普通高等教育“十一五”国家级规划教材

高等学校计算机系列

计算机常用算法与程序设计教程

杨克昌　主编

人民邮电出版社
POSTS & TELECOM PRESS
北　京

图书在版编目（CIP）数据

计算机常用算法与程序设计教程 / 杨克昌主编. —北京：人民邮电出版社，2008.11
普通高等教育“十一五”国家级规划教材. 高等学校计算机系列
ISBN 978-7-115-17832-9

Ⅰ. 计… Ⅱ. 杨… Ⅲ. ①电子计算机—算法理论—高等学校—教材②程序设计—高等学校—教材 Ⅳ. TP301.6 TP311.1

中国版本图书馆 CIP 数据核字（2008）第 033281 号

内 容 提 要

本书遵循“内容实用，难易适当，面向设计，注重能力培养”的要求，讲述了穷举、回溯、分治、递归、递推、贪心算法与动态规划等计算机常用算法，同时简要介绍了模拟、智能优化与并行处理。本书注重常用算法的设计与应用，算法设计与程序实现的结合，以及算法的改进与程序优化，力求理论与实际相结合，算法与程序相统一。

书中所介绍的算法通常给出完整的 C 程序，并在 TC（VC++）环境下编译通过，为学习计算机常用算法与程序设计提供了范例。为便于读者练习，每章都附有习题，同时在附录中给出了习题求解的算法提示。

本书可作为高等院校计算机及相关专业“算法设计与分析”、“计算机常用算法与程序设计”课程的教材，也可供软件设计人员与计算机爱好者学习参考。

普通高等教育“十一五”国家级规划教材
高等学校计算机系列
计算机常用算法与程序设计教程

◆ 主　　编　杨克昌
责任编辑　张孟玮

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街 14 号
邮编　100061　　电子函件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京华正印刷有限公司印刷

◆ 开本：787×1092　1/16
印张：17.25
字数：420 千字　　2008 年 11 月第 1 版
印数：1 – 3 000 册　　2008 年 11 月北京第 1 次印刷

ISBN 978-7-115-17832-9/TP

定价：28.00 元

读者服务热线：(010)67170985　印装质量热线：(010)67129223
反盗版热线：(010)67171154

前　言

计算机程序设计是一种创造性思维活动，其教育必须面向应用。“计算机常用算法与程序设计”是计算机专业的核心课程，其教学目的是提高学生算法与程序设计水平，培养通过程序设计解决实际问题的能力。

通过对现有计算机专业本科生学习“算法设计与分析”课程效果的了解与调查，情况不容乐观。很多同学对所学过的算法描述与实施步骤不清楚，有些甚至对算法的基本概念与设计思想不甚了解，无法通过设计程序解决一些常见的应用问题。造成这一现象的原因是多方面的，缺少适合计算机本科层次的“算法与程序设计”教材是其中一个重要方面。作为普通高等教育“十一五”国家级规划教材，本书在内容选材与深度的把握上，在理论阐述与设计应用的结合上进行了精心设计，力图适合高校本科教学的目标与知识结构的要求。本书遵循“内容实用，难易适当，面向设计，注重能力培养”的宗旨编写，在以下几个方面对算法教材进行了改革探索。

（1）注重常用算法的设计与应用。教材在选材上避免贪多求全、贪广求深，以至出现本科阶段与研究生阶段的教学内容混杂不分的局面，只讲基本的常用算法，以及常用算法中要求本科学生掌握的基本内容，去除一些难度大、理论深、应用少的带研究性质的算法内容。

（2）注重算法设计与程序实现的紧密结合。算法与程序实际上是一个统一体，不应该也不能将它们对立或分割。教材在材料的组织上克服了以往罗列算法多、应用算法设计解决实际问题少、算法与程序设计脱节、算法理论与实际应用脱节的问题，在讲述每一种常用算法时，力求理论与实际相结合、算法与程序相统一，突出算法在解决实际问题中的应用，切实提高对常用算法的理解和掌握。针对每一种常用算法，精选典型的实际应用问题或课题，使用 C（C++）语言进行算法描述与程序设计，从问题提出、算法描述到程序实现连成一体，切实提高学生应用算法与程序设计解决实际问题的水平和举一反三的能力。

（3）注重算法改进与程序优化。教材对典型例题提供了多种不同的算法设

计，体现了算法的灵活性和适用性。算法与程序设计都不是一成不变的，算法改进与程序优化的过程是算法设计水平提高的过程。通过算法改进与程序优化，努力培养学生的优化意识与创新能力。

本书结构清晰简明，内容深入浅出。全书共分 9 章。

第 1 章算法与程序设计简介，对算法描述与算法的复杂度作了简要阐述。

第 2 章穷举与回溯，在简要介绍穷举这一基础算法的应用及其优化的同时，着重阐述了“聪明搜索”回溯法的设计要点与实际应用。

第 3 章递归与分治，这是最常用也是应用最广的有效算法。

第 4 章递推，在概括常用递推模式的基础上，着重列举了递推在数列与数阵求解，特别在应用递推求解计数问题方面的应用。

第 5 章贪心算法，举例说明了贪心算法在最优化求解方面高效率的应用。

第 6 章动态规划，介绍了在最优化问题求解中应用最广的动态规划的设计思想、设计要点与操作步骤，着重列举了动态规划在 0-1 背包、最长子序列与最优路径探索方面的应用。

第 7 章模拟，着重讲述了运算模拟、随机模拟与操作过程模拟等方面的具体应用。

第 8 章智能优化，简要介绍了模拟退火、遗传算法、粒子群算法、神经网络等智能优化算法的概念与基本内容。

第 9 章并行算法简介，介绍近年来发展非常迅速的并行计算的基本理论与简单应用。

本书由杨克昌任主编。第 1、6、7 章由杨克昌编写，第 2、4 章由吴小明编写，第 3、8、9 章由郭观七、李文彬、卢泽勇编写，第 5 章由谭用秋编写，最后由杨克昌统稿、定稿。本书的编写得到湖南理工学院领导和计算机与信息工程系的大力支持，编者在此表示衷心感谢。

由于编者学识水平所限，书中疏漏与错误之处在所难免，恳请广大读者批评指正。

编者

2008 年 2 月

目　录

第 1 章　算法与程序设计简介 …… 1
1.1　算法与算法描述 …… 1
1.1.1　算法 …… 1
1.1.2　算法描述 …… 2
1.2　算法复杂性分析 …… 6
1.2.1　时间复杂度 …… 6
1.2.2　空间复杂度 …… 10
1.3　程序设计简介 …… 11
1.3.1　算法与程序 …… 11
1.3.2　结构化程序设计 …… 14
习题 …… 15
第 2 章　穷举与回溯 …… 17
2.1　穷举及其应用 …… 17
2.1.1　穷举概述 …… 17
2.1.2　穷举应用 …… 18
2.2　穷举设计的优化 …… 22
2.2.1　优选穷举对象 …… 22
2.2.2　优化穷举循环参量 …… 23
2.2.3　精简穷举循环 …… 27
2.3　回溯法及其描述 …… 30
2.3.1　回溯的基本概念 …… 30
2.3.2　回溯法描述 …… 30
2.3.3　回溯法的效益分析 …… 33
2.4　回溯设计应用 …… 34
2.4.1　桥本分数式 …… 34
2.4.2　排列组合 …… 36
2.4.3　德布鲁金环序列 …… 41
2.4.4　高斯皇后问题及其拓展 …… 45
2.5　回溯设计的优化 …… 51
习题 …… 54

第 3 章　递归与分治 …… 56
3.1　递归及其应用 …… 56
3.1.1　递归与递归调用 …… 56
3.1.2　递归应用 …… 57
3.2　分治法概述 …… 61
3.2.1　分治法基本思想 …… 61
3.2.2　分治算法设计方法和特点 …… 62
3.2.3　分治法的时间复杂度 …… 64
3.3　分治法的基本应用 …… 65
3.3.1　数据查找与排序 …… 65
3.3.2　计数逆序排名问题 …… 70
3.3.3　投资问题 …… 72
3.4　消除递归 …… 73
3.4.1　一般的递归转非递归 …… 73
3.4.2　分治算法中的递归转化 …… 76
习题 …… 77
第 4 章　递推 …… 79
4.1　递推概述 …… 79
4.1.1　递推算法 …… 79
4.1.2　递推实施步骤与描述 …… 80
4.2　递推数列 …… 82
4.2.1　裴波那契数列与卢卡斯数列 …… 83
4.2.2　分数数列 …… 85
4.2.3　幂序列 …… 87
4.2.4　双关系递推数列 …… 90
4.3　递推数阵 …… 93
4.3.1　杨辉三角 …… 93
4.3.2　折叠方阵 …… 95
4.4　应用递推求解应用题 …… 97
4.4.1　猴子爬山问题 …… 98
4.4.2　整币兑零问题 …… 100
4.4.3　整数划分问题 …… 102
4.5　递推与递归比较 …… 105
习题 …… 107
第 5 章　贪心算法 …… 109
5.1　贪心算法概述 …… 109
5.2　贪心算法的理论基础 …… 110
5.3　删数字问题 …… 111
5.4　背包问题 …… 112
5.4.1　0-1 背包问题 …… 112

5.4.2 可拆背包问题 113
5.5 覆盖问题 115
5.6 图的着色问题 117
5.7 遍历问题 120
5.8 最小生成树 123
5.9 哈夫曼编码 130
习题 134

第 6 章 动态规划 135
6.1 一般方法与求解步骤 135
6.1.1 一般方法 135
6.1.2 动态规划求解步骤 136
6.2 装载问题 137
6.3 插入乘号问题 141
6.4 0-1 背包问题求解 145
6.4.1 0-1 背包问题 146
6.4.2 二维 0-1 背包问题 151
6.5 最长子序列探索 156
6.5.1 最长非降子序列 156
6.5.2 最长公共子序列 158
6.6 最优路径搜索 161
6.6.1 点数值三角形的最优路径搜索 161
6.6.2 边数值矩形的最优路径搜索 163
6.7 动态规划与其他算法的比较 166
6.7.1 动态规划与递推比较 166
6.7.2 动态规划与贪心算法比较 166
习题 167

第 7 章 模拟 168
7.1 模拟概述 168
7.2 运算模拟 168
7.2.1 运算模拟描述 168
7.2.2 n 个 1 的整除问题 170
7.2.3 尾数前移问题 172
7.2.4 阶乘与幂的计算 174
7.2.5 求圆周率π 176
7.3 随机模拟 178
7.3.1 进站时间模拟 178
7.3.2 蒙特卡罗模拟计算 179
7.3.3 模拟发扑克牌 181
7.4 操作过程模拟 183
7.4.1 洗牌 183

7.4.2 泊松分酒 …… 185
7.4.3 模拟小孔流水 …… 188
7.5 模拟外索夫游戏 …… 190
习题 …… 194
第 8 章 智能优化 …… 195
8.1 模拟退火算法 …… 195
8.1.1 物理退火过程和 Metropolis 准则 …… 195
8.1.2 模拟退火算法概述 …… 196
8.1.3 应用举例 …… 198
8.2 遗传算法 …… 199
8.2.1 生物的进化与遗传 …… 200
8.2.2 遗传算法概述 …… 200
8.2.3 遗传算法关键参数 …… 205
8.2.4 遗传算法应用举例 …… 206
8.3 粒子群优化算法 …… 208
8.3.1 粒子群算法的基本结构 …… 209
8.3.2 粒子群算法的关键参数 …… 209
8.3.3 应用举例 …… 210
8.4 人工神经网络 …… 212
8.4.1 神经网络模型 …… 213
8.4.2 神经网络学习规则 …… 214
习题 …… 215
第 9 章 并行算法简介 …… 216
9.1 基本概念 …… 216
9.1.1 并行计算机系统结构模型 …… 216
9.1.2 并行计算性能评价 …… 217
9.2 并行算法设计 …… 219
9.2.1 SIMD 共享存储模型 …… 220
9.2.2 SIMD 互连网络模型 …… 224
9.2.3 MIMD 共享存储模型 …… 225
9.2.4 MIMD 异步通信模型 …… 229
9.3 并行程序开发 …… 231
9.3.1 并行程序设计概念 …… 232
9.3.2 共享存储系统并行编程 …… 232
9.3.3 分布存储系统并行编程 …… 238
习题 …… 243
附录 1 习题解答算法提要 …… 244
附录 2 C 常用库函数 …… 264
参考文献 …… 267

第 1 章　算法与程序设计简介

1.1　算法与算法描述

在计算机科学与技术中，算法（algorithm）是一个常用的基本概念。本节论述算法的定义与算法的描述。

1.1.1　算法

在计算机科学中，算法一词用于描述一个可用计算机实现的问题求解方法。算法是程序设计的基础，是计算机科学的核心。计算机科学家哈雷尔在《算法学——计算的灵魂》一书中指出：“算法不仅是计算机学科的一个分支，它更是计算机科学的核心，而且可以毫不夸张地说，它和绝大多数科学、商业和技术都是相关的。”

在计算机应用的各个领域，技术人员都在使用计算机求解他们各自专业领域的问题，他们需要设计算法，编写程序，开发应用软件，所以学习算法对于越来越多的人来说变得十分必要。

什么是算法？我们首先给出算法的定义。

算法是指解决某一问题的运算序列。或者说算法是问题求解过程的运算描述，一个算法由有限条可完全机械地执行的、有确定结果的指令组成。指令正确地描述了要完成的任务和它们被执行的顺序。计算机按算法所描述的顺序执行算法的指令能在有限的步骤内终止，或终止于给出问题的解，或终止于指出问题对此输入数据无解。

算法是满足下列特性的指令序列。

1. 确定性

组成算法的每条指令是清晰的，无歧义的。

在算法中不允许有诸如“x/0”之类的运算，因为其结果不能确定；也不允许有“x 与 1 或 2 相加”之类的运算，因这两种可能的运算应执行哪一个，并不确定。

2. 可行性

算法中的运算是能够实现的基本运算，每一种运算可在有限的时间内完成。

在算法中两个实数相加是可行的；两个实数相除，例如求 2/3 的值，在没有指明位数时需由无穷个十进制位表示，并不可行。

3. 有穷性

算法中每一条指令的执行次数有限，执行每条指令的时间有限。

例如，如果算法中的循环步长为零，运算进入无限循环，这是不允许的。

4. 输入

一个算法有零个或多个输入。

5. 输出

一个算法至少产生一个量作为输出。

通常求解一个问题可能会有多种算法可供选择，选择的主要标准是算法的正确性和可靠性。其次是算法所需要的存储空间少和执行时间短等。

有人也许会认为：今天计算机运算速度这么快，算法还重要吗？诚然，计算机的计算能力每年都在飞速增长，价格也在不断下降。可我们不要忘记，日益先进的纪录和存储手段使我们需处理的信息量在爆炸式的增长，互联网的信息流量也在快速增长。在科学研究方面，随着研究手段的进步，数据量更是达到了前所未有的程度。无论是三维图形、海量数据处理、机器学习、语音识别，都需要极大的计算量。在网络时代，越来越多的挑战需要靠卓越的算法来解决。

算法并不局限于计算机和网络。在高能物理研究方面，很多实验每秒钟都能产生若干个 TB 的数据量，但因为处理能力和存储能力的不足，科学家不得不把绝大部分未经处理的数据舍弃。在气象方面，算法可以更好地预测未来天灾的发生，以拯救生命。所以，如果你把计算机的发展放到应用和数据飞速增长的大环境下，你一定会发现，算法的重要性不是在日益减小，而是在日益增强。

在实际工程中我们遇到许多的高难度计算问题，有的问题在巨型计算机上采用一个劣质的算法来求解可能要数个月的时间，而且很难找到精确解。但采用一个优秀的算法，即使在普通的个人计算机上，可能只需数秒钟就可以求得解答。计算机求解一个工程问题的计算速度不仅仅与计算机的设备水平有关，更取决于求解该问题的算法技术水平的高低。世界上许多国家，从大学到研究机关都高度重视对计算机算法的研究，已将提高算法设计水平看作是一个提升国家技术竞争力的战略问题。

对同一个计算问题来说不同的人会有不同的计算方法，而不同算法的计算效率、求解精度和对计算资源的需求有很大的差别。

本书具体介绍分治、递归与递推、贪心算法、回溯法、动态规划与模拟等常用算法。同时对智能优化与并行处理作简要介绍。

1.1.2　算法描述

要使计算机能完成人们预定的工作，首先必须为如何完成这些工作设计一个算法，然后再根据算法编写程序。

一个问题可以设计不同的算法来求解，同一个算法可以采用不同的形式来表述。

算法是问题的程序化解决方案。描述算法可以有多种方式，如自然语言方式、流程图方式、伪代码方式、计算机语言表示方式与表格方式等。

当一个算法使用计算机程序设计语言描述时，就是程序。本书采用 C 语言与自然语言相结合来描述算法。之所以采用 C 语言来描述算法，因为 C/C++语言功能丰富、表达能力强、使用灵活方便、应用面广，既能描述算法所处理的数据结构，又能描述计算过程，是目前大学阶段学习计算机程序设计的首选语言。

为方便算法描述与程序设计，下面把 C 语言的基本要点作简要概括。

1. 标识符

可由字母、数字和下划线组成，标识符必须以字母或下划线开头， 大小写的字母分别认为是两个不同的字符。

2. 常量

整型常量：十进制常数、八进制常数（以 o 开头的数字序列）、十六进制常数（以 ox 开头的数字序列）。

长整型常数（在数字后加字符 L 或 l）。

实型常量（浮点型常量）：小数形式与指数形式。

字符常量：用单引号（撇号）括起来的一个字符，可以使用转义字符。

字符串常量：用双引号括起来的字符序列。

3. 表达式

（1）算术表达式

整型表达式：参加运算的运算量是整型量，结果也是整型数。

实型表达式：参加运算的运算量是实型量，运算过程中先转换成 double 型，结果为 double 型。

（2）逻辑表达式

用逻辑运算符连接的整型量，结果为一个整数（0 或 1），逻辑表达式可以认为是整型表达式的一种特殊形式。

（3）字位表达式

用位运算符连接的整型量，结果为整数。字位表达式也可以认为是整型表达式的一种特殊形式。

（4）强制类型转换表达式

用“（类型）”运算符使表达式的类型进行强制转换，如（float）a。

（5）逗号表达式（顺序表达式）

形式为：表达式 1，表达式 2，…，表达式 n

顺序求出表达式 1，表达式 2，…，表达式 n 的值，结果为表达式 n 的值。

（6）赋值表达式

将赋值号“=”右侧表达式的值赋给赋值号左边的变量，赋值表达式的值为执行赋值后被赋值的变量的值。

（7）条件表达式

形式为：逻辑表达式？表达式 1：表达式 2

逻辑表达式的值若为非 0（真），则条件表达式的值等于表达式 1 的值；若逻辑表达式的值为 0（假），则条件表达式的值等于表达式 2 的值。

（8）指针表达式

对指针类型的数据进行运算。例如 p−2、p1−p2、&a 等（其中 p、p1、p2 均已定义为指针变量），结果为指针类型。

以上各种表达式可以包含有关的运算符，也可以是不包含任何运算符的初等量。例如，常数是算术表达式的最简单的形式。

表达式后加“；”，即为表达式语句。

4. 数据定义

对程序中用到的所有变量都需要进行定义，对数据要定义其数据类型，需要时要指定其存储类别。

（1）数据类型标识符有：

int（整型），short（短整型），long（长整型），unsigned（无符号型），char（字符型），float（单精度实型），double（双精度实型），struct（结构体名），union（共用体名）。

（2）存储类别有：

auto（自动的），static（静态的），register（寄存器的），extern（外部的）。

变量定义形式：存储类别 数据类型 变量表列

如：static float x,y

5. 函数定义

存储类别　数据类型　<函数名> (形参表列)

{函数体}

6. 分支结构

（1）单分支：

if(表达式) <语句 1> [else <语句 2>]

功能：如果表达式的值为非 0（真），则执行语句 1；否则（为 0，即假），执行语句 2。所列语句可以是单个语句，也可以是用{}界定的若干个语句。应用 if 嵌套可实现多分支。

（2）多分支：

switch(表达式)

{ case 常量表达式 1：<语句 1>

case 常量表达式 2：<语句 2>
…
case 常量表达式 n：<语句 n>
default：<语句 n+1>
}

功能：取表达式 1 时，执行语句 1；取表达式 2 时，执行语句 2；…，其他所有情形，执行语句 n+1。

case 常量表达式的值必须互不相同。

7. 循环结构

（1）while 循环：

while(表达式) 语句

功能：表达式的值为非 0（条件为真），执行指定语句（可以是复合语句）。 直至表达式的值为 0（假）时，脱离循环。

特点：先判断，后执行。

（2）Do-while 循环：

do　语句
while（表达式）

功能：执行指定语句，判断表达式的值非 0（真），再执行语句；直到表达式的值为 0（假）时，脱离循环。

特点：先执行，后判断。

（3）for 循环：

for（表达式 1；表达式 2；表达式 3）语句

功能：解表达式 1；求表达 2 的值：若非 0（真），则执行语句；求表达式 3；再求表达式 2 的值；……；直至表达式 2 的值为 0（假）时，脱离循环。

以上三种循环，若执行到 break 语句，提前终止循环。若执行到 continue，结束本次循环，跳转下一次循环判定。

顺便指出，在不致引起误解的前提下，有时对描述的 C 语句进行适当简写或配合汉字标注，用以简化算法框架描述。

例如，从键盘输入整数 n，按 C 语言的键盘输入函数应写为：

scanf(“%d”,&n);

可简写为：scanf(n);

或简写为：输入整数 n;

要输出整数量 $a(1),a(2),\cdots,a(n)$，按 C 语言的输出函数应写为：

```
for(k=1;k<=n;k++)
    printf(“%d ”,a[k]);
```

可简写为：

printf(a(1)−a(n))；

或简写为：输出 a(1−n)；

【例 1.1】 求两个整数 *a*,*b*(*a*>*b*)的最大公约数的欧几里德算法：

（1）*a* 除以 *b* 得余数 *r*；若 *r*=0，则 *b* 为所求的最大公约数。

（2）若 *r*≠0，以 *b* 为 *a*，*r* 为 *b*，继续（1）。

注意到任两整数总存在最大公约数，上述辗转相除过程中余数逐步变小，相除过程总会结束。

欧几里德算法又称为“辗转相除”法，具体描述如下：

```
输入正整数 a，b；
if(a<b)
    {c=a;a=b;b=c;}                    /*  交换 a，b，确保 a>b */
r=a%b;
while(r!=0)
  { a=b;b=r;                          /*  实施“辗转相除”*/
    r=a%b;
  }
printf(最大公约数 b);
```

1.2 算法复杂性分析

算法复杂性的高低体现运行该算法所需计算机资源的多少。算法的复杂性越高，所需的计算机资源越多；反之，算法的复杂性越低，所需的计算机资源越少。

计算机资源，最重要的是时间资源与空间资源。因此，算法的复杂性有时间复杂性与空间复杂性之分。需要计算机时间资源的量称为时间复杂度，需要计算机空间资源的量称为空间复杂度。时间复杂度与空间复杂度集中反映出算法的效率。

算法分析是指对算法的执行时间与所需空间的估算，定量给出运行算法所需的时间数量级与空间数量级。

1.2.1 时间复杂度

算法作为计算机程序设计的基础，在计算机应用领域发挥着举足轻重的作用。一个优秀的算法可以运行在计算速度比较慢的计算机上求解问题，而一个劣质的算法在一台性能很强的计算机上也不一定能满足应用的需求。因此，在计算机程序设计中，算法设计往往处于核心地位。如何去设计一个适合特定应用的算法是众多技术开发人员所关注的焦点。

要想充分理解算法并有效地应用于求解实际问题，关键是对算法的分析。通常我们可以利用实验对比方法、数学方法来分析算法。

实验对比分析很简单，两个算法相互比较，它们都能解决同一问题，在相同环境下，哪个算法的速度快我们一般就会认为这个算法性能好。

数学方法能更为细致的分析算法，能在严密的逻辑推理基础上判断算法的优劣。但在完

成实际项目过程中，我们很多时候都不能去做这种严密的论证与推断。因此，在算法分析中，我们往往采用能近似表达性能的方法来展示某个算法的性能指标。例如，当 n 比较大的时，计算机对 n^2 和 n^2+2n 的响应速度几乎没有什么区别，我们便可直接认为这两者的复杂度均为 n^2。在分析算法时，隐藏细节的数学表示法成为大写 O 记法，它可以帮助我们简化算法复杂度的许多细节，提取主要成分，这和遥感图像处理中的主成分分析思想相近。

一个算法的时间复杂度是指算法运行所需的时间。一个算法的运行时间取决于算法所需执行的语句（运算）的多少。算法的时间复杂度通常用该算法执行的总语句（运算）的数量级决定。

就算法分析而言，一条语句的数量级即执行它的频数，一个算法的数量级是指它所有语句执行频数之和。

观察下面三个程序段：

（1）
```
x=x+1;
```
（2）
```
for(k=1;k<=n;k++)
    { x=x+y;
      y=x+y;
      s=x+y;
    }
```
（3）
```
for(t=1,k=1;k<=n;k++)
    {t=t*2;
     for(j=1;j<=t;j++)
        s=s+j;
    }
```

如果把以上 3 个程序段看成 3 个相应算法的主体，我们来看 3 个算法的数量级。

在（1）中，语句执行频数为 1，算法的数量级为 1；

在（2）中，每个赋值语句执行频数为 n，该算法的数量级为 $3n$；

在（3）中，内循环的赋值语句执行频数为 $2+2^2+\cdots+2^n=2(2^n-1)$。

算法的数量级直接决定算法的时间复杂度。

定义　对于一个数量级为 $f(n)$ 的算法，如果存在两个正常数 c 和 m，对所有的 $n\geqslant$m，有

$$|f(n)|\leqslant c|g(n)|$$

则记作 $f(m)=O(g(n))$，称该算法具有 $O(g(n))$的运行时间，是指当 n 足够大时，该算法的实际运行时间不会超过 $g(n)$的某个常数倍时间。

显然，以上所列举的（1）与（2），其计算时间即时间复杂度分别为 $O(1)$，$O(n)$。

据以上定义，（3）的数量级为 $2(2^n-1)$，取 $c=2$，对任意正整数 n，有

$$2(2^n-1)<2\cdot 2^n$$

即得（3）的计算时间为 $O(2^n)$，即算法（3）的时间复杂度为 $O(2^n)$。

以上 3 个程序段前两个所代表的算法是多项式时间算法。最常见的多项式算法时间，其关系概括为

$$O(1)<O(\log n)<O(n)<O(n\log n)<O(n^2)<O(n^3)$$

约定 log*n* 表示以 2 为底的对数。程序段（3）所代表的是指数时间算法。以下 3 种是最常见的指数时间算法，其关系为

$$O(2^n)<O(n!)<O(n^n)$$

随着 *n* 的增大，指数时间算法与多项式时间算法在所需的时间上相差非常大，表 1.1 具体列出了时间复杂度常用函数增长情况。

表 1.1　　时间复杂度常用函数增长情况

	log*n*	*n*	*n*log*n*	n^2	n^3	2^n
n=1	0	1	0	1	1	2
n =2	1	2	2	4	8	4
n =4	2	4	8	16	64	16
n =8	3	8	24	64	512	256
n =16	4	16	64	256	4 096	65 536
n =32	5	32	160	1 024	32 768	4 294 967 296

一般地，当 *n* 取值比较大时，在计算机上实现指数时间算法是不可能的，就是比 $O(n\log n)$ 时间复杂度高的多项式时间算法运行也很困难。

根据时间复杂度符号 *O* 的定义，有

定理 1.1　关于时间复杂度符号 *O* 有以下运算规则：

$$O(f)+O(g)=O(max(f,g)) \tag{1.1}$$

$$O(f)O(g)=O(fg) \tag{1.2}$$

证明　设 $F(n)=O(f)$，根据 *O* 定义，存在常数 c_1 和正整数 n_1，对所有的 $n\geqslant n_1$，有 $F(n)\leqslant c_1 f(n)$。同样，设 $G(n)=O(g)$，根据 *O* 定义，存在常数 c_2 和正整数 n_2，对所有的 $n\geqslant n_2$，有 $G(n)\leqslant c_1 g(n)$。

令 $c_3=max(c_1,c_2),n_3=max(n_1,n_2),h(n)=max(f,g)$。对所有的 $n\geqslant n_3$，存在 c_3，有

$F(n)\leqslant c_1 f(n)\leqslant c_3 f(n)\leqslant c_3 h(n)$

$G(n)\leqslant c_2 g(n)\leqslant c_3 g(n)\leqslant c_3 h(n)$

则　$F(n)+G(n)\leqslant 2c_3 h(n)$

即　$O(f)+O(g)\leqslant 2c_3 h(n)=O(h)=O(max(f,g))$

令 $t(n)=fg(n)$，对所有的 $n\geqslant n_3$，有

$F(n)G(n)\leqslant c_1 c_2 t(n)$

即　$O(f)O(g)\leqslant c_1 c_2 t(n)=O(fg)$。

式（1.1）、式(1.2)成立。

定理 1.2　如果 $f(n)=a_m n^m+a_{m-1}n^{m-1}+\cdots+a_1 n+a_0$ 是 *n* 的 *m* 次多项式，$a_m>0$，则

$$f(n)=O(n^m) \tag{1.3}$$

证明　当 $n\geqslant 1$ 时，据符号 *O* 定义有

$$\begin{aligned} f(n) &= a_m n^m + a_{m-1} n^{m-1} + \cdots + a_1 n + a_0 \\ &\leqslant |a_m| n^m + |a_{m-1}| n^{m-1} + \cdots + |a_1| n + |a_0| \\ &\leqslant (|a_m| + |a_{m-1}| + \cdots + |a_1| + |a_0|) n^m \end{aligned}$$

取常数 $c = |a_m| + |a_{m-1}| + \cdots + |a_1| + |a_0|$，据定义，式（1.3）得证。

【例 1.2】 估算以下程序段所代表算法的时间复杂度。

```
for(k=1;k<=n;k++)
for(j=1;j<=k;j++)
{x=k+j;
s=s+x;
}
```

解：每个赋值语句执行频率为 $1+2+\cdots+n=\dfrac{n(n+1)}{2}$，该算法的数量级为 $n(n+1)$；取 $c=2$，对任意正整数 n，有

$$n(n+1) \leqslant 2 \cdot n^2 \quad \Leftrightarrow \quad n \leqslant n^2$$

即得该程序段的计算时间为 $O(n^2)$，即所代表算法的时间复杂度为 $O(n^2)$。

【例 1.3】 估算下列程序段代表算法的时间复杂度。

（1）
```
t=1;m=0;
 for(k=1;k<=n;k++)
   {t=t*2;
    for(j=t;j<=n;j++)
      m++;
    }
```

（2）
```
d=0;
 for(k=1;k<=n;k++)
 for(j=k*k;j<=n;j++)
       d++;
```

解：（1）设 $n=2^x$，则（1）中 d++语句的执行次数为：

$S=(n+1-2)+(n+1-2^2)+(n+1-2^3)+\cdots+(n+1-2^x)$

$=x(n+1)-2(2^x-1)$

$=(x-2)n+x+2$

注意到 $x=\log n$，则当 $n\geqslant 2$ 时有

$s\leqslant xn=(\log n)n$

可知（1）时间复杂度为 $O(n\log n)$。

（2）设 $n=m^2$，则（2）中 m++语句的执行次数为：

$S=(n+1-1^2)+(n+1-2^2)+(n+1-3^2)+\cdots+(n+1-m^2)$

$=m(n+1)-(1+2^2+\cdots+m^2)$

$=m(n+1)-m(m+1)(2m+1)/6$

$=m(6n+6-2m^2-3m-1)/6$

注意到 $m=\sqrt{n}$，当 $n>3$ 时有 $S<2n\sqrt{n}/3$。

可知（2）时间复杂度为 $O(n\sqrt{n})$。

关于算法的运行时间还有Ω记号、Θ记号与 o 记号等标注。

称一个算法具有 $\Omega(g(n))$ 的运行时间，是指当 n 足够大时，该算法的实际运行时间至少需要 $g(n)$ 的某个常数倍时间。例如，$f(n)=2n+1=\Omega(n)$。

称一个算法具有 $\Theta(g(n))$ 的运行时间，是指当 n 足够大时，该算法的实际运行时间大约为 $g(n)$ 的某个常数倍时间。例如，$f(n)=3n^2+2n+5=\Theta(n^2)$。

标注 $o(g(n))$表示增长阶数小于 $g(n)$的所有函数的集合。$f(n)=o(g(n))$表示一个算法的运行时间 $f(n)$的阶比 $g(n)$低。例如，$f(n)=3n+2=o(n^2)$。

以后在分析算法时间复杂度时对这些标记一般不作过多论述，通常只应用 O 记号来标注算法的时间复杂度。

同时，一个算法的运行时间，与问题的规模相关，也与输入的数据相关。基于算法复杂度简化表达的思想，我们通常会对算法进行最坏情况分析和平均情况分析。对于一个给定的算法，如果能保证它的最坏情况下的性能依然不错当然很好，但是在某些情况下，程序的最坏情况算法的运行时间和实际情况的运行时间相差很大，在实际应用中几乎不会碰到最坏情况下的输入，因此通常省略对最坏情况分析。算法的平均情况分析可以帮助我们估计程序的性能，作为算法分析的基本指标之一。

例如，对给定的 n 个整数 $a(1),a(2),\cdots,a(n)$，应用基本比较法进行由小到大的排序，可以通过以下二重循环实现：

```
for(i=1;i<=n-1;i++)
    for(j=i+1;j<=n;j++)
        if(a[i]>a[j])
            {h=a[i];a[i]=a[j];a[j]=h;}
```

其中 3 个赋值语句的执行频数之和，最理想的情形下为零（当所有 n 个整数已从小到大排列时），最坏情形下为 $3n(n-1)/2$（当所有 n 个整数为从大到小排列时）。按平均情形来分析，其时间复杂度为 $O(n^2)$。

对于一个实用算法，我们通常不必深入研究它时间复杂度的上界和下界，只需要了解该算法的特性，然后在合适的时候应用它。

为了求解某一问题，设计出复杂性尽可能低的算法是追求的重要目标。或者说，求解某一问题有多种算法时，选择其中复杂性最低的算法是选用算法的重要准则。对算法的改进与优化，主要表现在有效缩减算法的运行时间与所占空间。例如，把求解某一问题的算法时间从 $O(n^2)$ 优化缩减为 $O(n\log n)$ 就是一个了不起的成果。或者把求解某一问题的算法时间的系数缩小，例如从 $2n$ 缩小为 $3n/2$，尽管其时间数量级都是 $O(n)$，系数缩小了也是一个算法改进的成果。1969 年斯特拉森（V.Strassen）在求解两个 n 阶矩阵相乘时利用分治策略及其他的一些处理技巧，用了 7 次对 $n/2$ 阶矩阵乘的递归调用和 18 次 $n/2$ 阶矩阵的加减运算，把矩阵乘算法从 $O(n^3)$ 优化为 $O(n^{2.81})$，曾轰动了数学界。这一课题的研究看来并不到此止步，在斯特拉森之后，又有许多算法改进了矩阵乘法的计算时间复杂性，据悉目前最好的计算时间上界是 $O(n^{2.376})$。

1.2.2　空间复杂度

算法的空间复杂度是指算法运行的存储空间，是实现算法所需的内存空间的大小。

一个程序运行所需的存储空间通常包括固定空间需求与可变空间需求两部分。固定空间需求包括程序代码、常量与结构变量等所占的空间。可变空间需求包括数组元素所占的空间与运行递归所需的系统栈空间等。

通常用算法设置的变量（数组）所占内存单元的数量级来定义该算法的空间复杂度。如果一个算法占的内存空间很大，即使其时间复杂度很低，在实际应用时该算法也是很难实现的。

先看以下 3 个算法的变量设置：

（1）int x,y,z;

（2）#define N 1000
　　int k,j,a[N],b[2*N];

（3）#define N 100
　　int k,j,a[N][10*N];

其中（1）设置三个简单变量，占用三个内存单元，其空间复杂度为 $O(1)$。

（2）设置两个简单变量与两个一维数组，占用 $3n+2$ 个内存单元，显然其空间复杂度为 $O(n)$。

（3）设置两个简单变量与一个二维数组，占用 $10n^2+2$ 个内存单元，显然其空间复杂度为 $O(n^2)$。

由上可见，二维或三维数组是空间复杂度高的主要因素之一。在算法设计时，为降低空间复杂度，要注意尽可能少用高维数组。

空间复杂度与前面时间复杂度概念相同，其分析相对比较简单，在以下论述某一算法时，如果其空间复杂度不高，不至于因所占有的内存空间而影响算法实现时，通常不涉及对该算法的空间复杂度讨论。

1.3　程序设计简介

1.3.1　算法与程序

算法是程序设计的基础，是程序的核心。程序是某一算法用计算机程序设计语言的具体实现。事实上，当一个算法使用计算机程序设计语言描述时，就是程序。具体来说，一个算法使用 C 语言描述，就是 C 程序。

程序设计的基本目标是应用算法对问题的原始数据进行处理，从而解决问题，获得所期望的结果。在能实现问题求解的前提下，要求算法运行的时间短，占用系统空间小，上机实践是检验算法与程序的标准。对于求解某一问题的两个算法（程序），一个能圆满解决问题，

另一个不能得到求解结果，前者是成功的，而后者是不成功的。同样，两个算法（程序），都能通过运行得到问题的求解结果，一个只需 2 秒钟，另一个需要 20 分钟，从时间复杂度比较，前者要优于后者。

初学者往往把程序设计简单地理解为编写一个程序，这是不全面的。程序设计反映了利用计算机解决问题的全过程，通常先要对问题进行分析并建立数学模型，然后考虑数据的组织方式，设计合适的算法，并用某一种程序设计语言编写程序来实现算法，上机调试程序，使之运行后能产生求解问题的结果。显然，一个程序应包括对数据的描述与对运算操作的描述两个方面的内容。

著名计算机科学家沃思（Nikiklaus Wirth）就此提出一个公式：

数据结构+算法=程序　　（1.4）

数据结构是对数据的描述，而算法是对运算操作的描述。

【例 1.4】 程序实现求两个整数 $a,b(a>b)$的最大公约数(a,b)的欧几里德算法（见例 1.1），并应用欧几里德算法求 n 个整数 $m_1,m_2,\cdots,m_n$ 的最大公约数$(m_1,m_2,\cdots,m_n)$。

解：在欧几里德算法描述基础上进行数据描述即为求整数的最大公约数的程序。

（1）求两个整数的最大公约数程序实现

设置算法中的相关变量 a,b,c,r 为长整型变量，即有

```
/* 求整数 a,b 的最大公约数（a,b）*/
#include<stdio.h>
void main()
{ long a,b,c,r;
  printf("请输入整数 a,b: ");
  scanf("%ld,%ld",&a,&b);                    /*  输入整数 a,b */
  printf("(%ld,%ld)",a,b);
  if(a<b)
    {c=a;a=b;b=c;}                           /*  交换 a,b，确保 a>b*/
  r=a%b;
  while(r!=0)
    {a=b;b=r;                                /*  实施"辗转相除" */
     r=a%b;
    }
  printf("=%ld\n",b);                        /*  输出求解结果 */
}
```

程序运行示例：

```
请输入整数 a,b: 10920,21420
      (10920,21420)=420
```

（2）求 n 个整数的最大公约数程序实现

对于 3 个或 3 个以上整数，最大公约数有以下性质：

(a1,a2,a3)=((a1,a2),a3)

(a1,a2,a3,a4)=((a1,a2,a3),a4),…

应用这一性质，要求 *n* 个数的最大公约数，先求出前 *n*−1 个数的最大公约数 *b*，再求第 *n* 个数与 *b* 的最大公约数；要求 *n*−1 个数的最大公约数，先求出前 *n*−2 个数的最大公约数 *b*，再求第 *n*−1 个数与 *b* 的最大公约数；依次类推。因而，要求 *n* 个整数的最大公约数，需应用 *n*−1 次欧几里德算法。

为输入与输出方便，把 *n* 个整数设置成 *m* 数组，*m* 数组与算法中的相关变量 *a*,*b*,*c*,*r* 设置为长整型变量，个数 *n* 与循环变量 *k* 设置为整型。即有：

```
/* 求 n 个整数的最大公约数*/
#include<stdio.h>
void main()
{ int k,n;
  long a,b,c,r,m[100];
  printf("请输入整数个数 n: ");                    /* 输入原始数据 */
  scanf("%d",&n);
  printf("请依次输入%d 个整数: ",n);
  for(k=0;k<=n-1;k++)
    { printf("\n 请输入第%d 个整数：  ",k+1);
      scanf("%ld",&m[k]);
    }
  b=m[0];
  for(k=1;k<=n-1;k++)                              /* 控制应用 n-1 次欧几里德算法 */
  {a=m[k];
  if(a<b)
    {c=a;a=b;b=c;}                                 /*  交换 a,b，确保 a>b*/
  r=a%b;
  while(r!=0)
    { a=b;b=r;                                     /*  实施"辗转相除" */
        r=a%b;
     }
  }
  printf("(%ld",m[0]);                             /* 输出求解结果 */
  for(k=1;k<=n-1;k++)
    printf(",%ld",m[k]);
  printf(")=%ld\n",b);
}
```

程序运行示例：

```
请输入整数个数 n：4
请依次输入 4 个整数：
请输入第 1 个整数：10920
```

请输入第 2 个整数：11340

请输入第 3 个整数：21420

请输入第 4 个整数：17500

(10920,11340,21420,17500)=140

要提高程序的质量，提高编程效率，主要是使程序实现算法，具有良好的可读性、可靠性、可维护性以及良好的结构。设计好的算法，编制好的程序，应当是每位程序设计工作者追求的目标。而要做到这一点，就必须掌握正确的程序设计方法与技术。

实际上，算法设计与程序实现是相关联的一个整体。为了防止在算法教学中算法设计与程序实现脱节，算法理论与实际应用脱节。本书在讲述每一种常用算法时，把算法设计与程序实现紧密结合起来，突出算法在解决实际问题中的应用，努力提高对相应算法的理解，切实提高我们应用算法设计解决实际问题的能力。

1.3.2 结构化程序设计

近年来，一些面向对象的计算机程序设计语言陆续问世，打破了以往只有面向过程程序设计的单一局面。如果认为有了面向对象的程序设计之后，面向过程的程序设计就过时了，这是不正确的。不应该把面向对象与面向过程对立起来，在面向对象程序设计中仍然要用到面向过程的知识。面向过程程序设计仍然是程序设计工作者的基本功。而面向过程程序设计通常由结构化程序设计实现。

算法的实现过程是由一系列操作组成的，这些操作之间的执行次序就是程序的控制结构。1996 年，计算机科学家 Bohm 和 Jacopini 证明了这样的事实：任何简单或复杂的算法都可以由顺序结构、选择结构和循环结构这三种基本结构组合而成。所以，顺序结构、选择结构和循环结构被称为程序设计的三种基本结构，也是结构化程序设计必须采用的结构。

结构化程序设计方法是目前国内外普遍采用的一种程序设计方法。自 20 世纪 60 年代由荷兰学者 E.W.Dijkstra 提出后，结构化程序设计方法在实践中不断发展和完善，已成为软件开发的重要方法，在程序设计中占有十分重要的位置。

结构化程序设计是一种进行程序设计的原则和方法，按照这种原则和方法可设计出结构清晰、容易理解、容易修改、容易验证的程序。结构化程序设计是按照一定的原则与原理，组织和编写正确且易读的程序的软件技术。结构化程序设计的目标在于使程序具有一个合理结构，以保证程序的正确性，从而开发出正确、合理的程序。

结构化程序设计的基本要点为：

（1）自顶向下，逐步求精；

（2）模块化设计；

（3）结构化编码。

自顶向下是指对设计求解的问题要有一个全面的理解，从问题的全局入手，把一个复杂问题分解成若干个相互独立的子问题，然后对每个子问题再作进一步的分解，如此重复，直到每个子问题都容易解决为止。

逐步求精是指程序设计的过程是一个渐进的过程，先把一个子问题用一个程序模块来描

述，再把每个模块的功能逐步分解细化为一系列的具体步骤，以致能用某种程序设计语言的基本控制语句来实现。

逐步求精总是和自顶向下结合使用，将问题求解逐步具体化的过程，一般把逐步求精看作自顶向下设计的具体体现。

模块化是结构化程序设计的重要原则。所谓模块化就是把大程序按照功能分为若干个较小的程序。一般地讲，一个程序是由一个主控模块和若干子模块组成的。主控模块用来完成某些公用操作及功能选择，而子模块用来完成某项特定的功能。在 C 语言中，子模块通常用函数来实现。当然，子模块是相对主模块而言的，作为某一子模块，它也可以控制更下一层的子模块。这种设计风格，便于分工合作，将一个大的模块分解为若干个子模块分别完成。然后用主控模块控制、调用子模块。这种程序的模块化结构如图 1.1 所示。

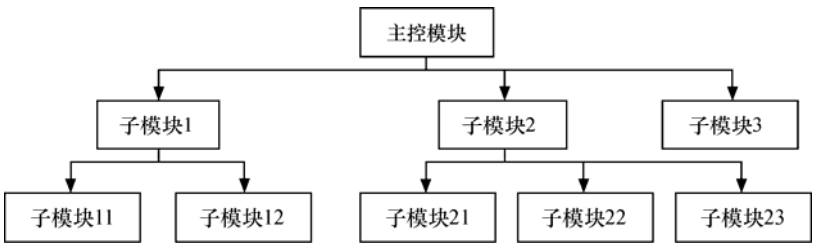


图 1.1　程序的模块化结构

在设计好一个结构化的算法之后，还需进行结构化编码，将已设计好的算法用计算机语言来表示，编写出能在计算机上进行编译与运行的程序。

结构化程序设计的过程就是将问题求解由抽象逐步具体化的过程。这种方法符合人们解决复杂问题的普遍规律，可以提高程序设计的质量和效率。

习　　题

1. 简述算法的定义与性质。
2. 当今计算机的计算速度已相当高了，为什么还要研究算法与改进算法？
3. 分别求出以下程序段所代表的算法的时间复杂度。

（1）
```
m=0;
for(k=1;k<=n;k++)
for(j=k;j>=1;j−)
        m=m+j;
```

（2）
```
m=0;
 for(k=1;k<=n;k++)
 for(j=1;j<=k/2;j++)
         m=m+j;
```

（3）
```
t=1;m=0;
for(k=1;k<=n;k++)
```

```
        {t=t*k;
         for(j=1;j<=k*t;j++)
             m=m++;
        }
（4）for(a=1;a<=n;a++)
      {s=0;
       for(b=a*100-1;b>=a*100-99;b-=2)
         {for(x=0,k=1;k<=sqrt(b);k+=2)
            if(b%k==0)
               {x=1;break;}
          s=s+x;
   }
     if(s==50)
       printf("%ld \n",a);break;}
}
```

4. 简述算法的多项式计算时间与指数计算时间的区别。

5. 若 $p(n)$是 n 的多项式，证明：$O(\log(p(n)))=O(\log n)$。

6. 试把例 1.1 中的欧几里德算法描述成函数形式，并设计程序通过函数调用实现求 n 个整数的最大公约数。

7. 简述程序设计的内容与方法。

8. 简述算法与程序的关系。

第 2 章　穷举与回溯

穷举法是计算机程序设计引导入门的基础算法，也是在数量较小的问题求解中应用广泛的算法。应用穷举设计可以非常简明地解决许多实际问题。与穷举的“笨拙”搜索相比，回溯法则是一种“聪明”的求解效益更高的搜索法。

本章介绍穷举与回溯两种算法及其应用。

2.1　穷举及其应用

通常程序设计入门都是从穷举设计开始的。今天，计算机的运算速度非常快，应用穷举设计程序可快捷地解决一般数量的实际应用问题。

2.1.1　穷举概述

穷举法又称列举法，其基本思想是逐一列举问题所涉及的所有情况，并根据问题提出的条件检验哪些是问题的解，哪些应予排除。

穷举法的特点是算法设计比较简单，解的可能为有限种，一一列举问题所涉及的所有情形。

穷举法常用于解决“是否存在”或“有多少种可能”等问题。其中许多实际应用问题靠人工推算求解是不可想象的，而应用计算机来求解，充分发挥计算机运算速度快、擅长重复操作的特点，穷举判断，快速简便。

应用穷举法时应注意对问题所涉及的有限种情形须一一列举，既不能重复，又不能遗漏。重复列举直接引发增解，影响解的个数的准确性；而列举的遗漏可能导致问题解的遗漏。

穷举通常应用循环结构来实现。在循环体中，根据所求解的具体条件，应用选择结构实施判断筛选，求得所要求的解。

穷举法的框架描述：

```
n=0;
for(k=<区间下限>;k<=<区间上限>;k++)        /* 根据指定范围实施穷举 */
    if(<约束条件>)                          /* 根据约束条件实施筛选 */
    { printf(<满足要求的解>);               /* 输出满足要求的解 */
      n++;                                  /* 统计解的个数 */
```

```
}
```

尽管穷举比较简单，在应用穷举设计求解实际问题时要认真分析与确定穷举范围与约束条件。

应用穷举设计求解，通常分以下几个步骤：

（1）根据问题的具体情况确定穷举量（简单变量或数组）；

（2）根据指定范围设置穷举循环；

（3）根据问题的具体要求确定筛选的约束条件；

（4）设计穷举程序并运行、调试，对运行结果进行分析与讨论。

当问题所涉及数量非常大时，穷举的工作量也就相应较大，程序运行时间也就相应较长。为此，应用穷举求解时，应根据问题的具体情况分析归纳，寻找简化规律，精简穷举循环，优化穷举策略。

2.1.2 穷举应用

穷举设计常通过循环结构来实现。下面从 3 个实例的设计求解说明穷举法的应用。

【例 2.1】 求最大公约数与最小公倍数。

试求两个已知正整数 *a,b* 的最大公约数与最小公倍数。为方便表述，记

(*a,b*)为正整数 *a,b* 的最大公约数;

{*a,b*}为正整数 *a,b* 的最小公倍数。

（1）算法设计

求两个整数 *a,b* 的最大公约数，例 1.4 采用“辗转相除”法求解实现。实际上，直接按最大公约与最小公倍的定义应用穷举设计求解，显得更为直观，也更为方便。

注意到 $1\leqslant(a,b)\leqslant\min(a,b)$，不妨设 $\min(a,b)=b$，则 *a,b* 的最大公约数 *k* 的穷举范围为 $k=b,b-1,\cdots,1$，最先满足公约数条件 $a\%k=0$ 且 $b\%k=0$ 的 *k* 即为(*a,b*);

同样，$max(a,b)\leqslant\{a,b\}\leqslant a*b$，不妨设 $max(a,b)=a$，则 *a,b* 的最小公倍数 *k* 的穷举范围为 $k=a,a*2,\cdots,a*b$,最先满足公倍数条件 $k\%a=0$ 且 $k\%b=0$ 的 *k* 即为{*a,b*}。

注意到两整数 *a,b* 的最小公倍数与最大公约数有以下简单关系式：

$\{a,b\}(a,b)=ab$

因而由求得的最大公约数(*a,b*)即可据上式求得最小公倍数{*a,b*}，同样由求得的最小公倍数{*a,b*}即可据上式求得最大公约数(*a,b*)。

（2）求最大公约数穷举设计

```
#include <stdio.h>
void main()
{  long a,b,k,t;
   printf("请输入正整数 a,b:");
   scanf("%ld,%ld",&a,&b);
   if(a<b)
     {t=a;a=b;b=t;}
   for(k=b;k>=1;k--)                          /* 实施穷举 */
```

```
    if(a%k==0 && b%k==0)                    /* 实施判别 */
      {printf("(%ld,%ld)=%ld \n",a,b,k);
       printf("{%ld,%ld}=%ld \n",a,b,a*b/k);
       break;                               /* 输出后须立即退出循环 */
      }
}
```

（3）求最小公倍数穷举设计

```
#include <stdio.h>
void main()
{  long a,b,k,t;
   printf("请输入正整数 a,b:");
   scanf("%ld,%ld",&a,&b);
   if(a<b)
    {t=a;a=b;b=t;}
   for(k=a;k<=a*b;k=k+a)                    /* 实施穷举 */
    if(k%a==0 && k%b==0)                    /* 实施判别 */
      {printf("{%ld,%ld}=%ld \n",a,b,k);
       printf("(%ld,%ld)=%ld \n",a,b,a*b/k);
       break;                               /* 输出后须立即退出循环 */
      }
}
```

（4）程序运行示例与说明

运行程序，输入 a,b:29682,8148，得

```
{29682,8148}=415548
(29682,8148)=582
```

注意，在求最大公约数时，循环设计由大至小穷举，最先所得的公约数当然是最大的。而在求最小公倍数时，循环设计由小至大穷举，最先所得的公倍数当然是最小的。

以上穷举法求最大公约数的时间复杂显然比“辗转相除法”（见例 1.1）要高，但直接穷举无需数论专业知识，直观明了，可读性好，且穷举的时间复杂度也不高，最坏情形时为 $O(n)$。

【例 2.2】 求真分数递增序列。

统计分母在区间[a,b]的最简真分数（分子小于分母，且分子分母无公因数）共有多少个，并求这些最简真分数升序序列中的第 k 项（正整数 a,b,k 从键盘输入）。

（1）算法设计

为排序方便，设置数组 c 存储分子，数组 d 存储分母。真分数升序排序后的第 k 项为 $c(k)/d(k)$。

在指定范围[a,b]穷举分数 i/j 的分母 j: a,a+1,…,b；

对每一个分母 j 穷举分子 i: 1,2,…,j−1。

若分子 i 与分母 j 存在大于 1 的公因数，说明 i/j 非最简，忽略不计；否则赋值得一个最

简真分数 $c(n)/d(n)$。数组下标 n 统计最简真分数的个数。应用冒泡法排序后即可打印出指定的第 k 项 $c(k)/d(k)$。

（2）求最简真分数增序列程序设计

```
/* 求分母为[a,b]的最简真分数的增序列 */
#include <stdio.h>
void main()
{int a,b,k,n,i,j,h,t,u,c[3000],d[3000];
 printf("请依次输入 a,b,k:");
 scanf("%d,%d,%d",&a,&b,&k);                    /* 输入区间的上下限与指定序号 k */
 n=0;
 for(j=a;j<=b;j++)
 {for(i=1;i<=j-1;i++)
    {for(t=0,u=2;u<=i;u++)
       if(j%u==0 && i%u==0) {t=1;break;}         /* 分子分母有公因数舍去 */
     if(t==0) {n++;c[n]=i;d[n]=j;}
    }
  }
 for(i=1;i<=n-1;i++)                            /* 应用冒泡法排序 */
    for(j=1;j<=n-i;j++)
      if(c[j]*d[j+1]>c[j+1]*d[j])
        {h=c[j];c[j]=c[j+1];c[j+1]=h;            /* 排序分子分母同时交换 */
         h=d[j];d[j]=d[j+1];d[j+1]=h;}
 printf("n=%d \n",n);
 printf("第%d 项为:%d/%d \n",k,c[k],d[k]);
}
```

运行程序，输入 a=10,b=99,k=1000，得：

```
n=2976
第 1000 项为: 27/80
```

【例 2.3】 已知集合 A 定义如下:

（1）$1 \in A$，$2 \in A$

（2）$x,y \in A \Rightarrow 2x+3y \in A$

（3）再无其他数属于 A。

试求集合 A 中元素从小到大排列的序列的前 n 项。

（1）按第 n 项的大小循环设计

递推关系 $2x+3y$ 看似简单，实际上非常复杂。因 x,y 可以是已产生的所有已有项中的任意两项，已产生项越多，递推生成的新项也就越多。同时，递推产生的项大小也是摆动的，并无规律可循。

穷举循环变量 k 从 3 开始递增 1 取值，到第 n 项时 k 的终值尚无法确定。可约定一个较大的终值（例如 10 000 或更大）。若 k 可由已有的项 $a(j)$，$a(i)$ $(j<i)$推得，即若 k 满足条件

k=2∗a(j)+3∗a(i)或 k=2∗a(i)+3∗a(j)，说明 k 是 a 数列中的一项，赋值给 a(t)。当项数 t 达到规定项数 n 时，则退出穷举。

（2）按项的大小循环设计程序实现

```
/*   2x+3y 按项的大小循环设计   */
#include <stdio.h>
void main()
 {int n,t,k,i,h,j, a[30000];
  printf("请输入 n: "); scanf("%d",&n);
  a[1]=1;a[2]=2;t=2;
  for(k=3;k<=10000;k++)
     {h=0;for(i=2;i<=t;i++)
     {for(j=1;j<=i−1;j++)
        if(k==2∗a[j]+3∗a[i] || k==2∗a[i]+3∗a[j]) /∗ 判断 k 为递推项 ∗/
           {h=1;t++;a[t]=k;
            if(k==2∗a[j]+3∗a[i])
               {printf("%3d(%3d)=2∗%2d+3∗%2d    ",k,t,a[j],a[i]);break;}
            if(k==2∗a[i]+3∗a[j])
               {printf("%3d(%3d)=2∗%2d+3∗%2d    ",k,t,a[i],a[j]);break;}
           }
       if(h==1) break;}
       if(t==n) break;}
 }
```

（3）按项数循环设计

已知前 2 项，t 循环从 3 开始递增 1 取值到指定的项数 n。第一项的值 k 从 2 开始递增取值，对每一个 k 取值，标记 h=0 赋值；若 k 可由已有的项 a(j),a(i) (j＜i＝推得，即若 k 满足条件 k=2∗a(j)+3∗a(i) 或 k=2∗a(i)+3∗a(j)，说明 k 是 a 数列中的一项，赋值给 a(t)，同时标记 h=1 赋值。对某项数 t 若 h=0 时，则 t 减 1 后循环，即对于原 t 使 k 增值后继续，直到达到规定项数 n 为止。

（4）按项数循环设计程序实现

```
#include <stdio.h>
void main()
 {int n,t,k,i,h,j,a[30000];
  printf("请输入 n: "); scanf("%d",&n);
  a[1]=1;a[2]=2;k=2;
  for(t=3;t<=n;t++)
     {k++;h=0;
      for(i=2;i<=t−1;i++)
         {for(j=1;j<=i−1;j++)
            if(k==2∗a[j]+3∗a[i] || k==2∗a[i]+3∗a[j])       /∗ 判断 k 为递推项 ∗/
```

```
                {h=1;a[t]=k;
                 if(k==2*a[j]+3*a[i])
                   {printf("%3d(%3d)=2*%2d+3*%2d   ",k,t,a[j],a[i]);break;}
                 if(k==2*a[i]+3*a[j])
                   {printf("%3d(%3d)=2*%2d+3*%2d   ",k,t,a[i],a[j]);break;}
                }
            if(h==1) break;
            }
        if(h==0) t--;
        }
}
```

运行程序，输入 20，得

```
  7( 3)=2* 2+3* 1      8( 4)=2* 1+3* 2     17( 5)=2* 7+3* 1     19( 6)=2* 8+3* 1
 20( 7)=2* 7+3* 2     22( 8)=2* 8+3* 2     23( 9)=2* 1+3* 7     25(10)=2* 2+3* 7
 26(11)=2* 1+3* 8     28(12)=2* 2+3* 8     37(13)=2* 8+3* 7     38(14)=2* 7+3* 8
 40(15)=2*17+3* 2     41(16)=2*19+3* 1     43(17)=2*20+3* 1     44(18)=2*19+3* 2
 46(19)=2*20+3* 2     47(20)=2*22+3* 1
```

上述程序不仅能计算并输出数列的前 n 项，而且能表明每一项是如何递推得到的。

2.2 穷举设计的优化

穷举设计求解一些数量大的问题时，为使穷举易于实现，可根据问题的具体实际进行必要的优化，精简一些不必要的操作，从而缩减穷举求解的时间。

2.2.1 优选穷举对象

面对一个具体问题的穷举设计，选择合适的穷举对象是穷举优化的首要环节。

【例 2.4】 把一个 6 位整数分为前后两个 3 位数，若该数等于所分两个 3 位数和的平方，则称该数为 6 位分段和平方数。试求出所有 6 位分段和平方数。

（1）对所有 6 位数穷举

穷举对象选为 6 位整数。在 6 位整数 a 的穷举循环中，检验若 a 是一个平方数，应用除法与求余运算把 a 分段为前后两个 3 位整数 x、y，通过条件判别：若满足 $a=(x+y)^2$，即找到分段和平方数，作打印输出。

穷举程序设计为:

```
#include <stdio.h>
#include <math.h>
void main()
```

```
{long int a,b,x,y;
 printf("6 位分段和平方数有: ");
 for(a=100000;a<=999999;a++)                    /* 设置 a 穷举所有 6 位数 */
     {b=sqrt(a);
      if(a==b*b)                                /* 若 a 是一个平方数则进行分解 */
        {x=a/1000; y=a%1000;                    /* 6 位数 a 分为前后两个 3 位数 */
         if(b==x+y)                             /* 分段和条件检验 */
            printf("%ld   ",a);
        }
      }
 }
```

（2）对 3 位数穷举

穷举对象选为部分 3 位整数 b，b 的平方为一个 6 位整数。在 3 位整数 b 的循环中，求出 $a=b*b$,平方数 a 分段为前后两个 3 位整数 x、y，应用 3 位数 b 是否等于 $x+y$ 作判别。

穷举循环程序设计：

```
#include <stdio.h>
#include <math.h>
void main()
{long int t,b,x,y,a;
 printf("6 位分段和平方数有: ");
 t=sqrt(100000);
 for(b=t+1;b<=999;b++)                          /* 设置 b 穷举所有三位数 */
     {a=b*b;
      x=a/1000; y=a%1000;                       /* 6 位平方数 a 分为前后两个 3 位数 */
      if(b==(x+y))                              /* 分段和条件检验 */
         printf("%ld   ",a);
      }
 }
```

运行程序，得

```
6 位分段和平方数有:    494209      998001
```

（3）总结与思考

以上两个穷举循环设计，显然选择 3 位数作为穷举对象是合理的，时间复杂度由 $O(n)$优化为 $O(\sqrt{n})$，且省去了平方数的判别。

思考：把以上穷举应用在 4 位数，可得 9 801 为 4 位分段和平方数；上面得到 998 001 为 6 位分段和平方数。我们猜想，99 980 001，9 999 800 001 等是否也是相应的分段和平方数？请从数学上证明：99…9800…01（其中有连续 n 个 9，连续 n 个 0）是一个 2n+2 位分段和平方数。

2.2.2 优化穷举循环参量

穷举对象确定后，通常应用循环来实现穷举。优化穷举循环参量可有效减少或消除无效循环，提高穷举效率。

【例 2.5】 把某一指定整数分解为质因数之积。要求把整数表示为质因数从小到大顺序排列的乘积形式。如果被分解的数本身是素数，则予以注明。

例如，90=2*3*3*5, 91=素数。

（1）优化算法设计

对被分解的整数 *n*，赋值给 *b*（以保持判别运算过程中 *n* 不变），用 *k*（从 2 开始递增 1 取值）试商：

若 *k* 不能整除 *b*，说明该数 *k* 不是 *b* 的因数，*k* 增 1 后继续试商。

若 *k* 能整除 *b*，说明该数 *k* 是 *b* 的因数，打印输出“*k**”，*b* 除以 *k* 的商赋给 *b*(*b*=*b*/*k*) 后继续用 *k* 试商（注意，可能有多个 *k* 因数），直至不能整除，*k* 增 1 继续。

按上述从小至大试商确定的因数为 *n* 的质因数。

如果整个试商后 *b* 的值没有任何缩减，仍为原待分解数 *n*，说明 *n* 是素数，作素数说明标记。

（2）质因数分解乘积形式 C 程序实现

```
/* 质因数分解乘积形式 */
#include <stdio.h>
#include <math.h>
void main()
{  long int b,k,n;
   printf("整数 n 分解质因数.请输入 n:");
   scanf("%ld",&n);
   printf("%ld=",n);
   b=n;k=2;
   while(k<=sqrt(n))                    /* k 为试商因数，终值取为 sqrt(n) */
     {if(b%k==0)
        {b=b/k;
          if(b>1)
            {printf("%ld*",k);continue;}     /* k 为质因数,返回再试 */
          if(b==1) printf("%ld \n",k);
        }
      k++;
        }
 if(b>1 && b<n) printf("%ld\n",b);          /* 输出大于 n 平方根的因数 */
 if(b==n) printf("(素数!)\n");              /* b=n,表示 n 无质因数 */
}
```

运行程序，输入 n=20072008，得

20072008=2*2*2*11*23*47*211

（3）说明与注意

试商 k 循环终值如何确定，一定程度上决定了程序的效率。终值定为 $n-1$，或 $n/2$ 是可行的，但试商循环次数都较大，无效循环太多。优化循环终值定为 n 的平方根 $sqrt(n)$，这样可大大精简试商的次数，时间复杂度由 $O(n)$优化为 $O(\sqrt{n})$。

例如，n=1 000 000 时，终值定为 $n-1$ 需循环 999 999 次；终值定为 $n/2$ 需循环 500 000 次；而循环终值定为 n 的平方根 $sqrt(n)$，只需循环 1 000 次。

当然，对于大于 $sqrt(n)$的因数（至多一个），在试商循环结束后一定要注意补上，不要遗失。

【例 2.6】 求解高斯 8 皇后问题。

8 皇后问题是数学家高斯（Gauss）于 1850 年提出的趣题:在国际象棋的 8×8 方格的棋盘上如何放置 8 个皇后，使得这 8 个皇后不能相互攻击，即任意 2 个皇后不允许处在同一横排，同一纵列，也不允许处在同一与棋盘边框成 45° 角的斜线上。

（1）算法设计

高斯 8 皇后问题的一个解用一个 8 位数表示，8 位数解的第 k 个数字为 j，表示棋盘上的第 k 行的第 j 格放置一个皇后。

任意两个皇后不允许处在同一横排，同一纵列，要求 8 位数中数字 1～8 各出现一次，不能重复。因而解的范围区间应为[12345678,87654321]。注意到数字 1～8 的任意一个排列的数字和为 9 的倍数，即数字 1～8 的任意一个排列均为 9 的倍数，因而穷举 a 循环的穷举范围定为[12345678, 87654321]，其循环步长可优化为 9。

为了判别数字 1～8 在 8 位数 a 各出现一次，设置 f 数组，$f(x)$统计 a 中数字 x 的个数。若 $f(1)$～$f(8)$均等于 1，即数字 1～8 在 a 中各出现 1 次。否则，返回测试下一个 8 位数 a。

任意两个皇后不允许处在同一与棋盘边框成 45° 角的斜线上，设置 g 数组，若 a 的第 k 个数字为 x，则 $g(k)=x$。要求解的 8 位数的第 j 个数字与第 k 个数字的绝对值不等于 $j-k$(设置 $j>k$)。若出现

$$|g(j)-g(k)| = j-k$$

表明 j 与 k 出现同处在与棋盘边框成 45° 角的斜线上，返回测试下一个 8 位数 a。

在穷举范围内均通过以上两道筛选的 8 位数即为一个解，打印输出（每行打印 6 个解），同时用变量 s 统计解的个数。

（2）穷举求解高斯 8 皇后问题 C 程序实现

```
/* 穷举求解高斯 8 皇后问题 */
  #include <stdio.h>
  #include <math.h>
  void main()
 {int s,k,i,j,t,x,f[9],g[9];
  long a,y;
  s=0;
  printf("高斯 8 皇后问题的解为：\n");
  for(a=12345678;a<=87654321;a=a+9)        /* 步长为 9 穷举八位数 */
    {y=a;k=0;
```

```
    for(i=1;i<=8;i++) f[i]=0;
    while(y>0)
      {x=y%10;f[x]=f[x]+1;y=y/10;
       k++;g[k]=x;}                          /*  分离各数字,用 f 数组统计  */
    for(t=0,i=1;i<=8;i++)
      if(f[i]!=1) t=1;                       /*  数字 1--8 出现不为 1 次，返回  */
    if(t==1) continue;
    for(k=1;k<=7;k++)                        /*  同处在 45 度角的斜线上，返回  */
    for(j=k+1;j<=8;j++)
       if (fabs(g[j]-g[k])==j-k) t=1;
    if(t==1) continue;
    s++;                                     /*  输出 8 皇后问题的解 */
    printf("%ld   ",a);
    if(s%6==0) printf("\n");
    }
  printf("\n s=%d. ",s);
  }
```

（3）程序运行与说明

运行程序，得

```
15863724   16837425   17468253   17582463   24683175   25713864
25741863   26174835   26831475   27368514   27581463   28613574
31758246   35281746   35286471   35714286   35841726   36258174
36271485   36275184   36418572   36428571   36814752   36815724
36824175   37285146   37286415   38471625   41582736   41586372
42586137   42736815   42736851   42751863   42857136   42861357
46152837   46827135   46831752   47185263   47382516   47526138
47531682   48136275   48157263   48531726   51468273   51842736
51863724   52468317   52473861   52617483   52814736   53168247
53172864   53847162   57138642   57142863   57248136   57263148
57263184   57413862   58413627   58417263   61528374   62713584
62714853   63175824   63184275   63185247   63571428   63581427
63724815   63728514   63741825   64158273   64285713   64713528
64718253   68241753   71386425   72418536   72631485   73168524
73825164   74258136   74286135   75316824   82417536   82531746
83162574   84136275
 s=92
```

以上穷举设计把循环步长定为 9 的优化处理，可把穷举循环的次数缩减到 1/9。

穷举法求解，程序设计比较简单，但当 n 较大时速度较慢，本章后面介绍的回溯求解速度则较快。

2.2.3　精简穷举循环

应用穷举求解实际问题，精简一些不必要的循环，可大大提高穷举的效率，缩减穷举求解的时间。

【例 2.7】 整币兑零求解。

计算把一张 1 元整币兑换成 1 分、2 分、5 分、1 角、2 角和 5 角共 6 种零币的不同兑换种数。

进一步计算把一张 2 元整币与一张 5 元整币兑换成上述 6 种零币的不同兑换种数。

（1）穷举设计求解

设整币的面值为 n 个单位，面值为 1、2、5、10、20、50 单位零币的个数分别为 $p1, p2, p3, p4, p5, p6$。显然需解一次不定方程

$p1+2*p2+5*p3+10*p4+20*p5+50*p6=n$　　($p1, p2, p3, p4, p5, p6$ 为非负整数)

对这 6 个变量实施穷举，确定穷举范围为 $0 \leqslant p1 \leqslant n$, $0 \leqslant p2 \leqslant n/2$, $0 \leqslant p3 \leqslant n/5$, $0 \leqslant p4 \leqslant n/10$, $0 \leqslant p5 \leqslant n/20$, $0 \leqslant p6 \leqslant n/50$。

在以上穷举的 6 重循环中，若满足条件

$$p1+2*p2+5*p3+10*p4+20*p5+50*p6=n$$

则为一种兑零方法，输出结果并通过变量 m 增 1 统计不同的兑换种数。

（2）穷举求解 C 程序设计

```
/*   整币兑零穷举设计 1   */
#include <stdio.h>
void main()
{int p1,p2,p3,p4,p5,p6,n; long m=0;
 printf("\n n=");scanf("%d",&n);
 printf("      1 分   2 分   5 分   1 角   2 角   5 角 \n");
 for(p1=0;p1<=n;p1++)
 for(p2=0;p2<=n/2;p2++)
 for(p3=0;p3<=n/5;p3++)
 for(p4=0;p4<=n/10;p4++)
 for(p5=0;p5<=n/20;p5++)
 for(p6=0;p6<=n/50;p6++)
   if(p1+2*p2+5*p3+10*p4+20*p5+50*p6==n)     /* 根据条件检验 */
       {m++;
        printf(" %5d%5d%5d",p1,p2,p3);
        printf("%5d%5d%5d\n",p4,p5,p6);
       }
 printf(" %d(1,2,5,10,20,50)=%ld \n",n,m);
}
```

运行程序，输入 100，即得 1 元整币兑换成 1 分、2 分、5 分、1 角、2 角、5 角共 6 种

零币的不同兑换方法及种数为：

1 分	2 分	5 分	1 角	2 角	5 角
0	0	0	0	0	2
0	0	0	0	5	0
0	0	0	1	2	1

……

100(1,2,5,10,20,50)=4562

共有 4 562 个解，即有 4 562 种不同的兑换种数。

（3）精简穷举循环设计

在上述程序的 6 重循环中，我们可精简 $p1$ 循环，在循环内应用

$$p1=n-(2*p2+5*p3+10*p4+20*p5+50*p6)$$

给 $p1$ 赋值。如果 $p1$ 为非负数，对应一种兑换法。

```
/* 精简循环设计 2  */
#include <stdio.h>
void main()
{int p1,p2,p3,p4,p5,p6,n; long m=0;
 printf("\n n=");scanf("%d",&n);
 printf("      1 分   2 分   5 分   1 角   2 角   5 角 \n");
 for(p2=0;p2<=n/2;p2++)                              /* 已精简了 p1 循环 */
 for(p3=0;p3<=n/5;p3++)
 for(p4=0;p4<=n/10;p4++)
 for(p5=0;p5<=n/20;p5++)
 for(p6=0;p6<=n/50;p6++)
   {p1=n-(2*p2+5*p3+10*p4+20*p5+50*p6);      /* p1 为一分币的个数 */
    if(p1>=0)
       {m++;
        printf(" %5d%5d%5d",p1,p2,p3);
        printf("%5d%5d%5d\n",p4,p5,p6);
       }
   }
 printf(" %d(1,2,5,10,20,50)=%ld \n",n,m);
}
```

运行程序，输入 200，即得 2 元整币兑换成 1 分、2 分、5 分、1 角、2 角、5 角共 6 种零币的不同兑换种数为

200(1,2,5,10,20,50)=69118

精简穷举循环设计使穷举循环次数缩减为以上设计的 1/n。

（4）穷举设计的进一步优化

以上程序的循环次数已经大大精简了。进一步的分析，我们看到在程序的循环设置中，$p3$ 循环从 0～n/5 可改进为 0～(n−2*$p2$)/5，因为在 n 中 $p2$ 已占去了 2*$p2$。依次类推，对 $p4$,

*p*5, *p*6 的循环可作类似的循环参量优化。

```
/* 优化穷举设计 3 */
#include <stdio.h>
void main()
{int p1,p2,p3,p4,p5,p6,n; long m=0;
 printf("\n n=");scanf("%d",&n);
 printf("      1 分   2 分   5 分   1 角   2 角   5 角 \n");
 for(p2=0;p2<=n/2;p2++)
 for(p3=0;p3<=(n-2*p2)/5;p3++)          /* 缩减 p3,p4,p5,p6 循环范围 */
 for(p4=0;p4<=(n-2*p2-5*p3)/10;p4++)
 for(p5=0;p5<=(n-2*p2-5*p3-10*p4)/20;p5++)
 for(p6=0;p6<=(n-2*p2-5*p3-10*p4-20*p5)/50;p6++)
     {p1=n-(2*p2+5*p3+10*p4+20*p5+50*p6);
      if(p1>=0)
        {m++;
         printf(" %5d%5d%5d",p1,p2,p3);
         printf("%5d%5d%5d\n",p4,p5,p6);
        }
     }
 printf(" %d(1,2,5,10,20,50)=%ld \n",n,m);
}
```

运行程序，输入 *n*=500，即得 5 元整币兑换成 1 分、2 分、5 分、1 角、2 角、5 角共 6 种零币的不同兑换种数为

500(1,2,5,10,20,50)=3937256

（5）以上 3 个穷举设计比较

以上 3 个穷举求解程序体现了程序的改进与优化过程，因循环设置的差异与循环参量的不同，直接影响程序求解的速度。

以上穷举设计的时间复杂度为 $O(n^6)$，精简循环改进设计与进一步优化穷举设计的时间复杂度为 $O(n^5)$，测试 3 个穷举算法中条件判断语句执行次数如表 2.1 所示。

表 2.1　　3 个穷举算法中条件判断语句执行次数比较

n 取值	100	200	500
整币兑零穷举设计 1	21 417 858	961 353 855	1.8e+11
精简循环设计 2	212 058	4 782 855	369 769 686
优化穷举设计 3	4 562	69 118	3 937 256

由表 2.1 所列举的数据可见，求解同一个问题的穷举设计，精简循环的求解时间缩减为原穷举设计的数百分之一，而优化算法的求解时间缩减为精简循环设计的数百分之一。由此可见，算法的改进与优化对于提高求解效率的作用。

由于穷举计算所需的时间随 *n* 增加而迅速增加，以致当 *n* 比较大或所兑零币种数增多时

求解变得无望，改进算法才能快速解决整币兑零种数问题。

2.3 回溯法及其描述

在了解穷举搜索设计之后，下面介绍回溯设计及其应用，体会回溯法的特点与优势。

2.3.1 回溯的基本概念

有许多问题，当需要找出它的解集或者要求回答什么解是满足某些约束条件的最佳解时，往往要使用回溯法。

回溯法是一种试探求解的方法：通过对问题的归纳分析，找出求解问题的一个线索，沿着这一线索往前试探，若试探成功，即得到解；若试探失败，就逐步往回退，换其他路线再往前试探。因此，回溯法可以形象地概括为“向前走，碰壁回头”。

回溯法的基本做法是试探搜索，是一种组织得井井有条的、能避免一些不必要搜索的穷举式搜索法。这种方法适用于一些组合数比较大的问题。回溯算法在问题的解空间树中，从根结点出发搜索解空间树，搜索至解空间树的任意一点，先判断该结点是否包含问题的解；如果肯定不包含，则跳过对该结点为根的子树的搜索，逐层向其父结点回溯；否则，进入该子树，继续搜索。

从解的角度理解，回溯法将问题的候选解按某种顺序进行枚举和检验。当发现当前候选解不可能是解时，就选择下一个候选解；倘若当前候选解除了不满足问题规模要求外，满足所有其他要求时，继续扩大当前候选解的规模，并继续试探。如果当前候选解满足包括问题规模在内的所有要求时，该候选解就是问题的一个解。在回溯法中，放弃当前候选解，寻找下一个候选解的过程称为回溯。

与穷举法相比，回溯法的“聪明”之处在于能适时“回头”，若再往前走不可能得到解，就回溯，退一步另找线路，这样可省去大量的无效操作。因此，回溯与穷举相比，回溯更适宜于量比较大，候选解比较多的问题。

应用回溯设计求解实际问题，由于解空间的结构差异，很难计算与估计回溯产生的结点数，因此回溯计算复杂度是分析回溯法效率时遇到的主要困难。回溯法产生的结点数通常只有解空间结点数的一小部分，这也是回溯法的计算效率大大高于穷举法的原因所在。

2.3.2 回溯法描述

为了具体说明回溯的实施过程，先看一个简单实例。

1. 4 皇后问题回溯实施与求解过程

如何在 4×4 的方格棋盘上放置 4 个皇后，使它们互不攻击，即任意两个皇后不允许处在同一横排，同一纵列，也不允许处在同一与棋盘边框成 45° 角的斜线上。

图 2.1 所示为应用回溯的实施过程，其中方格中的×表示试图在该方格放置一个皇后但由于受前面已放置的皇后的攻击而放弃的位置。

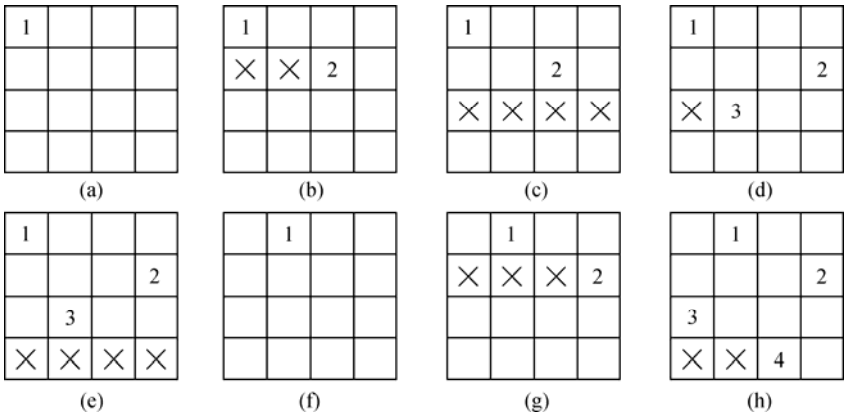


图 2.1　4 皇后问题回溯实施求解

图（a）为在第 1 行第 1 列放置一个皇后的初始状态。

图（b）中，第 2 个皇后不能放在第 1、2 列，因而放置在第 3 列上。

图（c）中，表示第 3 行的所有各列均不能放置皇后，则返回第 2 行，第 2 个皇后需后移。

图（d）中，第 2 个皇后后移到第 4 列，第 3 个皇后放置在第 2 列。

图（e）中，第 4 行的所有各列均不能放置皇后，则返回第 3 行；第 3 个皇后后移的所有位置均不能放置皇后，则返回第 2 行；第 2 个皇后已无位可退，则返回第 1 行；第 1 个皇后需后移。

图（f）中，第 1 个皇后后移至第 2 格。

图（g）中，第 2 个皇后不能放在第 1，2，3 列，因而放置在第 4 列上。

图（h）中，第 3 个皇后放在第 1 列；第 4 个皇后不能放置 1，2 列，于是放置在第 3 列。

这样经以上回溯，得到 4 皇后问题的一个解：2413。

继续以上的回溯探索，可得 4 皇后问题的另一个解：3142。

2. 回溯的一般方法

下面简要阐述回溯的一般方法。

回溯求解的问题 P，通常要能表达为：对于已知的由 n 元组$(x_1,x_2,\cdots,x_n)$组成的一个状态空间 $E=\{(x_1,x_2,\cdots,x_n)|x_i\in s_i,i=1,2,\cdots,n\}$，给定关于 n 元组中的一个分量的一个约束集 D，要求 E 中满足 D 的全部约束条件的所有 n 元组。其中 s_i 是分量 x_i 的定义域，且$|s_i|$有限，$i=1,2,\cdots,n$。称 E 中满足 D 的全部约束条件的任一 n 元组为问题 P 的一个解。

解问题 P 的最朴素的方法就是穷举法，上面已经作了介绍，即对 E 中的所有 n 元组逐一地检验其是否满足 D 的全部约束，若满足，则为问题 P 的一个解。显然，其计算量是相当大的。

对于许多问题，所给定的约束集 D 具有完备性，即 i 元组$(x_1,x_2,\cdots,x_n)$满足 D 中仅涉及 $x_1,x_2,\cdots,x_i$ 的所有约束，意味着 $j(j<i)$元组$(x_1,x_2,\cdots,x_j)$一定也满足 D 中仅涉及 $x_1,x_2,\cdots,x_j$ 的所有

约束，i=1,2,…,n。换句话说，只要存在 0≤j≤n−1，使得(x_1,x_2,…,x_j)违反 D 中仅涉及 x_1,x_2,…,x_j 的约束之一，则以(x_1,x_2,…,x_j)为前缀的任何 n 元组(x_1,x_2,…,x_j,x_{j+1},…,x_n)也一定违反 D 中仅涉及 x_1,x_2,…,x_i 的一个约束，n≥i>j。因此，对于约束集 D 具有完备性的问题 P，一旦检测断定某个 j 元组(x_1,x_2,…,x_j)违反 D 中仅涉及 x_1,x_2,…,x_j 的一个约束，就可以肯定，以(x_1,x_2,…,x_j)为前缀的任何 n 元组(x_1,x_2,…,x_j,x_{j+1},…,x_n)都不会是问题 P 的解，因而就不必去搜索它们，即省略了对部分元素(x_{j+1},…,x_n)的操作与测试。回溯法正是针对这类问题，利用这类问题的上述性质而提出来的比穷举法效率更高的算法。

3. 回溯算法框架描述

对于一般含参量 m,n 的搜索问题，回溯法框架描述如下：

```
/* 输入正整数 n,m,(n≥m) */
  i=1;a[i]=<元素初值>;
  while (1)
{
    g=1;for(k=i−1;k>=1;k−−)
        if( <约束条件 1> ) g=0;                    /* 检测约束条件,不满足则返回 */
        if(g && <约束条件 2>)
            printf(a[1−m]);                        /* 输出一个解 */
        if(i<n && g) {i++;a[i]=<取值点>;continue;}
        while(a[i]==<回溯点> && i>1) i−−;          /* 向前回溯 */
        if(a[i]==n && i==1) break;                 /* 退出循环，结束 */
        else a[i]=a[i]+1;
}
```

具体求解问题的试探搜索范围与要求不同，在应用回溯设计时，需根据问题的具体实际确定数组元素的初值、取值点与回溯点，同时需把问题中的约束条件进行必要的分解，以适应上述回溯流程。

其中实施向前回溯的循环

```
while(a[i]==<回溯点> && i>1) i−−;
```

是向前回溯一步，还是回溯两步或更多步，完全根据 $a[i]$是否达到回溯点来确定。例如，回溯点是 n，i=6，当 a[6]=n 时回溯到 i=5；若 a[5]=n 时回溯到 i=4；依次类推，到 $a[i]$达到回溯点则停止。

图 2.1 所示的 4 皇后问题回溯求解过程描述为：

```
n=4;i=1;a[i]=1;
while (1)
  {
      g=1;for(k=i-1;k>=1;k−−)
      if(a[i]=a[k] && abs(a[i]−a[k])=i−k) g=0;
                                                 /* 检测约束条件,不满足则返回 */
       if(g && i==4)
```

```
            printf(a[1−4]);                    /* 输出一个解 */
        if(i<n && g) {i++;a[i]=1;continue;}
        while(a[i]==n && i>1) i−−;             /* 向前回溯 */
        if(a[i]==n && i==1) break;             /* 退出循环结束 */
        else a[i]=a[i]+1;
}
```

2.3.3　回溯法的效益分析

回溯求解过程实质上是一个遍历一棵“状态树”的过程，只是这棵树不是遍历前预先建立的。回溯算法在搜索过程中，只要所激活的状态结点满足终结条件，应该把它输出或保存。由于在回溯法求解问题时，一般要求输出问题的所有解，因此在得到结点后，同时也要进行回溯，以便得到问题的其他解，直至回溯到状态树的根且根的所有子结点均已被搜索过为止。

组织解空间便于算法在求解集时更易于搜索，典型的组织方法是图或树。一旦定义了解空间的组织方法，这个空间即可从开始结点进行搜索。

回溯法的时间通常取决于状态空间树上实际生成的那部分问题状态的数目。对于元组长度为 n 的问题，若其状态空间树中结点总数为 $n!$，则回溯算法的最坏情形的时间复杂度可达 $O(p(n)n!)$；若其状态空间树中结点总数为 2^n，则回溯算法的最坏情形的时间复杂度可达 $O(p(n)2^n)$，其中 $p(n)$为 n 的多项式。

对于不同的实例，回溯法的计算时间有很大的差异。对于很多具有大 n 的求解实例，应用回溯法一般可在很短的时间内求得其解，可见回溯法不失为一种快速有效的算法。

对于某一具体实际问题的回溯求解，常通过计算实际生成结点数的方法即蒙特卡罗方法（Monte carlo）来评估其计算效率。蒙特卡罗方法的基本思想是在状态空间树上随机选择一条路径$(x_0,x_1,\cdots,x_{n-1})$，设 X 是这一路径上部分向量$(x_0,x_1,\cdots,x_{k-1})$的结点，如果在 X处不受限制的子向量数是 m_k，则认为与 X 同一层的其他结点不受限制的子向量数也都是 m_k。也就是说，若不受限制的 x_0 取值有 m_0 个，则该层上有 m_0 个结点；若不受限制的 x_1 取值有 m_1 个，则该层上有 m_0m_1 个结点；依次类推。由于认为在同一层上不受限制的结点数相同，因此，该路径上实际生成的结点数估计为

$$m = 1 + m_0 + m_0m_1 + m_0m_1m_2 + \cdots$$

计算路径上结点数 m 的蒙特卡罗算法描述如下：

```
/* 已知随机路径上取值数据 m0,m1,…,mk−1*/
m=1;t=1;
for(j=0;j<=k−1;j++)
    {t=t*m[j];
     m=m+t;
    }
printf("%ld",m);
```

把所求得的随机路径上的结点数（或若干条随机路径的结点数的平均值）与状态空间树上的总结点数进行比较，由其比值可以初步看出回溯设计的效益。在下面的 n 皇后问题的回溯求解时将具体应用以上蒙特卡罗算法估计回溯设计的效益。

2.4 回溯设计应用

本节从 4 个实际问题的回溯设计求解说明回溯法的应用。

2.4.1 桥本分数式

1. 问题提出

日本数学家桥本吉彦教授于 1993 年 10 月在我国山东举行的中日美三国数学教育研讨会上向与会者提出以下填数趣题：把 1,2,…,9 这 9 个数字填入下式的 9 个方格中（数字不得重复），使下面的分数等式成立

$$\frac{\square}{\square\square}+\frac{\square}{\square\square}=\frac{\square}{\square\square}$$

桥本教授当即给出了一个解答。这一分数式填数趣题究竟共有多少个解答？试求出所有解答（等式左边两个分数交换次序只算一个解答）。

2. 回溯算法设计

这一填数趣题的解是否唯一？如果不唯一究竟有多少个解？ 由人工推算求解难度太大，通过程序设计由计算机来探求更为合适。

我们采用回溯法逐步调整探求。把式中 9 个□规定一个顺序后，先在第一个□中填入一个数字（从 1 开始递增），然后从小到大选择一个不同于前面□的数字填在第二个□中，依次类推，把九个□都填入没有重复的数字后，检验是否满足等式。若等式成立，打印所得的解。然后第九个□中的数字调整增 1 再试，直到调整为 9（不能再增）；返回前一个□中数字调整增 1 再试；依次类推，直至第一个□中的数字调整为 9 时，完成调整探求。

可见，问题的解空间是 9 位的整数组，其约束条件是 9 位数中没有相同数字且必须满足分式的要求。

为此，设置 a 数组，式中每一□位置用一个数组元素来表示：

$$\frac{a(1)}{a(2)a(3)}+\frac{a(4)}{a(5)a(6)}=\frac{a(7)}{a(8)a(9)}$$

同时，记 $m1=a(2)a(3)=a(2)*10+a(3)$

$m2=a(5)a(6)=a(5)*10+a(6)$

$m3=a(8)a(9)=a(8)*10+a(9)$

所求分数等式等价于整数等式 $a(1)*m2*m3+a(4)*m1*m3=a(7)*m1*m2$ 成立。这一转化可以把分数的测试转化为整数测试。

注意到等式左侧两分数交换次序只算一个解，为避免解的重复，设 $a(1)<a(4)$。

式中 9 个□各填一个数字，不允许重复。为判断数字是否重复，设置中间变量 g：先赋值 $g=1$；若出现某两数字相同（即 $a(i)=a(k)$）或 $a(1)>a(4)$，则赋值 $g=0$（重复标记）。

首先从 $a(1)=1$ 开始，逐步给 $a(i)(1\leqslant i\leqslant 9)$赋值，每一个 $a(i)$赋值从 1 开始递增至 9。直至

a(9)赋值，判断：

若 *i*=9,*g*=1,a(1)**m*2**m*3+a(4)**m*1**m*3=a(7)**m*1**m*2 同时满足，则为一组解，用 *n* 统计解的个数后，格式打印输出这组解。

若 *i*<9 且 *g*=1，表明还不到 9 个数字，则下一个 *a*(*i*)从 1 开始赋值继续。

若 *a*(9)=9，则返回前一个数组元素 *a*(8)增 1 赋值（此时，*a*(9)又从 1 开始）再试。若 *a*(8)=9，则返回前一个数组元素 *a*(7)增 1 赋值再试。依次类推，直到 *a*(1)=9 时，已无法返回，意味着已全部试毕，求解结束。

按以上所描述的回溯的参量：*m*=*n*=9

元素初值：a[1]=1，数组元素初值取 1。

取值点：a[i]=1，各元素从 1 开始取值。

回溯点：a[i]=9，各元素取值至 9 后回溯。

约束条件 1：a[i]==a[k] || a[1]＞a[4]，其中（*i*＞*k*）。

约束条件 2：i=9 && a[1]*m2*m3+a[4]*m1*m3=a[7]*m1*m2

3. 桥本分数式回溯 C 程序设计

```
/* 把 1,2,...,9 不重复填入*/
#include <stdio.h>
void main()
{  int g,i,k,n,a[10]; long m1,m2,m3;
   i=1;a[1]=1;n=0;
   while (1)
    {g=1;
     for(k=i-1;k>=1;k--)
        if(a[i]==a[k] || a[1]>a[4]) g=0;                    /* 两数相同或 a[1]>a[4],标记 g=0 */
     if(i==9 && g==1 )
      {m1=a[2]*10+a[3];m2=a[5]*10+a[6];m3=a[8]*10+a[9];
       if(a[1]*m2*m3+a[4]*m1*m3==a[7]*m1*m2)      /* 判断是否满足等式 */
          { n++;printf("(%2d)",n);
            printf("%d/%ld+%d/",a[1],m1,a[4]);
            printf("%ld=%d/%ld   ",m2,a[7],m3);
            if(n%3==0) printf("\n");
             }
      }
    if(i<9 && g==1) {i++;a[i]=1;continue;}                  /* 不到 9 个数,往后继续 */
    while(a[i]==9 && i>1) i--;                              /* 往前回溯 */
    if(a[i]==9 && i==1) break;                              /* 至第 1 个数为 9,结束 */
    else a[i]++;
   }
}
```

4. 程序运行结果与说明

(1) 1/26+5/78=4/39　　(2) 1/32+5/96=7/84　　(3) 1/32+7/96=5/48

(4) 1/78+4/39=6/52　　(5) 1/96+7/48=5/32　　(6) 2/68+9/34=5/17

(7) 2/68+9/51=7/34　　(8) 4/56+7/98=3/21　　(9) 5/26+9/78=4/13

(10) 6/34+8/51=9/27

得 10 个解。

关于桥本分数式求解，已有应用程序设计得到 9 个解的报导，遗失了一个解。可见在程序设计求解时，如果程序中结构欠妥或参量设置不当，都可能造成增解或遗解。不要认为计算机求解万无一失，程序设计时掉以轻心同样会造成失误。只要我们在算法与程序设计时缜密分析，对运行结果作必要的检验，遗解或增解是可以避免的。

5. 问题变通

把 0,1,2,…,9 这 10 个数字填入下式的 10 个方格中（数字不得重复,且要求 0 不得填在各分数的分子与分母的首位），使下面的分数等式成立

$$\frac{\square}{\square\square}+\frac{\square}{\square\square\square}=\frac{\square}{\square\square}$$

这一分数等式填数趣题究竟共有多少个解答？试求出所有解答。

在以上回溯设计基础上改变参量：m=n=10。

元素初值：a[1]=0，数组元素初值取 0。

取值点：a[i]=0，各元素从 0 开始取值。

回溯点：a[i]=9，各元素取值至 9 后回溯。

约束条件 1：a[i]==a[k] || a[1]*a[2]*a[4]*a[5]*a[8]*a[9]==0，其中（i>k）

约束条件 2：i=10 && a[1]*m2*m3+a[4]*m1*m3==a[8]*m1*m2

请注意数组元素的初值、取值点与回溯点，以及问题中的约束条件的设置差异。

2.4.2 排列组合

排列组合是基础数学知识，从 n 个不同元素中任取 m 个（约定 $1<m\leqslant n$），按任意一种次序排成一列，称为排列，其排列种数记为 A(n,m)。从 n 个不同元素中任取 m 个（约定 $1<m<n$）成一组，称为一个组合，其组合种数记为 C(n,m)。计算 A(n,m)与 C(n,m)只要简单进行乘运算即可，要具体展现出排列的每一列与组合的每一组，决非轻而易举。

本节应用回溯设计来具体实现排列与组合。

1. 实现排列 A(n,m)

对指定的正整数 m,n(1<m≤n),具体实现排列 A(n,m)。

（1）回溯算法设计

应用回溯法产生排列 A(n,m)，设置一维数组 a，a(i)(i=1,2,…,m)在 1～n 中取值。首先从 a(1)=1 开始，逐步给 a(i)(1≤i≤m)赋值，每一个 a(i)赋值从 1 开始递增至 n。为判断数字是

否重复，设置中间变量 g：先赋值 g=1；若出现某两数字相同（即 $a(i)=a(j)$），则赋值 g=0（重复标记）。

若 $i=m$ 与 g=1 同时满足，则为一组解，用 s 统计解的个数后，格式打印输出这组解。

若 $i<m$ 且 g=1，表明还不到 m 个数字，则下一个 $a(i)$从 1 开始赋值，继续。

若 $a(i)=n$，则返回前一个数组元素 $a(i-1)$增 1 赋值（此时，$a(i)$又从 1 开始）再试。若 $a(i-1)=n$，则返回前一个数组元素 $a(i-2)$增 1 赋值再试。一般地，若 $a(i)=n(i>1)$，则回溯到前一个数组元素 $a(i-1)$增 1 赋值再试。直到 $a(1)=9$ 时，已无法返回， 意味着已全部试毕，求解结束。

问题的解空间是由数字 1～n 组成的 m 位整数组，其约束条件是没有相同数字。

按以上所描述的回溯的参量：$m,n(m\leqslant n)$

元素初值：a[1]=1，数组元素初值取 1。

取值点：a[i]=1，各数组元素从 1 开始取值。

回溯点：a[i]=n，各数组元素取值至 n 后回溯。

约束条件 1：a[j]==a[i]，其中（i>j），排除相同取值。

约束条件 2：i=m，已取 m 个不同数值时输出一个排列。

（2）回溯实现 A(n,m)的 C 程序实现

```
/* 实现排列 A(n,m) */
#include <stdio.h>
#define N 30
void main()
{int n,m,a[N],i,j,g;
 long s=0;
 printf(" input n   (n<10):"); scanf("%d",&n);
 printf(" input m(1<m<=n):"); scanf("%d",&m);
 i=1;a[i]=1;
 while(1)
   {g=1;
    for(j=1;j<i;j++)
       if(a[j]==a[i]){g=0;break;}                    /* 出现相同元素时返回 */
    if(g && i==m)
       {s++;
        for(j=1;j<=m;j++)
             printf("%d",a[j]);                      /* 输出一个排列 */
        printf("   ");
        if(s%10==0) printf("\n");
       }
    if(g && i<m) {i++;a[i]=1;continue;}
    while(a[i]==n) i--;                               /* 回溯到前一个元素 */
    if(i>0) a[i]++;
    else break;
```

```
    }
  printf("\n s=%ld\n",s);
}
```

（3）程序运行示例

运行程序，输入 n=5，m=3，得

```
123   124   125   132   134   135   142   143   145   152
153   154   213   214   215   231   234   235   241   243
245   251   253   254   312   314   315   321   324   325
341   342   345   351   352   354   412   413   415   421
423   425   431   432   435   451   452   453   512   513
514   521   523   524   531   532   534   541   542   543
s=60
```

特别是当输入 m 与 n 相同时，则输出 n 的全排列。

当排列的元素超过 10 个时，为区别 12 是一个元素 12 还是两个元素 1、2，可在输出排列的每一个元素后加空格。

（4）程序变通

注意到组合与组成元素的顺序无关，约定组合中的组成元素按递增排序。因而，把以上程序中的约束条件 1（a[j]==a[i]）修改为 a[j]>=a[i]，即可实现从 n 个不同元素中取 m 个（约定 $1<m<n$）的组合。

把以上程序中的输出语句 printf("%d",a[j])改为 printf("%c",a[j]+64);排列（或组合）输出由前 n 个正整数改变为前 n 个大写英文字母输出。

把以上程序中的输出语句 printf("%d",a[j])改为 printf("%c",n+65−a[j]);排列（或组合）输出由前 n 个正整数改变为前 n 个大写英文字母逆序输出。

2. 一类复杂排列探索

探讨实现从 n 个不同元素中取 r（约定 $1<r\leqslant n$）个元素与另外 m 个相同元素组成的复杂排列。

（1）回溯探索复杂排列算法设计

这里应用回溯法探索从 n 个不同元素中取 r 个元素与另外 m 个相同元素组成的排列。

设 n 个不同元素为数字 1～n，m 个相同元素为 m 个数字 0。设置一维 a 数组，应用回溯法产生由数字 0～n 这 n+1 个元素取 r+m 个数字组成的 r+m 元组，检验每一个 r+m 元组，若非 0 元素（即数字 1～n）有重复时舍去。设置变量 h 统计 0 的个数，若 $h\neq m$，舍去。

与以上求解 $A(n,m)$不同，元素的取值从 0 开始。判断元素取值是否存在重复时需加上非零限制（即 a[k]!=0 && a[i]==a[k]）。同时需统计 0 的个数 h，判断时应注意 h 是否等于指定的 m。

问题的解空间是由数字 0～n 组成的 r+m 位整数（0 可在首位）组，其约束条件是没有相同的非 0 数字且 0 的个数为 m 个。

按以上所描述的回溯的参量：m,r,n ($r\leqslant n$)

元素初值：a[1]=0，数组元素初值取 0。

取值点：a[i]=0，各数组元素从 0 开始取值。

回溯点：a[i]=n，各数组元素取值至 *n* 时回溯。

约束条件 1：a[k]!=0 && a[i]==a[k]，（其中 *i*>*k*），非零元素相同时排除。

约束条件 2：i=r+m && h=m，（其中 *h* 为 *a*(*i*)中 0 的个数），当已取 *r*+*m* 个值且其中有 *m* 个零时输出一个排列。

（2）复杂排列的 C 程序实现

```
/* 从 n 个不同元素取 r 个与另 m 个相同元素的复杂排列 */
#include <stdio.h>
#include <math.h>
void main()
{  int i,g,h,k,j,m,n,r,a[200];
   long s;
   printf(" input n: "); scanf("%d",&n);
   printf(" input r(1<r<=n): "); scanf("%d",&r);
   printf(" input m: "); scanf("%d",&m);
   i=1;s=0;a[1]=0;
   while (1)
      {g=1;
       for(k=i-1;k>=1;k--)
            if(a[k]!=0 && a[i]==a[k] ) g=0;        /* 非零元素相同则返回  */
       if(i==r+m && g==1)
          {h=0;
           for(j=1;j<=r+m;j++)
               if(a[j]==0) h++;
           if(h==m)                                /* 判别是否有 m 个零 */
              {s++;
                 for(j=1;j<=r+m;j++)
                    printf("%d",a[j]);
                 printf("    ");
                 if(s%10==0) printf("\n");
              }
          }
       if(i<r+m && g==1)
          {i++;a[i]=0;continue;}
       while(a[i]==n && i>1) i--;                  /* 向前回溯 */
       if(a[i]==n && i==1) break;
          else a[i]=a[i]+1;
      }
   printf("\n s=%ld\n",s);                         /* 输出解的个数 */
}
```

（3）程序运行与变通

运行程序，输入 n=4，r=2，m=2，得

```
0012   0013   0014   0015   0021   0023   0024   0025   0031   0032
0034   0035   0041   0042   0043   0045   0051   0052   0053   0054
0102   0103   0104   0105   0120   0130   0140   0150   0201   0203
……
4010   4020   4030   4050   4100   4200   4300   4500   5001   5002
5003   5004   5010   5020   5030   5040   5100   5200   5300   5400
s=120
```

以上为从 1～4 中取 2 个元素与 2 个 0 所得的 120 个排列。

变通以上算法，实现 n 个相同元素与另 m 个相同元素的排列。

3. 允许重复的组合

在 n 个不同元素中取 m 个允许重复元素的组合，其组合数为 $c(n+m-1,m)$，相当于 m 个无区别的球放进 n 个有标志的盒子，每个盒子放的球不加限制的方案数。设计程序，显示这些组合。

（1）回溯算法设计

为实现允许重复的组合，约定 $1 \leqslant a(1) \leqslant a(i) \leqslant a(m) \leqslant n$，即按非递减顺序排列。在以上实现基本组合基础上作两点修改：

当 $i<m$ 时，i 增 1，$a(i)$从 $a(i-1)$开始取值（因为可重复）；直至 $i=m$ 时输出结果。

当 $a(i)=n$ 时回溯（因为组合的每一位置最大都可以取 n），直至 $i=0$ 时结束。

问题的解空间是由数字 1～n 组成的 m 位整数组，其约束条件是整数组中允许有相同数字但整数组中的数字随序号的增加而不减。

按以上描述的回溯的参量：$m,n(m \leqslant n)$

元素初值：a[1]=1，数组元素取初值 1。

取值点：a[i]=a[i−1]，从前一个元素值取值，体现不降。

回溯点：a[i]=n，数组元素取值至 n 时回溯。

约束条件：i=m，已有 m 个值时输出一个组合。

（2）许重复组合的 C 程序实现

```
/* 从 n 个中取 m 个允许重复的组合展现 */
#include <stdio.h>
#define N 30
void main()
{int n,m,a[N],i,j;
 long   c=0;
 printf(" input n :"); scanf("%d",&n);
 printf(" input m(1<m<=n):"); scanf("%d",&m);
 i=1;a[i]=1;
 while(1)
```

```
    {if(i==m)
        {c++;
         for(j=1;j<=m;j++) printf("%d",a[j]);
         printf("    ");
         if(c%10==0) printf("\n");
        }
      else
        {i++; a[i]=a[i-1]; continue;}              /* a(i)从 a(i-1)开始取值 */
      while(a[i]==n) i--;                          /* 调整或回溯 */
      if(i>0) a[i]++;
      else break;
    }
  printf("\nc=%ld\n",c);
}
```

（3）程序运行示例与说明

运行程序示例:

```
input n :5
input m(1<m<=n):3
111   112   113   114   115   122   123   124   125   133
134   135   144   145   155   222   223   224   225   233
234   235   244   245   255   333   334   335   344   345
355   444   445   455   555
c=35
```

注意：当组合的元素为数字且有两位以上时，每一组合中的元素要注意分隔。

2.4.3　德布鲁金环序列

1. 问题提出

由 2^n 个 0 或 1 组成的数环，形成的由相连 n 个数字组成的 2^n 个二进制数恰好在环序列中都出现一次。这个序列被称作 n 阶德布鲁金（Debrujin）环序列。

为构造与统计方便，约定 n 阶德布鲁金环序列由 n 个 0 开头。

二阶德布鲁金环序列非常简单，显然只有 0011 这一个解，2 个数字组成的二进制数依次为 00,01,11,10（因为是环，尾部 0 即开头的 0，下同），共 4 个，每个各出现一次。

三阶德布鲁金环序列也不复杂，由 000 开头，第 4 个数字与第 8 个数字显然都为 1（否则会出现 0000 出界）。余下三个数字组合只能为 011,110,101 三种情形。而 00011011 未出现 111,且有 110,011 等重复，显然不满足三阶德布鲁金环序列条件。因而三阶德布鲁金环序列有 00010111 与 00011101 两个解：

解 00010111 中每 3 个数字组成的二进制数依次为 000，001，010，101，011，111，110，

100，这 8 个数各出现一次。

解 00011101 中每 3 个数字组成的二进制数依次为 000，001，011，111，110，101，010，100，这 8 个数各出现一次。

分析这两个解，事实上是互为顺时针方向与逆时针方向的关系，其中一个解为顺时针方向，则另一个解为逆时针方向。

随着阶数的增加，求解德布鲁金序列难度也相应增大。

下面首先应用穷举法求解 4 阶德布鲁金环序列，然后应用回溯设计求解 *n* 阶德布鲁金环序列。

2. 穷举求解 4 阶德布鲁金环序列

求解由 16 个 0 或 1 组成的环序列，形成的由每相连 4 个数字组成的 16 个二进制数恰好在环中都出现一次。

（1）穷举算法设计

约定序列由 0000 开头，第 5 个数字与第 16 个数字显然都为 1（否则会出现 00000）。余下 10 个数字应用穷举探求。

设置一维 *a* 数组，由约定 0000 开头，即 *a*(0)～*a*(3)均为 0；*a*(4)=1,*a*(15)=1。其余 10 个数字 *a*(5)～*a*(14)通过穷举探求。因为是环序列，*a*(16)～*a*(18)即为开头的 0。

分析 10 个数字 0、1 组成的二进制数，高位最多 2 个 0（否则出现 0001 或 0000 重复），即循环的初值可定为 $n1=2^7$。同时，高位不会超过 4 个 1（否则出现 11111 超界），即循环的终值可定为 $n2=2^9+2^8+2^7+2^6$。

对区间[*n*1,*n*2]中的每一个整数 *n*（为不影响循环，赋值给 *b*），通过除以 2 取余转化为 10 个二进制数码。用变量 *i* 统计该二进制数中 1 的个数，若 *i*≠6 返回，确保 16 个数字中有 8 个 1 时转入下面的检验：计算 *m*1,*m*2 并通过比较，检验 *a*(0)～*a*(18)中每 4 个相连数字组成的二进制数若有相同，显然不能满足题意要求，标记 *t*=1 退出。若所有由 4 个相连数字组成的 16 个二进制数没有相同的，满足德布鲁金环序列条件，作打印输出。

（2）4 阶德布鲁金环序列穷举 C 程序实现

```
#include <stdio.h>
void main()
{  int b,m,m1,m2,n,n1,n2,i,j,k,t,a[20];
   m=0;
   n1=128;
   n2=512+256+128+64;                          /* 确定穷举范围 */
   for(n=n1;n<n2;n++)
     {for(k=0;k<=18;k++) a[k]=0;
      a[4]=1;a[15]=1;
      b=n;
      for(i=0,k=14;k>=5;k--)                   /* 正整数 n(即 b)转化为二进制数*/
        {a[k]=b%2; b=b/2;
         i+=a[k];
```

```
        }
    if(i!=6)   continue;                          /* 确保 8 个 1 转入以下检验 */
    for(t=0,k=0;k<=14;k++)
    for(j=k+1;j<=15;j++)                          /* 计算并检验 16 个二进制数是否相同 */
      {m1=a[k]*8+a[k+1]*4+a[k+2]*2+a[k+3];
       m2=a[j]*8+a[j+1]*4+a[j+2]*2+a[j+3];
       if(m1==m2)
         {t=1;break;}
      }
    if(t==0)                                      /* 若 16 个二进制数没有相同，输出结果*/
     {m=m+1;
        printf("   No(%2d): ",m);
        for(j=0;j<=15;j++)                        /* 依次输出 16 个二进制数 */
            printf("%1d",a[j]);
        if(m%2==0) printf(" \n");
       }
     }
}
```

（3）运行结果分析

运行程序,得

```
No(1): 0000100110101111        No(2): 0000100111101011
No(3): 0000101001101111        No(4): 0000101001111011
No(5): 0000101100111101        No(6): 0000101101001111
No(7): 0000101111001101        No(8): 0000101111010011
No(9): 0000110010111101        No(10): 0000110100101111
No(11): 0000110101111001       No(12): 0000110111100101
No(13): 0000111100101101       No(14): 0000111101001011
No(15): 0000111101011001       No(16): 0000111101100101
```

4 阶德布鲁金环序列共有以上 16 个解。

分析这 16 个解，事实上可分为 8 组，每组两个解互为顺时针与逆时针方向的关系，即其中一个解为顺时针方向，则另一个解为逆时针方向。

4 阶德布鲁金环序列问题可以“编码转动盘”形式提出：一个编码盘分成 16 个相等的扇面，每个扇面上标注 0 或 1，每相邻的 4 个扇面组成的 4 位二进制数，要求共 16 个 4 位二进制输出没有重复。

3. 回溯求解 n 阶德布鲁金环序列

（1）算法设计

在 n 阶德布鲁金环序列中，共有 $m=2^n$ 个二进制数字。设置一维 a 数组，同样按以上约定，$a(n)=1$，$a(m-1)=1$，其余数组元素为 0。

应用回溯法探求 $a(n+1)\sim a(m-2)$，这些元素取 0 或 1。问题的解空间是由数字 0 或 1 组成的 $m-n-2$ 位整数组，其约束条件是 0 的个数为 $m/2-n$ 个，且没有相同的由相连 n 个数字组成的二进制数。

当 $i\leqslant m-2$ 时，$a(i)$从 0 取值；

当 $i>n+1$ 且 $a(i)=1$ 时回溯；

当 $i=n+1$ 且 $a(i)=1$ 时退出。

当 $a(n+1)\sim a(m-2)$已取数字时，设置 h 统计其中 0 的个数，若 $h\neq m/2-n$，则返回；若 $h=m/2-n$，则进一步通过循环计算 $m1$,$m2$，判断是否有相同的由 n 个数字组成的二进制数。若存有相同的由 n 个数字组成的二进制数，标注 $t=1$；若不存在有相同的由 n 个数字组成的二进制数，保持 $t=0$，作打印输出。

按以上所描述的回溯的参量：n，（计算 m=2n）

元素初值：a[n]=1，a[m−1]=1，其余数组元素初值取 0。

取值点：a[i]=0，各数组元素从 0 开始取值。

回溯点：a[i]=1，各数组元素取值至 1 时回溯。

约束条件 1：i=m−2 且 h=m/2−n（其中 h 为 $a(i)$中 0 的个数）

约束条件 2：m1≠m2，m1 与 m2 分别为环序列中所有由相连的 n 个数字组成的前后的二进制数。

（2）n 阶德布鲁金环序列的 C 程序实现

```
#include <stdio.h>
#include <math.h>
void main()
{  int d,i,h,k,j,m,m1,m2,n,s,t,x,a[200];
   printf("请输入（2<n）n: "); scanf("%d",&n);
   m=1;
   for(k=1;k<=n;k++)   m=m*2;
   s=0;
   for(k=0;k<=m+n;k++) a[k]=0;
   a[n]=1;a[m-1]=1;
   i=n+1;
   while(1)
     {if(i==m-2)
        {for(h=0,j=n+1;j<=m-2;j++)
           if(a[j]==0) h++;
         if(h==m/2-n)                          /* 判别是否有 m/2-n 个零 */
           {for(t=0,k=0;k<=m-2;k++)
            for(j=k+1;j<=m-1;j++)
              {d=1;m1=0;m2=0;
                                               /* 检验是否有相同的由 n 相连数字组成的二进制数 */
               for(x=n-1;x>=0;x--)
```

```
                    {m1=m1+a[k+x]*d; m2=m2+a[j+x]*d;d=d*2;}
                  if(m1==m2)
                    {t=1;break;}
               }
             if(t==0)
                {s++;
                 if(n<=4 || (n>4 && s<=3))
                     {printf("NO(%5d): ",s);
                      for(j=0;j<=m-1;j++)
                          printf("%d",a[j]);
                      printf("\n");
                     }
                 }
             }
          }
       if(i<m-1)
           {i++;a[i]=0;continue;}
       while(a[i]==1 && i>n+1) i--;                    /* 向前回溯 */
       if(a[i]==1 && i==n+1) break;
          else a[i]=1;
      }
  }
```

（3）运行结果与变通

运行程序，输入 n=5，得 5 阶德布鲁金环序列的前 3 个解。

```
NO(1): 00000100011001010011101011011111
NO(2): 00000100011001010011101101011111
NO(3): 00000100011001010011111010110111
```

当输入 n=3 或 n=4 时，可得相应的 3 阶或 4 阶环序列。

当 n＞5 时，程序运行的时间迅速增加。解决 5 阶以上的德布鲁金环序列问题，还要从算法上进行优化与改进。

如果约定 n 阶德布鲁金环序列由 n 个 1 开头，算法应如果变通？

2.4.4　高斯皇后问题及其拓展

在前面对高斯 8 皇后问题进行了简单的穷举设计，本节应用回溯设计解决一般的 n 皇后问题，然后拓广到求解 m 个皇后控制 $n\times n$ 棋盘问题。

1．n 皇后问题

（1）问题提出

一般的 n 皇后问题：要求在 $n\times n$ 方格棋盘放置 n 个皇后使它们不相互攻击（即任意 2 个皇后不允许处在同一横排，同一纵列，也不允许处在同一与棋盘边框成 45° 角的斜线上。正整数 n 从键盘输入，$n>2$）。

（2）回溯探求设计

设置数组 $a(n)$，数组元素 $a(i)$表示第 i 行的皇后位于第 $a(i)$列。求 n 皇后问题的一个解，即寻求 a 数组的一组取值，该组取值中每一元素的值互不相同（即没有任 2 个皇后在同一列），且第 i 个元素与第 k 个元素相差不为$|i-k|$，（即任 2 个皇后不在同一 45° 角的斜线上）。

首先 $a(1)$从 1 开始取值。然后从小到大选择一个不同于 $a(1)$且与 $a(1)$ 相差不为 1 的整数赋给 $a(2)$。再从小到大选择一个不同于 $a(1)$，$a(2)$且与 $a(1)$相差不为 2，与 $a(2)$相差不为 1 的整数赋给 $a(3)$。依次类推，至 $a(n)$也作了满足要求的赋值，打印该数组即为找到的一个 n 皇后的解。

为了检验 $a(i)$是否满足上述要求，设置标志变量 g，g 赋初值 1。若不满足上述要求，则 $g=0$。按以下步骤操作:

令　$x=|a(i)-a(k)|$　$(k=1,2,\cdots,i-1)$

判别：若 $x=0$ 或 $x=i-k$，则 $g=0$。

若出现 $g=0$，则表明 $a(i)$不满足要求，$a(i)$调整增 1 后再试，依次类推。

若 $i=n$ 且 $g=1$，则满足要求，用 s 统计解的个数后，格式打印输出这组解。

若 $i<n$ 且 $g=1$，表明还不到 n 个数，则下一个 $a(i)$从 1 开始赋值继续。

若 $a(n)=n$，则返回前一个数组元素 $a(n-1)$增 1 赋值（此时，$a(n)$又从 1 开始） 再试。若 $a(n-1)=n$，则返回前一个数组元素 $a(n-2)$增 1 赋值再试。一般地，若 $a(i)=n(i>1)$，则回溯到前一个数组元素 $a(i-1)$增 1 赋值再试。直到 $a(1)=n$ 时，已无法返回，意味着已完成试探，求解结束。

问题的解空间是由数字 1～n 组成的 n 位整数组，其约束条件是没有相同数字且每两位数字之差不等于其所在位置之差。

按以上所描述的回溯的参量：n

元素初值：a[1]=1，数组元素取初值 1。

取值点：a[i]=1，各元素从 1 开始取值。

回溯点：a[i]=n，各元素取值至 n 时回溯。

约束条件 1：a[i]=a[k] || |a[i]−a[k]|=i−k, (其中 $i>k$)，排除同一列或同对角线上出现 2 个皇后。

约束条件 2：i=n，当取值达 n 个时输出一个解。

（3）n 皇后问题 C 程序设计

```
/* n 皇后问题 */
#include <stdio.h>
void main()
{  int i,g,k,j,n,s,x,a[20];
   printf("请输入整数 n:");   scanf("%d",&n);
   printf("%d 皇后问题的解:\n");
   i=1;s=0;a[1]=1;
```

```
    while (1)
      {g=1;
       for(k=i-1;k>=1;k--)
          {x=a[i]-a[k];if(x<0) x=-x;
           if(x==0 || x==i-k) g=0;}                    /* 相同或同处一对角线上时返回 */
           if(i==n && g==1)                            /* 满足条件时输出解 */
             {for(j=1;j<=n;j++)
            printf("%d",a[j]);
      printf("     ");s++;
      if(s%5==0) printf("\n");}
           if(i<n && g==1)
              {i++;a[i]=1;continue;}
       while(a[i]==n && i>1) i--;                      /* 往前回溯 */
       if(a[i]==n && i==1) break;
          else a[i]=a[i]+1;}
    printf("\n 共%d 个解.\n",s);
}
```

（4）程序运行示例与说明

运行程序，输入整数 5

5 皇后问题的解：

13524　14253　24135　25314　31425

35241　41352　42531　52413　53142

共 10 个解

运行程序若输入 n=8，即输出高斯 8 皇后问题的所有 92 个解。

（5）时间分析

运用蒙特卡罗方法估计 8 皇后问题回溯设计实际生成的结点数，随机生成 4 条路径分别为：(1,3,0,2,4)、(3,5,2,4,6,0)、(0,7,5,2,6,1,3)与(2,5,1,6,0,3,7,4)。这 4 次试验不受限的结点数从图 2.2 可以得到。例如，对于路径(1,3,0,2,4)的状态空间树上的第 1 层的结点数（即 x_0 的取值）为 8；因为 x_0=1，所以 x_1 的可取不相冲突的列号只有 5 个；因为 x_1=3，所以 x_2 可取不相冲突的列号只有 4 个；……所以，这一路径的状态空间树上实际生成的结点数目为

$$m = 1 + 8 + 8\times 5 + 8\times 5\times 4 + 8\times 5\times 4\times 3 + 8\times 5\times 4\times 3\times 2 = 1649$$

显然这 4 次试验中实际生成的结点数的平均值为 1 633 个。而 8 皇后问题状态空间树的结点总数为

$$s = 1 + \sum_{j=0}^{7}\prod_{i=0}^{j}(8-i) = 109\,601$$

因此，实际生成的结点数大约占总结点数 s 的 1.49%。

同时，可应用 C 语言的测试函数 clock()或 time()对穷举与回溯设计的运行时间进行相对比较。对例 2.6 穷举求 8 皇后问题的程序进行测试，程序运行时间约为 11 秒。在十几秒至几十秒时间内计算并输出 92 个解，感觉还不算慢。应用时间测试函数对以上回溯设计求 8 皇后

问题进行测试，算法只需约 30 毫秒，只有穷举求解的数百分之一。比较之下，可见回溯求解的效率明显高于穷举设计求解。

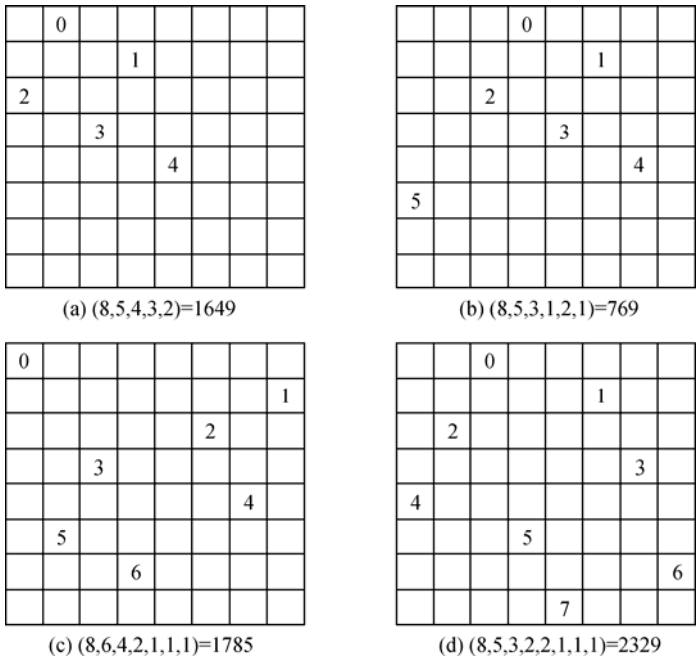


图 2.2　估计 8 皇后问题状态空间树实际大小

2. *m* 个皇后控制 *n*×*n* 棋盘问题

（1）问题提出

在 *n*×*n* 的国际象棋棋盘上，如何放置 *m* 个皇后($m \leqslant n$)，可以控制棋盘的每一个方格而皇后互相之间不能攻击呢？（即任意两个皇后不允许处在同一横排，同一纵列，也不允许处在同一与棋盘边框成 45°角的斜线上。）

（2）回溯算法设计

皇后控制棋盘问题比以上的 *n* 皇后问题求解难度更大些。采用回溯法探求。

设置数组 *a*(*n*)，数组元素 *a*(*i*)表示第 *i* 行的皇后位于第 *a*(*i*)列，当 *a*(*i*)=0 时表第 *i* 行没有皇后。

求 *m* 个皇后控制 *n*×*n* 棋盘的一个解，即寻求 *a* 数组的一组取值，该组取值中 *n*−*m* 个元素值为 0，*m* 个元素的值大于零且互不相同（即没有任两个皇后在同一列），第 *i* 个元素与第 *k* 个元素相差不为|*i*−*k*|，（即任两个皇后不在同一 45°角的斜线上），且这 *m* 个元素可控制整个棋盘的每一格。

程序的回溯进程同 *n* 皇后问题设计，所不同的是所有元素从 0 开始取值，且 *n* 个元素中要确保 *n*−*m* 个元素取 0。因而在判断元素取值是否存在重复时需加上非零限制（即 a[k]!=0 && a[i]==a[k]），同时设置变量 *h* 统计 0 的个数，判断时应注意 *h* 是否等于指定的 n−m。

为了检验是否控制整个棋盘，设置二维数组 *b*(*n*,*n*)表示棋盘的每一格，数组的每一个元素置 0。对一个皇后放置在 *a*(*f*)，其控制的范围为：

控制同行：if(c==f) b[c][j]=1; (1≤c≤n)

控制同列：if(j==a[f]) b[c][j]=1; (1≤j≤n)

控制斜线：if(fabs(c−f)==fabs(j−a[f])) b[c][j]=1;　(1≤c≤n，1≤j≤n)

对皇后控制的范围内的所有 b 数组元素赋 1。所有 m 个皇后控制完成后，检验 b 数组是否全为 1：只要有一个不为 1，即不是全控；若 b 数组所有元素都为 1，棋盘全控，打印输出解，用变量 s 统计解的个数。

问题的解空间是由数字 0～n 组成的 n 位整数（0 可在首位）组，其约束条件是没有相同的非 0 数字且 0 的个数为 $n−m$ 个，同时须满足全控要求。

按以上所描述的回溯的参量：n,m (m≤n)

元素初值：a[1]=0，数组元素取初值 0。

取值点：a[i]=0，各元素从 0 开始取值。

回溯点：a[i]=n，各元素取值至 n 时回溯。

约束条件 1：a[k]!=0 && a[i]=a[k] || a[i]*a[k]＞0 && fabs(a[i]−a[k])=i−k，（其中 k＜i），排除同一列或同对角线上出现 2 个皇后。

约束条件 2：i=n　&& h=n−m && b[1−n][1−n]=1，当取值达 n 个，其中 $n−m$ 个零，且棋盘全控时输出一个解。

（3）m 个皇后控制 n×n 棋盘 C 程序设计

```
/* m 皇后控制 n×n 棋盘求解问题 */
#include <math.h>
#include <stdio.h>
void main()
{  int i,g,h,k,c,d,e,f,j,m,n,t,s,x,a[20],b[20][20];
   printf(" input n(n<10):"); scanf("%d",&n);
   printf(" input m(m<=n):"); scanf("%d",&m);
   i=1;s=0;a[1]=0;
   while (1)
     {g=1;
      for(k=i−1;k>=1;k−−)
         {x=a[i]−a[k];
          if(a[k]!=0 && x==0 || a[i]*a[k]>0 && fabs(x)==i−k) g=0;
         }                                          /* 非零元素不能相同，不同对角线  */
      if(i==n && g==1)
         {h=0;
          for(j=1;j<=n;j++)
              if(a[j]==0) h++;
          if(h==n−m)                                /* 判别是否有 n−m 个零 */
             {for(c=1;c<=n;c++)
              for(j=1;j<=n;j++)
                  b[c][j]=0;
```

```
            for(f=1;f<=n;f++)
              {if(a[f]!=0)
                 {for(c=1;c<=n;c++)
                  for(j=1;j<=n;j++)
                    {if(c==f) b[c][j]=1;        /* 控制同行 */
                     if(j==a[f]) b[c][j]=1;     /* 控制同列 */
                     if(fabs(c-f)==fabs(j-a[f])) b[c][j]=1;
                    }                           /* 控制四方向对角线 */
                 }
              }
            t=0;
            for(c=1;c<=n;c++)
            for(j=1;j<=n;j++)
                if(b[c][j]==0) t=1;             /* 只要棋盘中有一格不能控制，则 t=1 */
            if(t==0)                            /* 棋盘所有格都能控制，输出结果 */
              {s++;
               for(j=1;j<=n;j++)
                   printf("%d",a[j]);
               printf("    ");
               if(s%5==0) printf("\n");
              }
          }
      }
      if(i<n && g==1)
          {i++;a[i]=0;continue;}
      while(a[i]==n && i>1) i--;                 /* 向前回溯 */
      if(a[i]==n && i==1) break;
        else a[i]=a[i]+1;
    }
  printf("\n s=%d\n",s);                         /* 输出解的个数 */
}
```

运行程序，输入 *n*=5，*m*=3，得 16 个解。

```
02401   03025   03041   03105   03501
04205   10403   10420   10530   14030
30205   30401   50130   50203   50240
52030
s=16
```

（4）解的讨论

在以上程序设计求解输出数字解的基础上，应用 C 语言中的画矩形与画线段函数可对每

一个解画出示意图。

例如，输入 *n*=8，*m*=5，得 728 个解，其中解 08420370 的图形如图 2.3 所示。

若输入 *n*=8，*m*=8，得 8 皇后问题的 92 个解，其中解 28613574 的图形如图 2.4 所示。

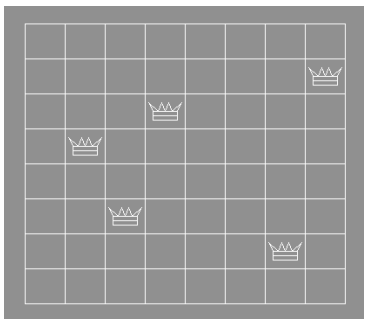


图 2.3　5 皇后控制 8×8 棋盘的一个解图示

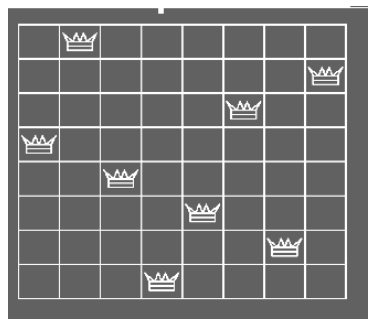


图 2.4　高斯 8 皇后问题的一个解图示

综合 *m* 个皇后控制 *n*×*n* 棋盘（3≤*m*≤*n*≤9）的解数如表 2.2 所示。

表 2.2　　*m* 皇后控制 *n*×*n* 棋盘（3≤*m*≤*n*≤9）的解数

皇后数 *m*	4×4	5×5	6×6	7×7	8×8	9×9
3	16	16	0	0		
4	2	32	120	8	0	0
5		10	224	1262	728	92
6			4	552	6912	7744
7				40	2456	38732
8					92	10680
9						352

从表 2.2 各列上端的非零项可知，全控 8×8 或 9×9 棋盘至少要 5 个皇后，全控 6×6 或 7×7 棋盘至少要 4 个皇后。

同时，由表 8×8 列的下端可知，用 8 皇后控制 8×8 棋盘（显然是全控），实际上即高斯 8 皇后问题，共有 92 个解。从表中其它各列的下端知 6 皇后问题有 4 个解，而 7 皇后问题有 40 个解，等等。当输入 *m*=*n* 时，即输出 *n* 皇后问题的解。这就是说，以上设计求解的 *m* 个皇后控制 *n*×*n* 棋盘问题引伸与推广了 *n* 皇后问题。

最后指出，若 *n*≥10，为避免解中的二位数与一位数的混淆，输出解时须在两个 *a* 数组元素之间加空格。

2.5　回溯设计的优化

回溯算法在搜索执行的同时产生解空间，是一种系统地搜索问题解答的方法。回溯算法避免了生成那些不可能产生解的状态，不断去除那些实际上不可能产生所需解的结点，以减少问题的计算量。

在搜索过程中，为了摆脱当前失败状态，返回搜索步骤中的上一点，去寻求新的路径，以求得答案。

回溯算法的优化问题要从它的约束条件着手，可以利用对称性裁减子树、划分成子问题等。

估计回溯算法的平均效率即搜索树中平均遍历的结点，影响回溯算法效率的因素有：（a）搜索树的结构；（b）看其分支情况：分支均匀否；（c）树的深度；（d）对称程度，对称是否适合裁减；（e）解的分布，在不同子树中分布多少，是否均匀；（f）分布深度；（g）约束条件的判断，计算是否简单。

回溯算法的改进途径如下：

（1）根据树分支设计优先策略，结点少的分支优先，解多的分支优先；

（2）根据求解问题调整与改进搜索数组元素的取值点与回溯点；

（3）根据求解问题调整与改进搜索的约束条件。

下面举两个实例说明回溯设计的优化。

【例 2.8】 实现组合 $C(n,m)$设计优化。

对指定的正整数 m,n（约定 $1<m\leqslant n$），具体展现 $A(n,m)$的每一个组合。作为一般的设计求解，只要在展现排列设计的基础上，把其中的排除相同取值约束条件 a[j]==a[i]，其中（$i>j$）修改为：a[j]>=a[i]即可。

下面给出实现组合 $C(n,m)$的优化设计。

（1）算法改进要点

注意到组合与元素的顺序无关，约定组合中的元素按升序排序。

回溯实现从 1～n 这 n 个数中每次取 m 个数的组合，设置 a 数组，数组下标 i 从 1 开始取值。约定 $a(1),\cdots,a(i),\cdots,a(m)$按递增顺序排列，$a(i)$后有 $m-i$ 个大于 $a(i)$的元素，其中最大取值为 n，显然 $a(i)$最多取 $n-m+i$，即 $a(i)$回溯的条件是 $a(i)=n-m+i$。

当 $i<m$ 时，i 增 1，$a(i)$从 $a(i-1)+1$ 开始取值；直至 $i=m$ 时输出结果。

当 $a(i)=n-m+i$ 时回溯，直至 $i=0$ 时结束。

按以上所描述的回溯的参量：$m,n(m\leqslant n)$

元素初值：a[1]=1，数组元素取初值 1。

取值点：a[i]=a[i−1]+1，从前一个元素值加 1 取值，体现升序，排除相同。

回溯点：a[i]=n−m+i，数组元素取值至 $n-m+i$ 时回溯。

约束条件：i=m，已有 m 个值时输出一个组合。

（2）从 n 个中取 m 个组合回溯优化设计

```
/* 从 n 个中取 m 个的组合展现  */
#include <stdio.h>
#define N 30
void main()
{int n,m,a[N],i,j; long c=0;
 printf(" input n :"); scanf("%d",&n);
 printf(" input m(1<m<=n):"); scanf("%d",&m);
 i=1;a[i]=1;
 while(1)
```

```
    {if(i==m)
        {c++;
         for(j=1;j<=m;j++) printf("%d",a[j]);
         printf("    ");
         if(c%10==0) printf("\n");
        }
      else {i++; a[i]=a[i−1]+1; continue;}              /* 元素按递增取值 */
      while(a[i]==n−m+i) i−−;                          /* 调整或回溯或终止 */
      if(i>0) a[i]++;
      else break;
  }
  printf("\nc=%ld\n",c);
}
```

（3）程序运行示例与说明

运行程序，输入 n=7，m=3，得在数字 1～7 中取 3 个的组合为

123　124　125　126　127　134　135　136　137　145

146　147　156　157　167　234　235　236　237　245

246　247　256　257　267　345　346　347　356　357

367　456　457　467　567

c=35

在以上的设计改进中，元素的取值 a[i]=a[i−1]+1，即从取值上决定元素递增，省略了数组元素之间的大小判别，同时为回溯点确定为 a[i]==n−m+i 打下基础。

【例 2.9】 探索复杂排列算法的改进。

应用回溯法探索从 n 个不同元素中取 m（约定 $1<m\leqslant n$）个元素与另外 $n-m$ 个相同元素组成的复杂排列。事实上，这一回溯设计可在“一类复杂排列探索”基础上改进。

（1）算法设计改进要点

引入变量 k 来控制 0 的个数，当 $k<n-m$ 时，a[i]=0，元素需从 0 开始取值；否则，0 的个数已达 $n-m$ 个，a[i]=1，即从 1 开始取值。这样处理，使 0 的个数不超过 $n-m$，减少一些无效操作，提高了回溯效率。

按以上所描述的回溯的参量：n,m($m\leqslant n$)

元素初值：a[1]=0，数组元素取初值 0。

取值点：当 k＜n−m 时，a[i]=0，需从 0 开始取值；否则，a[i]=1，即从 1 开始取值。

回溯点：a[i]=n，各元素取值至 n 时回溯。

约束条件 1：a[k]!=0 && a[i]=a[k] || a[i]*a[k]＞0 && fabs(a[i]−a[k])=i−k, (其中 $i>k$)，排除同一列或同对角线上出现 2 个皇后。

约束条件 2：i=n　&& h=n−m && b[1−n][1−n]=1, 当取值达 n 个，其中 $n-m$ 个零，且棋盘全控时输出一个解。

（2）复杂排列回溯优化设计

```
#include <stdio.h>
```

```
#define N 30
void main()
{int n,m,a[N],i,j,k,h,t;
 long s=0;
 printf(" input n    (n<10):"); scanf("%d",&n);
 printf(" input m(1<m<=n):"); scanf("%d",&m);
 i=1;a[i]=0; k=1;
 while(1)
    {t=1;
     for(j=1;j<i;j++)
        if(a[j] && a[j]==a[i]) {t=0;break;}          /* 非零元素相同，则返回 */
     if(t && k==n-m && i==n)                         /* 已取 n 个值且 0 的个数为 n-m 时输出解 */
        {s++;
         for(j=1;j<=n;j++) printf("%d",a[j]);
         printf("    ");
         if(s%10==0) printf("\n");
        }
     if(t && (k<n-m || i<n))
       {i++;
        if(k<n-m){a[i]=0; k++;}                     /* 0 的个数增加 1 */
        else a[i]=1;                                /* 若 0 的个数已达到 n-m，则不再取 0 了 */
        continue;
  }
     while(a[i]==n) i--;                            /* 调整或回溯或终止 */
     if(i>0)
        {if(a[i]==0) k--;                           /* 改变取值为 0 的元素值前要先把 0 的个数 k 减少 1 */
         a[i]++;
        }
     else break;
  }
 printf("\n s=%ld\n",s);
}
```

原设计当 i=n 时统计零的个数，优化后应用 k 控制零的个数不超过 n−m，从而减少了回溯实施过程中的无效操作，提高了计算效率。

习　　题

1. 求指定区间上的完全数。

正整数 n 的所有小于 n 的正因数之和若等于 n 本身，称数 n 为完全数。例如，6 的正因数为 1,2,3，而 6=1+2+3，则 6 是一个完全数。试求指定区间[x,y]内的完全数。

2. 一个世纪的 100 个年号全为合数（即不存在一个素数），该世纪称为合数世纪。设计程序探索最早的合数世纪。

3. 圆圈循环报数问题称为 Joseph 问题：有 n 个小朋友按编号顺序 1,2,…,n 逆时针方向围成一圈。从 1 号开始按逆时针方向 1,2,…,m 报数，凡报数 m 者出列（显然，第一个出圈的为编号 m 者）。

问：最后剩下一个未出圈者的编号是多少？指定的第 p 个出圈者的编号为多少?

4. 求所有 m 位巧妙平方数：在 1,2,…,9 这九个数字中选 m 个，组成没有重复数字的平方数（整数 m 从键盘输入，$1 \leqslant m \leqslant 9$）。

5. 求 n 个 0 与 m 个 1 组成的排列（n，m 从键盘输入）。

6. 求序列中的最长不降子序列。

给定一个由 n 个正整数组成的序列，从中删除若干个数，使剩下的序列非降。探求并输出所有最长非降子序列。

7. 对已知的 $2n$（n 从键盘输入）个正整数，确定这些数能否分成 2 个组，每组 n 个数，且每组数据的和相等。

第 3 章　递归与分治

递归是算法设计中的一种重要的方法。递归方法即通过函数或过程调用自身将问题转化为本质相同但规模较小的子问题。递归方法具有易于描述和理解、证明简单等优点，在动态规划、贪心算法、回溯法等诸多算法中都有着极为广泛的应用，是许多复杂算法的基础。递归方法中所使用的“分而治之”的策略称为分治策略。本章将重点探讨分治算法设计的基本思想、方法和特点以及应用，最后讨论分治算法求解过程中的递归改写成非递归算法的一般方法。

3.1　递归及其应用

3.1.1　递归与递归调用

一个函数在它的函数体内调用它自身称为递归（recursion）调用。是一个过程或函数在其定义或说明中直接或间接调用自身的一种方法，通常把一个大型复杂的问题层层转化为一个与原问题相似的规模较小的问题来求解。递归策略只需少量的程序就可描述出解题过程所需要的多次重复计算，大大地减少了程序的代码量。递归的能力在于用有限的语句来定义对象的无限集合。用递归思想写出的程序往往十分简洁易懂。一般来说，递归需要有边界条件、递归前进段和递归返回段。当边界条件不满足时，递归前进；当边界条件满足时，递归返回。

使用递归要注意以下几点。

（1）递归就是在过程或函数里调用自身；

（2）在使用递增归策略时，必须有一个明确的递归结束条件，称为递归出口递归和分治是相统一的，递归算法中含有分治思想，分治算法中也常用递归算法。

例如有函数 *r* 如下：

```
int r(int a)
{
   b=r(a−1);
   return b;
}
```

这个函数是一个递归函数，但是运行该函数将无休止地调用其自身，这显然是不正确的。为了防止递归调用无终止地进行，必须在函数内有终止递归调用的手段。常用的办法是加条件判断，满足某种条件后就不再作递归调用，然后逐层返回。

构造递归方法的关键在于建立递归关系。这里的递归关系可以是递归描述的，也可以是递推描述的。

3.1.2　递归应用

下面举例说明递归设计的简单应用。

【例 3.1】 用递归法计算 $n!$。

$n!$的计算是一个典型的递归问题。使用递归方法来描述程序，十分简单且易于理解。

（1）描述递归关系

递归关系是这样的一种关系。设$\{U_1,U_2,U_3,\cdots,U_n,\cdots\}$是一个序列，如果从某一项 k 开始，U_n 和它之前的若干项之间存在一种只与 n 有关的关系，这便称为递归关系。

注意到，当 $n\geqslant 1$ 时，$n!=n*(n-1)!$（$n=0$ 时，$0!=1$），这就是一种递归关系。对于特定的 $k!$，它只与 k 与$(k-1)!$有关。

（2）确定递归边界

在（1）的递归关系中，对大于 k 的 U_n 的求解将最终归结为对 U_k 的求解。这里的 U_k 称为递归边界（或递归出口）。在本例中，递归边界为 $k=0$，即 $0!=1$。对于任意给定的 $N!$，程序将最终求解到 $0!$。

确定递归边界十分重要，如果没有确定递归边界，将导致程序无限递归而引起死循环。例如以下程序：

```
#include <stdio.h>
int f(int x){
    return(f(x-1));
}
main(){
printf(f(5));
}
```

它没有规定递归边界，运行时将无限循环，会导致错误。

（3）写出递归函数并译为代码

将（1）和（2）中的递归关系与边界统一起来用数学语言来表示，即

$$n!=n*(n-1)! \qquad \text{当 } n\geqslant 1 \text{ 时}$$

$$n!=1 \qquad \text{当 } n=0 \text{ 时}$$

再将这种关系翻译为代码，即一个函数：

```
long ff(int n){
long f;
if(n<0)    printf("n<0,input error");
else if(n==0||n==1) f=1;
```

```
    else f=ff(n−1)*n;
return(f);
}
```

（4）完善程序

主要的递归函数已经完成，将程序依题意补充完整即可。

```
#include <stdio.h>
long ff(int n){
long f;
 if(n<0)   printf("n<0,input error");
else if(n==0||n==1) f=1;
      else f=ff(n−1)*n;
 return(f);
 }
void main()
{ int n;
 long y;
 printf("\n input a integer number:\n");
scanf("%d",&n);
 y=ff(n);
 printf("%d!=%ld",n,y);
 }
```

程序中给出的函数 *ff* 是一个递归函数。主函数调用 *ff* 后即进入函数 *ff* 执行，如果 *n*<0,*n*==0 或 *n*=1 时都将结束函数的执行，否则就递归调用 *ff* 函数自身。由于每次递归调用的实参为 *n*−1，即把 *n*−1 的值赋予形参 *n*，最后当 *n*−1 的值为 1 时再作递归调用，形参 *n* 的值也为 1，将使递归终止，然后可逐层退回。

下面我们再举例说明该过程。设执行本程序时输入为 5，即求 5!。在主函数中的调用语句即为 *y*=*ff*(5)，进入 *ff* 函数后，由于 *n*=5，不等于 0 或 1，故应执行 *f*=*ff*(*n*−1)***n*，即 *f*=*ff*(5−1)***5。该语句对 *ff* 作递归调用即 *ff*(4)。递归分为递推和回归，展开结果如图 3.1 所示。

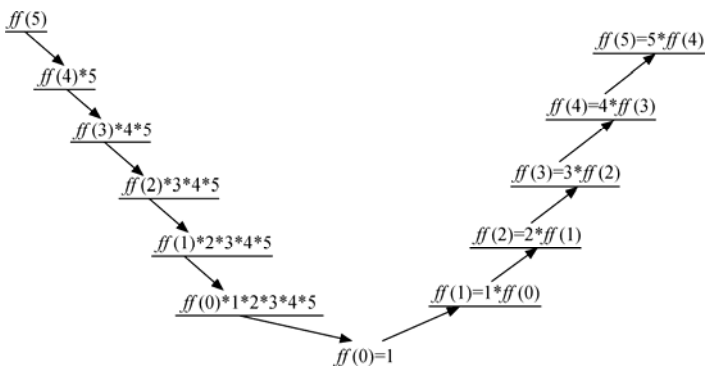


图 3.1　递归展开图

进行 4 次递归调用后，*ff* 函数形参取得的值变为 1，故不再继续递归调用而开始逐层返回主调函数。*ff*(1)的函数返回值为 1，*ff*(2)的返回值为 1*2=2，*ff*(3)的返回值为 2*3=6，*ff*(4) 的返回值为 6*4=24，最后返回值 *ff*(5)为 24*5=120。

综上，得出构造一个递归方法基本步骤，即描述递归关系、确定递归边界、写出递归函数并译为代码，最后将程序完善。

以上例 3.1 也可以不用递归的方法来完成。如可以用递推法，即从 1 开始乘以 2，再乘以 3……直到 *n*。递推法比递归法更容易理解和实现。但是有些问题则只能用递归算法才能实现。典型的问题是 Hanoi 塔问题。

【例 3.2】 一块板上有三根针，A、B、C。A 针上套有 *n* 个大小不等的圆盘，大的在下，小的在上。要把这 *n* 个圆盘从 A 针移动 C 针上，每次只能移动一个圆盘，移动可以借助 B 针进行。但在任何时候，任何针上的圆盘都必须保持大盘在下，小盘在上。从键盘输入 *n*，要求给出移动的次数和方案。

解：由圆盘的个数建立递归关系。当 *n*=1 时，只要将唯一的圆盘从 A 移到 C 即可。当 *n*>1 时，只要把较小的(*n*−1)片按移动规则从 A 移到 B，再将剩下的最大的从 A 移到 C（即中间“借助”B 把圆盘从 A 移到 C），再将 B 上的(*n*−1)个圆盘按照规则从 B 移到 C（中间“借助”A）。

本题的特点在于不容易用数学语言写出具体的递归函数，但递归关系明显，仍可用递归方法求解。代码如下：

```
#include <stdio.h>
hanoi(int n,int x,int y,int z)
{
  if(n==1)
    printf("%c—>%c\n",x,z);
  else
  {
    hanoi(n−1,x,z,y);
    printf("%c—>%c\n",x,z);
    hanoi(n−1,y,x,z);
  }
}
void main()
{
 int h;
 printf("\ninput number:\n");
 scanf("%d",&h);
 printf("the step to moving %2d diskes:\n",h);
 hanoi(h,'a','b','c');
}
```

从程序中可以看出，hanoi 函数是一个递归函数，它有四个形参 *n*、*x*、*y*、*z*。*n* 表示圆盘

数，*x*、*y*、*z* 分别表示三根针。hanoi 函数的功能是把 *x* 上的 *n* 个圆盘移动到 *z* 上。当 *n*==1 时，直接把 *x* 上的圆盘移至 *z* 上，输出 *x*−−>*z*。如 *n*−1 则分为三步：递归调用 move 函数，把 *n*−1 个圆盘从 *x* 移到 *y*；输出 *x*−−>*z*；递归调用 move 函数，把 *n*−1 个圆盘从 *y* 移到 *z*。在递归调用过程中 *n*=*n*−1，故 *n* 的值逐次递减，最后 *n*=1 时，终止递归，逐层返回。当 *n*=4 时程序运行的结果为：

```
input number: 4
the step to moving 4 diskes:
a−−>b    a−−>c    b−−>c    a−−>b    c−−>a    c−−>b
a−−>b    a−−>c    b−−>c    b−−>a    c−−>a    b−−>c
a−−>b    a−−>c    b−−>c
```

上述的两个示例中都应用了函数的递归调用，并且使问题变得简单，算法的复杂度也不高，但并不是所有的问题都用递归可以简化问题，如下例。

【例 3.3】 将正整数 *s* 表示成一系列正整数之和，*n*=*n*1+*n*2+…+*nk*，其中 *n*1>*n*2>…>*nk*, *k*>=1。正整数 *s* 的不同划分个数称为 *s* 的划分数，记为 *p*(*s*)。例如 6 有 11 种不同的划分，所以 p(6)=11，分别是：

6；5+1；4+2；4+1+1；3+3；3+2+1;3+1+1+1；

2+2+2；2+2+1+1；2+1+1+1+1；1+1+1+1+1+1。

应用递归设计求整数 *s* 的拆分数。

（1）递归算法设计

设 *n* 的“最大零数不超过 *m*”的拆分式个数为 *q*(*n*,*m*)，则

$$q(n,m)=1+q(n,n-1) \qquad (n=m)$$

等式右边的“1”表示 *n* 只包含等于 *n* 本身；*q*(*n*,*n*−1)表示 *n* 的所有其他拆分，即最大零数不超过 *n*−1 的拆分。

$$q(n,m)=q(n,m-1)+q(n-m,m) \qquad (1<m<n)$$

其中 *q*(*n*,*m*−1)表示零数中不包含 *m* 的拆分式数目；*q*(*n*−*m*,*m*)表示零数中包含 *m* 的拆分数目，因为如果确定了一个拆分的零数中包含 *m*，则剩下的部分就是对 *n*−*m* 进行不超过 m 的拆分。

加入递归的停止条件。第一个停止条件：*q*(*n*,1)=1，表示当最大的零数是 1 时，该整数 *n* 只有一种拆分，即 *n* 个 1 相加。第二个停止条件：*q*(1,*m*)=1，表示整数 *n*=1 只有一个拆分，不管上限 *m* 是多大。

（2）递归程序实现

```
/* 整数拆分递归计数 */
#include<stdio.h>
long q(int n,int m)                          /* 定义递归函数 q(n,m) */
  {if(n<1 || m<1)   return 0;
  if(n==1 || m==1) return 1;
  if(n<m) return q(n,n);
  if(n==m) return q(n,m−1)+1;
  return q(n,m−1)+q(n−m,m);
  }
```

```
void main()
  {int z,s;                                   /* 调用递归函数 q(s,s) */
  printf("请输入 s:"); scanf("%d",&s);
  printf(" p(%d)=%ld \n",s,q(s,s));
}
```

（3）程序运行示例与说明

运行程序，输入 20，得

p(20)=627

以上程序计算 s 的划分数，分别多次调用 $q(1,1),\cdots,q(s-1,s-1)$，这样的程序会造成子问题重复计算，所以复杂度较高。要将递归算法改写为效率较高的递推，将在下一章介绍。

3.2　分治法概述

3.2.1　分治法基本思想

分治法（Divide and Conquer）是一种用得较多的有效方法，它的基本思想是将问题分解成若干子问题，然后求解子问题。子问题较原问题要容易些，先得出子问题的解，由此得出原问题的解，就是所谓“分而治之”的思想，在上面的递归方法介绍中所使用的“分而治之”的策略称为分治策略。

“分而治之”的思想经常应用于现实生活中，对于求解一个复杂的问题或一个较大的问题，经过系统地分析，将其划分成一些简单问题或较小问题进行解决。当这些子问题解决后，把他们的解联结起来，得到原问题的解。

在算法设计中，首先对求解问题进行系统的分析，之后将其分解成若干性质相同的子问题，所得结果称为求解子集，再对这些求解子集分别处理。如果某些子集还需分而治之，再递归的使用上述方法，直到求解子集不需要再细分为止。最后归并子集的解即得原问题的解。

宏观的看，分治法可以划分成问题的分解和子集解的合并两个过程，如图 3.2 所示。

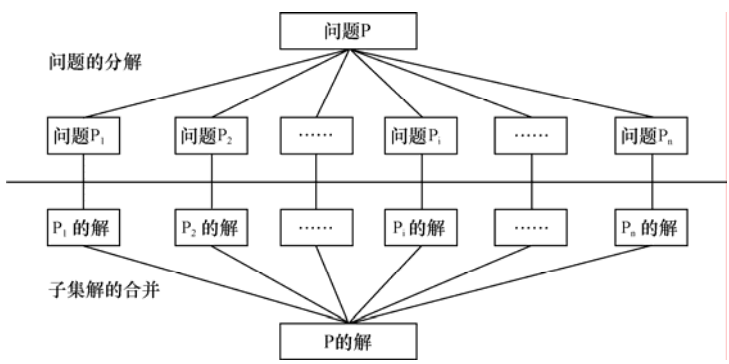


图 3.2　分治算法设计过程图

因而对分治法算法设计过程可以如下描述：

设原问题输入为 $a[n]$，简记为$(1,n)$；

子问题输入为 $a[p]\rightarrow a[q]$，$1\leqslant p\leqslant q\leqslant$n，简记为$(p,q)$。

已知：

```
SOLUTION;
int divide (int, int);
int small (int, int);
SOLUTION conquer (int, int);
SOLUTION combine (SOLUTION, SOLUTION);
```

分治法的抽象控制算法为：

```
SOLUTION DandC(p,q)   /* divide and conquer */
{
    if(small(p,q))
        return conquer(p,q);
    else
        {
         m=divide(p,q);
        return combine( DandC(p,m), DandC(m+1,q) );
        }
}
```

3.2.2 分治算法设计方法和特点

分治法求解思想可以从下面的一个实例的求解中体现：

【例 3.4】 在含有 n 个不同元素的集合 $a[n]$中同时找出它的最大和最小元素。不妨设 $n=2^m, m\geqslant 0$。

对求 n 个数的最大和最小，可以设计出许多种算法，这里使用分治法设计求解。

1. 直接搜索

```
StraitSearch(n, &max, &min)
{   *max=*min=a[0];
    for(i=1; i<n; i++)     /* 语句 1 */
    {    if(a[i]>*max) *max=a[i];
         if(a[i]<*min) *min=a[i];
    }
}
```

其中比较次数为 $2(n-1)$；

如果我们将 StraitSearch( )函数种语句 1 改写成

```
if(a[i]>*max) *max=a[i];
```

```
    else if(a[i]<*min) *min=a[i];
```

则有：

最好情况比较次数为 $n-1$，最坏情况比较次数为 $2(n-1)$，平均情况比较次数为 $3/2(n-1)$。

2. 分治法

集合只有一个元素时：

```
    *max=*min=a[i];
```

集合只有两个元素时：

```
    if(a[i]<a[j]) { *max=a[j]; *min=a[i];}
    else { *max=a[i]; *min=a[j];}
```

集合中有更多元素时，将原集合分解成两个子集，分别求两个子集的最大和最小元素，再合并结果。

算法如下：

```
    typedef struct {
        ElemType max;
        ElemType min;
    } SOLUTION;
    SOLUTION MaxMin(i, j)
    {
    SOLUTION s, s1, s2;
      if(i==j) { s.max=s.min=a[i]; return s; }
      if(i==j-1)
       {    if(a[i]<a[j]) { s.max=a[j]; s.min=a[i];}
            else { s.max=a[i]; s.min=a[j];}
            return s;
       }
       k=(i+j)/2;
       s1=MaxMin(i, k); s2=MaxMin(k+1, j);
       (s1.max>=s2.max) ? (s.max=s1.max):( s.max=s2.max);
       (s1.min<=s2.min) ? (s.min=s1.min):( s.min=s2.min);
       return s;
    }
```

输入一组数 a={22,10,60,78,45,51,8,36}，调用 MaxMin 函数，划分区间，区间划分将一直进行到只含有 1 个或 2 个元素时为止，然后求子解，并返回。上述算法执行流程如图 3.3 所示。

通过对例 3.4 执行过程的分析，可以得出分治算法设计的两个基本特征。

（1）分治法求解子集是规模相同、求解过程相同的实际问题的分解。

（2）求解过程反复使用相同的求解子集来实现的，这种过程可以使用递归函数来实现算法，也可以使用循环。用分治法设计出来的程序一般是一个递归算法，因而例 3.4 中是用递归来实现的。

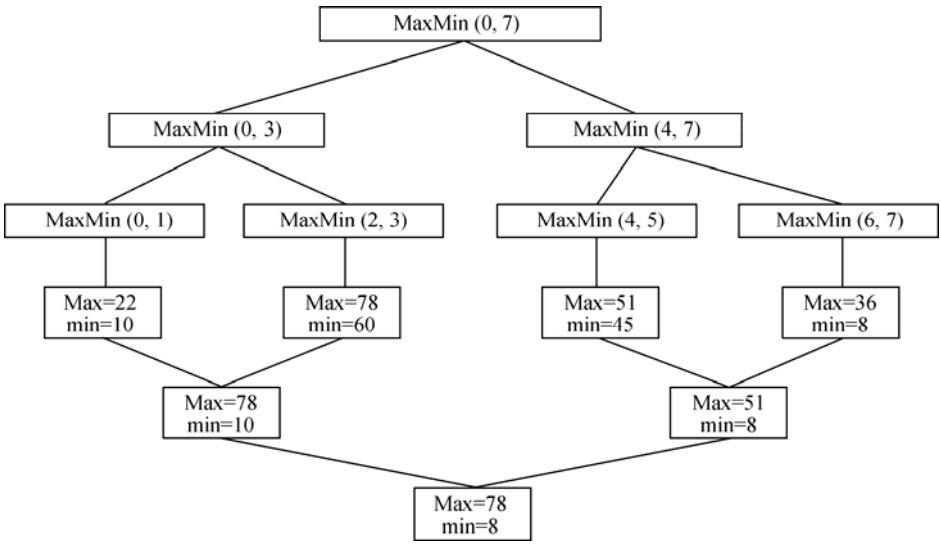


图 3.3　MaxMin 算法执行过程图

3.2.3　分治法的时间复杂度

算法的时间复杂度是衡量一个算法优劣的重要指标，在分治的过程中，一般采用的算法是一个递归算法，因而分治法的计算效率在于在递归的求解过程中的时间消耗。

对例 3.4 中 MaxMin 过程的性能分析（仅考虑算法中的比较运算），则计算时间 $T(n)$为：

$$T(n)=\begin{cases}0 & n=1\\ 1 & n=2\\ T(\lfloor n/2 \rfloor)+T(\lceil n/2 \rceil) & n>2\end{cases}$$

当 n 是 2 的幂时 $(n=2^k)$，化简上式有：

$T(n)=2T(n/2)+2$

$=2(2T(n/4)+2)+2$

…

$=2k-1T(2)+$

$=2k-1+2k-2$

$=3n/2-2$

性能比较。

1．与 StraitSearch 算法相比，比较次数减少了 25%($3n/2-2 : 2n-2$)，已经达到了以元素比较为基础的找最大最小元素的算法计算时间的下界。

2．递归调用存在以下几个问题。

（1）空间占用量大。

有$\lfloor \log n \rfloor+1$级的递归；

入栈参数有 i、j、$fmax$、$fmin$ 和返回地址 5 个值。

（2）时间可能不比预计的理想。

当元素 $A(i)$和 $A(j)$的比较与 i 和 j 的比较在时间上相差不大时，算法 MaxMin 不可取。

因为分治法设计算法的递归特性，所以若将原问题所分成的两个子问题的规模大致相等，则总的计算时间可用递归关系式表示如下：

$$T(n)=\begin{cases} g(n) & n \text{ 足够小；} \\ 2T(n/2)+f(n) & \text{否则} \end{cases}$$

其中 $T(n)$表示输入规模为 n 的计算时间；$g(n)$表示对足够小的输入规模直接求解的计算时间；$f(n)$表示对两个子区间的子结果进行合并的时间（该公式针对具体问题有各种不同的变形）。

3.3　分治法的基本应用

分治法在算法设计中有着广泛的应用，主要应用于若干不同的领域：数据的查找与排序、矩阵乘法、两个大整数相乘、马的周游路线、计数逆序排名问题、消除信号的噪声和棋盘覆盖等。它也经常与其他算法设计方法组合成一个有效的算法，例如与动态规划组合可以对基本序列比较问题产生一个空间有效的解法。本节选取其中的一些问题为例，来探讨分析分治法的应用。

3.3.1　数据查找与排序

【例 3.5】 二分检索：已知 n 个按非降次序排列的元素 $a[n]$，查找元素 x 是否在表中出现，若找到，返回其下标值，否则返回一负数。

二分检索是数据查找中典型使用分治法的实例。

原问题的描述：$(n,a[0],\cdots,a[n-1],x)$，其中 n 表示元素个数，x 表示要查找的元素；

根据分治的思想将数据分组划分为：$a[0]\sim a[k-2],a[k-1]$和 $a[k]\sim a[n-1]$；

划分后的子问题描述为：$(k-1,a[0],\cdots,a[k-2],x),(1,a[k-1],x)$和$(n-k,a[k],\cdots,a[n-1],x)$。

其算法如下：

```
int BinSearch(p,q,x)
{    int k=(p+q)/2;
     if(q<p) return –1;   /* 参数错误 */
       if(x==a[k]) return k;
       if(x<a[k]) return BinSearch1(p,k–1,x);
       if(x>a[k]) return BinSearch1(k+1,q,x);
}
```

计算复杂度分析：

算法执行过程的主体是 x 与一系列中间元素的比较。可用一棵二元树描述这一过程，称

为二元比较树，如图 3.4 所示。

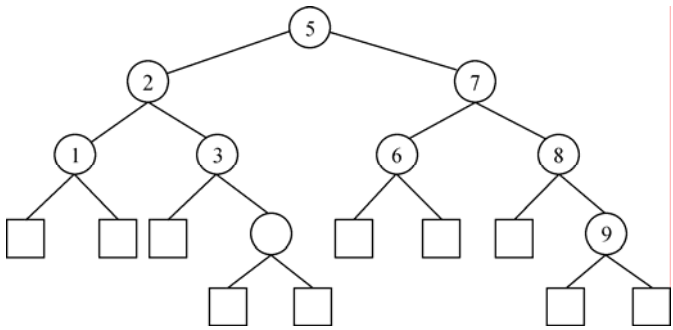


图 3.4　n=9 的二元比较树

图中结点分为内结点和外结点两种，其中。

内结点代表一次元素比较，用圆形结点表示，存放一个 mid 值（下标），代表成功检索情况；

外结点用方形结点表示，表示不成功检索情况；

路径代表检索中元素的比较序列；其中二分检索树的深为 $\log\lfloor n\rfloor+1$；二元比较树的深度为 $\log\lfloor n\rfloor+2$；

综合所述，二分检索在各种情况下的计算时间是：

成功检索			不成功检索		
最好	最坏	平均	最好	最坏	平均
$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

二分检索的一个示例为：设 $A(1:9)=(-15,-6,0,7,9,23,54,82,101)$，在 A 中检索 x=101,−14,82。执行轨迹见表 3.1。

表 3.1　　运行轨迹示例

x=101			x=−14			x=82		
low	high	mid	low	high	mid	low	high	mid
1	9	5	1	9	5	1	9	5
6	9	7	1	4	2	6	9	7
8	9	8	1	1	1	8	9	8
9	9	9	2	1	找不到			找到
		找到						
成功的检索			不成功的检索			成功的检索		

【例 3.6】 合并排序。将原始数组 A 中的元素分成两个子集合：$A1(1:\lfloor n/2\rfloor)$和 $A2(\lfloor n/2\rfloor+1:n)$。分别对这两个子集合单独分类，然后将已分类的两个子序列合并成一个含有 n 个元素的分类好的序列，这样的一个分类过程称为合并排序。

1. 合并排序步骤

（1）当 n=1 时，终止排序；

（2）否则将待排序元素分成大小大致相同的两个子集序列，再分别对 2 个子集序列进行排序；

（3）将排好序的子集进行合并，成为所要求的排好序的集合。

2. 算法描述

```
MERGESORT(low,high)
{
//A[ow:high]是一个全程数组，low 和 high 分别指示当前待分类区
//间的最小下标和最大下标，含有 high−low+1≥0 个待分类的元素
     if (low<high)   //当含有 2 个及 2 个以上的元素时，作划分与递//归处理
     {
    mid=(low+high)/2;                    //计算中分点
    call MERGESORT(low,mid)              //在第一个子集合上分类（递归）
    call MERGESORT(mid+1,high)           //在第二个子集合上分类（递归）
    call MERGE(low,mid,high)             //归并已分类的两子集合
    }
 }
MERGE(low,mid,high)
{
  //A 是一个全程数组，它含有两个放在 A1 和 A2 中的已分类的子集合。
  //目标是将这两个已分类的集合归并成一个集合，并存放到 A 中
     int h,I,j,k,;                       // low≤mid<high
     int B[high];
     h=low;i=low; j=mid+1;
     while(h<=mid && j<=high)            /当两个集合都没有取尽时,将较小的元素先存放到 B 中
       {
         if (A[h]<=A[j])   B[i]=A[h]; h=h+1  //如果前一个数组中的元素较小
          else B[i]=A[j]; j=j+1              //如果后一个数组中的元素较小
          i=i+1
       }
       // 处理尚有剩余元素的集合//
       if (h>mid)
            for (k=j;k<=high;k++) { B[i]=A[k]; i=i+1;}
       else
            for (k=h;k<=mid;k++) { B[i]=A[k];i=i+1; }
       for (k=low;k<=high;k++) A[k]=B[k] ; //将已归并的集合复制到 A 中
}
```

依照上述算法，合并排序的一个示例为：设 A=(310,285,179,652,351,423,861,254,450,520)

（1）划分过程如图 3.5 所示。

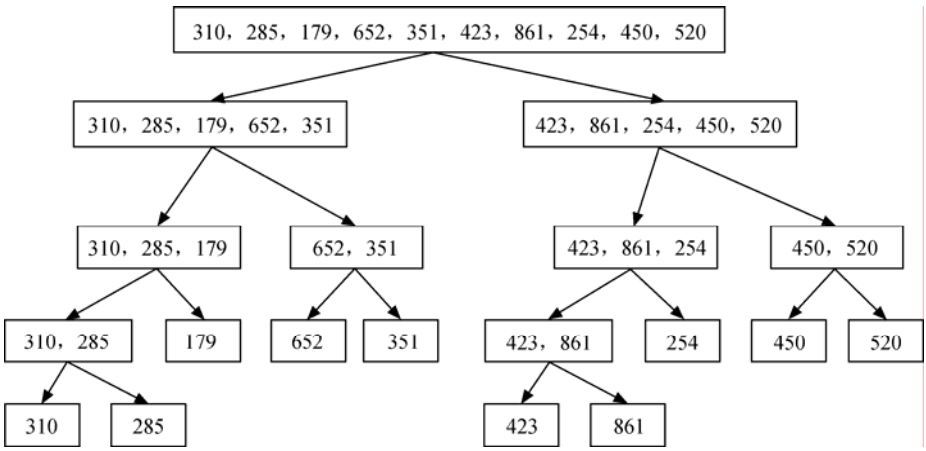


图 3.5　划分过程示意

（2）归并过程

首先进入左分枝的划分与归并。

第一次归并前的划分状态是：

(310 | 285 | 179 | 652,351 | 423,861,254,450,520)

第一次归并：(285,310 | 179 | 652,351 | 423,861,254,450,520)

第二次归并：(179,285,310 | 652,351 | 423,861,254,450,520)

第三次归并：(179,285,310 |351,652 | 423,861,254,450,520)

第四次归并：(179,285,310,351,652 | 423,861,254,450,520)

进入右分枝的划分与归并过程（略）

【例 3.7】 快速排序。

快速排序是一种基于划分的分类方法。

选取待排序集合 *A* 中的某个元素 *t*，按照与 *t* 的大小关系重新整理 *A* 中元素，使得整理后 *t* 被置于序列的某位置上，而序列中所有在 *t* 以前出现的元素均小于等于 *t*，而所有出现在 *t* 以后的元素均大于等于 *t*。这一元素的整理过程称为划分（Partitioning）。元素 *t* 称为划分元素。

所谓快速排序就是通过反复地对待排序集合进行划分达到排序目的的排序算法。

1. 划分过程

用 *A*[*m*]划分集合 *A*

```
PARTITION(m,p)
{
    int m,p,i;
    v=A[m]; i=m                          //A(m)是划分元素//
    while(1){
```

```
        while(A(i)≥v)   i=i+1;          // i 由左向右移
        while(A(p)≤v)   p=p−1;          //p 由右向左移
        if (i<p)
            call INTERCHANGE(A[i],A[p]))
        else break;
        }
    A[m]=A[p]; A[p]=v                   //划分元素在位置 p
}
```

算法对集合 A 进行划分。并使用待划分区间的第一个元素 $A[m]$作为划分元素；$A[p]$不在划分区间内，但被定义，且 $A[p]\geq A[m]$，用于限界。

2. 一个划分实例

```
    (1)   (2)   (3)   (4)   (5)   (6)   (7)   (8)   (9)   (10)    i    p
A: 65     70    75    80    85    60    55    50    45    +∞      2    9
          |.....................................................|（交换划分元素）
A: 65     45    75    80    85    60    55    50    70    +∞      3    8
              |.....................................|
A: 65     45    50    80    85    60    55    75    70    +∞      4    7
                  |.....................|
A: 65     45    50    55    85    60    80    75    70    +∞      5    6
                        |.........|
A: 65   45    50    55    60    85    80    75    70    +∞      6    5
    |.............................|
A: 60   45    50    55    65    85    80    75    70    +∞
                          |  划分元素定位于此
```

经过一次“划分”后，实现了对集合元素的调整；以划分元素为界，左边子集合的所有元素均小于等于右边子集合的所有元素。

3. 快速排序

按同样的策略对两个子集合进行分类处理。当子集合排序完毕后，整个集合的排序也完成了，该过程避免了子集合的归并操作，这一排序方法称为快速排序。

所以快速排序算法是一种通过反复使用划分过程 PARTITION 实现对集合元素排序的算法。算法如下：

```
QUICKSORT(p,q)
{
  // 将数组 A 的元素 A(p),…A(q)按递增方式分类。A(n+1)有定义，假定 A(n+1)=+∞
      int p,q;
      if (p<q) {
          j=q+1;                    //进入时，A(j)定义了划分区间[p,q]的上界，首次调用时 j=n+1
```

```
        call PARTITION(p,j);        //出口时 j 带出此次划分后划分元素所在下标位置
        call QUICKSORT(p,j−1);      //对前一子集合递归调用
        call QUICKSORT(j+1,q);      //对后一子集合递归调用
        }
}
```

3.3.2 计数逆序排名问题

这里考虑在排名分析中产生的问题，排名对当前许多应用变得重要起来。例如，许多网站使用一种叫作协同过滤的技术，利用这种技术把你（对于书、电影、餐馆）的嗜好与因特网上其他人的表现与嗜好进行匹配。另一个应用出现在搜索工具中，在不同的引擎上执行相同的查询。然后通过在搜索引擎返回的各种排名中寻找类似性与差别来综合这些结果。

在类似这样的应用中，一个核心问题是对两个排名进行比较的问题。你对一组 n 个电影排名，然后一个协同过滤系统向其数据库咨询来寻找有着“类似”排名的其他人。但是如何从数量上来度量两个人的排名有多相似呢？显然一个相等的排名是非常相似的，一个完全相反的排名是非常不同的；我们想要的是介于整个中间区域的东西。

考虑把你与一个陌生人对同一组 n 个电影进行排名比较，通常的方法是依照你的排名从 1 到 n 标记这些电影，然后依照这个陌生人的排名排序这些标记，看看有多少对“次序出错”。更具体的说，给定 n 个数的一个排序数列 $a_1a_2\cdots a_n$。假设所有的数是不同的，定义一个度量，它将告诉我们这个表与处于上升顺序的表相差多远；如果 $a_1<a_2<\cdots<a_n$，这个度量的值应该是 0，并且应该随着数变得更加杂乱而增加。

把这个概念量化的通常的方式是级数逆序的个数，我们说两个指标 $i<j$ 构成一个逆序，如果 $a_i>a_j$，即两个元素是“次序出错”的。我们想确定在 $a_1a_2\cdots a_n$ 中的逆序个数。

先放下这个定义，考虑一个例子，其中的序列是 1,4,1,3,5。在这个序列中有三个逆序：(2,1)、(4,1)、(4,3)。也存在一个几何方法把它形象化，如图 3.6 所示。把输入数的序列按照它们被提供的次序画出来，而下面的序列处于上升次序。然后在顶部表格中的每个数与下面表各种相同的数之间划一条线段。每对交叉线段于在两个表格中相反次序的一对数——即一个逆序对应。

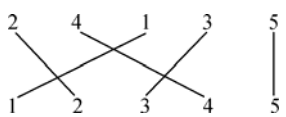


图 3.6 逆序对应图

注意逆序个数怎样作为一个度量使其平滑的介于完全一致（当序列是上升的次序，那么没有逆序）与完全不一致（如果序列是下降的次序，那么每对数构成一个逆序，并且因此存在 $\binom{n}{2}$ 个序列）之间。

什么是计数逆序的最简单算法？显然能够检查每对数 (a_i, a_j) 并且确定是否它们组成一个逆序；这将用 $O(n^2)$ 时间。现在显示怎样快得多的用 $o(n\log n)$ 时间计数逆序。注意可能存在平方数个计数序，这样一个算法必须在任何时候不用个别的看每个逆序而能够计算总数，基本的思想是令 $m=\lceil n/2 \rceil$ 并把这个表分成两部分 $a_1, a_2, \cdots, a_m, a_{m+1}, \cdots, a_n$。首先分别计录在这两半中每部分的逆序个数，然后计数逆序 (a_i, a_j)，做这部分工作，注意到这些前一

半或后一半的逆序有着一种特别好的形势：精确地说，他们是一个对(a_i, a_j)，其中a_i在前一半，a_j在后一半，且$a_i > a_j$。

为了有助于技术在两半边之间的逆序个数，对这个算法也递归的对两半边的数进行排序，让递归步骤多做一点工作，使得这个算法的组合部分更容易。于是在这个处理中的关键程序是 Merge-and-Count，假设已经递归的排序了这个表格的前一半部分和后一半部分的的逆序。那么现在有两个排好序的表 A 和 B，分别包含前一半和后一半我们想把它们合并产生一个排好的表 C 同时也计数(a,b)对的个数，其中$a \in A, b \in B$，且 $a>b$。更具我们前面的讨论，这恰好就是我们计算前一半或后一半逆序数的组合步所需要的。

这与前面讨论的比较简单的问题密切相关，这个问题构成了关于归并排序的相对应的“组合”步：在那里有两个排好序的列表 A 与 B，并且想在$o(n)$时间内把它们贵宾分成一个排好序的表，这里的区别在于多做了一点事情：不仅应该从 A 和 B 产生一个排好序的表，而且还应该计数“逆序对”(a,b)的个数，其中$a \in A, b \in B$，且 $a>b$。

原来能按照对归并所使用的几乎完全相同的风格做这件事。但 Merge-and-Count 程序将通过排好序的表 A 与 B，从前面移走元素并且将他们排列到被排列的表 C 中，在给定的一步，这里有一 Crrent 指针指向每个表，显示当前的位置。假设这些指针当前在元素a_i与a_j，把较小的元素从他的表中移走，并且把它加到表 C 的尾部。承担起了归并的任务，那么怎样的计算逆序个数呢？由于 A 和 B 是排好的，实际上非常容易记下遇到的逆序个数。每次元素a_i被加到 C 中，不会遇到新的逆序因为a_i小于表 B 中的任何一个其他元素，并且它出现在他们大家的前面。另一方面，如果b_j被加到 C 中，那么它比表 A 中所有元素都小，并且它出现在他们之后，因此把对逆序数的计数奖赏 A 中剩下的元素个数。这是一个关键的思想：在常数时间内，已经解决了潜在的大数量的逆序。算法如下：

```
Merge_and_Count(A,B)
一个 Current 指针指向每个表，初始化指向首元素
一个变量 Count 用于记录逆序的个数，初始为 0
while 两个表都不空
    令 ai,bi 是由 Current 指针指向的元素
    把这两个中较小的加到输出表中
    if bi 是较小的元素
        把 Count 加上在 A 中剩余的元素数；
    把较小元素选出的表中的 Current 指针前移
一旦一个表空，把另一个表剩余的所有元素加到输出中
返回 Count 和合并后的表
```

在一个递归过程中使用这个 Merge_and_Count()函数，这个过程排序并计数在一个表 L 中的逆序个数。

```
Sort_and_Count(L)
    If  这个表有一个元素
        没有逆序
    Else
        把这个表划分为 2 个部分
```

A 中包含前$\lceil n/2 \rceil$个元素

B 中包含后$\lceil n/2 \rceil$个元素

(r_a,A)= Sort_and_Count(A)

(r_B,B)= Sort_and_Count(B)

(r,L)= Sort_and_Count(L)

Return r= r_a+r_B+r 以及排好序的表 L

由 Merge-and-Count 过程用 $o(n)$ 时间，整个 Sort-and-Count 过程的运行时间 $T(n)$满足 $T(n) \leqslant 2T\left(\frac{n}{2}\right)+O(n)$ 有 Sort-and-Count 算法正确的对输入表排序并且计数逆数个数；他对具有 n 个元素的表运行在 $o(n\log n)$ 时间。

3.3.3 投资问题

假设有一个高计算量的投资公司咨询员，需要做多项求解，这个问题的一个典型实例如下所述。在一项模拟中，他们从过去的某点开始对一支给定的股票连续看 n 天，让我们把这些天记录为数 $i=1,2,\cdots,n$；对每天 i，他们有当天这支股票每股的价格 $p(i)$（为简单起见，假设这个价格在每一天之内是固定的）。假设在这个时间区间内，某天他们想买 1000 股并且在以后的某天卖出所有的这支股。他们想知道：为了挣到最多的钱，他们应该什么时候买且应该什么时候卖？（如果在这 n 天里没有办法挣钱，应该对此给出报告。）

举例说，假设 $n=3, p(1)=9, p(2)=1, p(3)=5$。那么你应该返回“2 买，3 卖”。

很清楚，存在一个 $O(n^2)$ 时间的简单算法：对所有的买天/卖天构成的对进行尝试，看看哪个能使他们挣到最多的钱。

显示怎样在 $O(n\log n)$ 的时间找到正确的 i和 j 。

在这一章我们已经看到一些实例，其中通过分治策略我们可以把在元素对上的穷举搜索减少到 $O(n\log n)$ 时间。这里由于面对的是类似情况，让我们考虑可以怎样应用分治策略。

一种简单的方法将是分别考虑前 $n/2$ 天和后 $n/2$ 天，对这两个集合中的每一个问题递归的求解，然后指出怎样从这些用 $O(n)$ 时间得到一个全局的解。这将给出常用的递推式 $T(n) \leqslant 2T\left(\frac{n}{2}\right)+O(n)$，因此得到 $O(n\log n)$。

此外，将使用 n 是 2 的幂，这并不减少一般性：如果 n' 是比 n 大的下一个 2 的幂，可以对在 n 与 n' 之间的所有 i，令 $p(i)=p(n)$，以这种方式，不用改变答案，至多使输入的规模加倍。

现在，设 S 是 1 天，……，n/2 天的集合，S' 时 2/n+1 天，……，n 天的集合，分治算法将基于下面的观察：有一个最优解使投资者在 n/2 天结束时持有这支股票，或者没有。如果没有，那么这个最优解是集合 S 的最优解与集合 S' 的最优解中较好的一个；如果有一个最优解使得他们在 n/2 天结束时持有这支股票，那么这个解的值是 $p(j)-p(i)$，其中 $i \in S$，且 $j \in S'$，但是这个值是通过简单选择年 i 使得 $p(i)$ 最小，并且选择 j 使得 $p(j)$ 最大而达到最大化的。

于是，算法是在下面 3 个可能的解中取最好的：

（1）S 上的最优解；

（2） S' 上的最优解；

（3） $p(j)-p(i)$，对所有的 $i \in S$, 且 $j \in S'$。

前两个选择中的每一个在 $T(n/2)$ 时间内被递归的计算，第三个选择通过找 S 中的最小和 S' 上的最大而计算，这就要用 $O(n)$ 的时间，于是，运行时间 $T(n)$ 满足：$T(n) \leqslant 2T(n/2)+O(n)$，正如所求。

注意到这并不是对这个问题所能达到的最好的运行时间，事实上，人们可以用动态规划在 $O(n)$ 时间内找到最优的 2 天，动态规划是后面的课题，在动态规划结束时，将对此问题再作为练习提出。

3.4 消除递归

3.4.1 一般的递归转非递归

前面我们讲过了递归算法，递归算法的程序设计有以下几个特点：

（1）符合人的递推求解问题的自然思维习惯；

（2）算法的结构简单，代码精炼，可读性好；

（3）递归程序比非递归的程序要花费更多的时间，效率低。

但是由于许多问题的求解，递归比非递归容易设计。因此，通常先设计出递归程序，然后再转化为非递归程序，而不是直接写出非递归程序。

对于转换过程中的等价性，我们先来讨论转换过程中的一些问题：

1．系统自动的为递归设置一个栈，因此在等价的非递归程序中，需要设置栈 S，初始为空；

2．对于递归调用的子程序入口处，在等价变换时，用无条件的转移语句实现，并在入口处设置一个标号；

3．对程序中的每个递归调用，用以下几个等价操作替换。

（1）保留现场：开辟栈顶存储空间，用于保存返回地址和调用层的形参和局部变量；

（2）准备数据：为被调子程序准备参数；

（3）执行 goto；

（4）在返回处设一个标号。

4．对返回语句，如果栈不空，则依次执行如下操作，否则结束本子程序，返回。

（1）回传数据，若有变参或函数，将值保存到回传变量中；

（2）恢复现场，从栈顶中取出所有值，退栈；

（3）返回，执行 goto 语句。

下面我们将通过 2 个示例来了解消除递归的一般步骤。

【例 3.8】 写一个递归函数 reverse (char * s)，按逆序输出一个字符串，并将此递归算法改写成相应的迭代算法。

递归算法如下：

```
void reverse (char * s)
{     if( *s!='\0' )
          {   reverse(s+1);
              putchar (*s);
          }
      return;
 }
```

输出 s="abc"的递归过程如表 3.2 所示。

表 3.2　　递归调用过程

进入递归调用, top=0				递归调用返回, top=6		
顺序	条件	栈中元素	top=, s=	顺序	条件	addr, s
1	*s='a'	stack[1]=s,('a') stack[2]=L2, (putchar)	2, s+1	1	top=6	addr=stack[6] s=stack[5]
2	*s='b'	stack[3]=s, ('b') stack[4]=L2 , (putchar)	4, s+2	2	top=4	addr=stack[4] s=stack[3]
3	*s='c'	stack[5]=s, ('c') stack[6]=L2 , (putchar)	6, s+3	3	top=2	addr=stack[2] s=stack[1]
4	*s='\0'	调用结束，转返回处理		4	top=0	完全返回

将上面的递归过程改写为非递归算法，如下：

```
void reverse (char * s)
{       extern ElemType stack[2*n+1], top=0;
    L1:
        if( *s!='\0' )
        {     stack[++top]=s;    stack[++top]=L2;    s=s+1;
              goto L1;
         L2:
              putchar(*s);
        }           /*接下来处理返回语句*/
        if(top==0 ) return;                          /*栈为空*/
        else
        {     addr=stack[top--];                     /*恢复地址*/
              s=stack[top--];                        /*恢复参数*/
              if(addr == L2) goto L2;
         }
 }
```

将上面的非递归算法进行优化，如下:

```
void reverse(char * s)
     {     int top=0;
```

```
        while(*s!='\0')
        {    top++;
             s++;
         }
         while (top!=0)
         {    putchar(*s);
              s--; top--;
          }
    }
```

【例 3.9】 写一个求数组 $a[n]$中的最大元素的递归算法并将其改写成迭代算法。

递归算法如下：

```
int max(int i)
{    int j=i, k;
     if(i<n-1)
     {    j=max(i+1);
          if (a[i]>a[j]) k=i;
          else k=j;
      }
     else k=n-1;
     return k;
}
```

改写的过程如表 3.3 所示。

表 3.3　　求最大元素过程

（0）开始，照搬（所有不涉及递归调用和返回语句的代码都照搬）
（1）定义和初始化堆栈（存储参数、局部变量、返回值和返址）
	int max(int i)
	{
	　　extern　stack[4*n+1], top=0;
	　　int j=i, k;

（2）对第一条可执行语句标号 L1，每次递归调用执行步骤
（3）将所有参数和局部变量进栈
（4）建立第 i 个返回地址的标号 Li，进栈
（5）计算本次递归调用实参的值，并赋给形参变量
（6）无条件转移到 L1 进入递归
（7）如果是带返回值的调用，从栈顶取回返回值并代替调用，其余代码按原方式处理，并将（4）建立的标号附于该语句；如果是不带返回值的调用，则将（4）建立的标号直接附于（6）对应的语句之后的那条语句
（8）如果栈为空，直接返回
（9）否则，恢复返回地址和参数：
从栈中取返回地址，并把此地址赋给一个未使用的变量；
从栈中取所有局部变量和参数，并赋给相应的变量
（10）如果这个过程是函数，插入一条语句，计算紧跟在 return 后面的表达式并将其入栈
（11）用返回地址标号的下标实现对该标号的转移
	L1:
	　　if(i<n-1)
	　　{　stack[++top]=i;
	　　　stack[++top]=j;
	　　　stack[++top]= L2;
	　　　i=i+1;
	　　　goto L1；
	L2:　j=stack[top--];
	　　　if(a[i]<a[j]) k=i;
	　　　else k=j;
	　　}
	　　else k=n-1;
	if(top==0) return k;
	　else
	　{　　addr=stack[top--];
	　　j=stack[top--];
	　　i=stack[top--];
	　　stack[++top]=k;

	if(addr==L2) goto L2;
	}
	}

优化后的非递归算法为：

```
int max (int i)
{    int j, k=n−1;
     for (j=n−2; j>=i; j−−)
         if (a[j]>a[k])
              k=j;
      return k;
}
```

3.4.2 分治算法中的递归转化

对于分治算法中的递归转化为非递归，我们在这里主要讲分治的抽象控制递归算法的转化。

分治算法的抽象控制递归算法如下：

```
SOLUTION DandC (p, q)   /* divide and conquer */
{    int m; SOLUTION s1, s2, s;
     if( small (p, q) ) s=conquer (p, q);
     else
     {     m=divide (p, q);
           s1=DandC (p, m);
           s2=DandC (m+1, q);
           s=combine (s1, s2);
     }
     return s;
}
```

依上述转化过程，转化后的结果如下：

```
SOLUTION DandC (p, q)
{ extern ElemType stack[5*n+1], top=0;
   int m; SOLUTION s1, s2;
L1: if(small(p,q)) s=conquer(p,q);
     else
     {    m=divide(p,q);
          stack[++top]=p; stack[++top]=q;
          stack[++top]=m; stack[++top]=L2;
          p=p; q=m;
```

```
                goto L1;
        L2:   s1= stack[top--];
                stack[++top]=p; stack[++top]=q;
                stack[++top]=m; stack[++top]=L3;
                p=m+1; q=q;
                goto L1;
        L3:   s2=stack[top--]; s=combine(s1,s2);
            }
    if(top==0) return s;
    else
      {   addr=stack[top--];
          m=stack[top--];
          q=stack[top--];
          p=stack[top--];
          stack[++top]=s;
          if(addr==L2)
                goto L2;
          else
                goto L3;
        }
    }
```

递归方法在算法与数据结构中的应用无所不在，如分治法、动态规划、回溯法（深度优先搜索）等，只有熟悉掌握函数递归调用的编程方法，深入理解分治策略的重要思想，才能更好的学习算法，理解算法，编写出好的程序。

习　　题

1. 某石油公司计划建造一条由东向西的主输油管道。该管道要穿过一个有 n 口油井的油田。从每口油井都要有一条输油管道沿最短路径（或南或北）与主管道相连。如果给定 n 口油井的位置，即它们的 x 坐标（东西向）和 y 坐标（南北向），应如何确定主管道的最优位置，即使各油井到主管道之间的输油管道长度总和最小的位置？证明可在线性时间内确定主管道的最优位置。

给定 n 口油井的位置，编程计算各油井到主管道之间的输油管道最小长度总和。

2. n 个元素的集合{1,2,…,n}可以划分为若干个非空子集。例如，当 n=4 时，集合{1,2,3,4}可以划分为 15 个不同的非空子集如下：

{{1},{2},{3},{4}},

{{1,2},{3},{4}},

{{1,3},{2},{4}},

{{1,4},{2},{3}},
{{2,3},{1},{4}},
{{2,4},{1},{3}},
{{3,4},{1},{2}},
{{1,2},{3,4}},
{{1,3},{2,4}},
{{1,4},{2,3}},
{{1,2,3},{4}},
{{1,2,4},{3}},
{{1,3,4},{2}},
{{2,3,4},{1}},
{{1,2,3,4}}

其中，集合{{1,2,3,4}}由 1 个子集组成；集合{{1,2},{3,4}},{{1,3},{2,4}},{{1,4},{2,3}},{{1,2,3},{4}},{{1,2,4},{3}},{{1,3,4},{2}},{{2,3,4},{1}}由 2 个子集组成；集合{{1,2},{3},{4}},{{1,3},{2},{4}},{{1,4},{2},{3}},{{2,3},{1},{4}},{{2,4},{1},{3}},{{3,4},{1},{2}}由 3 个子集组成；集合{{1},{2},{3},{4}}由 4 个子集组成。

给定正整数 n 和 m，计算出 n 个元素的集合{1,2,…,n}可以划分为多少个不同的由 m 个非空子集组成的集合。

3. 在一个按照东西和南北方向划分成规整街区的城市里，n 个居民点散乱地分布在不同的街区中。用 x 坐标表示东西向，用 y 坐标表示南北向。各居民点的位置可以由坐标(x,y)表示。街区中任意 2 点(x_1,y_1)和(x_2,y_2)之间的距离可以用数值$|x_1-x_2|+|y_1-y_2|$度量。

居民们希望在城市中选择建立邮局的最佳位置，使 n 个居民点到邮局的距离总和最小。

给定 n 个居民点的位置，编程计算 n 个居民点到邮局的距离总和的最小值。

4. 大于 1 的正整数 n 可以分解为：$n=x_1\times x_2\times\cdots\times x_m$。

例如，当 n=12 时，共有 8 种不同的分解式：

12=12;
12=6*2;
12=4*3;
12=3*4;
12=3*2*2;
12=2*6;
12=2*3*2;
12=2*2*3

对于给定的正整数 n，编程计算 n 共有多少种不同的分解式。

第 4 章 递　　推

在上一章介绍分治与递归设计基础上，本章探讨与递归紧密关联的递推算法及其在数列、数阵求解与解计数应用题方面的应用。

4.1 递推概述

在纷繁变幻的世界，所有事物都随时间的流逝发生着微妙的变化。许多现象的变化是有规律可循的，这种规律往往呈现出前因后果的关系。某种现象的变化结果与紧靠它前面变化的一个或一些结果紧密关联。递推的思想正体现了这一变化规律。

4.1.1 递推算法

所谓递推，是在命题归纳时，可以由 $n-k,\cdots,n-1$ 的情形推得 n 的情形。一个线性递推可以形式地写成

$$a_n = c_1 a_{n-1} + \cdots + c_k a_{n-k} + f(n)$$

其中 $f(n)=0$ 时递推是齐次的，否则是非齐次的。递推的一般解法要用到 n 次方程的求根。

递推关系是一种高效的数学模型，是组合数学中的一个重要解题方法，在组合计数中有着广泛的应用。在概率方面利用递推可以解决一类基本事件个数较大的概率问题。在对多项式的求解过程中，很多情况可以使用递推算法来实现。 在行列式方面，某些 n 阶行列式只用初等变换难以解决，但如果采用递推求解则显得较为容易。

递推关系不仅在各数学分支中发挥着重要的作用，由它所体现出来的递推思想在各学科领域中更是显示出其独特的魅力。

递推是利用问题本身所具有的一种递推关系求解问题的一种方法。设要求问题规模为 n 的解，当 $n=1$ 时，解或为已知，或能非常方便地得到解。能采用递推法构造算法的递推性质，能从已求得的规模为 $1,2,\cdots,i-1$ 的一系列解，构造出问题规模为 i 的解。这样，程序可从 $i=0$ 或 $i=1$ 出发，重复地由已知至 $i-1$ 规模的解，通过递推，获得规模为 i 的解，直至得到规模为 n 的解。

递推算法的基本思想是把一个复杂的庞大的计算过程转化为简单过程的多次重复，该算法充分利用了计算机的运算速度快和不知疲倦的特点，从头开始一步步地推出问题最终的结果。使用递推算法编程，既可使程序简练，又可节省计算时间。

对于一个序列来说，如果已知它的通项公式，那么要求出数列中某项之值或求数列的前 n 项之和是简单的。但是，在许多情况下，要得到数列的通项公式是困难的，甚至无法得到。然而，一个有规律的数列的相邻位置上的数据项之间通常存在着一定的关系，可以借助已知的项，利用特定的关系逐项推算出它的后继项的值，直到找到所需的那一项为止。递推算法避开了求通项公项的麻烦，把一个复杂的问题的求解，分解成连续的若干步简单运算。

递推算法的首要问题是得到相邻的数据项之间的关系，即递推关系。它针对这样一类问题：问题的解决可以分为若干步骤，每个步骤都产生一个子解（部分结果），每个子解都是由前面若干子解生成。不同的子解，其所相关的问题规模也随子解不同而递增。我们把这种由前面的子解得出后面的子解的规则称为递推关系。

我们在设计求解问题前，要通过细心的观察，丰富的联想，不断尝试推理，尽可能归纳总结其内在规律，然后再把这种规律性的东西抽象成递推数学模型。

4.1.2 递推实施步骤与描述

利用递推求解实际问题，需要掌握递推的具体描述及其实施步骤。

1. 实施递推的步骤

（1）确定递推变量

应用递推算法解决问题，要根据问题的具体实际设置递推变量。递推变量可以是简单变量，也可以是一维或多维数组。

（2）建立递推关系

递推关系是指如何从变量的前一些值推出其下一个值或从变量的后一些值推出其上一个值的公式（或关系）。递推关系是递推的依据，是解决递推问题的关键。有些问题，其递推关系是明确的，大多数实际问题并没有现成的明确的递推关系，需根据问题的具体实际，细心的观察，丰富的联想，不断尝试推理，才能确定问题的递推关系。

（3）确定初始（边界）条件

对所确定的递推变量，要根据问题最简单情形的数据确定递推变量的初始（边界）值，这是递推的基础。

（4）对递推过程进行控制

递推过程不能无休止地重复执行下去。递推过程在什么时候结束，满足什么条件结束，这是编写递推算法必须考虑的问题。

递推过程的控制通常可分为两种情形：一种是所需的递推次数是确定的值，可以计算出来；另一种是所需的递推次数无法确定。对于前一种情况，可以构建一个固定次数的循环来实现对递推过程的控制；对于后一种情况，需要进一步分析出用来结束递推过程的条件。

2. 递推算法框架描述

递推通常由循环来实现，一般在循环外确定初始（边界）条件，在设置的循环中实施递推。

下面归纳常用的递推模式并作简要的框架描述。

首先，从递推流向可分为顺推与逆推。为了实现递推，可以设置简单变量通过变量迭代实现，更多的通过设置数组来完成。简单递推问题设置一维数组实现，较复杂的递推问题需设置二维或二维以上数组。

（1）简单顺推算法

顺推即从前往后推，从已求得的规模为 1,2,…,i−1 的一系列解，推出问题规模为 i 的解，直至得到规模为 n 的解。

简单顺推算法框架描述：

```
f(1−i−1)=<初始值>;                    /* 确定初始值 */
for(k=i;k<=n;k++)
    f(k)=<递推关系式>;                /* 根据递推关系实施递推 */
printf(f(n));                         /* 输出 n 规模的解 f(n) */
```

（2）简单逆推算法

逆推即从后往前推，从已求得的规模为 n,n−1,…,i+1 的一系列解，推出问题规模为 i 的解，直至推出规模为 1 的解。

简单逆推算法框架描述：

```
f(n−i+1)=<初始值>;                    /* 确定初始值 */
for(k=i;k>=1;k−−)
    f(k)=<递推关系式>;                /* 根据递推关系实施递推 */
printf(f(1));                         /* 输出解 f(1) */
```

当规模为 i 的解为规模为 1,2,…,i−1 的解通过计算处理决定时，可设置二重循环处理这一较为复杂的递推。

（3）计算处理顺推算法

```
f(1)=a;f(2)=b;                        /* 用 a,b 赋初值 */
for(i=3;i<=n;i++)
    for(j=1;j<=i−1;j++)
        {<对 f(1−i−1)计算处理>
         f(i)=<递推关系式>;           /* 递推得 f(i) */
        }
    printf(f(n));                     /* 输出解 f(n) */
```

当规模为 i 的状态为规模为 1,2,…,i−1 的状态递推决定，且状态数不确定时，可设置二重循环处理这类的递推。

（4）状态不确定的顺推算法

```
f(1)=a;f(2)=b;k=2;                    /* 用 a,b 赋初值 */
for(i=2;i<=n;i++)
    for(j=1;j<=i−1;j++)
        {<对 f(1−i−1)计算处理>
```

```
        k++;
        f(k)=<递推关系式>;                    /* 递推得 f(k) */
       }
  printf(f(k));                               /* 输出解 f(k) */
```

较复杂的递推问题需设置二维或二维以上数组。

（5）二维数组顺推算法

设递推的二维数组为 $f(k,j)$，$1 \leqslant k \leqslant n, 1 \leqslant j \leqslant m$，由初始条件分别求得 $f(1,1), f(1,2), \cdots, f(1,m)$，需求 $f(n,m)$，则据给定的递推关系由初始条件依次顺推得 $f(2,1-m), f(3,1-m), \cdots$，直至得到规模为 n 的解 $f(n,1-m)$。

二维数组顺推算法框架描述：

```
f(1,1−m)=<初始值>;                            /* 赋初始值 */
for(k=2;k<=n;k++)
for(j=1;j<=m;j++)
   f(k,j)=<递推关系式>;                       /* 根据递推关系实施递推 */
printf(f(n,m));                               /* 输出 n 规模的解 f(n,m) */
```

二维或二维以上数组递推常用于动态规划设计的最优值求解过程。当递推关系包含两个或两个以上关系式时，通常应用多关系分级递推算法求解。

（6）多关系分级递推算法

```
f(1−i−1)=<初始值>;                            /* 赋初始值 */
for(k=i;k<=n;k++)
   if(<条件 1>)
      f(k)=<递推关系式 1>;                    /* 根据递推关系 1 实施递推 */
   if(<条件 2>)
      f(k)=<递推关系式 2>;                    /* 根据递推关系 2 实施递推 */
   ……
   if(<条件 m>)
      f(k)=<递推关系式 m>;                     /* 根据递推关系 m 实施递推 */
printf(f(n));                                 /* 输出 n 规模的解 f(n) */
```

以上算法（3）～（6）的逆推模式类似，这里从略。

如果在一重循环中可完成递推，通常其相应的时间复杂度为 $O(n)$。在实际应用中，由于递推关系的不同，往往需要二重或更复杂的循环结构才能完成递推，其相应的时间复杂度为 $O(n^2)$或更高。

4.2 递推数列

数列，是若干个相关数据的有序排列，有时又称序列。

各类数列的表现形式与构成规律不同，导致数列性质特点的差异。递推数列以及某些指

定构成条件的数列通过递推处理是适宜的。

递推数列的递推公式中的下标变量实际上就是数组元素的下标，应用数组解决各种递推数列问题较为简便。

4.2.1　裴波那契数列与卢卡斯数列

1. 问题的提出

裴波那契数列是意大利数学家裴波那契（Fibonacci）在所著《算盘全集》一书中借助“兔子生崽”引入的一个著名的递推数列。裴波那契数列的应用相当广泛，国际上已有许多关于裴波那契数列的专著与学术期刊。裴波那契数列定义为：

$$F_1 = F_2 = 1$$

$$F_n = F_{n-1} + F_{n-2} \qquad (n > 2)$$

卢卡斯数列（Lucas）是与裴波那契数列密切相关的另一个著名的递推数列，卢卡斯数列定义为：

$$L_1 = 1, L_2 = 3$$

$$L_n = L_{n-1} + L_{n-2} \qquad (n > 2)$$

试求解裴波那契数列与卢卡斯数列的第 n 项与前 n 项之和（n 从键盘输入）。

2. 递推算法设计

注意到 F 数列与 L 数列的递推关系相同，可一并处理这两个数列。设置一维数组 $f(n)$，数列的递推关系为

$$F(k)=f(k-1)+f(k-2) \qquad (k>2)$$

注意到 F 与 L 两个数列初始值不同，在输入整数 p 选择数列（p=1 时为 F 数列，p=2 时为 L 数列）后，初始条件可统一为

$$f(1)=1，f(2)=2*p-1$$

应用顺推模式（1）设计求解，从已知前二项这一初始条件出发，逐步推出第 3 项，第 4 项，……，以至推出指定的第 n 项。

至于求和，在 k 循环外给和变量 s 赋初值 $s=f(1)+f(2)$，在 k 循环内实施求和，每计算一项 $f(k)$即累加到和变量 s 中：$s=s+f(k)$。

以上递推算法的时间复杂度与空间复杂度均为 $O(n)$。

3. 裴波那契数列与卢卡斯数列的程序实现

```
/*  裴波那契数列与卢卡斯数列递推求解  */
#include <stdio.h>
void main()
{int k,n,p;
long s,f[50];
```

```
  printf("请选择 1：裴波那契数列；2：卢卡斯数列 :");
  scanf("%d",&p);                    /* 选定数列 */
  printf("求数列的第 n 项与前 n 项和,请输入 n:");
  scanf("%d",&n);
  f[1]=1;
  f[2]=2*p-1;
  s=f[1]+f[2];                       /* 数组元素与和变量赋初值 */
  for(k=3;k<=n;k++)
      {f[k]=f[k-1]+f[k-2];           /* 实施递推 */
       s+=f[k];                      /* 实施求和 */
}
  if(p==1)
     printf("F 数列");
  else
     printf("L 数列");
  printf("第%d 项为:%ld,   ",n,f[n]);
  printf("前%d 项之和为:%ld \n",n,s);
  }
```

运行程序示例：

```
请选择 1：裴波那契数列；2：卢卡斯数列：1
求数列的第 n 项与前 n 项和，请输入 n：40
F 数列第 40 项为：102334155，前 40 项之和为：267914295
```

4. 应用变量迭代求 *F-L* 数列

递推变量也可以设置为简单变量，应用变量迭代完成递推。

设置 *a,b,c* 三个变量，变量 *a,b* 赋初值；*c=a+b* 计算第 3 项。然后 *a=b,b=c*，通过 *c=a+b* 计算第 4 项。依此迭代，依次可计算到指定的第 *n* 项。

```
/*  变量迭代求 F-L 数列  */
#include <stdio.h>
void main()
{int k,n,p;
long a,b,c,s;
  printf("请选择 1：裴波那契数列；2：卢卡斯数列 :");
  scanf("%d",&p);                    /* 选定数列 */
  printf("求数列的第 n 项与前 n 项和，请输入 n:");
  scanf("%d",&n);
  a=1;
  b=2*p-1;
  s=a+b;                             /* 元素变量与和变量赋初值 */
```

```
for(k=3;k<=n;k++)
    {c=a+b;a=b;b=c;              /* 应用变量迭代实施递推 */
     s+=c;                       /* 实施求和 */
    }
if(p==1)
   printf("F 数列");
else
   printf("L 数列");
printf("第%d 项为：%ld,   ",n,c);
printf("前%d 项之和为：%ld \n",n,s);
}
```

运行程序示例：

```
请选择 1：裴波那契数列；2：卢卡斯数列：2
求数列的第 n 项与前 n 项和,请输入 n：30
L 数列第 30 项为：1860498，前 30 项之和为：4870844
```

4.2.2　分数数列

【例 4.1】 老师写出一个递推分数数列的前 6 项：1/2,3/5,4/7,6/10,8/13,9/15,…，引导学生注意观察数列的构成规律：第 i 项的分母 d 与分子 c 存在以下关系：$d=c+i$，而分子 c 为与前 $i-1$ 项中的所有分子、分母均不相同的最小正整数。

试求出该数列的第 2 008 项，并求出前 2 008 项中的最大项。

1. 算法设计

注意到递推需用到前面的所有项，设置数组 $c(i)$表第 i 项的分子，$d(i)$表第 i 项的分母（均表现为整数）。

初始条件：$c(1)=1,d(1)=2$；$c(2)=3$，$d(2)=5$。

递推关系：$d(i)=c(i)+i$; $c(i)$为与前 $i-1$ 项中的所有分子、分母均不相同的最小正整数。

前面所述裴波那契数列与卢卡斯数列的递推式是已知的，本题的递推式并不明确。已知前 $i-1$ 项，如何通过递推确定 $c(i)$?

显然 $c(i)>c(i-1)$，同时可证当 $i>2$ 时，第 i 个分数的分子 $c(i)$总小于第 $i-1$ 个分数的分母 $d(i-1)$。应用复杂顺推结构（3）完成对分子 $c(i)$递推，置 k 在区间$(c(i-1),d(i-1))$取值，k 分别与 $d(1),d(2),\cdots d(i-1)$比较，若有相同，则 k 增 1 后再比较；若没有相同的，则产生第 i 项，作赋值：$c(i)=k,d(i)=k+i$。

递推过程描述：

```
c(1)=1;d(1)=2;
c(2)=3;d(2)=5;                          /* 数组元素赋初值 */
    for(i=3;i<=n;i++)
        for(k=c(i−1)+1;k<d(i−1);k++)
```

```
        {t=0;                                   /* k 穷举探求第 i 项分子 c */
         for(j=1;j<=i-1;j++)
           if(k==d(j))                          /* 若 k 与 d(j)相同则返回 */
             {t=1;break;}
         if(t==0)
           {c(i)=k;d(i)=k+i;                    /* 给 c(i),d(i)赋值 */
            break;
           }
    }
```

为了准确求出数列前 n 项中的最大项，设最大项为第 x 项（x 赋初值 1），每产生一项（第 i 项），如果有

$$c(i)/d(i)>c(x)/d(x) \quad <=> \quad c(i)*d(x)>c(x)*d(i)$$

即第 i 项要比原最大项第 x 项还大，则作赋值 $x=i$，把产生的第 i 项确定为最大。前 n 项递推产生完毕，最大项也比较出来了。

上述递推算法的时间复杂度 $O(n^2)$。

2. 分数数列程序设计

```
/* 分数递推数列 */
#include <stdio.h>
void main()
{ int n,i,k,t,j,kmax;
  static long c[3001],d[3001];
  printf("请输入整数 n(1--3000):");            /* 键盘输入确定整数 n */
  scanf("%d",&n);
  c[1]=1;d[1]=2;
  c[2]=3;d[2]=5;
  kmax=1;                                     /* 数组与最大项数号赋初值 */
  for(i=3;i<=n;i++)
     {for(k=c[i-1]+1;k<d[i-1];k++)
        {t=0;                                 /* k 穷举探求第 i 项分子 c */
         for(j=1;j<=i-1;j++)
           if(k==d[j])
             {t=1;break;}
         if(t==0)
             {c[i]=k;d[i]=k+i;                /* 第 i 项分子 c,分母 d 赋值 */
              break;
             }
         }
```

```
    if(c[i]*d[kmax]>c[kmax]*d[i])
      kmax=i;                                   /* 比较得最大项的序号 kmax */
    }
  printf("数列的第%d 项为：%ld/%ld.\n",n,c[n],d[n]);
  printf("数列前%d 项中最大项为：\n",n);
  for(i=1;i<=n;i++)                             /* 检查有多个最大项时输出 */
    if(c[i]*d[kmax]==c[kmax]*d[i])
  printf("   第%d 项：%ld/%ld.\n",i,c[i],d[i]);
}
```

3. 运行程序示例与说明

运行程序，请输入整数 n(1--3000)：2008

数列的第 2008 项为：3249/5257.

数列前 2008 项中最大项为第 1597 项：2584/4181.

顺便指出，上述分数的分子和分母构成的数对(1,2),(3,5),(4,7),(6,10),…，常称为 Wythoff 数对。Wythoff 数对在数论和对策论中有着广泛的应用。

4.2.3　幂序列

本节探讨双幂序列与复合幂序列这两个典型的递推幂序列问题的求解。

1. 双幂序列

【例 4.2】 输出集合 $\{2^x,3^y \mid x\geqslant 1, y\geqslant 1\}$ 元素由小到大排列的双幂序列第 n 项与前 n 项之和。

（1）递推算法设计

集合由 2 的幂与 3 的幂组成，实际上给出的是两个递推关系。为了实现从小到大排列，设置 a,b 两个变量，a 为 2 的幂，b 为 3 的幂，显然 $a\neq b$，应用递推模式（6）实施递推。

设置 k 循环（k=1,2,…,n，其中 n 为键盘输入整数），在 k 循环外赋初值：a=2;b=3;s=0;在 k 循环中通过比较赋值：

当 $a<b$ 时，由赋值 $f[k]=a$ 确定为序列的第 k 项；然后 $a=a*2$，即 a 按递推规律乘 2，为后一轮比较作准备；

当 $a>b$ 时，由赋值 $f[k]=b$ 确定为序列的第 k 项；然后 $b=b*3$，即 b 按递推规律乘 3，为后一轮比较作准备。

递推过程描述：

```
a=2;b=3;                                    /* 为递推变量 a,b 赋初值 */
for(k=1;k<=n;k++)
 {if(a<b)
   {f[k]=a;a=a*2;}                          /* 用 a 给 f[k]赋值 */
 else
   {f[k]=b;b=b*3;}                          /* 用 b 给 f[k]赋值 */
```

```
    }
```

在这一算法中，变量 *a*,*b* 是变化的，分别代表 2 的幂与 3 的幂。

上述递推算法的时间复杂度与空间复杂度均为 $O(n)$。

（2）双幂序列程序实现

```
/*   双幂序列求解   */
#include <stdio.h>
void main()
{int k,n,t,p2,p3; long a,b,s,f[100];
 printf("求数列的第 n 项与前 n 项和,请输入 n:");
 scanf("%d",&n);
 a=2;b=3;s=0;p2=0;p3=0;
 for(k=1;k<=n;k++)
     { if(a<b)
          { f[k]=a;a=a*2;                    /* 用 2 的幂给 f[k]赋值 */
            t=2;p2++;                        /* t=2 表示 2 的幂，p2 为指数 */
          }
         else
          { f[k]=b;b=b*3;                    /* 用 3 的幂给 f[k]赋值 */
            t=3;p3++;                        /* t=3 表示 3 的幂，p3 为指数 */
                }
         s+=f[k];
     }
 printf("数列的第%d 项为:%ld",n,f[n]);
 if(t==2)                                    /* 对输出项进行标注 */
     printf("(2^%d)\n",p2);
 else
     printf("(3^%d)\n",p3);
 printf("数列的前%d 项之和为：%ld\n",n,s);
}
```

运行程序，输入 n=40，得

```
数列的第 40 项为：33554432 (2^25)
数列的前 40 项之和为：88632221
```

2. 复合幂序列求和

【例 4.3】 由集合 $\{2^x3^y \mid x \geqslant 0, y \geqslant 0, x+y>0\}$ 元素组成的复合幂序列，求复合幂序列的指数和 $x+y \leqslant n$（正整数 n 从键盘输入）的各项之和

$$s = \sum_{x+y=1}^{n} 2^x 3^y, x \geqslant 0, y \geqslant 0$$

（1）递推算法设计

① 确定递推关系

为探索 $x+y=i$ 时各项与 $x+y=i-1$ 时各项之间的递推规律，剖析指数和 $x+y$ 的前若干项情形：

$x+y=1$ 时，序列有 2,3，共 2 项（初始条件）；

$x+y=2$ 时，序列有 2*2=4,2*3=6,3*3=9，共 3 项；

x+y=3 时，序列有 2*2*2=8,2*2*3=12,2*3*3=18,3*3*3=27，共 4 项；

……

可归纳出以下递推关系：

$x+y=i$ 时，序列共 $i+1$ 项，其中前 i 项是 $x+y=i-1$ 时的所有 i 项分别乘 2 所得；最后一项为 $x+y=i-1$ 时的最后一项乘 3 所得（即 3 的 i 次幂）。可应用复杂顺推模式（4）实施递推。

② 递推描述

```
f(1)=2;f(2)=3;k=2;                    /* 用 2，3 赋初值 */
for(i=2;i<=n;i++)
   {for(j=1;j<=i;j++)
      {k++;
       f(k)=f(k-i)*2;                  /* 实施前 i 项乘 2 递推 */
      }
    k++;f(k)=3^i;                      /* 最后一项乘 3 */
    }
```

上述递推算法的时间复杂度为 $O(n^2)$。

（2）复合幂序列求和程序实现

```
/*   复合幂序列求和   */
#include <stdio.h>
void main()
{int i,j,k,n; long s,t,f[300];
 printf("请输入幂指数和至多为 n:");
 scanf("%d",&n);
 f[1]=2;f[2]=3;
 k=2;t=3;
 s=2+3;
 for(i=2;i<=n;i++)
   {for(j=1;j<=i;j++)
      {k++;
       f[k]=f[k-i]*2;
       s+=f[k];
      }
    t=t*3;
    k++;f[k]=t;                        /* 用 t 给 f[k]赋值 */
    s+=f[k];
```

```
        }
      printf("幂指数和至多为%d 的幂序列之和为：%ld\n",n,s);
    }
```

（3）求和递推的优化

注意到问题只需求复合幂序列之和，并不要求出复合序列的每一项，因而可以简化求和的递推关系。

① 简化求和递推关系

当 $x+y=1$ 时，$s1=2+3$；

当 $x+y=2$ 时，$s2=2^2+2\times3+3^2=2\times s1+3^2$

当 $x+y=3$ 时，$s3=2^3+2^2\times3+2\times3^2+3^3=2\times s2+3^3$

一般地，当 $x+y=k$ 时，$sk=2\times s(k-1)+3^k$

应用变量迭代，即有递推关系：

$$sk=2\times sk+3^k$$

其中 3^k 也可以通过变量迭代实现。这样可以省略数组，简化为一重循环实现复合幂序列求和，算法的时间复杂度优化为 $O(n)$。

② 优化递推求和程序实现

```
/*   复合幂序列求和递推优化   */
#include <stdio.h>
void main()
{int k,n; long s,t,sk;
 printf("请输入幂指数和至多为 n:");
 scanf("%d",&n);
 t=1;sk=1;
 s=0;
 for(k=1;k<=n;k++)
    {t=t*3;                    /* 迭代得 t=3^k */
     sk=2*sk+t;                /* 实施递推 */
     s=s+sk;
    }
 printf("幂指数和至多为%d 的幂序列之和为:%ld\n",n,s);
}
```

运行程序，输入幂指数和至多为 n：15

幂指数和至多为 15 的幂序列之和为：64439009

4.2.4 双关系递推数列

【例 4.4】 集合 M 定义如下：

（1）$1\in M$

（2）$x\in M\Rightarrow 2x+1\in M$, $3x+1\in M$

（3）再无别的数属于 M

试求集合 M 元素从小到大排列的第 500 个元素。

序列构成提供了 $2x+1,3x+1$ 两个递推关系，我们试用两种不同的递推方法设计。

1. 常规递推设计

（1）算法描述

由初始条件 $m(1)$开始，经两个递推式分别递推得 $m(2)$，$m(3)$；这两者比较，较小者确定为 $m(2)$，另一个为 $m(3)$。

然后由 $m(2)$递推得 $m(4)$、$m(5)$。$m(3)$逐个与 $m(4)$、$m(5)$比较：若 $m(3)$大，则交换，确保 $m(3)$最小；若出现 $m(3)$与某一数相等，为避免重复，另一数置为一个出界的“大数”。

一般地，已得 $m(i)$后，递推得 $m(2i)$，$m(2i+1)$；$m(i+1)$与 $m(j)(j=i+2,i+3,\cdots,2i+1)$逐个比较：若 $m(i+1)>m(j)$，则 $m(i+1)$与 $m(j)$交换，确保 $m(i+1)$最小。若出现 $m(i+1)=m(j)$，为避免重复，则置 $m(j)$为一个出界的“大数”。

设置 i（从 1～n）循环，由 $m(1)$开始递推得 $m(2)$、$m(3)$，至推得 $m(n)$结束。

递推过程描述：

```
m[1]=1;
for(i=1;i<=n;i++)
   { m(2*i)=2*m(i)+1;
     m(2*i+1)=3*m(i)+1;
     for(j=i+2;j<=2*i+1;j++)
       { if(m(i+1)>m(j))                    /*   m(i+1)与 m(j)比较 */
           {h=m(j); m(j)=m(i+1);m(i+1)=h;} /* 交换，使 m(i+1)最小 */
         if(m(i+1)==m[j])
           m[j]=20000+j;                    /* 置 m(j)为一出界大数，以避免重复 */
       }
    }
```

上述递推算法的时间复杂度为 $O(n^2)$。

（2）常规递推求解程序实现

```
/*   双关系 2x+1,3x+1 递推   */
#include <stdio.h>
void main()
{ int n,i,j,h,m[1500];
  printf("n=");
  scanf("%d",&n);
  m[1]=1;
  for(i=1;i<=n;i++)
    { m[2*i]=2*m[i]+1;
      m[2*i+1]=3*m[i]+1;
      for(j=i+2;j<=2*i+1;j++)
```

```
        { if(m[i+1]>m[j])                    /*  m(i+1)与 m(j)比较 */
            {h=m[j]; m[j]=m[i+1];m[i+1]=h;}   /* 交换，使 m(i+1)最小 */
          if(m[i+1]==m[j])
            m[j]=20000+j;                    /* 置 m(j)为一出界大数，以避免重复 */
        }
    }
  for(i=1;i<=n;i++)
    { printf("   %4d",m[i]);
      if(!(i%10))   printf("\n");
    }
}
```

2. 递推优化

（1）算法设计要点

设 n 个数在数组 m 中，$2x+1$ 与 $3x+1$ 均作为一个队列，从两队列中选一排头（数值较小者）送入数组 m 中。所谓“排头”就是队列中尚未选入 m 的最小的数（下标）。这里用 $p2$ 表示 $2x+1$ 这一列的排头的下标，用 $p3$ 表示 $3x+1$ 这一列的排头的下标。

```
if(2*m(p2)<3*m(p3))
    { m(i)=2*m(p2)+1;p2++;}
if(2*m(p2)>3*m(p3))
    { m(i)=3*m(p3)+1;p3++;}
```

特别注意：两队列若出现相等时，给 m 数组赋值后，两排头都要增 1。

```
if(2*m(p2)==3*m(p3))
    { m(i)=2*m(p2)+1;
      p2++; p3++;      /* 为避免重复项，p2，p3 均须增 1 */
    }
```

（2）优化递推程序实现

```
 /*  双关系 2x+1,3x+1 队列递推 */
 #include <stdio.h>
#define s 1000
void main()
{ int m[s],n,p2,p3,i;
  printf("n=");
  scanf("%d",&n);
  m[1]=1; p2=p3=1;                       /* 排头 p2,p3 赋初值 */
  for(i=2;i<=n;i++)
    { if(2*m[p2]<3*m[p3])                /* 分不同情形实施递推 */
        { m[i]=2*m[p2]+1; p2++;}
      else if(2*m[p2]>3*m[p3])
```

```
        { m[i]=3*m[p3]+1; p3++;}
      else
        { m[i]=2*m[p2]+1;
          p2++;p3++;                          /* 为避免重复项，p2 与 p3 均增 1 */
        }
      }
    printf("m(%d)=%5d\n",n,m[n]);
  }
```

3. 运行示例与思考

运行程序，输入 n=100，输出 m(100)=418

输入 n=500，输出 m(500)=3351

以上优化递推算法的时间复杂度为 $O(n)$，且避免了多余数组元素所占的内存空间。

有资料对本题设计程序，省去了 if(2*m[p2]==3*m[p3])时 $p2$ 与 $p3$ 需同时增 1，即忽略了两队列相等的情形，因而导致数组 m 中出现重复项，这显然与集合元素的互异性相违。

作为思考，把上述程序中的递推循环体改为以下结构是否可行？

```
      {if(2*m[p2]<3*m[p3])                  /* 分三种情形实施递推 */
        {m[i]=2*m[p2]+1; p2++;}
       if(2*m[p2]>3*m[p3])
        {m[i]=3*m[p3]+1; p3++;}
       if(2*m[p2]==3*m[p3])
        {m[i]=2*m[p2]+1; p2++;p3++;}
       }
```

4.3　递推数阵

作为一维形态的数列的推广，数阵是表现为二维形态的数据集合。常见的矩阵，方阵，数字三角形等都属于数阵。其中有些数阵是递推构造的，自然可应用递推算法处理这类数阵。

4.3.1　杨辉三角

1. 问题的提出

杨辉三角，历史悠久，是我国古代数学家杨辉揭示二项展开式各项的系数的数字三角形，奥妙无穷：每一行的首尾两数均为 1；第 k 行共 k 个数，除首尾两数外，其余各数均为上一行的肩上两数的和。图 4.1 为 5 行杨辉三角。

设计程序，打印杨辉三角形的前 n 行（n 从键盘输入）。

```
        1
      1   1
    1   2   1
  1   3   3   1
1   4   6   4   1
```

图 4.1　5 行杨辉三角形

2. 应用数组递推

（1）算法设计

考察杨辉三角形的构成规律，三角形的第 i 行有 i 个数，其中第 1 个数与第 i 个数都是 1，其余各项为它的两肩上数之和（即上一行中相应项及其前一项之和）。这一递推性质实际上体现为组合公式：

$$C_i^j = C_{i-1}^{j-1} + C_{i-1}^j \quad \text{其中 } i=3,\cdots,n;\ j=2,\cdots,i-1$$

设置二维数组 $a(n,n)$，根据构成规律实施递推：

初始值：$a(i,1)=a(i,i)=1(i=1,2,\cdots,n)$

递推关系：$a(i,j)=a(i-1,j-1)+a(i-1,j)$ $(i=3,\cdots,n;j=2,\cdots,i-1)$

应用二维数组顺推模式（5）可完成递推。

为了打印输出左右对称的等腰数字三角形，设置二重循环：i 控制打印 n 行，每一行开始换行，打印 $40-3i$ 个空格；设置 j 循环控制打印第 i 行的 i 个数组元素 $a[i][j]$。

上述递推算法的时间复杂度与空间复杂度均为 $O(n^2)$。

（2）杨辉三角形程序实现

```
/* 杨辉三角形  */
#include<stdio.h>
void main()
{ int n,i,j,k,a[20][20];
  printf("input n: ");
  scanf("%d",&n);
  for(i=1;i<=n;i++)
      {a[i][1]=1;a[i][i]=1;}                    /*  确定初始条件  */
  for(i=3;i<=n;i++)
  for(j=2;j<=i-1;j++)
      a[i][j]=a[i-1][j-1]+a[i-1][j];            /*  递推得到每一数组元素  */
  for(i=1;i<=n;i++)                             /*  控制输出 n 行  */
    { printf("\n");
      for(k=1;k<=40-3*i;k++)
          printf(" ");
      for(j=1;j<=i;j++)                         /*  控制输出第 i 行的 i 个数组元素  */
          printf("%6d",a[i][j]);
    }
}
```

3. 应用变量递推

（1）递推算法设计

杨辉三角形实际上是二项展开式各项的系数，即第 $m+1$ 行的 $m+1$ 个数分别是从 m 个元素中取 $0,1,\cdots,m$ 个元素的组合数 $c(m,0),c(m,1),\cdots,c(m,m)$。注意到

$$\frac{c(m,k)}{c(m,k-1)}=\frac{m-k+1}{k}\quad(k=1,2,\cdots,m)$$

根据这一规律，可不用数组，直接应用简单变量递推，递推关系即上式的变形

$$cmk=cmk*(m-k+1)/k$$

每推得一个数即作打印，输出指定行的杨辉三角。

上述递推算法的时间复杂度为 $O(n^2)$。

（2）应用变量递推 C 程序实现

```
/* 应用变量递推求解  */
#include<stdio.h>
void main()
{int m,n,cmk,k;
 printf("input n: ");
 scanf("%d",&n);
 for(k=1;k<=40;k++)
     printf(" ");
 printf("%6d\n",1);                          /* 输出第 1 行的"1" */
 for(m=1;m<=n-1;m++)
   { for(k=1;k<=40-3*m;k++)
         printf(" ");
     cmk=1;
     printf("%6d",cmk);                      /* 输出每行开始的"1" */
     for(k=1;k<=m;k++)
         { cmk=cmk*(m-k+1)/k;                /* 计算第 m 行的第 k 个数 */
           printf("%6d",cmk);
         }
     printf("\n");
   }
}
```

运行程序，如果输入 10，则打印 10 行杨辉三角形如图 4.2 所示。

```
                     1
                   1   1
                 1   2   1
               1   3   3   1
             1   4   6   4   1
           1   5  10  10   5   1
         1   6  15  20  15   6   1
       1   7  21  35  35  21   7   1
     1   8  28  56  70  56  28   8   1
   1   9  36  84 126 126  84  36   9   1
```

图 4.2　10 行的杨辉三角形

4.3.2 折叠方阵

定义 n 阶折叠方阵是把从指定起始数开始的 n^2 个连续整数折叠为 n 行 n 列的数方阵：起始数置于方阵的左上角，然后从起始数开始递增，按顺时针方向层层折叠地排列为顺时针折叠方阵，或按逆时针方向层层折叠地排列为逆时针折叠方阵。图 4.3 所示为起始数是 10 的 4 阶顺时针折叠方阵。

【例 4.5】 给定起始数 a 与阶数 n（a,n 从键盘输入确定），选择打印输出的顺时针或逆时针折叠方阵。

10 11 14 19
13 12 15 20
18 17 16 21
25 24 23 22

图 4.3 4 阶折叠方阵

1. 递推算法设计

设置二维数组 z，从给定的起始数 a 开始，按递增 1 取值，据顺时针折叠方阵的构造特点给二维数组 $z(n,n)$赋值。

显然起始数 a 赋值给 $z(1,1)$。

除 $z(1,1)$外，n 阶方阵还有叠折的 $n-1$ 层：

第 i 层(i=2,3,…,n)的起始位置为$(1,i)$，随后列号 y 不变行号 x 递增，至 x=i 时折转，行号 x 不变列号 y 递减，至 y=1 时该层结束，在每一位置分别按递增值赋值给 $z(x,y)$。

递推过程描述：

```
z[1][1]=a;                          /* 数组 z 赋初值 a */
for(i=2;i<=n;i++)                   /* 从第 2 层至第 n 层赋值 */
   { x=1;y=i;
     z[x][y]=a+(i-1)*(i-1);         /* 每层的第一个数赋值 */
     while(x<i)
          z[++x][y]=z[x][y]+1;      /* 给每元素递推赋值 */
     while(y>1)
          z[x][--y]=z[x][y]+1;
    }
```

赋值完成，用二重循环打印输出顺时针折叠方阵。

如果选择打印逆时针折叠方阵，赋值同上，只需在打印输出时把数组的行号与列号互换即可。

上述递推算法的时间复杂度为 $O(n^2)$。

2. 折叠方阵程序设计

```
/* 折叠方阵  */
#include<stdio.h>
void main()
 { int a,i,n,p,x,y,z[50][50];
 printf("起始数为 a 的 n 行折叠方阵,请输入 a,n:");
 scanf("%d,%d",&a,&n);
 printf("   方阵有两种折叠方式:" );
```

```
    printf("1: 顺时针折叠    2: 逆时针折叠 ");
    scanf("%d",&p);
    z[1][1]=a;                              /* 数组 z 赋初值 a */
    for(i=2;i<=n;i++)                       /* 方阵共 n 层,按层赋值 */
        {x=1;y=i;
        z[x][y]=a+(i-1)*(i-1);              /* 按折叠规律给 z 数组赋值 */
        while(x<i)
            z[++x][y]=z[x][y]+1;
        while(y>1)
            z[x][--y]=z[x][y]+1;
        }
    printf("起始数为%d 的%d 阶",a,n);
    if(p==1)
        printf("顺时针折叠: \n");
    else
        printf("逆时针折叠: \n");
    for(x=1;x<=n;x++)                       /* 打印起始数为 a,n 阶折叠方阵 */
        {for(y=1;y<=n;y++)
            if(p==1 )
              printf("%4d",z[x][y]);
            else
              printf("%4d",z[y][x]);
        printf("\n");
       }
}
```

3. 程序运行示例

运行程序，输入 a=10,n=6,p=1,2 的折叠方阵如图 4.4 所示。

10	11	14	19	26	35
13	12	15	20	27	36
18	17	16	21	28	37
25	24	23	22	29	38
34	33	32	31	30	39
45	44	43	42	41	40

(a) 顺时针折叠

10	13	18	25	34	45
11	12	17	24	33	44
14	15	16	23	32	43
19	20	21	22	31	42
26	27	28	29	30	41
35	36	37	38	39	40

(b) 逆时针折叠

图 4.4　顺时针折叠与逆时针折叠示意图

4.4 应用递推求解应用题

当我们考虑应用递推求解某些计数应用问题时，根据问题的具体实际寻找递推关系并确定初始条件是递推求解的关键。

4.4.1 猴子爬山问题

【例 4.6】 一个顽猴在一座有 30 级台阶的小山上爬山跳跃，猴子上山一步可跳 1 级，或跳 3 级，试求上山的 30 级台阶有多少种不同的爬法。

1. 算法设计

这一问题实际上是一个整数有序可重复拆分问题，试应用数组递推求解。

设爬 k 级台阶的不同爬法为 $f(k)$种，首先探求 $f(k)$的递推关系。

上山最后一步到达第 30 级台阶，完成上山，共有 $f(30)$种不同的爬法；到第 30 级之前位于哪一级呢？无非是位于第 29 级（上跳 1 级即到），有 $f(29)$种；或位于第 27 级（上跳 3 级即到），有 $f(27)$种；于是

$$f(30)=f(29)+f(27)$$

依此类推，一般地有递推关系：

$$f(k)=f(k-1)+f(k-3) \qquad (k>3)$$

初始条件：

$$f(1)=1；即\ 1=1$$

$$f(2)=1；即\ 2=1+1$$

$$f(3)=2；即\ 3=1+1+1；3=3$$

根据以上递推关系与初始条件，设置一重循环应用递推模式（1）即可求出 $f(n)$。

2. 猴子爬山程序实现

```
/* 猴子爬山 n 级，一步跨 1 级或 3 级台阶 */
#include<stdio.h>
void main()
{ int k,n; long f[1000];
   printf("请输入台阶总数 n:");
   scanf("%d",&n);
   f[1]=1;f[2]=1;f[3]=2;          /* 数组元素赋初值 */
   for(k=4;k<=n;k++)
       f[k]=f[k-1]+f[k-3];        /* 按递推关系实施递推 */
```

```
  printf("s=%ld",f[n]);
}
```

运行程序，输入 n=30，得

```
    s=58425
```

3. 问题引申

把问题引申为爬山 n 级，一步有 m 种跨法，一步跨多少级均从键盘输入。

（1）分级递推算法设计

设爬山 t 台阶级的不同爬法为 $f(t)$，设从键盘输入一步跨多少级的 m 个整数分别为 $x(1),x(2),\cdots,x(m)$（约定 $x(1)<x(2)<\cdots<x(m)<n$）。

这里的整数 $x(1),x(2),\cdots,x(m)$为键盘输入，事前并不知道，因此不能在设计时简单地确定 $f(x(1)),f(x(2)),\cdots$。

事实上，可以把初始条件放在分级递推中求取，应用多关系分级递推算法（6）完成递推。

首先探讨 $f(t)$的递推关系：

当 $t<x(1)$ 时，$f(t)=0$；$f(x(1))=1$。（初始条件）

当 $x(1)<t\leqslant x(2)$时，第 1 级递推：$f(t)=f(t-x(1))$；

当 $x(2)<t\leqslant x(3)$时，第 2 级递推：$f(t)=f(t-x(1))+f(t-x(2))$；

……

一般地，当 $x(k)<t\leqslant x(k+1)$，$k=1,2,\cdots,m-1$，有第 k 级递推：

$$f(t)=f(t-x(1))+f(t-x(2))+\cdots+f(t-x(k))$$

当 $x(m)<t$ 时，第 m 级递推：

$$f(t)=f(t-x(1))+f(t-x(2))+\cdots+f(t-x(m))$$

当 $t=x(1)$，或 $t=x(2),\cdots$，或 $t=x(m)$时，按上递推求 $f(t)$外，还要加上 1。道理很简单，因为此时 t 本身即为一个一步到位的爬法。为此，应在以上递推基础上添加：

$$f(t)=f(t)+1 \qquad (t=x(2),x(3),\cdots,x(m))$$

我们所求的目标为：

$$f(n)=f(n-x(1))+f(n-x(2))+\cdots+f(n-x(m))$$

这一递推式是我们设计的依据。

在递推设计中我们可把台阶数 n 记为数组元素 $x(m+1)$，这样处理是巧妙的，可以按相同的递推规律递推计算，简化算法设计。最后一项 $f(x(m+1))$即为所求 $f(n)$。

最后输出 $f(n)$即 $f(x(m+1))$时必须把额外所添加的 1 减去。以上分级递推算法是新颖的，其时间复杂度为 $O(nm)$，空间复杂度为 $O(n)$。

（2）分级递推 C 程序实现

```
#include <stdio.h>
void main()
{ int i,j,k,m,n,t,x[10];
  long f[200];
  printf("请输入总台阶数:");
  scanf("%d",&n); /* 输入台阶数 */
```

```
printf("一次有几种跳法:");
scanf("%d",&m);
printf( "请从小到大输入一步跳几级。\n" );
for(i=1;i<=m;i++)                          /* 输入 m 个一步跳级数 */
   { printf("第%d 个一步可跳级数:",i);
     scanf("%d",&x[i]);
   }
for(i=1;i<=x[1]-1;i++) f[i]=0;             /* 确定初始条件 */
x[m+1]=n;f[x[1]]=1;
for(k=1;k<=m;k++)
  for(t=x[k]+1;t<=x[k+1];t++)
     { f[t]=0;
       for(j=1;j<=k;j++)                   /* 按公式累加实现分级 */
          f[t]=f[t]+f[t-x[j]];
       if(t==x[k+1])                       /* t=x(k+1)时增 1 */
          f[t]=f[t]+1;
     }
printf( "共有不同的跳法种数为:" );
printf( "%d(%d",n,x[1]);                   /*  按指定格式输出结果 */
for(i=2;i<=m;i++)
   printf(",%d",x[i]);
printf( ")=%ld. \n",f[n]-1);
}
```

运行程序

```
请输入总台阶数：50
一次有几种跳法：4
请从小到大输入一步跳几级。
第 1 个一步可跳级数：2
第 2 个一步可跳级数：3
第 3 个一步可跳级数：5
第 4 个一步可跳级数：6
共有不同的跳法种数为：50(2,3,5,6)=106479771.
```

4.4.2 整币兑零问题

【例 4.7】 把一张 n 分整币兑换成 m 种零币 $x(1),x(2),\cdots,x(m)$（单位为分），求不同的兑换种数。

1. 算法设计

整币兑零实际上是一个整数无序可重复拆分问题。

因为零币的种数较多时，应用穷举显然不能胜任，考虑应用递推求解。应用递推求解的关键在于寻求递推关系。

整币为 n 个单位，m 种指定零币从键盘输入分别为 $x(1),x(2),\cdots,x(m)$（约定 $x(1)<x(2)<\cdots<x(m)<n$）个单位。

记 $a(j,i)$为整币是 i，最大零币是 $x(j)$的兑换种数。当兑去一个 $x(j)$后，整体数变为 $p=i-x(j)$，最大零币可为 $x(1),x(2),\cdots,x(j)$（可重复），于是得到以下递推关系：

$$a(j,i)=a(1,p)+a(2,p)+\cdots+a(j,p) \qquad (\text{其中 } p=i-x(j))$$

可据整币 i 能否被 $x(1)$整除确定初始条件：

$$a(1,i)=1 \qquad (\text{当 } i \text{ 能被 } x(1)\text{整除时})$$

$$a(1,i)=0 \qquad (\text{当 } i \text{ 不能被 } x(1)\text{整除时})$$

按以上递推分别计算得 $a(1,n),a(2,n),\cdots,a(m,n)$，求和即得所求的整币兑零种数为：

$$n(x(1),x(2),\cdots,x(m))=a(1,n)+a(2,n)+\cdots+a(m,n)$$

所求得的和即为整币兑零种数。以上递推算法的时间复杂度为 $O(nm^2)$，空间复杂度为 $O(nm)$。

2. 整币兑零递推程序设计

```
/*  整币兑零递推求解 */
#include<stdio.h>
void main()
{ int p,i,j,m,n,k;
  static int x[12];
  static long int a[12][1001];
  long b,s;
  printf("请输入整币值(单位数):");                    /* 输入处理数据 */
  scanf("%d",&n);
  printf("请输入零币种数:");
  scanf("%d",&m);
  printf("(从小至大依次输入每种零币值)\n");
  for(i=1;i<=m;i++)
    { printf("第%d 种零币值(单位数):",i);
      scanf("%d",&x[i]);}
  for(i=0;i<=n;i++)                                  /* 确定初始条件 */
     if(i%x[1]==0) a[1][i]=1;
     else   a[1][i]=0;
  for(s=a[1][n],j=2;j<=m;j++)                        /* 递推计算 a(2,n),a(3,n),…*/
    { for(i=x[j];i<=n;i++)
       { p=i-x[j];b=0;
         for(k=1;k<=j;k++)
             b+=a[k][p];
```

```
            a[j][i]=b;
          }
      s+=a[j][n];                          /* 累加 a(1,n),a(2,n),…*/
  }
  printf("整币兑零种数为:%ld\n",s);          /* 输出兑零种数 */
}
```

运行程序

```
请输入整币值（单位数）：1000
请输入零币种数：9
从小至大依次输入每种零币值：1,2,5,10,20,50,100,200,500
整币兑零种数为：327631321
```

4.4.3 整数划分问题

【例 4.8】 正整数 *s*（简称为和数）的划分（又称分划或拆分）是把 *s* 分成为若干个正整数（简称为零数或部分）之和，划分式中允许零数重复，且不记零数的次序。试求 *s* 共有多少个不同的划分式？展示出 *s* 的所有这些划分式。

1. 探索划分的递推关系

为了建立递推关系，先对和数 k 较小时的划分式作观察归纳：

k=2：1+1；2

k=3：1+1+1；1+2；3

k=4：1+1+1+1；1+1+2；1+3；2+2；4

k=5：1+1+1+1+1；1+1+1+2；1+1+3；1+2+2；1+4；2+3；5

由以上各划分看到，除和数本身 *k*=*k* 这一特殊划分式外，其他每一个划分式至少为两项之和。约定在所有划分式中零数作不减排列，探索和数 *k* 的划分式与和数 *k*−1 的划分式存在以下递推关系：

（1）在所有和数 *k*−1 的划分式前加零数 1 都是和数 *k* 的划分式。

（2）和数 *k*−1 的划分式的前两个零数作比较，如果第 1 个零数 *x*1 小于第 2 个零数 *x*2，则把第 1 个零数加 1 后成为和数 *k* 的划分式。

2. 递推算法设计

设置三维数组 *a*，*a*(*k*, *j*, *i*)为和数 *k* 的第 *j* 个划分式的第 *i* 个数。

从 *k*=2 开始，显然递推的初始条件为：

a(2,1,1)=1；*a*(2,1,2)=1; *a*(2,2,1)=2。

根据递推关系，实施递推：

（1）实施在 *k*−1 所有划分式前加 1 操作

```
a(k,j,1)=1;
  for(t=2;t<=k;t++)
```

```
    a(k,j,t)=a(k−1,j,t−1);    /* k−1 的第 t−1 项变为 k 的第 t 项 */
```

（2）若 $k-1$ 划分式第 1 项小于第 2 项，第 1 项加 1，变为 k 的第 i 个划分式

```
if(a(k−1,j,1)<a(k−1,j,2)
  { a(k,i,1)=a(k−1,j,1)+1;
    for(t=2;t<=k−1;t++)
      a(k,i,t)=a(k−1,j,t);
  }
```

以上递推算法的时间复杂度与空间复杂度为 $O(n^2u)$，其中 u 为　　n 划分式个数。注意到 u 随 n 增加非常快，难以估算其数量级，其时间复杂度与空间复杂度是很高的。

3. 整数划分的程序实现

```
/* 整数 s 划分展示 */
#include <stdio.h>
void main()
{ int s,i,j,k,t,u;
  static int a[21][800][21];
  printf("input s(s<=20):"); scanf("%d",&s);
  a[2][1][1]=1;a[2][1][2]=1;a[2][2][1]=2;
  u=2;
  for(k=3;k<=s;k++)
    { for(j=1;j<=u;j++)
        { a[k][j][1]=1;
          for(t=2;t<=k;t++)                    /* 实施在 k−1 所有划分式前加 1 操作 */
            a[k][j][t]=a[k−1][j][t−1];
        }
      for(i=u,j=1;j<=u;j++)
        if(a[k−1][j][1]<a[k−1][j][2])          /* 若 k−1 划分式第 1 项小于第 2 项 */
          { i++;                               /* 第 1 项加 1 为 k 的第 i 个划分式的第 1 项 */
            a[k][i][1]=a[k−1][j][1]+1;
            for(t=2;t<=k−1;t++)
              a[k][i][t]=a[k−1][j][t];
          }
      i++;a[k][i][1]=k;                        /* k 的最后一个划分式为：k=k */
      u=i;
    }
  for(j=1;j<=u;j++)                            /* 输出 s 的所有划分式 */
    { printf("%3d: %d=%d",j,s,a[s][j][1]);
      i=2;
      while(a[s][j][i]>0)
```

```
            {printf("+%d",a[s][j][i]);i++;}
          printf("\n");
        }
  }
```

运行程序，输入 s=12，得

```
input s(s<=20):12
   1: 12=1+1+1+1+1+1+1+1+1+1+1+1
   2: 12=1+1+1+1+1+1+1+1+1+1+2
   3: 12=1+1+1+1+1+1+1+1+1+3
   …
  75: 12=5+7
  76: 12=6+6
  77: 12=12
```

运行程序，输入 s=20，可得 20 的共 627 个划分式。

4. 整数划分递推设计的优化

考察以上应用三维数组 $a(k,j,i)$完成递推过程，当由 $k-1$ 的划分式推出 k 的划分式时，$k-1$ 以前的数组单元已完全闲置。为此可考虑把三维数组 $a(k,j,i)$改进为二维数组 $a(j,i)$。二维数组 $a(j,i)$表示和数是 $k-1$ 的已有划分式，根据递推关系推出 k 的划分式：

（1）把 $a(j,i)$依次存储到 $a(j,i+1)$，加上第一项 $a(j,1)=1$；这样完成在 $k-1$ 的所有划分式前加 1 的操作，转化为 k 的划分式。

```
for(t=i;t>=1;t--)
    a(j,t+1)=a(j,t);
a(j,1)=1;
```

（2）对已转化的 u 个划分式逐个检验，若其第 2 个数小于第 3 个数（相当于 $k-1$ 时的第 1 个数小于第 2 个数），则把第 2 个数加 1，去除第一个数后，作为 k 时增加的一个划分式，为第 t（t 从 u 开始，每增加一个划分式，t 增 1）划分式。

```
for(t=u,j=1;j<=u;j++)
  if(a(j,2)<a(j,3))       /* 若 k-1 划分式第 1 项小于第 2 项 */
     {t++;
     a(t,1)=a(j,2)+1;   /* 第 1 项加 1  作为 k 的第 t 个划分式的第 1 项*/
      i=3;
      while(a(j,i)>0)
          {a(t,i-1)=a(j,i);i++;}
      }
```

改进的递推设计把原有的三维数组优化为二维数组，降低了算法的空间复杂度，拓展了算法的求解范围。

5. 优化递推设计的程序实现

```
/* 整数 s 划分优化递推设计 */
#include <stdio.h>
void main()
{ int s,i,j,k,t,u;
  static int a[1600][25];
  printf("input s(s<=24):");
  scanf("%d",&s);
  a[1][1]=1;a[1][2]=1;a[2][1]=2;u=2;
  for(k=3;k<=s;k++)
    { for(j=1;j<=u;j++)
        { i=k-1;
          for(t=i;t>=1;t--)                    /* 实施在 k-1 所有划分式前加 1 操作 */
              a[j][t+1]=a[j][t];
          a[j][1]=1;
        }
      for(t=u,j=1;j<=u;j++)
        if(a[j][2]<a[j][3])                    /* 若 k-1 划分式第 1 项小于第 2 项 */
          { t++;
            a[t][1]=a[j][2]+1;                 /* 第 1 项加 1 */
            i=3;
            while(a[j][i]>0)
                {a[t][i-1]=a[j][i];i++;}
          }
      t++;a[t][1]=k;                           /* 最后一个划分式为：k=k */
      u=t;
    }
  for(j=1;j<=u;j++)                            /* 输出所有 u 个划分式 */
    { printf("%3d: %d=%d",j,s,a[j][1]);
      i=2;
      while(a[j][i]>0)
          {printf("+%d",a[j][i]);i++;}
      printf("\n");
    }
}
```

注意：因划分式的数量 u 随和数 s 增加相当迅速，尽管改进为二维数组，求解的和数 s 不可能太大。

4.5 递推与递归比较

递归和递推是计算机科学和数学中很重要的工具，在计算方法的各个领域中都有广泛的应用。

递归其实就是利用系统堆栈，实现函数自身调用，或者是相互调用的过程。在通往边界的过程中，都会把单步地址保存下来，再按照先进后出进行运算，递归的数据传送也类似。

递归的运算方法，决定了它的效率较低，因为数据要不断的进栈出栈。在应用递归时，只要输入的 n 值稍大，程序求解就比较困难。因而从计算效率来说，递推往往要高于递归。

递推免除了数据进出栈的过程，即不需要函数不断的向边界值靠拢，而直接从边界出发，逐步推出函数值。

例如，递归求 5!，递归所包含的递推和回归过程如图 3.1 所示。计算过程中同一个子问题每次遇到都要求解，显然做了大量的重复计算。

而递推从初始条件 1!=1 出发，按递推关系 $n!=n*(n-1)!$逐步直接推出 5!，其执行过程简洁得多：

1!=1（初始条件）→2!=2*1!=2→3!=3*2!=6→4!=4*3!=24→5!=5*4!=120

又如计算裴波那契数列第 5 项 $f(5)$，应用递归计算 $f(5)$ 的过程如图 4.5 所示。

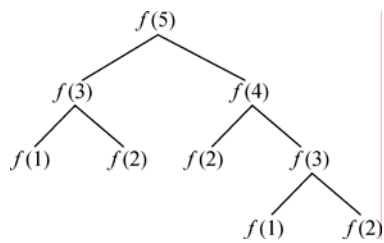


图 4.5　计算 $f(5)$的递归树

由图 4.5 可见 $f(1)$被调用了 2 次，$f(2)$被调用了 3 次，$f(3)$被调用了 2 次，作了很多重复工作。

递推计算 $f(5)$，从初始条件 $f(1)=f(2)=1$ 出发，依据递推关系 $f(n)=f(n-2)+f(n-1)$逐步直接推出 $f(5)$：

$f(1)=f(2)=1$→$f(3)=f(1)+f(2)=2$→$f(4)=f(2)+f(3)=3$

→$f(5)=f(3)+f(4)=5$

递推过程直观明了。

在有些情况下，递归可以转化为效率较高的递推。但是递归作为重要的基础算法，它的作用不可替代，在把握这两种算法的时候应该特别注意。

【例 4.9】 求整数的划分数。

正整数 s 的划分是把 s 分成为若干个正整数之和，划分式中允许零数重复，且不记零数的次序。试求 s 共有多个不同的划分式？

解：前面已经应用递推展示了整数 s 的划分式。如果不具体展示划分式，只计算划分式的个数，例 3.3 进行了计算划分式的个数的递归设计。

把递归算法设计转化为递推设计：

（1）递推算法设计

设 n 的“最大零数不超过 m”的划分式个数为 $q(n,m)$。

① 建立递推关系

所有 $q(n,m)$个划分式分为两类：零数中不包含 m 的划分式有 $q(n,m-1)$个；零数中包含 m 的划分式有 $q(n-m,m)$个，因为如果确定了一个划分的零数中包含 m，则剩下的部分就是对 $n-m$

进行不超过 m 的划分。因而有

$$q(n,m)=q(n,m-1)+q(n-m,m) \qquad (1\leqslant m<n\leqslant s)$$

其中　$q(n-m,m)=q(n-m,n-m)$　　(若 $n-m<m$)

注意到 n 等于 n 本身也为一个划分式，则有

$$q(n,n)=1+q(n,n-1)$$

② 确定初始条件

$q(n,0)=0$

$q(1,m)=1$　　($m=1,2,\cdots,s$，因整数 1 只有一个划分，不管 m 是多大)

以上的递推关系与初始条件与递归算法基本相同。

（2）递推算法描述

```
scanf("%d",&s);                              /* 输入划分的整数 */
for(m=1;m<=s;m++)
  {q[m][0]=0;q[1][m]=1;}                     /* 确定初始条件 */
for(n=2;n<=s;n++)
  {for(m=1;m<=n-1;m++)
     { if(n-m<m) q[n-m][m]=q[n-m][n-m];
       q[n][m]=q[n][m-1]+q[n-m][m];          /* 实施递推 */
     }
   q[n][n]=q[n][n-1]+1;                      /* 加上 n=n 这一个划分式 */
  }
 printf(q[s][s]);                            /* 输出递推结果 */
```

在递推设计中设置有二维数组 q[n][n]，其空间复杂度为 $O(n^2)$，显然限制了递推的计算范围。

（3）递推与递归计算效率比较

为了计算 s 的划分式的个数，第 3 章进行了递归设计，上面进行了递推设计。为了比较这两个算法的计算效率，可应用时间测试函数在不同的 s 点对递归与递推进行计算时间测试，测试结果如表 4.1 所示。

表 4.1　　**递归算法与递推算法计算时间测试结果**

整数 s	20	40	60	80	100
划分式个数	627	37338	966467	15796476	190569292
递归时间（毫秒）	0	10	130	2143	25377
递推时间	0	0	0	0	0

可见，和数 s 越大，递归与递推的计算效率相差越大。

必须说明，表中数据只是作效率的相对比较。时间为“0”并不是说不需要时间，只是因运行太快测试反映不出来。应用时间测试函数进行时间测试随系统的不同而可能不同，这次测试与下一次测试也可能存在差异，这是正常的。

习　题

1. 递推实施步骤有哪些?

2. 汉诺塔的移动次数

汉诺（Hanoi）塔问题是一个古典数学问题：在一个古塔内有 *A*、*B*、*C* 三个座，开始时 *A* 座自下而上、由大到小顺序放着 64 个圆盘。想把这 64 个盘子从 *A* 座移到 *C* 座，移动过程中可以借助 *B* 座，但每一次只能移动一个盘，且不允许大盘放在小盘的上面。

应用递推求解 *n* 个圆盘的移动次数。

3. 三元幂数列

输出集合 $\{2^x,3^y,5^z \mid x\geqslant 1,y\geqslant 1,z\geqslant 1\}$ 元素由小到大排列的幂序列第 *n* 项与前 *n* 项之和。

4. 双递推摆动数列

已知递推数列：$a(1)=1,a(2i)=a(i),a(2i+1)=a(i)+a(i+1)$（*i* 为正整数），试求该数列的第 *n* 项。

5. 旋转方阵

把整数 $1,2,\cdots,n^2$ 从外层至中心按顺时针方向螺旋排列所成的 $n\times n$ 方阵，称顺转 *n* 阶方阵；按逆时针方向螺旋排列所成的称逆转 *n* 阶方阵。

设计程序选择分别打印这二种旋转方阵。

6. 购票排队问题

一场球赛开始前，售票工作正在紧张的进行中，每张球票为 50 元。现有 $2n$ 个人排队等待购票，其中有 *n* 个人手持 50 元的钞票，另外 *n* 个人手持 100 元的钞票，假设开始售票时售票处没有零钱。问这 $2n$ 个人有多少种排队方式，使售票处不至出现找不开钱的局面?

第 5 章 贪心算法

本章首先介绍贪心算法的概念，然后给出贪心算法的理论基础，最后应用该算法给出了删数字问题、背包问题、覆盖问题、图的着色问题、遍历问题、哈夫曼编码、最小代价生成树等问题的求解方法，部分问题求解给出了详细的 C 语言代码。

5.1 贪心算法概述

有一艘大船准备用来装载货物。所有待装货物都装在货箱中且所有货箱的大小都一样，但货箱的重量都各不相同。设第 i 个货箱的重量为 $w_i(1\leqslant i\leqslant n)$，而货船的最大载重量为 c，目的是在货船上装入尽可能多的货物。

这个问题可以作为最优化问题进行描述：设存在一组变量 x_i，其可能取值为 0 或 1。如 x_i 为 0，则货箱 i 将不被装上船；如 x_i 为 1，则货箱 i 将被装上船。我们的目的是找到一组 x_i，使它满足限制条件 $n_i=\Sigma w_i x_i\leqslant c$ 且 $x_i\in[0,1],1\leqslant i\leqslant n$。相应的优化函数是 $n_i=\Sigma w_i x_i$ 取极值。满足限制条件的每一组 x_i 都是一个可行解，能使 $n_i=\Sigma w_i x_i$ 取得极大值的方案是最优解。

当一个问题具有最优子结构性质时，我们会想到用动态规划法去解它。但有时会有更简单有效的算法。举一个找硬币的例子。假设有四种硬币，它们的面值分别为二角五分、一角、五分和一分。现在要找给某顾客六角三分钱。这时，我们会不假思索地拿出 2 个二角五分的硬币，1 个一角的硬币和 3 个一分的硬币交给顾客。这种找硬币方法与其他的找法相比，所拿出的硬币个数是最少的。这里，使用了这样的找硬币算法：首先选出一个面值不超过六角三分的最大硬币，即二角五分；然后从六角三分中减去二角五分，剩下三角八分；再选出一个面值不超过三角八分的最大硬币，即又一个二角五分，如此一直做下去。这个找硬币的方法实际上就是贪心算法。

贪心算法（也称贪心策略）总是作出在当前看来是最好的选择。如上面的找硬币问题本身具有最优子结构性质，它可以用动态规划算法来解。但我们看到，用贪心算法更简单，更直接且解题效率更高，这利用了问题本身的一些特性。例如，上述找硬币的算法利用了硬币面值的特殊性，如果硬币的面值改为一分、五分和一角一分 3 种，而要找给顾客的是一角五分钱。还用贪心算法，我们将找给顾客 1 个一角一分的硬币和 4 个一分的硬币。然而 3 个五

分的硬币显然是最好的找法。

贪心算法在求解最优化问题时，从初始阶段开始，每一个阶段总是作一个使局部最优的贪心选择，不断把将问题转化为规模更小的子问题。也就是说贪心算法并不从整体最优考虑，它所作出的选择只是在某种意义上的局部最优选择。这样处理，对大多数优化问题来说能得到最优解，但也并不总是这样。从求解效率来说，贪心算法比动态规划更高，且不存在空间限制的影响。另外，对一些 NP 完全问题或规模很大的优化问题，可通过仿贪心算法能得到很好的近似解，而用动态规划根本无法解。

贪心算法的基本思想是通过一系列选择步骤来构造问题的解，每一步都是对当前部分解的一个扩展，直至获得问题的完整解。所做的每一步选择都必须满足。

（1）可行的：必须满足问题的约束。

（2）局部最优：当前所有可能的选择中最佳的局部选择。

（3）不可取消：选择一旦作出，在后面的步骤中就无法改变了。

贪心算法是通过做一系列的选择来给出某一问题的最优解，对算法的每一个决策点，做一个当时（看起来）是最佳的选择，这种启发式策略并不总是能产生出最优解。

5.2 贪心算法的理论基础

从硬币找零的问题来看，贪心算法是最接近人类认知思维的一种解题策略。但是，越是显而易见的方法往往越难以证明。下面我们简单介绍贪心算法的理论基础——“矩阵胚”理论。

“矩阵胚”理论是一种能够确定贪心算法何时能够产生最优解的理论，虽然这套理论还很不完善，但在求解最优化问题时发挥着越来越重要的作用。

定义　矩阵胚是一个序对 $\boldsymbol{M}=[\boldsymbol{S},\boldsymbol{I}]$，其中 $\boldsymbol{S}$ 是一个有序非空集合，$\boldsymbol{I}$ 是 $\boldsymbol{S}$ 的一个非空子集，成为 $\boldsymbol{S}$ 的一个独立子集。

如果 $\boldsymbol{M}$ 是一个 $n\times m$ 的矩阵的话，即：

$$\boldsymbol{M}=\begin{bmatrix} a_{11},a_{12},\cdots,a_{1m} \\ a_{21},a_{22},\cdots,a_{2m} \\ \cdots \\ a_{n1},a_{n2},\cdots,a_{nm} \end{bmatrix}$$

$\boldsymbol{S}$ 是 $\boldsymbol{M}$ 的各行，$\boldsymbol{S}=(a_1,a_2,\cdots,a_n)$，$I$ 是线性无关的若干行：$a_i,a_j,a_p,\cdots$。

若 $\boldsymbol{M}$ 是无向图 $\boldsymbol{G}$ 的矩阵胚的话，则 $\boldsymbol{S}$ 为图的边集，$\boldsymbol{I}$ 是所有构成森林的一组边的子集。

如果对 $\boldsymbol{S}$ 的每一个元素 $X(X\in\boldsymbol{S})$赋予一个正的权值 $W(X)$，则称矩阵胚 $\boldsymbol{M}=(\boldsymbol{S},\boldsymbol{I})$为一个加权矩阵胚。

适宜于用贪心算法来求解的许多问题都可以归结为在加权矩阵胚中找一个具有最大权值的独立子集的问题，即给定一个加权矩阵胚，$\boldsymbol{M}=(\boldsymbol{S},\boldsymbol{I})$，若能找出一个独立且具有最大可能权值的子集 $\boldsymbol{A}$，且 $\boldsymbol{A}$ 不被 $\boldsymbol{M}$ 中比它更大的独立子集所包含，那么 $\boldsymbol{A}$ 为最优子集，也是一个最大的独立子集。

矩阵胚理论对于我们判断贪心算法是否适用于某一复杂问题是十分有效的。

5.3　删数字问题

对给定的 n 位高精度正整数，去掉其中 $k(k<n)$个数字后，按原左右次序将组成一个新的正整数，使得剩下的数字组成的新数最大。

操作对象是一个可以超过有效数字位数的 n 位高精度数，存储在数组 a 中。

每次删除一个数字，选择一个使剩下的数最大的数字作为删除对象。之所以选择这样“贪心”的操作，是因为删 k 个数字的全局最优解包含了删一个数字的子问题的最优解。

当 k=1 时，在 n 位整数中删除哪一个数字能达到最大的目的？从左到右每相邻的两个数字比较:若出现增，即左边小于右边，则删除左边的小数字。若不出现减，即所有数字全部升序，则删除最右边的大数字。

当 k>1（当然小于 n），按上述操作一个一个删除。删除一个达到最大后，再从头即从串首开始，删除第 2 个，依此分解为 k 次完成。

若删除不到 k 个后已无左边小于右边的增序，则停止删除操作，打印剩下串的左边 $n-k$ 个数字即可（相当于删除了若干个最右边的数字）。

下面我们给出采用贪心算法的删数字问题的 C 语言代码：

```
 /* 贪心删数字 */
#include<stdio.h>
void main()
{ int i,j,k,m,n,t,x,a[200];
   char b[200];
   printf("请输入整数：");
   scanf("%s",b);
   for(n=0,i=0;b[i]!='\0';i++)
      {n++;a[i]=b[i] -48;}
   printf("删除数字个数 k:");scanf("%d",&k);
   printf("以上%d 位整数中删除%d 个数字分别为: ",n,k);
 i=0;m=0;x=0;
 while(k>x && m==0)
    {i=i+1;
    if(a[i-1]<a[i])                    /* 出现递增,删除递增的首数字 */
        {printf("%d ",a[i-1]);
         for(j=i-1;j<=n-x-2;j++)
             a[j]=a[j+1];
         x=x+1;                        /* x 统计删除数字的个数 */
         i=0;                          /* 从头开始查递增区间 */
        }
```

```
    if(i==n-x-1)                    /* 已无递增区间,m=1 脱离循环 */
        m=1;
    }
  printf("\n 删除后所得最大数: ");
  for(i=1;i<=n-k;i++)               /* 打印剩下的左边 n-k 个数字 */
    printf("%d",a[i-1]);
}
```

运行程序示例：

请输入整数：762091754639820463

删除数字个数：6

以上 18 位整数中删除 6 个数字分别为：0 2 6 7 1 4

删除后所得最大数：975639820463

5.4 背包问题

5.4.1 0-1 背包问题

0-1 背包问题中，需对容量为 c 的背包进行装载。从 n 个物品中选取装入背包的物品，每件物品 i 的重量为 w_i，价值为 p_i。对于可行的背包装载，背包中物品的总重量不能超过背包的容量，最佳装载是指所装入的物品价值最高，即 $n_i=\sum p_i x_i$ 取得最大值。约束条件为 $n_i=\sum w_i x_i \leqslant c$ 和 $x_i \in [0,1](1 \leqslant i \leqslant n)$。

在这个表达式中，需求出 x_i 的值。x_i=1 表示物品 i 装入背包中，x_i=0 表示物品 i 不装入背包。0-1 背包问题是一个一般化的货箱装载问题，即每个货箱所获得的价值不同。如船的货箱装载问题转化为背包问题的形式为：船作为背包，货箱作为可装入背包的物品。

0-1 背包问题有好几种贪心算法，每个贪心算法都采用多步过程来完成背包的装入。在每一步过程中利用贪心准则选择一个物品装入背包。一种贪心准则为：从剩余的物品中，选出可以装入背包的价值最大的物品，利用这种规则，价值最大的物品首先被装入（假设有足够容量），然后是下一个价值最大的物品，如此继续下去。这种策略不能保证得到最优解。例如，考虑 n=2,w=[100,10,10],p=[20,15,15],c=105。当利用价值贪心准则时，获得的解为 x=[1,0,0]，这种方案的总价值为 20。而最优解为[0,1,1]，其总价值为 30。

另一种方案是重量贪心准则：从剩下的物品中选择可装入背包的重量最小的物品。虽然这种规则对于前面的例子能产生最优解，但在一般情况下则不一定能得到最优解。考虑 n=2,w=[10,20],p=[5,100],c=25。当利用重量贪心算法时，获得的解为 x=[1,0]，比最优解[0,1]要差。

还可以利用另一方案，价值密度 p_i/w_i 贪心算法，这种选择准则为：从剩余物品中选择可装入包的 p_i/w_i 值最大的物品，这种策略也不能保证得到最优解。利用此策略试解 n=3,w=[20,15,15],p=[40,25,25],c=30 时的最优解。

0-1 背包问题是一个 NP-复杂问题。对于这类问题，也许根本就不可能找到具有多项式

时间的算法。虽然按 p_i/w_i 非递（增）减的次序装入物品不能保证得到最优解，但它是一个直觉上近似的解。我们希望它是一个好的启发式算法，且大多数时候能很好地接近最后算法。在 600 个随机产生的背包问题中，用这种启发式贪心算法来解有 239 题为最优解。有 583 个例子与最优解相差 10%，所有 600 个答案与最优解之差全在 25 %以内。该算法能在 $O(n\log n)$ 时间内获得如此好的性能。那么是否存在一个 $x(x<100)$，使得贪心启发法的结果与最优值相差在 x%以内。答案是否定的。为说明这一点，考虑例子 n=2,w=[1,y],p= [10,9y]，和 c=y。贪心算法结果为 x=[1,0]，这种方案的值为 10。对于 $y\geqslant 10/9$，最优解的值为 9y。

因此，贪心算法的值与最优解的差对最优解的比例为$((9y-10)/9y*100)$%，对于大的 y，这个值趋近于 100%。但是可以建立贪心启发式方法来提供解，使解的结果与最优解的值之差在最优值的 x% ($x<100$) 之内。首先将最多 k 件物品放入背包，如果这 k 件物品重量大于 c，则放弃它。否则，剩余的容量用来考虑将剩余物品按 pi/wi 递减的顺序装入。通过考虑由启发法产生的解法中最多为 k 件物品的所有可能的子集来得到最优解。

考虑 n=4,w=[2,4,6,7],p=[6,10,12,13],c=11。当 k=0 时，背包按物品价值密度非递减顺序装入，首先将物品 1 放入背包，然后是物品 2，背包剩下的容量为 5 个单元，剩下的物品没有一个合适的，因此解为 x=[1,1,0,0]。此解获得的价值为 16。

现在考虑 k=1 时的贪心启发法。最初的子集为{1}、{2}、{3}、{4}。子集{1}、{2}产生与 k=0 时相同的结果，考虑子集{3}，置 x_3 为 1。此时还剩 5 个单位的容量，按价值密度非递增顺序来考虑如何利用这 5 个单位的容量。首先考虑物品 1，它适合，因此取 x_1 为 1，这时仅剩下 3 个单位容量了，且剩余物品没有能够加入背包中的物品。通过子集{3}开始求解得结果为 x=[1,0,1,0]，获得的价值为 18。若从子集{4}开始，产生的解为 x=[1,0,0,1]，获得的价值为 19。考虑子集大小为 0 和 1 时获得的最优解为[1,0,0,1]。这个解是通过 k=1 的贪心启发式算法得到的。

若 k=2，除了考虑 $k<2$ 的子集，还必需考虑子集{1,2}、{1,3}、{1,4}、{2,3}、{2,4}和{3,4}。首先从最后一个子集开始，它是不可行的，故将其抛弃，剩下的子集经求解分别得到如下结果：[1,1,0,0]、[1,0,1,0]、[1,0,0,1]、[0,1,1,0]和[0,1,0,1]，这些结果中最后一个价值为 23，它的值比 k=0 和 k=1 时获得的解要高，这个答案即为启发式方法产生的结果。

这种修改后的贪心启发方法称为 k 阶优化方法（k-optimal）。也就是，若从答案中取出 k 件物品，并放入另外的 k 件，获得的结果不会比原来的好，而且用这种方式获得的值在最优值的$(100/(k+1))$%以内。当 k=1 时，保证最终结果在最佳值的 50%以内；当 k=2 时，则在 33.33%以内等，这种启发式方法的执行时间随 k 的增大而增加，需要测试的子集数目为 $O(nk)$，每一个子集所需时间为 $O(n)$，因此当 $k>0$ 时总的时间开销为 $O(nk+1)$，实验得到的性能要好得多。对于背包问题的更一般的情况，也可称之为可拆物品背包问题。

5.4.2　可拆背包问题

已知 n 种物品和一个可容纳 c 重量的背包，物品 i 的重量为 w_i，产生的效益为 p_i。装包时物品可拆，即可只装每种物品的一部分。显然物品 i 的一部分 x_i 放入背包可产生的效益为 $x_i p_i$，这里 $0\leqslant x_i \leqslant 1$, $p_i>0$。问如何装包，使所得整体效益最大。

（1）算法设计

应用贪心算法求解。每一种物品装包，由$0 \leqslant x_i \leqslant 1$，可以整个装入，也可以只装一部分，也可以不装。

约束条件：$\sum_{1 \leqslant i \leqslant n} w_i x_i \leqslant c$

目标函数：$\max \sum_{1 \leqslant i \leqslant n} p_i x_i$

$$0 \leqslant x_i \leqslant 1,\ p_i > 0,\ w_i > 0,\ 1 \leqslant i \leqslant n;\ \sum_{1 \leqslant i \leqslant n} w_i x_i \leqslant c$$

要使整体效益即目标函数最大，按单位重量的效益非增次序一件件物品装包，直至某一件物品装不下时，装这种物品的一部分把包装满。

解背包问题贪心算法的时间复杂度为 $O(n)$。

（2）物品可拆背包问题 C 程序设计代码如下：

```
/*  可拆背包问题 */
#include <stdio.h>
#define N 50
void main()
{float p[N],w[N],x[N],c,cw,s,h;
 int i,j,n;
 printf("\n input n:"); scanf("%d",&n);          /*  输入已知条件  */
 printf("input c:"); scanf("%f",&c);
 for(i=1;i<=n;i++)
     {printf("input w%d,p%d:",i,i);
      scanf("%f,%f",&w[i],&p[i]);
     }
for(i=1;i<=n-1;i++)                              /*  对 n 件物品按单位重量的效益从大到小排序 */
for(j=i+1;j<=n;j++)
    if(p[i]/w[i]<p[j]/w[j])
    { h=p[i];p[i]=p[j]; p[j]=h;
        h=w[i];w[i]=w[j]; w[j]=h;
    }
cw=c;s=0;                                        /* cw 为背包还可装的重量 */
for(i=1;i<=n;i++)
{if(w[i]>cw) break;
    x[i]=1.0;                                    /*  若 w(i)<=cw,整体装入*/
    cw=cw-w[i];
    s=s+p[i];
}
x[i]=(float)(cw/w[i]);                           /*  若 w(i)>cw,装入一部分 x(i) */
s=s+p[i]*x[i];
```

```
printf("装包：");                                    /*   输出装包结果 */
for(i=1;i<=n;i++)
   if(x[i]<1)   break;
   else
      printf("\n 装入重量为%5.1f 的物品.",w[i]);
if(x[i]>0 && x[i]<1)
         printf("\n 装入重量为%5.1f 的物品百分之%5.1f.",w[i],x[i]*100);
printf("\n 所得最大效益为：%7.1f ",s);
}
```

运行程序，

```
input n:5
input c:90.0
input w1,p1:32.5,56.2
input w2,p2:25.3,40.5
input w3,p3:37.4,70.8
input w4,p4:41.3,78.4
input w5,p5:28.2,40.2
装包：装入重量为 41.3 的物品.
        装入重量为 37.4 的物品.
        装入重量为 32.5 的物品百分之 34.8.
 所得最大效益为：168.7
```

5.5 覆盖问题

二分图是一个无向图，它的 n 个顶点可二分为集合 A 和集合 B，且同一集合中的任意两个顶点在图中无边相连（即任何一条边都是一个顶点在集合 A 中，另一个在集合 B 中）。当且仅当 B 中的每个顶点至少与 A 中一个顶点相连时，A 的一个子集 A' 覆盖集合 B（或简单地说，A' 是一个覆盖）。覆盖 A' 的大小即为 A' 中的顶点数目。当且仅当 A' 是覆盖 B 的子集中最小的时，A' 为最小覆盖。

图 5.1 所示的二分图就是具有 17 个顶点的二分图，A={1,2,3,16,17}和 B={4,5,6,7,8,9,10,11,12,13,14,15}，子集 A' = {1,16,17}是 B 的最小覆盖。在二分图中寻找最小覆盖的问题为二分覆盖问题。最小覆盖是很有实用价值的，因为它能解决“在会议中使用最少的翻译人员

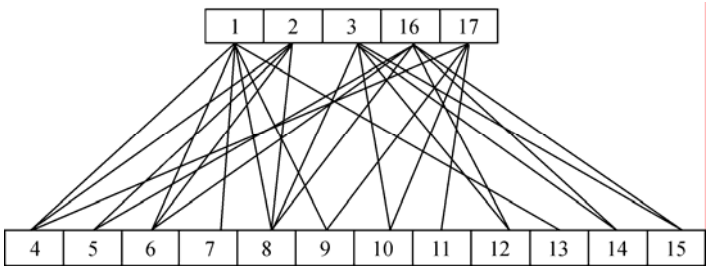


图 5.1　17 个顶点的二分图

进行翻译”这一类的问题。

二分覆盖问题类似于集合覆盖问题。可以将集合覆盖问题转化为二分覆盖问题（反之亦然），即用 ***A*** 的顶点来表示 $S_1,\cdots,S_k$，***B*** 中的顶点代表 ***U*** 中的元素。当且仅当 ***S*** 的相应集合中包含 ***U*** 中的对应元素时，在 ***A*** 与 ***B*** 的顶点之间存在一条边。

例：$S=\{S_1,\cdots,S_5\}$,$U=\{4,5,\cdots,15\}$,$S_1=\{4,6,7,8,9,13\}$,$S_2=\{4,5,6,8\}$,$S_3=\{8,10,12,14,15\}$,$S_4=\{5,6,8,12,14,15\}$,$S_5=\{4,9,10,11\}$。$S'=\{S_1,S_4,S_5\}$是一个大小为 3 的覆盖，没有更小的覆盖，S'即为最小覆盖。这个集合覆盖问题可映射为图 4-1 所示的二分图，即用顶点 1、2、3、16 和 17 分别表示集合 S_1、S_2、S_3、S_4 和 S_5，顶点 j 表示集合中的元素 j，$4\leqslant j\leqslant 15$。

集合覆盖问题为 NP-复杂问题。由于集合覆盖与二分覆盖是同一类问题，二分覆盖问题也是 NP-复杂问题。因此可能无法找到一个快速的算法来解决它，但是可以利用贪心算法寻找一种快速启发式方法。一种可能是分步建立覆盖 ***A′***，每一步选择 ***A*** 中的一个顶点加入覆盖。顶点的选择利用贪心准则：从 ***A*** 中选取能覆盖 ***B*** 中还未被覆盖的元素数目最多的顶点。

考察图 4.1 所示的二分图，初始化 ***A′***且 ***B*** 中没有顶点被覆盖，顶点 1 和 16 均能覆盖 ***B*** 中的 6 个顶点，顶点 3 覆盖 5 个，顶点 2 和 17 分别覆盖 4 个。因此，在第一步往 ***A′***中加入顶点 1 或 16，若加入顶点 16，则它覆盖的顶点为{5,6,8,12,14,15}，未覆盖的顶点为{4,7,9,10,11,13}。顶点 1 能覆盖其中 4 个顶点（{4,7,9,13}），顶点 2 覆盖一个({4})，顶点 3 覆盖一个（{10}），顶点 16 覆盖零个，顶点 17 覆盖 4 个{4,9,10,11}。下一步可选择 1 或 17 加入 ***A′***。若选择顶点 1，则顶点{10,11}仍然未被覆盖，此时顶点 1、2、16 不覆盖其中任意一个，顶点 3 覆盖一个，顶点 17 覆盖两个，因此选择顶点 17，至此所有顶点已被覆盖，得 $A'=\{16,1,17\}$。

我们给出覆盖问题的贪心算法的伪代码，可以证明：

（1）当且仅当初始的二分图没有覆盖时，算法找不到覆盖；

（2）启发式方法可能找不到二分图的最小覆盖。

算法描述如下：

```
m=0; //当前覆盖的大小
对于 A 中的所有 i，Malloc[i]=Degree[i]
对于 B 中的所有 i，Cov[i]=false
while (对于 A 中的某些 i,Malloc[i]>0) {
    设 v 是具有最大的 New[i]的顶点；
    C[m++]=v ;
    for(  所有邻接于 v 的顶点 j) {
```

```
    if(!Cov[j]) {
    Cov[j]= true;
    对于所有邻接于 j 的顶点，使其 N e w [ k]减 1
    } } }
  if (有些顶点未被覆盖)  失败
  else  找到一个覆盖
```

该算法的时间复杂度取决于数据的存储方式，若使用邻接矩阵，则需花 $O(n^2)$的时间来寻找图中的边；若用邻接链表，则需$(n+e)$的时间。故覆盖算法总的复杂性为 $O(n^2)$或 $O(n+e)$。

5.6　图的着色问题

图着色问题是其实对应很多应用的例子，假设要在足够多的会场里安排一批活动，并希望使用尽可能少的会场，设计一个有效的贪心算法进行安排。(这个问题实际上是著名的图着色问题。若将每一个活动作为图的一个顶点，不相容活动间用边相连。使相邻顶点着有不同颜色的最小着色数，相应于要找的最小会场数。)

活动安排问题：设有 n 个活动 $a_1,a_2,\cdots,a_n$，每个活动都要求使用同一资源，而同一时间内只允许一个活动使用这一资源。已知活动 a_i 要求使用该资源的起始时间为 s_i，结束时间为 f_i，且 $s_i \leqslant f_i$ 。要求在 $a_1,a_2,\cdots,a_n$ 中选出最大的相容活动子集。

两活动 $a_i,a_j\ (i\neq j)$相容是指：$[s_i,f_i)\cap[s_j,f_j)=\Phi$ 。

例：n=4，活动：a_1　a_2　a_3　a_4

s_i：　4　2　1　8

f_i：　7　4　5　10

贪心选择策略：按结束时间从早到晚安排活动，每次选择与已选出的活动相容的结束时间尽可能早的活动。

局部最优性：每次为资源留下尽可能多的时间以容纳其他活动。

该算法的时间复杂性：$\Theta(n\log n)$。

使用贪心算法求解图的着色问题并不总能得到最优解，只保证得到近似最优解。其实，顶点的编号方法决定了这种算法的结果。至少存在一种编号方法使这个贪心算法能得到最优解。下面给出图的着色问题的 C 语言程序。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#define MAX_BUF 1024
int ** graph;
void usage(char *argv)
```

```
{
printf("Usage %s [−h] [−p inputfile] [−n number_of_nodes]\n",argv);
    exit(0);
}
void my_exit(char * msg,int quit)
 {
     printf("%s\n",msg);
     exit(quit);0
}
void color_up(int nodes)    //着色函数
{
       int i,j,k = 1;
        int *color;
        color = (int*)malloc((sizeof(int))*(nodes+1));                  //初始化颜色数
        for(i=1; i<= nodes; i++)
              color[i] = 0;
        color[1]=1;
        do
        {
             for(i=2; i<=nodes; i++)
                {
                   if(color[i] > 0) continue;
                   for(j = 1; j < i ; j ++)
                         if((color[j] == k) && (graph[i−1][j−1] != 0)) break;
                   if(j == i) color[i] = k;
                }
                j = 2;
                while((color[j] > 0) && (j <= nodes)) j++;
                k++;
        }while(j <= nodes);
        for(i = 1; i <=nodes; i ++)
           printf("node %d color: %d\n",i,color[i]);
}
int main(int argc,char *argv[])
{
        int opt,nodes_num = 0,i,j,k;
        char * pfile = NULL;
        FILE * fpfile;
        char * head;
```

```
char * get;
char buf[MAX_BUF];
while((opt = getopt(argc,argv,"hp:n:")) != EOF)
{
    switch(opt)
        {
          case 'h':
                usage(argv[0]);
                break;
          case 'p':
                pfile = optarg;
                break;
          case 'n':
                nodes_num = atoi(optarg);
                if(nodes_num <= 0)
                    my_exit("Please tell me how many nodes!",100);
                break;
                default: usage(argv[0]);
                    break;
        }
}
if((fpfile = fopen(pfile,"r")) == NULL)
  {
    my_exit("I can't open input file!",99);
  }
else
{
    //为图申请空间
    graph = (int **) malloc(sizeof(int *) * nodes_num);
    for(i = 0; i < nodes_num; i ++)
        graph[i] = (int *)malloc(sizeof(int) * nodes_num);
      //初始化图
      for(i = 0; i < nodes_num; i ++)
      {
          if(fgets(buf,MAX_BUF,fpfile)!= NULL)
          {
              head = buf;
              for(j = 0; j < (nodes_num - 1); j ++)
                {
```

```
                                get = strstr(head," ");
                                if(get != NULL)
                                   *get = '\0';
                                else
                                 {
                                    for(i = 0; i < nodes_num; i ++)
 free(graph[i]);
 free(graph);
 my_exit("Read input file error!",98);
  }
 graph[i][j] = atoi(head);
 printf("%d ",graph[i][j]);
 head = get + 1;
                                      }
 graph[i][j] = atoi(head);
 printf("%d ",graph[i][j]);
 printf("\n");
                                         }
  else
  {
    for(i = 0; i < nodes_num; i ++)
 free(graph[i]);
              free(graph);
              my_exit("Invalid input file!",98);
           }
                    }
          color_up(nodes_num);
 }
                                                          //释放图空间
for(i = 0; i < nodes_num; i ++)
         free(graph[i]);
  free(graph);
  return 1;}
```

5.7 遍历问题

马的遍历问题。在 $n \times n$ 方格的棋盘上，从任意指定方格出发，为马寻找一条走遍棋盘每

一格并且只经过一次的一条路径。在这里仅以 8×8 的棋盘为例。

1. 算法初步描述

首先这是一个搜索问题，运用深度优先搜索进行求解。算法如下：

（1）输入初始位置坐标 x,y；

（2）初始化 c：

如果 c>64 输出一个解，返回上一步骤 c--

$$(x,y)\leftarrow c$$

计算(x,y)的八个方位的子结点，选出那此可行的子结点

循环遍历所有可行子结点，c++

重复（2）

显然（2）是一个递归调用的过程，大致如下：

```
void dfs(int x,int y,int count)
{
    int i,tx,ty;
if(count>N*N)
    {
        output_solution();                          //输入一个解
    return;
    }
    for(I=0;i<8;++i)
    {
        tx=hn[i].x;//hn[]保存八个方位子结点
    ty=hn[i].y;
        s[tx][ty]=count;
    dfs(tx,ty,count+1);                             //递归调用
        s[tx][ty]=0;
    }
}
```

这样做是完全可行的，它输入的是全部解，但是马遍历当 8×8 时解是非常之多的，用天文数字形容也不为过，这样一来求解的过程就非常慢，并且出一个解也非常慢。

2. 快速地得到部分解

其实马踏棋盘的问题很早就有人提出，且早在 1823 年，J.C.Warnsdorff 就提出了一个有名的算法。在每个结点对其子结点进行选取时，优先选择“出口”最小的进行搜索，“出口”的意思是在这些子结点中它们的可行子结点的个数，也就是“孙子”结点越少的越优先跳，为什么要这样选取，这是一种局部调整最优的做法，如果优先选择出口多的子结点，那出口少的子结点就会越来越多，很可能出现“死”结点（顾名思义就是没有出口又没有跳过的结点），这样对下面的搜索纯粹是徒劳，这样会浪费很多无用的时间，反过来如果每次都优先选

择出口少的结点跳，那出口少的结点就会越来越少，这样跳成功的机会就更大一些。其实这种求解就是利用了贪心算法，它对整个求解过程的局部做最优调整，它只适用于求较优解或者部分解，而不能求最优解。这样的调整方法叫贪心策略，至于什么问题需要什么样的贪心策略是不确定的，具体问题具体分析。实验可以证明马遍历问题在运用到了上面的贪心策略之后求解速率有非常明显的提高，如果只要求出一个解甚至不用回溯就可以完成，因为在这个算法提出的时候世界上还没有计算机，这种方法完全可以用手工求出解来，其效率可想而知。

3. 马踏棋盘问题的 C 语言代码（8×8）:

```
int ways_out(int x,int y)
{                                                    //计算结点出口多少
    int i,count=0,tx,ty;
    if(x<0||y<0||x>=N||y>=N||s[x][y]>0)
    return -1;                                       // -1 表示该结点非法或者已经跳过了
    for(i=0;i<8;++i)
    {
        tx=x+dx[i];
        ty=y+dy[i];
        if(tx<0||ty<0||tx>=N||ty>=N)
            continue;
        if(s[tx][ty]==0)
            ++count;
    }
    return count;
}
void sortnode(h_node *hn,int n)                      //采用简单排序法，因为子结点数最多只有 8
{                                                    //按结点出口进行排序
    int i,j,t;
    h_node temp;
    for(i=0;i<n;++i)
    {
        for(t=i,j=i+1;j<n;++j)
            if(hn[j].waysout<hn[t].waysout)
            t=j;
        if(t>i)
        {
            temp=hn[i];
            hn[i]=hn[t];
            hn[t]=temp;
```

```
            }
        }
}
void dfs(int x,int y,int count)
{                                                    //修改后的搜索函数
    int i,tx,ty;
    h_node hn[8];
    if(count>N*N)
    {
        output_solution();
        return;
    }
    for(i=0;i<8;++i)                                 //求子结点和出口
    {
        hn[i].x=tx=x+dx[i];
        hn[i].y=ty=y+dy[i];
        hn[i].waysout=ways_out(tx,ty);
    }
    sortnode(hn,8);
    for(i=0;hn[i].waysout<0;++i);                    //不考虑无用结点
    for(;i<8;++i)
    {
        tx=hn[i].x;
        ty=hn[i].y;
        s[tx][ty]=count;
        dfs(tx,ty,count+1);
        s[tx][ty]=0;
    }
}
void main()
{                                                    //主函数
    int i,j,x,y;
    for(i=0;i<N;++i)                                 //初始化
        for(j=0;j<N;++j)
            s[i][j]=0;
    printf("Horse jump while N=%d\nInput the position to start:",N);
    scanf("%d%d",&x,&y);                             //输入初始位置
    while(x<0||y<0||x>=N||y>=N)
    {
```

```
        printf("Error! x,y should be in 0~%d",N-1);
        scanf("%d%d",&x,&y);
        }
        s[x][y]=1;
        dfs(x,y,2);                                    //开始搜索
}
```

该算法代码稍加补充即可运行。

5.8 最小生成树

假设已知一无向连通图 $G=(V,E)$，其加权函数为 $w:E\rightarrow R$，我们希望找到图 G 的最小生成树。后文所讨论的两种算法都运用了贪心方法，但在如何运用贪心法上却有所不同。

下列的算法 GENERNIC-MIT 正是采用了贪心算法，每步形成最小生成树的一条边。算法设置了集合 A，该集合一直是某最小生成树的子集。在每步决定是否把边(u,v)添加到集合 A 中，其添加条件是 $A\cup\{(u,v)\}$仍然是最小生成树的子集。我们称这样的边为 A 的安全边，因为可以安全地把它添加到 A 中而不会破坏上述条件。

GENERNIC-MIT(G,W)

1．$A\leftarrow\varnothing$

2．while A 没有形成一棵生成树

3．　　do　找出 A 的一条安全边(u,v);

4．　　$A\leftarrow A\cup\{(u,v)\}$;

5．return A

注意从第 1 行以后，A 显然满足最小生成树子集的条件。第 2～4 行的循环中保持着这一条件，当第 5 行中返回集合 A 时，A 就必然是一最小生成树。算法最棘手的部分自然是第 3 行的寻找安全边。必定存在一生成树，因为在执行第 3 行代码时，根据条件要求存在一生成树 T，使 $A\subseteq T$，且若存在边$(u,v)\in T$ 且$(u,v)\notin A$，则(u,v)是 A 的安全边。

定理 5.1　设图 $G(V,E)$是一无向连通图，且在 E 上定义了相应的实数值加权函数 w，设 A 是 E 的一个子集且包含于 G 的某个最小生成树中，割$(S,V-S)$是 G 的不妨碍 A 的任意割且边(u,v)是穿过割$(S,V-S)$的一条轻边，则边(u,v)对集合 A 是安全的。

下面介绍两个算法：Kruskal 算法和 Prim 算法

Kruskal 算法是直接基于上面中给出的一般最小生成树算法的基础之上的。该算法找出森林中连结任意两棵树的所有边中具有最小权值的边(u,v)作为安全边，并把它添加到正在生长的森林中。设 C_1 和 C_2 表示边(u,v)连结的两棵树。因为(u,v)必是连 C_1 和其他某棵树的一条轻边，所以由定理 5.1 可知(u,v)对 C_1 是安全边。Kruskal 算法同时也是一种贪心算法，因为算法每一步添加到森林中的边的权值都尽可能小。

Kruskal 算法的实现类似于计算连通支的算法。它使用了分离集合数据结构以保持数个互相分离的元素集合。每一集合包含当前森林中某个树的结点，操作 FIND-SET(u)返回包含 u

的集合的一个代表元素，因此我们可以通过 FIND-SET(v)来确定两结点 u 和 v 是否属于同一棵树，通过操作 UNION 来完成树与树的联结。

MST-KRUSKAL($\boldsymbol{G}$,w)
1. $\boldsymbol{A}\leftarrow\varnothing$
2. for 每个结点 v $\boldsymbol{V}$[$\boldsymbol{G}$]
3.　　do MAKE-SET(v)
4. 根据权 w 的非递减顺序对 $\boldsymbol{E}$ 的边进行排序
5. for 每条边$(u,v)\in\boldsymbol{E}$，按权的非递减次序
6.　　do if FIND-SET(u) ≠ FIND−SET(v)
7.　　then $\boldsymbol{A}\leftarrow\boldsymbol{A}\cup\{(u,v)\}$
8.　　UNION(u,v)
9.　return $\boldsymbol{A}$

Kruskal 算法在图 $\boldsymbol{G}=(\boldsymbol{V},\boldsymbol{E})$上的运行时间取决于分离集合这一数据结构如何实现。我们采用在分离集合中描述的按行结合和通路压缩的启发式方法来实现分离集合森林的结构，这是由于从渐近意义上来说，这是目前所知的最快的实现方法。初始化需占用时间 $O(\boldsymbol{V})$，第 4 行中对边进行排序需要的运行时间为 $O(\boldsymbol{E}\log\boldsymbol{E})$；对分离集的森林要进行 $O(\boldsymbol{E})$次操作，总共需要时间为 $O(\boldsymbol{E}\alpha(\boldsymbol{E},\boldsymbol{V}))$，其中$\alpha$函数为 Ackerman 函数的反函数。因为$\alpha(\boldsymbol{E},\boldsymbol{V})=O(\log\boldsymbol{E})$，所以 KruskaI 算法的全部运行时间为 $O(\boldsymbol{E}\log\boldsymbol{E})$。

正如 Kruskal 算法一样，Prim 算法也是第上面中讨论的一般最小生成树算法的特例。Prim 算法的执行非常类似于寻找图的最短通路的 Dijkstra 算法。Prim 算法的特点是集合 $\boldsymbol{A}$ 中的边总是只形成单棵树。因为每次添加到树中的边都是使树的权尽可能小的边，因此上述策略也是贪心的。

有效实现 Prim 算法的关键是设法较容易地选择一条新的边添加到由 $\boldsymbol{A}$ 的边所形成的树中，在下面的伪代码中，算法的输入是连通图 $\boldsymbol{G}$ 和将生成的最小生成树的根 r。在算法执行过程中，不在树中的所有结点都驻留于优先级基于 key 域的队列 $\boldsymbol{Q}$ 中。对每个结点 v，key[v]是连接 v 到树中结点的边所具有的最小权值；按常规，若不存在这样的边则 key[v]=∞。域π[v]说明树中 v 的“父母”。在算法执行中，GENERIC-MST 的集合 A 隐含地满足：

A={(v,π[v])|v∈V−{r}−Q}

当算法终止时，优先队列 Q 为空，因此 G 的最小生成树 A 满足：

A={(v,π[v])|v∈V−{r}}

MST−PRIM(G,w,r)
1. $Q\leftarrow V[G]$
2. for 每个 $u\in Q$
3.　　do key[u]←∞
4. key[r]←0
5. π[r]←NIL
6. while $Q\neq\varnothing$
7.　　do u←EXTRACT−MIN(Q)
8.　　for 每个 $v\in$Adj[u]
9.　　do if $v\in Q$ and $w(u,v)$<key[v]

10. then $\pi[v] \leftarrow u$

11. $\text{key}[v] \leftarrow w(u,v)$

Prim 算法的性能取决于我们如何实现优先队列 $\boldsymbol{Q}$。若用二叉堆来实现 $\boldsymbol{Q}$，我们可以使用过程 BUILD-HEAP 来实现第 1～4 行的初始化部分，其运行时间为 $O(\boldsymbol{V})$。循环需执行$|\boldsymbol{V}|$次，且由于每次 EXTRACT-MIN 操作需要 $O(\log \boldsymbol{V})$的时间，所以对 EXTRACT-MIN 的全部调用所占用的时间为 $O(\boldsymbol{V}\log \boldsymbol{V})$。第 8～11 行的 *for* 循环总共要执行 $O(\boldsymbol{E})$次，这是因为所有邻接表的长度和为 $2|\boldsymbol{E}|$。在 for 循环内部，第 9 行对队列 $\boldsymbol{Q}$ 的成员条件进行测试可以在常数时间内完成，这是由于为每个结点空出 1 位（bit）的空间来记录该结点是否在队列 $\boldsymbol{Q}$ 中，并在该结点被移出队列时随时对该位进行更新。第 11 行的赋值语句隐含一个对堆进行的 DECREASE-KEY 操作，该操作在堆上可用 $O(\log \boldsymbol{V})$的时间完成。因此，Prim 算法的整个运行时间为 $O(\boldsymbol{V}\log \boldsymbol{V}+\boldsymbol{E}\log \boldsymbol{V})=O(\boldsymbol{E}\log \boldsymbol{V})$，从渐近意义上来说，它和实现 Kruskal 算法的运行时间相同。

通过使用 Fibonacci 堆，Prim 算法的渐近意义上的运行时间可得到改进。在 Fibonacci 堆中我们已经说明，如果$|\boldsymbol{V}|$个元素被组织成 Fibonacci 堆，可以在 $O(\log \boldsymbol{V})$的平摊时间内完成 EXTRACT-MIN 操作，在 $O(1)$的平摊时间里完成 DECREASE-KEY 操作（为实现第 11 行的代码），因此，若我们用 Fibonacci 堆来实现优先队列 $\boldsymbol{Q}$，Prim 算法的运行时间可以改进为 $O(\boldsymbol{E}+\boldsymbol{V}\log \boldsymbol{V})$。

```
// 图的存储结构以数组邻接矩阵表示,用普里姆(Prim)算法构造图的最小生成树。
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define   TRUE 1
#define   FALSE 0
#define   NULL 0
#define   OVERFLOW -2
#define   OK       1
#define   ERROR 0
typedef int Status;
typedef int VRType;
typedef char InfoType;                          //弧相关的信息
typedef char VertexType[10];                    //顶点的名称为字符串
#define INFINITY              32767             //INT_MAX   最大整数
#define MAX_VERTEX_NUM    20                    //最大顶点数
typedef   enum{DG,DN,AG,AN} GraphKind;          //有向图,有向网,无向图,无向网
typedef struct ArcCell{
    VRType      adj;                            // VRType 是顶点的关系情况,对无权图用1或0表示
                                                   有关系否,对带权图(网),
                                                // 则填权值。
    InfoType   *info;                           // 指向该弧相关信息的指针。
} ArcCell, AdjMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM];
```

```
typedef struct{
    VertexType  vexs[MAX_VERTEX_NUM];    // 顶点数据元素
AdjMatrix    arcs;                       // 二维数组作邻接矩阵
int  vexnum,arcnum;                      // 图的当前顶点数和弧数
    GraphKind    kind;                   // 图的种类标志
} MGraph;

Status CreateGraph(MGraph &G,GraphKind kd){    // 采用数组邻接矩阵表示法，构造图 G
    Status  CreateDG(MGraph &G);
    Status  CreateDN(MGraph &G);
    Status  CreateAG(MGraph &G);
    Status  CreateAN(MGraph &G);
    G.kind=kd;
    switch(G.kind){
    case DG: return CreateDG(G);                 //构造有向图 G
    case DN: return CreateDN(G);                 //构造有向网 G
    case AG: return CreateAG(G);                 //构造无向图 G
    case AN: return CreateAN(G);                 //构造无向网 G
        default: return  ERROR;
  }
}

Status  CreateDG(MGraph &G){
 return OK;
}
Status  CreateDN(MGraph &G){
 return OK;
}
Status  CreateAG(MGraph &G){
 return OK;
}
Status  CreateAN(MGraph &G){//构造无向网 G
    int i,j,k;
char v[3],w[3] ,vwinfo[10]={"  "}; //边有关信息置空
char v[10][3]={"v1","v1","v2","v2","v5","v5","v6","v6","v4","v4"};
char w[10][3]={"v2","v3","v5","v3","v6","v3","v4","v3","v1","v3"};
int  q[10]={    6,  1,  3,  5,  6,  6,  2,  4,  5,  5  };
char vwinfo[10]={"  "};
printf("输入要构造的网的顶点数和弧数：\n");
```

```
scanf("%d,%d",&G.vexnum,&G.arcnum);
G.vexnum=6;     G.arcnum=10;
      printf("依次输入网的顶点名称 v1   v2   …等等：\n");
for (i=0; i<G.vexnum; i++) scanf("%s",G.vexs[i]);//构造顶点数据
      strcpy(G.vexs[0],"v1");       strcpy(G.vexs[1],"v2");       strcpy(G.vexs[2],"v3");
strcpy(G.vexs[3],"v4");       strcpy(G.vexs[4],"v5");       strcpy(G.vexs[5],"v6");
      for (i=0; i<G.vexnum; i++)
        for (j=0; j<G.vexnum; j++) { G.arcs[i][j].adj=INFINITY;   G.arcs[i][j].info=NULL;}//初始化邻接矩阵
      printf("按照：  顶点名 1   顶点名 2   权值   输入数据:\n");
         for (k=0; k<G.arcnum; k++){
scanf("%s %s   %d",v,w,&q);
        for(i=0;i<G.vexnum; i++) if(strcmp(G.vexs[i],v[k])==0) break;
//查找出 v 在 vexs[]中的位置 i
            if(i==G.vexnum) return ERROR;
        for(j=0;j<G.vexnum; j++) if(strcmp(G.vexs[j],w[k])==0) break;
//查找出 v 在 vexs[]中的位置 j
            if(j==G.vexnum) return ERROR;
      G.arcs[i][j].adj=q[k];   //邻接矩阵对应位置置权值
      G.arcs[j][i].adj=q[k];   //邻接矩阵对称位置置权值
G.arcs[i][j].info=(char *)malloc(10); strcpy(G.arcs[i][j].info,vwinfo);
//置入边有关信息
 }
 return OK;
}
void PrintMGraph(MGraph &G){
 int i,j;
 switch(G.kind){
    case DG:
          for (i=0; i<G.vexnum; i++) {
            for (j=0; j<G.vexnum; j++){
                printf("       %d | ",G.arcs[i][j].adj);
                if(G.arcs[i][j].info==NULL)
                   printf("NULL");
                else
                   printf("%s 路",G.arcs[i][j].info);
            }
            printf("\n");
      }
      break;
```

```
        case DN:
            for (i=0; i<G.vexnum; i++) {
              for (j=0; j<G.vexnum; j++){
                if(G.arcs[i][j].adj!=0) printf("       %d | ",G.arcs[i][j].adj);
                   else printf("    ∞    | ");
              }
              printf("\n");
        }
        break;
  case AG:
            for (i=0; i<G.vexnum; i++) {
              for (j=0; j<G.vexnum; j++){
                   printf("    %d | ",G.arcs[i][j].adj);
         }
              printf("\n");
         }
         break;
           case AN:
              for (i=0; i<G.vexnum; i++) {
                for (j=0; j<G.vexnum; j++){
        if(G.arcs[i][j].adj<INFINITY) printf("    %d | ",G.arcs[i][j].adj);
                    else {printf("    ∞| ");    }
        }
                printf("\n");
           }
      }
  return;
}

Status MiniSpanTree_PRIM(MGraph G,VertexType u){
        int i,j,k,r,min;
  struct{
        VertexType adjvex;
        VRType         lowcost;
  }closedge[MAX_VERTEX_NUM];                          //定义辅助数组
    k=LocateVex(G,u);
  for(i=0;i<G.vexnum; i++) if(strcmp(G.vexs[i],u)==0) break;
//查找出 v 在 vexs[]中的位置 i
        if(i==G.vexnum) return ERROR;
```

```
        k=i;
    for(j=0; j<G.vexnum; ++j)                          //辅助数组初始化
      if(j!=k) {strcpy(closedge[j].adjvex,u);   closedge[j].lowcost=G.arcs[k][j].adj;}
    closedge[k].lowcost=0;                             //初始,U={0,即 v1}
    for(i=1; i<G.vexnum; ++i){
            k=mininum(closedge);                       //求权值最小的顶点
            min=INFINITY;
              for(r=0; r<G.vexnum; r++){
              if(closedge[r].lowcost > 0 && closedge[r].lowcost<min){
                  k=r; min=closedge[r].lowcost;}
          }
          printf("k=%d    %s  －－－> %s\n",k,closedge[k].adjvex,G.vexs[k]);
  //输出边
          closedge[k].lowcost=0;                       //第顶点并入 U 集
          for(int j=0; j<G.vexnum; ++j)
              if(G.arcs[k][j].adj < closedge[j].lowcost){
                                                       //新顶点并入 U 集后,重新选择最小边
                 strcpy(closedge[j].adjvex,G.vexs[k]);
              closedge[j].lowcost=G.arcs[k][j].adj;
          }
      }
  return OK;
  }
  void main(){
          MGraph ANN;
          printf("构造无向网\n");
          CreateGraph(ANN,AN);                         // 采用数组邻接矩阵表示法，构造有向网 AGG
          PrintMGraph(ANN);
          MiniSpanTree_PRIM(ANN,"v1");
          return;
  }
```

5.9 哈夫曼编码

哈夫曼编码是广泛地用于数据文件压缩的十分有效的编码方法。其压缩率通常在 20%～90%之间。哈夫曼编码算法用字符在文件中出现的频率表来建立一个用 0、1 串表示各字符的最优表示方式。

给出现频率高的字符较短的编码，出现频率较低的字符以较长的编码，可以大大缩短总码长。

例如一个包含 100 000 个字符的文件，各字符出现频率不同，如表 5.1 所示。定长变码需要 300 000 位，而按表中变长编码方案，文件的总码长为：

(45×1+13×3+12×3+16×3+9×4+5×4) ×1 000=224 000。

比用定长码方案总码长较少约 45%。

表 5.1　　定长码与变长码

字　　符	a	B	c	d	e	f
频率(千次)	45	13	12	16	9	5
定长码	000	001	010	011	100	101
变长码	0	101	100	111	1101	1100

对每一个字符规定一个 0、1 串作为其代码，并要求任一字符的代码都不是其他字符代码的前缀。这种编码称为前缀码。

编码的前缀性质可以使译码方法非常简单。

表示最优前缀码的二叉树总是一棵完全二叉树，即树中任一结点都有 2 个子结点。

平均码长定义为：

$$B(T)=\sum_{c\in C} f(c)\,d_T(c)$$

使平均码长达到最小的前缀码编码方案称为给定编码字符集 C 的最优前缀码。哈夫曼提出构造最优前缀码的贪心算法，由此产生的编码方案称为哈夫曼编码。哈夫曼算法以自底向上的方式构造表示最优前缀码的二叉树 T。算法以$|C|$个叶结点开始，执行$|C|-1$ 次的“合并”运算后产生最终所要求的树 T。

```
HUFFMAN(c)
1.  n←|c|
2.  Q←C
3.  for i←to n−1
4.    do allocate a new code z
5.    left[z]←x←EXTRACT-MIN(Q)
6.    right[z]←y←EXTRACT-MIN(Q)
7.    f[z]=f[x]+f[y]
8.    INSERT(Q,z)
9.  return EXTRACT-MIN(Q)
```

时间分析，我们假设 Q 是作为最小二叉堆实现的。对包含个字符的集合 C，第二行中对 Q 的初始化可用在前面建堆章节中介绍所用的时间 $O(n)$内完成。第 3～8 行中的 for 循环执行了 $n-1$ 次，又因每一次堆操作需要 $O(\log n)$时间，故整个循环需要 $O(n\log n)$时间。这样，作用于 n 个字符集合的 HUFFMAN 的总的运行时间为 $O(n\log n)$。

```
/* Huffman 编码问题的设计和实现  */
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
```

```
#define MAXLEN      100
#define MAXVALUE    10000
/* 结点结构定义 */
typedef   struct
{ int weight;     /*权值*/
   int flag;       /*标记*/
   int parent;     /*指向父结点的指针*/
   int lchild;
   int rchild;
}HuffNode;
/*Huffman 编码结构*/
typedef struct
{   char bit[MAXLEN];
    int len;
    int weight;
}Code;
/* HuffTree 初始化 */
void HuffmanInit(int   weight[], int n, HuffNode hufftree[])
{   int i;
  /* huffman 结构初始化，n 个叶结点的二叉树有 2n−1 个结点   */
  for(i = 0; i < 2 * n − 1; i++){
       hufftree[i].weight = (i < n) ? weight[i] : 0;
       hufftree[i].parent = −1;   /*根，无父结点 */
       hufftree[i].flag = 0;
       hufftree[i].lchild = −1;   /* i 不可能是某结点的左子树或右子树*/
       hufftree[i].rchild = −1;
    }
}
/*建立权值为 weight[0..n−1]的 n 个结点的 HuffTree */
void Huffman(int   weight[], int n, HuffNode hufftree[])
{   int i,j,m1,m2,x1,x2;
    HuffmanInit(weight,n,hufftree); /*初始化 */
    /* 构造 n − 1 个非叶结点 */
    for(i = 0; i < n − 1; i++){
       m1 = m2 = MAXVALUE;   /* m1 <= m2   */
       x1 = x2 = 0;
       for(j = 0; j < n + i; j++){/*在森林中找两个权值最小的结点*/
          if(hufftree[j].flag == 0){ /* 该结点未加入到 huffman 树中 */
             if(hufftree[j].weight < m1){
```

```
                    m2 = m1;
                    x2 = x1;
                    m1 = hufftree[j].weight;
                    x1 = j;
                }else if(hufftree[j].weight < m2){
                    m2 = hufftree[j].weight;
                    x2 = j;
                }
            }
        }
        hufftree[x1].parent = n + i;
        hufftree[x2].parent = n + i;
        hufftree[x1].flag = 1;
        hufftree[x2].flag = 1;
    hufftree[n + i].weight = hufftree[x1].weight + hufftree[x2].weight;
        hufftree[n + i].lchild = x1;
        hufftree[n + i].rchild = x2;
    }
 }
/* huffman 编码函数 */
void    HuffmanCode(HuffNode hufftree[],int n,Code huffcode[])
{   Code    cd;
    int i,j,child,parent;
    for(i = 0; i < n; i++){    /* 求第 i 个结点的 Huffman 编码 */
        cd.len = 0;
        cd.weight = hufftree[i].weight;
        child = i;
        parent = hufftree[i].parent;    /* 回溯 */
        while(parent    != -1){
          cd.bit[cd.len++] = (hufftree[parent].lchild == child) ?'0' : '1';
            child = parent;
            parent = hufftree[child].parent;
        }
        for(j = 0; j < cd.len; j++)
            huffcode[i].bit[j] = cd.bit[cd.len - 1 - j];
        huffcode[i].bit[cd.len] = '\0';
        huffcode[i].len = cd.len;
        huffcode[i].weight = cd.weight;
    }
```

```
}
/*  打印 huffman 编码  */
void PrintCode(Code  c[],int n)
{ int  i;
   printf("OutPut code :\n");
   for(i = 0; i < n; i++)
       printf("weight = %d  code  %s\n",c[i].weight,c[i].bit);
}
/* 测试程序 */
void main(void)
{ int w[] = {3,1,4,8,2,5,7};
   HuffNode  huff[100];
   Code hcode[10];
   Huffman(w,7,huff);
   HuffmanCode(huff,7,hcode);
   PrintCode(hcode,7);
   getch();
}
```

习　　题

1．一个数列由 n 个正整数组成。对该数列进行一次操作：去除其中两项 a、b，添加一项 $a\times b+1$。经 $n-1$ 次操作后该数列剩一个数 a，试求 a 的最大值。

2．随机产生一个 n 位高精度正整数,去掉其中 $k(k<n)$个数字后按原左右次序将组成一个新的正整数。对给定的 n,k 寻找一种方案,使得剩下的数字组成的新数最小。

3．利用贪心 kruscal 算法和 prim 算法求出下图的最小生成树（要求画出生成树，过程可以省略，如图 5.2 所示）。

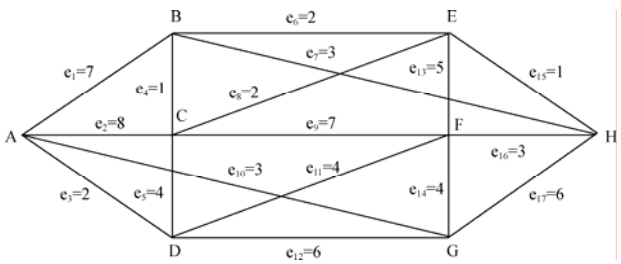


图 5.2

4．一个有向图 $\boldsymbol{G}$，它的每条边都有一个非负的权值 $c[i,j]$，“路径长度”就是所经过的所有边的权值之和。对于源点需要找出从源点出发到达其他所有结点的最短路径，试用贪心算法解此问题。

第 6 章　动 态 规 划

6.1　一般方法与求解步骤

动态规划是运筹学的一个分支，是求解决策过程最优化的数学方法。20 世纪 50 年代美国数学家贝尔曼（Rechard Bellman）等人在研究多阶段决策过程的优化问题时，提出了著名的最优性原理，把多阶段决策过程转化为一系列单阶段问题逐个求解，创立了解决多阶段过程优化问题的新方法——动态规划。

动态规划问世以来，在经济管理、生产调度、工程技术和最优控制等方面得到了广泛的应用。例如最短路线、库存管理、资源分配、装载等问题，用动态规划方法比用其他方法求解更为方便。

6.1.1　一般方法

多阶段决策问题，是指这样的一类特殊的活动过程，问题可以分解成若干相互联系的阶段，在每一个阶段都要做出决策，形成一个决策序列，该决策序列也称为一个策略。对于每一个决策序列，可以在满足问题的约束条件下用一个数值函数（即目标函数）衡量该策略的优劣。多阶段决策问题的最优化求解目标是获取导致问题最优值的最优决策序列（最优策略），即得到最优解。

【例 6.1】 已知 6 种物品和一个可载重量为 60 的背包，物品 $i(i=1,2,\cdots,6)$的重量为 w_i 分别为(15,17,20,12,9,14)，产生的效益为 p_i 分别为(32,37,46,26,21,30)。在装包时每一件物品可以装入，也可以不装，但不可拆开装。确定如何装包，使所得装包总效益最大。

这就是一个多阶段决策问题，装每一件物品就是一个阶段，每一个阶段都要有一个决策：这一件物品装包还是不装。这一装包问题的约束条件为 $\sum_{i=1}^{6} x_i w_i \leqslant 60$，目标函数为 $\max \sum_{i=1}^{6} x_i p_i, x_i \in \{0,1\}$。

对于这 6 个阶段的问题，如果每一个阶段都面临 2 个选择，则共存在 2^6 个决策序列。应用贪心算法，按单位重量的效益从大到小装包，得第 1 件与第 6 件物品不装，依次装第 5、3、

2、4 件物品，这就是一个决策序列，或简写为整数 x_i 的序列(0,1,1,1,1,0)，该策略所得总效益为 130。第 1 件与第 4 件物品不装，第 2、3、5、6 件物品装包，或简写为整数 x_i 的序列(0,1,1,0,1,1)，这一决策序列的总载重量为 60，满足约束条件，使目标函数即装包总效益达最大值 134，即最优值为 134；因而决策序列(0,1,1,0,1,1)为最优决策序列，即最优解，这是应用动态规划求解的目标。

在求解多阶段决策问题中，各个阶段的决策依赖于当时的状态并影响以后的发展，即引起状态的转移。一个决策序列是随着变化的状态而产生的。应用动态规划设计使多阶段决策过程达到最优（成本最省，效益最高，路径最短等），依据动态规划最优性原理：“作为整个过程的最优策略具有这样的性质，无论过去的状态和决策如何，对前面的决策所形成的状态而言，余下的诸决策必须构成最优策略”。也就是说，最优决策序列中的任何子序列都是最优的。

“最优性原理”用数学语言描述：假设为了解决某一多阶段决策过程的优化问题，需要依次作出 n 个决策 $D_1,D_2,\cdots,D_n$，如若这个决策序列是最优的，对于任何一个整数 k，$1<k<n$，不论前面 k 个决策是怎样的，以后的最优决策只取决于由前面决策所确定的当前状态，即以后的决策序列 $D_{k+1},D_{k+2},\cdots,D_n$ 也是最优的。

最优性原理体现为问题的最优子结构特性。当一个问题的最优解中包含了子问题的最优解时，则称该问题具有最优子结构特性。最优子结构特性使得在从较小问题的解构造较大问题的解时，只需考虑子问题的最优解，从而大大减少了求解问题的计算量。最优子结构特性是动态规划求解问题的必要条件。

例如，在下例中求得在数字串 847313926 中插入 5 个乘号，使乘积最大的最优解为：

8*4*731*3*92*6=38 737 152

该最优解包含了在 84731 中插入 2 个乘号使乘积最大为 8*4*731；在 7313 中插入 1 个乘号使乘积最大为 731*3；在 3926 中插入 2 个乘号使乘积最大为 3*92*6 等子问题的最优解，这就是最优子结构特性。

最优性原理是动态规划的基础。任何一个问题，如果失去了这个最优性原理的支持，就不可能用动态规划设计求解。能采用动态规划求解的问题都需要满足以下条件：

（1）问题中的状态必须满足最优性原理；

（2）问题中的状态必须满足无后效性。

所谓无后效性是指：“下一时刻的状态只与当前状态有关，而和当前状态之前的状态无关，当前状态是对以往决策的总结”。

6.1.2 动态规划求解步骤

动态规划求解最优化问题，通常按以下几个步骤进行。

（1）把所求最优化问题分成若干个阶段，找出最优解的性质，并刻划其结构特性。

最优子结构特性是动态规划求解问题的必要条件，只有满足最优子结构特性的多阶段决策问题才能应用动态规划设计求解。

（2）将问题发展到各个阶段时所处不同的状态表示出来，确定各个阶段状态之间的递推关系，并确定初始（边界）条件。

通过设置相应的数组表示各个阶段的最优值，分析归纳出各个阶段状态之间的转移关系，

是应用动态规划设计求解的关键。

（3）应用递推求解最优值。

递推计算最优值是动态规划算法的实施过程。具体应用顺推还是逆推，与所设置的表示各个阶段最优值的数组密切相关。

（4）根据计算最优值时所得到的信息，构造最优解。

构造最优解就是具体求出最优决策序列。通常在计算最优值时，根据问题具体实际记录更多的信息，根据所记录的信息构造出问题的最优解。

以上步骤前 3 个是动态规划设计求解最优化问题的基本步骤。当只需求解最优值时，第 4 个步骤可以省略。若需求出问题的最优解，则必须执行第 4 个步骤。

6.2　装载问题

【例 6.2】 有 n 个集装箱要装上两艘载重量分别为 c_1,c_2 的轮船，其中集装箱 i 的重量为 w_i，且 $\sum_{i=1}^{n} w_i \leqslant c_1 + c_2$，$c_1, c_2, w_i \in N^+$（不考虑集装箱的体积）。

试求解一个合理的装载方案，把所有 n 个集装箱装上这两艘船。

1. 算法设计

（1）问题求解策略

试采用以下的装载策略：

① 首先将第一艘船尽可能装满；

② 将剩余的集装箱装上第二艘船。

设装载量为 c_1 的船最多可装 $\max c_1$，如果满足不等式

$$\sum_{i=1}^{n} w_i - c_2 \leqslant \max c_1 \leqslant c_1 \ \Leftrightarrow\ \max c_1 \leqslant c_1,\ \sum_{i=1}^{n} w_i - \max c_1 \leqslant c_2$$

则装载问题有解。

装载问题不一定总有解。例如，当 n=3,c_1=c_2=50,w={15,40,40}，显然无法把这三个集装箱装上这两艘轮船。当问题无解时，作无解说明。

（2）动态规划设计

为了求取 $\max c_1$，应用动态规划设计。

目标函数：$\max \sum_{i=1}^{n} x_i w_i$

约束条件：$\sum_{i=1}^{n} x_i w_i \leqslant c_1,\quad (x_i \in \{0,1\}, c_1, w_i \in N^+, i = 1, 2, \cdots, n)$

按装载每一个集装箱为一个阶段，共分为 n 个阶段。

① 建立递推关系

设 $m(i,j)$为船 c_1 还有载重量为 j，可取集装箱编号范围为：i,i+1,…,n 的最大装载重量值。

则

当 $0 \leqslant j < w(i)$时，物品 i 不可能装入。$m(i,j)$与 $m(i+1,j)$相同。

而当 $j \geqslant w(i)$时，有两种选择：

不装入物品 i，这时最大重量值为 $m(i+1,j)$；

装入物品 i，这时已增加重量 $w(i)$，剩余载重容量为 $j-w(i)$，可以选择集装箱 $i+1,\cdots,n$ 来装，最大载重量值为 $m(i+1,j-w(i))+w(i)$。我们期望的最大载重量值是两者中的最大者。于是有递推关系

$$m(i,j)=\begin{cases} m(i+1,j) & 0 \leqslant j < w(i) \\ \max(m(i+1,j), m(i+1,j-w(i))+w(i)) & j \geqslant w(i) \end{cases}$$

以上 j、c_1 与 $w(i)$均为正整数，$i=1,2,\cdots,n$,

所求最优值为 $m(1,c_1)$。

② 递推计算最优值

```
for(j=0;j<w[n];j++) m[n][j]=0;
for(j=w[n];j<=c1;j++) m[n][j]=w[n];                    /*  首先计算 m(n,j)  */
for(i=n-1;i>=1;i--)                                    /*  逆推计算 m(i,j)  */
for(j=0;j<=c1;j++)
        if(j>=w[i]   && m[i+1][j]<m[i+1][j-w[i]]+w[i])
              m[i][j]=m[i+1][j-w[i]]+w[i];
        else
              m[i][j]=m[i+1][j];
    printf("%d",m[1][c1]);
```

（3）递推过程分析

例如，$n=5$,5 个集装箱的重量分别为：$w(1)=2$,$w(2)=5$,$w(3)=13$,$w(4)=8$,$w(5)=4$，两船的载重量分别为：$c_1=16$,$c_2=18$；

展示以上所述递推过程如下：

① 首先赋初值

$m(5,0)=\cdots=m(5,3)=0$　　(不装 $w(5)$)

$m(5,4)=m(5,5)=\cdots=m(5,16)=4$　　(装 $w(5)$)

② 递推过程展示

按递推关系分别列出 $n=4,3,2,1$ 时 m 数组的取值如下：

表 6.1　　推出 **n=4** 时 **m** 数组值

j	0～3	4～7	8～11	12～16
$m(4,j)$	0	4	8	12

表 6.2　　推出 **n=3** 时 **m** 数组值

J	0～3	4～7	8～11	12	13～16
$m(3,j)$	0	4	8	12	13

表 6.3　　推出 **n=2** 时 **m** 数组值

J	0～3	4	5～7	8	9～11	12	13～16
$m(2, j)$	0	4	5	8	9	12	13

表 6.4　　推出 i=1 时 m 数组值

j	0～1	2～3	4	5	6	7	8
$m(1, j)$	0	2	4	5	6	7	8
j	9	10	11	12	13	14	15～16
$m(1, j)$	9	10	11	12	13	14	15

最后所得目标值即船 c_1 的最大装载量为：$m(1,16)=15$。注意到 5 个集装箱的总重量为 32，满足 $32-18 \leqslant 15 \leqslant 16$，此装载问题有解。

（4）构造最优解

构造最优解即给出所得最优值时的装载方案。

if(m[i][cw]>m[i+1][cw])　（其中 cw 为当前的装载量，i=1,2, n−1）

　　装载 w[i];

else　不装载 w[i];

if(所载集装箱重量≠m(1,c_1))　装载 w[n].

以上述举例说明：

由 $m(1,16)>m(2,16)$，装 $w(1)=2$；

　$m(2,14)=m(3,14)$，不装 $w(2)$；

　$m(3,14)>m(4,14)$，装 $w(3)=13$；

　$w(1)+w(3)=15=m(1,16)$，不装 $w(5)$。

于是得装载方案：

　　c1:　2　13　(15)

　　c2:　5　8　4　(17)

（5）算法复杂度分析

以上动态规划算法的时间复杂度为 $O(nc_1)$，空间复杂度也为 $O(nc_1)$。通常 $c_1>n$，因而上述算法的时间与空间复杂度均高于 $O(n^2)$。

2. 装载问题的 C 程序实现

```
/*  装载问题 */
#include <stdio.h>
#define N 100
void main()
{int n,c1,c2,i,j,s,t,cw,sw,w[N],m[N][N];
 printf(" input c1,c2: "); scanf("%d,%d",&c1,&c2);
 printf(" input n: "); scanf("%d",&n);
 s=0;
 for(i=1;i<=n;i++)        /*  输入 n 个集装箱重量整数 */
    {scanf("%d",&w[i]);
```

```
        s+=w[i];
    }
if(s>c1+c2)
    return     /* 确保 n 个集装箱重量之和不大于 c1+c2 */
printf("集装箱重量：");
for(i=1;i<=n;i++)
    printf("    %d",w[i]);
printf("\n n=%d,s=%d ",n,s);
printf("\n c1=%d, c2=%d \n",c1,c2);
for(j=0;j<w[n];j++) m[n][j]=0;
for(j=w[n];j<=c1;j++) m[n][j]=w[n];                    /*   首先计算 m(n,j)   */
for(i=n-1;i>=1;i--)                                    /*   逆推计算 m(i,j)   */
    for(j=0;j<=c1;j++)
        if(j>=w[i]   && m[i+1][j]<m[i+1][j-w[i]]+w[i])
                m[i][j]=m[i+1][j-w[i]]+w[i];
        else
                m[i][j]=m[i+1][j];
printf("maxc1=%d \n",m[1][c1]);                        /* 得最优值 m(1,c1) */
if(m[1][c1]<=c1 && m[1][c1]>=s-c2)                     /* 判断是否有解 */
   {printf("C1:");
    cw=m[1][c1];
    for(sw=0,i=1;i<=n-1;i++)                           /*   构造最优解，输出船 1 的装载 */
       if(m[i][cw]>m[i+1][cw])
          {cw-=w[i];sw+=w[i];
           printf(" %3d",w[i]);
           w[i]=0;                                     /* w(i)装载后赋 0,为装船 2 作准备 */
          }
    if(m[1][c1]-sw==w[n])
       {printf(" %3d",w[n]);
        sw+=w[n];
        w[n]=0;
       }
    printf("    (%d)\n",sw);
    printf("C2:");
    for(sw=0,i=1;i<=n;i++)                             /*   输出船 2 的装载 */
       if(w[i]>0)
          {sw+=w[i];
           printf(" %3d",w[i]);
          }
```

```
        printf("    (%d)\n",sw);
      }
  else     printf("此装载问题无解！");                /*   输出无解信息 */
}
```

3. 程序运行与说明

运行程序，输入：

```
input c1,c2:120,126
input n:15
```

集装箱重量：26　19　24　13　10　20　15　12　6　5　22　7　17　27　20

```
    n=15,s=243
    c1=120, c2=126
    maxc1=120
    C1:   15   12   22    7   17   27   20   (120)
    C2:   26   19   24   13   10   20    6    5   (123)
```

注意：上述所求解的装载问题中，要求各个集装箱的重量与两船的载重量 c1,c2 均为正整数。

6.3 插入乘号问题

在指定数字串中插入运算符号问题，包括插入若干个乘号求积的最大最小，或插入若干个加号求和的最大最小，都是比较新颖且有一定难度的最优化问题。这里通过限制数字串的长度来降低设计求解的难度。

【例 6.3】 在一个由 n 个数字组成的数字串中插入 r 个乘号($1 \leqslant r < n < 10$)，将它分成 $r+1$ 个整数，找出一种乘号的插入方法，使得这 $r+1$ 个整数的乘积最大。

例如，对给定的数串 847313926，如何插入 $r=5$ 个乘号，使其乘积最大?

1. 动态规划设计

（1）算法设计

① 建立递推关系

设 $f(i,k)$表示在前 i 位数中插入 k 个乘号所得乘积的最大值，$a(i, j)$表示从第 i 个数字到第 j 个数字所组成的 $j-i+1(i \leqslant j)$位整数值。

为了寻求递推关系，先看一个实例：对给定的数串 847313926，如何插入 $r=5$ 个乘号，使其乘积最大? 我们的目标是为了求取最优值 $f(9,5)$。

设前 8 个数字中已插入 4 个乘号，则最大乘积为 $f(8,4)*6$;

设前 7 个数字中已插入 4 个乘号，则最大乘积为 $f(7,4)*26$;

设前 6 个数字中已插入 4 个乘号，则最大乘积为 $f(6,4)*926$;

设前 5 个数字中已插入 4 个乘号，则最大乘积为 $f(5,4)*3926$。

比较以上 4 个数值的最大值即为 $f(9,5)$。

依此类推，为了求 $f(8,4)$：

设前 7 个数字中已插入 3 个乘号，则最大乘积为 $f(7,3)*2$；

设前 6 个数字中已插入 3 个乘号，则最大乘积为 $f(6,3)*92$；

设前 5 个数字中已插入 3 个乘号，则最大乘积为 $f(5,3)*392$；

设前 4 个数字中已插入 3 个乘号，则最大乘积为 $f(4,3)*1392$。

比较以上 4 个数值的最大值即为 $f(8,4)$。

一般地，为了求取 $f(i,k)$，考察数字串的前 i 个数字，设前 j（$k \leqslant j<i$）个数字中已插入 $k-1$ 个乘号的基础上，在第 j 个数字后插入第 k 个乘号，显然此时的最大乘积为 $f(j,k-1)*a(j+1,i)$。于是可以得递推关系式：

$$f(i,k)=max(f(j,k-1)*a(j+1,i)) \qquad (k \leqslant j<i)$$

前 j 个数字没有插入乘号时的值显然为前 j 个数字组成的整数，因而得边界值为：

$$f(j,0)=a(1,j)\ (1 \leqslant j \leqslant i)$$

② 递推计算最优值

```
for(d=0,j=1;j<=n;j++)
    {d=d*10+b[j-1]-48;                  /* 把 b 数组的一个字符转化为数值 */
     a[1][j]=d;
     f[j][0]=a[1][j];                   /* f(j,0)赋初始值  */
    }
 for(k=1;k<=r;k++)
 for(i=k+1;i<=n;i++)
 {for(amax=0,j=k;j<i;j++)
   {for(d=0,u=j+1;u<=i;u++)
        d=d*10+b[u-1]-48;
    a[j+1][i]=d;
    if(amax<f[j][k-1]*a[j+1][i])        /* 求 f(j,k-1)*a(j+1,i)最大值 */
        amax=f[j][k-1]*a[j+1][i];
    }
  f[i][k]=amax;                         /* 赋值给 f(i,k) */
 }
printf("最优值为%ld",f[n,r]);
```

③ 递推计算最优值的改进

事实上，数组 $a(i,j)$可简化一个简单变量。同时，求最大值的 amax 也可以省略。以上递推可简化为：

```
for(d=0,j=1;j<=n;j++)
    {d=d*10+b[j-1]-48;                  /* 把 b 数组的一个字符转化为数值 */
     f[j][0]=d;                         /*  计算初始值 f[j][0]  */
    }
```

```
 for(k=1;k<=r;k++)
 for(i=k+1;i<=n;i++)
 for(j=k;j<i;j++)
   {for(d=0,u=j+1;u<=i;u++)
        d=d*10+b[u-1]-48;              /* 计算 d 即为 a(j+1,i) */
    if(f[i][k]<f[j][k-1]*d)            /* 递推求取 f[i][k] */
        f[i][k]=f[j][k-1]*d;
    }
printf("最优值为%ld"，f[n,r]);
```

④ 构造最优解

为了能打印相应的插入乘号的乘积式，设置标注位置的数组 $t(k)$与 $c(i,k)$，其中 $c(i,k)$为相应的 $f(i,k)$的第 k 个乘号的位置，而 $t(k)$标明第 k 个乘号“*”的位置，例如，$t(2)=3$，表明第 2 个“*”号在在第 3 个数字后面。

当给数组元素赋值 $f(i,k)=f(j,k-1)*d$ 时，作相应赋值 $c(i,k)=j$，表明 $f(i,k)$的第 k 个乘号的位置是 j。在求得 $f(n,r)$的第 r 个乘号位置 $t(r)=c(n,r)=j$ 的基础上，其他 $t(k)(1\leqslant k\leqslant r-1)$可应用下式逆推产生

$$t(k)=c(t(k+1),k)$$

根据 t 数组的值，可直接按字符形式打印出所求得的插入乘号的乘积式。

（2）插入乘号问题 C 程序实现

```
/* 在一个数字串中插入若干个*号，使积最大 */
#include <stdio.h>
void main()
{char b[11];
 int n,i,j,k,u,r,t[10],c[10][10];
 long f[10][10],d;
 printf("请输入整数：");
 scanf("%s",b);
 for(n=0,i=0;b[i]!='\0';i++) n++;
 printf("请输入插入的乘号个数 r：");
 scanf("%d",&r);
 printf("在整数%s 中插入%d 个乘号，使乘积最大：\n",b,r);
 for(d=0,j=1;j<=n;j++)
   {d=d*10+b[j-1]-48;              /* 把 b 数组的一个字符转化为数值 */
    f[j][0]=d;                     /* f[j][0]赋初始值  */
   }
 for(k=1;k<=r;k++)
 for(i=k+1;i<=n;i++)
 for(j=k;j<i;j++)
   {for(d=0,u=j+1;u<=i;u++)
```

```
            d=d*10+b[u-1]-48;
        if(f[i][k]<f[j][k-1]*d)          /* 递推求取 f[i][k] */
            {f[i][k]=f[j][k-1]*d;
             c[i][k]=j;
            }
        }
    t[r]=c[n][r];
    for(k=r-1;k>=1;k--)
      t[k]=c[t[k+1]][k];                 /* 逆推出第 k 个*号的位置 t[k] */
    t[0]=0;t[r+1]=n;
    for(k=1;k<=r+1;k++)
      {for(u=t[k-1]+1;u<=t[k];u++)
          printf("%c",b[u-1]);           /* 输出最优解 */
       if(k<r+1)
          printf("*");
       }
    printf("=%ld\n ",f[n][r]);           /* 输出最优值 */
    }
```

（3）程序运行与变通

运行程序，请输入整数：847313926

请输入插入的乘号个数 r：5

在数字串 847313926 中插入 5 个乘号，使乘积最大：

8*4*731*3*92*6=38737152

变通：如果需求插入乘号后的最小值，程序如何变通？

2. 基于组合穷举的插入乘号设计

作为验证，应用于组合穷举求取插入乘号的乘积最大值及其乘积式。

注意到 n 个数字之间共有 $n-1$ 个插入乘号的位置，在这 $n-1$ 个位置中组合选取 r 个位置 $t(1),t(2),\cdots,t(r)$供插入乘号。计算被这 r 个乘号分成的 $r+1$ 个整数的积 d。每计算一个 d 与存储最大乘积的变量 max 比较，若 $d>max$，则作赋值 $max=d$，同时乘号位置 t 数组赋值给 s 数组。

完成所有组合穷举后，得乘积最大值 max，按 s 数组的值打印插入乘号的乘积式及其最大乘积 max。

```
/* 基于组合穷举的插入乘号设计 */
#include <stdio.h>
#define N 30
void main()
{char b[11];
 int n,i,j,k,u,r,t[N],s[N];
 long y,d,max;
```

```
 printf("请输入整数：");
 scanf("%s",b);
 for(n=0,i=0;b[i]!='\0';i++) n++;
 printf("请输入插入的乘号个数 r：");
 scanf("%d",&r);
 printf("在整数%s 中插入%d 个乘号，使乘积最大：\n",b,r);
 i=1;t[i]=1;t[0]=0;t[r+1]=n;max=0;
 while(1)
    {if(i==r)
        {for(y=1,k=1;k<=r+1;k++)
           {for(d=0,u=t[k−1]+1;u<=t[k];u++)
               d=d*10+b[u−1]−48;
            y=y*d;                          /* 求各段数之积,以便比较求最大 */
           }
        if(y>max)
           {max=y;
            for(k=0;k<=r+1;k++) s[k]=t[k];
           }
        }
     else {i++; t[i]=t[i−1]+1; continue;}
     while(t[i]==n−r+i−1) i−−;              /* 调整或回溯或终止 */
     if(i>0) t[i]++;
     else break;
  }
 for(k=1;k<=r+1;k++)
    {for(d=0,u=s[k−1]+1;u<=s[k];u++)
        d=d*10+b[u−1]−48;
     if(k==r+1)
        printf("%ld",d);                    /* 输出插入*号后的乘积式 */
     else
        printf("%ld*",d);
     }
 printf("=%ld",max);
}
```

运行程序，输入：637829156

请输入插入的乘号个数 r：4

在整数 637829156 中插入 4 个乘号，使乘积最大：

63*7*82*915*6=198529380

6.4 0-1 背包问题求解

0-1 背包问题是应用动态规划设计求解的典型例题。本节在应用动态规划采用逆推与顺推两种设计求解一般 0-1 背包问题基础上，拓广到带两个约束条件的二维 0-1 背包问题的设计求解。

6.4.1 0-1 背包问题

【例 6.4】 已知 n 种物品和一个可容纳 c 重量的背包，物品 i 的重量为 w_i，产生的效益为 p_i。在装包时物品 i 可以装入，也可以不装，但不可拆开装。物品 i 可产生的效益为 $x_i p_i$，这里 $x_i \in \{0,1\}, c, w_i, p_i \in N^+$。问如何装包，所得装包总效益最大。

1. 最优子结构特性

0-1 背包的最优解具有最优子结构特性。设 $(x_1, x_2, \cdots, x_n)$，$x_i \in \{0,1\}$ 是 0-1 背包的最优解，那么 $(x_2, x_3, \cdots, x_n)$ 必然是 0-1 背包子问题的最优解：背包载重量 $c - x_1 w_1$，共有 $n-1$ 件物品，物品 i 的重量为 w_i，产生的效益为 p_i，$2 \leqslant i \leqslant n$。若不然，设 $(z_2, z_3, \cdots, z_n)$ 是该子问题的最优解，而 $(x_2, x_3, \cdots, x_n)$ 不是该子问题的最优解，由此可知

$$\sum_{2 \leqslant i \leqslant n} z_i p_i > \sum_{2 \leqslant i \leqslant n} x_i p_i \quad \text{且} \quad x_1 w_1 + \sum_{2 \leqslant i \leqslant n} z_i w_i \leqslant c$$

因此

$$x_1 p_1 + \sum_{2 \leqslant i \leqslant n} z_i p_i > \sum_{1 \leqslant i \leqslant n} x_i p_i \quad \text{且} \quad x_1 w_1 + \sum_{2 \leqslant i \leqslant n} z_i w_i \leqslant c$$

显然 $(x_1, z_2, z_3, \cdots, z_n)$ 比 $(x_1, x_2, \cdots, x_n)$ 收益更高，$(x_1, x_2, \cdots, x_n)$ 不是背包问题的最优解，与假设矛盾。因此，$(x_2, x_3, \cdots, x_n)$ 必然是 0-1 背包子问题的一个最优解。最优性原理对 0-1 背包问题成立。

2. 动态规划逆推求解

（1）算法设计

与一般背包问题不同，0-1 背包问题要求 $x_i \in \{0,1\}$, 即物品 i 不能折开，或者整体装入，或者不装。当约定每件物品的重量与效益均为整数时，可用动态规划求解。

按每一件物品装包为一个阶段，共分为 n 个阶段。

目标函数：$\max \sum_{i=1}^{n} x_i p_i$

约束条件：$\sum_{i=1}^{n} x_i w_i \leqslant c, \quad (x_i \in \{0,1\}, c, w_i, p_i \in N^+, i = 1,2,\cdots,n)$

① 建立递推关系

设 $m(i, j)$为背包容量 j，可取物品范围为 i,i+1,…,n 的最大效益值。则

当 $0 \leqslant j < w(i)$时，物品 i 不可能装入。最大效益值与 $m(i+1,j)$相同。

而当 $j \geqslant w(i)$时，有两个选择：

不装入物品 i，这时最大效益值为 $m(i+1,j)$；

装入物品 i，这时已产生效益 $p(i)$，背包剩余容积 $j-w(i)$，可以选择物品 $i+1,\cdots,n$ 来装，最大效益值为 $m(i+1,j-w(i))+p(i)$。

我们期望的最大效益值是两者中的最大者。于是有递推关系

$$m(i,j)=\begin{cases} m(i+1,j) & 0 \leqslant j < w(i) \\ max(\,m(i+1,j),\, m(i+1,j-w(i))+p(i)\,) & j \geqslant w(i) \end{cases}$$

其中 $w(i)$,$p(i)$均为正整数，$x(i) \in \{0,1\}$, $i=1,2,\cdots, n$。

边界条件为:

$m(n,j)=p(n)$,　当 $j \geqslant w(n)$；

$m(n,j)=0$,　当 $j < w(n)$。

所求最大效益即最优值为 $m(1,c)$。

② 逆推计算最优值

```
for(j=0;j<=c;j++)
   if(j>=w[n] ) m[n][j]=p[n];              /*  首先计算 m(n,j)  */
   else     m[n][j]=0;
for(i=n-1;i>=1;i--)                        /*  逆推计算 m(i,j)  */
for(j=0;j<=c;j++)
    if(j>=w[i] && m[i+1][j]<m[i+1][j-w[i]]+p[i])
       m[i][j]= m[i+1][j-w[i]]+p[i];
    else
        m[i][j]=m[i+1][j];
printf("最优值为%d",m(1,c));
```

③ 构造最优解

若 $m(i,cw)>m(i+1,cw)$, $i=1,2,\cdots,n-1$

则 $x[i]=1$；装载 $w(i)$。其中 $cw=c$ 开始，$cw=cw-x(i)*w(i)$。

否则，$x(i)=0$,不装载 $w(i)$。

最后，所装载的物品效益之和与最优值比较，决定 $w(n)$是否装载。

（2）0/1 背包问题逆推 C 程序实现

```
/*  逆推 0/1 背包问题  */
#include <stdio.h>
#define N 50
void main()
  {int p[N],w[N],m[N][5*N];
  int i,j,c,cw,n,sw,sp;
  printf("\n input n:"); scanf("%d",&n);        /*  输入已知条件  */
  printf("input c:"); scanf("%d",&c);
  for(i=1;i<=n;i++)
```

```
        {printf("input w%d,p%d:",i,i);
        scanf("%d,%d",&w[i],&p[i]);
        }
    for(j=0;j<=c;j++)
    if(j>=w[n] )
        m[n][j]=p[n];                          /*  首先计算 m(n,j)  */
    else
        m[n][j]=0;
    for(i=n-1;i>=1;i--)                        /*  逆推计算 m(i,j)  */
    for(j=0;j<=c;j++)
        if(j>=w[i] && m[i+1][j]<m[i+1][j-w[i]]+p[i])
            m[i][j]= m[i+1][j-w[i]]+p[i];
        else
            m[i][j]=m[i+1][j];
cw=c;
printf("c=%d \n",c);
printf("背包所装物品：\n");
printf(" i          w(i)        p(i)       \n");
for(sp=0,sw=0,i=1;i<=n-1;i++)                  /* 以表格形式输出结果 */
    if(m[i][cw]>m[i+1][cw])
        {cw-=w[i];sw+=w[i];sp+=p[i];
         printf("%2d         %3d         %3d \n",i,w[i],p[i]);
        }
  if(m[1][c-sp==p[n])
    {sw+=w[i];sp+=p[i];
     printf("%2d         %3d         %3d \n ",n,w[n],p[n]);
    }
printf("w=%d,    pmax=%d \n",sw,sp);
}
```

3. 动态规划顺推求解

（1）算法设计

目标函数、约束条件与分阶段同上。

① 建立递推关系

设 $g(i,j)$为背包容量 j，可取物品范围为：1,2,…,i 的最大效益值。则

当 $0 \leqslant j<w(i)$时，物品 i 不可能装入。最大效益值与 $g(i-1,j)$相同。

而当 $j \geqslant w(i)$时，有两种选择：

不装入物品 i，这时最大效益值为 $g(i-1,j)$；

装入物品 i，这时已产生效益 $p(i)$，背包剩余容积 $j-w(i)$可以选择物品 1,2,…,$i-1$ 来装，最

大效益值为 $g(i-1,j-w(i))+p(i)$。期望的最大效益值是两者中的最大者。

于是有递推关系

$$g(i,j)=\begin{cases} g(i-1,j) & 0 \leqslant j<w(i) \\ max(g(i-1,j),g(i-1,j-w(i))+p(i)) & j \geqslant w(i) \end{cases}$$

其中 $w(i)$,$p(i)$均为正整数，$x(i)\in\{0,1\}$, i=1,2,…,n。

边界条件为：

$g(1,j)=p(1)$,　当 $j\geqslant w(1)$；

$g(1,j)=0$,　当 $j<w(1)$。

所求最大效益即最优值为 $g(n,c)$。

② 顺推计算最优值

```
for(j=0;j<=c;j++)
    if(j>=w[1] ) g[1][j]=p[1];              /*  首先计算 g(1,j)  */
    else   g[1][j]=0;
for(i=2;i<=n;i++)                           /*  顺推计算 g(i,j)  */
for(j=0;j<=c;j++)
    if(j>=w[i] && g[i−1][j]<g[i−1][j−w[i]]+p[i])
          g[i][j]= g[i−1][j−w[i]]+p[i];
    else   g[i][j]=g[i−1][j];
printf("最优值为%d",g(n,c));
```

③ 构造最优解

若 $g(i,cw)>g(i-1,cw)$, $i=n,n-1,\cdots,2$

则 $x(i)=1$；装载 $w(i)$。其中 $cw=c$ 开始，$cw=cw-x(i)*w(i)$。

否则，$x(i)=0$，不装载 $w(i)$。

最后，所装载的物品效益之和与最优值比较，决定 $w(1)$是否装载。

（2）0/1 背包问题 C 程序设计

```
/*  顺推 0/1 背包问题 */
#include <stdio.h>
#define N 50
void main()
   {int p[N],w[N],g[N][5*N];
    int i,j,c,cw,n,sw,sp;
    printf("\n input n:"); scanf("%d",&n);          /*  输入已知条件  */
    printf("input c:"); scanf("%d",&c);
    for(i=1;i<=n;i++)
       {printf("input w%d,p%d:",i,i);
        scanf("%d,%d,%d",&w[i],&p[i]);
       }
    for(j=0;j<=c;j++)
```

```
    if(j>=w[1] )
      g[1][j]=p[1];                                  /*  首先计算 g(1,j)  */
    else
      g[1][j]=0;
    for(i=2;i<=n;i++)                                /*  顺推计算 g(i,j)  */
    for(j=0;j<=c;j++)
      if(j>=w[i] && g[i−1][j]<g[i−1][j−w[i]]+p[i])
        g[i][j]= g[i−1][j−w[i]]+p[i];
      else
        g[i][j]=g[i−1][j];
    cw=c;
    printf("c=%d \n",c);
    printf("背包所装物品：  \n");                      /* 构造最优解 */
    printf(" i        w(i)       p(i)      \n");
    for(sp=0,sw=0,i=n;i>=2;i−−)                      /* 以表格形式输出最优解 */
      if(g[i][cw]>g[i−1][cw])
        {cw−=w[i];sw+=w[i];sp+=p[i];
         printf("%2d       %3d       %3d \n",i,w[i],p[i]);
        }
      if(g[n][c]−sp==p[1])
       {sw+=w[i];sp+=p[i];
         printf("%2d       %3d       %3d \n ",1,w[1],p[1]);
       }
    printf("w=%d,   pmax=%d \n",sw,sp);
  }
```

运行程序，输入原始数据后，得：

```
input n:6
input c:60
input w1,p1:15,32
input w2,p2:17,37
input w3,p3:20,46
input w4,p4:12,26
input w5,p5:9,21
input w6,p6:14,30
c=60
```

背包所装物品：

```
 i        w(i)       p(i)
 2        17         37
```

```
3        20        46
5         9        21
6        14        30
w=60,   pmax=134
```

即装第 2、3、5、6 四件，装包重量为 60，获取最大效益 134。

4. 算法复杂度分析

以上动态规划算法的时间复杂度为 $O(nc)$，空间复杂度也为 $O(nc)$。通常 $c>n$，因而算法的时间复杂度与空间复杂度均高于 $O(n^2)$。

6.4.2　二维 0-1 背包问题

【例 6.5】 已知 n 种物品和一个可容纳 c 重量、d 容积的背包，物品 i 的重量为 w_i，容积为 v_i，产生的效益为 p_i。在装包时物品 i 可以装入，也可以不装，但不可拆开装，物品 i 可产生的效益为 $x_i p_i$，这里 $x_i \in \{0,1\}, c, w_i, p_i \in N^+$。问如何装包，使所得效益最大。

本例在例 6.4 基础上增加了容积的约束条件。下面应用穷举与动态规划两种算法设计求解。

1. 给定物品种数时穷举设计

（1）算法设计

当给定物品种数时，例如对 8 种物品，每一种物品的重量 $w(k)$、容积 $v(k)$与效益 $p(k)$ $(1\leqslant k\leqslant 8)$可任意从键盘输入，应用穷举设计，可设计 8 重循环：第一种物品的循环变量为 $x(1)$，其取值为 0 或 1；第 2 种物品的循环变量为 $x(2)$，其取值也为 0 或 1；……。因而穷举可由循环结构

```
for(x[1]=0;x[1]<=1;x[1]++)
for(x[2]=0;x[2]<=1;x[2]++)
……
for(x[8]=0;x[8]<=1;x[8]++)
```

来实现。对每一组 $x(k)(1\leqslant k\leqslant 8)$，计算重量之和 sw，容积之和 sv 与效益之和 sp，当 $sw\leqslant c$ 且 $sv\leqslant d$ 时通过 sp 与 $pmax$ 比较求取效益的最大值 $pmax$。

以上穷举设计的时间复杂度为 $O(2^n)$，空间复杂度为 $O(n)$。

（2）给定物品种数时的穷举程序实现

```
/*  二维 0/1 背包问题穷举求解  */
#define N 9
void main()
{int p[N],w[N],v[N],x[N],y[N];
 int i,k,c,d,cw,cv,sw,sv,sp,pmax;
 printf("input c:"); scanf("%d",&c);            /*  输入已知条件  */
 printf("input d:"); scanf("%d",&d);
 for(k=1;k<=8;k++)
```

```
    {printf("input w%d,v%d,p%d:",k,k,k);
     scanf("%d,%d,%d",&w[k],&v[k],&p[k]);
    }
  pmax=0;
  for(x[1]=0;x[1]<=1;x[1]++)
  for(x[2]=0;x[2]<=1;x[2]++)
  for(x[3]=0;x[3]<=1;x[3]++)
  for(x[4]=0;x[4]<=1;x[4]++)
  for(x[5]=0;x[5]<=1;x[5]++)
  for(x[6]=0;x[6]<=1;x[6]++)
  for(x[7]=0;x[7]<=1;x[7]++)
  for(x[8]=0;x[8]<=1;x[8]++)
    {for(sw=0,sv=0,sp=0,k=1;k<=8;k++)
       {sw=sw+x[k]*w[k];                          /* 求物品重量之和 */
        sv=sv+x[k]*v[k];                          /* 求物品容积之和 */
        sp=sp+x[k]*p[k];                          /* 求物品效益之和 */
       }
     if(sw<=c && sv<=d && sp>pmax)
       {pmax=sp;cw=sw;cv=sv;                      /* 通过比较求最大效益 pmax */
        for(i=1;i<=8;i++)
            y[i]=x[i];
        }
    }
  printf("c=%d, d=%d \n",c,d);
  printf(" i       w(i)     v[i]      p(i)      \n");
  for(i=1;i<=8;i++)                               /* 以表格形式输出结果 */
   if(y[i]==1)
      printf("%2d        %3d        %3d          %3d \n",i,w[i],v[i],p[i]);
  printf("sw=%d,   sv=%d,   pmax=%d \n",cw,cv,pmax);
}
```

（3）程序运行示例

```
input c:80
input d:150
input w1,v1,p1:10,19,28
input w2,v2,p2:12,23,35
input w3,v3,p3:15,29,44
input w4,v4,p4:18,35,55
input w5,v5,p5:20,38,58
input w6,v6,p6:16,31,47
```

input w7,v7,p7:14,27,41

input w8,v8,p8:22,40,60

c=80, d=150

i	w(i)	v[i]	p(i)
1	10	19	28
4	18	35	55
5	20	38	58
6	16	31	47
7	14	27	41

w=78,　v=150,　pmax=229

2. 动态规划设计

当物品种数 n 从键盘输入确定，每一件物品的重量、容积与效益均为正整数时，可应用动态规划设计求解。

（1）算法设计

与以上一维的背包问题相同，二维的 0−1 背包问题同样要求 $x_i \in \{0,1\}$, 即物品 i 不能拆开，或者整体装入，或者不装。与以上一维的背包问题不同，二维的 0−1 背包问题增加了容积的限制。

目标函数：$\max \sum_{i=1}^{n} x_i p_i$

约束条件：$\sum_{i=1}^{n} x_i w_i \leqslant c$,　$\sum_{i=1}^{n} x_i v_i \leqslant d$,

$$x_i \in \{0,1\}, c, d, w_i, v_i, p_i \in N^+, i = 1,2,\cdots,n$$

① 建立递推关系

设三维数组 $m(i,j,k)$为背包载重量 j，容积为 k，可取物品范围为：i,i+1,…,n 的最大效益值。则

当 0≤j<$w(i)$或 0≤k<$v(i)$时，物品 i 不可能装入。最大效益值与 $m(i+1,j,k)$相同。

而当 $j \geqslant w(i)$且 $k \geqslant v(i)$时，有两种选择：

不装入物品 i，这时最大效益值为 $m(i+1,j,k)$;

装入物品 i，这时已产生效益 $p(i)$；剩余载重量为 $j-w(i)$，可装容积为 $k-v(i)$，可以选择物品 i+1,…,n 来装，最大效益值为 m(i+1, $j-w(i)$,$k-v(i)$)+$p(i)$。

我们期望的最大效益值是两者中的最大者。于是有递推关系

$$m(i,j,k) = \begin{cases} m(i+1,j,k) & 0 \leqslant j < w(i) \text{ 或 } 0 \leqslant k < v(i) \\ \max(\, m(i+1,j,k), m(i+1,j-w(i),k-v(i)\,) + p(i)\,) & j \geqslant w(i) \text{ 且 } k \geqslant v(i) \end{cases}$$

其中 $w(i)$、$v(i)$、$p(i)$均为正整数，$x(i) \in \{0,1\}$, i=1,2,…,n。

边界条件为：

$m(n,j,k)=p(n)$，当 $j \geqslant w(n)$且 $k \geqslant v(n)$;

$m(n,j,k)=0$，当 $j<w(n)$ 或 $k<v(n)$。

背包可容重量 $c>0$，容量 $d>0$。

② 逆推计算最优值

```
for(j=0;j<=c;j++)
for(k=0;k<=d;k++)
    if(j>=w[n] && k>=v[n]) m[n][j][k]=p[n]; /*  首先计算 m(n,j,k)  */
    else   m[n][j][k]=0;
for(i=n-1;i>=1;i--)                          /*  逆推计算 m(i,j,k)  */
for(j=0;j<=c;j++)
for(k=0;k<=d;k++)
    if(j>=w[i] && k>=v[i] && m[i+1][j][k]<m[i+1][j-w[i]][k-v[i]]+p[i])
       m[i][j][k]= m[i+1][j-w[i]][k-v[i]]+p[i];
    else
       m[i][j][k]=m[i+1][j][k];
printf("最优值为%d",m(1,c,d));
```

③ 构造最优解

if(m[i][cw][cv]>m[i+1][cw][cv])

则 $x(i)=1$，装载第 i 件物品；

（其中 $cw=c$ 开始，$cw=cw-x(i)*w(i)$;$cv=d$ 开始，$cv=cv-x(i)*v(i)$）

否则，$x(i)=0$，不装载第 i 件物品。

最后，所装载的物品效益之和与最优值比较，决定第 n 件物品是否装载。

④ 算法复杂度分析

以上动态规划算法的时间复杂度为 $O(ncd)$，空间复杂度也为 $O(ncd)$。

以下程序实现中 c 按 $5n,d$ 按 $8n$ 设置，必要时可进行修改。当 n,c,d 比较大时，算法所占空间很大，大大限制了该算法的求解范围。

（2）二维 0/1 背包问题 C 程序设计

```
/*  二维 0/1 背包问题  */
#include <stdio.h>
#define N 9
void main()
{int p[N],w[N],v[N],m[N][5*N][8*N];
 int i,j,k,c,d,cw,cv,n,sw,sv,sp;
 printf("\n input n: "); scanf("%d",&n);          /*  输入已知条件  */
 printf("input c: "); scanf("%d",&c);
  printf("input d: "); scanf("%d",&d);
 for(i=1;i<=n;i++)
    {printf("input w%d,v%d,p%d: ",i,i,i);
    scanf("%d,%d,%d",&w[i],&v[i],&p[i]);
    }
```

```
  for(j=0;j<=c;j++)
  for(k=0;k<=d;k++)
     if(j>=w[n] && k>=v[n])
        m[n][j][k]=p[n];                          /*  首先计算 m(n,j,k)  */
     else
        m[n][j][k]=0;
  for(i=n-1;i>=1;i--)                             /*  逆推，计算 m(i,j,k)  */
for(j=0;j<=c;j++)
  for(k=0;k<=d;k++)
     if(j>=w[i] && k>=v[i] && m[i+1][j][k]<m[i+1][j-w[i]][k-v[i]]+p[i])
        m[i][j][k]= m[i+1][j-w[i]][k-v[i]]+p[i];
       else
          m[i][j][k]=m[i+1][j][k];
cw=c; cv=d;
printf("c=%d, d=%d \n",c,d);
printf("背包所装物品：\n");
printf(" i          w(i)          v[i]          p(i)          \n");
for(sp=0,sw=0,sv=0,i=1;i<=n-1;i++)              /* 以表格形式输出结果 */
   if(m[i][cw][cv]>m[i+1][cw][cv])
     { cw-=w[i];cv-=v[i];
       sw+=w[i];sv+=v[i];sp+=p[i];
       printf("%2d          %3d          %3d          %3d \n",i,w[i],v[i],p[i]);
     }
if(m[1][c][d]-sp==p[n])
   {sw+=w[i];sv+=v[i];sp+=p[i];
     printf("%2d          %3d          %3d          %3d \n ",n,w[n],v[n],p[n]);
     }
printf("sw=%d,    sv=%d,    pmax=%d \n",sw,sv,sp);
}
```

运行程序，输入与输出如下。

```
input n: 8
input c: 40
input d: 70
input w1,v1,p1: 8,14,20
input w2,v2,p2: 6,10,14
input w3,v3,p3: 11,19,28
input w4,v4,p4: 13,22,31
input w5,v5,p5: 5,9,12
input w6,v6,p6: 15,25,37
```

```
input w7,v7,p7: 12,20,27
input w8,v8,p8: 9,15,22
c=40, d=70
背包所装物品：
 i       w(i)      v[i]       p(i)
 3         11        19         28
 5          5         9         12
 6         15        25         37
 8          9        15         22
sw=40,   sv=68,   pmax=99
```

3. 以上两种算法比较

穷举法只适合于物品种数事先给定的情形，程序设计较为呆板。当物品种数 n 比较大时，时间复杂度为 $O(2^n)$，程序运行时间长。该算法的空间复杂度为 $O(n)$，而且当各种物品的重量、容积与效益不是整数时同样有效（修改相应的变量类型即可）。

动态规划可适应于物品种数 n 从键盘给定的情形，程序设计比较灵活。时间复杂度为 $O(ncd)$，空间复杂度高于 $O(n^3)$，当 n、c、d 较大时所占内存空间大。而且不适合各种物品的重量、容积与效益不是整数的情形。

6.5 最长子序列探索

本节应用动态规划探索两个典型的子序列问题：最长非降子序列与两个序列的最长公共子序列。

6.5.1 最长非降子序列

【例 6.6】 给定一个由 n 个正整数组成的序列，从该序列中删除若干个整数，使剩下的整数组成非降子序列，求最长的非降子序列。

1. 算法设计

设序列的各项为 $a(1),a(2),\cdots,a(n)$（可随机产生，也可从键盘依次输入），对每一个整数操作为一个阶段，共为 n 个阶段。

（1）建立递推关系

设置 b 数组，$b(i)$表示序列的第 i 个数（保留第 i 个数）到第 n 个数中的最长非降子序列的长度，$i=1,2,\cdots,n$。对所有的 $j>i$，比较当 $a(j)\geqslant a(i)$时的 $b(j)$的最大值，显然 $b(i)$为这一最大值加 1，表示加上 $a(i)$本身这一项。

因而有递推关系：

$b(i)=max(b(j))+1 \quad (a(j)\geqslant a(i),1\leqslant i<j\leqslant n)$

边界条件：$b(n)=1$

（2）逆推计算最优值

```
b[n]=1;
  for(i=n−1;i>=1;i−−)
    {max=0;for(j=i+1;j<=n;j++)
                if(a[i]<=a[j] && b[j]>max)
                    max=b[j];
     b[i]=max+1;                          /* 逆推得 b[i] */
     }
```

逆推依次求得 $b(n-1),\cdots,b(1)$，比较这 $n-1$ 个值得其中的最大值 $lmax$，即为所求的最长非降子序列的长度即最优值。

（3）构造最优解

从序列的第 1 项开始，依次输出 $b(i)$分别等于 $lmax,lmax-1,\cdots,1$ 的项 $a(i)$，这就是所求的一个最长非降子序列。

（4）算法复杂度分析

以上动态规划算法的时间复杂度为 $O(n^2)$，空间复杂度为 $O(n)$。

2. 求最长非降子序列程序设计

```
/* 在 n 个数的序列中求最长非降子序列 */
#include <stdio.h>
#include <math.h>
void main()
{ int i,j,n,t,x,max,lmax,a[2000],b[2000];
  t=time()%1000;srand(t);                 /*  随机数发生器初始化 */
  printf(" input n(n<2000):"); scanf("%d",&n);
  for(i=1;i<=n;i++)
    {a[i]=rand()%900+100;                 /* 产生并输出 n 个数组成的序列 */
     printf("%d   ",a[i]);
     }
  b[n]=1;lmax=0;
  for(i=n−1;i>=1;i−−)                     /* 逆推求最优值 lmax */
    {max=0;
     for(j=i+1;j<=n;j++)
         if(a[i]<=a[j] && b[j]>max)
             max=b[j];
     b[i]=max+1;                          /* 逆推得 b[i] */
     if(b[i]>lmax) lmax=b[i];             /* 比较得最大非降序列长 */
    }
```

```
    printf("\n L=%d.\n",lmax);                    /* 输出最大非降序列长 */
    x=lmax;
    for(i=1;i<=n;i++)
      if(b[i]==x)
        {printf("%d  ",a[i]);x--;}                /* 输出一个最大非降序列 */
}
```

3. 程序运行示例与变通

运行程序，输入 n=12，得

903 101 252 399 143 216 278 632 781 998 323 102

L=7.

101 143 216 278 632 781 998

注意，所给序列长度为 7 的非降子序列可能有多个，这里只输出其中一个。

（1）如果要求最长递增子序列，如何变通？

（2）如果要求最长非增子序列，如何变通？

6.5.2 最长公共子序列

一个给定序列的子序列是在该序列中删去若干个元素后所得到的序列。用数学语言表述，给定序列 $X=\{x_1,x_2,\cdots,x_m\}$，另一序列 $Z=\{z_1,z_2,\cdots,z_k\}$，X 的子序列是指存在一个严格递增下标序列 $\{i_1,i_2,\cdots,i_k\}$ 使得对于所有 j=1,2,⋯,k 有 $z_j=x_{i_j}$。例如，序列 $Z=\{b,d,c,a\}$ 是序列 $X=\{a,b,c,d,c,b,a\}$ 的一个子序列，或按紧凑格式书写，序列“bdca”是“abcdcba”的一个子序列。

若序列 Z 是序列 X 的子序列，又是序列 Y 的子序列，则称 Z 是序列 X 与 Y 的公共子序列。例如，序列“bcba”是“abcbdab”与“bdcaba”的公共子序列。

【例 6.7】 给定两个序列 $X=\{x_1,x_2,\cdots,x_m\}$ 和 $Y=\{y_1,y_2,\cdots,y_n\}$，找出序列 X 和 Y 的最长公共子序列。

1. 算法设计

求序列 X 与 Y 的最长公共子序列可以使用穷举法：列出 X 的所有子序列，检查 X 的每一个子序列是否也是 Y 的子序列，并记录其中公共子序列的长度，通过比较最终求得 X 与 Y 的最长公共子序列。对于一个长度为 m 的序列 X，其每一个子序列对应于下标集{1,2,⋯,m}的一个子集，即 X 的子序列数目多达 2^m 个。由此可见应用穷举法求解是指数时间的。

最长公共子序列问题具有最优子结构性质，应用动态规划设计求解。

（1）建立递推关系

设序列 $X=\{x_1,x_2,\cdots,x_m\}$ 和 $Y=\{y_1,y_2,\cdots,y_n\}$ 的最长公共子序列为 $Z=\{z_1,z_2,\cdots,z_k\}$，$\{x_i,x_{i+1},\cdots,x_m\}$ 与 $\{y_j,y_{j+1},\cdots,y_n\}$ (i=0,1,⋯,m;j=0,1,⋯,n)的最长公共子序列的长度为 $c(i,j)$。

若 $i=m+1$ 或 $j=n+1$，此时为空序列，$c(i,j)=0$（边界条件）。

若 $x(1)=y(1)$，则有 $z(1)=x(1)$,$c(1,1)=c(2,2)+1$（其中 1 为 $z(1)$这一项）；

若 $x(1)\neq y(1)$，则 $c(1,1)$取 $c(2,1)$与 $c(1,2)$中的最大者。

一般地，若 $x(i)=y(j)$，则 $c(i,j)=c(i+1,j+1)+1$；

若 $x(i)\neq y(j)$，则 $c(i,j)=max(c(i+1,j),c(i,j+1))$。

因而归纳为递推关系：

$$c(i,j)=\begin{cases} c(i+1,j+1)+1 & 1\leqslant i\leqslant m,1\leqslant j\leqslant n,x_i=y_j \\ \max(c(i,j+1),c(i+1,j)) & 1\leqslant i\leqslant m,1\leqslant j\leqslant n,x_i\neq y_j \end{cases}$$

边界条件：$c(i,j)=0$　　（$i=m+1$ 或 $j=n+1$）

（2）逆推计算最优值

根据以上递推关系，逆推计算最优值 $c(0,0)$流程为：

```
for(i=0;i<=m;i++) c[i][n]=0;                          /*  赋初始值  */
 for(j=0;j<=n;j++) c[m][j]=0;
 for(i=m−1;i>=0;i−−)                                  /* 计算最优值 */
 for(j=n−1;j>=0;j−−)
     if(x[i]==y[j])
        c[i][j]=c[i+1][j+1]+1;
     else   if(c[i][j+1]>c[i+1][j])
              c[i][j]=c[i][j+1];
     else   c[i][j]=c[i+1][j];
 printf("最长公共子串的长度为：%d",c[0][0]);          /* 输出最优值 */
```

以上算法时间复杂度为 $O(mn)$。

（3）构造最优解

为构造最优解，即具体求出最长公共子序列，设置数组 $s(i,j)$，当 $x(i)=y(j)$时 $s(i,j)=1$；当 $x(i)\neq y(j)$时 $s(i,j)=0$。

X 序列的每一项与 Y 序列的每一项逐一比较，根据 $s(i,j)$与 $c(i,j)$取值具体构造最长公共子序列。实施 $x(i)$与 $y(j)$比较，其中 $i=0,1,\cdots,m-1$;$j=t,1,\cdots,n-1$；变量 t 从 0 开始取值，当确定最长公共子序列一项时，$t=j+1$。这样处理可避免重复取项。

若 $s(i,j)=1$ 且 $c(i,j)=c(0,0)$时，取 $x(i)$为最长公共子序列的第 1 项;

随后，若 $s(i,j)=1$ 且 $c(i,j)=c(0,0)-1$ 时，取 $x(i)$最长公共子序列的第 2 项;

一般地，若 $s(i,j)=1$ 且 $c(i,j)=c(0,0)-w$ 时（w 从 0 开始，每确定最长公共子序列的一项，w 增 1），取 $x(i)$最长公共子序列的第 w 项。

构造最长公共子序列描述：

```
for(t=0,w=0,i=0;i<=m−1;i++)
for(j=t;j<=n−1;j++)
     if(s[i][j]==1 && c[i][j]==c[0][0]−w)
        {printf("%c",x[i]);
         w++;t=j+1;break;
        }
```

（4）算法的复杂度分析

以上动态规划算法的时间复杂度为 $O(mn)$。

2. 最长公共子序列 C 程序实现

```
/* 最长公共子序列 */
```

```
#include <stdio.h>
#define N   100
void main()
{char x[N],y[N];
 int i,j,m,n,t,w,c[N][N],s[N][N];
 printf("请输入序列 x：");                          /* 先后输入序列 */
 scanf("%s",x);
 printf("请输入序列 y：");
 scanf("%s",y);
 for(m=0,i=0;x[i]!='\0';i++) m++;
 for(n=0,i=0;y[i]!='\0';i++) n++;
 for(i=0;i<=m;i++) c[i][n]=0;                       /*  赋边界值  */
 for(j=0;j<=n;j++) c[m][j]=0;
 for(i=m-1;i>=0;i--)                                /* 递推计算最优值 */
 for(j=n-1;j>=0;j--)
     if(x[i]==y[j])
        {c[i][j]=c[i+1][j+1]+1;
         s[i][j]=1;
        }
     else
        {s[i][j]=0;
         if(c[i][j+1]>c[i+1][j])
               c[i][j]=c[i][j+1];
         else   c[i][j]=c[i+1][j];
        }
 printf("最长公共子序列的长度为：%d",c[0][0]);      /* 输出最优值 */
 printf("\n 最长公共子序列为：");                    /* 构造最优解 */
 t=0;w=0;
 for(i=0;i<=m-1;i++)
 for(j=t;j<=n-1;j++)
     if(s[i][j]==1 && c[i][j]==c[0][0]-w)
        {printf("%c",x[i]);
         w++;t=j+1;break;
        }
 printf("\n");
}
```

运行程序示例：

```
请输入序列 x：hsbafdreghsbacdba
请输入序列 y：acdbegshbdrabsa
最长公共子序列的长度为：9
最长公共子序列为：adeghbaba
```

6.6 最优路径搜索

本节应用动态规划探讨两类最优路径搜索问题，一类是点数值路径，即连接成路径的每一个点都带有一个数值；另一类是边数值路径，即连接成路径的每一条边都带有一个数值。

6.6.1 点数值三角形的最优路径搜索

点数值三角形是一个二维数阵：三角形由 n 行构成，第 k 行有 k 个点，每一个点都带有一个数值数。点数值三角形的数值可以随机产生，也可从键盘输入。最优路径通常由路径所经各点的数值和来确定。

【例 6.8】 随机产生一个 n 行的点数值三角形（即三角形的每一个点都带有一个正整数），寻找从顶点开始每一步可沿左斜（L）或右斜（R）向下至底的一条路径，使该路径所经过的点的数值和最大。

例如，n=6 时给出的点数值三角形如图 6.1 所示。

```
          7
       22   11
    22    9   28
  5   14   23   33
11   23    1   16    6
13   8   14   32   28    3
```

图 6.1　6 行点数值三角形

1. 算法设计

应用动态规划设计求解。

点数值三角形的数值存储在二维数组 $a(n,n)$。

（1）建立递推关系

设数组 $b(i,j)$为点(i,j)到底的最大数值和，字符数组 $stm(i,j)$指明点(i,j)向左或向右的路标。$b(i,j)$与 $stm(i,j)$$(i=n-1,\cdots,2,1)$的值由 b 数组的第 i+1 行的第 j 个元素与第 j+1 个元素值的大小比较决定，即有递推关系：

$b(i,j)=a(i,j)+b(i+1,j+1)$;$stm(i,j)$=“R”　　$(b(i+1,j+1)>b(i+1,j))$

$b(i,j)=a(i,j)+b(i+1,j)$;$stm(i,j)$=“L”　　$(b(i+1,j+1)\leqslant b(i+1,j))$

其中 $i=n-1,\cdots,2,1$

边界条件：$b(n,j)=a(n,j)$, $j=1,2,\cdots,n$。

所求的最大路径数值和即问题的最优值为 $b(1,1)$。

（2）逆推计算最优值

```
for(j=1;j<=n;j++)   b[n][j]=a[n][j];
for(i=n-1;i>=1;i--)          /* 逆推得 b[i][j] */
for(j=1;j<=i;j++)
   if(b[i+1][j+1]>b[i+1][j])
      {b[i][j]=a[i][j]+b[i+1][j+1];stm[i][j]='R';}
    else
      {b[i][j]=a[i][j]+b[i+1][j];stm[i][j]='L';}
Printf("%d",b(1,1));
```

（3）构造最优解

为了确定与并输出最大路径，利用 *stm* 数组从上而下查找：
先打印 *a*(1,1)，若 *stm*(1,1)=“R”，则下一个打印 *a*(2,2)，否则打印 *a*(2,1)。
一般地，在输出 *i* 循环（*i*=2,3,…,n）中：
若 *stm*(*i*−1,*j*)=“R”则打印“−R−”;*a*(*i*,*j*+1)；同时赋值 *j*=*j*+1。
若 *stm*(*i*−1,*j*)=“L”则打印“−L−”；*a*(*i*,*j*)；
依此打印出最大路径，即所求的最优解。
（4）算法的复杂度分析
以上动态规划算法的时间复杂度为 $O(n^2)$，空间复杂度也为 $O(n^2)$。

2. 最大路径寻求与测试的 C 程序

```
/* 点数值三角形的最大路径 */
#include <math.h>
#include <stdio.h>
main()
{ int n,i,j,t,s;
  int a[50][50],b[50][50];char stm[50][50];
  printf("请输入数字三角形的行数 n:");
  scanf("%d",&n);
  t=time()%1000;srand(t);                          /*随机数发生器初始化 */
  for(i=1;i<n;i++) j=rand();
  for(i=1;i<=n;i++)
     {for(j=1;j<=36−2*i;j++) printf(" ");
      for(j=1;j<=i;j++)
      {a[i][j]=rand()/1000+1;
       printf("%4d",a[i][j]);                      /* 产生并打印 n 行数字三角形 */
      }
      printf("\n");}
  printf("请在以上点数值三角形中从顶开始每步可左斜或右斜至底");
  printf("寻找一条数字和最大的路径.\n ");
  for(j=1;j<=n;j++) b[n][j]=a[n][j];
  for(i=n−1;i>=1;i−−)                              /* 逆推得 b[i][j] */
     for(j=1;j<=i;j++)
      if(b[i+1][j+1]>b[i+1][j])
          {b[i][j]=a[i][j]+b[i+1][j+1];stm[i][j]='R';}
       else
          {b[i][j]=a[i][j]+b[i+1][j];stm[i][j]='L';}
  printf("最大路径和为:%d\n",b[1][1]);             /* 输出最大数字和 */
  printf("最大路径为:%d",a[1][1]);j=1;             /* 输出和最大的路径 */
  for(i=2;i<=n;i++)
```

```
        if(stm[i-1][j]=='R')
            { printf("-R-%d",a[i][j+1]);j++;}
        else
           printf("-L-%d",a[i][j]);
    printf("\n");
}
```

运行程序，对于数据如图 6.1 所示的点数值三角形，输出如下。

```
最大路径和为：127
最大路径为：7-R-11-R-28-R-33-L-16-L-32
```

6.6.2　边数值矩形的最优路径搜索

边数值矩形也是一个二维数阵：矩形由 n 行 m 列构成，每一行有 $m-1$ 条横边，每一列有 $n-1$ 条竖边，每一条边都带有一个数值。最优路径通常由路径所经各边的数值和来确定。

【例 6.9】 已知 n 行 m 列的边数值矩阵，每一个点可向右或向下两个去向，试求左上角顶点到右下角顶点的所经边数值和最小的路径。

例如，给出一个 4 行 5 列的边数值矩形如图 6.2 所示。

```
  .  26  .  28  .  32  .   8  .
14      32      28      31      27
  .  15  .  20  .  22  .  20  .
18      24      4       14      4
  .  22  .  16  .  17  .  19  .
12      29      9       6       14
  .  23  .   6  .  18  .  22  .
```

图 6.2　一个 4 行 5 列的边数值矩形

1. 动态规划算法设计

设矩阵的行数 n，列数 m，每点为(i, j)，i=1,2,…,n；j=1,2,…,m。显然，该边数值矩阵每行有 $m-1$ 条横向数值边，每列有 $n-1$ 条纵向数值边。

从点(i,j)水平向右的边长记为 $r(i,j)(j<m)$，点(i,j)向下的边长记为 $d(i,j)(i<n)$。

（1）建立递推关系

设 $a(i,j)$为点(i,j)到右下角顶点的最短路程。$st(i,j)$为点(i,j)路标数组，其值取为{‘d’,‘r’}。

$a(i,j)$的值由 $a(i+1,j)+d(i,j)$与 $a(i,j+1)+r(i,j)$比较，取其较小者得到，即有递推关系：

$$a(i,j)=min(a(i+1,j)+d(i,j),a(i,j+1)+r(i,j))$$

$$st(i,j)=\{‘d’,‘r’\}$$

其中 i=1,2,…,$n-1$；j=1,2,…,$m-1$。

注意到右边纵列与下边横行只有唯一出口，因而有边界条件：

$$a(i,m)=a(i+1,m)+d(i,m);\quad i=n-1,n-2,\cdots,1$$

$a(n,j)=a(n,j+1)+r(n,j)$;　　$j=m-1,m-2,\cdots,1$

（2）逆推计算最优值

```
for(i=n−1;i>=1;i−−)
    {a[i][m]=a[i+1][m]+d[i][m];st[i][m]='d';}          /* 右边纵列初始化 */
for(j=m−1;j>=1;j−−)
    {a[n][j]=a[n][j+1]+r[n][j];st[n][j]='r';}          /* 下边横行初始化 */
for(i=n−1;i>=1;i−−)                                    /* 逆推求解 a(i,j) */
for(j=m−1;j>=1;j−−)
    if(a[i+1][j]+d[i][j]<a[i][j+1]+r[i][j])
      {a[i][j]=a[i+1][j]+d[i][j];st[i][j]='d';}
    else
      {a[i][j]=a[i][j+1]+r[i][j];st[i][j]='r';}
```

所求左上角顶点到右下角顶点的最短路程即最优值为 a(1,1)。

（3）构造最优解

利用路标数组输出最优解，从点(1,1)即 $i=1,j=1$ 开始判断：

```
    if(st[i][j]=='d')
        {printf("−%d−",d[i][j]);i++;}
    else
        {printf("−%d−",r[i][j]);j++;}
```

必要时可打印出点坐标。

（4）算法的复杂度分析

以上动态规划算法的时间复杂度为 $O(mn)$。

2. 求边数值矩阵图的最短路径 C 程序设计

```
/* 求边数值矩阵图的最短路径  */
#include <math.h>
#include <stdio.h>
void main()
{ int m,n,i,j,t,s,a[50][50],r[50][50],d[50][50];
  char st[50][50];
  t=time()%1000;srand(t);                          /*  随机数发生器初始化 */
  printf("在矩形图中寻找一条路程最小的路径.\n");
  printf("请输入矩阵的行数 n,列数 m:");
  scanf("%d,%d",&n,&m);                            /* r(1,1)−r(n,m−1);d(1,1)−d(n−1,m) */
  printf("\n");
  for(i=1;i<=n−1;i++)                              /* 随机产生并打印数值边矩阵 */
     {for(j=1;j<=m−1;j++)
        {printf("     . ");
         r[i][j]=rand()/1000+1; printf("%4d",r[i][j]);}
```

```
        printf("      . \n\n");
        for(j=1;j<=m;j++)
          {d[i][j]=rand()/1000+1; printf("%4d          ",d[i][j]);}
        printf("\n\n");
       }
   for(j=1;j<=m-1;j++)
       {printf("      . ");
        r[n][j]=rand()/1000+1; printf("%4d",r[n][j]);}
   printf("      . \n\n");
   for(i=n-1;i>=1;i--)
       {a[i][m]=a[i+1][m]+d[i][m];st[i][m]='d';}
   for(j=m-1;j>=1;j--)
       {a[n][j]=a[n][j+1]+r[n][j];st[n][j]='r';}
   for(i=n-1;i>=1;i--)
   for(j=m-1;j>=1;j--)
        if(a[i+1][j]+d[i][j]<a[i][j+1]+r[i][j])
            {a[i][j]=a[i+1][j]+d[i][j];st[i][j]='d';}
        else
            {a[i][j]=a[i][j+1]+r[i][j];st[i][j]='r';}
   printf("\n  最短路程为：%d.",a[1][1]);          /* 打印最短路程 */
   printf("\n  最短路径为：[1,1]");
   j=1;i=1;                                       /* 打印最短路径 */
   while(i<n || j<m)
       if(st[i][j]=='d')
           {printf("-%d-",d[i][j]);i++;
            printf("[%d,%d]",i,j);}
       else
           {printf("-%d-",r[i][j]);j++;
            printf("[%d,%d]",i,j);}
  printf("\n");
  }
```

3. 运行示例与说明

运行程序，对 4 行 5 列矩阵得：

最短路程为：98

最短路径为：

[1,1] -14- [2,1] -15- [2,2] -20- [2,3] -4- [3,3] -17- [3,4] -6- [4,4] -22- [4,5]

为操作简单，以上各例中的三角形或矩形的点数据或边数据是应用 C 语言的随机函数产生的。对于求解某些实际路径问题，具体的点数据或边数据可把随机产生改为通过键盘输入，然后进行搜索，确定最优路径。

6.7 动态规划与其他算法的比较

前几章介绍的递推、贪心算法与动态规划相关联，本节试就动态规划与递推、贪心算法进行简单比较。

6.7.1 动态规划与递推比较

动态规划根据不同阶段之间的状态转移，通过递推求得问题的最优值。这里，注意不能把动态规划与递推两种算法相混淆，不要把递推当成是动态规划，也不要把动态规划当成递推。

（1）动态规划是用来求解多阶段最优化问题的有效算法，而递推一般是解决某些判定性问题或计数问题的方法，两者求解对象有区别。

例如，用 m 种零币 $t_1, t_2, \cdots, t_m$（单位为分，约定 $t_1 < t_2 < \cdots < t_m$）来兑换 n 分钱整币，可通过动态规划设计求兑换的最少零币数，通过递推求不同的兑换种数。

（2）动态规划求解的多阶段决策问题必须满足最优子结构特性，而递推所求解的计数问题无需满足最优子结构特性。

（3）动态规划在求得问题的最优值后通常需构造出最优值的最优决策序列，即求出最优解，而递推在求出计数结果后没有最优解的构造需求。

（4）从算法的时间复杂度而言，动态规划通常设置二维以上数组，通过二重以上循环递推完成最优值求解，其时间复杂度与二重循环以上的递推时间复杂度基本相同，一般都在 $O(n^2)$ 以上。

（5）当动态规划与递推需设置三维数组时，其空间复杂度都比较高，大大限制了求解范围，这是动态规划与递推所面临的共同问题。

6.7.2 动态规划与贪心算法比较

动态规划与贪心算法都是求解最优化问题的常用算法，通过比较了解这两种算法在应用上的不同点。

（1）动态规划算法求解最优化问题，通过建立每一阶段状态转移之间的递推关系，并经过递推来求取最优值。贪心算法在求解最优化问题时，从初始阶段开始，每一个阶段总是作一个使局部最优的贪心选择，不断把将问题转化为规模更小的子问题，最后求得最优化问题的解。

（2）动态规划算法是求解最优化问题的常用算法，其结果总是最优的。贪心算法在求解最优化问题时，每一决策只着眼于当时局部最优的贪心选择。这样处理，对大多数优化问题能得到最优解，但有时并不能求得最优解。

（3）动态规划存在一个空间的问题，随着问题维数的增加，其效率与求解范围受到限制。

从求解效率来说，贪心算法比动态规划更高，且一般不存在空间限制的影响。例如，对一些 NP 完全问题或规模很大的优化问题，可通过贪心算法得到较优解，而用动态规划可能达不到目的。

习　　题

1. 简述动态规划求解最优化问题的求解步骤。

2. 用 m 种零币 $t_1, t_2, \cdots, t_m$（单位为分，约定 $t_1 < t_2 < \cdots < t_m$）来找 n 分钱整币，试求最少的零币数。

3. 随机产生一个 n 行 m 列的点数值矩阵,寻找从矩阵左上角至右下角，每步可向下(D)或向右(R)的一条数值和最大的路径。

4. 已知边数值三角形每两点间的距离，每一个点有向左或向右两条边，求边数值三角形顶点 A 到底边的最短路径。

5. 在一个由 n 个数字组成的数字串中插入 r 个加号($1 \leqslant r < n < 10$)，将它分成 r+1 整数，找出一种加号插入方法，使得这 r+1 个整数的和最小。

例如，如何在整数 468214962 中插入 5 个加号，使所成的 6 个整数的和最小？

6. 给定 n 个整数（可能为负整数）组成的序列 $a_1, a_2, \cdots, a_n$，求该序列形如 $\sum_{k=i}^{j} a_k$ 的字段和的最大值。

第 7 章 模　　拟

应用计算机程序设计模拟自然界的随机现象，模拟特定条件下的操作过程，是程序设计难以把握且颇具魅力的课题之一。通过模拟设计，可解决一些人工操作力所不及的疑难问题，同时可培养我们分析问题的能力与探索创新的意识。

本章具体介绍运算模拟、随机模拟与过程模拟的几个实际应用。模拟自然过程的“智能算法”将在下一章介绍。

7.1 模拟概述

在自然界与日常生活中，许多现象带有不确定性，有些问题甚至很难建立确定的数学模型，因而对这些实际问题很难建立与应用常用递推、递归或回溯等算法处理。此时可试用模拟法进行探索求解。

根据模拟对象的不同特点，计算机模拟可分为决定性模拟与随机性模拟。

决定性模拟是对决定性过程进行的模拟，其模拟的事件按其固有的规律发生发展，最终得出一个明确的结果。

例如，运算模拟就是决定性模拟，应用模拟整数的乘除运算求解一些高精度计算问题，结果是确定的。

随机性模拟的对象是随机事件，其变化过程相当复杂。随机模拟要利用随机数设定某一范围内的随机值，并将这些随机值作为参数实施模拟。

例如数字模拟（又称数字仿真）可应用计算机语言所提供的随机数进行一些疑难定积分的近似计算。

7.2 运算模拟

运算模拟是按整数的四则运算法则进行模拟操作，最后得出模拟运算的结果。

7.2.1　运算模拟描述

运算模拟，主要是模拟整数逐位乘除的运算过程，求解一些整数计算问题。

在实施乘除运算模拟之前，必须根据参与运算整数的实际设置模拟量，以模拟乘除运算进程中数值的变化，并判定运算是否结束。

1. 模拟除法运算

模拟整数除运算，设除运算过程中被除数为 a，除数为 p，试商所得的商为 $b=a/p$，所得余数为 $c=a\%p$。

实施除运算，需根据问题的具体实际确定终止运算的循环条件。通常以试商的余数是否为 0 作为运算的终止条件：当 $c\neq0$ 时，继续试商下去，直至余数 $c=0$ 时，实现整除，终止运算。

除运算模拟框架描述：

```
输入<原始数据>
确定<初始量>
while(<循环条件>)
    { a=c*10+m;          /* 构造被除数 a，m 为<构造量> */
      b=a/p;             /* 实施除运算,计算商 b   */
      printf(b);
      c=a%p;             /* 试商得余数 c */
    }
```

其中<原始数据>，<初始量>，<循环条件>与<构造量>必须根据模拟除运算问题的具体实际确定。

2. 模拟乘法运算

乘运算通常从低位开始，乘积结果须从高位到低位输出，因此有必要设置数组。设 w 数组表示乘运算的一个乘数，也表示该数乘以 p（另一个乘数）的积：$w(1)$表示个位数，$w(2)$为十位数，……。

实施乘运算必须考虑进位。设进位数为 m，显然，乘数的第 k 位数 $w(k)$乘以 p 的结果为 $a=w(k)*p+m$，然后把 a 的个位数存储为积的第 k 位数：$w(k)=a\%10$；而 a 的十位及以上的值作为下轮运算的进位数：$m=a/10$。

w 数组（一个乘数）与进位数 m 的初值、乘运算的结束条件由所求问题的具体实际确定，通常使乘运算达到某一特定值或达到某一规定位数后结束。

乘运算模拟框架描述：

```
输入<原始数据>
确定<初始量>
while(<循环条件>)
    { k=k+1;
      a=w(k)*p+m;          /* 计算乘积 a，m 为<进位数> */
      w(k)=a%10;           /* 乘积 a 的个位存储到 w(k) */
```

```
        m=a/10;                          /* 乘积 a 的十位以上作为下轮的进位数 */
    }
输出(w(d),w(d−1),…,w(1));              /* 从高位到低位输出乘积 */
```

乘运算模拟的<原始数据>，<初始量>，<循环条件>与<进位数>根据模拟乘运算问题的实际确定。

除运算与乘运算模拟通常在设置的相应循环中完成，其时间复杂度一般为 $O(n)$。

7.2.2 *n*个 1 的整除问题

首先介绍一个应用除运算模拟求解的简单趣题。

1. *n* 个 1 被 2009 整除问题

【例 7.1】 一个整数由 *n* 个 1 组成，问该数能被 2009 整除，*n* 至少为多大？

该问题解的存在性是肯定的，可应用抽屉原理给出简单证明。问题是应用什么方法才能求出满足题意要求的 *n* 值，决非轻而易举。

（1）模拟除运算设计

模拟整数除运算，每次试商的被除数为 *a*，除数 *p*=2009，每次试商的余数为 *c*。显然，被除数 *a*=*c*∗10+1（构造量 *m*=1 由问题要求每一位数为 1 决定），每次试商所得余数为 *c*=*a*%2009。

设置初始值 *c*=1111,*n*=4，进入模拟整除循环。循环条件为 $c \neq 0$。每循环一次，计算 *a*=*c*∗10+1 为本轮运算的被除数，并由 *c*=*a*%2009 求余数，同时统计“1”的个数的变量 *n* 增 1。

若余数 *c*=0，结束，输出 *n* 的值。

因而 *n* 个 1 被 2009 整除问题模拟除法设计的参量为：

初始量：*n*=4; *c*=1111；

循环条件：*c*!=0；

构造量：*m*=1。

（2）*n* 个 1 被 2009 整除的 C 程序实现

```
/*   n 个 1 被 2009 整除   */
#include<stdio.h>
void main()
{ int a,c,n;
  c=1111;n=4;                        /* 确定初始值 */
  while(c!=0)
    { a=c*10+1;                      /* 构造被除数 a */
      c=a%2009; n++;                 /* 实施除运算,得余数 c */
    }
  printf("至少%d 个 1 才能被 2009 整除.\n",n);
}
```

运行程序，得

至少 210 个 1 才能被 2009 整除.

（3）讨论与变通

应用模拟整数除运算较为简便地解决了 n 个 1 被 2009 整除问题。

除数为给定的 2009，省略了除数变量 p。同时，无需输出商，省略了商变量 b。因需统计被除数中 1 的个数，增设了变量 n。可见模拟量的设置并不是一成不变的。

在循环前，也可以设置初始值 n=3；c=111；或 n=2；c=11 等，即初始值的确定也是灵活的。

2. 积为 n 个 1 的数字游戏

把上一例的除数从 2009 作适当引申。

【例 7.2】 两位计算机爱好者在进行“积为 n 个 1 的数字游戏”：其中一位给定一个正整数 p（约定整数 p 为个位数字不是 5 的奇数），另一位寻求正整数 q，使得 p 与 q 之积为全是 1 组成的整数。

（1）模拟除运算设计

设整数除运算每次试商的被除数为 a，除数为 p（即给定的正整数），每次试商的商为 b，相除的余数为 c。

被除数 a=c*10+1，试商余数 c=a%p，商 b=a/p 即为所寻求数 q 的一位。若余数 c=0，结束；否则，继续下一轮运算，直到 c=0 为止。

每商一位，设置变量 n 统计积中“1”的个数，同时输出商 b（整数 q 的一位数）。

以余数 c≠0 作为条件设置条件循环，循环外赋初值：c=111,n=3;或 c=11,n=2 等。

“积为 n 个 1 的数字游戏”模拟除法设计的参量：

原始数据：个位数字不是 5 的奇数 p（从键盘输入）；

初始量：c=111；n=3；（或 c=11；n=2）

循环条件：c!=0；

构造量：m=1。

（2）积为 n 个 1 的 C 程序实现

```
/*   积为 n 个 1 的乘数探求     */
#include<stdio.h>
void main()
{ int a,b,c,p,n;
printf("\n 请输入整数 p:"); scanf("%d",&p);
printf("\n 给出的整数 p=%d",p);
printf("\n 寻求的整数 q=");
n=3; c=111;                                    /* 确定初始值 */
while(c!=0)
     { a=c*10+1;
       c=a%p; b=a/p; n++;                      /* 实施除运算模拟 */
       printf("%d",b);}                        /* 输出整数 q 的一位数 */
printf("\n 乘积 p*q 为%d 个 1.\n",n);
}
```

运行程序，请输入整数 p：89
寻求的整数 q=124843945068664169787765293383270911360799
乘积 p*q 为 44 个 1.

7.2.3 尾数前移问题

【例 7.3】 整数 *n* 的尾数是 9，把尾数 9 移到其前面(成为最高位)后所得的数为原整数 *n* 的 3 倍，原整数 *n* 至少为多大?

这是《数学通报》上发表的一个具体的尾数前移问题。我们要求解一般的尾数前移问题：

整数 *n* 的尾数 *q*（限为一位）移到 *n* 的前面所得的数为 *n* 的 *p* 倍，记为 *n*(*q*,*p*)。这里约定 $2 \leqslant p \leqslant q \leqslant 9$。

对于指定的尾数 *q* 与倍数 *p*，求解 *n*(*q*,*p*)。

下面试用模拟除运算与模拟乘运算两种方法设计求解。

1. 模拟整数除法

（1）算法设计

设 *n* 为 *efg*…*wq*（每一个字母表示一位数字），尾数 *q* 移到前面变为 *qefg*…*w*，它是 *n* 的 *p* 倍，意味着 *qefg*…*w* 可以被 *p* 整除，商即为 *n*=*efg*…*wq*。注意到尾数 *q* 前移后数的首位为 *q*，第二高位 *e* 即为所求 *n* 的首位，第三高位 *f* 即为 *n* 的第二高位，等等。这一规律是构造被除数的依据。

应用模拟整数除法：首先第一位数 *q* 除以 *p*（注意约定 *q*≥*p*），余数为 *c*，商为 *b*。输出数字 *b* 作为所求 *n* 的首位数。

进入模拟循环，当余数 *c*=0 且商 *b*=*q* 时结束，因而循环条件为 *c*!=0 || *b*!=*q*。在循环中计算被除数 *a*=*c*∗10+*b*，注意运算量 *m*=*b*；试商得 *b*=*a*/*p*，输出作为所求 *n* 的一位；求得余数 *c*=*a*%*p*；然后 *b* 与 *c* 构建下一轮试商的被除数，依此递推。

因而尾数前移问题模拟整数除法的参量为。

原始数据：尾数字 *q*，倍数 *p*；

初始量：*b*=*q*/*p*；*c*=*q*%*p*；

循环条件：*c*!=0 || *b*!=*q*；

构造量：*b*（即上一轮除运算的商）。

（2）尾数前移问题模拟整数除法程序实现

```
/*   模拟除法求解尾数前移问题 */
#include<stdio.h>
void main()
{ int a,b,c,p,q;
printf("请输入整数 n 的指定尾数 q:");
   scanf("%d",&q);                                  /* 输入处理数据 q,p */
   printf("请输入前移后为 n 的倍数 p:");
   scanf("%d",&p);
```

```
    b=q/p;c=q%p;                          /* 确定初始条件 */
    printf("n(%d,%d)=%d",q,p,b);          /* 输出 n 的首位 b */
    while(c!=0 || b!=q)                   /* 试商循环处理 */
      { a=c*10+b;
        b=a/p;c=a%p;                      /* 模拟整数除法 */
        printf("%d",b);
      }
   printf("\n");
   }
```

运行程序，得

```
请输入整数 n 的指定尾数 q:9
请输入前移后为 n 的倍数 p:3
n(9,3)=310344827586206896551724137 9
```

2. 模拟整数乘法

（1）算法设计

设置存储数 n 的 w 数组。从 $w(1)=q$ 开始，乘数 p 与 n 的每一位数字 $w(i)$相乘后加进位数 m，得 $a=w(k)*p+m$；积 a 的十位以上的数作为下一轮的进位数 $m=a/10$；而 a 的个位数此时需赋值给乘积的下一位 $w(i+1)=a\%10$。

当计算的被除数 a 为尾数 q 时结束。

因而尾数前移问题模拟整数乘法参量为。

原始数据：输入尾数字 q，倍数 p；

初始量：$w(1)=q$;$m=0$;k=1;$a=p*q$；

循环条件：$a!=q$；

进位数：$m=a\%10$。

（2）程序设计与运行示例：

```
/*  模拟乘法求解尾数前移问题 */
#include<stdio.h>
void main()
{ int a,m,j,k,p,q,w[100];
  printf("请输入尾数字 q,倍数 p:");
  scanf("%d,%d",&q,&p);
  for(j=1;j<100;j++) w[j]=0;              /* 数组清零 */
  w[1]=q;m=0;k=1;a=p*q;                   /* 输入初始量 */
  while(a!=q)
    { a=w[k]*p+m;
      k++;
      w[k]=a%10;m=a/10;                   /* 模拟整数乘法，m 为进位数 */
    }
```

```
    printf("n(%d,%d)=",q,p);
    for(j=k-1;j>=1;j--)                          /* 从高位到低位打印每一位 */
      printf("%d",w[j]);
    printf("\n 共%d 位。\n",k-1);
    }
```

运行程序，请输入尾数字 q，倍数 p:7,6，得

n(7,6)=118644067796610169491525423728813559322033898305084745762 7

共 58 位。

以上所求的数为高精度多位数，靠人工计算或其他算法尚难以入手，而应用模拟乘除运算得到较好解决。

7.2.4 阶乘与幂的计算

【例 7.4】 高精度计算阶乘 $n!$与幂 p^n 的值（正整数 n,p 从键盘输入）。

1. 模拟乘运算设计

模拟乘运算，为输出积方便，设置一维数组 w 存储一个乘数，也存储乘运算的积：$w(1)$为个位数，$w(2)$为十位数，余类推。因乘运算中有进位，设置 m 为进位数。

首先要解决高精度计算到多少位。为此，通过对阶乘 $n!$与幂 p^n 实施常用对数求和

$$s=\lg 2+\lg 3+\cdots+\lg n$$

或 $s=n*\lg p$

则 $z=[s]+1$ 即为所求阶乘 $n!$或幂 p^n 的高精度位数。

设置 h 循环控制乘 n 次；设置 k 条件循环控制乘到 z 位。试用 t 去乘 w 的每一个元素，得

$$a=w(k)*t+m;\ w(k)=a\%10;\ m=a/10;\ (k=1,2,\cdots z)$$

乘数 t 随所计算的对象而异: 计算阶 $n!$时 $t=1,2,\cdots,n$；计算幂 p^n 时，$t=p$。完成模拟乘运算后，从高位开始输出所得乘积。

模拟乘运算算法参量为：

原始数据：j:1 为计算 $n!$，2 为计算 p^n；n,p（当 $j=2$ 时）；

初始量：$w(1)=1$; $m=0$;$k=0$;

循环条件：$k<z$;

构造乘积：$a=w(k)*t+m$；$m=a/10$;$t=h$（计算 $n!$时）或 $t=p$(计算 p^n 时)。

2. 阶乘 $n!$与幂 p^n 模拟乘运算 C 程序实现

```
/* 计算阶乘 n!与幂 p^n */
#include<stdio.h>
#include<math.h>
void main()
{int a,m,n,t,k,h,j,p,z,w[2001]; float s;
 printf("请选择 1: 计算 n!   2: 计算 p^n ");
 scanf("%d",&j);                          /* 输入选择的数据 */
```

```
    printf("请输入 n: ");
    scanf("%d",&n);
    if(j==2)
        { printf("请输入 p: ");
          scanf("%d",&p);
         }
    s=0;
    for(k=1;k<=n;k++)                    /* 计算高精度计算的位数 */
      {if(j==1) t=k;
            else t=p;
       s+=log10(t);
      }
    z=s+1;
    for(k=1;k<=z;k++) w[k]=0;
    w[1]=1;
    for(h=1;h<=n;h++)                    /* 设置循环乘 n 次 */
      {m=0;k=0;
       if(j==1) t=h;                     /* 确定乘数 t */
       else t=p;
       while(k<z)                        /* 实施乘运算，乘到 z 位 */
          {k++;a=w[k]*t+m;
          w[k]=a%10; m=a/10;
       }
      }
    if(j==1) printf("\n %d!=",n);
    else printf("\n %d^%d=",p,n);
    for(k=z;k>=1;k--) printf("%d",w[k]);   /* 从高位开始逐位输出 */
    printf("\n 共有%d 位.",z);
  }
```

3. 运行示例

输入 j=1,n=40，得

40!=815915283247897734345611269596115894272000000000

共有 48 位.

输入 j=2,n=30,p=23，得

23^30=710943487911513630243895542864209967984 49

共有 41 位.

以上模拟乘运算设计数组预定计算到 2000 位，必要时可进行增减。模拟乘运算的时间复杂度为 $O(nz)$，其中 z 为乘积的位数。

应用运算模拟可进行整数的准确计算，也可以进行一些无理数指定精度的近似计算。

7.2.5 求圆周率π

关于圆周率π的计算，历史非常久远。我国数学家祖冲之最先把圆周率π计算到 3.141 592 6，领先世界一千多年。其后，德国数学家鲁特尔夫把π计算到小数点后 35 位，日本数学家建部贤弘计算到 41 位。1874 年英国数学家香克斯利用微积分倾毕生精力把π计算到 707 位，但 528 位后的数值是错的。

应用计算机计算圆周率π曾有过计算到数百万位至数千万位的报导，主要是通过π的计算说明计算机的速度与计算软件的性能。

【例 7.5】 计算圆周率π，精确到小数点后指定的 x 位。

1. 算法设计

（1）选择计算公式

计算圆周率π的公式很多，选取收敛速度快且容易操作的计算公式是设计的首要一环。

我们选用以下公式：

$$\frac{\pi}{2}=1+\frac{1}{3}+\frac{1\times 2}{3\times 5}+\frac{1\times 2\times 3}{3\times 5\times 7}+\cdots+\frac{1\times 2\times\cdots\times n}{3\times 5\times\cdots\times(2n+1)}$$

$$=1+\frac{1}{3}\left(1+\frac{2}{5}\left(1+\cdots+\frac{n-1}{2n-1}\left(1+\frac{n}{2n+1}\right)\cdots\right)\right) \qquad (7.1)$$

（2）确定计算项数

首先，要依据输入的计算位数 x 确定所要加的项数 n。显然，若 n 太小，不能保证计算所需的精度；若 n 太大，会导致作过多的无效计算。可证明，式中分式第 n 项之后的所有余项之和 $R_n<a_n$。因此，只要选取 n，满足 $a_n<\frac{1}{10^{x+1}}$ 即可。即只要使

$$\lg 3+\lg\frac{5}{2}+\cdots+\lg\frac{2n+1}{n}>x+1 \qquad (7.2)$$

即可。

于是可设置对数累加实现计算到 x 位所需的项数 n。为确保准确，算法可设置计算位数超过 x 位（例如 x+5 位），只打印输出 x 位。

（3）模拟乘除综合运算

设置 a 数组，下标根据计算位数预设 5 000，必要时可增加。计算的整数值存放在 $a(0)$，小数点后第 i 位存放在 $a(i)$中(i=1,2,…)。

依据公式（7.1），应用模拟乘除运算进行计算：

数组除以 2n+1，乘以 n，加上 1；再除以 2n−1，乘以 n−1，加上 1；…。这些数组操作设置在 j(j=n,n−1,…,1)循环中实施。

按公式实施除法操作：被除数为 c，除数 d 分别取 2n+1,2n−1,…,3。商仍存放在各数组元素（$a(i)=c/d$）。余数（$c\%d$）乘 10 加在后一数组元素 $a(i+1)$上，作为后一位的被除数。

按公式实施乘法操作：乘数 j 分别取 n,n−1,…,1。乘积要注意进位，设进位数为 b，则对计算的积 $a(i)=a(i)*j+b$，取其十位以上数作为进位数 $b=a(i)/10$，取其个位数仍存放在原数组

元素 $a(i)=a(i)\%10$。

模拟乘除运算描述：

```
for(c=1,j=n;j>=1;j--)                    /* 按公式分步计算 n 次 */
  {d=2*j+1;
   for(i=0;i<=x+4;i++)                   /* 各位实施除 2j+1 */
       {a(i)=c/d; c=(c%d)*10+a(i+1);}
   a(x+5)=c/d;
   for(b=0,i=x+5;i>=0;i--)               /* 各位实施乘 j */
       {a(i)=a(i)*j+b; b=a(i)/10;a(i)=a(i)%10;}
   a(0)=a(0)+1;c=a(0);                   /* 整数位加 1 */
  }
for(b=0,i=x+5;i>=0;i--)                  /* 按公式各位乘 2 */
  {a(i)=a(i)*2+b; b=a(i)/10;a(i)=a(i)%10;}
```

（4）输出结果

循环实施除乘操作完成后，按数组元素从高位到低位顺序输出。因计算位数较多，为方便查对，每一行控制打印 50 位，每 10 位空一格。

（5）复杂度分析

以上模拟乘除运算算法的时间复杂度 $O(n\log n)$，其中 n 为所需计算的位数，$\log n$ 约为所需计算的项数。

2. 圆周率π的 C 程序实现

```
/* 高精度计算圆周率π  */
#include <math.h>
#include<stdio.h>
 void main()
  { float s; int b,x,n,c,i,j,d,l,a[5000];
    printf("\n 请输入精确位数:"); scanf("%d",&x);
    for(s=0,n=1;n<=5000;n++)             /* 累加确定计算的项数 n */
      {s=s+log10((2*n+1)/n);
    if (s>x+1) break;}
for(i=0;i<=x+5;i++) a[i]=0;
for(c=1,j=n;j>=1;j--)                    /* 按公式分步计算 */
  {d=2*j+1;
   for(i=0;i<=x+4;i++)                   /* 各位实施除 2j+1 */
      {a[i]=c/d; c=(c%d)*10+a[i+1];}
   a[x+5]=c/d;
   for(b=0,i=x+5;i>=0;i--)               /* 各位实施乘 j */
      {a[i]=a[i]*j+b; b=a[i]/10;a[i]=a[i]%10;}
  a[0]=a[0]+1;c=a[0];                    /* 整数位加 1 */
```

```
    }
for(b=0,i=x+5;i>=0;i--)                          /* 按公式各位乘 2 */
      {a[i]=a[i]*2+b; b=a[i]/10;a[i]=a[i]%10;}
printf("\n          pi=%d.",a[0]);                /* 遂位输出计算结果 */
for(l=10,i=1;i<=x;i++)
    { printf("%d",a[i]);l++;
      if (l%10==0) printf(" ");
      if (l%50==0) printf("\n");}
printf("\n");
}
```

运行程序，输入 x=200，得

```
          pi=3.1415926535 8979323846 2643383279 5028841971
6939937510 5820974944 5923078164 0628620899 8628034825
3421170679 8214808651 3282306647 0938446095 5058223172
5359408128 4811174502 8410270193 8521105559 6446229489
5493038196
```

7.3 随机模拟

随机模拟就是应用计算机语言提供的随机函数值来模拟随机发生的事件，或模拟自然界的一些随机现象。为了使随机更加贴近自然现象，要注意对所提供的随机数发生器进行初始化。

7.3.1 进站时间模拟

【例 7.6】 根据统计资料，车站进站口进一个人的时间至少为 2 秒，至多为 8 秒。试求 *n* 个人进站所需时间。

1. 随机模拟算法

一个人的进站时间至少为 2 秒，至多为 8 秒，设时间精确到小数点后一位，则每一个人进站的时间在 2.0,2.1,2.2,…,8.0 等数据中随机选取。

应用 C 语言库函数 *srand*(*t*)进行随机数发生器初始化，其中 *t* 为所取的时间秒数。这样可避免随机数从相同的整数取值。C 库函数中的随机函数 *rand*()产生−90～32 767 之间的随机整数，在随机模拟设计时，为产生区间[*a*,*b*]中的随机整数，可以应用 C 语言的整数求余运算实现：

```
rand()%(b−a+1)+a;
```

为简化设计，把每一个人的进站时间乘以 10 转化为整数，即每一个人的进站时间为

rand()%61+20，随机取值范围为 20,21,22,…,80，单位为 1/10 秒。则 *n* 个人的进站时间为

```
for(t=0,i=1;i<=n;i++)
    t=t+rand()%61+20;
```

求和完成后，转化为时间的分，秒输出。

2. 进站时间模拟程序实现

```
/* 进站时间模拟 */
#include <stdio.h>
void main()
{int i,n,m,s; long t;
 printf("请输入进站人数 n：");
 scanf("%d",&n);
 t=time()%1000;srand(t);                    /*  随机数发生器初始化 */
 printf("%d 人进站所需时间约为:",n);
 for(t=0,i=1;i<=n;i++)
     t=t+rand()%61+20;                      /* 计算进站时间总和 */
 m=t/600;
 s=(t%600)/10;                              /* 转化为分秒输出 */
 printf("%d 分%d 秒.\n",m,s);
}
```

3. 运行示例与说明

运行程序，输入 *n*=300，得

300 人进站所需时间约为：25 分 4 秒。

再运行一次，输入 *n*=300，所得进站时间又变了（由于随机数发生器进行了初始化），这才是正常的。正像昨天进站 300 人所用的时间与今天 300 人所用的时间不一定相等一个道理。

若需显示进站的每一个人所用的进站时间，可在循环体中加入一个输出表示秒的随机数 (rand()%61+20)/10 的语句即可。

7.3.2　蒙特卡罗模拟计算

1. 蒙特卡罗法计算定积分的基本思想

用蒙特卡罗法计算定积分

$$s=\int_a^b f(x)dx$$

其中 $a<b,0<f(x)<d,x\in[a,b],d\geqslant\max[f(x)]$，如图 7.1 所示。

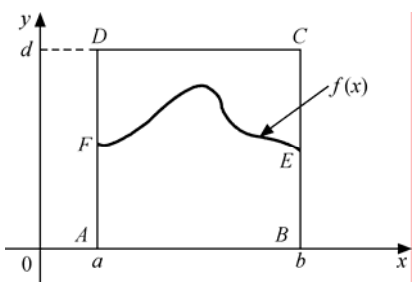


图 7.1　计算定积分示意图

产生 *n*（*n* 足够大）个随机分布在长方形 *ABCD* 上

的随机点(x_i, y_i)，其中x_i是随机分布在[a,b]上的随机数，y_i是随机分布在[0,d]上的随机数。设其中落在曲边梯形*ABEF*上的随机点数为m，则曲边梯形*ABEF*的面积即定积分s的值为

$$s = \frac{m}{n}(b-a)d$$

2. 蒙特卡罗算法描述

```
输入正整数 n;
m=0;
for(k=1;k<=n;k++)
    {x=a+(b−a)∗rand()        /∗ rand()为（0,1）区间内的随机数 ∗/
     y=d∗rand()
     if(y<f(x))   m=m+1;
     }
s=m∗(b−a)∗d/n;
输出定积分值 s;
```

3. 蒙特卡罗模拟计算举例

【例 7.7】 计算定积分

$$s = \int_0^3 x^2\sqrt{1+x}dx$$

解：用蒙特卡罗法计算定积分程序设计

```
#include <stdio.h>
#include <math.h>
void main()
{ long int m,n,k,t;
   float a,b,d,s,x,y;
   printf("请输入 n：");                    /∗ 输入试验次数 ∗/
   scanf("%ld",&n);
   printf("请输入 a,b：");                  /∗ 输入积分的上下限 ∗/
   scanf("%f,%f",&a,&b);
   t=time()%1000;srand(t);                  /∗  随机数发生器初始化 ∗/
   m=0; d=0;
   for(x=a;x<=b;x=x+0.01)
      if(d<x∗x∗sqrt(1+x))
         d=x∗x∗sqrt(1+x);                   /∗ 计算纵坐标最大值 d ∗/
for(k=1;k<=n;k++)
      {x=a+(b−a)∗(rand()%1000/1000.0);
       y=d∗(rand()%1000/1000.0);
       if(y<x∗x∗sqrt(1+x))   m=m+1;         /∗ 随机点在曲边梯形内 m 增 1 ∗/
```

```
        }
    s=m*(b-a)*d/n;                                  /* 计算曲边梯形的面积 */
    printf("所求定积分 s=%7.4f \n",s);
     }
```

运行程序，输入 n=100000，得

```
请输入 a,b：0,3
所求定积分 s=16.1698
```

7.3.3　模拟发扑克牌

玩扑克牌是人们喜爱的文娱活动，常见的有桥牌，升级等不同玩法。通过程序设计模拟发扑克牌是随机模拟的有趣课题。

【例 7.8】 模拟扑克升级发牌，把含有大小王的共 54 张牌随机分发给 4 家，每家 12 张，底牌保留 6 张。

1．模拟算法设计

（1）模拟花色与点数

模拟发牌必须注意随机性。所发的一张牌是草花还是红心，是随机的；是 5 点还是 J 点，也是随机的。

同时要注意不可重复性。如果在一局的发牌中出现两个黑桃 K 就是笑话了。 同时局与局之间必须作到互不相同，如果某两局牌雷同，也不符合发牌要求。

为此，对应 4 种花色，设置随机整数 x，对应取值为 1～4。对应每种花色的 13 点，设置随机整数 y，对应取值为 1～13。为避免重复，把 x 与 y 组合为三位数：$z=x*100+y$，并存放在数组 $m(54)$中。发第 $i+1$ 张牌，产生一个 x 与 y，得一个三位数 z，数 z 与已有的 i 个数组元素 $m(0),m(1),\cdots,m(i-1)$逐一进行比较，若不相同则打印与 x,y 对应的牌（相当于发一张牌）后，然后赋值给 $m(i)$，作为以后发牌的比较之用。若有相同的，则重新产生随机整数 x 与 y 得 z，与 m 数组值进行比较。

（2）模拟大小王

注意到在升级扑克中有大小王，它的出现给程序设计带来一定的难度。大小王的出现也是随机的，为此，把随机整数 y 的取值放宽到 0～13，则 z 可能有 100，200，300，400。定义 z=200 时对应大王，z=100 时对应小王，同上作打印与赋值处理 。若 z=300 或 400，则返回重新产生 x 与 y。

（3）随机生成模拟描述

在已产生 i 张牌并存储在 m 数组中，产生第 $i+1$ 张牌的模拟算法：

```
for(j=1;j<=10000;j++)
      {x=rand()%4+1; y=rand()%14;                /* x 表花色，y 表点数 */
       z=x*100+y;
       if(z==300 || z==400) continue;
       t=0;
```

```
        for(k=0;k<=i-1;k++)
          if(z==m[k]) {t=1;break;}                    /* 与前产生的牌比较确保牌不重复 */
        if(t==0)
          {m[i]=z;break;}                             /* 产生的新牌赋值给 m(i) */
        }
```

（4）打印输出

打印直接应用 C 语言中 ASCII 码 1～6 的字符显示大小王与各花色。设置字符数组 *d*，打印点数时把 y=1、13、12、11 分别转化为 A、K、Q、J。

为实现真正的随机，根据时间的不同，设置 t=time()%10000;srand(t) 初始化随机数发生器，从而达到真正随机的目的。

2. 发扑克牌 C 程序实现

```
/* 发扑克升级牌,有大小王,4 个人每人 12 张牌,底牌 6 张. */
#include <stdio.h>
void main()
{int   x,y,z,t,i,j,k,m[55];
 char d[14]=" A234567891JQK";
 printf("\n      E          S          W          N \n");
 t=time()%1000;srand(t);                              /*  随机数发生器初始化 */
 m[0]=0;
 for(i=1;i<=54;i++)
   {if(i==49)   printf("bottom: \n");
    for(j=1;j<=10000;j++)
      {x=rand()%4+1; y=rand()%14;
       z=x*100+y;
       if(z==300 || z==400) continue;
       t=0;
       for(k=0;k<=i-1;k++)
         if(z==m[k]) {t=1;break;}                     /* 确保牌不重复 */
       if(t==0)
         {m[i]=z;break;}
       }
   if(z==100 || z==200) printf("      %c     ",x);
   else if(y==10) printf("     %c10   ",x+2);
   else printf("     %c%c     ",x+2,d[y]);
   if(i%4==0) printf("\n");
   }
 printf("\n");
}
```

3. 程序运行示例

运行程序，随机得一副升级牌如图 7.2 所示。

图 7.2　一副升级扑克牌

7.4　操作过程模拟

过程模拟，包括各类事件操作过程的模拟与变化过程的模拟，内容比较繁杂，模拟方法也没有统一模式。

7.4.1　洗牌

【例 7.9】 给你 $2n$ 张牌，编号为 $1,2,3,\cdots n,n+1,\cdots,2n$，这也是最初牌的顺序。一次洗牌是把序列变为 $n+1,1,n+2,2,n+3,3,n+4,4,\cdots,2n,n$。可以证明，对于任意自然数 n，都可以在经过 m 次洗牌后重新得到初始的顺序。

编程对于小于 10 000 的自然数 n（n 从键盘输入）的洗牌，求出重新得到初始顺序的洗牌次数 m 的值，并显示洗牌过程。

1. 过程模拟设计

设洗牌前位置 k 的编号为 $p(k)$，洗牌后位置 k 的编号变为 $b(k)$。

我们寻求与确定洗牌前后牌的顺序改变规律。

前 n 个位置的编号赋值变化：位置 1 的编号赋给位置 2，位置 2 的编号赋给位置 4，……，位置 n 的编号赋给位置 $2n$。即 $b(2k)=p(k)(k=1,2,\cdots,n)$。

后 n 个位置的编号赋值变化：位置 $n+1$ 的编号赋给位置 1，位置 $n+2$ 的编号赋给位置 3，……，位置 $2n$ 的编号赋给位置 $2n-1$。即 $b(2k-1)=p(n+k)(k=1,2,\cdots,n)$。

约定洗牌 10 000 次（可增减），设置 m 循环，在 m 循环中实施洗牌，每次洗牌后检测是否得到初始的顺序。

2. 模拟洗牌过程程序实现

```
#include<stdio.h>
void main()
{int k,n,m,y,p[10000],b[10000];
printf("\n n=");scanf("%d",&n);
printf("\n     ");
for(k=1;k<=2*n;k++)                          /* 最初牌的顺序 */
     {p[k]=k; printf("%d   ",p[k]);}
for(m=1;m<=20000;m++)
   {y=0;
    for(k=1;k<=n;k++)                        /* 实施一次洗牌 */
        {b[2*k]=p[k];
         b[2*k-1]=p[n+k];}
    for(k=1;k<=2*n;k++)
           p[k]=b[k];
    printf("\n%d: ",m);                      /* 打印第 m 次洗牌后的结果 */
    for(k=1;k<=2*n;k++)
        printf("%d   ",p[k]);
    for(k=1;k<=2*n;k++)                      /* 检测是否回到初始的顺序 */
        if(p[k]!=k) y=1;
    if(y==0)
        {printf("\n m=%d\n",m);break;}       /* 输出回到初始的洗牌次数 */
    }
}
```

3. 程序运行示例

运行程序，n=8 的洗牌过程如下：

```
   1   2   3   4   5   6   7   8   9   10   11   12   13   14   15   16
1: 9   1   10   2   11   3   12   4   13   5   14   6   15   7   16   8
2: 13   9   5   1   14   10   6   2   15   11   7   3   16   12   8   4
3: 15   13   11   9   7   5   3   1   16   14   12   10   8   6   4   2
4: 16   15   14   13   12   11   10   9   8   7   6   5   4   3   2   1
5: 8   16   7   15   6   14   5   13   4   12   3   11   2   10   1   9
6: 4   8   12   16   3   7   11   15   2   6   10   14   1   5   9   13
7: 2   4   6   8   10   12   14   16   1   3   5   7   9   11   13   15
8: 1   2   3   4   5   6   7   8   9   10   11   12   13   14   15   16
m=8
```

7.4.2　泊松分酒

【例 7.10】　法国数学家泊松（Poisson）曾提出以下分酒趣题：某人有一瓶 12 品脱(容量单位)的酒，同时有容积为 5 品脱与 8 品脱的空杯各一个。借助这两个空杯，如何将这瓶 12 品脱的酒平分?

我们要解决一般的平分酒问题：借助容量分别为 b_v 与 c_v（单位为整数）的两个空杯，用最少的分倒次数把总容量为偶数 a 的酒平分。这里正整数 b_v，c_v 与偶数 a 均从键盘输入。

1. 模拟算法设计

求解一般的“泊松分酒”问题：借助容积分别为整数 b_v,c_v 的两个空杯，用最少的分倒次数把总容量为偶数 a 的酒（并未要求满瓶）平分，采用直接模拟平分过程的分倒操作。

为了把键盘输入的偶数 a 通过分倒操作平分为两个 i: $i=a/2$（i 为全局变量），设在分倒过程中：

瓶 A 中的酒量为 a,$(0\leqslant a\leqslant 2*i)$;

杯 B(容积为 b_v)中的酒量为 b,$(0\leqslant b\leqslant b_v)$;

杯 C(容积为 c_v)中的酒量为 c,$(0\leqslant c\leqslant c_v)$;

我们模拟下面两种循环分倒操作:

（1）按 $A->B->C$ 顺序操作

① 当 B 杯空(b=0)时，从 A 瓶倒满 B 杯。

② 从 B 杯分一次或多次倒满 C 杯。

$b>c_v-c$，倒满 C 杯，操作③

$b\leqslant c_v-c$,倒空 B 杯，操作①

③ 当 C 杯满($c=c_v$)时，从 C 杯倒回 A 瓶。

分倒操作中，用变量 n 统计分倒次数，每分倒一次，n 增 1。

若 b=0 且 $a<b_v$ 时，步骤①无法实现（即 A 瓶的酒倒不满 B 杯）而中断，记 n=−1 为中断标志。

分倒操作中若有 $a=i$ 或 $b=i$ 或 $c=i$ 时，显然已达到平分目的，分倒循环结束， 用试验函数 $Probe(a,b_v,c_v)$返回分倒次数 n 的值。否则，继续循环操作。

模拟操作描述：

```
while (!(a==i || b==i || c==i))
  {if(!b) {a-=bv;b=bv;}                    /* 从 A 瓶倒满 B 杯 */
   else if (c==cv) {a+=cv;c=0;}            /* 从 C 杯倒回 A 瓶 */
   else if (b>cv-c) {b-=(cv-c);c=cv;}      /* 从 B 倒满 C 杯 */
   else {c+=b;b=0;}                        /* 从 B 倒 C，倒空 B 杯 */
   printf("%6d%6d%6d\n",a,b,c);
  }
```

（2）按 $A->C->B$ 顺序操作

这一循环操作与（1）实质上是 C 与 B 杯互换， 相当于返回函数值 $Probe(a,c_v,b_v)$。

试验函数 $Probe()$的引入是巧妙的，可综合摸拟以上两种分倒操作避免了关于 c_v 与 b_v 大

小关系的讨论。

同时设计实施函数 *Practice*(a,b_v,c_v)，与试验函数相比较，把 n 增 1 操作改变为输出中间过程量 a、b、c，以标明具体分倒操作进程。

在主函数 *main*()中，分别输入 a、b_v、c_v 的值后，为寻求较少的分倒次数，调用试验函数并比较 m_1=*Probe*(a,b_v,c_v)与 m_2=*Probe*(a,c_v,b_v);

若 m_1<0 或 m_2<0，表明无法平分（均为中断标志）。

若 m_2<0，只能按上述（1）操作；若 0<m_1<m_2，按上述（1）操作分倒次数较少（即 m_1）。此时调用实施函数 *Practivce*(a,b_v,c_v)。

若 m_1<0，只能按上述（2）操作；若 0<m_2<m_1，按上述（2）操作分倒次数较少（即 m_2）。此时调用实施函数 *Practice*(a,c_v,b_v)。

实施函数打印整个模拟分倒操作进程中的 a、b、c。最后打印出最少的分倒次数。

2. 泊松分酒的 C 程序实现

```
/* 泊松分酒模拟操作 */
int i,n;
#include<stdio.h>
void main()
{ int a,bv,cv,m1,m2;
printf("\n 请输入酒总量(偶数):");
scanf("%d",&a);
printf("两空杯容量 bv,cv 分别为:");
scanf("%d,%d",&bv,&cv);
i=a/2; m1=probo(a,bv,cv); m2=probo(a,cv,bv);
if (m1<0 && m2<0)
    printf("无法平分!");
if (m1>0 && (m2<0 || m1<m2))
    { n=m1;practice(a,bv,cv);}
if (m2>0 && (m1<0 || m2<m1))
    { n=m2;practice(a,cv,bv);}
}
#include<stdio.h>
practice(a,bv,cv)                    /* 模拟实施函数 */
int a,bv,cv;
{ int b=0,c=0;
printf("平分酒的分法:\n");
printf("酒瓶%d 空杯%d 空杯%d\n",a,bv,cv);
printf("%6d%6d%6d\n",a,b,c);
while (!(a==i || b==i || c==i))
   { if(!b) {a-=bv;b=bv;}
```

```
        else if(c==cv) {a+=cv;c=0;}
          else if(b>cv-c) {b-=(cv-c);c=cv;}
            else {c+=b;b=0;}
          printf("%6d%6d%6d\n",a,b,c);
       }
    printf("平分酒共分倒%d 次.\n",n);
    }
    int probo(a,bv,cv)                          /* 试验函数 */
    int a,bv,cv;
    { int n=0,b=0,c=0;
    while (!(a==i || b==i || c==i))
      { if(!b)
            if(a<bv) {n=-1;break;}
            else { a-=bv;b=bv;}
      else if(c==cv) {a+=cv;c=0;}
          else if(b>cv-c) {b-=(cv-c);c=cv;}
              else {c+=b;b=0;}
        n++;
      }
    return(n);
    }
```

3. 程序运行示例

```
请输入酒总量（偶数）：12
两空杯容量 bv,cv 分别为：5,8
平分酒的分法：
酒瓶 12  空杯 8  空杯 5
    12      0      0
     4      8      0
     4      3      5
     9      3      0
     9      0      3
     1      8      3
     1      6      5
平分酒共分倒 6 次。
请输入酒总量（偶数）：20
两空杯容量 bv,cv 分别为：12,7
平分酒的分法：
酒瓶 20 空杯 12 空杯 7
```

```
20    0    0
 8   12    0
 8    5    7
15    5    0
15    0    5
 3   12    5
 3   10    7
```

平分酒共分倒 6 次。

7.4.3 模拟小孔流水

【例 7.11】 设计一流水演示装置，要求：

（1）有抽水系统，把水从下容器抽到装有高低不同两流水孔的圆柱体桶中；

（2）水抽满后，同时打开两流水孔，水从两流水孔以不同流速流出，两束水流射程与水位差相关；

（3）随水在桶中水位的降低，水流射程相应缩短，直至水流完，水束射程变化与桶中水位关系应满足实际规律；

（4）以上演示重复两次。

1. 模拟算法设计

模拟流水演示程序主要应用以下作图函数:

rectangle(x1,y1,x2,y2);	画对角顶点(*x*1,*y*1),(*x*2,*y*2)的矩形。
bar(x1,y1,x2,y2);	画对角顶点(*x*1,*y*1),(*x*2,*y*2)的矩形并填充。
putpixel(x,y,c);	按颜色 *c* 为点(*x*,*y*)着色。
line(x1,y1,x2,y2);	从点(*x*1,*y*1)至点(*x*2,*y*2)画直线段。
circle(x,y,r);	以(*x*,*y*)为圆心 *r* 为半径画圆。
setfillstyle(SOLID_FILL,3);	全部着色填充模式。

首先用画矩形显示水槽装置。抽水时由下至上匀速画直线段表示水位升高。流水时由上至下由快至慢按背景色画直线段（相当于消除线段）表示水位降低。

关键是画流水水束，水束是有规律地动的。水流的出口流速与水位差的平方根成正比，水束的射程与出口流速成正比。依此设计水束的点的坐标并显示点，随时间水点消失，重新显示新的水点，这是模拟的关键所在。由于更新速度较快，效果看起来为连续变化。

模拟水束位置控制：

```
for(y=60;y<=347;y++)
for(j=0;j<=264;j++)
  {d=sqrt(204−y);d1=2∗sqrt(j)∗d;d2=2∗sqrt(j+1)∗d;
                                        /∗d,d1,d2 控制水点位∗/
  x1=260+d1;y1=204+j;d=d2−d1+(204−y)/70;
  for(i=0;i<=d;i++)
```

```
    if(x1+i<618 && y1<490−y/3) putpixel(x1+i,y1,7);
}
```

显示水束后，用 putpixel(x,y,0)予以清除。

2. 流水演示动画 C 程序实现

```
/*  小孔流水演示 */
#include <graphics.h>
#include <math.h>
main()
{ float d,d1,d2;
   int i,j,k,t,u,x,y,x1,y1;
   int graphdriver=DETECT,graphmove;
   initgraph(&graphdriver,&graphmove,"");                    /* 初始化 */
cleardevice();
setactivepage(0);setvisualpage(0);
rectangle(50,30,248,348);rectangle(48,30,250,350);           /* 画矩形 */
bar(30,25,77,35);bar(63,25,77,60);                           /* 画矩形并填充 */
bar(30,25,40,472);bar(40,462,50,472);
bar(248,202,260,206);
rectangle(255,202,258,194);bar(253,194,260,190);
bar(248,344,260,348);
rectangle(255,336,258,344);bar(253,336,260,332);
setfillstyle(SOLID_FILL,4);bar(2,351,250,370);
line(50,374,50,472);line(50,472,620,472);                    /* 画直线段 */
line(620,472,620,374);
setfillstyle(SOLID_FILL,3); pieslice(55,15,0,360,15);
 for(k=1;k<=2;k++)                                           /* 控制演示 2 次 */
   {setfillstyle(SOLID_FILL,3);
   bar(51,378,618,471);t=getch();bar(67,60,72,347);
   for (y=347;y>=60;y−−)
   { pieslice(55,15,0,−y*40,15);
   for(x=51;x<=247;x++) putpixel(x,y,3);                     /* 按颜色 3 画点,连线 */
   for(x=51;x<=619;x++) putpixel(x,375+(350−y)/3,0);
   for(i=1;i<=20;i++)
   for(t=1;t<=1000;t++);}
   t=getch();
   for (y=60;y<=347;y++)
   { for(x=51;x<=247;x++) putpixel(x,y,0);
   if (y<=203)
```

```
    for(j=0;j<=264;j++)
    { d=sqrt(204−y);d1=2∗sqrt(j)∗d;d2=2∗sqrt(j+1)∗d;   /∗d,d1,d2 控制点位∗/
    x1=260+d1;y1=204+j;d=d2−d1+(204−y)/70;
    for(i=0;i<=d;i++)
    if(x1+i<618 && y1<490−y/3) putpixel(x1+i,y1,7);}
    for(j=0;j<=120;j++)
    { d=sqrt(348−y);d1=2∗sqrt(j)∗d;d2=2∗sqrt(j+1)∗d; /∗d,d1,d2 控制点位∗/
    x1=260+d1;y1=346+j;d=d2−d1+(348−y)/70;
    for(i=0;i<=d;i++)
    if (y1<490−y/3 && x1+i<618) putpixel(x1+i,y1,7);}
    if (y%3==0)
    {for(x=51;x<=619;x++) putpixel(x,491−y/3,3);}
    u=1;   if (y>203) u=20;
    for(i=1;i<=20;i++)
    for(t=1;t<=u∗sqrt(y−58);t++);
    setfillstyle(EMPTY_FILL,0);bar(260,200,618,490−y/3);}}
  getch();
  closegraph();
  }
```

运行程序，显示各水槽装置，开始抽水；水满后由安装在水槽侧面的高低不同的两个流水孔流出，水束的射程与水位差相关。整个演示的节凑由操作者按任意键决定。

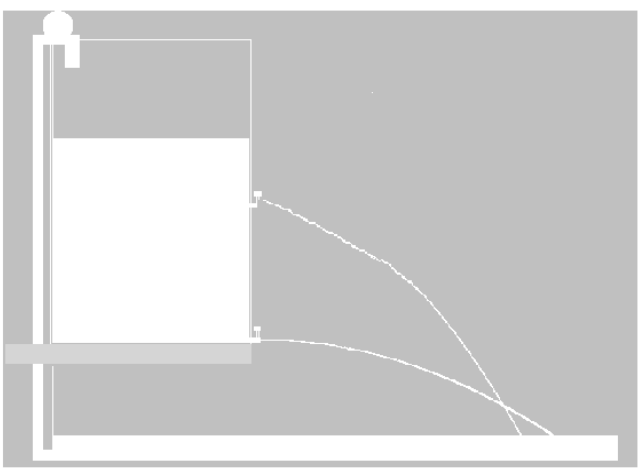


图 7.3　小孔流水演示

7.5　模拟外索夫游戏

【例 7.12】 外索夫（Wythoff）游戏的参与双方 *A*,*B* 交替地从已有的两堆石子中按下面的规则取石子：可以从某一堆取出若干个石子，数量不限；也可以同时从两堆取石子，要求两堆取出的石子数相等。每次必须取石子，取出最后一粒石子者为胜。

设计程序，计算机为一方，操作者为另一方，模拟外索夫游戏过程。

1. 模拟游戏算法设计

怎样才能在游戏中取得胜利？对于任何一方，只要在他取完之后出现下面的局势（不妨称为胜势组，括号中的两个数字分别为两堆石子数），可导致胜利：

(1,2),(3,5),(4,7),(6,10),(8,13),…

“胜势组”的两个整数即著名的 Wythoff 对，其构成规律是:第一组为(1,2)；第 $i(i>1)$组中的较小数 $c(i)$为与前 $i-1$ 组中所有已有数不同的最小正整数，第 i 组中的较大数 $d(i)=c(i)+i$（即第 i 组的两堆石子数之差为 i）。

胜势的证明:当 *A* 方取子之后出现(1,2)，*B* 方必须按规则取子。*B* 若取完某一堆，则 A 取完另一堆胜。否则，*B* 若在“2 枚”堆中取 1 枚，剩下(1,1),*A* 可取完胜。即不管 *B* 如何取子，总会导致 *A* 胜。

一般地，当 *A* 方取子之后出现胜势中的第 i 组$(c(i),d(i))(i>1,d(i)=c(i)+i)$，无论 *B* 如何取石子，*A* 可胜或下调为一个较小的胜势组。

（1）对方 *B* 若取完某一组，*A* 取完另一组胜；或 *B* 取石子使得两堆石子数相同，*A* 则同时取完两堆胜。

（2）对方 *B* 若在胜势组（$c(i),d(i)$）取子，该两堆剩下的石子数为 m、n、$e=|m-n|$，而 $m>c(e),n>d(e)$，则 *A* 在两堆同时取$(m+n-c(e)-d(e))/2$，即下调为第 e 组胜势组。

（3）对方 *B* 若在胜势组（$c(i),d(i)$）取子，该两堆剩下的石子数为 m,n，其中一堆为第 j 个胜势组$(j<i)$中的一个元素，而另一堆大于第 j 个胜势组中的另一个元素，则在另一堆取子后下调为第 j 个胜势组。

上述胜势的证明即游戏取胜的操作要领。

一般地，对操作者取石子只要作错误提示即可。计算机取石子时，若遇上胜势组，随意取石子应付，等待时机；其他情形则按上述策略下调为胜势组，一步步导致胜利。

计算机面对两堆数为 m、n 的取子策略描述：

```
if(m==0 && n>0) {m2=0;n2=n;}
else if(n==0 && m>0){n2=0;m2=m;}
else if(m==n){m2=m;n2=n;}
else if(m==c[e] && n==d[e] || n==c[e] && m==d[e])
        {m2=1;n2=0;}                    /* 面临胜势组，随意取 */
else if(m>c[e] && n>c[e])
        {m2=(m+n-c[e]-d[e])/2;n2=m2;}
```

```
    else
      {k=1;
       while(k<=e)
        {if(m==c[k] && n>d[k]){m2=0;n2=n−d[k];break;}
         if(n==c[k] && m>d[k]){n2=0;m2=m−d[k];break;}
         if(m==d[k] && n>c[k]){m2=0;n2=n−c[k];break;}
         if(n==d[k] && m>c[k]){n2=0;m2=m−c[k];break;}
         k=k+1;
        }
```

为书写简便，记原始两堆石子数用{}括起来，计算机取后两堆石子数用[]括起来，操作者取后两堆石子数用()括起来。

一般来说，先行者有利，他可在第一次取子时转换为胜势组，因而胜率较大。 但是也不尽然，若游戏开始时随机产生的两堆石子数即为某一胜势组，先行者不利。

2. 外索夫游戏 C 程序实现

```
/*    外索夫游戏 */
#include <math.h>
#include<stdio.h>
void main()
{int m,n,m1,n1,m2,n2,i,j,k,e,zs,t,c[300],d[300];
 c[1]=1; d[1]=2;
 for(i=2;i<300;i++)                    /* 计算胜势组待用 */
 {for(k=c[i−1]+1;k<=1000;k++)
   {t=0;
    for(j=1;j<=i−1;j++)
        if(k==d[j]){t=1;j=i;}
    if(t==0){c[i]=k;d[i]=k+i;k=1000;}
   }
 }
 printf("正在作游戏准备,请按任意键开始.\n");
 t=time()%1000;srand(t);                    /*   随机数发生器初始化 */
 m=rand()%100+10;n=m+rand()%50;             /* 随机产生两堆石子数 */
 printf("第一堆石子数:%d          第二堆石子数:%d \n",m,n);
   t=getch();
   if(t%2==0) {zs=1;printf( "计算机猜得先取.\n");}
   else {zs=0;printf( "操作者猜得先取.\n");}
   printf("{%d,%d}",m,n);
   i=1;
   while (m>0 || n>0)
```

```
    {if(zs==0 || i>1)
        {printf("操作者取石子!\n");
         printf("操作者在第一堆取:");scanf("%d",&m1);
         printf("操作者在第二堆取:");scanf("%d",&n1);
         if(m1==0 && n1==0) {printf("必须取石子!");continue;}
         if(m1>m || n1>n) {printf("石子不够,重取!");continue;}
         if(m1>0 && n1>0 && m1!=n1) {printf("违规,重取!");continue;}
         m=m-m1;n=n-n1;
         printf("->(%d,%d)\n",m,n);
         if(m==0 && n==0) {printf("全取完,操作者胜!祝贺您!");break;}
        }
        {printf("计算机取石子!");e=abs(m-n); /* 计算机猜得先取开始处 */
         if(m==0 && n>0) {m2=0;n2=n;}
         else if(n==0 && m>0){n2=0;m2=m;}
         else if(m==n){m2=m;n2=n;}
         else if(m==c[e] && n==d[e] || n==c[e] && m==d[e]){m2=1;n2=0;}
         else if(m>c[e] && n>c[e]){m2=(m+n-c[e]-d[e])/2;n2=m2;}
         else
           {k=1;
            while(k<=e)
              {if(m==c[k] && n>d[k]){m2=0;n2=n-d[k];break;}
               if(n==c[k] && m>d[k]){n2=0;m2=m-d[k];break;}
               if(m==d[k] && n>c[k]){m2=0;n2=n-c[k];break;}
               if(n==d[k] && m>c[k]){n2=0;m2=m-c[k];break;}
               k=k+1;
              }
           }
         printf("计算机在第一堆取:%d ",m2);
         printf("计算机在第二堆取:%d \n",n2);
         m=m-m2;n=n-n2;
         printf("->[%d,%d]\n",m,n);
         if(m==0 && n==0) {printf("全取完,计算机胜!");break;}
        }
      i=i+1;
    }
    printf("再见!欢迎下次再玩.\n");
}
```

运行程序，游戏简单进程为。

第一堆石子数:67　　第二堆石子数:90

{67,90}−>(67,30) −>[49,30] −>(9,30) −>[9,15] −>(2,8) −>[2,1] −>(1,0) − [0,0]

全取完，计算机胜!

再见!欢迎下次再玩。

习　　题

1. 从 1 开始按正整数的顺序不间断连续写下去所成的整数称为连写数。要使连写数 123456789101112…*m*（连写到整数 *m*）能被指定的整数 *p*（<1000）整除，*m* 至少为多大?

2. 应用蒙特卡罗法计算圆周率π。

3. 模拟发桥牌：桥牌共 52 张，无大小王。按 E、S、W、N 顺序把随机产生的 52 张牌分发给各方，每方 13 张。发完后，分类从大到小整理每方的牌。

4. 模拟巴什游戏：一堆石子有 *n* 粒，参与巴什（Bash）游戏的二人轮流从这堆石子中取石子，规定每次至少取一粒，最多取 *m* 粒（正整数 *m*,*n* 随机产生，*m*<*n*）。最后取完石子者得胜。

第 8 章 智 能 优 化

在工程实践中，经常会接触到一些比较“新颖”的算法或理论，比如模拟退火、遗传算法、粒子群算法、神经网络等。这些算法或理论都有一些共同的特性，就是模拟自然过程，通称为“智能算法”。智能优化算法要解决的一般是最优化问题。最优化问题可以分为以下两类：

（1）求解一个函数中，使得函数值最小的自变量取值的函数优化问题。

（2）在一个解空间里面，寻找最优解，使目标函数值最小的组合优化问题。

本章将初步地介绍模拟退火算法、遗传算法、粒子群算法、神经网络优化算法等智能优化算法的基本理论和实现技术以及简单应用，并从结构上对算法进行描述。

8.1 模拟退火算法

8.1.1 物理退火过程和 Metropolis 准则

从物理学的有关知识知道，固体在恒定的温度下达到热平衡的过程可以用 Monte Carlo 法进行模拟，Monte Carlo 方法的算法简洁，但必须大量采样才能得到比较准确的结果，因而计算量大。基于物理系统在自由状态下有能量向较低状态转移的趋势，而热运动有妨碍它准确落入最低状态的考虑，Metropolis 于 1953 年提出对有重要贡献的状态进行采样，从而较快地达到比较理想的结果，其方法可以描述如下。

先给定粒子相对位置表征的初始状态 i，作为固体的当前状态，该状态的能量为 R_i。然后用摄动装置使随机选取的某个粒子的位移产生一微小变化，得到一个新状态 j，新状态的能量是 E_j。如果 $E_j<E_i$，则该新状态就可以作为“重要”状态，如果 $E_j>E_i$，则考虑到热运动的影响，该新状态是否为“重要”状态需要根据固体处于该状态的概率来判断。设固体处于状态 i 与状态 j 的概率比值为 r，r 是一个小于 1 的数，用随机数产生器产生一个[0,1)区间的随机数 ξ，若 r> ξ，则新状态 j 作为重要状态，否则舍去。

若新状态 j 使重要状态，就以 j 取代 i 成为当前状态，否则仍以 i 为当前状态，再重复以上新状态的产生过程，在固体状态经过大量变换（常称之为迁移）后，系统趋于能量较低的

平衡状态。

上述接受新状态的准则成为 Metropolis 准则，相应的算法成为 Metropolis 算法。研究表明，一般情况下 Metropolis 算法计算量明显少于 Minte Carlo 算法。

通过对固体退火过程的研究可知，高温状态下的物质降温时其内能随之下降，如果降温过程充分缓慢，则在降温过程中物质体系始终处于平衡状态。从而降到某一低温时其内能可达最小，称这种降温为退火过程，模拟退火过程的寻优方法称为模拟退火（Simulated Annealing，SA）算法。

8.1.2 模拟退火算法概述

1. 模拟退火的基本思想

模拟退火算法可以分解为解空间、目标函数和初始解三部分。

解空间：它为问题的所有可能（可行的或包括不可行的）解的集合，它限定了初始解选取和新解产生时的范围。对无约束的优化问题，任一可能解即为一可行解，因此解空间就是所有可行解的集合；而在许多组合优化问题中，一个解除满足目标函数最优的要求外，还必须满足一组约束，因此在解集中可能包含一些不可行解。为此，可以限定解空间仅为所有可行解的集合，即在构造解时就考虑到对解的约束；也可允许解空间包含不可行解，而在目标函数中加上所谓罚函数以“惩罚”不可行解的出现。

目标函数：它是对问题的优化目标的数学描述，通常表述为若干优化目标的一个和式。目标函数的选取必须正确体现对问题的整体优化要求。例如，如上所述，当解空间包含不可行解时，目标函数中应包含对不可行解的罚函数项，借此将一个有约束的优化问题转化为无约束的优化问题。一般地，目标函数值不一定就是问题的优化目标值，但其对应关系应是明显的。此外，目标函数式应当是易于计算的，这将有利于在优化过程中简化目标函数差的计算以提高算法的效率。

初始解：是算法迭代的起点，试验表明，模拟退火算法是鲁棒的（Robust），即最终解的求得几乎不依赖于初始解的选取。算法对应演示图如图 8.1 所示。

其中模拟退火算法新解的产生和接受可分为如下四个步骤。

第一步是由一个产生函数从当前解产生一个位于解空间的新解；为便于后续的计算和接受，减少算法耗时，通常选择由当前新解经过简单地变换即可产生新解的方法，如对构成新解的全部或部分元素进行置换、互换等，注意到产生新解的变换方法决定了当前新解的邻域结构，因而对冷却进度表的选取有一定的影响。

第二步是计算与新解所对应的目标函数差。因为目标函数差仅由变换部分产生，所以目标函数差的计算最好按增量计算。事实表明，对大多数应用而言，这是计算目标函数差的最快方法。

第三步是判断新解是否被接受,判断的依据是一个接受准则，最常用的接受准则是 Metropo1is 准则：若 $\Delta t'<0$ 则接受 S'作为新的当前解 S，否则以概率 $exp(-\Delta t'/T)$接受 S'作为新的当前解 S。

第四步是当新解被确定接受时，用新解代替当前解，这只需将当前解中对应于产生新

解时的变换部分予以实现，同时修正目标函数值即可。此时，当前解实现了一次迭代，可在此基础上开始下一轮试验。而当新解被判定为舍弃时，则在原当前解的基础上继续下一轮试验。

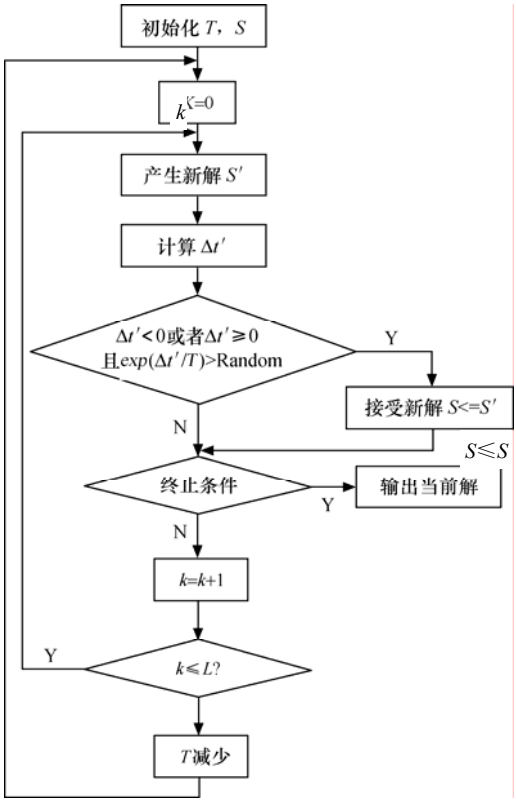


图 8.1 模拟退火算法流程图

模拟退火算法与初始值无关，算法求得的解与初始解状态 S（是算法迭代的起点）无关；模拟退火算法具有渐近收敛性，已在理论上被证明是一种以概率 1 收敛于全局最优解的全局优化算法；模拟退火算法具有并行性。

模拟退火算法简单步骤描述如下。

（1）初始化：初始温度 T（充分大），初始解状态 S（是算法迭代的起点），每个 T 值的迭代次数 L。

（2）对 $k=1,\cdots,L$ 做第（3）步至第（6）步。

（3）产生新解 S'。

（4）计算增量 $\Delta t'=C(S')-C(S)$，其中 $C(S)$为评价函数。

（5）若 $\Delta t'<0$ 则接受 S'作为新的当前解，否则以概率 $exp(-\Delta t'/T)$接受 S'作为新的当前解。

（6）如果满足终止条件则输出当前解作为最优解，结束程序。终止条件通常取为连续若干个新解都没有被接受时终止算法。

（7）T 逐渐减少，且 $T->0$，然后转第 2 步。

2. 模拟退火算法的主要问题

模拟退火算法的应用很广泛，但其参数难以控制，其主要问题有以下三点。

（1）温度 T 的初始值设置问题。温度 T 的初始值设置是影响模拟退火算法全局搜索性能的重要因素之一。初始温度高，则搜索到全局最优解的可能性大，但因此要花费大量的计算时间；反之，则可节约计算时间，但全局搜索性能可能受到影响。实际应用过程中，初始温度一般需要依据实验结果进行若干次调整。

（2）退火速度问题。模拟退火算法的全局搜索性能也与退火速度密切相关。一般来说，同一温度下的“充分”搜索（退火）是相当必要的，但这需要计算时间。实际应用中，要针对具体问题的性质和特征设置合理的退火平衡条件。

（3）温度管理问题。温度管理问题也是模拟退火算法难以处理的问题之一。实际应用中，由于必须考虑计算复杂度的切实可行性等问题，常采用如式 8.1 所示的降温方式：

$$T(t+1)=k\times T(t) \tag{8.1}$$

式 8.1 中 k 为正的略小于 1.00 的常数，t 为降温的次数。

8.1.3 应用举例

模拟退火算法应用，讨论货郎担问题（Travelling Salesman Problem，TSP 问题）：设有 n 个城市，用数字 $1,\cdots,n$ 代表。城市 i 和城市 j 之间的距离为 $d(i,j)$，其中 $i,j=1,\cdots,n$。TSP 问题是要找遍访每个域市恰好一次的一条回路，且其路径总长度为最短。

求解 TSP 的模拟退火算法模型可描述如下。

（1）解空间。解空间 S 是遍访每个城市恰好一次的所有回路，是$\{1,\cdots,n\}$的所有循环排列的集合，S 中的成员记为$(w_1,w_2,\cdots,w_n)$，并记 $w_{n+1}=w_1$。初始解可选为$(1,\cdots,n)$；

（2）目标函数。此时的目标函数即为访问所有城市的路径总长度或称为代价函数：

$$f(w_1,w_2,\cdots,w_n)=\sum_{j=1}^{n}d(w_j,w_{j+1})$$

要求此代价函数的最小值。

（3）新解的产生。随机产生 $1\sim n$ 之间的两相异数 k 和 m，若 $k<m$，则将$(w_1,w_2,\cdots,w_k,w_{k+1},\cdots,w_m,\cdots,w_n)$变为：

$(w_1,w_2,\cdots,w_m,w_{m-1},\cdots,w_{k+1},w_k,\cdots,w_n)$

如果是 $k>m$，则将

$(w_1,w_2,\cdots,w_k,w_{k+1},\cdots,w_m,\cdots,w_n)$

变为：

$(w_m,w_{m-1},\cdots,w_1,w_{m+1},\cdots,w_{k-1},w_n,w_{n-1},\cdots,w_k)$

上述变换方法可简单说成是“逆转中间或者逆转两端”。

也可以采用其他的变换方法，有些变换有独特的优越性，有时也将它们交替使用，得到一种更好方法。

代价函数差 设将$(w_1,w_2,\cdots,w_n)$变换为$(u_1,u_2,\cdots,u_n)$，则代价函数差为：

$$\Delta f = f(u_1,u_2,\cdots,u_n) - f(w_1,w_2,\cdots,w_n)$$
$$= \sum_{j=1}^{n} d(u_j,u_{j+1}) - \sum_{j=1}^{n} d(w_j,w_{j+1})$$

根据上述分析，可写出用模拟退火算法求解 TSP 问题的伪程序：

```
Procedure TSPSA:
  begin
    init−of−T; { T 为初始温度}
    S = {1,⋯,n}; {S 为初始值}
    termination = false;
    while termination = false
 begin
        for i = 1 to L do
            begin
              generate(S′form S); {  从当前回路 S 产生新回路 S′}
                Δt:   = f(S′))−f(S);{f(S)为路径总长}
                IF(Δt<0) OR (EXP(−Δt/T)>Random−of−[0,1])
                S = S′;
                IF the−halt−condition−is−TRUE THEN
                termination = true;
            End;
          T_lower;
       End;
  End
```

模拟退火算法的应用很广泛，可以较高的效率求解最大截问题、0－1 背包问题、图着色问题、调度问题等。

8.2 遗传算法

遗传算法是一类模拟生物进化的智能优化算法，它是由 J.H.Holland 于 60 年代提出的。目前，遗传算法已成为进化计算研究的一个重要分支。与传统优化方法相比，遗传算法的优点是：群体搜索、不需要目标函数的导数、概率转移准则。

早在 20 世纪 50 年代就有将进化原理应用于计算机科学的努力，但缺乏一种普遍的编码方法，只能依赖于变异而非交配产生新的基因结构。50 年代末到 60 年代初，受一些生物学家用计算机对生物系统进行模拟的启发，Holland 开始应用模拟遗传算子研究适应性。1967 年在 Bagley 关于自适应下棋程序的论文中，他应用遗传算法搜索下棋游戏评价函数的参数集，并首次提出了遗传算法这一术语。1975 年 Holland 出版了遗传算法历史上的经典著作《自然和人工系统中的适应性》，系统阐述了遗传算法的基本理论和方法，并提出了模式定理，证

明在遗传算子选择、交叉和变异的作用下，具有低阶、短定义距以及平均适应度高于群体平均适应度的模式在子代中将以指数级增长，这里的模式是某一类字符串，其某些位置有相似性。同年，DeJong 完成了他的博士论文《遗传自适应系统的行为分析》，将 Holland 的模式理论与他的计算试验结合起来，进一步完善了选择、交叉和变异操作，提出了一些新的遗传操作技术。

8.2.1 生物的进化与遗传

我们先了解生物进化的基本过程，如图 8.2 所示。

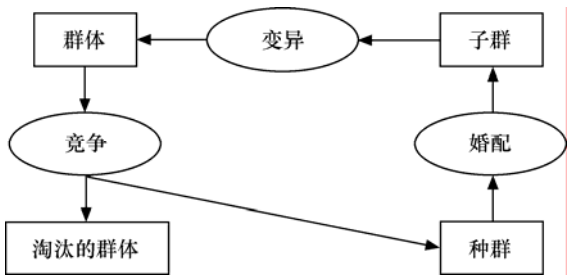


图 8.2　生物进化过程

以这个循环圈的生物群体为起点，经过竞争，一部分群体被淘汰而无法再进入这个循环圈，而另一部分则成为种群（reproduction）。优胜劣汰在这个过程中起着非常重要的作用，这在自然界尤为突出。因为自然天气的恶劣和天敌的侵害，大自然中的很多动物的成活率是非常低的，即使在成活群体中，还要通过竞争产生种群。种群通过婚配的作用产生子代群体（简称子群）。在进化的过程中，可能会因为变异产生新的个体。综合变异的作用，子群成长为新的群体而取代旧群体。在新的一个循环过程中，新的群体将替代旧群体而成为新循环的开始。

遗传算法是模拟达尔文的遗传选择和自然淘汰的生物进化过程的计算模型，也就是说，遗传算法主要借用生物进化中“适者生存”的规律。该规律揭示了大自然生物进化过程中的一个规律：最适合自然环境的群体往往产生了更大的后代群体。遗传算法是仿真生物遗传学和自然选择机理，通过人工方式所构造的一类搜索算法，从某种程度上说遗传算法是对生物进化过程进行的数学方式仿真，是一类借鉴生物界自然选择和自然遗传机制的随机化搜索方法，早在 20 世纪 60 年代初期就由美国大学的教授提出，并且在 1975 年教授发表了系统论述遗传算法的专著《自然与人工系统中的自适应》，其主要特点是群体搜索策略和群体中个体之间的信息交换，搜索不依赖于梯度信息。所以它的应用范围非常广泛，尤其适合于处理传统搜索方法难于解决的复杂和非线性问题，可广泛用于组合优化、机器学习、自适应控制、规划设计和人工生命等领域，从而确立了它在 21 世纪的智能计算技术中的关键地位。

8.2.2　遗传算法概述

遗传算法已有了许多发展，但一般来说，其基本过程是：首先采用某种编码方式将解空间映射到编码空间[①]，每个编码对应问题的一个解，称为染色体或个体。一般通过随机方法确定起始的一群个体，称为种群，在种群中根据适应值或某种竞争机制选择个体[②]，使用各种遗传操作算子（包括杂交、变异、倒位等）产生下一代（下一代可以完全替代原种群，即非重叠种群；也可以部分替代原种群中一些较差的个体，即重叠种群），如此进化下去，直到满足期望的终止条件。

从上面的原理可以看出，从搜索角度来讲，遗传算法具有许多独特的优点。即不必非常明确描述问题的全部特征，通用性和鲁棒性强，能很快适应问题和环境的变化；对领域知识依赖程度低，不受搜索空间限制性假设的约束，不必要求连续性、可导或单峰等。从多点进行搜索，如同在搜索空间上覆盖的一张网，搜索的全局性强，不易陷入局部最优；具有隐并行性，非常适合于并行计算。

在遗传算法的研究中，可以看到主要有三类研究方向。

（1）研究遗传算法本身的理论基础；

（2）用遗传算法作为工具解决工程问题，主要是进行优化，关心的是是否能在传统方法上有所提高；

（3）用遗传算法研究演化现象，一般涉及人工生命和复杂性科学领域。

在工程实践中的遗传算法应用主要是利用了其并行性和全局搜索的特点来进行优化。尽管遗传算法本身是一种通用弱方法，仍需要尽量结合特定领域的知识，实现解决特定问题的一个遗传算法特定实现，其范围可能更窄，但效果会更好。不过，从遗传算法的来源，即自然界现象看，生物演化的目的并非取得某一限制条件下的某些参数的最优，而是适应环境。生物进化的途径多种多样，没有哪一种是最优的，但是，成功的生物必然是适应其环境和环境内的其他生物的，对于环境的演化和其他生物的进化（它自己也在改变着环境，人类更是前所未有地改变着环境），它能够适应新的变化，继续生存。

遗传算法是模拟生物进化过程的计算模型。主要借鉴生物进化的一些特征，其主要特征表现为以下几方面。

（1）进化发展在解的编码上，这些编码按生物学的术语称为染色体。由于对解进行了编码，优化问题的一切性质都通过编码来研究。编码和解码是遗传算法的一个主题。

（2）自然选择规律决定哪些染色体产生超过平均数的后代。遗传算法中，通过优化问题的目标而人为地构造适应函数以达到好的染色体产生超过平均数的后代。

（3）当染色体结合时，双亲的遗传基因的结合使得子女保持父母的特征。

（4）当染色体结合时，随机的变异会造成子代同父代的不同。

---

① 可以是位串、实数、有序串、树或图，Holland 最初的遗传算法是基于二进制串的，类似于生物染色体结构，易于用生物遗传理论解释，各种遗传操作也易于实现。另外，可以证明，采用二进制编码式，算法处理的模式最多。但是，在具体问题中，直接采用解空间的形式进行编码，可以直接在解的表现型上进行遗传操作，从而易于引入特定领域的启发式信息，可以取得比二进制编码更高的效率。实数编码一般用于数值优化，有序串编码一般用于组合优化。

② 适应值就是解的满意程度，可以由外部显式适应度函数计算，也可以由系统本身产生，如由协同演化时不同对策的博奕确定，或者由个体在群体中的存活量和繁殖量确定。

遗传算法包括以下的主要处理步骤。

（1）对优化问题的解进行编码，称一个解的编码为一个染色体，组成编码的元素称为基因。编码的目的主要是用于优化问题解的表现形式和利于遗传算法中的计算。

（2）适应函数的构造和应用。适应函数基本上依据优化问题的目标函数而定。当函数确定之后，自然选择规律是以适应函数值的大小决定的概率分布来确定哪些染色体适应生存，哪些被淘汰。生存下来的染色体组成种群，形成一个可以繁衍下一代的群体。染色体的结合。双亲的遗传基因结合是通过编码之间的交配（crossover）达到下一代的产生。新一代的产生是一个生殖过程，它产生了一个新解。

（3）变异。新解产生过程中是可能发生基因变异，变异使某些解的编码发生变化，使解有更大的遍历性。

下面介绍编码方法和个体适应度函数以及遗传算法的基本步骤。

1. 编码方法

编码是应用遗传算法时要解决的首要问题，也是设计遗传算法时的一个关键步骤。编码方法除了决定个体的染色体排列式之外，它还决定个体从搜索空间的基因型变换到解空间的表现型时的解码方法，编码方法也影响到交叉算子、变异算子等遗传算子的运算方法。编码方法在很大程度上决定了如何进行群体的遗传进化运算以及遗传进化运算的效率。

对于一个具体的应用问题，如何设计一种完美的编码方案一直是遗传算法的应用难点之一，也是遗传算法的一个重要研究方向。可以说目前还没有一套既严密又完整的指导理论及评价准则能够帮助我们设计编码方案。一般地，我们参考 De Jong 曾提出的两条操作性较强的实用编码原则。

（1）有意义积木块编码原则：应使用能易于产生与所求问题相关的且具有低价、短定义长度模式的编码方案。

（2）最小字符集编码原则：应使用能使问题得到自然表示或描述的具有最小编码字符集的编码方案。需要说明的是，这两条编码原则仅仅是给出了设计编码方案时的一个指导性大纲，它并不适合所有的问题。所以对于实际问题，仍必须对编码方法、交叉运算方法、变异运算方法、解码方法等统一考虑，以寻求一种对问题的描述最为方便、遗传运算率最高的编码方案。

由于遗传算法应用的广泛性，迄今为止人们已经提出了许多种不同的编码方法。总的来说，这些编码方法可以分为三大类：二进制编码方法、浮点数编码方法、符号编码方法。

2. 个体适应度函数

遗传算法中，采用适应度来度量群体中各个个体在优化计算中有可能达到、接近于或有助于找到最优解的优良程度。适应度较高的个体以较高的概率遗传到下一代；而适应度较低的个体遗传到下一代的概率就相对小一些。遗传算法的一个特点是它仅使用所求问题的目标函数值就可得到下一步的有关搜索信息。而对目标函数值的使用是通过评价个体的适应度来体现的。评价个体的适应度的一般过程是：

（1）对个体编码申进行解码处理后，可得到个体的表现型；

（2）由个体的表现型可计算出对应个体的目标函数值；

（3）根据最优化问题的类型，由目标函数值按一定的转换规则求出个体的适应度。

最优化问题可分为两大类，一类为求目标函数的全局最大值，另一类为求目标函数的全局最小值。对于这两类优化问题，可用由解空间中某一点的目标标函数值 $f(x)$到搜索空间对应个体的适应度函数值 $F(X)$的转换方法：

对于求最大值的问题，做如下转换：

$$F(X)=\begin{cases} f(X)+C_{\min}, & \text{当} \quad f(X)+C_{\min}>0 \\ 0, & \text{当} \quad f(X)+C_{\min}\leqslant 0 \end{cases} \tag{8.2}$$

式 8.2 中，$C_{\min}$ 相当一个适当的相对较小的数。

对于求最小值问题，做如下转换：

$$F(X)=\begin{cases} C_{\max}-f(X), & \text{当} \quad f(X)<C_{\max} \\ 0, & \text{当} \quad f(X)\geqslant C_{\max} \end{cases} \tag{8.3}$$

式 8.3 中，$C_{\max}$ 相当于一个适当地相对较大的数。

遗传算法中，群体的进化过程就是以群体中个体的适应度为依据，通过一个反复迭代过程，不断地寻求出适应度较大的个体，最终就可得到问题的最优解或近似最优解。

这里，我们以 TSP 问题为例：设有 n 个城市，城市 i 和城市 j 之间的距离为 $d(i,j)i,j=1,\cdots,n$。TSP 问题是要找遍访每个域市恰好一次的一条回路，且其路径总长度为最短。

（1）编码与译码

许多应用问题结构很复杂，但可以化为简单的位串形式编码表示，我们将问题结构变换为位串形式编码表示的过程叫编码；而相反将位串形式编码表示变换为原问题结构的过程叫译码。我们把位串形式编码表示叫染色体，有时也叫做个体。

对 TSP 可以按一条回路城市的次序进行编码，比如码串 134567829 表示从城市 1 开始，依次是城市 3、4、5、6、7、8、2、9，最后回到城市 1。一般情况是从城市 w_1 开始，依次经过城市 $w_2,\cdots,w_n$，最后回到城市 w_1，我们就有如下编码表示：$w_1\ w_2\ \cdots w_n$。

由于是回路，记 $w_{n+1}=w_1$。它其实是 $1,\cdots,n$ 的一个循环排列。要注意 $w_1,w_2,\cdots,w_n$ 是互不相同的。

（2）适应度函数

为了体现染色体的适应能力，引入了对问题中的每一个染色体都能进行度量的函数，叫做适应度函数。通过适应度函数来决定染色体的优、劣程度，它体现了自然进化中的优胜劣汰原则。对优化问题，适应度函数就是目标函数。TSP 的目标是路径总长度为最短，路径总长度的倒数就可以为 TSP 的适应度函数：

$$f(w_1,w_2,\cdots,w_n)=1/\sum_{j=1}^{n}d(w_j,w_{j+1})$$

请注意其中 $w_{n+1}=w_1$。适应度函数要有效反映每一个染色体与问题的最优解染色体之间的差距，一个染色体与问题的最优解染色体之间的差距越小，则对应的适应度函数值之差就越小。适应度函数的取值大小与求解问题对象的意义有很大的关系。

（3）遗传算法基本步骤

下面是遗传算法基本步骤的简单描述：

① 初始化群体；

② 计算群体上每个个体的适应度值；

③ 按由个体适应度值所决定的某个规则选择将进入下一代的个体；

④ 按概率 P_c 进行交叉操作；

⑤ 按概率 P_c 进行突变操作；

⑥ 没有满足某种停止条件，则转第②步，否则进入⑦。

⑦ 输出种群中适应度值最优的染色体作为问题的满意解或最优解。

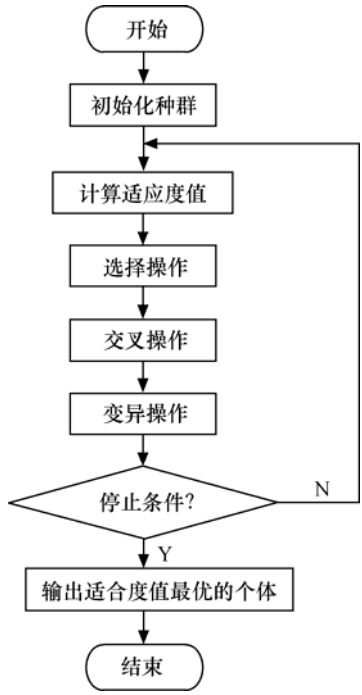


图 8.3　遗传算法基本步骤

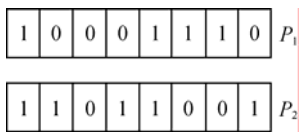
程序的停止条件最简单的有如下两种：完成了预先给定的进化代数则停止；种群中的最优个体在连续若干代没有改进或平均适应度在连续若干代基本没有改进时停止。

根据遗传算法的基本思想可以画出如图 8.3 所示的简单遗传算法框图：

简单遗传算法的遗传操作主要有三种：选择（selection）、交叉（crossover）、变异（mutation）。改进的遗传算法大量扩充了遗传操作，以达到更高的效率。

选择操作也叫复制操作，根据个体的适应度函数值所度量的优、劣程度决定它在下一代是被淘汰还是被遗传。一般地说，选择将使适应度较大（优良）个体有较大的存在机会，而适应度较小（低劣）的个体继续存在的机会也较小。简单遗传算法采用赌轮选择机制，令 $\sum f_i$ 表示群体的适应度值之总和，f_i 表示种群中第 i 个染色体的适应度值，它产生后代的能力正好为其适应度值所占份额 $f_i / \sum f_i$。

交叉操作的简单方式是将被选择出的两个个体 P_1 和 P_2 作为父母个体，将两者的部分码值进行交换。假设有如下 8 位长的二各体：



产生一个在 1～7 之间的随机数 c，假如现在产生的是 3，将 P_1 和 P_2 的低三位交换：P_1 的高五位与 P_2 的低三位组成数串 10001001，这就是 P_1 和 P_2 的一个后代 Q_1 个体；P_2 的高五位与 P_1 的低三位组成数串 11011110，这就是 P_1 和 P_2 的一个后代 Q_2 个体。其交换过程如图 8.4 所示：

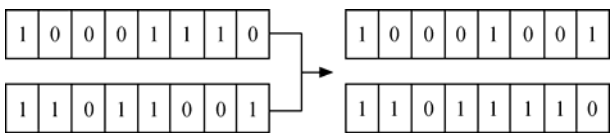


图 8.4　交叉示意图

变异操作的简单方式是改变数码串的某个位置上的数码。我们先以最简单的二进制编码

表示方式来说明，二进制编码表示的每一个位置的数码只有 0 与 1 这两个可能，比如有如下二进制编码表示：

1	0	1	0	0	1	1	0

其码长为 8，随机产生一个 1～8 之间的数 k，假如现在 $k=5$，对从右往左的第 5 位进行变异操作，将原来的 0 变为 1，得到如下数码串（加粗的数字 1 是被变异操作后出现的）：

1	0	1	**1**	0	1	1	0

二进制编码表示时的简单变异操作是将 0 与 1 互换：0 变异为 1，1 变异为 0。

现在对 TSP 的变异操作作简单介绍，随机产生一个 1～n 之间的数 k，决定对回路中的第 k 个城市的代码 w_k 作变异操作，又产生一个 1～n 之间的数 w，替代 w_k，并将 w_k 加到尾部，得到：

$$w_1\ w_2\ \cdots w_{k-1}\ w\ w_{k+1}\ \cdots w_n\ w_k$$

发现这个串有 $n+1$ 个数码，注意数 w 其实在此串中出现重复了，必须删除与数 w 相重复的，得到合法的染色体。

并不是所有被选择了的染色体都要进行交叉操作和变异操作，而是以一定的概率进行，一般在程序设计中交叉发生的概率要比变异发生的概率选取得大若干个数量级，交叉概率取 0.6～0.95 之间的值；变异概率取 0.001～0.01 之间的值。

种群的染色体总数叫做种群规模，它对算法的效率有明显的影响，规模太小不得于进化，而规模太大将导致程序运行时间长。对不同的问题可能有各自适合的种群规模，通常种群规模为 30～100。

另一个控制参数是个体的长度，有定长和变长两种，它对算法的性能也有影响。

遗传算法中最重要的过程就是选择和交叉。选择要能够合理的反映“适者生存”的自然法则，而交叉必须将由利的基因尽量遗传给下一代，编码的过程要能够使编码后的染色体能充分反映个体的特征并且能够方便计算。

8.2.3　遗传算法关键参数

遗传算法中需要选择的运行参数主要有个体编码串长度 l、群体大小 M，交叉概率 p_c、变异概率 p_m、终止代数 T 等。这些参数对遗传算法的运行性能影响较大，需要认真选取。

（1）编码串长度 l

使用二进制编码来表示个体时，编码串长度 l 的选取与问题所要求的求解精度有关:使用浮点数编码来表示个体时，编码串长度 l 与决策变量的个数 n 相等；使用符号编码来表示个体时，编码串长度 l 由问题的编码方式来确定；另外，也可使用变长度的编码来表示个体。

（2）群体大小 M

群体大小 M 表示群体中所含个体的数量。当 M 取值较小时，可提高遗传算法的运算速度，但却降低了群体的多样性，有可能会引起遗传算法的早熟现象；而当 M 取值较大时，又会使得遗传算法的运行效率降低。一般地，取值范围为 20～1 000。

（3）交叉概率 p_c

交叉操作是遗传算法中产生新个体的主要方法，所以交叉概率一般应取较大的值。但若取值过大，又会破坏群体中的优良模式，对进化运算反而产生不利的影响；取值过小，产生新个体的速度以较慢。一般取 0.4～0.99。另外，也可以使用自适应的思想来确定交叉概率 p_c，随着遗传算法在线性能的提高，可以增大交叉概率 p_c 的取值。

（4）变异概率 p_m

若变异概率 p_m 取值较大，虽然能够产生出较多的新个体，但也有可能破坏群体中很多较好的模式，使得遗传算法的性能近似于随机搜索算法的性能；若变异概率 P_m 取值较小，则变异操作产生新个体的能力和抑制早熟现象的能力就会较差，一般都取 0.000 1～0.10。

（5）终止代数 T

终止代数 T 是表示遗传算法运行结束条件的一个参数，它表示遗传算法运行到指定的进化代数之后就停止运行，并将当前群体中的最佳个体作为所求问题的最优解输出，一般取 100～10 000。

8.2.4 遗传算法应用举例

求[0,31]范围内的 $y = (x-10)^2$ 的最小值。

（1）编码算法选择为“将 x 转化为 2 进制的串”，串的长度为 5 位（等位基因的值为 0 或 1）；

（2）计算适应度的方法是：先将个体串进行解码，转化为 int 型的 x 值，然后使用 $y = (x-10)^2$ 作为其适应度计算合适（由于是最小值，所以结果越小，适应度也越好）；

（3）正式开始，先设置群体大小为 4，然后初始化群体（在[0,31]范围内随机选取 4 个整数就可以编码）；

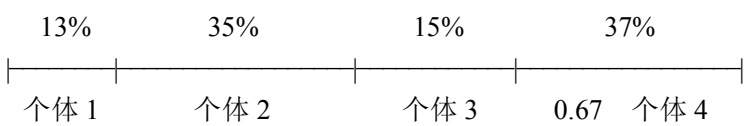
（4）计算适应度 Fi（由于是最小值,可以选取一个大的基准线 1 000，$Fi = 1\,000 - (x-10)^2$）

（5）计算每个个体的选择概率。选择概率要能够反映个体的优秀程度。这里用一个很简单的方法来确定选择概率 $P = Fi/TOTAL(Fi)$。

（6）选择

根据所有个体的选择概率进行淘汰选择。这里使用的是一个赌轮的方式进行淘汰选择。先按照每个个体的选择概率创建一个赌轮，然后选取 4 次，每次先产生一个 0～1 的随机小数，然后判断该随机数落在那个段内就选取相对应的个体。这个过程中，选取概率 P 高的个体将可能被多次选择,而概率低的就可能被淘汰。

下面一个简单的赌轮的例子。



随机数为 0.67 落在了个体 4 的端内。本次选择了个体 4。

被选中的个体将进入配对库（配对集团）准备开始繁殖。

（7）简单交叉

先对配对库中的个体进行随机配对，然后在配对的 2 个个体中设置交叉点，交换 2 个个体的信息后产生下一代。

比如(|代表简单串的交叉位置)

(0110|1,1100|0) --交叉--> (01100,11001)

(01|000,11|011) --交叉--> (01011,11000)

2 个父代的个体在交叉后繁殖出了下一代的同样数量的个体。

复杂的交叉在交叉的位置,交叉的方法,双亲的数量上都可以选择。其目的都在于尽可能地培育出更优秀的后代。

（8）变异

变异操作是根据基因座得出的。比如说每计算 2 万个基因座就发生一个变异（每个个体有 5 个基因座，也就是说要进化 1000 代后才会在其中的某个基因座发生一次变异）。变异的结果是基因座上的等位基因发生了变化。我们这里的例子就是把 0 变成 1 或 1 变成 0。

至此，我们已经产生了一个新的（下一代）集团。然后回到第（4）步。

```
foreach individual in population
{
    individual = Encode(Random(0,31));
}
while (App.IsRun)
{
    //计算个体适应度
    int TotalF = 0;
    foreach individual in population
    {
     individual.F = 1000 − (Decode(individual)−10)^2;
     TotalF += individual.F;
    }
    //−−−−−−−选择过程,计算个体选择概率−−−−−−−−−−−
    foreach individual in population
    {
        individual.P = individual.F / TotalF;
    }
    //选择
    for(int i=0;i<4;i++)
    {
        //SelectIndividual(float p)是根据随机数落在段落计算选取哪个个体的函数
        MatingPool[i] = population[SelectIndividual(Random(0,1))];
    }
    //−−−−−−−简单交叉−−−−−−−−−−−−−−−−−−−−−−−−−−−
    //由于只有 4 个个体,配对 2 次
    for(int i = 0;i<2;i++)
    {
        MatingPool.Parents[i].Mother = MatingPool.RandomPop();
```

```
            MatingPool.Parents[i].Father = MatingPool.RandomPop();
        }
    //交叉后创建新的集团
        population.Clean();
        foreach Parent in MatingPool.Parents
        {
            //注意在 copy 双亲的染色体时在某个基因座上发生的变异未表现.
            child1 = Parent.Mother.DivHeader + Parent.Father.DivEnd;
            child2 = Parent.Father.DivHeader + Parent.Mother.DivEnd;
            population.push(child1);
            population.push(child2);
        }
    }
```

8.3 粒子群优化算法

设想这样一个场景：一群鸟在随机搜索食物。在这个区域里只有一块食物。所有的鸟都不知道食物在那里。但是他们知道当前的位置离食物还有多远。那么找到食物的最优策略是什么呢？最简单有效的就是搜寻目前离食物最近的鸟的周围区域。

粒子群优化算法（PSO）是由 Kennedy 和 Eberhart 在 1995 年提出的一种基于群智能的演化计算技术。粒子群算法源于对鸟群觅食行为的研究。研究者发现鸟群在飞行过程中经常会突然改变方向、散开、聚集，其行为不可预测，但其整体总保持一致性，个体与个体间也保持着最适宜的距离。通过对类似生物群体的行为的研究，发现生物群体中存在着一种社会信息共享机制，它为群体的进化提供了一种优势，这也是粒子群算法形成的基础。其搜索思想是：较优粒子的邻域适应值高的位置的概率较大，所以在群最优粒子的邻域分配更多粒子用以增强算法的搜索效率；同时，每个粒子也通过其他的信息避免盲目陷入局部最优，由于粒子群算法采用了不同于遗传算法的随机搜索策略，在解决某些问题时显现出更优的搜索效能，它是一种基于迭代的优化工具。系统初始化为一组随机解，通过迭代搜寻最优值。

基本粒子群算法中，粒子群由 n 个粒子组成，每个粒子的位置代表优化问题在 D 维搜索空间中潜在的解。每个粒子根据它的位置通过优化函数计算出一个适应值，而且还有一个速度来决定其飞行方向和距离粒子，根据如下三条原则来更新自身状态：

（1）保持自身惯性；

（2）按自身的最优位置来改变状态；

（3）按群体的最优位置来改变状态。

鸟群中的每只鸟在初始状态下是处于随机位置向各个随机方向飞行的，但是随着时间的推移，这些初始处于随机状态的鸟通过自组织逐步聚集成一个个小的群落，并且以相同速度朝着相同方向飞行，然后几个小的群落又聚集成大的群落，大的群落可能又分

散为一个个小的群落。这些行为和现实中的鸟类飞行的特性是一致的鸟群的同步飞行，这个整体的行为只是建立在每只鸟对周围的局部感知上面，而且并不存在一个集中的控制者。

8.3.1　粒子群算法的基本结构

粒子群算法与其他进化算法相类似，也采用“群体”与“进化”的概念，同样也是根据个体（粒子）的适应值大小进行操作。所不同的是，粒子群算法不像其他进化算法那样对个体使用进化算子，而是将每个个体看作是在 n 维搜索空间中的一个没有重量和体积的粒子，并在搜索空间中以一定的速度飞行。该飞行速度由个体的飞行经验和群体的飞行经验进行动态调整。

如果粒子的群体规模为 M，则第 $i(i=1,2,\cdots,M)$个粒子的位置可表示为 X_i，它所经历过的“最好”位置记为 $pbest[i]$，它的速度用 V_i 表示，群体中“最好”粒子的位置的索引号用 g 表示，其位置表示为 $gbest[g]$。所以粒子 i 将根据下面的公式来更新自己的速度和位置。

$$V_i = V_i + c_i \times Rand() \times (pbest[i] - X_i) + c_2 \times rand() \times (gbest[g] - X_i)$$

$$X_i = X_i + V_i$$

其中，c_1、c_2 为常数，称为学习因子。$Rand()$和 $rand()$是[0,1]上的随机数。

基本粒子群算法的流程可以表示为图 8.5：

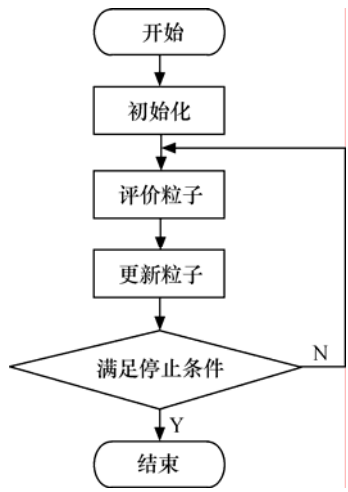


图 8.5　粒子群算法的流程图

基本粒子群算法的步骤。

（1）初始化：初始搜索点的位置 X_i^0 及其速度 V_i^0 通常是在允许的范围内随机产生的，每个粒子的 $pbest$ 坐标设置为其当前位置，且计算出其相应的个体极值（即个体极值点的适应度值），而全局极值（即全局极值点的适应度值）就是个体极值中最好的，记录该粒子的序号，并将 $gbest$ 设置为该粒子的当前位置。

（2）评价每一个粒子：计算粒子的适应度值，如果好于该粒子当前的个体极值，则将 $pbest$ 设置为该粒子的位置，且更新个体极值。如果所有粒子的个体极值中最好的好于当前的全局极值，则将 $pbest$ 设置为该粒子的位置，记录该粒子的序号，且更新全局极值。

（3）粒子的更新：用

$$V_i = V_i + c_1 \times Rand() \times (pbest[i] - X_i) + c_2 \times rand() \times (gbest[g] - X_i)$$

$$X_i = X_i + V_i$$

对每一个粒子的速度和位置进行更新。

（4）检验是否符合结束条件：如果当前的迭代次数达到了预先设定的最大次数（或达到最小错误要求），则停止迭代，输出最优解，否则转到第（2）步。

8.3.2 粒子群算法的关键参数

为了改善基本粒子群算法的收敛性能，Y. Shi 与 R. C. Eberhart 首次在速度进化方程中引入惯性权重，即

$$V_i = w \times V_i + c_1 \times Rand() \times (pbest[i] - X_i) + c_2 \times rand() \times (gbest[g] - X_i)$$

式中，w 为惯性权重，因此，基本粒子群算法是惯性权重 $w = 1$ 的特殊情况。惯性权重 w 使粒子保持运动惯性，使其有扩展搜索空间的趋势，有能力探索新的区域。

对全局搜索，通常的好方法是在前期有较高的探索能力以得到合适的种子，在后期有较高的开发能力以加快收敛速度。为此，可将 w 设定为随着进化而线性减少。Y. Shi 与 R. C. Eberhart 的仿真实验结果也表明 w 的线性减少取得了较好的试验结果。这种方法称为是粒子群算法的标准版本，即标准粒子群算法。

标准粒子群参数包括：惯性权重函数 w，学习因子 c_1,c_2，最大迭代次数 $iter_{\max}$ 和群体规模 M。

c_1,c_2 可视为加速度常量。c_1 反映了微粒飞行过程中所记忆的最好位置（pbest）对微粒飞行速度的影响，称为“认知系数”，c_2 反映了整个微粒群所记忆的最好位置（pbest）对微粒飞行速度的影响，称为“社会学习系数”。大量实验证明，c_1,c_2 之和最好接近 4.0，通常 $c_1 \approx c_2 = 2.05$。由式

$$V_i = w \times V_i + c_1 \times Rand() \times (pbest[i] - X_i) + c_2 \times rand() \times (gbest[g] - X_i)$$

可以看出，公式的右边由三部分组成。第一部分是粒子更新前的速度，而后两部分反映了粒子速度的更新。美国的 Shi 与 Eberhart 研究发现公式，等式的第一部分 V_i 由于具有随机性且其本身缺乏记忆能力，有扩大搜索空间，探索新的搜索区域的趋势。因此，具有全局优化的能力。在考虑实际优化问题时，往往希望先采用全局搜索，使搜索空间快速收敛于某一区域，然后采用局部精细搜索以获得高精度的解。因此，在公式 g 前乘以惯性权重 w，w 较大算法具有较强的全局搜索能力，w 较小则算法倾向于局部搜索。一般的做法是将 w 初始为 0.9 并使其随迭代次数的增加线性递减至 0.4，以达到上述期望的优化目的，令

$$w = w_{\max} - \frac{w_{\max} - w_{\min}}{iter_{\max}} \times iter$$

群体规模 M 越大，算法的寻优能力越强，但计算量越大。

另外，粒子在不断根据速度调整自己的位置时，还要受到最大速度 $V_{\max}$ 的限制，当 V_i 超过 $V_{\max}$ 时被限定为 $V_{\max}$。

8.3.3 应用举例

粒子群算法被提出之初，主要是用于求函数极值这类连续空间的优化问题。由于该算法结构简单、速度快、基本思想易于理解，一经提出就被应用于求解诸多问题，TSP 就是其中之一，一些针对 TSP 问题的粒子群算法也被提了出来。

应用 PSO 解决优化问题的过程中有两个重要的步骤: 问题解的编码和适应度函数。

PSO 的一个优势就是采用实数编码，不需要像遗传算法一样是二进制编码（或者采用针

对实数的遗传操作。例如对于问题 $f=x_1^2+x_2^2+x_3^2$ 求解，粒子可以直接编码为(x_1,x_2,x_3)，而适应度函数就是 f。接着我们就可以利用前面的过程去寻优。这个寻优过程是一个迭代过程，中止条件一般为设置为达到最大循环数或者最小错误。PSO 中并没有许多需要调节的参数，下面列出了这些参数以及经验设置。

粒子数：一般取 20～40。其实对于大部分的问题 10 个粒子已经足够可以取得好的结果，不过对于比较难的问题或者特定类别的问题，粒子数可以取到 100 或 200。

粒子的长度：这是由优化问题决定，就是问题解的长度。

粒子的范围：由优化问题决定，每一维可是设定不同的范围。$V_{\max}$：最大速度，决定粒子在一个循环中最大的移动距离，通常设定为粒子的范围宽度，例如上面的例子里，粒子$(x_1,x_2,x_3)x_1$ 属于[−10,10]，那么 $V_{\max}$ 的大小就是 20 学习因子：c_1 和 c_2 通常等于 2。不过也有其他的取值。但是一般 c_1 等于 c_2 并且范围在 0～4 之间。

中止条件：最大循环数以及最小错误要求。例如，在上面的神经网络训练例子中，最小错误可以设定为 1 个错误分类，最大循环设定为 2 000，这个中止条件由具体的问题确定。

下面以 TSP 问题为例。TSP 问题可以简单地描述成：设有 n 个城市并已知各城市间的旅行费用，找一条走遍所有城市且费用最低的旅行路线。其数学描述如下：设有一城市集合 C=$\{c_1,c_2,\cdots,c_n\}$，其每对城市 $c_i,c_j\in$C 间的距离为 $d(c_i,c_j)\in Z^+$。求一条经过 C 中每个城市正好一次的路径$(c\pi(1),c\pi(2),\cdots,c\pi(n))$，使得：

$\sum_{i=1}^{n-1} d(c_{\pi(i)},c_{\pi(i+1)})+\sum_{i=1}^{n-1} d(c_{\pi(n)},c_{\pi(1)})$ 最小。这里$(\pi(1),\pi(2),\cdots,\pi(n))$是$(1,2,\cdots,n)$的一个置换。

每个粒子表示一个可行解，并采用路径表示法。若有 N 个城市的 TSP，将城市从 0～$N-1$ 编号。在初始化粒子群时，对可行解$(0,2,\cdots,N-1)$进行随机次数的翻转。这里将速度定义为一组子路径的集合，并设这些子路径都只包括两个城市。分别设计了 4 种速度的生成方式：

（1）设 N 个城市的 TSP，定义 E_1、E_2、E_3 为(0,1)间的常量，E_1*N 为从全局最优路径中选择子路径的次数，E_2*N 为从个体最优路径中选择子路径的次数，E_3*N 为新速度中子路径的最大数目。先从最优路径中随机选择 E_1*N 次，除去其中重复的子路径；再从个体最优路径中随机选择 E_2*N 次，再去掉重复的路径，若选择路径中有城市在速度中已出现过两次，则淘汰掉距离最长的子路径；最后将原速度中的子路径加入新速度中，同时去掉重复的路径，保留通过同一城市的子路径中的最优的两条。

（2）E_1、E_2、E_3 的意义不变，只是在从全局最优路径、个体最优路径中选择子路径时不再是随机选择，而是先对路径中的所有子路径进行排序，子路径越短序号越大，并根据序号分配选择概率 P_i：

$$p_i=\frac{2\times S_i}{(1+N)\times N}$$

其中，S_i 是第 i 条子路径的序号，N 是子路径数。然后按照选择概率使用轮盘赌法进行选择。

（3）与 R_2 不同之处在于从全局最优路径、个体最优路径中选择子路径时的选择概率的生成方式。先对所有可行的子路径按照起始城市进行排序，得到一个排序表。如：由城市 c_i 出发有 $N-1$ 条可直接到达其它城市的子路径，按照距离由近到远的顺序将序号 $N-1,\cdots,1$ 分配

给它们。在从全局最优路径、个体最优路径中选择子路径时，按下式为路径 L 的每条子路径分配选择概率：

$$P_i = \frac{S_{L_i,L_{i+1}}}{S_{L_{N-1},L_0} + \sum_{j=0}^{N-2} S_{L_j,L_{j+1}}}$$

其中，L_i 表示路径 L 的第 i 座城市的编码，L_i,L_{i+1} 表示子路径（L_i,L_{i+1}）在排序表中的值。选择时仍使用轮盘赌法。

（4）在计算选择概率时考虑当前粒子的情况，即在选择 L 中由 L_i 出发的子路径时，还有考虑在当前路径 P 中由 L_i 出发的子路径。此时 L 中第 i 条子路径的选择概率为：

$$P_i = \frac{S_{L_i,L_{i+1}} - S_{p_k,p_{k+1}} - \alpha}{S_{L_{N-1},L_0} + \sum_{j=0}^{N-2} S_{L_j,L_{j+1}} - S_{p_{N-1},p_0} - \sum_{j=0}^{N-2} S_{p_j,p_{j+1}} - N \cdot \alpha}$$

其中 $L_i = P_k$，$\alpha = \min((S_{L_i,L_{(i+1) \bmod N}} - S_{p_k,p_{(k+1) \bmod N}})i,k = 0,1,\cdots,N-1;L_i = P_k)$ 选择时仍使用轮盘赌法。

初始化速度时，为每个粒子随机生成多个子路径构成初始速度。在由当前路径产生新路径时，根据速度中的子路径对当前路径进行翻转，使当前路径包含该子路径。其过程如下：

```
For(所有的粒子)
While(若速度中还有没处理完的子路径)
{
    取速度中的一条子路径 Ri；
    在当前路径中找到子路径中的第一个城市 Ri,1；
    若当前路径中已包含该子路径，结束对该子路径的处理；
    否则
    在当前路径中找到该子路径的另一个城市 Ri,2；
    将当前路径中之间的路径翻转，并要保证不影响已处理的子路径；
}
```

8.4 人工神经网络

“人工神经网络”（ARTIFICIAL NEURAL NETWORK，简称 ANN）是在对人脑组织结构和运行机制的认识理解基础之上模拟其结构和智能行为的一种工程系统。早在 20 世纪 40 年代初期，心理学家 McCulloch、数学家 Pitts 就提出了人工神经网络的第一个数学模型，从此开创了神经科学理论的研究时代。其后，F Rosenblatt、Widrow 和 J.J.Hopfield 等学者又先后提出了感知模型，使得人工神经网络技术得以蓬勃发展。人工神经网络是由大量的神经元广泛互连而成的系统，它的这一结构特点决定着人工神经网络具有高速信息处理的能力。

人脑的每个神经元大约有 103～104 个树突及相应的突触，一个人的大脑总计约形成

1 014～1 015 个突触。用神经网络的术语来说，即是人脑具有 1 014～1 015 个互相连接的存储潜力。虽然每个神经元的运算功能十分简单，且信号传输速率也较低（大约 100 次/秒），但由于各神经元之间的高速并行互连功能，最终使得一个普通人的大脑在约 1 秒内就能完成现行计算机至少需要数 10 亿次处理步骤才能完成的任务。

人工神经网络的知识存储容量很大。在神经网络中，知识与信息的存储表现为神经元之间分布式的物理联系。它分散地表示和存储于整个网络内的各神经元及其连线上。每个神经元及其连线只表示一部分信息，而不是一个完整具体概念。只有通过各神经元的分布式综合效果才能表达出特定的概念和知识。

由于人工神经网络中神经元个数众多以及整个网络存储信息容量的巨大，使得它具有很强的不确定性信息处理能力。即使输入信息不完全、不准确或模糊不清，神经网络仍然能够联想思维存在于记忆中的事物的完整图像。只要输入的模式接近于训练样本，系统就能给出正确的推理结论。正是因为人工神经网络的结构特点和其信息存储的分布式特点，使得它相对于其他的判断识别系统（如专家系统等），具有另一个显著的优点：健壮性。生物神经网络不会因为个别神经元的损失而失去对原有模式的记忆，最有力的证明是，当一个人的大脑因意外事故受轻微损伤之后，并不会失去原有事物的全部记忆。人工神经网络也有类似的情况。因某些原因，无论是网络的硬件实现还是软件实现中的某个或某些神经元失效，整个网络仍然能继续工作。

人工神经网络是一种非线性的处理单元。只有当神经元对所有的输入信号的综合处理结果超过某一门限值后才输出一个信号。因此神经网络是一种具有高度非线性的超大规模连续时间动力学系统。它突破了传统的以线性处理为基础的数字电子计算机的局限，标志着人们智能信息处理能力和模拟人脑智能行为能力的一大飞跃。

8.4.1　神经网络模型

神经网络是一个并行和分布式的信息处理网络结构，该结构一般由多个神经元组成，每个神经元有一个单一的输出，它可以连接到其他的神经元，其输入有多个连接通路，每个连接通路对应一个连接权系数。

人工神经网络结构可分为以下几种类型如图 8.6 所示。

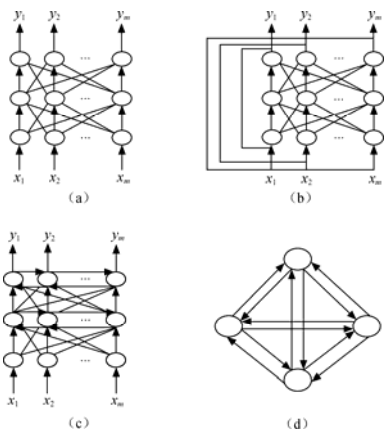


图 8.6　人工神经网络结构的几种类型

（1）不含反馈的前向网络，如图 8.6（a）所示。

（2）从输出层到输入层有反馈的前向网络，如图 8.6（b）所示。

（3）层内有相互连接的前向网络，如图 8.6（c）所示。

（4）反馈型全相互连接的网络，如图 8.6（d）所示。

人工神经网络的模型要求发展神经网络型计算机来替代传统的计算机。这种计算系统不再是传统的顺序执行命令的运行过程，而是希望对输入进行平行处理，这种计算机系统不再是只包含一个或几个复杂的计算设备，而是由众多简单设备有机组成在一起，这种计算系统处理信息时，不再将信息存储在一个精确的位置上，而是通过神经元的内部相连关系达到信息存储的功能。

8.4.2 神经网络学习规则

学习规则是修正神经元之间连接强度或加权系数的算法，使获得知识结构适用周围环境的变换。

1. 无监督 Hebb 学习规则

Hebb 学习是一类相关学习，它的基本思想是：如果有两个神经元同时兴奋，则它们之间的连接强度的增强与它们的激励的乘积成正比。用 y_i 表示单元 i 的激活值（输出），y_j 表示单元 j 的激活值，w_{ij} 表示单元 j 到单元 i 的连接加权系数，则 Hebb 学习规则可表示如下：

$$\Delta w_{ij}(k) = \eta \cdot y_i(k) y_i(k) \tag{8.4}$$

式 8.4 中 η 为学习速率。

2. 有监督 δ 学习规则或 Widow-Hoff 学习规则

在 Hebb 学习规则中引入教师信号，将上式中的 y_i 换成网络期望目标输出 d_i 与实际输出 y_i 之差，即为有监督 δ 学习规则

$$\Delta W_{ij}(k)=\eta \cdot [d_i(k)-y_i(k)] \cdot y_j(k)=\eta \cdot \delta \cdot y_j(k)$$

$$\delta = d_i(k) - y_i(k) \tag{8.5}$$

式 8.5 表明，两神经元之间的连接强度的变化量与教师信号 $d_i(k)$ 和网络实际输出 y_i 之差成正比。

3. 有监督 Hebb 学习规则

将无监督 Hebb 学习规则和有监督学习规则两者结合起来，组成有监督 Hebb 学习规则，即

$$\Delta w_{ij}(k) = \eta \cdot [d_i(k) - y_i(k)] \cdot y_i(k) \cdot y_i(k) \tag{8.6}$$

这种学习规则使神经元通过关联搜索对未知的外界作出反应，即在教师信号 $d_i(k)-y_i(k)$ 的指导下，对环境信息进行相关学习和自组织，使相应的输出增强或消弱。

人工神经网络在视觉、语言、信号处理和机器人等方面有成功应用的示例。

对于神经网络的总体评价，可以归纳为以下几个方面：

（1）提供信息处理的一种新手段，主要在于人工神经网络的自适应性、学习能力和大规模的平行计算能力；

（2）发展已较为成熟，主要表现在模型的建立和数学理论的支持，计算工具的迅速发展

和神经生物学的发展；

（3）人工神经网络的发展仍然受到限制，主要表现在计算机设备在存储、速度和用户柔性方面的非适应能力，限制了人工神经网络的发展。

习　　题

1. 用模拟退火算法求解图的着色问题（注：每种颜色赋一个权值）。
2. 遗传算法中 3 个基本遗传算子的特点是什么？

第 9 章　并行算法简介

9.1　基本概念

近十几年来，并行计算已经从计算机科学的边缘成为了计算机科学的主流，相比计算机科学的其他领域，它的发展是非常迅速的。目前有多种形式的并行计算机，所包括的处理器数也从 2 个到成千上万个不等。相比于熟悉的串行计算机（它有一个统一的冯 •诺依曼模型），我们不能假定有一个通用的计算模型并希望它适合所有的并行计算机。同样，并行算法的设计、分析和正确性证明比起相应的串行算法来说也要复杂得多。

不能期望短短一章就可覆盖并行计算的所有或大部分领域，这里只给出一个粗略的基本介绍，深入细致的讨论请参阅文中指出的相关参考文献。并行算法的设计介绍可以按照串行算法一样依照排序、选择、搜索、串匹配、数值计算和图论算法这样分专题来详细讲述，但限于篇幅，这里只是选择了在四种模型中几个典型问题的处理方法加以叙述。本章最后部分介绍了 OpenMP 和 MPI 编程，也只是浅尝辄止，希望能起到抛砖引玉的作用。

9.1.1　并行计算机系统结构模型

1. 并行计算机系统的基本概念

传统的计算机是串行结构的，每一时刻只能按照一条命令指令对一个数据进行操作。所谓串行计算机就是只有单个处理单元顺序执行计算程序的计算机，也称为顺序计算机。为了克服这种传统结构对提高运算速度的限制，从 20 世纪 60 年代起开始将并行处理技术引入计算机的结构设计中来，利用它对计算机系统结构进行改进。现实中的问题有些是固有串行的，然而大多数问题可以在某种程度上并行化，使用的处理器越多（在某个界限内）算法就越快。这种所谓的并行处理就是首先把一个传统串行处理的任务分解开，然后将其分配给多个处理器同时进行处理，也就是说在同一时间间隔内增加计算机的操作数量。并行计算的演变从时间上并行和空间上并行两个方向进行。时间上并行主要是各种流水线技术。空间上并行主要有 SIMD（单指令流多数据流）并行机，它用同一控制器同步地控制所有的处理器阵列执行相同操作来开发空间上的并行性；如果用不同的控制器异步地控制相应的处理单元执行各自

的操作，则派生出另一类非常主要的 MIMD（多指令流多数据流）并行机；其中，如果各处理单元通过公用存储器中的共享变量实现相互通信，则称为多处理机（Multiprocessors）；如果处理单元之间使用消息传递的方式来实现相互通信，则称为多计算机（Multicomputers），它也是当今最流行的并行计算机。

2. Flynn 分类法

1966 年 Flynn 按照指令流和数据流的多倍性概念将计算机系统结构进行了分类。其中，指令流是指机器所执行的指令序列，数据流是指指令流所调用的数据序列，而多倍性是指机器的瓶颈部件上所可能并行执行的最大指令或数据的个数。根据指令流和数据流的不同组合，计算机系统可分为单指令流单数据流（Single Instruction Stream Single Data Stream，SISD），单指令流多数据流（Single Instruction Stream Multiple Data Stream，SIMD），多指令流单数据流（Multiple Instruction Stream Single Data Stream，MISD），多指令流多数据流（Multiple Instruction Stream Multiple Data Stream，MIMD）。

其中，SISD 计算机代表了如今使用的大多数串行计算机，即传统的单处理机，是单指令流对单数据流进行操作。

SIMD 计算机是所谓的阵列机，它有许多个处理单元（PE），由同一个控制部件管理，所有 PE 都接受控制部件发送的相同指令，对来自不同数据流的数据集合序列进行操作。

MISD 计算机从概念上讲，有多个 PE，接受不同的指令，对相同的数据进行操作。一般认为它是一种不太实际的计算机，但也有学者把超标量机和脉动（Systolic）阵列机归属于此类。

MIMD 计算机最初包括多处理机和多计算机两类，它们都由各自执行自己程序的多处理器组成。其中，多处理机以各处理机共享公共存储器为特征，而多计算机以各处理器经通信链路传递信息为特征。它们与 SIMD 计算机的根本区别在于：SIMD 机中每台处理器只能执行中央处理器的指令，而 MIMD 机中的每台处理器仅接受中央处理器分配给它的任务，它执行自己的指令，可达到指令、任务并行。发展到现在，出现了所谓的混合系统和网格。混合系统是指具有独立地址空间的节点集群，每个节点包含几个共享存储器的处理器。而网格是指使用分布式的、异类资源的系统，这些资源通过 LAN 或 WAN 连接，通常互连网络是 Internet。网格的思想已经演化为一种共享异类资源的常用方式，网格中的资源不需要具有一个用于管理的公用点。

根据 Flynn 分类法，通用的并行计算机分为 SIMD 机和 MIMD 机两大类。对于并行算法设计而言，不能局限于某种具体的并行机而设计并行算法，而必须从算法的角度，将各种并行机的基本特征加以理想化，抽象出所谓的并行计算机模型，然后在此基础上研究和设计各种有效的并行算法。由 Flynn 分类法，可将并行计算机分为两大类，这两大类也确定了两大类并行计算模型，即 SIMD 和 MIMD 两类并行计算模型。这两大类还可以进一步细分，SIMD 模型可细分为基于共享存储器的 SIMD 模型和基于互连网络的 SIMD 模型；MIMD 模型可细分为基于共享存储的 MIMD 模型和基于异步通信的互连网络模型。我们将在第二节并行算法设计中做更详细的介绍。

9.1.2 并行计算性能评价

传统串行算法的复杂度是指运算时间步和存储空间，它们都是求解问题规模 n 的函数，当 n 趋向无穷大时，就得到算法复杂度的渐近表示。一般而言，我们关心的是 n 充分大时的复杂度，这时它与渐近复杂度相差不大，在算法分析时，往往对这两者不给予区分。对于一个给定规模 n 的问题，通常有各种可能的输入集合。在一般情况下，很难做到对输入的分布作出一个适当的假定，所以感兴趣的是分析在某些输入时，使得算法的时空复杂度呈现最坏情况下的算法复杂度，这种复杂度称为最坏情况下的复杂度。并行算法可朴素地解释为适合在各种并行计算模型上求解问题和处理数据的算法。并行算法可用不同的标准度量，但我们主要关心的是算法与求解问题规模 n 之间的关系。

在 SIMD 计算模型上的并行算法，将分析算法的运行时间和所需的处理器数，在最坏情况下与问题规模 n 的关系。

运行时间 $T(n)$：运行时间就是在给定模型上求解问题所需的时间，即算法从开始执行到结束所经过的时间。此时间通常包含在一个处理器内执行算术和逻辑运算所需要的计算时间，以及数据从源处理器经互连网络到达目的处理器所需要的选路时间。通常，两者分别用计算时间步和选路时间步作单位。

处理器数 $P(n)$：求解给定问题所需的处理器的数目，显然它是问题规模 n 的函数。当 n 很大时，$P(n)>n$ 显然是不合适的。通常 $P(n)$应是 n 的亚线性函数（Sublinear Function）。虽然 $P(n)=\log n$ 或 $P(n)=n^{1/2}$ 也是 n 的亚线性函数，但它们的取值对实际的计算机系统而言并不灵活。因此，$P(n)$通常限制为 $P(n)=n^{1-\varepsilon}$，$0<\varepsilon<1$。

在 MIMD 计算模型上，对基于共享存储的算法，其时间还应该包含同步等时间，它们与 SIMD 模型上的复杂度度量基本一致；而对于基于异步通信的分布式计算模型，其算法的度量标准主要有两个。

通信复杂度：指算法在整个执行期间所传送的消息的总数，它可能是基本长度的消息总数，或者是传送消息的二进制位数的总和。

时间复杂度：指算法从第一台处理器上开始执行到最后一台处理器上终止的这段时间间隔。

在基于异步通信的分布式计算模型中，由于处理器之间传递的消息虽然可以在有限的时间内到达目的地，但此时间的长短却是不确定的；同时算法的执行与处理器互连的拓扑结构密切有关，所以要想精确地分析算法的时间复杂度是非常困难的。因此，目前估算出的复杂度都是假定相邻处理器之间的通信可在常数时间内完成这一基础上得出的。

在求解一个给定问题的众多的并行算法中，一般采用下面 4 种评价并行算法性能的标准。

（1）并行算法的成本 $C(n)$

成本 $C(n)$定义为并行算法的运行时间 $T(n)$与其所需的处理器数 $P(n)$的乘积

即

$$C(n)=T(n)*P(n)$$

它相当于在最坏的情况下求解某一问题的总执行步数。如果求解一个问题的并行算法的成本，在数量级上等于最坏情况下的串行求解此问题所需的执行步数，那么称此并行算法是成本最优的。

（2）加速比 $Sp(n)$

对于一个求解问题，令 $Ts(n)$是最快的串行算法在最坏的情况下的运行时间，$Tp(n)$是求解同一问题的某并行算法在最坏情况下的运行时间，则该并行算法的加速比 $Sp(n)$可定义为

$$Sp(n)=Ts(n)/Tp(n)$$

由定义可知，加速比是评价算法的并行性对运行时间的改善程度。$Sp(n)$越大，则并行算法越好。由于任何并行算法都可以在一台串行机器上模拟实现，因此 $Tp(n)*P(n)\geqslant Ts(n)$，从而

$$1\leqslant Sp(n)\leqslant P(n)$$

当 $Sp(n)=P(n)$时，则具有这样加速比的并行算法是最优的并行算法。但是在一般情况下，一个问题不可能分解成具有相同执行步数并且可以并行执行的子问题，因此，要达到 $Sp(n) = P(n)$几乎是不可能的。

（3）并行算法的效率 $Ep(n)$

并行算法的效率可定义为算法的加速比与处理器数目之比，即

$$Ep(n) = Sp(n)/P(n)$$

并行算法的加速比不能反映处理机的利用率，一个并行算法的加速比可能很大，但是处理机的利用率却可能很低，并行算法的效率反映了在执行算法时处理机的利用情况。由于 $1\leqslant Sp(n)\leqslant P(n)$，所以

$$0<Ep(n)\leqslant 1$$

如果一个并行算法的效率等于 1，那么说明在执行该算法的过程中每台处理机的能力得到了充分发挥。此时，$Sp(n) = P(n)$，因此，该并行算法的串行模拟是求解问题的最佳串行算法。

（4）并行算法的可扩放性

除了上面介绍的加速比和效率外，可扩放性（Scalability）也是评价并行算法的重要性能指标之一。它的含义是在确定的应用背景下，算法的性能能否随处理器数的增加而按比例地提高。通常使用大系统是很昂贵的，可扩放性指标允许我们首先在一个小系统上推导出可扩放性函数，然后用它来预测大系统的性能。目前已经提出了很多具体的可扩放性度量标准，如等效率（Isoefficiency）、等速度（Isospeed）和等利用率（Isoutilization）。任何系统的效率可表示成工作负载和机器规模的函数。如果我们将效率固定为某个常数并对工作负载求解效率方程，则所得到的函数就称为系统的等效率函数。等效率值越小，则当机器规模增大时为保持相同效率所需增加的工作负载就越小，因此就有更好的系统可扩放性。等速度的概念类似于等效率。不保持固定效率，而在扩展机器规模和问题规模同时，保持固定速度。

一个理想的可扩放性指标应有以下 2 个性质：

- 像等效率或等速度一样，它能预测相对于机器规模增加而需使工作负载增长的值；
- 它应与执行时间一致，即一个可扩放性好的系统总有更短的执行时间。

如果将利用率固定为某一常数并对工作负载求解利用率方程式，那么所得到的函数便称为等利用率函数或简称等利用率。等利用率能预测，相对于机器规模增大，为保持相同利用率，应使工作负载增加多少。具有较小等利用率的系统比有较大等利用率的有更好的可扩放性。此外，在很合理的条件下可以看到，有较小等利用率的系统总是有较短的执行时间。

9.2 并行算法设计

并行程序设计包括将一个问题分解成若干部分，然后由各个处理器对各部分分别进行计算。一个理想的并行计算是能被立即分解成许多完全独立部分且它们能同时执行的计算，可以贴切地称为自然并行。对这种问题进行并行化是一目了然的，不需要特殊技巧或算法就可得到一个成功解。一个理想的自然并行计算在各个进程间没有通信。每个进程需要不同（或相同）的数据，并由其输入数据产生最终的结果，而不需要使用其他进程生成的结果。这里所需的惟一结构，是对数据进行简单地分配并启动各进程。大量著名的实际应用是属自然并行的，或至少是接近于自然并行的，例如图像的几何转换、显示曼德勃罗特（Mandelbrot）集和蒙特卡罗法等问题。

然而，还有许多问题不是自然并行的，需要使用一些技巧来解决。设计并行算法大体上有 3 种方法：

（1）检测和开拓现有串行算法中的固有并行性而直接将其并行化；

（2）修改已有的并行算法使其可求解另一类相似问题；

（3）从问题本身的描述出发，从头开始设计一个全新的并行算法。

对一类具有内在顺序性的串行算法则难于并行化；修改已有的并行算法有赖于特定的一类问题；设计全新的并行算法，尽管技术上尚不成熟且似乎又有些技巧，但也不是无章可循。目前普遍使用的设计方法有平衡树方法、划分和分治策略和流水线技术等。平衡树方法是将输入元素作为叶节点构筑一棵平衡二叉树，然后自叶向根往返遍历。这种方法成功的部分原因是在树中能快速地存取所需要的信息，它对数据的播送、压缩、抽取和前缀计算等很有效。划分和分治，不仅仅运用在串行算法中，同样也是并行程序设计中的两种最基本的技术。所谓划分是将问题简单地分为几个独立的部分，而且每部分都独立计算。划分技术可以应用于程序的数据，称为数据划分；也可以应用于程序的功能，称为功能分解。分治方法的特点是将一个问题分为与原来的较大问题有相同形式的子问题。进一步分为较小的问题通常是通过递归，递归是我们在顺序程序设计中常使用的一种方法。递归的方法会将问题不断分为更小的问题直到不能再分为止。接着就完成这些非常简单的任务并把结果合并，然后按更大的任务继续合并，直到获得最终的结果。流水线并行处理技术广泛适用于本质上是部分串行的问题，也就是说这些问题必须执行一系列步骤。因此我们可以采用流水线对其中的顺序代码进行并行化。假定一个问题能够被分解成一系列的顺序任务，那么采用下列 3 种计算类型，可以使用流水线方法来取得加速：

（1）如果将执行整个问题的多个实例；

（2）如果必须处理一系列的数据项，而每个数据项需要多次操作；

（3）如果进程在完成自己的所有内部操作之前能够把下一个进程启动所需的信息向前传送。

本节接下来将依次讨论 SIMD 共享存储模型、SIMD 互连网络模型、MIMD 共享存储模型和 MIMD 异步通信模型上的一些典型问题的处理方法，部分体现了上述并行设计方法的运用。

9.2.1　SIMD 共享存储模型

SIMD 共享存储模型是假定有有限或无限个功能相同的处理器，每个处理器拥有简单的算术运算和逻辑判断能力，在理想的情况下假定存在一个容量无限大的共享存储器，在任何时刻，任意一个处理器均可通过共享存储器的共享单元同其他任何处理器互相交换数据，也称之为 PRAM（Parallel Random Access Machine）模型，即并行随机存取机器。由于实际情况是共享存储器的容量是有限的，因此在同一时刻，当多个处理器访问同一单元时就会发生冲突。根据模型解决冲突的能力，也就是处理器对共享存储单元同时读、同时写的限制，PRAM 模型又可进一步分为：不允许同时读和同时写的 PRAM 模型，简记为 PRAM－EREW；允许同时读但不允许同时写的 PRAM 模型，简记为 PRAM-CREW；允许同时读和同时写的 PRAM 模型，简记为 PRAM-CRCW。显然，允许同时写是不现实的，于是又对 PRAM-CRCW 模型做了进一步的约定：只允许所有的处理器同时写相同的数，称之为公共的 PRAM-CRCW，简记为 CPRAM-CRCW；只允许最优先的处理器先写，称之为优先的 PRAM-CRCW，简记为 PPRAM-CRCW；允许任意处理器自由写，称之为任意的 PRAM-CRCW，简记为 APRAM-CRCW。上述模型中，PRAM-EREW 是最弱的计算模型，而 PRAM-CRCW 是最强的计算模型。

在 SIMD 共享存储模型中设计算法颇受算法研究者欢迎，因为它抛开了各种具体的体系结构，使人们能集中精力从问题本身出发，研究和挖掘求解问题本身固有的并行性，简化算法的设计和分析，易于算法间的相互比较，使得并行算法的研究成为一项独立的活动。接下来讨论几个在该模型上的典型并行算法。

在讨论并行算法前，先来讨论一下怎么描述它的问题。描述一个算法，可以使用自然语言进行物理描述；也可以使用某种程序设计语言进行形式化描述。同描述串行算法所选用的语言一样，类 PASCAL、类 C 等语言都可以选用。在这些语言中，允许使用任何类型的数学描述，通常也没有数据类型的说明部分，但只要需要，任何数据类型都可以引进。在描述并行算法时，所有描述串行算法的语句及过程调用等等都可以使用，而只是为了表达并行性而引入几条并行语句：

par-do 语句当算法的若干步要并行执行时，可使用“do in parallel”语句，简记之为“par-do”进行描述：

for i = 1 to n par-do

……

end for

for-all 语句 当几个处理器同时执行相同的操作时，可以使用“for all”语句描述：

for all P_i，where　$0 \leqslant i \leqslant k$ do

……

end for

为了算法书写简洁，在意义明确的前提下，参数类型总是省去。

【例 9.1】 广播算法

广播算法能解决 SIMD-EREW 模型上读冲突问题，因此，在此模型上的许多求解问题中都要用到它。假定有 N 个处理器（编号从 1～N），都要读取共享存储器中的数据 m，对于这

个简单问题，在 SIMD-CREW 模型上，显然只需一次并行读操作就可以完成。在 SIMD-EREW 模型上，由于不允许读冲突，故若采用处理器依次读取数据 m，则是一种串行操作，没有发挥并行处理机的并行处理能力，其运算时间为 $O(N)$。下面给出解决此问题的并行算法。假定 N 是 2 的幂，则它的基本思想是使用一个长度为 N 的共享数组 B（开始为空）。第一步由处理器 P_1 把 m 写入 $B(1)$。第二步由处理器 P_1 把 $B(1)$写入 $B(2)$；接下来由 P_1、P_2 把 $B(1)$、$B(2)$并行写入 $B(3)$、$B(4)$；由 $P_1,P_2,\cdots,P_i$ 把 $B(1),B(2),\cdots,B(i)$并行写入 $B(i+1),B(i+2),\cdots,B(2i)$；最后由 $P_1,P_2,\cdots,P_{N/2}$ 把 $B(1),B(2),\cdots,B(N/2)$并行写入 $B(N/2+1),B(N/2+2),\cdots,B(N)$。第三步各个处理器 $P_i(i=1,2,\cdots,N)$从 $B(i)$中并行读取数据 m。该算法可描述如下：

（1）处理器 P_1 将 m 复制到自己的存储器中，然后将其写入 $B(1)$

（2）for (i = 0;i≤log N－1;i++)

for j = 2^i + 1 to 2^{i+1} par-do

{处理器 P_j 将 $B(j-2^i)$复制到自己的存储器中；然后将其写入 B(j);}

end for

（3）for i = 1 to N par-do

处理器 P_i 从 B(i)中读取数据 m ;

end for

从算法的描述容易看出，第（1）、第（2）和第（3）步的运算时间分别为 $O(1)$、$O(\log N)$和 $O(1)$。因此，该算法的时间复杂度为 $O(\log N)$。

接下来评价一下该广播算法的性能：

（1）成本。由成本的定义式可直接推出该算法成本为 $O(N\log N)$。

（2）加速比。由于此问题的最佳串行算法的运算时间为 $O(N)$，所以该算法的加速比为 $S_P(N)=O(N/\log N)$

（3）效率。该算法的效率为

$$E_P(N)=S_P(N)/N=O(1/\log N)$$

在上述算法中假定了处理器个数 N 是 2 的幂，如果该条件不满足，那么将算法第（2）步中 for i = 0 to log N－1 do 改成 for i = 0 to $\lceil \log N-1 \rceil$ do 即可。修改后的算法的运算时间和性能都不会产生变化。

【例 9.2】 求和算法

假定共享存储器中有 N 个数据 $S_1,S_2,\cdots,S_N$。利用 $N-1$ 个处理器 $P_2,\cdots,P_N$ 求出局部与整体和 $S_1+S_2+\cdots S_j(1<j\leqslant N)$。

此问题求解可以通过并行执行下面的操作来实现：

第 1 步　处理器 $P_i(2\leqslant i\leqslant N)$完成 $S_1\leftarrow S_i+S_{i-1}$；

第 2 步　处理器 $P_i(3\leqslant i\leqslant N)$完成 $S_1\leftarrow S_i+S_{i-2}$；

第 3 步　处理器 $P_i(5\leqslant i\leqslant N)$完成 $S_1\leftarrow S_i+S_{i-4}$；

……

第 J 步　处理器 P_i（$2^{j-1}+1\leqslant i\leqslant N$）完成 $S_1\leftarrow S_i+S_{i-2^{j-1}}$，其中 $j=\lceil \log N \rceil$。其基本思想是充分利用上次累加结果来做下次并行计算。算法可以描述如下：

for{j = 1;j≤$\lceil \log N \rceil$; j ++ }

for i = 2^{j-1} + 1 to N par−do

{处理器 P_i 经共享存储器获得 S_i、$S_{i-2^{j-1}}$；然后计算：$S_i \leftarrow S_i+S_{i-2^{j-1}}$};

end for

容易看出，在该算法执行过程中不存在读、写冲突，因此该算法可在 SIMD-EREW 模型上实现。由于每一步并行仅仅包含 3 个操作，所以算法的时间复杂度是 $O(\log N)$。因为求和问题的最佳串行算法的时间复杂度是 $O(N)$，所以该并行算法的加速比是 $O(N/\log N)$，效率是 $O(1/\log N)$。对该算法中的并行操作加以修改，就可解决其它一些问题，如并行求积算法、并行求最大值算法等。

【例 9.3】 k-选择算法

假定输入序列 $S=(x_1,\cdots,x_n)$，且系统中有 $N=n^{1-\varepsilon}(0<\varepsilon<1)$个处理器可用，欲求 S 中第 k 个最小者（$1\leqslant k\leqslant n$），其过程可叙述如下：

① 先将 S 分成若干个段，每段指派一个处理器；

② 各段同时并行求取各自的中值（可使用任意的顺序选择算法）；

③ 求各中值的中值；

④ 以所求得的中值为基准，将 S 划分成小于、等于、大于该中值的 3 个子序列；

⑤ 以一定的规则判断各子序列长度与 k 值的大小关系，以确定 k 值，或继续在相应的子序列中重复上述过程，直到找到第 k 个最小者为止。

在上述的第④步中，为了将 S 进行划分，需要把中值的中值播送到各个处理器中，这可以用广播算法实现；在判断各子序列的长度时，还要将子序列中的元素计数求和，可以用上面所说的求和算法实现。

1．SIMD-EREW 模型上的 k-选择算法可描述如下。

（1）if |S|<3 then 使用一个处理器 return(k);

else {将 S 分成长度各为 n^{ε} 的 $n^{1-\varepsilon}$ 个序列，每个分别指派一个处理器; }

（2）for i=1 to $n^{1-\varepsilon}$ par−do

P_i 使用顺序选择算法找其所辖子序列的中值 m_i;

P_i 将 m_i 写入共享存储器中数组 M 的第 i 个单元 $M(i)$;

end for

（3）递归找 M 的中值 m。

（4）将 S 分成 3 个子序列 S_1,S_2,S_3，其中元素分别小于、等于、大于 m。

（5）if $|S_1| \geqslant k$ then PARALLEL SELECT(k,S_1);

else if $|S_1|+|S_2|\geqslant k$ then return(m);

else PARALLEL SELECT($k-|S_1|-|S_2|$,S_3);

2．对该算法进行分析如下。

（1）假定共享存储器中 S 的首地址 A，问题规模 n 和 k 值在算法开始前已经播送给所有的处理器，这可用广播算法花费 $O(\log n^{1-\varepsilon})$的时间完成。处理器 P_i 可在常数时间内计算出其子序列的首地址 $A+(i-1)n^{\varepsilon}$ 和末地址 $A+in^{\varepsilon}-1$。所以算法第(1)步所需时间为 $c_1\log n$，其中 c_1 是常数。

（2）每个处理器，使用顺序选择算法可在 $O(n^{\varepsilon})$时间内求出中值。所以算法第（2）步所

需的时间为 c_2n^{ε}，其中 c_2 是常数。

（3）第（3）步是递归调用，所以时间是 $t(n^{1-\varepsilon})$。

（4）将 S 划分成 S_1、S_2、S_3，可使用归并划分法，时间复杂度是 $O(n^{\varepsilon})$，具体方法如下：处理器 $P_i(i=1,\cdots,n^{1-\varepsilon})$将其子序列按小于 m、等于 m 和大于 m 划分成 S_1^i、S_2^i 和 S_3^i（时间为 $O(n^{\varepsilon})$）；然后将相应的 S_1^i、S_2^i 和 S_3^i 归并成 S_1、S_2 和 S_3（时间为 $O(n^{\varepsilon})$）。为此，令 $S_i=|S^i|$，对于每个 $i(1\leqslant i\leqslant n^{1-\varepsilon})$计算 $Z_i=\sum_{j=1}^{i}S_j$，所有这样的求和可用 $n^{1-\varepsilon}$ 处理器，调用上述求和算法在 $O(\log n^{1-\varepsilon})$时间内完成。取 $Z_0=0$，所有的处理器可同时将其局部存储器中的 S_1^i 写到共享存储器而形成 S_1。对于 P_i 而言，其写入首地址是 $Z_{i-1}+1$，而时间正比例于最长的 S_1^i 的长度，它不会超过 n^{ε}。所以算法第(4)步的时间复杂度是 c_3n^{ε}，其中 c_3 是常数。

（5）因为 m 是 M 的中值，所以 S 中的 $n^{1-\varepsilon}/2$ 个元素可保证不会大于它，而且 M 中的每个元素至多小于 S 的 $n^{\varepsilon}/2$ 个元素，因此$|S_1|\leqslant(3/4)n$。类似地，$|S_3|\leqslant(3/4)n$。所以算法第（5）步的时间复杂度是 $t(3/4\ n)$。

因此整个算法的运算时间为

$$t(n)=c_1\log n+c_2n^{\varepsilon}+c_3n^{\varepsilon}+t(n^{1-\varepsilon})+t(3/4\ n)$$

可解出 $t(n)=O(n^{\varepsilon})$，$n>4$。因为 $p(n)=n^{1-\varepsilon}$，所以成本 $c(n)=O(n)$，故该算法是最佳的。

9.2.2 SIMD 互连网络模型

SIMD 互连网络模型，简记为 SIMD-IN，也称为分布存储的 SIMD 模型，简记为 SIMD-DM。在这种模型中，每个处理器在控制器控制下处于活动状态，或不活动状态。活动状态的处理器都执行相同的指令，处理器之间的数据交换是通过互连网络进行的。其中各处理器（包括算术逻辑单元和本地存储器）可以通过多种互连方式连接，从而又可以进一步细分为多种不同类型，这里不一一详述。许多实验性的和商品并行机几乎都是基于这种结构。

在基于互连网络的模型中，由于数据分布存储，信息通过互连网络进行传递，因此算法与处理器或处理器与存储器互连的拓扑结构紧密相关。下面通过示例来讨论几个常见的互连网络拓扑结构上的并行算法。

【例 9.4】 一维线性模型上的并行排序算法

一维连接又称为线性阵列（Linear Array，LA）。这种连接方式是并行机中最简单、最基本的互连方式。其中每个处理器只与其左、右近邻相连（头尾处理器除外），所以也叫做二近邻连接。当首尾处理器相连时可构成循环移位连接，在拓扑结构上等同于环。下面我们来构造这种模型上 n 个数的并行排序算法。

假定待排序的 n 个数的输入序列 $S=\{x_1,x_2,\cdots,x_n\}$，处理器 $P_i(1\leqslant i\leqslant n)$存有输入数据 X_i，算法步骤如下：

① 所有奇数编号的处理器 P_i 接受来自 P_{i+1} 中的 X_{i+1}。如果 $X_i>X_{i+1}$，那么 P_i 和 P_{i+1} 彼此交换其内容。

② 所有偶数编号的处理器 P_i 接受来自 P_{i+1} 中的 X_{i+1}。如果 $X_i>X_{i+1}$，那么 P_i 和 P_{i+1} 彼此交换其内容。

交替重复上述两并行步，经$\lceil n/2 \rceil$次迭代后算法结束。

该算法的形式化描述如下：

```
for ( k= 1; k ≤ ⌈n/2⌉ ; k++ )
    {for each Pi : i=1,3,…,2⌈n/2⌉−1 par-do
        if  Xi>Xi+1  then Xi←→Xi+1 ;
     end for
     for each Pi : i=2,4,…,2⌊(n−1)/2⌋ par-do
        if  Xi>Xi+1  then Xi←→Xi+1 ;
     end for
}
```

可运用归纳法证明该算法至多经过 n 步可完成排序，同时也证明了算法的正确性。由上所述，很容易求出该算法的运算时间 $T(n) = O(n)$，成本 $C(n) = O(n^2)$，加速比为 $O(\log n)$。

【例 9.5】 树形模型上的求最小值算法

二叉树是大家都非常熟悉的数据结构。树形连接方式就是利用二叉树这种数据结构组织而成的。树连接（Tree-Connected，TC），除了根节点和叶节点外，每个内节点都与其父节点和两个子节点相连，因此二叉树连接可以看做是三近邻连接。假定二叉树有 d 级（也称为 d 层，编号自根至叶为 0～d−1），则共有 $n = 2^d-1$ 个节点。这种结构中处理器的工作方式通常是：根和叶节点具有 I/O 功能，且叶节点执行并行计算，而内节点仅仅负责叶节点间的通信及简单的逻辑运算。

在求集合 $S=\{x_1,x_2,\cdots,x_n\}$的最小元素问题中，首先假定 $n = 2^d$（d>0）因此树形模型中共有 2^d 个叶子，总共有 $2^{d+1}-1$ 个节点，即有 $P(n) = 2^{d+1}-1$ 个处理器。其中每个叶处理器能存放一个数，非叶处理器能存放两个数并且可以决定哪个更小。算法的基本思想是：首先将 S 中 n 个元素加载到各个叶处理器中，然后从 d 级开始每个非叶处理器从左右子节点中取出两个数进行比较，并保存较小者，最后使根节点保存集合 S 中的最小元素。该算法可以形式化描述如下：

```
for i = 2^d to 2^(d+1)−1 par-do
Pi 读取 Xi−2^d+1 ;
end for
for( k = d−1 ; d ≥ 0 ; d−)
     {for  j = 2^k  to  2^(k+1)−1  par-do
     A(j)←min{A(2j)，A(2j+1)};
/*从左、右儿子中取出两个数进行比较，并保存较小者*/
     end for
}
```

显然，算法中的第一个 for 循环运算时间为 $O(1)$；因为 $d = \log n$，所有算法中的第二个嵌套 for 循环运算时间为 $O(\log n)$；因此算法的运算时间 $T(n) = O(\log n)$。由于求极值的最佳串行算法的时间复杂度是 $O(n)$，所以该算法的加速比是 $O(n/\log n)$。算法的成本是 $O(n\log n)$，所以该算法不是成本最佳的。

9.2.3 MIMD 共享存储模型

共享存储的 MIMD 计算模型是一个异步的 PRAM 模型，系由多个处理器组成，它的特点是每个处理器都有自己的本地存储器、局部时钟和局部程序；处理器间的通信经过共享全局存储器；没有全局时钟，各个处理器异步地执行各自的指令；处理器任何时间依赖关系必须明确地在各处理器的程序中加入同步（路）障（Synchronization Barrier）；一条指令可在非确定但有限的时间内完成。所谓异步算法是一组进程的组合，它们的部分或全部均可在一些有效的处理器上并行地执行，而执行的过程大致是这样的：开始时所有的处理器都是空闲的，当一并行算法在任意选定的处理器上启动执行时，它就生成一些称之为进程的待执行的计算任务。所以进程相应于算法的段，可能有好几个各自具有不同参数的进程具有相同的算法段。一旦生成一个进程，它就必须在某一处理器上执行。如果有一空闲的处理器，进程就指派给它，此处理器就执行进程所指定的计算；否则的话 ，进程就必须排队等待一个空闲的处理器。当处理器执行完一个进程时，它就变为空闲，等待的进程就可立即指派给它；如果没有等待的进程，处理器就排队等待新进程的产生。

下面我们来讨论几个该模型上的典型算法。

【例 9.6】 并行求和算法

假定有 P 个处理器，把它们标记为 $P_0,P_1,\cdots,P_{P-1}$，全局存储器存有变量 $a_0,a_1,\cdots,a_{n-1}$，它们包含有待加的各数的值，全局变量 g 存放最终结果。算法的基本思路是：对于每个处理器 P_i（$0\leqslant i<p$），它们各自利用其局部变量 l_i 计算 $a_i+a_{i+p}+a_{i+2p}+\cdots+a_{i+kip}(n-p\leqslant i+k_{ip}\leqslant n)$，然后，将求得的子和加到全局变量 g 中。显然，此算法存在存储冲突，解决的办法是当一个处理器访问全局变量 g 时对其加锁，访问完毕后立即开锁。该算法可以形式化描述如下：

```
g = 0;
for each Pi：0≤i ＜ p  par−do
li = 0 ;
for( j = 0 ; j ≤ n ; j += p )
li = li + aj;
lock(g);
g = g + li;
unlock(g);
end for
```

接下来分析上述算法在最坏情况下的运算时间。如果不考虑生成进程所需要的时间，那么算法的时间开销是：

（1）局部求和所需要的时间 $\Theta(n/p)$；

（2）p 个处理器串行地对全局变量 g 进行加锁、修改它的值、开锁操作所需的时间 $\Theta(p)$；

（3）同步开销 $\Theta(p)$。

因此，上述算法在最坏情况下的运算时间是 $\Theta(n/p+p)$。

【例 9.7】 矩阵乘法的并行算法

可以将串行的矩阵求积算法改造为 MIMD 共享存储模型上的并行算法。一个 $m\times n$ 阶的 A 矩阵与一个 $n\times k$ 阶的 B 矩阵相乘就得到一个 $m\times k$ 阶的 C 矩阵，即

$$A_{m\times n}\times B_{n\times k}=C_{m\times k}$$

其中 C 矩阵的元素 $C_{ij}=\sum_{s=1}^{n}a_{is}\times b_{sj}$，$1\leqslant i\leqslant m$，$1\leqslant j\leqslant k$。

一个典型的矩阵相乘串行算法如下：

```
for ( i =1 ;i ≤ m ; i++)
    for( j =1 ;j ≤ k ; j++)
        {Cij = 0 ;
        for ( s = 1 ; s ≤ n ; s++)
            Cij = Cij + ( ais×bsj);
        }
```

假定 $m\leqslant n$，$k\leqslant n$，显然上述算法的运行时间为 $O(n^3)$。如果从两矩阵不管如何相乘，都要产生 n^2 个输出结果的观点来看，矩阵相乘的下界应该是$\Omega(n^2)$。可以通过把循环并行化来达到并行计算的目的。由于算法中 3 个 for 循环都可以并行化，现在的问题是哪一层循环并行化后加速最大？

设某一问题在单处理机上最坏情况下的时间复杂度为 $t(n)$，那么在有 p 个处理器的并行机上的时间复杂度至少是$\Omega(t(n)/p)$。为了经由单一的全局信号量同步，每个进程必须加锁、增一和解锁信号量。

不失一般性，假定它取一个单位时间，那么 p 个处理器同步至少需$\Omega(p)$个时间单位。所以求解给定问题的任何并行算法的时间复杂度为$\Omega(max\{t(n)/p,p\})$，此函数当 $p=\sqrt{t(n)}$ 时具有最小值。因而，该并行算法的时间复杂度为$\Omega(t(n)/p)=\Omega(\sqrt{t(n)})$，所以加速比为

$$t(n)/\sqrt{t(n)}=O(\sqrt{t(n)})$$

现在，从两矩阵相乘的串行算法可知：最内层循环的时间复杂度为 $\Theta(n)$，并行化该层所能达到的最大加速为 $O(\sqrt{n})$；中层循环的时间复杂度为 $\Theta(n^2)$，并行化该层所能达到的最大加速为 $O(\mathrm{n})$；最外层循环的时间复杂度为 $\Theta(n^3)$，并行化该层所能达到的最大加速为 $O(n^{1.5})$。当然还有一些其他因素，诸如算法的划分，A 和 B 的元素的竞争等都导致不可能达到如此大的加速比。

根据上面的讨论，两矩阵相乘的并行算法应该是把相应串行算法的最外层循环并行化。算法描述如下。

假定算法开始时已由某个处理器生成了一些所希望的进程。令 $P(m)$表示第 m 个进程，而 $i(m)$、$j(m)$、$k(m)$和 $t(m)$均为该进程的局部变量。

```
for all P(m)，1≤m≤p par−do /*p 为进程数*/
for ( i(m)= m ; i(m)≤n ; i(m)+= p )
for (j(m)= 1 ; j(m)≤ n ; j(m)++)
{ t(m)= 0 ;
for ( k(m)= 1 ; k(m)≤ n ; k(m)++)
t(m)= t(m)+ a[i(s)k(m)]× b[k(m)j(m)] ;
}
```

算法有 p 个进程，每个进程计算 C 矩阵的 n/p 行。计算一行所需的时间为 $\Theta(n^2)$，所以每

个进程的计算复杂度为 $\Theta(n^2 \cdot (n/p)) = \Theta(n^3/p)$；诸进程正好同步一次，所以同步的开销为 $\Theta(p)$。因此整个并行算法的时间复杂度为 $\Theta(n^3/p + p)$。注意，因为只有 n 行，所以至多生成 n 个进程执行该算法。如果不考虑存储器的竞争，预计的加速可达线性。

【例 9.8】 MIMD-CREW 模型上的异步枚举排序算法

对于基于枚举比较原理的异步排序算法，为了排序 n 个数的序列 $S = (x_1,\cdots,x_n)$，算法要生成 n 个进程。进程 $i(1 \leqslant i \leqslant n)$将 x_i 与 S 中其余元素进行比较，并且使用局部变量 k 记下所有小于 x_i 的元素的数目。当所有的比较都完成时，就将 x_i 置入排序序列中的 $k+1$ 位置上，因此每个进程都可能彼此独立地执行，而没有通信的要求。

令 X 是存在共享存储器中长度为 n 的数组，开始时放入被排序的序列；当算法结束时，结果置于共享存储器中的 T 数组内。变量 i、j、k 是算法生成的每个进程的局部变量，于是算法可形式化描述如下：

```
（1）for ( i = 1; i ≤ n ; i ++)
            Create process i;
（2）process i:
 k = 0;
for ( j = 1 ; j ≤ n ; j ++ )
     {if X(i) ＞X(j) then k = k+1 ;
else if ( X(i) = X(j) and i＞j ) then k = k+1 ;
}
T(k+1) = X(i)
```

在上述算法中，因为所有的进程都必须同时访问整个数组 X，所以算法的模型是 MIMD-CREW。在算法中，通常一个处理器执行一个进程。算法的第（1）步可任意选定一个处理器来生成各个进程。当所有的进程都生成后，此处理器便可释放，并能执行另一个等待的进程。

【例 9.9】 MIMD-TC 模型上的异步快排序算法

快排序开始是找序列 S 的中值 m，将其置于有序序列的$\lceil n/2 \rceil$位置上；然后 S 按 m 划分成小于和大于 m 的两个子序列 S_1 和 S_2，再对它们递归调用快排序。当子序列长度小于 3 时，此子序列至多用一次比较可直接排序之。

假定在 MIMD 共享存储的模型中，各处理器之间以二叉树的方式互连。为了易于理解，令 n 是 2 的方幂。树的每个节点存有算法执行时所产生的一个子序列。节点中的数字表示子序列的长度。在树的第 0 级首先找 S 的第 $n/2$ 个最小者，然后 S 被划分成长度各为 $n/2-1$ 和 $n/2$ 的两个子序列，它们由根的两个子节点表示；接着在树的第 1 级按同样的方法继续划分。当子序列的长度小于或等于 2 时划分停止。因为这样的树有 $n/2$ 个叶子，所以总共有 $n-1$ 个节点。这就意味着算法所生成的进程数为 $n-1$ 个。

令 X 是存在共享存储器中长度为 n 的数组，开始时 $X(i) = \{X_i\}(1 \leqslant i \leqslant n)$；$Q_i$ 为 X 的子数组，且 q_i 为 Q_i 中第一个元素的首地址，$|Q_i| = s_i$；R 为共享存储器中长度为 $2^{\log n}-1$ 的用以存放(q_i,s_i)的数组。于是算法可形式化描述如下。

（1）令 $Q_1 = X$；

（2）R(1)＝(q_1，n)；

（3）生成进程 1；

（4）进程 i：

由 R(i)读取(q_i,s_i)；

if　S_i ≤ 2 then 直接排序 Q_i；

else {

找中值 m(即 Qi 的第$\lceil s_i/2 \rceil$个最小元素)；

将 m 置换在 X 的最后单元中；

将 Q_i 划分成其元素分别小于和大于 m 的 Q_{2i} 和 Q_{2i+1}；

R(2i) = (q_{2i}，s_{2i})；

R(2i + 1) = (q_{2i+1}，s_{2i+1})；

生成进程 2i 和 2i + 1；

}

在上述算法中，第（1）、第（2）和第（3）步指派给处理器 P_1 执行；并且处理器 P_1 执行进程 1。如果处理器 P_k 生成 Q_{2i}，则在执行进程 $2i$ 时，处理器 P_k 总是得到优先权。当进程 $2i$ 结束时，如果 Q_{2i+1} 仍在等待，那么它就指派给处理器 P_k。

9.2.4　MIMD 异步通信模型

MIMD 异步通信计算模型可以抽象为一个无向图，其中顶点集对应处理器集合，边集对应处理器间的双向通信链集合。每个处理器都赋予惟一的编号，且只具有知晓与其有线相连的近邻处理器的局部知识。系统中并无共享存储器，各处理器之间的通信是通过发送和接受消息完成的。在算法运行期间，每个处理器除了执行自己的计算任务外，还向邻近的处理器发送消息和接受并处理来自邻近处理器的消息。假定计算时间远远小于通信时间，并且假定通信是无故障的，这样每个处理器发送给近邻处理器的消息总可以在有限的（但不确定的）时间内到达。在一条通信链上同一方向上所到达的消息，服从先进先出的规则。

在这种基于异步通信的分布式计算模型上开发的算法也叫做分布式算法。换一种说法，分布式算法是由通信链接的多个场点或节点协同完成某项任务的算法。在该种算法中，假定记录元素都是分步在若干场点中的局部存储器中，各场点间可以任意形式的互连网络连接，一组进程的通信都经由固定的一组交换信息的通信。通信的双方都约定在一个发送、另一个接受的规程上。令 $N=(P,C)$是一通信网络，其中，P 是一组进程，C 是一组通道，输入数据 I 分步在各进程中，所谓分布式算法就是相对于 N 和 I 对问题 Q 求解。

【例 9.10】 分布式 k 选择算法

通常所称的 k 选择算法有两种：一是随机 k 选择算法，另一个是确定 k 选择算法。前者平均复杂度是线性的，最坏情况下是二次的；后者最坏情况下的复杂度是线性的。此两种算法具有很多共同之处，差别仅在于划分元素的选取方式不同，一是随机选取，二是按某一确定方法选取，从而各得其名为随机 k 选择算法和确定 k 选择算法。

1．随机 k 选择算法

（1）顺序随机 k 选择算法

令 $B=\{b_1,\cdots,b_n\}$是元素的集合，欲从其中选取第 k 个元素，则单处理机上随机 k 选择算法可描述如下：

① 如果$|B|=1$，则返回此元素；否则执行以下各步；

② 随机从 B 中挑选一个元素 m（以下称其为划分元素）；

③ 将 B 划分成 3 个子集合 BL、BE、BG，它们分别包含＜、＝、＞m 的那些元素。

$$\text{令 } B'=\begin{cases} BL，\text{若 } k \leqslant |BL| \\ BE，\text{若} |BL|<k\leqslant |BL|+|BE| \\ BG，\text{若 } k>|BL|+|BE| \end{cases}$$

如果 $B'=BE$，则返回元素 m；否则按下式计算 k'的值：

$$k'=\begin{cases} k，\text{若 } B'=BL \\ k-|BL|-|BE|，\text{若 } B'=BG \end{cases}$$

④ 递归调用本算法，以求出 B'第 k'个元素。

（2）分布式随机 k 选择算法

令 $B=\{b_1,\cdots,b_n\}$是元素的集合，$S=\{s_1,\cdots,s_p\}$是场点集合，$L\subseteq S\times S$ 是通信链集合，且假定分布式网络已形成了一株生成树，则上述算法的非递归部分（即第（1）步～第（3）步）可以直接了当地转换成分布式算法，对于递归部分只要由根节点协调好递归的入口和出口即可。一个场点，也称为生成树的一个节点，分配一个处理器，运行一个进程。MIMD-AC 模型上的随机 k 选择算法可描述如下：

① 通过对有根生成树的一次扫描，根节点就可计算出总的元素数｜B｜。

如果｜B｜＝1，则根节点通知该元素所在节点将此元素送往根节点，算法结束；否则执行以下各步；

② 分布随机地从｜B｜个元素中挑选出一个元素 m（划分元素）送到根节点。其过程是：假定每个进程（节点）给其元素和其孩子都赋予一个固定的序号，并且还假定每个节点都知道 $t(1),\cdots,t(p)$。其中 $t(i)$是它的第 i 个子树中所有元素的数目（$1\leqslant i\leqslant p$）。根节点在区间 1～n 随机地选择一整数 i，为了找相应的元素，它首先检查驻留在自己局部存储器中 t 个元素是否是此元素；如果 $i\leqslant t$，则说明有此元素，否则根节点发送命令 $LOCATE(j)$给第 f 个子节点，其中 $j=i-t-t(1)-\cdots-t(f-1)$(取最小正整数)。根据接受的 $LOCATE(j)$信息，接收进程就像根节点一样作出类似的反应。当已经定位到一个元素时，它就被发送至根并作为 k 选择算法的划分元素，发送给所有其他节点。此步所需的交换消息数为 $O(p)$。

③ 每个进程 i 将其局部存储器中的元素按 m 划分成三个子集合 BL_i、BE_i、BG_i，它们分别包含＜、＝、＞m 的那些元素。通过对生成树从叶子到根的一次扫描，在根节点可计算出$|BL|$、$|BE|$、$|BG|$。一旦计算出$|BL|$、$|BE|$、$|BG|$，根节点就可以根据 B' 和 k' 决定算法是以选中 m 而结束，还是继续递归调用。根节点向所有其他节点广播这一决定，以便让每个节点 i 知道集合 BL_i 和 BG_i 中哪一个应作为下一次递归调用的参数，这一步需要交换的消息数为 $O(p)$。

④ 根据新的参数 B'和 k'，算法就可自动递归调用算法了。在分布式环境中，递归调用时其入口和出口均由根节点完成。它分布地计数现有活跃元素的数目。如果很多，则根节点通知所有其他节点，它们都递归调用它们局部的程序。当只剩下一个元素时，根节点就令其他节点将此元素发送给它，从而得到了第 k 个元素。此时每个进程都可以从递归调用中退出而

无需与根节点进一步通信就可结束。

不难分析出上述算法所需的交换消息数（即通信复杂度）：在平均情况下为 $O(p\log n)$，在最坏情况下为 $O(pn)$，其中 p 为进程数。至于所需的存储空间可分析如下：因为每个进程均有 p 个近邻，所以必须保持一数组 $t(1),\cdots,t(p)$；为了划分元素，每个进程只要求常数存储空间；计数$|BL|$、$|BE|$、$|BG|$每个进程也只需常数存储空间；递归调用时，常数存储空间也足够了。所以如果场点的度数看做常数，每个进程的总的存储空间的要求也将是常数。

2. 确定 k 选择算法

（1）顺序确定 k 选择算法

单处理机上的确定 k 选择算法可描述如下：

① 如果$|B|$比较小（例如少于 50 个元素），那么可使用排序的方法求第 k 个元素；否则执行以下各步；

② 将 B 按 5 个一组进行分组（至多有一组包含少于 5 个元素，称此组为零头）；

③ 采用排序法求每组的中值，从而形成中值集合 M；

④ 自身递归调用，求集合 M 的中值 m；

⑤ 同单处理机上随机 k 选择算法的第③步；

⑥ 递归调用本算法，以求出 B'中的第 k'个元素。

（2）分布式确定 k 选择算法

MIMD-AC 模型上的确定 k 选择算法可描述如下：

① 类似 MIMD-AC 模型上的随机 k 选择算法的第①步，分布地求出$|B|$，如果此值足够小可以装入根部，则在根部求出第 k 个元素；否则：

② 每个进程均分布地按 5 个元素（在它的局部存储器中）一组进行划分。但由于各进程都可能有零头，所以总零头可能很大。为解决该问题，可令每个进程都从其子节点接收零头并拼成 5 个一组后，再将其零头传送到它的父节点。这里可能有 $O(p)$的消息交换；

③ 局部地求 5 个元素的中值；

④ 以 M 作为参数，递归调用求 M 的中值 m；

⑤ 类似分布式随机 k 选择算法的第（3）、第（4）步。

协同递归的入口和出口都在根部完成。根节点分布计数现有的活跃元素；如果所剩不多，那么根节点通知所有其他进程将这些元素发送到根节点，然后由它求出第 k 个元素；如果有很多活跃元素，那么根节点通知所有其他进程，于是它们都递归地调用局部的程序。

9.3 并行程序开发

并行编程是通过并行语言来表达的，并行语言的产生方式主要有 3 种：（1）设计全新的并行语言；（2）扩展原来的串行语言的语法成分，使它支持并行特征；（3）不改变串行语言，仅为串行语言提供可调用的并行库。

设计一种全新的并行语言的优点是可以完全摆脱串行语言的束缚，从语言成分上直接支

持并行，这样就可以使并行程序的书写更方便、更自然，相应的并行程序也更容易在并行机上实现。但是，由于并行计算至今还没有一个统一的标准，因此并行机、并行算法和并行语言的设计和开发方法千差万别，虽然有多种全新的并行语言出现，但至今还没有任何一种并行语言能成为人们普遍接受的标准。

一种重要的对串行语言的扩充方式就是标注，即将对串行语言的并行扩充作为原来串行语言的注释。对于这样的并行程序，若用原来的串行编译器来编译，标注的并行扩充部分将不起作用，仍将该程序作为一般的串行程序处理；若使用扩充后的并行编译器来编译，则该并行编译器就会根据标注的要求，将原来串行执行的部分转化为并行执行。对串行语言进行并行扩充，相对于设计全新的并行语言，显然实现难度有所降低，但需要重新开发编译器，以支持扩充的并行部分。

仅仅提供并行库，是一种对原来的串行程序设计改动最小的并行化方法。这样，原来的串行编译器也能够使用，不需要任何修改，编程者只需要在原来的串行程序中加入对并行库的调用，就可以实现并行程序设计。

9.3.1 并行程序设计概念

目前两种最重要的并行编程模型是数据并行和消息传递。数据并行编程模型的编程级别比较高，编程相对简单，但它仅适用于数据并行问题；消息传递编程模型的编程级别相对较低，但消息传递编程模型可以有更广泛的应用范围。

数据并行，即将相同的操作同时作用于不同的数据，因此适合在 SIMD 并行计算机上运行。数据并行编程模型提供给编程者一个全局的地址空间。一般这种形式的语言本身就提供并行执行的语义，因此对于编程者来说，只需要简单地指明执行什么样的并行操作和并行操作的对象，就实现了数据并行编程。它可以解决一大类科学与工程计算问题，但对于非数据并行类的问题，如果通过数据并行的方式来解决，一般难以取得较高的效率。数据并行不容易表达甚至无法表达其他形式的并行特征。

消息传递，即各个并行执行的部分之间通过传递消息来交换信息、协调步伐、控制执行。消息传递一般是面向分布式内存的，但是它也可以适用于共享内存的并行机。消息传递为编程者提供了更灵活的控制手段和表达并行的方法，一些用数据并行方法很难表达的并行算法，都可以用消息传递模型来实现。它一方面为编程者提供了灵活性，另一方面，也将各个并行执行部分之间复杂的信息交换和协调、控制的任务交给了编程者，这就在一定程度上增加了编程者的负担，这也正是消息传递编程模型级别低的原因。虽然如此，消息传递的基本通信模式是简单和清楚的，学习和掌握这些部分并不困难，因此目前大量的并行程序设计是消息传递并行编程模式。

9.3.2 共享存储系统并行编程

在本节中，我们将概述在共享存储器中进行编程的一些方法，包括进程、线程、并行程序设计语言以及具有编译器命令和库例程的顺序语言的使用。

在一个共享存储器系统中，任一个处理器都可以访问全部的存储单元。所谓单一编址空间就是每一个存储单元都由一个单地址范围内的某个特定地址所指定。对于少量处理器的系

统，一个通用的体系结构就是单总线的体系结构。其中，所有的处理器和存储模块都连接在同一总线上。对于具有较多处理器的系统，为获得足够带宽可以使用全交叉开关那样的多重互连，也可以使用包括多级互连网以及交叉开关和总线组合在内的其他互连结构。理想情况下，共享存储器系统应具有均匀存储器存取，即从任何处理器对任何存储单元的访问都具有相同的高速访问时间。

OpenMp 是一个共享存储器标准，是为在多处理机上编写并行程序而设计的一个应用编程接口，得到许多硬件和软件供应商的支持，如 DEC、Intel、IBM 等。它由一个小型的编译器命令集、一个扩展的小型库例程和使用 Fortran 和 C/C++基本语言的环境变量所组成，以合成方式为 Unix 和 Windows NT 平台提供共享存储器 API。也就是说，OpenMP 通过与标准 FORTRAN，C 和 C++结合来工作。

在共享存储的并行程序中，标准的并行模式是 FORK-JOIN 式并行。在最初的 FORK- JOIN 结构中，一个 FORK 语句产生一个新的并发进程的路径，并且并发进程在其结尾使用 JOIN 语句。当原进程和新产生的进程都到达 JOIN 语句后，代码继续以顺序的方式执行。如果需要更多的并发进程，则需要执行更多的 FORK 语句。当程序开始执行的时候只有一个称作主线程的线程存在。主线程执行算法的顺序部分，当遇到需要进行并行运算时，主线程派生出一些附加线程。在并行区域内，主线程和这些派生的线程协同工作。在并行代码段结束时，派生的线程退出或者挂起，同时控制流回到单独的主线程中。

也可以把顺序执行的程序看做是共享存储模型程序的一种特殊情况，即没有 FORK-JOIN 的形式。无论是仅仅在一个循环并行执行中使用了一个 FORK-JOIN 的并行程序，还是那些大部分代码都被并行执行的程序，它们都属于共享存储模型的并行程序。因此，共享存储模型支持增量并行化，即一次操作并行化程序中的一段代码，进行多次这样的操作从而可以将整个的顺序程序转化为并行程序。下面通过介绍一些典型问题的处理方法来了解 OpenMP 中的一些基本语句用法。

1．对 for 循环的并行化

在 C 程序中固有的并行操作常常以 for 循环的形式表现。OpenMP 可以方便地对其进行一个 for 循环的迭代被并行地执行。例如，考察下面的循环。

```
for ( i= first; i<size; I += prime ) marked [i] = 1
```

很明显，在这个循环的每次迭代之间不存在相关性。我们怎样才能把它转化为并行循环呢？在 OpenMP 中，只需要简单的告诉编译器这个 for 循环可以被并行地执行；而编译器会负责生成派生进程和调度并行迭代的代码，并且把循环的迭代分配给线程。

（1）parallel/parallel for 编译指导语句

C 或者 C++语言编译器指导语句在英语中记作 pragma。编译指导语句起着与编译器交互信息的作用。它提供的信息不是必需的，因此编译器可以在忽略它的情况下仍然生成正确的目标程序，但是这些信息无疑可以辅助编译器进行程序优化。编译指导语句在 C 或者 C++程序中的文法如下：

```
# pragma omp < rest of pragma >
```

使用这条语句可以写一个非常简单的并行程序如下：

```
#include <stdio.h>
#include <omp.h>
```

```
int main()
{
#pragma omp parallel
    {
    printf("Hello World \n");
}
}
```

该并行程序中的每一个线程将在标准输出上打印出字符串。当程序运行时，OpenMP 运行时系统创建大量的线程，每个线程将在并行构造中执行指令。如果程序员没有指定需要创建的线程数目，则将使用一个默认的数目。

对于 parallel for 语句来说，当这一行下面紧跟着一个 for 循环的时候，它将指示编译器把 for 循环并行化：

```
# pragma omp parallel for
for ( i = first; i<size; i += prime ) marked [i] = 1;
```

在该 for 循环中，控制子句具备规范格式并且在循环体中不存在提前退出，于是编译器可以生成让循环迭代并行执行的代码。

在 for 循环并行执行的过程中，主线程创建若干派生线程，所有这些线程协同工作共同完成循环的所有迭代。每个线程有各自的执行现场，也就是上下文。执行现场包括静态变量，堆中动态分配的数据结构已经运行时堆栈中的变量。

执行现场包括线程本身的运行时堆栈，这个堆栈保存着调用函数的框架信息。其他变量或者是共享的，或者是私有的。共享变量在所有线程的执行现场中的地址都是相同的，所有线程都可以对共享变量进行访问。私有变量在各个线程的执行现场中的地址不同，一个线程可以访问它自己的私有变量，但不能访问其他线程的私有变量。在 parallel for 编译指导语句中，变量默认设置为共享，而循环编号变量除外，它是私有变量。

运行时系统是如何获得所需创建的线程个数呢？环境变量 OMP_NUM_THREADS 的值提供了代码并行区的默认线程数。

另外一种策略是将线程数设置为多处理器的 CPU 个数。为此，有相应的 OpenMP 函数。

（2）omp_get_num_procs 函数

函数 omp_get_num_procs 返回并行程序中的可用的物理处理器的个数。它返回一个整数，有可能小于多处理机的处理器的物理个数，这取决于运行时系统所允许的进程可进行的处理器访问。

（3）omp_set_num_threads 函数

函数 omp_set_num_procs 用参数值来设置代码并行区中活动的线程个数。函数的声明如下：

void omp_set_num_threads(int n)

这个函数可以在程序的多处调用，从而可以根据并行的粒度大小或者代码段的其他特点来设置动态的并行度。该函数可以与 omp_get_num_procs 函数结合使用，直接把当前可用的处理器数目作为所用设置的线程数目：

```
int t;
…
tomp_get_num_procs ();
omp_set_num_threads(t)
```

2. 私有变量的处理

我们先来看一个较为复杂的循环结构。

```
for ( i = 0 ;   i < M ; i++ )
    for ( j = 0 ; j < N ; j++ )
    …
```

这种结构中，两个循环都可以并行地执行。如果将内层循环并行化，那么程序将在外重循环的每次迭代中都进行派生和会合操作。而如果对外层循环进行并行化，将仅仅引入一次派生——会合操作。

可以简单地指导编译器对变量 *i* 索引的循环进行并行化。然而，必须注意各个并行线程所访问的变量。按照默认情况，所有的变量将被设定为共享的，除了循环的索引变量 *i*。这样做可以使得线程之间相互通信较为简便，但是有可能导致一些问题出现。

接下来仔细分析当多个线程对循环的各次迭代进行并行处理时的具体情形。程序本想使针对 *i* 值的每个线程遍历 *j* 的 *N* 各取值。试想一下这样的情形，多个线程试图并行执行以 *i* 编号的不同的循环迭代，对于每个以 *i* 编号的外层循环迭代，我们希望每个线程能做完以 *j* 编号的 *N* 个内层循环迭代。然而，所有的线程都试图对同一个共享的变量 *j* 初始化和进行增量运算——于是很有可能某些线程不能完整地执行所有 *N* 个迭代。

而解决的办法非常明显，那就是将 *j* 设为私有变量。

（1）private 子句

子句是编译指导语句可选的附加部分。Private 子句指导编译器将一个或若干个变量私有化。语法如下：

```
private ( < variable list>)
```

指导语句告诉编译器为每个执行这条指导语句的代码段的线程分配一个该变量的私有副本。一个使用 private 子句的双层嵌套循环的 OpenMP 实现如下所示：

```
#pragma omp parallel for private ( j )
for ( i = 0 ;   i < M ; i++ )
    for ( j = 0 ; j < N ; j++ )
    …
```

变量 *j* 的私有副本只有在 for 循环内部才能访问到。在循环的入口和出口处该变量都是未定义的。即使 *j* 在并行的 for 混淆区之前已经被提前赋值了，也没有一个线程能够访问到那个值。同样的，无论在并行执行 for 循环时线程为 *j* 赋了什么值，共享的 *j* 值不会受到影响；另一方面，当进入并行结构的时候一个私有变量的默认值为未定义，当并行结构结束的时候这个变量的值也是未定义。

（2）firstprivate 子句

有时我们又希望一个私有变量继承共享变量的值。例如下面的代码段：

```
x[0] = complex_function() ;
for ( i = 0 ; i < n ; i ++ ) {
for   ( j = 1 ; j < 4 ; j ++ )
x[j] = g ( i ,x[j－1] );
answer [ i] = x [1]  －  x [3];
```

```
}
```

假设函数 *g* 没有副作用，当把 *x* 设为私有变量的时候，我们可以让每个外层循环的迭代被并行执行。然而，x[0]在外层 for 循环之前已经被初始化了并且在内存循环的第一次迭代中被引用了。如果把 x[0]的初始化移到外层 for 循环里是不现实的，因为那样做会非常耗时。不过，我们可以使用 firstprivate 语句让每个线程的数组元素 x [0]的私有副本继承在主线程中所对应的共享变量被赋的值。该语句的语法如下：

```
firstprivate (< variable list>)
```

它将指导编译器在进入循环时创建各个私有变量，并使得它们与由主线程所控制的相应变量有相同的值。下面是正确使用该子句的并行循环代码：

```
x[0] = complex_function();
    #pragma omp parallel for private (j) firstprivate (x)
    for (i = 0 ; i< n ; i ++){
    for (j = 1 ; j< 4 ; j ++)
    x[j] = g (i ,x[j－1]);
    answer [ i] = x [1]－x [3];
}
```

在 firstprivate 变量表中，各个变量的值将在线程创建时初始化，而不是在进行每次迭代时初始化。如果某个线程执行了该循环的多次迭代，并对其中的某个变量进行了改动，那么之后的各次迭代所得到的该变量的值将是改变后的值。

（3）lastprivate 子句

循环的顺序末次迭代是指当循环被串行执行时的最后一次迭代。lastprivate 子句将指示编译器产生可以在循环的并行执行结束时将其私有变量复制回主线程的对应变量的代码。

例如，如果将下面的代码并行化：

```
for (i = 0 ; i < n ; i ++)
{    x[0] = 1.0;
    for   (j = 1;j<4;j++)
        x[j] = x[j－1]×(i+1);
        sum of powers[i] = x[0]+ x[1]+ x[2]+ x[3];
} n cubed = x [3];
```

在这个循环串行执行到最后一个循环时，x[3]将被赋值为 *n*3。为了使这个值能在 for 循环外访问到，必须将 *x* 声明为 lastprivate 类型的变量。该代码段的并行版本如下：

```
#pragma omp parallel for private (j) lastprivate (x)
for (i = 0;i<n;I++)
{    x[0] = 1.0;
    for   (j = 1;j<4;j++)
        x[j] = x[j－1]×(i+1);
        sum of powers[i] = x[0]+ x[1]+ x[2]+ x[3];
} n cubed = x [3];
```

一个 parallel for 编译指导语句可以同时包含 firstprivate 和 lastprivate 子句。如果该指导语句包含了这两个子句，那么这两个子句中也可以包含同样的元素。

3. 临界区

先来考察一段通过矩形法则的数值积分方法来估算π值的 C 程序。

```
double area,pi,x;
int i,n;
…
area = 0.0;
for (i = 0;i<n;i++)
  {    x = (i+0.5) / n;
area+= 4.0 / (1.0+ x×x);
}
pi=area/n;
```

这个例子与前面所举的 for 循环并不相同，这次的循环不同次的迭代之间并不相互独立。每次迭代都会读取并更新变量 area 的值。如果简单地进行并行化循环：

```
double area ,pi ,x;
int i ,n;
…
area ＝ 0.0；
#pragma omp parallel for private (x)
for (i = 0;i<n;i++)
{      x = (i+ 0.5) / n;
area+=   4.0 / (1.0+ x×x);/*竞争条件*/
}
pi = area / n;
```

这样做很可能得不到正确的结果，因为赋值操作并不能保证原子性。这是一种竞争状况，在这种情况中由于多个线程访问共享变量时，计算会呈现非确定性的特征。设想以下情况，线程 *A* 和线程 *B* 在同时地进行同一循环的不同迭代。线程 *A* 读取了当前的 area 变量的值，并且计算 area 的值。在它将该计算结果写回时，线程 *B* 读取了 area 的值。然后线程 *A* 将 area 的值进行更新。线程 *B* 之后计算 area 的值并将结果写回。这时 area 的值就是错误的。解决这种错误情况的办法就是把 area 变量的读取和更改必须放在同一个临界区内部，同一时刻仅仅只能有一个线程执行这里的代码。

critical 编译指导语句，在 OpenMP 中我们可以使用下面的编译指导语句来定义一个代码临界区：

```
#pragma omp critical
```

该编译指导语句将告诉编译器生成相应代码，使得试图执行该段代码的线程之间互斥。上面的代码可以这样修改：

```
double area,pi,x;
int i,n;
…
```

```
area ＝ 0.0；
#pragma omp parallel for private (x)
for (i = 0;i<n;i++)
{      x=(i+ 0.5) / n;
#pragma omp critical
area+= 4.0 / (1.0+ x×x);/*竞争条件*/
}
pi = area / n;
```

这样做 C/OpenMP 程序段就能产生正确的结果。for 循环的各个迭代将在不同的线程中完成，而同一时刻只能有一个线程执行赋值语句并更新 area 的值。

总之，除了编译器命令外，OpenMP 还提供了一组运行时的库例程以及相应的环境变量。它们用来控制和查询并行执行环境，提供通用的加锁功能，设置执行方式等。

9.3.3 分布存储系统并行编程

对于分布存储系统来说，单一编址空间的假设不成立，组成系统的各计算机有自己的处理器和本地主存储器，不能相互访问各自的主存储器，只能通过传递消息来进行交互。这种分布存储并行计算机上开发并行算法一般基于 Ian Foster 所提出的任务/通道模型。

任务/通道模型将并行计算表示为一系列任务，任务之间通过使用通道发送消息进行相互交互。这里的任务指的是一个程序、其本地存储以及一组 I/O 端口。本地存储包含了程序的指令以及其私有数据，任务能够通过输出端口将其本地数据值发送给其他任务。反过来，任务也能够通过输入端口接受来自其他任务的数据值。通道是连接一个任务的输出端口与另一任务输入端口的一个消息队列，数据值按其在通道另一端的输出端口所放置的次序出现在输入端口里。显然，只有在通道一端发送某个数据之后才能在通道的另一端接受该数据。如果一个任务试图在输入端接受某个值而没有任何可获得的值的话，则该任务必须等待直到该值出现为止。此时，称接受值的任务已被阻塞。反之，发送消息的进程从不阻塞，即使在这之前沿着同一通道发送的消息仍未被接受。也就是说，任务/通道模型中发送是异步操作，接受却是同步操作。

Ian Foster 提出了一个分为四步的并行算法设计过程。该过程将与机器相关的考虑延后，以促进可扩展并行算法的开发。该设计方法中的四步分别被称为划分、通信、聚集和映射。在开始一个并行算法设计的时候，常希望发现尽可能多的并行性。划分是将计算和数据进行分片的过程，每一片作为一个原始任务，分片的目的就是尽可能多地识别原始任务。在识别出原始任务之后，下一步就是确定它们之间的通信模式。并行算法有两种通信模式：局部通信和全局通信。当一个任务为执行某个计算而需要来自少数几个其他任务的值的时候，需要创建从供应数据的任务到消费数据的任务的一条通道，这是局部通信的例子。相反，全局通信存在于为执行计算而需要大量原始任务贡献数据的时候，例如计算原进程所拥有值的和。在这前两步中，我们的注意力放在尽可能多地识别并行性。此时，我们的设计很可能无法在真实的并行计算机上高效执行，如任务数超过了处理器数目几个数量级的情况。所以在设计的后两步中，需要考虑目标体系结构，考虑怎样将原始任务合并成大的任务并映射到物理处

理器上以减少并行开销的量。聚集是为改善性能或简化编程而将任务合并为大的任务的过程。聚集的结果是给每个处理器分配一个任务，目标是降低通信开销，维持并行设计的可扩展性和减少软件工程上的开销。最后一步映射是将任务分配给处理器的过程。映射的目标是最大化处理器的利用率和最小化处理器之间的通信。

通过在 C 或者 Fortran 语言中增加进程间消息传递函数，可以完成大多数的并行程序设计。MPI 标准是最流行的并行编程消息传递规范，几乎所有商业的并行机都支持它，同时也有众多支持 MPI 标准的开放软件库可供使用。消息传递编程模型和任务/通道模型相似。消息传递模型假设底层是一组处理器，每个处理器有自己的本地内存，并且通过互连网络实现与其他处理器的消息传递。任务/通道模型中的任务对应着消息传递模型中的进程，互连网络的存在就意味着每两个处理器之间都有一个管道连通，也就是说每个处理器都可和其他任意一个处理器进行通信。

MPI 标准目前主要有两个版本。最初的草案于 1991 年 11 月推出，直到 1994 年 4 月才完成了对该草案的细化，于同年 5 月推出了 1.0 版。随后，改进标准的工作继续进行，于 1997 年 4 月形成了 MPI-2。MPI 是一个库而不是一门语言，对它的使用必须和特定的语言结合起来进行。在 MPI-1 中明确提出了 MPI 与 FORTRAN 77 与 C 语言的绑定，并且给出了通用接口和针对 FORTRAN 77 与 C 语言的专用接口说明。在 MPI-1 中，共有 128 个调用接口，在 MPI-2 中有 287 个。MPI 比较庞大，完全掌握这么多的调用对于初学者来说比较困难。但从理论上来看，MPI 所有的通信功能可以用它的 6 个基本的调用来实现。下面分别对这 6 个调用组成的子集来做一初步介绍。

（1）MPI 调用的参数说明

在具体介绍调用之前，首先来看 MPI 是如何对其 FORTRAN 77 和 C 的调用进行说明的。对于有参数的 MPI 调用，MPI 首先给出一种独立于具体语言的说明，对各个参数的性质进行介绍，然后再给出它相对于 FORTRAN 77 和 C 的原型说明。MPI 对参数说明的方式有 3 种，分别是 IN、OUT 和 INOUT。它们的含义分别是。

IN：调用部分传递给 MPI 的参数，MPI 除了使用该参数外不允许对这一参数作任何修改；

OUT：MPI 返回给调用部分的结果参数，该参数的初始值对 MPI 没任何意义；

INOUT：调用部分首先将该参数传递给 MPI，MPI 对这一参数引用、修改后，将结果返回给外部调用，该参数的初始值和返回结果都有意义。

在本节后面所有的 MPI 调用说明中，首先给出的是 MPI 调用不依赖任何语言的说明，然后给出这个调用的标准 C 版本。

（2）MPI 初始化

```
MPI_INIT()
int MPI_Init(int *argc,char ***argv)
```

MPI_INIT 是 MPI 程序的第一个调用，它完成 MPI 程序所有的初始化工作。所有 MPI 程序的第一条可执行语句都是该条语句。

（3）MPI 结束

```
MPI_FINALIZE()
int MPI_Finalize(void)
```

MPI_FINALIZE()是 MPI

程序的最后一个调用，它结束 MPI 程序的运行，它是 MPI 程序的最后一条可执行语句，否则程序的运行结果是不可预知的。

（4）当前进程标识

MPI_COMM_RANK(comm,rank)

IN comm 该进程所在的通信域(句柄)

OUT rank 调用进程在 comm 中的标识号

int MPI_Comm_rank(MPI_Comm comm,int *rank)

这一调用返回调用进程在给定的通信域中的进程标识号，有了这一标识号，不同的进程就可以将自身和其他的进程区别开来，实现各进程的并行和协作。

（5）通信域包含的进程数

MPI_COMM_SIZE(comm,size)

IN comm 通信域(句柄)

OUT size 通信域 comm 内包括的进程数(整数)

这一调用返回给定的通信域中所包括的进程的个数，不同的进程通过这一调用得知在给定的通信域中一共有多少个进程在并行执行。

（6）消息发送

MPI_SEND(buf,count,datatype,dest,tag,comm)

IN buf 发送缓冲区的起始地址(可选类型)

IN count 将发送的数据的个数(非负整数)

IN datatype 发送数据的数据类型(句柄)

IN dest 目的进程标识号(整型)

IN tag 消息标志(整型)

IN comm 通信域(句柄)

int MPI_Send(void * buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm)

MPI_SEND 将发送缓冲区中的 count 个 datatype 数据类型的数据发送到目的进程，目的进程在通信域中的标识号是 dest，本次发送的消息标志是 tag，使用该标志，就可以把本次发送的消息和本进程向同一进程发送的其他消息区别开来。

MPI_SEND 操作指定的发送缓冲区是由 count 个类型为 datatype 的连续数据空间组成，起始地址为 buf。其中 datatype 数据类型可以是 MPI 的预定义类型，也可以是用户自定义的类型。通过使用不同的数据类型调用 MPI_SEND，可以发送不同类型的数据。

（7）消息接受

MPI_RECV(buf,count,datatype,source,tag,comm,status)

OUT buf 接收缓冲区的起始地址（可选数据类型）

IN count 最多可接受的数据的个数（整型）

IN datatype 接收数据的数据类型（句柄）

IN source 接收数据的来源即发送数据的进程的进程标识号

IN tag 消息标识，与相应的发送操作的消息标识相匹配

IN comm 本进程和发送进程所在的通信域

OUT status 返回状态（状态类型）

int MPI_Recv(void* buf,int count,MPI_Datatype datatype,int source,int tag,MPI_Comm comm,MPI_Status * status)

MPI_RECV 从指定的进程 source 接受消息，并且该消息的数据类型、消息标识和本接收进程指定的 datatype 和 tag 相一致，接收到的消息所包含的数据元素的个数最多不能超过 count。接收缓冲区是由 count 个类型为 datatype 的连续元素空间组成，由 datatype 指定其类型，起始地址为 buf。接收到消息的长度必须小于或等于接收缓冲区的长度，否则溢出。其中 datatype 数据类型可以是 MPI 的预定义类型，也可以是用户自定义的类型。通过指定不同的数据类型调用 MPI_RECV，可以接收不同类型的数据。

接下来我们看具体怎么运用上述函数来进行编程。先来看一个显示“Hello World”的 MPI 程序：

```
#include<mpi.h>
#include<stdio.h>
#include<math.h>
void main(argc,argv)
int argc;
char *argv[];
{
    int myid,numprocs;
    int namelen;
char processor_name[MPI_MAX_PROCESSOR_NAME];
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Get_processor_name(processor_name,&namelen);
fprintf(stderr,"Hello World!Porcess&d of %d on %s \n",
myid,numprocs,processor_name);
MPI_Finalize();
}
```

第一部分，首先要有 MPI 相对于 C 实现的头文件 mpi.h。

第二部分，定义程序中所需要的与 MPI 有关的变量。

MPI_MAX_PROCESSOR_NAME 是 MPI 预定义的宏，即某一 MPI 的具体实现中允许机器名字的最大长度，机器名放在变量 processor_name 中，整型变量 myid 和 numprocs 分别用来记录某一个并行执行进程的标识和所有参加计算的进程的个数；namelen 是实际得到的机器名字的长度。

第三部分，MPI 程序的开始和结束必须是 MPI_Init 和 MPI_Finalize，分别完成 MPI 程序的初始化和结束工作。

第四部分，MPI 的程序体，包括各种 MPI 过程调用语句和 C 语句。MPI_Comm_rank 得到当前正在运行的进程标识号，放在 myid 中；MPI_Comm_size 得到所有参加运算的进程的个数，放在 numprocs 中；MPI_Get_processor_name 得到本进程运行的机器的名称，结果放在

processor_name 中，它是一个字符串，而该字符串的长度放在 namelen 中；fprintf 语句将本进程的标识号、并行执行的进程的个数以及本进程所运行的机器的名字打印出来，与一般的 C 程序不同的是这些程序体中的执行语句是并行执行的，每一个进程都要执行。

再看一个比较复杂一点的 MPI 程序。该程序是计算 $\sum foo(i)$ 的一个 SPMD(Single Process,Multiple Data)程序：

```
#include<mpi.h>
int foo(i) int i;{…}
main (argc,argv)
int argc;
char * argv[];
{     int i,tmp,sum=0,group_size,my_rank,N;
      MPI_Init(&argc,&argv);
      MPI_Comm_size(MPI_COMM_WORLD,&group_size);
      MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
      if(my_rank= =0){
            printf("Enter N: ");scanf("%d",&N);
            for(i=1,i<group_size;i++)
S1:               MPI_Send(&N,1,MPI_INT,I,I,MPI_COMM_WORLD);
            for(i=my_rank;i<N;i=i+group_size)
                  sum=sum+foo(i);
            for(i=1,i<group_size;i++){
S2:               MPI_Recv(&tmp,1,MPI_INT,I,I,MPI_COMM_WORLD,&status);
                  sum=sum+tmp;
    }
    printf("\n the result=5d",sum);
            }
            else{/* if my_rank!=0*/
S3:               MPI_Recv(&N,1,MPI_INT,0,I,MPI_COMM_WORLD,&status);
                  for(i=my_rank;i<N;i=i+group_size)
                  sum=sum+foo(i);
S4:               MPI_Send(&sum,1,MPI_INT,0,I,MPI_COMM_WORLD);
}
      MPI_Finalize();
}
```

在该程序中，我们可看到在调用任何其他的 MPI 例程之前，在每个进程中必须先调用例程 MPI_INIT 以初始化 MPI。然后通过调用 MPI_Comm_size 和 MPI_Comm_rank 例程可找到缺省的组的大小和每个进程的等级。进程间的相互通信是通过调用 MPI_Send 和 MPI_Recv 例程来实现消息的发送和接收。最后，当不再需要 MPI 函数时，每个进程就调用 MPI_Finalize 终止 MPI 环境。这 6 个例程就形成了在 MPI 中编写完整消息传递程序的一个最小集。程序

中的 MPI_COMM_WORLD 由所有进程组成，是一个缺省进程组，缺省进程组大小等于节点数。当并行进程装入时就创建了所有进程，它们一直存在直到整个程序终止。对于进程 $k(k=0,1,\cdots,n)$，MPI_Comm_size 例程将在 group_size 中返回 n，而 MPI_Comm_rank 将在 my_rank 中返回 k。进程 0 完成所有的 I/O 操作。进程 0 首先从用户处读得值 N，然后将此值发送给 for 循环中所有其他进程。接下来进程 0 计算它的工作份额 sum = $foo(0) + foo(n) + foo(2n) + \cdots$。此后，进程 0 从其他进程处接收 $n-1$ 个部分和，并将它们累加成一个总和，再打印输出。对于其余 $n-1$ 个进程中的每一个，要完成如下操作：进程 $k(0<k<n)$接收由进程 0 发送来的 N，计算它的工作份额 $sum=foo(k)+ foo(n+k)+foo(2n+k)+\cdots$，然后将此部分和发送给进程 0。进程 0 中的发送语句 $S1$ 与进程 k 中的接收语句 $S3$ 相匹配，而 $S4$ 则与 $S2$ 相匹配。

总之，MPI 的核心构造是消息传递：一个进程将信息打包成消息，并将该消息发送给其他进程。但是，MPI 包含比简单的消息传递更多的内容。MPI 包含一些例程，这些例程可以同步进程、求分布在进程集中的数值的总和、在一个进程集中分配数据，以及实现更多的功能。MPI 的底层构造与并行计算机的 MIMD 模型紧密结合，使得 MPI 程序员能够精确地控制并行计算的展开方式，并编写高效的程序，而且能编写可移植的并行程序。

习　　题

1. 试画出 SISD、SIMD、MISD 和 MIMD 的结构框图，并简述其原理。

2. 试比较 SIMD 和 MIMD 并行执行方式的优缺点。

3. 如果 CPU 的速度每 5 年增加 10 倍，那么性能翻倍需要多长时间？

4. 今天所有的超级计算机是否都是并行计算机？所有并行计算机是否都是超级计算机？解释你的回答。

5. 当处理机个数 N 不是 2 的幂时必须对广播算法进行修改，证明修改后算法的运算时间和性能保持不变。

6. 对输入序列 S={7,6,5,4,3,2,1}运用一维线性模型上的并行排序算法排序，试画出排序过程示意图。

7. 试讨论一个系统的效率是否能大于 100％？

8. 物理上构造出消息传递多计算机和共享存储器多处理机两者混合的系统是可能的。请写出一份论证报告，说明如何加以实现以及该混合系统相对于纯消息传递系统和纯共享存储系统的优越性体现在什么地方？

9. MPICH 是一种最重要的 MPI 实现，可以免费从 http://www-unix.mcs.anl.gov/mpi/mpich 取得，请下载并安装；然后编译运行“Hello World”程序。

10. OpenMP 官方网站的地址试 www.OpenMP.org。可以从该站点下载 C/C++和 Fortran 版本的 OpenMP 规范与编译器。试编译运行“Hello World”程序。

附录 1　习题解答算法提要

习题 1

3．分别求出以下程序段所代表算法的时间复杂度。

（1）$O(n^2)$。

解：因 $s=1+2+\cdots+n=n(n+1)/2$

（2）$O(n^2)$。

解：设 $n=2u+1$，语句 $m=m+1$ 的执行频数为

$s=1+1+2+2+3+3+\cdots+u+u=u(u+1)=(n-1)(n+1)/4$

设 $n=2u$，语句 $m=m+1$ 的执行频数为

$s=1+1+2+2+3+3+\cdots+u=u^2=n^2/4$

（3）$O((n+1)!)$。

解：因 $s=1+2\times2!+3\times3!+\cdots+n\times n!=(n+1)!-1$

（4）$O(n\sqrt{n})$。

解：因 a 循环计为 n 次；对每一个 a,b 循环 50 次；对每一个 b,k 循环 $\sqrt{b}/2$ 次。因而 k 循环体的执行次数 s 满足

$$s<25(\sqrt{99}+\sqrt{199}+\cdots+\sqrt{100n-1})<250(1+\sqrt{2}+\cdots+\sqrt{n})$$

$$<250\cdot\frac{4n+3}{6}\sqrt{n}<250n\sqrt{n}$$

6．试把例 1.1 中的欧几里德算法描述成函数形式，并设计程序通过函数调用实现求 n 个整数的最大公约数。

（1）欧几里德算法描述成函数形式

```
long gcd(long a,long b)
{ long c,r;
        if(a<b)
     {c=a;a=b;b=c;}          /*  交换 a,b ,确保 a>b*/
   r=a%b;
   while(r!=0)
     { a=b;b=r;              /*  实施"辗转相除" */
       r=a%b;
```

```
        }
return b;
}
```

（2）调用函数的主程序

```
/* 求 n 个整数的最大公约数*/
#include<stdio.h>
void main()
{ int k,n;
  long x,y,m[100];
  printf("请输入整数个数 n: ");
  scanf("%d",&n);
  printf("请依次输入%d 个整数: ",n);
  for(k=0;k<=n-1;k++)
      { printf("\n 请输入第%d 个整数：  ",k+1);
       scanf("%ld",&m[k]);
      }
  x=m[0];
  for(k=1;k<=n-1;k++)
      { y=m[k];
        x=gcd(x,y);
      }
  printf("(%ld",m[0]);
  for(k=1;k<=n-1;k++)
        printf(",%ld",m[k]);
  printf(")=%ld\n",x);
}
```

习题 2

1．正整数 n 的所有小于 n 的正因子之和若等于 n 本身，称数 n 为完全数。例如，6 的正因子为 1,2,3,而 6=1+2+3,则 6 是一个完全数。试求指定区间内的完全数。

算法设计。

对指定区间中的每一个整数 a 实施穷举判别。根据完全数的定义,为了判别整数 a 是不是完全数，用试商法找出 a 的所有小于 a 的因子 k。显然，$1 \leqslant k \leqslant a/2$。注意到 1 是任何整数的因子，先把因子 1 定下来，即因子和 s 赋初值 1。然后设置 k 从 2～$a/2$ 的循环,由表达式 $a\%k=0$ 判别 k 是否是 a 的因子，并求出 a 的因子累加和 s。

最后若满足条件 $a=s$ 说明 a 是完全数，作打印输出，把 n 的因数从 1 开始，由小到大排列，写成和式。

穷举的算法描述：

```
scanf("%ld,%ld",&x,&y);
      for(a=x;a<=y;a++)
       {s=1;n=0;
        for(k=2;k<=a/2;k++)                    /* 试商寻求 a 的因数 k */
if(a%k==0)
              {s=s+k;n++;c[n]=k;}              /* a 的因数 k 赋值给 c 数组 */
       if(a==s)
{printf("%ld=1",a);                  /* 从小到大打印因数之和 */
            for(i=1;i<=n;i++) printf("+%d",c[i]);
printf("\n");
}
       }
```

2．一个世纪的 100 个年号全为合数（即不存在一个素数），该世纪称为合数世纪。设计程序探索最早的合数世纪。

算法设计：

应用穷举搜索，由于具体的搜索范围难以确定，我们约定在[21,200000]这一区间中穷举。当找到（即最早的）后退出循环，不会增加无效循环。若在这一区间未找到，可试把搜索区间范围进一步扩大。

设置 *a* 循环在约定区间穷举世纪，设置 *b* 循环表征 *a* 世纪的 50 个奇数年号（偶数年号无疑均为合数）。用变量 *s* 统计这 50 个奇数中的合数个数。对于 *a* 世纪,若 *s*=50，即 50 个奇数都为合数，找到 *a* 世纪为最早的合数世纪,打印输出后退出循环结束。

穷举算法描述：

```
for(a=21;a<200000;a++)                    /* 在约定区间内穷举世纪 */
    {s=0;
     for(b=a*100－99;b<=a*100－1;b+=2) /* 穷举每个奇数年号 b */
        {x=0;
         for(k=3;k<=sqrt(b);k+=2)
            if(b%k==0)
                {x=1;break;}
         s=s+x;}                              /* 年号 b 为合数时,x=1,s 增 1 */
     if(s==50)                                /*  s=50，即有 50 个合数  */
        {printf("最早出现的合数世纪为:%ld  世纪!\n",a);break;}
     }
```

3．圆圈循环报数问题称为 Joseph 问题。有 *n* 个小朋友按编号顺序 1,2,…,*n* 逆时针方向围成一圈。从 1 号开始按逆时针方向 1,2,…,*m* 报数,凡报数 *m* 者出列（显然,第一个出圈的为编号 *m* 者）。

问：最后剩下一个未出圈者的编号是多少？指定的第 *p* 个出圈者的编号为多少？

算法设计。

设置数组 *a*(*n*)，每一数组元素赋初值 1。每报数一人，和变量 *s* 增 1。当加 *a*(*i*)后和变量

s 的值为 *m* 时，*a*(*i*)=0，标志编号为 *i* 者出圈，设置 *k* 统计出圈人数。同时，*s*=0 并重新逆时针方向后报数累加。

当出圈人数 *k* 为指定的 *p* 时，*x*=*i*,*x* 即为第 *p* 个出圈者的编号。

当出圈人数 *k* 达 *n*−1 时，即未出圈者只剩 1 人，终止报数，打印最后剩下者的编号。因而条件穷举循环的条件为 *k*<*n*−1。

穷举算法描述：

```
scanf("%d,%d,%d",&n,&m,&p);
   for(i=1;i<=n;i++) a[i]=1;
   k=0;s=0;x=0;i=0;
   while(k<n−1)                          /* 按出圈人数穷举 */
{i++;
 if(i>n) i=1;                            /* 一圈报完接着下一圈 */
s=s+a[i];                                /* 按逆时针顺序报数 */
        if(s==m) {a[i]=0;s=0;k++;}        /* 报到指定的 m 者出圈赋 0 */
        if(k==p && x==0) x=i;             /* 第 p 个出圈者 i 号赋给 x */
       }
      printf("报数最后剩下一个人编号为:");
      for(i=1;i<=n;i++)
         if(a[i]!=0) printf("%d   ",i);   /* 打印剩下最后一人未出列者 */
      printf("\n 第%d 个出列者编号是:%d 号.\n",p,x);
```

4．求所有 *m* 位巧妙平方数：在 1,2,…,9 这九个数字中选 *m* 个，组成没有重复数字的平方数（整数 *m* 从键盘输入，1≤*m*≤9）。

算法设计：

计算最小的 *m* 位数 10^(*m*−1)开平方取整数 *t*1，最大的 *m* 位数（10^*m*）−1 开平方取整数 *t*2，以 *t*1+1，*t*2 作为循环的初值与终值设置穷举 y 循环。检验 *m* 位平方数 *f*=*y***y*,经 *m* 次求余分离 *f* 的每一位数字 *k*，若 *k*=0，即 *f* 含数字 0，返回；设置 *b* 数组，应用 *b*(*k*)统计数字 *k* 的个数，若 *b*(*k*)>1，即 *f* 中含有重复数字，返回。

经检验若 *f* 不含数字 0,也没有重复数字，则打印输出。

穷举算法描述：

```
scanf("%d",&m);
 n=0;t=1;
 for(y=1;y<=m−1;y++)
     t=t*10;
 t1=sqrt(t);t2=sqrt(10*t);
 for(y=t1+1;y<=t2;y++)
     {f=y*y;d=f;                          /* 数 f 为 m 位平方数 */
      for(i=1;i<=9;i++) b[i]=0;
      for(x=0,i=1;i<=m;i++)
          {w=f/10;k=f%10;
```

```
            if(k==0) {x=1;break;}        /* 含数字 0 则记 x=1 */
            else b[k]=b[k]+1;            /* 统计数字 k 的个数 */
            f=w;
           }
       for(i=1;i<=9;i++)
          if(b[i]>1) {x=1;break;}        /* 若有重复数字，则记 x=1 */
       if(x==0)
          {n++;printf("%ld=%d^2    ",d,y);
           if(n%3==0) printf("\n");
          }
     }
```

5．求 *n* 个 0 与 *m* 个 1 组成的排列（*n*、*m* 从键盘输入）。

算法设计。

设置 *a* 数组，*m*+*n* 个元素 *a*(*i*)在 0 与 1 中取值。

若 i<n+m，则 i++;a[i]=0;

若 a[i]==1 且 i>1，则 i—; 向前回溯;

若 a[i]==1 且 i==1，则返回;

若 i=m+n,变量 h 统计 0 的个数。h=n 时输出一个排列。

算法描述：

```
printf(" input n: "); scanf("%d",&n);
printf(" input m: "); scanf("%d",&m);
i=1;a[1]=0;
while (1)
  {if(i==n+m)
     {h=0;
      for(j=1;j<=n+m;j++)
          if(a[j]==0) h++;
      if(h==n)                              /* 判别是否有 n 个零 */
        {s++;
          for(j=1;j<=n+m;j++)
              printf("%d",a[j]);
          printf("    ");
          if(s%5==0) printf("\n");
        }
     }
   if(i<n+m)
      {i++;a[i]=0;continue;}
   while(a[i]==1 && i>1) i—;                /* 向前回溯 */
   if(a[i]==1 && i==1) break;
```

```
        else a[i]=1;
    }
```

6．求序列中的最长不降子序列。

给定一个由 n 个正整数组成的序列，从中删除若干个数，使剩下的序列非降。探求并输出所有最长非降子序列。

算法设计。

设在序列中删除若干个数后剩下的数组成的非降序列的长度为 m，显然，$1 \leqslant m \leqslant n$。

设 b 数组存储 n 个整数，a 数组每一个整数的序号，$1 \leqslant a(i) \leqslant n$。

设置穷举 m 循环，m 从 n 开始递减 1 取值。对每一个 m，在 n 个数中取 m 个数，$a(1)$从 1 开始取值。约定 $a(1),\cdots,a(i),\cdots,a(m)$按递增顺序排列，$a(i)$后有 $m-i$ 个大于 $a(i)$的元素,其中最大取值为 n，显然 $a(i)$最多取 $n-m+i$。

当 $i<m$ 时，i 增 1，$a(i)$从 $a(i-1)+1$ 开始取值；直至 $i=m$ 时输出结果。

当 $a(i)=n-m+$i 时 $i=i-1$ 回溯，直至 $i=0$ 时结束。

若 $i=m$，即已取了 m 个数，测试这 m 个数组成的子序列是否存在有递增现象，若存在 b[a[k]]>b[a[j]]（a[k]<a[j]），则返回重取。否则，打印由 m 个整数组成的非降序列。因 m 从 n 开始递减 1 取值。最先出现的非降序列的长度即为最长非降序列的长度。

算法描述：

```
输入 n,b[1－n]。
for(s=0,m=n;m>=1;m−−)
    {a[1]=1;i=1;
    while(1)
      {if(i==m )                          /* 满足条件时输出解 */
          {for(t=1,k=1;k<=m−1;k++)
           for(j=k+1;j<=m;j++)
              if(b[a[k]]>b[a[j]]){t=0;break;}
           if(t)
            {s++; printf("\n %3d: ",s);
             for(j=1;j<=m;j++)
             printf("%d   ",b[a[j]]);
            }
          }
       else {i++; a[i]=a[i−1]+1; continue;}
       while(a[i]==n−m+i) i−−;                /* 返回调整 */
       if(i>0) a[i]++;
       else break;
      }
   if(s>0)
     {printf("\n     max=%d!",m);break; }
   }
```

7．对已知的 $2n$(n 从键盘输入)个数,确定这些数能否分成 2 个组,每组 n 个数,且每组数据

的和相等。

算法设计。

仿照上题设置 $n-1$ 重循环并不可取。我们可采用回溯法逐步实施调整。

对于已有的存储在 b 数组的 $2n$ 个数，求出总和 s 与其和的一半 $s1$（若这 $2n$ 个数的和 s 为奇数，显然无法分组）。把这 $2n$ 个数分成二个组,每组 n 个数。为方便调整，设置数组 a 存储 b 数组的下标值,即 $a(i)$:1～$2n$。

考察 $b(1)$所在的组，只要另从 $b(2)$～$b(2n)$中选取 $n-1$ 个数。即定下 $a(1)=1$, 其余的 $a(i)(i=2,\cdots,n)$在 2～$2n$ 中取不重复的数。因组合与顺序无关,不妨设

$$2 \leqslant a(2)<a(3)<\cdots<a(n) \leqslant 2n$$

从 $a(2)$取 2 开始，以后 $a(i)$从 $a(i-1)+1$ 开始递增 1 取值，直至 $n+i$ 为止。这样可避免重复。

当 $a(n)$已取值，计算 $s=b(1)+b(a(2))+\cdots+b(a(n))$，对和 s 进行判别：

若 $s=s1$，满足要求，实现平分。

若 $s\neq s1$，则 a(n)继续增 1 再试。如果 $a(n)$已增至 $2n$，则回溯前一个 $a(n-1)$增 1 再试。如果 $a(n-1)$已增至 $2n-1$，继续回溯。直至 $a(2)$增至 $n+2$ 时,结束。

二组均分问题并不总有解。有解时，找到并输出所有解；没有解时，显示相关提示信息“无法实现平分”。

算法描述：

输入 n 及 $2n$ 个不同的整数。

```
a[1]=1;i=2;a[i]=2;m=0;
 while(1)
   {if(i==n)
      {for(s=0,j=1;j<=n;j++) s+=b[a[j]];
       if(s==s1)                          /* 满足均分条件时输出 */
         {m++; printf("NO%d:   ",m);
          for(j=1;j<=n;j++) printf("%d    ",b[a[j]]);
          printf("\n 上述%d 个数之和为%d,实现二组均分.\n",n,s1);
          }
       }
    else {i++; a[i]=a[i-1]+1; continue;}
    while(a[i]==n+i) i--;                 /* 调整或回溯 */
    if(i>1) a[i]++;
    else break;
    }
```

习题 3

1．某石油公司计划建造一条由东向西的主输油管道。该管道要穿过一个有 n 个口油井的油田。从每口油井都要有一条输油管道沿最短路经（或南或北）与主管道相连。如果给定 n

口油井的位置，即它们的 x 坐标（东西向）和 y 坐标（南北向），应如何确定主管道的最优位置，即使各油井到主管道之间的输油管道长度总和最小的位置？证明可在线性时间内确定主管道的最优位置。

给定 n 口油井的位置,编程计算各油井到主管道之间的输油管道最小长度总和。

算法设计：

设 n 口油井的位置分别为, $p_i=(x_i,y_i),1\leqslant i\leqslant n$。由于主输油管道是东西向的，因此可以用主轴线的 y 坐标惟一确定其位置，主管道的最优位置 y 应使 $\sum_{i=1}^{n}d(y,y_i)$ 达到最小，其中，$d(y,y_i)=|y-y_i|$，所以 y_i(其中$1\leqslant i\leqslant n$) 的中位数 y_k 即位输油管道的最优解。

2．题目略。

算法设计。

所求的数 $S(n,m)$，满足如下递归式：

$$S(n,m)=mS(n-1,m)+S(n-1,m-1)$$

$$S(n,n+1)=0,S(n,0)=0,S(0,0)=0$$

```
void SNumber(int n, int m)
{    int i,j,min;
S[0][0]=1;
     for(i=1,i<=m;i++)S[i][0]=0;
     for(i=0,i<m,i++)S[i][i+1]=0;
     for(i=1,i<=m;i++){
        if(i<n)min=i; else min=n;
        for(j=1,j<=I;j++)
           S[i][j]=j*S[i－1][j]+S[i－1][j－1];
     }
}
```

3．在一个按照东西和南北方向划分成规整街区的城市里，n 个居民点散乱地分布在不同的街区中。用 x 坐标表示东西向，用 y 坐标表示南北向。各居民点的位置可以由坐标(x,y)表示。街区中任意 2 点(x_1,y_1)和(x_2,y_2)之间的距离可以用数值$|x_1-x_2|+|y_1-y_2|$度量。

居民们希望在城市中选择建立邮局的最佳位置，使 n 个居民点到邮局的距离总和最小。

给定 n 个居民点的位置,编程计算 n 个居民点到邮局的距离总和的最小值。

算法设计。

在平面上 2 点， $p(x_p,y_p),q(x_q,y_q)$ 之间的距离定义为

$$d(p,q)=|x_p-x_q|+|y_p-y_q|$$

设平面上的点 p_i 到邮局的长度是均等的，即分别带权 w=1/n，所以问题转化为求平面上一定点 p，使 $\sum_{i=1}^{n}w_id(p,p_i)$ 达到最小。

4．大于 1 的正整数 n 可以分解为：$n=x_1\times x_2\times\cdots\times x_m$。

例如，当 n=12 时，共有 8 种不同的分解式：

12=12；

12=6×2；

12=4×3；

12=3×4；

12=3×2×2；

12=2×6；

12=2×3×2；

12=2×2×3。

对于给定的正整数 n，编程计算 n 共有多少种不同的分解式。

算法设计：

对每个因子进行递归搜索。即：

```
void solve(int n)
{
    int i;
    if(n==1) total++;
    else for(i=2,i<=n,i++) if(n%i==0) solve(n/i);
}
```

习题 4

2．汉诺（Hanoi）塔问题是一个古典数学问题：在一个古塔内有 A、B、C，3 个座开始时 A 座自下而上、由大到小顺序放着 64 个圆盘。想把这 64 个盘子从 A 座移到 C 座，移动过程中可以借助 B 座，但每一次只能移动一个盘，且不允许大盘放在小盘的上面。

应用递推求解 n 个圆盘的移动次数。

算法设计。

当 n=1 时，只一个盘，移动一次即完成。

当 n=2 时，由于条件是一次只能移动一个盘，且不允许大盘放在小盘上面，首先把小盘从 A 座移到 B 座；然后把大盘从 A 座移到 C 座；最后把小盘从 B 座移到 C 座，移动 3 次完成。

设移动 n 个盘的汉诺塔需 $g(n)$次完成。首先将 n 个盘上面的 $n-1$ 个盘子借助 C 座从 A 座移到 B 座上，需 $g(n-1)$次；然后将 A 座上第 n 个盘子移到 C 座上（1 次）；最后，将 B 座上的 $n-1$ 个盘子借助 A 座移到 C 座上，需 $g(n-1)$次。因而有

递推关系：$g(n)=2*g(n-1)+1$

初始条件：$g(1)=1$。

3．输出集合 $\{2^x,3^y,5^z \mid x\geq 1,y\geq 1,z\geq 1\}$ 元素由小到大排列的幂序列第 n 项与前 n 项之和。

算法设计。

集合由 2、3、5 的幂组成，实际上给出的是 3 个递推关系。为了实现从小到大排列，设置 a、b、c 3 个变量，a 为 2 的幂，b 为 3 的幂，c 为 5 的幂，显然 $a\neq b\neq c$。

设置 k 循环（k=1,2,…,n，其中 n 为键盘输入整数），在 k 循环外赋初值：a=2;b=3;c=5;s=0;在 k 循环中通过比较赋值：

当 $a<b$ 且 $a<c$ 时，由赋值 $f(k)=a$ 确定为序列的第 k 项，累加到和变量 s 中；然后 $a=a\times2$，即 a 按递推规律乘 2, 为后一轮比较作准备；

当 $b<a$ 且 $b<c$ 时，由赋值 $f(k)=b$ 确定为序列的第 k 项，累加到和变量 s 中；然后 $b=b\times3$，即 b 按递推规律乘 3，为后一轮比较作准备。

当 $c<a$ 且 $c<b$ 时，由赋值 $f(k)=c$ 确定为序列的第 k 项，累加到和变量 s 中；然后 $c=c\times5$，即 c 按递推规律乘 5，为后一轮比较作准备。

递推描述。

```
a=x;b=y;c=m;
 for(k=1;k<=n;k++)
    {if(a<b && a<c)
       {f[k]=a;a=a*x;}              /* 用 2 的幂给 f[k]赋值 */
     else if (b<a && b<c)
          {f[k]=b;b=b*y;}           /* 用 3 的幂给 f[k]赋值 */
        else
          {f[k]=c;c=c*m;}           /* 用 5 的幂给 f[k]赋值 */
        }
```

在这一算法中，变量 a、b、c 是变化的，分别体现了 2、3、5 的幂。

4．已知递推数列：$a(1)=1,a(2i)=a(i),a(2i+1)=a(i)+a(i+1)$,($i$ 为正整数)，试求该数列的第 n 项。

算法设计。

该数列分项序号为奇或偶两种情况作不同递推，所得数列呈大小有规律的摆动。

设置 a 数组，赋初值 $a(1)=1$。根据递推式，在循环中分项序号 i(2～n) 为奇或偶作不同递推：

i%2=0(即 i 为偶数),$a(i)=a(i/2)$

i%2=1(即 i 为奇数),$a(i)=a((i+1)/2)+a((i-1)/2)$。

递推描述：

```
输入 n;
a[1]=1;
for(i=2;i<=n;i++)
    if(i%2==0)
        a[i]=a[i/2];
    else
        a[i]=a[(i+1)/2]+a[(i-1)/2];
输出 a[n];
```

5．把整数 1,2,…,n^2 从外层至中心按顺时针方向螺旋排列所成的 $n\times n$ 方阵，称顺转 n 阶方阵；按逆时针方向螺旋排列所成的称逆转 n 阶方阵。

设计程序选择分别打印这二种旋转方阵。

算法设计。

打印二种旋转方阵关键在于数组元素的赋值以及赋值与打印的巧妙结合。

对应方阵的 n 行 n 列设置二维数组 $a(n,n)$。令 m=int(n/2)，当 n 为偶数时，方阵共 m 圈。当 n 为奇数时，方阵除 m 圈外正中间还有一个数 $a(m+1,m+1)=n\times n$。

对于 m 圈，每圈有上下左右 4 条边。最外圈定义为第 1 圈，从外往内依次定义为第 2 圈，……，第 m 圈，第 i 圈每边有 $n-2i+1$ 个数。

为了实现旋转准确对各圈各边的每一个数组元素赋值,我们引入中间变量 s、t：

```
s=n−2i+1
t=t+4s   (t 置初值 0)
```

设置 i(1～m)循环对第 i 圈操作，设置 j(i～$n-i$)循环对第 i 圈的 4 条边的 $n-2i+1$ 个元素操作。i,j 二重循环可对方阵的每一元素赋值。

逆时针转方阵递推赋值描述:

```
for(i=1;i<=m;i++)                    /*   按规律给 a 数组赋值 */
  {s=n+1−2*i;
   for(j=i;j<=n−i;j++)
     {a[i][j]=t+1−i+j;      /* 上行赋值，其中+j 体现往右递元素值增 1 */
      a[j][n+1−i]=a[i][j]+s;   /* 右列为即在 a(i,j)的基础上增 s */
      a[n+1−i][j+1]=a[i][j]+3*s−1+2*i−2*j;   /* 下行赋值 */
      a[j+1][i]=a[n+1−i][j+1]+s;   /*左列即在 a(n+1−i,j+1)基础上增 s*/
      }
   t=t+4*s;
  }
```

在顺时针转方阵中,还是上述赋值,只是打印输出时把行列互换。

6．一场球赛开始前，售票工作正在紧张的进行中，每张球票为 50 元。现有 2n 个人排队等待购票，其中有 n 个人手持 50 元的钞票，另外 n 个人手持 100 元的钞票，假设开始售票时售票处没有零钱。问这 2n 个人有多少种排队方式，使售票处不至出现找不开钱的局面？

算法设计。

这是一道典型的组合计数问题，考虑用递推求解。

令 $f(m,n)$表示有 m 个人手持 50 元的钞票，n 个人手持 100 元的钞票时共有的方案总数。我们分情况来讨论这个问题。

（1）n=0

n=0 意味着排队购票的所有人手中拿的都是 50 元的钱币，那么这 m 个人的排队总数为 1，即 $f(m,0)=1$

（2）$m<n$

若排队购票的 $m+n$ 个人中有 m 个人手持 50 元的钞票，n 个人手持 100 元的钞票，当 $m<n$ 时，即使把 m 张 50 元的钞票都找出去，仍会出现找不开钱的局面，所以这时排队总数为 0，即 $f(m,n)=0$

（3）其他情况

我们思考 $m+n$ 个人排队购票的情景，第 $m+n$ 个人站在第 $m+n-1$ 个人的后面，则第 $m+n$

个人的排队方式可由下列两种情况获得。

① 第 $m+n$ 个人手持 100 元的钞票，则在他之前的 $m+n-1$ 个人中有 m 个人手持 50 元的钞票，有 $n-1$ 个人手持 100 元的钞票，此种情况共有 $f(m,n-1)$。

② 第 $m+n$ 个人手持 50 元的钞票，则在他之前的 $m+n-1$ 个人中有 $m-1$ 个人手持 50 元的钞票，有 n 个人手持 100 元的钞票，此种情况共有 $f(m-1,n)$。

由加法原理得到 $f(m,n)$的递推关系：

$f(m,n)=f(m,n-1)+f(m-1,n)$

初始条件：

当 $m<n$ 时，$f(m,n)=0$

当 $n=0$ 时，$f(m,n)=1$

算法描述：

```
scanf("%d,%d",&m,&n);
 for(j=1;j<=m;j++)                          /* 确定初始条件 */
      f[j][0]=1;
 for(j=0;j<=m;j++)
 for(i=j+1;i<=n;i++)
      f[j][i]=0;
 for(i=1;i<=n;i++)
 for(j=i;j<=m;j++)
      f[j][i]=f[j-1][i]+f[j][i-1];          /* 实施递推 */
printf("   f(%d,%d)=%ld.\n",m,n,f[m][n]);    /* 输出结果 */
```

习题 5

1．一个数列由 n 个正整数组成。对该数列进行一次操作：去除其中两项 a、b，添加一项 $a\times b+1$。经 $n-1$ 次操作后该数列剩一个数 a，试求 a 的最大值。

算法设计。

设数列为 3 项 x、y、$z(x<y<z)$，由

$(x\times y+1)\times z>(x\times z+1)\times y>(y\times z+1)\times x$

知操作时去掉的 a、b 两项选数列中的最小的两项，可使积最大。

我们采用贪心算法，当数列中有 3 项以上时，每次操作选择去掉最小的 2 项。

算法描述：

```
void main()
{int k,i,j,n;
 long h,x,y,z,a[200];
 printf("请输入数列中整数的个数：");
 scanf("%d",&n);
 for(k=1;k<=n;k++)                          /*  逐个输入数列中的各个整数 */
   {printf("请输入 a(%d): ",k);
```

```
    scanf("%ld",&a[k]);
    }
  printf(" %d 个原始数据为:   ",n);
  for(k=1;k<=n;k++)                          /*   显示原始数据 */
    printf("%ld   ",a[k]);
  for(k=1;k<=n-1;k++)                        /*   操作 n-1 次 */
   {for(i=k;i<=n-1;i++)                      /*   对 n-k+1 项从小到大排序 */
    for(j=i+1;j<=n;j++)
      if(a[i]>a[j])
        {h=a[i];a[i]=a[j];a[j]=h;}
    if(k==1)
      {printf("\n   原始数据排序:   ");
       for(j=1;j<=n-1;j++)                   /*   显示原始数据排序结果 */
         printf("%ld   ",a[j]);
       printf("%ld   ",a[n]);
       }
    x=a[k];y=a[k+1];a[k+1]=x*y+1;            /*   实施操作 */
    z=a[k+1];
    printf("\n 第%d 次操作后为:",k);         /*   输出操作结果 */
    for(i=k+1;i<=n-1;i++)                    /*   对 n-k+1 项从小到大排序 */
    for(j=i+1;j<=n;j++)
      if(a[i]>a[j])
        {h=a[i];a[i]=a[j];a[j]=h;}
    for(j=k+1;j<=n;j++)
      {printf("   %ld",a[j]);
       if(a[j]==z)                           /* 注明操作数 */
         printf("(%ld*%ld+1)",x,y);
      }
    }
}
```

2．随机产生一个 n 位高精度正整数，去掉其中 $k(k<n)$个数字后按原左右次序将组成一个新的正整数。编程，对给定的 n、k 寻找一种方案,使得剩下的数字组成的新数最小。

算法设计。

操作对象是一个可以超过有效数字位数的 n 位高精度数，存储在数组 a 中。

每次删除一个数字，选择一个使剩下的数最小的数字作为删除对象。之所以选择这样“贪心”的操作，是因为删 k 个数字的全局最优解包含了删一个数字的子问题的最优解。

当 k=1 时，在 n 中删除哪一个数字能达到最小的目的？从左到右每相邻的两个数字比较：若出现减，即左边大于右边，则删除左边的大数字；若不出现减, 即所有数字全部升序，则删除最右边的大数字。

当 $k>1$（当然小于 n 的位数），按上述操作一个一个删除。删除一个达到最小后，再从头即从串首开始，删除第 2 个，依此分解为 k 次完成。

若删除不到 k 个后已无左边大于右边的减序，则停止删除操作，打印剩下串的左边 $n-k$ 个数字即可（相当于删除了若干个最右边的大数字）。

算法描述：

```
void main()
{ int i,j,k,m,n,t,x,a[2000];
  t=time()%1000;srand(t);                    /*  随机数发生器初始化 */
  printf(" input n(n<2000):"); scanf("%d",&n);
  printf("删除数字个数 k:");scanf("%d",&k);
  printf("%d 位整数为:",n);
  for(i=1;i<=n;i++)
    {a[i]=rand()%10;                    /* 产生并输出 n 个数组成的序列 */
     if(a[1]==0)
         continue;
     printf("%d",a[i]);
     }
  printf("\n 在以上整数中删除%d 个数字:",k);
 i=0;m=0;x=0;
 while(k>x && m==0)
   {i=i+1;
   if(a[i]>a[i+1])                /* 出现递减,删除递减的首数字 */
       {printf("%d ",a[i]);
        for(j=i;j<=n-x-1;j++)
            a[j]=a[j+1];
        x=x+1;                          /* x 统计删除数字的个数 */
        i=0;                            /* 从头开始查递减区间 */
       }
   if(i==n-x-1)                        /* 已无递减区间,m=1 脱离循环 */
       m=1;
   }
 printf("\n 删除后以下数最小:");
 for(i=1;i<=n-k;i++)               /* 只打印剩下的左边 n-k 个数字 */
   printf("%d",a[i]);
 }
```

4．一个有向图 G，它的每条边都有一个非负的权值 c[i,j]，“路径长度”就是所经过的所有边的权值之和。对于源点需要找出从源点出发到达其他所有结点的最短路径。

算法设计。

E.Dijkstra 发明的贪婪算法可以解决最短路径问题。算法的主要思想是：分步求出最短路

径，每一步产生一个到达新目的顶点的最短路径。下一步所能达到的目的顶点通过如下贪婪准则选取：在未产生最短路径的顶点中，选择路径最短的目的顶点。

设置顶点集合 S 并不断作贪心选择来扩充这个集合。当且仅当顶点到该顶点的最短路径已知时该顶点属于集合 S。初始时 S 中只含源。

设 u 为 G 中一顶点，我们把从源点到 u 且中间仅经过集合 S 中的顶点的路称为从源到 u 特殊路径，并把这个特殊路径记录下来（例如程序中的 dist[i,j]）。

每次从 $V-S$ 选出具有最短特殊路径长度的顶点 u，将 u 添加到 S 中，同时对特殊路径长度进行必要的修改。一旦 $V=S$，就得到从源到其他所有顶点的最短路径，也就得到问题的解。

习题 6

2．用 m 种零币 $t_1, t_2, \cdots, t_m$（单位为分，约定 $t_1<t_2<\cdots<t_m$）来找 n 分钱整币，试求最少的零币数。

（1）建立递推关系

设 $g(k,j)$为用硬币 $t_1, t_2, \cdots, t_k$ $(1\leqslant k\leqslant m)$找 j 分钱的最少硬币数。对 t_k 面临以下两种选择：不找 t_k，其硬币数为 $g(k-1,j)$；找一个 t_k，其硬币数为 $g(k,j-t(k))+1$。比较这两种情形，选其较小者，得递推关系式：

$g(k,j)=min\{g(k-1,j),g(k,j-t(k))+1\}$　　$(j\geqslant t(k),k=2,3,\cdots,m)$

边界条件：

$g(1,j)=j/t(1)$　　$(j\%t(1)=0)$

$g(1,j)=0$　　$(j\%t(1)>0)$

（2）计算最优值

```
for(i=0;i<=n;i++)                                   /* 确定初始条件 */
        if(i%t[1]==0) g[1][i]=i/t[1];
      else g[1][i]=0;
  for(k=2;k<=m;k++)
for(j=0;j<=n;j++)                                   /* 递推计算 g(k,j) */
        if(j>=t[k] && g[k][j-t[k]]+1<g[k-1][j])
            g[k][j]=g[k][j-t[k]]+1;
else
      g[k][j]=g[k-1][j];
输出最优值 g[m][n];
```

3．随机产生一个 n 行 m 列的点数值矩阵，寻找从矩阵左上角至右下角，每步可向下（D）或向右（R）的一条数值和最大的路径。

算法设计。

采用动态规划，即从右下角逐行反推至左上角.确定 n、m 后，随机产生的整数二维数组 $a(n,m)$作矩阵输出，同时赋给部分和数组 $b(n,m)$。

注意到最后一行与一列各数只有一个出口，于是由 $b(n,m)$ 开始向左逐个推出同行的

$b(n,j),(j=m-1,\cdots,2,1)$；向上逐个推出同列的 $b(i,m),(i=n-1,\cdots,2,1)$。

$b(i,j)$与 $c(i,j)(i=n-1,\cdots,2,1,j=m-1,\cdots,2,1))$的值由同一列它下面的整数 $b(i+1,j)$与同一行其右边的整数 $b(i,j+1)$行的大小反推决定：

若 $b(i,j+1)>b(i+1,j)$，则

$$b(i,j)=b(i,j)+b(i,j+1),\ c(i,j)=\text{“}R\text{”}\ (\text{表向右})$$

否则　$b(i,j)=b(i,j)+b(i+1,j),\ c(i,j)=\text{“}D\text{”}\ (\text{表向下})$

这样所得 $b(1,1)$即为所求的最大路径数字和。

为了打印最大路径，利用 c 数组从上而下查找：先打印 $a(1,1)$，若 $c(1,1)$=R，则下一个打印 $a(1,2)$，否则打印 $a(2,1)$。一般地，打印路径中的 $a(i,j)$后：

若　c(i,j)=R 则 j 增 1,即 j=j+1,然后打印“- R-”; $a(i,j)$;

若　c(i,j)=D 则 i 增 1,即 i=i+1,然后打印“- D-”; $a(i,j)$;

依次类推，直至打印到终点 $a(n,m)$。

计算最优值与构造最优解描述：

```
printf("寻找一条数字和最大的路径.\n ");
  for(j=m-1;j>=1;j--)
{b[n][j]+=b[n][j+1];c[n][j]='R';}
  for(i=n-1;i>=1;i--)
{b[i][m]+=b[i+1][m];c[i][m]='D';}
  for(i=n-1;i>=1;i--)
     for(j=m-1;j>=1;j--)
 if(b[i][j+1]>b[i+1][j])
    {b[i][j]+=b[i][j+1];c[i][j]='R';}
 else
    {b[i][j]+=b[i+1][j];c[i][j]='D';}
printf("最大路径和为:%d\n",b[1][1]);
  printf("最大路径为:%d",a[1][1]);i=1;j=1;
  while(i<n || j<m)
      if(c[i][j]=='R')
                {j++;printf("-R-%d",a[i][j]);}
      else
                {i++;printf("-D-%d",a[i][j]);}
```

4．已知边数值三角形每两点间的距离，每一个点有向左或向右两条边，求边数值三角形顶点 A 到底边的最短路径。

算法设计。

设边数值三角形为 n 行（不包含作为边终止点的三角形底边），每点为(i,j)，$i=1,2,\cdots,n$；$j=1,2,\cdots,i$。从点(i,j)向左的边长记为 $l(i,j)$，点(i,j)向右的边长记为 $r(i,j)$。

设置 $a(i,j)$为点(i,j)到底边的最短路程。显然

$$a(i,j)=min(a(i+1,j)+l(i,j),a(i+1,j+1)+r(i,j))$$

$$st(i,j)=\{\text{'l'},\ \text{'r'}\}$$

应用逆推求解，所求的顶点 *A* 到底边的最短路程为 *a*(1,1)。

计算最短路程与构造最短路径描述：

```
for(i=n;i>=1;i—)              /* 逆推求取最短路程 */
{for(j=1;j<=i;j++)
    if(a[i+1][j]+l[i][j]<a[i+1][j+1]+r[i][j])
       {a[i][j]=a[i+1][j]+l[i][j];st[i][j]='l';}
    else
       {a[i][j]=a[i+1][j+1]+r[i][j];st[i][j]='r';}
   }
printf("\n 最短路程为：%d",a[1][1]);
printf("\n 最短路径为：顶点 A ");
for(j=1,i=1;i<=n;i++)
   if(st[i][j]=='l')
        printf("L−%d−",l[i][j]);
   else
        { printf("R−%d−",r[i][j]);j++;}
```

5. 插入加号使和最小

在一个由 *n* 个数字组成的数字串中插入 *r* 个加号(1≤*r*<*n*<10)，将它分成 *r*+1 整数，找出一种加号插入方法，使得这 *r*+1 个整数的和最小。

例如，如何在整数 468 214 962 中插入 5 个加号，使所成的 6 个整数的和最小？

按插入的加号数来划分阶段，若插入 *k* 个加号，可把问题看做是 *k* 个阶段的决策问题。

（1）建立递推关系

设 *g*[*i*,*k*]表示在前 *i* 位数中插入 *k* 个加号所得的最小值，*a*[*i*,*j*]表示从第 *i* 位到第 *j* 位所组成的自然数。

用 *g*[*i*,*k*]存储阶段 *k* 的每一个状态，可以得递推关系式：

g[*i*,*k*]=*min*{*g*[*j*,*k*−1]**a*[*j*+1,*i*]}（*k*<=*j*<*i*）

边界值：*g*[*j*,0]=*a*[1,*j*] (1<*j*<=*i*)

所求的最优值为 *g*[*n*,*r*]。因为是求最小值, 边界以外的数组元素 *g*[*j*][*k*]应赋一个比较大的初值。

（2）构造最优解

其他标注位置的数组 *t*[*k*]与 *c*[*i*,*k*]同例 6.3 所述。

为了检验所求结果是否可靠，程序设计计算各段整数之和 *y*，如果 *y*=*g*[*n*,*r*]，则通过检验打印输出结果。

递推求最优值注意：必须给 *g*[*j*,*k*]赋数值大的初始值，例如

```
for(k=1;k<=r;k++)
for(j=1;j<=n;j++)
    g[j][k]=10000;                    /* 给 g[j,k]赋初始值 */
```

6．给定 n 个整数（可能为负整数）组成的序列 $a_1, a_2, \cdots, a_n$，求该序列形如 $\sum_{k=i}^{j} a_k$ 的字段和的最大值。

动态规划设计。

记 b[j]=max$\left(\sum_{k=i}^{j} a_k\right)$,1≤i≤j≤n,知

当 b[j－1]>0 时，b[j]=b[j－1]+a[j];

当 b[j－1]≤0 时，b[j]=a[j];

因而得递推关系：

b[j]=max(b[j－1]+a[j],a[j]), 1≤j≤n.

递推求最优值：

```
sum=0;b=0;
    for(j=1;j<=n;j++)
       { if(b>0) b+=a[j];
         else b=a[j];
         if(b>max)    max=b;
       }
        printf(max);
```

习题 7

1．从 1 开始按正整数的顺序不间断连续写下去所成的整数称为连写数。

要使连写数 123456789101112…m（连写到整数 m）能被指定的整数 p（<1 000）整除，m 至少为多大?

算法设计：

要使连写数 1234…m 能被键盘指定的整数 p 整除，模拟除法操作：设被除数为 a，除数为 p，商为 b=a/p，余数为 c=a%n。

当 c-0 且 m 为 1 位数时，a=c×10+m 作为下一轮的被除数继续. 若 m 为 2 位数时，a=c×100+m 作为下一轮的被除数继续。一般地 m 为一个 i 位数时，则计算 t=10^i,a=c×t+m 作为下一轮的被除数继续。

直至 c=0 时，所得连写数能被指定的 p 整除，结束。在整个模拟除法过程中，m 从 1 开始按顺序递增 1。

除运算模拟描述：

```
scanf("%d",&p);
 c=1;m=1;k=1;
    while(c!=0)
      { m++;n=m;t=1;i=0;
        while(n!=0)                    /* 判断 m 的位数 i  */
          {n=n/10;t*=10;i++;}          /* 计算 t=10^i */
```

```
            a=c*t+m;c=a%p;k+=i;              /* 实施除运算 */
    }
        printf("m=%d, k=%d \n",m,k);
```

2．应用蒙特卡洛法计算圆周率π。

在 a=0,b=2,d=2 的正方形 $ABCD$ 中，曲线取第一象限中的圆 $x^2+y^2=4$，显然曲边梯形为圆面积的 1/4，即 $s=\pi$。

应用蒙特卡洛法进行模拟计算：

```
输入正整数 n;
m=0;d=2;b=d;
for(k=1;k<=n;k++)
    {x=b*rand();          /* rand()为（0,1）区间内的随机数 */
     y=d*rand();
     if(y<sqrt(4-x*x)    m=m+1;
     }
s=m*b*d/n;
输出 s(即为π);
```

3．模拟发桥牌。桥牌共 52 张，无大小王。按 E、S、W、N 顺序把随机产生的 52 张牌分发给各方，每方 13 张。发完后，分类从大到小整理每方的牌。

模拟算法设计。

随机产生扑克的各色各点同发升级牌，只是少了对大小王的处理。但增加了发牌位置 E、S、W、N 的确定。

为了各方排序方便，把 y 的取值变为 2～14，其中 14 最大代表 A。在桥牌的分类整理程序段，把每家所有牌分花色应用排序实现点从大到小（即 A,K,Q,J,10,9,…,2）的顺序排列。

模拟算法描述：

```
for(i=1;i<=52;i++)                    /*  随机产生 52 张不同的扑克牌 */
    {for(j=1;j<=10000;j++)
        {x=rand()%4+1; y=rand()%13+2;
         z=x*100+y; t=0;
         for(k=0;k<=i-1;k++)
            if(z==m[k]) {t=1;break;}
         if(t==0)
            {m[i]=z;break;}           /* 确保牌不重复 */
         }
    }
 for(k=1;k<=13;k++)                   /*  依次把 52 张扑克牌分发到四家 */
    {e[k]=m[4*k-3]; s[k]=m[4*k-2];
     w[k]=m[4*k-1]; n[k]=m[4*k];}
```

以下排序整理略。

4．模拟巴什游戏：一堆石子有 n 粒，参与巴什（Bash）游戏的二人轮流从这堆石子中取

石子，规定每次至少取一粒，最多取 m 粒（正整数 m、n 随机产生，$m<n$）。最后取完石子者得胜。

算法设计。

如果 $n=m+1$，那么由于一次最多只能取 m 个，所以，无论先取者拿走多少个，后取者都能够一次拿走剩余的石子，后者取胜。

由此我们发现了取胜的法则：如果 $n=(m+1)r+s$，（r 为任意自然数，$s\leqslant m$），那么先取者要拿走 s 个石子，如果后取者拿走 k（$\leqslant m$）个，那么先取者再拿走 $m+1-k$ 个，结果剩下 $m+1$ 的整数倍个。以后保持这样给对手留下 $m+1$ 整数倍数的取法，那么先取者肯定获胜。

为书写简便，记原始两堆石子数用{}括起来，计算机取后两堆石子数用[]括起来，操作者取后两堆石子数用( )括起来。

如果开始时的石子数不是 $m+1$ 整数倍数，先取者可稳操胜券。如果开始时的石子数就是 $m+1$ 整数倍数，后取者可稳操胜券。

对于已知的 n、m，计算机取子算法描述：

```
printf("计算机取石子!");
        s=n%(m+1);
        if(s==0)
          n2=m-1-rand()%(m-1);          /* 计算机随机取作应付 */
      else
          n2=s;            /* 计算机取于胜势，取 s 后留下 m+1 的整数倍 */
        printf("计算机取:%d ",n2);
        n=n-n2;
        printf("->[%d]",n);
        if(n==0) {printf("全取完,计算机胜!\n");break;}
```

习题 8

略

习题 9

5．把 for (i = 0 ;i≤log N －1; i ++)

改为：

for (i = 0 ;i≤$\lceil \log N-1 \rceil$; i ++)

附录 2　C 常用库函数

1．输入输出函数：stdio.h

函数名称	功　能	用　法
scanf	用于格式化输入	int scanf(char *format[,argument,…])
printf	产生格式化输出的函数	int printf(char *format…)
putch	输出字符到控制台	int putch(int ch)
putchar	在 stdout 上输出字符	int putchar(int ch)
getch	从控制台无回显地取一个字符	int getch(void)
getchar	从 stdin 流中读字符	int getchar(void)
fclose	关闭 fp 所指向的文件，释放文件缓冲区	int fclose（fp）
feof	检查文件是否结束	int feof（fp）
fgetc	从 fp 指向的文件中取一个字符	int fget（fp）
fgets	从 fp 指向的文件中取一个长度为 $n-1$ 的字符串，存入到以 buf 为起始地址的存储区中	char *fgets（buf,n,fp）
fopen	以 mode 指定的方式打开以 filename 为文件名的文件	file *open（filename,mode）
fprintf	将 args 的值以 format 指定的格式输出到 fp 所指向的文件中	int fprintf（fp,format,arg,…）
fputc	将字符 ch 输出到 fp 指向的文件中去	int fputc（ch,fp）
fputs	将 str 指向的字符串输出到 fp 所指向的文件	int fputs（str,fp）
fread	从 fp 指向的文件中读长度为 size 的 n 个数据项，存放在 fp 所指向的存储区中	int fread（pt,size,n,fp）
fscanf	从 fp 指向的文件中按 format 规定的格式将输入的数据存入到 args 所指向的内存中去	int fscanf（fp,format,args,…）
fseek	将 fp 所指向的文件的位置指针移到以 base 所指向的位置为基准，以 offset 为位移量的位置	int fseek（fp,offset,base）
getw	从 fp 指向的文件中读取下一个字	int getw（fp）
putw	将一个字输出到 fp 所指向的文件中去	int putw（w,fp）
open	以 mode 设定的打开已经保存的名为 filemane 的文件	int open（filemane,mode）
rewind	将 fp 指向的文件的位置指针设置为文件开头位置，并清除文件结束标志和错误标志	void rewind（fp）
write	从 buf 指向的缓冲区输出 count 个字符到 fp 所指向的文件中	int write（fd,buf,count）

2．数学函数：math.h

函数名称	功　　能	用　　法
sin	计算 sin(*x*)的值	double sin（double x）
cos	计算 cos(*x*)的值	double sin（double x）
exp	计算 *e* 的 *x* 次方	double exp（double x）
fabs	计算 *x* 的绝对值	double abs（double x）
fmod	计算出整数 *x*/*y* 的余数	double fmod（x,y）
ceil	计算不小于 *x* 的最小整数	double ceil（x）
floor	计算不大于 *x* 的最大整数	double floor（x）
pow	计算出 *x* 的 *y* 次方	double pow（x,y）
sqrt	计算出 *x* 的平方根	double sqrt（x）
tan	计算出 tan(*x*)的值	double tan（double x）
srand	初始化随机数发生器	Void srand（unsigned seed）
rand	产生并返回一个随机数	int rand()

3．字符函数：ctype.h

函数名称	功　　能	用　　法
isapha	检查 ch 是否为字母	int isapha（ch）
isdigit	检查 ch 是否是数字（0～9）	int isdigit（ch）
islower	检查 ch 为小写字母，返回 1；否则返回 0	int islower（ch）
isupper	检查 ch 为大写字母，返回 1；否则返回 0	int isupper（ch）
tolower	将 ch 字符转换为小写字符	int tolower（ch）
toupper	将 ch 字符转换为大写字符	int toupper（ch）

4．字符串函数：string.h

函数名称	功　　能	用　　法
strcat	将字符串 str2 接到 str1 后面，str1 字符串后面的“\0”自动取消	char *strcat(str1,str2)
strcmp	比较两个字符串，str1<str2,返回值为负数 str1=str2,返回值为 0 str1<str2，返回值为正数	int strcmp（str1,str2）
strcpy	将 str2 指向的字符串复制到 str1 中去	char *strcpy(str1,str2)
strlen	计算字符串 str 的长度（不包含“\0”），返回值为字符的个数	unsigned int strlen（str）

5．图形函数：graphics.h

函数名称	功　　能	用　　法
arc	以 *r* 为半径，*x*、*y* 为圆心，*s* 为起点，*e* 为终点画一条圆弧	void arc(x,y,s,e,r)
Bar	以 1，t 为左上角坐标，r、b 为右下角坐标画一个矩形框	void bar(1,t,r,b)
Circle	以 *x*、*y* 为圆心，*r* 为半径画一个圆	void cirle（x,y,r）

续表

函数名称	功　　能	用　　法
Closegraph	关闭图形工作方式	void closegraph（void）
Floodfill	对一个有界区域着色	void floodfill（x,y,border）
Getbkcolor	返回当前背景颜色	int far getbkcolor（）
Getcolor	返回当前画线颜色	int getcolor（）
Gotoxy	将字符屏幕的光标移动到 *x*、*y* 处	void gotoxy（x,y）
Initgraph	按 drive 指定的图形驱动器装入内存，屏幕显示模式由 mode 指定，图形显示器路径由 path 定	void initgraph(drive,mode,path)
Line	从（*sx*,*sy*）到（*ex*,*ey*）画一条直线	void line(sx,sy,ex,ey)
Outtext	在光标所在位置上输出一个字符串	void outtext（str）
Rectangle	使用当前的画线颜色从（left,top）为左上角，（right，bottom）为右下角画一个矩形	void rectangle（left，top，right，bottom）
Setbkcolor	重新设定背景颜色	void setbkcolor（color）
Setcolor	设置当前画线的颜色	void setcolor（color）
setfillstyle	设置图形的填充式样和填充颜色	void far setfillstyle（pa，color）
settextstyle	设置图形字符输出字体，方向和字符大小	void far settextstyle（font，direct，size）

6．字符屏幕处理函数：conio.h

函数名称	功　　能	用　　法
Clrscr	清除整个屏幕，将光标定位到左上角处	void clrscr（）
Gotoxy	将字符屏幕的光标移动到 *x*、*y* 处	void gotoxy（x,y）
textbackgroud	设置字符屏幕的背景	void textbackgroud(color)
Textcolor	设置字符屏幕下的字符颜色	void textcolor（color）
Window	建立字符窗口	void window（left，top，right，bottom）

参 考 文 献

1　王晓东．算法设计与分析．北京：清华大学出版社，2003

2　王晓东．算法设计与分析习题解答．北京：清华大学出版社，2006

3　徐士良．计算机常用算法(第二版)．北京：清华大学出版社，2002

4　陈慧南．算法设计与分析——C++语言描述．北京：电子工业出版社，2006

5　余祥宣，崔国华，邹海明.计算机算法基础．武汉：华中科技大学出版社，2006

6　Mark Allen Weiss.陈越改编．数据结构与算法分析——C 语言描述（英文版第二版）．人民邮电出版社，2005

7　谭浩强．C 程序设计(第三版)．北京：清华大学出版社，2005

8　王岳斌等．C 程序设计案例教程．北京：清华大学出版社，2006

9　杨克昌．计算机程序设计经典题解．北京：清华大学出版社，2007

10　杨克昌等．C 语言程序设计．武汉：武汉大学出版社，2007

11　[美]Jeri Hanly　Elliot Koffman 著. C 语言详解(第 5 版).北京：人民邮电出版社，2007

12　[美]K.N.King 著．C 语言程序设计：现代方法．北京：人民邮电出版社，2007

13　[美]Ira Pohl 著．C++教程．北京：人民邮电出版社，2007

14　郑山红等．C 语言程序设计．北京：人民邮电出版社，2007

15　Jon Kleinberg and Eva Tardos．算法设计．北京：清华大学出版社，2005

16　Herbert Schildt，C：The Complete Reference，McGraw－Hill．C 语言大全．北京：电子工业出版社，1990

17　B.W.Kernighan and P. J. Plauger， The Elements of Programming Style．中译本：程序设计技巧．晏晓焰编译．北京：清华大学出版社，1985

18　陈国良编著．并行算法的设计与分析．北京：高等教育出版社，2002

19　Quinn，M.j.著．MPI 与 OpenMP 并行程序设计．陈文光，武永卫等译．北京：清华大学出版社，2004

20　Barry Wilkinson，Michael Allen 著．并行程序设计．陆鑫达等译．北京：机械工业出版社，2005

21　黄铠，徐志伟著．可扩展并行计算技术、结构与编程．陆鑫达等译．北京：机械工业出版社，2000