，它们是效率一样的算法吗？

问题的核心是，，我们需要在某个地方画一条线，凡是在此线范围内的算法的效率被认为差不多，而超过这个范围就是效率有很大不同，我们需要的是对数量级的表述。

下面以素性测试为例加以说明。

如果将自然数 N 以计算机里的二进制来表示，以该二进制的字位数作为参数，则第一个素性测试算法的时间复杂性为指数级，即运算时间在 $2^{\log N}$。

也许有人会说，$2^{\log N}$ 不就是 N 吗？N 不是代表线性吗？线性是效率很高的算法，不是很好吗？如果读者也这么想，那么请再想想：对于计算机来说，每一个位都需要硬件来存放，计算机的运算也是一位一位进行的。当然，在某些情况下，通过增加硬件我们能够获得一定的并行，但这样并不减少表示每个位所需要的硬件成本。因此，将 N 转换为计算机里面的二进制表示后，该数所占用的计算机位数即是该问题的输入尺寸，而计算机进行运算的时间自然应该以该尺寸为衡量标准。

那么，第二个素性测试的算法时间复杂性是多少呢？答案是 $2^{\log \sqrt{N}}$ 。

那么 $2^{\log N}$ 和 $2^{\log \sqrt{N}}$ 有多大区别呢？在 N 很小的时候，区别不大。但随着 N 趋向于无穷的时候，这两个值的差别越来越大，即对于很大的 N 来说，这两个算法的效率呈现出巨大的区别。这种考虑算法在输入规模趋向无穷时的效率分析就是所谓的渐近分析。

渐近分析就是我们的大思维：**忽略具体机器、编程或编译器的影响，只观察在输入尺寸 n 取趋向无穷时算法效率的表现！**即我们关注的是趋势，打个比方就是“只见人影不见人”，如图 2-3 所示。

图 2-3 渐近分析关注的是趋势（只见人影不见人）

用渐近表示的另一个好处是：大大降低了分析算法的难度。本章前面说过，计数是算法分析的核心，但计数并不是一件容易的事情。如果要非常精确地来计数，则难度就更大了，有时甚至是不可能的，而渐近表示则免除精确计数的负担，从而使算法分析的任务变得可以控制。接下来我们就来引入各种具体的渐近表示。

2.6 O、Ω、Θ 表示

对于任何数学函数 $f(n)$，O、Ω、Θ 这三个记号可以用来度量其“渐近表现”，即当 n 趋于无穷大时 $f(n)$ 的阶的情况，这是算法分析中非常重要的概念。大家可以把它们分别想象成 $\leqslant$、$\geqslant$和 $=$，分别估计了函数的渐近上界、渐近下界和准确界。诚然，渐近关系和确切大小关系是有区别的，但当问题规模很大时，忽略这种区别能大大降低算法分析的难度。

下面我们就来具体定义这三种记号的表示。

设函数 $f(n)$代表某一算法在输入大小为 n 的情况下的工作量（效率），则在 n 趋向很大的时候，我们将 $f(n)$与另一行为已知的函数 $g(n)$进行比较：

1）如果 $\lim\limits_{n\to\infty} f(n)/g(n)=0$，则称 $f(n)$在数量级上严格小于 $g(n)$，记为 $f(n)=o(g(n))$。

2）如果 $\lim\limits_{n\to\infty} f(n)/g(n)=\infty$，则称 $f(n)$在数量级上严格大于 $g(n)$，记为 $f(n)=\omega(g(n))$。

3）如果 $\lim\limits_{n\to\infty} f(n)/g(n)=c$，这里 c 为非 0 常数，则称 $f(n)$在数量级上等于 $g(n)$，即 $f(n)$和 $g(n)$是同一个数量级的函数，记为：$f(n)=\Theta(g(n))$。

4）如果 $f(n)$在数量级上小于或等于 $g(n)$，则记为 $f(n)=O(g(n))$。

5）如果 $f(n)$在数量级上大于或等于 $g(n)$，则记为 $f(n)=\Omega(g(n))$。

这里我们假定 $f(n)$，$g(n)$是非负单调的，且极限 $\lim\limits_{n\to\infty} f(n)/g(n)$ 存在。如果这个极限不存在，则无法对 $f(n)$和 $g(n)$进行比较。在进行此种计算时，一个经常用到的技术是洛必达（L'Hôpital）法则。该法则由 17 世纪法国数学家 Guillaume de L'Hôpital 发现（也有人认为是瑞士数学家 Johann Bernoulli 发现的）。该法则声称，两个函数的比率极限等于两个函数的导数的比率极限，这里当然假定两个函数的导数比率的极限存在，即有：

$$\text{若 } \lim_{n\to\infty}\frac{f'(n)}{g'(n)} \text{ 存在，则 } \lim_{n\to\infty}\frac{f(n)}{g(n)}=\lim_{n\to\infty}\frac{f'(n)}{g'(n)}$$

有了这个定义，就可以对素性测试的两个算法进行比较了。

$\lim\limits_{n\to\infty}\dfrac{\sqrt{2}^{\log N}}{2^{\log N}}=0$，符合第 1 个定义，因此这两个素性测试算法的效率差异是数量级的差异。

在算法分析中，最常选取的 $g(n)$有如下一些，见表 2-1。

表 2-1 常见函数 $g(n)$的测试数量级

函　数	测试数量级
$g(n)=1$	常数级
$g(n)=\log n$	对数级
$g(n)=n$	线性级
$g(n)=n^2$	平方级
$g(n)=n^3$	立方级
$g(n)=2^n$	指数级

图 2-4 描述的是这些函数在 n 趋向无穷时的增长趋势。

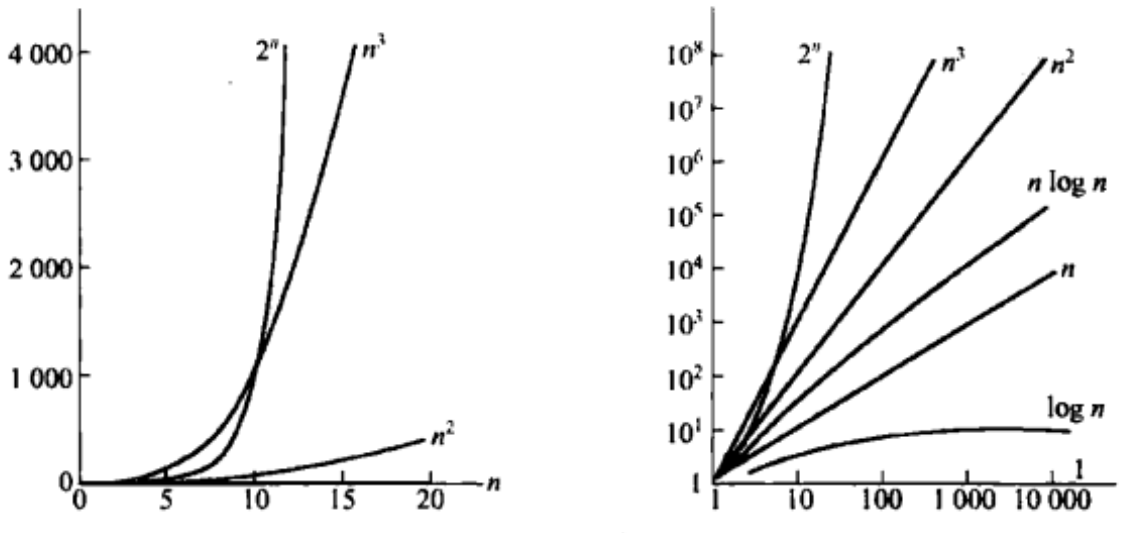


图 2-4 常见函数的渐近趋势

一个值得提醒的问题是，根据定义，对于任意一个 $g(n)$函数来说，可能存在很多个函数 $f(n)$，使得 $f(n)=O(g(n))$，即 $O(g(n))$表示的实际上是一个函数的集合，这里的等于也不是普通意义上的等于，而是说明 $f(n)$是函数集合 $o(g(n))$里的一员，即 $f(n)=O(g(n))$并不意味着 $f(n)$等于 $O(g(n))$。等于号的这种使用令那些严谨的科学家非常不快甚至愤怒，但计算机界人士很喜欢这种马虎的表示。不过，我们在心里应该知道，$f(n)=O(g(n))$并不意味着 $f(n)\neq O(g(n))$。不然，我们就被自己骗了！

等号在其他渐近表示中的使用也可以同样解释。

2.7 最好、最坏、平均

如果一个程序运行多次，则有时候它会快点儿，有时候它会慢点儿。算法也一样，在输入 1 的情况下和输入 2 的情况下，其执行效率不一定一样。即算法会随着输入数据的不同而有秩序效率的不同，有时候会快点儿，有时候会慢点儿。例如，对一个已经排好序的序列进

行排序就要相对容易一些。另外，输入规模的大小也影响算法的运行时间。例如，一个短的序列就比一个很长的序列容易排序。因此，我们在对算法进行分析时需要考虑输入的大小。

一般来说，我们希望获得一个算法的时间效率下限，因为所有人都喜欢某种保证：即算法无论如何不会低于我们保证的效率。这种分析就是所谓的最坏情况分析. 最坏情况分析指的是在给定输入尺寸的情况下，一个算法运行的效率的下限。

除了最坏情况外，我们更有兴趣的是平均情况。因为这更能反映大多数情况下算法的表现。平均情况分析就是对所有输入尺寸为 n 的输入，让算法运转一遍，然后取它们的平均值。当然，实际中不可能将所有可能的输入都运行一遍，因此平均情况通常指的是一种数学期望值，而计算数学期望值则需要对输入的分布情况进行假设。

有时候我们还需要知道最好情况是什么，这有两层意义：一是我们想知道如果运气好，能好到什么程度；二是如果我们能够证明好运气与我们同在，当然需要知道运气好的时候算法表现如何。这种最好分析就是在给定输入规模的时候，看看哪种输入能使算法的运行最有效率。当然，有人认为这种最好情况分析有点假：我们可以操控输入来使一个本来很慢的算法表现得很快，从而达到蒙蔽人的效果。

如果一种情况导致算法的执行效率最高，则这种情况称为最好情况。导致最低效率的情况自然就是最坏情况。

例如，在一个数组里面通过顺序扫描找到值 x，我们需要多少次比较？

在最好的情况下，我们只需要比较一次，因为第一个元素就是我们要找的元素。而在最坏的情况下，需要扫描整个数组，因为要找的元素恰恰在最末尾。但如果一个人的运气一般，既不是最好，也不是最坏，则需要比较的次数大约是一半的数组元素，即 $T(n)= n/2$。

因此，对一个算法进行分析，我们也可以分最好、最坏和平均三种情况进行。当然，最好情况通常不太现实或者说归于乐观；而最坏情况分析则过于悲观，而平均是我们最想知道的。因为一个算法如果运行很多遍，其平均表现就是这个平均情况。表 2-2 给出的是各种排序算法在输入规模为 100 万个数据元素时的最好、最坏及平均情况下的排序时间。

表 2-2　输入规模为 100 万个数据元素时的最好、最坏及平均情况排序时间（单位：毫秒）

排 序 算 法	平 均 情 况	最坏情况（逆序）	最好情况（正序）
冒泡排序	549 432	1 534 035	366 936
选择排序	478 694	587 240	367 658
插入排序	253 115	515 621	0.897
希尔排序/增量 3	61	203	35
堆排序	79	126	74.8
归并排序	70	140	61
快速排序	39	93	30
基数排序/进制 100	117	118	116
基数排序/进制 1 000	89	90	88

注：1. 算法运行环境为 Intel 酷睿 2 双核 E8400，3.0G, Windows 7*64.

2. 本表数据由作者所上“数据结构”课的胡嘉斌同学测试所得.

从表 2-2 可以看出，最好与最坏情况下的时间差别可能很大，也可能很小。而这个差别

到底是大还是小，则取决于具体的排序算法。差别最大是插入排序，其最好情况下的时间为0.897 毫秒，而最坏情况下的时间则达到了 515 621 毫秒。相差最小的是基数排序，其最好与最坏情况几乎没有什么区别，而这是由基数排序的性质所决定的。其他排序算法的最好与最坏的差别则介于插入和基数排序之间。其实，对于某些算法来说，表 2-2 并没有给出严格的最好和最坏情况。例如，快速排序的最好和最坏情况并一定在正序和逆序时出现，这取决于杠杆点的选择方法。本书第 10 章将详细讨论各种排序算法。

这里需要注意的是，最好情况并不是当 n=1 的时候。就像在一个数组里面查找特定数，我们不能说最好情况是数组只有一个元素的情况，这个时候比较次数为 1。渐近表示代表的是一个函数在输入变量或输入尺寸 n 趋向无穷时的增长速度，而最好情况下指的是在输入尺寸相同的情况下，哪种具体输入数据会导致最少的执行时间，如表 2-2 所示。

2.8 O、Ω、Θ 的另一类定义

除了用极限来定义渐近表示外，在算法分析时还常用另外一种定义。

对于非负函数 $T(n)$，存在两个正常数 c 和 n_0，并且对于所有的 $n>n_0$，有 $T(n)\leqslant cf(n)$，则称 $T(n)$在集合 $O(f(n))$ 里，记为 $T(n)=O(f(n))$。直观意义是，对于足够大的数据集合，本算法执行的步骤数总是少于 $cf(n)$。

很显然，大 O 表示的是一个上限。例如，如果 $T(n)=3n^2$，则 $T(n)$ 在集合 $O(n^2)$里。在进行算法分析时，希望获得的是尽可能紧凑的上界。例如，虽然 $T(n)=3n^2$ 是在 $O(n^3)$里，但我们更喜欢 $O(n^2)$，因为 $O(n^2)$比 $O(n^3)$更紧凑。

例 2-1 在一个数组里面找到值 x，平均需要多少时间或多少次比较？显然，$T(n)=c_sn/2$。

由于对于所有的 $n>1$，$c_sn/2\leqslant c_sn$，因此根据定义，有：$T(n)=O(n)$，这里 $n_0=1$，$c=c_s$。

例 2-2 计算函数 $T(n)=c_1n^2+c_2n$ 的上限。

由于，对于 $n>1$，有 $c_1n^2+c_2n\leqslant c_1n^2+c_2n^2\leqslant (c_1+c_2)n^2$，所以有 $T(n)\leqslant cn^2$。这里 $c=c_1+c_2$，$n_0=1$。因此，$T(n)$在集合 $O(n^2)$里，即 $T(n)=O(n^2)$。

我们关心的是当输入规模很大时算法的效率。此时，时间复杂性函数中阶最高的部分对其贡献占主导地位，其他部分对复杂性函数的贡献皆可忽略（与最高阶部分之比的极限趋于0）。在不需要很精确的时候，最高阶部分的系数也可忽略。

对大 Ω 的定义可以类似进行，大 Ω 给出的是下限。

例 2-3 计算 $T(n)=c_1n^2+c_2n$ 的下限。

因为，对于所有的 $n>1$，我们有 $c_1n^2+c_2n\geqslant c_1n^2$，因此，$T(n)\geqslant cn^2$，这里 $c=c_1$，$n_0=1$。根据定义，我们有 $T(n)$在集合 $\Omega(n^2)$里。

如果大 Ω 和大 O 交界，则我们用大 Θ 来表示，即如果一个算法的效率既在 $O(f(n))$里，又在 $\Omega(f(n))$里，则我们说，该算法的效率在 $\Theta(f(n))$里。用我们的常规定义来时就是，如果存在正常数 c_1、c_2，对于所有 $n>n_0$，我们有 $c_1f(n)\leqslant T(n)\leqslant c_2f(n)$，则 $T(n)=\Theta(f(n))$。也就是说，当 n 趋向无穷时，$T(n)$的增长速度与 $f(n)$相同。

2.9　O、Ω、Θ 的性质

O、Ω、Θ 具有很多有趣的性质，利用它们可以大大简化对算法的分析！

性质 1　O、Ω、Θ 是自反的，传递的。例如：

- 自反性，$f(n)=O(f(n))$。
- 传递性，若 $f(n)=O(g(n))$，$g(n)=O(h(n))$，则 $f(n)=O(h(n))$。

性质 2　Θ 还具有对称性：$f(n)=\Theta(g(n))\Leftrightarrow g(n)=\Theta(f(n))$。

性质 3　对于任意的 $f(n)$，$g(n)$，我们有：$f(n)+g(n)=\Theta(\max\{f(n), g(n)\})$，即对一个由若干部分组成的串行程序，其总的渐近复杂性等于其复杂性最高的部分。这是一个十分有用的性质。

性质 4　如果 $f(n)=O(kg(n))$，这里 k 是一个常数，则 $f(n)=O(g(n))$。这条规则告诉我们，系数是无关紧要的。因为我们关心的是数量级的比较，而不是同一数量级里面的细小不同。

性质 5　如果 $f_1(n)=O(g_1(n))$，并且 $f_2(n)=O(g_2(n))$，则 $(f_1+f_2)(n)=O(\max(g_1(n), g_2(n)))$。

性质 6　如果 $f_1(n)$ 在 $O(g_1(n))$ 里，并且 $f_2(n)$ 在 $O(g_2(n))$ 里，则 $f_1(n)f_2(n)$ 在 $O(g_1(n)g_2(n))$ 里。

上述所有性质都可以证明。我们这里证明性质 3，其他性质读者自己证明。

证明　性质 3：不失一般性，假定 $f(n)\geqslant g(n)$，即 $\max\{f(n), g(n)\}=f(n)$，则：

$$\lim\frac{[f(n)+g(n)]}{\max\{f(n),g(n)\}}=\lim\frac{[f(n)+g(n)]}{f(n)}=1+\lim\frac{g(n)}{f(n)}\leqslant 1+1=2$$

因此，$f(n)+g(n)=\Theta(\max\{f(n), g(n)\})$。　□

2.10　要更快的计算机还是要更快的算法

也许有人认为，随着计算机速度的不断提升，研究更快的算法可能没有太大必要。下面我们就来看一下，计算机速度的增长到底可以在多大程度上解决问题。假定目前计算机能够计算的输入大小为 n，如果计算机速度增长 10 倍，则能计算的输入规模会增加多少呢？

表 2–3 给出的是新、老计算机在 $T(n)$ 为不同数量级的情况下，计算量的变化。

表 2–3　计算机速度提升 10 倍时的计算量的变化

$T(n)$	n	n'	计算改变量	n'/n
$10n$	1 000	10 000	$n'=10n$	10
$5n\log n$	250	1 842	$\sqrt{10}\,n<n'<10n$	7.37
$2n^2$	70	223	$n'=\sqrt{10}\,n$	3.16
2^n	13	16	$n'=n+3$	…

我们看到，如果一个算法效率是线性的，则一个速度快 10 倍的计算机可以处理的数据量将是原来计算机的 10 倍，即呈线性增长。但如果算法效率是平方级的，则一个快 10 倍的计算机能够处理的数据量仅增加了约 2 倍。如果算法效率是指数级的，则几乎没有任何增长。对于一个指数级成本的算法来说，在 n 趋向很大的时候，计算机速度的增长不会带来任何实际收益。

如果上述数据比较抽象，那么下面我们就来看一个具体的：斐波那契数列的计算。一个朴素的计算任意斐波那契数的算法，即按照斐波那契数列定义进行计算的算法，它的时间复杂性约为 1.6^n，即计算 F_{n+1} 的时间约是计算 F_n 的 1.6 倍。如果计算机按照摩尔定律（每 18 个月翻倍的规律）改进速度，则每年改进约 0.6 倍至原来速度的 1.6 倍。这样，如果今年能计算 F_n，则明年的计算机只能计算到 F_{n+1}，即每过一年，我们只能多计算一个斐波那契数。

截至 2009 年 8 月，世界上最快的超级计算机是位于美国拉斯阿拉莫斯（Los Alamos）国家实验室的 IBM 竞跑者（Roadrunner），其浮点计算能力为每秒 1 105 千万次。它比 2002 年世界上最快的计算机——日本的 NEC 地球模拟器（Earth Simulator）快了近 30 倍。从绝对意义上讲，这种速度提升不可谓不大。但是对于朴素的计算斐波那契数的算法来说，这种改进没有任何意义：用同样的时间，竞跑者只能比 NEC 地球模拟器多算 7 个数！

如果我们设计出一个优良的算法，则可以将斐波那契数的计算成本降为 $\log n$（本书后面章节将阐述这一算法）。这样，在新的算法下，用同样的时间，竞跑者能比 NEC 的地球模拟器多算 10^{30} 个斐波那契数！而这几乎是一个“无穷大”的改进！

可见，改进算法比提升计算机速度的效果要大得多！因此，追求算法的效率是我们永恒的执著，而效率也无愧于被称为算法之魂！

思考题

1. 有同学说，我的算法的最好情况是当 $n=1$ 的时候，因为那是算法运行最快的时候。这句话有什么问题吗？为什么？
2. 如果 $f_1=O(n^2)$, $f_2=O(n)$，假设它们对应解决同一问题的两个不同算法，那么第一个算法是不是完全没有用武之地？
3. 证明：如果 $f_1(n)=O(g_1(n))$，并且 $f_2(n)=O(g_2(n))$，则 $(f_1+f_2)(n)=O(\max(g_1(n), g_2(n)))$。
4. 证明：渐近表示的两种定义互相等价，即从一个定义可以推导出另一个定义。
5. 计算下面（C 程序片段）嵌套循环里 laugh++语句执行的次数。

```
for (i=1; i<=n; i*=2)
 for (j=1; j<=i; j++)
    laugh++;
```

6. 对一个 n 位的整数进行平方是否比对两个 n 位的整数进行相乘更快呢？为什么？
7. 求解下面的递归表达式的最紧的渐近上限。

（a）$T(n)=3T(n/3)+O(\log n)$

（b）$T(n)=2T(n/8)+\sqrt[3]{n}$

（c）$T(n)=T(n/3)+T(n/4)+5n$

（d）$T(n)=\begin{cases}8T\left(\frac{n}{2}\right)+\Theta(1) & n^2>M\\ M & n^2\leqslant M\end{cases}$，这里 M 是独立于 n 的变量。

8. 证明下面渐近表示性质：

（a）如果 $f(n)\neq O(g(n))$，且 $g(n)!=O(f(n))$，则 $g(n)=\Theta(f(n))$。

（b）$f(n)+O(f(n))=\Theta(f(n))$。

9. 谈谈你对渐近分析的理解，这种分析和简单的计数分析有何联系？

10. 假如一个算法的平均时间复杂性与最坏情况一样，但最好情况下的时间复杂性则好很多。因此，要想成功就得保证最好情况的出现。试讨论有哪些办法可以保证最好情况的出现。

PDG

CHAPTER 3

第 3 章　分治与递归

“你站在桥上看风景，看风景的人在楼上看你，明月装饰了你的窗子，你装饰了别人的梦。”

——卞之琳《断章》

上面的这首诗包含的是一个递归概念，而图 3–1 这幅画则明显地展示了递归。如果细心些，你还会发现，生活中到处都是递归！

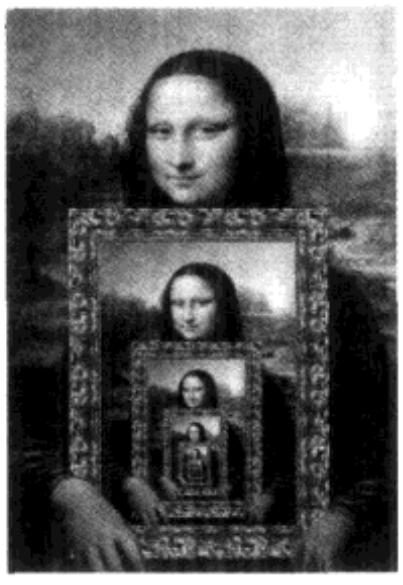


图 3–1　递归无处不在：递归的蒙娜丽莎

例如，很多人恐怕都见过一个这样的称球游戏：给定 n 个球，其中 1 个球为次品。次品从外表上看与正常球一样，但重量有区别。它可能比正常球重，也可能比正常球轻。现在给你一个天平，我们的问题是，需要称几次才能将次品甄别出来？

这个问题的解答思路十分简单：就是将次品所在的范围不断缩小，直到只有 1 个球为止。例如，先将所有的球分解为两个相等的部分（如果 n 为奇数，则无法分解为相等的部分，但本思路仍然适用），将其置于天平的两端，如图 3–2 所示。此时必然一端重一端轻。问题是我们并不知道次品是重还是轻。因此需要分别对每部分的 $n/2$ 个球再进行一次上述这样的测试。如果平衡，则这部分的 $n/2$ 个球均为正品，即次品在另外的 $n/2$ 个球里。否则，次品就在这 $n/2$ 个球里。这样我们就将寻找次品的范围缩小了一半。如果读者很细心，就会

发现第 1 次称秤（*n*/2 个球对 *n*/2 个球）并不需要，你看出来了吗？

我们可以按照这个方法不断缩小次品范围，直到问题解决为止（次品范围只有 1 个球的时候）。而这种将次品范围不断缩小，但解题方法维持不变的方法就是递归。

图 3-2 用天平来称球并辨别次品用到了递归策略

当然，本章对递归进行讨论并不是因为我们要称球，或者生活中到处都存在递归，而是因为在算法的设计与分析中，递归是一个普遍且难以回避的解题方法。

而且更有意思的是，递归又常常与算法里面的另一个重要概念“分治”（分而治之）紧密联系。事实上，递归在很多时候就是因为分治策略的使用才出现的，可以说，没有递归，就没有分治。分析一个分治策略的优劣就经常牵涉对递归表达式的分析。例如，我们前面的称球游戏在使用递归的同时也使用了分治：将规模较大的问题（在 *n* 个球里面寻找次品）化解为同类型的规模较小的问题（在 *n*/2 个球里面寻找次品）。

本章就来探讨分治与递归。这两个概念密不可分，它们是算法设计和分析的根本。

3.1 分而治之为上策

当你面临一个复杂的问题时，该如何下手呢？当然是首先将问题简化，即将复杂的大问题分解为简单的小问题，然后分而治之。这正是算法里一个非常重要的战略。听说过中国古代总结出来的分治策略吗？秦帝国就是通过合纵连横的分治策略而统一了当时的六国。

如果读者对古代的合纵连横不感兴趣，那么对现代的各种体育及文艺竞赛总有点兴趣吧。而这里面同样蕴涵着分治的思想。例如，在足球世界杯赛中（见图 3-3），我们目的是从全球近 200 支球队里面选出最好的。但直接在这么多的球队里面选择难度很大，成本很高。而明智的策略就是将全球的球队分解为不同赛区，在每个赛区选出屈指可数的几支球队出来，然后在这些屈指可数的球队里面再进行评优，难度和复杂性就低多了。而这种方法就是分治：将一个大问题（在全球近 200 支球队里面选优）分解为多个类型相同规模更小的小问题（在每个赛区选优），分别解决小问题（选出每个赛区的优秀球队），最终使大问题迎刃而解。

图 3-3 司空见惯的世界杯足球赛蕴涵着分治的思想

前面已经列举了生活中的例子，接下来我们用一个简单的数学例子来彰显分而治之策略的优越性。假定 x 和 y 是两个长度为 n 位的整数，为方便起见，假定 n 是 2 的指数次方（不是指数次方也可以同样处理）。我们的目标是要计算 x 和 y 的乘积。

如果我们直接用蛮力相乘来求取乘积，则时间复杂性为 n^2。这个时间复杂性读者也许觉得并不坏，但如果一个程序需要进行大量的乘法运算，则这个时间复杂性就不是那么理想了。

那么有没有更好的算法呢？有。策略就是分治！将大问题分解成小问题。

问题是乘法运算怎么分治解决呢？或者说把什么分解为更小的单位呢？当然是将乘数和被乘数进行分解（难道还有别的东西可以分解的吗）。

首先将 x 和 y 分别分解为左、右两半，每一半均为 $n/2$ 位长。例如，如果 $x=10110110$，则 $x_L=1011$，$x_R=0110$。则 x 和 y 之积可以表示为：

$$xy=(2^{n/2}x_L+x_R)\ (2^{n/2}y_L+y_R)=\ 2^n x_L y_L\ +\ 2^{n/2}(x_L y_R+x_R y_L)+x_R y_R \qquad (3\text{-}1)$$

即我们可以根据式（3-1）右面的表达式来计算 x 和 y 之积。

乍一看，感觉右边的表达式远比左边的 xy 表示复杂，难道其效率反而更高？

那我们仔细来看：右面表达式里面的加法运算可在常数时间内完成，2 的指数次方运算也可在线性时间内完成（只需将计算机里面的字进行左移即可）。因此，右面表达式里面的主要复杂性在于其 4 个乘法运算。但是，这 4 个乘法运算的操作数长度均只有 $n/2$ 位，比原始的乘法运算操作数的长度少了一半！这样，我们需要做的乘法运算就是 4（$n/2\times n/2$）$=n^2$。

这好像没有改进嘛。我们费了很大的力气（耗去很多脑细胞），却没有带来任何效率的改进。

难道分治错了吗？

德国数学家高斯很早就发现，虽然两个复数的乘法初看上去涉及 4 次实数乘法运算，但实际上可以化简为 3 次实数乘法运算。受此启示，式（3-1）的右面部分也可以简化为 3 次乘法运算，即计算 x 和 y 的乘积所需要的乘法运算实际上是 3 个规模为（$n/2$）×（$n/2$）的运算，即（3/4）n^2。你能看出来可以简化为怎样的 3 个乘法运算吗？

这一下总算有了改进：从 n^2 降到到 $3n^2/4$。也许有人认为这个改善微不足道，况且本书第 2 章在论述渐近分析时不是说过吗，在渐近分析时，我们可以将系数忽略，这样 n^2 和 $3n^2/4$ 在 n 趋向无穷的时候就是一回事了。

如果我们的分治仅仅进行一步就停止，那么上述说法是完全正确的。关键是，我们为什么要停止继续分治呢？如果式（3-1）的右面可以改进为 3 个乘法运算，那么这些乘法运算又可以进一步经分治而变成 3 个长度再折半的乘法运算。而我们可以一直这样递归分治下去，最后必到达操作数长度为 1 位的情况，而此时的乘法运算成本就微不足道了（常数级啦！）。

这样，我们的乘法运算的运算效率将满足下述递归表达式：

$$T(n)=3T(n/2)+O(n) \qquad (3\text{-}2)$$

这里，$T(n)$代表长度为 n 位的乘法运算。按照式（3-1），它可以变为 3 个长度为 $n/2$ 的乘法运算。$O(n)$是将 3 个长度为 $n/2$ 的乘法运算结果组装起来所需要的时间（加法和移位操作）。

那么式（3-2）的求解是什么呢？答案是 $O(n^{1.59})$，这是一个巨大的改善。

$O(n^{1.59})$的时间复杂性也许不容易从递归式一眼看出，但如果我们画出一棵递归树（如图 3-4 所示），则这种结果就较容易看出了。

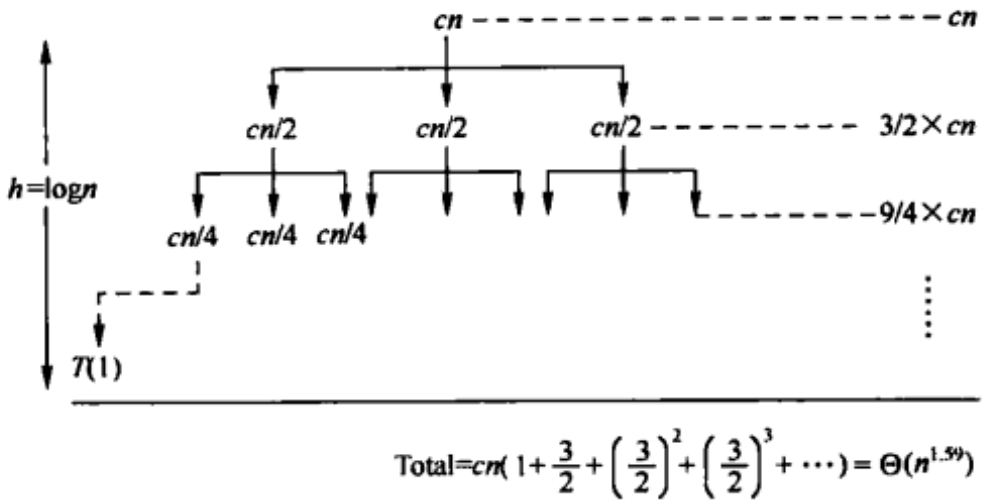


图 3-4 递归树帮助我们求解递归式 $T(n)=3T(n/2)+O(n)$

这里的关键是，在递归树的每一级，分支因子是 3，即每一级问题被分解为 3 个更小的同样问题。因此，在第 k 级，子问题树的数量增加到 3^k 个，但每个子问题的大小则为 $n/2^k$。由于树的高度为 $\log n$，因此，最后一级是 $k=\log n$。根据前述推理，这一级的工作量为 $O(3^{\log n})$，通过变换，该表达式也就是 $O(n^{\log 3})$，即我们前面给出的 $O(n^{1.59})$。由于每一级的时间成本呈几何级数增加，所以该成本就是整个乘法运算的成本。

3.2 分治策略

由上面的例子可见，采取分而治之策略解决一个问题有 3 个步骤：

1）将问题分解为若干个小的子问题。每个子问题与大问题同型，但规模更小。

2）递归解决这些子问题。

3）将子问题的解答合并，获得大问题的解答。

其中的第 2）步递归解决子问题指的是按照同样的分治策略进行求解，即通过将这些子问题分解为更小的孙子问题来进行求解。就这样一直下去，直到分解出来的子问题简单到只用常数操作时间即可解决为止。而递归是彰显分治优势的放大器。没有递归，则分治策略的效果不是没有就是微不足道。就像我们的分治乘法运算，如果只分解一次，效率的改善不足挂齿。

在分解到子问题规模达到微不足道的境界时，子问题的解即可用常数时间求得。然后我们仿照递归的顺序由底至上将子问题的解合并起来，逐级上推就构成了对原问题的解。

在分而治之的策略下，真正的工作也由上述 3 个步骤构成，即所有的工作分散于这 3 个地方：分解部分、递归部分和合并部分。而整个分而治之策略的时间复杂性也由这 3 部分的时间复杂性之和构成。由于在不断递归后，最后的子问题将变得极为简单，以至于小学生都能轻而易举地解决，其解决的时间复杂性在整个策略中的比重微乎其微，可以忽略不计。因此，分而治之策略的真正成本实际上由分解和合并两个部分构成，即到底需要分解多少次，每次分解的子问题数到底是多少？到底要合并多少次，每次分解和每次合并需要多少时间？而其中最为关键的是分解出来的子问题数、每个子问题的大小，以及分解与合并本身的时间成本。这些是决定分治策略是否奏效的决定因素。因此，设计优良的分解合并策略就十分重要。

3.3　递归表达式求解

在实施分治策略时，我们常常会碰到算法效率的递归表达式。这是由分治策略的递归特性所决定的。例如，我们在使用分治求解乘法的时候就碰到了 $T(n)=3T(n/2)+O(n)$的递归表达式。事实上，**标准分治策略的定义里面就包含着递归**。因此，求解递归表达式就是分析分治策略的重要基础。

一般来讲，分治策略将一个大的复杂的问题划分为 a 个同样的子问题，这些子问题的大小为 n/b。如果一直递归分解下去，我们获得的算法时间复杂性的递归表达式将是：

$$T(n) = aT(n/b)+f(n) \tag{3-3}$$

这里，b 是每次分解时问题规模减小的因子，a 是每次分解时产生的子问题个数，$f(n)$是分解出（和合并）a 个大小为 n/b 的子问题的成本。很显然，$a\geqslant 1$，$b>1$，并且 f 的渐近趋势为正。（如果该函数渐近表现为负会是什么情况？）注意，如果 $b\leqslant 1$，则说明这种分治完全失败：因为分解出来的子问题规模与源问题相当或超过了源问题！

这样，求解一个分而治之算法的时间效率就是求解上述递归表达式。那么上述递归表达式的解是什么呢？

3.3.1　递归树法

对于很多人来说，理解抽象的东西不如理解具体的东西更容易，而一个递归表达式就是一个抽象的东西，不容易看出其中所隐含的执行序列和规律，如每层递归的成本。但如果我们将该抽象表达式用图形的方式加以展开，则抽象变具体，就容易理解了。虽然不是所有抽象的东西都可以用具体的形象来描述，但抽象递归表达式恰恰可以被具体化。

而将抽象递归表达式具体化的最佳图形表示就是递归树。该方法我们在解乘法运算的递归表达式时已经使用过。这种递归树给出的是一个算法递归执行的成本模型。该模型以输入规模为 n 开始，一层层地分解，直到输入规模变为 1 为止。而这个时候的解决方案已经是琐细的了。图 3-5 为表达式 $T(n) = T(n/4) + T(n/2)+ n^2$ 的递归树。

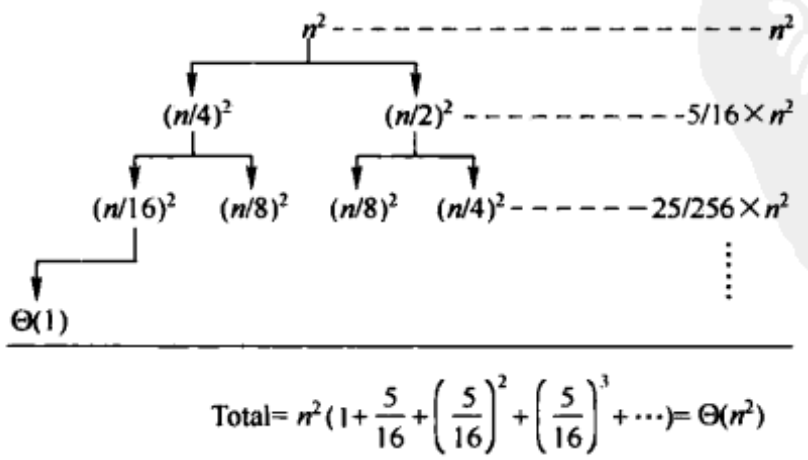


图 3-5　$T(n) = T(n/4) + T(n/2)+ n^2$ 的递归树

很显然，递归树解法的优点是直观，很多情况下我们可以直接看出来答案。但缺点是不

一定可靠，就像任何使用省略号的方法一样。（省略号到底是省略了什么？）例如，图 3-3 里面的省略号总让我们觉得这个结果不怎么令人放心：虽然第 1 层、第 2 层的成本没有任何问题，但我们怎么能断定没有画出的第 3 层的成本是$(5/16)^2n^2$呢？当然，为了保险，我们可以画出第 3 层并验证结果就是$(5/16)^2n^2$，但第 4 层的结果又如何肯定呢？

很显然，将每一层都画出来并不现实，因此，省略号就成为画递归树时不可缺少的工具。而这个省略号就成为很多人心头解不开的疙瘩。

3.3.2 替换解法

既然递归树的结果不怎么令人放心，那怎么克服这个缺点呢？当你对一个东西不是十分确信的时候，你怎么办呢？当然是进行验证！

因此，要解决递归树方法的可靠性问题，只要在递归树方案上增加一个步骤：将递归树获得的答案进行验证。例如，我们已经从递归树获得表达式 $T(n) = T(n/4)+T(n/2)+n^2$ 的解为 $\Theta(n^2)$。但这个答案是否可靠呢？只要加以证明即可。

根据 $\Theta(n^2)$的定义，我们只需要分别证明 $T(n) = O(n^2)$ 和 $T(n) = \Omega(n^2)$即可。

而根据 $\Omega(n^2)$的定义，我们只需要证明存在 n_0，对于 $n>n_0$，我们有 $T(n) \geqslant cn^2$，这里 c, n_0 为正的常数即可。而此种证明使用数学归纳法即可迅速获得。

假定上述条件对于所有小于 n 的输入都成立，则有：

$$T(n) = T(n/4) + T(n/2)+ n^2 \geqslant c(n/4)^2+c(n/2)^2+n^2=cn^2+(1-11c/16)n^2$$

我们只要选择 $c \leqslant 16/11$，上述表达式即满足 $\geqslant cn^2$ 的条件。（读者看出应取的 n_0 是多少吗？）

因此，我们确实证明了 $T(n)=\Omega(n^2)$。

证明 $T(n) = O(n^2)$的过程与上述过程类似。根据 $O(n^2)$的定义，我们只需要证明存在 n_0，对于 $n>n_0$，有 $T(n) \leqslant cn^2$，这里 c, n_0 为正的常数即可。

同样，假定上述条件对于所有小于 n 的输入都成立，则有：

$$T(n) = T(n/4)+T(n/2)+n^2 \leqslant c(n/4)^2+c(n/2)^2+n^2=cn^2-(11c/16-1)n^2$$

我们只要选择 $c \geqslant 16/11$，上述表达式即满足 $\leqslant cn^2$ 的条件。（你看出应取的 n_0 是多少吗？）

因此，我们又证明了 $T(n)=O(n^2)$。合并上述两个证明，我们有 $T(n)=\Theta(n^2)$。

这样，我们对根据递归树展开所获得的答案就有完全的信心了。

仔细观察上述归纳求证过程可以发现，我们是将一个假定的解答代入递归表达式，然后求解出最终结果，这种解法也被称为**替换解法**。根据上面的例子，我们看出使用替换解法的步骤如下：

1）猜一个答案。

2）使用归纳法对答案进行验证（根据猜测的答案形成归纳假设，然后对归纳假设进行验证）。

3）解决表达式中的常数。

当然，在猜测一个答案的时候，如果一眼看不出来，则可以使用递归树作为帮助。

例 3-1 求解递归表达式 $T(n) = 4T(n/2) + n$ 的解。

解　步骤如下：

1）由于表达式里面有 n 项，我们猜大一点，将解猜为 $O(n^3)$。

2）根据猜测获得归纳假设：对于任意 $k<n$，$T(k)\leqslant ck^3$。将该假设代入递归式里有：

$$\begin{aligned} T(n) &= 4T(n/2)+n \\ &\leqslant 4c(n/2)^3+n && \text{（归纳假设）} \\ &= (c/2)n^3+n && \text{（代数变换）} \\ &= cn^3-((c/2)n^3-n) && \text{（所期值-剩余值）} \\ &\leqslant cn^3 && \text{（所期望的值）} \end{aligned}$$

3）最后解决常数 c。如果第 2）步最后的不等式要成立，则必须满足$(c/2)n^3-n\geqslant 0$。而这个条件在 $c\geqslant 2$ 且 $n\geqslant 1$ 的时候成立。

这里需要注意的是，在上述证明过程中，我们没有考虑初始条件，而初始条件是归纳法能够成立的基础。那么上述归纳证明的初始条件是什么呢？当然是 $T(1)\leqslant c$。这个条件成立吗？当然，只要选择足够大的 $c\geqslant 1$ 即可。

由于大多数时候初始条件的满足显而易见，所以我们在证明时通常忽略这个步骤。我们只在初始条件的满足并不明显的时候才会单独对初始条件进行讨论。

至此，我们证明了 $T(n)=4T(n/2)+n$ 的上限为 $O(n^3)$，但这个上限对我们并没有太大帮助。因为范围太宽松了。该上限并不是我们讨论的递归表达式的一个最紧的上限！而一个不紧的上限有可能导致误解：我们可能认为 $T(n)=4T(n/2)+n$ 的效率是立方级的！而事实却不是。

那怎么办呢？当然是猜一个更紧的上限，如平方级。即猜 $T(n)=O(n^2)$。那么这个猜想是否正确呢？只要用归纳法来证明即知。设对于任意 $k<n$，$T(k)\leqslant ck^2$，将其代入递归式有：

$$T(n)=4T(n/2)+n\leqslant 4c(n/2)^2+n=cn^2+n=cn^2-(-n)$$

令人遗憾的是，$cn^2-(-n)$无论在什么情况下（无论怎么选择 c）也不会小于 cn^2。我们的证明似乎陷入了僵局。难道我们的猜想有问题吗？如果画一棵递归树的话，你会发现我们的猜想似乎是正确的。也许是我们的证明错了？

而改正的方法是增强我们的递归假设！即如果证明不了一个假设，那就证明一个更难的假设！这听上去似乎反直觉，但谁说过算法是一个靠直觉的学科呢？（有兴趣的读者可参阅《计算机的心智：操作系统之哲学原理》[⊖]一书对直觉与非直觉学科的讨论。）

我们的反直觉思维是这样的：重新设计归纳假设，使得其比前面的归纳假设更严格。新的归纳假设为对于所有 $k<n$，$T(k)\leqslant c_1k^2-c_2k$。在新的假设下，有：

$$\begin{aligned} T(n) &= 4T(n/2)+n \\ &\leqslant 4(c_1(n/2)^2-c_2(n/2))+n \\ &= c_1n^2-2c_2n+n \\ &= c_1n^2-c_2n-(c_2n-n) \\ &\leqslant c_1n^2-c_2n \end{aligned}$$

⊖ 该书已由机械工业出版社出版，书号为 ISBN 978-7-111-26642-6。

上述最后一行的不等式成立的条件是$(c_2n-n)>0$，而该不等式在 $c_2 \geqslant 1$ 的情况下就成立。当然了，我们还需要选择足够大的 c_1 来满足初始条件。（这个初始条件是什么？）

3.3.3 大师解法

替换解法虽然比递归树解法可靠，但也存在诸多缺陷。第一个缺陷是要将答案猜得不多不少并不容易。猜松了会导致误解，猜紧了又证明不出来。其次，即使猜测的答案正确，也不一定能够容易地证明。上面的例子已经说明了这一点。再次，这种方法只能针对具体的递归表达式，而不能对抽象的递归表达式进行求解。例如，对于 $T(n) = aT(n/b) + f(n)$的抽象递归式，替换解法就只能望洋兴叹了。

那有没有更好的办法，既能克服替换解法的缺点，又能对抽象的递归式进行一般求解呢？

我们先来将抽象的递归式进行展开看看能得到什么结果。

$$\begin{aligned}
T(n) &= aT(n/b) + f(n) \\
&= a^2T(n/b^2) + af(n/b) + f(n) \\
&= a^3T(n/b^3) + a^2f(n/b^2) + af(n/b) + f(n) \\
&= a^{\log_b n}T(1) + a^{\log_b n-1}f\left(\frac{n}{b^{\log_b n-1}}\right) + \cdots + a^2f\left(\frac{n}{b^2}\right) + af\left(\frac{n}{b}\right) + f(n) \\
&= n^{\log_b a}T(1) + \sum_{j=0}^{\log_b n-1} a^j f\left(\frac{n}{b^j}\right) \\
&= \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n-1} a^j f\left(\frac{n}{b^j}\right) \qquad (3\text{-}4)
\end{aligned}$$

到这一步，我们已经将抽象递归式分解为两大块：一部分是 $\Theta(n^{\log_b a})$，这是一个恒定的数量级 $n^{\log_b a}$，另一部分似乎是一个类似级数的求和。

而根据渐近表示的性质 3，两项之和的数量级小于两项之中的较大项的数量级。因此，我们只要将式（3-4）最右边一项与 $\Theta(n^{\log_b a})$比较，取其较大者即可。而对于式（3-4）最右边一项来说，a、b 都是常数，起主导作用的是函数 f 的渐近数量级。因此，我们只要比较函数 f 和 $\Theta(n^{\log_b a})$即可。

而对两个函数进行比较的结果只有 4 种可能：大于、小于、等于和不能相比。下面就这 4 种情况分别进行讨论。

1. $f(n)<\Theta(n^{\log_b a})$，即函数 f 的数量级小于 $\Theta(n^{\log_b a})$

在这种情况下，不失一般性（为什么），我们假定 $f(n) = O(n^{\log_b a-\varepsilon})$，这里 ε 是一个大于 0 的常数，也就是 $f(n)$比 $n^{\log_b a}$ 的渐近增长要慢，且慢一个因子 n^{ε}。

此时式（3-4）最右边的一项可以化简如下：

$$\begin{aligned}
\sum_{j=0}^{\log_b n-1} a^j f(n/b^j) &= \sum_{j=0}^{\log_b n-1} a^j O\left(\left(\frac{n}{b^j}\right)^{\log_b a-\varepsilon}\right) = O\left(\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\varepsilon}\right) \\
&= O\left(\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n^{\log_b a-\varepsilon}}{b^{j\log_b a-\varepsilon}}\right)\right) = O\left(\sum_{j=0}^{\log_b n-1} n^{\log_b a-\varepsilon}\left(\frac{a^j}{b^{j\log_b a-\varepsilon}}\right)\right)
\end{aligned}$$

$$= O\left(n^{\log_b a-\varepsilon}\sum_{j=0}^{\log_b n-1}\left(\frac{a}{b^{\log_b a-\varepsilon}}\right)^j\right) = O\left(n^{\log_b a-\varepsilon}\sum_{j=0}^{\log_b n-1}\left(\frac{ab^\varepsilon}{a}\right)^j\right) = O\left(n^{\log_b a-\varepsilon}\sum_{j=0}^{\log_b n-1}(b^\varepsilon)^j\right)$$

$$= O\left(n^{\log_b a-\varepsilon}\left(\frac{b^{\varepsilon\log_b n}-1}{b^\varepsilon-1}\right)\right) = O\left(n^{\log_b a-\varepsilon}\left(\frac{n^\varepsilon-1}{b^\varepsilon-1}\right)\right)$$

$$= O\left(n^{\log_b a-\varepsilon}n^\varepsilon\right) = O\left(n^{\log_b a}\right)$$

因此，有：$T(n)=\Theta(n^{\log_b a})+O(n^{\log_b a})=\Theta(n^{\log_b a})$。

2. $f(n)=\Theta(n^{\log_b a})$

此时式（3-4）最右面的一项可以化简如下：

$$\sum_{j=0}^{\log_b n-1}a^j f(n/b^j) = \sum_{j=0}^{\log_b n-1}a^j\Theta\left(\left(\frac{n}{b^j}\right)^{\log_b a}\right) = \Theta\left(\sum_{j=0}^{\log_b n-1}a^j\left(\frac{n}{b^j}\right)^{\log_b a}\right)$$

$$= \Theta\left(n^{\log_b a}\sum_{j=0}^{\log_b n-1}\left(\frac{a}{b^{\log_b a}}\right)^j\right) = \Theta\left(n^{\log_b a}\sum_{j=0}^{\log_b n-1}1\right) = \Theta(n^{\log_b a}(\log_b n))$$

因此，有：$T(n)=\Theta(n^{\log_b a})+\Theta(n^{\log_b a}(\log_b n))=\Theta(n^{\log_b a}(\log n)$。

实际上，这个条件我们还可以放宽到 $\Theta(n^{\log_b a}\log^k n)$（因为对数级离常数级的距离远小于其与线性级的距离），这里的 $k\geqslant 0$ 为一个常数。即 $f(n)$与 $\Theta(n^{\log_b a})$比较在一个对数级范围内波动。此时式（3-4）最右边的一项可以化简如下：

$$\sum_{j=0}^{\log_b n-1}a^j f(n/b^j) = \sum_{j=0}^{\log_b n-1}a^j\Theta\left(\left(\frac{n}{b^j}\right)^{\log_b a}\log^k\frac{n}{b^j}\right)$$

$$= \Theta\left(\sum_{j=0}^{\log_b n-1}a^j\left(\frac{n}{b^j}\right)^{\log_b a}\log^k\frac{n}{b^j}\right) = \Theta\left(n^{\log_b a}\sum_{j=0}^{\log_b n-1}\left(\frac{a}{b^{\log_b a}}\right)^j(\log^k n-\log^k b^j)\right)$$

$$= \Theta\left(n^{\log_b a}\sum_{j=0}^{\log_b n-1}(\log^k n-\log^k b^j)\right) = \Theta\left(n^{\log_b a}\left(\sum_{j=0}^{\log_b n-1}\log^k n-\sum_{j=0}^{\log_b n-1}\log^k b^j\right)\right)$$

$$= \Theta\left(n^{\log_b a}\left(\sum_{j=0}^{\log_b n-1}\log^k n-\sum_{j=0}^{\log_b n-1}\log^k b^j\right)\right)$$

$$= \Theta\left(n^{\log_b a}\left(\log^k n\log n-\sum_{j=0}^{\log_b n-1}\log^k b^j\right)\right) = \Theta(n^{\log_b a}\log^{k+1}n)$$

因此，有：$T(n)=\Theta(n^{\log_b a})+\Theta(n^{\log_b a}\log^{k+1}n)=\Theta(n^{\log_b a}\log^{k+1}n)$。

仔细的读者可能已经看出来，这是前面情况的一般化，或者前面是后面在 $k=0$ 的特例。

3. $f(n)$**的渐近增长趋势>**$\Theta(n^{\log_b a})$

在此情况下，由于 $\sum_{j=0}^{\log_b n-1}a^j f(n/b^j)>f(n)$（正数项的和大于和里面的每一项），因此有：

$$\sum_{j=0}^{\log_b n-1}a^j f(n/b^j)=\Omega(f(n))$$

又由于$f(n)>\Theta(n^{\log_b a})$，式（3-4）的时间效率将完全由$\sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$决定。因此，

$$T(n)=\Omega(f(n))$$

至此我们找出了递归表达式的下限。但仅仅找出下限是没有多大意思的，因为我们只知道该递归表达式的效率最好只可能是 $f(n)$。但我们更想知道的是 $T(n)$的上限。根据题意，$T(n)$的上限就是$\sum a^j f(n/b^j)$的上限。

但是，$\sum a^j f(n/b^j)$的上限是什么呢？

我们先来看看函数 f 代表的意思：f 代表的是分治策略中的分解（成子问题）和合并（子问题）的成本。由于$f(n)$的渐近增长趋势$>\Theta(n^{\log_b a})$，所以该分治策略的分解和合并成本高于子问题的解决成本，即分解和合并成本高于解决问题的成本。而如果在这种情况要获得解，分解和合并的成本应该逐级下降；否则，分解和合并成本随着分解的推进将呈发散态势，这样总成本有可能不会收敛。那么这种分治策略显然就没有什么意义了，或者说，我们的分治策略应用错了！

而如果分解与合并成本逐级下降，则意味着函数 f 满足 $af(n/b)\leqslant cf(n)$，这里 $c<1$ 是一个正常数。这时我们可以很快获得式（3-4）的上限如下：

$$\sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \leqslant \sum_{j=0}^{\log_b n-1} c^j f(n) \leqslant f(n)\sum_{j=0}^{\infty} c^j = f(n)\left(\frac{1}{1-c}\right)=O(f(n))$$

合并我们前面的下限解$\sum a^j f(n/b^j)=\Omega(f(n))$，我们有$\sum a^j f(n/b^j)=\Theta(f(n))$。因此，

$$T(n)=\Theta(n^{\log_b a})+\Theta(f(n))=\Theta(f(n))\text{（因为}f(n)\text{大于}n^{\log_b a}\text{）}$$

因此，我们的解为 $T(n)=\Theta(f(n))$，这也是我们所猜测的结果。（你预料到这个结果了吗？）

4. 无法比较

这种情况目前尚无一般规律可循，求解需要个人的灵感和聪明才智。

上面介绍的 4 种情况里面的 3 种情况都有规律可循，而这 3 种情况适用的情景很多，具有较大普遍性，可应用于各种分治场合，因此称为**大师解法**。如果使用递归树，则大师解法的直观意义就很明显了（如图 3-6 所示）。

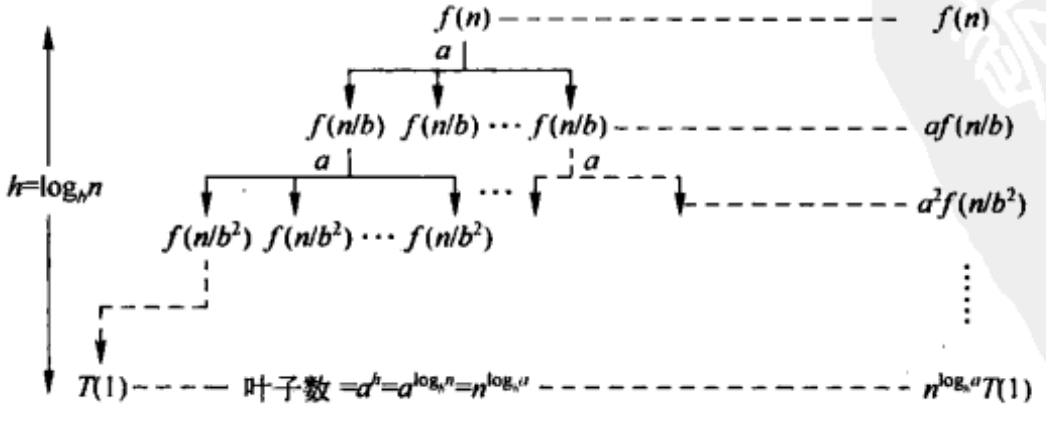


图 3-6 大师解法的递归树演示

大师解法的第一种情况代表的是递归树的每层成本从根至下呈几何级数增长，成本在叶子一层达到最高，即最后一次递归是整个递归过程中成本最高的一次。因此递归分治的总成

本在渐近趋势上与叶子层成本一样，即成本为 $\Theta(n^{\log_b a})$。

大师解法的第二种情况对应的递归树状态是每层成本一样（在渐近趋势上），即每层成本均为 $n^{\log_b a}$。由于一共有 $\log n$ 层，因此，总成本为每一个层的成本乘以 $\log n$，即 $\Theta(n^{\log_b a}\log n)$。

大师解法的第三种情况对应递归树的状态为每层成本呈几何级数递减，树根这一层的成本占主导地位。因此，总成本就是树根层的成本，即 $\Theta(f(n))$。

至此，大师解法就介绍完了。但细心的读者可能还发现，我们在证明大师解法的过程中只考虑了 n 被 b^i 整除的情况。那么不能整除的情况是否结论也是一样呢？这个问题就留给读者考虑吧。

3.4 分治策略举例1：乘方运算

乘方运算就是对一个数取若干次方的运算，也就是自己乘以自己若干次。问题的具体定义如下：对于 $n\in\mathrm{N}$，计算 a^n。

如果不假思索，直接进行 n 次乘法，则一共需要进行 n 次乘法。即这种天真的算法的效率是 $\Theta(n)$次乘法（每次乘法本身的效率在这里并不考虑，因为它不影响分析）。

但是如果仔细分析，我们发现：

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

这样，我们就可以应用分治策略，将 n 次方的乘方运算变为两个 $n/2$ 次方的乘方运算和一次普通的乘法运算（或两个$(n-1)/2$ 次方的乘方运算和二次普通的乘法运算）。如果再看仔细，发现分解出的两个 $n/2$（或$(n-1)/2$）次方的乘方运算实际上是同一个乘方运算！

因此，我们可以将这种分治策略的递归表达式写为：$T(n) = T(n/2) + \Theta(1)$。这里 $T(n)$为分治策略的效率，$\Theta(1)$表示每次分解后合并答案所需的时间（常数次乘法）。

从大师解法可以得出该递归表达式的解为：$T(n) = \Theta(\log n)$。由此可见，分治策略比起天真的直接解法效率要高得多！

3.5 生命中不能承受之重：矩阵乘法

矩阵乘法是另一个使用分治策略的好地方。由于科学计算中大量使用矩阵乘法，高效率地解决这个问题显然意义重大。下面我们先看看矩阵乘法的定义。

定义 给定两个矩阵 A、B，记 $A=[a_{ij}]$，$B=[b_{ij}]$ $(i,j=1,2,\cdots,n)$。那么 A、B 的乘积矩阵 $C=[c_{ij}]=AB$ $(i,j=1,2,\cdots,n)$，即

$$\begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{pmatrix}$$

记为：

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

如果使用天真的直接相乘的办法来计算，则矩阵乘法的算法如下：

```
for(i=1; i<=n; i++) {
    for(j=1; j<=n; j++) {
        cij=0;
        for(k=1; k<=n; k++) {
            cij=cij+ aik·bkj;
        }
    }
}
```

显而易见，这个天真的解法的时间复杂性是 $\Theta(n^3)$（以乘法和加法次数为单位）。

也许读者觉得三次方的矩阵乘法效率并不差，但是如果考虑到矩阵乘法在很多领域是一种非常频繁的操作，且在这些领域里，每个矩阵的规模巨大，那么，这种天真解法并不令人满意。

那有更好的办法吗？有，分治呀。由于直接矩阵乘法的时间复杂性高（$O(n^3)$），所以我们自然想到能否采取分而治之的策略来提高效率：即将两个将要进行乘法运算的矩阵划分为 4 个大小一样的小矩阵，分别求出这些小矩阵的相关乘积，再进行组合。

我们的思考是将一个 $n \times n$ 矩阵分解为 2×2 个$(n/2)(n/2)$ 的子矩阵，然后求得这些子矩阵的乘积后再进行组装而获得原来大矩阵的乘积。即

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} e & f \\ g & h \end{pmatrix}$$

$$C = A \cdot B$$

这样，原来的矩阵乘法可以改变为下面的 8 个子矩阵乘法和 4 个加法操作：

$$r=ae+bg$$
$$s=af+bh$$
$$t=ce+dg$$
$$u=cf+dh$$

而上述的 8 个子矩阵乘法又可以递归使用上述分治策略，从而获得递归表达式如下：

$$T(n) = 8T(n/2)+\Theta(n^2)$$

上面式子里的 $8T(n/2)$表示 8 个$(n/2)$大小的子矩阵乘法，$\Theta(n^2)$表示将子矩阵乘积相加的时间

复杂性。而根据大师解法：$n^{\log_b a}= n^{\log_2 8}= n^3 \Rightarrow$情况 $1 \Rightarrow T(n) = \Theta(n^3)$。

虽然这种分解办法将问题分解为 8 个大小为原来一半的问题，但使用大师解法后发现，这种办法的时间复杂性仍然为（$O(n^3)$）。难道分而治之策略在矩阵乘法上派不上用场吗？

分治错了吗？

答案取决于你能否将递归式子里面的 8 个子矩阵乘法减少一个或几个。如果能，则分治就有效果；如果不能，则分治就是多此一举。

幸运的是，确实可以将递归式子里面的 8 降低到 7。仔细分析发现，实际上可以用 7 个子问题来取代正常分解下需要的 8 个子问题，即分解后只需要进行 7 次子矩阵的乘法运算，而不是原来的 8 次。这个发现由德国数学家沃卡·斯特劳森（Volker Strassen）所做出。下面给出这个从 8 降低到 7 的过程。

首先，进行如下 7 个子矩阵乘法运算：

$$P_1= a(f-h)$$
$$P_2= (a+ b)h$$
$$P_3= (c+ d)e$$
$$P_4= d\,(g-e)$$
$$P_5= (a+ d)(e+ h)$$
$$P_6= (b-d)(g+ h)$$
$$P_7= (a-c)(e+f)$$

其次，对前面的中间结果再进行加减运算，即获得最后的结果如下：

$$r = P_5+P_4-P_2+P_6$$
$$s = P_1+P_2$$
$$t = P_3+P_4$$
$$u = P_5+P_1-P_3-P_7$$

例如：

$$\begin{aligned} r &= P_5+P_4-P_2+P_6 \\ &=(a+d)(e+h)+d(g-e)-(a+b)h+(b-d)(g+h) \\ &=ae+ah+de+dh+dg-de-ah-bh+bg+bh-dg-dh \\ &=ae+bg \end{aligned}$$

这样，一共需要 7 次乘法、18 次加法和减法。递归表达式改变为：

$$T(n) = 7T(n/2) + \Theta(n^2)$$

按照大师解法，有：$n^{\log_b a}= n^{\log 27} \approx n^{2.81} \Rightarrow$情况 $1 \Rightarrow T(n) = \Theta(n^{\log 7}) \cong \Theta(n^{2.81})$ 。

也许 2.81 与 3 比较起来没有多大区别，但要想到这个数字出现的位置是幂上面，其所带来的效率改善是非常可观的，尤其是在 n 很大的时候！事实上，在 $n \geqslant 32$ 时，斯特劳森的算法就已经超越了天真的直乘法。

由于矩阵乘法广泛应用在天文、地理、气象等诸多方面，所以人们一直持久地努力改善矩阵乘法的效率，目前最好的算法能够达到的效率为 $\Theta(n^{2.376\cdots})$ 。不过这个数量级只在理论上成立或者只针对某种特定规模的矩阵乘法才成立，在实际的计算机上针对通用的矩阵乘法

规模运行时，则通常不能实现这个效率。

3.6 魔鬼序列：斐波那契序列

公元 1202 年，比萨的列奥纳多（Leonardo of Pisa），又称为斐波那契（Fibonacci）在他的《算经》（Liber Abaci）一书里谈到了斐波那契序列。这个序列是因为描述理想状态下兔子的数量增长而得出的（见图 3-7）。该兔子增长模型如下：

原点 一对（雌雄）兔子。

规律 每对兔子每个月产下一对兔子，且一生只能产两次，并且在第二次生产后老死。

1）第 0 个月，只有一对（雌雄）兔子。而额外的兔子对数为 0（相对于原始兔子对数而言）。

2）第 1 个月，这对初始的兔子生下一对小兔子，即额外的兔子对数为 1（相对第 0 个月）。

3）第 2 个月，两对兔子各生下一对兔子，最先的那对兔子老死。额外的兔子对数为 1（相对于第 1 个月）。

4）第 3 个月，剩下的三对兔子各生下一对兔子，最老的一对兔子老死，额外的兔子对数为 2（相对于第 2 个月）。

……

如果定义第 n 个月的兔子数量为 $F(n)$。由于这个时刻只有在第 $n-2$ 个月还活着的兔子才能产下后代，因此，$F(n-2)$对兔子加上当前的兔子数量 $F(n-1)$就是第 n 个月的兔子数量，即 $F(n) = F(n-1) + F(n-2)$。

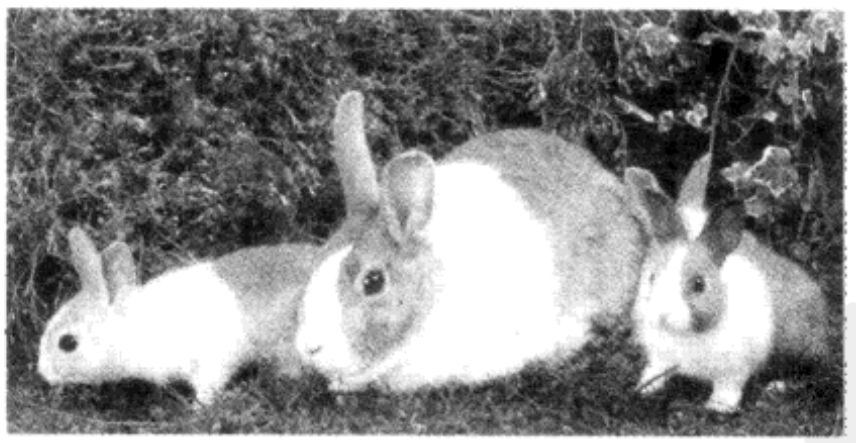


图 3-7 斐波那契序列与兔子的数量存在着密切的关系

斐波那契序列是一个非常有趣且故事很多的序列。有人说，它是一个和谐完美的数列；有人说，其实斐波那契序列并不是由列奥纳多首次提出的，它在古印度就已经存在了，当时被应用在诗文的韵律上；又有人说，它是鹦鹉螺壳上面的纹路的数字体现；还有人说，它趋向于黄金分割；还有人说，它是魔鬼序列，里面隐藏着魔鬼（即撒旦）引诱人类犯罪的证据。各种说法扑朔迷离，不一而足。

我们看一下该序列的前几项以点数来排开的效果，如图 3-8a 所示。你觉得很优美吗？如果我们将其排成图 3-8b 的样子，你又能看出何种名堂？什么也没看出来？不会吧。仔细看！

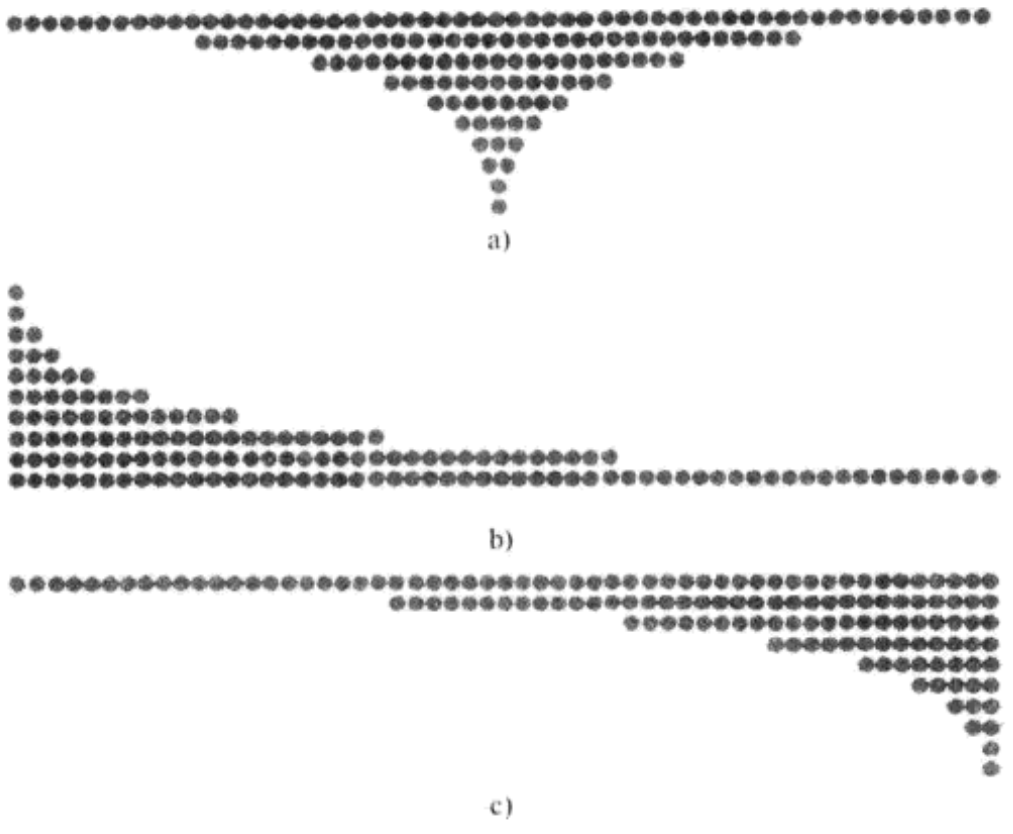


图 3-8 斐波那契序列的各种图示

当然，本书不去探讨该序列的种种神奇传说及其与魔鬼的千丝万缕的联系。我们关心的是如何求取该数列的任意一项的值。例如，F_{100}。

这个问题也许看上去稀松平常。求取 F_{100}，那不是很简单吗，它等于 $F_{99}+F_{98}$ 呀！那 F_{99} 和 F_{98} 的值是什么呢？简单呀，分别是 $F_{98}+F_{97}$ 和 $F_{97}+F_{96}$。这样一直递归下去，直到 F_1 和 F_0，不就获得 F_{100} 的值了吗？

这个方法从概念上看倒是不错，但这个算法的效率是多少呢？

将这个问题推广，考虑 F_n 的计算。

显而易见，上述递归要一直到 F_0 和 F_1 才结束。因此，要获得此算法的效率，就需要知道有多少个 F_1 和 F_0。由于 F_1 不再需要递归，而比 F_1 大的 F_2 递归时会同时产生 F_1 和 F_0，因此，F_1 的个数要多于 F_0。因此 2 倍 F_1 的个数要多于 F_1 和 F_0 个数之和。因此，上述算法的效率完全由 F_1 的个数决定。那么这种递归算法会产生多少个 F_1 呢？

由于递归一直到 F_1 和 F_0 结束，而 F_0 的值为 0，F_1 的值为 1，因此，递归产生的 F_1 个数就是 F_n 的值。即上述算法的时间复杂性就是 F_n。那么 F_n 是多少呢？

将斐波那契的递归表达式重新表述：

$$F(n+2) = F(n+1) + F(n)$$

则可以得出该线性递归公式的特征方程为 $x^2=x+1$。解该特征方程可得到它的两个根为：ϕ和 $1-\phi$。这里$\phi=1+\frac{1+\sqrt{5}}{2}$。（看出来这是什么东西吗？）

将 F_n 用上面的两个方程根表示有：

$$F_n = y\phi^n + z(1-\phi)^n$$

将 F_0 和 F_1 的值代入，可获得 $y=1/\sqrt{5}$，$z=-1/\sqrt{5}$。

因此，我们获得 F_n 的通式为：$F_n=\frac{1}{\sqrt{5}}[\phi^n-(1-\phi)^n]$。

由于 $\phi>1-\phi$，当 n 趋向无穷时，ϕ^n 会在渐近趋势上大于 $(1-\phi)^n$，即 n 趋向无穷时，我们有

$$F_n\approx\phi^n/\sqrt{5}$$

因此，递归解法需要计算的 F_1 的次数为指数次方，这就是天真递归解法的效率，非常低！

那我们可以对此加以改进吗？下面我们给出三种改进方式。

3.6.1 由底至上

当然，天真的递归算法里重复计算了很多的 F_1 和 F_0，如果想消除这些重复计算，可以由底至上，从 F_0 和 F_1 开始，按照 F_0、F_1、F_2、…、F_n 的顺序一个个往上计算，这样就可以避免重复。显然，这种算法的效率就是 $\Theta(n)$（n 次计算）。这是一个很大的效率改进！

3.6.2 使用通式

当然，既然我们已经确定 $F(n)$的通式，为什么不直接使用通式来计算 F_n 呢？由于要计算 n 次方，该算法的效率仍为 $\Theta(n)$（次乘法运算）。与由底至上的效率一样！

当然，我们还可以利用前面讨论过的乘方运算的优化算法，即递归求平方，可将算法效率提高到 $\Theta(\log n)$（次乘法运算）。与由底至上的效率相比，有很大的提高！

而这种办法的问题是通式中有无理数，有可能导致精度问题。这里需要注明的是，如果由人使用通式计算 F_n，不会产生任何精度问题。这是因为通式中的无理数 $\sqrt{5}$ 都会在代数演变中被约去，获得的数全部是精确的整数！但是我们不太可能由人工来计算 F_{100}、$F_{1\,000}$ 等。这太烦琐了，谁愿意去求一个式子的 100 次、1 000 次方呢？

因此，我们还是要求助于计算机。这样问题就来了：计算机不会进行约去操作，它只能将式子的数值直接按浮点数进行运算。从而就会带来精度问题！

那还有更好的算法吗？直白地说，我们需要的是既有精度，又有效率（$\Theta(n)$）的算法。

3.6.3 使用矩阵乘方

使用矩阵乘方？矩阵乘法不是效率很低的一种运算吗？没错。矩阵乘法的运算效率是低，但前提是矩阵的维数不断增大。如果矩阵的维数保持在一个很小的不变值上面，如 2，则这种矩阵的乘法就和一般的数的乘法的效率在一个数量级上！

那么使用哪一个矩阵呢？由于斐波那契数列是一个二阶递推数列，所以存在一个 2×2 维的矩阵 A，使得

$$(F_{n+2},F_{n+1})=(F_{n+1},F_n)\times A$$

代入斐波那契数列的前几项值，可获得 A 为 $\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}$，即

$$(F_{n+2}, F_{n+1}) = (F_{n+1}, F_n) \times \begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix} = (F_n, F_{n-1}) \times \begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^2 \cdots = (F_1, F_0) \times \begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^{n+1}$$

当然，也可以将表达式写成另外一种方式：$\begin{pmatrix}F_{n+1} & F_n\\ F_n & F_{n-1}\end{pmatrix} = \begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^n$。

这样，求解 F_n 就变成求解矩阵 A 的 n 次方！而根据我们前面获得乘方运算结果可知，该操作的效率是 $\Theta(\log n)$。更为重要的是，矩阵 A 的 n 次方的计算结果没有浮点误差！

3.7 VLSI 布线

在本章结束之前，我们在看一个实际中经常遇到的问题：VLSI 布线问题。布线是集成电路设计里一个重要的方面。布线的时候要考虑的因素很多：线与线之间的间距、输入线的散入限制、输出线的散出限制、线的最大长度限制等。

这里有一个故事，讲的是某大学生初到一家公司上班的时候，老板给他布置了一个芯片布局设计的任务，并问他要多长时间可以设计出来。这个学生不假思索地说：一个星期！老板一听，惊奇得很，心想国内名牌大学的学生就是厉害，一个星期就能将一个复杂的芯片布局设计搞定！

一个星期后，这个大学生将设计图交给了老板。老板一看，什么也没说，就让他离开公司了。原来这个大学生设计的图纸没有考虑到线与线之间的间距、散入散出限制、散热以及晶片面积最小化等问题，而只是画了一张图将各种器件按照逻辑关系连接起来而已！

好了，闲话少说，VLSI 布线的问题就是要在一个尽量小的空间面布置好各种元件，并且达到散入、散出、散热、间距等要求。例如，为了使布线工艺尽量简单，我们要对芯片里面元器件的布局求为一棵完全二叉树的形式。这样我们的 VLSI 布线问题就变成如何将一个有 n 个叶子的完全二叉树放置在一个空间尽量小的网格里面？

一个天真、直观的布线就是将二叉树原封不动地挪移到网格上，形成如图 3-9 所示的布局。

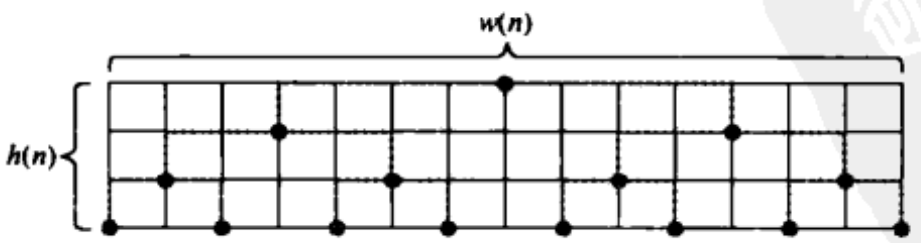


图 3-9 完全二叉树在芯片晶格里面的直接排列方式

此种布局占用的宽度和高度分别为 $w(n)$和 $h(n)$。关键是这个宽度和高度分别是什么呢？从布局图看到，一个叶子结点数位为 n 的完全二叉树需要占用的高度是 $h(n)$是一个叶子结点数为 $n/2$ 的完全二叉树的高度加 1，即

$$h(n)= h(n/2) + \Theta(1)= \Theta(\log n)$$

而宽度 $w(n)$则是一个叶子结点数为 $n/2$ 的完全二叉树的宽度的 2 倍加 1，即

$$w(n)= 2\ w(n/2) + \Theta(1)= \Theta(n)$$

这样，此种布局所占的空间大小为 $\Theta(n\log n)$。

这是最优的布局吗？

显然不是，因为很多空间都被浪费掉了：如图 3-9 中最上面一行只有一个结点，其他结点都空着！而第二行只有 2 个结点，其他结点都空着！

很显然，这个问题非常适合使用分治的办法：将 n 个叶子结点的完全二叉树分解为两个 $n/2$ 叶子结点的完成二叉树，对这两个子完成二叉树进行递归布线，然后将结果合并即可。

这样分治的结果将形成如图 3-10 所示的布局。

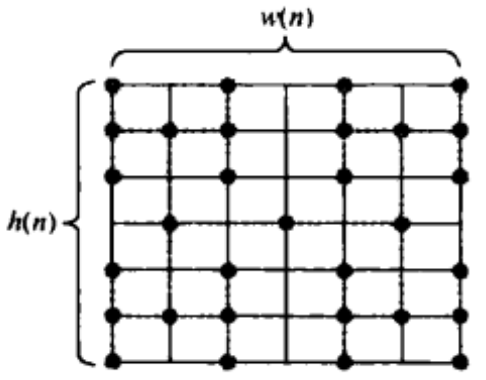


图 3-10 完全二叉树在芯片晶格里面的递归排列方式

该布局的形状是一个正方形，即 $w(n)=h(n)$。正方形的每个边的长度可由下面的递归表达式表示：

$$w(n) =h(n)= 2w(n/4) + \Theta(1)$$

根据大师解法，得 $w(n) = h(n) = \Theta(\sqrt{n})$。因此，整个布局所占空间为：$\Theta(n)$。这是一个对数级的改善！

3.8 多项式乘法

本章讨论的整数的乘法问题可以扩展到浮点数和多项式上，即两个多项式相乘也可用通过分治来降低算法的复杂性。但这种分治牵扯到多项式的点值表示与傅里叶变换。有兴趣的而读者可参阅文献[1]的相关章节，本书这里不再赘述。

3.9 分治就在潜意识

分治并不是一个特定的算法，而是一种算法思想。这个思想来源于人们偏向处理简单的事情，因为简单的东西比复杂的东西好应付。而且现实世界确实由简单的东西构成：如果将物质不断细分，发现其最后的构成都是一样的。因此，分治策略可以说植根在人们的潜意识深处！

与分治策略不可分割的一个概念就是递归。没有递归，分治也就无从落地，而成了空中楼阁。因此，如何求解递归就成为分治策略的根基。从某种程度上说，分治与递归互为因果。

思考题

1. 请给出如何将两个 n 位数的乘法分解为 3 个 $n/2$ 位的乘法运算。
2. 按照我们的分治方法，计算一个长度为 n 位的数的立方的效率。
3. 对于表达式 $T(n)=4T(n/2)+n$，有同学是这样证明该递归表达式的解为 $T(n)=O(n^2)$的：
   对 $k<n$，设 $T(k) \leqslant ck^2$，则有
   $$T(n)=4T(n/2)+n \leqslant 4c(n/2)^2+n=cn^2+n=O(n^2)$$
   证毕。请问该同学的证明有什么问题吗？请予以解释。
4. 平方运算与乘法运算的不同点是乘数和被乘数相同，这点不同对我们在设计算法时有何影响？平方运算能否比普通乘法运算快呢？这里假定所有操作数的长度为 n 位。
5. 分治策略一定包含递归吗？如果是，解释为什么；如果不是，给出一个不包含递归的分治例子，并阐述这种分治和包含递归的分治的主要不同。
6. 给定 n 个球，其中 1 个球为次品。次品从外表上看与正常球一样，但重量有区别。它可能比正常球重，也可能比正常球轻。给你一个天平，需要称几次就能将次品甄别出来？注意，你的思路应该比本章开篇的思路更加高效。
7. 请问第 9 题里面的策略包含着分治的思想吗？为什么？
8. 我们说过分治通常都包含着递归，但是否递归也通常包含着分治呢？为什么？
9. 一次大型派对的最后节目是选出一位幸运人士，该人士将获得派对组织者准备的一个钻石戒指。而选择幸运人士的办法是让所有人员一字排列，然后从左至右点数，凡是奇数号的全部剔除。对于剩下的人员，又从左至右点数，逢奇数号就剔除。如此不断递归下去，直到只剩下一人为止。此人即为幸运之人。请设计一个递归算法计算幸运之人所在的位置，并分析该算法的时间效率。
10. 请给出一个计算整数 x 和 y 的最小公倍数的算法，并分析其时间成本。
11. 请问计算第 n^2 个斐波那契数 F_n^2 能够在多少时间内完成？是 $O(\log n)$吗？
12. 递归在编译器的设计中随处可见，请举出一个例子，并说明编译器设计是如何解决此种递归的。编译器的解递归方法是分治吗？为什么？
13. 在本章的例子 3-1 证明 $T(n)$的复杂性为 n^2 时，我们在归纳证明失败时，选择的是增强归纳前提的限制来予以证明。我们为什么不是选择降低归纳前提从而使证明更容易呢？按理说，归纳前提的增强应该增加证明的难度，但实际情况却恰恰相反，这是为什么呢？
14. 给定 n 个瓶子，每个瓶子里装着 m 个球，其中一个瓶子里面的球全部是次品。设正品球的重量为 x 克，次品球的重量为 $x+1$ 克。又设我们有一杆秤，秤的最小识别单位为克，秤的最大称重为无限。请问需要秤多少次，就能将装着次品球的瓶子找出来？你的算法战略是什么？

15. 假定某种绳索由于材质不均匀而导致燃烧时速度时快时慢，但如果将一根绳索点燃后，烧完所需的时间刚好是 n 分钟。问：

（a）能否用这种绳索度量 $n/2$ 分钟？

（b）能否用这种绳索度量 $n/2^m$ 分钟的时间？这里 m 为非负整数，如果能，如何实现？需要使用多少根这样的绳索？如果不能，试说明理由。

16. 有同学提出，大师解法的第 2 种情形和第 3 种情形实际上是一回事。因为 $f(n)=\Theta(n^{\log_b a}\log^k n)>\Theta(n^{\log_b a})$。因此，如果将第 3 种情形下的 $af(n/b)\leqslant c(f(n))$的条件移植过来，则第 2 种情形下的 $T(n)$的解将与第 3 种情形一样，这样就与第 2 种情形得出的原始结论 $\Theta(n^{\log_b a}\log^{k+1} n)$ 产生了矛盾。请问你如何调和这个矛盾。

17. 我们在讨论 $f(n)<\Theta(n^{\log_b a})$ 的情况时，说不失一般性可以假设 $f(n)=\Theta(n^{\log_b a-\varepsilon})$，为什么？

PART TWO

第二篇　算法设计篇

CHAPTER 4

第 4 章　动态规划思想

本书第 3 章阐述了算法中的分治思想。该思想的核心是将复杂的问题分解为小的、简单的问题，解决这些小问题，然后将小问题的解答合并成为原来大问题的解。

施行分治策略是基于以下几点认识:

1）小问题比大问题更容易解决。

2）将小问题解答组装成大问题解答所需要的成本低于直接解决大问题的成本。

3）小问题又可以按照同样方法分解为更小的问题。

分治策略在很多问题上给我们带来了高效的解决方案。但我们所讲到的分治策略是比较简单的、直接的，并没有对子问题的属性进行分析，从而丧失了对某些子问题属性加以利用的机会。

2009 年 12 月 26 日，耗时 4 年半、耗资 1 200 亿元的武广客运专线开通营运。该线路沿着原来的京广铁路武汉 —— 广州段途经的城市行走，并沿途新建所有车站。这样，沿途的每个城市就有两个车站：一个为原来的京广铁路车站；一个为武广客运专线的新车站。每个城市的两个车站间都有公交系统相连接。两条线路都有列车行走。这样，乘客从武汉到广州就可以选择两条线路中的任意一条，并且在任何中间站可以选择换到另一条铁路上，如图 4-1 所示。

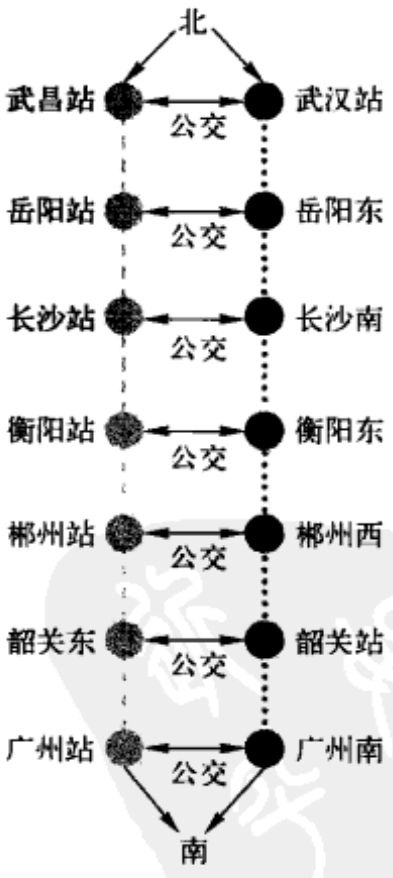


图 4-1　从武汉到广州的两条线路：黑线为武广客运专线，灰线为原来的京广线南段

显然，走新的武广客运专线将比走原京广线更快。但正是由于速度快，窗外的风景将一闪而过，难以欣赏。而走原京广线速度慢，但可以欣赏窗外的风景。如果欣赏风景那令人愉悦的感觉可以与时间进行相抵，那么走原京广线反而会比走武广客运专线快。例如，从长沙到衡阳走客运专线所花时间为 50 分钟（见图 4-2），而走原京广线花费时间为 90 分钟，但由于沿途美景而抵掉 45 分钟时间（例如在衡阳车站北面 30 公里处铁轨两侧，可以欣赏到推马救列车的欧阳海同志纪念碑和伟大领袖毛主席的“为人民利益而死，就比泰山还重”的苍劲题词），因此走原京广线反而在心理时间上只有 45 分钟，即比走武广客运专线更快。

如果每对相邻城市间走高速铁路和走普速铁路的时间都已知，并且走普速铁路的心理时间抵掉值也已知，那么乘客要做的决策就是如何选择行走线路而获得最佳的心理时间成本。具体来说，就是乘客在每个中间站点都可以做出如下两种选择之一：（a）继续当前线路前往下一站；（b）乘坐公交到同城的另一个车站，从另一条线路继续前行。不同的选择将导致不同的心理时间成本，而一个城市两个车站之间的公交交通时间也有不同。乘客应该如何选择是好呢？也就是乘客在到达每个车站后都要考虑是否需要改乘另一条线路，从而以最短心理时间达到终点。

我们用什么样的办法来给乘客进行规划呢？

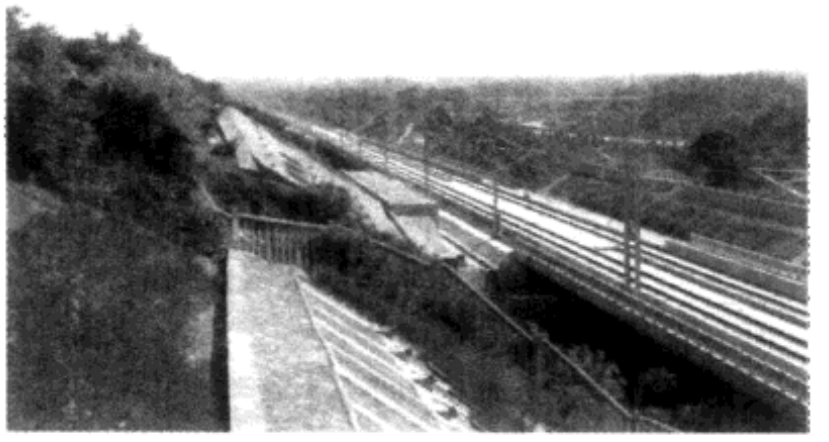


图 4-2 武广客运专线衡阳段

一种很直接的解答是尝试所有的可能，然后选择时间最短的方案。显然，这种方式的时间复杂性很高：在每个城市有 2 种选择，这样一共有 2^6 种可能。我们需要计算这 2^6 种可能线路的每一条线路所需的时间。计算 2^6 种可能也许计算机还能胜任，但如果这个 n 很大呢？

本书前面章节已经讨论过，当 n 很大时，2^n 的渐近增长趋势非常快！

那用什么办法呢？

回想第 3 章计算斐波那契数时，标准的分治策略是按照其递归定义进行求解，将更大的一项分解为小的两项，这样一直递归到 T_0 和 T_1。但这种分治的效率并不高，由于需要多次重复计算同样的项，所以其渐近趋势为指数级。但如果采用由底至上的策略，从 T_0 和 T_1 开始，按照 T_0、T_1、T_2、…、T_n 的顺序一个个计算，这样就可以避免重复，获得线性级效率！而这种由底至上构建大问题解答的方案就是本章要论述的动态规划策略！

4.1　什么是动态规划

由前面的论述可知，动态规划使用的就是分而治之的策略。只不过它比一般的分而治之策略更加高明，或者说它是一种更加有针对性的分而治之策略。

此种策略由理查德·贝尔曼（Richard E. Bellman）于 1957 年在其著作《Dynamic Programming》（动态规划）一书中提出。同年，贝尔曼和莱斯特·福特（Lester Ford Jr.）一起设计了一个求图里面最短路径的 Bellman-Ford 算法。该算法克服了 Dijkstra 最短路径算法的缺陷，能够应对负权重的存在。本书将在第 12 章中对 Bellman-Ford 算法进行详细论述。

那么，动态规划的高明之处在什么地方呢？我们先来看一下动态规划的英文名字：Dynamic Programming。这个名字听上去似乎与我们所熟知的程序设计有关。不过，如果读者这么想，那就错了。在这里，编程（programming）与程序设计没有任何关系，而是指表格查询法（tabular method），即将每一步计算的结果存储在表格里，供随后的计算查询使用。

说到这里，动态规划与本书第 2 章讨论的分而治之策略之间的不同已经清楚了。即在动态规划下，我们将问题分解为小问题，**但分解的小问题有许多是重复的**，这样，用表格将已经计算出的结果存起来可以节省重复计算，从而降低时间复杂性。

但这一点还不是动态规划的全部。动态规划被提出来的主要目的是优化，即我们不只是要解决一个问题，而是要以最优的方式解决这个问题，或者说，针对特定问题寻求最优解。

那么动态规划有什么特点呢？

我们来看一下斐波那契数列的计算，如果不适用通式（假定我们没有发现通式），则 F_n 的计算依赖前面两项的计算。如果计算 F_n 的方式最优，则意味着前面两项的计算方式也应该最优。这样又推论到再前面的项的计算也要最优。因此，我们必须以最优的方式计算出前面的每一项才能够以最优方式计算出我们需要的项。而由底至上恰恰是最优的，因为每个项只需要计算一次，还有什么比一次更优的呢？

由此，动态规划的特点如下：

1）分析一个最优解决方案应该具备的结构。

2）递归定义最优解决方案。

3）由底至上构建一个最优解决方案。

4.2　流水线问题

本章最前面的例子通常以一个在计算机科学中更为一般化的流水线问题来表述：一台计算机有两条流水工作线（这个问题也可以扩展到 3 条或 3 条以上流水线的情况），每条流水线有 n 个梯级，分别为 $S_{1,1},\cdots,S_{1,n}$ 和 $S_{2,1},\cdots,S_{2,n}$。两条流水线上相应位置上的梯级 $S_{1,j}$、$S_{2,j}$ 所完成的任务一样，但完成的时间不同，分别为 $c_{1,j}$、$c_{2,j}$。指令从内存发射到流水线的时间分别为 $c_{1,0}$、$c_{2,0}$，从流水线上流出的时间分别为 x_1、x_2，如图 4-3 所示。

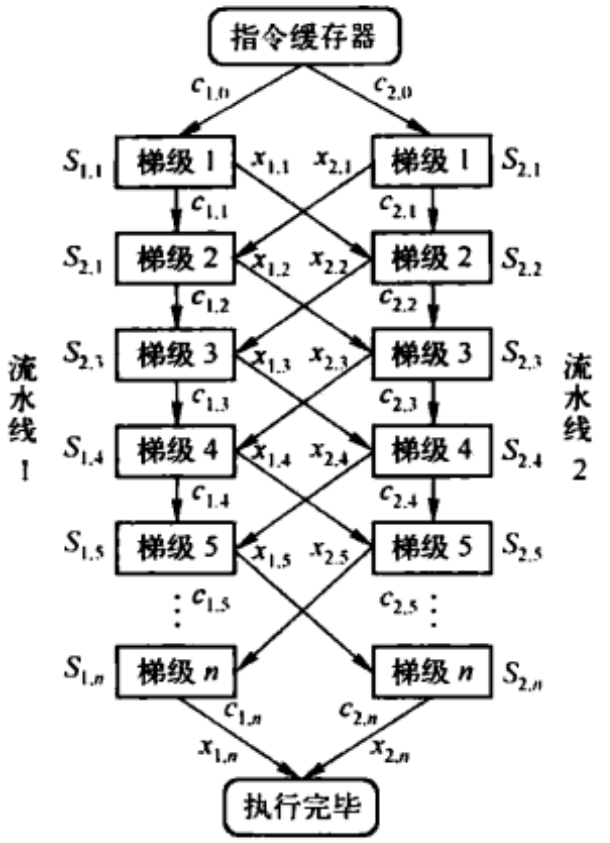


图 4-3 计算机流水线

在经过一个梯级 $S_{i,j}$ 后，一条指令可以：

1）前进到同一条流水线上的下一个梯级，不产生任何流动成本。

2）转移到另一条流水线上的下一个梯级，产生流动成本 $x_{i,j}$。

我们的问题是，给定上述所有的成本，我们应该选择流水线 1、2 上的哪些梯级才能使得指令从进入流水线到流出流水线的时间最短？

一种很直接的解答是尝试所有的可能，然后选择时间最短的方案。显然这种方式的时间复杂性为指数级 2^n（以梯级数量作为输入参数）。一般的分而治之的办法也不行（读者可作为练习来证明，使用一般的分而治之策略将产生指数级时间复杂性）。

那么办法是什么呢？你猜到了吗？动态规划。

下面我们看一下这个问题是否适合使用动态规划，首先考虑最优解的结构。

我们考虑从出发到梯级 $S_{1,j}$ 的最快的路线，可以分为两种情况：

1）如果 $j=1$，则只需要决定经过梯级 $S_{1,1}$ 需要使用多长时间即可。

2）如果 $j \geqslant 2$，则到达梯级 $S_{1,j}$ 又有两种路线选择：

- 从 $S_{1,j-1}$，过来，直接进入 $S_{1,j}$。
- 从 $S_{2,j-1}$,过来，经转换进入 $S_{1,j}$。

假定最快的路线是由 $S_{1,j-1}$ 过来，则我们注意到：从出发到 $S_{1,j-1}$ 的路线必须是最快的路线；否则，我们可以使用另一条更快的路线到达 $S_{1,j-1}$ 再进入 $S_{1,j}$，从而获得一条更快的从出发到梯级 $S_{1,j}$ 路线。

假定最快的路线是由 $S_{2,j-1}$ 过来，则我们注意到：从出发到 $S_{2,j-1}$ 的路线必须是最快的路线；否则，我们可以使用另一条更快的路线到达 $S_{2,j-1}$ 再转换进入 $S_{1,j}$，从而获得一条更快的从出发到梯级 $S_{1,j}$ 路线。

一般来说，一个问题的最优解决方案里面包含了对子问题的最优解。即从出发到梯级 $S_{1,j}$ 的最优方案包括了从出发到梯级 $S_{1,j-1}$ 或者梯级 $S_{2,j-1}$ 的最优方案。这就是我们所说的**最优子结构**。

有了最优子结构，我们就可以从子问题的解答构建原问题的解答。

到达梯级 $S_{1,j}$ 最快的路线为以下二居其一：

- 到达梯级 $S_{1,j-1}$ 最快的路线，然后直接进入梯级 $S_{1,j}$。
- 或者到达梯级 $S_{2,j-1}$，最快的路线，然后从流水线 2 转换到流水线 1 进入梯级 $S_{1,j}$。

对称地，我们有到达梯级 $S_{2,j}$ 最快的路线必为以下的二居其一：

- 到达梯级 $S_{2,j-1}$ 最快的路线，然后直接进入梯级 $S_{2,j}$。
- 或者到达梯级 $S_{1,j-1}$,最快的路线，然后从流水线 1 转换到流水线 2 进入梯级 $S_{2,j}$。

因此，寻找到达梯级 $S_{1,j}$ 和梯级 $S_{2,j}$ 的最快路线就转换为递归寻找到达梯级 $S_{1,j-1}$ 和梯级 $S_{2,j-1}$ 的最快路线。

如此，我们获得递归解决方案如下。设：$t_i[j]$ = 通过 $S_{i,j}$，（$i=1,2$ 和 $j=1,\cdots,n$）的最短时间，t^*=通过整个流水线的最短时间，则有：

$$t^* = \min(t_1[n] + x_1,\ t_2[n] + x_2)$$

$$t_1[1] = c_{1,0} + c_{1,1}$$

$$t_2[1] = c_{2,0} + c_{2,1}$$

并且，对于 $j=2,\cdots,n$，我们有：

$$t_1[j] = \min(t_1[j-1] + c_{1,j},\ t_2[j-1] + x_{2,j-1} + c_{1,j})$$

$$t_2[j] = \min(t_2[j-1] + c_{2,j},\ t_1[j-1] + x_{1,j-1} + c_{2,j})$$

这里，$t_i[j]$ 给出的就是最有解决方案的值（最优方案需要的最短时间）。

我们不仅要知道最优方案需要多少时间通过流水线，而且我们还需要知道这个最优方案到底是什么，到底通过哪些梯级。那么如何构建这个最优方案呢？

很简单，只需要再增加一个变量即可。设 $l_i[j]$ =到达 $S_{i,j}$ 最快路线上梯级 $j-1$ 所处的线路（1 或 2）。也即在通过梯级 $S_{i,j}$ 的最快路线上，梯级 $S_{l_i[j],j-1}$ 在梯级 $S_{i,j}$ 之前，这里的 $i=1,2, j=2,\cdots,n$。再设 l^*= 最后梯级的流水线号。

例如，表 4-1 和表 4-2 的取值给出的就是图 4-4 中那条阴影的最快线路。

表 4-1　通过梯级 $S_{i,j}$，$i=1,2$，$j=1,\cdots,n$ 的最短时间

j	1	2	3	4	5	6	7
$t_1[j]$	4	9	17	24	22	24	30
$t_2[j]$	6	8	14	17	20	23	31

表 4-2　最快通过流水线使用的梯级

j	2	3	4	5	6	7
$l_1[j]$	1	1	1	2	1/2	1/2
$l_2[j]$	2	2	2	2	2	2

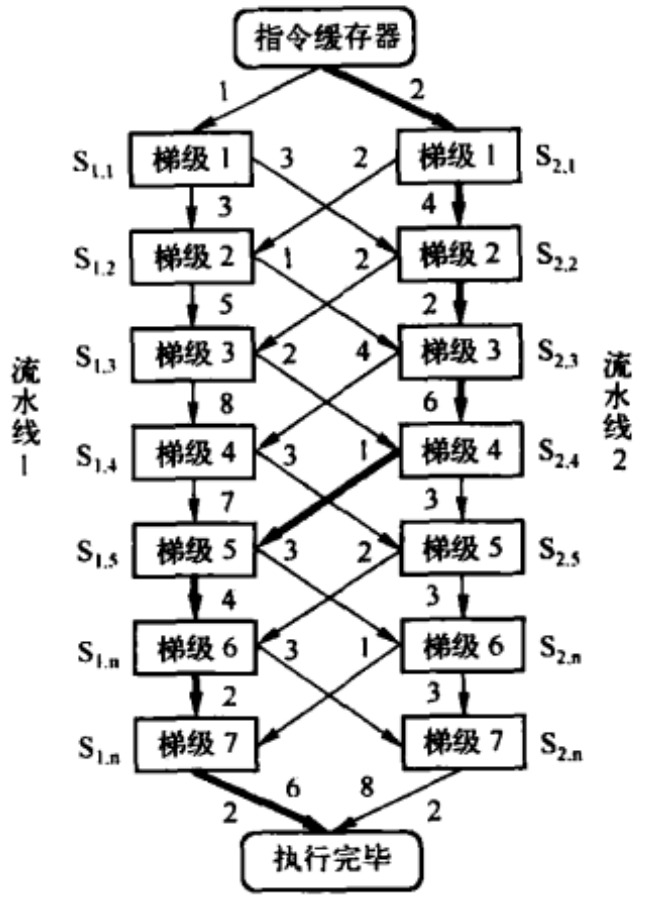


图 4-4 计算机流水线的一条最优线路（粗线条）

计算最优解决方案

根据上面推出的递归关系，可以直接写出一个递归算法如下，但该算法的效率却不高。如果设 $r_i(j)$=该递归算法访问 $t_i[j]$的次数，则有：

$$r_1(n)=r_2(n)=1$$

$$r_1(j)=r_2(j)=r_1(j+1)+r_2(j+1),\ j=1,\cdots,n-1$$

我们称 $r_i(j)=2^{n-j}$。

证明 在 j 上递减使用归纳法。

归纳前提:

$$j=n$$

$$2^{n-j}=2^0=1=r_i(n)$$

归纳步骤: 假定 $r_i(j+1)=2^{n-(j+1)}$ ，

则 $r_i(j)=r_i(j+1)+r_2(j+1)=2^{n-(j+1)}+2^{n-(j+1)}=2^{n-(j+1)+1}=2^{n-j}$。

因此，仅 $t_1[1]$ 就需要被访问 2^{n-1} 次!所以，由上至下的直接递归不是一个好办法。这很像斐波那契数的由上至下的计算!

但我们有更好的办法。仔细观察我们发现：$t_i[j]$ 只依赖于 $t_1[j-1]$ 和 $t_2[j-1]$ $(j\geqslant 2)$。因此，我们可以由底至上进行计算，从而避免重复访问。

```
FASTEST-WAY (c, x, n)        //计算最快路线，本程序为伪代码，不能直接执行
```

```
t1[1] =c1,0+ c1,1;
t2[1] =c2,0 + c2,1;
for(intj = 2;j<=n;j++){
        if(t1[j-1]+c1, j<=t2[j-1]+ x2, j-1 + c1, j){
            t1[j]=t1[j-1] + c1, j;
            l1[j]=1;
}
  else{
       t1[j]=t2[j-1] + x2, j-1 + c1, j;
       l1[j]=2;
}
if(t2[j-1]+c2, j<=t1[j-1]+x1, j-1+c2, j){
      t2[j]=t2[j-1]+c2, j;
      l2[j]=2;
}
else{
     t2[j]=t1[j-1]+x1, j-1+c2, j;
     l2[j]=1;
}
if(t1[n]+x1<=t2[n]+x2){
      t*=t1[n]+x1;
      l*=1;
}
else{
      t*=t2[n]+x2;
l* = 2;
}
PRINT-STATIONS(1, n)                                    //输出计算结果
i =l*;
cout<<"流水线"<<i<<", 梯级"<<n;
for(j = n; j>=2; j--) {
       i = l1[j];
       cout<<"流水线"<<i<<", 梯级"<<j-1;
}
```

上述由底至上算法的效率是线性的！

4.3 最长公共子序列

人类最喜欢将各种东西进行分类，尤其是对其他的人！一个人要么是我们的同类或朋友，要么是我们的潜在对手或敌人。而人类判别朋友和敌人一个重要标准是看他人与自己有多少共同点。如果共同点多，我们将其判为自己人，或者朋友；如果共少异多，则视为潜在

的敌人。

而这种社会学上的行为的一个变种是寻找两个人之间的最大或最多共同点！寻找最大或最多共同点的问题映射到算法里就是著名的最长公共子序列（Longest Common Subsequence，LCS）问题。

我们对这个问题感兴趣不仅是因为其与人类社会生活的联系，更为重要的是，这个问题能够精彩淋漓地演示动态规划策略。在你练习如何找出你和你喜欢的人之间的最大最多共同点的时候，先来练习如何找出两个序列的最长公共子序列吧。

最长公共子序列问题的定义如下：设有两个序列 $S_1[1..m]$ 和 $S_2[1..n]$，需要寻找它们之间的一个最长公共子序列。

例如：假定我们有如下两个序列

S_1: I N T H E B E G I N N I N G

S_2: A L L T H I N G S A R E L O S T

则 S_1 和 S_2 的一个最长公共子序列为：THING。又如：

S_1: A B C B D A B

S_2: B D C A B A

则 S_1 和 S_2 的一个最长公共子序列为：BCBA。

这里需要注意的是，一个子序列不一定必须是连续的，即中间可以被其他字符分开，但它们的顺序必须是正确的。另外，最长公共子序列不一定只有一个，而我们要寻找的是其中一个！

当然，如果要求子序列里面的元素必须连成一片也是可以的。实际上，连成一片的版本比本章讨论的版本更为容易。读者可在阅读完本节后自行解答。

4.3.1 第一种解法：蛮力策略

很显然，最简单的办法是使用蛮力。对每一个可能的字串进行检查，看看其是否是一个共享子序列，然后在所有公共子序列里面寻找最长的即可。就像愚公移山那样。检查步骤如下：

1）检查 $S_1[1..m]$ 里的每一个子序列。

2）看看其是否也是 $S_2[1..n]$里的子序列。

3）在每一步记录当前找到的子序列里面的最长的子序列。

这么一种蛮力策略的效率显然十分低下：对每一个子序列的检查需时 $O(n)$，而总共有 2^m 子序列需要检查，因此，本算法的最坏时间复杂性是 $O(n2^m)$。而这是指数级！

看来，愚公移山的精神虽然可嘉，但其实际效果却不甚令人满意。那如何才能改进呢？

回想前面一节讨论过的流水线调度问题：我们将寻找最佳流水线调度变为寻找最佳调度的时间计算，然后从时间上反向推出最佳调度。而这种变化使得问题得到极大的简化！

那么最长公共子序列问题能否也进行这种转化呢？当然可以！我们先来找出最长公共子序列的长度，然后将寻找长度的算法扩展来寻找子序列本身。这样就获得第二种解法。

4.3.2　第二种解法：动态规划

利用动态规划寻找最长公共子序列的步骤如下：

1）先寻找最长公共子序列的长度。

2）扩展寻找长度的算法来获得最长子序列。

策略：考虑序列 S_1 和 S_2 的前缀序列。

设 $c[i,j]=|\mathrm{LCS}(S_1[1\ldots i], S_2[1\ldots j])|$ ，则有 $c[m,n]=|\mathrm{LCS}(S_1, S_2)|$。

仔细观察我们发现：

$$c[i,j]=\begin{cases} c[i-1,j-1]+1 & S_1[i]=S_2[j] \\ \max\{c[i-1,j], c[i,j-1]\} & S_1[i]\neq S_2[j] \end{cases}$$

下面我们予以证明。按照 $S_1[i]$与 $S_2[j]$的关系，这个证明可以分为两种情况：

情况 1　$S_1[i]=S_2[j]$，如图 4-5 所示。

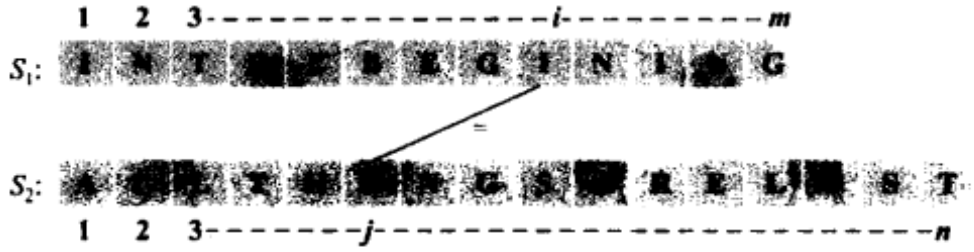


图 4-5　$S_1[i]=S_2[j]$的情况

设 $CS[1\ldots k]$=LCS($S_1[1\ldots i]$, $S_2[1\ldots j]$)，这里 $c[i,j]=k$，则 $CS[k]=S_1[i]=S_2[j]$；否则，我们将 CS 进行扩展。因此，$CS[1\ldots k\text{–}1]$ 是 $S_1[1\ldots i\text{–}1]$ 和 $S_2[1\ldots j\text{–}1]$ 的最长公共子序列，并且有：

$$CS[1\ldots k\text{–}1]=\mathrm{LCS}(S_1[1\ldots i\text{–}1], S_2[1\ldots j\text{–}1])$$

这点可以用反证法证明。如果 $S_1[1\ldots i\text{–}1]$ 和 $S_2[1\ldots j\text{–}1]$之间存在一个更长的公共子序列 w，即$|w|>k\text{–}1$，那么可以将 $CS[k]$ 直接连接在 w 之后而形成 $S_1[1\ldots i]$, $S_2[1\ldots j]$之间一个更长的公共子序列 $w \,\|\, CS[k]$ ，这里$|w \,\|\, CS[k]|>k$。从而与我们的假设 $CS[1\ldots k]$=LCS($S_1[1\ldots i]$, $S_2[1\ldots j]$)矛盾。因此，$c[i\text{–}1,j\text{–}1]=k\text{–}1$，也就是 $c[i,j]=c[i\text{–}1,j\text{–}1]+1$！

情况 2　$S_1[i]\neq S_2[j]$

我们还是设 $CS[1\ldots k]$= LCS($S_1[1\ldots i]$, $S_2[1\ldots j]$)，这里 $c[i,j]=k$。这个时候有三种情况需要考虑：

- $CS[k]=S_1[i]$，则 $CS[k]\neq S_2[j]$，因此 $CS[1\ldots k]$ 是 $S_1[1\ldots i]$, $S_2[1\ldots j\text{–}1]$的公共子序列。
- $CS[k]=S_2[j]$，则 $CS[k]\neq S_1[i]$，因此 $CS[1\ldots k]$ 是 $S_1[1\ldots i\text{–}1]$, $S_2[1\ldots j]$的公共子序列。
- $CS[k]\neq S_1[i]$ 并且 $CS[k]\neq S_2[j]$，则 $CS[1\ldots k]$ 一定是 $S_1[1\ldots i\text{–}1]$, $S_2[1\ldots j\text{–}1]$ 的公共子序列，并且该子序列就是 $S_1[1\ldots i\text{–}1]$, $S_2[1\ldots j]$ 或者 $S_1[1\ldots i]$, $S_2[1\ldots j\text{–}1]$的公共子序列。

综合上面三种情况，有：$c[i,j]=\max\{c[i\text{–}1,j], c[i,j\text{–}1]\}$。

如此我们获得了该问题的第一个特性：最优子结构特性，即一个最优解决方案包含了子

问题的最优解决方案，也就是说，如果 CS= LCS (S_1, S_2)，则 CS 的任何前缀子串必是 S_1 和 S_2 的某前缀的最长公共子序列。

如果设 LCS (S_1, S_2, i, j)表示 $S_1[1..i]$和 $S_2[1..j]$的最长公共子序列的长度，则有：

```
if(S1[i]==S2[j])
c[i,j]=LCS(S1,S2,i-1,j-1)+1;
else
c[i,j]=max{LCS(S1,S2,i-1,j),LCS(S1,S2,i,j-1)};
```

时间效率分析

最坏的情况在 $S_1[i] \neq S_2[j]$时出现，这个时候我们的一个问题被分解为两个子问题，且每个子问题只有一个参数减少 1 个单位。一共需要解决多少个子问题呢？

本书前面章节说过，如果一眼看不出递归表达式的结构，可以画一棵递归树来看看，如图 4-6 所示。

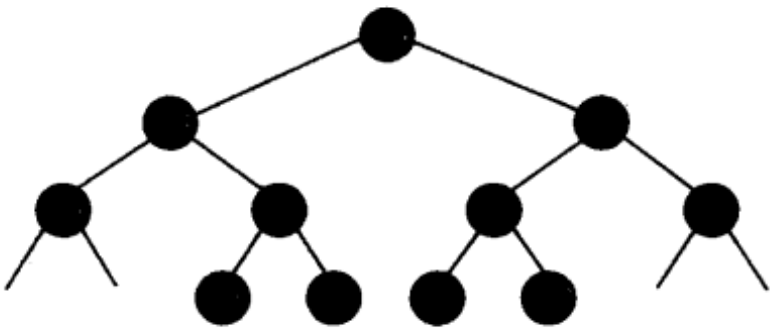


图 4-6 m= 3, n= 4 时的递归树

我们很快看出来，这棵树的高度是 $m+n$，即我们要解决的子问题数为指数级！

但是仔细观看发现，这里面的很多子问题是重复的！这点与计算斐波那契数列的数项的情况类似！而解决这种问题的办法是防止重复计算！即动态规划！

前面一节已经说过，重复子问题是使用动态规划时考虑的第二个特性：一个递归解决方案包含的不同子问题个数相对较少，绝大部分是重复的子问题！对于我们的 LCS 问题，如果两个子串的长度分别是 m 和 n，则它们潜在的不同的最长子序列个数是 mn。而 mn 远比 2^{m+n} 少（在 m 和 n 趋向无穷的时候）。

计算最长公共子序列长度的动态规划算法如下：

```
1. 初始化 c[i,j]，0≤i≤m，0≤j≤n
2. for(int i=1; i<=m; i++)
3. for (intj=1; j<=n; j++)
4. if(S1[i] ==S2[j])
5.          c[i, j] =LCS(S1, S2,i-1,j-1)+1;
6. else if(c[i-1, j]<=c[i,j-1])
7.       c[i,j]=c[i,j-1];
8. else
9.       c[i,j]=c[i-1,j];
```

该算法的正确性由前面分析直接得出，而时间复杂性是 $\Theta(mn)$，空间复杂性是 $\Theta(mn)$。

细心的读者可能已经发现上述算法有一个问题，就是只计算了最长公共子序列的长度，并没有给出最长公共子序列本身！那如何才能获得最长公共子序列本身呢？

这个实际上很简单，只需要在计算 $c[i, j]$时，记录下 $c[i, j]$每次递增时所对应的字符即可。具体细节则请读者自己完成。

4.4　最长公共子序列变种

这个最长公共子序列问题还有很多变种：最长递增子序列、最长递减子序列、编辑距离等。而编辑距离还直接与 DNA 分类鉴定有关。人体里面的 DNA 就是一个很长的字符串，而这个字符串里面重复出现的不同片段就是不同的基因。DNA 研究的一个重要问题是找出未知基因所扮演的角色或功能，而这项研究所采用的最重要的方法就是将未知基因和已知基因进行比较来推断其可能的功能。这种比较就基于编辑距离，即两个基因（已知和未知）的字母排列有多少差别。这个问题是最长公共子序列问题的一个变种，因此可以用动态规划来有效地解决。

4.5　记忆递归法

本章前面说过，动态规划的核心问题是找出子问题及其表现形式。而大多数时候，子问题的表现形式是递归的。因此，我们可以直接用递归来解决这类问题。但是因为大量重叠的子问题，使得递归方式的效率极为低下。所以我们选择使用动态规划方法来解决此类问题。

难道递归方法就派不上用场了吗？

既然递归方式的效率低下是因为需要重复解决相同的子问题，所以我们可以对递归算法进行修改，将已经解出来的子问题存放在表格里。这样每次要解决一个问题时，先查表看看该问题是否已经解决，如果是，直接将表格里的结果抄下来；如果不是，则继续解决该问题。这样，由于消除了重复计算，所以这种修改后的递归方式的效率与动态规划的效率一样，只不过其常数系数因递归的成本而比动态规划的要高。

由此可见，记忆递归法的法则就是“存储，不要重复计算”。例如，对于最长公用子序列问题，如果使用记忆递规法，将获得如下算法：

```
1. LCS (x,y,i,j)
2. if (c[i,j]== NIL)
3.     if(x[i]==y[j])
4.                 c[i,j]=LCS(x,y,i-1,j-1)+1;
5.     else
6.                 c[i,j]=max{LCS(x,y,i-1,j), LCS(x,y,i,j-1)};
```

该算法只比直接递归算法增加了第 2 行 **if** $c[i, j]$ = =NIL。该行条件判断对 $c[i, j]$的值进行检查，如果前面已经计算过，那么就不用重复计算了。这样每个 $c[i, j]$只被计算一次，所以

该算法的时间成本只能是 $\Theta(mn)$，而空间复杂性自然也是 $\Theta(mn)$，如图 4-7 所示。

		I	N	T	H	I	S	A	R	E	L	O	S	T
	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	0	0	0	0	0	0	0	0	0	0	0
L	0	0	0	0	0	0	0	0	0	0	0	0	0	0
L	0	0	0	0	0	0	0	0	0	0	0	0	0	0
T	0	0	0	1	1	1	1	1	1	1	1	1	1	1
H	0	0	0	1	2	2	2	2	2	2	2	2	2	2
I	0	1	1	1	2	2	2	2	2	3	3	3	3	3
N	0	1	2	2	2	2	2	2	2	3	4	4	4	4
G	0	1	2	2	2	2	2	2	3	3	4	4	4	5
S	0	1	2	2	2	2	2	2	3	3	4	4	4	5
A	0	1	2	2	2	2	2	2	3	3	4	4	4	5
R	0	1	2	2	2	2	2	2	3	3	4	4	4	5
E	0	1	2	2	2	2	2	2	3	3	4	4	4	5
L	0	1	2	2	2	2	2	2	3	3	4	4	4	5
O	0	1	2	2	2	2	2	2	3	3	4	4	4	5
S	0	1	2	2	2	2	2	2	3	3	4	4	4	5
T	0	1	2	2	2	2	2	2	3	3	4	4	4	5

图 4-7 $c[i,j]$的计算过程

不过有一点需要请读者注意，修改后的记忆递归不一定能达到理想效果，因为动态规划在程序开始时就确定子问题的范围，从而建立适当大小的表格；但有的递归算法要递归到一定程度才知道子问题的范围，从而难以预先建立表格。

4.6 空间效率改善

我们花了很多时间来讨论动态规划的时间效率，但对其空间效率却没有太多考虑。直观来看，动态规划需要计算每个子问题的解答，并将其存储起来供后面使用。因此，动态规划的空间效率似乎就是其所包括的子问题的个数，即 $\Theta(mn)$。

我们真的需要同时保存所有这些子问题的解吗？不一定。很多时候，我们可以将动态规划的空间效率提高到与子问题呈线性关系的程度。这里的关键是一个子问题只被仅仅少数几个更大的问题所需要，因此，一旦这些依赖于其的较大的问题被解决，那么这个子问题就无需被继续保持，其所占空间就可以释放出来。这样，空间效率可以降为 $\Theta(\min(m, n))$。

4.7 最优二叉搜索树

另外一个非常经典的可由动态规划解决的问题是所谓的最优二叉搜索树（Optimal Binary Search Tree，OBST）问题，或最优 BST 问题。顾名思义，最优二叉搜索树就是一棵

具有某种最优性质的二叉搜索树。一棵二叉搜索树是一棵满足如下条件的树：

1）每个结点包含一个键值。

2）每个结点最多有两个孩子。

3）对于任意两个结点 x 和 y，它们满足下述搜索性质：

- 如果 y 在 x 的左子树里，则 key[y]≤key[x]。
- 如果 y 在 x 的右子树里，则 key[y]≥key[x]。

而最优二叉搜索树是整个搜索成本最低的二叉搜索树。具体来说就是，给定键值序列 $K=\langle k_1, k_2, \cdots, k_n\rangle$，$k_1< k_2<\cdots<k_n$，其中键值 k_i 被搜索的概率为 p_i，要求以这些键值构建一棵二叉搜索树 T，使得搜索的期望成本最低（搜索成本为检查的结点数）。

对于键值 k_i，如果其在构造的二叉搜索树里的深度（离开树根的分支数）为 $\text{depth}_T(k_i)$，则搜索该键值的成本为 $\text{depth}_T(k_i)+1$（需要加上深度为 0 的树根结点）。由于每个键值被搜索的概率已知，我们有：

$$E[T\text{ 的搜索成本}]=\sum_{i=1}^{n}(\text{depth}_T(k_i)+1)p_i$$

$$=\sum_{i=1}^{n}\text{depth}_T(k_i)p_i+\sum_{i=1}^{n}p_i$$

如果每次搜索都是针对二叉搜索树里面存在的某个结点，则所有结点被搜索的概率之和将等于 1。因此，我们可以简化上述等式为：

$$E[T\text{ 的搜索成本}]=\sum_{i=1}^{n}\text{depth}_T(k_i)p_i+1$$

例如，假定我们有 6 个键值：k_1、k_2、k_3、k_4、k_5、k_6，它们被搜索的概率如表 4-3 所示。

表 4-3　每个结点被搜索的概率

结　点	k_1	k_2	k_3	k_4	k_5	k_6
被搜索概率	0.24	0.18	0.09	0.13	0.3	0.06

如果我们构造的二叉搜索树如图 4-8 所示，则每个键值的搜索成本期望值如表 4-4 所示。

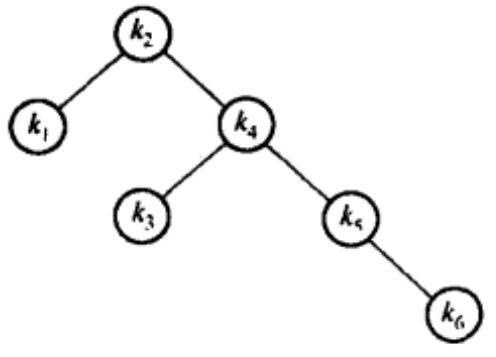


图 4-8　一种可能的二叉搜索树

表 4-4　图 4-8 二叉搜索树的搜索成本

结点 k_i	深度 $\text{depth}_T(k_i)$	深度×概率 $\text{depth}_T(k_i)p_i$
k_1	1	0.24
k_2	0	0.00
k_3	2	0.18
k_4	1	0.13
k_5	2	0.60
k_6	3	0.18
和值		1.33

因此，搜索的成本期望值 E[搜索成本] = 1.33+1=2.33。

但如果构造的二叉搜索树如图 4-9 所示，则每个键值的搜索成本期望值将如表 4-5 所示。

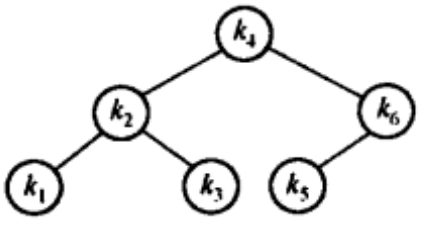


图 4-9 一棵更加平衡的二叉搜索树

表 4-5 图 4-9 二叉搜索树的搜索成本

结点 k_i	深度 $\text{depth}_T(k_i)$	深度×概率 $\text{depth}_T(k_i)\,p_i$
k_1	2	0.48
k_2	1	0.18
k_3	2	0.18
k_4	0	0.00
k_5	2	0.60
k_6	1	0.06
和值		1.50

因此，搜索的成本期望值 E[搜索成本]=1.50+1=2.50。因此，这棵搜索树比前面的一棵要更次！但图 4-8 的二叉搜索树是最优的吗？一眼似乎不容易看出。

那我们能看出什么呢？能看出的只有一点：更加平衡（高度小）的二叉搜索树不一定比更不平衡（高度大）的二叉搜索树更优。那这一点对我们构建最优 BST 有什么帮助吗？有。就是按照平衡二叉树的方式肯定不行。

也许读者会想，如果把搜索概率高的键值安排在根结点或离根结点更近的结点上就能获得最优的二叉搜索树。情况是这样的吗？我们只需要构建一棵按照概率高低从上至下排列的二叉搜索树就知道了。图 4-10 为如此构建的二叉搜索树，其搜索成本如表 4-6 所示。

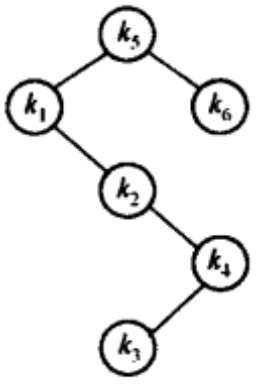


图 4-10 一棵按搜索概率由高至低排列的二叉搜索树

表 4-6 图 4-10 二叉搜索树的搜索成本

结点 k_i	深度 $\text{depth}_T(k_i)$	深度×概率 $\text{depth}_T(k_i)\,p_i$
k_1	1	0.24
k_2	2	0.36
k_3	4	0.36
k_4	3	0.39
k_5	0	0.00
k_6	1	0.06
和值		1.41

结果令人失望：这棵按照概率高低排列的二叉搜索树的搜索成本高于图 4-8 的二叉搜索

树的搜索成本。因此，按照概率高低来构建二叉搜索树的想法也不可行。

剩下的办法似乎只有蛮力的穷举法了：将每种二叉搜索树都构造一遍，然后计算它们的搜索成本期望值，取其中最小的。但问题是由 n 个键值构造的可能的二叉搜索树实在太多，至少有 $\Omega(4^n/n^{3/2})$棵不同的二叉搜索树，（读者看出来了吗？）一个指数级的数量！

显然，天真的穷举办法是行不通的，那么我们如何来构建最优 BST 呢？

既然穷举行不通，我们就要进行分析。而分析什么呢？当然是分析最优 BST。我们希望通过分析发现最优 BST 有什么规律可循！而分析一棵树还能有什么别的东西可检查呢？当然是树的子树啦。那么最优 BST 的子树有没有什么特点呢？

如果仔细查看一棵 BST 的任意一棵子树，我们发现：它包括一个连续的键值序列：$k_i,\cdots,k_j$（因为 $k_1< k_2<\cdots<k_n$），这里 $1\leqslant i\leqslant j\leqslant n$。如果该 BST 是最优的，则该子树是以键值 $k_i,\cdots, k_j$ 为基础构建的一棵最优 BST。这一点很容易看出来，因为如果该子树不是最优子 BST，则我们可以用一棵包括 k_i，…，k_j 的最优子 BST 替换该字树，从而形成一棵更优的 BST。

而有了最优子结构，动态规划也许能派上用场。对于键值 $k_i,\cdots, k_j$，如果构建最优 BST，则它们之间必有某个键值成为树根。不失一般性，设树根为 k_r，$i\leqslant r\leqslant j$，则左子树包括的键值为 $k_i,\cdots, k_{r-1}$，而右子树包括的键值则为 $k_{r+1},\cdots, k_j$，如图 4-11 所示。

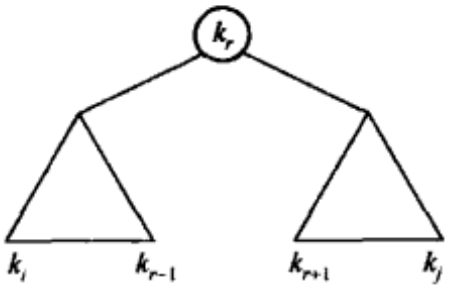


图 4-11　二叉搜索树的最优子结构

如果我们对所有潜在的树根键值进行检查，并找出包括 $k_i,\cdots$，k_{r-1} 的最优子 BST 和包括 $k_{r+1},\cdots,k_j$ 的最优子 BST，则我们可以保证找到一棵包括键值 $k_i,\cdots, k_j$ 的最优 BST。

4.7.1　递归解法

子问题空间为：找出由键值 $k_i,\cdots, k_j$ 构成的最优 BST，这里 $i\geqslant 1, j\leqslant n, j\geqslant i-1$。

定义　$e[i, j]$ =包括键值 $k_i,\cdots, k_j$ 的最优 BST 的搜索成本期望值，则有：

1）如果 $j=i-1$，则 $e[i, j]=0$。

2）如果 $j\geqslant i$，则：

a）选择某个键值 k_r，作为树根，$i\leqslant r\leqslant j$。

b）构建一棵包括键值 $k_i,\cdots, k_{r-1}$ 的最优 BST 作为其左子树。

c）构建一棵包括键值 $k_{r+1},\cdots, k_j$ 的最优 BST 作为其右子树。

3）当一棵子树被挂在一个键值下的时候：

a）该子树里所有键值的深度增加 1。

b）该子树的搜索成本增加 $w(i, j)=\sum_{l=i}^{j} pl$。

如果 k_r 是包括键值 $k_i,\cdots, k_j$ 的一棵最优 BST 的根，则有：

$$e[i, j] = \mathrm{Pr}+(e[i, r-1] + w(i, r-1))+ (e[r+1, j] + w(r+1, j))。$$

由于 $w(i, j)= w(i,r-1)+ \mathrm{Pr}+ w(r+1, j)$，则有：

$$e[i, j] = e[i, r-1] + e[r+1, j] + w(i, j)。$$

到这里问题似乎解决了。但上述等式还存在一个问题，就是它假定我们知道 k_r 的取值。但事实是我们并不知道！而解决的办法自然是测试每个键值，然后选择最优的。因此有：

$$e[i, j]=\begin{cases}0 & j=i-1\\ \min_{i\leqslant r\leqslant j}\{e[i,r-1]+e[r+1,j]+w(i,j)\} & i\leqslant j\end{cases} \qquad (4\text{-}1)$$

如果再仔细观察，我们发现该递归表达式包括了许多重复子问题。因此，动态规划最合适不过了。

4.7.2 计算最优答案

与前面一样，为避免重复计算，我们将已计算出来的结果存放在表格里：$e[1..n+1, 0..n]$。表格项 $e[i, j]$的值为键值为 $k_i,\cdots, k_j$ 组成的最优二叉搜索树的搜索成本期望值。根据前面的分析，这个表格里只有 $j\geqslant i-1$ 的 $e[i, j]$项才会有值，其他项都没有意义。

而除了计算 $e[i, j]$的值外，我们还需要计算由键值 $k_i, \ldots, k_j$ 构成的最优子 BST 树的树根。因为我们不仅需要计算出最优 BST 树的搜索成本，更需要计算出这棵 BST 本身。为此，我们在算法中构造一张额外的表 *root*，其中元素 $root[i, j]$存放的就是由键值 $k_i,\cdots, k_j$ 构成的最优子 BST 树的树根。

式（4-1）里还有一项 $w(i, j)$需要计算。同样，为避免重复，我们也用一张表格 w（$w[1..n+1, 0..n]$）将已经计算出来的 $w(i, j)$值存放起来，后面需要的时候直接使用即可。表 w 的计算方式如下：

$$w[i, i-1] = 0 \qquad 1\leqslant i\leqslant n$$

$$w[i, j] = w[i, j-1] + p_j \qquad 1\leqslant i\leqslant j\leqslant n$$

下面为我们的动态规划 BST 算法：

```
OPTIMAL-BST (p, q, n)

1. for (inti =1; i<=n+1; i++) {
2.        w[i,i-1]=0;
3.        e[i,i-1]=0;
4. }
5. for (int l=1; l<=n; l++)
6.       for (int i=1; i<=n-l+1; i++) {
7.            j=i+l-1;
```

```
8.              e[i,j]=∞;
9.              w[i,j]=w[i,j-1] + pj;
10.             for (int r=i, r<=j;r++) {
11.                   t=e[i, r-1]+ e[r+1, j]+ w[i,j];
12.                   if (t < e[i,j]) {
13.                               e[i,j]=t;
14.                               root[i,j]=r;
15.                                     }
16.             }
17. }
18. return e and root;
```

算法第 1、2、3、4 三行初始化 *w*、*e* 两张表。第 5 行为算法的主循环，每次循环考虑的问题是如何由 *l* 个键值构造出最优 BST，即根据尺寸从小到大的顺序依次计算最优的子 BST。当然，*l* 循环到 *n* 的时候，本算法就计算出了整个最优 BST 的构造了。对于表 4-3 给出的例子，算法 OPTIMAL-BST 计算出的 *w*、*e*、root 三张表如图 4-12 至图 4-14 所示。

		j						
	w	**0**	**1**	**2**	**3**	**4**	**5**	**6**
i	**1**	0.00	0.24	0.42	0.51	0.64	0.94	1.00
	2		0.00	0.18	0.27	0.40	0.70	0.76
	3			0.00	0.09	0.22	0.52	0.58
	4				0.00	0.13	0.43	0.49
	5					0.00	0.30	0.36
	6						0.00	0.06
	7							0.00

图 4-12　*w* 表

		j						
	e	**0**	**1**	**2**	**3**	**4**	**5**	**6**
i	**1**	0.00	0.24	0.60	0.84	1.19	2.01	2.19
	2		0.00	0.18	0.36	0.71	1.36	1.53
	3			0.00	0.09	0.31	0.83	0.95
	4				0.00	0.13	0.56	0.68
	5					0.00	0.30	0.42
	6						0.00	0.06
	7							0.00

图 4-13　*e* 表

		j					
	root	**1**	**2**	**3**	**4**	**5**	**6**
i	**1**	1	1	2	2	2	2
	2		2	2	2	4	5
	3			3	4	5	5
	4				4	5	5
	5					5	5
	6						6

图 4-14　root 表

从算法 OPTIMAL-BST 的伪代码实现很容易看出，其时间复杂性最坏为 $O(n^3)$，因为算法里有一个 3 层循环嵌套（第 5、6、10 行）。而每层嵌套循环最多 *n* 次。事实上，还可以证明，该算法最好也不会超过立方级，即其时间复杂性为 $\Omega(n^3)$。因此，该算法的时间复杂性为 $\Theta(n^3)$。

有了上述三张表的值，就可以直接构建最优 BST 的算法：

```
CONSTRUCT-OPTIMAL-BST (root)
    r =root[1,n];
    cout<< "k"<<r<<"为树根";
    CONSTRUCT-OPT-SUBTREE (1, r-1, r, "left", root);
    CONSTRUCT-OPT-SUBTREE (r+1, n, r, "right", root);
CONSTRUCT-OPT-SUBTREE (i, j, r, dir, root)
    if(i <=j) {
            t=root[i,j];
            cout<<"k"<<t<<"是 k"<<r<<"的"<<dir<<"孩子";
            CONSTRUCT-OPT-SUBTREE (i, t-1, t, "left", root);
            CONSTRUCT-OPT-SUBTREE (t+1, j, t, "right", root);
}
```

对于表 4-3 给出的例子，算法 CONSTRUCT-OPTIMAL-BST 构造出的最优二叉搜索树的总成本为 2.19，而该最优 BST 树的结构则如图 4-15 所示。

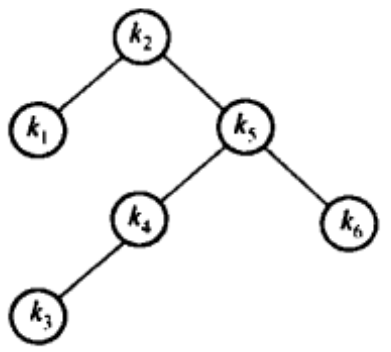


图 4-15 根据表 4-3 数据构造的最优二叉搜索树，总搜索成本为 2.19

随机 BST 构造 虽然使用动态规划可以构造出最优二叉搜索树，但 $\Omega(n^3)$的时间复杂性仍然显得太高。但如果所有键值被访问的概率相等，则该时间复杂性将能有效降低。降低的办法是随机构造一个二叉搜索树！结果就将非常接近最优二叉搜索树。而随机构造 BST 的成本只有 $O(n\log n)$（具体算法留给读者思考吧）。也许这就是人算不如天算吧！

4.8 最优子结构与重叠子问题

动态规划策略是一种分治策略，即将问题分解为一个或多个子问题。动态规划策略的特点就是根据问题的具体情况对我们的选择进行动态调整，并且在每一步的调整中做出的都是一个最优选择（因为做出选择所需要的所有信息都已经知道）。而且该最优选择与对子问题的最优解组合获得原问题的最优解。因此，动态规划要能够成功，需要注意原问题的两点：

1）最优子结构。

2）重叠子问题。

因此，在考虑使用动态规划策略时，我们的一般战略如下：

1）证明问题的解决方案中包括一个选择，选择之后将留下一个或多个子问题。

2）设计子问题的递归描述方式。

3）证明对原问题的最优解里包括对所有子问题的最优解。

4）证明子问题之间重叠。

对于第 3）点的证明通常使用反证法，即假定最优解不包括对子问题的最优解，则将对子问题的最优解揉进到对原问题的解里将获得一个更优的解，从而导致矛盾。

第 4）点并不是动态规划算法正确性所必需的，而是此种算法效率保证所必需的。如果子问题之间不发生重叠，则动态规划算法的优越性将荡然无存，它将与一般的递归算法并无二致。

另外，在讨论子问题的时候需要注意考虑到所有的子问题，即要考虑所有选择所产生的后果，然后从中选择一个最优的选项。而在表述子问题样本空间的时候力求简单，在需要的时候再进行扩展。例如：对于流水线问题：我们的问题样本空间就是从流水线起点到梯级 $S_{1,j}$ 和 $S_{2,j}$ 的最快路径。而比这更复杂的空间描述没有任何必要，比如从梯级 $S_{1,i}$ 和 $S_{2,i}$ 到梯级 $S_{1,j}$ 和 $S_{2,j}$ 的最快路径就无需考虑（这个样本空间显然更为复杂，解决的难度将更大）。

4.8.1　最优子结构

不同问题的最优子结构也不尽相同。在考虑动态规划时需要考虑的因素包括：

1）最优解里需要解决的子问题数量有多少？

2）在判断使用哪些子问题时，需要进行多少选择？

对于流水线问题，我们的子问题为 1 个，即从进入流水线到梯级 $S_{i,j}$ 的问题分解为从进入流水线到梯级 $S_{i-1,j-1}$ 的问题。而选择为 2 个，即前面的梯级是使用 $S_{1,j-1}$ 还是 $S_{2,j-1}$。（注意这两者之间只有一个会被选择，因此子问题数为 1 个）。对于最长公共子序列问题，子问题数量也是 1 个，即 X_{i-1} 和 Y_{j-1} 的 LCS，X_{i-1} 和 Y 的 LCS，X 和 Y_{j-1} 的 LCS 三个子问题的 1 个。而选择则根据 $x_i = y_j$ 成立否而分别为 1 个或 2 个选项。

- 如果 $x_i = y_j$，则需计算 X_{i-1} 和 Y_{j-1} 的 LCS，此时只有 1 个选项。
- 如果 $x_i \neq y_j$，则需计算 X_{i-1} 和 Y 的 LCS，X 和 Y_{j-1} 的 LCS，此时有 2 个选项。

而动态规划的时间成本则可以简单表示为：

全部子问题数量×选择数量

例如，对于流水线问题，共有 $\Theta(n)$个子问题，每个梯级有 2 个选择，因此成本为 $\Theta(n)$；对于最长公共子序列问题，共有 $\Theta(mn)$个子问题，小于等于 2 个选择，因此总成本为 $\Theta(mn)$；对于最优二叉搜索树问题来说，共有 $\Theta(n^2)$个子问题，每次有 $O(n)$个选择，因此总成本为 $O(n^3)$。

4.8.2　重叠子问题

重叠子问题指的是一个递归解决方案里包括的子问题数量虽然很多，但不同子问题的数量很少，也就是说，少量的子问题被重复解决了很多次。例如，对于最长公共子序列问题，如果两个字串的长度分别是 m 和 n，则不同的子问题数量只有 mn。这种重叠子问题情况的

出现是因为递归算法不断重复解决同一个问题所导致。一个好的分治策略应该每次分解的时候都产生全新的子问题。如果不是这样，则该分治策略的设计有缺陷。例如，归并排序所采用的分治就是每次产生的都是新问题（如图 4-16 所示）。

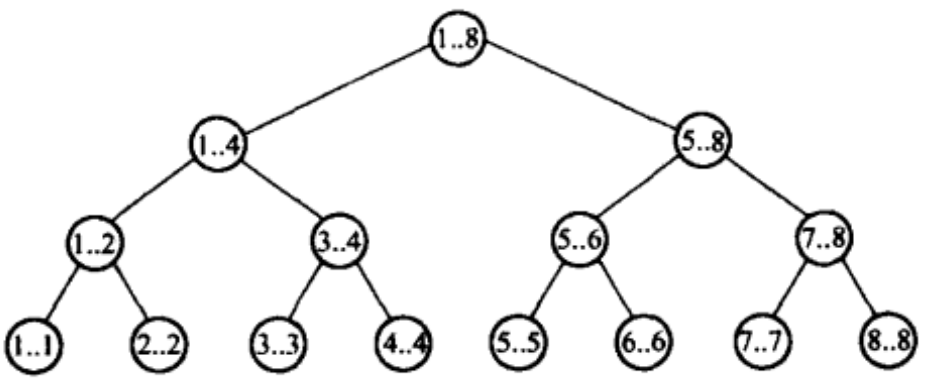


图 4-16 最优二叉搜索树的重叠子问题

当然，有了重叠子问题并不要紧，关键是必须认识到这点。这样我们可以加以利用，避免重复计算，从而提高算法的效率。

4.9 动态规划与静态规划的关系

读者也许会想到，既然有动态规划，那么也应该有静态规划吧！不错，算法世界里（或运筹学里）确实还存在静态规划的概念，只不过它们不是被直接称呼为静态规划，而是有着更加动听的名字：线性规划和非线性规划。

虽然本书不打算对线性规划和非线性规划进行阐述（这个任务留给运筹学完成），但因为它们是静态的，我们姑且将它和动态规划进行比较，以加深读者对动态规划的理解。

首先，动态规划与静态规划（线性规划和非线性规划等）研究的问题都是优化问题，其研究对象本质上都是在若干约束条件下的函数极值问题。但它们各有优缺点。

与静态规划相比，动态规划具有许多优越性：

- 动态规划的核心是找出一个问题所包含的子问题及其表现形式。虽然找出子问题的表现方式需要创造力和实验，但也存在一些常见的形式，如子问题是原问题的前缀，子问题是原问题的中缀，子问题是原问题的子树。因此，动态规划经常有迹可循。
- 动态规划比静态规划更容易获得最优解。静态规划可能由于约束条件确定的约束集合复杂而变得困难。而动态规划把原问题分解为一系列结构相似的子问题，每个子问题的变量个数大大减少，约束集合也简单得多，因此相对更容易求解。
- 动态规划可以得到一族最优解（原问题及子问题的最优解），而非线性规划只能得到全过程的一个最优解。
- 动态规划的时间效率很容易获得：子问题的数量×子问题的时间效率。

当然，与静态规划相比，动态规划也存在缺点：

- 找出子问题的表现方式需要创造力和实验，经常需要对每类问题进行具体分析，非熟练的分析人员难以准确对原问题进行合理分解，导致应用上的局限。

- 状态空间可能呈指数增长。如果一维状态变量有 m 个取值，则 n 维问题的状态就有 m^n 个值。对于 n 较大的实际问题，其计算成本上无法容忍。

4.10 动态规划与静态规划的相互转换

静态规划和动态规划看上去好像水火不相容的一对矛盾，但正如中国传统哲学里所说的，阴阳虽然对立，但却可以相互转换。同样，静态规划和动态规划之间也可以进行相互转换。

动态规划 ⇒ 静态规划

一般来说，动态规划问题可以转换为静态规划来解决。如果将动态规划看做求一系列决策 $d_1,d_2,\ldots,d_n$，使得指标函数 $F_{1n}(x,d_1,d_2,\ldots,d_n)$达到最优（最大或最小），这里 x 为输入；将动态规划的状态转移方程（每进行一个选择，系统就进入一个不同的状态）、端点条件（输入和输出）、允许状态集（什么状态是可以达到的）、允许决策集（什么决策是可以允许的）看做约束条件，则同样一个问题原则上可以用静态规划里的非线性规划方法求解。

静态规划 ⇒ 动态规划

对于静态规划来说，只要适当引入阶段变量、状态、决策等，就可以用动态规划方法求解。例如，可用动态规划方法来求解下列非线性规划问题：

$$\max\sum_{k=1}^{n} f_k(c_k)，\text{约束条件为}\sum_{k=1}^{n} c_k = a，\ c_k \geqslant 0$$

其中 $f_k(c_k)$为任意的已知函数。

我们可以按变量 c_k 的序号划分阶段，将其看做 n 个决策。设状态序列为 $s_1,s_2,\ldots,s_n$，取问题中的变量 $c_1,c_2,\ldots,c_n$ 为决策。状态转移方程为 $s_1=a$, $s_{k+1}=s_k-c_k$, $k=1,2,\ldots,n$（注意 $s_{n+1}=0$）。

取 $f_k(c_k)$为阶段指标，最优值函数的基本方程为：

$$g_k(s_k) = \max_{0<ck<sk}[f_k(s_k) + g_{k+1}(s_{k+1})] \qquad 0 \leqslant s_k \leqslant a, k = n, n-1, \cdots 2, 1$$

$$g_{n+1}(0)=0$$

按照逆序解法求出对应于 s_k 每个取值的最优决策 c_k^* (s_k)，计算至 $g_1(a)$后，即可利用状态转移方程得到最优状态序列$\{s_k^*\}$和最优决策序列$\{c_k^*(s_k^*)\}$。

在本书后面讨论经典算法设计时，还会常常用到动态规划。

思考题

1. 请对最长公共子序列问题的解答进行优化，将空间效率提高到 $O(\min\{m, n\})$。
2. 解决公共子序列问题的时候能否不使用动态规划，而时间复杂性仍然是 $\Theta(mn)$，空间复杂性仍然是 $\Theta(mn)$？请给出详细说明。
3. 本书在第3章介绍了矩阵乘法，讨论了如何以最有效的方式实现两个矩阵的相乘。而链式矩阵乘法则是一个不同的问题：给定 n 个矩阵 M_1、M_2、M_3、…、M_n，其维数分别是

$m_1 \times m_2$、$m_2 \times m_3$、$m_3 \times m_4$、…、$m_{n-1} \times m_n$，求 $M_1 \times M_2 \times M_3 \times \cdots \times M_n$。由于矩阵乘法遵守结合律，我们有很多种方式来进行链式矩阵乘法。你的任务是找出其中的最优者。请用动态规划思想找出链式矩阵乘法的最优结合方式？

4. 请将本章的流水线扩展到三条，在每个梯级将有三种选择：在本线上直接进入下一梯级，或转移到另外两条线的任意一条上后再进入下一梯级。它和两条线时有何不同？
5. 证明用穷举法构造本章所讨论的最优二叉搜索树的成本为 $\Omega(4^n/n^{3/2})$。
6. 在最优 BST 的讨论中，如果所有键值被访问的概率相等，那么你能对动态规划解法进行改良，以降低构造最优 BST 的构造成本吗？
7. 如果一个问题具有最优子结构属性，但不具有重叠子问题属性，那么是否仍然可用动态规划策略予以解决。说明你的理由。
8. 在进行 VLSI 芯片设计的时候经常需要将各种元件用导线连接起来。假定某种芯片里只有两种布线：水平布线和垂直布线，水平布线与垂直布线的连接通过焊点实现。如图 4-17 所示，所有的布线都与网格线重合，所有的针脚都在一条直线上。

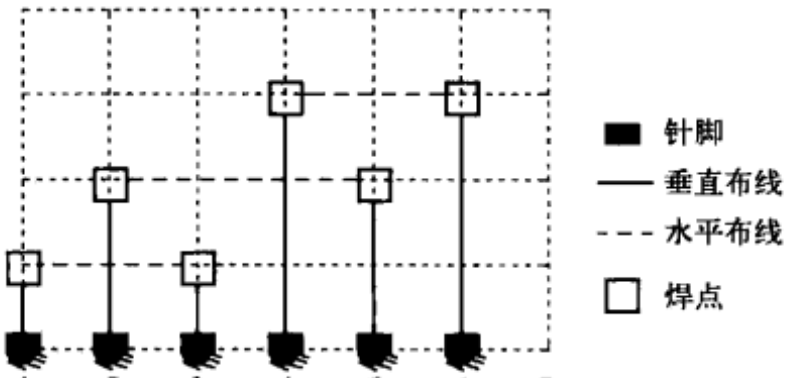


图 4-17 布线实例

我们的目标是使用最少的水平布线将一组针脚对连接起来。例如，图 4-17 使用了 3 条水平布线，但实际上并不需要这么多，2 条即可。

设 $L=\{(p_1,q_1),(p_2,q_2),\ldots,(p_n,q_n)\}$ 为一组针脚对，每个针脚只在列表里出现一次。我们的目标是要使用最少的水平布线将每一对针脚都连接起来。这个问题为针脚的最小连接问题。例如，图 4-17 对应的针脚最小连接问题是 $L=\{(1,3),(2,5),(4,6),(8,9)\}$。

请设计一个算法，解决 n 对针脚的最小连接问题。证明正确性并分析算法的时间复杂性。

9. 你走在街上的时候需要从左边马路走到右边马路上，而一路上的十字路口有 n 个。你可以在任意一个十字路口穿越马路，前提当然是在绿灯的时候横穿（不然很危险）。如果横穿信号为红灯，则需要等待。每个十字路口信号灯的红绿时间长度不同，设绿灯每亮 1 分钟，红灯亮的时间长度为 h_i 分钟，$i=1,2,\cdots,n$，这里 n 是十字路口数。而从路口 i 步行到下一个路口 $i+1$ 需要时间 b_i。我们的目标是尽早走到另外一边马路上。请设计算法告诉我们应该在哪个路口横穿，期望的等待时间为多少？并分析算法的复杂性。

CHAPTER 5

第 5 章　贪婪选择思想

天国就好像一个国王，要和他的仆人算账。在开始算的时候，有人带了一个欠 1 000 万银子的人来。因为他没有偿还的债务很多，主人吩咐把他和他妻子儿女，及一切所有的都卖了偿还。那仆人就俯伏在地说，主人啊，宽限我一些日子吧，将来我都要还清的。那仆人的主人就动了慈心，把他释放了，并且赦免了他的所有债务。那仆人出来，遇见他的一个同伴，欠他 10 两银子，便揪着他，掐住他的喉咙说，你把所欠的还我。他的同伴就俯伏央求他说，宽限我些日子吧，将来我必还清。他不肯，竟然把他下在监牢里，等他还了所欠的债。众同伴看见他所做的事，就甚忧愁，把这事都告诉了主人。于是主人叫他来，对他说，你这恶奴才，你央求我，我就把你所欠的债都免了。你不应当像我怜惜你那样怜惜你的同伴吗？主人大怒，把他交给掌刑的，等他还清了所欠的债。

——摘自《圣经 · 马太福音》

圣经故事里的恶奴因为贪婪而被主人下到了大牢。但对于很多人来说，贪婪仍然是敛财或者摄取权利地位的不二法门。不过，除了可以用来敛财之外，贪婪还可以用来设计更为有效的算法，只不过在算法设计里，贪婪必须适可而止，否则就会物极必反了。

5.1　仅有动态规划是不够的

动态规划是对分治思想的一种改善，它在发现分解出来的子问题有重叠时，使用由底至上的策略来避免重复计算，从而提升了算法的效率。但避免重复计算并不是最高明的策略。

动态规划的一个关键特点是在每次做选择前，将所有选择的效果进行了计算。在此计算的基础上选择能够达到最优的选项。因此，动态规划的每次选择都是最优的。问题是，这种策略在选项数量巨大的情况下将不堪重负。例如，在下棋的时候，如果使用动态规划策略，则需要先对每步可能走出的棋的影响进行计算，然后比较，选取最优的走法。但每一步可进行的走法实在是太多，如果再考虑到一盘棋有几乎不计其数的步骤，这个计算任务就是不可完成的。即使对超级计算机来说，也无法胜任。在这个时候，显然，我们需要新的算法策略。

新的策略是什么呢？有一点是肯定的：我们不能将所有可能选项的后果都计算一遍！事实上，我们可以在进行选择的时候并不进行任何计算，而是根据当时的情况做出我们认为最好的选择！这样我们就避免了大量计算，从而大大地提高算法的效率。

这种提高算法效率的策略就是本章要介绍的另一种算法思想：贪婪！

5.2 什么是贪婪

“良禽择木而栖，良将选主而事”，人类似乎永远在追求最好的东西。公司雇人要雇最能干的人，大学招生想招最好的学生，各种评优选优或者选“*N* 大杰出 XXX”等活动无一不散发着贪婪的气味。因为这种按照某种标准总是挑选最接近该标准的人或物的做法就是贪婪策略。或许我们不曾觉察，我们一生都在贪婪。

在算法研究中，人类也不可避免地展现了自己的本性。将人类贪婪的本性应用到算法设计中，就是本章所要讨论的贪婪选择思想或者贪婪算法。

什么是贪婪算法呢？下过围棋的人都知道，要想在高手较量中取胜，每一步棋都需要考虑随后的很多步。谁考虑的步数多，谁取胜的可能性就更大。如果一个棋手每步棋只看眼前的效果，则这个棋手恐怕得经常输棋。这种目光短浅的做法虽然在下围棋上不可取，但在有些时候却是获得最优（或较优）结果的简单易行的方法。例如，在找男女朋友时，我们总是试图在所认识的人里面寻找最优秀者，以此获得最优结果。

这种每步只看眼前效果最好的做法就是贪婪。以贪婪作为决策基础的算法就是贪婪算法。

说到这里，贪婪算法的基本策略已经清晰可见：一步一步地构建问题的最优解决方案，其中每一步均只需考虑眼前的最佳选择，即希望通过局部最优到达全局最优。

在寻求最优方面，贪婪算法与动态规划算法的目的一样，但是在具体行为上不同：在每次需要做出选择时，贪婪算法总是选择当前看上去最好的那个，希望这种贪婪的行为能够引导到全局最优。其实，贪婪算法并不是某种特定的具体算法，而是一类抽象的算法，或者说是一种思想。其具体表现为，对解的空间进行搜索时，不是搜遍所有的空间，而是在局部范围内进行择优选取，决定下一步搜索的方向，这样就能得到惊人的高效性。

贪婪算法的目的不是找到全部解，而是只找出一种可行解。在一定的情况（一定的条件，或一些情况）下，贪婪算法找出的将是最优解。

5.3 背包问题

在电影《木乃伊》里有一个镜头，一个进入到地宫藏宝室的人由于贪婪，将金银珠宝装满了自己衣服的口袋和随身带的背包。由于背包实在太沉重，逃出地宫的时候他行走不便且极为缓慢，导致被倒塌的地宫活埋！这就是贪婪的代价！

不过本书要讨论的贪婪却不会导致这样的结果。因为我们是搞计算机的人，所以我们不会傻到像电影里面的贪婪人那样，只顾贪婪，而不顾生路。所以我们讨论的贪婪是以保持生

路为前提。如果因贪婪而丧失生命，这种贪婪还有何意义呢？

因此，本章讨论的贪婪是一种有节制的贪婪，既能够带来收益，又不会付出代价（被倒塌的地宫活埋）。或者说，我们是要做一个聪明的贪婪者！

下面我们就以一个进入地宫发现大批珍宝的人为例加以说明。

背包问题　假定你因天缘进入一个藏宝地宫（见图 5-1），里面有奇珍异宝 n 件（n 当然大得你数不过来）。你当然想尽可能多地带走珍宝，无奈你只有一个背包，且该背包只可承重 Wkg。超出此重量，背包就会裂开。地宫里的每件珍宝都有重量 w_i 和价值 p_i。请问，如何挑选珍宝才能使得你背出的珍宝价值最大？当然是在背包不断裂的情况下。

图 5-1　地宫里遍布金银财宝、钻石玛瑙

解答此题的一个显而易见的方法是将所有珍宝按照某种方式进行排序，然后按该次序挑选珍宝。由于有总重量限制，按照珍宝的价值排序将是不明智的，因为价值最高的珍宝可能其重量很重，导致背出的件数很少，从而导致背出去珍宝的总价值并没有达到最高。

因此，一个好的排序应该同时考虑价值和重量，或者说以 p_i/w_i 的比率为标准从高到低进行排列，然后按照这个次序进行挑选。即优先挑选价值重量比最高或单位重量价值最高的珍宝，这种每次拣最好的东西拿的思维就是贪婪！而贪婪是非常直接的策略，地球人都能很快、很自然地想到！别的策略也许需要花时间来学习，但贪婪人人都会！

解决背包问题的贪婪策略如下：

首先，按照 p_i/w_i 比率对所有珍宝进行排序，形成：$p_i/w_i \geqslant p_{i+1}/w_{i+1}$。

```
KNAPSACK (p, w, W)
load =0;
i=1;
while (load <W&&i<=n) {
        if (wi<=W-load) {
            take item i;
            load=load+wi;
        }
        i=i+1;
}
```

显然，上述算法的时间复杂性最坏不超过 $O(n)$（这里排除了一开始对珍宝进行排序需要的时间，该问题将在第 10 章讨论），似乎是非常优秀的算法（线性）。但问题是这个算法不一定让你背出价值最大的珍宝！

假定我们有珍宝 1、2、3、4、5，其重量和价值如表 5-1 所示。我们的背包的总承重为 50kg。

表 5-1 宝库里面各种宝物的价值和重量

i	1	2	3	4	5
p_i	60	100	120	30	40
w_i	10	20	30	10	20
p_i/w_i	6	5	4	3	2

按照上述算法，我们装进背包的珍宝为 1、2 和 4，总价值 190 万元，总重量 40kg。虽然，背包还有 10kg 承重空间，但剩下的珍宝都不能装进去了。

而最优的解决方案是挑选 2 和 3，总价值 220 万元，总重量 50kg。所有承重都得到充分利用。

贪婪算法之所以没有达到找出最优答案是因为它没有搜遍解的空间，它只搜索了部分空间！但这种放弃最优解的好处是算法的高效！我们再强调一遍，贪婪算法的目的不是找到全部解，而是只找出一种可行解。但是，如果运气好，也就是说，在一定的情况（或一定的条件）下，贪婪算法找出的解将是最优解。

例如，如果对背包问题加以修改，使得取珍宝的时候不一定要取走完整的 1 件，而是可以取半件、1/3 件，甚至一件珍宝的任何分之一，那么贪婪算法将获得最优结果。还是以上面的例子为例，这个时候贪婪算法取的是整个珍宝 1、整个珍宝 2 和 2/3 件珍宝 3。这样总价值为 240 万元，总重量为 50kg。该最优算法如下：

```
FRACTIONAL-KNAPSACK (p,w,W)
Load =0;
i=1;
while (load<W &&i<=n) {
    if (wi<=W -load) {
        take item i;
        load=load+wi;
    }
    else {
        take (W -load)/wiof item i;
        load=W;
    }
    i=i+1;
}
```

显然，该算法时间的最坏情况仍然是 $O(n)$。

那么，这两个问题到底有什么不同而使得贪婪算法解决一个问题时获得最优结果，而另

一个不能呢？细心的读者也许已经看出了名堂。这个不同就是后面的问题具有所谓的贪婪选择属性！

5.4 贪婪选择属性

所谓的贪婪选择属性，就是指一个（全局）最优的解答是经由局部最优（贪婪）的选择而获得。

本章一开始就说过，贪婪策略也是一种分治策略，即将大问题化为小问题，然后在以最优方式解决小问题后获得大问题的解。与一般分治不同的是，贪婪策略每步解决的子问题数量为 1 个。对于背包问题来说，子问题就是每种珍宝的选择。而每一步需要决定的是对哪一种珍宝进行选择，即在所有子问题里面（所有珍宝种类的选择上）挑选一个子问题（一种珍宝）进行考虑。由于这种挑选是以价值重量比率最高为标准，因此是贪婪选择。如果经由一系列贪婪选择可以获得一个问题的最优解，则该问题具有所谓的贪婪选择属性。

因此，贪婪策略要想获得最优解，必须满足下面两个条件：

1）每个大问题的最优解里面包括下一级小问题的最优解。

2）每个小问题可由贪婪选择获得。

前面一个条件是最优子问题属性，该属性在动态规划里面也同样存在；而后面一个条件则是贪婪选择属性，该属性不是动态规划所必需的属性。正是该属性的存在，才使得贪婪策略在分解的时候总是只有一个子问题（即贪婪选择）。如果没有该属性，则贪婪选择将不能保证得到最优解。

但是，如果一个问题不具备上述条件，并不说明不能采用贪婪策略，只不过贪婪策略将不能保证获得最优解。这一点可以这么看：在分解大问题的时候，通常有多个子问题，但是贪婪选择永远只（贪婪地）选择一个子问题进行解答；如果该子问题的最优解不包括在大问题的最优解里面（也就是不具备贪婪选择属性），则贪婪策略将不能获得最优解。

当然，如果我们的目的不是获得最优解，而是一个次优或近似解，则不管一个问题是否具备贪婪选择属性，均可使用贪婪策略。事实上，在面对难解（intractable）问题或 NP 完全问题时（即有效解不存在的问题，将在本书第 15 章和第 16 章予以讨论），贪婪策略是我们使用的法宝之一。我们可以这么想，反正找不到最优解，何不使用贪婪策略来降低算法复杂性呢?

另外，有的问题一开始看上去似乎不具备上述两个条件，但可以将其转换为符合上述条件的等价问题，从而可以使用贪婪策略获得最优解。

下面我们以**教室规划**为例加以说明。

5.5 教室规划问题

假定某大学有 n 门课程都需要使用同一个教室来上课。显然，我们不能在一个教室同时

上两门或多门课。因此，每门课使用教室的方式是独享的。假定这 n 门课的集合为 $C=\{c_1, \ldots, c_n\}$，每门课需要使用教室的时间为 $[s_i, f_i]$，这里 s_i=开始时间，f_i=结束时间。我们的目标是设计一个算法对教室的使用进行规划以满足学校的具体要求。这个要求可以是容纳尽可能多的课程数，使教室的使用率最高，使最重要的课程都得到安排等。

这里我们假定学校的要求是容纳尽可能多的课程数。

例如，假定一共有 12 门课需要使用某个教室，它们的使用时间如表 5-2 所示。那么如何排教室才能使得能容纳的课程数最多呢？

表 5-2 12 门课程的起始和终止时间

i	1	2	3	4	5	6	7	8	9	10	11	12
s_i	0	2	4	3	6	8	9	11	12	10	13	15
f_i	3	5	7	8	10	10	11	15	15	13	17	18

也许，从表 5-2 中不太容易看出各门课程使用教室的先后关系，画个图就清楚多了，如图 5-2 所示。

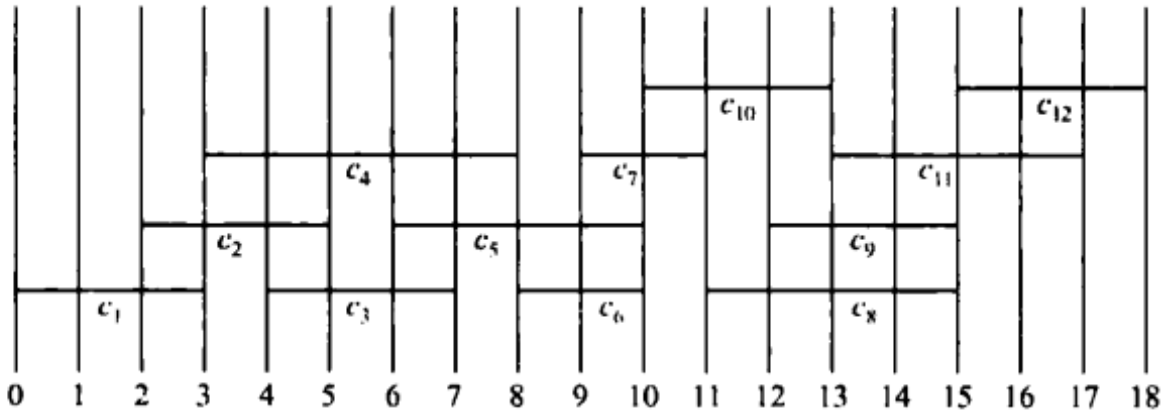


图 5-2 课程之间的时序关系图

由图 5-2 可以看出，能够容纳的最大课程数为 5 门课：$\{c_1, c_3, c_6, c_{10}, c_{11}\}$。但这个最大集合并不是唯一的，其他最大的课程集合有：$\{c_1, c_4, c_6, c_{10}, c_{11}\}$，$\{c_1, c_4, c_6, c_{10}, c_{11}\}$等。

这个问题如何求解呢？一个看上去直截了当的办法是分治：将所有的课程分解成两个或多个集合，分别对每个子集合里面的课程进行排课，排好后与其他子集合排好的课程合并即可。但问题是：如何分解这个集合呢？哪些课程应该在一个子集里呢？更为重要的是，对每个子集排好的课程是否可以直接与另外一个排好课程的集合合并呢？答案是否定的。因为一个子集排好的课程不一定与另一子集排好的课程之间没有冲突。除非你能找到一个分解方法使得分解出的每个子集合与其他子集合之间没有冲突。但这样又可能造成我们过于保守，从而合并出来的结果达不到最优。这样，剩下的唯一办法是在将所有可能的分解都进行考虑、比对后选择合适的分解。而这又将导致需要考虑的子集合为指数级！当然，如果这些子集合里面有重复的，那么我们可以使用动态规划来提高算法效率。

其实，我们没必要这样麻烦，因为这个问题有更好的解。那这个更好的解是什么呢？

要回答这个问题，先看看我们是如何得出前面例子里面最大集合的。

当然是看出来的嘛（开玩笑的）！课程数少的时候，也许画一张图查看一番即可，但要是课程数成百上千的时候，你还看得过来吗？再说，计算机不可能同时看到所有这些课程，它得一门一门地考虑！对，一门一门地考虑不是正好给了我们思路吗？

这个思路就是一门课一门课地考虑，在考虑每一门课的时候，要么将其加入到可排的课程集合里，要么将其排除在外（即不能容纳）。但关键是，一门课是否可以排得下，与别的课是有关系的，即在考虑一门课是否能排的时候，必须兼顾其他课。这样情况就会很复杂！

那么有没有办法使得我们在考虑一门课的时候，无需考虑别的课程呢？

要解答上述问题，需要对这个问题进行分析。首先我们需要将这个问题用公式表述出来。

设 S_{ij}= $\{c_k \in S: f_i \leqslant s_k < f_k \leqslant s_j\}$=所有在课程 c_i 结束后开始但在 c_j 开始前结束的课程，如图 5-3 所示。

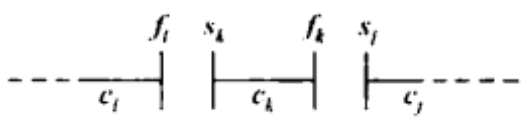


图 5-3 课程之间的时序关系

那么 S_{ij} 里面的课程与所有满足下列条件的课程兼容，即可以被同时容纳：

1）所有在 f_i 或之前结束的课程。

2）所有在 s_j 或之后开始的课程。

为了简化我们的表示，引入两个莫须有的课程 $c_0 = [-\infty, 0)$和 $c_{n+1} = [\infty, \infty+1)$，这样我们可将所有课程的集合 S 表示为 $S_{0,n+1}$。因此 S_{ij} 的下标范围为 $0 \leqslant i, j \leqslant n+1$。

与背包问题一样，我们假定所有的课程都以结束时间按照非递减方式排列，即

$$f_0 \leqslant f_1 \leqslant f_2 \leqslant \cdots \leqslant f_n < f_{n+1},$$

则 $i \geqslant j \Rightarrow S_{ij} = \varnothing$。因此，我们只需要考虑满足 $0 \leqslant i < j \leqslant n+1$ 的 S_{ij} 即可。

假如 S_{ij} 的某解决方案里面包含课程 c_k，则我们可以把 S_{ij} 分解为 S_{ik} 和 S_{kj} 两个子问题，而 S_{ij} 的某解决方案就是

$$(S_{ik} \text{的解决方案}) \cup \{c_k\} \cup (S_{kj} \text{的解决方案})$$

由于课程 c_k 不属于任何子问题，所以这两个子问题之间没有重叠，也就是说，

$$|S_{ij} \text{的解决方案}| = |S_{ik} \text{的解决方案}| + 1 + |S_{kj} \text{的解决方案}|$$

如果一个 S_{ij} 的最优解决方案包括课程 c_k，则 S_{ik} 的解决方案和 S_{kj} 的解决方案也必须是最优的。这一点可以很容易地证明，过程如下：

设 A_{ij} 是 S_{ij} 的一个最优解决方案，则 $A_{ij}=A_{ik} \cup \{c_k\} \cup A_{kj}$（这里假定 S_{ij} 非空且 c_k 已知）。因此，如果 A_{ik} 不是最优的，那么我们可以用最优的方案替换 A_{ik} 而获得一个更优的方案。这与假设矛盾。因此，A_{ik} 和 A_{kj} 是最优的。 □

这样我们就获得了解决排课问题的递归解法。令 $cs[i, j]$为 S_{ij} 里相互兼容的课程的最大门数。因为 $i \geqslant j \Rightarrow S_{ij} = \varnothing$，则有 $cs[i, j] = 0$。

如果 $S_{ij} \neq \varnothing$，且假定我们知道某课程 c_k 在该子集里，则 $cs[i, j] = cs[i, k] + 1 + cs[k, j]$。但事

实上，我们并不知道哪一个 c_k 在该子集里，因此，有：

$$cs[i,j]=\begin{cases}0 & S_{ij}=\varnothing\\ \max_{i<k<j,a_i\in S_{ij}}\{cs[i,k]+cs[k,j]+1\} & S_{ij}\neq\varnothing\end{cases}$$

为什么 k 的取值范围是这样的呢？因为由 $S_{ij}=\{c_k\in S: f_i\leqslant s_k<f_k\leqslant s_j\}$ 可以推出 c_k 不可能是 c_i 或 c_j，同时需要确保 c_k 在 S_{ij} 里（注意：条件 $i<k<j$ 不能保证 c_k 在 S_{ij} 里）。

到目前为止，这个问题看上去属于动态规划的范畴：

- 最优解里面包括的子问题个数为 2 个。
- 每次分解时可以选择的分解点为 $j-i-1$ 个。

用动态规划可以轻松地解决这个问题。但是，我们可以有更好的方法，而这个更好的方法依赖于下面的定理（或观察）。

定理 设 $S_{ij}\neq\varnothing$，且 c_m 是 S_{ij} 里面完成时间最早的课程，即 $f_m=\min\{f_k: c_k\in S_{ij}\}$，则：

1）c_m 必定属于 S_{ij} 里面某个最大兼容子集。

2）$S_{im}=\varnothing$，因此，选择 c_m 导致 S_{mj} 成为唯一非空的子集。

证明 证明第 1 条。设 A_{ij} 是 S_{ij} 的一个最大兼容集合。我们将该集合里面的所有课程按照结束时间进行非递减排序。

设 c_k 为 A_{ij} 里最早结束的课程，则下列情况必居其一：

1）如果 $c_k=c_m$，则证毕，因为 c_m 确实属于某个最大兼容集。

2）否则，构建 $A'_{ij}=A_{ij}-\{c_k\}\cup\{c_m\}$，即以 c_m 替换 c_k，则 A'_{ij} 是兼容集合。这是因为 A_{ij} 为兼容集合，c_k 是 A_{ij} 里完成时间最早的课程，$f_m\leqslant f_k$（这样 c_m 将不会与 A'_{ij} 里面的任何课程冲突）。由于 $|A'_{ij}|=|A_{ij}|$ 且 A_{ij} 是最大兼容集合，所以 A'_{ij} 也是最大兼容集合。

证明第 2 条。假设存在某个 $c_k\in S_{im}$，则 $f_i\leqslant s_k<f_k\leqslant s_m<f_m\Rightarrow f_k<f_m$。

所以，$c_k\in S_{ij}$ 并且其完成时间早于 f_m，而这与 c_m 的选择矛盾。因此，不可能存在任何 $c_k\in S_{im}$，所以 $S_{im}=\varnothing$。 □

到现在，我们已经看出来规律了：

- 最优解里面包括的子问题个数为 1 个。
- 需要考虑的选择数量为 1 个。
- 最优解里包含了子问题的最优解。

而这恰恰符合贪婪策略的要素：分解为一个子问题，且该子问题的最优解将导致全局最优解！这样，我们就可以使用贪婪策略来解决排课问题：

1）每次选择一门课时，选择完成时间最早的 $c_m\in S_{ij}$，这是我们的贪婪选择！

2）然后解决问题 S_{mj}。

例如，原始问题是 $S_{0,n+1}$。

假设我们第 1 个选择 c_{m1}，则剩下的子问题是 $S_{m1,n+1}$；

如果我们接下来选择 c_{m2}，则剩下的子问题变成 $S_{m2,n+1}$；

……

这样每个子问题的大小都比前面一个问题小 1，因此最多经过 n 次选择后结束。更为重要的是，每次选择的时候，我们只需要考虑一门课程，即完成时间最早的课程。

递归算法如下（假定所有课程已经按结束时间排好序）：

```
COURSE-SCHEDULER (s, f, i, n)
m=i+1;
while (m<=n&&s_m<f_i){      //找出第 1 个终结的课程
    m=m+1;
}
if (m<=n)
    return {c_m}∪COURSE-SCHEDULER (s, f, m, n);
else
    return ∅;
```

最初的调用是 COURSE-SCHEDULER $(s, f, 0, n)$ 。

当然，也可以用循环实现。我们只需要对 $c_{i+1}, c_{i+2}, \ldots, c_n$ 逐个检查，直到找到第 1 个与 c_i(满足 $s_m \geqslant f_i$)兼容的课程 c_m。如果找到这样的 c_m $(m \leqslant n)$，那么我们接着解决 $S_{m,n+1}$，并返回该方案和 c_m。如果找不到这样的 $c_m (m > n)$，则返回空集。

```
GREEDY-COURSE-SCHEDULER (s, f, n)
A={c_1};
i=1;
for (m=2; m<=n; m++)
      if (s_m>=f_i){
            A=A∪{c_m};
            i=m;
      }
return A
```

很显然，该算法的时间复杂性为 $\Theta(n)$。

5.6　最小生成树

如果让你对计算机进行组网，你需要在不同的计算机对之间建立连线。我们的目标是使所有的计算机连通，且连线的维护成本最低。你会怎么办呢？

这个问题可以转换为一个图论的问题：图中的每个结点代表一台计算机，结点之间的无向边代表潜在的网络连线，每条连线都有一个维护成本，即每条边都有一个权重。我们的目标是挑选足够数量的边使所有结点保持连通，并且使整个网络的维护成本最低，如图 5-4 所示。

我们要做的就是在该图中找出一个包括所有结点的连通子图，使得其所有边的权重之和最小。

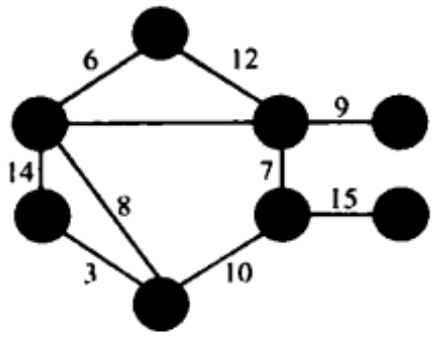


图 5-4 一个连通图

一个显而易见的事实是：最优的一组边必然不含有循环。因为，删除循环中的任意一条边将在保持所有结点连通的前提下降低整个网络的维护成本。

因此，最后找出的包括所有结点的连通子图必须没有环路。这种连通且没有环路的连通图就称为树，因为这种图可以以任意一个结点为杠杆提起来，从而形成一个由上往下的节节分支的倒树形状。在一个连通图里删除所有的环路而形成的树叫做该图的**生成树**。对于计算机组网问题来说，需要找出的树由于具有最小总权重，因此又称为**最小生成树**。

最小生成树在实际生活中的应用非常多，除了能在计算机网络及路由控制方面发挥作用外，它还可以在公路及铁路的路网规划，甚至人际关系网的构建等方面提供颇有意思的算法支持。

对这些生活中的问题进行抽象，我们获得最小生成树问题的定义如下：

输入：带权重的连通无向图 $G=(V,E)$，每条边的权重为 w_i。

输出：最小生成树，即一棵连接所有结点的树 T，并且 $w(T)=\sum_{(u,v)\in T} w(u,v)$ 为最小。

这个问题用什么方法解决呢？

由于生成树必须包括原图里的所有结点，所以关键的问题是在边上，即哪些边需要包括在生成树里，哪些边需要排除在生成树外。换一个角度看，就是对所有的边进行检查，找出 $n-1$ 条边，使得所有结点连通，并且权重最小！

这个问题也许可以用标准的分治算法来解，首先将所有的边（加上它们的结点）分解为两个或多个集合，然后考虑每个集合里面的边的选择问题，最后将子集合的解决方案合并。但这样将面临排课时遇到的同样问题，而且情况会更为复杂。

正如排课问题，这个问题的解答也可以通过对边一条一条地进行考虑来实现。即我们对图 G 里的每条边分别进行考虑，要么将其加入生成树，要么将其排除在外。问题是每条边的考虑与其他边是有关联的。因此要实现这种思维，就必须将这种关联消除，或者至少也要将这种关联考虑降低到可以管理的程度，也就是降低到多项式复杂性上来。

那怎么样才能达到这一点呢？我们先来看看最小生成树有什么特点。

定理 设有最小生成树 T（见图 5-5）。如果删除边 $(u,v)\in T$，则 T 将被分解为两个子树：T_1 和 T_2。因此，T_1 是图 $G_1=(V_1,E_1)$ 的最小生成树，T_2 是图 $G_2=(V_2,E_2)$ 的最小生成树。这里 G_1 是由 T_1 导出的图 G 的子图：$V_1=T_1$ 的所有结点，$E_1=\{(x,y)\in E:x,y\in V_1\}$；$G_2$ 是由 T_2 导出的图 G 的子图：$V_2=T_2$ 的所有结点，$E_2=\{(x,y)\in E:x,y\in V_2\}$。

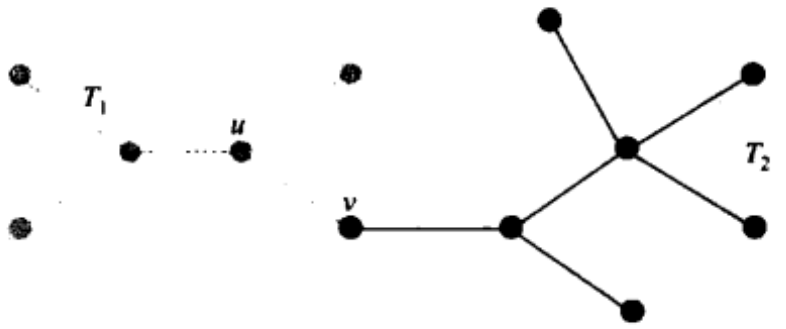


图 5-5 边(u,v)是连接子树 T_1 和 T_2 的唯一的边

证明 由于 T_1 和 T_2 的证明类似，所以我们只需证明 T_1 的情况即可。因为，

$$w(T) = w(u, v) + w(T_1) + w(T_2)$$

如果图 G_1 存在一个成本更低的生成树 T_1'，则 $T'=\{(u, v)\} \cup T_1' \cup T_2$ 将是 G 的一个成本更低的生成树，这与我们的假设（T 是图 G 的最小生成树）矛盾。 □

上述特性是最小生成树的最优子结构属性。有了这个属性，问题似乎可以用动态规划来解决。但如果用动态规划来解决最小生成树问题，其复杂性将是指数级的，因为在每层递归的时候需要考虑所有可能的子图的最小生成树问题，而子图的个数是指数级的。但是别着急，最小生成树问题还具备另一个非常可爱的属性，它能使我们的效率大为提高。

定理 设 T 为图 $G=(V,E)$的最小生成树，并且 $A \subseteq V$ 是图 G 里任意大小的结点集合。假设$(u,v) \in E$ 是连接 A 和 $V-A$ 的所有边里权重最小的边，则$(u,v) \in T$。

证明 使用反证法。假设$(u, v) \notin T$。

由于 T 是一棵树，因此从 u 到 v 存在唯一一条路径。该条路径必包含一条连接 A 和 $V-A$ 的边（否则这棵树将是不连通的）。如果将这条边以(u,v)来替换，那么我们将获得一个权重更低的生成树。而这与 T 是最小生成树矛盾。 □

该定理所阐明的属性也被称为最小生成树的**切割属性**。假定边集 T 是图 $G=(V, E)$的某个最小生成树的一部分。一个切割指的是将结点划分为两个集合：S 和 $V-S$。该属性的含义是，如果 T 不包括任何横跨这两个集合（S 和 $V-S$）的边，则挑选横跨该切割的权重最小的边（即边的一端在 S，另一端在 $V-S$）是安全的，即这条边必是某棵最小生成树的一条边（见图 5-6）。

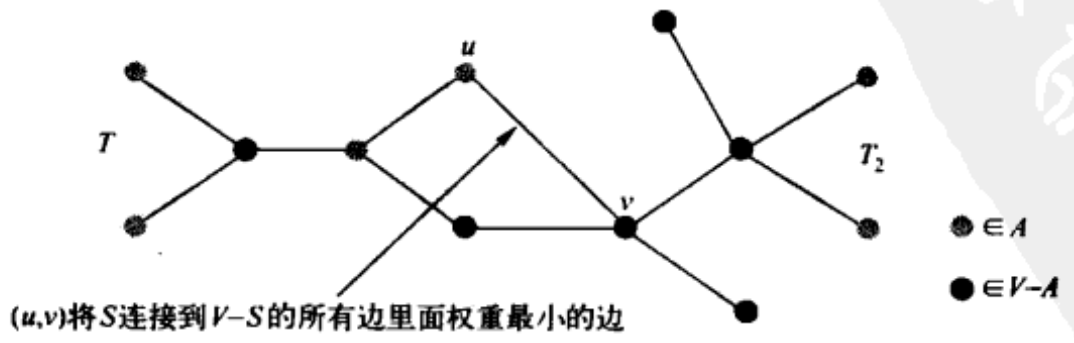


图 5-6 最小生成树的切割属性

这个切割属性就是最小生成树问题“贪婪选择属性”。

这条定理给出了一个计算最小生成树的直截了当的算法：

1）初始化一棵最小生成树 T，它包括图 G 的所有结点，但不包括任何边。

2）对图 G 里的所有边按权重从小到大进行排序，然后从小到大考虑每条边。

3）对于每一条边，如果它的加入能够连接图 T 里面两个不连通的部分（即这条边的加入不形成环路），则这条边就属于最小生成树，将其加入；否则，排除在外。

由于每次进行遴选的时候，选取的是当前未被加入到 T 里的所有边中最小的，所以上述算法自然是一个贪婪算法。该算法由美国数学家 Joseph Kruskal 于 1956 年提出（发表于《Proceedings of the American Mathematical Society》，pp. 48—50），所以被称为 Kruskal 算法。

Kruskal 算法的描述与我们上面给出的有些许不同，它的原始描述如下：

1）从空图 T 开始，按如下规则重复从给定图中遴选边，直到 $n-1$ 条边都包括在 T 中为止。

2）找到不（与已经选好的边，即图 T 里面的边）产生环路的权重最小的边，加入到图 T 中。

```
KRUSKAL-MST-ALGORITHM(G, E)
Q=E;                                    //将所有的边放置于队列 Q 里
i=0;                                    //最小生成树的边数初始化为 0
T=∅;                                    //最小生成树 T 一开始为空
while (Q≠∅&& i<n-1) {
u= EXTRACT-MIN(E);                      //在队列 Q 里抽取权重最小的边
if (u 不与 T 里面的边形成环路) {          //如果边 u 不与 T 里面的边形成环路，则加入 T 中
        T=T+{u};
        i=i+1;
    }
}
```

最后 T 里的边就是最小生成树。

Kruskal 算法是一条边一条边地构建最小生成树，除了注意每次选的边不能与已经选好的边之间产生环路外，只需要在剩下的边里选取权重最小的边即可。这样循环往复直到 $n-1$ 条边被加入到图 T 中为止。下面的图 5-7 至图 5-11 演示的是 Kruskal 算法的运算过程（其中选择第 5 条边和第 6 条边的过程被省略了）。

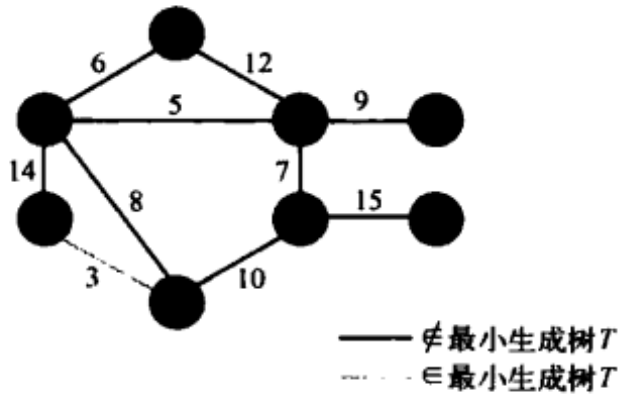


图 5-7 选取第 1 条边

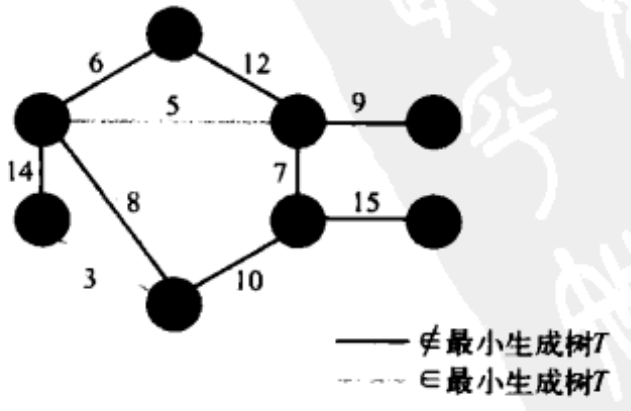


图 5-8 选取第 2 条边

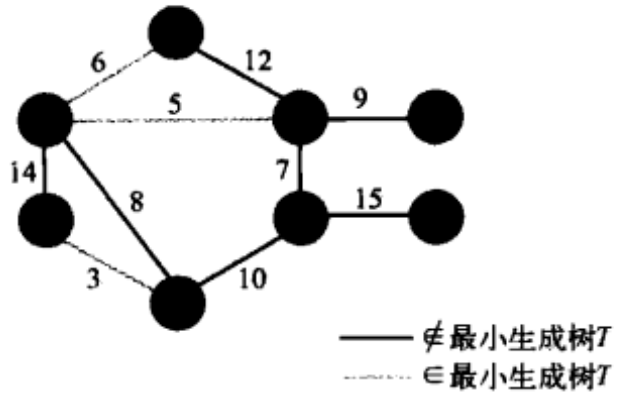


图 5-9　选取第 3 条边

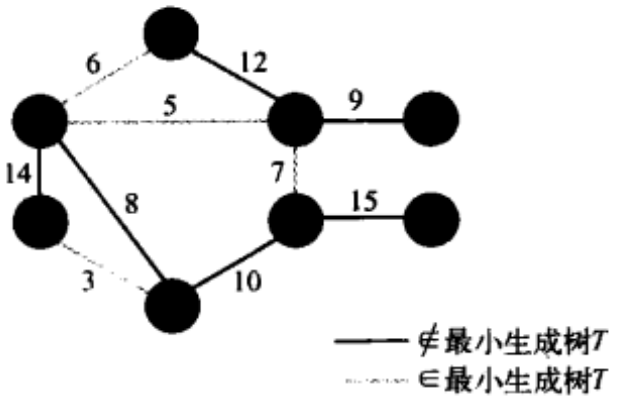


图 5-10　选取第 4 条边

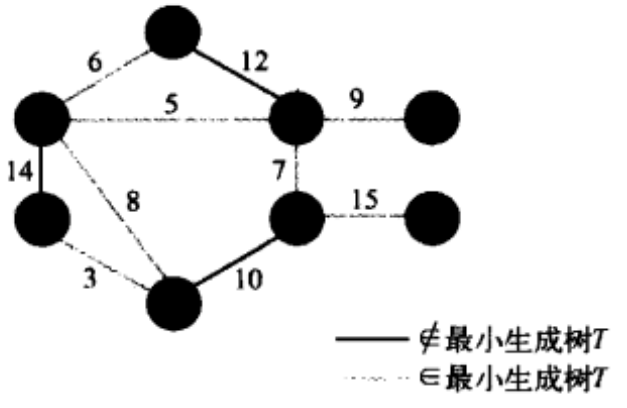


图 5-11　按同样规则选取第 5、6、7 条边后的结果

5.6.1　Kruskal 算法的正确性

现在来证明 Kruskal 算法的正确性。在任意时刻，已经挑选出来的边形成一个部分的解决方案：一组连通的组件，每个组件本身具有树的结构。下一条加进来的边 e 将把这些组件中的两个连接起来，让我们称其为 T_1 和 T_2。由于 e 是不产生环路的权重最小的边，毫无疑问，它是连接 T_1 和 $V-T_1$ 之间的权重最小的边，因此满足切割属性。

根据本章前面讨论过的切割属性性质，这样加上去的边都属于某棵最小生成树的边。因此，当加入的边数达到 $n-1$ 时，形成的必然是一棵最小生成树。

5.6.2　Kruskal 算法的时间分析

如果给出的边是无序的，则 Kruskal 算法需要 $O(|E|\log|V|)$的时间来对边进行排序（排序将在第 10 章讨论），而查找的时间复杂性也是 $O(|E|\log|V|)$（查找将在第 11 章讨论）。由于排序的时间复杂性已经处于最低状态，所以对查找的任何改善都将无济于事。在这种情况下，Kruskal 算法的时间复杂性就是 $O(|E|\log|V|)$。

但是，如果给出的边是按权重顺序排列的，或者权重的值都很小，那么我们就可以在线性时间内完成边的排序（使用插入排序和计数排序）。在这种情况下，查找的时间复杂性 $O(|E|\log|V|)$将超过排序的时间复杂性 $O(|E|)$，查找和合并成为整个算法的瓶颈。

我们可以将查找和合并做得更快吗？能。方法是使用分离集结构。只不过在对分离集进

行查找与合并的时候需要进行路径压缩，即随着我们由下往上寻找树的根，将下面的所有结点的指针都改为指向根，而不是它们的父结点，这样所有的路径都压缩到一层的高度（从所有结点直接到根结点）。此种结构的变动将查找的摊销（amortized）时间复杂性从 $O(\log n)$降低到 $O(1)$（摊销将在第 8 章讨论）。这样，在给出的边为有序排列的情况下，使用路径压缩可将 Kruskal 算法的时间复杂性降低到 $\max\{O(|E|), O(|V|\log|V|)\}$。该解法的具体实现留做习题。

5.7 Prim 算法

与 Kruskal 算法齐名的最小生成树算法是所谓的 Prim 算法。Kruskal 算法是一条边一条边地考虑，而 Prim 算法则是一个结点一个结点地考虑。这也是很自然的一种考虑。事实上，凡是牵扯到图的算法，都可以按边和按结点进行考虑。

我们可以从一棵空的最小生成树开始，每一步增加该生成树的尺寸，直到其包括所有的结点为止。这种思路就是 Prim 算法。该算法自然地用到 5.6 节讨论过的切割属性。该属性告诉我们，每次增加生成树的尺寸时，只要遵守贪婪的原则，即选择最小的跨越当前最小生成树内外的边，就会正确地产生最小生成树。

Prim 算法的时间复杂性依使用的数据结构不同而不同。在使用数组时，时间复杂性为 $O(V^2)$；使用（二元）堆结构时，为 $O(E\log V)$；使用斐波那契堆结构的情况下，为 $O(E+V\log V)$。

当然，我们考虑的方式仍然是贪婪。

由于边带有权重，所以贪婪的考虑一目了然。但按结点考虑，则需要给结点赋某种值，使得我们的贪婪选择有实现的基础。那么，给结点赋什么数值呢？

还是回到我们的定理，即切割属性上来。该属性告诉我们，如果将所有结点分为两个（分离）集合，则连接两个集合的所有边里面最小的一条边必然属于某棵最小生成树。这样，如果其中一个集合代表正在构建的生成树，另一个集合代表尚未考虑的结点，则我们每次从尚未考虑的结点里选取距离正在构建的集合最短（或最近）的结点将是正确的。

这就提示我们可以按照一个结点到其中一个分离集合（即正在构建的最小生成树）的最短距离来赋值。这个值当然是该结点到给定集合里面所有边里面最小的值。

这样，Prim 算法的轮廓就清晰起来了：

1）构建一个空的最小生成树 T，并将所有结点的赋值设为无穷。

2）选择图 G 的任意一结点，将其置于欲构建的最小生成树 T 里，另外一个结点集合为 $V\text{-}T$。

3）对 $V\text{-}T$ 里的结点赋值进行修改（因为 T 里多了一个结点，这些距离可能发生变化）。

4）从 $V\text{-}T$ 集合中选择赋值最小的结点，加入到 T 里。

5）如果 $V\text{-}T$ 为非空，则继续步骤 3～5；否则，算法终结。

```
PRIM-MST-ALGORITHM(G, V)
```

```
1. Q=V;
2. for (all v∈V)
3.       key[v]=∞;
4. key[s]=0;          //s 为 V 中任选的一点
5. while(Q≠∅) {
6.     u=EXTRACT-MIN(Q);     //抽取队列 Q 里面的最小元素
7.     for (each v∈Adj[u]) {
8.          if (v∈Q &&w(u, v) <key[v]) {
9.               key[v]=w(u, v); //降距操作
10.              π[v]=u;
11.         }
12.    }
13. }
```

在算法结束时，二元对组{(*v*, π[*v*])}给出的就是一棵最小生成树。

该算法一开始将所有结点分解为两个集合：*Q* 和 *V*–*Q*。开始时 *V*–*Q* 里没有元素，而 *Q* 中所有元素离 *V*–*Q* 里元素的距离为无穷。然后我们选择任意结点作为起始结点，并将该元素离 *V*–*Q* 集合里元素的距离初始化为 0。然后，算法的主要部分就可以开始了。

算法主要由第 5 步的 while 循环构成。该循环以队列 *Q* 里是否还有元素作为是否终结条件。只要 *Q* 里还有元素，算法就继续进行：从队列里面选取离集合最近的元素 *u* 加入到集合 *V*–*Q* 里，然后对每个与 *u* 有边直接相连的结点进行考虑（第 7 行的 for 循环）。

算法的第 9 个步对结点 *v* 到 *S* 的距离进行调整。这个调整是由新加入到 *S* 的结点触发的。每当一个新结点加入到 *S* 后，对于任意还在 *Q* 里的结点 *v*，我们只需要考虑该结点与新加入结点之间的边长 *w*(*u*, *v*)与当前已经计算出的 *v* 到 *S* 的距离 key[*v*]的相对关系。如果边长小于当前距离，就将当前距离调降为边长。因此，这个步骤也称为降距（decrease-distance）。

该算法由捷克数学家 Vojtěch Jarník 于 1930 年提出，后来被计算机科学家 Robert C. Prim 于 1957 年独立发现。1959 年 Edsger Dijkstra 重新发现该算法。因此，该算法有很多名称：DJP 算法、Prim 算法、Prim-Jarník 算法和 Jarník 算法。图 5-12 至图 5-17 为该算法的一个演示。

图 5-12 给出的是选取任意结点作为起始结点，将其加入到集合 *S* 后的状态。此时除源点到集合 *S* 的最近距离为 0 外，其他结点与 *S* 的距离都被初始化为∞。然后我们考察与 *S* 里唯一结点有边相邻的结点，发现 7、10、15 三个结点，我们降距其离 *S* 的距离后形成图 5-13。

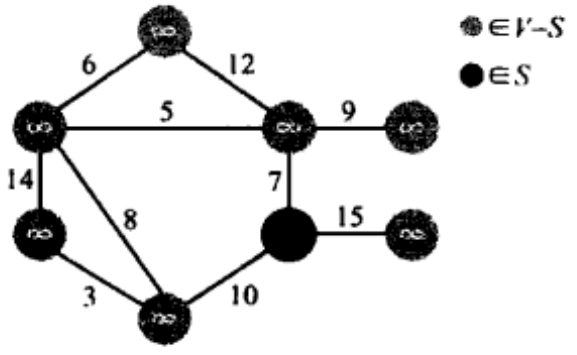


图 5-12　从任意结点开始，其他为∞

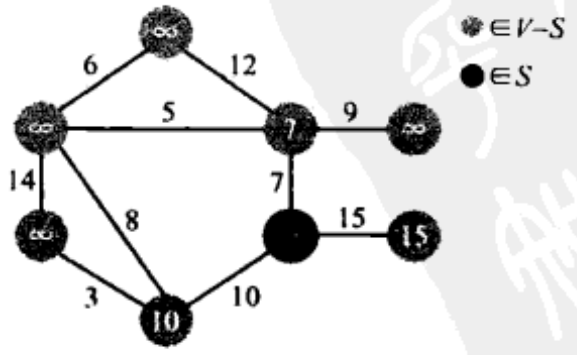


图 5-13　与源点 0 有边相连的结点被降低

在所有结点里，我们按照贪婪策略选取离 S 最近的结点，即 7，加入到 S 里，并对与结点 7 有边直接相连的所有结点进行距离降距，得到图 5-14。然后在所有 S 外面的结点里选择离 S 最近的结点，即 5，加入到 S 里，并将与结点 5 直接有边相连的结点进行距离降距，得到图 5-15。

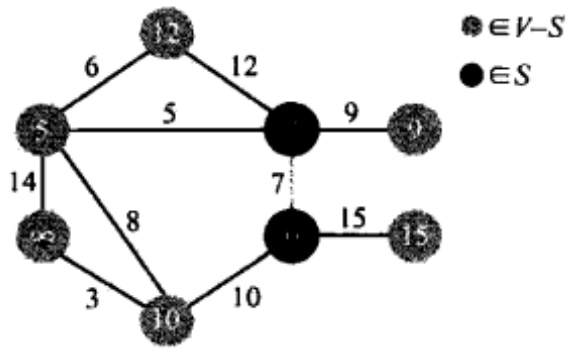


图 5-14 最近的结点 7 被加入

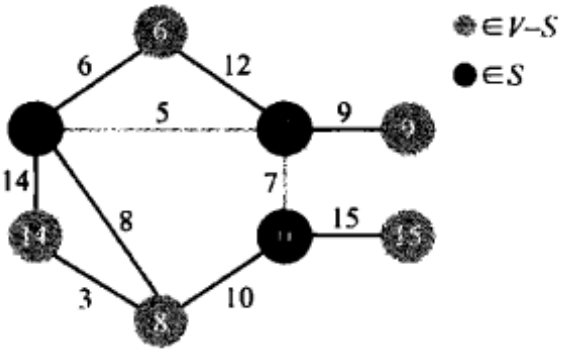


图 5-15 降距相邻结点，加入最近结点 5

然后选择离 S 最近的结点，即 6，加入到 S 中，并降距与结点 6 直接有边相连的结点的距离，形成图 5-16。这样循环往复，最后得到图 5-17。此时，最小生成树已经形成。

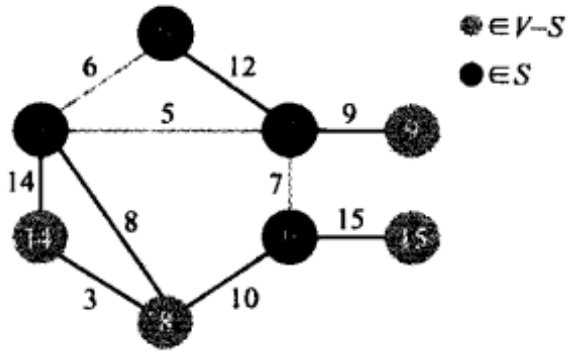


图 5-16 降距相邻结点，加入最近结点 6

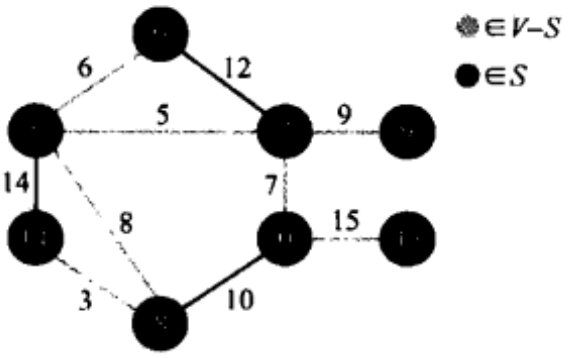


图 5-17 加入结点 8、9、14、15 后的状态

Prim 算法的分析

分析下面的 Prim 算法：

```
PRIM-MST-ALGORITHM(G, V)
1. Q=V;
2. for (all v∈V)
3.       key[v]=∞;
4. key[s]=0;                          //s 为 V 中任选的一点
5. while(Q≠∅) {
6.       u=EXTRACT-MIN(Q);
7.       for(each v∈Adj[u]) {
8.           if(v∈Q &&w(u, v) <key[v]) {
9.               key[v]=w(u, v);
10.              π[v]=u;
11.          }
```

```
12.      }
13. }
```

很显然，第 1～4 行的总成本为 $\Theta(V)$，第 7～11 行的内循环一共循环结点 u 的度数（degree(u)）次，第 5 行的外层循环的次数最多为$|V|$次。因此，该算法的时间复杂性为：

$$\Theta(V)T_{\text{EXTRACT-MIN}}+\Theta(E)\,T_{\text{DECREASE-DISTANCE}}$$

显然，$T_{\text{EXTRACT-MIN}}$ 和 $T_{\text{DECREASE-DISTANCE}}$ 所需要的时间与实现队列 Q 所使用的数据结构相关。如果直接使用数组，则每次选取最小元素需要的时间成本为 $O(V)$。（读者能否看出来如何在 $O(V)$时间内找出 V 个元素里面的最小值？）而对一个结点进行降距操作为常数时间。因此，在使用数组作为队列 Q 的实现机制时，Prim 算法的总时间成本为 $O(V^2)$。

如果使用二进制堆来存放 Q 的元素，则抽取最小元素的时间成本将降低到 $O(\log V)$，但在堆里对一个元素进行降距操作所需要的时间也是 $O(\log V)$，从而导致 Prim 算法的总时间成本为 $O(E\log V)$。如果图为稀疏图（即 E 远远小于 V^2），则该时间复杂性要好于数组的情况；如果图为稠密图，则时间复杂性接近 $O(V^2\log V)$，显然不如使用数组的情况。

如果使用改进的斐波那契堆来实现队列 Q，则抽取最小元素的时间成本为 $O(\log V)$（摊销成本），但降距的时间成本将重新回归到常数时间（也是摊销成本），从而使 Prim 算法的总时间成本降为 $O(E+V\log V)$，好于数组或二进制堆的情况。这里的摊销成本分析指的是针对一组操作进行最坏情况分析后得出的每个具体操作的成本。本书将在第 8 章节专门论述摊销分析。

表 5-3 给出的是在不同数据结构下，Prim 算法的时间复杂性。

表 5-3　Prim 最小生成树算法的时间成本

Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-DISTANCE}}$	总　计
数组	$O(V)$	$O(1)$	$O(V^2)$
堆	$O(\log V)$	$O(\log V)$	$O(E\log V)$
斐波那契堆	$O(\log V)$ 摊销时间	$O(1)$ 摊销时间	$O(E+V\log V)$ 摊销时间

这里需要指出的是斐波那契堆是一种所谓的“可归并的堆”。它实际上是一组堆（或树）的集合。此种数据结构由于理解困难和编程过程复杂，所以在实际上并没有多少价值。因此，表 5-3 里给出的 $O(E+V\log V)$摊销时间成本只有理论上的价值。在实际中，Prim 算法的时间成本通常被认为是 $\min(O(V^2), O(E\log V))$。

5.8　霍夫曼树和霍夫曼编码

按照热力学第二定律，整个宇宙是向着越来越混乱、越来越没有用的方向发展（有没有发现，汽车碰撞之后总是变得更破烂，而不是变得更好）。而衡量一个封闭体系混乱程度的度量就是熵。熵是一个系统随机程度的衡量。熵越大，该系统的随机程度越高。宇宙所有物质的熵都在持续增大（当然，为了使进化论不与热力学第二定律矛盾，我们只能对热力学第二定律的应用加以限制：即虽然整个系统的熵是不断增大，但局部范围内的熵可以减少）。

问题是我们怎么知道一个系统的熵有多大呢？或者我们如何断定这个世纪比上个世纪更加混乱呢？（当然，有人认为今天远胜于昨天。）你也许觉得这个似乎与算法没有什么关系。但这是真的吗？难道计算熵的时候不需要使用算法？

虽然物理学上如何测量或计算熵不是我们关心的问题，但在算法上我们有个简单的办法来估算一个系统的混乱程度。要衡量一个系统的熵，首先要对该系统进行取样，即按某种标准对系统在不同时间点上进行赋值。取样后对所获得的数据进行**无失真压缩编码**，即在保持所有信息量不丧失的情况下使用最短的编码！压缩编码的总长度越短，该系统的熵越小，即随机程度越低；压缩编码的长度越长，则系统的熵越大，随机程度越高。

这与我们的直观理解非常符合：压缩编码越短，说明某些模式出现的频率很高（因此数据压缩的比例就大），即系统呈有序排列的概率更高，因而随机程度越低。

那么如何进行压缩编码，从而使得同样的数据压缩出来的编码最短呢？

这显然是一个优化问题。不过这个问题比起排序和最小生成树的问题简单多了。

重新叙述这个问题的描述：

输入：给定符号表和一段消息（数据）。

输出：给每个符号赋以一个编码，使得给定消息在此编码下的长度最短。

例如：给定符号 A、C、S、T 和消息 CAST CAST SAT AT A TASA。我们如何编码呢？

最直截了当的办法当然是均匀编码，既然一共有 4 个字符，我们用两位二进制数编码即可，如：C:00 A:01 S:10 T:11，这样上面消息编码后的结果为：00011011 00011011 10011101111001 11011001，一共 36 位。

由于这个办法地球人都会，应该不会是最好的办法。

直观经验告诉我们，不是所有的字符都获得同等频率的使用。有的字符出现频率高，有的出现频率低。如果采用均匀编码，则这种频率高低的不同将不能被利用！事实上，如果采取这种方式，那么我们根本就不需要知道待编码的信息是什么。

略微想想可推测，如果要使一个消息编码后的长度最短，显然应该将出现频率高的符号赋以更短的编码，而出现频率低的符号可以使用较长一点的编码。例如，如果使用编码：A:0 T:10 S:110 C:111，则上述消息的编码结果为 11101100 11101100 110010 010 0 1001100，一共 33 位。比起均匀编码短了 3 位！

这种按照频率进行区别编码，频率越高，其编码越短；频率越低，其编码越长的可变长编码就是所谓的霍夫曼编码。该编码由当时还是麻省理工学院博士生的 David A. Huffman 在 1952 年发表的论文“A Method for the Construction of Minimum-Redundancy Codes”里首先提出。

从上面的描述中，可以得出霍夫曼编码的概念是:

1）对所有符号按照出现或使用频率由高到低进行排序。

2）顺序取出一个符号。

3）赋以当前可用的最短编码。

4）如果还有符号没有编码，则重复第 2～3 步；否则，结束。

显然，霍夫曼编码符合一切贪婪的属性：每次选取使用**频率最高**的符号，而赋以的则是

当前可用的**最短编码**！因此，霍夫曼编码是一种贪婪策略！

问题是，霍夫曼编码到底是如何得出或者实现的呢？使用频率最高比较容易判断，但什么是当前最短的编码呢？或者，具体地说，我们怎么知道进行 A:0　T:10 S:110 C:111 这种编码？又如何知道这是最优的呢？

要解答这个问题，最好的方式就是霍夫曼树。

5.8.1　霍夫曼树

我们来观察给定的消息 CAST CAST SAT AT A TASA。在 4 个符号里，A 使用了 7 次，居第 1；T 使用了 5 次，居第 2 位；S 使用了 3 次，居第 3 位；C 使用了 2 次，居最末。

如果我们以这 4 个字母构建一棵树，而目标是整棵树带权重的路径总长为最小。一棵树带权重的路径总长度 WPL=$\sum_{i=0}^{n-1} w_i l_i$，这里 w_i 为叶子结点 i 的权重，l_i 为该叶子结点到树根的距离（分支次数）。例如，对于 4 个叶子结点，权重分别为 1、2、3、4，可形成如图 5-18 所示的树，其中中间一棵树的权重路径长度为最小。这种带权重的路径总长度最小的二叉树就被称为霍夫曼树。

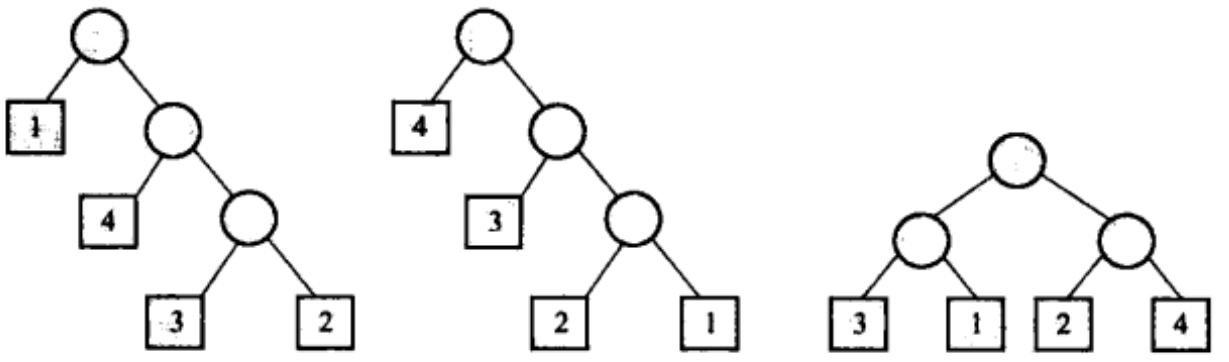


图 5-18　霍夫曼与非霍夫曼树

根据对霍夫曼树的描述我们可以发现，在构建这棵树的时候应该将频率高的结点置于离树根较近的位置上，这样树的路径总长度才可能最小。而根据这个推导，我们获得构造此种树的算法如下：

1）给定权重值集合 $W=\{w_0, w_1,\cdots,w_{n-1}\}$。

2）首先构造一个森林 F，它由下面的 n 棵树构成：

a）一个结点的二叉树 $T_0, T_1,\cdots,T_{n-1}$。

b）每个 T_i 只有一个权重为 w_i 的根结点。

3）然后，循环往复执行下列步骤，直到霍夫曼树被完全构成为止：

a）在森林里选取根权重最小的两棵树。

b）将它们合并为一棵树（通过增加一个根作为该两个结点的父结点）。

c）合并后的树的根结点权重为两棵子树根结点权重之和。

d）将刚才使用过的两棵树从森林中删除，将新合并的树加入到森林 F 里。

例如，如果用该办法对 A、C、S、T 四个符号构造霍夫曼树，将获得如图 5-19 所示的

总权重为 17 的结果。

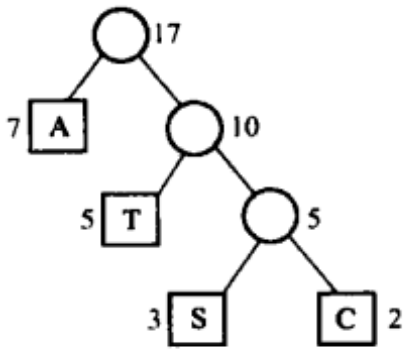


图 5-19 根据 A、C、S、T 四个符号使用频率构造的霍夫曼树

5.8.2 霍夫曼编码

有了霍夫曼树后，进行霍夫曼编码就是非常容易了：从霍夫曼树的根结点开始，顺着分支往下，左边分支赋值 0，右边分支赋值 1（也可用反过来），直到所有分支上都有赋值为止。此时，一个结点的编码就是从根结点到该叶子结点路径上所有赋值的连接。例如，对图 5-19 的霍夫曼树进行编码，获得图 5-20 的霍夫曼编码。编码树的编码结果就是：A:0 T:10 S:110 C:111，这正是前面讨论过的最优编码（之一）。

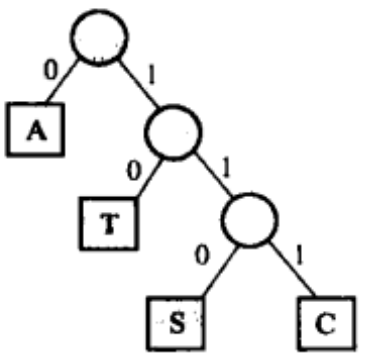


图 5-20 霍夫曼编码

再举一例：假如我们要编码的消息是：In the beginning was the word。这里面包含 13 个符号，这些符号和它们的出现次数如表 5-4 所示。

表 5-4 符号出现的频率与编码

符 号	出现次数	均匀编码	变长编码
A	1	0000	00000
B	1	0001	00001
D	1	0010	00010
E	3	0011	100
G	3	0100	101
H	2	0101	0011
I	3	0110	110
N	4	0111	111
O	1	1000	00011
R	1	1001	00100
S	1	1010	00101
T	2	1011	0100
W	2	1100	0101
	5	1101	011

如果使用均匀编码，由于有 14 个符号，所以需要使用 4 位，这样获得的编码如表 5-4 中“均匀编码”所示。而上述消息的编码结果为：

01100111110110110101001111010001001101000110011101110110011101001101110000001010110110110101001111011100100010010010

一共 116 位。

如果按照符号的出现次数构建霍夫曼树，则结果如图 5-21 所示。

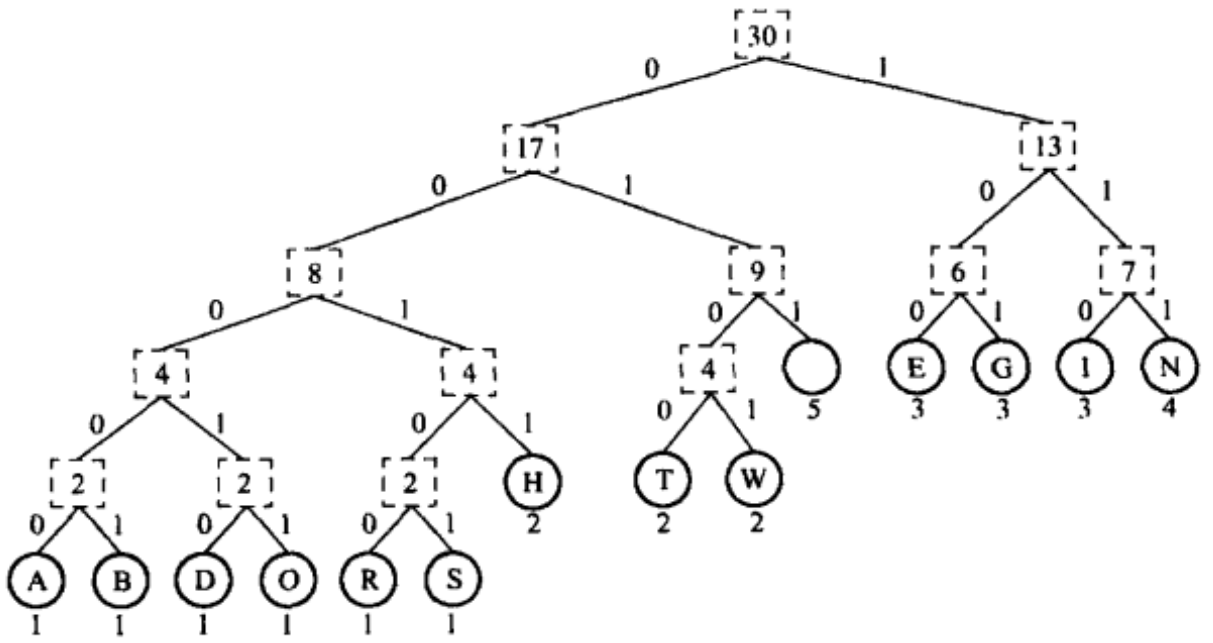


图 5-21　“In the beginning was the word”的霍夫曼树

在该霍夫曼树的每条边上赋值 0 和 1，得到图 5-21 的结果，将从根结点到叶子结点的所有分支上的 0 和 1 收集起来，就获得一个叶子结点的编码，其编码如表 5-4“变长编码”一栏所示，而上述消息的编码结果为：

11011101101000011100011000011001011101111111101111010110101000000010101101000011100011010100011001000010

一共 81 位，节省了 25 位，节省率为 21.6%！

5.8.3　霍夫曼编码的无前缀编码性质

霍夫曼编码的核心是按照符号出现的频率来进行编码：频率越高，编码使用的位数越少；频率越低，使用的位数越多。这种可变长编码是一种打破平均的策略。就像我们平时说的“吃大锅饭（均匀编码）效率不高，而让一部分人先富起来（长短不一编码）就可以提高整个社会的效率”一样。使用可变长度编码看上去是打破了大锅饭，提高了效率，但也带来一个潜在的问题：编码后的消息可能有多种译码方式。例如，如果某个字母的编码是另一个字母编码的前缀时，就可能产生译码歧义。因此，在可变长编码下不产生译码歧义就是霍夫曼编码要解决的一个（主要）问题。而不产生歧义就是不能出现一个字母编码是另一个字母的编码的前缀的情况。因此，这种不产生歧义的可变长编码也称为**无前缀编码**。霍夫曼编码恰恰满足了无前缀编码的要求。

5.9 进程调度问题

进程调度是操作系统实现进程模型的两大根本手段之一。在多进程并发的环境里，虽然从抽象上看有多个进程在同时执行，但在单一 CPU 下，实际上在任何时刻只能有一个进程处于执行状态，其他进程则处于非执行状态。这就有一个需要解决的问题：如何确定在任意时刻到底执行了哪个进程呢？这就涉及进程调度：选择下一个要运转的进程。

要知道如何选择进程，就要知道进程调度的目标。不同的关注点可导出不同的调度目标，但良好的系统响应时间通常为其中的一个目标。这是因为人类缺乏耐心，希望系统能够很快响应用户的请求。当我们提交一个程序执行时，我们希望很快获得结果，结束运行。

如何设计调度算法，才能使系统的响应时间最短呢？

先来看一个例子。设有程序 1、2、3，其执行时间分别为 4、5、6 秒。而三个程序的排列顺序有六种，其响应时间如表 5-5 所示。

表 5-5 不同进程调度顺序的响应时间

调 度 次 序	程序 1 响应时间	程序 2 响应时间	程序 3 响应时间	总 计
1、2、3	4	4+5=9	4+5+6=15	28
1、3、2	4	4+6+5=15	4+6=10	29
2、1、3	5+4=9	5	5+4+6=15	29
2、3、1	5+4+6=15	5	5+6=11	31
3、1、2	6+4=10	6+4+5=15	6	31
3、2、1	6+5+4=15	6+5=11	6	32

从表 5-5 中看出，1、2、3 的调度次序的响应时间最快，总计 28 秒。由于 1 的执行时间最短、3 的时间最长，因此 1、2、3 的调度次序从执行时间上看似乎是由小到大，即有点贪婪选择的味道。那么，贪婪选择战略是否总是获得最佳的响应时间呢？

答案是肯定的，因为排列在前面的进程的执行时间会被重复计算在系统的总响应时间里。例如，如果有 n 个进程，则排在第 1 的进程的执行时间会被计算 n 次，排在第 2 的进程会被计算 $n-1$ 次，以此类推，最后一个进程的执行时间仅被计算一次。这样，如果执行时间短的进程排在执行时间长的进程前面，那么被重复计算次数多的为较短执行时间的进程，其总响应时间自然比重复计算次数多的为较长执行时间的进程的调度要短。上述结论也可以很严谨地用反证法证明，具体证明留给读者完成。

这样，我们就获得了最佳响应时间的贪婪调度算法：每次选择执行最短的进程执行。

5.10 贪婪选择属性

与动态规划一样，贪婪策略是一种分治策略。与动态规划不同的是，贪婪策略将待要解

决的问题分解为一个子问题（而不是动态规划里面的多个子问题）！这个选择加上对剩下子问题的最优解将合成对原问题的最优解。不过需要注意的是，贪婪策略在分解的时候所做出的选择是当前看上去似乎最好的选项，而不肯定是最优的选择。因此，贪婪策略并不是总能获得最优解。但如果一个问题具有所谓的贪婪选择属性，则可以保证我们的贪婪选择是最优的。从而导致贪婪算法成为最优算法。因此，贪婪策略要想获得最优算法，在考虑使用贪婪策略时必须抓住问题的两个重要特征：

1）贪婪选择属性。

2）最优子结构。

事实上，只有在一个问题具有最优子结构和贪婪选择性质时，使用贪婪策略才能获得最优结果。因此，在使用贪婪策略的时候，一般战略是：

1）发掘面临问题的最优子结构。

a）对输入进行某种处理使其形成一种贪婪次序。

b）如果是动态数据，使用优先队列。

2）构造一个递归，将一个问题的解答归结为对其子问题的解答。

3）证明最好子问题的选择是一个贪婪选择。

a）查看全局最优解决方案。

b）如果该方案包括贪婪选择属性，则结束。

c）否则，修改最优解决方案使得其包括贪婪选择，而结果仍然是最优。

4）证明在贪婪选择后，只有一个非空子问题剩下。

5）构造递归的贪婪算法。

6）如果需要，将递归算法转换为循环算法。

由于一个问题通常按照多种方式分解，因此，在考虑问题如何分解的时候要时刻注意那些可能导致贪婪选择的问题分解方法。例如，对于排课问题，我们分解问题的方式是按照课程结束的时间进行，从而导致贪婪选择性质成立：总是选择最早结束的课程，并且选择后只剩下一个子问题。然后又进一步证明其最优子结构，即我们的贪婪选择 c_m（S_i 里面最先完成的课程）加上对剩下唯一子问题 S_m 的最优解就是原问题 S_i 的最优解。

使用贪婪策略的典型步骤如下：

1）将原优化问题表述为一个做出一个选择，然后剩下唯一一个子问题的形式。

2）证明在所有的最优选择里面总有一个是贪婪选择。

3）证明贪婪选择加上对剩下子问题的最优解导致原问题的最优解。

如果贪婪选择属性不成立，则使用贪婪策略将得不到最优结果。例如，在计算机里面对磁盘进行调度的时候，最短寻道优先算法就是一种贪婪策略，即每次都处理寻道时间最短的磁盘读写请求进行处理，这样以期达到总寻道时间最短的目标。这种调度策略听上去似乎挺好，但实际上达不到最优目标。例如，如果磁头当前处于第 10 号位置，所有请求如图 5-23 所示，则最短寻道调度方式的处理顺序如图中的折线所示。总寻道数为 1+3+7+15+31=57，而最优的调度的总寻道数为：1+5+16+23+8=53。

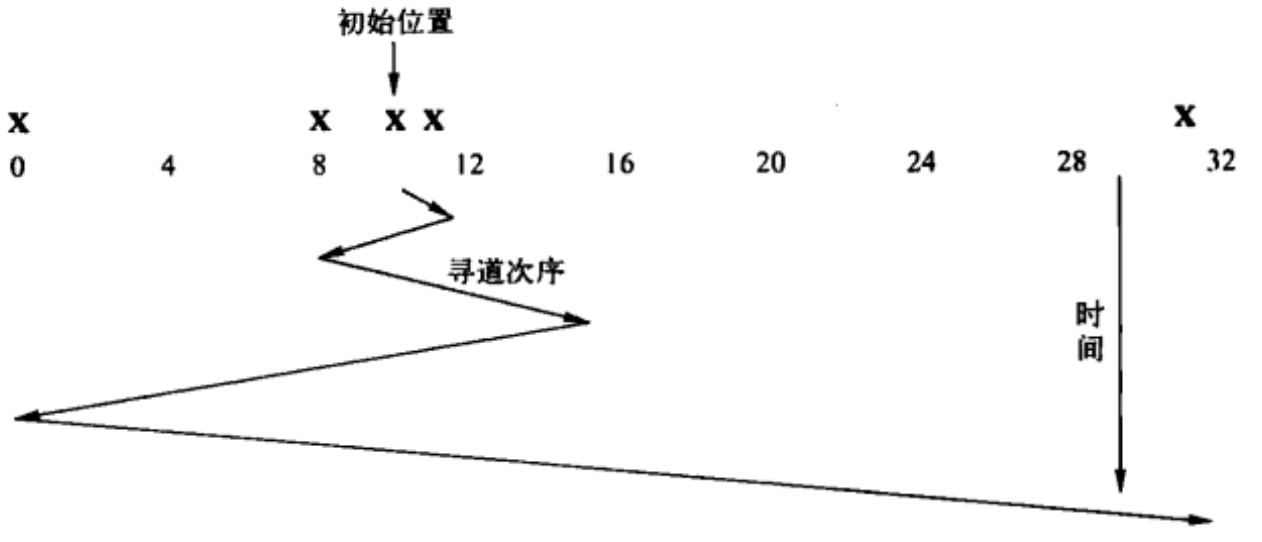


图 5-22 贪婪磁盘调度策略

5.11 标准分治、动态规划和贪婪选择的比较

标准分治、动态规划、贪婪选择可以说是孙子兵法里面的下、中、上三策。标准分治虽然将大问题分解为小问题，但每个小问题都要一个个解决，相当于逢城必攻，属于下策；动态规划则聪明地发现很多子问题相同，并不需要重新解决，就是不对每个城市都进行攻城，从而节省兵力和精力，但仍然需要攻克子问题中的相当部分，属于中策；而贪婪策略则将子问题限于一个，即将攻城数量减少到了最低，从而最大地节省了精力和兵力，属于上策。

不过就像兵法里面所云，上策运用得不好，也会有失策的时候。贪婪思想运用得不当，或在条件不充分或不明朗的情况下运用，则会大败而归！

从另一个方面看，三种策略都是为了在求解问题的时候使成本尽量低。因此，从这个方面看，三种策略的目标一致。但是，标准分治策略的目标只是获得问题的解，动态规划和贪婪选择则不仅要获得一个解，而且应该是个最优解。因此，贪婪选择与动态规划之间的相似性更多。

虽然动态规划和贪婪选择有许多相似的地方，但有时并不是很容易看出来。下面，我们主要给出贪婪选择和动态规划的比较。

贪婪选择与动态规划

贪婪选择属性指的是一个全局最优解决方案可以通过做出一个局部最优的选择（贪婪选择）获得。对于贪婪算法来说，可能大家都看出来其做出的选择具有贪婪的性质。但对于动态规划来说，这一点似乎不太容易看出来。动态规划进行了贪婪选择吗？

如果仔细察看动态规划就可以发现，动态规划的每一步都需要做出一个选择，例如在最长公共子序列问题中，我们需要选择在什么地方将序列断开。但这个选择是贪婪选择吗？答案是肯定的。只不过此时我们的选择不是在不知道子问题解的情况下做出的，而是在子问题

已经解出来的基础上进行的。而我们选择的是最好的分解，因此当然是贪婪选择。

动态规划是先解决子问题，自底向上，因此，每次选择我们都知道是正确的。而贪婪策略是在解决子问题前做出选择，然后希望做出的选择是正确的，是自顶向下。但不管是先做出选择还是在子问题解决后再做出选择，都是试图做出最好的选择！表 5-6 是对标准分治、动态规划、贪婪选择三种策略的比较。

表 5-6　标准分治、动态规划、贪婪选择三种策略的比较

	标准分治	动态规划	贪婪选择
适用类型	通用问题	优化问题	优化问题
子问题结构	每个子问题不同	很多子问题重复	只有一个子问题
最优子结构	NA	必须满足	必须满足
子问题数	全部子问题都要解决	全部子问题都要解决	只要解决一个子问题
子问题在最优解里	全部	部分	部分
选择与求解次序	先选择后解决子问题	先解决子问题后选择	先选择后解决子问题

思考题

1. 假定你需要与 n 个有权贵的人维持沟通渠道，但维持这种关系显然是需要成本的（送礼，甚至是送大礼），而你又不是一个资源无限的人。因此你希望以最小的成本维持与所有权贵的关系。请问有何算法能够使用（假定你不是那种“安能摧眉折腰事权贵，使我不得开心颜”的人）？给出详细的答案。
2. 你是 X 国的皇帝，因年事已高需要考虑继承人的问题。但你高山仰止的高尚品德使你不愿意开创家天下或党天下或裙带天下的恶劣先例，因此，决定在全国海选皇帝继承人。经过无数激烈的较量，有 n 个人胜出成为候选人。现在，作为皇帝，你需要在 n 个候选人里面选出一个人来接替你的位置。请从算法的角度提供几种策略，并分析比较每种策略的优劣。主要考虑的问题：是否保证选出最佳人选和获得该最佳人选需要的成本。
3. 背包问题应该有 4 个版本。除了本章论及的财宝可以部分地拿取和不可以部分拿取两种版本外，还有每件财宝的件数有限和无限两种版本。每一种版本对应的解的性质都不尽相同。如果财宝不可以部分拿取，但每样财宝的件数无限，请问贪婪策略能够获得最优解吗？为什么？有什么办法获得最优解呢？
4. 如果财宝不能重复取的话，第 3 题的解答会有什么不同？
5. 假定你开车去香格里拉。出发前你的油箱是满的，可以行驶路程 d。路上一共有 n 个加油站，加油站之间的距离由数组 A[1..n]给出，这里 A[i] 从第 i–1 个加油站到第 i 个加油站之间的距离。最后一个加油站正好在旅程的终点——香格里拉。你希望沿途停车加油的次数越少越好。请设计一个算法，计算沿途需要停车加油的地方，并给出输入和输出。（假定 A[i] $\leqslant d$）。
6. 你在餐馆吃饭花了不到 100 元人民币，结账时，你给服务员一张 100 元钞票，而服务员

希望用张数最少的纸币找给你。假设餐馆提供数目不限的面值为 50 元、20 元、10 元、5 元和 1 元的纸币。请问，服务员应该如何找零才能达到所有纸币张数最少的目标？

7. 证明：若 T 为图 G 的一棵最小生成树，S 为 G 的一个连通子图，则 $T \cap S$ 属于 S 的某棵最小生成树。
8. 给出下面消息的霍夫曼编码："In life we all search for something"。该消息的霍夫曼编码与固定长度编码相比，空间节省率为多少？
9. 喜欢动态选择和喜欢贪婪选择的人有什么显著不同？你喜欢哪种选择？为什么？
10. 本章在讨论 Kruskal 算法时说过，在给出的边为有序排列的情况下，通过使用分离集和路径压缩，可以将查找的摊销时间复杂性从 $O(\log n)$ 降低到 $O(1)$，从而将 Kruskal 算法的时间复杂性降低到 $\max\{O(|E|), O(|V|\log|V|)\}$。请实现此种改进的 Kruskal 算法并给出其时间复杂性的详细推导过程。

CHAPTER 6

第 6 章　随机化思想

凡牛群、羊群中，一切从杖下经过的，每第十只要归给耶和华为圣洁。不可问是好是坏，也不可更换；若定要更换，所更换的与本来的牲畜都要成为圣洁，不可赎回。

——摘自《圣经 · 利未记》

上述两句圣经经文是神耶和华吩咐给摩西的缴纳十一税的方法，也就是算法。该算法要求，在牲畜回栏的时候，举起一根手杖，每数到第 10 头，即归为耶和华（如图 6-1 所示）。

图 6-1　用什么算法从羊群里面选出十分之一作为奉献呢

初看，这个算法没有什么神奇之处。但仔细思索，却发现其中的精妙设计。对于一个非常虔诚的人来说，他总是会将上好的牛羊奉献给神。但这样的话，他剩下的牛羊的质量就会较差，这样久而久之，其牛羊必然每况愈下。对于一个滑头的人来说，他则总是会尽量挑选质量差的牛羊奉献给神，而这当然是神所不悦的。

为了不让虔诚人的牛羊质量每况愈下，也为了防止滑头的人总是挑选质量差的牛羊来奉献，神耶和华给出了这个用手杖分离出 1/10 牲畜的办法。这种思想就是随机化思想，因为每个第 10 头牲畜的质量好坏是人不可预料的，也就是随机的。使用随机化思想的算法就是随机算法！

随机从某种程度上说就是运气。在算法中，使用随机就是将算法的某个部分置于运气的控制之下。这种做法也许看上去有点向命运投降的意思，似乎不应该受到推崇。但事实恰恰相反。随机化算法在很多领域获得了广泛的应用。也许你不会相信，世界上一些最快、最智能的算法依赖于机会（或运气），即在这些算法中的某一步骤（也可以是某几个步骤），通过抛硬币来决定算法的下一步行为。本章要详细讨论的就是这种随机化算法。

6.1 为什么要随机化

到目前为止，本书讨论的算法都是确定性的。这个确定性体现在下面几个方面：

1）每个步骤是确定的，即在某一步到底该做什么是确定的。

2）算法的结果是确定的，即输出的东西是什么事先就知道。

3）步骤的数量是确定的，即算法的时间复杂性是确定的。

也许读者会说，我们并不知道一个算法在运行时到底需要运行多少步骤，也不知道它会输出何种结果，而且每个步骤到底做什么我们也不知道呀！如果读者有这个疑问，那是因为尚未准确理解“确定性”的含义。确定性不是指你作为一个用户对一切事情都完全掌控，而是指一旦输入确定，一切都已经确定，只不过你还不知道而已。就像一个学生考研面试完后我们已经做出是否录取的决定，只不过学生自己尚不知道，因而还会忐忑不安。

也许换一个角度可能更容易理解：一旦输入给定，我们就可以根据这个输入一步一步推导出运行的过程，而不会存在任何模糊。例如，我们前面介绍过的排课、最小生成树问题的算法都具备上述特性，因此都是确定性算法。对于排课来说，一旦输入给定，我们知道排出来的肯定是一个最大课程集合，而且具体是哪一个课程集合也已经确定。最小生成树也是如此。

然而，确定性却并不是所有算法都必须具备的一个特性。事实上，有时候我们还真需要一点随机性。20 世纪 80 年代，本人还是一个中学生，一天与朋友路过一个棋局摊子，摆摊人摆出的是一个中国象棋残局，任何人都可以执红棋以先手挑战摆局者。如果挑战失败，挑战者须付给摆局人 1 元钱；如果挑战成功，则摆局人付给挑战者 2 元钱。我的朋友和路边的一些人纷纷挑战，但无一例外败下阵来。最后在朋友的催促下，我亦决定试试运气。我开局随机地走了一步没有什么用处的棋。由于这步棋在残局的棋谱里没有，摆局人一下子想不起如何应对（看来不是专业棋师），走了一步非常差的应着。抓住对方的这次失误，我赢得了挑战。摆局人在付给我 2 元钱后，拉着我要求再下一局。当然，我没有答应，因为，这种随机化策略如果再次使用（在这种特殊情况下），恐怕效果就要大打折扣了。

当然，生活中需要随机的时候很多。在派对上进行抽奖使用的是随机抽奖，彩票的中奖号码需要随机产生（如图 6-21 所示）。如果抽奖和中奖是事先确定的，那么很多人会不高兴的（被安排中奖的人例外）。作为计算机核心软件系统，操作系统的调度策略也有一个伪随机调度策略（彩票调度）。

图 6-2　大部分人都希望彩票开奖号码是随机选出来的

如果仔细思考上面给出的例子，我们发现，随机化所起到的作用是防止某种特别令人不满的结果的出现。用本书引子的例子来说，就是要防止好人吃亏和坏人得逞。不过，这并不是随机化的唯一功用。

读者一定经历过各种考试（如果你从来没有经历过考试，而你又在看这本书，那么这件事情本身就是一个奇迹）。而考过试的人都有这样的经历：不知道某些题如何解答，怎么办呢？放弃？当然不。至少胡乱写一通，捞点同情分！而对于选择题来说，放弃就更加不明智了。总共有 4～5 个选项，随便选一个答案也有 20%～25%的概率答对。而放弃就没有可能得分。

这种在不知道如何解答问题的时候胡乱地写一通或任意选择一个答案就是随机策略，即随机是我们在不知所措的时候所采取的一种策略。这有点类似古代圣人在遇到疑难问题时的占卦，其实占卦本身并不会启示将来发生的事情（至少对于相信进化论的普罗大众来说，这是不可能的，也是不能相信的），它起的作用就是帮助占卦者做出决定。这就是“疑难困惑之时，圣者占卦以示之”的含义。而这也就是随机化的另一个目的，即解答我们解答不了的问题。虽然这个解答不一定正确！将此结论推广到算法，随机化算法的另一个目的就清楚了：解答我们不能解答的问题。不，更准确地说，是解答我们不能确定性地解答的问题！

6.2　随机的平方

关于随机，存在着两派针锋相对的观点：一派观点认为，随机在这个世界并不存在，一切都是冥冥中自有天意，或者说万事都有因果；但另一派观点认为，这个世界本来就是随机的，从来就不存在任何的因果关系，一个人的成功纯属运气，与能力毫无关系（其实还有第三派观点认为既有随机也有因果）。当然，本书无意于涉及这种永无休止的争论中来，我们关心的是，在算法中使用随机有无必要。如果一切都是随机的，那么随机化算法就似乎没有存在的必要了：难道有必要在随机上面再随机一次吗？

本书当然认为随机化算法非常有必要。

首先，我们并不肯定这个世界是完全随机的。至少对于我们很多的成功人士来说，他们拒绝这一说法（在他们眼里，他们的成功完全或一大部分取决于他们自己的能力或实力）。既然世界不是完全随机的，那么在某些时候随机就很有必要：防止坏人得逞！

其次，就算是世界是随机的，我们还是可以再随机一次，使其更加随机或者成为随机的平方，更加不可琢磨。这样不更好吗？也许你的对手运气很好，能够猜出来一次随机，但他能够猜出来两次随机吗？至少困难多了吧。

6.3 什么是随机化算法

前面说过，随机化算法就是将算法的某一步或某几步置于运气的控制之下，即该算法在运行过程中的某一步或某几步涉及一个随机决策，或者说其中的一个决策依赖于某种随机事件。例如，我们可以使用一个随机数产生器来产生一个随机数，再根据这个随机数的值来决定下一步的行为：如果随机数产生的数值为 1，算法执行 A 操作；否则，执行 B 操作。

这是否有点像抛硬币呢？抛出正面执行一个操作，抛出反面执行另一个操作。从某种程度上看，随机化算法就是抛硬币的算法：在某一个步骤通过抛硬币来决定下一步行动的方向（如图 6-3 所示）。

而随机化的目的，前面已经说过，主要有两个：

1）防止对手破坏，防止好人吃亏。

2）解决确定性算法不能解决的问题。

为第一种目的而设计的随机化算法称做拉斯维加斯算法(Las Vegas Algorithm)。读者一定知道拉斯维加斯是美国著名的赌城，而对于一个赌博者来说，什么最重要？当然是公平！即需要防止赌场暗中做手脚，以免赌徒吃亏。而随机化恰恰是保证公平的重要手段。（难道还有其他保证公平的手段吗？）因此，为此目的而设计的随机算法就叫做拉斯维加斯算法（如图 6-4 所示）。

图 6-3 随机化算法就是用抛硬币来决定下一步行动的算法

图 6-4 赌城拉斯维加斯：拉斯维加斯算法就因其而命名

为第二种目的而设计的随机化算法称为**蒙特卡罗算法**(Monte Carlo Algorithm)。不过，这

里的蒙特卡罗与摩洛哥的赌城蒙特卡罗并无关系，而是发明此种算法的人为纪念抚养过他的叔叔而起的名字（看来，他叔叔的名字里一定包含蒙特卡罗这个词）。

对于拉斯维加斯算法来说，虽然使用了随机策略，但这个策略为的是保证公平，因而计算的结果仍然是正确的。而为了要保证公平，就不免要费点时间。因此，拉斯维加斯算法保障计算结果的正确性，但不保障算法的时间效率。蒙特卡罗算法主要是用来解决难解即复杂性很高的问题，因而其主要追求的是算法的时间效率。而为了这个效率，就不得不牺牲准确性甚至是正确性。因此，蒙特卡罗算法保证算法的运行时间，但不保证算法结果的正确性。

也许读者会觉得，一个算法若不能保证正确性，还有什么意义呢？

当然有意义。就像考试的时候，你不能保证解答正确的题都放弃吗？当然不。正如占卦无法启示未来，人类不还是一直在进行各种与占卦相关的活动吗？人类社会酷爱进行的各种预测从根本上说都是占卦，因为谁也不知道这些预测是否正确。只不过预测时用到的占卦工具不是签条、龟片，而是更加好听一点的各种统计数据和随机模型而已。

使用蒙特卡罗算法的重大意义在于它有很大的可能计算出正确的结果。虽然随机选择答案并不能保证选择的正确性，但恰恰因为是随机，它就有正确的可能！而只要有正确的可能，为什么不试一下呢？其次，使用此种算法求解的问题都是确定性算法无法或者很难解答的问题。既然解答不了或解答的复杂性人类无法承受，那么用随机算法来试一下又有何妨呢？另外，有时候，我们并不需要一个问题的精确结果，而只需要大概的估算即可。例如，对于一场战争的预估，很多时候我们只需要知道是否能赢，而打赢战争所需要的具体成本就不是特别关心了。商业风险分析也是类似。

这里，我们必须时刻牢记蒙特卡罗算法的结果不一定正确。这样，对待蒙特卡罗算法给出的结果，我们就会心中有数。

6.4 拉斯维加斯算法

前面已经说过，拉斯维加斯算法的目的是保障公平，即为了不让对手处于优势，或者说为了不让最坏的情况发生，我们给算法增加了一个随机的步骤。而增加一个随机步骤是要增加成本的，因此，拉斯维加斯算法是以牺牲效率来换取公平。（这好像很值得嘛！我们不是为了公平甚至愿意抛头颅洒热血吗？）。当然，这个增加不应该对算法产生过大的影响。

虽然在大部分时候，我们可以将这个随机步骤所增加的算法成本限制在可以接受的水平，但在某些特殊情况下，这个成本仍有可能太大，以至于我们不得不放弃使用此种算法，也就是说放弃公平（如果维护公平的成本太高，我们当然放弃）。

因此，拉斯维加斯算法的特点是：

1）永远产生正确答案，即从来不给出错误答案。

2）通常情况下算法的效率不错，但有时候会很慢。

以赌博来比喻，拉斯维加斯算法赌的是资源的使用，而不是结果的正确性。

如果一个问题存在拉斯维加斯算法，并且该算法的运算时间的期望值为多项式，则该问

题称为**零错概率多项式时间问题** (Zero-error Probabilistic Polynomial Time Problem)。

一般来说，使用拉斯维加斯算法的基本思路是：产生一个合适的随机数，然后根据这个随机数的取值决定下一步行动。能够淋漓尽致展示拉斯维加斯算法的例子是随机化快速排序。排序的杠杆点的选择是随机的，但排序的结果却是正确、确定的。本书将在第 10 章对随机化快速排序进行详细论述。此外，本书后面章节讨论的多个算法都是拉斯维加斯算法。因此，本章暂不对拉斯维加斯算法进行举例，而将例子留到后面的章节。

6.5 蒙特卡罗算法

前面已经提到，蒙特卡罗算法是用来解决那些确定性算法无法或难以解决的问题。由于其通过随机选择来判断解答问题的方向，所以出现错误当然就在所难免。虽然如此，我们还是希望蒙特卡罗算法能够在大部分时候产生正确的结果，或者产生的结果与正确结果之间的距离不要太过遥远。如果达不到这样的目的，那么使用蒙特卡罗算法的意义也就不大了。因此，使用蒙特卡罗算法对使用者的素质是有较高要求的。

就像一个对算法一窍不通的人来考试算法，即使他随机地选择一番，恐怕也得不了多少分！也就是说，一个没有任何把握的人来进行算法考试基本上是没有意义的，一个对运算结果毫无感觉的随机算法也是没有多少意义的。这就是说，即使做随机选择，我们也要有一点概念，了解应该如何做这种随机选择。

由此可见，蒙特卡罗算法的重要特点是：

1）大部分时候输出正确答案。

2）一部分时候输出错误答案，但错误的概率有限。

3）资源的使用是确定的。

注意，上述第 3 点是至关重要的。没有这一点，蒙特卡罗算法就没有存在的意义了。（你看出来了吗？）与拉斯维加斯算法相比，蒙特卡罗算法将正确性，而不是资源的使用，放在赌博的桌子上，即我们宁愿算错，也不愿意投入无限的精力。

蒙特卡罗算法的另外一个属性则是启发式，或者说，它是一种启发式算法（以占卦来启示）。如果我们能够获得多遍启示，则计算的正确性应该提高。因此，如果一个蒙特卡罗算法运行多遍，其正确性应该不断得到加强。一个经典的例子是拉宾（Rabin）的素性测试算法：对于任何合数 n，则一个随机选择的 x 至少有 75%的概率证明该数为合数。因此，如果 n 的确不是素数，但 x 说它可能是素数，则我们观察到一个发生概率只有 25%的事件。如果 10 个不同的随机 x 都说 n 是素数，则我们观察到的是一个发生概率只有百万分之一的事件。

一般来说，蒙特卡罗算法的思路是：产生一个合适的随机数，然后观测有多少这样的随机数遵守某一性质。然后根据这个比例来估计结果。例如，在计算数学里，通过幸运地选择找出正确答案。本章的后面几节就来以具体的例子来说明蒙特卡罗算法的巨大功用。

6.6 素性测试

本书前面的章节已经提到过素性测试问题：给你一个自然数 N，判断 N 是否是素数。

正如我们前面已经论述过，最直接的素性测试办法是将所有大于 1 但小于该数的整数作为潜在因子，一个一个地检查。如果其中的某个数确实是因子，则该数不是素数。如果所有潜在因子都被证明不是真的因子，则该数就是素数。当然，这种判断需要检查 N-2 个数。

但我们注意到，如果 N 真的能被分解为两个大于 1 的因子之积，则至少其中一个因子将小于等于 $\sqrt{N}$。因此，我们实际上只需要检查从 2 到 $\sqrt{N}$ 的数即可。不过这种改善还是不够的。因为它仍然是指数级的（以 N 在计算机里面的表示位数来看）。看来，对于人类来说，素性测试似乎是很难的一件事。

有没有办法改善素性测试的时间复杂性呢？

显然，按照素数的定义，要想确定一个数是素数，唯一的办法就是确定它没有除 1 和它自身以外的任何因子。因此，从素数定义上看，我们的指数级算法没有任何改进余地了。

但由于素数的重要性（计算机安全有赖于此，很多散列算法也有赖于此），我们必须进行突破。要突破，就必须绕开素数的直接定义，从而避开因式分解这个难题。因此，问题就变成：是否只能根据素数的定义来进行素性判定呢？或者说，我们能否找到素数的另外一种性质，它可以很容易被检测，但合数又不具备呢？

如果有这样的性质，则问题就能得到解答。

问题是，这样的性质存在吗？

答案既是“是”又是“否”。

“是”是因为确实存在一种素数的性质，这种性质很容易被检查和验证；“否”是因为这种属性某些合数也具备，但是具备此性质的合数数量很少。这种性质对于一个确定性算法来说当然是不够的，但如果我们愿意牺牲一部分正确性，即把正确性拿到赌桌上去赌，则这个性质就绰绰有余了。这种思路就带来了随机化素性测试算法。

当然，发现这个性质与素性测试的关系并不是一蹴而就的，而是经过了漫长的岁月和很多人的研究才得到的。这个性质就是费马小定理。

费马小定理 *如果 p 是素数，则对于任意整数 a，a^p-a 可以被 p 整除。以模算术表示为：*

$$a^p \equiv a \bmod p$$

费马小定理的另外一种表述是：如果 p 是一个素数，a 是与 p 互为素数的整数（即 a 不包含因子 p），则 $a^{p-1}-1$ 可以被 p 除尽。以模算术表示就是：

$$a^{p-1} \equiv 1 \bmod p$$

费马小定理就是随机化素性测试算法的理论基础。

根据费马小定理，我们可以设计一个随机化素性测试算法：随机选取一个小于 N 的正整数 a，判断 $a^{N-1}=1 \bmod N$ 是否成立。如果成立，则宣布 N 为素数；如果不成立，则 N 为合数。

但遗憾的是，这个随机化素性测试算法的结果并不一定正确。因为费马小定理给出的只

是一个必要条件，而不是一个充分条件。即 $a^{N-1}=1 \mod N$ 是 N 为素数的必要条件，但不是充分条件。换句话说，某些合数也能通过费马小定理测试。

幸运的是，对于合数 N，绝大部分的整数 a 都通不过费马小定理测试。这样只要测试的 a 值数量达到一定的程度，我们就将检测出 N 为合数。因此，我们可以将随机化素性测试算法重复很多次，每次使用一个不同的 a 值，从而将测试结果的正确概率提高到任意小于 1 的值（但永远也不会等于 1）。我们的希望是，对于一个合数来说，当测试的 a 值增多时，总有一个通不过费马小定理的测试！

至此，一切看上去都非常好。

但不幸的是，宇宙中存在一种称为卡米克尔（Carmichael）数的合数，该数可通过所有的费马小定理的测试。此种合数由美国数学家罗伯特·丹尼尔·卡米克尔发现。对于这种数，无论我们选取何种 a 值，费马小定理都坚挺不倒。

由此可见，基于费马小定理的素性测试算法在遇到卡米克尔数时将出现错误。但是，幸运的是（没办法，人生就是“幸运”和“不幸”的交替更迭），卡米克尔合数的数量极其稀少，一般的人或者一般的应用不太可能会碰到它。当然，如果你真的碰上这种数，你也只能认栽了。（换个角度看，你应该庆幸，因为碰到卡米克尔数的概率比中六合彩的概率低多了，所以你运气不错，赶快去买彩票！）

假定我们没有幸运到天天能中彩票的程度，即假定我们不需要考虑卡米克尔数，则我们就可以设计一个蒙特卡罗算法来进行素性测试，并且可以使错误的概率达到任意小的程度，即将测试结果正确的概率提高到任意接近 100% 的值。下面我们就来设计这个算法。

素性测试的蒙特卡罗算法

我们的蒙特卡罗素性测试算法非常简单，它直接由费马小定理导出：

```
RANDOMIZED-PRIMALITY-TEST(N)
1. 随机选择一个正整数 a<N;
2. if(a^(N-1)≡1 mod N)
        return "YES";
3. else
        return "No";
```

根据前面的分析，这个算法不能保证100%的正确。但它到底能够保证正确到什么程度呢？我们说过，如果抛开卡米克尔数，其他任何合数至少会在一个 a 值上通不过费马小定理的测试。而从这一点又可推导出一个重要的引理。

引理 *如果存在一个相对 N 为素数的 a 使得 $a^{N-1}\neq 1 \mod N$，则小于 N 的所有数里面存在至少一半（$N/2$ 个）满足这种条件的整数。*

证明 如果存在整数 a，有 $a^{N-1}\neq 1 \mod N$，则对于任意小于 N 且满足费马小定理测试的整数 b，整数 ab 也将使得费马小定理测试不成立：

$$(a\cdot b)^{N-1}\equiv a^{N-1}\cdot b^{N-1}\equiv a^{N-1}\neq 1 \mod N$$

□

因此，在抛开卡米克尔数不论的前提下，我们就可以断定：

- 如果 N 是素数，则 $a^{N-1}=1 \bmod N$ 对于所有的 $a<N$ 都成立。
- 如果 N 是合数，则 $a^{N-1}=1 \bmod N$ 对于至多一半的 $a<N$ 成立。

这样，算法 RANDOMIZED-PRIMALITY-TEST 就具有一种我们很喜欢的概率行为：

- 如果 N 为素数，则算法回答“是”的概率为 1。
- 如果 N 为合数，则算法回答“是”的概率小于等于 1/2。

由于算法 RANDOMIZED-PRIMALITY-TEST 的概率行为，我们可以将该算法运行多次，从而将 N 为合数且算法回答“是”的概率降低到任意小（但大于 0）的一个值。例如，如果运行该算法 100 次，则该算法判断错误的概率只有 2^{-100}。这个概率甚至小于一个宇宙随机射线在计算机正在进行素性测试时击毁该计算机的概率。

6.7 矩阵乘积验证器

本书第 3 章已经讨论过矩阵乘法的问题，我们花费了很多精力，历经了许多曲折，终于将矩阵乘法的算法效率从 n^3 提高到 $n^{2.81}$。虽然后来有人在各种特殊情况下将此效率提高到 $n^{2.795}$ 和 $n^{2.376}$，但将时间复杂性提高到平方级似乎是难以做到的。

矩阵乘法的效率也许无法被提高到平方级，但我们却可以在平方级时间效率下验证一个矩阵乘积是否正确。而这个验证算法依赖于随机化。

矩阵乘积验证器的问题定义如下：给定 3 个 $n\times n$ 矩阵 $\boldsymbol{A}$、$\boldsymbol{B}$、$\boldsymbol{C}$，要求验证 $\boldsymbol{AB}=\boldsymbol{C}$ 是否成立。

很显然，如果要确定性地验证 $\boldsymbol{AB}=\boldsymbol{C}$，唯一的办法是将 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 相乘，计算出结果，然后与 $\boldsymbol{C}$ 进行比较。而这样就无法将验证的时间效率提高到平方级。

因此，要提高验证的效率，我们必须避免将 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 相乘。那么我们面临的问题是，如何在避免进行 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 相乘的情况下，验证 $\boldsymbol{AB}=\boldsymbol{C}$ 是否成立呢？显然，要验证正确性，某种相乘操作是避免不了的。问题是，能不能不要进行这么多的乘法？矩阵乘法的效率之所以低，是因为两个矩阵的维数都是 $n\times n$ 的。如果其中一个是 $1\times n$ 或者 $n\times 1$，也就是矢量，则这样两个矩阵的乘法操作可以在平方级时间内完成。

根据矩阵乘法的法则可知，如果 $\boldsymbol{AB}=\boldsymbol{C}$，则等式两边乘以同一个矢量后等式仍然成立，即

$$\boldsymbol{zAB}=\boldsymbol{zC}$$

这里 $\boldsymbol{z}$ 是一个 $1\times n$ 的矢量。这个新等式的验证可以在平方级时间内完成。因为 $\boldsymbol{zAB}=(\boldsymbol{zA})\boldsymbol{B}$，而 $\boldsymbol{zA}$ 能够在平方级时间内完成，并且其结果仍然是一个矢量。而结果矢量乘以 $\boldsymbol{B}$ 的操作同样也是平方级。这样，我们就获得一个基本的算法思路：

1）选择一个随机二进制矢量 $z[1..n]$，并且 $\Pr[z_i=1]=\frac{1}{2}$，$\Pr[z_i=0]=\frac{1}{2}$，$i=1,\ldots,n$。

2）然后检查 $\boldsymbol{zAB}=\boldsymbol{zC}$。

3）如果该等式成立，则判定原等式也成立，也就是 $\boldsymbol{C}$ 确实是 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 的乘积。

4）如果该等式不成立，则原来的等式也不成立。

问题是，这样的想法站得住脚吗？

假定 $D=AB$，我们需要检查等式 $D=C$ 是否成立。显然，我们只需要考虑两种情况：

1）$D=C$，则毫无疑问 $zD=zC$，因此我们算法的输出将是“相等”。

2）$D\neq C$，则一定存在某些 z，有 $zD\neq zC$。即在某些矢量 z 下，该算法的输出将是“不等”。

上述第 2 点隐含的意思是即使 $D\neq C$，在乘以矢量 z 后，$zD=zC$ 也有可能成立。因此，使用这种乘积验证器有可能得出错误的结论：即两个不等的矩阵被判为相等。这就是说，我们的算法将结果的正确性放在了赌博桌上。既然赌博，我们就必须估计胜算的概率。

我们误判的概率有多大呢？很显然，误判不会超过 50%。因为两个相等的矩阵不会被判为不等。如果给出的矩阵相等和不相等的概率均等，则即使不等的矩阵全部被判为相等，误判的概率也只有 50%。但是，50%的概率显然太高了。我们需要更大的把握。

我们误判的概率到底有多大呢？这就需要知道在所有可能的矢量 z 里，有多少个矢量使得在 $D\neq C$ 的时候 $zD\neq zC$？这样的矢量到底有多少个呢？

首先我们注意，如果 $D\neq C$，则 D 至少有 1 行与 C 不同。假设它们第 i 列不同，如图 6-5 所示的 d 列和 c 列不等，即 $d\neq c$。而这一行的不同又说明，它们至少有一个元素不同，假设 $d_i\neq c_i$，那么，$zd=zc$ 当且仅当 $z(d-c)=0$。而

$$z(d-c)=z_1(d_1-c_1)+z_2(d_2-c_2)+\cdots+z_i(d_i-c_i)+\cdots+z_n(d_n-c_n)$$

$$=\sum_{j\neq i}z_j(d_j-c_j)+z_i(d_i-c_i)=z_i(d_i-c_i)$$

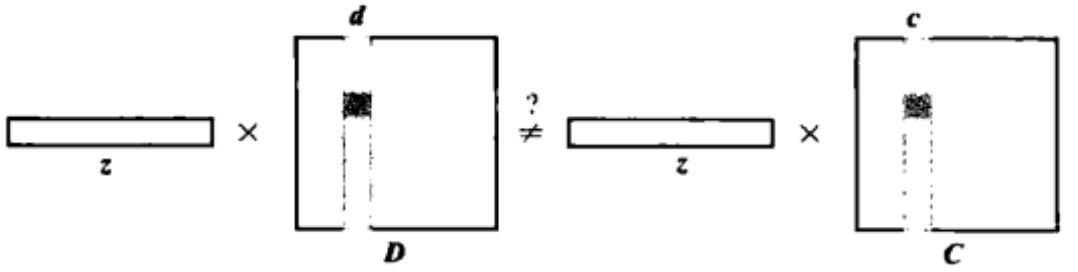


图 6-5 矩阵 D 和 C 的第 i 列不同，$d\neq c$

因此，$z(d-c)=0$ 当且仅当 $z_i(d_i-c_i)=0$，并且 $\Pr[z(d-c)=0]=\Pr[z_i(d_i-c_i)=0]$。那么 $\Pr[z_i(d_i-c_i)=0]$是多少呢？由于 $d_i\neq c_i$，$z_i(d_i-c_i)=0$ 只能在 $z_i=0$ 的情况下达到。而 z_i 可以随机取 1 或者 0，因此 $z_i=0$ 的概率是 1/2，也即 $\Pr[z_i(d_i-c_i)=0]=1/2$。

因此，在有一个元素不同的情况下，两个矩阵被判为相等的概率为 1/2。

如果 D 和 C 有两个元素不等呢？这种情况下 D 和 C 被判为相等的概率是多少呢？

我们可以分几种情况来分析：

1）如果这两个不等的元素出现在同一列，则当矢量 z 对应这两个位置的元素均为 0 的时候，D 和 C 将被判为相等。此概率为 1/4；当矢量 z 对应这两个元素的位置为一个 0、一个 1 的时候，D 和 C 将被判为不等，此概率为 1/2；当矢量 z 对应这两个元素的位置为二个 1 的时候，并且 D 里面这两个元素之和等于 C 里面对应的两个元素之和时，D 和 C 将被判为相等，此概率非常的小，肯定远远小于 1/4。因此，此时我们有 $\Pr[zc=zd]<1/2$。

2）如果这两个不同的元素出现在同一行，则 $\Pr[zc=zd]=1/2$。

3）如果这两个不同的元素出现在不同的行与列，则 Pr[*zc*=*zd*] =1/4。

因此，无论发生什么情况都有 Pr[*zc*=*zd*]≤1/2。

如果 *D* 和 *C* 不同元素的个数再增多，则 Pr[*zc*=*zd*]将更小。因此，我们有 Pr[*zc*=*zd*] <1/2。这样，我们就获得一个矩阵乘积验证器的蒙特卡罗算法：

```
MATRIX-PRODUCT-VERIFIER(A,B,C)
1. 选择一个随机二进制矢量 z[1..n]，其中 Pr[z_i=1]=½ ， Pr[z_i=0]=½, i=1…n;
2. x=zAB;
3. y=zC;
4. if(x==y)
5.       报告"AB=C";
6. else
7.       报告"AB≠C";
```

很显然，该算法的时间复杂性为平方级，即 $O(n^2)$，这里 n 为矩阵 ***A***、***B***、***C*** 的维数。

根据前面的分析，如果给出的矩阵乘积是正确的，则算法输出“***AB*=*C***”的概率等于 1。如果给出的矩阵乘积是错误的，则算法输出“***AB*≠*C***”的概率小于 1/2。

提高算法结果的准确性

本章前面说过，蒙特卡罗算法的特点是运行的次数越多，正确性越高。因此，如果我们想提高算法 MATRIX-PRODUCT-VERIFIER(*A*,*B*,*C*)的准确性，多运行几次即可。例如，我们想将误判的概率降低到不超过 1/4，怎么办呢？答案很简单：将该算法运转两次，并且只在两次结果都为“***AB*=*C***”的时候才输出“***AB*=*C***”。其他情况都输出“***AB*≠*C***”！

证明　假定算法的两次运算完全独立（即随机矢量的选择相互没有联系），则：

1）如果给出的矩阵乘积是正确的，则两次运算的输出都是“***AB*=*C***”的概率等于 1。

2）如果给出的矩阵乘积是错误的，则两次运算的输出都是“***AB*=*C***”的概率小于 1/4。即 Pr[输出=“***AB*=*C***”] = Pr[第 1 次运算输出=“***AB*=*C***”，第 2 次运算输出=“***AB*=*C***”] = Pr[第 1 次运算输出=“***AB*=*C***”]×Pr[第 2 次运算输出=“***AB*=*C***”]<1/2×1/2=1/4。　□

如果想再进一步提高正确性的概率，我们可以再多运行几遍。例如，运行 3 遍，误判概率将降低到 1/8；运行 4 遍，误判概率降低到 1/16；运行 n 遍，误判概率只有 $1/2^n$。

6.8　随机化最小生成树算法

本书第 5 章讨论过最小生成树问题。效率最高的最小生成树算法是 Prim 和 Kruskal 算法，它们的时间复杂性都是 $O(E\log V)$，从结点的个数来看，都在平方级以上。但由于两种算法都是贪婪策略，其算法效率应该是非常优异的，想进一步改善似乎余地不大。

但真的不能改善了吗？

如果仔细分析一棵最小生成树，我们发现它的成本在于边的选择。而构造本身的成本只

有 $V+E$。如果我们知道哪些边属于一棵最小生成树，构造起来就不费力气了。但是要想知道那些边属于最小生成树，不是需要与其他边进行比对吗？

也许我们并不需要这样做。我们可以用一个随机数来获知一条边是否属于最小生成树。准确地说，我们用抛硬币来决定一条边是否应该被纳入最小生成树里，这样就可以将时间成本降低到线性。这种想法就是随机化线性时间最小生成树算法（A Randomized Linear Time MST Algorithm）。该算法由 David Karger、Philips Klein 和 Robert Tarjan 于 1995 年提出，它可以在 $O(V+E)$的期望时间内完成最小生成树构造。

6.8.1 Karger-Klein-Tarjan 算法

当然，不会因为边的选择是随机的，这条边就一定属于某棵最小生成树。因此，我们在随机挑选边的时候也需要进行某种测试，以衡量此边是否合适。为此，我们需要对边进行某种划分，这种划分就是 F 重边和 F 轻边。下面给出 F 重边和 F 轻边的具体定义。

定义 设 G 为一带权重的图，F 为 G 的一个森林。$F(x, y)$为森林 F 里连接 x 和 y 的路径，$w_F(x, y)$为路径 $F(x, y)$上权重最大的边。如果 x 和 y 没有连通，则 $w_F(x, y)=\infty$。对于任意一条边$\{x, y\}$，如果 $w(x, y) > w_F(x, y)$，则称该边为 F 重边；否则，就称为 F 轻边。

显然，森林 F 里的所有边都是 F 轻边，而 F 重边不可能出现在 G 的最小生成森林里。（为什么？）

取样引理 以概率 p 独立地选择 G 里面的边（即 G 的每条边被选中的概率为 p），由此构成子图 H。又设 F 为 H 的最小生成森林。则 G 里 F 轻边的条数最多为 n/p，这里 n 为图 G 的结点数。

证明 我们通过构建子图 H 及其最小生成森林 F 来进行证明。

假设 H 和 F 最初都为空。按照权重递增次序对图 G 的每一条边 e 进行考虑：

1）如果 e 的两端在 F 的同一个连通分量里，则 e 是 F 重边，因为当前 F 里的所有边都是 F 轻边。

2）用抛硬币的办法来决定是否将 e 加入到 H 里。

3）如果 e 被加入到 H 里，并且是 F 轻边，则将 e 加入到森林 F 里。

不难看出，这样构造的森林 F 与将 Kruskal 算法应用到图 H 上产生的结果一样，因此，F 是图 H 的最小生成森林。另外请注意，一条 F 重边 (F 轻边)在经过上述处理后仍然为 F 重边 (F 轻边)。

现在我们要证明的是图 G 里 F 轻边的数量稀少。

由于一条边是否被加入子图 H 是通过抛硬币决定的，所以一条加入 H 的边只有是 F 轻边的时候才被加入到森林 F，而 F 里的 F 轻边数不会超过 $n-1$。（为什么？）那么我需要抛多少次硬币才能使 F 的边数达到 $n-1$ 呢？设 Y 是需要抛硬币的次数，则 Y 就是图 G 里 F 轻边条数的上限。而 Y 的分布是一个负二项式分布（Negative Binomial Distribution），其参数为 $n-1$ 和 p。根据概率论，随机变量 Y 的数学期望值为$(n-1)/p$。因此，图 G 里 F 轻边的条数最多为 n/p。 □

6.8.2　结点降低算法

在正式介绍线性时间最小生成树算法前，我们还需要介绍结点降低算法。该算法将图里的多个结点用一个结点替换，从而降低图里的总结点数。该结点降低算法来源于捷克数学家奥塔卡·博鲁韦卡（Otakar Borůvka）的最小生成树算法，因此称为 Borůvka 结点降低算法。

Borůvka 结点降低算法描述如下：

1）对于图 G 每个结点 v，选择与 v 相连的边里面权重最小的边。

2）收缩所有选中的边，将由选中的边相连的连通分量用一个结点替换。

3）删除自循环。

4）删除单个独立的结点。

5）如果两个连通分量之间有多条边相连，保留权重最小的边，删除其余的边。

很显然，在经过上述结点降低算法后，一个图的结点数至少减少一半。而且，该结点降低算法的时间复杂性最坏为 $O(m\log n)$，而期望时间成本为线性 $O(m)$。

6.8.3　线性时间最小生成树算法

线性时间最小生成树算法描述如下：

1）对 G 连续应用 Borůvka 结点降低算法两次，获得图 G_1。

2）以 1/2 的概率独立地在图 G_1 里面选择每一条边，得到导出子图 H。

3）对 H 递归进行同样操作（以 1/2 的概率独立地选择每一条边），获得 H 的最小生成森林 F。

4）找出图 G_1 里面所有的 F 重边，并删除。

5）对剩下的图（删除 F 重边后的 G_1）递归使用本算法，获得最小生成森林 F_1。

6）第 1 步收缩的边和 F_1 里面的边一起构成图 G 的最小生成树。

算法正确性证明　第 1 步里面所有被收缩的边属于最小生成树，而最小生成树里剩下的边将构成图 G_1 的最小生成树。而第 4 步里删除的边必不属于任何最小生成树。因此，第 5 步的递归算法调用将正确计算出图 G_1 的最小生成森林。　□

6.8.4　线性时间最小生成树算法的时间成本分析

由于整个算法都是对边进行考虑，所以我们只有弄清楚一共考虑了多少条边才可以获得算法的时间成本。由于算法包括两个递归步骤 2 和 3，因此，只需要对这两个步骤递归完成时总共需要考虑的边数进行计算即可。第 2 步由于每次递归时所处理的子图的边数为原图的一半，因此，总共需要考虑的边的条数$\leqslant \sum_{i=0}^{\infty} m/2^i = 2m$。根据取样引理，第 3 步需要考虑的边的条数为图中结点数的 2 倍。而图中结点数每次递归减少至少 4 倍，因此总结点数$\leqslant$

$\sum_{i=1}^{\infty} 2^{i-1} n/4^i = n/2$。

由此可见，整个算法的运行时间的期望值为 $O(2m+n)$，即线性。

如果读者有兴趣，还可以进一步证明，线性时间最小生成树算法在线性时间内结束的概率为 $1-e^{-\Omega(m)}$。这一点就留给读者作为练习。

6.9 随机数的生成

在本章前面的讨论中经常用到随机数，但我们并没有说明随机数是如何产生的。甚至连什么是随机数也没有进行精确定义。那么随机数到底是什么呢？或者说我们怎么知道变量 x 的取值是一个随机数呢？例如，18 是一个随机数吗？

就一个数的本身来说，世界上没有什么数是随机数。随机数必须相对于一个数列或一组数才有意义。如果一个数与某一组数没有任何关系，则相对于这组数来说，该数就是一个随机数！因此，单独考虑一个数是不是随机数是没有任何意义的。

判断一个数是否是随机数需要有一个数列或一组数作为参照，而判断的标准则是看一个数在一个数列里面的出现是否有迹可寻。如果有迹可寻，则这个数不是随机数；如果无迹可寻，即没有任何规律能够解释为什么这个序列里面有这个数，则这个数就是随机数。

松散地说，随机数就是那种其出现完全是偶然的数，它与前后左右出现的数没有任何关系。而为了能够在算法中使用随机数，我们必须采用某种办法来产生随机数。那么，如何产生随机数呢？

一种办法是使用表格，即用一张表格来存放一些看上去不相干的数作为随机数。但是，也许我们需要的随机数数量众多，而使用表格就限制了这个数量的大小。况且，一旦表格制作完毕，所有表格里面的数就存在一种关系，即表格关系，从而导致随机属性的丧失。因此，我们似乎需要设计一种方法来在需要的时候再产生随机数。或者说，最好的办法似乎是按照某种规律来在需要的时候产生随机数！

但问题是，这样产生的随机数是真的随机数吗？

当然不是。因为产生随机数的函数本身是一个算法的体现，该算法的步骤在编程的时候已经确定，即产生随机数的过程是确定性的。因此，这样产生的随机数不能是真正的随机数。事实上，按照某种规律在需要的时候产生随机数与随机数的定义是完全矛盾的！

因此，使用某种算法产生的随机数被称为**伪随机数**，也就是假的随机数。但读者也不必为这个随机数不是真正的随机数而感到忧虑烦恼。因为，虽然这样产生的不是真正的随机数，但只要你的对手不知道这个伪随机数的产生序列，那么对于他来说，这就是随机的。就像虽然那些相信宿命论的人士认为：你的一生早已经注定，但因为你并不知道，因此对于你来说，你的人生仍然是随机的，也就说，你可以按照自己的意志安排人生！

既然我们只能产生伪随机数，那么我们的随机化算法也只能称做伪随机化算法！

6.10　随机化算法的应用

前面的讨论应该已经让读者相信，随机化算法的用途十分广泛。除了前面提到过的用途外，我们在对数值分析问题的解答进行估计时也经常用到随机化。此外，有的问题虽然有确定性算法，但可能时间成本或复杂性太高，而用随机化算法就可以很简单地解决。有时候，我们并不需要精确的结果，而只需要大概的估算即可。这对于那些牵扯到很多不可知因素的问题来说十分有用。例如，对于一场战争的预估，很多时候我们只需要知道是否能赢，而打赢所需要的具体成本就不是特别关心了。在商业上，进行风险分析也大量地使用随机化算法。

思考题

1. 在矩阵乘积验证器的算法设计中，我们使用的随机矢量 $z[1..n]$为二进制矢量，即该矢量的每个元素的取值只有 0 和 1 两种可能。如果去掉这个限制，例如矢量的元素取值可以是任何实数，那么矩阵乘积验证器算法是否仍然成立？为什么？
2. 假定在打牌的时候，如果你思索了很长时间才出牌，这张牌是否是一张随机的牌？
3. 现代考试常出现的题型是所谓的单选题，即在多个选项里选择一个正确答案（准确地说应该是选择一个最好的答案）。假定某次考试一共有 n 道考题，每道题有 m 个选项，选对一题得 $k(k>0)$分，选错或多选得 0 分。假定你不知道任何一道题的答案，于是你选用随机化策略来进行答题：每次都是从 m 个选项中随机选择一个。请求解此种算法策略获得最后总分数（期望值）。你可以做出合理假设。
4. 重做第 3 道题，只不过此次你采取另外一种策略，即每道题选择相同的选项。
5. 重做第 3 道题，但现在答错一题需要倒扣 k 分（即计$-k$ 分）。
6. 重做第 4 道题，但现在答错一题需要倒扣 k 分（即计$-k$ 分）。
7. 在算法中使用随机化是向命运低头吗？说明你的理由。
8. 古代的卜签占卦在很多时候起的作用是帮助占卦者做出决定，而这个决定就是随机的。这个随机的决定有用吗？请从算法的角度予以说明。
9. 你认为这个世界上存在随机事件吗？请从算法的角度予以阐述。
10. 请设计一个算法来输出一个素数。
11. 分析线性时间最小生成树算法在最坏情况下的时间成本。
12. 证明：线性时间最小生成树算法算法在线性时间结束的概率为 $1-e^{-\Omega(m)}$。

PDG

PART THREE

第三篇　算法分析篇

PDG

CHAPTER 7

第7章 概率分析

1831年6月，年仅22岁的查尔斯·达尔文（Charles Darwin，如图7-1所示）从位于英国剑桥的基督学院（Christ's College）毕业。因为没有找到工作，就赋闲在家，顺便做一些自己感兴趣但却与所学专业不相关的事情。他的家人和朋友看不过去，便为他申请了一个在HMS Beagle船上当随船陪伴的职位。虽然没有薪水，但却解决了食宿问题。

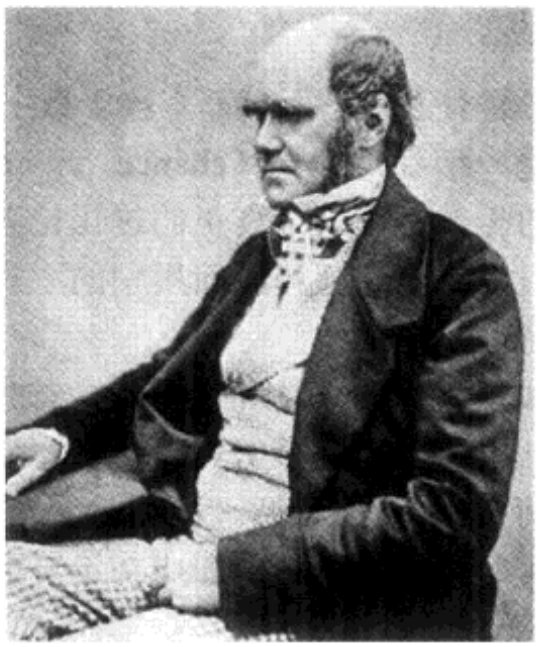


图7-1 正襟危坐的生物进化论鼻祖查尔斯·达尔文

（图片来源：Wikipedia）

同年12月27日，达尔文手拿着一本书登上了HMS Beagle号商船。作为一个神学院毕业的学生，众人以为达尔文手拿的肯定是圣经。但众人的猜测只对了一半：没错，他拿的的确是“圣经”，只不过不是达尔文在神学院学习过的神的话语的那本圣经，而是地质学上的“圣经”：查尔斯·莱尔（Charles Lyell）的《Principles of Geology》（地质学原理）。达尔文靠着这本书打发了他在船上航行时的漫长无聊时光。

莱尔的《Principles of Geology》一书的主要论点是均构学（uniformitarianism）：地球的形成是自然力（如海水的不断升降侵蚀、地壳本身的漂移和升降）在漫长年代里作用的结果，而

且这种推动地质运动的自然力在今天仍然发挥着作用。用英国地质学家詹姆斯·哈特（James Hutton）的话说就是："现在是打开通往过去大门的钥匙（the present is the key to the past）。"达尔文对莱尔的理论非常推崇，并发挥自己的想象力，将莱尔的理论从地质学推广到了生物学。达尔文想到了生物甚至人类自身也可能是通过一个漫长的自然过程演变而来的。

1859 年，达尔文发表了他改变时代的著作：《Origin of Species by the Means of Natural Selection or the Preservation of Favored Races in the Struggle for Life》（基于自然选择的物种起源和在生命挣扎中的优势种族的保存）。在该书中，达尔文正式抛出了生物进化论：人的出现像地壳的变化一样，是自然界在漫长年代里选择的结果，而且这种选择并不具有目的性。也就是说，人的出现完全是一个随机的自然事件，即概率事件。

虽然达尔文心目中的老师查尔斯·莱尔并不赞同人是随机进化出来的（莱尔相信人是被创造的），但进化论却逐渐被越来越多其他的人所接受。终于，在很多人的心目中，"人是神造的"的普世观被"人是随机进化的"的人道主义观所取代。

7.1 一切都在概率中

有人云：一切都在概率中。又有人云：万事都有赖于概率。

将这句话翻译成白话，就是一切都是运气（chance 或 luck）。既然人的出现都是一种偶然，那还能有什么事情是必然的呢？对于一个真正的进化论者来说，当然一切都是或然的！换句话说，所有事件都能够以概率来充分解释，而用别的方式解释都不具备重复性，也就是说，其他解释在科学上都是错误的。

例如，我们如何解释一个成功人士的成功呢（当然，首先假设世界上有所谓"成功"的概念）？按照成功人士自己的话来说，他从小努力、认真，或者是因为他做了甲、乙、丙、丁等事情，所以就成功了，似乎成功有迹可循。但如果另外一个人按照此人的做法再做一次，多半不会成功。至少没有人敢肯定会成功。即成功不可能复制，这就说明了成功的偶然性。

而从概率的角度来看，理解成功就是很简单的事情了。成千上万的人努力做一件事情，按照概率来看，总会有人成功（或者总会有一个做得最好）。就像买彩票，无数人买，而买的人当中总会有一个人中奖。这是概率所昭示的。到底谁中奖当然是随机的，即靠的是运气。但从微观上看，中奖或成功的具体人士就会开始大事吹嘘自己的能耐了。其实，他不知道（也许不想知道或不能知道），这完全是一个概率事件。

这就是一切都在概率中。

既然一切都在概率中，那算法作为万有的一个部分，是否也在概率中呢？或者说，算法与概率有关系吗？答案当然是肯定的。也许有人坚持认为算法是确定性的，但本书第 6 章已经讨论过随机化算法，而随机化算法就涉及了概率。而本章将进一步阐述，对算法进行分析也不时地涉及概率，并且有时候，概率分析是我们分析一个算法的唯一手段！

7.2 什么是概率分析

到目前为止，本书所进行的算法分析大都是所谓的确定性分析，即分析的时候并不涉及概率。这对于最坏和最好情况分析来说，确实如此。但对于我们所讨论过的平均情况分析来说，该论断并不正确，因为平均情况分析已经涉及了概率。只不过这只是简单地对所有的输入分布取平均而已，比较隐蔽，如果粗心就容易忽略。

对于一个细心的读者来说，也许会觉得我们所讨论的平均情况分析过于简单，因为我们将各种可能的输入分布均看做完全等价，即每种输入分布发生的概率相等。但是，如果情况不是这样，（谁能保证是这样呢？）那么我们的平均情况分析还会奏效吗？

更为重要的是，有时候，也许我们并不能将每种输入分布情况都列举出来，分别计算出相应的效率，然后加以平均。这时该怎么办呢？

这时，我们就需要考虑各种情况发生的概率，并根据这些概率的不同而计算出其对整体算法效率的影响，并据此计算出最后的算法效率的期望值。这就是概率分析。

概率分析的一般思路是：首先对输入的分布情况进行某种假设，最常见的假设当然是均匀分布，这也是在没有其他额外信息时的唯一可能的假设；然后以此假设为前提，对一个已知的算法进行此种输入分布下的复杂性和效率分析，如图 7-2 所示。

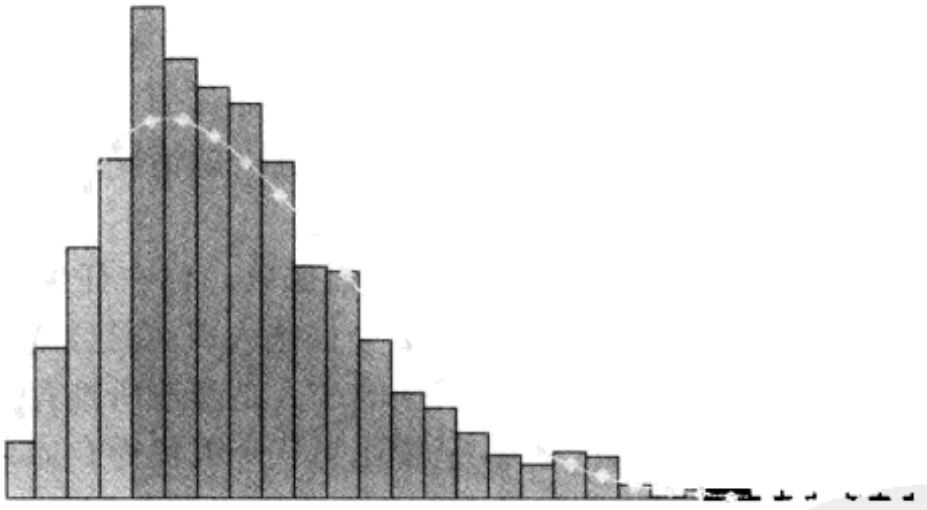


图 7-2 概率分析的前提是对输入分布进行某种概率假设

当然，概率分析也可以用来进行算法正确性和其他标的物的分析。其实，在第 6 章我们已经用概率分析的手段对算法正确性进行了分析。另外，也可以通过概率分析的手段来帮助设计新的有针对性（针对特定输入分布）的算法。只不过这种行为应该被称为**概率设计**（probabilistic design），而不是容易造成误解的“概率分析”。

下面我们就以人人都熟知的寻找情人的问题来阐述概率分析。

7.3 梦幻情人的代价

对于大部分人来说，恋爱是人生几乎不可避免的事情。虽然伤感不断，但却乐此不疲。

“……我痴痴地等待你，等待已融入我心底；我要爱你更多，永远都不止息……”（…I’ll be waiting for you, here inside my heart, I'm the one who wants to love you more…）

这是加拿大名歌手塞琳·迪翁（Celine Dion）的“爱你更多”（To Love You More）歌曲的高潮里面最为煽情的一句。但是，要想“爱你更多”，先得找到一个值得“爱得更多”的人！或者，更理想的是寻找到一个“梦幻情人”（见图 7-3）。

图 7-3 稀松平常的恋爱问题也蕴涵着算法的奥秘

对于一个不善交流或所处环境闭塞的人来说，要找到一个可以“爱你更多”的梦幻情人恐怕并非易事。但也不是无计可施：至少可以通过婚姻介绍所。

对，通过婚姻介绍所应该是一个不错的选择。

假如你通过婚姻介绍所来寻找自己的异性朋友，当你列出了你的条件后，婚姻介绍所就会开始给你介绍候选人。婚姻介绍所通常的做法是定期（如每周）给你介绍一个候选异性朋友（显然，你必须是个异性恋者）的情况。在你看了候选人的情况后决定是否要对其进行面试（即约会）。

当然，不同的人有不同的面试策略。但由于害怕孤单，你在任何时候都需要有一个恋人，如果没有，则你可能会精神崩溃。因此，第 1 个候选人将总是会成为你的恋人。另外，由于你的虚荣心很强，你在任何时候都要保持所谈的恋人是当前所推荐过的所有候选人里最优秀的。当然，我们假设你有能力判断一个人是否比另一个人优秀！如果你不能做出这种判断，或许也不需要学习什么算法了。

因此，当婚姻介绍所送来一份候选人情况介绍后，你有两种选择：

1）如果新候选人比你现在的恋人强，你当然决定与新推荐的候选人见面。不过，你需要与当前的恋人分手。（同时谈两个恋人似乎是不道德的。）

2）如果新候选人不如你当前的恋人，你则继续与当前的恋人约会（即拒绝新候选人）。

由于各种主观因素和客观因素的限制，婚姻介绍所最多将只提供 n 个候选人的情况。也许你的财力只够查看 n 个人的资料；也许婚姻介绍所一共只有 n 个候选人的资料；也许你在看过 n 个人的资料后已不再幻想；也许在提供 n 个人的资料后，婚姻介绍所被你的吹毛求疵搞得精神崩溃，无法继续为你提供服务等。总而言之，你最多只能考察 n 个潜在的恋人。

这样，这种见好就变的恋爱算法可以由下述伪代码表示：

```
LOVE-FINDER (n)
```

```
best=0                    //为剔出对边界条件的判断，构造虚构的 0 号恋人
for(i=1; i<=n; i++)
    if (候选人 i 比当前恋人更优秀)
    {
        best=i            //用候选人 i 取代当前的恋人
        约会候选人 i
    }
```

当然，谈恋爱是有成本的（如图 7-4 所示）。婚姻介绍所每次介绍一个候选人都会向你收一笔介绍费 c_i。如果你决定与新候选人相处，则需要花费相处费 c_d（用于吃饭、送花、看演出等支出和各种感情的付出）和支付婚姻介绍所介绍成功费 c_j。另外，由于喜欢上新候选人需要与当前的恋人分手，需要和平分手费 c_s（你是一个很善良的人，与人分手你觉得对不起她，因此，主动付出一笔成本）。

图 7-4 恋爱当然是有成本的，如何才能将成本降低呢

现在我们要问的问题是，上述寻找情人的算法 LOVE-FINDER 需要支付多大的成本？

7.3.1 直接分析

如果 n 个候选人中，我们谈了 m 个朋友，则成本为：

$$O(nc_i+mc_d+mc_j+(m-1)c_s)$$

这里 nc_i 为获取 n 个候选人的资料需要支付的成本，mc_d 则是谈 m 个恋人的成本，mc_j 则是支付给婚姻介绍所的 m 次成功介绍费，$(m-1)c_s$ 则是与 $m-1$ 个恋人分手的成本。由于不管最后到底谈多少次恋爱，获取资料的成本总是不变，即 nc_i 是一个恒定的成本，因此，我们只需要考虑真正谈恋爱与分手的成本，即 $mc_d+mc_j+(m-1)\,c_s$。而

$$mc_d+mc_j+(m-1)\,c_s \leqslant mc_d+mc_j+mc_s=m(c_d+c_j+c_s)$$

为分析简便起见，令 $c=c_d+c_j+c_s$，因此，分析的成本就是 mc。那么这个成本是多少呢？

很显然，mc 随着算法的每次运行会有不同，它依赖于我们获取候选人资料的顺序！

7.3.2 最坏情况分析

显然，最坏情况是你与这 n 个候选人都谈了一次恋爱。而这种情况发生的唯一可能是你所看到的候选人一个比一个优秀，也就是婚姻介绍所将候选人按照从差到好的顺序发送给你。（这种顺序婚姻介绍所将获得最大的收益，何乐不为呢？）在这种情况下，谈恋爱的成本为：

$$O(nc_i+nc) = O(nc)$$

很显然，成功介绍费、谈恋爱的成本和分手的成本之和一定远远大于看一份候选人资料的成本，即 $c >> c_i$。

7.3.3 最好情况分析

你只谈了一次恋爱。这种情况只有在所看到的候选人一个比一个差，也就是婚姻介绍所将候选人按照从好到差的顺序发送给你。（这种顺序婚姻介绍所将获得最小的收益，似乎不太可能呢？）在这种情况下，谈恋爱的成本为：

$$O(nc_i+1c) = O(c)$$

该等式的成立有赖于我们的假设：$c > nc_i$，因为看再多的资料也不如真正谈一次恋爱的成本高，即使只计算感情这一样，其付出的代价就可能超过看 n 份资料所需要付出的代价（不过一点时间成本而已）。

7.3.4 平均情况分析

到目前为止，最坏和最好情况的分析似乎非常简单。但简单的东西通常不太常见。因此，平均情况分析才最能反映现实。而在平均情况下，你不太可能谈 n 次恋爱，也不太可能只谈一次恋爱，最有可能的是，你谈 $n/2$ 次恋爱。这种情况可以发生在多种顺序下。此时，谈恋爱的成本为：

$$O(nc_i+n/2c) = O(nc)$$

问题是，这种平均情况分析正确吗？或者说，平均情况就是我们所期望发生的情况吗？

乍看之下，一切似乎合理，最高最低取平均不就是平均情况吗？如果如此，你能够说出来什么情况下，平均情况会发生吗？

一个显然的答案是前面 $n/2+1$ 个候选人一个比一个好，而后面的 $n/2-1$ 个人都不如第 $n/2$ 个候选人。另外一种情况是前面 $n/2+1$ 个候选人一个比一个差，而后面的 $n/2-1$ 个人都比第 1 个候选人好。也许你会觉得只要在候选人序列里面有一个包含 $n/2$ 个人的资格递增子序列即可。而这种情况的出现有 1/2 的概率，因此，我们的平均分析正确。

但遗憾的是，这种观察是错误的。要做出精确的判断，我们将不得不使用概率。

7.3.5 平均情况下成本的概率分析

为方便起见，我们赋予每个候选人一个排名，假如 rank(i)是第 i 个候选人的排名，即该候选人的优秀程度名列第 rank(i)名。显然，rank(i)取值在 $1 \sim n$ 之间。则〈rank(1), rank(2),…,rank(n)〉给出的将是数列〈1,2,…,n〉的一个排列。

假设候选人出现的顺序是随机的（这是概率分析所需要的概率假设），设 X_i 代表候选人 i 被你看中的事件，即你将用候选人 i 替换当前的恋人。则表达式 $X = X_1+X_2+\cdots+X_n$ 表示的是你谈恋爱的次数。这时我们的任务就变成计算随机变量 X 的期望值 $E[X]$。因为有了这个期望值，我们就可以直接计算出 LOVE-FINDER 算法的恋爱成本了。

而计算 $E[X]$需要我们计算 $X_1+X_2+\cdots+X_n$ 的数学期望值，而计算该表达式的数学期望值需要知道第 i 个候选人被看中的事件的发生概率 Pr{候选人 i 被看中}，那么这个概率是多少呢？

要回答这个问题，我们需要知道候选人 i 被你看上的条件是什么。根据题意，候选人 i 被看上的条件是候选人 i 比前面所有的候选人都强。（看出来了吗？）

由于我们假设候选人出现的顺序是随机的，因此，前面 i 个候选人出现的顺序也是随机的。即该 i 个人里面的任何一个人都可能是该 i 个人里面最优秀的，并且任意一个人是最优秀的概率完全相等。因此，我们有 Pr{候选人 i 被看中}=1/i，而这就意味着 $E[X_i]=1/i$。

而有了上述结果，我们就可以计算 $E[X]$:

$$E[X]=E\left[\sum_{i=1}^{n}X_i\right]=\sum_{i=1}^{n}E[X_i]=\sum_{i=1}^{n}\frac{1}{i}=\ln n+O(1)$$

因此，你谈恋爱的次数为 ln$n$ 次（不是 n/2 次），而整个寻找情人的期望成本为 $O(c\ln n)$，而这与平均情况 $O(nc)$相差甚大！看来我们的平均情况分析大大高估了谈恋爱的成本。

由此可见，概率分析与平均情况分析是完全不同的。

7.3.6 概率分析结果的有效性

概率分析的结果表明，谈恋爱的成本并没有我们随便一算的那么高。事实上，它比平均情况下的成本低很多。（看来可以大大地多谈恋爱了！）但是，期望值就是我们在实际中付出的真正成本吗？或者说，我们计算出来的恋爱成本与实际情况相符合吗？答案是不一定。

原因是我们计算期望值的前提是假设候选人的出现顺序是完全随机的！而根据本书第6章的讨论，完全随机的东西纯属子虚乌有。也就是说，我们上面的概率分析结果也许并不可靠。而对于算法分析这门学问来说，这是不能容忍的。

而且，更为重要的是，商人是唯利是图的，婚姻介绍所是不太可能将候选人按随机顺序给你。为了利润最大化，婚姻介绍所很有可能将名单按递增顺序发给你。这样我们的平均情况分析和期望值分析都完全失去了意义。等待的将是最坏情况。所谓“无商不奸”。

但我们有什么应对措施吗？

显然，我们应对的方法无非是三种。第一种方法是搞清楚候选人出现的顺序到底是按什么模式分布的，然后按照这种模式重新计算成本。但此种方法的难度甚高，无法应用到实际中。再说，此种手段也无法应付婚姻介绍所采取的利益最大化手腕。

第二种方法则做好最坏的打算，不管是运气差还是奸商故意作难，只要有将冤大头做到底的决心，还有什么可怕的呢？不过，有此种决心的人似乎并不多见。

那就只剩下最后一种方法：想办法使我们的概率分析正确！而这需要我们的方法既能够摆平概率非均匀分布的自然对手，也能摆平婚姻介绍所采取的利益最大化手腕。

我想读者应该想到了这种方法：随机化！

我们的概率分析基于候选人出现顺序的完全随机化。虽然没有人能知道或肯定候选人出

现的顺序是随机的，但我们却可以对候选人名单进行一个随机重排，从而获得随机分布，使得我们的假设成立，从而保证概率分析的结果正确。而且，随机化方法也能够击败婚姻介绍所采取的任何利益最大化计谋。

7.3.7 正确概率分析的保障

由上述分析可知，随机化是确保概率分析结果能够挺立的强健手段。从哲学角度上看，这种随机化实际上就是一种对事情的控制。虽然我们不知道名单出现的顺序，但我们可以通过随机化确保其随机和均匀。当你不能以最有利于自己的方式安排事情时，你应该做的就是随机化一切，使得所有的机会均等。在英语里称之为“level the playing field”，就是中文所说的“费厄泼赖”。如果此时再出现很坏或最坏的结果，那就只能认命了。但令人宽慰的是，这个坏结果不是你的仇人或对手加在你头上的！

加以随机化后，寻找情人的算法修改如下：

1）婚姻介绍所事先发送 n 个候选人的名单给我们，但无需提供任何候选人的资料。

2）每周我们在名单中随机抽选一个，要求婚介所发送其详细资料。

3）然后，按照前面的 LOVER-FINDER 算法继续。

当然，在随机抽取名单的时候，我们只选择那些尚未查阅过详细资料的候选人。通过这种处理，我们就将假设候选人随机出现变成事实上的随机出现。下面是我们的随机化寻找情人算法：

```
RANDOMIZED-LOVER-FINDER (n)
    将候选人名单进行一个随机排列
    LOVER-FINDER (n)
```

由概率分析可知，上述算法的成本复杂性为 $O(c\ln n)$，一个非常低廉的成本。

7.4 梦幻情人的概率

我们的情人寻找算法 LOVER-FINDER 能够让我们每次见到更优秀的异性时立即见异思迁，大大过了一把“良禽择木而栖，好男（女）择女（男）而爱”的瘾。而且更为重要的是，这样一种奢华的恋爱算法居然成本低廉，仅仅是对数级的 $O(c\ln n)$。

一切似乎完美得不能再完美。

但仔细一想，发现这个算法还是有问题。虽然对数级成本在数量级上较低，但这毕竟是多次谈恋爱。而谈多次恋爱的成本毕竟是不少人在精神上、时间上和经济上负担不起的。如果你恰恰又是一个清教徒，则谈多次恋爱还有可能违背你的信仰原则。（谁能保证谈恋爱的时候不会出差池呢？）因此，你决定只能谈一次恋爱！

虽然你是清教徒，或者没有精神和精力去谈多次恋爱，但找到最好的伴侣却是人人都向往的！因此，一个自然的问题是：如果只谈一次恋爱，是否仍能找到称心如意的恋人呢？

那就得看你有没有这个本事啦。不过，这里的本事指的不是忽悠异性的本事，而是算法设计与分析的本事！

显而易见，既然只能谈一次恋爱，但又必须找到最称心如意的梦幻情人，我们便不得不在查看多个候选人的情况下选择一个人作为恋人。为帮助你达到此目标，婚姻介绍所允许你对候选人进行面谈（是见面，或者说相亲，不是真正的谈恋爱）。每次面谈完你必须马上决定是否与该候选人确定恋爱关系。如果决定谈下去，婚姻介绍所即完成使命，将不再提供任何候选人。而如果你不喜欢当前的候选人，则可要求婚姻介绍所换一个人。当然，婚姻介绍所最多提供 n 个候选人。

面谈虽然不如真正谈恋爱那么伤感情和花费金钱，但也是花费时间和精力的。由于时间和精力有限，你希望将面谈的人数限制在一个较小的范围。因此，你决定先面试 k 个人，并通通予以拒绝，然后在后面面试中出现的第 1 个超出前面这 k 个人的人就是你的恋人。如果后面再无比前面 k 个人都好的人，则第 n 个人就是你的恋人。即最坏情况下，你需要面试所有 n 个候选人。因此，我们新的更加节省成本的情人寻找算法如下：

OPT-LOVE-FINDER (n)

1）选择一个 $k<n$。

2）面试婚姻介绍所送来的前 k 个候选人，并予以拒绝。

3）后面出现的第 1 个超出前面这 k 个候选人的人就是恋人。

4）如果后面不出现超出前面 k 个人的人，则第 n 个人就是你的恋人。

这里我们要问的问题就是，应该如何确定 k 的取值，以使得你的恋人是所有 n 个候选人里面最好的概率最大！并且计算出这个概率！

对于很多人来说，这个问题乍看似乎无从下手。

但仔细分析题意可以看出（也许不容易看出），解答这个问题的步骤应该是先假设我们已经找到了某个 k，然后计算在该选定 k 的取值下，我们获得最佳候选人的概率是多少。然后再根据计算出的概率表达式求极大值即可算出这个 k 的取值。

现在，假定我们已确定了 k 值的大小。很显然，在 OPT-LOVE-FINDER 算法的策略下，我们找到梦幻情人（最佳情人）的唯一可能是最好的人选出现在后面的 $n-k$ 个人当中。如果最佳候选人出现的位置是 i，则其前面 $i-1$ 个人里面的最优秀的人必须出现在前 k 个人里面。否则，该最优候选人将成为你的恋人，而不是整个 n 个候选人里面最好的人！

设 $E(i)$ 为候选人 i 是所有 n 个人里面最佳候选人的事件，$E(k,i)$ 代表前 i 个人里面的最佳候选人出现在前 k 个人里面的事件，E 代表你的恋人是整个 n 个人里面最佳候选人的事件。

根据上述分析，有：

$$\Pr[E]=\Pr[E(k+1)E(k,k)]+\Pr[E(k+2)E(k,k+1)]+\cdots+\Pr[E(n)E(k,n-1)]$$

$$=\sum_{i=k+1}^{n}\Pr\{E(i)E(k,i-1)\}$$

这里，$\Pr[E(k+1)E(k,\ k)]$代表的是第 $k+1$ 号候选人是最佳人选，并且前 k 个人里面的局部最佳人选出现在前 k 个人里的事件的发生概率；$\Pr[E(k+2)E(k,\ k+1)]$代表的是第 $k+2$ 号候选人是最

佳人选，并且前 k+1 个人里面的局部最佳人选出现在前 k 个人里的事件的发生概率；Pr[$E(n)E(k,n-1)$]代表的是第 n 号人选是最佳人选，并且前 n–1 个人里面的局部最佳人选出现在前 k 个人里的事件的发生概率。

由于任何一个候选人是全局最佳人选的概率均等（如果不是这样，我们可以通过随机化手段使得该条件满足），则有：Pr{$E(i)$}=1/n，且 Pr{$E(k,i)$}=k/I。因此，

$$\Pr\{E\}=\sum_{i=k+1}^{n}\Pr\{E(i)E(k,i-1)\}=\sum_{i=k+1}^{n}\Pr\{E(i)\}\Pr\{E(k,i-1)\}$$

$$=\sum_{i=k+1}^{n}\frac{1}{n}\cdot\frac{k}{i-1}=\frac{k}{n}=\sum_{i=k+1}^{n}\frac{1}{i-1}=\frac{k}{n}\sum_{i=k}^{n-1}\frac{1}{i}$$

$$=\frac{k}{n}\left(\sum_{i=1}^{n-1}\frac{1}{i}-\sum_{i=1}^{k-1}\frac{1}{i}\right)\approx\frac{k}{n}(\ln(n-1)-\ln(k-1))$$

$$=\frac{k}{n}\left(\ln\frac{n-1}{k-1}\right)\approx\frac{k}{n}\ln\frac{n}{k}$$

这样，在 k 值给定的情况下，获得最佳人选的概率为(k/n)lg(n/k)。对此表达式求取极大值，得到 Pr{E}在 k=n/e 的时候，获得最大值，并且这个最大值为 1/e。

该算法描述如下：

1）对所有的候选人进行随机排列。

2）面试前 n/e 个候选人，并予以拒绝。

3）后面出现的第 1 个超出前面这 n/e 个候选人的人就是恋人。

4）如果后面不出现超出前面 n/e 个人的人，则第 n 个人就是你的恋人。

根据上面的分析，新的情人寻找算法寻找到梦幻情人的概率为 1/e，即约 36%。但该算法的恋爱成本为常数，即 c。而面谈的成本 $c_i n$/e，只有第 1 个情人寻找的约 1/3。

7.5 随机排列问题

前面讨论梦幻情人问题时大量使用了概率分析。为了保证概率分析结果的有效性，我们在考察候选人之前对候选人名单进行了随机化，即将名单重新排列了一次，以使名单的顺序完全随机。细心的读者可能已经发现，我们并没有说明如何进行这种完全随机的排列。

显然，能否有效地进行真正的随机排列就成了情人寻找问题的破与立的分界点。

我们能否有效地进行随机排列呢？答案当然是肯定的，否则我们还啰唆什么？

但是，如何进行随机排列呢？

这个问题对于我们来说很简单：我们可以用闭着眼睛用笔在纸上随便一画，点到谁就是谁来。但对于计算机来说，这似乎不是个办法：闭着眼睛乱画对于计算机来说意味着什么呢？或者说计算机如何实现闭着眼睛乱画呢？事实上，计算机不可能闭着眼睛乱画！

如果仔细分析我们在梦幻情人问题上的随机化策略，我们会发现，对人员名单进行一个

随机排列实际上等价于随机叫号：我们要求婚姻介绍所给每个候选人编一个号（婚姻介绍所为了自己管理方便，也会给所有参与人一个编号的），我们每天随机叫号即可。例如，我们今天叫 15 号，明天叫 58 号，后天叫 3 号等。只要每天叫的这个号是随机的，就可以保证随机化。

而随机叫号对于计算机来说实在是太简单了：计算机的随机数产生器不就是一个随机叫号器吗。由此可见，这个随机数产生器也可以被用来产生一个随机排列。例如，如果下次随机产生的数是 5，我们就选第 5 位候选人；如果产生的是 15，则选择第 15 位候选人，这与福利或体育彩票摇奖类似。基于此种思路，我们可以立即获得一个随机排列产生算法：

```
RANDOM-PERMUTATION(A, n)    // A[1..n]是一个包含 n 个元素的数组
for (i=1; i<=n; i++)
{
    j=RANDOM(j, n);          // RANDOM(i, n)产生一个介于 i 和 n 之间的随机数
    temp=A[i];
    A[i]=A[j]
    A[j]=temp;
}
```

该算法的思想非常简单：在第 i 次循环，从数组 $A[i..n]$ 里随机选取一个元素作为第 i 个位置上的元素，而原来处于第 i 个位置上的元素则被调换到刚才选出的元素所在位置。由于循环变量单向递增的特性，一个元素一旦选出放入位置 i，它所在的位置将不再变化。这样在算法循环 n 次后，所有 n 个位置上的元素都被重新安排了。也就是说，该算法产生了一个新的排列。

问题是，通过随机数来对序列的次序重新排定是否能够产生一个随机排列呢？

要回答这个问题，首先得知道一个随机排列意味着什么，或者说什么是一个随机排列。很显然，针对某个特定的排列，没有什么随机不随机的。一个排列只有放在一大堆排列中看，才有随机不随机的概念：随机即意味着一个排列的产生不以别的排列的产生为转移。也就是一个排列的产生相对于其他排列来说呈现出随机性。从概率上看，随机就是每种可能的排列都以相同的概率出现。因此，判断一个算法能否产生随机排列就是要判断在该算法下，每种排列出现的概率是否相同。RANDOMIZE-PERMUTATION 算法是否满足这个条件呢？

看上去如像满足，但需要严格的证明我们才放心。

由于 n 个元素一共有 $n!$种排列，因此我们要证明的是任意一个排列出现的概率为 $1/n!$。

如果该算法确实产生的是均匀随机排列，则在每次循环后，例如第 k 次循环后，前面 k 个已经排定位置的元素应该形成整个数组排列的一个 k-排列。这里一个 k-排列是一个包含 n 个元素里面 k 个元素的排列。这种排列的个数有多少呢？答案是 $n!/(n-k)!$。（读者看出来了吧？）

如果算法产生的是均匀随机排列，则截至每次循环前，所产生的 k 排列也应该是均匀随机的。也就是说，在第 k 次循环后，对于任意一个 k 个元素的排列，子数组 $A[1..k]$包含该排列的概率为$(n-k)!/n!$，即 k-排列个数 $n!/(n-k)!$的倒数。

事实是否如此呢？我们当然希望如此，否则本章前面的分析不都打了水漂？而证明我们猜想的最好办法自然是数学归纳法（因为牵扯到循环次数）。

下面我们就以 i 作为循环变量，用数学归纳法来证明我们的猜想。

引理 *在第 i 次循环后，前面 i 个元素应该是整个数组排列的一个随机 k-排列。*

初始情况：i=0，即还没有开始任何循环的时候。显然对于任意的 0-排列，子数组 $A[1..0]$ 包含该 0-排列的概率为$(n-0)!/n!=1$。这与实际情况相符：$A[1..0]$是一个空子数组，而0-排列没有元素，空数组里面的排列当然是一个 0-排列。因此，概率为 1。

归纳假设：假设在 $i=k$ 次循环前，每个$(k-1)$-排列出现在子数组 $A[1..k-1]$里的概率为$(n-k+1)!/n!$，则在 $i=k$ 次循环后，我们需要证明每个 k-排列出现在子数组 $A[1..k]$里的概率为$(n-k)!/n!$。

下面我们来考虑一个特定的 k 排列：$\langle x_1, x_2,\cdots, x_k\rangle$。很显然，该排列为一个$(k-1)$-排列$\langle x_1, x_2,\cdots, x_{k-1}\rangle$ 后面紧跟元素 x_k。设 E_1 表示我们的算法在子数组 $A[1..i-1]$里形成一个$(i-1)$-排列，则根据归纳假设，有：$\Pr\{E_1\}=(n-i+1)!/n!$。

又设 E_2 表示第 k 次循环将元素 x_k 放入 $A[k]$里，则当且仅当 E_1 和 E_2 同时发生时，我们在子数组 $A[1..k]$里获得一个 k-排列 $\langle x_1, x_2,\cdots, x_k\rangle$，即算法产生 k-排列$<x_1, x_2,\cdots, x_k>$的概率为：

$$\Pr\{E_1 \cap E_2\}$$

由于 $\Pr\{E_1 \cap E_2\}=\Pr\{E_2|E_1\}Pr\{E_1\}$，并且 x_k 是随机从 $n-k+1$ 个可能性里面选取的一个元素。因此，

$$\Pr\{E_2|E_1\}=1/(n-k+1)$$

综上所述，有：

$$\Pr\{E_1 \cap E_2\}=\Pr\{E_2|E_1\}\ \Pr\{E_1\}=[1/(n-k+1)][(n-k+1)!/n!]=(n-k)!/n!$$

因此，引理成立。 □

将 $i=n$ 代入引理中，有：

$$(n-i)!/n!=(n-n)!/n!=1/n!$$

即 RANDOMIZE-PERMUTATION 算法确实产生的是 n 个元素的一个随机排列。

很显然，该算法的时间复杂性是 $O(n)$，因为每次循环的成本为 $O(1)$，而一共有 n 次循环。至此，我们的梦幻情人的概率分析应该可以立住了吧？

7.6 跳转表问题

在数据结构书籍和课程中，有时会讨论一种所谓的跳转表的结构。这种结构是在链表结构上做出的一种改进结构，由多个连通的链表构成，它允许我们在查找过程中跳过部分元素，从而大大提高链表的查找速度。跳转表的一般形式如图 7-5 所示（本书只讨论跳转表的操作效率分析，如果读者对跳转表的实现有兴趣，请参阅我的《数据结构之

弦》一书）。

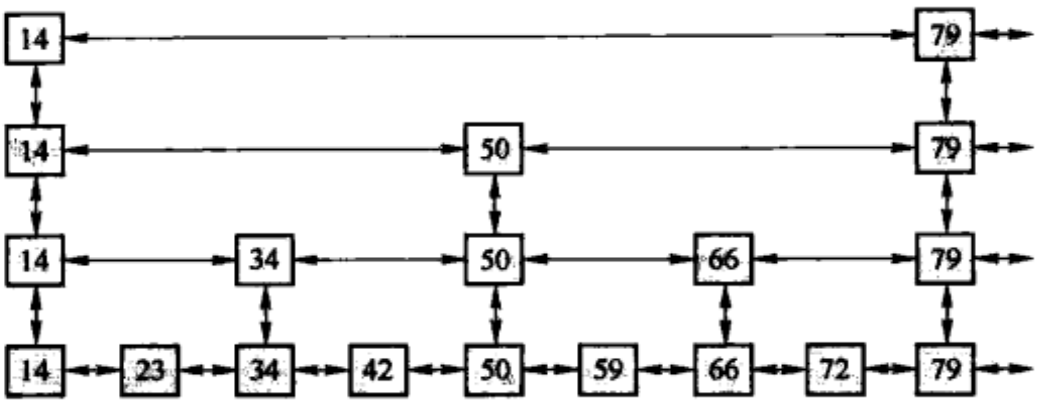


图 7-5 理想的跳转表

我们在图 7-5 所示的跳转表里查找元素 72 的过程如下：

1）首先在最上面一层的跳转表查找（由左往右），发现需要在元素 79 的地方转到下一条链表。

2）在第 2 条链表（第 2 层）往左推进到元素 50 的时候，转入下一条链表。

3）在第 3 条链表（第 1 层）往右推进到元素 79 时，转入下一条链表。

4）在最下面一层的链表（第 0 层）里往左推进到元素 72 结束，查找成功。

查找元素 72 一共进行的比较次数为 6 次。如果使用传统的链表结构，则比较次数为 8 次，跳转表的效率提高了 25%。

也许读者觉得 25%的效率提升不值得我们花这么大的力气来构造看上去非常复杂的跳转表，但图 7-5 所示的跳转表并不是我们所能构造的最好的跳转表。最优跳转表的下面一层的链表长度为其上面一层的链表长度的平方，如图 7-6 所示。

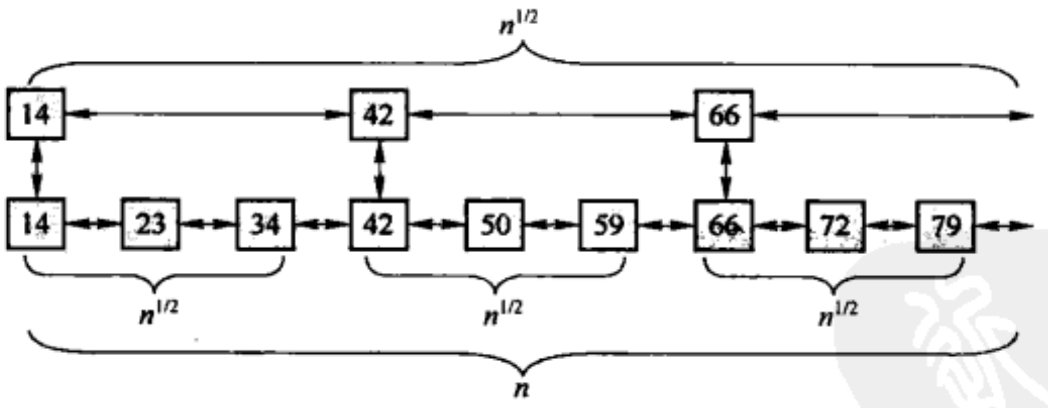


图 7-6 最优的跳转表构造

在最优的跳转表结构下，搜索成本依据链表个数的不同分别为：

- 2 个有序链表：搜索成本为 $2\sqrt{n}$ 。
- 3 个有序链表，搜索成本降为 $3n^{1/3}$。
- k 个有序链表，搜索成本为 $kn^{1/k}$。
- $\log n$ 个有序链表，搜索成本为 $\log n \times n^{1/\log n} = 2\log n$。

由此可见，当链表个数达到 $\log n$ 时，搜索成本已经降低到对数级，与有序线性表的折

半搜索效率相同。这是一个比较令人满意的搜索效率。

7.6.1 跳转表插入操作

跳转表虽然将链表的查找时间从线性级提升到对数级，但却需要在别的地方付出代价。这个地方就是插入操作。因为在有多条链表的情况下，插入一个元素可能需要同时修改多条链表！那么插入操作的代价到底会增加多少呢？

显然，增加的程度依赖于到底有多少链表需要修改。对于任意元素 x 来说，将其插入到跳转表里意味着它首先将被插入到最底层的链表里，因为最底层的链表总是包括所有的元素。但除了最底层的链表外，x 还应该被插入到哪些链表呢？

最直观但却很复杂的做法是：在最底层链表更新后，按照最优构造办法（如图 7-6 所示）依次更新上面的所有链表。不过，这样就意味着上面的每层链表都要更新，有的链表的绝大部分元素都需要被替换。这样的成本显然太高。有没有更好的办法呢？

当然有，办法就是随机化！我们的思想是在将元素 x 插入到最底层链表后，通过抛硬币的方式来决定是否将其插入到上面一层的链表。如果光面朝上，我们将其插入到上面一层，然后再抛硬币决定是否在插入到更上面一层；如果在某次抛硬币时麻面朝上，则插入过程结束。

在这种随机化算法下，元素 x 被插入到每层链表的概率如下：

- 插入到第 0 层（最底层）的概率：1。
- 插入到第 1 层的概率：1/2。
- 插入到第 2 层的概率：1/4。
- 插入到第 3 层的概率：1/8。
- ……

这种情况下，插入操作的成本毫无疑问地下降了。但是，下降了多少呢？这样构造的跳转表是否还是最优的结构呢？

7.6.2 随机化跳转表构建算法

我们先来看一下使用这种插入算法构建的跳转表是否平衡。为了简化讨论，我们增加一个$-\infty$值作为每个链表的第 1 个元素，如图 7-7 所示。

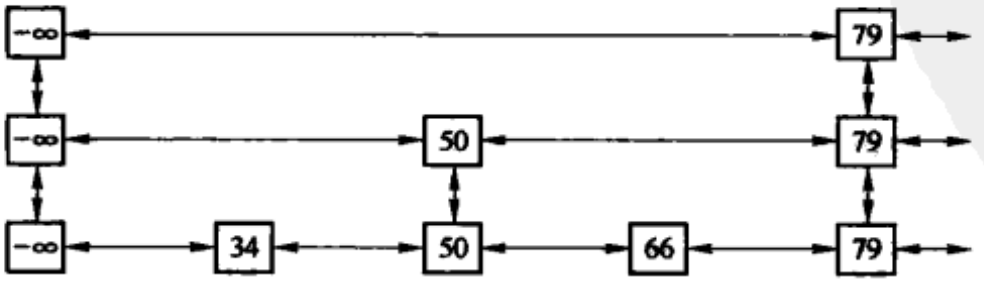


图 7-7 增加$-\infty$值后的跳转表结构

我们通过在一个空跳转表（仅包含元素$-\infty$的跳转表）里不断插入元素来构造跳转表。

而插入的策略就是 7.6.1 节给出的随机插入。我们的问题是：这样构建的跳转表有多好？

从直觉上来说，在这种插入构建方式下，有：

- 所有的元素都插入到第 0 层。
- 1/2 的元素会被插入到第 1 层。
- 1/4 的元素会被插入到第 2 层。
- 1/8 的元素会被插入到第 3 层。
- ……

这样的跳转表看上去似乎比较平衡。但结果真是如此吗？

定理 *在通过随机插入构建的跳转表中，进行查找的操作成本有很高的概率，为* $O(\log n)$。

什么是很高的概率呢？通俗地讲，如果对于任意 $\alpha \geqslant 1$，都可以找出一个合适的常数因子，使得事件 E 发生的概率最低达到 $1-O(1/n^{\alpha})$，则事件 E 称为高概率事件，或者事件 E 发生的概率很高。并且 $O(\log n)$里的常数因子仅依赖于 α。定义如下：

定义 *对于任意* $\alpha \geqslant 1$，*都可以找出合适的常数，使得* E_{α} *发生的概率最低达到* $1-c_{\alpha}/n^{\alpha}$，*则事件* E_{α} *称为高概率事件。*

我们的想法是，通过将 α 设置得很大，从而使得事件 E 不发生的概率 $O(1/n^{\alpha})$达到很小的值。例如，如果将 α 设置为 100，则 $O(1/n^{\alpha})$的值将变得几乎可以忽略不计。

下面就来证明该定理。

根据布尔不等式，对于任意随机事件序列 $\langle E_1, E_2, \cdots, E_k \rangle$，有：

$$\Pr\{E_1 \cup E_2 \cup \ldots \cup E_k\} \leqslant \Pr\{E_1\} + \Pr\{E_2\} + \cdots + \Pr\{E_k\}$$

如果 $k = n^{O(1)}$，且每个事件 E_i 都以高概率发生，则事件 $E_1 \cap E_2 \cap \cdots \cap E_k$ 也将以高概率发生。

引理 *经随机插入构建的包含* n *个元素的跳转表的链表条数为* $O(\log n)$*是一个高概率事件。*

证明 跳转表里的链表条数多于 $c\log n$ 的概率为：

$\Pr\{$多于 $c\log n$ 条链表$\}$

$\leqslant n\Pr\{$元素 x 被插入至少到 $c\log n$ 个链表里$\}$ //布尔不等式

$= n\Pr\{$元素 x 被插入至少 $c\log n$ 次$\}$

$= n(1/2^{c\log n})$

$= n(1/n^{c})$

$= 1/n^{c-1}$

即跳转表里的链表条数多于 $c\log n$ 的概率$\leqslant 1/n^{c-1}$。而这个概率最大为 $1/n^{\alpha}$，这里 $\alpha = c-1$。通过选择 $O(\log n)$限界里的常数 c 可以迫使 α 为一任意大的值。 □

定理 *随机插入构建的包含* n *个元素的跳转表的查找操作成本为* $O(\log n)$*是一个高概率事件。*

为了利于分析，分析将与查找的顺序相反。查找是由左往右，由上往下；分析就从右到左，从下往上。在反向分析查找过程中，访问每一个结点后有两种可能：

- 如果该结点没有被提升过（即在插入该结点后抛硬币为麻面），则往左（即查找到该结点时是从左面过来的）。
- 如果该结点被提升过（即在插入该结点后抛硬币为光面），则往上（即查找到该结点

时是从上面下来的）。

反向查找过程一直持续到根或者$-\infty$。反向分析时，需要进行的往上和往左的次数就是正常查找过程进行比较的次数。

往上的次数<跳转表的级数$\leqslant c\log n$，根据引理，该不等式成立的概率很高（高概率事件）。这意味着，查找需要的步骤数最多为获得 $c\log n$ 个光面所需要抛硬币的次数。（为什么往上的次数能够推出查找所需要的步骤数呢？）

需要进行多少次抛硬币才能获得 $c\log n$ 个光面呢？很显然，抛硬币的次数$=\Omega(\log n)$，即至少需要抛硬币 $c\log n$ 次。但最多需要抛多少次呢？我们声称，为获得 $c\log n$ 个光面所需要进行的抛硬币的次数$=\Theta(\log n)$的概率很高。下面举例说明抛硬币的次数$=O(\log n)$。

假定我们抛硬币 10 倍 $c\log n$ 次，那么什么时候会获得至少 $c\log n$ 个光面呢？

$$\Pr\{\text{刚好}c\log n\text{个光面}\}=\binom{10c\log n}{c\log n}\cdot\left(\frac{1}{2}\right)^{c\log n}\cdot\left(\frac{1}{2}\right)^{9c\log n}$$

$$\Pr\{\text{最多 }c\log n\text{个光面}\}\leqslant\binom{10c\log n}{c\log n}\cdot\left(\frac{1}{2}\right)^{9c\log n}$$

由于，$\binom{y}{x}\leqslant\left(\mathrm{e}\frac{y}{x}\right)^{x}$

有：

$$\Pr\{\text{最多 }c\log n\text{ 个光面}\}\leqslant\binom{10c\log n}{c\log n}\cdot\left(\frac{1}{2}\right)^{9c\log n}$$

$$\leqslant\left(\mathrm{e}\frac{10c\log n}{c\log n}\right)^{c\log n}\cdot\left(\frac{1}{2}\right)^{9c\log n}=(10\mathrm{e})^{c\log n}\cdot 2^{-9c\log n}$$

$$=2^{\log(10\mathrm{e})c\log n}\cdot 2^{-9c\log n}=2^{[\log(10\mathrm{e})-9]c\log n}=1/n^{\alpha}$$

这里即 $\Pr\{\text{最多 }c\log n\text{ 个光面}\}\leqslant 1/n^{\alpha}$，这里 $\alpha=[9-\log(10\mathrm{e})]c$。

如果抛硬币的次数不是 10 倍 $c\log n$ 次，而是∞倍 $c\log n$ 次，则对于任何常数 c 来说，$\alpha\to\infty$。因此，对于任何常数 c，都可以求得一个抛硬币的 $c\log n$ 级倍数，使得 α 足够大。□

由此可见，随机插入构建的包含 n 个元素的跳转表的查找操作成本有很高的概率，为线性对数级。也就是说这样构建的跳转表确实接近最优跳转表。

7.7 南柯一梦：从无穷到无有

本书多次说过，概率分析虽然基本上是用来分析算法的时空效率，但也可以用来分析算

法本身的结果。现在我们再来看第 1 章已经讨论过的“从无有到无穷”的问题。

第 1 章已经讨论过，如果每次取的球是 $10n$，即第 n 次放球后取出的是编号为 $10n$ 的球，则罐子在零点整的时候将含有无数个球。但如果每次取出球的编号为 n，即第 n 次放球后取出的是编号为 n 的球，则罐子在零点整时将是空的。

现在我们将实验再改动：每次拿球的时候不是按照给定的编号拿球，而是随机取球，即

在差 1 分钟到零点时，将标号为 1～10 的 10 个球放进罐子，然后从罐子里随机拿出一个球；

在差 1/2 分到零点时，将标号为 11～20 的 10 个球放进罐子，然后从罐子里随机拿出一个球；

在差 1/4 分到零点时，将标号为 21～30 的 10 个球放进罐子，然后从罐子里随机拿出一个球。

……就这样将游戏进行下去。假设放球和取球还是不占时间，请问，当时钟指向零点时，罐子里剩有多少个球？是无有，还是无穷？还是别的一个什么数呢？

读者能够数出来吗？

答案是：0！即在零点整时，罐子里剩下的球为 0！不，精确地讲，应该是罐子里面球的个数为 0 的概率是 1。（读者发现这两种陈述有什么不同吗？）而要证明这个结果，就需要进行概率分析。

怎么进行分析呢？

当然是针对一个特定编号的球，看看在零点的时候该球还在罐子里或不在罐子里的概率是多少，然后将这个结果推广到所有的球，就可以得出罐子在零点时的含有多少个球。

不失一般性，我们可以考虑编号为 1 的球。我们定义 E_n 为取出 n 个球后，球 1 仍然在罐子里的事件，则事件 E_n 发生的概率为：

$$P(E_n)=[9\times18\times27\times\cdots\times(9n)]/[10\times19\times28\times\cdots\times(9n+1)]$$

因此，在时钟指向零点时，球 1 仍然在罐子里的事件可以表示为：$\bigcap_{n=1}^{\infty}E_n$。

由于事件 E_n 是概率递减事件，则有：

$$\begin{aligned}P\{\text{零点时球 1 仍在罐子里}\}&=P(\bigcap_{n=1}^{\infty}E_n)\\&=\lim_{n\to\infty}P(E_n)\\&=\prod_{n=1}^{\infty}[9n/(9n+1)]\\&=0\end{aligned}$$

如果以 F_i 表示时钟指向 0 点时球 i 在罐子里的事件，则罐子在时钟指向零点时为非空的

事件为$\bigcup_{i=1}^{\infty}F_i$，而这个事件发生的概率为：$P(\bigcup_{i=1}^{\infty}F_i)$。那么这个概率是多少呢？

我们已经证明 $P(F_1)=0$。同样，我们可以证明对于所有的 i, $P(F_i)=0$，

根据布尔不等式，有 $P(\bigcup_{i=1}^{\infty}F_i)\leqslant\sum_{i=1}^{\infty}P(F_i)=0$。

结论：在零点时，罐子为空的概率为 1。

这里要提请读者注意的是，球的个数为 0 和球的个数为 0 的概率为 1 是完全不同的两回事。这个微妙的区别读者能理解吗？由于这个问题已经超出了算法领域，因此本书在此略去。

至此，拿球方式略微改变，罐子里球的个数就从无穷又变为无有了！

7.8 概率分析的其他应用

概率分析是算法分析的四种方法之一。在算法平均情况下的时间成本和空间成本分析中，这种分析常常发挥关键的作用，而在涉及概率分布的时候，它更是必不可少的分析手段。

对于确定性算法来说，概率分析意味着平均情况分析，即在所有输入分布都可能发生的情况下求取算法时间成本或空间成本的期望值。对于概率或随机化算法来说，概率分析除了考虑要输入的分布情况外，对算法过程中的随机步骤也需要考虑周全。此外，概率分析并不只用来分析算法的效率和成本，也可以用来分析算法的结果输出或正确性。

这里要提请读者注意的是，概率分析与概率算法（probabilistic algorithm）是两码事情。概率分析是使用概率手段对算法进行分析，而概率算法是一种结果或时空效率具有或然性的算法。概率算法需要概率分析的手段，而概率分析却不一定只应用在概率算法上。

本章介绍的概率分析方法将在本书后面的章节中得到广泛的应用。例如，在分析随机化快速排序的时候，概率分析就发挥了淋漓尽致的效用。

思考题

1. 请计算 n 个候选人序列里面存在一个长度为 $n/2$ 的（个人资格）递增子序列的概率。
2. 你对本章中随机排列问题的解答有什么看法？你觉得我们的分析正确吗？为什么？
3. 如果不使用婚姻介绍所，而是用自由认识的方式来寻找梦幻情人，你能设计出什么算法来帮助单身人士以最低的成本寻找到他们各自的梦幻情人？
4. 定义 k-排列为一个包含 n 个元素里面 k 个元素的排列。证明：这种排列的个数是 $n!/(n-k)!$。
5. 本章给出的 RANDOM-PERMUTATION 算法与我们在梦幻情人问题随机化时使用的随机

叫号算法有什么关系？请说明你的回答。

6. 如果将放球取球的实验再改动一下：每次放进去的球不是特定编号的，而是随机的，即
   在差 1 分钟到零点时：将任意 10 个球放进罐子，然后从罐子里随机拿出一个球；
   在差 1/2 分到零点时：将任意 10 个球放进罐子，然后从罐子里随机拿出一个球；
   在差 1/4 分到零点时：将任意 10 个球放进罐子，然后从罐子里随机拿出一个球；
   ……
   那么，在零点整的时候，罐子里有多少个球呢？请加以说明。
7. “罐子里球的个数为 0”和“罐子里球的个数为 0 的概率是 1”，这两句话有什么不同吗？
8. 本章通过概率分析得出寻找到“梦幻情人”的概率为 36%，你相信吗？给出理由。
9. 本章在随机排列问题解答时证明了我们获得的是一个均匀随机排列。这是真的吗？你有不同想法吗？给出你的理由。
10. 你觉得你现在的状态是你努力或放弃的结果，还是运气使然？说明你的理由。
11. 如果将放球取球的实验再改动：将球的编号拿掉，且每次随机地选取 m_i 个球放进罐子，再随机的从罐子里拿出 n_i 个球，这里假定 $m_i \geq n_i$。
    在差 1 分钟到零点时：将任意 m_1 个球放进罐子，然后从罐子里随机拿出 n_1 个球。
    在差 1/2 分到零点时：将任意 m_2 个球放进罐子，然后从罐子里随机拿出 n_2 个球。
    在差 1/4 分到零点时：将任意 m_3 球放进罐子，然后从罐子里随机拿出 n_3 个球；
    ……
    那么在零点整的时候，罐子里有多少个球呢？请加以说明。
12. 在我们对跳转表的分析中，我们是以 $\log n$ 个链表构成的跳转表为分析目标的。这样就有一个问题：如果将链表个数提高到 $\log n$ 个以上，是否还能继续提升查找效率？

PDG

CHAPTER 8

第8章 摊销分析

中国很多古装电影里不时会出现这样一个镜头：某大臣因办某件事情失败，被皇帝怪罪，这时大臣（或旁边的人）就会求情说，看在该大臣曾经辅佐先帝或立下汗马功劳等，希望能够从轻发落或者免除死罪（如图 8-1 所示）。这种请求的意思就是，如果将其一生摊开来看，这一次失利并不显得怎样罪大恶极。这就是摊销！即将某件事情摊销到所有的事情上来看或者分析。

摊销这种功过相抵的评价事物的方法并不只在中国使用。当阿兰·图灵被发现是一个同性恋者时，英国政府曾打算对他进行严惩（在 20 世纪 40 年代的时候，同性恋在英国属于犯罪）。不料很多人为他说情，而主要的说辞就是图灵为盟军在第二次世界大战中的胜利立下了汗马功劳（图灵破译了德国海军密码“谜”），可以抵掉他同性恋的罪过。

图 8-1 为大臣求情时常常用到摊销

显然，一个人总会有犯错或者栽跟斗的时候。这个时候我们当然希望别人会考虑到我们以前也曾经做过好事，可以抵掉或者至少减轻当前的错误。

对于算法来说，也会存在某次运算特别不顺畅，以至于时间复杂、成本高昂。但也许在其他一些时候，算法运转的非常顺利，成本低廉。这样多次运算综合起来进行平均分析就是摊销分析（amortized analysis）。摊销分析有时也叫做平摊分析。但是平摊分析这个表述有很大的误导作用，容易让人感觉成本是均匀（即平均）摊派在个体的操作上。但摊销分析并

不总是将成本平均摊派下来。因此，为了精确起见，本书一律称为摊销分析。

摊销在商业活动中经常出现。对于一个公司，尤其是上市公司来说，它的各种报告里经常会出现摊销（amortized）这个词。例如，它会将某次购买大额设备的成本分摊到多个季报里面，从而使得每个季报所显示的开销较低，避免季报之间盈利的大幅震动。

既然摊销分析并不一定是将成本平均摊派下来，那么到底什么是摊销分析呢？

8.1 什么是摊销分析

2010 年 4 月 27 日～5 月 2 日，我作为 IEEE 软件工程知识体系（SWEBOK）领域编辑（KA Editor），应邀前往洛杉矶参加该领域编辑会议。由于费用全部由 IEEE 计算机协会承担，会后需要提交费用报销表格。报销表格以每天为单位填写。由于机票发票只有一张 8 351 元，租车发票也只有一张 1 550 元（227 美元），因此只能将其放在一天，这样就导致 4 月 27 日一天的费用很高，为 9 900 元，而其他天的费用则很低，为 0 元。如果采用直接的计数分析，则一天的费用最高为 9 900 元，那么 6 天的总费用最高可达 59 400 元。这种分析显然比较离谱。

但是，如果考虑到租车的费用和飞机票覆盖的是 6 天，如果第 1 天已经记录了这两个费用，其他天就不可能再出现这类费用，即 6 天的总费用最高也是 9 900 元，平均一天 1 650 元。

这就是摊销分析。

从上面的描述里，读者很可能会认为摊销分析就是求平均。这种理解也对也不对。对的部分是，摊销分析在某些情况下确实是求某种平均，即一系列操作下每个操作的平均成本；而不对的就是我们前面说过的，摊销分析在另外一些时候并不是将成本平均摊派到个体操作上。更为紧要的是，即使是在求平均的时候，摊销分析里的这种平均也并不是我们平时所熟知的考虑所有情况的平均，而是在最坏情况下，一系列操作的平均成本！

例如，假如一个算法的平均时间复杂性为 $n\log n$，而最坏情况下的时间复杂性为 n^2。现在我们问，如果这个算法运转一次，在最坏情况下，时间复杂性是多少？答案显而易见：n^2。但如果这个算法运转两次（当然输入可能不同），如果考虑最坏的情况，请问时间复杂性是多少？很多人都可能回答是 $2n^2$（每次都是最坏情况）。如果三次呢？$3n^2$。n 次呢？当然是 $nn^2 = n^3$。这样，平均来看，最坏情况下，每次运转的时间复杂性都是 n^2，即最坏情况下，一组操作里的每个操作的成本都是 n^2。这显然与平均情况下的算法时间复杂性是不一样的。而这种不一样正是摊销分析与平均情况分析的关键不同。

也许读者会觉得，既然是对一组操作在最坏情况下进行的分析，那当然是每次操作按最坏情况分析，因此摊派到每次操作上的时间成本不就仍然与最坏情况下的分析一样吗？

这看上去似乎挺有道理，但是，这真的有道理吗？

也许你有过这样的生活经验，当一个人倒霉到一定程度的时候，他就会时来运转。也就是说，倒霉不会一直持续下去。英语里有句俗话叫做：When you are at the bottom, the only

way to go is up!中文也有俗话叫否极泰来！这些话说的是，我们连续运转某个算法 n 次，每次都碰到最坏情况的可能性应该不太大。或者说，我们对 n 次运转时的最坏情况分析太过悲观了。因为一连串的运转，总会有运气好的时候。更为重要的是，有时因为特定的算法和数据结构，最坏情况不可能连续发生。因此，上述运转 n 次的最坏情况分析很可能没有反映现实，而反映现实的分析需要新的方法。这个方法就是本章要讨论的摊销分析！

到目前为止，我们讨论的算法分析都是针对算法运行一次的情况下，最坏、最好和平均的时间成本。但这种分析是否符合我们的一切需要呢？答案是否定的。因为一个算法最坏情况是立方级，并不表示它每次运行都是立方级。如果将算法运行很多次，我们希望对这很多次的运行时间进行最坏情况下的成本分析。我们的希望是，如果运行很多次，即使在最坏情况下，也不可能每次都出现最坏情况！（有点拗口吗？）

到这里，摊销分析和平均情况分析的不同就非常的明显了：摊销分析针对的是最坏情况下的平均，而平均情况分析针对的是平均情况下的平均（也许有点拗口，但读者应该知道这是什么意思）。从另一个角度说，摊销分析针对的是一个操作序列在最坏情况下运行的状况，而平均情况分析针对的是输入在概率分布下算法运行一次的状况。换句话说，平均情况分析需要使用概率，而摊销分析却不需要使用概率。

8.2 摊销分析与数据结构

更为特殊的一点是，在算法领域，摊销分析经常是针对某个数据结构进行的，（为什么？）也就是要找出在对该数据结构进行一系列操作时（一个操作序列），在最坏情况下每个操作的平均成本。这个成本就称为最坏情况下一个操作所需要的摊销成本。

摊销分析的目的就是要证明：虽然某些个体操作成本高昂，但平均来讲，每个操作的成本却是较小或者可控制的。记住，这里的“平均”不意味着“输入的概率分布平均”。

摊销分析的一个重要考虑就是哪些操作序列是可能发生的。对于一个算法来说，这可能涉及概率分析，但对于数据结构，这种判断经常是直截了当的。这是因为一个数据结构常常会维持在某个稳定或一致的状态，直到某个操作改变这个状态为止，即最坏情况发生了！虽然改变状态的成本非常高昂，但是数据结构的状态一旦改变，将在接下来较长时间维持在新的状态下而不再改变，因此我们可以将刚才改变状态的昂贵操作“摊销”到接下来的操作序列里。当然，我们也可以摊销到改变状态前的操作序列里。

例如，在程序设计时，人们有时会用到动态数组。动态数组的一个特点是在程序运行过程中可以动态改变大小。开始我们会给数组一个初始容量空间。只要这个空间没有充满，那么这个动态数组所占用的空间就是一个不变的稳定状态。但是，如果该数组空间用罄后，下一个插入操作（即最坏情况）将导致数组溢出，此时就需要扩展动态数组，而这个扩展操作显然是要产生成本的，并且是高成本，从而导致最后这次插入的成本远高于以前的插入操作。如果数组空间的扩展比较明智，即一次扩展较多的空间（如将数组的空间加倍），则随后的多次插入操作将不会导致数组扩展，从而使插入操作的成本再度降低到

可接受的范围。

在极端的情况下，一次插入操作的时间复杂性可能是 $O(n)$（即这次操作导致溢出，并进而导致空间扩展，且扩展的幅度为 n 个单元）。但是，由于随后的 $n-1$ 个操作都不会产生溢出，从而将成本维持在单位成本内，使得 n 个插入操作的总时间复杂性也为 $O(n)$ 。这样，在摊销分析下，每个操作的平均时间复杂性为 $O(n)/n = O(1)$。

显然，对于一个用户来说，他并不需要关心某一个插入操作导致数组扩展从而使得成本增加，他关心的是这个动态数组的总性能如何，或者说，每个插入操作的平均时间性能。只要这个平均值是可以接受的，则整个动态数组的设计就是可以接受的。

8.3 摊销分析的几种方法

前面已经说过，摊销分析是针对一组操作进行的（如果针对一个操作，还摊销到哪里去呢）。那么，如何对一组操作进行分析呢？按照常规思维，对一组操作进行分析可以有两种方式。

第一种方式是直接记录这一组操作的总成本，然后除以操作数即可获得每个操作的摊销成本。由于个体的成本不是显性记录，而是聚集在总成本中，因此这种摊销分析被称为**聚类分析**（aggregate analysis），即将一个类里的所有操作的成本全部聚集在一起。聚类分析的最大优点是可以避免对每个个体操作进行逐个分析，从而降低了分析的复杂性。

有左就有右，有白就有黑。既然可以对一组操作进行整体考虑，当然就有针对每个个体操作进行考虑的分析方法。而这恰恰是另外一种摊销分析方式：对每个个体操作进行分析，计算每个操作的摊销成本，然后累加每个个体操作的摊销成本而获得一组操作的总摊销成本。这种摊销分析由于很像会计做账时每笔账务都要清楚记录在案，最后得出一个总报表（总收益或亏损），因此被称为会计方法（accounting method）。

这里需要注意的是，在会计方法里，由于是先计算个体成本，然后计算总成本，似乎与摊销的概念正好相反。但是，因为计算个体成本的方法是“摊销”，且在算出总成本后，仍然可以再摊销一次到每个个体操作，所以，会计方法仍然是摊销分析。

8.4 聚类分析

前面已经说过，聚类分析就是将属于一类的所有操作聚集起来作为一个操作来计算成本，然后除以操作数即获得每个操作的摊销成本。例如，如果打一场篮球赛，结束后一个队的得分就是这个队所有队员的总得分，用这个得分除以队员数就获得每个队员的平均得分。当然，这个例子与算法里的聚类分析还是有所区别：聚类分析针对的是最坏情况下每个操作的平均成本。但如果一个球队在打球过程中每个队员的表现都是平时最差的水平，则刚才的分析就接近摊销分析中的聚类分析了。下面我们以最常见的栈操作来阐述算法里的聚类分析。

8.4.1 栈操作的聚类分析

栈是一个后进先出数据结构，其主要操作是压栈（PUSH）和出栈（POP）。

PUSH(S, x): 将元素 x 压到栈顶，时间成本为 $O(1)$。如果进行 n 次压栈，则时间为 $O(n)$。

POP(S): 将栈顶元素弹出，时间成本为 $O(1)$。如果进行 n 次出栈，则时间为 $O(n)$。

一摞书就像一个先进后出栈，需要从最上面的书开始拿起，如图 8-2 所示。

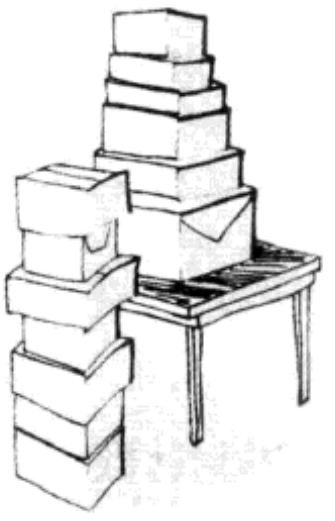


图 8-2 一摞书就像是一个先进后出栈，需要从最上面的书开始拿起

现在，假定增加一个操作：多次弹出操作（MULTIPOP）。该操作一次弹出多个元素。例如，MULTIPOP(S, k)将从栈 S 连续弹出 k 个元素。如果栈里的元素个数不到 k，则一直将栈弹空为止。MULTIPOP 实际上是使用 POP 来实现的：

```
MULTIPOP(S, k)
while(S ≠∅&&k>0) {
    POP(S);
    k =k - 1;
}
```

从上面的实现可以看出，MULTIPOP 的成本为实际进行的弹出操作的次数，而 POP 操作执行的次数为 while 循环的次数。上述 while 循环的次数为 min ($|S|$, k)，这里$|S|$代表栈 S 里的元素个数。由于我们已经假设每次 PUSH/POP 操作的成本为 1，因此，MULTIPOP(S, k)的操作成本为 min ($|S|$, k)。

引入 MULTIPOP 后，我们现在来考虑一组压栈、出栈操作：即一系列的 PUSH 和 MULTIPOP 操作。假定我们一共进行了 n 次操作，那么最坏情况下，这 n 次操作的成本是多少呢？

由于没有说明 n 次操作里 PUSH 和 MULTIPOP 各进行了多少次，似乎无法进行分析。但我们这里做的是摊销分析！而摊销分析需要按照最坏的情况进行分析！而最坏的情况是可以推断出来的。由于 n 次操作最多可将 n 个元素压入栈里，栈 S 里最多将有 n 个元素。而 MULTIPOP 在最坏情况下无非是弹出栈里所有元素（此时成本最高），因此最坏情况下的成本为 $O(n)$。而最坏情况是进行的 n 次操作全部都是 MULTIPOP，即 MULTIPOP 操作进行了 n 次，每次 MULTIPOP 操作也处于最坏情况，即成本为 $O(n)$，因此总成本为 $O(n^2)$。

但这个分析正确吗？

仔细查看栈这种数据结构可知，一个元素被弹出的次数不能超过其被压入栈的次数。如果只进行了最多 n 次 PUSH，则 POP 操作执行的次数不能超过 n 次。因此，我们的分析实际上是太过悲观，如果 n 次操作全部都是 MULTIPOP 操作，则每个 MULTIPOP 里面的 POP 操作将为 0 次，因为没有栈里没有元素。细心的读者可能已经察觉，最坏情况是该组操作里面

包括 $n-1$ 个压栈操作，而最后一个操作为 MULTIPOP。这样一共将有 $2(n-1)$次压栈、出栈操作，即总成本为 $2(n-1)$。因此，一组 n 次 PUSH 和 MULTIPOP 的操作序列的总成本只能为 $O(n)$。把这个时间分摊在 n 个操作上，每个操作的摊销成本只有 $O(1)$。

这里有两点需要提请读者注意：

1）由于不知道 PUSH 和 MULTIPOP 各进行了多少次，所以单独计算个体操作的成本就不太容易，但总成本却可以很容易算出上限。因此，非常合适使用聚类分析。

2）这里的分析没有用到概率。我们的计算方法是先证明整个操作序列的总成本在最坏情况下为 $O(n)$，即直接计算总成本，然后除以操作数获得每个操作的摊销成本为 $O(1)$。

上面所说的就是聚类分析，将一类操作聚集在一起进行总成本分析，然后平摊到每个操作上。

8.4.2 二进制计数器的聚类分析

另一个彰显聚类分析功能的非常有说服力的例子是所谓的二进制计数器。这种计数器对于计算机的正常运转非常重要：计算机里的时钟、闹铃、控制器的序列产生器根本上都是二进制计数器，或者说递增器。例如，图 8-3 描述的就是计算机里面的时钟计数器。

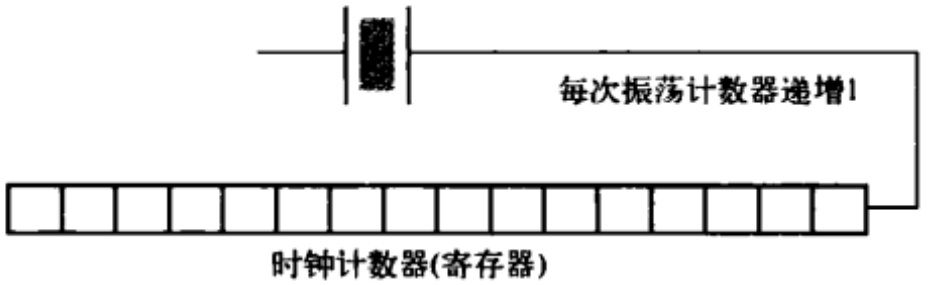


图 8-3 计算机里面的时钟计数器

所有的时钟计数器从抽象层面上看都是一个二进制计数器。一个 n 位的二进制计数器可以用数组 $A[0..n-1]$来表示，这里 $A[0]$是最低有效位，$A[n-1]$为最高有效位。

假定计数器从 0 开始往上计数，一直到计数器溢出为止，即计数到 2^n-1 为止。显然，在计数过程中，计数器的不同字位可能发生变化：1 变为 0 或者 0 变为 1。而这种 0 到 1 或 1 到 0 的翻转操作是有成本的。假定每翻转一个字位的成本为 1，如果某次递增操作翻转了 x 个字位，则该次递增操作的成本为 x。如果某次递增操作没有翻转任何位，则该次递增操作的成本为 0（显然不翻转任何位的递增操作是不存在的）。例如，第 1 次递增将把最小有效位从 0 翻转为 1，且没有别的翻转，因此其成本为 1。但如果在计数器为 0111 的情况下递增，则需要翻转 4 位，因此，这次递增操作的成本为 4。递增函数实现如下：

```
INCREMENT(A)
i = 0;
while(i <n&&A[i] = 1) {
    A[i]= 0;
    i =i+1;
```

```
    }
    if(i<n)
      A[i] = 1;
```

现在我们的问题是：在计数终止时（即计数到溢出时为止）平均每次计数的摊销成本为多少？

显然，在最坏情况下，一次递增操作可能需要翻转 n 位，到计数终止时 INCREMENT 一共被调用 2^n 次，因此，该计数器总成本为 $O(n\times 2^n)$次。

但是，成本真的有这么高吗，我们以一个 4 位的计数器计数到 8 来具体计算，计数器每次递增时的成本如表 8-1 所示。

表 8-1 计数器每次递增时的成本

计数器值	字位变化	成本
0	0000	0
1	0001	1
2	0010	2
3	0011	1
4	0100	3
5	0101	1
6	0110	2
7	0111	1
8	1000	4

这样 8 次递增操作的总成本为 15（次翻转），除以 8 之后的每次递增平均成本小于 2。而按照最坏情况分析，总成本为 $4\times 2^3=32$（次翻转）。每次递增的成本为 4！

显然，我们对最坏情况的分析太过悲观了，因为不可能每次递增操作都会翻转所有字位。事实上，字位翻转的频率远低于最坏估计。仔细分析发现，每个字位翻转的频率如表 8-2 所示。

表 8-2 计数器每个字位的翻转频率

字位	翻转频率(%)	翻转次数
0	100	n
1	50	floor(n/2)
2	25	floor(n/4)
3	12.5	floor(n/8)
…	…	…
i	$(1/2^i)$	floor(n/2^i)

因此，在整个 n 次递增操作序列里，字位翻转的总次数为：

$$\sum_{i=0}^{2^n-1}[n/2^i] < n\sum_{i=0}^{\infty}[1/2^i] = n(1/(1-1/2)) = 2n$$

因此，n 次递增操作的总成本只有 $O(n)$，所以平均每次递增的摊销成本为 $O(1)$。

这就是聚类分析的能力！

8.5 会计分析

聚类分析的特点是将所有操作看做一个整体来计算其总成本，然后用总成本除以操作数获得摊销成本。但这并不是获得摊销成本的唯一办法，另一个办法是分别计算每个操作的成本，然后累计再平均。这就是会计分析。如果还用打篮球做比喻，则是按照球员来记录进球数，最后将一个队的所有球员的进球数累加起来除以队员数即获得摊销进球数，如图 8-4 所示。

根据上面的描述，在会计分析里，每个操作的成本都在操作发生的时候立即记录，而每个操作的成本有可能不同。由于分析个体操作的精确成本有时候很困难，因此，我们给每个操作赋以一个不同的（摊销）成本，而这个摊销成本值有可能高于或低于一个操作的实际成本。如果这样计算的最后总成本高于实际总成本，则将超额计算的成本存起来作为信用，用于支付将来（后续操作）的债务（即后续操作的实际成本大于摊销成本的情况）。

图 8-4 按球员个体来记录每次进球，最后累积全队进球数就是会计分析

由于我们的目标是计算一组操作的成本上限，因此信用不能为负；否则我们计算出的摊销总成本将不能成为相关操作序列的成本上限！从而导致计算出的结果毫无意义（因为它不能告诉我们一组操作的总摊销成本最多是多少）。如果设 c_i=第 i 个操作的实际成本，$\hat{c}_i$=第 i 个操作的摊销成本，则我们要求所有 n 个操作的序列都满足下述不等式：

$$\sum_{i=1}^{n}\hat{c}_i \geqslant \sum_{i=1}^{n}c_i$$

而在该操作序列结束后，余留的信用则为：

$$\sum_{i=1}^{n}\hat{c}_i - \sum_{i=1}^{n}c_i \geqslant 0$$

由此可见，会计分析与聚类分析除一个对整体进行分析一个对个体进行分析外，它们之间还有一个显著不同：聚类分析里的所有操作的摊销成本一样，而会计分析里不同的操作可以有不同的摊销成本！例如，以栈的操作为例，我们给栈的三个操作赋以如图 8-3 所示的摊销成本。

表 8-3 栈操作的摊销成本

操作	实际成本	摊销成本
PUSH	1	2
POP	1	0
MULTIPOP	min(|S|, k)	0

这样赋以摊销成本的理由如下：当压入一个元素时，实际成本为 1，但是压入的东西终归是要弹出的，即它将在将来产生 1 个弹出成本。但与聚类分析不同的是，我们现在就将其将来需要的成本计入，因此其摊销成本为 2。由于后续弹出操作的成本已经计入 PUSH 里，因此 POP 和 MULTIPOP 的摊销成本为 0！而这种摊销成本分派将导致栈里每个元素的信用为 1，即栈的总信用≥0（注意，栈里元素个数不可能为负）。这样，如果我们面对的是一个 n 个操作的序列，则总摊销成本可以很容易地计算出来：最多 $2n$。因此，总摊销成本为 $O(n)$，而这确实是实际总成本的上限，并且与我们前面的聚类分析结果一样！

二进制计数器的会计分析

如果使用会计分析来分析二进制计数器的总摊销成本，则将字位从 0 翻转为 1 的成本指派为 2 个：1 个用于将字位设置为 1，1 个用于将来将其翻转回去。这样，字位从 1 翻转为 0

将没有摊销成本。这种指派将导致计数器里面的每个 1 都有 1 个信用，因此，总信用≥0。

递增函数 INCREMENT 的摊销成本计算如下：将 1 的字位翻转为 0 的成本由信用来抵消，因此可以不予以考虑。而将 0 翻转为 1 的次数最多为 1 次，因此，其摊销成本≤2。也就是说，INCREMENT 函数每次调用的成本最多为 2，而对于 n 个操作来说，总摊销成本为 $\leqslant 2n=O(n)$。这与前面的聚类分析结果相同。

8.6 势能分析

会计分析的特点是将每个操作的当前成本和以后将产生的成本同时记录，并且保存在个体操作的档案里，这个成本就是给每个操作赋以的摊销成本。但由于以后的操作还没有发生，所以我们提前记录的是超出实际的成本，因此这种摊销成本里面就包含着信用。由于这种信用被记录在个体操作里，因此这种方法被称为**会计方法**（会计做账时就是如此）。

但是将信用存在个体操作上并不是信用存放的唯一方法。我们前面说过，摊销分析通常与某个数据结构相关，因此我们也许可以将信用直接存放在整个数据结构里供所有元素共享，用于偿付将来的操作成本，而不隶属于任何个体操作。由于这种存放在数据结构里的信用可以看做是一个数据结构的潜在支付能力，因此它被称为势能，而这种分析方法也就被称为**势能分析**（potential method）。由于信用不依附于任何具体元素，因此，势能分析非常灵活！

更为重要的是，会计分析对一个操作进行摊销成本赋值时总让人感觉到有点随意（当然我们不是随意的）或者不严谨。而势能分析将克服这个问题，使一切看上去严谨精确！

势能分析的具体方法如下：设

D_i = 在第 i 步操作后的数据结构

D_0 = 初始的数据结构

c_i = 第 i 个操作的实际成本

$\hat{c}_i$ = 第 i 个操作的摊销成本

设势能函数为 Φ: $D_i \rightarrow R$，这里 $\Phi(D_i)$ 为数据结构 D_i 的势能。而操作 i 的摊销成本与实际成本的关系可以通过势能函数连接起来：

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = c_i + \Delta\Phi(D_i)$$

这里，$\Delta\Phi(D_i)$是因第 i 个操作导致的势能变化。这个公式我们称之为**势能变化公式**。

有了上述摊销成本的表示，由 n 个操作构成的操作序列总摊销成本为：

$$\sum_{i=1}^{n}\hat{c}_i = \sum_{i=1}^{n}(c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^{n}c_i + \Phi(D_n) - \Phi(D_0)$$

如果要求对于所有的 i 有 $\Phi(D_i)\geqslant\Phi(D_0)$，则摊销成本将总是实际成本的上限。在实际情况下，$\Phi(D_0)=0$, $\Phi(D_i)\geqslant 0$，$i=1,2,3,\cdots,n$。下面我们来对栈和二进制计数器进行势能分析。

由势能分析的定义可以看出，势能分析的关键是设计一个势能函数，这个函数必须满足一些条件。一旦有了这个势能函数，给每个操作赋以摊销成本就变成了按公式行事了！这看

上去比会计分析中似乎随意的摊销成本赋值要精确多了！下面以例子来说明势能分析。

8.6.1 栈操作的势能分析

要对栈进行势能分析，就要定义一个合适的势能函数。而要定义一个合适的势能函数，就要分析什么是栈的势能。根据 8.5 节的会计分析，栈的势能似乎应该是栈里元素的个数，因为，每推入一个元素到栈里，就意味着将来需要为出栈付出代价。这个代价必须作为信用存放在栈里，而这就是栈的势能。因此，定义栈的势能函数为：Φ=栈里元素的个数。

在 i=0 的时候，尚未发生过任何操作，栈里元素为空。因此，D_0 的势能为 0，即 $\Phi(D_0)=0$。

由于栈里的元素个数总是≥0，我们有 $\Phi(D_i)\geqslant 0=\Phi(D_0)$。因此，我们的势能函数定义完全符合势能分析的要求。

有了这个势能函数，按照势能公式，就可以得出每个操作的摊销成本如表 8-4 所示。

表 8-4 栈操作的势能分析

操 作	实际成本	ΔΦ	摊销成本
PUSH	1	$(s+1)-s=1$，s 为最初元素个数	1+1=2
POP	1	$(s-1)-s=-1$	1−1=0
MULTIPOP	$K'=\min(k, s)$	$(s-k')-s=-k'$	$k'-k'=0$

从上面的分析可见，每个操作的摊销成本计算都是按照公式进行的，给人很精确的感觉。而有了每个操作的摊销成本，一个由 n 个操作构成的操作序列的摊销成本很显然就是 $O(n)$。

非常简单，优美！不是吗？

8.6.2 二进制计数器的势能分析

对于二进制计数器来说，从势能的角度看，计数器在任何时候的势能是其字位 1 的个数。因此势能函数为：

$$\Phi=b_i=\text{第 } i \text{ 次递增后计数器里 1 的个数}$$

假设第 i 次操作将 t_i 个字位设置为 0，则 $c_i\leqslant t_i+1$ (复位 t_i 个字位，设置为 1 的字位个数≤1)。此时，有下面两种情况需要考虑：

1）$b_i=0$，即第 i 次递增后计数器里面 1 的个数为 0。这意味着在第 i 次递增前整个计数器里面的字位都是 1，即第 i 次操作复位所有 n 个字位，而没有设置 1（溢出）。所以，

$$b_{i-1}=t_i=n \Rightarrow b_i=0=b_{i-1}-t_i$$

2）如果 $b_i>0$，第 i 次操作复位 t_i 个字位，设置 1 个 1，所以 $b_i=b_{i-1}-t_i+1$。

不管哪种情况，都有 $b_i\leqslant b_{i-1}-t_i+1$。因此，

$$\Delta\Phi(D_i)\leqslant (b_{i-1}-t_i+1)-b_{i-1}=1-t_i$$

根据势能公式，有，

$$\hat{c}_i=c_i+\Delta\Phi(D_i)\leqslant(t_i+1)+(1-t_i)=2$$

即每次操作的摊销成本最多为 2！因此，n 个操作的摊销总成本为 $O(n)$（当然，假定计数器从 0 开始，即 $\Phi(D_0)=0$）。这与前面的聚类分析和会计分析结果一样！

8.7 摊销分析应用：表格扩展的代价

人活着总是需要空间来存放各种个人物品，例如用书架来装书，用衣橱来装衣服。随着时间的推移，我们收集的书或衣服可能越来越多，终于有一天书架不够用了。这个时候我们就会买一个新的、更大的书架，把以前的书都转移到新的书架上，并把旧的书架丢掉（释放掉）。当然，我们的藏书也可能越来越少（因为不断将伪书和垃圾书扔掉），到了一定时候就会发现所用的书架太大。为了节省空间，我们就会买一个更小的书架，将书从大书架转移到小书架上，并将大书架扔掉（如图 8-5 所示）。但如此地对书进行挪动需要消耗多少精力呢？

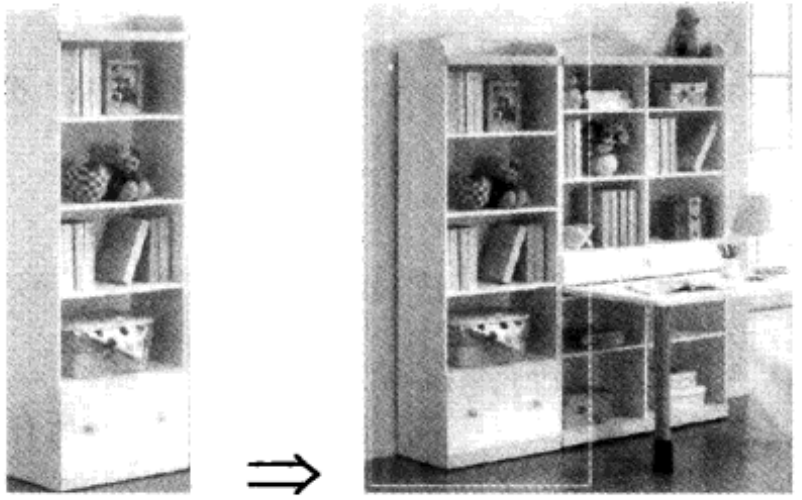


图 8-5 将小书架换成大书架需要多少（移动书本的）成本呢

程序也一样。它们也需要空间来存放各种东西。当然这个东西不是书，而是数据！程序在运行的过程中所收集的数据也有可能增加或减少。如果增加，原来分配的空间就可能不够，那么就需要就进行空间扩展。如果减少，原来分配的空间就可能太大，需要进行空间减缩（严格来讲，不是需要进行减缩，而是最好进行减缩，以节省空间）。而具备这种伸张与收缩的数据存放机制就是动态表格。当然，动态表格是一个抽象的数据结构，在程序设计里的具体实现有可能是数组、栈、队列、散列表等。

但是，不管动态表格的具体数据结构如何实现，它在装满数据的时候都会发生溢出。此时若想再插入数据，必须对动态表格进行空间扩展。这个扩展过程包括如下几个动作：

1）分配一个新的更大的动态表格。

2）将旧表里的数据转移（拷贝）到新的表里。

3）释放旧表的空间。

如果一个动态表的空间使用率很低，则需要对其进行收缩：

1）分配一个新的、更小的表。

2）将旧表里面的数据转移到新的表里。

3）释放旧表的空间。

显然，每次表格扩展或收缩都会产生成本：数据移动（拷贝）的成本。如果数据很多，这个成本将很高！如果事先知道一个程序使用的数据量，那么我们可以一次分配一个足够大的空间，从而避免发生空间扩展，进而免除此种高昂的成本。但由于我们在很多时候并不能事先知道一个程序到底有多少数据，从而不得不使用动态表。只要表格扩展和收缩的次数不太多，这个成本也许是可以忍受的。下面我们就来分析一下动态表格的摊销成本。

设 T 为一个动态表，x 为待插入的元素，num(T)表示动态表里当前的元素个数，size(T)表示动态表 T 的总体容量（总大小）。假定有一个创建动态表的操作 CREATE-TABLE(n)，其功能是创建一个大小为 n 的空表；一个插入元素的操作 insert(T,x)，其功能是将元素 x 插入到动态表 T 里。为简单起见，我们只考虑表格插入操作：往表格里插入一个元素的成本记为单位成本。当表格装满时，我们分配一个空间是原来表格两倍大的新表。

下面是插入操作的伪代码程序：

```
INSERT(T, x)                                  //在动态表 T 里面插入元素 x
if(size(T) == 0) {
      T=CREATE-TABLE(1);                      //创建表格，分配 1 个单元
      size(T) = 1;
}
if(num(T) == size(T)) {                       //扩展表格
      NT= CREATE-TABLE(2·size(T));            //创建一个 2size(T)大小的新表格
      size(NT)=2·szie(T);
      for (i=1; i<=size(T), i++) {
            NT[i]=T[i];                       //将原表格所有元素插入到新表格
      }
      free T;                                 //释放原表格 T
      T=NT;
      size(T) =size(NT);
}
T[i+1]=x;                                     //插入元素 x 到 T 里，1 次插入操作
num(T)= num(T) + 1;
```

表格的初始状态为 num(T) = size(T) = 0。图 8-6 演示的是一个插入过程。

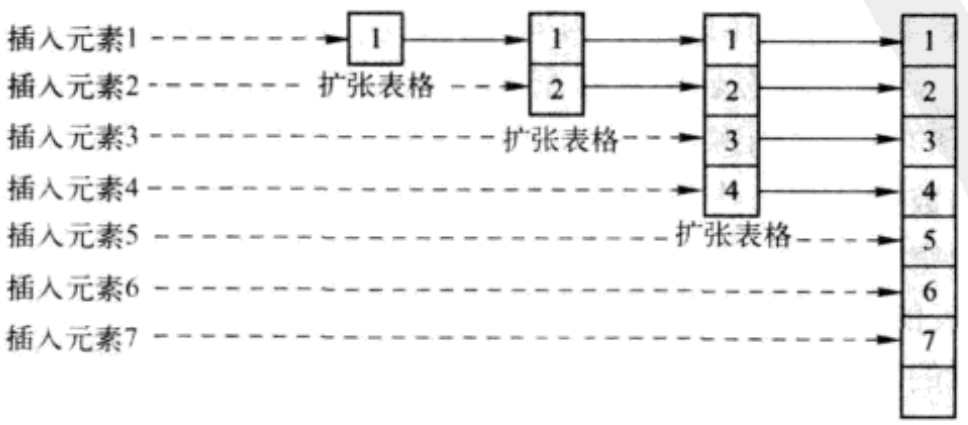


图 8-6 动态表的前 7 次插入操作

那么，动态表的插入算法时间成本怎样呢？

显然，对于普通分析（非摊销分析）来说，上述插入操作的时间成本最坏为 $\Theta(n)$，这里的 n 就是 size(T)，即动态表 T 在本次插入操作前含有的元素个数。因为在最坏情况下，插入将导致溢出，从而需要进行数据从旧表到新表的转移，导致 n+1 次插入。

如果考虑一组 n 次插入操作，则最坏情况的时间复杂性似乎应该是 $n\Theta(n) = \Theta(n^2)$。但这个分析显然太过悲观，因为在此种插入算法的实现模式下，连续的插入操作均为最坏情况的可能性是不存在的。事实上，可以证明最坏情况下连续 n 次插入的时间复杂性 $\ll \Theta(n^2)$。因此，普通的算法分析在这里将导致误判。我们需要的是摊销分析。

下面我们就分别使用摊销分析的三种方法来对动态表的插入算法进行分析。

8.7.1 动态表插入操作的聚类分析

聚类分析的关键是考虑整个操作序列总共的插入成本。设 c_i=第 i 次插入的成本，则 n 次插入的成本=$\sum_{i=1}^{n} c_i$。问题是，c_i 是多少呢？

c_i 的取值显然与该次插入是否需要进行表格扩展有关，而表格是否扩展则与在第 i 次插入前表格是否已满有关。由于我们的表格扩展是按 2 的幂值进行，因此，如果第 i 次插入需要扩展表格，则说明 i-1 是 2 的幂值，此时的插入成本是 i（本次插入加上前面 i-1 个元素的重新插入）；而如果 i-1 不是 2 的幂值，则第 i 次插入不需要扩展表格，因此该次插入的成本是 1，如表 8-5 所示。

表 8-5 动态表每次插入的实际成本

i	1	2	3	4	5	6	7	8	9	10	…
$size_i$	1	2	4	4	8	8	8	8	16	16	…
c_i	1	2	3	1	5	1	1	1	9	1	…

显然，n 次插入操作的成本可以分解为如下两个部分：

1）每次插入操作本身的插入成本。

2）表格扩展时重新插入的成本。

第 1 项成本很清楚，n 次插入总共需要 n 个插入成本。而第 2 项成本则需要计算 n 次插入操作里会产生多少次表格扩展。因为我们按 2 的幂值分配表格，则每满 2 的幂值次插入就会发生一次扩展，因此 n 次插入共有 flooring(log(n−1))次扩展。而第 j 次扩展需要重新插入的元素个数为 2^j。因此，第 2 项的成本为 $\sum_{j=1}^{\text{flooring}(\log(n-1))} 2^j$，而这项和值不超过 $2n$。这样，n 次插入的成本是：

$$\sum_{i=1}^{n} c_i = n + \sum_{j=1}^{\text{flooring}(\log(n-1))} 2^j \leqslant n + 2n \leqslant 3n = \Theta(n)$$

因此，每次插入操作的摊销成本为 $\Theta(n)/n=\Theta(1)$。

也就是说，动态表的效率并没有想象的那么差，事实上，这个时间复杂性挺好的！

8.7.2 动态表插入操作的会计分析

对于会计分析来说，关键是计算每次插入需要多少摊销成本。显然，每次插入的当前成本为 1，因为需要插入一个元素。但除此之外，每插入一个元素，就意味着将来在扩展表格时需要支付转移的成本。因此，我们在计算一次插入操作时，还需要将未来的转移成本计算在本次插入操作上。这样，每次插入操作的摊销成本似乎为 2 个单元成本。

但这样分析正确吗？

答案是否定的。而要获得这个答案，就得分析在发生扩展时需要进行的操作。很显然，在每次表里元素个数达到 2 的幂数后的下一次插入都将发生溢出而需要扩展表格。那么对于这次发生溢出的插入来说，除自己需要使用一个单位的成本外，还需要将表里当前含有的所有元素重新插入一次。而需要重新插入的元素可以分为两个部分：一个部分是那些第 1 次需要重新插入（即前面一次扩展后本次扩展前插入）的元素；另一部分是那些在上次扩展前就已经插入的元素。对于第 1 次需要重新插入的元素来说，该次插入不产生新的插入成本，因为其扩展成本已经在第 1 次插入时支付掉了。但对于在上次扩展前插入的元素来说，其预先支付的扩充成本已经在上次扩展时用掉，因此，这次重新插入需要支付成本，而这个成本可以用后面插入的元素来支付。这样，在此次扩展后再插入的元素必须支付下面三种成本：

1）为扩展前表里包含的元素支付扩展所需要的成本。

2）为本次插入的元素支付将来的扩展成本。

3）为本次插入操作支付成本。

这里的“此次”当然是指第 1 次扩展后的任何一次扩展。也就说，除了第 1 次插入外，每次插入操作需要计算摊销成本 $\hat{c}_i$=3。这 3 个单元成本里面中的 1 个成本用于插入本元素，2 个成本用于将来的表格扩展成本，其中 1 个单元用于偿付最近插入的元素的移动（上次扩展后至下次扩展前插入的元素）所需的成本，1 个单元用来偿付旧的元素（上次扩展前插入的元素）的转移成本。表 8-6 给出的是第 1～10 次每次插入的实际成本、摊销成本和余留信用。

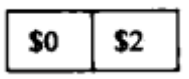
表 8-6 动态表每次插入的实际成本与摊销成本

i	1	2	3	4	5	6	7	8	9	10	…
$size_i$	1	2	4	4	8	8	8	8	16	16	…
c_i	1	2	3	1	5	1	1	1	9	1	…
$\hat{c}_i$	2	3	3	3	3	3	3	3	3	3	…
余留信用	1	2	2	4	2	4	6	8	2	4	…

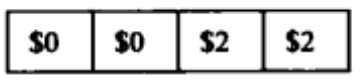
从表 8-6 可以看出，除了第 1 次插入外，其他任何一次插入的摊销成本为 3。第 1 次插入的摊销成本不是 3，而是 2。这是因为第 1 次扩展的前面没有过扩展，因此，该次插入只需要计算本次插入和本次插入元素的以后扩展成本，所以，其摊销成本为 2。

下面几个表格演示的是表格的4次扩展及动态表的总余留信用。

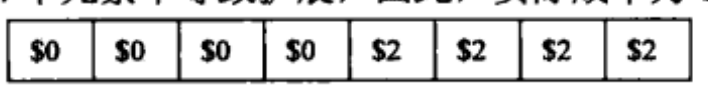
插入第1个元素导致第1次扩展，实际成本为1，摊销成本为2，余留信用为1。



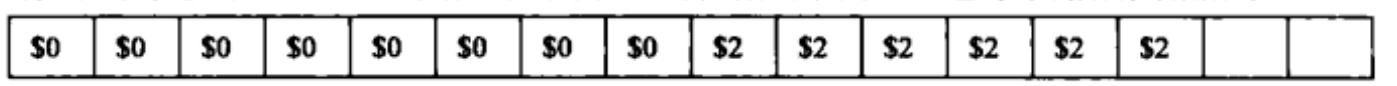
插入第2个元素导致第2次扩展，实际成本为2，摊销成本为3，总余留信用增加到2。



插入第3个元素导致第3次扩展，实际成本为3，摊销成本为3，总余留信用维持为2；插入第4个元素不导致扩展，因此，实际成本为1，摊销成本为3，总余留信用增加到4。



插入第5个元素导致第4次扩展，实际成本为5，摊销成本为3，总余留信用降为为2；插入第6个元素不导致扩展，因此，实际成本为1，摊销成本为3，总余留信用增加到4；插入第7个元素不导致扩展，因此，实际成本为1，摊销成本为3，总余留信用增加到6；插入第8个元素不导致扩展，因此，实际成本为1，摊销成本为3，总余留信用增加到8。



插入第9个元素导致第5次扩展，动态表容量增加到16个位置，因此该次插入的实际成本为9，摊销成本为3，总余留信用降为2；插入第10个元素不导致扩展，因此，实际成本为1，摊销成本为3，总余留信用增加到4；插入第11个元素不导致扩展，因此，实际成本为1，摊销成本为3，总余留信用增加到6；插入第12个元素不导致扩展，因此，实际成本为1，摊销成本为3，总余留信用增加到8……

从上述表格可以看出，由于预留信用总是大于等于0，因此，摊销成本之和确实是实际成本之和的上限。这说明我们的分析确实是正确的，即每次插入操作的摊销成本为$O(1)$。因此，n次插入操作的摊销成本最坏为$\Theta(n)$。

8.7.3 动态表插入操作的势能分析

势能分析与会计分析的唯一不同是将信用存放在整个数据结构里，而不是存放在每个元素上。其关键是设计合理的势能函数。那么，对于动态表来说，势能函数应该如何设计呢？

由于每插入一个元素就要为将来的扩展预先支付成本，同时还要为最近一次扩展前就已经在动态表里的元素预先支付扩展成本。因此，每插入一个元素就需要预先支付2个单元的成本。这样，势能函数似乎应该是$\Phi(D_i) = 2i$。但这个势能函数忽略了一个重要的事实：系统累积的势能在每次扩展的时候都会全部消耗掉！而这部分消耗的势能需要从$2i$里面减去。

那么，在第i次插入操作的时候，有多少个元素的势能被消耗掉了呢？当然是该次插入操作前的最近一次扩展时动态表的元素个数，这个数是多少呢？当然是$2^{\text{flooring}(\log i)}$。而每个元素的势能为2，因此，在第$i$次插入操作时，消耗掉的势能应该是$2\times 2^{\text{flooring}(\log i)}=2^{\text{ceiling}(\log i)}$。根据此分析，可以得出势能函数的定义似乎应该是$\Phi(D_i)=2i-2^{\text{ceiling}(\log i)}$。

为了表示的一致性，需要假定$2^{\text{ceiling}(\log 0)}=0$。

显然，有 $\Phi(D_0)=0,\Phi(D_i)\geqslant 0$，$i=1,2,3,\ldots,n$。因此，我们的势能函数定义符合规范。

势能分析和会计分析在第 3 次插入后的状态对比如图 8-7 所示。

a) 势能分析　　b) 会计分析

图 8-7　势能分析和会计分析状态对比

根据会计分析，此时系统的余留信用为 2，即系统提前支付的成本（即信用）为 2。而根据势能分析，有：

$$\Phi(D_3)=2\times 3-2^{\text{ceiling(log3)}}=6-2^2=2$$

即系统当前的势能为 2。

两种分析方式的结果完全相同。这样，有了势能函数，计算第 i 次插入的摊销成本就不费吹灰之力了。

根据势能公式，第 i 次插入的摊销成本为

$$\hat{c}_i=c_i+\Phi(D_i)-\Phi(D_{i-1})$$

而第 i 次插入操作的实际成本根据 $i-1$ 是否为 2 的幂数分为两种情况：i 或者 1。因此，有：

1）如果 $i-1$ 是 2 的幂数（第 i 次插入需要扩展表格，$c_i=i$），则第 i 次插入的摊销成本为：

$$\begin{aligned}\hat{c}_i&=c_i+\Phi(D_i)-\Phi(D_{i-1})\\&=i+(2i-2^{\text{ceiling(log}i)})-(2(i-1)-2^{\text{ceiling(log}\,i-1)})\\&=i+2-2^{\text{ceiling(log}i)}+2^{\text{ceiling(log}i-1)}\\&=i+2-2(i-1)+(i-1)\quad\text{（有没有看出来 }2^{\text{ceiling(log}i)}=2(i-1)\text{）}\\&=i+2-2i+2+i-1=3\end{aligned}$$

2）如果 $i-1$ 不是 2 的幂数（第 i 次插入不需要扩展表格，$c_i=1$），则：

$$\begin{aligned}\hat{c}_i&=c_i+\Phi(D_i)-\Phi(D_{i-1})\\&=1+(2i-2^{\text{ceiling(log}\,i)})-(2(i-1)-2^{\text{ceiling(log}\,i-1)})\\&=1+2-2^{\text{ceiling(log}\,i)}+2^{\text{ceiling(log}\,i-1)}\\&=3\qquad\text{(因为 }2^{\text{ceiling(log}\,i)}=2^{\text{ceiling(log}\,i-1)}\text{，看出来了吧！)}\end{aligned}$$

因此，每次插入操作的摊销成本为 3，即 $\Theta(1)$，而 n 次插入操作的摊销成本最坏为 $\Theta(n)$。这个结果与聚类分析和会计分析的结果完全相同！

8.8　运气不好就摊销

摊销分析的目的是对一个算法运行多次下的最坏情况分析。我们的思路是一个人的运气不可能一直很坏，而一个算法也不太可能每次运转都出现最坏情况，尤其是对于与数据结构相关的算法，即对于那些在每次运转之间需要维持某个数据结构的算法来说，这种连续最坏的情况在数据结构的限制下根本就不可能发生。此时，进行摊销分析就有着重要的意义。因

为，我们想知道，如果运气一直很差，算法的效率到底能坏到什么程度。而实际情况是，在最坏情况下，这些与数据结构紧密相关的算法效率并不坏，甚至还很好！

摊销分析的方法一般有两种：聚类分析和会计分析。聚类分析将一串操作（算法的多次运行）当成一个整体来估算成本；而会计分析则针对每个个体操作来估算摊销成本。在会计分析中，因为信用记录的方式或计算个体摊销成本的不同而又派生出势能分析。

聚类分析相对较为简单，估计读者不会感到困难。但会计分析却更符合人们的直觉，因为这就是会计做账所用的方法！其基本思想是选择某种操作作为基本操作，并将此种操作的成本记为 1 个单元；然后，给每次操作都赋以一个摊销成本。这个成本用来覆盖当前的基本操作和因此次操作导致的将来需要的操作成本。显然，会计分析的核心是选择合适的最小操作单位和对每个操作赋以合理的摊销成本，而达到这两点则有赖于对问题的深刻理解和对复杂性上限的准确估算。从这点来说，摊销分析确实不太容易。

会计分析通常被用来证明一组操作里面的单个操作时间为 $O(1)$，而它的最难点则在于某些操作可能需要比常数成本更高的成本。这就意味着，没有任何常数成本能够覆盖一个操作的最坏情况。但是，如果选择的成本合理，这个困难还是可以被克服。而成本高昂的操作只能在预留信用足够的情况下才能发生。

势能分析可以看做是会计分析（或债务方法）的推广或一般化。它与会计分析方法几乎是一模一样的，至少是相互对应的。它与会计分析的根本不同是将信用记录在整个系统里，而不是记录在单个的操作上。因此，势能分析能够通过设计一个势能函数来计算系统的总势能，并通过势能函数来给个体操作赋以合理的摊销成本。对于那些很难给个体元素直接赋以信用额度的问题，势能分析能够大展身手。

思考题

1. 如果一个算法并不使用任何持久的数据结构，即一次运转到下一次运转之间没有任何的联系，我们是否可以对这种算法也进行摊销分析呢？怎么分析？
2. 人类社会习惯将一个人一生的功过统一考虑后进行正面或负面评价。请问这种评价是更加类似于摊销分析，还是更加类似于平均情况分析？说明你的理由。
3. 你觉得摊销分析和平摊分析两种表述，哪一种更加精确地表述了本章所论述的算法分析方法？为什么？
4. 证明：在 i–1 为 2 的幂值的情况下有 $2^{\text{ceiling}(\log i)}=2(i-1)$。
5. 有一样你喜欢的东西只能在很远的一个商店买到。但你又觉得只为这一样东西跑一趟路有点不划算。此时你的一位朋友说他正好也要买这件东西，你要买的话就买两件。也许你觉得买两件就跑一趟还是不划算。此时又有三位朋友也想要同一件物品。这样，如果你跑一趟，就能买 5 件，满足 5 个人的要求。此时你就会觉得跑一趟很值得。请问这种心理的变化是否就是潜意识的摊销分析呢？请说明你的理由。
6. 给定某个非空极大堆和 m 个取值不同的元素。我们欲将这 m 个元素插入到给定的极大

堆里。但是插入操作是与取极大值穿插进行的，即我们在原始堆上进行一系列取极大值和插入操作，其中要插入的元素就是给定的 m 个元素。假定总操作数为 n。请分析此 n 个操作的摊销成本。如果需要，可自行做出合理的假设。

7. 你认为算法里的摊销分析能够应用到对历史人物的功过是非的评价上吗？如果能，请设计一种评价规则；如果不能，说明理由。
8. 你能想出生活中其他的摊销分析情形吗？大学生活里有什么重要的摊销行为吗？

CHAPTER 9

第9章 竞争分析

五年前，叶芙还是一个不为人所知的、没有申请过任何国家科研经费的研究人员，由于她对文学和哲学非常有兴趣，因此喜欢将她的研究和思想以文学和哲学的形式表达出来，从更高的角度把握科学研究的方向和趋势。她用故事来讲述理论，以准确的文字表述替换过于繁杂的公式和符号，提出的研究课题也经常与众不同，大出常规思维之外。

但是叶芙的这种研究风格并不被各种“专家”所理解。

“专家”在看到她的申请报告时，通常表现出困惑：“这个研究有重大的意义，但是研究的具体步骤是什么呢？”“申请项目的研究内容阐述过于简单，对所定义函数中的参数如何定义、度量没有阐述。没有提出本项目的关键科学问题。”这是专家惯常使用的语言。即使将报告写得连小学生都能看懂，到了专家那里还是“表述不够清晰、也未清晰表达拟解决的科学技术问题”。专家将报告批得体无完肤之后，还不忘循循善诱：“影响系统可靠性的最重要因素还是系统本身，与系统运行相关环境的研究对开发高可靠系统没有任何帮助……”

在此情形下，叶芙自掏腰包进行研究，并在这些研究的基础上研制了自己的产品。在几经周折后，产品获得了市场的接纳，各种风险投资蜂拥而至。叶芙开起了公司，被邀请到各地做报告，办书展，甚至还获得各种“成功人士”和“重量级人物”的接见。

很多“专家”在听了叶芙的报告后，颇觉过瘾，经常在会后问叶芙：“你怎么没有申请过国家项目呢？要是我见到你的申请报告，我肯定会理解你的独特思路和新颖的研究方法！”“你的研究完全可以获得国家重点项目资助！”当然，专家此时已不会记得他们对叶芙“申请报告中要提供令人信服的验证材料或实验（评测）结果”（即要求在研究前就知道研究结果）的告诫。而叶芙那些被认为“过于笼统、不具有可行性、难以进行准确研究”的各种课题也被各路“专家”欢呼为“清晰、可行、具有重大价值和前瞻性”。

9.1 什么是竞争分析

这个世界里充满竞争。各种评优，各种打分，各种奖励等层出不穷。如果你申请项目未果，其给出的理由通常会包括一句类似“这次的项目评选竞争十分激烈……”的话。而世界

上每天都会评选出各种“最”什么的东西出来，令人目不暇接。当然，这也让很多人乐此不疲。例如，在 2008 年奥运会后，世界上又出现了一个“最×××”：美国密歇根大学的迈克·菲尔普斯（Michael Phelps）成为人类奥运史上获得奥运金牌最多的人（见图 9-1）！

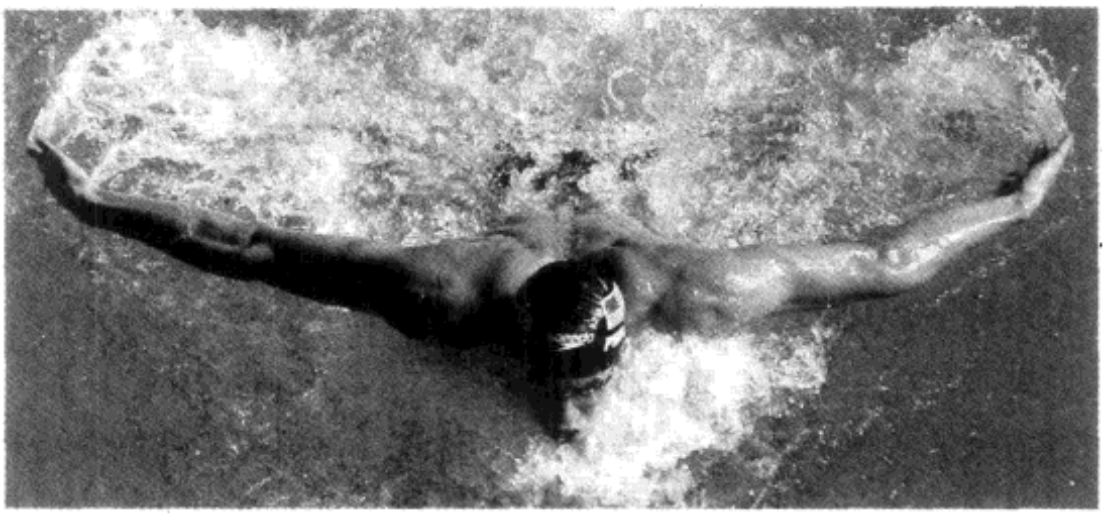


图 9-1 人类历史上获得奥运金牌最多的选手：美国密歇根大学的迈克·菲尔普斯

而所有这些评选的基础是（或者我们希望是）两个字：“竞争”，因为评优评奖的前提是有竞争者存在；否则就不存在评选的问题。当然，这个“竞争”是真的还是假的并不是本书要探讨的问题。本书关心的是在算法领域里施展的“评优”，即所谓的竞争分析！

竞争当然需要对手。算法里的竞争分析当然也是针对某个对手而言。到目前为止，我们所进行的算法分析是对特定算法本身，而没有将其与其他对手放在一起来判断优劣。也许你会说，我们在分析某一算法后不是经常会和别的算法比较吗？难道这不是竞争分析？

答案是既对又不对。

对，是因为我们确实会将两个算法的效率分析结果进行比较，并遴选较好的算法进行使用。从这一点上看，是有那么一点评优的味道在里面，也就说，有那么一点点竞争的意思。

那么如何评比才是真正的选优呢？当然是以最优的可能性或者“完美”为评判标准，并且达到这个最优标准的某一个程度（如百分比）才能被称为优秀。而这就是算法竞争分析的精髓。也许这就是神耶和华赐给以色列人一部无人能及的完美律法的原因吧。

换句话说，竞争分析是将一个算法和最优的算法进行比对，判断一个算法离最优算法的距离大小，并根据这个距离大小评判一个算法的竞争力。而只有达到一定竞争力的算法才能被称为优秀算法！

最优的算法是什么呢？或者说，我们怎么知道什么算法最优呢？如果有最优算法已经存在，我们还设计别的“不那么优”的算法呢？

要解答什么是最优算法，就得先解释什么是在线算法和离线算法。

9.2 在线算法和离线算法

读者有没有玩过一个叫“俄罗斯方块”（tetris）的游戏（见图 9-2）。在这个游戏中，一

块块不同形状的拼版从上面掉下，玩家需要将这些拼版尽量拼整齐，目标是让每一行都尽量填满，不留空格。而填满的行就会销掉，从而腾出空间来接收后面的拼版。如果玩家来不及消除足够的行数而导致拼版累积到框顶，那么游戏就宣告结束。而在结束前填满行数的多少（或计算出相关的结果）就是玩家的得分。

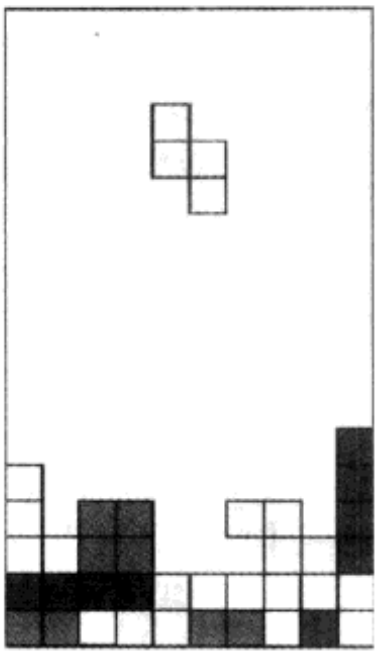


图 9-2 俄罗斯方块游戏

这个游戏的特点是拼版是一块一块地落下，玩家在拼一块拼版的时候，并不知道后面落下的一块拼版是什么形状。因此在拼的时候无法根据未来的情况来决定当前最优的拼法。也就是说，在每一块拼版出现的时候，我们需要对其进行立即处理，而不能等到后面的拼版出现后才处理。这种处理模式就是所谓的“在线处理”。即在线上的时候处理。如果一个算法也是如此，输入或请求是一个一个出现，但每个请求和输入必须立即得到处理，且在处理一个输入或请求的时候不知道后面的输入和请求，这种处理算法就是所谓的“在线算法”。

与此对应的是所谓的“离线算法”。这种算法可以事先就知道所有的输入和请求，这样就可以对所有的输入和请求进行最优的规划。注意，这里的“可以”是指理论上的。实际上也许是不可能的，例如输入数量巨大，我们根本看不出有什么最优的规划（找不出来）！这里重要的是离线和在线的比较优势：离线方式下，最优是可能的；而在线方式下，最优即使在理论上也是不可能的。你连后面的输入是什么都不知道，又如何能够进行最优规划呢？很多人将人生比喻成一个在线算法，从而判断人生无法活出最优！

由此可见，最优的算法必定是离线算法！而竞争分析就是将一个在线算法与离线算法进行比较来判断该在线算法的优秀程度如何。下面我们就给出在线算法和离线算法的准确定义。

定义 *满足如下条件的算法称为在线算法：*

1）*一次只提供一个操作序列 S 里面的一个操作。*

2）*对于每个操作，算法必须立即执行所需的操作（在不知道将来的操作是什么的情况下）。*

定义 *满足如下条件的算法称为离线算法：离线算法可以事先看到整个操作序列 S。*

在我们见过的算法里，有的只能是在线算法，有的只能是离线算法，而有的既可以是在线算法，也可以是离线算法。例如，俄罗斯方块游戏就具有在线算法的特点，交互式的象棋、围棋对弈也是一个在线问题（对手走了一步，你必须立即应付，而不能先知道对手后面要走的棋），而对一组数进行排序则具有离线算法的特性：所有的数都在一开始就提供给算法了。

操作系统里的磁盘调度则既可以设计为在线算法，也可以设计为离线算法。例如，我们既可以设计在线算法，对每一个到来的读写请求立即进行处理，而这种处理方式是在不知道下一个请求是什么的情况下进行。当然也可以设计一个离线算法，将一串磁盘读写请求在缓冲区缓存起来，然后一起调度。由于这种处理方式是在知道后面的请求是什么的情况下进

行，因此通常能够做得更优。

很多时候在线和离线是可以转化的。例如，只要我们有耐心等待，也许可以推迟对当前输入的反应，等待后续输入出现后才做处理，从而将一个在线问题转化为离线问题。而对于离线问题，你也可以只在看到一个输入的时候就进行处理。但要注意的是，这种转化通常会导致算法的效率或计算的结果偏离最优，因此，进行这种转化时必须非常小心。

9.3 竞争力

竞争分析针对的是在线算法，其分析目的就是将一个在线算法与一个最优的离线算法进行性能比较，并由此推导出该算法的竞争力（competitive ratio）。这个竞争力就是该在线算法的成本除以最优离线算法的成本。设最优的离线算法成本为 o，讨论中的在线算法成本为 x，则该在线算法的竞争力为 $CR=x/o$。

显然，竞争力取值越小，算法越优（似乎与平时的概念正好相反）。如果这个竞争力的取值是有限的（有上限），则该在线算法就被称为有竞争力的算法，即优秀算法。

定义 一个在线算法 A 如果满足下列条件，则称为 **α-竞争力的**：存在一个常数 k，对于任意一个操作序列 S，则有

$$C_A(S) \leqslant \alpha C_{OPT}(S) + k$$

这里的 OPT 是最优离线算法，也可以称为上帝的算法。

由此定义可见，竞争分析涉及摊销分析，因为我们在判断一个算法的竞争力时需要考虑一组操作（为什么），即算法的多次运行，而不是一个操作或算法的一次运行。

对于很多算法来说，性能并不单单取决于输入的大小，而且依赖于这些输入的具体取值。例如，在排序的时候，如果给出的序列已经排好序，则排序的时间复杂性与给出的序列是随机时的就会不同。这种对输入数据取值有依赖的算法必须进行平均和最坏情况分析。

9.4 健忘对手和优良对手

既然是竞争，自然就有对手。在竞争分析时，我们需要假定有一个对手，它故意给我们的算法选择“困难”（不好）的输入数据，从而尽可能地增大我们的竞争力，也就是使得我们的算法性能与某个最优离线算法的成本比值最大。对于确定性算法来说，任何对手都可以计算一个算法在将来任意时刻的状态，从而恰当地选择“困难”的数据来增大算法的成本。

对于随机化算法来说，竞争对手的能力可以不同：有的对手对于算法所做的随机选择并不知情，当然也就不能完全有针对性地增大算法的时空成本。这种对手被称为**健忘对手**，因为它不知道算法所做的随机选择。能力最强的对手则是所谓的优良对手或自适应对手。这种对手对算法的工作机制非常清楚，对算法执行过程中任意时刻的内部状态了如指掌，因此它可以尽可能地增加算法运行成本。

例如，对于快速排序算法来说，我们需要选择一个杠杆点（pivot），来对数据进行分

解。这个杠杆点的选择通常不会离数据序列的中值太远。快速排序根据这个杠杆点将数据分解为两个部分：一个部分数据小于杠杆点；另一部分的数据大于杠杆点。如果这个杠杆点的选择是确定性的（例如总是选择正中间的元素），那么对手可以很容易地构造一个数据序列，使得快速排序总是在最坏的情况下运行。如果杠杆点的选择是随机的，对于一个不知道这个随机数序列的对手来说，他将不能保证给出最坏的数据。

9.5 线性表更新问题

竞争分析里的一个经典算法是所谓的线性表更新问题：给定一个线性表和一组访问请求，如果访问位置靠前的元素比位置靠后的元素成本低，那么如何设计算法使得访问线性表的成本最低？当然，在每次访问结束后，允许算法对线性表进行重新排列。

显然，重新排列需要花费时间（就像我们重新整理书橱或者衣橱需要时间一样）。事实上，该算法设计的核心就是如何重新排列线性表，也就是线性表的更新。

那么怎样设计算法才能使其效率很高呢？如果想不出来答案，那就想想自己的衣橱吧（见图 9-3）！

图 9-3 线性表更新与衣橱的整理类似

相信大部分人家里都有衣橱。衣橱里的衣服按照某种顺序（当然也可以是随机的）一件件叠放着（一个线性表）。每天出门的时候，我们从这摞衣服里面取出一套穿上（访问线性表）。晚上回来后，再把衣服放回到衣橱（我们生活在一个水如明镜月如钩的洁净环境里，不用洗衣服）。在放回的时候，当然可以对衣服的叠放顺序进行重新安排（线性表更新）。

显然，拿上面的衣服比拿下面的衣服要省力。那么如何叠放衣服才能使得拿衣服的精力最省呢？读者一定想出了答案：将最常穿的衣服放在较上面的位置。

如果是一个离线算法，我们知道未来一生中每天穿的衣服是什么，这样我们可以计算出每套衣服被穿的次数，然后按照这个次数来进行衣服叠放，从而获得精力最省的排列。

但是我们当然不知道未来一生里，每天要穿的衣服是什么，那么怎么在不知道每天要穿的衣服是什么的情况下，设计出一个衣服的最佳叠放顺序呢？

对于很多人来说，一个直截了当的办法是：每天早上在衣服堆里面选取一套衣服。这套

衣服可能在衣服堆里面的任意位置。但是晚上回来的时候，我们多半会将衣服放在这一堆衣服的最顶上！因为这样做不需要重新整理衣服，就可以省去整理的开销。这就是我们要设计的在线算法！这个算法的名字就是**前置移动**（Move-To-Front，MTF）算法。

那么这个策略或算法的总成本会是最低的吗？

显然，直接估算这个算法的效率非常困难，因为没有具体的数据！当然，我们可以假定每天要穿的衣服是随机的，这样可以获得该算法的时间成本为 $n/2$。

这是最好的算法吗？如果是普通的算法分析，即非竞争分析，则这个问题没有太大意思，恐怕也不容易回答。但我们现在要做的是竞争分析，这个问题就问得十分恰当，因为我们要分析的是该算法的竞争力。而这种分析可以通过（只能是）与最优算法进行比较而获得。而这个最优的算法当然是一个已经知道我们一生要穿什么衣服的算法，即离线算法！

线性表的访问成本

为了方便分析，我们先定义一些术语。假定线性表 L 的大小为 n，即一共有 n 个元素。访问其中元素 x 的成本为 x 离开表头的距离，定义为 rank(x)。在每次访问后，可以将新近被访问的元素与其相邻的元素进行置换来重新安排表格。每次置换的成本为 1。

例如，对于图 9-4 的线性表，访问元素 42 的成本为 4，将 42 和 50 进行置换的成本为 1。

图 9-4 线性表 L 中访问元素 42 的成本为 4

最坏情况分析

对手总是访问线性表的最后一个元素，因此对于任何在线算法 A，其成本为：

$$C_A(S) = \Omega(|S|n)$$

这里，$|S|$为访问的次数，即对手一共进行了$|S|$次访问线性表操作。

平均情况分析

假定元素 x 被访问的概率为 $p(x)$，则该线性表的访问成本的期望值为：

$$E[C_A(S)] = \sum_{x \in L} p(x) \operatorname{rank}_L(x)$$

显然，在线性表的元素按照访问概率递减排列的情况下，该表达式取得最小值。

启发式线性表调整算法

通过上面的平均情况分析我们知道，如果能够知道每个元素被访问的概率，则可以按照该概率从高到低排列元素从而使得访问线性表的成本期望值最低。

但问题是，我们怎么能够事先知道一个元素被访问的概率呢？显然，我们并不知道。因此上述最优排列在实际上是不可能的。不过也无需失望，因为我们有一个近似的办法：记录每个元素被访问的次数，将表 L 按照元素被访问的次数从高到低排列。我们的思路是：一个元素被访问的次数越多，其被访问的概率应该也越大。这样，用次数来替换概率应该是一种

比较合理的接近最优的做法。这就是我们的启发式线性表调整算法。

不过，按照访问次数动态调整线性表里元素的排列是需要成本的。一种更为简单的启发式线性表调整算法是将最近被访问的元素调整到表的最前面，即前置移动算法。

9.6 前置移动算法的竞争分析

前面已经讲过，前置移动算法就是将新近被访问的数据放在线性表的最前面。虽然很多人在生活中一直在使用这种在线算法，但却没有意识到这个算法实际上是一个非常优秀的算法。事实上，我们可以证明，无论对手如何使坏，前置移动算法的成本不会超过最优算法的2倍，也就是说它的竞争力是2。因此，该算法的实际效果非常好。

在前置移动算法里，一个元素 x 被访问后将被移动到表的前面，而移动到前面的操作需要进行一系列的置换，其成本是需要置换的次数，它等于元素 x 的 rank，即

$$\text{移动成本} = 2\text{rank}_L(x)$$

我们称 MTF 算法是 $O(1)$-竞争力的。

定理 *线性表的 MTF 算法是 4-竞争力的。*

证明 设 L_i 为在 MTF 算法下，第 i 次访问后的线性表状态；L_i^* 为最优（离线）算法下，第 i 次访问后的线性表状态。又设 c_i=MTF 算法第 i 次访问的成本，如果它访问的是元素 x，则 $c_i = 2\text{rank}_{L_{i-1}}(x)$；$c_i^*$ =最优算法第 i 次访问的成本，如果它访问的是元素 x，则 $c_i^* = \text{rank}_{L_{i-1}^*}(x) + t_i$，这里，$t_i$ 为最优算法 OPT 执行的置换次数。

那么到底如何进行竞争分析呢？前面说过，竞争分析是相对某个最优算法进行，因此，一种自然的想法是将 MTF 算法与 OPT 算法的每一步进行比较，看看每一步之后 MTF 算法导致的线性表的变化是趋向更优还是更坏，然后将这些变化累计就可以得出两个算法的比较态势。

用什么分析方法才能对 MTF 算法和 OPT 算法每个步骤的优劣进行比较呢？从上段文字的解说中，读者应该已经猜出来了——势能分析！因为我们要在任意一次访问后，对 OPT 算法和 MTF 算法下线性表的状态进行比较。因此，我们需要定义一个势能函数，这个势能函数需要将 MTF 算法和 OPT 两个算法联系起来，或者说，能够将两个算法的差异体现出来。这样，势能函数值的累积就是 MTF 算法和 OPT 算法差异的累积。

用什么方式体现两个算法的差异呢？对于 L 和 L^*两个表来说，它们包含的元素是一样的，因此，差异只能在元素的排列顺序上！由于最优算法的每一步都是最优的，因此每一步后，线性表的状态也应该是最优的，即每次访问后元素顺序的排列都是最优的！这样，在 MTF 算法下，每次访问后线性表的元素排列顺序可以和 OPT 算法下的线性表元素顺序进行比较，与此顺序不同的程度就是它与 OPT 算法的差异，而这个差异将导致算法效率的差异。

综上所述，我们定义势能函数 Φ：$\{L_i\} \to$R（实数集）如下：

$$\Phi(L_i) = 2 \times (L \text{ 和 } L^* \text{里面的反序对数}) = 2 \times |\{(x, y) : x <_{L_i} y \text{ 且 } y <_{L_i^*} x\}|$$

这里，$x<_{L_i}y$ 表示元素 x 在表 L_i 里面的位置在元素 y 前面，$y<_{L_i^*}x$ 表示元素 y 在表 L_i* 里面的位置在元素 x 前面。

例如，如果 MTF 算法和 OPT 算法在第 i 步操作后，其线性表的状态分别如图 9-5 所示。

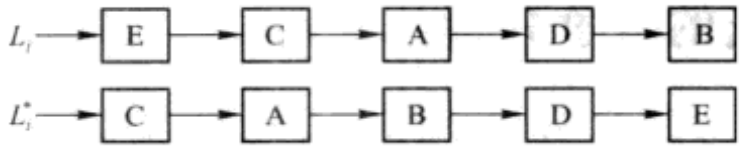


图 9-5 线性表中正序和反序

则有：

$$\Phi(L_i) = 2 \times |\{(E,C),(E,A),(E,D),(E,B),(D,B)\}| = 10$$

注意 $\Phi(L_i) \geqslant 0(i = 0, 1, \ldots)$，并且，如果 MTF 算法和 OPT 算法从同一个表开始，则 $\Phi(L_0) = 0$。

有了上述势能函数定义，我们下面要做的分析就是判断 MTF 算法在对线性表进行访问而导致线性表更新的时候对势能函数的改变。那么对线性表的更新会造成势能函数的何种变化呢？例如，如果只进行一次置换（两个元素之间顺序的反转），Φ 的变化是多少？

显然，相对于 OPT 算法的线性表 L^*来说，MTF 算法每进行一次置换要么增加一个同序，要么增加一个反序，因此 $\Delta\Phi = \pm 2$。

我们知道，每发生一次访问，MTF 算法将进行 $\text{rank}_{L_{i-1}}(x)$次元素置换。那么这么多次置换会导致势能函数发生何种变化呢？要想知道这个答案，我们需要对 MTF 算法和 OPT 算法分别对应的线性表里的元素进行分类，而这个分类依被访问的元素 x 进行。

假如操作 i 访问的元素为 x，我们定义下面 4 个元素集合：

- $A=\{y \in L_{i-1}: y <_{L_{i-1}} x \text{ 且 } y <_{L_{i-1}^*} x\}$，即两个算法表中都在元素 x 前面的元素集合。
- $B=\{y \in L_{i-1}: y <_{L_{i-1}} x \text{ 且 } y >_{L_{i-1}^*} x\}$，即 MTF 算法表中在 x 前面，OPT 算法中在 x 后面的元素集合。
- $C=\{y \in L_{i-1}: y >_{L_{i-1}} x \text{ 且 } y <_{L_{i-1}^*} x\}$，即 MTF 算法表中在 x 后面，OPT 算法中在 x 前面的元素集合。
- $D=\{y \in L_{i-1}: y >_{L_{i-1}} x \text{ 且 } y >_{L_{i-1}^*} x\}$，即在两个算法表中都在元素 x 后面的元素集合。这些集合和两个算法所对应的线性表之间的关系如图 9-6 所示。

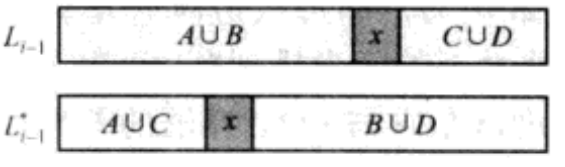


图 9-6 线性表中元素位置分类

显然，元素 x 在 MTF 算法的线性表中的排名 r 是：

$$r = \text{rank}_{L_{i-1}}(x) = |A| + |B| + 1$$

元素 x 在 OPT 算法的线性表中的排名 r^*是：

$$r^* = \text{rank}_{L_{i-1}^*}(x) = |A| + |C| + 1$$

当 MTF 将元素 x 移动到表的最前面时，它将新增$|A|$个反序，减少$|B|$个反序，而 OPT 算法每次置换新增反序的对数≤1。因此，有：

$$\Phi(L_i) - \Phi(L_{i-1}) \leqslant 2(|A| - |B| + t_i)$$

相对于势能函数 Φ，MTF 算法的第 i 次操作的摊销成本为：

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(L_i) - \Phi(L_{i-1}) \\
&\leqslant 2r + 2(|A| - |B| + t_i) \\
&= 2r + 2(|A| - (r - 1 - |A|) + t_i) \qquad (\text{因为 } r = |A| + |B| + 1) \\
&= 2r + 4|A| - 2r + 2 + 2t_i \\
&= 4|A| + 2 + 2t_i \\
&\leqslant 4(r^* + t_i) \qquad (\text{因为 } r^* = |A| + |C| + 1 \geqslant |A| + 1) \\
&= 4\, c_i^*
\end{aligned}$$

因此，有：

$$\begin{aligned}
C_{\text{MTF}}(S) &= \sum_{i=1}^{|S|} c_i = \sum_{i=1}^{|S|} (\hat{c}_i + \Phi(L_{i-1}) - \Phi(L_i)) \\
&\leqslant \left(\sum_{i=1}^{|S|} 4c_i^* = \right) + \Phi(L_0) - \Phi(L_{|S|}) \\
&\leqslant 4C_{\text{OPT}}(S) \qquad (\text{因为 } \Phi(L_0) = 0 \text{并且} \Phi(L_i) \geqslant 0)
\end{aligned}$$

因此，线性表的 MTF 算法是 4-竞争力的。 □

推论 如果我们把元素 x 移动到表头的成本记为 0，即我们能够用常数时间将 x 从 L 里切割出来附加在头上（这在链表的情况下可以实现），则 MTF 算法就是 2-竞争力的。

一点思考

一切似乎已经尘埃落定。但还有一个情况：如果 $L_0 \neq L_0^*$怎么办，即 MTF 算法和 OPT 算法的开始表格不一样会有什么影响？这当然有点钻牛角尖，但也不是完全凭空臆想。毕竟我们可以在算法运行的任何时刻开始对算法来进行比较！

但即使如此，也不用担心。如果是这样，则$\Phi(L_0)$在最坏情况下可以是$\Theta(n^2)$，因此，$C_{\text{MTF}}(S) \leqslant 4C_{\text{OPT}}(S) + \Theta(n^2)$。但这仍然是 4-竞争力的，因为当$|S| \to \infty$的时候，$n^2$ 是一个常数。

9.7 聚类问题

竞争分析除了用于对算法的时空成本进行分析外，也可以用于对算法的结果（即算法找

出的解）进行分析。此时，我们需要做的事情是将待分析的算法计算的解和最优解进行比较，以获得相应算法的竞争力。下面我们以聚类问题来展示竞争分析在分析算法结果上的应用。

聚类（clustering）是数据挖掘里经常用到的一种方法，其要解决的问题是在一个空间的所有数据点上划分出给定个数的区间，使得每个区间里最远两点的距离最小。这里的距离有可能是某高维空间里真实的点与点之间的欧几里得距离，也有可能是进行某种“近似度或相似度”测量后得出的距离。一个区间里面最远两点的距离也称为该区间的直径。

所有这些距离满足下列 4 条公理：

1）$\forall x,y, d(x, y) \geqslant 0$，距离不能为负。

2）$d(x, y) = 0$ 当且仅当 $x = y$，只有到自身的距离才能为 0。

3）$d(x, y) = d(y, x)$，x 到 y 的距离与 y 到 x 的距离相同。

4）$d(x, y) \leqslant d(x,z) + d(z,y)$，三角不等式。

聚类问题定义 设某空间 X 有 n 个数据点，即 $X=\{x_1,x_2,\cdots,x_n\}$，并有距离度量函数 d: $X\times X\rightarrow R$。要求将空间 X 划分成 k 个区间 $C_1,C_2,\cdots,C_k$，其中 C_m 的直径为：

$$d_m = \max_{x_i,x_j \in c_m} \{d(x_i, x_j)\}$$

使得所有区间的直径 $\max_{1\leqslant m\leqslant k} d_m$ 最小。直观地说，要使数据点尽可能“聚集”在所属区间中。

由于聚类问题是一个难解的问题（本书第五部分将讨论难解问题），直接找最优解十分困难。因此，我们退一步求其次，寻找一个次优解。但需要对次优解的质量进行分析，以确保其质量在可以接受的范围内。

9.7.1 聚类问题的次优解算法

聚类问题的次优解算法（CLUSTERING-ALGORITHM (X)）：

1）首先随机选择一个点 μ_1 作为第 1 个区间 C_1 的中心。

2）接着按下面的规则依次选择 $\mu_2, \mu_3,\cdots,\mu_k$ 分别作为区间 $C_2,C_3,\cdots,C_k$ 的中心：对于 $2\leqslant m\leqslant k$，μ_m 应该是离 $\mu_1, \mu_2,\cdots,\mu_{m-1}$ 距离最远的点。

3）把每个点指派给离它最近的区间，即点 x 所属区间为 C_y，这里 $y=\min\{d(x, \mu_y)\}$。

很显然，CLUSTERING-ALGORITHM 算法确实将空间 X 划分为了 k 个区间，但问题是这种划分是否达到每个区间直径最小化了呢？或者说，CLUSTERING-ALGORITHM 算法所计算出的划分的质量如何呢？要回答这个问题，就需要进行竞争分析。

9.7.2 CLUSTERING-ALGORITHM 算法的竞争分析

设 x 为离我们选出的 k 个中心点 $\mu_1, \mu_2,\cdots, \mu_k$ 最远的点，设 r 为 x 与离其最近的中心点的距离。则空间 X 里的每个点与其最近的中心点的距离都不会超过 r。根据三角不等式公理，每个区间的直径不会超过 $2r$。

这个直径与最优划分的直径相比质量如何呢？下面我们来推导该计算结果与最优结果的近似比。

算法共生成 k 个区间，这 k 个区间的直径上界为 $2r$。接下来考察顶点$\{\mu_1, \mu_2, \cdots, \mu_k, x\}$。根据 x 的选择方式，它们中任意两者之间的距离至少为 r。而一个最优解需要将这 $k+1$ 个顶点置于 k 个区间里，这将导致至少有一个区间包含这 $k+1$ 个顶点里面的两个顶点。这样，该区间的直径将不可能小于 r。因此，对任意最优解，其每个区间的直径下界为 r。

由于 CLUSTERING-ALGORITHM 算法计算出的区间最大直径为 $2r$，而最优解的区间最大直径不小于 r，因此 CLUSTERING-ALGORITHM 算法计算出的解不差于最优算法计算出的解的 2 倍，即我们的算法具有 2 倍竞争力。

9.8 竞争分析与普通算法分析

竞争分析是将一个算法与理论上能够达到的最优算法进行比较，获得其与最优算法之间的差距。而离最优算法的差距越小，算法当然也就越优。这种竞争分析的着眼点是一个算法和最优算法的成本或计算结果比率，而不是该算法的时间成本或计算结果的具体取值。因此，从这个角度来说，竞争分析比普通的针对一个算法本身效率的数值进行评价来说似乎要简单一些。

不过从另一个角度来看，竞争分析不一定比普通的算法分析简单。因为竞争分析不仅要考虑待分析的算法本身，还需要知道完成同样任务的最优算法是如何表现的，即实际上要考察两个算法。这样说来，与普通的算法效率分析相比，竞争分析似乎更为困难。

当然，是否困难完全取决于分析者的具体情况。

思考题

1. 竞争分析是否可以应用到离线算法上？如果可以，如何应用？如果不可以，给出理由。
2. 竞争分析为什么要对一组操作或算法的多次运行来进行？可否只针对算法的一次运行进行竞争分析？为什么？
3. 我们可否对一个并不涉及持久数据结构的算法进行竞争分析？给出理由。
4. 世界上一些无聊的机构经常发布所谓的各国竞争力排名，而很多国家也把这些排名当成一回事情在对待。请针对任意一份这样的报告，从竞争分析的视角对其进行分析来判断此种报告的说服力（可以只针对某一个方面的竞争力）。
5. 现代考试常出现的题型是所谓的单选题，即在多个选项里选择一个正确答案（准确地说，应该是选择一个最好的答案）。假定某次考试一共有 n 道考题，每道题有 m 个选项。选对一题得 $k>0$ 分，选错或多选得 0 分。假定你不知道任何一道题的答案，于是你选用随机化策略来进行答题，即每次从 m 个选项中随机选择一个。请对此算法进行竞争分析。
6. 你是某个领域的“专家”，被要求对一些项目申请报告进行评估。一共有 n 份项目申请报

告，你需要选择其中的 m 份。请问，如果你是一个“伪专家”，你如何进行此种选择？如果你是一位真专家，你又如何进行筛选？（提示：使用竞争分析。）

7. 设有无向图 $G=(V, E)$，要求计算出一个顶点集合的最小子集 $S\subseteq V$，使得所有边的至少一个端点落在 S 里。由于该问题的准确解难以获得，因此我们设计了一个近似算法：

   VERTEX-COVER-APPROXIMATION

   - 首先设置空集 M。
   - 依次考虑每条边 $e\in E$。
   - 如果 e 不与 M 里的边共享端点，则将 e 加入 M。
   - 在上述算法结束后，将 M 里的所有结点作为找到的顶点集。

   请对上述算法找出的顶点集的质量进行竞争分析。

8. 由于奥林匹克数学（简称奥数）竞赛成绩对升学的影响日渐扩大，中国的父母不得不费尽心思让自己的儿女参加各种奥数补习班。某父亲决定让自己的儿子参加一个每周一次的奥数班。但由于儿子自己对此并无兴趣，随时有撂挑子的可能，因此，每一周都可能是最后一次上奥数班，即整个奥数课程将在一个未知的第k周结束。问题是父亲需要为儿子支付学费。学费可以一次性支付，也可以一周一周地支付，直到儿子退出奥数班为止。如果一次性支付，总成本为A元；如果每周支付，则每次R元。在学习的中途仍可以选择一次性支付，但前面已经支付的周成本不计在内，即如果中途一次性支付，将在已经支付的周成本基础上，再支付A元。请设计一个2倍竞争力的算法，即你所设计的算法所需支付的总成本不超过最优算法所需成本的2倍。

9. 由于你忽悠水平高超，被同行吹捧为“专家”，并被邀请参加一次重大项目评审。参加此次重大评审的项目报告一共有 n 份，你需要从中挑选 m 份作为中标报告。但问题是，你其实并不是什么专家，对项目所属领域并不清楚。不过幸运的是，你的算法修为还不错！请利用算法的知识设计一个最为简单且可行的项目评审算法。详细说明你的算法思路。

10. 你觉得竞争分析能够在你与别人的竞争中发挥帮助作用吗？为什么？

11. 这是本书第 4 章出现过的思考题。一天你在街上行走的时候，需要从左边马路走到右边马路上，而一路上的十字路口有 n 个。你可用在任意一个十字路口穿越马路，前提当然是在绿灯的时候（不然很危险！当然，考虑到国内驾车族的素质，就算是绿灯横穿马路也不一定安全）。如果横穿信号为红灯，则需要等待。每个十字路口的信号灯的红绿时间长度不同，设绿灯每亮 1 分钟，红灯亮的时间长度为 h_i 分钟（i=1,2,…,n），这里 n 是十字路口数。而从路口 i 步行到下一个路口 i+1 需要时间 b_i。请设计一个最有竞争力的算法，并计算该算法的竞争力，并分析算法的复杂性。

12. 确定性算法无法保证公平性，而随机算法能保证一定的公平性。例如，遗传算法需要对种群进行选择，通过选择的个体才会保留，而无法通过选择的个体则被舍弃，在这个过程中，需要用到一定的随机性，使得每个个体都有成活的可能，也有被舍弃的可能。如果用确定性算法，那么会导致只有适应值高的元素才会被保留……同样，在现实世界中，也有很多这样的问题，比如大学录取时，只有过线的学生才被录取，而未过线的学生只有被舍弃的命运；评奖学金时，永远只有评价分数最高的人才评得上，而表现也不错的，却可能因一分之差而失去奖金学的机会，这二者之间可能并没有什么区别。

13. 给定一组元素，其中每个元素都有一个权重，请设计一个随机算法，给出该组元素的一个随机排列，该排列应尽量体现这个权重。即虽然该算法每次运行得到的排列都不尽相同（或者说并不确定），但如果运次的次数无穷多，那么统计下来，权重大的元素出现在排列靠前位置的几率也就相应较大。

（a）设计并实现该算法。

（b）为该算法设计竞争力标准。

（c）根据（b）的结果对该算法进行竞争分析。

PDG

PART FOUR

第四篇 经典算法篇

PDG

CHAPTER 10

第 10 章　排序与次序

某国家为了向世界展示其所谓的“强大国力”，决定举办盛大的海陆空三军立体阅兵式，并邀请世界各国元首和政要来观摩。但在打印观看阅兵式的嘉宾名单时却发了愁：这么多总统、国王、酋长、公主、王储、总理、部长等，他们的名字按什么顺序排列才最妥当呢？

要知道，这个排名至关重要。谁的名字在前，谁的名字在后是有着外交影响的。因此，如何排名就是一个很重要的外交任务了。

再说，“阅兵式”本身也需要对受阅人员进行某种排列，总不能大家一窝蜂地拥挤向前吧？例如，在进行海上分列式时，总需要对各艘舰艇进行某种前后排列（见图 10-1）。

图 10-1　阅兵式的海上分列式：哪艘舰艇排在最前呢

怎么排名才能让各位嘉宾满意呢？

当然是按国家重要程度、官阶大小、与东道主密切程度进行排列！官阶高的必须排在官阶低的前面，重要国家的人物排在次要国家的人物前面，与东道主关系密切的排在与东道主关系不太密切的前面，而在满足上述条件的情况下则可任意排列。

10.1　排序无处不在

虽然我们被告知人人生来平等，但在生活中却总是要排在某些人后面。上学时有成绩排

名，大学有大学排名，据说中学也有排名，领导出场则有先后次序，甚至连宇宙都存在某种秩序。正如人们经常所说“乾坤有序，万物有律”。有些秩序的存在为必然，有些则为偶然。有的秩序无法变更，有的则可以打破重来。在很多时候，人们只能被动地接受各种物理和社会伦理上面的秩序。要在社会上生活得顺利，知道自己的排序和位置是非常重要的。从某种程度上说，知道自己的定位就是人生最重要的事情。这也是华夏始祖伏羲绘制八卦的首要动机。要知道自己的定位或者位置，就得知道如何排序。

既然排序在生活中如此重要，它在算法里面成为一个重要问题也就不足为怪了。

将排序问题抽象出来，就是将一个数列进行改变而使得数列的所有元素以非递增或非递减方式排列，如果以输入和输出来进行定义，我们有这样的定义：排序问题就是给定一个数列作为输入，而输出则是数列的一个非递减或非递增序列，例如：

- **输入**：一个数的序列$\langle a_1, a_2, \cdots, a_n\rangle$
- **输出**：序列$\langle a_1, a_2, \cdots, a_n\rangle$的一个排列$\langle a'_1, a'_2, \cdots, a'_n\rangle$，且满足条件：$a'_1 \leqslant a'_2 \leqslant \cdots \leqslant a'_n$

这里需要提请读者注意两点：

1）非递增序列就是我们平时所说的“降序”，但降序在语义上存在模糊。如果有两个或多个元素的取值相同，则降序的说法将不成立。因此我们使用更加精确的术语“非递增序列”。非递减序列就是我们所熟知的“升序”。同理，由于升序在表述上的不严谨，本书使用语义上更加精确的“非递减序列”来描述类似升序的一个序列。

2）所有序列都可以转化为数列，因为从计算机的角度看，所有的东西从根本上都是数字。

例如，给定数列8 2 4 9 3 6，按照非递减次序排好序的数列为2 3 4 6 8 9。那么我们如何获得这个排序的结果呢？

10.2 插入排序

对于很多人来说，排序似乎是件很简单的事情。不是吗？我们多数人都玩过扑克，而玩扑克的时候我们经常需要将手上的牌进行排序。我们是如何进行排序的呢？如果读者用心回想自己抓牌的过程，就会发现，在任何时刻，一手牌都可以分成两个部分：一部分已经抓来握在手上，另一部分还未抓，尚在牌桌上的牌堆里。在一开始，手上的牌为空，所有的牌都在牌桌上。而每抓一张牌，就将新抓来的牌插入到手中牌里的合适位置上，使手上的所有纸牌保持某个秩序。当所有的牌都抓完时，所有的牌都已经抓在手上，而牌桌上的牌堆里已没有纸牌。至此，一手牌的次序也就完成了（这里讨论的是大部分人在打“争上游”时的持牌方式）。

如果注意分析上述抓牌和理牌过程，读者可以发现，在任何时候，手上的牌为有序的，而牌桌上的牌为无序的。整个抓牌过程使有序部分不断增大，无序部分不断缩小，直到无序部分为空为止。这种不断从无序部分抽取一个元素（一张牌），并将其**插入**到有序部分合适位置上的排序方法就是有名的**插入排序**。

从上述讨论可知，插入排序的特点就是将待排序的元素一个个插入到已经排好序的数列里去，而这涉及插入位置的确定。显然，要确定一个元素的插入位置，需要将待排序的元素

与已排序好的元素进行比较。最简单的比较方式当然是一个一个元素地进行。这样，我们获得插入排序的算法（伪代码程序）如下：

```
INSERTION-SORT (A, n)                    //A[1..n]为包括 n 个元素的数组
for (j =2; j<=n; j++) {
     key =A[j];
     i =j-1;
     while (i >0 &&A[i] >key) {
            A[i+1] =A[i];
            i =i -1;
            }
     A[i+1] = key;
}
```

在上述伪代码表述的插入排序算法里，数组 *A* 里是欲待排序的元素，而排好序的元素还是存放在数组 *A* 里。这种在排序过程中不使用临时存储空间，即排序前和排序后元素都存在同一个地方的排序称为**原地排序**（in-place sort）。

细心的读者也许已经发现，该伪代码程序使用了一个小技巧，即排好序的部分并不是从空开始，而是从 1 个元素开始（即第 1 号元素）。这是因为 1 个元素的序列永远是排好序的序列！因此，一开始，*A*[1..1]部分就是已经排好序的部分，而需要插入的元素不过是 *A*[2..*n*]。

新元素与有序部分元素的比较是从后面开始，一个个往前比较，直到找到一个比新元素小的元素或者有序部分的元素已经比较完（数组已经到尽头）。此时将新元素插入到刚才找到的比它小的元素后面（或数组的首位置）。图 10-2 描述的是该算法的表达不变式和元素的比较顺序。

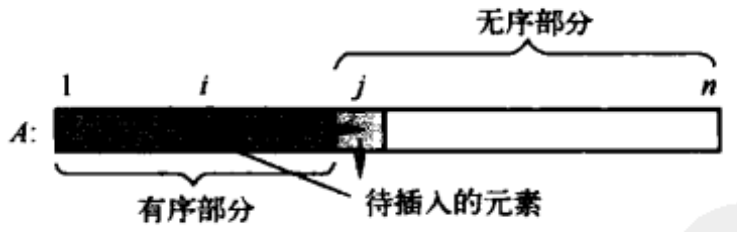


图 10-2　插入排序示意图

下面的例子展示了插入排序的整个过程（黑色为已经排好序的部分）：

8 2　4　9　3　6　//开始状态，第 1 个元素 8 是唯一排好序的部分，即有序部分。

2 8　4　9　3　6　//第 1 步，元素 2 插入在元素 8 前面，有序部分增加到 2 个元素。

2 4　8　9　3　6　//第 2 步，元素 4 插入在元素 8 前面，有序部分增加到 3 个元素。

2 4　8　9　3　6　//第 3 步，元素 9 位置不变，有序部分增加到 4 个元素。

2 3 4　**8**　**9**　6　//第 4 步，元素 3 插入在元素 4 前面，有序部分增加到 5 个元素。

2 3　4　6　8　9　//第 5 步，元素 6 插入在元素 8 前面，有序部分增加到 6 个元素。此时，无序部分为空，排序完毕。

插入排序的过程非常直观，排序的原理也非常简单，并且正确性也很容易证明（使用表

达不变式即可证明，本书在此略去）。但是排序的效率如何呢？

10.2.1 插入排序的效率分析

从伪代码程序来看，插入排序有两层嵌套循环，而其整个成本由总循环的次数所决定。外层循环的次数是固定的，为 $n-1$ 次，而内层循环的次数则有赖于元素之间的相对关系。

在最坏的情况下，内层循环的次数为 i 次，此种情况在输入元素的次序与要排的次序正好相反时出现。在此情况下，循环的总次数为：

$$T(n) = 1+2 + 3 + \cdots + (n-1) = \sum_{j=2}^{n} \Theta(j) = (n-1)n/2 = \Theta(n^2)$$

在最好的情况下，每次内层循环的次数为 0，此种情况在输入元素的次序与最后次序正好一样时出现。在此种情况下，总循环次数为：

$$T(n) = 1 + 1 + 1 + \cdots + 1 = \sum_{j=2}^{n} \Theta(1) = n - 1 = \Theta(n)$$

在平均情况下，输入序列的各种排列次序都可能出现。在没有任何根据和附加信息的情况下，我们只能假定各种次序排列出现的概率相等。这样，内层循环的次数平均为 $i/2$ 次。因此，插入排序循环的总次数为：

$$T(n) = 1/2 + 2/2 + 3/2 + \cdots + (n-1)/2 = \sum_{j=2}^{n} \Theta(j/2) = (n-1)n/4 = \Theta(n^2)$$

因此，插入排序的时间复杂性在最坏和平均情况下均为 $\Theta(n^2)$，而在最好情况下为 $\Theta(n)$。由于排序的时间复杂性不可能优于线性（至少要将每个元素看一遍，而这已经需要线性时间），因此，最好情况下的插入排序是最优的排序。但是其平均情况的效率却不容乐观。因为，平方级 $\Theta(n^2)$的时间复杂性相当于每个元素和每个其他的元素都比较了一次。显然，一个元素在和所有其他元素都比较后，其在整个序列里面的位置次序当然就确定了。因此，平方级算法是最笨的办法都能达到的。基于此认识，插入排序应该不是什么优秀的算法。

那么有没有比平方级算法更优的平均情况下的算法呢？

10.2.2 折半插入排序

要回答上述问题，需要检查插入排序的特点：每个元素和每个其他元素都比较了一次（虽然实际上这不一定是事实，但其算法复杂性与此种最笨的办法相同）。难道真的需要将每个元素和序列里其他每个元素都比较一次吗？答案是否定的。

由于序列的前面部分为已经排好序的，因此新元素在插入过程中与有序部分的比较可以折半进行，即先与有序部分中间的元素比较，如果新元素与中间元素一样大，则插入位置就在该中间元素之后。否则需要考虑两种情况：新元素比中间元素大，则将比较范围缩小为中间元素到有序部分最后的元素区域；新元素比中间元素小，则将比较范围缩小到有序部分的

起始元素到中间元素的区域。然后按照上述过程重复进行，直到找到排序位置为止。

这种在插入位置寻找过程中使用折半搜索的插入排序就称为**折半插入排序**。

显然，在折半插入排序算法里，每次插入新元素所需要比较的次数在最坏情况下就不再是 i 次，而是 log$i$ 次，这是我们内层循环的次数。这样，插入排序的总成本为：

$$T(n) = \log 1 + \log 2 + \log 3 + \cdots + \log(n-1)$$

$$= \sum_{j=2}^{n} \Theta(\log j) \leqslant \sum_{j=2}^{n} \Theta(\log n) = \Theta(n \log n)$$

这实在是太好了，经过一个小小的改进，插入排序的时间复杂性从平方级 $\Theta(n^2)$降低到**线性对数级** $\Theta(n\log n)$。而且这个改进的效果十分明显。本书第 2 章列出的各种排序算法比较（图 2-4）充分显示了 $\Theta(n\log n)$数量级相对 $\Theta(n^2)$数量级的优越性。

10.3　归并排序

虽然对插入排序进行改善而得来的折半插入排序的时间复杂性为线性对数级 $\Theta(n\log n)$，但这并不是唯一获得线性对数级时间效率的算法。如果仔细分析排序的要求，我们发现排序问题非常适合使用分治策略来解决：将一个大序列的排序问题分解为对两个或多个子序列的排序问题，然后对子序列递归使用同样的方式进行排序，在子序列排好序后，将结果合并起来即可。利用这种想法就获得了算法中另一个著名的排序方法：归并排序（merge sort）。之所以称之为归并排序，是因为整个算法的成本由归并部分决定。

归并排序算法（伪代码程序实现）如下：

MERGE-SORT $A[1 \,..\, n]$

1）如果 $n = 1$，排序完成。

2）将序列分解为两个子序列 $A[1 \,..\, (n/2)]$和 $A[(n/2)+1 \,..\, n]$。

3）递归对子序列 $A[1 \,..\, (n/2)]$和 $A[(n/2)+1 \,..\, n]$进行排序。

4）将排好序的两个子序列进行归并（合并）。

归并排序使用的是分治策略，而分治策略的关键是分解和合并。归并排序的分解部分比较简单，直接将序列从中间斩断。但归并的时候需要小心将两个排好序的子序列按次序进行合适的穿插，从而获得原序列的一个排序。显然，对于数组的数据结构来说，从中间斩断只需要常数时间，因此，分解步骤的时间复杂性非常琐细。但合并的时间复杂性为多少呢？

我们先来看一个归并的例子。图 10-3 给出的是 A、B 两个序列合并的过程：

合并时候我们保持两个指针：一个指向 A 序列里下一个欲待归并的元素，另一个指向 B 序列里下一个欲待归并的元素。每步我们都对这两个指针指向的元素（图中阴影标记的元素）进行比较，小的将被归并到 C 序列里，并且指向较小的指针往后推进一个元素，另一个指针则维持不动。这样，循环往复，直到一个子序列为空为止。此时，另一个子序列剩下的所有元素直接拷贝到 C 序列末尾即可。

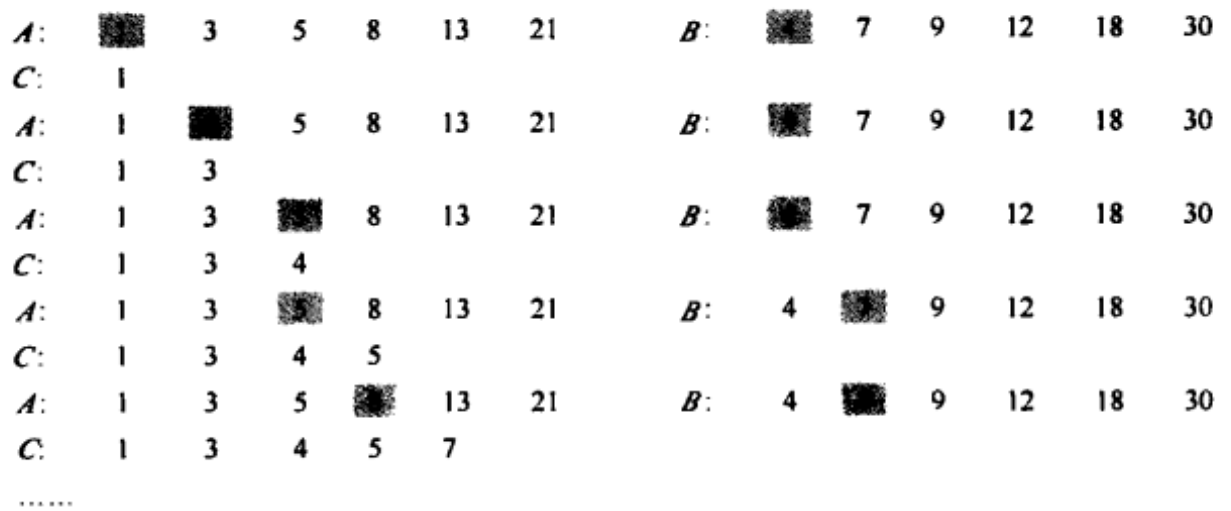


图 10-3 归并排序过程演示

显然，在其中一个子序列变为空之前，每次比较都将归并一个且仅一个元素；而在一个子序列变为空后，我们可以一次归并另一个非空子序列的所有元素。这样，最好的情况是其中一个子序列的最大元素不大于另一个子序列的最小元素，此时的总比较次数为 $n/2$。最坏的情况为两个子序列元素的大小形成穿插态势，此时总比较次数为 n。而这两种情况的渐近时间复杂性完全一样，均为 $\Theta(n)$。即合并时间为线性！

这样，我们获得归并排序每步的时间成本如下：

1）分解（divide）：简单、直接，时间为常数级。

2）治之（conquer）：递归对两个子序列进行排序，时间为 $2T(n/2)$。

3）合并（merge）：将排好序的两个子序列进行穿插合并，时间为线性级。

由此可见，我们可以将归并排序的时间复杂性表示为 $T(n) = 2T(n/2) + \Theta(n)$ 。这个表达式也可以从前面的伪代码表示上获得：第 1 步和第 2 步两步的时间复杂性为 $O(1)$，第 3 步的时间复杂性为 $2T(n/2)$，第 4 步的时间复杂性为 $\Theta(n)$。当然，递归表达式里的 $2T(n/2)$项写得有点儿马虎，精确的表示应该是：

$$T[\text{ceiling}(n/2)] + T[\text{flooring}(n/2)] + \Theta(n)$$

但这种更为复杂的表示在结果上与我们的马虎表示完全一样（在渐近趋势上）。因此，我们使用更为简单的：

$$T(n) = 2T(n/2) + \Theta(n)$$

根据大师解法可知，上述递归表达式的解为 $T(n) = \Theta(n\log n)$，即线性对数级。

如果画一棵递归树，归并排序的时间复杂性就看得更加清楚了，如图 10-4 所示。

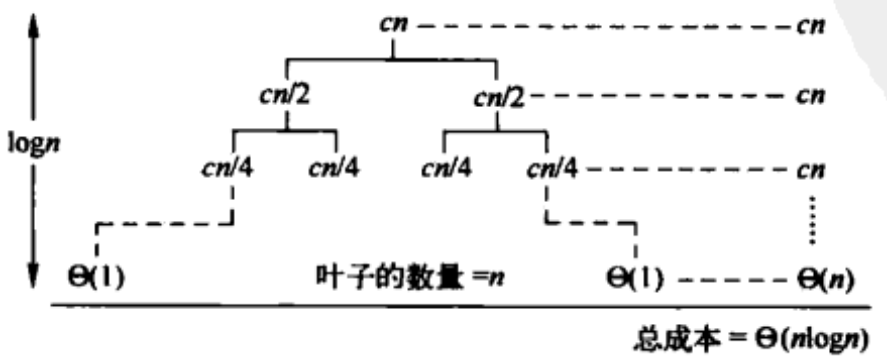


图 10-4 归并排序的递归树

实验表明，归并排序在 $n>30$ 后表现出比（标准）插入排序更加优越的性能。

这里提请读者注意的是，归并排序的 $\Theta(n\log n)$时间复杂性适用于所有情况：最好、最坏或平均。这是归并排序的一个显著特点——一视同仁。

10.4　快速排序

归并排序在理念上非常直截了当：将要排序的序列一分为二，分别对分开的两个长度相等的子序列进行归并排序。在子序列排好序后，对它们进行归并（合并）。在这种方法下，分解部分微不足道，直接从中间砍断；复杂性的关键在于归并。因此，它被称为归并排序。

仔细思索可以发现，归并排序的特点是前半部分不费力气，快刀斩乱麻，一下子就分到极致。但在这大快之后，却要花工夫在归并上。这是典型的“先享受后付出代价”的思维方式。但作为人来说，总觉得“先享受后付出代价”的方式有点儿不对。

另外，归并排序还有一个缺陷：它是一个所谓的“异地排序”（out of place sort）算法，即在排序的过程中需要使用额外的存储空间（如图 10-3 中的 C 序列）。

为了节省存储空间，更为了很多人崇尚的另外一种方式——先付出后享受劳动成果，人们发明了快速排序。此种排序的思路是，如果在分开的时候，不是从中间位置上分界，而是按照元素的大小分开为两个一大一小的子序列（一个子序列的所有元素大于另一个子序列里的所有元素），这样的话，因为两个子序列之间的相对次序已经正确，所以在合并的时候就不需要花费任何时间。此种排序算法由于合并时间变得微不足道而称为**快速排序**（quicksort）（因为它让人感觉很快）。

快速排序由 C.A.R. Hoare 在 1962 年提出。显然，它是一种分治算法，并且是原地排序算法（这一点我们将马上看到）。这种排序一经提出，就获得了广泛的接受，而且成为所有使用比较操作来进行排序的算法中最为广泛使用的算法。那么这种算法有什么样的特点呢？

虽然快速排序在归并上没有什么成本，但由于分解是按照元素大小进行，因此它在分解这一步颇费工夫，即先付出代价；在合并的时候不用费力，即后享受。因此，快速排序是一种“先付出后享受劳动果实”的算法。不过，快速排序之所以获得人们的青睐并不仅仅是因为“先付出后享受劳动成果”的思维方式。要明白它的缘由，就需要对快速排序进行深刻剖析。

10.4.1　快速排序的过程

按照前面所述，快速排序分为 4 个步骤：

1）选择杠杆点（分界点）。在待排序的序列里面按照某种方式选取一个元素，这是杠杆点。

2）分解。以杠杆点为界，将序列分为两个子序列，其中一个子序列里的所有元素小于等于杠杆点，另一个子序列里的所有元素大于杠杆点，如图 10-5 所示。

≤杠杆点	杠杆点	>杠杆点

图 10-5　快速排序的循环不变式

3）治之。递归对两个子序列进行快速排序。

4）合并。将排好序的两个子序列合并为大序列。

前面已经论述，快速排序的时间成本取决于分解这一步。因此，分解的好坏将对整个算法的效率产生决定性影响。但分解的好坏取决于杠杆点的选择。一旦杠杆点选好，分解操作本身并无任何稀奇之处，甚至是十分直接明了。下面就是标准的序列分解算法：

```
PARTITION(A, m, n)                          // A[m..n] 为一个序列
x=A[m];                                     // A[m]为选择的杠杆点
i=m;
for(j=m+1; j<=n; j++) {
   if(A[j] <=x) {
         i=i+ 1;
         temp=A[i];                         //以下三行交换 A[i]和 A[j]的内容
         A[i]=A[j];
         A[j]=temp;
         }
   }
temp=A[i];
A[i]=A[m];
A[m]=temp;
return i;
}
```

上述算法将子序列 $A[m..n]$以 $A[m]$为界（杠杆点）分解为两个子序列：$A[m..i-1]$和 $A[i+1..n]$。在 for 循环的每一次，$A[m..i]$里的元素都小于等于杠杆点 x，$A[i+1..j]$里元素则是大于杠杆点 x，而 $A[j+1..n]$里的元素是尚未分解到子序列里的元素。算法的循环不变式如图 10-6 所示。

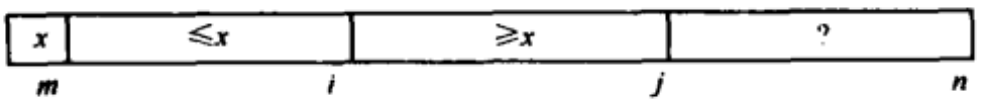


图 10-6 快速排序的序列分解算法的循环不变式

分解的正确性可以通过对循环不变式进行扩展而获得。图 10-7 为该分解算法的举例演示。

有了分解算法后，快速排序算法就可以定下来了：

```
QUICKSORT(A, p,r)
if (p<r) {
         q=PARTITION(A, p, r);
         QUICKSORT(A, p, q-1);
         QUICKSORT(A, q+1, r);
}
```

程序的初始调用为 QUICKSORT(A, 1, n)。

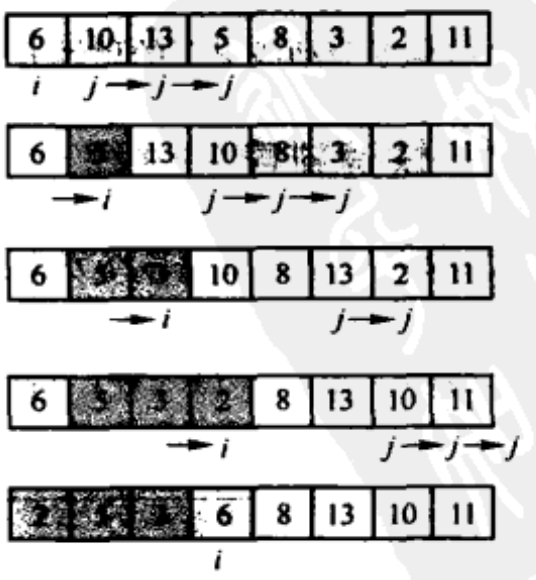


图 10-7 快速排序算法的分解过程演示

10.4.2　快速排序的时间复杂性分析

快速排序的时间复杂性体现在分解上（合并是微不足道的），因此，分解的成本将决定快速排序的成本。那么分解的成本是多少呢？

显然，分解的次数是其中一个决定因素，另一个因素则是每次分解需要进行的比较次数！对于一个有 n 个元素的序列来说，分解的次数最多只能是 $n-1$，即每次分解都形成一个空子序列和一个包含 $n-1$ 个元素的子序列；最少分解次数则是 $\log n$，即每次分解出的两个子序列长度都相当。剩下的问题是计算每次分解所需要进行的元素比较次数。

10.4.3　最坏情况分析

每次分解需要将杠杆点与被分解序列里面的其他所有元素进行比较，因此，第 1 次分解的成本是 $n-1$。第 1 次分解后形成两个子序列，假定每个序列的元素个数分别为 x_1 和 x_2，则 $x_1+x_2=n-1$。如果 x_1 和 x_2 均大于 0，则下一次分解这两个子序列的成本将分别为 x_1-1 和 x_2-1，总分解成本为 $x_1-1+x_2-1=x_1+x_2-2=n-3$。如果 x_1 和 x_2 中有一个等于 0，如 $x_1=0$，则 $x_2=n-1$。此时，下一次的分解成本将为 $x_2-1=n-1-1=n-2>n-3$。因此，每次分解的成本在其中一个子序列的元素个数为 0 的时候达到最大。而且按前面的分析，此时分解的次数也达到最大，因此，此种情况为快速排序的最坏情况。此种情况下分解次数为 $n-1$，每次分解比较的元素次数则分别为 $n-1$、$n-2$、…、1，因此，总成本为：

$$1+2+3+\cdots+(n-1)=n(n-1)/2=\Theta(n^2)$$

如果用递归表达式来表示此种最后情况下的分解，则有：

$$T(n)=T(0)+T(n-1)+\Theta(n)=\Theta(1)+T(n-1)+\Theta(n)=T(n-1)+\Theta(n)=\Theta(n^2)$$

如果画一棵递归树，则可以很容易地看出此种情况下的时间复杂性，如图 10-8 所示。

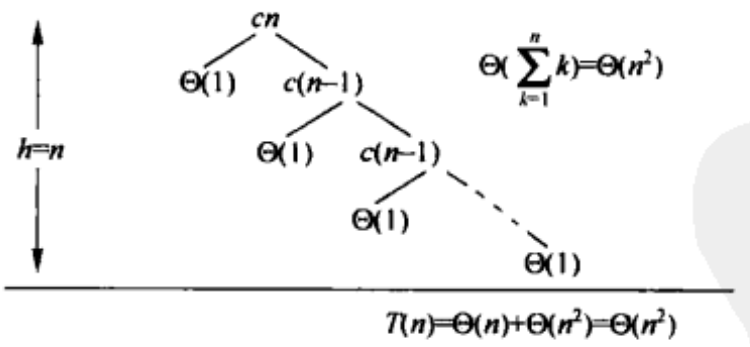


图 10-8　快速排序最坏情况下的递归树

那么什么时候会出现分解后的一个子序列为空的情况呢？当然是杠杆点元素是整个序列里最大或最小的时候。对于前面给出的分解算法来说，由于使用序列的第 1 个元素作为杠杆点，因此最坏情况出现在整个输入序列是已经排好序（正序或反序）的时候！

10.4.4　最好情况分析

快速排序的最坏情况的时间复杂性为平方级，似乎不如归并排序，也没有比插入排序

好。看来我们只能希望其在最好情况和平均情况下的表现出色些。最好的情况如何呢？

从前面的分析我们知道，如果每次分解产生的两个子序列都不为空，则每次分解的成本依次减少 2^{i-1}，这里 i 表示第 i 次分解（每次分解将去掉两个杠杆点元素）。因此，成本最低的时候将出现在分解次数最少的情况下。而分解次数最少的情况出现在每次分解都形成两个相等长度的子序列的时候，即均匀分解的时候。在这种最好情况下，整个快速排序的分解次数为 $\log n$，这里 n 是原序列的元素个数。由于每次分解需要比较的元素个数不会超过 n，因此，快速排序的总成本将小于 $n\log n$。

如果用递归表达式来表示，均匀分解情况下快速排序的时间复杂性可以表示为：

$$T(n)= 2T(n/2) + \Theta(n)$$

根据大师解法可知，$T(n)=\Theta(n\log n)$，即线性对数级。这个时间复杂性与归并排序的时间复杂性一样，但不如插入排序在最好情况下的时间效率。看来快速排序要想翻身只能靠平均情况下的出色表现。问题是，快速排序在平均情况下真的表现优异吗？

下面我们就来分析快速排序的平均情况。

10.4.5 平均情况分析

如果快速排序的子序列分解在正中间进行，则将产生最好的结果；如果在顶端分开，则出现最坏的情况。但在多数时候，其分解点可能既不是中间也不是两头。那么此时的情况如何呢？也就说，如果每次分解不产生两个长度一样的子序列，也不产生空序列，那么快速排序的时间复杂性会是怎样呢？例如，如果每次分解后一个子序列包含 10%的元素，另一个子序列包含 90%的元素，那么算法的时间复杂性会发生怎样的变化呢？

如果是此种情况，递归表达式为：

$$T(n)= T(n/10) +T(9n/10)+ \Theta(n)$$

按照该递归表达式画出递归树，如图 10-9 所示。

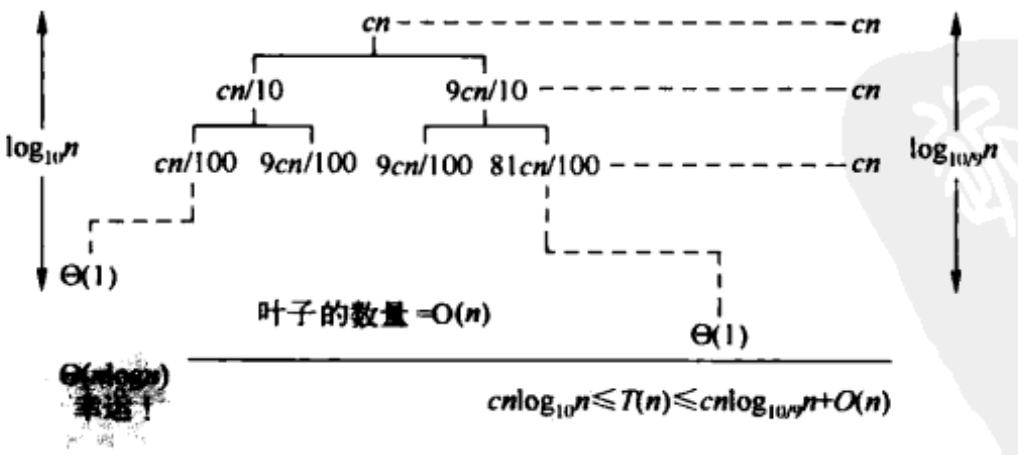


图 10-9 快速排序 9:1 分解情况下的递归树

结果是 $T(n)= \Theta(n\log n)$。这与均匀分解的结果一样！也就是说，此种分解与最优分解没有什么不同。事实上，可以证明（这个问题留给读者去证明），只要分解的时候产生的两个子序列的长度比为常数级，即 $c_1<L_1/L_2<c_2$，这里 L_1、L_2 分别为两个子序列的长度，c_1、c_2 为

任意的正常数，则不管这两个子序列绝对长度的差异有多大，哪怕是 1%对 99%，其时间复杂性仍然是 $\Theta(n\log n)$。由此可见，快速排序算法的韧性很强，只要分解不是按照最坏情况进行，结果就是最优！

事实上，快速排序的优点还不只有这些。假定我们的运气是好坏交替（这不就是人生吗），即一次最优分解接一次最差分解，这样交替往复，情况又如何呢？

最优的分解可以表示为 $L(n)= 2U(n/2) + \Theta(n)$（均匀分解），而接下来一次最差的分解可以表示为 $U(n)= L(n-1) + \Theta(n)$（分解为一个子问题）。将后面的表达式代入前面的一个，则有：

$$L(n)= 2(L(n/2-1) + \Theta(n/2)) + \Theta(n)= 2L(n/2-1) + \Theta(n)= \Theta(n\log n)$$

非常幸运！由此可见，在好运与厄运交替的情况下，快速排序的时间复杂性与最优情况一样。而且，前面的分析已经指出，哪怕分解的是 1%对 99%的不公平，快速排序也是最优情况。可以说，只要给它一点儿机会，快速排序就能取得最好的结果。这与人类的追求是多么的相像！难怪人们非常喜欢快速排序！

快速排序还具备其他的优点：它是一个原地排序算法（读者看出来了吧）；它能够充分发挥高速缓存的优势（为什么），而这在计算机时代是非常重要的一个品质；另外，人们可以通过对杠杆点的选择进行算法微调，快速排序从代码优化中所获得的改善巨大……而且实验表明，在通常情况下，快速排序的速度为归并排序的两倍！

另外，在生活中我们有时也会使用快速排序，别以为打牌的时候我们只使用插入排序，例如，如果牌很多（如 4 个人打三副牌），当我们感觉抓不下的时候，就会将一部分牌放在桌上，对手上剩下的部分进行整理，然后再对桌上的部分进行整理，然后将两部分整理好的牌合成一手牌。由于在分牌的时候通常按照花色进行（如将黑桃和梅花放在桌上，红桃和方块留在手上），此种分解再合并的排序显然符合我们的快速排序定义！

10.5　随机化快速排序

前面已经说过，只要给一丝希望，快速排序就能做出辉煌的结果；即使厄运不断，但只有你还给它一些好运，快速排序算法仍将取得最优。一切似乎说明，快速排序的最坏情况不会出现，或者出现的概率可以低到不影响大局的程度。试想，有谁运气那么背，每次分解都产生一个空子序列呢？因此，也许我们应该高枕无忧。但真是这样吗？

诚然，最坏情况只出现在每次分解都是最坏情况下，属于极小概率事件，就像现实中一个人从来只倒霉不走运的概率着实很低。但从另外一方面看，谁又能保证极小概率事件不会出现呢？大数定律不是告诉过我们小概率事件重复多次就成为必然事件吗？而更为要紧的是，如果提供数据的是你的对手，而他又知道你的分解算法，则他完全可以通过对数据进行某种安排使得最坏情况总是出现！也就是一直走霉运。

对于一个算法来说，这种极端情况我们不能不防，而防卫的手段就是随机化！

我们知道，快速排序分解出的子序列是否为空取决于杠杆点与其他元素的相对大小。因此，在快速排序的时候，要想保证不被对手玩弄，选择杠杆点的时候就不能是确定性的。如

果每次都是随机选择一个元素作为杠杆点，则无论对手进行多少次试验也不能得出任何肯定的结论，自然也就不能有针对性地进行数据次序的安排来使我们总是蒙受不幸。

那么这样一个随机化策略是否会增加算法的时间复杂性呢？或者说，采用随机化杠杆点选择后的快速排序的时间复杂性如何呢？显然，时间复杂性将不再依赖于输入数据的次序，而依赖于随机化选择。事实上，此时将没有任何一种特定输入次序会一定导致最坏情况出现，最坏情况的出现只与随机数产生器有关！

随机化快速排序的时间成本分析

设 $T(n)$为随机化快速排序算法在输入序列元素个数为 n 的情况下的运转时间，显然，$T(n)$是一个随机变量。又假设随机数产生器产生的随机数是相互独立的（这真的是一个大大的假设）！对于 $k=0, 1, \ldots, n-1$，我们定义如下标示随机变量（indicator random variable）：

$$X_k = \begin{cases} 1 & \text{如果分解产生的两个子序列长度比为 } k:(n-k-1) \\ 0 & \text{其他情况} \end{cases}$$

如果序列里没有重复元素，且所有可能的分解均概率相等（这是随机化的结果），则有：

$$E[X_k] = \Pr\{X_k = 1\} = 1/n$$

考虑到所有可能的分解，有：

$$T(n) = \begin{cases} T(0)+T(n-1)+\Theta(n) & \text{两子序列长度比为 } 0:(n-1) \\ T(1)+T(n-2)+\Theta(n) & \text{两子序列长度比为 } 1:(n-2) \\ \cdots & \\ T(n-1)+T(0)+\Theta(n) & \text{两子序列长度比为 } (n-1):0 \end{cases}$$

利用我们定义的标示随机变量 X，则可以将 $T(n)$表示为：

$$T(n) = \sum_{k=0}^{n-1} X_k \left(T(k)+T(n-k-1)+\Theta(n) \right) \tag{10-1}$$

对式(10-1)两边取期望值，则有：

$$\begin{aligned} E[T(n)] &= E\left[\sum_{k=0}^{n-1} X_k (T(k)+T(n-k-1)+\Theta(n))\right] = \sum_{k=0}^{n-1} E[X_k (T(k)+T(n-k-1)+\Theta(n))] \\ &= \sum_{k=0}^{n-1} E[X_k] E[T(k)+T(n-k-1)+\Theta(n)] \\ &= \frac{1}{n}\sum_{k=0}^{n-1} E[(T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n}\sum_{k=0}^{n-1}\Theta(n) \\ &= \frac{2}{n}\sum_{k=0}^{n-1} E[(T(k)] + \Theta(n) \end{aligned}$$

现在，剩下的工作是求解递归式$\frac{2}{n}\sum_{k=2}^{n-1}E[T(k)]+\Theta(n)$，该递归式解是多少呢？

显然，这个递归表达式的形式不能用大师解法进行求解，画递归树也不太好画，剩下的办法是替换法，而替换法需要先猜出一个解，然后予以证明。那么我们的猜测是什么呢？

根据我们前面对快速排序的分析可知，只有在最坏情况下其时间复杂性是平方级，其他情况都是 $n\log n$ 级，因此，我们猜出的解自然是 $cn\log n$，即 $E[T(n)] \leqslant cn\log n$，这里的 c 是一个正常数。

将该猜测解代入到递归表达式，则有：

$$E[T(n)]=\frac{2}{n}\sum_{k=2}^{n-1}c\,k\log k+\Theta(n) \qquad \text{代入猜测解}$$

$$\leqslant 2c/n\,(1/2n^2\log n-1/8n^2)+\Theta(n) \qquad \text{使用不等式}\sum_{k=2}^{n-1}k\log k \leqslant \frac{1}{2}n^2\log n-\frac{1}{8}n^2$$

$$=cn\log n-(cn/4-\Theta(n)) \qquad \text{将表达式表示成需要的形式}$$

$$\leqslant cn\log n \qquad \text{选择 } c \text{ 足够大，使得 } cn/4 \text{ 渐近大于 } \Theta(n)$$

因此，随机化快速排序的时间成本确实是线性对数级 $\Theta(n\log n)$。

10.6　排序的下限

好奇心强的读者可能会想，在计算随机化快速排序算法的时间复杂性时，我们猜测的为什么不是线性或者对数级，而是一个线性对数级成本呢？到目前为止我们讨论的三种排序算法的时间复杂性在最坏情况下都是 $\Theta(n\log n)$或以上（更差）。这是巧合吗？事实上，还有好几种本书没有讨论的排序算法：堆排序、树排序、冒泡排序、选择排序等，其最坏情况下的时间复杂性都是 $\Theta(n\log n)$或以上。难道这里有什么蹊跷吗？或者 $\Theta(n\log n)$是不可逾越的界限？

要回答这个问题也并不困难，只需稍做分析即可。

假如我们要对序列〈a_1, a_2, …, a_n〉进行排序。对于任何两个元素 a_i，a_j 来说，它们之间的关系只能是大于、小于和等于三种关系之一：$a_i>a_j$、$a_i<a_j$ 和 $a_i=a_j$。我们可以根据这三种关系将整个排序过程表示为一棵决策树：树的结点是 $a_i:a_j$ 之间的比较。假定没有重复结点，则 $a_i<a_j$ 的情况归结到结点 $a_i:a_j$ 的左子树里，$a_i>a_j$ 的情况归结到结点 $a_i:a_j$ 的右子树里，叶子结点代表的则是排好序的序列。例如，对于 6 个元素的序列来说，这样画出的决策树如图 10-10 所示。

由于 n 个元素的相对次序有 $n!$ 种可能，排序过程的决策树的叶子结点个数自然也就是 $n!$。而算法效率的最坏情况就是该树的高度 h，这是从开始排序（树根）到排好序（叶子）需要进行的比较次数的上限。高度越低，算法在最坏情况下的效率越高。显然，决策树可能的最低高度将是任何算法能够达到的最坏情况下的效率极限。

但是决策树的高度能低到什么程度呢？

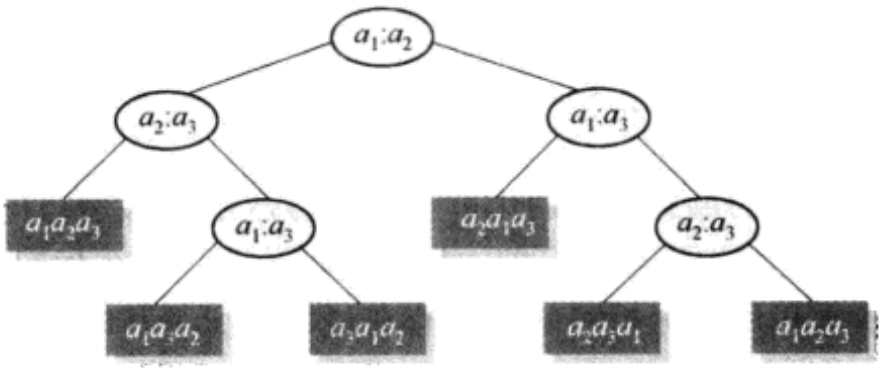


图 10-10 对 6 个元素进行排序的决策树

由于一棵高度为 h 的二叉树所能包含的所有结点数不超过 2^h，而由前面的分析可知，该树的结点数至少为 $n!$（叶子就已经是这个数了）因此，有 $2^h \geqslant n!$。因为：

$$2^h \geqslant n! \geqslant n(n-1)(n-2)\cdots(n/2) \geqslant (n/2)^{n/2}$$

所以：

$$h \geqslant \log((n/2)^{n/2}) \geqslant (n/2)\log((n/2)) = (n/2)(\log n - \log 2) = \Omega(n\log n)$$

根据以上分析，我们得出如下定理：

下限定理 *任何使用比较方法对 n 个元素进行排序的决策树的高度至少为 $\Omega(n\log n)$。*

该定理说明排序算法在最坏情况下的时间复杂性不可能低于 $n\log n$ 级！这就是说，我们前面探讨过的折半插入排序和归并排序已经是最优的算法了！而堆排序也是如此（堆排序在最坏情况下的时间效率分析留给读者自己去研究）。

例如：如果对 3 个元素进行比较排序，则 n=3，$n!$=6，log6=2.58。因此，对 3 个元素排序在最坏情况下需要进行 3 次比较！

10.7 线性排序

前面的分析告诉我们，最坏情况下，没有算法的时间效率可以好于 $n\log n$。但是人的贪心是没有止境的。从理论上说，对 n 个元素进行排序需要的时间至少是线性，因为我们至少需要将所有 n 个元素查看一遍，而这就需要与 n 成比例的线性时间。问题是，能否将整个排序的时间复杂性降低到理论上的极限，也就是线性呢？

答案似乎是不能，因为前面的分析已经阐明 $n\log n$ 是排序算法能够达到的最坏情况下的极限。但真是这样的吗？

答案是否定的。仔细观察可以发现，我们到目前所讨论过的算法有一个共同特点：都是通过对元素大小进行比较而决定不同元素之间的相对次序。虽然决策树定理给出了最坏情况下排序时间的下限为线性对数级 $n\log n$，但它只适用于使用比较方法进行排序的算法。

我们的思路是，如果不采用比较方法来对元素进行排序，那么我们就不受下限定理给出的时间复杂性的限制。换句话说，如果想要超越 $n\log n$ 级的渐近时间复杂性，就不能采用对元素进行比较的方法来排序。问题是，不进行比较又如何知道两个元素之间的大小关系呢？

例如，3 和 6 这两个数孰大孰小呢？当然是 6 大。如何知道 6 更大呢？这是因为我们在

心里对这两个数进行了比较而得出结论。但如果我们不对这两个数进行比较是否仍能知道是 6 更大吗？而要知道这个答案就得分析排序的核心要素：排序的目的到底是什么？

排序的根本目的是为每个元素找到其应在的位置并将其放在该位置，使凡是比它小的元素都在它前面，凡是比它大的元素都放在它后面。对于一个序列里的特定元素 x 来说，要知道比该元素小的元素的个数并不一定需要将该元素与序列里面所有的元素进行比较！

如果我们有一组顺序排列的盒子，每个盒子从左至右标记为 1、2、3、4、5、6……在我们看到一个数后便将该数扔进标有该数的盒子里。例如，看到 3 后将 3 扔进 3 号盒子，看到 6 后将其扔到 6 号盒子，看到 7 则将其扔进 7 号盒子……扔完所有的数后，我们将盒子里面的数从左至右收集起来。这样 3 将排在 6 的前面，而 6 又排在 7 的前面，正是我们要的次序！

上述操作将 3、6、7 进行了正确排序，而且整个排序过程没有在这 3 个数之间进行任何比较，整个排序的时间复杂性为线性！

10.8　计数排序

将 10.7 节给出的例子进行推广可知，对于特定元素 x 来说，如果我们知道给定序列里比它小的元素个数，则 x 的位置就已经确定。例如，如果比 x 小的元素个数为 m，则 x 在输出数组里面所占的位置就是 $m+1$，这就是我们要讨论的计数排序：对每一个元素，我们计数比它小的元素个数，然后将该元素放入合适的位置上。下面我们给出计数排序的算法。

输入数组：$A[1..n]$，这里 $A[j] \in \{1, 2, \ldots, k\}$，即输入序列里元素的取值范围不超过 k。
输出数组：$B[1..n]$，用于存放排好序的序列。
计数数组：$C[1..k]$，用于计数。
计数排序的程序代码如下：

```
COUNTING-SORT (A, B, C)
for (int i=1; i<=k; i++)          //将计数数组初始化清零
    C[i] =0;
for (int j=1; j<=n; j++)
    C[A[j]] =C[A[j]] + 1;     //计数取值为 A[j]的元素个数
for (i=2; i<=k; i++)
    C[i] =C[i] + C[i-1];      //计数取值小于等于 i 的元素个数，存放在 C[i]里
for (int j=n; j>=1; j--) {
    B[C[A[j]]] =A[j];         //元素 A[j]在输出数组里所处的位置为 B[C[A[j]]]
    C[A[j]] =C[A[j]] -1;      //剩下的比取值小于等于 A[j]的元素个数减 1
}
```

由于数组 A、B、C 在第 3 重循环时需要同时使用，因此计数排序确实需要使用 3 个数组。除了输入数组 A 外，另外两个数组 B 和 C 均是额外需要的空间。也就是说，计数排序的空间需求比我们前面讨论过的任何比较排序算法都高！

计数排序算法的正确性是显而易见的，因为元素 x 放置的位置在所有比它小的元素之后！

例如，假如我们对如下输入数组里的元素进行计数排序：

A	6	1	4	5	8	3	4

第 1 个循环，计数数组 C 初始化为 0（8 个元素）：

C	0	0	0	0	0	0	0	0

第 2 个循环后，数组 C 的元素变为：

C	1	0	1	2	1	1	0	1

第 3 个循环后，数组 C 的元素变为：

C	1	1	2	4	5	6	6	7

第 4 个循环将每个元素放入它们应在的位置。随着循环变量 j 从 n 减为 1，数组 A、B、C 的元素改变过程如下：

j=7:

A	6	1	4	5	8	3	4	
B				4				
C	1	1	2	3	5	6	6	7

j=6:

A	6	1	4	5	8	3	4	
B		3		4				
C	1	1	1	3	5	6	6	7

j=5:

A	6	1	4	5	8	3	4	
B		3		4			8	
C	1	1	1	3	5	6	6	6

j=4:

A	6	1	4	5	8	3	4	
B		3		4	5		8	
C	1	1	1	3	4	6	6	6

j=3:

A	6	1	4	5	8	3	4	
B		3	4	4	5		8	
C	1	1	1	2	4	6	6	6

j=2:

A	6	1	4	5	8	3	4	
B	1	3	4	4	5		8	
C	0	1	1	2	4	6	6	6

j=1:

A	**6**	1	4	5	8	3	4
B	1	3	4	4	5	**6**	8
C	0	1	1	2	4	**5**	6	6

至此，整个排序工作完成，*B* 数组里就是已经排好序的序列：1、3、4、4、5、6、8。

计数排序的时间复杂性分析

由于计数排序没有进行元素之间的比较，因此其时间复杂性将不受下限定理的约束。但是计数排序是否达到我们所期望的线性时间复杂性呢？分析就知道了。

计数排序一共只有 4 个循环，第 1 个循环需要执行 k 次，第 2 个循环执行 n 次，第 3 个循环执行 k 次，第 4 个循环执行 n 次。因此，计数排序的时间复杂性为 $\Theta(n+k)$。如果 $k=O(n)$，则计数排序的时间复杂性为 $\Theta(n)$。这就是线性！

除了时间复杂性为线性外，计数排序还有一个优点：稳定！即取值相同的元素在排序前后的相对位置不变！这点通过对前面例子里 *A*、*B* 两个数组的比较就可以看出（如图 10-11 所示）。

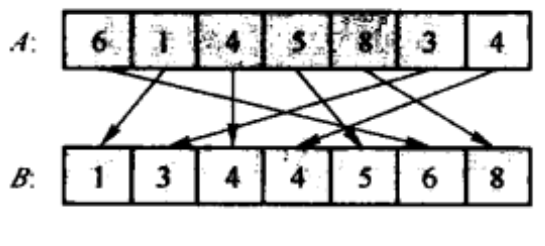


图 10-11　计数排序为稳定排序

排序的稳定性有时是很重要的一个性质。在很多时候，排序的元素并不只是一个单纯的数值，而是一个记录里的某个域。按这个域进行排序的时候，相应的记录都将发生位置变动。如果能够保持相等元素的相对位置不变，则前面进行的任何排序或安排都将得到保持。

至此，一切看上去都非常美好：我们可以对取值不超过 k 的 n 个整数在 $O(n+k)$ 的时间内排好序！如果 k 与 n 为同一个数量级，则整个排序算法的时间复杂性就是线性！

但一切真的都很美好吗？

细心的读者可能已经看出，计数排序的线性时间复杂性有个前提：k 的数量级不高于 n 的数量级。这可是个很大的假设！如果这个假设不成立，则一切都化为南柯一梦。

在计数排序算法里，计数数组 *C* 的大小为输入序列里最大元素的取值。输入元素取值的范围越大，则计数数组 *C* 的空间需求也越大。如果输入数组的元素个数很少，则数组 *C* 的绝大部分空间都将浪费。要命的是，如果输入数组元素的取值过大，恐怕计算机里就没有空间来容纳数组 *C* 了！而更要命的是，如果 $k>>n$，则计数排序就不再是线性的了！例如，如果 $k= O(n^2)$，则 $\Theta(n+k)= \Theta(n^2)$！这就是计数排序的阿喀琉斯脚后跟！

上述缺陷的存在使得计数排序只在输入元素的取值比较有限的情况下才好使用，而如果输入元素的取值可以任意大的时候，我们就需要另外一种线性排序算法了！

10.9 基数排序

在输入序列里元素取值很大的时候，计数排序算法在效率和可行性上都将崩溃，而基数排序却可以应付数值很大的情况。此种排序算法来源于 IBM 公司创始人 Herman Hollerith（见图 10-12）发明的纸卡排序机。1890 年，美国进行了一次全国人口普查。而普查完后需要进行各种数据统计，而这些统计就无可避免地涉及了排序。人工排序显然是不实际的。为此， Herman Hollerith 发明了一种机械排序装置。这种机械排序装置使用的算法就是基数排序算法，即一个数位一个数位地进行排序！而在对每个数位的排序则使用某种稳定排序算法。

图 10-12 基数排序发明人，IBM 公司创始人之一：Herman Hollerith

（图片来源：Wikipedia）

Hollerith 一开始想到的是从最高有效位到最低有效位进行排序：即先按照最高有效位对所有元素进行一次排序，然后对结果序列按次高有效位进行排序，就这样一直下去，直到最低有效位。但遗憾的是，此种排序方法所排出来的结果非常糟糕：结果序列根本就没有任何次序！例如，如果对 123、312、245、531 四个数进行排序，第 1 次按百位数排序后，结果为：123、245、312、531；第 2 次按十位数排序后，结果变为：312、123、531、245；最后一次也就是第 3 次按个位数排序后，结果就会变为：531、312、123、245。而这个结果显然不是我们所希望看到的！

因此，Hollerith 将排序方法变成从最低有效位到最高有效位进行排序，结果就令人非常满意。还是以刚才的四个数为例，第 1 次按个位数排序后结果为：531、312、123、245；第 2 次按十位排序后，结果为 312、123、531、245；第 3 次也是最后一次按百位数排序后，结果为：123、245、312、531。完成正确！

之所以要先排最低有效位后排最高有效位，是因为越是后排的数位，其对结果次序的影响越大！而最高有效位显然比最低有效位对数的大小影响更大！

图 10-13 描述的是一个更为复杂的基数排序的例子。第 1 次按个位数排序，第 2 次按十位数排序，第 3 次按百位数排序，第 4 次按千位数进行排序。

2323	3420	3420	8139	1338
5456	2323	2323	2323	2323
6657	4934	4934	1338	3420
8139	5495	1338	3420	4934
5495	5456	8139	5456	5456
1338	6657	5456	5495	5495
3420	1338	6657	6657	6657
4934	8139	5495	4934	8139

图 10-13　基数排序演示

10.9.1　基数排序的正确性

从本书给出的两个例子可以看出基数排序的正确性。但用例子来证明正确性显然是不够的。因此，下面用数学归纳法来证明基数排序的正确性。

证明　假定一组序列已经按照低 $t-1$ 个数位排好序了，现在按第 t 位数位进行排序。我们只需证明对于任意两个数来说，它们在第 t 次排序后其相对位置正确即可。

对于任意两个数来说，在按第 t 个数位进行排序的时候，只有两种情况需要考虑：

1）它们在第 t 位数位上的值不同。此时，按照第 t 位数位排序后，它们的次序将正确（如图 10-13 中按百位数排序时的 2323 和 3420）。

2）它们在第 t 位数位上的值相同。此时，按照第 t 位数位排序后，由于我们使用的是稳定排序，它们的次序将与之前（即在按 t 位数排序之前）一样，因此，其次序仍然保持正确！（如图 10-13 中按百位数排序时 2323 和 1338）。

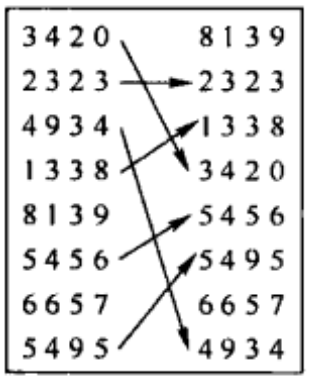


图 10-14　基数排序正确性演示

图 10-14 描述的就是基数排序的正确性证明。

因此，基数排序算法确实正确。　□

10.9.2　基数排序的时间效率分析

在前面论述基数排序的时候，我们并没有说明使用何种算法来对每个数位进行排序，唯一确定的是排序所用算法必须是稳定的（否则基数排序的结果将可能不正确）。那么用何种排序算法比较适合呢？也许读者已经猜出来了：计数排序。

首先，计数排序是一个稳定排序；其次，由于是按数位进行排序，一个一位数的数值能有多大呢？当然不会超过数的进制！对于十进制数来说，每个数位的取值不会超过 9！也就是说，数的取值范围很小，而这很适合计数排序！即使是十六进制、三十二进制，数的最大取值也只有 15 和 31！

我们假设：

- 使用计数排序作为对每个数位进行排序的算法。

- 需要排序的序列一共有 n 个元素。
- 每个元素占用 b 个二进制字位。

如果以 r 个字位为一个数位，即使用 2^r 进制，则每个数占用 b/r 个 2^r 进制数位。由于基数排序按数位进行，数位的个数就是基数排序的次数，即需要调用计数排序 b/r 次。

计数排序的时间复杂性为 $\Theta(n+k)$，这里 n 是数的个数，而 k 是数的取值上限。如果每个 b 二进制位的数被分解为 r 位长的段，则每次计数排序的时间复杂性为 $\Theta(n+2^r)$。由于需要进行 b/r 次计数排序，因此，总时间成本为 $\Theta(b/r(n+2^r))$。

例如，对于占用 32 个二进制字位的数来说，可以将其分解为 4 段，每段 8 个二进制字位的数。这样，数的进制为 2^8，一个数占 4 个进制为 2^8 的数位。而基数排序的次数为数位的个数，因此，在上述分解模式下，对 32 个字位的数进行排序一共需要运行 4 次计数排序。

显然，要做的就是选择适当的 r 使得整个成本最低。虽然 r 的增加能导致计数排序的次数减少，但当 r 增加到远远大于 $\log n$ 的时候（此时 $n<<2^r$），每次计数排序的成本将迅速增加。因此，最佳的选择是保持每次计数排序的时间复杂性为线性！而此种情况下，r 的选择应该是 $\log n$。此时，基数排序的总时间成本为 $\Theta(bn/\log n)$。

如果输入数据的取值范围为 $0\sim n^d-1$，则有 $b=d\log n$，此时，基数排序的成本为 $\Theta(dn)$。

下面为基数排序的 C 程序实现（有兴趣的读者可以到计算机上试一试）：

```
void radix(int A[], int B[], int n, int b, int r, int C[])
// n 为元素个数
// A[]为待排序的元素序列，大小为 n
// B[]计数排序所使用的额外空间，大小为 n
// b 为最大元素的长度（字位数）
// r 为每个数位的长度（即基数排序使用 2^r 进制）
// C[]为计数数组，大小为 2^b
{
      int j;
      m=b/r;
      for (int i=0, t=1; i<m; i++, t*=2^r) {
            for (j=0; j<r; j++)
                  C[j] = 0;                          //初始化计数数组
            for(j=0; j<n; j++)
                  C[(A[j]/t)%2^r]++;                 //C[j]为等于 j 的元素个数
            for (j=1; j<r; j++)
                  C[j] = C[j-1] + C[j];              //C[j]为小于等于 j 的元素个数
            for (j=n-1; j>=0; j--)
                  B[--C[(A[j]/t)%2^r]]=A[j];         //将元素按次序放入数组 B 中
            for (j=0; j<n; j++)
                  A[j] = B[j];
            }
}
```

在实际中，基数排序非常适合对取值很大的数进行排序，并且也较容易编码和维护。例如，如果对 2 000 个 32 位的数进行排序，基数排序只需要进行 4 次（按 2^8 进制分解）或 2 次（按 2^{16} 进制分解）计数排序，而归并和快速排序均需要进行至少 log2000=11 次递归！

基数排序的缺点是不呈现时空局域性，因为在按照每个数位进行排序的过程中，一个数的位置可能发生巨大的变化，从而不能充分利用缓存提供的优势。

10.10　桶排序

1995 年 7 月～1996 年 8 月，我工作于朗讯（Lucent）公司的 Westminster 分部，负责一个 27 人的网络打印小组。也许有人认为网络打印用不了 27 个人来管理，但当时朗讯的 Westminster 分部有 2 000 多名工作人员，大约 5 000 台电脑和 400 台打印机，其中有高速阵列打印机 8 台，彩色复印打印机 1 台。这种高速阵列激光打印机在当时能够每分钟打印 180 张纸，当打印机开动的时候，打印出来的纸张像雪片一般纷至沓来，颇为壮观！每台售价都在好几万美元，而彩色激光复印打印机就更贵了。由于彩色复印能用来复印钞票或者别的什么证件，因此对其进行严格的管理势在必行。由于这些高速阵列打印机和彩色复印打印机的贵重性，它们被放置于一个专门的中央机房，由网络打印小组统一管理。

需要使用这些打印机的人可直接将打印任务发送到这些打印机上，但自己不能到打印机上取文件。取件任务由网络打印小组完成。我们需要将所有打印出的文件分类排序，放在一个公共地方供用户取用。由于打印量巨大，分类排序的工作相当繁重。为此，我们在打印机房外面设置了一个打印件收发室。该收发室里布满了很多方格，每个方格属于 Westminster 分部里面的某一个人（领导）或某个组（普罗大众）。凡是这个人或组打印的东西都将放置在相关方格里。网络打印小组的一个日常任务就是将打印出来的文稿分发到每个方格里，而方格则是事先就按照一定的顺序排好的。因此，分发到方格里后，排序就基本完成。

这种排序就是桶排序！

在生活中，邮局里分拣信件使用的就是桶排序。事实上，桶排序来源于邮局分拣信件（见图 10-15）。

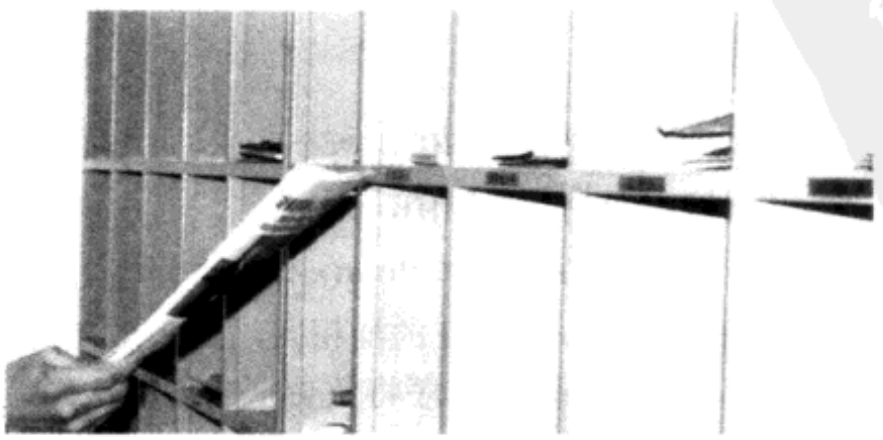


图 10-15　桶排序来源于邮局的信件分捡方法

10.10.1 桶排序的定义

桶排序（bucket sort），有时也称为**盒子排序**（bin sort），因为此种排序来源于邮局使用的盒子信件分发方法。不过和邮局信件分发不同的是，桶排序的有效性需要假定输入数据是由一个完全随机过程产生，而邮局信件分发却不需要进行此种假设（当然邮局信件也可以看做是随机产生的，但是统计数据显示，邮局信件表现出较多的聚集效应，即发往某个地方的信件可能远远多于发往另一个地方的信件）。我们要求桶排序的输入数据呈均匀分布，例如，输入数据随机均匀分布在区间[0, 1)内。

在上述假设下，桶排序的思想如下：

1）将区间[0, 1)分解为 n 个大小相等的桶。

2）将 n 个输入数据按照其取值不同分配到 n 个桶里。

3）对每个桶里面的数据进行排序。

4）然后将桶里面的元素按顺序收集。

桶排序的伪代码程序如下：

```
输入数组：A[1..n]，对任意 i，有 0≤A[i]<1
辅助数组：B[0..n-1]个链表，初始状态均为空

BUCKET-SORT (A, n)
1.for(i=1; i<=n; i++)
2.      将数据 A[i]插入链表 B[floor(n×A[i])]里;
3.for(i=0; i<=n-1; i++)
4.      对链表 B[i]里面的数据进行插入排序;
5.将链表 B[0], B[1],…,B[n-1] 首尾相连连接起来;
```

在上述桶排序的算法里，对每个桶里的元素进行排序使用的是插入排序，而这是一个平方级的排序算法。为什么不选择渐近趋势更好的算法呢？其中一个原因是在我们假定输入元素是均匀分布后（这是为什么我们要假设输入元素是由一个随机过程产生的），分配到任意一个特定桶里面的元素个数不会太多（否则就不均匀，而是聚集），因此，插入排序的时间复杂性可以不用计较。但辅助排序选用插入排序还有另一个原因，读者可自行思索答案。

10.10.2 桶排序的正确性

要证明桶排序的正确性，只需要证明对于任意两个元素 $A[i]$和 $A[j]$，它们在桶排序后的相对次序正确即可。不失一般性，假定 $A[i] \leqslant A[j]$，则有：

$$\text{floor}(nA[i]) \leqslant \text{floor}(nA[j])$$

因此，元素 $A[i]$被放进的桶要么与 $A[j]$所在的桶相同，要么比 $A[j]$所在的桶更靠前。

如果它们放进同一个桶里，则辅助插入排序将负责把它们的次序调整好；如果它们被放进不同的桶，则在按顺序收集所有桶的数据时，连接将保证它们之间的相对次序正确。

因此，桶排序算法的结果正确。

10.10.3　桶排序的时间复杂性分析

从桶排序算法的伪代码实现可以看出，除了插入排序部分外，其他部分的伪代码行都是线性级时间复杂性 $\Theta(n)$。在前面的论述中说过，使用插入排序作为桶排序的辅助算法的前提是每个桶里面的元素个数不能太多，否则由于插入排序的时间复杂性为平方级，整个算法就将变成平方级算法！但问题是，桶里面元素不多就能保证是线性算法吗？

从直觉上看，n 个元素分配到 n 个桶，感觉应该每个桶分到一个元素。至少，我们感觉每个桶获得的元素个数似乎应该是个常数，因此，对每个桶内数据的排序时间为常数，对所有桶都排序一遍的时间复杂性为 $O(n)$。这样，整个桶排序的时间成本也是 $O(n)$。

不过，直觉归直觉，很多时候直觉并不正确。因此，我们需要某种形式化的证明来告诉我们所做的决定是正确的。这样，仔细分析就很有必要。

设随机变量 n_i=落入桶 $B[i]$里的元素个数，则算法的总时间复杂性为：

$$T(n) = \Theta(n) + \sum_{i=0}^{n} O(n_i^2)$$

两边取期望值，有：

$$E[T(n)] = E[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)] = \Theta(n) + \sum E[O(n_i^2)]$$

$$= \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2]) \qquad (10\text{-}2)$$

现在的问题就变成求解 $E[n_i^2]$，这里 $i=0,1,\cdots,n-1$。这是对桶 i 内的元素进行插入排序所需要的时间期望值。那么这个期望值是多少呢？

根据先前的讨论，这个值应该是个常数，否则桶排序的线性时间复杂性就不能成立。显然，我们需要求取 n_i 的表示才能计算出 $E[n_i^2]$。为此，我们定义一个标示随机变量：

$$X_{ij} = I\{\text{元素 } A[j]\text{落入桶 } i\}$$

根据我们的均匀假设，每个元素落入 n 个桶的某个桶的概率应该是 $1/n$，因此有：

$$Pr\{\text{元素 } A[j]\text{落入桶 } i\} = 1/n$$

而落入桶 i 的元素个数就是所有标示随机变量 X_{ij} 之和，即

$$n_i = \sum_{j=1}^{n} X_{ij}$$

对上面 n_i 的表达式两边取平方后，再取期望值，有：

$$E[n_i^2] = E\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] = E\left[\sum_{j=1}^{n} X_{ij}^2 + 2\sum_{j=1}^{n-1}\sum_{k=j+1}^{n} X_{ij}X_{ik}\right]$$

$$= \sum_{j=1}^{n} E[X_{ij}^2] + 2\sum_{j=1}^{n-1}\sum_{k=j+1}^{n} E[X_{ij}X_{ik}]$$

而上述表达式里的第 1 项为：

$$E[X_{ij}^2] = 0^2 \Pr\{A[j]\text{不落入桶}i\} + 1^2 \Pr\{A[j]\text{落入桶}i\}$$
$$= 0\left(1-\frac{1}{n}\right)+1\frac{1}{n}=\frac{1}{n}$$

因此，剩下的工作是计算表达式里的第 2 项 $E[X_{ij}X_{ik}]$。该项取值为多少呢？

因为 $j\neq k$（$j=k$ 的情况全部包括在 $E[X_{ij}^2]$的分析中了），而 X_{ij} 和 X_{ik} 为独立的随机变量（两个不同元素落入到同一个桶的事件之间没有任何因果或其他联系），因此有：

$$E[X_{ij}X_{ik}] = E[X_{ij}]E[X_{ik}] = (1/n)(1/n) = 1/n^2$$

综上所述，有：

$$E[n_i^2] = \sum_{j=1}^{n} E[X_{ij}^2] + 2\sum_{j=1}^{n-1}\sum_{k=j+1}^{n} E[X_{ij}X_{ik}]$$
$$= \sum_{j=1}^{n}\frac{1}{n} + 2\sum_{j=1}^{n-1}\sum_{k=j+1}^{n}\frac{1}{n^2} = 1+2\binom{n}{2}\frac{1}{n^2}$$
$$= 1+2\frac{n(n-1)}{2}\frac{1}{n^2} = 1+\frac{n-1}{n}$$
$$= 2-\frac{1}{n}$$

因此，对任意一个桶 i 内的元素进行插入排序的时间复杂性确实不超过一个常数（2）。将此结果代入式（10-2），就获得桶排序的总时间成本为：

$$E[T(n)] = \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2])$$
$$= \Theta(n) + \sum_{i=0}^{n-1} O\left(2-\frac{1}{n}\right)$$
$$= \Theta(n) + O(n) = \Theta(n)$$

因此，桶排序的确是一个线性排序算法。

与其他线性排序算法一样，桶排序也不进行任何元素比较，而是利用元素的取值直接映射到相应的桶里。不过从本节的分析可知，如果输入数据不是均匀分布的，则上述桶排序的时间复杂性分析就完全站不住脚了。因此，只有当输入元素是均匀分布时，桶排序才好使用。

10.11 次序选择

到目前为止，本章一直在讨论各种排序算法。由于对时间效率的执著追求，人们对排序算法进行了不懈改进，并最终获得了一批线性排序算法。但排序本身并没有什么意义，它的意义在于排序后对有序序列的使用上。而这种意义有两个：一是方便将来的查找工作（将在第 11 章详细论述）；二是告诉我们任何特定元素在一个团体里面的次序（排名）。显然，

一旦排序完成，则所有元素在整个序列里面的次序就都已敲定。

但如果我们想知道的并不是所有元素的次序，而只是某一个或某几个元素在序列里的次序，或者处于某个特定位置上的元素是谁，则并不一定需要进行排序。最简单的例子就是找出一组数里的最大值。显然，找出最大值并不需要对所有元素排序。同理，在一群人里寻找最优或“最 X”的人时，我们也并不需要对所有的人来一个排序才能决定；否则，要找出全城或者全国“最 X”的人，恐怕就会因计算复杂性太高而无法进行。

此时，我们有更加有效的办法来处理。而在一组数（一个序列）里面挑选出处于特定位置的元素的问题就是所谓的次序选择问题。

具体来说，次序选择问题的一般定义如下：在 n 个元素里面选择出第 i 小（或大）的元素，即找出排名为 i 的元素。当 i=1 时，我们要的是最小值；当 i=n 时，则是寻找最大值；当 i=(n+1)/2 时，则是求中值。这里需要说明的是，如果 n 为偶数时，则序列里有两个中值，分别处于第 n/2 和 n/2+1 的位置。此时的“中值”可以指其中任意一个。不失一般性，本书均指前面一个中值。

显然，次序问题可以用排序来解决：先排序，然后读取处于位置 i 的元素即可。这样整个算法的时间复杂性就是排序的时间复杂性，即 $O(n\log n)$（使用比较排序的情况下）。而这显然太过了，因为我们只需要某一个位置上的元素，而排序则决定了所有位置上的元素，做的工作似乎超过所需。因此，一个自然的想法是：是否存在更快的算法呢？

10.12　快速次序选择算法

仔细分析发现，次序选择只需要知道比特定元素（如 x）小的元素个数。而对于比 x 小的所有元素来说，它们之间的次序关系我们并不关心。这个发现就能帮助我们设计出一个比直接使用排序更优的次序选择算法。读者能否想到这样一个算法呢？

读者可以回想一下我们讨论过的快速排序算法。该算法采用的是分治策略：每次把序列以杠杆点为界分为大小两个子序列。虽然此时两个子序列尚未排好序，但杠杆点在整个序列里的位置已经确定！如果杠杆点的次序正好是我们要寻求的位置（如中值），则我们已经完成任务。如果杠杆点的次序不是我们要寻求的位置，则我们要寻求的元素必然处于刚刚分解出来的两个子序列里面的一个！此时，我们便可将下一步的寻找限制在一个子序列里。这样就获得了一个比排序更优的次序选择算法，而这个次序选择算法就称为**快速次序选择算法**。这个算法是从快速排序算法改进而来的！这也许是快速排序讨人喜欢的另一个原因吧！

快速次序选择算法的伪代码程序如下：

```
ORDER-SELECTION(A, p, q, i)          //在序列 A[p..q]选择第 i 小的元素
1. if(p==q)
2.     return A[p];                  //已经找到第 i 小的元素
3. r=PARTITION(A, p, q);             //调用快速排序的分解算法
4. k=r-p+1;                          //k=rank(A[r])
5. if (i==k)
```

```
6.    return A[r];                          //已经找到第 i 小的元素
7. if (i<k)
8.    return ORDER-SELECTION (A, p, r -1, i);
                                            //在序列 A[p..r-1]选择第 i 小元素
9. else
10.   return ORDER-SELECTION (A, r + 1, q, i -k);

                                            //在序列 A[r+1..q]选择第 i-k 小元素
```

该算法在每次调用快速排序的分解算法后的状态如图 10-16 所示。

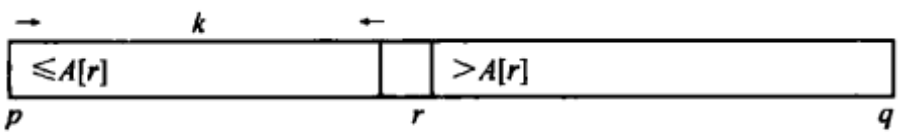


图 10-16 次序选择的循环不变式

显然，如果 $A[r]$的次序（伪代码里的变量 k）不是序列里第 i 小的元素，则有两种情况：

1）$A[r]$的次序大于 i，因此第 i 小的元素在元素 $A[r]$的左面，即 $A[p..r-1]$里。此时，需要在 $A[p..r-1]$里寻找第 i 小的元素。

2）$A[r]$的次序小于 i，因此第 i 小的元素在元素 $A[r]$的右面，即 $A[r+1..q]$里。此时，需要在 $A[r+1..q]$里寻找第 i 小的元素，只不过此时我们已经丢掉了 $A[p..r]$里面的所有元素，因此，整个序列里面第 i 小的元素在 $A[r+1..q]$里则是 $i-k$ 小的元素。

例如，如果在下列数组里选择第 5 小的元素：

4	6	8	9	2	23	1	34	15	20

使用快速排序的分解算法，杠杆点=4，分解如下：

1	2	4	9	6	23	8	34	15	20

分解算法返回的杠杆点的次序（排名）为 3，而这个排名比我们要找的第 5 名小，因此，我们将搜寻范围缩小到杠杆点的右面，即表格中阴影部分数字的范围。但此时，由于我们已经丢掉了 3 个比第 5 名更小的元素，因此在阴影部分里应该选择第 5-3=2 小的元素。

9	6	23	8	34	15	20

再次使用快速排序的分解算法，杠杆点=9，分解如下：

6	8	9	23	34	15	20

分解算法返回的杠杆点的次序（排名）为 3，而这个排名比要找的第 2 名大，因此，将搜寻范围缩小到杠杆点的左面，即下面表格中阴影部分数字的范围。

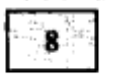
6	8

再次使用快速排序的分解算法，杠杆点=6，分解如下：

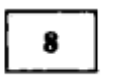
6	8

分解算法返回的杠杆点的次序（排名）为 1，而这个排名比要找的第 2 名小，因此，将搜寻范围缩小到杠杆点的右面，即下面表格中阴影部分数字的范围。但此时由于我们已经丢

掉了 1 个比第 2 名更小的元素，因此在阴影部分数字里应该选择第 2-1=1 小的元素。



再次使用快速排序的分解算法，杠杆点=8，分解如下：



分解算法返回的杠杆点的次序（排名）为 1，而这个排名正好等于要找的第 1 名，此时，整个算法结束，返回元素 8。即 8 就是要寻找的整个序列里面第 5 小的元素。

快速次序选择算法的时间成本分析

显然，快速次序选择算法的正确性可从快速排序正确性的分析上推导出来。但其时间复杂性是多少呢？由于快速次序选择算法所做的工作不会超过快速排序算法所做的工作，因此其时间成本当然不会比快速排序高。但问题是它会比快速排序算法的时间成本低吗？

假定不存在重复元素，最好的情况是每次分解去掉一半元素，则获得算法的时间复杂性为：

$$T(n) = T(n/2) + \Theta(n) = \Theta(n)$$

如果每次都只去掉一个元素（即杠杆点），则算法时间复杂性为：

$$T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$$

如果每次去掉 1/10 的元素，则有：

$$T(n) = T(9n/10) + \Theta(n) = \Theta(n)$$

由此可见，快速次序选择算法除了在最坏情况下与快速排序算法的时间复杂性一样外，在其他任何情况下，其时间成本都比快速排序的时间成本低，为线性！而线性则是次序选择能够达到的最优时间效率。因为任何一个元素都有可能是第 i 小的元素，所以至少需要将每个元素考察一遍，而这就需要线性时间！

10.13　随机快速次序选择算法

快速次序选择算法虽然在绝大部分时候的时间成本为线性，但最坏情况下却是平方级。虽然最坏情况不是什么大概率事件，但就像我们在分析快速排序的时候说过，谁也不能保证最坏情况就一定不会频繁发生。如果提供数据的是我们的对手，则此种情况几乎总是会发生。

因此，像快速排序一样，我们也使用随机化手段来防止最坏情况总是出现的可能。这样就获得如下的随机快速次序选择算法：

```
RANDOM-SELECTION(A, p, q, i)          //在序列 A[p..q]选择第 i 小元素
1. if (p==q)
2.       return A[p];                 //已经找到第 i 小的元素
3. r=PAND-PARTITION(A, p, q);         //调用快速排序的分解算法
4. k=r-p+1;                           //k = rank(A[r])
5. if (i=k)
6.       return A[r];                //已经找到第 i 小的元素
```

```
7. if (i<k)
8.        return RANDOM-SELECTION (A, p, r-1, i);
                                          //在A[p..r-1]选择第i小元素
9. else
10.       return RANDOM -SELECTION (A, r+1, q,i-k);
                                        //在A[r+1..q]选择第i-k小元素
```

此算法与快速次序选择算法的唯一不同是在分解的时候：不是选择 $A[p]$作为杠杆点，而是从序列 $A[p..q]$里面随机选择一个元素作为杠杆点。

随机快速次序选择算法的时间期望值分析

设 $T(n)$ 为随机快速次序选择算法在输入序列元素个数为 n 的情况下的运转时间。显然，$T(n)$是一个随机变量。同时假设（用于随机分解的）随机数产生器产生的随机数相互独立。

对于 $k=0, 1, \ldots, n-1$，我们定义如下标示随机变量：

$$X_k = \begin{cases} 1 & \text{如果分解产生的两个子序列长度比为 } k:(n-k-1) \\ 0 & \text{其他情况} \end{cases}$$

如果序列里没有重复元素，并且所有可能的分解均概率同等（这是随机化的结果），则有：

$$E[X_k] = \Pr\{X_k = 1\} = 1/n$$

为了获得时间复杂性上限，假定每次分解后，需要的元素都落入更大的子序列，则：

$$T(n) = \begin{cases} T(\max\{0,n-1\}) + \Theta(n) & \text{两子序列长度比为 } 0:(n-1) \\ T(\max\{1,n-2\}) + \Theta(n) & \text{两子序列长度比为 } 1:(n-2) \\ T(\max\{n-1,0\}) + \Theta(n) & \text{两子序列长度比为} (n-1):0 \end{cases}$$

利用我们定义的标示随机变量 X，我们可以将 $T(n)$表示为：

$$T(n) = \sum_{k=0}^{n-1} X_k \ (T(\max\{k, n-k-1\}) + \Theta(n))$$

对式子两边取期望值，则有：

$$\begin{aligned} E[T(n)] &= E\sum_{k=0}^{n-1} X_k (T(\max)\{k, n-k-1\}) + \Theta(n))] = \sum_{k=0}^{n-1} E[X_k (T(\max)\{k, n-k-1\}) + \Theta(n))] \\ &= \sum_{k=0}^{n-1} E[X_k] E[T(\max)\{k, n-k-1\}] + \Theta(n)] \\ &= \frac{1}{n}\sum_{k=0}^{n-1} E[T(\max)\{k, n-k-1\}] + \frac{1}{n}\sum_{k=0}^{n-1} \Theta(n) \\ &\leqslant \frac{2}{n}\sum_{k=n/2}^{n-1} E[T(k)] + \Theta(n) \end{aligned}$$

现在，剩下的工作是求解递归式 $\frac{2}{n}\sum_{k=\frac{n}{2}}^{n-1}E[T(k)]+\Theta(n)$。那么该递归式解是多少呢？

显然，这个递归表达式的形式不能用大师解法进行求解。画递归树也不太容易。剩下的办法是替换法。而替换法需要先猜出一个解，然后予以证明。我们的猜测是什么呢？

根据我们前面对随机次序选择的分析，它只有在最坏情况下时间复杂性是平方级，其他情况都是线性级。因此，我们的猜测很自然地是 cn，即 $E[T(n)] \leqslant cn$，这里 c 是一个正常数。

将该猜测代入到递归表达式，则有：

$$
\begin{aligned}
E[T(n)] &\leqslant \frac{2}{n}\sum_{k=\frac{n}{2}}^{n-1}ck + \Theta(n) && \text{（代入猜测解）}\\
&\leqslant \frac{2c}{n}\left(\frac{3}{8}n^2\right)+\Theta(n) && \text{（使用不等式 } \sum_{k=\frac{n}{2}}^{n-1}k \leqslant \frac{3}{8}n^2\text{）}\\
&= cn-\left(\frac{cn}{4}-\Theta(n)\right) && \text{（将表达式表示成需要的形式）}\\
&\leqslant cn && \text{（选择}c\text{足够大，使得 } \frac{cn}{4} \text{ 渐近大于}\Theta(n)\text{）}
\end{aligned}
$$

因此，随机快速次序选择算法的时间复杂性的期望值为线性。虽然最坏情况下是平方级，比很多排序算法的成本还高，但它在实际使用中效率很高。

10.14　最坏情况下的线性选择算法

虽然随机快速次序选择算法在实际使用中效率很高，但对于吹毛求疵的人来说，一个自然的问题是，是否存在一个算法能够保证最坏情况下的时间复杂性也是线性呢？

根据前面的分析，对于随机快速次序选择算法来说，只要不是每次都运气很差，就可以保证线性时间效率。而运气不差的关键就是杠杆点选择的不是数组里最小或最大的元素！即如果我们能够保证不选择序列里最大或最小的元素作为杠杆点，那么我们就能保证随机快速次序选择算法的时间成本为线性。

有没有办法保证我们随机选择的杠杆点不选到最大或最小值呢？答案是肯定的。这就是我们下面要讨论的**最坏情况下的线性选择算法**。该算法由 Blum、Floyd、Pratt、Rivest 和 Tarjan 在 1973 年提出。该算法的核心就是尽量选择一个靠中间的杠杆点，其步骤如下，

```
LINEAR-SELECTION(i, n)                    //在 n 个元素里面找出第 i 小的元素
1. 将 n 个元素分解为 5 元组，直接用蛮力找出每个 5 元组的中值；
2. 在获得的所有中值里递归找出它们的中值 x；
3. 将元素按照最后获得的中值进行分解，设 k=rank(x)；
4. If(i==k)
```

```
        return x;                    //已经找到所需元素，结束算法
5. else if(i==k)
                                     在小于 x 的部分递归选择第 i 小的元素
6. else
                                     在大于 x 的部分递归选择第 i-k 小的元素;
```

图 10-17 至图 10-19 描述了上述算法中的杠杆点选择过程。图 10-17 描述的是将所有元素分为多个 5 元组的情况（该图里有 8 个 5 元组，多余的 2 个元素忽略不计）。图 10-18 描述的是在每个 5 元组里找出中值（图中的浅灰色点）。图 10-19 描述的是在前面找出的所有中值里再寻找中值。这个中值就是我们要寻找的杠杆点（图 10-19 中的 x）。这样选择的杠杆点不可能是一个序列里面的最大和最小值。

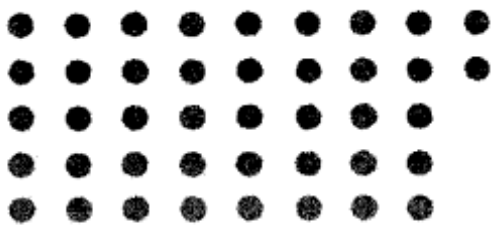


图 10-17 将元素分为 5 元组

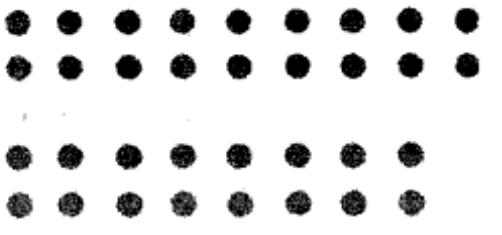


图 10-18 求取 5 元组的中值

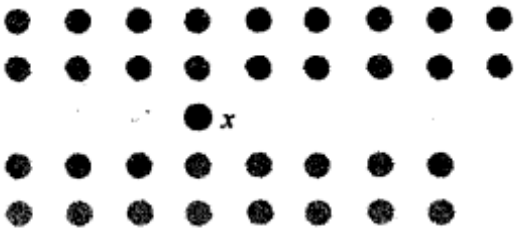


图 10-19 求取 $n/5$ 个中值的中值

10.14.1 杠杆点好坏分析

我们说了，上述选择算法不会将一个序列里的最大或最小值选为杠杆点，为什么呢？

根据杠杆点的选取方法可知，$n/5$ 个中值中至少有一半小于等于杠杆点，即$(n/5)/2=n/10$ 个元素小于等于杠杆点（见图 12-20）。因此，整个序列里面至少有 $3n/10$ 个元素小于等于杠杆点（见图 12-21）。同理，至少 $3n/10$ 个元素大于等于杠杆点（见图 12-22）。

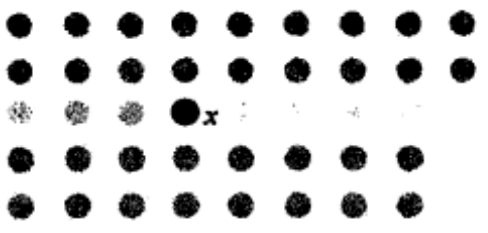


图 10-20 $n/5$ 个中值的一半$\leqslant x$

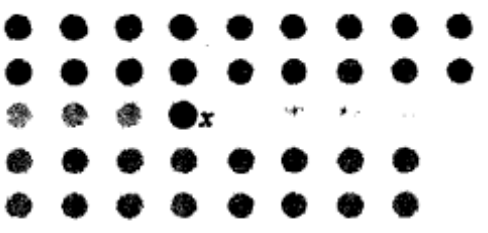


图 10-21 所有元素中 $3n/10\leqslant x$

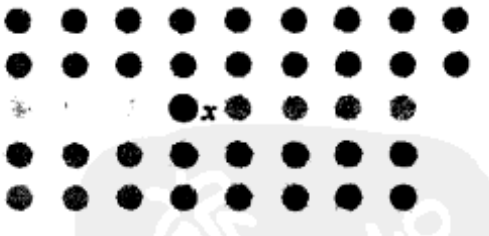


图 10-22 所有元素中 $3n/10\geqslant x$

因此，这样选择出来的杠杆点 x 对序列进行分解，两个子序列里面最短的一个也有 $3n/10$ 个元素，从而不可能出现一边子序列为空的情况。例如，对于 $n\geqslant 50$ 来说，$3n/10\geqslant n/4$。因此，对于 $n\geqslant 50$ 个元素来说，递归步骤应用的范围不超过 $3n/4$ 个元素。

10.14.2 算法时间复杂性分析

最坏情况下的线性选择算法的时间复杂性由下面几个部分构成：

第 1 步 分解为 5 元组：$\Theta(n)$。

第 2 步 在 5 元组递归选择杠杆点：$T(n/5)$。

第 3 步　　以杠杆点进行分解：$\Theta(n)$。

第 4、5、6 三步　递归次序选择：$\leqslant T(3n/4)$。

因此，算法时间复杂性的递归表达式为：

$$T(n)\leqslant T(n/5)+T(3n/4)+\Theta(n)$$

猜测 $T(n)\leqslant cn$，并以替换法求解，有：

$$T(n)\leqslant cn/5+3cn/4+\Theta(n)=19cn/20+\Theta(n)=cn-(cn/20-\Theta(n))\leqslant cn$$

上述不等式的成立条件是选择足够大的 c，使得 $cn/20\geqslant\Theta(n)$。

虽然该算法的最坏情况下的时间复杂性是线性级，但它在实际中的运行效率却不如前面的随机快速次序选择算法。这是因为该算法效率虽然为 cn，但常数系数 c 的值很大！另外，该算法的思路也比快速次序选择算法复杂，比较难理解。因此，这个算法除了在无聊的时候锻炼一下脑筋外，几乎没有什么用处。

也许这就是渐近表示与分析带给我们的误区？

思考题

1. 证明：只要分解的时候产生的两个子序列的长度比为常数级，即 $c_1<L_1/L_2<c_2$，这里 L_1、L_2 分别为两个子序列的长度，c_1、c_2 为任意的正常数，则快速排序的时间复杂性为 $n\log n$。
2. 既然插入排序的时间复杂性是我们所学排序方法里面的最高的，为什么人们在打牌的时候却使用它而不使用诸如快速排序、归并排序等其他渐近效率更高的排序方法呢？
3. 如果在应用归并排序的时候不是分解为 2 个子序列，而是 3 个、4 个或 5 个，情况有何变化？为什么？
4. 如果待排序的数列里面有很多相同的元素，例如，30%的元素是相同的，你能否为快速排序设计一个更好的分解算法？
5. 桶排序的辅助排序算法使用的是插入排序，而插入排序的时间复杂性是平方级。除了本书给出的一个理由外，还有什么理由让我们选择插入排序呢？
6. 如果使用一个 $n\log n$ 级的排序算法作为桶排序的辅助算法，桶排序的时间复杂性会是多少呢？如果使用一个线性排序算法作为辅助算法，情况又怎么样呢？
7. 本章提到过，如果输入元素不是均匀的，则桶排序的时间复杂性分析不能成立。但桶排序的算法正确性是否还成立呢？
8. 本章给出的次序选择算法是用于任意位置元素的选择。如果我们只是想选择第 1 个或最后一个位置的元素，你有什么更加简便的办法吗？它们的时间复杂性是多少呢？
9. 如果我们同时需要求取最大和最小元素，你能设计出什么好办法吗？
10. 为什么不分解为 3 元组呢？
11. 在次序选择问题上，有人提出一个更简单的办法：既然我们只要杠杆点不是最大或最小元素即可，那何不如先用简单的办法求取最大和最小值，然后随机选择一个元素，将此

元素与最大、最小两个元素比较，如果不相等，则以其作杠杆点，否则另外选择一个元素作为杠杆点。这样可以保证每次选择的杠杆点不会是最大元素和最小元素，从而保证随机次序选择算法的时间复杂性为线性。你对此建议有何评价？为什么？

12. 市面上流行一种儿童学习中国文字的工具叫“中华字经”，该字经宣传将中国的 8 000 个文字全部编成成语的形式，朗朗爽口，容易让儿童背。而更为重要的是，所有这些成语里面居然没有一个重复的汉字。尤尔不相信这点，于是就一个个成语进行检查对比，看看到底有没有重复的汉字。不过没坚持多久，尤尔就有点受不了了，因为这个比较实在太花工夫。你能否帮助尤尔设计一个算法来检查中华字经里面是否存在重复的汉字？你的算法的时间成本是多少？
13. 证明：递归表达式 $T(n) = T[\text{ceiling}(n/2)]+T[\text{flooring}(n/2)]+ \Theta(n)$的解是 $\Theta(n\log n)$。
14. 证明：递归表达式 $T(n)= T(n/10) +T(9n/10)+ \Theta(n)$ 的解是 $\Theta(n\log n)$。
15. 如果一个数组 $A[1...n]$有超过一半的元素相同，则该数组有一个优势元素。请设计一个算法以最高的效率判断一个数组是否存在优势元素。如果存在，则找出这个元素。数组里的元素之间不一定存在序关系（偏序或全序），即 $A[i]>A[j]$这种比较是没有意义的。但是，我们可以在常数时间内判断 $A[i] =A[j]$是否成立。你能设计出一个线性时间算法吗？
16. 假如某包括 n 个元素的数组的元素只有 100、1 000 和 10 000 三个取值，你可以多快将数组排好序？你的算法的空间效率又是多少？

CHAPTER 11

第 11 章　搜索与散列

人类自从诞生那刻起就开始了搜索：搜索水源、搜索食物、搜索财富、搜索名誉。

说搜索是人类所有活动中最为重要的恐怕并不过分。我们很多其他活动都是为了搜索而做准备。本书第 10 章讨论的排序也只不过是为搜索打基础。没有搜索，排序本身也就毫无意义了。

从哲学的角度看，搜索还是人类面临的永恒话题（见图 11-1）。正如英语里的一句俗语：In life we all search for something（在生命的过程中我们一直在搜索某种东西）。但是，能否搜索到我们想要的东西，或者多快和使用多少成本搜索到我们想要的东西则完全是另外一回事。

图 11-1　搜索似乎是人类生命中不可缺少的活动

也许对于大部分人来说，搜索就像美国的名歌手凯伦·卡朋特（Karen Carpenter）的歌所唱的：

经年来的徒劳搜索终于走到尽头

寂寞空虚将是我剩下的唯一朋友

All the years of useless search have finally reached an end

Loneliness and empty days will be my only friend

确实，大多数人都会在某个时候对搜索感到厌倦或失去信心，只有对少数人来说，搜索才是一个充满感悟和盼望的生命旅程。我们撰写本章就是希望算法里的搜索和搜索算法令读者感到的不是徒劳无益后的失望，而是感悟和欣喜。

11.1 搜索问题

每个人都知道搜索是什么意思：在衣橱里寻找一件合适的衣服，在学生名单中查找一个特定的学生，在互联网上搜索一个特定的术语，在茫茫人海里搜索自己的另一半等。

将所有的搜索问题进行抽象，就可以得出算法里面搜索问题的一般定义：

搜索问题 给定一个 n 个元素的序列或集合，在该序列或集合里面搜索一个特定的元素。

严格地说，上述问题应该归类为查找问题，因为我们要做的是在一个序列或集合里面查看是否存在某个特定的元素。这和一般的搜索并不完全相同，因为搜索并不一定要事先知道搜索的是什么。事实上，查找是搜索的一个特例。不过，本书不打算在这两个术语的语义上纠缠，而是将搜索和查找看做语义相同的两个术语，并随意地交换使用。

很多时候，序列或集合里的元素并不是一个纯数值，而是一个包含多个域的记录。此时搜索问题就变成搜索一个特定的记录。而用来搜索的**标的**（标准）就是记录里面某个域的取值。例如，我们在搜索人事档案的时候，搜索的目标当然是某个特定人的档案文件，而搜索用的标的则既可以是人名，也可以是某种（档案）编号，如图 11-2 所示。

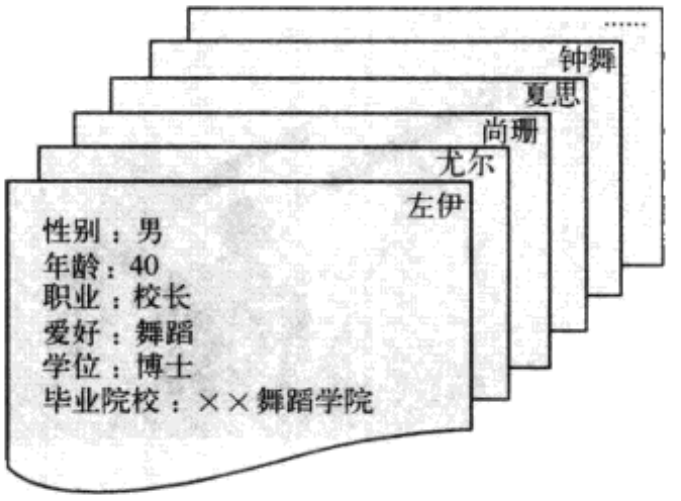


图 11-2 搜索就是在一堆记录里面寻找需要的记录

搜索的结果只能有两种：成功和失败。如果搜索成功，则返回匹配记录在序列里的位置，当然也可以是匹配记录本身；如果搜索失败，即序列里面不存在这样的记录，则报告“失败”。

从上述描述可知，每一个搜索函数都有两个输入参数：搜索的范围（序列）和要搜索的标的值（如记录某个域的取值）；在搜索成功时，有一个输出：匹配记录本身或匹配记录在序列里的位置；在不成功时，报告搜索失败。

由于所有的事物在计算机中都以数的形式表示，因此，对数的搜索就可以代表所有的搜索问题。因此，本章使用更为简单的搜索问题定义：在一组数中搜索一个给定的数。如果这个数存在，则返回该数所在的位置；否则，报告搜索失败。

11.2　顺序搜索

我们如何从一组数里搜索一个特定的数呢？显然，最直接，也是人人都能想到的办法是顺序扫描整个序列！也就是从序列的一端开始，按自然顺序一个数一个数地查找比对，直到找到要寻找的数或搜索抵达序列的末端为止。

这种搜索就是所谓的顺序搜索（sequential search）。

顺序搜索的算法非常简单，其伪代码实现如下：

```
SEQUENTIAL-SEARCH(S[1..n], x)        // 在序列 S 里面搜索元素 x
1. for(i=1; i<=n; i++)
2.     if(x==S[i])
3.         return i;
4. return not_present;
```

顺序搜索的正确性显而易见：由于顺序扫描过来，它不可能错过任何一个元素。因此，只要搜索的标的 x 存在于序列里，顺序搜索就一定会搜索到。

顺序搜索的时间复杂性与所寻找元素在序列中的位置有关。如果元素不存在于序列中，则需要搜索 n 个元素；如果元素存在于序列中的第 1 个位置，则只需要 1 次搜索；如果元素在序列末尾，则需要搜索 n 次；而平均来说，一次成功的搜索需要搜寻 $n/2$ 个记录。因此，顺序搜索在最好情况下的时间复杂性为$\Theta(1)$，在最坏和平均情况下的时间复杂性均为$\Theta(n)$，而这是线性的！

不过先别高兴。虽然到目前为止我们一直认为线性时间复杂性是非常好的一个数量级，但在搜索上却不是这样。事实上，顺序搜索是所有搜索算法里面最慢的！在最坏情况下的搜索不就是每个记录都看一遍吗？难道还有什么搜索需要对每个记录看上两遍或三遍的吗？因此，线性搜索时间是任何搜索方法和地球人都能达到的！

不过，尽管如此，顺序搜索也不是没有用处的。首先，如果序列里的元素排列没有任何规律可循，则顺序搜索是我们唯一能使用的办法；其次，如果序列里的元素是按照访问频率从高到低排列的，则顺序搜索不失为一种好的算法。

例如，如果序列里元素被访问的频率从开头至末尾按指数频率递减，即

$$p_i = \begin{cases} \dfrac{1}{2^i} & 1 \leqslant i \leqslant n-1 \\ \dfrac{1}{2^{n-1}} & i = n \end{cases}$$

则顺序访问的搜索成本期望值为：

$$\bar{C}_n \approx \sum_{i=1}^{n} \frac{i}{2^i} \approx 2$$

而这是常数时间！任何算法的效率都敌不过常数，除非是上帝，不用时间。

11.3 折半搜索

如果序列中的元素是按照访问频率由高到低排列的，顺序搜索也许是一个很好的搜索算法。但大多数时候序列里的元素并不是按照这个规律排列。而且很多时候，我们并不知道一个元素被访问或使用的频率或者获得此种数据的成本很高，自然也就无法按照频率进行排序。

因此，在通常情况下，顺序搜索的时间复杂性仍然是$\Theta(n)$，而这是任何搜索都能够达到的时间效率。那么能否对搜索算法进行改进呢？

前面说过，如果序列里元素的排列没有任何规律可循，则顺序搜索是我们唯一的武器。但问题是我们可以使用排序来对序列里的元素进行排列，从而使序列呈升序或降序排列。而对这样一个有序的序列来讲，进行顺序搜索就显得不合时宜了。事实上，我们排序的目的就是为后面的搜索做铺垫。如果一个排好序的序列不会再被搜索，则排序就是白白浪费时间。而如果排好序后我们还使用顺序搜索，则排序就变成一种莫大的讽刺了！

那么在序列元素有序排列的情况下，我们怎么搜索呢？

记得怎样在图书馆查找所需要的图书吗？当你获得你需要的图书的编号后，你就在书架上查找这本书。你是将书架上的书一本本地翻看查找呢，还是看到一本书的编号后，跳过一部分书再查看呢？显然，大部分人都知道应该跳跃性地查找。例如，先检查整排书里最中间的书，如果该书的书号比要查找的书号大，则往左搜索；如果该书的书号比要查找的书号小，我们则往右搜索；如果该书的书号就是要查找的书号，则恭喜你成功！

这种搜索算法就是算法里面的折半搜索（binary search）。

顾名思义，折半搜索就是一次将搜索的范围减掉一半！搜索算法中的折半搜索就是将目标与序列正中间的记录进行比较，如果相等，则搜索成功。如果目标小于中间点，则将搜索范围缩小到中间点前面的半个序列里；如果目标大于中间点，则将搜索范围缩小到中间点后面的半个序列里。然后在缩小的搜索范围内再使用折半搜索。图 11-3 描述了折半搜索。

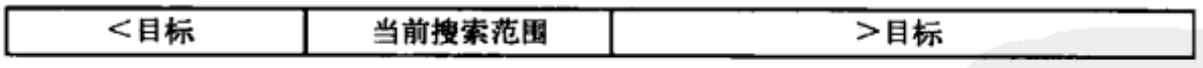


图 11-3 折半搜索示意：每次搜索折掉半个序列

下面为折半搜索的伪代码实现：

```
BINARY-SEARCH(S[p..q], x)          // 在有序序列 S 里面搜索元素 x
1.  if(p<q) {
2.      if(x==S[(q-p)/2])
3.          return (q-p)/2;
4.      else if(x<S[(q-p)/2])
5.          return BINARY-SEARCH(S[p..(q-p)/2-1], x);
6.      else
7.          return BINARY-SEARCH(S[(q-p)/2+1..q], x);
```

```
8.  }
9.  else if (x==S[q])
10.       return q;
11. else
12.       return not_present;
```

由于每次搜索都在范围缩小一半的空间进行，因此搜索的成本可以表示为：

$$T(n)=T(n/2)+\Theta(1)$$

根据大师解法，$T(n)=\Theta(\log n)$。相对于$\Theta(n)$的顺序搜索来说，这是一个巨大的改进。

11.4　常数搜索

对于很多人来说，折半搜索的对数级$\Theta(\log n)$时间复杂性是可以接受的，似乎没有必要再折腾了。但问题是，折半搜索虽然有着对数级的时间复杂性，但它只能用在有序序列上。问题是一个序列怎么会有序呢？当然是我们排序的结果。而排序需要 $O(n\log n)$的时间（比较排序）。这样一来，折半搜索的综合成本实际上是高于$\Theta(\log n)$的！当然，随着搜索次数的增加，平摊到每次搜索上的排序成本将不断降低（还记得摊销分析吗），但这个成本总是存在的！

更为严峻的是，搜索问题通常具有两个特点：一是搜索的范围很大，二是搜索是一个频繁的操作，即经常要执行的任务（人类不是一直在搜索人生的意义吗）。这样，$\Theta(\log n)$级的时间复杂性就显得有点高了。因为如果 n 非常大，如趋近无穷，$\log n$ 就会很大，而如果搜索执行的频率又很高（你不是经常在东翻西找吗），则搜索的聚类成本就相当高了（记得聚类分析吗）。例如，编译器在编译程序的过程中，绝大部分时间都是在搜索：搜索符号表、搜索常数表等。而对于需要频繁查找的编译器来说，$\log n$ 的时间复杂性实在是太高了！

因此，探索更有效的搜索算法就是一件非常必要的事情。而要进行此种探索，就需要知道搜索的效率上限在哪里，或者说搜索的成本下限在什么地方。

仔细分析搜索问题发现，我们并不需要知道要搜索的元素与其他元素的关系，即我们在搜索元素 x 的时候，并不需要知道 x 和其他元素相比较究竟处于什么位置上，或者有多少元素比 x 小，又有多少元素比它大。这些信息虽然能够帮助我们进行搜索，但并不是搜索所必需的！即从根本上说，搜索一个元素只是这一个元素的事情，并不涉及别的元素。也就是说，从理论上讲，我们应该可以设计出常数时间的搜索算法！到目前为止，我们设计的搜索算法达不到常数级并不是搜索问题本身的理论限制，而是我们设计的思路有问题。

这里需要提醒读者注意的是，搜索与排序不同，排序必须考察每个元素才有可能进行排序；也与次序选择不同，次序选择涉及一个元素和其他元素之间的大小关系，因此也需要对其他元素进行考察。基于这些原因，排序和次序选择都无法在效率上超越线性级。

既然理论上并没有限制我们获得常数搜索时间，那么我们当然应该奋力拼搏一番了。从上述讨论可知，要想获得常数搜索时间，整个搜索过程不应该与非搜索目标产生联系。如何办到这一点呢？

11.5 散列搜索

这是一个司空见惯的情景：你去某个机构拜见该机构的领导，当你告诉彬彬有礼的前台小姐，你是来见她的顶头上司时，前台小姐会熟练地拨通该领导的电话询问是否属实并做出相应安排。也许这个情景太过熟悉，并不能勾起你任何联想（当然我们这里指的仅仅是算法上的联想），但这个情景却与我们要讨论的常数时间搜索十分相关（见图 11-4）！

难道你没有觉察到，前台小姐在拨通领导电话时并没有去查看电话号码本吗？也就是说，领导的电话记在她的脑子里了。从搜索的角度来看，她在搜索领导电话时，用的是常数时间，或者说，她根本就没有进行我们所熟知的东翻西找的那种搜索！而这就给我们设计常数搜索算法提供了启示：想办法记住每个元素所在的地方！

这种记住每个元素所在地方的搜索方法就是散列搜索。

图 11-4 前台小姐每天做的工作隐藏着散列的思想

散列是目前唯一一种期望时间成本为常数的搜索算法。显然，做任何事情都需要时间，而最快的就是常数时间了。因此，散列搜索是所有搜索算法里面效率最高的！

简单地讲，散列是通过某种函数将数据元素映射到一个特定的位置，如数组索引（整数）上。这相当于前台小姐将领导的电话号码记在脑子里！这个映射函数称为**散列函数**，这个数组称为**散列表**。在需要查找某个数据元素时，只需将该元素作为变量代入到散列函数里，即可求得该数据元素所在的散列表（数组）索引，从而找到该数据元素。如果对应该索引的散列表项内容为空，则说明该记录不存在。散列插入操作同样简单：将需要插入的数据元素作为变量值代入到散列函数中，即可求得该数据元素所应当插入的散列表位置（索引）。

散列函数

散列函数是散列算法的核心，它将一个数据元素映射到一个整数值上，也就是散列表里

面的一个索引上。如果散列表一共有 M 个位置，则索引值的范围为 0 到 M-1。对于任何处于允许范围内的数据元素 K，有 $h(K)=i$（$0 \leqslant i<M$），这里 h 是散列函数。如果 U 代表潜在的数据元素集合，则散列函数 h 将集合 U 映射到集合{0, 1, …, m-1}上，如图 11-5 所示。

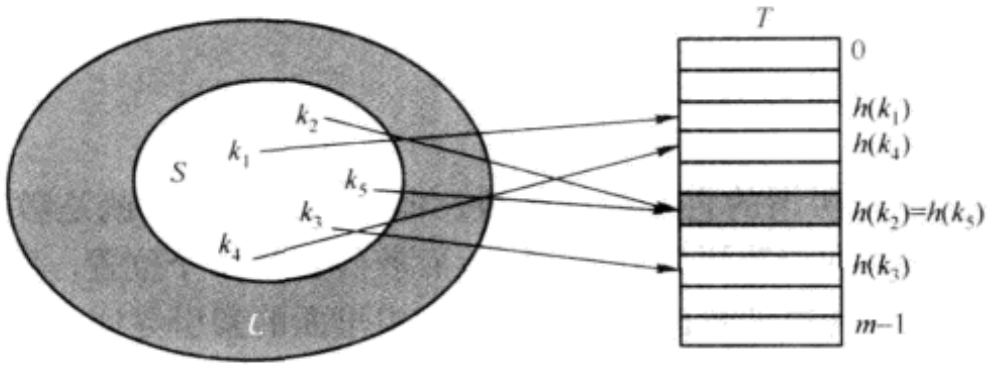


图 11-5　散列就是将数据元素集合 U 映射到散列表 T 上

如果需要散列的元素个数小于等于散列表的大小，则理论上可以做到对于不同的关键值，其散列结果也不同。但如果需要散列的元素个数大于散列表的大小，则无论如何都将发生碰撞，即两个或多个元素被散列到同一个索引上。例如，在图 11-5 里，数据元素 k_5 和 k_2 就发生了碰撞，二者都被散列到同一个散列表位置！此时，我们就需要采取其他手段来解决碰撞。而碰撞的频率则决定了一个散列函数的好坏。碰撞越多，散列的效率将越加低。如果一个散列函数导致大量的碰撞，从而使得大部分元素聚集在一个或少数几个散列位置上，则我们说该散列函数呈现出**聚集效应**。显然，聚集效应将大大降低散列的效率。因此，选择一个好的散列函数对散列算法的效率有着重大影响。

如何选择或设计好的散列函数呢？要回答这个问题，需要知道什么是好的散列函数，或者说如何评判一个散列函数的好坏。很显然，一个好的散列函数应该产生尽可能少的碰撞。如果碰撞不可避免，则碰撞的发生也应该均匀到每个位置上，即每个散列位置接纳的元素个数应该相等或相近。也就是说，散列函数应该将所有元素均匀散射到散列表里面。

均匀散射的意思是，如果 x、y 是任意两个待散列的元素，散列表大小为 M，则散列函数 h 将 x 和 y 散列到同一个位置的概率是 $1/M$。或者说，如果函数 h 将 x 和 y 散列到同一个位置的概率是 $1/M$，则该函数 h 就是一个好的散列函数，而此种散列行为称为**均匀映射**或**均匀散列**。显然，均匀散列将不会产生聚集效应。

11.6　散列函数选择

散列函数选择的目标就是使得所有元素均匀散射到散列表里，而且这种均匀性不应该受到数据元素取值分布的影响。如果我们对数据元素的取值分布毫不知情，则可以选择将元素的**取值范围**均匀映射到散列表的全部空间上，同时防止任何明显的元素聚集（很多元素集中在散列表的一个局部空间里）。如果我们知道元素的取值分布，则可以选择有针对性的散列函数。

选择散列函数的另外一个考虑是散列函数本身不应该太复杂，我们当然不想在每次计算散列函数值时花费很多的工夫。如何找到既简单又均匀的散列函数呢？

虽然不能保证，但一些技巧可以帮助我们选择简单却又比较接近均匀散列的函数。这些散列函数包括直接散列、除法（模除法）散列、乘法散列和乘方散列。

11.6.1 直接散列

最简单的散列函数是所谓的直接散列函数，即数据元素本身就是散列表位置索引。例如，如果 n 个数据元素的取值范围为 0 到 n-1，而散列表有 n 个位置，则可以使用直接寻址散列函数：$h(K) = K$。也许读者看出来了，这与数组里面的索引寻址（直接用索引来访问数组）非常相似（注意是相似，不是相同）。例如，元素 10 在散列表的位置就是 10 号位。

直接散列的优点是散列函数的计算非常简单：没有计算！因为没有计算，所以散列的过程非常迅速。而更为重要的一个优点是不会发生碰撞。因为元素取值本身是散列表索引，两个元素只有在它们的取值一样的情况下才会发生碰撞，而取值一样的元素当然可以看做是同一个元素。

尽管有上述优点，直接散列在很多时候却并不适用（否则用数组就行了，何必发明散列）。例如，我们需要存放 1 000 个记录，这些记录的取值范围为 0～36 728。如果使用直接寻址，则需要一个有 36 728 个位置的散列表，而这显然是不现实的，而且也很浪费，因为要存放的记录个数只有 1 000 个！因此，理想的情况是散列表的规模为 1 000 个存储单元，但所有 0～36 728 范围的整数都能被散列函数转换为 0～999 的某个值。例如，我们可以设计散列函数 $h(K) = K \bmod 1000$，这样所有非负整数都将被转换为 0～999 的某个值。

11.6.2 除法（模除法）散列

除法散列当然是在散列函数的计算中使用了除法。如果所有的元素都为整数，散列表大小为 m，我们可定义散列函数 $h(k) = k \bmod m$。如果所有元素的取值分布均匀，且 m 的取值也经过仔细斟酌，则这种方法将所有元素的取值范围均匀映射到散列表的所有位置上。例如，m=20 和 k=91，则 $h(k) = 11$，即元素 91 在散列表的位置为第 11 个位置。

除法散列的优点是简单、速度快，因为只需要进行一次除法运算（当然不如直接散列快，但也差不多）。但缺点是如果散列表大小 m 选择得不好，容易造成元素聚集在散列表的某些部分。例如，如果 m 包含一个大于 1 的小因子 d，且大部分元素的模 d 余数相等，则除法散列函数将达不到均匀映射。例如，如果 m 为偶数（包含小因子 d=2），而所有的元素都是偶数（模 2 余数相等），则散列表的所有奇数位置都将为空！

更极端的情况是 m 为 2 的幂值，即 $m = 2^r$，则散列结果甚至并不依赖于元素 k 的所有有效字位。例如，散列函数 $x \bmod 16$ 的取值将完全依赖于数值 x 的最低 4 个有效位的取值，而其他有效位均不起作用。如果 $k = 1011000111011010_2$，$r = 6$，则 $h(k) = 011010_2$。如果 k 是字符串，并将其看做一个进制为 2^p 的数的话，则 $m = 2^p-1$ 将是非常坏的选择！因为对字符串里的字符进行重新排列将不改变其散列值！由此可见，散列表的大小 m 不能太靠近 2 的幂值！

m 既不能包含较小因子，也不能太靠近 2 的幂值。而能够保证没有较小因子的 *m* 当然是素数！因为只要选择的素数不靠近 2 或 10 或任何常用进制的幂值，除法散列将取得（比较）均匀散射，即均匀散列的效果。

只不过这样选择的 *m* 有一个小问题：构造一个素数个位置的散列表并不方便，因为计算机在分配空间的时候均是按 2 的幂值进行的！

尽管如此，此种方法还是非常受欢迎，因为此种散列函数可以写在一个语句里。例如下面的实际 C 程序片段（不是伪代码）就在 return 语句里包含了一个除法散列函数：

```
int h(char* x) {
    inti,sum;
    for (sum=0,i=0;x[i]!='\0';i++)
        sum+=(int)x[i];
    return(sum % M);
}
```

该程序片段的功能是将一个字符串映射到一个大小为 *M* 的散列表上。它首先将字符串里每个字符的 ASCII 值累加，然后在返回语句 return 里面进行 *M* 的求模运算，从而获得散列值。

UNIX System V Release 4 里的 ELF（Executable & Linking Format）散列函数就包含了除法散列，下面为 ELF 散列的具体 C 程序实现（不是伪代码）：

```
int ELFhash(char* key) {
    unsigned long h = 0;
    while(*key) {
        h = (h << 4) + *key++;
        unsigned long g = h & 0xF0000000L;
        if (g) h ^= g >> 24;
        h &= ~g;
    }
    return h % M;
}
```

上述程序片断的 ELFhash 也是将一个字符串（key）映射到一个大小为 *M* 的散列表里。只不过除法散列并不是此散列程序所唯一使用的手段。读者看出来整个散列函数的构造了吗？

11.6.3　乘法散列

除法散列虽然简单、快速，但需要将散列表大小设计为一个素数值。由于计算机分配空间都是按照 2 的幂值进行，我们自然希望设计一种散列函数，它能够与大小为 2 的幂的散列表很好配合。除法散列显然不能胜任这一任务，所以我们将注意力转移到乘法上面。

假定所有元素都是整数，散列表大小为 2 的幂值，$m=2^r$，计算机的字宽度为 w 位。

我们定义乘法散列函数如下：

$$h(k)=(Ak \bmod 2^w) \text{ rsh } (w-r)$$

这里的 rsh 表示右移位操作，其后面的参数为右移的字位数；A 为一个奇数，取值范围为 $2^{w-1}<A<2^w$，并且不能很接近 2^{w-1} 或 2^w。（为什么？）

例如，$m=8=2^3$，$w=7$，$A=1011001$。如果 $k=1101011$，则 k 的散列值计算如下：

$$\begin{array}{rr} & 1011001 \quad A \\ \times & 1101011 \quad k \\ \hline & 1001010\mathbf{011}0011 \quad A\times k \end{array}$$

最后一行加黑的三位就是元素 k 的散列值，也就是 k 将被散列到散列表的第 3 个位置上。这是我们费了半天力气获得的结果，但这个结果（散列值 3）有什么稀奇的吗？

稀奇之处就是该散列值依赖于数据元素 k 的所有字位！（有没有看出来？）即每个字位都对散列值的取值产生了影响。这样，如果两个记录的取值不同，则不管其不同的字位在什么地方，其散列值都存在不同的可能！即不太可能发生聚集效应，至少不会经常或者有规律地发生！（而无规律地发生则是人所不能控制了的！）

另一点同样重要的是，对于计算机来说，模 2^w 的乘法运算速度很快，比起除法运算更快！而且右移操作也很快，尤其是对于提供了水桶移位器（barrel shifter）的计算机来说。因此，此种散列函数的计算效率很高。可见，此种散列函数着实有点说服力。

问题是该散列函数看上去很复杂！

虽然计算机计算起来不费吹灰之力，但一般人理解起来却经常颇费周折！而理解的难点在于：这样一个复杂的表达式是怎么想出来的呢？

11.6.4 乘法散列的赌徒原理

如果你是一个赌徒，就能很容易明白其中的道理（见图 11-6）。但如果你不是赌徒，那你至少从电视上见过所谓的轮盘转的赌博游戏吧？就是一个大转盘，转盘上刻有不同的格度，如 8 个或 16 个格度，每个格度都标有一个奖金额度或者奖品名字。参与赌博的人在众目睽睽之下用手使劲转动转盘（有人说不要使劲，这样获得大奖的概率更高）。如果你在观看这种轮盘赌的时候比较细心，一定能发现这个转盘的每次推动几乎总是停在一个不同的地方。只有在所有不同的格度都停过之后才会出现重复（即停在以前停过的刻度上），而这恰恰是我们进行散列所需要的！即每次散列都在一个不同的位置上，直到所有的散列表位置用完才出现重复（仅从概率上说）。

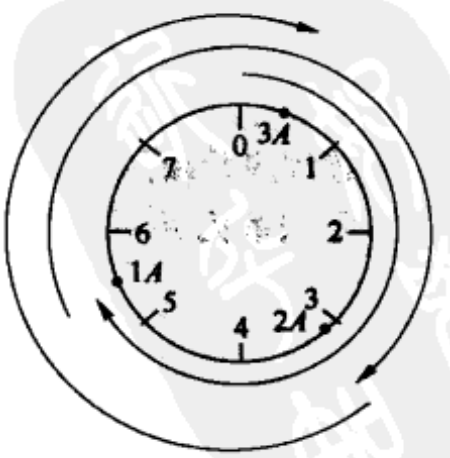


图 11-6 乘法散列函数借用的是轮盘赌原理

那么如何确保每次转动都让轮盘停在一个不同的地方呢？假定赌徒的推力可以将大转盘转动长度为 A 的旋转距离，则如果 A 的值与轮盘的圆周长值相对为素（即没有不等于 1 的公因子），且又不靠近轮盘格度大小的倍数时，就会出现我们上面描述的结果。而这正是乘法散列函数的精髓！也正是我们在选择 A 的取值时需要注意的事项！对于我们给出的乘法散列函数来说，k 的大小代表赌徒转动长度为 A 的距离的能力，即推力为 k 的赌徒可将转盘转动 $A\times k$ 的距离。而不同的元素就是不同的赌徒。虽然不同赌徒的推力大小各不相同（元素取值不同），但由于 A 与 M 相对为素，不同赌徒转动轮盘后的落点都将与前次转动的落点不同（仅从概率上说），直到所有落点都用完为止。这样就实现了散列函数的均匀散射！

由上述论述可知，A 的选择决定着乘法散列是否均匀。虽然满足条件的 A 值不只有 1 个（可能很多），但不是所有的 A 值都有一样的效果。根据输入的不同，不同的 A 值的效果也不尽相同。Donald Knuth 建议使用 $A\approx(\text{sqrt}(5)-1)/2$，即黄金分割点。

11.6.5　乘方取中法

如果觉得上面的乘法散列函数复杂，可以使用其一种简单的变种：乘方取中法。该方法将元素 k 进行 n 次乘方，然后取其中间的 r 个字位作为散列值（散列表大小为 $m=2^r$）。即

$$h(k)=k^n\text{mid}\ (r)$$

这里 mid (r)代表取中间的 r 个字位。它可以由左移和右移操作来实现。

为了限制散列函数本身的计算复杂性，乘方取中法里面的 n 通常限制在 2，从而成为平方取中法。假定 k 占用 x 个字位，则平方取中法的散列函数可以粗略表示为：

$$h(k)=(k^2\ \text{lsh}\ (2x-r)/2)\ \text{rsh}\ (2x-r)$$

这里的 lsh 和 rsh 分别表示左移和右移操作。该方法思路简单，并且实际效果也不错。

11.7　散列算法的碰撞问题

俗话说，人算不如天算。不管是直接散列函数、除法散列函数、乘法散列函数还是乘方取中散列函数，也不管我们在选择 M、A、n 或其他参数时多么小心，元素碰撞总是难以避免的。而对碰撞进行处理就成为散列算法不得不面对的一个重要问题。事实上，碰撞处理策略的好坏在很大程度上影响着一个散列算法的成败。碰撞处理的问题很简单：如果发生碰撞怎么办？

很显然，一旦发生碰撞，处理办法只有三个：

1）置之不理，散列到同一个位置的元素覆盖前面的元素。

2）纵深扩展，将散列表的每个位置扩展为一个链表，散列到同一个位置的元素以链表连接。

3）横向扩展，为发生碰撞的元素寻找另一个散列表位置。

第 1 种处理办法当然不能接受，因为它将导致散列搜索算法的整体失败。而第 2 种和第 3 种处理方法均可以解决碰撞问题，但它们对散列算法的影响却不相同。第 2 种纵深扩展处

理办法由于扩展了散列表空间，即散列表实际上不只有 M 个存储单元，而被称为开放寻址散列。第 3 种方式因为维持散列表大小不变，被称为封闭寻址散列。下面分别予以讨论。

11.7.1 开放寻址散列

开放寻址散列就是上面所说的第 2 种处理碰撞的办法：纵深扩展。该处理方法就是将发生碰撞的元素链接在一起，如图 11-7 所示。

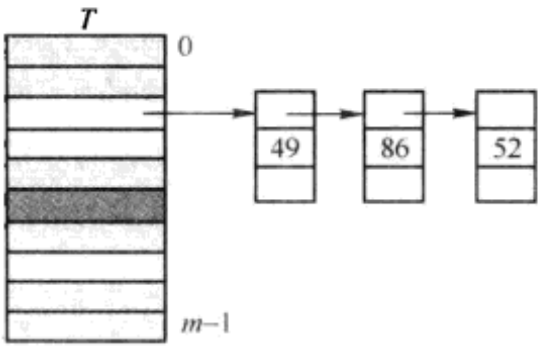


图 11-7 开放寻址碰撞解决方案：链接（$h(49) = h(86) = h(52) = i$）

这种解决方案的优点是散列表所占用的空间可以超出原来分配的空间，也就是说分配的散列表大小不是封闭的，而是开放的，即每个位置都是打开的，可以向外延伸。如果要增加散列的元素个数，只需要增加链表的长度，而不需要扩展散列表，从而避免诸多麻烦。

在开放寻址下，搜索数据元素 k 的过程如下：

1）首先计算数据元素 k 的散列值 $h(k)$。

2）获得位置 $h(k)$的首记录。

3）检查该记录是否等于 k。

4）如果相等，则成功结束搜索。

5）如果不等，顺着指针获得链表的下一个记录。

6）如果下一个记录为空，搜索失败；否则返回步骤 3。

插入过程与搜索过程非常相似。唯一不同的是需要顺着链表一直走下去，直到指针为空时为止，此时将待插入的元素插入到最后的空位上。当然，如果想节省插入成本，也可以在链表头进行插入。这样只需要计算 k 的散列值，然后将元素插入到对应该散列值的链表头上。

11.7.2 开放寻址散列的时间成本

我们前面说过，散列搜索是一个常数搜索：一步操作。不过这个结论的前提是没有碰撞！由于解决碰撞需要时间成本，因此开放寻址散列算法的时间成本就不只是一步操作那么简单啦。

那么开放寻址散列算法的搜索时间复杂性如何呢？

极端的情况是所有的数据元素都被散列到散列表的同一个位置。在此种情况下，搜索时

间为 $\Theta(n)$，这里 n 为散列的元素个数。此时，散列搜索算法退化为顺序搜索。

最好的情况是不同的元素被散列到不同的位置，此时，搜索只有一步操作，时间为 $\Theta(1)$。

但平均情况下的搜索时间为多少呢？显然，这个时间成本与散列函数的好坏有关。假设我们选择的散列函数是好的，即散列函数将所有元素均匀散射到散列表里。换句话说，任意元素 k 被散列到表 T 里任意位置的概率相等，且与其他元素散列到的位置无关。

设 n 为散列表里元素的个数，m 为散列表的大小，在均匀散列下，每个散列表位置平均接纳的元素个数为 $\alpha = n/m$，这个值也称为散列表 T 的加载因子。不成功搜索的期望时间成本为 $\Theta(1+\alpha)$，这里的 1 代表计算散列函数所需时间，α 代表搜索该散列位置上链表所有元素所需的时间。显然，如果 $\alpha=O(1)$（即 $n=O(m)$），则期望搜索时间为 $\Theta(1)$。

那么成功搜索所需的时间如何呢？直观上看，成功搜索似乎应该是计算散列函数的常数成本加上搜索链表长度的一半，即 $\Theta(1+\alpha/2)=\Theta(1+\alpha)$，也就是与不成功搜索的成本一样。

但这个结论是正确的吗？

11.7.3　开放寻址下成功搜索的时间成本

假设一共有 n 个元素，需要搜索的元素为 x，则在搜索 x 过程中需要检查的元素个数是：x 所在链表里处于 x 前面的元素个数+1。假设对链表的插入操作在链表头进行，则链表里处于 x 前面的元素都是在 x 后面插入到散列表里的。

因此，需要计算的任务就变成：平均来说，在 n 个元素里，有多少个元素是在元素 x 之后被插入到 x 所在的链表里的。为此，设 x_i 是第 i 个插入到散列表里的元素（$i=1,2,\cdots,n$），又设 $k_i=\text{key}[x_i]$，则对于所有的 i 和 j（$i \neq j$），我们定义标示随机变量

$$X_{ij}=\mathrm{I}\{h(k_i)=h(k_j)\}$$

则搜索 x_i 的成本就是 $1+\sum_{j=i+1}^{n} X_{ij}$。

考虑到所有元素被搜索的概率相同，我们得到搜索任意元素的成本为：

$$T(n) = \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} X_{ij}\right)$$

两边取期望值有：

$$E[T(n)] = E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} X_{ij}\right)\right] = \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} E[X_{ij}]\right) \qquad (11\text{-}1)$$

由于是均匀散列，有：

$$\Pr\{h(k_i)=h(k_j)\} = 1/m$$
$$E[X_{ij}] = 1/m$$

将上述结果代入式（11-1），有：

$$E[T(n)]=E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right]=\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}E[X_{ij}]\right)$$

$$=\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right)=1+\frac{1}{nm}\sum_{i=1}^{n}(n-i)$$

$$=1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right)=1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right)$$

$$=1+\frac{n-1}{2m}=1+\frac{\alpha}{2}-\frac{\alpha}{2n}$$

由上可知，我们获得一次成功搜索的期望时间成本为：$\Theta\ (1+\alpha/2-\alpha/2n)=\Theta(1+\alpha)$。由此可见，成功搜索的期望成本与不成功搜索的期望成本果然完全一样。也就说，平均情况下的开放寻址散列搜索成本为 $\Theta(1+\alpha)$。

11.7.4 封闭寻址散列

既然有开放寻址，当然也就有封闭寻址。开放寻址的优点是不需要改变散列表的大小 M，也不需要重新计算一个元素在散列表里的位置。但这种不需要改变散列表大小的优点是以在散列表外使用额外的链表为代价的。而这种做法在很多时候并不方便，至少不利于我们对程序所需空间进行估计。我们喜欢的是给定一个散列表，所有的需要都可以在这个空间满足。如果发生碰撞，也只能在散列表里另外寻找位置。

这种不使用散列表外空间的碰撞解决办法就是封闭寻址散列。在此种策略下，所有元素都必须在散列表的里面寻找位置进行存储。也就是说，散列表是封闭的，谁也出不到散列表外面去。

在封闭寻址散列下，每个元素 i 都有一个所谓的家位，即原始散列值 $h(k_i)$。如果这个家被另外一个元素占用，则需要在散列表里另外找一个位置给 i。而不同的寻找方法就导致了不同的封闭寻址散列算法，当然也导致了不同的散列搜索和插入效率。

在散列表里寻找另外位置的操作称为**探寻**。所有的封闭寻址散列算法都需要设计一个探寻策略。显然，探寻策略应对散列表的每个位置按一定顺序进行检查以搜索所需的元素或者寻找一个空位来进行插入。从抽象上说，探寻策略就是针对散列表的所有位置索引$\{0, 1, \cdots, m-1\}$设计一个排列 $h(k,0), h(k,1), \cdots, h(k,m-1)$，这里 $h(k,i)\neq h(k,j)$，$i\neq j$ 且 $0\leqslant h(k,i)\leqslant m-1$。这个排列就是在搜索和插入发生碰撞时寻找另外位置的次序。

如果搜索成功，散列搜索返回的是元素 k 在散列表的位置索引；如果搜索不成功，则返回 NIL。下面是封闭寻址散列算法的搜索伪代码程序：

```
CLOSE-ADDRESSING-HASH-SEARCH (T, k)    //T 是一个散列表，k 是要搜索的元素
1. i=0;
2. do {
3.       j=h(k,i);
4.       if(T[j]==k)
```

```
5.            return j;
6.       i=i+1;
7. } while (T[j]!=NIL && i!=m)
8. return NIL;
```

而散列插入则返回元素 k 被插入后的位置索引。如果散列表已满，则返回“溢出”。散列插入的伪代码程序如下：

```
CLOSE-ADDRESSING-HASH-INSERT (T, k)
1. i=0;
2. do {
3.     j=h(k,i);
4.     if(T[j]==NIL) {
5.           T[j]=k;
6.           returnj;
7.     }
8.     else
9.           i=i+1;
10. } while (i!=m)
11. return OVERFLOW;
```

例如，在封闭寻址散列算法下，如果我们要插入元素 $k=496$，则可能要进行如图 11-8 所示的探寻。

如果我们要搜索元素 496，则需要经过相同的探寻序列，如图 11-9 所示。

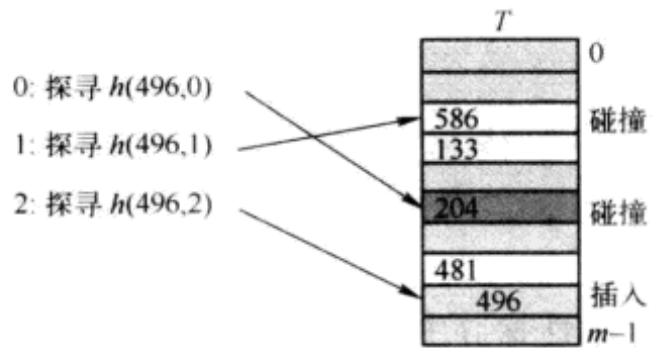


图 11-8　封闭寻址散列算法下，插入元素 496 经过 3 次探寻才找到空位插入

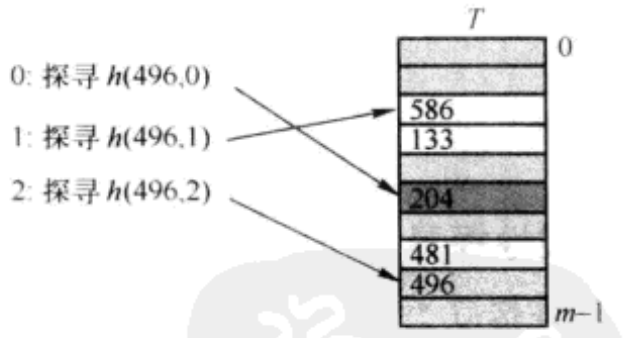


图 11-9　封闭寻址散列算法下，搜索元素 496 经过同样的 3 次探寻才找到

11.7.5　探寻序列的设计

显然，在封闭寻址散列下，探寻序列的好坏对散列效率的影响巨大。搜索或插入时探寻次数越多，散列效率就越低。因此，设计一个能够降低每次插入和搜索时的探寻次数的序列就非常重要。那么如何进行探寻呢？或者说如何计算一个合理的{0, 1, …, m–1}的排列呢？经过漫长的探索，人类总结出四大类的探寻方式：线性探寻、非线性探寻、双重散列和伪随机探寻。

1. 线性探寻

这是最直观的探寻方式。如果要插入或搜索的位置由其他元素占用，则直接查看相邻的位置。如果邻位还是被占用，则按同一方向检查该位的邻位，这样如此下去，直至找到空位或要搜索的元素，或整个散列表都被探寻过为止。即给定任意散列函数 $h'(k)$，实际上线性探寻使用的是在此基础上按线性函数构建的新散列函数：

$$h(k,i) = (h'(k) + i) \bmod m$$

线性探索虽然简单，但是问题也很明显，就是会出现顶级聚集（primary clustering），即连续被占序列的长度将不断增长，从而导致搜索的平均时间也不断增长。而且，越长的被占序列越容易变得更长（见图 11-10）。因此，在实际中，线性探寻并不被看好。

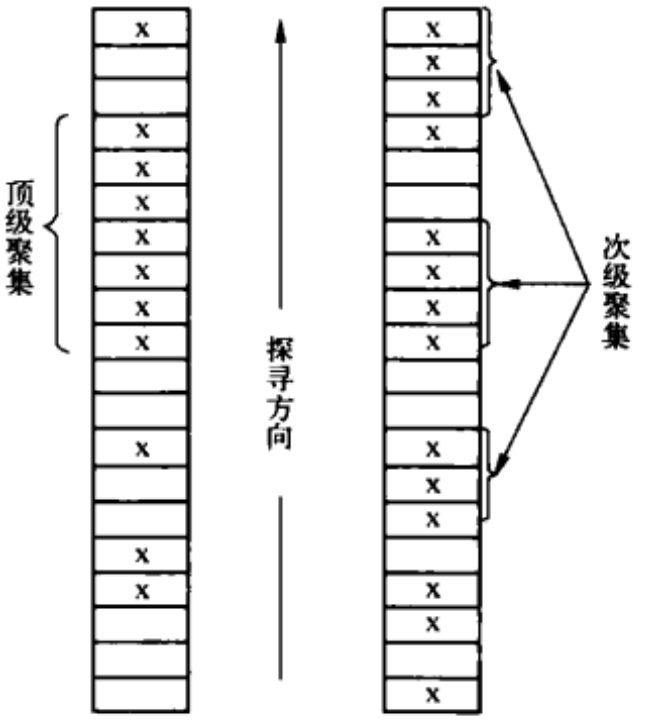


图 11-10 散列表里的顶级聚集和次级聚集

2. 非线性探寻

线性探寻不行，自然用非线性探寻来替换。而非线性探寻中最容易想到的是平方探寻（quadratic probe）。与线性探寻一样，平方探寻也是从一个原始的散列函数开始，但探寻的序列不再是线性，而是在散列表里面按照一个平方级函数跳跃着寻找空位，即

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

这里的 c_1，c_2 为不等于 0 的常数。很显然，必须小心选择 c_1、c_2 和 m，使得随着 i 的递增，$h(k,i)$的值形成一个〈0, 1, …, m−1〉的完整排列。

由于平方探寻是跳跃着在散列表里寻找空间，因此不会出现顶级聚集现象，但却有可能形成次级聚集（secondary clustering）（见图 11-10），即原始散列值相同的不同元素，其探寻序列是一样的！

当然也可以使用更高阶的多项式作为探寻函数，但效果并没有什么根本不同。

3．双重散列探寻

事实上，不管使用何种探寻方式，只要在探寻序列的计算过程只有一个散列函数值，就不可避免地会产生次级聚集。因此，要避免次级聚集，在探寻序列的计算过程中需要使用多个散列函数。当然，为了避免设计散列函数时所费精力过多，通常将散列函数限制为 2 个，这就是**双重散列探寻**。此种散列探寻使用两个散列函数 $h_1(k)$和 $h_2(k)$来构造新散列函数

$$h(k,i) = (h_1(k) + ih_2(k)) \bmod m$$

对于任意两个不同的元素来说，两个不同的散列函数的散列值均相同的概率显然要低于一个散列函数的散列值相同的概率，因此，双重散列探寻的聚集效应将大为减少。实际上，双重散列的效果也确实非常好。不过，这里有一点要求是 $h_2(k)$的取值与 m 必须相对为素。而相对为素也比较容易做到：如果 m 是 2 的幂，则 $h_2(k)$只要产生奇数即可。

4．伪随机探寻

双重散列探寻虽然大大降低了聚集效应，但只要探寻序列是有规律产生的，聚集效应就不能避免。因此，理想的探寻策略是随机选择下一个要探寻的位置，这样出现聚集的可能性将微乎其微（除非天意难违）。因此，使用随机探寻似乎是一个很好的办法。但显然我们不能使用真正的随机探寻（当然也不可能实现真正的随机），因为这样就不能保证对同一个元素的探寻过程总是相同！因此，我们需要的是所谓的伪随机探寻。

伪随机探寻策略就是选择一个 1～M-1 的随机排列 r_1, r_2, …, r_{M-1}作为探寻序列，而插入和搜索都需要按照该序列进行。例如，如果 M = 101，随机探寻序列为 r_1=2、r_2=5、r_3=32，如果 $h(k_1)$=30，$h(k_2)$=28，则 k_1 的探索序列为 30，32，35，62；k_2 的探寻序列则是 28，30，33，60。遗憾的是，伪随机探寻也存在着次级聚集效应。

11.7.6　封闭寻址散列的效率分析

由于解决碰撞的方法不同，封闭寻址散列的搜索成本与开放寻址散列的搜索成本也将不同。但哪一种更好呢？直觉告诉我们，因为有聚集效应，封闭寻址的成本应该高一些。根据前面的分析，开放寻址散列的成功和不成功搜索的时间成本都是 $\Theta(1+\alpha)$。因此，封闭寻址散列的搜索成本应该在 $\Theta(1+\alpha)$之上。事实是否如此呢？分析便知。

假设散列表大小为 m，散列表里的元素个数为 n。散列函数将所有元素均匀散射到散列表里，且探寻策略也很均匀，即{0,1,2,…,m−1}的任意排列成为探寻序列的概率相同。另外，我们暂时假设没有删除操作，且任一元素被访问的概率相同。

11.7.7　搜索不成功的时间成本

对于一次不成功的搜索来说，搜索过程中的每次探寻都碰到一个被占用的位置，只有到最后一次才探寻到空位。设 X 为一次不成功搜索需要探寻的次数，定义 A_i (i=1,2,…)为第 i 次探寻发生且探寻到的是一个被占用的位置的事件，则有：$X \geqslant i$ 当且仅当第 1,2,…,i-1 次探寻都已发生并且探寻到的都是被占用的位置，即

$$\Pr\{X \geqslant i\} = \Pr\{A_1 \cap A_2 \cap A_3 \cap \cdots \cap A_{i-1}\}$$

由概率论里的贝叶斯定理，有：

$$\Pr\{A_1 \cap A_2 \cap A_3 \cap \cdots \cap A_{i-1}\}$$
$$= \Pr\{A_1\}\Pr\{A_2|A_1\}\Pr\{A_3|A_1 \cap A_2\}\cdots\Pr\{A_{i-1}|A_1 \cap A_2 \cap A_3 \cap \cdots \cap A_{i-2}\}$$

现在我们需要做的事情是计算该表达式中的每个概率。那么这些概率是多少呢？

$\Pr\{A_1\}$：第 1 次探寻时，散列表一共有 m 个可能被探寻的位置，散列表里的元素个数为 n，因此第 1 次探寻碰到一个被占用的位置的概率为 n/m，即 $\Pr\{A_1\}=n/m$。

$\Pr\{A_2|A_1\}$：在第 1 次探寻碰到一个被占用位置后，需要进行第 2 次探寻，此时尚有 $m-1$ 个位置可探寻（由于是均匀散列，即探寻序列是〈0,1,…,$m-1$〉的一个排列，因此，下次探索的位置必定是此前尚未探索过的位置），而这 $m-1$ 个可探位置里存放着 $n-1$ 个元素，因此第 2 次探寻碰到被占用位置的概率为$(n-1)/(m-1)$，即 $\Pr\{A_2|A_1\}=(n-1)/(m-1)$。

$\Pr\{A_3|A_1 \cap A_2\}$：在第 1 次、第 2 次探寻均碰到被占用位置后，需要进行第 3 次探寻，此时尚有 $m-2$ 个位置可探寻，而这 $m-2$ 个可探寻位置里存放着 $n-2$ 个元素，因此第 3 次探寻碰到被占用位置的概率为$(n-2)/(m-2)$，即 $\Pr\{A_3|A_1 \cap A_2\}=(n-2)/(m-2)$。

依次类推，可得出：

$$\Pr\{A_{i-1}|A_1 \cap A_2 \cap A_3 \cap \cdots \cap A_{i-2}\} = \frac{n-i+1}{m-i+1}$$

将上述结果汇集起来，则有：

$$\Pr\{X \geqslant i\} = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdot \cdots \cdot \frac{n-i+1}{m-i+1}$$

由于是封闭寻址，散列表里元素的个数不可能超过散列表的大小，即 $n \leqslant m$。

如果 $n=m$，则有：

$$n/m=(n-1)/(m-1)=(n-2)/(m-2)=\cdots=(n-i+1)/(m-i+2)=\alpha$$

因此，

$$\Pr\{X \geqslant i\}=n/m \cdot (n-1)/(m-1) \cdot (n-2)/(m-2) \cdot \cdots \cdot (n-i+1)/(m-i+1)=\alpha^{i-1}$$

如果 $n<m$，则有：$(n-j)/(m-j) \leqslant n/m$，$j \geqslant 0$，

因此，

$$\Pr\{X \geqslant i\} \leqslant (n/m)^{i-1}=\alpha^{i-1}$$

综合上述两种情况，则有：

$$\Pr\{X \geqslant i\} \leqslant \alpha^{i-1}$$

根据概率论，则有：

$$E[X] = \sum_{i=1}^{\infty} \Pr\{X \geqslant i\} \leqslant \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^{i}$$

如果 $\alpha<1$，则上述求和的结果为：$1/(1-\alpha)$；如果 $\alpha=1$，则上述求和不收敛，$E[X]$没有定义。

定理 封闭寻址下不成功搜索的探寻次数期望值最多为 $1/(1-\alpha)$，这里 $\alpha=n/m<1$ 为加载因子。

该定理说明，如果 α 是一个常量，则封闭寻址下的不成功搜索时间为常数时间。如果散列表只有一半满，则不成功的搜索所需探索次数的期望值为 1/(1–0.5) = 2。如果表 90%满，则探索次数的期望值为 1/(1–0.9) = 10。这符合我们的常识：表越满，探寻到一个空位的时间就越长。而且这个时间成本期望值确实比开放寻址下的搜索成本高！

另外，从该定理可以推出，插入所需要的探寻次数的期望值也最多是 $1/(1-\alpha)$，因为在没有删除的情况下，插入的探寻序列与不成功搜索的探寻序列完全一样！

11.7.8　成功搜索的效率分析

首先，总体上来说，由于不成功搜索需要一直探寻到空位才停下来，而成功搜索则在未到达空位时即停止，因此成功搜索的成本期望值似乎应该比不成功搜索的成本期望值低。而要获得成功搜索的时间成本，就要仔细分析一个元素在散列表里的位置与探寻序列的关系。

显然，对元素 k 的成功搜索所需要的探寻序列与最初插入键值 k 时的探寻序列一样。根据本章前面对插入的分析可知，如果元素 k 是第$(i+1)$个被插入的键值，则当时的加载因子 $\alpha=i/m$。因此，对元素 k 的搜索所需要的探寻次数的期望值为 $1/(1-i/m)=m/(m-i)$。

对 n 个键值求平均就获得成功搜索的探寻次数的期望值最多为：

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i}=\frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i}=\frac{1}{\alpha}(H_m-H_{m-n})$$

如果 $n<m$，我们可以使用积分对上述调和级数进行界定：

$$\frac{1}{\alpha}(H_m-H_{m-n})=\frac{1}{\alpha}\sum_{k=m-n+1}^{m}\frac{1}{k}\leqslant\frac{1}{\alpha}\int_{m-n}^{m}\frac{1}{x}\mathrm{d}x=\frac{1}{\alpha}\ln\frac{m}{m-n}=\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$$

因此，有下面的定理：

定理　封闭寻址下成功搜索的探寻次数最多为$(1/\alpha)\ln(1/(1-\alpha))$，这里 $\alpha=n/m<1$ 为加载因子。

11.8　散列表元素删除

到此为止，我们对封闭寻址散列算法的所有分析都假定没有删除操作。由于没有删除，搜索的探寻序列与插入时用的探寻序列一样。但如果有删除的话，情况会如何呢？

显然，我们不能直接将元素删除，因为这样将在散列表里留下空位，导致散列表里的一些元素无法被搜索。具体来说，凡是探寻序列包括此空位的元素都不能被搜索到。例如，假定我们欲删除的元素 k 占用位置为 j，但在插入元素 k 后，我们又曾经插入过元素 k'，而在插入元素 k' 的过程中探寻过位置 j。此时，如果删除元素 k 并在位置 j 上留下一个空位，则此后将无法搜索到元素 k'，因为在搜索 k' 时必然探寻到位置 j，如果位置 j 为空，那么探寻将在此停止，并报告搜索失败。

因此，对于封闭寻址散列表来说，删除记录时不能留下空位，解决方法就是在删除

位置上留下一个特殊标志，这个特殊标志就被称为**墓碑**（tombstone）。墓碑告诉探寻算法这里原来是有元素的，而不是真正的空位，因此应该继续往前探寻，从而避免前面所说的问题。

当然，有了墓碑之后插入算法就得进行修改：在碰到墓碑的时候即可进行插入，而无需等到真正的空位才进行插入操作。

虽然墓碑解决了删除空位问题，但墓碑的使用将增加搜索时探寻的次数，同时降低插入时探寻的次数。这样，搜索和插入的时间成本将不再依赖于加载因子α。不过，如果插入和搜索的频率相当或相差不太远，则此种情况将不是什么大问题。但如果插入操作非常稀少，则需要对墓碑进行清理，即通过对散列表进行重新散列以消除墓碑，从而降低搜索的时间成本。

需要注意的是，元素删除所导致的问题在开放寻址散列算法中不会出现，因为每个元素只能映射到自己散列值所对应的位置，不会占用别的元素所处的位置，而散列值相同的元素以链表形式连接，而对链表删除一个元素不会造成后续的搜索问题。

11.9 随机化散列

到此，对于散列算法来说，一切问题似乎都已解决：散列函数设计问题、元素碰撞问题、散列表删除空位问题、开放寻址散列和封闭寻址散列的时间复杂性分析等全部有了答案。

用一句电影里的台词就是：西线无战事！

但真是这样吗？

根据本章的分析可知，在开放寻址散列算法下，搜索的时间复杂性为 $\Theta(1+\alpha)$；在封闭寻址散列算法下，成功搜索的时间复杂性为$(1/\alpha)\ln(1/(1-\alpha))$，而不成功搜索的时间复杂性为 $1/(1-\alpha)$。如果读者够仔细的话，就会发现所有这些分析结果均有赖于一个假设：均匀散列！

因为均匀，所以每个位置接受的元素数是相同或相似的，并且在成功搜索下，每个元素被搜索的概率也是相同的。因此，对任意一个元素来说，对其进行搜索的成本都是在可控范围内。但问题是，实际情况是这样的吗？或者实际生活有这样美好吗？

也许生活是美好的，因为我们确实可以针对不同的输入分布设计出均匀的散列函数！但是我们怎么能够保证事先知道输入元素的取值分布呢？就像本章前面说过的：人算不如天算。如果运气不佳，所有的元素被散列到同一个地址怎么办呢？这个时候散列就退化成顺序搜索！甚至还不如顺序搜索，因为顺序搜索起码不会这么折腾一番，浪费这么多脑细胞！

也许很多人认为所有元素均散列到同一个位置的可能性不大。但对于设计算法的人来说，可能性不大并不令人满意。实际上，对于任意的散列函数来说，都存在某种元素的集合，使得所有元素在经过散列函数后映射到同一个地址上。

除了运气不佳而导致散列效率的急速下降外，对手的蓄意破坏也可能彻底击溃我们的散

列算法。例如，编译器通常使用散列算法来存储与查找程序中出现的各种符号。如果该散列函数被对手知道，对手就可以通过精心选择一个程序的变量名，使得该程序在编译时，所有变量均映射到编译器符号表的同一个地址里，从而造成编译器效率的急剧下降，进而达到诋毁该编译器的目的（或者别的目的）。

因此，恶劣的运气与对手的挑衅就是设计散列算法时不得不面对的一个问题。那么怎样才能应付此种挑战呢？答案似乎是不让对手知道我们所使用的散列函数，即恪守秘密。由于对手不知道我们的散列算法，自然无法有针对性地设计出恶劣的数据元素集合。看上去这个办法不错，但实际上并不能从根本上解决问题。因为只要我们使用的散列函数是固定的，对手只要有耐心并且执著，他总可以通过试验慢慢找出我们所使用的散列函数的特征，从而设计出摧毁整个散列算法的程序。由此可见，光靠保密并不解决问题。而且，保守秘密也无法应对厄运的袭击。

那出路是什么呢？答案是随机化，即程序的每次运转都使用一个不同的散列函数！或者说，每次程序运行时使用哪个散列函数完全是随机的。例如，对于编译器来说，可以在每次运转编译器时选择一个不同的散列函数来构建符号表。这样，由于散列函数的选择是随机的，因此对手自然无法通过试验来找出散列函数的特征（因为每次试验时散列函数都发生了变化）。

当厄运来袭时，如果数据元素集合对于某个散列函数来说呈现聚集效应，则对于另一个完全不同的散列函数来说也呈现聚集效应的概率就很小了。由于每次运行使用的散列函数不同，前一次遭受厄运就意味着下一次将与厄运分手（至少我们希望如此）。而这就是随机化散列。

11.10　全域散列

由上述分析可知，随机化散列就是在程序每次运行的时候随机选择一个不同的散列函数。而只使用一个散列函数无法达到此目的。我们需要做的事情是找到一组散列函数，每次随机使用其中一个不同的散列函数，这一组散列函数共同工作而达到我们的目的。

显然，每次使用的不同散列函数仍然需要满足我们前面说过的“好的散列函数”要求。不然的话，不用对手和厄运动手，我们自己就将自己的散列算法打倒在地。

剩下的问题是，怎么确保随机选择的散列函数是一个好的散列函数呢？要保证该目标的实现，唯一的办法是确保我们找到的这组散列函数的每一个都是好的。这种一组散列函数全部都是好的散列函数就称为**全域散列**，因为从整个散列函数组的域来看，全都是好的！

从哪里找来这么多好的散列函数呢？或者说如何选择或构建这样一个散列函数组呢？这就是全域散列要解决的问题。

定义　设 U 是所有元素的集合，H 为一个有限的散列函数集合，H 里面的每个散列函数 h 将集合 U 映射到散列表的 m 个位置上 $\{0, 1, \cdots, m-1\}$，如果集合 H 满足条件：

$$\forall x, y \in U, x \neq y, |\{h \in H: h(x)=h(y)\}|=|H|/m$$

则 H 称为**全域散列**。

该定义说的是，全域散列函数组里，将任意一对元素 x 和 y 映射到同一位置的散列函数个数为$|H|/m$。换句话说，如果我们从 H 里面随机选择一个散列函数 h，该散列函数将任意元素对 x 和 y 散列到同一个位置的概率是 $1/m$，如图 11-11 所示。

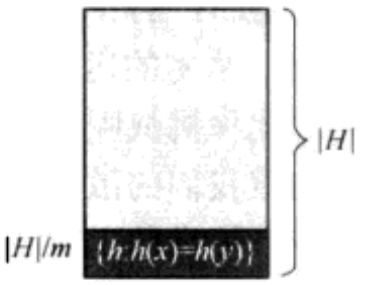


图 11-11 全域散列函数集合 H

全域散列函数有什么优势吗？有！因为它里面的每一个散列函数都是均匀散列函数！即对于每个元素 x 来说，散列表里与其碰撞的元素个数小于散列表的加载因子。

下面的定理说明了全域散列的均匀性。

定理 设 h 是从全域散列函数组 H 里随机选取的一个散列函数，如果用 h 将任意 n 个元素散列到大小为 m 的散列表 T 里，则对于任意元素 x，与 x 发生碰撞的元素个数小于散列表的加载因子。如果 C_x 为在散列表 T 里与元素 x 相碰撞的元素个数，则 $E[C_x]<n/m$。

证明 设 C_{xy} 为在散列表 T 里元素 x 与元素 y 发生碰撞的事件，且令

$$C_{xy}=1 \quad h(x)=h(y)$$
$$C_{xy}=0 \quad h(x)\neq h(y)$$

则有：

$$C_x=\sum_{y\in T-\{x\}} C_{xy}$$

两边同时取期望值有：

$$E[C_x]=E[\sum_{y\in T-\{x\}} C_{xy}]=\sum_{y\in T-\{x\}} E[C_{xy}] \qquad (11\text{-}2)$$

根据全域散列的定义，有：$E[C_{xy}]=1/m$。将此代入式（11-2）有：

$$E[C_x]=\sum_{y\in T-\{x\}} E[C_{xy}]=\sum_{y\in T-\{x\}} \frac{1}{m}=\frac{n-1}{m}\leqslant\frac{n}{m}$$

因此，全域散列函数确实都是均匀散列函数。 □

推论 使用链接（开放寻址）和全域散列，每次搜索操作的期望时间为 $O(1)$。

全域散列构造

前面的定理告诉我们，全域散列确实是均匀的，它的散列行为是我们所期望的。问题是，从哪里去找这么一组均匀散列函数呢？

从全域散列定义可知，这组函数的个数至少应该是 m，否则$|H|/m$ 将小于 1。而且，为了增加对手破解的难度，这组散列函数的个数是越多越好。最好是 m 的倍数，这样，$|H|/m$ 将能够除尽而获得一个整数值（当然这个不是必需的）。

找出一个好的散列函数不难，但要找出这么多的好散列函数可能并不容易（我们可是要求该全域散列函数组里面的每一个散列函数都是好的），就像我们要求一个团队或班级里面每个成员都是优秀一样，似乎非常不现实。

真的不现实？在生活里也许是这样。幸运的是，在算法中构造一组优秀的散列函数比在生活中寻找一个成员全都优秀的团队要容易得多！下面就来构造这种函数：

1）设 m 为素数，将每个数据元素 k 分解为进制为 m 的数位，即每个数位的取值范围为 $\{0, 1, \cdots, m-1\}$。也就说分解 $k=\langle k_0, k_1, \cdots, k_r\rangle$（$0 \leqslant k_i < m$）。假定分解后有 $r+1$ 个数位。

2）随机选择一个进制为 m，长度是 $r+1$ 个数位的数 $a=\langle a_0, a_1, \cdots, a_r\rangle$，这里每个 a_i 的值都是从集合$\{0, 1, \cdots, m-1\}$里随机选取的。

3）定义散列函数

$$h_a(k)=\sum k_i a_i \bmod m \text{（模 } m \text{ 的点积）}$$

则所有这样构造的散列函数 h_a 组成的集合 H 就是一个全域散列组。

证明　我们先来看 $H=\{h_a\}$的大小。由于散列函数 h_a 的个数完全取决于不同 a 的个数，而 a 有 $r+1$ 个数位，每个数位有 m 个取值可以选择，因此，不同 a 的个数为 m^{r+1}，即 $|H|=m^{r+1}$。

下面我们来看该组散列函数是否均匀。这点只要考察对于两个不同的元素，有多少散列函数将它们映射到同一个位置即可。我们希望这个数值是：

$$|H|/m=m^{r+1}/m=m^r$$

设 $x=\langle x_0, x_1, \cdots, x_r\rangle$ 和 $y=\langle y_0, y_1, \cdots, y_r\rangle$ 是两个不同的元素，则它们至少有一个数位不同。不失一般性，假定它们的第 0 个数位不同，即 $x_0 \neq y_0$。

那么有多少个散列函数 $h_a \in H$ 会将 x 和 y 散列到同一个位置上呢？

根据全域散列函数的构造，如果要使 $h_a(x)=h_a(y)$，则必有：

$$\sum_{i=0}^{r} x_i a_i \equiv \sum_{i=0}^{r} y_i a_i (\bmod m)$$

即

$$\sum_{i=0}^{r} a_i (x_i - y_i) \equiv 0 \ (\bmod m)$$

也就是：

$$a_0(x_0 - y_0) + \sum_{i=1}^{r} a_i (x_i - y_i) \equiv 0 (\bmod m)$$

而上述等式可以转换为：

$$a_0(x_0 - y_0) \equiv - \sum_{i=1}^{r} a_i (x_i - y_i) (\bmod m) \qquad (11\text{-}3)$$

根据数论理论，如果 m 为素数，则任何小于 m 且不等于 0 的数都存在唯一模 m 的倒数，即$\forall z \in Z_m$，$z \neq 0$，$\exists z^{-1} \in Z_m$ 使得 $zz^{-1} \equiv 1 \ (\bmod m)$。例如，如果 $m=7$，则数 1、2、3、4、5、6 的模 7 的倒数分别是（该定理证明请参阅数论方面的资料，本书不予以证明）：

z：	1	2	3	4	5	6
z^{-1}：	1	4	5	2	3	6

由于 $x_0 \neq y_0$，且 $x_0 - y_0 < m$，所以 $x_0 - y_0$ 相对于模 m 的倒数 $(x_0 - y_0)^{-1}$ 必然存在。在式（11-3）两边同时乘以 $(x_0 - y_0)^{-1}$，则有：

$$a_0 \equiv \left(-\sum_{i=1}^{r} a_i(x_i - y_i)\right)(x_0 - y_0)^{-1} \pmod{m}$$

由此可见，对于任意 $a_1, a_2, \cdots, a_r$，只有一种选取 a_0 的方式使得元素 x 和 y 碰撞。那么 $a_1, a_2, \cdots, a_r$ 的选择有多少可能呢？

按照 a 的定义，每个数位 a_i 有 m 种取值 $\{0,1,2,\cdots,m-1\}$。因此，在 r 个数位 $a_1, a_2, \cdots, a_r$ 上有不同的数 a 的个数是 m^r。由于在这 r 个数位取值完成后，a_0 的取值只有一个，即

$$a_0 = \left(-\sum_{i=1}^{r} a_i(x_i - y_i)(x_0 - y_0)^{-1} \pmod{m}\right)$$

因此，使得 x 和 y 碰撞的散列函数个数为：

$$m^r 1 = m^r = |H|/m$$

因此，我们如此构造的散列函数组确实是全域散列函数组。 □

11.11 完美散列

有了全域散列后，应该说问题都已经解决了，我们的对手似乎也无计可施了。

遗憾的是，击败所有对手并不是满足的理由。事实上，我们根本就不能满足！因为全域散列的成功（即不发生聚集效应）有赖于运气的帮忙。虽然全域散列函数都是均匀的，但毕竟存在一个这些函数同时都很糟糕的概率，至少对于某些特定的数据元素分布来说存在这样的恶劣情况：虽说对于某个散列函数来说呈现聚集效应的数据元素集合在另一个完全不同的散列函数上呈现聚集效应的概率很小，但这个概率并不是没有。而这正是我们的担心所在。

显然，依赖于运气总是让人有点心神不定。因为运气是琢磨不定的，也许在我们踌躇满志的时候，厄运突袭，一切就玩完了。

因此，我们并不能在得到全域散列后休养生息，而是继续奋斗。所谓的生命不息，奋斗不止嘛。而奋斗的目标就是完美散列：保证在**任何情况下**散列的搜索时间成本都不超过 $\Theta(1)$！而且散列表的大小还不能太浪费，只能与元素个数成线性比例，即 $m=O(n)$。

具体来说，完美散列的问题定义如下：给定 n 个元素，构建一个 $m=O(n)$ 大小的静态散列表，使得搜索的最坏时间复杂性为 $\Theta(1)$。

那么怎么构造出完美散列呢？很显然，完美散列要求在整个散列过程中不发生碰撞，即使发生，也必须是常数次。那么怎么才能不发生碰撞呢？或者发生碰撞的次数是常数呢？

本章前面多次说过，好的散列函数能够将元素均匀映射到散列表。而全域散列帮我们解决了如何构造最好的散列函数问题。因此，在散列函数上面已经没有改进余地，剩

下能够动心思的地方只能是散列表的构造本身，即为了不发生碰撞，必须构思一个精巧的散列表。

怎样构造散列表才能避免碰撞呢？我们只需看一下全域散列情况下，n 个元素散列到一个大小为 m 的散列表时，发生碰撞的次数即可。

由 11.10 节的讨论可知，对于一个全域散列函数来说，任意一对元素 x 和 y 被散列到同一个位置的概率是 $1/m$。而对于 n 个元素来说，一共有 C_n^2。因此，对不同的元素对，使用该全域散列函数发生的总碰撞次数的期望值为：

$$C_n^2 \cdot \frac{1}{m} = \frac{n(n-1)}{2} \cdot \frac{1}{m} = \frac{n(n-1)}{2m}$$

而要使得整个散列过程不发生碰撞，则需要 $n(n-1)/2m<1$。求解这个不等式可得：$m> n(n-1)/2$。因此，散列表大小 $m=\Theta(n^2)$时，发生碰撞的次数可以期望少于 1，也就是不发生碰撞。

完美散列表的构造似乎清楚了，但还存在两个问题：

1）如此构造的散列表只是碰撞的期望次数小于 1，并不是说碰撞次数小于 1！而这并不符合构造完美散列的要求。

2）更为重要的一点，$\Theta(n^2)$的空间要求实在是太大了，既非常浪费，恐怕也是我们在实际中承受不了的。例如，为了在散列 1 000 个元素时不发生碰撞，我们需要的散列表大小为 100 万个元素空间！浪费巨大！

但这条道路似乎是不发生碰撞的唯一途径。有什么解决办法吗？

解决办法是双层散列！而每一层都使用全域散列，第 1 层散列表大小与元素个数相同，为 n 个存储单元，而第 2 层散列表大小设为第 1 层散列时发生碰撞的元素个数的平方！由于第 1 层使的是全域散列，因此在第 1 层发生碰撞的元素个数不会太多，这样第 2 层的空间使用将被减少到我们能够承受的范围（我们的希望）。这样，按照上述分析，我们可以期望在第 2 层散列表上不发生任何碰撞！

例如，图 11-12 给出的是一个双层结构的完美散列。

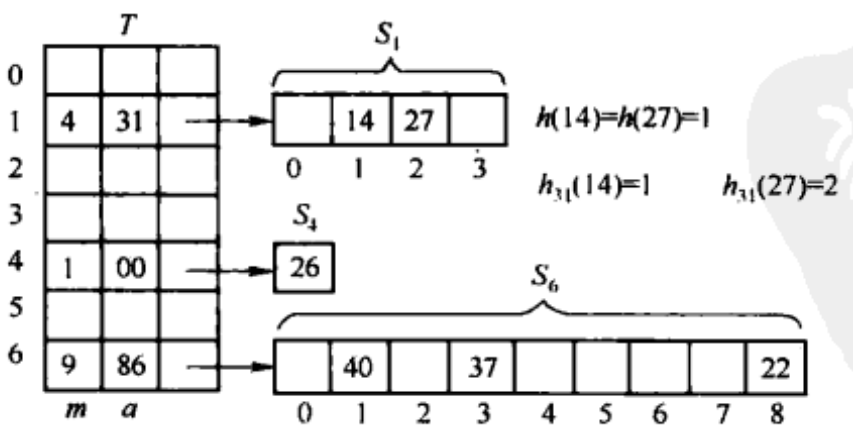


图 11-12　完美散列：第 2 层散列表上没有碰撞

T 为第 1 层散列表，大小与元素个数 n 一样。S 为第 2 层散列表。第 1 层散列表里不存放数据元素，而是第 2 层散列表的大小及其使用的散列函数。第 2 层散列表用来存放元素。凡是在第 1 层发生碰撞的元素将共享同一个第 2 层散列表。例如，元素 14 和 27 在第

1 层散列都映射到第 1 个位置，因此它们共享同一个第 2 层散列表 S_1。而第 2 层散列表的大小则为落于第 2 层散列表里元素个数的平方。由于落入到 S_1 的元素个数为 2，因此 S_1 的大小为 2^2=4。这个大小记录在表 T 里的第 1 个域里。T 里的第 2 个域存放的是相应第 2 层散列表使用的全域散列函数的构造参数 a（请参阅全域散列）。例如，S_1 的全域散列构造参数 a=31。

定理 设 H 是一组全域散列函数，散列表大小 $m = n^2$，如果从 H 里面随机选取一个散列函数 h 用来将 n 个元素散列到表 M 里，则发生碰撞的次数期望值最多为 1/2。

上述定理说的是，在定理给定的条件下，我们甚至不期望发生 1 次碰撞！或者说，我们期望没有碰撞！这个定理的证明非常简单。

证明 根据全域散列的定义，任意两个元素在散列表 m 里发生碰撞的概率为 $1/m = 1/n^2$。由于可能发生碰撞的键值对为 C_n^2，因此，碰撞次数的期望值为：

$$C_n^2 \times 1/n^2 = n(n-1)/2 \times 1/n^2 < 1/2$$ □

有了上述定理，我们就可以得出如下推论：

推论 在第 2 层散列表不发生碰撞的概率至少为 1/2。

证明 要证明此推论，我们需要将期望值与概率建立关系。根据马尔科夫不等式（Markov inequality），对于任意非负随机变量 X，有：

$$\Pr\{X \geqslant t\} \leqslant E[X]/t$$

如果 X 代表在全域散列函数下第 2 层散列表发生碰撞的次数的随机变量。则马尔科夫不等式告诉我们，发生 t 次或更多次碰撞的概率小于 X 的期望值除以 t。

令 t=1，则有：

$$\Pr\{X \geqslant 1\} \leqslant E[X]/1 = E[X] = 1/2$$

因此，发生一次或更多次碰撞的概率最多是 1/2，即不发生碰撞的概率至少为 1/2。 □

到目前为止，我们只证明了第 2 层散列表不发生碰撞的概率至少为 1/2，或者说，第 2 层散列发生碰撞的次数的期望值小于 1/2 次。但并没有证明第 2 层散列表不会发生碰撞，而完美散列需要的是 100%不发生碰撞。我们能做到吗？

答案当然是能。方法很简单：随机从 H 里面选取散列函数 h 进行测试，如果 h 导致碰撞，则另外再选择一个，直到不发生碰撞为止。此时我们将保证第 2 层散列表没有碰撞！

那么需要检查多少个散列函数才会找到一个不发生碰撞的函数呢？由于任意一个散列函数不发生碰撞的概率至少为 1/2，因此，平均检测两个散列函数就应该得到一个不发生碰撞的散列函数！

完美散列的存储空间分析

双层全域散列表已经达到了保证第 2 层散列不会发生碰撞。但它在达到这个能力时使用的存储空间似乎太大——n^2 呢！幸运的是（也许必然的是），这个平方的自变量 n 不是所有的元素个数，而是落到第 2 层同一个散列表的元素个数！而根据全域散列的定义，落到同一

个位置的元素个数不会太多！因此，这里的 n^2 也许并不大。

下面就来仔细分析一下存储空间的使用：第 1 层散列表 T 的大小为 n，以保证在运气好的情况下所有元素在第 1 层就不发生碰撞；而第 2 层散列表的大小各不相同，设 n_i 为落到表 T 位置 i 上的元素个数，则相应的第 2 层散列表 S_i 大小为 n_i^2。

这样，第 2 层散列表使用的总空间容量的期望值为：$E\left[\sum_{i=0}^{m-1}\Theta(n_i^2)\right]$。

这个数值是多少呢？$\Theta(n)$。还记得桶排序吗？桶排序的时候，我们分析过 $E[\Sigma\Theta(n_i^2)]$，结论是 $E\left[\sum_{i=1}^{n}\Theta(n_i^2)\right]=\Theta(n)$ (务必注意 $m=n$)。因此，利用桶排序的分析成果，我们得知完美散列的空间效率为线性，而这正是我们所希望的（注意，虽然桶排序里分析的是时间复杂性，而这里是空间复杂性，但表达式都是一样的）。

因此，使用 $\Theta(n)$存储空间可保证在第 2 层散列表上不发生碰撞！而这就叫做**完美**！

思考题

1. 对于有序表查找，有人提出了一种基于斐波那契序列的搜索算法。设有序表 T 的元素个数为 n，F_k 为大于等于 n 的最小斐波那契数，则所谓的斐波那契搜索算法如下：

```
FINBONACCI-SEARCH (T, x)
i=k;
while (i≠0) {
     if(T[F_{i-1}]==x) {
          break;
     }
     else if (T[F_{i-1}]<x) {
          清除元素 T[1..F_{i-1}];
          将元素序列 T[F_{i-1}+1..n]移动到 T[1]开始的位置上;
          n=n-F_{i-1};
          i=i-2;
    }
    else if (T[F_{i-1}]>x) {
          清除元素 T[F_{i-1}+1..n];
          n=F_{i-1};
          i=i-1;
   }
}
if (i≠0)
     return i;
else
```

```
        return FAILURE;
```

这个算法能够正确搜索到元素 x 吗？如果能，请证明，并分析该算法的时间复杂性；如果不能，请给出反例。

2. 本章列出了 UNIX System V Release 4 里面 ELF 散列的 C 程序实现：

```
int ELFhash(char* key) {
    unsigned long h=0;
    while(*key) {
        h=(h<<4)+*key++;
        unsigned long g=h & 0xF0000000L;
        if(g)h^=g >> 24;
        h&=~g;
    }
    return h%M;
}
```

请还原该程序片断使用的完整散列函数。

3. 在本章讨论的乘法散列函数 $h(k) = (Ak \bmod 2^w)$ rsh $(w-r)$里，A 为一个奇数，取值范围为 $2^{w-1}<A<2^w$，并且不能很靠近 2^{w-1} 或 2^w，请说明这样做的理由。
4. 请分析开放寻址散列算法的插入和删除时间。
5. 证明：在封闭寻址散列算法下，聚集效应不可避免。
6. 我们在讨论散列函数的时候经常假定所有的键值都是整数。这个假设是否影响我们讨论的有效性？为什么？
7. 在讨论乘法散列函数的时候谈到了轮盘赌。我们说每次推动都使轮盘停在一个不同的地方，是因为赌徒推动的旋转距离 A A 与轮盘的周长相对为素，且又不靠近轮盘格度大小的倍数。但是每个赌徒的推力显然并不一样，并且赌徒每次推动的时候也不一定使出同样的力气，但为什么轮盘转动还是呈现出我们所看到的均匀结果呢？
8. 有同学认为本章的封闭寻址下不成功搜索的探寻次数$(1/(1-\alpha))$的证明比较复杂，你能否找到一个更加简单的证明方法？
9. 对于封闭寻址散列表来说，如果加载因子$\alpha=1$，请分析成功和不成功搜索的时间成本。
10. 对于封闭寻址散列表来说，在有删除操作并使用墓碑的情况下，我们前面的搜索时间的期望值分析还成立吗？为什么？
11. 在完美散列算法里，第 2 层散列表的大小被设计为落到这个散列表内元素个数的平方，从而保证了发生碰撞的概率很低。而这个定理与我们熟知的生日悖论有着不易觉察的联系：当一个房间的人数比一年里的日子数少很多时（如小于 365 的平方根），则两个人生日为同一天的概率非常小。但一旦超出这个限制，如大于 365 的平方根的时候，两个人为同一生日的概率迅速增加！请计算出在房间里人数达到多少时，存在两个人生日相同的概率就超过 75%。

12. 完美散列真的完美吗？你能找出一些完美散列的缺陷吗？
13. 对于数据元素可以动态变化的情况，能否构造完美散列？为什么？我们能保证什么？
14. 假如有一个 $3n$ 个容量的散列表，该散列表的冲突解决策略是封闭（链接）的。假如果 $n/4$ 元素被插入到散列表，则对于 $i = 1, 2, \cdots, n/4$ 和 $j = 1, 2, \cdots, 3n$，我们定义如下标示随机变量：

$$X_{ij} = \begin{cases} 1 & \text{元素 } i \text{ 落到位置 } j \\ 0 & \end{cases}$$

假设简单均匀散列(每个元素散列到任意一个位置的概率相等)，求落入到位置 $3n$ 的元素个数的期望值。

PDG

CHAPTER 12

第12章 最短路径

条条大路通罗马。

这不是对罗马所处位置的抽象概括，也不是对罗马帝国的厚颜吹捧。这是真实的历史。

如果漫步西班牙、法国、意大利、希腊、马其顿、小亚细亚、巴勒斯坦、埃及、摩洛哥或其他北非国家和城市，你就会见到罗马人在 2 000 多年前修建的碎石马路。这些罗马大道厚实、平整，能够承载大规模军队人员和辎重。事实上，这些地区的很多现代公路就是修建在罗马古道（见图 12-1）的基础之上。罗马人建造了发达的公路网，把首都罗马与帝国的各个角落都连接在一起，因此，在远古世界说“条条大路通罗马”是毫不夸张的。

图 12-1 罗马古道

不过这些大道并不只是给罗马带来财富与荣耀，它也一度带来了罗马的生死对手：迦太基军。

12.1 剑指罗马

公元前 218 年，迦太基卓越的军队统帅汉尼拔（见图 12-2）决定进军罗马，准备最后解决罗马问题。而进军罗马当然需要仔细选择行军路线。因为条条大路通罗马，选择的范围自然很大。如何选择最佳的进军路线就是摆在汉尼拔面前的首要任务。

图 12-2 人类历史上最伟大的军事统帅之一——汉尼拔（Hannibal）

如果绘制一张古罗马时代的陆海交通图，它大概与图 12-3 差不多。图中黑色圆点代表由迦太基及其同盟控制的地盘，灰色圆点代表罗马及其同盟控制的地盘。实线代表陆上线路，虚线代表水上线路。每对结点间如果有线条，就表示有路可走，而且每条路均有长度。途经己方地盘显然要安全得多，而途经对方城市则意味着战争。

图 12-3 罗马帝国的陆海交通图

汉尼拔需要判断的是选择哪一条线路进军。应该如何选择线路呢？

显然，选择线路的考虑条件并不是线路的长度本身，而是行走此线路需要的时间。虽然水路可能比陆路的公里数要短，但水路行走速度慢，因此所花时间不一定短。另外，走己方及盟国控制的地盘虽然安全，但可能线路长。如果经过的城市可以很快被攻克，走对方及其同盟控制的地盘也不一定是件坏事。

到底应该走哪条线路呢？如果你是汉尼拔的参谋，你会给出什么建议呢？

初看起来，似乎考虑的因素颇多：是否是己方地盘，是海路还是陆路，每条线路的长度。似乎很难想出个所以然来。但如果我们将所有因素都转换为时间，则问题就会简单很多。例如，长度长的路花费的时间多，行走己方地盘比行走对方地盘时间短（在路径长度相同的基础上），走海路比走陆路慢（在公里数相同的基础上），这样我们可以画出图 12-4 所示的这张图。

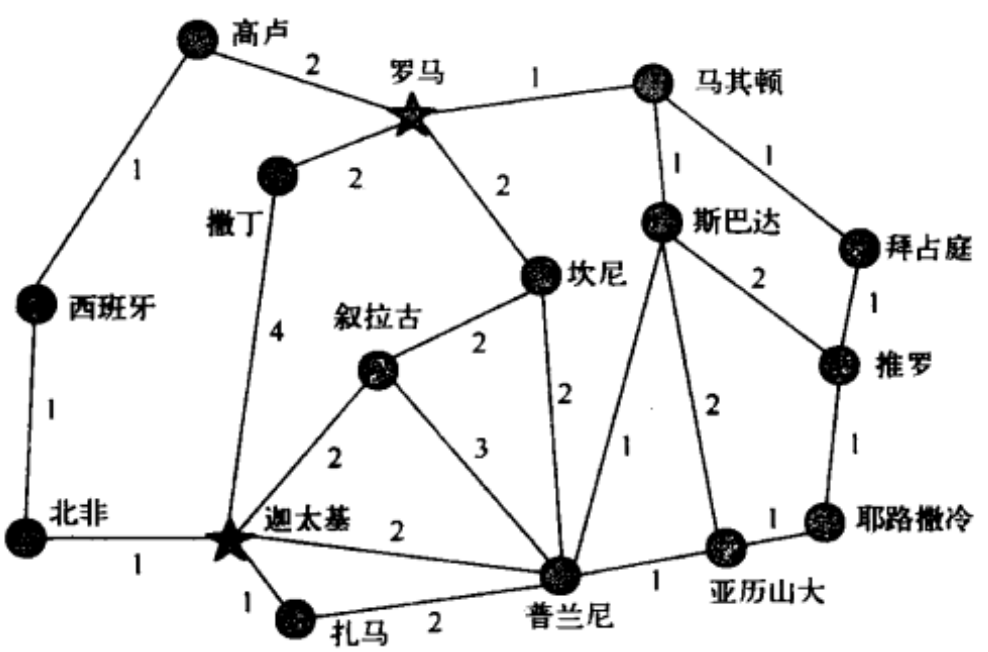


图 12-4 罗马帝国交通图——时间抽象版

这样，从迦太基发兵到罗马选择线路的问题就变成在图 12-4 里寻找一条从迦太基到罗马的最短路径。虽然从迦太基到罗马的路有很多条，而最短的路或许只有一条。

那么哪一条路才是最短路径呢？这就是本章要讨论的最短路径问题。

12.2 最短路径问题

除了汉尼拔有寻找最短路径的需求外，我们日常生活中对最短路径的需求也非常普遍：信差总是寻找最短的投递路径，计算机网络路由也尽量走最短（或成本最低的）路径，我们开车从一个城市到另一个城市去的时候也总是希望走最短的路径。

要定义最短路径，先定义路径权重的概念。给定一个带权重的图 $G=(V, E)$，边的权重函数为 $w: E \rightarrow R$，则一条路径 $p=v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k$ 的权重为该路径上每条边的权重之和：

$$w(p)=\sum_{i=1}^{k-1} w(v_i, v_{i+1})$$

例如，图 12-5 的路径权重为 1。

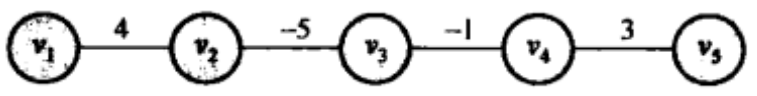


图 12-5 从 v_1 到 v_5 的路径权重为 $w(p)=4-5-1+3=1$

这里每条边的权重可以是这条边的长度、通过成本或其他我们感兴趣的度量。另外，图中的边可以有方向，也可以没有方向。如果有方向，从结点 u 到结点 v 有边并不等于从结点 v 到结点 u 有边；而如果没有方向，则从结点 u 到结点 v 的边与从结点 v 到结点 u 的边是一回事情。一个图的边有方向，则称该图为有向图；如果图的边没有方向，则称该图为无向图。例如，图 12-15 给出的是一个无向图，而图 12-6 给出的是一个有向图。

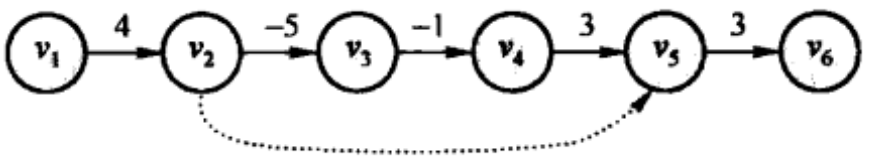


图 12-6 有向图

本章讨论的多数算法都同时适用于有向图和无向图，如有例外，我们将在发生例外的地方指出来。在没有例外的时候，我们的例子将随机使用有向图或无向图。

不管是有向图还是无向图，从任意结点 u 到另一结点 v 的最短路径则是所有从 u 到 v 路径中权重最小的路径。最短路径权重的定义为：

$$\delta(u, v) = \min\{w(p): p \text{ 是从 } u \text{ 到 } v \text{ 的路径}\}$$

显然，并不是所有结点对之间都存在最短路径。如果两个结点之间没有通道（即没有连通），则它们之间不存在任何路径。不过从抽象上看还是可以认为它们之间存在一条路径，只不过该路径的权重为无穷大，即如果从 u 到 v 不存在路径，则 $\delta(u, v) = \infty$。如果一个图包含一个权重为负的循环，则有些结点之间将不存在最短路径。例如，图 12-7 中的结点 u 到结点 v 之间就不存在最短路径，因为不管你找出的路径有多短，我们总是可以沿着负环路多绕行一次而获得一条更短的路径！

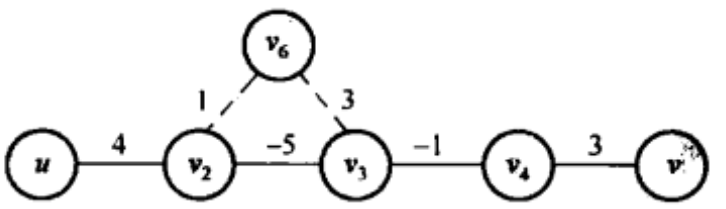


图 12-7 含有负循环的图

对于没有负循环的连通图来说，任意两个结点之间都存在最短路径，而且该最短路径必不会包括任何环路。因为如果包含环路，我们可以将环路拿掉而获得一条更短的路径！

如何高效地计算出任意一对结点之间的最短路径及其路径权重就是所谓的**最短路径问题**。不，严格地讲，这是所谓的**单源单点最短路径问题**。这里，单源指的是只有一个始点，单点指的是只有一个终点。单源单点即指从一个始点到一个终点。也许有的读者会觉得奇怪，最短路径难道不就是从一个始点到一个终点的最短路径吗？难道还有别的最短路径问题？

没错，思维正常的人都会这么认为。但是谁说过研究算法的人都思维正常了呢？事实上，正是研究算法的人抛出了常人看来非常奇怪的“单源多点最短路径”和“多源多点最短路径”两种版本的最短路径问题。而抛出这些版本的最短路径问题的原因本章后面将会解释。

12.3 单源单点最短路径问题

按照定义，要想找出一对结点之间的最短路径，就需要对它们之间存在的所有路径进行比较。而要找出一对结点之间的所有路径则需要对图的所有结点至少遍历一次，即按照某种方式对图的每个结点进行考察，看看我们所求的最短路径是否需要经过当前被考察的结点（如果有结点没有考察过，就无法断定当前找到的路径为最短路径，因为至少还有可能的路径没有考察过）。因此，对图进行遍历就成为寻找最短路径的基础。那么如何对图进行遍历呢？

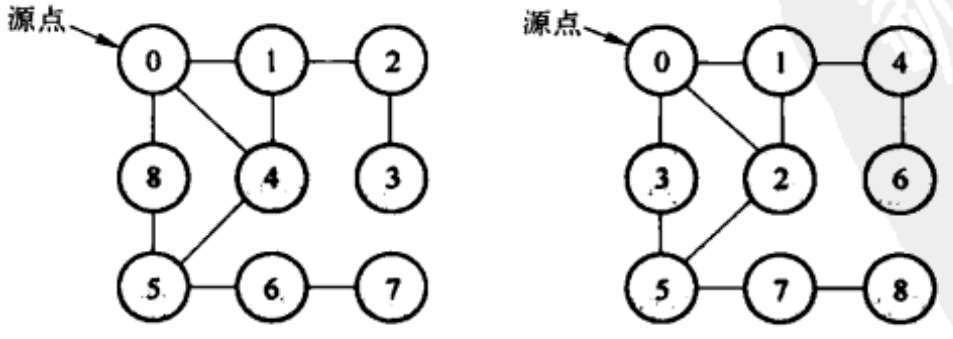
最简单的遍历办法是从源结点 S 开始搜索，先考察源结点的相邻结点，因为这是从源结点出发可以直接到达的地方，所谓的千里之行始于足下。但问题是源结点可能有很多相邻结点，到底应该考虑哪一个呢？由于我们不是无所不知的，因此不知道应该先考察哪一个。在这种情况下，我们有很自然的应对办法：随机。即从源结点的相邻结点里随机选择一个，假定我们随机选择的结点为 v_1。接下来要考虑的问题是，在到达了一个相邻结点 v_1 后，是以新的结点 v_1 为源点重复刚才的过程，还是考察原来源结点 S 的另一个相邻结点呢？而这种不同的思索就导致了图论里的深度优先搜索（Depth First Search，DFS）和广度优先搜索（Breadth First Search，BFS）算法的出现。

12.3.1 深度优先与广度优先搜索

如果以行路来进行比喻，深度优先搜索就是一直往前走，只要前面还有路，就往前，直到没有地方可走为止。此时，我们转回到最近的一个岔路口，再探索另外一个分叉。用刚才的例子来说，就是在达到了邻结点 v_1 后，以新的结点 v_1 为源点重复刚才的过程。

广度优先搜索则是以源点为圆心，画出一个个越来越大的圆圈，探索过程按照这些圆圈一圈圈往外搜索直到所有结点都被访问为止。用刚才的例子来说，就是在达到了邻结点 v_1 后，返回到原来的源结点 S，考察 S 的另一个邻结点。

图 12-8 给出的是这两种不同搜索算法搜索同一幅图的结果。图中的编号即为该结点被发现的顺序（注意，图中给出的顺序不是唯一的）。



a) 深度优先搜索　　b) 广度优先搜索

图 12-8　深度优先搜索和广度优先搜索

按照上面的陈述，我们可以很快获得深度优先搜索算法：

```
DEPTH-FIRST-SEARCH(G)
for (all v in G)
    visited[v]=false;              //初始时所有结点都未被探索过
for (all v in G)                   //逐个考察图里的每个结点
    if (!visited[v])               //如果被考察结点 v 是第 1 次碰到，考察本结点
          TRAVERSE(v, visited);

TRAVERSE(v, visited)               //结点考察子过程
visited[v] = true;                 //给本结点打上标记，表示结点 v 已被考察过
for (all w adjacent to v)          //逐个考察结点 v 的邻结点
    if (!visited[w])               //如果邻结点 w 是第 1 次碰到，则考察该邻结点
          TRAVERSE (w, visited);
```

同样，广度优先搜索算法也可以很容易从上面的描述里推导出来。不过，由于我们是一圈圈地往外搜索，因此在每个结点上需要搜索该结点的所有相邻结点，然后依次对这些邻结点进行考察。只有考察完这些邻结点后，才能考察邻结点的邻结点。这种圆圈由一个队列 Q 来维持。

```
BREADTH-FIRST-SEARCH(G)
for (all v in G)
    visited[v]=false;              //初始时所有结点都未被探索过
for (all v in G)                   //逐个考察图里面的每个结点
    if (!visited[v])               //如果被考察结点 v 是第 1 次碰到
        将结点 v 添加在队列 Q 末尾;      //将其置于队列 Q 的末尾
        while(Q≠∅){                //逐个依次考察队列 Q 里面的结点
            取出队列 Q 的头结点 w;
            if(!visited[w]){       //如果被考察结点 w 是第 1 次碰到，则考察该结点
                visited[w]=true;
                将 w 的所有邻结点添加到队列 Q 末尾;     //形成下一次扩展的圆圈
            }
            从队列 Q 中删除结点 w;
 }
```

显然，对一个图进行遍历所需的时间与结点个数呈线性关系，因为每个结点只需遍历一次。因此，其时间成本为 $O(V)$。对上面两个算法的伪代码程序上进行分析，也得到同样的结论。

在自然中，广度优先现象更为普遍。例如，火灾爆发后，其蔓延的趋势呈现出广度优先模式，即离火源点越近的地方，其着火的紧迫度就越高。另外，洪水、病毒（生物或者数字病毒）、火山、地震、台风等的破坏效应也是一圈圈地往外扩展。

在人类社会中，深度优先现象更为普遍。例如，皇帝总是将位子传给儿子，儿子传给孙

子，在没有儿子或孙子可传的情况下，则传给兄弟或别的亲戚。没有亲戚的时候传给朋友等。总是在某条线索上进行最深度搜索不果后，才会回溯考察别的线索。当今社会里的裙带关系也呈现出深度优先的模式。

12.3.2 深度优先解法

12.3.1 节说过，深度优先遍历和广度优先遍历的时间成本都是 $O(V)$，似乎是一种不错的搜索最短路径的办法。但仔细考察发现，使用深度或广度优先搜索算法来寻找一对给定结点之间的最短路径的时间成本不会这么低！图的遍历只是对每个结点考察一次即可，而寻找最短路径则至少每条边都需要查看一下，因为我们的目的不是仅仅查看由源点出发能够达到哪些结点，而是要查看到达某一个结点的最短路径。因此，寻找单源单点最短路径的时间复杂性将远远超过图的遍历的复杂性，其下限不会好于$\Omega(E)$。

再思考一下可以得知，在寻找一对结点之间的最短路径的时候，每条边的考察次数不一定是一次，而很有可能是多次，因为一条边可以包括在从 S 到 T 的多条路径中！由此我们判断，寻找单源单点最短路径的时间复杂性将大大超过边的条数。到底超过多少呢？分析便知。我们先对深度优先搜索的遍历算法进行修改以使其能够搜索最短路径。

采用 DFS 的单源单点最短路径算法如下：

1）从源点开始，初始化总成本初值为∞。

2）从源点开始，按照 DFS 算法对图进行遍历，每经历一个新的结点时，对总成本更新如下：

a）如果总成本为∞，将新的成本（当前结点到新结点的边的权重）设置为总成本。

b）如果总成本不等于∞，则将新的成本加到总成本上。

3）如果抵达目标结点，则将该条路径及其长度记录下来，我们找到了一条路径。

4）如果无法往前推进，则进行回溯，回溯的时候需要将被回溯的边的权重从总成本中减去。如果总成本减到 0，则将总成本设置为∞（表示从源点开始探索另一条路径）。

5）当 DFS 遍历结束后，比较所有找出来的路径，长度最短的即为最短路径。

DFS 解法的思路是：每次推进到一个新结点时，继续往前推进，即考虑从新的结点能够直接到达的结点。在这些能够到达的结点里随机选取一个，并将刚才这一步的成本（边的权重）加到总成本上。然后在这个新的结点上继续 DFS 算法，直到抵达目标结点或无法往前推进时为止。如果抵达目标结点，则获得一条从源点到目标点的路径及其长度（如图 12-9 中的迦太基→叙拉古→坎尼→罗马，长度为 6）。我们记录下这个长度作为当前最短路径长度。然后回溯到上一个结点继续 DFS 搜索，即选择另一个邻结点进行考察（如图 12-9 在叙拉古时选择普兰尼进行考虑）。当一个结点的所有邻结点都已考察过时，则回溯到上一个结点（如叙拉古的所有邻结点都考察完后，回溯到迦太基）。而每次找到一条不同的路径时，都需要与当前找到的最短路径比较，留下更短的。这样最后将获得一条最短路径！

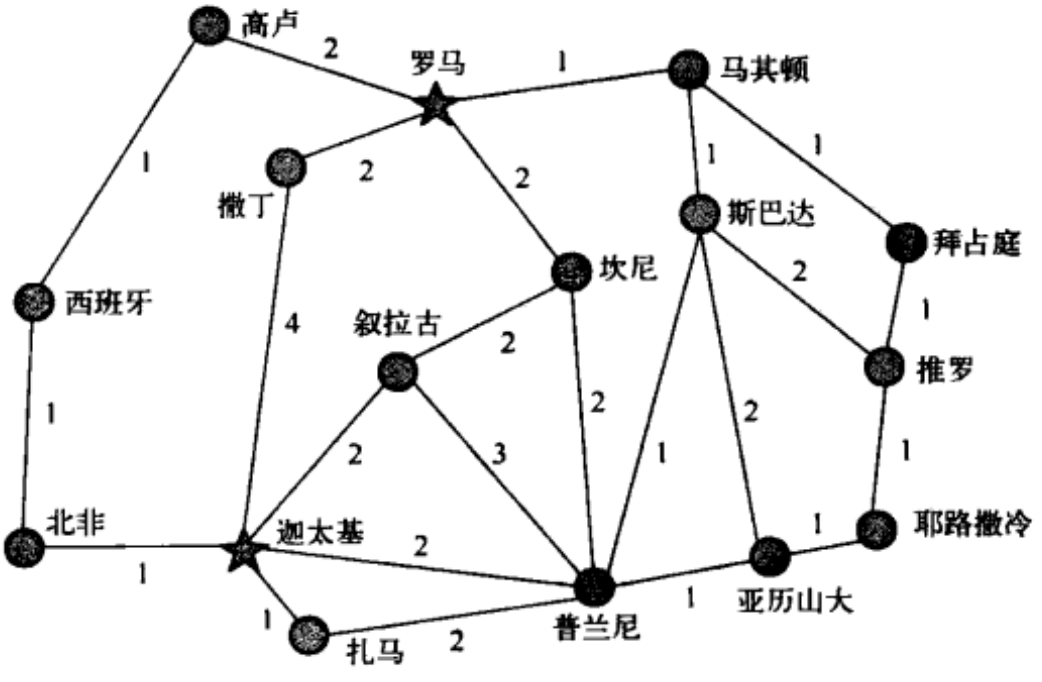


图 12-9 罗马地图上的深度优先搜索和广度优先搜索

这种算法的时间复杂性是多少呢？显然，这个算法的时间复杂性与一对结点之间的路径数成正比，那么一对结点之间可能存在多少条路径呢？在最坏情况下，假定我们不考虑包含环路的路径，则每推进一个结点，下一步要考虑的结点数将减 1。这样，一对结点间的路径条数至少是$(V-2)!$，至多则是$(V-1)!$。用渐近表示，这个路径数为 $O(V!)$。阶乘级的复杂性显然太高，因为它比任何多项式复杂性的数量级都高，属于难解问题范畴！

使用 BFS 情况也将一样。读者可自行对 BFS 遍历算法进行修改来解决单源单点最短路径问题。

当然，我们可以使用分支限界来减少需要考虑的路径数量，即在某一条路径长度已经超过当前最短路径长度时，该条路径就无需再搜索下去，从而减少搜索空间。但即使如此，平均情况下的搜索复杂性为 $V!/2$，仍然太高！

不过有一个例外：如果图的所有边的权重相等，或者是无权重的图，则可以使用 BFS 获得一个效率相当不错的算法，这个留给读者自己思索。

对于一般的带权重的图来说，有没有更好的办法呢?

如果是无向图，求取一对结点之间的路径可能没有更好的办法了，因为我们刚才已经说过，在最坏情况下，每条路径都需要考虑，而路径总数的复杂性在最坏情况下就是 $O(V!)$。

12.4 单源多点最短路径问题

难道情况有如此糟糕吗？当然没有。

如果细心观察我们前面的单源单点最短路径解法，我们发现，在搜索一对结点之间的最短路径的同时，我们已经考察了源结点到任意结点的所有路径！既然如此，我们何不一次性找出源结点到所有结点的最短路径？这样平摊下来（还记得摊销分析吧），花费在一对结点之间的最短路径上的成本将大为降低！

更为重要的是，将寻找一对结点之间的最短路径改变为一个结点到所有其他结点的最短路径问题还赋予我们一个更有利的优势：贪心策略的使用。而这将使得最短路径的搜索成本大为下降。贪心加摊销，就使得最短路径问题就从一个难解问题变成一个易解问题！

这就是狄杰斯特拉（Dijkstra）算法。该算法由荷兰计算机科学家埃德加·沃波·狄杰斯特拉（Edsger Wybe Dijkstra）于 1959 年发明，用来求解一个源点到所有其他结点之间的最短路径，即所谓的单一源点最短路径问题，或**单源多点最短路径问题**。该问题的定义如下：

给定一个源点 $s \in V$，找出该结点到图 G 其他所有结点的最短路径权重 $\delta(s, v)$，这里 $v \in V$。

如果边的权重为非负，则所有最短路径都存在。

12.4.1 最短路径的性质

Dijkstra 算法的特点是摊销加贪心。那么最短路径问题具备贪婪选择属性吗？

我们随便考察一条最短路径，例如从 v_1 到 v_6 的最短路径，如图 12-10 所示。显然，这条路径上中间任意两个结点之间的路径是相应两个结点之间的最短路径！例如，该路径上的 v_2 到 v_5 路径段就是结点 v_2 到 v_5 的最短路径。这点非常容易证明：如果还存在一个更短的 v_2 到 v_5 的路径，则可以用这个更短的路径替换图中的 v_2 到 v_5 路径段，从而获得一个更短的 v_1 到 v_6 的路径。而这与假设 v_1 到 v_6 已经是最短路径相矛盾！

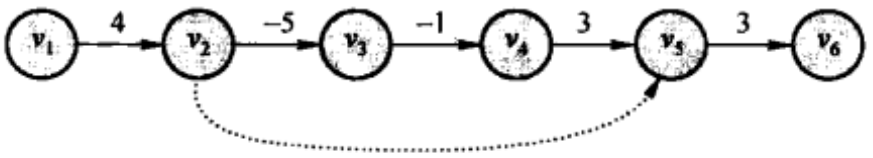


图 12-10 最短路径的最优子结构

这样，我们就获得了贪婪策略使用的第一个条件：最优子结构。也就是说，最短路径里的任意一段路径都是相关两个结点之间的最短路径。

那么最短路径问题是否还具有贪婪选择属性呢？答案是肯定的。

从源结点的角度来看，路径往外延伸的下一个结点就是离源点距离最近的结点。为什么这样说呢？因为这个选择是一个最优选择：该结点离源点的距离就是源点到该结点的最短路径。也就是说，我们做出这种选择后，即获得了源点到一个新的结点的最短路径！而证明这点很简单：只需要证明该结点目前到源点的距离在将来不会因经过别的结点而变得更短即可。

这个证明只需要使用三角不等式即可完成。

三角不等式 对于任意三个结点 u，v，$x \in V$，有：

$$\delta(u, v) \leqslant \delta(u, x) + \delta(x, v)$$

这个性质可以很容易地用反证法证明：假设该不等式不成立，即 $\delta(u, v) > \delta(u, x) + \delta(x, v)$，则我们把从 u 到 x 的最短路径连接 x 到 v 的最短路径，将获得一个更短的从 u 到 v 的路径，从而与 $\delta(u, v)$已经是 u 到 v 之间的最短路径权重相矛盾。这个性质在图 12-11 里可以看

得更清楚。

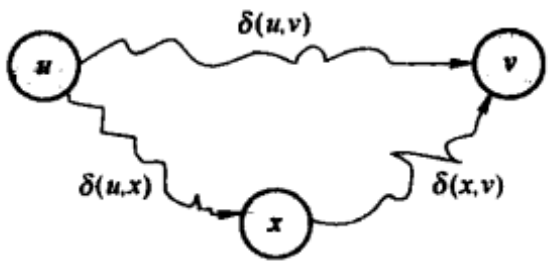


图 12-11 最短路径问题的三角不等式

有了上面的最优子结构和贪婪选择属性，就可以构造最短路径算法了：

1）将所有结点分为两部分：一部分是最短路径已经知道的，另一部分是最短路径尚不清楚的。

2）一开始，已知最短路径的结点集合为空，所有结点都属于路径尚不清楚部分。

3）在算法的整个过程中，维持一张表，表里面是每个结点到源点的当前最短距离。

4）一开始，源点距离为 0，而其他结点到源点的距离都为无穷。

5）循环往复下面的步骤直到所有结点的最短路径都已找出为止：

a）从尚不知道路径的结点里面选取离源点距离最短的结点 u（第 1 次选择是源点本身）。

b）将 u 加到已知路径结点集合，结点 u 当前的距离就是源点到 u 的最短路径。

c）对所有在路径尚不清楚且又与 u 相邻的结点 v（有边连接的结点）进行如下操作：

i. 将 S 到 u 的最短路径加上边(u,v)的权重，获得值 D。

ii. 将 D 与 v 当前离 S 的距离进行比较，取其小者作为 v 到 S 的新距离。

iii. 重复第 5 步。

上述算法里面的第 c 步操作的目的是降低结点 v 到源点 S 的距离，因此被称为**降距操作**（decrease-distance）。

这就是 Dijkstra 算法。

12.4.2 Dijkstra 最短路径算法

该算法的伪代码实现如下：在任何时候维持一个结点的集合 S，该集合里结点到源点 s 的最短路径都已经找到；而对于 S 集合之外的结点，每次添加进入 S 的是离 S 距离最近的结点 u；对于 u 的所有邻结点，更新其到源点的距离。

```
Dijkstra-SHORTEST-PATH-ALGORITHM(G, V, s)
1.  d[s]=0;                          //源点到源点的距离当然是 0
2.  for (each v∈V-{s})               //所有其他结点到源点的距离初始化为无穷大
3.      d[v]=∞;
4.  S=∅;                             //一开始，路径已知的结点集合为空
5.  Q=V;                             //Q 是 V-S 结点集合组成的优先队列，开始为所有结点
```

```
6.  while (Q ≠∅) {              //只要队列Q里还有结点，即循环寻找最短路径
7.      u=EXTRACT-MIN(Q);       //找出Q里离源点距离最短的结点u
8.      S=S∪{u};                    //结点u的最短路径已经找到，因此加入集合S
9.      for (each v∈Adj[u])     //对u的邻结点实施降距操作
10.         if(d[v]>d[u]+w(u,v))//降距操作
11.             d[v]=d[u]+w(u,v);
12. }
```

12.4.3 Dijkstra 算法举例

我们以图 12-12 为例，用 Dijkstra 算法求从结点 *A* 到所有其他结点的最短路径。

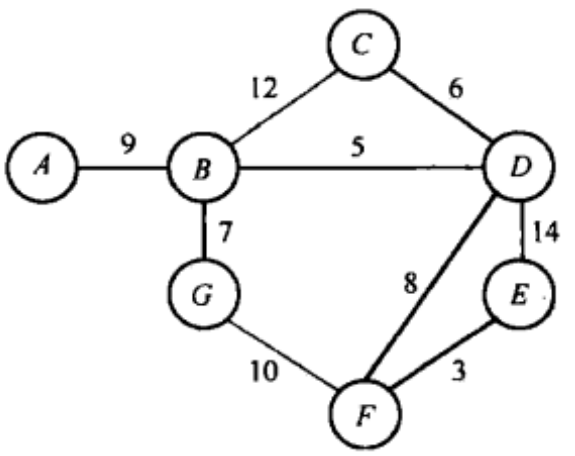


图 12-12 Dijkstra 算法演示用图

一开始 *S* 初始化为 ∅，而所有结点到源点 *A* 的距离初始化如下：

结　点	*A*	*B*	*C*	*D*	*E*	*F*	*G*
离源点距离	**0**	∞	∞	∞	∞	∞	∞

在所有不属于 *S* 的结点里选取离源点距离最短的结点，无疑是源点本身，将源点 *A* 加入到 *S*，此时 *S*={*A* }。每个在 *V*–*S* 里的结点到源点的距离也因为 *A* 加入到 *S* 而需要更新：

结　点	*A*	*B*	*C*	*D*	*E*	*F*	*G*
离源点距离	**0**	**9**	∞	∞	∞	∞	∞

在所有不属于 *S* 的结点里选取离源点距离最短的结点，无疑是 *B*，将 *B* 加入到 *S*，此时 *S*={*A*，*B*}。每个在 *V*–*S* 里的结点到源点的距离也因为 *B* 加入到 *S* 而需要更新：

结　点	*A*	*B*	*C*	*D*	*E*	*F*	*G*
离源点距离	0	9	**21**	**14**	∞	∞	**16**

在所有不属于 *S* 的结点里离源点距离最短的结点是 *D*，将 *D* 加入到 *S*，此时 *S*={*A*，*B*，*D*}。每个 *V*–*S* 里的结点到源点的距离也因为 *D* 加入集合 *S* 而需要更新：

结　点	A	B	C	D	E	F	G
离源点距离	0	9	20	14	28	22	16

下一步选择的结点将是 G，此时 $S=\{A, B, D, G\}$。每个 $V-S$ 里的结点到源点的距离也因为 G 的加入集合 S 而需要更新（没有更新）：

结　点	A	B	C	D	E	F	G
离源点距离	0	9	20	14	28	22	16

下一步选择的结点将是 C，此时 $S=\{A, B, D, G, C\}$。每个 $V-S$ 里的结点到源点的距离也因为 C 加入而需要更新（没有更新）：

结　点	A	B	C	D	E	F	G
离源点距离	0	9	20	14	28	22	16

下一步选择的结点将是 F，此时 $S=\{A, B, D, G, C, F\}$。每个 $V-S$ 里的结点到源点的距离也因为 F 加入集合 S 而需要更新：

结　点	A	B	C	D	E	F	G
离源点距离	0	9	20	14	25	22	16

下一步选择的结点将是 E，此时 $S=\{A, B, D, G, C, F, E\}$。所有结点都囊括在 S 里，算法结束。每个结点到源点的最短距离如上表所示。达到这些最短距离的最短路径如图 12-13 所示（实线为路径）。

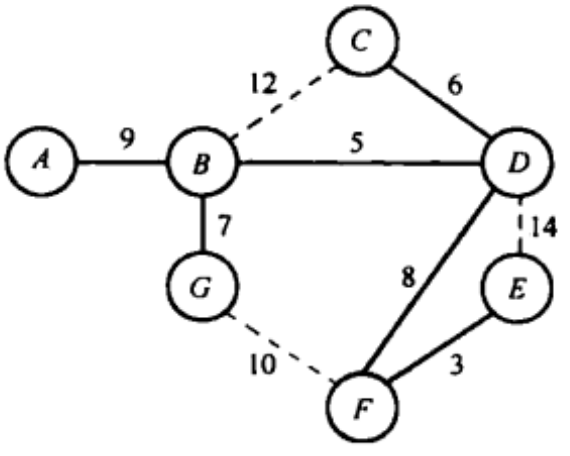


图 12-13　Dijkstra 算法找出来的从 A 到所有结点的最短路径（实线）

12.4.4　Dijkstra 算法与洪水泛滥

仔细分析 Dijkstra 算法可以发现，它与洪水泛滥有些类似。把图 12-13 的所有结点想象成洪水将要经过的城镇，所有边想象成连接这些城镇的河道，假定洪水在源点 A 城突然爆发

(见图 12-14)。

在时刻 0，洪水从源头 A 城开始泛滥。洪水监测预报员可以估计洪水到达 B 城、C 城、G 城、D 城的时刻分别为第 9 小时和第 1 000、1 000、1 000（即相当于无穷）小时。前者是最保守的估计，情况不可能更糟了。假设洪水监测员的职责仅仅是正确预报每个城市最早被淹的时刻，则他大可睡上 min{9，1 000}=9 小时。当洪水到达 B 城时，预报员检查所有与 B 相邻的有河道相连且还未被洪水淹没的城市，发现洪水从 B 城推进到 C 城只需要 12 小时，推进到 D 城只需要 5 小时，推进到 G 城只需要 7 小时，于是他将 C 城、D 城、G 城被淹的时刻提前到第 21、14、16 小时。而这个将时刻提前的动作就是 Dijkstra 算法里面的降距操作。

图 12-14 Dijkstra 算法与洪水泛滥似乎有点联系

12.4.5 Dijkstra 算法的正确性

从上面的例子来看，Dijkstra 算法无疑是正确的。从我们为引入 Dijkstra 算法而进行的推理来看，Dijkstra 算法也是正确的，但光靠例子或推理来说明正确性显然不够。我们需要的是理论证明。证明 Dijkstra 算法正确就是要证明对于任意结点 $v \in V$，$d[v] = \delta(s, v)$成立。

我们分三个步骤来证明上述等式对于所有结点成立：第一步证明任何处于集合 S 外面的结点，其距源点的距离不短于其至源点的最短路径长度；第二步证明，任何一个结点的最短路径在其前驱结点的最短路径确立后才能确立；第三步证明任意结点在加入到集合 S 的时候，其离源点的最短路径已经确定，也就是等式 $d[v] = \delta(s, v)$成立。

1. 第一步

引理 1 在 Dijkstra 算法初始化后，对于任意一个结点 $v \in V$ 来说，不等式 $d[v] \geqslant \delta(s, v)$ 必成立，且是整个算法的一个不变式，在算法执行的整个过程中都成立。

证明 首先，在初始化后，除源点外每个结点的距离都被设置为无穷大，因此，不等式 $d[v] \geqslant \delta(s, v)$成立。接下来我们用反证法证明该不变式在算法的所有步骤都成立。由于结点到源点的距离只在降距操作时改变，因此，若要不变式不成立，则必在某次降距操作时发生。设 v 是第一个在降距操作时打破不变式的结点，即 $d[v] < \delta(s, v)$。又设 u 是导致结点 v 降距的

结点，即 $d[v]=d[u]+w(u,v)$，则有：

$$\begin{aligned} d[v] &< \delta(s,v) && \text{（反证假设）}\\ &\leqslant \delta(s,u)+\delta(u,v) && \text{（三角不等式）}\\ &\leqslant \delta(s,u)+w(u,v) && \text{（最短路径显然短于任意具体路径）}\\ &\leqslant d[u]+w(u,v) \end{aligned}$$

即 $d[v]<d[u]+w(u,v)$，而这与 $d[v]=d[u]+w(u,v)$矛盾！因此，引理 1 必成立。 □

2. 第二步

引理 2 设结点 u 是从源点到结点 v 的最短路径上结点 v 的前驱（紧挨着 v 的结点）。如果 $d[u]=\delta(s,u)$，则在边(u,v)用于降距后，有 $d[v]=\delta(s,v)$.

证明 首先，我们注意到，由于 u 是 v 的前驱，则有：

$$\delta(s,v)=\delta(s,u)+w(u,v)$$

其次，假设在降距操作考虑边(u,v)前有 $d[v]>\delta(s,v)$ （不然引理已经成立）。

综合上述等式和引理里的条件，则有：

$$d[v]>\delta(s,v)=\delta(s,u)+w(u,v)=d[u]+w(u,v)$$

即 $d[v]>d[u]+w(u,v)$，因此，在算法对 u 的邻结点进行降距操作时将设置：

$$d[v]=d[u]+w(u,v)=\delta(s,v)$$

□

3. 第三步

定理 Dijkstra 算法终结时，$d[v]=\delta(s,v)$，这里 $v\in V$。

证明 我们只需要证明在 v 被加入到集合 S 的时候 $d[v]=\delta(s,v)$即可。

使用反证法：假设 u 是第一个被加入到集合 S 的违反上述等式的结点，即 $d[u]>\delta(s,u)$。

那么从源点到 u 的最短路径必经过 S 集合外面的某个或某些结点（为什么）。设 y 是该最短路径经过集合 S 外面的第一个结点，又设该路径上结点 y 的前驱是结点 x，则 x 必然属于集合 S。那么在结点 u 被加入到集合 S 前，有图 12-15 所示情景（阴影部分为 S）。

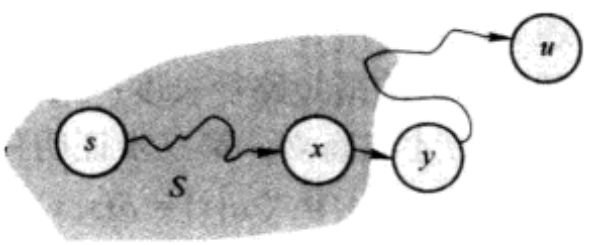


图 12-15 结点 u 加入集合 S 前的状态

由于 u 是第一个违反引理等式 $d[v]=\delta(s,v)$的结点，则有 $d[x]=\delta(s,x)$。

当 x 被加入到 S 的时候，x 的邻结点 y 的距离将被降距，而这意味着

$$d[y]=\delta(s,y)\leqslant\delta(s,u)<d[u]$$

但是 u 的选择使得 $d[u]\leqslant d[y]$，从而产生矛盾。因此，引理成立。 □

由此可见，Dijkstra 算法确实能正确计算出源点 s 到所有结点的最短路径。

12.4.6 Dijkstra 算法的时间复杂性

显然，在 Dijkstra 算法里，第 1、4、5 三行步骤执行所需的时间为常数时间，可以忽略不计。第 2、3 两行的循环需执行 V 遍，每遍的时间复杂性为常数，因此，总时间成本为 $O(V)$。因此，该算法时间成本由第 6 行开始的 while 循环所支配。

为方便起见，我们将 while 循环重新拷贝如下：

```
6.  while (Q ≠∅) {                  //只要队列 Q 里还有结点，即循环寻找最短路径
7.      u =EXTRACT-MIN(Q);          //找出 Q 里离源点距离最短的结点 u
8.      S=S∪{u};                    //结点 u 的最短路径已经找到，因此加入集合 S
9.      for (each v∈Adj[u])         //对 u 的邻结点实施降距操作
10.         if (d[v]>d[u]+w(u,v))   //降距操作
11.             d[v]=d[u]+w(u,v);
12. }
```

显然，while 循环执行的次数为图中的结点数，即 V。循环里的步骤中，第 8 行为常数时间，可以忽略不计。需要讨论的是 EXTRACT-MIN(Q)和 for 循环。操作 EXTRACT-MIN(Q)的功能是找出优先队列 Q 里取值最小的元素（即离源点距离最近的结点）。该操作的执行时间当然与优先队列的实现有关。而 for 循环完成的功能是对 u 的邻结点进行降距。那么在整个算法中，要进行多少次降距操作呢？显然，降距操作的次数与 for 循环的执行次数相当，而 for 循环的执行次数则与每个结点的邻结点个数有关。如果直接计算特定结点 u 的邻结点数，那么会陷入到按最坏情况估计为 V-1，从而导致对整个算法运行中 for 循环次数的计算结果为 V^2 而大大超过现实情况。但如果我们注意到每条边只可能在一次降距操作中使用，就可以得出整个算法运行中 for 循环的执行次数为 $O(E)$。

由上述分析，得出 Dijkstra 算法的时间复杂性为：

$$T_D = \Theta(VT_{\text{EXTRACT-MIN}} + ET_{\text{DECREASE-DISTANCE}})$$

而要让 Dijkstra 算法的时间成本维持在低水平上，则需要高效完成最小元素提取和降距两个操作，即尽量降低 $T_{\text{EXTRACT-MIN}}$ 和 $T_{\text{DECREASE-DISTANCE}}$。这两个操作的成本实际上都是搜索的成本：$T_{\text{EXTRACT-MIN}}$ 是从一堆元素里面寻找最小值，即次序选择问题；$T_{\text{DECREASE-DISTANCE}}$ 是从一堆元素里面寻找一个特定的元素（u 的邻结点）。而次序选择和搜索分别在本书第 10 和第 11 章讨论过了。剩下的任务是设计一个数据结构来高效地完成这两个操作。

如果用数组来实现优先队列，本书第 10 章讨论过的次序选择算法告诉我们，在 V 个元素里找出最小元素的成本是 V，而在数组里搜索一个特定元素的成本为 $O(1)$；如果用堆结构来实现优先队列 Q（这是优先队列的通常实现方式），则找出最小元素的成本为 $\log V$，找到特定元素的成本也是 $\log V$（希望读者在此之前学过堆结构。如果没有学过，请参阅数据结构方面的相关资料）；如果用斐波那契堆来实现 Q，则找出最小元素的成本为 $\log V$，但寻找特定元素的成本为 $O(1)$。由此，我们获得表 12-1 所列出的各种数据结构下的 Dijkstra 算法的时间成本。

表 12-1 Dijkstra 算法的时间成本

Q 的实现方式	$T_{\text{EXTRACT-MIN}}$	T_{for}	总 计
数组	$O(V)$	$O(1)$	$O(V^2)$
堆	$O(\log V)$	$O(\log V)$	$O(E\log V)$
斐波那契堆	$O(\log V)$ 聚类分析时间	$O(1)$ 聚类分析时间	$O(E+V\log V)$ 最坏时间

细心的读者可能已经看出，上述时间复杂性与 Prim 的最小生成树算法的时间复杂性一样！（但这是为什么呢？）

斐波那契堆的内容本书没有介绍，主要是两个理由：一是这种数据结构不太常见；二是本书尽量不与数据结构联系太过紧密。这些内容应该在数据结构的书籍或教程里介绍。

12.5 Bellman-Ford 算法

Dijkstra 最短路径算法虽然很流行，但却有一个不足：如果图中包含有负权重，则 Dijkstra 算法计算出来的最短路径可能并不是最短路径！例如，如果 Dijkstra 最短路径算法运行于图 12-16 上，则获得的 A 到 C 的最短路径为 2，而真正的最短路径是 1（A 到 B 再到 C）。也就是说，Dijkstra 算法不能解决包含负权重图中的最短路径问题。这是为什么呢？

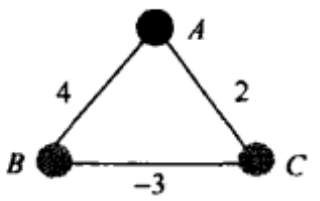


图 12-16 Dijkstra 算法不能计算有负权重的最短路径

这是因为，负权重的引入将导致本章前面介绍过的三角不等式不再成立，而三角不等式是 Dijkstra 算法正确性的前提，它用于证明我们的引理 1。

也许有人会提出用这样一种“等效变换”的方法来解决带负权值边的最短路径问题：设 x 是图 G 所有边中权值最小的边，这里 $x<0$，将图 G 中所有边的权值减去 $x-1$ 到图 G'（保证图 G' 中所有边的权重为正），再对图 G' 执行 Dijkstra 算法，所求得的最短路径也是 G 的最短路径。

但这种等效变换能行吗？从图 12-16 就可以看出，这种方法是行不通的。（为什么？）

也许读者会认为负权重的图不对应世界上任何实际的物理路径，世界上难道有长度为负的道路？因此研究针对负权重的最短路径算法纯粹是浪费时间！

世界上当然不存在负的路径，但谁说过最短路径问题研究的路径就是物理世界里的道路长度呢？如果将图中边的权重不看做是物理道路的距离，而看做是行走该条道路的成本，则负值就没有什么大惊小怪的了。例如，从 A 到 B 有两条路，如果以在路上的花费算作为成本，则其中一条路可以让你在路上赚一笔钱（例如发表一次有偿演说），而另一条路荒无人烟，自然不能赚到任何钱。这样前面一条路有可能导致成本为负（赚的钱超过行路成本）。

另外，图论并不是真的或主要针对抽象的数学想象图形而发明的！它的出现实际上是为了解决各种实际问题而抽象出来的一种工具。例如，著名的线性规划问题、矩阵相乘问题、VLSI 排线布局问题等都可以转换为图的问题来更加有效地予以解决。而这些问题转换成图

之后就很有可能出现负权重和负循环！再说，真正的物理世界里也充满各种负值：听说过一些物理学家所津津乐道的时间旅行吗？经过时间隧道旅行，时间就是负数！（越旅行，越年轻，直到回到胎儿，甚至直到无有呢！）

既然负权重并不是什么荒诞不经的虚构，那么对付它就是算法研究的人的不二责任。

问题是，如何应对负权重呢？

12.5.1 负权重的应对方式

仔细观察发现，Dijkstra 算法的特点是一旦一个结点被加入到集合 S，则这个结点到源点的最短距离便已经最后确定，不再改变。这个结论在权重为正的情况下成立，但在权重为负的情况下就不成立了。即对有负权重边的图，对于任何一个结点来说，加不加入集合 S 没有任何区别：它们到源点的最短路径都不能确定。因此，设立集合 S 变得没有任何必要。

那么什么时候一个结点到源点的最短距离才被最后确定呢？答案是显然的：在降距操作不产生任何新的更短的路径长度时！我们怎么知道降距操作不产生新的更短路径长度呢？

仔细查看 Dijkstra 算法里的降距操作发现，每次降距都是针对某一条边进行，因为降距的核心就是从一条新的边获得更短的路径。因此，改变 Dijkstra 算法的策略，把针对结点进行降距，改为针对每条边进行降距。当一轮降距（即每条边都考察一次后）结束时，如果没有任何距离被降低，则说明最短路径已全部找出。如果在一轮里有距离降低，则需要再来一轮降距。那么最多需要多少轮降距操作呢？

答案是 V 轮。我们稍后就解释为什么这样。

由此，应对负权重的方法也已经水落石出：只需要对 Dijkstra 算法进行一点修改，不再将所有结点划分为两个集合 S 和 V-S，而是针对所有的边进行 V 轮降距操作。这就是贝尔曼-福特算法（Bellman-Ford 算法）。该算法由美国数学家理查德·贝尔曼（Richard Bellman）（对，就是发明动态规划的这个人）和小莱斯特·福特（Lester ford, Jr.）发明。该算法的伪代码程序如下：

```
Bellman-Ford-SHORTEST-PATHS(G, s)
1.  d[s]=0;
2.  for (each v∈V-{s})
3.      d[v]=∞;
4.  for(i=1; i<=V-1; i++)
5.      for(all (u, v)∈E)
6.          if(d[v]>d[u]+w(u,v))
7.              d[v]=d[u]+w(u,v);
```

算法的第 1 行将源点到源点的距离设置为 0，第 2、3 行将其他所有结点到源点的距离设置为无穷。这两步与 Dijkstra 算法一样。从第 4 行开始，Bellman-Ford 算法与 Dijkstra 算法开始不同，而这一步是 Bellman-Ford 算法的核心：对图中所有的边进行 V 轮（实际上是 V-1 轮）降距操作。第 5 行的内循环依次对每条边进行考察，并根据考察情况进行降距操作。而第 6、7 行为实际的降距操作。这里需要注意的是第 5 行内循环在考察边的时候顺序

是任意的。

这个算法是否正确呢？从前面的分析看似乎正确，但我们需要更加有力的证据。不过在进行正确性证明之前，我们先来看一个例子。

给定带有负权重的 5 个结点的图（见图 12-17），则 Bellman-Ford 算法的演变过程如图 12-18 至图 12-29 所示。

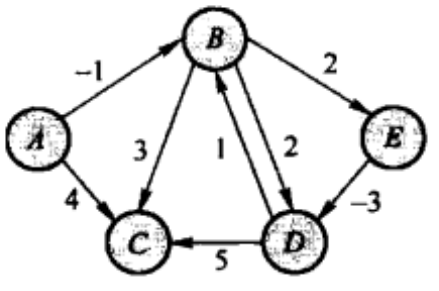


图 12-17 一个带负权重的图

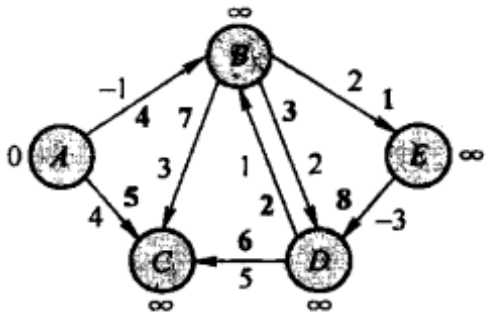


图 12-18 结点距离初始化

算法开始任选一个源点，假定为结点 *A*。然后算法（第 1～3 行）对所有结点的距离进行初始化（见图 12-18）。第 4 行是 Bellman-Ford 算法的核心，一共执行 *V*-1 次。算法第 5 行的内循环依次对每条边进行考虑，并进行相应的降距操作。由于边被考虑的次序是任意的，我们随意假定算法对边的考虑次序如图 12-18 中的黑色标记所示。根据这些标号，首先考虑的边是{*B*,*E*}，由于 *B* 的距离是∞，*E* 的距离也是∞，而 *w*(*B*,*E*)=2，2+∞=∞，因此 *E* 的距离维持为∞，如图 12-19 所示。而考虑的第 2 条边{*D*,*B*}也带来同样的结果，如图 12-20 所示。

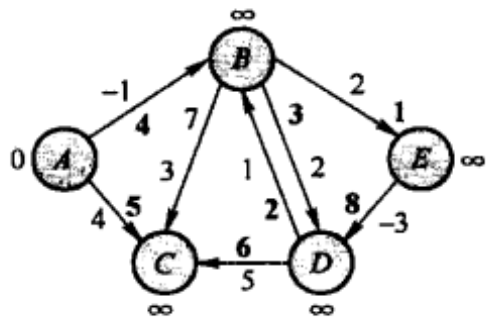


图 12-19 考察边{*B*,*E*}，没有修改

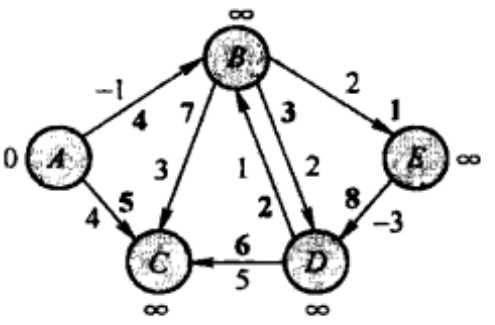


图 12-20 考察边{*D*,*B*}，没有修改

第 3 条考虑的边为{*B*,*D*}，由于 *B* 的距离此时仍然是无穷，*D* 的距离维持不变，如图 12-21 所示。第 4 条考虑的边是{*A*,*B*}，由于 *d*[*A*]=0, *d*[*B*]=∞，*w*(*A*,*B*)=−1，而 0+(−1)=−1<∞，因此，将结点 *B* 的距离降低到−1，即 *d*[*B*]=−1，如图 12-22 所示。

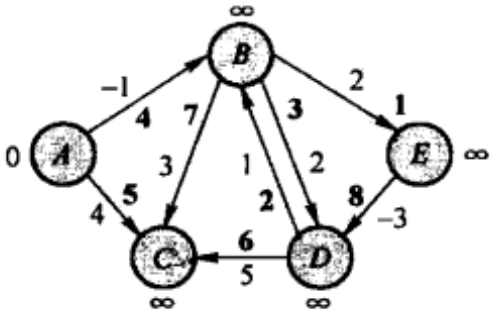


图 12-21 考察边{*B*,*D*}，没有修改

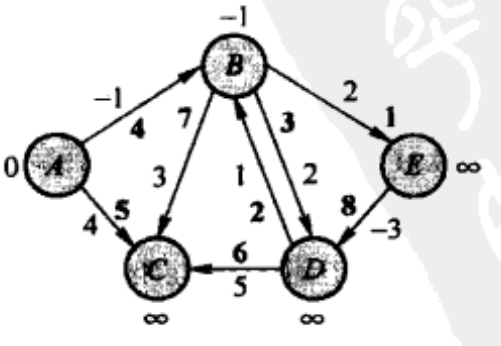


图 12-22 考察边{*A*,*B*}，修改 *d*[*B*]=−1

第 5 条考虑的是边{*A*,*C*}，由于 *d*[*C*]=∞，*d*[*A*]=0，*w*(*A*,*C*)=4，而 0+4=4<∞，因此，调整 *d*[*C*]=4，如图 12-23 所示。第 6 条考虑的是边{*D*,*C*}，由于 *d*[*C*]=4，*d*[*D*]=∞，*w*(*D*,*C*)=5，而 5+∞=∞>4，因此，*d*[*C*]的值维持不变，如图 12-24 所示。

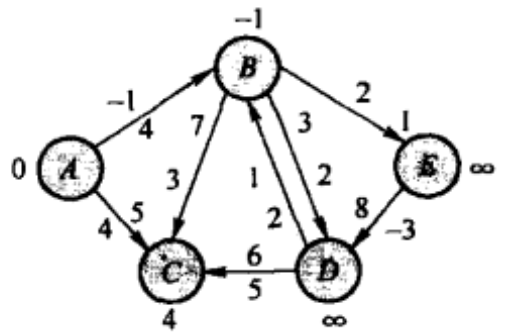


图 12-23　考察边{*A*,*C*}，修改 *d*[*C*]=4

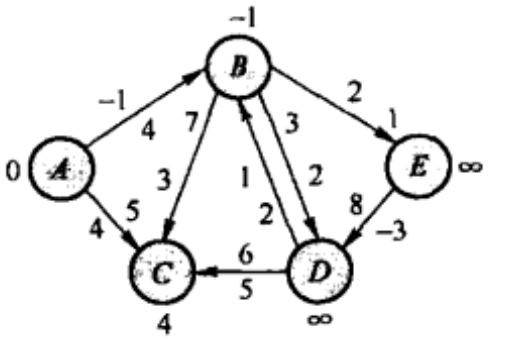


图 12-24　考察边{*D*,*C*}，没有修改

第 7 条考虑的是边{*B*,*C*}，由于 *d*[*C*]=4，*d*[*B*]=−1，*w*(*B*,*C*)=3，而(−1)+3=2<4，因此，调整 *d*[*C*]=2，如图 12-25 所示。第 8 条考虑的是边{*E*,*D*}，由于 *d*[*D*]=∞，*d*[*E*]=∞，*w*(*E*,*D*)=−3，而(−3)+∞=∞，因此，*d*[*D*]的值维持不变，如图 12-26 所示。

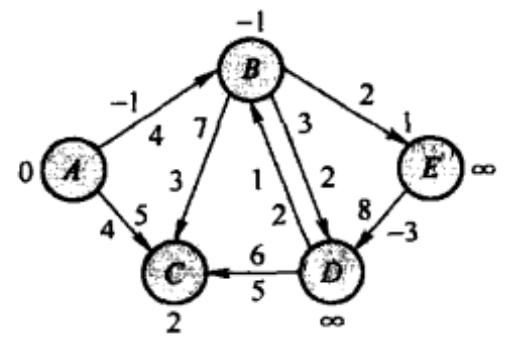


图 12-25　考察边{*B*,*C*}，修改 *d*[*C*]=2

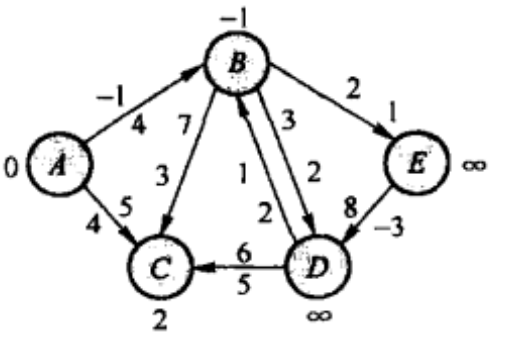


图 12-26　考察边{*E*,*D*}，没有修改

至此，所有的边都考察了一次。第 1 轮循环结束。接下来为第 2 轮循环，即对所有的边进行第 2 次考察。还是按照前面的顺序进行，当考察完所有的边后，获得图 12-27 至图 12-29 所示的结果。为节省篇幅，这里只列出了距离发生降低的情况，没有发生降距操作的情况均略去。

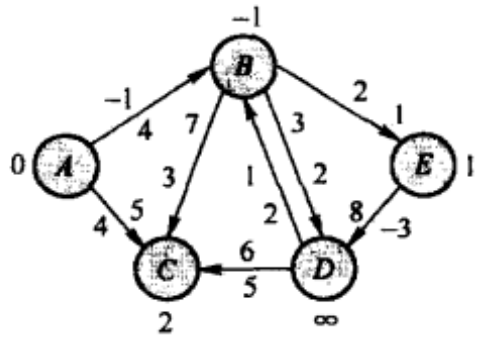


图 12-27　考察边{*B*,*E*}，修改 *d*[*E*]=1

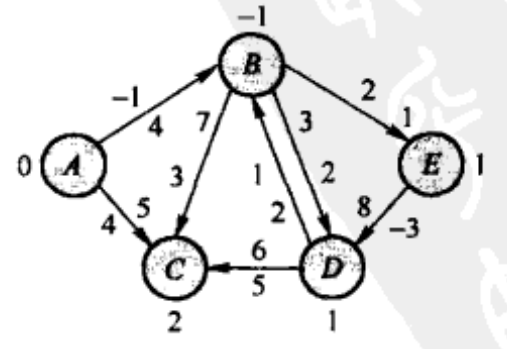


图 12-28　考察边{*B*,*D*}，修改 *d*[*D*]=1

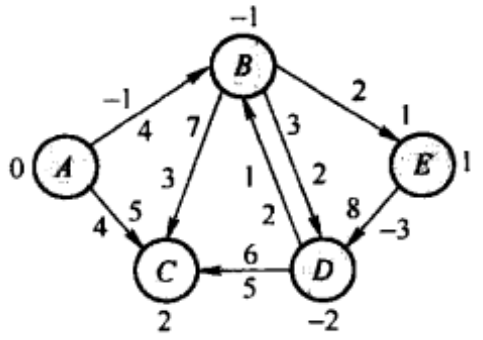


图 12-29 考察边{E,D}，修改 d[D]=-2

然后是第 3 轮、第 4 轮循环，在这两轮的循环中，结点的距离没有发生任何改变。因此，最短距离已经全部最后敲定！

细心的读者可能已经发现，可以对 Bellman-Ford 算法进行一个小改进：$V-1$ 是迭代轮数的上界，实际上可能用不着那么多轮迭代，一轮降距操作过后，若数组 d 没有发生改变，即可结束算法。我们可以通过设标志变量来检测数组 d 是否改变。

12.5.2 Bellman-Ford 算法的正确性

从 12.5.1 节的例子可以看出，Bellman-Ford 算法确实能够在负权重的情况下计算出最短路径，但是只看上去正确是不够的。下面我们就进行严格的证明。

定理 如果图 $G=(V, E)$不包括负循环，则在 Bellman-Ford 算法终止后，$\forall v \in V$, $d[v] = \delta(s, v)$。

证明 设 $v \in V$ 为任意结点。我们考虑从源点 s 到任意结点 v 的最短路径 p，如图 12-30 所示。

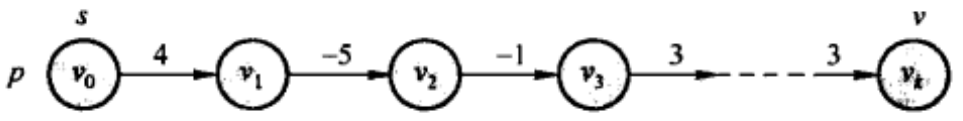


图 12-30 从源点 s 到任意结点 v 的最短路径将被 Bellman-Ford 算法找出

由于 p 是最短路径，则有：$\delta(s, v_i) = \delta(s, v_{i-1}) + w(v_{i-1}, v_i)$。

在算法开始，第 1 行的初始化将把源点 s 的距离设置为 0，即 $d[v_0] = 0 = \delta(s, v_0)$，并且 $d[v_0]$在算法的执行过程中保持不变。

在第 1 轮降距循环中，由于所有的边都考虑了一遍，因此，边(s, v_1)一定会被考虑到，从而导致 v_1 的最短距离被更新为真正的最短路径长度，即 $d[v_1] = \delta(s, v_1)$。

在第 2 轮降距循环中，由于所有的边又轮转考虑了一遍，因此，边(v_1, v_2)一定会被考虑到，从而导致 v_2 的最短距离被更新为真正的最短路径长度 $d[v_2] = \delta(s, v_1)+w(v_1, v_2)$，即 $d[v_2] = \delta(s, v_2)$。

……

在第 k 轮降距循环中，由于所有的边又轮转考虑了一遍，因此，边(v_{k-1}, v_k)一定会被考虑到，从而导致 v_k 的最短距离被更新为真正的最短路径长度 $d[v_k] = \delta(s, v_{k-1})+w(v_{k-1},v_k)$，即 $d[v_k] =$

$\delta(s, v_k)=\delta(s, v)$。

由于 G 不包括负循环，因此路径 p 必定是一条简单路径（不包括循环的路径）。如果不是简单路径，则我们可以将路径上的循环去掉，结果仍然是一条最短路径。路径 p 最长只有 V-1 条边，而算法循环的次数恰恰是 V-1。因此，算法结束后，最短路径 p 将被发现。即对于任意结点 v，Bellman-Ford 算法确实能将源点到结点 v 的最短路径求出。 □

通过上面的证明，Bellman-Ford 算法为什么需要进行 V-1 轮降距操作也清楚了！

12.5.3 负循环检查问题

Bellman-Ford 算法虽然能应对负权重，但却不能求解有负循环的所有最短路径。我们前面说过，如果存在负循环，则有的结点之间可能不存在最短路径。例如，在图 12-31 中，结点 u、v 之间因有一个负环路而不存在任何最短路径。

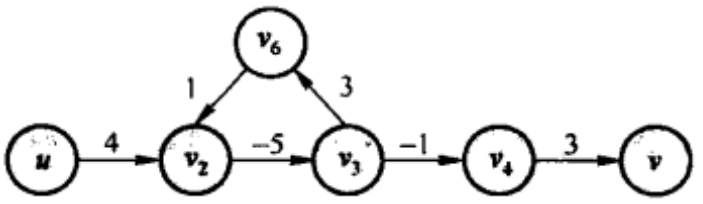


图 12-31 由于负环路的存在，结点 u 到结点 v 之间不存在最短路径

既然不存在最短路径，那么讨论或求解最短路径就没有意义。但我们可以检测一个图是否存在负循环。仔细分析我们对 Bellman-Ford 算法的正确性证明发现，如果没有负环路，最短路径包含的边的条数最多只有 V-1 条，因此，在 V-1 轮降距操作后，所有最短路径都已经找出。这也告诉我们，如果存在负环路，则某些最短路径包含的边的条数将超过 V-1（实际上为无穷），因此，在 V-1 轮降距操作后，仍然存在 $d[v] > d[u] + w(u, v)$的情况。由此，得出下面的推论。

推论 如果任意 $d[v]$在 $V-1$ 遍循环后还是不收敛（即仍可降低），则该图存在一个负循环。

根据上述推论，可以对 Bellman-Ford 算法进行修改，在算法结束后再增加一轮降距检查：如果有 $d[v] > d[u] + w(u, v)$的情况出现，则存在负循环。

下面是修改后的 Bellman-Ford 算法。

```
Bellman-Ford-SHORTEST-PATHS-II(G, s)
1.  d[s]=0;
2.  for(each v∈V-{s})
3.      d[v]=∞;
4.  for(i=1; i<=V-1; i++)
5.      for(each edge(u, v)∈E)
6.          if(d[v]>d[u]+w(u,v))
7.              d[v]=d[u]+w(u,v);
8.  for(each edge(u,v)∈E)
```

```
9.      if(d[v]>d[u]+w(u,v))
10.         报告负环路存在;
```

12.5.4 Bellman-Ford 算法的时间复杂性

该算法的时间复杂性分析非常容易看出：第 1 行的成本为常数，第 2、3 行的成本为 V，第 4、5 两行为一个嵌套循环，外循环 V 次，内循环 E 次，因此该嵌套循环的时间成本为 $O(VE)$；第 8 行的循环执行次数为 E。因此 Bellman-Ford 算法的总时间成本为：

$$O(1)+O(V)+O(VE)+O(E)=O(VE)$$

该复杂性超过了 Dijkstra 算法的 $O(E\log V)$时间。因此，在没有负权重的情况下，应该尽量使用 Dijkstra 算法。

12.6 多源多点最短路径问题

至此，我们讨论了单源单点和单源多点最短路径问题。单源单点最短路径解决的是某对特定结点之间的最短路径问题，而单源多点最短路径解决的是从一个结点至所有其他结点的最短路径问题。对于没有权重或权重均等的图来说，我们可以使用广度优先搜索算法来计算最短路径，时间复杂性为 $O(E+V)$（这个问题的分析留给读者去完成）；如果权重不均匀，但是非负，则使用 Dijkstra 算法，时间复杂性为 $O(E+V\log V)$；而对于包括负权重的图，则使用 Bellman-Ford 算法，时间复杂性为 $O(VE)$。如果考虑到我们得到的不是一条最短路径，而是从一个结点（源点）至所有其他结点的一共 V-1 条最短路径，则摊销在一条最短路径上的成本最多只有 $O(VE/V)=O(E)$。

这远远低于我们在求取单源单点最短路径时的时间成本 $O(V!)$。因此，同时考虑多条最短路径似乎是一个不错的战略。那么，如果我们再多考虑一些最短路径，是否能够进一步降低每条最短路径上的摊销成本呢？

这个问题就将我们带到所谓的多源多点最短路径问题。多源多点最短路径问题就是寻找所有结点至所有其他结点之间的最短路径。其具体定义如下：

给定输入 图 $G=(V, E)$，$V=\{1, 2, \cdots, n\}$，权重函数 $w: E\rightarrow\mathbf{R}$。

要求输出 $n\times n$ 矩阵，每个矩阵元素 $\delta(i,j)$ 为从结点 i 到结点 j 的最短路径长度。

比起单源多点最短路径问题来说，多源多点最短路径问题需要寻找的最短路径条数增加 V 倍！因此，我们希望摊销到每条最短路径上的成本能够降低 V 倍。

我们的希望能够实现吗？

12.6.1 多源多点最短路径问题解决思路

显然，对于多源多点最短路径问题，最简单的办法是将解决单源多点最短路径的算法运行 V 遍，即针对每个结点运行一遍。这样，对于没有权重或权重均等的图来说，时间复杂性

为 $O(VE)$；如果权重不均匀，但是非负，则使用 Dijkstra 算法，时间复杂性为 $O(VE+V^2\lg V)$；对于一般包括负权重的图来说，则使用 Bellman-Ford 算法，针对每个结点运转一次 Bellman-Ford 算法，时间复杂性为 $O(V^2E)$，摊销到每条最短路径上，成本最坏为 $O(E)$。

我们想象中的改善没有出现。

但是，没有改善也许是因为我们并没有费力气设计新的算法，我们所做的仅仅是把单源多点最短路径算法重复了 V 遍，也许我们应该设计新的算法？

如何设计新的算法呢？

在讨论 Dijkstra 算法时，曾经证明了单源多点最短路径问题具有最优子结构和贪婪选择属性，因此使用了具有贪婪属性的 Dijkstra 算法。而对于带负权重的图，虽然贪婪选择属性不再成立，但最优子结构属性仍然存在。因此，我们自然想到了动态规划。

动态规划策略是否适用于这个问题呢？

如果仔细考察任意一条最短路径，我们发现该最短路径所包含的边数可以从 1 条到 V-1 条。这样，一条从结点 u 到结点 v 的最短路径就可以按如此方法来构造：

1）如果只使用 0 条边，结点 u 到结点 v 的最短路径是什么？

2）如果只使用 1 条边，结点 u 到结点 v 的最短路径是什么？

3）如果只使用 2 条边，结点 u 到结点 v 的最短路径又是什么？

4）……

5）如果使用 V-1 条边，结点 u 到结点 v 的最短路径又是什么？

如果结点 u 到结点 v 的最短路径在增加边的条数后不发生变化，则此时的最短路径就是最终的最短路径。即结点 u 到结点 v 的最短路径已经最后确定。注意，这里假设没有负环路。

此种构造方式有三个特点：

1）初始状态很容易计算，只使用 0 条边的最短路径显然是无穷大，除非 u 和 v 是同一个结点，此时最短路径长度为 0。

2）后续最短路径的计算都将使用到前面已经构造的最短路径。

3）所有这些最短路径里有很多重复。

因此，这种构造方式明显适用动态规划。这样我们就获得了多源多点最短路径问题的第一个解法：直接动态规划算法。

12.6.2 直接动态规划解法

根据 12.6.1 节的分析，定义 $d_{ij}^{(m)}$为 i 到 j 使用最多 m 条边的最短路径的长度，则有：

$$d_{ij}^{(0)}=\begin{cases}0 & i=j\\ \infty & i\neq j\end{cases}$$

但是，对于 $m=1,2,\cdots,n-1$，$d_{ij}^{(m)}$等于多少呢？

显然，要想使用动态规划策略，后面的最短路径就需要使用前面已经构建出来的最短路径来表示。为获得这个表示，我们考察一条从 i 到 j 含有 m 条边的最短路径 p。假定该条最

短路径上 j 的前驱结点为 k，则路径 p 由从 i 到 k 的边数为 m-1 的最短路径和从结点 k 到结点 j 的一条边组成。由于我们并不知道 k，因此需要考察所有的 k，即对图中每个结点 k，我们将从 i 到 k 包含 m-1 条边的路径长度加上 k 到 j 的边的长度的和值 a_{ij} 计算出来，在所有这些和值计算结果里面取最小值。这个最小和值就是从 i 到 j 使用 m 条边的最短路径，如图 12-32 所示。

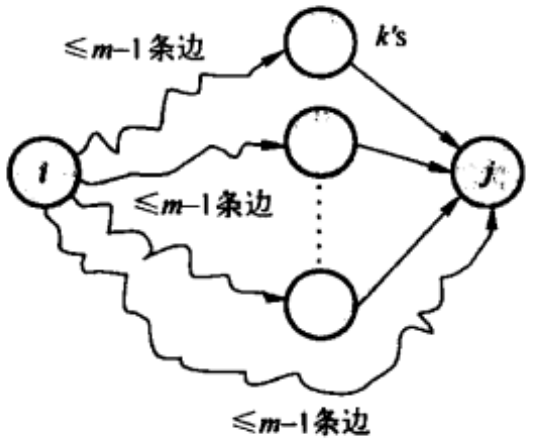


图 12-32 边数为 m 的最短路径由边数为 m-1 的最短路径加一条边构成

根据上述分析，可得递归表达式：

$$d_{ij}^{(m)}=\min_k\{d_{ik}^{(m-1)}+a_{kj}\}$$

用伪代码来表示，即

```
for k=1 to n do
    if dij>dik+akj then dij=dik+akj
```

有了上述结果，多源多点最短路径的直接动态规划算法就水落石出了：

```
DP-SHORTEST-PATHS(G, A)      //多源多点最短路径的动态规划算法，A 为 G 的邻接矩阵
1.  for(m=1;m<=n-1;m++)      //最短路径的边数从 1 递增到 n-1，这里 n=V
2.      for(i=1;i<=n;i++)    //考察每个结点至所有其他结点的最短路径
3.          for(j=1;j<=n;j++)        //对于每个 i，计算 i 到 j 的最短路径
4.              for(k=1;k<=n;k++)    //动态规划的递归表达式
5.                  if(dij>dik+akj)
6.                      dij=dik+akj ;
```

上述算法的正确性已在前面的分析中讨论过了。但该算法的运行时间为 $\Theta(n^4)$（即 $\Theta(V^4)$），与直接使用 Bellman-Ford 算法的最坏时间复杂性一样！看来我们白白折腾了半天。

但真是这样吗？

如果读者够仔细且够有眼光，可能已经看出一点端倪来了。如果将上述动态规划算法最里面一层（第 4～6 行）循环里的执行语句 $d_{ij}=d_{ik}+a_{kj}$ 的“+”号以“×”号替换，这不就是矩阵乘法吗？

矩阵乘法的最坏时间复杂性是 $\Theta(n^3)$！这样我们就获得一个新算法：矩阵乘法。

12.6.3 矩阵乘法解法

如果对操作符进行转换："+" → "min"，"·" → "+"，则矩阵乘法 $c_{ij}=\sum_{k=1}^{n} a_{ik}\cdot b_{kj}$ 将变为 $c_{ij}=\min_k\{a_{ik}+b_{kj}\}$。这样，递归表达式 $d_{ij}^{(m)}=\min_k\{d_{ik}^{(m-1)}+a_{kj}\}$通过变换就可以推导出：

$$\boldsymbol{D}^{(m)}=\boldsymbol{D}^{(m-1)}\cdot\boldsymbol{A}$$

这里 $\boldsymbol{A}$ 是图的邻接矩阵，$\boldsymbol{D}^{(m)}$是所有结点对之间边数为 m 的最短路径长度组成的矩阵，其第 i 行第 j 列的元素就是 $d_{ij}^{(m)}$。这样，最短路径问题就转换为矩阵乘法问题。

根据前面一节的分析，有：

$$\boldsymbol{D}^0=(d_{ij}^{(0)})=\begin{pmatrix}0&\infty&\infty&\infty\\\infty&0&\infty&\infty\\\infty&\infty&0&\infty\\\infty&\infty&\infty&0\end{pmatrix}$$

这样，有：

$$\boldsymbol{D}^{(1)}=\boldsymbol{D}^{(0)}\cdot\boldsymbol{A}=\boldsymbol{A}^1$$
$$\boldsymbol{D}^{(2)}=\boldsymbol{D}^{(1)}\cdot\boldsymbol{A}=\boldsymbol{A}^2$$
$$\cdots\cdots$$
$$\boldsymbol{D}^{(n-1)}=\boldsymbol{D}^{(n-2)}\cdot\boldsymbol{A}=\boldsymbol{A}^{n-1}$$

而 $\boldsymbol{D}^{(n-1)}=(\delta(i,j))$就是我们需要的多源多点最短路径长度构成的矩阵。

该算法的时间成本是多少呢？

乍一看，该算法的时间复杂性为 $\Theta(nn^3)=\Theta(n^4)$（n 次矩阵乘法），似乎仍然不理想。

但仔细一看，我们可以对矩阵乘法进行改善，因为每次乘上去的矩阵都是一样的！这样我们可以利用乘方窍门 $A^{2k}=A^kA^k$ 来改善算法的效率。因为计算 A^2, A^4, …, $A^{2\text{ceiling}(\log(n-1))}$一共只有 $O(\log n)$次乘方运算，所以时间复杂性为 $\Theta(\log nn^3)=\Theta(n^3\log n)$。总算有了改进！

检测负循环也很简单：只需要检查最后的矩阵 $D^{(n-1)}$的对角线上是否有负数，其时间成本为 $O(n)$。因此，整个算法的总时间复杂性仍然是 $\Theta(n^3\log n)$。

12.6.4 Floyd-Warshall 算法

我们费了很大力气，总算搞出来一个 $\Theta(n^3\log n)$效率的多源多点最短路径问题的解法。但整个过程似乎过于复杂：进行操作符映射，使用矩阵乘法，又利用乘方运算的窍门！

对前面的动态规划算法进行分析发现，既然可以按照边的条数不断构建包含边数越来越多的最短路径，当然也可以按照所使用结点数不断增加的方式由底至上构建最短路径！而这将获得一个效率更高的算法：弗洛伊德-华沙算法（Floyd-Warshall 算法）。

Floyd-Warshall 算法由美国计算机科学家罗伯特·弗洛伊德（Robert Floyd）和史蒂文·华沙（Stephen Warshall）于 1959 年发明。该算法的关键就是将前面的动态规划算法的分解标准从**边的条数**改为**结点集合**。

首先定义：$c_{ij}^{(k)}$= 从 i 到 j 其中间结点只属于集合{1, 2, …, k}的最短路径的长度，根据定义，有：

$$\delta(i,j) = c_{ij}^{(n)} \text{且 } c_{ij}^{(0)} = a_{ij}$$

与使用边的条数类似，中间结点只属于集合{1, 2, …, k}的最短路径（见图 12-33）可以从中间结点只属于{1, 2, …, k-1}的最短路径递归获得：

$$c_{ij}^{(k)} = \min_k\{c_{ij}^{(k-1)}, c_{ik}^{(k-1)} + c_{kj}^{(k-1)}\}$$

这个表达式称为 Floyd-Warshall 递归表达式。

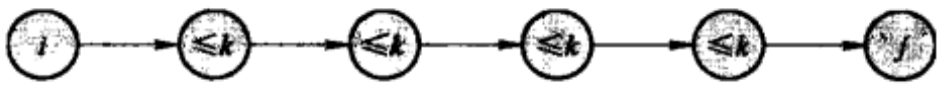


图 12-33 从 i 到 j 的最短路径，路径上的结点皆属于{1, 2, …, k}

该表达式的正确性可以从图 12-34 看出来。

从 i 到 j 只经过集合{1, 2, …, k}里结点的最短路径要么是从 i 到 j 只经过集合{1, 2, …, k-1}里结点的最短路径，要么是从 i 到 k 的只经过集合{1, 2, …, k-1}里结点的最短路径加上从 k 到 j 的只经过集合{1, 2, …, k-1}里结点的最短路径。

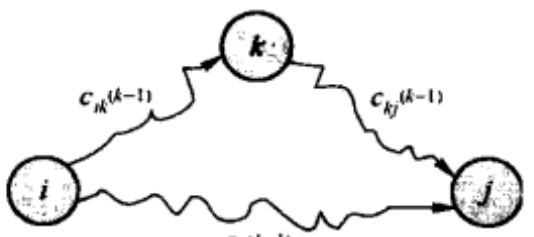


图 12-34 $c_{ij}^{(k)} = \min_k\{c_{ij}^{(k-1)}, c_{ik}^{(k-1)} + c_{kj}^{(k-1)}\}$

有了 Floyd-Warshall 递归表达式，我们可以很容易地得出计算所有结点对的最短路径算法：

```
Floyd-Warshall-SHORTEST-PATHS(G)
1.  for (k=1; k<=n; k++)              //最短路径上能够使用的结点从 1 递增到 n
2.      for (i=1; i<=n; i++)          //考察每个结点至所有其他结点的最短路径
3.          for (j=1; j<=n; j++)      //对于每个 i，计算 i 到 j 的最短路径
4.              if (cij>cik+ckj)
5.                  cij=cik+ckj;      //动态规划的递归表达式
```

注意，我们在算法里略去了上标，因为多几次降距操作不会有什么副作用。

显然，这个算法的时间复杂性为 $\Theta(n^3)$，比起前面的动态规划算法的 $\Theta(n^3\log n)$有所改善，而且该算法理解起来容易，实现简单。在实际中，该算法的效率也相当不错！如果将这个时间复杂性摊销到每条最短路径上，则寻找单条最短路径的时间成本为 $\Theta(n)$，线性！

12.6.5 Johnson 算法

Floyd-Warshall 算法适用于权重可正可负的一般情况。在前面的分析中我们提到过，如

果一个图的所有边权重都为非负，则我们可以运行 V 次 Dijkstra 的算法来获得多源多点最短路径，其时间成本为 $O(VE+V^2\log V)$。而这个时间复杂性要优于 Floyd-Warshall 算法的 $O(V^3)$，因为只有在稠密图的情况下，$O(VE)$才与 $O(V^3)$等级；而其他情况下，$O(VE)$要低于 $O(V^3)$。

问题是 Dijkstra 算法只适用于非负权重图，对于负权重图不适用。但是，能否将负权重进行调整，将其转化为正的权重，然后使用 Dijkstra 算法呢？

也许读者会认为我们不应该问这个问题。我们前面不是说过，“等效变换”的方法不能解决带负权重值的最短路径问题吗？现在为什么又来折腾人呢？

其实，不是等效变换的思想有问题，而是前面设计的“等效变换”并不是真正的等效变换。因为将图 G 中所有边的权重值减去 x-1（这里 x 是所有边中权重值最小的边，x<0）对图里不同路径的长度改变程度是不一样的。例如，对于图 12-35 的带负权重的图来说，如果按照我们前面讨论过的“等效变换”将每条边的权重值减去-4，将获得图 12-36 所示的图。

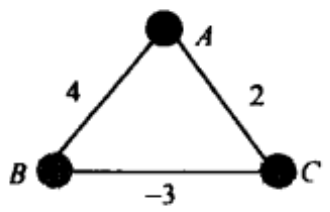


图 12-35 带负权重的图

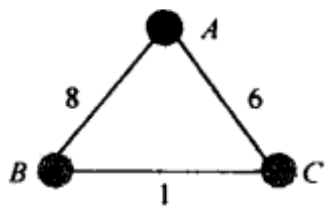


图 12-36 “等效变换”：权重增加 4

在图 12-36 中，结点 A 到结点 C 的直接路径增加的额度为 4，而 A 经过 B 再到 C 的路径增加的额度为 8。因此，此种“等效变换”无疑将导致错误的结果！

看来，问题出在等效变换的具体实现上，而不是等效变换的概念上。因此，如果我们能够发明一种真正的“等效变换”，则 Dijkstra 算法就可以派上用场。当然了，这个真正的等效变换本身不能太费周折和时间，不然成本都转嫁到转换上就没有什么意思了。

这种使用等效变换后再应用 Dijkstra 算法的多源多点最短路径问题算法就是约翰逊（Johnson）算法。该算法由美国计算机科学家唐纳德·约翰逊（Donald B. Johnson）于 1977 年发表于《Journal of ACM》期刊上（论文题目：Efficient Algorithms for Shortest Paths in Sparse Networks）。该算法的关键是等效变换。

12.6.6 Johnson 等效变换

定理 给定任意一个函数 h: $V\rightarrow R$，对图 G 里面的每条边$(u, v)\in E$ 分配新的权重如下：

$$w_h(u, v) = w(u, v) + h(u) - h(v)$$

则对于任意一对结点，它们之间的所有路径都将增加同样的额度。

证明 设 $p= v_1\rightarrow v_2\rightarrow\cdots\rightarrow v_k$ 为图 G 里面的一条路径，有：

$$w_h(p)=\sum_{i=1}^{k-1} w_h(v_i,v_{i+1})$$

$$= \sum_{i=1}^{k-1} (w(v_i,v_{i+1})+h(v_i)-h(v_{i+1}))$$

$$= \sum_{i=1}^{k-1} w(v_i, v_{i+1}) + h(v_1) - h(v_k)$$

$$= w(p) + h(v_1) - h(v_k)$$

由此可见，只要给定起始结点和终结结点，则该对结点之间的所有路径所增加的长度均一样，为 $h(v_1)-h(v_k)$。□

从上述证明我们又可以推论出等式：

$$\delta_h(u, v) = \delta(u, v) + h(u) - h(v)$$

这样，在进行了等效变换使得所有边的权重都为正后，就可以使用 Dijkstra 算法了。而这就是 Johnson 算法。

Johnson 算法（Johnson-SHORTEST-PATHS(G)）描述如下：

1）寻找函数 h: $V \rightarrow \boldsymbol{R}$ 使得$\forall(u, v)\in E$，$w_h(u, v) \geqslant 0$（注意：$w_h(u, v) \geqslant 0$ 当且仅当 $h(v)-h(u) \leqslant w(u, v)$）。

2）针对新图中的每个结点运行 Dijkstra 算法（使用权重函数 w_h）来计算 $\delta_h(u, v)$。

3）针对每对结点$(u, v)\in V\times V$，按如下公式计算原图中 u 到 v 的最短路径长度：

$$\delta(u, v) = \delta_h(u, v) - h(u) + h(v)$$

该算法的正确性已经无需多言（本章的一大堆讨论应该已经说明了这点）。我们关注的问题是该算法的效率如何。因为使用了 Dijkstra 算法，我们当然希望其时间复杂性低于 Floyd-Warshall 或其他动态规划算法。在算法的三个步骤中，第 2、3 步的时间复杂性非常清楚：第 2 步运行 V 遍 Dijkstra 算法的时间复杂性为 $O(VE+V^2\lg V)$。第 3 步的时间复杂性显而易见是 $O(V^2)$。因此，剩下的问题是第 1 步需要多少时间呢？

如果第 1 步的时间复杂性小于第 2 步，则整个算法的时间复杂性将为 $O(VE+V^2\lg V)$。该复杂性好于 Floyd-Warshall 算法复杂性，因为其在最坏情况下（稠密图）才是立方级的！

那么第 1 步的时间复杂性小于第 2 步的时间复杂性吗？

根据前面的分析，第 1 步的目的是将所有的负权重全部转换为正权重。换句话说，就是寻找一个函数 h，使其满足条件：

$$\forall u, v\in E,\ h(v) - h(u) \leqslant w(u, v)$$

细心的读者可能已经看出，这是一个线性规划问题（将每个 $h(v)$ 用一个变量替换即可看出）！如果读者更加仔细看，还会发现这是一种特殊的线性规划问题：每个不等式都是两个变量之差小于等于某个数值。这种两个变量之差小于等于某个数值的线性规划问题也被称为**差限**（difference constraint）问题。由于是线性规划问题，因此解决这个问题当然可以借助于各种线性规划的工具。但由于其特殊性，即每个不等式都是一个差限，因此我们可以使用最短路径问题来解决这个线性规划问题！这听上去挺奇怪的：解决最短路径问题导致一个线性规划问题，而解决该线性规划问题又导致另一个最短路径问题！看来，否定之否定无处不在。

用最短路径来解决差限问题使得本章的内容完全自含，而不用去查找线性规划的知识。

12.6.7 差限问题解决

如果要用最短路径解决方案来解决差限问题，首先需要将差限问题转换为最短路径问题。而转换的方法也直截了当：差限不等式的两个变量作为两个结点，差限值本身作为两个结点之间的边的权重，而边的方向从减数指向被减数。例如，给定如下差限不等式组：

$$x_1-x_2 \leqslant 3$$
$$x_2-x_3 \leqslant -2$$
$$x_1-x_3 \leqslant 2$$

我们可以构造差限图如图 12-37 所示。

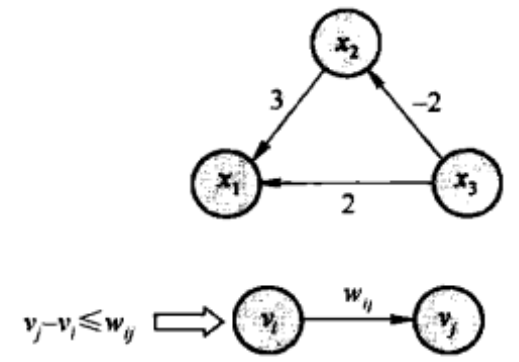


图 12-37 由差限不等式构造的差限图

由于差限图由差限不等式组转换而来，因此差限图的属性应该反映出差限不等式的属性。而最重要的是两点：一是能否从差限图检测差限不等式组是否有解；二是在有解的时候获得这个解。

幸运的是（也许与幸运没有关系），利用差限图恰恰可以解决这两个问题！

1. 差限不可满足定理

定理 如果由差限不等式组构造的差限图包括负循环，则整个差限不等式组是不可满足的。

证明 假定差限图里包括一个负循环：$v_1 \to v_2 \to \cdots \to v_k \to v_1$。根据差限图的构造方法，有：

$$x_2-x_1 \leqslant w_{12}$$
$$x_3-x_2 \leqslant w_{23}$$
$$\cdots\cdots$$
$$x_k-x_{k-1} \leqslant w_{k-1,k}$$
$$x_1-x_k \leqslant w_{k1}$$

将所有不等式相加，得：$0 \leqslant \sum w$，因为该循环为负循环，所以，$\sum w < 0$。这与由不等式相加获得的结果相矛盾。

因此，没有任何 x_i 赋值能够满足该差限不等式组。 □

2. 差限可满足定理

定理 如果由差限不等式组构造的差限图没有负循环，则整个差限不等式组是可满足的。

证明 如何证明一个差限问题是可满足的呢？当然最简单的办法是找出一个解！

我们在差限图上增加一个新结点 s，在 s 和所有结点 $v_i \in V$ 之间增加一条权重为 0 的边，这样我们将获得一张类似图 12-38 的图。因为没有增加任何负权重的边，所以这种改造不会给原来的图增加任何负循环。

定理 给每个变量赋值 $x_i = \delta(s, v_i)$ 将满足所有的差限不等式。

证明 考虑任意一个差限不等式 $x_j - x_i \leqslant w_{ij}$，以及从 s 到 v_j 和 s 到 v_i 的两条最短路径（见图 12-38）。

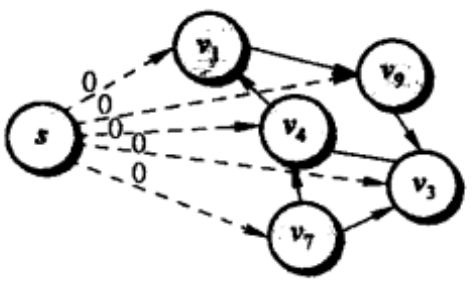


图 12-38 在差限图上增加一个结点 s 不会导致负循环

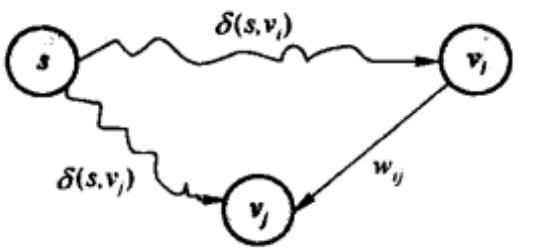


图 12-39 从 s 到结点 v_i 和 v_j 的两条最短路径

根据三角不等式，有：

$$\delta(s,v_j) \leqslant \delta(s, v_i) + w_{ij}$$

将 $\delta(s, v_i)$移到不等式左边有：

$$\delta(s,v_j) - \delta(s, v_i) \leqslant w_{ij}$$

而 $x_i = \delta(s, v_i)$ 和 $x_j = \delta(s, v_j)$，因此，$x_j - x_i \leqslant w_{ij}$，即差限不等式 $x_j - x_i \leqslant w_{ij}$ 获得满足。 □

上述证明同时给出了求解差限不等式（求解任意结点 v 的 $h(v)$值）的算法：

H(v)-COMPUTATION

1）构造差限图。

2）在差限图上增加结点 s，在 s 和所有结点 $v_i \in V$ 之间增加一条权重为 0 的边。

3）在新差限图上以 s 为源点运行 Bellman-Ford 算法。

4）如果 Bellman-Ford 算法没有报告负循环，则进行赋值 $h(v_i) = x_i = \delta(v_0, v_i)$。

5）如果 Bellman-Ford 算法报告负循环存在，则该差限问题无解。

上述算法的第 1、2 两步需要构造 $n+1$ 个结点，$E+n$ 条边的差限图，因此时间复杂性为 $\Theta(E+n)$。第 3 步的时间复杂性为 $O((n+1)(E+n)) = O(n^2+nE)$。第 4 步的时间复杂性为 $\Theta(n)$。而整个解决差限问题的时间复杂性为：$O(n^2 + nE)$。这样，Johnson 算法的第 1 步终于得到解决，且时间复杂性低于 Johnson 算法的第 2 个步骤。因此，整个 Johnson 的算法仍然是 $O(VE + V^2 \log V)$！摊销到每条最短路径上，则寻找单条最短路径的成本最多不超过线性！

最后需要注意的是，如果在算法 H(v)-COMPUTATION 中运行的 Bellman-Ford 算法报告负环路的存在，则说明无法进行等效变换，即该图不能使用 Dijkstra 算法！

12.7 天意难违

有了本章所介绍的多种寻找最短路径的算法，令汉尼拔头痛（也不一定头痛）的问题自然也就迎刃而解了。此时读者也应该算出来了，迦太基军队行军的路线是：

迦太基→北非→西班牙→高卢→罗马

出人意料的是，迦太基最后并没有击败罗马。卓越的军事天才和神速的军事行动也抵不过天意！虽然占领意大利达 16 年之久，但迦太基从未攻克罗马城。公元前 202 年，在回师应付罗马对迦太基本土的攻击时，绕罗马城南下的迦太基主力在北非的扎马（Zama）被罗马军队彻底击溃（见图 12-40），从而开始了结束汉尼拔辉煌军事生涯和迦太基辉煌历史的漫漫熊途……

图 12-40 公元前 202 年，迦太基主力在扎马被罗马军队击溃，从此由盛转衰

也许，寻找到捷径并不能保证最后的成功？抑或，捷径带来的是痛苦的提前降临？

思考题

1. 从表面上看，单源单点最短路径问题似乎应该比单源多点最短路径问题简单，但事实却相反。为什么看上去简单的单源单点最短路径问题反而比看上去复杂的单源多点最短路径问题更加复杂呢？
2. 在 Dijkstra 的最短路径算法里使用了贪心策略，那么在我们求取一对结点之间的最短路径时是否也能使用贪心策略呢？为什么？
3. 在 Dijkstra 算法里，我们可否使用散列表来实现优先队列 Q，为什么？
4. 在所有结点对最短路径的动态规划解法中，当转换为矩阵运算后，我们是否可以用 Strassen 方法来降低矩阵乘法的复杂性呢？
5. Dijkstra 算法的时间复杂性和 Prim 最小生成树算法的时间复杂性一样，请比较这两个算法的异同，并说明为什么它们有着同样的时间复杂性。
6. 在论述 Dijkstra 算法的局限时说过，权重为负值将导致三角不等式的失败。但是在使用 Bellman-Ford 算法来解决差限问题的时候，我们又利用了三角不等式。你觉得这有什么问题吗？为什么？
7. 给定一个无环图，请设计一个算法求取该图中最长的一条路径，并分析算法的复杂性。
8. 给定无环图，请针对每一对结点，判断其间是否存在一条通道，并分析算法的复杂性。

9. 对于一个没有权重的图，即如果对于所有$(u, v) \in E$ 都有 $w(u,v)=1$，你能否对 Dijkstra 的最短路径算法进行改善？
10. 如果图是有向的，即每条边有方向，则从 u 到 v 有边并不等于从 v 到 u 有边。如果该图没有环路，你能否设计一个更好的求取一对结点之间最短路径算法呢？
11. 请修改 BFS 遍历算法来解决单源单点最短路径问题。
12. 对于一个没有权重的图，请设计一个线性时间复杂性的单源单点最短路径算法。
13. 我们能否在单源单点最短路径问题上使用贪婪策略？为什么？
14. 为什么在单源多点最短路径问题被提出来前，人们在单源单点最短路径问题上没有想到 Dijkstra 算法或类似的方法？
15. 对于带有负权重的图，单源多点最短路径问题是否仍然具有贪婪选择属性？为什么？
16. 在讨论多源多点最短路径问题时，Johnson 算法采取的方法是对带负权重的图进行等效变换后再运行 Dijkstra 算法。我们能否将此种办法用到单源多点最短路径上，即对带有负权重的图先进行 Johnson 等效变换，然后使用 Dijkstra 算法计算单源多点最短路径？
17. 如果将求解单源多点和多源多点最短路径问题的时间成本摊销到单条最短路径上，我们发现单条最短路径的成本只有线性或低于线性。请问这是摊销吗？这种想法与本书第 8 章讨论的摊销分析是一回事情吗？说明你的理由。
18. 蜜月旅行问题：新婚燕尔的左怡和尤尔决定以远足的方式来度过他们的蜜月。他们手中有一张地图，记为无向图 $G=(V,E)$，地点就是图中的结点，小道就是边。其中有些地点是公交站点，以 S 表示，这里 $S \in V$。左怡和尤尔进行的远足必须满足下列条件：
    - 远足的开始结点和终结结点为不同的公交站点。
    - 所有的上坡在远足的开始，所有的下坡在远足的最后，即不能出现上坡下坡交替轮番出现的情景。

设 $w(e) \in R$ 为小道 $e \in E$ 的长度，$h(v) \in R$ 为结点 $v \in V$ 的海拔高度。为使问题简单，假定所有地点的海拔都不相同。

（a）请给出一个算法来计算最短可行的路径

（b）请给出一个算法来计算最长可行的路径。

PART FIVE

第五篇　难解与无解篇

PDG

CHAPTER 13

第 13 章　易解与难解

1861 年 11 月 7 日，在艳阳照耀下的加勒比海哈瓦那港，一艘飘扬米字旗的英国邮轮特伦特号（TRENT）正缓缓离开古巴，驶向英国。

一切似无异常之处。但平常里隐含着不平常，该船上载着两名特殊身份的客人：美国南方联邦政府代表詹姆斯·梅森（James Mason，见图 13-1）和约翰·斯利德尔（John Slidell，见图 13-2）。他们刚刚费尽千辛万苦从北方海军的封锁中逃离查尔韦斯顿（Charleston）城来到古巴。他们是奉南方联邦政府的使命前往欧洲寻求英国和法国的外交承认。此时，南方军队在战场上正捷报连连。

图 13-1　南方特使，美国参议院外事委员会前主席詹姆斯·梅森

图 13-2　南方特使，新奥尔良名律师约翰·斯利德尔

第二天，当特伦特号行驶在大西洋上时，由查尔斯·威尔克斯（Charles Wilkes）上校率领的美国北方联邦政府的圣亚辛托号（San Jacinto）战舰拦截了特伦特号邮轮，逮捕了这两名南方使节并将他们移送至北方港城波士顿接受审判。消息传到英国，伦敦为之哗然。议员和将军们群情激奋地要求政府采取断然行动，以教训这帮“不知天高地厚”的美国清教徒们。此时英国的掌门人是好战的帕莫斯通（Henry John Temple Palmerston）勋爵，他向美国政府发出了最后通牒：立即释放南方使节，立即向英国政府道歉并赔偿物质和精神损失；如若不从，将立即向美国开战。为表明其严正立场，英国派遣了一支 8 000 人的舰队进驻加拿大。

大西洋两岸的人都在注视着林肯……

令所有人大跌眼镜的是，这位“骨瘦如柴、面容丑陋、意志坚定”的高大男子却做出了令当时观察家们困惑不解的决定：命令美国驻英国大使亚当斯向英国政府道歉，表示圣亚辛托号是在没有政府指示下的擅自行动，并同时下令无条件释放了南方两位使节。

一场即将爆发的英美大战就这样被化解于无形之中。

也许林肯被英帝国主义分子的屠刀吓到了？也许他觉得攘外必先安内？也许……但真实的原因只有林肯自己知道：智慧和勇气！因为对于一个智者来说，向前也许并不需要勇气，更无需任何智慧，而需要智慧和勇气的恰恰是后退……

也许，在面临不可解难题的时候，我们也需要后退的智慧和勇气！

13.1 我们战无不胜吗

到目前为止，我们似乎无所不能，战无不胜：我们从数数开始，相继发明了分治、动态规划、贪婪选择、随机化、概率分析、摊销分析、竞争分析、排序与次序、搜索与散列、最短路径等，我们一直在“如何高效地解决问题”的道路上不断推进，不断接近“完美”。

一切似乎都在我们的掌控之中。

但我们真的无所不能吗？算法是无所不能的吗？

不幸的是，答案是否定的。事实上，我们目前所讨论过的问题基本上属于所谓的“易解的”（tractable）一类，即容易解答的问题。但世界上的问题并不都是易解的。事实上，世界上多数的问题不是易解的，而是我们所称之为“难解的”（intractable）问题。

13.2 易解与难解

那么什么是“易解的”问题呢？首先我们看看英语里“tractable”这个词的含义。在英语里，“tract”有很多意思，而其中一个意思是论文、短文或手册。“tractable”的意思就是可写在论文、短文或手册里面的意思，即可以表述，引申为可塑的，能够处理的，也就是易解的意思。“intractable”自然是不能记录下来的东西，引申为不可塑的、无法处理的，也就是难解的意思。

那么到底哪些问题是易解的，哪些是难解的呢？

一个问题被称为易解，首先必须是有解的，即存在一个解答；其次，这个解答能够被找出来。而这第二个属性非常重要。这个属性意味着寻找解答的算法的时间和空间效率必须是合理的，可以达到的。如果解答一个问题需要的时间或空间是无限的，自然我们就不能寻到答案。因为目前世界上尚不存在一台空间无限的计算机，而且也没有什么人能长生不老。

但这个定义似乎比较模糊。有更清楚的解释吗？

还记得本书前面章节提到过，一旦一个算法的时间效率是指数级，我们便说这个算法效

率太低。这是因为指数级函数的增长趋势是如此之快，当自变量 n 增大时，该函数很快就超越了人们和计算机的计算能力。因此对应指数级算法的问题就被认为是难解的问题！

从另一个角度看，解决“难解问题”的最好办法是在指数级个潜在的答案里面进行穷举搜索！这种算法即使用世界上最快的计算机也需要很多年，甚至几十年才能算出结果！

易解的问题就是多项式问题。这是本书到目前为止所讨论过的算法。从另一个角度看，我们只需要多项式时间就能够从一个指数级样本空间里面寻找到答案！

一个自然的推论就是，所有与指数级增长同步或比指数级增长更快的问题也是难解的问题。而所有与多项式级增长同步或比多项式级增长更慢的问题也是易解的问题。

这里特别提请读者注意的是，所谓“在多项式时间内可解”是相对于输入的规模而言的，即求得解所花费的成本可以表示为输入规模的一个多项式。如果输入规模本身就是指数级的，则无论算法如何构造，其最终结果必然是指数级的（因为检查一下输入就得指数级）。因此，独立于输入规模来谈多项式级或指数级是没有意义的。

13.3　决策问题和优化问题

也许你觉得本书前面讨论过的问题各种各样，五花八门，有排序与次序、有搜索与散列、矩阵乘法、寻找梦幻情人、最短路径、贪婪选择等。但如果仔细查看，却发现所有的问题都可以归纳为两种类型：决策问题和优化问题！

决策问题讨论的是“一个特定的表述”是否为真，而优化问题讨论的则是寻找一个最好或得分最高的解答（这个最好或得分最高是按照人们事先规定的标准进行判断的）。用哈姆雷特的话，决策问题讨论的是“to be or not to be”，而优化问题讨论的是“how to be”。用人生来做比喻，决策问题就是“选择活着还是选择死亡”，优化问题就是“怎样活出最好的人生”。（当然了，最好的人生就仁者见仁智者见智了。）

例如，对于一个带权重的连通图，寻找一棵最小生成树就是一个优化问题，即在所有可能的生成树里（所有潜在答案），要找那棵成本最低的生成树！（优化目标）。而同一个问题，也可用决策问题来表示，给定一个带权重的连通图和一个成本值 c，是否存在一棵其成本小于等于 c 的生成树。

这里需要注意的是，优化问题又可以分为两种类型：最小优化问题和最大优化问题。前者试图找出一种成本最低的解答，而后者试图找出一个收益最高的解答！例如，上面给出的最小生成树问题就是一个最小优化问题。而本书前面讨论过的背包问题：给定 n 个物品、物品 i 价值 v_i、重量 w_i 和一个总重量 w；要求找出一个物品的组合，使其在总重量不超过 w 的情况下，总价值最高。这是一个最大优化问题！

细心的读者也许已经注意到，优化问题和决策问题是相伴而行的，即有一个优化问题，就有一个对应的决策问题，而每一个决策问题也对应一个优化问题。例如，对于背包问题，其对应的决策问题是：给定 n 个物品、物品 i 的价值 v_i、重量 w_i、一个总重量 w 和实数 c；是否存在一个物品组合，其价值大于等于 c，而总重量小于等于 w？

再举一个例子，旅行售货员问题（见图 13-3）。这个问题本是一个优化问题：给定一个带权重的完全图，要求寻找一个权重最小的汉密尔顿回路。但它存在一个对应的决策问题：给定一个带权重的完全图和实数 c，是否存在一个权重不超过 c 的汉密尔顿回路？

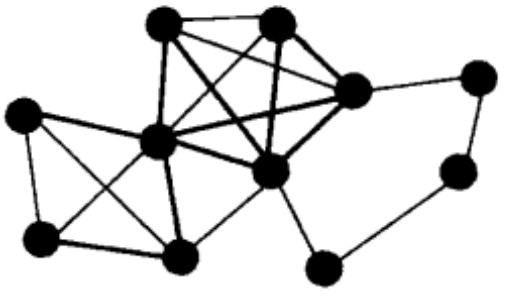


图 13-3 旅行售货员问题

从某种角度看，决策问题和优化问题是可以相互转换的。例如，“选择活着或选择死亡”看上去是一个决策问题，但完全可以转换为一个优化问题：“如何选择才能为祖国或人类做出最大的贡献呢？”对于某些人来说，“选择活着”会贡献大点，对另外一些人（如占用资源格外多的人）来说，“选择死亡”的贡献可能更大！而“怎样活着”的优化问题也可以很容易地转换为决策问题。（你看出来如何转换了吗？）

这个决策和优化问题之间的关系和转换对于算法研究非常重要。正是由于它们之间存在的对应及相互转换，我们下面可以只讨论决策问题，而不用关心优化问题！

13.4 决策问题

在算法中，我们通常碰到的是优化问题。但对于 NP 完全的研究来说，决策问题更容易讨论。由于优化与决策之间的相互转换关系，我们对于决策问题的讨论同样适用于优化问题。由于优化和决策是所有问题的终极表现形式，因此我们的讨论适用于宇宙间的所有问题！

简单地说，决策问题需要回答的就是“是”和“否”两个答案中的一个。但是每个决策问题都必须有一个输入，而我们就是在这个给定输入的环境下来解答问题。

例如，对于旅行销售员问题，输入由以下两个部分构成：

1）一个带权重的完全图。

2）一个实数。

而我们就是在此输入回答“是否存在一个权重不超过 c 的汉密尔顿回路”这个问题。

一个特定的输入也常常被称为相关问题的一个实例（instance），而每个实例都有一个大小（所需要的计算机内存）。例如，对于一个 n 个结点的图来说，这个实例的大小通常为 $n(n-1)/2$。

如果我们对一个决策问题的回答是“是”，则通常需要提供一个“证人”来证明我们的回答。例如，对于汉密尔顿回路问题，结点 v_1，v_2，…，v_n 的任意排列都是一个潜在的证人。但如果在此排列下，v_1 与 v_2 相邻，v_2 与 v_3 相邻…… v_n 与 v_1 相邻，则这个证人就是“真”证人。而对于旅行售货员问题，任何一个汉密尔顿回路都是一个潜在证人，而如果该证人的成本不超过 c，则这个潜在证人就是一个“真”证人。

13.5　P 类问题

有了上面的铺垫，现在我们可以来定义算法（或计算理论）中著名的 P 类问题了。

一个决策问题 D，如果其满足下列条件，则被认为是多项式时间可求解的：

1）存在一个算法 A，A 的输入是 D 的实例，A 总是正确地输出“是”和“否”的答案。

2）存在一个多项式函数 p，如果 D 的实例大小为 n，则 A 在不超过 $p(n)$个步骤里终结。

如果一个问题是多项式时间可求解的，则我们说这个问题属于 P 类问题！而所有满足上述条件的问题就构成了 P 类问题的集合!

对一个算法来说，如果其最坏情况下的时间复杂性是输入规模的一个多项式函数，则该算法被称为多项式限定的（polynomial bounded）。具有多项式限定算法的问题称为多项式限定问题。从这个角度看，P 类问题就是多项式限定的决策问题。例如，最小生成树问题就是一个 P 类问题。

通常，人们将 P 类问题等同于计算可行性问题，或者说，易解的问题。但这个看法并不现实。例如 $n^{1\,000}$ 问题是易解的问题吗？但即便如此，算法分析中一个基本观点还是多项式（或更低阶）函数是合理增长的、可控的，即易解问题；指数（或更高阶）函数是爆炸式上升的、不可接受的，即难解问题。

13.6　NP 类问题

与 P 类问题对应是所谓的 NP 问题。也许大家都听说过 NP，但 NP 代表什么意思呢？有人说 NP 代表“不是问题”（No Problem），有人说它代表非多项式可解（Not Polynomial），还有人说代表不可能（Not Possible）等。不过这些说法都不正确。因为如果上述一种说法正确的话，则 NP 就没有任何研究价值了，或者说需要研究的目的已经达到。

那么 NP 到底是什么呢？它是英文 non-deterministically polynomial-time solvable 的缩写，即**非确定性多项式时间可解**的意思。而它与 P 类的多项式时间可解是对应的。也许读者觉得奇怪，多项式时间可解怎么与非确定性多项式时间可解对应呢？多项式时间可解似乎应该与非多项式时间可解对应才对呀！读者如果这么认为完全情有可原。但如果我们指出，“多项式时间可解”实际上指的是“确定性多项式时间可解”，你还会这么认为吗？

那什么是非确定性多项式时间可解呢？

一个决策问题 D，如果满足下列条件，我们就称其为**非确定性多项式时间可解**：

1）存在一个算法 A，A 的输入是 D 的潜在证人，A 总是正确辨认该证人的真假。

2）存在一个多项式函数 p，如果潜在证人对应的 D 的实例大小为 n，则 A 在不超过 $p(n)$个步骤里终结。

如果一个问题是非确定性多项式时间可求解的，则我们说这个问题属于 NP 类问题！

乍一看，NP 的定义似乎与 P 的定义一样。我在上算法课时经常碰到学生说这两个定义是一回事情。如果你觉得是一回事情，那再仔细看看，真的是一回事情吗？

当然不是。它们之间有重大区别！这个区别就在第 1 条上。P 定义的第 1 条是能够给出答案，而 NP 定义的第 1 条是能够指出一个答案是否正确！众所周知，给出答案和判断答案是难度很不相同的两回事情。这就是为什么一般人更愿意做选择题，而不愿意做解答题！

如果到这里读者还没有很清楚，那我们就从另外一个角度看：NP 代表的是可以被非确定性图灵机在多项式时间解决的所有问题！其等价的定义是，NP 代表所有其解答可以被一个确定性图灵机在多项式时间内验证的问题。更清楚了吗？

没有？不知道确定性图灵机和非确定性图灵机的区别？

13.7 （确定性）图灵机

图灵机，自然是拜图灵（英国数学家阿兰·图灵）所赐。他在人们还没有搞清楚什么是计算机的时候就提出了一个虚无缥缈的图灵机概念。该虚拟机器（注意不要与今天的虚拟机搞混了）虽然简单，但是功能极为强大。它怎样强大呢？我们来看图灵机的定义。

确定性图灵机，或者简单地说，图灵机，是一个状态机。我们可以将图灵机抽象成为一个带有很长磁带的机器，机器的磁头在磁带上左右移动。磁头下面的字母为图灵机的输入，如图 13-4 所示。

图 13-4 图灵机示意图

该状态机的状态转换函数具有如下能力：对于给定状态和输入符号，确定三件事情，即：

- 输出符号。
- 磁头移动方向（左或右）。
- 下一个状态。

例如在状态 3 的时候，如果输入符号为 X，则图灵机可能输出符号 Y，将磁头往左移动一个位置，并进入状态 4。换句话说，确定性图灵机在给定状态和输入下其行为是唯一确定的。

也许你觉得图灵机功能很有限，无非就是按照输入字符和所处状态进行输出和状态转换。如果你这样想，那就错了。对字符串进行处理是对一般问题的抽象化！因为所有的问题都可以转换为对字符串的处理。任何一个问题的输入都可以表述为一个字符串，任何一个算

法也可用表述为一个字符串，甚至任何问题本身都可以表示为字符串！因此，图灵机的这三个简单的输出动作能够解决世界上很多的问题！（但不是所有问题！）

13.8　非确定性图灵机

非确定性图灵机与确定性图灵机的区别是，在给定状态和输入时，其行为将不是唯一确定的。也就是说，对应同一个状态和输入，非确定性图灵机可以有多种行为来选择。例如，在状态 3 的时候，如果输入符号为 X，则一个非确定性图灵机可能输出符号 Y，将磁头往左移动一个位置，并进入状态 4；它也有可能输出符号 X，将磁头往右移动一位，并留在状态 3 里；它也完全有可能输出符号 Z，磁头往左移动一位，进入状态 5 等。

从另一个角度来说，确定性图灵机表述的是因果关系，而非确定性图灵机表述的不是因果关系！对于大部分人来说，确定性图灵机比较容易理解，而非确定性图灵机显得很抽象。因为人们习惯了因果关系，对于没有因果关系的事情难以理解。这也就是为什么发生任何重大变故后，人们总是要找原因，也许就是这种根深蒂固的因果思维在作怪！

如果你还是不能理解非确定性图灵机，那我们再打个比喻。当你的左脸突然被人猛击一拳的时候，你会做何反应呢？你会回击？你会逃跑？你会破口大骂？你会打电话叫警察？或者将右脸伸过去让他打？事实上，你会做何反应并不是我们要关心的问题，而且没有人能肯定（你自己也不能肯定）你会做何反应，我们在乎的是上述每种可能都有发生的概率。而这种在输入相同的情况下，结果有可能不确定的问题就是非确定性图灵机！

当然，非确定性图灵机要比你高明，它的行为虽然我们不能确定，但它总是会选择最好的反应！（而你却不一定能选择最好的反应！）换一个角度来说，非确定性图灵机是世界上最幸运的猜谜手，它总能在无数可能中猜中最好的选择，即选择最好的状态转换以达到其最终目的（进入接受状态），如果这样一种选择存在的话！

另一个角度看，这个问题就是非确定性图灵机能够同时进入所有可能状态，也就是说，能够像孙悟空那样，变出无数个自己，从而同时在多个地方出现！也许你听说过量子理论中的一个假说——所有可能发生的都已经发生，只不过它们发生在不同的宇宙而已！即你既是中国人，也是美国人（这里排除双重国籍），这两种可能都是事实！只不过，在这个宇宙里你只是中国人。（而在另一个宇宙里，你是美国人！）

现在，你也许看出来图灵机的强大了！而且我们也可以看清确定性图灵机和非确定性图灵机的区别了：确定性图灵机只能跟踪一条计算路径，而非确定性图灵机则拥有一个计算树，即同时跟踪多个计算路径。如果其中一个路径引向终止状态，则我们说非确定性图灵机接受了给定的输入。

13.9　非确定性算法

本书前面讲解的算法都是确定性算法，它们都运行在确定性图灵机上，因此算法的每个

步骤都是确定的。而在非确定性图灵机上运行的算法则是非确定性算法，它们的步骤并不是唯一确定的。从对非确定性图灵机的描述可知非确定性算法由三个阶段组成：

- **阶段 1** 非确定性“猜想”阶段。任意猜想一个答案 s，并将字符串 s 写在内存的某个地方（每次机器运行的时候，这个 s 都有可能不同）。
- **阶段 2** 确定性“验证”阶段。一个正常的算法以该决策问题和答案 s 作为输入，对 s 进行验证。验证的结果既可能是真也可能是假。
- **阶段 3** 输出阶段。如果验证阶段输出真，则输出“是”，否则输出“否”。

也许你觉得这个算法不能够在实际中实现，给出这个算法纯属浪费时间。但这种想法是错误的。虽然该算法在实际中没有任何意义，但它有理论上的意义：它可以帮助我们对问题进行分类！下面我们以素性测试为例对非确定性算法加以说明。

我们知道天真的素性测试算法（即对每个可能的因子进行排除测试）的时间复杂性为 $2^{\log n}$，这里 n 是欲测之数，即如果以输入的字位数来看，该算法的时间复杂性是指数级！一个不怎么样的算法。但是，测试任意给定自然数 d 是否真是 n 的一个因子则非常简单：

```
isFactor (n, d)
if(n|d&&d≠1 &&d≠n)
    return true;
else
    return false;
```

显然，如果 n 是合数，则上述函数必定在某些 d 上面返回真值；如果 n 为素数，则上述函数永远返回假值。我们的天真算法之所以效率低下，是因为一个时候只能测试一个潜在因子 d，即我们的算法是运行在确定性图灵机上。但如果我们有一个非确定性图灵机呢，结果会怎样呢？

如果我们有一个非确定性图灵机，素性测试将变得非常简单！因为该算法可以在 $O(\log n)$ 步骤内分支到 n 的所有潜在因子上，然后针对每个潜在因子调用上面的测试函数 isFactor(n, d)。如果任意一个分支返回真值，则所测之数为合数；否则，为素数。

由此可见，将确定性图灵机升级为非确定性图灵机可将算法的效率从 $2^{\log n}$ 提高到 $O(\log n)$，这是一个巨大的改善！由此可见非确定性图灵机的能力！

至此，我们可以清楚地看出确定性算法和非确定性算法的区别：确定性算法基于给定的信息做出判断，而非确定性算法则通过一系列“正确的猜测”获得答案！而由于现代的计算机无法做到“一系列正确的猜测”，它们只能运行确定性算法！

13.10 回到 NP 类问题

现在回到 NP 类问题上。如果一个问题的解答可以在多项式时间内被证实，则这个问题就是属于 NP 类（这里假设我们可以正确猜出一个真证人）。而对于到目前为止所讨论的所有决策问题，我们都可以提出一个候选的解决方案供检查。

使用非确定性算法，我们可以给 NP 下一个精确的定义：如果一个问题存在非确定性算法，可以在多项式时间内解决，则该问题就是一个 NP 问题，或者说，该问题属于 NP（类）。

另一个更为松散的定义是：如果一个问题的潜在解答可以在多项式时间内被证实或证伪，则该问题属于 NP。NP 类包含的问题数量巨大，可以说是无穷的。例如，完全子图问题、图的着色问题、汉密尔顿回路问题、子集和（subset sum）问题、可满足性问题，以及旅行售货员问题等都是 NP 问题。

完全子图问题（见图 13-5）：

- 优化版。给定图 $G=(V,E)$，找出其最大完全子图。
- 决策版。给定图 $G=(V,E)$和整数 k，是否存在一个尺寸为 k 的完全子图?

显然，直接的办法是将图里所有大小为 k 的子图全部考虑一遍，看是否有完全子图。这种天真办法的时间复杂性与 n^k 成比例，这里 $n=|V|$，但 k 却不是一个常数! 因此这个算法效率不高。而且到目前为止，也没有人发现效率更高的算法。因此，很多人怀疑这个问题不存在有效的算法（即多项式时间算法），但是这个怀疑并没有被证明。

图的着色问题就是对一个给定图的结点进行着色，如图 13-6 所示。条件是相邻结点的颜色必须不同（即一条边的两个结点必须颜色不同），而着色一个图需要的最少颜色种类就是该图的色度基数（chromatic number）。图的着色问题也存在决策和优化两个版本，分别如下：

- 优化版。给定图 $G=(V,E)$，找出其色度基数。
- 决策版。给定图 $G=(V,E)$和整数 k，该图是否可以用 k 中颜色进行着色?

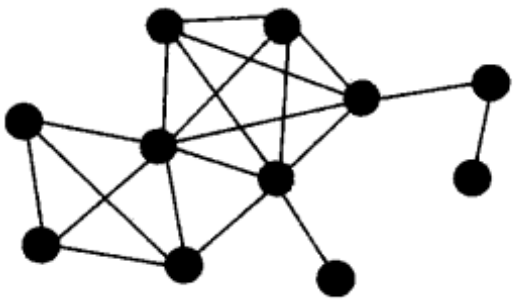


图 13-5　完全子图问题

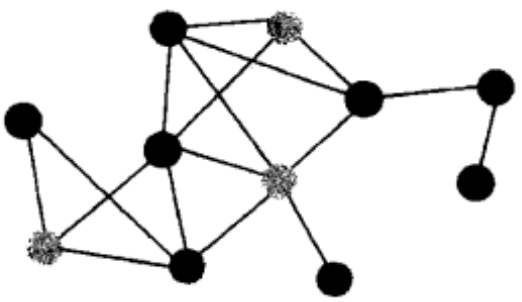


图 13-6　图的着色问题

子集和问题：给定一个正整数集合 S 和整数 k，是否存在 S 的一个子集 R，且 R 的所有元素之和等于 k? 例如，如果给定 S={1,16,64,256,1 040,1 041,1 093,1 284,1 344} 和 k=3 754，则该问题存在一个解决方案 R={1,16,64,256,1 040,1 093,1 284}。

可满足性问题：给定布尔表达式，它是否可以满足？即是否存在一组变量的 0 和 1 赋值，使得整个布尔表达式的求值为 1?

13.11　P 和 NP

对于 NP 来说，一个常见的误解是人们认为 NP 问题不存在多项式时间解，而这是完全错误的。正如我们前面说过，NP ≠ Non Polynomial！而是 NP = Non-deterministic Polynomial。

NP 不但不意味着不存在多项式时间解，而且事实上人们为很多 NP 问题找到了多项式时间解。例如，素性判断就是一个找到了多项式时间解的 NP 问题。

这是否意味着 P=NP 呢？或者说，P 类集合是否与 NP 类问题集合完全重合呢？这个问题是 21 世纪数学界（和计算机科学理论界）面临的一个重大问题。

显然，所有的 P 类问题都是 NP 问题，因为确定性图灵机能够解决的问题当然能够被非确定性图灵机解决。所以，我们的问题变成是否所有 NP 问题都是 P 问题。凭着我们的直觉，NP 应该不属于 P。原因很简单：非确定性图灵机比确定性图灵机强大得多，很难相信一个强大得多的机器所能够解决的问题都可以被一个功能更弱的机器解决！

但直觉归直觉，在算法领域或数学领域，我们不能凭直觉去说服其他的人。因此，我们必须拿出证据来说明 NP 不属于 P。要证明这一点，我们只需要证明某个 NP 问题不属于 P 即可，而这似乎是个很简单。但遗憾的是，到目前为止尚未有人证明 NP 不属于 P。当然，也没有人证明 NP 属于 P。也就说，P 与 NP 是否等价是一个既没有证实也没有证伪的问题！

除了凭直觉感觉 NP 比 P 大之外，经验数据也代表着同样的观点。因为成千上万的计算机科学家和数学家为某些 NP 里的问题设计多项式时间解的时候都遭遇了失败，所以，对于这些问题，科学家给它们起了一个与普通 NP 问题有所区别的名称：NP 难（NP-hard）！

13.12 搜索问题、决策问题和优化问题

前面我们说过，世界上所有问题都可以分为两种：决策和优化。而后我们又讨论了决策问题和优化问题其实是可以互相转换的。因此，所有的问题实际上都可以表述为决策问题。这一点并不难理解，因为，从一定的角度看，对于任何一个问题求解都是做出某个决定。

但是，如果从另一个角度看，所有问题也可以表述为搜索问题。解答一个问题不就是搜索这个问题的解吗？当然，搜索的空间就是解的空间，而搜索就是在解的空间找出需要的一个。对于大部分问题来说，其解空间的规模为输入规模的指数函数甚至更高。显然，在如此巨大的解空间里面实施穷举将是费时、费力、低效率的。因此，寻找更加有效的搜索手段就是算法不断推进的源动力。（有意思吧，提高搜索的效率需要使用到搜索！）而本书前面介绍的分治思想、动态规划思想、贪婪选择思想和随机化思想都采用了各种巧妙手段成功地避开了“指数魔咒”——把搜索空间的规模缩小到多项式级以内。

恐怕读者到这里已经想到，搜索问题、决策问题和最优化问题之间是等价的（见图 13-7）。这个想法是正确的。例如，对于旅行售货员问题（在一个权重非负的带权图中，找一个经过所有顶点的回路，使回路总长度小于给定的阈值 B），如果最优化问题的解（最短回路，其总长为 l^*）已知，那么解决搜索问题不费吹灰之力：如果阈值 B 小于 l^*，则报告无解；否则返回已知的最短回路即可。另一方面，如果搜索问题有多项式时间的解，那么最优化问题也有多项式时间的解。方法如下：首先，将 B 设为所有边的权值之和 l，若此时搜索问题报告无解，则最优化问题必无解；否则，必有解。然后，将 B 设为 floor(l/2)，若此时搜索问题报

告有解，则最短回路长度必属于[0,floor(*l*/2)]；否则最短回路长度必属于[floor(*l*/2)+1,*l*]。接着再将 *B* 设为相应减半的区间上下限的中间点上，依此类推。采用这种二分搜索算法，我们最终可以获得一个最优解，而需要搜索的次数最多为 log*l*。如果 *l* 独立于图的结点数，则搜索的成本同样在多项式级以内。

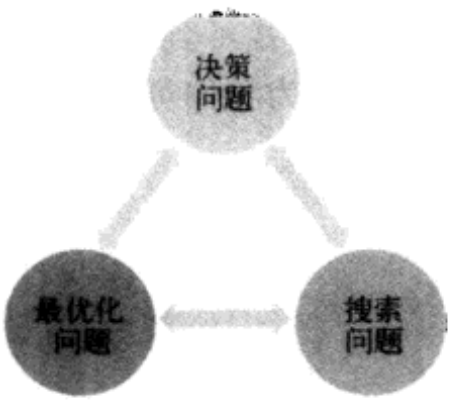


图 13-7　决策问题、搜索问题和最优化问题可以相互转换

13.13　有没有解和是否可决定

需要注意的是，非确定性算法回答“否”并不意味着实例 *I* 无解，因为算法可能猜测了一个不正确的解 *S* 。我们说非确定性算法能“解”一个问题，当且仅当对每个输出应为“是”的实例，非确定算法能在某些运行中输出“是”，即它能猜对解至少一次并验证其正确性。当然，对输出应为“否”的实例，非确定算法必须输出“否”。

注意　一个问题有没有解和一个问题是否是可决定的是不同的。例如，九字迷宫问题，其可决定性和其是否有解就是不同的。

所谓的九字迷宫问题指的是在一个 3×3 的网格里按先行后列的顺序将 1、2、…、9 填入方格里，如图 13-8a 所示。

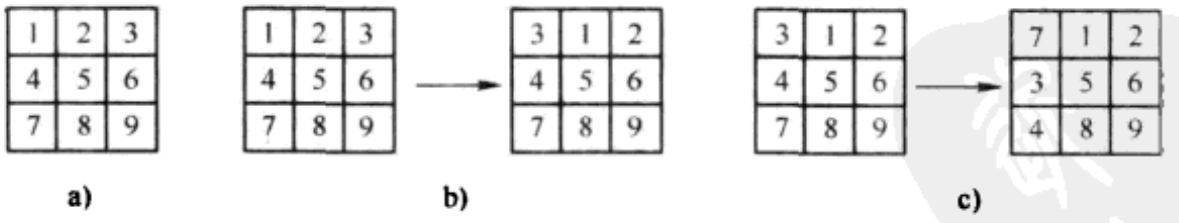


图 13-8　九字迷宫问题

读者可以对网格进行转换，而转换的方式是旋转。旋转既可以绕行进行，也可以绕列进行。例如，对图 13-8a 的第 1 行进行右旋可得到一个新的网格，如 13-8b 所示。在 13-8 b 的基础上如果再对第 1 列进行下旋，则获得图 13-8c 所示的图。

能否找到一个旋转序列，使得网格到达原来状态的转序（transposes），即如图 13-9 所示的状态？

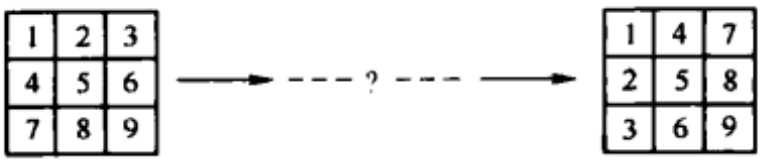


图 13-9 九字迷宫问题的转序

这个问题是一个无解的问题，即我们找不到一个旋转序列能够进行图 13-9 所示的转换。但这个问题是一个可决定的问题。因为我们可以很容易地证明这个问题不可能解决。你能证明吗？

思考题

1. 对于你来说，“难解”意味着什么？
2. 证明：九字迷宫问题无解。
3. 随机化算法是确定性还是非确定性算法？
4. 非确定性图灵机能否实现？说明你的理由？
5. 有人说，任何一台计算机，不管多旧，也不管多新颖，都与一台确定性图灵机等价。你同意这个说法吗？说出理由。
6. 本章在讨论搜索问题和最优化问题之间的等价关系时，提到了如果搜索问题有多项式时间解，则最优化问题也有多项式时间解。但这个结论的前提是 l 独立于图的结点数。你觉得这个前提站得住脚吗？如果站不住脚，你能得出何种结论？
7. 如果给你一个非确定性图灵机，你是否能够解决世界上的所有问题呢？给出理由。
8. 在一座人迹罕至的山中有一座寺庙。凡是到这个寺庙的和尚都会得到一个钵子。钵子里面有 27 个佛珠。其中 15 个为红色佛珠，12 个为绿色佛珠。每次寺庙的钟声响起时，每个和尚可进行如下两个操作中的任意 1 个：
   （a）如果钵子里面有至少 3 颗红色的佛珠，则和尚可以从钵子里拿走 3 颗红色的佛珠，并放入 2 颗绿色的佛珠（即用 2 颗绿色佛珠替换 3 颗红色的佛珠）。
   （b）和尚也可以在任何一次钟声响起时，将钵子里的所有佛珠置换为另外一种颜色的佛珠（即将原来的红色佛珠替换为绿色佛珠，将原来的绿色佛珠替换为红色佛珠）。

   例如：在第 1 次钟声响起时，和尚可将钵子里的 3 个红色佛珠拿走，并加入 2 颗绿色的佛珠，这样，和尚的钵子里将有 12 颗红色佛珠和 14 颗绿色佛珠；在第 2 次钟声响起时，和尚可将钵子里的每个佛珠置换为相反的颜色。这样，和尚的钵子里将有 14 颗红色佛珠和 12 颗绿色佛珠……当和尚的钵子里面剩下的佛珠为 5 颗红色和 5 颗绿色时，和尚就可以离开寺庙去享受外面的精彩世界了。请问：在不违反上述规则的情况下，和尚应该如何操作才能离开寺庙？
9. 你觉得人生中的问题是“难解”的多，还是“易解”的多？给出你的理由？
10. 众所周知，不同的人写出的程序代码具有不同的质量。如何判定一个人的代码是否优于另外一个人写的代码呢？这个问题是“难解”，还是“易解”？给出理由。

CHAPTER 14

第 14 章　NP 完全问题

在云南丽江的玉龙雪山（见图 14-1）脚下东巴谷里的一条小路上立有一块木牌，请求过路行人对一个案子进行审判。说的是有三个人，甲、乙、丙在玉龙山上射杀了一只老鹰，这三个人的行为因违反《野生动物保护法》而被警察拘捕，但是这三个人都否认老鹰是自己射杀的。经过各种盘查和询问，警察得出了如下三个结论：

1）如果甲是无罪的，则乙和丙都有罪。

2）乙和丙中必有一人有罪。

3）要么甲无罪，要么乙有罪（但两者不能同时成立）。

图 14-1　位于云南省丽江市西北方向的玉龙雪山

现在，警方请求你协助判断甲、乙、丙三人谁有罪，谁没有罪，并在你认为有罪的人前面的筐里投入一块石头。笔者查看了三个筐发现，乙的筐里石头最多，甲和丙的筐里则差不多。

那么，旅客们的判断是否正确呢？这三个人真的都有罪吗？

14.1　玉龙雪山下的审判

这种确定一个逻辑命题是否可以满足称为 SAT（satisfiability）问题。

如果我们以 A、B、C 分别代表甲、乙、丙三人有罪的逻辑命题，则 $\neg A$、$\neg B$、$\neg C$ 就分别代表甲、乙、丙无罪的逻辑命题。而警察得出的三个结论可以表示为以下逻辑命题：

1）$\neg A \rightarrow B \wedge C$。

2）$B \vee C$。

3）$\neg A \oplus B$。

那么，这个问题的解答实际上就是为上述三个布尔表达式找出一个真值指派，使得上述三个表达式同时为真。例如，A 为真、B 为真、C 为真就是一个满足上述三个表达式的真值指派，即甲、乙、丙三个人都有罪。但这并不是唯一的指派，例如，A 为真、B 为真、C 为假的真值指派也满足上述三个布尔表达式。因此，这种情况下有两种可能的解答。但这两种解答里面 A 和 B 都为真，因此甲、乙二人必定有罪。而 C 有真假两种取值，因此丙既可能有罪也可能无罪，即我们从上述条件里面不能肯定丙是否有罪。因此，我们应该投入石块到甲、乙的筐里。

看到这里，不知读者是否意识到，找出一个真值指派似乎很容易。但如果你这么想，那我们请你再想一下。上述真值指派容易是因为我们只有三个子句，且每个子句包含的文字只有 2～3 个。如果子句很多，每个子句里的文字个数很多，指派就不会这么容易了。例如，你能很快地找出下面布尔表达式的一个真值指派使得整个表达式为真吗？

$$(P \vee Q \vee R \vee S) \wedge (\neg P \vee \neg Q \vee \neg S) \wedge (\neg P \vee \neg R) \wedge (\neg R \vee Q \vee \neg S) \wedge (\neg P \vee \neg Q \vee \neg R \vee S)$$

这恐怕得花一些时间吧。对于 SAT 问题，最简单的办法是检查所有可能的真值指派，看看哪一个能够使一个逻辑命题为真。而对于一个有 n 个变量的逻辑命题来说，真值表将有 2^n 行。当 n 超过 30 的时候，整个表的行数将超过 10 亿行！那么有没有什么高效的算法来确定一个 SAT 问题是否有解呢？

答案是：没有人知道！

因此，当你不能很快地确定上述逻辑命题是否能够满足的时候，无需感到难过，因为有效的判定方法到现在还没有发明。事实上，SAT 问题已经被证明是一个 NP 完全问题。

那么，什么是 NP 完全问题呢？

14.2 NP 完全问题的定义

前面一章介绍了 P、NP 和 NP 难，那么这里所说的 NP 完全是什么呢？第 13 章对 NP 难的定义是那些科学家费尽力气也未能找到多项式时间解的 NP 问题。这个定义十分不严谨，它只是给人们一个粗略的概念而已。而真正的 NP 难定义是这样的：如果 NP 里的每一个问题都可以多项式时间规约到 S，则 S 被称为 NP 难。这里的规约指的是转换，即一个问题 Q 可以规约到 S 指的是 Q 可以转换为 S。而多项式规约指的是这个转换可以在多项式时间内完成，因此解决了 S 就解决了 Q，并且它们的解决方案的效率之差不会超过一个多项式。这样，如果 S 能够在多项式时间内解决，则 Q 也能在多项式时间内解决。因此，S 是 NP 难表示的是：S 不比 NP 里面的任何问题容易！

注意　NP 难并不意味着在 NP 里并且很难，因此，本书第 13 章的 NP 难定义从严格意义上来说是错误的！不过，这不妨碍这种定义被很多人士所接受。

如果一个问题 S 既是 NP 难，又是 NP 里的问题，则该问题就被称为 NP 完全问题。因此，NP 完全有两个条件：S 属于 NP，S 为 NP 难。

NP 完全的意思也是两个：

1）非确定性算法多项式时间可解。

2）完全：解决一个，解决一切。

第一条属性来源于 NP 完全问题属于 NP 类，因此有非确定性多项式时间解。第二条属性来源于 NP 难的定义，所有 NP 里的问题都可以规约到 S。因此解决了 S（指找出确定性多项式时间解），就解决了 NP 里的所有问题。事实上，如果能够找到一个 NP 完全问题的确定性多项式时间解，则就证明了 NP=P。

由于决策问题与优化问题对应，因此，如果我们找到了一个 NP 难优化问题的解，则 NP=P。事实上，我们有下面的定理。

定理　如果 NP 难里的任何一个优化问题存在一个多项式时间解，则 P = NP。

证明　令 O 为某一 NP 难优化问题，不失一般性，设该问题为最小优化问题；A 是解决该问题的一个多项式时间算法。设与 O 对应的决策问题为 D，则 D 的一个实例 J 将具有形式(I, c)，这里 I 是 O 的一个实例，c 是一个数。那么对 D 问题实例 J 的回答可以通过如下办法获得：

1）将算法 A 运行在实例 I 上而获得一个解。

2）检查该解的成本是否超越 c。

因此，决策问题 D 存在一个多项式时间解。根据 NP 完全的定义，P= NP。　□

14.3　NP 完全的重要性

直观地讲，NP 完全问题就是 NP 里面最困难的问题！所谓困难，就是到目前为止人们无法在多项式时间内解决。这些问题也许以后也不能在多项式时间内解决，那么讨论这样的问题有何意义呢？当然有意义，NP 完全对于算法设计人员和工程师来说意义重大！

假定分配给你一个任务，而你的同事在这个任务上花了很多的时间，但却没有找到精确解。如果你能够证明这个问题是 NP 完全问题，则就无需再花时间寻找精确解了，而只要找到一个有启发性的近似解（heuristic approximation）即可。这样将节省大量的时间。

研究 NP 完全问题的一个重要讨论是“如何识别困难的问题”。我们要建立这样一个观念：认识到问题的困难性和掌握解决问题的方法同样重要。

NP 完全是一个类，其中包含了上千个问题，它们看似形态各异：图论、集合论、数论、数学规划、计算几何……但它们中的任意两个问题都是可以相互规约的，即如果一个问题能在多项式时间内求解，则另一个问题也能够。

与经典的算法相比，NP 完全性理论只有不到 40 年的历史，但发展却极其迅速。其奠基

人斯蒂芬·库克（Stephen Cook）在 1971 年发表的具有划时代意义的论文“The Complexity of Theorem Proving Procedures”中，证明了电路可满足问题（CIRCUIT-SAT）是 NP 完全问题。次年，理查德·卡普（Richard Karp）提出并证明了 21 个 NP 完全问题。如今，人们已知的 NP 完全问题已超过了 3 000 个。著名的 NP 完全问题包括可满足性问题、汉密尔顿回路问题、完全子图问题、图的着色问题、子集和问题以及旅行售货员问题等。

那么如何证明一个问题是 NP 完全问题呢？

14.4 多项式时间规约

证明一个问题为 NP 完全的办法就是多项式时间规约。

那么什么是规约呢？

假如我们要解决问题 R，而我们已经有一个解决问题 S 的算法，并且有一个转换函数 T，能够将问题 R 转换为问题 S，即如果 R 在输入为 x 时的正确答案为“是”当且仅当 S 在输入为 $T(x)$时的正确答案为“是”，则称 R 可规约到 S（见图 14-2）。

如果转换函数 T 的时间复杂性为多项式，则称 R 可多项式规约到 S。多项式规约的实际意义是，S 的难度不比 R 小，即如果 S 能够解决，则 R 就能够解决。

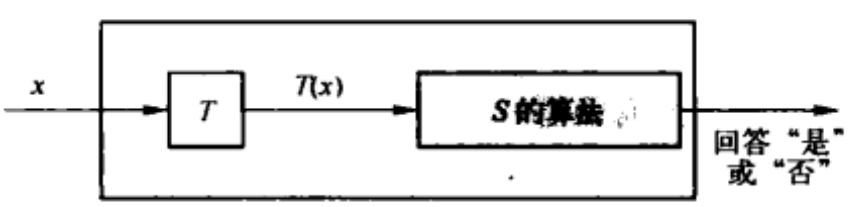


图 14-2 多项式时间规约

如果 E 和 D 为两个决策问题，并且满足如下条件，则称 D 可以多项式时间规约到 E：

1）存在算法 A，该算法以 D 的实例为输入，并总是正确输出“是”和“否”的答案。

2）A 在运行过程中调用一个用于解决问题 E 的假想算法 B。

3）存在多项式 p，对于每个大小为 n 的 D 的实例，算法 A 在不超过 $p(n)$里终结。这里每次 B 子程序调用计做 m 步，这里 m 是 B 的实际输入规模。

多项式时间规约举例

定理 汉密尔顿回路问题可多项式规约到旅行售货员决策问题。

证明 给定图 G，结点为 $v_1, \cdots, v_n$，我们在 G 上构造带权重的完全图 H 如下：如果一条边$\{v_i, v_j\}$在原来的图 G 里，则该边的权重为 1；否则，该边的权重为 2。

显然，图 G 的汉密尔顿回路问题可以通过解决图 H 里的旅行售货员决策问题来解决，这里的旅行销售员实例为$(H, n+1)$。

14.5　如何证明一个问题 S 是 NP 完全问题

有了上面对规约的讨论，我们可以将 NP 完全的证明方法罗列如下：

1）证明 *S* 在 NP 里面。

2）选择一个已知的 NP 完全问题 *R*。

3）证明 *R* 可以多项式规约到 *S*。

由于 *R* 是已知的 NP 完全问题，所有 NP 里的问题都可以多项式时间规约到 *R*，而由于第 3 步证明了 *R* 可以多项式规约到 *S*，因此，所有 NP 里的问题都可以多项式规约到 *S*。由于第 1 步证明了 *S* 在 NP 里，因此 *S* 是 NP 完全。

细心的读者也许能看出来，这个证明方法存在一个问题：第 1 个 NP 完全问题怎么办呢？在证明第 1 个 NP 完全问题的时候，尚不存在任何已知的 NP 完全问题，因此上述方法无法施行。

因此，我们需要一种不同的办法来证明第 1 个 NP 完全问题！而这个证明要难得多，所谓的万事开头难。

14.6　第 1 个 NP 完全问题的证明

用什么办法来证明第 1 个 NP 完全问题呢？自然，没有别的办法，只能根据 NP 完全的定义来证明，也就是要将所有的 NP 问题规约到所要证明的问题上。但是，NP 问题数量繁多，这样证明得过来吗？即使精力过人，也未免会漏掉某个 NP 问题，从而导致证明失败。况且， NP 类问题的数量到底有多少谁也说不清！

希望似乎在丧失，但不要气馁！显然，我们不可能逐个 NP 问题来规约，但谁说过我们必须这么做呢？还记得 NP 问题的定义吗？该定义使用了图灵机！也就是说，所有 NP 问题都可以用图灵机来表示，因此，我们可以将所有 NP 问题一般化，抽象成一个问题！这样，我们只需要证明一次即全部搞定！这就是抽象的能力！

这正是斯蒂芬·库克用的方法。库克第在 1971 年证明了布尔可满足性问题（SAT 问题）是 NP 完全问题（该证明出现在其发表于 1971 年的“The Complexity of Theorem Proving Procedures”文章里），从而启开了 NP 完全理论的风帆，并因此于 1982 年获得图灵奖。利奥尼德·莱文（Leonid Levin）在其 1972 年发表的论文“Universal Search Problems”里面也独立地证明了该问题为 NP 完全。下面我们就来重温库克的极为精彩的证明！

14.7　库克定理

库克定理　*布尔可满足性问题属于 NP 完全问题。*

换个方式说，任何可以在多项式时间内被非确定性图灵机解决的问题都可以多项式规约到判断布尔可满足性问题是否可满足。该定理的一个直接推论就是，如果我们找出了解答布

尔可满足性的一个多项式时间算法，则任何 NP 问题的多项式算法都将获得解决。

布尔可满足性问题的实例是所谓的布尔表达式，而布尔表达式是由布尔操作符连接的布尔变量。根据布尔理论，所有布尔表达式都可以表示为合取范式：一个合取范式（CNF）由若干个子句（clause）用逻辑“与”连接起来，每个子句由若干个文字（literal）用逻辑“或”连接起来，每个文字是一个布尔变量或其否定。

我们称一个布尔表达式是可以满足的，如果存在一个真值指派（即每个布尔变量的取值），使所有子句都为真（从而整个 CNF 为真）。可满足性问题（SAT 问题）是说，给定一个 CNF，要么给出它的一个真值指派，要么报告它不存在真值指派。

下面我们就来证明库克定理。

证明 首先，我们证明布尔可满足性问题属于 NP。

由于一个非确定性图灵机可以在多项式时间内完成下列任务：

1）猜测一个真值指派。

2）在该真值指派下，对布尔表达式求值。

3）如果求值为真，则接受整个布尔表达式为真，即判该布尔表达式为可满足的。

按照 NP 的定义，布尔可满足性问题属于 NP。

然后，我们证明 NP 里的所有问题都可以规约到布尔可满足性问题。

这里的难度是如何表述 NP 里的所有问题。从 NP 的定义知道，一个问题是 NP 问题，如果确定性图灵机可以在多项式时间内验证一个潜在证人是否为真，或者一个非确定性算法可以在多项式时间内解决这个问题。因此，可以将所有 NP 问题表述为非确定性图灵机可解决的问题。

设 NP 里面的问题可以被非确定性图灵机 $M=(Q,\Sigma,s,F,\delta)$ 解决，即 M 将在 $p(n)$时间内接受或拒绝 NP 问题的一个实例。其中 n 是实例的大小，p 是一个多项式函数。这里：

- Q 是图灵机 M 所有状态的集合：$0,1,2,\cdots,q-1$。
- Σ 是所有（输入输出）符号的集合：$a_1,a_2,\cdots,a_m$。
- $s\in Q$ 为图灵机最初状态；$F\subseteq Q$ 是终结状态集合；$\delta: Q\times\Sigma\rightarrow Q\times\Sigma\times\{-1,+1\}$ 为状态转换函数。

不失一般性，问题的实例被写在图灵机磁带的 1,2,3,…,n 格子里，如图 14-3 所示。

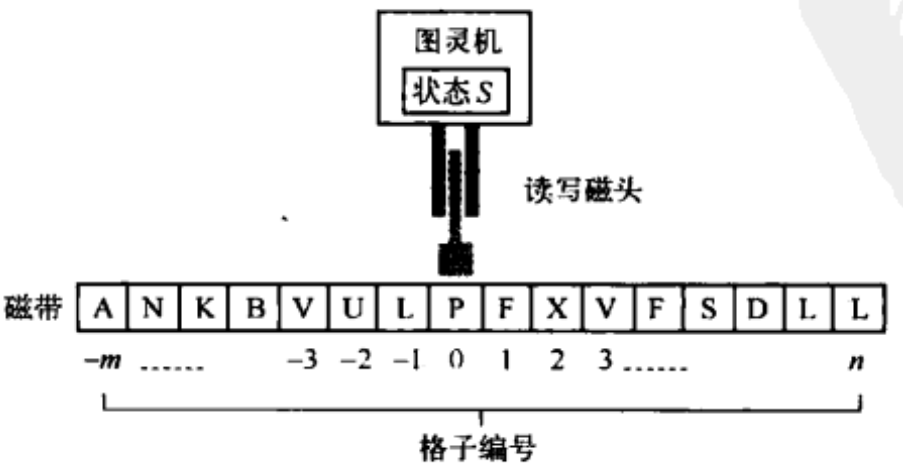


图 14-3 库克定理的证明用到了图灵机

潜在的证人写在格子$-m,\cdots,-2,-1$里，这是我们的正确猜测。格子 0 是分隔符。

该图灵机进入终结状态后停止运行。

我们证明的基本思路是对于每个输入 I 构造一个布尔表达式，该表达式是可满足的当且仅当图灵机 M 接受输入 I（即处理完输入 I 后，机器正好到达某个接受状态）。

我们构造布尔表达式所用的变量如表 14-1 所示，这里 $q\in Q$，$-p(n)\leqslant i\leqslant p(n)$，$j\in\Sigma$ 且 $0\leqslant k\leqslant p(n)$。

表 14-1　构造布尔表达式所用的变量列表

变　量	变 量 含 义	数　量
T_{ijk}	如果第 k 步时格子 i 的内容为 j，则该变量指派真值，否则指派假值	$O(p(n)^2)$
H_{ik}	如果第 k 步时磁头在格子 i 上，则该变量指派真值，否则指派假值	$O(p(n)^2)$
Q_{qk}	如果第 k 步时 M 处于状态 q，则该变量指派真值，否则指派假值	$O(p(n))$

这里需要注意的是，根据非确定性图灵机的定义，最多只能有 $p(n)$步骤，因此，$0\leqslant k\leqslant p(n)$。$j$ 代表的是符号，而状态机的符号表是常数；i 代表的是磁带上的格子，最多只能往左右两边各延伸 $p(n)$个格子，因此$-p(n)\leqslant i\leqslant p(n)$。

我们将布尔表达式 B 定义为表 14-2 中所有子句的合取，这里$-p(n)\leqslant i\leqslant p(n)$ 且 $0\leqslant k\leqslant p(n)$。

表 14-2　构造的布尔表达式 B 为表里所有子句的合取

子　句	条　件	含　义	数　量
T_{ij0}	格子 i 的内容为 j	磁带的初始内容	$O(p(n))$
Q_{s0}	—	M 的初始状态	$O(1)$
H_{00}	—	读写头的初始位置	$O(1)$
$T_{ijk}\rightarrow\neg T_{ij'k}$	$j\neq j'$	一个单元只有一个符号	$O(p(n)^2)$
$T_{ijk}=T_{ij(k+1)}\vee H_{ik}$	—	磁带状态不变	$O(p(n)^2)$
$Q_{qk}\rightarrow\neg Q_{q'k}$	$q\neq q'$	一次只能一个状态	$O(p(n))$
$H_{ik}\rightarrow\neg H_{i'k}$	$i\neq i'$	磁头一次只有一个位置	$O(p(n)^2)$
以下子句的析取 $(H_{ik}\wedge Q_{qk}\wedge T_{i\sigma k})\rightarrow$ $(H_{(i+d)(k+1)}\wedge Q_{q'(k+1)}\wedge T_{i\sigma'(k+1)})$	$(q,\sigma,q',\sigma',d)\in\delta$	第 k 步时磁头在位置 i 的所有可能状态转换	$O(p(n)^2)$
所有终结状态 Q_f 的析取	$f\in F$	必须到达终结状态	$O(1)$

如果按照表 14-2 的子句组成合取范式 B，则有：

1）如果图灵机 M 接受输入 I（即进入终结状态），则 B 是可满足的。此时只要给所有变量 T_{ijk}, H_{ik}，Q_{ik} 进行表 14-2 中的真值指派即可满足 B。

2）如果 B 可满足，则存在一个状态转变序列，使得 M 接受 I（即进入终结状态）。此时只需要按照真值指派的计算结果移动图灵机和进行状态转换即可。

例如，如果图灵机接受 I，则 M 在计算过程中所经历的状态都将获得真的真值指派，所有状态转换都将被转换为取值为真的蕴含子句，因而表 14-2 里所有子句的合取范式将取值为真。

因此，非确定性图灵机 $M=(Q,\Sigma,s,F,\delta)$，即所有的 NP 问题都可以规约到布尔可满足性问题上。现在我们需要知道的是此种规约是否为多项式时间。

显然，规约的时间不会超过构造出来的 B 的大小。那么 B 有多大呢？我们来看：

1）布尔变量的个数不会超过 $O(p(n)^2)$，每个变量占用的空间不会超过 $O(\log p(n))$。

2）布尔子句的条数不会超过 $O(p(n)^2)$，所以 B 的大小不会超过 $O((\log p(n))\,p(n)^2)$。

因此，整个转换过程为多项式时间，这里 n 是输入的规模。

库克定理证明完毕。 □

有了库克定理，证明其他 NP 完全问题就简单多了。下面我们就来证明一些。

14.8 3-SAT 问题

3-SAT 是布尔可满足性的一个特例，在该特例下，所有的子句都只包括 3 个且仅包括 3 个文字。虽然 3-SAT 看上去比一般的布尔表达式更简单，但它也是 NP 完全问题！

显然，3-SAT 的 NP 完全性表明一般的布尔可满足性问题也是 NP 完全问题，但反向推理却不成立。也许是子句的长度增加使得可满足性问题变得困难？

显然，1-SAT 不是 NP 完全问题。它的解答十分容易，只要将每个子句对应的文字赋值真值即可满足。而如果这个赋值不可能（因为有矛盾），则该表达式就是不可满足。不管什么情况，这种解答所需要的时间不会超过子句个数。而子句个数不会超过 n 个（变量个数）。

2-SAT 属于 P 问题，这点请读者自行证明。

下面我们证明 3-SAT 属于 NP 完全问题。

定理 3-SAT 是 NP 完全问题。

证明 首先，3-SAT 属于 NP。因为给定一个真值指派，我们可以在多项式时间内验证每个子句是否为真。

其次，我们来证明 3-SAT 是 NP 难。我们通过将一般的布尔可满足性问题规约到 3-SAT 来证明这点。我们的规约按照子句的长度来进行。

将 SAT 规约到 3-SAT。

不失一般性，假定 SAT 问题的某一子句 C_i 包含 k 个文字。我们将 C_i 改造为长度刚好为 3 的子句的合取。我们的改造根据 k 的大小来进行。

如果 $k=1$，即 $C_i=\{z_1\}$，则增加两个新变量 v_1 和 v_2，并以下面 4 个子句来替换原来的 C_i：

$$\{v_1, v_2, z_1\}\wedge\{v_1, \neg v_2, z_1\}\wedge\{\neg v_1, v_2, z_1\}\wedge\{\neg v_1, \neg v_2, z_1\}$$

可以很容易验证，上述 4 个子句的合取表达式的取值与 $\{z_1\}$ 的取值完全等价！

如果 $k=2$，即 $C_i=\{z_1, z_2\}$，则增加一个新变量 v_1，并将 C_i 用下面 2 个子句来替换：

$$\{v_1, z_1, z_2\} \wedge \{\neg v_1, z_1, z_2\}$$

如果 $k=3$，即 $C_i=\{z_1, z_2, z_3\}$，我们直接使用。

如果 $k>3$，即 $C_i=\{z_1, z_2, \cdots, z_k\}$，则增加 $k-3$ 个新变量 v_1，…，v_{k-3}，并将原来的子句用下面的 $k-2$ 个子句来替换：

$$\{z_1, z_2, v_1\} \wedge \{\neg v_1, z_3, v_2\} \wedge \{\neg v_2, z_4, v_3\} \wedge \cdots \wedge \{\neg v_{k-3}, z_{k-1}, z_k\}$$

这样，所有的布尔可满足性问题的子句都被转换为长度为 3 的子句。由于任何 SAT 的解决方案也将满足如此构造的 3-SAT 问题，而任何 3-SAT 的解决方案同样也满足原来的 SAT，因此，这样转换出来的 3-SAT 问题与原来的 SAT 问题完全等价。

这个转换需要多少时间呢？如果原来的 SAT 实例有 n 个子句，并使用 m 个不同的文字，则我们的转换最多需要时间 $O(nm)$。因此，SAT 多项式时间规约到 3-SAT。

一个更简单的估计是注意到 4 种转换方式，第一种转换方式将 1 个子句转换为 4 个子句；第二种转换将 1 个子句转换为 2 个；第三种转换将 1 个子句转换为 1 个；第四种转换将 1 个子句转换为 $k-2$ 个子句，而 $k-2$ 不会超过 n。因此，在最坏的情况下，原来的 n 个子句被转换为不超过 n^2 个子句。因此，这个转换为多项式时间。

这样，我们就证明了 SAT 可以多项式时间规约到 3-SAT。由于 SAT 是 NP 完全问题，因此，3-SAT 也是 NP 完全问题。 □

如果对 3-SAT 的证明稍加修改，就可以证明 4-SAT, 5-SAT……等都是 NP 完全问题。也许，至少包括 3 个文字是使得 SAT 问题成为 NP 完全问题的关键因素！

14.9 证明 NP 难的技巧

NP 难的证明是一门技术，不是一门科学。没有什么一成不变的公式可以采用。用一句算法领域的行话说，就是 NP 难的证明不存在一个确定性算法！唯一的办法是不断练习。但是一旦熟练了，你就会发现 NP 难的证明非常简单。

对于 NP 难的证明，还有一些小贴士可供参考。

贴士 1 尽量简化原问题。例如，不要直接证明旅行销售员问题为 NP 难，而是证明汉密尔顿回路问题为 NP 难。因为汉密尔顿回路问题比旅行销售员问题要容易证明得多，而汉密尔顿回路问题又可以很容易地规约到旅行销售员问题上！而对于更精明的人来说，甚至汉密尔顿回路问题都不需要证明，直接证明汉密尔顿回路问题即可。

贴士 2 选择合适的 NP 完全问题作为规约的对象。比较常使用的 NP 完全问题包括：

- 3-SAT。当其他问题都难以规约到所要证明的问题时可考虑使用。
- 整数分割（integer partition）。如果要证明的问题涉及很大的数时使用。
- 顶点覆盖（vertex cover）。证明任何需要进行选择的、与图相关的问题时使用。
- 汉密尔顿回路（Hamiltonian path）。证明任何依赖排序的问题时考虑使用。

贴士 3 从战略高度着想，然后构造各种巧件（gadget）来实现战术。构造巧件前多问问自己如下几个问题：如何才能迫使选择 A 或 B，但不是同时被选择？如何迫使 A 在 B 之前

被选择？如何清除那些没有被选择的东西呢？当有了这些解答后，你就可以更容易地构造所需的东西。

贴士 4 使用限制。证明问题的一个特例或一部分是 NP 完全问题。

贴士 5 局部替换。将局部的东西进行替换从而达到规约。如我们的 SAT 到 3-SAT 规约。

贴士 6 当遇到困难时，也许应该看看这个问题是否可以用算法解决？

下面我们就用上面介绍的方法来证明几个具体的 NP 难和 NP 完全问题。

14.10 整数规划

整数规划（Integer Programming，IP）有时也称为整数线性规划（Integer Linear Programming），因为它是标准线性规划的一个特例，即每个解的取值都必须是整数！

整数规划的问题定义是：给定一个整数变量集合 V 和在 V 上定义的一组不等式，另有一个定义在 V 上的函数 $f(v)$和一个整数 B，那么是否存在一组变量赋值，使得所有的不等式成立，并且 $f(v) \geqslant B$? 下面给出的是一个整数规划的具体例子：

- $v_1 \geqslant 1, v_2 \geqslant 0$，$v_1 + v_2 \leqslant 3$。
- $f(v) = 2v_2$。
- $B = 3$。

这看上去似乎是一个很简单的问题。v_1=1，v_2=2 就是上述整数规划问题的一个解。但这个问题真的简单吗？当然不是。如果不等式很多，函数很复杂的时候，恐怕我们就不能一眼看出答案了。事实上，该问题是一个 NP 难问题，但不一定是 NP 完全问题。下面我们就来证明。

定理 *整数规划问题是 NP 难问题。*

证明的方法是将 SAT 问题归约到整数规划上，即对于一个任意的 SAT 问题，我们可以将其转换为一个 IP 问题。如果该 SAT 问题有 n 个文字和 m 个子句，则转换出来的 IP 问题有 $2n$ 个变量，即每个文字和它的补文字满足下面的不等式：

$$0 \leqslant v_i \leqslant 1 \tag{14-1}$$

$$0 \leqslant \neg v_i \leqslant 1 \tag{14-2}$$

$$1 \leqslant v_i + \neg v_i \leqslant 1 \tag{14-3}$$

对于每一个子句 $C = \{v_1, \neg v_2, \cdots, v_i\}$，构造不等式如下：

$$v_1 + \neg v_2 + \cdots + v_i \geqslant 1 \tag{14-4}$$

而 B=1。从这个转换可以看出，我们需要证明任何 SAT 问题的解可以导出上述 IP 问题的解，而任何上述 IP 问题的解也可以导出原 SAT 问题的解。

⇒ 任何 SAT 问题的解可以导出上述 IP 问题的解。

如果原 SAT 问题有解，则该解也是上面 IP 问题的解。只需将赋值指派为“真”的变量在 IP 问题里取 1，而指派为“假”的变量在 IP 里取值为 0 即可。由于所有变量的取值只有 1

和 0 两种可能，因此，式（14-1）、式（14-2）、式（14-3）不等式都成立。又由于该真值指派是 SAT 的解，因此每个子句 C 的取值都为真，也就是 C 子句里至少有一个文字的取值为 1，因此，子句 C 里所有文字变量之和必然大于等于 1。

⇐ 任何上述 IP 问题的解也可以导出原 SAT 问题的解。

反过来看，如果上述 IP 问题有解，则原 SAT 问题也有解。由于上述 IP 问题里，一个变量的取值只可能是 1 或者 0，我们将取值为 1 的变量在原 SAT 问题里指派“真”，而取值为 0 的在原 SAT 问题里指派“假”。由于式（14-3）的存在，一个变量和其补文字有且仅有一个为“真”，也就是上述构造的真值指派是一个合法指派。又由于式（14-4）成立，因此，每个子句 C 里至少有一个文字的真值指派为“真”，也就是 C 的取值为真。因此原 SAT 问题有解。

这里需要注意的是，由于我们可以给一个变量赋予一个巨大的值，因此将该值写下来就需要超出多项式时间的复杂性，从而导致对一个潜在解的检查超出多项式时间。因此，整数规划并不属于 NP 问题，所以它不是 NP 完全问题。但它是 NP 难问题。

14.11　独立集问题

对于一个无向图 $G=(V, E)$，如果一个顶点集合 I 中任意两点之间都没有边相连，则 I 称为**独立集**（independent set）。设 $VC \in V$ 是另一个顶点集合，如果图中的每条边所依附的两个顶点中，至少有一个在 VC 中，我们称 VC 覆盖了 G 的所有边，即 VC 是 G 的顶点覆盖。另设 $C \in V$，若 C 中任意两个顶点间都有边相连（即 C 中的顶点构成了一个完全图），则 C 称为**团集**。一般来说，我们希望找到尽可能大的独立集、尽可能小的顶点覆盖和尽可能大的团集。

独立集问题的定义是：给定一个图 $G=(V, E)$ 和整数 k，是否存在一个至少 k 个结点的子集 S，使得 E 里没有边连接 S 里的任何两个结点。例如，图 14-4 里的结点集合 $\{A,C,E,G\}$ 就是一个 4 个结点的独立集；而 $\{A,B,C\}$ 就不是一个独立集，因为 $\{A,B\}$ 被 E 里的一条边连接。

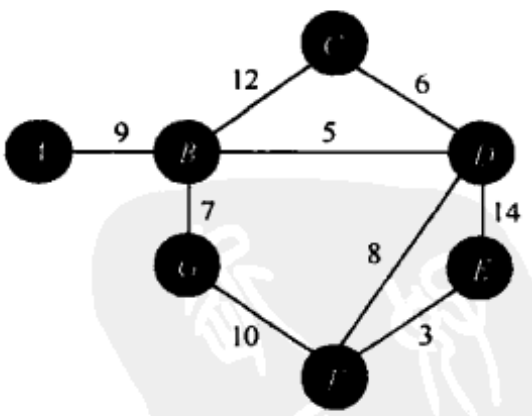


图 14-4　独立集问题

这看上去似乎也是一个简单问题，但与整数规划问题一样，也是 NP 完全问题。

定理　独立集问题是 NP 完全问题。

证明　首先，独立集问题属于 NP 问题。给定任意一个结点的子集，我们可以检查集合里的结点之间不存在边，而这个检查可以在多项式时间内完成。

下面我们证明该问题是 NP 难。而证明的方法则是将 3-SAT 问题归约到独立集问题上。那么如何规约呢？显然，要从 3-SAT 规约到独立集，则必须将 3-SAT 里面的子句转换为图。由于 3-SAT 里的每个子句都包含 3 个文字，我们很自然地将其转换为一个三个顶点的完全子

图。例如，如果子句为$\{\neg v_1,v_2,\neg v_3\},\{v_1,\neg v_2,\neg v_4\},\{\neg v_2,\neg v_4,v_5\},\{v_3,v_4,v_5\}$，则可以构造出如图14-5的4个完全子图。

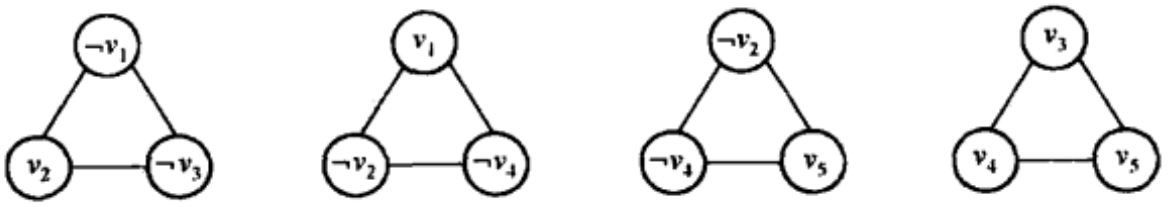


图14-5 根据子句构造的4个完全三顶点子图

这样，由于一个3-SAT的解将使得任何一个子句都得到满足，即每个子句里面至少有一个文字赋值为“真”。如果一个子句里只有一个“真”值，则我们就将取“真”值的顶点归于独立集。如果一个子句里面有多个“真”值，则任选一个放入独立集。如果一个3-SAT问题有k个子句，则这样构造出的独立集的大小为k。而3-SAT的任意一个解都可以按照上面描述转换为一个大小为k的独立集解。

但是这样构造的独立集解是否能推导出原3-SAT问题的解呢？答案是不一定。虽然我们可以在每个完全子图里面选取一个结点作为独立集集合的一个元素，从而形成一个大小为k的独立集，但我们不能通过对独立集里的元素赋“真”值来取得3-SAT的解。原因是，这样形成的独立集有可能包括一个文字的正负两面，从而导致无法进行真值指派。

那么如何防止这个问题呢？答案也很简单，只需要在对应正负文字的顶点间增加一条边即可。例如，对于前面的例子，在增加这些边后，我们获得如图14-6所示的一个构造。这个图是否存在一个大小为k的独立集与有k个子句的3-SAT问题可以相互转换，即原3-SAT是可满足的当且仅当图中存在大小为k的独立集。下面给出具体证明。

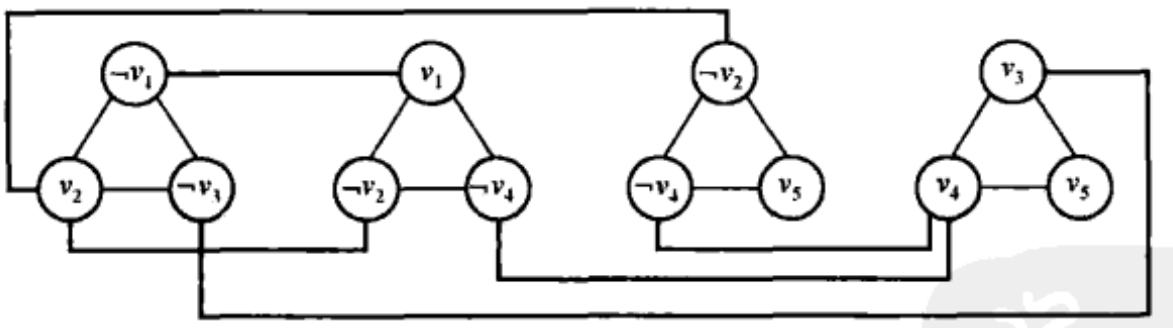


图14-6 添加附加边后的连通图（黑线加粗线为附加边）

证明 ⇒3-SAT是可满足的，则图中存在大小为k的独立集。

如果3-SAT有解，则存在真值指派，且每个子句中必存在（至少）一个文字，它在真值指派中为真。从每个子句中选出一个这样的文字（若有多个，则任意选择一个）。这k个文字对应的顶点构成大小为k的独立集。因为每个子句中仅选出一个文字，故不会违反第一类边的约束；又因为一个3-SAT的解里面不可能包括x和$\bar{x}$同时为真的可能，因此x和$\bar{x}$不可能被同时选入，故该结点集合也不会违反第二类边的约束。

⇐如果图中存在大小为k的独立集，则3-SAT是可满足的。

如果这样构造出来的独立集问题有解，则每个完全三顶点子图里面有且仅有一个顶点入

选独立集。多于 1 个肯定是不可能的，但少于 1 个则无法形成一个大小为 k 的顶点集。又由于图中正负都存在的文字间有边相连，则代表文字 x 和 $\bar{x}$ 的顶点不可能同时入选独立集。这样一来独立集里的文字指派为“真”将不会产生矛盾，而这个“真”值指派完全满足原 3-SAT！

因此，3-SAT 问题可以规约到独立集上。又由于这种规约是多项式时间，因此，独立集问题为 NP 完全问题。 □

细心的读者可能已经发现，独立集、顶点覆盖和团集是三个密切相关的问题。它们之间可以互相转化，因为证明了独立集为 NP 完全问题，也就证明了其他两个问题也是 NP 完全问题。读者可自行进行证明。

14.12　汉密尔顿回路问题

汉密尔顿回路问题的定义是：给定一个图 G，是否存在一个包括所有结点 1 次且仅 1 次的环路？汉密尔顿回路问题是由爱尔兰物理学家和数学家威廉·罗万·汉密尔顿（William Rowan Hamilton）于 1857 年发明的一个数学游戏——汉密尔顿迷宫，其目标是在一个十二面体里寻找一个汉密尔顿回路。虽然汉密尔顿利用艾科西亚演算（Icosian 微积分）解决了一些特殊情况下的汉密尔顿回路问题，但对于一般的图来说，汉密尔顿回路问题仍然是一个难解之题。事实上，我们有如下的定理。

定理　汉密尔顿回路问题是一个 NP 完全问题。

证明　首先，汉密尔顿回路问题属于 NP 问题。因为给定任意一个结点序列（潜在的汉密尔顿回路），我们可以在多项式时间内验证相邻的两个结点是否有边相连，并且最后一个结点是否与第 1 个结点有边相连。因此，汉密尔顿回路问题为 NP 问题。

要证明汉密尔顿回路为 NP 难，根据前面讨论过的 NP 完全证明技巧，我们选择另一个已知的 NP 完全的图的问题来规约。由于汉密尔顿回路问题属于图论中的问题，很自然，我们选择一个图论问题作为规约源。我们选择的是顶点覆盖问题（事实上，几乎所有的图论 NP 完全问题都可以作为规约源），顶点覆盖问题的 NP 完全性质留给读者去证明。

我们的思路就是对于图 G 和整数 k，我们构建另一个图 H，使得图 H 存在汉密尔顿回路当且仅当 G 有一个大小为 k 的顶点覆盖。那么这另一个图 H 如何构建呢？

规约的思路 1：子构件的构造

要回答上面提出的问题，先得分析一下顶点覆盖问题的性质。顶点覆盖的核心是使用一组顶点来覆盖所有的边！而每条边$\{u, v\}$被覆盖的方式只有三种：被顶点 u 覆盖、被顶点 v 覆盖、被顶点 u 和 v 同时覆盖。因此，在构建图 H 时，我们针对图 G 的每条边，需要构造某种子构件，使得该构件被囊括在图 H 的一个汉密尔顿回路里面的方式也有三种，分别对应图 G 里该条边被覆盖的三种情形。这样我们便可以较容易地得出子构件的设计。

对于图 G 顶点覆盖问题里的每一条边$\{u, v\}$，我们把它变为图 H 的汉密尔顿回路问题里的一个子构件 W_{uv}。该子构件有两排 12 个结点，每排 6 个结点，分别对应边$\{u,v\}$的一个结点。两排之间用 4 条边连接，如图 14-7 所示。

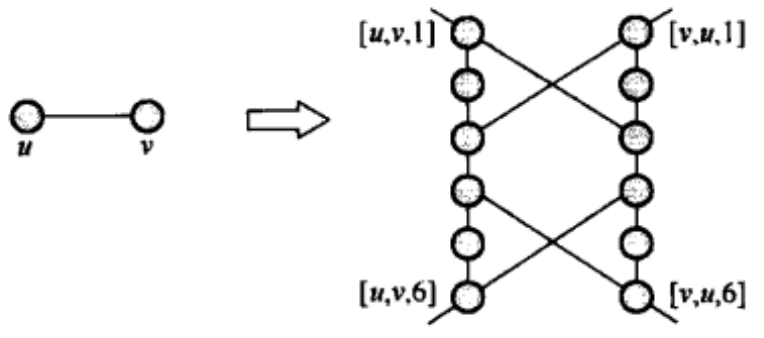


图 14-7 针对每条边$\{u, v\}$构造一个子构件 W_{uv}

不难看出，囊括此种子构件里所有结点的回路只有图 14-8 给出的三种情形。这里图 14-8a 对应的是图 G 的边$\{u, v\}$被顶点 u 和 v 同时覆盖的情形，14-8b 对应被顶点 v 覆盖的情形，而 14-8c 对应被顶点 u 覆盖的情形。

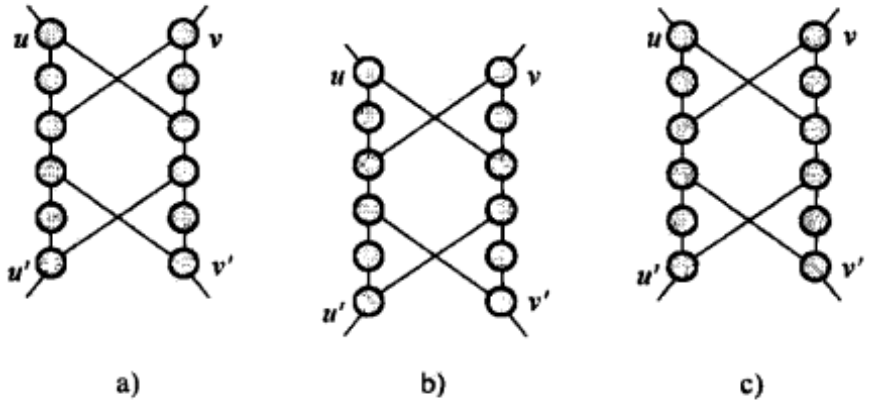


图 14-8 每个子构件只有 3 种可能的环路

规约的思路 2：子构件的连接

在顶点覆盖问题里，一个顶点可以覆盖与其相连的所有边。因此，在构造图 H 的时候，我们必须将一个顶点所对应的所有子构件连接起来，使其可以镶嵌在图 H 的某个回路上。这样，我们就获得了子构件的连接设计：对于图 G 的每一个顶点 v，我们将图 H 里面由边$\{v, w\}$所构建的子构件按如下方式进行连接：假定 v 的邻结点为 $x_1, \cdots, x_r$，增加边：$\{[v,x_1,6], [v,x_2,1]\}, \{[v,x_2,6], [v,x_3,1]\}, \cdots, \{[v,x_{r-1},6], [v,x_r,1]\}$，也就是将按边构造的子构件里面对应同一个顶点的那排（6 个结点）进行首尾相连，形成**子构件串**，如图 14-9 所示。

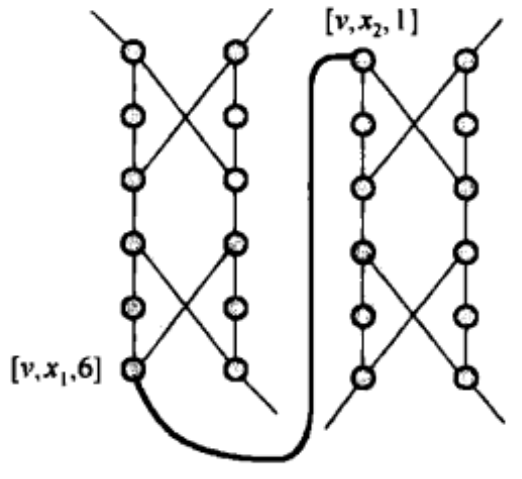


图 14-9 将不同子构件里面对应同一个顶点的那排首尾相接形成子构件串

图 14-10 描述的是从一个完整的图 G 构建对应图 H 的子构件及其连接的情况。

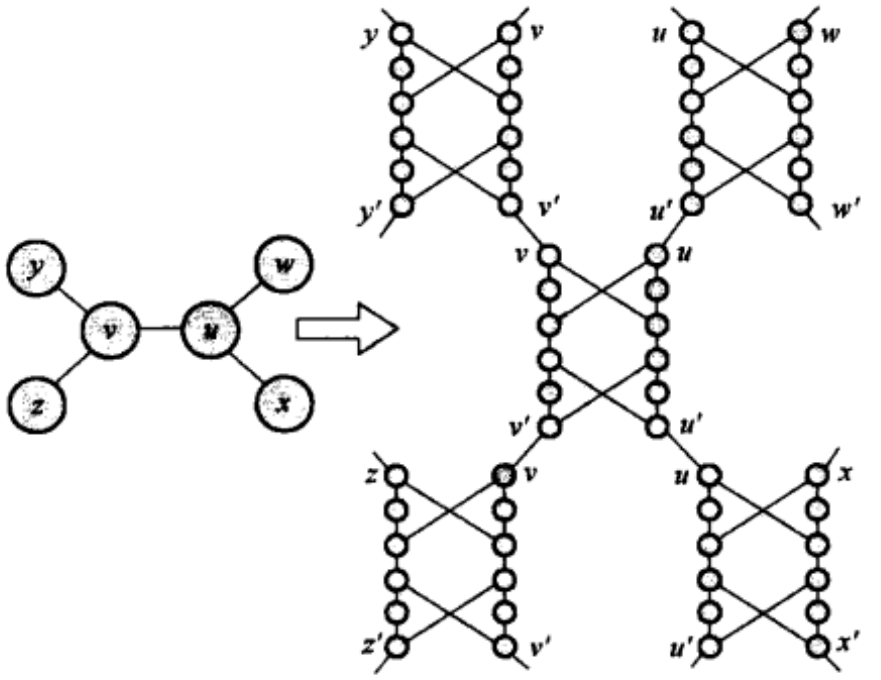


图 14-10　从左面的图 G 构造出图 H 的子构件及其连接

至此，所有的子构件及其连接都构造完毕。下面需要将顶点覆盖问题里的整数 k 构建到图 H 里来。最直接的构造方式就是在图 H 里增加 k 个顶点，分别对应图 G 里被选择的 k 个顶点。但我们如何将这 k 个顶点与 H 里面已经构造好的子构件进行连接呢？

规约的思路 3：顶点的连接

我们在图 G 里添加 k 个选择器顶点 $s_1,\cdots, s_k$，它们代表顶点覆盖问题里被选择的顶点。将每个选择顶点连接在每个子构件串的首尾顶点形成一个回路。将每个子构件串的第 1 个顶点$[v,x_1,1]$和最后一个顶点$[v,x_r,6]$都连接在每个选择顶点 s_i 上，如图 14-11 所示。

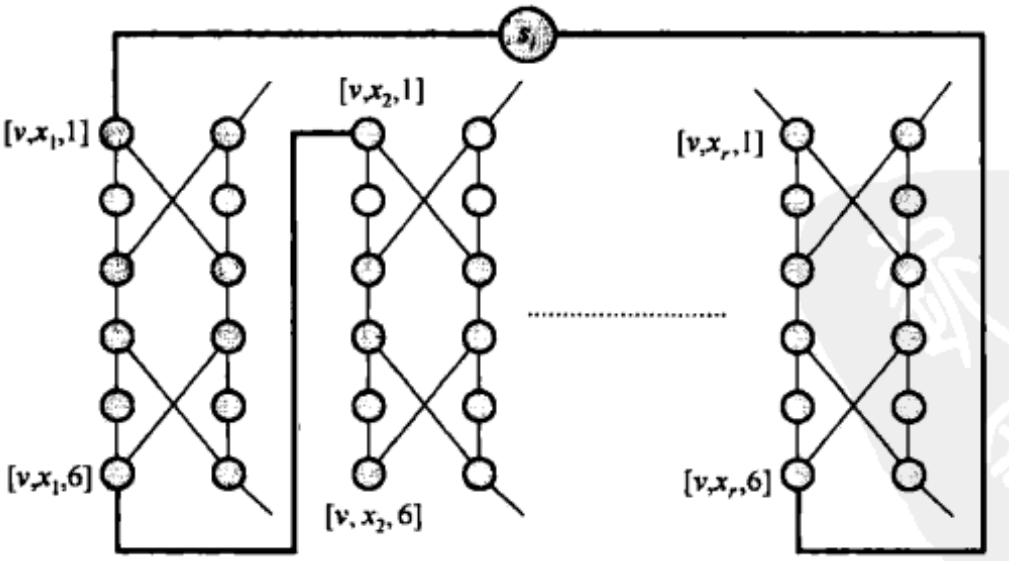


图 14-11　将每个选择顶点与由每个顶点所构造的子构件串连接起来

图 14-12 描述的是从一个完整的图 G 构建对应图 H 的子构件串及其与选择顶点的连接。

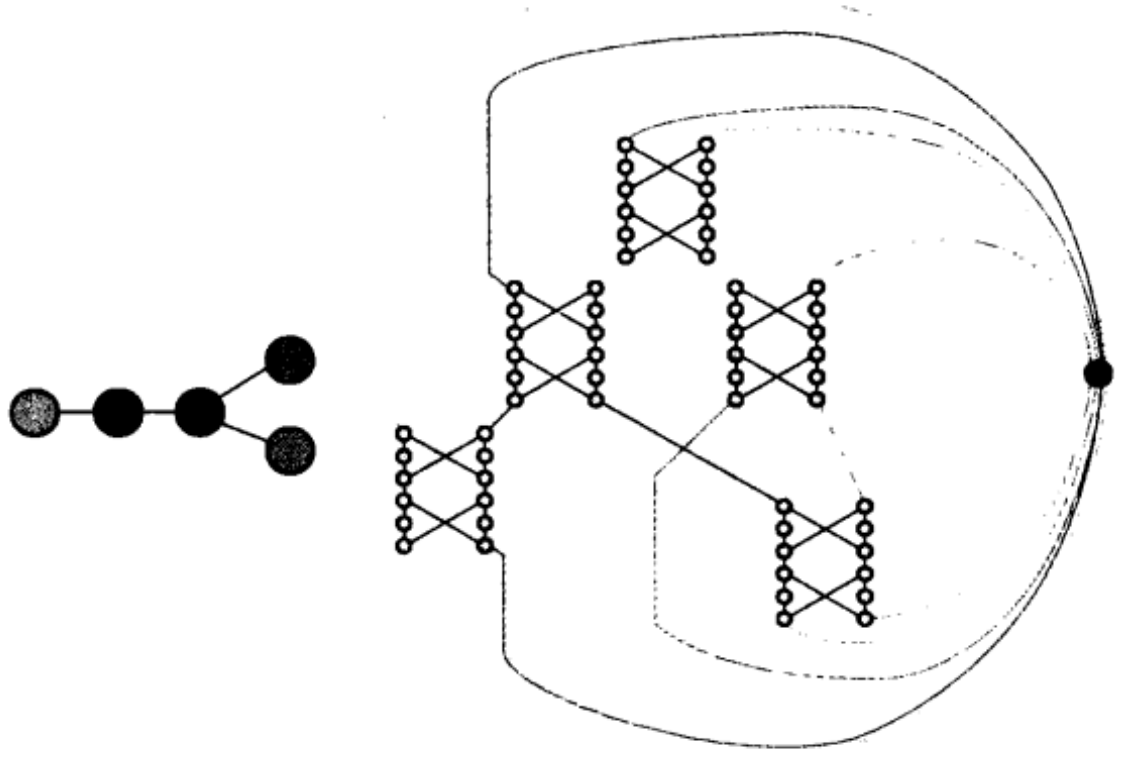


图 14-12 从左面的图 G 构造出图 H 的子构件串及其与任意一个选择顶点的连接

至此，我们的图 H 构造完成。而图 G 存在一个大小为 k 的顶点覆盖当且仅当图 H 存在一个汉密尔顿回路，而且图 H 的构造可在多项式时间内完成。

正确性证明

⇒ 图 G 存在大小为 k 的顶点覆盖，则图 H 存在汉密尔顿回路。

如果图 G 存在大小为 k 的顶点覆盖，设该覆盖为 $\{v_1,v_2,\cdots,v_k\}$。则我们在图 H 里如此构造一个回路：从选择顶点 s_1 开始，连接由顶点 v_1 所构建的子构件串，然后连通 s_2，之后再连接由顶点 v_2 所构建的子构件串……然后连通 s_k，之后再连通到由顶点 v_k 所构建的子构件串，最后在连通到选择结点 s_1。显然，如此构造的肯定是一个回路。问题是，它是否是汉密尔顿回路呢？由于 $\{v_1,v_2,\cdots,v_k\}$ 覆盖了所有的边，因此由这 k 个结点开始的子构建串囊括了图 H 里的子构件，加上回路里包含的所有选择结点，我们上面构造的回路通过了图 H 所有的顶点。因此，该回路确实是汉密尔顿回路。

⇐ 如果图 H 存在汉密尔顿回路，则图 G 存在大小为 k 的顶点覆盖。

如果图 H 存在汉密尔顿回路，由于选择顶点之间没有直接的边，则该汉密尔顿回路从一个选择顶点出来，下一个顶点必然是一个子构件的起始顶点。又由于不同的子构建串之间没有直接的边，因此它们必须通过选择顶点连接。由于图 H 一共有 k 个选择顶点，因此任何汉密尔顿回路一共只能经过 k 个且仅 k 个子构建串。而对应这 k 个子构建串的顶点数也是 k 个！问题是，这 k 个顶点是否真是图 G 的一个覆盖呢？由于这 k 个子构件串里面的子构件就是按照图 G 里的边所构建的，因此对应这 k 个子构建串的顶点覆盖了 G 里面的所有边。所以，图 G 存在一个大小为 k 的顶点覆盖集合。

因此，顶点覆盖问题确实可以规约到汉密尔顿回路问题上。又由于这种规约是多项式时间（留给读者去验证），因此汉密尔顿回路问题为 NP 完全问题。 □

14.13　讨论：弱 NP 完全、强 NP 完全和中 NP 完全

在计算复杂性理论中，把运行时间是输入数值（而非长度）的多项式函数的算法称为**伪多项式时间的**（pseudo-polynomial time）。当输入数值在“合理”的范围内时，这类算法能很好地工作；只有当输入数值呈指数级增长时，算法的运行时间才表现出指数增长的性质。这样的 NP 完全问题称为**弱 NP 完全**。而其他一些 NP 完全问题，如旅行售货员问题，当输入数值以多项式级增长时，算法的运行时间就呈指数级增长，它们称为**强 NP 完全**。

换句话说，强 NP 完全问题是指那些在数字表示为单元（unary）时即是 NP 完全的问题，弱 NP 完全问题是指那些仅在数字表示为二进制时才是 NP 完全的问题。弱 NP 完全问题在输入数据以单元方式表示的时候可以在多项式时间内获得解。例如：测试一个 n 是否为合数就是一个弱 NP 完全问题：如果 n 以单元表示，则该问题是多项式可解；而如果 n 以二进制表示，则该问题为 NP 完全问题。

另外，在 NP 类中，还存在人们既无法证明其属于 P，又无法证明其属于 NP 完全的“尴尬”问题。例如，因式分解问题。虽然这个问题被公认为难解问题，但到现在也没有人证明该问题是 NP 完全问题。这类问题就被称为**中 NP 完全**（即难度介于 P 和 NP 完全之间）。另一个中 NP 完全的例子是图同构问题（graph isomorphism）。

记住，NP 完全问题并不说明没有多项式时间解，而是说多项式时间解尚未被发现!

思考题

1. NP 完全问题就是很难的问题吗？说明理由。
2. 阐述 NP 难和 NP 完全的区别。并给出一个是 NP 难却不是 NP 完全的问题。你觉得这种区别有意义吗？如果有，意义何在？如果没有，为什么人们会做出此种区别呢？
3. 最长路径问题是在一个图中找出从结点 s 到结点 t 的一条路径，而该路径上的边的权重之和最大。请证明这个问题是 NP 完全问题。
4. 在第 5 章中，我们已经探讨过背包问题，并为其中一个版本给出了贪婪解决方案。其中的“0-1 背包”（每件物品只能取一次）问题使用贪婪策略得不到最优解。这个问题得不到最优解并不是偶然的，因为它是一个 NP 完全问题。请证明“0-1 背包”问题为 NP 完全。
5. 证明顶点覆盖问题的 NP 完全性质。
6. 本章在证明汉密尔顿回路问题为 NP 完全时，选择的规约源是顶点覆盖问题。请证明本章的规约过程是多项式时间内完成的。
7. 证明：三维匹配问题为 NP 完全。设有 n 个男人、n 个女人和 n 条宠物，它们之间的相容关系由一个三元组集合表示。三元组(b, g, p) 表示 b 男人、g 个女人和 p 只宠物相处融洽，而你要做的是找出 n 个互不重叠的三元组，换句话说，三维匹配问题就是在给定的男、女、宠物三元组集合里面找出一个子集，使得每个男人、女人和宠物在子集里只出现 1 次且仅 1 次。

8. 证明：汉密尔顿子图分解问题为 NP 完全。给定图 G 和正整数 k，能否将 G 的结点分解为至多 k 个分离集合以至于由每个分离集合所导出的子图里面包含一个汉密尔顿回路。
9. 某单位领导的一份日常工作是将工作任务分配给三班倒的三个班次。假定他一共有 n 个任务和每个任务需要的时间 $T[1..n]$。假定 $T[1..n]$的元素全部是整数，其中 $T[i]$是完成第 i 个任务所需的时间。该领导必须将每个任务分配到三个班次里的某一个。只有三个班次的工作时间差最小的分配才是合理的分配。例如，如果 n=7，T={1,1,2,3,1,1,3}，将任务 1、2 分配给第 1 班，任务 3、4 分配给第 2 班，任务 5、6、7 分配给第 3 班是不合要求的。正确的分派应该是：第 1、2、3 个任务分配给第 1 班，任务 4、5 分配给第 2 班，任务 6、7 分配给第 3 班，这样三个班次的工作时间差为 0。由于长年累月的工作让领导烦不胜烦，于是他雇用你来设计一个高效的算法来寻找合理的任务分配。请要么证明这个问题为 NP 完全，要么给出一个有效的算法。

CHAPTER 15

第 15 章　无解与近似

据圣经上记载：有一天，耶稣基督在给众人讲道的时候，来了一个年轻人。等耶稣中途停顿的时候，这个年轻人问耶稣：“良善的老师，我需要做什么善事，才能够得永生？”

耶稣回答说：“你为什么称我为良善的老师呢？只有一位是善的。你若要进入天国，就要遵守神的诫命。”年轻人问：“什么诫命？”耶稣回答说：“就是不可以杀人，不可以奸淫，不可以偷盗，不以可作假见证，孝敬自己的父母，像爱自己一样爱邻舍和他人。”年轻人说：“这一切我从小就一直遵守，还缺少什么吗？”耶稣说：“你说的没错，这些你确实做到了，但还有一件事，就是去变卖你的所有家产，分发给穷人，你就会积累财富在天上。然后你跟随我。”

听到这里，年轻人就顿感忧愁，转身离开了耶稣。

因为他的产业很多。

见到年轻人离去，耶稣就对身边的众人说：“我实实在在地告诉你们，富人是很难进天国的。”接着耶稣又说：“我还告诉你们，把骆驼从针眼里穿过，也比让富人进神的国还容易。”如图 15-1 所示。

众人听见这话非常吃惊，忙问：“这样的话，那谁还能得救呢？”

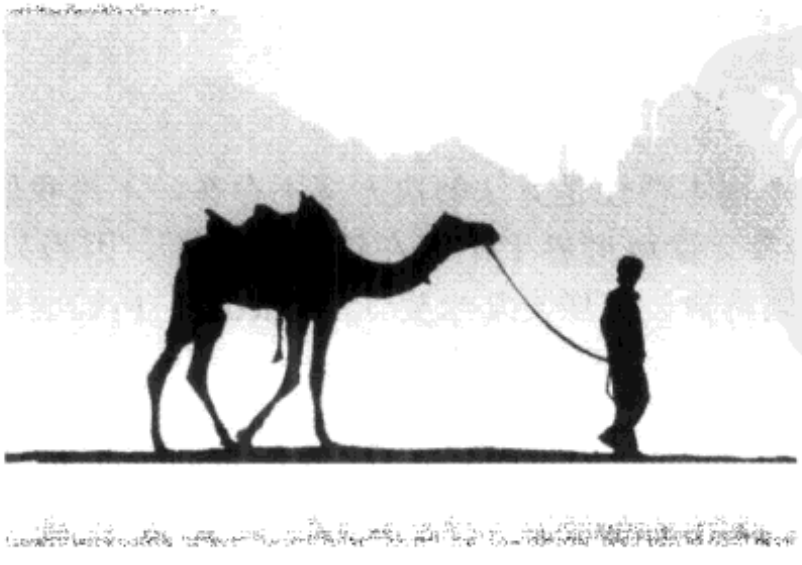


图 15-1　富人进天堂比骆驼穿过针眼还困难。富人是否不可能进入天堂呢

读完这个圣经故事，很多人都会得出结论：富人进不了天堂。因为骆驼是不可能从针眼里面穿过去的，而富人进天堂比骆驼穿针眼还要难，自然是毫无可能了。

如果用算法的术语来描述，就是富人进天堂是一个无解的问题。

但这是真的吗？富人真的进不了天堂吗？这是一个无解的问题吗？

读完本章读者自然会有答案。

15.1 难解问题

经过第 13 章和第 14 章的连续折磨，读者可能会觉得 NP、NP 难和 NP 完全是算法领域中很难解决的问题，甚至有读者会认为 NP 完全问题就是世界上最难的问题了。但实际上，NP 完全问题并不是我们所能碰到的最难的问题！我们需要强调的是，NP 完全只不过是 NP 里最难的问题！世界上比 NP 完全问题更难的问题多得是。事实上，NP 完全问题甚至还不能被标为难解（intractable）问题！因为并没有人证明 NP 完全问题的多项式时间解是不存在的！

到目前为止，从难度上划分，我们遇到的问题可以分为三种类型：

1）易解问题。

2）NP 完全问题。

3）难解问题。

在上述三种类型的问题里，易解问题存在多项式时间解，NP 完全问题到目前尚未找到多项式时间解，而难解问题则不存在多项式时间解。但找不到或不存在多项式时间解，并不说明这些问题没有解！上述三种类型的问题都有解，即都存在算法解决方案！只不过这些算法解决方案的时间或空间效率不怎么样罢了！或者说，只是这些算法没有实际价值而已。从这个角度上说，这些问题都是可决定的问题（decidable problem）。只要给出足够的时间，总能算出解！

但这个世界上还存在一类问题，它们是无解的，即不可能找到任何解，哪怕花费的时间成本是指数级或指数的指数级都无济于事。这些问题被称为不可决定问题（undecidable problem）。

15.2 不可决定问题

不可决定问题的一个根本特性是算法解决方案不存在，不论我们愿意花费多少时间和精力都无济于事。也许读者会觉得世界上不存在这样的问题，因为只要给定无限的时间和空间，似乎什么问题都可以解决。就像阿基米德（Akimedea）所说的，给他一个杠杆和一个支点，他就能把地球撬动！（也许还能把宇宙撬动！）

遗憾的是，没有人能够为阿基米德提供杠杆和支点，而且我们普通人也无阿基米德的豪情壮志，在很多问题面前还真是一筹莫展。不相信？看看下面的问题：

给定一个多变量多项式方程式，如：$x^3yz+2y^4z^2-3xy^5z=18$，是否存在一组变量 x, y, z 的赋值能够满足该方程式？

这是一个决策问题。这个问题的解答是什么呢？没有人知道。也许你会偶然碰到这个问

题的答案，但是你却不能设计出一个算法来计算这个问题的答案！事实上，这个问题不存在任何算法解，即使你考虑指数级、指数的平方、指数的指数或者更为恶劣的算法！

因此，上述问题是一个不可决定问题，因为无法确定性地根据某一方法或步骤来计算解。不可决定问题又可根据潜在证人数量是否有限而分为部分不可决定（partially undecidable）和高度不可决定（highly undecidable）问题。如果一个不可决定问题的潜在证人数量有限（我们只是没有任何办法找出来），则该问题称为**部分不可决定问题**。如果潜在证人数量为无限，则称为**高度不可决定问题**。换句话说，部分不可决定是因为至少其潜在证人的数量是确定的（有限的），而高度不可决定是因为没有任何东西是确定的。

高度不可决定还不是问题难度的最高级。因为它至少存在潜在证人。如果一个问题连证人都不存在，则这个问题就是不可解问题（unsolvable problem）。

这样，我们就获得问题分类的 6 个级别：

- 易解问题：多项式时间解决方案已知。
- NP 完全问题：多项式时间解决方案未知。
- 难解问题：多项式时间解决方案不存在。
- 部分不可决定问题：算法解决方案不存在，但证人数量有限。
- 高度不可决定问题：算法解决方案不存在，且证人数量无限。
- 不可解问题：任何解决方案都不存在。

表 15-1 给出是这 6 类问题的概括。

表 15-1　问题的 6 种分类

问 题 类 型	有 理 论 解	有 实 际 解	潜在证人有限
不可解问题	否	否	没有证人
高度不可决定问题	未知	否	否
部分不可决定问题	未知	否	是
难解问题	是	否	是
NP 完全问题	是	是 / 否	是
易解问题	是	是	是

世界上第 1 个不可决定问题由图灵在 1936 年发现（世界上不可决定的问题那么多，以前怎么就从来没有人发现过），该问题涉及的是程序终结的判断。

15.3　程序终结的判断

假如给你一个程序（当然是用你最喜欢的程序设计语言编写的）和一个特定的输入，能否确定该程序在该输入下是否终结？

显然，这是一个决策问题，而且是一个非常合理的问题。因为，归根结底，一个程序能否终结对我们搞计算机的人来说是很重要的。但不幸的是，这个决策问题没有解，即我们不

能肯定地知道一个程序是否会终结。你不相信？那你对图灵的能力太过轻视了！

假定你是对的，我们能够设计一个算法来判断一个程序在给定输入情况下能否终结。姑且给这个算法起名 TERMINATE (*p*, *x*)（不是电影《终结者》）。该算法的输入是一个内容为待测程序 *p* 的文件（即一个字符串）和一个包含待测程序输入数据 *x* 的另一个文件（也是字符串）。

算法 TERMINATE(*p*, *x*)应该在有限个步骤后告诉我们程序 *p* 是否能在输入 *x* 上终结。

如果上述算法存在，则我们可以设计一个新的邪恶程序如下：

```
EVIL (z: file)
1: if TERMINATE(z, z) goto 1
```

该程序在执行过程中调用算法 TERMINATE 作为子程序。这里请注意，EVIL 程序终结的充要条件是程序 *z* 在将自己作为输入的时候不终结。

然后，我们运行 EVIL(EVIL)。

这个时候问题来了：这个执行过程会终结吗？还是会永远运行下去？

奇怪的是，这两个问题的答案都是“否”。因为程序 EVIL 终结的条件是当且仅当程序 EVIL 在以自己作为输入时不会终结！而这个自相矛盾的结论的得出是因为我们假设存在 TERMINATE(*p*, *x*)这个算法可以告诉我们 *p* 运行在 *x* 上是否终结。

因此，我们唯一的结论是具备这样功能的算法根本就不可能存在。即没有算法可以告诉我们一个程序在给定输入下是否终结。

也许读者会觉得，平时我们细读一个程序的时候经常能够判断一个程序是否终结。因此，我们上述的论断似乎与实际不符？你真觉得实际观察与上述论断矛盾吗？

程序终结无法判定这个结论对于程序设计来说意义重大。就因为这个缘故，程序永远也不会是全自动的，即不可能由程序自己来写程序、启动程序、控制程序。也就是说，像黑客帝国那样的情景永远也不会出现！而这个隐含的推理是程序设计永远也离不开程序员的训练有素、聪明才智和不懈探索。这个结论也同时告诉我们，自动判断一个程序是否存在死循环是不可能的。(但不是很多软件工程研究声称能够检测死循环吗！)

如果上述问题看上去难以理解，那么我们可以看一个似乎更加简单的问题：

```
while (n>1)
    if (odd(n))
      n=3n+1;
    else
      n=n/2;
```

上述程序片段是否能够在 *n* 为任意值时终结呢？

答案是没有人知道。虽然我们试过的 *n* 都终结了，但还是不能肯定任何 *n* 都能终结。

15.4 难解之题的求解

现在我们知道，对于 NP 完全问题来说，有效解（多项式时间解）尚未被发现，而且很

有可能不会被发现。对于难解问题来说，有效解不存在；对于不可决定问题来说，已知无解。那我们应该如何应对呢？是放弃吗，还是知难而上？

人类总是不甘轻易屈服于困难的（这也是人类不同于动物的一个特性）。在实际应用中，我们会遇到大量 NP 完全甚至更难的问题，如果对这些问题我们都放弃的话，恐怕就没有什么问题需要我们来解决了。幸运的是，坚持不懈是被人类推崇的高尚品德。

对于无解的问题，我们只能选择放弃。在无解的问题面前选择继续并不是什么高尚，而是执迷不悟。如果有信仰，将这类问题交给上帝吧（例如，宇宙运行的规律从何而来）。

对于 NP 完全和难解问题，我们就没有必要完全放弃。对于这些问题，我们只是没有找到或找不出最优解。但谁说过什么事情都得最优呢？要是人人都活出最优的人生，恐怕世界就乱了套了。成功人士之所以成功是因为大部分人（至少是很多人）都不成功。要是人人都成功，世界上就没有成功人士了。

因此，找不出最优解没有什么了不起的。我们可以并愿意接受次优解！当然，这个次优解应当越靠近最优解越好。就像人类的恋爱问题，寻找合适的男、女朋友或伴侣似乎是个很难的问题。人们想找的当然是完美无暇的白马王子或白雪公主，这是每个人或大部分人心中的标杆。但恐怕人人都清楚想找到完全符合的并不容易，但也不能证明这个人不存在，这是不是有点像 NP 完全问题（你找不出最优解，但又不能证明没有最优解）？你放弃了吗？当然没有。那你怎么解决这个问题呢？求次优解嘛！在每一次的恋爱中逐步靠近那个标杆。虽然也许永远也达不到那个标杆，但只要努力靠近，也许最终的结局还是可以接受的！

因此，求次优解就是我们应对一切难题的制胜法宝。注意，这里的次优解有三层意思：一是指用来获得答案的算法是次优的，而不是指使用该算法获得的答案本身。例如，通过一次次恋爱也许能找到你的梦中情人（答案最优），但是这种必须一次次亲力亲为的恋爱方式不是最优的！第二层意思是这个答案（解）是次优的，但获得该解的算法则不一定次优，甚至可能很高效。例如，在异性群中随机挑选一个作为情人，这个算法在效率不可谓不迅速，但找到的解（挑选出的情人）可能不是最优的（最令人满意的）。第三层意思当然指的是算法和解都是次优的。例如，通过多次恋爱寻找梦中情人在算法上不是最优，这样找到的情人也难保是最优的。

15.5　智能穷举、近似算法和本地搜索

对于易解问题，我们有各种确定性算法战略予以精确解决。对于难解问题，我们用什么方法来求解呢？如果读者深刻理解了 15.4 节里的“在每一次的恋爱中逐步靠近那个标杆”这句话，就知道了难解之题的所有求解方法。

显然，求解方法根据我们是否要求获得准确解或最优解而不同：如果需要准确解或最优解，则当然需要对解空间进行全面搜索。但因为解空间的巨大，我们不能盲目穷搜，而需要在搜索中使用一点头脑，也就是智能。这样就得出了难解问题的第 1 种求解方法：智能穷举。

如果我们并不奢望获得最优解或准确解，或者我们并不需要最优解，则可以寻求获得次

优解，即找到一个差不多的解即可。这就得出了求解难题的第 2 种方法：近似算法。

如果连差不多的解也不需要，只是有个解即可，则得出第 3 种求解难题的方法：本地搜索。

智能穷举的核心思想有两个：穷举（在最坏情况下要对整个解空间进行指数级搜索）+智能（利用各种途径减少搜索量）。智能穷举策略里使用的最多的是回溯法（backtracking）和分支限界（branch-and-bound），它们分别用于寻找问题的一般解和最优解。对于搜索树中的一个结点 R，如果它已经违反了解的约束，并且我们能预先判定它所有的孩子结点也都不可能是解，此类问题就可以用回溯法进行优化：立即将以 R 为根的整棵子树抛弃（称为“剪枝”）。分支限界是另一种剪枝策略，若能预见以 R 为根的子树产生的所有解比当前已发现的最优解更差，则同样将其“剪枝”，以避免徒劳无益的搜索。

对于最优化问题，只要时间充足，智能穷举总能找到最优解。

近似算法则不然，它仅能保证找到的解与最优解之间的差距“在某个固定范围内”。我们用近似比（approximation ratio）或竞争力（competitive ratio，还记得竞争分析吗）来衡量这个范围，即在最坏情况下近似解的质量能够达到最优解的某一水平。许多近似算法本身并不复杂，但算法的分析往往需要大量的推理与技巧，尤其是需要使用竞争分析。而有的近似算法（如遗传算法）则很难或根本不能进行理论分析，完全是跟着感觉走。

本地搜索则是一种贪婪策略：不断向更优的可行解移动。它既不能保证算法的运行时间，也不能保证解的质量。当算法停止时，算法可能仅仅找到一个“局部最优解”。而为了克服本地搜索可能被限制在一个局部坑洼里面，人们发明了随机化算法和模拟退火（simulated annealing）算法。局部搜索虽然理论性不强，但在人工智能等领域应用广泛。

智能穷举、近似算法和本地搜索这三种方法都在恋爱问题里面得到了充分展示（见图 15-2）：对于那些一定要找到梦幻情人的人来说，他通常会使用智能穷举，即在茫茫人海中使用某种智能的判断进行搜索，对于那些一看就不顺眼的人种或社会阶层统统“剪枝”；对于那些只要求找个差不多的伴侣的人来说，他多半会使用近似算法来进行搜索，即将标准降低一个或几个数量级；对于那些只要是异性即可（假定该人为异性恋）的人来说，使用本地搜索最好，即先找一个身边的人，随着自己的搬迁走动，碰到新的人后，再在身边寻找更好的人。

图 15-2 求解难题的所有方法都在恋爱过程中得当了充分展示

智能穷举和本地搜索都发展出了一定的策略，而近似算法则基本上依赖人的直觉和天赋。

15.6　智能穷举之回溯策略

与其说回溯是一种算法，不如说它是一种策略，它是智能穷举的一种，用于寻找问题的一般解和最优解。其思想是抛弃那些不符合约束的解空间，从而降低搜索空间大小。该方法可以灵活运用到不同场景中。在实际运用中，它与深度优先搜索有着很自然的依存关系。

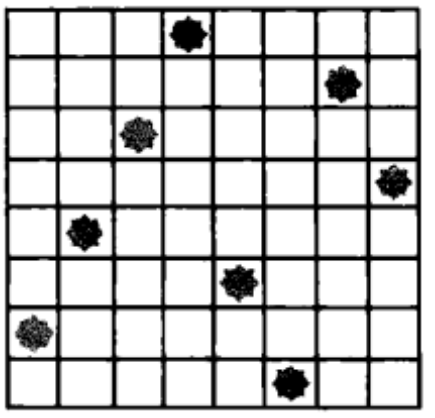


图 15-3　八皇后问题一个解

我们下面以经典的八皇后问题作为例子来对此种策略加以说明（见图 15-3）。下过国际象棋的人都知道，皇后为权力最大的棋子：她可以横向、竖向、斜向行走，并且可以吃掉任何处于这三个方向上的棋子。我们的问题是：在一个 8×8 的棋盘上放置 8 个皇后，有多少种不同的放置方法能够使得任意两个皇后间不相互攻击，即不在同一行、同一列或同一对角线上？

天真的解法是直接使用深度优先搜索。

```
EIGHT-QUEEN-DFS-SOLUTION (k)          //尝试在第 k 行放置皇后
if(k>8){
     if (当前解满足八皇后约束条件)
          counter=counter+1;
     return;
}
for(i=1;i<=8;i++) {                   //穷举第 k 行的 1 到 8 列
     row[k]=i;                        //记录第 k 行皇后所处的列位置
     EIGHT-QUEEN-DFS-SOLUTION (k+1);
}
```

初始化：counter=0; EIGHT-QUEEN-DFS-SOLUTION(1)

EIGHT-QUEEN-DFS-SOLUTION 算法的正确性可以由深度搜索的特性得出。但该算法的效率却很低，因为每次都是在 8 个皇后都放置后才检查是否有冲突。这将导致大量的无用功。

如果使用回溯策略，在每放置一个皇后即进行冲突检查，则可以大大改善算法的效率。

```
BACKTRACKING-EIGHT-QUEEN-DFS-SOLUTION (k)  //尝试在第 k 行放置皇后
if (k>8) {
     counter=counter+1;
     return;
```

```
}
for(i=1; i<=8; i++) {                                    //穷举第 k 行 1 到 8 列
    if(在第 k 行第 i 列放置皇后不会与已放置好的皇后冲突) {
        row[k]=i;                                        //第 k 行皇后所处列位置
        BACKTRACKING-EIGHT-QUEEN-DFS-SOLUTION (k+1);
    }
}
```

BACKTRACKING-EIGHT-QUEEN-DFS-SOLUTION 算法与前面一个算法的不同仅仅是将对解的正确性判断从放完所有皇后之后移到尝试放一个皇后（结点扩展）之前。但就是这点不同使得算法效率得到显著提高，EIGHT-QUEEN-DFS-SOLUTION 算法需要考虑 8^8=16 777 216 种可能的放置方式，而后面使用回溯的算法考虑的可能放置方式只有 5 508 种！

但 BACKTRACKING-EIGHT-QUEEN-DFS-SOLUTION 算法还有改进的余地：对于第一行的皇后，只需考虑第 1～4 列即可，因为第 5～8 列的情况完全对称。最后再将结果乘以 2。这样，我们又将搜索树砍掉了一半。在组合问题中经常可以利用这样的对称性。

15.7 智能穷举之分支限界

分支限界是回溯策略的一种改进，当你在搜索的时候，需要记录该搜索当前累积的成本。当该累积成本超过某个给定的值后，即不再往前搜索，而是退回去再搜索另外一条路径。而这个给定值既可以是事先定下的，也可以是在搜索过程中自我产生的。例如，在使用穷举法解决单源单点最短路径问题时，我们将一个图的直径（一个图里面相距最遥远的两个结点的距离）作为这个给定值。在搜索过程中，如果当前所经历的部分路径的累积长度超过此给定值，则当前的搜索方向肯定不会获得最优解，这样就可以终结当前搜索，而另寻一条搜索路径。

如果一个问题没有什么明显的限界可以使用，则可以在搜索过程中动态地产生一个。例如，在单源单点最短路径问题上，当我们获得一个从源点到目标的路径时，我们记录该路径的长度成本。然后在后续的搜索中，只要累积成本达到记录的这个值的时候，就无需再往前搜索。因为该解决方案的成本将肯定高于我们已经获得的解决方案。这样就可以将搜索空间大为减少。而且，分支限界可以在算法运行的过程中动态调整。例如，在单源单点最短路径问题上，每当获得一个更短的从源点到目标的路径后，就用这个新的值取代原来的限界值。

15.8 贪婪近似策略

本书在第 2 章详细讨论了贪婪战略及其应用。如果一个问题具备最优子结构和贪婪选择属性，则贪婪策略将可以低成本获得最优解。但是，如果一个问题不具备贪婪选择属性，并不是说贪婪策略就不能使用了。实际上，贪婪策略在任何问题上都可以使用，只不过我们不

一定获得最优解。但如果我们一开始并不奢望获得最优解，而只是一个近优解，则使用贪婪策略将大大节省算法的时间成本。而对于一个难解问题来说，反正也不太可能获得最优解，此时使用贪婪策略也就未尝不可了。

例如，集合覆盖问题是一个难解的问题，但我们可以使用贪婪策略取得一个接近最优的结果。集合覆盖问题的定义是：给定一个集合 B 和一组 B 的子集合 $S_1, S_2, \cdots, S_m$；如何找出该组子集合里的一组数量最小的子集，使得 B 的所有元素均包括在选取的这组子集里面。

该问题的最优方案并不容易获得，但用贪婪算法可以获得一个接近最优的解决方案：**每次选取包含最多尚未被覆盖元素的子集，一直到集合 B 所有的元素均被覆盖为止。**

我们可以容易地证明，此种策略所选取的子集个数最多不会超过最优结果的 $\ln n$ 倍。这里证明的关键是意识到：如果在 t 轮子集选取后尚未被覆盖的元素个数为 n，则下一轮（t+1 轮）后尚未被覆盖的元素个数将小于等于 $n-n/k$，这里 k 表示最优结果所选取的子集数。这是因为剩下的所有子集里存在一个覆盖至少 n/k 个元素的子集，从而我们的第 t+1 轮选取的子集将至少包括 n/k 个元素。因此，第 t+1 轮后，剩下的元素将被减少至少 n/k。经过简单的推导可知，在 $t=k\ln n$ 后，所有元素将被覆盖。因此，$\ln n$ 是贪婪算法与最优算法的最大差距。当然，随着输入的不同，这个差距会有所不同，但最坏不会超过这个差距，这个差距就是该近似算法的近似因子或竞争力。

本书第 9 章就是对近似算法进行的论述，因此，对近似算法感兴趣的读者请参看本书第 9 章，也可以参看参考文献[1]。

15.9　启发式搜索策略

智能穷举虽然有着“剪枝”的智能，但在选择下一个搜索点的时候通常是随机的。启发式搜索，顾名思义，就是在搜索的时候并不是随机选择一条路径来推进。而是根据某些知识（如某个函数的取值）来推断应该前进的方向。当面临一系列选择的时候，根据某个函数（称为启发函数）对每个选项进行估值，然后根据估值情况决定下一步行动。

2009 年 7 月 16 日早上 8 点 40 分，我下楼来到小区的停车棚，发现我借用他人的电瓶车已不翼而飞。由于我是前一天（15 日）中午 12 时左右将电瓶车放置于停车棚的，电瓶车被偷的时间只能是在 7 月 15 日中午 12 点至 7 月 16 日早上 8 点 40 分这个区间。

我马上想到调看小区录像来查看这段时间小区里面，尤其是大门的情况。小区多处装有摄像头，共计 8 处，其中地面 5 处，地下停车场 3 处。地面的 5 处中的 3 处分别是小区的 3 个出入口（大门），另外 2 处为小区内部道路。如图 15-4 中 A、B、C、D、E 所示。

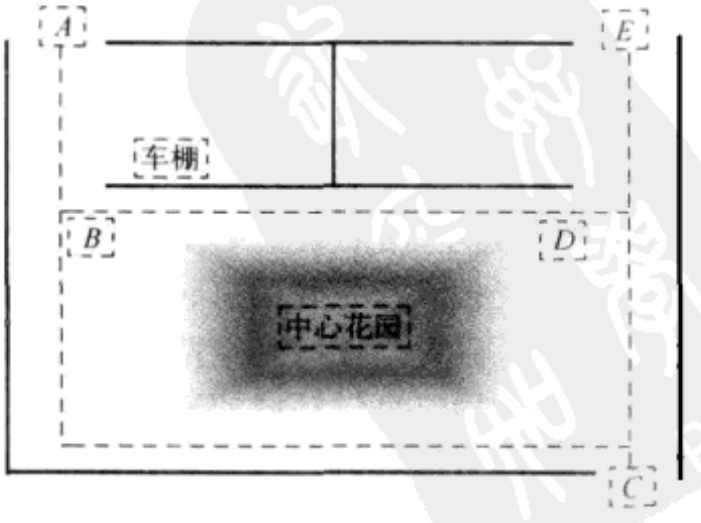


图 15-4　小区结构简化图：虚线为小区道路

7 月 17 日上午，小区保安告知我他们已经查看了所有录像，未发现电瓶车被偷的任何视频。我问他们查看了何处录像，保安说是三个大门的录像。由于小区业主都是比较富有的人士，偷小小电瓶车的概率不大。因此，偷车人很可能是外面混进来的人士。他们偷车之后肯定需要将电瓶车运出小区。因此，3 个大门是其必经之处。

因此，保安查看 3 个大门的录像似乎是非常合理的。

我当然对这个答案很不满意，于是要求亲自查看录像。经过各种手续之后，保安将我带到监控室查看录像。从 15 日中午到 16 日早上，每一处录像的视频超过 20 小时，地面 5 处录像加在一起有超过 100 小时的视频。显然，如果使用穷搜，5 处每个时段的录像都要查看，将非常的费时费力。即使快放，也要费时至少 1 天 1 夜！一个难解问题！

因此，某种启发式的搜索是我所迫切需要的。那么查看哪处的录像和什么时段的路线呢？

首先，偷电瓶车人不太可能在白天作案。因为光天化日，人来人往，且电瓶车主人随时有可能来用电瓶车，此时下手风险相当大。但是深夜作案也不方便，因为这个时候小区夜阑人静，小偷过大门保安时非常容易引起注意。因此最有可能作案的时间段应该是傍晚到晚上的时间。此时光线昏暗，能见度较低，但是小区人来人往，比较容易混出去。基于此种分析，我将查看录像的时间段定在晚上 6～10 点。

查看哪一处的录像呢？由于 *A* 门离电瓶车最近，保安认为走 *A* 门的可能性最大，推荐查看 *A* 门录像。但我认为，正因为 *A* 门离电瓶车停车位很近，而且离电瓶车主人的居所很近（小区有多个停车棚，业主当然是将车停在离自己居所最近的停车棚里），所以从这个门出去有相当的风险。再说 *A* 门出去是一个广场，而广场上还有广场保安。

如果不走 *A* 门，偷车人只能走 *C* 或 *E* 两个门。但查看两个门的录像视频还是太费时间。能否只查看一个录像视频呢？答案是“能”。因为，无论是 *C* 门还是 *E* 门，偷车人必先经过 *B* 摄像头所监控的道路。因此，只要检查 *B* 处录像视频即可。于是保安调出 7 月 15 日 *B* 处录像，从傍晚 6:00 点开始重放，当看到时间为 8 点 2 分时，我终于发现偷车人推着电瓶车向 *B* 镜头的方向走来。接着马上调看 *C* 门 8 点 2 分后面的录像，果然又在 8 点 3 分 37 秒记录着偷车人骑着偷来的电瓶车从小区 *C* 门出去。

我花了 1 个小时（查看录像时进行了快放）就找到了需要的证据，而多个保安人员查看了一天也没有发现。这种有引导的搜索就是启发式搜索！读者看出这个启发式函数了吗？

其实，在这个破解电瓶车失窃的事件时，使用的并不只有启发式搜索一种策略。读者能否看出其他被使用的策略是什么呢？

启发式搜索在生物信息领域的应用非常广泛。由于生物信息技术领域的许多个体问题的输入数据量巨大，任何确定性的算法都将面临解空间规模爆炸的痛苦困境。因此，启发式算法就成了生物信息学领域的不二选择。这些算法在平均情况下的运行速度还是可以的，并且在很多时候能够找到一个与最优解距离不超过$(1+\varepsilon)$的近优解，这里 ε 是一个合理的较小的值。但是，启发式算法存在一个严重缺陷：常常无法在数学上进行精确的分析。

15.10　模拟退火算法

在一些古装电影中，时不时地会出现这样的镜头（见图 15-5）：某个铸剑高人挥汗如雨地用锤击打一块烧红的形似刀剑的铁块，在捶打到差不多的时候，只见铸剑高人猛然一挥，将红红的剑身插入寒冷的冰堆中，宝剑顿时散发出神秘的光芒，一把旷世奇剑就这样诞生了……

这种将烧红的剑插入水中或冰中的做法就是退火。它要达到的目的很简单：获得最大强度。当然，真正的退火是将剑插入水中，而不是冰中。只不过电影为了渲染，才会插入冰中。

图 15-5　铸剑过程的退火为的是让宝剑达到最大强度和韧度

在物理学中，对固体物质进行退火是将其加热到临界温度以上某一温度，此时物体中的粒子能够自由运动。在保温一段时间，待物体全部软化后，以大于临界冷却速度的冷速对物体进行快速制冷。随着物质温度的下降，粒子形成了低能态的晶格。此时，如果在凝结点附近的温度下降速度足够慢（这就是为什么宝剑插入水中的缘故），则固体物质一定会形成最低能量的基态，从而达到最优的强度或韧度。

这种退火的过程对于求解难解的优化问题提供了启示。难解优化问题之所以难解通常是因为解空间巨大，常规的搜索方法将产生指数级时间成本的算法。但如果我们能够像退火那样进行搜索，先以某种办法迅速降低搜索空间（迅速冷却），然后在接近最优解的地方慢慢搜索（在凝结点附近慢慢地降低温度），那么获得最优或近优解的成本可能就大大减少了。

这种将物理退火过程引入算法领域而形成的优化问题的近似算法就称之为**模拟退火算法**（simulated annealing）。该算法不是一个特定的具体算法，而是一种通用的概率性的启示算法思想：在一个巨大的搜索空间中寻找全局最优。此种算法尤其适用于离散空间的情况。如果我们寻求的是一个可以接受的解，而不是绝对的最优解，则模拟退火算法可能大有用处。

模拟退火算法由库克帕特里克（Kirkpatrick）、格拉特（Gelatt）和纬奇（Vecchi）等人于 1983 年提出。它是梅特波利斯-黑丝汀（Metropolis-Hastings）算法在算法领域的应用，该算法由美国物理学家尼古拉斯·梅特波利斯（Nicholas Metropolis）在 1953 年发明，是一个用来产生热力学系统样本空间的蒙特卡洛算法。

15.10.1 模拟退火算法的思想

由前所述，退火算法的核心思想是先迅速靠近最优解，然后再慢慢搜索。问题是如何迅速靠近最优解呢？如果我们知道最优解在哪儿，自然可以一步登天地靠近它。事实上，这种情况下，也不用费周折去靠近了，干脆一步到位定点在最优解上，自然也用不着什么退火或者别的搜索方法了。我们之所以要退火，是因为我们并不知道最优解在什么地方。而既然不知道最优解在什么地方，那又如何迅速靠近呢？

答案当然是随机化。我们随机选择一些点，然后对这些点进行检查，看看每次随机是否导致解的逐步变优。如果是，则这种随机在向着正确方向前进，即所谓的“退火正常”；否则，就是随机的方向发生错误，此时反转随机方向即可。当然，如果一个问题的最优解不呈现聚拢或者收敛效应，即近优解在搜索空间上并不靠近最优解，各个近优解之间在空间上也不相互靠近，则退火算法将难以达到寻找到最优或近优结果的目标。

由于逼近最优点的方式是随机的，这种操作并不是真正的退火，因为物理界真正的退火不是随机冷却到临界温度的。这也就是为什么这种算法被称为“模拟退火”。

在模拟退火的实现上，我们用一个全局参数 T 来表示问题当前的温度（模拟退火嘛），模拟退火算法的每一步都随机选择一个离当前已发现的解决方案很近的解决方案来替换当前的解决方案，即施行本地搜索。而这个随机选择并不是完全随机的，而是依赖于相应函数值与全局温度 T 的差值。这里 T 的值在整个算法运行过程中是逐渐减少的（温度冷却）。在 T 很大的时候，当前解决方案的改变几乎是完全随机的；但随着 T 的值趋向于 0，解决方案的改变呈现出下坡模式。为了避免解决方案停留在局部最优里面，在需要的时候也可以进行上坡操作。

在模拟退火方法里，搜索空间的每个点都类似某个物理系统的一个状态，而需要极小化的函数则类似于该物理系统在此状态下的内部能量。我们的目标则是将系统从任意初始状态转换到一个能量最低的状态。

15.10.2 模拟退火算法的基本循环

在算法每一步，模拟退火的启发式函数考虑的是当前状态 s 的某个邻近状态 s'，然后概率性地决定是否将系统状态改变到 s'（另一个选择是停留在状态 s）。概率的选择应该导致系统最终趋向转换为一个能量更低的状态。通常来说，这个步骤需要一直重复到系统的状态改变到应用或用户能够接受的地步，或者计算资源的消耗达到某个极限的时候为止，如图 15-6 所示（注意，图 15-6 里面的“温度”就是退火过程所使用的控制温度）。

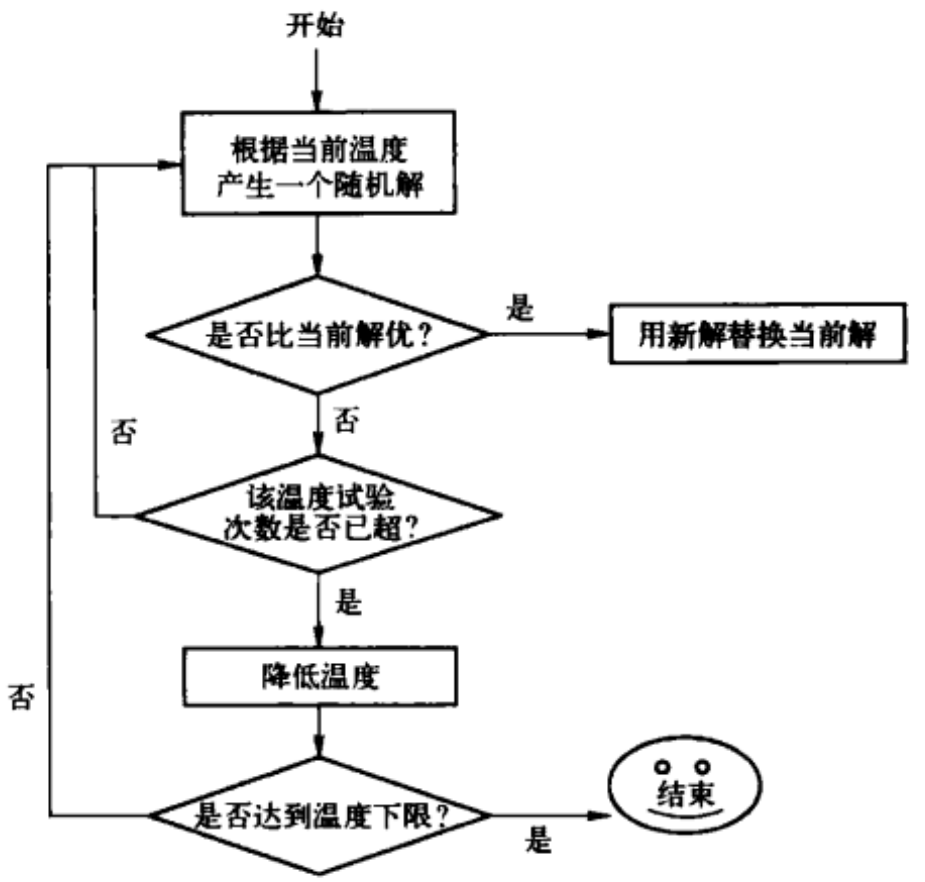


图 15-6　模拟退火算法的流程图

每个状态的邻近状态（候选迁移状态）由用户注明，通常与具体应用相关。例如，对于旅行售货员问题，每个状态通常被定义为一个具体的周游（被访城市的一个排列）。这样，一种可能的邻近状态定义可以是任何通过交换两个连续的城市对所能达到的状态。

15.10.3　退火算法描述

设系统所有可能状态 $S=\{s_1,s_2,\cdots,s_n\}$，与系统状态相关的能量函数为 E，控制参数为温度 T，我们的目标是找到某一个系统状态 s^*,使 $E(s^*)=\min(s_i,s_i\in S)$。

模拟退火的目标是让 T 从一个足够高（但不要太高）的值慢慢下降，对每个 T，用抽样法在计算机上模拟该系统在此 T 下的热平衡状态，即对当前状态 s_i 进行随机扰动产生一个新状态 s_j，然后计算系统的能量增加值 $\triangle E=E(s_j)-E(s_i)$，并以概率 $e^{(-\triangle E/T)}$ 接受 s_j 作为当前状态。

```
SIMULATED-ANNEALING (E, T, s0)
1. si=s0;
2. 产生一随机扰动△s,计算△E=E(si+△s)-E(si);
3. 如果△E<0 则跳转到第 6 步;
4. 否则 ς=RANDOM(0,1);
5. 如果 e^(-△E/T) ≤ς，则跳转到第 2 步;
6. si=si+△s;
7. E=E+△E;
```

8．在该 T 下，检验系统是否稳定，若不稳定则跳转到第 2 步；
9．以某一方式降低 T 的值；
10．退火过程是否结束（结束条件见随后的讨论）？如未结束则跳转到第 2 步；

注意，在上述过程中，模拟退火过程是否达到能量 E 的最小值，取决于 T 的初始值是否足够高和 T 下降得是否充分慢以及对应每个 T 的系统是否稳定。而选取 T 的初始值、检验系统是否稳定、降低 T 值和算法终止的方法和方式分别如下：

1）T 的初始值选取有三种方式

- 均匀地随机抽样 v，取 $E(v)$的方差为 T 的初值。
- 在所有可能状态中，选取 v_i和 v_j，使$\triangle E=|E(v_i)-E(v_j)|$最大，设 T 为该值的某一倍数。
- 由经验给出。

2）检验系统是否稳定的方法

- 检验 E 的均值是否稳定。
- 检查是否连续若干步中 E 的变化都比较小。
- 按某一固定步长进行抽样。

3）T 值降低方式：令 $T=aT,a\in[0.8,0.99]$。

4）算法终止方式

- 取 T 小于某一阈值。
- 检验系统的熵是否已达最小。

这里需要提请注意的是，上面介绍的各种方法或方式只是人们根据长期实践观察后进行的推荐，并不是只能如此或者是唯一的可行方法。

15.11 基因/遗传算法

本章前面介绍的各种求解难解问题的算法均有一个共同点：都使用了某种规则，即按一定规则来进行求解或近似。例如，模拟退火算法就是按照退火的规则来向最优解靠近。这些算法的效用有一个前提，我们大概知道最优解在什么位置，或至少知道最优解有什么形态。

有的问题可能很难，以至于没有任何规则可循。例如，对于有的问题，最优解是什么都可能无法知道，那么接近最优多半就没有什么规则。在这种情况下，剩下的唯一武器是随机。基于此种思维，美国密歇根大学（University of Michigan-Ann Arbor）的约翰·荷兰（John Holland）教授提出了充斥着随机因素的基因算法（genetic algorithm）。

基因算法通过借鉴生物界的进化和遗传现象来进行随机化搜索。在此种算法里，解被划分为一代代的父、子、孙辈等。每一代的解都是由上一代的解经过某种类似生物繁衍的操作而获得。由于基因是遗传的基本物质，因此基因算法也被称为遗传算法。约翰·荷兰于 1975 年出版了其著作《*Adaptation in Natural and Artificial Systems*》（自然与人工系统的自适应性），约翰在该书里提出了一套用来预测下一代解的质量的形式化框架，从而开始了遗传算法的纪元。

15.11.1　生物进化与遗传

遗传算法的核心是模拟生物种代之间的进化与遗传过程。按照伟大的查尔斯·达尔文（Charles Darwin）的推测，生物进化的规律是适者生存，优胜劣汰。物种经过自然界的选择后留下优势个体，只有这些优势个体才有资格和机会繁衍产生后代。虽然个别情况下，劣势个体也会产生后代，但他们后代的质量很低，很难经得起自然的残酷选择，以至于终将灭亡。因此，从总体上看，只有优势个体才能留下后代继续繁衍生息。格里高·孟德尔（Gregor Mendel）又发现，生物繁衍的后代保留其父或母或二者的基因。这种保留父母基因的机制就是所谓的遗传机制。根据孟德尔观察，父体和母体的质量越高，所产下的后代质量就越高。

生物进化论的观点认为：对于那些生生不息的生物物种来说，其后代的质量通常高于父母的质量，否则一代不如一代的话就一定会毁灭，而不会生生不息；而如果一代好于一代，这样一代代繁衍下去，总有一天个体的质量会达到一个令人“十分满意”的最优境界。

这就是遗传算法的精华。

模仿生物界的行为，遗传算法将问题的解看做是独立的个体生物，每一代解经过自然选择的优胜劣汰而留下质量高的解。这些质量高的解通过生物繁衍过程产生新一代的解。由于每次用来繁衍下一代的个体都是经过自然选择的高质量解，因此一代代繁衍下去，解的质量越来越高，直到达到最优解。当然，有时候天捉弄人，一代代繁衍下去并不产生最优解，而繁衍的成本却已经令人无法忍受，此时我们也会停止求解。

15.11.2　遗传算法的基本要义

从对生物界的进化过程描述，我们可以推测出遗传算法的三大要素：

1）有一个初始种群。

2）有自然选择的方法。

3）有繁衍后代的方式。

显然，对于生物进化来说，得有一个初始种群的存在，从这个种群开始繁衍后代。对于遗传算法来说，就是得有一组初始解。当然，这些初始解通常不令人满意，就像猿猴作为人的初始种群是难以令人满意的。（如果初始解令人满意，恭喜你，无需进化，我们已一步登天！）

其次，需要一种办法对生物进行选择，优胜劣汰。对于生物物种来选，这种选择是自然选择。对于遗传算法来说，显然不可能让自然来对解进行选择，只能是我们人来选择。但选择得有某种标准，以这个标准对每个个体进行评估，选择评估分高的个体作为繁衍后代的优势个体，其他个体则被消灭。

在进行了“自然选择”后，留下的优势个体就要繁衍后代。生物繁衍的方法可分为无性和有性。在遗传算法中，产生后代解的方法也分为有性和无性。有性就是两个或多个个体进行交配而产生后代，无性就是由一个个体就可以产生后代。

上述过程循环往复，直到出现最优解或令人满意的近优解，或者运算成本达到某个极限为止。例如，在任何时候，如果某个个体的分足够高，则直接以该个体作为整个问题的解，结束算法；如果没有任何个体解的质量令人满意，则继续“选择—繁衍—选择—繁衍”的无穷循环。

15.11.3 遗传算法的实现

从 15.11.2 节对遗传算法的要义讨论中可以发现，要实现遗传算法需要解决如下几个问题：

1）解的表示。
2）初始解的生成。
3）解的评估。
4）解的自然选择。
5）解的繁衍。
6）算法的终止条件。

1. 解的表示

在遗传算法中，解被看做是独立的个体，我们对它们进行选择，用它们进行繁衍。因此解的表示必须能够让我们对不同的解进行评估比较，并能够对解进行繁衍操作。这就要求解的表示具备某种统一性和可操作性。

如果读者仔细阅读了本书前面的章节，应该知道本书隐含的“万物为数”的观念。任何事物都可以用数，而且是整数来表示。如果问题的解表示为数，则进行比较、评估、选择就是一件很容易的事情。但问题是，如何将一个具体问题的解转换为一个具体的数呢？这个问题的回答当然依赖于具体问题的特点，因此不能一概而论。但不管如何转换，这种转换需要满足一些条件。首先是完备性，即问题的所有解都可以转换为数；其次是唯一性，即不同的解转换出来的数是不同的；其次是健全性，即这些数进行操作的结果仍然代表问题空间的解。

例如，对于第 5 章讨论过的 0-1 背包问题，每个解可以表示为一组二进制码，每个字位的 0 和 1 分别表示对应宝物的取与舍。因此，0101010 的解代表取第 2、4、6 件宝物，第 1、3、5、7 件宝物不取。显然，在此种表示法里，所有的解都可以表示为一个二进制码，不同的解表示为不同的码，而每个码也对应某个解。

2. 初始解的生成

生物进化的基础是存在初始的生物，遗传算法的基础是存在初始的解。初始生物的出现不需要人来操心，但初始解的出现却需要我们来负责。那么如何生成初始解群呢？答案是：随机。这也是模仿生物出现的随机性（达尔文认为生物的出现是完全随机的）。例如，对于背包问题来说，我们可以通过抛硬币来决定每个字位的取值，这样可以产生一系列随机初始解，如 0101010、0101001、1010100 等。

生成初始解时，需要注意一个问题：这些随机产生的初始解应该数量足够多，使得其可能产生的后代能够覆盖最优解。为了保证这一点，我们有时也使用非随机的手段来进行初始

解的生成。例如，我们可以根据问题的固有特性，设法把握最优解所占空间在整个问题空间中的分布范围，然后，在此分布范围内设定初始群体。对于背包问题来说，如果我们知道宝物 3 的价值远远大于其他所有宝物的价值，则我们的随机解选择将围绕在宝物 3 被选中的所有随机解上进行。即在抛硬币的时候，第 3 次抛硬币的结果总是为 1。

3. 解的评估

有了初始解群，下面的操作就是自然选择了，而选择的前提是对解进行评估。这个评估值应该反映解的质量，质量越高，评估值越高。在遗传算法中，这个评估值称为适应值，从解生成适应值的映射就是适应值函数，或评估函数。这个函数通常与所求问题的目标函数一致或者是从该函数演变而来。例如，对于背包问题，目标函数是取得不超过背包载重的最大价值的宝物集合，这个函数就可以作为背包问题解的适应值函数。

由于适应值函数需要应用到每个个体解上，其效率高低直接影响到遗传算法的性能。

4. 解的自然选择

在生成每个解的适应值后，就可以进行自然选择了。常见的选择方法包括局部选择法、适应值比例选择算法和随机遍历抽样法。

局部选择算法将适应值在某个阈值以下的解全部删除，也就是依次选择适应值高的解保留下来。该算法非常简单，也最符合优胜劣汰的原则。但由于存活个体的多样性不足，可能导致算法过早收敛于一个较差的解上。

一种改善存活个体多样性的方法是不要如此粗鲁地将适应值低的个体删除，而是也给它们一点存活的机会。生物进化论告诉我们，在生物进化过程中，整个群体中的一定比例的个体被选择出来，用来繁衍新一代。这种选择并不总是选择最优势的个体，有时候因为特定环境或其他原因（如适应值高的个体正好在一个火山爆发的时候待在火山旁边，而适应值低的个体没有在火山旁边），一些适应值低的个体被选择出来。基于这种原因，在选择的时候我们通常采取概率选择：让质量高的解被选中的概率高，质量低的被选中的概率低。低适应值的解被选中可以保证后代群体的多样性，防止过早收敛于较差的解上。

这样，我们就获得另一种选择算法：适应值比例选择算法。在该方法中，各个个体的选择概率和其适应度值成比例。设群体大小为 n，其中个体 i 的适应度为 a_i，则 i 被选择的概率 $a_i \Big/ \sum_{j=1}^{n} aj$。该算法由于与赌场里面轮盘赌所用到的算法相同，也被称为轮盘赌算法（看来赌博对研究算法很重要，还记得散列中的赌徒原理吗）。

适应值比例选择算法的优点是逻辑简单，适应值低的个体也有存活的机会，从而可以保证存活个体的多样性，增加获得最优解的可能。缺点是计算量较大，需要计算所有个体的选择概率。一种降低计算量的方法是随机遍历抽样法。该算法通过多轮随机比较选择来决定存活的个体：将所有的个体随机分成不同的小组，每个小组中适应值高的解被保留，再进入下一轮随机分组比较。在随机比较一定次数后的优胜者脱颖而出，作为繁衍后代的备选解。

细心的读者可能发现，此种算法模拟的是球类比赛时的循环赛。因此，此种算法也称为

循环赛选择法，此种算法的效率很高，实际中常被采用。

对于背包问题来说，这三种选择算法都可以使用。

5. 解的后代繁衍

选择了备选解后，就可以进行后代繁衍了。如何繁衍后代呢？生物学知识告诉我们，繁衍后代有无性和有性两种方法。无性就是通过对一个个体解进行某种变化产生后代，有性就是对两个或两个以上的解进行某种交配产生后代。

无性繁衍后代的方法有两种：复制和变异。有性繁衍方法也有两种：交配和组配。

复制

复制是将自身复制下来，即新的个体与原来的个体完成相同。对于遗传算法来说，这种繁衍相当于将当前解不加修饰地传递到下一代。

变异

变异是在单一个体上进行某种修改获得新的个体。例如，我们将某个字位的值进行翻转而获得一个新个体。变异的方式很多，凡是能够按某种规律对某些字位值进行变化的方法都可以用来进行变异。一般来说，变异操作有两个步骤：首先对所有个体以事先设定的概率判断是否进行变异，然后对需要变异的个体随机选择字位进行变异。

变异带给遗传算法的功效是提供局部的随机搜索能力。当遗传算法已接近最优解时，利用变异这种局部随机搜索能力可以进一步向最优解收敛。

交配

交配是指把两个父代个体的部分结构加以替换重组而生成新个体，因此交配也被称为重组。由于基因重组在生物进化过程中起核心作用，交配操作在遗传算法中也起着核心作用。对于普通的交配操作来说，根据某种交配率将一对对的“父母”解避选出来，然后按照事先想好的排列组合产生下一代，期望将有益的基因组合在一起。通常的交配方法包括实值重组和二分交配。实值重组对父母的基因进行某种组合，如前三个字位来自父体的前三个字位，后四个字位来自母体的后四个字位。当然这种组合有无数方式。

二分交配则需要对父母的基因进行运算，也就是真正的交配。如将父母某个字位或整体进行与、或、异或操作而获得新的个体。常见的交配方式有单点交配（对某个字位进行运算）、多点交配（对多个字位进行运算）和均匀交配（对所有字位进行运算）。当然也可以进行进一步的变化，如洗牌交配（对父母字位进行打乱后进行运算）和缩小代理交配（将父母按某种比例缩小后再进行上述的交配操作）。

组配

组配是交配的扩展，将父母两个个体的交配扩展到多个雌雄个体的群交。组配也可以有实值重组和多分交配。例如，对于组配中的实值重组，下一代解的前 2 位可以取自父体 A，接下来 2 位取自父体 B，接下来 2 位取自母体 C，最后面的字位全部取自母体 D 等。组配纯粹是遗传算法本身的发明，在生物界似乎并不存在（至少人类是反对群交的）。

交配和组配操作带给遗传算法的功效是在全局范围内快速地扫描逼近最优解。

遗传算法中，交配和组配操作因其全局搜索能力而作为主要操作，变异因其局部搜索能

力而作为辅助操作。好的遗传算法通过交配、组配、变异、复制的合理组合而使其具备兼顾全局和局部的均衡、快速搜索能力。

对于背包问题来说，上述四种繁衍算法都可以使用。

6. 终止条件

遗传算法中的后代繁衍到什么时候可以结束呢？当然是找到了最优解的时候。不管是复制、交配、组配还是变异，新一代解通常保留父母解的很多特性（父母的很多字位保留在新个体里），这也是遗传算法名称的由来。一般来说，由于每次用于产生后代的父母解是一个解群里面较优的解，新一代的解的适应值通常高于父辈解的适应值。这样循环一定次数后，适应值将达到一个令人满意的状态，此时最优解或者近优解已经找出，算法结束。

有时候，我们也许并不需要最优解，而是一个能满足最低要求的解即可。有时候，在还未找到最优解的时候，运算的成本可能已经超过了我们能够承受的能力，此时也应该结束算法。一般来说，终止条件包括如下几种情况：

- 最优解被找到。
- 满足某种最低条件的解被找到。
- 繁衍的代数达到某个阈值。
- 进一步的繁衍所产生的后代的质量没有再提高。

例如，对于背包问题来说，我们可以设定所拿宝物的总价值超过某个心理预期时结束算法。

15.11.4　遗传算法的基本运算过程

1）初始化：选择一个最初的、解数量足够多的解群。

2）对解群中每个个体解，计算其适应值。

3）重复下述循环直道终止条件满足：

a）如果当前解群中存在满足条件的解，结束算法。

b）否则按照某种选择算法选择足够数量的个体作为繁衍后代的解。

c）通过复制、交配、组配、变异等运算生成新的解。

下面以 0-1 背包问题来展示遗传算法的使用。假定我们有一个总承重为 50 千克的背包，山洞里有 5 件宝物 1、2、3、4、5，其重量和价值如表 15-2 所示。

表 15-2　宝库里面各种宝物的价值和重量

i	1	2	3	4	5
p_i（元）	60	100	120	30	40
w_i（千克）	10	20	30	10	20
p_i/w_i（元/千克）	6	5	4	3	2

解的表示：根据前面的分析，背包问题的解可以用一个五位二进制数表示。

初始解生成：随机生成 5 个解：s_1=01001，s_2=10100，s_3=00110，s_4=11111，s_5=01110。

解的评估：

$p(s_1)$=100+40=140；　　$w(s_1)$=20+20=40<50

$p(s_2)$=60+120=180；　　$w(s_2)$=10+30=40<50

$p(s_3)$=120+30=150；　　$w(s_3)$=30+10=40<50

$p(s_4)$=60+100+120+30+40=350；　　$w(s_4)$=10+20+30+10+20=90>50

$p(s_5)$=100+120+30=250；　　$w(s_5)$=20+30+10=60>50

上述解中，s_4 和 s_5 都超过载重限额，为非法解。只有解 s_1、s_2、s_3 合法。

解的选择：在合法的 3 个解中实行局部选择法，将最低适应值的解 s_1 删除，选中 s_2、s_3。

后代繁衍：对选中的两个解进行均匀交配（或操作）获得新解：s_6=10110。

新解评估：$p(s_6)$=60+120+30=210；　　$w(s_6)$=10+30+10=50≤50

新解合法，且超过我们的心理预期 200（假设我们的心理预期为价值 200 元），算法结束。

15.11.5 遗传算法的现状

遗传算法由于简单、容易理解（中小学生也可以理解）、可调的参数多（评估函数、繁衍方式、选择方法等）、应用范围广（几乎无所不及），引来了一大堆人对其进行“研究”。这些研究导致遗传算法的应用领域不断扩大，目前已经扩展到机器学习、神经网络、模糊推理、混沌理论、并行处理，并与进化算法不断交融，导致数量繁多的新“成果”或者新“发明”。这些成果和发明是如此之多，如果要全部论述，恐怕得再写 10 本书。鉴于此，本书干脆就不费此力，有兴趣的读者可自行查阅相关资料。

15.12 概率尽在一切中

到目前为止，我们已经知道，世界上的绝大部分问题是所谓的“难解”或“无解”问题。一方面，求解这些问题的精确解是人类无法完成的任务，另外一方面，人类的好强本性又驱使我们去寻找答案。这样在我们的理想与现实之间就出现了一道鸿沟，而架起这似乎不可跨越的鸿沟的桥梁就是“近似解”。即使近似解离真实解（如果问题存在解）很远，或者根本就不是解（如果问题是无解），它也总是会让我们感觉到一丝心里安慰。

对于绝大部分问题来说，求精确解通常只有一个，求近似解则可能有无数。不幸的是，在大部分情况下，近似解到底近似到何种程度并不清楚。而且哪一种近似才是更加近似也不太清楚。我们可以从各种方面来近似或迫近，每个不同的近似方向就可以导致一种不同的近似解法，而不同的近似程度显然也意味着不同的近似解。我们当然可以比较不同近似解的好坏，但这种比较终归有点“五十步笑百步”的味道，因此，比较的结果并不总是能让所有人信服，甚至是随机的。而这种随机性总让人不舒服。遗传算法虽然在实际中可能效果不错，但它毕竟缺乏理论依据，纯粹是根据实际效果和个人好恶设定各种操作和评判标准。例如，根据遗传算法推测，单点交配由于其破坏力小（仅对父母的基因作微小变动），其性能应该

优于均匀交配，而实际结果则是，均匀交配的性能优于单点交配。这样，虽然我们不能从逻辑上解释为什么均匀交配要优于单点或多点交配，但在实际中我们选择的是均匀交配。

从某种程度上说，近似算法是人类已经没有能力解决问题的时候，想通过随机，把（部分）命运交给天意来解决这些难题，这本质上是一种脚踏西瓜皮，滑到哪里算哪里的做派。但这种脚踏西瓜皮，滑到哪里算哪里的做派可以带给我们极大的变通性、充盈的随机性、判断标准的模糊性，导致近似算法成为一个大有作为的领域，因为任何人都可以进行任何设计和近似，即使这些设计和近似是匪夷所思甚至荒谬的。在这种情况下，我们能够依赖的恐怕只有概率了。“解”的好坏或者“近似”的程度就取决于这个概率，从这个意义上说，也许概率尽在一切中。

不，也许没有什么也许，因为人生就是概率。正如法国数学家拉普拉斯所说：“生命中最重要的问题，从根本上说，不过都是概率问题。”（....The most important questions of life are, for the most part, really only problems of probability.）

思考题

1. 有同学认为判断方程式 $x^3yz+2y^4z^2-3xy^5z=18$ 是否存在解的问题不是不可决定问题，而是 NP 问题，因为给定 x、y、z 的任意一组值，我们可以在 $O(1)$时间内判定该组值是否是方程的解。你对此有何看法？详细说明你的推理过程。
2. 你能否设计一个算法告诉我们一个程序在给定输入下是否产生缓冲区溢出？给出理由。
3. 本章讨论了程序终结的判断是不可能的。但平时我们细读一个程序的时候经常能够判断一个程序是否终结。请问这两点之间是否存在矛盾？为什么？
4. 根据本章的讨论，自动判断一个程序是否存在死循环是不可能的。但很多软件工程研究声称它们设计的算法能够检测死循环，请问你相信哪个论断？为什么？
5. 在本章讨论的破解电瓶车失窃事件里，启发式函数是什么？
6. 在本章讨论的破解电瓶车失窃事件里，使用的并不只有启发式搜索一种策略。请问还使用了什么算法策略？说明你的理由。
7. 有同学提出，启发式搜索实际上是一种贪婪选择策略。你同意这个观点吗？说明理由。
8. 近似算法与竞争分析有什么关系？
9. 模拟退火算法的哪一个特点说明它是一个蒙特卡洛算法？
10. 证明 BACKTRACKING-EIGHT-QUEEN-DFS-SOLUTION 考虑的皇后放置数为 5 508 种。
11. 遗传算法的最大缺点是什么？予以详细说明。
12. 在“优化编译器”的设计中，人们总是试图对前端编译后的中间代码进行某种“优化”以达到改善目标代码的执行效率或能耗。请问，判断一种“优化”手段是否达到其目的是一个什么样的问题？无解？难解？还是别的什么类型问题？给出你的理由。
13. 评价一本书的优劣是一种何级别的问题？你能否设计一个近似算法来评价一本书的优劣？请详细说明你的算法，并给出算法的近似程度。

PDG

算 法 之 道

1969 年、1970 年、1973 年，美国麻省理工学院（Massachusetts Institute of Technology）的金融经济学者罗伯特·默顿（Robert C. Merton）相继推出了默顿投资组合问题（Merton Portfolio Problem）、默顿模型（Merton Model）、差时资本资产定价模型（Intertemporal Capital Asset Pricing Model）等金融经济理论。在这些充满令人眼花缭乱的概率分析、数学推导和算法设计的模型里，默顿第一次告诉人们应该怎样消费和投资，公司如何为普通股建立期权，如何使用套头交易来保护投资，并破天荒地提出了将公司的股票市值作为公司总资产的构成部分的神奇建议。一句话，默顿希望教育人们“如何像人一样地生活”。

虽然这些理论、建议、模型被某些金融从业人员视为垃圾，没有任何用处和漏洞百出的“狂人呓语”，但这些“垃圾”却赢得了其他一些的人“崇高敬意”。默顿在麻省理工学院的导师保罗·安东尼·萨缪尔森（Paul Anthony Samuelson）甚至称赞默顿为现代金融业的牛顿。而默顿自己更是踌躇满志，意气风发。他觉得自己肩负着“将人类从金融混沌中拯救出来”的历史使命。而完成这一使命的理论指导原则已经确立，它们就是默顿的各种概率推导和算法设计。

更令人崇敬的是，默顿并不甘于只在理论上指点几下江山。为了将人们“从混沌和贫穷中拯救出来”，默顿不惜赤膊上阵，与另一位金融经济学者麦荣·思考乐（Myron Scholes）及两位华尔街的投资“专家”一起创办了美国长期资本管理公司（Long-Term Capital Management），他要在金融实践中向人们展示其模型的威力和其“拯救”世人出混沌的惊天动地的革命精神。

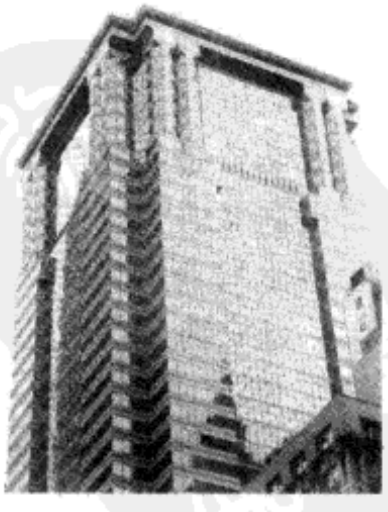


图 1　美国长期资本管理公司在崩盘前的办公之处

这种大无畏的革命主义精神深深地感动了位于瑞典斯德哥尔摩的诺贝尔奖的评委们。他们对默顿的各种金融理论甚为折服，毅然在 1997 年授予了默顿诺贝尔经济学奖（同时获奖的还有默顿的合伙人麦荣·思考乐）。就在普罗大众对默顿感到高山仰止的时刻，奇迹发生了。1998 年，美国长期资本管理公司（见图 1）在 4 个月内亏损了 46 亿美元，导致公司无以为继而轰然倒地，迫使美联储不得不出面拯救。这位号称要“解救世界”的人看来先需要被别人解救一次。

当然，一次倒闭不能说明问题，胜败乃兵家常事。在美国长期资本管理公司倒塌后，邀请默顿进行演讲、报告和担任咨询或顾问职位的公司仍然络绎不绝。在经过仔细考虑后，默顿于 2007 年接受了美国 Trinsum Group 公司的邀请，担任其首席科学家。然而令人遗憾的是，2009 年 1 月 30 日，Trinsum Group 公司宣布破产（申请破产保护）……

至此，默顿的闪耀生涯似乎走完了一个阶段。

至此，本书也已经写到了尾声。

也许在翻看了这几百页的文字后，我们对算法的理解达到了一个新的高度，我们或许可以有一些满足，一丝欣慰，一点感想，但不可有的是踌躇满志。因为本书所阐述的算法只是沧海一粟，对算法思想的论述也只不过是开启了一扇小窗。窗内的风景虽然宜人，但窗外的原野却更为广袤无垠。更为重要的是，仅有算法是远远不够的，我们还需要很多其他知识，才能够真正成为计算机领域的知路人。就像美国 EMC 公司的“英雄软件”SRDF，算法精妙只不过是它优良品质里的一个小小组成部分，它之所以能够在 9·11 事件中挽摩根斯坦利（Morgan Stanley）公司的数据与信息系统于既倒，算法之外的部分也同样功不可没，甚至更为关键。

因此，就算是掌握了算法，也没有任何值得骄傲的地方。况且，我们对算法的掌握实在有限。即使是对于那些非常经典的算法，我们对自己的理解也不一定能够深信不疑。

事实上，当一个人踌躇满志的时候，当一个人对自己的理论、知识、智慧或者任何方面充满自信的时候，也许灾难就在附近了。因为我们所赖以自信的基础其实并不牢靠。就像默顿对自己各种金融模型里所设计或使用的算法、所进行的概率分析非常信赖那样，我们也许对各种经典算法有着同样的信赖。而在这种深度信赖里面，我们会忘记算法只是人类大脑中的一个想象。它本来是无所谓有，无所谓无的。有了人脑的想象，就有了算法；没有人脑的想象，就没有算法。既然算法是存在于人类的大脑中，就难免不会因为大脑里各种电信号的交互错误而导致问题。而算法的精妙或许就在这电光闪耀的一刻而流失。就像从无有到无穷，又从无穷到无有莫测变幻一样。这种变幻的诱因就是人类思想在一刹那的转念。

也许就像人类一直在探索的人生意义。这个问题也许有一个算法解，但为什么不同的人得出的结论是如此的不同呢？难道人们在解答这个问题上的算法有如此根本的差异？答案或许就在本书的字里行间，只不过很多人不一定能够看出来而已。

经上写着：引到灭亡的门是宽的，路是阔的，进去的人也多；而引到永生的那门是狭小的，路是窄的，找着的人也少。

也许，算法的精髓就是如此，通向它的路是窄的，门是狭小的，找到的人很少（见图 2）。甚至于没有人找到路。而那些自称或所谓的“找到路的人”，即所谓的“算法大师”，也许只是另一个或另一些罗伯特·默顿而已。有一天当我们真正睁开眼睛的时候，却发现其实什么也没有。而导致从无穷到无有也许正是自己所赖以安身立命的“算法”。

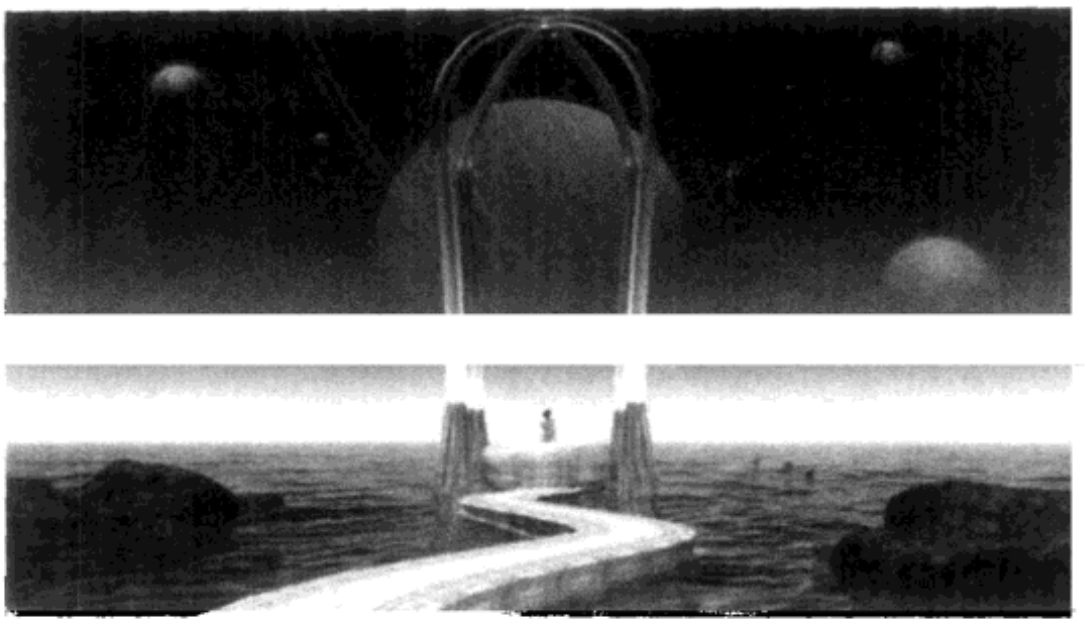


图 2 通向算法精髓的门是狭小的，路是窄的，找着的人也少

不管怎样，无论是寻找到了奥秘，还是变得更加迷茫，我都希望阅读本书对于读者是一次身心愉悦的曼妙历程。如果不是，则只能是我的遗憾了。反正世界上的遗憾很多，多这一次也无所谓了。不过，请不要失望。因为只要有一丝信念，就还有一线希望。

寻找，必寻见；叩门，就开门；相信，便得救。

也许这就是算法之道？

也许这就是生命之路？

PDG

算 法 随 想

原作：彭小朋[㊀]　　改编：邹恒明

从小时候就开始数了。
数到懂事，数到成熟，
还没有数清。
为什么天上的星星数不清呢？
像记忆和幻想，
永远背负着固执的谜……

——童年的歌

数数真的很难吗？

给你一个无限容积的罐子和无限个球，球从 1 开始连续编号。

在差 1 分钟到零点时：将标号为 1～10 的十个球放进罐子，然后将 10 号球从罐子拿出。

在差 1/2 分钟到零点时：将标号为 11～20 的十个球放进罐子，然后将 20 号球从罐子拿出。

在差 1/4 分钟到零点时：将标号为 21～30 的十个球放进罐子，然后将 30 号球从罐子拿出。

……

就这样将游戏进行下去。假定放球和取球不占时间，请问，当时钟指向零点时，罐子里剩有多少球？如果我们改变实验中的拿球方式，将每次取 10、20、30……号球分别变为拿 1、2、3……号球，结果又怎样呢？

对于很多人来说，算法就是解决问题的方法，具体来说就是一系列解决问题的步骤。这些步骤所推进的顺序和规则十分重要，就像上面的两个实验。只需将拿球的编号进行改动（取球的顺序变换），结果就会大相径庭：第一种情况下罐子里剩有无限个球，第二种情况罐子里则没有球……

㊀ 彭小朋是我算法课的学生，这篇文章是她在一学期算法课结束后所写的感想。

这个结果听上去似乎合理，因为取球顺序的变化使得算法发生了变化，即我们实际上讨论的是两个算法。

可仔细一想又觉得不对，因为两个算法都是每次放进 10 个球，取出 1 个球，即从根本上说，这是两个一样的算法，怎么会有截然相反的结果呢？难道标号的不同使最后的球的数量发生了变化?

没错。就是这个标号对结果产生了深远的影响。从某种意义上说，标号是虚的，它只存在于我们的想象中，但却确实对现实结果产生了影响。即我们的思维使得算法发生了变化。从这个意义上说，算法是一种思维方式（algorithmic thinking），或者说一种哲学，一种人生。

事实上，算法如人生，人生也如算法。它由许多过程所构成，又因这些过程而改变。虽然不能肯定结果正确，但求解过程却可以合理。透过算法的本质，我们可以看到芸芸众生，各色人相，多彩的生活。只要愿意，每个人都能寻找一个属于自己一生的独特“算法”，为自己的将来绘出一张或简单或复杂，或单调或反复，或直接或递归，或串行或并行，或随机或确定，或成功或失败（假如世界上存在所谓的成功和失败）的蓝图。

有人天生喜欢“遍历”，踏遍千山万水，遍享万种风情。扮演各种角色，希望人生丰富多彩；有人一生“贪婪”，眼界不宽，及时行乐；有人注定适用“穷搜”，辛辛苦苦、勤勤恳恳一辈子，付出很多，收获有限；有人善用“时空权衡”，用最少的时间办最多的事情，的确精明；有人会“分治”，再多的难题也能迎刃而解；有人常“回溯”，错的太多，后悔太多；有的人压根没有算法，于是盲目生活，盲目做事，最后所获无几；有人“动态规划”，从而积少成多。

智者希望统筹兼顾，努力设计一个最优的人生。遇到一个大的工程，他们分而治之，治而合之。今天做一部分，明天做一部分，终究会做完。尽管有时效率不高，但总比一直放在那里，叹息发愁有意义。当一个问题的最优解，包含了子问题的最优解的时候，他们就选择动态规划，通过选择子问题的最优解来构造原问题的最优解。

贪婪者希望运气长在，他们将每一次的选择缩小到一种贪心的选择。如果运气好，也许会成功；而在多数时候，却只会让人误入歧途。

“智者千虑，必有一失。”如果猛然发现自己做错了选择，不必恐慌，试试回溯吧，退回去，重新来过。在下一次的选择时，记得使用分支估界，从某些途径得到一些经验，来判断哪些路径不好。当然，我们也可以随机选择一些路径来实施，说不定能立竿见影。但总的来说，回溯不是万能的，人生的路，有时一旦走过，便无法返回。毕竟人生有限，频繁回溯，重做选择，就只有原地踏步的份了。分叉路口，做一个恰当的选择的确是困难的，在对与错、是与否、灵与肉、坚持与背叛、努力与放弃之间，我们需要细心。但一定要踏出下一步。至少，我们就可以知道它并不在这里。

有时候，生活中总会有那么一点无奈，无论你花多少力气，用多大的毅力，也不能产生效果。那是因为很多事情即使是可行的，也不一定是有意义的。所以聪明的人懂得用意念改变现实，用智慧区分可行和不可行，用信念辨别有意义和无意义。如果觉得生活太累，环节太多，试试跳转表吧，也许有些环节并不必须。如果觉得生活中某个步骤成本太高，试试摊

销分析吧，也许我们别的步骤走得太过容易。如果时间珍贵，就试试完美散列，直接将自己定位到目标上。不过记住，完美散列并不完美，它的高昂代价也许你的灵魂承受不起。

智者用渐近分析获得问题的内在复杂性。当明了一个问题的计算时间下界，就可以评价解决该问题的各种算法的效率，进而确定对已有算法还有多少改进的余地。如果是 P 类问题，就奋力继续；如果遇到的是 NP 完全问题，就找一个近似的最优解。就像大多数人都无法（不能或不愿）回避的找对象问题：每人的心中都有一个标杆（白马王子或白雪公主），想找到完全符合的并不容易，但也不能证明这个人不存在，这不就是 NP 完全问题吗？怎么解决？求近似解吧，在每一次的恋爱中逐步靠近那个标杆。

算法要求于至简。漫漫人生，相信每个人心中都有一个与生俱来的梦想，这是算法的灵魂——循环不变式。它或许会随着经历的不同而发生变化，但它在你心中的位置是不会变的。位置有远有近，到达的路途有平坦有崎岖，那又有什么关系？当你实现了最初的梦想，所有的循环终将定位在最美的一霎——那便是永恒不变的信念。

在人生的成长过程中，糊里糊涂的人过着糊里糊涂的生活，不知道自己离心中的那个位置是越来越近，还是渐行渐远。而聪明人算法庞大却精妙，因为他们常常反省自己，检查这个不变式，不断简化，每一步都坚定迈向至简，即便是 “劝君更尽一杯酒，西出阳关无故人” 的独自前行。

算法终将归于永恒。人生的算法林林总总。有人天生智慧、美丽，有人生来愚笨、丑陋。你并不需要为此庆幸、骄傲，或者伤心、气馁，甚至抱怨上帝的不公。这些是无用，也是不必要的。因为如果一个算法足够健壮，初始条件带来的影响几乎是微不足道的。我们感到疲惫不堪、精疲力竭和毫无乐趣的，通常并非繁重的人生，而是没有意义和希望的人生，因为“没有目标的算法无法向正确推进”。

不同的算法演绎不同的人生。它的影响不限于今世，而是达于另一个维度。世俗精彩过眼消散，万种风情终将逝去。意识到这点，我们突然发现，过去看重的很多活动、目标、追求，甚或其他问题，都突然显得微不足道和不值一提。我们会重新订立优先次序，将算法的目标锁定为永恒做准备，升华我们的灵魂。因为这个算法将是我们面对造物主的终极答卷。

“一花一世界，一叶一菩提。”算法，一段神奇的代码，演绎一段传奇的人生，锁住一段永恒的痕迹。理解算法、把握人生，让我们用算法的一生，刻印下我们一生的算法，在没有时间也没有空间的虚无中循环往复，留下我们那 “莫愁前路无知己，天下谁人不识君” 的不灭印记。这就是求于至简，归于永恒的境界……

参 考 文 献

[1] Hengming Zou. Courseware for Algorithm Design and Analysis[G]. 2006-2010.

[2] Sanjoy Dasgupta, Christos Papadimitrious, Umesh Vazirani. 算法概论[M]. 钱枫，邹恒明，注释. 北京：机械工业出版社，2009.

[3] Thomas H Cormen，et al. Introduction to Algorithm [M]. 2nd ed. 北京：高等教育出版社，2006.

[4] Sheldon Ross. A First Course in Probability[M].4th ed. Prentice Hall, 1994.

[5] G H Hardy, E M Wright. Introduction to the Theory of Numbers[M]. Oxford University Press, 1980.

[6] Michael R Garey, David S Johnson. Computers and Intractibility: A Guide to the Theory of NP-Completeness[M]. W. H. Freeman & Co., 1979.

[7] B Dixon, M Rauch, R E Targan. Verification and sensitivity analysis of MSTs in linear time[J]. SIAM J. on Computing, 1992: 1184-1192.

[8] David R Karger, Philip N Klein, Robert Endre Tarjan. A Randomized Linear-Time Algorithm to Find Minimum Spanning Trees[J]. Journal of the ACM, 1995: 321-328.Http//www. cs.umd.edu/~gasarch/651/KKT.pdf.

[9] Clifford A Shaffer. A Practical Introduction to Data Structures and Algorithm Analysis [M]. Prentice Hall, 2002.

[10] D R Stinson, Cryptograph. Theory and Practice[M]. Chapman and Hall, 2005.

[11] R Motwani, P Raghavan. Randomized Algorithms[M]. Cambridge University Press, 1995.

[12] http://www.wikipedia.com.

[13] 邹恒明. 计算机的心智：操作系统之哲学原理[M]. 北京：机械工业出版社，2009.

[14] 邹恒明. 有备无患：信息系统之灾难应对[M]. 北京：机械工业出版社，2009.

[15] Rick Warren. Purpose Driven Life[M]. 上海：上海三联书店, 2007.

[16] Kenneth Rosen.Discrete Mathematics and Its Applications[M].5th ed. 北京：机械工业出版社，2007.

[17] Jean Hugard.Encyclopedia of Card Tricks [M]. Dover Publications, 1974.

[18] Stephen W. Hawking. A Brief History of Time[M].Reprint edition. Bantam, 2008.

[19] Isaac Newton. The Principia: Mathematical Principles of Natural Philosophy[M]. University of California Press, 1999.

[20] Herbert Lockyer Sr., et al. Illustrated Dictionary of the Bible[M]. Nelson, 1997.

[21] Linda R. Monk. The Words We Live By[M]. Hyperion, 2003.